BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M19

# "A Multiple-Process Implementation of Threads"

by

Tsung-Jen Chou

# A Multiple-Process Implementation of Threads

Tsung-Jen Chou
B.S., National Taiwan University, 1989
Sc. M., Brown University, 1993

May 1993

# Abstract

Based on the *Brown Threads* package which runs on a single process, I built a thread package running on multiple processes. In order to allow threads to migrate among different processes, the address space should look the same in all the processes from the thread's point of view and each process's file-descriptor table should be consistent with those of the other processes so the threads can run on any one of the processes.


# 1 Introduction

## 1.1 An Ideal User-Level Thread Package

A thread is a sequential flow of control. In an ideal thread package, each thread runs sequentially, and has its own stack and program counter to keep track of its state. All threads have the same address space, share the same opened files, child processes and signals.

An ideal user-level thread package should support these features fully and efficiently. The ideal thread package should use as few system resources for itself as possible so that user programs can utilize it and perform the system operations with minimal overhead.

The package should be user-friendly and flexible. The thread package should support extensions of system functionality, so that the user can customize it to fit his or her needs, e.g. adding various scheduling mechanisms.

## 1.2 Real Thread Packages

In traditional user-level thread packages, multiple threads run on a single process. By this strategy they make sure that all the threads share the same address space, open files, child processes and signals. An obvious drawback with this scheme is with system I/O operations. Some system I/O operations will block a whole process by default, e.g. reading from the terminal.

So a whole process can be blocked by one thread, which makes the remaining threads of the process unrunnable in spite of the fact that they themselves might be in a runnable state. The traditional way to deal with this is to use asynchronous I/O: a thread invoking a would-be-blocking system calls, such as *read*ing from a terminal, is put onto a queue. This thread is pulled off from the queue and put back into running when the process gets a *SIGIO* signal and finds out the the I/O operation which the thread is waiting for is completed. This method requires a great deal of overhead.

In contrast with Solaris Threads which are built on multiple lightweight processes (within a normal process), we approximate LWPs with overlapping normal processes.

In our thread package, multiple threads run on multiple processes, thus even if a process blocks, the other threads still can run on different processes. In order to make threads have the same address space, processes must share part of the data section[1]. All the processes should share a common heap as well. Processes also should share the effects of all I/O operations: a file opened by one should become open in the other processes, with the same file descriptor, file offset location, etc.

When implementing this thread package, we should keep in mind that we may treat each process as a virtual processor. We can port this thread package from a single-processor machine to a multi-processor machine without modification. This portion of the project has not been tested.

Each thread has a chunk of memory associated with it which not only stores the specific control data about the thread but also contains the thread's stack.

# 2 Overall Strategy

All the processes in the thread package appear almost the same, so the threads can migrate among these processes and data accessed from one pro-

[1] We roughly divide a process into the text section, data section, heap, and stack.

cess will be consistent with data accessed from any other process. From a thread's point of view, there is a single address space, regardless of on which process the thread is running. The thread package makes all the file-descriptor tables consistent from the thread package's view in spite of the fact that each process has a separate, kernel-implemented file-descriptor table; thus, from a thread's point of view, there is a single space of open files.

Each thread is associated with a chunk of memory of a size specified by the user. This chunk of memory, residing in shared memory accessible to all the processes in the thread package, serves two purposes. The lower part, called the *thread control block*, stores specific information about a thread, such as the priority of the thread, whether the thread is locked or not, and etc. The rest of this chunk of memory serves as a run-time stack. Switching between different user threads means changing the stack pointer from pointing to the current stack of one thread to pointing to the current stack of the other. Because this chunk of memory is accessible to all the processes, the user thread can run on any one of the processes.

A process in our thread package can be divided into two portions: one portion is loaded by the ordinary OS loader, which we call the *loader portion*; the other portion is incrementally loaded by the loader portion. It is called the *application portion.*

Because there is more than one process in this thread package, we may need some data which is specific to each process, for instance, which thread is running on the process. When we implemented this package, we had to decide whether we really wanted to share the entire data section among all the processes or reserve some data in a region which is only accessible to a particular process.

At first glance, sharing the entire data section seems a reasonable and natural way of resolving the above question because all the threads should share the same address space. We can have an array keeping track of such specific data as which thread is running on the process. If we can find the appropriate index to the array, we can retrieve or modify this data. There are two ways to find this index. We can store the process id in the array and use the value returned by *getpid* as a key to find the appropriate in-

dex in this array. The problem with this method is the overhead involved with invoking *getpid* system calls. This overhead may be quite severe because we need to know this index to find out which thread is running on the process.

Another way to find the index is use the stack pointer. Each thread has a space reserved exclusively for its stack. We can find out which thread is running by determining which thread's stack the stack register is pointing to. Even though we already find out the address the current running thread's control block, we still sometime need to know some information about this process, such as if there are any singals pending on this process. As we access the running thread's control block, we can access the index if we save it in the thread control block. This strategy does not invoke the *getpid* system call or other system calls. If all the threads have the same size stack, we may find the current running thread by a simple mathematical equation:

p0 : the first tcb's starting address.

p1 : the second tcb's starting address.

sp : the address of stack pointer.

$index = (sp - p0)/(p1 - p0)$.

But the threads don't necessarily have the same size of stack, since the users specify the size of their threads' stacks. We can keep the lower and upper bounds of the stacks for all of the threads in a table, and we can search through this table to find out which thread's stack the stack pointer is at. The problem with this method is that whenever we create or delete threads, this table has to be modified. Also, searching through this table is expensive. Through good implementation we can reduce the amount of time for the search, but since we need to execute the search in almost every function, the time is still too expensive.

An alternative is to store such specific data at a region only accessible by a particular process. What is important is that the location containing this process-specific data is at the same address in each process. This is very efficient because the process can access this data without any overhead. The downside is that when we want to port this thread package to a platform which already has a supported thread mechanism, it could cause problems.

We adopted a compromise. We keep all such data in an array located in shared memory, but we also keep an index to the array in the region only accessible from a particular process. Thus, we eliminate the overhead of calculating the index and reduce the potential difficulties with porting.

The loader portion resides in the ordinary non-shared memory segment. Every process in the thread package has its own loader portion, so changes to data in this portion won't be seen by others. We put the data exclusively reserved for each process in this portion. This portion comprises all the elements of a Unix process.

The application portion resides in the shared memory segment. All the processes in the package share this portion, so we put the the data which should be shared and the application code in this portion.

We also need to take care of the heap which is normally allocated exclusively for each process and utilized by dynamic memory operations such as *malloc* and *free*. Because each process's heap space is protected from other processes, the data residing in the normal heap cannot be shared. We preallocate a shared memory segment which serves as a common heap among these processes.

## 2.1 The Loader Portion

Three goals must be accomplished in this portion.

- Define some private data specific to each process.
- Request shared memory and attach it to the address space.
- Load the application portion into the shared memory and run it.

In an ordinary Unix process, only the text section is shared among different processes. Each process has its own data section, heap, and stack which no other process can access. All the global variables defined in the data section can only be accessed from this particular process, so we define the process's private variables here.

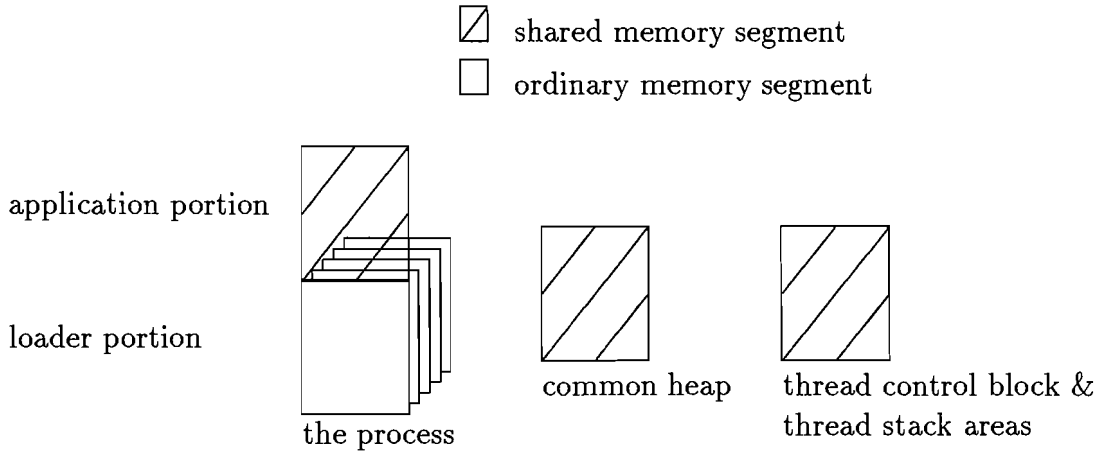


Figure 1: A diagram of the processes and shared memory.

If part of a process address space is in a shared memory segment, this part will be shared among this process and its child processes. We request a shared memory segment big enough and incrementally load the object file containing the data which should be shared among the processes and the executable code manipulates on these shared data.

Since the symbol table of this portion of the program will be taken as a basis on which the loaded object file defines additional symbols[2], this executable file must be *ld*ed with *-Bstatic* option, so that *ld* may pin down all the symbols. Also in this way, this executable file can incrementally load the to-be-loaded program and run it without worrying about the name binding problem.

## 2.2 The Application Portion

All the threads in the package should have the same address space. An intuitive way to accomplish this is to allocate shared memory segment and

[2]Symbols here are referring to the variable names and the function names.

attach it to the same point in each process's address space. We then load the data portion of the program into this shared segment. In other words, we must do the job which *ld* would have done for us - relocating all the symbols referenced in the application portion. We can do this, but it would add to the complexity of our thread package and make our program error prone. So we use another approach.

Let's clarify the difference between the process data section and the program data section before we go further. An object program has a program data section. After *ld*ing one or some object programs together, we get an executable file. All the program data sections of the object programs are combined to form a single program data section of the executable file. When this executable file runs, it runs within a process and the program data section is loaded into the process data section. Normally the process data section and the program data section are the same.

A process can incorporate at run-time another object program into its address space and run it. This incorporation is called *incremental loading.* In this case, there are two program data sections in the process data section.

The process data section is always private, but shared regions are shared with the child processes. If we load the application portion into a shared memory segment, the program data section of this application portion is shared among all the processes in the thread package. The application portion is loaded into something other than the process data section. Whatever data which should be shared among various processes can be defined as global data in the application portion. Also the code dealing with the global data must be put into the application portion.

In order to make it possible for code in the loader portion to reference code in the application portion and code in the application to reference code in the loader portion, the application portion is *ld*ed with *-N -x -T -A -e* options. The *-N* option will cause *ld* not to make the text portion read-only, implying *-Bstatic.* The *-Bstatic* option causes the binding completed at compile time instead of at run time so all the references are solved and the thread package does not need to invoke the run-time linker. The *-x* option preserves only global symbols in the output symbol tables; only enter external sym-

bols. This option saves some space in the result object program. Because the application portion is not loaded by the OS loader, the loading point is different from the default one used by the OS loader. We must notify *ld* where the loading point would be. Using the *-T hex* option causes *ld* to relocate the text section starting at the address *hex*, so that the object program can be incrementally loaded. The value of *hex* is the address where the loader portion would put the object program. By using *-A name*, *ld* will make the symbol table of file *name* be taken as a basis on which the object program defines additional symbols. Therefore, the object program can access the variables and functions defined in the executable file. The *-e* option defines the entry point of this object program.[3]

## 2.3 File-Descriptor Tables in Kernel Space

Every process has a file-descriptor table in kernel space. The tables of all the processes should always be the same from the view of a thread, e.g. each open file must be referenced by the same file descriptor and have the same file offset in different processes. Although the program data section of the application portion is shared among different processes in our design, these processes still have their own file-descriptor tables. A file opened in one process can be accessed in another - it just has to open the file, but in this case the file offsets in these processes will be different. Since the file offset in different processes must be the same in the thread package, the normal *open* system call cannot be used to open the file because the entries in two process file-descriptor tables will point into different entries in the system file table and the file offset value will be different in the two processes. Different processes cannot share the same file offset unless either the file descriptor is passed from one process to the other which adds a new entry to its file-descriptor table or one process is an ancestor of the other.

The entries in the file-descriptor tables point into the system file table. The entries of the system file contain, among other things, the file offset. We want the processes' file-descriptor tables to be identical, so that their entries all point to the same system file-descriptor table entries. There are two ways

[3]For details, see the ld(1) man page.

to pass an open-file descriptor around. One is to *fork* a child process, which causes the system to copy the file-descriptor table from the parent process to the child process. The other way is to use *sendmsg* and *recvmsg*.

The socket functions, *sendmsg* and *recvmsg*, are supported by both 4.3BSD and System V Release 4. These are the only two functions which can pass an open-file descriptor between two unrelated processes. A process can pack and send an open-file descriptor to another process with *sendmsg*. The other process just uses *recvmsg* to receive and implicitly to open this file as well. For *recvmsg* and *sendmsg* to deliver message reliably, there should be a *connection* between these two processes[4].

We cannot predict which files will be opened before we fork all the processses, so after multiple processes are created, we will use *sendmsg* and *recvmsg* to pass the open-file descriptor around. Since *open*, *socket* and *accept* create a new file descriptor, we *wrap* these three system calls so that it will broadcast the change made in one process to the remaining processes with *sendmsg* and *recvmsg*.

The results of *close*, *dup* and *dup2*, which modify the process file-descriptor table are predictable, i.e. if two file-descriptor tables were initially the same and the same operation is invoked upon these tables respectively, then these two tables are still the same after executing the operation. We *wrap* these three system calls so when any one of these three system calls is invoked in a process, each of the other processes will repeat the operation.

# 3 Implementation

## 3.1 Shared Memory

There are two ways to establish a portion of memory to be accessed by different processes in Unix. The first is to use the *mmap* system call, which establishes a mapping between one process's address space and a memory object represented by a file-descriptor. Multiple processes can invoke *mmap*,

[4]For more details, see the *sendmsg* and *recvmsg* man pages.

mapping this memory object into their own address spaces. Therefore, these processes share this portion of memory.

The second method is to use *shmget* and *shmat* system calls. This kind of shared memory belongs to the category of System V IPC. There is an upper limit on the size of the memory we can request by *shmget* which is less than $2^{20}$ bytes. After requesting a shared memory segment, we attach it to the process's address space by invoking *shmat.*

We use *shmget* and *shmat* to implement our shared memory. The shared memory requested by *shmget* and attached by *shmat* without specifying the desired attaching address will be attached to each process at the same address. So this address can serve as the argument of the *-T* option in *ld*, which is used for relocating the application program. When porting this thread package to other machines, this value can be determined through experimentation.

## 3.2 Loading and Executing

There is a header at the beginning of every object and executable file. The format for the header can be found in */usr/include/a.out.h.*

We are dealing with loading only one executable file and we don't need to take care of name binding and external references. We load in the text and data sections of the application program from disk directly, and then reserve space for the bss section.

The starting address can be found in the header's _entry item.

## 3.3 File-Descriptor Table Updates

Since we use *recvmsg* and *sendmsg* to deal with *open*, there should be connections capable of carrying message among the processes. Connected, UNIX-domain, stream sockets, which are bidirectional and are used in 4.3BSD and in System V for passing file descriptors between processes, can be used to

provide the connection. There are many models for the connection layout. We consider a process as a node in a graph and a connection between two processes as an undirected path. We use $m$ to represent the number of processes.

The first model is to build a complete graph over which the processes can send (receive) the messages to (from) any other processes directly. This model is too expensive to be practical. First of all, we would need $\binom{m}{2}$ stream pipes. Each end of a stream pipe is considered as a file descriptor by the process, so while creating so many stream pipes in order to deal with the file-descriptor table consistence problem, we create a lot more problem to be solved. Also some operations must be done to pick the correct $m$ - *1* stream pipes out of $\binom{m}{2}$ ones to operate on when a process needs to send or receive a message.

Another model that comes to mind is the star layout in which the parent process lies at the center. When a process invokes *open*, it notifies its parent process, if it is not itself the parent process. The parent process will in turn ripple out this modification to all of the other processes. This model still uses too many resources. For $m$ processes, we still need $m$ - *1* stream pipes. In order to broadcast the modification correctly, some operations must be done to differentiate the center (parent) process from the leaf (child) processes, so we can know how to broadcast this modification correctly.

The model we have chosen to use employs only one stream pipe through which all the processes can read and write at both ends. If a process opens a file, it will send $m$ - *1* copies of data through one end of the stream pipe, and notifies each of the other $m$ - *1* processes to get one copy of it.

We use the signal mechanism and we reserve the signal *SIGUSR1* to notify the other processes that they should receive the data from the stream pipe. If a process *opens* a file, it sets a static global variable to indicate that we are doing an *open* and it *sendmsgs* data to the stream pipe and *signals* the remaining processes with *SIGUSR1*. If a process is *signaled* with *SIGUSR1*, it looks up that variable to find out what kind of operation it should take correspondingly. If this variable indicates an *open* has been invoked, it *recvmsgs* the data from the stream pipe. In this way, the newly opened file descriptor

is passed. Similar operations are executed if the *socket* or the *accept* system call is invoked.

For *close*, *dup* and *dup2*, the process which invokes the calls sets that variable to indicate what kind of operation it is executing, stores some necessary data at a shared memory region, and *signal*s all the remaining processes. When a process is *signal*ed and find out that the I/O operation just made by other process belongs to these three system calls, it will fetch the necessary data stored by that process and repeat the operation itself.

We use *socketpair*, is available in both 4.3BSD and in System V, to implement the bidirectional stream pipe. Since all the processes in the thread package run on the same machine, we use the Unix communication domain with *SOCK_STREAM* to make sure the data will not be lost.

Synchronization of all the operations mentioned above can be achieved by using a lock variable and a counter variable. At the beginning of each of the *open*, *socket*, *accept*, *close*, *dup* and *dup2* functions, the lock must be acquired in order to proceed further. This lock won't be released until reaching at the end of each of these functions. The counter variable is assigned one before any signals are sent to all of the other processes. Each signaled process will add one to this variable after the necessary operations are done. In the signaling process, a *while* loop will determine when the counter variable has become $m$. When the counter variable becomes $m$, the lock variable is unlocked.

Here below are the pseudo codes showing how the file-descriptor table is updated:

```
void
SIGUSR1_handler (int sig)
{
    switch (filetable_op) {
      case SYS_close:
        close(sys_fd[0]);
        break;
```

```
        .
        .
        .
        case SYS_open:
          .
          . receive the open-file descriptor
          .
          break;
    }
    THREADlock(&sysio_sync2);
    signal_recv_count++;
    if (signal_recv_count == m) { /* m is the number of processes */
        signal_recv_count = 1;
        THREADunlock(&sysio_sync);
    }
    THREADunlock(&sysio_sync2);
}

int
open (char *filename, int flags, int mode)
{
    THREADlock(&sysio_sync);
    .
    . open the file
    .
    filetable_op = SYS_open;
    .
    . send m - 1 copies of open-file descriptor and kill the
    . corresponding processes
    .
    if (m == 1) /* m is the number of processes */
        THREADunlock(&sysio_sync);

    return (fd);
}
```

## 3.4 THREADmonitorwaitevent

*THREADmonitorwaitevent*, supported in the thread package, is an enhanced version of the system call *select*. A process blocks when it invokes *select* and will unblock when the specified amout of time expires or at least one of the specified files is ready. A thread blocks when it invokes *THREADmonitorwaitevent* and will unblock, in addition to the above two situations, when the condition queue in the monitor is signaled. We use the *select* system call to implement the *THREADmonitorwaitevent* function, so we need to come up with a method to unblock the *select* system call when the condition queue in the monitor is signaled.

In an $m$-process thread package, we create $m$ stream pipes and thus we have $2m$ file descriptors associated with the stream pipes. Although a stream pipe is bidirectional, we only read from the first end of the stream pipe and write to the second end of it. For the first process, the first stream pipe is reserved for it to read, so it closes the second end; the other stream pipes are reserved for it to write, so it closes the second ends of all the other stream pipes. For other processes, similar operations are made as well. The thread package uses the *dup2* system calls to rearrange the file descriptors so that they appear the same in different processes. Before a thread is invoking the *THREADmonitorwaitevent* call, it will set the read bit of the stream pipe which is reserved for it to read. When the condition queue is signaled while the blocking *select* system call is still in effect, the process, on which the condition queue is signaled, just write a byte of random data to the write end of the appropriate stream pipe and the process, which has been blocked by the *select* system call, will unblock.

## 3.5 Machine-Dependent Code

Most of the machine-dependent code results from the need to force a context switch at user level which will change, among other registers, the stack pointer and program counter. Different processors have different instruction sets and registers, so machine-dependent code is inevitable.

On a Sparc workstation, a context switch is accomplished by simulating a return from a function call. A Sparc workstation has a certain number of

sets of registers, called register sets or register windows. At a given time, an instruction can access only one register window through *Current Window Pointer*. When a procedure is called, *CWP* will be decremented by one and therefore this called procedure will access a different register window from the window accessed by the caller. When a procedure returns to its caller, *CWP* will be incremented by one and therefore the caller procedure can access its original register set. There are some overlapping registers between two neighboring windows so that arguments can be passed. A *window_overflow* exception (trap) occurs if a procedure call is made when the windows are full. In this case, the user memory (the stack) is used to save the values of the registers when a procedure is called. A *window_underflow* exception (trap) occurs if a procedure return is made when the windows are empty. In this case, the user memory (the stack) supplies the values of the registers for the procedural context being returned to, and as many additional register sets as are required to fill the register windows. In addition to *window_underflow* and *window_underflow*, there are other kinds of traps as well. A trap is like an unexpected procedure call which decrements *CWP* to point to the next register window. The user code cannot expect to execute the context switch correctly by manipulating the window registers directly because a trap, which cannot be disabled by the non-supervisor code, may occur at any time and change *CWP* and the values of the window registers. The safe method for executing the context switch is to flush the register windows out to user memory (the stack) first by using software trap *#3* (designed for this purpose), and then, the user code can safely manipulate the user memory, instead of the register windows. In fact, the most important reason for flushing the windows is to make certain that a thread's most recent calling history is on its stack. A context switch is accomplished by first invoking the software trap #3 to flush the register windows out to the stack, setting a new stack pointer and the frame, and then invoking the instruction *RESTORE* to fake a *return-from-the-function*. When the instruction *RESTORE* is invoked, a *window_underflow* occurs due to the earlier flush, the system will update the registers by the values we just put on the stack, and a context switch is finished.

Another situation in which we use the machine-dependent code is when we want to *lock* a data item. We can use a system-supported semaphore to do so, but using such a semaphore is expensive. We can implement the *lock*

more efficiently if there are appropriate atomic instructions.

On Sun Sparc 10, *swap* is an atomic instruction which can do the job. A simple way to implement the *lock* function is to have a global integer variable which is unlocked when its value is zero and is locked when its value is one. Inside the *lock* function, we have a local variable which is initially assigned one. We *swap* the global variable and the local variable and, after the *swap*, test if the local variable equals zero or not. If this local variable is zero, we know the global variable was previously not locked and is now locked. If this local variable is one, we know the global variable was locked already, so we must try to lock it again. We don't use *swap* as mentioned above to determine when the global variable becomes zero, because *swap*, the atomic operation, is expensive. We use the normal test to find out when the global variable has become zero. If it becomes zero, we repeat the above procedure, using *swap*, to either get the lock or discover that some other thread has bent us to the lock.

# 4 Efficiency

There are three main sources of overhead in this thread package. The first arises from the fact that this package runs on multiple processes. Creating these processes and initializing the related data incurs overhead both in resources and in time. The package needs two file descriptors for each end of the stream pipe in order to broadcast the opened file descriptor and, for each process, it needs another file descriptor to handle *THREADmonitor-waitevent*. The time overhead involved here results from many system calls and the synchronizations of these operations. Additional overhead arises from the *wrappers* of the *open*, *socket*, *accept*, *close*, *dup*, and *dup2* system calls. This overhead increases as the the number of processes in the thread package increases. Overhead also arises from the context switching and the thread synchronization between different threads.

We cannot reduce the first and second kinds of overhead by just reducing the number of processes, because this package is built on the assumption of running on multiple processes to minimize the effects of the blocking I/O

| Standard I/O | One Process | Five Processes |
|---|---|---|
| 45.507784 | 47.650487 | 7482.047919 |

Table 1: The time (in seconds) of executing 10000 open and close calls.

system calls. On the surface, one may think that the overall performance may improve proportionally to the number of processes. This turns out not to be true. Unix uses a time-slicing scheduler and each process is given a specific unit of execution time. Since the threads can migrate among different processes and if we have $m$ processes, we can take advantage of $m$ units of execution time, intead of just one unit. One thing that needs to be kept in mind is that if the thread package runs on a work station, there may be not so many processes running on the system, so the $m$ units of time slices may be less beneficial than it appears. Specifying the appropriate number of processes is not an easy job for the user.

If the user program invokes a lot of *open*, *close*, *socket*, *accept*, *dup*, and *dup2* calls, overall performance will suffer. Table 1 shows us the time needed to perform 10000 times of pairs of *open* followed by *close* operations. From this table, if the thread package runs on only one process (The "One Process" column in the table), the time necessary to perform the pairs of *open* and *close* is slightly longer than the time required for normal open and close operations (The "Standard I/O" column). When the thread package runs on five processes, the calls require more than 166 times the time (The "Five Processes" column) than one process needs to perform the same operations. Judging from the huge difference, a program invoking a lot of these I/O system calls would probably not be suitable to run in this thread package.

Here is another comparison. A sample program implements a file copy command. In order to manifest the time needed to execute the *read* and *write* commands, only one character is read (written) by the sample program at one *read* (*write*) command. Table 2 shows the time needed to copy a file with 273101 characters. As we can see from this table, the performances of the various cases are similar. These two comparisons also confirm that the extra overhead for I/O occurs only for calls such as *open*, *close*, etc., that manipulate file descriptors. Since such calls are relatively rare in most

| Standard I/O | One Process | Five Processes |
|---|---|---|
| 21.249974 | 21.286403 | 21.44851 |

Table 2: The time (in seconds) of copying a file with 273101 bytes.

programs, the I/O part of the thread package is not terribly expensive to use.

# 5 Bugs, Features, Etc.

Because we support multiple threads with multiple processes, we can handle I/O system calls without the risk of unnecessarily blocking other threads. If too few processes are available and too many threads invoke blocking I/O system calls, all processes would be blocked and no more threads would be able to run until at least one process unblocks. The problem may result because too few processes were requested by the user or because of the operating system's upper limit on the number of processes per user. As for the first case, the user may increase the first argument of *THREADgo* which is the function the user invokes to start the thread package. For the second case, there is not much that can be done, short of increasing OS-specific limits.

We request segments of shared memory from OS in the thread package. Normally these segments of shared memory will be released by the thread package before exiting. If something goes wrong, these segments might not be released, and they would exist in the system and waste valuable resources. The shared memory we use is an interprocess communication facility. One can use the *ipcs* command to discover such orphaned resources and use the *ipcrm* command to clean them up.

# 6 Using This Thread Package

All the user programs should include *thread.h*. The user programs cannot have *main* program. The first function to be executed in the user program

may take two arguments: (int argc, char *argv[]), the same as the *main* function in a normal C program.

Suppose the first function to be executed is called *starter* and the target machine is *Sun Sparc 10.* A command to link this object code is:

```
ld -N -x -T ef6ff000 -A loadfunc -e _starter -Lwherethelibraryis
yourprogram.o -o yourprogram -lthread -lc
```

To run the program:

```
loadfunc yourprogram [arguments].
```

Note for *starter*, argc will be equal the number of arguments plus 1; argv[0] is yourprogram.

# Acknowledgements

This work is mainly based on the thread package built by Professor Tom Doeppner. David Edelsohn's source code dealing with increment loading helped me overcome the difficulty of increment loading.

# References

[1] Thomas W. Doeppner Jr. *A Threads Tutorial. Technical Report CS-87-06.*

[2] W. Richard Stevens. *Unix Network Programming.*

[3] Andrew S. Tanenbaum. *Modern Operating System.*

This work is mainly based on the *Brown Threads* package. Here is a list of the significant modifications I made or new code I added to the package:

- The atomic *lock*, *unlock*, and *condlock* functions. These functions are in *lock.s*.
- Adding wrappers for *open*, *close*, *read*, *write*, *socket*, and *accept* system calls. In order to make these wrapper work properly, I utilized the *SIGUSR1* signal and wrote a handler for this signal. These functions are in *io.c*.
- The *loadfunc* and *shmdemand* functions. The *loadfunc* function is based on David Edelsohn's source code. The *shmdemand* function requests shared memory from the O.S.. These two functions are in *loadfunc.c*.
- The *malloc*, *free*, and *memalign* functions. These functions replace their counterpart in the standard library. These functions are in *malloc.c*.
- Creating more than one process and initializing the related data. Most of the work can be found in *Threadgo* in *mp.c*.
- A new queue mechanism. In the *Brown Threads* package, a queue is represented by two pointer: one pointing to the first element in the queue and the other one pointing to the last element in the queue. The queue I wrote just has one pointer pointing to the last element in the queue. Also the queue is circular. The queue is defined in *queue.c*.
- Testing if there is any process idle due to *sigpause* in *MOVETORUNQ*. *MOVETORUNQ* is defined in *runQ.c*.
- Move all the definition from the header files to the source files. Leave only the declaration in the header files. The only exception is in *loadfunc_only.h* in which the variables specific to each process are defined.
- Adding more comments to the source code.

```
################################################################################
# modified from Brown cs132 sample C++ makefile
#
# Copy this file into the directory where you keep the source files of
# your program. Change the variables EXEC and OFILES below to reflect
# your source. When you are ready to compile your program, just type
# 'make' in the directory.
################################################################################

.KEEP_STATE:
.SILENT:
.SUFFIXES: .c .h .o .s


################################################################################
# Change EXEC to the name of the executable which you wish to
# create. Make OFILES be the name of all the .C files in your program,
# but change the .C to .o.
################################################################################

MACH=sun4

BINDIR = /u/tch/project/bin

LIBFILE = libthread.a
OFILES  = mp.o semaphore.o manager.o thread.o stack.o family.o runQ.o\
          tsignal.o logerr.o monitor.o excp.2nd.o excp.low.o\
          queue.o lock.o thread_type1.o io.o preempt.o \
          malloc.o
LOFILES = loadfunc.o

SCRIPT  = /u/twd/os/lwp/src/lowlevel/asm.sed.$(MACH)


# Compiler specifications
CC      = /usr/lang/acc -g
AR      = /usr/bin/ar
RANLIB  = /usr/bin/ranlib
AS      =/usr/bin/as


IFLAGS  =
LFLAGS  =
LIBS    = -lc

################################################################################
# These are different make options.  You can invoke them by typing
# the label name.  For example to time your make, type 'make time' --
# 'make' being the name of this command and 'time' being the label used
# below to define the result.
################################################################################

# Standard make, it makes everything.  You can type 'make all' or
# just 'make'
all:    loadfunc $(LIBFILE)
        echo -n "make finished at "; date

loadfunc: $(LOFILES)
        echo "acc -Bstatic ... -o loadfunc"
        $(CC) -Bstatic $(LOFILES) -o $(BINDIR)/loadfunc

$(LIBFILE): $(OFILES)
        echo "rm -f threadlib.a"
        -rm -f threadlib.a
        echo "ar ru threadlib.a $<"
        $(AR) ru $(LIBFILE) $(OFILES)
        echo "ranlib threadlib.a"
        $(RANLIB) $(LIBFILE)

# All .C files depend on their headers
%.c:    %.h

#defin a rule for building .o from .s file
.s.o:
        echo "compiling $<"
        $(AS) $< -o $*.o

# Define a rule for building .o from .C files
.c.o:
        echo "compiling $<"
        $(CC) -S $(IFLAGS) $<
        sed -f $(SCRIPT) < $*.s > $*.tmp
        as -o $*.o $*.tmp
        rm -f $*.s $*.tmp


# Clean up after making.
clean:
        echo "removing" *.o
        $(RM) *.o
        if [ -d /tmp/$(USER)tmp ]; \
         then echo "clearing /tmp/$(USER)tmp"; $(MAKE) tclean; fi

# Make tags for emacs and vi. The result will be stored in a text
# file 'TAGS' in your current directory.
tags:
        etags -t *.[CH]

# Time your make.
time:
        /usr/bin/time $(MAKE)
```

```
/call.*THREADsave_and_flush_window.*/,/.*/{                                    }
/call.*THREAD.*/d
a\
        st %sp, [%o0]           ! save sp \
        sub %sp, 64, %o7        ! compute end of next frame \
        st %o7, [%o1]           ! save next frame's sp \
        st %sp, [%o7+56]        ! save next frame's fp in frame \
        t       3               ! ST_FLUSH_WINDOWS trap
}

/call.*THREADflush_window.*/,/.*/{
/call.*THREAD.*/d
a\
        t 3         ! ST_FLUSH_WINDOWS trap
}

/call.*GETFRAME.*/,/.*/{
/call.*GET.*/d
a\
        mov %fp, %o0
}

/call.*SETFRAME.*/,/.*/{
/call.*SET.*/d
a\
        mov %o0, %fp            ! establish new frame\
        st %l0, [%fp]           ! save l0 across restore\
        st %l1, [%fp+4]         ! save l1 across restore\
        st %l2, [%fp+8]         ! save l2 across restore\
        st %l3, [%fp+12]        ! save l3 across restore\
        st %l4, [%fp+16]        ! save l4 across restore\
        st %l5, [%fp+20]        ! save l5 across restore\
        st %l6, [%fp+24]        ! save l6 across restore\
        st %l7, [%fp+28]        ! save l7 across restore\
        st %i0, [%fp+32]        ! save i0 across restore\
        st %i1, [%fp+36]        ! save i1 across restore\
        st %i2, [%fp+40]        ! save i2 across restore\
        st %i3, [%fp+44]        ! save i3 across restore\
        st %i4, [%fp+48]        ! save i4 across restore\
        st %i5, [%fp+52]        ! save i5 across restore\
        restore %g0, 0, %g0     ! load window
}

/call.*SETSTACK.*/,/.*/{
/call.*SET.*/d
a\
        mov %o0, %sp
}

/call.*GETSTACK.*/,/.*/{
/call.*GET.*/d
a\
        mov %sp, %o0
}

/^_THREADrun:/,/ret$/{
/%o0,%i0/d
}

/call.*THREADfixregs.*/,/.*/{
/call.*THREAD.*/d
a\
        ld [%fp+32], %i0\
        ld [%fp+36], %i1
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#include <stdio.h>

#include "manager.h"
#include "family.h"
#include "privatedata.h"
#include "excp.2nd.h"



/* I wonder if sun4 is defined */
#ifdef sun4
#include <sun4/setjmp.h>
#endif

/*****************************************************************************/
/* THREADexceptioncall                                                       */
/*                                                                           */
/* vax: entry mask is set by asm.sed to save registers 1 - 11                */
/* sun3: asm.sed adjusts entry code so that a2-a5 and d2-d7 are saved        */
/* mips: this needs to have a procedure epilogue identical to  that of       */
/* THREADsetexception.  This isn't easy, so we call THREADsetexception       */
/*****************************************************************************/

int
THREADexceptioncall ()
{

    /* if this thread was waiting for a child, we must turn off the flag that
       says it's doing this */
    THREAD_WAITCHILD_RESET(THREADcurrent);

    /* tell the thread's manager about the exception so that it can
       clean up */
    THREADtellmanagement(MANAGER_EXCEPTION, THREADcurrent);

#ifdef FORTRAN
    family_returnvalue(THREADcurrent) =
        (*THREADexception_handler(THREADcurrent))
            (&THREADexception_param(THREADcurrent));
#else
#ifdef PASCAL
    family_returnvalue(THREADcurrent) =
        FIXPASC(THREADexception_handler(THREADcurrent),
                THREADexception_param(THREADcurrent));
#else
    family_returnvalue(THREADcurrent) =
        (*THREADexception_handler(THREADcurrent))
            (THREADexception_param(THREADcurrent));
#endif
#endif FORTRAN

    THREADexception_inprogress(THREADcurrent) = 0;
    THREADexception_param(THREADcurrent) = 0;
    THREADexception_handler(THREADcurrent) = NULL;

    /* restore frame that was current at the time of the call to
       THREADsetexception */
#if !defined(sparc) && !defined(mips)
    RESTORE_FRAME(THREADexception_frame(THREADcurrent), THREADFRAMESIZE);
#else
#ifdef sparc
    THREADflush_window();
    bcopy(THREADexception_frame(THREADcurrent),
          THREADexception_stack(THREADcurrent), THREADFRAMESIZE);
    SETFRAME(THREADexception_stack(THREADcurrent));
#endif sparc
#ifdef mips
    /* it is too difficult to set up the stack frame correctly for the
       mips, so we just go to setexception which has a correct stack frame */
    (void)THREADsetexception(NULL, NULL);
    logerr(DISASTER, "THREADexceptioncall: should not be here!");
#endif mips
#endif

    return(1);
}


/*****************************************************************************/
/* THREADgetexceptioonreturn                                                 */
/*****************************************************************************/

int
THREADgetexceptionreturn ()
{
    return(family_returnvalue(THREADcurrent));

}


/*****************************************************************************/
/* THREADrestoreexception                                                    */
/*****************************************************************************/

void
THREADrestoreexception (EXCEPTION oldstate)
{
    THREADexception_state(THREADcurrent) = *oldstate;
}


/*****************************************************************************/
/* THREADgetexceptionspace                                                   */
/*****************************************************************************/

EXCEPTION
THREADgetexceptionspace ()
{
    EXCEPTION retcode;

    retcode = (EXCEPTION)_malloc(sizeof(*retcode));

    return(retcode);
}


/*****************************************************************************/
/* THREADfreeexceptionspace                                                  */
/*****************************************************************************/
```

```
void
THREADfreeexceptionspace (EXCEPTION space)
{
    _free(space);
}
```

```
/************************************************************************/
/*        thread - thread package.                                      */
/*                                                                      */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University      */
/*        - All rights reserved.                                        */
/************************************************************************/




#include <stdio.h>
#include <signal.h>

#include "tcb.h"
#include "preempt.h"
#include "logerr.h"
#include "mp.h"
#include "runQ.h"

#include "excp.low.h"

/************************************************************************/
/* THREADsetexception                                                   */
/*                                                                      */
/* A thread may define a single exception handler.                      */
/* When an exception is raised, the handler is called from the point    */
/* at which this call to setexception was made: this act of raising     */
/* the exception causes a value to be stored in the                     */
/* exception_param field of the tcb.  When the thread is made           */
/* current by THREADrun, it unwinds the thread's stack to the point     */
/* at which the setexception was made, then simulates a call to         */
/* THREADexceptioncall.   This routine calls the thread's exception     */
/* handler, stores the returned value in the tcb, then returns a 1,     */
/* which will appear as a second return from setexception (i.e.,        */
/* setexception is much like setjmp, and raise exception is much        */
/* like longjmp).                                                       */
/* THREADsetexception returns a 0 on its first return.                  */
/*                                                                      */
/* vax: entry mask is set by asm.sed to save registers 1 - 11           */
/* sun3: asm.sed adjusts entry code so that a2-a5 and d2-d7 are saved   */
/* ns32000: registers r2-r7 are saved via asm.sed                       */
/* mips: all savable registers are saved via asm.sed.  The frame size   */
/*  is hardwired (THREADFRAMESIZE), so changes to this routine          */
/*  may require changes to asm.sed.mips                                 */
/************************************************************************/


int THREADsetexception (int (*handler)(), EXCEPTION oldstate)
{
#ifdef mips
    int regs[24];

    dummy(regs);
#endif mips

    THREAD_PROTECT;

    if (oldstate != NULL)
        *oldstate = THREADexception_state(THREADcurrent);

    /* at this point we save the thread's current stack frame.  This is
       basically a "setjmp", but setjmp can't be used here, since we need
       to save the stack frame of this routine, whereas setjmp would save
       its own stack frame. */

#ifdef mips
    /* this effects a longjmp back to the point of the setexception;
       this should be doable in exceptioncall, but doing so requires
       that exceptioncall have the same stack frame as this routine,
       which is too hard to accomplish with a simple sed script */
    if (handler == NULL) {
        /* longjmp */
        bcopy(THREADexception_frame(THREADcurrent),
              THREADexception_stack(THREADcurrent)+4, THREADFRAMESIZE);
        SETSTACK(THREADexception_stack(THREADcurrent));
        return(1);
    }
#endif mips
#if !defined(sparc) && !defined(mips)
    /* copy the current stack frame so that it can be restored after
       the exception handler has completed execution */
    SAVE_FRAME(THREADexception_frame(THREADcurrent), THREADFRAMESIZE);

    /* get stack pointer at time of call to this routine */
    GET_PREVIOUS_STACK(THREADcurrent);
#else
#ifdef sparc
    {
        int dummy;

        THREADsave_and_flush_window(&THREADexception_stack(THREADcurrent),
                                    &dummy);
        bcopy(THREADexception_stack(THREADcurrent),
              THREADexception_frame(THREADcurrent), THREADFRAMESIZE);
    }
#endif sparc
#ifdef mips
    {
        char *sp = (char *)GETSTACK();

        bcopy(sp+4, THREADexception_frame(THREADcurrent), THREADFRAMESIZE);
        THREADexception_stack(THREADcurrent) = sp;
    }
#endif mips
#endif

    THREADexception_handler(THREADcurrent) = handler;

    THREAD_UNPROTECT;

    return(0);
}



/************************************************************************/
/* THREADraiseexception                                                 */
/*                                                                      */
/* An exception with # param is sent to tcbp. If the thread             */
/*   running in this process: do a longjmp, by calling THREADrun, no return*/
/*   running not in this process: issue a SIGTALRM                      */
/*   not running...                                                     */
/* we set appropriate data below, so THREADrun could check it out       */
/************************************************************************/


int THREADraiseexception (THREAD tcbp, int param)
```

```
{
    LOCK *qlock;

    THREAD_PROTECT;

    if (THREAD_WHICH_QUEUE(tcbp) && !THREAD_RUNNABLE(tcbp)) {
        qlock = &THREAD_WHICH_QUEUE(tcbp)->lock;
        THREADlock(qlock);
    } else
        qlock = NULL;
    THREADlock(&tcbp->context.lock);

    if (THREADexception_handler(tcbp) == NULL) {
        /* no handler, default is to ignore */
        THREADunlock(&tcbp->context.lock);
        if (qlock != NULL)
            THREADunlock(qlock);
        THREAD_UNPROTECT;
        logerr(INFO, "THREADraiseexception: called with no handler");
        return(0);
    }
#ifdef notdef
    if (param == 0) {
        /* param must be nonzero */

        THREADunlock(&tcbp->context.lock);
        THREAD_UNPROTECT;
        logerr(WARNING, "THREADraiseexception: param is zero");
        return(0);
    }
#endif notdef

    /*************************************************************
     * By comparing the current sp with THREADexception_stack
     * we can tell if this exception handler is still good or not.
     * Remember that stack grows downwards
     *************************************************************/
    if ((!THREADcompstack(tcbp, THREADexception_stack(tcbp))) &&
        /* delay this test if the thread is running on another cpu */
        !(THREAD_RUNNING(tcbp) && tcbp != THREADcurrent)) {
        /* no more good: the exception stack is beyond the current stack */
        THREADunlock(&tcbp->context.lock);
        if (qlock != NULL)
            THREADunlock(qlock);
        THREAD_UNPROTECT;
        logerr(ERROR, "THREADraiseexception: handler is no longer valid");
        return(0);
    }
#ifdef notdef
    if (THREADexception_param(tcbp) == 0) {
      /* If param is non zero, then there is an exception in progress */
#endif notdef
    if (!THREADexception_inprogress(tcbp)) {
        /* informs THREADrun about exception */
        THREADexception_inprogress(tcbp) = 1;
        THREADexception_param(tcbp) = param;
        THREAD_EXCEPTION_SET(tcbp);

        if (THREAD_RUNNING(tcbp)) {         /* qlock is not locked */
            if (tcbp == THREADcurrent) {
                /* If raising an exception in ourself, then have run do the
                   longjmp */
                THREADlock(&THREADrunqlock);
                THREAD_MOVETORUNQ(tcbp);
                THREAD_PASSPROTECT;
                THREADrun();
                logerr(DISASTER, "THREADraiseexception: should not be here!");
            } else {
                /* we must signal the other thread's processor */
                register int cpu;

                cpu = tcbp->context.processor;
                THREADunlock(&tcbp->context.lock);
                kill(cpu, SIGVTALRM);
                THREAD_UNPROTECT;
                return(2);
            }
        }
        if (!THREAD_RUNNABLE(tcbp)) {
            THREADpullfromqNL(tcbp);
            THREADunlock(qlock);
            THREADlock(&THREADrunqlock);
            THREAD_MOVETORUNQ(tcbp);
            THREADunlock(&THREADrunqlock);
        }
        /* otherwise the thread is already in the runq, and it will notice
           the exception soon (qlock is not locked) */
        THREADunlock(&tcbp->context.lock);
        THREAD_UNPROTECT;

        return(1);
    } else {
        THREADunlock(&tcbp->context.lock);
        if (qlock != NULL)
            THREADunlock(qlock);
        THREAD_UNPROTECT;
        logerr(WARNING, "THREADraiseexception: exception in progress");
        return(0);
    }
}


/*****************************************************************************/
/* THREADexceptionthreadinit                                                 */
/*****************************************************************************/

void
THREADexceptionthreadinit (THREAD t)
{
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/


#include <stdio.h>
#include <signal.h>
#include <alloca.h>
#include "tcb.h"
#include "privatedata.h"
#include "runQ.h"
#include "queue.h"
#include "family.h"
#include "thread.h"
#include "thread_type1.h"
#include "tsignal.h"
#include "preempt.h"
#include "logerr.h"




/*****************************************************************************/
/* fetch_next_exitingchildren                                                */
/*****************************************************************************/

static
THREAD
fetch_next_exitingchildren (THREAD t)
{
    THREAD p;
    if ((p = (family_exitingchildren(t))) != NULL) {
        family_exitingchildren(t) = family_sibling(p);
        family_ndchildcount(t)--;
    }

    return p;
}



/*****************************************************************************/
/* move_from_children_to_exitingchildren                                     */
/*                                                                           */
/* move the thread itself from its parent's children to exitingchildren      */
/*****************************************************************************/

static
int
move_from_children_to_exitingchildren (THREAD c)
{
    THREAD *tp;

    /* remove from children */
    tp = &(family_children(family_parent(c)));
    while ((*tp)) {
        if ((*tp) == c)
            break;
        tp = &(family_sibling(*tp));
    }

    if ((*tp) == NULL)
        return 0;

    /* now, tp is the addess of the item sibling pointing to *c */
    (*tp) = family_sibling(*tp);

    /* move to exitingchildren */
    family_sibling(c) = family_exitingchildren(family_parent(c));
    family_exitingchildren(family_parent(c)) = c;

    THREAD_EXITING_SET(c);

    return 1;
}




/*****************************************************************************/
/* THREADstartup                                                             */
/*                                                                           */
/* The first routine of any thread.                                          */
/* Step:                                                                     */
/* (1) It calls the func, but in case of being murdered while executing      */
/*     func, it stores the jmpbuf.                                           */
/* (2) Take care of the family business                                      */
/*****************************************************************************/


void
THREADstartup (int (*func)(char *[]), char *argv[])
{
    /* longjmp lies in THREADsuicide function */
    if ((_setjmp(family_terminate(THREADcurrent())) ||
        THREAD_MURDERED(THREADcurrent)) {
        if (!THREAD_INDEP(THREADcurrent))
            family_returnvalue(THREADcurrent) = 0;
    } else {
        family_returnvalue(THREADcurrent) = (*func)(argv);
    }

    /* this thread has done its own business, from now taking care of family */

    THREAD_PROTECT;
    THREADlock(&((THREADcurrent)->context.lock));

    /* don't proceed until all nondetached children disappear */
    while ((family_ndchildcount(THREADcurrent)) > 0) {
        THREAD tp;

        while ((tp = fetch_next_exitingchildren(THREADcurrent)) != NULL) {
            /* handle any that have terminated but haven't been picked up ! */

            THREADactivedecr;
            THREADunlock(&((THREADcurrent)->context.lock));

            /*
             * possible race condition here: if a parent gets murdered while
             * in this loop, it will go back to the setjmp and enter loop
             * again. The thread referred to by tp at the point of the murder
             * will not be destroyed and will be off the exiting list,
             * so never will be noticed again.  We can't leave the current
```

```
            * thread locked because we are calling user code below and we
            * can't move the removal of tp from the queue to below because
            * then we might end up calling the TERMINATE routine twice
            */

           THREAD_UNPROTECT;
           THREAD_TYPE1_THREAD_TERMINATE(THREADcurrent);
           THREAD_PROTECT; /* tch: I don't know why doing so */
           THREADlock(&(tp->context.lock));
           THREADdestroy(tp);
           THREADunlock(&((THREADcurrent)->context.lock));
       }

       if ((family_ndchildcount(THREADcurrent)) <= 0) /* all done */
           break;

       THREAD_WAITCHILD_SET(THREADcurrent);  /* set the status as waiting */
       THREADlock(&(waitq.lock));
       THREAD_TCBAPPEND(THREADcurrent, &waitq);
       THREADunlock(&(waitq.lock));

       THREAD_PASSPROTECT;
       THREADlock(&THREADrunqlock);

       THREADrun();                /* wait for child */

       THREAD_PROTECT;
       THREADlock(&((THREADcurrent)->context.lock));
   }

   /* from now, no children of this thread */

   if (THREAD_DETACHED(THREADcurrent)) {
       THREADunlock(&((THREADcurrent)->context.lock));
       THREAD_UNPROTECT;
       THREAD_TYPE1_THREAD_TERMINATE(THREADcurrent);
       THREAD_PROTECT;
       THREADactivedecr;
       THREAD_PASSPROTECT;
       THREADlock(&((THREADcurrent)->context.lock));
       THREADlock(&THREADrunqlock);
       THREADdestroy(THREADcurrent);

       logerr(DISASTER, "THREADstartup: should not be here (1)");
   } else if (THREAD_INDEP(THREADcurrent)) {
       /* what kind of situation ?? */
       if (family_parent(THREADcurrent)) {
           /* chou: parent is waiting for it ?? */
           if(!THREADcondlock(&(family_parent(THREADcurrent)->context.lock))){
               THREADunlock(&((THREADcurrent)->context.lock));
               THREADlock(&(family_parent(THREADcurrent)->context.lock));
               THREADlock(&((THREADcurrent)->context.lock));
           }

           THREAD_WAITCHILD_RESET(family_parent(THREADcurrent));
           THREADlock(&THREADrunqlock);
           THREAD_MOVETORUNQ(family_parent(THREADcurrent));
           THREADunlock(&THREADrunqlock);
           THREADunlock(&(family_parent(THREADcurrent)->context.lock));
           THREAD_RUNNING_RESET(THREADcurrent);
           THREADunlock(&((THREADcurrent)->context.lock));
           THREADcurrent = NULL; /* no need to save state of current thread */
           THREAD_PASSPROTECT;
           THREADrun();

           logerr(DISASTER, "THREADstartup: shouldn't be here (3)!");
       } else {
           THREAD_EXITING_SET(THREADcurrent);
           /* chou thinks this thread should be put onto freeit_list */
           THREADlock(&(waitq.lock));
           THREAD_TCBAPPEND(THREADcurrent, &waitq);
           THREADunlock(&(waitq.lock));
           THREAD_PASSPROTECT;
           THREADlock(&THREADrunqlock);
           THREADrun();

           logerr(DISASTER, "THREADstartup: shouldn't be here (4)!");
       }
   } else { /* NON-detached, this is the normal case */
       if (!THREADcondlock(&(family_parent(THREADcurrent)->context.lock))) {
           THREADunlock(&((THREADcurrent)->context.lock));
           THREADlock(&(family_parent(THREADcurrent)->context.lock));
           THREADlock(&((THREADcurrent)->context.lock));
       }

       /* now we have both self and parent's locks */

       if ((move_from_children_to_exitingchildren(THREADcurrent)) == NULL) {
           logerr(DISASTER,
                  "THREADstartup: dying thread's parent has no children");
       }
       THREAD_EXITING_SET(THREADcurrent);

       if (THREAD_WAITCHILD(family_parent(THREADcurrent))) {
           /* the parent waits for it, wake it up */
           THREADlock(&waitq.lock);
           THREADpullfromqNL(family_parent(THREADcurrent));
           THREADunlock(&waitq.lock);
           THREAD_WAITCHILD_RESET(family_parent(THREADcurrent));
           THREADlock(&THREADrunqlock);
           THREAD_MOVETORUNQ(family_parent(THREADcurrent));
           THREADunlock(&THREADrunqlock);
       }

       THREADunlock(&((family_parent(THREADcurrent))->context.lock));
       THREAD_RUNNING_RESET(THREADcurrent);
       THREADunlock(&((THREADcurrent)->context.lock));
       THREADcurrent = NULL; /* no need to save the state of it */

       THREAD_PASSPROTECT;
       THREADrun();
   }

   logerr(DISASTER, "THREADstartup: shouldn't be here (5)!");
}



/*****************************************************************************/
/* THREADwaitforchild                                                        */
/*                                                                           */
/* Suspend the caller until one of its nondetached children terminates.      */
/*****************************************************************************/



THREAD
THREADwaitforchild ()
```

```
{
    THREAD child;

    THREAD_PROTECT;

    if ((family_ndchildcount(THREADcurrent)) <= 0) {
        /* no non-detached children */
        THREADunlock(&((THREADcurrent)->context.lock));
        THREAD_UNPROTECT;
        return NULL;
    }

    while ((family_exitingchildren(THREADcurrent)) == NULL) {
        THREAD_WAITCHILD_SET(THREADcurrent);/* depend on children to wake up */
        THREADlock(&(waitq.lock));
        THREAD_TCBAPPEND(THREADcurrent, &waitq);
        THREADunlock(&(waitq.lock));
        THREAD_PASSPROTECT;
        THREADlock(&THREADrunqlock);

        THREADrun(); /* will be waked up by the child */

        THREAD_PROTECT;
        THREADlock(&((THREADcurrent)->context.lock));

        if ((family_ndchildcount(THREADcurrent)) <= 0) {
            /* children may be murdered */
            THREADunlock(&((THREADcurrent)->context.lock));
            THREAD_UNPROTECT;
            return NULL;
        }
    }

    /* tell caller about the first of its exiting children */
    child = fetch_next_exitingchildren(THREADcurrent);
    THREAD_DONE_SET(child);
    THREADunlock(&((THREADcurrent)->context.lock));
    THREADactivedecr;
    THREAD_UNPROTECT;
    return child;
}




/*****************************************************************************/
/* THREADeliminatechild                                                      */
/*                                                                           */
/* eliminate the THREAD passed in                                            */
/*****************************************************************************/


int
THREADeliminatechild (THREAD t)
{
    THREADlock(&(t->context.lock));
    if (!THREAD_DONE(t)) {
        THREADunlock(&(t->context.lock));
        logerr(ERROR, "THREADeliminatechild: child is not done.");
        return 0;
    }

    THREADunlock(&(t->context.lock));
    THREAD_TYPE1_THREAD_TERMINATE(t);
    THREADlock(&(t->context.lock));
    THREADdestroy(t);
    return 1;
}


/*****************************************************************************/
/* THREADreturnvalue                                                         */
/*                                                                           */
/* return the thread's return value                                          */
/*****************************************************************************/


int
THREADreturnvalue (THREAD t)
{
    return(family_returnvalue(t));
}


/*****************************************************************************/
/* THREADmurder                                                              */
/*                                                                           */
/* kill the thread passed in                                                 */
/*****************************************************************************/

void
THREADmurder (THREAD t)
{

    if (t == THREADcurrent) {
        THREADsuicide();
        return;
    }

    THREAD_PROTECT;

  getlocks:
    while (1) {
        /*
         * lock the run queue and the thread so that we can set the dying bit
         * and make certain that it gets noticed asap.  Since we don't know yet
         * the correct order for locking, this code will avoid deadlock
         */
        THREADlock(&(t->context.lock));
        if (THREAD_RUNNING(t) || THREAD_RUNNABLE(t) ||
            THREADcondlock(&THREADrunqlock))
            break;
        THREADunlock(&(t->context.lock));
    }

    /* Both t and runQlock have been locked already */

    if (THREAD_EXITING(t)) {
        /* thread is exiting, leave it alone */
        THREADunlock(&THREADrunqlock);
        THREADunlock(&(t->context.lock));
        THREAD_UNPROTECT;
        return;
    }

    THREAD_DYING_SET(t);

    if (THREAD_RUNNING(t)) {
```

```
        logerr(INFO, "THREADmurder will SIGVTALRM");
        /*
        kill(t->context.processor, SIGVTALRM);
        *//* this causes a reschedule */
        logerr(DISASTER, "sigvtalrm not implemented yet!!");
    } else {
        if (!THREAD_RUNNABLE(t)) {
            if (THREAD_WAITCHILD(t))
                THREAD_WAITCHILD_RESET(t);
            else {
                THREAD_QUEUE queue;

                queue = THREAD_WHICH_QUEUE(t);
                if (!THREADcondlock(&queue->lock)) {
                    THREADunlock(&THREADrunqlock);
                    THREADunlock(&t->context.lock);
                    goto getlocks;
                }
                THREADpullfromqNL(t);
                THREADunlock(&queue->lock);
            }

            THREAD_MOVETORUNQ(t);    /* don't we need to set it RUNNABLE ?? */
            THREADunlock(&THREADrunqlock);
        }
    }

    THREADunlock(&(t->context.lock));
    THREAD_UNPROTECT;
    return;
}



/*****************************************************************************/
/* THREADsuicide                                                             */
/*                                                                           */
/*****************************************************************************/

void
THREADsuicide ()
{
    int i, nchildren;
    THREAD *childlist, child;

    if (THREAD_CREATING(THREADcurrent)) {
        THREAD_DYING_SET(THREADcurrent);
        return;
    }


    THREAD_PROTECT;
    THREADlock(&((THREADcurrent)->context.lock));

    nchildren = family_ndchildcount(THREADcurrent);
    childlist = (THREAD *)alloca(nchildren*sizeof(THREAD));
    child = family_children(THREADcurrent);
    /*
     * we'd like to call THREADmurder for each child.  We can't do this in
     * the following loop, because children may terminate and remove
     * themselves from the list while we are searching it.  So we lock
     * the current thread, make a list of all children, then unlock the
     * current thread and kill off the children; if a child terminates before
     * we murder it, it will be marked as exiting and murder will cope
     */
    i = 0;
    while (child != NULL) {
        /* make a list of all nondetached children */
        childlist[i++] = child;
        child = family_sibling(child);
    }
    THREADunlock(&((THREADcurrent)->context.lock));
    THREAD_UNPROTECT;
    if (i != nchildren) {
        logerr(DISASTER, "THREADsuicide: ndchildcount is wrong");
    }
    for (i=0; i < nchildren; i++) {
        THREADmurder(childlist[i]);
    }

    /*
     * now that we've dealt with the children, longjmp back to startup so
     * that we can self-destruct
     */

    THREAD_MURDERED_SET(THREADcurrent);
    if (((int*) family_terminate(THREADcurrent))[5] != NULL) {
        /* check the saved fp to see if the setjmp has been done */
        /* this thread is murdered before it can do _setjmp */
        _longjmp(family_terminate(THREADcurrent), 1);

        logerr(DISASTER, "THREADsuicide should not be here");
    }

    /*
     * if the setjmp has not been done, then we return back to the thread
     * which will notice that the murdered bit has been set and terminate
     * itself
     */
    return;
}


/*****************************************************************************/
/* THREADfsiginit                                                            */
/*                                                                           */
/* Frozen parent's child tcb's first function. It will be a parameter to     */
/* THREADclonetcb                                                            */
/*****************************************************************************/


void
THREADfsiginit (struct sigparm *sp)
{
    struct frame *fp;

    if (!(THREAD_CHILDOFFREEZE(THREADcurrent))) {
        logerr(DISASTER, "THREADfsiginit: not child of freeze");
        return;
    }

    /* link the thread's stack to its parent's stack, so that a stack trace
       shows where the thread came from */

#ifdef vax
    fp = (struct frame *)GETFRAME();
    fp->fr_savfp = (int)THREAD_FRAME(THREAD_FROZEN_PARENT(THREADcurrent));
```

```
#endif
#ifdef ns32000
    fp = (struct frame *)GETFRAME();
    fp->link =
        (struct frame *)THREAD_FRAME(THREAD_FROZEN_PARENT(THREADcurrent));
#endif
#if defined(sun) && (defined(mc68010) || defined(mc68020))
    fp = (struct frame *)GETFRAME();
    fp->fr_savfp =
        (struct frame *)THREAD_FRAME(THREAD_FROZEN_PARENT(THREADcurrent));
#endif
#ifdef sparc
    /* some day ... */
#endif
#ifdef mips
    /* a day or two later ... */
#endif

#ifdef FORTRAN
    (*sp->handler)(&sp->sig, &sp->code, NULL);
#else
#ifdef PASCAL
    /* Args in reverse order in pascal, context not documented at all */
    (*sp->handler)(sp->code, sp->sig);
#else
    (*sp->handler)(sp->sig, sp->code, NULL);
#endif PASCAL
#endif FORTRAN

    /* we have done with the signal handler */
    /* We don't support the notion of "sigcontext" */

    THREAD_PROTECT;
    THREADlock(&THREADcurrent->context.lock);
    while (family_ndchildcount(THREADcurrent) > 0) {
        THREAD tp;

        /* do not proceed until all nondetached children disappear */
        while((tp = family_exitingchildren(THREADcurrent)) != NULL) {
            /* handle any that have terminated but haven't been picked up yet*/
            family_ndchildcount(THREADcurrent)--;
            THREADunlock(&THREADcurrent->context.lock);
            THREAD_TYPE1_THREAD_TERMINATE(tp);
            THREADactivedecr;
            THREADlock(&THREADcurrent->context.lock);
            THREADlock(&tp->context.lock);
            family_exitingchildren(THREADcurrent) = family_sibling(tp);
            /*leave the thread locked - one else should be touching it now*/
            THREADdestroy(tp);
        }
        if (family_ndchildcount(THREADcurrent) <= 0) {
            /* all done */
            break;
        }
        /* set our status as waiting */
        THREAD_WAITCHILD_SET(THREADcurrent);
        /* put thread on waitq to maintain "always on queue" invariant */
        THREADlock(&(waitq.lock));
        THREAD_TCBAPPEND(THREADcurrent, &waitq);
        THREADunlock(&(waitq.lock));
        /* child will make us runnable */
        THREAD_PASSPROTECT;
        THREADlock(&THREADrunqlock);
        THREADrun();
        THREAD_PROTECT;
        THREADlock(&THREADcurrent->context.lock);
    }

    THREADunlock(&THREADcurrent->context.lock);
    THREAD_TYPE1_THREAD_TERMINATE(THREADcurrent);
    THREADactivedecr;
    THREAD_PASSPROTECT;
    THREADunfreeze(THREAD_FROZEN_PARENT(THREADcurrent));
    logerr(DISASTER, "THREADfsiginit: child thawing parent didn't vanish");
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#include <syscall.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include "privatedata.h"
#include "io.h"
#include "mp.h"
#include "logerr.h"

/*****************************************************************************/
/* export global variables                                                   */
/*****************************************************************************/

LOCK THREADiotimer = UNLOCKED;
struct timeval THREADnowait;
int sfd[2];


/*****************************************************************************/
/* static global variables & internal used macros                            */
/*****************************************************************************/

static LOCK sysio_sync = UNLOCKED; /* serve as a sync. for system io ops. */
static LOCK sysio_sync2 = UNLOCKED;
static int signal_recv_count = 1;  /* how many filetables being changed */
static int sys_fd[2];              /* file ids to be used in handler */
static char sys_filename[108];     /* filename to be used in handler */
static int sys_flags;              /* mode to be used in handler */
static int filetable_op;           /* what kind of op. just executed */


/*****************************************************************************/
/* THREADsysiohandler                                                        */
/*                                                                           */
/* When the filetable need to be updated, the process will receive this      */
/* signal - SIGUSR1                                                          */
/*****************************************************************************/

void
THREADsysiohandler (int sig)
{
    switch (filetable_op) {
      case SYS_close:
        close(sys_fd[0]);
        break;
      case SYS_fork:
        dup(sys_fd[0]);
        break;
      case SYS_dup2:
        dup2(sys_fd[0], sys_fd[1]);
        break;
      case SYS_accept:
      case SYS_socket:
      case SYS_open:
        {
            /* we must use recvmsg to make sure pointing to the same entry in
               system filetable */

            struct msghdr msg;
            int fd;

            msg.msg_iov          = (struct iovec*) 0;
            msg.msg_iovlen       = 0;
            msg.msg_name         = (caddr_t) 0;
            msg.msg_accrights    = (caddr_t) &fd;
            msg.msg_accrightslen = sizeof(int);

          eintr_label:
            if (recvmsg(sfd[1], &msg, 0) < 0){
                char ss[100];
                if (errno == EINTR)
                    goto eintr_label;
                sprintf(ss, "THREADsysiohandler: recvmsg error # %d", errno);
                logerr(ERROR, ss);
                break;
            }
        }
        break;
    }

    THREADlock(&sysio_sync2);
    signal_recv_count++;
    if (signal_recv_count == proc_num) {
        signal_recv_count = 1;
        THREADunlock(&sysio_sync);
    }
    THREADunlock(&sysio_sync2);
}


/*****************************************************************************/
/* THREADsyssocket                                                           */
/*                                                                           */
/*****************************************************************************/


int
THREADsyssocket (int domain, int type, int protocol)
{
    struct msghdr msg;
    int pid, i;
    int sockfd;

    THREADlock(&sysio_sync);
    signal_recv_count = 1;

  socket_label:
    sockfd = socket(domain, type, protocol);
    if (sockfd < 0) {
        if (errno == EINTR) /* watch for *signal* interrupt error */
            goto socket_label;
        return (sockfd);
    }

    msg.msg_iov          = (struct iovec *) 0;
    msg.msg_iovlen       = 0;
    msg.msg_name         = (caddr_t) 0;
    msg.msg_accrights    = (caddr_t) &sockfd;;
```

```
    msg.msg_accrightslen = sizeof(int);
    filetable_op = SYS_socket;

    for (i = 0; i < proc_num; i++) {
        if (i == THREADn)
            continue;
        if ((sendmsg(sfd[0], &msg, 0)) < 0) {
            perror("in open; sendmsg");
        }
        kill(THREADnpid(i), SIGUSR1);
    }

    if (proc_num == 1)
        THREADunlock(&sysio_sync);

    return (sockfd);
}




/*****************************************************************************/
/* THREADsysopen                                                             */
/*                                                                           */
/*****************************************************************************/


int
THREADsysopen (char *filename, int flags, int mode)
{
    struct msghdr msg;
    int pid, i;
    int fd;

    THREADlock(&sysio_sync);
    signal_recv_count = 1;

  open_label:
    if (flags & O_CREAT)
        fd = open(filename, flags, mode);
    else
        fd = open(filename, flags);
    if (fd < 0) {
        if (errno == EINTR) /* watch for *signal* interrupt error */
            goto open_label;
        return (fd);
    }

    msg.msg_iov          = (struct iovec *) 0;
    msg.msg_iovlen       = 0;
    msg.msg_name         = (caddr_t) 0;
    msg.msg_accrights    = (caddr_t) &fd;;
    msg.msg_accrightslen = sizeof(int);
    filetable_op = SYS_open;

    for (i = 0; i < proc_num; i++) {
        if (i == THREADn)
            continue;
        if ((sendmsg(sfd[0], &msg, 0)) < 0) {
            perror("in open; sendmsg");
        }
        kill(THREADnpid(i), SIGUSR1);
    }

    if (proc_num == 1)
        THREADunlock(&sysio_sync);

    return (fd);
}




/*****************************************************************************/
/* THREADsysaccept                                                           */
/*                                                                           */
/*****************************************************************************/


int
THREADsysaccept (int s, struct sockaddr *addr, int *addrlen)
{
    struct msghdr msg;
    int pid, i;
    int fd;

    THREADlock(&sysio_sync);
    signal_recv_count = 1;

  accept_label:
    fd = accept (s, addr, addrlen);
    if (fd < 0) {
        if (errno == EINTR) /* watch for *signal* interrupt error */
            goto accept_label;
        return (fd);
    }

    msg.msg_iov          = (struct iovec *) 0;
    msg.msg_iovlen       = 0;
    msg.msg_name         = (caddr_t) 0;
    msg.msg_accrights    = (caddr_t) &fd;;
    msg.msg_accrightslen = sizeof(int);
    filetable_op = SYS_accept;

    for (i = 0; i < proc_num; i++) {
        if (i == THREADn)
            continue;
        if ((sendmsg(sfd[0], &msg, 0)) < 0) {
            perror("in open; sendmsg");
        }
        kill(THREADnpid(i), SIGUSR1);
    }

    if (proc_num == 1)
        THREADunlock(&sysio_sync);

    return (fd);
}




/*****************************************************************************/
/* THREADsysclose                                                            */
/*                                                                           */
```

```
/*****************************************************************************/


int
THREADsysclose (int fd)
{
    int pid, i;

    THREADlock(&sysio_sync);
    signal_recv_count = 1;

  close_label:
    if ((close(fd)) == -1) {
        if (errno == EINTR)
            goto close_label;
        return -1;
    }

    sys_fd[0] = fd;
    filetable_op = SYS_close;

    for (i = 0; i < proc_num; i++) {
        if (i == THREADn)
            continue;
        kill(THREADnpid(i), SIGUSR1);
    }

    if (proc_num == 1)
        THREADunlock(&sysio_sync);

    return (0);    /* sucessfully */
}



/*****************************************************************************/
/* THREADsysdup                                                              */
/*                                                                           */
/*****************************************************************************/


int
THREADsysdup (int fd)
{
    int pid, i, new_fd;

    THREADlock(&sysio_sync);
    signal_recv_count = 1;

  dup_label:
    if (((new_fd = dup(fd))) == -1) {
        if (errno == EINTR)
            goto dup_label;
        return -1;
    }

    sys_fd[0] = fd;
    filetable_op = SYS_dup;

    for (i = 0; i < proc_num; i++) {
        if (i == THREADn)
            continue;
        kill(THREADnpid(i), SIGUSR1);
    }

    if (proc_num == 1)
        THREADunlock(&sysio_sync);

    return (new_fd);    /* sucessfully */
}


/*****************************************************************************/
/* THREADsysdup2                                                             */
/*                                                                           */
/*****************************************************************************/


int
THREADsysdup2 (int fd1, int fd2)
{
    int pid, i, new_fd;

    THREADlock(&sysio_sync);
    signal_recv_count = 1;

  dup2_label:
    if (((new_fd = dup2(fd1, fd2))) == -1) {
        if (errno == EINTR)
            goto dup2_label;
        return -1;
    }

    sys_fd[0] = fd1;
    sys_fd[1] = fd2;
    filetable_op = SYS_dup2;

    for (i = 0; i < proc_num; i++) {
        if (i == THREADn)
            continue;
        kill(THREADnpid(i), SIGUSR1);
    }

    if (proc_num == 1)
        THREADunlock(&sysio_sync);

    return (new_fd);    /* sucessfully */
}



/*****************************************************************************/
/* THREADcopyfdset                                                           */
/*****************************************************************************/


void
THREADcopyfdset (fd_set *source, fd_set *target)
{
    *target = *source;
}


/*****************************************************************************/
/* THREADcopyfdsetlimited                                                    */
/*                                                                           */
/* copies limit bytes from *source to *target                                */
```

```
/*****************************************************************************/


void
THREADcopyfdsetlimited (fd_set *source, fd_set *target, int limit)
{
    int i;
    fd_mask *s = &(source)->fds_bits[0];
    fd_mask *t = &(target)->fds_bits[0];

    for (i = limit; i >= NFDBITS; i -= NFDBITS) { /* NFDBITS : bits per mask */
        *(t++) = *(s++);
    }
    if (i > 0) {
        fd_mask mask = (1<<i) - 1;

        *t = (*t & ~mask) | (mask & *s);
    }
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

/*****************************************************************************/
/* Programmer : Tsugn-Jen Chou                                               */
/*                                                                           */
/* This program is modified from the work done by                            */
/* David Edelsohn (edelsohn@sccs.Syr.EDU)                                    */
/*                                                                           */
/* This program                                                              */
/* (1) asks a shared memory (starting at the same place for the same kind    */
/*     of machine).                                                          */
/* (2) read in the to-be-loaded program to the starting address.             */
/* (3) pass the control flow to that starting address.                       */
/*                                                                           */
/* acc this file with -Bstatic  ...                                          */
/*    acc -Bstatic loadfunc wrapper.c sharem.c -o loadfunc                   */
/* -Bstatic  : because we want to load another executable onto our data sec.*/
/*             so we could only use -Bstatic                                 */
/* wrapper.c : because the to-be-loaded file will access loadfunc            */
/*             ( -A loadfunc) so wrapper can't appear in the following line.*/
/*             this file wraps the normal system io function.                */
/* to-be-loaded program should be linked with                                */
/*   /bin/ld -N -x -T f76f0000 -A loadfunc -e _starter (o files) (libs)      */
/*                                                                           */
/* current support Sun models : SPARCsystem 1, SPARCsystem 1+,               */
/* SPARCsystem 2, SPARCsystem IPC, SPARCsystem SL, SPARCsystem 600MP series,*/
/* SPARCsystem 10                                                            */
/*****************************************************************************/

#include <stdio.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <a.out.h>

#define Chunk 1040000
#define round(x,s) ((((x) -1) & ~ ((s) - 1)) + (s))

#include "loadfunc_only.h"
extern int errno;

/*****************************************************************************/
/* shmdemand --  *id <- shmget (key, size, flag|IPC_CREAT)                   */
/*               return (shmat(*id, addr, flag))                             */
/*                                                                           */
/* need to take care of shmget's error, so we call shmdemand instead of      */
/* call shmget directly                                                      */
/*****************************************************************************/


char*
shmdemand (int size, int *id)
{
#define flag 0666

    long shmaddr;
    struct shmid_ds buf;

  try_again:
    if ((*id = shmget(((key_t) rand()), size, flag|IPC_CREAT|IPC_EXCL)) < 0) {
        if (errno == EEXIST) {
            goto try_again;  /* we don't want interfere with our business */
        }

        if (errno == ENOSPC) {
            char buf[108];
            sprintf(buf, "No free shared memory available\nUsing ipcs and\
ipcrm to deal with\n");
            write(2, buf, strlen(buf));
            return NULL;
        }

        if (errno == EINVAL) {
            perror("shmget");
            return NULL;
        }
        else {
            perror("shmget");
            return NULL;
        }
    }


    if ((shmaddr = shmat(*id, 0, flag)) == -1) {
        perror("shmat");
        return NULL;
    }

    return (char*) shmaddr;
#undef flag
}



#define Key ((key_t) getpid())
#define Perm 0666

/* ****************************************************************
 *  Current supporting Architectures
 *
 *  Application       Kernel              Current Sun
 *  Architecture   Architecture           System Models
 *  sun4           sun4c          SPARCsystem 1, SPARCsystem 1+,
 *                                SPARCsystem 2, SPARCsystem IPC,
 *                                SPARCsystem SLC
 *  sun4           sun4m          SPARCsystem 600MP series,
 *                                SPARCsystem 10
 *
 * entry address
 * sparc 4c : 0xf76f0000
 * sparc 4m : 0xef6ff000
 * ****************************************************************/

#define sun4mEntryAddress 0xef6ff000
#define sun4cEntryAddress 0xf76f0000



/*****************************************************************************/
/* loadfunc                                                                  */
```

```
/*****************************************************************************/

void
loadfunc (char *filename, int ac, char *av[])
{
    int fd;
    caddr_t base;
    u_long readsize;
    u_long offset;
    int shmid;
    struct exec header;
    struct shmid_ds buf;
    int (*entry)(int, char*[]);

    /* open the file */
    if ((fd = open(filename, O_RDONLY, 0)) < 0) {
        perror("in loadfunc, open");
        exit(errno);
    }

    /* ********************************************************************
     * allocate the memory to load the executable.
     * we don't want to call shmget & shmat directly because we must take
     * care of some unexpected situations
     * ********************************************************************/
    if ((base = shmdemand(Chunk, &shmid)) == NULL) {
        printf("error in share memory allocation\n");
        exit(errno);
    }
#ifdef DEBUG
    printf("base is %x\n", base);
#endif

    /* ************************************************************
     * start to read in this file. header information first.
     * ************************************************************/
    if ((read(fd, &header, sizeof(struct exec))) != sizeof(struct exec)) {
        printf("couldn't read header from %s", filename);
        close(fd);
        exit(errno);
    }
    readsize = round(header.a_text, 4) + round(header.a_data, 4);

#ifdef DEBUG
    printf("text=0x%x,  data=0x%x,  bss=0x%x\n", header.a_text, header.a_data,
           header.a_bss);
    printf("readsize=0x%x\t%d\n", readsize, readsize);
#endif

    /* ************************************************************
     * read text + data from the executable, then close this file
     * ************************************************************ */
    offset = N_TXTOFF(header);
#ifdef DEBUG
    printf("actual data, i.e. N_TXTOFF is %x  %d\n", offset, offset);
#endif
    if ((lseek(fd, offset, 0)) != offset) {
        close(fd);
        printf("couldn't position file %s\n", filename);
        shmctl(shmid, IPC_RMID, &buf);
        exit(5);
    }
    if (read(fd, base, readsize) != readsize) {
        close(fd);
        printf("couldn't read data from %s\n", filename);
        shmctl(shmid, IPC_RMID, &buf);
        exit(6);
    }
    close(fd);

    bzero(base+readsize, header.a_bss); /* clear bss section */

    /* ******************************
     * jump to the loaded executable.
     * ******************************/
    entry = (int (*)(int, char *[])) header.a_entry;
    (*entry)(ac, av);  /* pass control */


    /* ************************************
     * came back from the loaded executable.
     * ************************************/
    if ((shmctl(shmid, IPC_RMID, &buf)) < 0) /* delete the share memory */
        perror("shmctl");
}



/*****************************************************************************/
/* main                                                                      */
/*****************************************************************************/
int
main (int argc, char *argv[])
{
    if (argc < 2) {
        printf("usage: %s to-be-loaded-file\n", argv[0]);
        exit(1);
    }

    srand(getpid());
    loadfunc(argv[1], argc-1, &(argv[1]));
}
```

```
/*****************************************************************************/
/*      thread - thread package.                                             */
/*                                                                           */
/*      Copyright 1986 Thomas W. Doeppner Jr. - Brown University             */
/*      - All rights reserved.                                               */
/*****************************************************************************/

#include <signal.h>
#include "privatedata.h"
#include "tcb.h"
#include "logerr.h"
#include "thread.h"
#include "preempt.h"

static int THREADerrorlevel = WARNING;
LOCK THREADdying = UNLOCKED;

/*****************************************************************************/
/* logerr                                                                    */
/*****************************************************************************/

void
logerr (int level, char *message)
{
    char buf[150];

    if (level < THREADerrorlevel)
        return;

    THREAD_PROTECT;

    switch(level) {
      case INFO:
        sprintf(buf, "\nTHREAD (info): %s\n", message);
        write(2, buf, strlen(buf));
        break;

      case WARNING:
        sprintf(buf, "\nTHREAD (warning): %s\n", message);
        write(2, buf, strlen(buf));
        break;

      case ERROR:
        sprintf(buf, "\nTHREAD (error): %s\n", message);
        write(2, buf, strlen(buf));
        break;

      case DISASTER:
      case TERMINATE:
        {
            int omask = sigblock(0xffffffff);
            /* give default handling to the following */
            signal(SIGHUP, SIG_DFL);
            signal(SIGINT, SIG_DFL);
            signal(SIGQUIT, SIG_DFL);
            signal(SIGILL, SIG_DFL);
            signal(SIGTRAP, SIG_DFL);
            signal(SIGIOT, SIG_DFL);
            signal(SIGEMT, SIG_DFL);
            signal(SIGFPE, SIG_DFL);
            signal(SIGBUS, SIG_DFL);
            signal(SIGSEGV, SIG_DFL);
            signal(SIGSYS, SIG_DFL);
            signal(SIGPIPE, SIG_DFL);
            signal(SIGTERM, SIG_DFL);
            if (!THREADcondlock(&THREADdying)) {
                write(2, "ABORT: ", 7);
                sprintf(buf, "%d aborting (pid %d)(disaster), THREADn(%d)\n",
                        THREADpindex,
                        getpid(), THREADn);
                write(2, buf, strlen(buf));
                _exit(1);
            }

            if (level == DISASTER)
                write(2, "\nTHREAD(disaster): ", 19);
            else
                write(2, "\nTHREAD(terminate): ", 20);
            sprintf(buf, "pid %d (%d): %s\n", THREADwhichprocessor,
                    THREADpindex, message);
            write(2, buf, strlen(buf));

            if (level == DISASTER)
                THREADsigbroadcast(SIGQUIT);
            else {
                int i;

                THREADsigbroadcast(31);

                /* give everyone a chance to do something, then blast 'em */
                for (i=0; i<1000000; i++)
                    ;
                THREADsigbroadcast(SIGKILL);
                for (i=0; i<1000000; i++)
                    ;
            }

            if (level == DISASTER) {
                kill(getpid(), SIGQUIT);
                sigsetmask(~(1<<(SIGQUIT-1)));
                write(2, "abort failed\n", 13);
            }
            exit(2);
        }
      default:
        logerr(ERROR, "LOGERR: Invalid message level");
    }

    THREAD_UNPROTECT;
}


/*****************************************************************************/
/* THREADset_error_level                                                     */
/*****************************************************************************/


void
THREADset_error_level (int level)
{
    if ((level < INFO) || (level > DISASTER))
        logerr(WARNING, "THREADset_error_level: invalid level");
    else
        THREADerrorlevel = level;
}
```

```
/*************************************************************************/
/* Build a dynamic memory allocation (malloc, free,.....)                */
/*                                                                       */
/* This program is modifed from the malloc.c in cs169's assignment. This */
/* version of malloc & free is dedicated to be used with THREAD package  */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/errno.h>
#include "lock.h"
#include "tcb.h"
#include "runQ.h"
#include "preempt.h"

extern int errno;

#define SHMSIZE (1024 * 1024 - 100)



/*************************************************************************/
/* below for dynamic memory management                                   */
/*************************************************************************/

struct BTag {
  int size;
  struct BTag *blink, *flink;
};

struct BTag avail;
int *freepool;
LOCK memlock;



/*************************************************************************/
/* malloc_init                                                           */
/*                                                                       */
/* return 0 : fail; return 1 : success                                   */
/*************************************************************************/

int
malloc_init()
{
    int sid;
    extern char* shmdemand(int, int*);

    if ((freepool = (int*) shmdemand(SHMSIZE, &sid))==NULL) {
        printf("in malloc_init error\n");
        fprintf(stderr, " error in dynamic memory initialization\n");
        exit (errno);
    }

    freepool[0] = SHMSIZE;
    freepool[SHMSIZE / 4 - 1] = SHMSIZE;

    avail.flink = avail.blink = (struct BTag*) (&freepool[1]);
    avail.flink->blink = avail.flink->flink = &avail;
    avail.flink->size = 2 * sizeof(int) - SHMSIZE;
    freepool[SHMSIZE / 4 - 2] = 2 * sizeof(int) - SHMSIZE;
    memlock = UNLOCKED;
    return sid;
}



/*************************************************************************/
/* _malloc -- allocate a memory of (((size+3)/4)*4)                      */
/*                                                                       */
/* We free the tcb on freeit_queue. DON'T try to memlock should be unlocked*/
/* before invoking THREAD_FREESTACK, because it will invoke _free in turn. */
/*************************************************************************/

char*
_malloc (int size)
{
    struct BTag *t0;
    THREAD tcbp;

    THREADlock(&freeit_queue.lock);
    while ((tcbp = THREADqueueGet(&freeit_queue)) != NULL)
        THREAD_FREESTACK(tcbp);
    THREADunlock(&freeit_queue.lock);

    THREAD_PROTECT;
    THREADlock(&memlock);

    if (freepool == 0) {
        THREADunlock(&memlock);
        THREAD_UNPROTECT;
        return (char*) 0;
    }

    if (size % 4)                       /* alignment to 4 */
        size += (4 - (size % 4));
    if (size < 2 * sizeof(char*))       /* must make room for two links */
        size = 2 * sizeof (char*);

    t0 = avail.flink;
    while (t0 != &avail) {
        if ((int)(t0->size + 2 * sizeof(int) + size) <= 0)
            goto found;
        t0 = t0->flink;
    }

    THREADunlock(&memlock);
    THREAD_UNPROTECT;
    return (char*)0;

  found:
    if ((int)(t0->size + 3 * sizeof(int) + size + sizeof(struct BTag*)) > 0){
        /*
          the free space is not big enough to support, allocate all this free
          space to allocated area
         */
        t0->size = -t0->size;
        ((int*)((char*)t0 + t0->size - sizeof(int)))[0] = t0->size;
        t0->blink->flink = t0->flink;
        t0->flink->blink = t0->blink;
        bzero((char*)((char*)t0 + sizeof(int)), size);

        THREADunlock(&memlock);
        THREAD_UNPROTECT;
```

```
        return (char*)((char*)t0 + sizeof(int));
    }

    /*
      below divide the free space into two parts, one is allocated, the other
      one still linked to freepool list with size changed
     */
    ((int*)((char*)t0 - t0->size - sizeof(int)))[0] = size + sizeof(int) * 2;
    t0->size = t0->size + size + 2 * sizeof(int);
    ((int*)((char*)t0 - t0->size - sizeof(int)))[0] = t0->size;
    ((int*)((char*)t0 - t0->size))[0] = size + sizeof(int) * 2;
    bzero((char*)((char*)t0 - t0->size + sizeof(int)), size);

    THREADunlock(&memlock);
    THREAD_UNPROTECT;
    return (char*)((char*)t0 - t0->size + sizeof(int));
}



/*****************************************************************************/
/* _free                                                                     */
/*****************************************************************************/

int
_free (struct BTag *b)
{
  struct BTag *t0, *t1, *t2;

  THREAD_PROTECT;
  THREADlock(&memlock);

#define PREV(x) (((int*)(x))[-1])

  b = (struct BTag*) &PREV(b);
  b->size = -b->size;

  if (PREV(b) < 0) {
    t0 = (struct BTag*)((char*)b + PREV(b));
    t1 = t0->flink;
    t2 = t0->blink;
    t1->blink = t2;
    t2->flink = t1;
    t0->size += b->size;
    b = t0;
  }
  t0 = (struct BTag*)((char*)b - b->size);
  if (t0->size < 0){
    t1 = t0->flink;
    t2 = t0->blink;
    t1->blink = t2;
    t2->flink = t1;
    b->size += t0->size;
    t0 = (struct BTag*)((char*)t0 - t0->size);
  }
  PREV(t0) = b->size;
  b->flink = avail.flink;
  b->blink = &avail;
  avail.flink->blink = b;
  avail.flink = b;

  THREADunlock(&memlock);
  THREAD_UNPROTECT;
  return 1;
}



/*****************************************************************************/
/* memalign                                                                  */
/*****************************************************************************/


char *
_memalign (unsigned alignment, unsigned size)
{
    char *p;

    p = _malloc(size);

    if (p == NULL)
        return NULL;

    if (((unsigned) p) % alignment) {
        char buf[108];
        sprintf(buf, "Error in memalign, not aligned\n");
        write(2, buf, strlen(buf));

        return NULL;
    } else
        return p;
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#include <stdio.h>
#include "tcb.h"
#include "manager.h"
#include "preempt.h"
#include "monitor.h"

#define manager_ops_num 2


int THREADmanager_offset; /* assigned in mp.c */

static
struct manager_ops monitor_ops[manager_ops_num] = {
    MANAGER_ABORT, THREADmonitorabort,
    MANAGER_EXCEPTION, THREADmonitorabort,
};



/*****************************************************************************/
/* THREADtellmanagement                                                      */
/*                                                                           */
/* when a thread is switched to run, it must check if it got exceptions      */
/* or not, if so it calls the corresponding function                         */
/*****************************************************************************/

void
THREADtellmanagement (int func_code, THREAD tcb)
{
    THREAD_MANAGER manager = (THREAD_MANAGER) manager_newest(tcb);
    int i;
    void (*func)();

    THREAD_PROTECT;

    switch (func_code) {
      case MANAGER_ABORT:
        while (manager != NULL) {
            /* try each manager in the manager stack */
            for (i = 0; i < manager->num_of_ops; i++) {
                /* check out the id's for a match */
                if ((func_code == manager->ops[i].id) &&
                    ((func = manager->ops[i].func)) != NULL) {
                    (*func)(tcb, manager);
                }
            }
            manager = manager_forward(manager);
            MANAGER_RELEASE(tcb);
        }
        break;
      case MANAGER_EXCEPTION:
        while ((manager != NULL) && MANAGER_IN_SCOPE(manager, tcb)) {
            /* try each manager that was created farther in the stack
               than where setexception was called */
            for (i = 0; i < manager->num_of_ops; i++) {
                /* check out the id's for a match */
                if ((func_code == manager->ops[i].id) &&
                    ((func = manager->ops[i].func)) != NULL) {
                    (*func)(tcb, manager);
                }
            }
            manager = manager_forward(manager);
            MANAGER_RELEASE(tcb);
        }
        break;
    }
    THREAD_UNPROTECT;
    return;
}



/*****************************************************************************/
/* MANAGER_ALLOCATE                                                          */
/*                                                                           */
/*****************************************************************************/

void
MANAGER_ALLOCATE (THREAD_MANAGER *manager, THREAD_MONITOR monitor, THREAD tcb)
{
    int size;

    size = sizeof(THREAD_MANAGER_BLOCK) +
        ((manager_ops_num - 2) * sizeof(struct manager_ops));
    if (*manager == NULL) {
        *manager = (THREAD_MANAGER) _malloc(size);
        bzero(*manager, size);
    } else {
        bzero(*manager, size);
        manager_onstack_set(*manager); /* don't interchange these two lines */
    }

    (*manager)->data = monitor;
    (*manager)->num_of_ops = manager_ops_num;
    bcopy(monitor_ops, (*manager)->ops,
          manager_ops_num * sizeof(struct manager_ops));

    THREADmanagerexcpptr((*manager)) = THREADexception_stack(THREADcurrent);

    /* chain the manager together, in the case of nest monitor */
    manager_forward((*manager)) = manager_newest(THREADcurrent);
    manager_newest(THREADcurrent) = (*manager);
}



/*****************************************************************************/
/* MANAGER_RELEASE                                                           */
/*                                                                           */
/* N.B. if manager_onstack is true, this manager is supported by the user,   */
/* so don't free it                                                          */
/*****************************************************************************/


void
```

```
MANAGER_RELEASE (THREAD tcb)
{
    THREAD_MANAGER manager = manager_newest(tcb);

    manager_newest(tcb) = manager_forward(manager);
    if (!manager_onstack(manager)) {
        _free(manager);
    }
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include "monitor.h"
#include "malloc.h"
#include "manager.h"
#include "privatedata.h"
#include "mp.h"
#include "queue.h"
#include "preempt.h"
#include "io.h"
#include "logerr.h"



/*****************************************************************************/
/* THREADmonitorCheckOK                                                      */
/*                                                                           */
/* this function determines if monitor & its condition appropriate or not    */
/*****************************************************************************/

static
int
THREADmonitorCheckOK (THREAD_MONITOR monitor, int condition, char *funcname)
{
    char s[108];

    if ((condition >= monitor->nr_of_conditions) || (condition < 0)) {
        sprintf(s, "%s: called with out-of-range condition", funcname);
        logerr(ERROR, s);
        return(0);
    }

    if (monitor_thread(monitor) != THREADcurrent) {
        /* not our monitor! */
        sprintf(s, "%s: you don't own this monitor", funcname);
        logerr(ERROR, s);
        return(0);
    }

    return 1;
}



/*****************************************************************************/
/* THREADmonitorinit                                                         */
/*                                                                           */
/* allocate and initialize the monitor                                       */
/*****************************************************************************/

THREAD_MONITOR
THREADmonitorinit (int conditions, void (*resetfunc)())
{
    THREAD_MONITOR monitor;
    THREAD_MANAGER manager;

    /* allocate enough space for the monitor structure and
       the condition structures */

    monitor = (THREAD_MONITOR)_malloc(sizeof(*monitor) + (conditions-1) *
                                      sizeof(THREAD_MONITOR_CONDITION_HEAD));

    if (monitor == NULL) {
        logerr(ERROR, "THREADmonitorinit: out of memory");
        return(NULL);
    }

    /* initialize the monitor data structure */
    THREADqueueInit(&(monitor->queue));
    monitor_nr_of_conditions(monitor) = conditions;
    monitor_reset(monitor) = resetfunc;
    monitor_clear_active(monitor);

    if (conditions > 0) {
        /* initialize the conditions */
        bzero(&monitor_condition(monitor, 0),
              conditions * sizeof(THREAD_QUEUE_HEAD));
    }

    return(monitor);
}



/*****************************************************************************/
/* THREADmonitorfree                                                         */
/*****************************************************************************/

void
THREADmonitorfree (THREAD_MONITOR monitor)
{
    _free(monitor);
}



/*****************************************************************************/
/* THREADmonitorentry                                                        */
/*****************************************************************************/

int
THREADmonitorentry (THREAD_MONITOR monitor, THREAD_MANAGER manager)
{
    MANAGER_ALLOCATE(&manager, monitor, THREADcurrent);

    THREAD_PROTECT

    THREADlock(&monitor->queue.lock);
    THREADlock(&((THREADcurrent)->context.lock));

    /* ask for permission to enter */
    if  (monitor_active(monitor)) {       /* does anyone hold this */
        /* permission denied, so we wait */
        if (monitor_thread(monitor) == THREADcurrent) {
            /* we already hold this monitor */

            THREADunlock(&((THREADcurrent)->context.lock));
            THREADunlock(&monitor->queue.lock);
            logerr(ERROR, "THREADmonitorentry: deadlock!");
            THREAD_UNPROTECT;
```

```
            return(0);
        }
        THREAD_TCBAPPEND(THREADcurrent, &(monitor->queue));
        THREADunlock(&monitor->queue.lock);
        THREAD_PASSPROTECT;
        THREADlock(&THREADrunqlock);

        THREADrun();
        /* it will only be waken up when other thread leave this monitor */
    } else {
        /* permission granted */

        monitor_set_active(monitor, THREADcurrent);
        THREADunlock(&((THREADcurrent)->context.lock));
        THREADunlock(&monitor->queue.lock);
        THREAD_UNPROTECT;

    }
    return(1);
}



/*****************************************************************************/
/* THREADmonitorexit                                                         */
/*                                                                           */
/* THREADcurrent will leave the monitor, wake up the first candidate, if     */
/* any.                                                                      */
/*****************************************************************************/

int
THREADmonitorexit (THREAD_MONITOR monitor)
{
    THREAD tcb;

    if (monitor_thread(monitor) != THREADcurrent) {
        /* not our monitor! */
        char s[100];
        sprintf(s, "monitorexit: threadcurrent is %x\n", THREADcurrent);
        write(2, s, strlen(s));

        logerr(ERROR, "THREADmonitorexit: you don't own this monitor");
        return(0);
    }

    THREAD_PROTECT;
    THREADlock(&monitor->queue.lock);

    if ((tcb = THREADqueueGet(&(monitor->queue))) != NULL) {
        /* someone else wants in from the entry queue */
        THREADlock(&tcb->context.lock);
        monitor_set_active(monitor, tcb);
        THREADlock(&THREADrunqlock);
        THREAD_MOVETORUNQ(tcb);
        THREADunlock(&THREADrunqlock);
        THREADunlock(&tcb->context.lock);
    } else {
        /* no one wants in */
        monitor_clear_active(monitor);
    }

    THREADunlock(&monitor->queue.lock);
    THREAD_UNPROTECT;

    MANAGER_RELEASE(THREADcurrent);

    return(1);
}



/*****************************************************************************/
/* THREADmonitorwait                                                         */
/*                                                                           */
/* The current thread will append to Queue condition and sleep               */
/*****************************************************************************/

int
THREADmonitorwait (THREAD_MONITOR monitor, int condition)
{
    THREAD tcb;

    if (!(THREADmonitorCheckOK(monitor, condition, "THREADmonitorwait")))
        return 0;

    THREAD_PROTECT;
    THREADlock(&monitor->queue.lock);

    if ((tcb = THREADqueueGet(&(monitor->queue))) != NULL) {
        /* someone else wants in from the entry queue */
        THREADlock(&tcb->context.lock);
        monitor_set_active(monitor, tcb);
        THREADlock(&THREADrunqlock);
        THREAD_MOVETORUNQ(tcb);
        THREADunlock(&THREADrunqlock);
        THREADunlock(&(tcb->context.lock));
    } else {
        /* no one is waiting, so monitor becomes inactive */
        monitor_clear_active(monitor);
    }

    THREADlock(&((THREADcurrent)->context.lock));
    THREADlock(&((monitor->conditions[condition]).queue.lock));
    THREAD_TCBAPPEND(THREADcurrent, &((monitor->conditions[condition]).queue));
    THREADunlock(&((monitor->conditions[condition]).queue.lock));
    THREADunlock(&monitor->queue.lock);

    THREAD_PASSPROTECT;

    THREADlock(&THREADrunqlock);
    THREADrun();

    /* when we get here, we must have just been signalled */
    return(1);
}



/*****************************************************************************/
/* THREADmonitorsignalandexit                                                */
/*                                                                           */
/* leave the monitor and signal the condition queue; if no candidate there   */
/* find somebody else on the standard queue                                  */
/*****************************************************************************/

int
THREADmonitorsignalandexit (THREAD_MONITOR monitor, int condition)
{
```

```
#define condq ((monitor->conditions[condition]).queue)

    THREAD tcb;

    if (!THREADmonitorCheckOK(monitor,condition,"THREADmonitorsignalandexit"))
        return 0;

    THREAD_PROTECT;
    THREADlock(&monitor->queue.lock);

    /* first check to see if there is anyone to signal */
    THREADlock(&(condq.lock));
    if ((tcb = THREADqueueGet(&(condq))) == NULL) {
        /* no one is need to signal */
        THREADunlock(&(condq.lock));

        if ((tcb = THREADqueueGet(&(monitor->queue))) != NULL) {
            /* someone else wants in from the entry queue */
            THREADlock(&tcb->context.lock);
            monitor_set_active(monitor, tcb);
            THREADlock(&THREADrunqlock);
            THREAD_MOVETORUNQ(tcb);
            THREADunlock(&THREADrunqlock);
            THREADunlock(&(tcb->context.lock));
        } else {
            /* no one is waiting, so monitor becomes inactive */
            monitor_clear_active(monitor);
        }

        THREADunlock(&monitor->queue.lock);
        THREAD_UNPROTECT;

        MANAGER_RELEASE(THREADcurrent);

        return(1);
    }

    /* tcb is the thread waiting on the condition queue */
    if (THREAD_IN_WAIT(tcb)) {
        /* ************************************************************
           tcb is blocking in wait and occupying some other process.
           THREADfirstpipe+THREAD_PORDER(tcb) is the *magic* pipe
           exclusive for the process it is on.
           we write a letter to it and wake it up from select call

           N.B. don't put tcb onto runq. it's already running.
           ************************************************************/
        write(THREADfirstpipe+THREAD_PORDER(tcb), "a", 1);
        monitor_set_active(monitor, tcb);
        THREADunlock(&(condq.lock));
        THREADunlock(&monitor->queue.lock);

        THREAD_UNPROTECT;

        MANAGER_RELEASE(THREADcurrent);
        return(2); /* return indication that we did something */
    }


    THREADlock(&(tcb->context.lock));
    monitor_set_active(monitor, tcb);
    THREADlock(&THREADrunqlock);
    THREAD_MOVETORUNQ(tcb);
    THREADunlock(&THREADrunqlock);
    THREADunlock(&(tcb->context.lock));
    THREADunlock(&(condq.lock));
    THREADunlock(&monitor->queue.lock);

    THREAD_UNPROTECT;

    MANAGER_RELEASE(THREADcurrent);

    return(2); /* return indication that we did something */

#undef condq
}



/*****************************************************************************/
/* THREADmonitorsignalandwait                                                */
/*                                                                           */
/*****************************************************************************/

int
THREADmonitorsignalandwait (THREAD_MONITOR monitor, int sigcond, int waitcond)
{
    THREAD tcb;
    THREAD_MONITOR_CONDITION condq;

    if (!(THREADmonitorCheckOK(monitor, sigcond, "THREADmonitorwait")))
        return 0;
    if (!(THREADmonitorCheckOK(monitor, waitcond, "THREADmonitorwait")))
        return 0;

    THREAD_PROTECT;
    THREADlock(&monitor->queue.lock);

    /* first check to see if there is anyone to signal */
    condq = &monitor_condition(monitor, sigcond);
    THREADlock(&(condq->queue.lock));
    if ((tcb = THREADqueueGet(&(condq->queue))) != NULL) {
        if (THREAD_IN_WAIT(tcb)) {
            /* ************************************************************
               tcb is blocking in wait and occupying some other process.
               THREADfirstpipe+THREAD_PORDER(tcb) is the *magic* pipe
               exclusive for the process it is on.
               we write a letter to it and wake it up from select call

               N.B. don't put tcb onto runq. it's already running.
               ************************************************************/
            write(THREADfirstpipe+THREAD_PORDER(tcb), "a", 1);
            monitor_set_active(monitor, tcb);
            THREADunlock(&(condq->queue.lock));
            THREADunlock(&monitor->queue.lock);

            /* put THREADcurrent to condition queue */
            condq = &monitor_condition(monitor, waitcond);
            THREADlock(&((THREADcurrent)->context.lock));
            THREADlock(&condq->queue.lock);
            THREAD_TCBAPPEND(THREADcurrent, &(condq->queue));
            THREADunlock(&(condq->queue.lock));

            THREAD_PASSPROTECT;

            THREADlock(&THREADrunqlock);
            THREADrun(); /* runq and THREADcurrent must be locked already */
```

```
            return(1); /* when we get here, we have just been signalled */
        }
        THREADlock(&tcb->context.lock);
    } else {
        if ((tcb = THREADqueueGet(&(monitor->queue))) != NULL) {
            THREADlock(&(tcb->context.lock));
        } else {
            /* no one is waiting, so monitor becomes inactive */
            monitor_clear_active(monitor);
        }
    }

    THREADunlock(&condq->queue.lock);
    /* put ourself on wait condition queue */
    condq = &monitor_condition(monitor, waitcond);
    THREADlock(&((THREADcurrent)->context.lock));
    THREADlock(&condq->queue.lock);
    THREAD_TCBAPPEND(THREADcurrent, &(condq->queue));
    THREADunlock(&(condq->queue.lock));

    /*
     * leave current thread locked for call to THREADrun, below
     * now, if we had found another thread to bring into the monitor, put
     * it on the runq
     */
    if (tcb != NULL) {
        monitor_set_active(monitor, tcb);
        THREADlock(&THREADrunqlock);
        THREAD_MOVETORUNQ(tcb);
        THREADunlock(&THREADrunqlock);
        THREADunlock(&tcb->context.lock);
    }
    THREADunlock(&monitor->queue.lock);

    THREAD_PASSPROTECT;

    THREADlock(&THREADrunqlock);
    THREADrun();

    /* when we get here, we have just been signalled */
    return(1);
}



/*****************************************************************************/
/* THREADmonitorabort                                                        */
/*                                                                           */
/* if tcb is in the monitor, then get it out.                                */
/*****************************************************************************/

void
THREADmonitorabort (THREAD tcb, THREAD_MANAGER manager)
{
    THREAD_MONITOR monitor;
    THREAD nexttcb;
    void (*resetfunc)();

    monitor = THREADmanagerdata(manager);

    THREAD_PROTECT;
    THREADlock(&monitor->queue.lock);
    if (monitor_thread(monitor) == tcb) {
        /* the thread is in the monitor */
        /* put the monitor back into a consistent state */
        if ((resetfunc = monitor_reset(monitor)) != NULL) {
            (*resetfunc)(monitor);
        }
        if ((nexttcb = THREADqueueGet(&(monitor->queue))) != NULL) {
            /* allow the next thread in the entry queue to enter the monitor */
            THREADlock(&nexttcb->context.lock);
            monitor_set_active(monitor, nexttcb);
            THREADlock(&THREADrunqlock);
            THREAD_MOVETORUNQ(nexttcb);
            THREADunlock(&THREADrunqlock);
            THREADunlock(&nexttcb->context.lock);
        } else {
            monitor_clear_active(monitor);
        }
    } else {
        if (THREAD_IN_WAIT(tcb)) {
            /* if tcb waiting in select? */
            write(THREADfirstpipe+THREAD_PORDER(tcb), "a", 1);
            monitor_clear_active(monitor);
        }
    }


    THREADunlock(&monitor->queue.lock);
    THREAD_UNPROTECT;

    return;
}



/*****************************************************************************/
/* THREADmonitorwaitevent                                                    */
/*                                                                           */
/* this function has a similar functionality with system call select         */
/*                                                                           */
/* If we don't need to wait (i.e. *timeout is clear) then we just call       */
/* select and return it value.                                               */
/* If we need to wait, we put the thread on condition queue, and let the     */
/* next candidate (if any) run on another processor (if any). We *don't*     */
/* give up this processor, because we are waiting for returning from         */
/* select                                                                    */
/*****************************************************************************/

int
THREADmonitorwaitevent (THREAD_MONITOR monitor, int condition, int limit,
                        fd_set *readvec, fd_set *writevec, fd_set *xvec,
                        struct timeval *timeout)
{
    int must_wait;
    THREAD tcb;
    int ret;

    if (!(THREADmonitorCheckOK(monitor, condition, "THREADmonitorwaitevent")))
        return -1;

    must_wait = ((timeout) && !(timerisset(timeout))) ? 0 : 1;

    THREAD_PROTECT;
    if (must_wait) {
        /* put itself to condition queue */
```

```
        THREADlock(&(monitor_condition(monitor, condition)).queue.lock);
        THREAD_TCBAPPEND(THREADcurrent,
                         &(monitor_condition(monitor, condition)).queue);
        THREADunlock(&(monitor_condition(monitor, condition)).queue.lock);
        FD_SET(THREADfirstpipe + THREAD_PORDER(THREADcurrent), readvec);

        /* if there is other candidates in monitor->queue, let it in */
        THREADlock(&monitor->queue.lock);
        tcb = THREADqueueGet(&(monitor->queue));
        if (tcb != NULL) {
            THREADlock(&tcb->context.lock);
            monitor_set_active(monitor, tcb);
            THREADlock(&THREADrunqlock);
            THREAD_MOVETORUNQ(tcb);
            THREADunlock(&THREADrunqlock);
            THREADunlock(&tcb->context.lock);
        } else
            monitor_clear_active(monitor);
        THREADunlock(&monitor->queue.lock);
    }

    ret = select(limit, readvec, writevec, xvec, timeout);

    if (must_wait) {
        if (FD_ISSET(THREADfirstpipe + THREAD_PORDER(THREADcurrent), readvec)){
            /* the queue on which tcb lies is *signaled* */
            char c;
            read(THREADfirstpipe+THREAD_PORDER(THREADcurrent), &c, 1);
            FD_CLR(THREADfirstpipe + THREAD_PORDER(THREADcurrent), readvec);
            if (ret == 1) { /* no one is available */
                ret = 0;
                THREAD_UNPROTECT;
            } else {
                ret += ((1 << 30) - 1);
            }
            if (monitor_active(monitor)) {
                if (monitor_thread(monitor) != THREADcurrent) {
                    logerr(ERROR, "THREADmonitorwaitevent: 1");
                }
                THREADunlock(&((THREADcurrent)->context.lock));
                THREAD_UNPROTECT;
            }
            return ret;
        } /* if FD_ISSET */
    }

    THREADlock(&monitor->queue.lock);
    if  (monitor_active(monitor)) {
        /* permission denied, so we wait */
        THREAD_TCBINSERT(THREADcurrent, &monitor->queue);
        THREADunlock(&monitor->queue.lock);
        THREAD_UNPROTECT;
        THREADlock(&THREADrunqlock);
        THREADrun();
        /* it will only be waken up when other thread leave this monitor */
    } else {
        /* permission granted */
        monitor_set_active(monitor, THREADcurrent);
        THREADunlock(&((THREADcurrent)->context.lock));
        THREADunlock(&monitor->queue.lock);
        THREAD_UNPROTECT;
    }

    return ret;
```

```
/*****************************************************************************/
/*      thread - thread package.                                             */
/*                                                                           */
/*      Copyright 1986 Thomas W. Doeppner Jr. - Brown University             */
/*      - All rights reserved.                                               */
/*****************************************************************************/


#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <sys/ipc.h>
#include <alloca.h>
#include <sys/shm.h>
#include <signal.h>
#include <sys/wait.h>

#include "lock.h"
#include "mp.h"
#include "family.h"
#include "thread.h"
#include "manager.h"
#include "runQ.h"
#include "thread_type1.h"
#include "tsignal.h"
#include "logerr.h"
#include "preempt.h"
#include "io.h"

#define MAXPROC 12

#ifndef private_share_m
struct private_data THREADprocs[MAXPROC];
#else
int THREADproc_pid[MAXPROC];
int THREADproc_sigpause[MAXPROC];
#endif


int THREADfirstpipe;
int proc_num = 0;     /* the current # of processes */
static LOCK sync_lock;       /* sync. lock for file table & new proc */



/*****************************************************************************/
/* MPROC_init                                                                */
/*                                                                           */
/* argument n is the # of processes we will gonna have.                      */
/* creat n + 1 socketpairs:                                                  */
/*   the first socketpair is for file table update use.                      */
/*   the rest are one for each process to deal with THREADmonitorwaitevent */
/*****************************************************************************/

void
MPROC_init (int n)
{
    int i, id[2];

    sync_lock = UNLOCKED;

    if ((socketpair(AF_UNIX, SOCK_STREAM, 0, sfd)) == -1) {
        logerr(DISASTER, "MPROC_init: Can not open a socketpair.");
    }

    for (i = 0; i < n; i++) {
        if ((socketpair(AF_UNIX, SOCK_STREAM, 0, id)) == -1) {
            logerr(DISASTER, "MPROC_init: Can not open a socketpair.");
        }
        if (i == 0)
            THREADfirstpipe = id[0];
    }
}


/*****************************************************************************/
/* alldone                                                                   */
/*****************************************************************************/


static
void
alldone (int sig)
{
    char buf[100];

    if (!THREADdying && THREADdonttreadonme) {
        THREAD_KEEPSIGNAL(sig);
        return;
    }

    if (!THREADdying) {
        sprintf(buf, "%d terminating\n", THREADpindex);
        write(2, buf, strlen(buf));
    } else {
        sprintf(buf, "%d aborting (pid %d)\n", THREADpindex, getpid());
        write(2, buf, strlen(buf));
    }
    exit(1);
}



/*****************************************************************************/
/* runprocessor                                                              */
/*                                                                           */
/* Create a private thread on this processor                                 */
/*                                                                           */
/* which : which processor. 0: main processor                                */
/*****************************************************************************/

static
void
runprocessor (int which, void (*func)(), int *args, int argsize,
              int stacksize, int priority)
{
    char buf[100];

    THREADmain = (THREAD)alloca (sizeof(*THREADmain) +
```

```
                                THREADdynamic_space(THREADtype1_extent));
    if (THREADmain == 0) {
        char buf[100];
        sprintf(buf, "alloca failure on processor %d\n", THREADwhichprocessor);
        write(2, buf, strlen(buf));
    }


    bzero(THREADmain, sizeof(*THREADmain) +
          THREADdynamic_space(THREADtype1_extent));
    THREADdynamic_set_size(THREADmain, THREADtype1_extent);

    THREADcontextthreadinit(THREADmain);

    THREADcurrent = THREADmain;
    THREAD_PRIORITY(THREADmain) = 31;  /* the worst priority */


    if (which == 0) {
        THREADcreate(func, args, argsize, 1, stacksize, priority);
        /*
         * nwo that the first thread has been created, we can decrement
         * the active count to amke up for having incremented it in
         * THREADgo
         */
        THREADactivedecr;
    }

    while (1) {
        if (THREADdying)
            alldone(31);
        THREADreschedule();

        if (THREADactivecount <= 0) {
            int x;

            for (x = 0; x < proc_num; x++)
                if (THREADnsigpause(x))
                    kill(THREADnpid(x), SIGVTALRM); /* interrupt sigpause */

            if (which != 0) {
                exit(0);
            }

            while ((wait(0)) >= 0) /* main process awaits others */
                ;
            THREADcurrent = NULL;
            return;
        }
        THREADnsigpause(which) = 1;
        sigpause(0);
        THREADnsigpause(which) = 0;
    }
}


/*****************************************************************************/
/* THREADgo  -  initialize the thread environment                            */
/*                                                                           */
/* This routine initializes the thread environment for a multiprocessor.     */
/* The strategy is to create a unix process for each processor in such a     */
/* way so that all data sections are shared.                                 */
/*                                                                           */
/* The sequence of events here is first to set up the initial thread, then   */
/* to start up each of the unix processes, putting them into their           */
/* scheduling loops.                                                         */
/*                                                                           */
/* For shared memory versions, there will be nprocs identical processes      */
/* created (only one copy of text & data section.                            */
/*                                                                           */
/* nprocs  : specify the # of processes (in shared memory version.)          */
/*           specify the # of processors used (in multiprocessor version)    */
/*****************************************************************************/

void
THREADgo(nprocs, moredata, func, args, argsize, stacksize, priority)
int nprocs;
int moredata;
void (*func)();
int *args;
int argsize;
int stacksize;
int priority;
{
    int i;
    char buf[108];

    int sharem_id;
    struct shmid_ds s_buf;

    sharem_id = malloc_init();

    if (nprocs >= MAXPROC) {
        char buf[100];
        sprintf(buf, "Warning! THREADgo's nproc = %d is reduced to %d\n",
                nprocs, MAXPROC);
        write(2, buf, strlen(buf));
        nprocs = MAXPROC;
    }


    if (THREADtypecount == 0) {
        THREADtypecount = THREADpresetdynamics;
        THREADtype1_extent = sizeof(thread_family) + sizeof(thread_manager_hd);
        THREADmanager_offset = sizeof(thread_family);
    }

    runQInit();
    THREADsiginit();

    /*
     * set the active count to 1 now, to make certain that no one sees
     * it as zero before we get around to creating the first thread
     */
    THREADactiveincr;

    /*
     * this for loop will create nproc-1 child processes. These processes
     * call "runprocessor" in turn. We treat processes as virtual processors
     *
     * using wait & signal to coordinate the control flow
     */
    MPROC_init(nprocs);

    proc_num = 1; /* for this process */
```

```
#ifndef private_share_m
    THREADn = 0; /* this is a private data */
#else
    THREADproc_pid[0] = getpid();
#endif
    THREADwhichprocessor = getpid();

    for (i = 1; i < nprocs; i++) {
        int cpid;
        char buf[100];

        THREADlock(&sync_lock);

        if ((cpid = fork()) == -1) {
            sprintf(buf, "THREADgo: fork failure! Only %d processes are used",
                    proc_num);
            write(2, buf, strlen(buf));
            break;
        } else if (cpid == 0) {  /* child process */
            int j;
#ifndef private_share_m
            THREADn = proc_num;    /* this is a private data */
#else
            THREADproc_pid[proc_num] = getpid();
#endif
            THREADwhichprocessor = getpid();
            for (j = 0; j < nprocs; j++) {
                if (j != THREADn) {
                    close((j * 2) + 1 + THREADfirstpipe);
                    if (j == 0)
                        continue;
                    dup2((j * 2) + THREADfirstpipe, THREADfirstpipe + j);
                    close((j * 2) + THREADfirstpipe);
                } else {
                    close((j * 2) + THREADfirstpipe);
                    dup2((j * 2) + 1 + THREADfirstpipe, THREADfirstpipe + j);
                    close((j * 2) + 1 + THREADfirstpipe);
                }
            }

            THREADunlock(&sync_lock);


            /* we treat each process as a virtual processor
             * so each child process calls "runprocessor" with NULL func.
             */

            runprocessor(i, 0, 0, 0, 0, 0);
            exit(1);

        }


        THREADlock(&sync_lock);
        THREADunlock(&sync_lock);
        proc_num++;


    }
    /* proc_num is the current # of processes we have */

    close(THREADfirstpipe);
    dup2(THREADfirstpipe + 1, THREADfirstpipe);
    close(THREADfirstpipe + 1);

    for (i = 1; i < nprocs; i++) {
        close((i * 2) + 1 + THREADfirstpipe);
        dup2((i * 2) + THREADfirstpipe, THREADfirstpipe + i);
        close((i * 2) + THREADfirstpipe);
    }

    runprocessor(0, func, args, argsize, stacksize, priority);


    shmctl(sharem_id, IPC_RMID, &s_buf); /* clean it before we back */

}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#include <stdio.h>
#include "tcb.h"
#include "privatedata.h"
#include "preempt.h"
#include "mp.h"

/*****************************************************************************/
/* THREADclockhandler                                                        */
/*                                                                           */
/* SIGVTALRM signal handler's routine                                        */
/*****************************************************************************/

void
THREADclockhandler (int sig)
{
    if (THREADcurrent == NULL)
        return;

    if (THREADnsigpause(THREADn) == 1) /* used to interrupt sigpause */
        return;

    if (THREADdonttreadonme) {
        THREAD_KEEPSIGNAL(sig);
        return;
    }

    THREADlock(&THREADcurrent->context.lock);
    if (THREAD_EXCEPTION(THREADcurrent)) {
        int param = THREADexception_param(THREADcurrent);

        /* an exception is pending for this thread */
        THREAD_EXCEPTION_RESET(THREADcurrent);
        /* "turn off" the exception so that it won't appear as if there is an
           exception in progress */
        THREADexception_inprogress(THREADcurrent) = 0;
        THREADexception_param(THREADcurrent) = 0;
        THREADunlock(&THREADcurrent->context.lock);
        /* now that we are the current thread, raise the exception again */
        sigsetmask(0);
        THREADraiseexception(THREADcurrent, param);
        return;
    }

    if (THREAD_SIG(THREADcurrent)) {
        /* a signal is pending for this thread */
        THREADunlock(&THREADcurrent->context.lock);
        sigsetmask(0);
        THREADfakesignal(THREADcurrent, THREADcurrent->sig.func,
                         THREADcurrent->sig.priority, THREADcurrent->sig.sig,
                         THREADcurrent->sig.code);
    } else
        THREADunlock(&THREADcurrent->context.lock);


    sigsetmask(0);
    THREADreschedule(); /* ******** the essence of this func. ********* */
}
```

```
/*****************************************************************************/
/* THREADprotect                                                             */
/*                                                                           */
/* for thread.external.h (export use)                                        */
/*****************************************************************************/


void
THREADprotect ()
{
    THREAD_PROTECT;
}



/*****************************************************************************/
/* THREADunprotect                                                           */
/*                                                                           */
/* for thread.external.h (export use)                                        */
/*****************************************************************************/


void
THREADunprotect ()
{
    THREAD_UNPROTECT;
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/


#include <stdio.h>
#include "lock.h"
#include "queue.h"
#include "tcb.h"
#include "logerr.h"
#include "privatedata.h"
#include "preempt.h"
#include "runQ.h"



/*****************************************************************************/
/* THREADqueueInit                                                           */
/*****************************************************************************/

void
THREADqueueInit (THREAD_QUEUE list)
{
    list->lock = UNLOCKED;
    list->last = NULL;
}



/*****************************************************************************/
/* THREAD_TCBAPPEND                                                          */
/*                                                                           */
/* tcb will be put at the last of the list                                   */
/*****************************************************************************/

void
THREAD_TCBAPPEND (THREAD tcb, THREAD_QUEUE list)
{
    if (list->last) {
        tcb->queue.next = (list->last)->queue.next;
        list->last = (list->last)->queue.next = tcb;
    } else
        list->last = tcb->queue.next = tcb;

    THREAD_WHICH_QUEUE(tcb) = list;
}



/*****************************************************************************/
/* THREAD_TCBINSERT                                                          */
/*                                                                           */
/* tcb will be put at the beginning of the list                              */
/*****************************************************************************/

void
THREAD_TCBINSERT (THREAD tcb, THREAD_QUEUE list)
{
    if (list->last) {
        tcb->queue.next = (list->last)->queue.next;
        (list->last)->queue.next = tcb;
    } else
        list->last = tcb->queue.next = tcb;

    THREAD_WHICH_QUEUE(tcb) = list;
}



/*****************************************************************************/
/* THREADqueueGet                                                            */
/*                                                                           */
/* the first tcb in the list will removed and returned                       */
/*****************************************************************************/


THREAD
THREADqueueGet (THREAD_QUEUE list)
{
    THREAD f;

    if (list->last == NULL)
        return NULL;

    f = (list->last)->queue.next;
    if (f == list->last)
        list->last = NULL;
    else
        (list->last)->queue.next = f->queue.next;

    THREAD_WHICH_QUEUE(f) = NULL;
    return f;
}


/*****************************************************************************/
/* THREADqueuePeep                                                           */
/*                                                                           */
/* peep the first element in the list. no side effect at all                 */
/*****************************************************************************/

THREAD
THREADqueuePeep (THREAD_QUEUE list)
{
    return ((list->last) ? ((list->last)->queue.next) : NULL);
}



/*****************************************************************************/
/* THREADpullfromqNL                                                         */
/*                                                                           */
/* NL stands for no lock                                                     */
/* this is a shortcut version of THREADpullfromq.  It is assumed that tcbp */
/* is locked, its queue is locked, and that it really is in the queue.      */
/*****************************************************************************/

THREAD
THREADpullfromqNL (THREAD tcbp)
{
    THREAD_QUEUE qp;
    THREAD t;

    THREAD_PROTECT;
```

```
    qp = THREAD_WHICH_QUEUE(tcbp);

    if (qp == NULL)
        goto notfound;

    if (((t = THREADqueuePeep(qp))) == tcbp) {
        THREADqueueGet(qp);  /* it's the first element */
    } else if (t == NULL) {
        goto notfound;  /* this queue is messed up */
    } else {
        THREAD *p;
        p = &(t->queue.next);
        while (((*p) != tcbp) && ((*p) != t))
            p = &((*p)->queue.next);
        if ((*p) == t)
            goto notfound;

        *p = (*p)->queue.next;  /* this queue had more than 2 tcb */
        THREAD_WHICH_QUEUE(tcbp) = NULL;
    }

    /* now tcbp is off the queue */

    if (THREAD_RUNNABLE(tcbp)) {
        THREAD_RUNNABLE_RESET(tcbp);
        if ((THREADqueuePeep(qp)) == NULL)
            whichQ &= ~(1 << THREAD_PRIORITY(tcbp));
    }

    THREAD_UNPROTECT;
    return tcbp;

  notfound:
    logerr(DISASTER,
           "THREADpullfromqNL: thread should be in queue, but isn't");

    THREAD_UNPROTECT; /* although never reached ... */
    return NULL;
}



/*****************************************************************************/
/* THREADpullfromq                                                           */
/*                                                                           */
/* first lock the tcb, then the queue it is on. To avoid deadlock, if the    */
/* queue_head is locked already, we release the lock on the tcb and try      */
/* it again                                                                  */
/*****************************************************************************/

THREAD
THREADpullfromq (THREAD tcbp)
{
    THREAD_QUEUE qp;
    THREAD t;

    THREAD_PROTECT;

  gettcb:
    THREADlock(&tcbp->context.lock);

    if ((tcbp != THREADcurrent) && THREAD_RUNNING(tcbp)) {
        THREADunlock(&tcbp->context.lock);
        logerr(WARNING,
               "THREADpullfromq: thread is running on another processor");
        THREAD_UNPROTECT;
        return(NULL);
    }

    if ((qp = THREAD_WHICH_QUEUE(tcbp)) == NULL) {
        /* thread is not in a queue */
        THREADunlock(&tcbp->context.lock);
        logerr(WARNING, "THREADpullfromq: thread not in queue");
        THREAD_UNPROTECT;
        return(NULL);
    }

    if (!THREADcondlock(&qp->lock)) {
        THREADunlock((THREAD_RUNNABLE(tcbp) ? &THREADrunqlock
                      : &tcbp->context.lock));
        goto gettcb;
    }

    /* from now on, we get the lock on tcbp and the queue it is on */

    if (((t = THREADqueuePeep(qp))) == tcbp) {
        THREADqueueGet(qp);  /* it's the first element */
    } else if (t == NULL) {
        goto notfound; /* this queue is messed up */
    } else {
        THREAD *p;
        p = &(t->queue.next);
        while (((*p) != tcbp) && ((*p) != t))
            p = &((*p)->queue.next);
        if ((*p) == t)
            goto notfound;

        *p = (*p)->queue.next;  /* this queue had more than 2 tcb */
        THREAD_WHICH_QUEUE(tcbp) = NULL;
    }

    /* now tcbp is off the queue */

    THREADunlock(&tcbp->context.lock);

    if (THREAD_RUNNABLE(tcbp)) {
        THREAD_RUNNABLE_RESET(tcbp);
        if ((THREADqueuePeep(qp)) == NULL)
            whichQ &= ~(1 << THREAD_PRIORITY(tcbp));
        THREADunlock(&THREADrunqlock);
    } else
        THREADunlock(&(qp->lock));

    THREAD_UNPROTECT;
    return tcbp;

  notfound:
    logerr(ERROR, "THREADpullfromq: messed up queue");
    THREADunlock((THREAD_RUNNABLE(tcbp) ? &THREADrunqlock
                  : &tcbp->context.lock));
    THREADunlock(&tcbp->context.lock);
    THREAD_UNPROTECT;
    return NULL;
}
```

```
/*****************************************************************************/
/*      thread - thread package.                                             */
/*                                                                           */
/*      Copyright 1986 Thomas W. Doeppner Jr. - Brown University             */
/*      - All rights reserved.                                               */
/*****************************************************************************/


#include <stdio.h>
#include <signal.h>
#include "mp.h"
#include "tcb.h"
#include "runQ.h"
#include "lock.h"
#include "queue.h"
#include "preempt.h"
#include "logerr.h"
#include "stack.h"


THREAD_QUEUE_HEAD THREADfreezequeue;
THREAD_QUEUE_HEAD freeit_queue;
THREAD_QUEUE_HEAD runQ[32];  /* for [0-31] priority tcb */
LOCK        THREADrunqlock = UNLOCKED;
THREAD_QUEUE_HEAD waitq;   /* if no queue is appropriate, then put tcb here */


int whichQ; /* if bit i = 1, it means queue i-1 holding some runnable tcb */


/*****************************************************************************/
/* runQInit                                                                  */
/*                                                                           */
/* Initial the system run queue. Called only once in THREADgo                */
/*****************************************************************************/

void
runQInit ()
{
    int i;

    for (i = 0; i < 32; i++)
        THREADqueueInit(&(runQ[i]));
    THREADrunqlock = UNLOCKED;

    THREADqueueInit(&freeit_queue);
    THREADqueueInit(&waitq);
    THREADqueueInit(&THREADfreezequeue);
}



/*****************************************************************************/
/* THREADreschedule                                                          */
/*                                                                           */
/* The current thread is about to leave; pick up the next candidate thread */
/* Note. priority 0: best; priority 31 : worst                               */
/*****************************************************************************/


void
THREADreschedule ()
{
    int pri;
    int currentpri;

    currentpri = (THREADcurrent)->context.priority;

    THREAD_PROTECT;
    THREADlock(&((THREADcurrent)->context.lock));
    if (THREAD_DYING(THREADcurrent)) {
        /*
         * the thread has been marked for death.  This happened because
         * THREADdestroy was called with this thread as the target, but
         * this thread was executing (on a different processor from the
         * caller), so a signal was sent to this thread's processor and now
         * we are completing the kill.  If it's on the IO processor,
         * then do this after it gets off
         */
        THREAD_DYING_RESET(THREADcurrent);
        THREADunlock(&((THREADcurrent)->context.lock));
        THREAD_UNPROTECT;
        THREADsuicide();
        THREAD_PROTECT;
        THREADlock(&((THREADcurrent)->context.lock));
    }
    THREADlock(&THREADrunqlock);

    /* runQ and the current thread are locked now */

    pri = ffs(whichQ) - 1;

    if (((pri >= 0) && (pri <= currentpri))||THREAD_EXCEPTION(THREADcurrent)) {
        if (THREADcurrent == THREADmain) {
            /* don't put the main thread on the run queue */
            THREAD_PASSPROTECT;
            THREADrun();
            return;
        }

        THREAD_MOVETORUNQ(THREADcurrent);
        THREAD_RUNNABLE_SET(THREADcurrent);

        THREAD_PASSPROTECT;

        logerr(INFO, "reschedule : will call threadrun (2)");

        THREADrun();
    } else {
        /* there are no other ready threads, so this is a no-op */
        THREADunlock(&THREADrunqlock);
        THREADunlock(&((THREADcurrent)->context.lock));
        THREAD_UNPROTECT;
    }
}



/*****************************************************************************/
/* THREADrun                                                                 */
/*                                                                           */
/* THREADcurrent will be switched out. It should be put onto appropriate    */
/* queue before this THREADrun is invoked.                                   */
/* Make the next thread in the runQ of highest priority to be running       */
/*                                                                           */
/* for multiprocessor, if the current thread is not NULL, the caller MUST   */
/* ensure that both current thread the runQ are locked                       */
/*****************************************************************************/
```

```
void
THREADrun ()
{
    int qindex;

    THREAD_PROTECT;

    if (THREADcurrent != NULL) {
        if (THREAD_DYING(THREADcurrent)) {
            /*
             * the thread has been marked for death. This happened because
             * THREADdestroy was called with this thread as the target, but
             * this thread was executing on a different processor from the
             * caller, so a signal was sent to this thread's processor and now
             * we are completing the kill.
             */
            THREAD_DYING_RESET(THREADcurrent);
            THREADunlock(&THREADrunqlock);
            THREADunlock(&THREADcurrent->context.lock);

            THREAD_UNPROTECT;
            THREADsuicide(); /* murderred set, and longjmp to startup */
            THREAD_PROTECT;

            THREADlock(&((THREADcurrent)->context.lock));
            THREADlock(&THREADrunqlock);
        }

        if (THREAD_RERUN(THREADcurrent)) {
            /*
             * the current thread has just been thawed and we must run it now
             * since we're already on its stack, but not in the proper frame
             */
            THREAD_RERUN_SET(THREADcurrent);
#ifdef sparc
            THREADflush_window();
#endif sparc
            goto run; /* ready to roll */
        }

        if (THREAD_DELETE(THREADcurrent)) {
            /* once we switch to another stack, we'll delete this threa */
            THREADlast = THREADcurrent;
        }

        THREAD_FRAME(THREADcurrent) = (struct frame*) GETFRAME();

#ifdef sparc

        THREADsave_and_flush_window(&THREAD_FRAME(THREADcurrent),
                                    &THREAD_STACK(THREADcurrent));
#endif sparc

        if (THREAD_STACKOUTOFBOUNDS(THREADcurrent))
            logerr(DISASTER, "suspending thread's stack has exceeded  bounds");

        THREAD_RUNNING_RESET(THREADcurrent);
        THREADunlock(&((THREADcurrent)->context.lock));
        THREADcurrent = NULL;
    } else {  /* THREADcurrent == NULL */
        THREADlock(&THREADrunqlock);
#ifdef sparc
        THREADflush_window();
#endif
    }

    if ((qindex = ffs(whichQ) - 1) < 0) {
        THREADcurrent = THREADmain;
        THREADlock(&((THREADcurrent)->context.lock));
        goto run;
    }

    THREADcurrent = THREADqueueGet(&(runQ[qindex]));
    THREADlock(&((THREADcurrent)->context.lock));
    if (THREADqueuePeep(&(runQ[qindex])) == NULL)
        whichQ &= ~(1 << qindex);

    if ((THREAD_ACTIVE(THREADcurrent)) == 0)
        logerr(DISASTER, "attempted to run inactive thread");

  run:
    SETSTACK(THREAD_STACK(THREADcurrent));
    SETFRAME(THREAD_FRAME(THREADcurrent));


    /*************************************************************/
    /* we are now in the context of the thread                   */
    /* if we can 'return' from this function, then resume exec. */
    /*************************************************************/

    THREAD_RUNNING_SET(THREADcurrent);
    THREAD_RUNNABLE_RESET(THREADcurrent);

    (THREADcurrent)->context.processor = THREADwhichprocessor;
    THREAD_PORDER(THREADcurrent) = THREADn;

    THREADunlock(&THREADrunqlock);
    THREADunlock(&((THREADcurrent)->context.lock));


    if (THREADlast != NULL) {
        /* malloc will take care of freeit_queue */
        THREADlock(&(freeit_queue.lock));
        THREAD_TCBAPPEND(THREADlast, &freeit_queue);
        THREADunlock(&(freeit_queue.lock));
        THREADlast = NULL;
    }

#ifdef sparc
    /* this is only for debugging */

    THREAD_FRAME(THREADcurrent) = (struct frame *) GETFRAME();
    if (THREAD_STACKOUTOFBOUNDS(THREADcurrent)) {
        logerr(DISASTER, "stack problem");
    }

#endif

    THREAD_UNPROTECT;

    if (THREAD_EXCEPTION(THREADcurrent) && !THREAD_IO(THREADcurrent)) {
        /* raising an exception in the thread */
        THREAD_EXCEPTION_RESET(THREADcurrent);
        THREADexceptioncall();
    }
```

```
    if (THREAD_SIG(THREADcurrent)) { /* a signal is pending for this thread */
        THREADfakesignal(THREADcurrent, THREADcurrent->sig.func,
                         THREADcurrent->sig.priority, THREADcurrent->sig.sig);
    }

    if (THREAD_DYING(THREADcurrent)) {
        /*
         * the thread has been marked for death.  This happened because
         * THREADdestroy was called with this thread as the target, but
         * this thread was executing (on a different processor from the
         * caller), so a signal was sent to this thread's processor and now
         * we are completing the kill.  If it's on the IO processor, then
         * do this after it gets off.
         */
        THREAD_DYING_RESET(THREADcurrent);
        THREADsuicide();
    }

#ifdef sparc
    /*
     * i0 and i1 need to be set to the address of the first function and
     * its args (if this thread is being run for the first time).
     * On nice architectures, this would be done automatically by restoring
     * these regs from the stack frame on return.  On the sparc, the
     * registers are "restored" when we switch to the thread's stack
     * (above).
     * Unfortunately, the compiler uses i0 and i1 for other purposes
     * between then and now, so we set i0 and i1 now
     */
    THREADfixregs();
#endif sparc
}



/*****************************************************************************/
/* THREAD_MOVETORUNQ                                                         */
/*****************************************************************************/


void
THREAD_MOVETORUNQ (THREAD t)
{
#define pri (THREAD_PRIORITY(t))

    THREAD_QUEUE q;

    THREAD_PROTECT;

    if (THREAD_ACTIVE(t) == 0)
        logerr(DISASTER, "THREADmovetorunq: moving inactive thread to runq");

    if (THREAD_FROZEN(t)) {
        THREADlock(&THREADfreezequeue.lock);
        THREAD_TCBAPPEND(t, &THREADfreezequeue);
        THREADunlock(&THREADfreezequeue.lock);
    } else {
        q = &runQ[pri];
        THREAD_RUNNABLE_SET(t);
        THREAD_TCBAPPEND(t, q);
        if (whichQ == 0) {  /* test if some process is in sigpause */
            int x;

            whichQ |= (1 << THREAD_PRIORITY(t));
            for (x = 0; x < proc_num; x++)
                if (THREADnsigpause(x) == 1)
                    kill(THREADnpid(x), SIGVTALRM);
        } else
            whichQ |= (1 << THREAD_PRIORITY(t));
    }

    THREAD_UNPROTECT;
#undef pri
}

/*****************************************************************************/
/* THREAD_QMOVETORUNQ                                                        */
/*                                                                           */
/* a streamlined version                                                     */
/*****************************************************************************/


void
THREAD_QMOVETORUNQ (THREAD t)
{
#define pri (THREAD_PRIORITY(t))

    THREAD_QUEUE q = &runQ[pri];

    THREAD_RUNNABLE_SET(t);
    THREAD_TCBAPPEND(t, q);
    whichQ |= (1 << THREAD_PRIORITY(t));

#undef pri
}



/*****************************************************************************/
/* THREADfrozen                                                              */
/*****************************************************************************/


int
THREADfrozen (THREAD tcb)
{
    return((THREAD_FROZEN(tcb)) ? 1 : 0);
}


/*****************************************************************************/
/* THREADfrozenparent                                                        */
/*****************************************************************************/

THREAD
THREADfrozenparent (THREAD tcb)
{
    return(THREAD_FROZEN_PARENT(tcb));
}



/*****************************************************************************/
/* THREADfreeze                                                              */
/*                                                                           */
/* this routine effectively "freezes" its argument. if the thread is on a    */
/* wait queue, it won't be moved, but any frozen thread will go to the       */
/* freeze queue instead of the runq.                                         */
```

```
/*                                                                          */
/* N.B.: if the thread is THREADcurrent, it is the caller's responsibility */
/* to do any rescheduling.                                                  */
/*                                                                          */
/* The caller locks tcb and, if tcb is on the runq, the runq. This routine */
/* will return with both unlocked.                                          */
/****************************************************************************/

int
THREADfreeze (THREAD tcb)
{
    THREAD_PROTECT;
    THREADlock(&THREADfreezequeue.lock);

    THREAD_FREEZE_SET(tcb);

    if (THREAD_RUNNABLE(tcb)) {
        THREADpullfromqNL(tcb);
        THREADunlock(&THREADrunqlock);
    }

    if (THREAD_WHICH_QUEUE(tcb) == NULL) {
        /* either it was just removed from the runq above, or it's
           THREADcurrent */
        THREAD_TCBAPPEND(tcb, &THREADfreezequeue);
    }

    THREADunlock(&tcb->context.lock);
    THREADunlock(&THREADfreezequeue.lock);
    THREAD_UNPROTECT;
    return(1);
}



/****************************************************************************/
/* THREADunfreeze                                                           */
/*                                                                          */
/* Thaw tcb; if the caller is the child of this freeze, then we get rid of */
/* the caller                                                               */
/****************************************************************************/


int
THREADunfreeze (THREAD tcb)
{
    THREAD_PROTECT;
    THREADlock(&THREADcurrent->context.lock);
    THREADlock(&THREADfreezequeue.lock);
    THREADlock(&THREADrunqlock);
    THREADlock(&tcb->context.lock);

    if (!(THREAD_FROZEN(tcb))) {
        THREADunlock(&THREADrunqlock);
        THREADunlock(&THREADfreezequeue.lock);
        THREADunlock(&tcb->context.lock);
        THREADunlock(&THREADcurrent->context.lock);
        logerr(ERROR, "THREADunfreeze: thread not frozen");
        THREAD_UNPROTECT;
        return(0);
    }

    THREAD_FREEZE_RESET(tcb);
    if (THREAD_FROZEN_PARENT(THREADcurrent) == tcb) {
        /* Our parent is being thawed, so we self-destruct */
        if (THREAD_WHICH_QUEUE(tcb) == &THREADfreezequeue) {
            /* the thread would be on the runq if it weren't frozen */
            if (THREADpullfromqNL(tcb) == NULL) {
                logerr(DISASTER, "THREADunfreeze: bad pullfromq");
            }
            THREADunlock(&THREADfreezequeue.lock);
            THREAD_MOVETORUNQ(tcb);
        } else {
            THREADunlock(&THREADfreezequeue.lock);
        }
        THREADunlock(&tcb->context.lock);
        THREAD_PASSPROTECT;
        THREADrun();
        logerr(DISASTER, "THREADunfreeze: shouldn't be here");
    } else {
        /* We are thawing someone else's parent */
        if (THREAD_WHICH_QUEUE(tcb) == &THREADfreezequeue) {
            /* the thread would be on the runq if it weren't frozen */
            if (THREADpullfromqNL(tcb) == NULL) {
                logerr(DISASTER, "THREADunfreeze: bad pullfromq");
            }
            THREADunlock(&THREADfreezequeue.lock);
            THREAD_MOVETORUNQ(tcb);
        } else
            THREADunlock(&THREADfreezequeue.lock);

        THREADunlock(&THREADrunqlock);
        THREADunlock(&tcb->context.lock);
        THREADunlock(&THREADcurrent->context.lock);
        THREAD_UNPROTECT;
        return(1);
    }
}



/****************************************************************************/
/* THREADrunqthreadinit                                                     */
/****************************************************************************/
THREADrunqthreadinit (THREAD t)
{
}



/****************************************************************************/
/* THREADqueuethreadinit                                                    */
/****************************************************************************/

THREADqueuethreadinit (THREAD t)
{
}



/****************************************************************************/
/* THREADsetpriority                                                        */
/****************************************************************************/
```

```
void
THREADsetpriority (THREAD tcb, int priority)

{
   THREAD_PROTECT;

   for (;;) {
      THREADlock(&tcb->context.lock);
      if (THREAD_RUNNABLE(tcb)) {
         if (!THREADcondlock(&THREADrunqlock)) {
            THREADunlock(&tcb->context.lock);
            continue;
         }
      }
      break;
   }

   if (priority > 31)
      priority = 31;
   if (priority < 0)
      priority = 0;

   THREAD_PRIORITY(tcb) = priority;

   if (THREAD_RUNNABLE(tcb)) {
      THREADpullfromq(tcb);
      /* I assume that the thread has not just been made frozen, hence
         I don't lock the freezequeue */
      THREAD_MOVETORUNQ(tcb);
      THREADunlock(&THREADrunqlock);
   }
   THREADunlock(&tcb->context.lock);
   THREAD_UNPROTECT;
}


/*****************************************************************************/
/* THREADanyonewantthecpu                                                    */
/*****************************************************************************/


int
THREADanyonewantthecpu ()
{
   return(whichQ ? 1 : 0);
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#include <stdio.h>
#include "privatedata.h"
#include "preempt.h"
#include "semaphore.h"
#include "runQ.h"



/*****************************************************************************/
/* THREADseminit                                                             */
/*****************************************************************************/



SEMAPHORE
THREADseminit (int initialvalue)
{
    SEMAPHORE sem;

    sem = (SEMAPHORE) _malloc(sizeof(*sem));
    sem->value = initialvalue;
    THREADqueueInit (&(sem->queue));

    return(sem);
}



/*****************************************************************************/
/* THREADpsem                                                                */
/*****************************************************************************/


void
THREADpsem (SEMAPHORE sem)
{
    THREAD_PROTECT;
    THREADlock(&sem->queue.lock);
    if (sem->value <= 0) {
        THREADlock(&THREADcurrent->context.lock);
        THREAD_TCBAPPEND(THREADcurrent, &(sem->queue));
        THREADunlock(&sem->queue.lock);
        THREAD_PASSPROTECT;
        THREADlock(&THREADrunqlock);
        THREADrun();
    } else {
        sem->value--;
        THREADunlock(&sem->queue.lock);
        THREAD_UNPROTECT;
    }
    return;
}



/*****************************************************************************/
/* THREADvsem                                                                */
/*****************************************************************************/



void
THREADvsem (SEMAPHORE sem)
{
    THREAD tcb;

    THREAD_PROTECT;
    THREADlock(&sem->queue.lock);

    if ((tcb = THREADqueuePeep(&(sem->queue))) != NULL) {
        tcb = THREADqueueGet(&(sem->queue));
        THREADlock(&tcb->context.lock);
        THREADlock(&THREADrunqlock);
        THREAD_MOVETORUNQ(tcb);
        THREADunlock(&THREADrunqlock);
        THREADunlock(&tcb->context.lock);
    } else {
        sem->value++;
    }
    THREADunlock(&sem->queue.lock);
    THREAD_UNPROTECT;
    return;
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#include <stdio.h>
#include "stack.h"
#include "logerr.h"
#include "preempt.h"



STACK   *THREADstackspace = NULL;
STACK   *THREADstackspaceend = NULL;
int     THREADstdstacksize = 0;
int     THREADnstacks = 0;
STACK   *THREADfreestacks = 0;




int
THREADstackspaceinit(stacksize, nstacks)
int stacksize;
int nstacks;
{
   static int onceonly;
   int i;

   if (onceonly++) {
      logerr(WARNING, "THREADstackspaceinit: called more than once");
      return(0);
   }

   THREADstdstacksize = (stacksize + 3) & 0xfffffffc;  /* make it a mult of 4 */
   THREADnstacks = nstacks;
   THREADstackspace = (STACK *)valloc(THREADstdstacksize*THREADnstacks);
   if (THREADstackspace == NULL)
      return(0);
   THREADstackspaceend = (STACK *)((int)THREADstackspace
        + THREADstdstacksize*THREADnstacks);

   THREADfreestacks = THREADstackspace;

   {
      register STACK *stack, *nextstack;

      for (i = 0, stack = THREADstackspace; i < THREADnstacks - 1; i++) {
         nextstack = (STACK *)((int)stack + THREADstdstacksize);
         stack->next = nextstack;
         stack->magic_number = THREADstackfree;
         stack = nextstack;
      }
      stack->next = NULL;
      stack->magic_number = THREADstackfree;
   }

   return(THREADnstacks);
}
```

```
/*****************************************************************************/
/* setStack                                                                  */
/*                                                                           */
/* set up the initial frame.                                                 */
/* set up stack so that it looks like:                                       */
/*                                                                           */
/*       0                                          fr_savpc                */
/*       addr of word before stack                  fr_savfp                */
/*       ...  (skip over rest of frame)                                     */
/*       saved pc (addr of startrtn)                fr_savpc                */
/*       saved framed ptr (previous frame)          fr_savfp                */
/*       ...  (skip over 4 input regs)                                      */
/*       arg ptr                                    fr_arg[1]               */
/*       first user rtn                       fr_arg[0]                     */
/*       ... (skip over local regs)                                         */
/*   fp ->                                                                   */
/*                                                                           */
/* For safety's sake, there is an empty stack frame at the top               */
/* of the stack.  The second stack frame is entered from THREADrun;          */
/* its input registers, after a return taken in its context, become          */
/* the output registers in the context of the first stack frame, and         */
/* thus contain the parameters passed to startrtn.                           */
/*                                                                           */
/* Stack set in this way, the control flow will be diverted to rtn           */
/*****************************************************************************/



struct frame*
setStack (void (*rtn)(int(*)(), char *[]),  char *stackbegin,
          int (*func)(), char*argp, int argsize)
{
    struct frame *firstframe;
    int *argbegin;

    /* we want to put argbegin on a double word boundary */
    argbegin = (int *)
        ((unsigned int)&(stackbegin[-argsize]) & 0xfffffff8);
    firstframe = (struct frame *)((int)argbegin - THREADFRAMESIZE - 32);
    firstframe->fr_savfp = (struct frame *)((int)argbegin);
    firstframe->fr_savpc = 0;
    /* save the arguments, etc, in the "wrong" stack frame: sun's #$%^ C
       compiler clobbers the registers which these would be loaded into,
       so load them later */
    firstframe->fr_arg[0] = (int)func;
    if (argsize != 0) {
        bcopy(argp, argbegin, argsize);
        firstframe->fr_arg[1] = (int)argbegin;
    } else {
        firstframe->fr_arg[1] = (int)argp;
```

```
    )
    firstframe = (struct frame *)((int)firstframe - THREADFRAMESIZE - 32);
    firstframe->fr_savfp = (struct frame *)((int)firstframe +
                                    THREADFRAMESIZE + 32);
    firstframe->fr_savpc = (int)rtn - 8; /* saved pc points to "call
                    instruction", account for both it and delay slot */

    return (firstframe);
}



/*****************************************************************************/
/* THREADcompstack                                                           */
/*                                                                           */
/* compare the current frame with the argument stack pointer.                */
/* returns true if the stack pointer points to a location in the current     */
/* stack, false otherwise.                                                   */
/*****************************************************************************/

int
THREADcompstack (THREAD tcbp, char *sp)
{
    register char *frame;

    if (tcbp == THREADcurrent) {
#ifndef mips
        frame = (char *)GETFRAME();
#else mips
        frame = (char *)GETSTACK();
#endif mips
    } else {
#ifndef mips
        frame = (char *)THREAD_FRAME(tcbp);
#else mips
        frame = (char *)THREAD_STACK(tcbp);
#endif mips
    }
    if (frame < sp)
        return(1);
    else
        return(0);
}

/*****************************************************************************/
/* THREAD_FREESTACK                                                          */
/*                                                                           */
/* free the thread control block (including the stack.)                      */
/*****************************************************************************/


void
THREAD_FREESTACK (THREAD tcbp)
{
    STACK *stack = (tcbp)->stack.stack_limit;

    THREAD_PROTECT;

    if ((stack < THREADstackspace) || (THREADstdstacksize == 0) ||
        (stack >= THREADstackspaceend))
        _free(stack);
    else {
        stack->next = THREADfreestacks;
        stack->magic_number = THREADstackfree;
        THREADfreestacks = stack;
    }

    THREAD_UNPROTECT;
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#include <stdio.h>
#include "stack.h"
#include "thread.h"
#include "tcb.h"
#include "family.h"
#include "thread_type1.h"
#include "runQ.h"
#include "preempt.h"
#include "logerr.h"

int THREADactivecount;
LOCK THREADactivecountlock = UNLOCKED;


/*****************************************************************************/
/* THREADmakethread                                                          */
/*                                                                           */
/* 1. request a chunk of memory to serve as a stack                          */
/* 2. set this stack so that the control will diverted as                    */
/*    (*rtn) --> (*func)  --> (*rtn) --> terminate itself                    */
/*****************************************************************************/

static
THREAD
THREADmakethread (void (*startrtn)(int(*)(),char*[]),
                  int sizeofstack, int (*func)(),
                  char *argp, int argsize, int dynamic_size, int priority)
{
   THREAD tcb;
#ifdef mips
   int *firstframe;
#else
   struct frame *firstframe;
#endif mips
   int *stackbegin;
   int tcbsize;
   int *argbegin;

   tcbsize = sizeof(*tcb) + THREADdynamic_space(dynamic_size);

   THREAD_ALLOCATESTACK(tcb, tcbsize, sizeofstack, stackbegin, int *);
   if (stackbegin == NULL) {
      logerr(ERROR, "THREADmakethread: no stack space");
      return(NULL);
   }

   /* set up its initial frame */
#ifdef vax
   argbegin = (int *)&((char *)stackbegin)[-argsize];
   /* first on the stack are the arguments to the startup routine, then
      goes the initial stack frame */
   firstframe = (struct frame *)((int)argbegin - sizeof(struct frame)
      - 3*sizeof(int));
   argbegin[-3] = 2;
   argbegin[-2] = (int)func;
   if (argsize != 0) {
      bcopy(argp, argbegin, argsize);
      argbegin[-1] = (int)argbegin;
   } else {
      argbegin[-1] = (int)argp;
   }

   firstframe->fr_mask = 0;
      /* no registers restored */
   firstframe->fr_psw = PSL_USERSET;
      /* user mode psw */
   firstframe->fr_s = 0;
      /* pretend that callg was used */
   firstframe->fr_savap = (int)(&argbegin[-3]);
      /* "saved" ap points to args stored at beginning of stack */
   firstframe->fr_savfp = (int)firstframe + sizeof(struct frame);
      /* "saved" fp points to just before firstframe, so that first frame
         gets itself popped */
   firstframe->fr_savpc = (int)startrtn + 2;  /* avoid register mask */
      /* when THREADrun returns to us, we should start execution in
         startrtn */
   THREAD_FRAME(tcb) = firstframe;
   THREAD_STACK(tcb) = (char *)firstframe;
#endif
#ifdef ns32000
   /* Set up stack so that it looks like:
            0
            arg ptr
            address of first user func
            saved pc (0)
            saved pc (addr of startrtn)
      fp ->  link (addr of 0 above)
            filler for local vars and regs
      sp ->

      After initial return from THREADrun, stack looks like:
      fp ->  0
            arg ptr
            address of first user func
      sp ->  saved pc (0)

      After enter instruction in startrtn:
            0
            arg ptr
            address of first user func
            saved pc (0)
      fp ->  link (address of 0 above)
            local vars
      sp ->  saved regs
   */
   argbegin = (int *)&((char *)stackbegin)[-argsize];
   firstframe = (struct frame *)((int)argbegin - 6*sizeof(int));
   argbegin[-1] = 0;
   argbegin[-3] = (int)func;
#ifdef notdef
   argbegin[-4] = (int)THREADmakethread + 0xf;  /* give startrtn a parent */
#endif
   if (argsize != 0) {
      bcopy(argp, argbegin, argsize);
      argbegin[-2] = (int)argbegin;
   } else {
      argbegin[-2] = (int)argp;
   }
   firstframe->link = (struct frame *)((int)firstframe +
```

```
        sizeof(struct frame) - sizeof(int));
    firstframe->ret = (char *)startrtn;
    THREAD_FRAME(tcb) = firstframe;
    THREAD_STACK(tcb) = (char *)((int)firstframe - 32*sizeof(int));
        /* leave plenty of room for local variables on stack plus regs */
#endif
#if defined(sun) && (defined(mc68010) || defined(mc68020))
    /* Set up stack so that it looks like:

            0                                       fr_arg[3]
            arg pointer                             fr_arg[2]
            first user routine of new thread        fr_arg[1]
            saved pc (0)                            fr_arg[0]
            saved pc (address of startrtn)          fr_savpc
    fp,sp ->  saved frame ptr (address of 0 above)  fr_savfp

       When this thread is started, it will return from THREADrun,
       into startrtn, with the stack looking like:

       fp ->  0
              arg pointer
              first user routine of new thread
       sp ->  saved pc (0)

       startrtn, starting at its first instruction, does a link, resulting in:

              0
              arg pointer
              first user routine of new thread
              saved pc (0)
    fp,sp ->  saved frame ptr (address of 0 above)

    */
    argbegin = (int *)&((char *)stackbegin)[-argsize];
    firstframe = (struct frame *)((int)argbegin - sizeof(struct frame)
         - 3*sizeof(int) /* arguments + 0 word for frame ptr */);
    firstframe->fr_savfp = (struct frame *)&firstframe->fr_arg[3];
    firstframe->fr_savpc = (int)startrtn;
    firstframe->fr_arg[0] = 0;
    firstframe->fr_arg[1] = (int)func;
    if (argsize != 0) {
       bcopy(argp, argbegin, argsize);
       firstframe->fr_arg[2] = (int)argbegin;
    } else {
       firstframe->fr_arg[2] = (int)argp;
    }
    firstframe->fr_arg[3] = 0;
    THREAD_FRAME(tcb) = firstframe;
    THREAD_STACK(tcb) = (char *)firstframe;
#endif
#ifdef sparc
    /* Set up stack so that it looks like:

            0                                       fr_savpc
            addr of word before stack               fr_savfp
            ...  (skip over rest of frame)
            saved pc (addr of startrtn)             fr_savpc
            saved framed ptr (previous frame)       fr_savfp
            ...  (skip over 4 input regs)
            arg ptr                                 fr_arg[1]
            first user rtn                          fr_arg[0]
            ... (skip over local regs)
       fp ->

       For safety's sake, there is an empty stack frame at the top
       of the stack.  The second stack frame is entered from THREADrun;
       its input registers, after a return taken in its context, become
       the output registers in the context of the first stack frame, and
       thus contain the parameters passed to startrtn.
    */

    argbegin =
         (int *)((unsigned int)&(((char *)stackbegin)[-argsize]) & 0xfffffff8);
         /* put it on a double word boundary */
    firstframe = (struct frame *)((int)argbegin - THREADFRAMESIZE - 32);
    firstframe->fr_savfp = (struct frame *)((int)argbegin);
    firstframe->fr_savpc = 0;
    /* save the arguments, etc, in the "wrong" stack frame: sun's #$%^ C
       compiler clobbers the registers which these would be loaded into,
       so load them later */
    firstframe->fr_arg[0] = (int)func;
    if (argsize != 0) {
       bcopy(argp, argbegin, argsize);
       firstframe->fr_arg[1] = (int)argbegin;
    } else {
       firstframe->fr_arg[1] = (int)argp;
    }
    firstframe = (struct frame *)((int)firstframe - THREADFRAMESIZE - 32);
    firstframe->fr_savfp = (struct frame *)((int)firstframe +
         THREADFRAMESIZE + 32);
    firstframe->fr_savpc = (int)startrtn - 8; /* saved pc points to "call
         instruction", account for both it and delay slot */
    THREAD_FRAME(tcb) = firstframe;
    THREAD_STACK(tcb) = (char *)((int)firstframe - 64);
    ((struct frame *)THREAD_STACK(tcb))->fr_savfp = firstframe;
#endif
#ifdef mips
    /* set up stack so that it looks like:

         real args (target of argp)
         passed args to startrtn (must be put into regs 4 and 5)
            argp
            func
         saved regs (ad hoc area) (80 bytes)
         local variables (28 bytes)
         register 31 (return address = addr(startrtn))
         arg build area (16 bytes)
    */
    argbegin = (int *)&((char *)stackbegin)[-argsize];
    firstframe = (int *)&argbegin[-3];
    if (argsize != 0) {
       bcopy(argp, argbegin, argsize);
       argbegin[-1] = (int)argbegin;
    } else {
       argbegin[-1] = (int)argp;
    }
    argbegin[-2] = (int)func;
    firstframe[-27] = (int)startrtn;
    THREAD_STACK(tcb) = (char *)&firstframe[-32];
#endif
    THREADdynamic_set_size(tcb, dynamic_size);
    THREAD_PRIORITY(tcb) = priority;
    /* component-dependent thread startup routines */

    THREADcontextthreadinit(tcb);

    return(tcb);
}
```

```
/*****************************************************************************/
/* THREADcreate                                                              */
/*                                                                           */
/* Create a new thread and put it on the run queue.                          */
/* The way a thread is started is to create a dummy stack frame, then when   */
/* THREADrun is called, it causes a "return" to the the startup              */
/* routine, which is the first routine of the new thread.                    */
/* Startup then calls the entry point for the new thread that was            */
/* supplied by the caller of create.  When this entry returns,               */
/* startup waits for nondetached children to terminate, then either          */
/* self-destructs if it's detached or synchronizes with the parent           */
/* if not detached.                                                          */
/*                                                                           */
/* args : this should be int*, but it confuses sdb                           */
/*****************************************************************************/


THREAD
THREADcreate (int (*func)(), int *args, int argsize, char detached,
              int stacksize, int priority)
{
    THREAD child;
    int dont_dispatch = 0;

    if (priority < 0) {  /* this is a kludge!! */
        priority = -priority;
        dont_dispatch = 1;
    }

    THREAD_CREATING_SET(THREADcurrent);
    child = THREADmakethread(THREADstartup, stacksize, func, (char*)args,
                             argsize, THREADtype1_extent, priority);
    if (child == 0) {
        logerr(ERROR, "THREADcreate: out of memory");
        return NULL;
    }

    THREAD_TYPE1_THREAD_INIT(child);
    THREADactiveincr;

    THREAD_PROTECT;
    THREADlock(&((THREADcurrent)->context.lock));

    switch(detached) {
      case 0:  /* maintain the relation */
        family_parent(child) = THREADcurrent;
        family_ndchildcount(THREADcurrent)++;
        family_sibling(child) = family_children(THREADcurrent);
        family_children(THREADcurrent) = child;
        break;
      case 1:  /* cut the relation */
        THREAD_DETACHED_SET(child);
        break;
      case 2:  /* why bother this? */
        THREAD_INDEP_SET(child);
        break;
      default:
        logerr(DISASTER, "THREADcreate: detached out of range");
    }

    if (!dont_dispatch) {
        THREADlock(&THREADrunqlock);
        THREAD_MOVETORUNQ(child);  /* this will be handled by THREADrun */
        THREADunlock(&THREADrunqlock);
    }

    THREAD_CREATING_RESET(THREADcurrent);

    if (THREAD_DYING(THREADcurrent)) {
        THREAD_DYING_RESET(THREADcurrent);
        THREADunlock(&((THREADcurrent)->context.lock));
        THREAD_UNPROTECT;
        THREADsuicide();
    } else {
        THREADunlock(&((THREADcurrent)->context.lock));
        THREAD_UNPROTECT;
    }

    return (child);
}



/*****************************************************************************/
/* THREADclonetcb                                                            */
/*                                                                           */
/*****************************************************************************/


THREAD
THREADclonetcb (THREAD tcb, void (*func)(), int *argp, int argsize,
                int priority)
{
    THREAD newtcb;
    int tcbsize;
#ifdef mips
    int *firstframe;
#else !mips
    struct frame *firstframe;
#endif mips
    int *stackbegin;
    int *argbegin;

    tcbsize = sizeof(*tcb) + THREADdynamic_space(tcb->dynamic.size);

    if ((tcb == THREADcurrent) || (tcb == NULL)) {
        /* we are executing on the stack of the cloner, so reserve some
           room for a few more subroutine calls */
        stackbegin = (int *)(GETSTACK() - tcbsize - 1024);
    } else {
#ifndef sparc
        stackbegin = (int *)((int)THREAD_STACK(tcb) - tcbsize);
#else
        stackbegin = (int *)((int)THREAD_FRAME(tcb) - tcbsize);
#endif sparc
    }
    /* the newtcb fits in the caller's stack */
    newtcb = (THREAD)stackbegin;
    bzero(newtcb, tcbsize);
    THREADstacklimit(newtcb) = THREADstacklimit(tcb);

#ifdef vax
    argbegin = (int *)&((char *)stackbegin)[-argsize];
    argbegin = (int *)((int)argbegin & 0xfffffffc);
```

```
    /* first on the stack are the arguments to the startup routine, then
       goes the initial stack frame; we leave room for 1 extra frame */
    /* ???? */
    firstframe = (struct frame *)((int)argbegin - 2*sizeof(struct frame)
                                  - 2*sizeof(int));

    argbegin[-2] = 1;
    if (argsize != 0) {
        bcopy(argp, argbegin, argsize);
        argbegin[-1] = (int)argbegin;
    } else {
        argbegin[-1] = (int)argp;
    }

    firstframe->fr_mask = 0; /* no registers restored */
    firstframe->fr_psw = PSL_USERSET; /* user mode psw */
    firstframe->fr_s = 0; /* pretend that callg was used */
    firstframe->fr_savap = (int)(&argbegin[-2]);
    /* "saved" ap points to args stored at beginning of stack */
    firstframe->fr_savfp = (int)firstframe + sizeof(struct frame);
    firstframe->fr_savpc = (int)func + 2;  /* avoid register mask */
    /* when THREADrun returns to us, we should start execution in func */
    THREAD_FRAME(newtcb) = firstframe;
    THREAD_STACK(newtcb) = (char *)firstframe;
#endif
#ifdef ns32000
    argbegin = (int *)&((char *)stackbegin)[-argsize];
    argbegin = (int *)((int)argbegin & 0xfffffffc);
    /* first on the stack are the arguments to the startup routine, then
       goes the initial stack frame */
    firstframe = (struct frame *)((int)argbegin - sizeof(struct frame)
                                  - sizeof(int));
    if (argsize != 0) {
        bcopy(argp, argbegin, argsize);
        argbegin[-1] = (int)argbegin;
    } else {
        argbegin[-1] = (int)argp;
    }
    firstframe->link = (struct frame *)((int)firstframe +
                                        sizeof(struct frame) - sizeof(int));
    /* "saved" fp points to just before firstframe, so that first frame
       gets itself popped */
    firstframe->ret = (char *)func;
    /* when THREADrun returns to us, we should start execution in
         startrtn */
    THREAD_FRAME(newtcb) = firstframe;
    THREAD_STACK(newtcb) = (char *)firstframe;
#endif
#if defined(sun) && (defined(mc68010) || defined(mc68020))
    argbegin = (int *)&((char *)stackbegin)[-argsize];
    argbegin = (int *)((int)argbegin & 0xfffffffc);

    firstframe = (struct frame *)((int)argbegin - 2*sizeof(struct frame)
                                  - sizeof(int));
    /* arguments go into frame on the sun */
    firstframe->fr_savfp = (struct frame *)argbegin;
    firstframe->fr_savpc = (int)func;
    firstframe->fr_arg[0] = 0; /* dummy */
    if (argsize != 0) {
        bcopy(argp, argbegin, argsize);
        firstframe->fr_arg[1] = (int)argbegin;
    } else {
        firstframe->fr_arg[1] = (int)argp;
    }
    THREAD_FRAME(newtcb) = firstframe;
    THREAD_STACK(newtcb) = (char *)firstframe;
#endif
#ifdef sparc
    argbegin =
        (int *)((unsigned int)&(((char *)stackbegin)[-argsize]) & 0xfffffff8);
    /* put it on a double word boundary */
    firstframe = (struct frame *)((int)argbegin - THREADFRAMESIZE - 32);
    firstframe->fr_savfp = (struct frame *)((int)argbegin);
    firstframe->fr_savpc = 0;
    /* save the arguments, etc, in the "wrong" stack frame: sun's #$%^ C
       compiler clobbers the registers which these would be loaded into,
       so load them later */
    if (argsize != 0) {
        bcopy(argp, argbegin, argsize);
        firstframe->fr_arg[0] = (int)argbegin;
    } else {
        firstframe->fr_arg[0] = (int)argp;
    }
    firstframe = (struct frame *)((int)firstframe - THREADFRAMESIZE - 32);
    firstframe->fr_savfp = (struct frame *)((int)firstframe +
                                            THREADFRAMESIZE + 32);
    firstframe->fr_savpc = (int)func - 8;
    /* saved pc points to "call instruction", account for both it and
       delay slot */
    THREAD_FRAME(newtcb) = firstframe;
    THREAD_STACK(newtcb) = (char *)((int)firstframe - 64);
    ((struct frame *)THREAD_STACK(newtcb))->fr_savfp = firstframe;
#endif
#ifdef mips
    argbegin = (int *)&((char *)stackbegin)[-argsize];
    firstframe = (int *)&argbegin[-2];
    if (argsize != 0) {
        bcopy(argp, argbegin, argsize);
        argbegin[-1] = (int)argbegin;
    } else {
        argbegin[-1] = (int)argp;
    }
    firstframe[-27] = (int)func;
    THREAD_STACK(newtcb) = (char *)&firstframe[-32];
#endif
    THREADdynamic_set_size(newtcb, tcb->dynamic.size);
    THREAD_PRIORITY(newtcb) = priority;
    THREADcontextthreadinit(newtcb);
    THREADexceptionthreadinit(newtcb);
    THREADqueuethreadinit(newtcb);
    THREADrunqthreadinit(newtcb);

    THREAD_FROZEN_PARENT(newtcb) = tcb;
    THREAD_CHILDOFFREEZE_SET(newtcb);
    return(newtcb);
}




/*****************************************************************************/
/* THREADdestroy                                                             */
/*                                                                           */
/*****************************************************************************/
```

```
int
THREADdestroy (THREAD t)
{
    THREAD_PROTECT;

    if (!THREAD_ACTIVE(t)) {
        logerr(DISASTER, "THREADdestroy attemp to destroy inactive thread");
        THREAD_UNPROTECT;
        return 0;
    }

    if (THREAD_CHILDOFFREEZE(t)) {
        THREAD_PASSPROTECT;
        /* pass the buck */
        THREADunfreeze(THREAD_FROZEN_PARENT(THREADcurrent));
        logerr(DISASTER, "THREADdestroy: child thawing parent didn't vanish");
    }


    /*
     * the caller locks tcb. if tcb is the caller, then the runQ is locked
     * by the caller as well
     */
    if ((THREAD_RUNNING(t)) && (t != THREADcurrent)) {
        /* we don't deal with killing a thread running on another processor
           in the lowlevel code */
        logerr(ERROR, "THREADdestroy: attempt to destroy thread running on\
another processor");
        THREAD_UNPROTECT;
        return(0);
    }

    THREAD_DEACTIVATE(t);

    if (t == THREADcurrent) {
        /*
         * since caller no longer exists, we must give control to someone else
         * delay freeing the stack until we are on someone else's stack.
         */
        THREAD_DELETE_SET(t);
        THREAD_PASSPROTECT;
        THREADrun();
        logerr(DISASTER, "THREADdestroy should not be here");
    } else {
          THREAD_FREESTACK(t);
    }

    THREADunlock(&t->context.lock);
    THREAD_UNPROTECT;

    return 1;
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/


#include "thread_type1.h"
#include "tcb.h"
#include "family.h"
#include "manager.h"
#include "thread.h"
#include "logerr.h"

static type_swtch THREADtype1_swtch[THREADMAXTYPES];

int THREADtypecount = 0;
int THREADtype1_extent;


/*****************************************************************************/
/* THREADtype1_register                                                      */
/*****************************************************************************/



int
THREADtype1_register (int size, void (*threadinitfunc)(),
                      void (*threadterminatefunc)(),
                      void (*systeminitfunc)(),
                      void (*systemterminatefunc)())
{
    int offset;

    /* we pretend that someone has registered the family and manager
       extensions */
    if (THREADtypecount == 0) {
        THREADtypecount = THREADpresetdynamics;
        THREADtype1_extent = sizeof(thread_family) + sizeof(thread_manager_hd);
        THREADmanager_offset = sizeof(thread_family);
    }

    if (THREADtypecount >= THREADMAXTYPES) {
        logerr(DISASTER, "THREADtype1_register: too many dynamic types");
    }

    offset = THREADtype1_extent;
    THREADtype1_extent += size;

    THREADtype1_swtch[THREADtypecount].threadinit = threadinitfunc;
    THREADtype1_swtch[THREADtypecount].threadterminate = threadterminatefunc;
    THREADtype1_swtch[THREADtypecount].systeminit = systeminitfunc;
    THREADtype1_swtch[THREADtypecount].systemterminate = systemterminatefunc;
    THREADtypecount++;

    return(offset);
}


/*****************************************************************************/
/* THREADdynamic                                                             */
/*****************************************************************************/

char*
THREADdynamic(THREAD tcb, int offset)
{
    return(THREAD_DYNAMIC(tcb, offset, char));
}

/*****************************************************************************/
/* THREADfinishitoff                                                         */
/*****************************************************************************/

void
THREADfinishitoff (THREAD newtcb)
{
    THREAD_TYPE1_THREAD_INIT(newtcb);
    THREADactiveincr;
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#include <stdio.h>
#include "tsignal.h"
#include "privatedata.h"
#include "mp.h"
#include "runQ.h"
#include "io.h"
#include "thread.h"
#include "preempt.h"
#include "logerr.h"


/*****************************************************************************/
/* the value THREADSIGMASk is assigned is to block                           */
/*   SIGQUIT(3, quit),                                                       */
/*   SIGILL(4, illegal instruction),                                         */
/*   SIGFPE(8, arithmetic exception),                                        */
/*   SIGBUS(10, bus error),                                                  */
/*   SIGSEGV(11, segmentation violation)                                     */
/* it protects us from everything except SIGQUIT and program faults          */
/*                                                                           */
/* for more details about these maskand flag, see sigvec(2) man page         */
/*****************************************************************************/

#define THREADSIGMASK   0xfffff973
#define THREADSIGFLAG   0

static SIGHANDLER sighandlers[NSIG]; /* tell how to deal with UNIX signals */


/*****************************************************************************/
/* THREADsighandler                                                          */
/*                                                                           */
/* this routine is registered with the UNIX kernel and called in response    */
/* to registered signals.  It looks up signal's type in the sighandlers      */
/* array and                                                                 */
/*      SIG_EXCEPTION: raise an exception with sig # to the current thread   */
/*      SIG_SENDSIG: suspend the current thread and run a handler            */
/*      SIG_CREATETHREAD: create a new, detached thread with sig # as argu.  */
/*      SIG_ABORT:                                                           */
/*      default:                                                             */
/*****************************************************************************/



static
void
sighandler (int sig, int code)
{
    SIGHANDLER *sighand;

    if (sighandlers[SIGQUIT].type != SIG_ABORT)
        if (THREADdonttreadonme) {
            THREAD_KEEPSIGNAL(sig);
            return;
        }

    sighand = &sighandlers[sig];

    switch (sighand->type) {
      case SIG_EXCEPTION:
        /* immediately pass the signal to the current thread */
        if (THREADcurrent == THREADmain) {
            logerr(ERROR, "THREADsighandler: exception while idling");
        } else if (THREADcurrent == NULL) {
            logerr(ERROR, "THREADsighandler: exception while idling");
        } else {
            sigsetmask(0);
            if (THREADraiseexception(THREADcurrent, sig) != 1) {
                /* the exception "didn't take" */
            } /* otherwise the exception did take and THREADrun was called */
        }
        break;

      case SIG_SENDSIG:
        {
            THREAD thread;
            THREAD newtcb;
            struct sigparm parms;
            extern void THREADfsiginit();


            if (sighand->whichthread == NULL)
                break;
            if ((thread = (*(sighand->whichthread))(sig, code)) == NULL)
                break;

          lockit:
            THREADlock(&thread->context.lock);
            if ((THREAD_RUNNING(thread)) &&
                (thread->context.processor != THREADwhichprocessor)) {
                /*
                 * we set this thread's sig structure data and SIGVTALRM it
                 * SIGVTALRM's handler checks if sig.sig == 0
                 * if not, doing func (sig)
                 */
                thread->sig.sig = sig;
                thread->sig.code = code;
                thread->sig.func = sighand->func;
                thread->sig.priority = sighand->priority;
                THREADunlock(&thread->context.lock);
                kill(thread->context.processor, SIGVTALRM);
                return;
            } else if (THREAD_RUNNABLE(thread)) {
                if (!THREADcondlock(&THREADrunqlock)) {
                    THREADunlock(&thread->context.lock);
                    goto lockit;
                }
            }

            /* freeze the desired thread */
            if (THREAD_FROZEN(thread)) {
                /* already frozen. We really should now freeze its child,
                   but that requires recursion and sounds too hard.  So
                   we just ignore it.  */
                THREADunlock(&thread->context.lock);
                return;
            }

            THREADfreeze(thread);
```

```
            parms.sig = sig;
            parms.code = code;
            parms.handler = sighand->func;
            newtcb = THREADclonetcb(thread, THREADfsiginit, (int*) &parms,
                                    sizeof(parms), sighand->priority);
            /* call 2ndlevel to initialize its portion of the new tcb */
            THREADfinishitoff(newtcb);
            THREADlock(&newtcb->context.lock);
            THREADlock(&THREADrunqlock);
            THREAD_QMOVETORUNQ(newtcb);
            THREADunlock(&THREADrunqlock);
            THREADunlock(&newtcb->context.lock);
            if (thread == THREADcurrent) {
                /* we now become part of the current thread */
                THREAD_PROTECT;
                sigsetmask(0);
                THREAD_PASSPROTECT;
                THREADlock(&THREADcurrent->context.lock);
                THREADlock(&THREADrunqlock);
                THREADrun();
            }
            break;
        }

      case SIG_CREATETHREAD:
        /* THREAD_PROTECT must be called in malloc */
        THREADcreate(sighand->func, sig, 0, 1, sighand->stacksize,
                     sighand->priority);
        break;

      case SIG_ABORT:
        {
            char errbuf[100];

            sprintf(errbuf, "SIG_ABORT:: pid(%d), THREADn(%d), tcb(%x), sig(%d)\n", get
pid(), THREADn, THREADcurrent, sig);
            write(2, errbuf, strlen(errbuf));

            signal(sig, SIG_IGN);

            sprintf(errbuf, "sighandler: SIGABORT, signal %d", sig);
            if (sig == SIGINT)
                logerr(TERMINATE, errbuf);
            else
                logerr(DISASTER, errbuf);
        }

      default:
        {
            char errbuf[100];

            sprintf(errbuf,
                    "sighandler: invalid signal type, signal %d", sig);
            logerr(WARNING, errbuf);
            break;
        }
    }
    return;
}
```

```
/*****************************************************************************/
/* THREADsiginit                                                             */
/*****************************************************************************/



void
THREADsiginit ()
{
#if defined(sun) || defined(ultrix)
    static struct sigvec defsyshandler = {
        sighandler, THREADSIGMASK, THREADSIGFLAG);
/*
    static struct sigvec timerhandler = {
        (void (*)())THREADtimer, THREADSIGMASK, 0);
*/
    static struct sigvec clockhandler = {
        (void (*)())THREADclockhandler, THREADSIGMASK, 0);
    static struct sigvec sysiohandler = {
        (void (*)())THREADsysiohandler, THREADSIGMASK, 0);
#else
    static struct sigvec defsyshandler = {
        (int (*)())sighandler, THREADSIGMASK, THREADSIGFLAG);
    static struct sigvec timerhandler = {
        (int (*)())THREADtimer, THREADSIGMASK, 0);
    static struct sigvec clockhandler = {
        (int (*)())THREADclockhandler, THREADSIGMASK, 0);
    static struct sigvec sysiohandler = {
        (int (*)())THREADsysiohandler, THREADSIGMASK, 0);
#endif
    static SIGHANDLER defhandler = (SIG_ABORT, NULL, NULL, 0, 0);

    /* broadcast signals */
    sigvec(SIGHUP, &defsyshandler, NULL);
    sighandlers[SIGHUP] = defhandler;
    sigvec(SIGINT, &defsyshandler, NULL);
    sighandlers[SIGINT] = defhandler;
    sigvec(SIGQUIT, &defsyshandler, NULL);
    sighandlers[SIGQUIT] = defhandler;
    sigvec(SIGTERM, &defsyshandler, NULL);
    sighandlers[SIGTERM] = defhandler;

    /* signals which are usually exceptions */
    sigvec(SIGFPE, &defsyshandler, NULL);
    sighandlers[SIGFPE] = defhandler;
    sigvec(SIGBUS, &defsyshandler, NULL);
    sighandlers[SIGBUS] = defhandler;
/*
    sigvec(SIGSEGV, &defsyshandler, NULL);
    sighandlers[SIGSEGV] = defhandler;
*/
    sigvec(SIGSYS, &defsyshandler, NULL);
    sighandlers[SIGSYS] = defhandler;
    sigvec(SIGPIPE, &defsyshandler, NULL);
    sighandlers[SIGPIPE] = defhandler;

    /* signals important for the thread runtime */
/*
    sigvec(SIGALRM, &timerhandler, NULL);
*/
```

```
    sigvec(SIGVTALRM, &clockhandler, NULL);
    sigvec(SIGUSR1, &sysiohandler, NULL);
}



/*****************************************************************************/
/* THREADregistersignal                                                      */
/*****************************************************************************/



int
THREADregistersignal (int sig, SIGHANDLER *newhandler, SIGHANDLER *oldhandler)
{
    int oldmask;

    THREAD_PROTECT;
    oldmask = sigblock(sigmask(sig)); /* block this signal */

    if (oldhandler != NULL)
        *oldhandler = sighandlers[sig];
    if (newhandler != NULL)
        sighandlers[sig] = *newhandler;

    THREAD_UNPROTECT;
    sigsetmask(oldmask);

    return(1);
}


/*****************************************************************************/
/* THRADfakesignal                                                           */
/*                                                                           */
/* An almost (but not quite) analogue of the UNIX kill system call           */
/*****************************************************************************/

void
THREADfakesignal (THREAD thread, void (*func)(), int priority, int sig,
                  int code)
{
    THREAD newtcb;
    struct sigparm parms;
    extern void THREADfsiginit();

    THREAD_PROTECT;

    thread->sig.sig = 0;    /* turn off the pending signal indicator */
    while(1) {
        THREADlock(&thread->context.lock);
        if ((thread != THREADcurrent) && THREAD_RUNNING(thread)) {
            /* force a preemption and it will then notice the pending signal*/
            thread->sig.sig = sig;  /* mark it as having a pending signal */
            thread->sig.code = code;
            thread->sig.func = func;
            thread->sig.priority = priority;
            kill(thread->context.processor, SIGVTALRM);/* force a preemption */
            THREADunlock(&thread->context.lock);
            THREAD_UNPROTECT;
            return;
        }
        if (THREAD_RUNNABLE(thread)) {
            if (THREADcondlock(&THREADrunqlock)) /* a deadlock-free way */
                break;
            } else
                break;
            THREADunlock(&thread->context.lock);
        }

        /* we have locked this thread, and runq if it is on it */

        THREADfreeze(thread);    /* freeze the desired thread */

        /* the thread and the runq are now unlocked */

        parms.sig = sig;
        parms.code = code;
        parms.handler = func;
        newtcb = THREADclonetcb(thread, THREADfsiginit, (int*)&parms,
                                sizeof(parms), priority);

        /* call 2ndlevel to initialize its portion of the new tcb */
        THREADfinishitoff(newtcb);
        THREADlock(&newtcb->context.lock);
        THREADlock(&THREADrunqlock);
        THREAD_QMOVETORUNQ(newtcb);
        THREADunlock(&THREADrunqlock);
        THREADunlock(&newtcb->context.lock);
        if (thread == THREADcurrent) {
            THREAD_PASSPROTECT;
            THREADlock(&THREADcurrent->context.lock);
            THREADlock(&THREADrunqlock);
            THREADrun();
        } else {
            THREAD_UNPROTECT;
        }
}



/*****************************************************************************/
/* THRADsigbroadcast                                                         */
/*                                                                           */
/*****************************************************************************/



void
THREADsigbroadcast (int sig)
{
   int i;

   for (i = 0; i < proc_num; i++) {
       if (i == THREADn)
           continue;
       kill(THREADnpid(i), sig);
   }
}
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef excp_2nd_h
#define excp_2nd_h

#include "tcb.h"
#include "excp.low.h"

extern int THREADexceptioncall ();
extern int THREADgetexceptionreturn ();
extern void THREADrestoreexception (EXCEPTION);
extern EXCEPTION THREADgetexceptionspace ();
extern void THREADfreeexceptionspace (EXCEPTION);

#endif excp_2nd_h
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef excp_low_h
#define excp_low_h


#include "tcb.h"

extern int THREADsetexception (int (*)(), EXCEPTION);
extern int THREADraiseexception (THREAD, int);
extern void THREADexceptionthreadinit (THREAD);

#endif excp_low_h
```

```
/*****************************************************************************/
/*      thread - thread package.                                             */
/*                                                                           */
/*      Copyright 1986 Thomas W. Doeppner Jr. - Brown University             */
/*      - All rights reserved.                                               */
/*****************************************************************************/


#ifndef family_h
#define family_h


#include <setjmp.h>
#include "tcb.h"



/*****************************************************************************/
/* thread_family definition                                                  */
/*****************************************************************************/

typedef struct {
    int status;
    int ndchildcount;   /* combined # of TCB on exitingchildren & chidren */
    int returnvalue;    /* return value of this thread */

    THREAD parent;          /* ptr to the parent thread */
    THREAD exitingchildren; /* ptr to the not-yet-picked-up exiting children */
    THREAD children;        /* ptr to non-detached live children */
    THREAD sibling;         /* ptr to brothers & sisters */

    jmp_buf terminate;
} thread_family;



/*****************************************************************************/
/*****************************************************************************/

#define FAMILY(tcb)  ((thread_family*)((tcb)->dynamic.assignable))

#define family_status(tcb)          ((FAMILY(tcb))->status)
#define family_ndchildcount(tcb)    ((FAMILY(tcb))->ndchildcount)
#define family_returnvalue(tcb)     ((FAMILY(tcb))->returnvalue)
#define family_parent(tcb)          ((FAMILY(tcb))->parent)
#define family_exitingchildren(tcb) ((FAMILY(tcb))->exitingchildren)
#define family_children(tcb)        ((FAMILY(tcb))->children)
#define family_sibling(tcb)         ((FAMILY(tcb))->sibling)
#define family_terminate(tcb)       ((FAMILY(tcb))->terminate)


/* thread is waiting for a child to terminate */
#define THREAD_WAITCHILD_M      0x1
#define THREAD_WAITCHILD(tcb)   (family_status((tcb)) & THREAD_WAITCHILD_M)
#define THREAD_WAITCHILD_SET(tcb) family_status((tcb)) |= THREAD_WAITCHILD_M
#define THREAD_WAITCHILD_RESET(tcb) family_status((tcb)) &= ~THREAD_WAITCHILD_M


/* nondetached thread has terminated, has been "waited for" by parent, */
/* and now needs to be eliminated                                      */
#define THREAD_DONE_M           0x2
#define THREAD_DONE(tcb)        (family_status((tcb)) & THREAD_DONE_M)
#define THREAD_DONE_SET(tcb)    family_status((tcb)) |= THREAD_DONE_M
#define THREAD_DONE_RESET(tcb)  family_status((tcb)) &= ~THREAD_DONE_M


/* thread is detached from its parent - the two are no longer related */
#define THREAD_DETACHED_M       0x4
#define THREAD_DETACHED(tcb)    (family_status((tcb)) & THREAD_DETACHED_M)
#define THREAD_DETACHED_SET(tcb) family_status((tcb)) |= THREAD_DETACHED_M


/* a nondetached thread is exiting */
#define THREAD_EXITING_M        0x8
#define THREAD_EXITING(tcb)     (family_status((tcb)) & THREAD_EXITING_M)
#define THREAD_EXITING_SET(tcb) family_status((tcb)) |= THREAD_EXITING_M


/* the thread has been marked for murder */
#define THREAD_MURDERED_M       0x10
#define THREAD_MURDERED(tcb)    (family_status((tcb)) & THREAD_MURDERED_M)
#define THREAD_MURDERED_SET(tcb) family_status((tcb)) |= THREAD_MURDERED_M


/* the thread is reproducing */
#define THREAD_CREATING_M       0x20
#define THREAD_CREATING(tcb)    (family_status((tcb)) & THREAD_CREATING_M)
#define THREAD_CREATING_SET(tcb) family_status((tcb)) |= THREAD_CREATING_M
#define THREAD_CREATING_RESET(tcb) family_status((tcb)) &= ~THREAD_CREATING_M


/* thread has no parent, but may be "waited for" */
#define THREAD_INDEP_M          0x40
#define THREAD_INDEP(tcb)       (family_status((tcb)) & THREAD_INDEP_M)
#define THREAD_INDEP_SET(tcb) family_status((tcb)) |= THREAD_INDEP_M
#define THREAD_INDEP_RESET(tcb) family_status((tcb)) &= ~THREAD_INDEP_M



/*****************************************************************************/
/* external declaration                                                      */
/*****************************************************************************/

extern void THREADstartup (int (*)(), char *[]);
extern THREAD THREADwaitforchild ();
extern int THREADeliminatechild ();
extern int THREADreturnvalue (THREAD);
extern void THREADmurder (THREAD);
extern void THREADsuicide ();

#endif family_h
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef frame_h
#define frame_h


#include <frame.h>


#define THREADFRAMESIZE  64



#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */

/*****************************************************************************/

#ifndef io_h
#define io_h

#include <sys/param.h>
#include <sys/time.h>
#include <sys/types.h>
#include "lock.h"


#ifdef sparc
#define POLLINTERVAL    1
#else
#define POLLINTERVAL    5
#endif sparc

extern LOCK THREADiotimer;
extern struct timeval THREADnowait;
extern int sfd[];


void THREADcopyfdset (fd_set*, fd_set*);
void THREADcopyfdsetlimited (fd_set*, fd_set*, int);

extern void THREADsysiohandler ();


#define THREADtestandsetiotimer(timer)       THREADcondlock(&(timer))
#define THREADcleariotimer(timer)            THREADunlock(&(timer))




#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef loadfunc_only_h
#define loadfunc_only_h

/*****************************************************************************/
/* this file *DEFINE* some data specific to be included by loadfunc.         */
/* these data are to be used by the to-be-loaded executable.                 */
/*                                                                           */
/*****************************************************************************/

#include "privatedata.h"

#ifndef private_share_m
int THREADn; /* this variable is for thread package use. */
#else
struct private_data processor_private;
#endif


#endif
```

```
#ifndef lock_h
#define lock_h


typedef int LOCK;

#define LOCKED     1
#define UNLOCKED   0



/*****************************************************************************/
/* these two functions' source code were written in ASSEMBLY, specifically */
/* using the SUN SPARC 10's atomic instruction 'swap'                        */
/*                                                                           */
/*****************************************************************************/

extern void swaplock (LOCK *);
extern void swapunlock (LOCK *);
extern int swapcondlock (LOCK *);

#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef logerr_h
#define logerr_h

#include "lock.h"

#define INFO            1
#define WARNING         2
#define ERROR           3
#define DISASTER        4
#define TERMINATE       5



/*****************************************************************************/
/* external functions & variables                                            */
/*****************************************************************************/

extern void logerr (int, char *);
extern LOCK THREADdying;

#endif logerr_h
```

```
/*****************************************************************************/
/*      thread - thread package.                                             */
/*                                                                           */
/*      Copyright 1986 Thomas W. Doeppner Jr. - Brown University             */
/*      - All rights reserved.                                               */
/*****************************************************************************/

#ifndef manager_h
#define manager_h



#include "tcb.h"
/*****************************************************************************/
/* struct manager_ops                                                        */
/*                                                                           */
/* when a thread tries to grab a monitor, it allocates manager associated    */
/* with the monitor. After exceptions occur, it calls the corresponding      */
/* func.                                                                     */
/*****************************************************************************/

struct manager_block;
struct manager_ops {
    int id;                          /* ABORT, EXCEPTION ... */
    void (*func)(THREAD, struct manager_block *);
};


/*****************************************************************************/
/* struct manager_block                                                      */
/*****************************************************************************/

typedef struct manager_block {
    int status;              /* ON_STACK_M ...*/
    char *excpptr;           /* save the exception stack so to *longjmp* */
    struct manager_block *forward;
    struct thread_monitor_block *data;     /* which monitor it serves on */
    int num_of_ops;
    struct manager_ops ops[2];
} *THREAD_MANAGER, THREAD_MANAGER_BLOCK;




/*****************************************************************************/
/*****************************************************************************/

typedef struct {
    THREAD_MANAGER newest;
} thread_manager_hd;


#define MANAGER_ABORT     0
#define MANAGER_EXCEPTION 1

#define manager_onstack_M       0x1

#define manager_onstack(manager)       ((manager)->status & manager_onstack_M)
#define manager_onstack_set(manager)   ((manager)->status |= manager_onstack_M)
#define manager_onstack_reset(man)     ((man)->status &= ~manager_onstack_M)

#define manager_forward(manager)              (manager)->forward
#define THREADmanagerdata(manager)            (manager)->data

#define THREADmanagerexcpptr(manager)         (manager)->excpptr

#define manager_newest(t) THREAD_DYNAMIC(t, THREADmanager_offset,\
                                         thread_manager_hd)->newest

#define MANAGER_IN_SCOPE(manager, tcb)                                    \
    (THREADmanagerexcpptr(manager) <= THREADexception_stack(tcb))
     /* manager points to exception handler stack location active at the
        time the manager was created */


/*****************************************************************************/
/* external variables and functions                                          */
/*****************************************************************************/


extern int THREADmanager_offset;


extern void MANAGER_RELEASE (THREAD);


#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef monitor_h
#define monitor_h



#include "tcb.h"
#include "runQ.h"
#include "queue.h"
#include "manager.h"



/*****************************************************************************/
/* struct thread_monitor_condition                                           */
/*****************************************************************************/

typedef struct thread_monitor_condition {
   int delayed_signal_count;
   THREAD_QUEUE_HEAD queue;
} *THREAD_MONITOR_CONDITION, THREAD_MONITOR_CONDITION_HEAD;



/*****************************************************************************/
/* struct thread_monitor_block                                               */
/*                                                                           */
/* resetfunc is called if an exception is raised in a thread that is         */
/* active? in the monitor                                                    */
/*****************************************************************************/

typedef struct thread_monitor_block {
    int status;
    THREAD current;          /* the tcb grabbing this monitor */
    THREAD_QUEUE_HEAD queue; /* candidate tcbs will be on the queue */
    void (*resetfunc)();
    short nr_of_conditions;  /* how many conditions does it have */
    THREAD_MONITOR_CONDITION_HEAD conditions[1];
} *THREAD_MONITOR, THREAD_MONITOR_BLOCK;



/*****************************************************************************/
/* macros                                                                    */
/*****************************************************************************/

#define monitor_active(monitor)            ((monitor)->current != NULL)
#define monitor_clear_active(monitor)      (monitor)->current = NULL;
#define monitor_set_active(monitor, tcb)   (monitor)->current = tcb;
#define monitor_thread(monitor)            (monitor)->current
#define monitor_queue_lock(monitor)        ((monitor)->queue.lock)
#define monitor_reset(monitor)             (monitor)->resetfunc
#define monitor_nr_of_conditions(monitor)  (monitor)->nr_of_conditions
#define monitor_condition(monitor, i)      (monitor)->conditions[i]



/*****************************************************************************/
/* external variables and functions                                          */
/*****************************************************************************/

extern THREAD_MONITOR THREADmonitorinit (int, void (*)());
extern int THREADmonitorentry (THREAD_MONITOR, THREAD_MANAGER);
extern int THREADmonitorexit (THREAD_MONITOR);
extern int THREADmonitorwait (THREAD_MONITOR, int);
extern int THREADmonitorsiganlandexit (THREAD_MONITOR, int);
extern int THREADmonitorsignalandwait (THREAD_MONITOR, int, int);
extern void THREADmonitorabort (THREAD, THREAD_MANAGER);

#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef mp_h
#define mp_h

#include "tcb.h"
#include "privatedata.h"

extern int proc_num;
extern int THREADfirstpipe;

#ifndef private_share_m
#define THREADnpid(n)  (((n>=0) && (n<proc_num)) \
                        ? THREADprocs[n].whichprocessor : -1)
#define THREADnsigpause(n) THREADprocs[n].sigpause
#else
extern int THREADproc_sigpause[];
extern int THREADproc_pid[];
#define THREADnpid(n)  (((n>=0) && (n<proc_num)) \
                        ? THREADproc_pid[n] : -1)
#define THREADnsigpause(n) THREADproc_sigpause[n]
#endif
#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#ifndef preempt_h
#define preempt_h

#include "privatedata.h"
#include "logerr.h"

/*****************************************************************************/
/* macro                                                                     */
/*****************************************************************************/

#define THREAD_PROTECT {(THREADdonttreadonme)++; (THREADprotcount)++;}
#define THREAD_UNPROTECT                                                    \
        {                                                                   \
           if (--(THREADprotcount) <= 0) {                                  \
              if (THREADpendingsignals) {                                   \
                 int sig;                                                   \
                 int omask;                                                 \
                                                                            \
                 /* One or more signals have occured while we had them     \
                    "blocked".  We want to make all of them happen again,   \
                    so block them all, then reissue the signals, then unblock \
                    them.  This rigamarole is necessary because a signal    \
                    handler might cause a thread switch, which might mean that\
                    it will be awhile before we return here.                \
                 */                                                         \
                 omask = sigblock(0xfffff973);                              \
                 while ((sig = ffs(THREADpendingsignals)) > 0) {            \
                    THREADpendingsignals &= ~(1 << (sig-1));                \
                    kill(THREADwhichprocessor, sig);                        \
                 }                                                          \
                 THREADprotcount = 0;                                       \
                 THREADdonttreadonme = 0;                                   \
                 sigsetmask(omask);                                         \
              } else {                                                      \
                 THREADprotcount = 0;                                       \
                 THREADdonttreadonme = 0;                                   \
              }                                                             \
           }                                                                \
        }

#define THREAD_PASSPROTECT                                                  \
        {                                                                   \
           if (THREADdonttreadonme == 0)                                    \
              logerr(DISASTER, "passprotect");                              \
           THREADprotcount = 0;                                             \
        }

#define THREAD_KEEPSIGNAL(sig) (THREADpendingsignals |= 1 << (sig - 1);)


/*****************************************************************************/
/* export variables and functions                                            */
/*****************************************************************************/

extern void THREADclockhandler ();
extern void THREADprotect ();
extern void THREADunprotect ();

#endif preempt_h
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef privatedata_h
#define privatedata_h

struct thread_control_block; /* forward declaration */

struct private_data {
    int whichprocessor;   /* this will be the pid of the main thread */
    struct thread_control_block *main; /* created on stack, never be on runq */
    struct thread_control_block *current; /* thread running on this processor*/
    struct thread_control_block *last; /* last thread running on this */
    int pindex;
    int donttreadonme; /* != 0 means being protected; i.e. not swtched out */
    int protcount;
    int pendingsignals;
    int Xtimerisset; /* how many event waiting on this processor */
    int sigpause;    /* 0: in sigpause, 1: not */
};

/*****************************************************************************/
/* in shared memory model, there are two places to put the private data      */
/*   (1) put all those data (THREADprocs[]) on global share region, and     */
/*       have THREADn in private data to serve as an index to THREADprocs[]*/
/*   (2) put the private data on private data region, namely processor_data*/
/*                                                                           */
/* private data region : global data in loadfunc.c                           */
/* global share region : global data in all files called in from loadfunc   */
/*****************************************************************************/

#ifndef private_share_m
extern struct private_data THREADprocs[];
extern int THREADn;
#define THREADcurrent    (THREADprocs[THREADn].current)
#define THREADlast       (THREADprocs[THREADn].last)
#define THREADmain       (THREADprocs[THREADn].main)
#define THREADwhichprocessor (THREADprocs[THREADn].whichprocessor)
#define THREADpindex         (THREADprocs[THREADn].pindex)
#define THREADdonttreadonme  (THREADprocs[THREADn].donttreadonme)
#define THREADprotcount      (THREADprocs[THREADn].protcount)
#define THREADpendingsignals (THREADprocs[THREADn].pendingsignals)
#define THREADtimerisset     (THREADprocs[THREADn].Xtimerisset)
#else
extern struct private_data processor_private;
#define THREADcurrent    (processor_private.current)
#define THREADlast       (processor_private.last)
#define THREADmain       (processor_private.main)
#define THREADwhichprocessor (processor_private.whichprocessor)
#define THREADpindex         (processor_private.pindex)
#define THREADdonttreadonme  (processor_private.donttreadonme)
#define THREADprotcount      (processor_private.protcount)
#define THREADpendingsignals (processor_private.pendingsignals)
#define THREADtimerisset     (processor_private.Xtimerisset)
#endif

#endif privatedata_h
```

```
/************************************************************************/
/*        thread - thread package.                                      */
/*                                                                      */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University      */
/*        - All rights reserved.                                        */
/************************************************************************/

#ifndef queue_h
#define queue_h

/************************************************************************/
/* include files & macros                                               */
/************************************************************************/

#include "tcb.h"



/************************************************************************/
/* struct thread_queue                                                  */
/*                                                                      */
/* a circular list. last will point to the last item. Easy to append,   */
/* insert, retreat                                                      */
/************************************************************************/

struct thread_queue {
    THREAD last;
    LOCK   lock;
};
typedef struct thread_queue THREAD_QUEUE_HEAD;
typedef struct thread_queue* THREAD_QUEUE;



/************************************************************************/
/* external variables and functions                                     */
/************************************************************************/

extern void THREADqueueInit (THREAD_QUEUE);
extern void THREAD_TCBAPPEND (THREAD, THREAD_QUEUE);
extern void THREAD_TCBINSERT (THREAD, THREAD_QUEUE);
extern THREAD THREADqueueGet (THREAD_QUEUE);
extern THREAD THREADqueuePeep (THREAD_QUEUE);

#endif queue_h
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef runQ_h
#define runQ_h


#include "queue.h"
#include "tcb.h"

/*****************************************************************************/
/* external variables and functions                                          */
/*****************************************************************************/

extern THREAD_QUEUE_HEAD runQ[];
extern LOCK THREADrunqlock;
extern THREAD_QUEUE_HEAD waitq;
extern int whichQ;
extern THREAD_QUEUE_HEAD freeit_queue;

extern void runQInit ();
extern void THREADrun ();
extern void THREAD_MOVETORUNQ (THREAD);
extern THREADrunqthreadinit (THREAD);
extern THREADqueuetheadinit (THREAD);


#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/



#ifndef semaphore_h
#define semaphore_h

#include "queue.h"

typedef struct semaphore {
   int  value;
   THREAD_QUEUE_HEAD queue;
} SEMBODY, *SEMAPHORE;

extern void THREADpsem();
extern void THREADvsem();
extern SEMAPHORE THREADseminit();


#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef stack_h
#define stack_h



#include "frame.h"
#include "tcb.h"
#include "mp.h"


/*****************************************************************************/
/* external functions and variables                                          */
/*****************************************************************************/

extern struct frame *setStack (void (*)(int(*)(), char*[]), char *, int (*)(),
                               char*, int);
extern STACK    *THREADstackspace;
extern STACK    *THREADstackspaceend;
extern int      THREADstdstacksize;
extern int      THREADnstacks;
extern STACK    *THREADfreestacks;

extern int      THREADcompstack (THREAD, char*);


/* STACK is defined in tcb.h */


#define THREADstackend          0x092239 /* serve as a magic number */
#define THREADstackfree         0x090551



#define THREAD_STACKOUTOFBOUNDS(tcbp)                                   \
    (((char *)THREAD_FRAME(tcbp) < (char *)(tcbp)->stack.stack_limit) ||    \
         (((tcbp)->stack.stack_limit != NULL) &&                        \
          ((tcbp)->stack.stack_limit->magic_number != THREADstackend)))


/*****************************************************************************/
/* macro                                                                     */
/*****************************************************************************/


#ifdef sun
#define THREAD_ALLOCATESTACK(tcbp, tcbsize, size, stackbegin, type)     \
{                                                                       \
   STACK *stackend;                                                     \
   int *magic;                                                          \
                                                                        \
   THREAD_PROTECT;                                                      \
                                                                        \
   if ((THREADfreestacks == NULL) || (size > THREADstdstacksize)        \
        || (THREADstdstacksize == 0)) {                                 \
      stackend = (STACK *)_memalign(4, size);                           \
      stackend->magic_number = THREADstackend;                          \
   } else {                                                             \
      size = THREADstdstacksize;                                        \
      stackend = THREADfreestacks;                                      \
      THREADfreestacks = stackend->next;                                \
      magic = &(stackend->magic_number);                                \
      if (*magic != THREADstackfree)                                    \
         logerr(DISASTER, "THREADallocatestack: munged free stack");    \
      else                                                              \
         *magic = THREADstackend;                                       \
   }                                                                    \
                                                                        \
   THREAD_UNPROTECT;                                                    \
                                                                        \
   if (stackend == NULL)                                                \
      stackbegin = NULL;                                                \
   else {                                                               \
      stackend->magic_number = THREADstackend;                          \
      /* the tcb fits in the high-address region of the stack */        \
      stackbegin = (type)&((char *)stackend)[size-tcbsize];             \
      (tcbp) = (THREAD)stackbegin;                                      \
      bzero(tcbp, tcbsize);                                             \
      (tcbp)->stack.stack_limit = stackend;                             \
      (tcbp)->stack.stacksize = size-tcbsize;                           \
   }                                                                    \
}
#else
#define THREAD_ALLOCATESTACK(tcbp, tcbsize, size, stackbegin, type)     \
{                                                                       \
   STACK *stackend;                                                     \
   int *magic;                                                          \
                                                                        \
   THREAD_PROTECT;                                                      \
                                                                        \
   THREADlock(&THREADstacklock);                                        \
   if ((THREADfreestacks == NULL) || (size > THREADstdstacksize)        \
        || (THREADstdstacksize == 0)) {                                 \
      THREADunlock(&THREADstacklock);                                   \
      stackend = (STACK *)_malloc(size);                                \
   } else {                                                             \
      size = THREADstdstacksize;                                        \
      stackend = THREADfreestacks;                                      \
      THREADfreestacks = stackend->next;                                \
      THREADunlock(&THREADstacklock);                                   \
      magic = &(stackend->magic_number);                                \
      if (*magic != THREADstackfree)                                    \
         logerr(DISASTER, "THREADallocatestack: munged free stack");    \
      else                                                              \
         *magic = THREADstackend;                                       \
   }                                                                    \
                                                                        \
   THREAD_UNPROTECT;                                                    \
                                                                        \
   if (stackend == NULL)                                                \
      stackbegin = NULL;                                                \
   else {                                                               \
      stackend->magic_number = THREADstackend;                          \
      /* the tcb fits in the high-address region of the stack */        \
      stackbegin = (type)&((char *)stackend)[size-tcbsize];             \
      (tcbp) = (THREAD)stackbegin;                                      \
      qbzero(tcbp, tcbsize);                                            \
      (tcbp)->stack.stack_limit = stackend;                             \
      (tcbp)->stack.stacksize = size-tcbsize;                           \
   }                                                                    \
}
```

```
#endif

#endif stack_h
```

```
/*****************************************************************************/
/*      thread - thread package.                                             */
/*                                                                           */
/*      Copyright 1986 Thomas W. Doeppner Jr. - Brown University             */
/*      - All rights reserved.                                               */
/*****************************************************************************/

#ifndef tcb_h
#define tcb_h

/*****************************************************************************/
/* include files & macros                                                    */
/*****************************************************************************/

#include "lock.h"
#include "frame.h"



/*****************************************************************************/
/* forward declaration                                                       */
/*****************************************************************************/

struct thread_queue;



/*****************************************************************************/
/* struct exception_state                                                    */
/*                                                                           */
/* when an exception was raised, the control will be rolled back to where    */
/* the exception was established. exception_stack keeps the stack ptr ?      */
/* and exception_frame keeps the frame (why of size THREADFRAMESIZE/4 ?)     */
/*****************************************************************************/

struct exception_state {
    int exception_frame[THREADFRAMESIZE/4];
    char *exception_stack;
    int (*exception_handler)();
};
typedef struct exception_state *EXCEPTION;



/*****************************************************************************/
/* struct stack                                                              */
/*                                                                           */
/* if you don't call THREADstackspaceinit, then just ignore these stuff      */
/*****************************************************************************/

struct stack{
    int magic_number;   /* serves as a check point */
    struct stack *next; /* while running out of stack, we can link another? */
    char rest[4];       /* ??? */
};
typedef struct stack STACK;



/*****************************************************************************/
/* struct thread_control_block                                               */
/*****************************************************************************/

struct thread_control_block {
    int system_error_code;   /* must be first - cerror.c depends on it */
    struct {
        LOCK lock;
        int status;
        int priority;
        int porder;          /* the order of the processes */
        int processor;       /* running on which processor, i.e. pid */
        struct frame *frame; /* keep the current when being preempted */
        char *stack;
    } context;
    struct {
        int status;
        struct thread_control_block *next;
        struct thread_queue *queue_head;   /* pointing the queue it's on */
        struct thread_control_block *frozenparent;
    } queue;
    struct {
        struct exception_state excp_state;
        int exception_param; /* got from raiseexcetion  &passed to handler */
        char in_progress;    /* 0: not in exception, o/w in exception */
    } exception;
    struct {
        STACK *stack_limit;
        int stacksize;
    } stack;
    struct {
        int sig;
        int code;
        int priority;
        void (*func)();
    } sig;
    struct {
        int size;
        char assignable[4];  /* must be the last - dynamic memory */
    } dynamic;
};
typedef struct thread_control_block *THREAD;



/*****************************************************************************/
/* definition for context structure                                          */
/*****************************************************************************/

#define THREAD_PRIORITY(tcb)    ((tcb)->context.priority)
#define THREAD_FRAME(tcb)       ((tcb)->context.frame)
#define THREAD_STACK(tcb)       ((tcb)->context.stack)
#define THREAD_PORDER(tcb)      ((tcb)->context.porder)

#define THREAD_ACTIVE_M         0x1
#define THREAD_EXCEPTION_M      0x2
#define THREAD_RUNNABLE_M       0x4
#define THREAD_RUNNING_M        0x8
#define THREAD_DYING_M          0x10
#define THREAD_IO_M             0x20
#define THREAD_FORCE_SWITCH_M   0x40
#define THREAD_RERUN_M          0x80
#define THREAD_DELETE_M         0x100
#define THREAD_FROZEN_M         0x200 /* originally defined on queue.status */
#define THREAD_IN_QUEUE_M       0x400 /* originally defined on queue.status */
#define THREAD_FREEZE_M         0x800 /* originally defined on queue.status */
#define THREAD_CHILDOFFREEZE_M  0x1000 /* originally defined on queue.status */
#define THREAD_IN_WAIT_M        0x2000 /* see THREADmonitorwaitevent */
```

```
#define THREAD_ACTIVE(tcb)       ((tcb)->context.status & THREAD_ACTIVE_M)
#define THREAD_ACTIVATE(tcb)     ((tcb)->context.status |= THREAD_ACTIVE_M)
#define THREAD_DEACTIVATE(tcb)   ((tcb)->context.status &= ~THREAD_ACTIVE_M)

#define THREAD_EXCEPTION(tcb)    ((tcb)->context.status & THREAD_EXCEPTION_M)
#define THREAD_EXCEPTION_SET(tcb) ((tcb)->context.status |= THREAD_EXCEPTION_M)
#define THREAD_EXCEPTION_RESET(t) ((t)->context.status &= ~THREAD_EXCEPTION_M)

#define THREAD_RUNNABLE(tcb)     ((tcb)->context.status & THREAD_RUNNABLE_M)
#define THREAD_RUNNABLE_SET(tcb) (tcb)->context.status |= THREAD_RUNNABLE_M
#define THREAD_RUNNABLE_RESET(tcb) (tcb)->context.status &= ~THREAD_RUNNABLE_M

#define THREAD_RUNNING(tcb)      ((tcb)->context.status & THREAD_RUNNING_M)
#define THREAD_RUNNING_SET(tcb)  (tcb)->context.status |= THREAD_RUNNING_M
#define THREAD_RUNNING_RESET(tcb) (tcb)->context.status &= ~THREAD_RUNNING_M

#define THREAD_DYING(tcb)        ((tcb)->context.status & THREAD_DYING_M)
#define THREAD_DYING_SET(tcb)    (tcb)->context.status |= THREAD_DYING_M
#define THREAD_DYING_RESET(tcb)  (tcb)->context.status &= ~THREAD_DYING_M

#define THREAD_IO(tcb)           ((tcb)->context.status & THREAD_IO_M)
#define THREAD_IO_SET(tcb)       (tcb)->context.status |= THREAD_IO_M
#define THREAD_IO_RESET(tcb)     (tcb)->context.status &= ~THREAD_IO_M

#define THREAD_FORCE_SWITCH(t) ((t)->context.status & THREAD_FORCE_SWITCH_M)
#define THREAD_FORCE_SWITCH_SET(tcb) ((tcb)->context.status |=\
                                THREAD_FORCE_SWITCH_M)
#define THREAD_FORCE_SWITCH_RESET(tcb) ((tcb)->context.status &=\
                                 ~THREAD_FORCE_SWITCH_M)

#define THREAD_RERUN(tcb)        ((tcb)->context.status & THREAD_RERUN_M)
#define THREAD_RERUN_SET(tcb)    ((tcb)->context.status |= THREAD_RERUN_M)
#define THREAD_RERUN_RESET(tcb)  ((tcb)->context.status &= ~THREAD_RERUN_M)

#define THREAD_DELETE(tcb)        ((tcb)->context.status & THREAD_DELETE_M)
#define THREAD_DELETE_SET(tcb)    ((tcb)->context.status |= THREAD_DELETE_M)
#define THREAD_DELETE_RESET(tcb)  ((tcb)->context.status &= ~THREAD_DELETE_M)

#define THREAD_FROZEN(tcb)        ((tcb)->context.status & THREAD_FREEZE_M)
#define THREAD_FREEZE_SET(tcb)    ((tcb)->context.status |= THREAD_FREEZE_M)
#define THREAD_FREEZE_RESET(tcb)  ((tcb)->context.status &= ~THREAD_FREEZE_M)

#define THREAD_CHILDOFFREEZE(t) ((t)->context.status & THREAD_CHILDOFFREEZE_M)
#define THREAD_CHILDOFFREEZE_SET(t) ((t)->context.status |=\
                               THREAD_CHILDOFFREEZE_M)
#define THREAD_CHILDOFFREEZE_RESET(t)  ((t)->context.status &=\
                                ~THREAD_CHILDOFFREEZE_M)

#define THREAD_IN_WAIT(tcb)       ((tcb)->context.status & THREAD_IN_WAIT_M)
#define THREAD_IN_WAIT_SET(tcb)   ((tcb)->context.status |= THREAD_IN_WAIT_M)
#define THREAD_IN_WAIT_RESET(tcb) ((tcb)->context.status &= ~THREAD_IN_WAIT_M)

#define THREADcontextthreadinit(t) ((t)->context.status = THREAD_ACTIVE_M;)



/*************************************************************************/
/* definition for exception structure                                    */
/*************************************************************************/

#define THREADexception_state(tcb)  (tcb)->exception.excp_state
#define THREADexception_frame(tcb)  (tcb)->exception.excp_state.exception_frame
#define THREADexception_stack(tcb)  (tcb)->exception.excp_state.exception_stack
#define THREADexception_handler(t)  (t)->exception.excp_state.exception_handler
#define THREADexception_param(tcb)  (tcb)->exception.exception_param
#define THREADexception_inprogress(tcb)   (tcb)->exception.in_progress



/*************************************************************************/
/* definition for sig structure                                          */
/*************************************************************************/

#define THREAD_SIG(t)  (t->sig.sig)



/*************************************************************************/
/* definition for dynamic structure                                      */
/*************************************************************************/

#define THREAD_DYNAMIC(tcb, offset, type) \
       ((type*)(&((tcb)->dynamic.assignable[offset])))

#define THREADdynamic_space(size)         (size - sizeof(char[4]))
#define THREADdynamic_set_size(tcb, new)  (tcb)->dynamic.size = new



/*************************************************************************/
/* definition for queue                                                  */
/*************************************************************************/

#define THREAD_QUEUE_NEXT(tcb)  ((tcb)->queue.next)
#define THREAD_WHICH_QUEUE(t)   ((t->queue).queue_head)
#define THREAD_FROZEN_PARENT(t) ((t)->queue.frozenparent)



/*************************************************************************/
/* definition for stack                                                  */
/*************************************************************************/

#define THREADstacklimit(tcb)          (tcb)->stack.stack_limit



/*************************************************************************/
/* Wrapper                                                               */
/*************************************************************************/

#define THREADlock(x)       swaplock(x)
#define THREADunlock(x)     swapunlock(x)
#define THREADcondlock(x)   swapcondlock(x)
#define THREADsetlock(x)    *x = LOCKED;
#define THREADclearlock(x)  *x = UNLOCKED;


#endif
```

```
/*

 thread - thread package.

 Copyright 1986 Thomas W. Doeppner Jr. - Brown University - All rights reserved.

*/

#ifndef _THREADS_
#define _THREADS_

/* opaque types */
/* (I.e, none of your goddam business) */

typedef struct thread_control_block *THREAD;
typedef struct thread_queue *THREAD_QUEUE;
typedef struct {
   int guts[2];
} THREAD_QUEUE_HEAD;

typedef struct semaphore *SEMAPHORE;

typedef struct exception_state *EXCEPTION;
typedef struct {
#ifdef mips
   int guts[40];
#else
   int guts[20];
#endif mips
} EXCEPTION_BLOCK;

typedef struct thread_monitor_condition *THREAD_MONITOR_CONDITION;
typedef struct {
   int guts[4];
} THREAD_MONITOR_CONDITION_HEAD;

typedef struct thread_monitor_block *THREAD_MONITOR;
typedef struct {
   int guts[25];
} THREAD_MONITOR_BLOCK;

typedef struct thread_manager_block *THREAD_MANAGER;
typedef struct {
   int guts[10];
} THREAD_MANAGER_BLOCK;

typedef struct sighandler {
   int type;
   THREAD (*whichthread)();
   void (*func)();
   int stacksize;
   int priority;
} SIGHANDLER;

#define SIG_EXCEPTION          1
#define SIG_SENDSIG            2
#define SIG_CREATETHREAD       3

typedef struct uthreadcontrol *UTHREAD_CONTROL;

/* entry points */

#ifdef abetterworld
extern void _free();
#endif
extern char *_malloc();
extern char *_memalign();
extern char *realloc();
extern int THREADaccept();
extern int THREADanyonewantthecpu();
extern void THREADbegin();
extern char *THREADbeginstack();
extern void THREADclockhandler();
extern int THREADclose();
extern int THREADconnect();
extern THREAD THREADcreate();
extern char *THREADdynamic();
extern int THREADeliminatechild();
extern void THREADend();
extern char *THREADendstack();
extern THREAD THREADexternal();
extern void THREADfakesignal();
extern void THREADfreeexceptionspace();
extern int THREADfrozen();
extern THREAD THREADfrozenparent();
extern int THREADgetexceptionreturn();
extern EXCEPTION THREADgetexceptionspace();
extern THREAD_QUEUE THREADmakequeuehead();
extern int THREADmonitorentry();
extern int THREADmonitorexit();
extern void THREADmonitorfree();
extern THREAD_MONITOR THREADmonitorinit();
extern int THREADmonitorsignalandexit();
extern int THREADmonitorwait();
extern int THREADmonitorwaitevent();
extern int THREADmovetorunq();
extern int THREADmovetowaitq();
extern void THREADmurder();
extern int THREADopen();
extern THREAD THREADparent();
extern void THREADprotect();
extern void THREADpsem();
extern THREAD THREADpullfromq();
extern THREAD THREADqueuenext();
extern int THREADraiseexception();
extern int THREADread();
extern int THREADregistersignal();
extern void THREADreschedule();
extern void THREADrestoreexception();
extern int THREADreturnvalue();
extern SEMAPHORE THREADseminit();
extern void THREADseterrorlevel();
extern int THREADsetexception();
extern void THREADsetpriority();
extern void THREADsetnononblockingio();
extern void THREADsignotify();
extern int THREADstackremaining();
extern int THREADstackspaceinit();
extern void THREADstartclock();
extern void THREADstdiorelease();
extern void THREADstdiotake();
extern void THREADstopclock();
extern void THREADsuicide();
extern void THREADtellmanagement();
extern void THREADunprotect();
extern UTHREAD_CONTROL THREADuthread_create();
extern void THREADuthread_go();
extern void THREADuthread_kill();
```

```
extern void THREADuthread_park();
extern void THREADuthread_ready();
extern void THREADvsem();
extern THREAD THREADwaitforchild();
extern int THREADwrite();

/* well-known globals */

#include "/u/tch/project/src/privatedata.h"

/* useful macros */

#define THREADinq_current()  THREADcurrent

extern int errno;

#define THREADerrno                                                    \
        ((THREADcurrent != NULL) ? (*(int *)THREADcurrent) : errno)

#define THREADseterrno(errcode)                                        \
        {                                                              \
           if (THREADcurrent != NULL)                                  \
              (*(int *)THREADcurrent) = errcode;                       \
           errno = errcode;                                            \
        }

/* standard I/O */

#include <stdio.h>

THREAD_MONITOR THREADstdiomon[64];    /* This should be NOFILE */
THREAD_MONITOR THREADstdiobufmon;
#undef getc
#undef putc
#define _getc(p)         (--(p)->_cnt>=0? \
        (int)(*(unsigned char *)(p)->_ptr++) : _filbuf(p))
#define _putc(x, p)      (--(p)->_cnt >= 0 ?\
        (int)(*(unsigned char *)(p)->_ptr++ = (x)) :\
        (((p)->_flag & _IOLBF) && -(p)->_cnt < (p)->_bufsiz ?\
                ((*(p)->_ptr = (x)) != '\n' ?\
                        (int)(*(unsigned char *)(p)->_ptr++) :\
                        _flsbuf(*(unsigned char *)(p)->_ptr, p)) :\
                _flsbuf((unsigned char)(x), p)))



#define open(a,b,c)   THREADsysopen(a,b,c)
#define dup(a)        THREADsysdup(a)
#define dup2(a,b)     THREADsysdup2(a,b)
#define creat(a,b)    THREADsysopen(a,O_WRONLY|O_CREAT|O_TRUNC,b)
#define accept(a,b,c) THREADsysaccept(a,b,c)
#define socket(a,b,c) THREADsyssocket(a,b,c)
#define close(a)      THREADsysclose(a)
#define malloc(a)     _malloc(a)
#define free(b)       _free(b)
#define memalign(a,b) _memalign(a,b)


#endif _THREADS_
```

```
/************************************************************************/
/*        thread - thread package.                                      */
/*                                                                      */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University      */
/*        - All rights reserved.                                        */
/************************************************************************/


#ifndef thread_h
#define thread_h

#include "tcb.h"
#include "privatedata.h"



#define THREADactiveincr   THREADlock(&THREADactivecountlock);\
                           (THREADactivecount++);\
                           THREADunlock(&THREADactivecountlock);
#define THREADactivedecr   THREADlock(&THREADactivecountlock);\
                           (THREADactivecount--);\
                           THREADunlock(&THREADactivecountlock);

/************************************************************************/
/* external functions & variables                                       */
/************************************************************************/


extern int THREADactivecount;   /* the # of active threads              */
extern LOCK THREADactivecountlock;


extern int THREADdestroy (THREAD);
extern THREAD THREADclonetcb (THREAD, void (*)(), int *, int, int);



#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef thread_type1_h
#define thread_type1_h

#include "tcb.h"


/*****************************************************************************/
/* typdef type_swtch                                                         */
/*****************************************************************************/

typedef struct {
    void (*threadinit)();
    void (*threadterminate)();
    void (*systeminit)();
    void (*systemterminate)();
    int offset;
} type_swtch;

/*****************************************************************************/
/* external functions and variables                                          */
/*****************************************************************************/

extern int THREADtypecount;
extern int THREADtype1_extent;

#define THREADMAXTYPES          16
#define THREADpresetdynamics     2
#define THREAD_TYPE1_THREAD_TERMINATE(tcb)
#define THREAD_TYPE1_THREAD_INIT(tcb)

#endif
```

```
/*****************************************************************************/
/*        thread - thread package.                                           */
/*                                                                           */
/*        Copyright 1986 Thomas W. Doeppner Jr. - Brown University           */
/*        - All rights reserved.                                             */
/*****************************************************************************/

#ifndef tsignal_h
#define tsignal_h

#include <signal.h>
#include "tcb.h"

/*****************************************************************************/
/*****************************************************************************/

typedef struct sighandler {
    int type;
    THREAD (*whichthread)();
    void (*func)();
    int stacksize;
    int priority;
} SIGHANDLER;


struct sigparm {
    int sig;
    int code;
    void (*handler)();
};




#define SIG_UNASSIGNED          0
#define SIG_EXCEPTION           1
#define SIG_SENDSIG             2
#define SIG_CREATETHREAD        3
#define SIG_ABORT               4



/*****************************************************************************/
/* external functions and variables                                          */
/*****************************************************************************/

void THREADsiginit ();
void THREADfakesignal (THREAD, void (*)(), int, int, int);



#endif
```

```
        !.file  "lock.s"
        .seg    "text"
        .proc   04
        .global _swaplock

        ! protocol: lock (int *)
        !
        ! when this int* is ZERO, means it's unlocked.
        ! using ATOMIC INSTRUCTION to swap int* with a local var which was ONE
        .align  4
_swaplock:
        !#PROLOGUE# 0
        sethi   %hi(LF12),%g1
        add     %g1,%lo(LF12),%g1
        save    %sp,%g1,%sp
        !#PROLOGUE# 1
        st      %i0,[%fp+0x44]
L14:
        .seg    "text"
        mov     0x1,%o0
        st      %o0,[%fp+-0x4]  ! initialize the local variable to be ONE
L15:
        ld      [%fp+-0x4],%o0
        ld      [%fp+0x44],%o1  ! this int*
        swap    [%o1],%o0       ! swap
        nop

        cmp     %o0,0x1         ! test the local variable
        bne     L17             ! not ONE then we get the lock
        nop

L16:
        ld      [%fp+0x44],%o1
        ld      [%o1],%o0

        cmp     %o0,0x1         ! wait for someone unlock the lock int*
        be      L16
        nop

L17:
LE12:
        ret
        restore
        .optim  "-O~Q~R~S"
       LF12 = -72
        LP12 = 64
        LST12 = 64
        LT12 = 64

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        .seg    "text"
        .proc   04
        .global _swapunlock

        .align  4
_swapunlock:
        !#PROLOGUE# 0
        sethi   %hi(LF26),%g1
        add     %g1,%lo(LF26),%g1
        save    %sp,%g1,%sp
        !#PROLOGUE# 1
        st      %i0,[%fp+0x44]
L28:
        ld      [%fp+0x44],%o0
        st      %g0,[%o0]
LE26:
        ret
        restore
        .optim  "-O~Q~R~S"
       LF26 = -64
        LP26 = 64
        LST26 = 64
        LT26 = 64

        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        .seg    "text"
        .proc   04
        .global _swapcondlock

        .align  4
_swapcondlock:
        !#PROLOGUE# 0
        sethi   %hi(LF30),%g1
        add     %g1,%lo(LF30),%g1
        save    %sp,%g1,%sp
        !#PROLOGUE# 1
        st      %i0,[%fp+0x44]
L32:
        mov     0x1,%o0
        st      %o0,[%fp+-0x4]  ! initialize the local variable to be ONE

        ld      [%fp+-0x4],%o0
        ld      [%fp+0x44],%o1
        swap    [%o1],%o0
        nop

        cmp     %o0,0x1
        be      L34
        nop

        mov     1,%o0   ! we got the lock, prepare to return 1
        b       LE30
        nop

L34:
        mov     0,%o0   ! we can't get the lock, prepare to return 0

LE30:
        mov     %o0,%i0
        ret
        restore
        .optim  "-O~Q~R~S"
       LF30 = -72
        LP30 = 64
        LST30 = 64
        LT30 = 64


        !ident  "acomp: (CDS) SunOS5.0 IDR3.5 (H6.1) 11/01/90"
```