Implementing Threads
A Unix process’s address space appears to be three regions of memory: a read-only text region (containing executable code); a read-write region consisting of initialized data (simply called data), uninitialized data (BSS — a directive from an ancient assembler (for the IBM 704 series of computers), standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a dynamic area; and a second read-write region containing the process’s user stack (a standard Unix process contains only one thread of control).

The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to sbrk system calls. Most programmers do not use this system call directly, but instead use the malloc and free library routines, which manage the dynamic area and allocate memory when needed by in turn executing sbrk system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)
Adding More Stuff

stack 1
stack 2
stack 3
mapped file 1
mapped file 2
mapped file 3
... mapped file 117
bss & dynamic
data
text
Unix was not designed with multithreaded programming in mind. A good example of the implications of this is the manner in which error codes for failed system calls are made available to a program: if a system call fails, it returns -1 and the error code is stored in the global variable `errno`. Though this is not all that bad for single-threaded programs, it is plain wrong for multithreaded programs.
The ideal way to solve the “errno problem” would be to redesign the C/system-call interface: system calls should return only an error code. Anything else to be returned should be returned via result parameters. (This is how things are done in the Win32 interface.) Unfortunately, this is not possible (it would break pretty much every Unix program in existence).

So we are stuck with errno. What can we do to make errno coexist with multithreaded programming? What would help would be to arrange, somehow, that each thread has its own private copy of errno. I.e., whenever a thread refers to errno, it refers to a different location from any other thread when it refers to errno.

What can we do? As shown in the slide, we might use the C preprocessor to redefine errno as something a bit more complicated — references to errno result in accessing an array of errnos, one for each thread. This might turn out to be impractical, but we can devise other approaches, as shown in the next slide.

Note: in Linux, when threads are used, errno is defined to be “(*__errno_location( ))” i.e., it dereferences the result of calling a function that retrieves the thread-specific address of errno.
There are enough other situations analogous to \texttt{errno} in which each thread should have its own private copy of an otherwise global variable that a general mechanism to deal with it is important. A data item that is global but private to a thread is called \textit{thread-specific data}. Rather than represent each such item as a separate array, as mentioned in the previous slide, what's done in POSIX threads is, effectively, to give each thread an array of pointer-size values (it doesn't have to be done this way, but the effect must be the same). Each element of the array either holds or points to a thread-specific-data (TSD) item. Thus TSD[0] might contain the address of thread-specific-data item \texttt{var1}, TSD[1] might contain the value of \texttt{errno}, and TSD[2] might contain the address of \texttt{var2}. What's important is that whenever a thread refers to the TSD array, it uses its own private array. This may sound like we're begging the question, but, as seen in the next slide, we introduce special machinery for getting at this array.
So that we can be certain that it’s the calling thread’s array that is accessed, rather than access the TSD array directly, one uses a set of POSIX threads library routines. To find an unused slot, one calls `pthread_key_create`, which returns the index of an available slot in its first argument. Its second argument is the address of a routine that’s automatically called when the thread terminates, so as to do any cleanup that might be necessary (it’s called with the key (index) as its sole argument, and is called only if the thread has actually stored a non-null value into the slot). To put a value in a slot, i.e., perform the equivalent of TSD[i] = x, one calls `pthread_setspecific(i, x)`. To fetch from the slot, one calls `pthread_getspecific(i)`. 

---

**The TSD Array**

- `pthread_key_create(&key, cleanuproutine)`
  - allocates a slot in the TSD arrays
  - provides a function to cleanup when threads terminate
- `value = pthread_getspecific(key)`
  - fetches from the calling thread’s array
- `pthread_setspecific(key, value)`
  - stores into the calling thread’s array
Here we see how we might use thread-specific data to implement `errno` in a thread-safe way. First we allocate a slot in the TSD array to hold each thread’s copy of `errno`. It’s important that this be done only once, but that the slot’s index be placed in a global variable (`errno_key` in this case) so that all threads can access it. (Since no cleanup is necessary when threads terminate, we don’t supply a cleanup routine.) Next we see a sketch of the write library routine, which is actually a wrapper for an instruction that traps into the kernel to make the actual call. On return from the trap, if there was an error, we put its code into the TSD slot that was allocated for `errno`.

```c
pthread_key_t errno_key; // global
pthread_key_create(&errno_key, 0);
    // done by only one thread

int write(...) { // wrapper for syscall
    int err = syscall(WRITE, ...);
    if (err)
        pthread_setspecific(errno_key, err);
    ...
}
```
We repeat our earlier example that used `errno`, however this time we've redefined references to `errno` to use `pthread_getspecific`.

```c
#define errno pthread_getspecific(errno_key)
   // make things easy to read and type

if (write(fd, buffer, size) == -1) {
   if (errno == EIO)
      fprintf(stderr, "IO problems \n");
... return(0);
}
```
ELF stands for “executable and linking format” and is the standard format for executable and object files used on most Unix (and other) systems. The __thread attribute tells gcc that the item being declared is to be thread-local, which is the same thing as thread-specific. A detailed description of how it is implemented can be found at http://people.redhat.com/drepper/tls.pdf.
Subroutines

```c
int main( ) {
    int i;
    int a;
    ...
    i = sub(a, 1);
    ...
    return(0);
}

int sub(int x, int y) {
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```

Subroutines are (or should be) a well understood programming concept: one procedure calls another, passing it arguments and possibly expecting a return value. We examine how the linkage between caller and callee is implemented on the Intel x86 architecture.
Subroutine linkage on an Intel x86 is fairly straightforward. (We are discussing the 32-bit version of the architecture.) Associated with each incarnation of a subroutine is a stack frame that contains the arguments to the subroutine, the instruction pointer (in register eip) of the caller (i.e. the address to which control should return when the subroutine completes), a copy of the caller’s frame pointer (in register ebp), which links the stack frame to the previous frame, space to save any registers modified by the subroutine, and space for local variables used by the subroutine. Note that these frames are of variable size—the size of the space reserved for local data depends on the subroutine, as does the size of the space reserved for registers.

The frame pointer register (ebp) points into the stack frame at a fixed position, just after the saved copy of the caller’s instruction pointer (note that lower-addressed memory is towards the bottom of the picture). The value of the frame pointer is not changed by the subroutine, other than setting it on entry to the subroutine and restoring it on exit. The stack pointer (esp) always points to the last item on the stack—new allocations (e.g. for arguments to be passed to the next procedure) are performed here.

This picture is idealized: not all portions of the stack frame are always used. For example, registers are not saved if the subroutine doesn’t modify them. The frame pointer is not saved if it’s not used, etc.

Intel x86:
Subroutine Code (1)

main:
pushl %ebp
movl %esp, %ebp
pushl %esi
pushl %edi
subl $8, %esp
...
pushl $1
movl -12(%ebp), %eax
pushl %eax
call sub
addl $8, %esp
movl %eax, -16(%ebp)
...

movl $0, %eax
popl %edi
popl %esi
movl %ebp, %esp
popl %ebp
ret

set up stack frame
push args
pop args; get result
set return value and restore frame
Intel x86: Subroutine Code (2)

sub:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $1, -4(%ebp)
  movl $0, -8(%ebp)
  movl -4(%ebp), %ecx
  movl -8(%ebp), %eax

beginloop:
  cmp 12(%ebp), %eax
  jge endloop
  imull 8(%ebp), %ecx
  addl $1, %eax
  jmp beginloop

init locals
get args

endloop:
  movl %ecx, -4(%ebp)
  movl -4(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
x86-64

- Twice as many registers
- Arguments may be passed in registers, rather than on stack
- No special-purpose frame pointer
  - use stack pointer instead
main:
subq $24, %rsp  # reserve space on stack for locals
...
movl 12(%rsp), %edi  # set first argument
movl $1, %esi  # set second argument
call sub
addl $24, %rsp
...
movl $0, %eax  # set return value
ret
...
sub:
testl %esi, %esi    # leaf function: no stack setup
jle  skiploop
movl $1, %eax
movl $0, %edx
loop:
imull %edi, %eax
addl $1, %edx
cmpl %esi, %edx
jne  loop
ret
skiploop:
movl $1, %eax
ret
The SPARC (Scalable Processor ARChitecture) is an example of a RISC (Reduced-Instruction-Set Computer). We won’t go into all of the details of its architecture, but we do cover what is relevant from the point of view of subroutine calling conventions. There are nominally 32 registers on the SPARC, arranged as four groups of eight—input registers, local registers, output registers, and global registers. Two of the input registers serve the special purposes of a return address register and a frame pointer, much like the corresponding registers on the 68000. One of the output registers is the stack pointer. Register 0 (of the global registers) is very special—when read it always reads 0 and when written it acts as a sink.

SPARC architecture manuals can be found at http://www.sparc.com/specificationsDocuments.html.
As its subroutine-calling technique the SPARC uses *sliding windows*: when one calls a subroutine, the caller's output registers become the callee's input registers. Thus the register sets of successive subroutines overlap, as shown in the picture.

Any particular implementation of the SPARC has a fixed number of register sets (of eight registers a piece)—seven in the picture. As long as we do not exceed the number of register sets, subroutine entry and exit is very efficient—the input and local registers are effectively saved (and made unavailable to the callee) on subroutine entry, and arguments (up to six) can be efficiently passed to the callee. The caller just puts outgoing arguments in the output registers and the callee finds them in its input registers. Returning from a subroutine involves first putting the return value in a designated input register (i0). In a single action, control transfers to the location contained in i7, the return address register, and the register windows are shifted so that the caller’s registers are in place again.

However, if the nesting of subroutine calls exceeds the available number of register sets, then subroutine entry and exit is not so efficient—the register windows must be copied to an x86-like stack. As implemented on the SPARC, when an attempt is made to nest subroutines deeper than can be handled by the register windows, a trap occurs and the operating system is called upon to copy the registers to the program's stack and reset the windows. Similarly, when a subroutine return encounters the end of the register windows, a trap again occurs and the operating system loads a new set of registers from the values stored on the program's stack.
The form of the SPARC stack is shown in the picture. Space is always allocated for the stack on entry to a subroutine. The space for saving the \emph{in} and \emph{local} registers is not used unless necessary because of a window overflow. The “hidden” parameter supports programs that return something larger than 32 bits—this field within the stack points to the parameter (which is located in separately allocated storage off the stack).
Here we see the assembler code produced by a compiler for the SPARC. The first step, in preparation for a subroutine call, is to put the outgoing parameters into the output registers. The first parameter, a from our original C program, is a local variable and is found in the stack frame. The second parameter is a constant. The call instruction merely saves the program counter in $o7$ and then transfers control to the indicated address. In the subroutine, the save instruction creates a new stack frame and advances the register windows. It creates the new stack frame by taking the old value of the stack pointer (in the caller’s $o6$), subtracting from it the amount of space that is needed (64 bytes in this example), and storing the result into the callee’s stack pointer ($o6$ of the callee). At the same time, it also advances the register windows, so that the caller’s output registers become the callee’s input registers. If there is a window overflow, then the operating system takes over.

Inside the subroutine, the return value is computed and stored into the callee’s $i0$. The restore instruction pops the stack and backs down the register windows. Thus what the callee left in $i0$ is found by the caller in $o0$. 

---

**SPARC Architecture: Subroutine Code**

```
ld [%fp-8], %o0
   ! put local var (a)
   ! into out register
mov 1, %o1
   ! deal with 2nd
   ! parameter
call sub
nop
st %o0, [%fp-4]
   ! store result into
   ! local var (i)
sub:
save %sp, -64, %sp
   ! push a new
   ! stack frame
add %i0, %i1, %i0
   ! compute sum
ret
   ! return to caller
restore
   ! pop frame off
   ! stack (in delay slot)
```
We now consider what happens with multiple threads of control. Each thread must have its own context, represented by a control block and a stack. Together these represent what needs to be known about a thread within a particular address space. What must be stored in the control block?

For the 32-bit x86, the stack pointer (%esp) and the frame pointer (%ebp) registers refer to the thread’s current stack context, and this context contains all the register values that are important to the thread’s execution. Thus it’s sufficient for the thread control block to contain the stack and frame pointers.

For the SPARC, the stack pointer refers to the thread’s stack context. The thread’s registers are contained in the combination of the stack itself and the current set of register windows in the processor. Thus if these windows are stored on the stack, it’s sufficient for the thread control block to contain the stack pointer.

For the 64-bit x86-64, we need not only the stack pointer, which refers to the thread’s stack context, but also all the registers that are used to pass parameters.
Switching Between Threads

- Coroutine linkage
Switching Between Threads

```c
1 void thread_switch(thread_t *next_thread) {
2     SaveContext(&CurrentThread->ctx);
3     CurrentThread = next_thread;
4     GetContext(&CurrentThread->ctx);
5     return;
6 }
```

This code is suggestive of how we might switch from one thread to another. The thread being switched out of calls `thread_switch`, passing it the address of the target thread’s control block. We save whatever registers are necessary to represent the state of the current thread in its control block by calling `SaveContext`, and we restore the registers of the thread being switched to by calling `GetContext`. `CurrentThread` is a global variable that we make sure always points to the control block of the currently running thread.

Unfortunately, this code isn’t correct. Consider what happens if thread 1 switches to thread 2, and then thread 2 switches back to thread 1. Thread 1’s state, including its instruction pointer (%eip on the x86) is saved a line 2. When it is restored by thread 2, the value of %eip will be restored, but it’s as it was when it was saved. Thus thread 1 resumes at line 2, and rather than return, it immediately switches (again) to thread 2).
In this version, we introduce SwapContext, which saves the current register context into the first argument, and restores the register context from the second argument.

Unfortunately, there’s still a bug: our intent is that CurrentThread, in line 4, be set to refer to the thread that was just switched to. But, at this point, we’re executing on the stack of the new thread, and next_thread is whatever next_thread referred to when this thread called thread_switch.
Switching Between Threads, Take 3

```c
1 void thread_switch(thread_t *next_thread) {
2    thread_t *oldCurrentThread = CurrentThread;
3    CurrentThread = next_thread;
4    SwapContext(&oldCurrentThread->ctx,
5        &CurrentThread->ctx);
6    return;
7 }
```

This version finally works. We change the value of CurrentThread before switching to next_thread’s stack.
A Simple Threads Implementation

- Basis for user-level threads package
- Straight-threads implementation
  - no interrupts
  - everything in thread contexts
  - one processor
Basic Representation

Thread object

Stack
Thread Switch (in C)

```c
void thread_switch( ) {
    thread_t NextThread, OldCurrent;

    NextThread = dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(&OldCurrent->context, &NextThread->context);

    // We're now in the new thread's context
}
```
Thread-Switch Exchange

```c
void thread_switch() {
    thread_t NextThread,
    OldCurrent;
    NextThread =
    dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext( &OldCurrent->context,
                 &NextThread->context);
}
```

... thread_switch();
...

Stack

SP

IP
Thread-Switch Exchange

```
void thread_switch() {
    thread_t NextThread,
    OldCurrent;

    NextThread =
        dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(
        &OldCurrent->context,
        &NextThread->context);

    ... thread_switch();...
```

SP

IP
Thread-Switch Exchange

```
void thread_switch( ) {
    thread_t NextThread, OldCurrent;
    NextThread = dequeue(RunQueue);      
    OldCurrent = CurrentThread;         
    CurrentThread = NextThread;         
    swapcontext(&OldCurrent->context,   
                &NextThread->context);   
}
```

Operating Systems In Depth  II-33 Copyright © 2016 Thomas W. Doeppner. All rights reserved.
Thread-Switch Exchange

```
void thread_switch() {
    thread_t NextThread,
    OldCurrent;

    NextThread =
        dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(
        &OldCurrent->context,
        &NextThread->context);
}
```

... thread_switch();
...

... thread_switch();
...
void thread_switch() {
    thread_t NextThread,
    OldCurrent;
    NextThread =
    dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext( &OldCurrent->context,
                &NextThread->context);
}

... thread_switch();
...
Thread-Switch Exchange

```c
void thread_switch() {
    thread_t NextThread,
    OldCurrent;

    NextThread =
        dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(
        &OldCurrent->context,
        &NextThread->context);
}
```

Operating Systems In Depth  Copyright © 2016 Thomas W. Davenport. All rights reserved.
Mutexes

```c
mutex_t mut;

mutex_lock(&mut);
x = x+1;
mutex_unlock(&mut);
```
Implementing Mutexes

```c
void mutex_lock(mutex_t *m) {
    if (m->locked) {
        enqueue(m->wait_queue, CurrentThread);
        thread_switch();
    }
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
    m->locked = 0;
    if (!queue_empty(m->wait_queue))
        enqueue(RunQueue, dequeue(m->wait_queue));
}
```

Note: this code doesn’t work!
Implementing Mutexes, Take 2

```c
void mutex_lock(mutex_t *m) {
    if (m->locked) {
        enqueue(m->queue, CurrentThread);
        thread_switch();
    } else
        m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
    if (queue_empty(m->queue))
        m->locked = 0;
    else
        enqueue(runqueue, dequeue(m->queue));
}
```