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
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

init locals

get args

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

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
...

Intel x86-64:
Subroutine Code (1)
Intel x86-64: Subroutine Code (2)

```
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 in and 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.
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?

We clearly need the stack pointer, otherwise when we switch between threads we won’t be able to switch stacks. If the frame pointer is also essential (as in x86), we’ll need that as well. We’ll also need the instruction pointer – we need to know where the thread is to be executing. In the next few slides we’ll look at how we switch between threads and then we’ll get back to what else must be in the thread control block.
The slide shows the flow of control of two threads running on a single processor. They explicitly yield the process to the other by calling thread_switch. Assume the left thread runs first. It calls thread_switch, yielding control to the right thread. At some point later, the right thread calls thread_switch, yielding control back to the left thread, which resumes after its original call to thread_switch. Thus from the point of view of the left thread, calling thread_switch was much like a procedure (or function) call to the right thread: it called thread_switch, passing control to the right thread, at some point later control returned to the statement following the call. But, sometime later, the left thread again calls thread_switch. Control returns to the right thread, to the location following its original call to thread_switch. From the right thread’s point of view, calling thread_switch behaves like a procedure call to left thread. Thus each thread behaves as if it is calling the other – this sort of back and forth between threads, where each explicitly yields control to the other, is known as coroutine linkage: rather than a procedure calling a subroutine, we have co-equal coroutines calling each other.
Switching Between Threads

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 at 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.
Quiz 1

Does this implementation of thread_switch work?

a) yes: in all cases
b) yes, except for a few edge cases
c) no
Unfortunately, there’s still a bug in the previous implementation: our intent was that `CurrentThread`, in line 4 of that implementation, 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`.

Here we have a version that (finally) works. We change the value of `CurrentThread` before switching to `next_thread`’s stack.
We now begin to describe a real implementation of threads, which is the basis for the uthreads programming assignment.
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(); ...
```
Thread-Switch Exchange

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

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

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

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

... thread_switch(); ...

... thread_switch(); ...

Operating Systems In Depth
void thread_switch() {
    thread_t NextThread,
    OldCurrent;
    NextThread =
    dequeue(SrunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(
        &OldCurrent->context,
        &NextThread->context);
}

... thread_switch(); ...

Return address
NextThread
OldCurrent

SP

Return address
NextThread
OldCurrent

IP
void thread_switch() {
    thread_t NextThread, OldCurrent;

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

... thread_switch(); ...

... thread_switch(); ...

Stack
Return address
NextThread
OldCurrent

Stack
Return address
NextThread
OldCurrent
void thread_switch() {
    thread_t NextThread,
    OldCurrent;

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

... thread_switch();
...
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!
Quiz 2

a) It works.
b) It works as long as there are just two threads.
c) It doesn’t work. Period.
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));
}
```