Computer Science 432/563
Operating Systems
The College of Saint Rose
Spring 2016

Topic Notes: Processes and Threads

What is a *process*? Our text defines it as "a program in execution" (a good definition).

Definitions from other textbooks:

- an abstraction of a running program
- An asynchronous activity
- the "locus of control" of a program in execution
- that which is manifest by the existence of a process control block in the OS
- that entity which is assigned to processors
- the "dispatchable" unit
- the "animated spirit" of a procedure

A process is sequential.

Parts of a process:

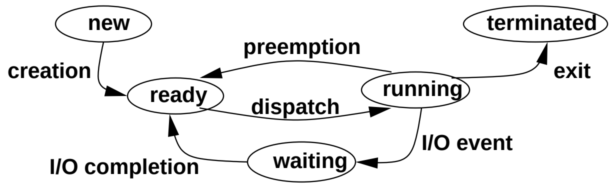
- program code (text section)
- program counter and other registers
- stack (local variables and function call information)
- data (global variables)
- heap (dynamically allocated variables)

A typical multiprogrammed system has many processes at any time. Try `ps -aux` or `ps -ef` to see the processes on your favorite Unix system. Only one of these processes can be on a CPU at a time.

If we look at the `ps` output on a Unix system, we will see a lot of processes owned by `root`, the "super-user". Many of these are essential parts of the system and are intended to continue running as long as the system is up. These processes are called *daemons* and are the motivation for the BSD logo.

So if a process is "in the system" but not executing on the CPU, where is it?

States of a process:



What information do we need to let a process transition among these states?

Think about it in terms of what a person needs to do to get back to what he or she was doing before being interrupted.

- you're sitting in the lab working hard on your OS project and someone interrupts you
- your attention shifts and you go off and do something else for a while
- then when you need to come back to the work, you need to remember *what* you were doing, and *where you were* in the process of doing it

We do this all the time, and many of us are really pretty good at it. A processor can't just pick up where it left off, unless we carefully remember everything it was doing when it was so rudely interrupted.

A *process control block* (PCB) is used to store the information needed to save and restore a process. This information includes:

- Process state (running, waiting, ready)
- Process identifier (PID)
- Program counter
- Other CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

In many Unix systems, the PCB is divided between two structures:

- The *proc* structure, which holds the process information needed at all times, even when the process is not active (swapped out).
- The *user* structure, which holds other information about the process.

Historically, it was important to keep as much in the user structure and as little in the proc structure as possible, because of memory constraints. As the size of memory has grown, the amount of memory used by PCBs has become less significant, so this division has become less important.

In FreeBSD, for example, most of the information is now in the proc structure. We can see it on your favorite FreeBSD 10.1 system (ascg) in `/usr/src/sys/sys/proc.h`. See `struct proc` defined starting at line 485.

The user structure is in `/usr/src/sys/sys/user.h` (line 235), and it is somewhat smaller. It contains per-thread information. Part of this is a `struct kinfo_proc`, which in turn includes a `struct pcb`, which is an architecture-dependent structure defined in `/usr/src/sys/amd64/include/pcb.h` (line 47). Here you find the actual registers that need to be saved when a process is removed from the CPU. For comparison, check out the MIPS version in `/usr/src/sys/mips/include` (it's tiny!).

The transition from one process running on the CPU to another is called a *context switch*. This is pure overhead, so it needs to be fast. Some systems do it faster than others. Hardware support helps - more on that later.

This requires at least two levels of privilege, or modes of operation:

- User mode: execution of user processes. Processes in this mode have restrictions on what they can do. The process can only access its own memory, cannot perform privileged instructions, etc.
- Monitor mode (kernel mode, system mode): This is operating system execution. In this mode, the process can do things like switch among user processes, shut down the system, manage memory at a lower level, etc.

Hardware support is needed. We need privileged instructions can be executed only while in monitor mode. We'll talk more about this as we go forward.

Process Creation/Deletion

In Unix, every process (except for the first) is created from an existing process. See `ps` again for examples. The processes form a tree, with the root being the `init` process (PID=1).

When a new process is created, what information does it inherit from or share with its parent?

- Does it get any resources that were allocated to the parent?
- Does the parent wait for the child to complete, or do they execute concurrently?
- Is the child a duplicate of the parent, or is it something completely different?
- If it's a duplicate, how much context do they share?
- Can the parent terminate before the child?

Creation of a new process is a highly privileged operation. It requires the allocation of a new PCB, insertion of this PCB into system data structures, among other operations that we would not trust to regular "user" programs.

The typical solution to this is that such operations are provided through *system calls*. These system calls are functions that are part of the operating system and are permitted to perform some tasks that a normal process is not able to do on its own. In effect, the program temporarily gains a higher privilege level while executing the system call (if the kernel grants the permission for the process to execute the call).

In Unix, the `fork()` system call duplicates a process. The child is a copy of the parent - in execution at the same point, the statement after the return from `fork()`.

The return value indicates if you are the new process (the child) or the original process (the parent).

0 is child, > 0 means parent, -1 means failure (e.g., a process limit has been reached, permission is denied)

A C program that wishes to create a new process will include code similar to this pattern:

```
pid=fork();
if (pid) {
  parent stuff;
}
else {
  child stuff;
}
```

A more complete program that uses `fork()` along with three other system calls (`wait()`, `getpid()`, and `getppid()`) is in `forking.c`:

See Example:
`/home/cs432/examples/forking`

Some comments about this program:

First, run it to observe what happens. Note that there is only one copy of the printout before the `fork()`, two of the one after. `fork()` is a very unusual function - you call it once, but it returns twice!

How do we know how these work? See the man pages! The `fork` page tells us what we need to include, and how to use it.

How about `wait`? If we issue the command `man wait`, we get the man page for `builtin`, as there is a built-in `wait` command in the shell. To get the page we want, we need to specify a manual "section." Section 2 is the system calls section. The command `man 2 wait` will get us the page we want.

Are there really two processes? Let's look at the output of `ps` as the program runs.

Again, these system calls let you, as a normal user, do things that only the system should be able to do. Your "user mode" process can get access to "kernel mode" functionality through these calls. But since you are accessing this functionality through the system call interface, there is control over your ability to do so.

How many processes can we create on various systems? Where does this limit come from? Can we create enough processes to take down a system?

See Example:
`/home/cs432/examples/forkbomb`

The FreeBSD implementation of `fork()` is in `/usr/src/sys/kern/kern_fork.c`.

Things to note here (for an example – don't worry about the details):

- the action is happening in `fork1()`, which starts on line 751.
- Line 824: actually allocate the new proc structure with `uma_zalloc`, a special `malloc` basically (see `zalloc(9)`).
- Line 882: initialize proc table entry
- Line 896: check if a new process is allowed based on global system limits.

  Note that `nprocs` is the kernel variable that stores the number of active processes in the system, `maxproc` is the upper limit on the number of processes allowed.

  Note also how a regular user is restricted from creating a process if the system is within 10 of the overall limit.
- Line 614 (back in `do_fork`): attach new process to parent – or to init process if a certain flag is set to say that the parent should not wait for its child.
- Line 723: set on run queue, so the child can start executing.

We will see that in many cases, you will want to use the `vfork(2)` system call instead of `fork()`. This one doesn't copy the address space of the process – it assumes you are going to replace the newly created process' program with a new one.

In the Windows world, there is a `CreateProcess()` Windows API call (the Windows API is like a POSIX for Windows) that creates a new process and loads the correct program into that new process.

Soon we will consider more system calls, including ones that let you do things more interesting than making a copy of yourself.

There is also the issue of how all this gets started – how does that first process get started that `fork`s everything else?

Processes may need to communicate with each other in some more interesting way than making copies of themselves. We will see a number of ways this can be done, including the use of a small chunk of POSIX shared memory.

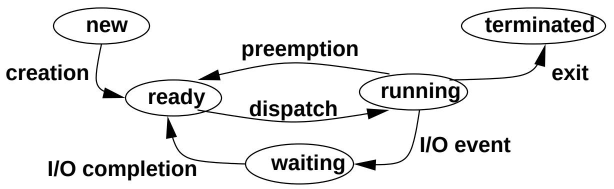
The textbook has examples of how to create a shared memory segment that can be accessed by your parent and child process.

Another possibility is to have processes pass messages to each other over the network or through the file system.

We will spend a good chunk of time later this semester on cooperating processes.

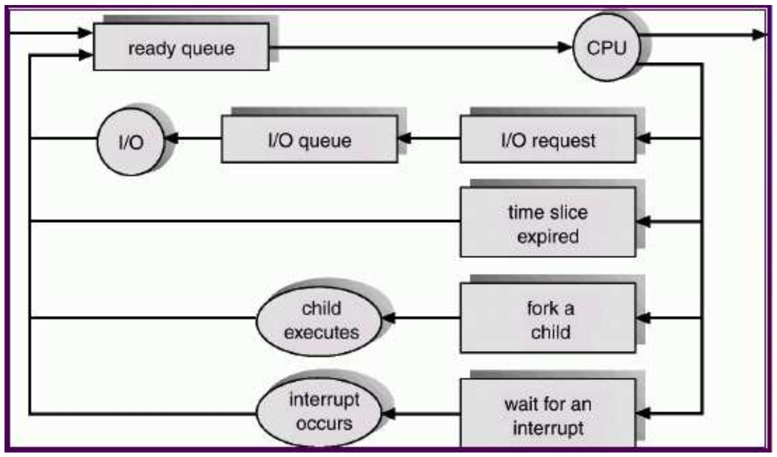
Process Scheduling Queues

Since the kernel is going to allow us to create all of these processes, it will need to manage them. Recall the process states.



Basically, each process in the system has to be in one of several queues. These are *process scheduling queues*

The system maintains a *process table* and the PCBs are stored in these process scheduling queues.



The selection of a process from the ready queue to run on the CPU is the topic for our CPU scheduling unit.

A fundamental job of the operating system is to manage these queues.

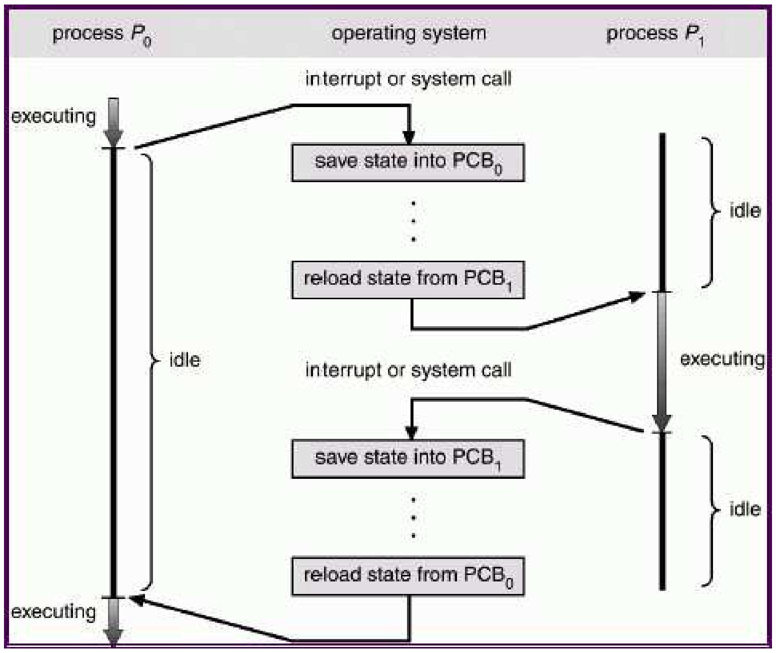
FreeBSD manages its run/ready queues in `/usr/src/sys/kern/kern_switch.c`

There are actually 64 regular queues (this number is defined in `/usr/src/sys/sys/runq.h`). We'll think more about what these are about when we talk about CPU scheduling.

As processes go through the system, processes will be assigned to the CPU (dispatched) or removed from the CPU dozens or even hundreds of times per second. Each of these swaps is a *context switch*, saving a CPU state in a PCB, then restoring the CPU state to the contents of another PCB.

Remember, the context switch is pure overhead. The system is not doing any useful computation when it's working on a context switch. It had better be fast.

Diagram of context switch:



FreeBSD does this in what is necessarily an architecture-dependent routine.

In fact, it's an assembly source file, not C!

- For the amd64 architecture: `/usr/src/sys/amd64/amd65/cpu_switch.S`
- For the i386 architecture: `/usr/src/sys/i386/i386/swtch.s`
- For the mips architecture: `/usr/src/sys/mips/mips/swtch.S`
- For the sparc architecture: `/usr/src/sys/sparc64/sparc64/swtch.S`.

Notes about the context switch (looking at the amd64 one):

- Line 93: an assembly code entry point (essentially function) `cpu_switch`.
- Line 99: save hardware registers into PCB.
- Line 248: restore PCB of new proc

- This function ends up returning to the new process, picking up where it left off the last time it was switched out! It never knows it was asleep.

Registers are just one part of a process' context. What about memory? What about cache lines?

We'll talk about main memory later. Each process in the ready/run queues has some main memory allocated to it.

But cache is another story. Remember how a typical cache is set up. It is the closest to the CPU and registers in the memory hierarchy.

As the CPU requests memory, lines of values from memory are brought into the cache. If all goes well, these lines will be reused.

But when a context switch occurs, the things from the process that was on the CPU that are now in cache seem unlikely to survive there until that process gets another chance on the CPU. We may force a lot of cache misses, adding more overhead to the context switch cost.

Interprocess Communication

Many processes are *independent*, they are not affected by and do not affect other processes in a system.

Cooperating process can affect or be affected by other processes, allowing them to share data or to coordinate in some controlled and useful way to accomplish some task.

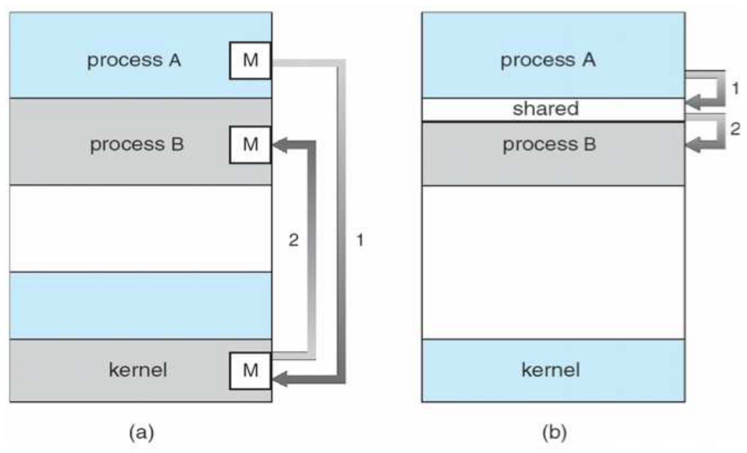
There are several motivations for cooperating processes:

- Information sharing
- Computational speedup (make use of multiple CPUs)
- Modularity or Convenience

It's hard to find a computer system where processes do not cooperate. Consider the commands you type at the Unix command line. Your shell process and the process that executes your command must cooperate. If you use a pipe to hook up two commands, you have even more process cooperation.

Cooperating processes must have some communication mechanism.

There are two primary approaches to *interprocess communication (IPC)*: *message passing* ((a) in the figure below) and *shared memory* ((b) in the figure below).



We will look in detail at many examples of cooperating process soon. For now, we will just briefly consider the ideas of message passing and shared memory.

Message Passing

An IPC message passing facility needs to provide two primitive operations: a *send* and a *receive*.

- `send(message)`
- `receive(message)`

For two processes to communicate, a *communication link* must be established between them, and they can then exchange messages using sends and receives across that link.

There are many implementation-specific questions to consider:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

With a *direct communication*, processes are named explicitly in sends and receives:

- `send (P, message)` send a message to process P

- `receive(Q, message)` receive a message from process Q

Typical properties in this type of communication:

- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

An *indirect communication* scheme utilizes the idea of *mailboxes* or *ports*.

- Each mailbox has a unique identifier (e.g., port number)
- Processes can communicate only if they share a mailbox

Typical properties in this type of communication:

- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional

The operations here are to create and destroy mailboxes, and to send and receive messages through these mailboxes.

The send/receive primitive operations here take the form:

- `send(A, message)` send a message to mailbox A
- `receive(A, message)` receive a message from mailbox A

Messages may be *blocking* or *non-blocking*.

Blocking messages are called *synchronous*:

- The sender blocks (i.e., waits) until the message is received
- The receiver blocks until a message is available

Non-blocking messages are called *asynchronous*:

- The sender sends the message and continues immediately
- The receiver receives a valid message if one has arrived, or null if not

The amount of *buffering* or the length of the queue of messages that a link can store determine some of its functionality:

- Zero capacity buffer: the sender must wait for receiver (known as *rendezvous*)
- Bounded capacity buffer (finite number of messages): the sender must wait if link buffer is full
- Unbounded capacity buffer: sender never needs to wait

IPC mechanisms

There are many real IPC implementations, some of which we will look at in detail in class and/or use in lab exercises:

- POSIX shared memory
- *Mach kernel* is entirely message-based
- The Windows API's *advanced local procedure call (ALPC)* allows a process to communicate with another process on the same system
- *Sockets* allow communication between processes even on different systems over IP
- *Remote procedure call (RPC)* allows procedure calls to be made by a process on one system to be executed on another
- *Pipes* allow communication between two local processes

Threads/Lightweight Processes

Some of you may be familiar with Java threads, especially if you did some Objectdraw programming with me, in which case we called them `ActiveObjects`. These gave you the ability to have your programs doing more than one thing at a time. Such a program is called *multithreaded*.

Threads are a stripped-down form of a process:

- A *task* or *process* can contain multiple *threads*
- Threads share process context

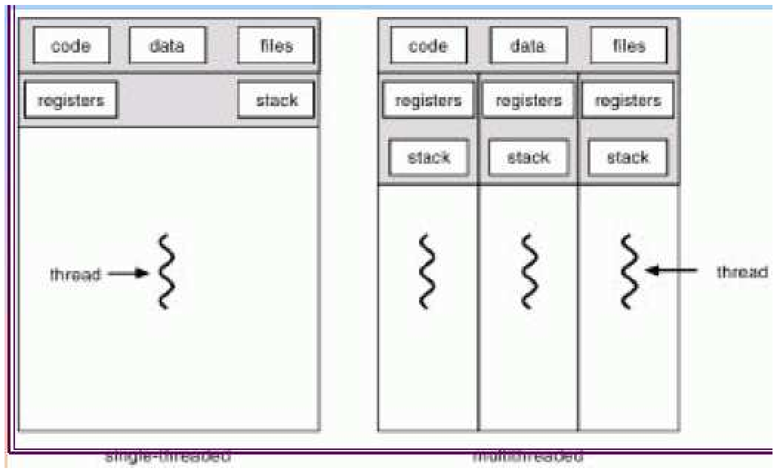
- Threads are asynchronous
- Threads have less context than processes
- Threads can be created/destroyed at a lower cost than processes
- Threads cooperate to do a process in parallel with (relatively) fine granularity of parallelism
- Threads are well-suited to symmetric multiprocessors (SMPs) and multi-core systems

Threads can be useful even in single-user multitasking systems:

- Foreground and background work: one thread to deal with user input, another to update display, etc such as in a spreadsheet or word processor.
- Asynchronous processing: auto-save features, background computation.
- Speed execution: compute with one batch of data while reading another. **Your** process can be in I/O and on the CPU at the same time!
- Organizing programs: cleaner and more elegant program design.

A "normal" single-thread process has its program text, global address space, a stack, and a PC.

A multithreaded process has its program text, a single shared global address space, but a stack and a PC for each thread.



Each thread also needs to be able to access the registers when it's on a CPU, so while a context switch between threads will not need to save and restore as much as a context switch between processes, but it will still need to save registers.

Multicore Programming

Multithreaded programs are well suited to take advantage of the multiple processing cores found in most modern computers. But writing a program to do so correctly and efficiently is challenging.

The following are some challenges that must be overcome:

- Dividing activities – the program needs to be broken down into separate tasks that can be run concurrently.
- Load balance – the tasks need to perform similar amounts of work since the entire program would run only as quickly as the slowest (most heavily loaded) task.
- Data partitioning – the data needed by each task should be associated with that task.
- Data dependency – if the execution of one task depends on a result computed by a second, synchronization must be performed to make sure the result from the second task is available before the first attempts to use it.
- Testing and debugging – execution of multithreaded programs includes tasks that execute concurrently or in different interleavings. It is very difficult to ensure that a program works correctly for all possible interleavings.

Sample thread application

We consider an NFS (network file system) server. We may look in more detail at network file services later in the semester, but for now we just think of the server as having to respond to many read/write/list requests from clients.

Each client that contacts the server needs attention from the server process. A single thread of execution would be unreasonable for all but the most lightly-loaded servers, as requests would need to be served in order. This could potentially cause long delays in service time. Realistically, a new process or thread needs to be created for each request.

With "heavyweight" processes, each request from the network would require a full-fledged process creation. A series of requests would result in many short-duration processes being created and destroyed. Significant overhead would result.

With multithreading, a single NFS server task (a process) can exist, and a new thread can be created within the task for each request (*"pop-up" threads*). There is less overhead, plus if there are multiple CPUs available, threads can be assigned to each for faster performance.

User Threads *vs.* Kernel Threads

Threads may be implemented within a process (in "user space") or by the kernel (in "kernel space").

With *user threads*, the kernel sees only the main process, and the process itself is responsible for the management (creation, activation, switching, destruction) of the threads.

With *kernel threads*, the kernel takes care of all that. The kernel schedules not just the process, but the thread.

There are significant advantages and disadvantages to each. Some systems provide just one, some systems, such as Solaris, provide both.

One big thing is that with user threads, it is often the case that when a user thread blocks (such as for I/O service), the entire process blocks, even if some other threads could continue. With kernel threads, one thread can block while others continue.

The text describes these in a bit more detail, but also points out that nearly all modern operating systems support kernel threads.

---

pthreads, POSIX threads

We saw how to use `fork()` to create processes in Unix. We can also create threads with system calls. The text discusses POSIX threads, Java threads, and Windows threads. We will look at and make heaviest use of just one type: POSIX threads, usually known as “pthreads”.

Instead of creating a copy of the process like `fork()`, create a new thread, and that thread will call a function *within the current task.*

This new thread is running the same copy of the program and shares data with other threads within the process. Any global variables in your program are accessible to all threads. Local variables are directly accessible only to the thread in which they were created, though the memory can be shared by passing pointers to your thread functions.

The basic pthread functions are:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);

int pthread_join(pthread_t thread, void **status);

void pthread_exit(void *value_ptr);
```

- `pthread_create(3)` – As expected, this creates a new thread. It takes 4 arguments:
    1. The first is a pointer to a variable of type `pthread_t`. Upon return, this contains a thread identifier that is used later in `pthread_join()`.
    2. The second is a pointer to a `pthread_attr_t` that specifies thread creation attributes. In our initial examples, we pass in `NULL`, which specifies that the thread should be created using the system default attributes.
    3. The third argument is a pointer to a function that will be called when the thread is started. This function must take a single parameter of type `void *` and return `void *`.

4. The fourth parameter is the pointer that will be passed as the argument to the thread function.

- `pthread_exit(3)` – This causes the calling thread to exit. This is called implicitly if the thread function returns. Its argument is a return status value, which can be retrieved by `pthread_join()`.
- `pthread_join(3)` – This causes the calling thread to block until the thread with the identifier passed as the first argument to `pthread_join()` has exited. The second argument is a pointer to a location where the return status passed to `pthread_exit()` can be stored. In the pthreadhello program, we pass in `NULL`, and hence ignore the value.

See Example:
`/home/cs432/examples/pthreadhello`

On all three systems we will use (FreeBSD, Linux, Mac OS X), compilation of a pthreads program requires that we pass the flag `-pthread` to `gcc` to indicate that it should use thread-safe libraries (more on that idea later).

It is also a good idea to do some extra initialization, to make sure the system will allow your threads to make use of all available processors. It may, by default, allow only one thread in your program to be executing at any given time. If your program will create up to n concurrent threads, you should make the call:

```
pthread_setconcurrency(n+1);
```

somewhere before your first thread creation. The "+1" is needed to account for the original thread plus the n you plan to create.

You may also want to specify actual attributes as the second argument to `pthread_create()`. To do this, declare a variable for the attributes:

```
pthread_attr_t attr;
```

and initialize it with:

```
pthread_attr_init(&attr);
```

and set parameters on the attributes with calls such as:

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

Then, you can pass in `&attr` as the second parameter to `pthread_create()`.

To get a better idea about what context is shared by threads and what context is shared by processes, we consider this example:

See Example:
`/home/cs432/examples/what_shared`

See Example:
`/home/cs432/examples/proctree_threads`

This example builds a "tree" of threads to a depth given on the command line. It includes calls to `pthread_self()`. This function returns the thread identifier of the calling thread.

We will use pthreads in several class examples and in lab. We will examine a number of additional pthread functions as we need them in our examples and assignments.