Processes

- This lecture starts a class segment that covers processes, threads, and synchronization
  - These topics are perhaps the most important in this class.
  - You can rest assured that they will be covered in the exams.
- Today’s topics are processes and process management
  - What are the units of execution?
  - How are those units of execution represented in the OS?
  - How is work scheduled in the CPU?
  - What are the possible execution states of a process?
  - How does a process move from one state to another?
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
- A process is sometimes called a job or a task or a sequential process
- A sequential process is a program in execution
  - It defines the sequential, instruction-at-a-time execution of a program
  - Programs are static entities with the potential for execution

Process Components

- A process contains all of the state for a program in execution
  - An address space
  - The code for the executing program
  - The data for the executing program
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - A set of operating system resources
    » Open files, network connections, etc.
- A process is named using its process ID (PID)
**Process Address Space**

- **Stack**
- **Heap** (Dynamic Memory Alloc)
- **Static Data** (Data Segment)
- **Code** (Text Segment)

**Address Space**

- 0xFFFFFFFF
- 0x00000000

**Process State**

- A process has an **execution state** that indicates what it is currently doing
  - **Running**: Executing instructions on the CPU
    - It is the process that has control of the CPU
    - How many processes can be in the running state simultaneously?
  - **Ready**: Waiting to be assigned to the CPU
    - Ready to execute, but another process is executing on the CPU
  - **Waiting**: Waiting for an event, e.g., I/O completion
    - It cannot make progress until event is signaled (disk completes)

- As a process executes, it moves from state to state
  - Unix “ps”: **STAT** column indicates execution state
  - What state do you think a process is in most of the time?
  - How many processes can a system support?
Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its particular state
- The OS data structure representing each process is called the Process Control Block (PCB)
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware
PCB Data Structure

- The PCB contains a huge amount of information in one large structure
  - Process ID (PID)
  - Execution state
  - Hardware state: PC, SP, regs
  - Memory management
  - Scheduling
  - Accounting
  - Pointers for state queues
  - Etc.

- It is a heavyweight abstraction

struct proc (Solaris)

```c
typedef struct proc {
    /* Fields requiring no explicit locking */
    struct vnode *p_exec;          /* pointer to a.out vnode */
    struct as *p_as;               /* process address space pointer */
    struct plock *p_lockp;         /* ptr to proc struct's mutex lock */
    kmutex_t p_crlock;             /* lock for p_cred */
    struct cred *p_cred;           /* process credentials */
    /* Fields protected by pidlock */
    int p_swapcnt;                 /* number of swapped out lwps */
    struct p_stat;                 /* status of process */
    char p_wcode;                  /* current wait code */
    ushort_t p_pidflag;            /* flags protected only by pidlock */
    int p_wdata;                   /* current wait return value */
    pid_t p_ppid;                  /* process id of parent */
    struct proc *p_link;           /* forward link */
    struct proc *p_parent;         /* ptr to parent process */
    struct proc *p_child;          /* ptr to first child process */
    struct proc *p_sibling;        /* ptr to next sibling proc on chain */
    struct proc *p_psibling;       /* ptr to prev sibling proc on chain */
    struct proc *p_sibling_ns;     /* ptr to siblings with new state */
    struct proc *p_child_ns;       /* ptr to children with new state */
    struct proc *p_next;           /* active chain link next */
    struct proc *p_prev;           /* active chain link prev */
    struct proc *p_nextofkin;      /* gets accounting info at exit */
    struct proc *p_orphan;         /* orphan proc */
    struct proc *p_nextorph;       /* next orphan proc */
    /* Per process signal stuff */
    k_sigset_t p_signals;          /* signals pending to this process */
    k_sigset_t p_ignore;           /* ignore when generated */
    k_sigset_t p_siginfo;          /* gets signal info with signal */
    struct sigqueue *p_sigqueue;   /* queued siginfo structures */
    struct sigqhdr *p_sigqhdr;     /* hdr to sigqueue structure pool */
    struct sigqhdr *p_signhdr;     /* hdr to signotify structure pool */
    int p_stopsig;                 /* jobcontrol stop signal */
    /* Flags defined below */
    clock_t p_utime;               /* user time, this process */
    clock_t p_stime;               /* system time, this process */
    clock_t p_cutime;              /* sum of children's user time */
    clock_t p_cstime;              /* sum of children's system time */
    caddr_t *p_segacct;            /* segment accounting info */
    caddr_t p_brkbase;             /* base address of heap */
    size_t p_brksize;              /* heap size in bytes */
    /* Other things */
} proc;
```

January 16, 2007  CSE 120 – Lecture 3 – Processes  9

January 16, 2007  CSE 120 – Lecture 3 – Processes 10
struct proc (Solaris) (2)

- Special pre-process flag when set will fix misaligned memory
- Reference
- Per process lwp and kernel thread stuff
- Proc_t:
  - __defined:
    - proc_t;
  - __endif
  - __defined:
    - proc_t;
  - __endif
  - __defined:
    - proc_t;
  - __endif

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Doors
  - struct door_node
    - *p_unref_list;
    - *p_door_list; /* active doors */
  - struct door_node
    - *p_unref_list;
    - *p_door_list; /* active doors */

- Microstate accounting, resource usage, and real-time profiling
  - struct prof
    - p_prof;
    - p_prof_lock;

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Per process lwp and kernel thread stuff
  - struct proc
    - *p_schedctl; /* available schedctl structures */
    - *p_schedctl; /* available schedctl structures */

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Microstate accounting, resource usage, and real-time profiling
  - struct prof
    - p_prof;
    - p_prof_lock;

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Per process lwp and kernel thread stuff
  - struct proc
    - *p_schedctl; /* available schedctl structures */
    - *p_schedctl; /* available schedctl structures */

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Microstate accounting, resource usage, and real-time profiling
  - struct prof
    - p_prof;
    - p_prof_lock;

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Per process lwp and kernel thread stuff
  - struct proc
    - *p_schedctl; /* available schedctl structures */
    - *p_schedctl; /* available schedctl structures */

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Microstate accounting, resource usage, and real-time profiling
  - struct prof
    - p_prof;
    - p_prof_lock;

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Per process lwp and kernel thread stuff
  - struct proc
    - *p_schedctl; /* available schedctl structures */
    - *p_schedctl; /* available schedctl structures */

- The user structure
  - struct user
    - p_user;
    - (see sys/user.h)

- Microstate accounting, resource usage, and real-time profiling
  - struct prof
    - p_prof;
    - p_prof_lock;
PCBs and Hardware State

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the process' PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB
  - What happens to the code that is executing?
- The process of changing the CPU hardware state from one process to another is called a context switch
  - This can happen 100 or 1000 times a second!

State Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS has one queue for each state
  - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another
There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)

- PCBs are data structures dynamically allocated in OS memory
- When a process is created, the OS allocates a PCB for it, initialized, and placed on the ready queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is deallocated
Context Switch in Nachos

Here's an example of how an operating system implements context switching:

- Threads instead of processes, but ignore that for now.

There are three classes involved:

- Thread: thread abstraction
- Scheduler: ready queue management
- Interrupt: interrupt simulation

... plus an interrupt handler for the interval timer and a machine specific function `SWITCH` for saving and loading a thread's state.

January 16, 2007 CSE 120 – Lecture 3 – Processes

```
TimerInterruptHandler: {
  interrupt->YieldOnReturn();
}

void Thread::Yield () {
  Thread *nextThread;
  IntStatus oldLevel = interrupt->SetLevel(IntOff);
  nextThread = scheduler->FindNextToRun();
  if (nextThread != NULL) {
    scheduler->ReadyToRun(this);
    scheduler->Run(nextThread);
  }
  (void) interrupt->SetLevel(oldLevel);
}
```
Context Switch in Nachos

```c
void Scheduler::ReadyToRun (Thread *thread) {
    thread->setStatus(READY);
    readyList->Append((void *)thread);
}

Thread * Scheduler::FindNextToRun () {
    return (Thread *) readyList->Remove();
}

void Scheduler::Run (Thread *nextThread) {
    Thread *oldThread = currentThread;
    currentThread = nextThread;
    currentThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);
}
```

This is called *round-robin scheduling*

- A thread runs for a *quantum* (the time it takes for the interval timer to fire)
- How long should a quantum be? Why?
- How long after it finishes with a quantum does a thread run again?
- What happens if there is only one thread (or all the other threads are waiting)?
Process Creation

- A process is created by another process
  - Parent is creator, child is created (Unix: ps "PPID" field)
  - What creates the first process (Unix: init (PID 0 or 1))?
- In some systems, the parent defines (or donates) resources and privileges for its children
  - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process Creation: NT

- The system call on NT for creating a process is called, surprisingly enough, CreateProcess:
  `BOOL CreateProcess(char *prog, char *args)` (simplified)
- CreateProcess
  - Creates and initializes a new PCB
  - Creates and initializes a new address space
  - Loads the program specified by "prog" into the address space
  - Copies "args" into memory allocated in address space
  - Initializes the hardware context to start execution at main (or wherever specified in the file)
  - Places the PCB on the ready queue
Process Creation: Unix

- In Unix, processes are created using `fork()`
  
  ```c
  int fork()
  ```

- `fork()`
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a **copy** of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue

- Fork returns **twice**
  - Returns the child’s PID to the parent, “0” to the child
  - Huh?

```c
int main(int argc, char *argv[])
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s is %d\n", name, getpid());
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}
```

What does this program print?
Example Output

alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486

Duplicating Address Spaces

child_pid = 486
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}

child_pid = 0
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
Divergence

```
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Parent

```
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

Child

Example Continued

```
alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> a.out
Child of a.out is 498
My child is 498
```

Why is the output in a different order?
Why fork()?

- Very useful when the child...
  - Is cooperating with the parent
  - Relies upon the parent’s data to accomplish its task
- Example: Web server
  ```c
  while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
      Handle client request
    } else {
      Close socket
    }
  }
  ```

Process Creation: Unix (2)

- Wait a second. How do we actually start a new program?
  ```c
  int exec(char *prog, char *argv[])
  ```
- exec()
  - Stops the current process
  - Loads the program "prog" into the process’ address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It does not create a new process
- What does it mean for exec to return?
- What does it mean for exec to return with an error?
Process Creation: Unix (3)

- fork() is used to create a new process, exec is used to load a program into the address space
  - Why does NT have CreateProcess while Unix uses fork/exec?
- What happens if you run “exec csh” in your shell?
- What happens if you run “exec ls” in your shell? Try it.

- fork() can return an error. Why might this happen?

Process Termination

- All good processes must come to an end. But how?
  - Unix: exit(int status), NT: ExitProcess(int status)
- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete
- Note that a process does not need to clean up itself
  - Why does the OS have to do it?
wait() a second...

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use `wait()` (WaitForSingleObject)
  - Suspends the current process until a child process ends
  - `waitpid()` suspends until the specified child process ends
- Wait has a return value…what is it?
- Unix: Every process must be reaped by a parent
  - What happens if a parent process exits before a child?
  - What do you think a “zombie” process is?

Unix Shells

```c
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        waitpid(child_pid);
    }
}
```
Process Summary

- What are the units of execution?
  - Processes
- How are those units of execution represented?
  - Process Control Blocks (PCBs)
- How is work scheduled in the CPU?
  - Process states, process queues, context switches
- What are the possible execution states of a process?
  - Running, ready, waiting
- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination
- How are processes created?
  - CreateProcess (NT), fork/exec (Unix)

Next time...

- Read Chapter 4
- Homework #1 due
- Project 0 due