Administrivia

- If you have not already, please make sure to enroll in piazza
  - ~40 students have not yet

- Lab 1 and HW1 will be released soon (by Monday).
  - Lab 0 released yesterday
Last class

- Architecture support for OS
- We started by discussing how the OS operates
  - **Privileged mode**: To perform its role OS must have exclusive powers not available to users
    - Privileged mode; how does the switch happen?
  - **OS is not running unless there is an event**: 
    - OS schedules a user process to run then goes to sleep
    - It wakes up (who wakes it?) to handle events
    - Many types of events
  - **Program view and system calls**: program asks the OS when it needs a privileged operation
Today, we start discussing the first abstraction that enables us to virtualize (i.e., share) the CPU – processes!
What is virtualization?

- What is a virtual something?
  - Somehow not real? But still functional?

- Provide illusion for each program of own copy of resources
  - Lets say the CPU or memory; every program thinks it has its own
  - In reality, limited physical resources (e.g., 1 CPU)
    » They must be shared! (in time, or space)

- Virtualization frees up programs from worrying about sharing
  - The OS implements sharing, creating illusion of exclusive resources
    » Virtualization!

- Virtual resource provided as an object with defined operations on it. → Abstraction
Virtualizing the CPU -- Processes

- This lecture starts a class segment that covers processes, threads, and synchronization
  - Basis for Midterm and Project 1

- Today’s topics are processes and process management
  - How do we virtualize the CPU?
    » Virtualization: give each program the illusion of its own CPU
    » What is the magic? We only have one real CPU
  - How are applications represented in the OS?
  - How is work scheduled in the CPU?
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling

- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - Starts from the program, but also includes dynamic state
    - As the representative of the program, it is the “owner” of other resources (memory, files, sockets, …)

- How does the OS implement this abstraction?
  - How does it share the CPU?
How to support this abstraction?

- First, we’ll look at what state a process encapsulates
  - State of the virtual processor we are giving to each program

- Next we will talk about CPU time sharing
  - How to implement the process illusion

- Next, we discuss how the OS implements this abstraction
  - What data structures it keeps, and the role of the scheduler

- Finally, we see the process interface offered to programs
  - How to use this abstraction
A process contains all the state for a program in execution
- An address space containing
  - Static memory:
    - The code and input data for the executing program
  - Dynamic memory:
    - The memory allocated by the executing program
    - An execution stack encapsulating the state of procedure calls
- Control registers such as the program counter (PC)
- 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)
Address Space (memory abstraction)

- **Address Space**: 0x00000000 to 0xFFFFFFFF
- **Stack**: 0xFFFFFFFF
- **Heap** (Dynamic Memory Alloc)
- **Static Data** (Data Segment)
- **Code** (Text Segment)

- **SP**
- **PC**

- Dynamic
- Static
Process Execution 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)
Execution state (cont’d)

- As a process executes, it moves from state to state
  - Unix “ps -x”: 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?

<table>
<thead>
<tr>
<th>PROCESS STATE CODES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Here are the different values that the s, stat and state output specifiers (header &quot;S&quot;) use to indicate process states:</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>Z</td>
</tr>
</tbody>
</table>

For BSD formats and when the stat keyword is used, additional characters may be displayed:
| < | high-priority (not nice to other users) |
| N | low-priority (nice to other users) |
| L | has pages locked into memory (for real-time and custom I/O) |
| s | is a session leader |
| l | is multi-threaded (using CLONE_THREAD, like NPTL pthreads do) |
| + | is in the foreground process group. |
Execution State Graph

- New
- Ready
- Running
- Waiting
- Terminated

- Create Process
- Schedule Process
- I/O Done
- Unschedule Process
- Process Exit
- I/O, Page Fault, etc.
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)
- PCB contains all of the info about a process
PCB Data Structure

- PCB also is where OS keeps all of a process’ hardware execution state when the process is not running
  - Process ID (PID)
  - Execution state
  - Hardware state: PC, SP, regs
  - Memory management
  - Scheduling
  - Accounting
  - Pointers for state queues
  - Etc.

- 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
enum procstate { NEW, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
    uint sz; // Size of process memory (bytes)
    pde_t *pgdir; // Linear address of proc's pgdir
    char *kstack; // Bottom of kernel stack for this process
    enum procstate state; // Process state
    volatile int pid; // Process ID
    struct proc *parent; // Parent process
    struct trapframe *tf; // Trap frame for current syscall
    struct context *context; // Switch here to run process
    void *chan; // If non-zero, sleeping on chan
    int killed; // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16]; // Process name (debugging)
};
/**
 * One structure allocated per active process. It contains all
 * data needed about the process while the process may be swapped
 * out. Other per-process data (user.h) is also inside the proc structure.
 * Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
 */

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 */
        char    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_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;
        struct proc *p_nextorph;
        /*
         * Fields protected by p_lock
         */
        kcondvar_t p_cv;                /* proc struct's condition variable */
        kcondvar_t p_flag_cv;
        kcondvar_t p_lwpexit;           /* waiting for some lwp to exit */
        kcondvar_t p_holdlwps;          /* process is waiting for its lwps */
        ushort_t p_pad1;                /* unused */
        uint_t  p_flag;                 /* protected while set. */
        /* 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 */
        /*
         * Per process signal stuff.
         */
        k_sigset_t p_sig;               /* 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_signhdr;      /* hdr to sigqueue structure pool */
        struct sigqhdr *p_sigqhdr;      /* hdr to signotify structure pool */
        uchar_t p_stopsig;              /* jobcontrol stop signal */
        */
struct proc (Solaris) (2)

/* Special per-process flag when set will fix misaligned memory references. */
char p_fixalignment;

/* Per process lwp and kernel thread stuff */
id_t p_lwpid;  /* most recently allocated lwpid */
int p_lwpcnt;  /* number of lwps in this process */
int p_lwprcnt; /* number of not stopped lwps */
int p_lwpwait; /* number of lwps in lwp_wait() */
int p_zombcnt; /* number of zombie lwps */
int p_zomb_max; /* number of entries in p_zomb_tid */
id_t *p_zomb_tid; /* array of zombie lwps */
kthread_t *p_tlist; /* circular list of threads */

/* /proc (process filesystem) debugger interface stuff. */
k_sigset_t p_sigmask; /* mask of traced signals (/proc) */
k_fltset_t p_fltmask; /* mask of traced faults (/proc) */
struct vnode *p_trace; /* pointer to primary /proc vnode */
struct vnode *p_plist; /* list of /proc vnodes for process */
kthread_t *p_agenttp; /* thread ptr for /proc agent lwp */
struct watched_area *p_warea; /* list of watched areas */
ulong_t p_nwarea; /* number of watched areas */
struct watched_page *p_wpage; /* remembered watched pages (vfork) */
int p_nwpage; /* number of watched pages (vfork) */
int p_mapcnt; /* number of active pr_mappage()s */
struct proc *p_rlink; /* linked list for server */
kcondvar_t p_srwchan_cv;
size_t p_stkszsize; /* process stack size in bytes */

/* Microstate accounting, resource usage, and real-time profiling */
hrttime_t p_mireal; /* elapsed time sum over defunct lwps */
hrttime_t p_acct[NMSTATES]; /* microstate sum over defunct lwps */
struct rusage p_ru; /* rusage sum over defunct lwps */
struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
uintptr_t p_rprof_cyclic; /* ITIMER_REALPROF cyclic */
uint_t p_defunct; /* number of defunct lwps */

/* profiling. A lock is used in the event of multiple lwp's */
using the same profiling base/size.

/*
kmutex_t p_pflock; /* protects user profile arguments */
struct prof p_prof; /* profile arguments */

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

/*
/*

/* Doors. */
kthread_t *p_server_threads;
struct door_node *p_door_list; /* active doors */
struct door_node *p_unref_list;
kcondvar_t p_server_cv;
char p_unref_thread; /* unref thread created */

/* Kernel probes */
uchar_t p_tnf_flags;
struct proc (Solaris) (3)

/*
 * C2 Security (C2_AUDIT)
 */
caddr_t p_audit_data; /* per process audit structure */
kthread_t *p_aslwtp; /* thread ptr representing "aslwp" */
#if defined(i386) || defined(__i386) || defined(__ia64)
/*
 * LDT support.
 */
kmutex_t p_ldtlock; /* protects the following fields */
struct seg_desc *p_ldt; /* Pointer to private LDT */
struct seg_desc p_ldt_desc; /* segment descriptor for private LDT */
int p_ldtlimit; /* highest selector used */
#endif
size_t p_swrss; /* resident set size before last swap */
struct aio *p_aio; /* pointer to async I/O struct */
struct timer **p_timer; /* interval timers */
k_sigset_t p_notifsigs; /* signals in notification set */
kcndvar_t p_notifcv; /* notif cv to synchronize with aslwp */
timeout_id_t p_alarmid; /* alarm's timeout id */
uint_t p_sc_unblocked; /* number of unblocked threads */
struct vnode *p_sc_door; /* scheduler activations door */
caddr_t p_usrstack; /* top of the process stack */
uint_t p_stkprot; /* stack memory protection */
model_t p_model; /* data model determined at exec time */
struct lwpchan_data *p_lcp; /* lwpchan cache */
/*
 * protects unmapping and initialzation of robust locks.
 */
kmutex_t p_lcp_mutexinitlock;
#endif
utrap_handler_t *p_utraps; /* pointer to user trap handlers */
refstr_t *p_corefile; /* pattern for core file */
#endif defined(__ia64)
caddr_t p_upstack; /* base of the upward-growing stack */
size_t p_upstksize; /* size of that stack, in bytes */
uchar_t p_isa; /* which instruction set is utilized */
#endif
void *p_rce; /* resource control extension data */
struct task *p_task; /* our containing task */
struct proc *p_taskprev; /* ptr to previous process in task */
struct proc *p_tasknext; /* ptr to next process in task */
int p_lwpdaemon; /* number of TP_DAEMON lwps */
int p_lwpdwait; /* number of daemons in lwp_wait() */
kthread_t **p_tidhash; /* tid (lwpid) lookup hash table */
struct sc_data *p_schedctl; /* available schedctl structures */
} proc_t;
PCBs and Hardware State

- When a process is running, its dynamic state is in memory and some hardware registers
  - Hardware registers include
    - Program counter, stack pointer, control registers, data registers, ..
- When the OS stops running a process, it saves the current values of the registers (usually in PCB)
- When the OS restarts executing a process, it loads the hardware registers from the stored values in 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.
State Queues

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
PCBs and State Queues

- PCBs are data structures dynamically allocated in OS memory.
- When a process is created, the OS allocates a PCB for it, initializes it, and places it 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.
Process Creation

- A process is created by another process
  - Why is this the case?
  - 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: Windows

- The system call on Windows for creating a process is called, surprisingly enough, CreateProcess:
  ```c
  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 saved 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()`
  
  ```
  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
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

[well ~]$ gcc t.c
[well ~]$ ./a.out
My child is 486
Child of a.out is 486
Duplicating Address Spaces

Parent

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

Child

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

Parent

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

Child

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

[well ~]$ gcc t.c
[well ~]$ .a.out
My child is 486
Child of a.out is 486

[well ~]$ ./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

- 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
  - OS will handle this on its behalf
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 is a “zombie” process?
Unix Shells

code snippet:

```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 {
        if (!(run_in_background))
            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)