CS 318 Principles of Operating Systems

Fall 2019

Lecture 3: Processes

Prof. Ryan Huang
• Homework 1

• Lab 0
  - Due this Thursday

• Find your project group member soon
  - Lab 1 starts this Friday
  - Fill out Google form of group info
Recap: Architecture Support for OS

• Manipulating privileged machine state
  - CPU protection: dual-mode operation, protected instructions
  - Memory protection: MMU, virtual address

• Generating and handling “events”
  - Interrupt, syscall, trap
  - Interrupt controller, IVT
  - Fix fault vs. notify proceed

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions (sync)</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Interrupts (async)</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>

• Mechanisms to handle concurrency
  - Interrupts, atomic instructions
### Ugly Hardware Interface ➔ Elegant Software Interface/Abstraction

<table>
<thead>
<tr>
<th>Application programs</th>
<th>Operating system</th>
<th>Hardware</th>
</tr>
</thead>
</table>

- **Beautiful interface**
- **Ugly interface**

#### Tools
- vim
- GCC
- Chrome
- iTunes

---

**09/10/19**

**CS 318 – Lecture 3 – Processes**
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?
Process Abstraction

• The process is the OS abstraction for CPU (execution)
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
  - Sometimes also called a job or a task

• A 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
How Should the OS Manage Processes?

A diagram showing processes and their interaction with the CPU and OS.
Simple Process Management: One-at-a-time

• Uniprogramming: a process runs from start to full completion
• What the early batch operating system does - Load a job from disk (tape) into memory, execute it, unload the job
• Problem: low utilization of hardware
  - an I/O-intensive process would spend most of its time waiting for punched cards to be read
  - CPU is wasted
  - computers were very expensive back then
Multiple Processes

• Modern OSes run multiple processes simultaneously
Multiple Processes

• Modern OSes run multiple processes simultaneously

• Examples (can all run simultaneously):
  - gcc file_A.c – compiler running on file A
  - gcc file_B.c – compiler running on file B
  - vim – text editor
  - firefox – web browser

• Non-examples (implemented as one process):
  - Multiple firefox or tmux windows (still one process)
Multiprogramming (Multitasking)

- **Multiprogramming**: run more than one process at a time
  - Multiple processes loaded in memory and available to run
  - If a process is blocked in I/O, select another process to run on CPU
  - Different hardware components utilized by different tasks at the same time

- **Why multiple processes (multiprogramming)?**
  - Advantages: speed & hardware utilization
    - higher throughput
    - lower latency
Multiple processes can increase CPU utilization
- Overlap one process’s computation with another’s wait

Multiple processes can reduce latency
- Running A then B requires 100 sec for B to complete
- Running A and B concurrently makes B finish faster
- A is slower than if it had whole machine to itself, but still < 100 sec unless both A and B completely CPU-bound
Kernel’s View of Processes
A process contains all 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.
Process Address Space

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

Address Space

SP

PC

0x00000000

0xFFFFFFFF
A Process’s View of the World

• Each process has own view of machine
  - Its own address space
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)

• *(char *)0xc000 means different thing in P1 & P2

• Simplifies programming model
  - gcc does not care that firefox is running

• Sometimes want interaction between processes
  - Simplest is through files: vim edits file, gcc compiles it
  - More complicated: Shell/command, Window manager/app.
Naming A Process

- A process is named using its process ID (PID)
Implementing Process

• Keep a data structure for each process
  - **Process Control Block** (PCB)
  - Contains all of the info about a process

• Tracks *state* of the process
  - Running, ready (runnable), waiting, etc.

• Includes information necessary for execution
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
  - PCB is also maintained when the process is *not* running
    • Needed by context switch mechanism

• Various other data about the process
  - Credentials (user/group ID), signal mask, priority, accounting, etc.
  - It is a heavyweight abstraction
```c
/* 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 */
    struct proc *p_link;  /* forward link */
    struct proc *p_parent;    /* ptr to parent process */
    struct proc *p_sibling;  /* ptr to next sibling proc on chain */
    struct proc *p_sibling;  /* ptr to next sibling proc on chain */
    struct proc *p_pchild;  /* ptr to first child process */
    struct proc *p_pchild;  /* ptr to first child process */
    struct proc *p_pchild;  /* ptr to first child process */
    struct proc *p_pchild;  /* ptr to first child process */
    struct proc *p_child;   /* ptr to children with new state */
    struct proc *p_child;   /* ptr to children with new state */
    struct proc *p_child;   /* ptr to children with new state */
    struct proc *p_child;   /* 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_sibling_ns; /* ptr to siblings with new state */
    struct proc *p_child_ns; /* ptr to children with new state */
    struct proc *p_nextorph; /* */
    struct proc *p_pglink;  /* process group hash chain link next */
    struct proc *p_ppglink; /* process group hash chain link prev */
    struct sess *p_sessp;  /* session information */
    struct proc *p_pidp;   /* process ID info */
    struct proc *p_ppid;   /* process ID info */
    /* Fields protected by p_lock */
    kcondvar_t p_cv;       /* proc struct's condition variable */
    kcondvar_t p_flag_cv;  /* proc struct's condition variable */
    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 */
    kcondvar_t p_cv;       /* proc struct's condition variable */
    kcondvar_t p_flag_cv;  /* proc struct's condition variable */
    kcondvar_t p_lwpexit;  /* waiting for some lwp to exit */
    kcondvar_t p_holdlwps; /* process is waiting for its lwps */
    /* Per process signal stuff. */
    k_sigset_t p_sig;      /* signals pending to this process */
    k_sigset_t p_ignore;   /* ignore when generated */
    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 */
    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_lwpcrent; /* 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_dstset_t p_filmask; /* 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_upage; /* remembered watched pages (vfork) */
int p_mpage; /* number of watched pages (vfork) */
int p_mapcnt; /* number of active pr_mappage()s */
struct proc *p_rlink; /* linked list for server */
kcomvar_t p_srvchan_cv;
size_t p_stksize; /* process stack size in bytes */
/
/*
 * Microstate accounting, resource usage, and real-time profiling
 */
hrt ime_t p_mstart; /* hi-res process start time */
hrt ime_t p_mterm; /* hi-res process termination time */
hrt ime_t p_mireal; /* elapsed time sum over defunct lwps */
hrt ime_t p_acct[NMSTATES]; /* microstate sum over defunct lwps */
struct lrusage [NMSTATES]; /* lrusage 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; /* p_server_threads */
struct door_node *p_door_list; /* active doors */
struct door_node *p_unref_list; /* p_unref_list */
kcomvar_t p_server_cv; /* p_server_cv */
char p_unref_thread; /* p_unref_thread */
struct proc (Solaris) (3)

/* protects unmapping and initialization of robust locks. */
kmutex_t p_lcp_mutexinitlock;

/* pointer to user trap handlers */
utrap_handler_t *p_utraps;

/* pattern for core file */
refstr_t *p_corefile;

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 */

09/10/19
Process State

• A process has an execution state to indicate what it is 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 State Graph

New \rightarrow Ready

Create Process \rightarrow Ready

Ready \rightarrow I/O Done

I/O Done \rightarrow Waiting

Interrupt \rightarrow Waiting

Waiting \rightarrow Schedule Process

Schedule Process \rightarrow Running

Running \rightarrow Process Exit

Process Exit \rightarrow Terminated

Terminated \rightarrow New

I/O wait, etc. \rightarrow Waiting

Create Process

Process Exit

Schedule Process

I/O Done

Interrupt

Process Exit
State Queues

• How does the OS keep track of processes?

• Naïve approach: process list
  - How to find out processes in the ready state?
    • Iterate through the list
  - Problem: slow!

• Improvement: partition list based on states
  - OS maintains a collection of queues that represent the state of all processes
  - Typically, 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 moved from one queue into another
State Queues

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
• **Which process should kernel run?**
  - if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
  - if >1 runnable, must make scheduling decision

• **Scan process table for first runnable?**
  - Expensive. Weird priorities (small pids do better)
  - Divide into runnable and blocked processes

• **FIFO?**
  - Put threads on back of list, pull them from front:
    - Pintos does this—see ready_list in thread.c

• **Priority?**

• **Next class discusses in detail**
Preemption

• When to trigger a process scheduling decision?
  - Yield control of CPU
    • voluntarily, e.g., `sched_yield`
    • system call, page fault, illegal instruction, etc.
  - Preemption

• Periodic timer interrupt
  - If running process used up quantum, schedule another

• Device interrupt
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable

• Changing running process is called a context switch
  - CPU hardware state is changed from one to another
  - This can happen 100 or 1000 times a second!
Context Switch

Process $P_0$ is executing. An interrupt or system call occurs, which results in saving the state into PCB$_0$. The system then switches to process $P_1$, which is in an idle state. $P_1$ is reloaded from its PCB and begins executing. When $P_1$ completes its execution, it results in an interrupt or system call, which triggers the process switch back to $P_0$. $P_0$ then begins executing again.
Context Switch Details

• Very machine dependent. Typical things include:
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations

• Non-negligible cost
  - Save/restore floating point registers expensive
    • Optimization: only save if process used floating point
  - May require flushing TLB (memory translation hardware)

• Usually causes more cache misses (switch working sets)
User’s (Programmer’s) View of Processes
Creating a Process

• 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))?

• Parent defines 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
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
CreateProcess function

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the `CreateProcessAsUser` or `CreateProcessWithLogonW` function.

Syntax

```
BOOL WINAPI CreateProcess(
_In_opt_ LPCTSTR lpApplicationName,
_Inout_opt_ LPCTSTR lpCommandLine,
_In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
_In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
_In_ BOOL bInheritHandles,
_In_ DWORD dwCreationFlags,
_In_opt_ LPVOID lpEnvironment,
_In_opt_ LPCTSTR lpCurrentDirectory,
_Our_ LPSTARTUPINFO lpStartupInfo,
_Our_ LPPROCESS_INFORMATION lpProcessInformation
);
```
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**
  - Huh?
  - Returns the child’s PID to the parent, “0” to the child
NAME
fork -- create a new process

SYNOPSIS
#include <unistd.h>

pid_t
fork(void);

DESCRIPTION
fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child processes resource utilizations are set to 0; see setrlimit(2).

RETURN VALUES
Upon successful completion, fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

ERRORS
fork() will fail and no child process will be created if:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.

[EAGAIN] The system-imposed limit MAXUSER (see/param.h) on the total number of processes under execution by a single user would be exceeded.
What does this program print?

```c
#include <stdio.h>
#include <unistd.h>
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;
    }
}
```
$ gcc -o fork fork.c

$ ./fork

My child is 486

Child of ./fork is 486
The hardware contexts stored in the PCBs of the two processes will be identical, meaning the EIP register will point to the same instruction.
Divergence

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

Parent

child_pid = 486

Child

child_pid = 0

child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
$ gcc -o fork fork.c

$ ./fork

My child is 486

Child of ./fork is 486

$ ./fork

Child of ./fork is 498

My child is 498

Why is the output in a different order?
• Wait a second. How do we actually start a new program?

```c
int exec(char *prog, char *argv[])
int execve(const char *filename, char *const argv[], char *const envp[])
```

• **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?

• Warning: Pintos **exec** more like combined fork/exec
Manipulating File Descriptors

- **int dup2 (int oldfd, int newfd)**
  - Closes newfd, if it was a valid descriptor
  - Makes newfd an exact copy of oldfd
  - Two file descriptors will share same offset
    - (lseek on one will affect both)

- **int fcntl (int fd, int cmd, .../* arg */)**
  - `fcntl(fd, F_SETFD, FD_CLOEXEC)`
    - Makes file descriptor non-inheritable by spawned programs

**Example:** redirsh.c
- Loop that reads a command and executes it
- Recognizes command `< input > output 2> errlog`
Example: command < input > output 2> errlog

```c
void doexec (void) {
    int fd;
    if (infile) { /* non-NULL for "command < infile" */
        if ((fd = open (infile, O_RDONLY)) < 0) {
            perror (infile);
            exit (1);
        }
        if (fd != 0) {
            dup2 (fd, 0);
            close (fd);
        }
    }
    /* ... do same for outfile→fd 1, errfile→fd 2 ... */
    execvp (av[0], av);
    perror (av[0]);
    exit (1);
}
```

2: [https://www.cs.jhu.edu/~huang/cs318/fall18/code/redirsh.c](https://www.cs.jhu.edu/~huang/cs318/fall18/code/redirsh.c)
Pipes

• int pipe (int fds[2])
  - Returns two file descriptors in fds[0] and fds[1]
  - Data written to fds[1] will be returned by read on fds[0]
  - When last copy of fds[1] closed, fds[0] will return EOF
  - Returns 0 on success, -1 on error

• Operations on pipes
  - read/write/close – as with files
  - When fds[1] closed, read(fds[0]) returns 0 bytes
  - When fds[0] closed, write(fds[1]):
    • Kills process with SIGPIPE
    • Or if signal ignored, fails with EPIPE

• Example: pipesh.c
  - Sets up pipeline command1 | command2 | command3 ...

09/10/19 CS 318 – Lecture 3 – Processes
Why `fork()`?

- **Most calls to** `fork` **followed by** `exec`
  - could also combine into one `spawn` system call

- **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
    }
}
```
Why `fork()`?

- **Most calls to `fork` followed by `exec`**
  - could also combine into one `spawn` system call

- **Very useful when the child...**
  - Is cooperating with the parent
  - Relies upon the parent’s data to accomplish its task

- **Example: web server**

- **Example: shell**
pid_t pid; char **av;
void doexec () {
    execvp (av[0], av);
    perror (av[0]);
    exit (1);
}

/* ... main loop: */
for (;;) {
    parse_next_line_of_input (&av, stdin);
    switch (pid = fork ()) {
    case -1:
        perror ("fork"); break;
    case 0:
        doexec ();
    default:
        waitpid (pid, NULL, 0); break;
    }
}
Why \texttt{fork()}?

- \textbf{Most calls to} \texttt{fork} \textbf{followed by} \texttt{exec}
  - could also combine into one \texttt{spawn} system call

- \textbf{Very useful when the child…}
  - Is cooperating with the parent
  - Relies upon the parent’s data to accomplish its task

- \textbf{Example: Web server}

- \textbf{Real win is simplicity of interface}
  - Tons of things you might want to do to child: manipulate file descriptors, set environment variables, reduce privileges, ...
  - Yet \texttt{fork} requires no arguments at all
Spawning a Process Without fork

• Without fork, needs tons of different options for new process
  - Example: Windows CreateProcess system call
    • Also CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, ...

```c
BOOL WINAPI CreateProcess(
    _In_opt_ LPCTSTR lpApplicationName,
    _Inout_opt_ LPTSTR lpCommandLine,
    _In_opt_ LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_ BOOL bInheritHandles,
    _In_ DWORD dwCreationFlags,
    _In_opt_ LPVOID lpEnvironment,
    _In_opt_ LPCTSTR lpCurrentDirectory,
    _In_ LPSTARTUPINFO lpStartupInfo,
    _Out_ LPPROCESS_INFORMATION lpProcessInformation
);
```
• Why Windows use `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?
The Microsoft “Response”

“A fork() in the road”
- Andrew Baumann (Microsoft Research), Jonathan Appavoo, Orran Krieger (Boston University), Timothy Roscoe (ETH Zurich)
- In Proceedings of HotOS 2019
- Paper link (optional read)

Controversial argument against fork()
- Mainly from security perspective

7 GET THE FORK OUT OF MY OS!
We’ve described how fork is a relic of the past that harms applications and OS design. There are three things we must do to rectify the situation.
Deprecate fork. Thanks to the success of Unix, future
Take it with a grain of salt!
7.5 began to change [35, 78], but there is more to do [35].
Fix our teaching. Clearly, students need to learn about fork, however at present most text books (and we presume instructors) introduce process creation with fork [7, 35, 78]. This not only perpetuates fork’s use, it is counterproductive—the API is far from intuitive. Just as a programming course would not today begin with goto, we suggest teaching either posix_spawn() or CreateProcess(), and then introducing fork as a special case with its historic context (§2).
All good processes must come to an end. But how?
- Unix: `exit(int status)`, Windows: `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

• Unix `wait(int *wstatus)` (Windows: `WaitForSingleObject`)
  - Suspends the current process until any 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?
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 Chapters 7, 8

• Lab 0 due

• Lab 1 starts