SE350: Operating Systems

Lecture 3: Process Management

Outline

- Safe control transfer
  - How do we switch from one mode to the other?
  - What should hardware provide?
- Native control of process
  - Can processes create other processes?
  - `fork()`, `exec()`, `wait()`, `signal()`

Recall: Three Types of Mode Transfer

- Syscall
  - Process requests system service, e.g., `exit`
    - Like function call, but outside process
  - Process does not have address of system function to call
    - Like a Remote Procedure Call (RPC) – for later
  - OS marshalls syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch, e. g., Timer, I/O device
    - Independent of user process
- Trap or exception
  - Internal synchronous event in process triggers context switch, e.g., protection violation (segmentation fault), divide by zero, …

Implementing Safe Mode Transfers

- It should be impossible for buggy or malicious user program to cause kernel to corrupt itself
  - Controlled transfer into kernel (e.g., interrupt vector table)
  - Separate kernel stack

- Carefully constructed kernel code should pack up user process state and set it aside
  - Details depend on the machine architecture

Need for Separate Kernel Stacks

- Kernel cannot put anything on user stack (Why?)
  - Reliability: what if user program's SP is not valid?
  - Security: what if other threads in user process change kernel's return address?

- Two-stack model
  - Kernel keeps separate stack for each thread in kernel memory (in addition to user stack in user memory)

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

Stack Pointer

```
tmp = 1
ret = ext
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

Stack Pointer

```
tmp = 1
ret = ext
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

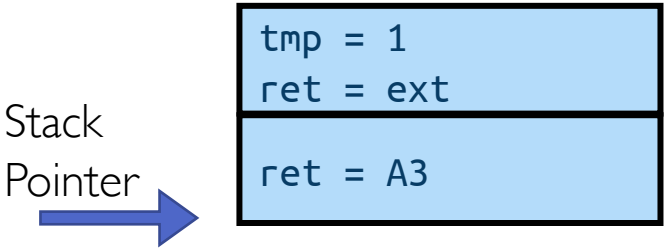
Stack Pointer

```
tmp = 1
ret = ext
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

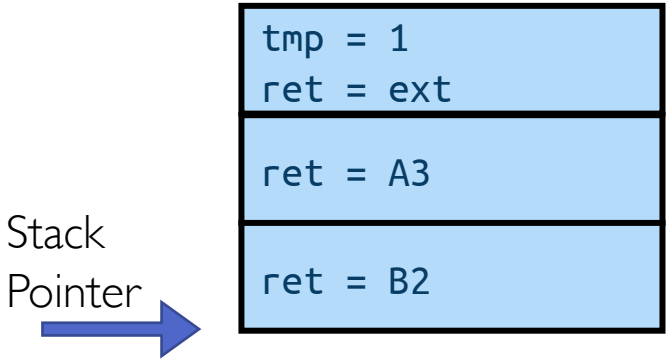
```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

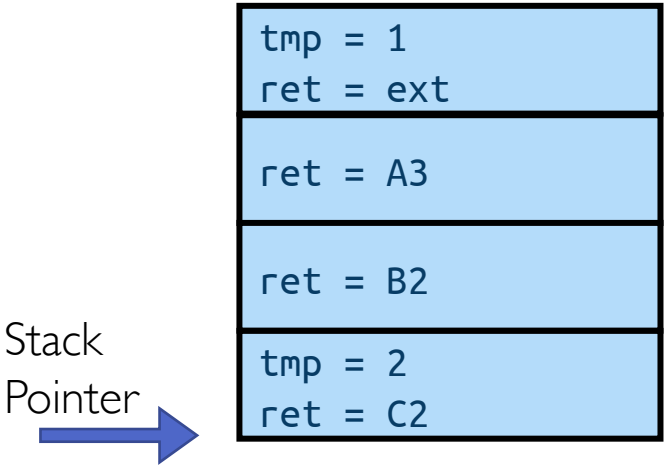
```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

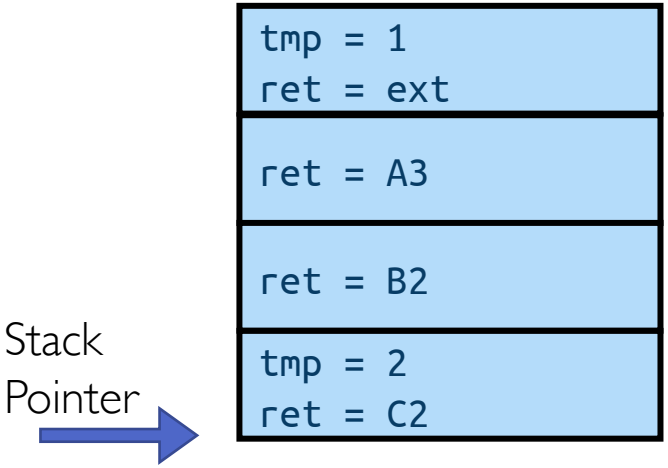
```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

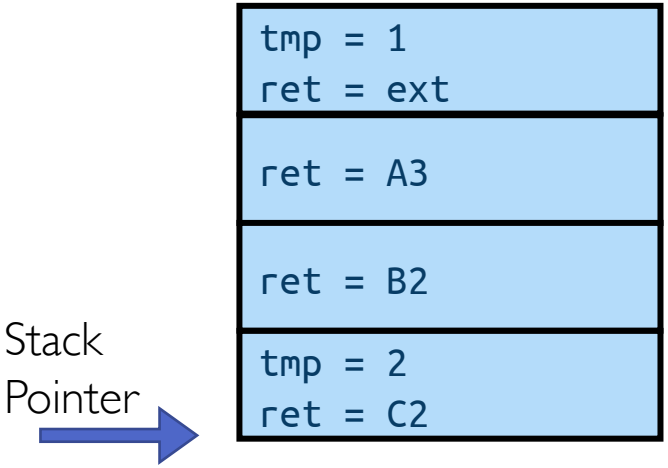



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

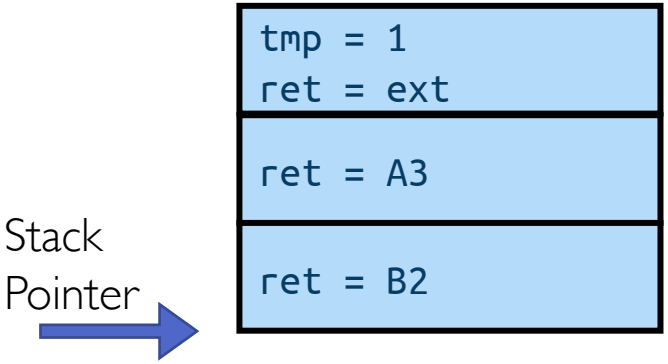



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

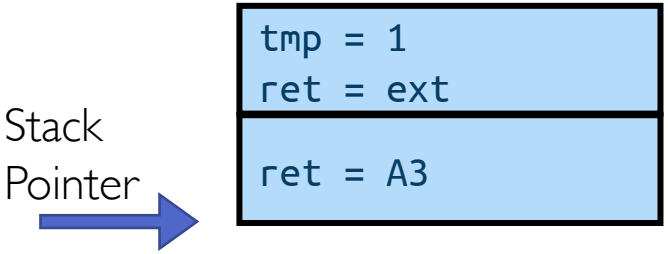



```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```




```
> 2
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

Stack Pointer

```
tmp = 1
ret = ext
```

```
> 21
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

Stack Pointer

```
tmp = 1
ret = ext
```

```
> 21
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

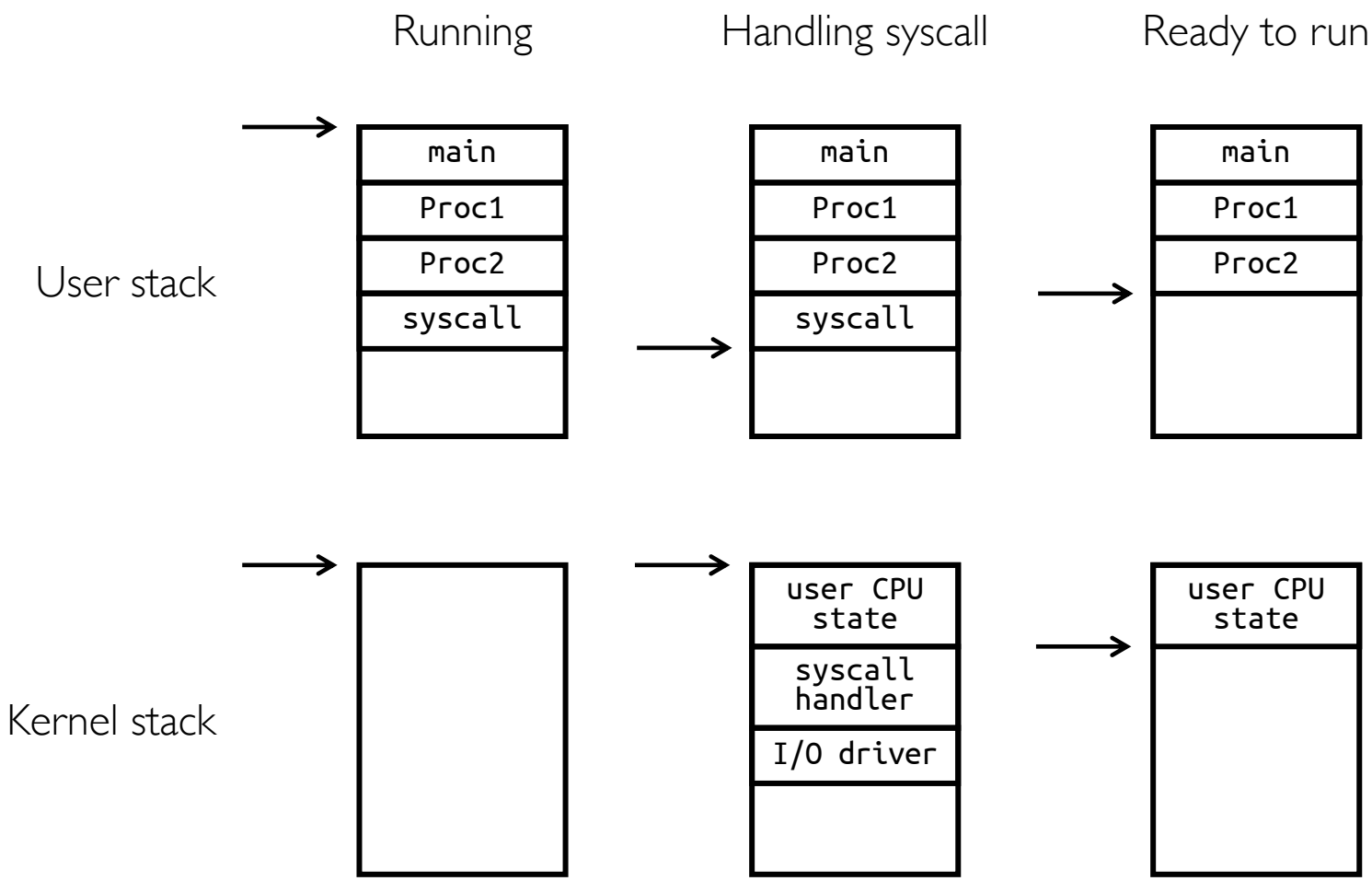
Stack Example

```
A0:  A(int tmp) {
A1:      if (tmp<2)
A2:      B();
A3:      printf(tmp);
A4:  }
B0:  B() {
B1:      C();
B2:  }
C0:  C() {
C1:      A(2);
C2:  }
     A(1);
ext:
```

```
> 21
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

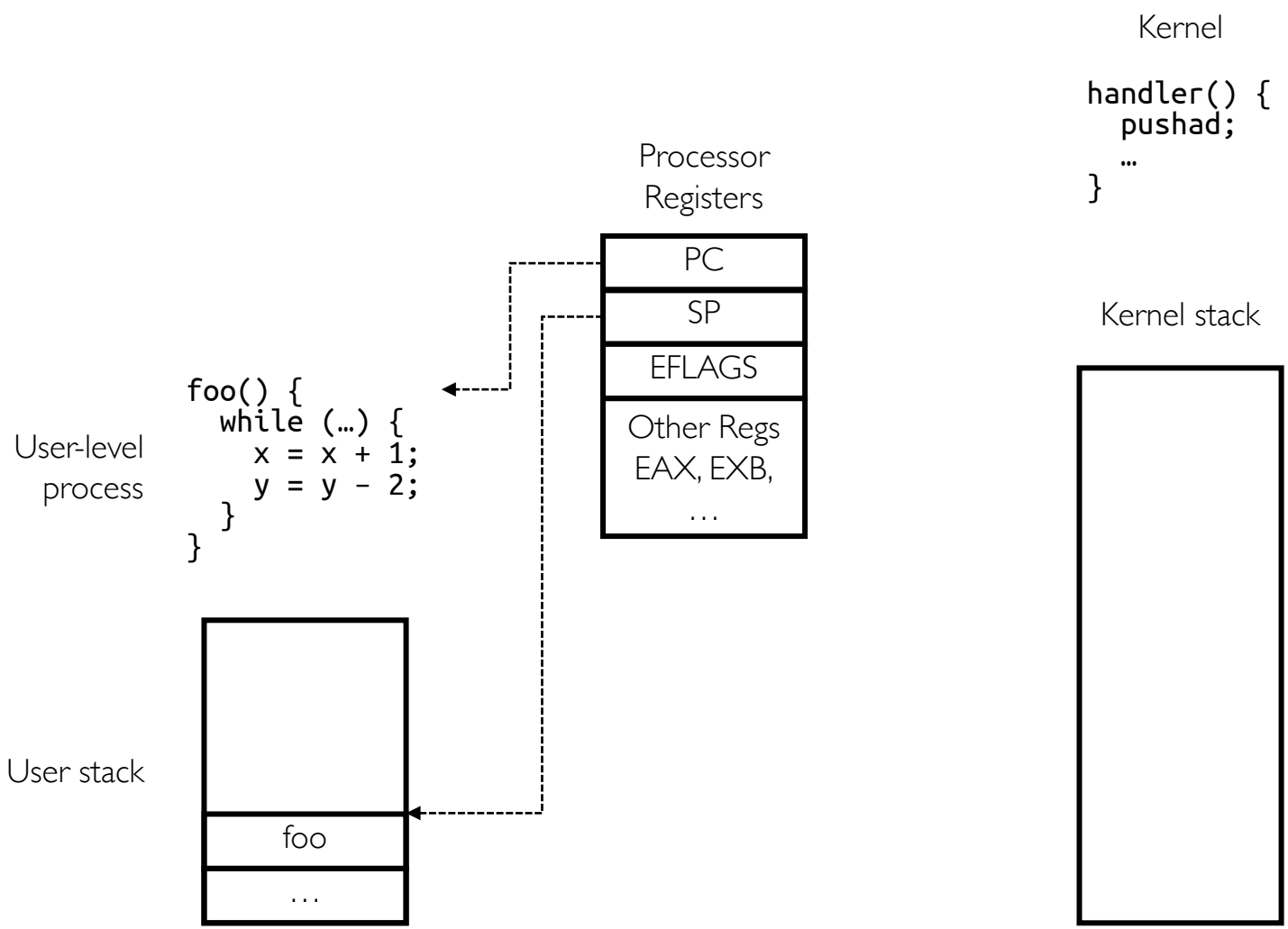
Two-Stack Model



Interrupt Masking

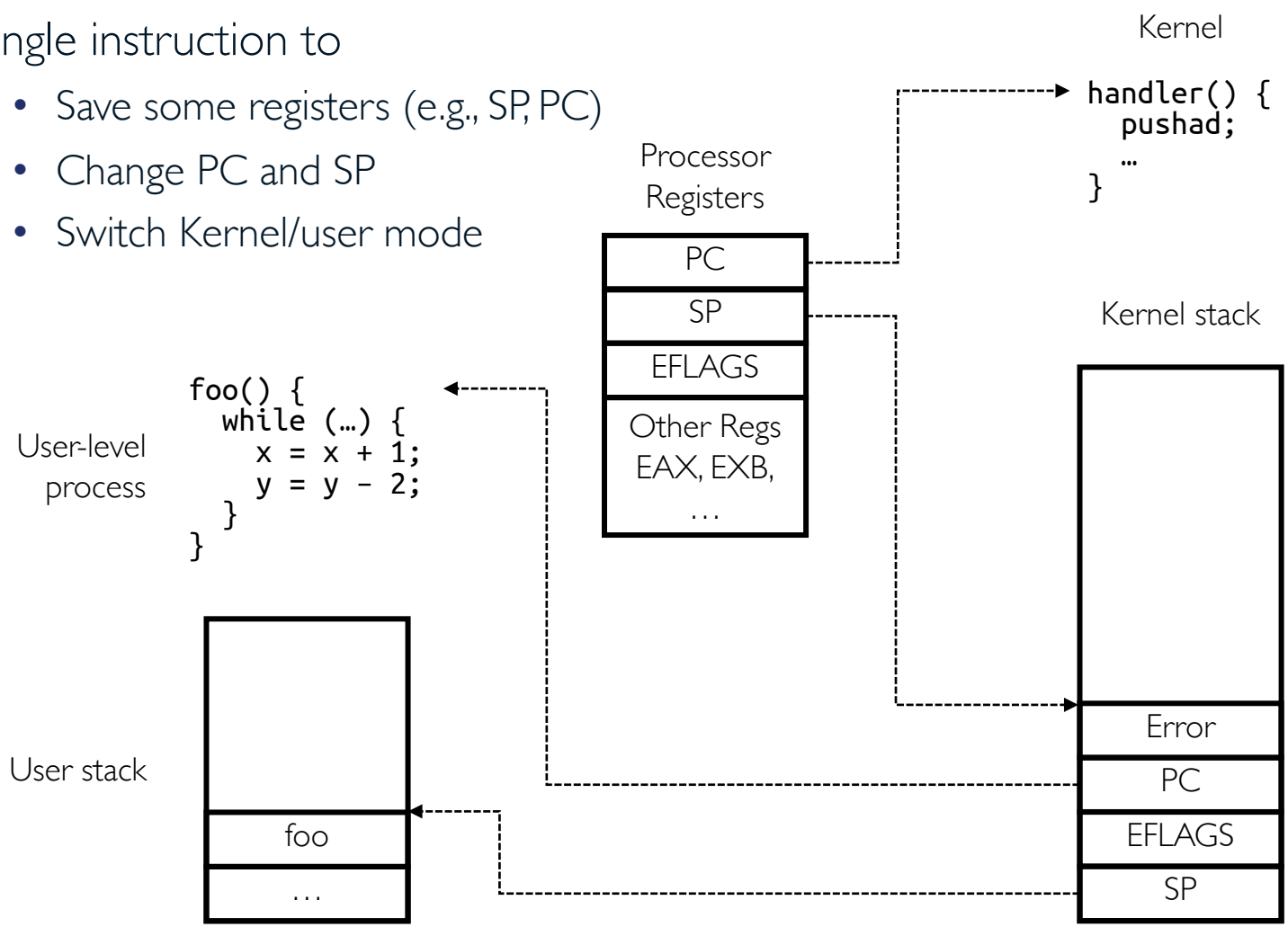
- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - E.g., when determining next process/thread to run
  - On x86
    - `cli`: disable interrupts
    - `sti`: enable interrupts
    - Only applies to current CPU (on a multicore)
- We will need this to implement synchronization (more on this later)

Atomic Transfer of Control



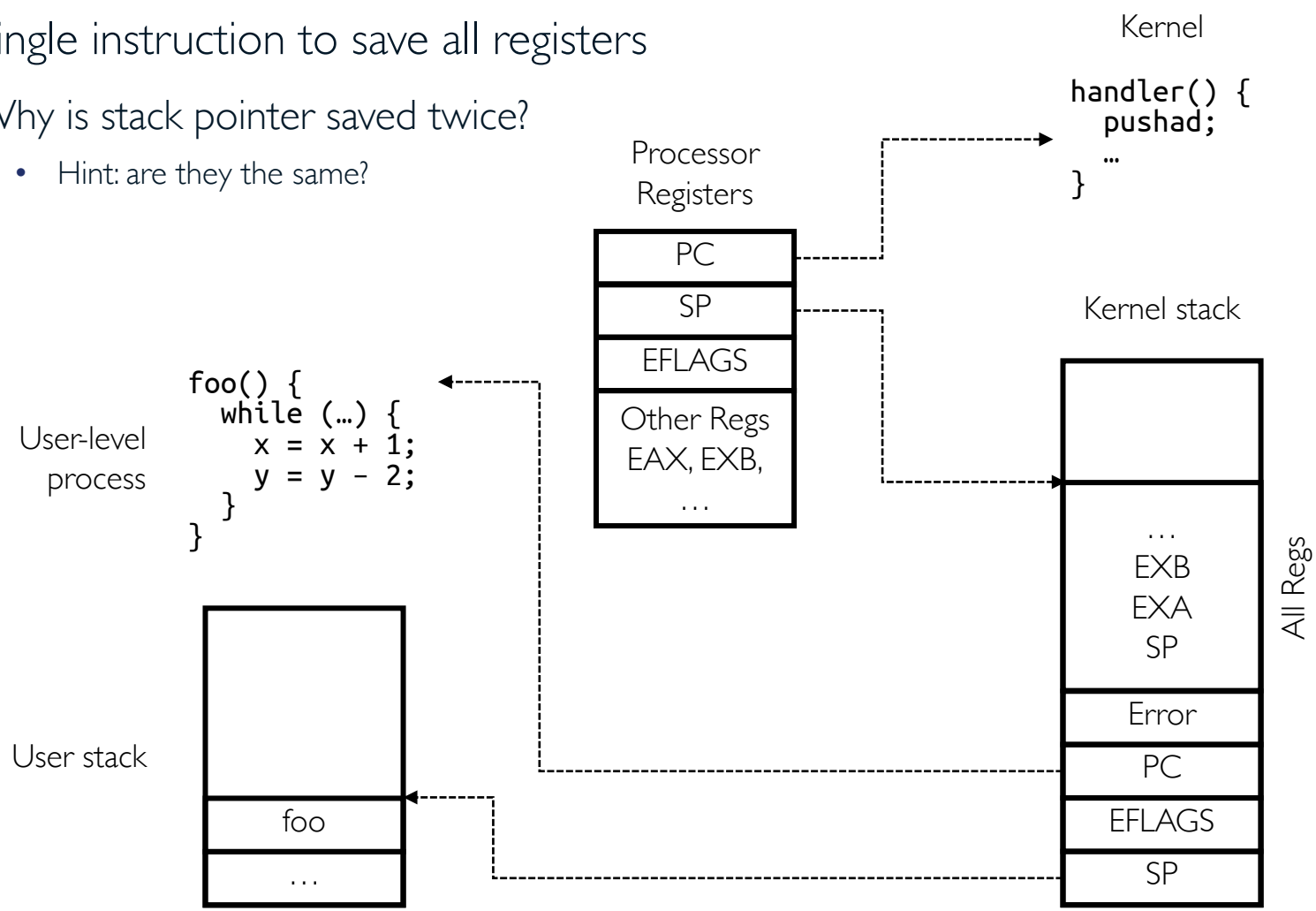
Atomic Transfer of Control (cont.)

- Single instruction to
  - Save some registers (e.g., SP, PC)
  - Change PC and SP
  - Switch Kernel/user mode



Atomic Transfer of Control (cont.)

- Single instruction to save all registers
- Why is stack pointer saved twice?
  - Hint: are they the same?



Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments (copy before check)
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

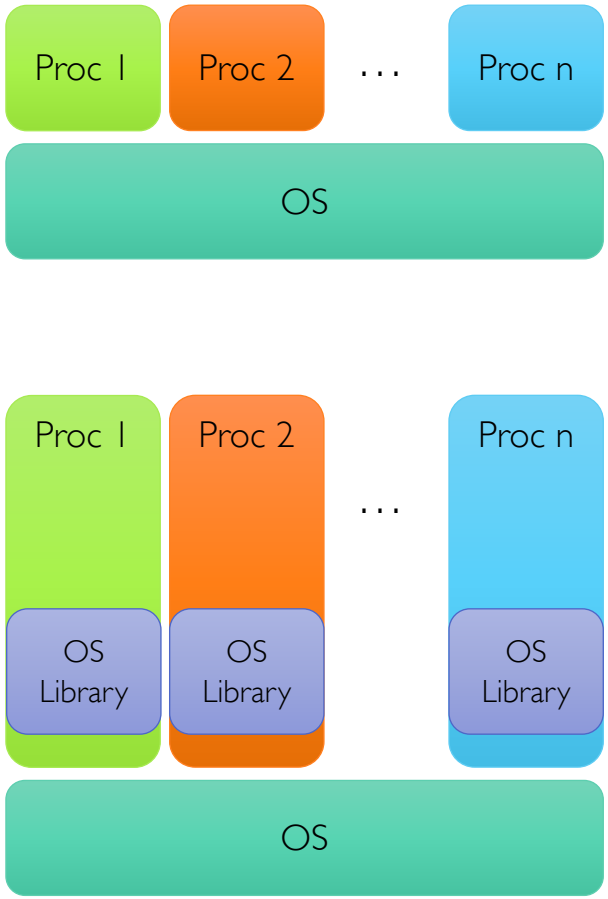
At the End of Handler

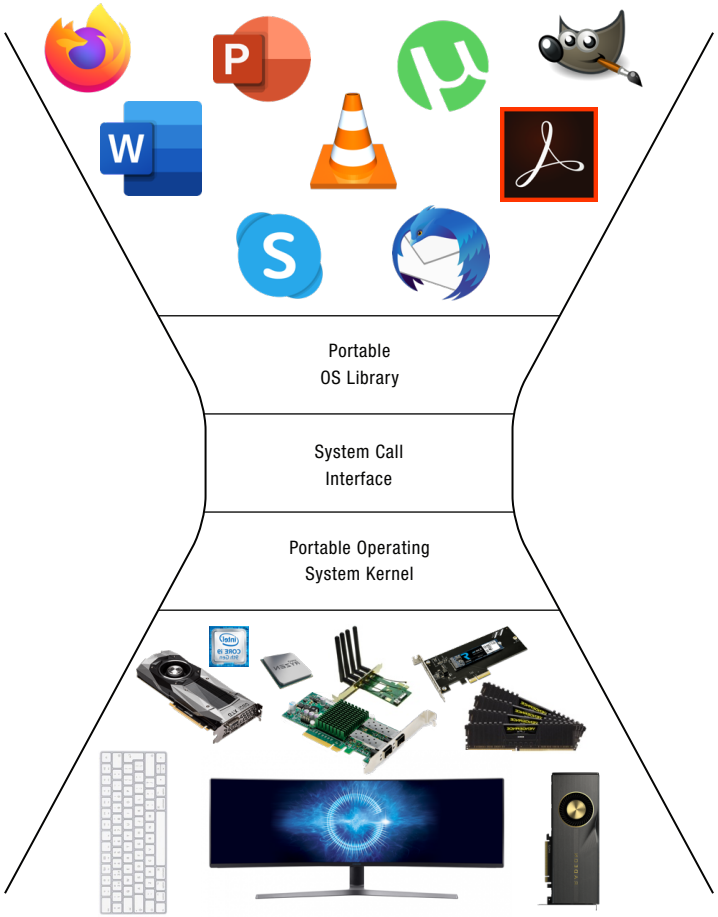
- Handler restores saved registers
- Atomically return to (interrupted) process/thread
  - Restore PC, SP
  - Restore processor status
  - Switch to user mode

How Does Kernel Provide Services?

- You said that applications request services from the operating system via syscall, but …
  - I've been writing all sort of useful applications and I never ever saw a "syscall" !!!
- That's right
- It was buried in the programming language runtime library (e.g., libc.a)
  - … Layering

OS Run-Time Library

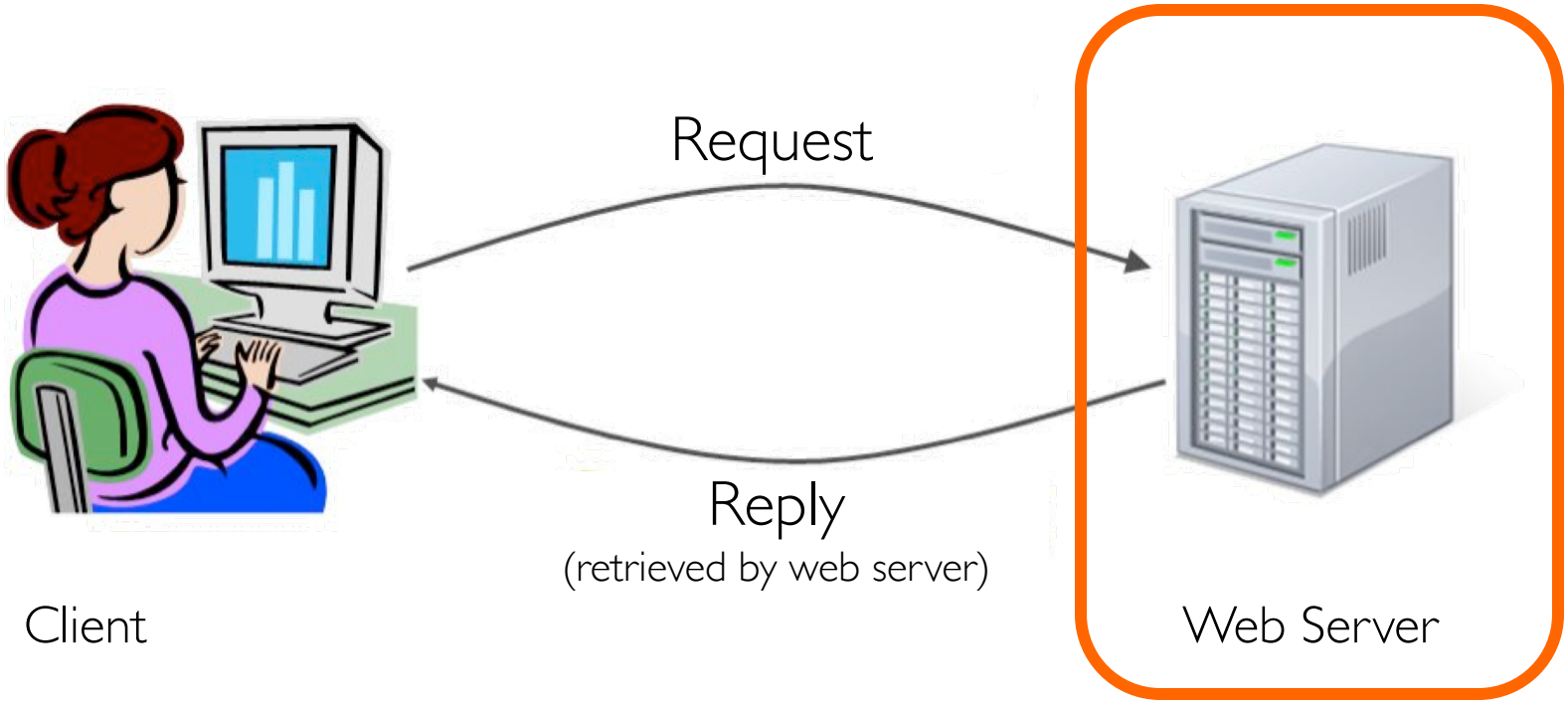




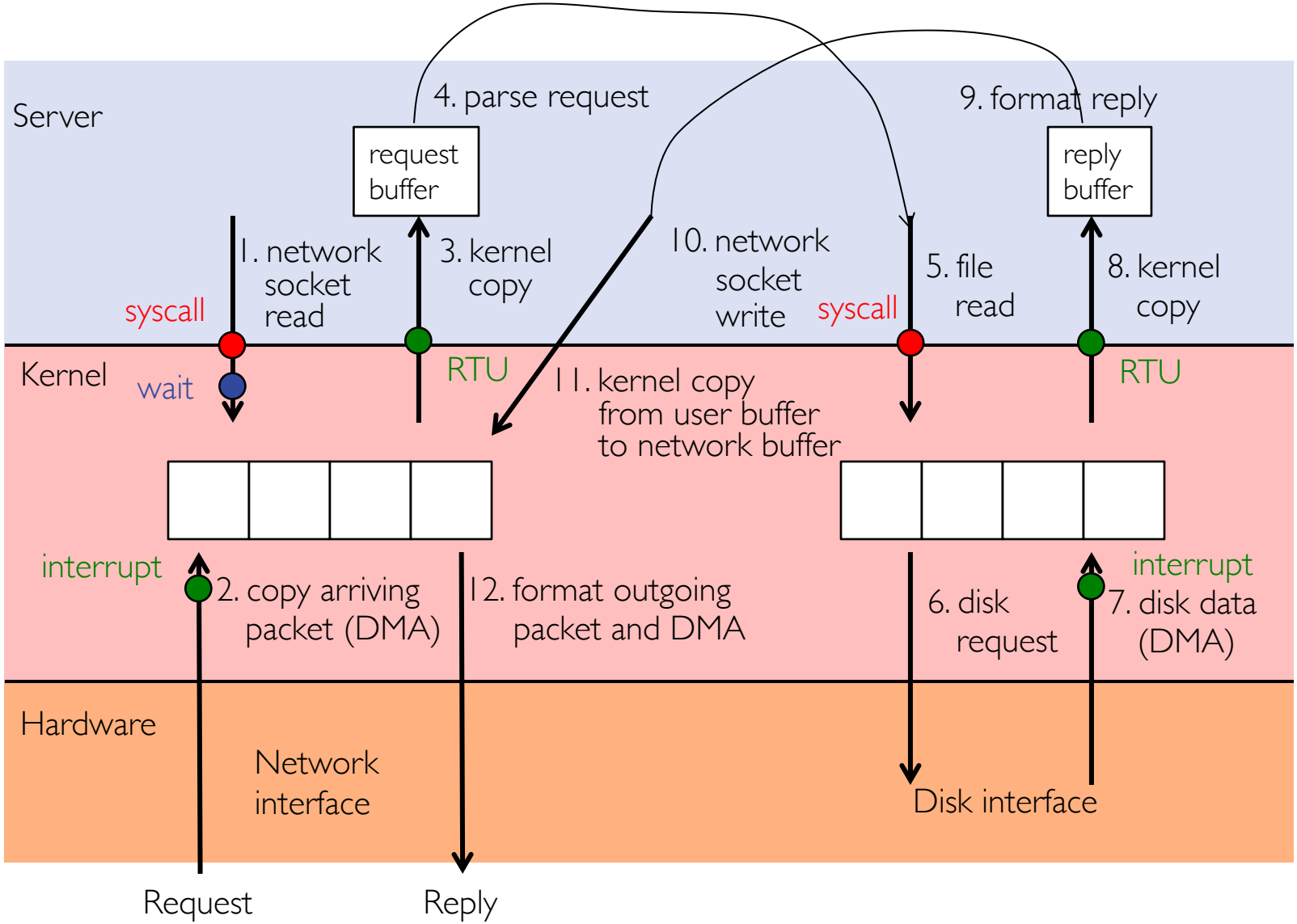
Kernel to User Mode Switch

- New process/new thread start
  - Jump to first instruction in program/thread
- Return from interrupt, exception, system call
  - Resume suspended execution
- Process/thread context switch
  - Resume some other process
- User-level *upcall* (UNIX *signal*)
  - Asynchronous notification to user program
    - Preemptive user-level threads
    - Asynchronous I/O notification
    - Interprocess communication
    - User-level excepting handling
    - User-level resource allocation

Putting it Together: Web Server



Putting it Together: Web Server (cont.)



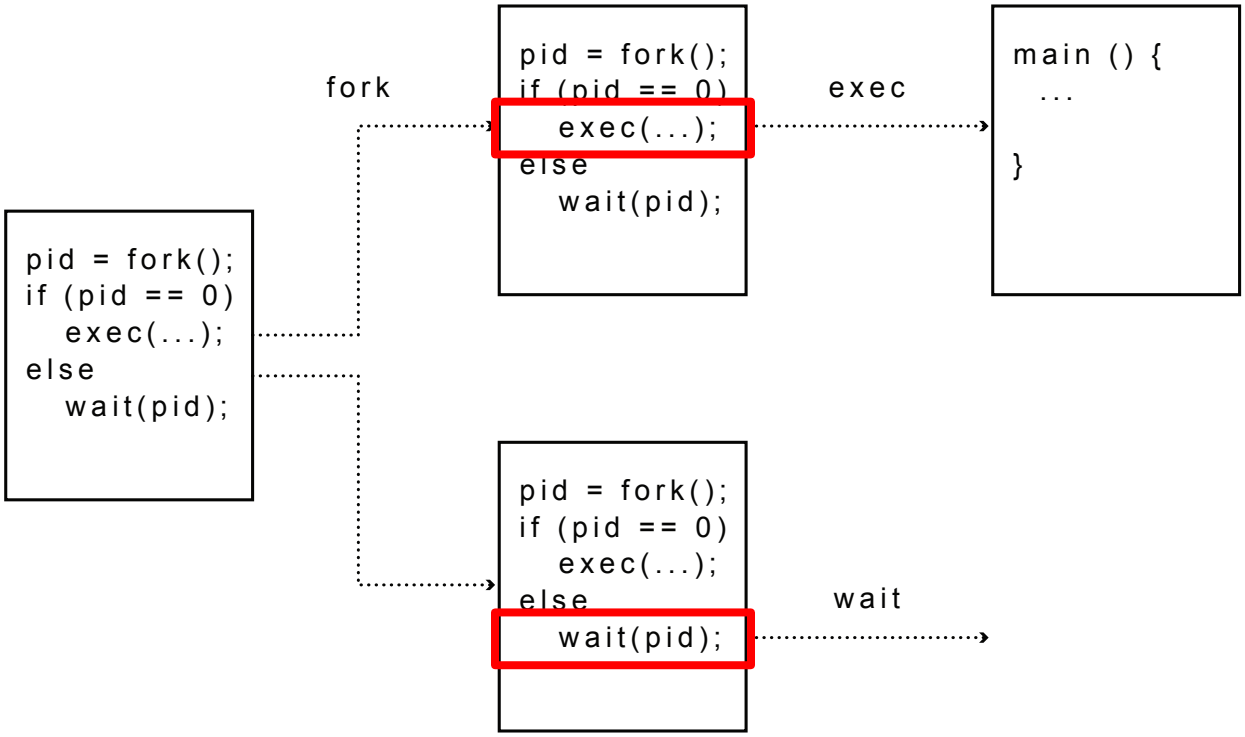
Summary of Hardware Support for OS

- Privilege modes
- Privileged instructions
- Memory translation
- Processor exceptions
- Timer interrupts
- Device interrupts
- Interprocessor interrupts
- Interrupt masking
- System calls
- Return from interrupt

Can a Process Create a Process ?

- `fork()` system call creates copy of current process with new unique process ID (PID)
- Return value from `fork()` is integer
  - When > 0
    - Running in (original) parent process
    - return value is PID of new child
  - When = 0
    - Running in new child process
  - When < 0
    - Error! Must be handled somehow
    - Running in original process
- All state of original process duplicated in both parent and child!
  - Memory, file descriptors (more on this later), etc…

UNIX Process Management



UNIX Process Management (cont.)

- `fork()`
  - Syscall to create copy of current process and start it
- `exec()`
  - Syscall to change program being run by current process
- `wait()`
  - Syscall to wait for process to finish
- `signal()`
  - Syscall to send notification to another process (e.g., `SIGKILL, SIGINT`)

Example

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();              /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                    /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {           /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();  // create a child process

    if (child_pid == 0) {
       exec(prog, args);        // I'm the child process. Run program
      // NOT REACHED
    } else {
       wait(child_pid); // I'm the parent, wait for child
       return 0;
    }
}
```

Summary

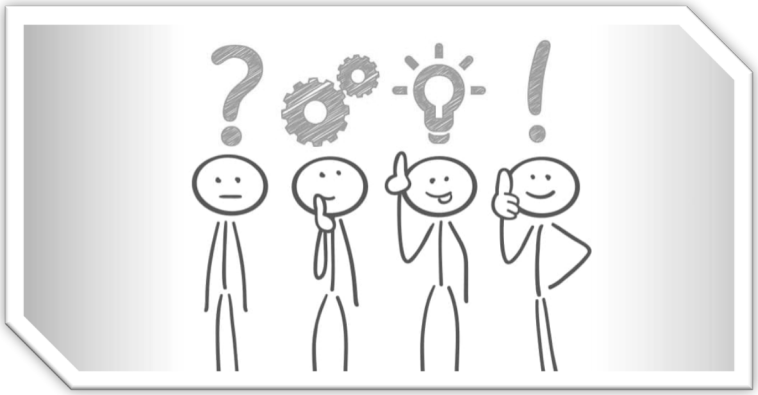
- Safe control transfer
  - How do we switch from one mode to the other?
  - What should hardware provide?
- Native control of process
  - Can processes create other processes?
  - `fork()`, `exec()`, `wait()`, `signal()`

Feedback

https://forms.gle/L6oS18zZApNF3ERb8

- Will be available until the end of term
- Will be checked regularly

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny