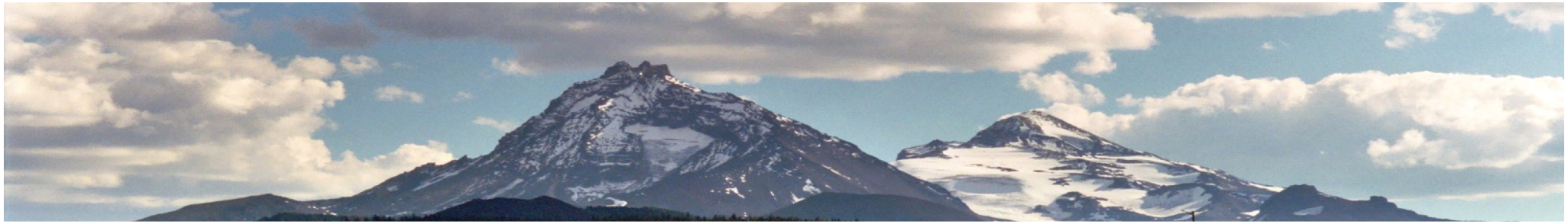


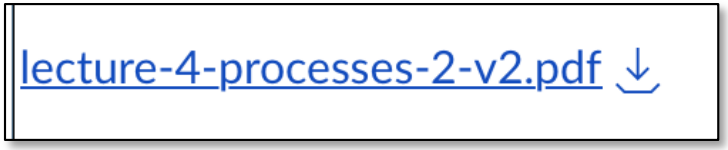
CS 415
Operating Systems
Processes

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

- Project 1 due in 10 days
- Labs: this week will be focusing on Q&A and project 1
- Read OSC Chapter 3
- Lecture slides versioning



- Get familiar with the Linux man pages

  https://man7.org/linux/man-pages/index.html

Outline

- Process concept
- Process operation
- System calls to create processes
- Process management
- Process scheduling

Overview of Processes

- We have programs, so why do we need processes?
- Questions that we explore
  - How are processes created?
    - from binary program to executing process
  - How is a process represented and managed?
    - process creation, process control block
  - How does the OS manage multiple processes?
    - process state, ownership, scheduling
  - How can processes communicate?
    - interprocess communication, concurrency, deadlock

Process Creation

- What happens?
  - New process object in the kernel is created
    - build process data structures
  - Allocate address space (abstract resource)
    - later, allocate actual memory (physical resource)
  - Add to ready queue
    - make it runnable

- who created the first process and how?
  - firmware → bootloader → kernel → first process
- Is the OS a process itself? Well, sort of …
- *Init* is the first user-world process in Linux originally
  https://en.wikipedia.org/wiki/Booting_process_of_Linux
- *Systemd* replaced *Init* to manage entire boot process
  https://en.wikipedia.org/wiki/Systemd

Why care about creating new processes?

- Clearly, we care about creating processes because that is how all new processes are created
  - Starts with system and forks to get your new process
- Your processes can create other processes!!!
- But why is that important?
- Process creation increases concurrency
- Concurrent execution is more powerful

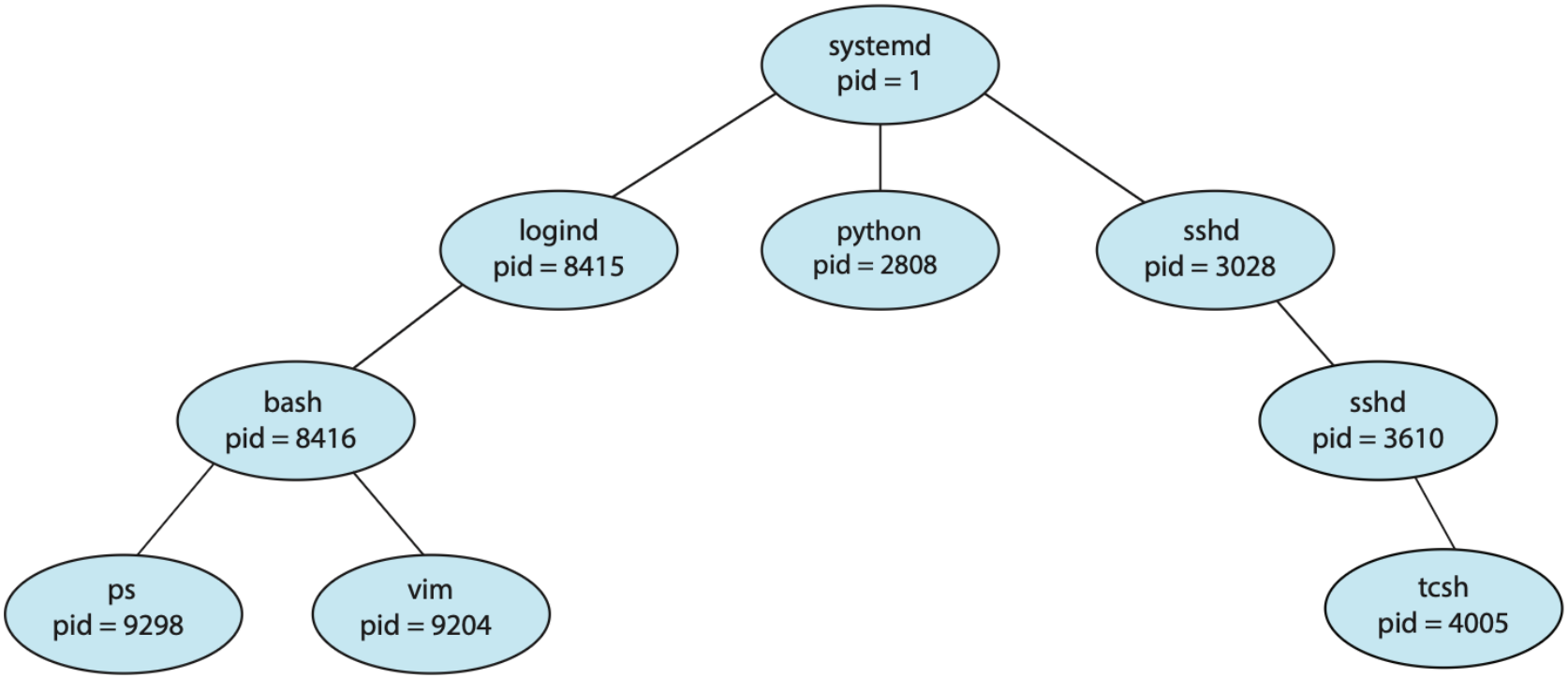
A process is the unit of task in a modern computing system.

Process Termination

- Process executes last statement and asks the operating system to delete it (*exit()*)
  - Output data from child to parent (via *wait()*)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (*abort()*)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - some operating system do not allow child to continue if parent terminates
    - all children terminated - cascading termination

A tree of processes on a typical system

systemd replaces *init*
in Linux versions now



Process Creation System Calls

- How are processes brought into existence?
- *fork(2)*
  - Copy the address space of parent
- *vfork(2)*
  - Use the parent's address space
  - Share address space between parent and child
    - includes data segment (and mapped physical memory)
  - Calling parent blocks until child terminates or calls *exec()*
  - Faster than *fork()*

C Program Forking Separate Process

Parent process

```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

C Program Forking Separate Process

Parent process

```
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

C Program Forking Separate Process

Parent process

```
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

C Program Forking Separate Process

Parent process

```
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Parent gets child pid > 0

Child process

```
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Child always gets 0

C Program Forking Separate Process

Parent process

```
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Parent gets child pid > 0

Child process

```
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Child always gets 0

C Program Forking Separate Process

Parent process

```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Child process

```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
    printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

C Program Forking Separate Process

Parent process

```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Child process

```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

C Program Forking Separate Process

Parent process

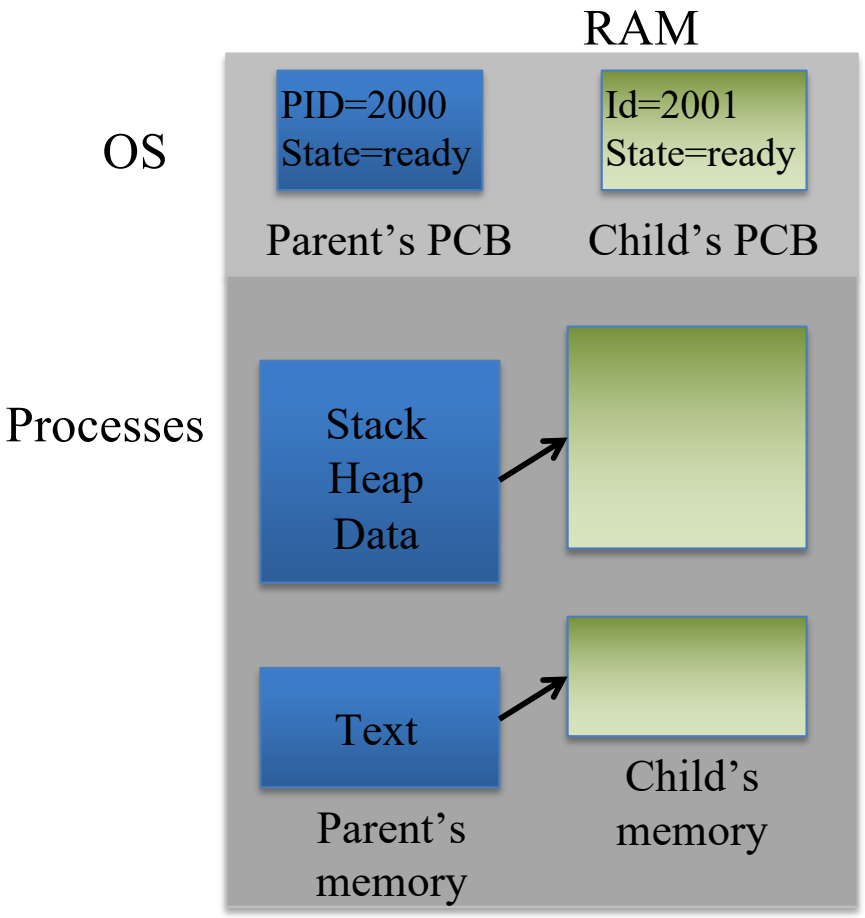
```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

Child process

```c
// headers omitted
int main() {
    printf("Hello before fork!\n);
    pid_t pid = fork();
    if (pid < 0) {
        return 1; // fork failed
    }
    elseif (pid == 0) // child process
        printf("Hello from child!\n);
    } else {// Parent process
        printf("Hello from parent!\n);
    }
    printf("Both parent and child
reach this point.\n");
    return 0;
}
```

See code example: fork.c

Forking a Child Process



Effects in memory after parent calls *fork()*

The kernel creates a **new PCB** for the child process.
This includes:

- A **new process ID (PID)**
- **Parent PID** (link to the parent)
- **Copy of CPU registers**, program counter (PC), stack pointer, etc.
- **Copy of memory mappings** (code, data, heap, stack segments)
- **Copied file descriptors** (pointing to same open files)
- **Scheduling info**, signal handlers, etc.

Why memory mapping?
(answer in next slide)

Copying the Parent's Memory

- While it is the case that the child process is initialized with the process address space of the parent, is the actual (logical) memory copied?
  - Logically, it is
  - In practice, it is not
- If the parent's memory (i.e., what it is actually using) is large, the copying cost would be huge
- In practice, what happens is that only the memory reference and (to be) modified is copied
  - This is called "copy on write" (much more efficient)
  - Involves magic in the physical memory management
  - More to come when we get to memory topic …

Process Wait System Calls

- How to let a parent wait for their child ?
- *wait(2)*
  - Waits for any child to finish.
  - Blocks the parent until one child exits.
  - Returns the PID of the terminated child.
  - Also return child exit status from argument
- *waitpid(2)*
  - Waits for the specified child to finish (need to provide child pid).
  - Also return child exit status from argument
  - Can be configured to either block or non-block

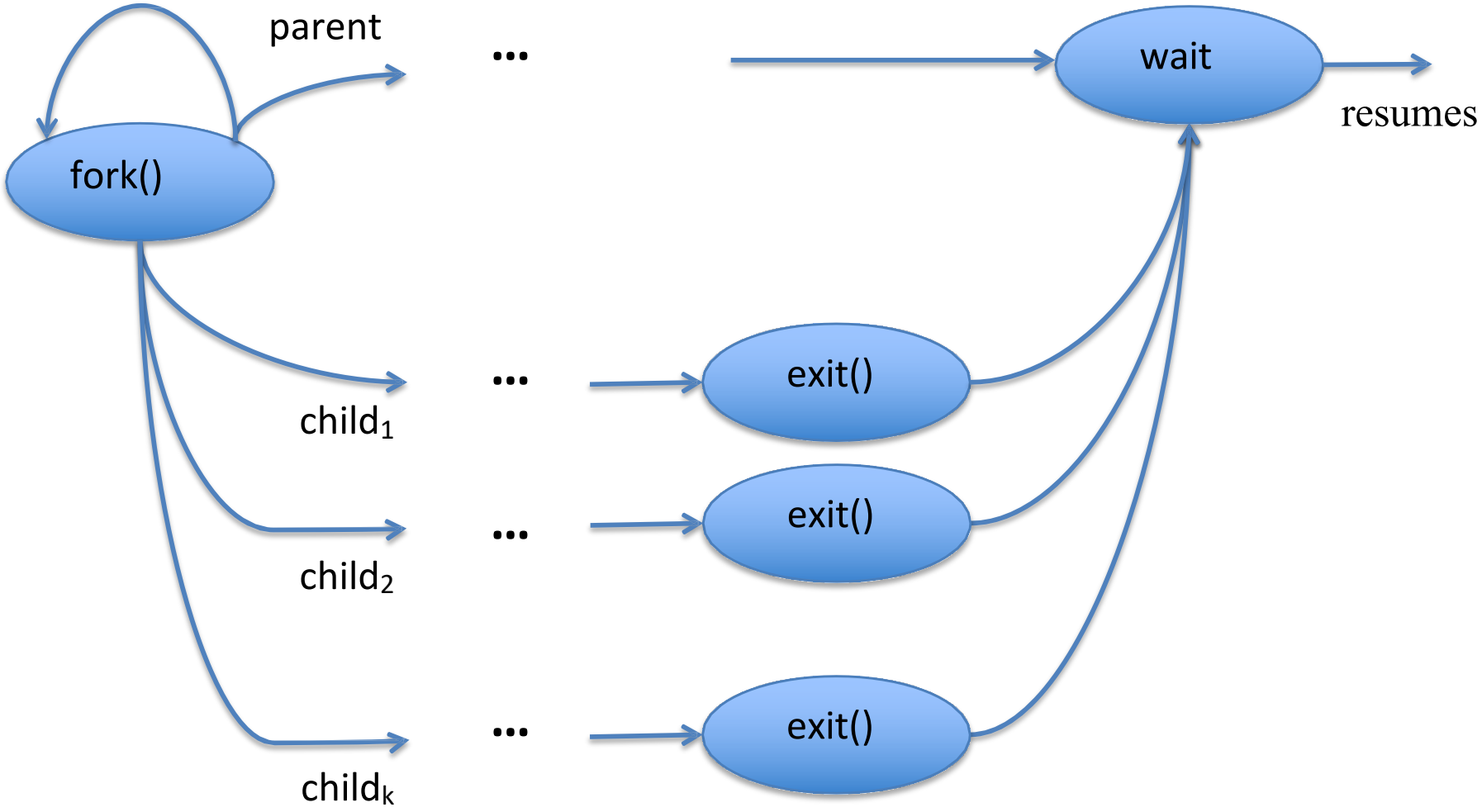
Process Wait System Calls

- Why does the parent process need to wait?

- **Sharing resource:** Parent and child may need to coordinate on using a resource (disk)
  - Must execute sequentially
- **Job managing:** Parent may act as a job manager and children are workers
  - Parent need to know when a job is finished and the status
- **Good practice:** it is always a good practice to call wait/waitpid child to timely release resources used by children

wait(2)/waitpid(2) are exclusive to a parent to wait for children
(only parent has *right* to wait)

Program with Multiple Processes



Process Wait System Calls

```
// header files omitted
int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Fork failed
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        // Child process
        printf("Child: PID = %d, doing some work...\n", getpid());
        sleep(3);  // Simulate work
        printf("Child: Done!\n");
        exit(42);  // Exit with code 42
    } else {
        // Parent process
        int status;
        printf("Parent: waiting for child PID = %d\n", pid);

        wait(&status);  // Wait for child to terminate

        if (WIFEXITED(status)) {
            printf("Parent: child exited with code %d\n", WEXITSTATUS(status));
        } else {
            printf("Parent: child did not exit normally\n");
        }
    }

    return 0;
}
```

See code example: fork-wait.c

```
// header files omitted
int main() {
    const int num_children = 3;
    pid_t pid;
    int i;

    for (i = 0; i < num_children; i++) {
        pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        } else if (pid == 0) {
            // Child process
            printf("Child %d: PID = %d, sleeping...\n", i, getpid());
            sleep(2 + i);  // Different sleep times
            printf("Child %d: Done!\n", i);
            exit(10 + i);  // Each child exits with a different code
        }
    }

    // Parent process
    int status;
    pid_t child_pid;
    while ((child_pid = wait(&status)) > 0) {
        if (WIFEXITED(status)) {
            printf("Parent: child PID %d exited with code %d\n",
                   child_pid, WEXITSTATUS(status));
        } else {
            printf("Parent: child PID %d did not exit normally\n", child_pid);
        }
    }

    printf("Parent: all children have terminated\n");
    return 0;
}
```

See code example:
fork-wait2.c

```c
int main() {
    pid_t pid1, pid2;
    int status;

    pid1 = fork(); // Fork first child
    if (pid1 == 0) {
        printf("Child 1 (PID %d) running...\n", getpid());
        sleep(3);
        printf("Child 1 done\n");
        exit(11);
    }

    pid2 = fork(); // Fork second child
    if (pid2 == 0) {
        printf("Child 2 (PID %d) running...\n", getpid());
        sleep(1);
        printf("Child 2 done\n");
        exit(22);
    }

     printf("Parent: waiting for Child 2 (PID %d)\n", pid2);
    waitpid(pid2, &status, 0);
    if (WIFEXITED(status)) {
        printf("Parent: Child 2 exited with code %d\n", WEXITSTATUS(status));
    }

    // Parent waits for **any remaining child**
    pid_t finished = waitpid(-1, &status, 0);
    if (WIFEXITED(status)) {
        printf("Parent: Child PID %d exited with code %d\n", finished, WEXITSTATUS(status));
    }

    printf("Parent: all children handled\n");
    return 0;
}
```

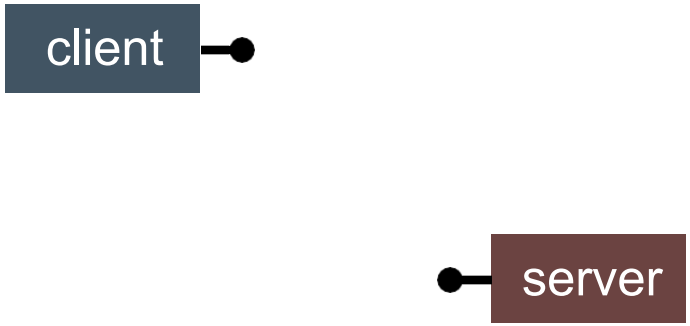
See code example: fork-waitpid.c

Process Actions in Client-Server (1)

- Example of forking to create a new process
- Consider a web server

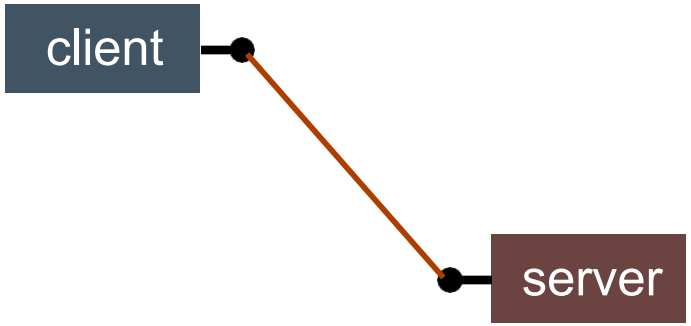
Process Actions in Client-Server (2)

- A remote "client" wants to connect and look at a webpage hosted by the web server



Process Actions in Client-Server (3)

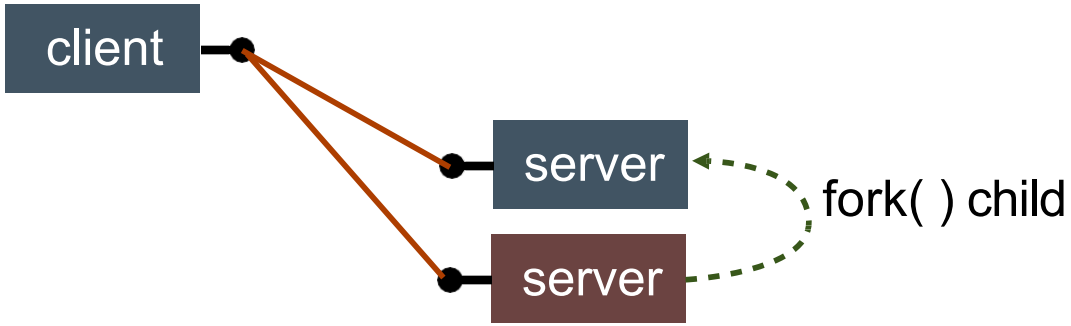
- An interprocess communication is made and a connection is established



- What if the web server only served this client until it was done looking at web pages?
- How can the web server support multiple "concurrent" clients?

Process Actions in Client-Server (4)

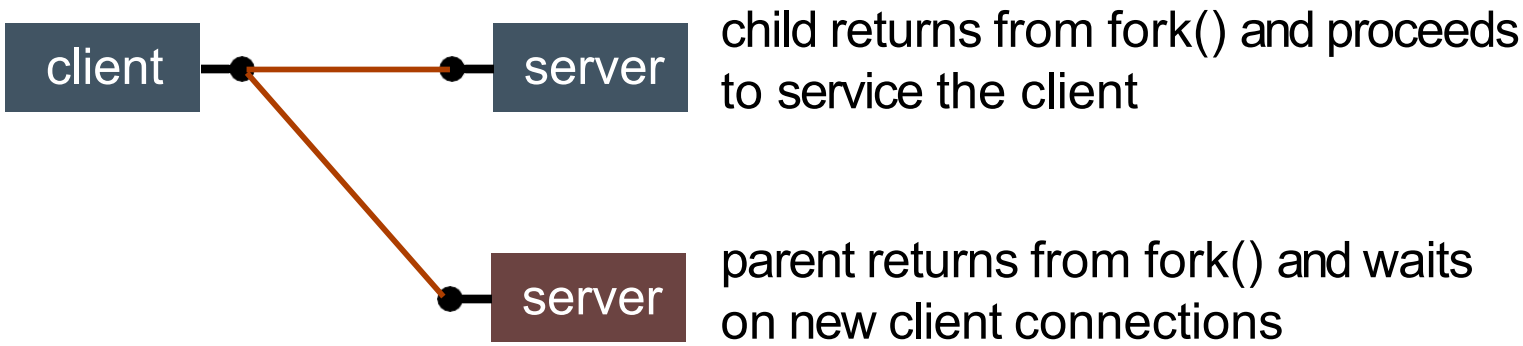
- Create more "concurrency" by creating more processes!



- A fork() copies the parent PCB and establish a child process initialized with that PCB
- All of the parent's state is inherited by the child, including the connection to the client

Process Actions in Client-Server (5)

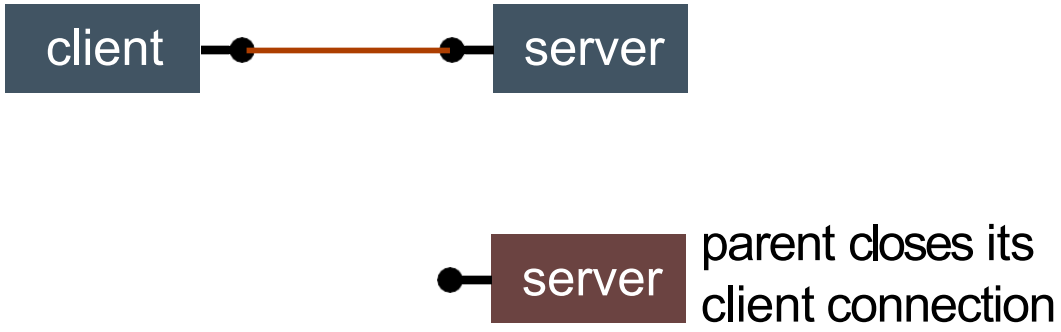
- Responsibility for "servicing" the client is handed off to the "child" server process



- The "parent" server process goes back to waiting for new clients

Process Actions in Client-Server (6)

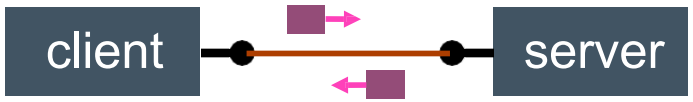
- The "parent" server should disengage from the client by releasing its (duplicate) connection



- The "child" server is now completely responsible for the client connection
- There is still a logical relationship between the child / parent server processes

Process Actions in Client-Server (7)

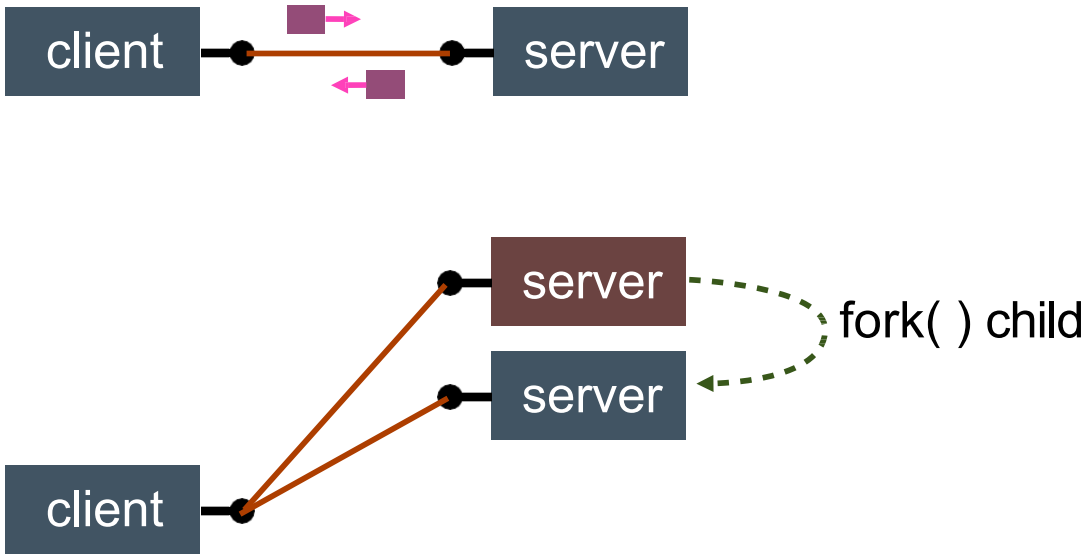
- Communication takes places between the client and "child" server process



- The "parent" server process is not involved

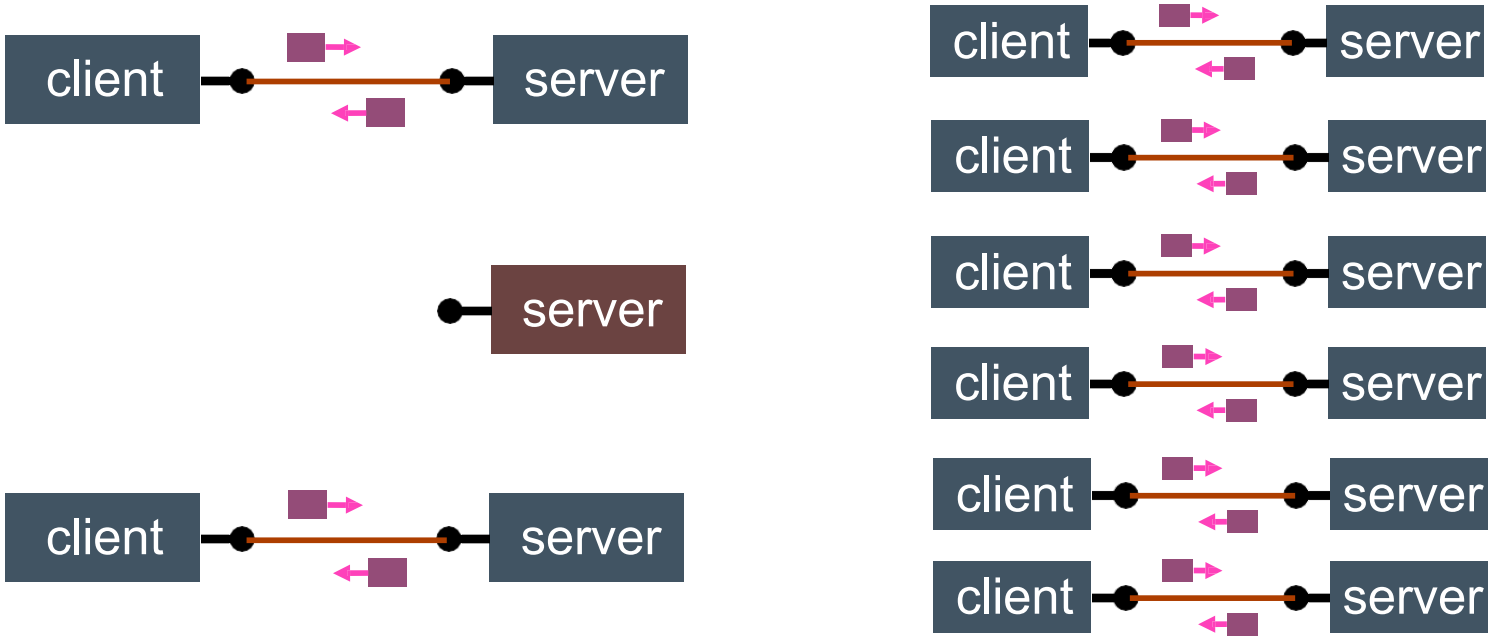
Process Actions in Client-Server (8)

- Now, this can procedure can continue as new client connections come in



Process Actions in Client-Server (9)

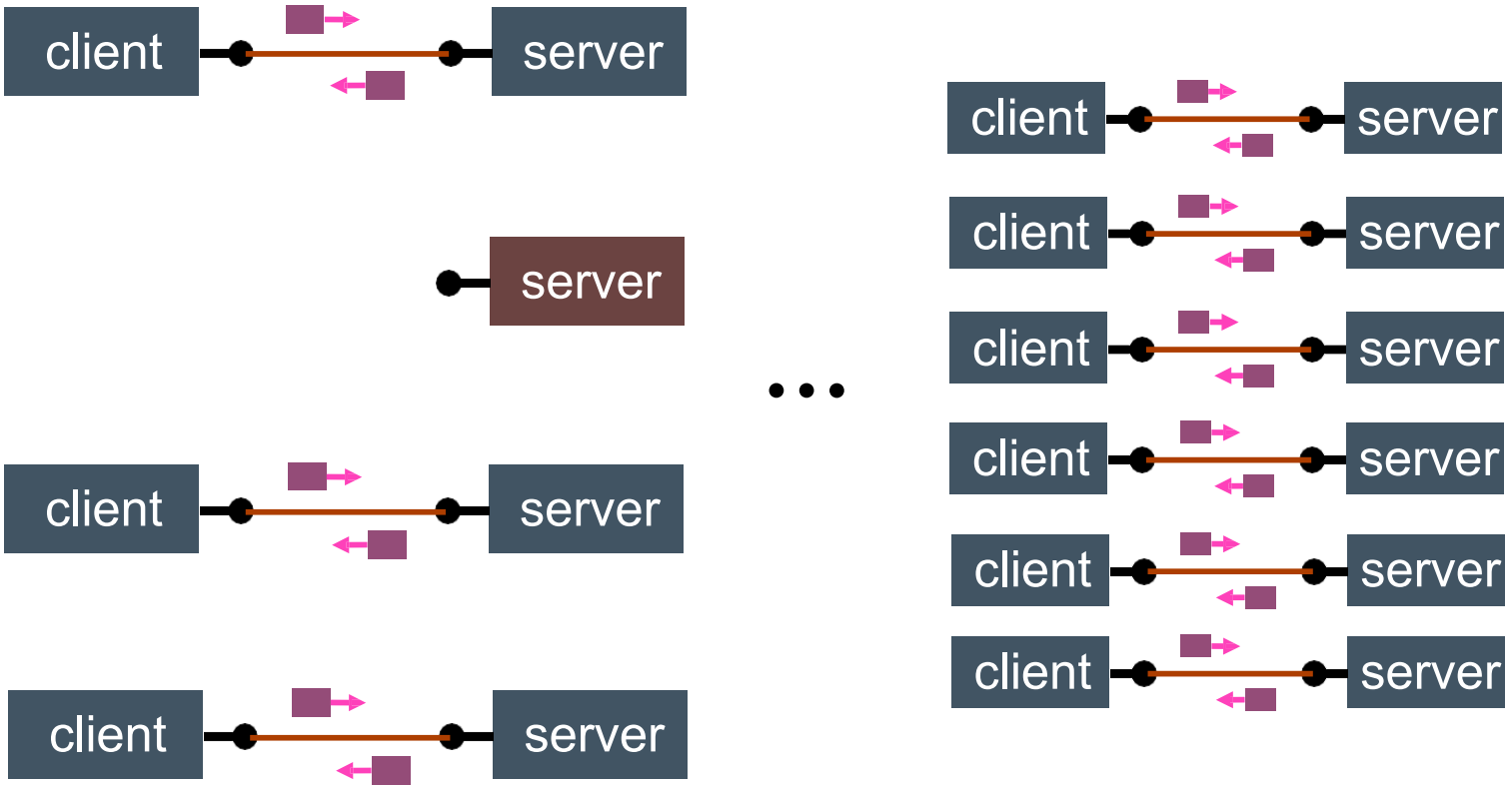
- In this way, the amount of "concurrency" increases with more server processes



- It gives both logical isolation between the servers, as well as simplifies the design and functionality

Process Actions in Client-Server (10)

- Many types of servers operate this way
- A objective is to avoid blocking in the server
  - If one server is stalled, others can run



Process Wait System Calls

- How do wait (by parent) and exit (by child) work with each other?
- When a child process terminates, it enters the "zombie" state:
  - The child's exit code and status are stored in the kernel.
  - The process resources (like its PID entry) are not fully released yet.
  - The kernel keeps them until the parent calls wait(2) (or waitpid(2)).
- wait(2) lets the parent:
  - Synchronize with the child's termination, and
  - Reap (collect) the child's exit status — removing the zombie entry.

What if a parent never waits?

Process Wait System Calls

- What if a parent never waits?
  - The child becomes a zombie after it finishes.
  - When the parent terminates, the child is adopted by systemd (PID 1) → called reparenting
  - The systemd process automatically calls wait() to reap it.
  - This prevents zombies from accumulating forever.

Note 1: calling wait won't force a child to exit. It only reclaim resources if a child already exited

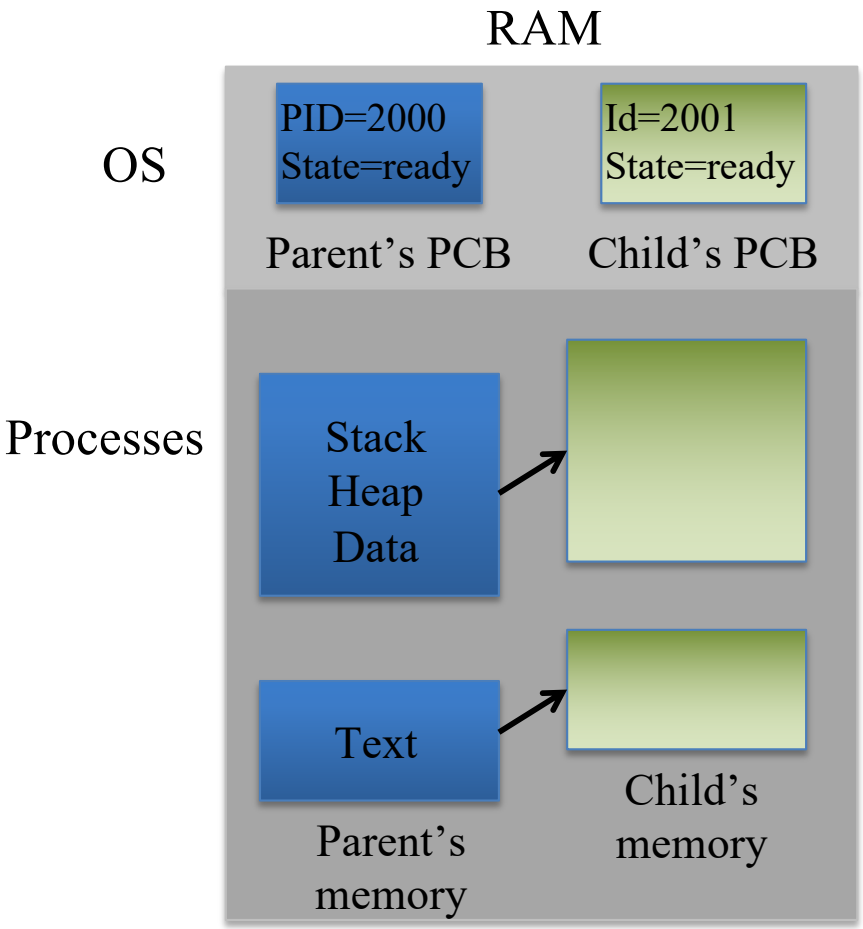
Note 2: A child can continue to run after parent exits early systemd won't force terminating it

Exec System Call Family

- How to let a child run a completely new program?
- *exec(3)*
  - replaces the current process image (code) with a new program.
  - The PID stays the same
  - The old code, data, and stack are gone
  - The new program starts executing from its main()
  - The process does not return from exec() unless it fails

- Many variant: exec(3), execl(3), execlp(3), execle(3), execv(3), execvp(3), execvpe(3)
- All built on top of execve(2)

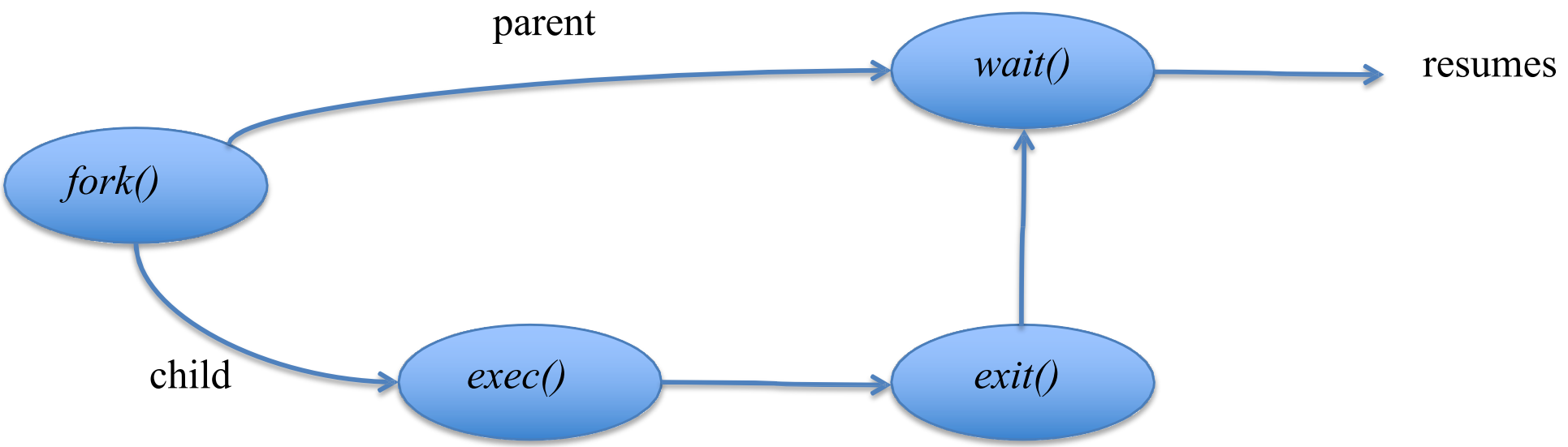
Forking a Child Process with exec(3)



Effects in memory after parent calls *fork()*

1. PCB with new PID created
2. Memory allocated for child initialized by copying over from the parent
3. If parent had called *wait()*, it is moved to a waiting queue
4. If child had called *exec()*, its memory is overwritten with new code and data
5. Child added to ready queue and is all set to go now!

Process Creation with New Program



- Parent process calls *fork()* to spawn child process
  - Both parent and child return from fork()
  - Continue to execute the same program
- Child process calls *exec()* to load a new program

Process Creation with New Program

```
// headers omitted..
int main() {
    pid_t pid;

    // Create a new process
    pid = fork();

    if (pid < 0) {
        // Error forking
        perror("fork failed");
        exit(1);
    } else if (pid == 0) {
        // Child process
        printf("Child process (PID %d) executing 'ls'\n", getpid());

        // Replace child process image with "ls -l"
        execlp("ls", "ls", "-l", NULL);

        // If exec returns, there was an error
        perror("execlp failed");
        exit(1);
    } else {
        // Parent process
        printf("Parent process (PID %d) waiting for child to finish...\n", getpid());
        wait(NULL);  // Wait for child to complete
        printf("Child finished. Parent exiting.\n");
    }

    return 0;
}
```

See code example: fork-exec.c

Exec System Calls

- If we can create separate processes to do different tasks, why do we need to use exec?
  - Separation of creation/execution (shells, servers)
  - Parent continues (monitors)
  - Resource management (Keeps FD, env, PID)
  - Security (Setup restricted context before exec)

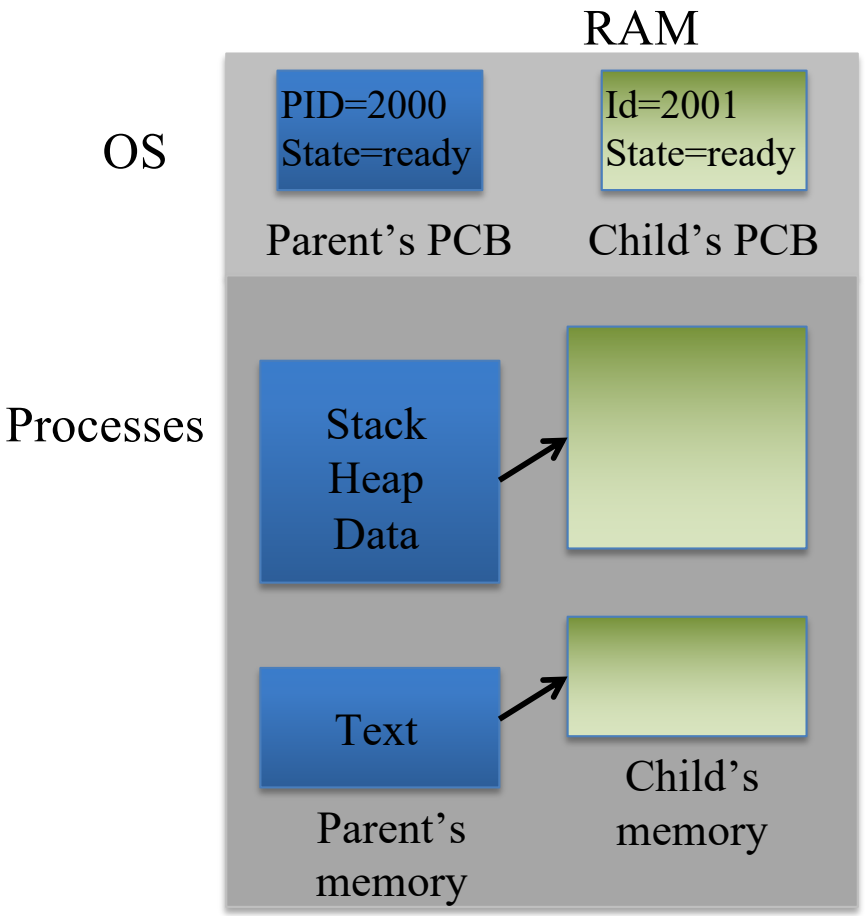
Exec System Calls

```
int main(void) {
    int fd = open("output.txt", O_WRONLY | O_CREAT |
O_TRUNC, 0644);

    pid_t pid = fork();

    if (pid == 0) {

        char fd_str[10];
        snprintf(fd_str, sizeof(fd_str), "%d", fd);

        execl("./writer", "writer", fd_str, NULL);
        perror("execl");
        exit(1);
    } else {
        const char *msg = "Message from parent\n";
        write(fd, msg, strlen(msg));

       wait(NULL);
        printf("Parent: done.\n");
    }

    close(fd);
}
```

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <fd>\n", argv[0]);
        exit(1);
    }

    int fd = atoi(argv[1]);
    const char *msg = "Message from exec'ed child\n";

    printf("Writer: inherited fd = %d, writing to it...\n", fd);
    write(fd, msg, strlen(msg));

    return 0;
}
```

See code example:
parent.c
writer.c

Forking a Child Process



Effects in memory after parent calls *fork()*

The kernel creates a **new PCB** for the child process.
This includes:

- A **new process ID (PID)**
- **Parent PID** (link to the parent)
- **Copy of CPU registers**, program counter (PC), stack pointer, etc.
- **Copy of memory mappings** (code, data, heap, stack segments)
- **Copied file descriptors** (pointing to same open files)
- **Scheduling info**, signal handlers, etc.

If a child is meant to load a new program with exec, why copy from the parent?

Process Creation System Calls

- How are processes brought into existence?
- *fork(2)*
  - Copy the address space of parent
- *vfork(2)*
  - Use the parent's address space
  - Share address space between parent and child
    - includes data segment (and mapped physical memory)
  - Calling parent blocks until child terminates or calls *exec()*
  - Faster than *fork()*

Process Creation System Calls

```
int main() {
    printf("Parent: PID = %d\n", getpid());

    pid_t pid = vfork();
    if (pid < 0) {
        perror("vfork");
        exit(1);
    } else if (pid == 0) {
        // Child process
        printf("Child: PID = %d, replacing itself with ls\n", getpid());
        execlp("ls", "ls", "-l", NULL);  // Safe: immediately exec
        _exit(1);  // Only reached if exec fails
    } else {
        // Parent resumes after child execs or exits
        wait(NULL);
        printf("Parent: child finished\n");
    }
    return 0;
}
```

See code example: vfork-exec.c

Process Creation System Calls

- Be careful with vfork(2)
  - the child temporarily shares the exact same address space, stack, and memory as the parent — no copying happens.
  - Parent is suspended until child exit or call exec(3)

What a child CAN do?

Safe action	Why it's safe
Call exec()	Immediately replaces memory — the correct and intended use.
Call _exit()	Ends the child quickly without disturbing shared memory.
Use local variables **read-only**	Reading doesn't change shared state.
Call simple system calls (like write() to FD 1)	Does not modify process memory.
Assign to local variables (if not used by parent later)	Technically allowed, but not recommended.

Process Creation System Calls

- Be careful with vfork(2)
  - the child temporarily shares the exact same address space, stack, and memory as the parent — no copying happens.
  - Parent is suspended until child exit or call exec(3)

See code example: vfork-wrong.c

What a child CANNOT do?

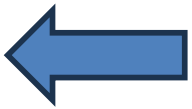
Forbidden action	Why it's unsafe
Return from the function that called vfork()	Returns to the parent's stack frame — corrupts parent's stack.
Modify any variable in parent's address space	Both share the same memory, so changes affect the parent.
Call exit() instead of _exit()	exit() flushes stdio buffers and runs atexit handlers — corrupts shared memory.
Allocate/free memory (e.g., malloc/free)	Modifies shared heap structures.
Call functions that modify globals (like printf(), setenv(), etc.)	May change shared libc state or buffered I/O.
Use standard I/O before exec	Because stdout and buffers are shared and can be corrupted.

Process Creation System Calls

- exit(3) vs. _exit(2)

exit(3): **high-level shutdown of C runtime**
1. Flushes all stdio buffers
2. Writes out buffered output from printf(), etc.
3. Closes open streams (fclose() on all FILE*)
4. Runs functions registered with atexit()
5. Deletes temporary files created by tmpfile()
6. Calls _exit(status) to actually end the process

_exit(2): low-level, direct way to end a process.
1. Immediately terminates the process.
2. Does not flush or close anything.
3. No stdio cleanup, no atexit(), no I/O flushing.



Safe for vfork

- Only need to worry about this if the child never calls exec.
- If child calls exec, it can exit normally with exit(3)

Process Creation Options (Parent and Child)

- Process hierarchy options
  - Parent process creates children processes
  - Child processes can create other child processes
  - Tree of processes
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

Process Scheduling

- Only one process can be running on any processor core at any instant
- We are highly likely to run more processes than the total number of cores we have in a system
  - Degree of multiprogramming = number of processes currently in memory
  - Degree of multiprogramming > # of cores
- Scheduling objective
  - Maximize CPU utilization: have some process running at all times
  - High intractability: switch a CPU core among processes so frequently that users can interact with each program while it is running.

Process State Management

- What do we need to track about a process?
  - What's the state of each of them?
- Process control block
  - Kernel data structure for tracking process context
- Process table
  - Kernel data structure tracking processes on system
- These take up real space in kernel memory
  - There is a limit on # processes
  - Typically Linux sets to 32,768 (2^{15}) processes
  - Actual limit (`cat /proc/sys/kernel/pid_max`)

/proc File System

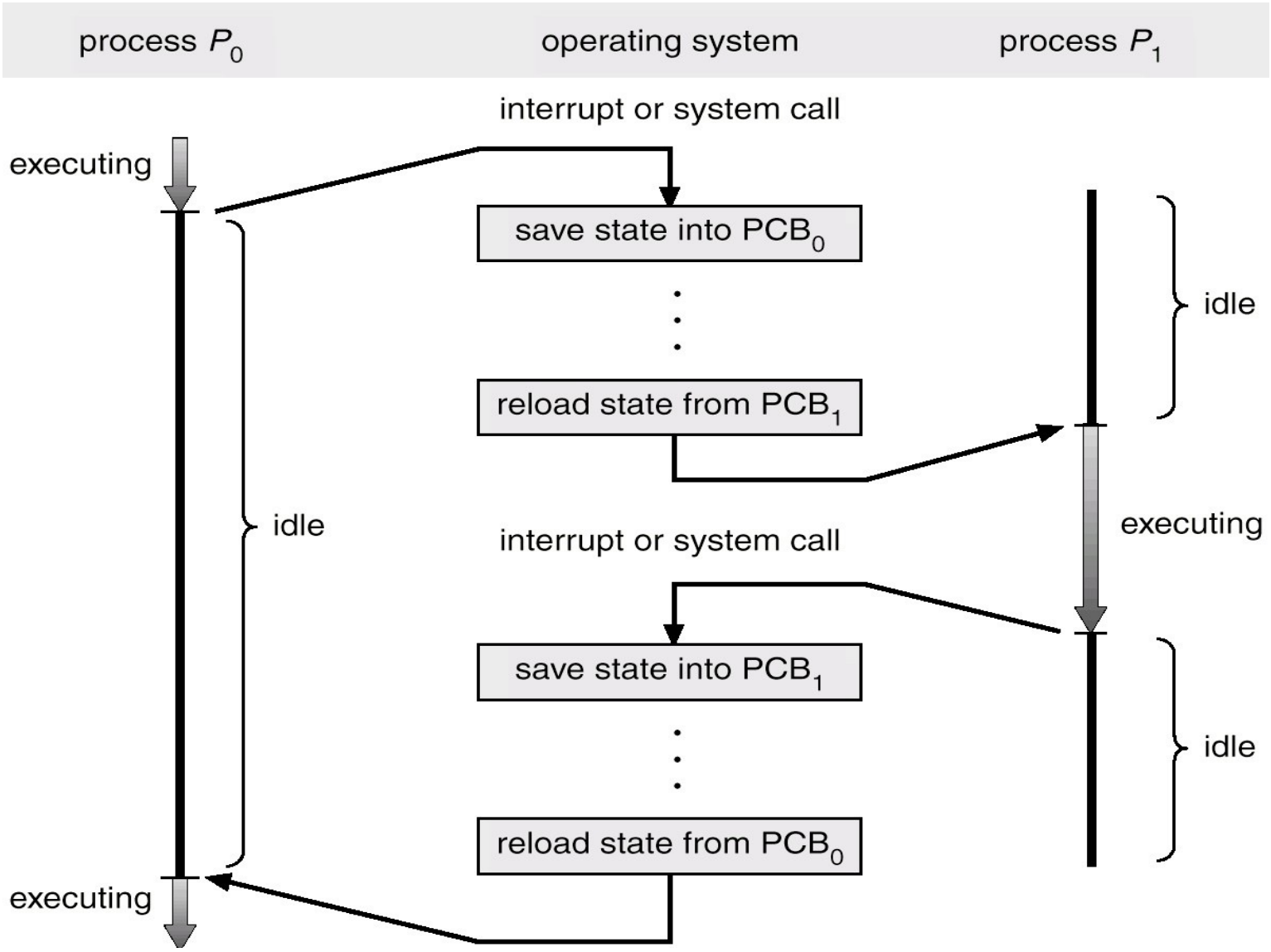
- Linux
  - `ls /proc`
  - Process information pseudo-file system
  - Does not contain "real" files, but runtime system information
    - System memory
    - Devices mounted
    - Hardware configuration
  - A directory for each process
  - Reading the files in /proc is equivalent to querying the kernel in real-time
- Various process information
  - `/proc/<pid>/io`
    I/O statistics
  - `/proc/<pid>/environ`
    Environment variables (in binary)
  - `/proc/<pid>/stat` and `/proc/<pid>/status`
    Process status and info
- Check out "man proc"

See program example

Context Switch

- OS switches from one execution context to another
  - One process to another process (user or system processes)
  - Involves going through the kernel
  - Generally occurs as a result of an interrupt
  - Process could also yield or exit
- Current process to new process
  - Save the state of the current process
    - process control block: describes the state of the process in the CPU
  - Load the saved context for the new process
    - load the new process's process control block into OS and registers
  - Start the new process

Context Switch



Context Switch Performance

- No useful work is being done during a context switch (i.e., no user process is running)
  - Want to speed up context switch processing
  - If a system call can be done in user mode, then the OS does not have to context switch to do certain things
- Hardware support helps in context switching
  - Multiple hardware register sets to capture CPU state
  - Must be able to quickly set up the processor
- More complex process context can be challenging
  - Managing address translation tables is difficult

Process Description Summary

- Serves two purposes
  - Track per process resources
  - Save CPU state on context switch
- Process control block
  - Represents both aspects
  - CPU state
    - progam counter, registers
  - Resources
    - links to other processes
    - memory management
    - open files
    - …

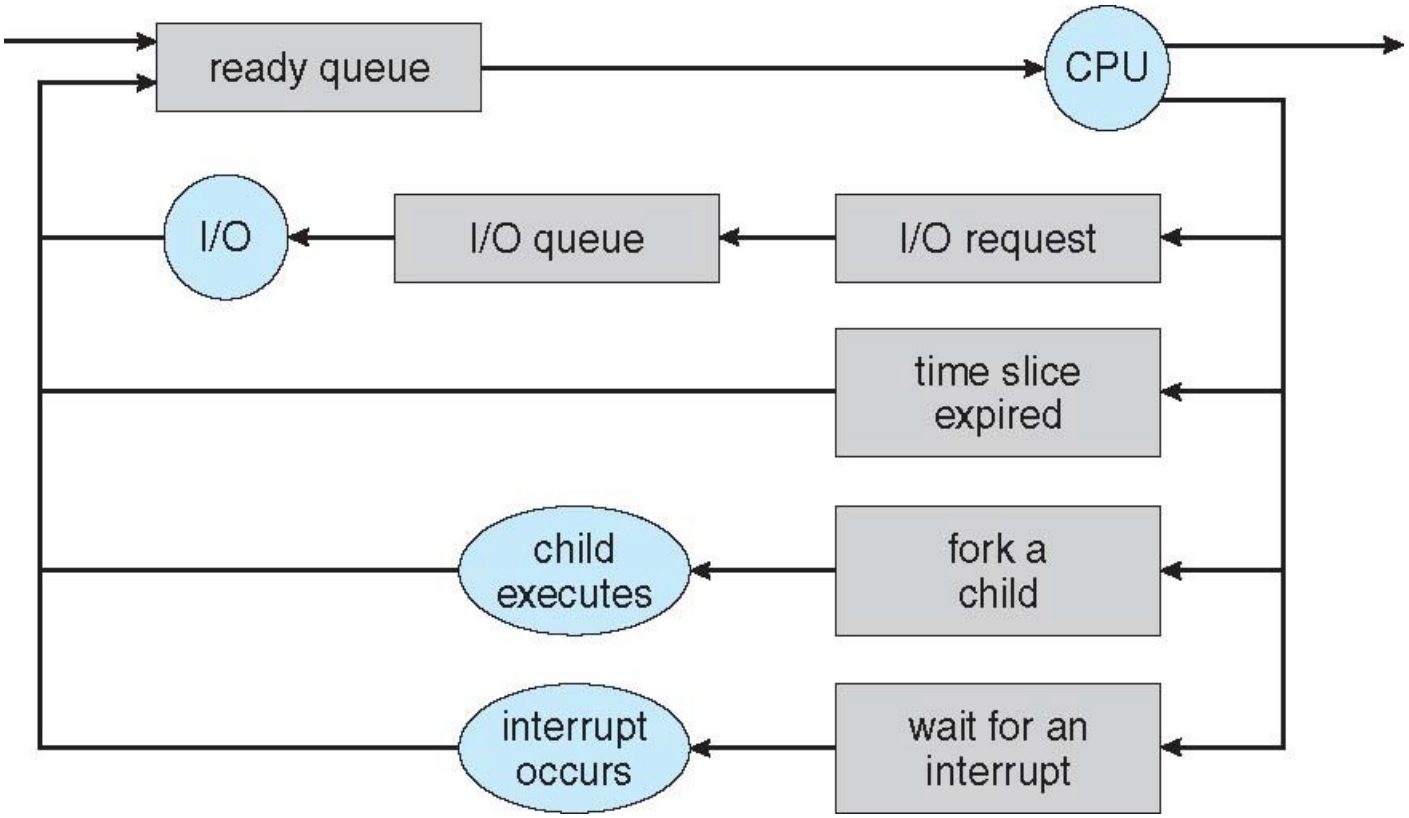
Do memory segments (data, stack, heap, text) need to be stored in PCB?

Process Scheduling

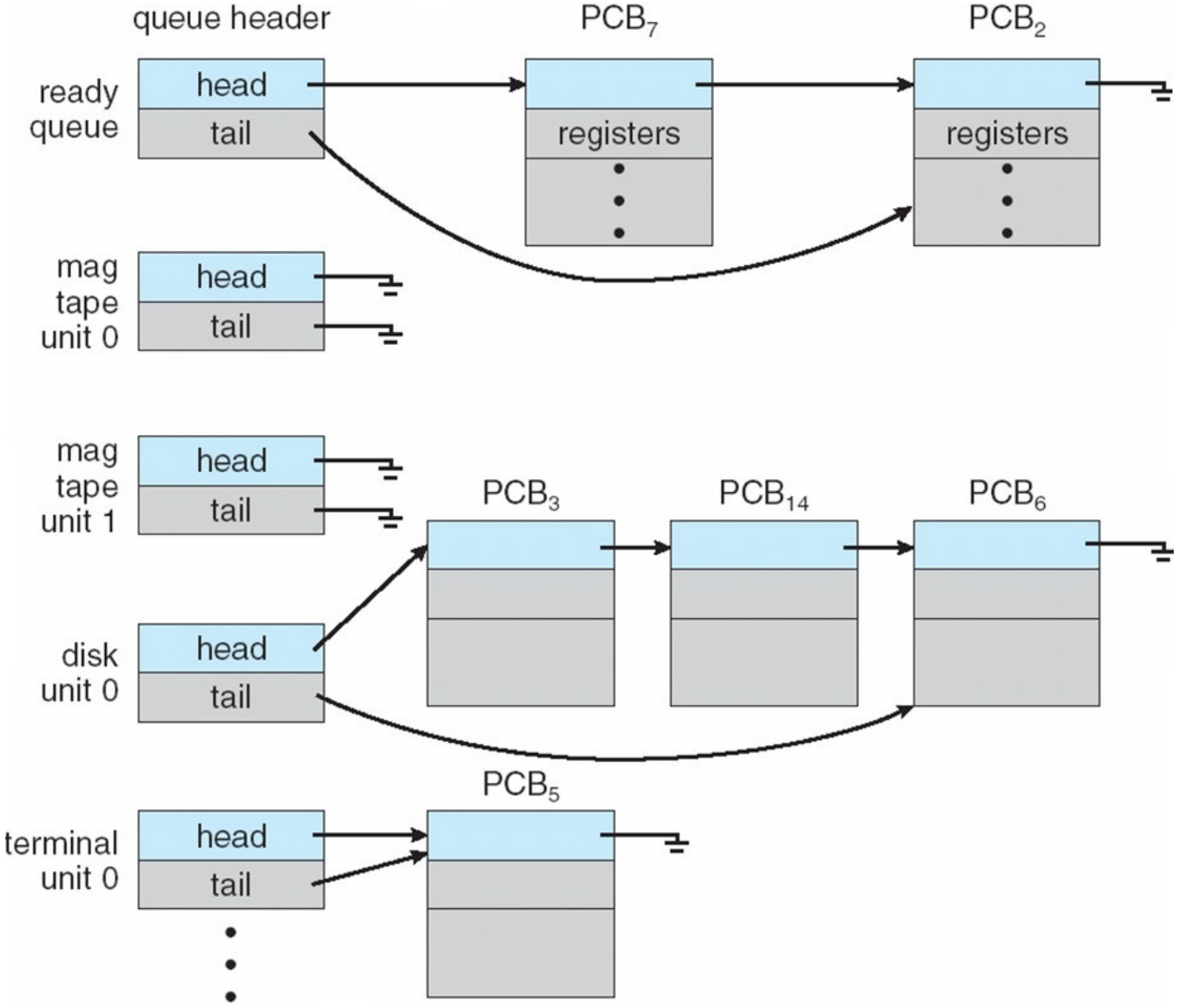
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - *Job queue* – set of all processes in the system
  - *Ready queue* – set of all processes residing in main memory, ready and waiting to execute
  - *Device queues* – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Representation of Process Scheduling

- Process scheduling queueing diagram represents:
  - queues, resources, flows
- Processes move through the queues

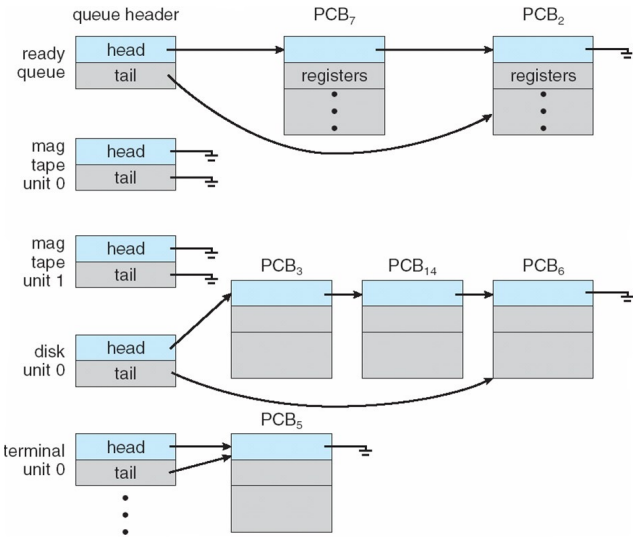


Ready Queue And Various I/O Device Queues



Process Scheduling

- New processes are put into a ready queue
  - Queue is generally stored as a linked list
  - Ready-queue header contains pointers to the first PCB in the list
  - Each PCB includes a pointer !eld that points to the next PCB
- A process executes for a while and eventually
  - terminates
  - Interrupted (e.g., time slice expired)
  - Waits for an event (e.g., I/O request, termination of child process)
- CPU/OS selects another process to execute

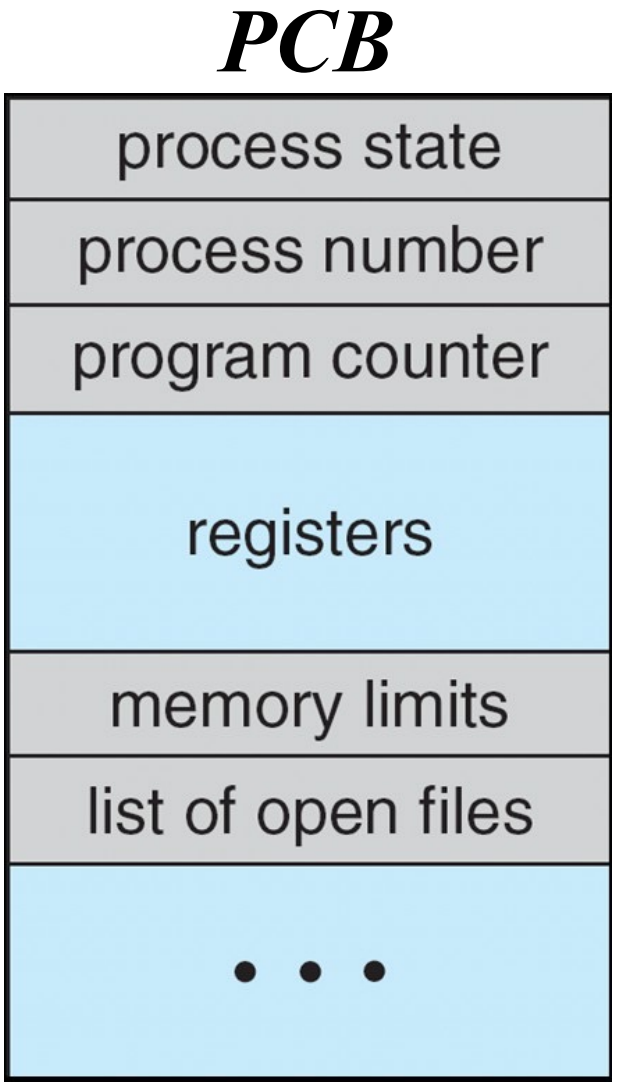


Process Model

- Much of the OS's job is keeping processes from interfering with each other … Why?
- Each process has its OWN resources to use
  - Program code to execute, address space, files, …
- Processes are important for *isolation* (*protection*)
  - Prevent one process from affecting another process
- Processes are *heavyweight* … Why do you think?
  - Pay a price for isolation
  - There is lots of process state to save and restore
  - A full "process swap" is required for multiprocessing
  - OS must context switch between them
    - intervene to save/restore all process state

Process Control Block

- Information associated with each process (aka *task control block)*
- Process state
  - Running, waiting, …
- Program counter
  - Location of instruction to next execute
- CPU registers
  - Contents of all process-centric registers
- CPU scheduling information
  - Priorities, scheduling queue pointers, …
- Memory-management information
  - Memory allocated to the process
  - Memory limits and other associated information
- Accounting information
  - CPU used, elapsed time, time limits, …
- I/O status information
  - I/O devices allocated to process, list of open files



Next Class

- Process scheduling
- Interprocess communication (IPC)