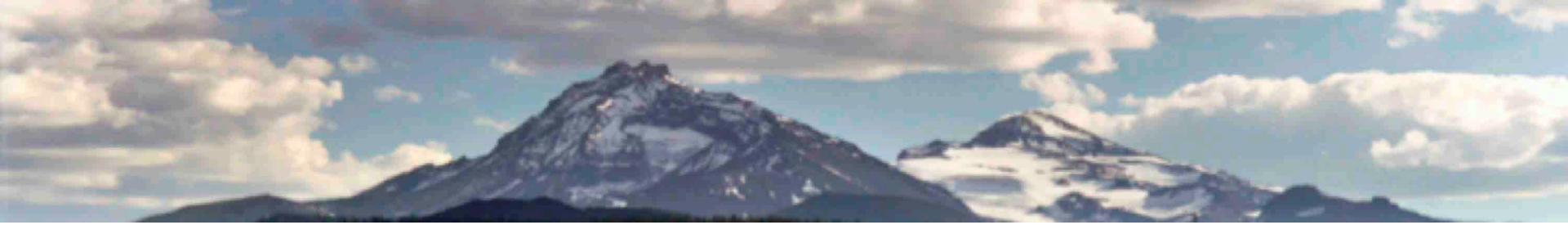


CIS 415
Operating Systems
Processes

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2020

UNIVERSITY OF OREGON

Logistics

- Project 1 posted
  - Due Sunday, April 19, 11:59pm
- Labs
  - Intended to support projects
  - Lab 2 to be posted shortly
- Read OSC Chapter 3

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

Superview and User Modes

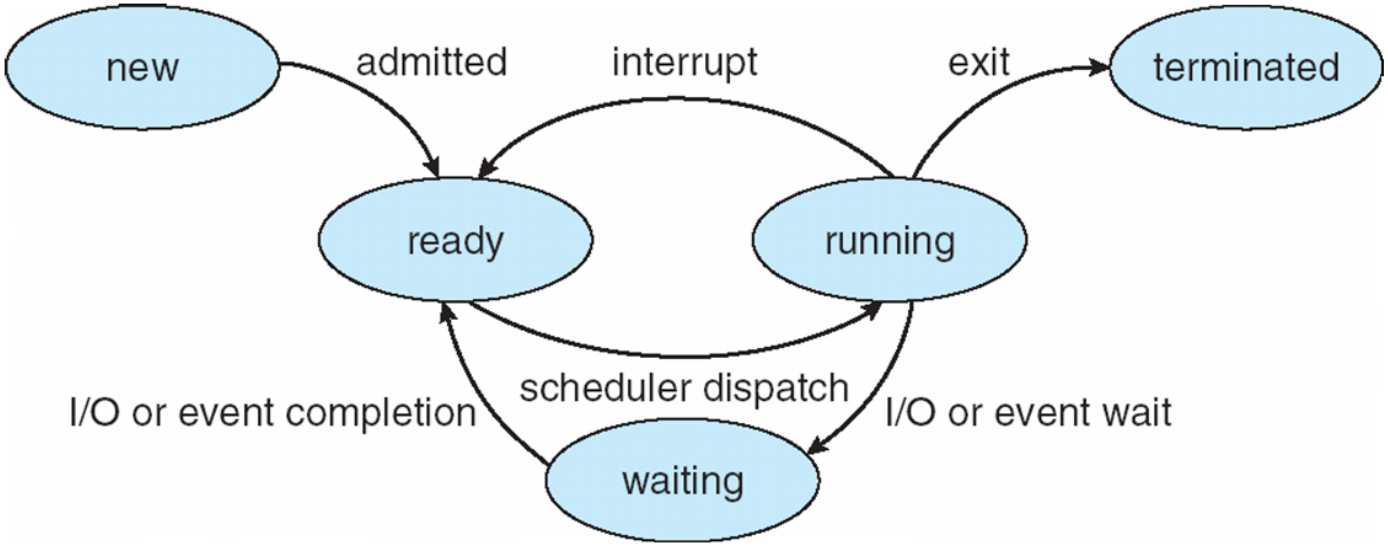
- OS runs in "supervisor" mode
  - Has access to protected (privileged) instructions only available in that mode (kernel executes in ring 0)
  - Allows it to manage the entire system
- OS "loads" programs into processes
  - Run in user mode
  - Many processes can run in user mode at the same time
- How does OS get programs loaded into processes in user mode and keep them straight?

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Terms *job* and *process* used almost interchangeably
- A process is a ***program in execution***
  - Process execution can result in more processes being created
- Multiple parts of a process
  - Program code (image, text)
  - Execution state (program counter, processor registers, …)
  - Stack containing temporary data (e.g., call stack frames)
    - function parameters, return addresses, local variables, …
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

Process Execution State

- As a process executes, it changes state
  - *New*: The process is being created (starting state)
  - *Running*: Instructions are being executed
  - *Waiting*: The process is waiting for some event to occur
  - *Ready*: The process is waiting to run
  - *Terminated*: The process has finished execution (end state)

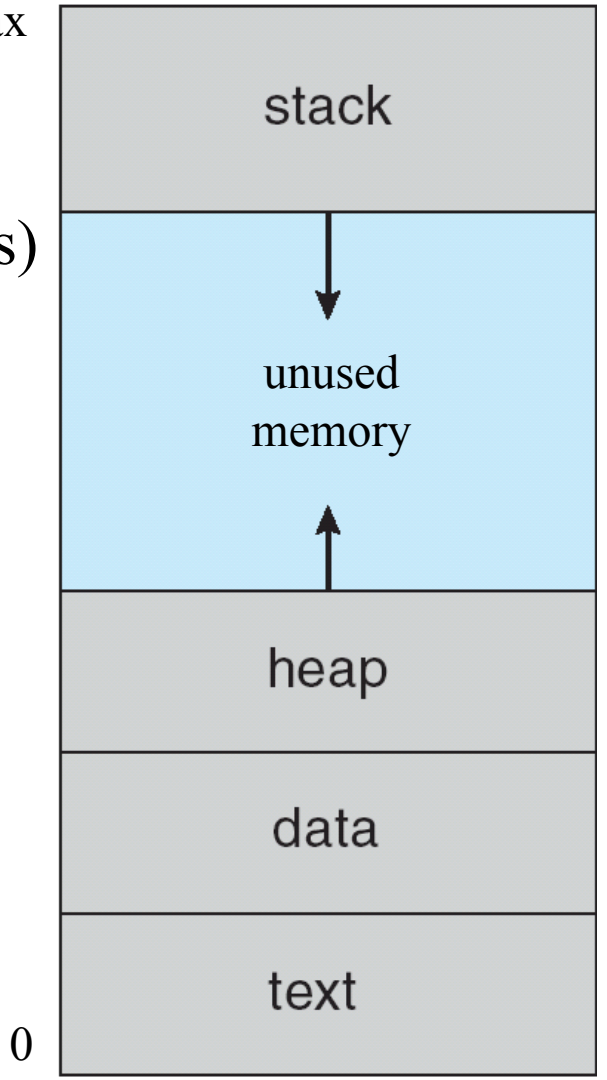


Process State

- Process state consists of:
  - Address space (what can be addressed by a process)
  - Execution state (what is need to execute on the CPU)
  - Resources being used (by the process to execute)
- Address space contains code and data of a process
- Processes are individual *execution contexts*
  - Threads are also include here (we will talk about later)
- Resources are physical support necessary to execute
  - Memory: physical memory, address translation support
  - Storage: disk, files, …
  - Processor: CPU (at least 1)

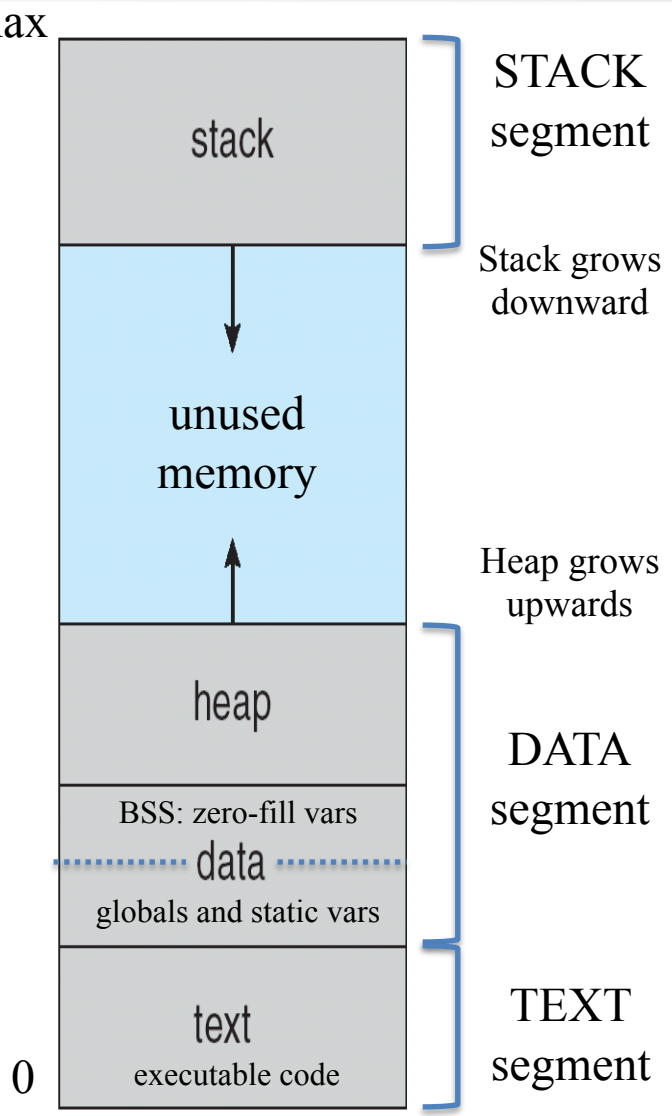
A Process in Memory

- A process has to reference memory for different purposes
  - Instructions
  - Stack (subroutine "frames", local variables)
  - Data: static and dynamic (heap)
- Shows where are these things located in "logical" memory for a process?
- Logical memory
  - What can be referenced by an address
  - # address bits in instruction adressess determine logical memory size
  - A "logical address" is from 0 to the size of logical memory (max)
- Compiler and OS determine where things get placed in logical memory



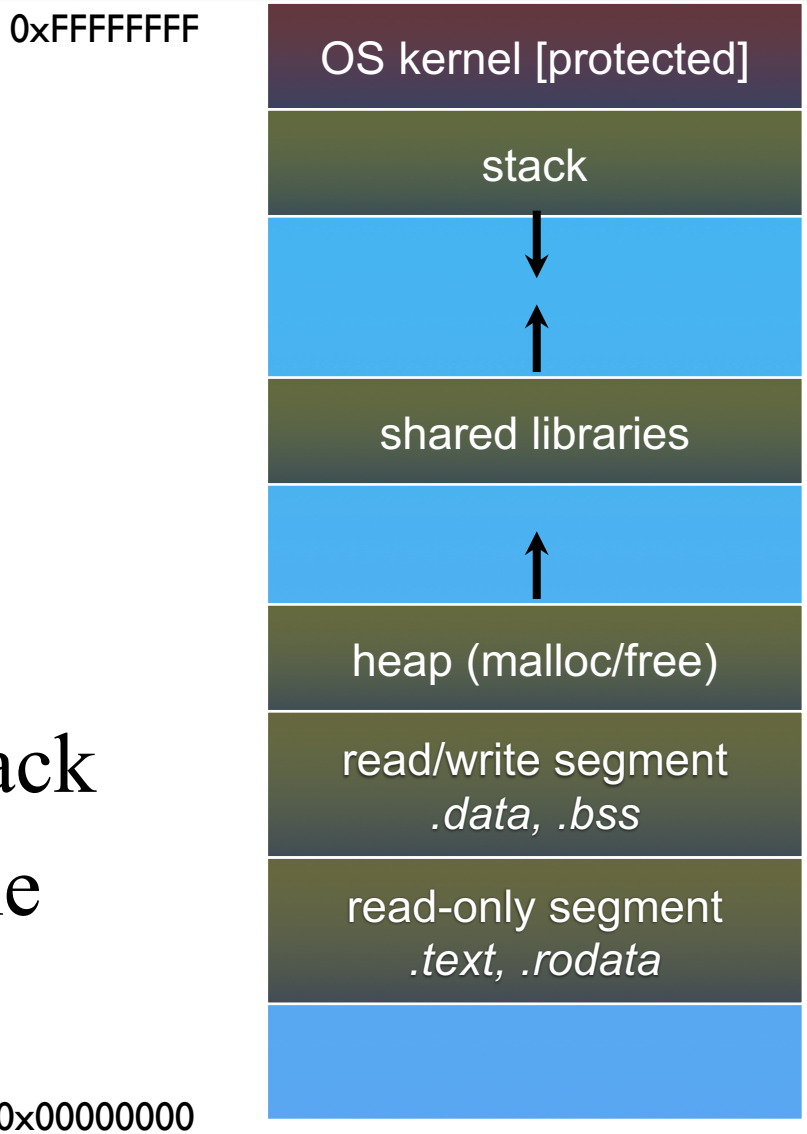
Process Address Space

- All locations addressable by process
  - Also called *logical address space*
  - Every process has one
- Restrict addresses to different areas
  - Restrictions enforced by OS
  - **Text segment** is where read only program instructions are stored
  - **Data segment** hold the data for the running process (read/write)
    - heap allows for dynamic data expansion
  - **Stack segment** is where the stack lives
- Process (logical) address space starts at 0 and runs to a high address



Process Address Space

- Program (Text)
- Global Data (Data)
- Dynamic Data (Heap)
  - Grows up
- Thread-local Data (Stack)
  - Grows down
- Each thread has its own stack
- # address bits determine the addressing range



Process Address Space in Action

```
int value = 5;                                  Global

int main()
{
  int *p;                                       Stack

  p = (int *)malloc(sizeof(int));               Heap

  if (p == 0) {
    printf("ERROR: Out of memory\n”);
  return 1;
}

*p = value;
printf("%d\n", *p);
free(p);
return 0;
}
```

Process Address Space in Action

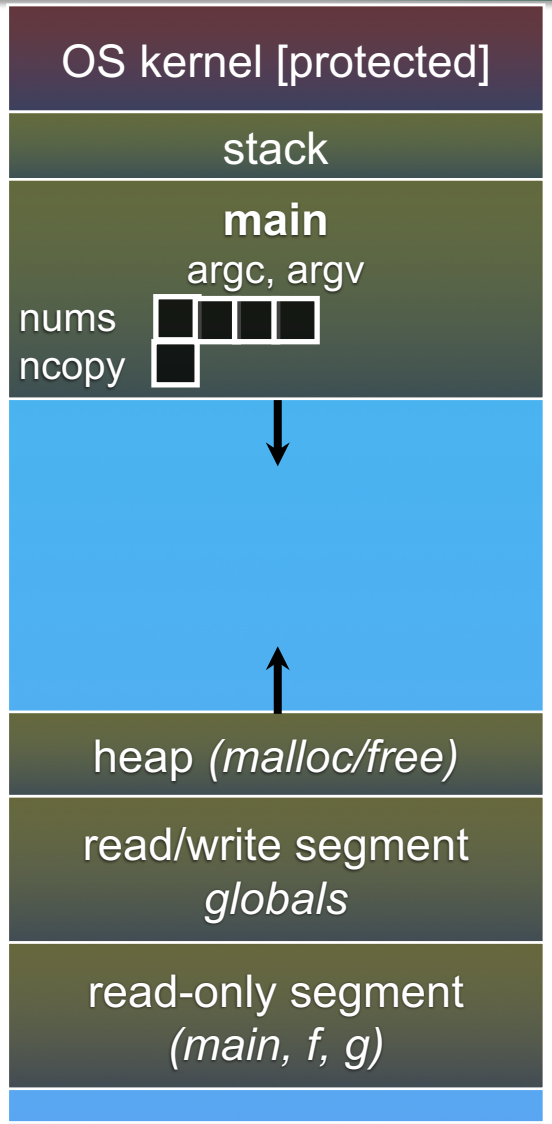
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

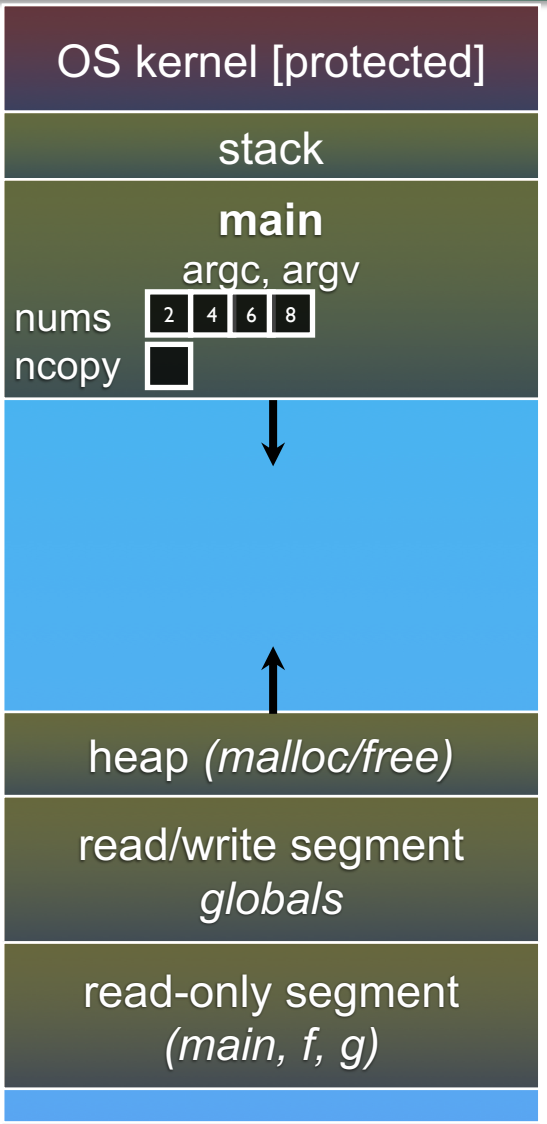
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

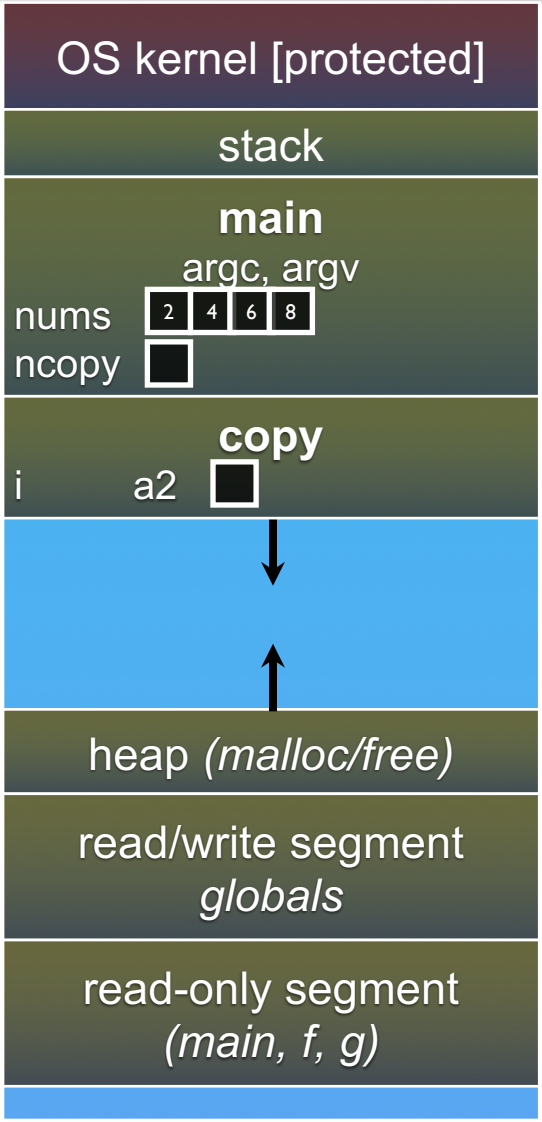
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

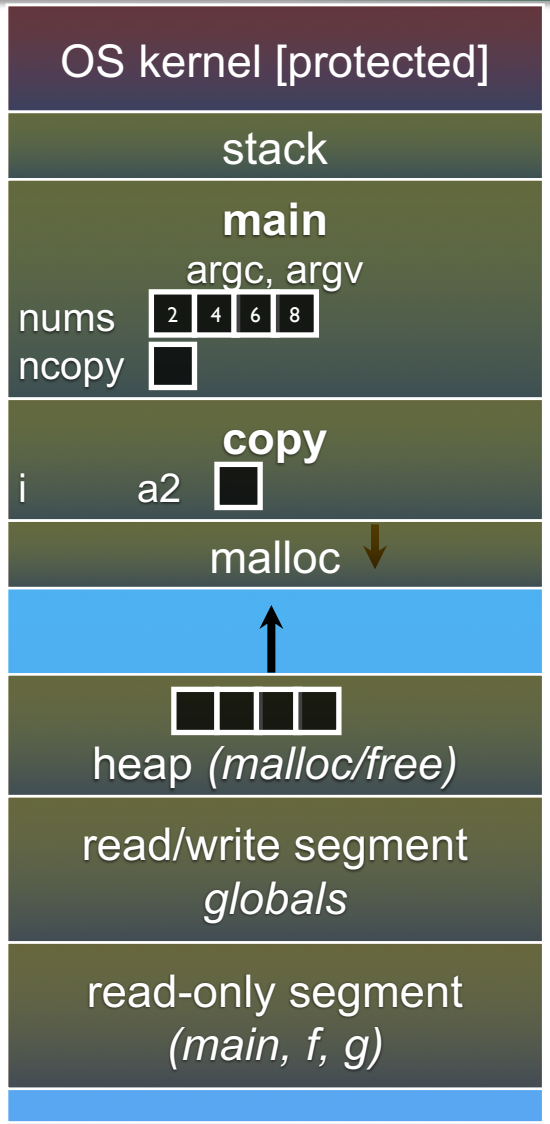
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

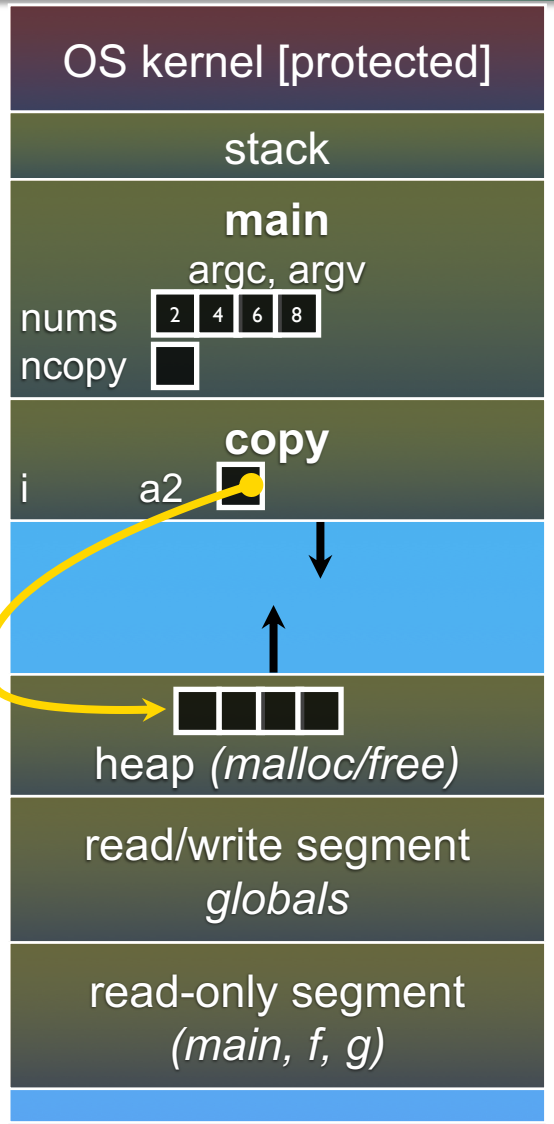
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

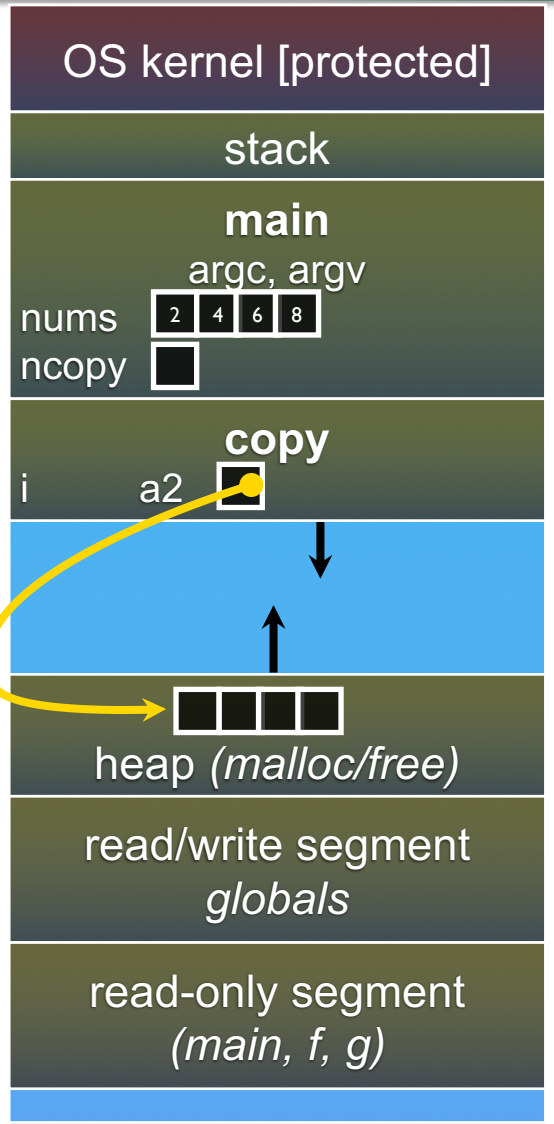
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

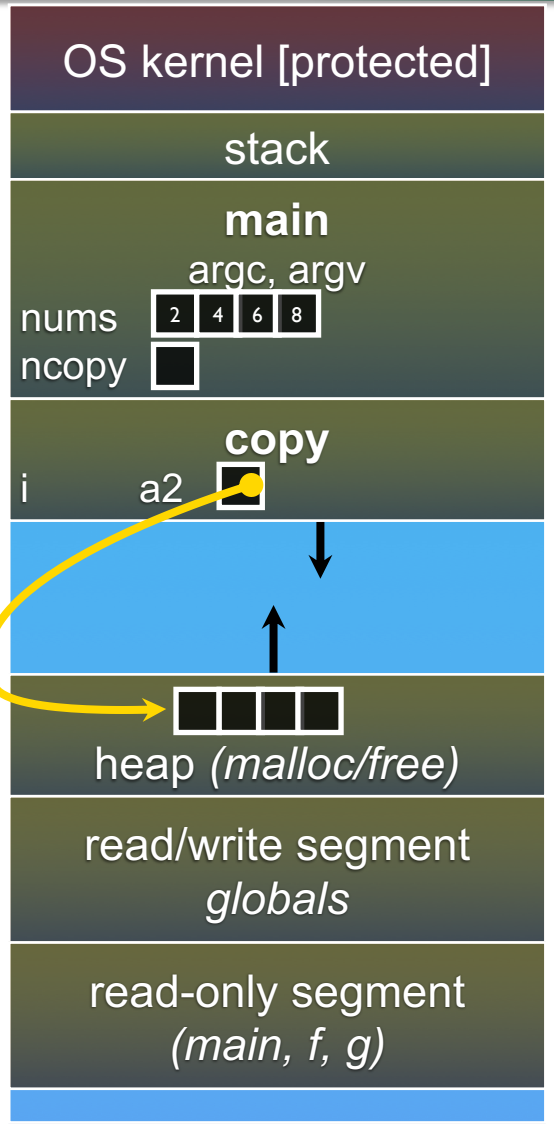
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

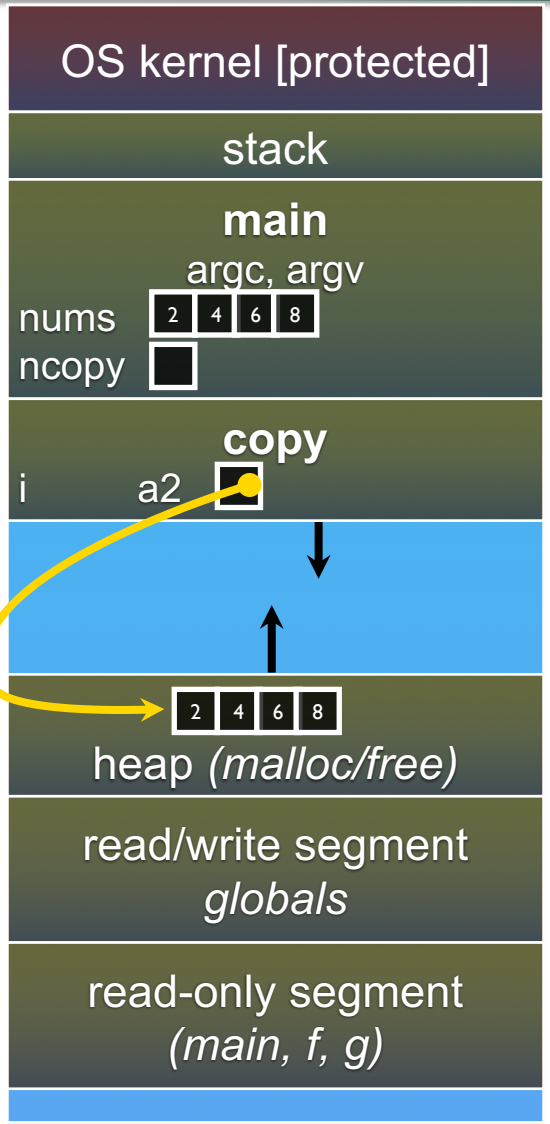
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

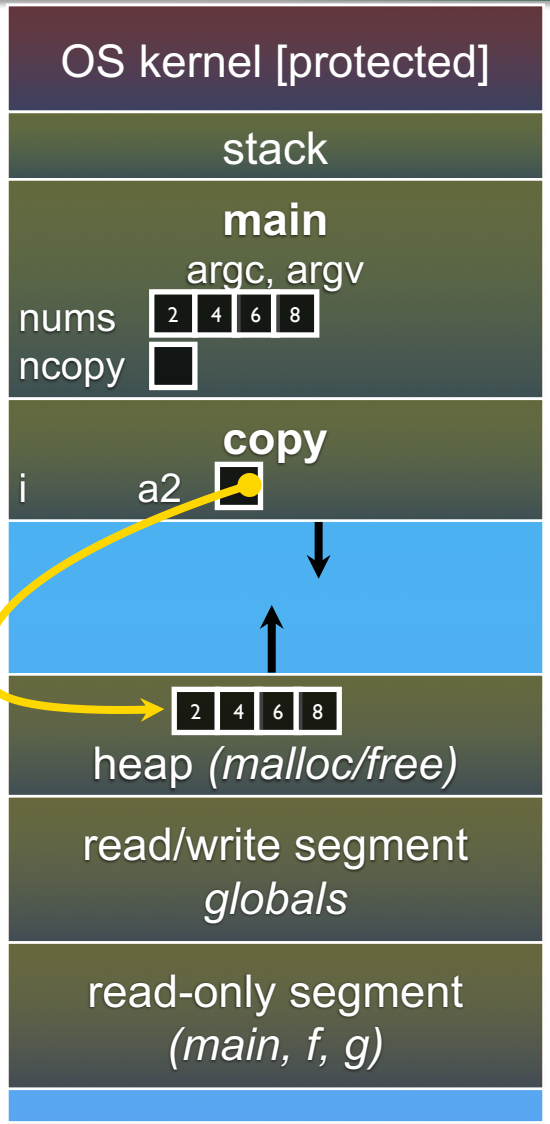
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

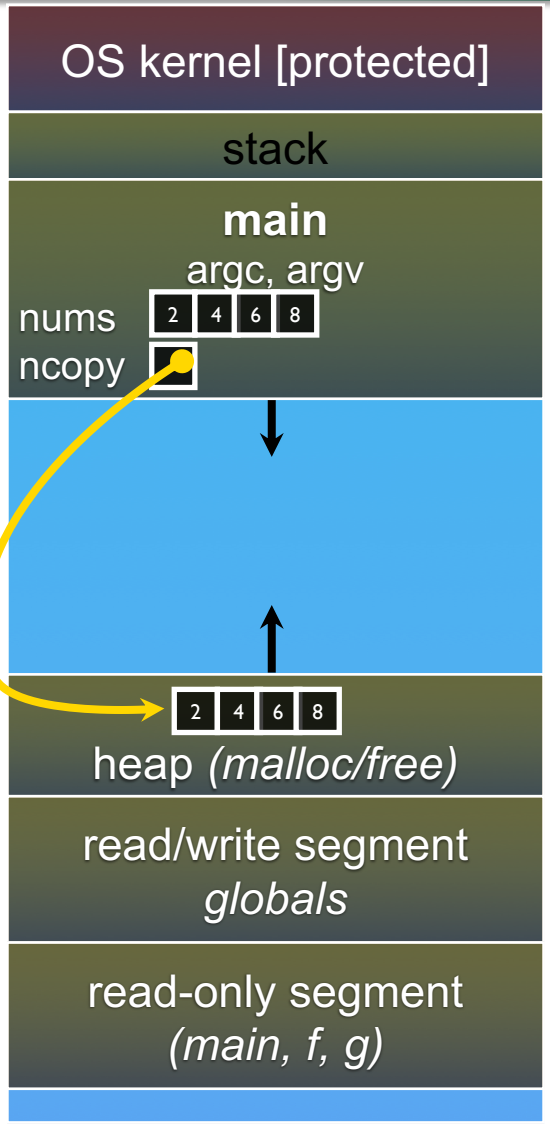
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

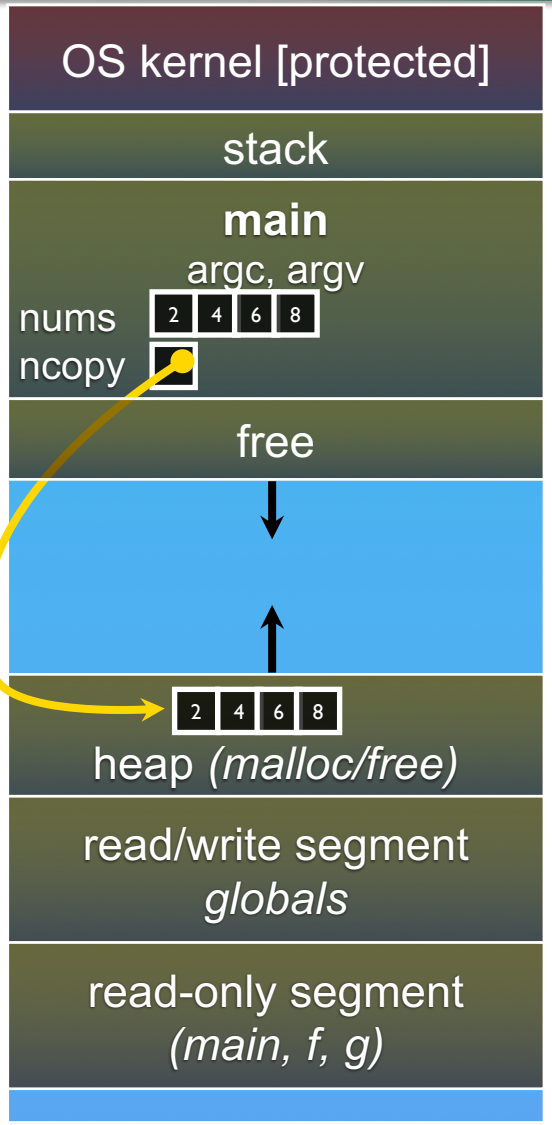
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```



Process Address Space in Action

```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

Process Address Space in Action

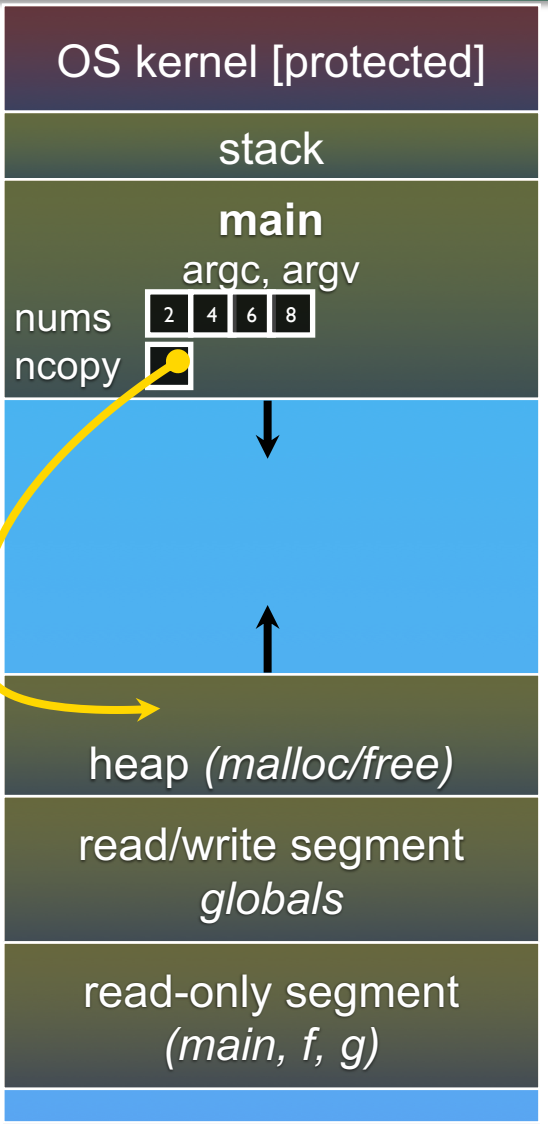
```c
#include <stdlib.h>

int *copy(int a[], int size) {
  int i, *a2;

  a2 = malloc(
    size * sizeof(int));
  if (a2 == NULL)
    return NULL;

  for (i = 0; i < size; i++)
    a2[i] = a[i];
  return a2;
}

int main(...) {
  int nums[4] = {2,4,6,8};
  int *ncopy = copy(nums, 4);
    // ... do stuff ...
  free(ncopy);
  return 0;
}
```

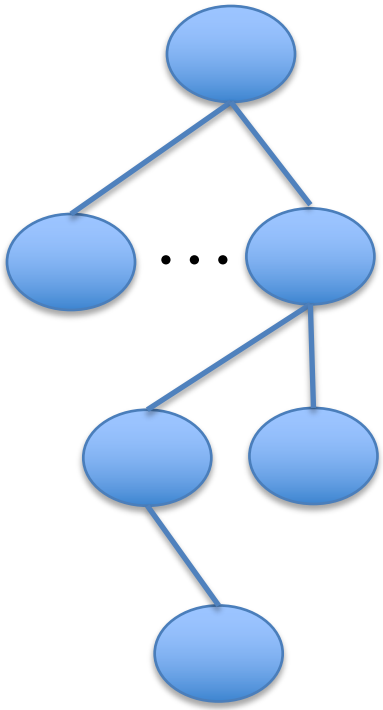


Process Creation

- What happens?
  - New process object in the OS kernel is created
    - build process data structures
  - Allocate address space (abstract resource)
    - later, allocate actual memory (physical resource)
  - Add to execution (ready) queue
    - make it runnable
- Hmm, who created the first process and how?
- Is the OS a process?

Process Creation Options (Parent and Child)

- Process hierarchy options
  - Parent process create 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



Executing a Process

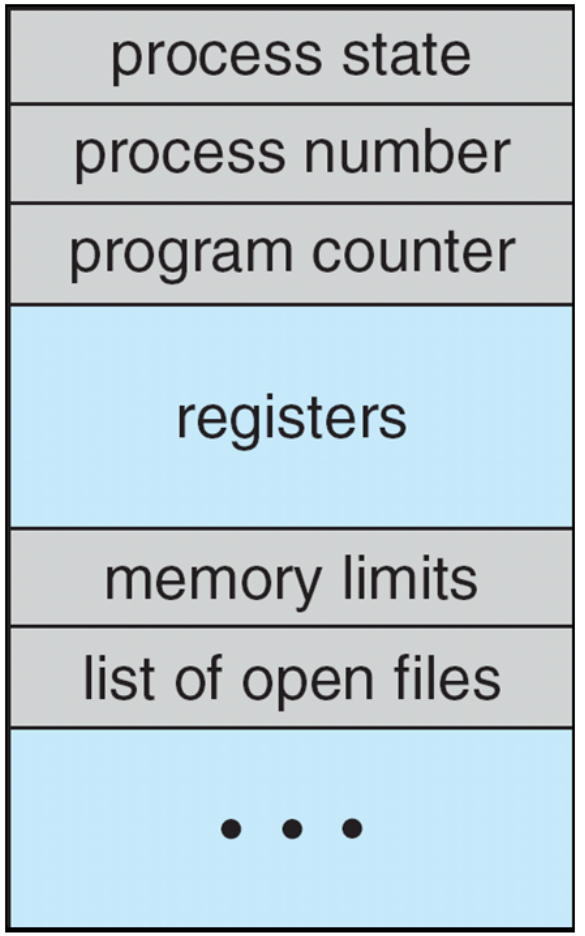
- What is required?
- Registers store state of execution in CPU
  - Stack pointer
  - Data registers
- Program count to indicated what to execute?
  - CPU register holding address of next instruction
- A "thread of execution"
  - Able to execute instructions
  - Has its own stack
  - Each process has at least 1 thread of execution

Executing a Process

- A process thread executes instructions found in the process's address space …
  - Usually the text segment
- … until a trap or interrupt
  - Time slice expires (timer interrupt)
  - Another event (e.g., interrupt from other device)
  - Exception (program error, oops)
  - System call (switch to kernel mode)

Process Control Block (PCB)

- Information associated with each process
  - Also called the *task control block*
- Process state: running, waiting, …
- Program counter
  - Location of instruction to next execute
- CPU registers for process thread
- CPU scheduling information
  - Priorities, scheduling queue pointers, …
- Memory-management information
  - Memory allocated to the process
- Accounting information
  - CPU used, clock time elapsed, …
- I/O status information
  - I/O devices allocated to process
  - List of open files

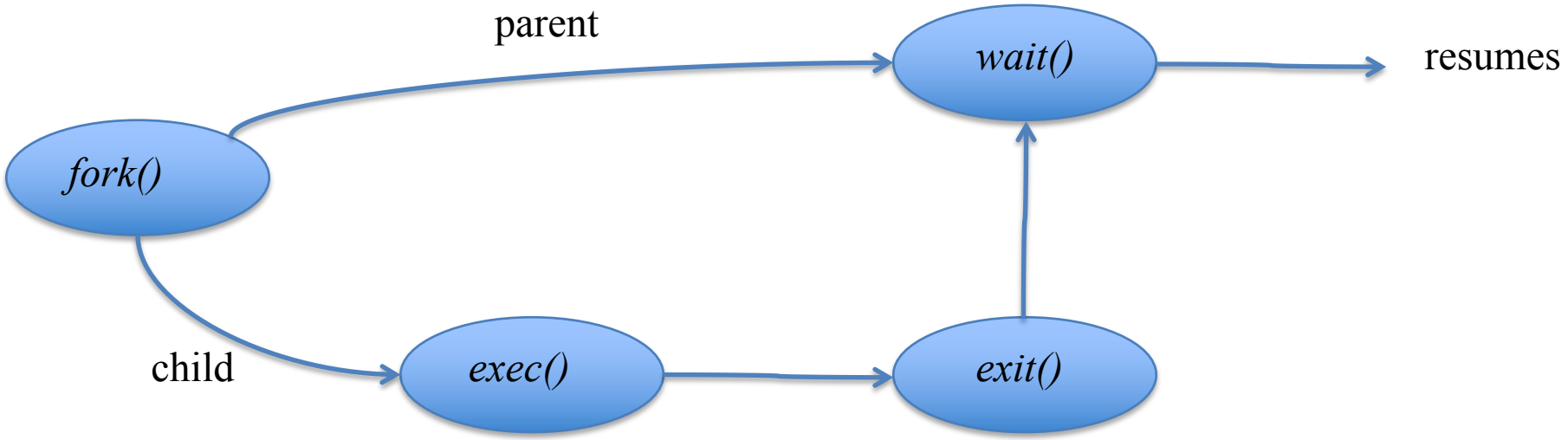


process-specific data structure in the OS

Program Creation System Calls

- *fork()*
  - Copy address space of parent and all threads
- *forkl()*
  - Copy address space of parent and only calling thread
- *vfork()*
  - Do not copy the parent's address space
  - Share address space between parent and child
- *exec()*
  - Load new program and replace address space
  - Some resources may be transferred (open file descriptors)
  - Specified by arguments

Process Creation with New Program



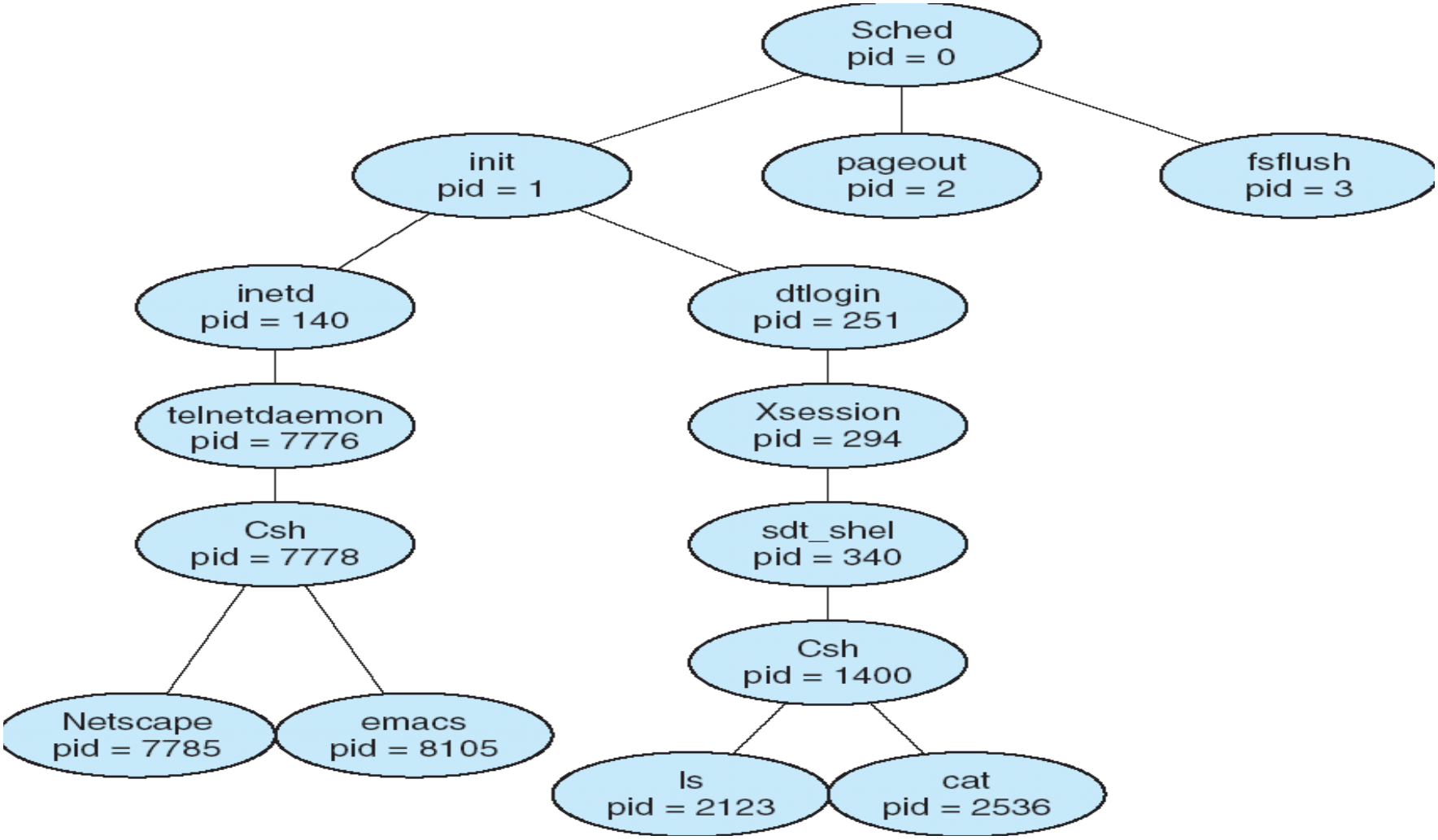
- Parent process calls *fork()* to spawn child process
  - Both parent and child return from fork()
  - Continue to execute the same program
- Child process calls *exec()* to load a new program

C Program Forking Separate Process

```
int main( )
{
pid_t  pid;
  /* fork another process */
  pid = fork( );
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls", "ls", NULL); /* exec a file */
  }
  else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf ("Child Complete");
    exit(0);
  }
}
```

`execl`, `execlp`, `execle`, `execv`, `execvp`, `execvpe` all execute a file and are frontends to `execve`

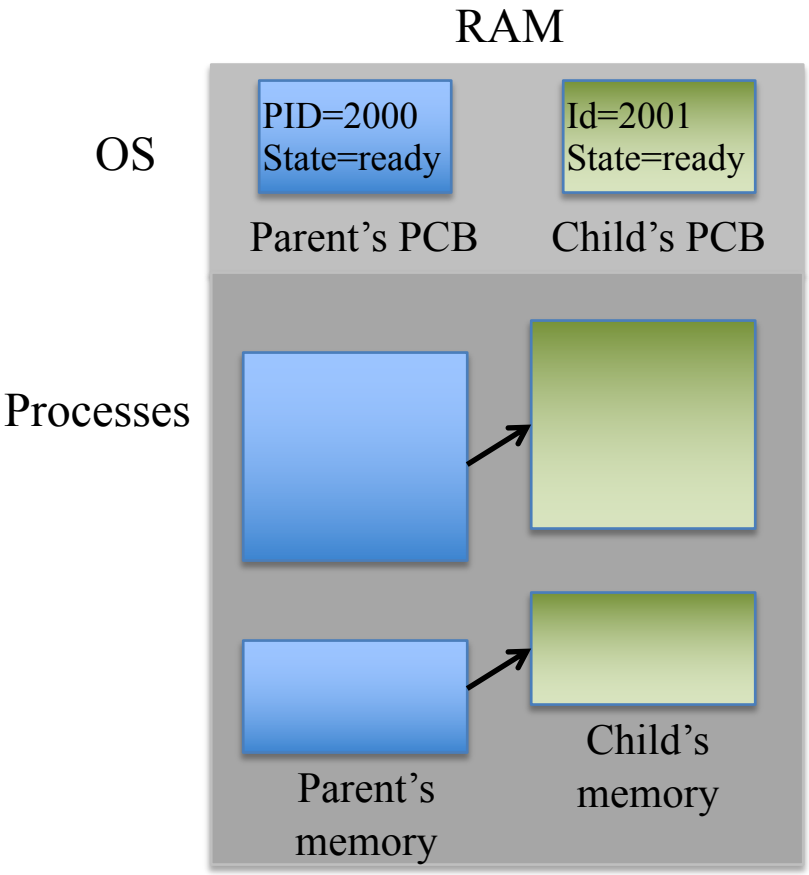
A tree of processes on a typical system



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

Process Layout



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!

Relocatable Memory

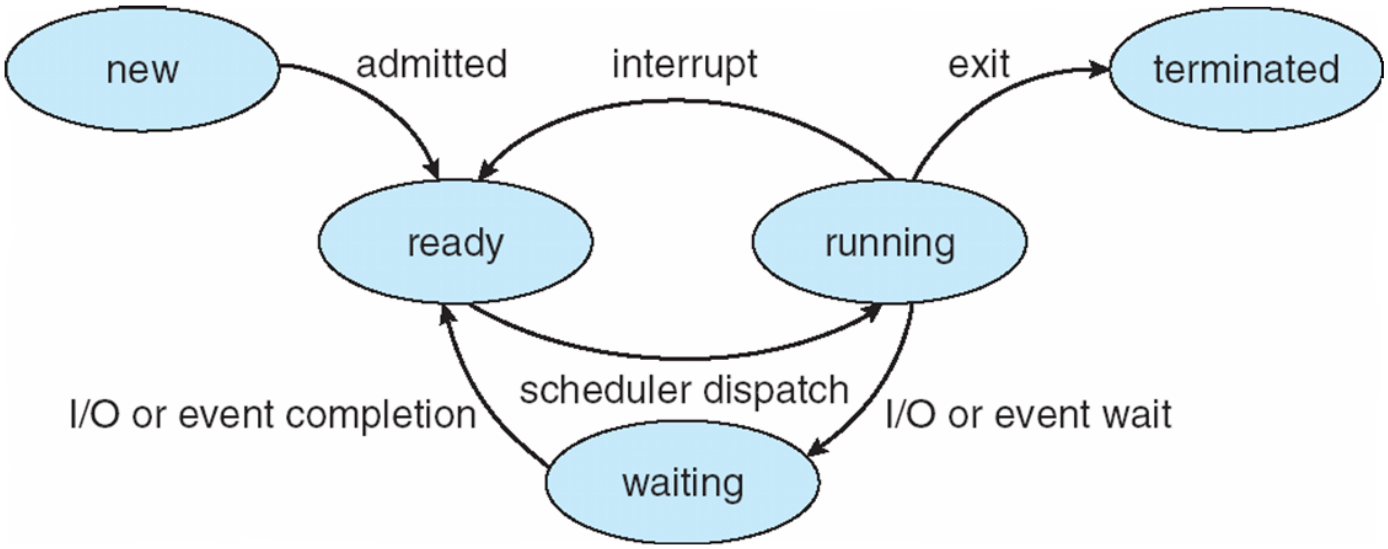
- Program instructions use addresses that logically start at 0
- Cannot place all programs in memory at physical address 0
- Relocation is the mechanism needed that enables the OS to place a program in an arbitrary location in memory
  - Gives the programmer the impression that they own the processor and the memory
- Program is loaded into memory at specific locations
  - Need some form of address translation (relocation) to do this
    - base-limit, segmentation, paging, virtual memory
  - We will talk about this later
- Also, may need to share program code across processes

Process State

- What do we need to track about a process?
  - How many processes?
  - What's the state of each of them?
- Process table
  - Kernel data structure tracking processes on system
- Process control block
  - Structure for tracking process context

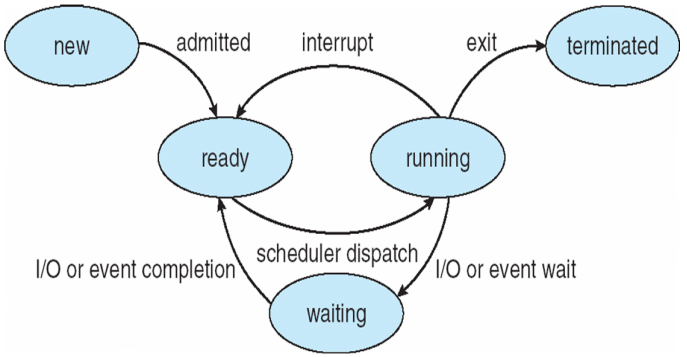
Scheduling Processes

- Processes transition among execution states



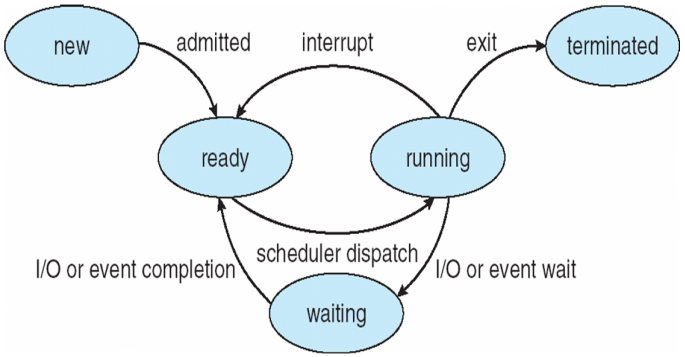
Process States

- Running
  - Process is executing in the processor and in memory with all resources to run
- Ready
  - Process in memory with all resources to run, but is waiting for dispatch onto a processor
- Waiting
  - Process is not running, instead waiting for some event to occur



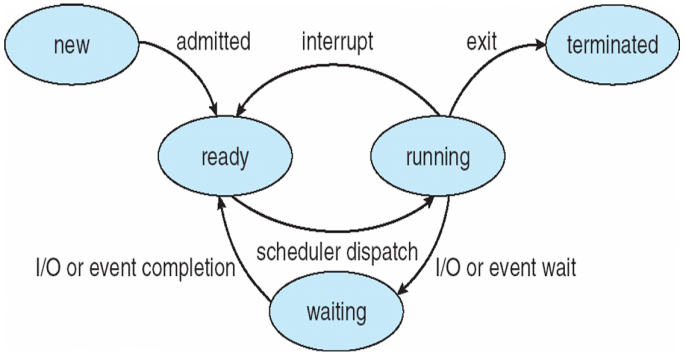
State Transitions

- New Process ==> Ready
  - Allocate resources necessary to run
  - Place process on process queue (usually at end)
- Ready ==> Running
  - Process is at the head of process queue
  - Process is scheduled onto an available processor
- Running ==> Ready
  - Process is interrupted
    - usually by a timer interrupt
  - Process could still run, in that it is not waiting something
  - Placed back on the process queue



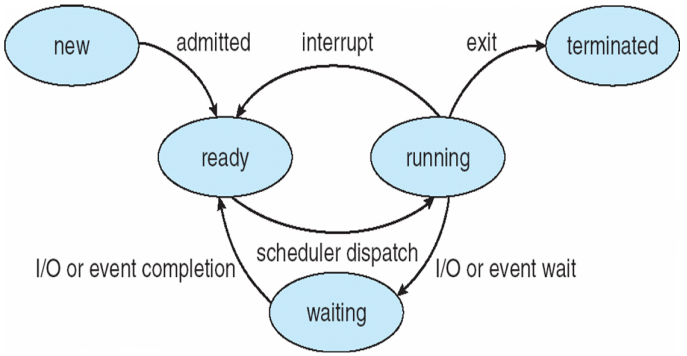
State Transitions: Page Fault Handling

- Running ==> Waiting
  - Either something exceptional happened that caused an interrupt to occur (e.g., page fault exception) …
  - … or the process needs to wait on some action (e.g., it made a system call or requested I/O)
  - Process must wait for whatever event happened to be serviced
- Waiting ==> Ready
  - Event has been satisfied so that the process can return to run
  - Put it on the process queue
- Ready ==> Running
  - As before…



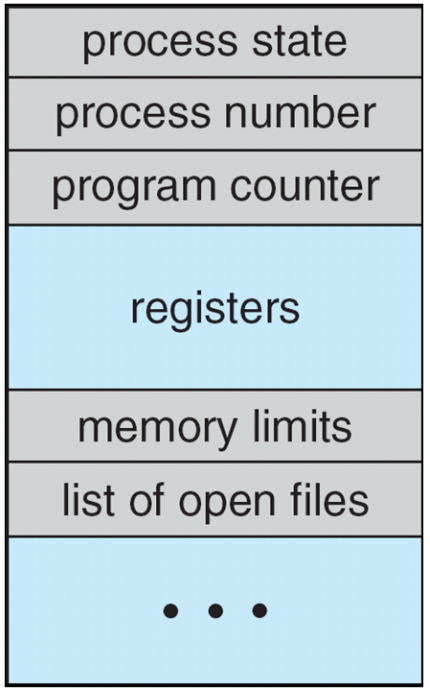
State Transitions: Other Issues

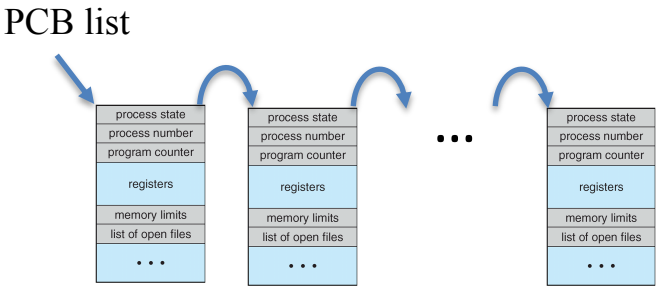
- Priorities
  - Can provide policy indicating which process should run next
    - more when we discuss scheduling…
- Yield
  - System call to give up processor voluntarily
  - For a specific amount of time (sleep)
- Exit
  - Terminating signal (Ctrl-C)



Process Control Block

- Information (metadata) associated with every process kept by the OS (kernel)
  - Process state
  - Program counter
  - CPU registers
  - Scheduling information
  - Memory management information
  - Accounting information
  - I/O status information
- OS maintains a list of PCBs for all processes
- Process state lists link in the PCBs





Per Process Control Information

- Process state
  - Ready, running, waiting (momentarily)
- Links to other processes
  - Children
- Memory Management
  - Segments and page tables
- Resources
  - Open files
- And much more…

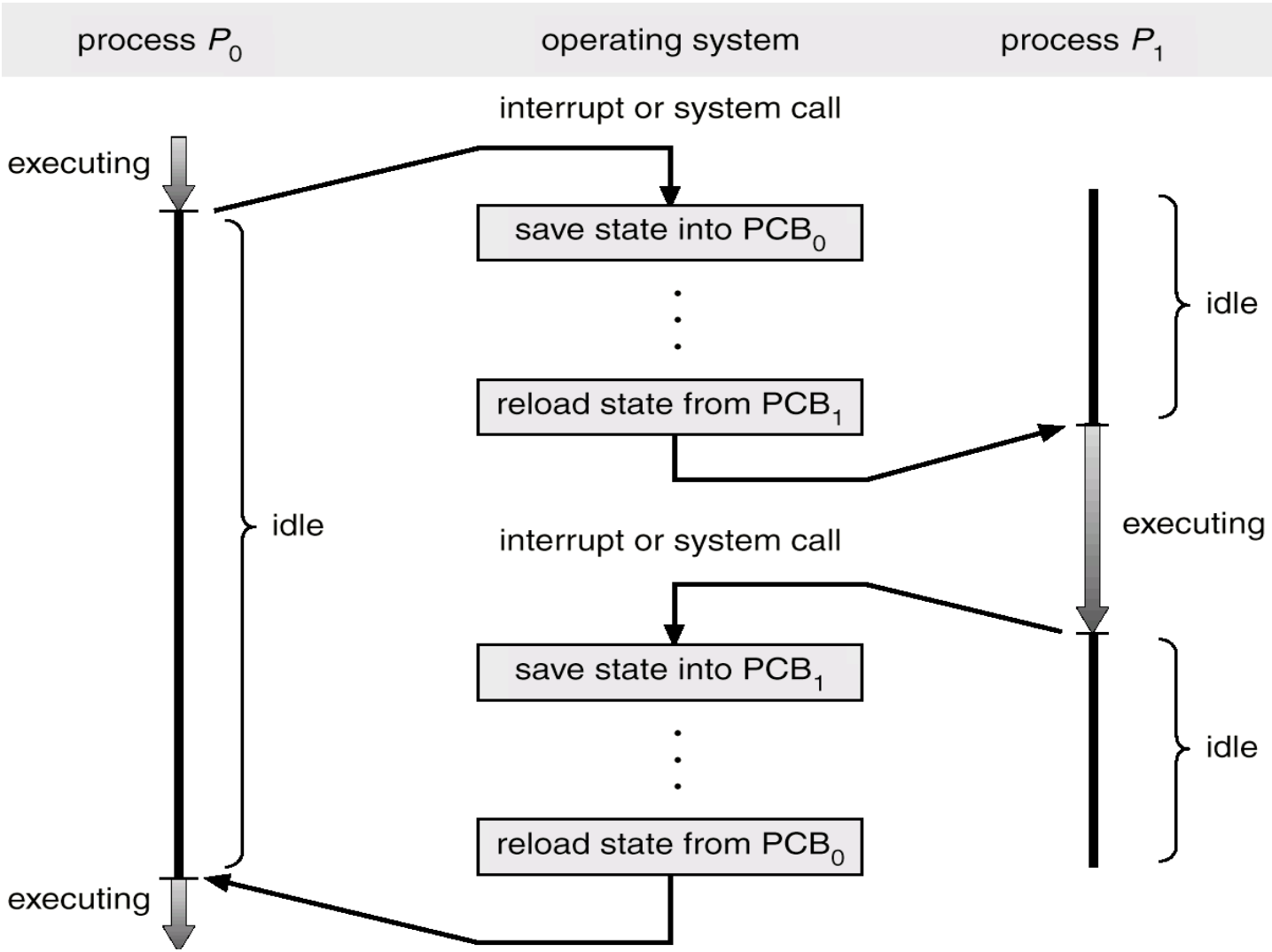
/proc File System

- Linux and Solaris
  - `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
- Various process information
  - `/proc/<pid>/io`
    I/O statistics
  - `/proc/<pid>/environ`
    Environment variables (in binary)
  - `/proc/<pid>/stat`
    Process status and info
- Check out "man proc"

Context Switch

- OS switches from one execution context to another
  - One process to another process
  - Interrupt handling
  - Process to kernel (mode transition, not context switch)
- 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
- Does this differ if we are running an interrupt handler?

Context Switch



Context Switch Performance

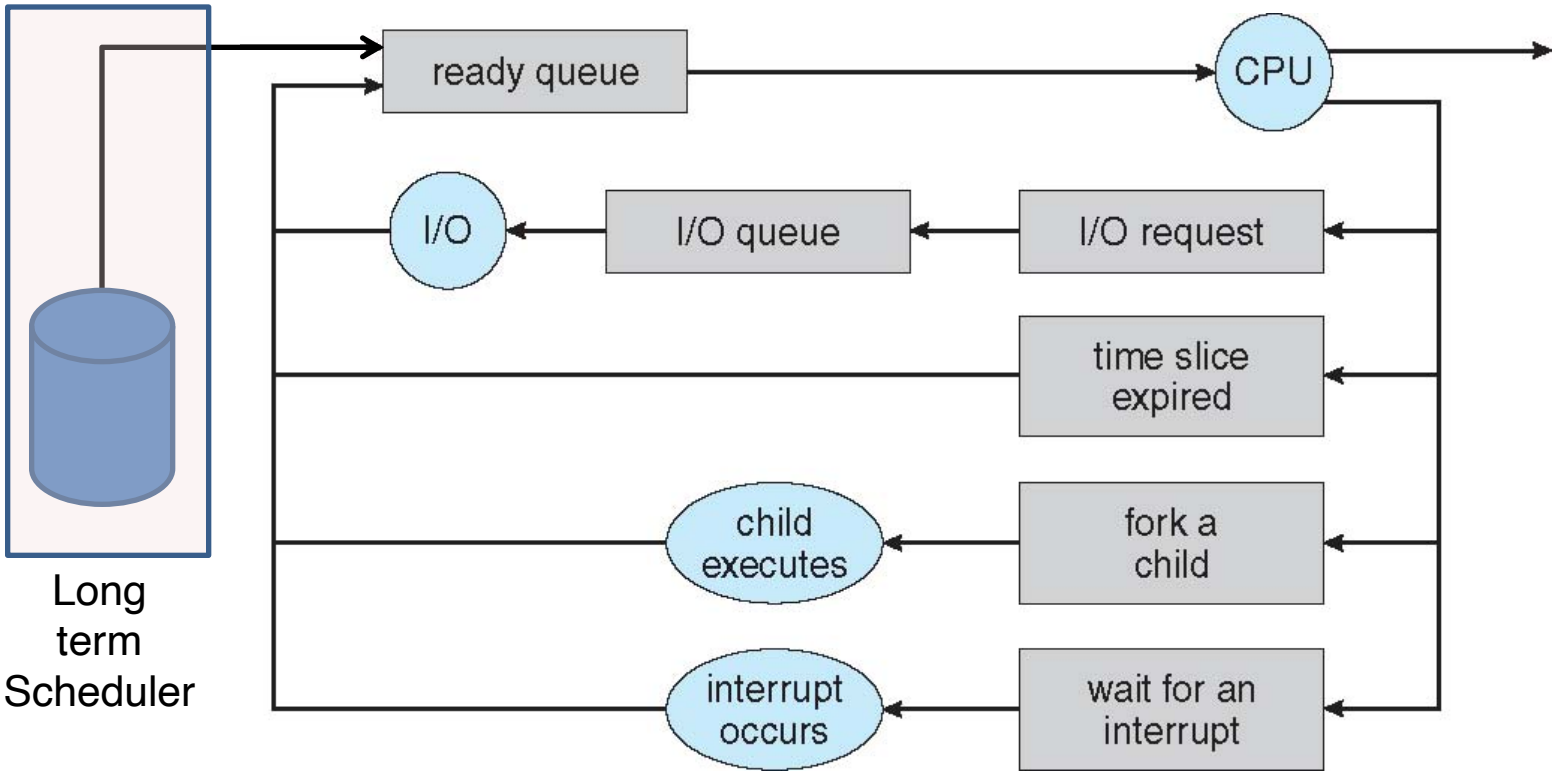
- No useful work is being done during a context switch (i.e., no user process is running)
  - Want to speed up context switch processing
  - If a system call can be don in user mode, then the OS does not have to context switch
- Hardware support helps in context switching
  - Multiple hardware register sets
  - Be able to quickly set up the processor
- However, hardware optimization may conflict
  - Managing address translation tables is difficult
  - Different virtual to physical mappings on different processes

Process Scheduling

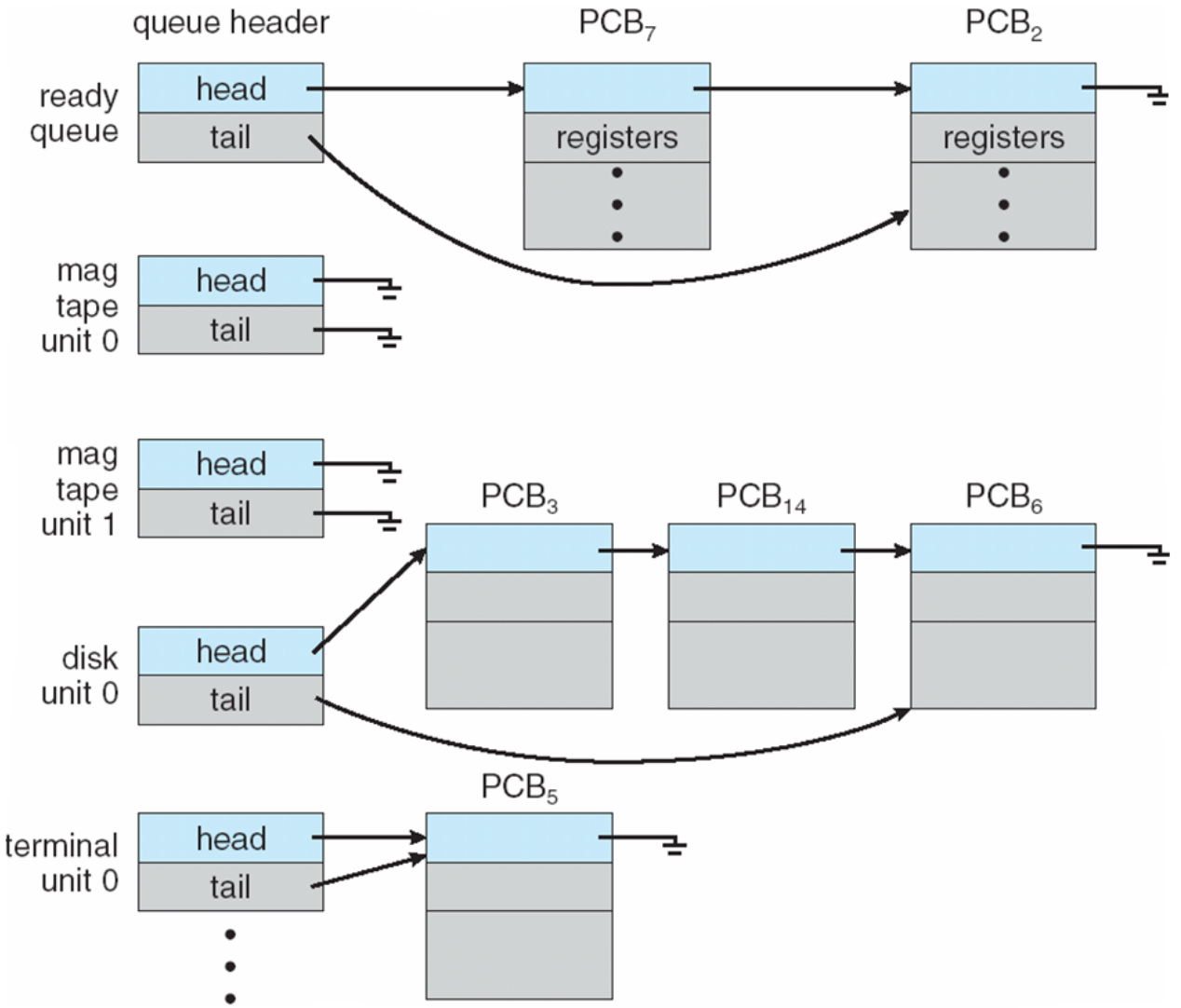
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - *Job queue* – set of all processes in the system
  - *Ready queue* – set of all processes residing in main 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



Schedulers

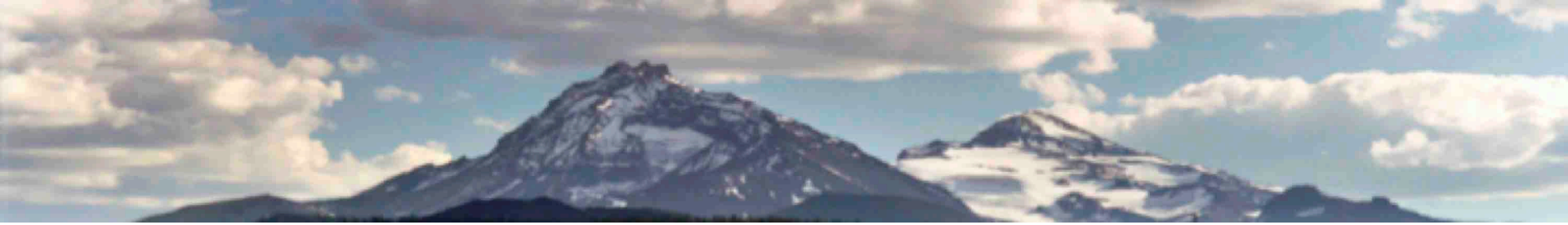
- *Short-term scheduler* (process scheduler)
  - Selects which process should be executed next
  - Allocates CPU to running process
- *Medium-term scheduler* (multiprogram scheduler)
  - Manages process (jobs) in execution
  - Moves partially executed jobs to/from disk storage
  - Adjusts the degree of multiprogramming
- *Long-term scheduler* (job scheduler)
  - Selects which jobs should be allowed to run
  - Loads process and makes it ready to run

Project 1 – Building a Pseudo-Shell

- A "shell" in Linux (Unix) is an interactive, command-line program that support to a user to run commands, manage control flow, and interact with the OS at a higher level
  - Popular shells: bash, (t)csh, ksh, zsh, …
- Shells must parse and process user commands
- This project will build a simple shell from scratch
  - Single threaded
  - Synchronous
  - Supports a simple command interface
  - Implements a limited set of functions
- Objective is to develop in interactive program that uses parsing/tokenization and system calls to implement command functions
- Project 2 will add more capabilities for process control

Next Class

- Process scheduling
- Interprocess communication (IPC)



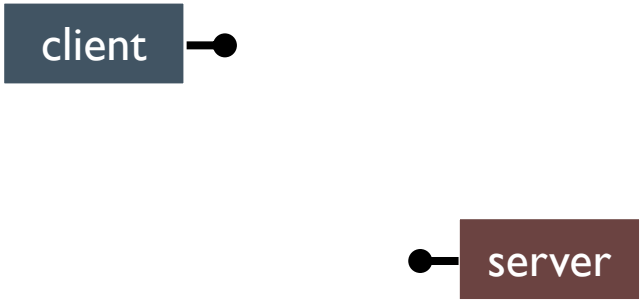
Extra Slides

Process Actions in Client-Server (1)

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

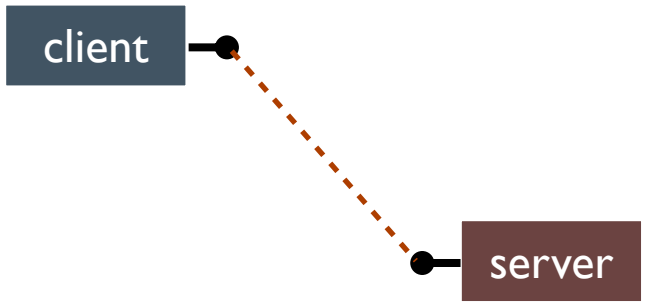
Process Actions in Client-Server (2)

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



Process Actions in Client-Server (3)

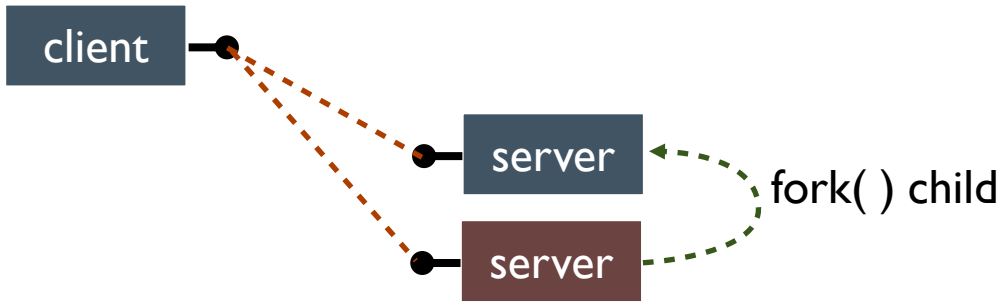
- An interprocess communication is made and a connection is established



- 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)

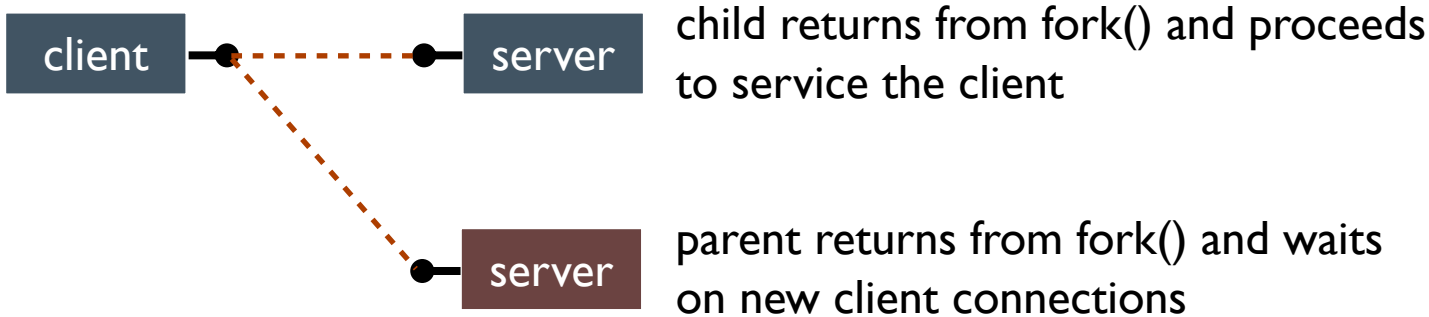
- Create more "concurrency" by creating more processes!



- 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)

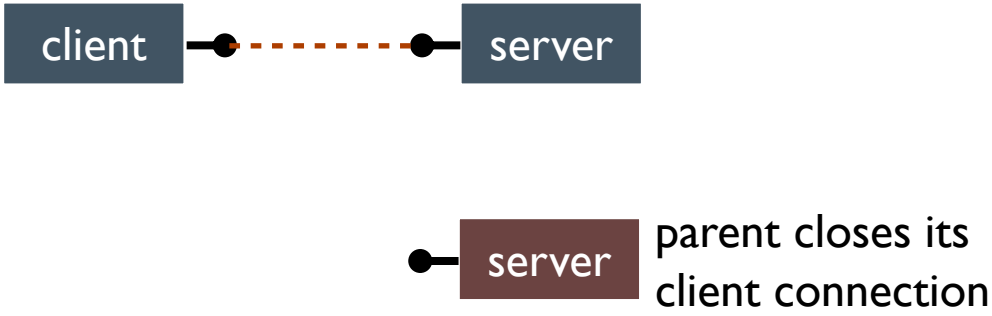
- Responsibility for "servicing" the client is handed off to the "child" server process



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

Process Actions in Client-Server (6)

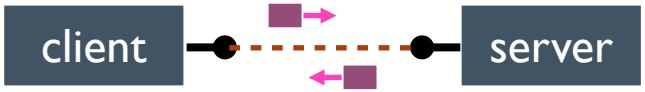
- The "parent" server should disengage from the client by releasing its (duplicate) connection



- 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)

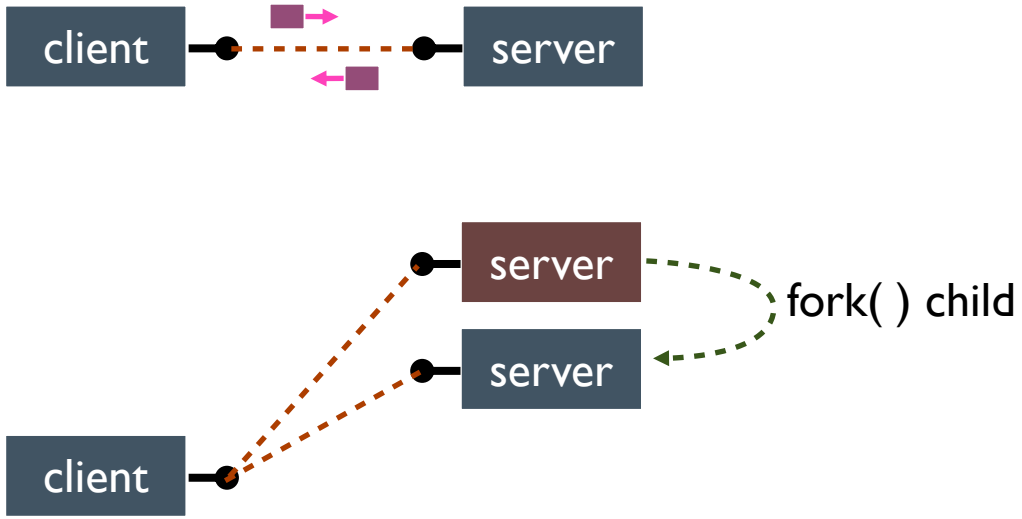
- Communication takes places between the client and "child" server process



- The "parent" server process is not involved

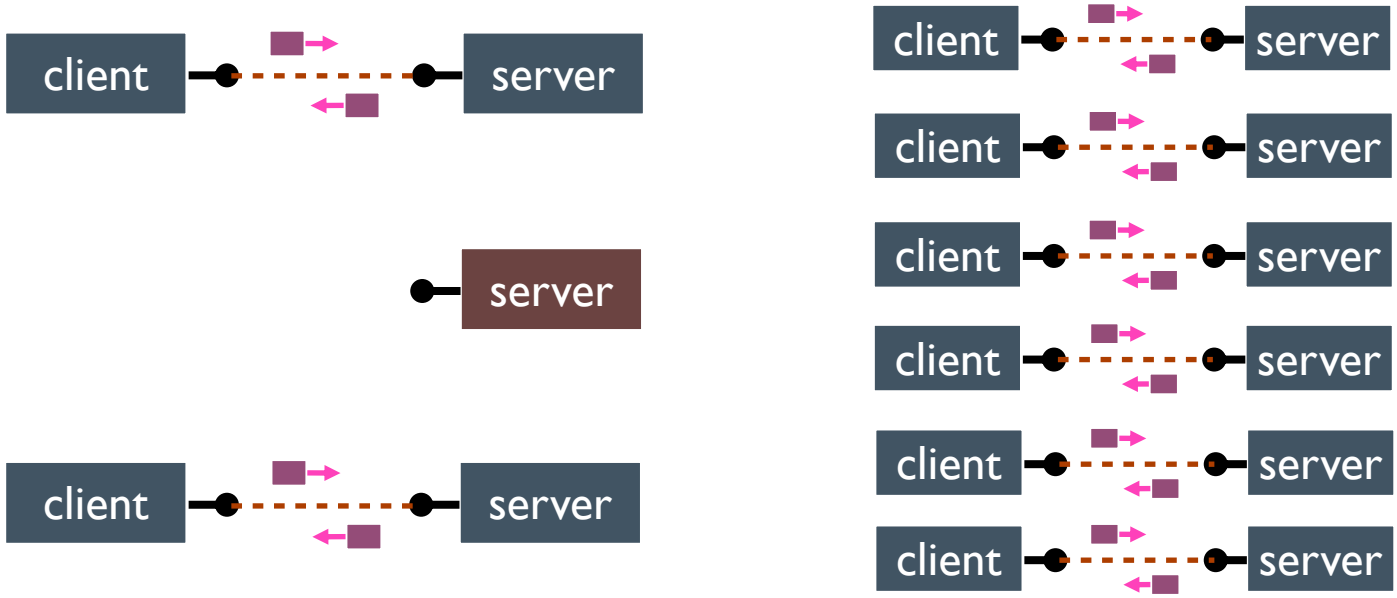
Process Actions in Client-Server (8)

- 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