UNIVERSITY OF OREGON

CIS 415: Operating Systems

Processes

Prof. Kevin Butler

Spring 2014

- Last class:
  - Operating system structure
- Today:
  - More basics, system calls, Process Management

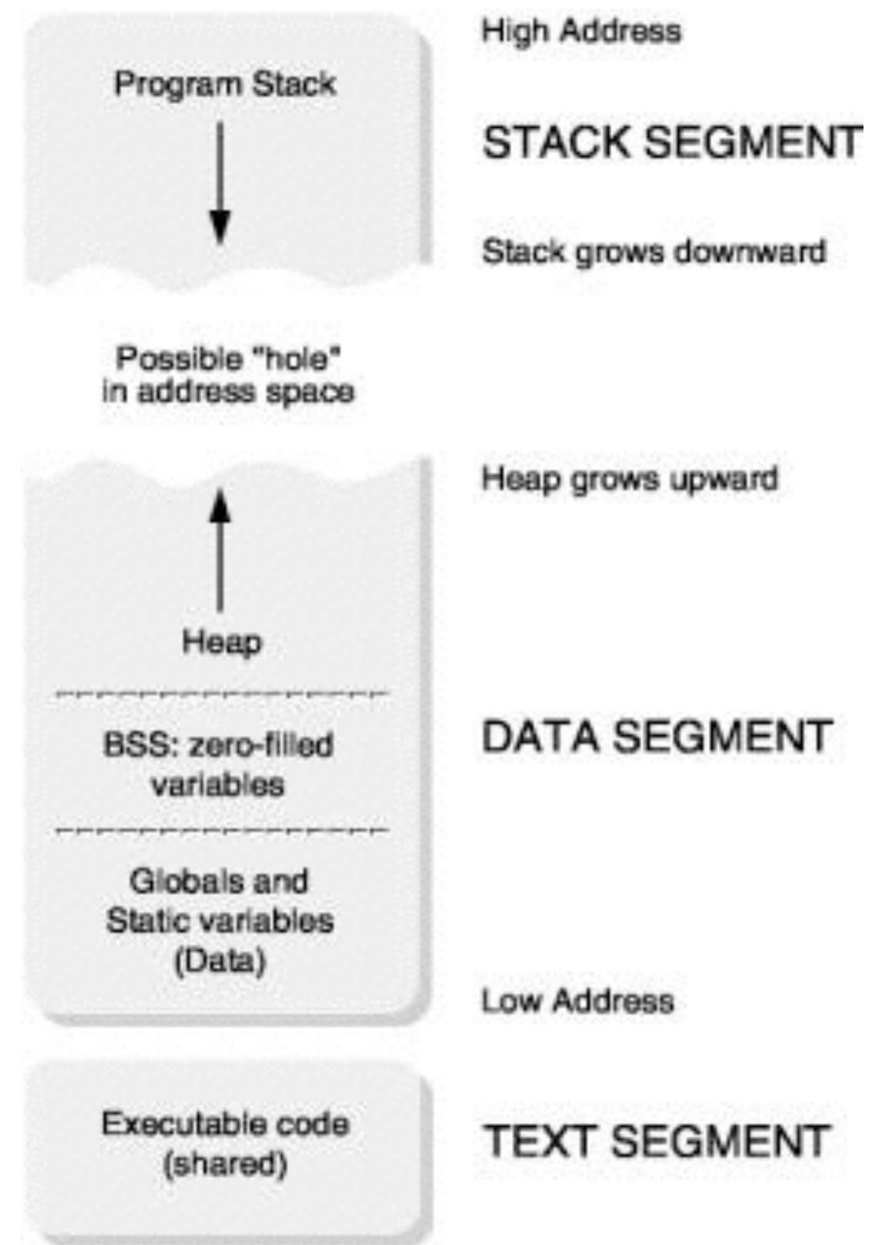
Administrivia

- Lab sections: everyone should know where you're going
  - this week: debugging
- Assignment 1: due April 22
- Project 1: out today, due April 24
- Manage your time wisely!

Process Address Space

UNIVERSITY OF OREGON

- All locations addressable by the process
- Can restrict use of addresses (RW)
- Restrictions enforced by OS
- Every running program can have its own private address space
  - How?



System Call Handling

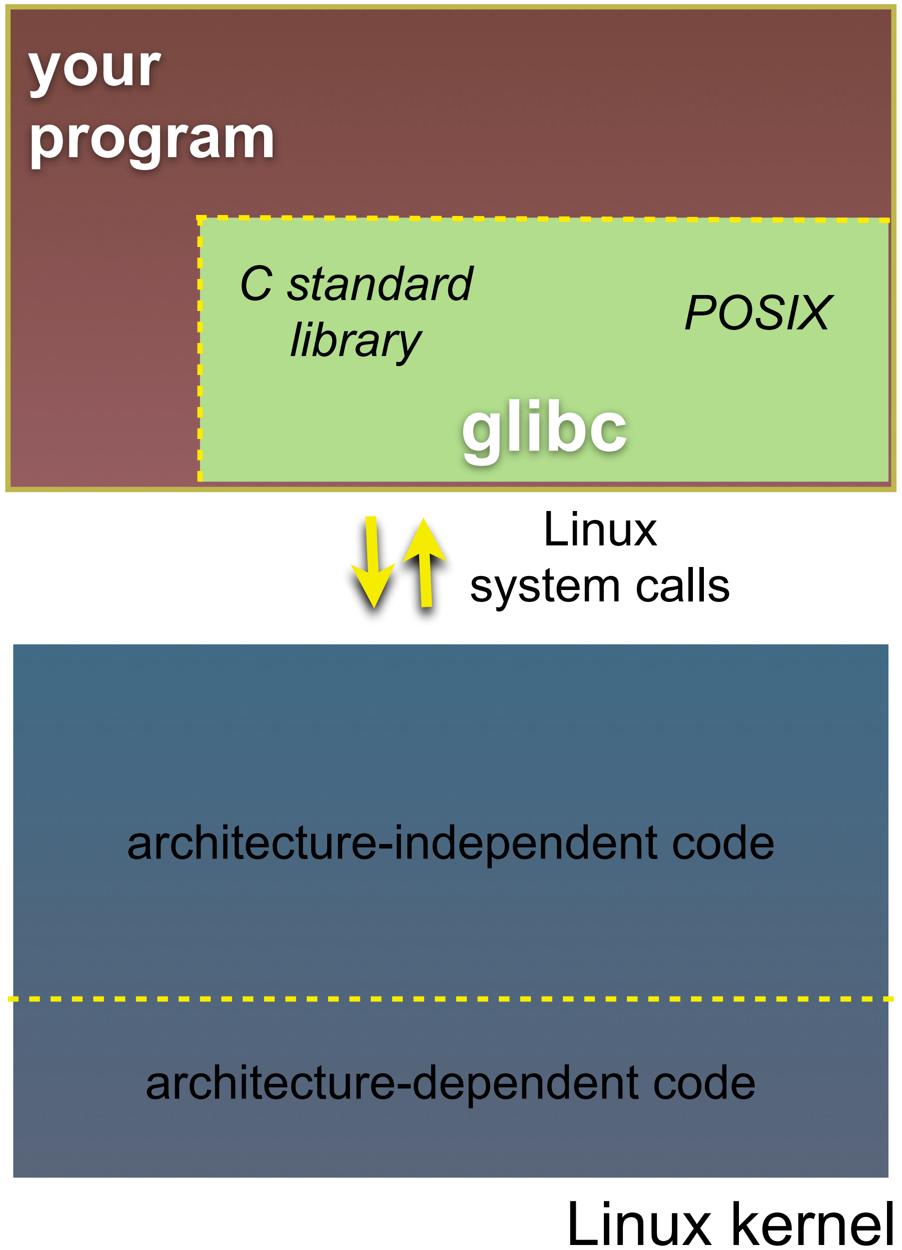
UNIVERSITY OF OREGON

Procedure call in user process	
Initial work in user mode	*(libc)*
Trap instruction to invoke kernel	*(int 0x80)*
Preparation	*(e.g., sys_read, mmap2)*
I/O command	*(read from disk)*
Wait	*(disk is slow)*
Completion	*(interrupt handling)*
Return-from-interrupt instruction	
Final work in user mode	*(libc)*
Ordinary return instruction	

Details on x86 / Linux

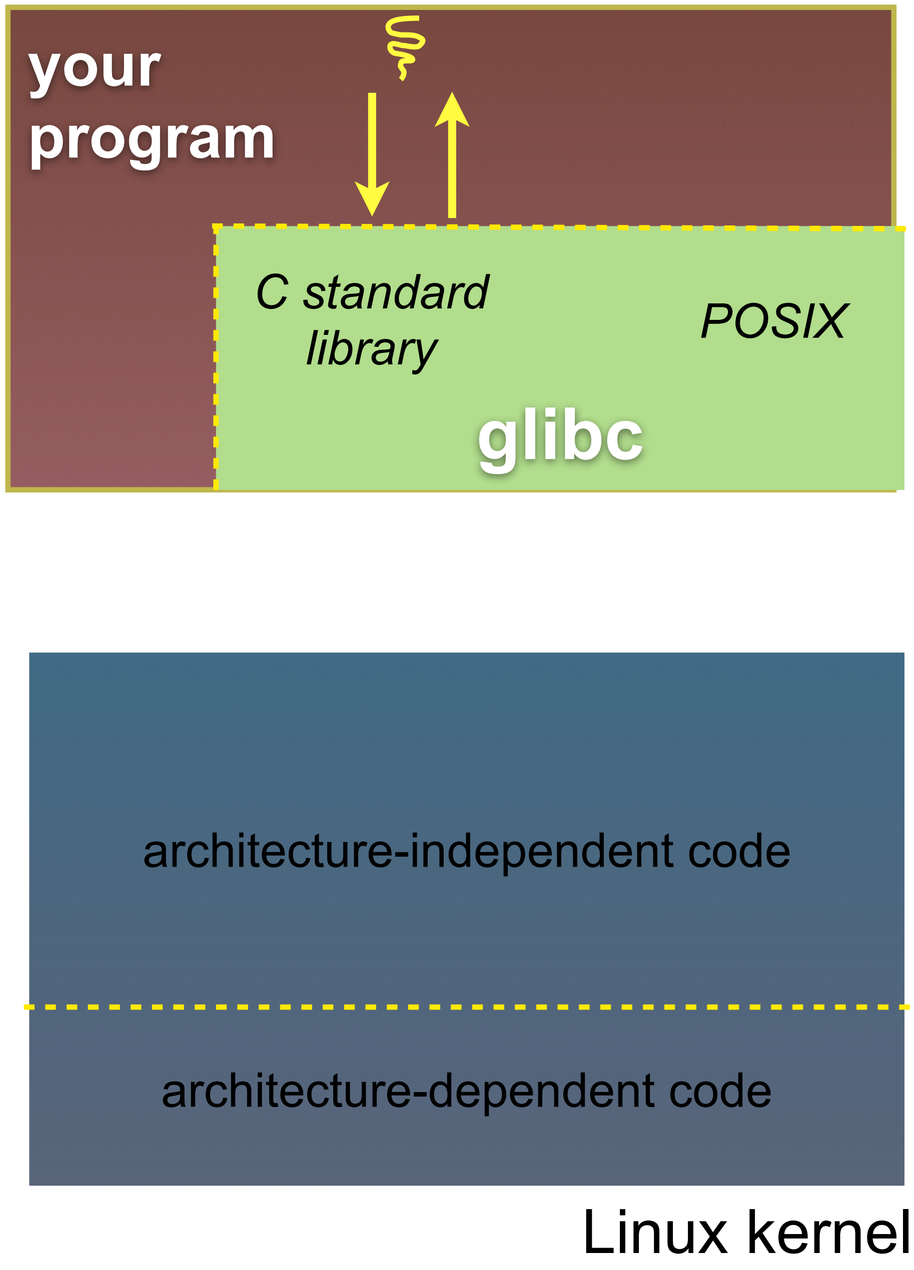
UNIVERSITY OF OREGON

- A more accurate picture:
  - consider a typical Linux process
  - its thread of execution can be several places
    - in your program's code
    - in **glibc**, a shared library containing the C standard library, POSIX support, and more
    - in the Linux architecture-independent code
    - in Linux x86-32/x86-64 code



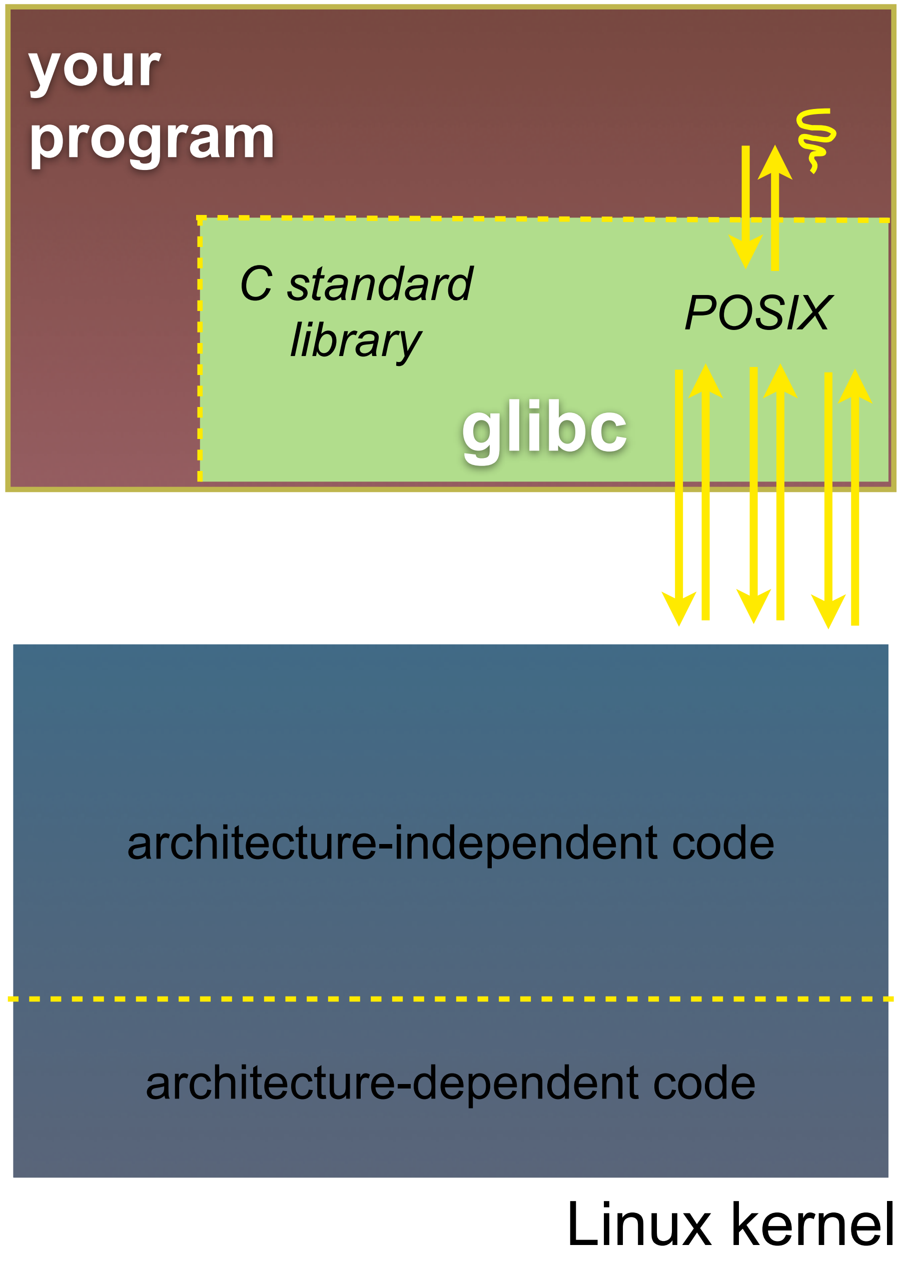
Details on x86 / Linux

- Some routines your program invokes may be entirely handled by glibc
  - without involving the kernel
    - e.g., **strcmp( )** from stdio.h
  - ∃ some initial overhead when invoking functions in dynamically linked libraries
  - but, after symbols are resolved, invoking glibc routines is nearly as fast as a function call within your program itself



Details on x86 / Linux

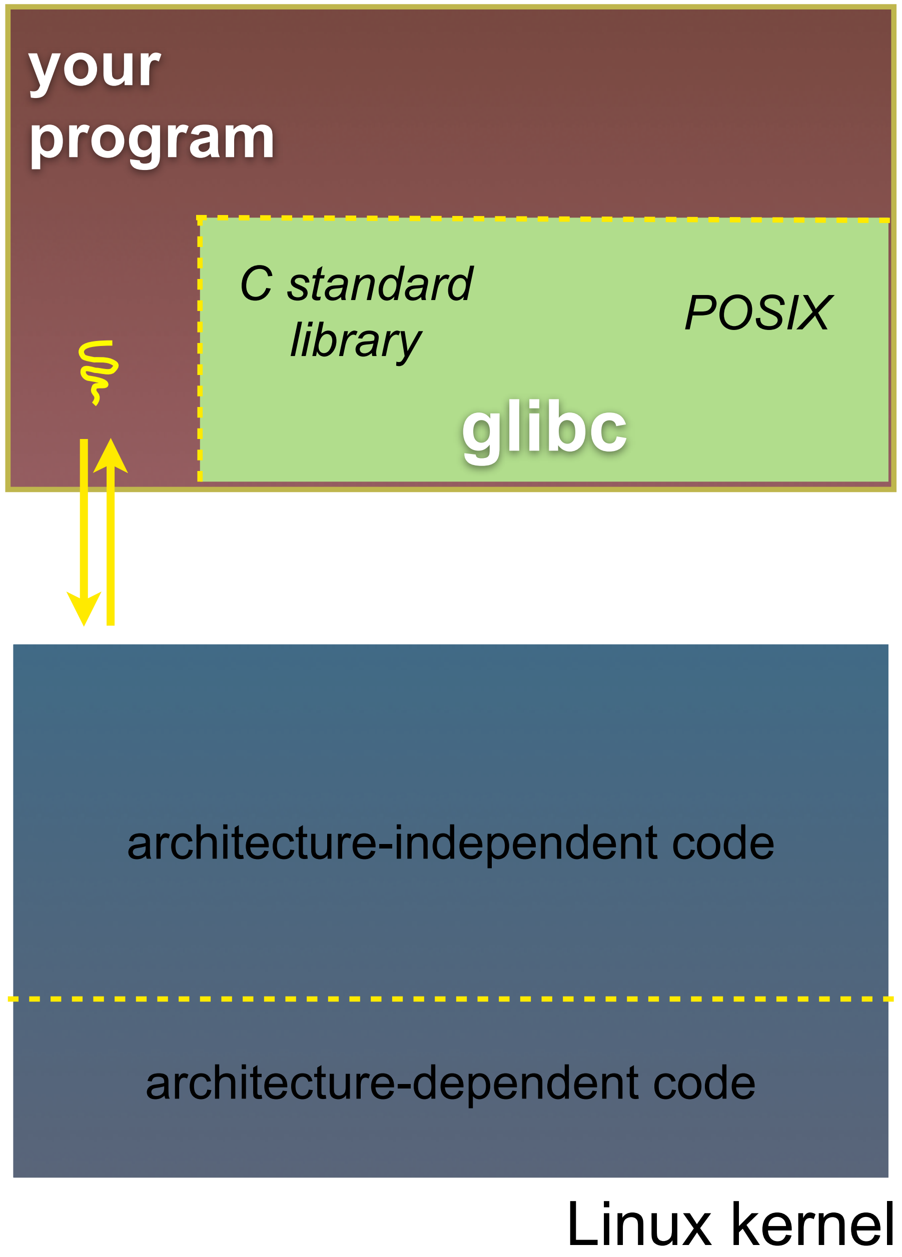
- Some routines may be handled by glibc, but they in turn invoke Linux system calls
  - e.g., POSIX wrappers around Linux syscalls
    - POSIX readdir( ) invokes the underlying Linux readdir( )
  - e.g., C stdio functions that read and write from files
    - fopen( ), fclose( ), fprintf( ) invoke underlying Linux open( ), read( ), write( ), close( ), etc.



Details on x86 / Linux

UNIVERSITY OF OREGON

- Your program can choose to directly invoke Linux system calls as well
  - nothing forces you to link with glibc and use it
  - but, relying on directly invoked Linux system calls may make your program less portable across UNIX varieties



File Interface

- Goal: Provide a uniform abstraction for accessing the OS and its resources
- Abstraction: File
  - Use file system calls to access OS services
  - Devices, sockets, pipes, etc.
  - And OS in general

I/O with System Calls

UNIVERSITY OF OREGON

- Much I/O is based on a streaming model
  - sequence of bytes
- `write()` sends a stream of bytes somewhere
- `read()` blocks until a stream of input is ready
- Annoying details:
  - might fail, can block for a while
  - file descriptors...
  - arguments are pointers to character buffers
  - see the read() and write() man pages

File Descriptors

- A process might have several different I/O streams in use at any given time
- These are specified by a kernel data structure called a *file descriptor*
  - each process has its own table of file descriptors
- `open()` associates a file descriptor with a file
- `close()` destroys a file descriptor
- Standard input and standard output are usually associated with a terminal
  - more on that later

Regular File

UNIVERSITY OF OREGON

- File has a pathname: /tmp/foo
- Can open the file
  - int fd = open(“/tmp/foo”, O_RDWR)
  - For reading and writing
- Can read from and write to the file
  - bytes = read(fd, buf, max); /* buf get output */
  - bytes = write(fd, buf, len); /* buf has input */

flags for read/write access

pointer to buffer

Socket File

UNIVERSITY OF OREGON

- File has a pathname: /tmp/bar
  - Files provide a persistence for a communication channel
  - Usually used for local communication (UNIX domain sockets)
- Open, read, and write via socket operations
  - sockfd = socket(AF_UNIX, TCP_STREAM, 0);
  - local.path is set to /tmp/bar
  - bind (sockfd, &local, len)
  - Use sock operations to read and write

Device File

UNIVERSITY OF OREGON

- Files for interacting with physical devices
  - /dev/null (do nothing)
  - /dev/cdrom (CD-drive)
- Use file system operations, but are handled in device-specific ways
  - `open`, `read`, `write` correspond to device-specific functions
    - Function pointers!
  - Also, use `ioctl` (I/O control) to interact (later)

Sysfs File and `/proc` Files

UNIVERSITY OF OREGON

- These files enable reading from and writing to kernel
- `/proc` files
  - enable reading of kernel state for a process
- Sysfs files
  - Provide functions that update kernel data
    - File's `write` function updates kernel based on input data

Other System Calls

UNIVERSITY OF OREGON

- It’s possible to hook the output of one program into the input of another: `pipe()`
- It’s possible to block until one of several file descriptor streams is ready: `select()`
- Special calls for dealing with network
  - `AF_INET` sockets, etc.
- Send a message to other (or all) processes: `signal()`
- Most of these in section 2 of manual
  - e.g., `man 2 select`

Syscall Functionality

UNIVERSITY OF OREGON

- System calls are the main interface between processes and the OS
  - like an extended "instruction set" for user programs that hide many details
  - first Unix system had a couple dozen system calls
  - current systems have many more (>300 in Linux, >500 in FreeBSD)
  - Understanding the system call interface of a given OS lets you write useful programs under it
- Natural questions to ask:
  - is this the right interface? how to evaluate?
  - how can these system calls be implemented?

Why Processes?

- We have programs, so why do we need processes?

Overview

UNIVERSITY OF OREGON

- Questions that we explore
  - How are processes created?
    - From binary program to 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

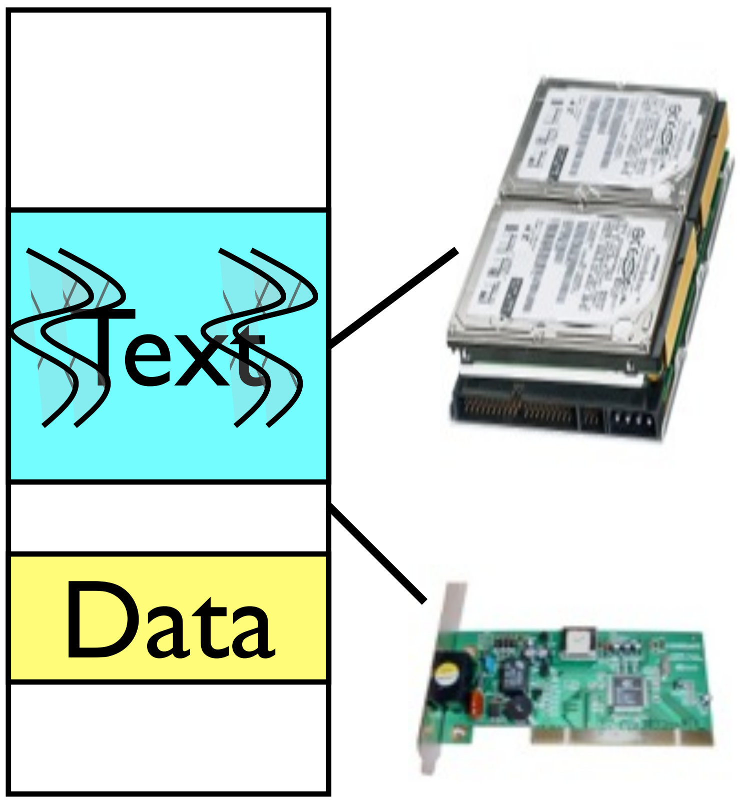
Supervisor and User Modes

UNIVERSITY OF OREGON

- OS runs in supervisor mode
  - Has access to protected instructions only available in that mode (ring 0)
  - Can manage the entire system
- OS loads processes into user mode
  - Many processes can run in user mode
- How does OS get programs loaded into processes in user mode and keep them straight?

Process

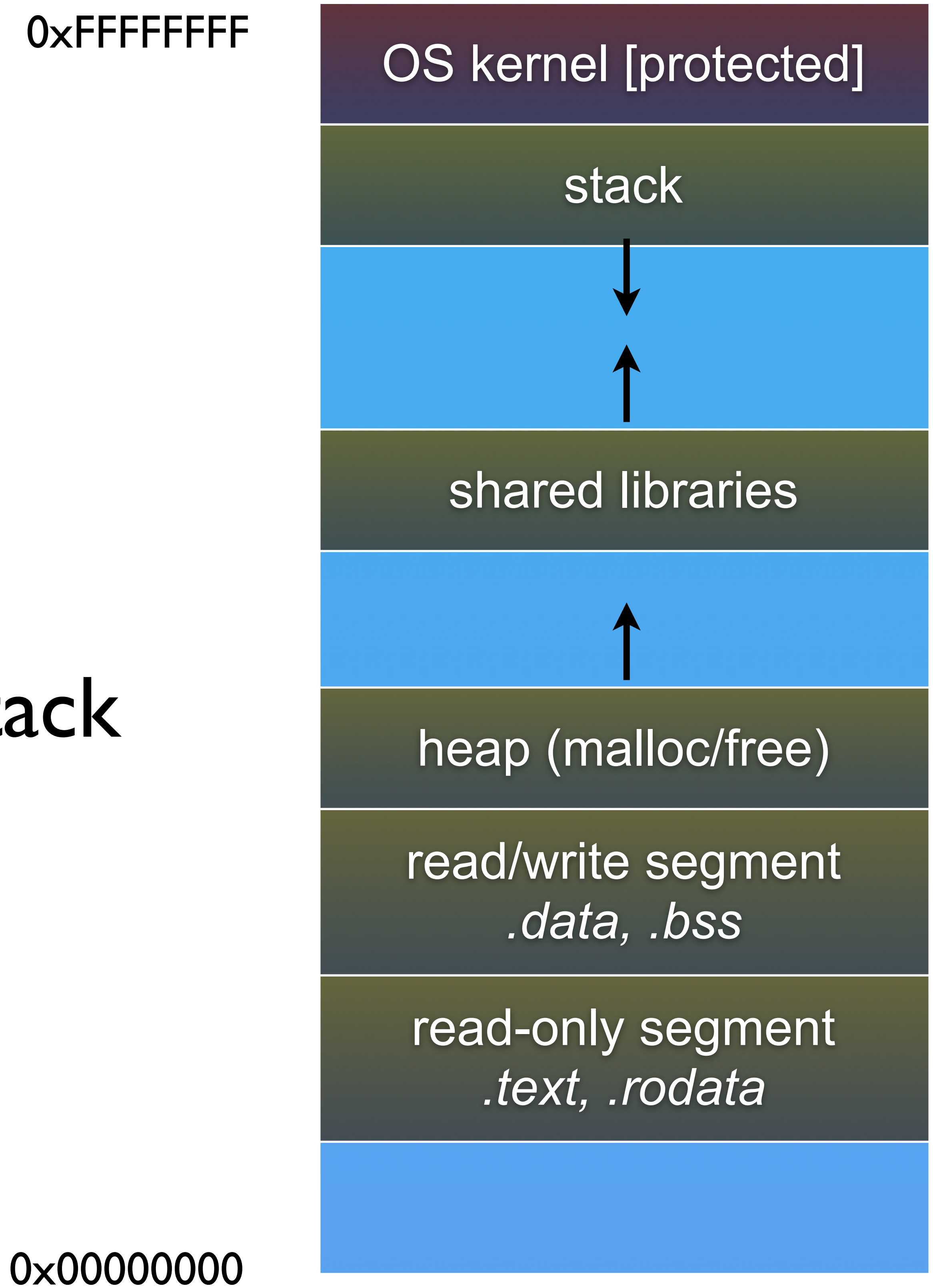
- Address space + threads + resources
- Address space contains code and data of a process
- Threads are individual execution contexts
- Resources are physical support necessary to run the process (memory, disk, …)



Process Address Space

UNIVERSITY OF OREGON

- Program (Text)
- Global Data (Data)
- Dynamic Data (Heap)
- Thread-local Data (Stack)
- Each thread has its own stack



Process Address Space

UNIVERSITY OF OREGON

```
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;
}
```

Heap + stack

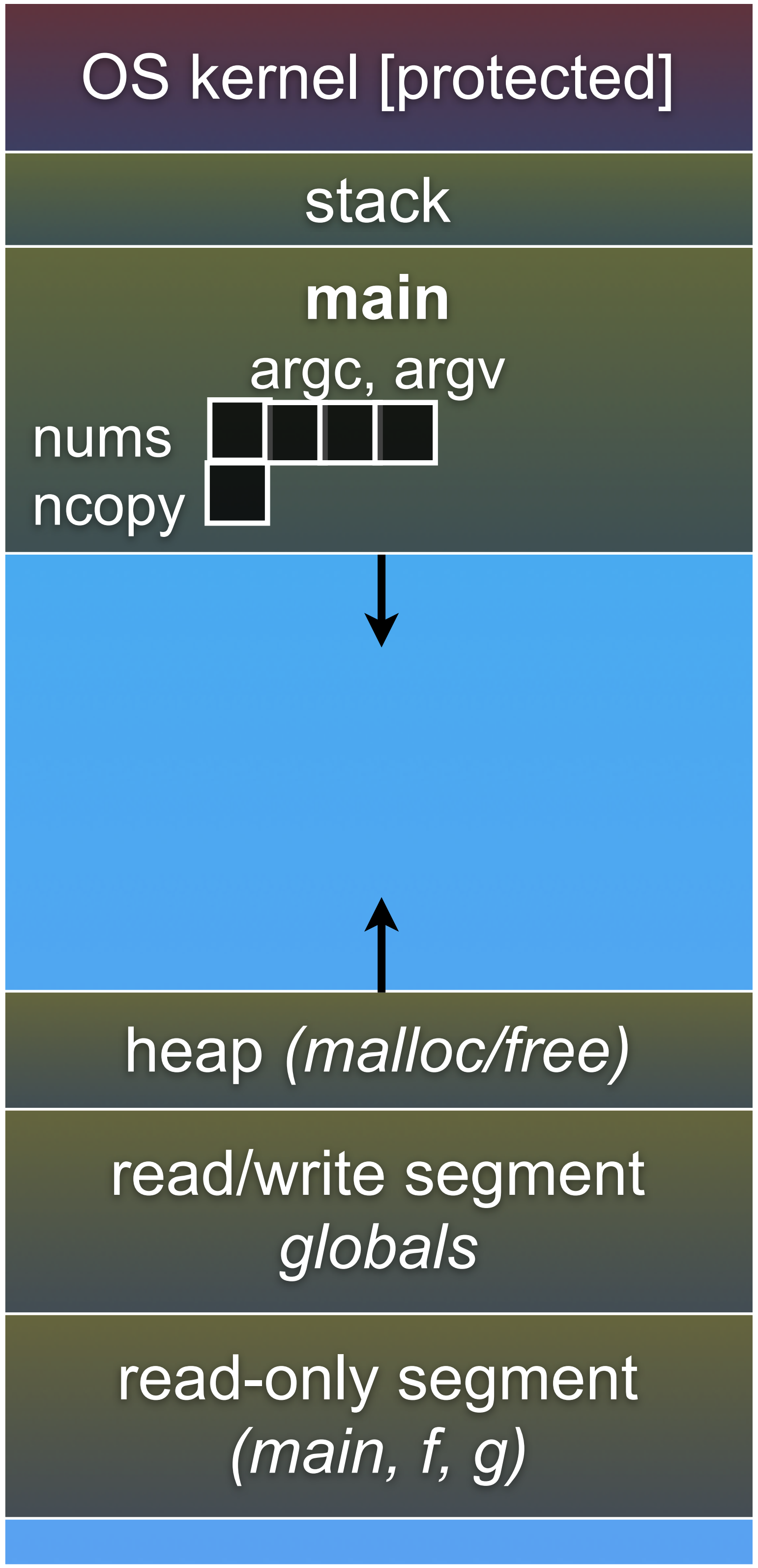

```c
#include <stdlib.h>

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;
}
```



Heap + stack

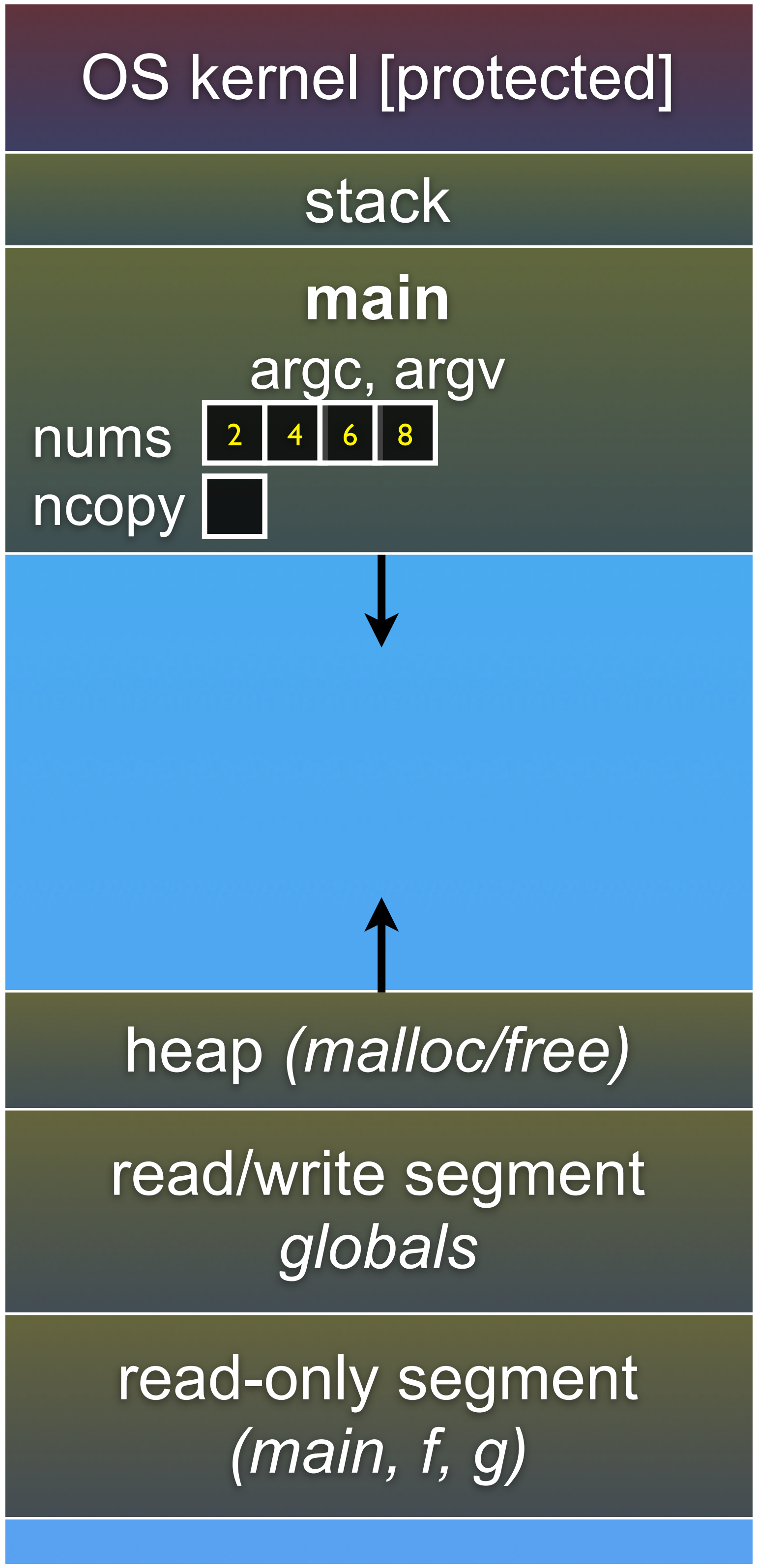

```c
#include <stdlib.h>

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;
}
```



Heap + stack

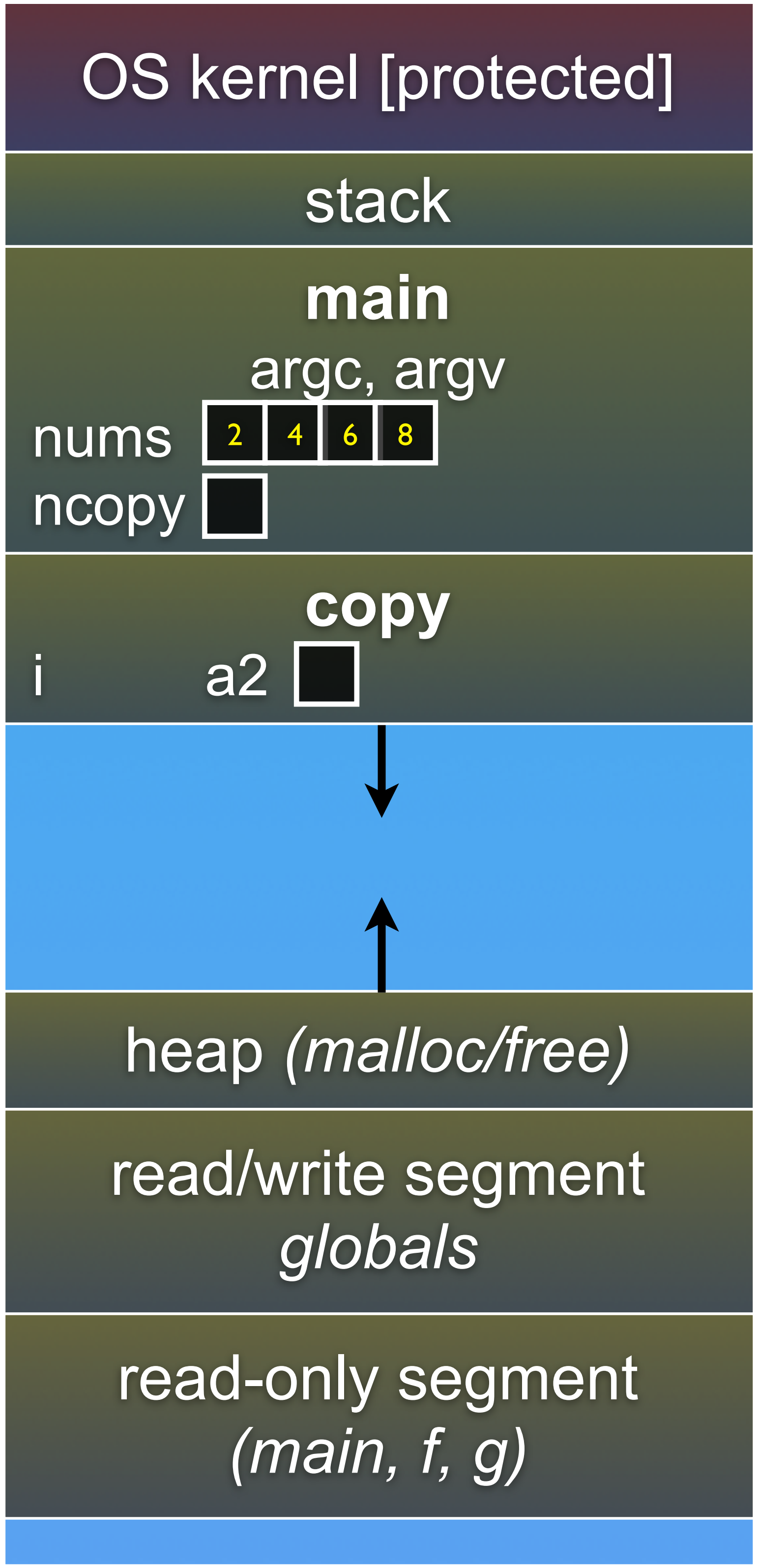
```
#include <stdlib.h>

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;
}
```



Heap + stack

UNIVERSITY OF OREGON

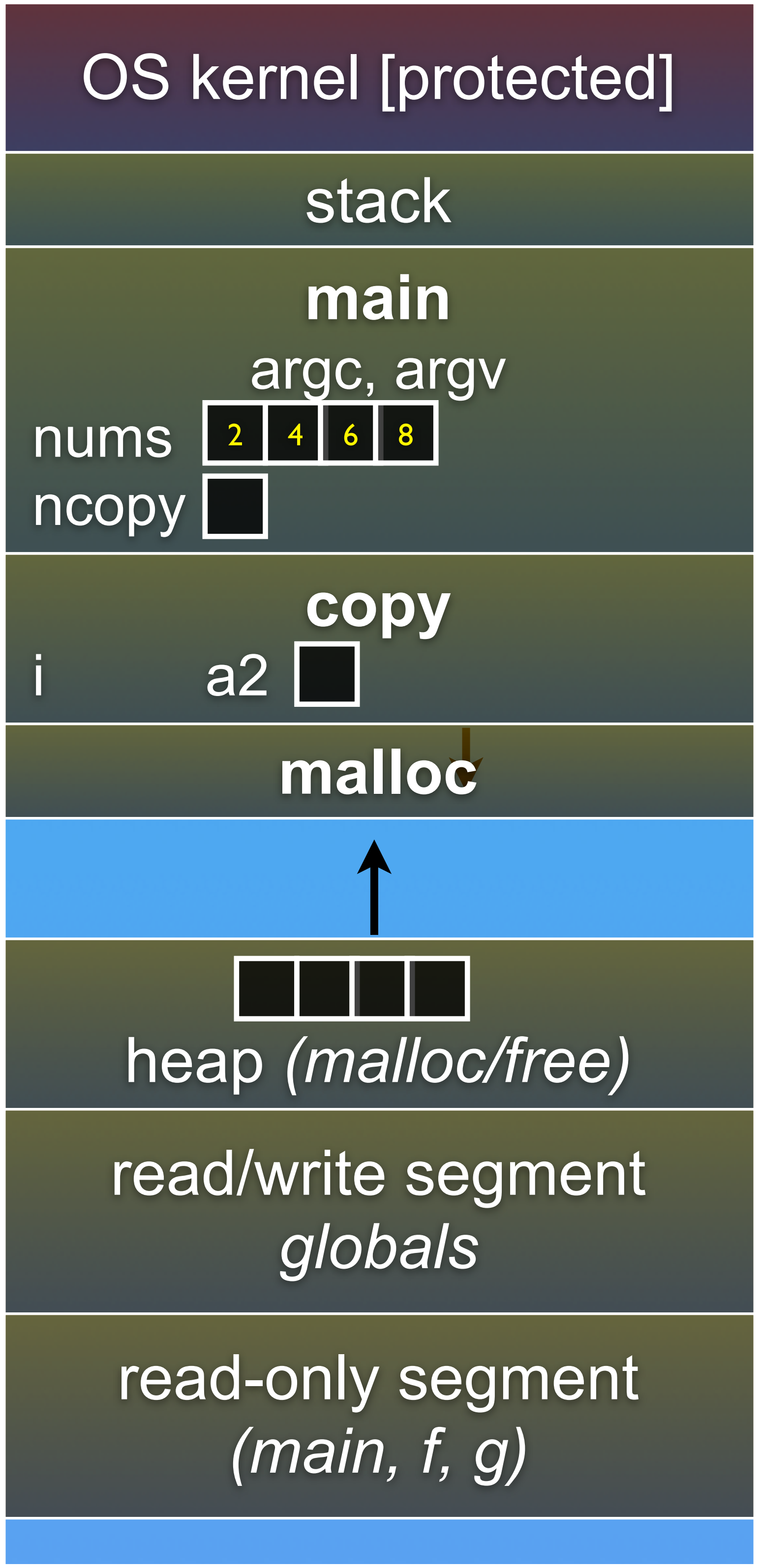
```c
#include <stdlib.h>

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;
}
```



Heap + stack

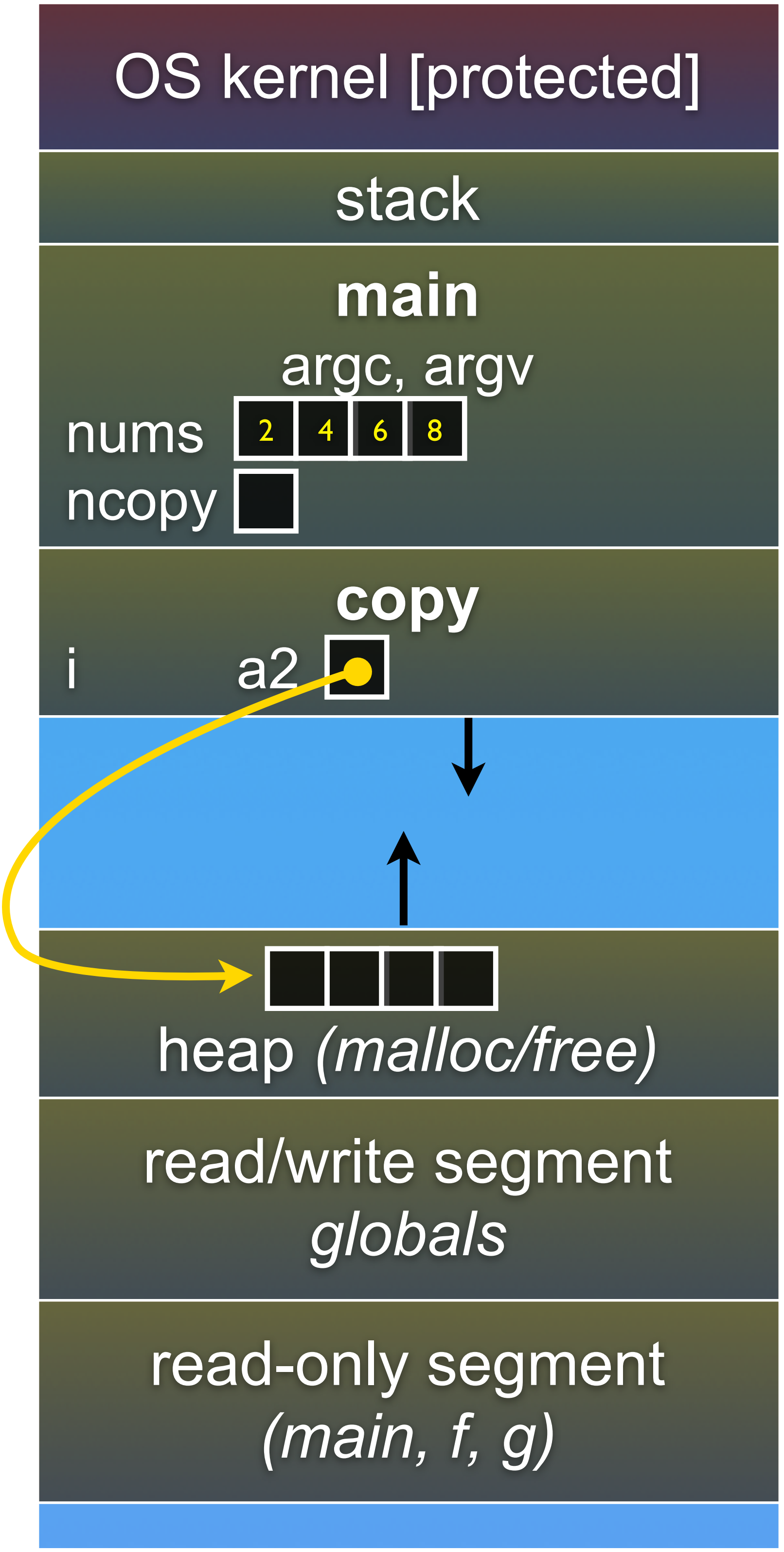
```
#include <stdlib.h>

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;
}
```



Heap + stack

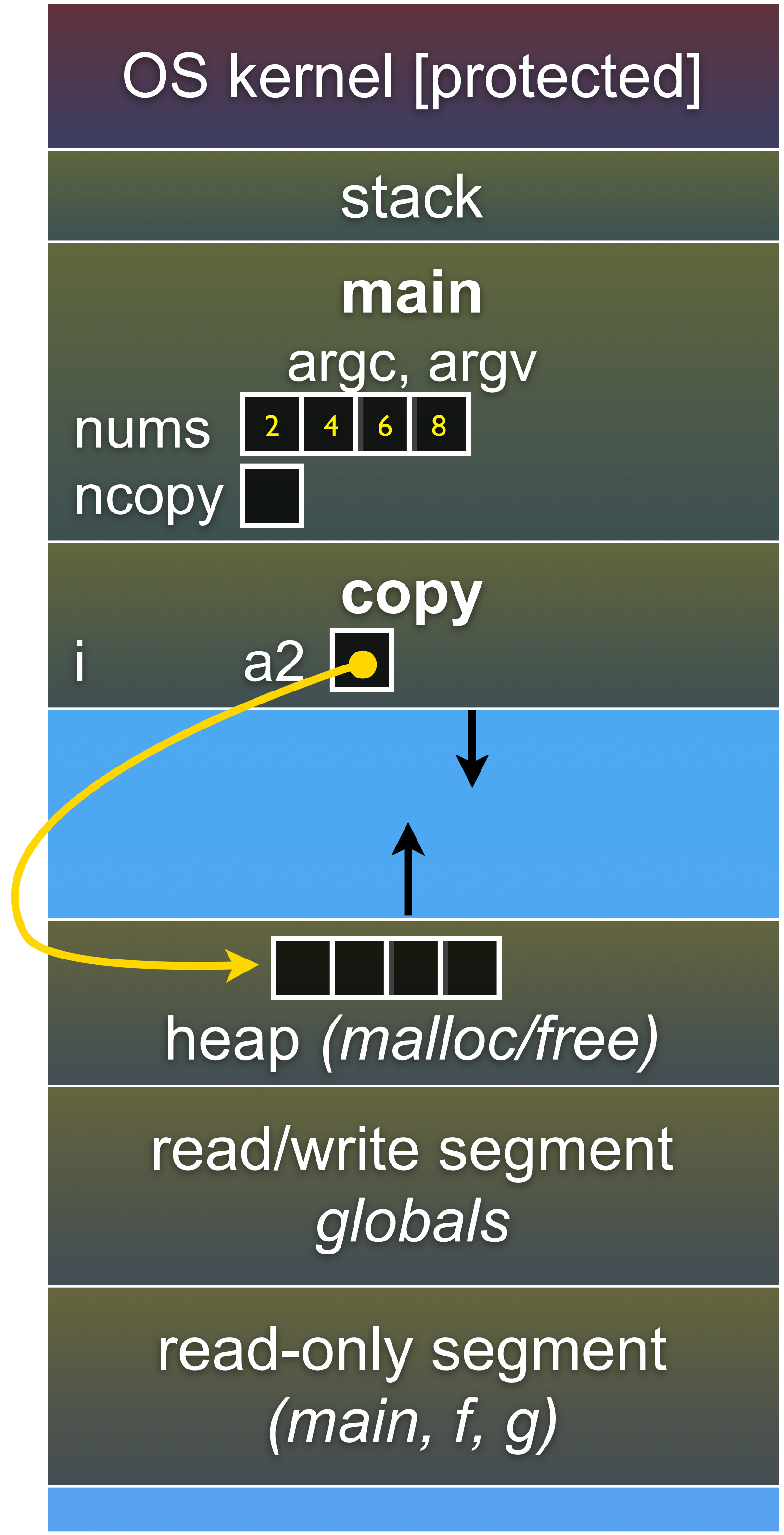
```
#include <stdlib.h>

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;
}
```



Heap + stack

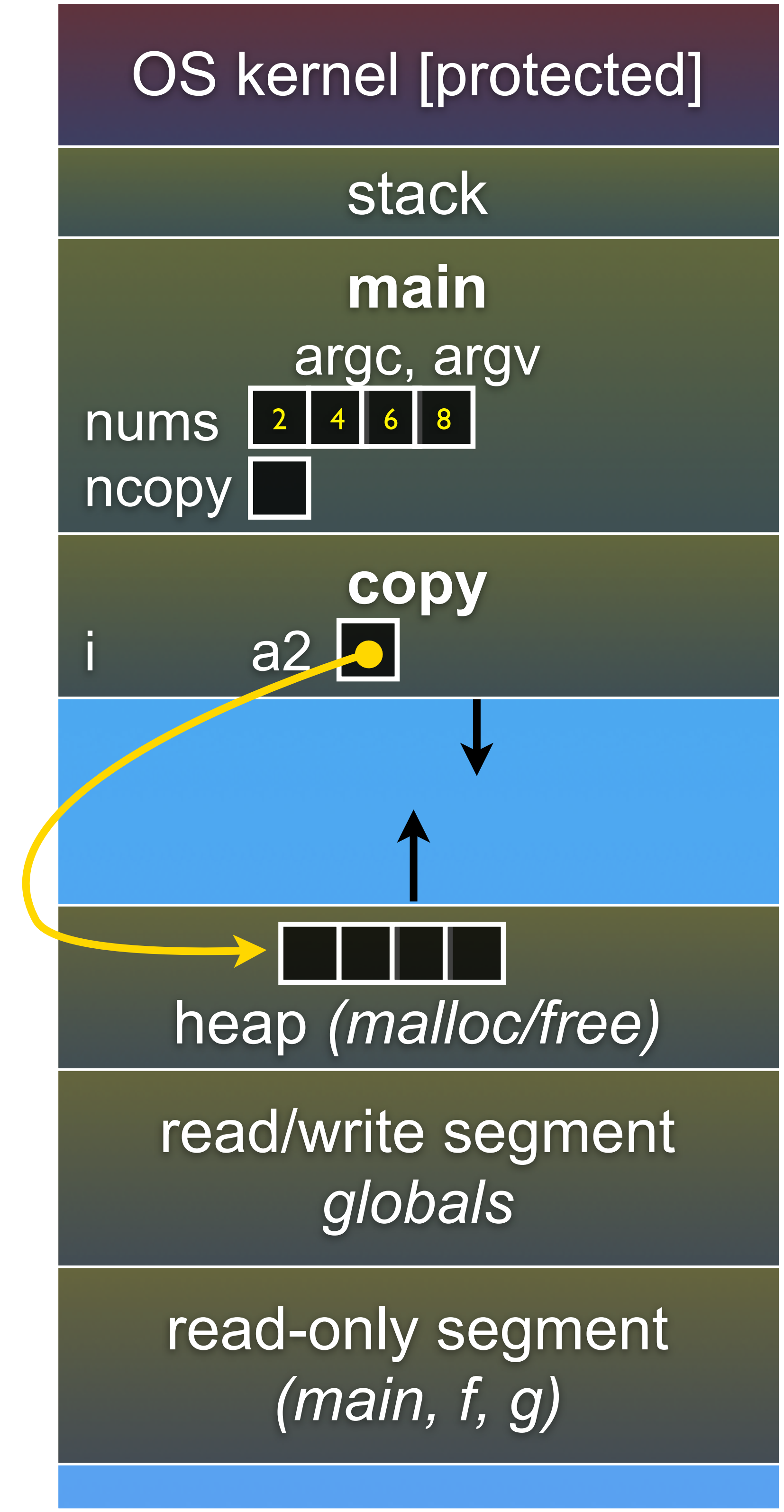
```c
#include <stdlib.h>

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;
}
```



Heap + stack

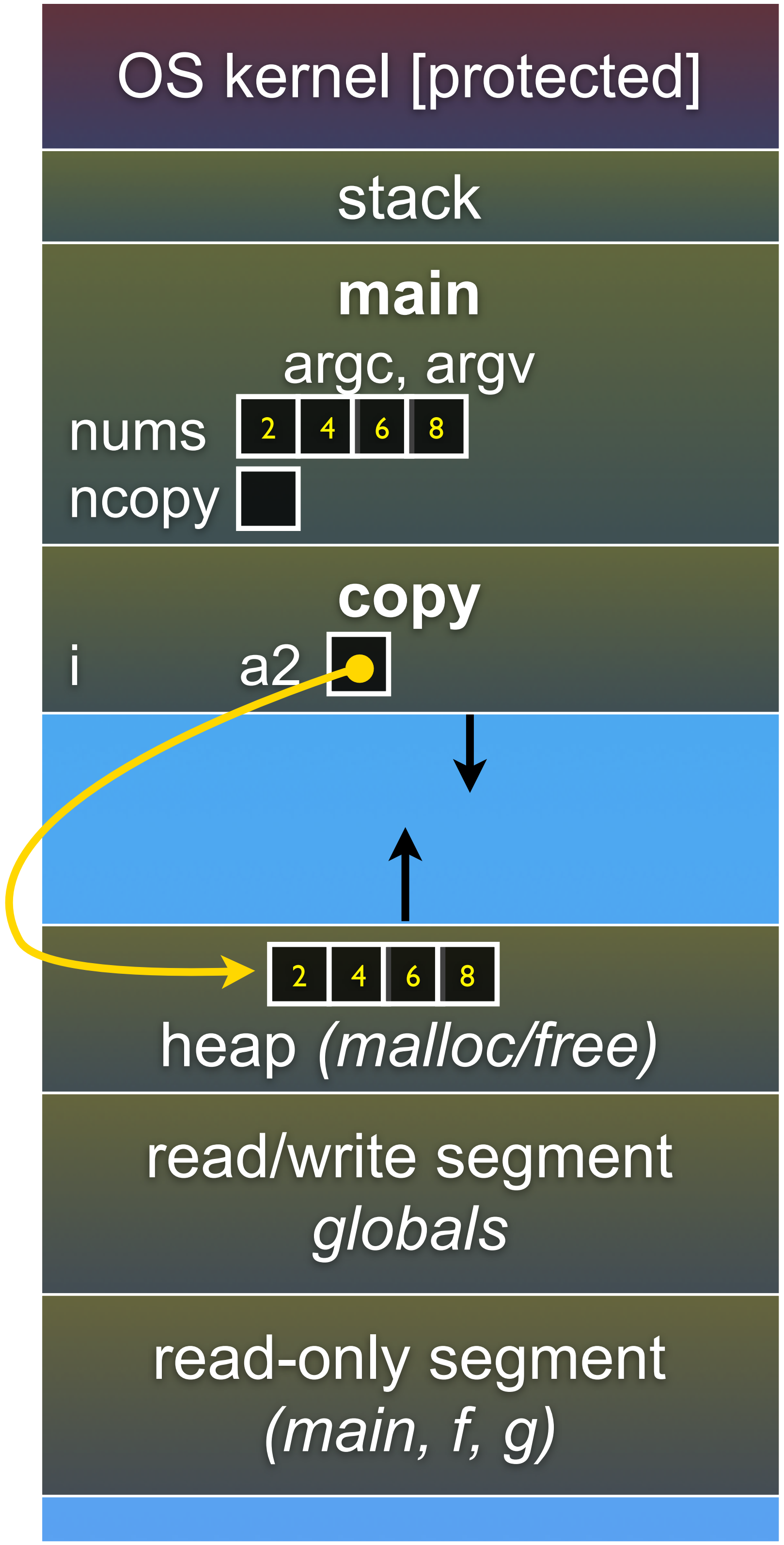
```c
#include <stdlib.h>

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;
}
```



Heap + stack

UNIVERSITY OF OREGON

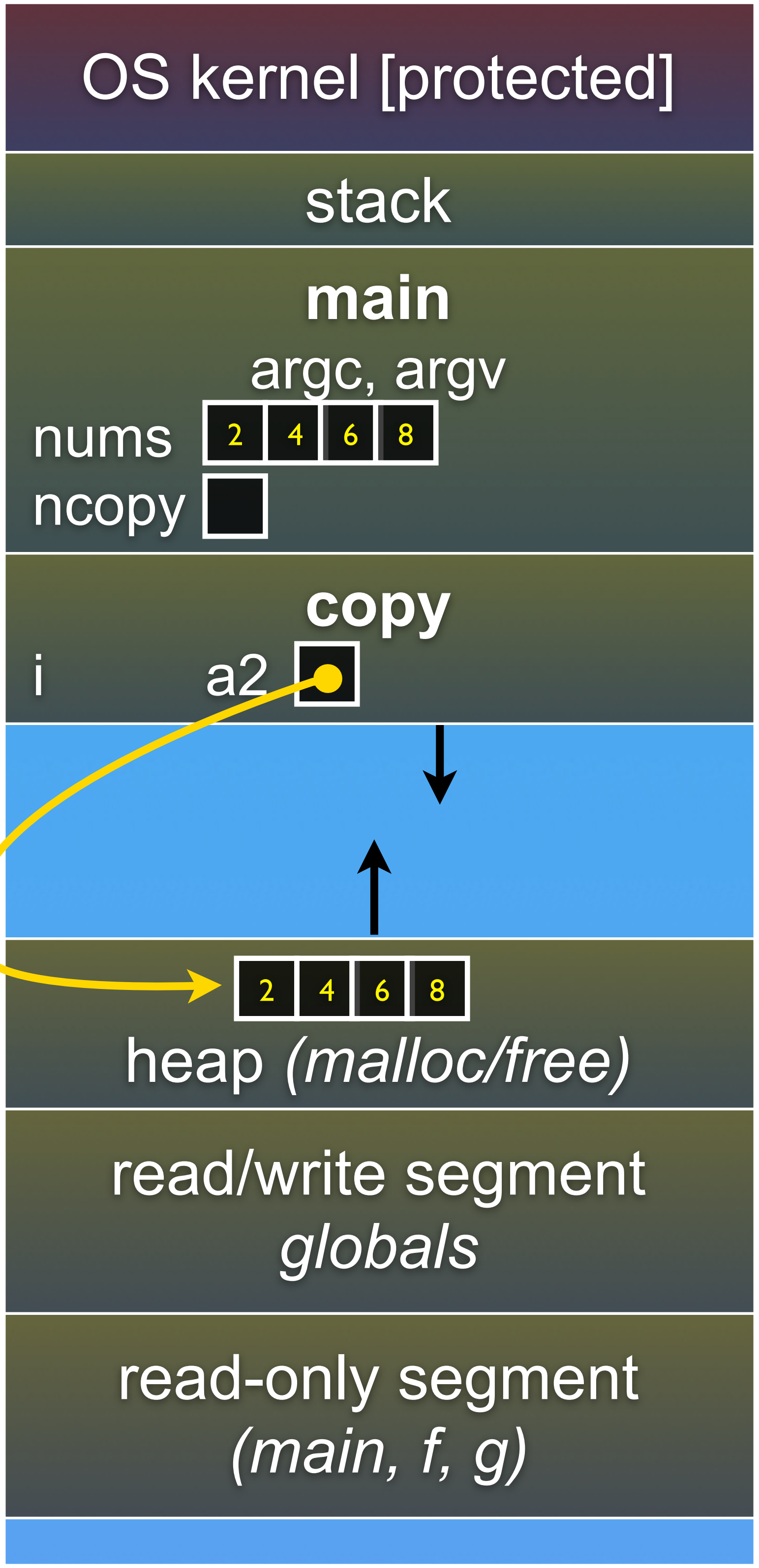
```
#include <stdlib.h>

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;
}
```



Heap + stack

UNIVERSITY OF OREGON

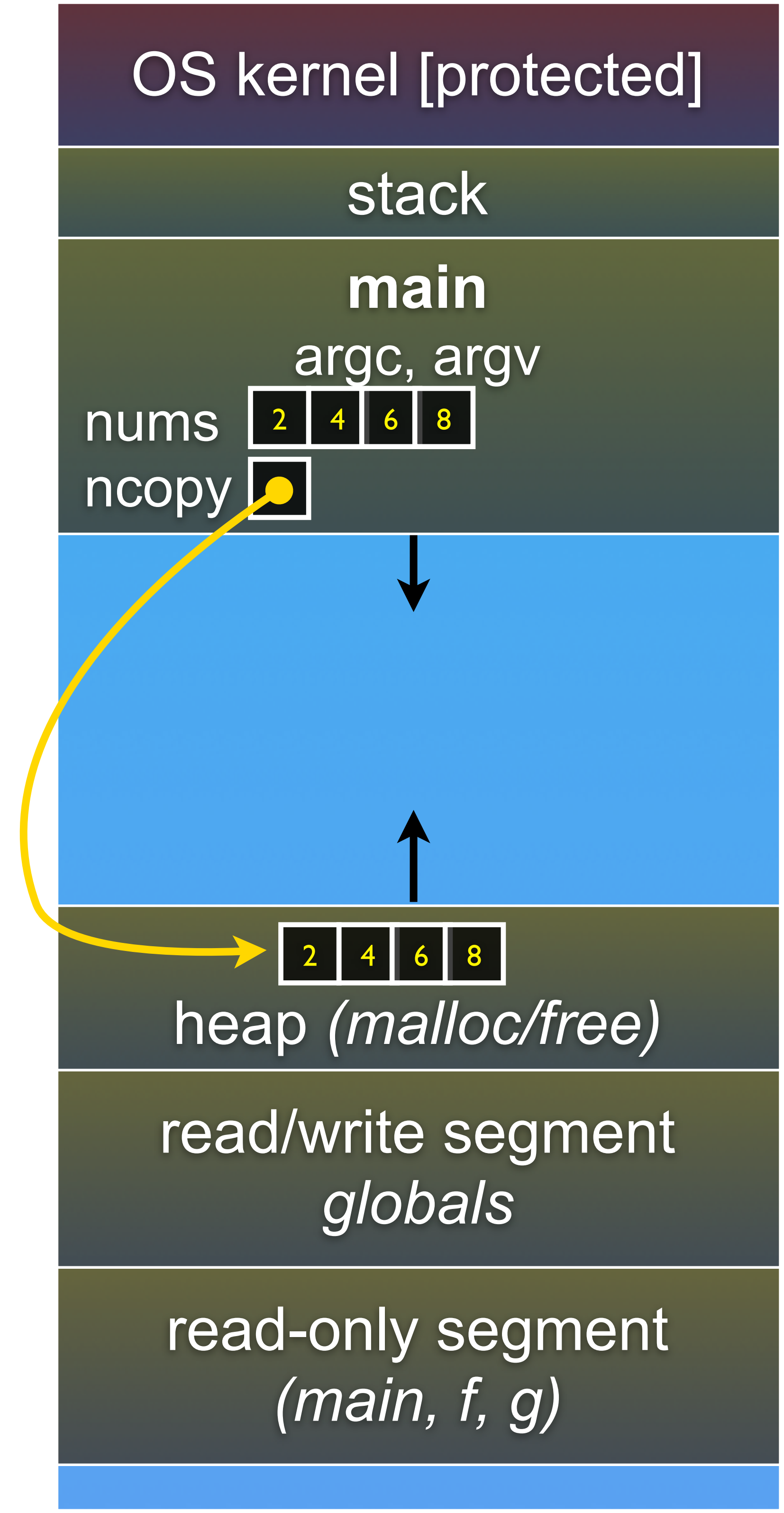
```
#include <stdlib.h>

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;
}
```



Heap + stack

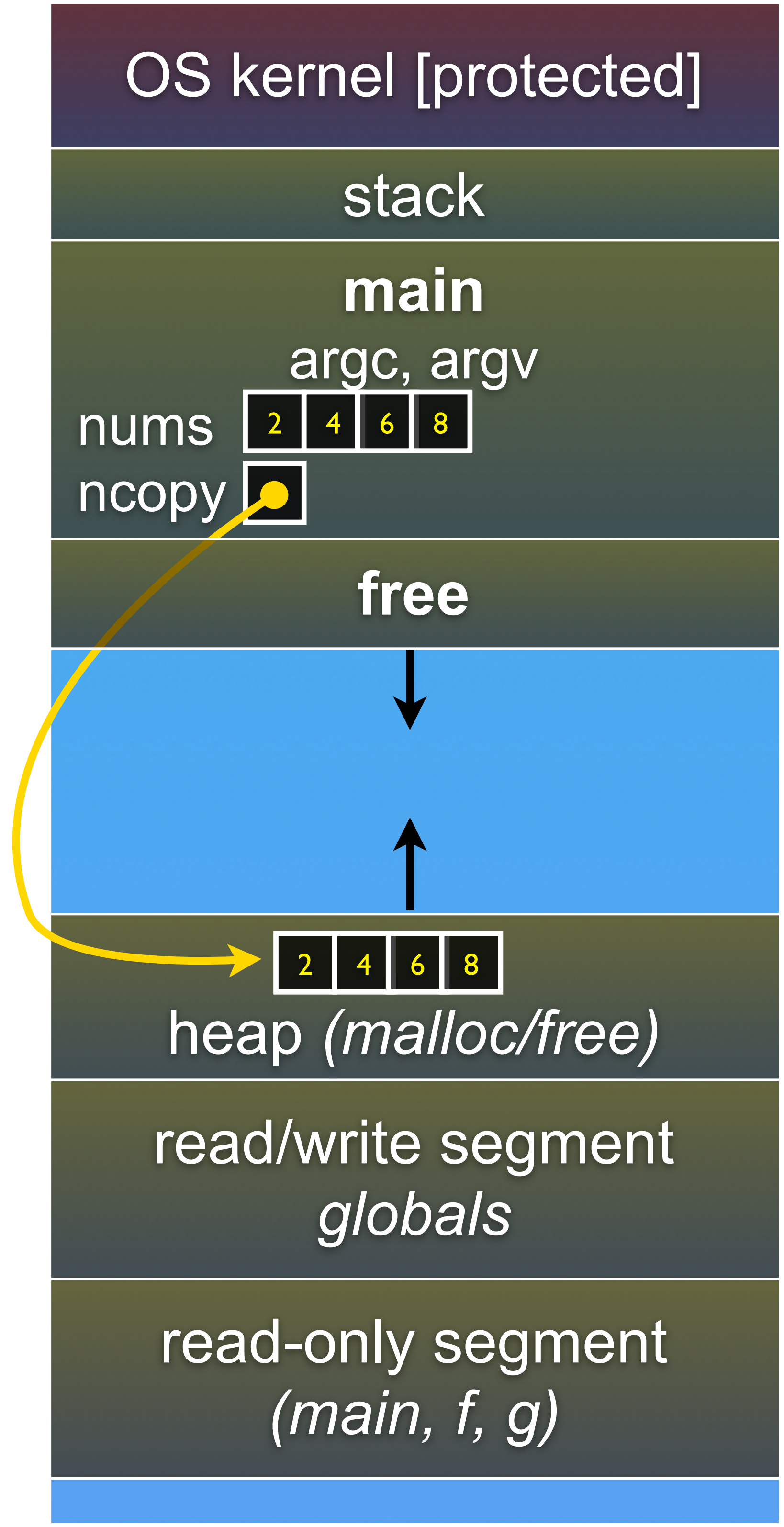

```c
#include <stdlib.h>

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;
}
```



Heap + stack

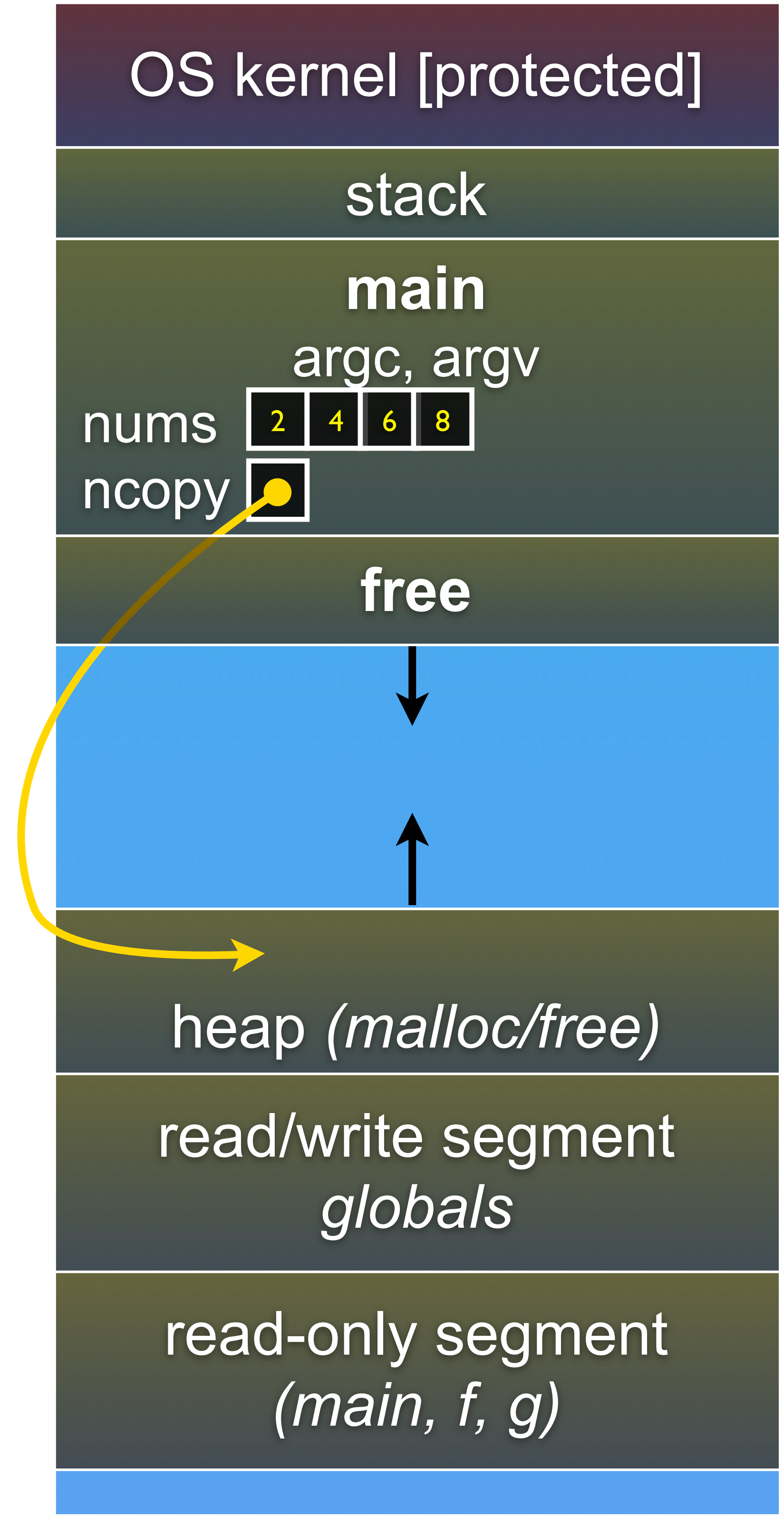
```
#include <stdlib.h>

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;
}
```



Heap + stack

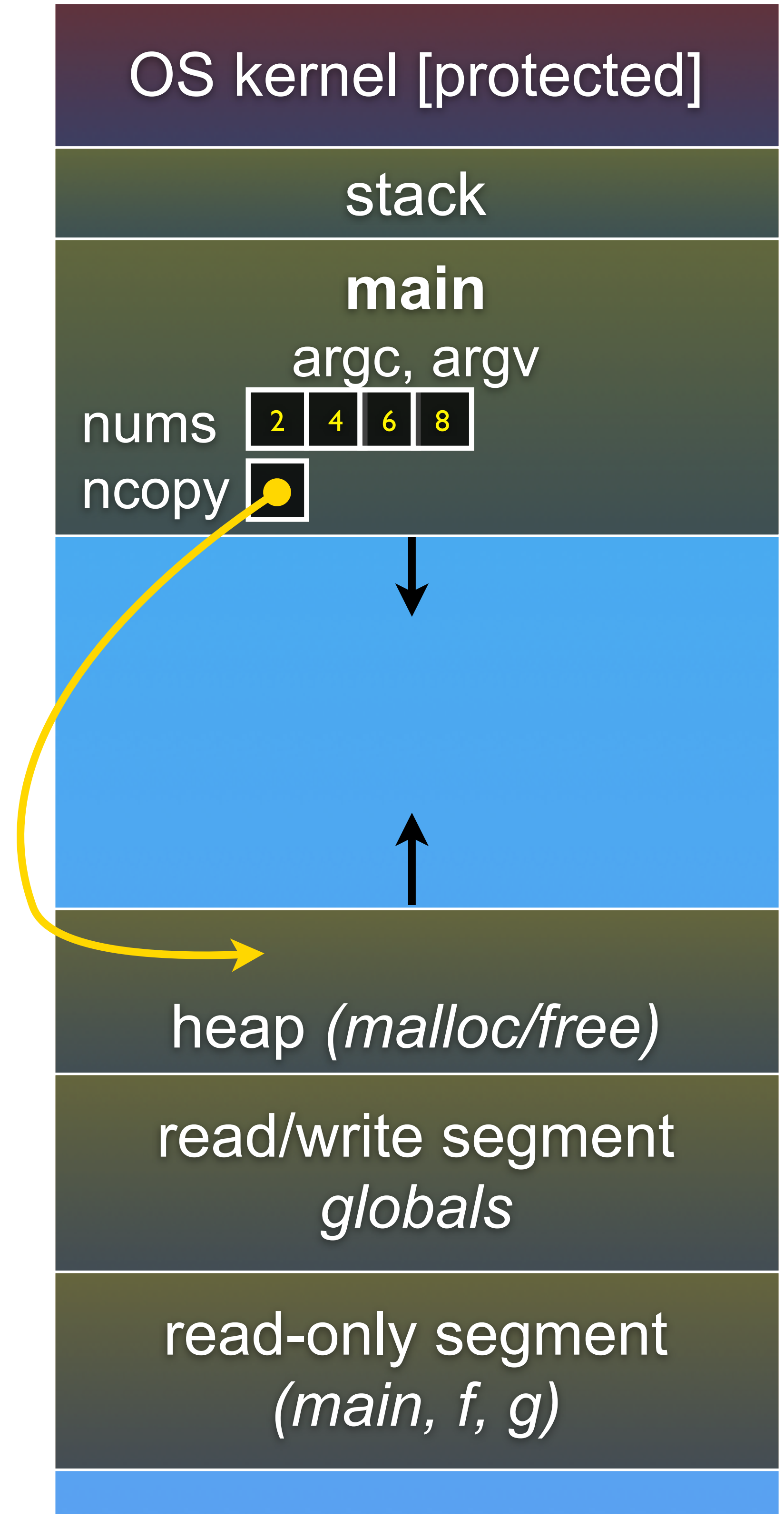
```c
#include <stdlib.h>

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

- Parent process create children processes,
  - which, in turn create other processes, forming a 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

Process Creation

UNIVERSITY OF OREGON

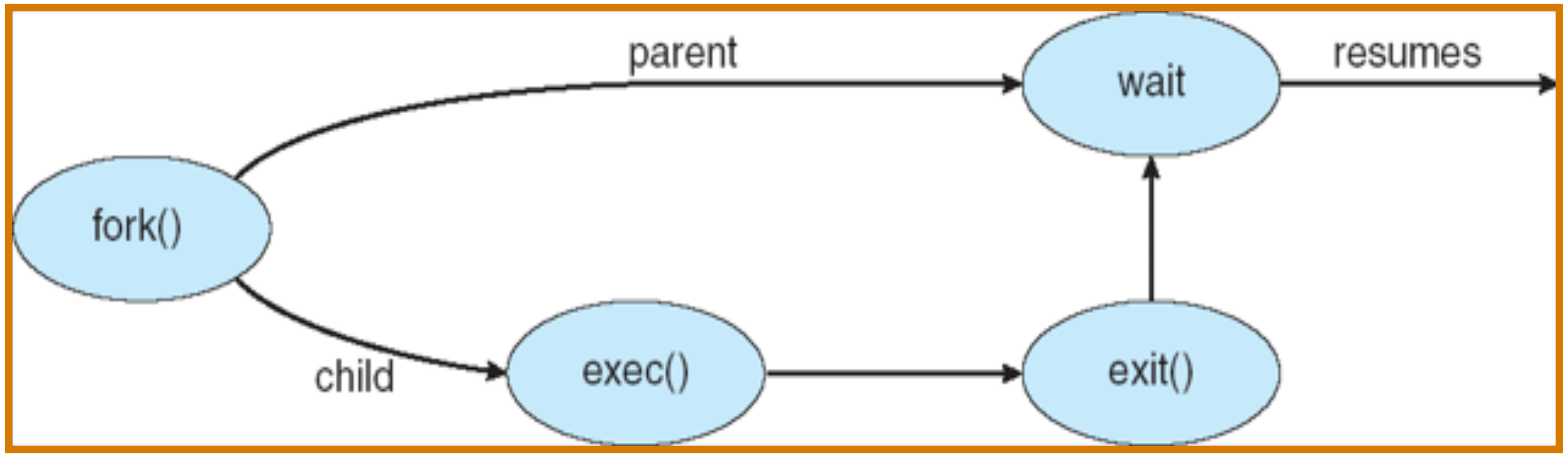
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork` system call creates new process
  - `exec` system call used after a fork to replace the process's memory space with a new program

Process Creation

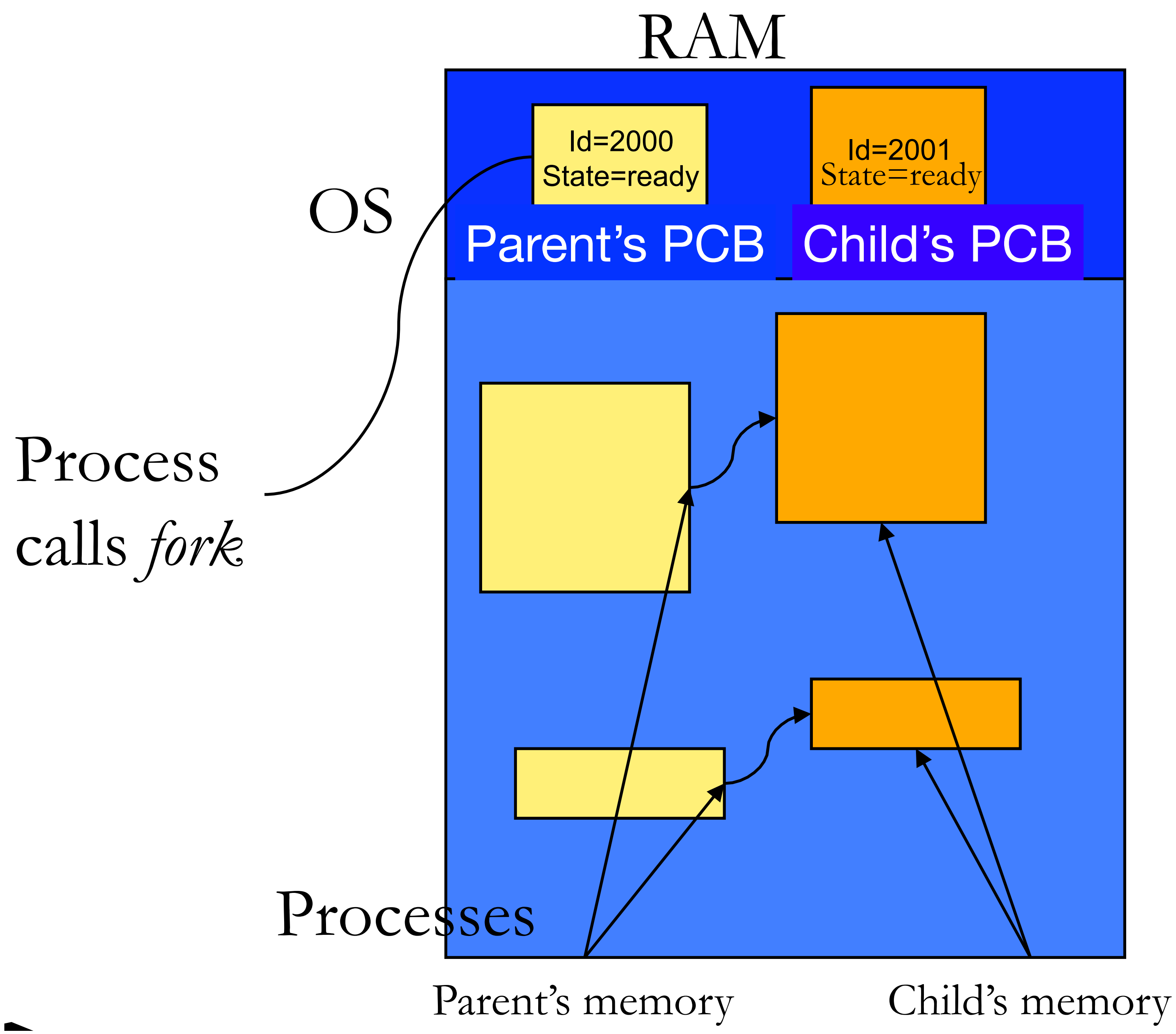
UNIVERSITY OF OREGON

- What happens?
  - New process object in kernel
    - Build process data structures
  - Allocate address space (abstract resource)
    - Later, allocate memory (physical resource)
  - Add to execution queue
    - Runnable?

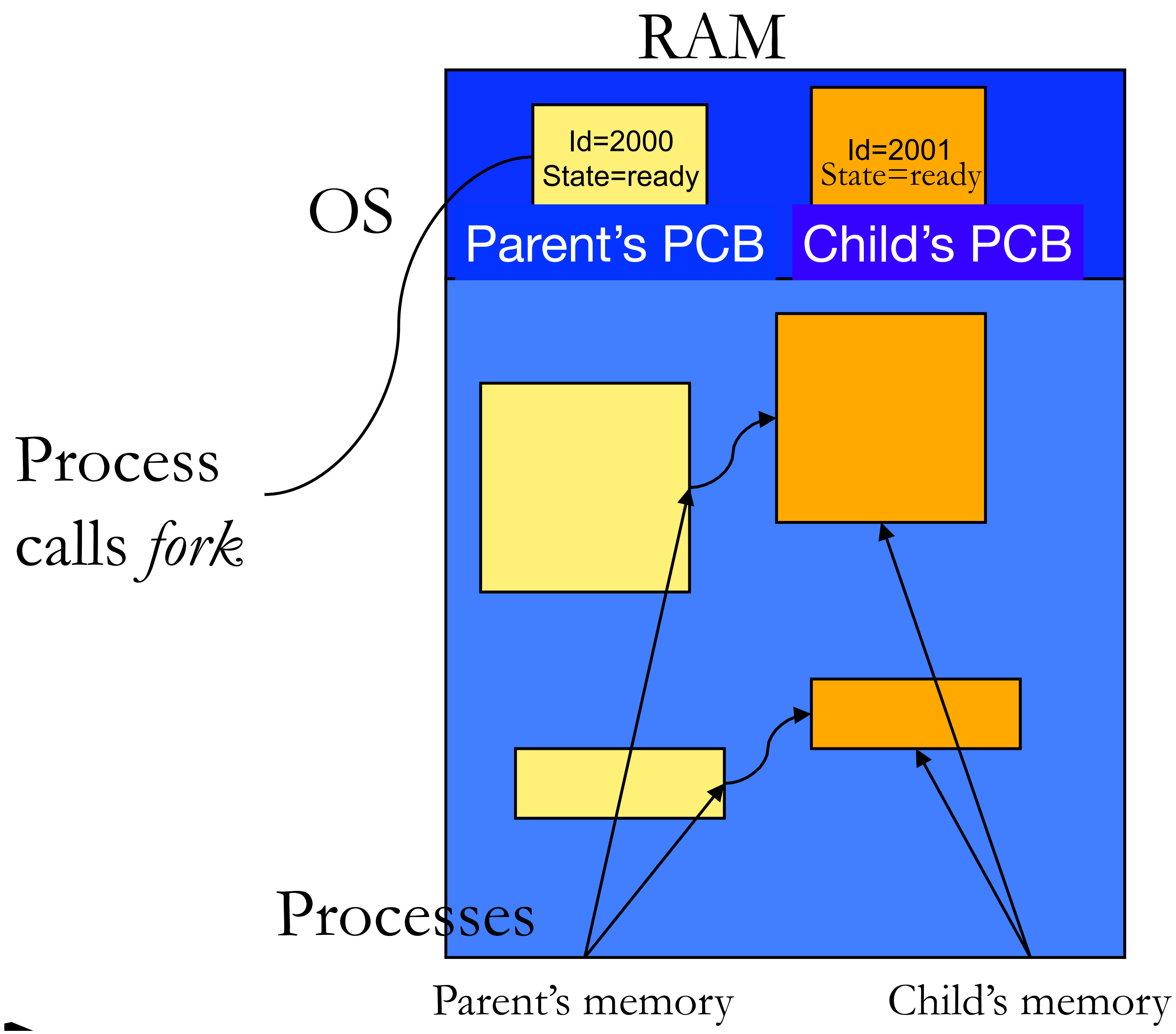
Process Creation



Process Layout



Process Layout

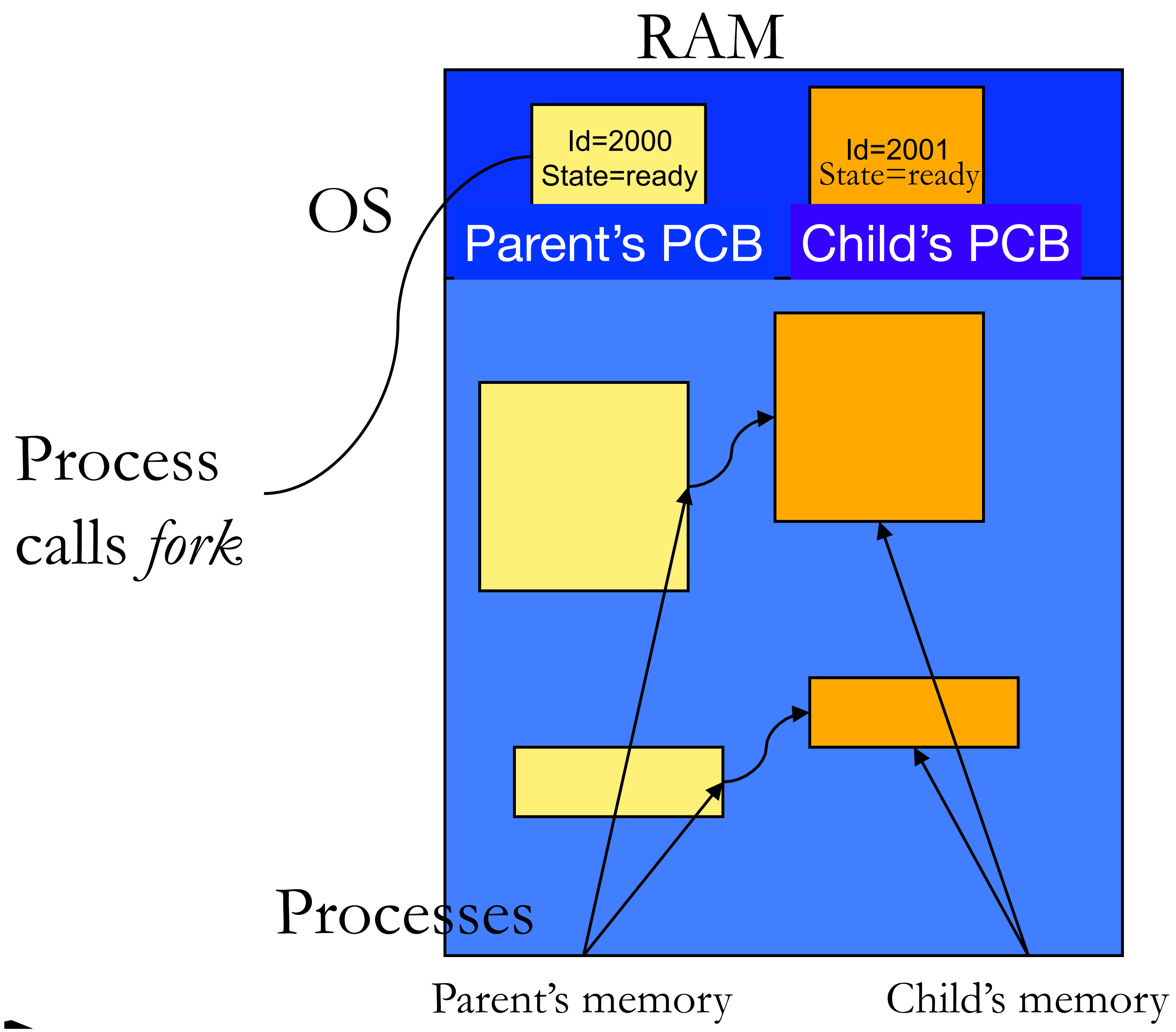


1. PCB with new Id created

2. Memory allocated for child

Initialized by copying over from the parent

Process Layout



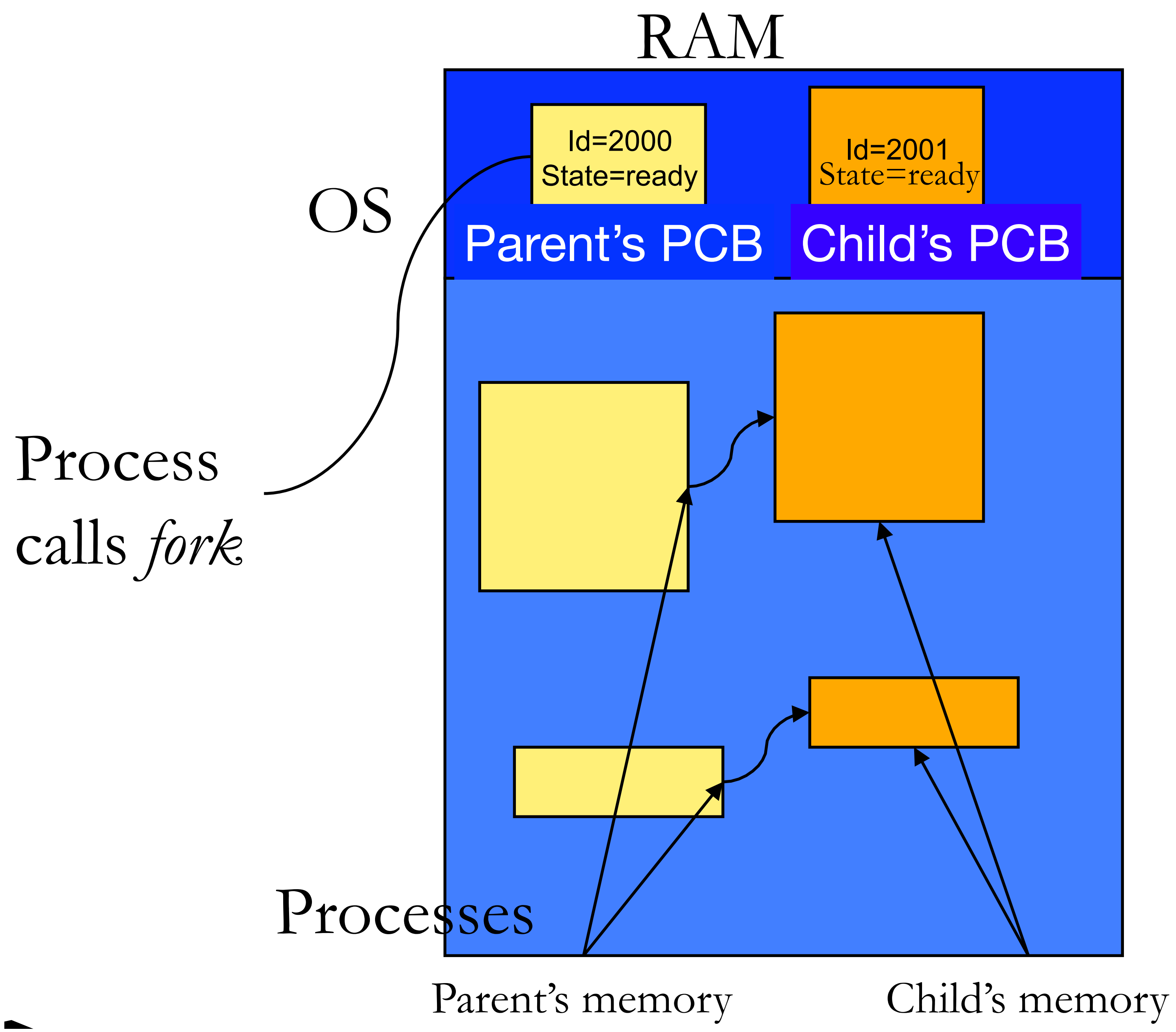
1. PCB with new Id created
2. Memory allocated for child

Initialized by copying over from the parent

3. If parent had called **wait**, it is moved to a waiting queue

Process Layout

UNIVERSITY OF OREGON



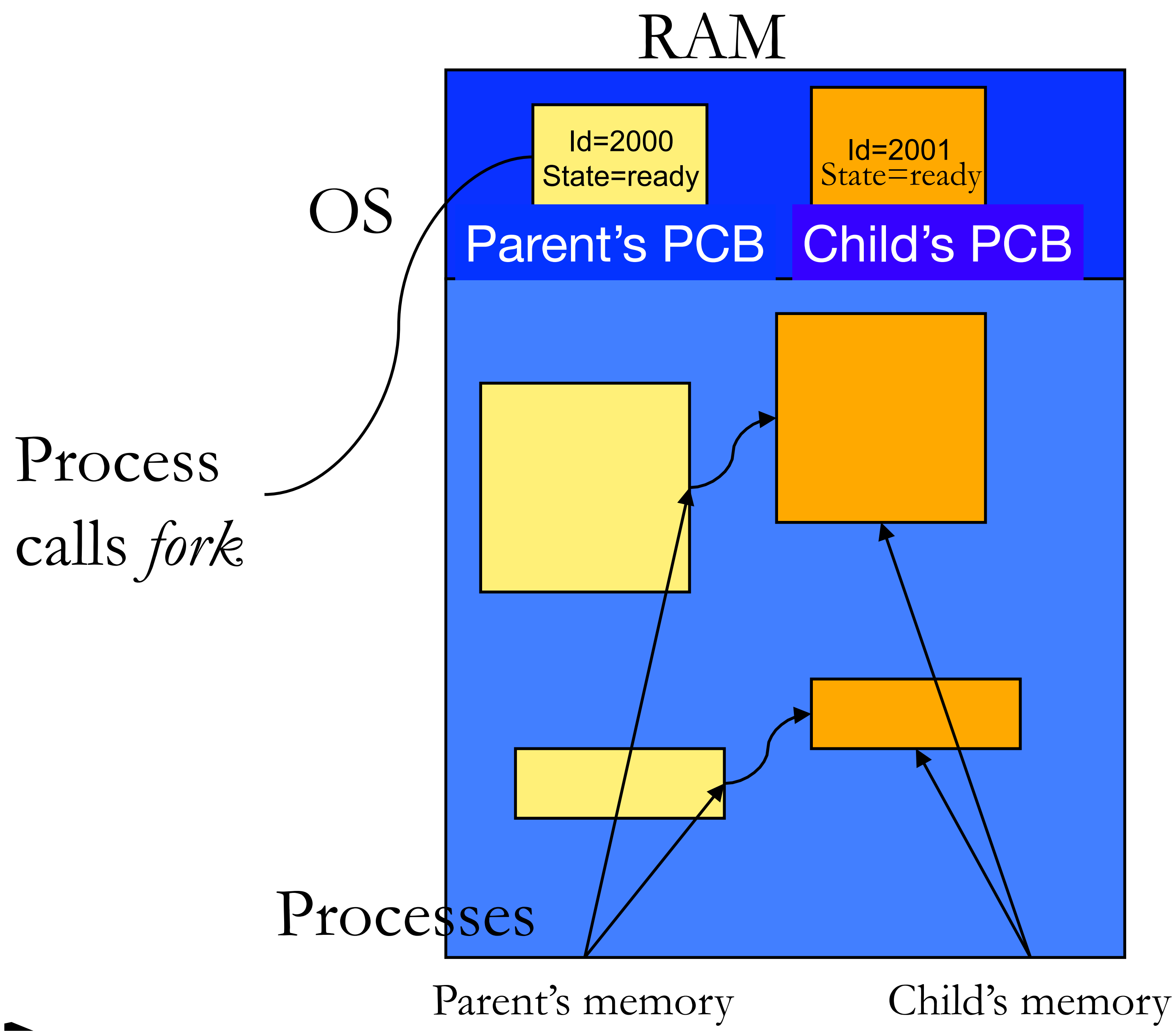
1. PCB with new Id created
2. Memory allocated for child

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 overwritten with new code & data

Process Layout

UNIVERSITY OF OREGON



1. PCB with new Id 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 overwritten with new code & data

5. Child added to ready queue, all set to go now!

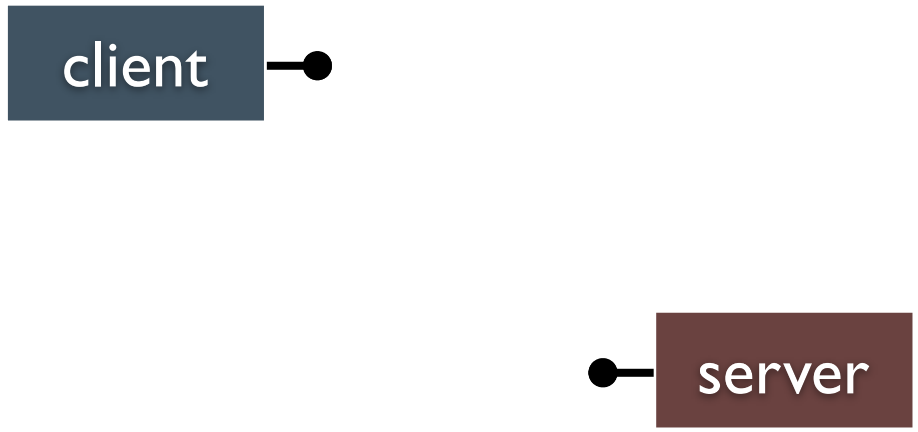
C Program Forking Separate Process

UNIVERSITY OF OREGON

```
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);
  }
  else { /* parent process */
   /* parent will wait for the child to
  complete */
   wait (NULL);
   printf ("Child Complete");
   exit(0);
  }
}
```

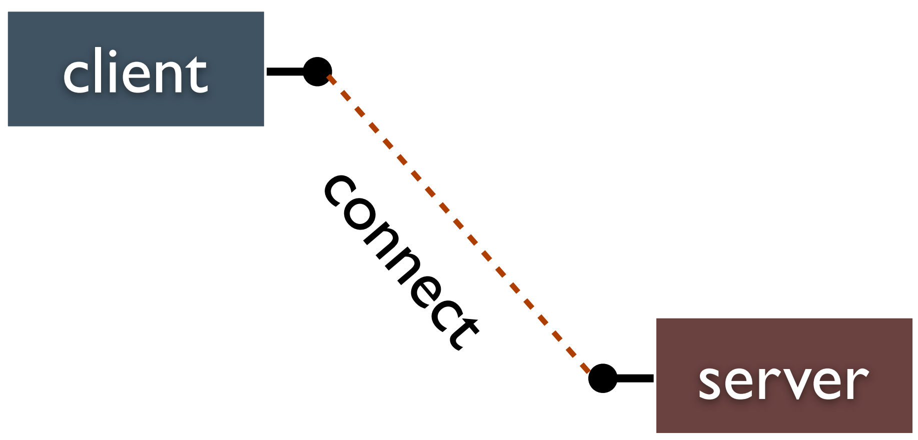
Graphically

Graphically

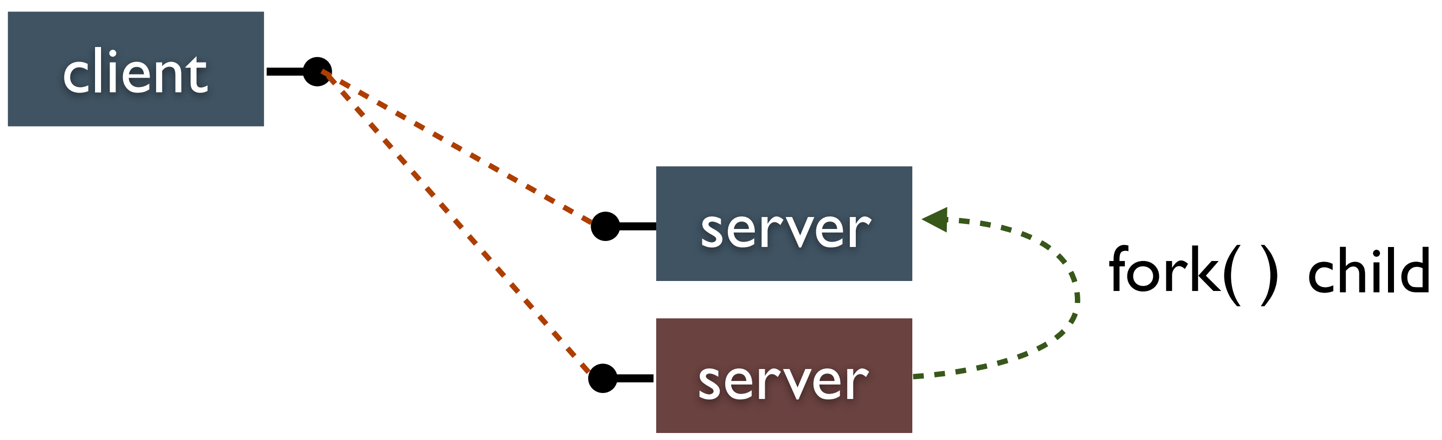


Graphically

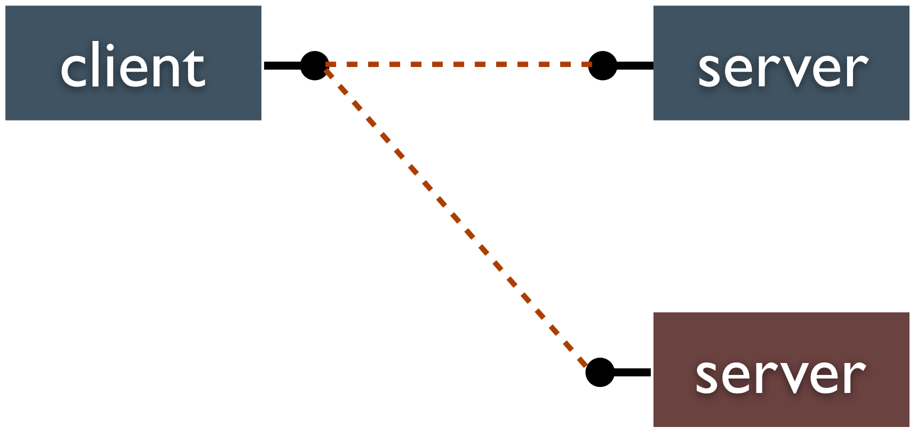
UNIVERSITY OF OREGON



Graphically

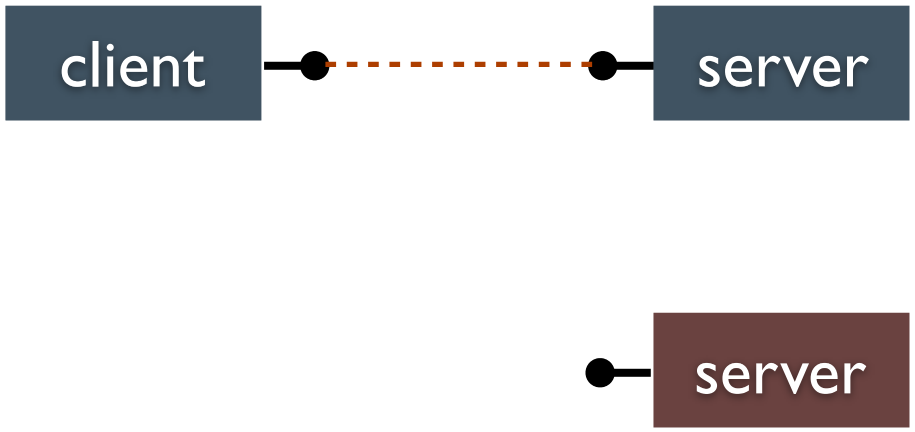


Graphically



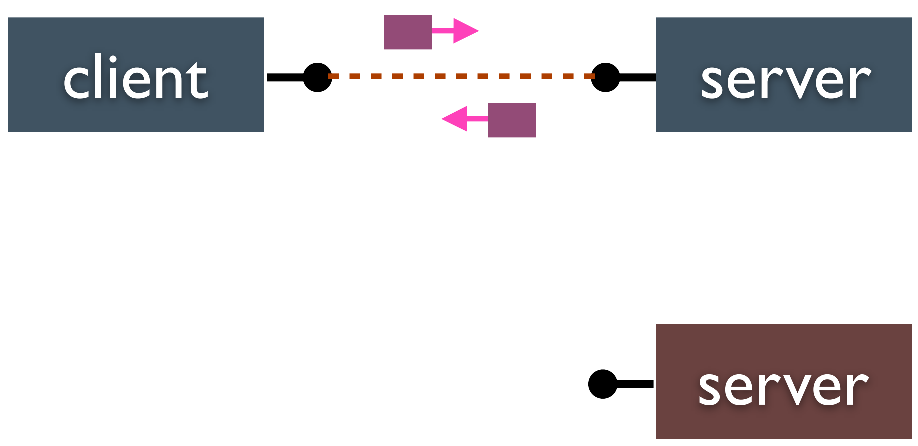
child exit( )'s / parent wait( )'s

Graphically

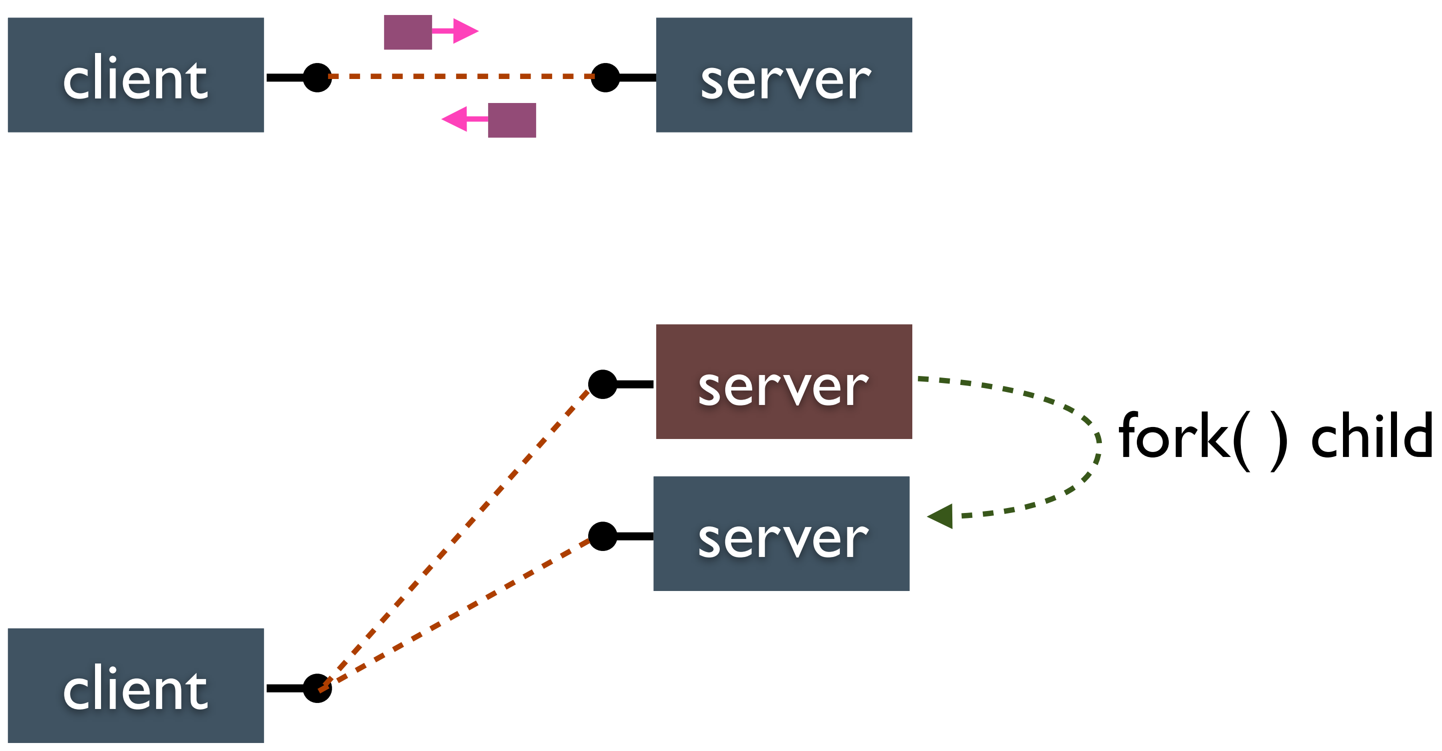


parent closes its client connection

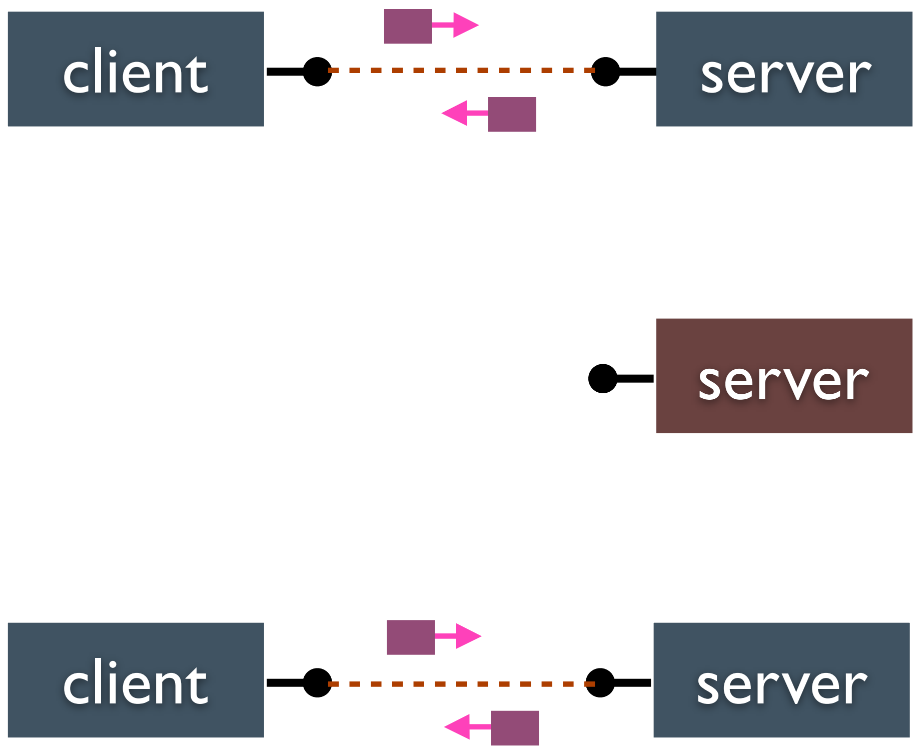
Graphically



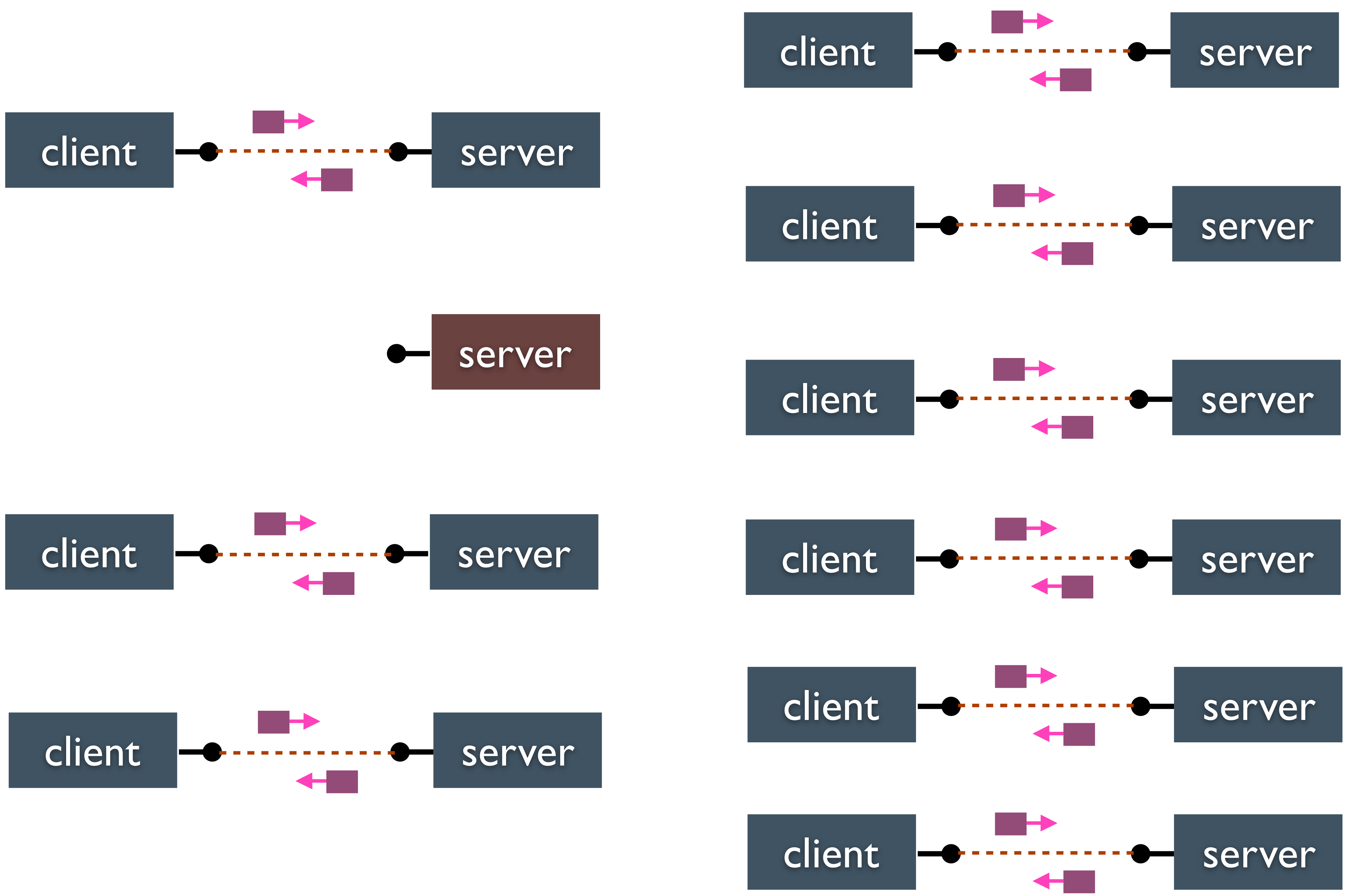
Graphically



Graphically



Graphically



Program Creation

UNIVERSITY OF OREGON

- Design Choices
  - Resource Sharing
    - What resources of parent should the child share?
    - What about after `exec`?
  - Execution
    - Should parent wait for child?
  - What is the relationship between parent and child?
    - Hierarchical or grouped or …?

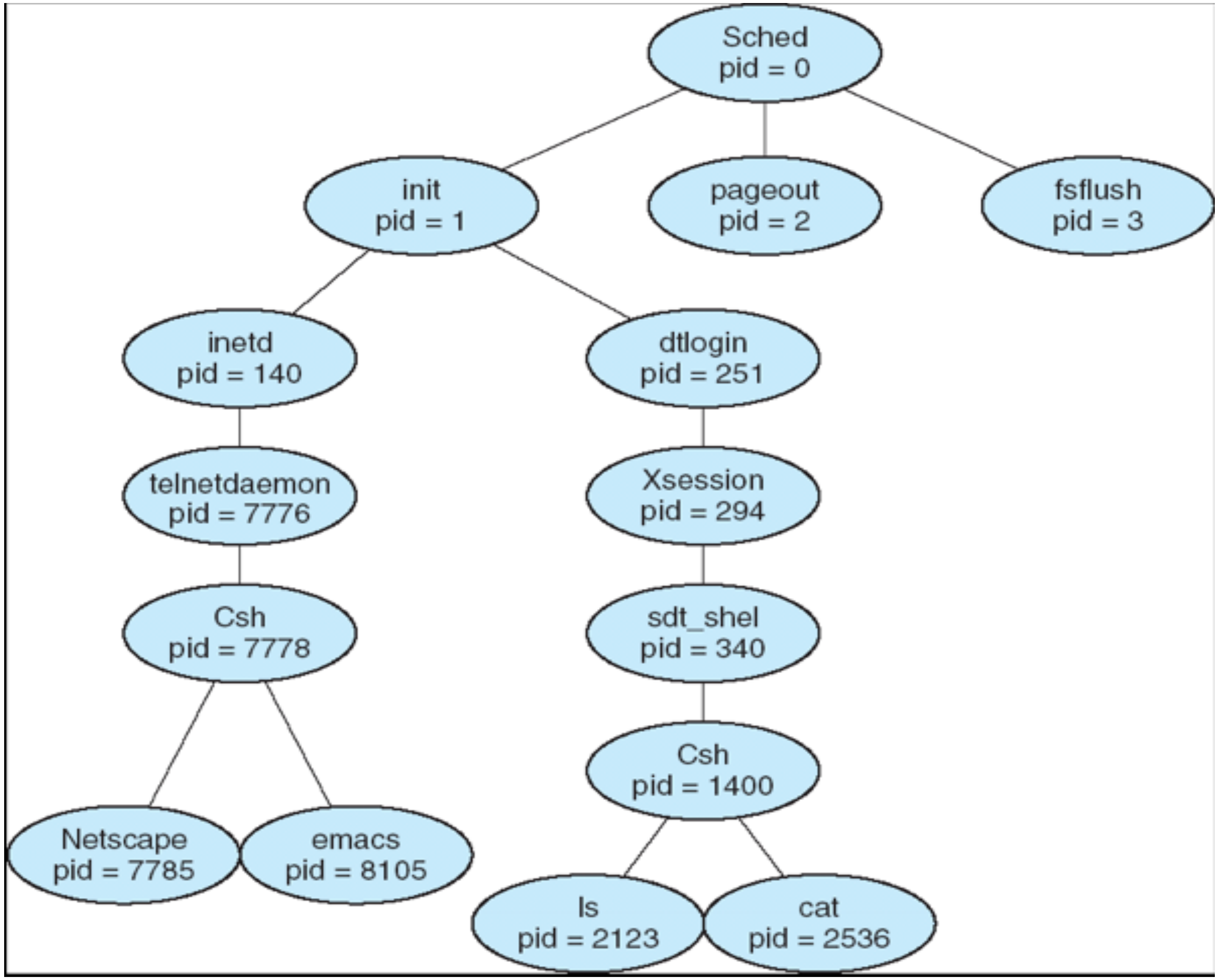
Program Creation

UNIVERSITY OF OREGON

- `fork` -- copy address space and all threads
- `forkl` -- copy address space and only calling thread
- `vfork` -- do not copy address space; shared between parent and child
- `exec` -- load new program; replace address space
    - Some resources may be transferred (open file descriptors)
    - Specified by arguments

A tree of processes on a typical system

UNIVERSITY OF OREGON



Process Termination

UNIVERSITY OF OREGON

- 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

Executing a Process

- What to execute?
  - Register that stores the program counter
    - Next instruction to be executed
- Registers store state of execution in CPU
  - Stack pointer
  - Data registers
- Thread of execution
  - Has its own stack

Executing a Process

UNIVERSITY OF OREGON

- Thread executes over 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 (oops)
  - **System call** (switch to kernel mode)

Relocatable Memory

UNIVERSITY OF OREGON

- Mechanism that enables the OS to place a program in an arbitrary location in memory
  - Gives the programmer the impression that they own the processor
- Program is loaded into memory at program-specific locations
  - Need virtual memory to do this
- Also, may need to share program code across processes

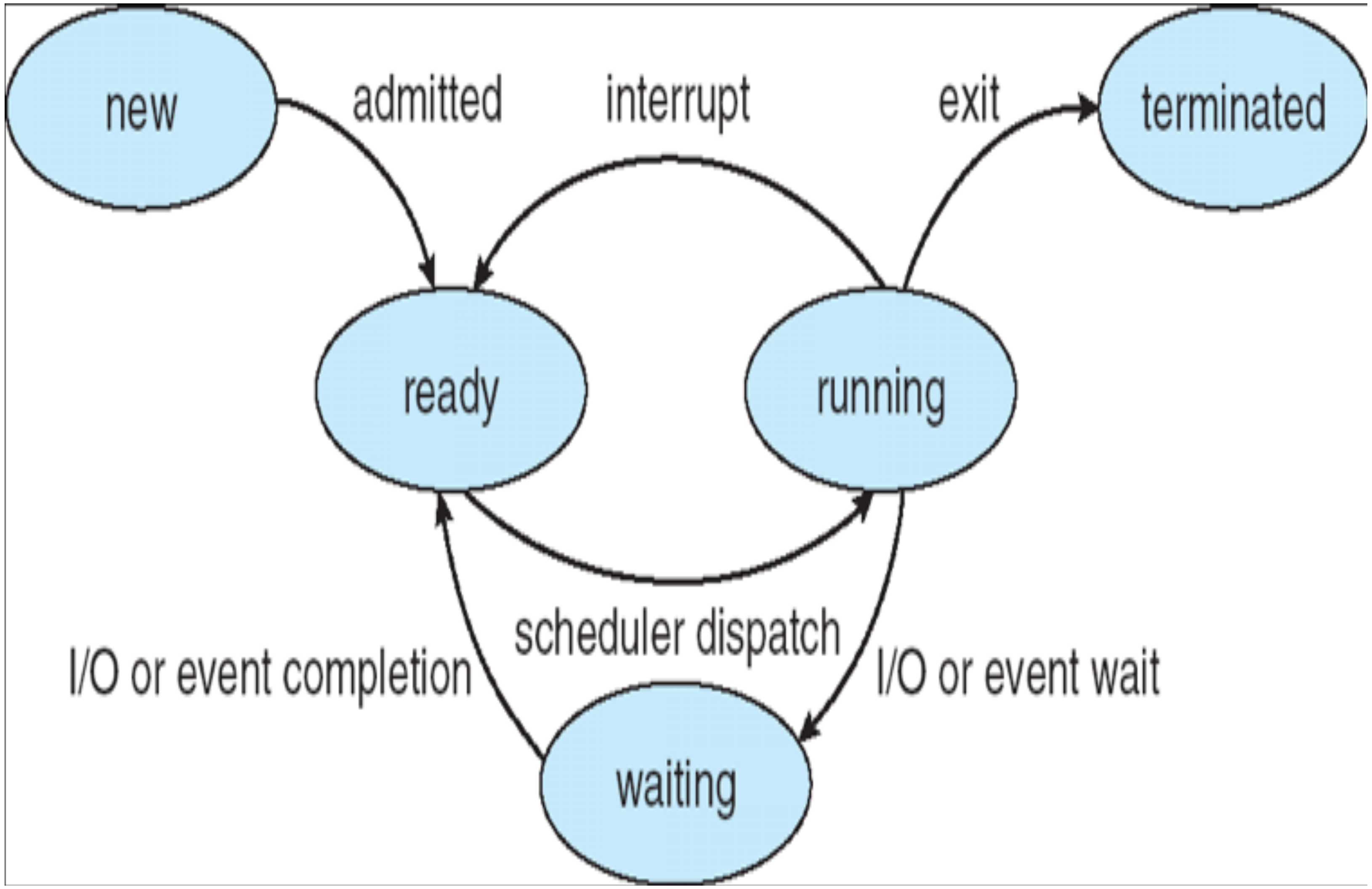
Process State

- 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

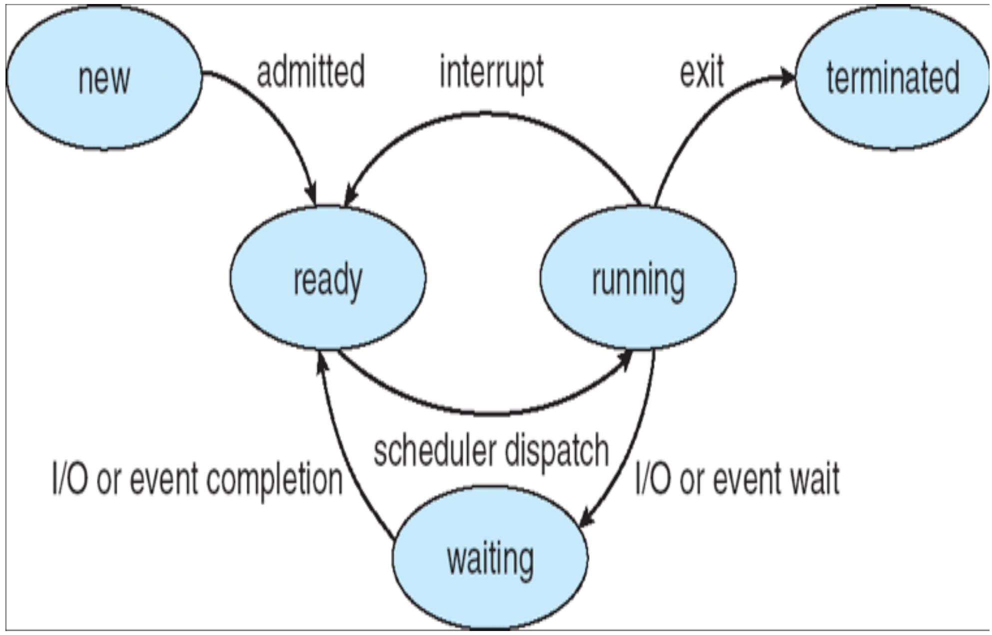
UNIVERSITY OF OREGON

- Processes transition among *execution states*



Process States

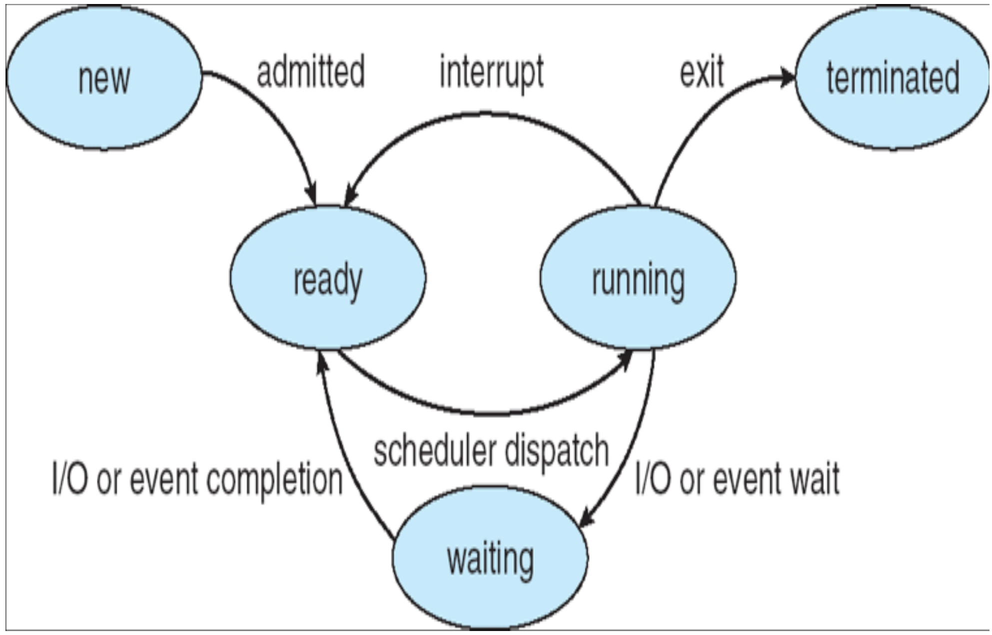
- Running
  - Running == in processor and in memory with all resources
- Ready
  - Ready == in memory with all resources, waiting for dispatch
- Waiting
  - Waiting == waiting for some event to occur



State Transitions

UNIVERSITY OF OREGON

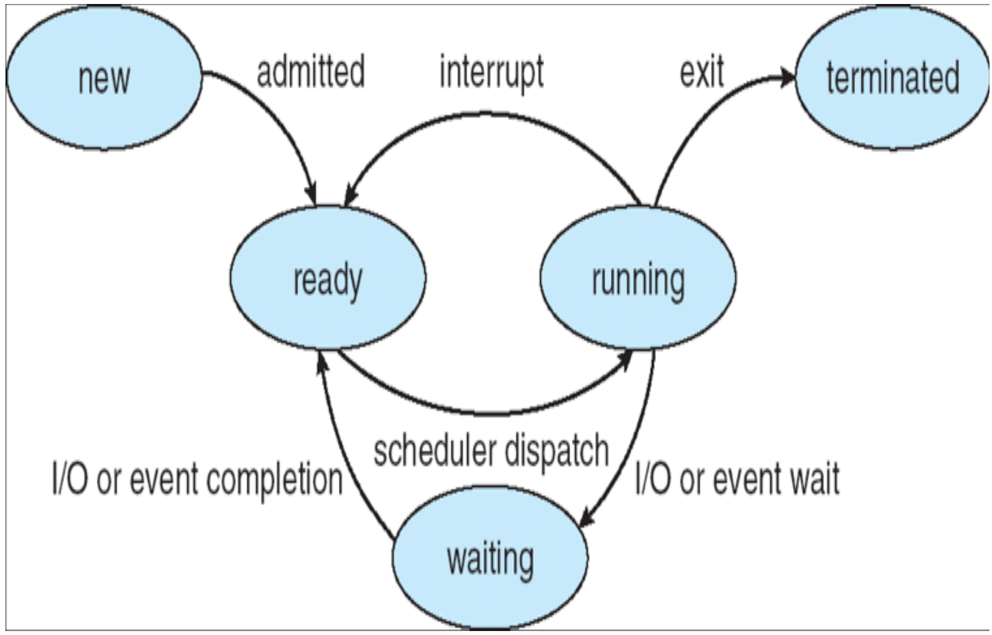
- New Process ==> Ready
  - Allocate resources
  - End of process queue
- Ready ==> Running
  - Head of process queue
  - Scheduled
- Running ==> Ready
  - Interrupt (Timer)
  - Back to end of process queue



State Transitions: Page Fault Handling

UNIVERSITY OF OREGON

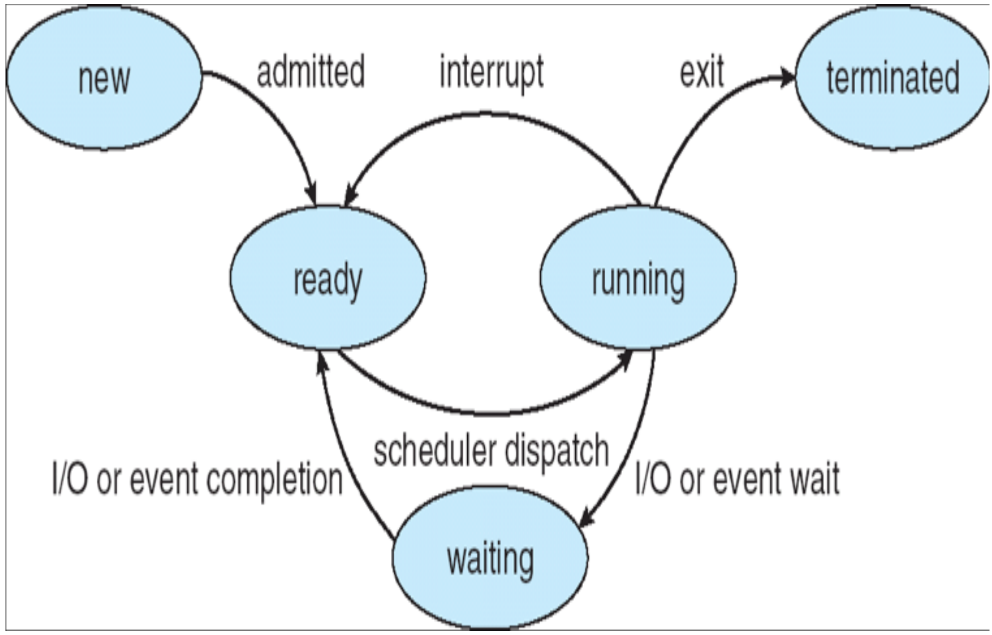
- Running ==> Waiting
  - Page fault exception (similar for syscall or I/O interrupt)
  - Wait for event
- Waiting ==> Ready
  - Event has occurred (page fault serviced)
  - End of process queue (or head?)
- Ready ==> Running
  - As before…



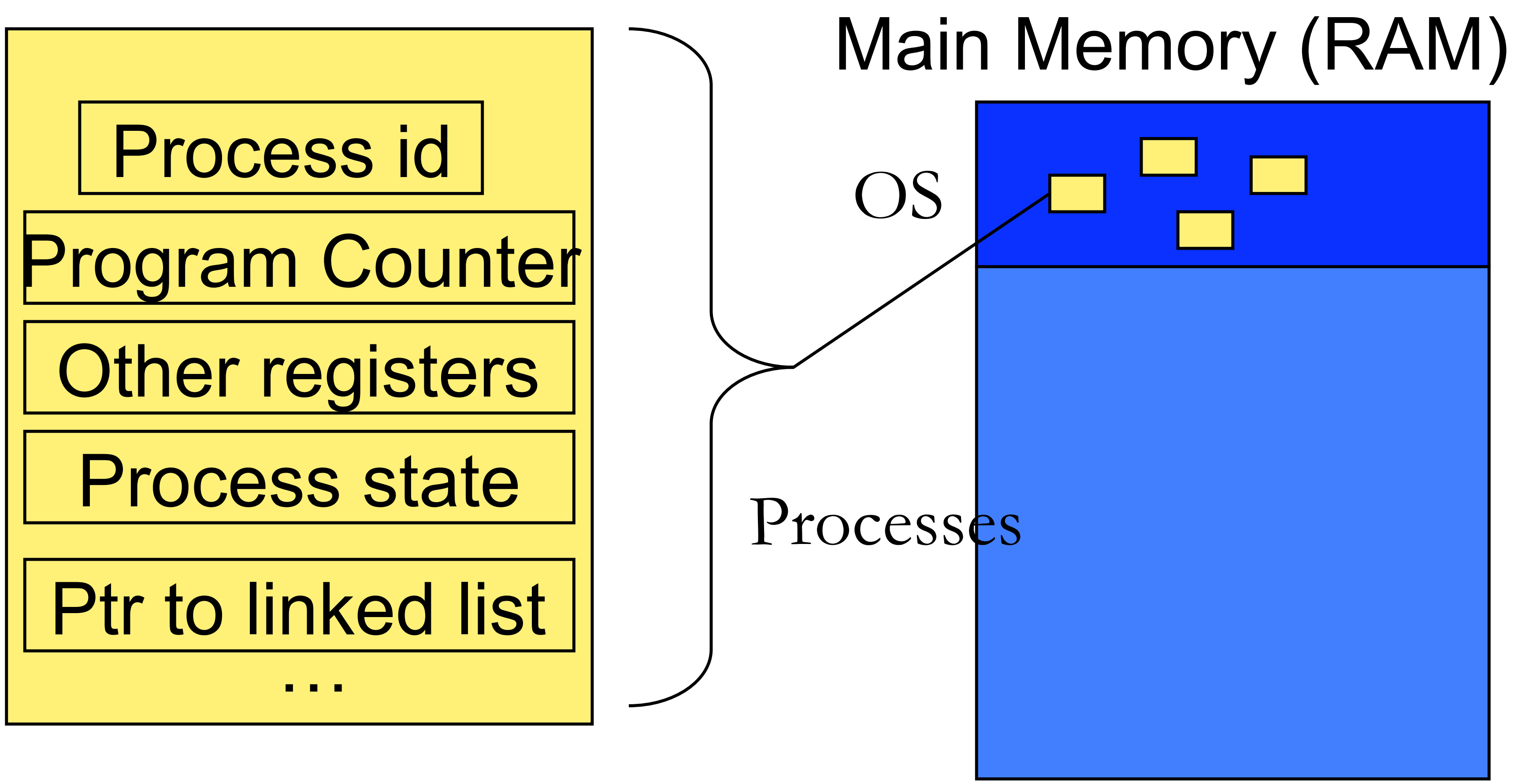
State Transitions: Other Issues

UNIVERSITY OF OREGON

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



Process Control Block



- State of running process
- Linked list of process control information

Per Process Control Info

UNIVERSITY OF OREGON

- 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

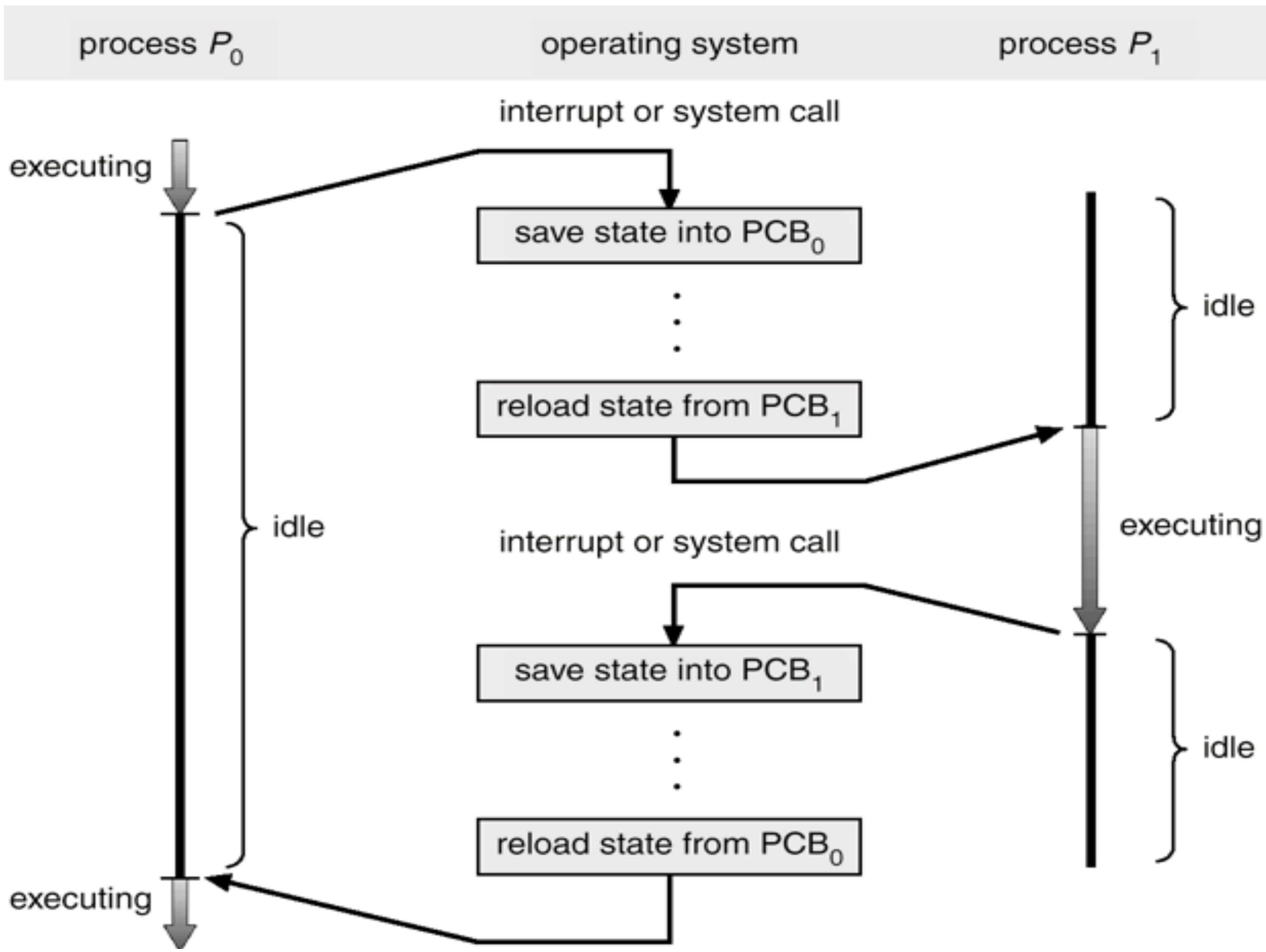
- Linux and Solaris
  - ls /proc
  - A directory for each process
- Various process information
  - `/proc/<pid>/io` -- I/O statistics
  - `/proc/<pid>/environ` -- Environment variables (in binary)
  - `/proc/<pid>/stat` -- process status and info

Context Switch

UNIVERSITY OF OREGON

- 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

- No useful work is being done during a context switch
  - Speed it up and limit system calls to things that can't be done in user mode
- Hardware support
  - Multiple register sets (Sun UltraSPARC)
- However, hardware optimization may conflict
  - TLB flush is necessary
  - Different virtual to physical mappings on different processes

Next class

UNIVERSITY OF OREGON

- IPC