UNIVERSITY OF OREGON

CIS 415: Operating Systems

Processes

Prof. Kevin Butler

Spring 2011

- Last class:
  - Operating system structure and basics
- Today:
  - Process Management


I VOTED
KATIE+ALEX
VOTE!
YOUR SCHOOL. YOUR VOICE

Administrivia

- Lab sections: everyone should know where you're going
  - this week: programming with system calls and signals
- Assignment 1: due next Thursday
- Project 1: out today

Why Processes?

UNIVERSITY OF OREGON

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

Overview

- 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

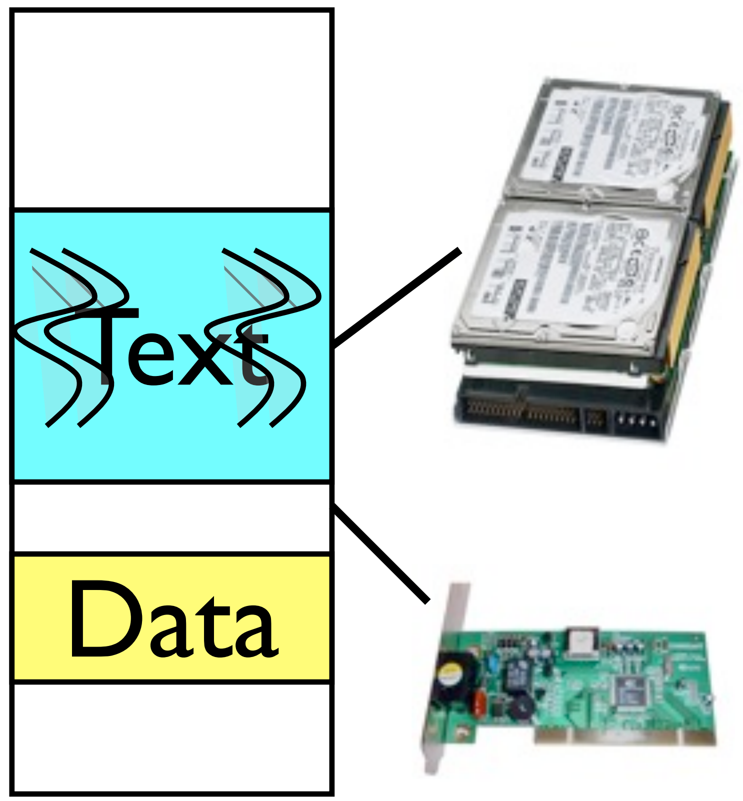
Supervisor and User Modes

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?

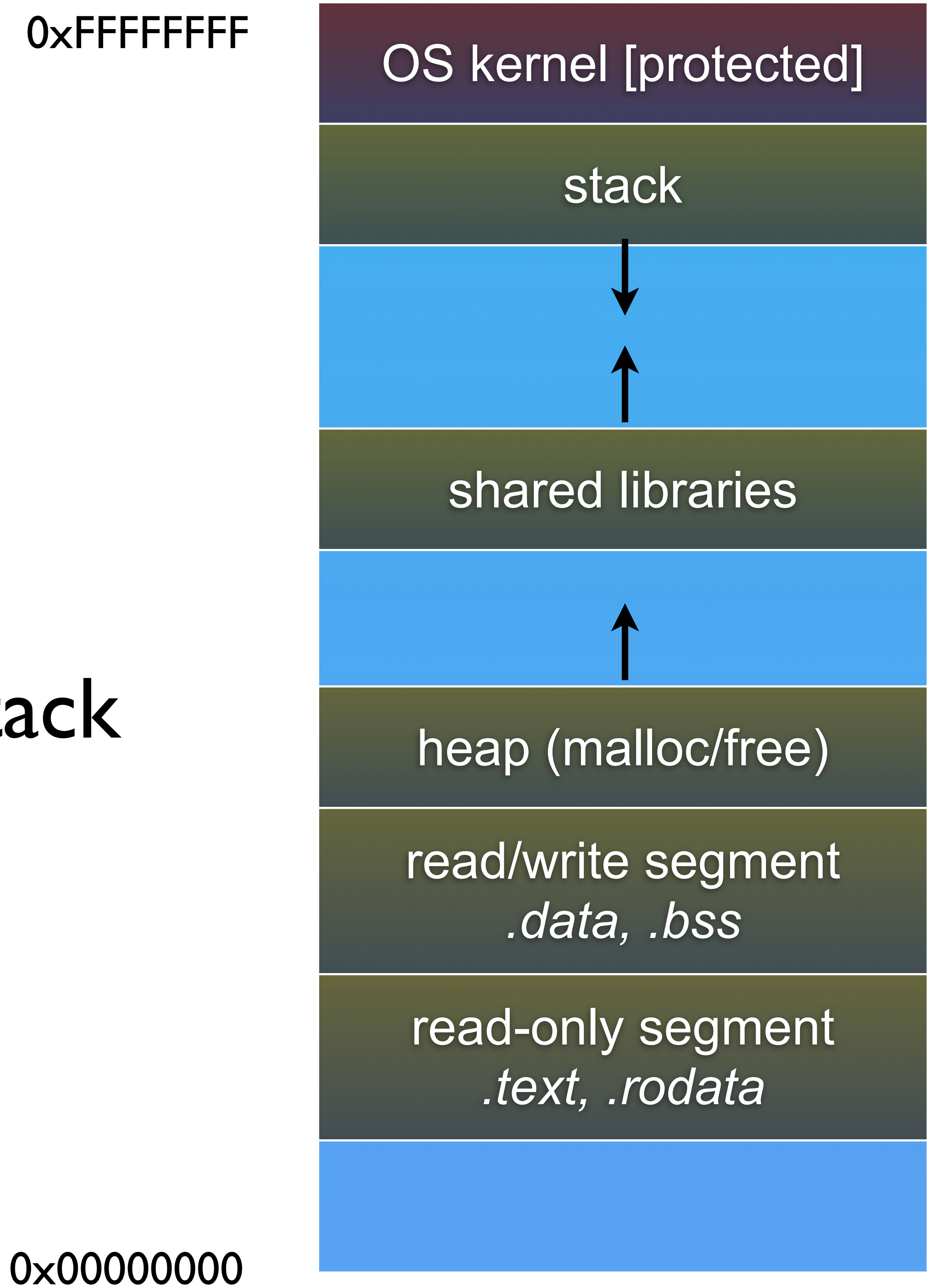
Process

- 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

- 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

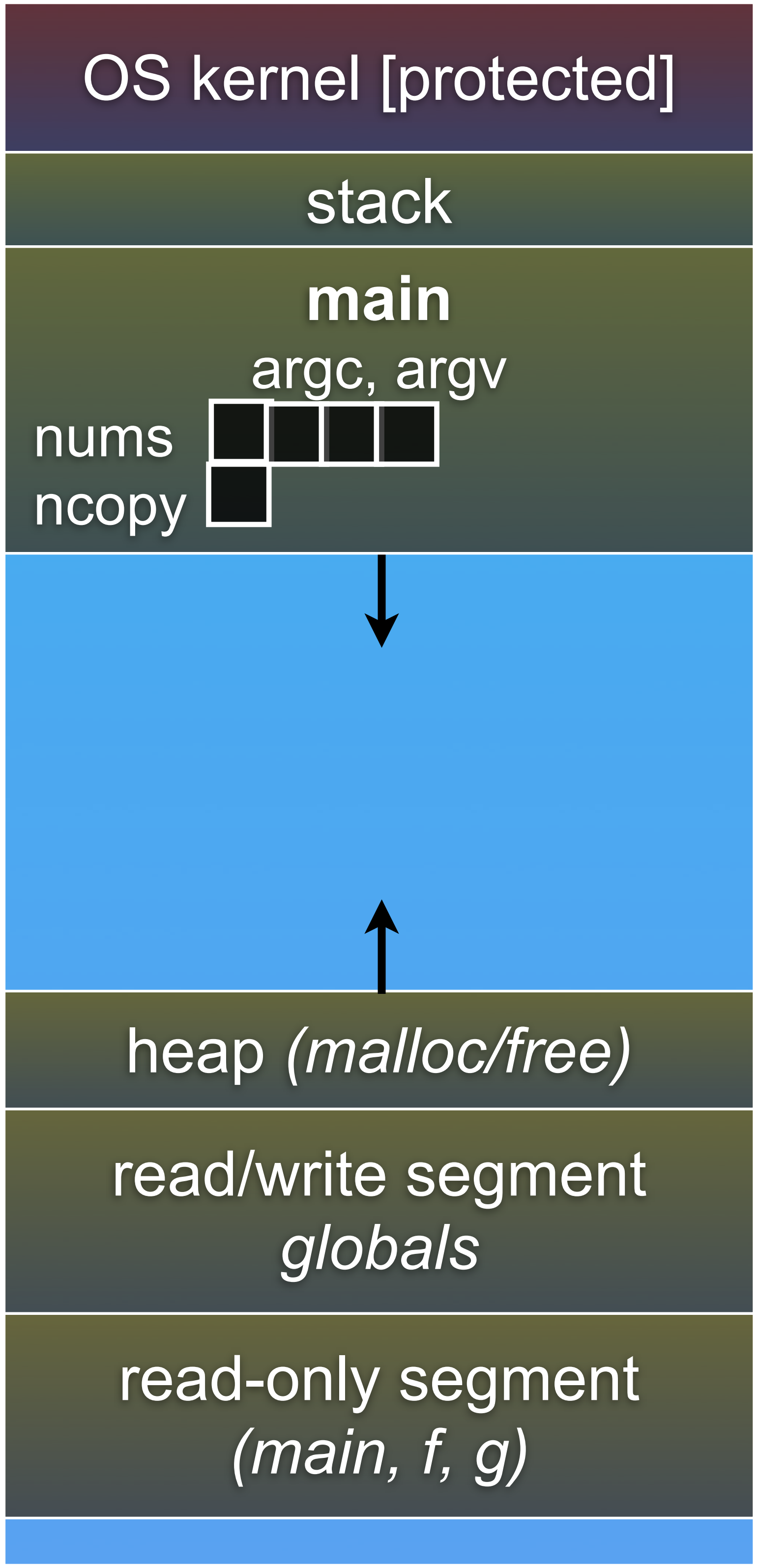

```c
#include <stdlib.h>

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

UNIVERSITY OF OREGON

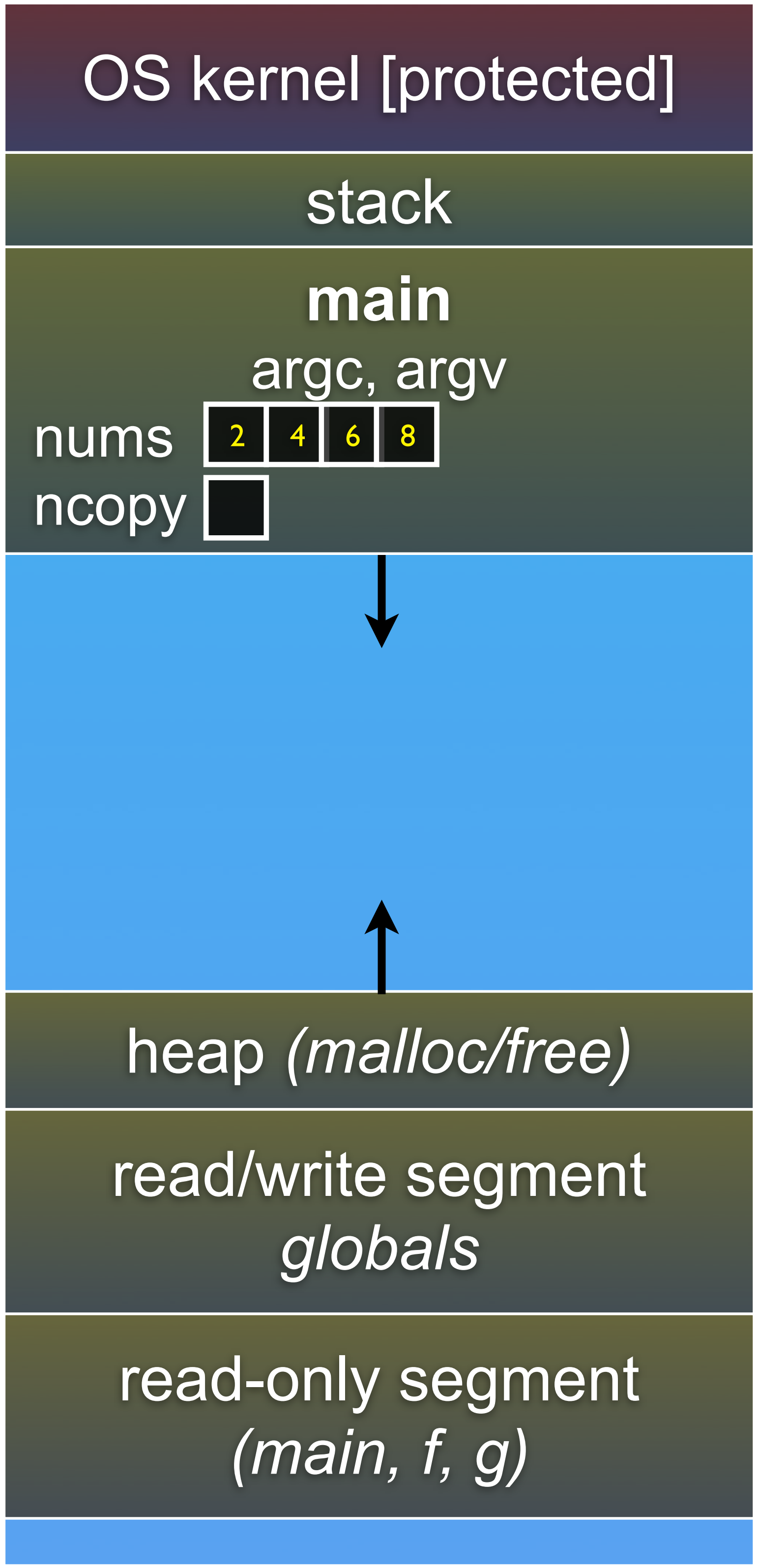
```c
#include <stdlib.h>

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

UNIVERSITY OF OREGON

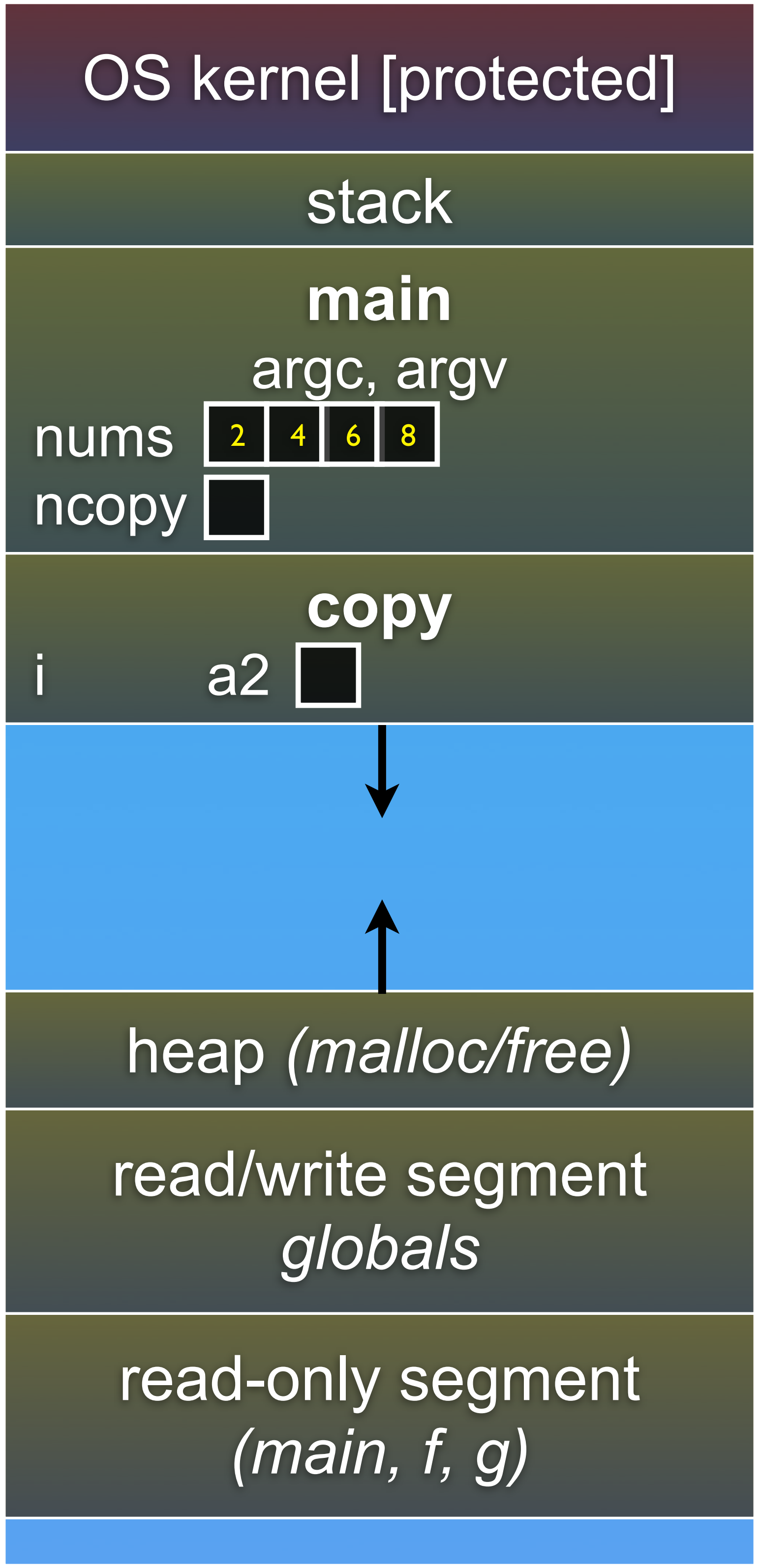
```c
#include <stdlib.h>

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

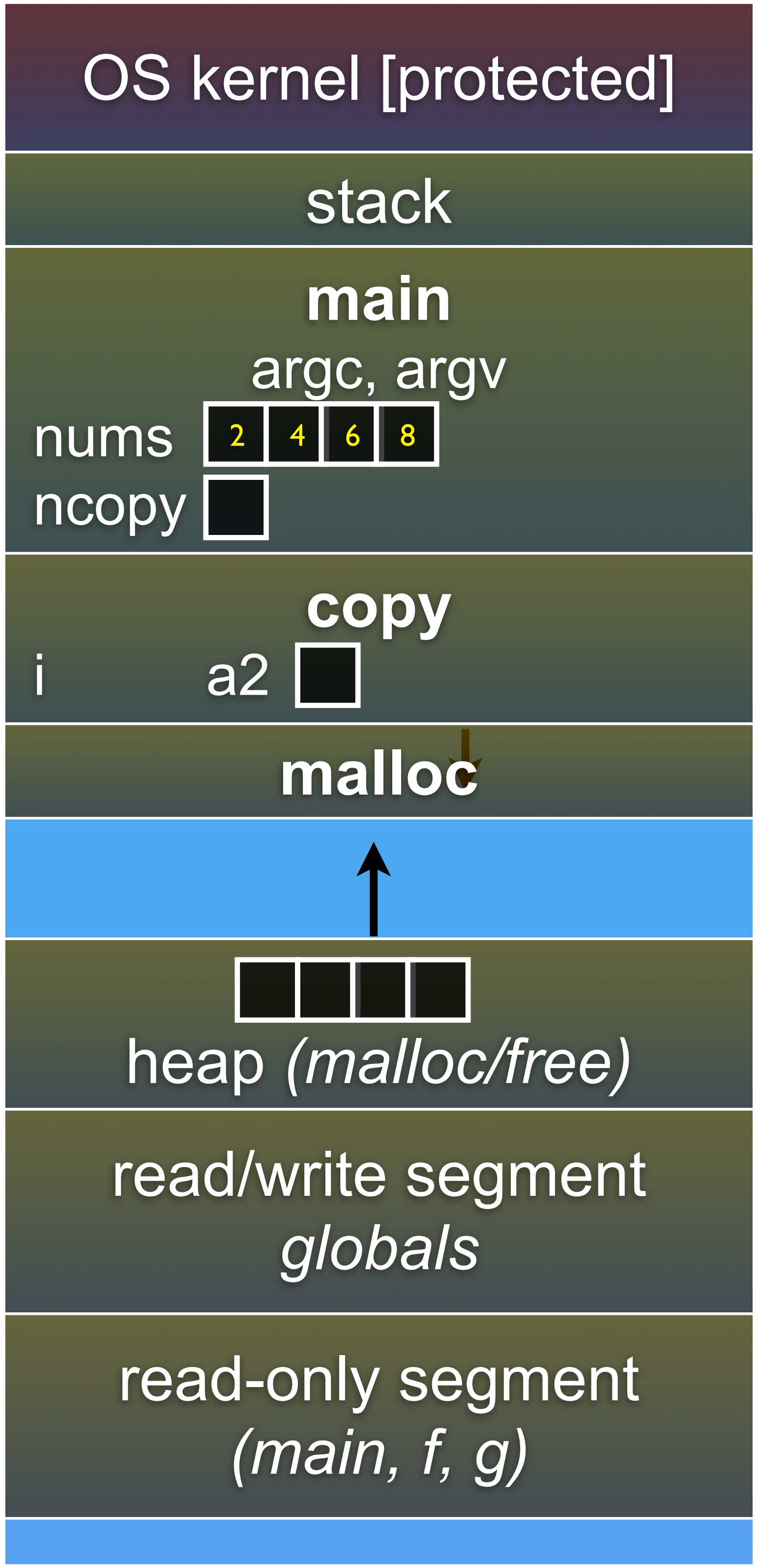
```c
#include <stdlib.h>

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

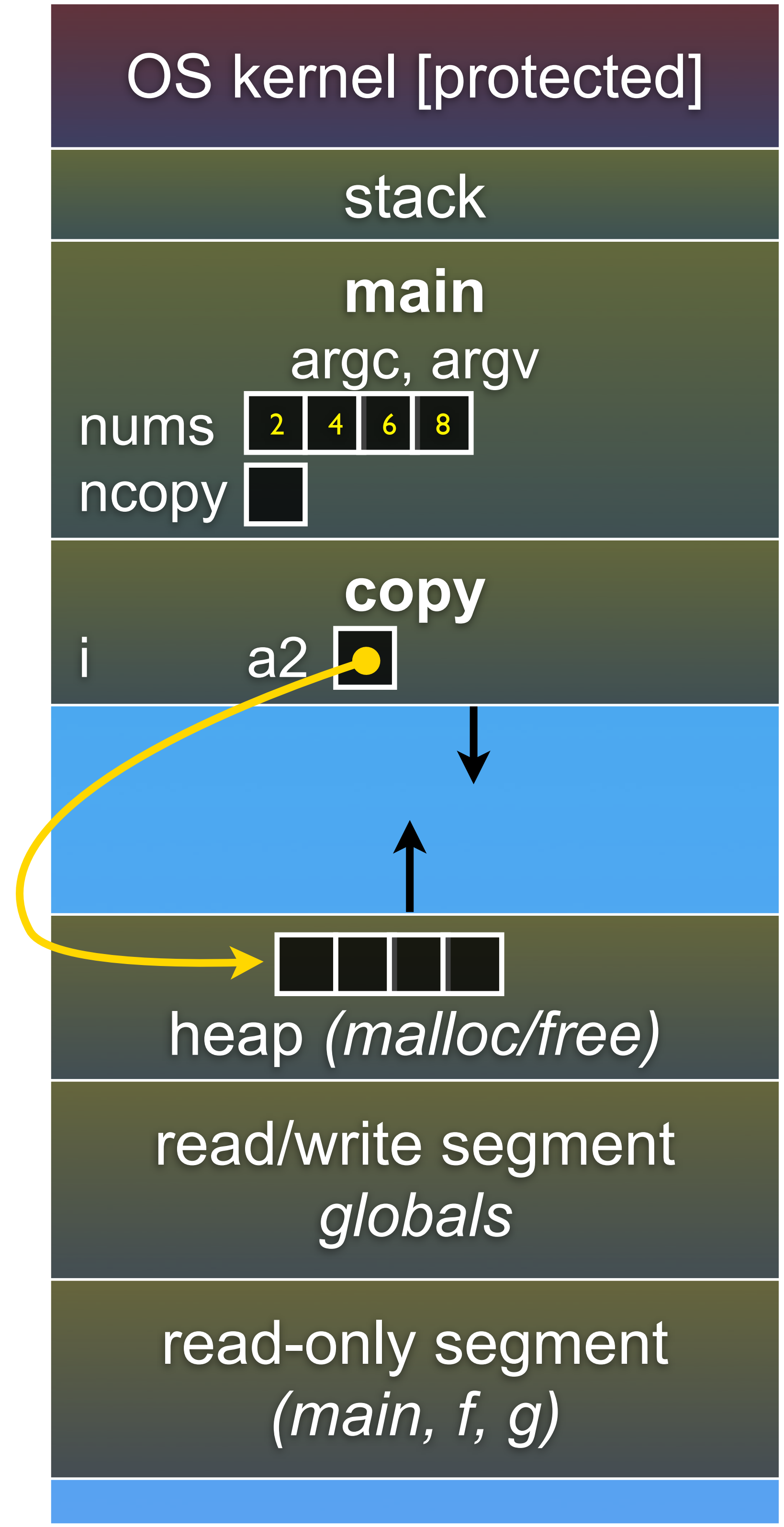
```c
#include <stdlib.h>

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

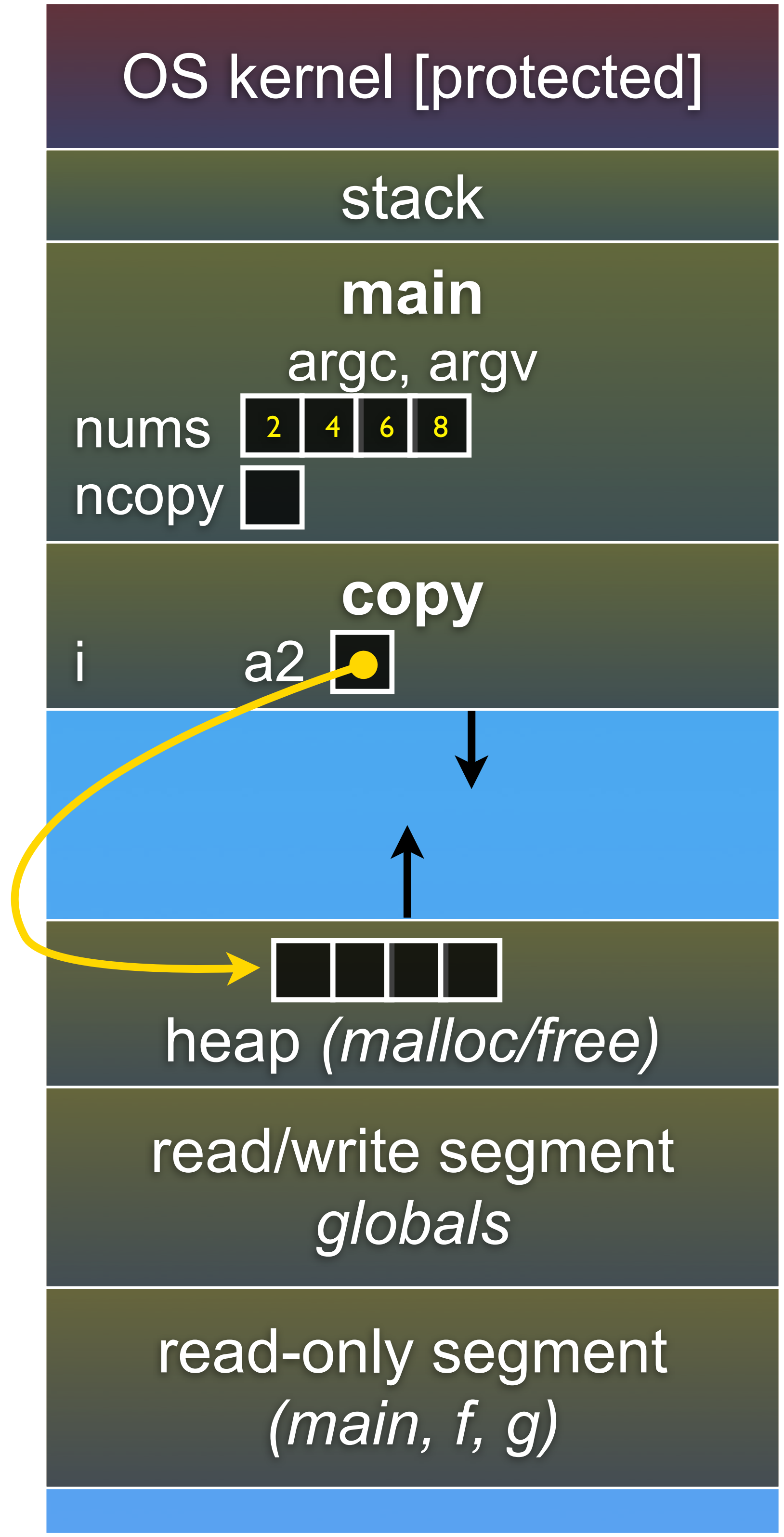

```c
#include <stdlib.h>

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

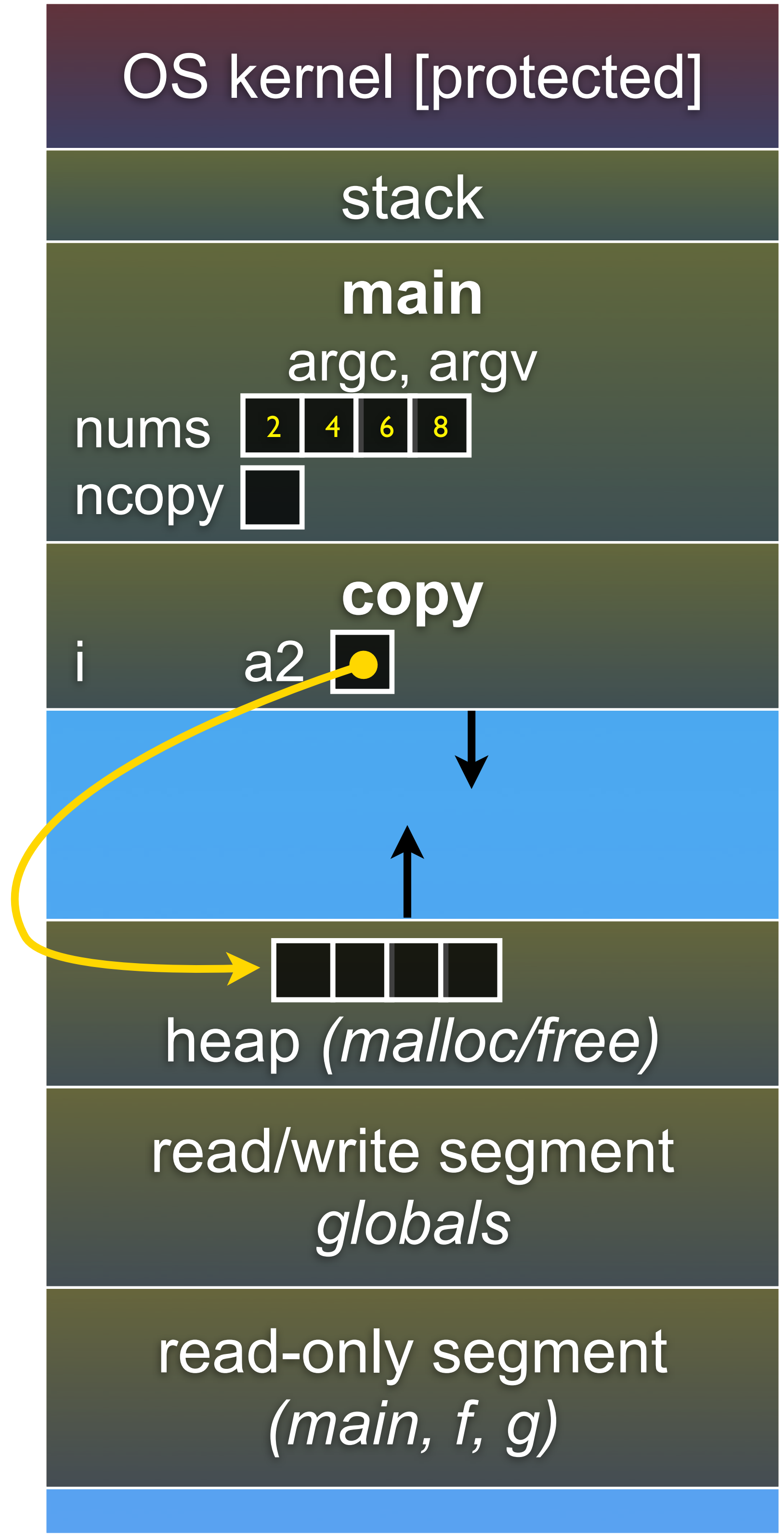
```c
#include <stdlib.h>

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

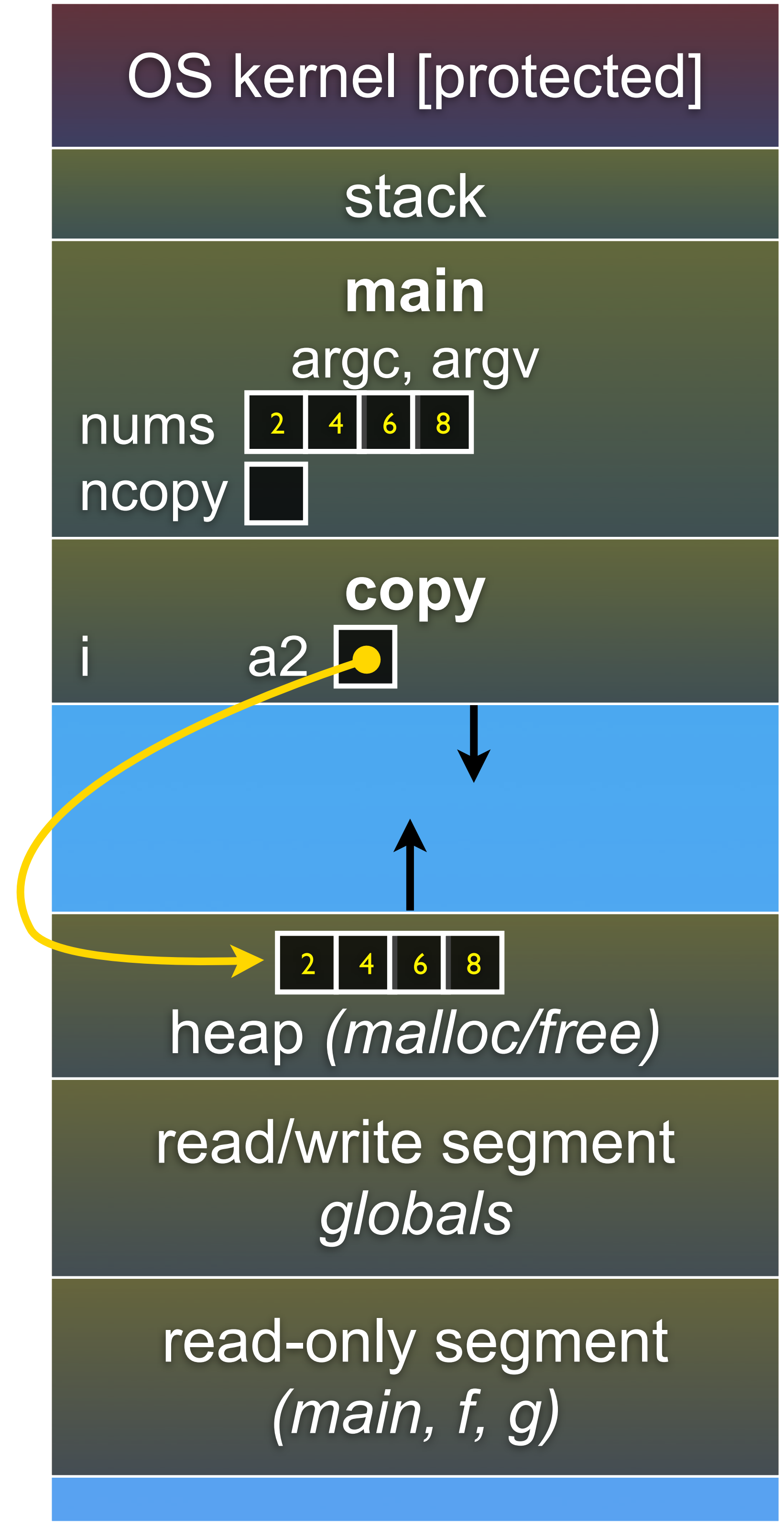
```c
#include <stdlib.h>

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

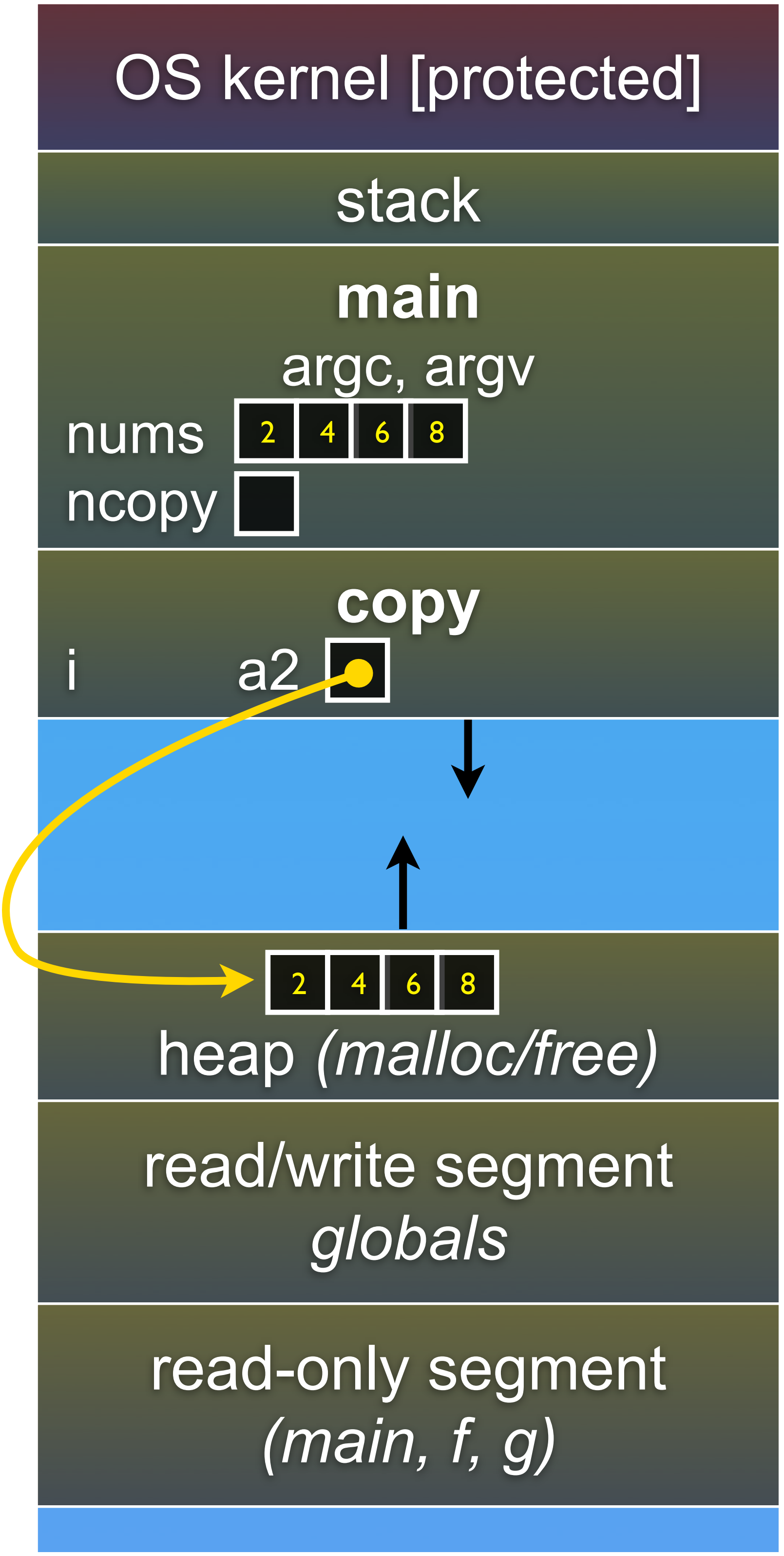
```c
#include <stdlib.h>

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

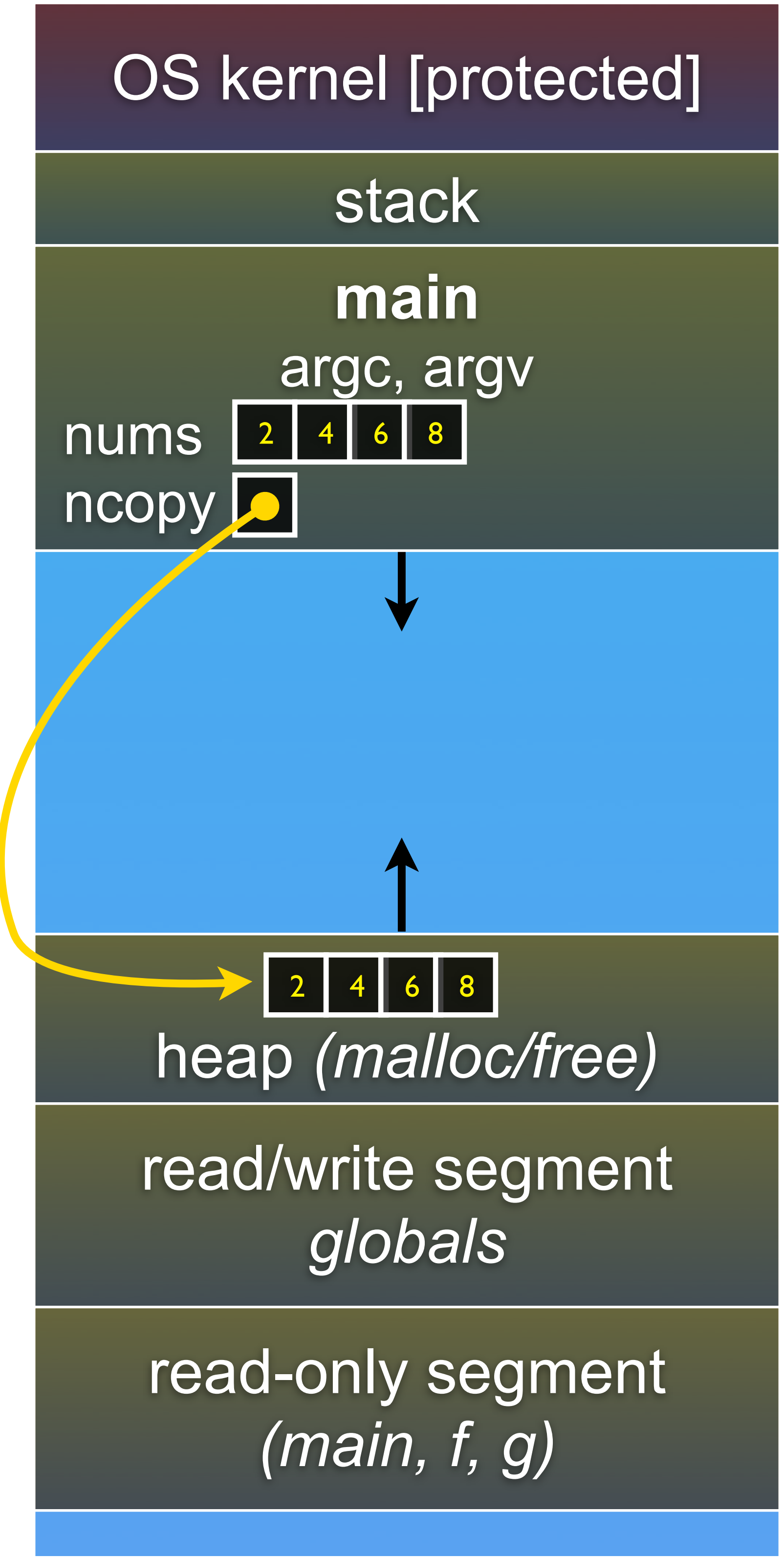
```c
#include <stdlib.h>

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

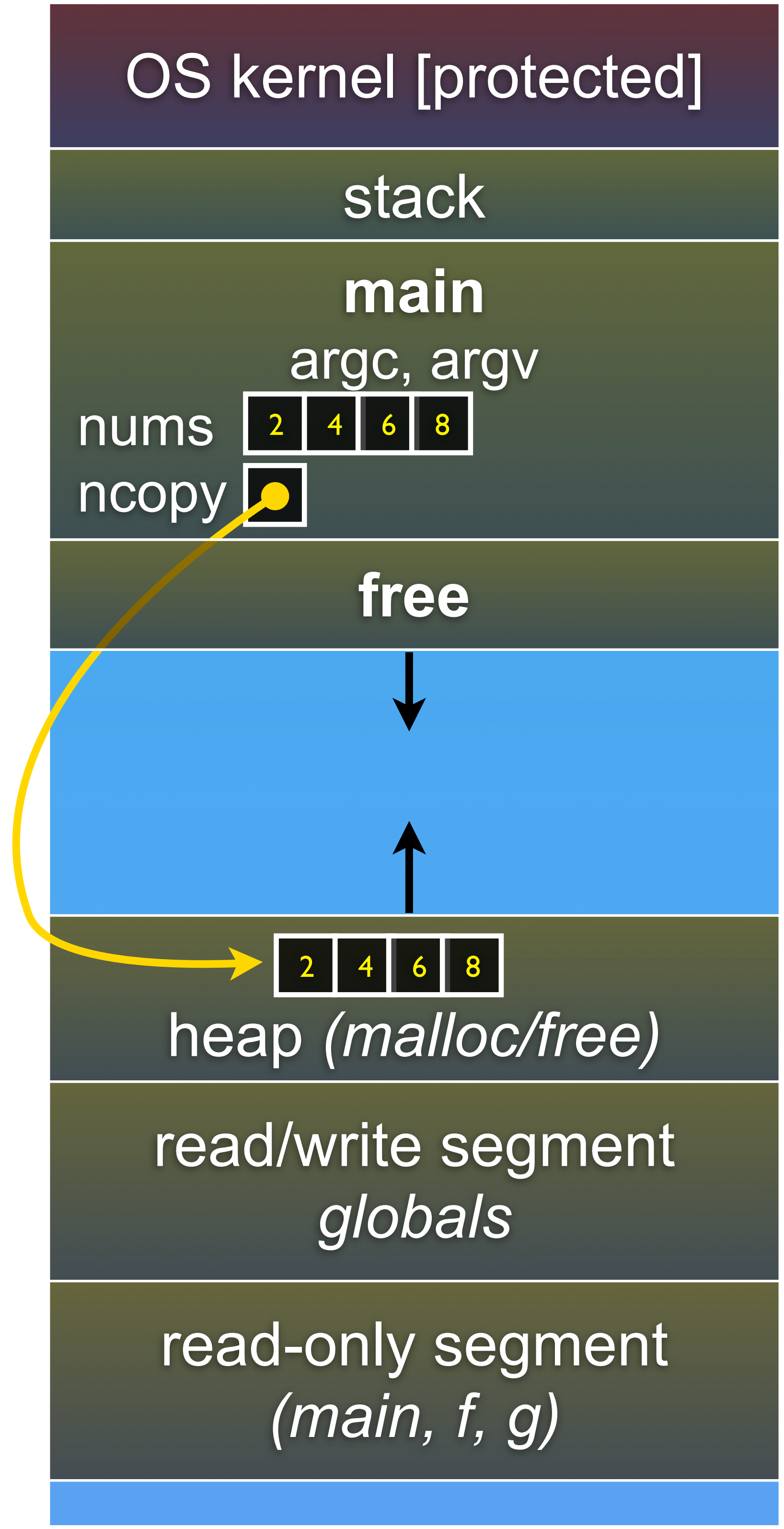

```c
#include <stdlib.h>

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

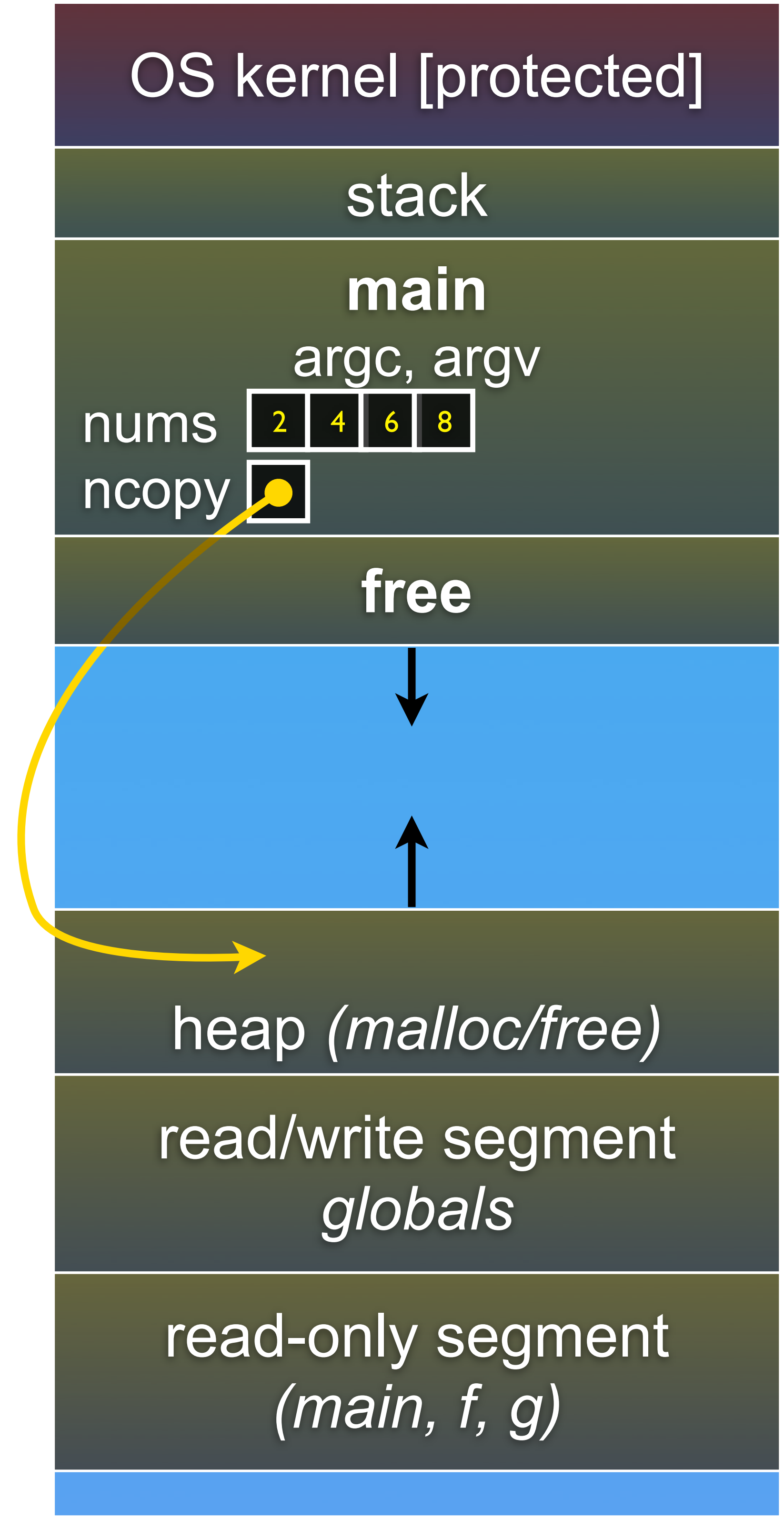
```
#include <stdlib.h>

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

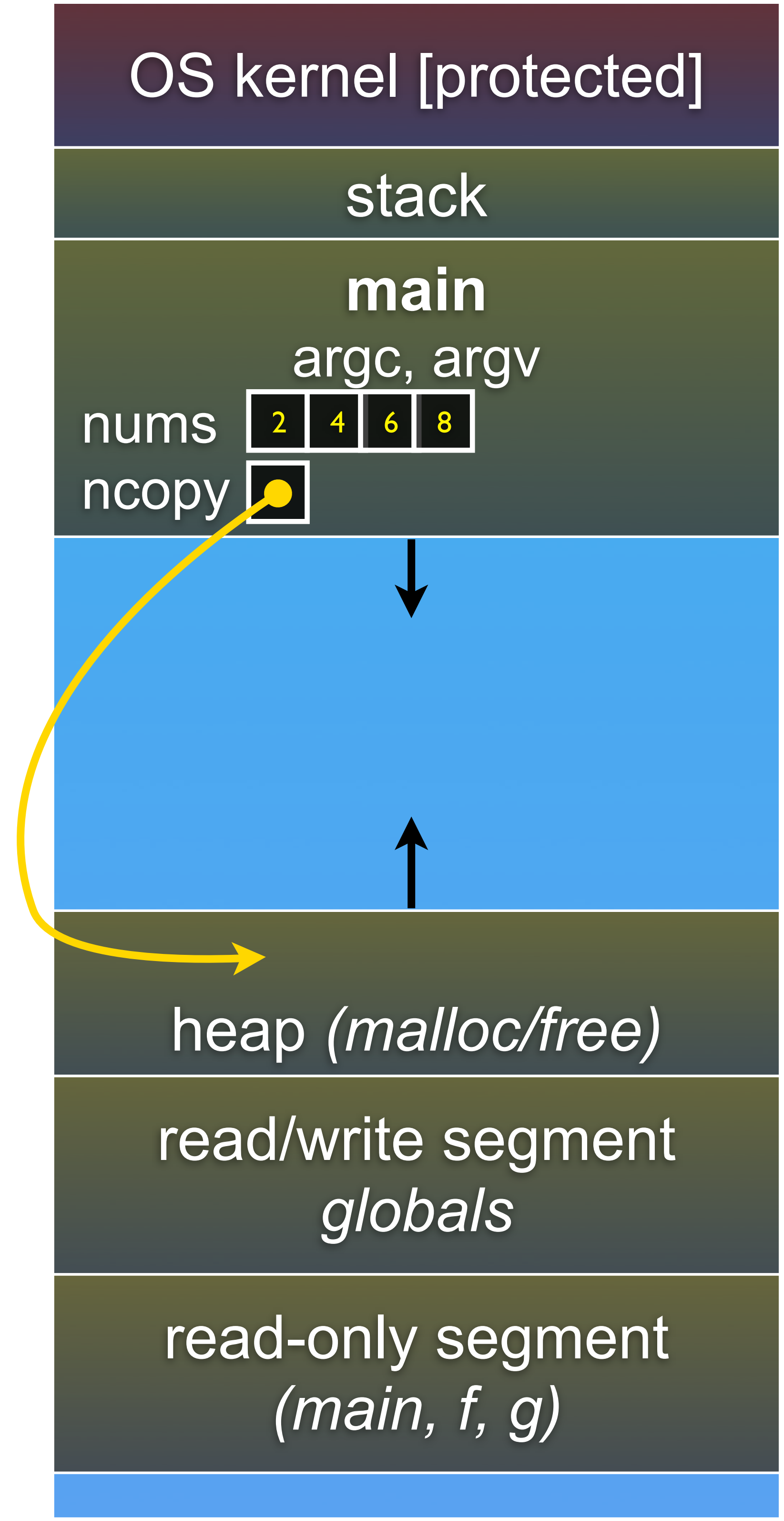
```c
#include <stdlib.h>

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

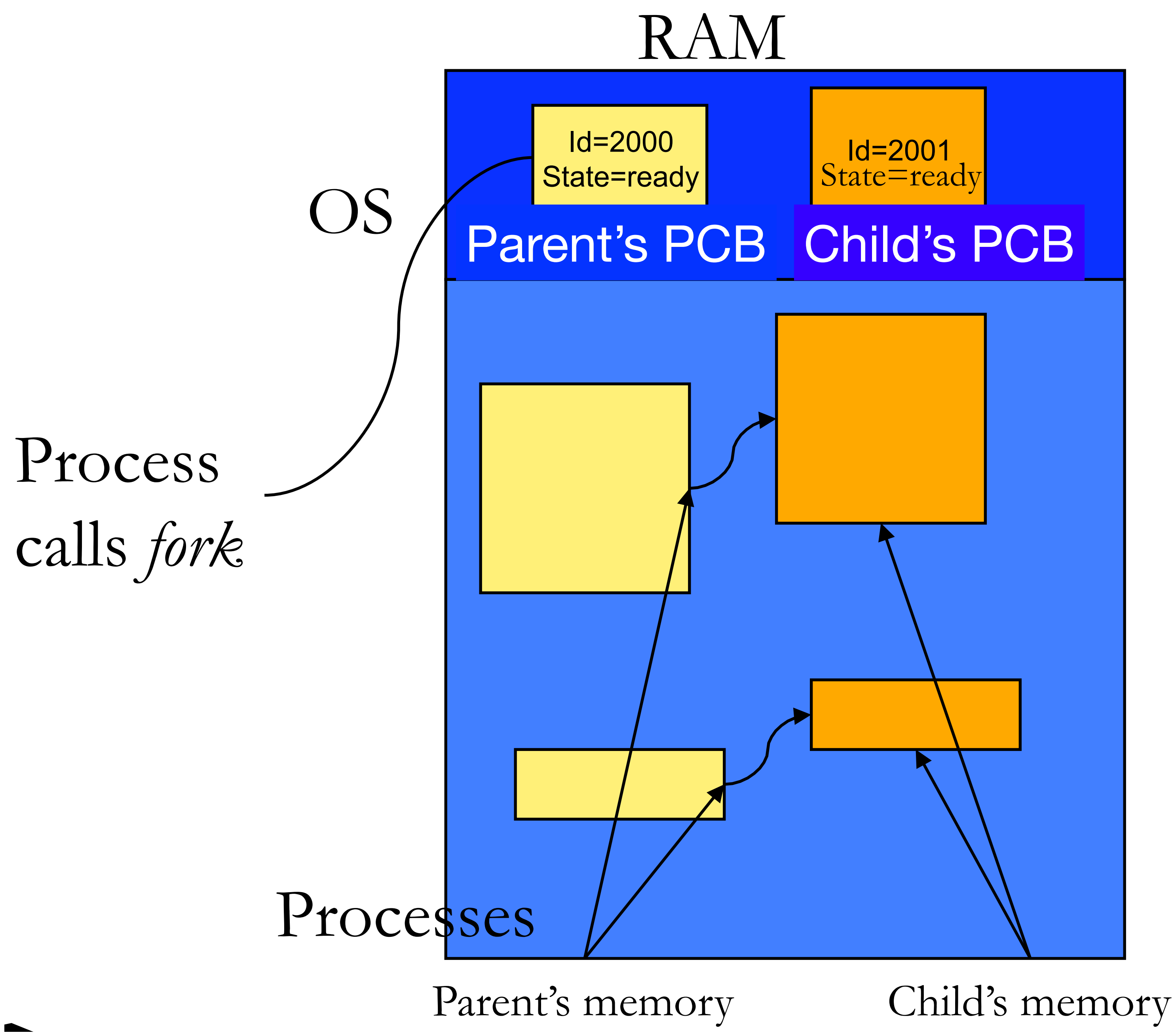
- Parent process create children processes,
  - which, in turn create other processes, forming a tree of processes
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent 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 Layout



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

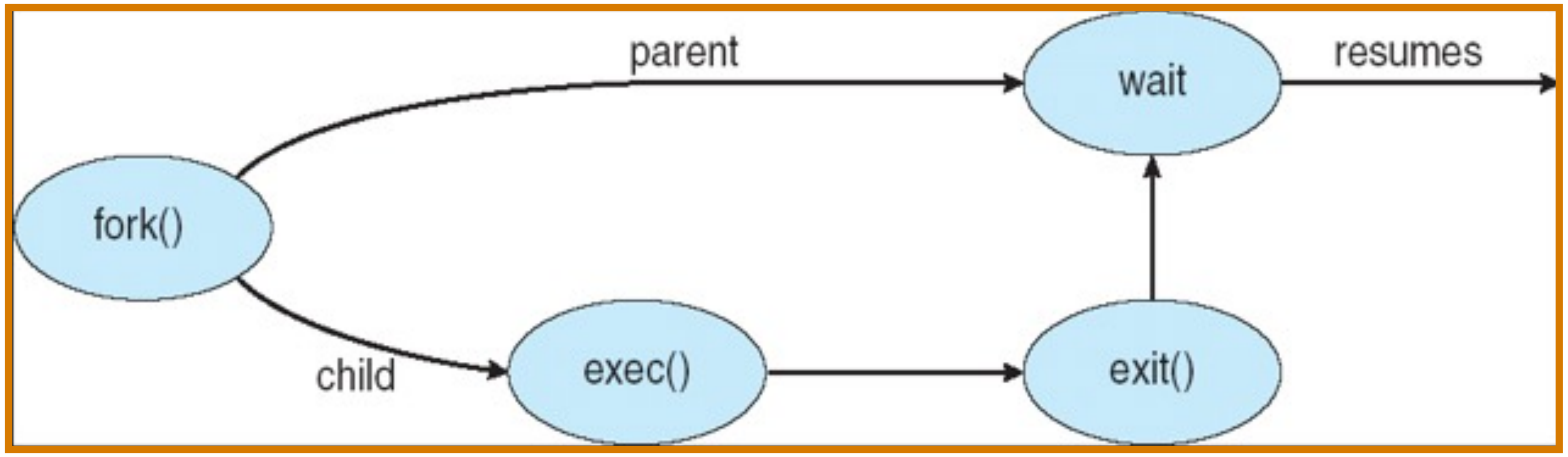
Initialized by copying over from the parent

3. If parent had called **wait**, it is moved to a waiting queue
4. If child had called **exec**, its memory overwritten with new code & data
5. Child added to ready queue, all set to go now!

Process Creation

- 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



C Program Forking Separate Process

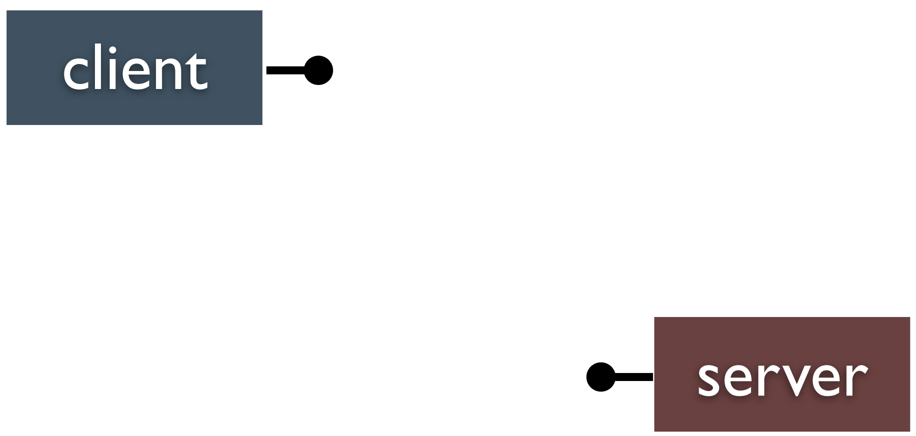
UNIVERSITY OF OREGON

```
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);
   }
}
```

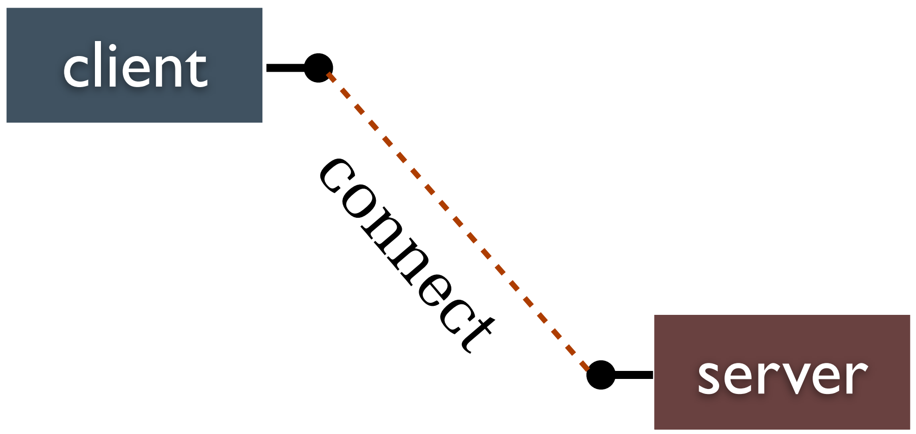
Graphically

Graphically

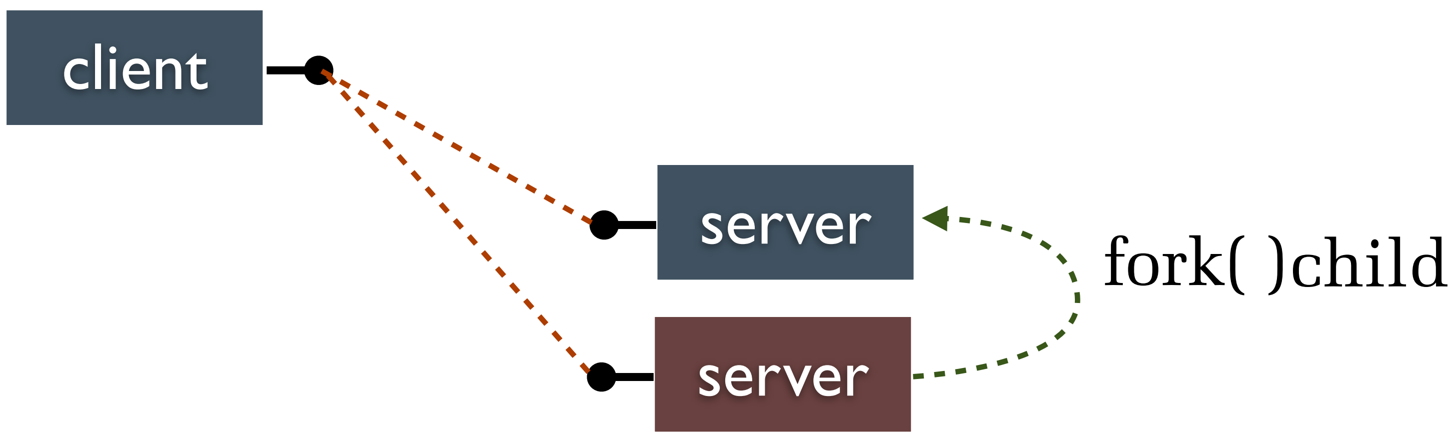
UNIVERSITY OF OREGON



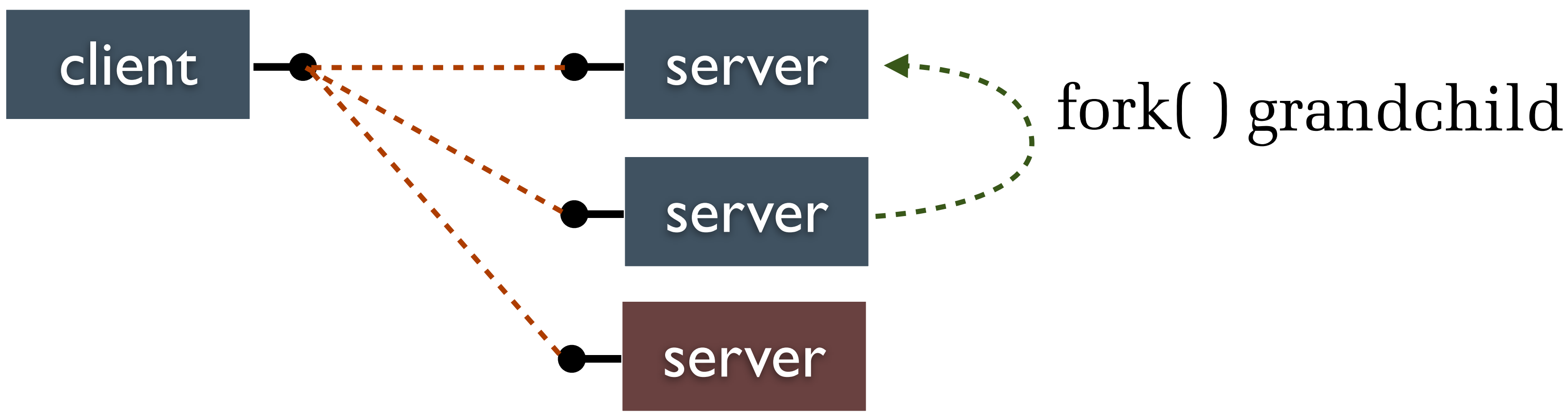
Graphically



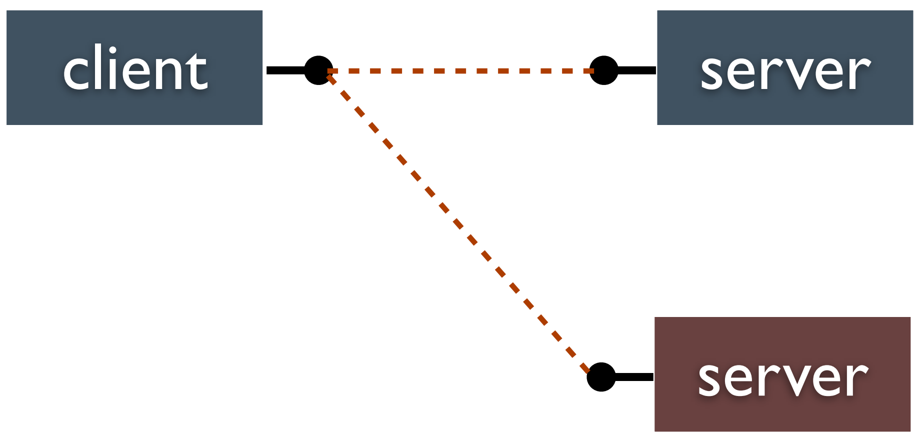
Graphically



Graphically

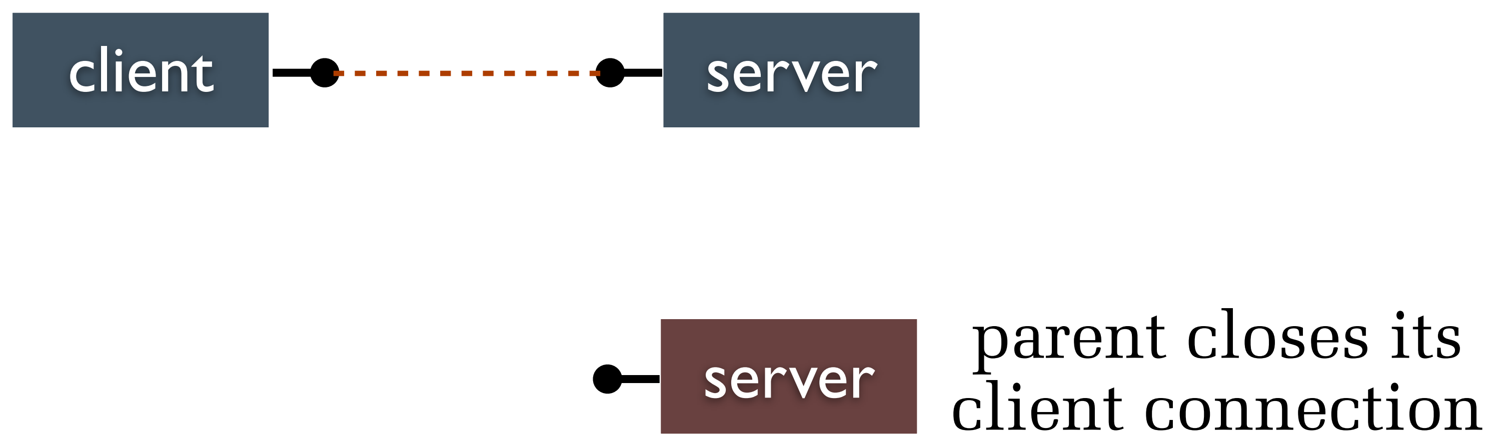


Graphically

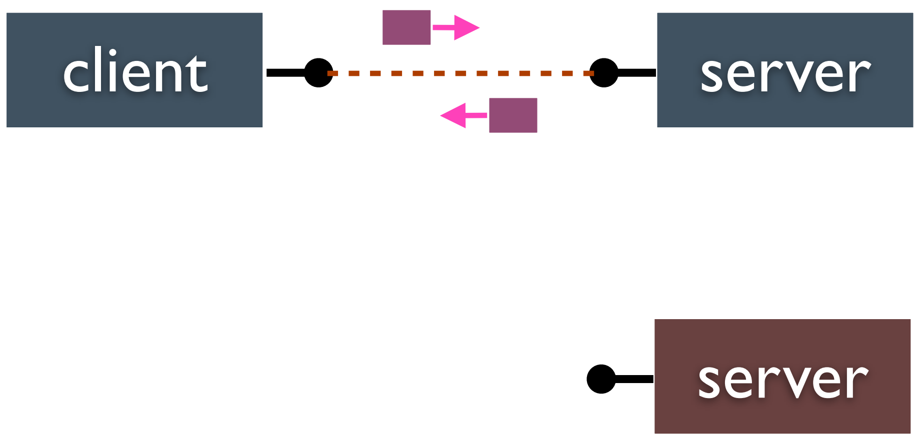


child exit( )'s / parent wait( )'s

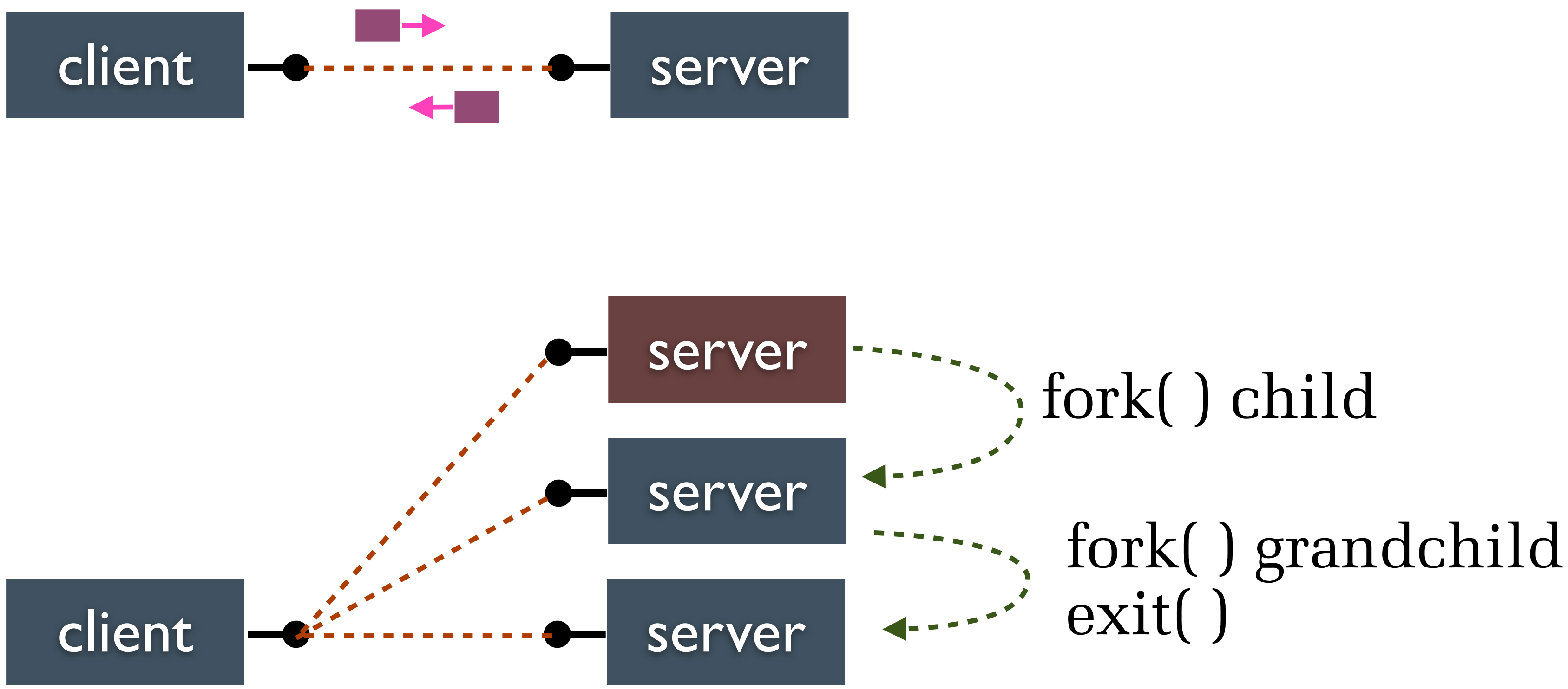
Graphically



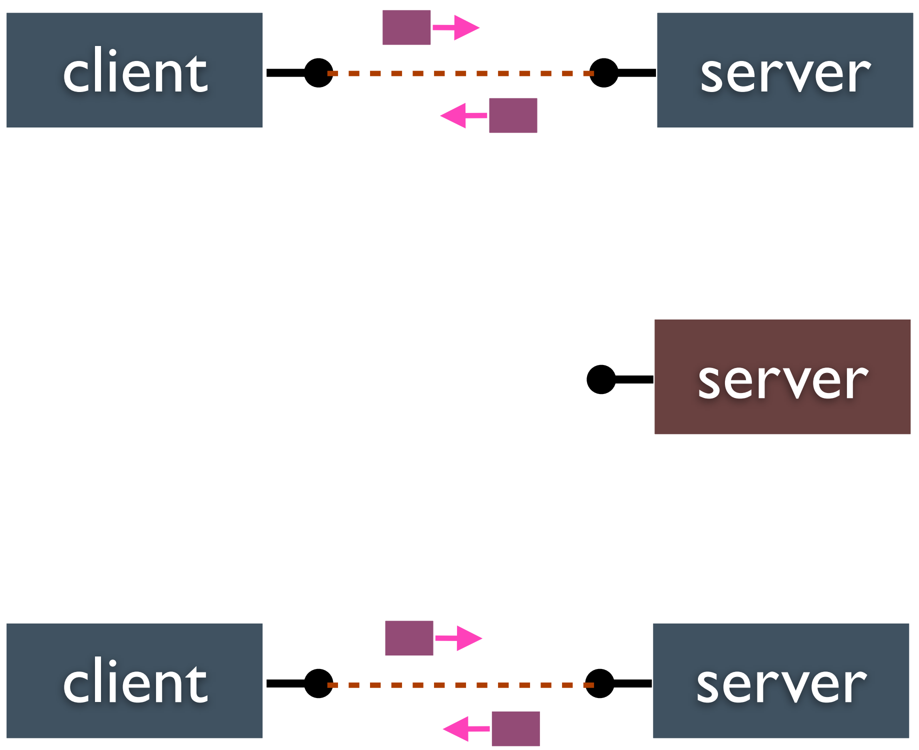
Graphically



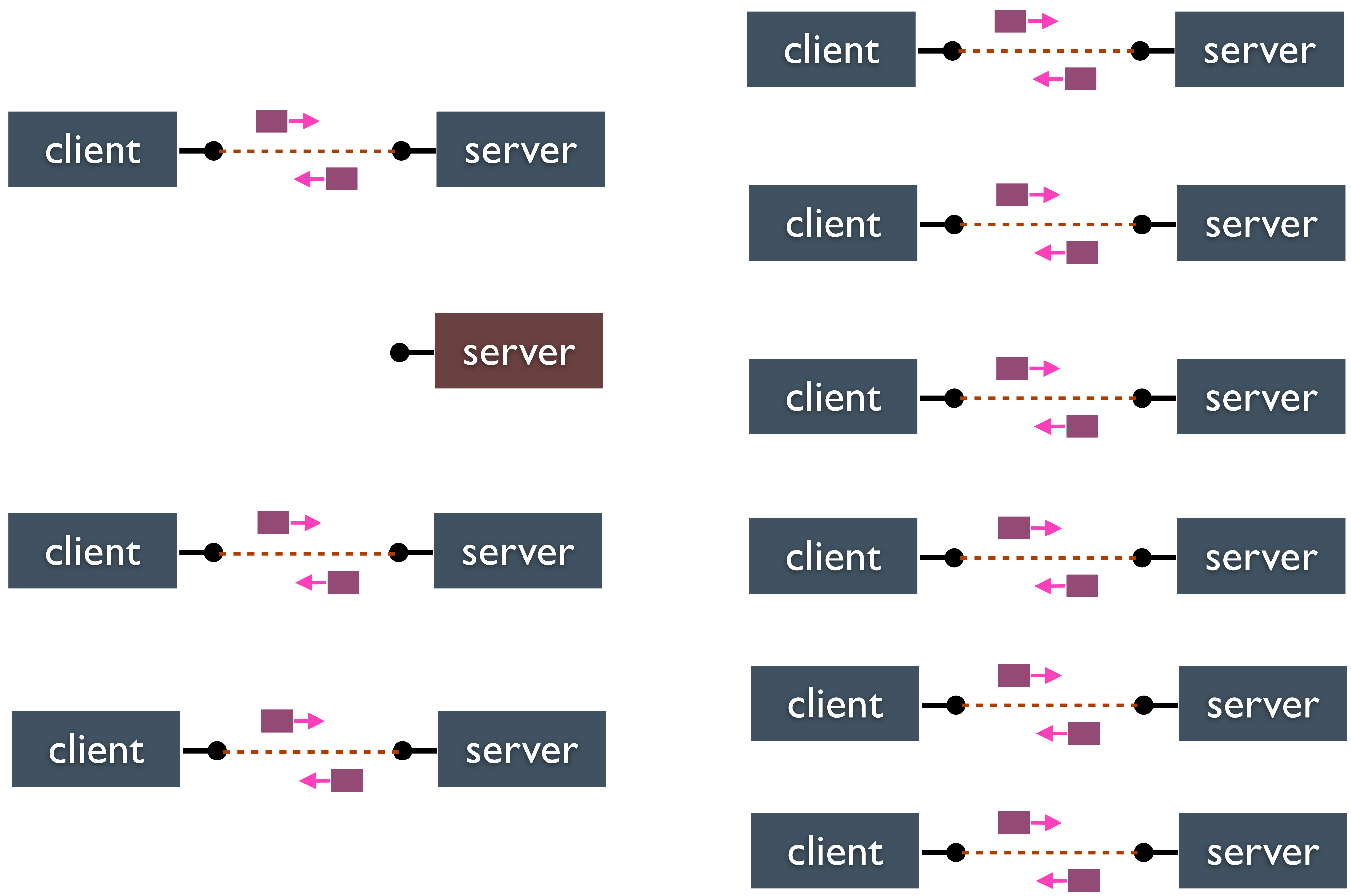
Graphically



Graphically



Graphically



Program Creation

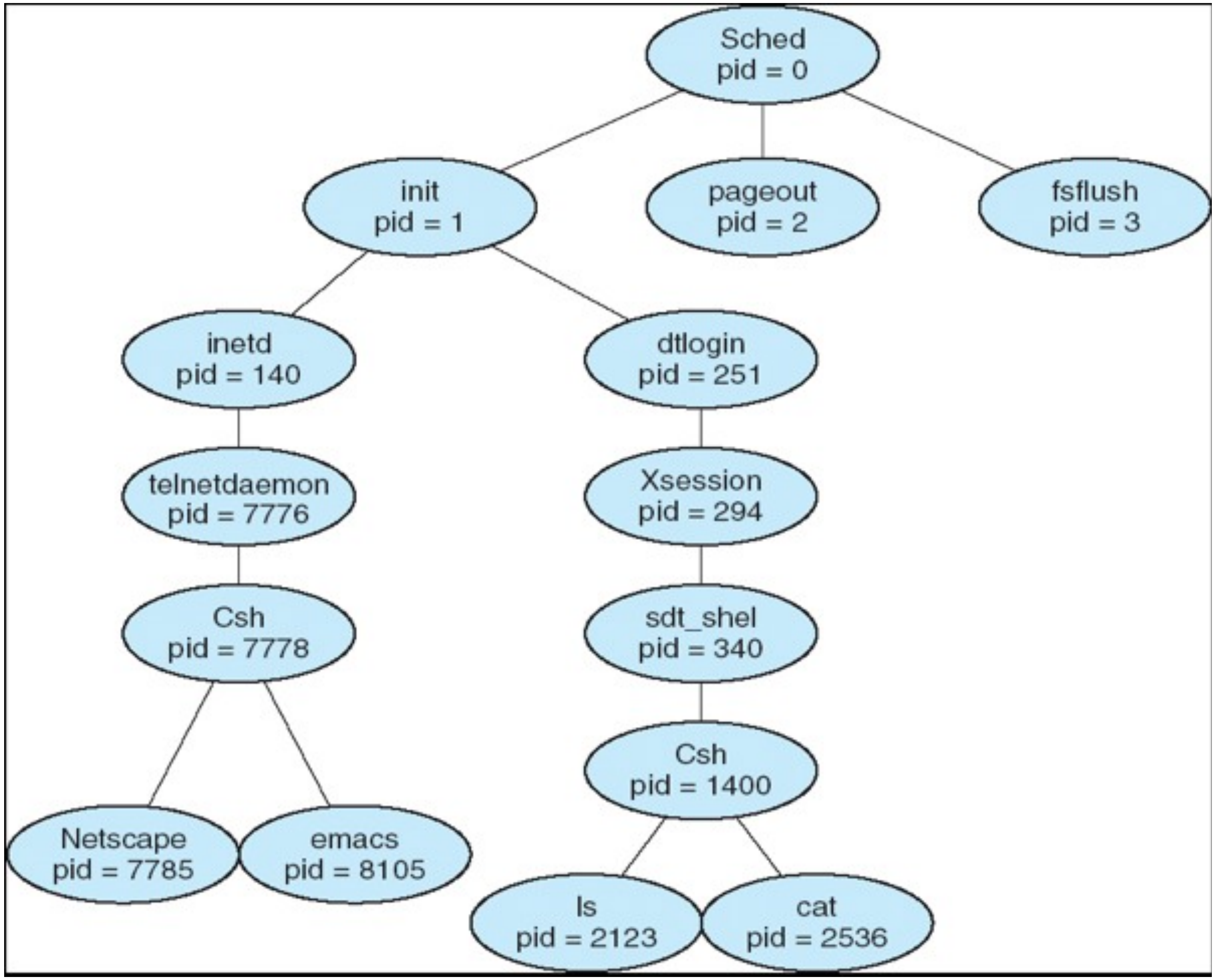
- Design Choices
  - Resource Sharing
    - What resources of parent should the child share?
    - What about after `exec`?
  - Execution
    - Should parent wait for child?
  - What is the relationship between parent and child?
    - Hierarchical or grouped or …?

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



Process Termination

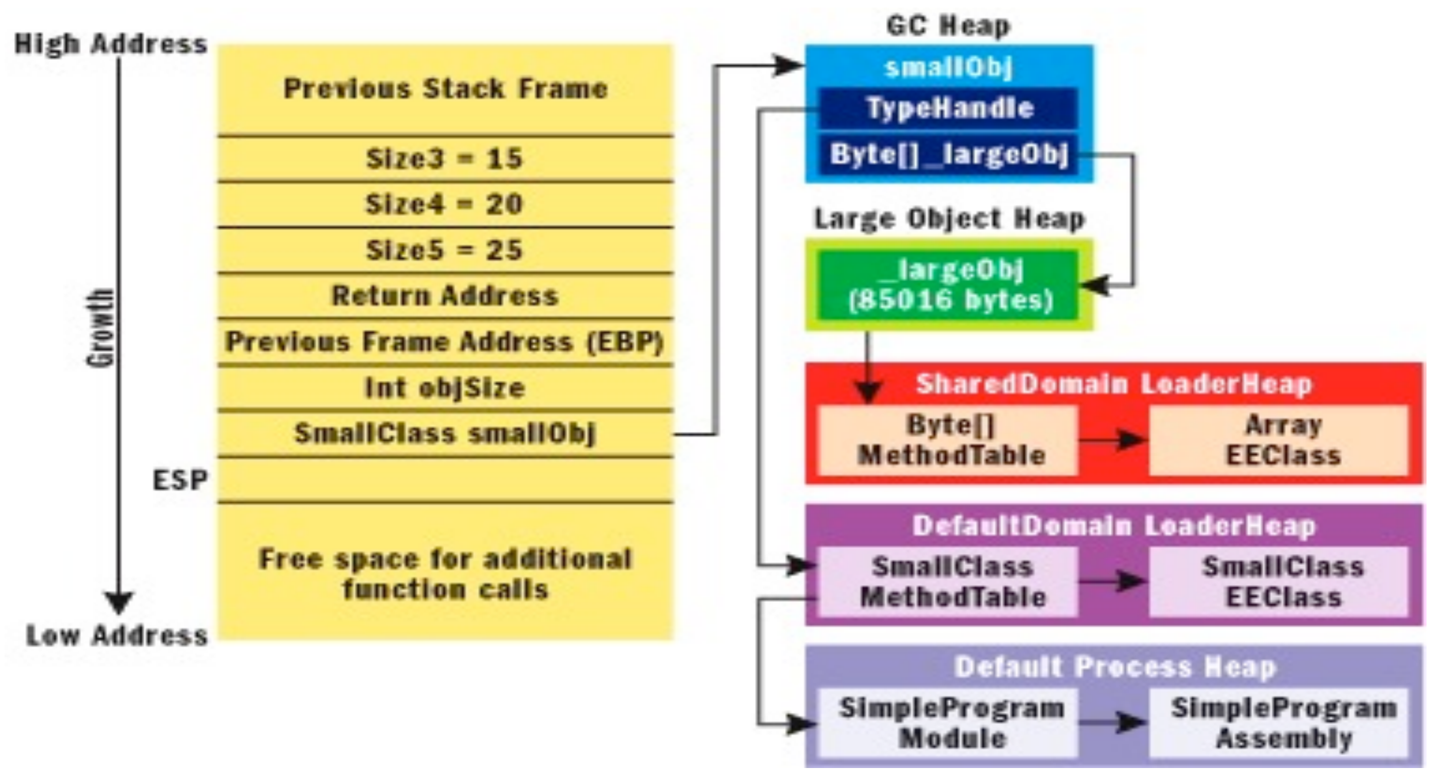
UNIVERSITY OF OREGON

- 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

UNIVERSITY OF OREGON

- What to execute?
  - Program status word
  - 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



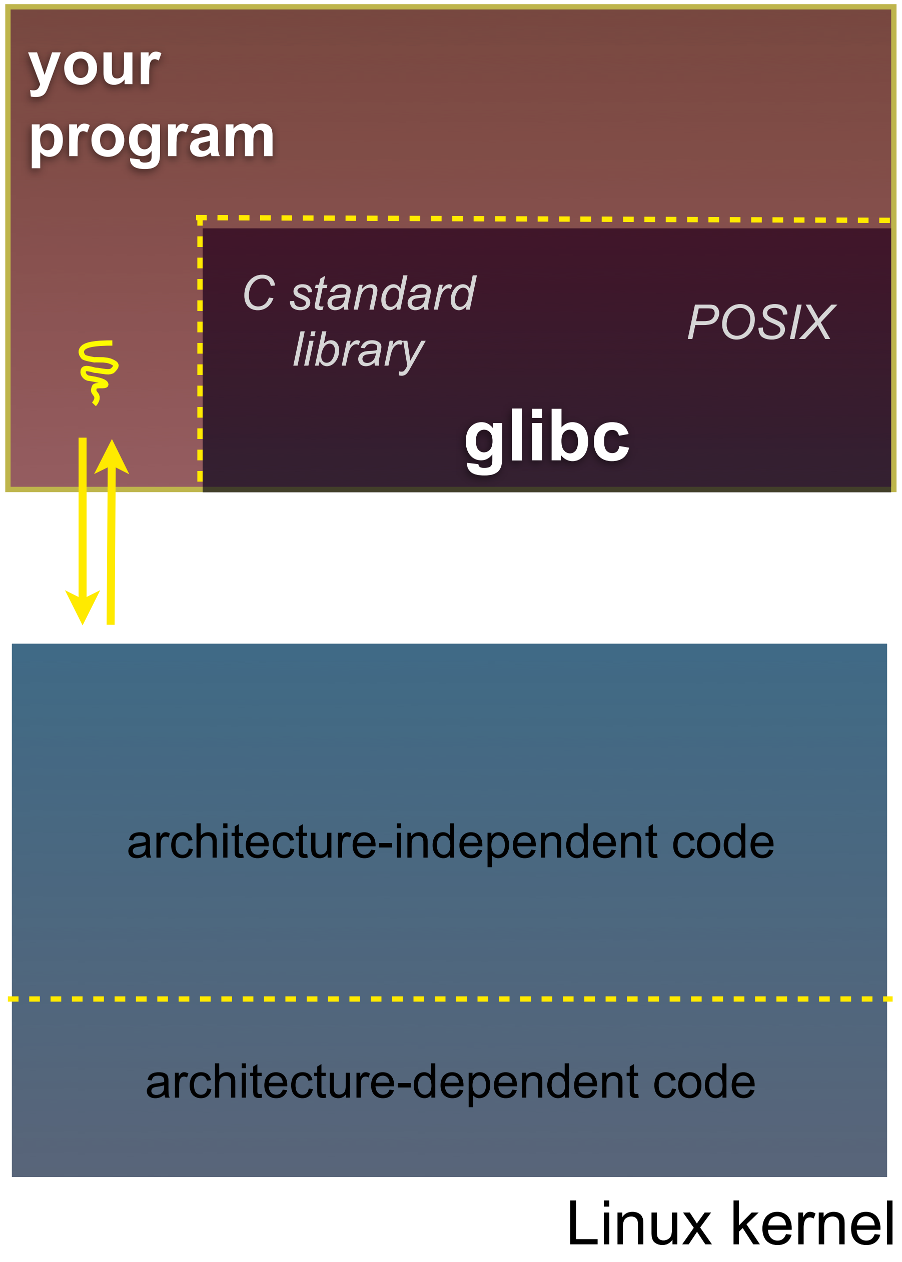
Executing a Process

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)

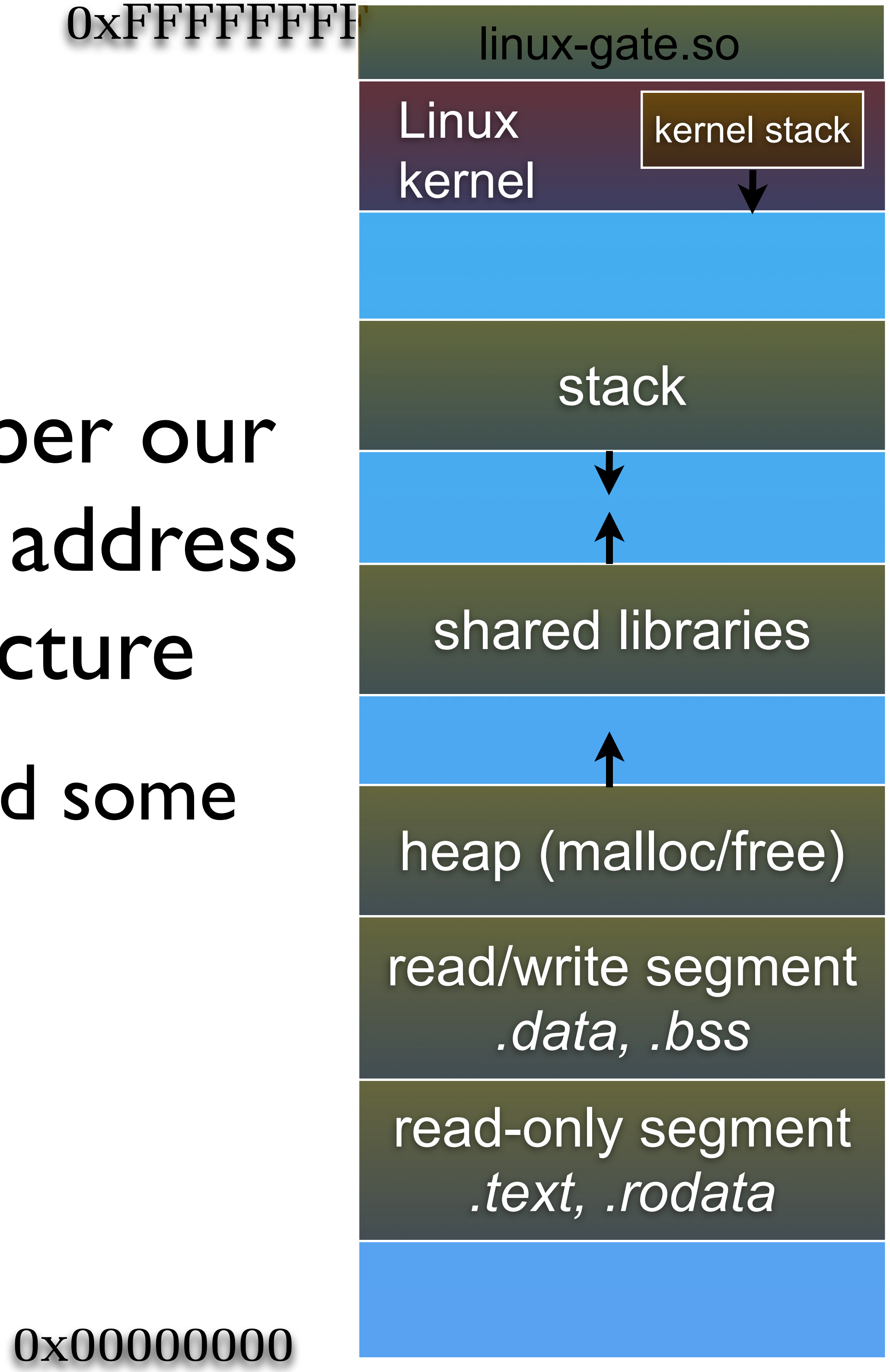
Details on x86 / Linux

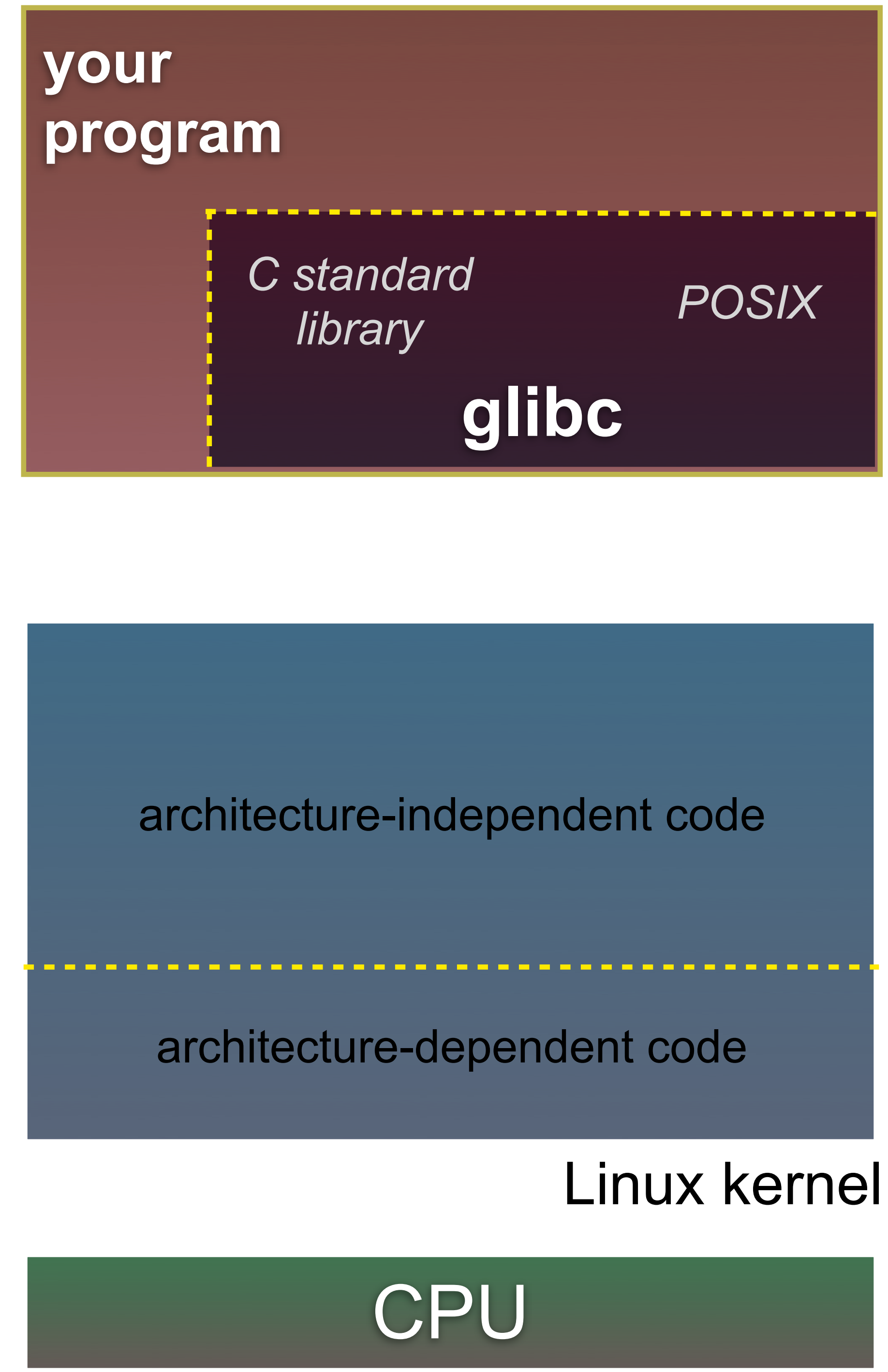
- Let's walk through how a Linux system call actually works
  - we'll assume 32-bit x86 using the modern SYSENTER / SYSEXIT x86 instructions



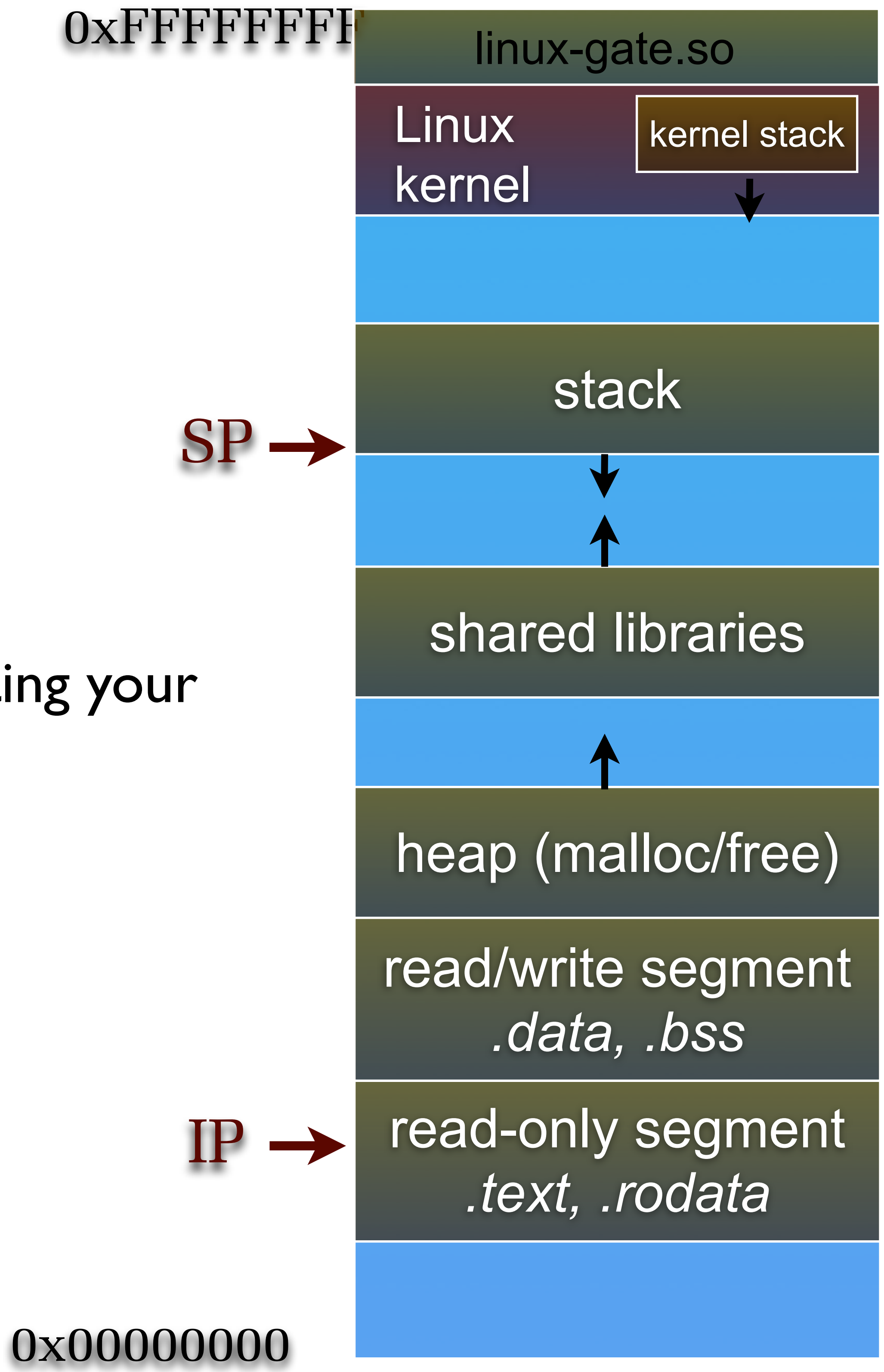
Details on x86 / Linux

- Remember our process address space picture
  - let's add some details

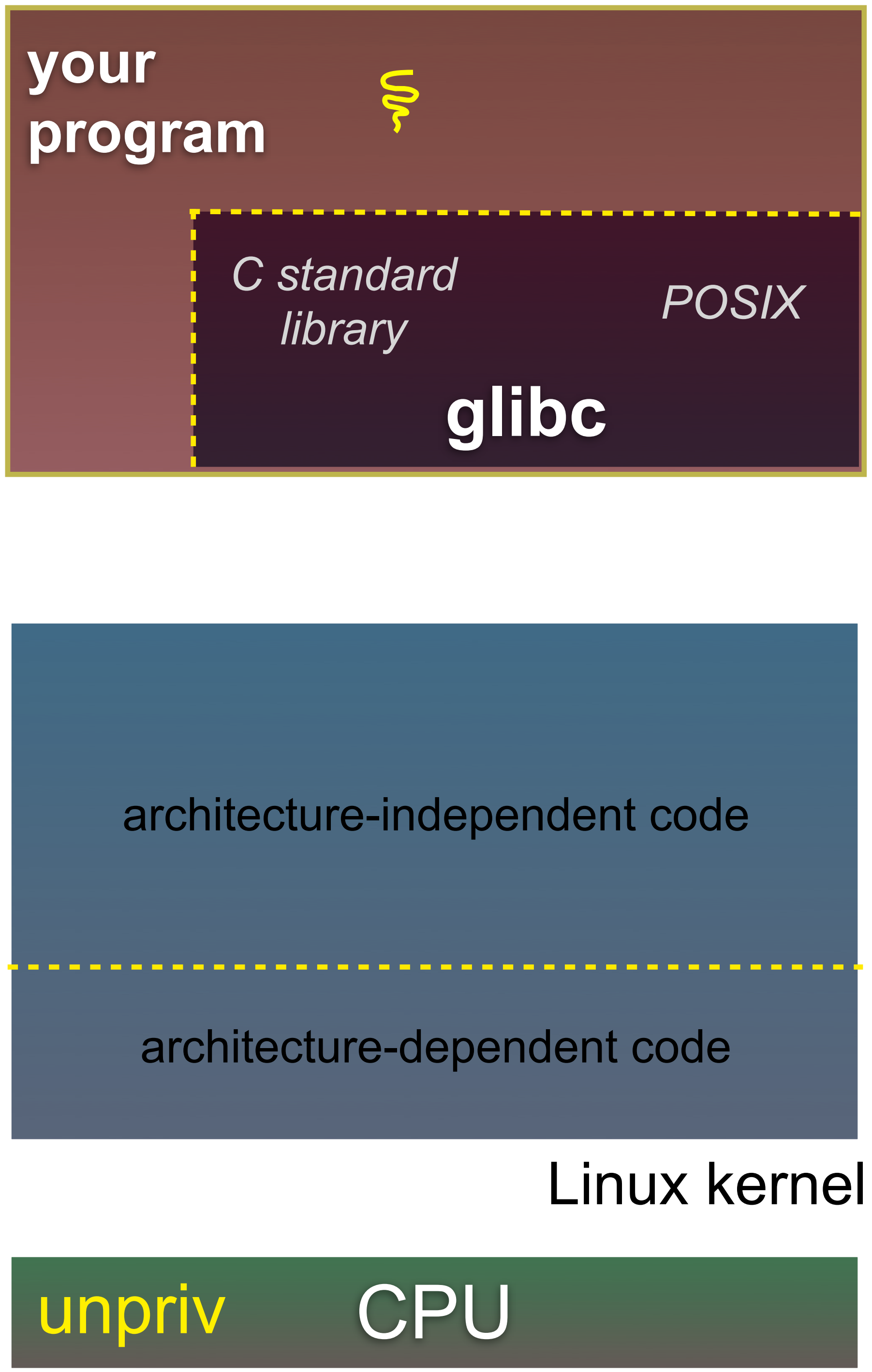




Details on x86 / Linux



process is executing your program code

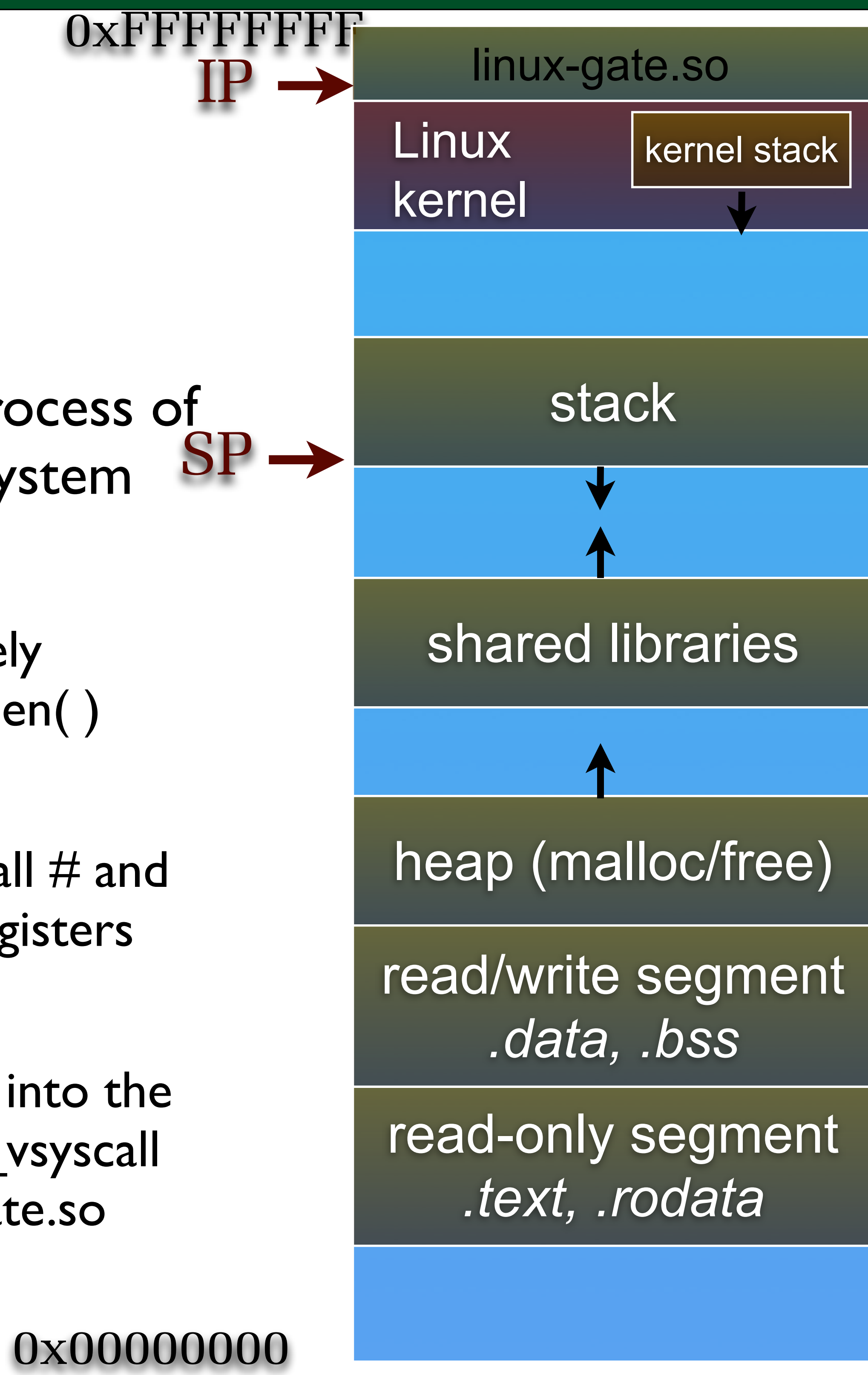


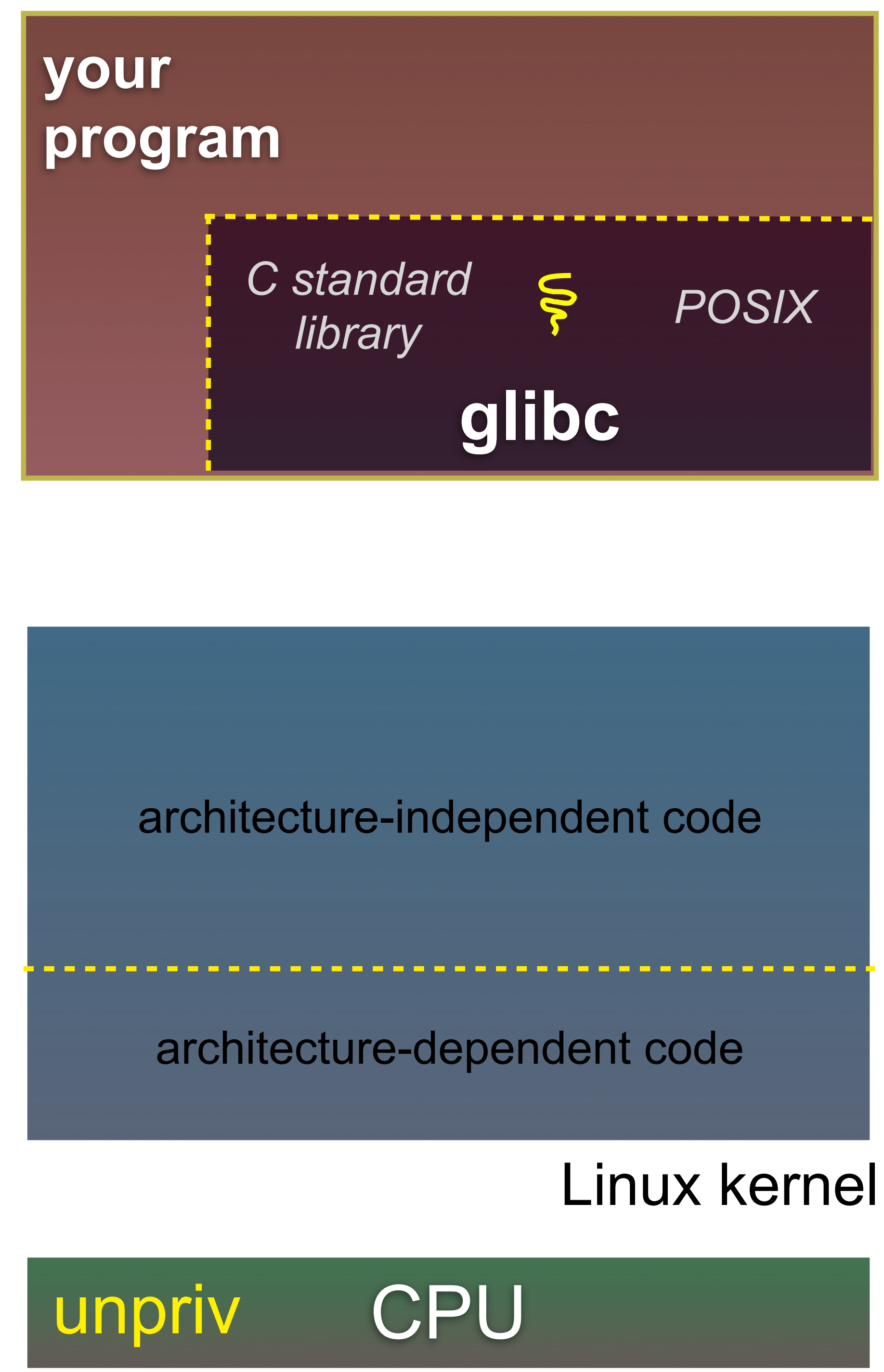
Details on x86 / Linux

UNIVERSITY OF OREGON

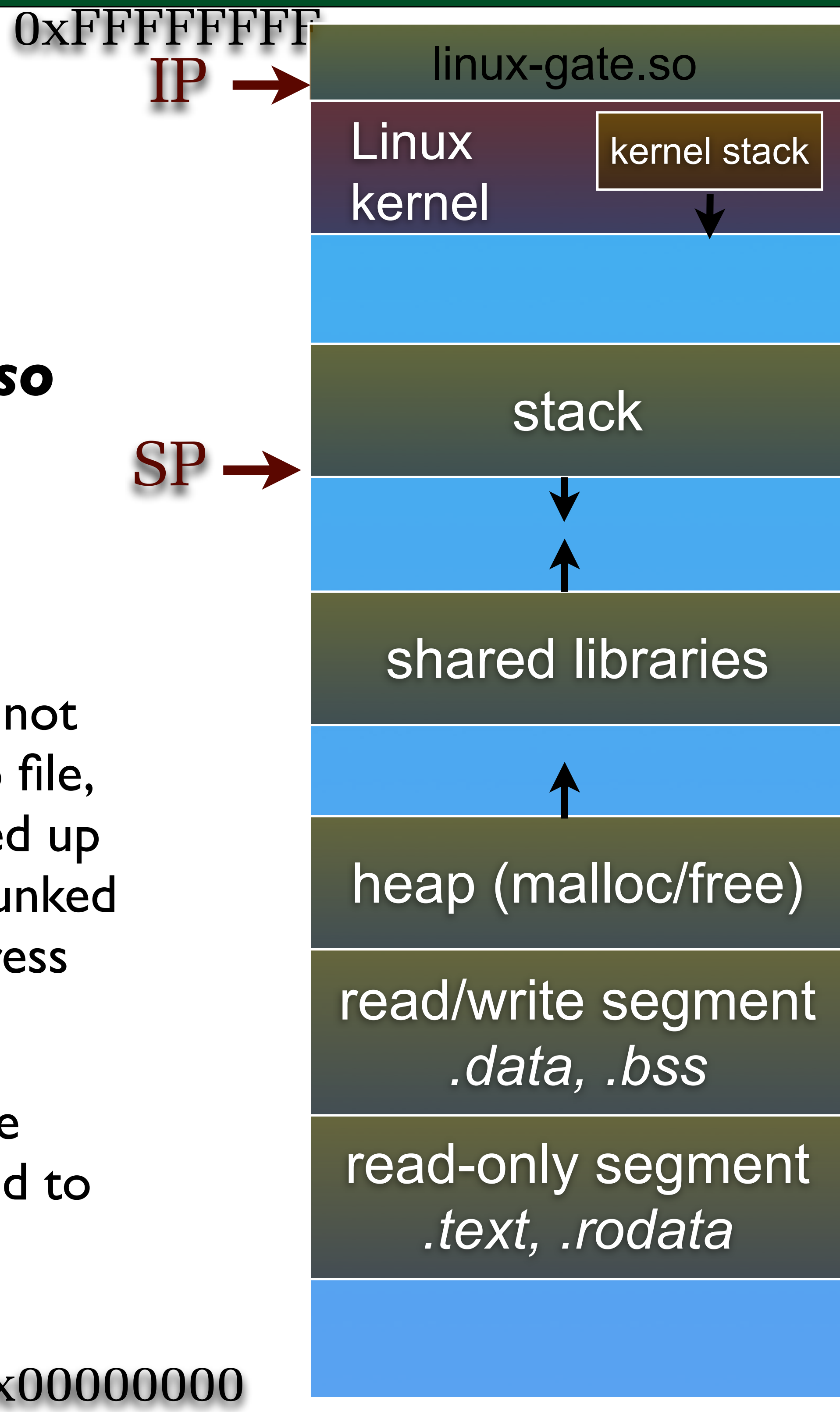
glibc begins the process of invoking a Linux system call

- glibc's fopen( ) likely invokes Linux's open( ) system call
- puts the system call # and arguments into registers
- uses the **call** x86 instruction to call into the routine __kernel_vsyscall located in linux-gate.so



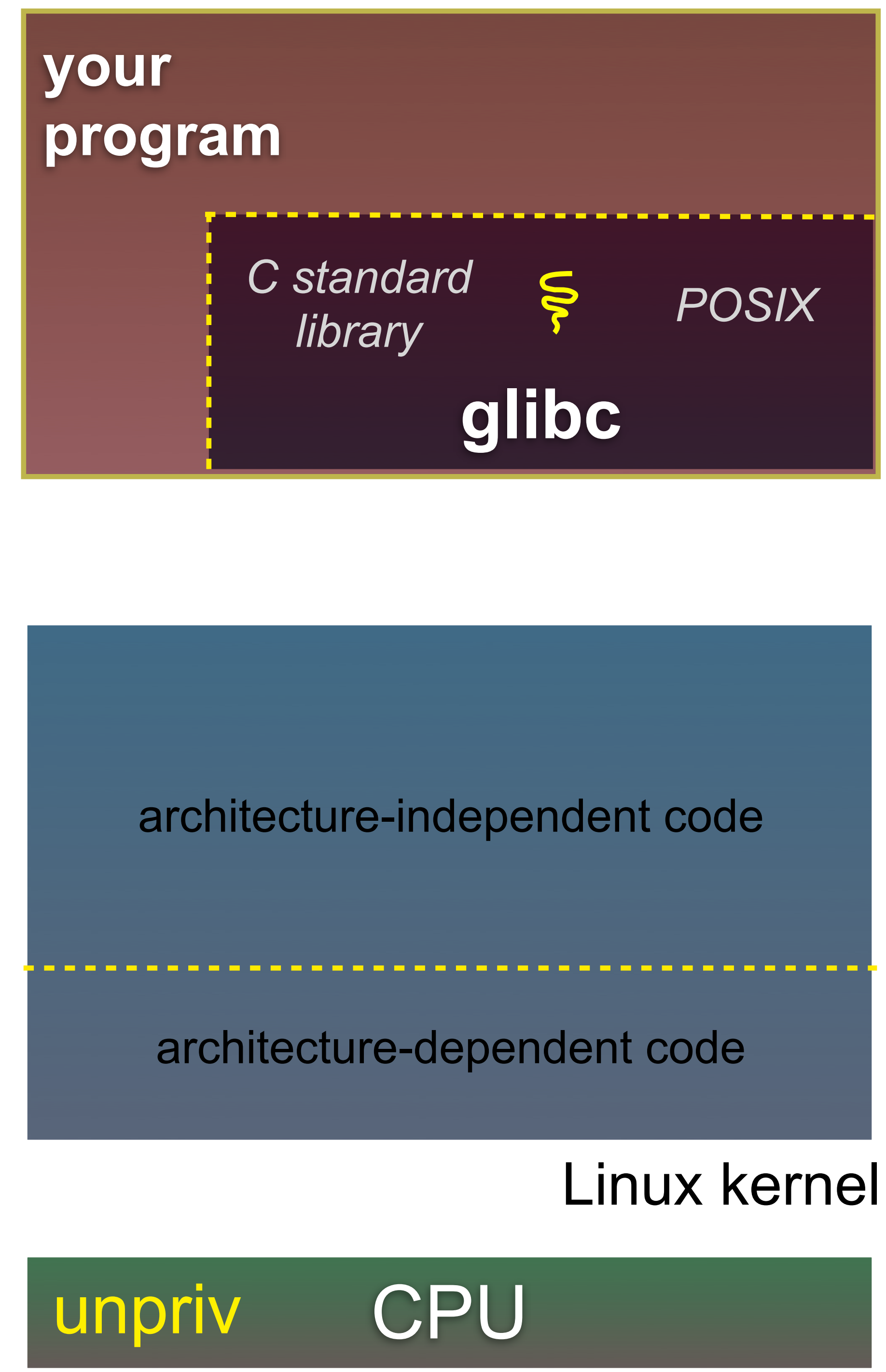


Details on x86 / Linux



linux-gate.so is a ***vdso***

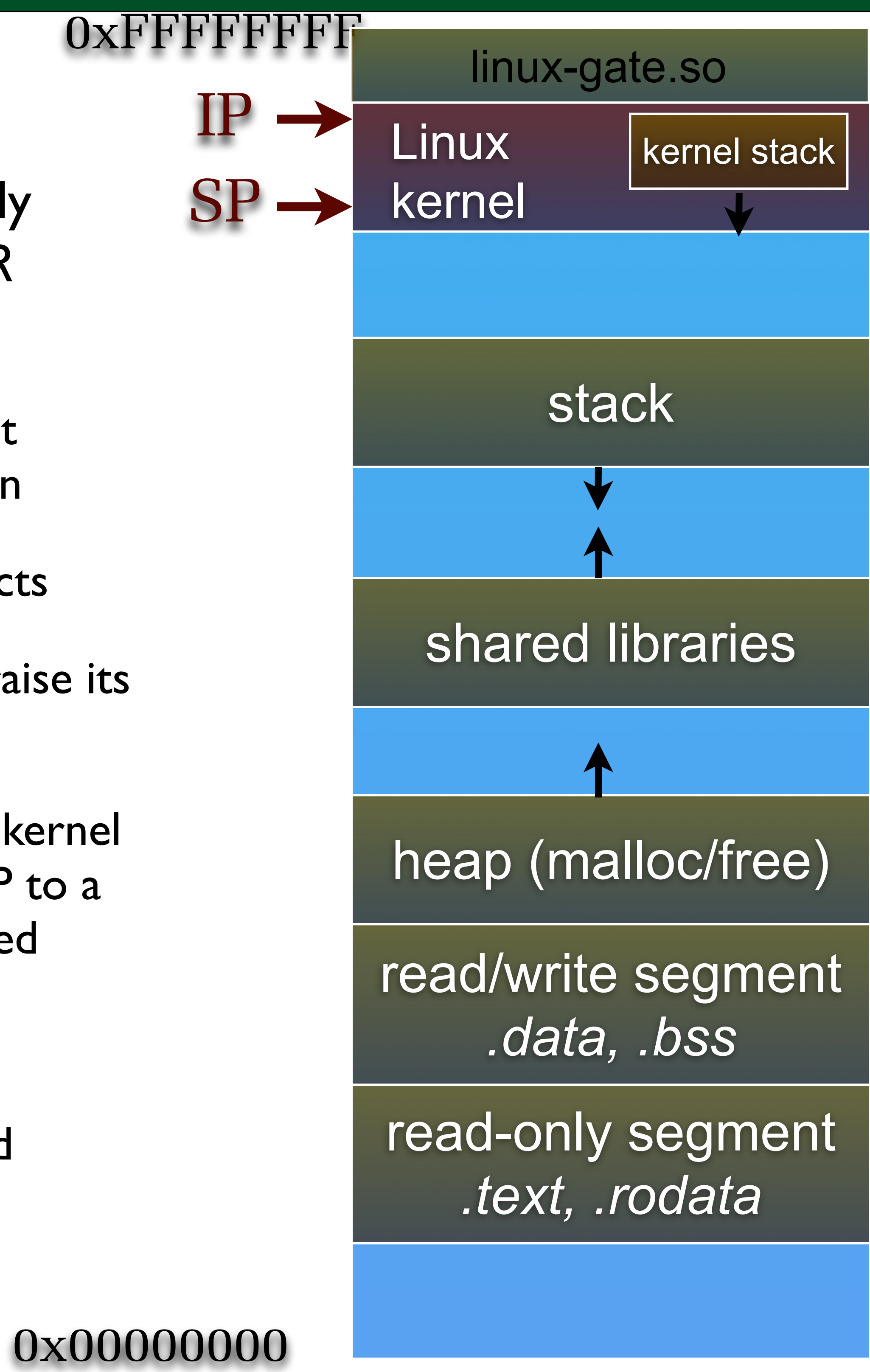
- a virtual dynamically linked shared object
- is a kernel-provided shared library, i.e., is not associated with a .so file, but rather is conjured up by the kernel and plunked into a process's address space
- provides the intricate machine code needed to trigger a system call

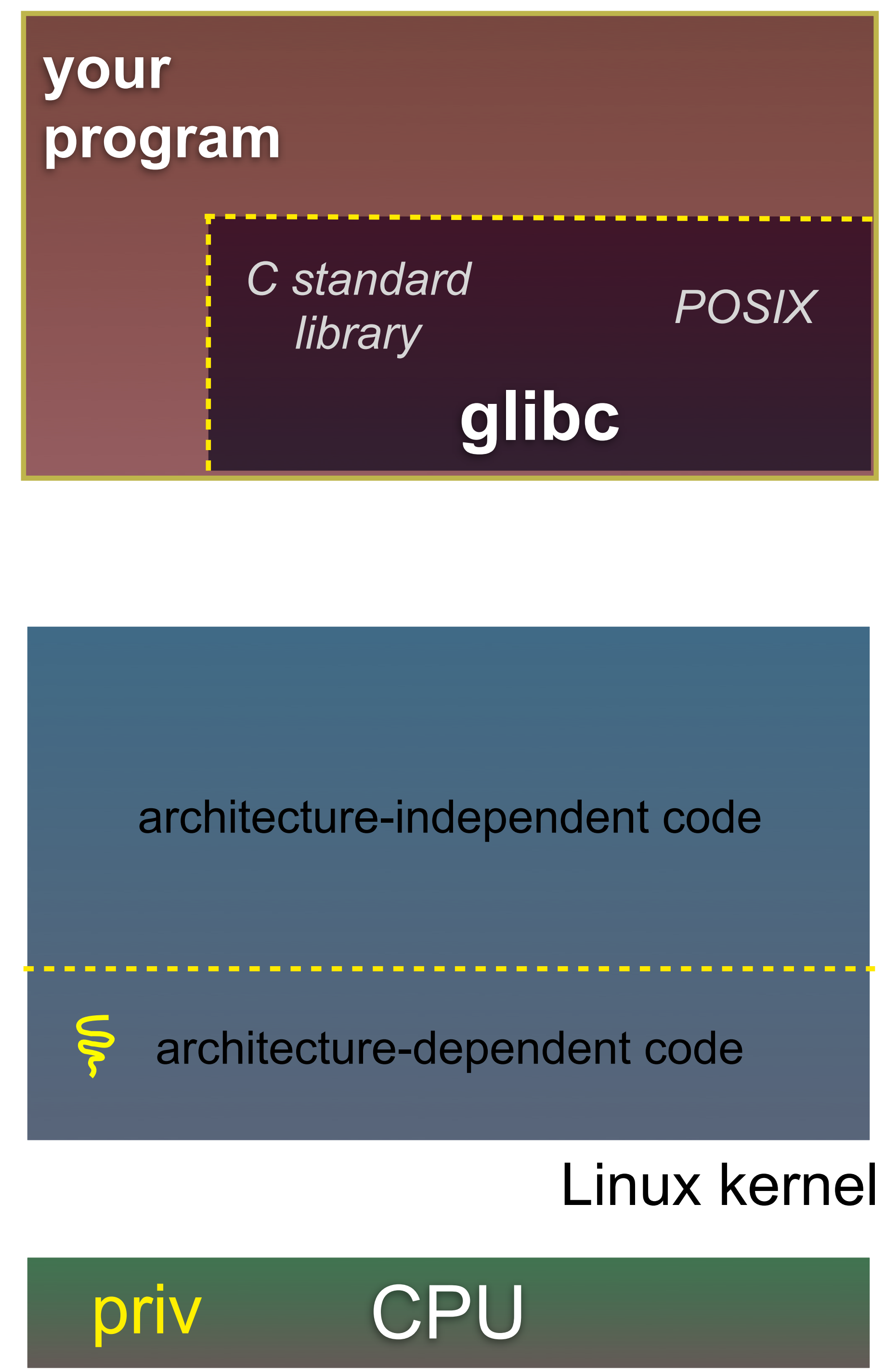


Details on x86 / Linux

linux-gate.so eventually invokes the *SYSENTER* x86 instruction

- *SYSENTER* is x86's "fast system call" instruction
- it has several side-effects
  - causes the CPU to raise its privilege level
  - traps into the Linux kernel by changing the SP, IP to a previously determined location
  - changes some segmentation related registers



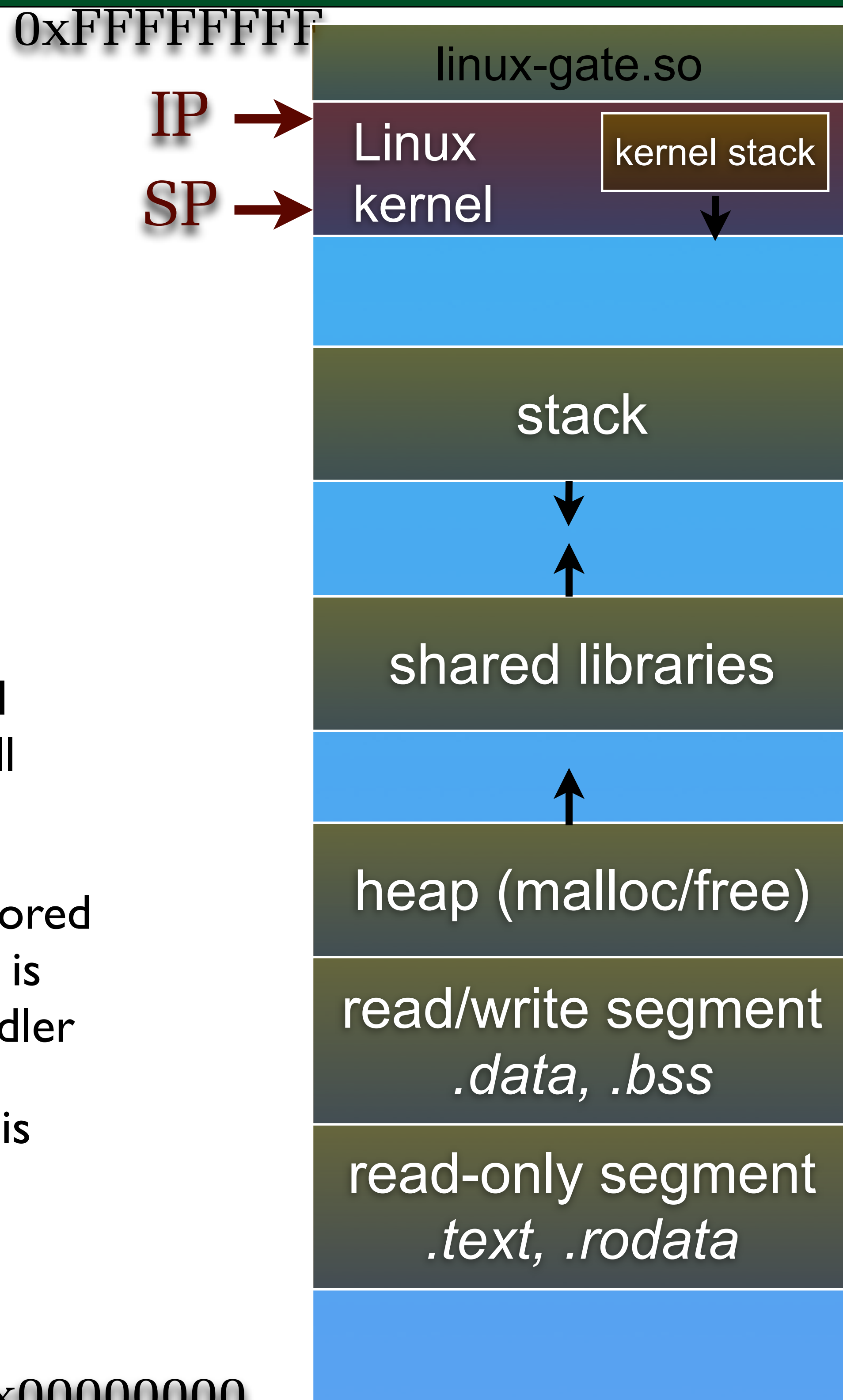


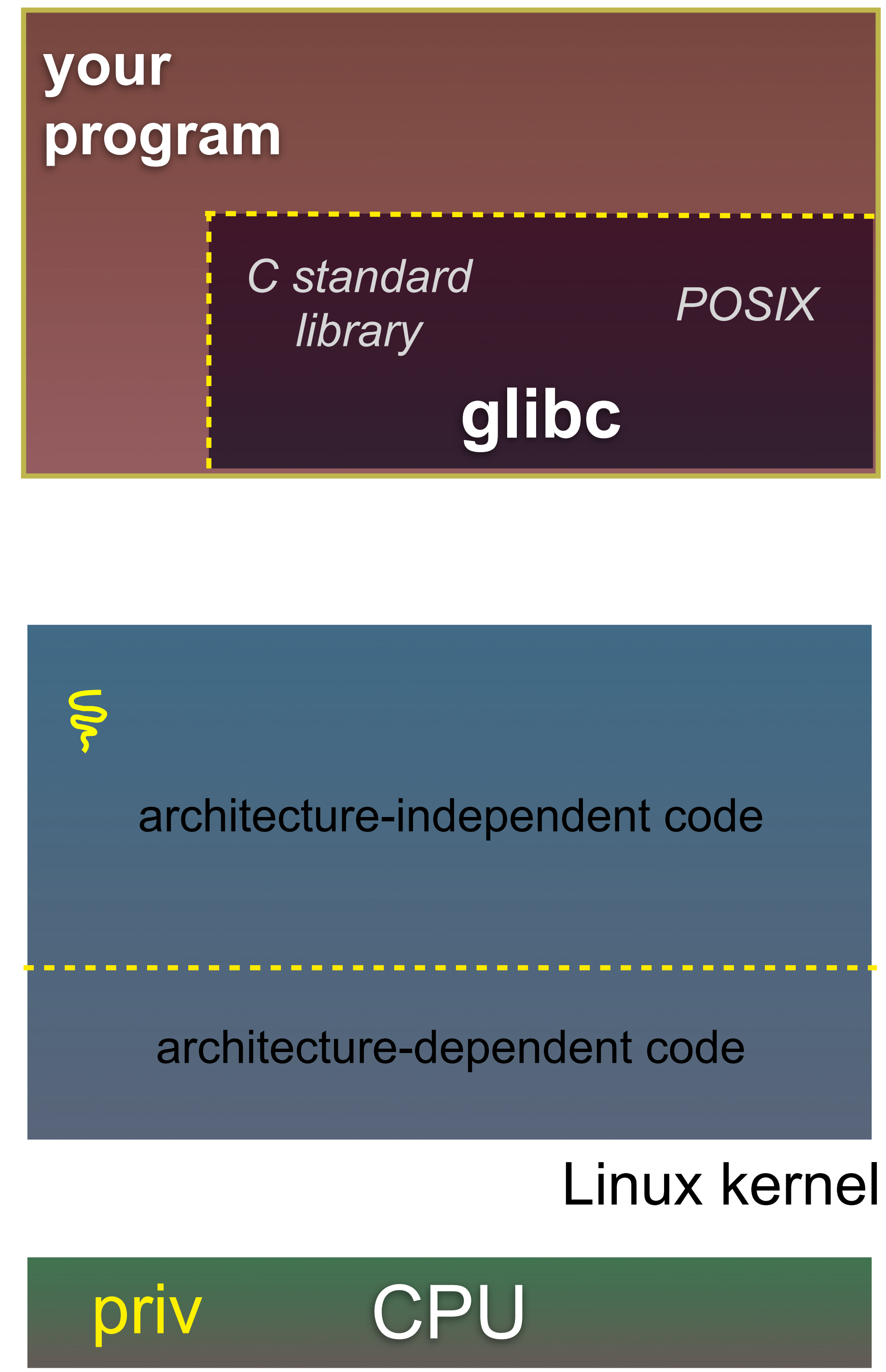
Details on x86 / Linux

UNIVERSITY OF OREGON

The kernel begins executing code at the *SYSENTER* entry point

- is in the architecture-dependent part of Linux
- its job is to:
  - look up the system call number in a system call dispatch table
  - call into the address stored in that table entry; this is Linux's system call handler
  - for *open()*, the handler is named *sys_open*, and is system call #5

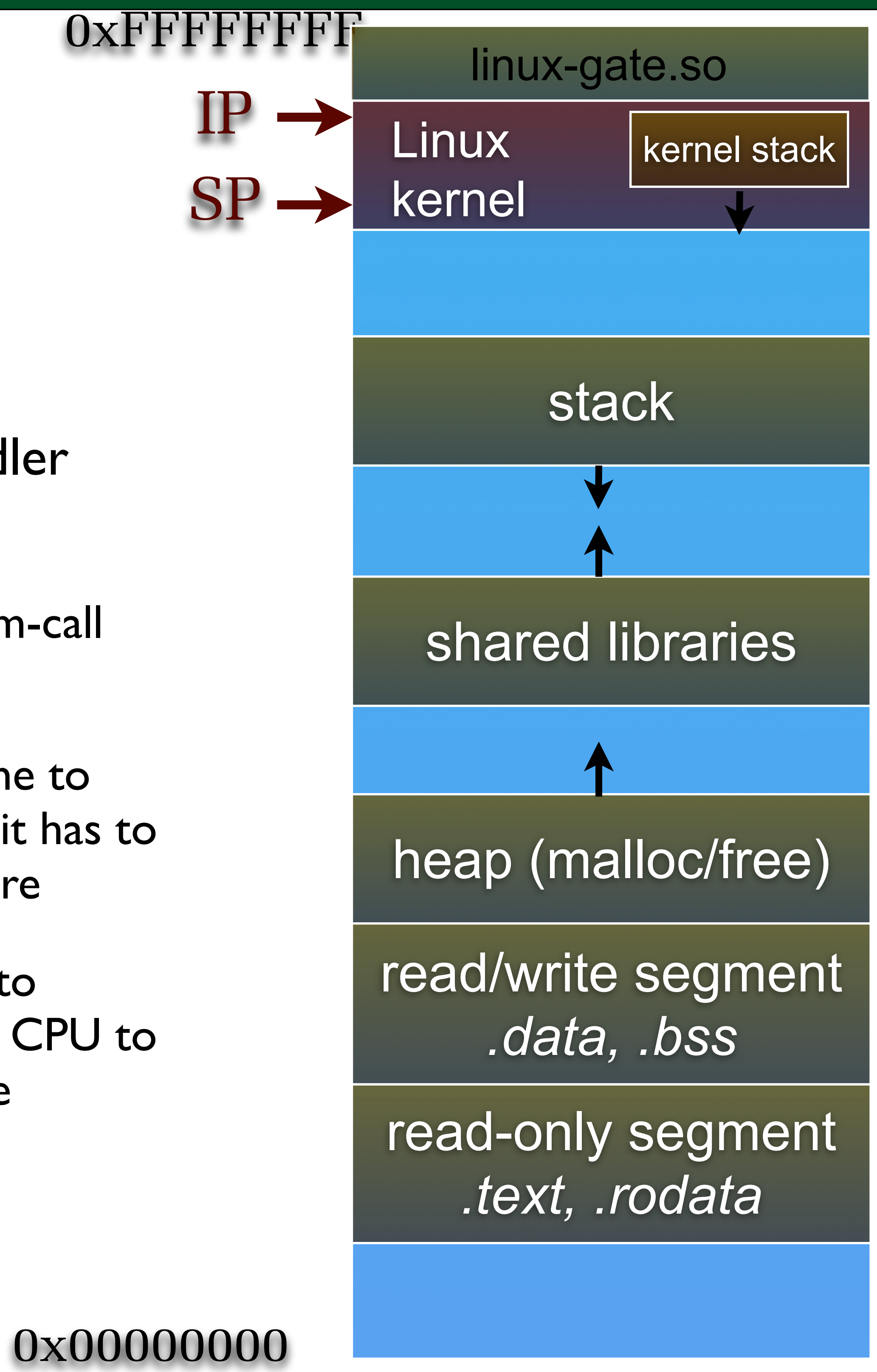


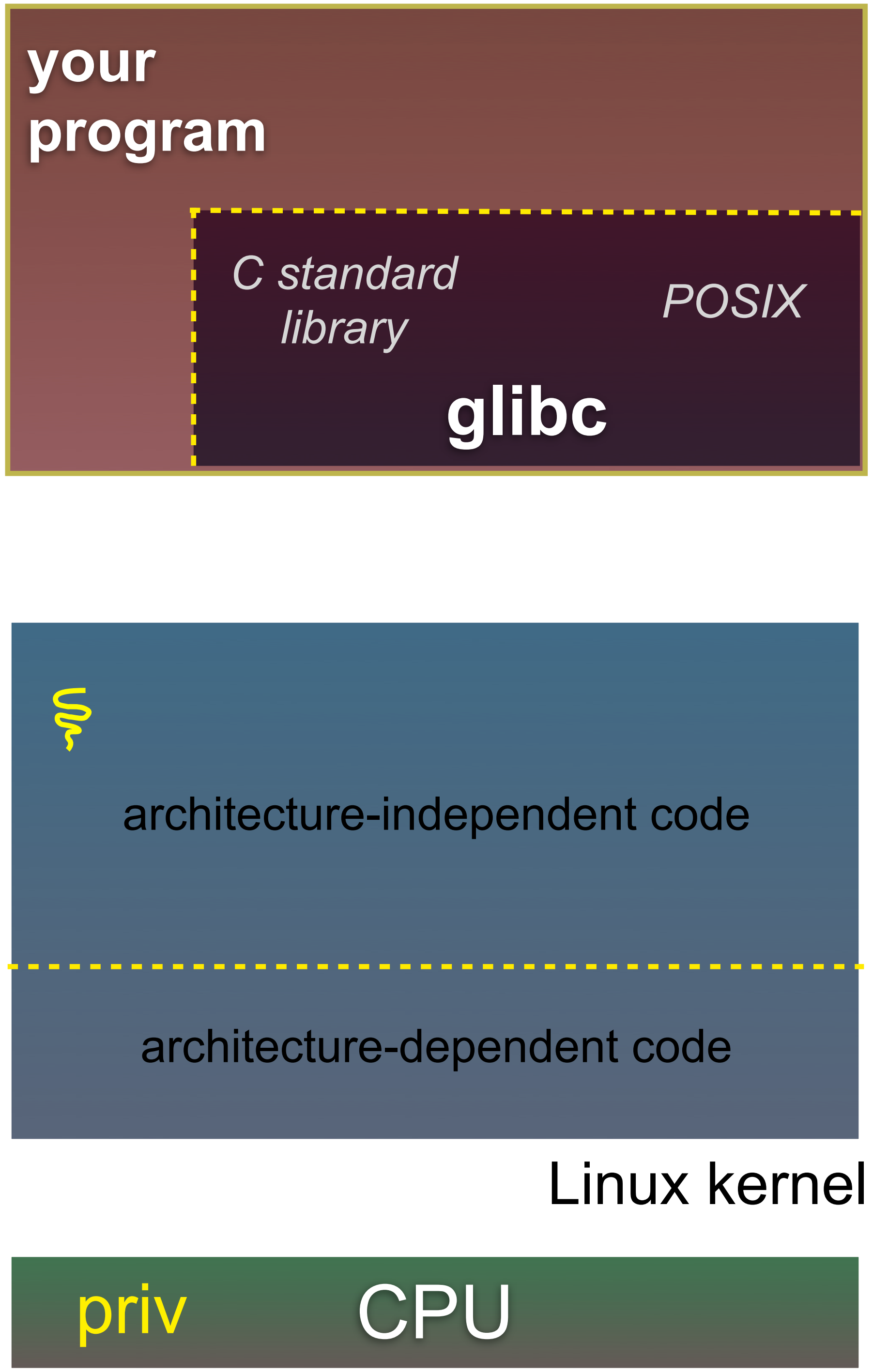


Details on x86 / Linux

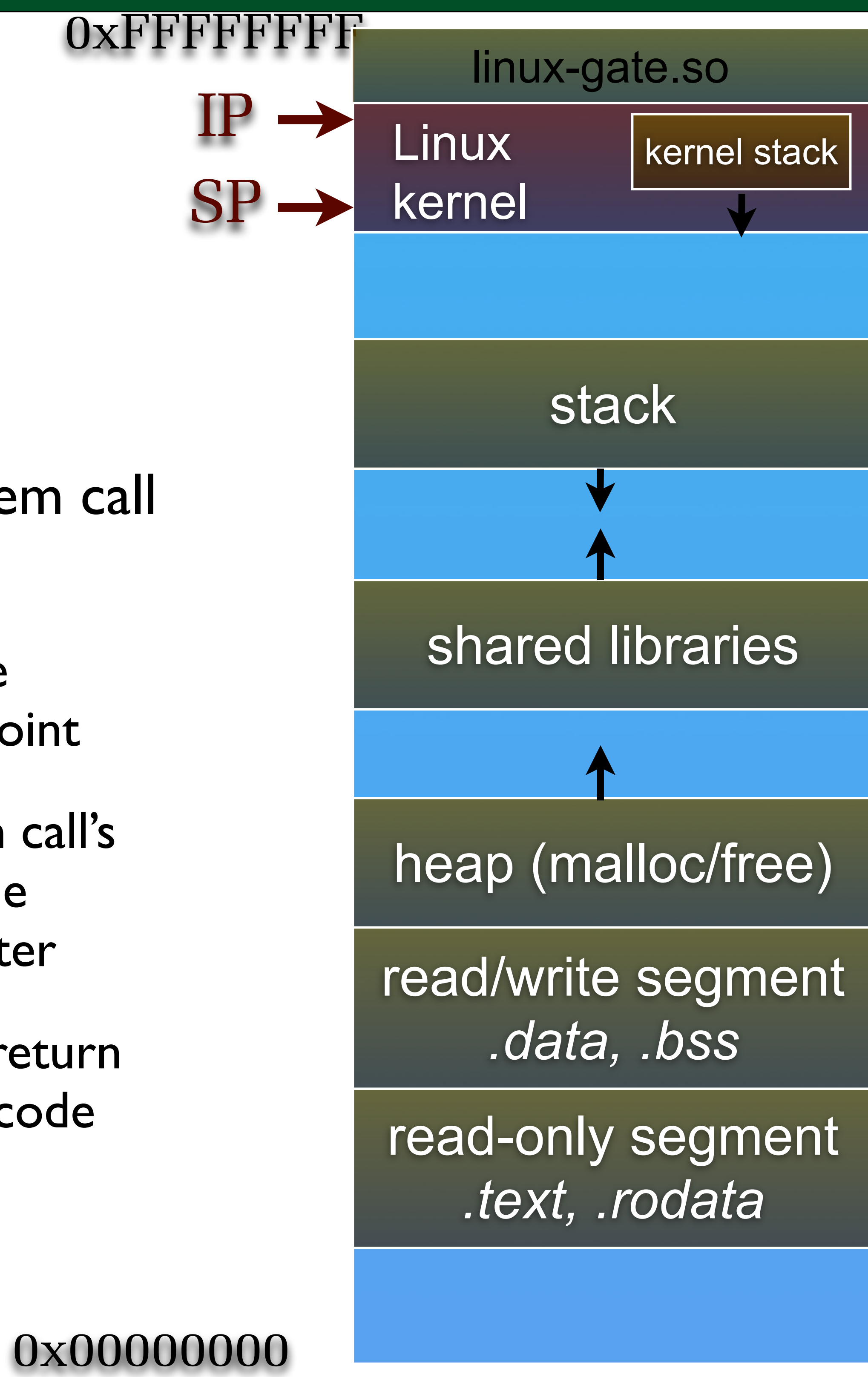
The system call handler executes

- what it does is system-call specific, of course
- it may take a long time to execute, especially if it has to interact with hardware
  - Linux may choose to context switch the CPU to a different runnable process



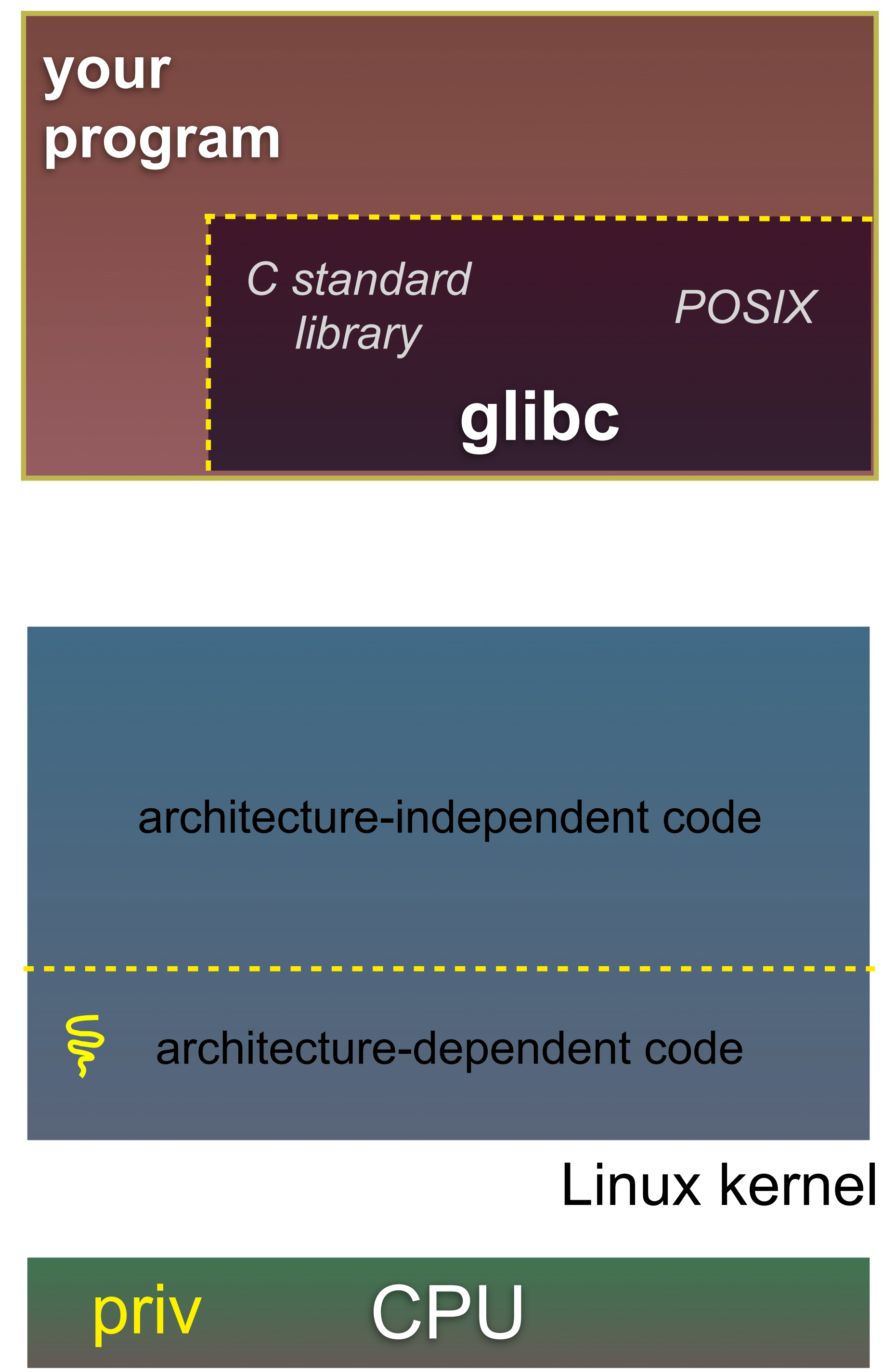


Details on x86 / Linux



Eventually, the system call handler finishes

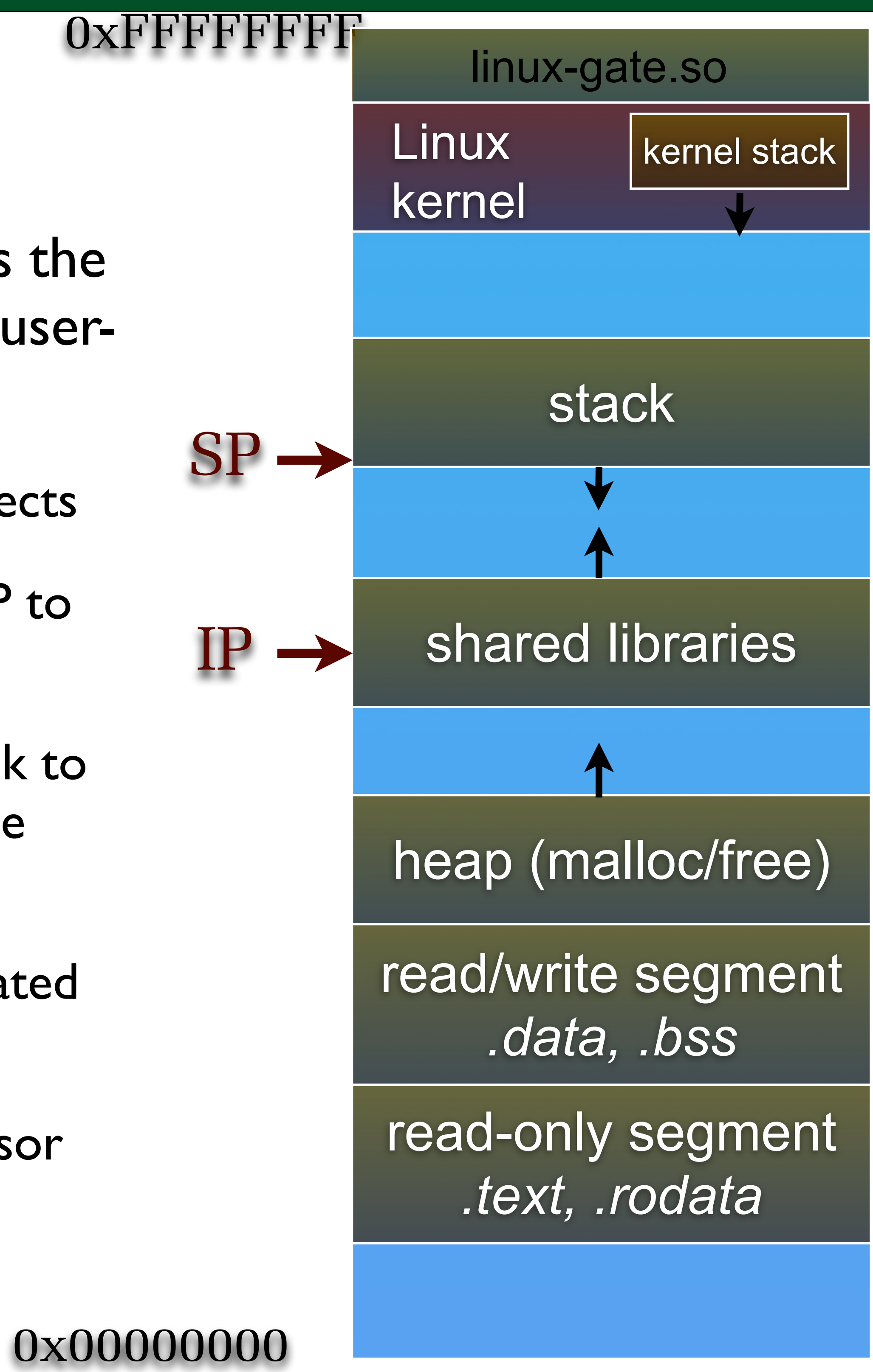
- returns back to the system call entry point
  - places the system call's return value in the appropriate register
  - calls *SYSEXIT* to return to the user-level code

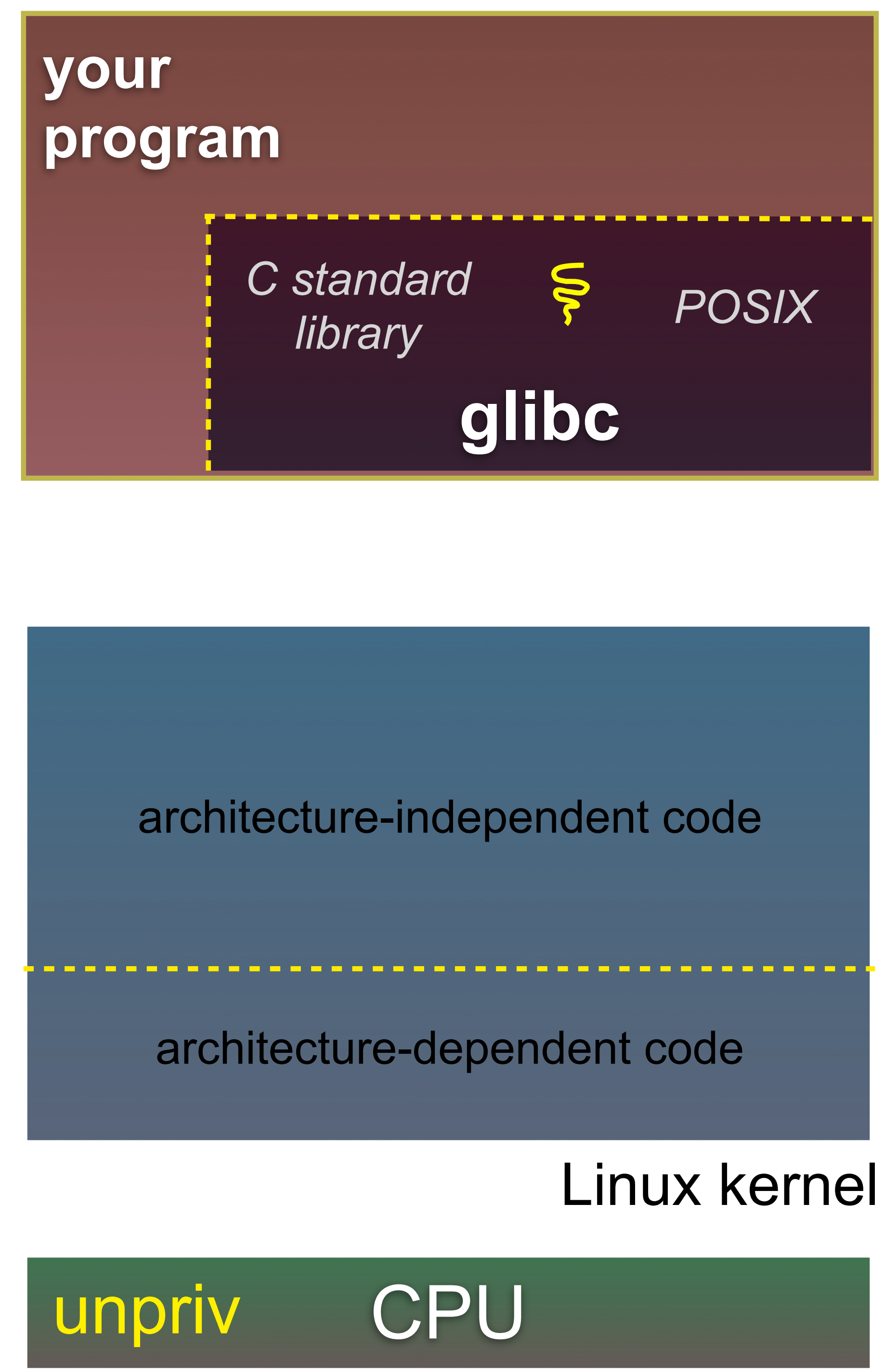


Details on x86 / Linux

SYSEXIT transitions the processor back to user-mode code

- has several side-effects
  - restores the IP, SP to user-land values
  - sets the CPU back to unprivileged mode
  - changes some segmentation related registers
- returns the processor back to glibc

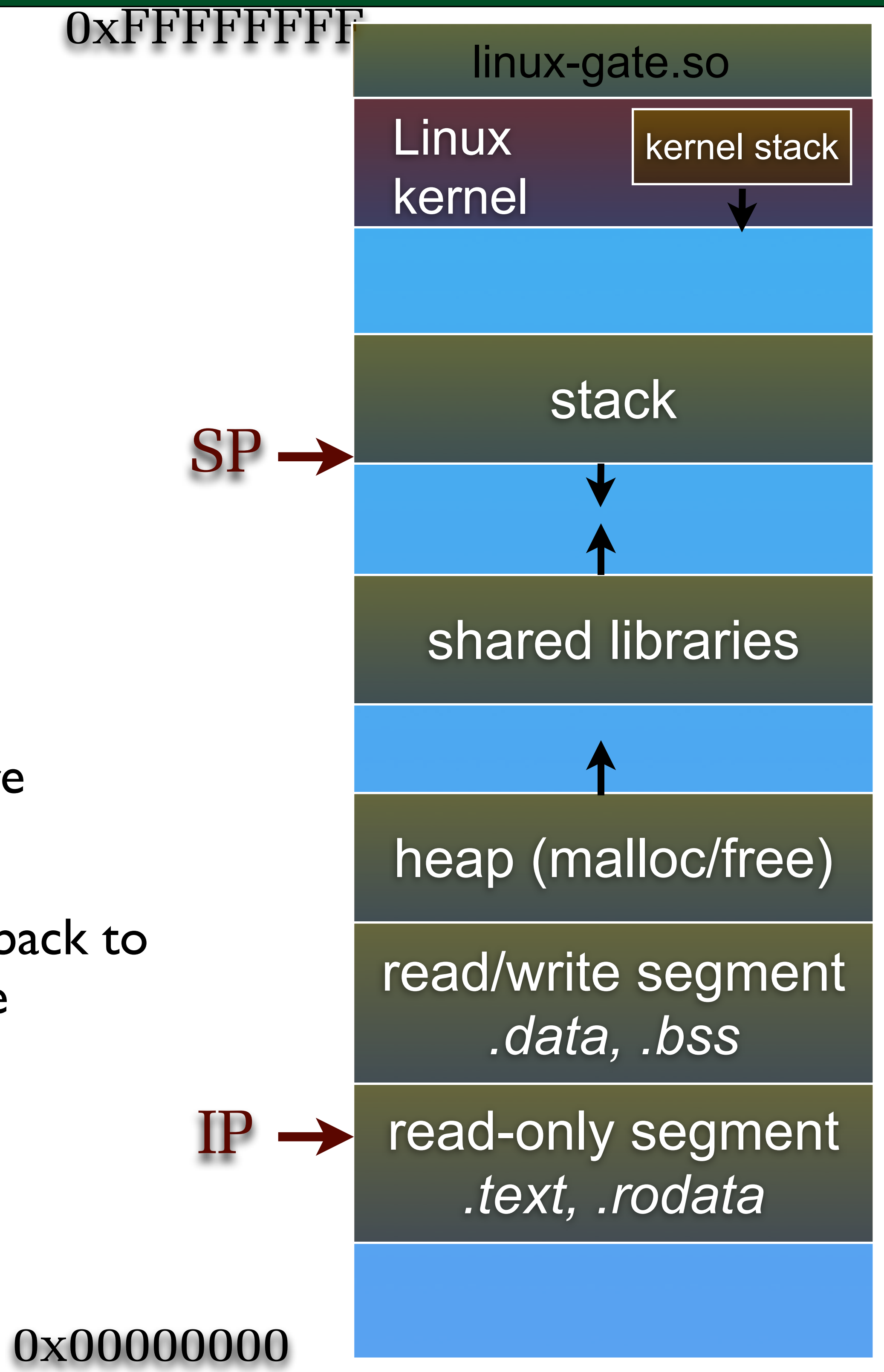


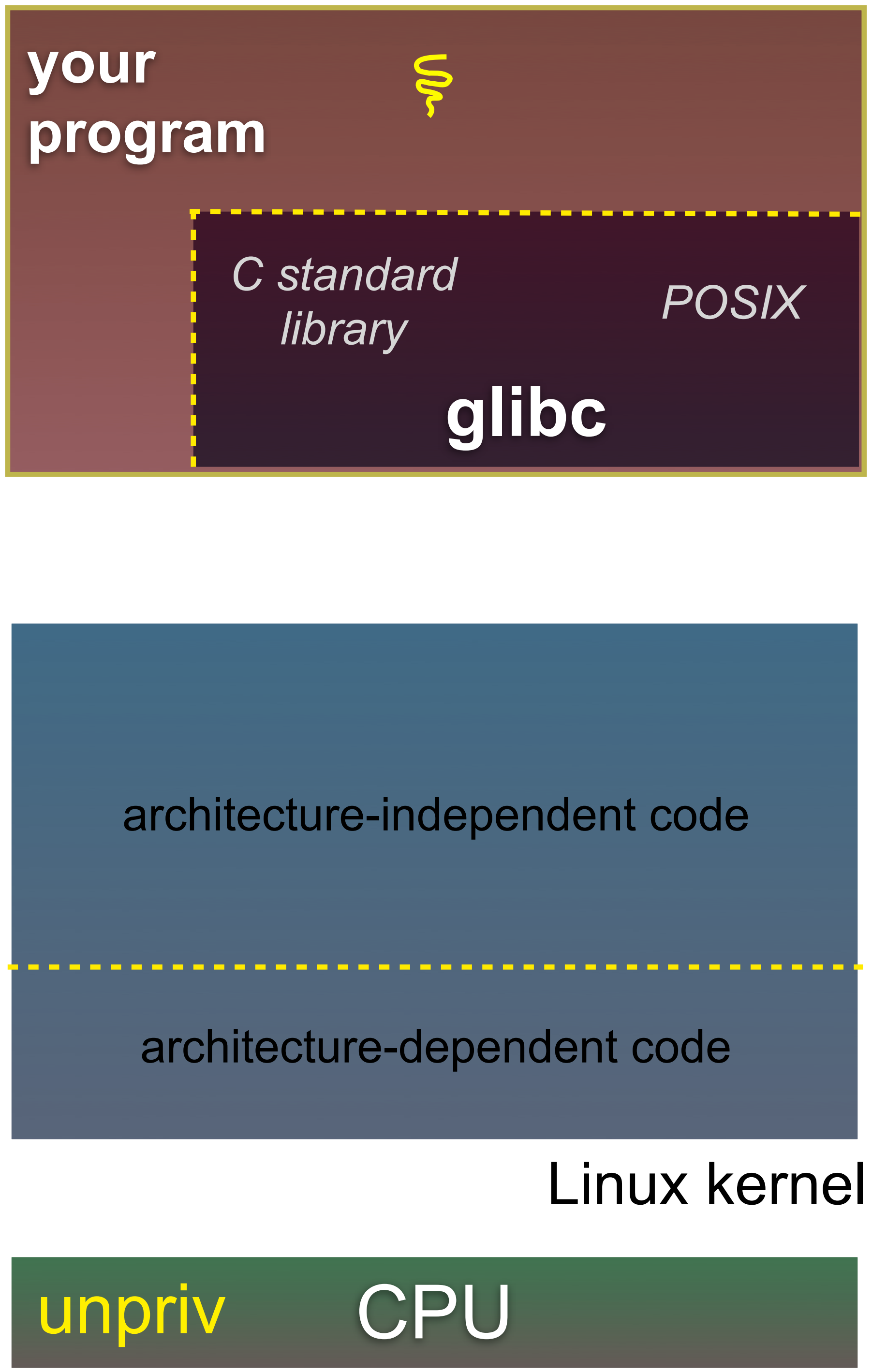


Details on x86 / Linux

glibc continues to execute

- might execute more system calls
- eventually returns back to your program code





Relocatable Memory

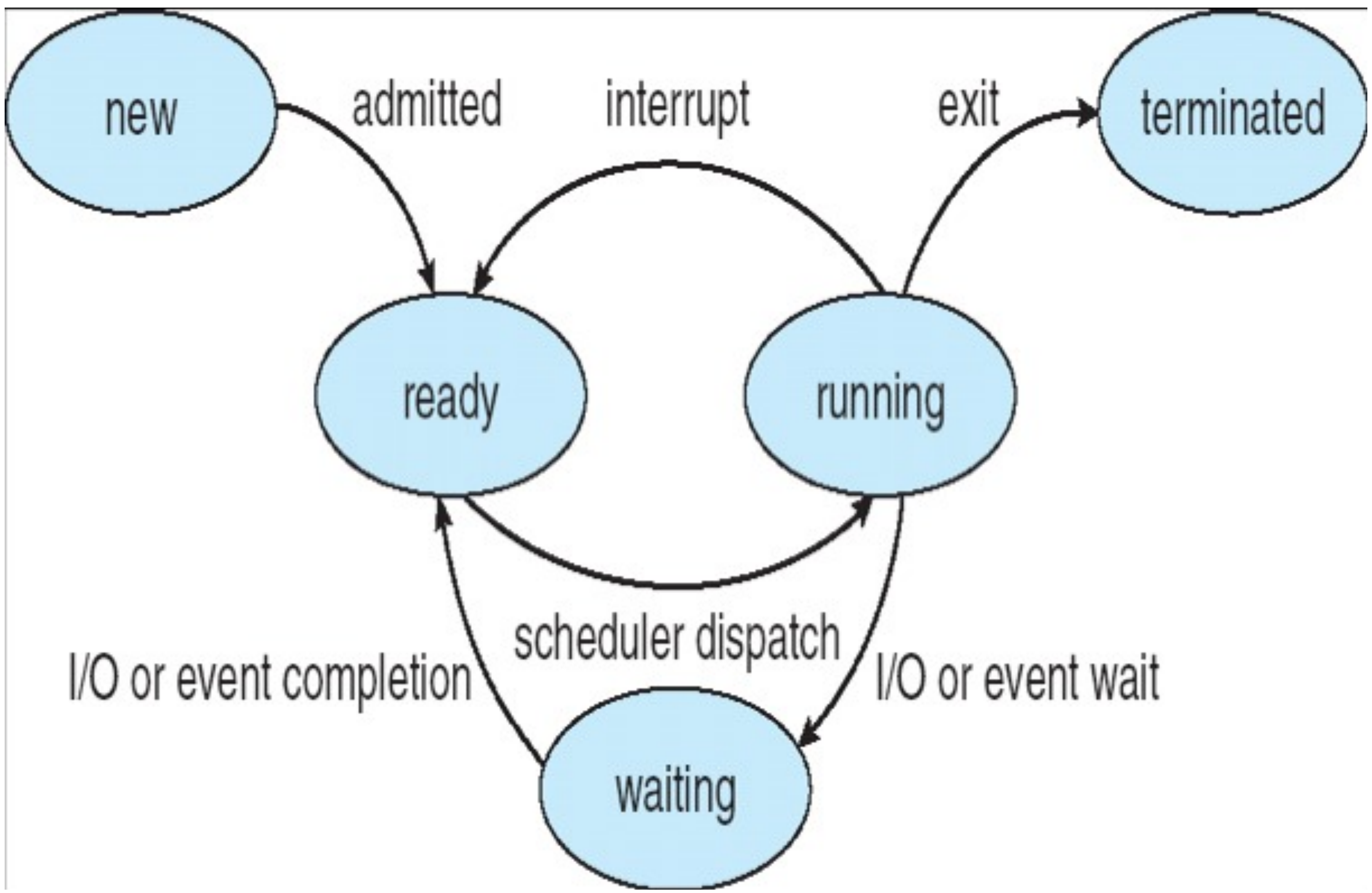
- Mechanism that enables the OS to place a program in an arbitrary location in memory
  - Gives the programmer the impression that they own the processor
- Program is loaded into memory at program-specific locations
  - Need virtual memory to do this
- Also, may need to share program code across processes

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*

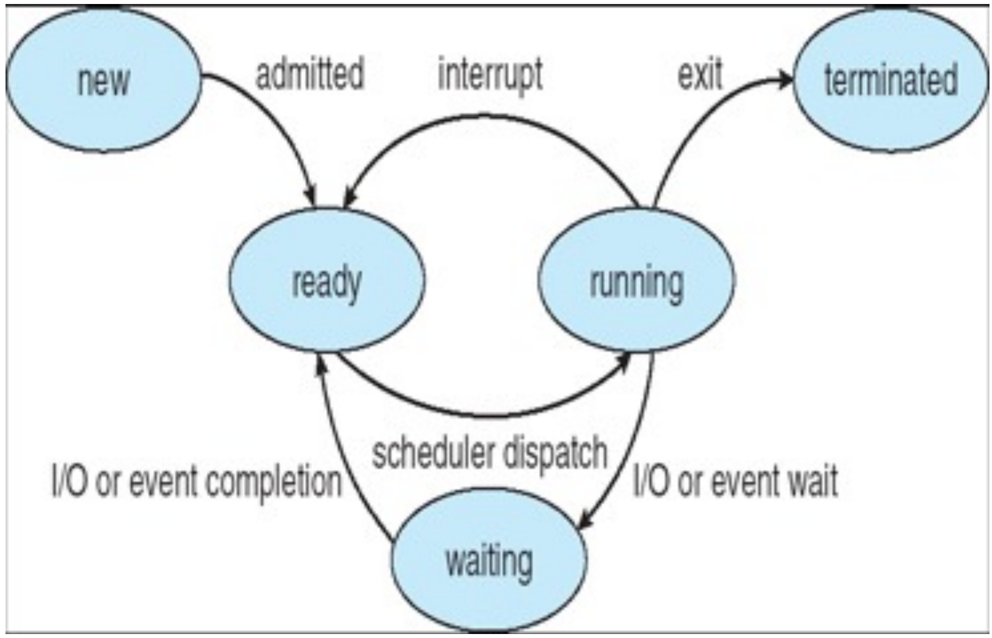
Scheduling Processes

- Processes transition among *execution states*



Process States

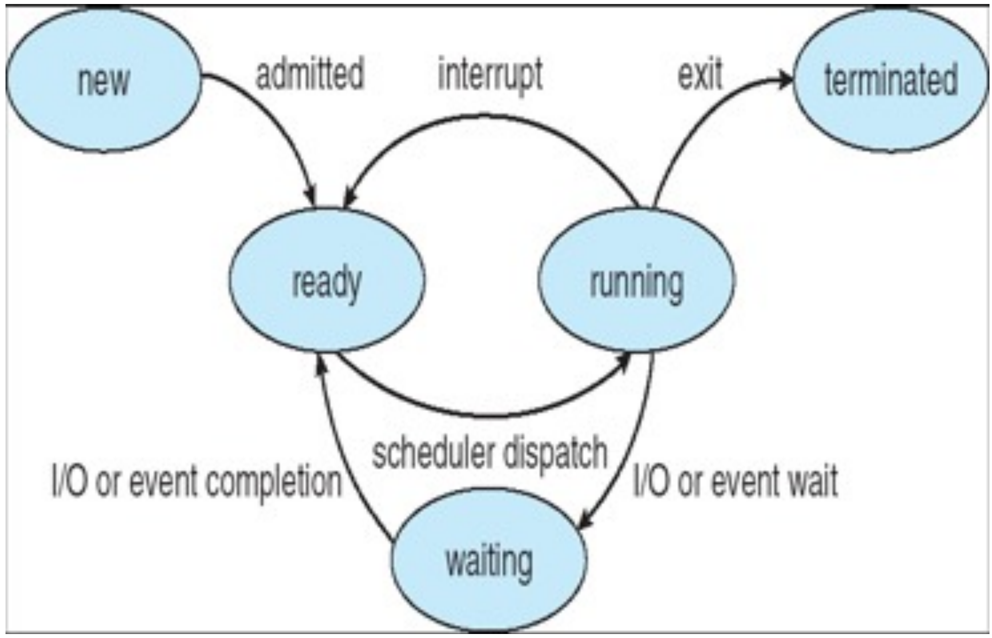
- Running
  - Running == in processor and in memory with all resources
- Ready
  - Ready == in memory with all resources, waiting for dispatch
- Waiting
  - Waiting == waiting for some event to occur



State Transitions

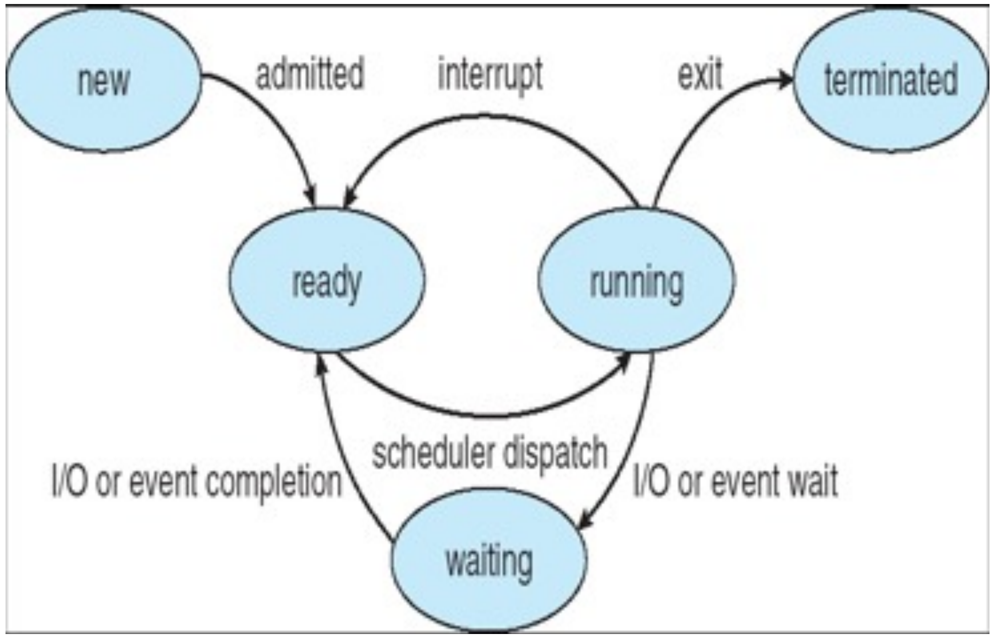
UNIVERSITY OF OREGON

- 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



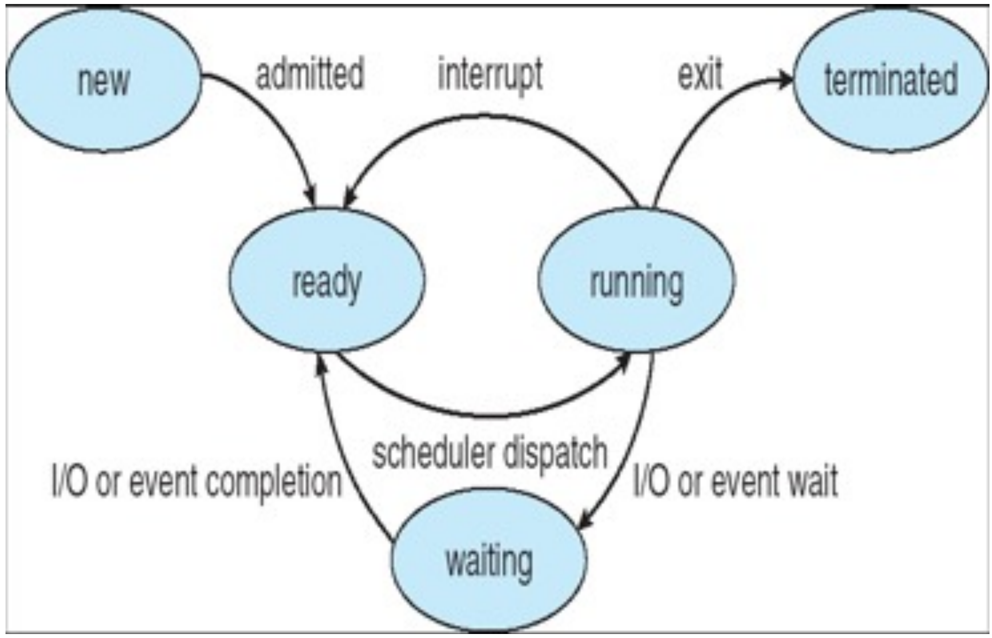
State Transitions: Page Fault Handling

- 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…

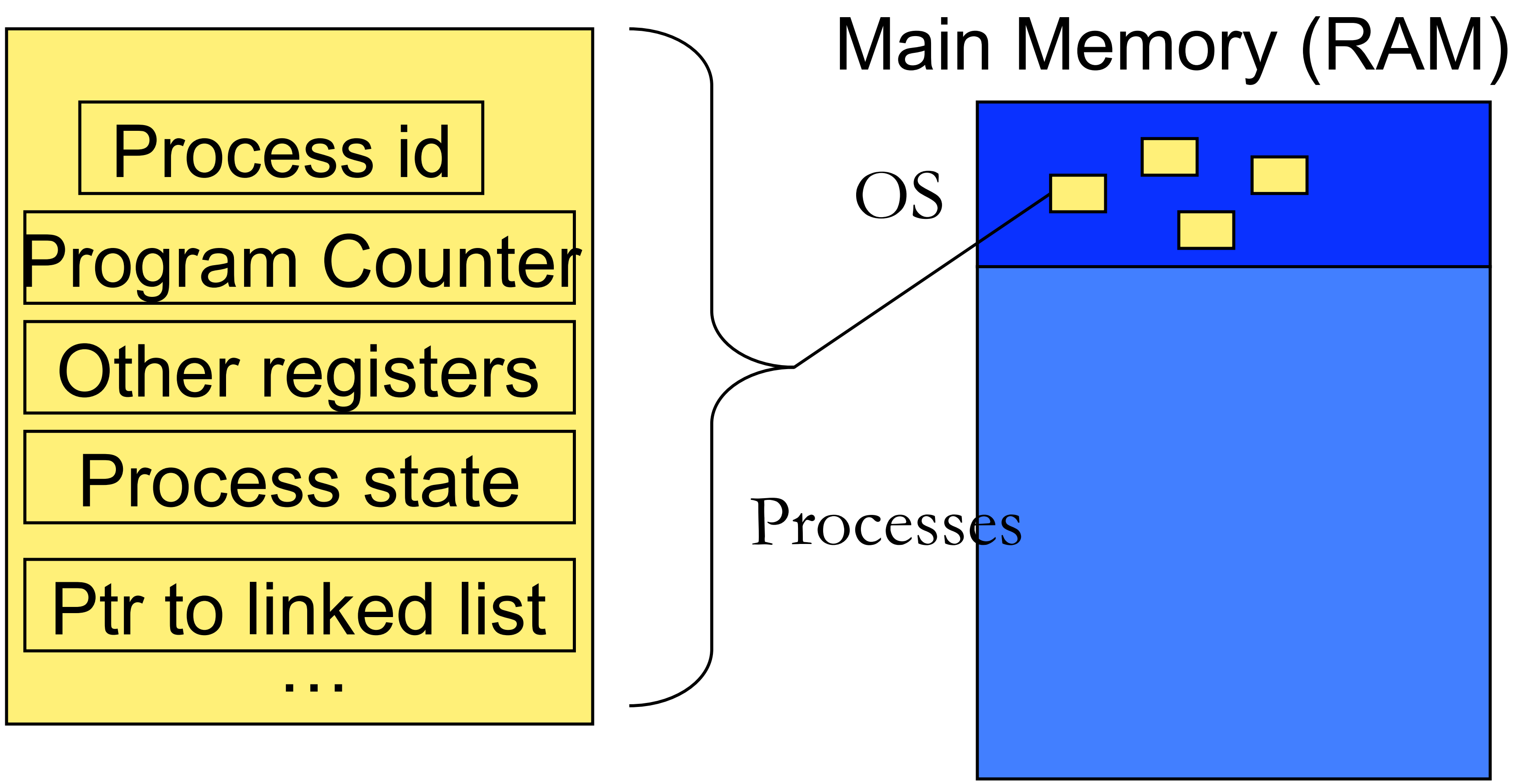


State Transitions: Other Issues

- 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…

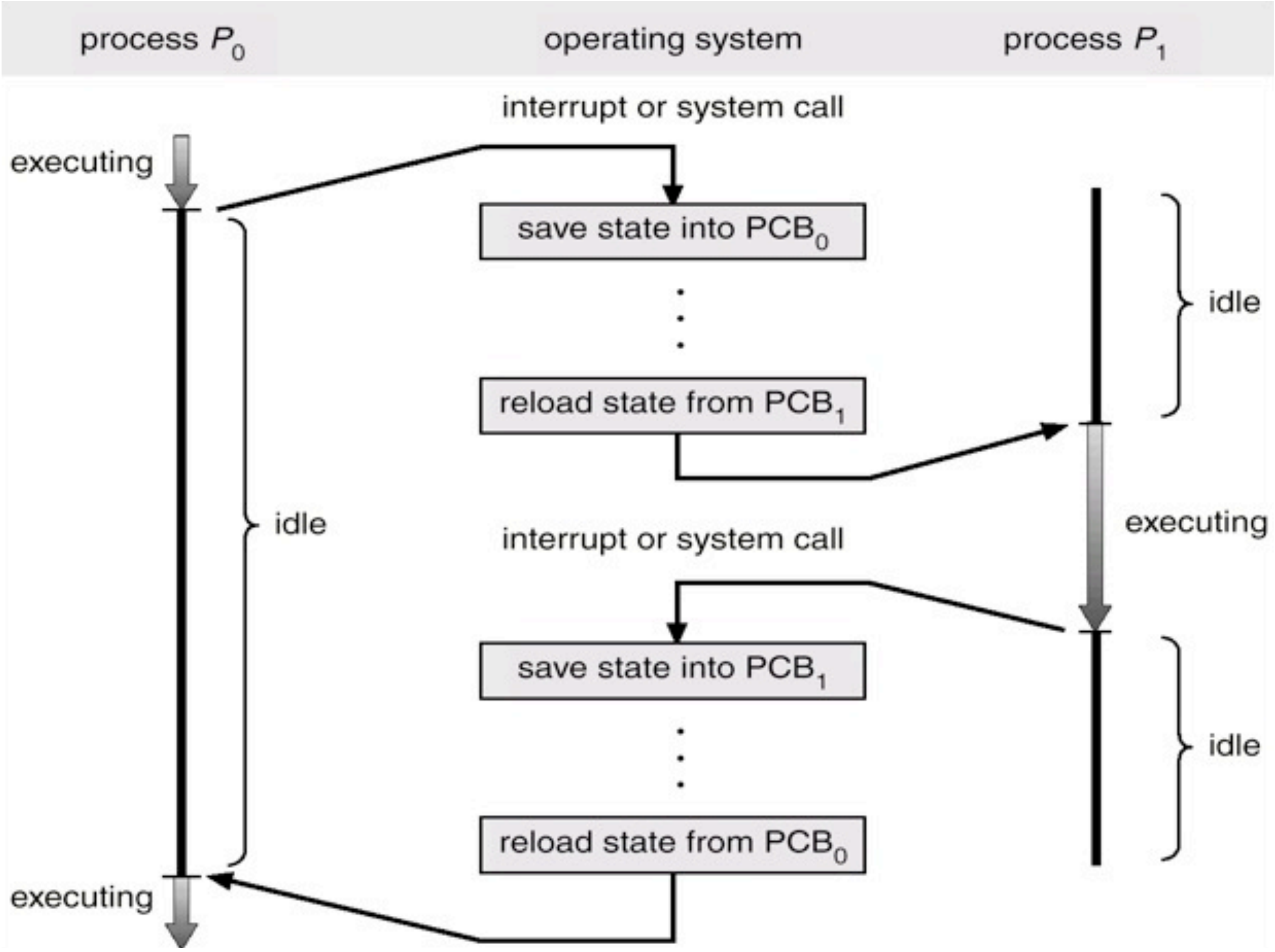
/proc File System

- 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

- 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

UNIVERSITY OF OREGON

- 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