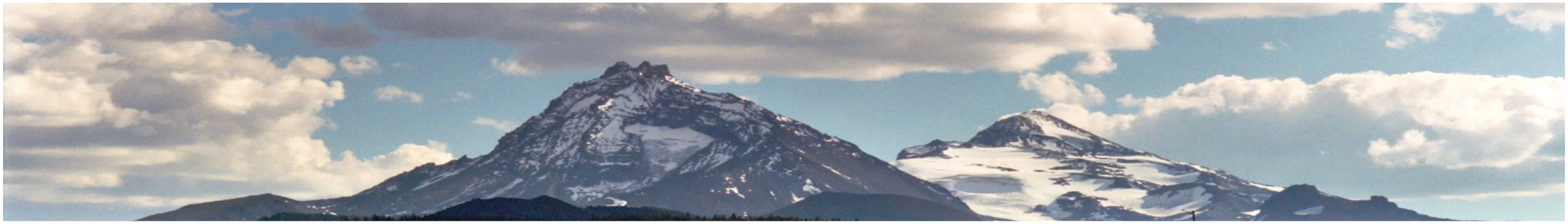


CS 415
Operating Systems
Processes

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

UNIVERSITY OF OREGON

Logistics

- Project 1 posted
- Labs
  - Intended to support projects
  - Lab 2 posted on Monday
- Read OSC Chapter 3
- Make use of Canvas discussion
- 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

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
  - User programs 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

- A process is a ***program in execution***
  - Process execution can result in more processes being created
- What makes up a process?
  - Program executable code (called *image* or *text*)
  - CPU state (program counter, CPU registers, …)
  - Stack containing temporary data (as a result of function calls)
    - function parameters, return addresses, local variables, …
    - called *stack frames*
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

Process Concept

- A process is a ***program in execution***
  - Process execution can result in more processes being created
- What makes up a process?
  - Code –instructions needed to execute
  - Data – data needed to complete the task
  - Execution context – what we are executing right now
  - Management information – what files are opened, what IO devices are used, etc.

All together is called process context

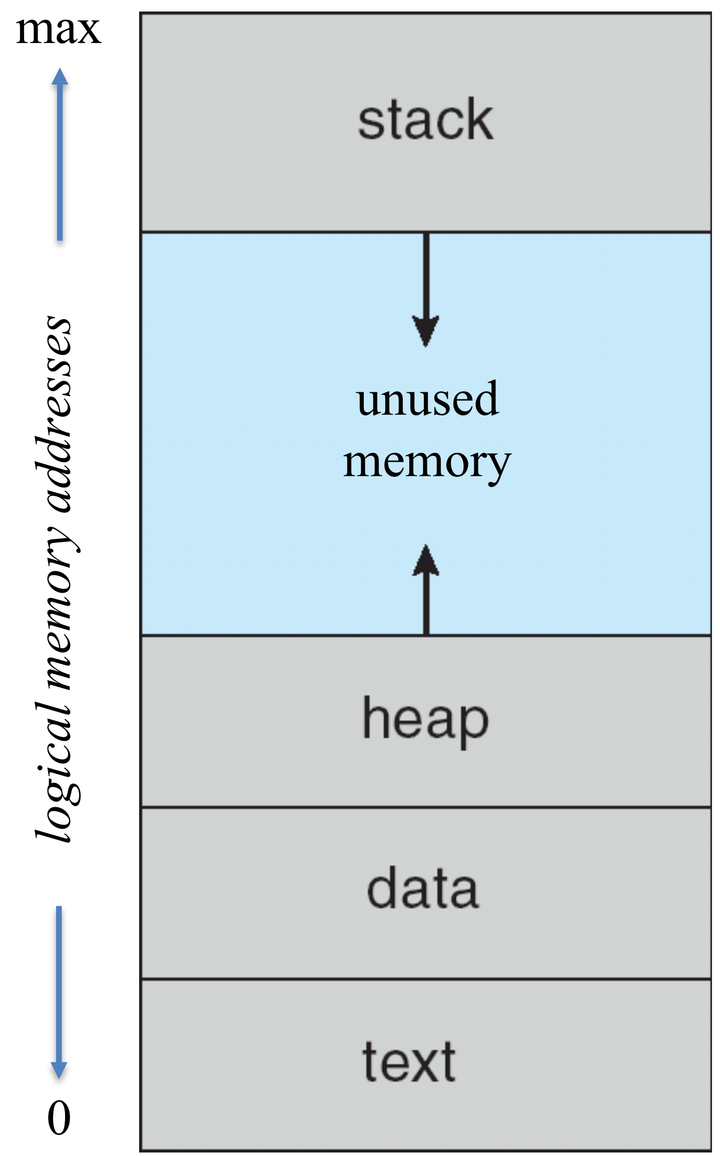
Process context is everything a process needs

Process Context

- *How is process context implemented? Where is it?*
  - Code + Data: **Process address space** (in memory)
  - Execution context: **CPU state** (program counter, CPU registers)
  - Management information: **Process Control Blocks** (in memory)
- **Process address space**: code, stack, data, heap
- **CPU states**
  - Updates only when executing
  - When not currently executing, it must be saved somewhere
  - (will go back to this when talking about context switch)
- **Process Control Blocks**
  - Process ID and state
  - Memory management info
  - Info about current usage on disk, files, …
  - Scheduling info: priority

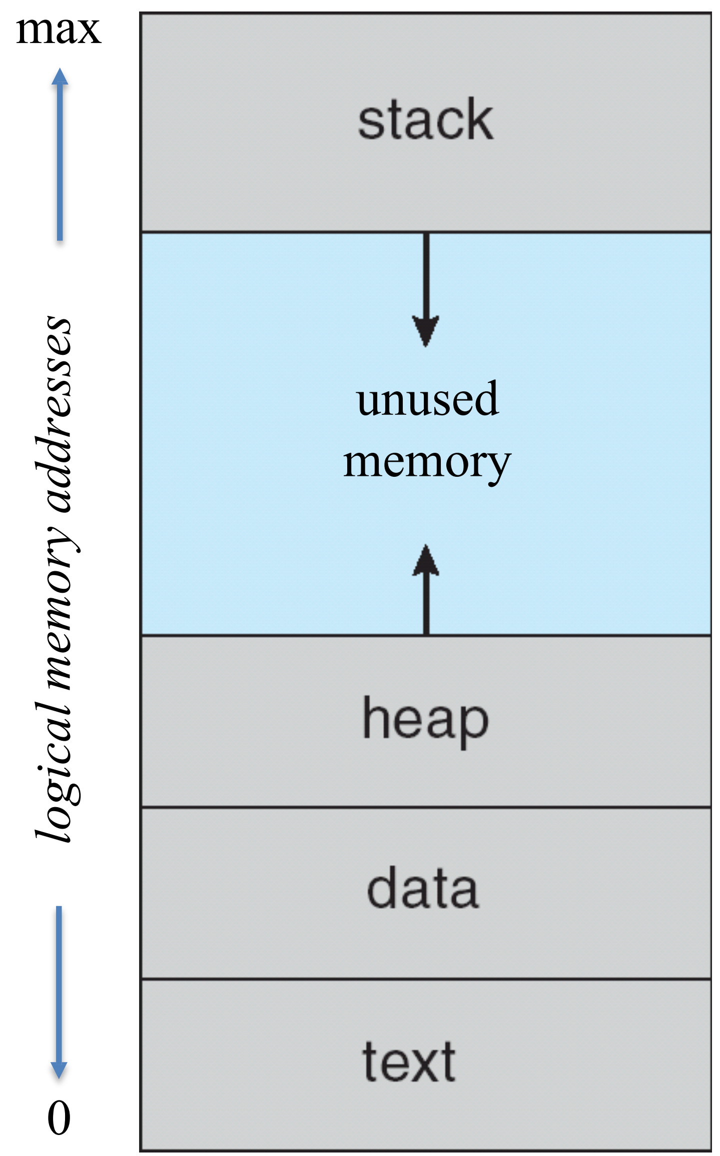
Process Address Space

- *Process address space* is all locations addressable by the process
  - Each process has its own address space
- Code (*Text*)
- Global Data (*Data*)
- Dynamic Data (*Heap*)
  - Grows up
- Local Data (*Stack*)
  - Grows down



Process Address Space

- A process has to reference memory for different purposes
  - Getting instructions (via PC)
  - Access data: static/global (data) and dynamic (heap)
- Where are these things located?
- Logical address
  - # address bits in instructions determine logical memory size
  - 48-bit can address up to 2^{48} bytes
  - A "logical address" is from 0 to the size of logical memory
  - Compiler and OS determine where things get placed in logical memory (will be back to this when we talk about virtual memory)



Process Address Space

```
int value = 5; → Global/Data

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

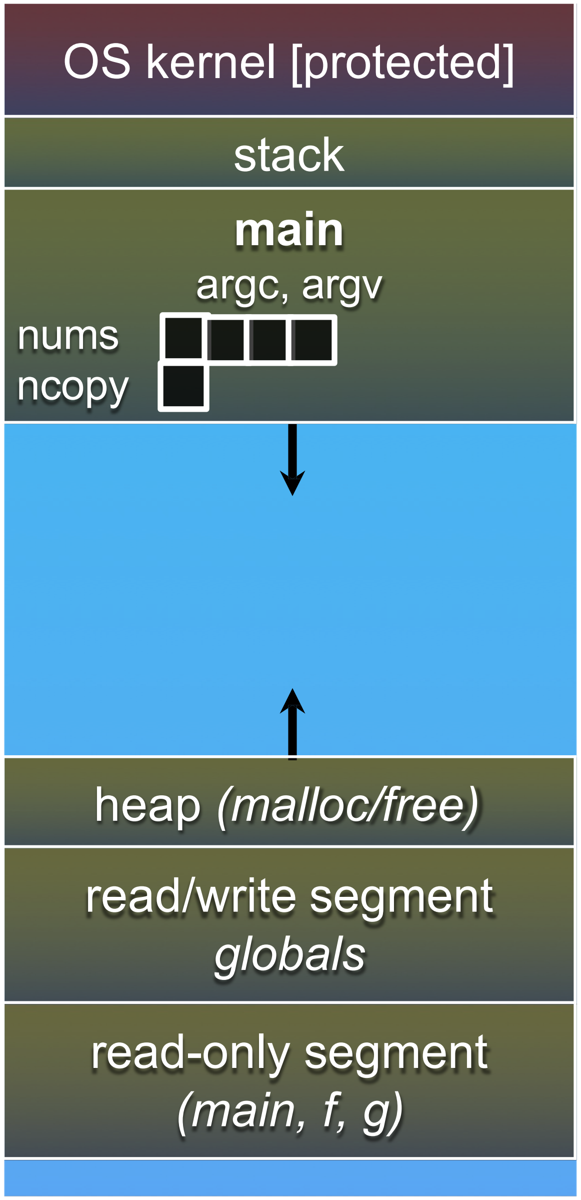
```
#include <stdlib.h>

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

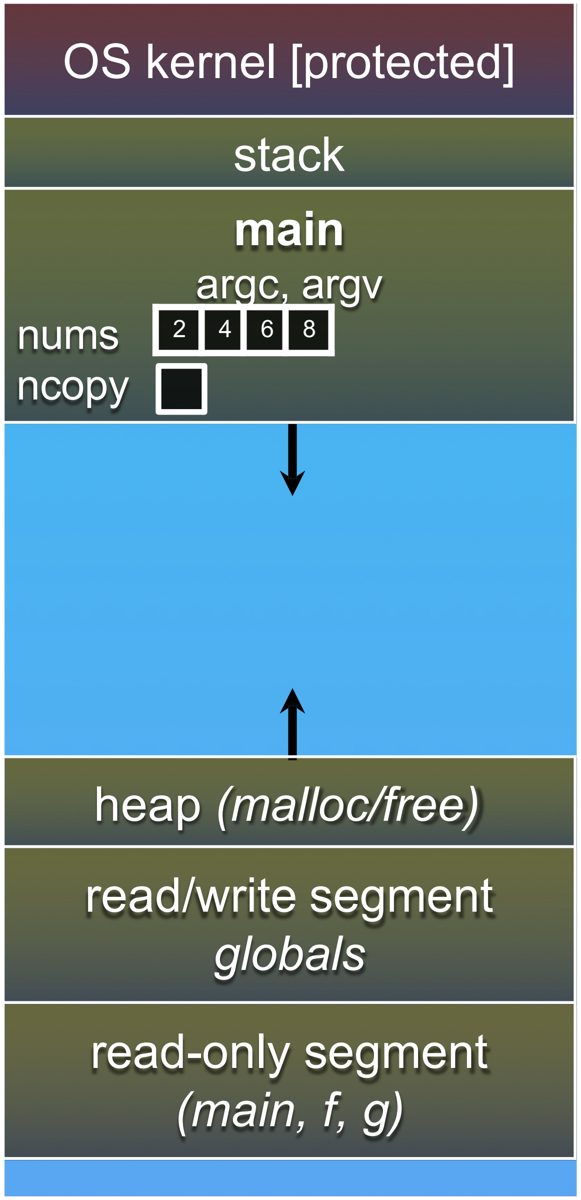
```c
#include <stdlib.h>

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

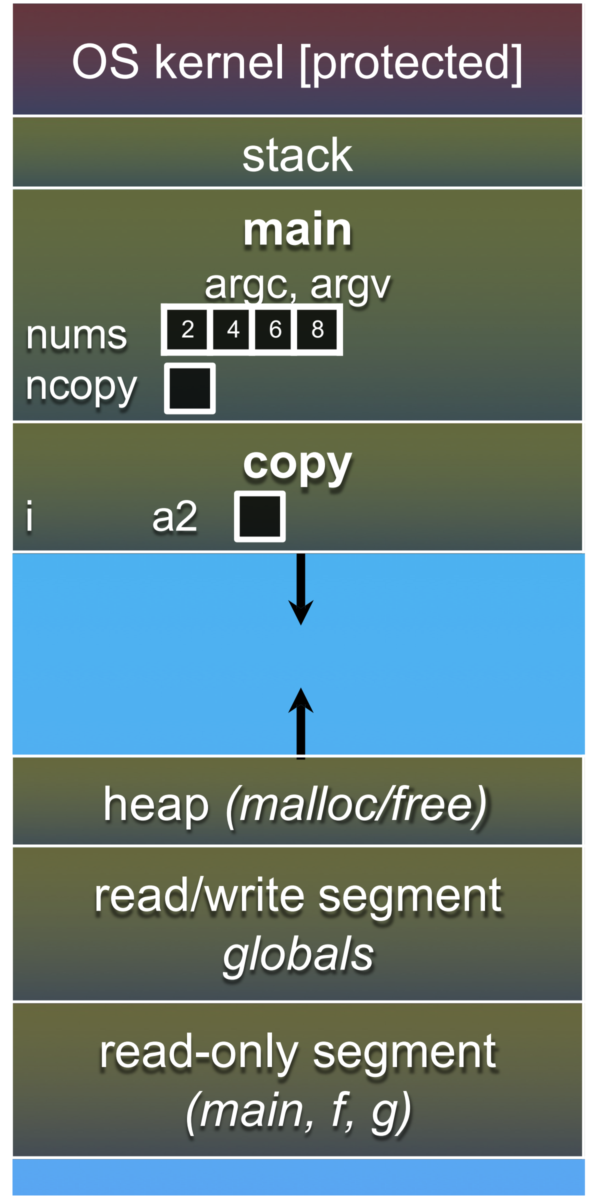
```c
#include <stdlib.h>

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

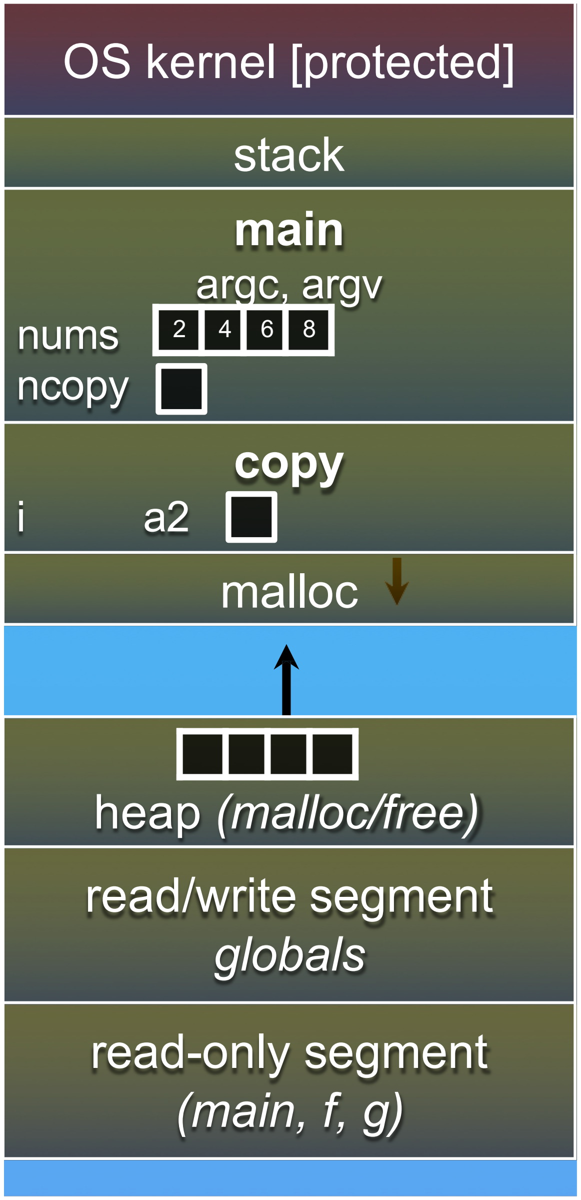
```
#include <stdlib.h>

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

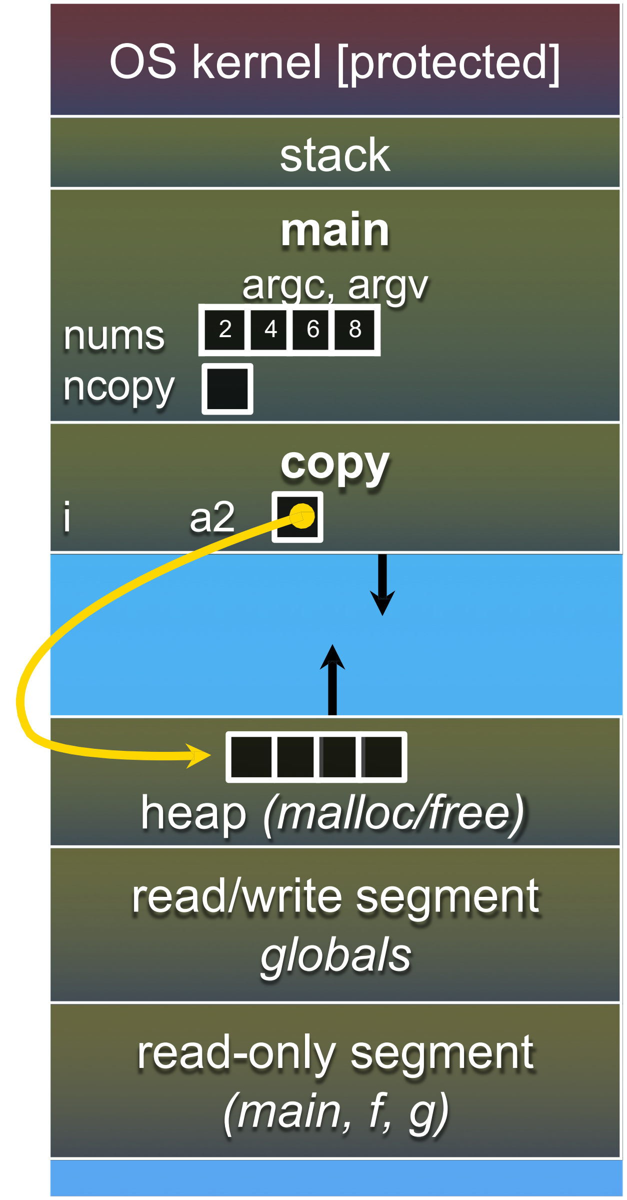
```
#include <stdlib.h>

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

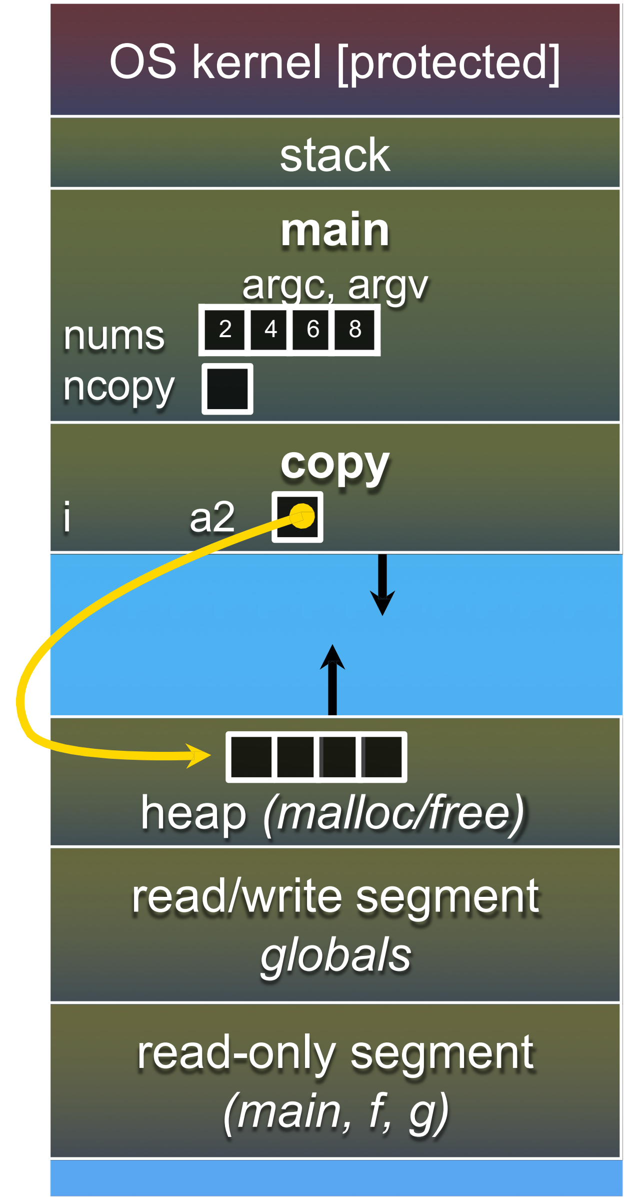
```c
#include <stdlib.h>

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

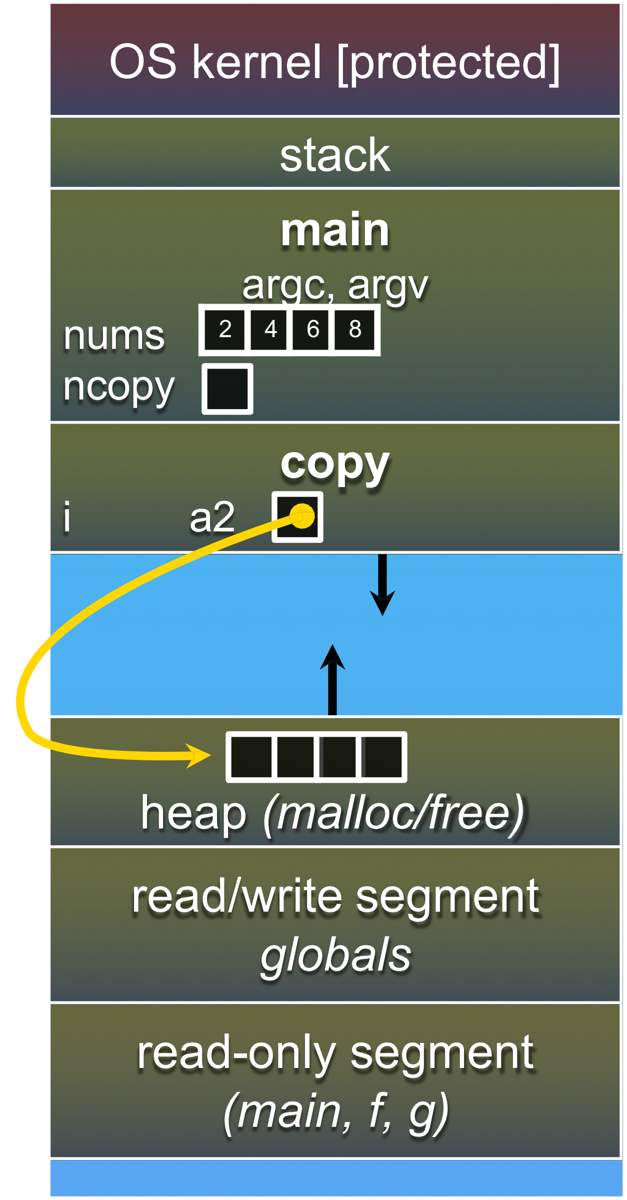
```
#include <stdlib.h>

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

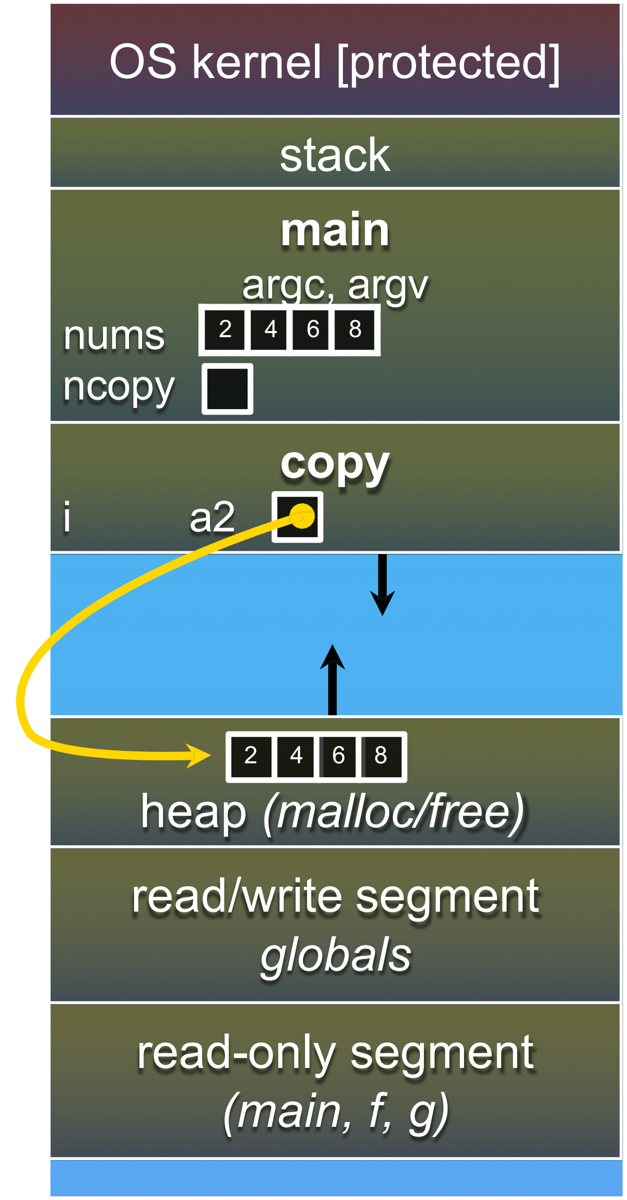
```c
#include <stdlib.h>

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;
}
```



Heap + Stack

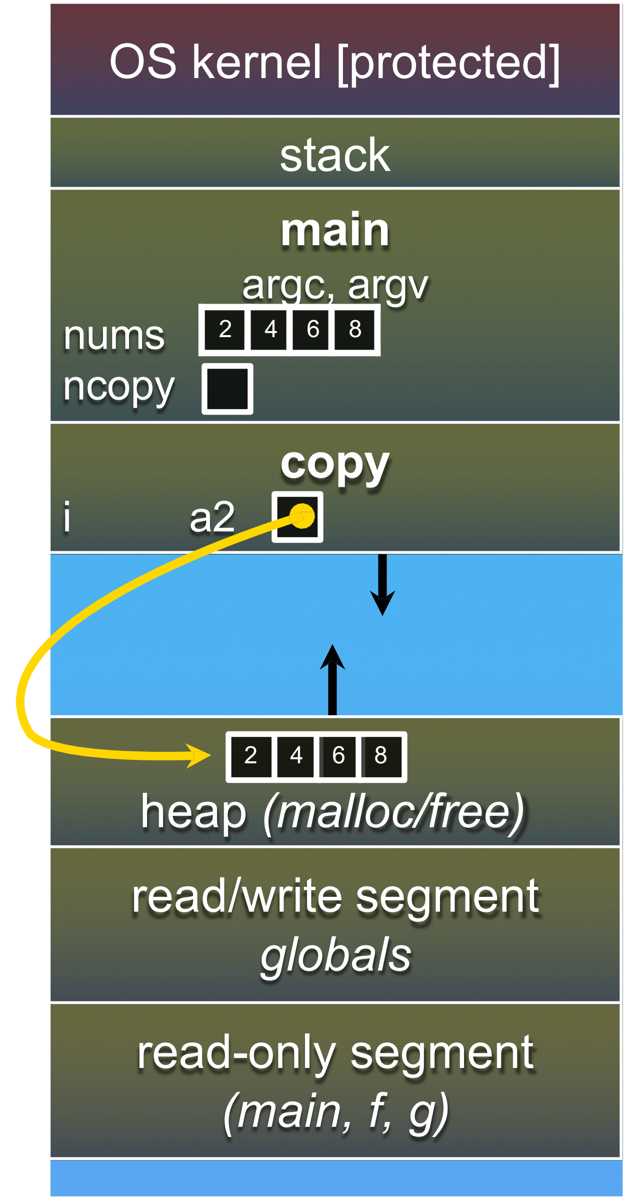
```
#include <stdlib.h>

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

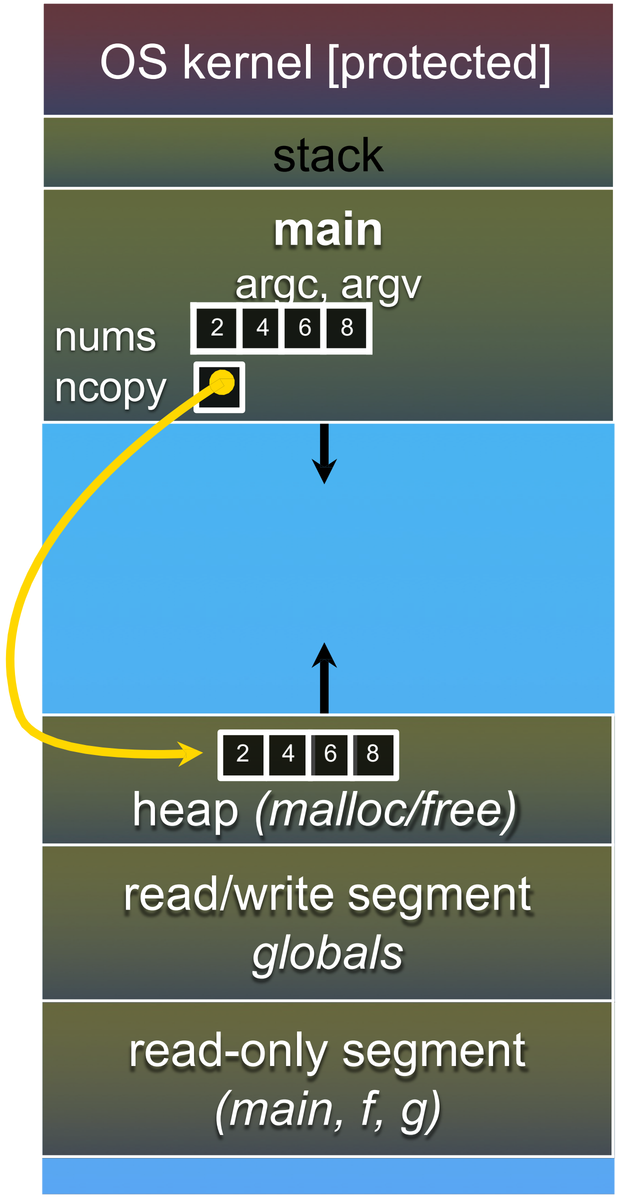
```
#include <stdlib.h>

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

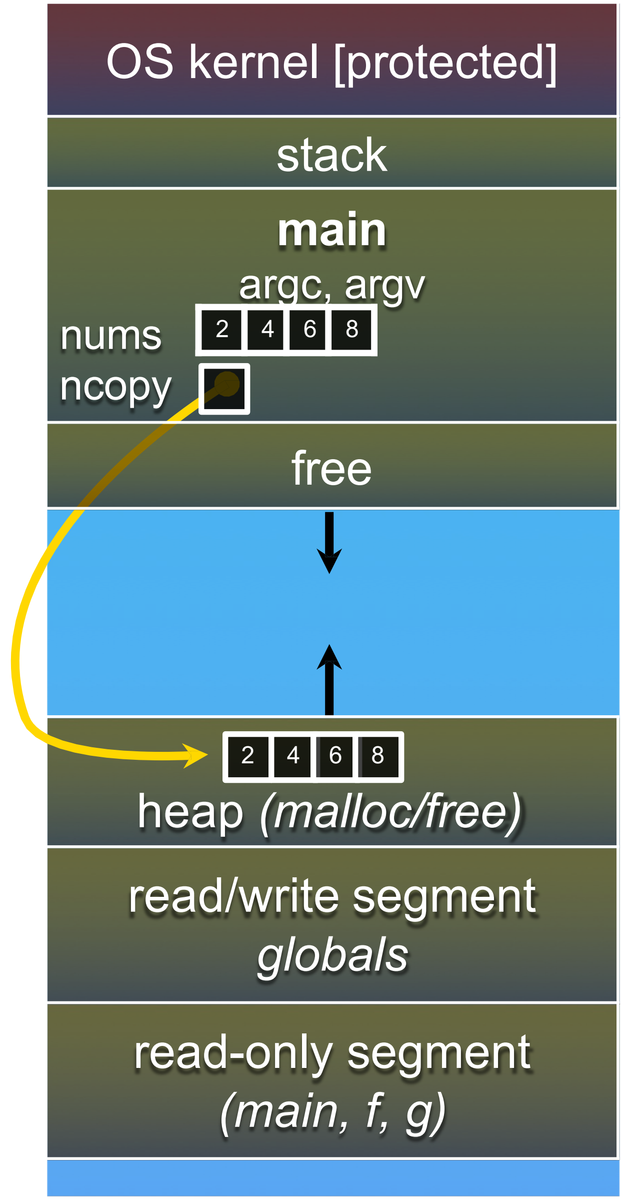
```
#include <stdlib.h>

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

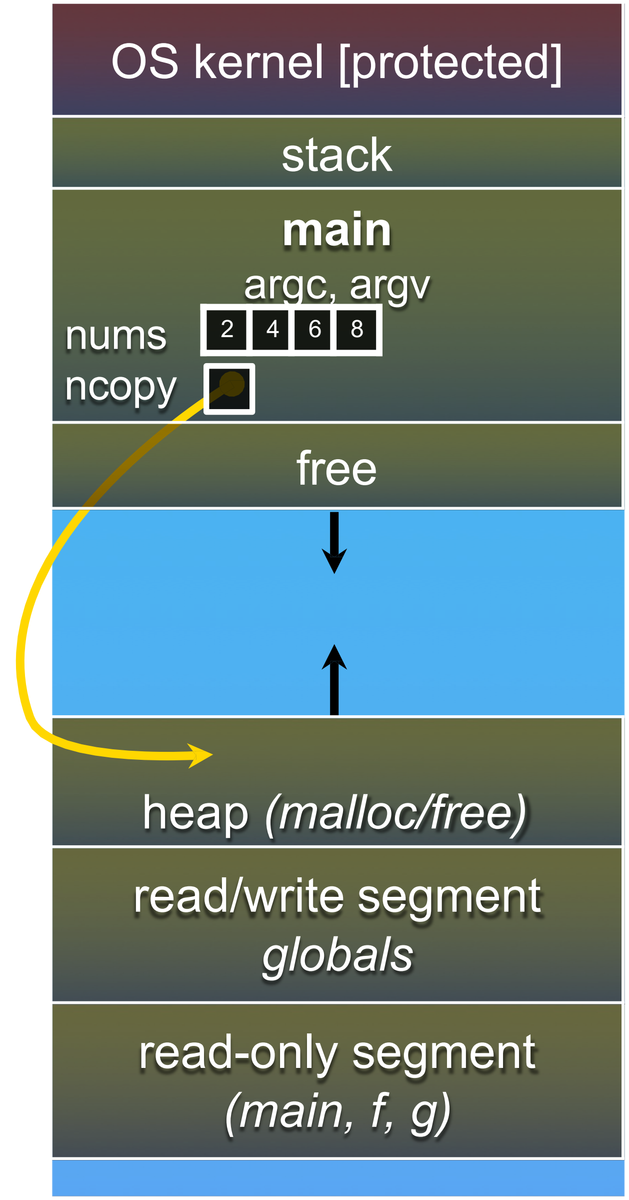
```c
#include <stdlib.h>

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

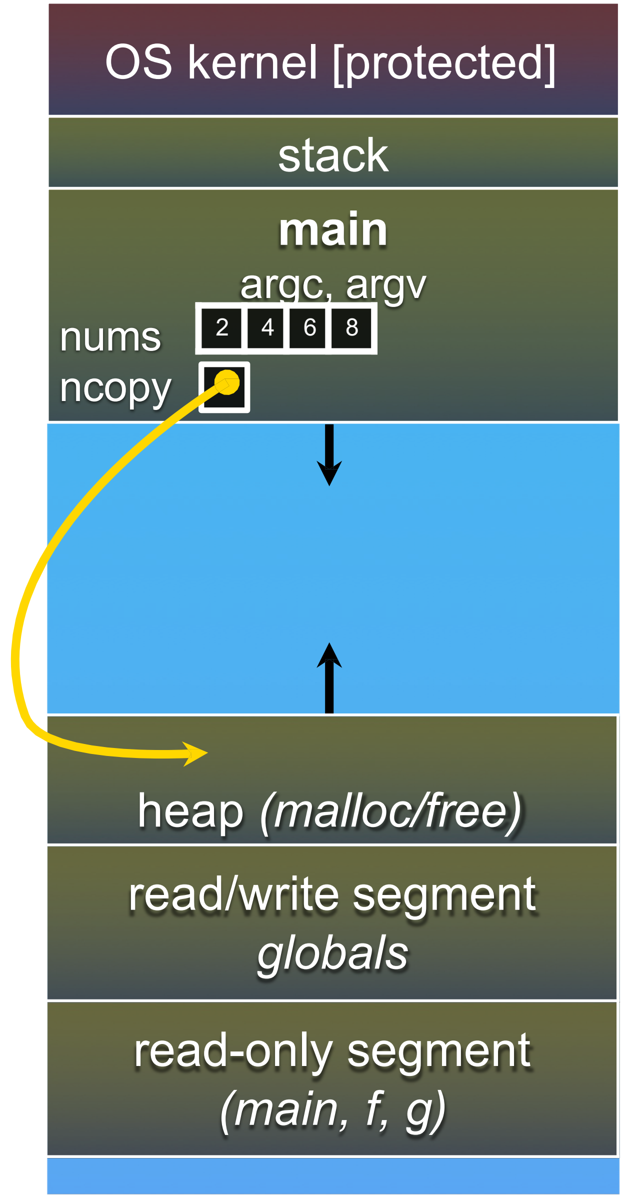
```c
#include <stdlib.h>

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;
}
```



Summary on Process Addresses

- A process executes instructions
- Addresses generated by the instructions
  - They could reference data
  - They could reference the stack
  - They could reference another instruction
- Program counter is an instruction address
- Stack pointer (stack register) is a stack address
- All addressess produced by process instruction execution on a CPU are logical addresses
- Actual physical addresses to physical memory are translated from logical address
  - Done by *memory address translation* hardware

Program to Process

- Program is stored in a binary format
  - Executable and Linkable Format (ELF)
  - `a.out`
- Binary format describes
  - Program sections: Text, Data, … (many types of sections)
  - What to load at execution time
- Determined by the compiler, loader, linker, and OS

ELF Files

```
+---------------------------------+
| ELF Header                      | ← Identifies ELF file type, architecture, entry point
+---------------------------------+
| Program Header Table            | ← Describes loadable segments for the loader
+---------------------------------+
| Section Contents                | ← Various sections used for linking/debugging
|   .text                         |     Machine code (instructions)
|   .rodata                       |     Read-only constants
|   .data                         |     Initialized global/static variables
|   .bss                          |     Uninitialized global/static variables
|  More…                          |
+---------------------------------+
| Section Header Table            | ← Metadata describing each section
+---------------------------------+
```

Why is there no stack/heap section?

- ELF sections describe **static content of the program (**what is known **before execution)**
- Their size and usage are **determined at runtime**.
- Created **dynamically by the OS when the program starts**.

CPU states

What is required to execute each instruction on CPU?

- Registers store the state of execution in CPU
  - Program counter (there is more actually…)
  - Data registers
- Program counter to indicate what to execute?
  - Holds address/index of next instruction to execute

```c
int main() {
    int a = 5;
    int b = 3;
    int c = a + b;
    return c;
}
```

C program

```
Addr 100: MOV R1, 5         ; Load a = 5
Addr 101: MOV R2, 3         ; Load b = 3
Addr 102: ADD R3, R1, R2    ; c = a + b
Addr 103: MOV R0, R3        ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU states

```
Addr 100: MOV R1, 5          ; Load a = 5
Addr 101: MOV R2, 3          ; Load b = 3
Addr 102: ADD R3, R1, R2     ; c = a + b
Addr 103: MOV R0, R3         ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU states

```
Addr 100: MOV R1, 5        ; Load a = 5
Addr 101: MOV R2, 3        ; Load b = 3
Addr 102: ADD R3, R1, R2   ; c = a + b
Addr 103: MOV R0, R3       ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU state:
PC = 100
R0 = ?
R1 = ?
R2 = ?
R3 = ?

CPU states

```
Addr 100: MOV R1, 5         ; Load a = 5
Addr 101: MOV R2, 3         ; Load b = 3
Addr 102: ADD R3, R1, R2    ; c = a + b
Addr 103: MOV R0, R3        ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU state:
PC = 101
R0 = ?
R1 = 5
R2 = ?
R3 = ?

CPU states

```
Addr 100: MOV R1, 5          ; Load a = 5
Addr 101: MOV R2, 3          ; Load b = 3
Addr 102: ADD R3, R1, R2     ; c = a + b
Addr 103: MOV R0, R3         ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU state:
PC = 102
R0 = ?
R1 = 5
R2 = 3
R3 = ?

CPU states

```
Addr 100: MOV R1, 5          ; Load a = 5
Addr 101: MOV R2, 3          ; Load b = 3
Addr 102: ADD R3, R1, R2     ; c = a + b
Addr 103: MOV R0, R3         ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU state:
PC = 103
R0 = ?
R1 = 5
R2 = 3
R3 = 8

CPU states

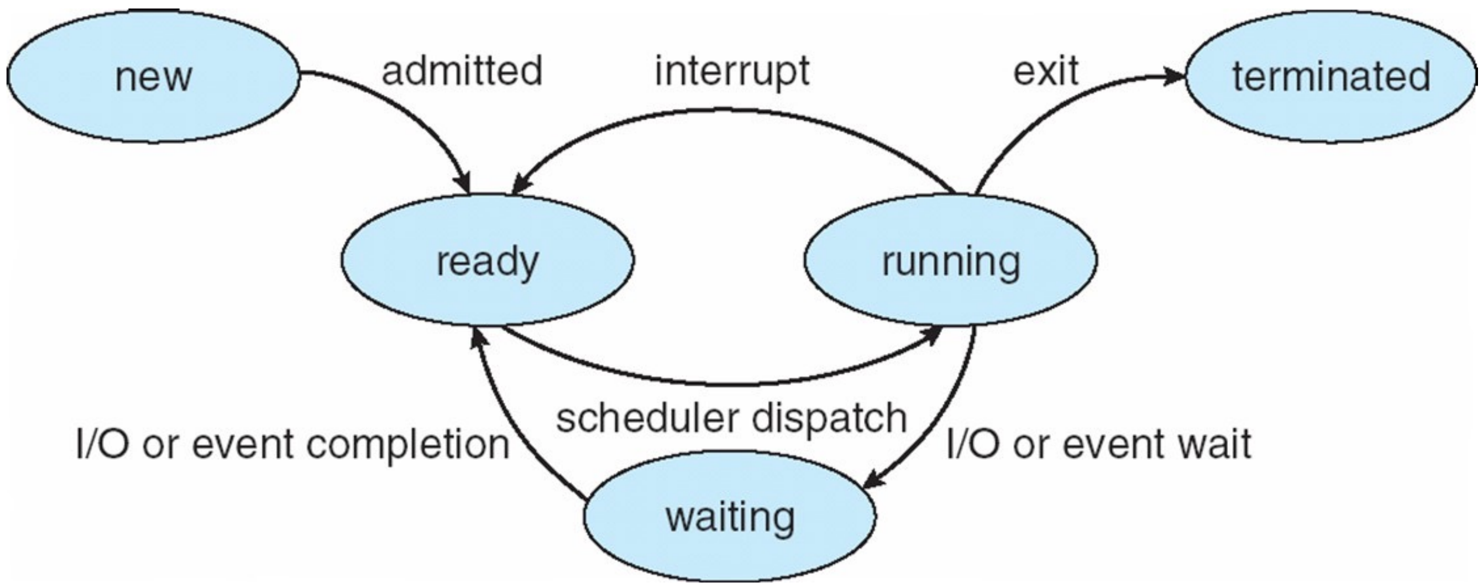
```
Addr 100: MOV R1, 5          ; Load a = 5
Addr 101: MOV R2, 3          ; Load b = 3
Addr 102: ADD R3, R1, R2     ; c = a + b
Addr 103: MOV R0, R3         ; return value in R0
Addr 104: HALT
```

Compile to assembly (simplified)

CPU state:
PC = 104
R0 = 8
R1 = 5
R2 = 3
R3 = 8

Process State

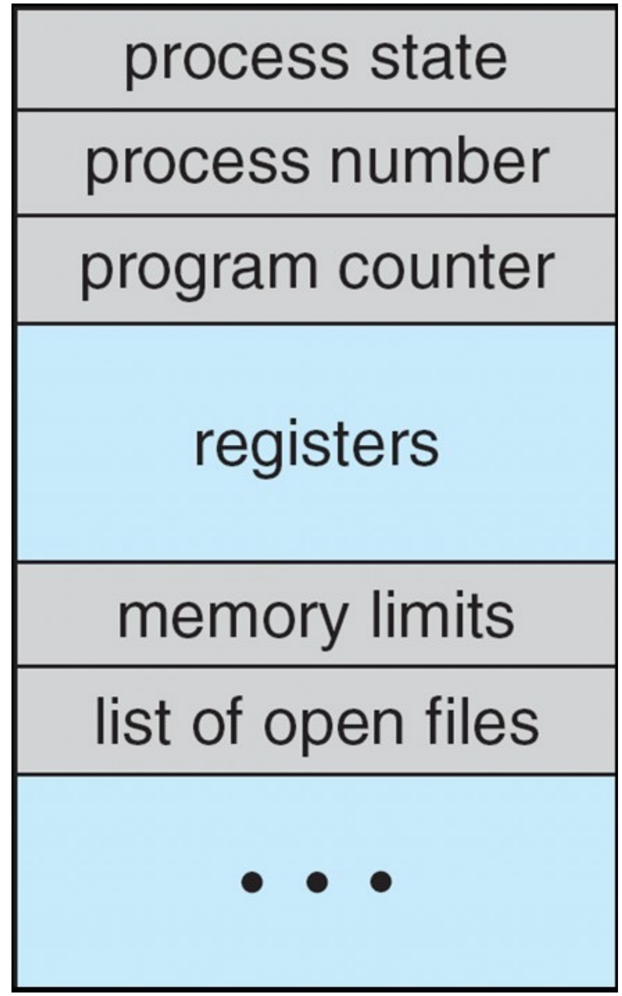
- As a process executes, it changes state
  - *New*: The process is being created (starting state)
  - *Running*: Instructions are being executed
  - *Waiting*: The process is waiting for some event to occur
  - *Ready*: The process is waiting to run
  - *Terminated*: The process has finished execution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



Process Control Block

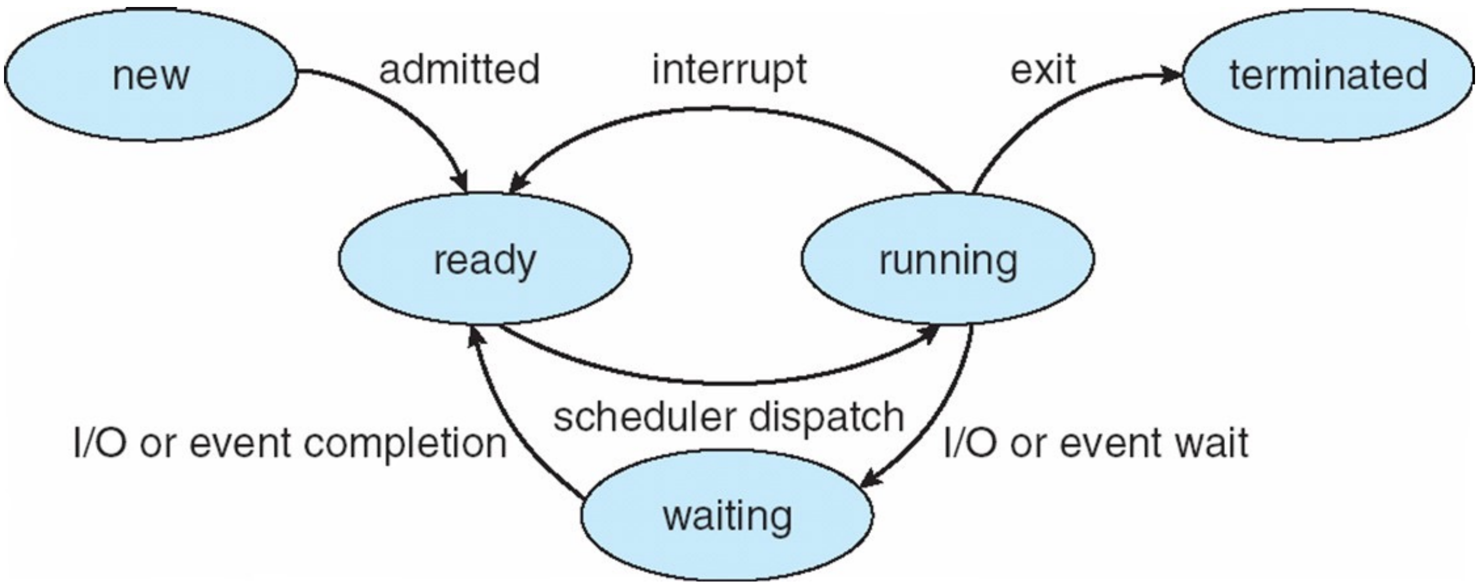
- Information associated with each process
  - Also called the *task control block*
- Process state: running, waiting, …
- Program counter
  - Location of instruction to execute next
- CPU registers for process thread (data)
- CPU scheduling information
  - Priorities, scheduling queue pointers, …
- 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

PCB



Scheduling Processes

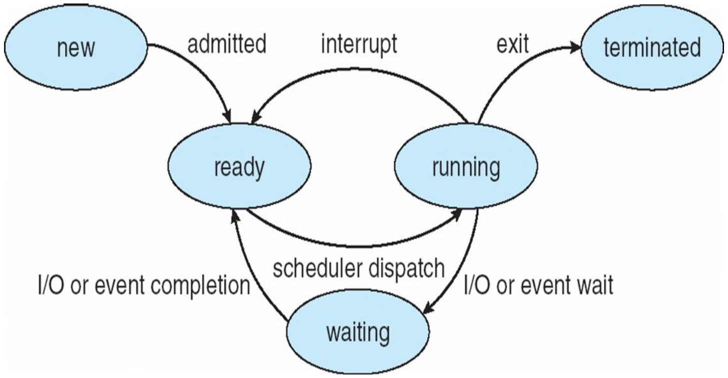
- Processes transition among execution states



Certain operating systems also more finely delineate process states

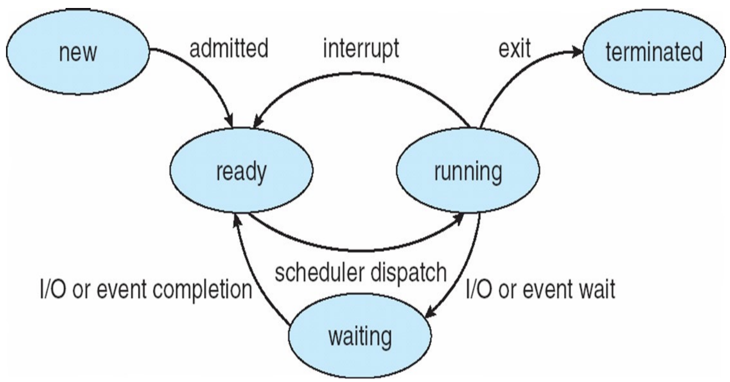
State Transitions

- New Process ==> Ready
  - Allocate resources necessary to run
  - Place process on ready queue (usually at end)
- Ready ==> Running
  - Process is at the head of ready 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 ready queue



State Transitions (2)

- 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 ready queue
- Ready ==> Running
  - As before…



Next Class

- Process scheduling