# Operating Systems
# CS 415

**Lecture 4:**

**Process**

UNIVERSITY OF OREGON


**Suyash Gupta**

Assistant Professor

Distopia Labs and ONRG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

# Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

# Assignment 1 is Out!

- **Deadline → April 21, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Midterm → April 28, 2026 (in-class)**
  - Closed book, no cheat sheets, no discussions.

# Last Class

- System Calls
- Linker and Loader

# What is a Process?

# What is a Process?

- A process is a *program in execution*.

- An operating system executes a variety of programs that run as a process.

# Program vs Process

# Program vs Process

- A program is a passive file containing instructions, typically on disk (an executable, script, etc.).

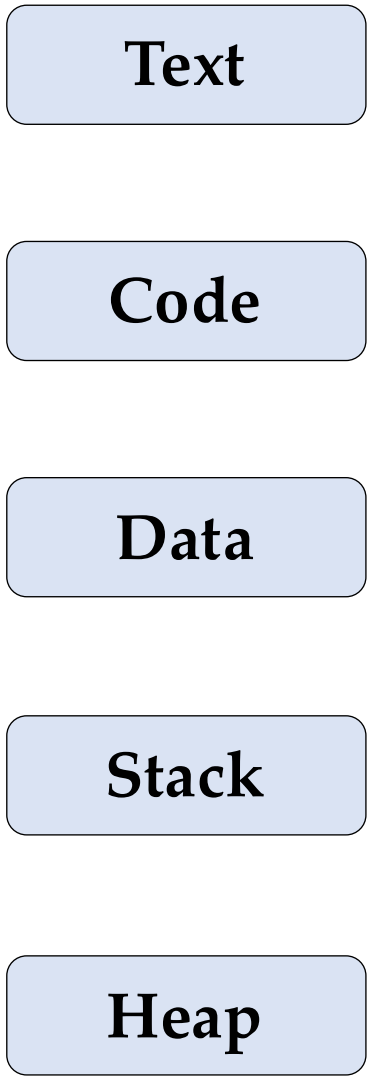
- A process is a currently executing instance of a program, with its own address space, registers, program counter, PID, and other resources.
  - When you click or write ./main → starts a process.

- One program can correspond to multiple processes if it's run multiple times.
  - Example: multiple terminals or browser windows.

- OS is basically wrapping up the passive file with active state as a process.

# Components of a Process

# Components of a Process

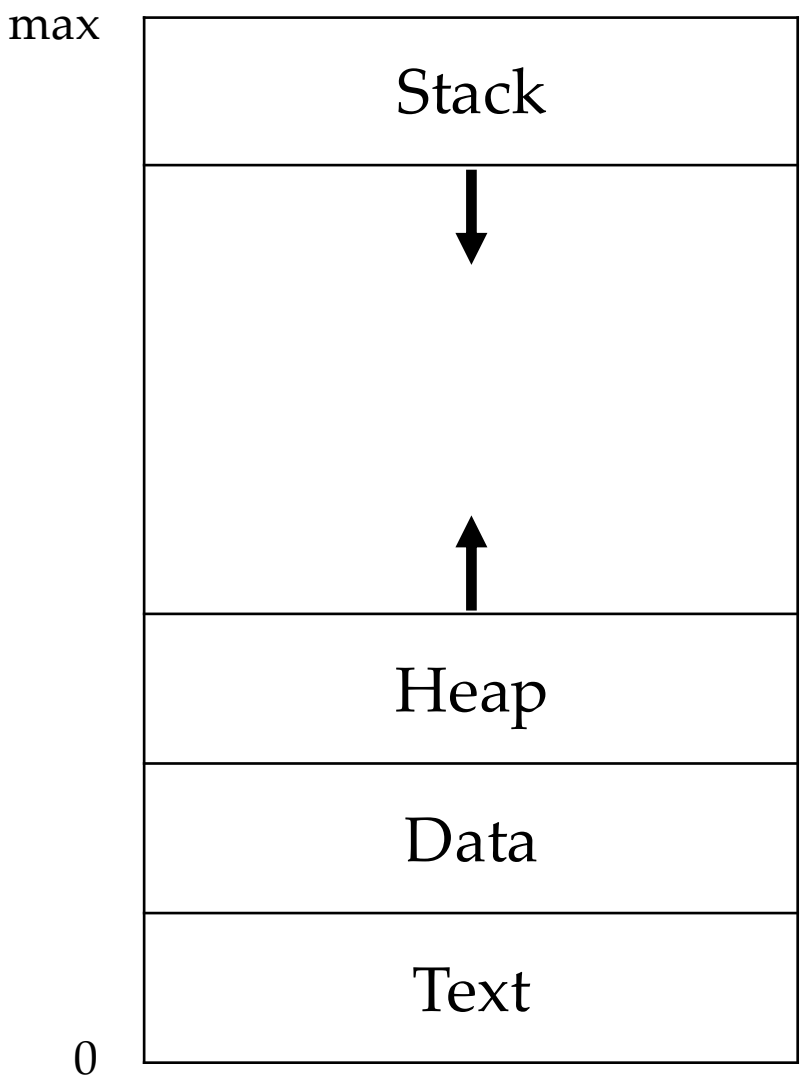


# Components of a Process

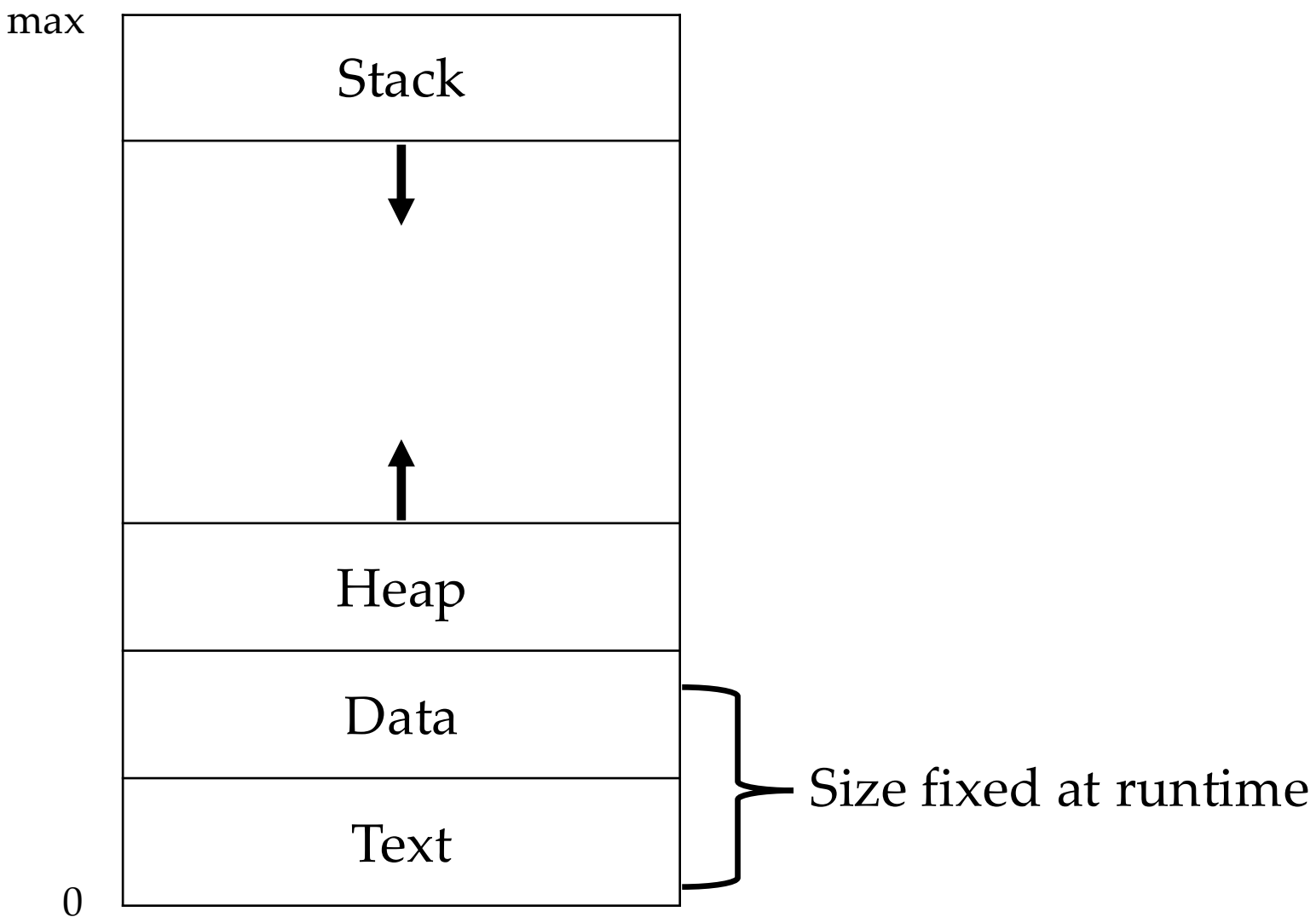
| | |
|---|---|
| **Text** | The program code. |
| **Code** | Current processor state: program counter, registers, stack pointer, etc. |
| **Data** | Global variables. |
| **Stack** | Temporary data storage when invoking functions (e.g. function parameters, return addresses, and local variables) |
| **Heap** | Memory that is dynamically allocated during program run Time. |

# Memory Layout of a Process

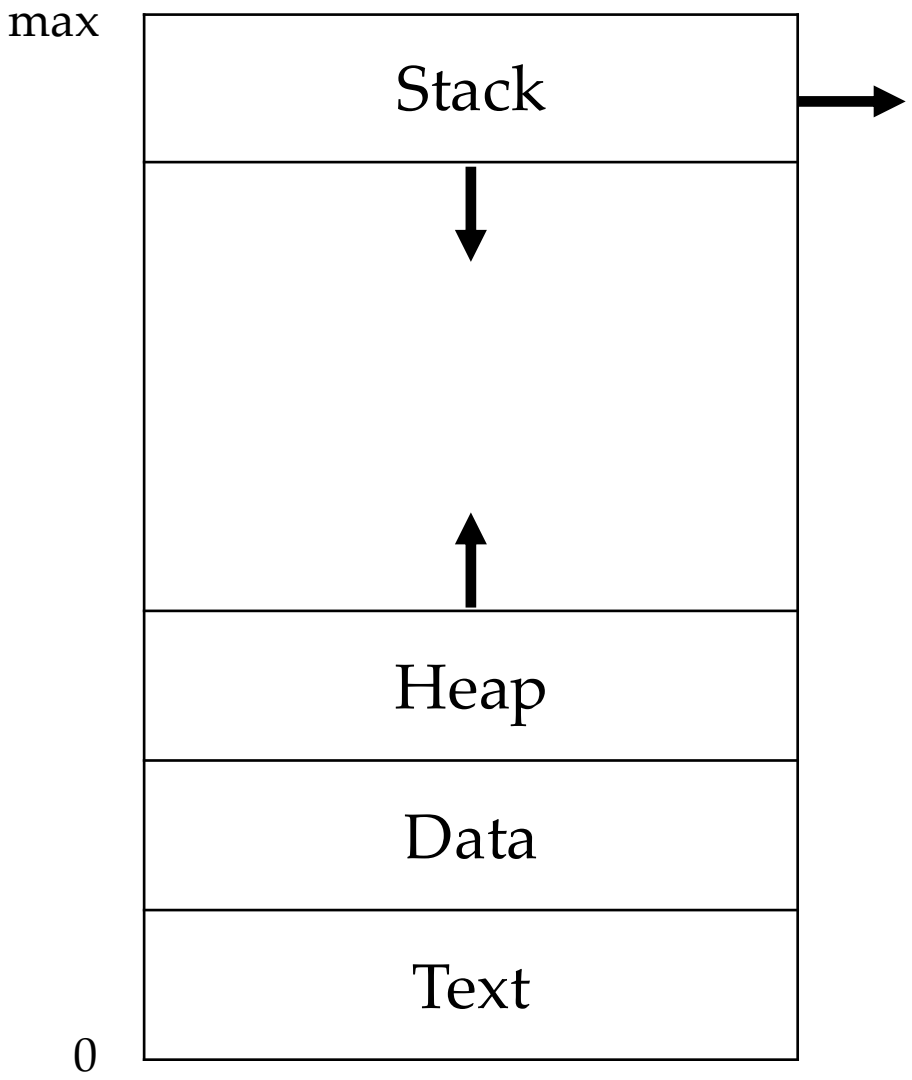
# Memory Layout of a Process



# Memory Layout of a Process



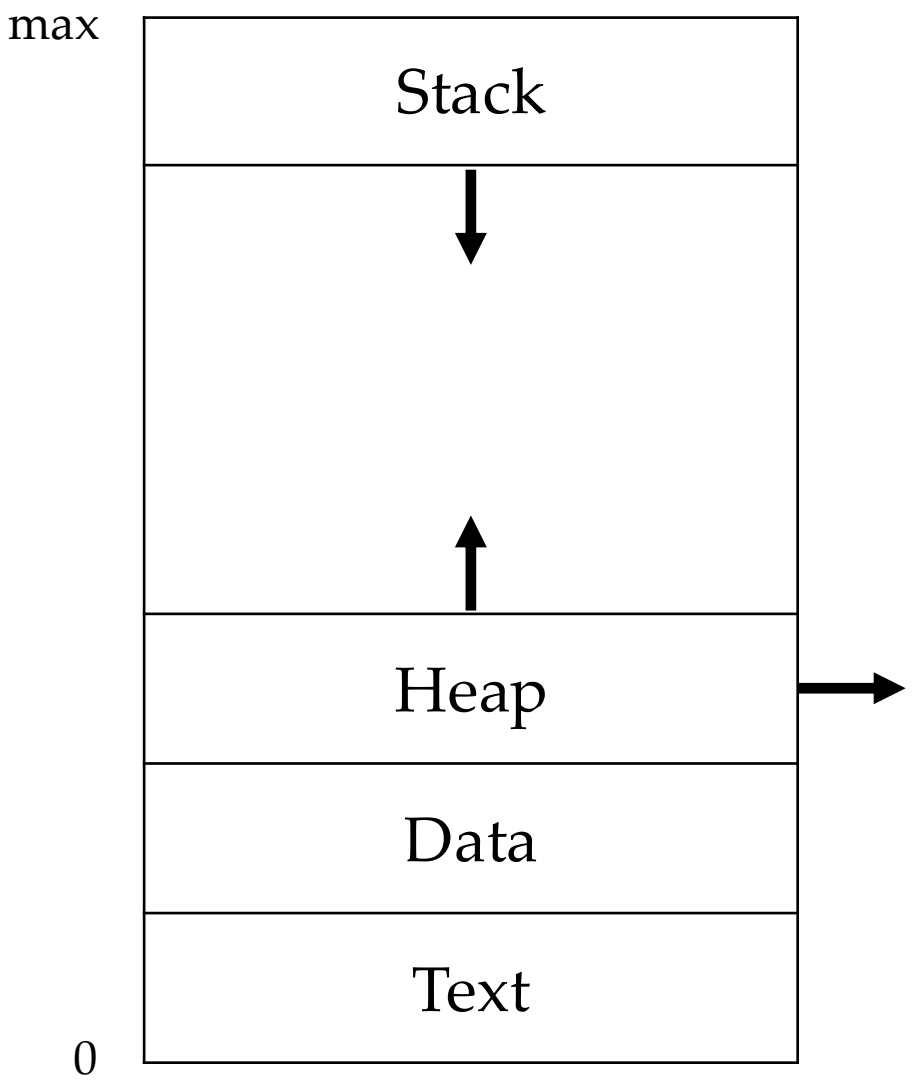
# Memory Layout of a Process




When a function is called an **activation record** containing function parameters, local variables, and the return address is pushed to stack.

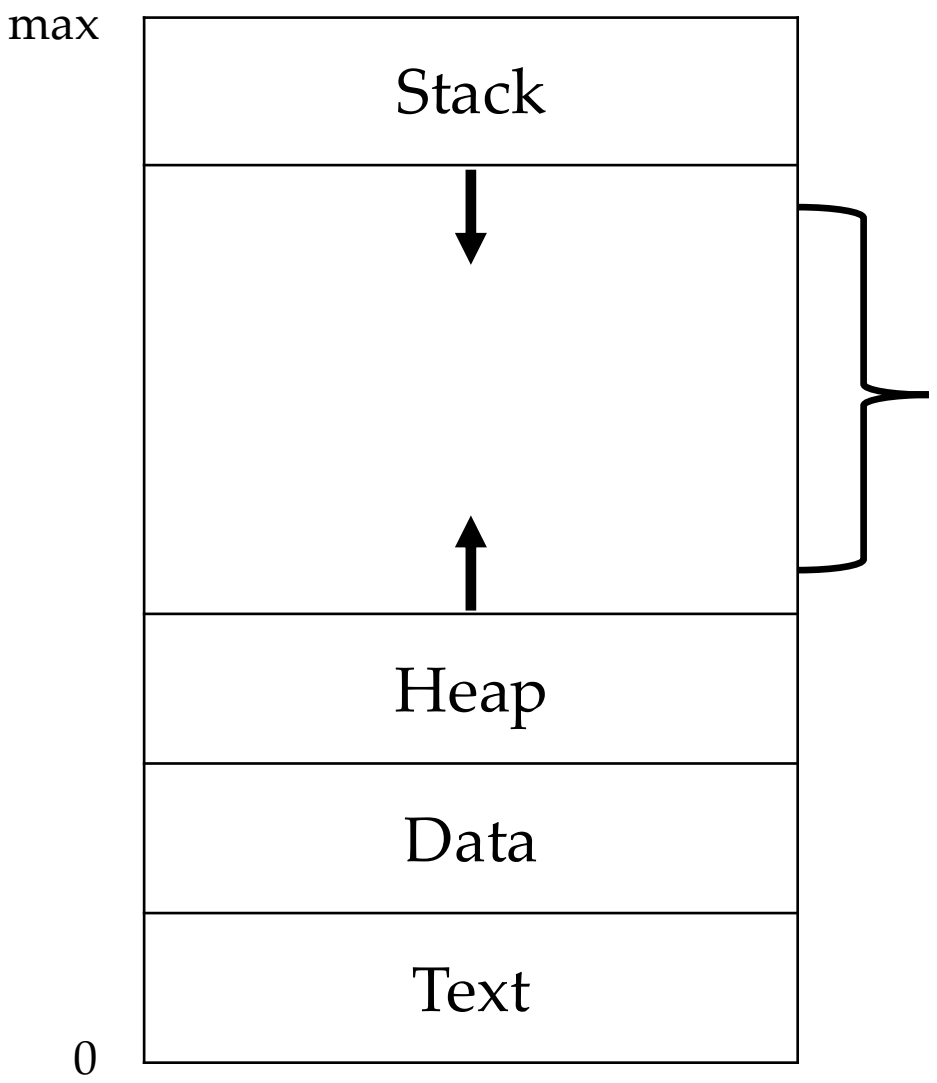
When control returns from the function, the activation record is popped from the stack.

# Memory Layout of a Process



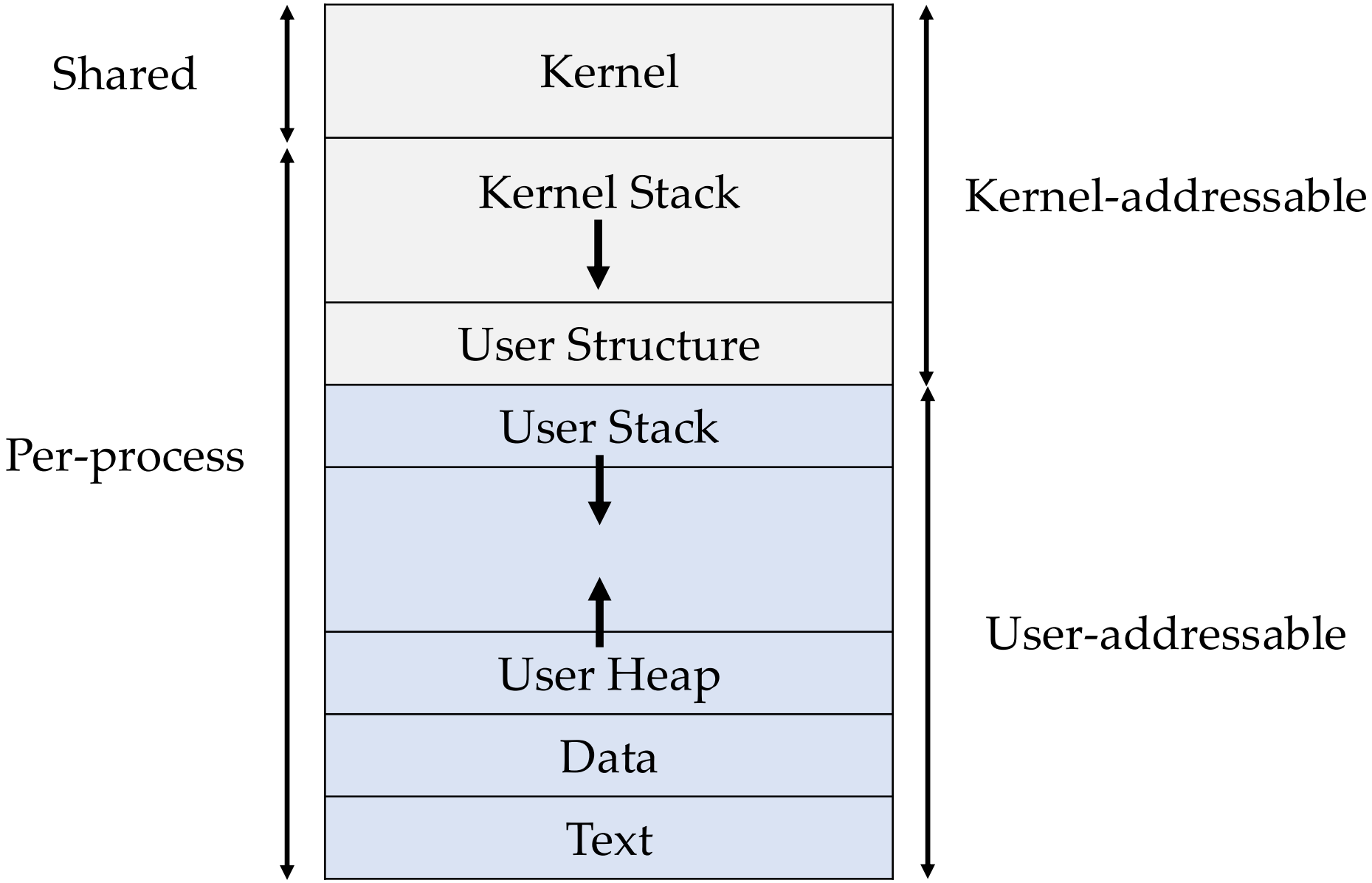

Heap grows as memory is dynamically allocated, and shrinks when memory is returned to the system.

# Memory Layout of a Process




OS must ensure that Stack and Heap do not overlap one another.

# Detailed Memory Layout



# Execution Context

- How can we locate what is the last thing process executed?
- If the process calls a function, then post return, where to resume the caller?

# Program Counter and Stack Pointer

- **Program Counter (PC)** → Which line of code am I on?

- **Stack Pointer (SP)** → Where is the notebook page where I wrote down all the stuff I need to come back to?

# Program Counter and Stack Pointer

- Program counter holds the address of the next instruction to execute.

- Stack pointer holds the address in memory where the current stack frame grows/shrinks. The stack holds:
  - Return addresses for function calls.
  - Saved registers.
  - Local variables and sometimes parameters.
  - Temporaries used by the calling convention.

- Why PC alone isn't enough?
  - When you call a function, you must save a return address somewhere so you know where to go back after the call.

# Process States

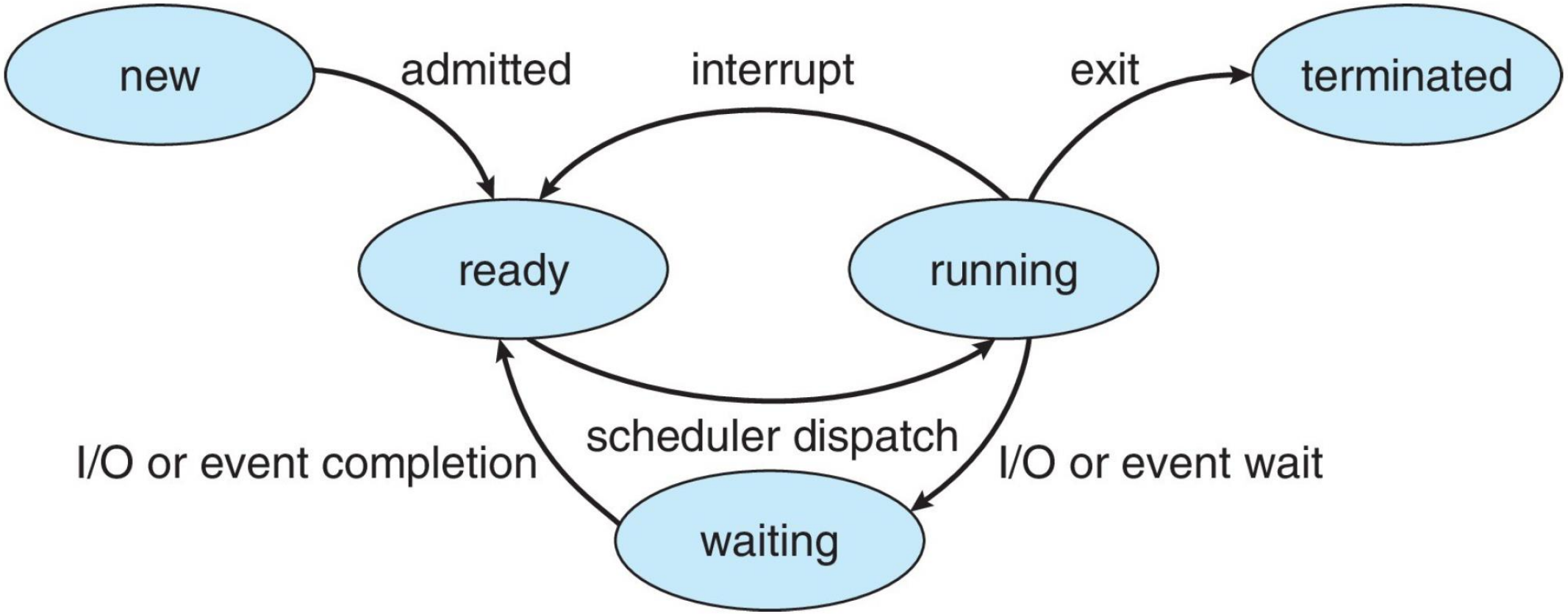
# Process States

- As a process executes, it changes state.
- It may be created, ready, executing, waiting, or terminated.
- The state model helps the OS reason about work.
- The transitions correspond to concrete events.

# Process States

- As a process executes, it changes state.
- **New** → The process was just created.
- **Ready** → The process is ready to run but is waiting to be assigned to CPU.
- **Running** → Instructions are being executed.
- **Waiting** → The process is waiting for some event to occur and is not able to use the CPU.
- **Terminated** → The process has finished execution

# Process States



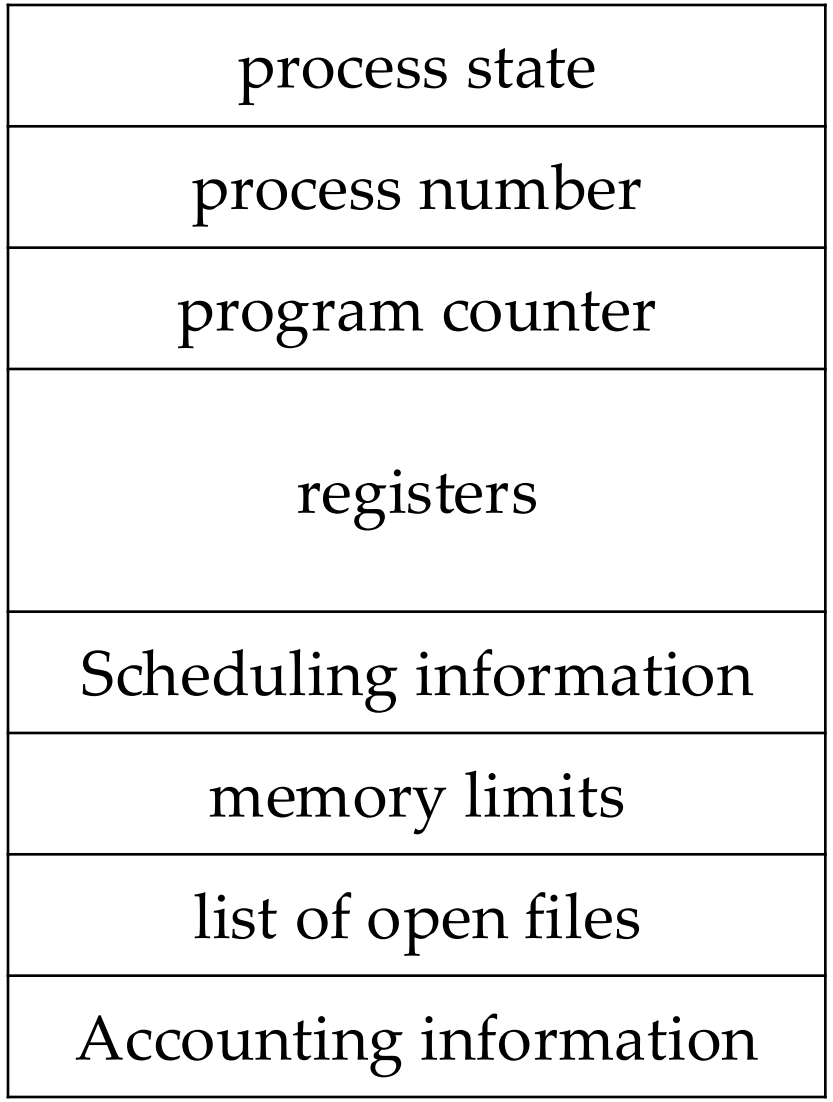
# What about suspended processes?

# What about suspended processes?

- Some OS allow them to be tracked separately.
  - So that they can get execution priority.

# Process Control Block

# Process Control Block




The PCB is the kernel's record for a process.

It stores the information needed to stop and resume execution.

# Process Control Block

| process state |
| --- |
| process number |
| program counter |
| registers |
| Scheduling information |
| memory limits |
| list of open files |
| Accounting information |

Stores priority, scheduling class, and queue links.

For memory addressing.

Amount of CPU and real time used, time limits.

# PCB representation in Linux

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;         /* address space of this process */
```

These are the fields in C structure → **task_struct**

Found in **<include/linux/sched.h>** in the kernel source-code directory

# Process View

```
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;
}
```

# Process View

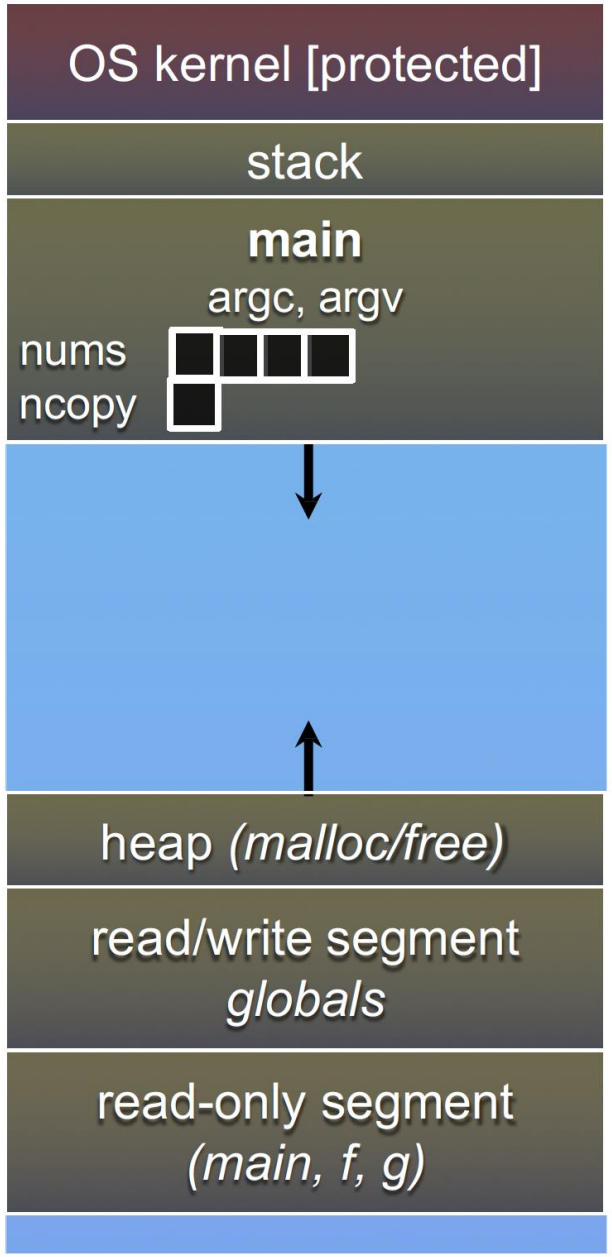
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

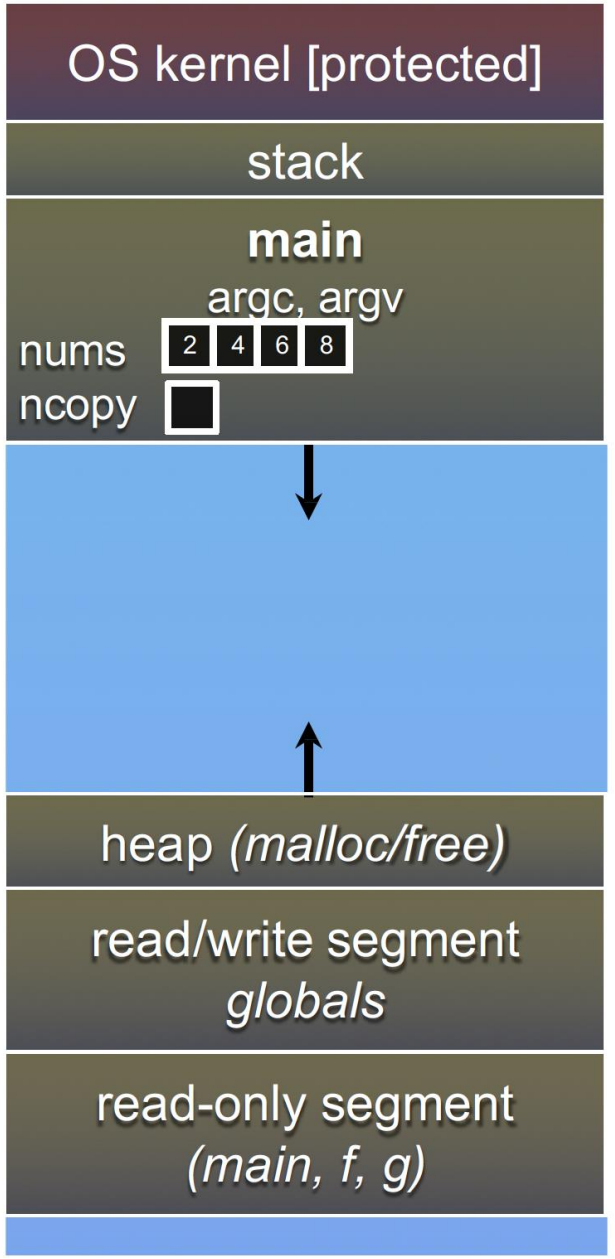
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

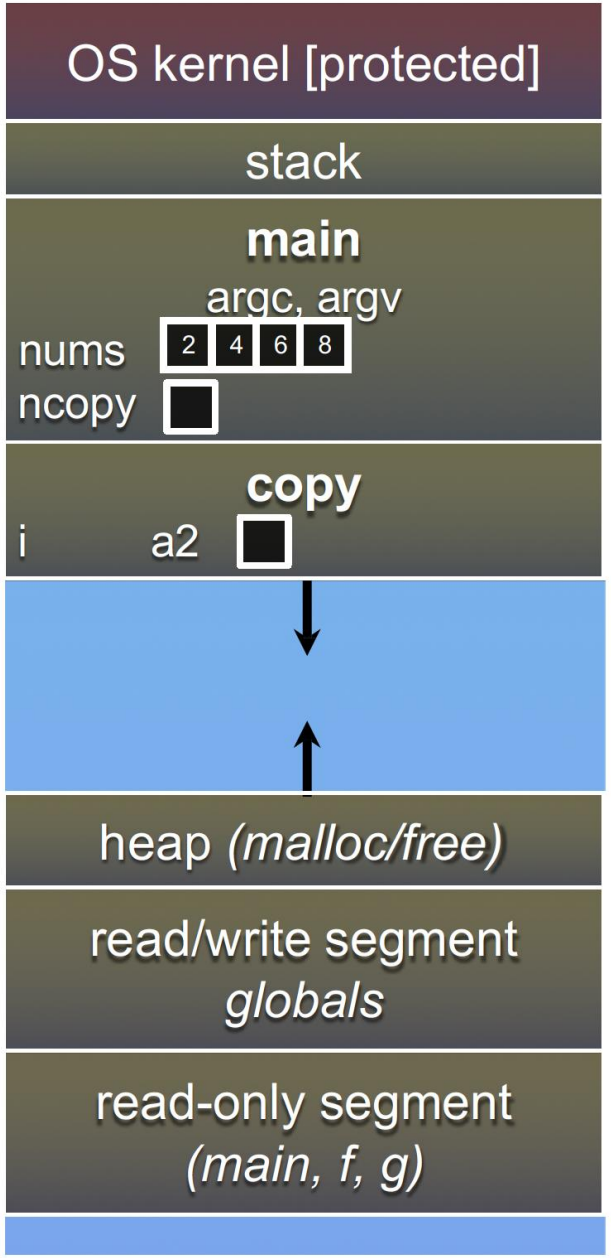
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

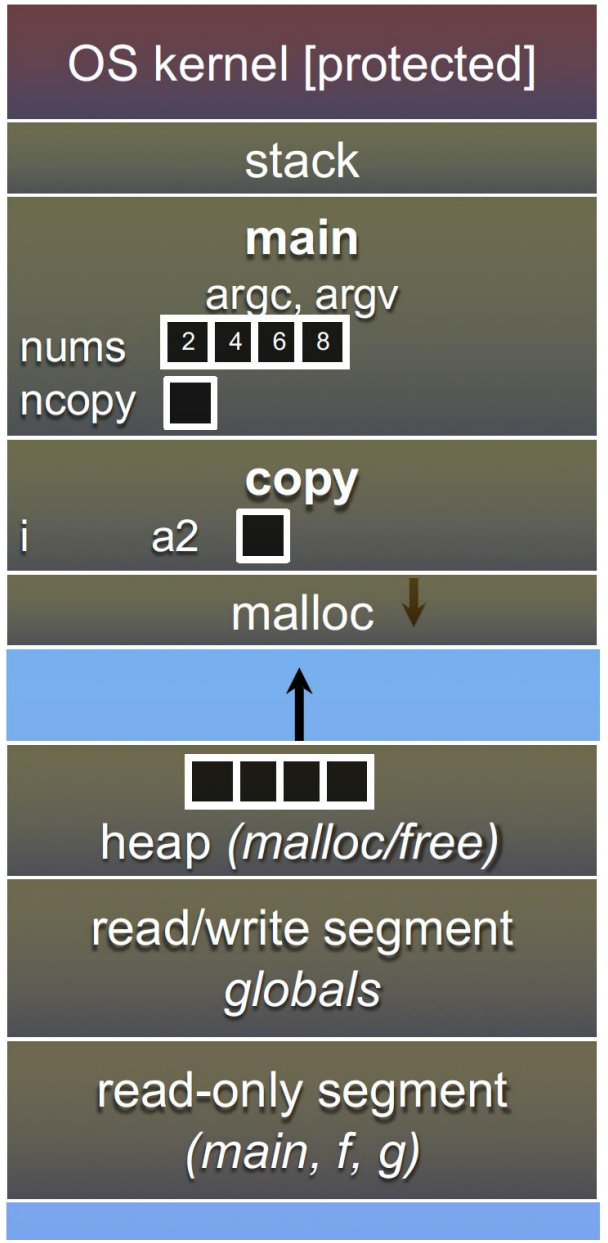
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

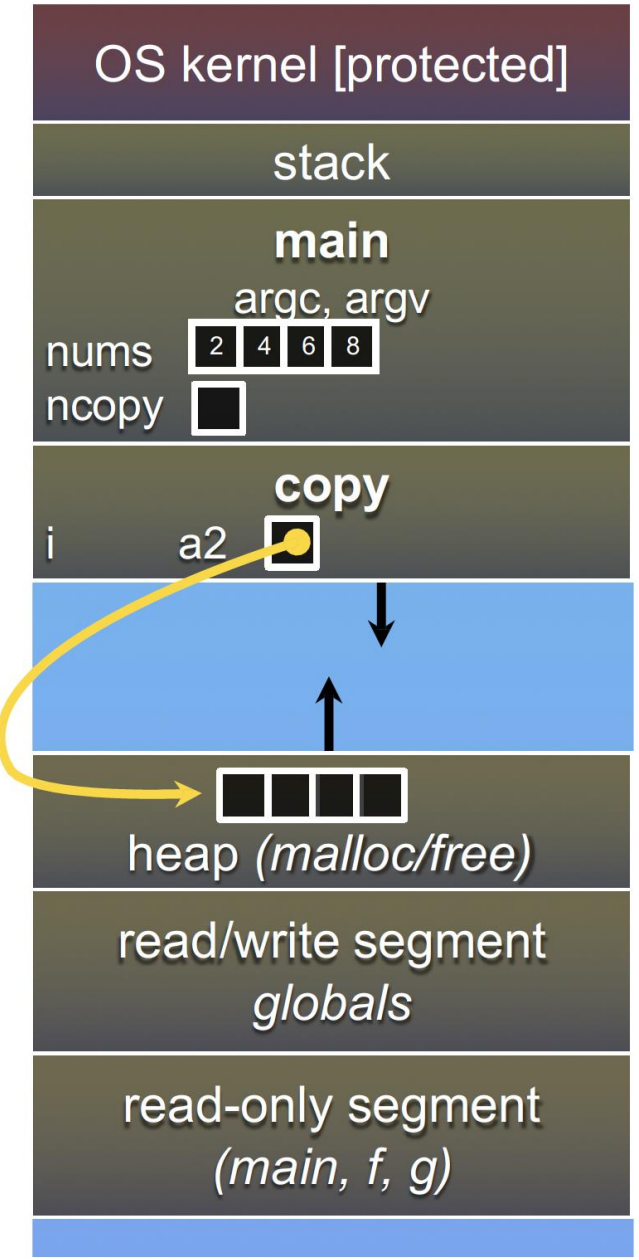
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

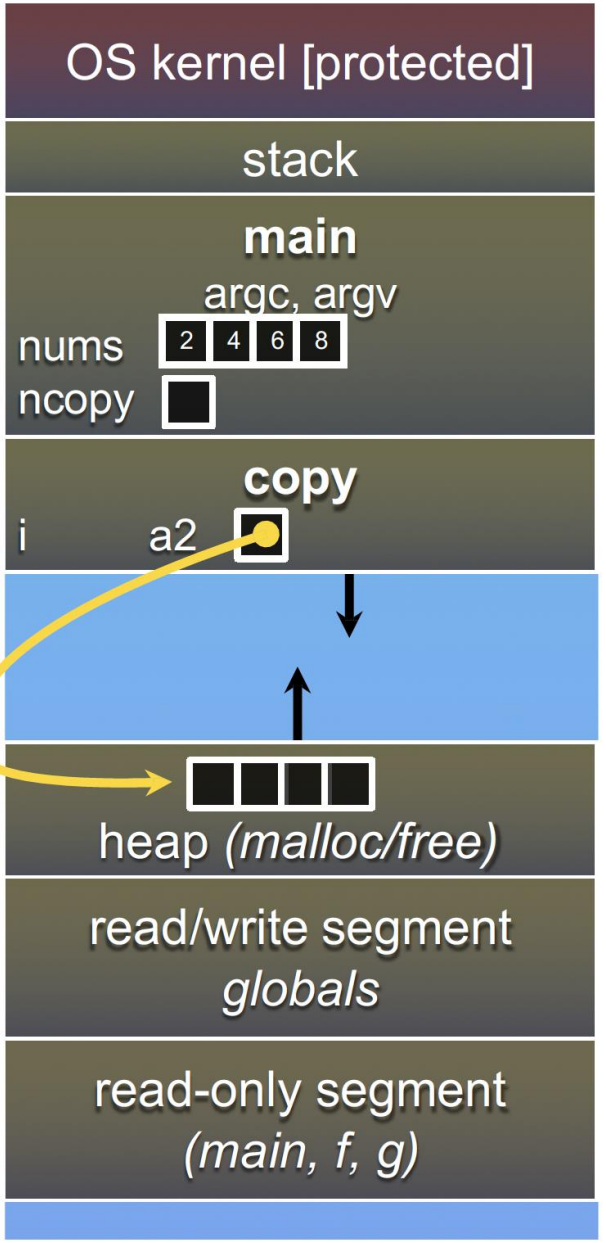
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

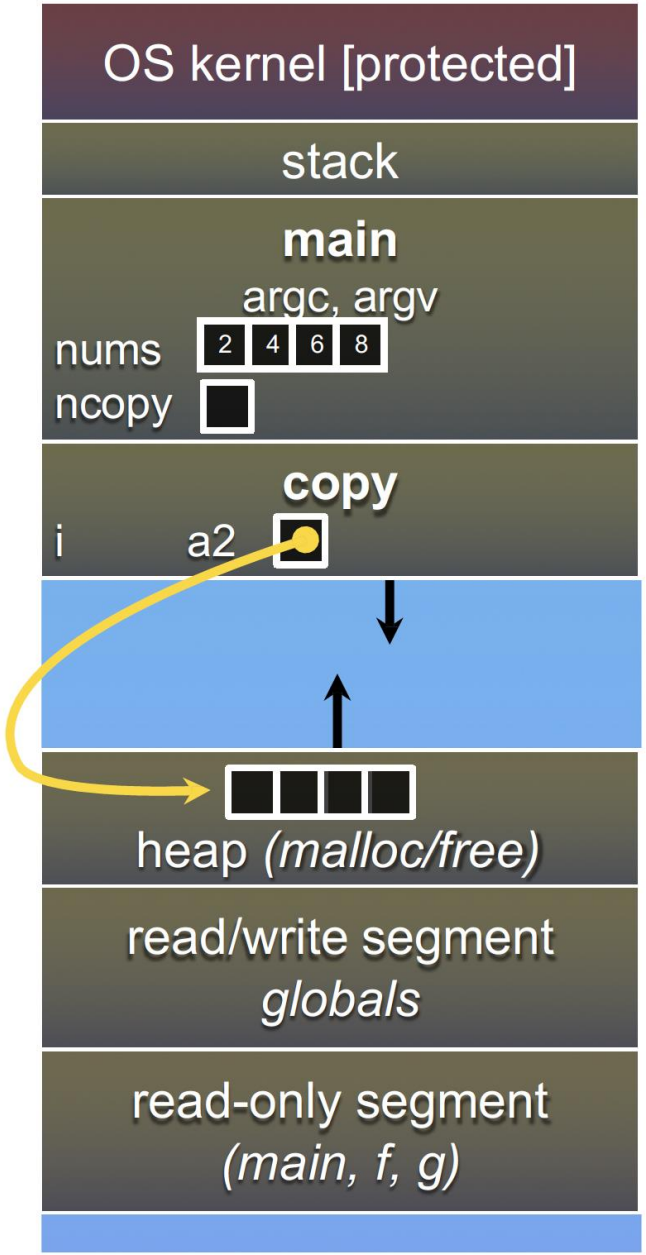
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

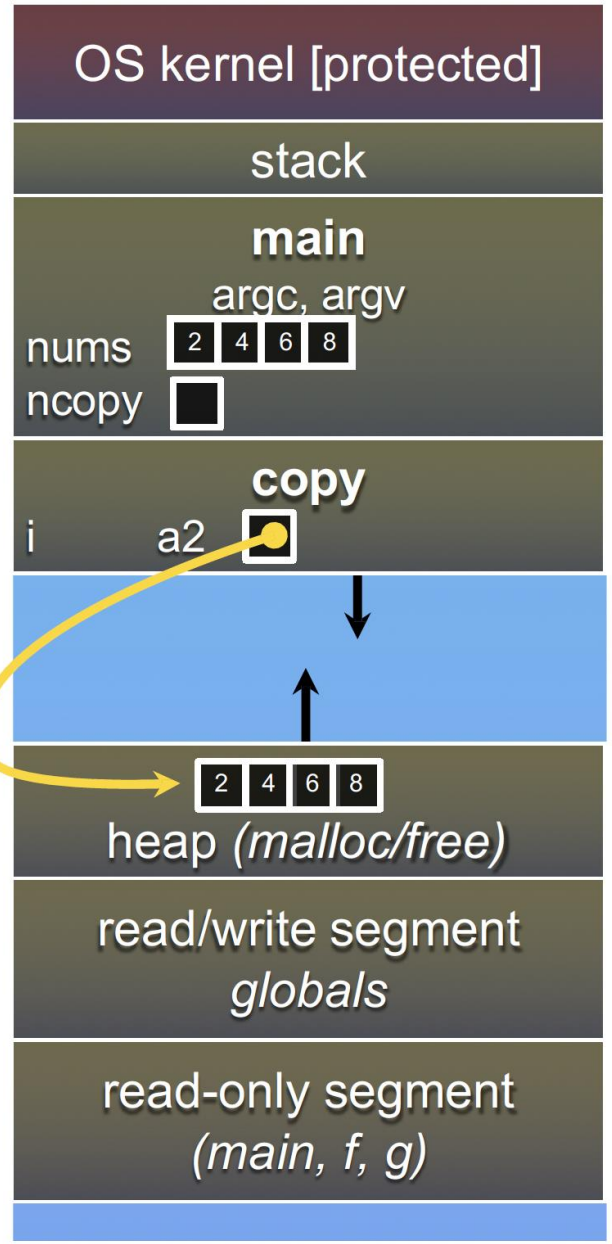
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

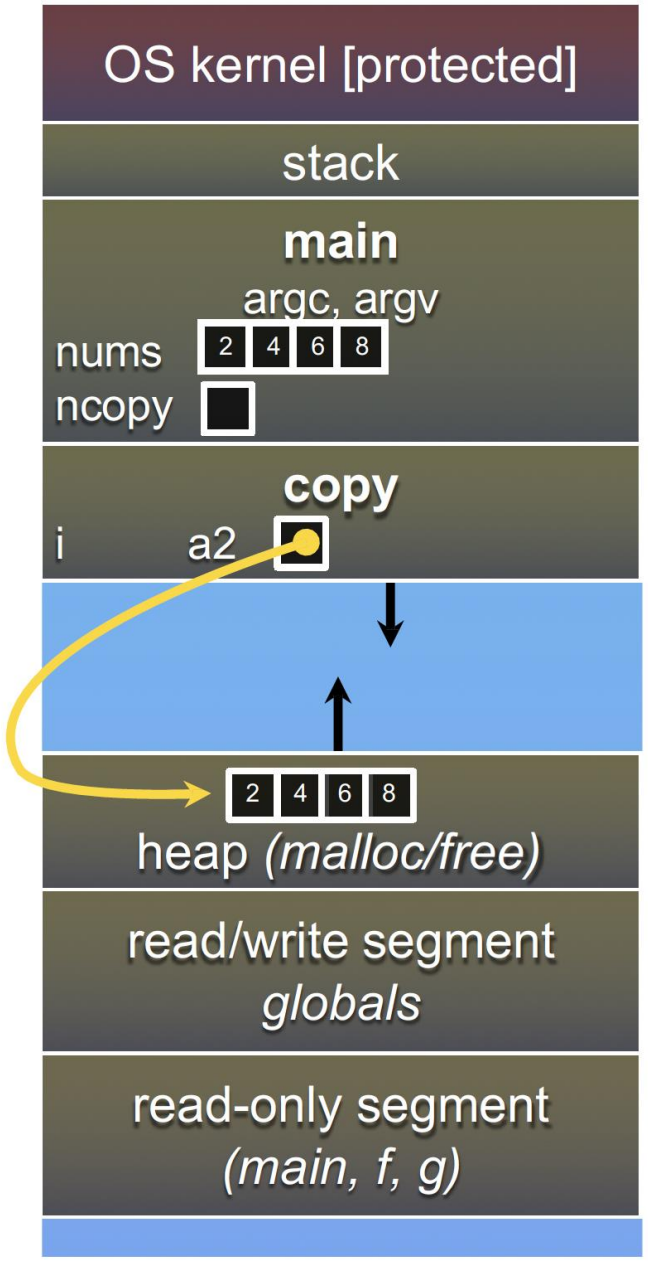
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

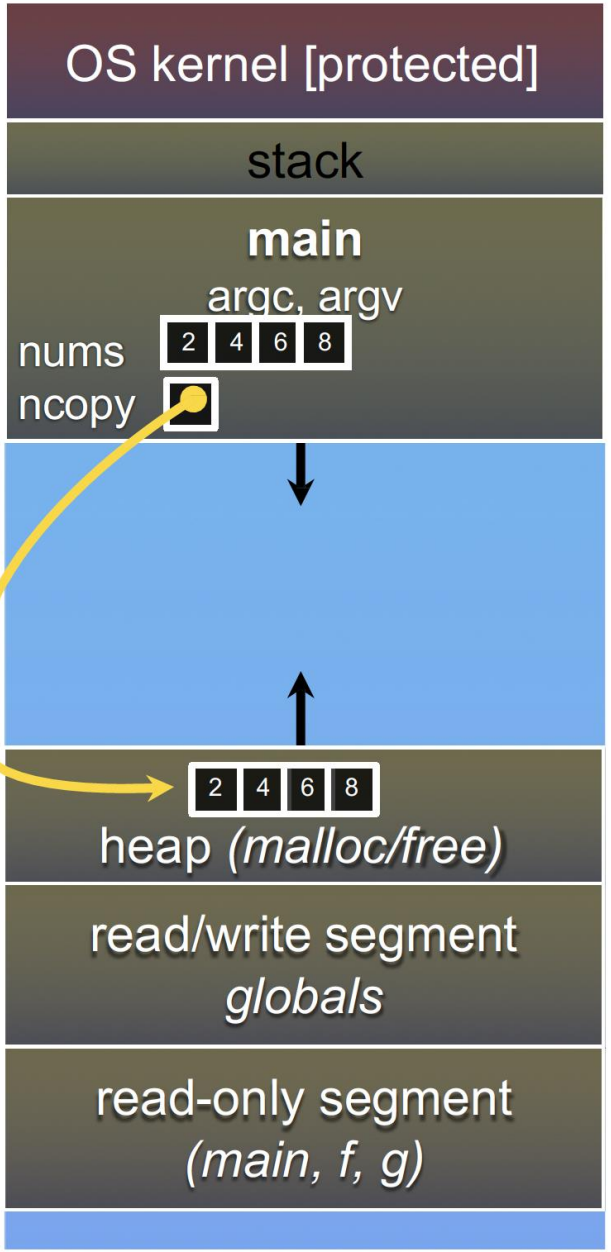
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

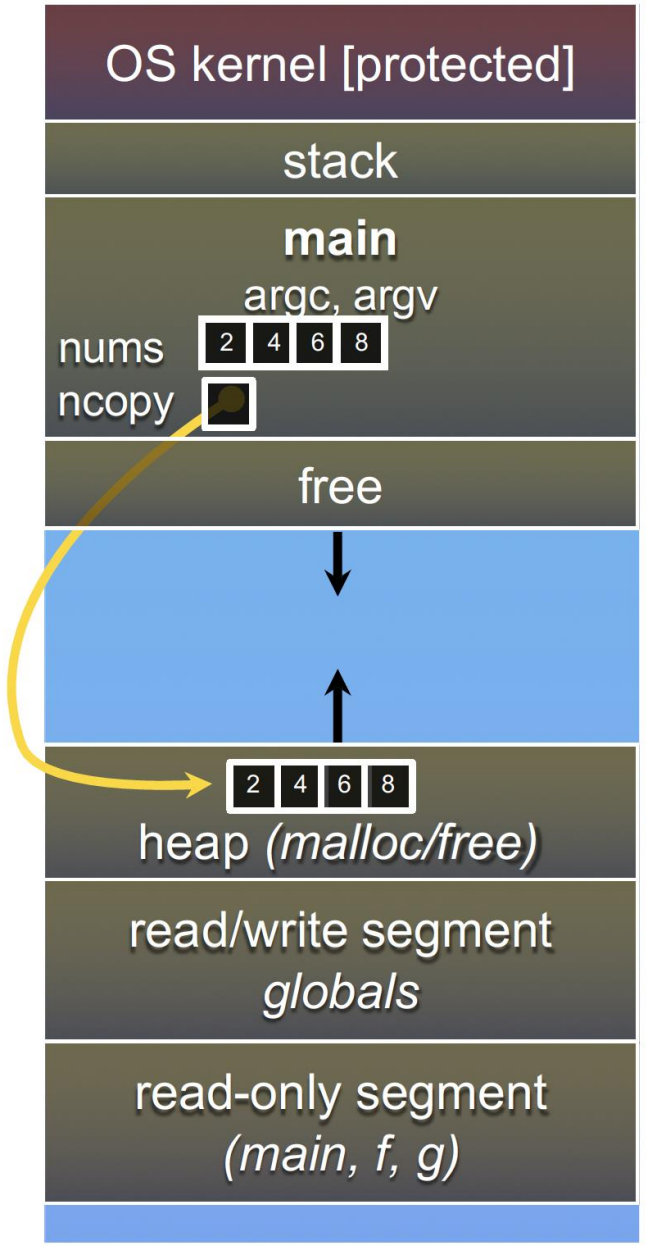
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

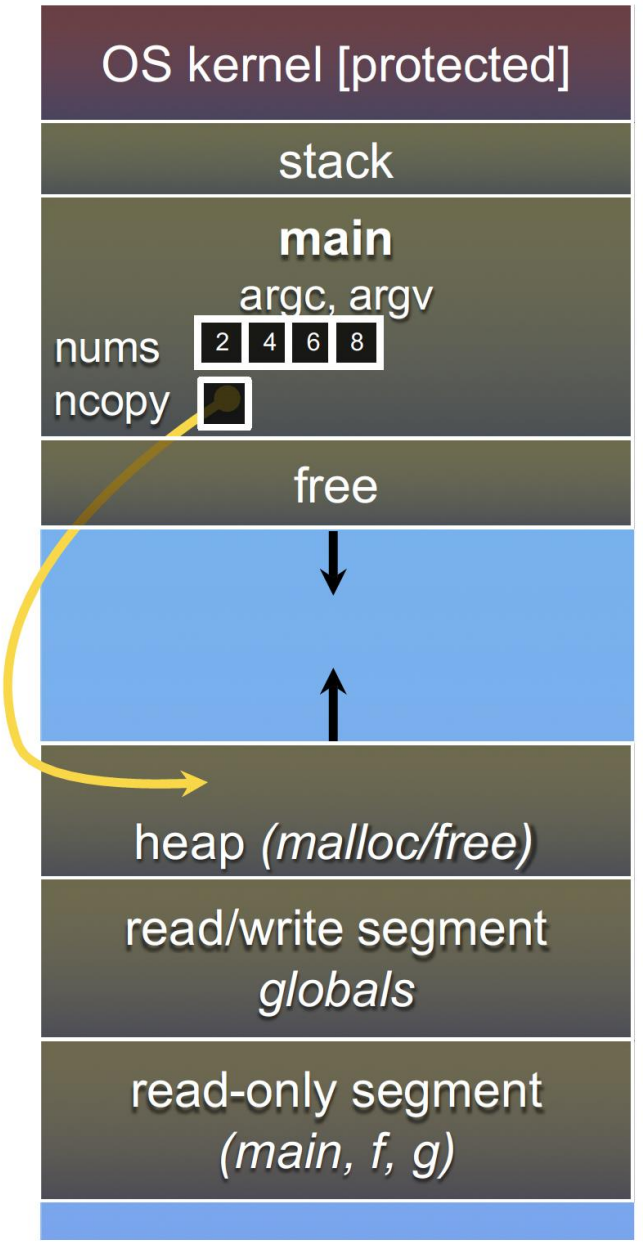
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen

# Process View

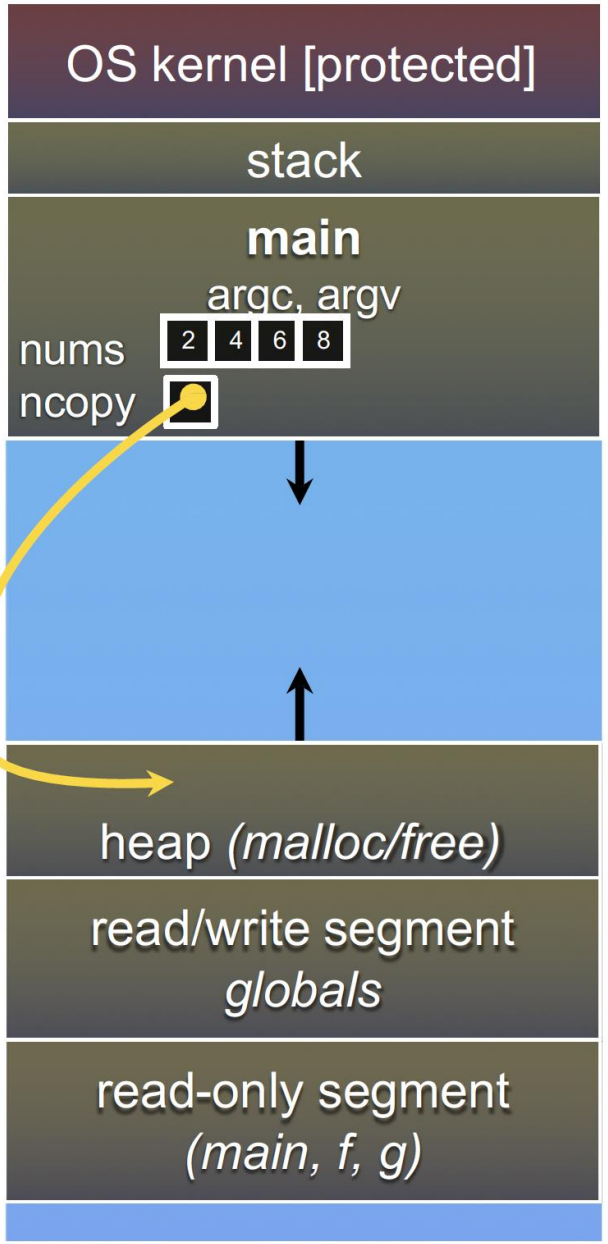
```c
#include <stdlib.h>

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;
}
```




Slide from Prof. Jieyang Chen