CMPSC 473
Operating Systems Design & Construction

CPU Virtualization (cont.)
Dynamic Memory Allocation

Instructor: Ruslan Nikolaev

The Pennsylvania State University

September 7, 2023 – Lecture 6

Interrupts

- CPU/OS design and operation for interrupts exactly like for traps
  - A CPU has its own well-defined set of interrupts
  - OS must implement a handler for each of these (part of OS code, just like trap handlers)
  - Table containing addresses of interrupt handles populated by OS during bootup, address of this table know to the CPU

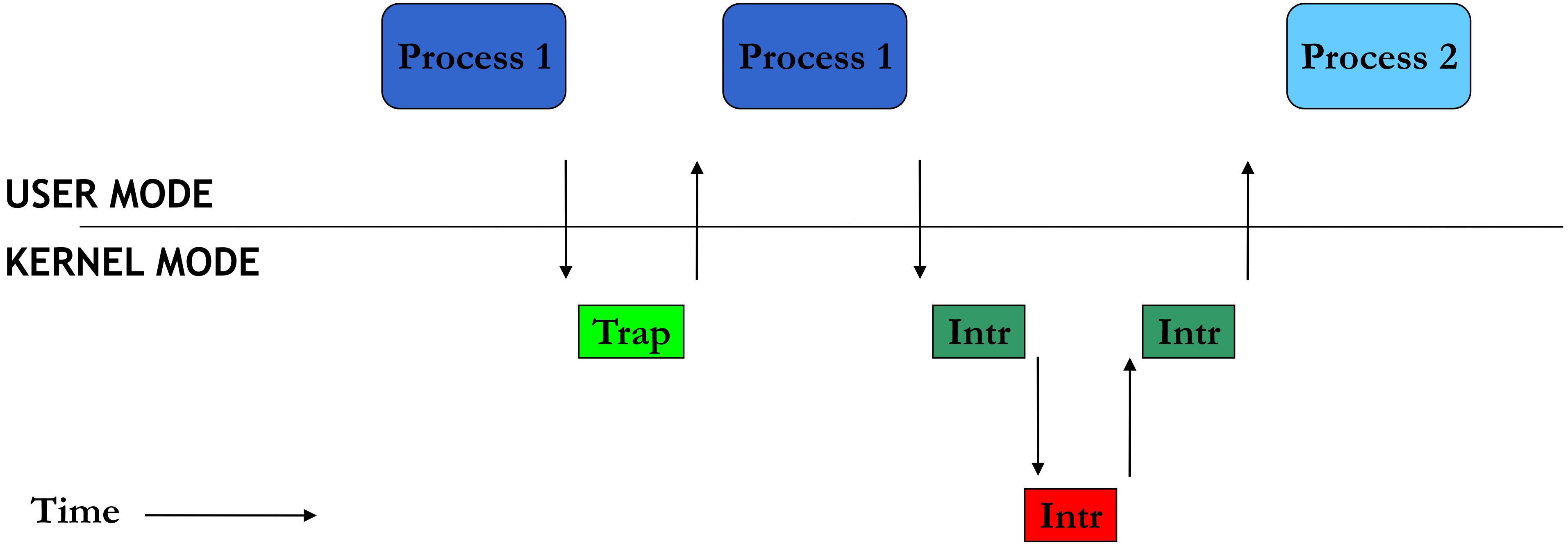
Interrupts

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) $\rightarrow$ k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) $\rightarrow$ proc_t(A) restore regs(B) $\leftarrow$ proc_t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) $\leftarrow$ k-stack(B) move to user mode jump to B's PC	
		Process B ...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

Interrupts and Traps



- Only two ways to enter kernel mode from user mode
  - We previously already considered traps and trap handlers
  - Interrupts and corresponding interrupt handlers – the other way to enter kernel mode

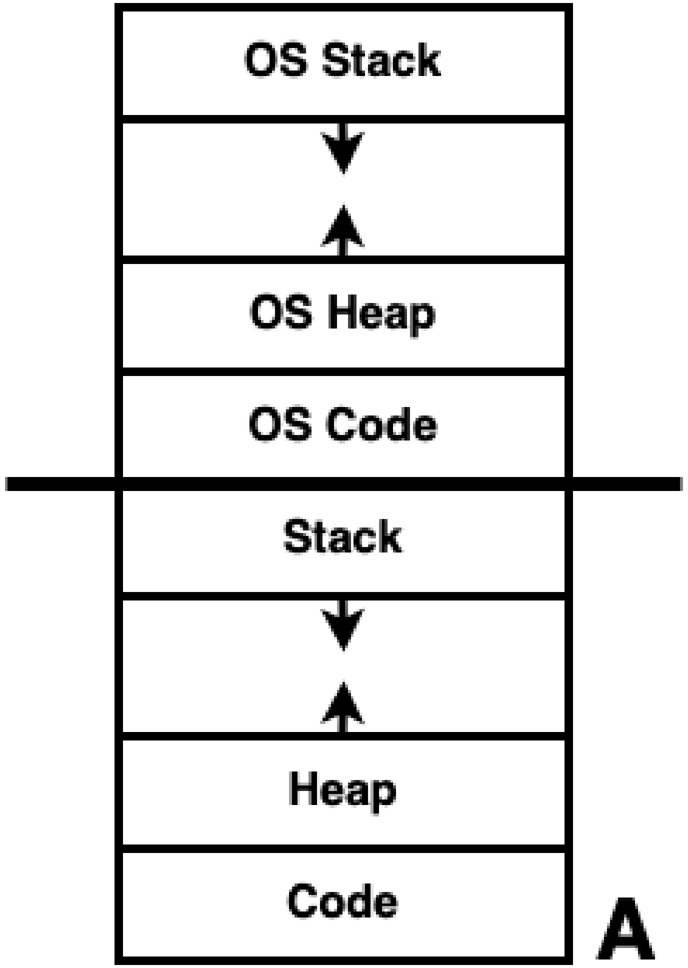
Question

Using recursion is generally considered undesirable in the design of trap/interrupt handlers. Why?

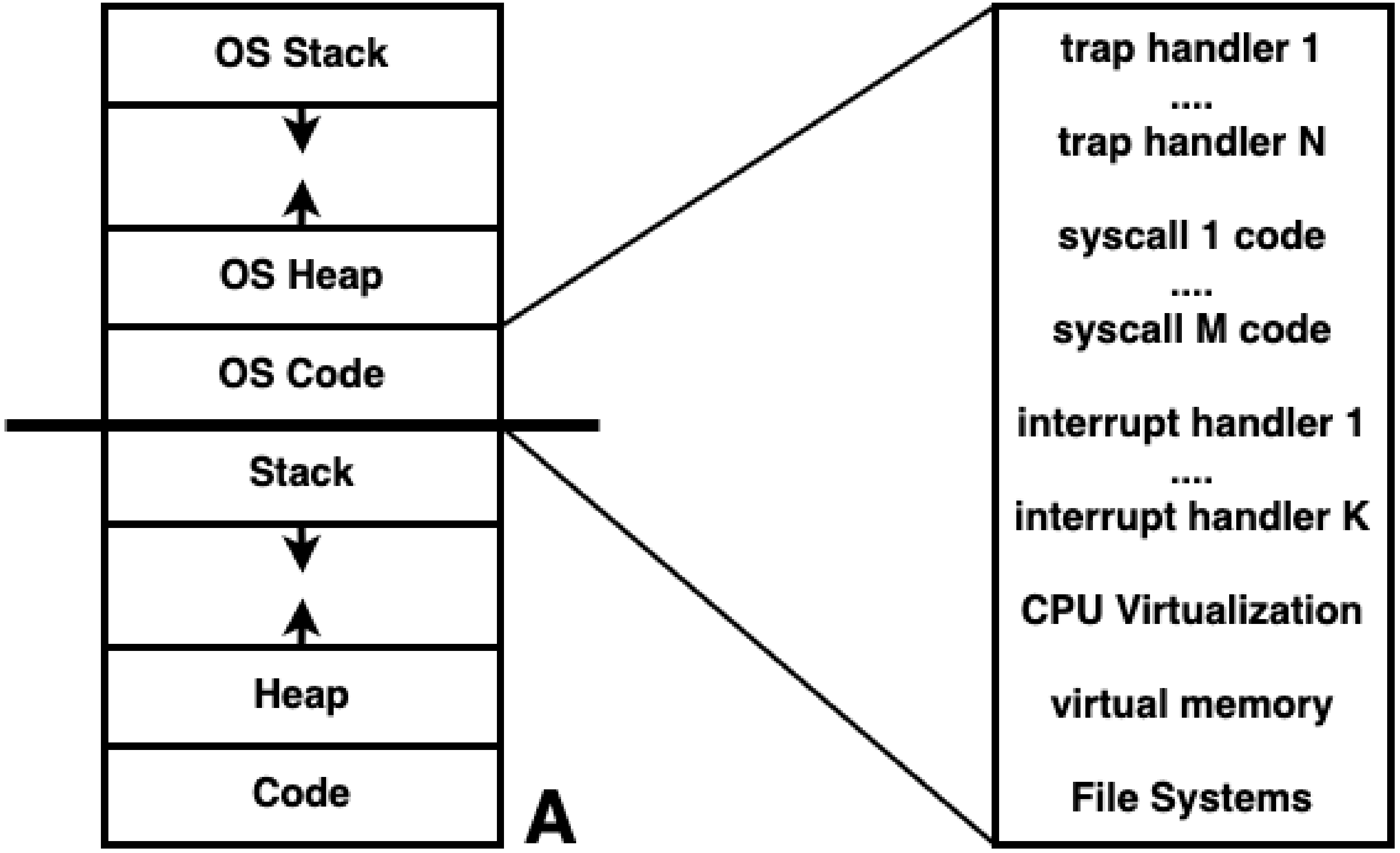
Signals vs. Interrupts/Traps

- Signals are application-level “interrupts”
- Creation: Two ways:
  - OS creates a signal for a process
  - A process creates a signal for another one with the OS help
    - kill system call
- Conveyance: by the OS
- Handling: Just like the OS implements interrupt handlers, a process implements signal handlers for all the signals defined by the OS

Signals vs. Interrupts/Traps

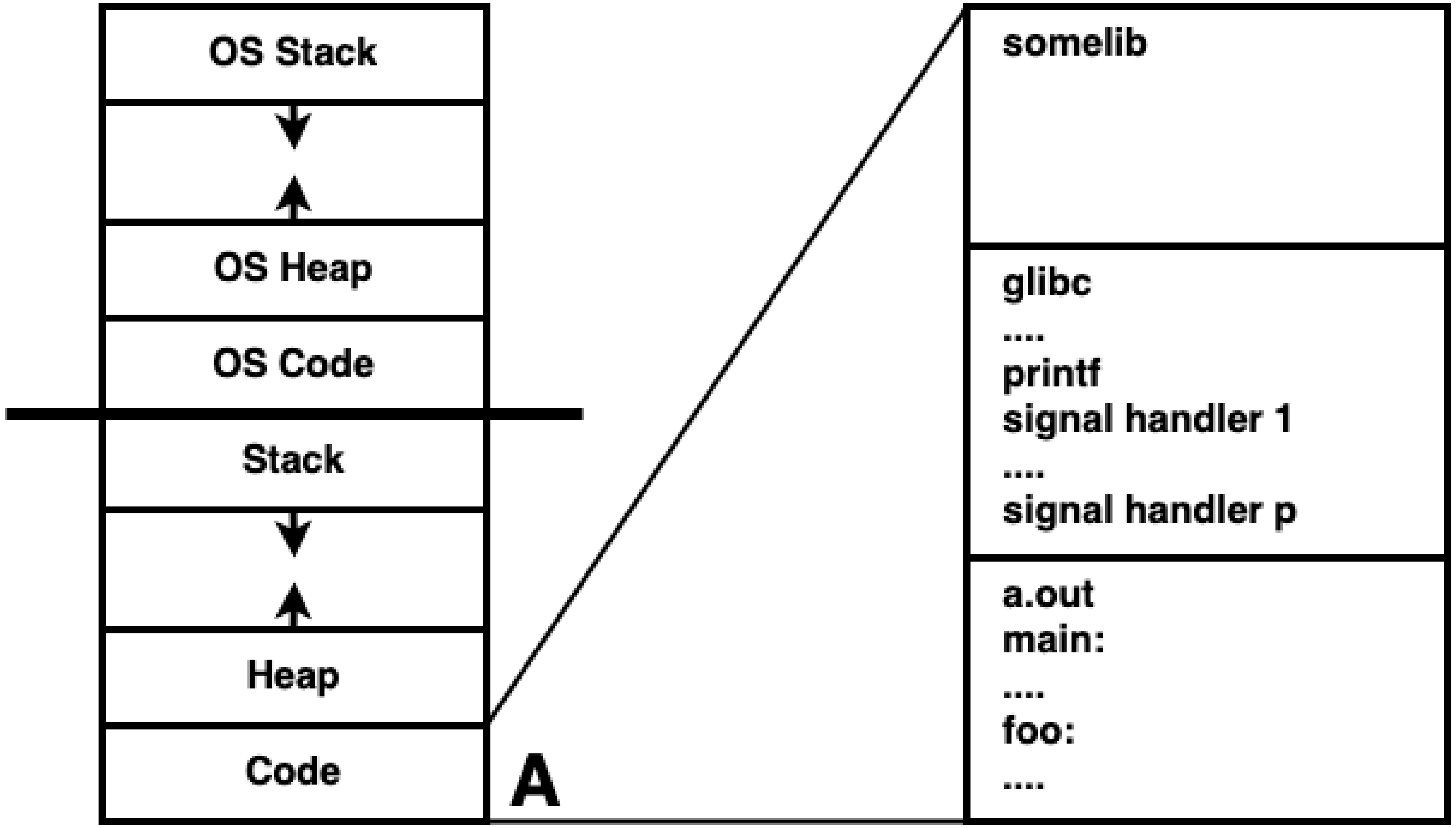


Signals vs. Interrupts/Traps



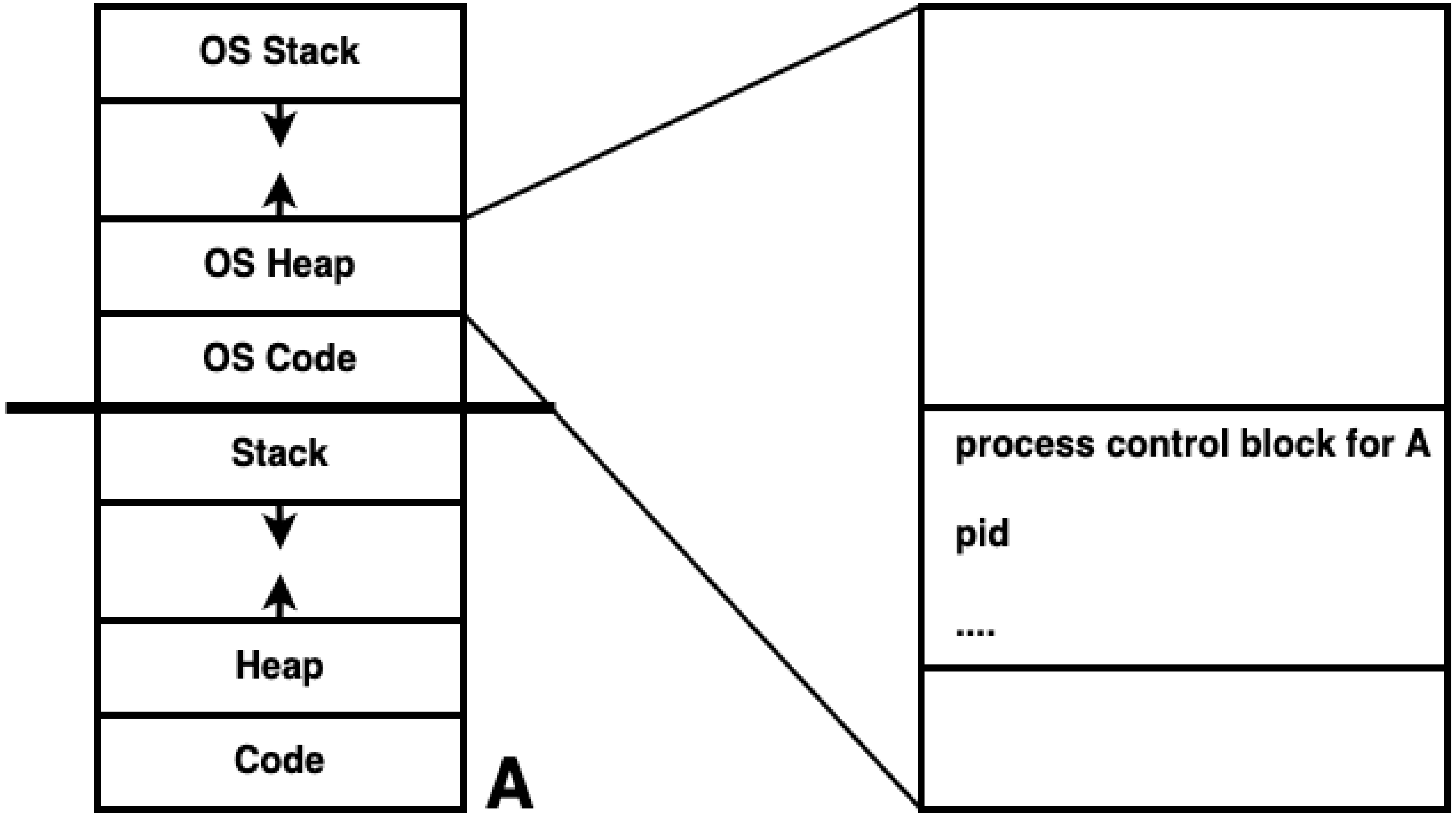
Signals vs. Interrupts/Traps

a.out

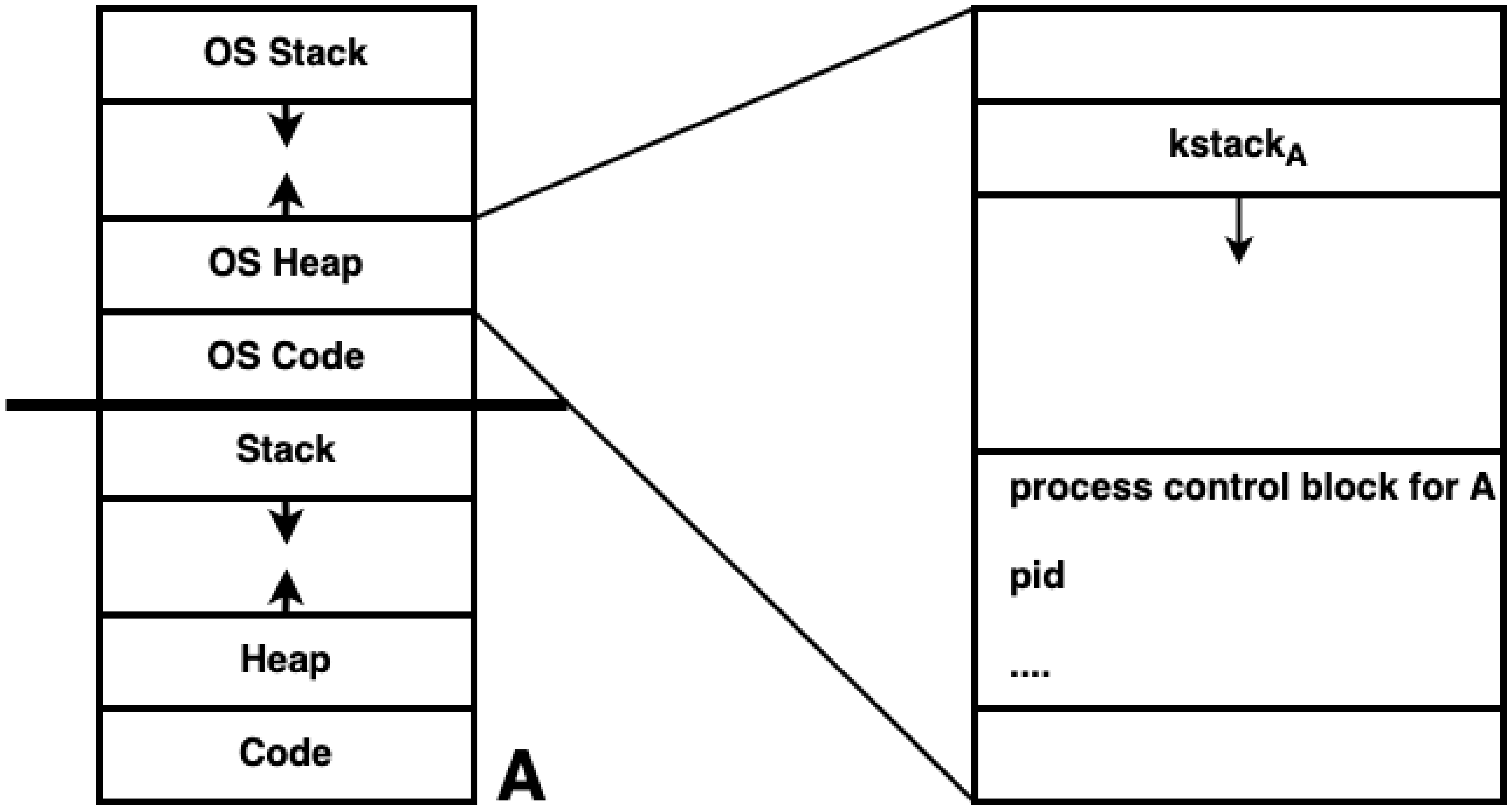


e.g., SIGSEGV

Signals vs. Interrupts/Traps



Signals vs. Interrupts/Traps

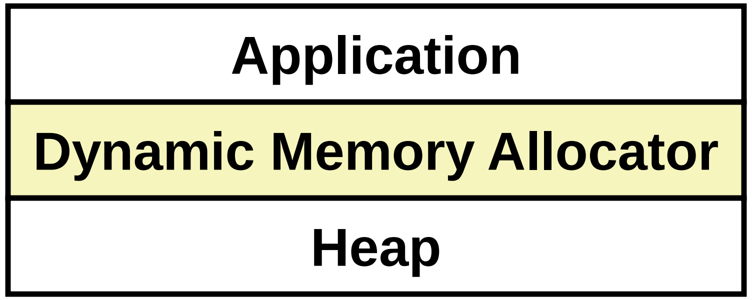


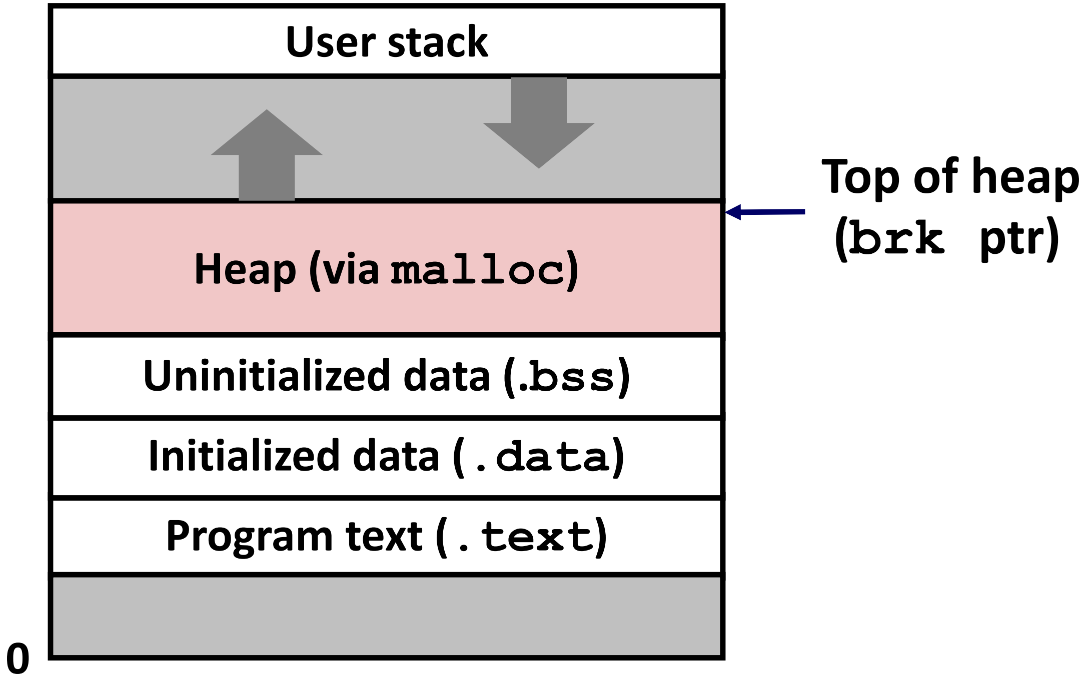
Overview

- Memory management occurs at 2 different granularities
- OS virtual memory management
  - Virtual memory is procured from the OS at a relatively coarse granularity (page)
- User-space dynamic memory allocation
  - Heap and stack management logic makes use of procured pages at a finer granularity (word)
  - We will study this first for the heap

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage the *heap*.





Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - ***Explicit allocator***: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp

The `malloc` Package

```
#include <stdlib.h>
```

```
void *malloc(size_t size)// see C notes on next slide
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

Other functions

- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

The `malloc` Package

```
#include <stdlib.h>
```

```
void *malloc(size_t size)// see C notes on next slide
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

Other functions

- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc:`** Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

Some C notes

- Void pointer
  - What?
    – No associated data type
    – Cannot dereference
    – Need to typecast before dereferencing
  - Why?
    – Need to return a dynamic data type, e.g., malloc
    – Need to implement an opaque object
    – Need to implement a generic function that will take different types of arguments determined at run-time

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Some C notes

- Void pointer
  - What?
    - No associated data type
    - Cannot dereference
    - Need to typecast before dereferencing
  - Why?
    - Need to return a dynamic data type, e.g., malloc
    - Need to implement an opaque object
    - Need to implement a generic function that will take different types of arguments determined at run-time

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Some C notes

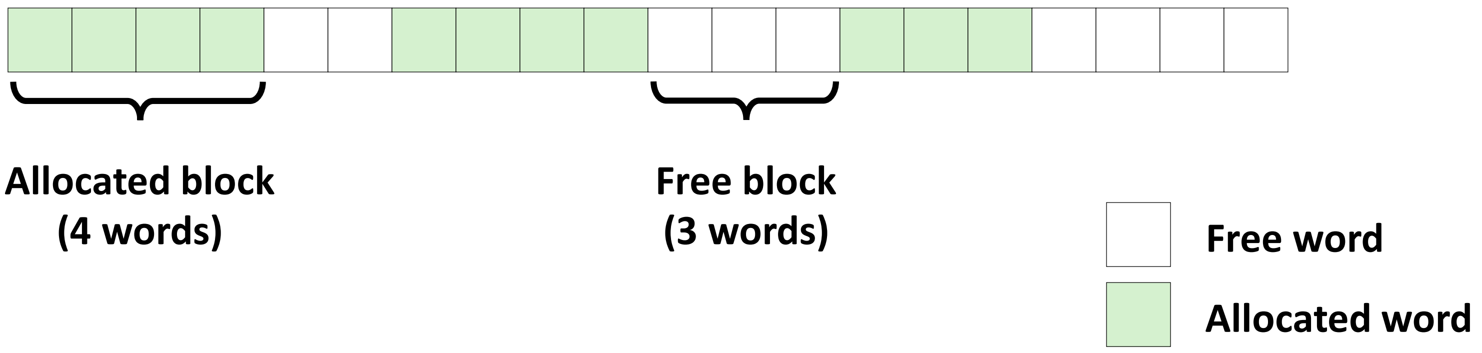
- What is size_t?
- What is errno?
  - IMP: thread safe

malloc Example

```
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
            p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

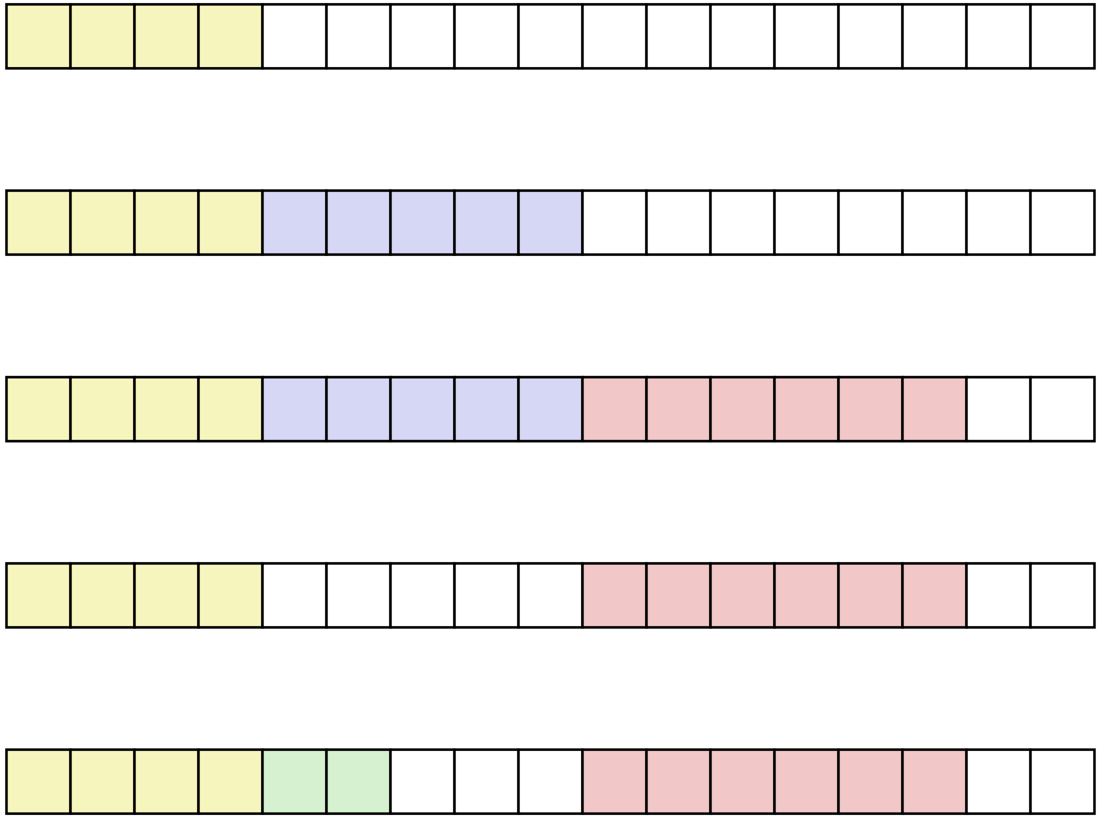
Presentation Assumptions

- Memory is word addressed
- Each square is a word (e.g., 4 bytes for 32-bit x86)
  - Exceptions: in some slides, a square can be just 1 byte (for simplicity) when bytes are implied



Allocation Example

```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```



Each square is 1 byte here

Constraints

- Applications
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**’d block
- Allocators
  - Cannot control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, cannot reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - <u>Can manipulate and modify only free memory</u>
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - <u>Cannot move the allocated blocks once they are **`malloc`**’d</u>
    - *i.e.*, compaction is not allowed