CS 31: Introduction to Computer Systems

19 Strings Recap, OS

04-03-2025

Announcements

- Practice Exam is out – try it out before Monday
- More practice at:
- https://diveintosystems.org/exercises/frontmatter.html

Reading Quiz

Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {
  for(j=0; j< M; j++) {
      sum += arr[i][j];
}}
```

```
for(j=0; j < M; j++) {
  for(i=0; i< N; i++) {
     sum += arr[i][j];
}}
```

Algorithmically, both O(N * M).

Is one faster than the other?

Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {
  for(j=0; j< M; j++) {
      sum += arr[i][j];
}}
```

A. is faster.

```
for(j=0; j < M; j++) {
  for(i=0; i< N; i++) {
     sum += arr[i][j];
}}
```

B. is faster.

Algorithmically, both O(N * M).

Is one faster than the other?

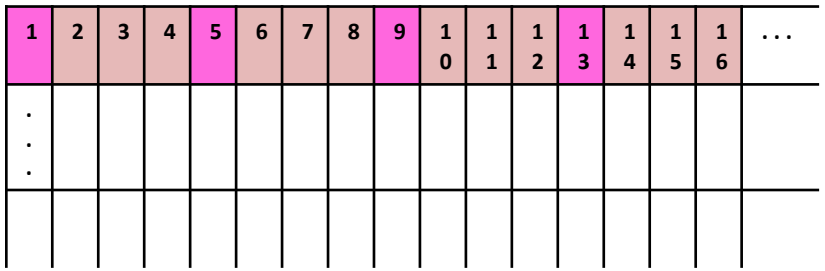
C. Both would exhibit roughly equal performance.

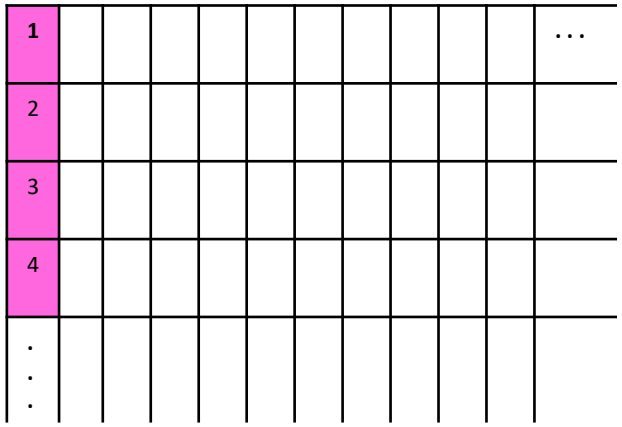
Cache Conscious Programming

The first nested loop is more efficient if the cache block size is larger than a single array bucket (for arrays of basic C types, it will be).

```
for(i=0; i < N; i++) {
  for(j=0; j< M; j++) {
      sum += arr[i][j];
}}
```

```
for(j=0; j < M; j++) {
  for(i=0; i< N; i++) {
     sum += arr[i][j];
}}
```

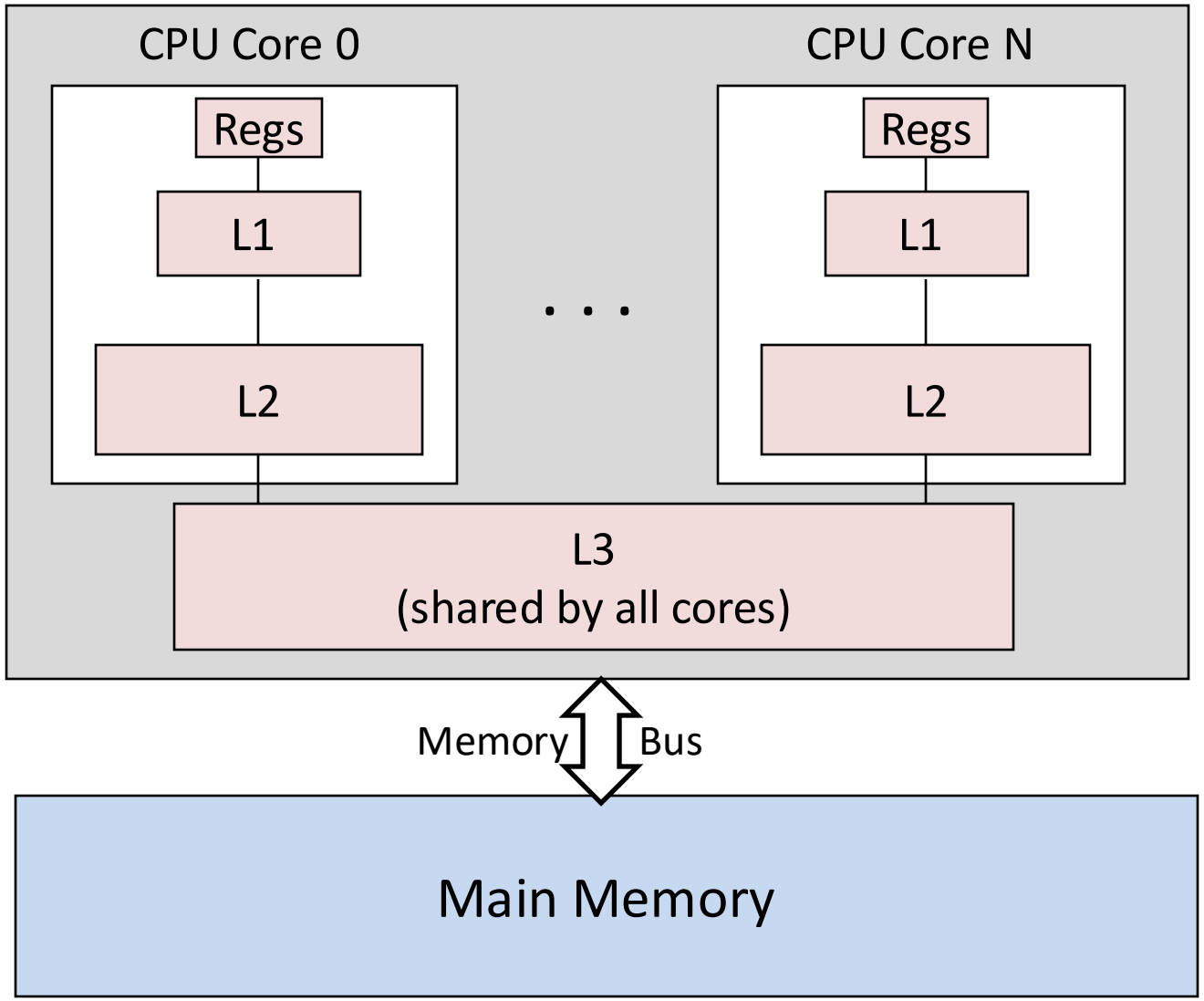




(ex) 1 miss every 4 buckets vs. 1 miss every bucket

FYI: Example Cache Organization

Multicore processor: multiple CPU cores/chip



Example Sizes & Access Times

(KB: 2^{10} MB: 2^{20} GB: 2^{30})

L1: Size: 32 KB
Access: 4 cycles
write through
low associativity

L2: Size: 256 KB
Access: 10 cycles

L3: Size: 8 MB
Access: 40 cycles
write back
higher associativity

Main Memory (off chip):
Size: 16 GB
Access: 100 cycles

Program Efficiency and Memory

- Be aware of how your program accesses data
  - Sequentially, in strides of size X, randomly, …
  - How data is laid out in memory

- Will allow you to structure your code to run much more efficiently based on how it accesses its data

- Don’t go nuts…
  - Optimize the most important parts, ignore the rest
  - “Premature optimization is the root of all evil.” -Knuth

Amdahl's Law

Idea: an optimization can improve total runtime at most by the fraction it contributes to total runtime

> If program takes 100 secs to run*, and you optimize a portion of the code that accounts for 2% of the runtime,* the best your optimization can do is improve the runtime by 2 secs!

Amdahl's Law tells us to focus our optimization efforts on the code that matters:

> Speed-up what is accounting for the largest portion of runtime to get the largest benefit. And, don't waste time on the small stuff.

Up Next:

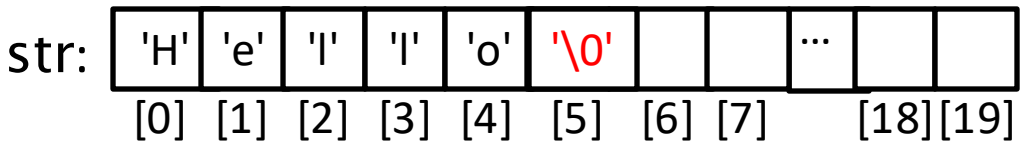
- Review of C Strings
- Operating systems, Processes

Review: Strings in C

- char is a basic C type that stores a numeric value usually for storing a single letter value (e.g. ‘a’)
  - Variable stores the ASCII encoding for the character
  - Can perform arithmetic ops on its ascii value:

```
char ch = 'a';
ch++; //ch gets ascii value of 'b'
```

- String literal value between " ": "hello"
- Strings are stored as arrays of char with special terminating null (end of string) char ('\0’) at the end. char str[20]; // array of 20 chars



The string "hello" stored in str

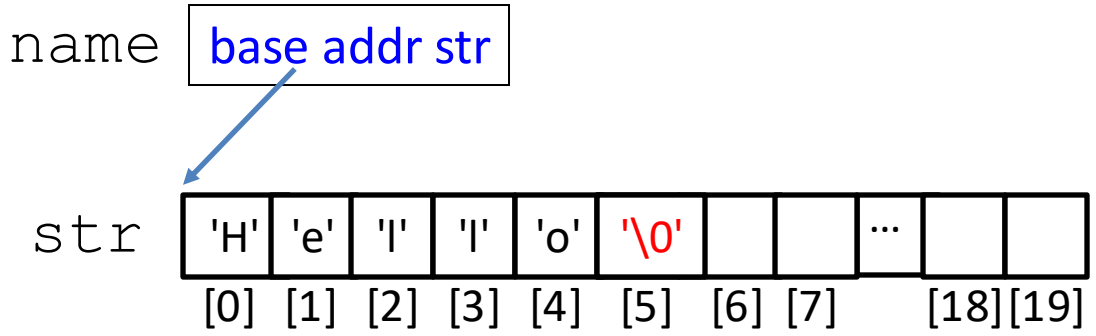
C Strings and Pointers

- Often accessed as (char *)

```
char str[20];
str = “hello”;
char *name;
name = str;
printf(”%s”, str);
printf(”%s”, name);
```

Q: How to print “ello” given these two variables str and name?

how many different ways can you come up with?



C Strings and Pointers

```
char str[20];
str = "hello";
char *name;
```

Q: How to print "ello" given these two variables str and name?

starting at str[1] is the string "ello"

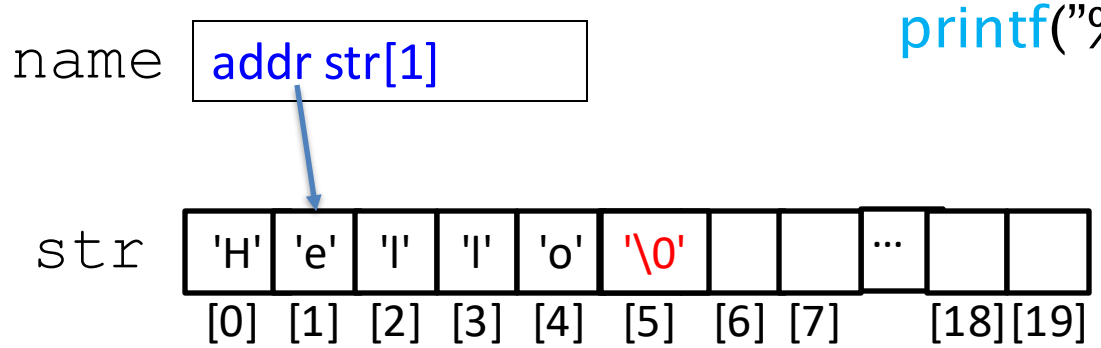
```
printf("%s", &str[1]);
```

or:

```
name = str;
printf("%s", &name[1]);
```

or:

```
name = &str[1];
printf("%s", name);
```



C Strings and Pointers

```
char str[20], str2[30], *str3;
str[0] = 'H';
str[1] = 'i';
str[2] = '\0';
```

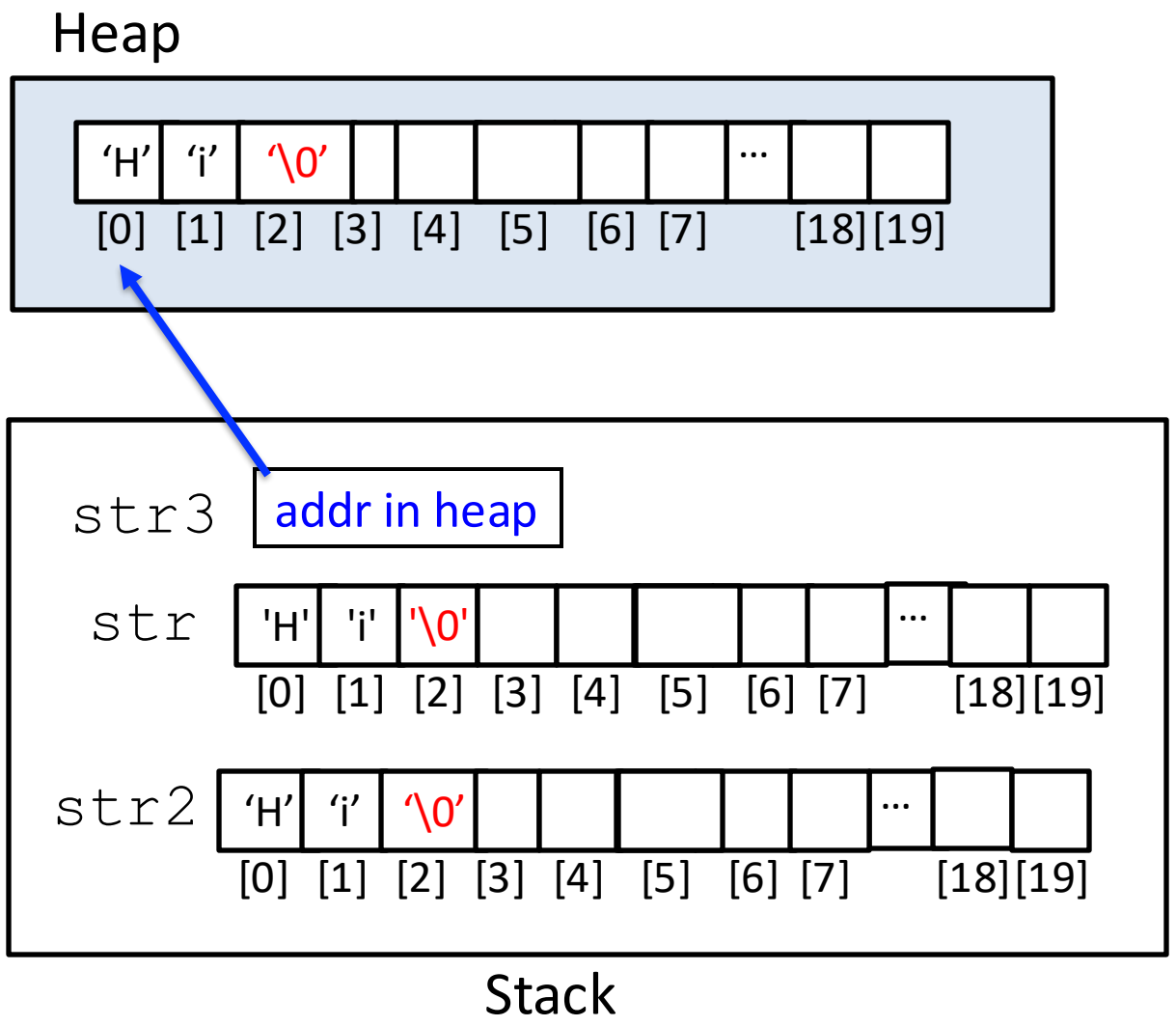
Q: How to initialize str2 and str3 to have the same string value, "Hi", as str?

(three separate strings, with same string value "Hi")

```
//copying strings using C code
str3 = malloc( sizeof (char) * 20);
i=0;
while(str[i] != '\0') {
   str2[i] = str[i];
   str3[i] = str[i];
   i++;
}
```

are we done?

```
str2[i] = '\0';
str3[i] = '\0';
```



C String Library

- #include<string.h>
  - Functions to manipulate strings
    - strcpy, strdup, strlen, strcat, strcmp, strstr, ...
  - User MUST ALLOCATE SPACE for string
    - Declare an array of char (char str[20];)
    - Declare a char * and dyamically allocate space
    - Make sure to include space for '\0' char
  - Library functions use '\0' to find end of string
    - Don't need to pass in size of string to function
      like you do for passing non-string arrays to function

C string examples

```
char str1[20], str2[40];
int val;

// remember to null terminate strings!!
str1[0]='E';
str1[1]='m';
str1[2]='m';
str1[3]='a';
str1[4]='\0';

printf("%s\n", str1); // prints Emma
scanf("%s", str2);
// reads a string from user stores in str2
```

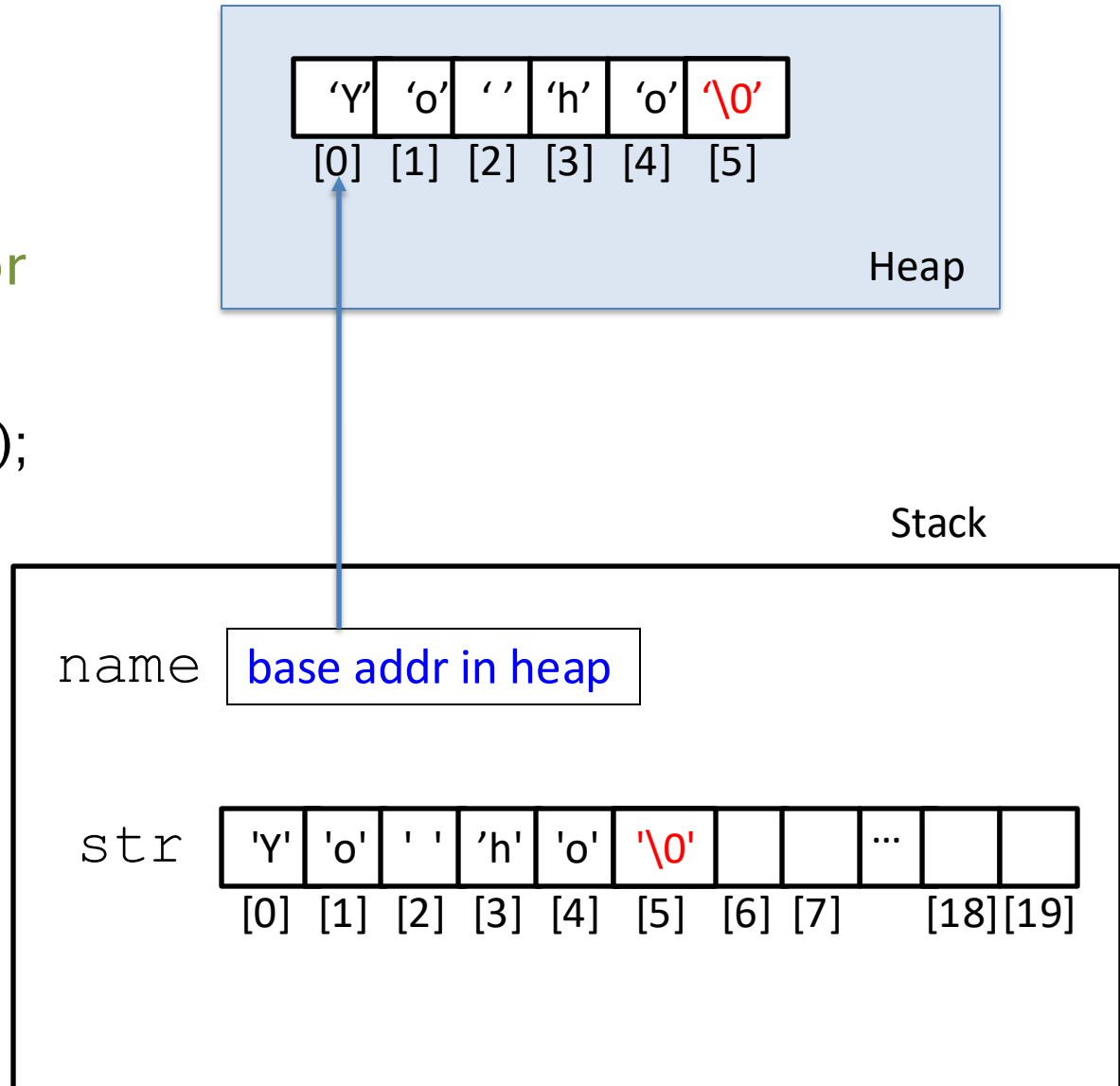
`printf` prints char values starting at `str1[0]` until finds first bucket storing `'\0'` (end of string)

```
// some string library functions do null termination
// for you:
strcpy(str2, "hello");  // str2 better have enough space!
printf("%s %s %d\n", str2, str1, strlen(str1));
// prints: hello Emma 4

// strcmp: compares ascii values of corresponding
                chars in str1 and str2 until '\0'
// returns:   0 if equal,
//               negative value if str1< str2,
//               positive value if str1 > str2
val = strcmp(str1, str2);
if(val) {
        printf("str1 and str2 are not equal\n");
}
```

Dynamically Allocated Strings

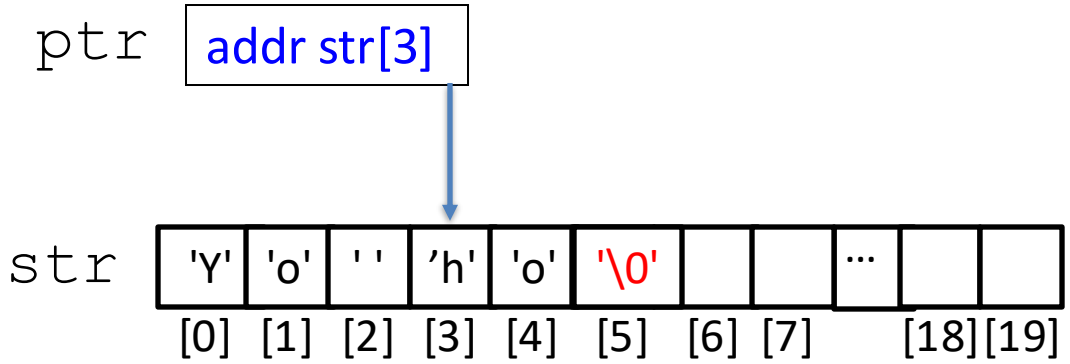
```
char *name, *ptr, str[20];
strcpy(str, "Yo ho");
//allocate one extra bucket in the array for
//the null character
name = malloc(sizeof(char)*(strlen(str)+1));
strcpy(name, str);
```



strstr: find string in a string

```
char *name, *ptr, str[20];
str = “Yo Ho”;
ptr = strstr(str, “ho”);
printf(“%s”, ptr);
```

Q: draw the stack diagram and output of the print statement.



Try out

```
char *str2, *ptr, str[64];
strcpy(str, “Hi How Are you?”);
```

1. code to see if `A` is in the string str (make your code work for any value stored in str, not just this example)
2. if so, create a copy of the string starting at the first `A` in str2.
3. if not concatenate (add to the end of) to str the string “ no As in here.”
4. compare str2 and str strings, and print out a message indicating which is greater than which or if they are equal

strcpy, strlen, strcat, strcmp, strstr, strchr

```
char *str2, *ptr, str[64];
strcpy(str, "Hi How Are you?");
```

1. code to see if `A` is in the string str (make your code work for any value stored in str, not just this example)

```
ptr = strchr(str, 'A'); //<-Note the single quotes to find a character
```

alternatively:

```
ptr = strchr(str, "A"); //<-Note the double quotes to a substring within a
                        // string
```

```
char *str2, *ptr, str[64];
strcpy(str, "Hi How Are you?");
```

1. code to see if `A` is in the string str (make your code work for any value stored in str, not just this example)
2. if so, create a copy of the string starting at the first `A` in str2.

```
ptr = strchr(str, 'A');  // Step 1
if(ptr != NULL) {
   str2 = malloc( sizeof ( char)* ( strlen(ptr) + 1));  //start at pointer, allocate
                                //an additional byte for the null terminating character
   if(str2 == NULL) { exit(1); }
   strcpy(str2, ptr);
}
```

```
char *str2, *ptr, str[64];
strcpy(str, “Hi How Are you?”);
```

1. code to see if `A` is in the string str (make your code work for any value stored in str, not just this example)
2. if so, create a copy of the string starting at the first `A` in str2.
3. if not concatenate (add to the end of) to `str` the string `“ no As in here”`

```
ptr = strstr(str, “A”);                                    // Step 1
if(ptr != NULL) {
   str2 = malloc ( sizeof ( char )*( strlen (ptr) + 1)); // Step 2
   if(str2 == NULL) { exit(1); }
   strcpy(str2, ptr);
} else {                                                   //Step 3
   strcat(str, “ no As in here”);
}
```

4. compare str2 and str strings, and print out a message indicating which is greater than which or if they are equal

```
void cmp_strings(str, str2){
    int result;
    result = strcmp(str2, str);
    if (result == 0) {
        printf("str and str2 are equal\n");
    } else if (result > 0) {
        printf("str2 is bigger than str\n");
    } else {
        printf("str is bigger than str2\n");
    }
}
```

```
int result;
char *str2, *ptr, str[64];

strcpy(str, "Hi How Are you?");
ptr = strstr(str, "A");
if(ptr != NULL) {
  printf("ptr = %s\n", ptr);
  str2 = malloc(sizeof(char)*(strlen(ptr) + 1));
  if(str2 == NULL) { exit(1); }
  strcpy(str2, ptr);
} else {
  strcat(str, "no A");
}
result = strcmp(str2, str);
if(result == 0) {
  printf("str and str2 are equal\n");
} else if (result > 0) {
  printf("str2 is bigger than str\n");
} else {
  printf("str is bigger than str2\n");
}
printf("str = %s\n", str);
printf("str2 = %s\n", str2);
```

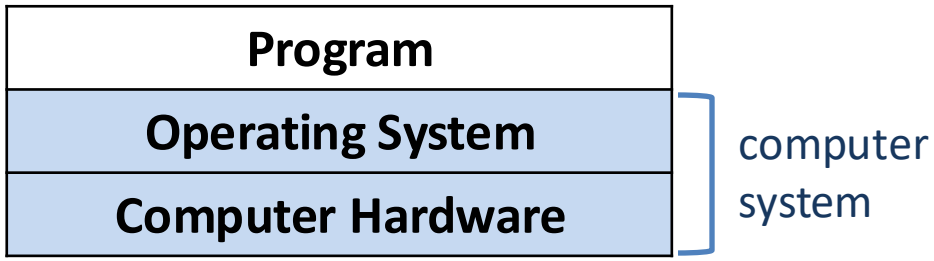
Safe versions of string library funcs

Take a size limit to avoid writing beyond the bounds of the destination array

```
#define N 64
...
char str1[N], str2[128];
...
strncpy(str1, str2, N);
str1[N-1] = '\0';
// need to '\0' terminate if str2 >= N chars
```

In general, uses these versions!

What do we know so far?



(1) How a Computer Runs a Program:

./a.out to x86 instructions being executed by HW

Program: how instructions and data are encoded to run on HW

HW: how implemented to run program instrs on program data

- CPU implementation and how it works (Fetch, Decode, Execute, Store)
- Memory Hierarchy, different storage types/devices
- CPU Cache implementation(DM and SA) and how it works

(2) How to Efficiently Run Programs

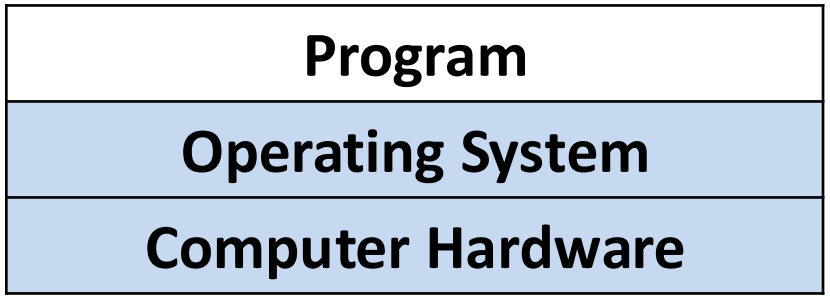
Compiler's role in producing efficient code (little bits of this)

The Memory Hierarchy and its effect performance

- Caching motivated by lots of locality in program execution

The Operating System

(1) Its role in how a Computer Runs a Program:



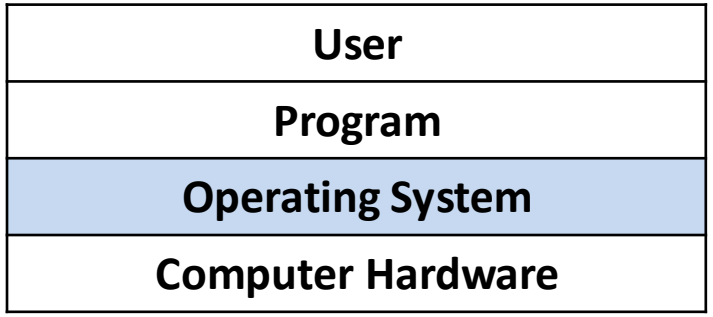
./a.out to IA32 instructions being executed by HW

(2) How to Efficiently Run Programs

- Compiler's role in producing efficient code
- The Memory Hierarchy and its effect performance
- OS abstractions for running programs efficiently

What is an Operating System?

Sits between the HW and the User/Program:



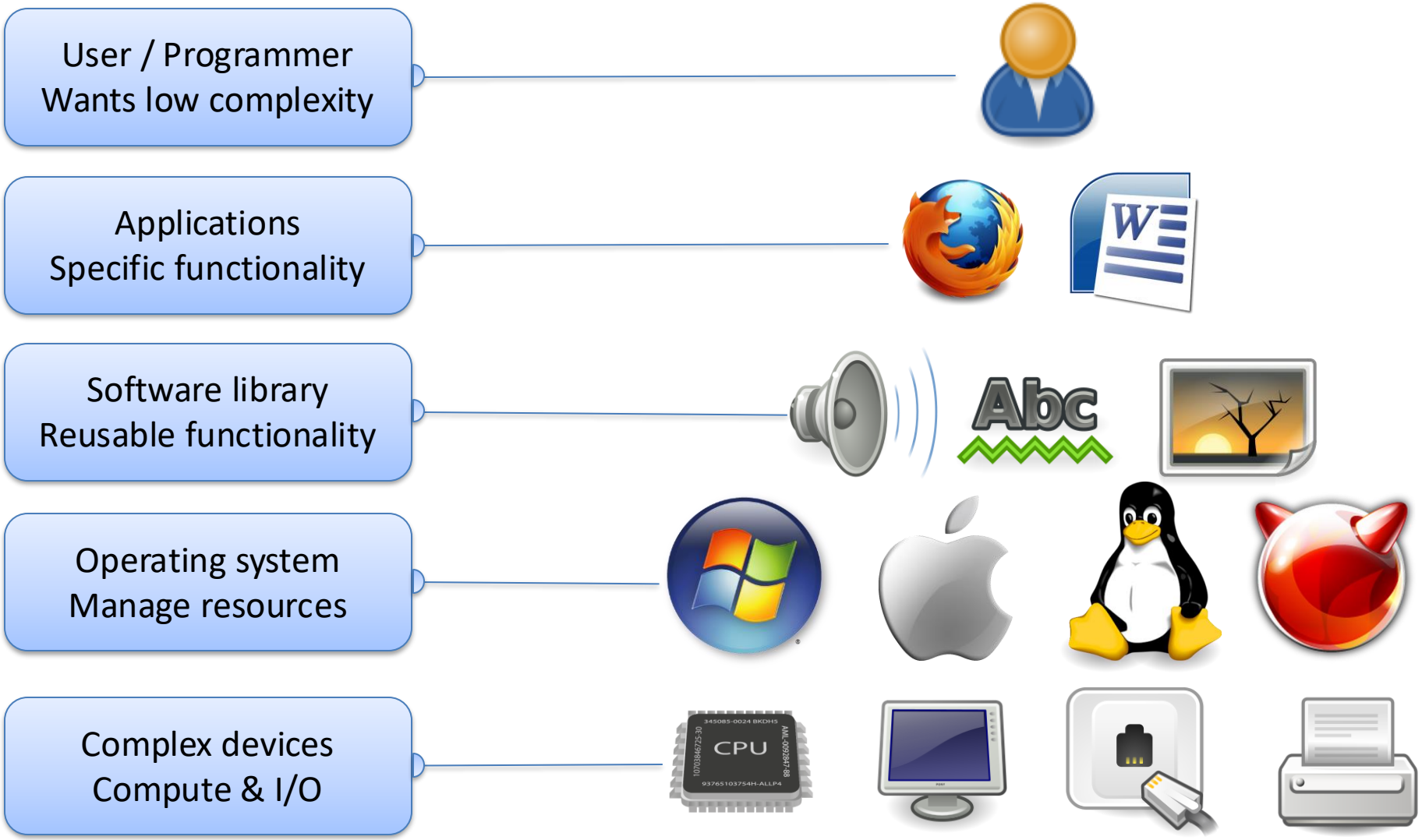
1. Manages the underlying HW
   - Coordinates shared access to HW

     ps –A (lots of processes sharing CPU, RAM, disk, …)
   - Efficiently schedules/manages HW resources
2. Provides easy-to-use interface to the HW
   - just type: ./a.out to run a program

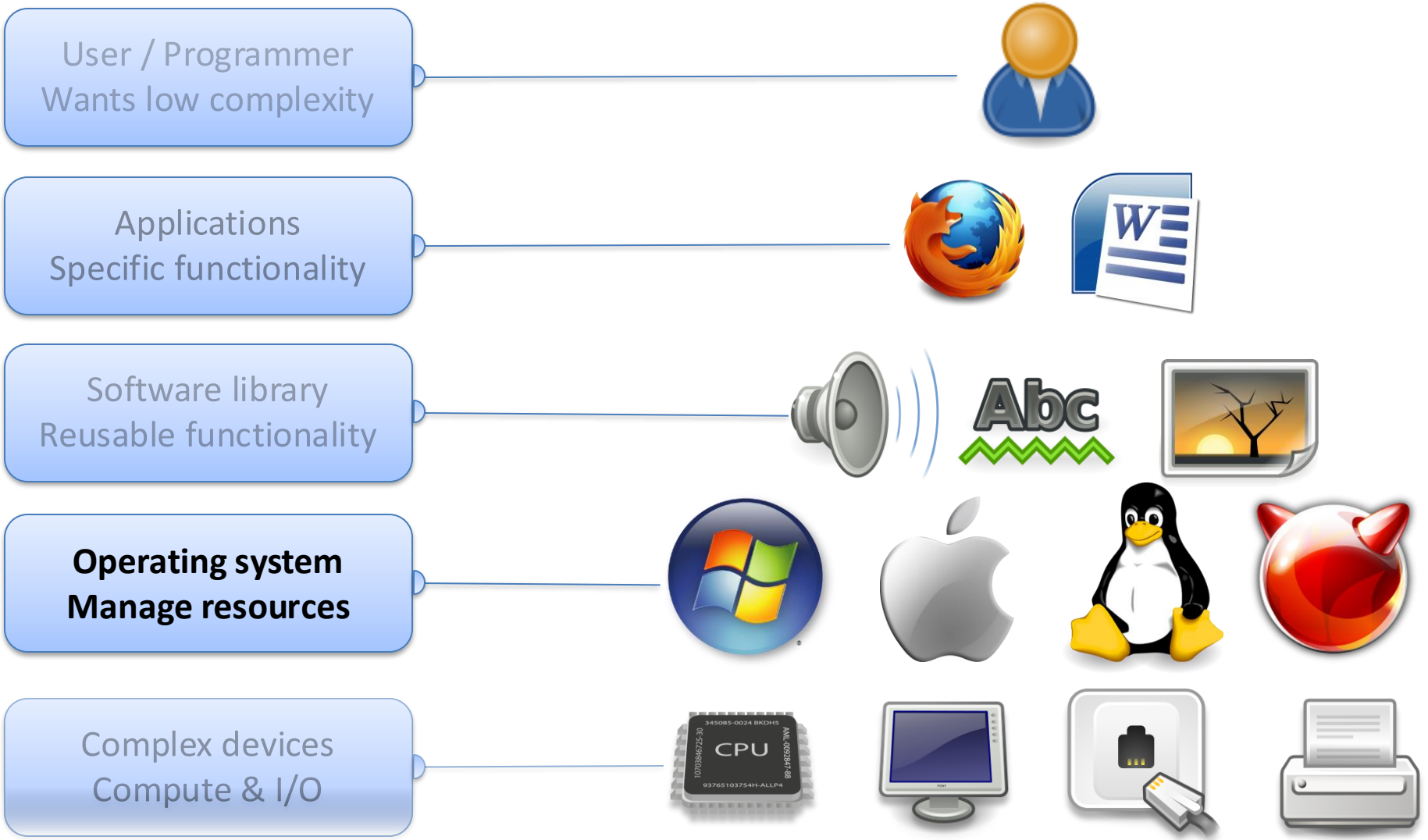
OS Big Picture Goals

- OS is a layer of code between user programs and hardware.
- Goal: Make life easier for users and programmers.
- How can the OS do that?

Abstraction



Abstraction



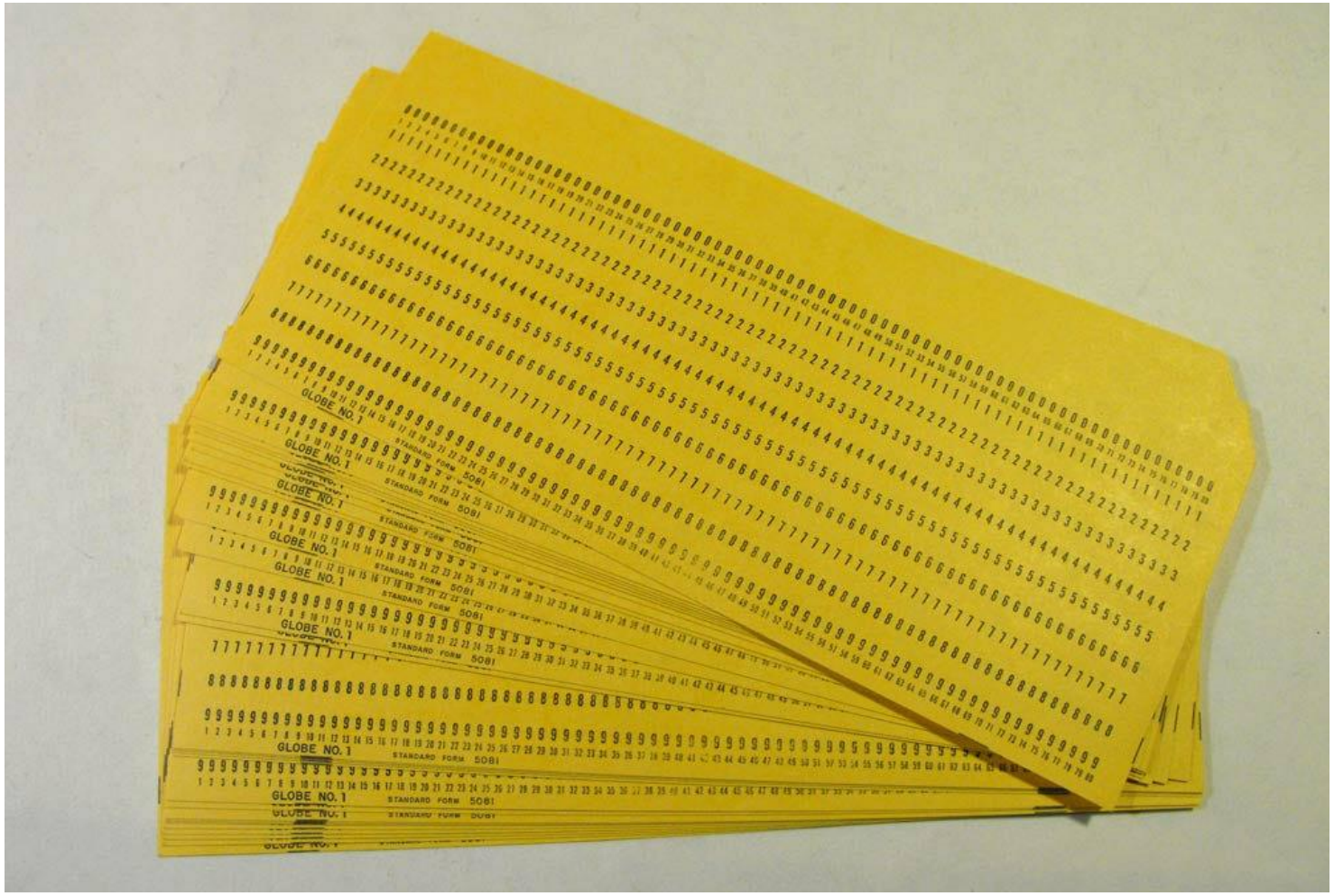
Key OS Responsibilities

1. Simplifying abstractions for programs
2. Resource allocation and/or sharing
3. Hardware gatekeeping and protection

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory

Before Operating Systems



- One program executed at a time…

Why is it not ideal to have only a single program available to the hardware?

A. The hardware might run out of work to do.

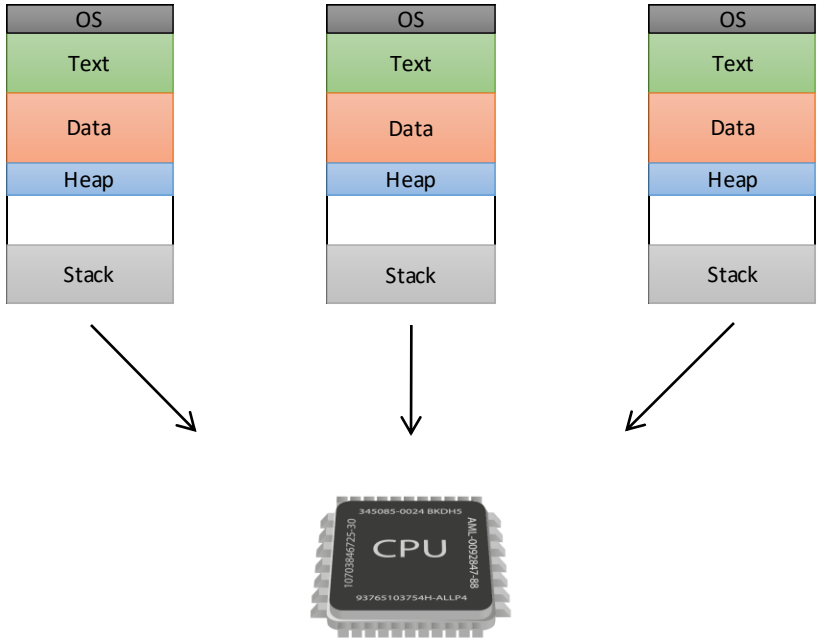
B. The hardware won't execute as quickly.

C. The hardware's resources won't be used as efficiently.

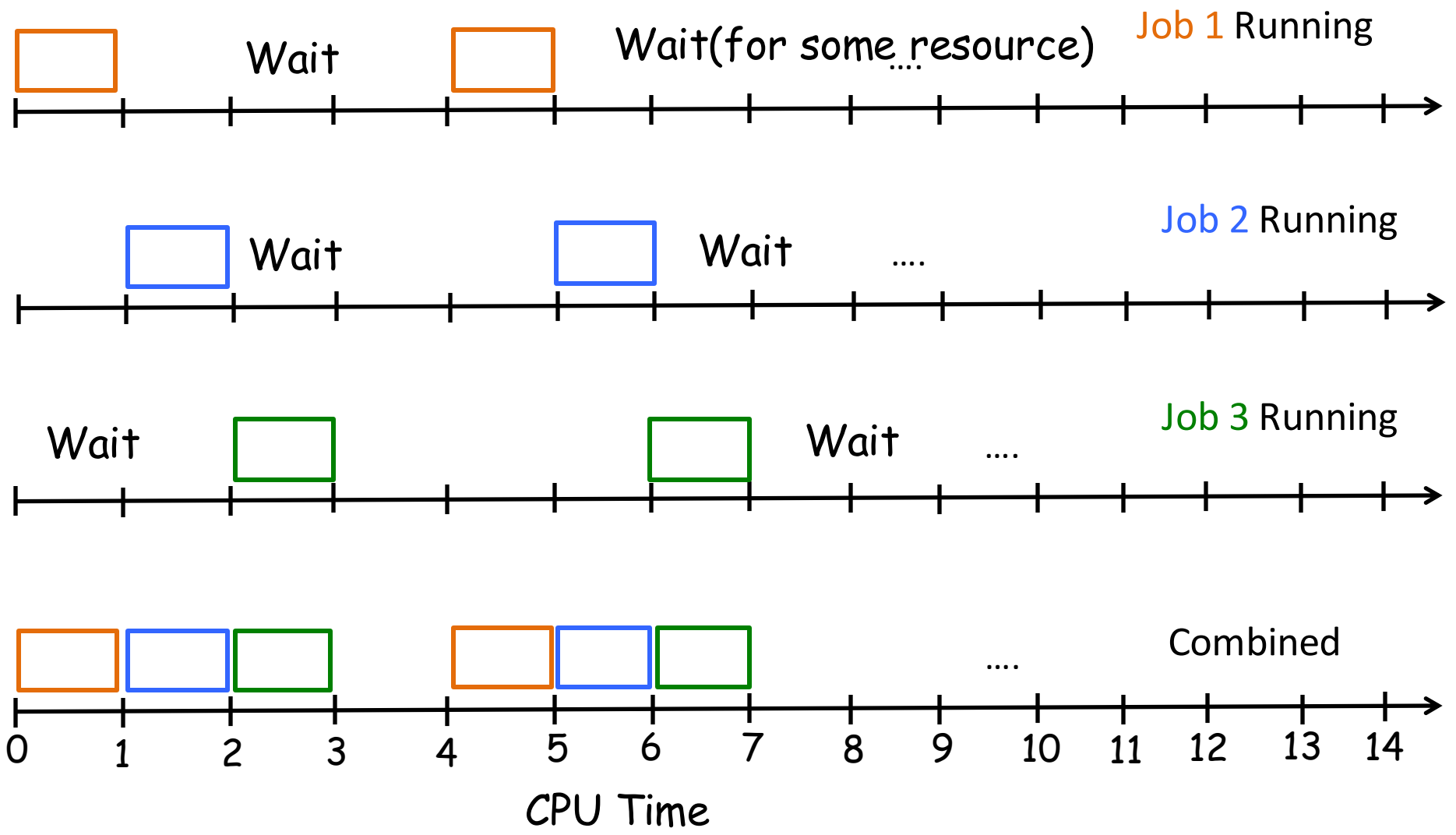
D. Some other reason(s). (What?)

Today: Multiprogramming

- Multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.



Multiprogramming on one core



How many programs do you think are running on a typical desktop computer?

A. 1-10

B. 20-40

C. 40-80

D. 80-160

E. 160+

Running multiple programs

More than 200 processes running on a typical desktop!

- Benefits: when I/O issued, CPU not needed
  - Allow another program to run
  - Requires yielding and sharing memory

- Challenges: what if one running program…
  - Monopolizes CPU, memory?
  - Reads/writes another's memory?
  - Uses I/O device being used by another?

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory
- Into desirable conveniences: illusions
  - Simple, easy-to-use resources
  - Multiple/unlimited number of processors
  - Large/unlimited amount of memory

Virtualization

- Rather than exposing real hardware, introduce a “virtual”, abstract notion of the resource

- Multiple virtual processors
  - By rapidly switching CPU use
- Multiple virtual memories
  - By memory partitioning and re-addressing
- Virtualized devices
  - By simplifying interfaces, and using other resources to enhance function

We'll focus on the OS 'kernel'

- "Operating system" has many interpretations
  - E.g., all software on machine minus applications (user or even limited to 3rd party)

- Our focus is the *kernel*
  - What's necessary for everything else to work
  - Low-level resource control
  - Originally called the nucleus in the 60's

The Kernel

- All programs depend on it
  - Loads and runs them
  - Exports system calls to programs
- Works closely with hardware
  - Accesses devices
  - Responds to interrupts (hardware events)
- Allocates basic resources
  - CPU time, memory space
  - Controls I/O devices: display, keyboard, disk, network

Tron, 1982

Kernel provides common functions

- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in central place (kernel)
  - Called by programs ("system calls")
  - Or accessed in response to hardware events
- What should functions be?
  - How many programs should benefit?
  - Might kernel get too big?

OS Kernel

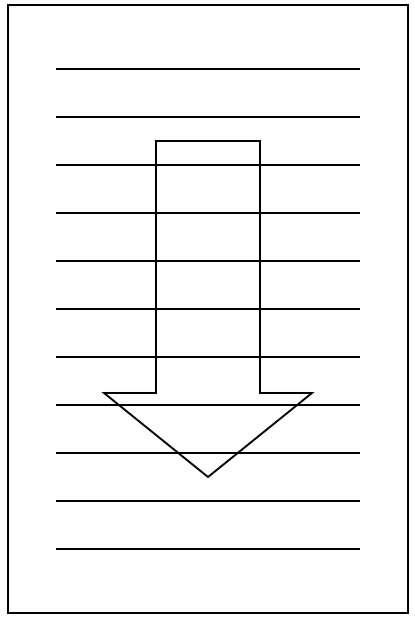
- Big Design Issue: How do we make the OS efficient, reliable, and extensible?

- General OS Philosophy: The design and implementation of an OS involves a constant tradeoff between simplicity and performance.

- As a general rule, strive for simplicity.
  - except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance
  - (strong reason = simulation or evaluation study)

Main Abstraction: The Process

- Abstraction of a running program
  - "a program in execution"
- Dynamic
  - Has state, changes over time
  - Whereas a program is static
- Basic operations
  - Start/end
  - Suspend/resume



Basic Resources for Processes

- To run, process needs some basic resources:
  - CPU
    - Processing cycles (time)
    - To execute instructions
  - Memory
    - Bytes or words (space)
    - To maintain state
  - Other resources (e.g., I/O)
    - Network, disk, terminal, printer, etc.

What sort of information might the OS need to store to keep track of a running process?

- That is, what MUST an OS know about a process?
- (Discuss with your neighbors.)

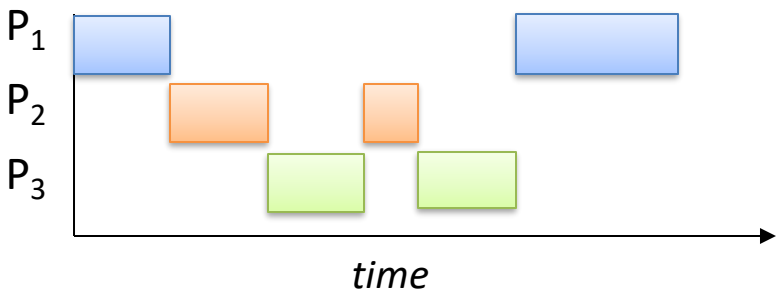
Machine State of a Process

- CPU or processor context
  - PC (program counter)
  - SP (stack pointer)
  - General purpose registers
- Memory
  - Code
  - Global Variables
  - Stack of activation records / frames
  - Other (registers, memory, kernel-related state)

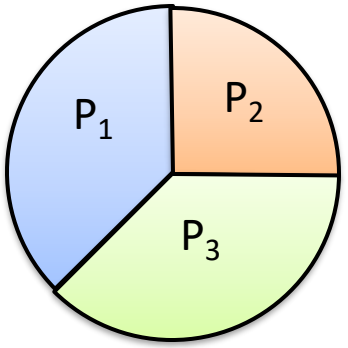
Must keep track of these for every running process !

Resource Sharing

CPU: Time



Memory: Space



Reality

- Multiple processes
- Small number of CPUs
- Finite memory

Abstraction

- Process is all alone
- Process is always running
- Process has all the memory

Resource: CPU

- Many processes, limited number of CPUs.
- Each process needs to make progress over time. Insight: processes don't know how quickly they *should* be making progress.
- Illusion: every process is making progress in parallel.

Timesharing: Sharing the CPUs

- Abstraction goal: make every process *think* it's running on the CPU all the time.
  - Alternatively: If a process was removed from the CPU and then given it back, it shouldn't be able to tell

- Reality: put a process on CPU, let it run for a short time (~10 ms), switch to another, … ("context switching")

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
  - Running: actually making progress, using CPU
  - Ready: able to make progress, but not using CPU
  - Blocked: not able to make progress, can't use CPU
- Kernel selects a ready process, lets it run
  - Eventually, the kernel gets back control
  - Selects another ready process to run, …

Multiprogramming

- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - If resource is busy (or slow), process can't proceed
  - "Voluntarily" gives up CPU to another process

- Mechanism: Context switching

Time Sharing / Multiprogramming

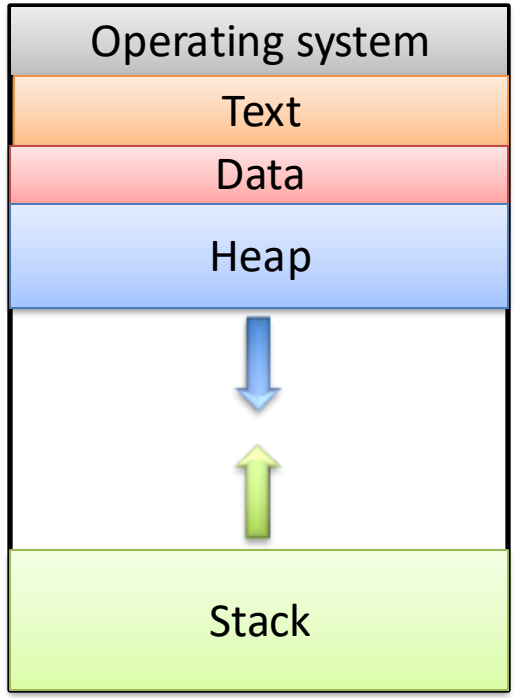
- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - If resource is busy (or slow), process can't proceed
  - "Voluntarily" gives up CPU to another process

- Mechanism: Context switching
- Policy: CPU scheduling

Resource: Memory

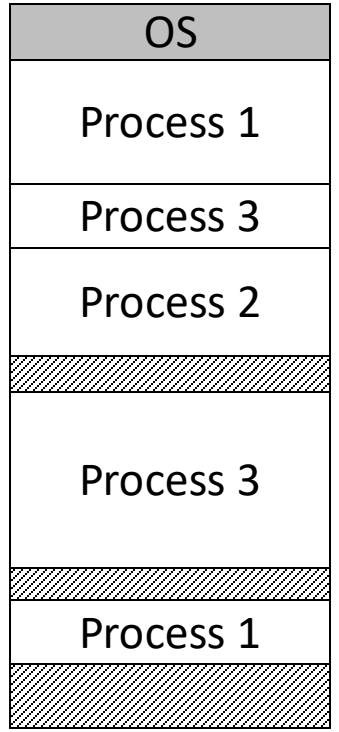
Abstraction goal: make every process think it has the same memory layout.

- MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.



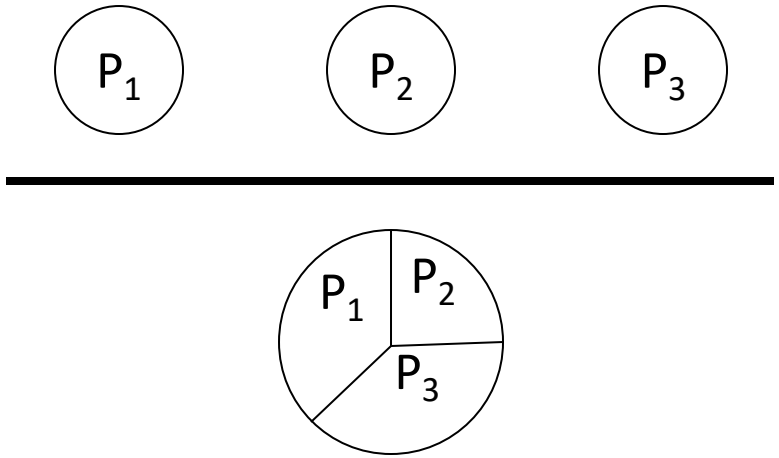
Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.
- Reality: there's only so much memory to go around
  - no two processes should use the same (physical) memory addresses (unless they're sharing).



OS (with help from hardware) will keep track of who's using each memory region.

Virtual Memory: Sharing Storage



- Like CPU cache, memory is a cache for disk.

- Processes never need to know where their memory truly is, OS translates virtual addresses into physical addresses for them.

Kernel Execution

- Great, the OS is going to somehow give us these nice abstractions.

- So…how / when should the kernel execute to make all this stuff happen?

The operating system kernel…

A. Executes as a process.

B. Is always executing, in support of other processes.

C. Should execute as little as possible.

D. More than one of the above. (Which ones?)

E. None of the above.

Process vs. Kernel

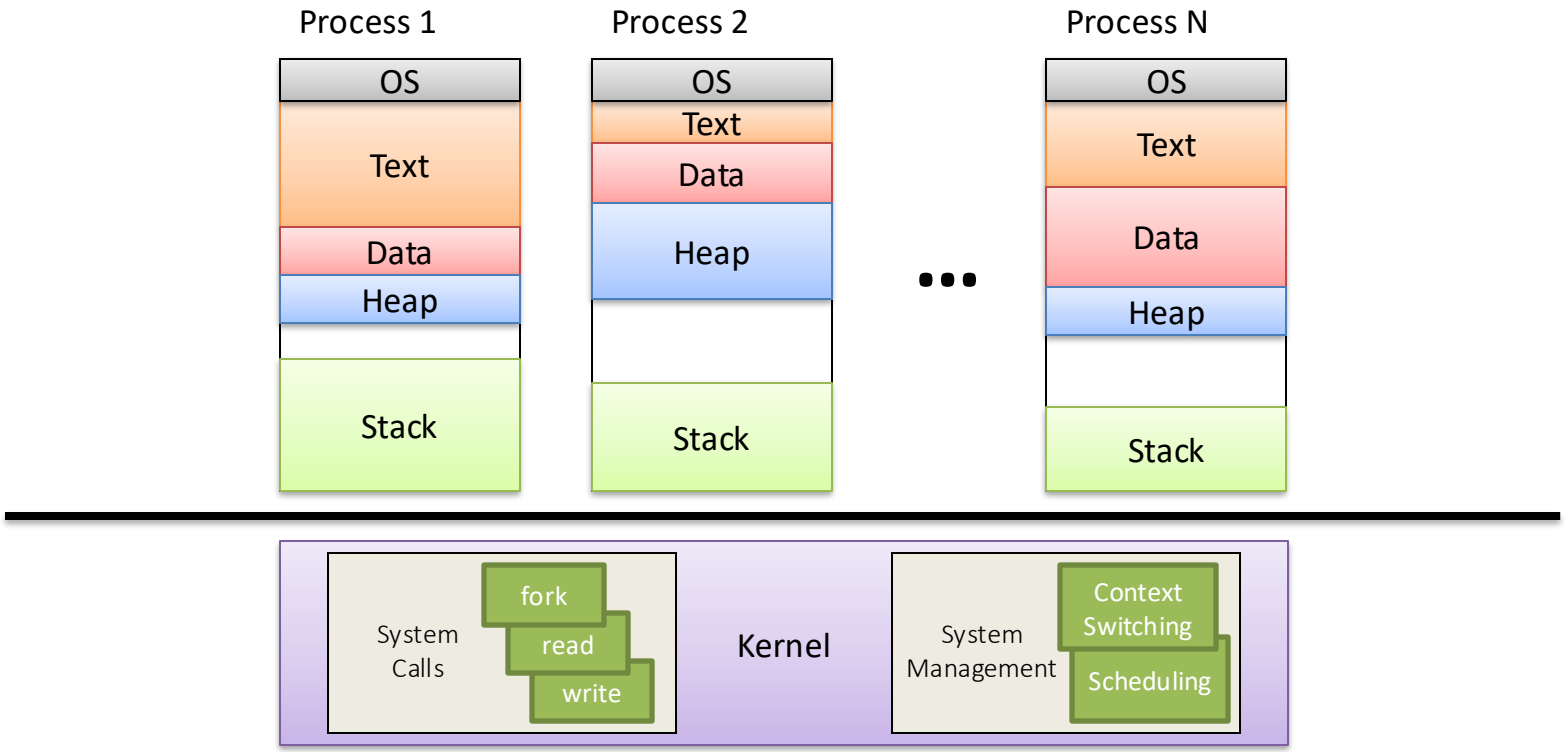
- Is the kernel itself a process?
  - No, it supports processes and devices

- OS only runs when necessary…
  - as an extension of a process making system call
  - in response to a device issuing an interrupt

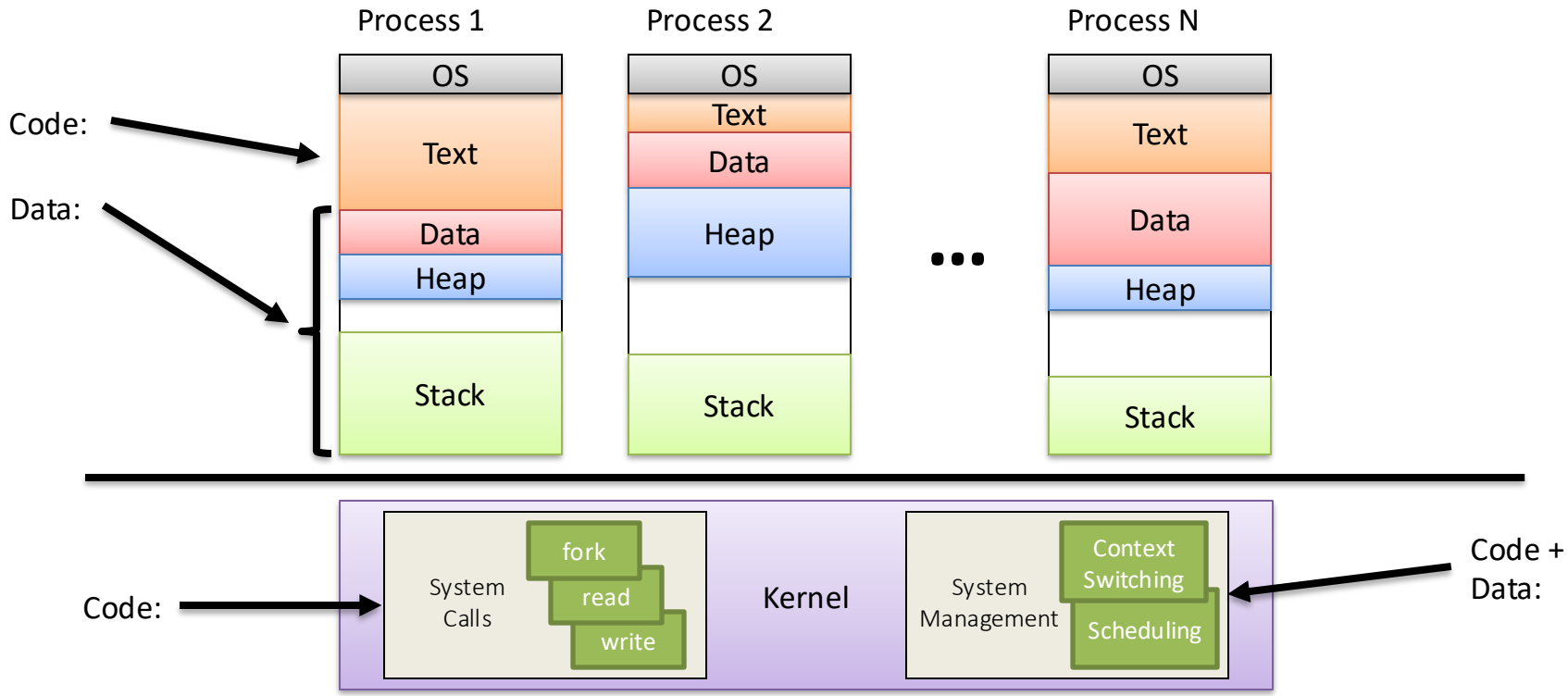
Process vs. Kernel

- The kernel is the code that supports processes
  - System calls: fork ( ), exit ( ), read ( ), write ( ), …
  - System management: context switching, scheduling, memory management

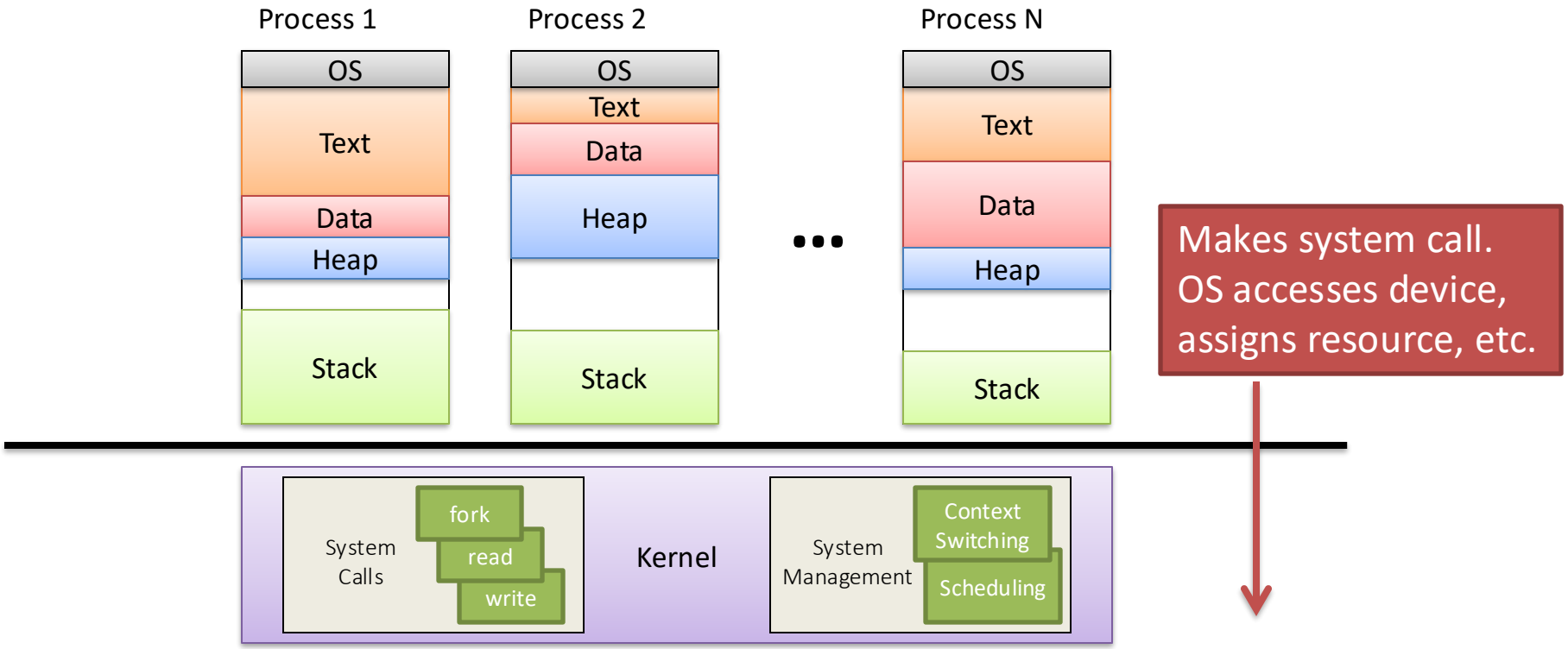
Kernel vs. Userspace: Model



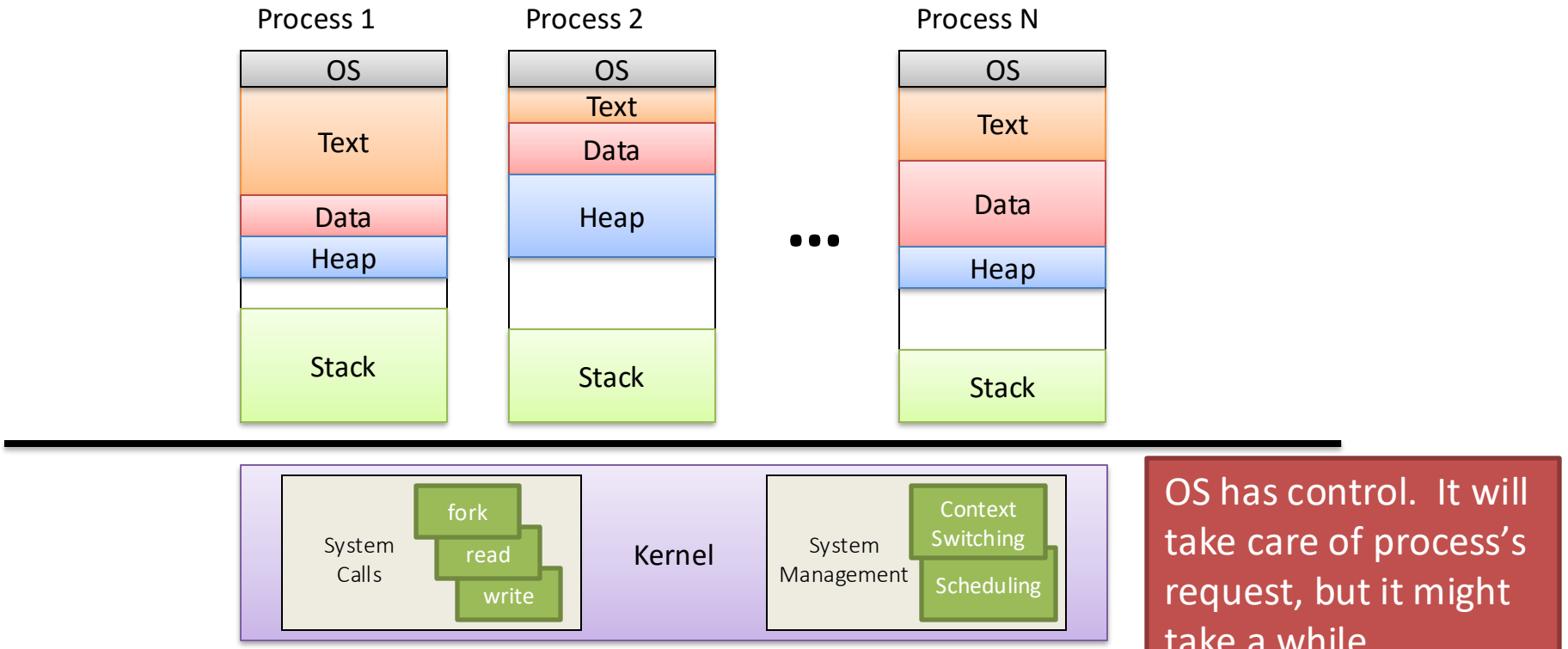
Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



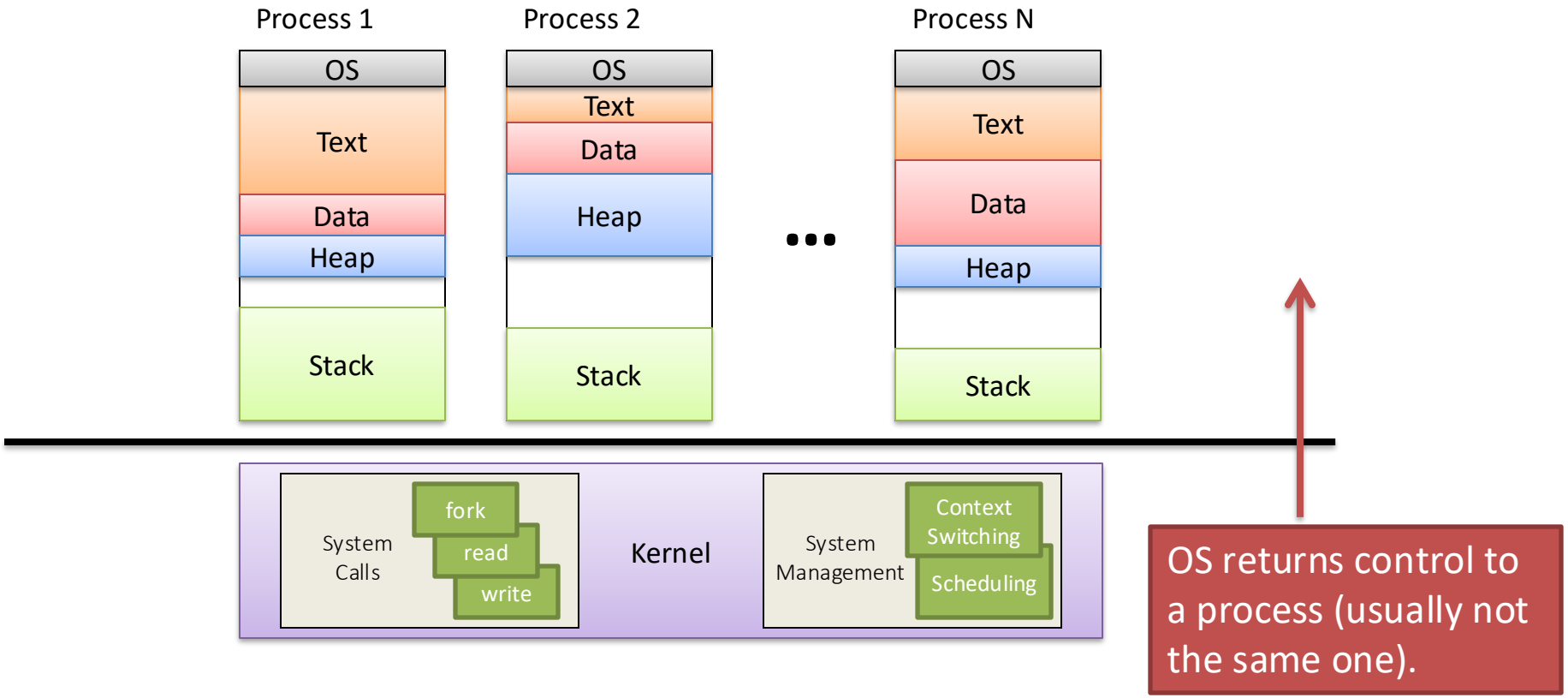
Kernel vs. Userspace: Model



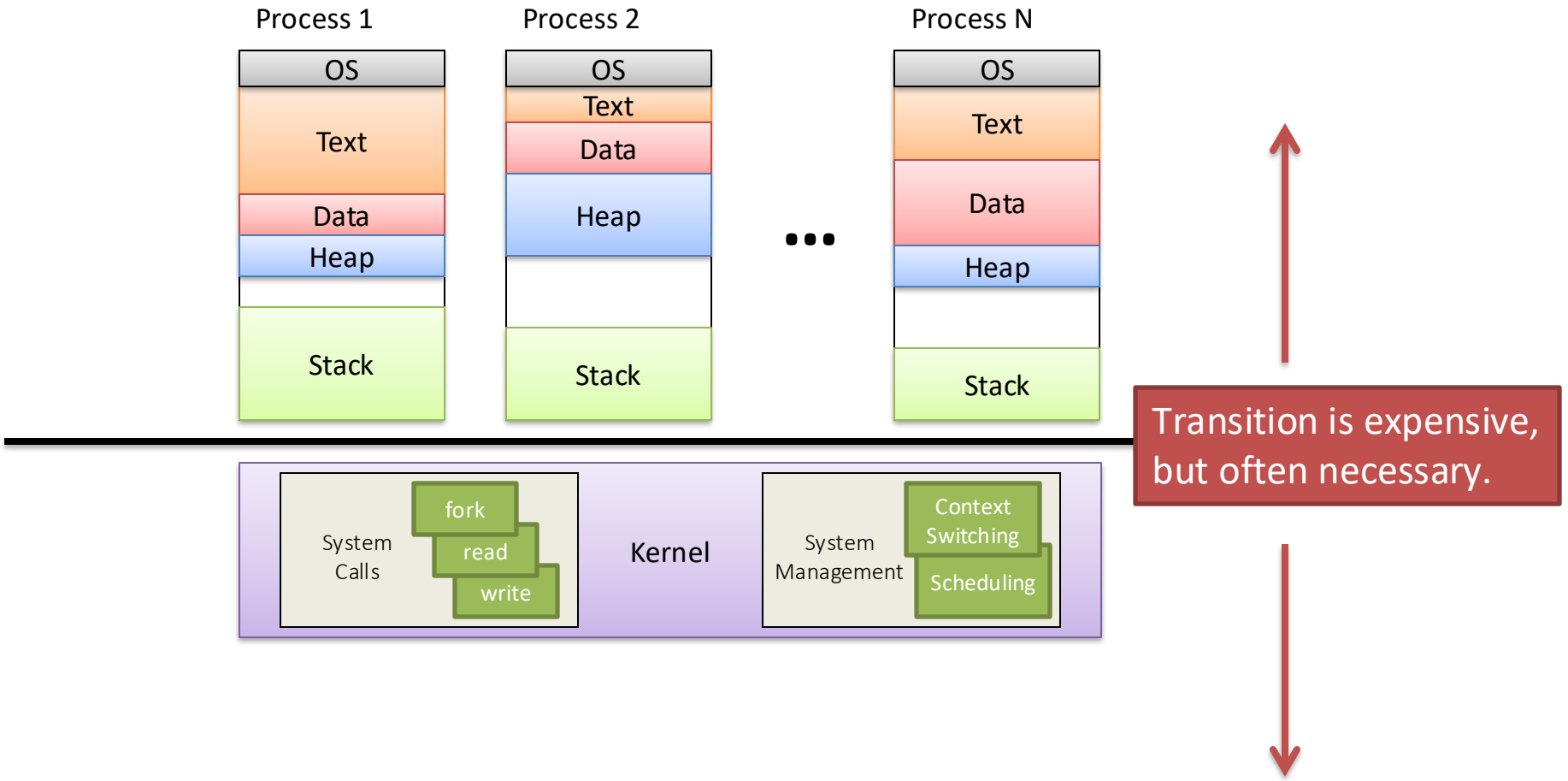
OS has control. It will take care of process's request, but it might take a while.

It can context switch (and usually does at this point).

Kernel vs. Userspace: Model

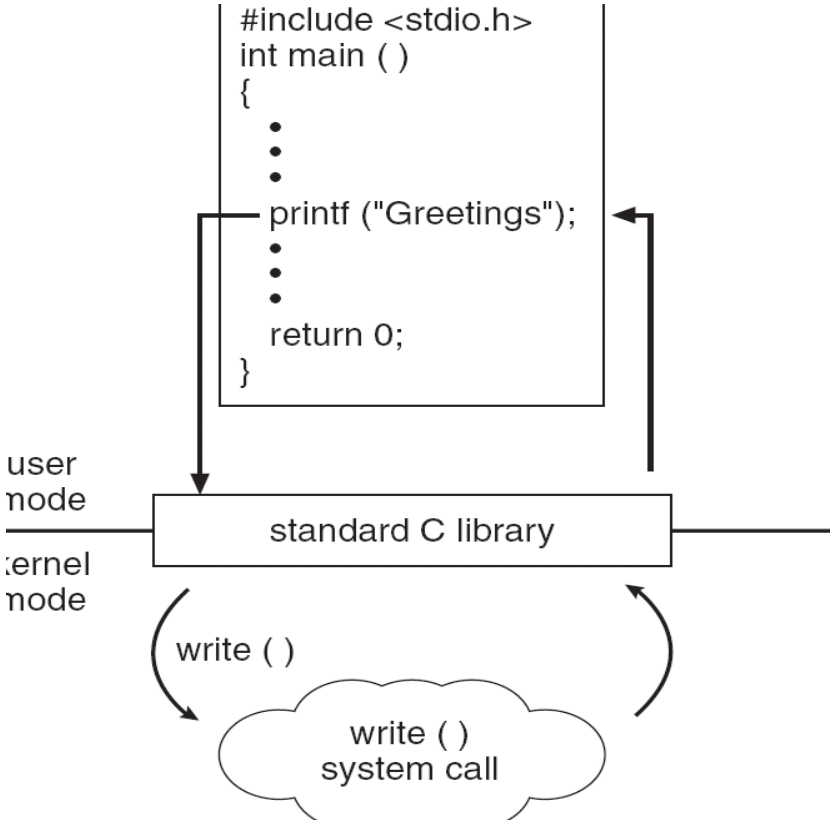


Kernel vs. Userspace: Model



Standard C Library Example

C program invoking printf() library call, which calls write() system call



Control over the CPU

- To context switch processes, kernel must get control:

1. Running process can give up control voluntarily
   - To block, call yield () to give up CPU
   - Process makes a blocking system call, e.g., read ()
   - Control goes to kernel, which dispatches new process

2. CPU is forcibly taken away: preemption

How might the OS forcibly take control of a CPU?

A. Ask the user to give it the CPU.

B. Require a program to make a system call.

C. Enlist the help of a hardware device.

D. Some other means of seizing control (how?).

CPU Preemption

1. While kernel is running, set a hardware timer.

2. When timer expires, a hardware interrupt is generated. (device asking for attention)

3. Interrupt pauses process on CPU, forces control to go to OS kernel.

4. OS is free to perform a context switch.