CS 344: Operating Systems I

01.18: Part I - Process

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

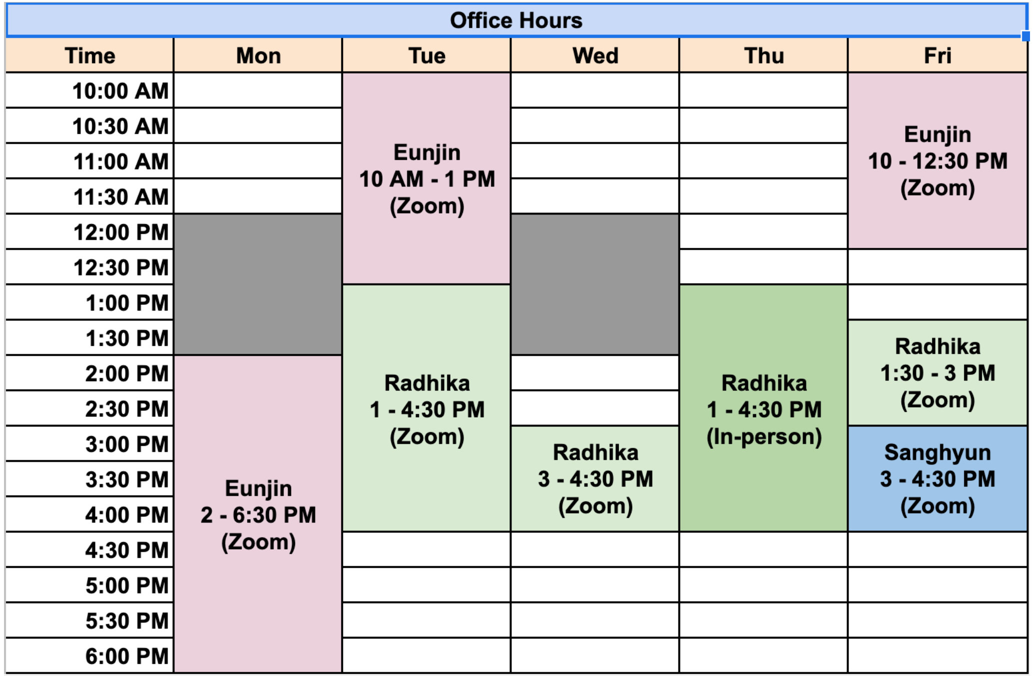
sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Notice

- Announcements
  - Begin office hours
    - Time and locations: available on Canvas
    - Other times: at Discord server



Office Hours					
Time	Mon	Tue	Wed	Thu	Fri
10:00 AM		Eunjin 10 AM - 1 PM (Zoom)			Eunjin 10 - 12:30 PM (Zoom)
10:30 AM					
11:00 AM					
11:30 AM					
12:00 PM					
12:30 PM					
1:00 PM		Radhika 1 - 4:30 PM (Zoom)		Radhika 1 - 4:30 PM (In-person)	
1:30 PM					Radhika 1:30 - 3 PM (Zoom)
2:00 PM	Eunjin 2 - 6:30 PM (Zoom)				
2:30 PM					
3:00 PM			Radhika 3 - 4:30 PM (Zoom)		Sanghyun 3 - 4:30 PM (Zoom)
3:30 PM					
4:00 PM					
4:30 PM					
5:00 PM					
5:30 PM					
6:00 PM					

Notice – cont'd

- Announcements
  - Begin office hours
    - Time and locations: available on Canvas
    - Other times: at Discord server
  - Notes
    - Discord: allow us a few hours to answer questions (2 TAs for 135+ students)
    - Discord: post questions to corresponding channels (e.g., #assignment-1 for the assignment 1)
    - Discord: feel free to DM instructor or TAs (Sanghyun, Radhika, or Eunjin)
    - All: help others, when you already know answers
      (*do not share your code with others)

Notice – cont'd

- Deadlines
  - (Passed) Syllabus quiz
  - (1/23 11:59 PM) Programming assignment 1
  - (1/30 11:59 PM) Midterm quiz 1

Topics for today

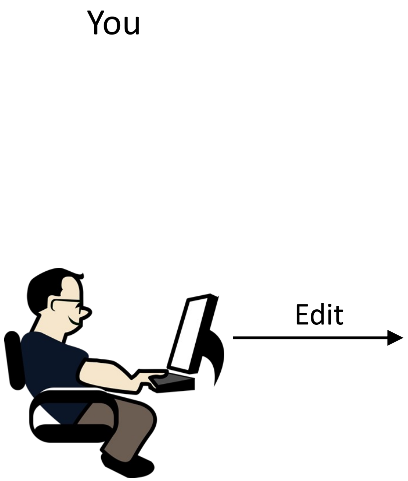
- Part I: Process
  - Provide abstraction
    - What is a program?
    - What is a process?
    - How does OS run a program?
  - Offer standard libraries
    - How do we run (or stop) a process?
    - How does OS manage the process(es) we ran?
  - Manage resources
    - (Note) We will talk about this in the “scheduling” class

Provide abstraction: a program

- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute
  - **An example program for Linux computer**

Provide abstraction: a program

- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute
  - **An example program for Linux computer**

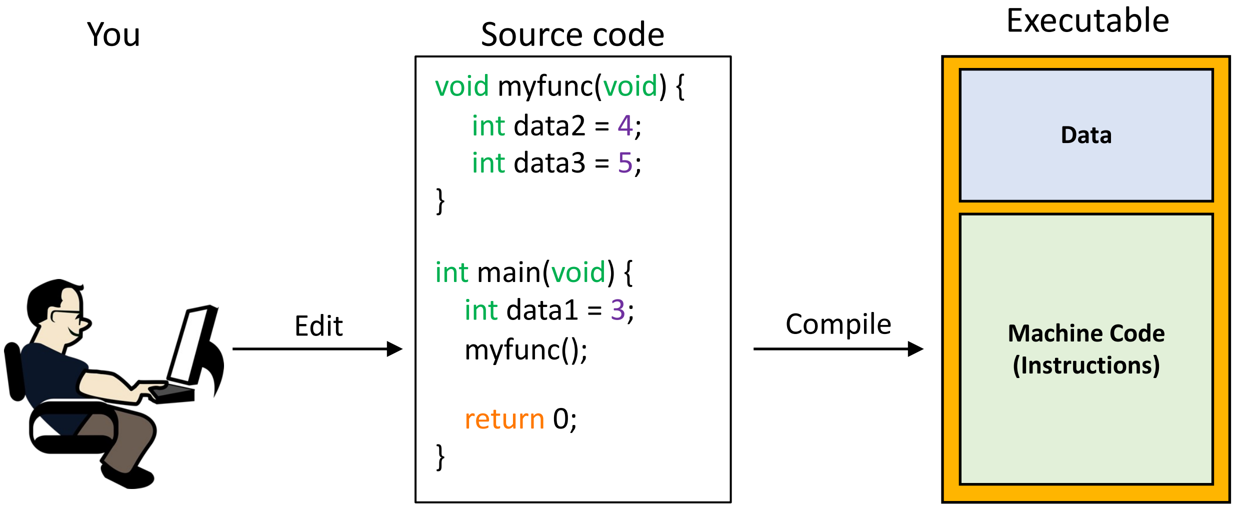


Source code

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

Provide abstraction: a program

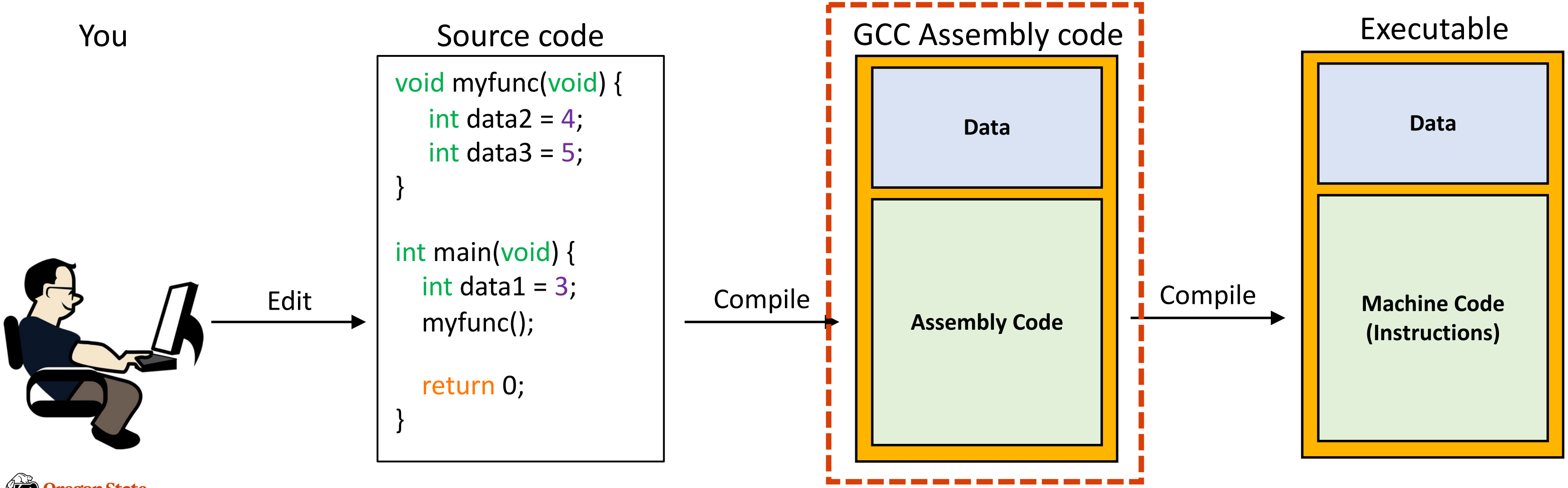
- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute
  - **An example program for Linux computer**



```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

Example: C compilation with GCC

- GCC compilation
  - It converts source code to **assembly** code ($ gcc -c -S <filename.c>)
  - It then converts the assembly code to **instructions** ($ gcc -c <filename.s> -o <filename.o>; gcc -o <filename.o> -o filename)



You

Edit

Source code

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

Compile

GCC Assembly code

Data

Assembly Code

Compile

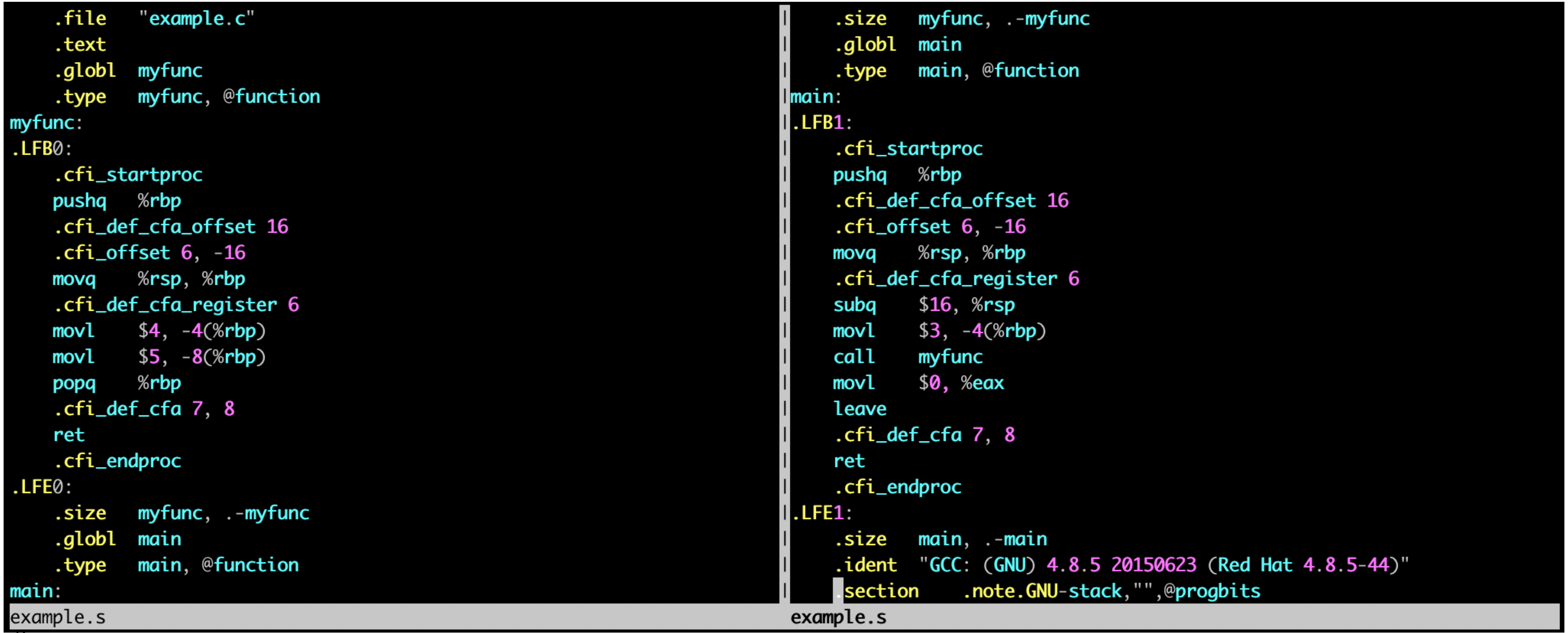
Executable

Data

Machine Code (Instructions)

Example: C compilation with GCC

- GCC compilation
  - It converts source code to **assembly** code ($ gcc -c -S <filename.c>)

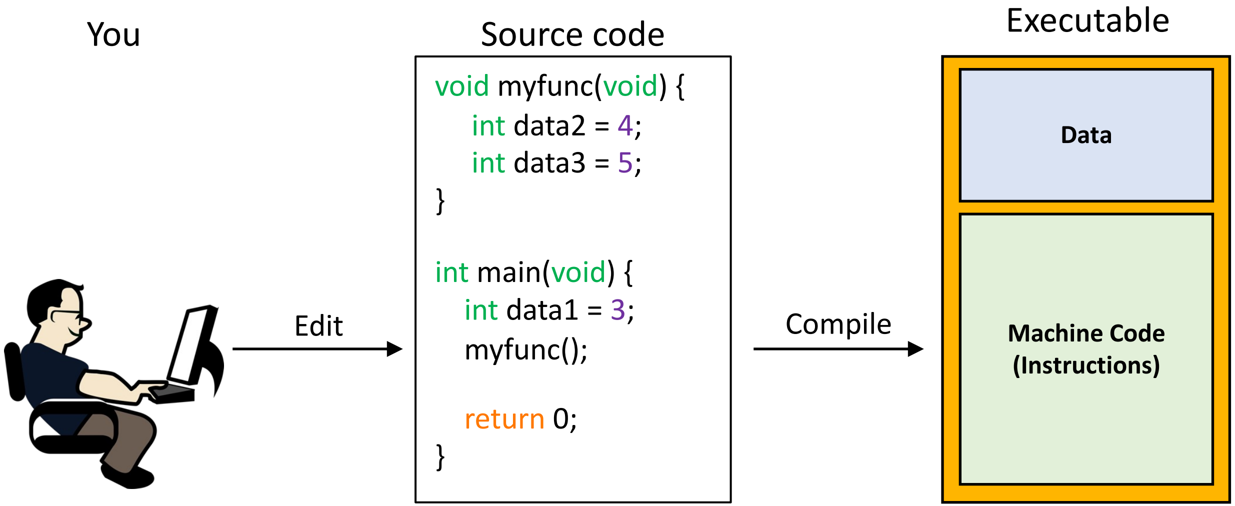


```
	.file	"example.c"
	.text
	.globl	myfunc
	.type	myfunc, @function
myfunc:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	$4, -4(%rbp)
	movl	$5, -8(%rbp)
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	myfunc, .-myfunc
	.globl	main
	.type	main, @function
main:
example.s
```

```
	.size	myfunc, .-myfunc
	.globl	main
	.type	main, @function
main:
.LFB1:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	$3, -4(%rbp)
	call	myfunc
	movl	$0, %eax
	leave
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE1:
	.size	main, .-main
	.ident	"GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
	.section	.note.GNU-stack,"",@progbits
example.s
```

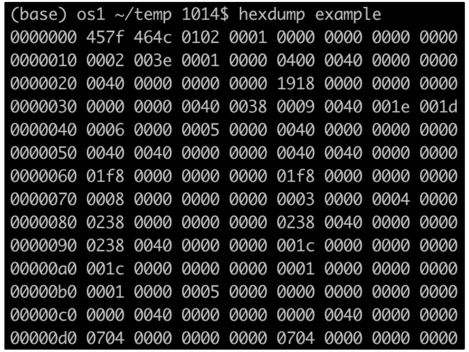
Provide abstraction: a program

- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute
  - **An example program for Linux computer**



```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
    int data1 = 3;
    myfunc();

    return 0;
}
```

ex. $ hexdump subvert.c

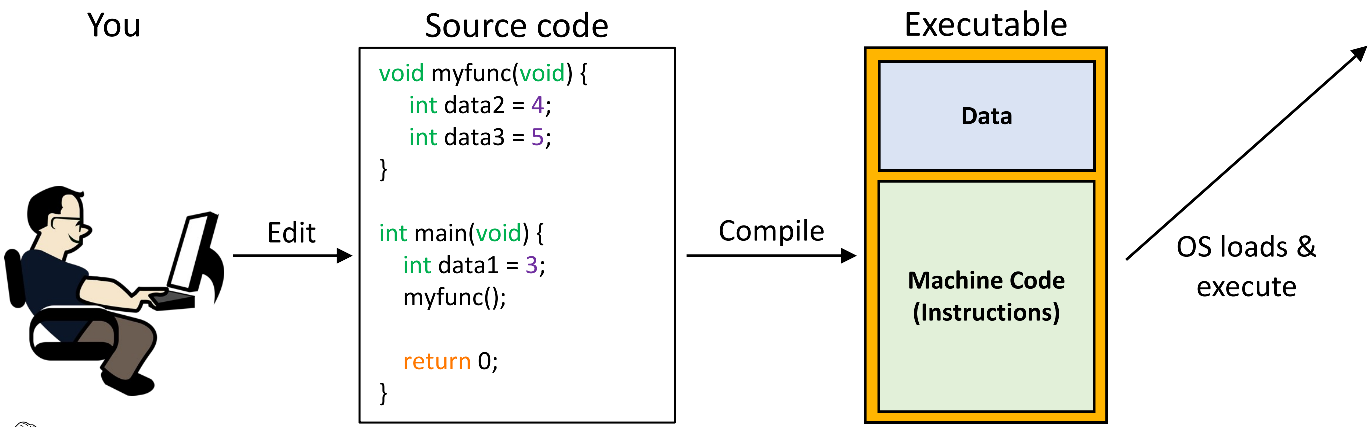


```
(base) os1 ~/temp 1014$ hexdump example
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0002 003e 0001 0000 0400 0040 0000 0000
0000020 0040 0000 0000 0000 1918 0000 0000 0000
0000030 0000 0000 0040 0038 0009 0040 001e 001d
0000040 0006 0000 0005 0000 0040 0000 0000 0000
0000050 0040 0040 0000 0000 0040 0040 0000 0000
0000060 01f8 0000 0000 0000 01f8 0000 0000 0000
0000070 0008 0000 0000 0000 0003 0000 0004 0000
0000080 0238 0000 0000 0000 0238 0040 0000 0000
0000090 0238 0040 0000 0000 001c 0000 0000 0000
00000a0 001c 0000 0000 0000 0001 0000 0000 0000
00000b0 0001 0000 0005 0000 0000 0000 0000 0000
00000c0 0000 0040 0000 0000 0000 0040 0000 0000
00000d0 0704 0000 0000 0000 0704 0000 0000 0000
```

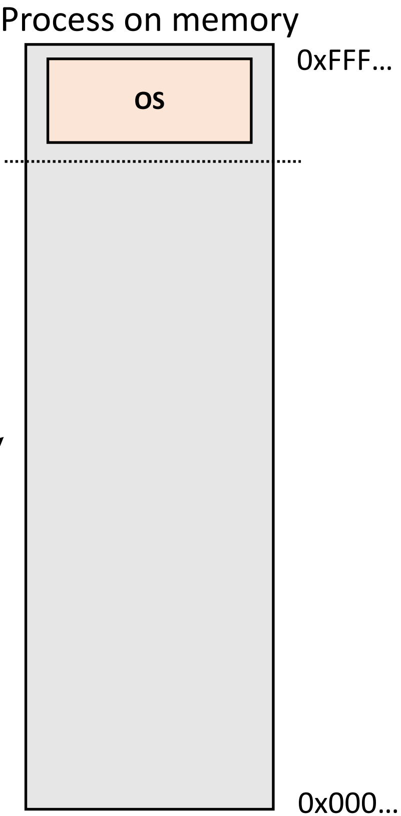
…

Provide abstraction: a process

- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Process segments:**
    - **Code**, **data**, **heap** and **stack**

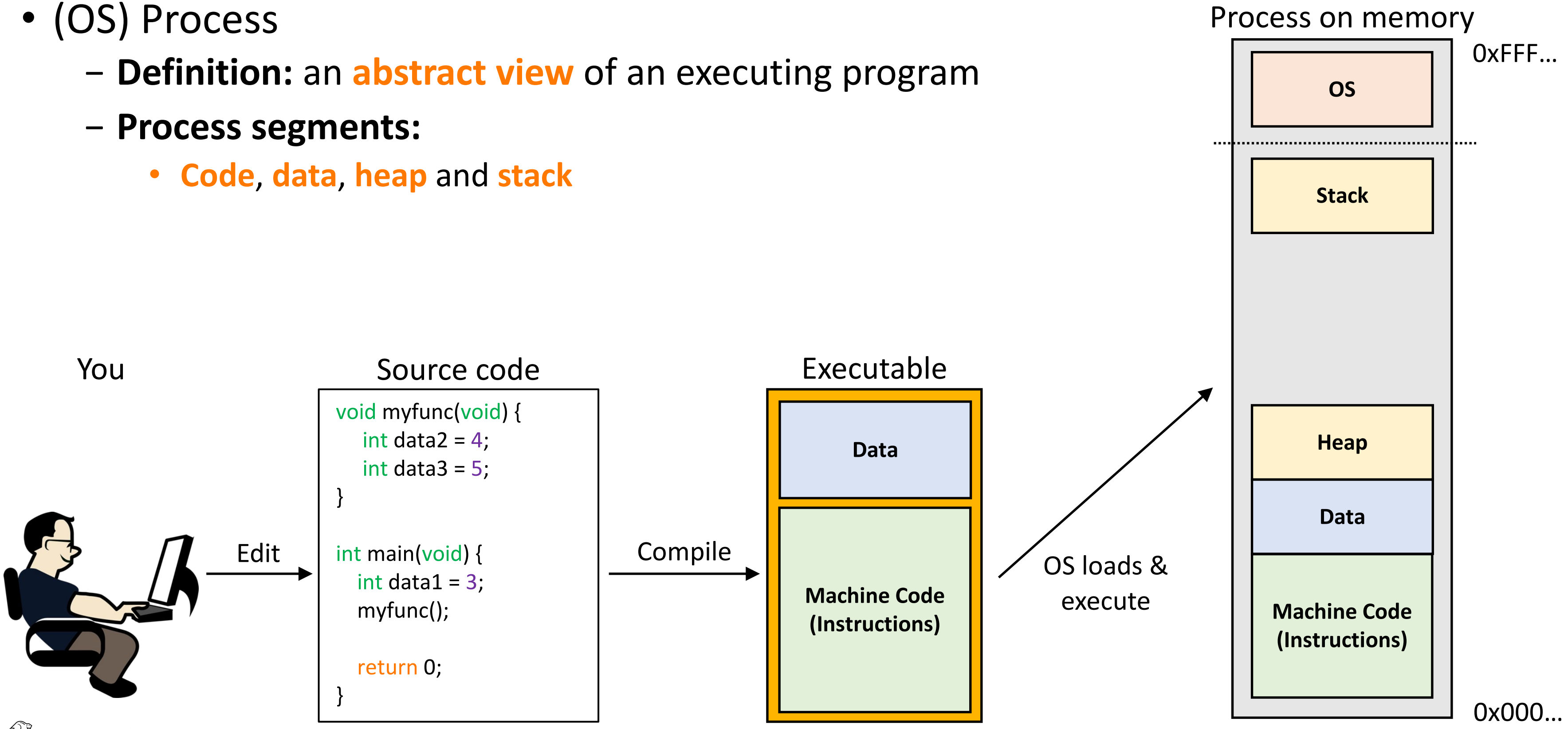


```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
    int data1 = 3;
    myfunc();

    return 0;
}
```



Provide abstraction: a process

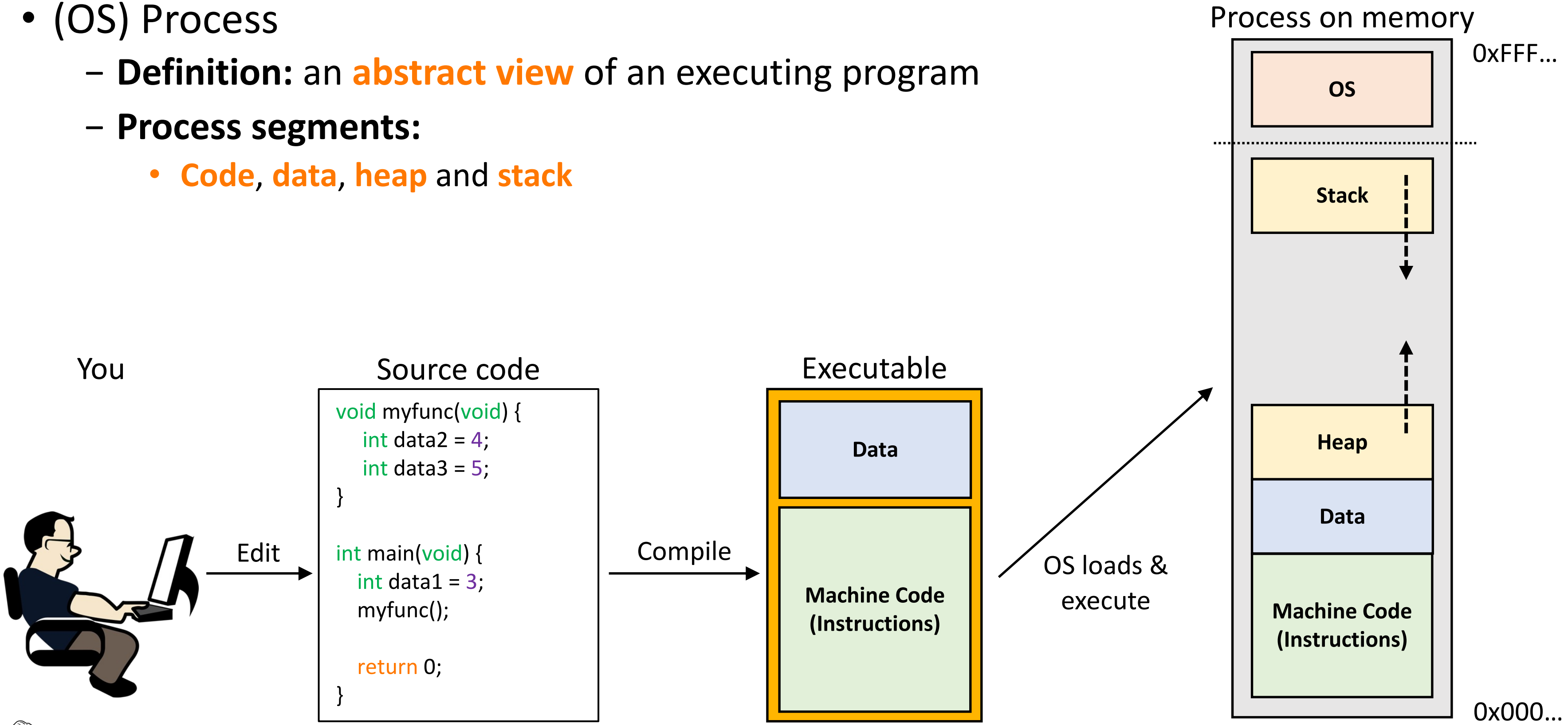
- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Process segments:**
    - **Code**, **data**, **heap** and **stack**



```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
    int data1 = 3;
    myfunc();

    return 0;
}
```

Provide abstraction: a process

- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Process segments:**
    - **Code**, **data**, **heap** and **stack**

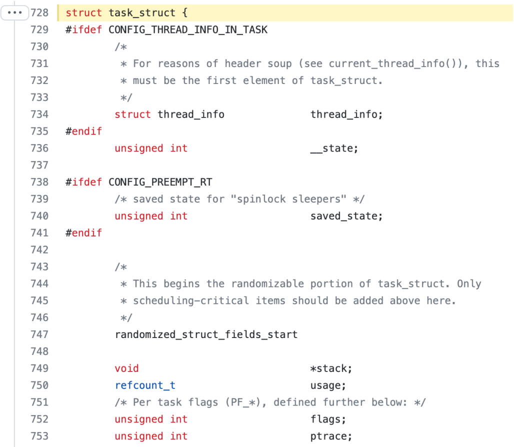


```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
    int data1 = 3;
    myfunc();

    return 0;
}
```

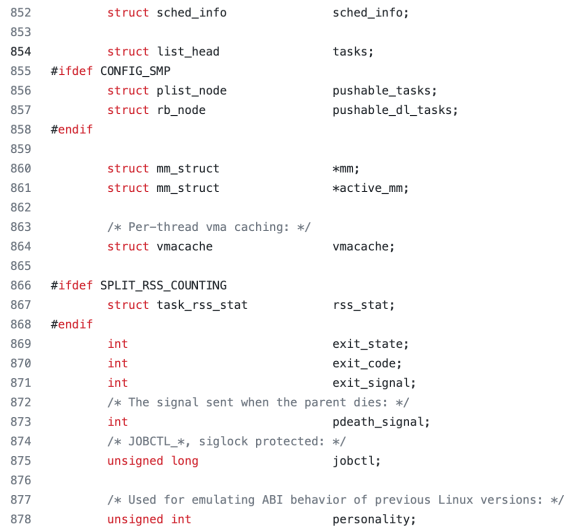
Provide abstraction: how OS defines a process?

- (Linux) has the process context
  - Code
    - Program counter
    - Instruction pointer
  - Stack and heap
    - Stack pointer
    - Heap pointer
  - Running context
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - (OS II) Scheduling policy
    - (OS II) Mem. virtualization
  - …

Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)



```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        unsigned int                    __state;

#ifdef CONFIG_PREEMPT_RT
        /* saved state for "spinlock sleepers" */
        unsigned int                    saved_state;
#endif

        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
        refcount_t                      usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
        unsigned int                    ptrace;
```

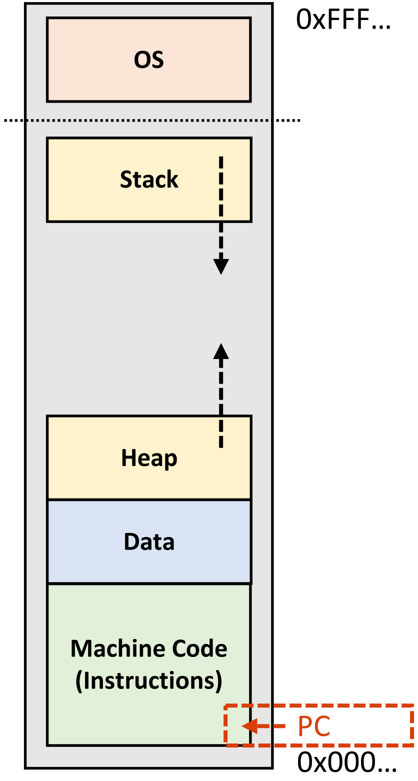


```
        struct sched_info               sched_info;

        struct list_head                tasks;
#ifdef CONFIG_SMP
        struct plist_node               pushable_tasks;
        struct rb_node                  pushable_dl_tasks;
#endif

        struct mm_struct                *mm;
        struct mm_struct                *active_mm;

        /* Per-thread vma caching: */
        struct vmacache                 vmacache;

#ifdef SPLIT_RSS_COUNTING
        struct task_rss_stat            rss_stat;
#endif
        int                             exit_state;
        int                             exit_code;
        int                             exit_signal;
        /* The signal sent when the parent dies: */
        int                             pdeath_signal;
        /* JOBCTL_*, siglock protected: */
        unsigned long                   jobctl;

        /* Used for emulating ABI behavior of previous Linux versions: */
        unsigned int                    personality;
```

Provide abstraction: how OS loads a process?

- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Load a process:**
    - **Code**: OS loads the instructions to "code" segments
    - **Data** : OS loads the data (such as static vars) to "data" segments
    - **Stack** and **heap**: OS creates those mem. spaces
    - (Ready) OS sets the program counter (PC) to the first code location

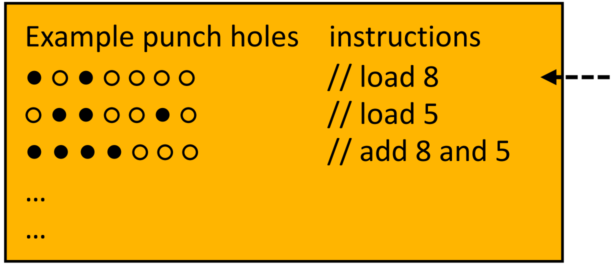
Process on memory



Provide abstraction: how OS runs a process?

- OS makes the CPU run the machine code
  - Example: IBM machines
    - Submit **a punch card** that have a set of **instructions**
    - Machine **reads** instructions line by line and **do sth.**

Punch card



Example punch holes	instructions
● ○ ● ○ ○ ○ ○	// load 8
○ ● ● ○ ○ ● ○	// load 5
● ● ● ● ○ ○ ○	// add 8 and 5
…	
…	

Provide abstraction: how OS runs a process? – cont'd

- OS makes the CPU run the machine code
  - Example: IBM machines
    - Submit **a punch card** that have a set of **instructions**
    - Machine **reads** instructions line by line and **do sth.**
  - Modern computers
    - Machine := a processor (CPU)
    - Instructions := instructions (100+ for Intel CPUs)
    - Punch card := a process in memory
    - Operates := execute the instructions

Punch card

Example punch holes	instructions
● ○ ● ○ ○ ○ ○	// load 8
○ ● ● ○ ○ ● ○	// load 5
● ● ● ● ○ ○ ○	// add 8 and 5
…	
…	

Memory

Example instructions	operations
0x11 0x12 0x05 0x00	// load 5 to r12
0x08 0x12 0x08 0x00	// add r12 and 8
0x12 0xF9 0xFF 0xF4	// store r12
…	
…	
…	

Provide abstraction: how OS runs a process? – cont'd

- OS makes the CPU run the machine code
  - Example: IBM machines
    - Submit **a punch card** that have a set of **instructions**
    - Machine **reads** instructions line by line and **do sth.**
  - Modern computers
    - Machine := a processor (CPU)
    - Instructions := instructions (100+ for Intel CPUs)
    - Punch card := a process in memory
    - Operates := execute the instructions

The program counter (PC) in a CPU is always holding the memory address where the next instruction to execute is

Punch card

Example punch holes	instructions
● ○ ● ○ ○ ○ ○	// load 8 ◄---
○ ● ● ○ ○ ● ○	// load 5
● ● ● ● ○ ○ ○	// add 8 and 5
…	
…	

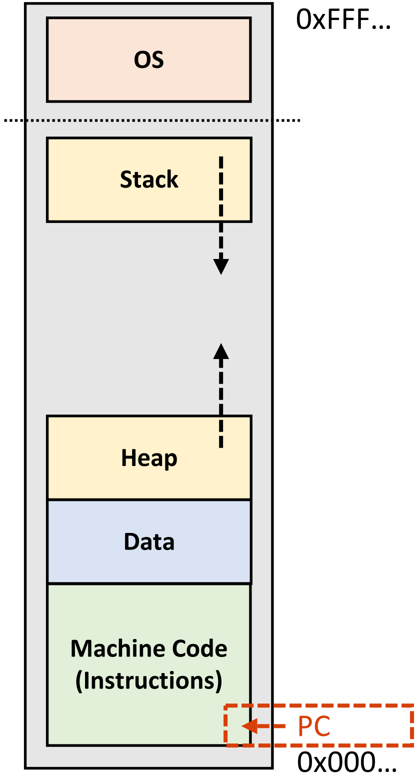
Memory

Example instructions	operations
0x11 0x12 0x05 0x00	// load 5 to r12 ◄---
0x08 0x12 0x08 0x00	// add r12 and 8
0x12 0xF9 0xFF 0xF4	// store r12
…	
…	
…	

Provide abstraction: how OS loads/runs a process?

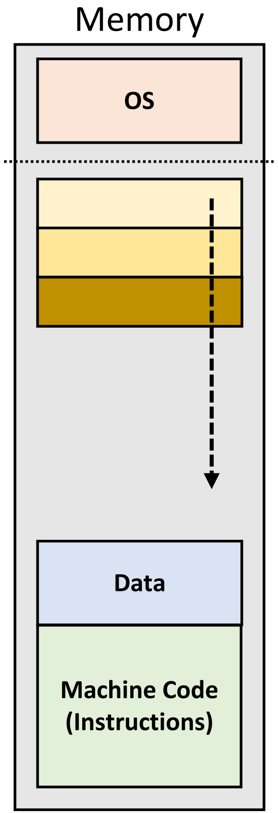
- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Load a process:**
    - **Code**: OS loads the instructions to “code” segments
    - **Data** : OS loads the data (such as static vars) to “data” segments
    - **Stack** and **heap**: OS creates those mem. spaces
    - **(Ready) OS sets the program counter (PC) to the first code location**

Process on memory



Provide abstraction: stack vs. heap

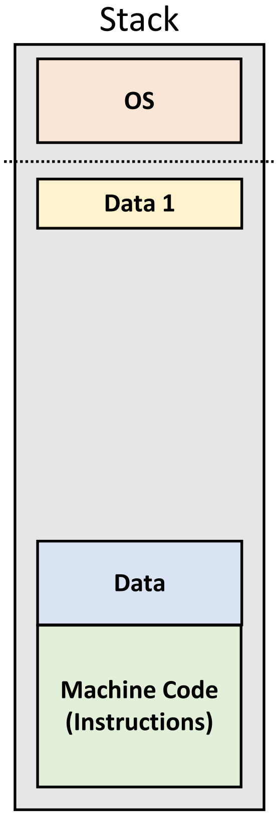
- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Stack
    - **OS controls** the memory allocations (size)
    - Store data in Last in first out (**LIFO**) manner
    - Stack mostly holds data initialized within a function



Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Stack
    - **OS controls** the memory allocations (size)
    - Store data in Last in first out (**LIFO**) manner
    - Stack mostly holds data initialized within a function

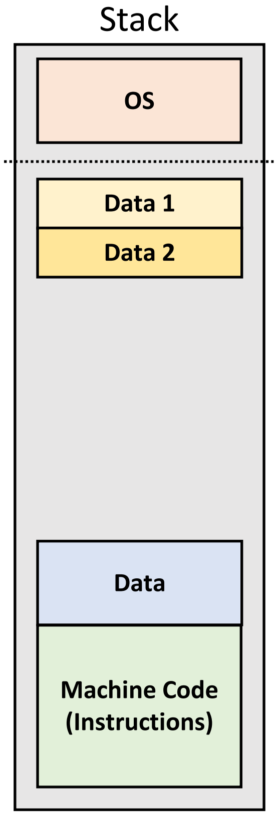
```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```



Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Stack
    - **OS controls** the memory allocations (size)
    - Store data in Last in first out (**LIFO**) manner
    - Stack mostly holds data initialized within a function

```
void myfunc(void) {
    int data2 = 4;          <----  Run
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

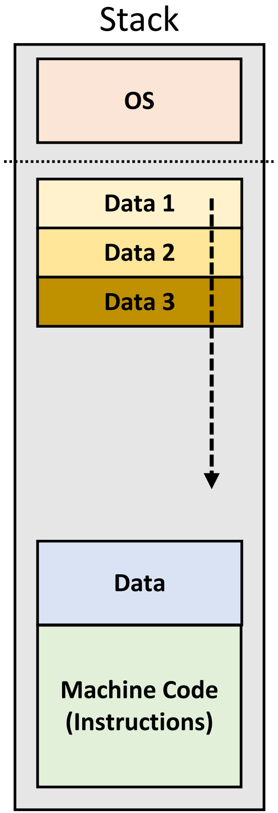


Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Stack
    - **OS controls** the memory allocations (size)
    - Store data in Last in first out (**LIFO**) manner
    - Stack mostly holds data initialized within a function

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

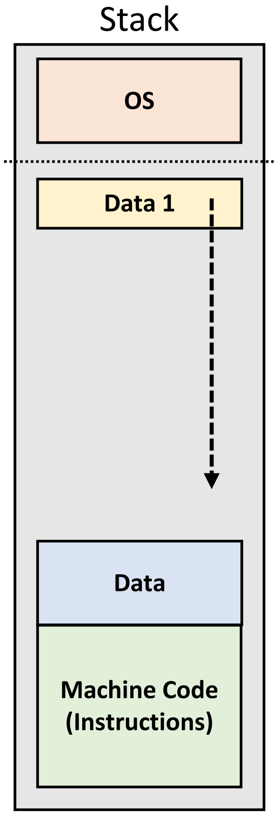
◄---- **Run**



Provide abstraction: stack vs. heap

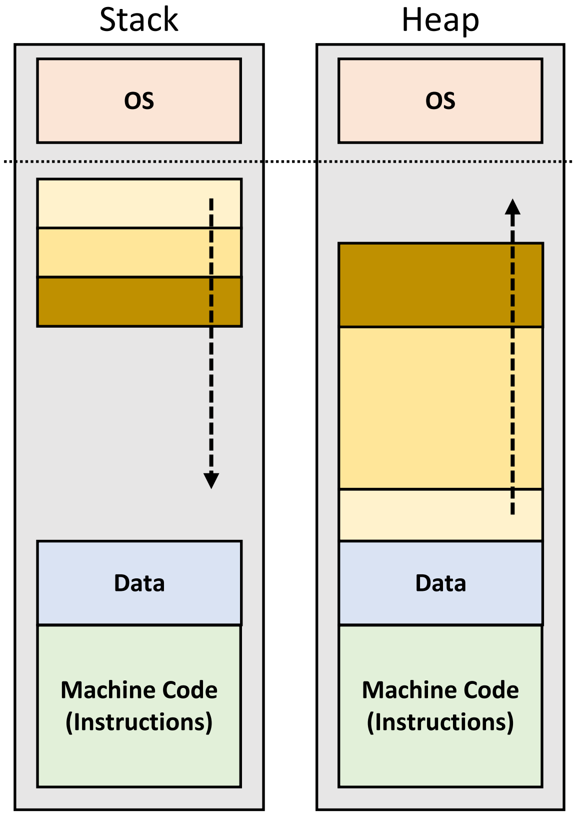
- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Stack
    - **OS controls** the memory allocations (size)
    - Store data in Last in first out (**LIFO**) manner
    - Stack mostly holds data initialized within a function

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;          <---- Run
}
```



Provide abstraction: stack vs. heap

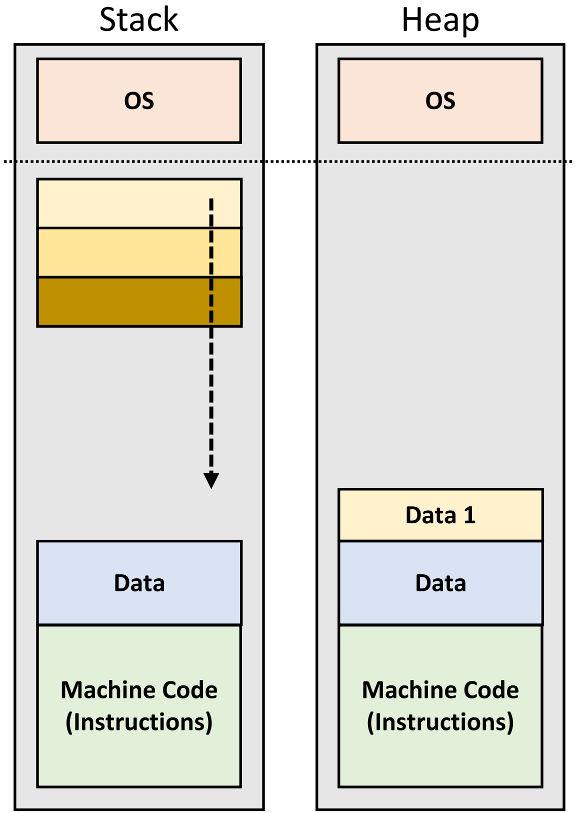
- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur



Provide abstraction: Stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur

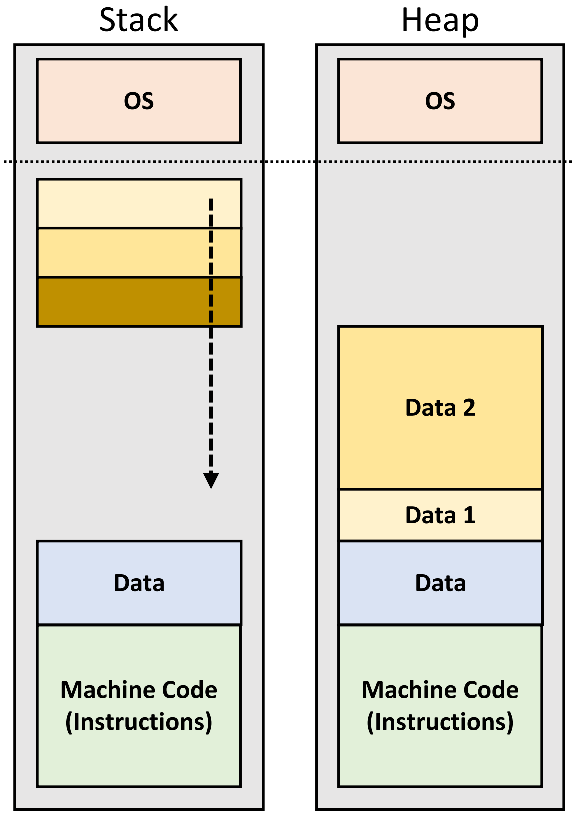
```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);
   free(data2);
}

int main(void) {
   char *data1 = (char *) malloc(1);        <---- Run
   myfunc();

   return 0;
}
```



Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur

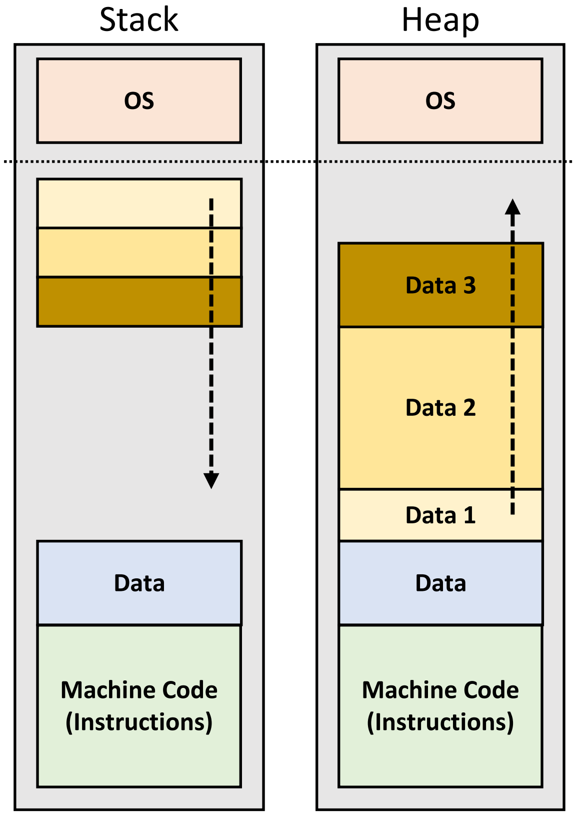
```
void myfunc(void) {
    char *data2 = (char *) malloc(5);        <---- Run
    char *data3 = (char *) malloc(2);
    free(data2);
}

int main(void) {
    char *data1 = (char *) malloc(1);
    myfunc();

    return 0;
}
```



Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur

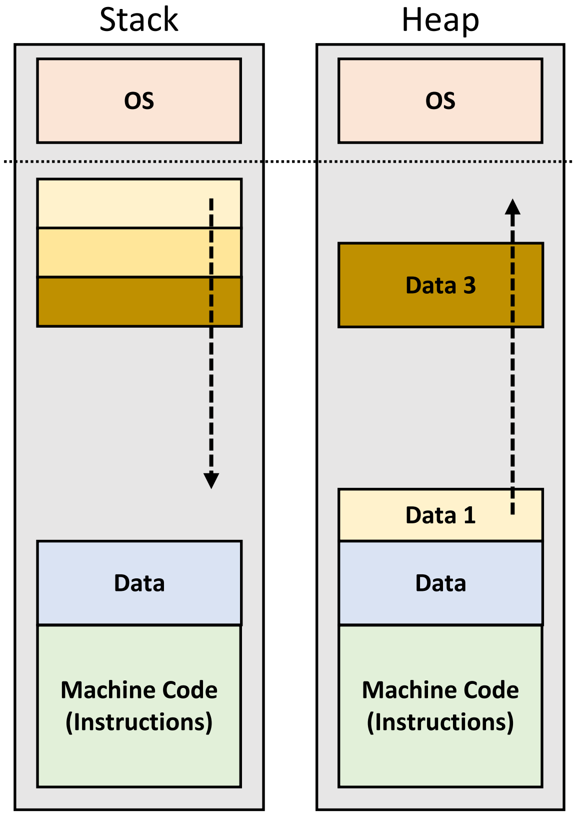
```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);        <----  Run
   free(data2);
}

int main(void) {
   char *data1 = (char *) malloc(1);
   myfunc();

   return 0;
}
```



Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur

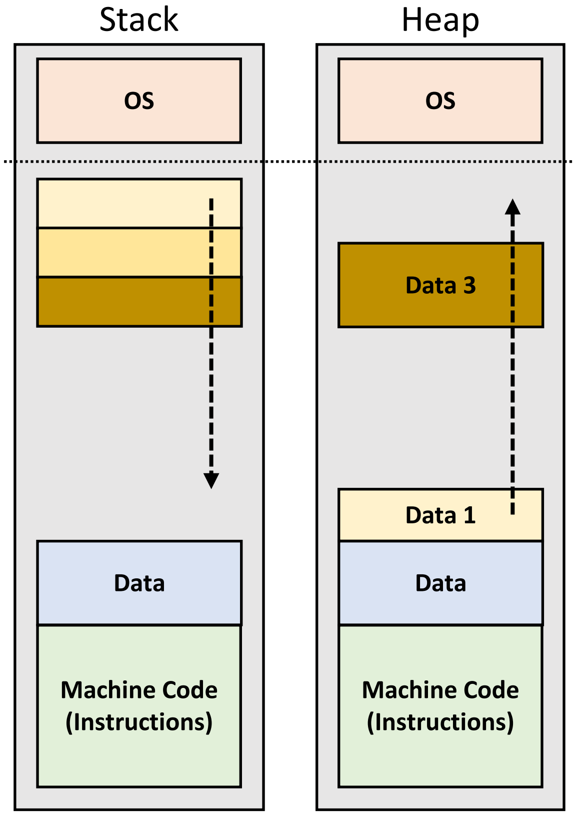
```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);
   free(data2);                          <---- Run
}

int main(void) {
   char *data1 = (char *) malloc(1);
   myfunc();

   return 0;
}
```



Provide abstraction: stack vs. heap

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - Heap
    - **User allocates** the memory with a specific size
    - **OS finds an empty space** and then place the mem.
    - Mem. fragmentation (also **mem. leak**!) can occur

```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);
   free(data2);
}

int main(void) {
   char *data1 = (char *) malloc(1);
   myfunc();

   return 0;                        <---- Run
}
```



Topics for today

- Part I: Process
  - Provide abstraction
    - What is a program?
    - What is a process?
    - How does OS run a program?
  - Offer standard libraries
    - How do we run (or stop) a process?
    - How does OS manage the process(es) we ran?
  - Manage resources
    - (Note) We will talk about this in the “scheduling” class

Offers standard interface

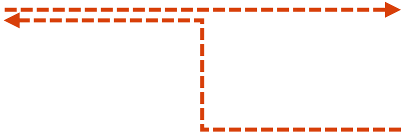
- How do we run a process?
  - Double click an icon
  - Type ./<program name> in the terminal
  - …

Offers standard interface: system call

- System call
  - **Definition:** a user-level function call to request a service from the OS
  - **Example:** when we allocate memory with "malloc()"

User program

```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);
   free(data2);
}

int main(void) {
   int num = 100;
   myfunc();

   return 0;
}
```

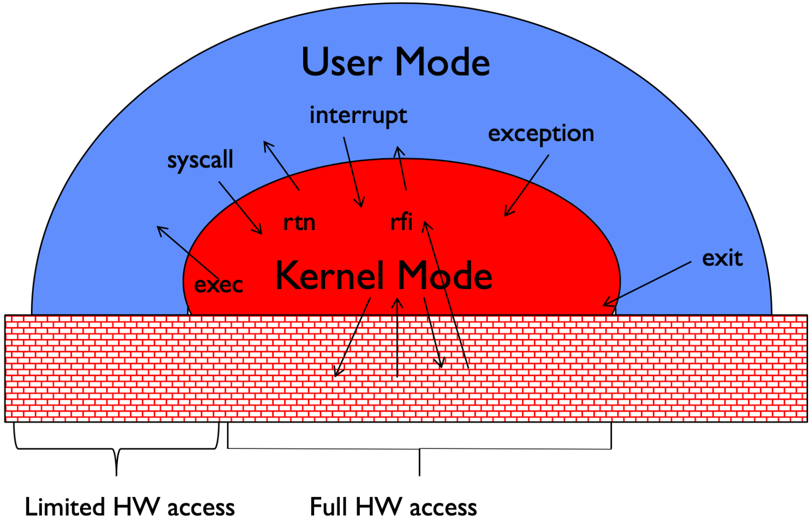


OS Kernel

Kernel allocates the memory space that has the size of 5 bytes, and return the starting address of this memory to the user's program

Offers standard interface: system call

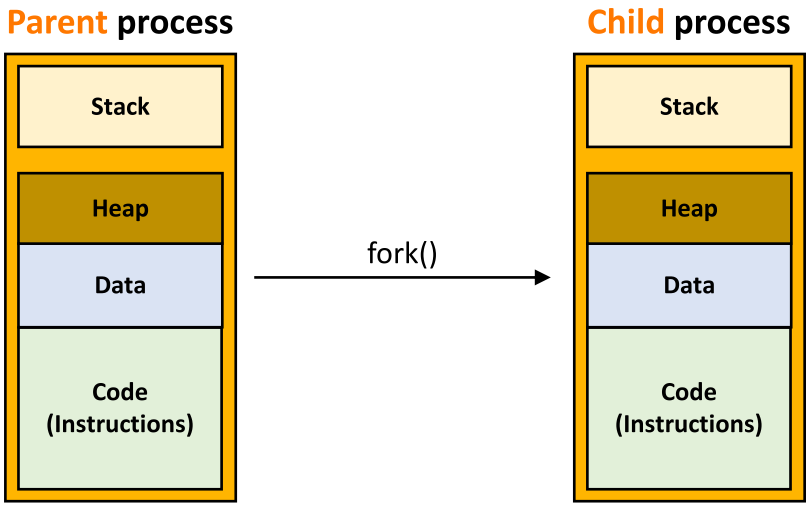
- OS offers a set of system calls
  - To create/terminate a process
  - To open/read/write/close a file
  - To request/release a device (such as display, mouse, etc.)
  - To request/modify system information
  - To initiate/close networking
  - To set the security properties
  - …



[1]Searched for this image with keyword "system calls" on Google

OFFERS STANDARD INTERFACE: FORK SYSTEM CALL

- fork() system call
  - **Operation:**
    - Create a new process that is an exact copy of the calling process
    - Return the process ID (PID) of a new process (and if it's in child, returns 0)



OFFERS STANDARD INTERFACE: FORK SYSTEM CALL

- folk() sample code in C

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
   int number = 10;
   pid_t pid;
   switch (pid = fork()) {
      case -1:
         perror ("fork");
         exit (1);
      case 0:
         number++;
         printf("I am a child process [%d]!", number);
         break;
      default:
         number--;
         printf("I am a parent process  [%d]!", number);
         break;
   }

   printf("I will be executed by both");
   return 0;
}
```

Parent process (pid = child’s PID) → **Child process** (pid = 0)

Execution result (sample):
I am a child process [11]!
I will be executed by both
I am a parent process [9]!
I will be executed by both

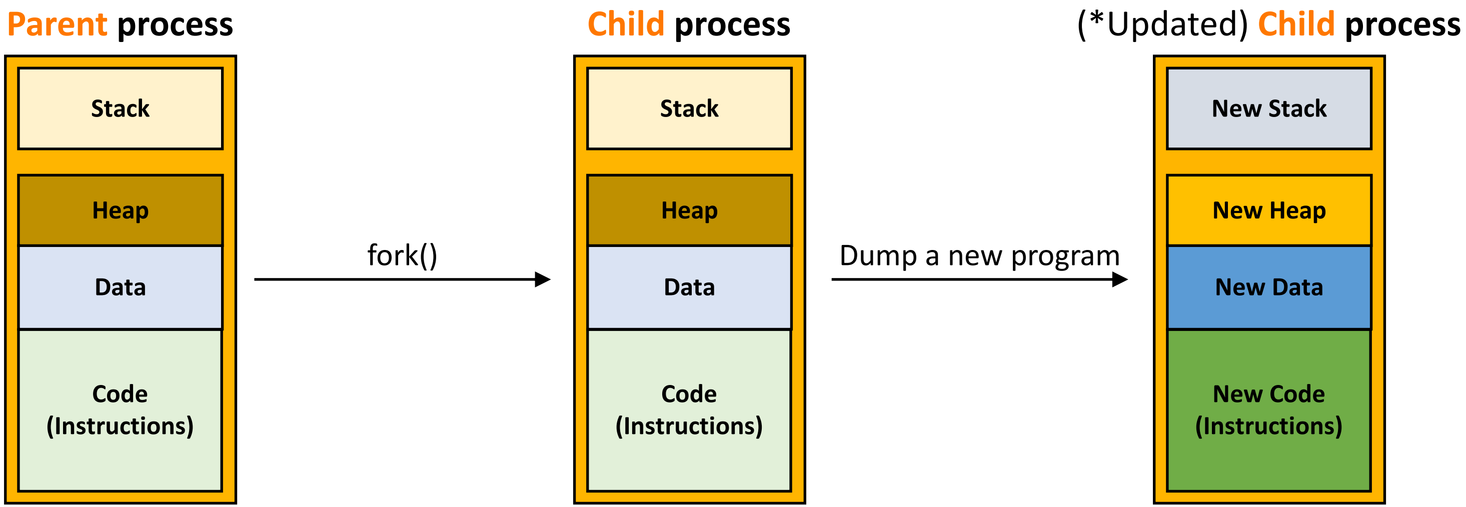
```
switch (pid = fork()) {
      case -1:
         perror ("fork");
         exit (1);
      case 0:
         number++;
         printf("I am a child process [%d]!", number);
         break;
      default:
         number--;
         printf("I am a parent process  [%d]!", number);
         break;
   }

   printf("I will be executed by both");
   return 0;
}
```

Offers standard interface: fork system call

- fork() system call
  - **Operation:**
    - Create a new process that is an exact copy of the calling process
    - Return the process ID (PID) of a new process (and if it's in child, returns 0)
- Other system calls
  - exec(program to run):
    - Create a new process with fork() and dump the program to run into it
    - Return 0 if exec() is successful; otherwise, it returns the corresponding error
  - wait(status) or wait(PID):
    - Make the current process wait until the status (of a process, PID) changes
    - Returns the PID of the process that changes the status; otherwise, -1
  - exit() or kill():
    - Terminate the process with the given PID

Offers standard interface: exec system call

- exec() system call
  - **Operation:**
    - Create a new process with fork() and dump the program to run into it
    - Return 0 if exec() is successful; otherwise, it returns the corresponding error



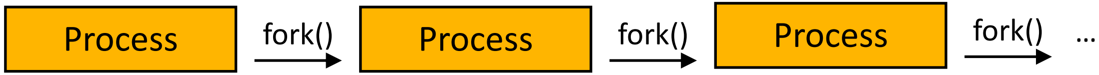
Offers standard interface: what if we do fork infinitely?

- fork() bomb (link)
  - A DoS attack that a process continuously fork() to deplete available system resources
  - **Consequence:** resource starvation
  - **Defense:** limit the number of processes a user can create (check with $ ulimit -u)

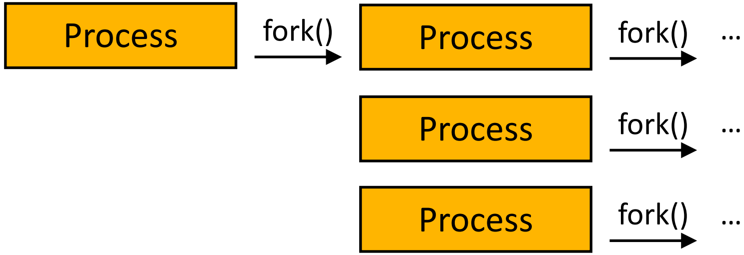
- **Take-aways**
  - An attacker can exploit the standard interfaces for achieving adversarial goals
  - We should consider the worst-cases when designing/offering such interfaces
  - Defense mechanisms should also be offered to defeat such attacks

Offer standard interface: how OS manages processes?

- Possible scenarios
  - S1: Recursively fork()



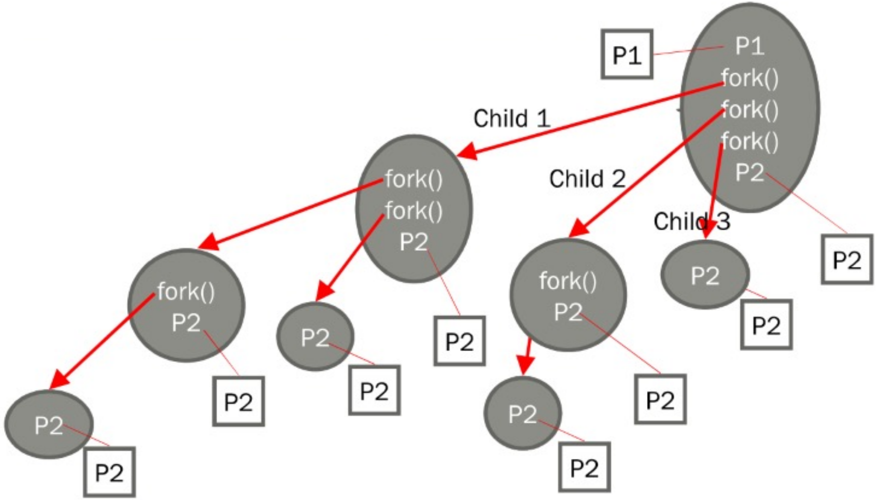
  - S2: Multiple fork()s from a process



What Would Be the Best Data Structure to Manage Processes?

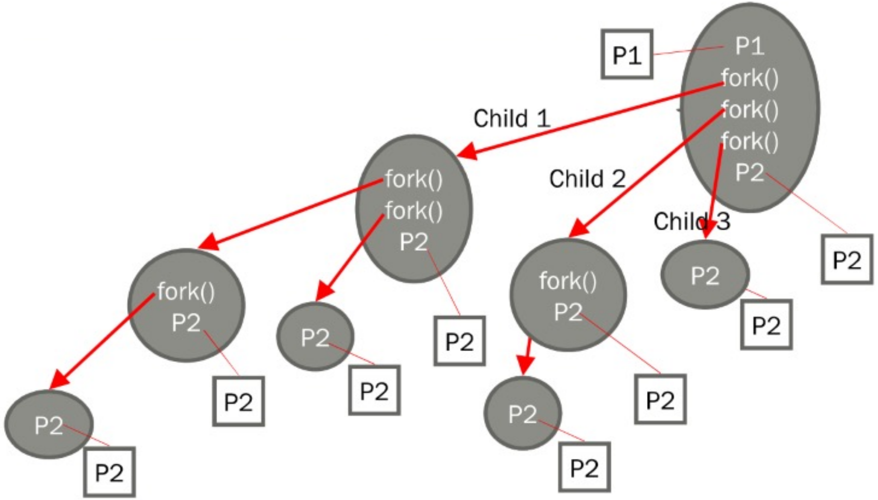
Offer standard interface: how OS manages processes?

- **fork() tree**
  - OS manages processes with a tree
  - Use ($ pstree) command to see the tree!
  - Root of the fork() tree (in Linux)
    - PID=0: **Sched** (swapper) process
    - PID=1: **Init** process



Offer standard interface: how OS manages processes?

- **fork() tree**
  - OS manages processes with a tree
  - Use ($ pstree) command to see the tree!
  - Root of the fork() tree (in Linux)
    - PID=0: **Sched** (swapper) process
    - PID=1: **Init** process



- Properties
  - User processes always have a parent
  - If we kill the parent, all the child processes will be killed, too (an exception, any process launched by $ nohup or $ disown)
  - PIDs allocated by OS increases as we fork() more

Topics covered today

- Part I: Process
  - Provide abstraction
    - What is a program?
    - What is a process?
    - How does OS run a program?
  - Offer standard libraries
    - How do we run (or stop) a process?
    - How does OS manage the process(es) we ran?
  - Manage resources
    - (Note) We will talk about this in the “scheduling” class

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab