# CS 374: Operating Systems I

# Part I – Process and threads

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu


# NOTICE – CONT'D

- Announcements
  - Office hours
    - Time: See Canvas
    - Other times: Discord
  - Notes
    - Discord: allow us a few hours to answer questions (2 TAs for 100+ students)
    - Discord: post questions to corresponding channels (e.g., #assignment-1 for the assignment 1)
    - Discord: feel free to DM instructor or TAs (Sanghyun, Jose, or Wang)
    - All: help others, when you already know answers
      (*do not share your code with others)

# NOTICE – CONT'D

- Deadlines
  - (Passed) Syllabus quiz
  - (4/13 11:59 PM) Programming assignment 1
  - (4/20 11:59 PM) Midterm quiz 1

# TOPICS FOR TODAY

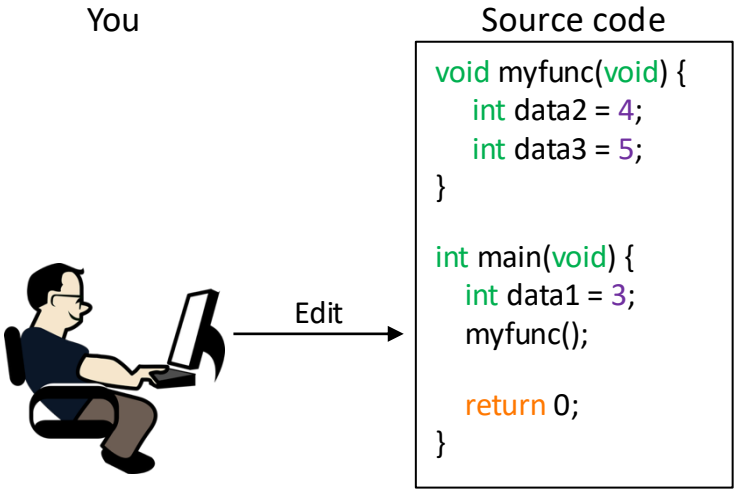
- Part I: Process
  - Provide abstraction
    - What is a program?
    - What is a process?
    - How does OS run a program?
  - Offer standard libraries
    - How do we run (or stop) a process?
    - How does OS manage the process(es) we ran?
  - Manage resources
    - (Note) We will talk about this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

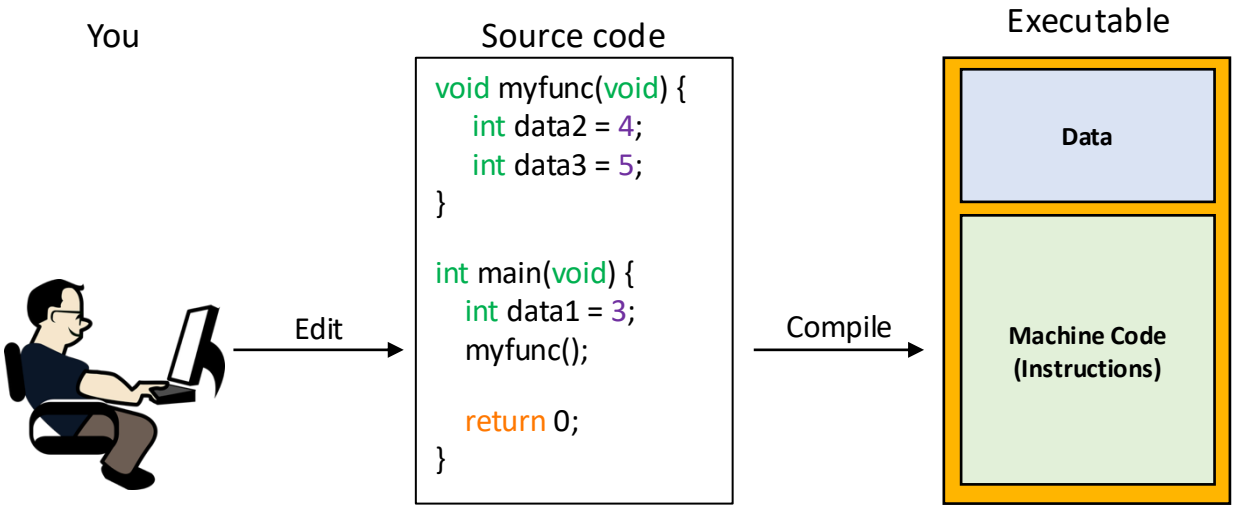
# PROVIDE ABSTRACTION: A PROGRAM

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



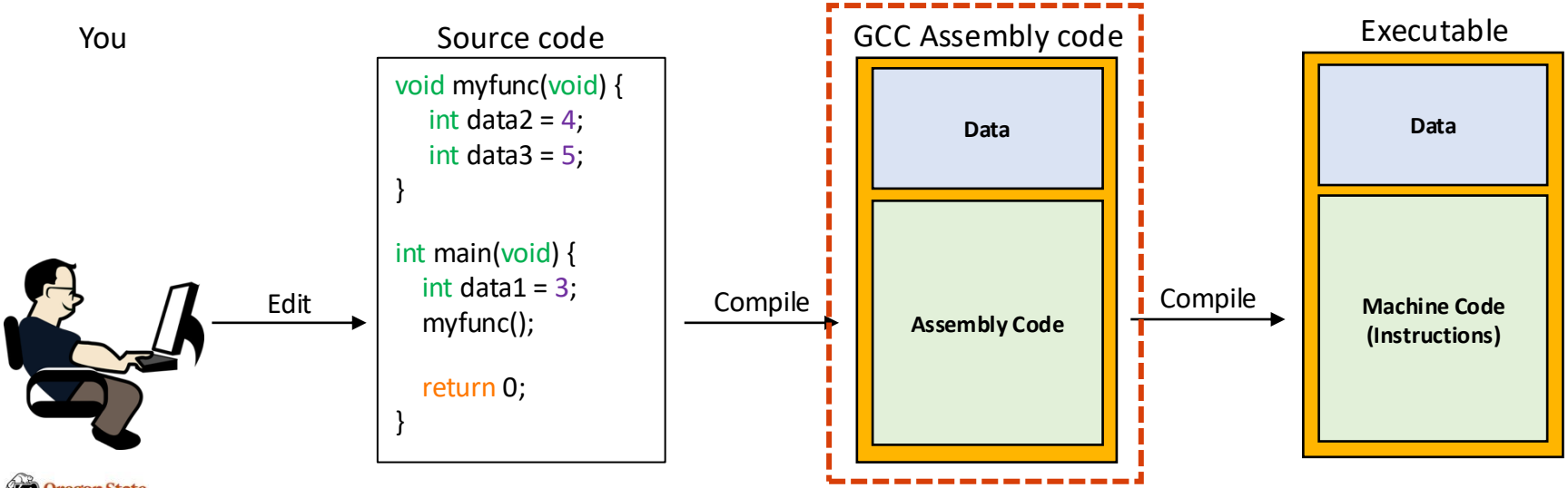
# PROVIDE ABSTRACTION: A PROGRAM

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



# 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)



# 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:
```

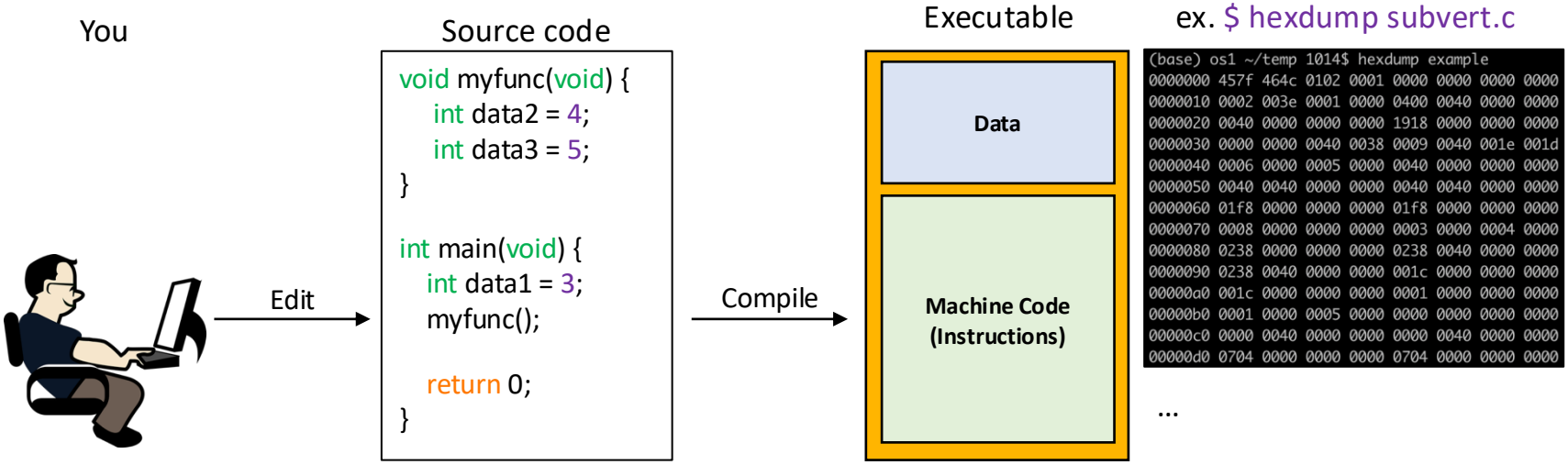
example.s

```
    .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

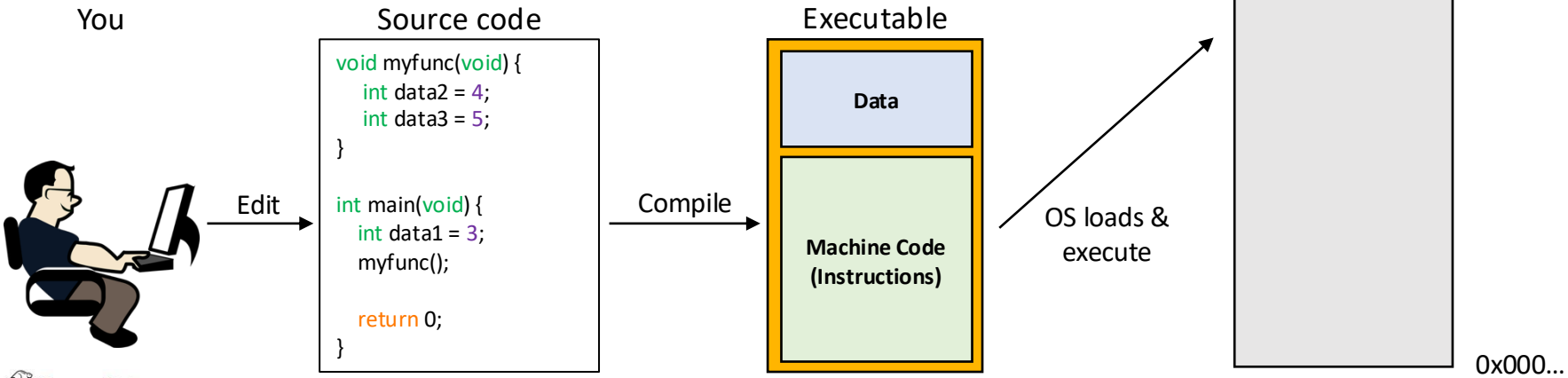
# PROVIDE ABSTRACTION: A PROGRAM

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



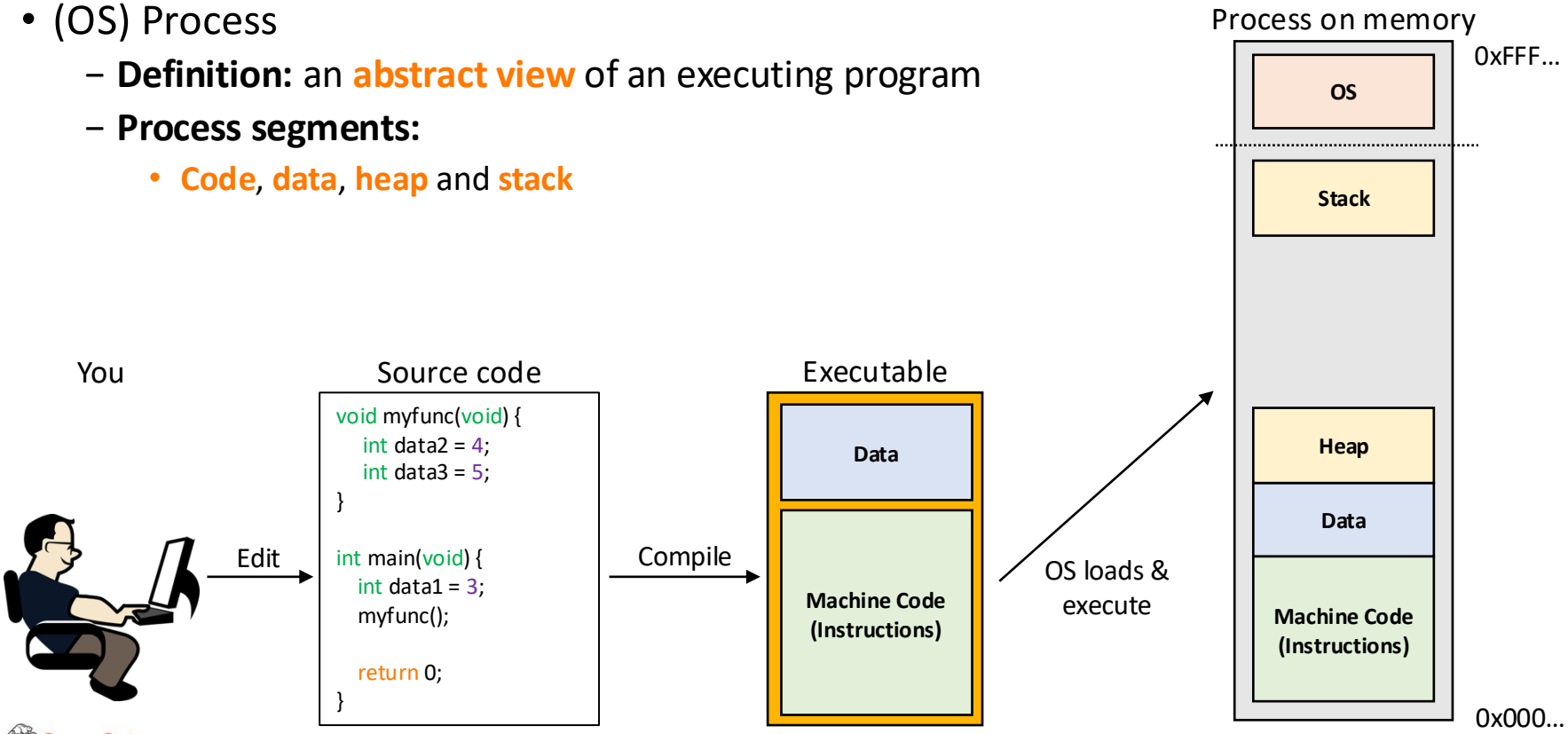
# PROVIDE ABSTRACTION: A PROCESS

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



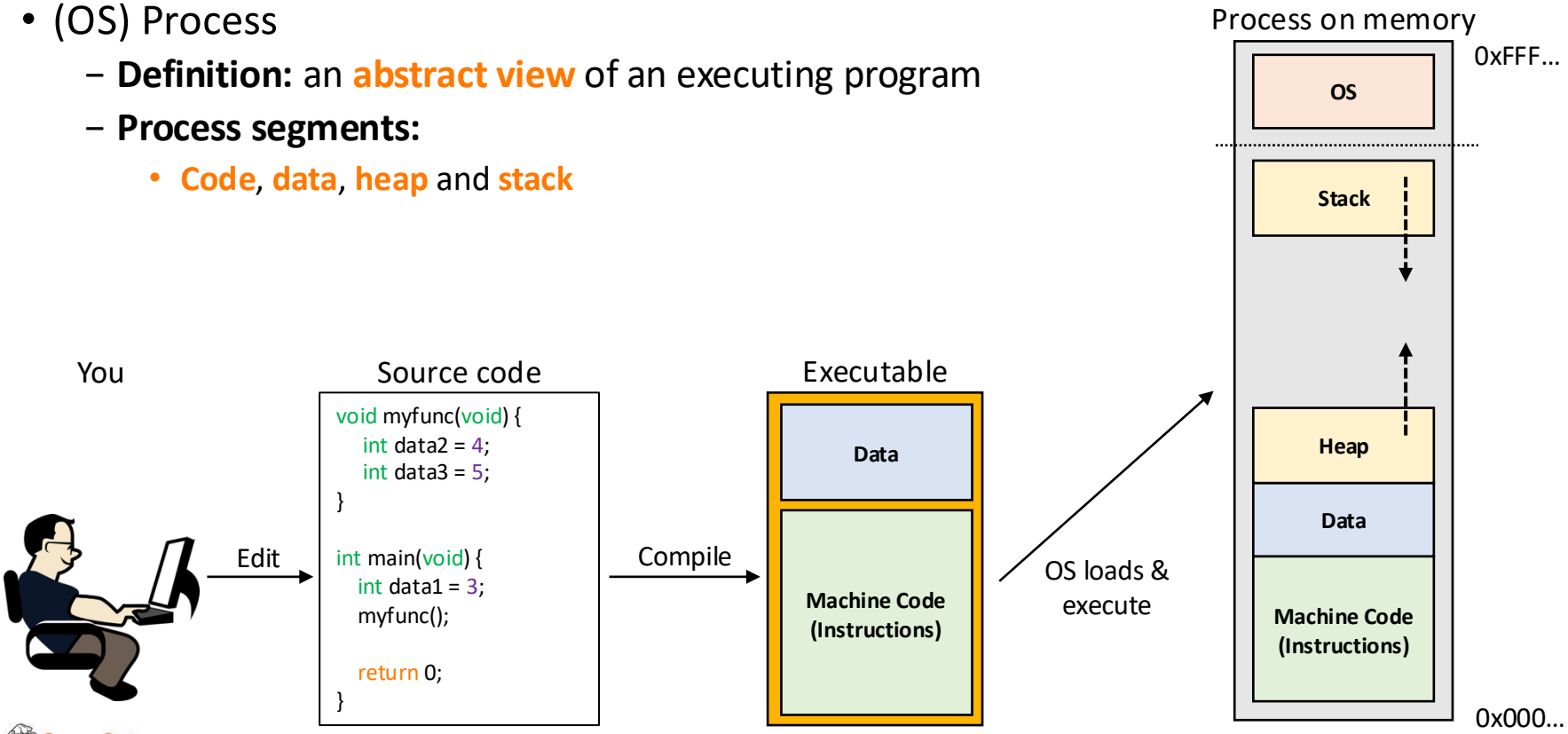
# PROVIDE ABSTRACTION: A PROCESS

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



# PROVIDE ABSTRACTION: A PROCESS

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



# 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
  - …

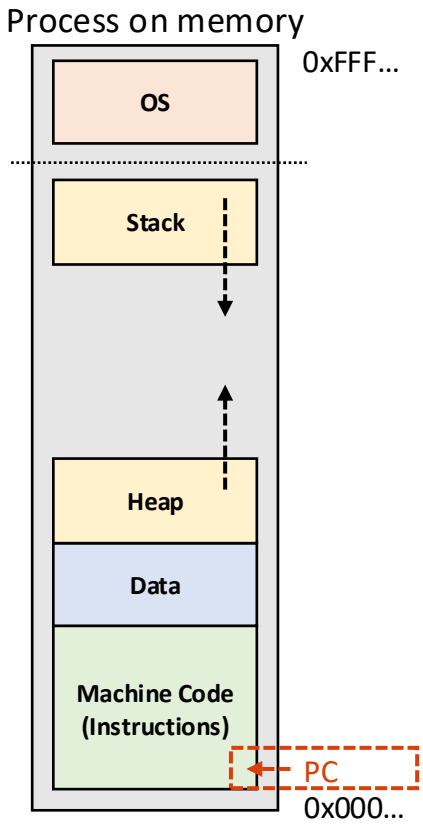
**Process Context:** A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)

```
728 struct task_struct {
729 #ifdef CONFIG_THREAD_INFO_IN_TASK
730         /*
731          * For reasons of header soup (see current_thread_info()), this
732          * must be the first element of task_struct.
733          */
734         struct thread_info              thread_info;
735 #endif
736         unsigned int                    __state;
737
738 #ifdef CONFIG_PREEMPT_RT
739         /* saved state for "spinlock sleepers" */
740         unsigned int                    saved_state;
741 #endif
742
743         /*
744          * This begins the randomizable portion of task_struct. Only
745          * scheduling-critical items should be added above here.
746          */
747         randomized_struct_fields_start
748
749         void                            *stack;
750         refcount_t                      usage;
751         /* Per task flags (PF_*), defined further below: */
752         unsigned int                    flags;
753         unsigned int                    ptrace;
```

```
852         struct sched_info               sched_info;
853
854         struct list_head                tasks;
855 #ifdef CONFIG_SMP
856         struct plist_node               pushable_tasks;
857         struct rb_node                  pushable_dl_tasks;
858 #endif
859
860         struct mm_struct                *mm;
861         struct mm_struct                *active_mm;
862
863         /* Per-thread vma caching: */
864         struct vmacache                 vmacache;
865
866 #ifdef SPLIT_RSS_COUNTING
867         struct task_rss_stat            rss_stat;
868 #endif
869         int                             exit_state;
870         int                             exit_code;
871         int                             exit_signal;
872         /* The signal sent when the parent dies: */
873         int                             pdeath_signal;
874         /* JOBCTL_*, siglock protected: */
875         unsigned long                   jobctl;
876
877         /* Used for emulating ABI behavior of previous Linux versions: */
878         unsigned int                    personality;
```

# PROVIDE ABSTRACTION: HOW OS LOADS A PROCESS?

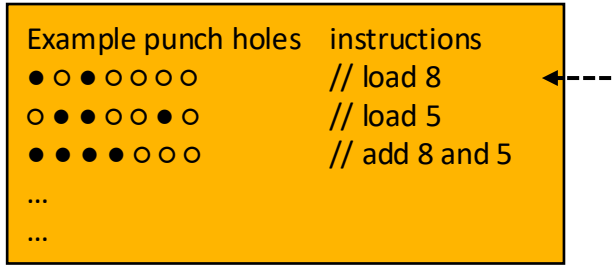
- (OS) Process
  - **Definition:** an **abstract view** of an executing program
  - **Load a process:**
    - **Code**: OS loads the instructions to "code" segments
    - **Data** : OS loads the data (such as static vars) to "data" segments
    - **Stack** and **heap**: OS creates those mem. spaces
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



# 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.**

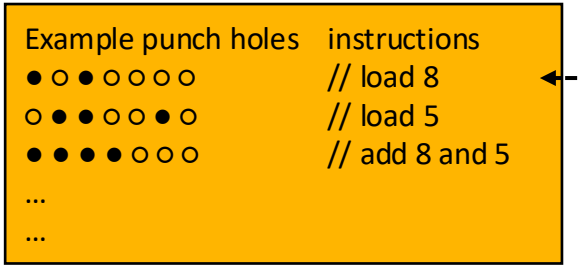
**Punch card**



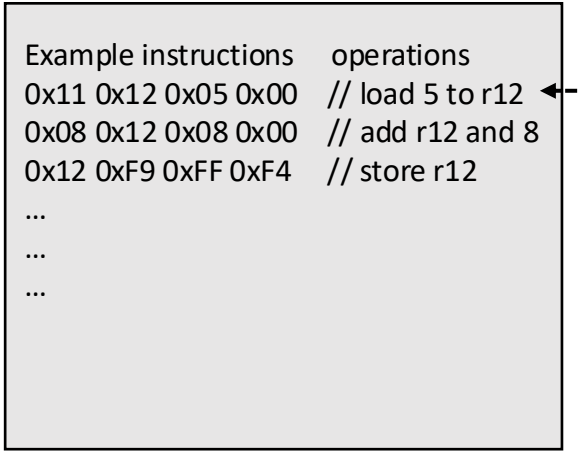
# PROVIDE ABSTRACTION: HOW OS RUNS A PROCESS? – CONT'D

- 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**




**Memory**

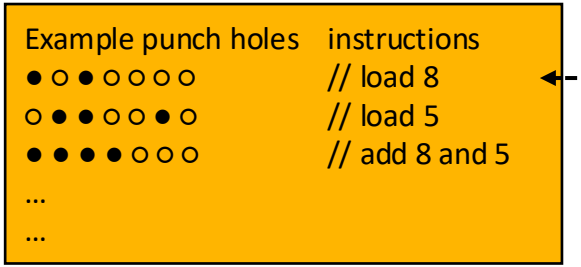


# PROVIDE ABSTRACTION: HOW OS RUNS A PROCESS? – CONT'D

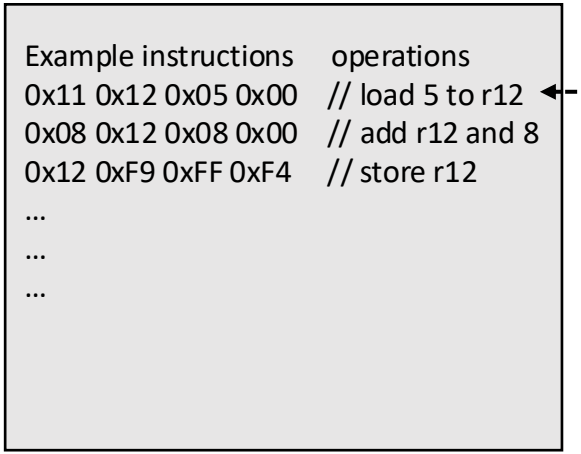
- OS makes the CPU run the machine code
  - Example: IBM machines
    - Submit **a punch card** that have a set of **instructions**
    - Machine **reads** instructions line by line and **do sth.**
  - Modern computers
    - Machine := a processor (CPU)
    - Instructions := instructions (100+ for Intel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

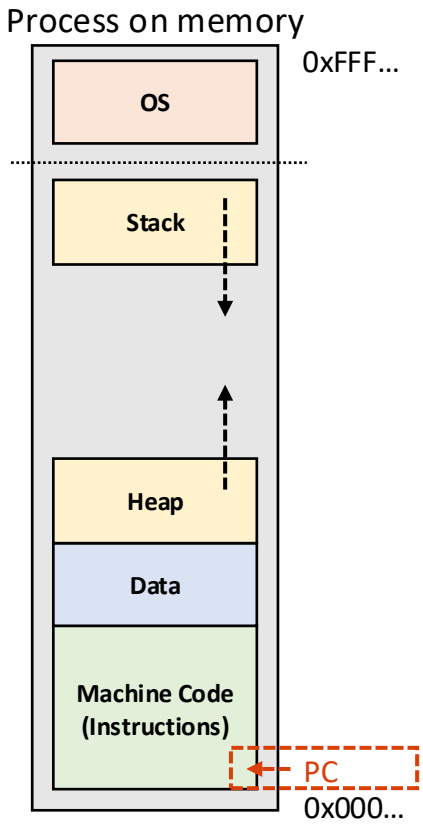



**Memory**



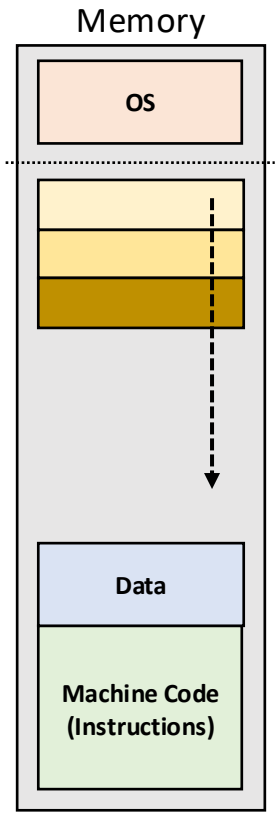
# PROVIDE ABSTRACTION: HOW OS LOADS/RUNS A PROCESS?

- (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**



# 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

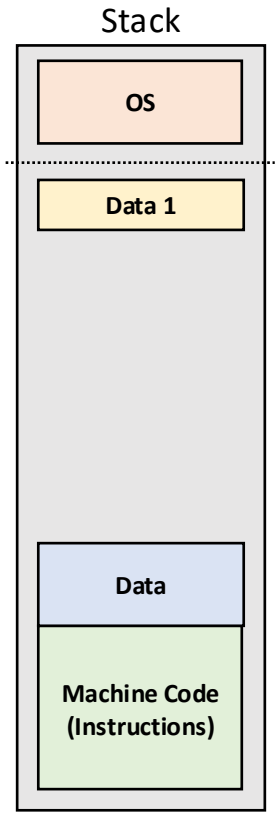


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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;
}
```

<---- Run

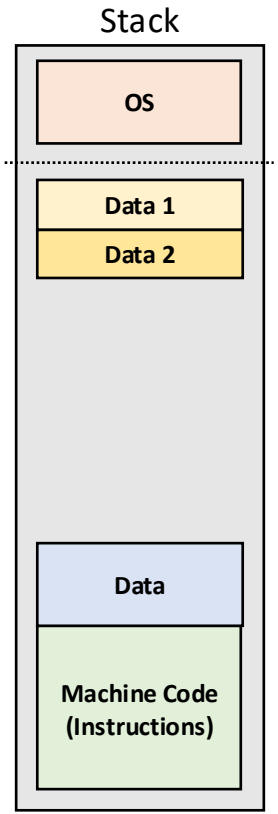


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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**

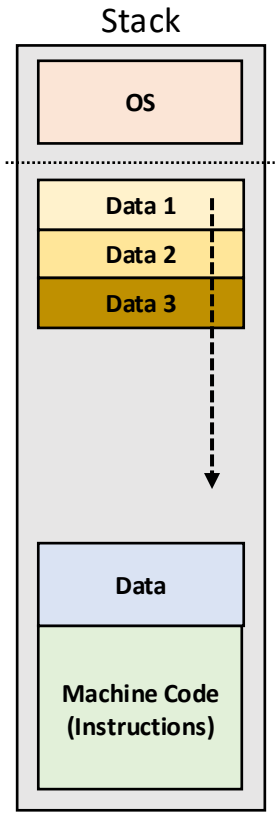


# PROVIDE ABSTRACTION: STACK VS. HEAP

- Stack vs. heap
  - **Definition:** Both are the **areas of memory**
  - 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

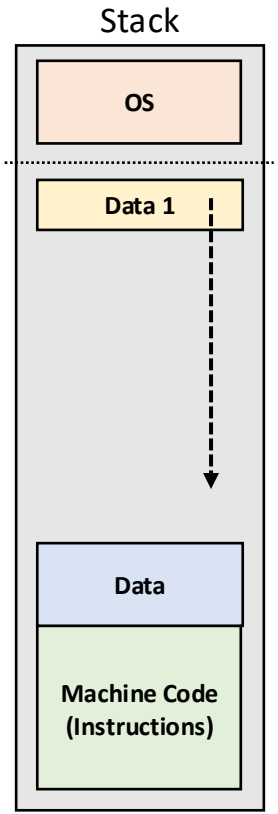


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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

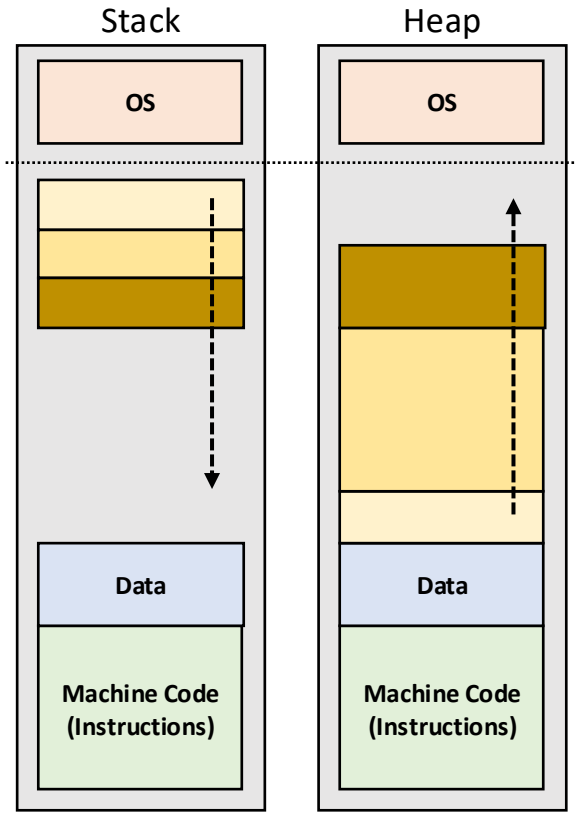
```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

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

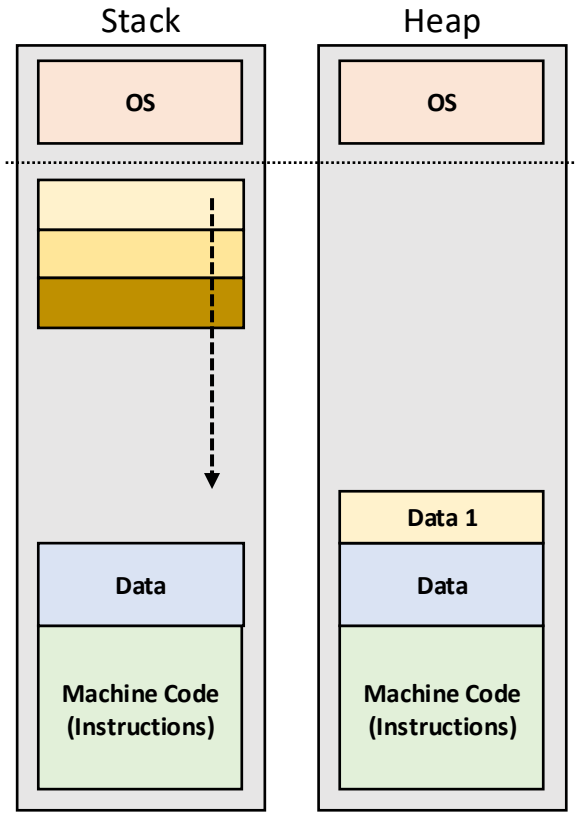


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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);
   myfunc();

   return 0;
}
```

Run

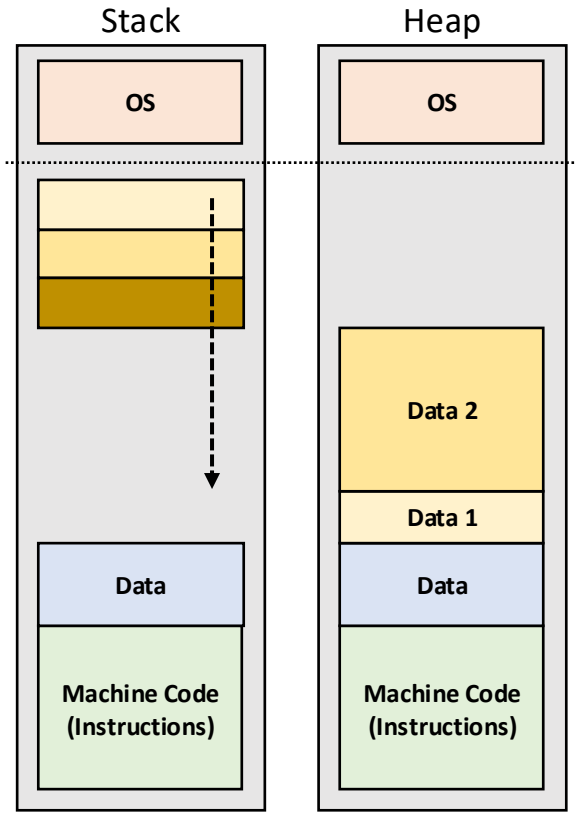


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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

```
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

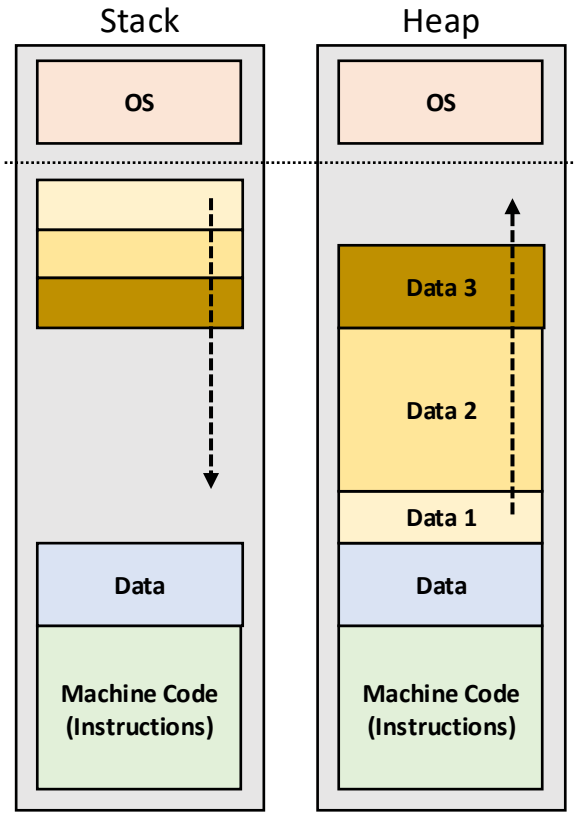


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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

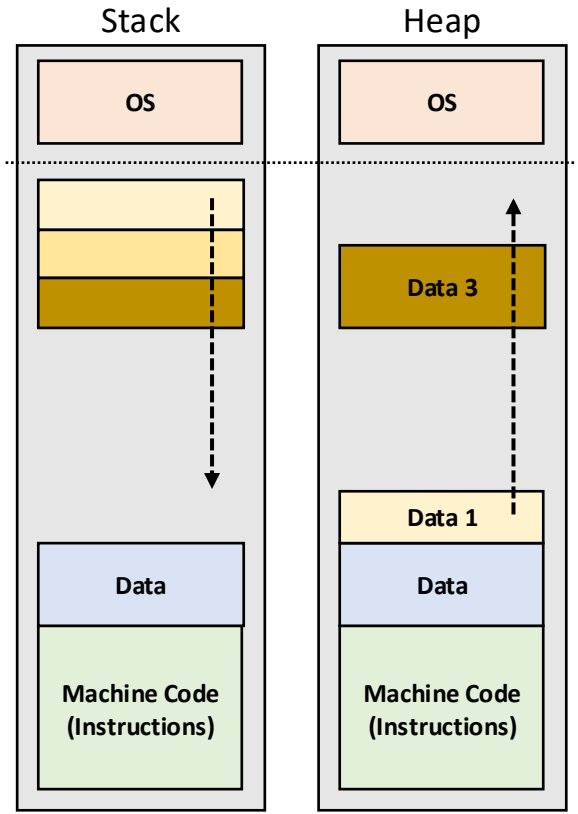


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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

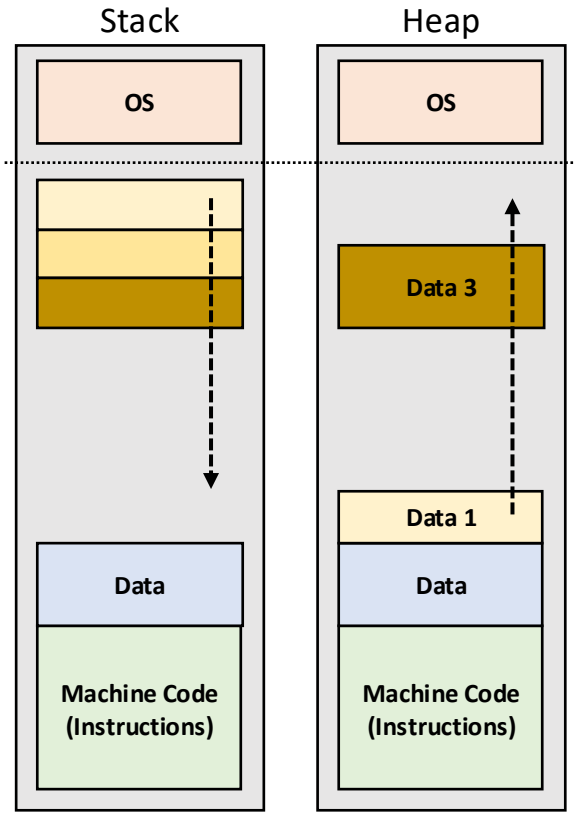


# PROVIDE ABSTRACTION: STACK VS. HEAP

- 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

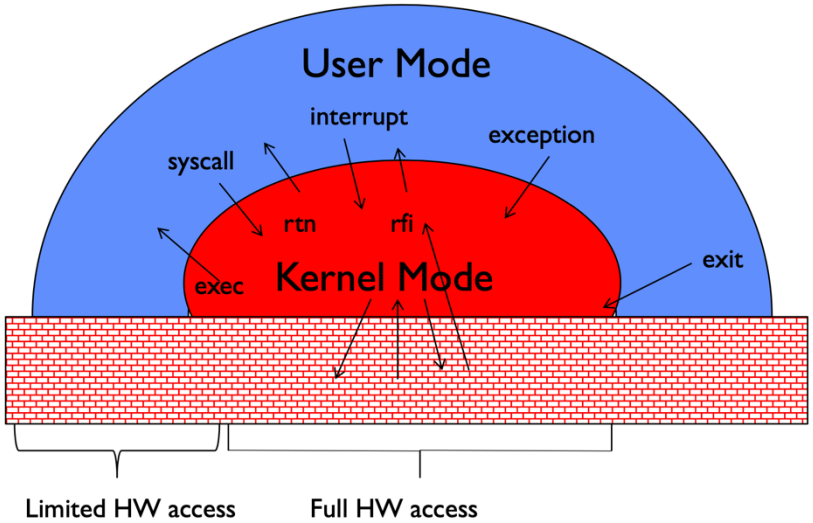
```
void myfunc(void) {
   char *data2 = (char *) malloc(5);
   char *data3 = (char *) malloc(2);
   free(data2);
}

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

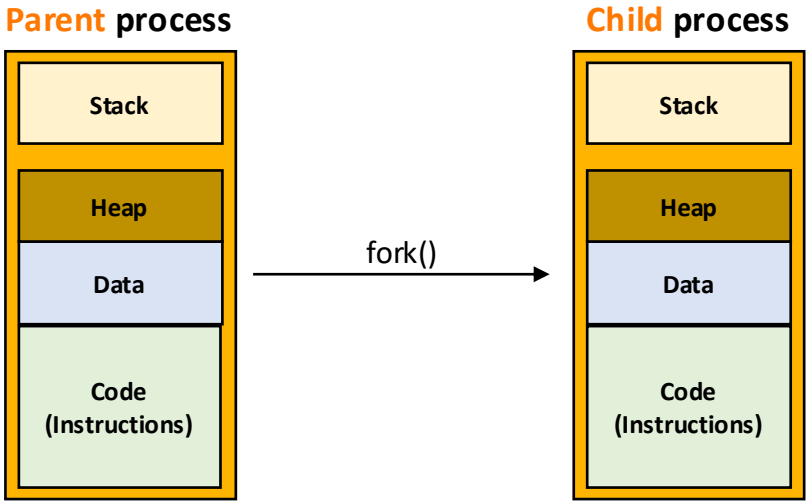
- OS offers a set of system calls
  - To create/terminate a process
  - To open/read/write/close a file
  - To request/release a device (such as display, mouse, etc.)
  - To request/modify system information
  - To initiate/close networking
  - To set the security properties
  - …




[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

- fork() 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)

```
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;
}
```

**Execution result (sample):**

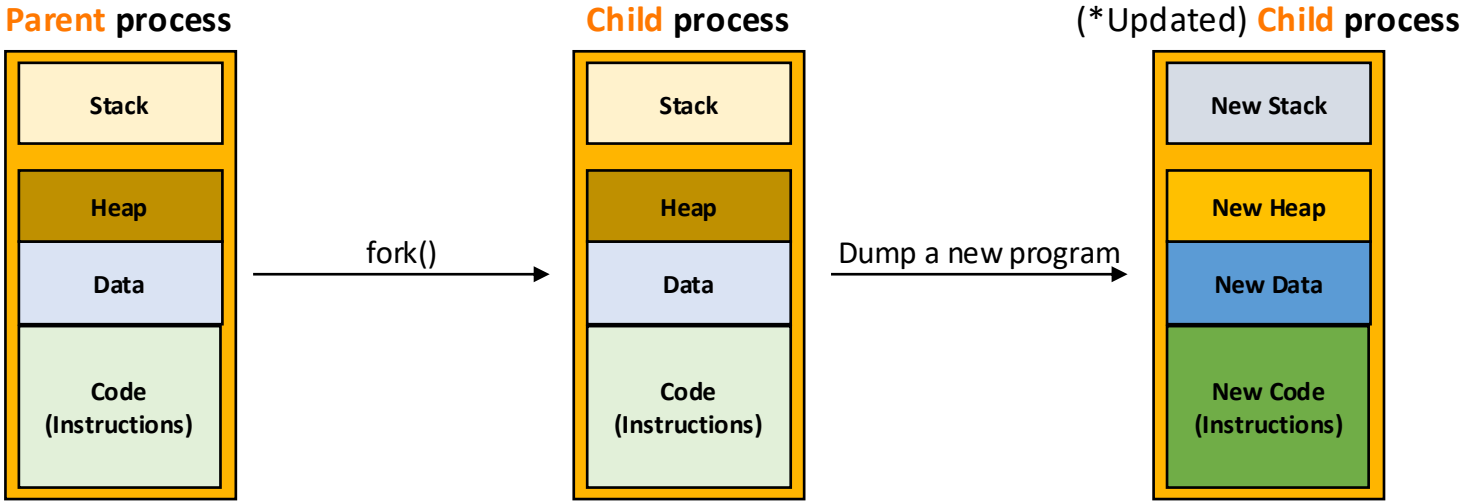
I am a child process [11]!
I will be executed by both
I am a parent process [9]!
I will be executed by both

# 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(program to run) 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?

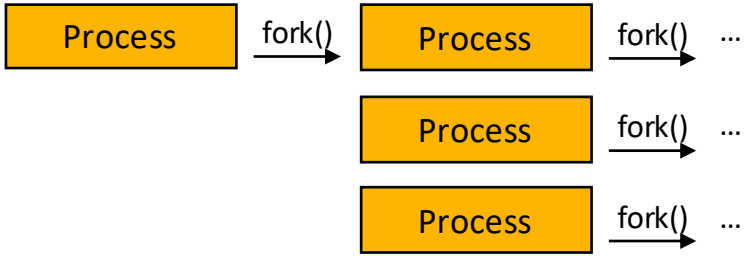
- fork() bomb (link)
  - A DoS attack that a process continuously fork() to deplete available system resources
  - **Consequence:** resource starvation
  - **Defense:** limit the number of processes a user can create (check with $ ulimit -u)

- **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()

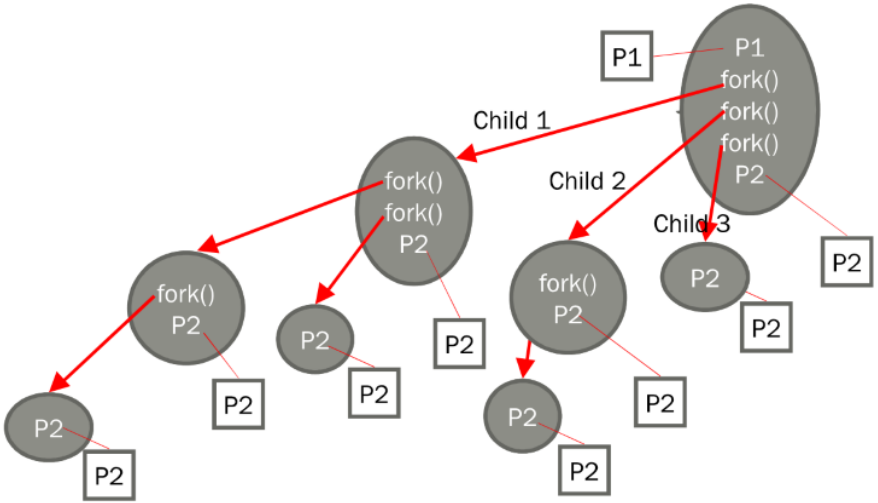

  - S2: Multiple fork()s from a process




**What Would Be the Best Data Structure to Manage Processes?**

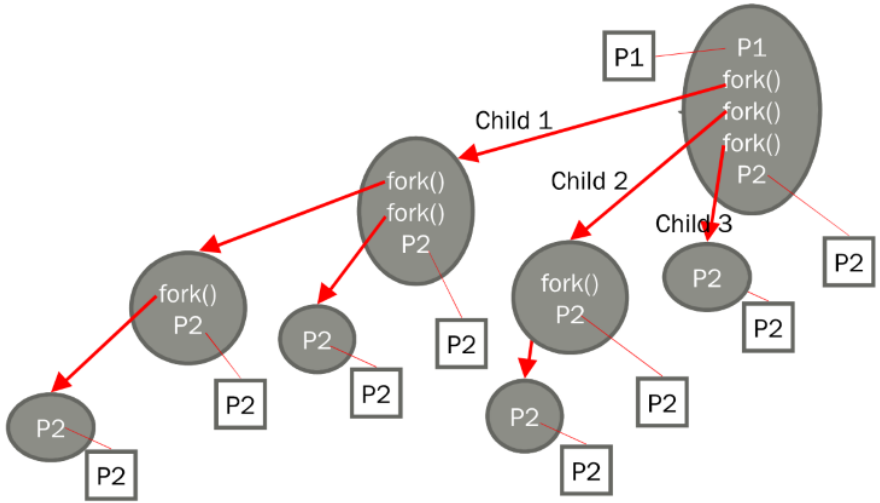
# OFFER STANDARD INTERFACE: HOW OS MANAGES PROCESSES?

- **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 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

# PRACTICE QUESTION: C

- **Static variables, functions, etc.**
  - What will be the “google” stock price?
  - What will be the prices of both stocks?
  - What will be printed out to Terminal?
  - What will be the prices of both stocks?
  - What will be printed out to Terminal?

**In C:**
1. Static variables are accessible within a file
2. Increment / decrement operators
   - Prefix: increase the value first and exec
   - Postfix: exec and increase the value

```
Required headers…

static int google_stock = 2000;

int increase_price(int stock_price, int amount) {
   stock_price = stock_price + amount;
   google_stock = google_stock + amount;
}

int main(void) {
   int apple_stock = 99;

   printf("Google stock price is %d\n", google_stock);
   printf("Apple   stock price is %d\n", apple_stock++);

   apple_stock = increase_price(apple_stock, 50);
   printf("Google stock price is %d\n", google_stock);
   printf("Apple   stock price is %d\n", ++apple_stock);

   return 0;
}
```

Run

Run

Run

# PRACTICE QUESTION: C

- **Pointers and strings**
  - What will be the value of “ilen”?
  - What will be the value of “slen”?
  - How many bytes “str” uses in memory?
  - What will be the execution result?

**In C:**
1. Pointer variable store the address (size of the var will be 4-/8-byte)
2. Pointer is required to init. String with “=”
3. strlen() returns the number of chars
4. The actual string in C ends with “\0”

```
Required headers…

int main(void) {
   int slen = 0;
   int *iptr = NULL;
Run ⇢  char str = "Hello world!";

Run ⇢  ilen = sizeof(iptr);
Run ⇢  slen = strlen(str);
   printf("The length of this string is %d\n", slen);

   return 0;
}
```

# PRACTICE QUESTION: C

- **Pointer operations**
  - What will be printed out to Terminal?
  - What will be printed out to Terminal?
  - What will be printed out to Terminal?
  - What will be printed out to Terminal?

**In C:**
1. Increasing pointer accesses the next addr != Increasing pointer value (*ptr)+1
2. Pointer holds the address of a variable

```
Required headers…

int swap(int num1, int *num2) {
   int temp = num1;
   num1 = num2;
   *num2 = temp;
   return num1;
}

int main(void) {
   int val1 = 1;
   int val2 = 2;
   int vals[] = { 10, 20, 30, 40, 50 };
   int *ptr = vals;

Run ···▸  printf("Val1 / 2 / 3: %d, %d, %d\n", val1, val2, vals[0]);
Run ···▸  printf("Val1 / 2: %d, %d\n", *(ptr+2), (*ptr)+2);

   val1 = swap(val1, ptr);
Run ···▸  printf("Val1 / 2 / 3: %d, %d, %d\n", val1, val2, vals[0]);
Run ···▸  printf("Val1 / 2: %d, %d\n", *(ptr+2), (*ptr)+2);

   return 0;
}
```

# PRACTICE QUESTION: PROCESS

- **Segments (components) of a process**
  - Which segment “counter1” is?
  - Which segment “ret” is?
  - Which segment “counter2” is?
  - Which segment “buf” is?
  - What are the counter1 and 2 values?
  - Which segment “ret” is?
  - Which segment “buf” is?

**In Heap:**
Memory fragmentation can happen
Memory leak can happen

```
Required headers…

#define  BUFSIZE   512

static int counter1 = 0;

int my_function() {
Run ···➔   int counter2 = 2;
Run ···➔   char *buf = (char *) malloc(BUFSIZE * sizeof(char));

   counter1 = counter1 + 1;
   counter2 = counter2 – 1;
Run ···➔   return counter2;
}

int main(void) {
Run ···➔   int ret = 0;

Run ···➔   ret = my_function();
   printf("Ret: %d\n", ret);

   return 0;
}
```

# TOPICS FOR TODAY

- Part I: Threads
  - Provide abstraction
    - What is a thread?
    - How is it different from a process?
    - How does OS run threads?
  - Offer standard libraries
    - How do we create/run/kill a thread?
    - How does OS manage the thread(s) we ran?
  - Manage resources
    - (Note) We will talk about this in the “scheduling” and “synchronization” classes

# RUNNING MULTIPLE PROCESSES: WEB-SERVER EXAMPLE

- Amazon.com:
  - What does the webserver do?

# WEB-SERVER EXAMPLE

- Amazon.com:
  - A user requests the website
  - A server accepts the connection
  - A server sends the webpage to the user
  - A user clicks something
  - A server sends the webpage as a response
  - … (continue)

**Pseudo code (server)**

```
int main(void) {
   // 1. server accepts the connection
   connection = accepts(user-request, …)

   // 2. server sends the webpage to user
   sends_webpage(connection, html-page)

   // 3. server starts accepting the user requests
   while (action = receive_request(connection)) {
      if (action == login) {
            if (!correct_credential(action.id, action.pw))
               return -1;    // return error, login fail
            connection.login_success = 1;
      }

      ….
   }

   return 0;    // halt the webserver, never reached
}
```

# WEB-SERVER EXAMPLE – CONT'D

- Amazon.com:
  - A user requests the website
  - A server accepts the connection
  - A server sends the webpage to the user
  - A user clicks something
  - A server sends the webpage as a response
  - … (continue)

**What would be a potential problem?**

**Pseudo code (server)**

```
int main(void) {
   // 1. server accepts the connection
   connection = accepts(user-request, …)

   // 2. server sends the webpage to user
   sends_webpage(connection, html-page)

   // 3. server starts accepting the user requests
   while (action = receive_request(connection)) {
      if (action == login) {
            if (!correct_credential(action.id, action.pw))
               return -1;   // return error, login fail
            connection.login_success = 1;
      }

      ….
   }

   return 0;   // halt the webserver, never reached
}
```

# WEB-SERVER EXAMPLE – CONT'D

- Amazon.com:
  - A user requests the website
  - A server accepts the connection
  - A server sends the webpage to the user
  - A user clicks something
  - A server sends the webpage as a response
  - … (continue)

This procedure will be the **same** for all users
> **Multi-process** web-server

**Pseudo code (server)**

```
int main(void) {
   // 1. server accepts the connection
   connection = accepts(user-request, …)

   // 2. server sends the webpage to user
   sends_webpage(connection, html-page)

   // 3. server starts accepting the user requests
   while (action = receive_request(connection)) {
      if (action == login) {
            if (!correct_credential(action.id, action.pw))
               return -1;    // return error, login fail
            connection.login_success = 1;
      }

      ….
   }

   return 0;    // halt the webserver, never reached
}
```

# MULTI-PROCESS WEB-SERVER EXAMPLE

- Amazon.com:
  - A user requests the website
  - A server accepts the connection
  - A server sends the webpage to the user
  - A user clicks something
  - A server sends the webpage as a response
  - … (continue)

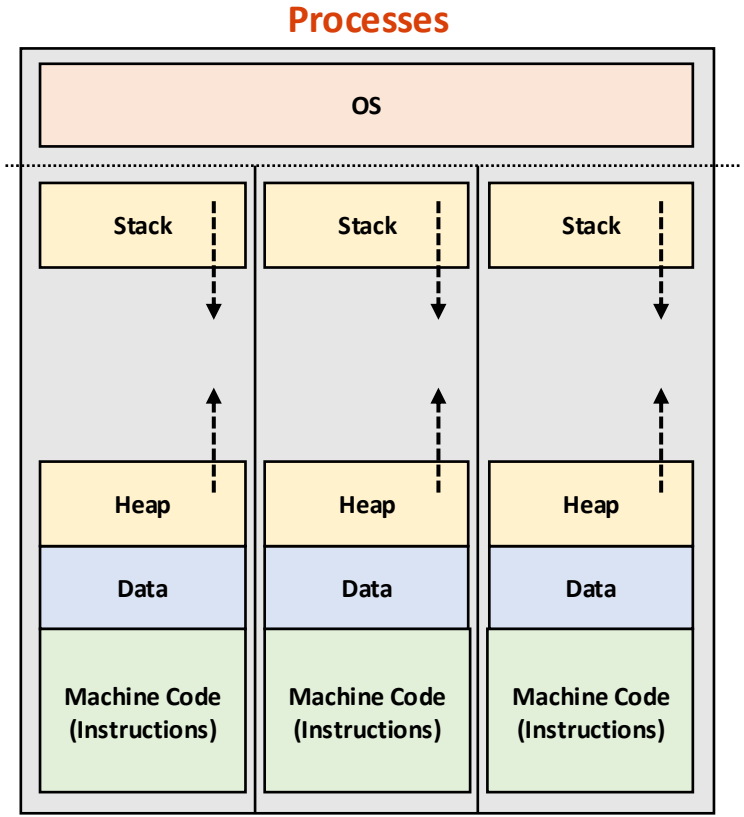
This procedure will be the **same** for all users
> **Multi-process** web-server

**Pseudo code (server)**

```
int main(void) {

  while(connection = accepts(user-request, …)) {
    // fork: create a new process
    switch(pid = fork()) {
      case 0:
        // server sends the webpage to user
        sends_webpage(connection, html-page)

        // server starts accepting the user requests
        while (action = receive_request(connection)) {
          if (action == login) {
            if (!correct_credential(action.id, action.pw))
              return -1;   // return error, login fail
            connection.login_success = 1;
          }
          ….
        }
    }   // end of switch …
  } …
```

# MULTI-PROCESS WEB-SERVER EXAMPLE: OS VIEW

- Amazon.com:
  - A user requests the website
  - A server accepts the connection
  - A server sends the webpage to the user
  - A user clicks something
  - A server sends the webpage as a response
  - … (continue)

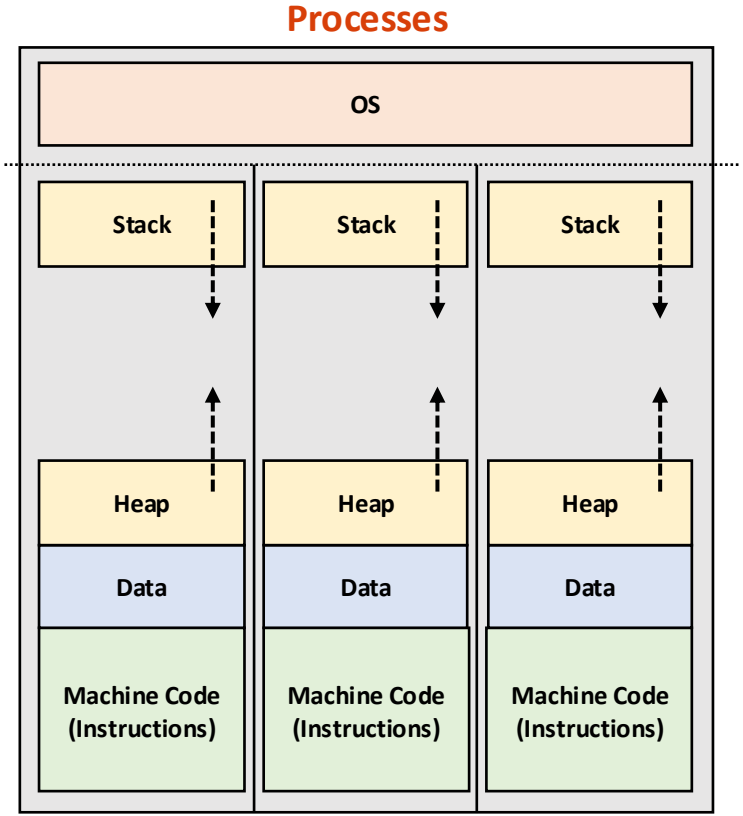


# MULTI-PROCESS WEB-SERVER EXAMPLE: POTENTIAL ISSUES

- Data is not shared between processes
  - A user requests the website
  - … (continue)

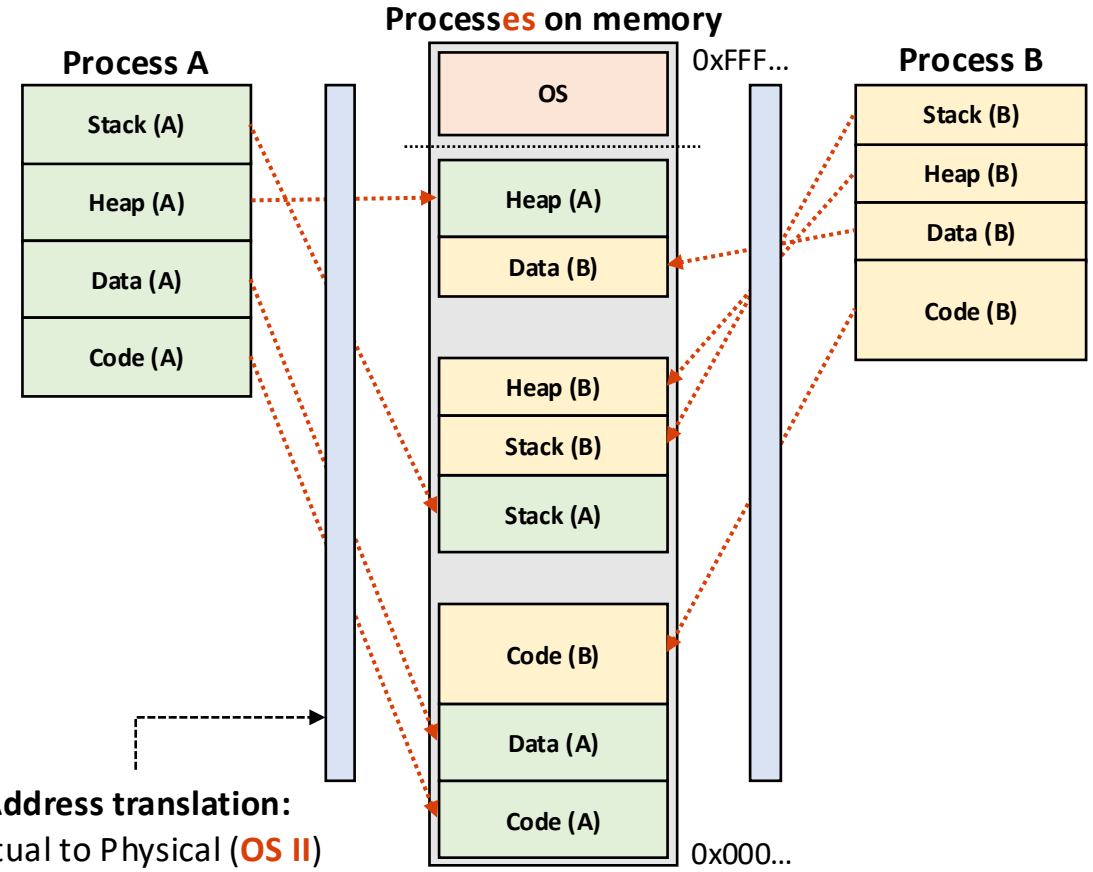
```
int main(void) {

  // initialize some data in this block

  while(connection = accepts(user-request, …)) {
    // fork: create a new process
    switch(pid = fork()) {
      case 0:
        // server sends the webpage to user
        sends_webpage(connection, html-page)

        …
```

The data in the above block **won't be shared** between processes; each process will have a copy of the same data (*causes memory overhead)



# NOTE: WHY ISN'T THE DATA SHARED BETWEEN PROCESSES?

- **Process isolation**
  - No segment is shared
  - Security reasons
    - Data breach
    - System crashes
    - Control other processes
    - …
  - **Access: seg-faults!**



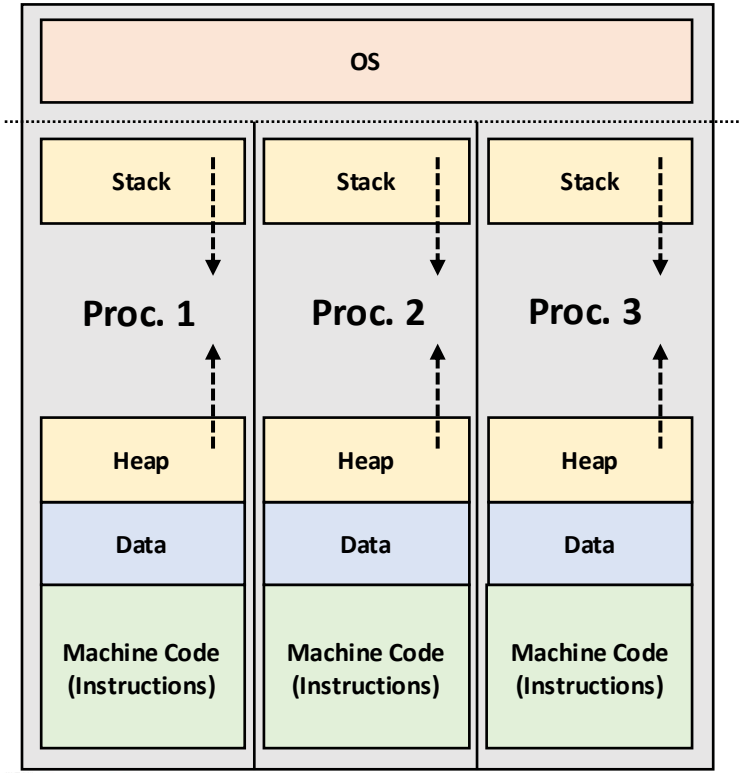
# SOLUTION: THREADS

# PROVIDE ABSTRACTION: A THREAD

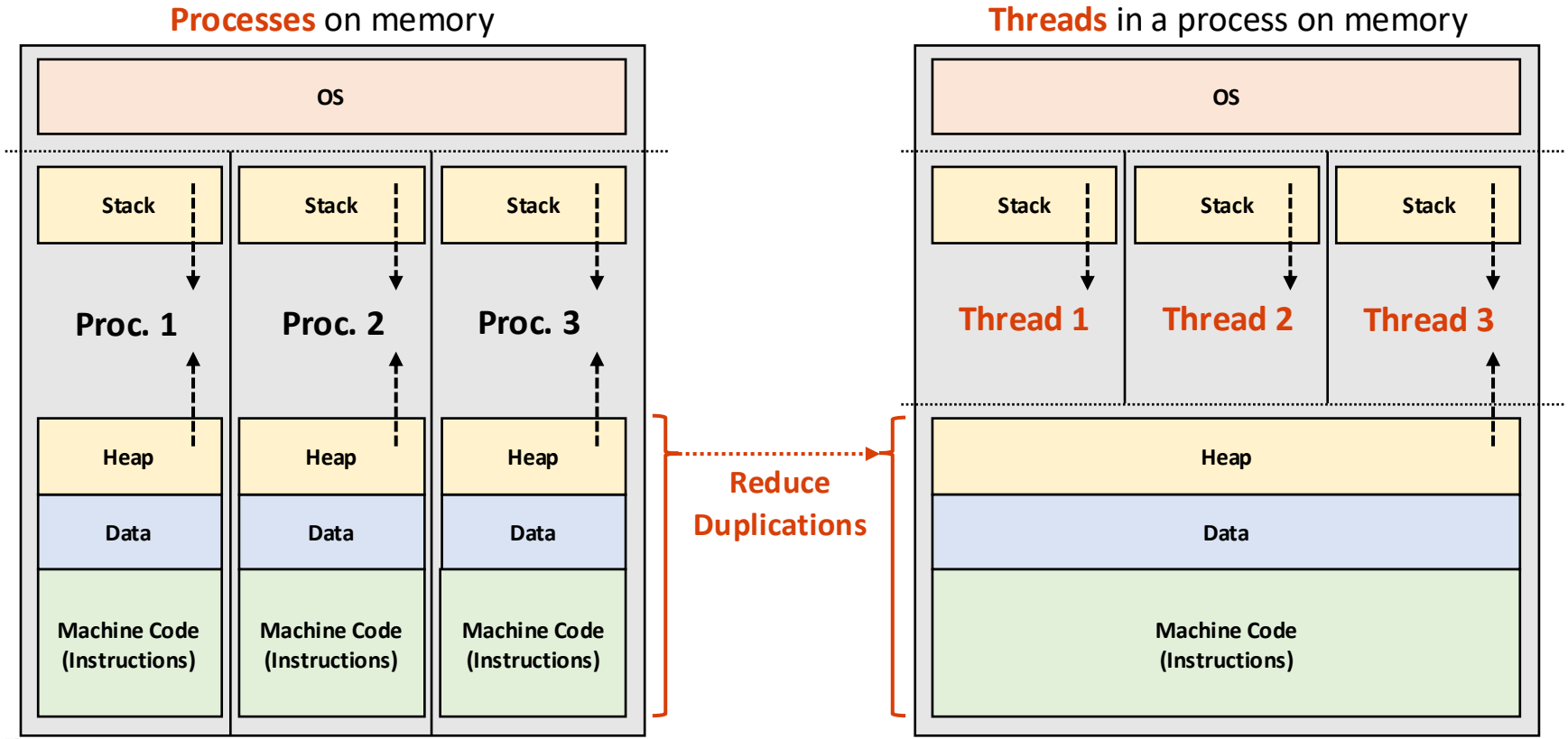
- Thread
  - **Definition:** a smallest schedulable execution context
  - **Terminology:**
    - Smallest: it's much light-weight than a process
    - Schedulable execution context: one thread can run on a CPU at a time

# PROVIDE ABSTRACTION: A THREAD – CONT'D

**Processes** on memory



# PROVIDE ABSTRACTION: A THREAD – CONT'D

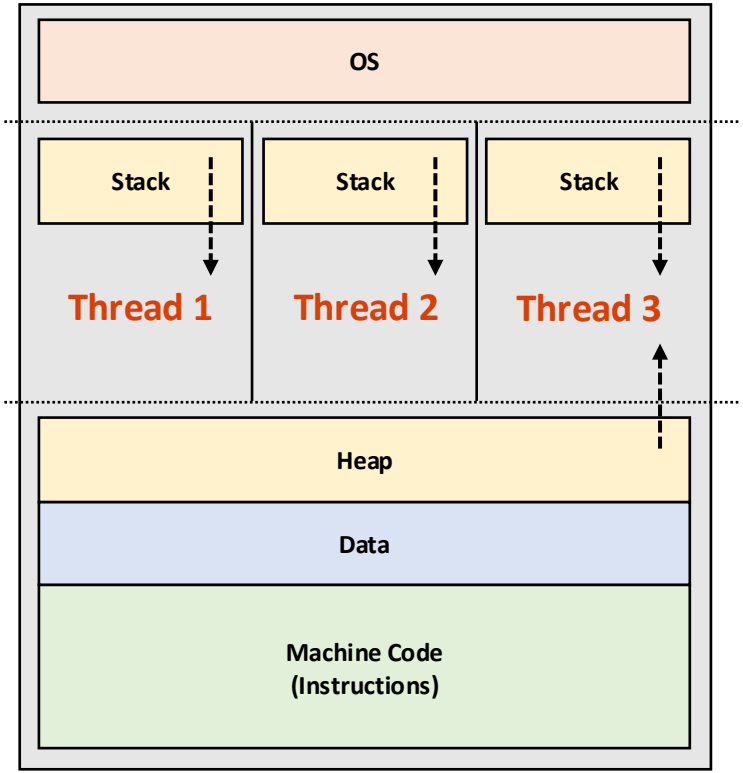


# PROVIDE ABSTRACTION: HOW IS IT DIFFERENT FROM A PROCESS?

- Threads share:
  - **Code** and **data** segments
  - **Heap** memory (ex. global variables)
  - Open files (ex. I/O access points)

- Threads **do not** share:
  - **Stack** segments, e.g.:
    - arguments passed when we launch them
    - local variables we initialize within them
    - return address, when they terminate (OS II)
  - Running contexts, e.g.:
    - process state
    - stack pointer
    - …

**Threads** in a process on memory



# PROVIDE ABSTRACTION: HOW OS DEFINES A THREAD?

- (Linux) has “thread control block”
  - **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
  - …

**Thread Control Block:** A set of information that OS requires to run a thread on a CPU, different from CPU vendors (ex. In Linux, it’s the same: *task_struct*, Link)

```
728 struct task_struct {
729 #ifdef CONFIG_THREAD_INFO_IN_TASK
730         /*
731          * For reasons of header soup (see current_thread_info()), this
732          * must be the first element of task_struct.
733          */
734         struct thread_info              thread_info;
735 #endif
736         unsigned int                    __state;
737
738 #ifdef CONFIG_PREEMPT_RT
739         /* saved state for "spinlock sleepers" */
740         unsigned int                    saved_state;
741 #endif
742
743         /*
744          * This begins the randomizable portion of task_struct. Only
745          * scheduling-critical items should be added above here.
746          */
747         randomized_struct_fields_start
748
749         void                            *stack;
750         refcount_t                      usage;
751         /* Per task flags (PF_*), defined further below: */
752         unsigned int                    flags;
753         unsigned int                    ptrace;
```

```
852         struct sched_info               sched_info;
853
854         struct list_head                tasks;
855 #ifdef CONFIG_SMP
856         struct plist_node               pushable_tasks;
857         struct rb_node                  pushable_dl_tasks;
858 #endif
859
860         struct mm_struct                *mm;
861         struct mm_struct                *active_mm;
862
863         /* Per-thread vma caching: */
864         struct vmacache                 vmacache;
865
866 #ifdef SPLIT_RSS_COUNTING
867         struct task_rss_stat            rss_stat;
868 #endif
869         int                             exit_state;
870         int                             exit_code;
871         int                             exit_signal;
872         /* The signal sent when the parent dies: */
873         int                             pdeath_signal;
874         /* JOBCTL_*, siglock protected: */
875         unsigned long                   jobctl;
```

**A process and a thread are the same for OS**

# Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu