National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Process

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

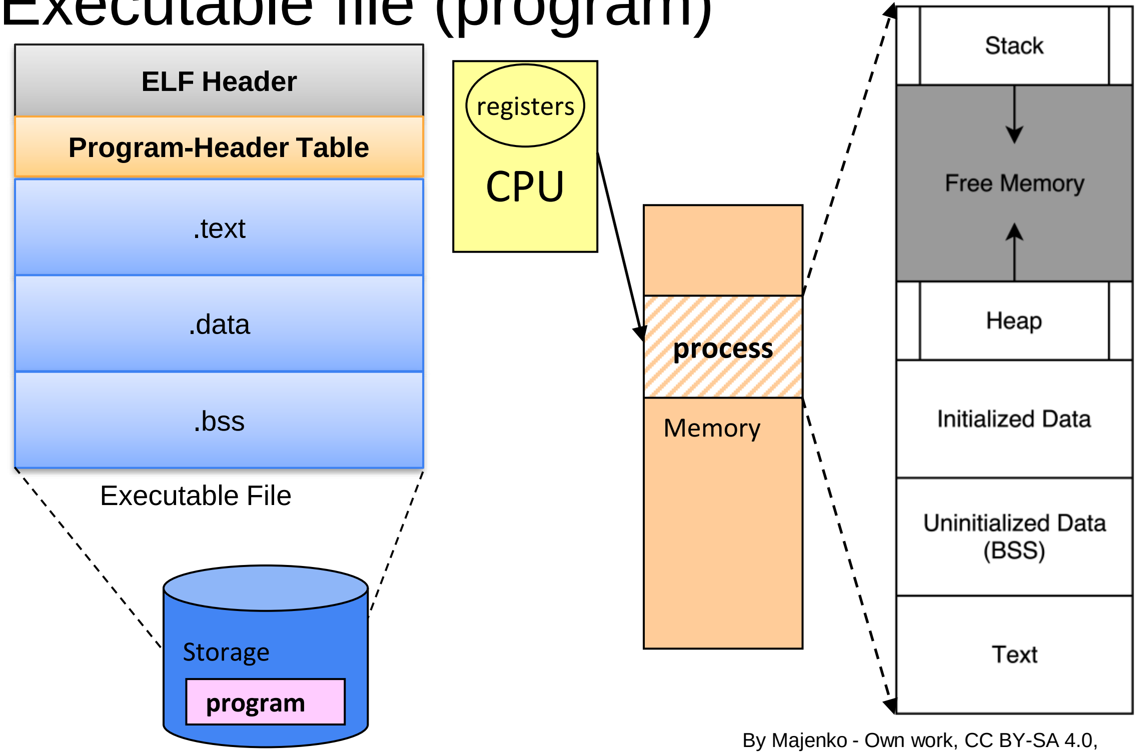
Program vs. Process

- A program
  - A program can create several processes
  - ELF header + program-header table + .text + .data + .bss
  - placed on hard drive
- A process
  - A process is a unique isolated entity
  - Prevent one process from wrecking on another process’s memory, CPU, file descriptors and the kernel itself
  - Dynamic instruction of code + heap + stack + process state
  - Placed on main memory

Program vs. Process

Executable file (program)

Memory layout (Process)



Program vs. Process

- **Process**
  - When a program is loaded into memory along with all the resources it needs to operate
  - <u>Each process has a separate memory address space</u>
  - A process runs independently and is isolated from other processes
  - How do multiple processes share a single CPU?
    - Context switch
    - Require some amount of time for saving and loading registers, memory, and other resources

Program vs. Process

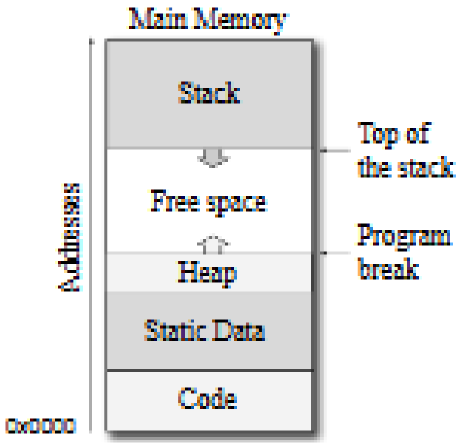
- **Process isolation**
  - A process has a private memory address space, which other processes cannot read or write
  - Each process has a separate page table that defines that process's address space
  - The page table translates a virtual address (the address that an instruction manipulates) to a physical address (an address that the CPU sends to main memory)

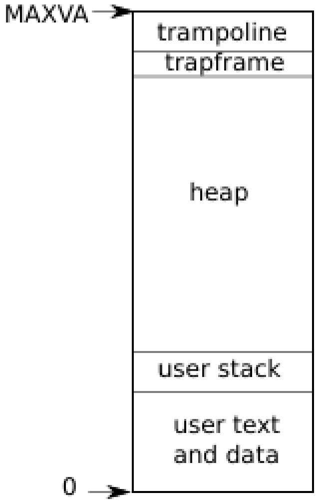
Outline

- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

The Memory Layout of a Process

- A stored-program
  - Stores both data and code on memory
  - The code space is a memory space
    - stores program codes starting at virtual address 0
  - The static data space is a memory space
    - Store the program static data (global variables)
  - The heap space is a memory space
    - Managed by the memory allocation librar (malloc())

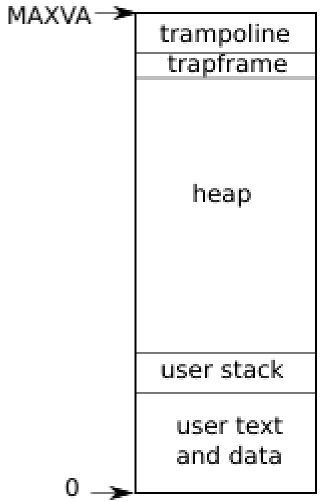




XV6 Kernel

The Memory Layout of a Process

- In 64-bit xv6 kernel
  - The hardware uses the low 39 bits when looking up virtual addresses in page tables
  - Xv6 only uses 38 of those 39 bits
  - The max address is $2^{38} - 1$ = 0x3fffffffff (MAXVA)
  - A page for trampoline/trapframe
    - Xv6 uses these two pages to transition into the kernel and back
    - Trampoline page contains codes to transition in/out the kernel
    - Mapping the trapframe save/restore the state of user process



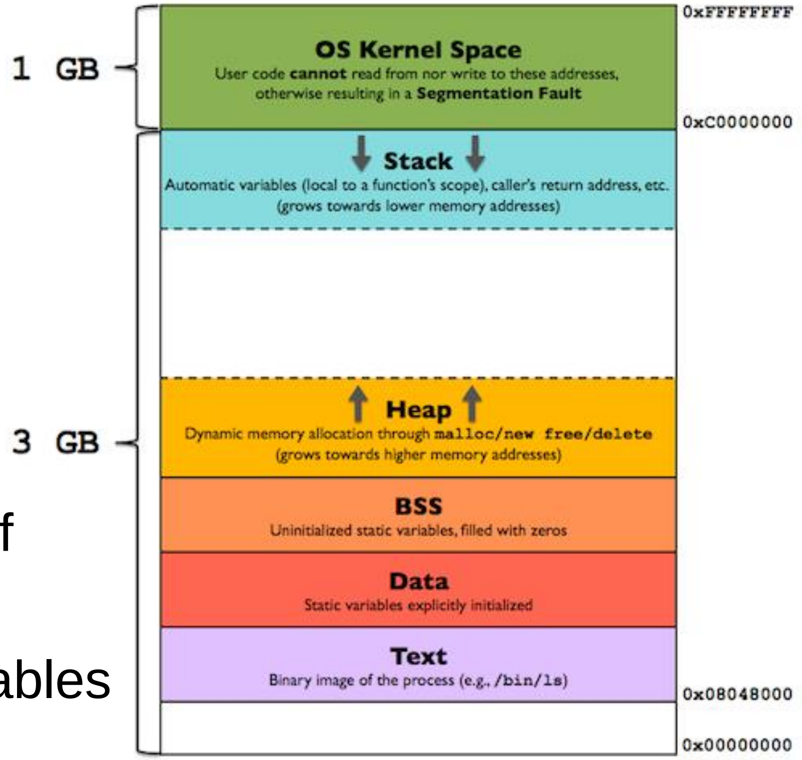
XV6 Kernel

The Memory Layout of a Process

- A processor's most important pieces of kernel state
  - Page table
  - Kernel stack
  - Run state

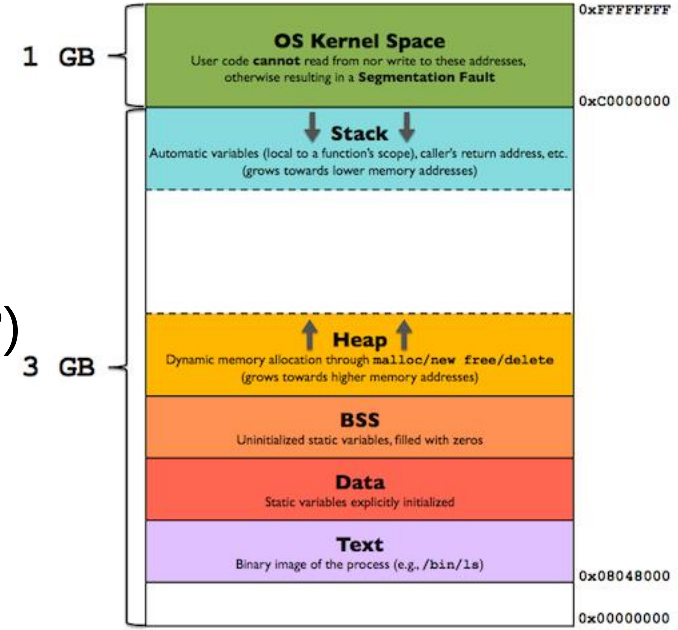
In-Memory Layout of a Process (1/6)

- **On a 32-bit machine**
  - Each process has 4 GB virtual address
  - 3GB – User
  - 1GB – kernel space
    - Shared among processes
    - Directly mapped to 1GB of RAM
    - Store kernel code, page tables



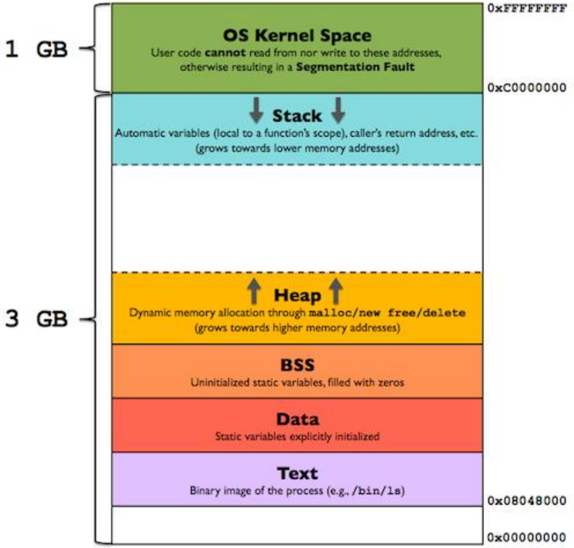
In-Memory Layout of a Process (2/6)

- **Text (code) segment**
  - Contains executable instructions of a program
  - Placed below the heap or stack (why?)
    - Prevent overflows from overwriting it
  - The text segment is often read-only/ execute (why?)
    - Prevent a program from accidentally being changed



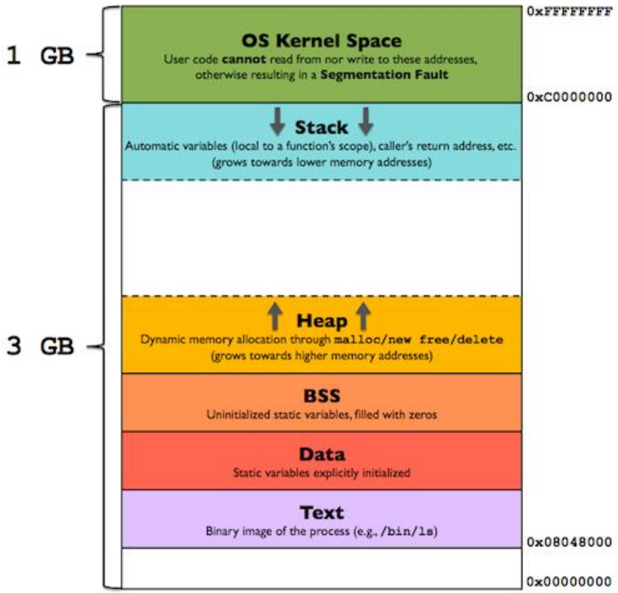
In-Memory Layout of a Process (3/6)

- **Data segment**
  - Initialized data segment
  - Contains global and stack variables initialized by the programmer
  - Not read-only (why?)
    - The values of variables can be altered
  - Read-only area (RoData)
    - const char *str = “hello world”
  - Read-Write area
    - char s[ ] = “hello world”



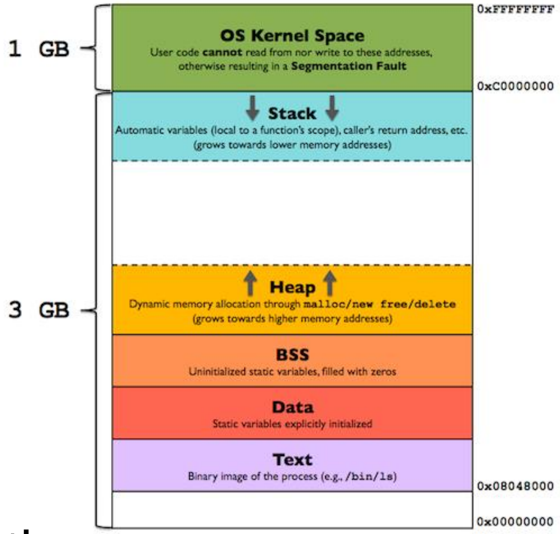
In-Memory Layout of a Process (4/6)

- **BSS segment**
  - Uninitialized data segment
  - This segment starts at the end of the data segment
  - Contains all **global and static** variables that are initialized to zero or don't have explicit initialization. E.g. **static int i;**
  - Read-write area



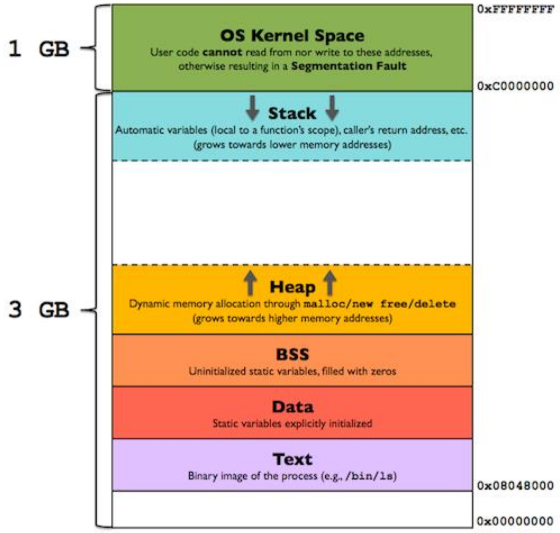
In-Memory Layout of a Process (5/6)

- **Stack**
  - Locate in the higher memory addresses right below the OS kernel space
  - Could switch the stack and heap?
  - Store all the automatic variables
    - Parameters passed as input to the function
    - The caller's return address
  - A **stack pointer** register tracks the top of the stack



In-Memory Layout of a Process (6/6)

- **Heap**
  - Dynamic memory allocation usually takes place
  - Managed by malloc/new, free/delete
  - Use the brk and sbrk system calls to adjust its size

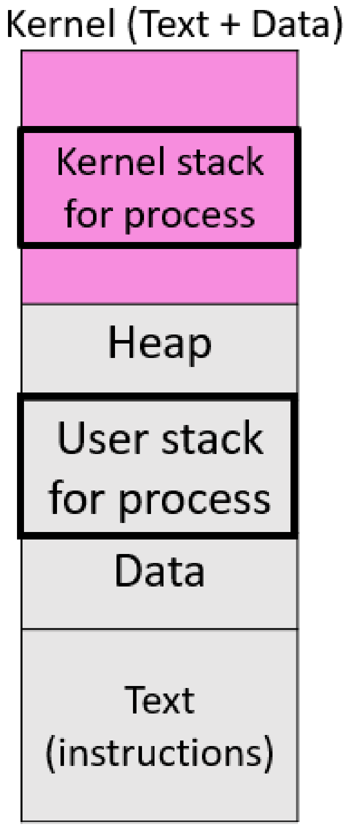


Outline

- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

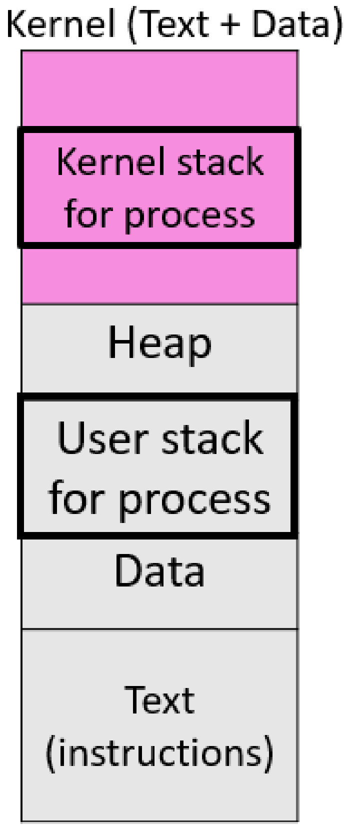
Process stacks

- **Kernel vs. user space stack**
  - **Kernel stack**
    - In the kernel space
    - During the syscall, the kernel stack of the running process is used
    - The size of the kernel stack is configured during compilation and remain fixed.
      - Two pages (8KB) for each thread
    - <u>When the process is executing user insn</u>
      - <u>Only its user stack is in use</u>
  - **Why is a separate kernel stack used?**



Process stacks

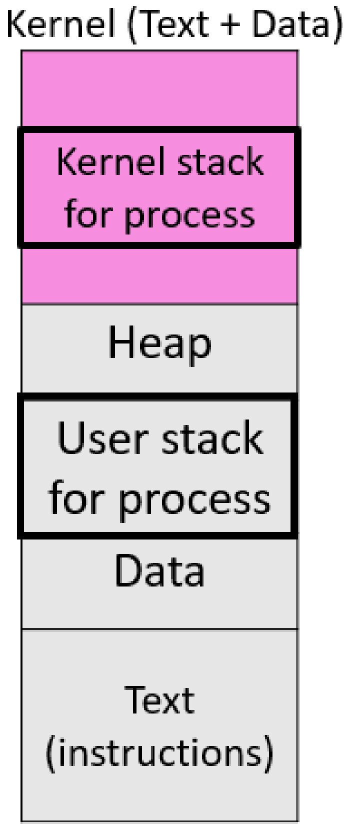
- **Why is a separate kernel stack used?**
  - Separation of privileges and security
  - The kernel cannot trust the user space stack pointer to be valid nor usable
  - The kernel can execute even if a process has wreck its user stack
- **Does each process have its own kernel stack?**
  - Each thread has its own kernel stack



Process stacks

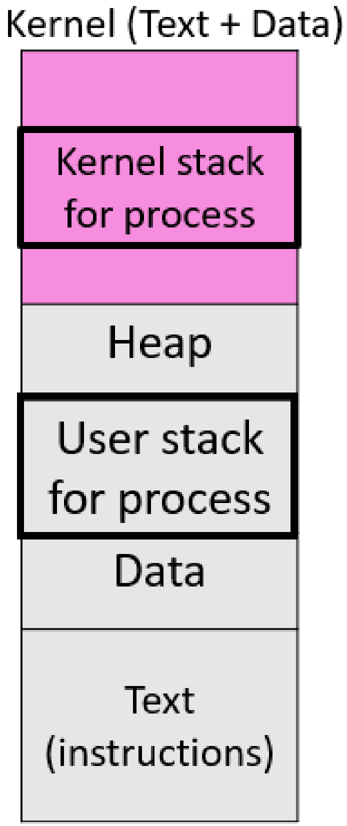
- **How to know the size of user space stack?**
  - We can change the user stack rather than kernel stack

```
# ulimit -s
8192
# ulimit -s 32768
# ulimit -s
32768
# ulimit -s unlimited
# ulimit -s
unlimited
#
```



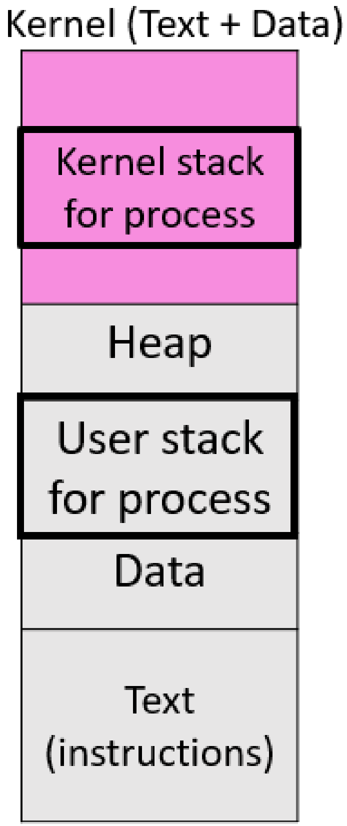
Process stacks

- **How to switch in/out kernel/process stack?**
    - A process makes a system call by execute RISC-V `ecall` instruction
        - Raise the hardware privilege level
        - Change the PC to a kernel-defined entry point
        - Switch to a kernel stack and exec. kernel instructions that implement system calls



Process stacks

- **How to switch in/out kernel/process stack?**
  - A process execute the `sret` instruction
    - Lowers the hardware privilege level
    - Resumes executing user instructions just after the system call instruction



Outline

- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

Process identifier (PID) (1/2)

- **Process identifier (PID)**
  - Each process has a unique PID
  - PIDs in Linux are of type pid_t (32-bit integer)
  - The default maximum number PIDs is 32768 (/proc/sys/kernel/pid_max)
    and you can set the value higher on 64-bit systems (up to 2^{22} = 4,194,304 (PID_MAX_LIMIT)
  - The kernel uses a bitmap to keep track of PIDs in use and assign a unique PID for new processes
  - PID eventually repeats because all the possible numbers are used up and the next PD rolls or starts over

Process identifier (PID) (2/2)

PPID stands for Parent Process ID

- **Which process is PID 0?**
  - The sched process
  - Responsible for paging and is a part of the kernel
  - The init process owns PID1 and is responsible for starting and shutting down the system

```
$ ps -eaf
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Feb25 ?        00:00:05 /sbin/init splash
root           2       0  0 Feb25 ?        00:00:00 [kthreadd]
root           3       2  0 Feb25 ?        00:00:00 [rcu_gp]
root           4       2  0 Feb25 ?        00:00:00 [rcu_par_gp]
root           9       2  0 Feb25 ?        00:00:00 [mm_percpu_wq]
root          10       2  0 Feb25 ?        00:00:00 [rcu_tasks_rude_]
```

Outline

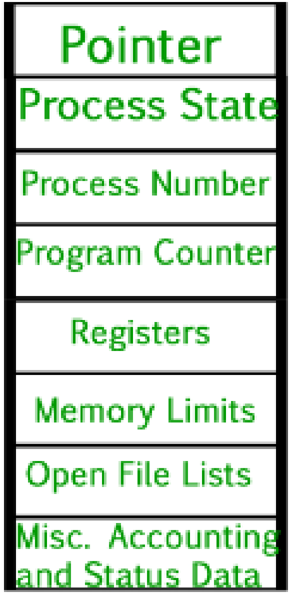
- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

Process Control Block (PCB) (1/2)

- **Process Control Block (PCB)**
  - Used to track the process’s execution status
  - Contains process state, program counter, stack pointer …
  - All this information is used when the process is switched from one state to another
- **What is the process table?**
  - <u>The process table is an array of PCBs</u>
  - Contains the information for all of the current processes in the system

Process Control Block (PCB) (2/2)

- **Process Control Block (PCB)**
  - **Pointer:** stack pointer
  - **Process state**
  - **Process number**: PID
  - **Program counter**: the address of the next instruction that is to be executed for the process
  - **Register**: store the values used when the process is scheduled to be run
  - **Memory limits**: page table, segment table
  - **Open files list**: the list of files opened for a process



Process Control Block

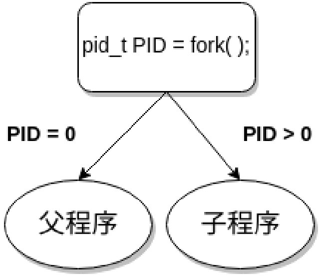
Outline

- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

Process Creation (1/4)

- **Using fork() system call**

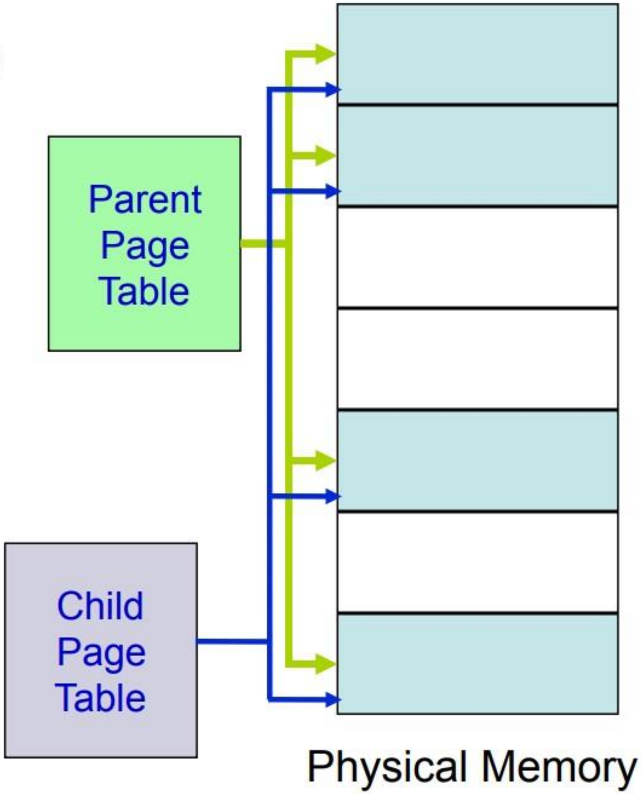
```
child process successfully created!
child_PID = 31497, parent_PID = 31496
parent process successfully created!
child_PID = 31496, parent_PID = 31491
```




```c
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main( ){
   pid_t child_pid;
   child_pid = fork (); // Create a new child process;
   if (child_pid < 0) {
      printf("fork failed");
      return 1;
   } else if (child_pid == 0) {
      printf ("child process successfully created
");
      printf ("child_PID = %d,parent_PID = %d
",
      getpid(), getppid( ) );
   } else {
      wait(NULL);
      printf ("parent process successfully created!
");
      printf ("child_PID = %d, parent_PID = %d", getpid( ), getppid( ) );
   }
   return 0;
}
```

Process Creation (2/4)

- Making a copy of a process is calling **forking**
  - Parent (is the original)
  - Child (is the new process)
  - Child is an exact copy of the parent
- **When the fork is invoked**
  - **All pages are shared between parent and child**
  - Easily done by copying the parent’s page table



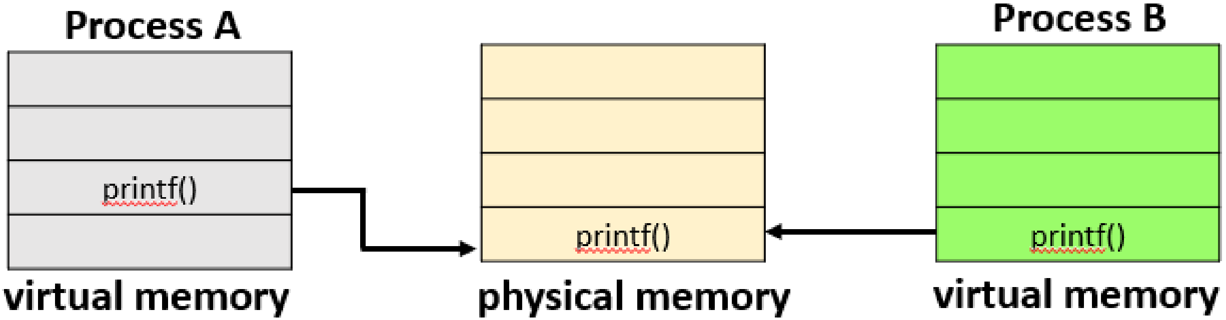
http://www.cse.iitm.ac.in/~chester/courses/16o_os/slides/5_Processes.pdf

Process Creation (3/4)

- **How can the process of cloning overhead be reduced?**
  - Copy-on-write (COW)
  - When data in any of the shared pages changes, OS intercepts and makes a copy of the page
  - Thus, parent and child will have different copies of this page
- **Why does COW work?**
  - Copying each page from parent and child would incur significant disk swapping -> huge performance penalties
  - Postpone copying of pages as much as possible

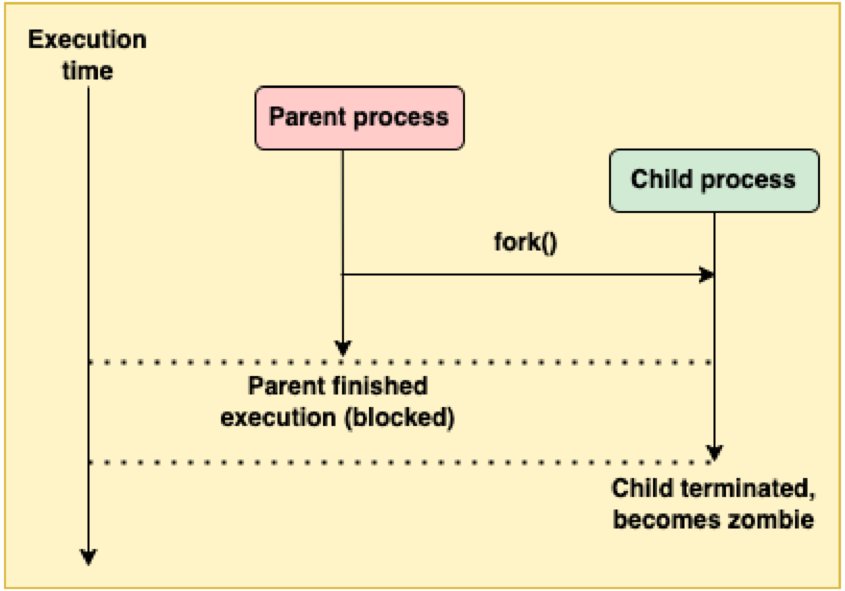
Process Creation (4/4)

- **How COW works ?**
  - When forking, the kernel makes COW pages as read-only
  - <u>Any writing to the pages would cause a page fault</u>
  - <u>The kernel detects that it is a COW page and duplicates the page</u>
  - Pages from shared libraries, shared between processes
  - E.g. printf() implements in shared libraries



Some Processes (1/4)

- **Zombie process**



Zombie process formation

1. An orphan process is formed when its parent dies while the process continues to execute
2. A zombie process is a process that has terminated but its entry is there in the system

Some Processes (2/4)

- **Zombie process**
  - A process which has finished the execution but still has entry in the process table
  - How are they formed?
    - When a parent fails to wait for its terminated child process
  - How can zombie processes be prevented in a program?
    - Ensuring the parent process waits for its child processes
    - wait() system call is used for the removal of zombie processes
    - wait() call ensures that the parent doesn't execute or sits idle till the child process is completed.

Some Processes (3/4)

- **Zombie process**
  - The child process completes through exit() system call
  - The child process issues ‘SIGCHLD’ to the parent
  - The child’s exit status is never read by the parent

```c
// A C program to demonstrate Zombie Process.

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        // Fork returns process id in parent process

    pid_t child_pid = fork();

        // Parent process
        if (child_pid > 0)
                sleep(60);

        // Child process
        else
                exit(0);

        return 0;
}
```

Some Processes (4/4)

- **Orphan process**
  - Processes that are still running even if their parent process has been terminated or finished.
  - Why do we have the orphan process?
    - Intentional orphaned: run in the background without any manual support
    - Unintentional orphaned: when the process crashes or terminates

Outline

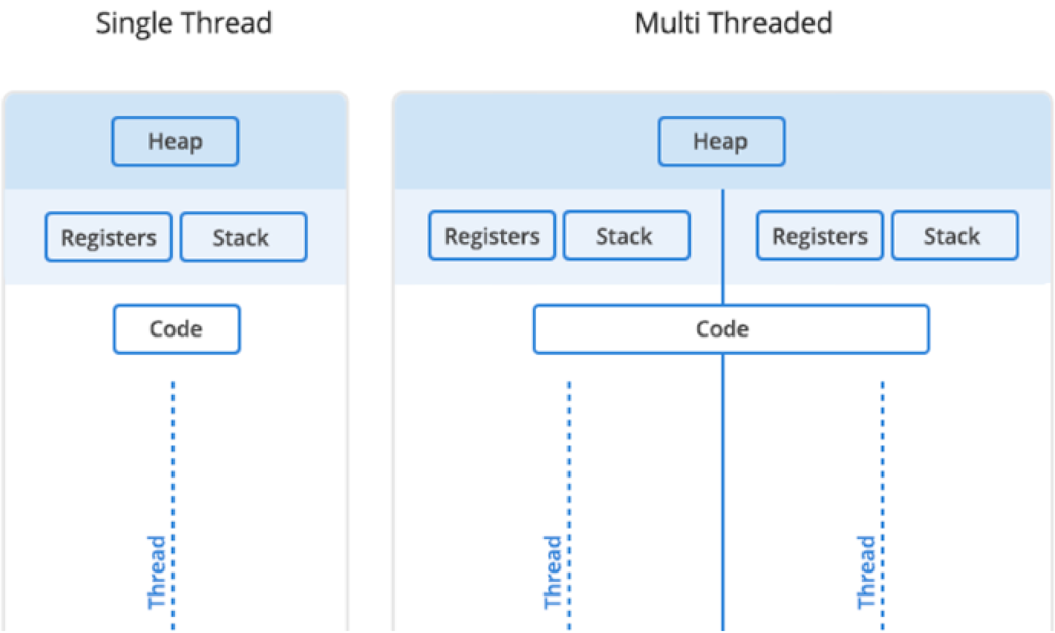
- Program vs. Process
- In-Memory Layout of a Process
- Process Stack
- Process Identifier (PID)
- Process Control Block (PCB)
- Process Creation
- Threads

Thread (1/2)

- **A thread is the unit of execution within a process**
  - Each thread has its own stack
  - All the threads in a process share the heap
  - Threads share the same address space as the process
    - easy to communicate between the threads

Thread (2/2)

- **A thread is the unit of execution within a process**



Takeaway Questions

- Which process is PID 0?
  - (A) sched
  - (B) init
  - (C) top
- A thread has its own?
  - (A) Heap
  - (B) Stack
  - (C) Register

Takeaway Questions

- What is the parent PID of a zombie process?
  - (A) 1
  - (B) 0
  - (C) Can’t be determined
- Which process is the parent of a zombie process whose parent has terminated?
  - (A) sched
  - (B) init
  - (C) top

Takeaway Questions

- **Question**: the kernel can address 1 GB of virtual addresses, translating to a maximum 1 GB of physical memory.
- **Answer:**
    - 2G/2G, 1G/3G split
    - Physical Address Extension (PAE) allows processors to access physical memory up to 64 GB

```
+----------+ 4 GB
|          |
|          |
|          |
| Kernel   |
|          |                +----------+
| Virtual  |                |          |
|          |                |          |
| Space    |                | High     |
|          |                |          |
| (1 GB)   |                | Memory   |
|          |                |          |
|          |                | (unused) |
+----------+ 3 GB           +----------+ 1 GB
|          |                |          |
|          |                |          |
|          |                |          |
|          |                | Kernel   |
|          |                |          |
|          |                | Physical |
|          |                |          |
|User-space|                | Space    |
|          |                |          |
| Virtual  |                |          |
|          |                |          |
| Space    |                |          |
|          |                |          |
| (3 GB)   |                +----------+ 0 GB
|          |
|          |                 Physical
|          |                  Memory
|          |
|          |
|          |
|          |
|          |
+----------+ 0 GB

 Virtual
 Memory
```