KOÇ UNIVERSITY

Process Management

COMP304
Operating Systems (OS)

Didem Unat

Lecture 3

Outline

- Process Concepts
- Process State
- Context Switch
- Schedulers
- Process Creation and Termination

- Reading
  - Chapter 3.1-3.4 from textbook – Very Good!
  - Linux Kernel Development – Chapter 3
  - HW #1 will be out this week
  - Requires Linux environment with a sudo access

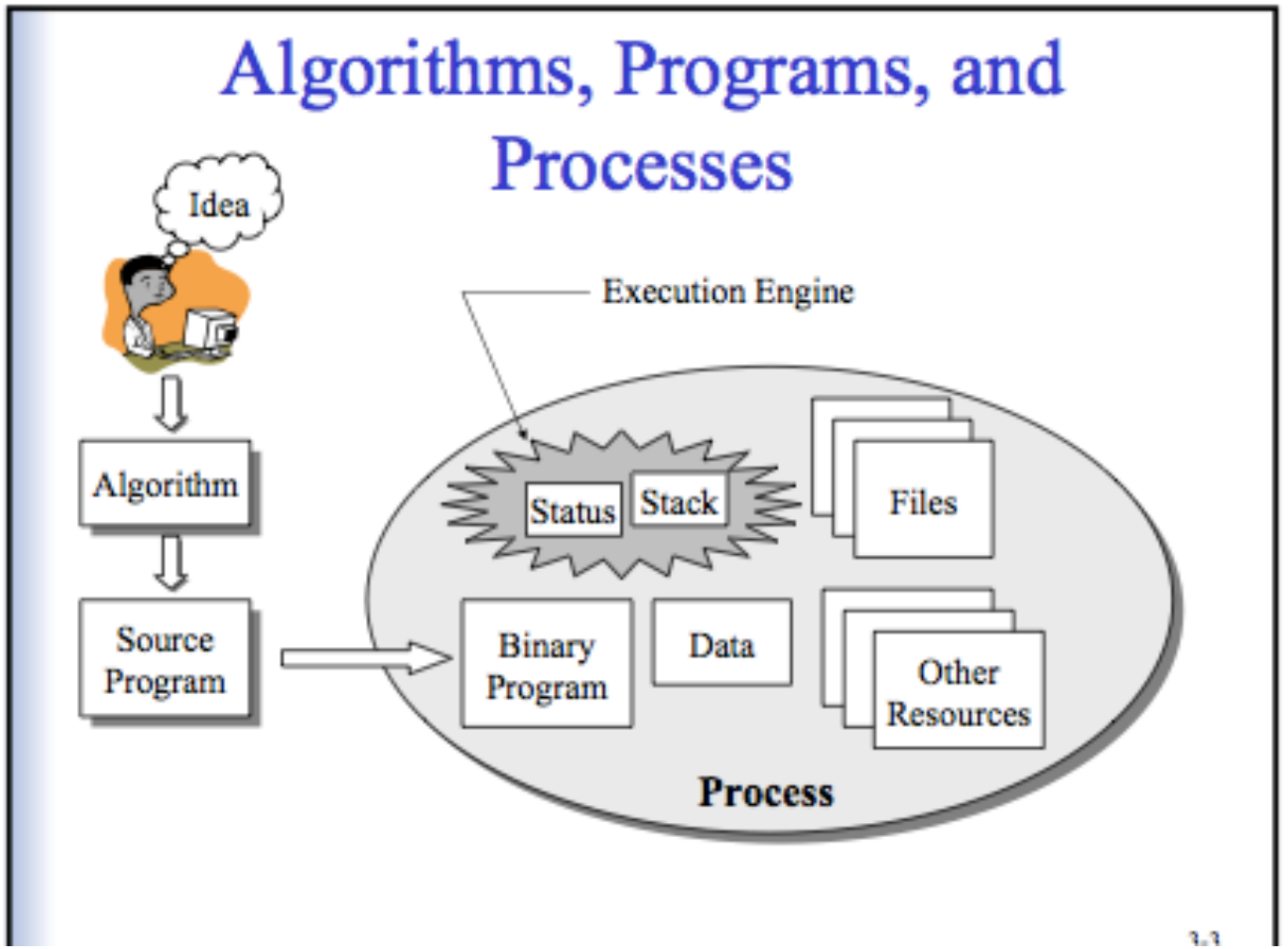
Process

- **Process** – a program in execution; process execution must progress in sequential fashion

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Terms ***job, task*** and ***process*** are almost interchangeably used

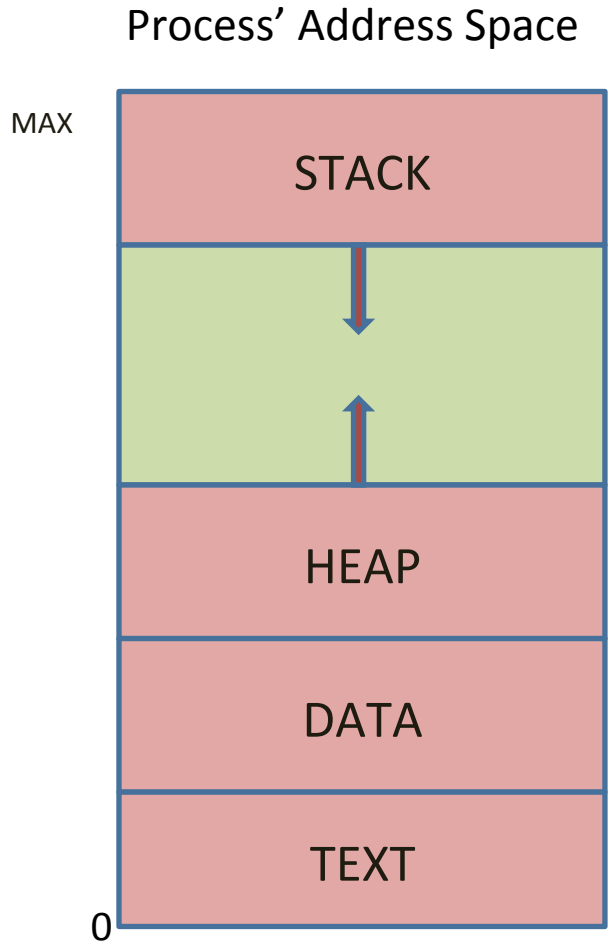
- Execution of a program starts via GUI mouse clicks, command line entry of its name, etc
- One program can have several processes
  - Consider multiple users executing the same program
  - Ex. Multiple browsers running at the same time

Algorithm, Program and Process



Process

- The program code, also called **text section**
  - **Binary code**
- Current activity includes **program counter** and other processor **registers**
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



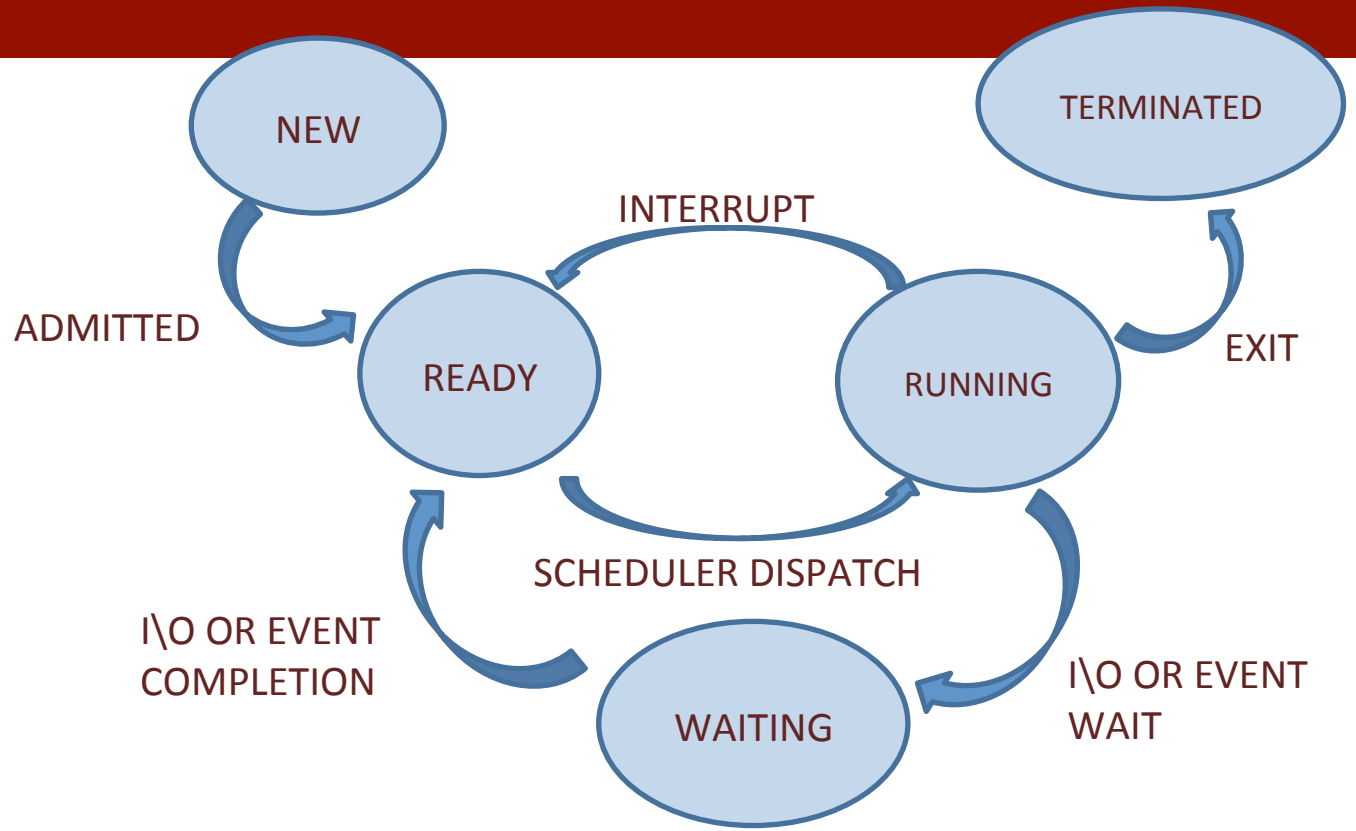
Concurrent Execution

- OS implements an abstract machine per process
- *Multiprogramming* enables
  - *N* programs to be *space-muxed* in executable memory, and *time-muxed* across the physical machine processor.
- Result: Have an environment in which there can be multiple programs in execution *concurrently*,* each as a process
  - Concurrently means processes appear to execute simultaneously, they all make some progress over time

Process State

- As a process executes, it changes its **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a CPU
  - **terminated**: The process has finished execution

Transition between Process States



- Process transitions from one state to another
- An animation for process states
  - http://williamstallings.com/OS/Animation/Queensland/PROCESS.SWF

Process Context

- Also called process control block

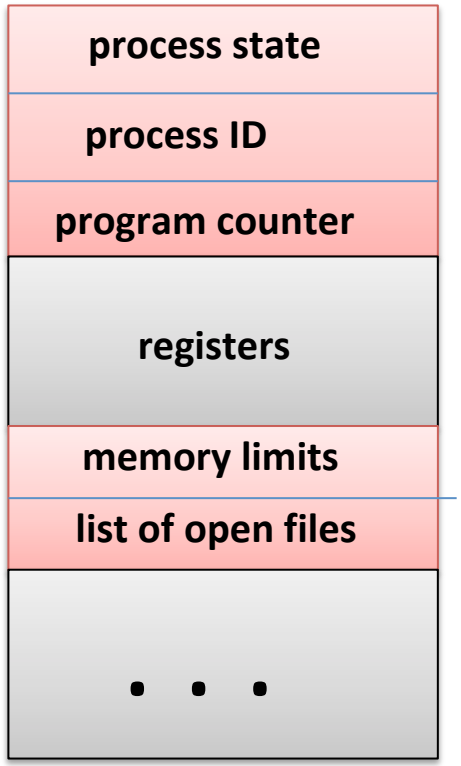
- When an interrupt occurs, what information OS needs to keep around so that we can reconstruct process's context as if it was never interrupted its execution?

Process Control Block (PCB)

Keeps the process context

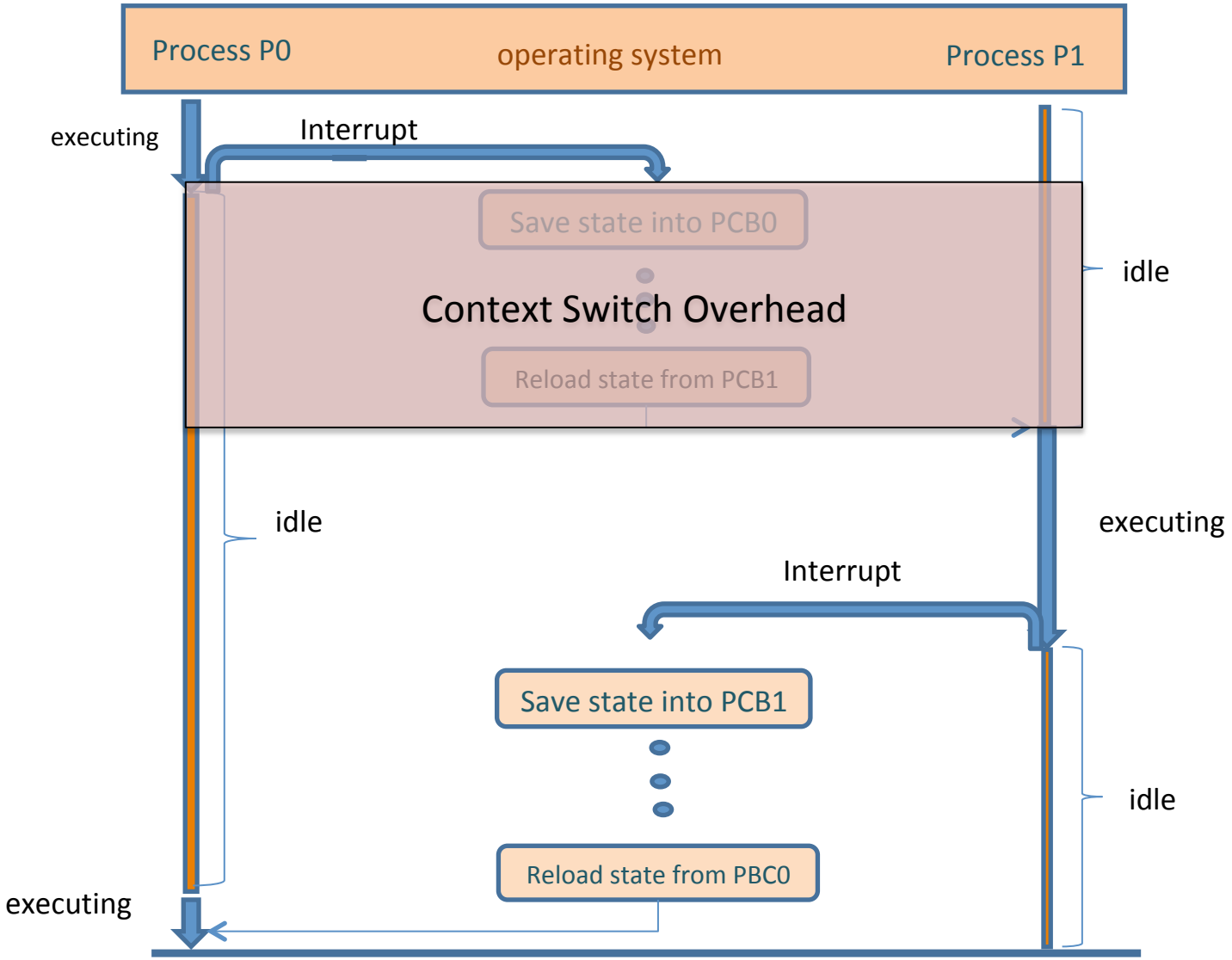
- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process registers
- **CPU scheduling information**- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

Metadata about a process



Context Switch

- OS needs to store and restore the context of a process
  - So that execution of the process can be resumed from the same point at a later time
  - This is called **context switch**
- Context switch is **pure overhead**
  - Why?
  - Should be very small (couple millisecond)
  - Hardware support for context switching improves the performance
- When does OS switch context?
  - In case of an interrupt
  - When process's time is up
    - Even though process has still some work to do
  - When a process terminates



- Switching between threads of a single process can be faster than between two separate processes

Threads

- A process has at least one thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
  - Word document
    - Spell checker – 1 thread
    - Typing text – 1 thread
- Must then have storage for thread details, multiple program counters in PCB or each thread has a PCB
- More on threads later (in Chapter 4)

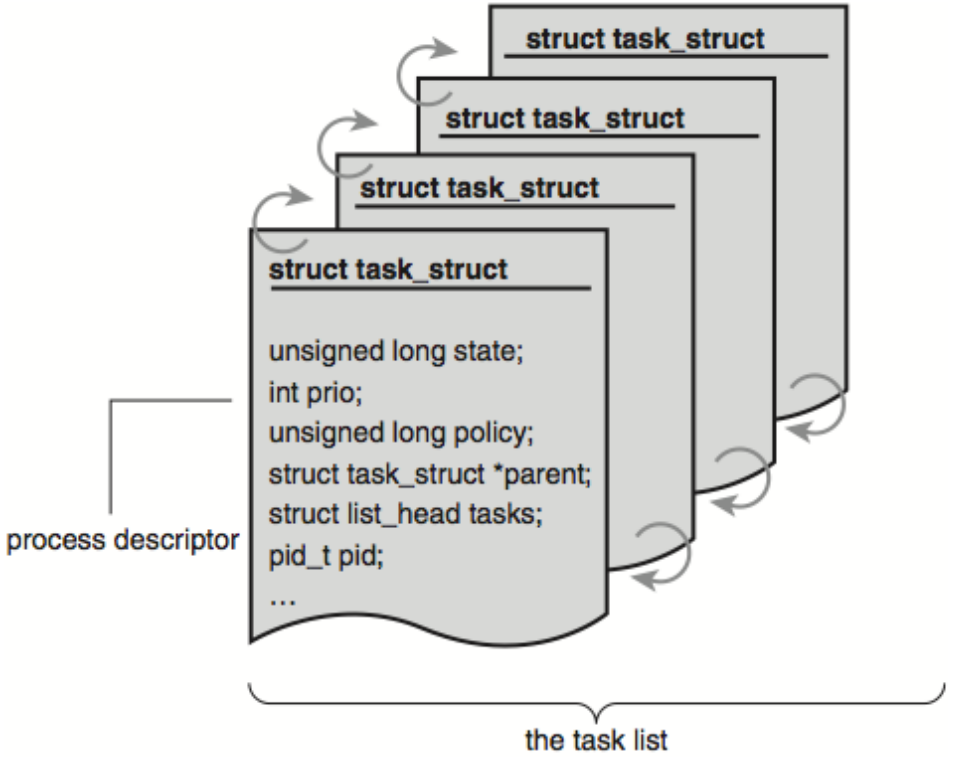
Unix Processes

- Each process has its own address space
  - Subdivided into text, data, & stack segment – a.out file describes the address space
- OS kernel creates *descriptor (PCB)* to manage process
- *Process identifier* (PID): User handle for the process (descriptor)

task_struct

- Represented by the C structure

```
task_struct {
    pid_t pid;
    /* process identifier */
    long state;
    /* state of the process */
    unsigned int time_slice
    /* scheduling information */
    struct task_struct *parent;
    /* this process's parent */
    struct list_head children;
    /* this process's children */
    struct files_struct *files;
    /* list of open files */
    struct mm_struct *mm;
    /* address space of this process */
   ...
```



Search for task_struct (line ~1200)
http://lxr.free-electrons.com/source/include/linux/sched.h

‘top’ and ‘ps’ commands

- top: Displays processor activity of POSIX-based OS and also displays tasks managed by kernel in real-time.
- ps: snapshot of process states

1. top

```
Processes: 189 total, 4 running, 6 stuck, 179 sleeping, 871 threads                 12:11:33
Load Avg: 1.11, 1.38, 1.48  CPU usage: 5.87% user, 6.88% sys, 87.24% idle
SharedLibs: 1080K resident, 0B data, 0B linkedit.
MemRegions: 137027 total, 2070M resident, 50M private, 424M shared.
PhysMem: 4299M used (1039M wired), 3340M unused.
VM: 448G vsize, 1025M framework vsize, 7240589(0) swapins, 7786978(0) swapouts.
Networks: packets: 13504407/14G in, 10367858/3318M out.
Disks: 2539387/63G read, 3710768/124G written.

PID    COMMAND      %CPU      TIME     #TH   #WQ   #PORT #MREGS MEM    RPRVT  PURG   CMPRS VPRVT
61818  screencaptur 0.4       00:00.09 4    2    50    88     1860K  808K   4096B  0B    30M
61815  top          18.9      00:05.36 1/1  0    23    37     3568K  3336K  0B     0B    44M
61813  ReportCrash  0.0       00:00.29 6    3    54    69     5040K  4372K  0B     0B    68M
61800  mdworker     0.0       00:00.19 4    0    54    70     7696K  6800K  0B     0B    66M
61790  mdworker     0.0       00:00.21 4    0    54    68     8428K  7512K  0B     0B    67M
61775  mdworker     0.0       00:00.08 4    0    51    58     8272K  7460K  0B     0B    60M
61759  ocspd        0.0       00:00.01 1    0    20    33     1016K  692K   0B     0B    36M
61677  Image Captur 0.0       00:00.21 3    0    150   83     4380K  3088K  0B     0B    58M
```

Creating/Destroying Processes

- UNIX `fork()` creates a process
  – Creates a new address space
  – Copies text, data, & stack into new address space
  – Provides child with access to open files of its parent

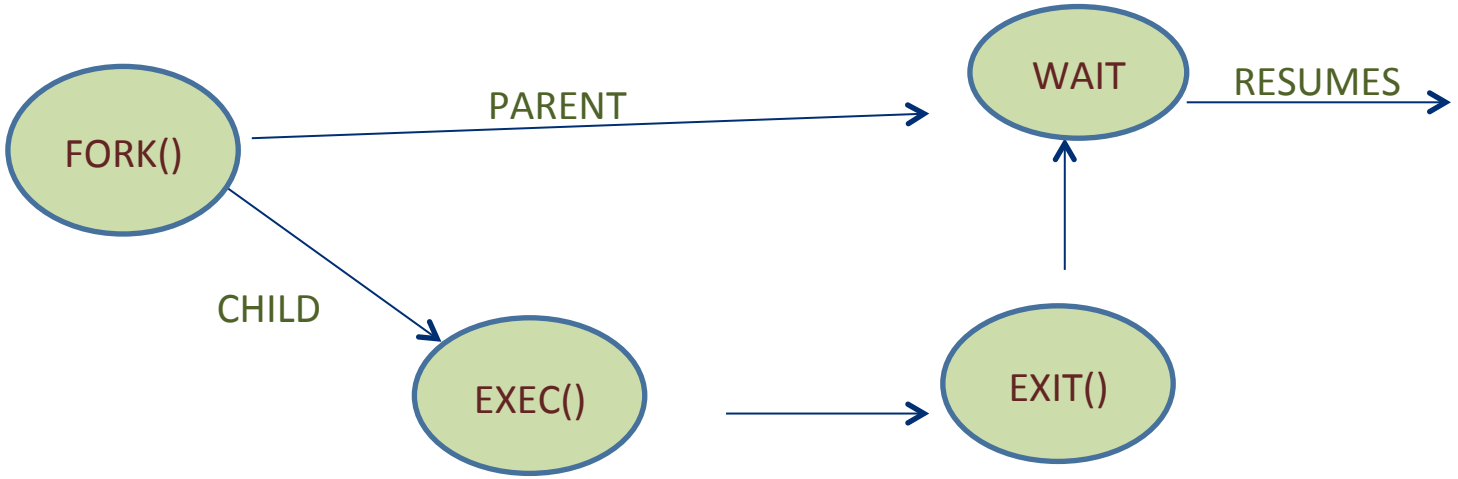
- UNIX `wait()` allows a parent to wait for a child to change its state
  – This is a blocking call, parent waits until it receives a signal
  – http://linux.die.net/man/2/wait

- UNIX `exec()` system call variants (e.g.execve()) allow a child to run a new program

Process Creation

- Address space
  - Child duplicates the address space of the parent
  - Child has a program loaded into it

- UNIX examples
  - **fork()** system call creates a new process
  - **exec()** system call is used after a **fork()** to replace the process' memory space with a new program



Creating a UNIX process

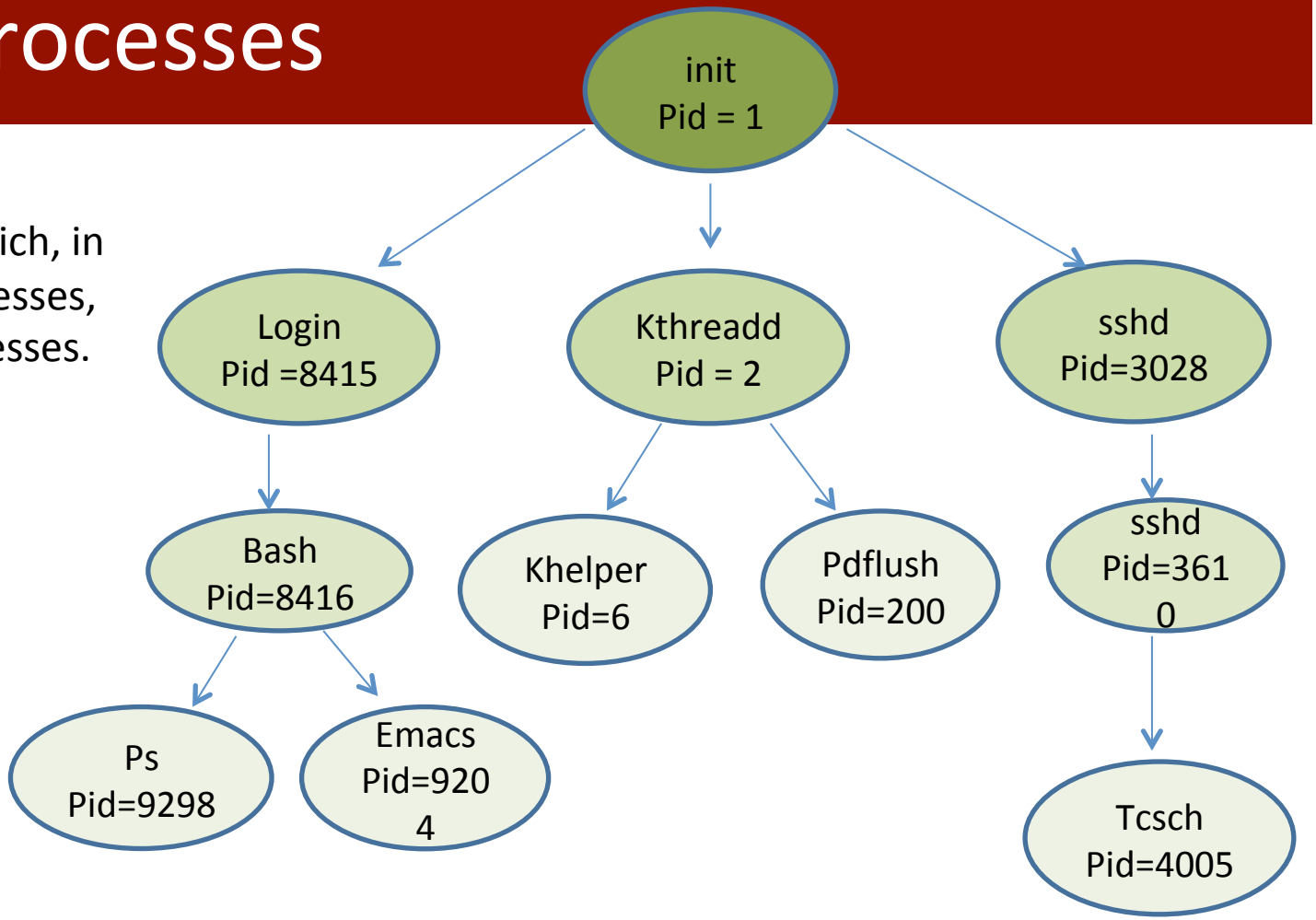
```
int value, mypid=-1;
 ...
value = fork(); /* Creates a child process */

/* value is 0 for child, nonzero for parent */
if(value == 0) {
     /* The child executes this code concurrently with parent */
     mypid = getpid();
     printf("Child's Process ID: %d\n", mypid);
     exit(0);
}
/* The parent executes this code concurrently with child */

parentWorks(..);
wait(…);
 ...
```

Child Executing a Different Program

```
int mypid;
 ...
/* Set up the argv array for the child */
 ...
/* Create the child */
if((mypid = fork()) == 0) {
  /* The child executes its own absolute program */
    execve("childProgram.out", argv, 0);
  /* Only return from an execve call if it fails */
    printf("Error in the exec … terminating the child …");
    exit(0);
}
 ...
wait(…);     /* Parent waits for child to terminate */
 ...
```

Tree of Processes

Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

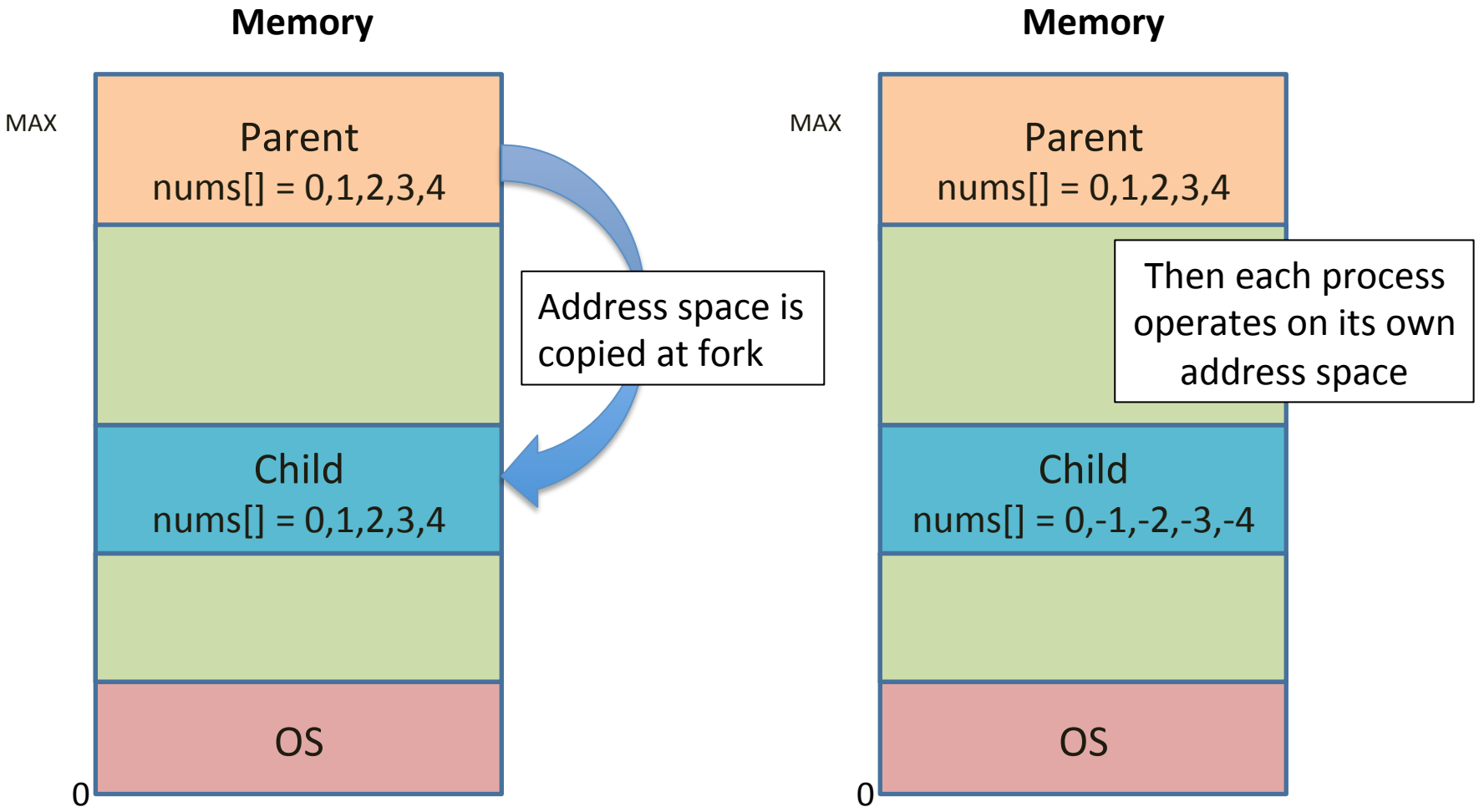


- init is very first process (pid =1)
- kthread is for system processes (pid=2)
- login process is for users directly logged in to the system
- sshd process is for users remotely logged in to the system
  - Starts an openSSH SSH daemon

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
  int i;
  pid_t pid;
  pid = fork();

  if (pid == 0) {
    for (i = 0; i < SIZE; i++) {
      nums[i] = -i;
      printf("CHILD: %d \n",nums[i]); /* LINE X */
    }
  }
  else if (pid > 0) {
    wait(NULL);
    for (i = 0; i < SIZE; i++)
      printf("PARENT: %d \n",nums[i]); /* LINE Y */
  }
  return 0;
}
```

What output will be at Line X and Line Y?

Address Spaces



Reading

- From text book
  - Read Chapter 3.1-3.3
  - Linux Kernel Development (Chapter 3)

- Acknowledgments
  - These slides are adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9th edition) Wiley