KOÇ UNIVERSITY

Process Creation/Termination

COMP304

Operating Systems (OS)

Didem Unat

Lecture 4

Outline

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

- Announcements
  - HW #1 will be out today
  - Monday, there will be a PS on HW1

Process

- **Process** – a program in execution;

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

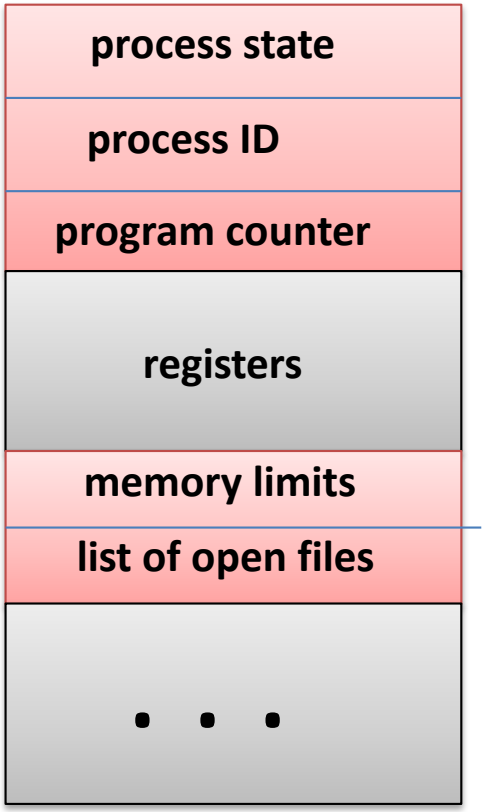
- Execution of a program starts via GUI mouse clicks, command line entry of its name, etc
- One program can have several processes
  - Consider multiple users executing the same program
  - Ex. Multiple browsers running at the same time

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

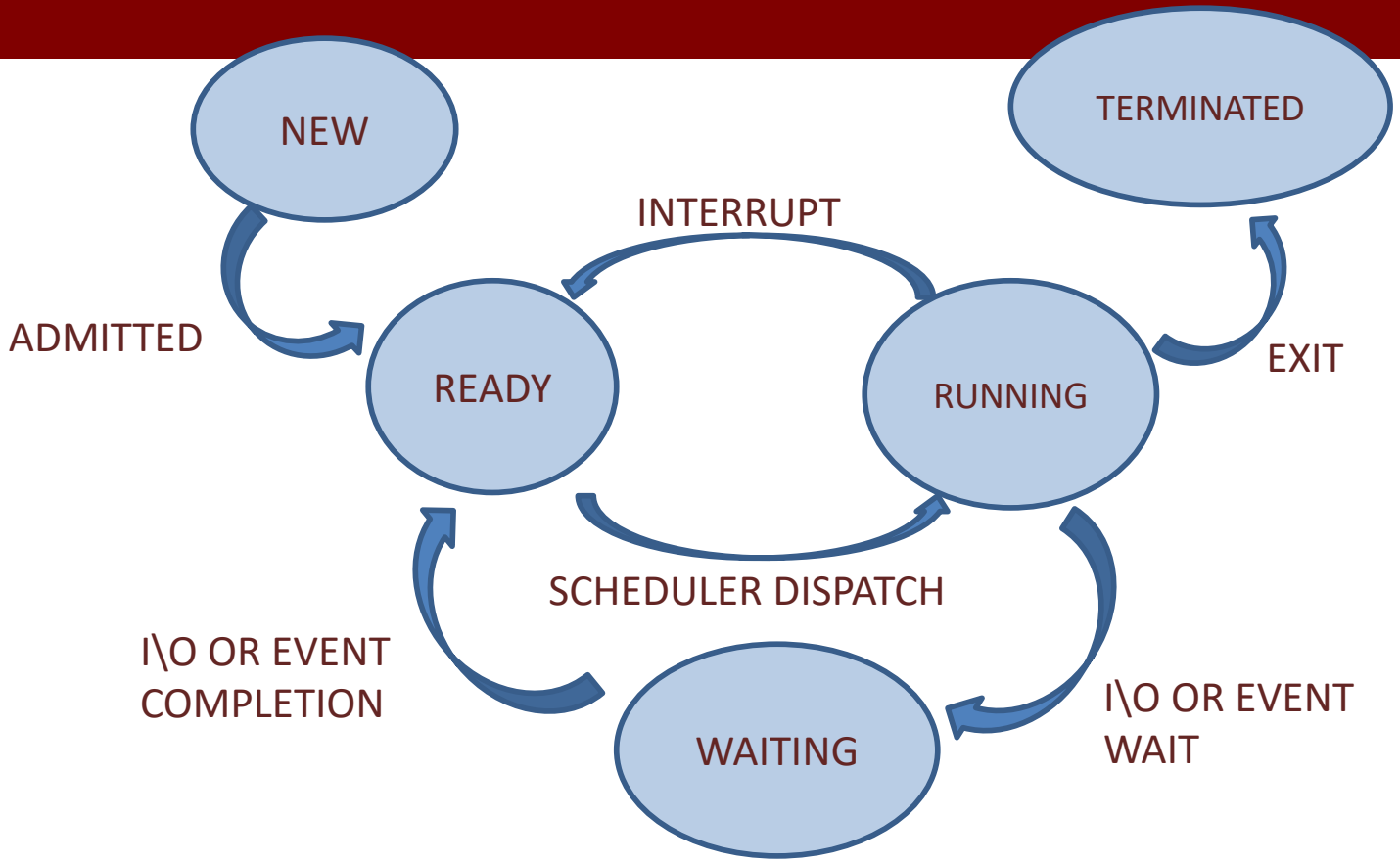
Metadata about a process



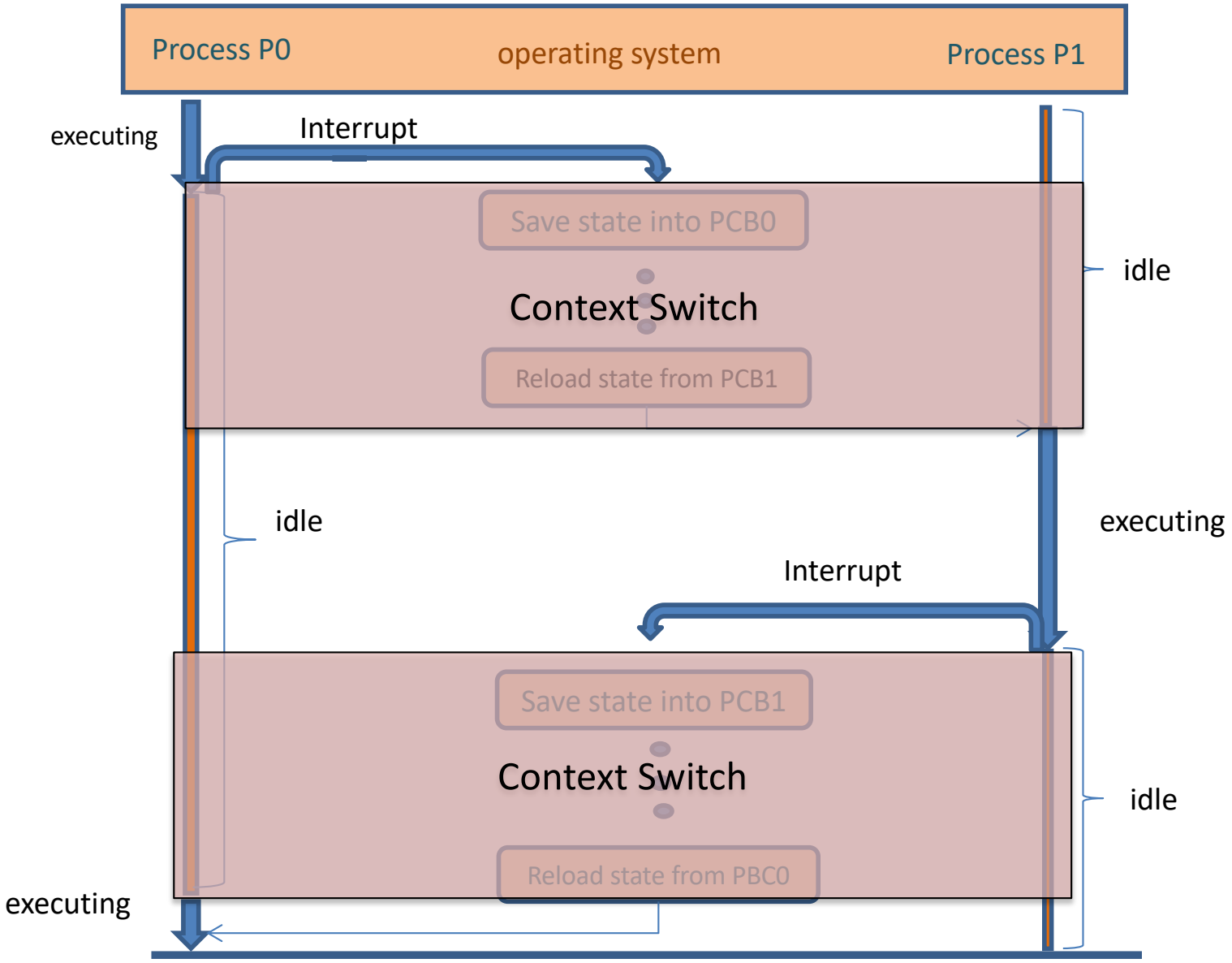
Process State

- 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 (e.g., IO) to occur
  - **ready**: The process is waiting to be assigned to a CPU
  - **terminated**: The process has finished execution

Transition between Process States



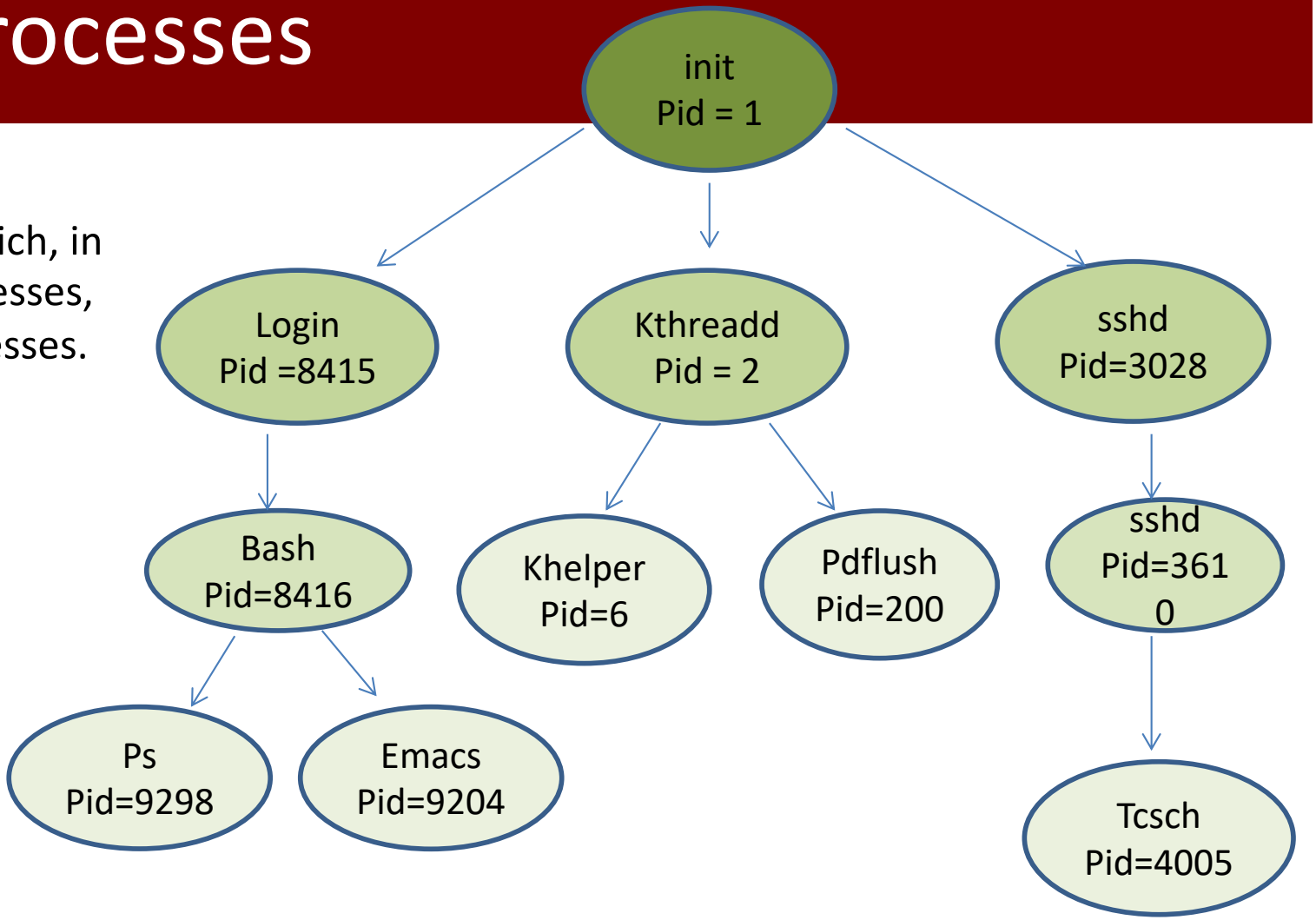
- Process transitions from one state to another
- An animation for process states
  - https://www.youtube.com/watch?v=Y3mQYaQsrvg



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

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

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

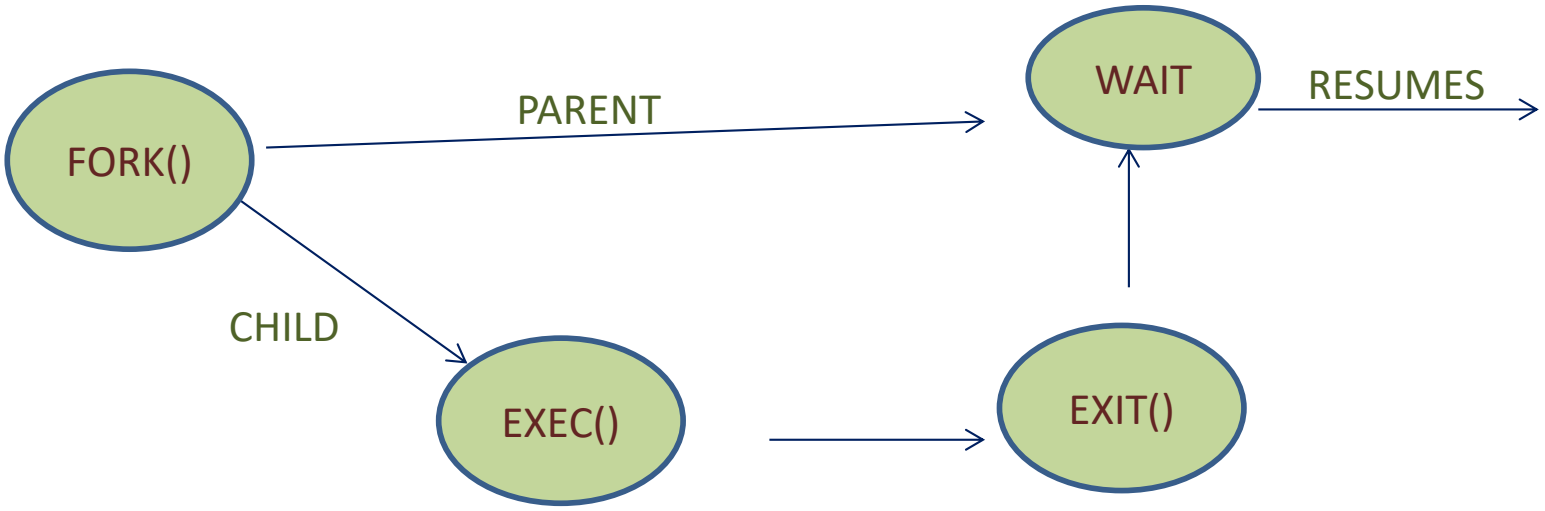
- UNIX `exec()` system call variants (e.g.execve()) allow a child to run a new program

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(…);
  ...
```

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



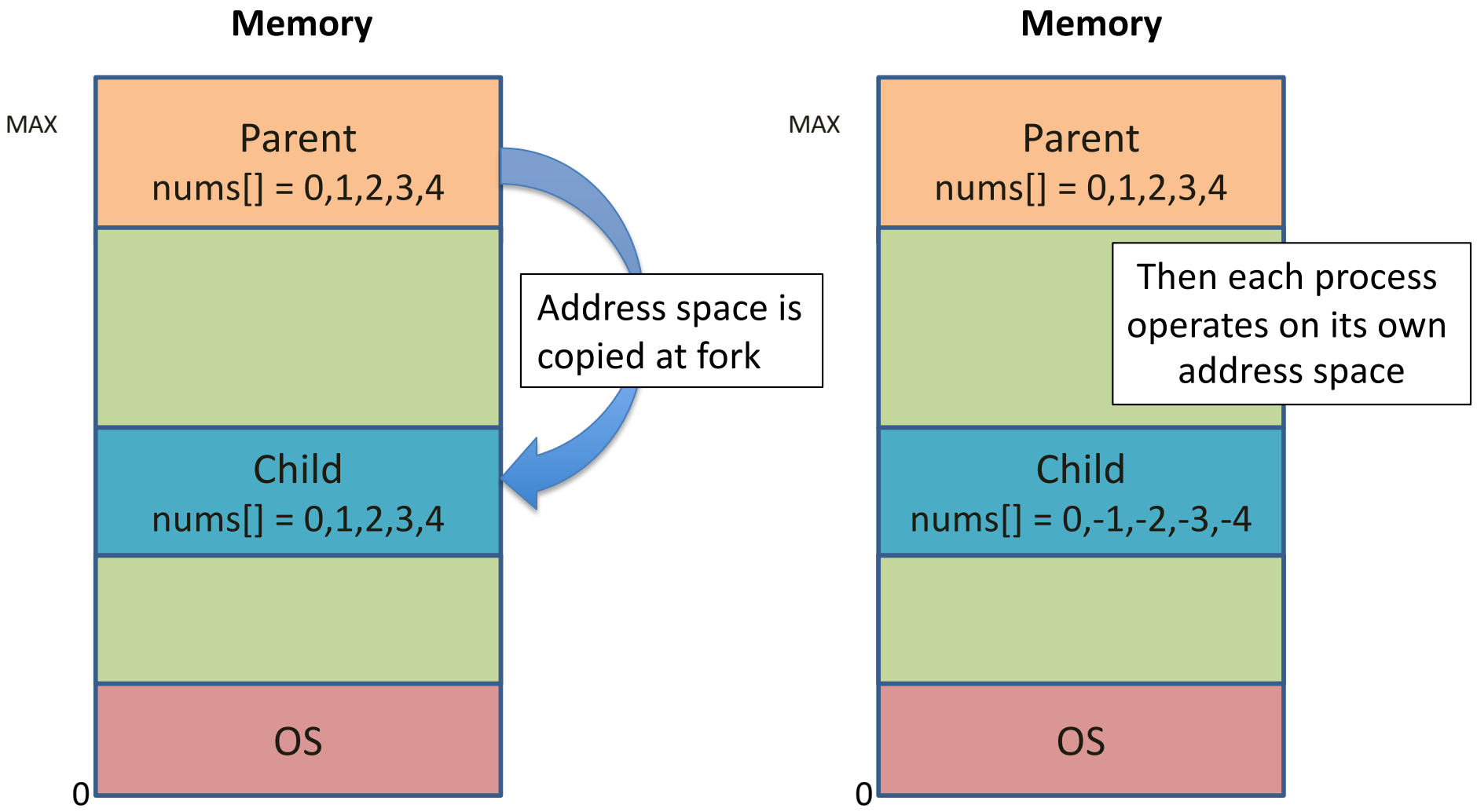
Child Executing a Different Program

```
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 */
  ...
```

```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



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit()**)
  - Output data from child to parent (via **wait()**)
  - Terminated process' resources are deallocated by operating system

- Parent may terminate execution of children processes
  - Via **kill()** system call

- A terminated process is a **zombie,** until its parent calls wait()
  - Still has an entry in the process table

Zombie

- A **zombie process** or **defunct process** is a process that has completed execution (via the exit system call)

- It still has an entry in the process table: it is a process in the "Terminated state".

- This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status:
  - Once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped"

What happens if the parent dies before the child?

Orphans

- Some operating systems do not allow child to continue without its parent
  - All children are terminated - **cascading termination**

- More common: If parent terminates, still executing children processes are called **orphans**
  - Those are adopted by *init* process
- **Init** periodically calls wait to terminate orphans

Question

```
#include <stdio.h>

int main()
{
        printf("Print 1: %d\n",getpid());

        fork();  /*fork 1*/
        printf("Print 2: %d\n",getpid());


        fork(); /*fork 2*/
        printf("Print 3: %d\n",getpid());

        fork(); /*fork 3*/
        printf("Print 4: %d\n",getpid());

        return 0;
}
```

How many processes are there?

How many prints are called?

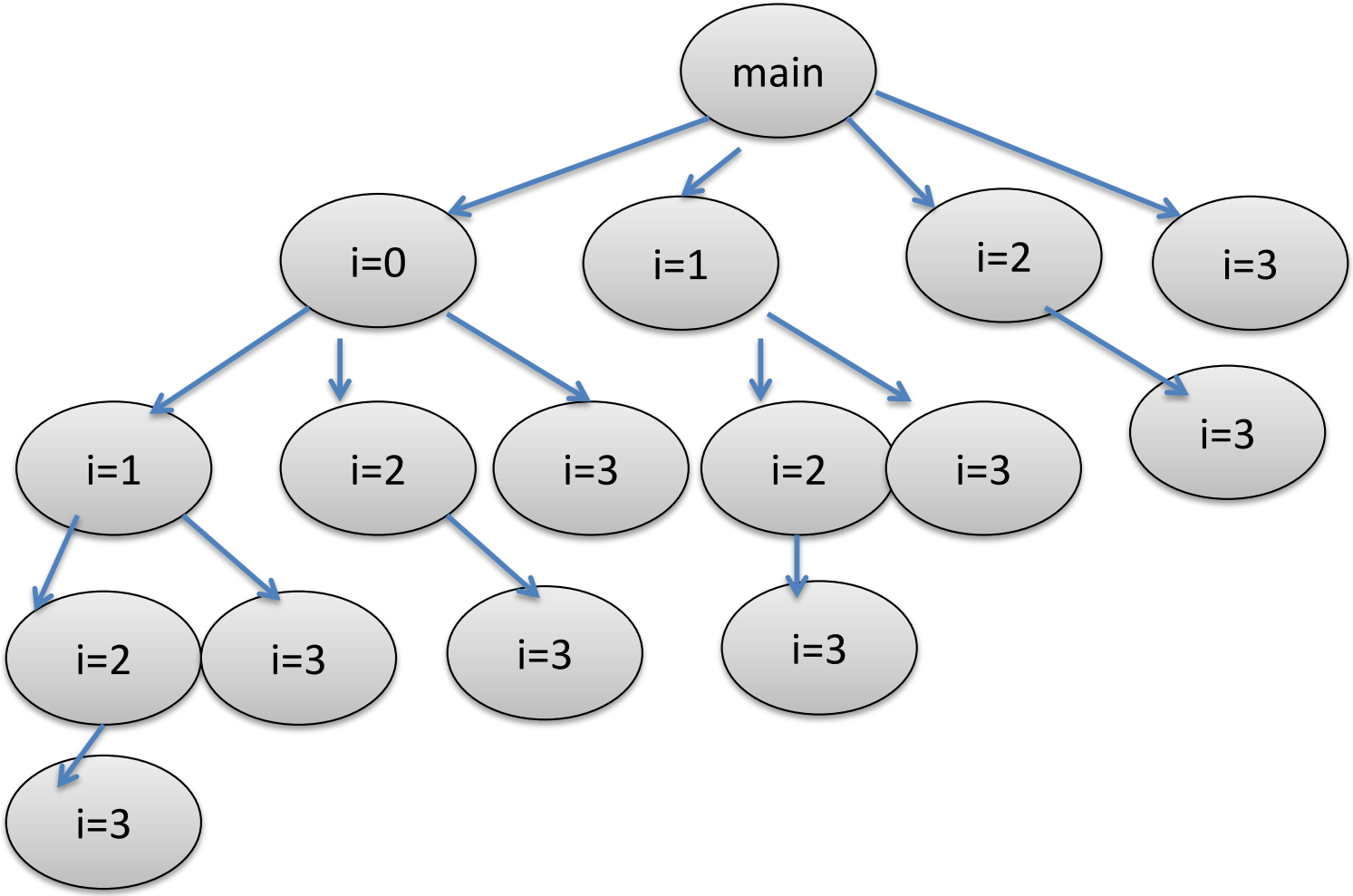
Question

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;
    for(i=0; i < 4 ; i++)
        fork();

    printf("PID %d\n", getpid());
    return 0;
}
```

Including the initial parent process, How many processes are created?

Draw a process tree starting from the initial parent process as the root!

Process Tree for the Question



Reading

- From text book
  - Read Chapter 3.1-3.4
  - Linux Kernel Development (Chapter 3)

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