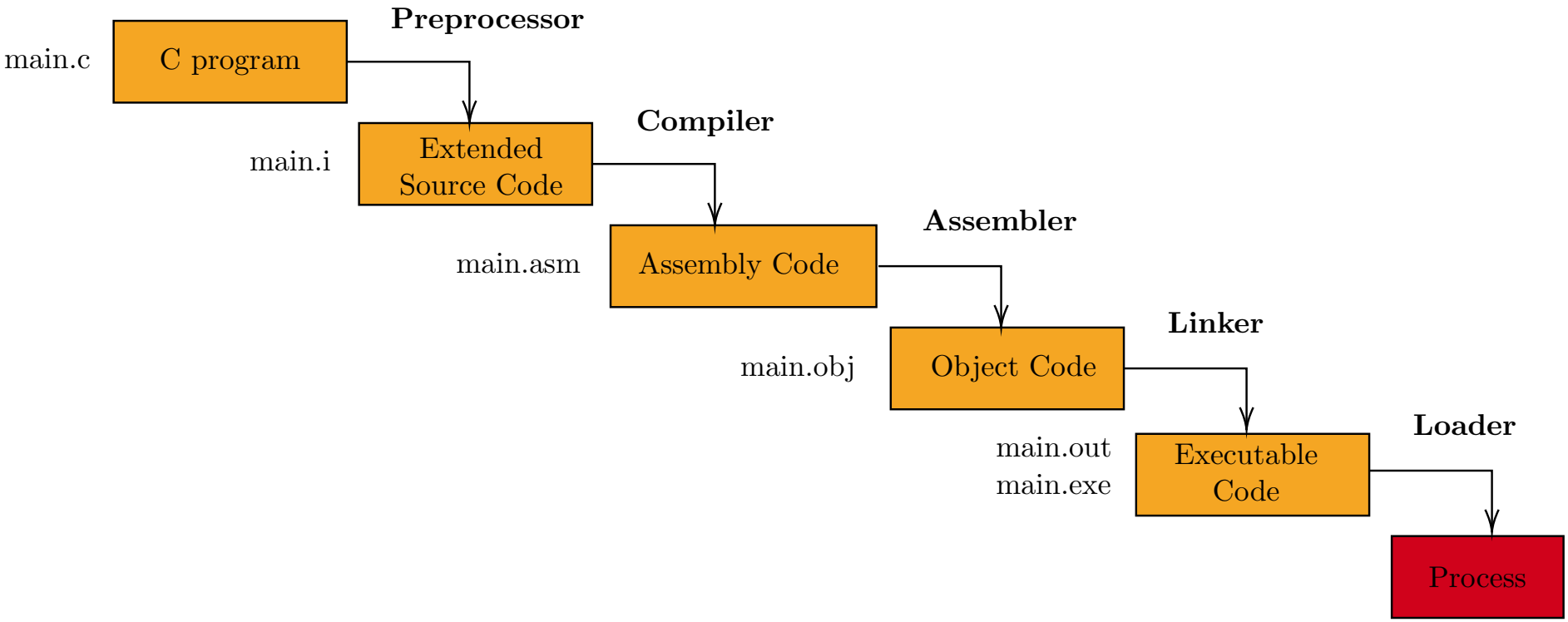
Lab 4 (CS): Processes & Threads

Innopolis University
Course of Operating Systems

Lab Objectives

- Review some OS concepts.
- Review the states of OS process.
- Recognize the zombie from orphan processes.
- Learn to manage multiple processes in C.
  - Inter-process communication will be covered next week.
- Learn to run jobs by multiple processes in the background.

Execution flow of C program



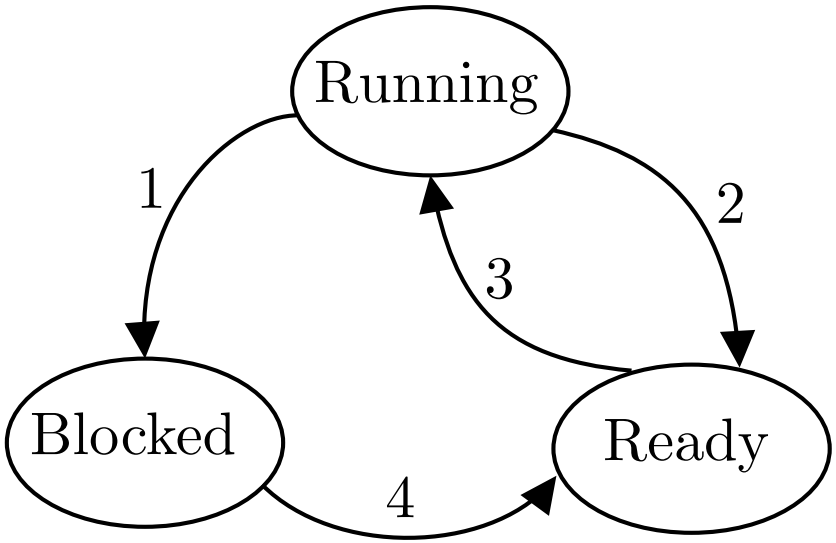
Program vs. Process

- **Program:** is a sequence or set of instructions in a programming language for a computer to execute.
- **Process:** is a program in execution.
- **Multiprogramming:** is the ability of the OS to load multiple processes into the memory which are available to run simultaneously on a **single** CPU.
- **Multiprocessing:** is the ability of the OS to load multiple processes into the memory and to allow them to be executed simultaneously on **multiple** CPUs.
- **System call:** The way in which the program enters the OS kernel to perform some task. The manual pages of all system calls can be accessed as (**man 2** ***system_call_name***).

Process States (From Lecture 04)

The three main states of the process.

- Running (actually using the CPU at that instant).
- Ready (runnable; temporarily stopped to let another process run).
- Blocked (unable to run until some external event happens).



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks the process
4. Input becomes available

From the textbook – Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Process Creation and Termination

- **Process creation**
  - Running a program.
  - **fork:** creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process. It is declared as: (Use *man 2 fork*)

```
#include <unistd.h>
pid_t fork(void);
```

- **Process termination**
  - The process is terminated, usually due to one of the following conditions:
    - Normal exit (voluntary).(Successful termination by exit system call)
    - Error exit (voluntary).(Unsuccessful termination by exit system call)
    - Fatal error (involuntary). (Errors or bugs)
    - Killed by another process (involuntary). (kill system call)
  - **exit:** causes process termination. The C standard specifies two constants, **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to indicate successful or unsuccessful termination, respectively. (Use *man 2 exit*)

Process Creation and Termination

Example 1:

- Write a program that declares a variable n=4, creates a new process and prints "Hello from `[PID - <n> - <&n>]`" from both processes respectively. What is the relationship between the processes? How to differentiate the parent from the child process?
- Terminate both processes. Terminate one of them as a normal exit and the other as an error exit.
- Run the program and modify the code to let the parent process kill the child process using the function *kill* with the signal **SIGKILL**. (Optional)
  - We won't cover signals in this lab. (For more info on Signals)
- What is the maximum value for process ID that you can create in your operating system? Can you specify the file path which stores this value? Which process has no parents like root (/) directory in case of files? What is the value of its ppid?
-

Some Process Management Utilities

Some useful utilities for sleeping, waiting and pausing processes.

- **sleep:** a "system call" used to sleep the calling thread (process) for a specific number of seconds passed as an argument.
- **wait:** a system call used to wait for state changes in one of the children of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. For instance, to let the parent wait for its n children, you need to call wait function n times and pass NULL to this function if you are interested in the state change information.
- **pause:** a system call which causes the calling process (or thread) to sleep until a signal is delivered.
  - Signals won't be covered in this lab.

Zombie and Orphan processes

Example 2:

- Write a program that creates a new process and prints the PID.
- Let the parent process sleep for a couple of seconds using the function *sleep*. Use *man 3 sleep* to see the documentation.
- Kill the child process immediately. Will the child be terminated properly? What will happen to the child process and what is the name of the process? Show the zombie process using *top* command. Explain the columns PID (process ID) and S (process state) in *top* command. Run the command `(man top | grep "Status" -A 12 -m 1)` to get more info.
- In contrast, create an orphan process by sleeping the child process and terminating the parent process. Who will be the new parent of the child process?
- How can we avoid getting an orphan process? Use **wait** system call to let the parent wait for its child.

Text file I/O in C

This slide is dedicated to give you an example on reading/writing from/to text files in C.

```c
/* read from a text file example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  // modes: r - read, a - append
  pFile = fopen("myfile.txt","r");
  char line[10];
  if (pFile!=NULL)
  {
    fscanf(pFile,"%s[^\n]", line);
    fclose(pFile);
  }
  printf("%s", line);
  return 0;
}
```

```c
/* write to a text file example */
#include <stdio.h>
int main ()
{
  FILE * pFile;
  // modes: w - write
  pFile = fopen("myfile.txt","w");
  if (pFile!=NULL)
  {
    fputs("new_line",pFile);
    fclose(pFile);
  }
  return 0;
}
```

Source: https://cplusplus.com/reference/cstdio/

Example 3 (1/2)

- Write a program *ex2.c* creates two vectors u and v using arrays of *120* elements. Set random values to the vectors from the range *[0-99]* using function rand.
- Create an array of n processes where n is an input number from stdin.
- Calculate the dot product of the vectors by the processes. Distribute the calculation among the processes **equally**. For instance, if $n = 6$ then each process will contribute in calculating only 20 components of the result vector. The final aggregation result should be printed by only the "main" process.
- Your program should open a file *temp.txt* for reading and writing computation results. This file will be the shared medium among the processes.

Example 3 (2/2)

- Each child process should write the result of its calculation to the shared file.
- The "main" process will read the calculation results from the file and aggregate them.
- All processes have access to both vectors u, v and the shared file.
- **Hint:** The dot product of u and v is : $u * v = \sum_{i,j} u_i * v_j$
- **Hint:** Make sure that the parent process waits for all its children.
- We did not determine a specific format for writing the computation results in the file. You may write each process result in a new line or in a one line separated with commas...etc
- **Note:** We assume that $n = 2k$ where $k \in \{1, 2, 3, 4, 5\}$.

Exercise 1

- Write a program *ex1.c* which creates two new processes, any of them should not be a child of the other (their parents should be the "main" process).
- Print the ID of each three processes created in the program and their parent IDs. Print the execution time for each process in milliseconds. The execution time starts from the first instruction after *fork*.
- **Hint:** Use the library **time.h** and function **clock**. Check the example https://cplusplus.com/reference/ctime/clock/.
- **Hint:** Use the library **unistd.h** for process creation.

Exercise 2

In this exercise, you will do an experiment, observe and record the results.

- Write a program *ex3.c* that calls fork() in a loop for n times and sleeps for 5 seconds after each call. Run the program as a background process and then run **pstree** command several times. The variable n should be read from the **command line** as an argument and not from stdin.
- Run the program for n=3. Look at the output and tell how many processes are created. Explain the result.
- Run the program so that it would call fork() 5 times. See how the result changes.
- Explain the results in each run and the difference between results.

Exercise 3

- Write a program *ex4.c* to create your own simplistic shell. It should read user input and be able to run a command without arguments, such as pwd, ls, top, pstree and so on.
- Extend your previous code to handle commands with arguments and run the processes in background (another process). Do not use here the shell operator & for running the process in the background.
- **Note:** Here, it is not permitted to use the library function *system* but you should use one of the system calls for executing the commands such as *execve*.
- **Hint:** use man fork and man execve.

End of lab 4