Lecture 18

Introduction to Process Management

Shibo Li
shiboli@cs.fsu.edu

Department of Computer Science
Florida State University

The slides are mainly from Sharanya Jayaraman

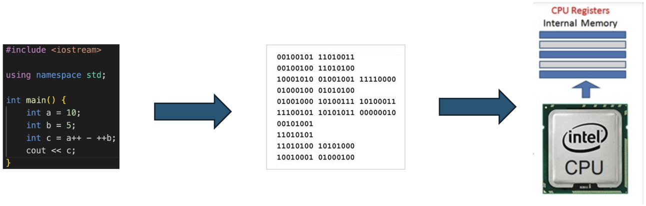
Multi-Tasking

- The UNIX Operating System provides an environment in which multiple “tasks” can run concurrently
  - The ability to run multiple programs on the same machine concurrently
  - Also used to indicate that multiple concurrent processes can execute at the same time in a single processor environment
- UNIX supports multi-tasking via
  - Process abstraction

Life-Cycle of A Program

Program vs. Prcoess

- **Program:** This is a static set of instructions written in a language like C++ that is stored on disk.
- **Process:** Once a program is loaded and executed, it becomes a process. A process is an active instance of a program that runs in memory



```
#include <iostream>

using namespace std;

int main() {
    int a = 10;
    int b = 5;
    int c = a++ - ++b;
    cout << c;
}
```

```
00100101 11010011
00100100 11010100
10001010 01001001 11110000
01000100 01010100
01001000 10100111 10100011
11100101 10101011 00000010
00101001
11010101
11010100 10101000
10010001 01000100
```

The Process Abstraction

- In the traditional systems a process executes a single sequence of instructions in an address space.
  - The program counter (PC) is a special hardware register thattracks the current instruction that is to be execute
  - In UNIX, many processes are active at the same time and the OS provides some aspects of a virtual machine
  - Processes have their own registers and memory, but rely on the OS for I/O, device control and interacting with other processes

The Process Abstraction

- Process abstraction is a key concept in operating systems that simplifies complex tasks by allowing us to view a running program as an independent, manageable entity — a process.
  - Run in a virtual address space
  - Content for resources such as processor(s), memory, andperipheral devices
  - All of the above is managed by the OS the memory management system; the I/O system; the process management and scheduling system, and the Interprocess Communication system (IPC)

The Process Abstraction

Overall, process abstraction allows developers to focus on the high-level logic of applications without needing to manage low-level system details directly.

- **Encapsulate the Execution State**: operates independently from other processes
- **Separate Resource Management**: OS takes care of scheduling, memory allocation for each process, enabling multitasking and efficient resource sharing.

The Process Abstraction

- **Facilitate Inter-Process Communication (IPC)**: Through mechanisms like pipes, message queues, and shared memory, processes can exchange information without direct access to each other's memory.
- **Ensure Isolation and Security**: Processes run in isolated memory spaces, preventing unauthorized access to each other's data, which enhances security and stability.

Multi-Processing

Multi-processing refers to a system in which multiple processes are executed **simultaneously**.

- Processes may belong to the same program or different programs
- Processes might communicate with each other via Inter-Process Communication
- Multi-processing leverages multiple CPUs or cores to execute these processes concurrently, thus improving performance in tasks that can be parallelized.

Multi-Programming

Multi-programming refers to a system where multiple programs are loaded into memory and executed concurrently by the operating system, often by quickly **switching** between them.

- Managing CPU time efficiently by keeping several programs in memory and letting them take turns running.
- System switches between programs when one is waiting for resources (like I/O), a technique called context switching.

Key Differences

- **Purpose:**
  - Executing multiple processes simultaneously
  - Efficiently managing CPU time among programs
- **Execution:**
  - Can run processes on multiple cores
  - Typically uses a single core with time-slicing

Key Differences

- **Resource Sharing:**
  - Processes are isolated, IPC needed for sharing
  - Programs share CPU and memory managed by the OS
- **Example Use:**
  - Web servers, parallel computing
  - Desktop OS running multiple applications

How to Create A Process

- Processes are created by the OS, typically by the **fork** command.
  - The process that calls fork is the **parent** and the new processis the **child**.
  - The child inherits a replica of the parent's address space and isessentially a clone.
  - Both continue to execute the identical program.
  - Fork returns the child's process id to the parent, and the value 0 to the child.
  - The **exec** system call loads another program and starts running this (typically in the child process)

How to Create A Process: Example

```
#include <iostream>
#include <unistd.h>
int main() {
   // Call fork() to create a new process
   pid_t pid = fork();

   // Check if fork() was successful
   if (pid < 0) {
      // Fork failed
   } else if (pid == 0) {
      // This is the child process
   } else {
      // This is the parent process
   }
}
```

Process IDs

fork() returns 0 to the child, and child PID to the parent.

- `getpid()`: get process id
- `getppid()`: get parent's process id

```
   ...
    } else if (pid == 0) {
// Child process
cout << "Child process - PID: " << getpid() << ", Parent
    PID: " << getppid();
   ....
```

Process States

- All processes in UNIX have a “state of execution” which indicates the current stage of the process in its life-cycle.
- A process can only be in one of the following states at a time

Process States

- **New**
  - **Description:** The process is being created. It hasn't yet been admitted by the operating system scheduler and therefore isn't ready for execution.
  - **Transition:** Moves to the Ready state once it's set up and admitted by the scheduler.
- **Ready**
  - **Description:** The process is loaded into memory and ready to execute, waiting for CPU time. Multiple processes can be in the ready state, waiting to be scheduled by the OS.
  - **Transition:** The scheduler picks it up, and it transitions to the Running state.

Process States

- **Running**
  - **Description:** The process is currently executing on the CPU. Only one process can be in the Running state per CPU core at any moment (on a single-core system).
  - **Transition:** A process in the Running state may:
    - Complete Execution and move to the Terminated state.
    - Wait for an I/O operation or resource, which moves it to the Waiting/Blocked state.
    - Be preempted by the scheduler (time slice end), moving it back to the Ready state.

Process States

- **Waiting/Blocked**
  - **Description:** The process is waiting for an event or resource, such as I/O completion or a specific signal, so it can continue executing. It cannot use the CPU while in this state.
  - **Transition:** Once the required resource or event becomes available, it returns to the Ready state.

Process States

- **Terminated (or Zombie)**
  - **Description:** The process has finished executing, either normally or due to an error. It has released most of its resources but still has an entry in the process table, allowing the parent process to read its exit status.
  - **Zombie State:** If a child process terminates but its parent process hasn't yet acknowledged (or "reaped") it, the process remains as a zombie.
  - **Transition:** The process entry is removed from the process table once the parent process reads the exit status using wait(), fully clearing it from the system.

Foreground and Background Processes

- A foreground process is one that occupies your shell (terminal window), meaning that any new commands that are typed have no effect until the previous command is finished
- If we're running a process that might take a very long time, and we want the terminal to be available in the meantime, we can run the process in the background.
- When a process is run as a background process, the terminal starts the job, then immediately displays the prompt waiting for the next command.
- When the background job is complete, its results can be displayed by bringing it back to the foreground.

Inter-Process Communication (IPC)

- **Why IPC?**: Processes do not share memory by default, so they need IPC mechanisms to exchange data.
- **Common IPC Methods:**
  - Pipes: Simple communication channel, unidirectional.
  - Shared Memory: Memory segment shared between processes.
  - Message Queues & Semaphores: Structured and synchronized ways to exchange data.

Waiting for a Child Process with `wait()`

- Ensures that the parent process does not proceed until the child has finished.
- wait() returns the PID of the terminated child.
- wait() prevents zombie processes: when wait() is called, the parent process "reaps" the child process, effectively removing it from the system. Without wait(), the child process might become a zombie, as it has terminated but hasn't been acknowledged by the parent.

Example wait()

```
// Parent process
std::cout << "Parent process (PID: " << getpid() << ") is
    waiting for the child to finish." << std::endl;

wait(nullptr); // Wait for the child process to complete

std::cout << "Parent process detected that child has
    finished." << std::endl;
```

Exiting Child Processes Gracefully

- Using exit() in the Child Process: Ensures proper termination.

```
// Child process
std::cout << "This is the child process with PID: " <<
    getpid() << std::endl;

// Simulate some work in the child process
sleep(2); // Pause for a couple of seconds

std::cout << "Child process is done. Exiting now." <<
    std::endl;
exit(0);
```

Process Priorities

- All Unix processes have a priority, which rates the importance f one process with respect to all the other processes currently in the system.
- Unix systems use a priority system with 40 priorities, ranging from -20 (highest priority) to 19 (lowest priority.
- Processes started by regular users usually have priority 0 (normal)