21CS2109AA

Operating Systems

Session 2

The Abstraction: The Process

How to provide the illusion of many CPUs?

- CPU virtualizing
  - The OS can promote the illusion that many virtual CPUs exist.
  - **Time sharing**: Running one process, then stopping it and running another
    - The potential cost is performance.

OS Process Abstraction

When you run an exe file, the OS Creates a process = a running program
OS timeshares CPU across multiple processes: virtualizes CPU
OS has a CPU Scheduler that picks one of the many active processes to execute on a CPU
- Policy: Which process to run
- Mechanism: how to "Context switch" between processes

A process is a running program.

Operating System Creates a Process

Allocates Memory and creates memory Image

- Loads code, data from disk exe
- Creates runtime stack, heap

Opens basic files

- STD IN, OUT, ERR

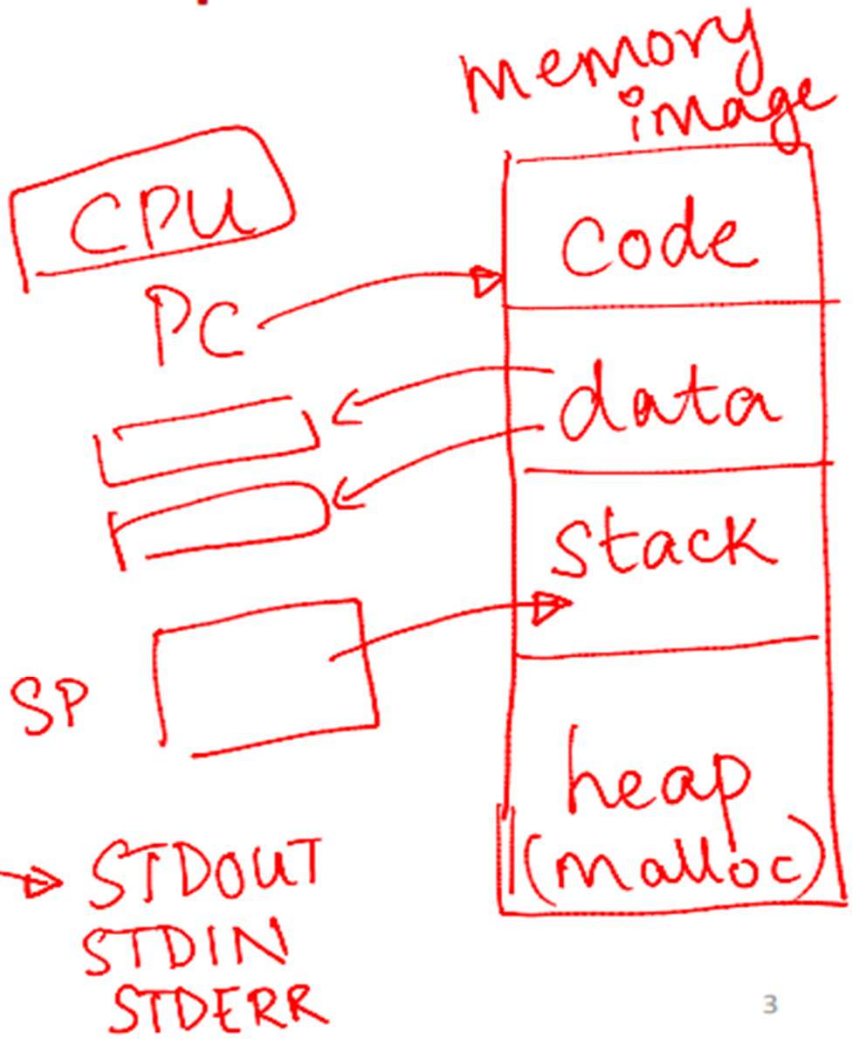
Inisitalizes CPU registers

- PC points to first instruction

Comprising of a process:
Memory (address space)
Instructions
Data section
Registers
Program counter-tells us which instruction of the program is currently being executed
Stack pointer-are used to manage the stack for function parameters

What constitutes a process?

- A unique identifier (PID)
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer
- File descriptors
  - Pointers to open files and devices



3

Process API

- These APIs are available on any modern OS.
  - **Create**
    - OS is invoked to Create a new process to run a program
  - **Destroy**
    - Halt a runaway process
  - **Wait**
    - Wait for a process to stop running
  - **Miscellaneous Control**
    - Some method to suspend a process and then resume it
  - **Status**
    - Get some status info about a process

Process Creation

1. **Load** a program code into memory, into the address space of the process.
   - Programs initially reside on disk in *executable format*.
   - OS perform the loading process lazily.
     - Loading pieces of code or data only as they are needed during program execution.

2. The program's run-time **stack** is allocated.
   - Use the stack for *local variables*, *function parameters*, and *return address*.
   - Initialize the stack with arguments → `argc` and the `argv` array of `main()` function(**argc** contains the number of arguments passed to the program. The name of the variable **argv** stands for "argument vector")

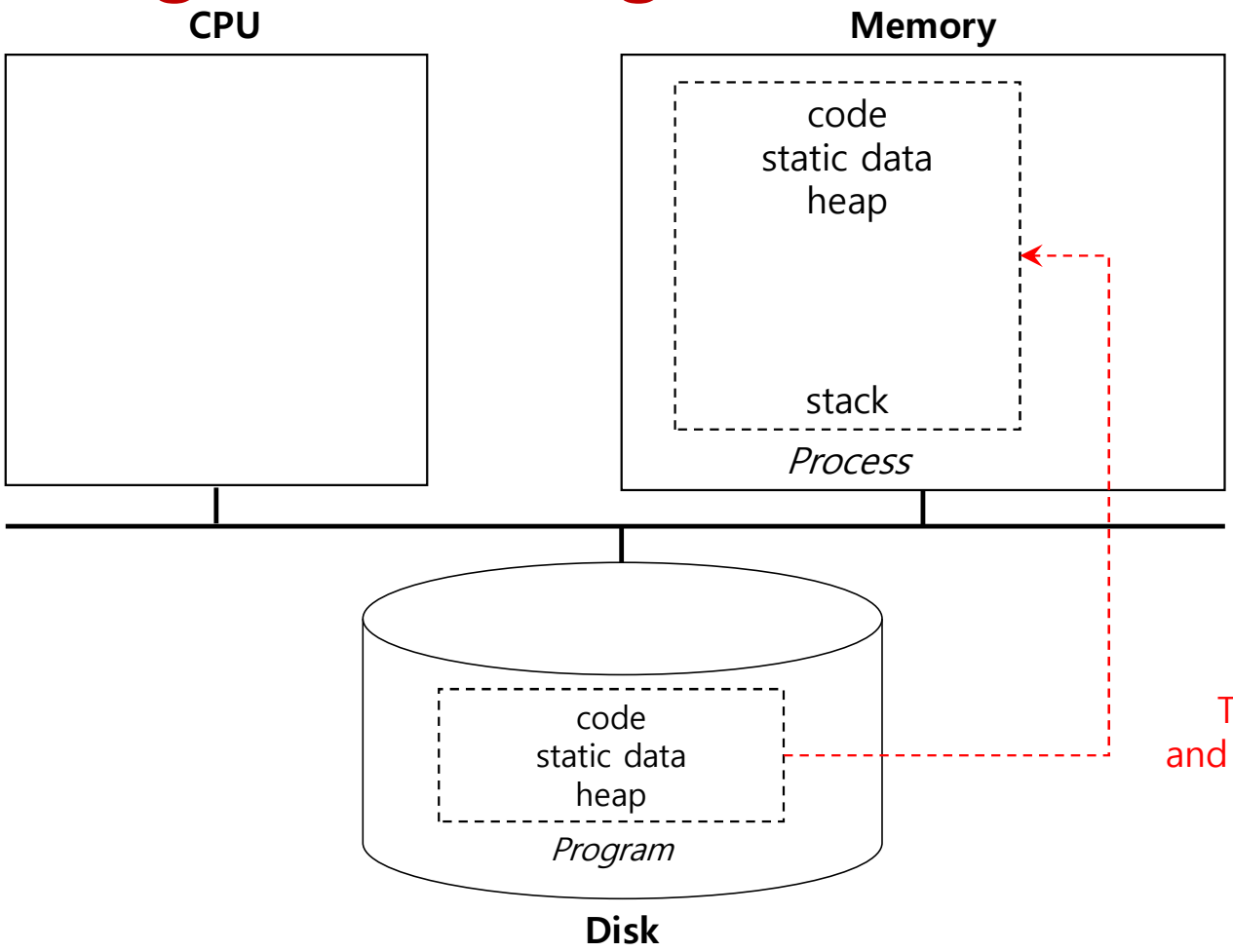
Process Creation (Cont.)

3. The program's **heap(area in memory to store program)** is created.
   - Used for explicitly requested dynamically allocated data.
   - Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS do some other initialization tasks.
   - input/output (I/O) setup
     - Each process by default has three open file descriptors.
     - Standard input, output and error

5. **Start the program** running at the entry point, namely `main()`.
   - The OS *transfers control* of the CPU to the newly-created process.

Loading: From Program To Process

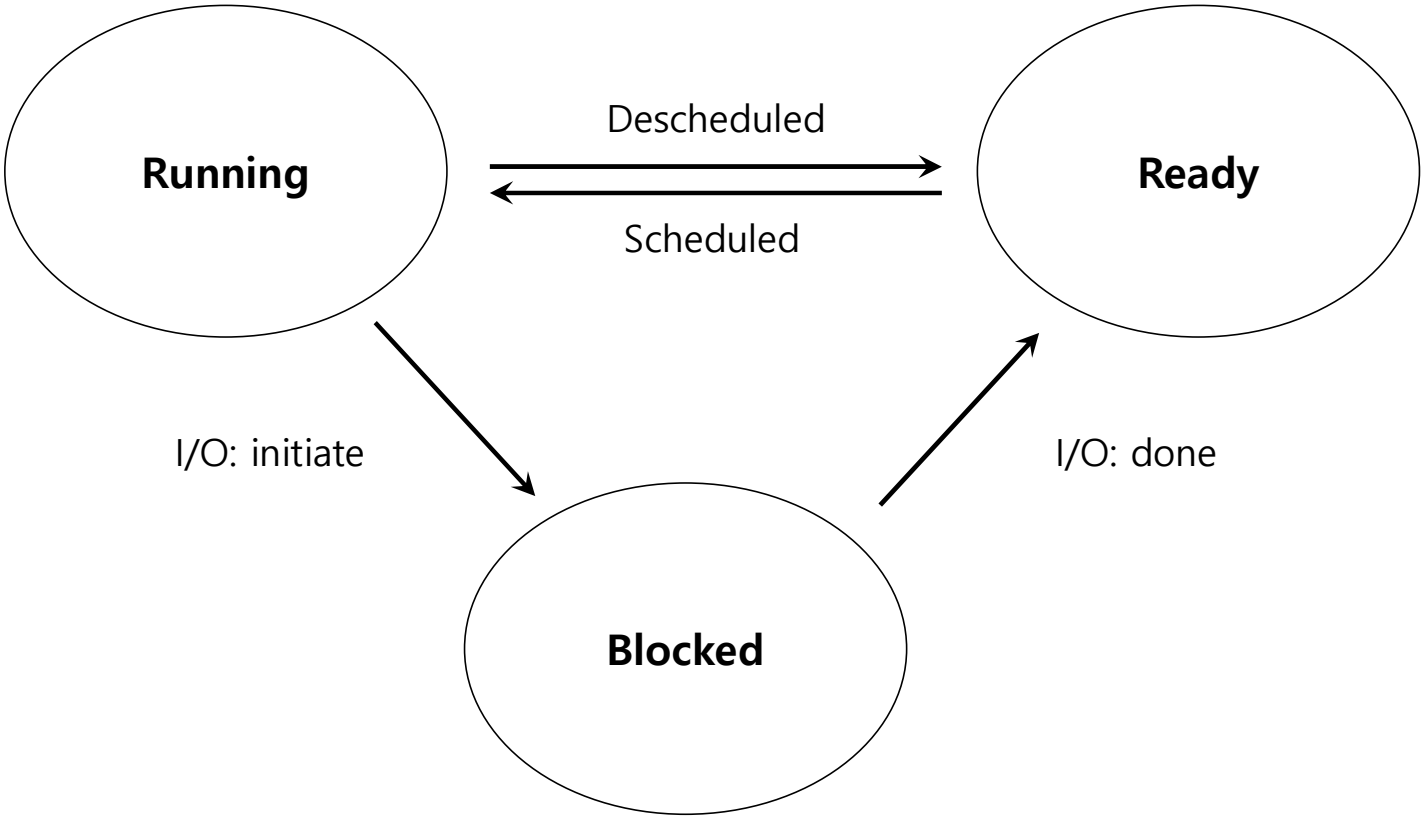


Loading:
Takes on-disk program and reads it into the address space of process

Process States

- A process can be one of three states.
  - **Running**
    - A process is running on a processor. This means it is executing instructions
  - **Ready**
    - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked**
    - A process has performed some kind of operation.
    - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition

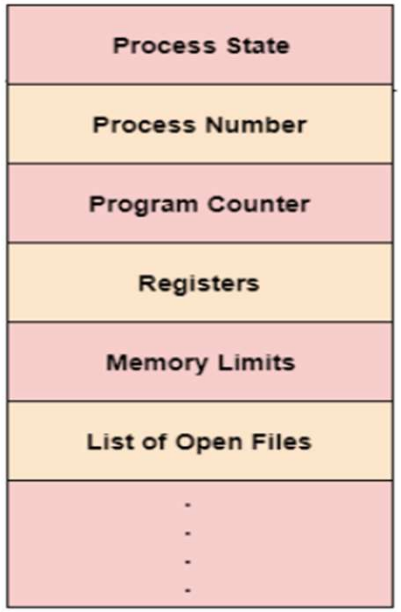


Data structures

- The OS has some key data structures that track various relevant pieces of information.
  - **Process list**
    - Ready processes
    - Blocked processes
    - Current running process
- **Register context:** When a process is stopped, its register state will be saved to this memory location; by restoring these registers (i.e., placing their values back into the actual physical registers), the OS can resume running the process.
- PCB(Process Control Block)
  - A C-structure that contains information about each process.

Process control block(PCB)

- Process Control Block is a data structure that contains information of the process related to it.



Process Control Block (PCB)

The following are the data items:

- Process State:

  This specifies the process state i.e. new, ready, running, waiting or terminated.

- Process Number:

  This shows the number of the particular process.

- Program Counter:

  This contains the address of the next instruction that needs to be executed in the process.

- Registers:

  This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.

- List of Open Files:

  These are the different files that are associated with the process

- CPU Scheduling Information:

  The process priority, pointers to scheduling queues etc. is the CPU scheduling information that is contained in the PCB. This may also include any other scheduling parameters.

- Memory Management Information:

  The memory management information includes the page tables or the segment tables depending on the memory system used. It also contains the value of the base registers, limit registers etc.

- I/O Status Information:

  This information includes the list of I/O devices used by the process, the list of files etc.

- **Accounting information:**

  The time limits, account numbers, amount of CPU used, process numbers etc. are all a part of the PCB accounting information.

- **Location of the Process Control Block:**

  The process control block is kept in a memory area that is protected from the normal user access. This is done because it contains important process information. Some of the operating systems place the PCB at the beginning of the kernel stack for the process as it is a safe location.

System calls related to process in linux

Process creation: fork()
Process blocked: wait()
Process : exec()

The fork() System Call

- Fork system call use for creates a new process, which is called ***child process***, which runs concurrently with process (which process called system call fork) and this process is called ***parent process***.
- After a new child process created, both processes will execute the next instruction following the fork() system call.
- A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork():

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

getppid() and getpid() in Linux

Both getppid() and getpid() are inbuilt functions defined in **unistd.h** library.

1. getppid() :

- returns the process ID of the parent of the calling process.
- If the calling process was created by the **fork()** function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process.
- Otherwise, this function returns a value of 1 which is the process id for **init**process.

2. getpid() : returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

The wait() System Call

- This system call won't return until the child has run and exited.
- A call to wait() blocks the calling process until one of its child processes exits or a signal is received.
- After child process terminates, parent ***continues*** its execution after wait system call instruction.

- In order to virtualize the CPU, the operating system needs to share the physical CPU among many jobs running seemingly at the same time.
- The basic idea is simple: run one process for a little while, then run another one, and so forth. By **time sharing the CPU in this manner,** virtualization is achieved.
- There are a few challenges, however, in building such virtualization : The first is **performance**: how can we implement virtualization without adding excessive overhead to the system?
- The second is **control**: how can we run processes efficiently while retaining control over the CPU? without control, a process could simply run forever and take over the machine

Problem 1: Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as …
  - Issuing an I/O request to a disk
  - Gaining access to more system resources such as CPU or memory

- **Solution**: Using protected control transfer:

The hardware assists the OS by providing different modes of execution.

- User mode: Applications do not have full access to hardware resources.
- Kernel mode: The OS has access to the full resources of the machine

System Call

- Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
  - Accessing the file system
  - Creating and destroying processes
  - Communicating with other processes
  - Allocating more memory

System Call (Cont.)

- To execute a system call, a program must execute a special **trap instruction.** This instruction Jump into the kernel and Raise the privilege level to kernel mode.
- once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process.
- When finished, the OS calls a special **Return-from-trap** instruction, which Return into the calling user program and Reduce the privilege level back to user mode

Problem 2: Switching Between Processes

- The next problem with direct execution is achieving a switch between processes. Switching between processes should be simple, right?
- The OS should just decide to stop one process and start another. If a process is running on the CPU, this by definition means the OS is not running.
- there is clearly no way for the OS to take an action if it is not running on the CPU.
- How can the OS regain control of the CPU so that it can switch between *processes*?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

A cooperative Approach: Wait for system calls

- Most processes, as it turns out, transfer control of the CPU to the OS quite frequently by making **system calls.**
- Processes periodically give up the CPU by making **system calls** such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - Ex) Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop.
→ Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
  - The timer raise an interrupt every so many milliseconds.
  - When the interrupt is raised :
    - The currently running process is halted.
    - Save enough of the state of the program
    - A pre-configured interrupt handler in the OS runs.

A timer interrupt gives OS the ability to run again on a CPU.

Saving and Restoring Context

- Scheduler makes a decision:
  - Whether to continue running the **current process**, or switch to a **different one**.
  - If the decision is made to switch, the OS executes context switch.

Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
  - **Restore a few** for the soon-to-be-executing process from its kernel stack
  - **Switch to the kernel stack** for the soon-to-be-executing process

Week 2 – Assignments

1. Shell Script the following Perform Operations on a File (A/P)
    a. Display the contents of the file
    b. Counting characters, words & lines in the file
2. Differentiate the Batch File and Shell Scripting.
3. Write a Shell Script to accept a number and find Even or ODD
4. Write a Shell Script to accept a year and find Leap Year or Not
5. Write a Shell Script to check a given number is a prime number or Not
6. Write a Shell Script to find Factorial of a given number
7. Write a Shell Script to find Greatest of given Three numbers
8. Write a Shell Script to emulate the UNIX ls-l command. (A/P)
9. Write a Shell Script to accept numbers and print sorted numbers (A/P)
10. Write a Shell Script to accept State name and print the Capital using CASE(A/P)
11. Write a Shell Script for Arithmetic Calculator using CASE(A/P)
12. Write a Shell Script to Illustrate Logical Operators using expression evaluator expr (A/P)
13. Write a Shell Script to accept a string and display the length of the string (A/P)
14. Write a Shell Script to Illustrate the Functions (A/P)
15. Explain Unix architecture in detail with neat diagram
16. What is Shell Script in Operating System and How does it work?

Week 2 – Projects

1. **Create a Menu Driven Shell Program for HELP Tutorial on Commands as Manual (A/P)**