MONASH University

Information Technology

FIT2100 Operating Systems

PROCESSES AND PROCESS CREATION

Week - 5
Semester 2 2024
(Reading: Tanenbaum, Chapter 2)

Dr Charith Jayasekara
Faculty of Information Technology

OUTLINE

- The Process Model
- Process Creation
- Process Termination
- Process Hierarchies
- Process States
- Implementation
- Modelling Multiprogramming.

LEARNING OUTCOMES

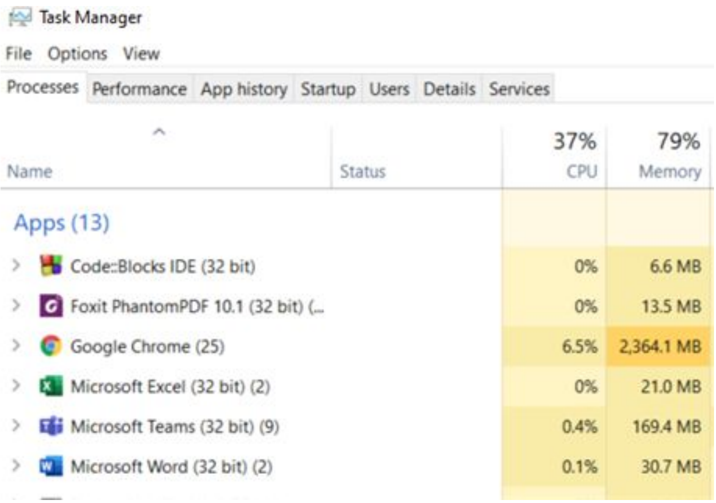
- Upon the completion of Week - 5, you should be able to:
  - Define the concept of **process**
  - Understand the relationship between **processes** and **process control blocks (PCB)**
  - Discuss the concepts of **process state** and **state transitions**
  - Understand how **multiprogramming** works in real world systems

THE PROCESS MODEL

PROCESSES

WHAT IS A PROCESS

- *Process :* an abstraction of a running system
- Oldest and most important abstraction an OS provides.
- Support ability to have concurrent operation even when there is only one CPU available.



Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	Status	37% CPU	79% Memory
Apps (13)			
Code::Blocks IDE (32 bit)		0%	6.6 MB
Foxit PhantomPDF 10.1 (32 bit) (...		0%	13.5 MB
Google Chrome (25)		6.5%	2,364.1 MB
Microsoft Excel (32 bit) (2)		0%	21.0 MB
Microsoft Teams (32 bit) (9)		0.4%	169.4 MB
Microsoft Word (32 bit) (2)		0.1%	30.7 MB

MONASH University

PROCESSES

THE PROCESS MODEL

- In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes.
- A process is just an instance of an executing program, including the current values of the program counter, registers, and variables.

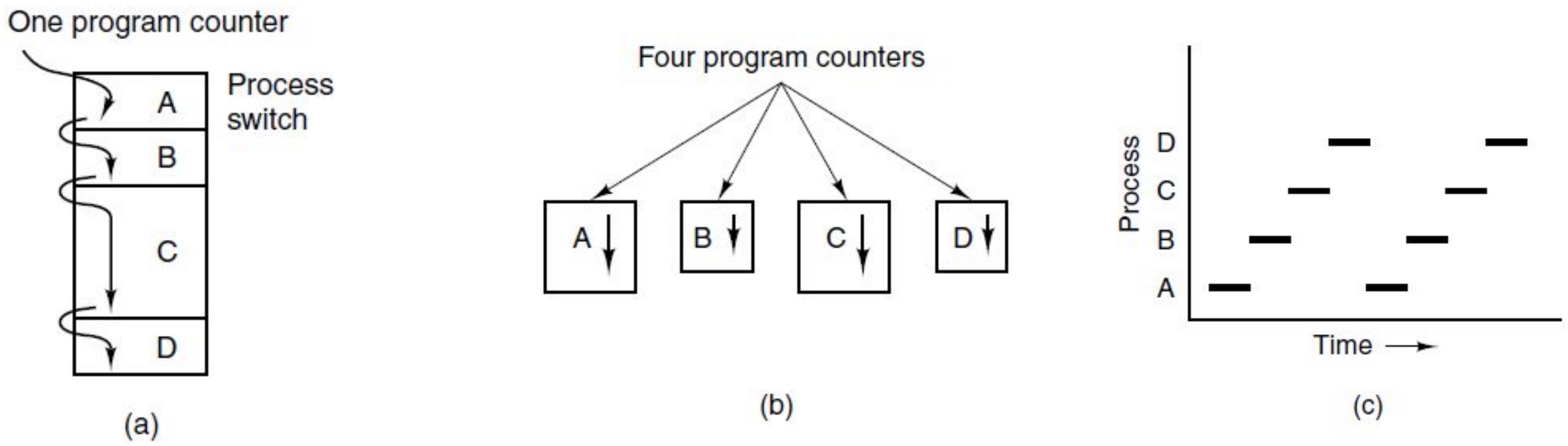
PROCESSES

THE PROCESS MODEL

- Conceptually, each process has its own virtual CPU.
- In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel than to try to keep track of how the CPU switches from program to program.
- This rapid switching back and forth is called multiprogramming

PROCESSES

THE PROCESS MODEL



A. Multiprogramming four programs.
B. Conceptual model of four independent, sequential processes.
C. Only one program is active at once.

PROCESS CREATION

PROCESS CREATION

PRINCIPLE EVENTS CAUSE PROCESSES TO BE CREATED

- System initialization.
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

PROCESS CREATION

SYSTEM INITIALIZATION

- When OS is booted, typically numerous processes are created.
    - Foreground processes
        - Interacts with users to perform work for them
    - Background processes
        - Not associated with users, but have some specific functions
        - Accepting an email, accepting incoming requests coming to a web browser from a web server.
        - Processes that stay in the background to handle some activity such as email, web pages, news, printing, and so on are called **daemons.**

PROCESS CREATION

EXECUTION OF A PROCESS-CREATION SYSTEM CALL BY A RUNNING PROCESS

- New processes can be created after booting the system.
- Often a running process will issue system calls to create one or more new processes to help it do its job.
- Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes.
  - For example, if a large amount of data is being fetched over a network for subsequent processing,
    - One process for fetch the data and put them in a shared buffer
    - Second to remove the data items from the buffer to process them.

PROCESS CREATION

A USER REQUEST TO CREATE A NEW PROCESS

- In interactive systems, users can start a program by typing a command or (double) clicking on an icon.
- Taking either of these actions starts a new process and runs the selected program in it.
  - In command-based UNIX systems running X, the new process takes over the window in which it was started.
  - In Windows, when a process is started it does not have a window, but it can create one (or more) and most do. In both systems, users may have multiple windows open at once, each running some process.
  - Using the mouse, the user can select a window and interact with the process, for example, providing input when needed.

PROCESS CREATION

INITIATION OF THE BATCH JOB

- The last situation in which processes are created applies only to the batch systems found on large mainframes.
- Think of inventory management at the end of a day at a chain of stores. Here users can submit batch jobs to the system (possibly remotely).
- When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

PROCESS CREATION

UNIX VS WINDOWS

UNIX

- There is only one system call to create a new process: **fork**.
- This call creates an exact clone of the calling process.
- The two processes, the **parent** and the **child**, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes *execve* or a similar system call to change its memory image and run a new program.

PROCESS CREATION

UNIX VS WINDOWS

WINDOWS

- A single Win32 function call, *CreateProcess*, handles both process creation and loading the correct program into the new process.
- This call has 10 parameters, which include the program to be executed, the command-line parameters to feed that program, various security attributes, bits that control whether open files are inherited, priority information, a specification of the window to be created for the process (if any), and a pointer to a structure in which information about the newly created process is returned to the caller.
- In addition to *CreateProcess*, Win32 has about 100 other functions for managing and synchronizing processes and related topics.

PROCESS CREATION

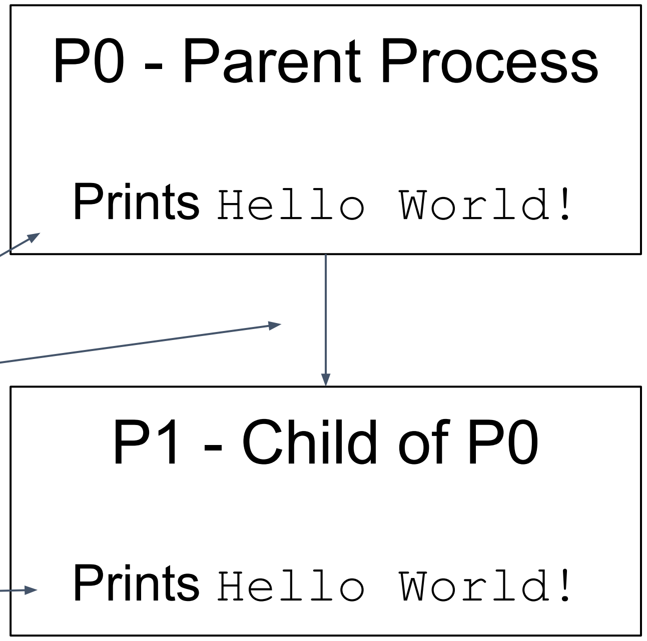
EXAMPLE - FORK() SYSTEM CALL

- Used to create new process.
- A child process will be created that runs concurrently with the parent process.
- After child process is created, both child and parent will execute from the next instruction following the `fork()` system call. The child process uses the same PC (program counter), same CPU registers, same open files which are used by the parent process.
- Takes no input arguments and returns an integer.
  - Return value
    - Positive value (process ID of the new process) will be returned to the parent process
    - 0 will be returned to child process
    - If the process creation is unsuccessful, a negative value will be returned.

PROCESS CREATION

EXAMPLE 1 - FORK() SYSTEM CALL

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    printf("Hello world!\n");
    return 0;
}
```



P0 - Parent Process

Prints `Hello World!`

P1 - Child of P0

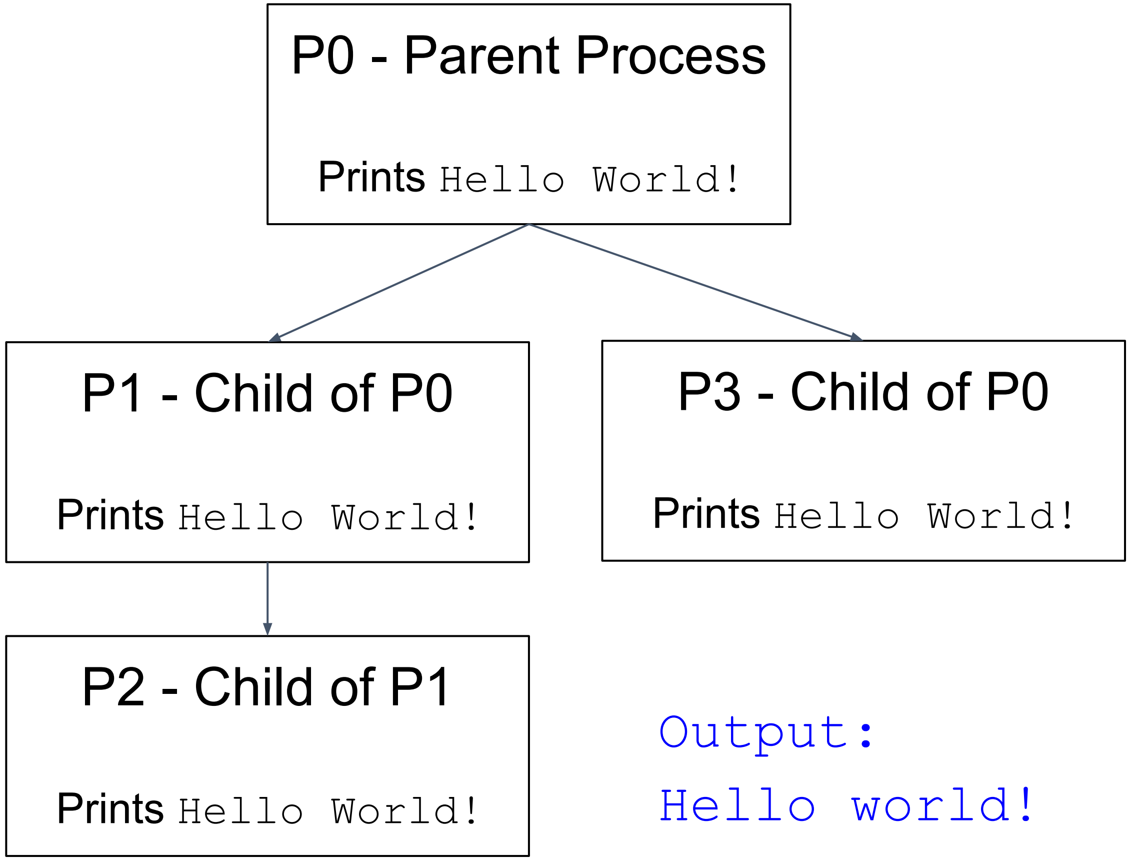
Prints `Hello World!`

```
Output:
Hello world!
Hello world!
```

PROCESS CREATION

EXAMPLE 2 - FORK() SYSTEM CALL

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   fork();  //creates P1
   fork();  //creates P2,P3
   printf("Hello world!\n");
   return 0;
}
```



```
Output:
Hello world!
Hello world!
Hello world!
Hello world!
```

PROCESS TERMINATION

PROCESS TERMINATION

TYPICAL CONDITIONS WHICH TERMINATE A PROCESS

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).

PROCESS TERMINATION

NORMAL EXIT

- Most processes terminate because they have done their work.
- When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished.
- This call is `exit()` in UNIX and ExitProcess in Windows.
- Screen-oriented programs also support voluntary termination. Word processors, Internet browsers, and similar programs often have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

PROCESS TERMINATION

ERROR EXIT

- The second reason for termination is that the process discovers a fatal error.
  - For example, if a user types the command

    ```
    cc foo.c
    ```

    to compile the program *foo.c* and no such file exists, the compiler simply announces this fact and exits.
- Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop up a dialog box and ask the user to try again.

PROCESS TERMINATION

FATAL ERROR

- The third reason for termination is an error caused by the process, often due to a program bug.
- Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero.
  - In some systems (e.g., UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.

PROCESS TERMINATION

KILLED BY ANOTHER PROCESS

- The fourth reason a process might terminate is that the process executes a system call telling the operating system to kill some other process.
- In UNIX this call is kill. The corresponding Win32 function is TerminateProcess .
  - In both cases, the killer process must have the necessary authorization.
- In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well. Neither UNIX nor Windows works this way, however.

PROCESS HIERARCHIES

PROCESS HIERARCHY

- In some systems, parent processes and child processes may continue to be associated in certain ways.
- The child process can itself create more processes and form a process hierarchy.
- A process has only one parent, but zero or more children.

PROCESS HIERARCHY

UNIX

- A process and all of its children and further descendants together form a process group.
- When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window).
- Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

PROCESS HIERARCHY

UNIX

- As another example, just after the computer is booted, a special process, called `init`, is present in the boot image.
- When it starts running, it reads a file telling how many terminals there are.
- Then it forks off a new process per terminal.
- These processes wait for someone to log in.
- If a login is successful, the login process executes a shell to accept commands.
- These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with `init` at the root.

PROCESS HIERARCHY

WINDOWS

- Windows has no concept of a process hierarchy.
- All processes are equal.
- The only hint of a process hierarchy is that when a process is created, the parent is given a special token (called a handle) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy.
- Processes in UNIX cannot disinherit their children.

PROCESS STATES

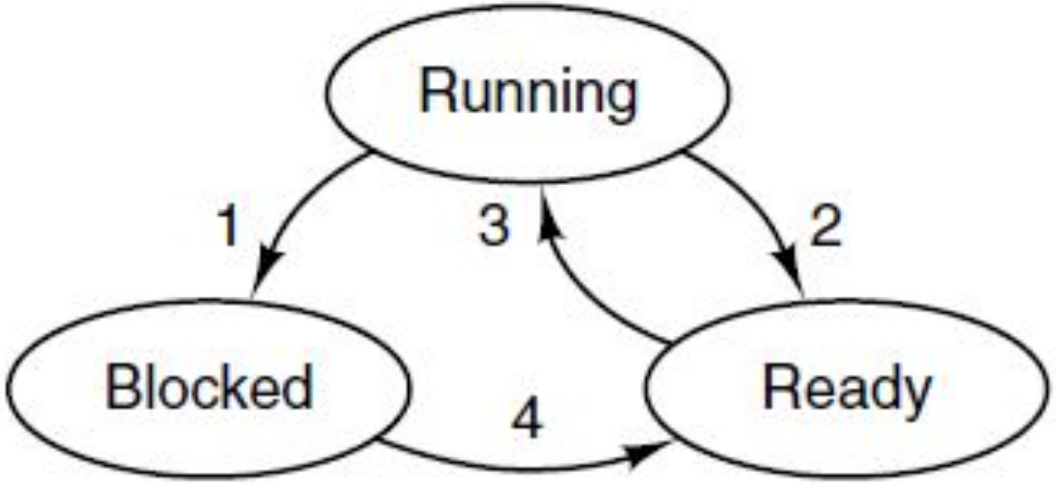
PROCESS STATES

THREE STATES A PROCESS CAN BE IN.

- 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).

PROCESS STATES

THREE STATES A PROCESS CAN BE IN.



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

A process can be in running, blocked, or ready state. Transitions between these states are as shown.

IMPLEMENTATION OF PROCESSES

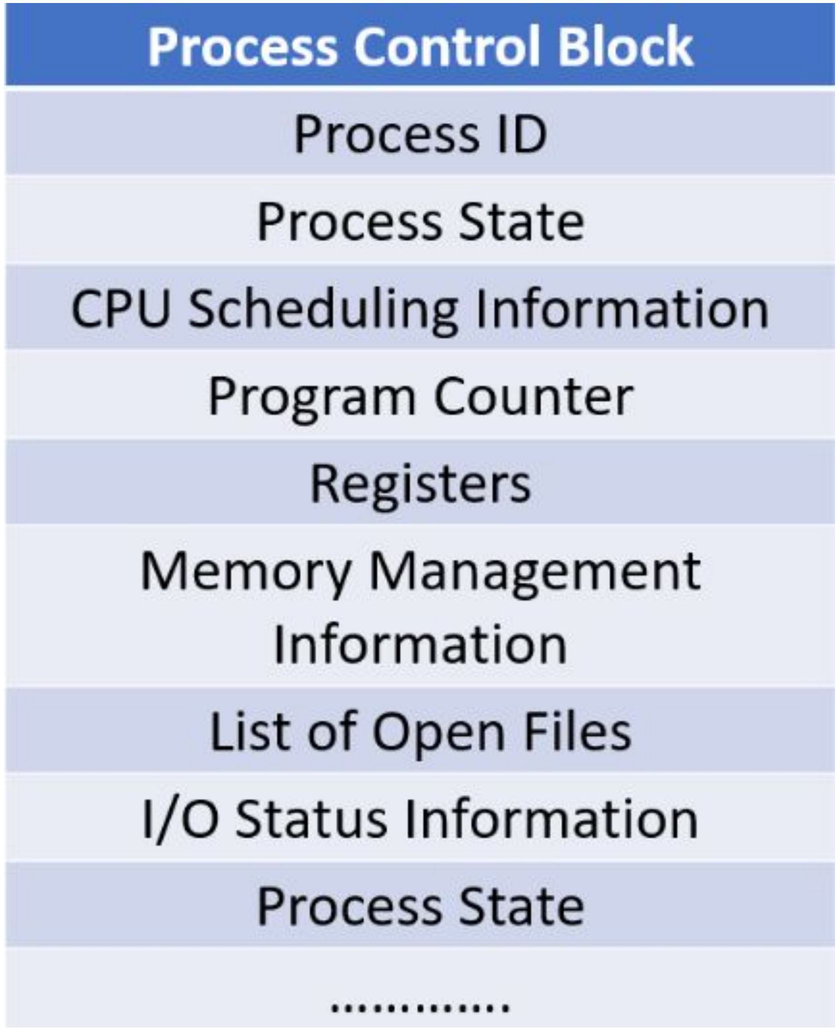
IMPLEMENTING A PROCESS

PROCESS CONTROL BLOCK

- To implement a process, the OS maintains a table (an array of structures)
- This table is called as **process table** and has one entry per process.
- Each entry is called as **Process Control Block** and contains process state.
    - programme counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information and anything else about the process that may require for switching from running to ready or blocked state.

IMPLEMENTING A PROCESS

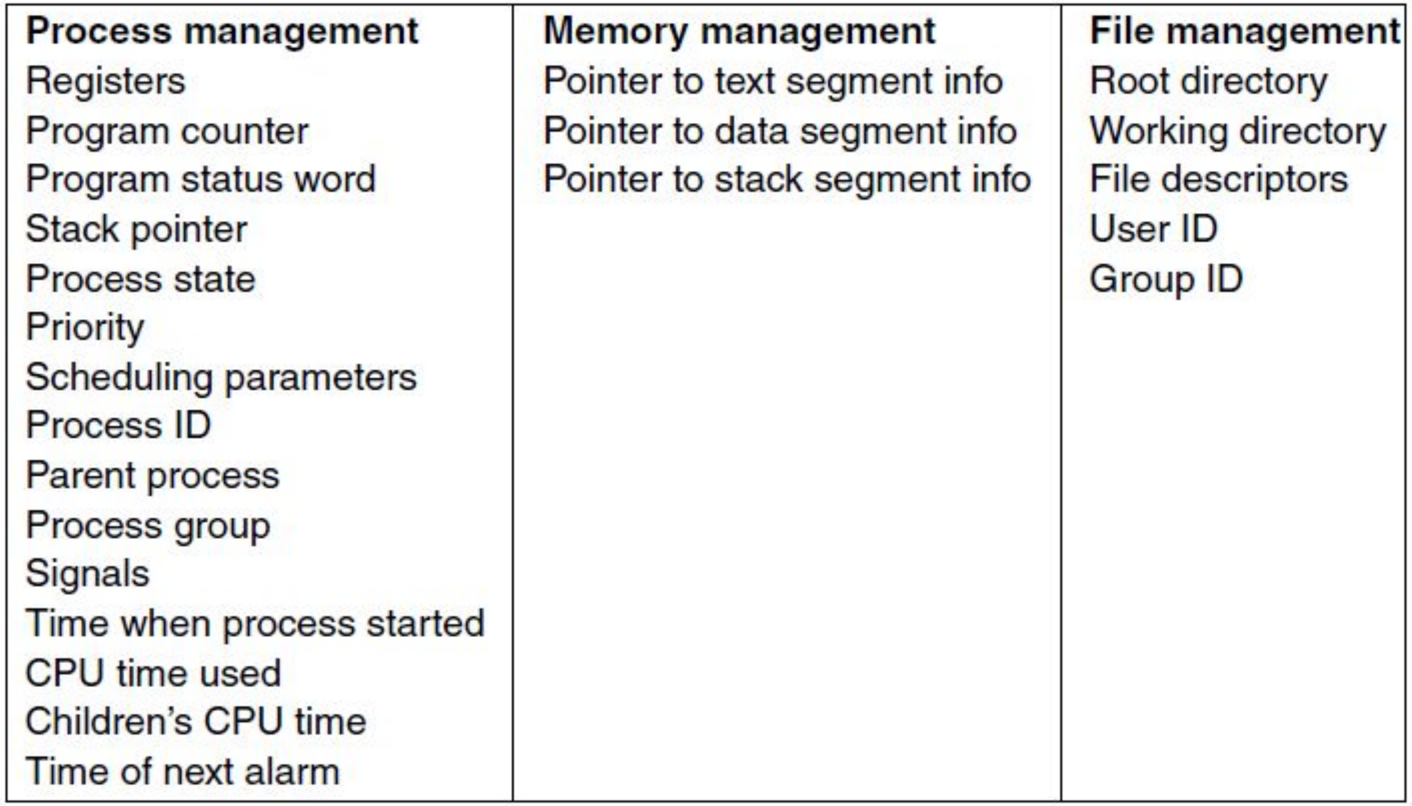
PROCESS CONTROL BLOCK



Process Control Block
Process ID
Process State
CPU Scheduling Information
Program Counter
Registers
Memory Management Information
List of Open Files
I/O Status Information
Process State
....

IMPLEMENTING A PROCESS

PROCESS CONTROL BLOCK



Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children’s CPU time		
Time of next alarm		

Some of the fields of a typical process table entry.

IMPLEMENTING A PROCESS

ILLUSION OF MULTIPLE SEQUENTIAL PROCESSES

- Associated with each I/O device category is a location (typically at a fixed location near the bottom of memory) called the interrupt vector.
- It contains the address of the interrupt service routine (ISR).
- E.g. A user process is running when a disk interrupt happens. User process' program counter, program status word, and sometimes one or more registers are pushed onto the (current) stack by the interrupt hardware.
- The computer then jumps to the address specified in the interrupt vector. That is all the hardware does. From here on, it is up to the software, in particular, the interrupt service routine.
- ISRs are mostly provided as part of the specific device drivers for a given device type.

IMPLEMENTING A PROCESS

INTERRUPT SERVICE ROUTINE (ISR)

- All interrupts start by saving the registers, often in the process table entry for the current process.
- Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler.

IMPLEMENTING A PROCESS

INTERRUPT SERVICE ROUTINE (ISR)

- Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly-language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is.
- When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type. (We assume the operating system is written in C, the usual choice for all real operating systems.)
- When it has completed its work, possibly making another process ready, the scheduler procedure is called to see which process is to run next.
- After that, control is passed back to the assembly-language code to load up the registers and memory map for the now-current process to start running it.

IMPLEMENTING A PROCESS

WHAT THE LOWEST LEVEL OF OS DOES WHEN AN INTERRUPT OCCURS

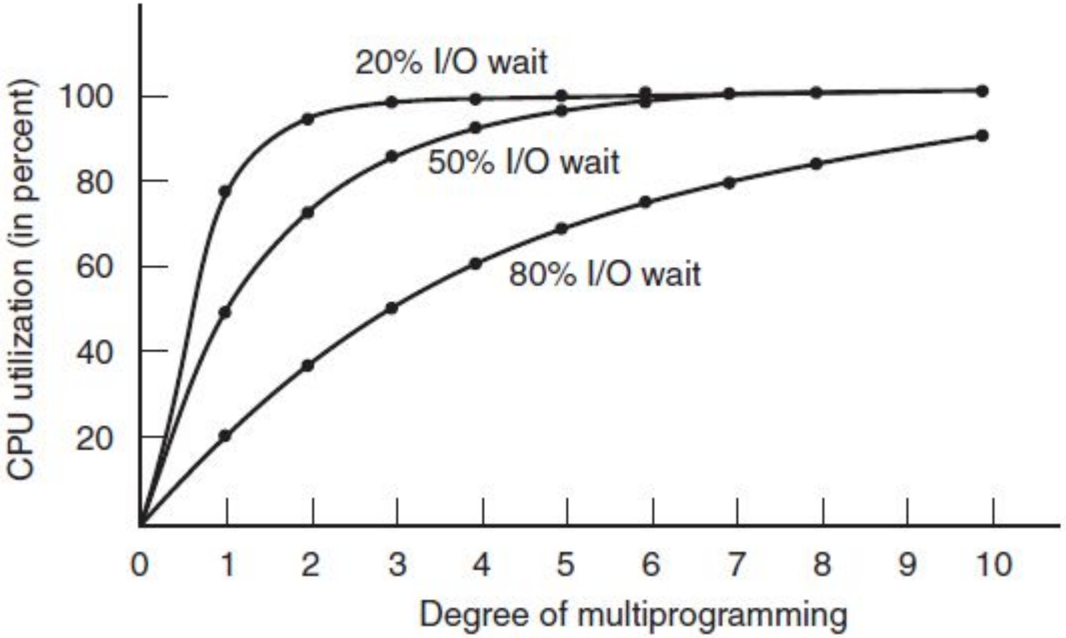
1. Hardware saves CPU register state, i.e. program counter, etc.
2. Hardware loads new program counter address from interrupt vector.
3. Assembly-language procedure saves all CPU state information, i.e. registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input, or manages device state).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

*This transition between the execution of processes is usually termed **“context switching”,** and its duration is a measurable performance parameter of any operating system*

MODELLING MULTIPROGRAMMING

MULTIPROGRAMMING

CPU UTILIZATION



CPU utilization as a function of the number of processes in memory.

Summary

- So far we have discussed
  - The process model
  - Process creation
  - Process termination
  - Process hierarchies
  - Process states
  - Implementation of processes
  - Modelling multiprogramming

- Next week
  - Process Scheduling

- Reading
  - Tanenbaum, Chapter 2(4th Edition)