CIIC 4050 / ICOM 5007
Operating Systems

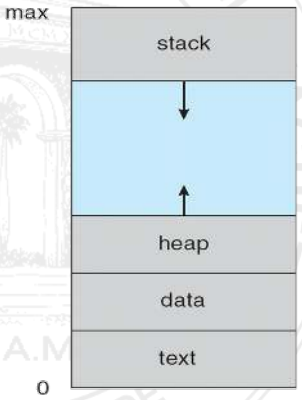
Lecture 3
Program Execution Models: Processes

Process Concept (1)

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- **Process** – a **program in execution**; process execution must progress in sequential fashion
  - One of the most important abstractions managed by an OS

Process Concept (2)

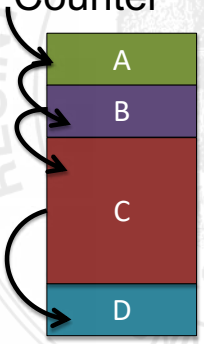
- A process includes:
  - stack
  - data section
  - Code (text) section
  - program counter



The Process Model (1)

- Multiprogramming of four programs.

One Program Counter

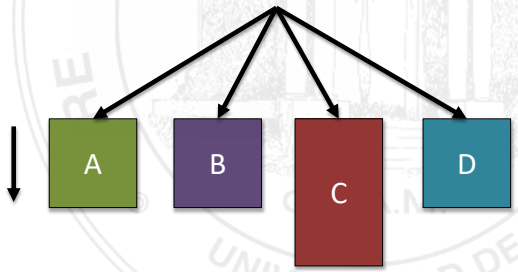


Context Switch
(between processes)

The Process Model (2)

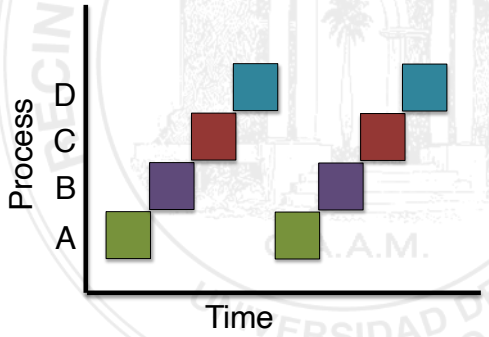
- Conceptual model of four independent, sequential processes.

Four Program Counters



The Process Model (3)

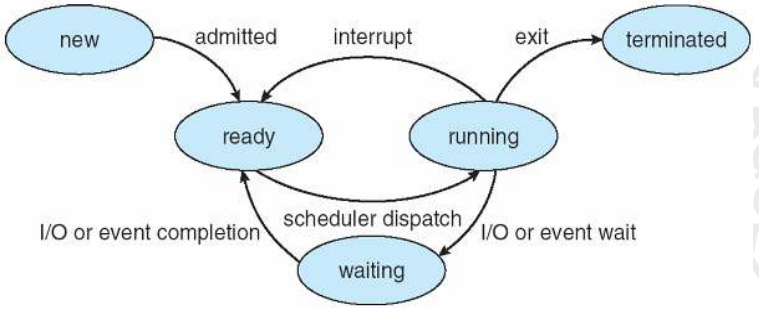
- Figure 2-1 (c) Only one program is active at any instant.



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State



States diagram for a process. The label at an edge describes a possible event that causes the transition.

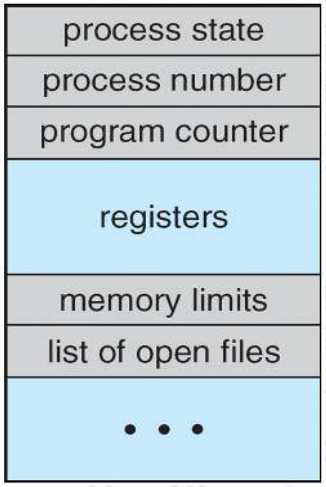
Process Control Block (PCB) (1)

- The system manages a data structure for each process: the **Process Control Block**
- It holds at least the following information about the process:
  - **Process state** - what state is the process (running, ready, …)
  - **Process id number** - used by the OS to identify the particular process
  - **Program counter** - where it goes in its code
  - **CPU registers** - contents of each register

Process Control Block (PCB) (2)

  - **CPU scheduling information** - process priority, pointer to scheduling queues, etc
  - **Memory-management information** - value of base and limit registers, page or segment tables, etc.
  - **Accounting information** - CPU time used, real time used, time limits, etc.
  - **I/O status information** - list of I/O devices allocated to the process, open files, etc.

Process Control Block (PCB)



Layout of the PCB data structure.

The PCB in Linux

The PCB in Linux is represented by a C structure: **`task_struct`**

- Some of its fields are:

```
pid_t pid;                          /* process identifier */
long state;                         /* state of the process */
unsigned int time_slice             /* scheduling information */
struct task_struct *parent;         /* this process's parent */
struct list_head children;          /* this process's children */
struct files_struct *files;         /* list of open files */
struct mm_struct *mm;               /* address space of this process */
```

As an illustration of how the kernel might manipulate one of the fields in the task struct for a specified process, let's assume the system would like to change the state of the process currently running to the value **`new_state`**. If `current` is a pointer to the process currently executing, its `state` is changed with the following:

```
current –> state = new state;
```

Implementation of Processes

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

- Figure 2-4. Some of the fields of the MINIX 3 PCB. The fields are distributed over the kernel, the process manager, and the file system.

Process Creation

- Principal events that cause processes to be created:
- 1. System initialization.
- 2. Execution of a process creation system call by a running process.
- 3. A user request to create a new process.
- 4. Initiation of a batch job.

Process Termination

- Conditions that cause a process to terminate:
- 1. Normal exit (voluntary).
- 2. Error exit (voluntary).
- 3. Fatal error (involuntary).
- 4. Killed by another process (involuntary).

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
  - Stored in PCB

Process Creation (2)

- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (3)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Some Important System Calls for Process Management (1)

- **`short fork()`** - creates a new process as an exact copy of the current one. Both continue executing from the return of the fork at their corresponding address spaces. Returns 0 for the child and the pid of the child for the parent process.

Some Important System Calls for Process Management (2)

- **`short wait(byte* return_code)`** - process waits (sleeps) for one for its children to terminate. Returns the pid of the terminating children. Any terminating children wakes the process. The value at **`return_code`** is an integer value in the range (0-255) stating some termination status. Usually, 0 is used to represent successful termination of corresponding child.

Some Important System Calls for Process Management (3)

- **`void exit(byte status)`** - terminates the current process while returning the value of status. Such value is the one captured by the parameter of the parents wait statement (if any) that is affected by the termination. Terminating process goes into ***zombie state*** …

Some Important System Calls for Process Management (4)

- **`void execl(parameters)`** - replaces the entire content of the current process's address space by the address space corresponding to the executable file identified by the parameters and with the initial argument values stated.

Some Important System Calls for Process Management (5)

- **`parameters`** is a variable number of string values. Last parameter is 0 (end of the list).
  - first parameter: name of the executable file
  - second parameter: name of the program (may be different from the name of the file). Such name is the first parameter received by the first entry of the **char argument in the main function of the corresponding executable…
  - other parameters are assigned to positions 1, 2, … of main's argument array … (recall **`int main(int argc, char* argv[]) {…}`**)

Some Important System Calls for Process Management (6)

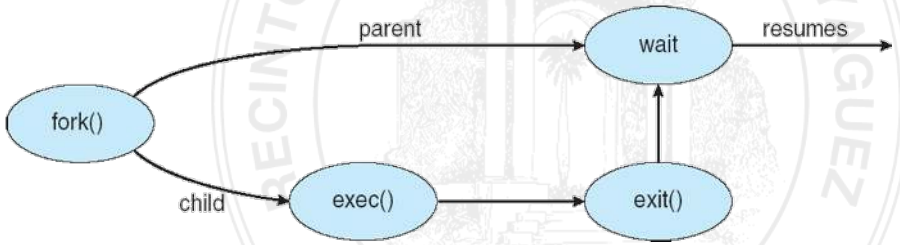
- Other system calls: **`open`**, **`close`**, **`read`**, **`write`** (see some examples)

  Note: the file descriptor is a number (0 …): the min value available when the **`open`** sc is executed.

Some Important System Calls for Process Management (7)

- **`pipe(int[] fd)`** - Creates an external structure that allows communication between parent and child process. Different processes may share a pipe by sharing the corresponding file descriptors. Parameter is an array of two entries. Entry **`fd[0]`** is set to the file descriptor for the output end of the pipe. Entry **`fd[1]`** is set to the file descriptor for the input end of the pipe.

Process Creation Stages



Process Creation in POSIX

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main (){
        pid_t pid;
        /* fork a child process */
        pid = fork();

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);
                printf("Child Complete!\n");
                exit (0);
        }
}
```


Process Creation in Win32

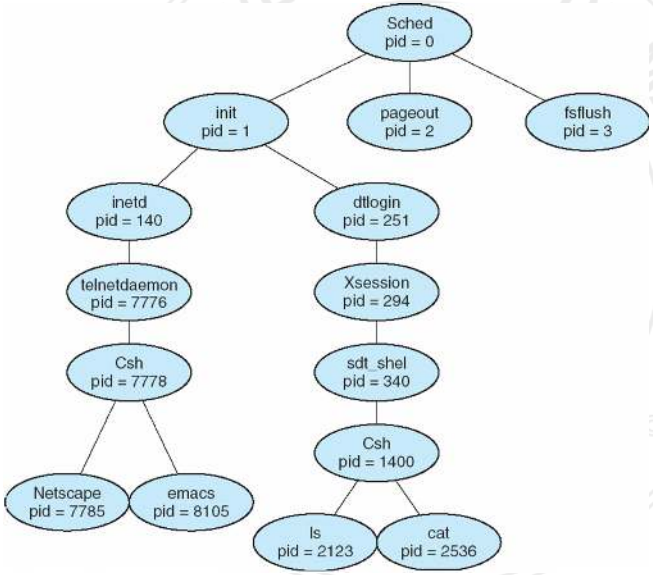
```
#include <stdio.h>
#include <windows.h>

int main(VOID){
          STARTUPINFO si;
          PROCESS_INFORMATION pi;
          // allocate memory
          ZeroMemory(&si, sizeof(si));
          si.cb = sizeof(si);
          ZeroMemory(&pi, sizeof(pi));

          // create child process
          if (!CreateProcess(NULL, //use command line
                    "C:\\WINDOWS\\system32\\mspaint.exe",
                    NULL, // don't inherit process handle
                    NULL, // don't inherit thread handle
                    FALSE, // disable handle inheritance
                    0, // no creation flags
                    NULL, // use parent's environment block
                    NULL, // use parent's existing directory
                    &si,
                    &pi))
                    {
                              fprintf(stderr, "Create Process Failed");
                              return -1;
                    }
                    // parent will wait for the child to complete
                    WaitForSingleObject(pi.hProcess, INFINITE);
                    printf("Child Complete");

                    // close handles
                    CloseHandle(pi,hProcess);
                    CloseHandle(pi.hThread);
}
```

Process Creation in Java

```
import java.io.*;

public class OSProcess {
        public static void main(String[] args) throws IOExeption {
                if (args.lenght != 1) {
                        System.err.println("Usage: java OSProcess <command>");
                        System.exit(0);
                }
                // args[0] is the command
                ProcessBuilder pb = new ProcessBuilder(args[0]);
                Process proc = pb.start();

                // obtain the input stream
                InputStream is = proc.getInputStream();
                InputStreamReader br = new BufferedReader(isr);

                // read what is returned by the command
                String line;
                while ( (line = br.readLine()) != null)
                        System.out.println(line);
                br.close();
        }
}
```

Java programs can generate new processes, which are executed separately from the JVM process. Such processes can only be used to execute existing executable programs on the particular OS. (Try the previous example. Execute "`java OSProcess cat`" and "`ps`" from another terminal window)

A Tree of Processes on Solaris



Process Termination (1)

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

Process Termination (2)

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - If a process terminates, all children must be terminated too (recursively) - **cascading termination**