

CSE 660: Introduction to Operating Systems

Process Concept

Presentation C

Process Concept

- A process is an operating system concept that captures the idea of a program in execution.
- A process has memory for its program and data, it has resources to compute with e.g. open files, and gets assigned a processor to do the computing.
- A process is always executing at some point in a program. When a given process is not running, since CPU is currently executing instructions of another process, the operating system keeps a copy of the hardware registers at the moment of interruption, program instructions, data and all other relevant information that describe a point in the computation that process is at. Thus, when this process should continue running all necessary information can be restored.

Process Concept (cont.)

- At any point in time, a processor executes some instruction, and the next instruction to be executed is either the next instruction in the flow of the user program (or operating system code) or if an exception (interrupt) happened, the next instruction to be executed is one with address that is loaded in PC during hardware processing of exception.
- Note that CPU can not distinguish between instructions of an operating system and user programs.

Process Implementation

- The basic idea of the implementation of processes in an operating system rests on a few simple ideas:
 1. the operating system is not a process but implements processes;
 2. each process is represented by a data structure called a process control block (PCB) and PCBs are kept in the operating system memory;
 3. Each process runs on the actual hardware with some restrictions:
 - CPU is shared between all processes and is switched between them, and the memory area of each process is restricted,
 - a process is prevented from using some machine instructions since it runs in user mode,
 - a process communicates with the operating system through system calls,
 - a running process can be interrupted at any time by hardware (I/O) interrupts (from a disk controller or timer).

Process Implementation (cont.)

4. When an exception occurs (an I/O interrupt from an I/O controller or an exception as result of those special instructions, e.g. Syscall in MIPS or Int n in Intel), the operating system gets control, handles the exception and then gives control back to some process (not necessary to interrupted process).

Functioning of Computer System: Example

1. Let us assume that there are 2 processes, Process A and Process B, ready to run and that currently CPU is executing instructions of the operating system (in kernel mode).
2. Operating system decides (based on its scheduling algorithm) that Process A should start running. Then, the operating system restores contents of registers from PCB_A , switches CPU mode of operation into user mode and makes sure that the next instruction CPU executes is the instruction in the code of Process A.
3. Now, CPU is executing (in user mode) instructions of Process A, thus Process A is running.
4. Let us assume that at the certain place in its code, Process A needs to read from a file. Thus, CPU will execute a code of the system call. The last instruction in a sequence of system call instructions is Syscall or Int n instruction, depending upon computer system. Execution of that special instruction will cause an exception, and CPU will start executing the appropriate routine in the operating system.

Functioning of Computer System: Ex. (cont.)

5. Now, CPU is executing (in kernel mode) code of operating system. The parameters of the system call are examined and the disc controller is instructed to read the certain block from the disc and to copy it (using DMA) into an operating system buffer.
6. The operating system declares Process A as a blocked process, thus ineligible for CPU, since Process A has to wait for a disk I/O to finish (that may take up to 20 milliseconds)
7. The operating system chooses one of ready to run processes, say Process B for running. Then, the operating system switches CPU mode of operation into user mode and makes sure that the next instruction CPU executes is an instruction in the code of Process B.
8. Now, CPU is executing (in user mode) code of Process B. We assume that Process B is such that for long time (longer than 20 milliseconds) its code would execute ordinary instructions without system calls.

Functioning of Computer System: Ex. (cont.)

9. At a certain point in time, while CPU is executing code of Process B, the disc controller finishes the task in 5. above. The controller then causes a hardware interrupt (by activating certain IRQ line). The interrupt will cause that CPU starts execution, in kernel mode, of the appropriate operating system routine. The operating system finds cause of the interrupt and after some processing it will declare Process A as a ready to run, thus eligible for CPU.
10. At this point, the operating system should decide which of ready to run processes should get CPU. Note that Process B is still in ready to run state, since it was interrupted during its normal execution.

Process States

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

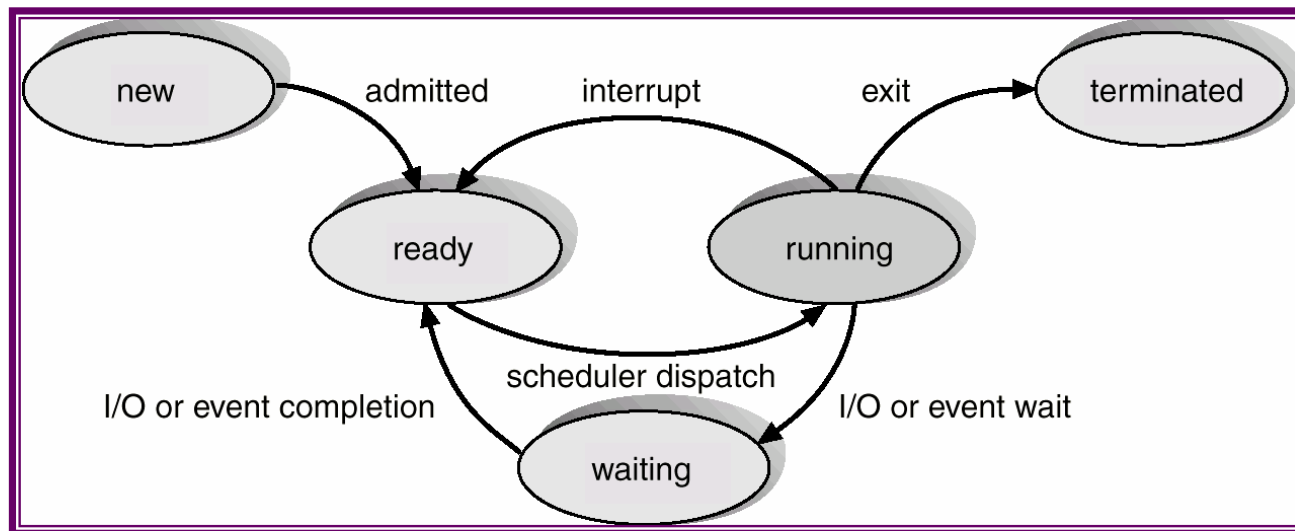


Figure 3.2 Diagram of Process States

Process Control Block (PCB)

Information associated with each process:

- Process state
- Process identification (Pid)
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- Pid of parent process

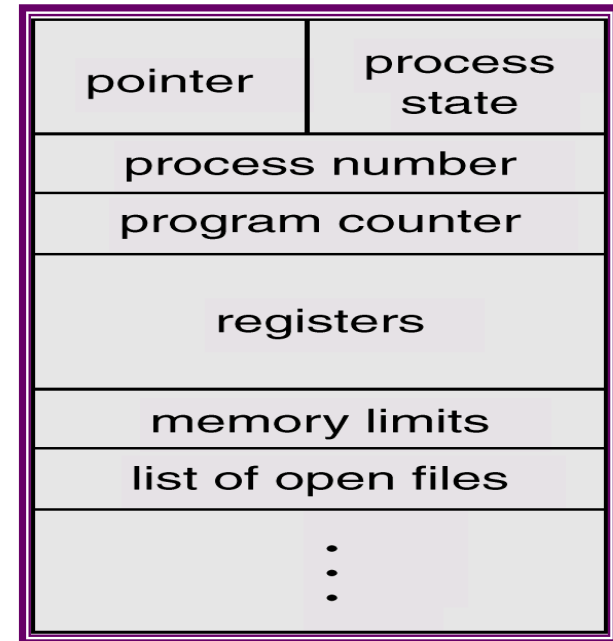


Figure 3.3

CPU Switch From Process to Process

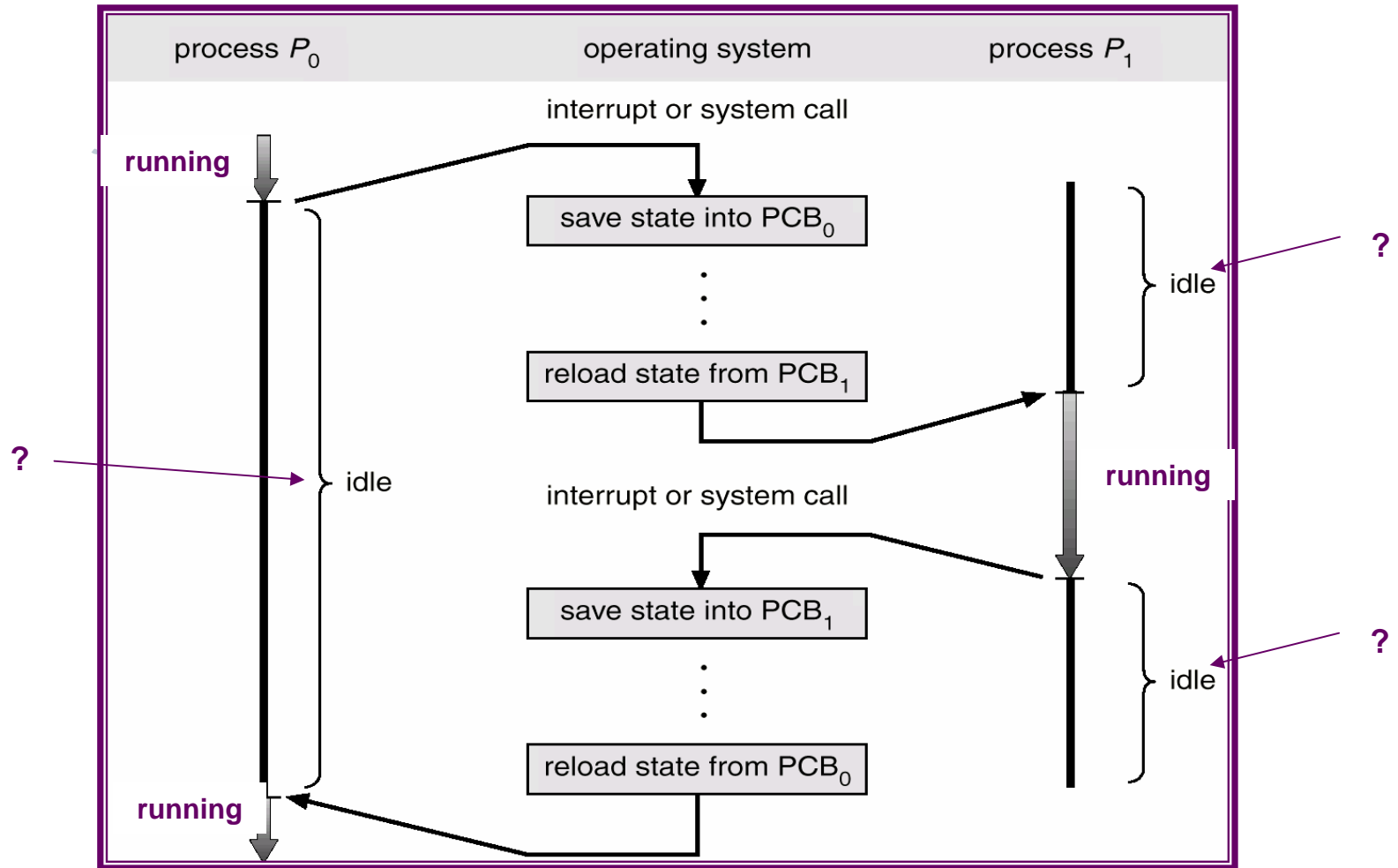


Figure 3.4

Unix Process Related System Calls

<u>System Call</u>	<u>Parameters</u>	<u>Returns</u>	<u>Notes</u>
fork	none	Pid or 0	Duplicated process
execv	programFile, args	None	Execute program in process
exit	return code	None	Destroy process
wait	return code address	pid	Wait for child process to exit

Unix Process Related System Calls (cont.)

`int fork(void);`

- A new process is created that is an exact copy of the process making the system call, i.e. a copy of the calling process is made and it runs as a new process. The copy is of the memory image of the calling process at the moment of the fork system call, not the program the calling process was started from.
- The new process does not start at the beginning but at the exact point of the fork system call.
- Thus, right after the fork there are two processes (the issuing process and newly created process) with identical memory images, and the only difference is the return value the operating system will give to the two returning forks.
- The return value to the parent is the process identifier of the new process. The return value to the child process is 0.

Unix Process Related System Calls (cont.)

`int execv(char * programName, char *argv[]);`

The program *programName* is loaded in the calling process address space. Thus, the old program in the calling process is overwritten and will no longer exist. The arguments are in the argument vector *argv*, which is an array of strings, that is, an array of pointers to characters. (There are 6 versions of exec system call)

`void exit(int returnCode);`

This system call causes a process to exit. The *returnCode* is returned to the parent process if it waits for its child process to terminate.

`int wait(int * returnCode);`

This system call will cause the calling process to wait until **any** process created by the calling process exits. The return value is the process identifier of the process that exited. The return code is stored in *returnCode*.

Unix Processes: Example 1

Write C code for a process A which creates another process with code from ProgB file and then process A exits.

ProgA

```
{  
int x; char *arg[1]={0};  
x=fork();  
if (x==0) execl("ProgB",arg);  
exit(3)  
}
```

In the example above, a child process does not terminate when its parent terminates and the **init** process “reparents” the orphan children. Note, the **init** process has Pid = 0.

Unix Processes: Example 2

Write C code for a process A which creates another process with code from ProgB file, then the process A waits for its child process to terminate, before it exits.

```
ProgA
{
  int x,y,z; char *arg[1]={0};
  x=fork();
  if (x==0) execv("ProgB",arg);
  y=wait(&z);
  exit(3)
}
```

```
ProgB
{
  -
  -
  exit(5); /*point A*/
  -
  -
  exit(1); /*point B*/
}
```

If the child process exits at point A, then z=5, while if it exits at point B then z=1.

Unix Processes: Example 3

This is the example from Figure 4.8 in the textbook. The example illustrates how Unix command **ls** can be issued from a program.

```
void main (int argc, char *argv [ ])
{
int pid
    pid = fork (); /* fork another process*/

    if (pid <0) { /* error*/ fprintf (stderr, "Fork failed")};

    else if (pid == 0) { /*child process*/ execlp ("/bin/ls", "ls",
NULL)};

        else { /*parent process*/
            wait(NULL) ; /*waits for child to terminate*/
            printf ("Child completed");
            exit(0); }
}
```

Unix Processes: Example 4

Write C code for a process A which creates another process with code from file "Compiler" with a list of 3 parameters passed. Then the process A waits for its child process to terminate, before it exits.

```
ProgA
{
    int x,y,z;
    char *argv[4]={"gcc","-o","fileTo Compile", 0};
    x=fork();
    if (x==0) execv("Compiler",argv);
    y=wait(&z);
    exit(0)
}
```

Unix Processes: Example 5

- Consider the following program:

```
{int x,y;  
x = fork();  
y = fork();  
if (x==0) execv ("A",...);  
if (y==0) execv ("B",...);  
-  
-  
- }
```

- This is the part of a code of a process P. When this code is executed some number of children will be created and initially all of them will have code P. This code or any part of it should be executed completely in all child processes, as well as in any grandchild (or grand-grand child) process. At the end of all those executions, indicate all processes created, parent-child- grandchild relationships, values of variables and the final code in each process.

Shell

- All operating system services are available through system calls. But in order to issue a system call you have to be running a program.
- If we wrote an interactive program that communicated with the user through terminal and with the operating system through the system calls we would make all the operating system services available interactively to a user sitting at a terminal.
- That is exactly what a Unix shell is, a process that provides a user interface to the operating system built on the system call interface.

Shell (cont.)

- The shell executes commands of the form:

`%commandName [arg [arg ...]] [&]`

This executes the executable program `commandName` unless it is `wait` or `cd`. The arguments are passed to the command. The shell waits for the command to complete unless the `&` is present in which case it returns immediately.

- Example: `%ls -a -l` (% is the shell prompt)
- The `wait` command waits for a program executed earlier with line ending in `&` to complete.
- The `cd` command changes current directory.

Shell (cont.)

- Many users are not aware of the process concept because they just run a program in a single process. Only a few users create processes using the actual system calls for process creation because normally other programs create processes for you. The shell gives your first process “for free”, that is it automatically created for you and started up running the program you requested. However, many users actually use multiple processes in their work.
- Effects of command: `%ls | more`
--> two processes and a pipe created
- A pipe is a channel (a buffer created by O.S.) between two processes in which one process can write a stream of characters, while the another can read them.

Shell (cont)

“Stripped-down and simple” shell code:

```
While (true)
```

```
{
```

```
    type_prompt();
```

```
    read_command(command, parameters);
```

```
    pid = fork();
```

```
    if (pid == 0) execv(command,parameters,0);
```

```
    x = wait(&y);
```

```
}
```

Basically, the shell collects words, puts them in an argument array and forks off a process to execute the command. It also handles output redirection, and pipes.

Init Process

- The `init` process is created automatically by the Unix as the first process in the system.
- The `init` process then creates an interactive process for each terminal in the system, as well as after any remote access to the system is initiated through a network.
- Each of those interactive processes is executing the login program (code) in it. After successful login, the interactive process obtains a code of the appropriate (for the logged in user) shell.

Process Interaction

Processes interact in two distinct ways:

1. **Indirectly**, through process competition for resources.

Problems to solve:

- scheduling of resources (CPU, memory, I/O controllers)
- deadlock (involving resource requests)

2. **Directly**, through process cooperation.

Problems to solve:

- inter-process communication through message exchange
- sharing common variables (critical section problem)
- synchronization, i.e. situations in which progress of one process depends upon the progress of another process.

Main Operating System Components

■ Process Management:

- process creation and deletion,
- process suspension and resumption,
- provision of mechanisms for:
 - a. process synchronization,
 - b. process communication,
 - c. deadlock handling.

■ Main Memory Management:

- keeping track of which parts of memory are currently being used and by whom,
- allocate and deallocate memory space as needed,
- deciding which processes to load when memory space becomes available.

Main Operating System Components (cont.)

■ File Management:

- file creation and deletion,
- directory creation and deletion,
- support of primitives for manipulating files and directories,
- mapping files onto secondary storage,
- file backup on stable (nonvolatile) storage media.

■ I/O System Management:

- a buffer-caching system,
- a general device-driver interface,
- drivers for specific hardware devices.

Main Operating System Components (cont.)

■ Secondary Storage (Disk) Management

- free space management
- storage allocation
- disk (head) scheduling

■ Protection System:

1. *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.

2. The protection mechanism must:

- distinguish between authorized and unauthorized usage.
- specify the controls to be imposed.
- provide a means of enforcement.

■ Networking

■ Command-Interpreter System

/* This program receives any number of parameters and prints them */

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
void main(argc, argv)
    char **argv; int argc;
{int x;
  x = 0;
  while (argc > x)
  { x++;
    printf ("argv[%d] = %s\n", x, *argv);
    argv++; }
  exit();
}
```