OPERATING SYSTEMS

PROCESSES

2 Processes

All modern computers often do several things at the same time. A modern operating system sees each software as a process. When a user PC is booted, many processes are secretly started. A process may be started up to wait for incoming e-mail, another process may run on behalf of the antivirus program to check periodically if any new virus definitions are available, explicit user processes may be running, printing files and burning a CDROM, all while the user is surfing the Web.

2 Processes

All this activity has to be managed, and a multiprogramming system supporting multiple processes comes in very handy here. In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While the CPU is running only one process, in the course of 1 second, it may work on several of them, giving the illusion of parallelism. Sometimes people speak of pseudo parallelism in this context, to contrast it with the true hardware parallelism of multiprocessor systems.

2 Processes

Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model that makes parallelism easier to deal with. That model, its uses, and some of its consequences form the subject of this chapter.

2.1 The Process Model

In this model, all software on a computer is organized into a number of sequential processes. A process is just an instance of an executing program, including the current values of

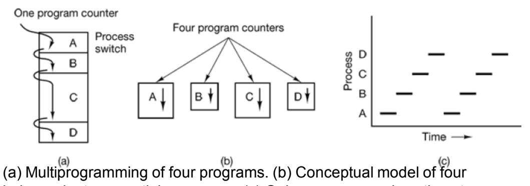
- the program counter,
- registers,
- and variables.

2.1 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.

2.1 The Process Model

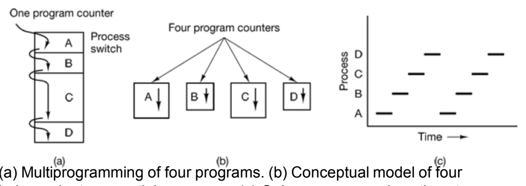
In Figure (a), there are four programs in memory, therefore we see four processes in Figure (b). Each with its own flow of control, and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter.



(a) Multiprogramming of four programs. (b) Conceptual model of four independent sequential processes. (c) Only one program is active at once.

2.1 The Process Model

When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory. In Figure (c) we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.



(a) Multiprogramming of four programs. (b) Conceptual model of four independent sequential processes. (c) Only one program is active at once.

2.1 The Process Model

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one. It is worth noting that if a program is running twice, it counts as two processes. The fact that two running processes happen to be running the same program does not matter; they are distinct processes. The operating system may be able to share the code between them so only one copy is in memory.

2.2 Process Creation

Operating systems need some way to make sure all the necessary processes exist. On the contrary of very simple systems, general-purpose systems need to create and terminate processes as needed during operation. There are four principal events that cause processes to be created:

1. System initialization.
2. Process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

2.2 Process Creation

In UNIX based OSs, there is only one system call to create a new process: "fork". This call creates an exact clone of the calling process. After the fork, the two processes (the parent and the child) have the same memory image, the same environment strings, and the same open files. Usually, the child process then executes a "exec" class system call to change its memory image and run a new program. The reason for this two-step process is to allow the child to manipulate its file descriptors.

2.2 Process Creation

In Windows, in contrast, 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, some parameters, various security attributes, control bits, priority information, window specifications, and a pointer about the newly created process caller. In addition to CreateProcess, Win32 has about 100 other functions for managing and synchronizing processes and related topics.

2.3 Process Termination

13

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

2.3 Process Termination

14

- Normal: 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.
- Error: an error caused by the process, often due to a program bug.
- Fatal error: the process discovers a fatal error.
- Killed by another: the process executes a system call telling the operating system to kill some other process.

2.4 Process Hierarchies

15

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. In UNIX, a process and all of its descendants form a process group together. 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. Individually, each process can use or ignore the signal.

2.4 Process Hierarchies

16

In contrast, Windows has no concept of a process hierarchy. All processes are equal.

The only hint of a process hierarchy is that when a child process is created, the parent gets process ID of the child to control it. However, it is free to pass this ID to some other process, thus invalidating the hierarchy.

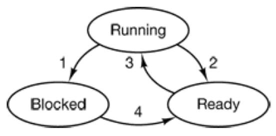
And, processes in UNIX cannot disinherit their children.

2.5 Process States

17

State diagram shows three states a process may be in:

1. Running (actually using the CPU at that instant).
2. Ready (temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

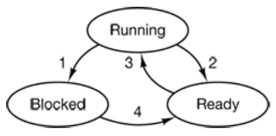


2.5 Process States

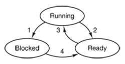
18

Four transitions are possible among these three states:

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

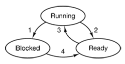


2.5 Process States



Transition 1 occurs when the operating system discovers that a process cannot continue right now. Usually when a process reads from a pipe or special file and there is no input available, the process is automatically blocked. Transitions 2 and 3 are caused by the process scheduler, a part of the operating system. Scheduling deciding which process should run when and for how long is an important OS task. Many scheduling algorithms have been devised to increase efficiency of the system and the processes.

2.5 Process States



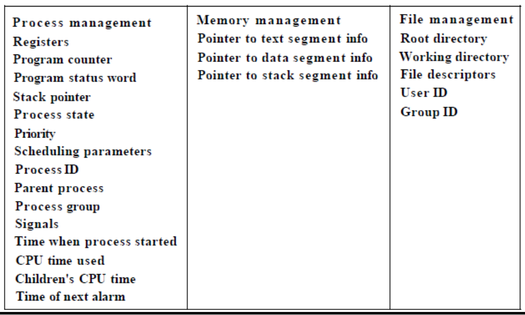
Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. Transition 4 occurs when the external event for which a process was waiting happens. If no other process is running at that instant, Transition 3 will be triggered and the process will start running. Otherwise it may have to wait in ready state for a while until the CPU is available.

2.6 Implementation of Process

To implement the process model, the operating system maintains an array of structures, called the process table, with one entry per process. Each entry is also called as process control block. An entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

2.6 Implementation of Process

Figure shows some of the key fields in a process table.



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		

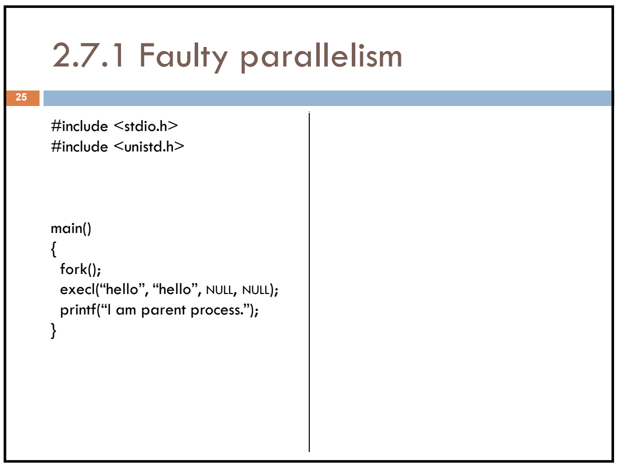
2.6 Implementation of Process

By the process table, it is possible to understand how multiple sequential processes are maintained on one CPU and many I/O device. Each I/O device is used via its interrupt vector which contains the address of the interrupt service procedure. Suppose that an user process is running when a disk interrupt happens. The user process's program counter, program status word, and registers are pushed onto the stack by the interrupt hardware. The computer then jumps to the address specified in the interrupt vector. That is all the hardware does. Then it is up to the interrupt service procedure.

2.7 System programming

To understand the processes, some C codes on system programming may be useful.

- Faulty parallelism
- Standard parallelism
- Serialized parallelism
- Murderer parent
- Zombie process
- Orphan process

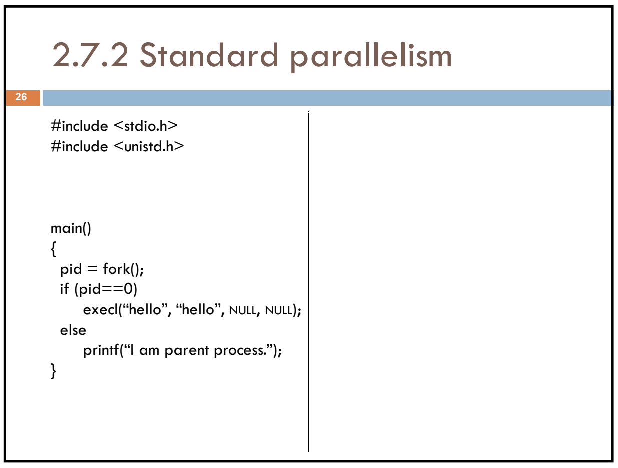


2.7.1 Faulty parallelism

25

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


main()
{
  fork();
  execl("hello", "hello", NULL, NULL);
  printf("I am parent process.");
}
```

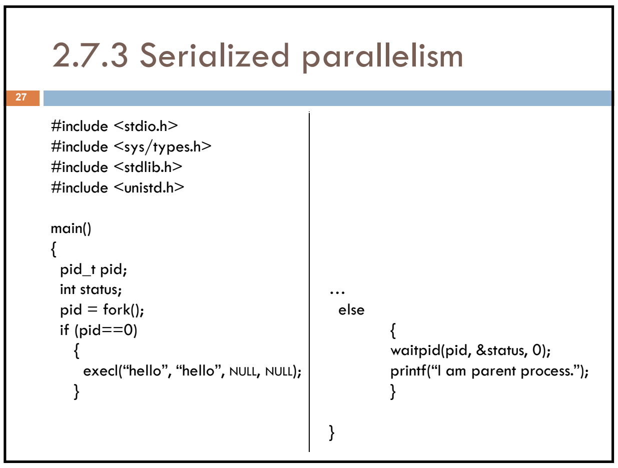


2.7.2 Standard parallelism

26

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


main()
{
  pid = fork();
  if (pid==0)
       execl("hello", "hello", NULL, NULL);
  else
       printf("I am parent process.");
}
```



2.7.3 Serialized parallelism

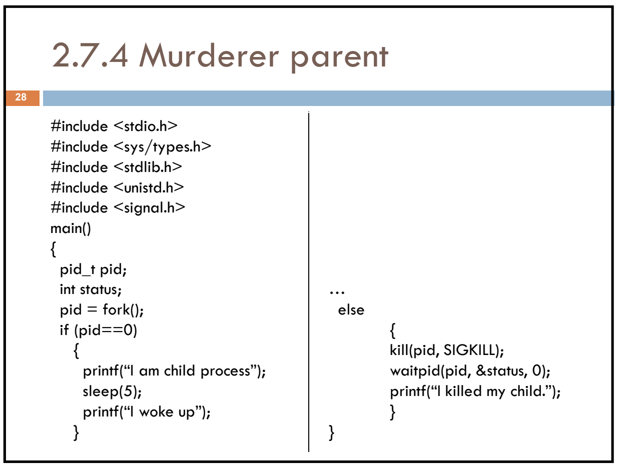
27

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

main()
{
  pid_t pid;
  int status;
  pid = fork();
  if (pid==0)
     {
       execl("hello", "hello", NULL, NULL);
     }
```

```
…
  else
            {
            waitpid(pid, &status, 0);
            printf("I am parent process.");
            }

}
```

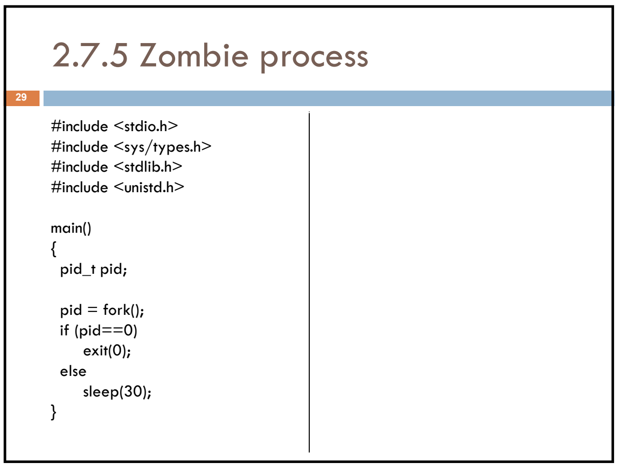


2.7.4 Murderer parent

28

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
main()
{
  pid_t pid;
  int status;
  pid = fork();
  if (pid==0)
     {
       printf("I am child process");
       sleep(5);
       printf("I woke up");
     }
```

```
…
  else
            {
            kill(pid, SIGKILL);
            waitpid(pid, &status, 0);
            printf("I killed my child.");
            }
}
```



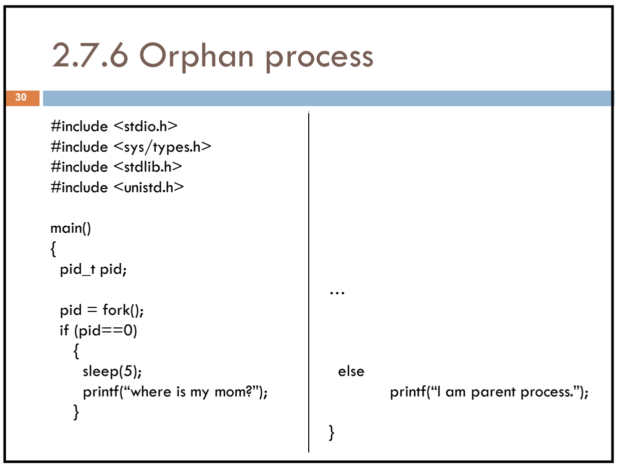
2.7.5 Zombie process

29

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

main()
{
  pid_t pid;

  pid = fork();
  if (pid==0)
       exit(0);
  else
       sleep(30);
}
```



2.7.6 Orphan process

30

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

main()
{
  pid_t pid;

  pid = fork();
  if (pid==0)
     {
       sleep(5);
       printf("where is my mom?");
     }
```

```
…



  else
            printf("I am parent process.");

}
```