CHAPTER 2

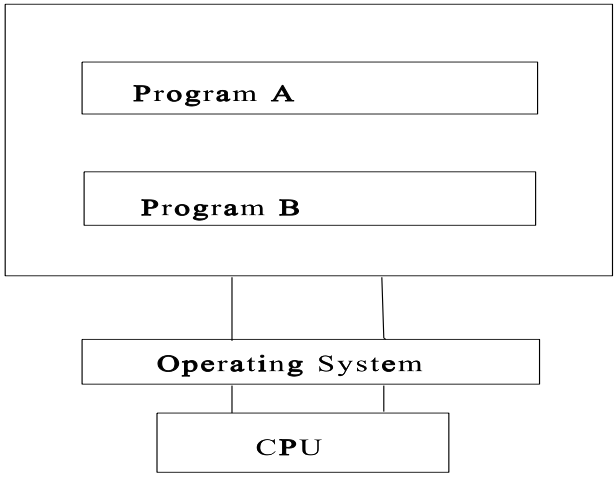
SHARED MEMORY PROCESSING

1.1 Multiprocessing

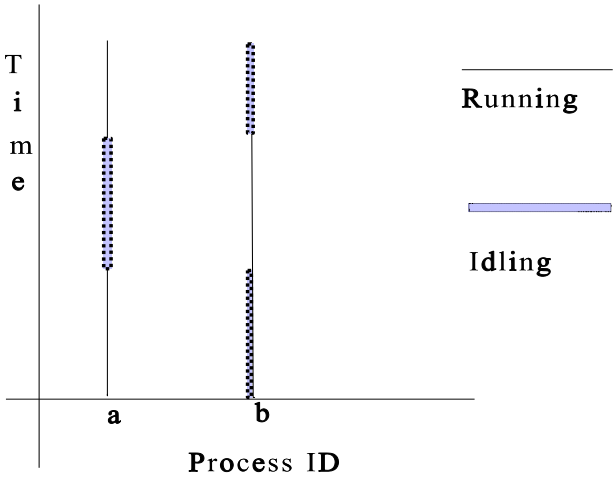
For the remainder of these notes, we assume that a multiprocessing, shared memory, synchronous computer with several processors, similar to *MARS* (*parallel computer with six processors*), is available. Such a machine is often also known as multiuser, multitasking computer. In such an environment, the operating system (OS) can serve several users (hence multiuser) who may have one or more programs running (hence multitasking). UNIX is an example of such an operating system. The OS controls the regions of memory the various CPUs can access. The processors communicate through common memory areas. Parallel programming on such a machine would then require careful attention to load balancing, contention etc., discussed under **MIMD** architectures.

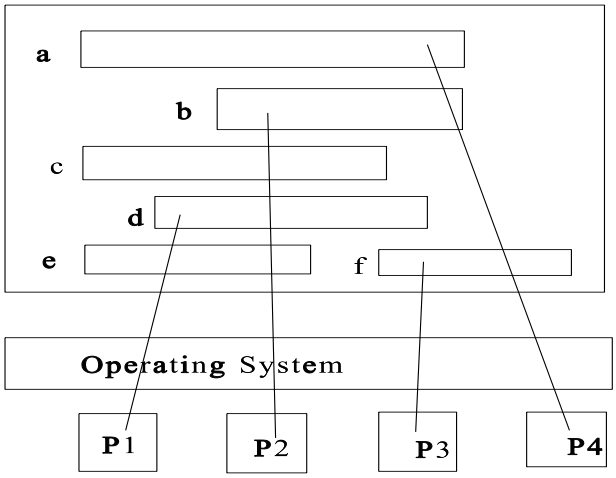
To understand parallel processing better, consider the case of a computer with a single CPU running on a multiuser, multitasking OS. It will soon become clear (when we discuss *processes*) that from the point of view of programming, it is immaterial whether we have a single or multiprocessor computer. Parallel programs can be written on a single CPU machine; however, because of overhead involved, they would run quite slow with no advantage gained. In such a computer, at a given moment, one or more users have programs running, all of which have not terminated. The function of OS is to ensure that all programs have access to the CPU and that the CPU accesses only the memory of the executing program and not any other.

Example: The diagram on the right illustrates a case with two programs A and B "running". At a given instant, only A or B is running. After A runs for a while, the operating system, on its own "idles" A and switches to B. When B is running, no reference may be made to A. (In UNIX, a program can be in several modes: asleep, ready-to-run, swapped-to-disk, etc.).

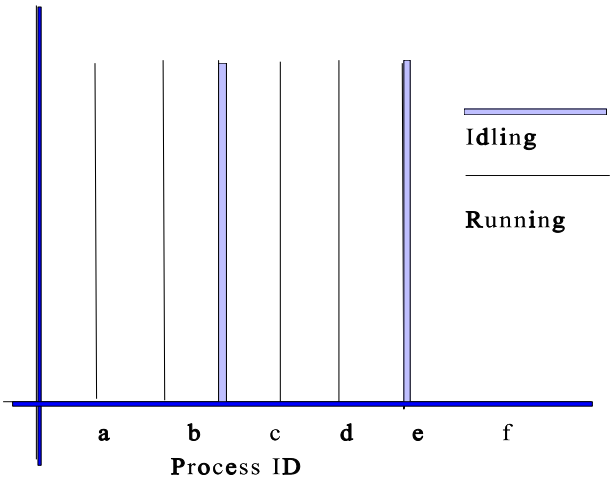


Time line diagrams can be drawn to indicate running status of programs. In our diagram, solid lines indicate running programs while broken (wavy rectangular) lines indicate idling programs. It is important to observe that the operating system has to manage other functions as well, such as I/O.

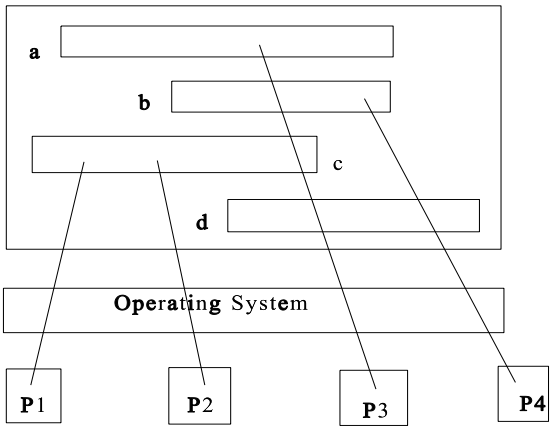




In the case of a computer with several processors, the situation is not very much different (see diagram on left). When there are several programs, each program is run on a separate processor. If there are not enough processors to run all the programs, some would be idling. In this model, no processor is favored over the other. The operating system schedules the programs to run on any available processor, and may itself be running on a processor. In the example, programs c and e are idle. The other four programs a, b, d, and f are using the various four processors as shown (e.g., a is running on P4). A corresponding time-line diagram is shown on the right. It is important to note that a CPU can run *only one program at a time*.

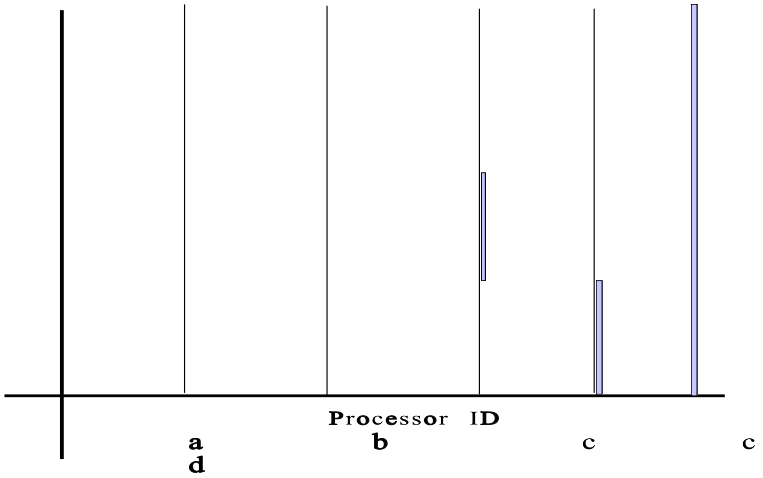


Parallel programs



In this case, a single program which contains code that can be apportioned among several processors is run on one or more processors. The OS may be running other programs as well (including itself). When a single program is run on more than one processor, each processor does only part of the calculation. In the following example, a machine

with four processors is running four programs. One program, c, is actually run on two processors, P1 and P2. Of the others, a is run on P3, b on P4, and d is idling. Notice again that a CPU can run only one program at a time.



The time-line diagram is shown on the right. It is important to note that while c is being run on two processors P1 and P2, some data may be shared, some private. When sharing data, they can both access the same memory location but only one at a time.

Contention

A fundamental maxim about the operating system is that all processes are *randomly* scheduled. That is at any given time, a process can be idled and restarted at a later time. It is not necessary that even one process would be scheduled at a given instant. The operating system does this in a way so as to maximize its processing resources. In particular, the amount of time taken to run a process is independent of the process itself; it is quite possible that amongst two processes requiring almost identical amount of work, because of idling, one process may take longer to finish than the other. This gives rise to the phenomenon known as *contention.*

When several processes share a variable, because of operating system idlings, it is

possible that the final value of a shared variable may be *changed incorrectly*. In fact the final value of such a variable will very much depend upon the order in which the various processes altered it. The last alteration is permanent. This is better understood by the following example:

```
    int i, id;
    i : shared; {pseudocode to indicate i is shared}
{

    i = 0;
    //fork 2 processes
    i = i + 1;
    cout<<"id = "<< id<<"  i = "<<i<<endl;
    //join 2 processes
}
```

When run, this program can produce several outputs. We consider three of those:

Output 1	Output 2	Output 3
id = 0 i = 1	id = 1 i = 1	id = 1 i = 1
id = 1 i = 2	id = 0 i = 2	id = 1 i = 1

To explain these outputs, consider the main line in the program:

```
    i = i + 1;
```

This high level code is equivalent to, in general, the following three assembly language instructions executed by the CPU:

(a) Load value of i into a CPU register.

(Symbolically, R <– i)

(b) Increment register by 1 (R^+)

(c) Store i back in memory (i <– R)

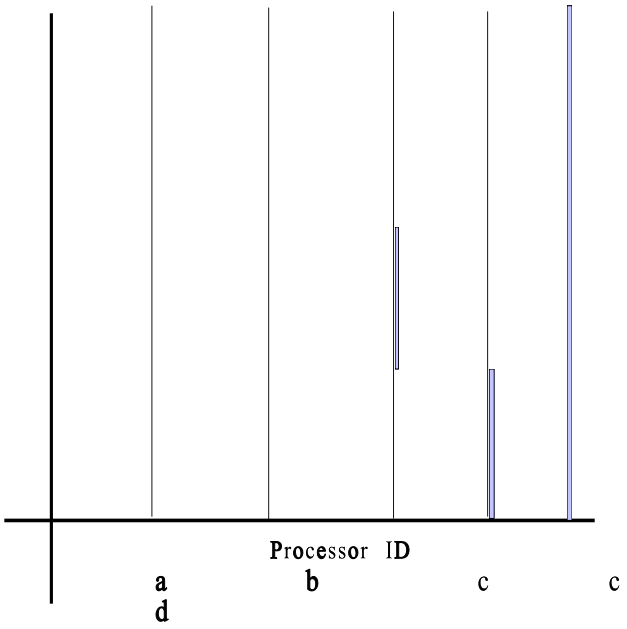
Assuming each of these operations is atomic, the operating system can idle a process at the end of any of these steps. In particular, conside the following scenario. This explains the first output.

FORK K PROCESS O PROCESS 1

Do (a) R <– 0 idle
Do (b) R^+ idle
Do (c) i = 1 idle

Print values Do (a) R <– 1
Wait Do (b) R^+
Wait Do (c) i = 2
Wait Print values

JOIN K Finish program Kill



The time-line diagram on the right explains this better.

The second output is the result of Processes 0 and 1 reversing their roles.

More interesting is the third output. It is the result of the following scenario:

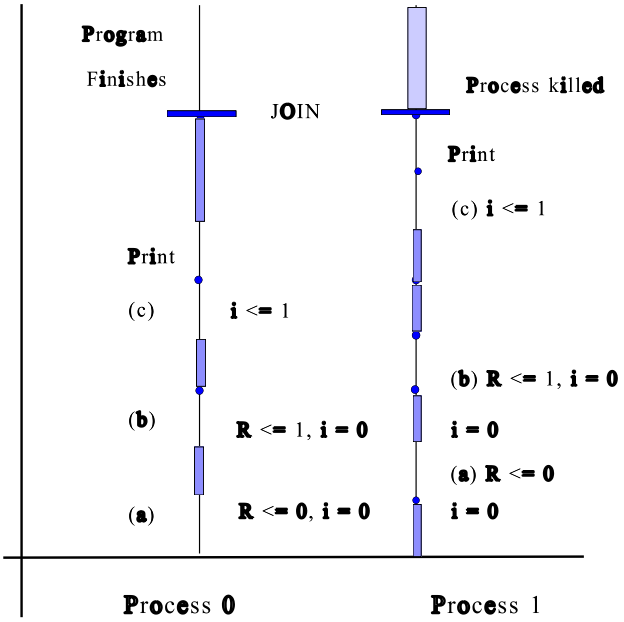
1. After forking, Process 1 is idled. Process does (a) and then is idled. Value of i is 0.

2. Process 1 wakes up, does (a), idles. Process 0 wakes up, does (b), and then is idled. Note that the value of i is still 0.

3. Process 1 wakes up, does (b), and then is idled. Process 0 wakes up, does (c) and then is idled. Value of i has just been changed to 1.

4. Process 1 wakes up, does (c), and then is idled. Value of i is still 1. Process 0 wakes up, does the printing and then executes join. It waits for Process 1 to finish.

5. Process 1 wakes up, does the printing, executes join and then dies. Process 0 finishes the program.



Again the time-line diagram illustrates the situation better.

The cause of the problem is the fact that two processes are trying to get access to shared memory and change the value there. One says they are *contending* for the same memory space; hence the name contention.

Semaphore: A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a multiprogramming operating system. Semaphores are Operating System resources that can be used to avoid contentions.