Processes

Introduction to Processes

The most central concept in any operating system is the process: an abstraction of a running program. Consider a user PC. When the system is booted, many processes are secretly started, often unknown to the user. For example, a process may run on behalf of the anti-virus program to check periodically if any new virus definitions are available. In addition, explicit user processes may be running, printing files and burning a CD-ROM, all while the user is surfing the web. All this activity is managed by a multiprogramming OS supporting multiple processes.

In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, 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 pseudoparallelism in this context, to contrast it with the true hardware parallelism of multiprocessor systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, OS designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with.

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, or just processes for short. A process is just an instance of an executing program, including the current values of the program counter, registers, and variables (Program counter: a special register visible to the programmer which contains the memory address of the next instruction to be fetched. After the instruction has been fetched, the program counter is updated to point to its successor. Another register is the stack pointer, which points to the top of the current stack in memory). Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process. This rapid switching back and forth is called multiprogramming.

![Diagram of multiprogramming](attachment:figure1.png)

Figure 1
In Fig. 1(a) we see a computer multiprogramming four programs in memory. In Fig. 1(b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. In Fig. 1(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.

The difference between a process and a program is subtle, but crucial. An analogy may help here. Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake. Now imagine that the computer scientist’s son comes running in screaming his head off, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

**Process creation**

Operating systems need some way to create processes and terminate processes as needed during operation. There are four principal events that cause processes to be created:

1- System initialization. When an OS is booted, typically several processes are created some of them foreground (processes that interact with users and perform work for them), some others background (not associated with particular users, but instead have some specific function) processes.

Processes that stay in the background to handle some activity such as e-mail, web pages, news, printing, and so on are called deamons.

In UNIX, the ps program can be used to list the running processes; in Windows the task manager can be used.

2- Execution of a process creation system call by a running process. Often a running process will issue system calls to create one or more new processes to help it do its job.

3- A user request to create a new process. In interactive systems, users can start a program by typing a command or (double) clicking an icon. Taking either of these actions starts a new process and runs the selected program in it.

4- Initiation of a batch job. This situation applies only to the batch systems found on large mainframes.
Technically, in all these cases, a new process is created by having an existing process execute a process creation system call.

In UNIX, 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.

In Windows, in contrast, a single Win32 function call, **CreateProcess**, handles both process creation and loading the correct program into the new process.

**Process Termination**

After a process has been created, it starts running and does whatever its job is. Sooner or later the new process will terminate, usually due to one of the following conditions:

1- Normal exit (voluntary). Process completed its work.
2- Error exit (voluntary). The process discovers a fatal error. e.g. user types `cc foo.c` and there is no file named `foo.c`
3- Fatal error (involuntary). An error is caused by a process, often due to a program bug.
4- Killed by another process (involuntary). A process executed a system call telling the OS to kill some other process. (In UNIX this call is `kill`, the corresponding Win32 function is `TerminateProcess`)

**Process States**

Sometimes one process may generate some output that another process uses as input. Depending on relative speeds of the two processes, it may happen that one is ready to run, but there is no input waiting for it. It must then block until some input is available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In Figure 2 we see a state diagram showing the three states a process may be in:

1. Running (actually using the CPU at that instant)
2. Ready (runnable; temporarily stopped to let another process run)
3. Blocked (unable to run until some external event happens)
In the first two cases the process is ready to run, only in the second one, there is temporarily no CPU available for it. The process in the third state cannot run, even if the CPU has nothing else to do. Four transitions are possible among these three states, as shown in the figure.

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

Transition 1 occurs when the operating system discovers that a process cannot continue. More commonly, when a process reads from a 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, without the process even knowing about them. 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 (such as the arrival of some input) happens.

**Modeling Multiprogramming**

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20% of the time it is sitting in memory, with five processes in memory at once, the CPU should be busy all the time. This model is unrealistically optimistic, however, since it tacitly assumes that all five processes will never be waiting for I/O at the same time.

A better model is to look at CPU usage from a probabilistic point of viewpoint. Suppose that a process spends a fraction $p$ of its time waiting for I/O to complete. With $n$ processes in memory at once, the probability that all $n$ processes are waiting for I/O (in which case the CPU will be idle) is $p^n$. The **CPU utilization** is then given by the formula

$$\text{CPU utilization} = 1 - p^n$$
Figure 3 shows the CPU utilization as a function of $n$, which is called the degree of multiprogramming.

![CPU utilization graph](image)

Figure 3

From the figure it is clear that if processes spend 80% of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10%.

For the sake of complete accuracy, it should be pointed out that the probabilistic model just described is only an approximation. It implicitly assumes that all $n$ processes are independent. Even though this model is simple-minded, it can nevertheless be used to make specific, although approximate, predictions about CPU performance.

Suppose, for example, that a computer has 512 MB of memory, with the operating system taking 128 MB and each user program also taking up 128 MB. These sizes allow three user programs to be in memory at once. With an 80% average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - 0.8^3$ or about 49%. Adding another 512 MB of memory allows the system to go from three-way multiprogramming to seven-way multiprogramming, thus raising the CPU utilization to 79%. In other words, the additional 512 MB will raise the throughput by 30%. Adding yet another 512 MB would only increase CPU utilization from 79% to 91%, thus raising the throughput by only another 12%. Using this model the computer’s owner might decide that the first addition is a good investment but that the second is not.
Interprocess Communication (IPC)

Processes frequently need to communicate with other processes. Below we will look at some issues related to this interprocess communication.

Race Conditions

In some operating systems, processes that are working together often share some common storage that each one can read and write. The shared memory may be in main memory or it may be a shared file. Let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name in a special spooler directory. Another process, a print daemon, periodically checks to see if there are any files to be printed, and if there are it prints them and removes their names from the directory.

Use Figure 4 as a visual guide to the following example. Imagine that our spooler directory has a large (potentially infinite) number of slots, numbered 0, 1, 2,…, each one capable of holding a file name. Also imagine that there are two shared variables, out, that points to the next file to be printed, and in, which points to the next free slot in the directory. These variables might be kept on a two-word file available to all processes. Assume at a certain instant, we have a situation depicted below. Process A reads in and stores the value 7 in a local variable called next_free_slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B. Process B also reads in and also gets 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things. Eventually process A runs again, starting from the place it left off. It looks at next_free_slot, finds 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes next_free_slot+1 which is 8 and sets in to 8. The printer daemon will not notice anything wrong, but process B will never get any output. Situations like this, where two or more processes are reading or writing some shared data and final result depends on who runs precisely when, are called race conditions. The difficulty above occurred because process B started using one of the shared variables before process A was finished with it.
**Critical Regions**

The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else, is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what is we need is mutual exclusion, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

The part of the program where the shared memory is accessed is called the critical section or critical region. If we could arrange matters such that no two processes were ever in their critical sections at the same time, we could avoid race conditions. We need four conditions for having parallel processes cooperate correctly and efficiently using shared data:

1. No two processes may be simultaneously inside their critical sections.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical section may block other processes.
4. No process should have to wait forever to enter its critical section.

Below we will examine various proposals for achieving mutual exclusion, so that if one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

**Disabling Interrupts.** The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupt can occur. This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose one of them did it, and never turned them on again. That could be end of the system.

**Lock Variables.** As a second attempt, let us look for a software solution. Consider having a shared, single variable, lock, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Unfortunately, this idea contains exactly the same disadvantage that we saw in the spooler directory. Suppose one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again it will also set the lock to 1, and two processes will be in their critical regions at the same time.

**Strict Alternation.** According to this method, the integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

```c
while(TRUE){
    while(turn!=0);
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```c
while(TRUE){
    while(turn!=1);
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)
Initially, process 0 inspects \textit{turn}, finds it to be 0, and enters its critical region (see case (a)). Process 1 also finds it to be 0, and therefore sits in a tight loop continually testing variable to see when it becomes 1 (see case (b)). Continuously testing a variable for some value to appear is called \textbf{busy waiting}. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used.

When process 0 leaves the critical region, it sets \textit{turn} to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with \textit{turn} set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting \textit{turn} to 1. At this point \textit{turn} is 1 and both processes are executing in their noncritical regions. Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because \textit{turn} is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets \textit{turn} to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other. This situation violates condition 3 set out above: one process may be blocked by a process not in its critical section. In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

\textbf{Peterson’s Solution}

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch Mathematician, T. Dekker, was the first one to devise the software solution to the mutual exclusion problem that does not require strict alternation.

In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker’s solution obsolete. Peterson’s algorithm is given below.

```c
#define FALSE 0
#define TRUE 1
#define N 2  /* number of processes */

int turn;    /* whose turn is it? */
int interested[N];  /* all values initially 0 (FALSE) */

void enter_region(int process) /* process is 0 or 1 */
{
    int other;  /* number of the other process */
    other = 1 - process; /* position 1 */
    interested[process] = TRUE; /* position 2 */
    turn = process; /* position 3 */
    while(turn == process && interested[other] == TRUE); /* null statement */
}

void leave_region(int process)
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```
The following explains how the algorithm works when interrupted at position 1:

Note that `turn` and `interested` are **global** variables, while `other` is a **local** variable.

<table>
<thead>
<tr>
<th>process 0</th>
<th>process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>other = 1</td>
<td>other = 0, interested[1] = TRUE, turn = 1</td>
</tr>
<tr>
<td>stop, interrupt by process 1</td>
<td>check while condition turn==1 (true) and interested[0]==TRUE (false)</td>
</tr>
<tr>
<td></td>
<td>then process 1 enters CS (critical section)</td>
</tr>
<tr>
<td></td>
<td>stop, interrupt by process 0</td>
</tr>
<tr>
<td>process 0 gets the CPU now</td>
<td>process 0 keeps working then finishes CS job, after that it sets interested[1] = FALSE and gives CPU to process 0</td>
</tr>
<tr>
<td>interested[0] = TRUE, turn = 0</td>
<td></td>
</tr>
<tr>
<td>check while condition turn==0 (true) and interested[1]==TRUE (true)</td>
<td></td>
</tr>
<tr>
<td>then busy wait and give CPU back to process 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>check while condition turn==1 (false) and interested[0]==TRUE (true)</td>
</tr>
<tr>
<td></td>
<td>then process 1 enters CS (critical section)</td>
</tr>
</tbody>
</table>

The following explains how the algorithm works when interrupted at position 2:

<table>
<thead>
<tr>
<th>process 0</th>
<th>process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>other = 1, interested[0] = TRUE</td>
<td>other = 0, interested[1] = TRUE, turn = 1</td>
</tr>
<tr>
<td>stop, interrupt by process 1</td>
<td>check while condition turn==1 (true) and interested[0]==TRUE (true)</td>
</tr>
<tr>
<td></td>
<td>then process 1 waits and gives the CPU back to process 0</td>
</tr>
<tr>
<td>process 0 gets the CPU now</td>
<td></td>
</tr>
<tr>
<td>turn = 0</td>
<td></td>
</tr>
<tr>
<td>check while condition turn==0 (true) and interested[1]==TRUE (true)</td>
<td></td>
</tr>
<tr>
<td>then busy wait and give CPU back to process 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>check while condition turn==1 (false) and interested[0]==TRUE (true)</td>
</tr>
<tr>
<td></td>
<td>then process 1 enters CS (critical section)</td>
</tr>
</tbody>
</table>
check while condition
turn==0 (**true**) and interested[1]==TRUE (**true**) then *busy wait* and give CPU back to process 1

| stop, interrupt by process 0 |

process 1 keeps working then finishes CS job, after that it sets interested[1] = FALSE and gives CPU to process 0

check while condition
turn==0 (**true**) and interested[1]==TRUE (**false**) then process 0 enters CS (critical section)

| process 1 keeps working then finishes CS job, after that it sets interested[1] = FALSE and gives CPU to process 0 |

The following explains how the algorithm works when interrupted at position 3:

<table>
<thead>
<tr>
<th>process 0</th>
<th>process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>other = 1, interested[0] = TRUE, turn = 0</td>
<td>other = 0, interested[1] = TRUE, turn = 1</td>
</tr>
<tr>
<td>stop, interrupt by process 1</td>
<td>check while condition turn==1 (<strong>true</strong>) and interested[0]==TRUE (<strong>true</strong>) then process 1 waits and gives CPU back to process 0</td>
</tr>
</tbody>
</table>

process 0 gets the CPU now

check while condition
turn==0 (**false**) and interested[1]==TRUE (**true**) then process 0 enters CS (critical section)

| process 0 keeps working then finishes CS job, after that it sets interested[0] = FALSE and gives CPU to process 1 |
| check while condition turn==1 (**true**) and interested[0]==TRUE (**true**) then process 1 waits and gives CPU back to process 0 |

Notes:
- Once a process works on a critical section, the other process has no chance to get into the critical section.
- This is a pure software solution, and it does not get any support from hardware or any structure design.
- However, this solution may cause "busy waiting".
His program uses two procedures \texttt{enter\_region} and \texttt{leave\_region}. A process once entered into critical region will be in this region until using procedure \texttt{leave\_region}. Other processes must wait during this time. This method also regulates the case when two processes call \texttt{enter\_region} simultaneously. Both will store their process number in shared variable. Whichever store is done last is the one that counts, the first one is lost. According to this method, program allows the process which read shared variable first to enter its critical region. Peterson’s method has the defect of requiring busy waiting, some interprocess communication primitives block instead of wasting the CPU time when they are not allowed to enter their critical sections. The two procedures \texttt{SLEEP} and \texttt{WAKEUP} could be used to solve busy waiting problem. \texttt{SLEEP} is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The \texttt{WAKEUP} awakes the process.