3.0 Inter-process Communication & Synchronization

- Multiprogramming allows multiple processes to run concurrently, even on a system with only a single processor. OS that support multiprogramming provide for these concurrent applications.
- IPC and synchronization primitives are important to OS designers both for use within the OS and as service to provide to applications running on the OS.
- A ***co-operating process*** is one that shares information with or controls the sequence of instructions executed by another process. Information sharing may be achieved by using shared memory locations, shared files, or by using OS services provided for that purpose.

  Advantages of process cooperation include:

  Information sharing: Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.

  Computation speed-up: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

  Modularity: We may want to construct the system in a modular fashion, dividing the system functions into separate processes**.**

  Convenience: Even an individual user may have many tasks to work on at one time. For instance, a user may be editing, printing, and compiling in parallel.

3.1 Inter-process communication (IPC)

- **Definition:** a set of interfaces that allow programmers to communicate between processes and allow programs to run concurrently.

  The communication is achieved through the following techniques:

a) Files

- Mostly used; information written into a file by one process can be read by another.
- Used by both concurrent and non-concurrent processes
- Amount of information that can be shared is limited only by the file size capacity of the file system.

b) Shared memory

- Integral to threaded systems.
- Some OS support a supervisor call that creates a shared memory space.
- On other systems, the file system allows the creation of a RAM disk – a virtual disk created from memory space. Files stored on the RAM are actually stored on memory.

c) Signalling

- A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred

d) Messaging (message queuing)

Message passing is a type of communication between processes. Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Communications are completed by the sending of messages (functions, signals and data packets) to recipients

Note: cooperating processes face two major problems: starvation and deadlock.

3.2 Process Synchronization

- **Definition**: the process by which OS have to regulate the order in which processes access data. It is a major concern in multiprogramming.
- If concurrent processes are entirely different subsets of systems resources, there is no problem, but if concurrent processes attempt to make use of the same system resource at the same time e.g. accessing a particular file, then there would be a problem.

3.2.1 Synchronization issues

A situation where the correctness of the computation performed by co-operating processes can be affected by relative timing of the processes execution is called a ***race condition.*** To be considered correct, co-operating processes may not be subjected to race conditions. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the critical section. To make such a guarantee, we require some form of synchronization of the processes.

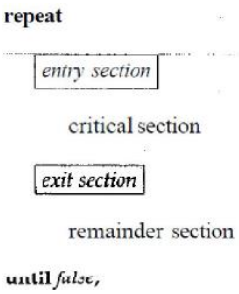
The Critical section

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute

The Critical section problem

Consider a system consisting of *n* processes $\{P_0, P_1,...,P_{n-1}\}$. Each process has a segment of code, called a *critical section,* in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is *mutually exclusive* in time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.



Such mechanisms operate under the following ***conditions*****:**

***Relative speed*:** no assumption may be made about the relative execution speeds of the co-operating processes except execution of each process proceeds at some non-zero speed.

No indivisibility: no assumption may be made about the indivisibility of machine instructions except that which is included as part of the definition of a synchronization primitive. Instructions may be broken up into a number of atomic actions; execution of other processes may be interleaved among those atomic units. However it may be assumed that access to shared memory locations is serialized so that no two processes may be reading or writing to a memory location simultaneously.

Bounded use: a process only executes in its critical section for some indefinite time and may not terminate while in its critical section.

- Such mechanisms of control must satisfy the following conditions:
  ***Mutual Exclusion*:** Only one process can be in the critical section at any time.
  Progress: Decision on who enters critical section cannot be indefinitely postponed.
  Bounded Wait: when a process requests access to critical section, a decision that grants it access may not be delayed indefinitely; a process may not be denied access because of starvation or deadlock.

3.2.2 Mutual Exclusion mechanisms

Mutual exclusion (often abbreviated to **mutex**) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

There exist both software and hardware solutions for enforcing mutual exclusion.

- ***Hardware solutions*** include: Interrupt disabling, *TestAndSet* instruction and Swap instruction.
- ***Software solutions*** include: wait and signal, semaphores, Dekker's algorithm, Peterson's algorithm, Bakery algorithm and Monitors.

a) Interrupt disabling

Each process disables all interrupts just after entering in its critical section and re-enables all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

```
DisableInterrrupt()
// critical section
EnableInterrrupt()
```

Advantage: easily implemented and incorporated into software.

Disadvantages

- It only works in a single processor environment.
- Interrupts can be lost if not serviced promptly. A process remaining in its critical section for any longer than a brief time would potentially disrupt the proper execution of I/O operations.
- Exclusive access to the CPU while in the critical section could inhibit achievement of the scheduling goals.
- A process waiting to enter into its critical section could suffer from starvation.

b) TestAndSet instruction
Instruction used to both test and (conditionally) write to a memory location as part of a single atomic (*i.e.* non-interruptible) operation. This means setting a value, but first performing some test (such as, the value is equal to another given value).
c) Swap instruction
It also executes atomically, like *TestAndSet* instruction.

d) Wait and signal
The OS can implement the primitives ***wait*** and ***signal*** to *create* a functionality to counter busy-waiting so as to increase performance benefits. Assumed in their implementation is a ***processQueue*** object with operations ***insert*** and ***remove***.

e) Semaphores
A *semaphore* is a nonnegative integer variable upon which two atomic operations are defined: ***P*** and ***V*** (named for the Dutch words ***proberen*** (test) and ***verhogen*** (to increment).

f) Dekker's Algorithm
A concurrent programming algorithm for mutual exclusion that allows two processes (threads) to share a single-use resource without conflict, using only shared memory for communication. It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

g) Peterson's Algorithm
It is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

h) Bakery algorithm
Lamport's bakery algorithm is a computer algorithm devised by computer scientist Dr. Leslie Lamport, which is intended to improve the robustness of multiple thread-handling processes by means of mutual exclusion.

Discussion
Each thread only writes its own storage, only reads are shared. It is remarkable that this algorithm is not built on top of some lower level 'atomic' operation, e.g. The original proof shows that for overlapping reads and writes to the same storage cell only the write must be correct. The read operation can return an arbitrary number. Therefore this algorithm can be used to implement mutual exclusion on 'memory' that lacks synchronization primitives, e.g., a simple shared SCSI disk between two computers.
The necessity of variable *Entering* might not be obvious as there is no 'lock' around lines 7 to 13.

i) Monitors
A **monitor** is an approach to synchronize two or more computer tasks that use a shared resource, usually a hardware device or a set of variables. With monitor-based concurrency, the compiler or interpreter transparently inserts locking and unlocking code to appropriately designated procedures, instead of the programmer having to access concurrency primitives explicitly.