Process Synchronization

Concurrent Processes

- Two processes are concurrent if their execution can overlap in time. In a single processor system, physical concurrency can be due to concurrent execution of the CPU and an I/O.
- Logical concurrency is obtained, if a CPU interleaves the execution of several processes.

Process Relationship

- Since processes may be executing on the same physical computer, they will compete for processor time, hardware resources and slow down each other but other than that they operate independently.
- But sometimes processes communicate with each other many reasons and in various ways and the communication may be repeated many times. Processes have to compete and coordinate with other for resources.

Two fundamental relations among concurrent process:

- Competition: By virtue of sharing resources of a single system all concurrent processes compete with each other for allocation of system resources needed for their operation.
- Cooperation: A collection of related processes that represent a single logical application cooperate with each other by exchanging data and synchronization signals.

- Synchronization among cooperating concurrent processes is essential for preserving precedence relationships and for preventing concurrently related timing problems.

Interprocess Interaction

There are three primary forms of explicit interprocess interaction:

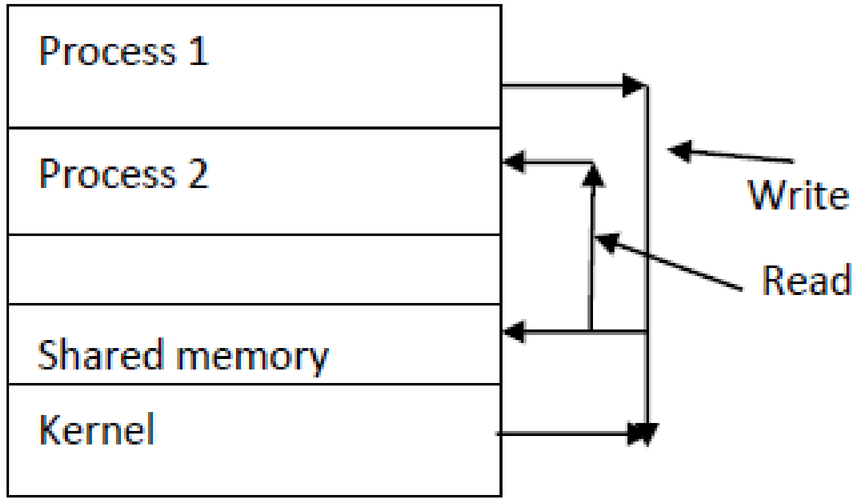
- Interprocess Synchronization: A set of protocols and mechanisms used to preserve system integrity and consistency when concurrent processes share resources that are serially reusable.
- Interprocess Signaling: Exchange of timing signals among concurrent processes or threads used to coordinate their collective progress.
- Interprocess Communication: Concurrent cooperating processes must communicate for some purposes like exchanging of data, reporting progress and accumulating collective results.

Concurrent processes generally interact through either of the following models:

- Shared memory
- Message Passing

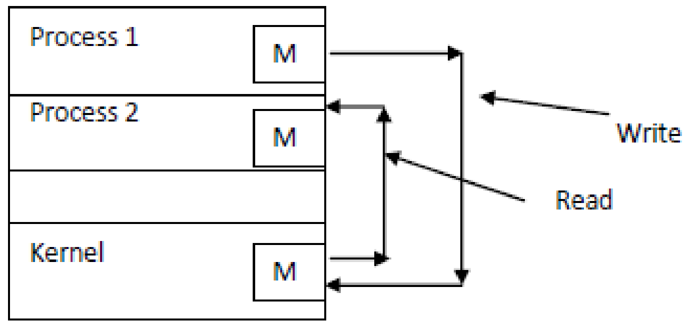
Shared Memory model

In the shared memory model, a region of memory that is shared by co-operating processes is established.
Processes can then exchange information by reading and writing a common variable or common data to shared region



Message Passing

In the message passing model, communication takes place by means of messages exchanged between the co-operating processes using sending and receiving primitives



Terms Related to Concurrency

- **Atomic operation** A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
- **Race condition** A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
- **Critical section** A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
- **Mutual exclusion** The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
- **Starvation** A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.
- **Deadlock** A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
- **Livelock** A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

Atomic action

- In most machines, a single machine instruction is an atomic action, i.e. once the instruction has begun execution, the entire instruction executes before any interrupts are handled.
- If a time shared single processor executes processes, a timer interrupt can occur and the running process can change between any two instructions but not within a single instruction.
- So the sequence of instructions executed are to be atomic.

Race Conditions

- Normally when two processes are running at the same time the results come out the same no matter which one finishes first.
- But it is not so when the results depend on the order of execution in Uniprocessor systems.
- In Multi core - Process 1 and process 2 are executing at the same time on different cores

Consider two processes, Producer and Consumer with following code,

```
process producer {
  while (true) {
    while (count == BUFFER_SIZE); // busy wait
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
  }
}
```

```
process consumer {
   while (true) {
      while (count == 0);  // busy wait
      --count;
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
   }
}
```

Race Condition

Assume *count* = 5 and both producer and consumer execute the statements *++count* and *--count*.

The results of *count* could be set to 4, 5, or 6 (but only 5 is correct).

- ++ count could be implemented as
  ```
  reg1 = count
  reg1 = reg1 + 1
  count = reg1
  ```
- -- count could be implemented as
  ```
  reg2 = count
  reg2 = reg2 - 1
  count = reg2
  ```

- Consider this execution interleaving with “count = 5” initially:

S0: producer executes	reg1 = count	{reg1 = 5}
S1: producer executes	reg1 = reg1 + 1	{reg1 = 6}
S2: consumer executes	reg2 = count	{reg2 = 5}
S3: consumer executes	reg2 = reg2 – 1	{reg2 = 4}
S4: producer executes	count = reg1	{count = 6}
S5: consumer executes	count = reg2	{count = 4}

- Variable count represents a shared resource

Critical Section Problem

- Concurrent access to shared data results in data inconsistency.
- Examples are variables not recording all changes; a process may read inconsistent values ; final value of the variable may be inconsistent.
- Maintaining data consistency in multi-process environment requires mechanisms to ensure orderly execution of cooperating processes i.e. processes are to be synchronized such a way that one process can access the variable at any one time.
- This is referred as the mutual exclusion problem.

Critical section: A critical section is a code segment, common to n cooperating processes, in which the processes may be accessing common/shared variables.

A critical section environment consists of

- Entry Section : Core requesting entry into critical section
- Critical Section : Code in which only one process can execute at any one time
- Exit Section : The end of critical section, releasing or allowing others in
- Remainder section : Rest of the code after the critical section.

The solution to the mutual exclusion must satisfy the following requirements:

- Mutual Exclusion condition: only one process can execute its critical section at any one time.
- Progress: When no process is executing in its critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- Bounded Waiting: When two or more processes compete to enter the critical section, a fair chance should be given all processes and there should not be any indefinite postponement.

Two general approach used to handle critical sections in OS

- *Preemptive kernels*, which allows a process to be pre-empted while it is running in a kernel mode,

- *Non-preemptive kernels*, which does not allow a process running in a kernel mode to pre-empt.

Simplest Solution

Disabling interrupts

- Each process disable all interrupts after entering CS and re- enable before leaving

Problem

- if user process does not turn on –> end of the system
- Kernel has to frequently disable interrupts for few instructions while it is updating variables or lists.

Software Solutions

- Only 2 processes, P_i and P_j
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

- Given below is a simple piece of code containing the components of a critical section used for solving CS problem.

```
do {
     while ( turn != i );                  /*   Entry Section*/
    /* critical section */
     turn = j;                             /*  Exit Section */
    /* remainder section */
} while(TRUE);
```

Algorithm 1

- Shared variables:
  - int turn;

  initially turn = 0
  - turn - i $\Rightarrow P_i$ can enter its critical section
- Process P_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress
  - Suppose that P_i finishes its critical section quickly and sets *turn* = j; both processes are in their non-critical parts. P_i is quick also in its non-critical part and wants to enter the critical section. As *turn* == j, it will have to wait even though the critical section is free.
  - Moreover, the behaviour inadmissibly depends on the relative speed of the processes

Algorithm 2

- Shared variables
  - boolean flag[2];
    initially flag [0] = flag [1] = false.
  - flag [i] = true $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

```
do {
    flag[i] := true;
    while (flag[j]) ;      critical section
    flag [i] = false;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3 (Peterson's Solution)

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,4,5...

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_\text{i}$ for i - 0, …, $n – 1$
- Shared data

```
boolean choosing[n];

int number[n];
```

Data structures are initialized to false and 0 respectively

Bakery Algorithm

```
do {
  choosing[i] = true;
  number[i] = max(number[0], number[1], …, number [n – 1])+1;
  choosing[i] = false;
  for (j = 0; j < n; j++) {
           while (choosing[j]) ;
           while ((number[j] != 0) && (number[ j] < number[i])) ;
  }
     critical section
  number[i] = 0;
     remainder section
} while (1);
```