Process Synchronisation

There are hundreds and thousands of the processes (threads) running concurrently/parallely.

- They do share data; and without synchronisation (a proper access control of the shared data) a race condition may occur and the data corrupts.
- Process synchronisation tools, both hardware ansd software, are employed to stop this race condition to prevent degradation of system performance and avoid catashtrophe

A race condition where the shared variables may be read and written by multiple processes leading to inconsistency has to be stopped using *mutual exclusion*.

The Critical Section Problem

In a system with 'n' processes (or threads) we have a section of the code in each that updates data in a region shared with one or more processes (threads). Such a region is known as *critical section*.
Critical-section-problem refers to a protocol using which the processes could be synchronised preventing corrupt data.
To propose an effective protocol we divide the code section into a number of sections; they are

- Entry section
- Critical section
- Exit section
- Remainder section

```
while (TRUE){
    <entry section>
    <critical section>
    <exit section>
    <remainder section>
}
```

Criteria to solve Critical Section Problem

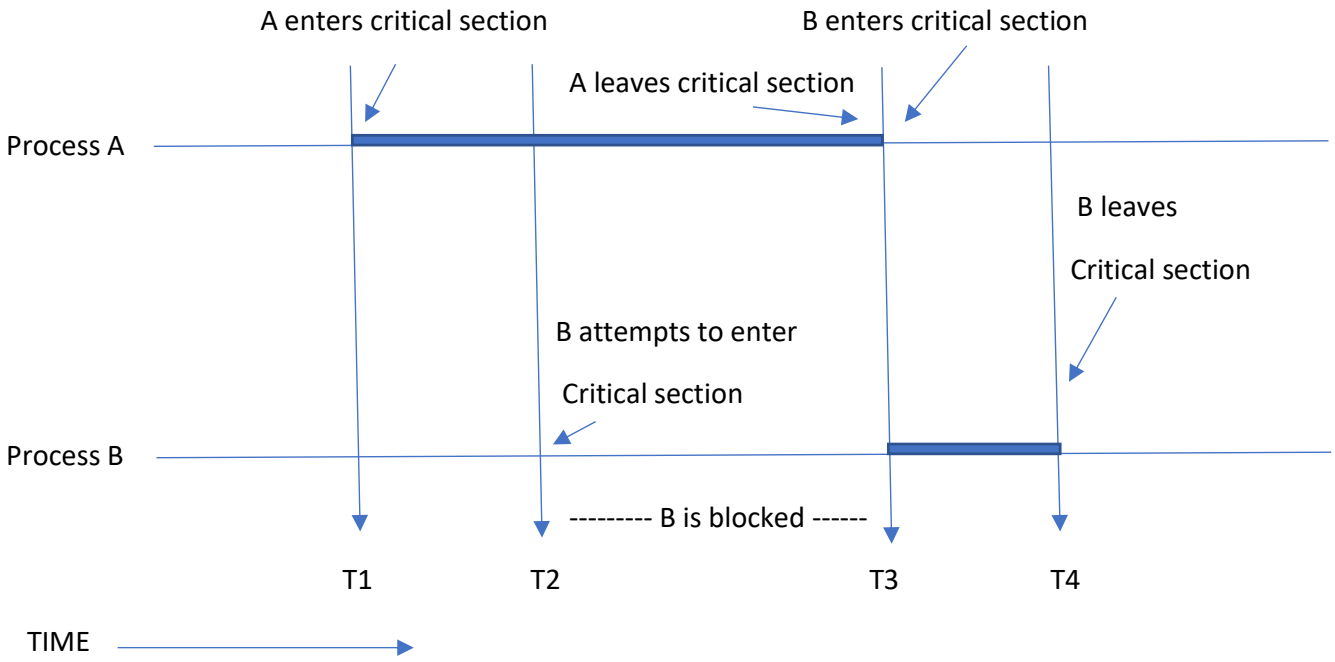
Any technique for a solution to the critical section problem must satisfy

- **Mutual exclusion:** No two processes concurrently be in their critical sections
- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
- **Bounded waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

It is assumed that no process is running at a speed zero – however the order of execution is uncertain as usual.

Critical Section: Timing Diagram for two processes

- Nobody in critical section (CS); process A enters CS at T1
- Process B tried at T2 but was blocked until A leaves CS at T3
- Process B enters CS at T3 and leaves at T4; so both A and B are out of their CS (initial condiiton)



Critical Section: Mutual Exclusion

To introduce the mutual exclusion there are several methods that can be examined

- Busy waiting
  - Disabling interrupts (For uniprocessor system it may be used by the kernel; not suitable for multiprocessor)
  - Lock variables
  - Strict Alternation
  - Peterson's solution
  - Test and Lock (TSL) instruction
- Sleep and wake-up
- Semaphores
- Mutexes
- Monitors
- Message passing
- Avoidiing locks : Read, copy and update

Mutual Exclusion contd.

Disabling interrupts: In a uniprocessor a process entering the CS may disbale all interrupts and enabling them just before leaving the CS. In multiprocessor disabling may block interrupts of one processor – the other may access the shared memory

Lock variables: Assume a shared variable *lock* is used and set to 0. Just before entering the CS a process checks it; if found 1 it waits; otherwise it sets it to 1 and enters the CS. Leads to inconsistency; e.g., just before setting it to 1 the process may be preempted and another sets it to 1. On getting back the access the 1st process sets the lock to 1. So, both the process would be in their CS.

Strict Alternation: In strict alteration 1st process would be in the CS followed by the 2nd and so on. However, for a large variation of the execution time of CS parts a process in its non-critical section may block entry of another to CS.

```
// PROCESS 0
While (TRUE){
        while (turn !=0) ; //
        critical_region();
        trun = 1;
        non_critical_region();
}
```

```
// PROCESS 1
 while(TRUE){
     while (turn != 1); // wait
         critical_region();
         turn = 0;
         non_critical_region();
  }
```

Mutual Exclusion contd.

In strict alteration 1st process would be in the CS followed by the 2nd and so on. However, for a large variation of the execution time of CS parts a process in its non-critical section may block entry of another to CS.

Say, process 0 is in its CS and while leaving sets it to 1. Process 1 enters its CS and quickly completes it and made turn = 0; Now both the process are in their non-CS portion and turn =0. Say, process 0 moves to its CS and immediately completes it and sets turn = 1; now both the process are in their non-CS section. Now, process 0 tries to enter to its CS – this will not be permited (as turn = 1) and process 1 is in non-CS (violating condition 3) – if the non CS part of process 1 is too long it would block the entry of process 0 into its CS.

In general it is not a good idea to use strict alteration when one process is much faster than the other.

Checking a variable continuously and waiting (*busy waiting*) wastes CPU time and should be avoided.

Peterson's Solution : Critical Section Problem

Combining the idea of taking turn and the lock variable Dekker proposed an algorithm that removes the constraint of strict alteration. Peterson, later on, simplifies the algorithm.
Before using the shared variables each process calls enter_cs() with its own process no; 0 or 1 in our case. This call may lead to wait before it is safe to enter the CS. After the CS part is done the process calls leave_cs to allow other process to enter to it's CS.

```
#define     N       2 // No. of processes
int turn;  // whose turn?
boolean into_cs[N]; // initialy 0
void enter_cs (int process){ int  other = 1 - process;
  into_cs[process] = true;  turn = process;
  while (into_cs[other] && turn == process); // null satement
      /* CS portion */
}
void leave_cs(int process){ into_cs[process] = false; }
```

Peterson's Solution contd.

- Initially nobody is in CS. Now, process 0 wanted and calls int_cs(). Process 0 sets turn to 0 and also sets the array flag.
- As process 1 is not interested Process 0 crosses past the null statement immediately and enters CS. During this time if Process 1 calls into_cs() it will wait there until process 0 leaves and make into_cs[0] = false. The same would happen if we start with process 1 entering it's CS first.
- Considering the processes trying to enter their CSs simultaneously – both will set the the flag TRUE and set the variable turn. But, who does it last did not get the chance to run into it's CS. Assume, Process 1 sets last by overwriting turn = 1. Now, both have reached at the while statement — where Process 0 is allowed immediate entry and Process 1 has to wait – until Porcess 0 leaves and set the flag; into_cs[1] = FALSE.

So, i) Mutual exclusion; ii) No process in their non-critical region blocking another to get CS; and iii) Finite wait to get into CS (only N-1 waits at most) are all complied.

Hardware Solution

Peterson's solition may not work in modern multiprocessor where intruction re-ordering takes place for better performance. Many processor support Test and set-lock (TSL) instruction for solution to Critical Section Problem in multiprocessor system. The general form of this instruction is

TSL Rx, LOCK // an indivisible operation which takes the contents of the location LOCK (a shared variable) to register Rx and set a +ve value (say, 1) in LOCK. Before entering the CS a process checks the lock and waits; if it busy (locked) else it enters and sets it free (unlock) while leaving.

```
enter_cs:
TSL Rx, LOCK       // an equivalent in X86 is   MOV R1, #1
          //  XCHG LOCK, R1
CMP  Rx, #0      // TSL instruction is indivisible
JNE enter_cs  // CPU executing TSL locks the memory bus to
RET // prevent other CPUs to accessing memory until it is done
leave_cs:
MOV LOCK, #0
RET
```

Hardware Solution ... contd.

Before entering its CS a process calls enter_cs that does busy-waiting until the lock is free. Finally, it acquires the lock and returns and enters its CS. After leaving the CS the process calls leave_region which resets the lock to 0.

- As with all solutions based on critical regions, the processes must call enter region and leave region at the correct times for the method to work.
- If one process cheats, the mutual exclusion will fail.
- Critical regions work only if the processes cooperate.
- Though Peterson's and hardware solutions are correct but they do busy waiting – that is the process sits in a tight loop waiting to enter its CS leading to an inefficient system.

Sleep and Wake-up

Previous solutions encroaches CPU time for busy-waiting degrading system performance. Hence, the new approaches relying on sleep and wake-up to avoid unnecessary CPU blockage and other unexpected situation, like *priority ionversion*.

Priority Inversion: Say a high priority process H has to run whenever it is ready preempting a low priority process L. Suppose, L is in CS and H becomes ready to run – so it will be allowed to run and would be doing busy-waiting (for the L to unlock) – for ever as L would not be allowed to leave its CS section (as H is running and scheduler would not schedule L again).

The *sleep* and *wakeup* calls may be utilized to implement synchronization. The sleep() call suspends the calling process until it is awkened by a wakeup() call. So, instead of busy waiting to enter to CS a process may go to sleep with the expectation that when the CS is safe to enter – somebody would wake it up.

Producer – Consumer : Sleep and Wake-up

```
#define N 100 /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
void producer(void)
{  int item;
   while (TRUE) { item = produce_item( ); /* generate next item */
if (count == N) sleep(); /* buffer is full, go to sleep */
insert_item(item); /* put item in buffer */
    count = count + 1;
    if (count == 1) wakeup(consumer); /* was buffer empty? */}
}
void consumer(void)
{ int item;
  while (TRUE) { if (count == 0) sleep(); /* if buffer is empty,
                              got to sleep */
item = remove_item(); /* take item out of buffer */
    count = count - 1;
    if (count == N-1) wakeup(producer);
consume_item(item);}
}
```

Producer – Consumer : Sleep and Wake-up contd.

There would be race condition as access to the shared variable *count* is not restricted.

- The consumer has read the value of count is 0 – buffer empty; at that point it is preempted
- The producer produces an item and inserts the same in the buffer and made count 1 – so
- the producer tries to wake-up the consumer; unforutunately the consumer is not logically asleep and the wake-up signal is lost.
- When the consumer runs again it tests the value of count previously read (it was 0) and goes to sleep.
- The producer would fill the buffer and goes to sleep; after that both the processes are asleep for ever.

The problem is that wakeup is sent to a process who is not really sleeping and the signal is lost. This can be solved by adding a wake-up waiting bit. However, for more processes you need more such bits.

Semaphore

The solution came from Dijkstra in 1965 in the form of integer variable (semaphore) to count the number of wake-ups saved for future use. The proposal:

- Allow two operations; *down* (alternately; *wait* or *P*) and *up* (alternately, *signal* or *V*) [similar to sleep and wakeup]
- The down operation on semaphore checks to see if it is greater that 0; if yes
- it decreases the value by 1 (i.e; uses one stored wake-ups) and continues
- If it is 0 the process is put to sleep (without doing the down)
- cheking the value, changing it, and possibly going to sleep are a single atomic operation; once started on a semaphore then it cannot be stopped
- The up operation increases the value of the semaphore.
- If one or more operations were sleeping on that semaphonre, unable to complete an earlier down; one of them is chosen randomly and allowed to compete its down. The up and wake-up are also atomic.
- So, after an up the semaphore is still 0 however one less process is sleeping on it.

Solving Producer-consumer problem using Semaphores

```
#define  N  100
typedef semaphore int;
semaphore mutex= 1; // to folow mutual exclusion
semaphore empty = N; // count empty buffer slots
semaphore full = 0; // count full buffer slots

void producer(void)
{
int item;
while (TRUE) { /* TRUE is the constant 1 */
item = produce_item();  /* item to put in buffer */
down(&empty);  /* decrement empty count */
down(&mutex);  /* enter critical region */
insert _item(item);  /* put new item in buffer */
up(&mutex);  /* leave critical region */
up(&full);  /* increment count of full slots */
}
}
```

Solving Producer-consumer problem using Semaphores

```
void consumer(void)
{
int item;
while (TRUE) { /* infinite loop */
down(&full); /* decrement full count */
down(&mutex); /* enter critical region */
item = remove_item( ); /* take item from buffer */
up(&mutex); /* leave critical region */
up(&empty); /* increment count of empty slots */
consume_item(item); /* do something with the item */
}
}
```

The solution uses 3 semaphores. The mutex (which was set to 1 in the beginning) and uses up and down operation here to control the entry to CS and ensures mutual exclusion is sometimes known as *binary semaphore*. The full and empty, on the other hand, are counting semaphores.

Mutexes

When the semaphore ability to count is not required (it is flipping between 1 and 0) a simplified version of semaphore, called a *mutex*, is used.

- Mutexes are good for implementing mutual exclusion to some shared resource or code.
- They are easy and efficient to implement
- Very useful in thread packages that are entirely implemented in users space

So, mutex is a shared variable and may be implemented using a bit (though in practice it would be an integer) and may be in locked or unlocked state. We have two procedures with mutex; *mutex_lock* and *mutex_unlock*. When a thread (or process) wishes to enter CS it calls mutex_lock. If it is not locked the thread enters CS. If it is locked the calling thread is blocked until the thread using its CS unlocks it by calling multex_unlock.

Mutex contd.

Code for user level thread package – Note that here instead of busy-wait the thread releases CPU (thread_yield) and allows another thread to run. Since thread_yield is just a call to the thread scheduler in user space, it is very fast. As a consequence, neither mutex lock nor mutex unlock requires any kernel calls. Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions.

```
mutex lock:
  TSL Rx,MUTEX ; copy mutex to register and set mutex to 1
  CMP Rx,#0 ; was mutex zero?
  JZ    ok ; if it was zero, mutex was unlocked, so return
  CALL thread_yield ; mutex is busy;
; release the CPU and schedule another thread
  JMP mutex_lock ; try again
ok:
  RET ; return to caller; critical region entered

mutex unlock:
  MOVE MUTEX,#0 ; store a 0 in mutex
  RET ; return to caller
```

Monitor

Even with semaphores and Mutexes we could not rule out synchronisation problems. Just change the order of the two downs in the producer's code of the previous example; i.e.;

- so, mutex is decrement done earlier than empty
- if the buffer were completely filled; the producer would block with mutex set to 0
- the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too
- Both processes would stay blocked forever – leading to a *deadlock*

So, care must be exercised to solve synhronisation problem with semaphores. To write correct programs Hansen and Hoare (1974) proposed a higher level form of synchronisation primitive; called a *Monitor*.

Monitor

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

```
monitor monitor_ name
{
/* shared variable declarations */
function P1 (...) {...}
function P2 (...) {...}
.
function Pn (...) {...}

initialization code ( ...) {...}
}
```

The monitor construct ensures that only one process at a time is active within the monitor. However, this is not enough to solve the synchronisation problem.

Monitor contd.

For taylor-made synchronisation we need some variable, say x, y, of type *condiiton x, y;*

- The only operations that can be done on a condition variable are wait() and signal()
- The operation x.wait() means the process invoking this operation is suspended until another process invokes x.signal().
- x.signal() resumes exactly one suspended process. If there is no suspended process then signal won't effect the state of x [this is unlike the semphore operation down() which changes the state of the semaphore.]

Let the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor

Monitor contd.

When P and Q both are within the Monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

- Signal and wait. P either waits until Q leaves the monitor or waits for another condition.
- Signal and continue. Q either waits until P leaves the monitor or waits for another condition

There are reasonable arguments in favor of adopting either option. On the one hand, since P was already executing in the monitor, the signal-andcontinue method seems more reasonable. On the other, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold. A compromise between these two choices exists as well: when thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed. P