1. Various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as **loosely coupled systems** or distributed systems. These systems processors differ in sizes and functions. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network

Since resources are being shared, computation is highly fast and durable

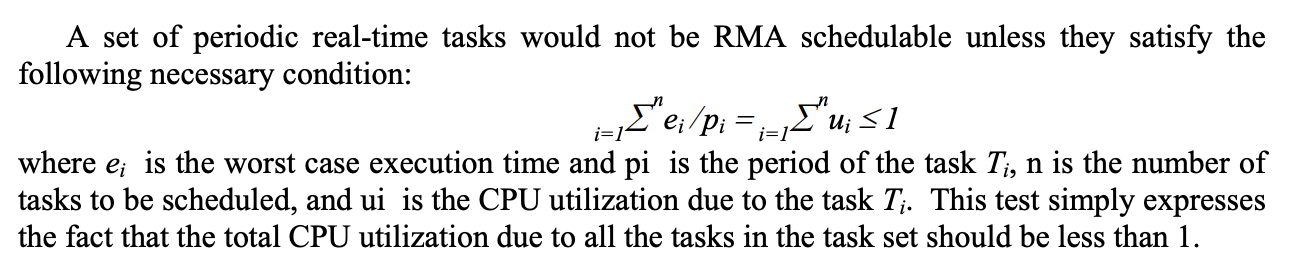
Real time: real-time transactions

2. (a) A **zombie process** is a [process](https://en.wikipedia.org/wiki/Process_(computing)) that has completed execution (via the [exit](https://en.wikipedia.org/wiki/Exit_(system_call)) [system call](https://en.wikipedia.org/wiki/System_call)) but still has an entry in the [process table](https://en.wikipedia.org/wiki/Process_table). This occurs for [child processes](https://en.wikipedia.org/wiki/Child_process), where the entry is still needed to allow the [parent process](https://en.wikipedia.org/wiki/Parent_process) to read its child's [exit status](https://en.wikipedia.org/wiki/Exit_status): once the exit status is read via the [wait](https://en.wikipedia.org/wiki/Wait_(system_call)) [system call](https://en.wikipedia.org/wiki/System_call), the zombie's entry is removed from the process table and it is said to be "reaped". A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then reaped by the system – processes that stay zombies for a long time are generally an error and cause a [resource leak](https://en.wikipedia.org/wiki/Resource_leak).

(b) When the parent process calls wait(), after the creation of child, it indicates that, it will wait for the child to complete and it will reap the exit status of the child. The parent process is suspended(waits in a waiting queue) until the child is terminated.

3. (a) Rate-monotonic analysis is optimal. Priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority. This fixed-priority scheduling policy is the optimum assignment of static priorities to processes, in that it provides the highest CPU utilization while ensuring that all processes meet their deadlines.

If any other fixed-priority scheduling policy can meet deadlines, so can RM.



(b) RMA possesses good transient overload handling capability. Good transient overload handling capability essentially means that, when a lower priority task does not complete within its planned completion time, it can not make any higher priority task to miss its deadline. Let us now examine how transient overload would affect a set of tasks scheduled under RMA. Will a delay in completion by a lower priority task affect a higher priority task? The answer is: ‘No’. A lower priority task even when it exceeds its planned execution time cannot make a higher priority task wait according to the basic principles of RMA − whenever a higher priority task is ready, it preempts any executing lower priority task. Thus, RMA is stable under transient overload and a lower priority task overshooting its completion time can not make a higher priority task to miss its deadline.

4. (a)*Priority Inversion* means that priority of tasks get inverted and *Priority Inheritance* means that priority of tasks get inherited. Both of these phenomena happen in priority scheduling. Basically, in *Priority Inversion*, higher priority task (H) ends up waiting for middle priority task (M) when H is sharing critical section with lower priority task (L) and L is already in critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion. One of the solution for this problem is *Priority Inheritance*. In *Priority Inheritance*, when L is in critical section, L inherits priority of H at the time when H starts pending for critical section. By doing so, M doesn’t interrupt L and H doesn’t wait for M to finish.

(b) Nested resource locks can lead to deadlock when you use the priority inheritance protocol since nested resource locks increase the wait time. You can avoid deadlock by allowing each task to own only one shared resource at a time. When this condition is met, the worst-case wait time matches the priority ceiling protocol's worst-case wait. In order to prevent misuse, some operating systems that implement priority inheritance don't allow nested locks.

5. No, Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity. So more kernel threads will have more overheads.

7. **Mutual Exclusion (互斥)：**當有一個 process 佔住 critical-section 時，其他 process 不能進入 critical section，不會有兩個 process 同時間在 critical-section 中工作。  
**Progress：**當沒有 process 要在 critical-section 中執行時，不能阻擋其他想要進入 critical section 工作的 process 進入 critical-section，要選擇其中一個候選 process 進入 critical-section，不能空在那邊。(延遲：postponed)  
**Bounded Waiting：**等待 critical-section 的時間，不能是無窮大的時間，是有個界線。也就是說，不能佔住了critical-section 就不出來了。

The problem consists of designing a protocol by which no philosopher remains hungry indefinitely, the starvation-freeness condition, assuming each eating session lasts only for a finite time1 . In addition, the progress condition means that there should be no deadlock: at any given time, at least one philosopher that is hungry should move to eating after a bounded period of time.

8. offset 用來計算某一個struct結構的成員相對於該結構起始位址的偏移量

((s \*)0): takes the integer zero and casts it as a pointer to s.

((s \*)0)->m: dereferences that pointer to point to structure member m.

&(((s \*)0)->m): computes the address of m.

(size\_t)&(((s \*)0)->m): casts the result to an appropriate data type.

**offsetof**將數值 **0**強制轉型成TYPE指標型別，**0** 會被當作該TYPE的起始地址，然後再指向某成員 (  (TYPE \*) 0 )->MEMBER，就可以得到MEMBER的位址，因為起始位址等於 **0**，所以MEMBER的位址也就等於MEMBER與起始位址  **0**的偏移(offset)。所以若將起始位址 **0**改成其它數值的話，得到的偏移(offset)就不對了。

Or

This can be understood as taking a null pointer of type structure st, and then obtaining the address of member m within said structure.

Since we are considering 0 as address of the structure variable, c will be placed after 16 bytes of its base address i.e. 0x00 + 0x10. Applying & on the structure element (in this case it is c) returns the address of the element which is 0x10. Casting the address to *unsigned int* (size\_t) results in number of bytes the element is placed in the structure.

**container\_of**

它需要引用**offsetof**巨集，它的作用是用來取得struct結構的起始點，只要知道該結構的任一個成員的位址，就可以使用**container\_of**巨集來算出該結構的起始位址。

container\_of可分解為2個動作

1. 定義一個指向**member**型別的指標**\_\_mptr**，然後將參數**ptr**餵給該指標。

2. 將**\_\_mptr**的位址減掉**member**與結構起始點的偏移(利用offsetof)就可得到該結構的起始點位址

offsetof用來得知struct中某一member的位址相對於該struct起始位址的距離，而container\_of是只要得知某一member的位址，就可以利用它進一步推算出strucct的起始位址。(offsetof和container\_of進一步的描述之前已寫在這一篇-->[Linux的container\_of與offsetof巨集](https://myao0730.blogspot.tw/2016/09/linuxcontainerof-offsetof.html) )。所以，list\_for\_each和ist\_entry這兩個巨集搭配使用就可以存取整個鏈結串列。

**prefetch**的含义是告诉cpu那些元素有可能马上就要用到，告诉cpu预取一下，这样可以提高速度，**用于预取以提高遍历速度；**

networking has a lot of regular lists that are often empty or a single entry, and prefetching is not going to do anything but add useless instructions

Peterson algorithm:

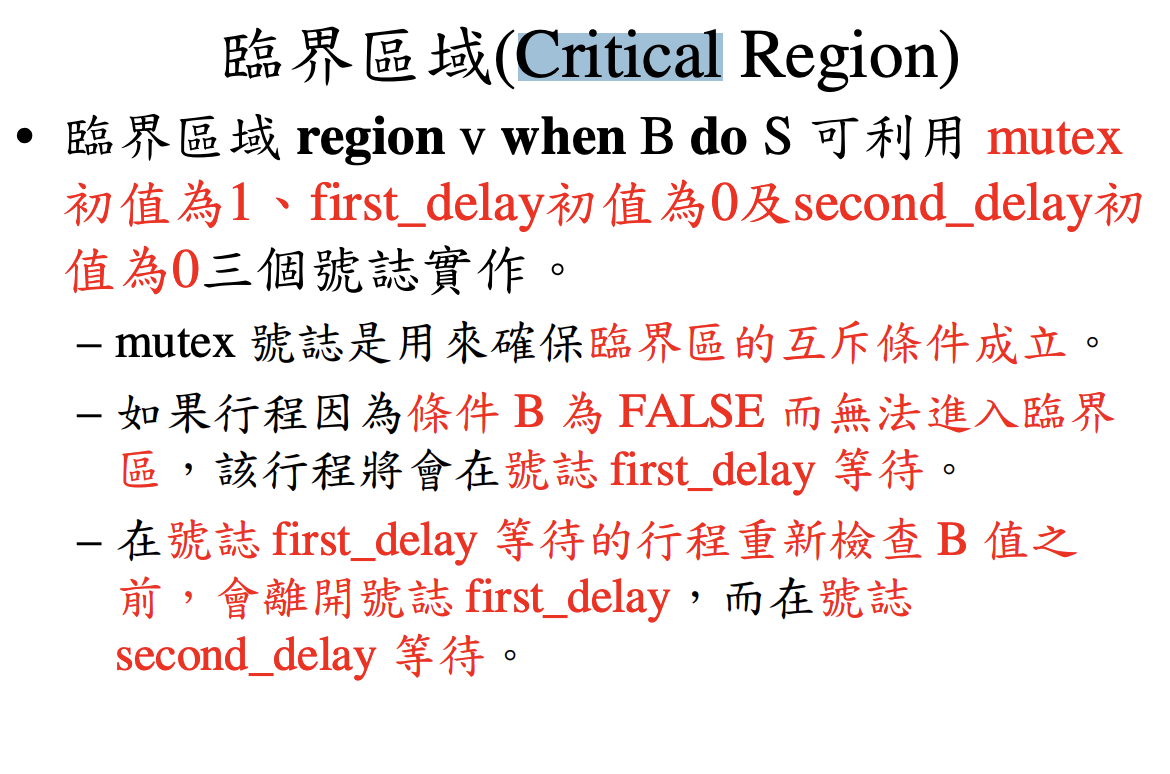
The producer consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producer produce an item and put it into buffer. If buffer is already full then producer will have to wait for an empty block in buffer. Consumer consume an item from buffer. If buffer is already empty then consumer will have to wait for an item in buffer. Implement Peterson’s Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.

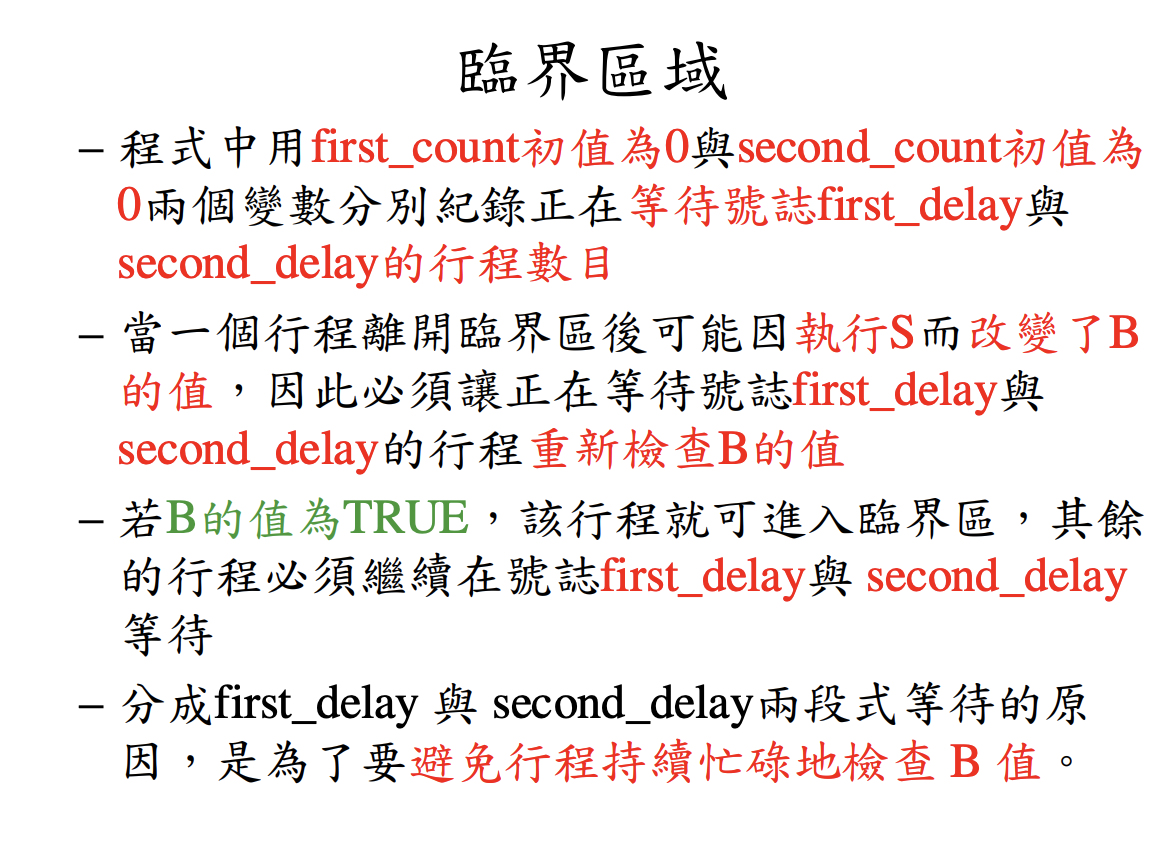
It can prevent deadlock.

Mutual Exclusion: The eventual value of turn determines which process enters the critical region.

Progress: A process can only be stuck in the while loop, and the process which can keep it waiting must be in its critical region.

Bounded waiting: Each process wait at most one entry by the other process.





3. Paravirtualization is a virtulization technology that requires modification of the kernel. It replaces nonvirtualizable instructions with hypercalls that communicate directly with the virtualization layer hypervisor.allowing an OS to actually recognize the presence of a hypervisor and communicate directly with that hypervisor to share activity that would otherwise be complex and time-consuming for the hypervisor's VM manager to handle.

4. Rate Monotonic Scheduling (RMS) is a real time preemptive scheduling algorithm. It is used for processes which meet the following conditions:

1) Processes should be periodic and there should be a time period for every process;

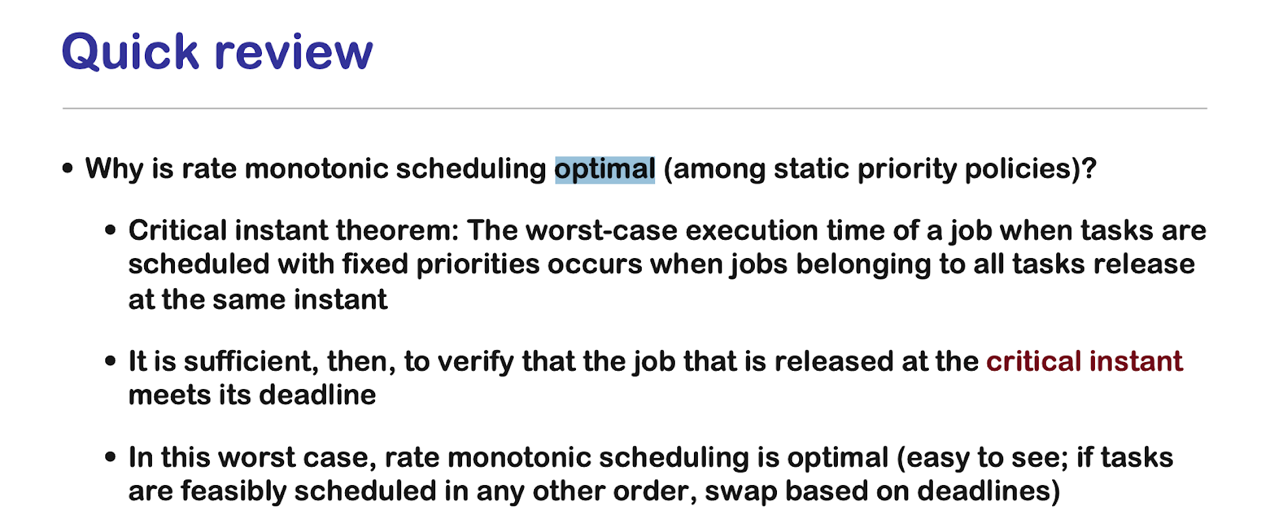
2) Every process must require the same amount of CPU time on each burst;

3) Every process should be independent;

4) If a process is not periodic, then it should not have a deadline.

Earliest Deadline First (EDF) Scheduling is a type of real time scheduling algorithm. In the EDF, the first two conditions of the RMS algorithm are not required. In EDF, processes are sorted by using their deadlines. A process which has the earliest deadline is run first. If a new process is ready, its deadline is checked. If the deadline is before the running process, then the new process pre-empts the running process.

Optimal: If any other fixed-priority scheduling policy can meet deadlines, so can RM.



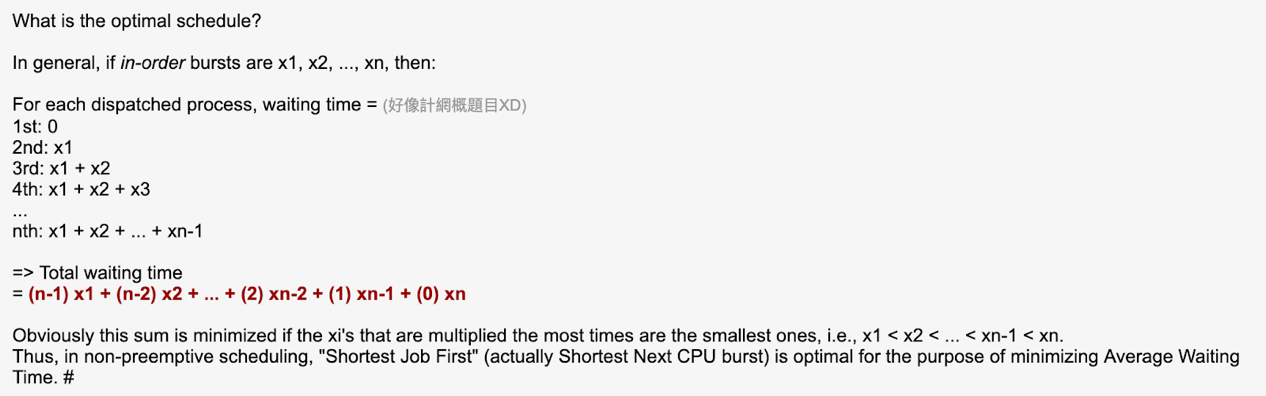
5.

(a)The primary benefit of a thread pool over creating a new thread for each task is that thread creation and destruction overhead is restricted to the initial creation of the pool, which may result in better [performance](https://en.wikipedia.org/wiki/Performance_tuning) and better system [stability](https://en.wikipedia.org/wiki/Stability_Model).

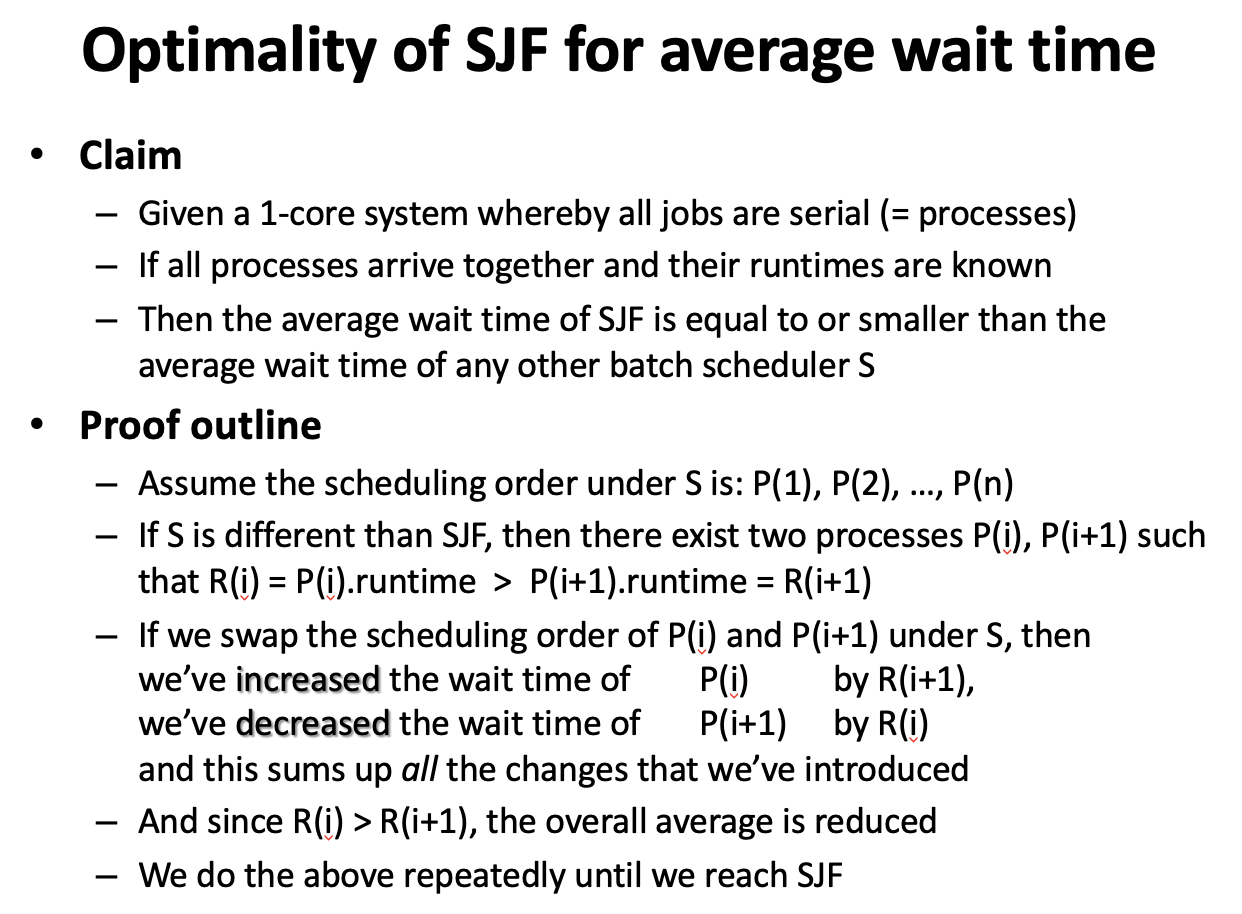
(b) If the number of request varies a lot from time to time, the utilization rate of the thread pool will vary from time to time too. If there are less requests, many threads will be idle, which is a waste of resource.

(c) Implicit threading reduces programmer’ s overhead of creating threads.

6.  Nonpreemptive:



Preemptive:



7.

Using min-heap we can get min virtual time process in O(1) time, h2eaps are array based and hence requires contiguous memory in kernel space. This is because the way heaps are implemented in Linux. See the files lib/prio\_heap.c and include/linux/prio\_heap.h and you'll note that heap is kmalloc'd using heap\_init. Once the multi-programming space becomes huge, maintaining thousands of struct sched\_entity requires lot of contiguous space (it runs in several pages). From time and performance point of view, one would prefer heap as hepify operation can run in background once min vruntime is picked but it's space requirement which makes bottleneck.

8.

int pid = 0

if ((pid = fork()) == 0){

printf(“child!”)

}else if(pid > 0){

printf(“Parent!”)

}else{

printf(“fork error!”)

}

9. No, Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity. So more kernel threads will have more overheads.

10.

Two-process solution

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

The two processes share two variables:

int turn;

Boolean flag[2]

The variable turn indicates whose turn it is to enter the critical section.

The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

1. This solution works for 2 processes, but this solution is best scheme in user mode for critical section.
2. This is also a busy waiting solution so CPU time is wasted. And because of that “SPIN LOCK” problem can come. And this problem can come in any of the busy waiting solution.

<https://stackoverflow.com/questions/46919797/why-is-a-monitor-implemented-in-terms-of-semaphores-this-way>