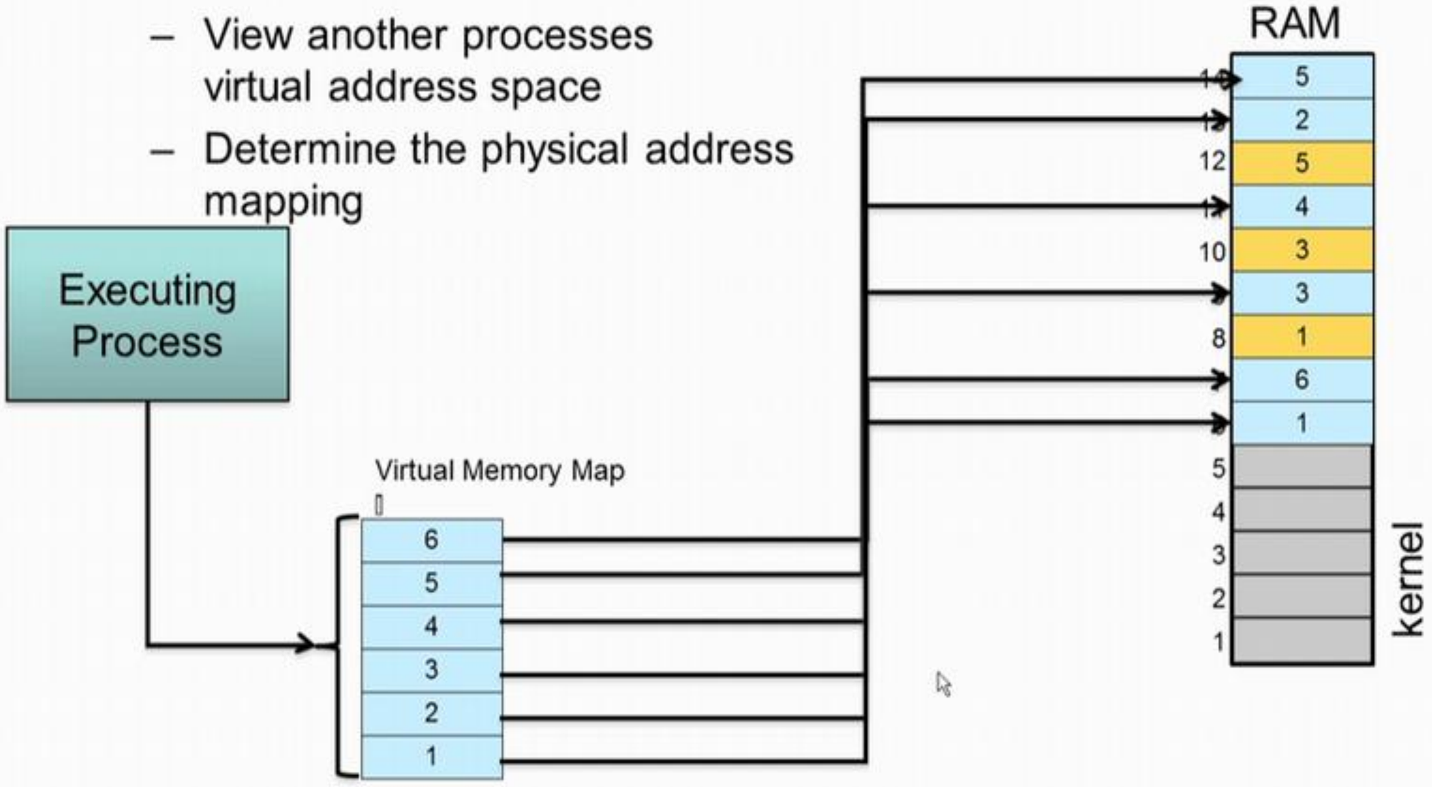
Interprocess Communication

Virtual Memory View

- During execution, each process can only view its virtual addresses,
- It cannot
  - View another processes virtual address space
  - Determine the physical address mapping



How does one process communicate with another process?

- The mechanism is known as Inter Process Communication.
- Essentially with IPC or inter process communication, two processes will be able to send and receive data between themselves.
- Each processes job is to only focus on a single aspect
- The advantage of IPC is that the processes could be modular
- So essentially each process is meant to do a single job and processes could then communicate with each other through IPC

Advantages of IPC

Inter Process Communication

Advantages of Inter Process Communication (IPC)
– Information sharing
– Modularity/Convenience

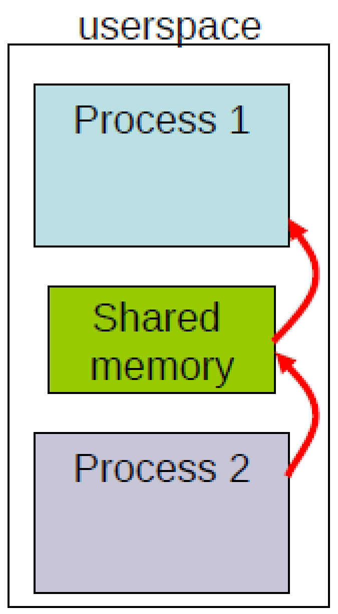
• 3 ways
– Shared memory
– Message Passing
– Signals

Independent and Cooperating process

- An independent process is not affected by the execution of other processes
- A co-operating process can be affected by other executing processes. Cooperating processes need Inter Process Communication (IPC)
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions.

Shared Memory

- One process will create an area in RAM which the other process can access
- Both processes can access shared memory like a regular working memory
  - Reading/writing is like regular reading/writing
  - Fast
- Limitation : Error prone. Needs synchronization between processes

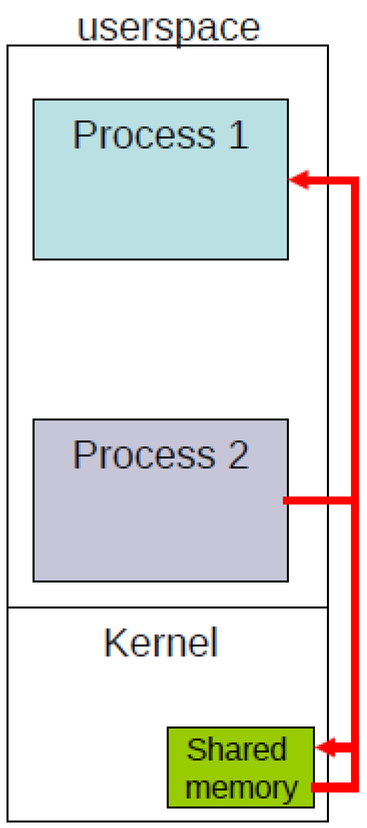


Shared Memory

- The advantage with this is that the communication is extremely fast, there are no system calls which are involved.
- only requirement is that define an array in shared memory and then fill the array in the shared memory which can then be read by the other process

Message Passing

- Shared memory created in the kernel
- System calls such as send and receive used for communication
  - Cooperating : each send must have a receive
- Advantage : Explicit sharing, less error prone
- Limitation : Slow. Each call involves marshalling / demarshalling of information



Message passing

- Shared memory is created as part of the user space
- In message passing the shared memory is created in the kernel
- Essentially we would then require system call such as send and receive in order to communicate between the two processes
- If a process 2 wants to send data to process 1, it will invoke the send system call. So this would then cause the kernel to execute and it would result in the data return into the shared memory.
- When process 1 invokes receive, data from the shared memory would be read by process 1.

Advantage and Disadvantage

- The advantage of this particular message passing is that the sharing is explicit. Essentially both process 1 and process 2 would require support for the kernel to transfer data between each other.
- The limitations is that it is slow, each call for the send or receive involves the marshalling or demarshalling of information. And a system call has significant overheads. Therefore, message passing is quite slow compared to shared memory.
- However, it is less error prone then shared memory because the kernel manages the sharing, and therefore would be able to do the synchronization between the process 1 and process 2.

Marshalling and Demarshalling

- "**marshalling**" refers to the process of converting the data or the objects into a byte-stream, and "**demarshalling**" is the reverse process of converting the byte-stream back to their original data or object

Synchronous and Asynchronous

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Signal

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - eg. 9: kill, 11: segmentation fault
- Send a signal to a process
  - kill(pid, signum)
- Process handler for a signal
  - sighandler_t signal(signum, handler);
  - Default if no handler defined

Signal

- A Third way of inter process communication is by what is known as Signals.
- Signals are unidirectional communication between processes.
- Signals are a limited form of inter-process communication (IPC), typically used in Unix, Unix-like, and other POSIX-compliant operating systems
- The operating system defines some predefined signals and these signals could be sent from the operating system to a process, or between one process to another.
- Signals are essentially small integers, each of these integers has a predefined meaning. For instance, 9 would mean to kill a process, while 11 would mean a segmentation fault and so on.

Synchronization

What can be the output?

shared variable

```
int counter=5;
```

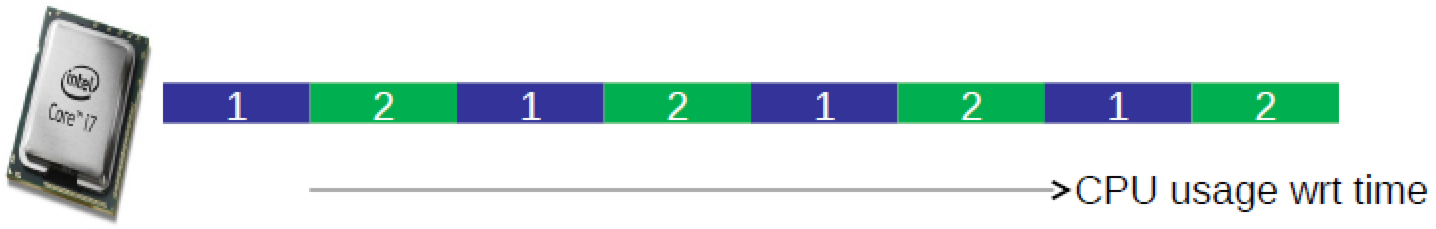
program 0

```
{
   *
   *
counter++
   *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core



Motivation for Synchronization

Shared variable

```
int counter=5;
```

program 0

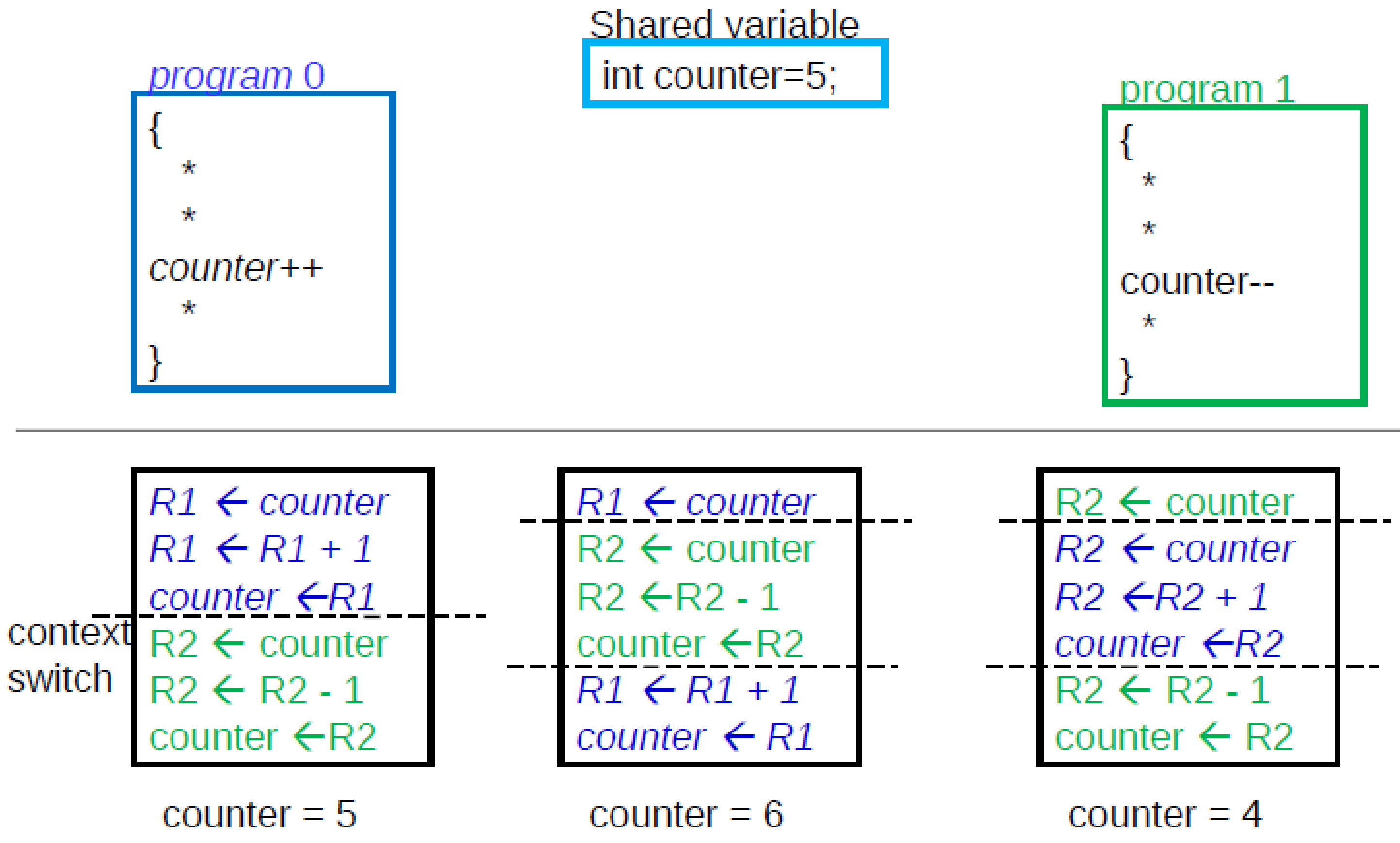
```
{
   *
   *
counter++
   *
}
```

program 1

```
{
   *
   *
counter--
   *
}
```

- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6

Motivation for Synchronization



Shared variable

```
int counter=5;
```

program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

```
R1 ← counter
R1 ← R1 + 1
counter ←R1
---------------- context switch
R2 ← counter
R2 ← R2 - 1
counter ←R2
```

counter = 5

```
R1 ← counter
----------------
R2 ← counter
R2 ←R2 - 1
counter ←R2
----------------
R1 ← R1 + 1
counter ← R1
```

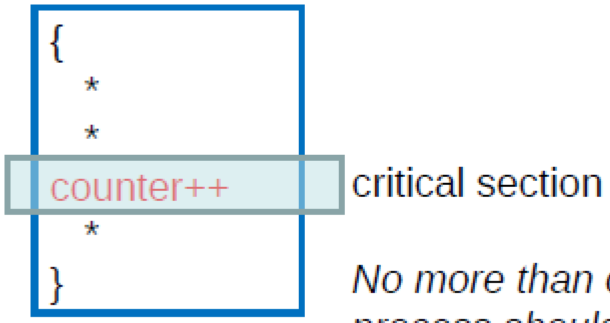
counter = 6

```
R2 ← counter
----------------
R2 ← counter
R2 ←R2 + 1
counter ←R2
----------------
R2 ← R2 - 1
counter ← R2
```

counter = 4

Race Condition

- Race conditions
  - A situation where several processes access and manipulate the same data (*critical section*)
  - The outcome depends on the order in which the access take place
  - Prevent race conditions by synchronization
    - Ensure only one process at a time manipulates the critical data



No more than one process should execute in critical section at a time

Race Condition

- A Race Condition is a situation where several processes access and manipulate the same data. So this part of the process which accesses common or the shared data is known as a Critical Section.
- The outcome of a race condition would depend on the order in which the accesses to that data take place.
- Depending on which program executes first as well as depending on how the context switch is occur the result would vary.
- The race conditions could be prevented by what is known as synchronization.
- Essentially, with synchronization we would ensure that only one process at a time would manipulate the critical data

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last

- To prevent race conditions, concurrent processes must be synchronized

Race Conditions in Multicore

shared variable

```
int counter=5;
```

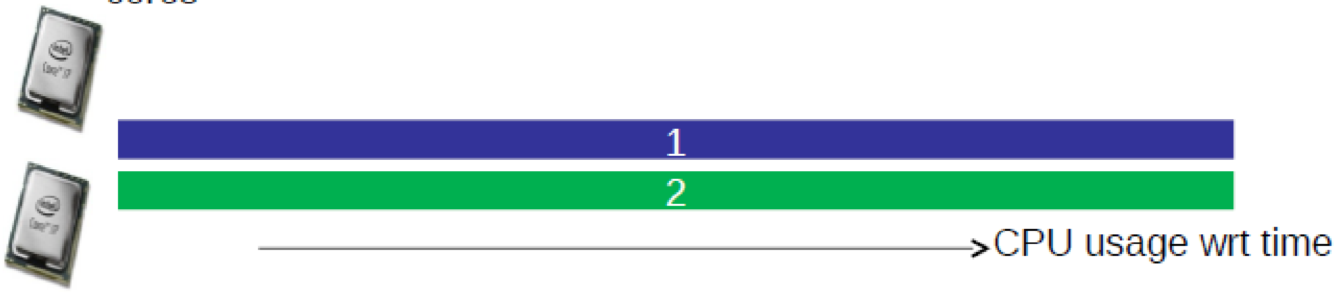
program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores



The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section

Solution to critical section problem

Critical Section

- Requirements
  - **Mutual Exclusion :** No more than one process in critical section at a given time
  - **Progress :** When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay
  - **No starvation (bounded wait):** There is an upper bound on the number of times a process enters the critical section, while another is waiting.

What do you mean by Bounded wait?

- Bounded wait means this particular scenario. Let us say program 0 is present in the critical section while program 1 has requested the lock. So there is an limit on the amount of time that program 1 has to wait before it gets access into the critical section, that is the solution should ensure that program 0 unlocks L and only then will program 1 enter into the lock. So, there is a bound on the amount of time that program 1 waits, if it requests the lock while program 0 is already in the critical section

Locks and Unlocks

shared variable

```
int counter=5;
lock_t L;
```

program 0

```
{
   *
   *
lock(L)
counter++
unlock(L)
   *
}
```

program 1

```
{
   *
   *
lock(L)
counter--
unlock(L)
   *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

Why Locking is required

- The use of lock and unlock constructs in a program will ensure that the critical section is atomic.

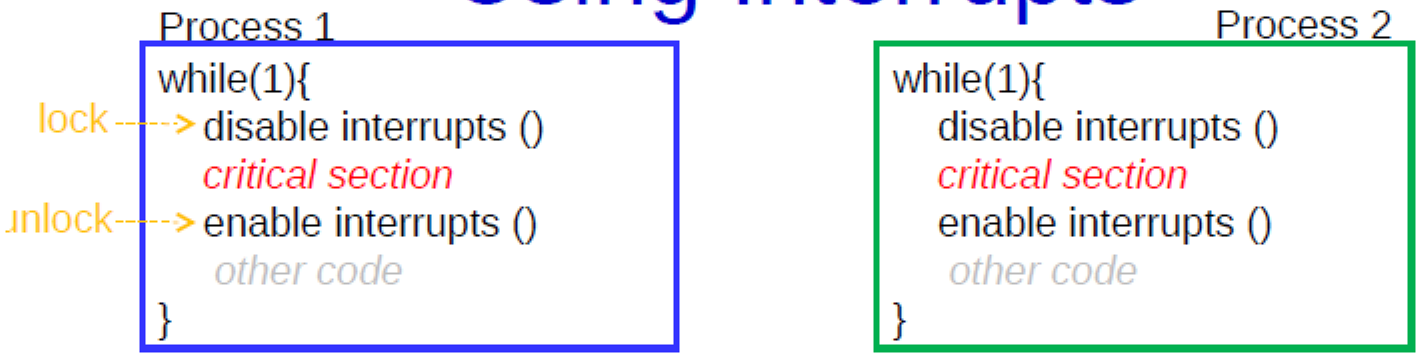
Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- 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

Using Interrupts



Process 1

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
     other code
}
```

Process 2

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
     other code
}
```

- Simple
  - When interrupts are disabled, context switches won't happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Algorithm 1

Software Solution (Attempt 1)

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

Process 2

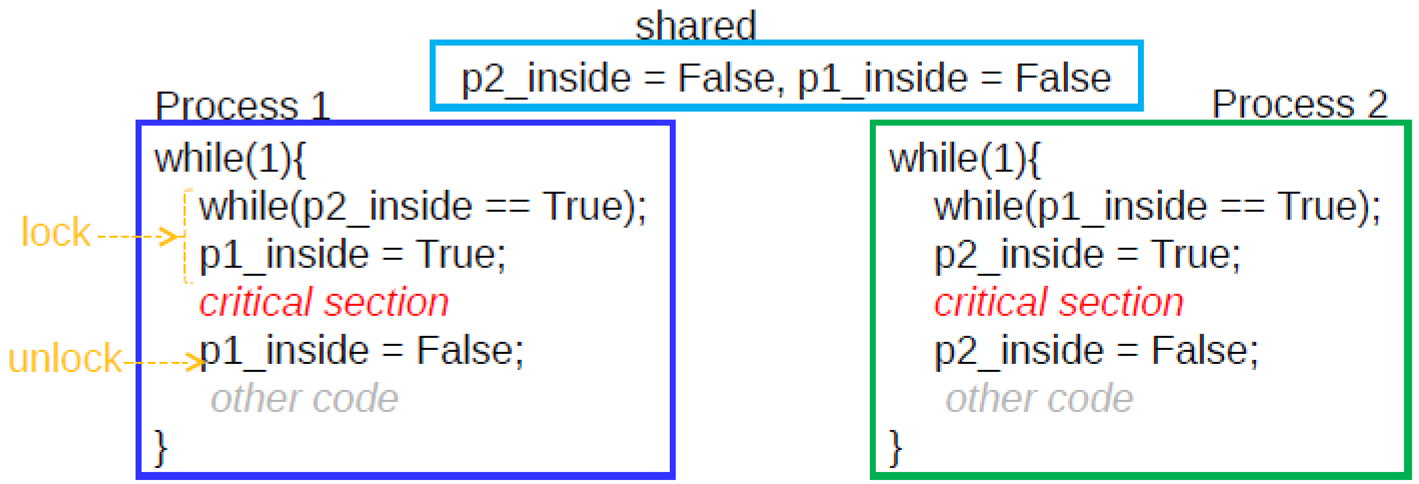
```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

- Achieves mutual exclusion
- Busy waiting – waste of power and time
- Needs to alternate execution in critical section

  process1 →process2 →process1 →process2

Algorithm 2

Software Solution (Attempt 2)



```
shared
p2_inside = False, p1_inside = False
```

Process 1

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
     other code
}
```

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
     other code
}
```

- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

Attempt 2: No mutual exclusion

time

CPU	p1_inside	p2_inside
while(p2_inside == True);	False	False
context switch		
while(p1_inside == True);	False	False
p2_inside = True;	False	True
context switch		
p1_inside = True;	True	True

Both p1 and p2 can enter into the critical section at the same time

Software Solution (Attempt 3)

globally defined

```
p2_wants_to_enter, p1_wants_to_enter
```

Process 1

```
         while(1){
              p1_wants_to_enter = True
lock ----->   while(p2_wants_to_enter = True);
              critical section
unlock---->   p1_wants_to_enter = False
              other code
         }
```

Process 2

```
while(1){
     p2_wants_to_enter = True
     while(p1_wants_to_enter = True);
     critical section
     p2_wants_to_enter = False
     other code
}
```

- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

Peterson’s Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
               favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

lock

unlock

If the second process wants to enter. favor it. (be nice !!!)

favored is used to break the tie when both p1 and p2 want to enter the critical section.

(* the process which sets favored last looses the tie *)

Peterson’s Solution

globally defined

```
p2_wants_to_enter, p1_wants_to_enter, favored
```

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
               favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

Process 2

```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
               favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

Algorithm 3

- 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

High Level Constructs

- Spinlock
- Mutex
- Semaphore

SPINLOCK

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

Spinlocks
(when should it be used?)

- Characteristic : busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.
    - Use mutex instead (…mutex)

Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process