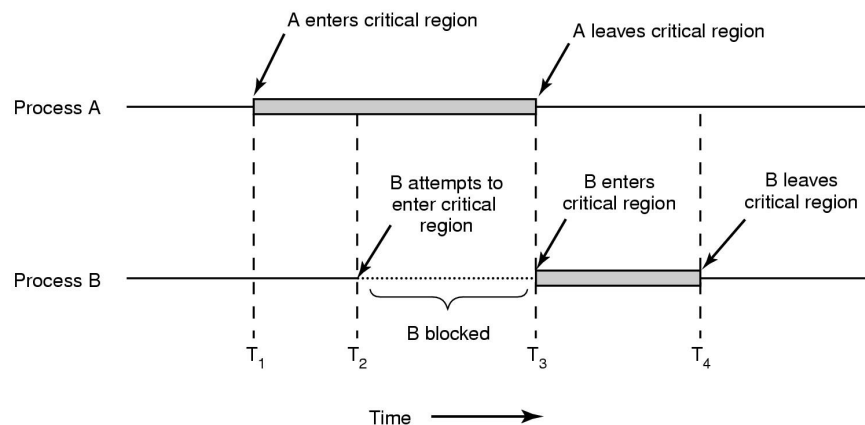


Requirements for Critical Section

- Mutual Exclusion
 - No other process must execute within its own critical section while a process is in it.
- Progress
 - If no process is waiting in its critical section and several processes are trying to get into their critical sections, then entry to the critical section cannot be postponed indefinitely.
 - No process running outside its critical section may block other processes
- Bounded Wait
 - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section.
 - No process should have to wait forever to enter its critical section
- Speed and Number of CPUs
 - No assumption may be made about speeds or number of CPUs

Critical Regions (2)



Mutual exclusively using critical regions

Synchronization With Busy Waiting

- Possible Solutions
 - Disabling Interrupts
 - Lock Variables
 - Strict Alternation
 - Peterson's solution
 - TSL
 - Sleep and Wakeup

Example of busy waiting on a lock (1/2)

- One could think of using a variable as a flag to be checked upon entering a critical section ...
- ... but the lock itself is a critical section!

```
Shared integer lock = 0;
Process i
.
.
while lock == 1;
lock = 1;
execute CS;
lock = 0;
.
```

Process A	Process B
.	.
.	.
while lock == 1;	while lock == 1;
lock = 1;	lock = 1;
.	.
.	.

Possible race condition

Example of busy waiting

- The correct implementation uses a test-and-set instruction to avoid race conditions

Semantic of test-and-set instruction

```
int test-and-set (int a) {  
    int rv = a;  
    a = 1;  
    return rv;  
}
```

Correct lock implementation

```
Process A  
  
Shared integer lock = 0;  
.  
.  
While( test-and-set(lock) ==1)  
    ;  
.  
.
```

Locks: pros and cons

- Pros:
 - simple and fast
 - ubiquitous: every processor has a test-and-set or equivalent operation
- Cons:
 - busy waiting is wasteful of resources (CPU cycles, memory bandwidth)

Semaphores - definition

- Proposed by Dijkstra, it was the first high level construct.
- A semaphore S is an integer variable on which two atomic operations are defined, $P(S)$ and $V(S)$, and with an associated queue.
- P and V semantics:

```
P(S): if  $S \geq 1$  then  $S := S - 1$ 
      else <block and enqueue the process>;
```

```
V(S): if <some process is blocked on the queue> then
      <unblock a process>
      else  $S := S + 1$ ;
```

Semaphores - properties

- The P operation may block a process, but V does not
- Two type of semaphores
 - binary: initial value is 1
 - resource counting: any initial value
- P and V are atomic operations

```
P(S): if  $S \geq 1$  then  $S := S - 1$ 
      else <block and enqueue the process>;
```

```
V(S): if <some process is blocked on the queue> then
      <unblock a process>
      else  $S := S + 1$ ;
```

Example of use

Shared var mutex: semaphore = 1;

Process *i*

begin

.

.

P(mutex);

execute CS;

V(mutex);

.

.

End;