Animations For Part 4

Race Condition

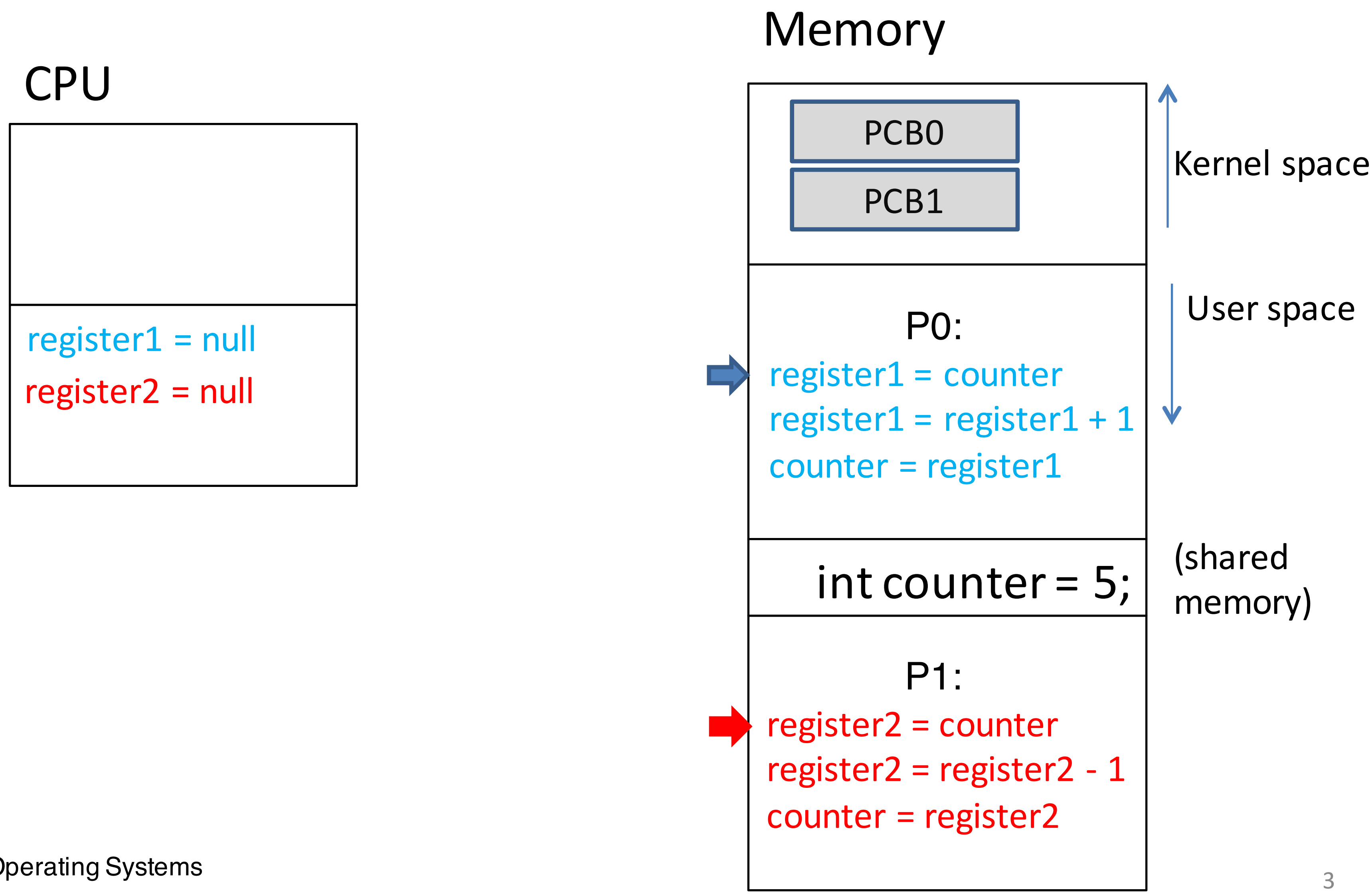
- counter++ could be implemented as
  register1 = counter /*Read counter from the main memory*/
  register1 = register1 + 1 /*Increase the CPU register by one*/
  counter = register1 /*Update counter in the main memory*/
- counter-- could be implemented as
  register2 = counter
  register2 = register2 - 1
  counter = register2
- Two processes: P0: counter++, P1: counter--.
- Consider executing counter++ and counter-- once.
- *counter* does <u>not</u> change if those operations are executed one after one.
  - Consider the following two orders with initial value of 5.

P0→P1:
If we first execute P0, counter=6.
Then we execute P1, counter=**5**.

P1→P0:
If we first execute P1, counter=4.
Then we execute P0, counter=**5**.

 What if context switch occurs in the middle??

Race Condition



Race Condition

CPU

register1 = counter

register1 = 5

register2 = null

Start executing P0

Memory

PCB0

PCB1

Kernel space

User space

P0:

register1 = counter

register1 = register1 + 1

counter = register1

int counter = 5;

(shared memory)

P1:

register2 = counter

register2 = register2 - 1

counter = register2

Race Condition

CPU

register1 = register1 + 1

register1 = 6

register2 = null

Memory

PCB0

PCB1

Kernel space

User space

P0:

register1 = counter

register1 = register1 + 1

counter = register1

int counter = 5;

(shared memory)

P1:

register2 = counter

register2 = register2 - 1

counter = register2

Race Condition

Context switch to P1

CPU

register2 = counter

register1 = 6

register2 = 5

Memory

PCB0

PCB1

Kernel space

User space

P0:

register1 = counter

register1 = register1 + 1

counter = register1

int counter = 5;

(shared memory)

P1:

register2 = counter

register2 = register2 - 1

counter = register2

Race Condition

CPU

register2 = register2 - 1

register1 = 6

register2 = 4

Memory

PCB0

PCB1

Kernel space

User space

P0:

register1 = counter

register1 = register1 + 1

counter = register1

int counter = 5;

(shared memory)

P1:

register2 = counter

register2 = register2 - 1

counter = register2

Race Condition

CPU

counter = register1

register1 = 6

register2 = 4

Context switch to P0

Memory

PCB0

PCB1

Kernel space

User space

P0:

register1 = counter

register1 = register1 +

counter = register1

P0 writes counter

int counter = **6**;

(shared memory)

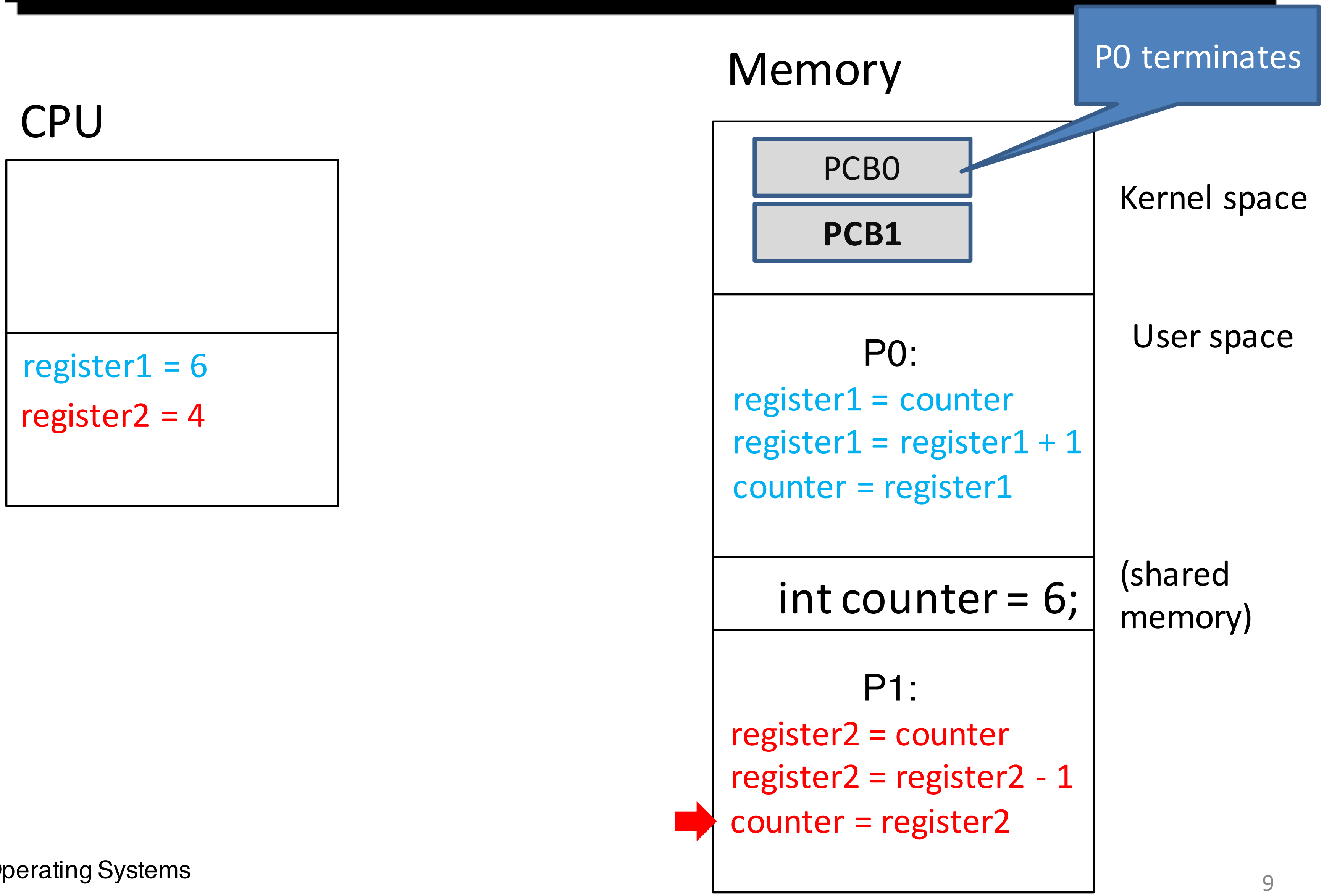
P1:

register2 = counter

register2 = register2 - 1

counter = register2

Race Condition



Race Condition

Context switch to P1

CPU

counter = register2

register1 = 6
register2 = 4

Memory

Kernel space

PCB1

User space

int counter = **6**;

(shared memory)

P1:
register2 = counter
register2 = register2 - 1
counter = register2

Race Condition

CPU

counter = register2

register1 = 6

register2 = 4

Memory

Kernel space

PCB1

User space

P1 writes counter

int counter = **4**;

(shared memory)

P1:

register2 = counter

register2 = register2 - 1

counter = register2

Race Condition

CPU

counter = register2

register1 = 6

register2 = 4

Memory

Kernel space

PCB1

User space

(shared memory)

int counter = **4**;

P1:

register2 = counter

register2 = register2 - 1

counter = register2

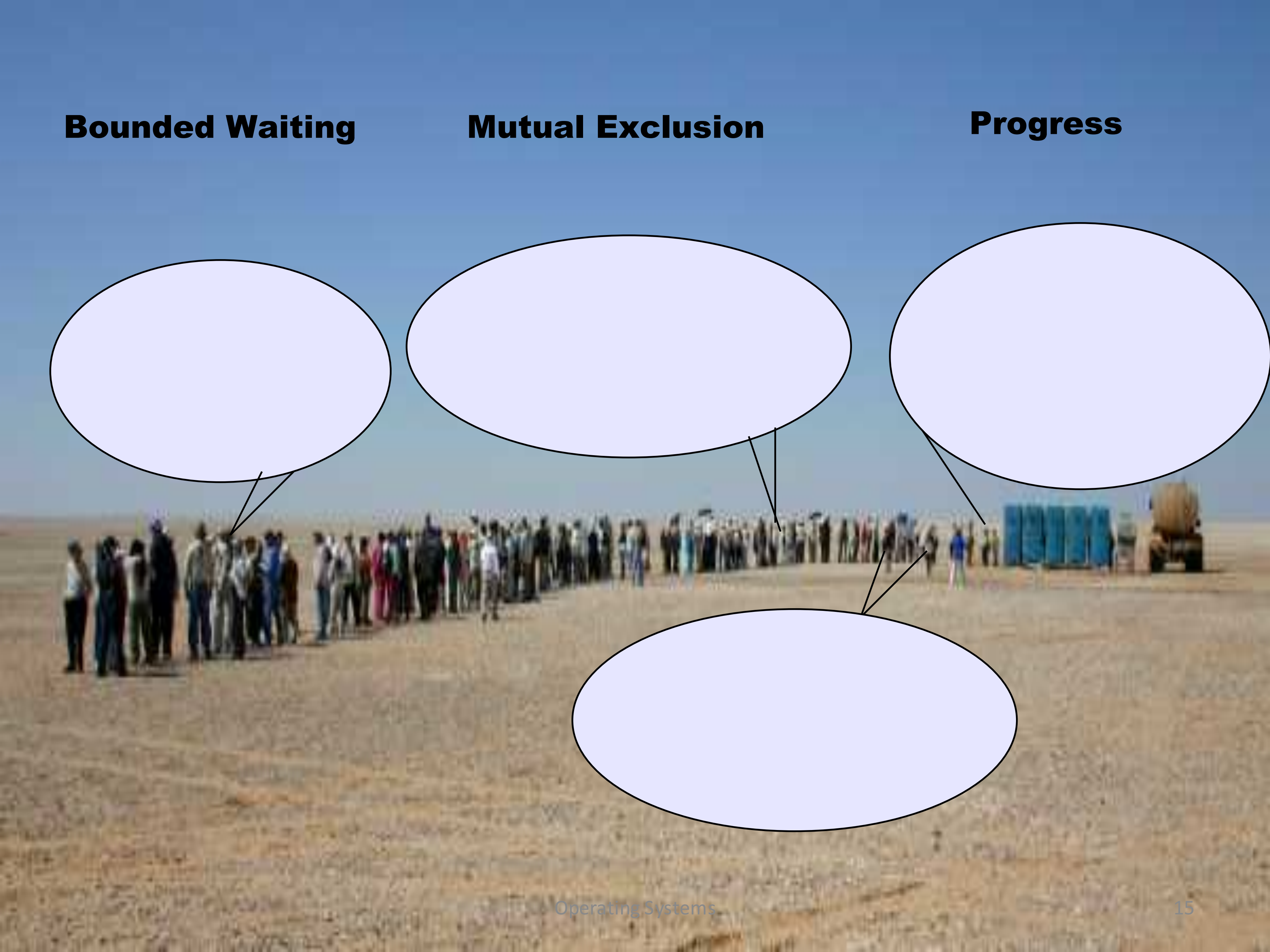
Race condition. Are there other possible values?

Bounded Waiting **Mutual Exclusion** **Progress**

Good concrete example...

*visual learning aid: '**bmp'** in alphabetical order*

Bounded Waiting **Mutual Exclusion** **Progress**


Bounded Waiting
Mutual Exclusion
Progress


Bounded Waiting
Mutual Exclusion
Progress
No cutting in!


Bounded Waiting
Mutual Exclusion
Progress
No cutting in!
Are there door locks?


Bounded Waiting
Mutual Exclusion
Progress
No cutting in!
Are there door locks?
We'll be first if we sprint


Bounded Waiting
Mutual Exclusion
Progress
No cutting in!
Are there door locks?
Well, Did you see anybody go in?
We'll be first if we sprint

- **Progress?**
- **Bounded Waiting?**

What's the difference?

- **Progress?**
  - **If *no process is waiting in its critical section* and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely**

- **Bounded Waiting?**
  - **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.**

Progress Violation of Algorithm 1

P0 (i=0, j=1)

Turn
0

P1 (i=1, j=0)

while (*turn* != *i*) ;

You can simply consider i and j are **local** variables in P0 or P1; but they are initialized with different values in P0 and P1.

Time

Progress Violation of Algorithm 1

P0 (i=0, j=1)

Turn
0

P1 (i=1, j=0)

while (*turn* != *i*) ;

Context switch

Time

Progress Violation of Algorithm 1

P0 (i=0, j=1)

Turn

P1 (i=1, j=0)

while (*turn* != *i*) ;

0

while (*turn* != *i*);

Context switch

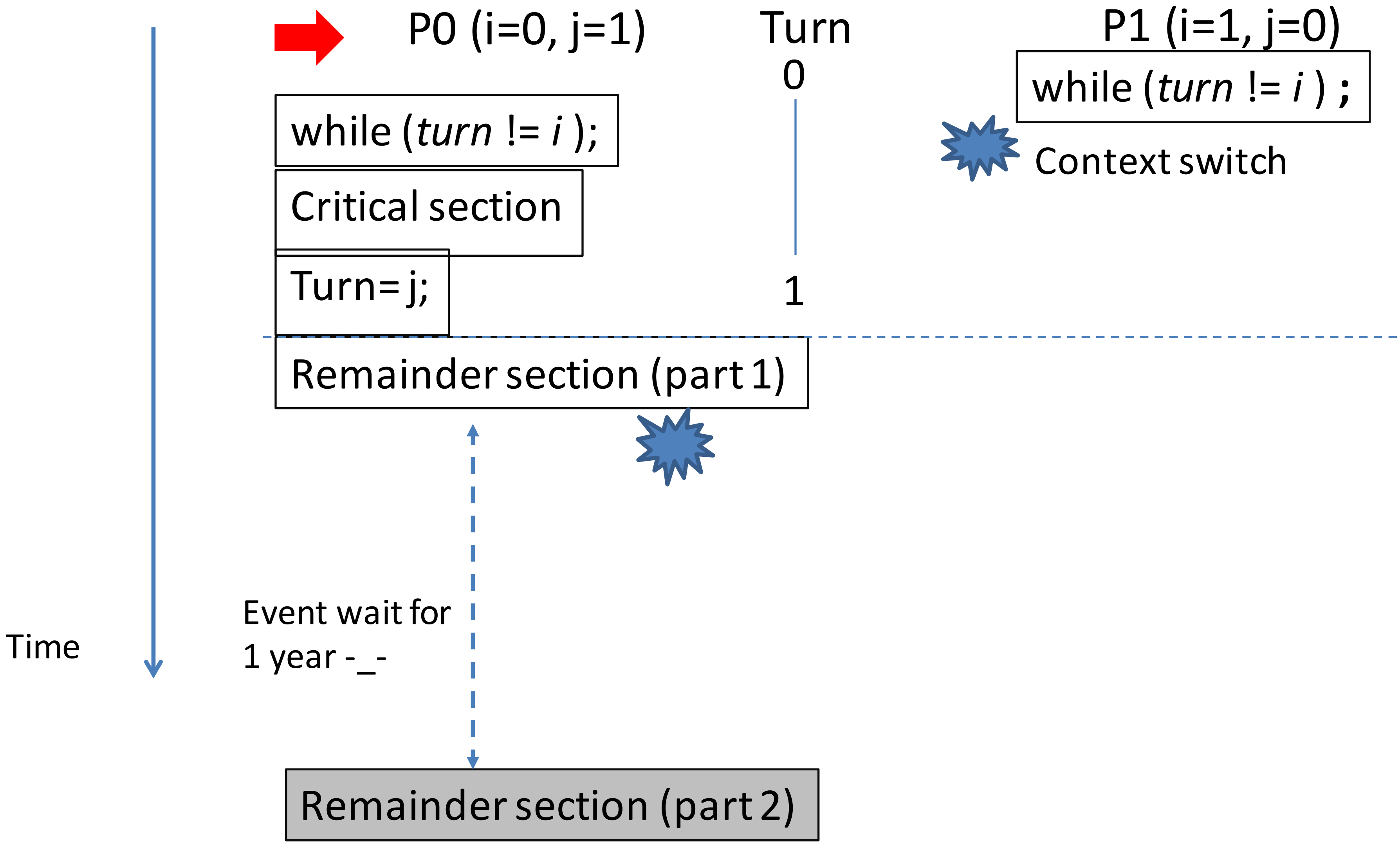
Critical section

Turn=j;

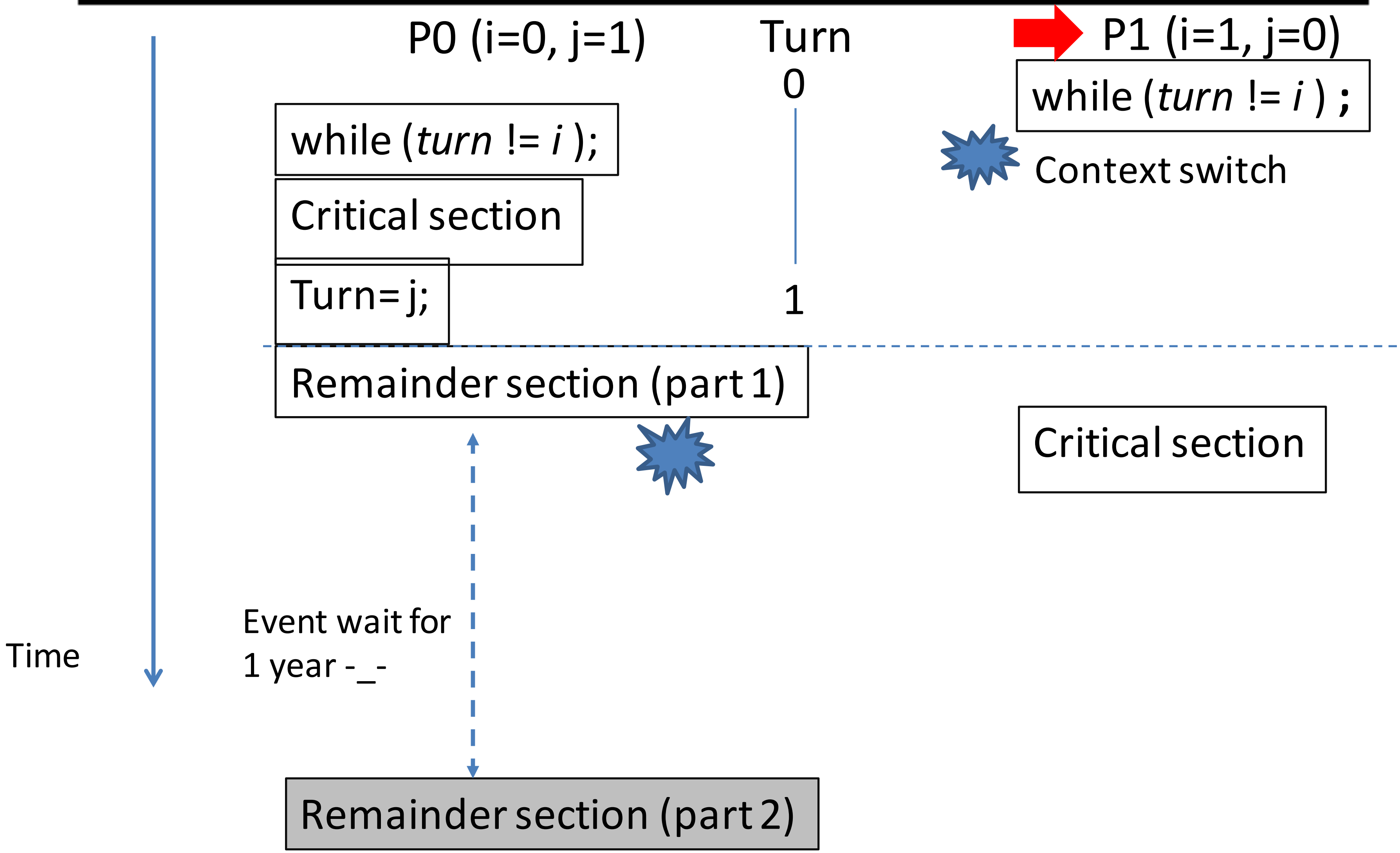
1

Time

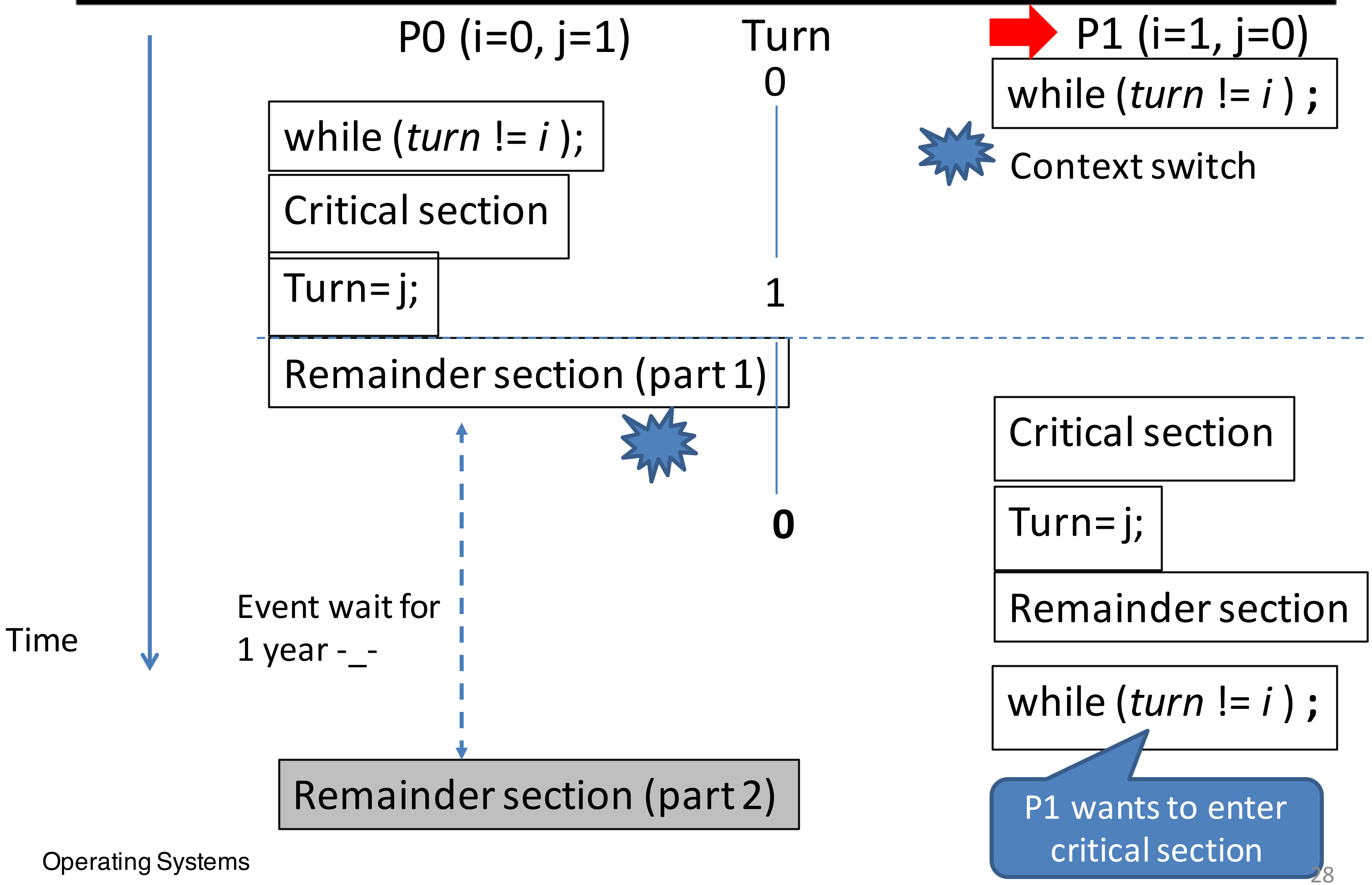
Progress Violation of Algorithm 1



Progress Violation of Algorithm 1



Progress Violation of Algorithm 1



Progress Violation of Algorithm 1

P0 (i=0, j=1) | Turn | P1 (i=1, j=0)

Turn: 0

P1: while (*turn* != *i*) ;

Context switch

P0:

while (*turn* != *i*);

Critical section

Turn=j;

Turn: 1

Remainder section (part 1)

Turn: **0**

P1:

Critical section

Turn=j;

Remainder section

while (*turn* != *i*) ;

Time

Event wait for
1 year -_-

Remainder section (part 2)

☹ **Looping,**
Progress violation!

Progress Violation of Algorithm 2

P0 (i=0, j=1)	*flag*[0]	*flag*[*1*]	P1 (i=1, j=0)
	false	*false*	

Time

Progress Violation of Algorithm 2

P0 (i=0, j=1)	*flag*[0]	*flag*[1]	P1 (i=1, j=0)
	false	*false*	
flag[*i*] = *true*;	**True**		
Context switch			

Time

Progress Violation of Algorithm 2

P0 (i=0, j=1) *flag*[0] *flag*[1] P1 (i=1, j=0)

false *false*

flag[*i*] = *true*;

True

flag[*i*] = *true*;

Context switch

True

while (*flag*[*j*]);

☹ **Looping!**

Time

Progress Violation of Algorithm 2

	P0 (i=0, j=1)	*flag*[0]	*flag*[1]	P1 (i=1, j=0)
		false	*false*	
	flag[*i*] = *true*;	**True**		
	Context switch		**True**	*flag*[*i*] = *true*;
				while (*flag*[*j*]);
				☹ **Looping!**
Time	while (*flag*[*j*]);			
	☹ **Looping!**			

☹ Progress violation!

Progress of Algorithm 3

P0 (i=0, j=1)	*flag*[0]	*flag*[*1*]	*Turn*	P1 (i=1, j=0)
	false	*false*	*0*	

Time

Progress of Algorithm 3

P0 (i=0, j=1)	*flag*[0]	*flag*[1]	*Turn*	P1 (i=1, j=0)
	false	*false*	*0*	
flag[*i*] = *true*;	***True***			
Context switch				

Time

Progress of Algorithm 3

P0 (i=0, j=1)	*flag*[0]	*flag*[1]	*Turn*	P1 (i=1, j=0)
	false	*false*	*0*	
flag[*i*] = *true*;	***True***			
		True		*flag*[*i*] = *true*;
Context switch			*0*	*turn* = *j*;
				while (*flag* [*j*] and *turn* = *j*); **Looping!**

Time

Progress of Algorithm 3

	P0 (i=0, j=1)	*flag*[0]	*flag*[1]	*Turn*	P1 (i=1, j=0)
		false	*false*	*0*	
	flag[*i*] = *true*;	**True**			
	Context switch		**True**		*flag*[*i*] = *true*;
				0	*turn* = *j*;
					while (*flag* [*j*] and *turn* = *j*); **Looping!**
	turn = *j*;			***1***	
Time	while (*flag* [*j*] and *turn* = *j*); **Looping!**				

Progress of Algorithm 3

P0 (i=0, j=1)	*flag*[0]	*flag*[1]	*Turn*	P1 (i=1, j=0)
	false	*false*	*0*	
flag[*i*] = *true*;	**True**			
Context switch		**True**		*flag*[*i*] = *true*;
			0	*turn* = *j*;
				while (*flag* [*j*] and *turn* = *j*); **Looping!**
turn = *j*;			***1***	
while (*flag* [*j*] and *turn* = *j*); **Looping!**				
				Critical section
				Progress!

Time

Mutual Exclusion of TestAndSet

lock P0 P1

false while(TestAndSet(&*lock*)); while(TestAndSet(&*lock*));

```
boolean TestAndSet (boolean *target) {
//OS disables context switches
        boolean rv = *target;
        *target = true;
        return rv;
//OS enables context switches
  }
```

True

false

Time

Mutual Exclusion of TestAndSet

lock P0 P1

false while(TestAndSet(&*lock*)); while(TestAndSet(&*lock*));

```
boolean TestAndSet (boolean *target) {
//OS disables context switches
        boolean rv = *target;
        *target = true;
        return rv;
//OS enables context switches
  }
```

True

false

Time

Keep returning true;

while(TestAndSet(&*lock*));

Mutual Exclusion of TestAndSet

lock P0 P1

false

while(TestAndSet(&*lock*));

while(TestAndSet(&*lock*));

boolean *TestAndSet* (boolean *target) {

//OS disables context switches

    boolean *rv* = **target*;

    **target* = *true*;

    return *rv*;

//OS enables context switches

}

True

false

Time

Keep returning true;

while(TestAndSet(&*lock*));

Critical section

Mutual Exclusion of TestAndSet

lock	P0	P1
false	while(TestAndSet(&*lock*));	while(TestAndSet(&*lock*));
	Return false	***Keep returning true;***
		while(TestAndSet(&*lock*));
True	Critical section	
False	*lock = false;*	

Time

Mutual Exclusion of TestAndSet

lock | P0 | P1

false

while(TestAndSet(&*lock*));

Return false

while(TestAndSet(&*lock*));

Keep returning true;

while(TestAndSet(&*lock*));

True

Critical section

False

lock = *false*;

Time

Return false;

while(TestAndSet(&*lock*));

Critical section

Critical Section for Semaphore

S (=mutex) P0 P1

1

P0:

```
wait(S){
    while (S<= 0);
    S--;
}
```

P1:

```
wait(S){
    while (S<= 0);
    S--;
}
```

Time

Critical Section for Semaphore

S (=mutex) **P0** P1

1

0

P0:

```
wait(S){
    while (S<= 0);
    S--;
}
```

P1:

```
wait(S){
    while (S<= 0);
    S--;
}
```

Time

Critical Section for Semaphore

S (=mutex)

1

0

Time

P0

```
wait(S){
    while (S<= 0);
    S--;
}
```

P1

```
wait(S){
    while (S<= 0);
    S--;
}
```

Looping

Critical Section for Semaphore

S (=mutex)

P0

1

wait(*S*){
    while (*S*<= 0);
    S--;
}

0

Critical section

signal(*S*);

1

Time

P1

wait(*S*){
    while (*S*<= 0);
    S--;
}

Looping

Wait(S) {
Line 1: perform context switch if necessary;
Line 2: OS disables context switch;
Line 3: register1= S;
Line 4: if (register 1<=0) go to **Line 1;**
Line 5: register1--;
Line 6: S=register1;
Line 7: OS enables context switch;
Line 8: perform context switch if necessary;
}

Atomic

Critical Section for Semaphore

S (=mutex) P0 P1

S (=mutex)	P0	P1
1	*wait*(*S*){	*wait*(*S*){
	while (*S*<= 0);	while (*S*<= 0); Looping
0	*S*--;	*S*--;
	}	}
	Critical section	
1	*signal*(*S*);	
		wait(*S*){
		while (*S*<= 0);
		S--;
		}
0		Critical section
		signal(*S*);

Time

Incorrect Usage of Semaphore: Deadlock

S	Q	P0	P1
1	*1*		
0	*1*	*wait(S);*	
0	***0***	*wait(Q);*	
1	*0*	*signal(S);*	
1	***1***	*signal(Q);*	
1	***0***		*wait(Q);*
0	*0*		*wait(S);*
0	***1***		signal(*Q*);
1	*1*		*signal(S);*

Time

no deadlock for this execution order.

Incorrect Usage of Semaphore: Deadlock

S	*Q*	P0	P1
1	*1*		
0	*1*	*wait*(*S*);	
0	***0***		*wait*(*Q*);
		wait(*Q*); Blocked	
			wait(*S*); Blocked

Time ↓

Deadlock for this execution order.

Bounded Buffer – Swapping wait(mutex) and wait(empty)

Producer:

```
item nextProduced;
while (1) {
    ...
    produce nextProduced;
    ...
    wait(mutex);
    wait(empty);
    ...
    add nextProduced to buffer;
    ...
    signal(mutex);
    signal(full);
}
```

Consumer:

```
item nextConsumed;
while (1) {
    wait(full)
    wait(mutex);
    ...
    nextConsumed = an item from buffer
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item nextConsumed
     ...
}
```

Time

Any problem?

Bounded Buffer – Swapping wait(mutex) and wait(empty)

Mutex

1

0

Producer:

```
item nextProduced;
while (1) {
    …
    produce nextProduced;
    …
    wait(mutex);
    wait(empty);
Blocked
}
```

Consumer:

```
item nextConsumed;
while (1) {
    wait(full);
    wait(mutex);
            Blocked
}
```

Time

Consider the buffer is full. → **Deadlock**.