Synchronization

Motivating Scenario-

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

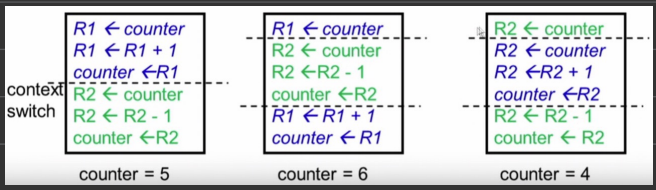
```
int counter = 5;
```

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

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

1	2	3	2	1	2	1	2

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



Race Conditions -

A situation where several processes access & 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.

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

→ critical section

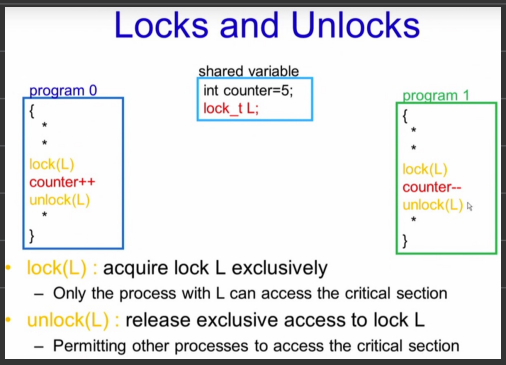
→ No more than one process should execute in critical section at a time

NOTE → Race conditions can occur in Multicore

Critical Section –

Any solution that solves it should satisfy the following –

- 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 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.



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

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

Problem with 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
}
```

- Had a common turn flag that was modified by both processes
- This required processes to alternate.
- Possible Solution : Have two flags – one for each process

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

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

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

Problem with 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
}
```

- The flag (p1_inside, p2_inside), is set after we break from the while loop.

Software Solution (Attempt 3)

globally defined

```
p2_wants_to_enter, p1_wants_to_enter
```

Process 1

```
while(1){
    p1_wants_to_enter = True
    while(p2_wants_to_enter = True);
    critical section
    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)

Attempt 3: Endless Wait

time ↓

CPU	P1_wants_to_enter	p2_wants_to_enter
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

There is a tie!!!

Both p1 and p2 will loop infinitely

Endless wait
Each process is waiting for the other
this is a deadlock

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

Deadlock

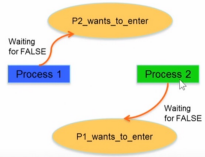
time ↓

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved
Each process is waiting for the other
this is a deadlock



Problem with Attempt 3

globally defined
p2_wants_to_enter, p1_wants_to_enter

Process 1

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

lock
unlock

- Deadlock
  Have a way to break the 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.

favored can take only two values : 1 or 2

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

Break the deadlock with a 'favored' process

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
}
```

- Deadlock broken because favored can only be 1 or 2.
  - Therefore, tie is broken. Only one process will enter the critical section
- Solves Critical Section problem for two processes

Bakery Algorithm (for #processes >2)

By Leslie Lamport.
Inspired from bakeries & banks token system. Everyone has to wait for their turn to come.

* Simplified Bakery algorithm.
Processes numbered from 0 - N-1
num (globally shared with each process) is an array N integers (initially 0)
- Each entry corresponds to a process.

To enter a critical section we need to execute the lock call.

assuming ~~ensure~~ that no 2 process gets same token
No contention switch occurs

```
lock(i) {
    num[i] = MAX(num[0], num[1] ..... num[N-1]) + 1
    for (p=0; p < N; ++p) {
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

Good way

CRITICAL SECTION

```
unlock(i) {
    num[i] = 0
}
```

But what if 2 values have the same value?? We won't be able to gain mutual exclusion.

for example

P_1	P_2	P_3	P_4	P_5	
0	0	0	0	0	num
0	0	1	0	0	
0	0	1	2	0	
0	0	1	2	3	
0	4	1	2	3	

assuming every one wants the lock simultaneously.

Let P_3 be first
Then P_4 executes
Then P_5 executes
Then P_2 executes

Original Bakery algorithm →

- Without atomic operation assumption
- Introduce an array of booleans of size N.

```
lock(i) {
    choosing[i] = true
    num[i] = max(num[0], num[1] ..... num[N-1]) + 1
    choosing[i] = false
    for (p = 0; p < N; p++) {
        while (choosing[p]);
        while (num[p] != 0 and (num[p], p) < (num[i], i))
    }
}
```

good way

choosing ensures that process is not at the doorway.

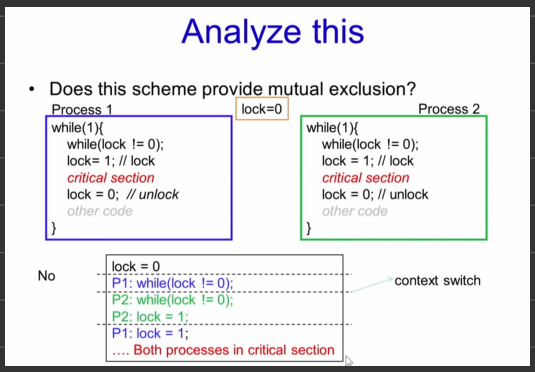
favour one process when there is a conflict i.e. favour smaller id.

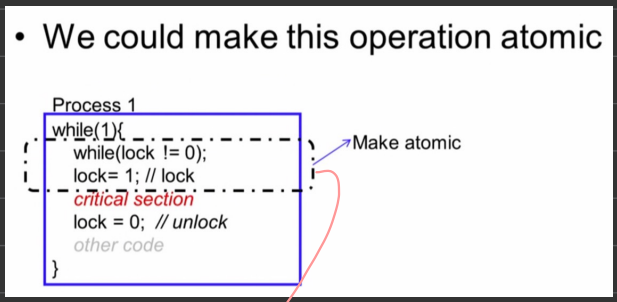
CRITICAL SECTION

```
Unlock(i) {
    num[i] = 0
}
```

NOTE $(a, b) < (c, d)$ is equivalent to $(a < c)$ or $((a == c)$ and $(b < d))$

Hardware Lock





if these become atomic i.e. no context switch until these are executed then we can avoid previous issues

We can take use of hardware to make them atomic.

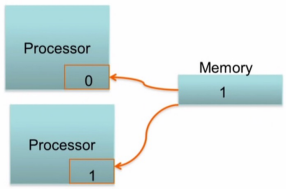
Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation (the entire function is executed atomically)



Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation (the entire function is executed atomically)

Usage for locking

```
while(1){
    while(test_and_set(&lock) == 1);
    critical section
    lock = 0;  // unlock
    other code
}
```

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0

if we wanted to do it via hardware

necessary

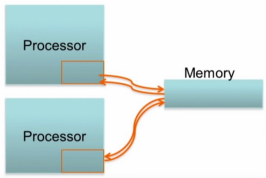
But Intel system doesn't support anything like test and set they have an instruction called as exchange.

Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem




```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

used to lock

used to unlock

This xchg is used to construct spinlocks, mutex & semaphores

Spinlocks

Spinlocks Usage

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

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
   locked = 0;
}
```

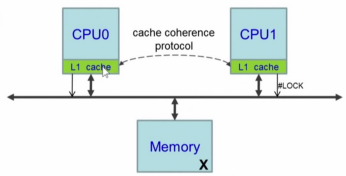
Issues with Spinlocks

xchg %eax, X

- No compiler optimizations should be allowed
  - Should not make X a register variable
    - Write the loop in assembly or use volatile
- Should not reorder **memory** loads and stores
  - Use serialized instructions (which forces instructions not to be reordered)
  - Luckly xchg is already implements serialization

More issues with Spinlocks

xchg %eax, X



- No caching of (X) possible. All xchg operations are bus transactions.
  - CPU asserts the LOCK, to inform that there is a 'locked ' memory access
- acquire function in spinlock invokes xchg in a loop…each operation is a bus transaction …. huge performance hits

A better acquire

```
%eax = value
xchg %eax, (addr)
}
```

```
void acquire(int *locked){
  reg = 1
  while(1)
     if(xchg(locked, reg) == 0)
        break;
}
```

Original.
Loop with xchg.
Bus transactions.
Huge overheads

```
void acquire(int *locked) {
  reg = 1;
  while (xchg(locked, reg) == 1)
      while (*locked == 1);
}
```

Better way
inner loop allows caching of
locked. Access cache instead of memor

Mutexes

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

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
   locked = 0;
   wakeup();
}
```

→ wakes up all processes which are sleeping

Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

Thundering Herd Problem

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
      sleep();
    }
  }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```

wake only one process

Locks and Priorities

- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - Priority Inversion
  - Possible solution
    - Priority Inheritance

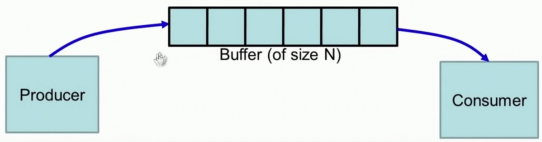
In this the priority of low priority task is escalated.

Semaphores

Producer – Consumer Problems

motivating problem

- Also known as *Bounded buffer Problem*
- Producer produces and stores in buffer, Consumer consumes from buffer
- Trouble when
  - Producer produces, but buffer is full → when producer is fast
  - Consumer consumes, but buffer is empty → when consumer is fast



we need to maintain synchronization.

Producer-Consumer Code

Buffer of size N
int count=0;
Mutex mutex, empty, full;

```
void producer(){
  while(TRUE){
    item = produce_item();
    if (count == N) sleep(empty);
    lock(mutex);
    insert_item(item); // into buffer
    count++;
    unlock(mutex);
    if (count == 1) wakeup(full);
  }
}
```

```
void consumer(){
  while(TRUE){
    if (count == 0) sleep(full);
    lock(mutex);
    item = remove_item(); // from buffer
    count--;
    unlock(mutex);
    if (count == N-1) wakeup(empty);
    consume_item(item);
  }
}
```

this can cause issues during context switch because at register level these are 2 statements

Lost Wakeups

- Consider the following context of instructions
- Assume buffer is initially empty

```
3 read count value // count <- 0
3 item = produce_item();
5 lock(mutex);
6 insert_item(item); // into buffer
7 count++;  // count = 1
8 unlock(mutex)
9 test (count == 1)  // yes
9 signal(full);
3 test (count == 0) // yes
3 wait();
```

context switch

Note, the wakeup is lost.
Consumer waits even though buffer is not empty.
Eventually producer and consumer will wait infinitely

consumer
still uses the old value of count (ie 0)

So even using 3 mutexes can't fully solve it.

Semaphores

- Proposed by Dijkstra in 1965
- Functions down and up must be atomic
- down also called **P** (Proberen Dutch for try)
- up also called **V** (Verhogen, Dutch form make higher)
- Can have different variants
  - Such as blocking, non-blocking
- If S is initially set to 1,
  - Blocking semaphore similar to a Mutex
  - Non-blocking semaphore similar to a spinlock

```
void down(int *S){
  while( *S <= 0);
  *S--;
}

void up(int *S){
  *S++;
}
```

shared in memory

Both are supposed to be atomic.

way sleep(?)

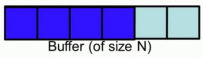
Producer-Consumer with Semaphores

Buffer of size N
int count;

full = 0, empty = N → initial value

Number of filled blocks in the buffer

Number of empty blocks in the buffer



Buffer (of size N)

N = 6
fill = 4
empty = 2

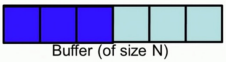
Producer-Consumer with Semaphores

Serializing Access to the Buffer

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);
        lock(mutex)
        insert_item(item); // into buffer
        unlock(mutex)
        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);
        lock(mutex)
        item = remove_item(); // from buffer
        unlock(mutex)
        up(empty);
        consume_item(item);
    }
}
```

to prevent the ~~buffer~~.



Buffer (of size N)

N = 6
fill = 3

empty = 3