The picture can't be displayed.

Chapter - 6 Process Synchronization

Background

- Processes can execute concurrently
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Process Synchronization: Objectives

- Concept of process synchronization.
- The critical-section problem, whose solutions can be used to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
- Classical process-synchronization problems
- Tools that are used to solve process synchronization problems

What will Cover

- Process Synchronization basic Concepts
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic problems of synchronization

Process Synchronization

- Process Synchronization means sharing system resources by processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.
- Process Synchronization was introduced to handle problems that arose while multiple process executions.

Illustration….

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers(actually support bounded buffer). We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true)
  {
                    /* produce an item and put in nextProduced
                   while (count == BUFFER_SIZE)
                           ; // do nothing
                   buffer [in] = nextProduced;
                   in = (in + 1) % BUFFER_SIZE;
                   count++;
  }
```

Consumer

```
while (1)
 {
                while (count == 0)
                         ; // do nothing
                nextConsumed =  buffer[out];
                out = (out + 1) % BUFFER_SIZE;
                count--;
                /*  consume the item in nextConsumed
 }
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

Race Condition

- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called **race condition.**

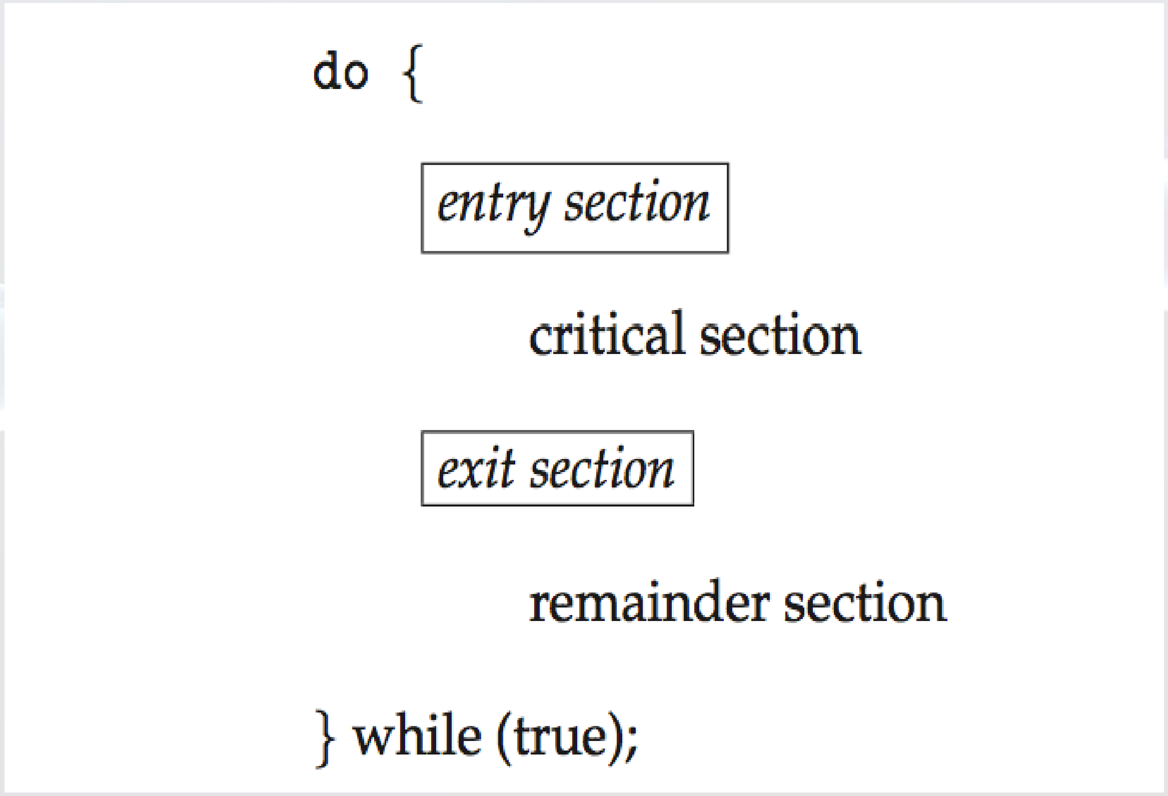
- **To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee , we require that the processes be synchronized in some way.**

Critical Section Problem

- Consider system of *n* processes {p_0, p_1, **…** $\boldsymbol{p}_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- *Critical section problem* is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

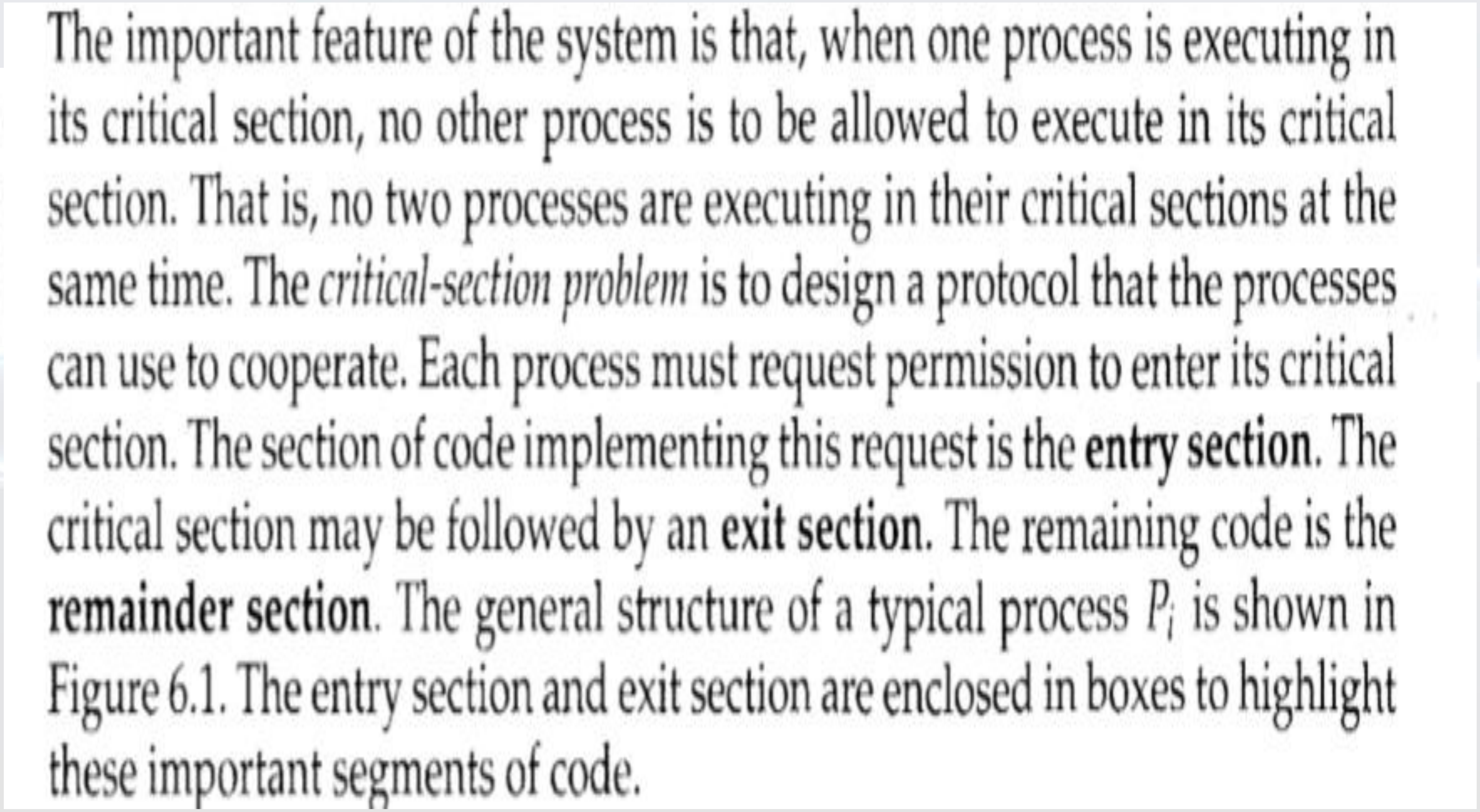
Critical Section

- General structure of process P_i



```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Critical Section Problem



The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed

Peterson's Solution

- A classic software based solution to the critical section problem known as Peterson's solution
- **Two process solution**
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two data items to be shared between the two process:
  - 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 P_i is ready!

Peterson's Solution for process P_i

```
do {
    flag[i] = TRUE ;
     turn = j ;
     while ( flag[j] && turn == j) ;
Do no-op
            CRITICAL SECTION

    flag [ i ] = FALSE ;

             REMAINDER SECTION

  } while (TRUE) ;
```

Problem

- For Peterson’s problem below conditions will applied.
- Each statement will take 2ms to complete.
- For process 0: i=0,j=1; and for process 1: i=1,j=0.
- Context switching will occur after 2ms.
- In critical section area carried only 3 statements.
- In remainder section area carried only 2 statements.
- Initial information common to both processes:
- turn=0;
- flag[0]=FALSE;
- flag[1]=FALSE;

Synchronization Hardware

- Many systems provide **hardware support** for critical section code
- Uniprocessors – could disable interrupts while modified a shared variable
  - Currently running code would execute without preemption. So, no unexpected modifications could be made to the shared variable.
  - This is the approach taken by non-preemptive kernels.
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable-time consuming
- Modern machines provide special atomic hardware instructions: TestAndSet ; Swap ;
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
            critical section
    release lock
            remainder section
} while (TRUE);
```

- A simple tool requires-lock
- Race conditions are prevented by requiring that critical section be protected by locks.

Analyze this

- Does this scheme provide mutual exclusion?

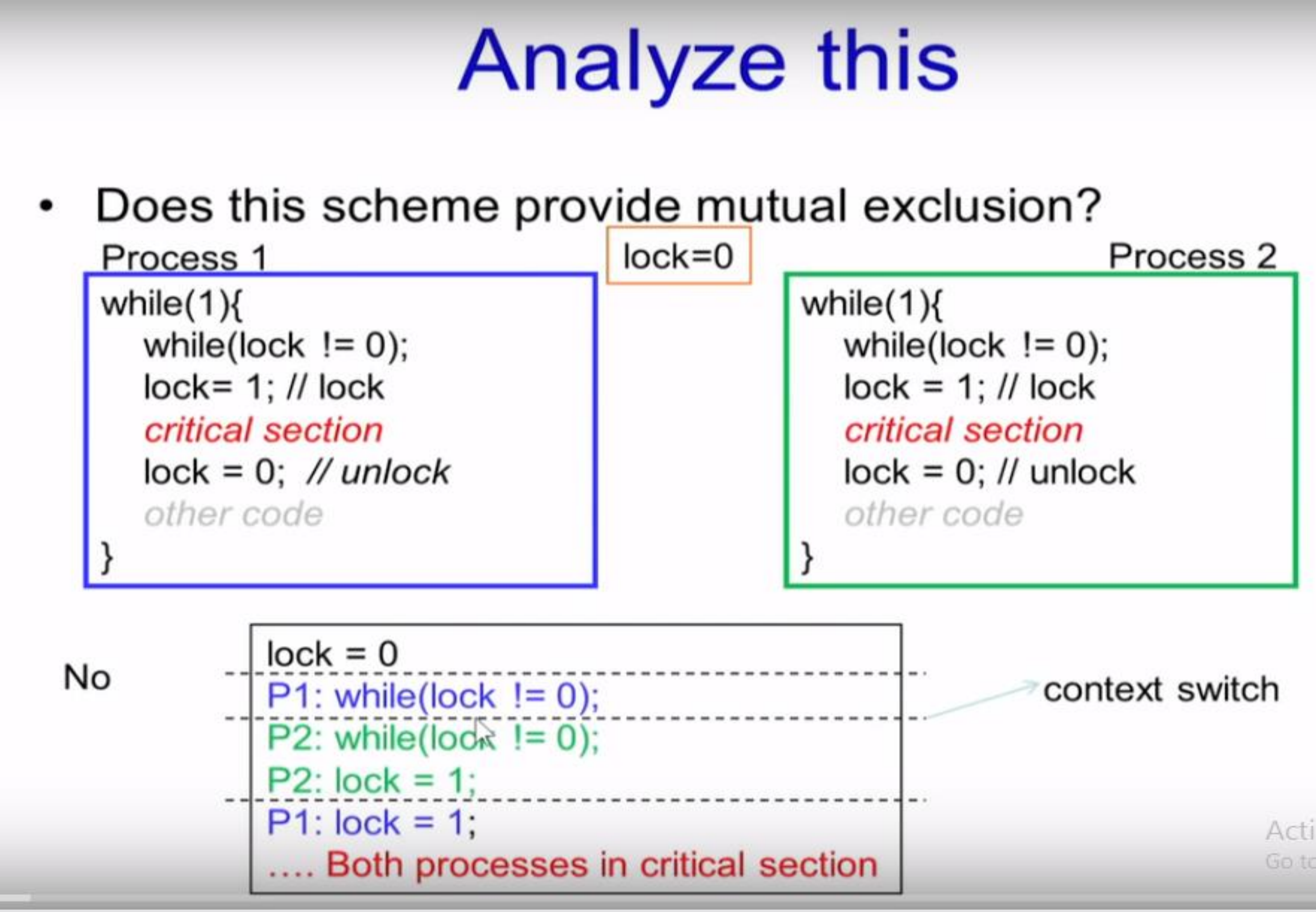
lock=0

Process 1

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

Process 2

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



Analyze this

- Does this scheme provide mutual exclusion?

lock=0

Process 1

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

Process 2

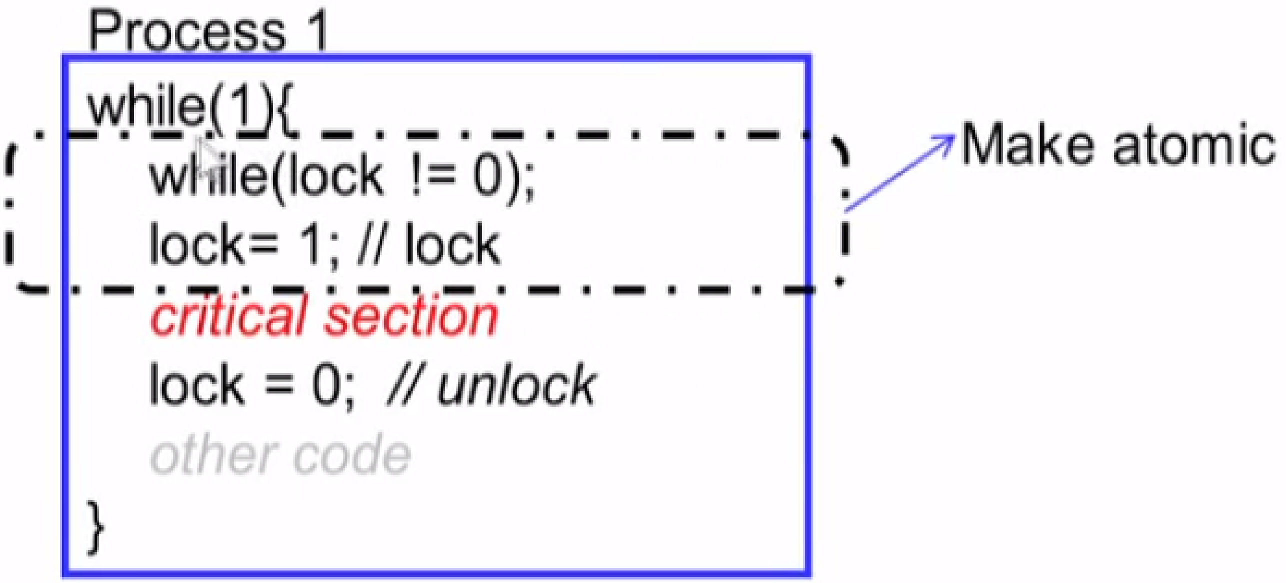
```
while(1){
   while(lock != 0);
   lock = 1; // lock
   critical section
   lock = 0; // unlock
   other code
}
```

No

```
lock = 0
-------------------------------------
P1: while(lock != 0);
-------------------------------------   context switch
P2: while(lock != 0);
P2: lock = 1;
-------------------------------------
P1: lock = 1;
.... Both processes in critical section
```

If only...

- We could make this operation atomic



Process 1

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

test_and_set Instruction

Definition:

```
boolean test_and_set (Boolean *target)
        {
            boolean rv = *target;
            *target = TRUE;
            return rv:
        }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do{
    while(test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */

 } while (true);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
 {
     boolean temp = *a;
     *a = *b;
     *b = temp:
 }
```

Solution using Swap

- **Shared Boolean variable lock** initialized to FALSE; Each process has a **local Boolean variable key**.
- Solution:

```
    do {
        key = TRUE;
         while ( key == TRUE)
               Swap (&lock, &key );

               //    critical section

         lock = FALSE;

             //      remainder section

     } while ( TRUE);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Semaphore Variables

- Hardware based solution to the Critical Section (CS) problem are complicated for application programmer to use. To overcome this difficulty, a synchronization tool called **semaphore** can be used.
- A **semaphore** is a protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment.
- **Semaphore** S – integer variable ; can not modify directly. Using two standard operations we can modify wait() and signal()
- Originally called **P()** and **V()** (Less complicated)

Semaphore Variable Defination

```
S: Semaphore;
    S.wait(): while (S ≤ 0) do skip;
                   S = S-1; //Outside the While Loop

    S.signal(): S = S+1;
```

S.wait() and S.signal() operations are atomic in nature that means when a process executing the process S.wait() or S.signal() that can not be interrupted.

Using semaphore we can implement mutual exclusion very easily.

Semaphore Variable

```
P(S): while S ≤ 0 do skip;
            S = S-1;
V(S): S = S+1;
```

- Implementation with mutex lock:

  S = 1;

```
P_i:    S.wait()
        CS              //Critical Section
        S.signal()
        RS              //Remainder Section
```

Two types of semaphore

- **Counting semaphore** –If there are more than one but limited resources, integer value can range over an unrestricted domain
- **Binary semaphore** – If there is a single resource(CS) only , integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks/ Same as a **mutex lock**
- Binary semaphore initialized to 1,Provides mutual exclusion
- *Can solve various synchronization problems:*
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**`synch`**" initialized to 0

P1:

    S_1;

    signal(synch);

P2:

    wait(synch);

    S_2;

- https://www.youtube.com/watch?v=DvF3AsTglUU

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Problem!!!

- Processes waiting on a semaphore must constantly check to see if the semaphore is not zero. This continual looping is clearly a problem in a real multiprogramming system (where often a single CPU is shared among multiple processes).
- This is called **busy waiting** and it wastes CPU cycles. When a semaphore does this, it is called a **spinlock**.

Semaphore Implementation with no Busy waiting

- To avoid busy waiting, each semaphore there is an associated waiting queue of process that are waiting to access the critical section.
- Rather than using a semaphore as a variable, we can use it as a structure or record which have two fields:
  - value (of type integer)
  - list of the processes/ pointer to next record in the list
- Two operations:
  - **block() & wakeup ()**
  - `typedef struct{`

```
int value;
struct process *list;
} semaphore;
```

Semaphore Implementation with no Busy waiting

- Type declaration of semaphore as a record or structure:

```
type semaphore = record
                        value : integer;
                        L : list of processor;
            end;
```

Semaphore Implementation with no Busy waiting

- Two operations provide: block & wakeup
  - OS provide the block() system call, which suspends the process that calls it, and the wakeup() system call which resumes the execution of blocked process P.

- "No busy waiting" means that whenever the process wakes up from waiting, the condition it was waiting for should hold. That is, it must not wake up, find the condition false, and again wait on the same semaphore.

Semaphore Implementation with no Busy waiting

```
Structure semaphore{
        int value;
        queue L;
}
   wait (S){
            if (s.value > 0) s.value=s.value-1;
            else {
                add this process S.L to waiting queue
                block();
            }
   }

  Signal (S){
                if (S.L != Empty) {
                      remove a process S.L from the waiting queue;
                      wakeup(P);
                }
                else S.value=S.value+1;
  }
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                          P1
    wait (S);                   wait (Q);
     wait (Q);                  wait (S);
       .                           .
       .                           .
       .                           .
    signal  (S);                signal (Q);
    signal (Q);                 signal (S);
```

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full
- and that the consumer won't try to remove data from an empty buffer.

Solution for Bounded-Buffer Problem

- The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
- In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

Inadequate implementation

- In the solution, two library routines are used, sleep and wakeup. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable itemCount holds the number of items in the buffer.

Inadequate implementation

```
procedure producer() {
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}
```

```
procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

The problem with this solution is that it lead to a deadlock

It contains a race condition

1. The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases itemCount.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. **This is because the consumer is only awakened by the producer when itemCount is equal to 1.**
6. The producer will loop until the buffer is full, after which it will also go to sleep.

Using semaphores

- Semaphores solve the problem of lost wakeup calls by using two semaphores, fillCount and emptyCount.
- fillCount is the number of items already in the buffer and available to be read.
- emptyCount is the number of available spaces in the buffer where items could be written.
- fillCount is incremented and emptyCount decremented when a new item is put into the buffer.
- If the producer tries to decrement **emptyCount** when its value is zero, the producer is put to sleep.
- The next time an item is consumed, **emptyCount is incremented** and the producer wakes up. The consumer works analogously.

Using semaphores

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining space
```

```
procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
        putItemIntoBuffer(item);
        up(fillCount);
    }
}
```

```
procedure consumer() {
    while (true) {
        down(fillCount);
        item = removeItemFromBuffer();
        up(emptyCount);
        consumeItem(item);
    }
}
```

The solution works fine when there is only one producer and consumer

With multiple producers and consumers

- this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time.
    - It could contain two actions, one determining the next available slot and the other writing into it.

With multiple producers and consumers

- If the procedure can be executed concurrently by multiple producers, then the following scenario is possible:
  - Two producers decrement emptyCount
  - One of the producers determines the next empty slot in the buffer
  - Second producer determines the next empty slot and gets the same result as the first producer
  - Both producers write into the same slot

Using semaphores

To overcome this problem, we need a way to make sure that only one producer is executing putItemIntoBuffer() at a time.

```
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;
procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            down(mutex);
                putItemIntoBuffer(item);
            up(mutex);
        up(fillCount);
    }
}
```

```
procedure consumer() {
    while (true) {
        down(fillCount);
            down(mutex);
                item = removeItemFromBuffer();
            up(mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

Readers-Writers Problem

Definition

- There is a data area that is shared among a number of processes.
- Any number of readers may simultaneously read to the data area.
- Only one writer at a time may write to the data area.
- If a writer is writing to the data area, no reader may read it.
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write
- A process that reads and writes to a data area must be considered a writer (consider producer or consumer)

Readers-Writers Problem

- Problem!!!
- allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- If a writer and some other thread access the database simultaneously, chaos may ensure.
- To ensure that these difficulties do not arise, we require that the writer have exclusive access to the shared database.
- This synchronization problem is referred to as readers-writers problems.

Readers-Writers Problem

- As solution to either problem may result in starvation.
- In the first case writers may starve and in the second case readers may starve.
- For this reason other variant of the problem have been proposed.
  - *the third readers-writers* adds the constrain that know thread shall be allowed to starve; that is the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.

Readers-Writers Problem

- The solution of the first Reader-writer problem:

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.
  - The semaphore wrt is common to both reader and writer processes.
  - The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object.

Readers-Writers Problem (Cont.)

The structure of a **reader** process

```
do  {
      wait (mutex) ;
      readcount ++ ;
      if (readercount == 1)  wait (wrt) ;
      signal (mutex)

                  // reading is performed

      wait (mutex) ;
      readcount -- ;
      if (redacount  == 0)  signal (wrt) ;
      signal (mutex) ;
} while (true)
```

● Needs mutually exclusive access while manipulating "readers" variable

● Does not need mutually exclusive access while reading database

● If this reader is the first reader, it has to wait if there is an active writer (which has exclusive access to the database) n First reader did a "P(write)"

● If other readers come along while the first one is waiting, they wait at the "P(mutex)"

● If other readers come along while the first one is actively reading the database, they can also get in to read the database

● When the last reader finishes, if there are waiting writers, it must wake one up

Readers-Writers Problem (Cont.)

- The structure of a **writer** process

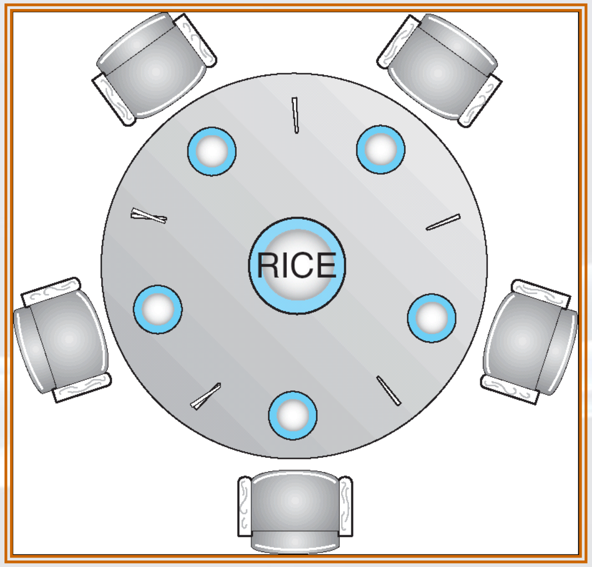
```
do  {
     wait (wrt) ;

     //    writing is performed

    signal (wrt) ;
} while (true)
```

- If there is an active writer, other writer has to wait (the active writer has exclusive access to database)
- If there are active readers, this writer has to wait (readers have priority) n First reader did a "P(write)"
- The writer only gets in to write to the database when there are no other active readers or writers
- When the writer finishes, it wakes up someone (either a reader or a writer it's up to the CPU scheduler)
- If a reader gets to go next, then once it goes through the "V(mutex)" and starts reading the database, then all other readers waiting at the top "P(mutex")" get to get in and read the database as well

Dining-Philosophers Problem

- Problem statement
- Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.
- Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply is assumed. assuming that no philosopher can know when others may want to eat or think.

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
Do  {
      wait ( chopstick[i] );
      wait ( chopStick[ (i + 1) % 5] );

           //  eat

      signal ( chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

          //  think

} while (true) ;
```

Dining-Philosophers Problem

- Must satisfy mutual exclusion - no two philosopher can use the same fork at the same time.
- Avoid deadlock and starvation!

Dining-Philosophers Problem

- **First attempt**: take left fork, then take right fork

  Wrong! Results in deadlock.

- **Second attempt**: take left fork, check to see if right is available, if not put left one down. Still has race condition and can lead to starvation.

Suggestion to solve the problem

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- An odd philosopher picks up her left chopstick first and even philosopher picks up her right chopstick first.

The picture can't be displayed.

End of Chapter 6
Thank You