

## Inter-process Communication

- IPC involves facilitating explicit or implicit cooperation
- How does OS facilitates IPC
- What is involves in IPC

Operating Systems

## IPC Synchronization

- The Critical-Section Problem
- Software Solutions
- Hardware support
- Semaphores
- Monitors
- Synchronization Examples

Operating Systems

## Critical Section Example: Implementation of Producer Consumer problem

- Suppose that we want to fill **all** the shared buffer cells.
- Use 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 entry,
  - It is decremented by the consumer after it consumes a buffer entry.
- *Note: previous implementations were based on use of n-1 cells out of n cells of the buffer.*

Operating Systems

## Producer: Earlier Algorithm

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

Operating Systems

## Consumer: Earlier Algorithm

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

Operating Systems

## Any Problems?

- What are the data structures shared?
  - BUFFER
  - COUNT
  - IN
  - OUT
- Any process may be interrupted because of many reasons, controlled by the processes and the operating system.
  - Few reasons
    - Time sharing interrupt
    - I/O start or completion interrupt
    - Alarm interrupt,
    - Etc..

Operating Systems

### Race Condition! Especially update of the shared objects used to control the sharing of other resources

- Shared variable `count++` could be implemented as

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

- Share variable `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}

### Solving the shared memory problem: First try: Alternating use of critical region Operation on the shared variable are named as "Critical Section"

```
initialize turn=0
do {
  while (turn is not equal to 0);
  critical section
  turn = 1;
  remainder section
} while(1);
```

```
do {
  while (turn is not equal to 1);
  critical section
  turn = 0;
  remainder section
} while(1);
```

### Solution to Critical-Section Problem

Required Conditions of using Critical Sections:

- Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
- Progress** - If no process is executing in the critical section and there exist some processes that wish to enter the critical section, then the selection of the processes that will enter the critical section next, cannot be postponed indefinitely
- Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter the critical sections after a process has made a request to enter the critical section:
  - Assume that each process executes at a nonzero speed
  - There is no assumption concerning relative speed of the **executing** processes

### Software Solution: Example: Peterson's Critical Section Solution

- Assume there are two independent but concurrent processes compete to enter the critical section
  - Whichever is ready is able to enter the CS.
  - Solution:
    - The two processes share two variables:
      - `int turn`; it indicates whose turn it is to enter the critical section.
      - Boolean `flag[2]`: it is used to indicate if a process is intending or ready to enter the critical section; initially it is set to **false**.
- $(flag[i] == true)$  implies that process  $P_i$  is ready!

### Peterson's Algorithm for Process $P_i$ (0 or 1)

```
do {
  flag[i] = TRUE;
  turn = i;
  while ( flag[j] && turn == j);

  CRITICAL SECTION

  flag[i] = FALSE;

  REMAINDER SECTION

} while (TRUE);
```

### Critical Section solution for n processes: Bakery Algorithm

- Processes take **consistent** and **unique** numbers first;
- Principle:** the smallest number process can enter the critical section.
- Data structures used:
- choosing**: shared array[0..n-1] of boolean;
  - number**: shared array[0..n-1] of integer; ...

## Bakery Algorithm: Pseudo Code

```
3 repeat // process i takes unique number
4   choosing[i] := true;
5   number[i] := max(number[0],number[1],...,number[n-1]) + 1;
6   choosing[i] := false;
7   // Check if it is possible to enter the Critical Section
8   for j := 0 to n-1 do begin
9     while choosing[j] do (* nothing *);
10    while number[j] <> 0 and (number[j], j) < (number[i],i) do
11      (* nothing *);
12  end;
13  // enter the critical section
14  (* critical section *)
15  // leave the critical section
16  number[i] := 0;
17  (* remainder section *)
18 until false;
```

"(a,b) < (c,d)" anami if a < c or if a = c and b < d

Operating Systems

## Why hardware support?

Process synchronization must guarantee safety and efficiency

- Software solutions are insecure and slow
- This is why hardware support is required

Operating Systems

## Process Synchronization

- Many systems provide hardware support for critical section code. The hardware support is used to implement correct CS solutions in software. Otherwise, software solutions cannot be guaranteed across different hardware platform.
- One such hw support is interrupt disable /enable pair of instructions
- Generally this is too inefficient on even in uniprocessor platforms, let alone multiprocessor ones.
- Modern machines provide special **atomic (indivisible)** hardware instructions, for such purposes.
  - ▶ Atomic = non-interruptable
  - For example, test a memory word and set to a value as one instruction: such as Test&Set(&a) instruction
  - Or swap contents of two memory words as one instruction: such as Swap(&a,&b)

Operating Systems

## TestAndSet Instruction

n Definition Algorithm: Indivisible  
boolean TestAndSet (boolean  
\*target) // target is global

```
{
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

Operating Systems

## Critical Section Solution using TestAndSet

- Lets use Shared global Boolean variable **lock** as a key to the critical section, initialized to **false**. **lock** is the parameter to TestandSet instruction.
- How to use this instruction to act as the key to CS problem?
- Solution:

```
do {
  while ( TestAndSet (&lock ))
    ; /* do nothing
  // critical section
  lock = FALSE;
  // remainder section
} while ( TRUE);
```

Operating Systems

## Indivisible Swap Instruction

- Definition Algorithm: swap a and b :

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

Operating Systems

## CS Solution using Swap

- Lets use Shared Boolean variable lock initialized to **FALSE**;
- Each process has a local Boolean variable key, and lock as global variable.
- Solution:

```
do {
    key = TRUE;
    while ( key == TRUE) Swap (&lock, &key);
        // critical section
    lock = FALSE;
        // remainder section
    } while ( TRUE);
```

Operating Systems

## Semaphore: Higher Level Synchronization tool

- Synchronization tool that may prevent busy waiting
- Semaphore is a special integer variable, say S
- Two standard indivisible operations modify S: **wait()** and **signal()**
  - Originally called **P()** and **V()**
- Semaphore Can only be accessed via two OS provided indivisible (atomic) operations (or functions)

On entrance to Critical Section (CS):

```
● wait (S) {
    while S <= 0
        ; // no-op, busy wait
    S--;
}
```

On completion of CS:

```
● signal (S) {
    S++;
}
```

Operating Systems

## Semaphore type

- **Counting** semaphore: integer value can range over an unrestricted integer domain
- **Binary** semaphore: integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Counting semaphore S with 0 and 1 values, can be used as a binary semaphore, to provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);  
Critical Section
  - signal (S);

Operating Systems

## Semaphore Implementation

- Must guarantee that no two processes can execute **wait ()** and **signal ()** on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section.
  - ▶ **wait** and **signal** implementations are with busy wait
  - ▶ but, it requires very small busy waiting
- Applications can use busy wait implementations, provided the busy wait is short...

Operating Systems

## None Busy Wait Semaphore Implementation

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - Block: place the process invoking the operation on the appropriate waiting queue.
  - Wakeup: remove one of processes in the waiting queue and place it in the ready queue.

Operating Systems

## Semaphore with Non Busy waiting (Cont.)

- Each semaphore initialized to an integer value and a queueing process...
- Implementation of wait:

```
wait (S)
{
    S.value--;
    if (S.value < 0)
    { add P to S.waitQueue(P); }
}
```

Operating Systems

## Semaphore with Non Busy waiting (Cont.)

- Implementation of signal:

```
Signal (S)
{
  S.value++;
  if (S.value >= 0)
  { remove a Process P
    from S.waitQueue (P); }
}
```

Operating Systems

## Problematic Semaphores Use

- **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

$P_0$	$P_1$
wait (S);	wait (Q);
wait (Q);	wait (S);
CS	CS
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Bad Scenario:** P0 grabs wait(S), P1 grabs wait(Q) before P0 can do it..
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Operating Systems

## Correct Semaphores Use

- Let S and Q be two semaphores initialized to 1

$P_0$	$P_1$
wait (S);	wait (S);
wait (Q);	wait (Q);
CS	CS
signal (S);	signal (S);
signal (Q);	signal (Q);

- P0 grabs S on wait(S), P1 waits for S at wait(S) until it released by the related Signal(S)..
- **No Starvation** – No blocking. P1 will be added to S.WaitQueue,; P1 will be removed by P0's execution of Signal(S) and will be put on the ReadyQueue.

Operating Systems