Process Synchronization
IPC (Inter-process Communication)

Prepared By,
Prof. Chetan K. Solanki
Asst. Prof., COED, CKPCET.

Outline

- The principles of concurrency
- Interactions among processes
  - Race Condition
  - Critical Section
- Mutual exclusion problem
- Mutual exclusion- solutions
  - Software approaches (Peterson's)
  - Hardware support (test and set atomic operation)
  - OS solution (semaphores)
  - PL solution (monitors)
  - Distributed OS solution (message passing)
- Reader/writer problem
- Dining Philosophers Problem

Principles of Concurrency

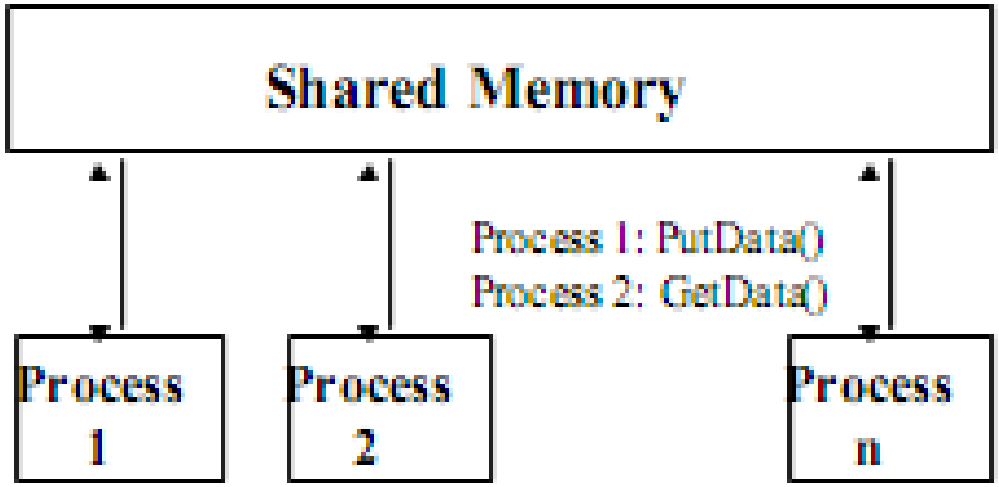
- Interleaving and overlapping the execution of processes.
- Consider two processes P1 and P2 executing the function *echo*:

```
{
  input (in, keyboard);
  out = in;
  output (out, display);
}
```

Concurrency (cont...)

- P1 invokes *echo,* after it inputs into *in* , gets interrupted (switched).
- P2 invokes *echo*, inputs into *in* and completes the execution and exits. When P1 returns in is overwritten and gone.
- Result: first character is lost and second character is written twice.
- This type of situation is even more probable in multiprocessing systems where real concurrency is realizable thru' multiple processes executing on multiple processors.
- Solution: Controlled access to shared resource
  - Protect the shared resource : *in* buffer; "critical resource"
  - one process/shared code. "critical region"

Interprocess Communication



- Processes frequently need to communicate with other processes
- Use shared memory
- Need a well structured way to facilitate interprocess communication
  - Maintain integrity of the system
  - Ensure predicable behavior
- Many mechanisms exist to coordinate interprocess communication.

Purposes for IPC

- Data Transfer
- Sharing Data
- Event notification
- Resource Sharing and Synchronization
- Process Control

Interactions among processes

In a multi-process application these are the various degrees of interaction:

1. **Competing processes**: Processes themselves do not share anything. But OS has to share the system resources among these processes "competing" for system resources such as disk, file or printer.

   Co-operating processes : Results of one or more processes may be needed for another process.

2. **Co-operation by sharing** : Example: Sharing of an IO buffer. Concept of critical section. (indirect)
3. **Co-operation by communication** : Example: typically no data sharing, but co-ordination thru' synchronization becomes essential in certain applications. (direct)

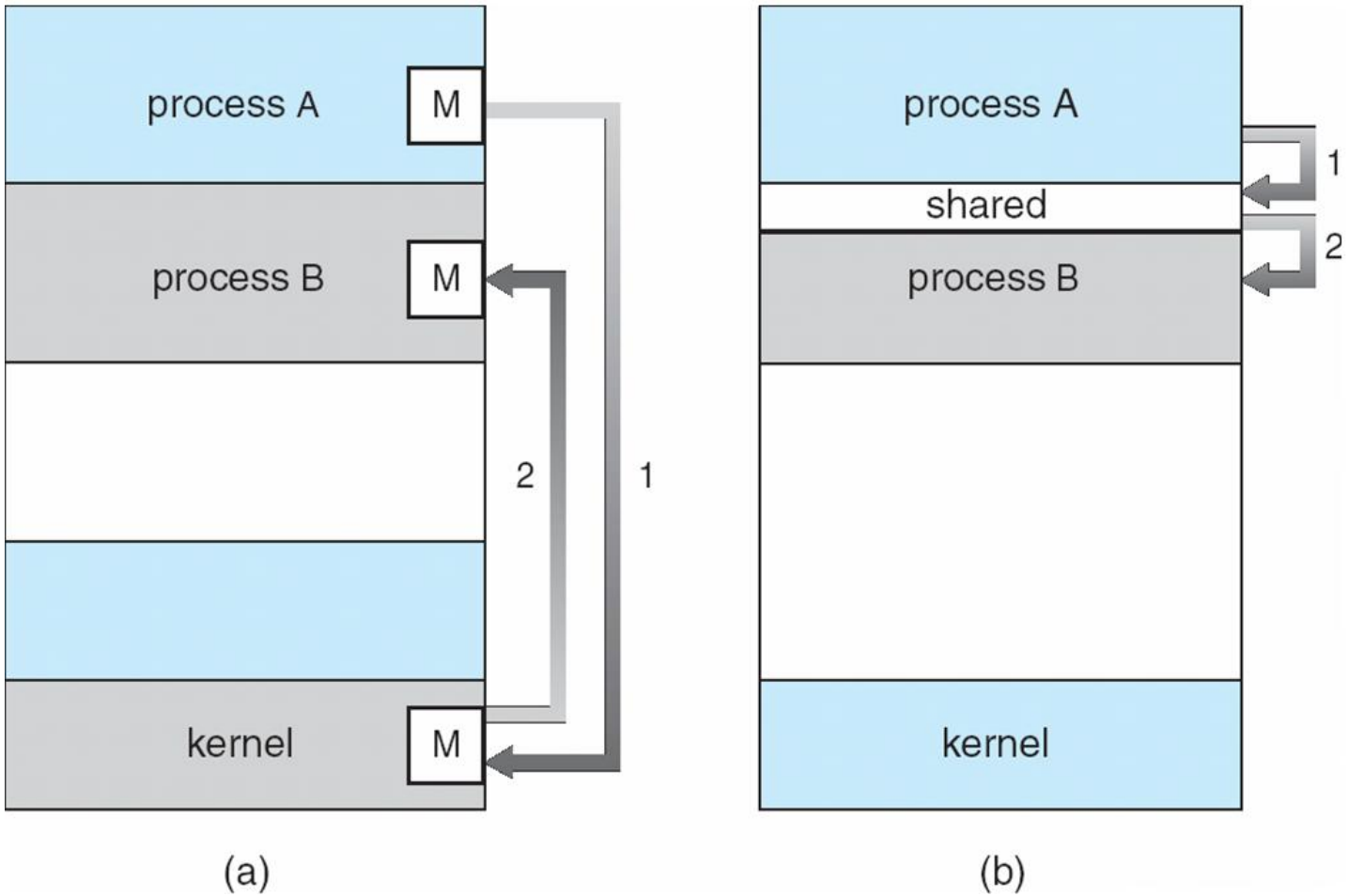
Interactions ...(contd.)

- Among the three kinds of interactions indicated by 1, 2 and 3 above:
  - 1 is at the system level: potential problems : deadlock and starvation.
  - 2 is at the process level : significant problem is in realizing **mutual exclusion**.
  - 3 is more a **synchronization** problem.

Interprocess Communication(Cont…)

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Dangers of process cooperation
  - Data corruption, deadlocks, increased complexity
  - Requires processes to synchronize their processing
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Interprocess Communication(Cont…)



Producer-Consumer Problem

```
int itemCount = 0;
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 contains a race condition that can lead to a deadlock.
- Consider the following scenario:
  - The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if block.
  - Just before calling sleep, the consumer is interrupted and the producer is resumed.
  - The producer creates an item, puts it into the buffer, and increases itemCount.
  - Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
  - 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.
  - The producer will loop until the buffer is full, after which it will also go to sleep.
  - Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

Race Conditions

- In most modern OS, processes that are working together often share some common storage
  - Shared memory
  - Shared file system
- A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

- A **Race Condition** is when the result of an operation depends on the ordering of when individual processes are run
  - Process scheduling is controlled by the OS and is non-deterministic

- Race conditions result in irregular errors that are very difficult to debug
  - Very difficult to test programs for race conditions

- Must recognize where race conditions can occur
  - Programs can run for months or years before a race condition is discovered

- For example, two processes P1 and P2 share the global variable X.
- Process P1 updates variable X to the value 2 and at some point in its execution P2 updates X to the value 3.
- These two tasks are in race to write variable X.
- In this, the process updates last determines the final value of X.
- If two processes P3 and P4 shares two global variables X=2 and Y=3.
  - While in execution, process P3 executes the assignment X= X+Y and process P4 executes Y=X+Y.
  - Both the process updates different variables.
  - The final values of two variables depend on the order in which the two processes executes these two assignment.

- Process Interaction:
  - Processes unaware of each other. This situation occurs in multiprogramming of multiple independent processes.
  - Processes indirectly aware of each other.
  - Processes directly aware of each other.
- Requirement for Mutual exclusion (ME):
  - ME should meet following requirement.
    - ME must be enforced only on process at a time is allowed into its critical section.
    - A process that halts in its noncritical section must do so without interfering with other processes.
    - It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock no starvation.
    - When no process is in its critical section, any process that ready must be permitted to enter without delay.
    - A process remain in its critical section for finite time only.

Critical section(CS) problem…

- Each process has a segment of code called critical section.
- is that part of the process code that affects the shared resource.
- Critical section is used to avoid race conditions on data items.
- In critical section, process maybe changing common variables, updating a table, writing a file and so on.
- At any time only one process can executes in its critical section.
- When one process is executing in its CS, no other process is to be allowed to execute in its critical section.
- The execution of critical sections by the process is mutually exclusive in time.
- Basic structure of Process :
- do

{

Enter CS

Critical Section.

Exit CS

Remaining section.

}

- Each process contains three section: entry section, exit section and remaining section.

Critical section(CS) problem…

- A solution to the CS problem must satisfy the following three requirements:
  - Mutual Exclusion
    - Suppose Pi process executing in its CS then no other processes are allowed to execute the same.
  - Progress
    - If no process is in CS and some process wish to enter their CS, then only those processes that are not executing in their remaining section can participate in the decision on which will enter its CS next.
    - This section can not be postponed indefinitely.
  - Bounded waiting
    - When process requests access to its CS, the decision that grants it access may not be delayed indefinitely.
    - A process may not be denied access because of starvation or deadlock.

Solution to the CS

- Two process solution
  - Peterson's Solution
  - Consider two processes P0 and P1
  - Algorithm - 1:
    - P0 and P1 share the common integer variable i.e. turn and initialized to 0 or 1.
    - If turn == 1,
      - Then P0 is allowed to execute in its CS.

Process P0 structure:

```
do
{
    while(turn!=i)
            critical section;
    turn = i;
            remaining section;
} while(1);
```

- For example if turn == 0 and P1is ready to enter into its CS, P1 can not do so, even though P0 may be in its remaining section.

Software Solutions: Algorithm 1

- Process 0

```
...
while turn != 0  do
            nothing;
// busy waiting
< Critical Section>
turn = 1;
...
```

- Process 1

```
...
while turn != 1  do
            nothing;
// busy waiting
< Critical Section>
turn = 0;
...
```

Problems : Strict alternation, Busy Waiting

Solution to the CS

- Algorithm-1 can not give sufficient information about the state of each process.
- It keeps only records of the process which is in its CS
- To solve this problem variable turn is replaced with flag and it is initialized with
  - Boolean flag[2]; and its values are false;
  - If flag[i] is true
    - Then Pi is ready to enter the CS.

Process P_i structure:

```
do
{   flag[i]=true;
    while(flag[j]);
        critical section;
    flag[i] = false;
        remaining section;
} while(1);
```

Process P_j structure:

```
do
{   flag[j]=true;
    while(flag[i]);
        critical section;
    flag[j] = false;
        remaining section;
} while(1);
```

- In this algo. Process Pi sets flag[i] to be true, then process Pi is ready to enter its CS.
- Process Pi also checks for process Pj.
- If process Pj were ready, then Pi would wait until flag[j]=false.
- So process Pi enter into its CS.

PROBLEM : Potential for deadlock, if one of the processes fail within CS.

Solution to the CS

- Algorithm – 3 gives the correct solution to the critical section problem.
- It satisfies all three requirements of the critical section.
- The processes shares two variables:
  - Boolean flag[2];
  - int turn;
  - Initialize flag[0]=flag[1]=false;

Process Pi structure:

```
do
{   flag[i]=true;
    turn=j;
    while(flag[j] && turn==j);
            critical section;
    flag[i] = false;
            remaining section;
} while(1);
```

- For process, to enter the CS first set flag[i] = true and then sets turn = j;
- If both processes try to enter at the same time, turn will set to both i and j at the same time.

Multiple Process Solutions…

- Bakery algorithm is used in multiple process solution.
- It solves the problem of CS for n processes.
- Each process requesting entry to critical section is given a numbered token such that the number on the token is larger than the maximum number issued earlier.
- This algorithm permits processes to enter the critical section in the order of their token numbers.
- If process Pi and Pj receive the same number and if i < j then Pi is served first.

Synchronization Hardware

- Test and modify the content of a word atomically.

```
boolean TestAndSet(boolean &target) {
   boolean rv = target;
   target = true;

   return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
   while (TestAndSet(lock)) ;
        critical section
   lock = false;
        remainder section
}
```

Synchronization Hardware

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to **false**):
  boolean lock;

- Process P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock, key);
        critical section
    lock = false;
        remainder section
}
```

Semaphores

- Think about a semaphore as a class
- Attributes: semaphore value,
- Functions:
  - init,
  - wait,
  - signal
- Support provided by OS
- Considered an OS resource, a limited number available: a limited number of instances (objects) of semaphore class is allowed.
- Can easily implement mutual exclusion among any number of processes.

Semaphores…

- Is used to solve the critical section problem.
- It is an integer value.
- Semaphore is a variable that has an integer value upon which the following three operations are defined.
  - Semaphore may be initialized to non negative value.
  - The wait operation
    - decrements the semaphore value.
    - If the value becomes negative, then the process executing the wait is blocked.
  - The signal operation
    - Increments the value of semaphore
    - If value is not positive then a process blocked by a wait operation is unblocked.

- Pseudo code:
  - Wait(s)

```
{ while(s<=0)
        s=s-1;
}
```

  - Signal(s)

```
{
  s=s+1;
}
```

- Semaphores are executed atomatically.

Critical Section of *n* Processes

- Shared data:
  Semaphore mutex; //initially *mutex* = 1
- Process *Pi:*

```
do {
    mutex.wait();
        critical section
    mutex.signal();
    remainder section
} while (1);
```

Semaphore Implementation

- Define a semaphore as a class:

```
class Semaphore
{
 int value; // semaphore value
 ProcessQueue L; // process queue

  //operations
     wait()
     signal()
}
```

- In addition, two simple utility operations:
  - **block()** suspends the process that invokes it.
  - **Wakeup()** resumes the execution of a blocked process **P.**

Semantics of wait and signal

- Semaphore operations now defined as

S.*wait()*:

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block(); // block a process
}
```

S.*signal()*:

```
S.value++;
if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(); // wake a process
}
```

Semaphores for CS

- Semaphore is initialized to 1.
- The first process that executes a *wait()* will be able to immediately enter the critical section (CS).
  - S.*wait()* makes S value zero.
- Now other processes wanting to enter the CS will each execute the wait() thus decrementing the value of S, and will get blocked on S.
  - If at any time value of S is negative, its absolute value gives the number of processes waiting blocked.
- When a process in CS departs, it executes S.*signal()* which increments the value of S, and will wake up any one of the processes blocked.
- The queue could be FIFO or priority queue.

Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.

- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement. ex: nachos
- Can implement a counting semaphore *using* a binary semaphore.

Classical Problems of Synchronization

- Bounded-Buffer Problem (Producer/Consumer Problem)
- Readers and Writers Problem
- Dining-Philosophers Problem

Producer/Consumer problem

- Producer

```
repeat
produce item v;
b[in] = v;
in = in + 1;
forever;
```

- Consumer

```
repeat
while (in <= out) nop;
w = b[out];
out = out + 1;
consume w;
forever;
```

Solution for P/C using Semaphores

- **Producer**

```
repeat
produce item v;
MUTEX.wait();
b[in] = v;
in = in + 1;
MUTEX.signal();
forever;
```

- **Consumer**

```
repeat
while (in <= out) nop;
MUTEX.wait();
w = b[out];
out = out + 1;
MUTEX.signal();
consume w;
forever;
```

- **What if Producer is slow or late?**
- **Ans: Consumer will busy-wait at the while statement.**

P/C: improved solution

- **Producer**

```
repeat
produce item v;
MUTEX.wait();
b[in] = v;
in = in + 1;
MUTEX.signal();
AVAIL.signal();
forever;
```

- **Consumer**

```
repeat
AVAIL.wait();
MUTEX.wait();
w = b[out];
out = out + 1;
MUTEX.signal();
consume w;
forever;
```

- **What will be the initial values of MUTEX and AVAIL?**
- **ANS: Initially MUTEX = 1, AVAIL = 0.**

P/C problem: Bounded buffer

- Producer

```
repeat
produce item v;
while((in+1)%n == out) NOP;
b[in] = v;
in = ( in + 1)% n;
forever;
```

- **How to enforce bufsize**?

- Consumer

```
repeat
while (in == out) NOP;
w = b[out];
out = (out + 1)%n;
consume w;
forever;
```

- **ANS: Using another counting semaphore.**

P/C: Bounded Buffer solution

- **Producer**

repeat
produce item v;
BUFSIZE.wait();
MUTEX.wait();
b[in] = v;
in = (in + 1)%n;
MUTEX.signal();
AVAIL.signal();
forever;

- **What is the initial value of BUFSIZE?**

- **Consumer**

repeat
AVAIL.wait();
MUTEX.wait();
w = b[out];
out = (out + 1)%n;
MUTEX.signal();
BUFSIZE.signal();
consume w;
forever;

- **ANS: size of the bounded buffer.**

Semaphores - comments

- Intuitively easy to use.
- wait() and signal() are to be implemented as atomic operations.
- Difficulties:
  - signal() and wait() may be exchanged inadvertently by the programmer. This may result in deadlock or violation of mutual exclusion.
  - signal() and wait() may be left out.
- Related wait() and signal() may be scattered all over the code among the processes.