Introduction

An important and fundamental feature in modern operating systems is concurrent execution of processes/threads. This feature is essential for the realization of multiprogramming, multiprocessing, distributed systems, and client-server model of computation.

Concurrency encompasses many design issues including communication and synchronization among processes, sharing of and contention for resources.

In this discussion we will look at the various design issues/problems and the wide variety of solutions available.

Topics for discussion

- The principles of concurrency
- Interactions among processes
- Mutual exclusion problem
- Mutual exclusion- solutions
  - Software approaches (Dekker’s and 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
- Mutual Exclusion, Synchronization and Classical InterProcess Communication (IPC) Problems

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);
  }
  ```

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.
  2. Co-operating processes: Results of one or more processes may be needed for another process.
  3. Co-operation by sharing: Example: Sharing of an IO buffer. Concept of critical section. (indirect)
  4. 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.
- We will study mutual exclusion and synchronization here, and defer deadlock, and starvation for a later time.

Race Condition
- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be synchronized.

Mutual exclusion problem
- Successful use of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion.
- **Critical section**: is that part of the process code that affects the shared resource.
- **Mutual exclusion**: in the use of a shared resource is provided by making its access mutually exclusive among the processes that share the resource.
- This is also known as the Critical Section (CS) problem.

Mutual exclusion
- Any facility that provides mutual exclusion should meet these requirements:
  1. No assumption regarding the relative speeds of the processes.
  2. A process is in its CS for a finite time only.
  3. Only one process allowed in the CS.
  4. Process requesting access to CS should not wait indefinitely.
  5. A process waiting to enter CS cannot be blocking a process in CS or any other processes.

Software Solutions: Algorithm 1
- Process 0
  - ...
  - while turn != 0 do nothing;
  - // busy waiting
  - < Critical Section>
  - turn = 1;
  - ...
- Problems: Strict alternation, Busy Waiting

Algorithm 2
- PROCESS 0
  - ...
  - flag[0] = TRUE;
  - while flag[1] do nothing;
  - <CRITICAL SECTION>
  - flag[0] = FALSE;

- PROCESS 1
  - ...
  - flag[1] = TRUE;
  - while flag[0] do nothing;
  - <CRITICAL SECTION>
  - flag[1] = FALSE;

PROBLEM: Potential for deadlock, if one of the processes fail within CS.
Algorithm 3
- Combined shared variables of algorithms 1 and 2.
- Process $P_i$
  - do {
    - flag $[i]$ := true;
    - turn := $j$;
    - while (flag $[j]$ and turn = $j$) ;
      - critical section
      - flag $[i]$ = false;
      - remainder section
    - } while (1);
- Meets all three requirements; solves the critical-section problem for two processes.

Synchronization Hardware
- Test and modify the content of a word atomically
  - boolean TestAndSet(boolean &target) {
    - boolean $rv$ = target;
    - $t$arget = 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;
  - boolean waiting[n];
- Process $P_i$
  - do {
    - key = true;
    - while (key == true)
      - Swap(lock,key);
        - critical section
      - lock = false;
      - remainder section
    - }

Semaphores
- Think about a semaphore ADT (class)
- Counting semaphore, binary semaphore
- 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

- Synchronization tool that does not require busy waiting.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

```c
wait(S):
  while $S \leq 0$ do no-op;
  $S$--;

signal(S):
  $S$++;
```

Critical Section of $n$ Processes

- Shared data:
  ```c
  semaphore mutex; // initially mutex = 1
  ```
- Process $P_i$:

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

Semaphore Implementation

- Define a semaphore as a record
  ```c
typedef struct {
    int value;
    struct process *L;
  } semaphore;
```
- Assume two simple operations:
  - `block` suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process $P$.

Implementation

- Semaphore operations now defined as
  ```c
  wait(S):
  S.value--;
  if (S.value < 0) {
    add this process to S.L;
    block;
  }
  
  signal(S):
  S.value++;
  if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
  }
```

Semaphore as a General Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$
- Use semaphore flag initialized to 0
- Code:

```c
P_i          P_j
  ::
A   wait(flag)
  ::
signal(flag)  B
```

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.
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);
```

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

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.

Can implement a counting semaphore $S$ as a binary semaphore.

Implementing $S$ as a Binary Semaphore

- **Data structures:**
  ```
  binary-semaphore S1, S2;
  int C:
  ```
- **Initialization:**
  ```
  S1 = 1
  S2 = 0
  C = initial value of semaphore S
  ```

Implementing $S$

- **wait operation**
  ```
  wait(S1);
  C--;
  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  signal(S1);
  ```

- **signal operation**
  ```
  wait(S1);
  C ++;
  if (C <= 0) {
    signal(S2);
    else
      signal(S1);
  }
  ```

Classical Problems of Synchronization

- **Bounded-Buffer 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?

  Answer: 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();
  - consume w;
  - 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?

  Answer: 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;
- **Consumer**
  - repeat
  - while (in == out) NOP;
  - w = b[out];
  - out = (out + 1)%n;
  - consume w;
  - forever;
- How to enforce bufsize?

  Answer: 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();
  - consume w;
  - forever;
- **Consumer**
  - repeat
  - AVAIL.wait();
  - MUTEX.wait();
  - w = b[out];
  - out = (out + 1)%n;
  - MUTEX.signal();
  - BUFSIZE.signal();
  - consume w;
  - forever;
- What is the initial value of BUFSIZE?

  Answer: 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.

Monitors

- This concept was formally defined by Hoare in 1974.
- Initially it was implemented as a programming language construct and more recently as library. The latter made the monitor facility available for general use with any PL.
- Monitor consists of procedures, initialization sequences, and local data. Local data is accessible only thru' monitor's procedures. Only one process can be executing in a monitor at a time. Other process that need the monitor wait suspended.
Monitors

```plaintext
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (…) {
        ...
    }
    procedure body P2 (…) {
        ...
    }
    procedure body Pn (…) {
        ...
    }
    { initialization code }
}
```

Monitors

- To allow a process to wait within the monitor, a condition variable must be declared, as:
  ```plaintext
  condition x, y;
  ```
- Condition variable can only be used with the operations `wait` and `signal`.
  - The operation `x.wait();` means that the process invoking this operation is suspended until another process invokes `x.signal();`
  - The `x.signal();` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.

Schematic View of a Monitor

Message passing

- Both synchronization and communication requirements are taken care of by this mechanism.
- More over, this mechanism yields to synchronization methods among distributed processes.
- Basic primitives are:
  ```plaintext
  send (destination, message);
  receive (source, message);
  ```

Monitor With Condition Variables

Issues in message passing

- Send and receive: could be blocking or non-blocking:
  - Blocking send: when a process sends a message it blocks until the message is received at the destination.
  - Non-blocking send: After sending a message the sender proceeds with its processing without waiting for it to reach the destination.
  - Blocking receive: When a process executes a receive it waits blocked until the receive is completed and the required message is received.
  - Non-blocking receive: The process executing the receive proceeds without waiting for the message(!).
- Blocking receive/non-blocking send is a common combination.
Reader/Writer problem
- Data is shared among a number of processes.
- Any number of reader processes could be accessing the shared data concurrently.
- But when a writer process wants to access, only that process must be accessing the shared data. No reader should be present.
- Solution 1: Readers have priority; If a reader is in CS any number of readers could enter irrespective of any writer waiting to enter CS.
- Solution 2: If a writer wants CS as soon as the CS is available writer enters it.

Reader/writer: Priority
Readers
- Writer:
  ForCS.wait();
  CS;
  ForCS.signal();

Readers
- Reader:
  CS;
  ForCS.signal();

Reader/writer: Priority

Dining Philosophers Example
monitor dp
{
  enum {thinking, hungry, eating};
  state[5];
  condition self[5];
  void pickup(int i) // following slides
  void putdown(int i) // following slides
  void test(int i) // following slides
  void init() {
    for (int i = 0; i < 5; i++)
      state[i] = thinking;}
}

Dining Philosophers
void pickup(int i) {
  state[i] = hungry;
  test[i];
  if (state[i] != eating)
    self[i].wait();
}

void putdown(int i) {
  state[i] = thinking;
  // test left and right neighbors
  test[(i+4) % 5];
  test[(i+1) % 5];
}

Dining Philosophers
void test(int i) {
  if ( (state[(i + 4) % 5] != eating) &&
       (state[i] == hungry) &&
       (state[(i + 1) % 5] != eating))
    {
      state[i] = eating;
      self[i].signal();
    }
}

Summary
- We looked at various ways/levels of realizing synchronization among concurrent processes.
- Synchronization at the kernel level is usually solved using hardware mechanisms such as interrupt priority levels, basic hardware lock, using non-preemptive kernel (older BSDs), using special signals.