Concurrent is the simultaneous execution of threads

- The system must support concurrent execution of threads
- Scheduling: Deals with execution of “unrelated” threads
- Concurrency: Deals with execution of “related” threads

Why is it necessary?

- Cooperation: One thread may need to wait for the result of some operation done by another thread
  e.g. “Calculate Average” must wait until all “data reads” are completed
- Competition: Several threads may compete for exclusive use of resources
  e.g. two threads trying to increment the value in a memory location
Concurrency

Thr A
load mem, reg
inc reg
store reg, mem

Thr B
load mem, reg
inc reg
store reg, mem

Critical Section
All three instructions of a thread must be executed before other thread
Mutual Exclusion

- If one thread is going to use a shared resource (critical resource)
  - a file
  - a variable
  - printer
  - register, etc
  - the other thread must be excluded from using the same resource

- **Critical resource**: A resource for which sharing by the threads must be controlled by the system

- **Critical section** of a program: A part of a program where access to a critical resource occurs
Ex: “echo” function

```c
void echo (char in)
{
    char out;
    read (stdin, in);
    out = in;
    write (stdout, out);
}
```

See this routine as a shared module, being called by any program that needs it. Only one copy of the program is kept in memory.

<table>
<thead>
<tr>
<th>Thr 1</th>
<th>Thr 2</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(stdin,in);</td>
<td>read(stdin,in);</td>
<td>in</td>
</tr>
<tr>
<td>BLOCKED</td>
<td>out = in;</td>
<td></td>
</tr>
<tr>
<td>out=in;</td>
<td>write(stdout,out);</td>
<td>y</td>
</tr>
<tr>
<td>write(stdout,out);</td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>

Thr 1

Thr 2

Variables

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>
Concurrency

- Erroneous execution: Thread 1 prints ‘y’, whereas it was supposed to print ‘x’...

- **Race Condition**: The outcome of the execution depends on the particular order in which the access takes place.

- Function echo, and hence the variables `in` and `out` are shared resources.

- **Solution: Mutual Exclusion**

  ```
  Thr 1                Thr 2
  (call echo)          (call echo)
  (enter echo)         (wait)
  read...              (enter echo)
  BLOCKED              (enter echo)
  ```
Concurrency

Two problems related to concurrency control

- Deadlock:

  Thr 1  Thr 2
  
  request R1  request R2
  allocate R1  allocate R2
  request R2  request R1
  (wait) (wait)
  
  Both threads wait forever.
Concurrency

- Starvation

<table>
<thead>
<tr>
<th>Thr 1</th>
<th>Thr 2</th>
<th>Thr 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>request R</td>
<td>request R</td>
<td>request R</td>
</tr>
<tr>
<td>allocate R</td>
<td></td>
<td>(wait)</td>
</tr>
<tr>
<td>release R</td>
<td>release R</td>
<td>release R</td>
</tr>
<tr>
<td>request R</td>
<td>allocate R</td>
<td>request R</td>
</tr>
<tr>
<td>(wait)</td>
<td>request R</td>
<td>(wait)</td>
</tr>
<tr>
<td>allocate R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>release R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Thr 3 waits indefinitely.
- The reason may be that Thr 1 and Thr 2 have higher priorities than Thr 3.
**Concurrency**

- Mutual Exclusion Mechanism

<table>
<thead>
<tr>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>enterCritical (R)</td>
</tr>
<tr>
<td>&lt;critical section&gt;</td>
</tr>
<tr>
<td>exitCritical (R)</td>
</tr>
</tbody>
</table>

Only one thread can execute this part at a time.
Concurrency Requirements

- among all threads that have CSs for the same resource
  - Only one thread at a time is allowed into its CS,
- It must not be possible for a thread requiring access to a CS to be delayed indefinitely
  - no deadlock
  - no starvation
- When no thread is in a CS, any thread requesting entry to the CS must be granted permission without delay
- No assumptions are made about the relative thread speeds or number of processors.
- A thread remains inside its CS for a finite time only.
Concurrency

- There are some methods for addressing concurrency problems
  - semaphores
  - message passing
  - monitors, etc.
- It is the responsibility of the OS (not the programmer) to enforce mutual exclusion.
Concurrency

- Two basic concepts:
  - Interrupt Disabling
    - The only way of providing threads to interleave is by the use of interrupts
    - If it is guaranteed that no interrupt occurs while a thread is in the CS, then no other thread can enter the same CS

- Simplest solution
  But it is not desirable to give a thread the power of controlling interrupts
  - In a multiprocessor environment, it does not work
  - This approach is often used by some OS threads (because they are short)
Semaphores

- Semaphore is a non-negative integer variable
  - Its value is initialized
  - Its value can be changed by two “atomic” instructions
    WAIT:
    Wait until the value is greater than 0. Then the value is decremented by 1.
    (The thread when waiting is moved to a wait queue)
    SIGNAL:
    The value is incremented by 1
    (If there is a thread waiting for that semaphore, it’s woken up and continues)
**Semaphores**

- **Ex: Initially, s=1**

<table>
<thead>
<tr>
<th>Thread 1..n</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>s</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT (s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;critical section&gt;</td>
<td>&lt;CS&gt;</td>
<td></td>
<td></td>
<td>0</td>
<td>T2</td>
</tr>
<tr>
<td>SIGNAL (s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Due to mutual exclusion only one thread is in CS at a time.
Ex: Initially, s=0

Aim: To make Thread 1 wait for Thread 2 at point A, and make sure that the code after A is executed only after Thread 2 finishes execution up to point B.

Scenario - 1: First Thread 1 reaches point A

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>s</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>wait</td>
<td>0</td>
<td>0</td>
<td>T1</td>
</tr>
<tr>
<td>blocked</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal</td>
<td>1</td>
<td>0</td>
<td>T1</td>
</tr>
<tr>
<td>exec</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Semaphores

- Ex: Initially, s=0

Aim: To make Thread 1 wait for Thread 2 at point A, and make sure that the code after A is executed only after Thread 2 finishes execution up to point B.

Scenario - 2: First Thread 2 reaches point B

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>s</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>signal</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>exec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Semaphores

- **Ex: Initially, s=n**
  - n: number of resources available

  There are 3 printers => Resources

  Use different semaphores for each printer, e.g. $s_1=1$, $s_2=1$, $s_3=1$ initially - all are free

  or

  Use a single semaphore for all printers, e.g. $s=3$.

<table>
<thead>
<tr>
<th>Thread 1..n</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>s</th>
<th>queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT (s)</td>
<td>wait</td>
<td>&lt;CS&gt;</td>
<td>wait</td>
<td>wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;critical section&gt;</td>
<td>&lt;CS&gt;</td>
<td>wait</td>
<td>&lt;CS&gt;</td>
<td>wait</td>
<td>2</td>
<td>T4</td>
</tr>
<tr>
<td>SIGNAL (s)</td>
<td>&lt;CS&gt;</td>
<td>wait</td>
<td>&lt;CS&gt;</td>
<td>wait</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Uses printer</td>
<td>signal</td>
<td>&lt;CS&gt;</td>
<td></td>
<td></td>
<td>1</td>
<td>T4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Synchronization

- Producer-Consumer Problem
  - A producer thread puts data into a buffer
  - A consumer thread takes data from the buffer
  - A producer cannot put data into a full buffer
  - A consumer cannot take data from an empty buffer
  - Only one thread can access buffer at a time → mutual exclusion

- One producer - one consumer

- Semaphores:
  - SlotFree: number of empty slots, initially n
  - ItemAvailable: number of items available in buffer, initially 0
Producer

while (1) {
    newItem = produceItem();
    WAIT (SlotFree);
    buffer[NextIn] = newItem;
    NextIn++;
    SIGNAL (ItemAvailable);
}

NextIn: Next slot in the buffer to be used by producer, initially 1

Consumer

while (1) {
    WAIT (ItemAvailable);
    nextItem=buffer[NextOut];
    NextOut++;
    SIGNAL (SlotFree);
    consumeItem (nextItem);
}

NextOut: Next slot in buffer to be used by consumer, initially 1
### Synchronization

**Ex:**

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
<th>SF</th>
<th>IA</th>
<th>SF queue</th>
<th>IA queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait (IA)</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait (SF)</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal (IA)</td>
<td></td>
<td>4</td>
<td>1</td>
<td></td>
<td>Consumer</td>
</tr>
<tr>
<td>wait (SF)</td>
<td>consume</td>
<td>3</td>
<td>0</td>
<td></td>
<td>Consumer</td>
</tr>
<tr>
<td>signal (IA)</td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wait (SF)</td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal (SF)</td>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signal (IA)</td>
<td></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization

- Multiple Producers, Multiple Consumers

Producers

```c
Semaphore SlotFree = BUFFERSIZE, ItemAvailable = 0, Mutex = 1;
while (1) {
    newItem = produceItem();
    WAIT (SlotFree);
    WAIT (Mutex);
    buffer[NextIn] = newItem;
    NextIn++;
    SIGNAL (Mutex);
    SIGNAL (ItemAvailable);
}
```

Consumers

```c
while (1) {
    WAIT (ItemAvailable);
    WAIT (Mutex);
    nextItem=buffer[NextOut];
    NextOut++;
    SIGNAL (Mutex);
    SIGNAL (SlotFree);
    consumeItem (nextItem);
}
```

Mutex: Guarantees only one thread can access buffer at a time
Synchronization

- Only one thread can access the buffer at a time

- Order of WAIT signals is crucial
  - E.g. if \texttt{WAIT(Mutex)} comes before \texttt{WAIT(SlotFree)} in producer algorithm, the system would go into deadlock when the buffer is full.
Implementation of Semaphores

- They must be atomic
- There must be a queue mechanism, for putting the waiting thread into a queue, and waking it up later
  - Scheduler must be involved

- Define a semaphore as a record

  ```c
  typedef struct {
      int value;
      struct process *PList;
  } semaphore;
  ```

- Assume two simple operations:
  - `block` suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process P.
Implementation of Semaphores

- Semaphore operations now defined as

  \[ \text{\textit{wait}}(S) : \]
  \[
  S\.value--; \\
  \text{if} (S\.value < 0) \{
    \text{add this process to } S\.PList; \\
    \text{block}; \\
  \}
  \]

  \[ \text{\textit{signal}}(S) : \]
  \[
  S\.value++; \\
  \text{if} (S\.value <= 0) \{
    \text{remove a process } P \text{ from } S\.PList; \\
    \text{wakeup}(P); \\
  \}
  \]

  REM: Note that, \( S \) can be negative with this implementation, where the negative value indicates the number of processes waiting for \( S \)…
Semaphores

- Semaphore mechanism is handled by OS

- Writing correct semaphore algorithms is a complex task

- All threads using the same semaphore are assumed to have the same priority.
  - Implementation does not take priority into account.