Chapter 9

High-level Synchronization
Introduction to Concurrency

- **Concurrency**
  - Execute two or more pieces of code "at the same time"

- **Why?**
  - No choice:
    - Geographically distributed data
    - Interoperability of different machines
    - A piece of code must "serve" many other client processes
    - To achieve reliability
  - By choice:
    - To achieve speedup
    - Sometimes makes programming easier (e.g., UNIX pipes)
### Possibilities for Concurrency

<table>
<thead>
<tr>
<th>Architecture:</th>
<th>Program Style:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniprocessor with:</td>
<td>Multiprogramming,</td>
</tr>
<tr>
<td>− I/O channel</td>
<td>multiple process system</td>
</tr>
<tr>
<td>− I/O processor</td>
<td>programs</td>
</tr>
<tr>
<td>− DMA</td>
<td></td>
</tr>
<tr>
<td>Multiprocessor</td>
<td>Parallel programming</td>
</tr>
<tr>
<td>Network of processors</td>
<td>Distributed Programs</td>
</tr>
</tbody>
</table>
Examples of Concurrency in Uniprocessors

Example 1: Unix pipes

Motivations:
- fast to write code
- fast to execute

Example 2: Buffering

Motivation:
- required when two *asynchronous* processes must communicate

Example 3: Client/Server model

Motivation:
- geographically distributed computing
Operating System issues to Support Concurrency

- **Synchronization**
  - What primitives should OS provide?

- **Communication**
  - What primitives should the OS provide to the interface communication protocol?

- **Hardware Support**
  - Needed to implement OS primitives
Operating System issues to Support Concurrency…

- **Remote execution**
  - What primitives should OS provide?
    - Remote Procedure Call (RPC)
    - Remote Command Shell

- **Sharing address space**
  - Makes programming easier

- **Light–weight threads**
  - Can a process creation be as cheap as a procedure call?
Definitions

- **Concurrent** process execution can be:
  - interleaved, or
  - physically simultaneous

- **Interleaved**
  - *Multi–programming* on uniprocessor

- **Physically simultaneous**
  - Uni– or multi–programming on *multiprocessor*
Definitions…

- **Process, thread, or task**
  - Scheduleable unit of computation

- **Granularity**
  - Process "size" or computation to
  - Communication ratio
    - Too small: excessive overhead
    - Too large: less concurrency
Consider writing a program as a set of tasks.

**Precedence graph:**

specifies execution ordering among tasks

\[
\begin{align*}
S1: & \quad A := X + Y \\
S2: & \quad B := Z + 1 \\
S3: & \quad C := A - B \\
S4: & \quad W := C + 1
\end{align*}
\]

Parallelizing compilers for computers with vector processors build dependency graphs.
What does the following graph represent?

\[ \text{S2} \text{ must be performed before S3 begins} \]

AND

\[ \text{S3 must be performed before S2 begins} \]

Precedence Graphs must be ACYCLIC
Concurrency Conditions

Let $S_i$ denote a statement.

**Read set of $S_i$:**

$$R(S_i) = \{ a_1, a_2, \ldots, a_n \}$$

Set of all variables referenced in $S_i$

**Write set of $S_i$:**

$$W(S_i) = \{ b_1, b_2, \ldots, b_m \},$$

Set of all variables changed by $S_i$
Concurrency Conditions…

\[ C := A - B \]
\[ R( C := A - B ) = \{ A, B \} \]
\[ W( C := A - B ) = \{ C \} \]

`scanf("%d", &A)`
\[ R( \text{scanf}("%d", &A) ) = \{ \} \]
\[ W( \text{scanf}("%d", &A) ) = \{ A \} \]
The following conditions must hold for two statements $S_1$ and $S_2$ to execute concurrently with valid results:

1) $R(S_1) \cap W(S_2) = \{\}$
2) $W(S_1) \cap R(S_2) = \{\}$
3) $W(S_1) \cap W(S_2) = \{\}$

These are called the Bernstein Conditions.
Parallel Language Constructs (Review)

**FORK and JOIN**

**FORK L**
Starts parallel execution at the statement labelled L and at the statement following the FORK.

**JOIN Count**
Recombines ‘Count’ concurrent computations.

Count := Count - 1;
If (Count > 0)
Then
   Terminate computation
else continue

Join is an *atomic* operation.
Structured Parallel Constructs

**PARBEGIN**

Sequential execution splits off into several concurrent sequences

**PAREND**

Parallel computations merge

```
PARBEGIN
  Statement 1;
  Statement 2;
  ...
  Statement N;
PAREND;
```

```
PARBEGIN
  Q := C mod 25;
  Begin
  N := N - 1;
  T := N / 5;
  End;
  Proc1 (X, Y);
PAREND;
```
Begin

PARBEGIN

A := X + Y;
B := Z + 1;
PAREND;
C := A - B;
W := C + 1;
End;

Begin
S1;
PARBEGIN
S3;
BEGIN
S2;
S4;
PAREND;
End;
PAREND;
S7;
End;

CS 3204
Monitors

- P & V are primitive operations
- Semaphore solutions are difficult to accurately express for complex synchronization problems
- Need a High-Level solution: Monitors
- A Monitor is a collection of procedures and shared data
- Mutual Exclusion is enforced at the monitor boundary by the monitor itself
- Data may be global to all procedures in the monitor or local to a particular procedure
- No access of data is allowed from outside the monitor
Condition Variables

- Within the monitor, Condition Variables are declared
- A queue is associated with each condition variable
- Only two operations are allowed on a condition variable:
  
<table>
<thead>
<tr>
<th>X.wait</th>
<th>The procedure performing the wait is put on the queue associated with x</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.signal</td>
<td>If queue is non-empty: resume <em>some</em> process at the point it was made to wait</td>
</tr>
</tbody>
</table>

- Note: V operations on a semaphore are "remembered," but if there are no waiting processes, the signal has no effect
- OS scheduler decides which of several waiting monitor calls to unlock upon signal
Monitor...

- Queue to enter monitor via calls to procedures
- Queues within the monitors via condition variables
- ADTs and condition variables only accessible via monitor procedure calls
Monitors contain procedures that control access to a `< CS >`, but not the `< CS >` code itself.

```plaintext
Program

Begin
    ...
    Request;
    `< CS >`
    Release;
    ...
End;

Monitor `<name>`
condition i;

Request
    ...
    ...
    ...

Release
    ...
    ...
    ...

end monitor
```
Monitor NCS {
    OK: condition
    Busy: boolean <-- FALSE

    Request() {
        if (Busy) OK.wait;
        Busy = TRUE;
    }

    Release() {
        Busy = FALSE;
        OK.signal;
    }
}

Procedure P {
    NCS.Request();
    <CS>;
    NCS.Release();
}

main() {
    parbegin P;P;P;P; parend }
Shared Variable Monitor

```java
monitor sharedBalance {
    int balance;

    public:
    Procedure credit(int amount)
    {
        balance = balance + amount;
    }

    Procedure debit(int amount)
    {
        balance = balance - amount;
    }
}
```
Reader & Writer Schema

reader() {
    while (true) {
        ...
        startRead();
        <read the resource>
        finishRead();
        ...
    }
}

writer() {
    while (true) {
        ...
        startWrite();
        <write resource>
        finishWrite();
        ...
    }
}

fork(reader, 0);
fork(reader, 0);
fork(writer, 0);
Reader & Writers Problem: An attempted solution

```java
monitor readerWriter_1{
  int numberOfReaders = 0;
  int numberOfWriters = 0;
  boolean busy = false;

  public:
    startRead()
    {
      while(numberOfReaders != 0);
      numberOfReaders = numberOfReaders+1;
    }

    finishRead()
    {
      numberOfReaders = numberOfReaders−1;
    }

    startWrite()
    {
      numberOfWriters = numberOfWriters+1;
      while(busy || numberOfReaders > 0);
      busy = true;
    }

    finishWrite()
    {
      numberOfWriters = numberOfWriters−1;
      busy = false;
    }
}
```

This solution does not work.
Reader & Writers Problem: The solution

```
monitor reader_writer_2{
    int numberOfReaders = 0;
    boolean busy = false;
    condition okToRead, okToWrite;

public:
    startRead()
    {
        if (busy || okToWrite.queue) okToRead.wait;
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal;
    }

    finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if (numberOfReaders == 0) okToWrite.signal;
    }

    startWrite()
    {
        if (busy || numberOfReaders != 0) okToWrite.wait;
        busy = true;
    }

    finishWrite() {
        busy = false;
        if (okToWrite.queue) okToWrite.signal;
        else okToRead.signal;
    }
}
```
Dining Philosophers’ Problem:
The solution

```java
enum status {eating, hungry, thinking};

monitor diningPhilosophers{
    status state[N]; condition self[N]; int j;
    // This procedure can only be called from within the monitor
test(int i) {
    if((state[i=1 MOD N] != eating) && (state[i] == hungry) && (state[i+1 MOD N] != eating)) {
        state[i] = eating;
        self[i].signal;
    }
}

public:
pickUpForks() {
    state[i] = hungry;
    test(i);
    if(state[i] != eating) self[i].wait;
}

putDownForks() {
    state[i] = thinking;
    test(i-1 MOD N); test(i+1 MOD N);
}

diningPhilosophers() { // Monitor initialization code
    for(int i=0; i<N; i++) state[i] = thinking;
}
```
Simple Resource Allocation with a monitor

monitor resourceAllocator;
var resourceInUse: boolean;
    resourceIsFree: condition;
procedure getResource;
begin
    if(resourceInUse) wait(resourceIsFree);
    resourceInUse := true;
end;
procedure returnResource;
beg
    resourceInUse := false;
    signal(resourceIsFree);
end;
beg
    resourceInUse := false;
end.

Can use as a Semaphore
Monitor implementation of a ring buffer

monitor ringBufferMonitor;

var ringBuffer: array[0..slots-1] of stuff;
    slotInUse: 0..slots;
    nextSlotToFill: 0..slots-1;
    nextSlotToEmpty: 0..slots-1;
    ringBufferHasData, ringBufferHasSpace: condition;

procedure fillASlot(slotData: stuff);
begin
    if (slotInUse = slots) then wait(ringBufferHasSpace);
    ringBuffer[nextSlotToFill] := slotData;
    slotInUse := slotInUse + 1;
    nextSlotToFill := (nextSlotToFill+1) MOD slots;
    signal(ringBufferHasData);
end;
Monitor implementation of a ring buffer...

```plaintext
procedure emptyASlot(var slotData: stuff);
begin
  if(slotInUse = 0) then wait(ringBufferHasData);
  slotData := ringBuffer[nextSlotToEmpty];
  slotInUse := slotInUse - 1;
  nextSlotToEmpty := (nextSlotToEmpty-1) MOD slots;
  signal(ringBufferSpace);
end;
begin
  slotInUse := 0;
  nextSlotToFill := 0;
  nextSlotToEmpty := 0;
end.
```