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>Multiple programming,</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**
  - Motivation:
    - 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

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

Granularity
- Process "size" or computation to
- Communication ratio
  - Too small: excessive overhead
  - Too large: less concurrency

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

Granularity
- Process "size" or computation to
- Communication ratio
  - Too small: excessive overhead
  - Too large: less concurrency

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

**Granularity**
- Process "size" or computation to
- Communication ratio
  - Too small: excessive overhead
  - Too large: less concurrency

Precedence Graph

Consider writing a program as a set of tasks.

**Precedence graph**
- specifies execution ordering among tasks

- S1
  - A := X + Y
- S2
  - B := Z + 1
- S3
  - C := A - B
- S4
  - W := C + 1

Parallelizing compilers for computers with vector processors build dependency graphs.

Cyclic Precedence Graph

**Precedence Graphs must be ACYCLIC**

What does the following graph represent?
- S2 must be performed before S3 begins
- AND
- S3 must be performed before S2 begins

Concurrency Conditions

Let S_i denote a statement.

**Read set of S_i**:
- R (S_i) = { a_1, a_2, ..., a_n }
  - Set of all variables referenced in S_i

**Write set of S_i**:
- W (S_i) = { b_1, b_2, ..., 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 (scanf("%d", &A)) = {}
- W (scanf("%d", &A)) = { A }
Bernstein's Conditions

The following conditions must hold for two statements $S_1$ and $S_2$ to execute concurrently with valid results:

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

These are called the Bernstein Conditions.

Parallel Language Constructs (Review)

**FORK and JOIN**

**FORK**

Starts parallel execution at the statement labelled $L$ and at the statement following the FORK

**JOIN**

Recombines ‘Count’ concurrent computations

```
Count := Count - 1;
If (Count > 0)
    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 / Parend Examples

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

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

Synchronization with Monitors

**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:
  - X.wait
  - X.signal

  The procedure performing the wait is put on the queue associated with
  X. If queue is non-empty: resume some process at the point it was made to wait

  • 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

N-Process Critical Section: Monitor Solution

Monitor NCS

Monitor NCS

OK: condition
Busy: boolean <- FALSE

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

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

main()
{
  parbegin P;P;P;P; parend
}

Shared Variable Monitor

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();
    ...}
}

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

```java
monitor readerWrite_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 % N] != eating) && (state[i] == hungry) && (state[(i + 1) % 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) % N); test((i+1) % N);
    }
    diningPhilosophers() { // Monitor initialization code
        for (int i=0; i<N; i++) state[i] = thinking;
    }
}
```

Simple Resource Allocation
with a monitor

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

Monitor implementation of a ring buffer

```java
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...

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