Chapter 9

High-level Synchronization

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

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 $S_i$

Concurrently Conditions...

$C \leftarrow A \cdot B$

$R(C \leftarrow A \cdot B) = \{ A, B \}$

$W(C \leftarrow A \cdot B) = \{ C \}$

$\text{cin} \gg A$

$R(\text{cin} \gg A) = \{ \}$

$W(\text{cin} \gg A) = \{ A \}$
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.

Structured Parallel Constructs

Sequential execution splits off into several concurrent sequences

Parallel computations merge

PARBEGIN
Statement 1;
Statement 2;
Statement N;
PAREND;

PARBEGIN
\begin{align*}
Q & := C \mod 25; \\
& \text{Begin} \\
N & := N - 1; \\
T & := N / 5; \\
& \text{End}; \\
& \text{Proc1}(X, Y); \\
\end{align*}
PAREND;

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

Parbegin / Parend Examples

\begin{verbatim}
Begin
PARBEGIN
A := X + Y;
B := Z + 1;
PARENT;
C := A + B;
W := C + 1;
End;
\end{verbatim}

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: The procedure performing the wait is put on the queue associated with $x$
  - X.signal: 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
Monitors...

Monitors contain procedures that control access to a `<CS>`, but not the `<CS>` code itself.

Program

```
begin
request;
<CS>
release;
end;
```

Monitor <name>

```
condition:
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

```
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: The solution

```
monitor reader_writer_1{
    int numberReaders = 0;
    int numberWriters = 0;
    condition okToRead, okToWrite;
    public:
        startRead()
            if [Busy] or [okToWrite.queue] okToRead.wait;
            numberReaders = numberReaders + 1;
            okToRead.signal;
        finishRead()
            numberReaders = numberReaders - 1;
            if [numberReaders = 0] okToWrite.signal;
        startWrite()
            if [Busy] and [numberReaders = 0] okToWrite.wait;
            Busy = true;
        finishWrite()
            Busy = false;
            if [okToWrite.queue] okToWrite.signal;
            else okToRead.signal;
    }
```

Dining Philosophers’ Problem

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

Example: Synchronizing Traffic

```java
monitor tunnel {
    int northbound = 0, southbound = 0;
    int nbSignal = RED, sbSignal = GREEN;
    condition busy;
    public:
    nbArrival() {
        if(southbound > 0) busy.wait();
        northbound++;
        nbSignal = GREEN; sbSignal = RED;
    }
    sbArrival() {
        if(northbound > 0) busy.wait();
        southbound++;
        nbSignal = RED; sbSignal = GREEN;
    }
}
```

Monitor implementation of a ring buffer

```java
monitor ringBufferMonitor;
var ringBuffer: array[0...slots-1] of stuff;
var slotInUse: 0..slots;
var nextSlotToFill: 0..slots-1;
var nextSlotToEmpty: 0..slots-1;
var ringBufferHasData, ringBufferHasSpace: condition;
procedure fillASlot(slotData: stuff) {
    if(slotInUse = slots) then wait(ringBufferHasSpace);
    ringBuffer[nextSlotToFill] := slotData;
    slotInUse := slotInUse + 1;
    nextSlotToFill := (nextSlotToFill + 1) MOD slots;
    signal(ringBufferHasData);
}
```

Example: Synchronizing Traffic

```java
monitor ringBufferMonitor;
var ringBuffer: array[0...slots-1] of stuff;
var slotInUse: 0..slots;
var nextSlotToFill: 0..slots-1;
var nextSlotToEmpty: 0..slots-1;
var ringBufferHasData, ringBufferHasSpace: condition;
procedure fillASlot(slotData: stuff) {
    if(slotInUse = slots) then wait(ringBufferHasSpace);
    ringBuffer[nextSlotToFill] := slotData;
    slotInUse := slotInUse + 1;
    nextSlotToFill := (nextSlotToFill + 1) MOD slots;
    signal(ringBufferHasData);
}
```

Example: Synchronizing Traffic

```java
departure(Direction exit) {
    if(exit = NORTH) {
        northbound--;
        if(northbound == 0) while(busy.queue())
            busy.signal();
    } else if(exit = SOUTH) {
        southbound--;
        if(southbound == 0) while(busy.queue())
            busy.signal();
    }
}
```