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>

CS 3204
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

CS 3204
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 $S_i$
Concurrent Conditions...

C := A - B

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

cin >> A

R (cin >> A) = {}
W (cin >> A) = {A}
Bernstein's Conditions

The following conditions must hold for two statements S1 and S2 to execute concurrently with valid results:

1) \( R(S1) \cap W(S2) = \{\} \)
2) \( W(S1) \cap R(S2) = \{\} \)
3) \( W(S1) \cap W(S2) = \{\} \)

These are called the Bernstein Conditions.
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;

CS 3204
Begin

PARBEGIN

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

Begin

S1;
PARBEGIN
S3;
BEGIN
S2;
S4;
PARBEGIN
S5;
S6;
PAREND;
End;
PAREND;
S7;
End;
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
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

Diagram:

- ADT's condition variables
  - Proc1
  - Proc2
  - Proc3

- queues
  - queue
Monitors...  

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

```
Monitor <name>
  condition i;

  Request
  __________
  __________
  __________

  Release
  __________
  __________
  __________

end monitor
```

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

CS 3204
Shared Variable Monitor

```cpp
monitor sharedBalance {

  int balance;

public:

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

  Procedure debit(int amount)
  {
    balance = balance - amount;
  }

}
```
Reader & Writer Schema

```c
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

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

d 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;
    }
}
```

CS 3204
while(TRUE) {
    think();
    eat();
}
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;
    }
}
Example: Synchronizing Traffic

- One-way tunnel
- Can only use tunnel if no oncoming traffic
- OK to use tunnel if traffic is already flowing the right way
Example: Synchronizing Traffic

monitor tunnel {
  int northbound = 0, southbound = 0;
  trafficSignal 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;
    };
}
Example: Synchronizing Traffic

depart(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();
    }
}
}
Monitor implementation of a ring buffer

```pascal
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...

```pascal
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.
```