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, multiple process system</td>
</tr>
<tr>
<td>- I/O channel</td>
<td>programs</td>
</tr>
<tr>
<td>- I/O processor</td>
<td></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 $S_i$

Concurrency Conditions...

$C = A - B$

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

$\text{cin} >> A$

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

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

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

These are called the Bernstein Conditions.

Structured Parallel Constructs

- \textsc{parbegin} / \textsc{parend}: Sequential execution splits off into several concurrent sequences
- Parallel computations merge

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

Parbegin / Parend Examples

```
Begin
\textsc{parbegin}
A = X + Y; \\
B = Z + 1; \\
\textsc{parend}; \\
C = A - B; \\
W = C + 1; \\
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

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:
  - \textbf{X.wait}: The procedure performing the wait is put on the queue associated with \( x \)
  - \textbf{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

```
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
```

```
monitor sharedBalance {
int balance;
public:
Procedure credit(int amount)
{
  balance = balance + amount;
}
Procedure debit(int amount)
{
  balance = balance - amount;
}
```

```
reader() {
while (true)
{
...
startRead();
<read the resource>
finishRead();
...
}
}
writer() {
while (true)
{
...
startWrite();
<write resource>
finishWrite();
...
}
}
```

```
fork(reader, 0);
fork(reader, 0);
fork(writer, 0);
```

```
Dining Philosophers’ Problem
while (TRUE) {
  think();
  eat();
}
```
**Dining Philosophers’ Problem: The solution**

```c
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) &
㎞

**Example: Synchronizing Traffic**

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

**Monitor implementation of a ring buffer**

```c
monitor ringBufferMonitor;
var ringBuffer: array[0..slots-1] of stuff;
slotInUse: 0..slots;
nextSlotToFill: 0..slots-1;
nextSlotToEmpty: 0..slots-1;
ringBufferHasData, ringBufferHasSpace:
procedure fillASlot(slotData: stuff);
begin
    if(slotInUse = slots)
zm

```c
procedure emptyASlot(var slotData: stuff);
begin
    if(slotInUse = 0)
zm

**Monitor implementation of a ring buffer...**

```c
begin
    slotInUse = 0;
    nextSlotToFill = 0;
    nextSlotToEmpty = 0;
}
```