

Process Model of an OS

- Modern OSes are a collection of cooperating processes that run on top of (and are supported by) an OS kernel
- The kernel is responsible for the following services:
 - Creation and destruction of processes
 - CPU scheduling, memory management, device management
 - Process synchronization tools
 - Process communication tools
- OS services provided by the kernel are invoked using *system calls*

Precedence & Concurrency

- Logical concurrency is achieved on a uni-processor system by quickly switching the CPU from one process to the next
- Consider the following two processes which share data:

P1 : $A = B + C$

P2 : $D = A * 2$

- P1 must precede P2 because we need value for A before we can do anything with it
- In general, the issues of precedence and concurrency are the same for logical or physical concurrency
- When is it okay for two or more processes to execute concurrently so that we always get consistent results?
 - it depends on what resources are being shared between the processes

Race Conditions

Consider the following example, where P1 and P2 share all data:

```
P1: z = 0;  
    B = 1;  
    C = 2 + B;
```

```
P2: z = 1;  
    B = 2;  
    D = z + B;
```

➤ When there is a dependency on the exact execution order of statements between two or more processes, it is called a *race condition*

Read & Write Sets

- A process' *read set* is the set of all data in RAM, secondary storage, or other existent data that a process reads (uses during execution)
 - read set for data denoted $R(P)$
- A process' *write set* is the set of all data that a process writes (changes during execution)
 - write set for data denoted $W(P)$
- E.g.: P1: $z = x + y$ P2: $a = a + 3$

 $R(P1) = \{x, y\}$ $R(P2) = \{a\}$
 $W(P1) = \{z\}$ $W(P2) = \{a\}$


Bernstein's Concurrency Conditions

- Used to dictate when two processes are able to execute concurrently, and always produce consistent results
- Bernstein's Concurrency Conditions are as follows:
In order for two processes P1 and P2 to run concurrently, the following 3 conditions must hold:
 1. $R(P1)$ cannot intersect $W(P2)$
 2. $W(P1)$ cannot intersect $R(P2)$
 3. $W(P1)$ cannot intersect $W(P2)$
- If two processes do not satisfy BCC, then they are said to have a *critical section* problem
 - Critical sections are the sections of code that violate BCC

Process Flow Graphs

- A *process flow graph* is a simple directed graph that depicts precedence and concurrency relationships among a group of processes
- The graph is said to be *properly nested* if it can be described by a simple composition of the process functions P and S:
 - $P(P1, P2)$ denotes a parallel execution of two processes, beginning and ending at the same point in the graph
 - $S(P1, P2)$ denotes a serial execution of P1 followed by P2
- Note that both P & S have only 2 arguments!

Process Flow Graph Examples



$s(s(s(1, 2), 3), 4)$

$p(1, p(p(2, 3), 4))$

Process Flow Graph Examples

$s(s(1, p(s(p(2, 3), 4), 5)), 6)$

- S & P compositions are difficult to read and write, and are unable to describe non-properly nested situations

Process Creation Constructs

- One mechanism for creating processes is called **fork and join**
- `Fork(label L)` produces 2 concurrent processes, one starts immediately after the fork statement, and one starts at label L
 - splits a single process execution into two concurrent processes
- `Join(int x)` recombines x processes into 1, effectively throwing away the first x-1 processes that reach it, and continuing execution after the Join statement, when the xth process reaches it

Fork and Join Examples



Cobegin/Coend Construct

- This is just another way of writing S() and P() functions
 - Only works for properly nested graphs
- Statements written between a cobegin/coend pair are executed in parallel
 - If statements are nested, then they all begin immediately after the cobegin statement, and the last one to finish does so immediately before the coend statement
- Statements written between a begin/end pair are executed in serial, in the order they appear

Cobegin/Coend Examples



Critical Sections

- Problem Definition
- Software Solutions
- Hardware Solutions
- Semaphores
- Monitors
- Inter-Process Communication

The Critical Section Problem

➤ Critical Sections:

- Sections of code in separate processes that do not obey Bernstein's conditions
- A solution will provide some method of only allowing one process to access their critical section at a time.
- Two critical sections are said to be *related* if they are in separate processes and do not obey Bernstein's conditions.

E.g. P1: $x = 1$
 P2: $x = 2$
 $y = 2$
 P3: $y = 3$

In this case, P1 and P2 have related critical sections, P2 and P3 have related critical sections, but not P1 and P3. P1 and P3 can execute in parallel just fine

Example: Producer / Consumer

Common data structure:

```
typedef struct node {  
    int item;  
    node *next; } NODE;
```

Producer:

```
/* produce a new item */  
    (big piece of code)  
newnode = (NODE *)malloc(sizeof(NODE));  
newnode->item =NewItem;  
newnode->next = first;  
first = newnode;
```

Consumer:

```
While(!first);  
//first is not null anymore  
when the producer has added  
something to the list  
MyNode = first;  
First = first->next;  
Item = myNode->item;  
/*Consume an item  
//Some other big piece of  
code
```

Example: Producer / Consumer

Producer's item ignored:

```
C: mynode = first
P: newnode->next = first
P: first = newnode
C: first = first->next
```

Consumer's deletion ignored:

```
C: mynode = first
P: newnode->next = first

C: first = first->next
P: first = newnode
```