Today’s Plan

Upcoming:
- Quiz #2
- Assignment 1

Today’s topics:
- From last time:
  - The Process Concept
- Concurrency
- Race Conditions
- Process Creation
- The Critical Section Problem

Last time:
- Finished chapter 2
- Process concept
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: $D = A \times 2$

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?
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)

- A process’ write set is the set of all data that a process writes (changes during execution)

- E.g.: P1: \( z = x + y \)  
  
  P2: \( a = a + 3 \)
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.
  2.
  3.

＞ If two processes do not satisfy BCC, then they are said to have a critical section problem
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:

Note that both P & S have only 2 arguments!
Process Flow Graph Examples

$s(s(s(1,2),3),4)$ \hspace{2cm} $p(1,p(p(2,3),4))$
Process Flow Graph Examples

\[ s(s(1,p(s(p(2,3),4),5)),6) \]

\( \Rightarrow \)  S & P compositions are difficult to read and write, and are unable to describe non-properly nested situations
Cobegin/Coend Construct

- This is just another way of writing $S()$ and $P()$ functions

- 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
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

- **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
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.
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:
Example: Producer / Consumer

Producer’s item ignored:

Consumer’s deletion ignored:
Conditions for a Solution

1. The execution of related critical sections must be mutually exclusive, that is, no two processes may access related critical sections simultaneously.

2. One process, A, outside of its critical section, cannot prevent another process, B, from entering any of B's critical sections.

3. When two processes want to enter their critical sections at the same time, the decision as to which one gets to go first cannot be postponed indefinitely.
Pi and Pj share data, are in infinite loops, and are identical except for i’s and j’s

Pi:
while(1) {
    while(turn != i);
    Critical Section
    turn = j;
    Non-Critical Section
}

Pj:
while(1) {
    while(turn != j);
    Critical Section
    turn = i;
    Non-Critical Section
}
Software Non-Solution #2

\[ P_i: \]
while(1) {
    while(flag[j]);
    flag[i] = true;
    Critical Section
    flag[i] = false;
    Non-critical Section
}

\[ P_j: \]
while(1) {
    while(flag[i]);
    flag[j] = true;
    Critical Section
    flag[j] = false;
    Non-critical Section
}
Software Non-Solution #3

\[ P_i: \]
\[
\text{while}(1) \{ \\
\quad \text{flag}[i] = \text{true}; \\
\quad \text{while}(\text{flag}[j]); \\
\quad \text{Critical Section} \\
\quad \text{flag}[i] = \text{false}; \\
\quad \text{Non-critical Section} \\
\}
\]

\[ P_j: \]
\[
\text{while}(1) \{ \\
\quad \text{flag}[j] = \text{true}; \\
\quad \text{while}(\text{flag}[i]); \\
\quad \text{Critical Section} \\
\quad \text{flag}[j] = \text{false}; \\
\quad \text{Non-critical Section} \\
\}
\]
A Software Solution

while(1) {
    flag[i] = true;
    while(flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j);
            flag[i] = true;
        }
    }
}

Critical Section

turn = j;
flag[i] = false;

Non-critical Section

while(1) {
    flag[j] = true;
    while(flag[i]) {
        if (turn == i) {
            flag[j] = false;
            while (turn == i);
            flag[j] = true;
        }
    }
}

Critical Section

turn = i;
flag[j] = false;

Non-critical Section