What Happened On Canvas?

• Late Wednesday, the pages for ECS 150 on Canvas disappeared.
• I chatted with the Canvas folks (on the phone, wait time was very long).
• They suggested I try something, I did, and it didn’t work.
• They escalated the problem, and around noon it got fixed.
• What happened?
  • Apparently when a new section was added, the ECS 150 pages on Canvas were re-initialized
  • The folks to whom the problem was escalated were able to put everything back (phew!)
New Section Is Opened

• The new section was approved late Wednesday
• It’s in 55 Roessler on Wednesday from 1:10pm to 2:00pm
• Please do not ask me if you will get in; I don’t know
  • I do know that graduating seniors in CS and ECS will have priority
• If you would prefer to go to the other discussion section, whichever one it is, you can just go and sit in if there is an empty seat
• Changing sections using a PTA has lots of problems
Interprocess Synchronization and Communication
What’s the Problem?

• Processes executing simultaneously
  • Multiple cores or CPUs
  • Process uses the GPU or FPU for computation

• Some statements must be completed before others are begun
  a ← x + y
  b ← z + 1
  c ← a + b
  d ← c + 1

• Here, the first two statements must be executed before the third or fourth (precedence constraint)

• But the first two can be done independently
Precedence, Process Flow Graphs

- Precedence graph focuses on statements
- Process flow graph focuses on processes
- Both must be acyclic graphs
- And, they are equivalent
Bernstein Conditions

- Describe when statements can be executed in parallel
- \( R(S_i) \): set of variables that are read in statement \( S_i \)
- \( W(S_i) \): set of variables that are written in statement \( S_i \)
- Bernstein conditions for statements \( S_i \) and \( S_j \):
  \[ R(S_i) \cap W(S_j) = \emptyset \text{ and } R(S_j) \cap W(S_i) = \emptyset \text{ and } W(S_i) \cap W(S_j) = \emptyset \]
Bernstein Conditions

• Remember this?
  
  \[
  \begin{align*}
  a & \leftarrow x + y \\
  b & \leftarrow z + 1 \\
  c & \leftarrow a + b \\
  d & \leftarrow c + 1 
  \end{align*}
  \]

• In the above example:
  
  \[
  \begin{align*}
  R(S_1) &= \{ x, y \} \\
  R(S_2) &= \{ z \} \\
  R(S_3) &= \{ a, b \} \\
  R(S_4) &= \{ c \} \\
  W(S_1) &= \{ a \} \\
  W(S_2) &= \{ b \} \\
  W(S_3) &= \{ c \} \\
  W(S_4) &= \{ d \} 
  \end{align*}
  \]

• As \( W(S_1) \cap R(S_3) = \{ a \} \neq \emptyset \), 1 and 3 must be executed sequentially.

• As \( R(S_1) \cap W(S_2) = \emptyset \) and \( R(S_2) \cap W(S_2) = \emptyset \) and \( W(S_1) \cap W(S_2) = \emptyset \), 1 and 2 can be executed in parallel.
Parallel Programming: *fork, join, quit*

• **fork** $L$
  - Split process in two; first begins after the fork, second begins at $L$
  
  Example:
  
  ```
  fork $L$
  a ← $x+y$;
  ...
  $L$: $b ← z + 1$
  ```

• **join** count, $L$
  - Decrement *count* and if 0, branch to $L$
  
  In other words:
  
  ```
  count ← count – 1
  if count = 0 then
goto $L$
  ```
Example

count ← 2;
fork dopar
a ← x + y;
goto endpar
dopar: b ← z + 1;
endpar: join count, next
quit
next: c ← a - b
d ← c + 1

• This computes:
  a ← x + y
  b ← z + 1
  c ← a + b
  d ← c + 1

with the first two lines executing in parallel, and then after those the last two lines execute sequentially
More Complicated Example

\[
\begin{align*}
t6 & \leftarrow 2; \\
t8 & \leftarrow 3; \\
S1; & \text{fork } p2; \text{fork } p5; \text{fork } p7; \\
p2: & S2; \text{fork } p3; \text{fork } p4; \text{quit} \\
p5: & S5; \text{join } t6,p6; \text{quit}; \\
p7: & S7; \text{join } t8,p8; \text{quit}; \\
p3: & S3; \text{join } t8,p8; \text{quit}; \\
p4: & S4; \text{join } t6,p6; \text{quit}; \\
p6: & S6; \text{join } t8,p8; \text{quit}; \\
p8: & S8; \text{quit};
\end{align*}
\]
Comments

• Advantages
  • simple
  • powerful
  • easy to derive from precedence or process flow graphs

• Disadvantages
  • clumsy
  • lots of gotos and goto-like structures
parbegin, parend

• These bracket statements or blocks to be done in parallel
• Eliminates gotos and goto-like structures
• Example:

```plaintext
parbegin
    a ← x + y
    b ← z + 1
parend
    c ← a - b;
    d ← c + 1
```
Comments

• Advantages
  • easy to read
  • uses principles of modular programming
  • avoids goto-like structures

• Disadvantages
  • not as powerful as the fork-join-quit primitives
Why?

• Consider the concept of proper nesting

• $S(a, b)$: represents serial execution of processes $a, b$

• $P(a, b)$: represents parallel execution of processes $a, b$

• A process flow graph is *properly nested* if it can be described by $P, S,$ and functional composition
Example of Proper Nesting

- The program
  \[
  \text{parbegin} \\
  a \leftarrow x + y \\
  b \leftarrow z + 1 \\
  \text{parend} \\
  c \leftarrow a - b; \\
  d \leftarrow c + 1
  \]

The process flow graph

The functional representation

So it is properly nested
CLAIM: This is not properly nested

PROOF: For something to be properly nested, it must be of the form $S(p_i, p_j)$ or $P(p_i, p_j)$ at most interior level.

It’s not $P(p_i, p_j)$ as there are no constructs of that form in the graph.

All serially connected processes $p_i, p_j$ have at least 1 more process $p_k$ starting or finishing at the node $n_{ij}$ between $p_i$ and $p_j$; but if $S(p_i, p_j)$ is the innermost level, there cannot be any such $p_k$, because if it existed, another, more interior $P$ or $S$ must be present, contradiction. So it’s not $S(p_i, p_j)$ either.
What This Means

• *fork, join, quit* can represent more complex structures than *parbegin* and *parent*

• *parbegin, parent* require the process flow graph to be properly nested
The Problem with Process Interaction

- Consider the following implementation of the *producer-consumer problem*
- One process (producer) generates items that it must pass to the other process (consumer)
  - Consumer must wait for the producer to produce an item
  - Producer must not produce more items when buffer is full
- Sometimes called the *bounded buffer problem*
The Problem with Process Interaction

• The variables
  • buffer and counter are shared variables
  • counter can assume values between 0 and n inclusive

```plaintext
var buffer: array [0..n-1] of item;
in, out: 0...n-1;
counter: 0...n
```
Producer and Consumer Code

• Producer code

```javascript
producer:
repeat
    make next p;
    while counter = n do
        (* nothing *);
    buffer[in] ← next p;
    in ← (in+1) mod n;
    counter ← counter + 1;
until false;
```

• Consumer code

```javascript
consumer:
repeat
    while counter = 0 do
        (* nothing *);
    next ← buffer[out];
    out ← (out + 1) mod n
    counter ← counter - 1;
until false;
```
Does It Work In Parallel?

• Suppose counter is 5, and consider the lines counter ← counter + 1 and counter ← counter − 1.

• They could compile into the following:

counter = counter + 1:

P1: r1 ← counter
P2: r1 ← r1 + 1
P3: counter ← r1

counter ← counter − 1:

C1: r2 ← counter
C2: r2 ← r2 − 1
C3: counter ← r2
A Race Condition

- Depending on how the statements intermingle, you get different values for count

- P1 P2 C1 C2 P3 C3  counter is 4
- P1 P2 P3 C1 C2 C3  counter is 5
- P1 P2 C1 C2 C3 P3  counter is 6
Critical Section Problem

• Critical section: block of code that only one process at a time can execute
  • When one process is in its critical section, no other process can be in its corresponding critical section

• Problem: design a protocol to do this

• *Generic description of solution framework*: entry section, critical section, exit section, remainder section
Requirements for Solution

• *Mutual Exclusion*: at most 1 process can be in the critical section at any time

• *Progress*: if no process is in the critical section, and several other processes wish to enter, then only processes not in the remainder section can take part in deciding which process enters

• *Bounded Wait*: a bound on the number of times other processes are allowed to enter the critical section after a process asks to enter the critical section and before it is allowed to

**Implicit assumption**: each process runs at non-zero speed, but *no assumption is made as to relative speed*
Background

• We use 2 processes, \( p_i \) and \( p_j \)
• Either \( i = 0 \) and \( j = 1 \) or \( j = 0 \) and \( i = 1 \)
• Current process is always \( p_i \) and the other one is \( p_j \)

• First, we’ll analyze several proposed solutions
Proposed Solution 1

var turn: 0..1; // whose turn it is
while turn ≠ i do // ... entry section
    /* nothing */

    // ... critical section

    turn = j; // ... exit section
Proposed Solution 1 Analysis

• Mutual exclusion? Yes; turn can only have 1 value, and second line blocks the process that does not have that value from entering critical section

• Progress? No; processes must enter the critical section in alternate order; so a process in the remainder section takes part in deciding which process enters the critical section
Proposed Solution 2

var inCS: array[0..1] of boolean = false;

    // who is in critical section

while inCS[j] do // ... entry section
    /* nothing */

inCS[i] = true

. . . // ... critical section

inCS[i] = false; // ... exit section
Proposed Solution 2 Analysis

• Mutual Exclusion: No; suppose $p_i$, $p_j$ execute the while statement at the same time. As both $\text{inCS}[i]$ and $\text{inCS}[j]$ are false, both enter the critical section.
Proposed Solution 3

var interested: array[0..1] of boolean = false;
    // who wants to enter critical section
interested[i] = true;  // ... entry section
while interested[j] do
    /* nothing */
. . .  // ... critical section
interested[i] = false;  // ... exit section
Proposed Solution 3 Analysis

- Mutual Exclusion: Yes; a process cannot enter the critical section unless $\text{interested}[j]$ is false but if a process is in the critical section, $\text{interested}[j]$ must be true.

- Progress: No; suppose both processes arrive at the while statement at the same time; as both elements of $\text{interested}[]$ are true, they loop forever.
Proposed Solution 4

var interested: array[0..1] of boolean = false;

    // who wants to enter critical section

turn: 0..1;

interested[i] = true;  // ... entry section

turn = j;

while interested[j] and turn = j do

    /* nothing */

. . .  // ... critical section

interested[i] = false;  // ... exit section
Proposed Solution 4 Analysis

• Mutual Exclusion: Yes. $p_i$ enters the critical section only if $\text{interested}[j]$ is false and turn is $i$. For $p_i$, $p_j$ both to be in the critical section, both elements of $\text{interested}[]$ must be true. Only one could have passed the while loop (as turn is $i$ or $j$ but not both) so one does the loop (say, $p_j$) and the other does the preceding lines. After the first line in entry section, both elements of $\text{interested}[]$ are true, but turn is $j$, so only $p_j$ enters the critical section. So only 1 process can be in the critical section at a time.
Proposed Solution 4 Analysis

- Progress: Yes. $p_i$ blocked from entering critical section only if it is stuck at the `while` loop, which means `interested[j]` is true and `turn` is $j$. If $p_j$ is not in the entry or critical sections, `interested[j]` is false and $p_i$ goes in.
  
  If $p_j$ is at the `while` statement, `turn` is either $i$ or $j$, and the process with index `turn` will go in. Once $p_i$ leaves the critical section, `interested[i]` is false and $p_j$ can go in. If $p_j$ resets `interested[j]` to true, then `turn` is set to $i$ and $p_i$ goes in. So only processes in the entry, exit, or critical section affect which process goes in, demonstrating progress.

- Bounded wait: Yes. At most one additional entry by $p_j$ will occur if both request entry at the same time, so the wait is bounded.