Announcements

• Extra Office Hour: tomorrow at 11am

• Slides for today are posted

• Slides from Friday are posted

• Homework is due April 13, not April 11
  • This includes extra credit

• All graduating seniors should have received a PTA for the new section. If you are a graduating senior and did not, please contact the Undergraduate Advisors immediately!!!!!
Interprocess Synchronization and Communication
Solutions in Software

• Last class’ solution was Peterson’s Solution
• Lamport’s bakery algorithm solves the $n$-process problem
Lamport’s Bakery Algorithm

var choosing: shared array[0..n-1] of boolean;
number: shared array[0..n-1] of integer;
...
repeat
    choosing[i] := false;           / ... eEntry section
    number[i] := max(number[0],number[1],...,number[n-1]) + 1;
    choosing[i] := false;
    for j := 0 to n-1 do begin
        while choosing[j] do         (* nothing *);
            while number[j] ≠ 0 and (number[j], j) < (number[i],i) do
                (* nothing *);
        end;
    / ... critical section
    number[i] := 0;                / ... exit section
    until false;
Explanation

• $choosing[i]$: true if process $i$ is choosing a number
• $number[i]$: number that process $i$ will use to enter the critical section; 0 if process $i$ is not trying to enter its critical section

**Entry section:**
• Process $i$ signals it is choosing a number
• Process $i$ tries to get a unique number
  • May not happen due to race
• Process $i$ indicates it is done
Explanation

*Which process goes in:*

- Process *i* waits until it has the lowest number of all the processes waiting to enter the critical section.
  - If two processes have the same number, the one with the smaller name (like *i*) goes in
  - If another process is choosing a number when process *i* tries to look at it, process *i* waits until it has done so before looking.

*Exit section*

- Process *i* no longer interested in entering its critical section, so it sets *number*[i] to 0.
Proof It Is a Solution

• Mutual exclusion: Suppose process $i$ is in critical section. Some other process $k$ ($k \neq i$) gets $number[k] \neq 0$. Assume $i < k$; then

$$(number[i],i) < (number[k],k).$$

Suppose process $k$ wants to enter the critical section, and process $i$ is in the critical section. When process $k$ is in the for loop, and $j = i$, then $number[i] \neq 0$ and $(number[i],i) < (number[k],k)$, so it loops in second while statement

• Are bounded wait and progress satisfied? Yes, as processes enter the critical section on FIFO basis.
Hardware Indivisible Test-and-Set Instruction

• This is atomic, and cannot be interrupted:

  ```plaintext
  function TaS(var Lock: boolean): boolean
  begin
    TaS: = Lock;
    Lock = True;
  end;
  ```

• It sets `Lock` to true and returns the previous value of `Lock`
Test-and-Set $n$ Process Solution: Variables

\[
\begin{align*}
\text{var } & \text{ waiting: shared array [0..}n-1\text{] of Boolean }\leftarrow \text{ false;} \\
\ & \text{ Lock: shared Boolean }\leftarrow \text{ false;} \\
\ & \ j: 0..n-1; \\
\ & \text{ key: boolean;}
\end{align*}
\]

- \textit{Waiting, Lock} are shared by all $n$ processes
- $j, \text{ key}$ are local variables
Test-and-Set $n$ Process Solution: Entry Section

repeat (* process Pi *)
    waiting[i] := true;
    key := true;
    while waiting[i] and key do
        key := TaS(Lock);
        waiting[i] := false;
• Process $i$ indicates it wants to go into critical section
• If $Lock$ is true, then $key$ will be true and process $i$ loops at the **while** statement
• When it can enter $key$ is false, so it resets $waiting[i]$ and enters. Note the $TaS(Lock)$ that sets $key$ to false also sets $Lock$ to true
Test-and-Set $n$ Process Solution: Exit Section

\[
j := i + 1 \mod n;
\]
while ($j \neq i$) and not waiting[$j$] do
\[
j := j + 1 \mod n;
\]
if $j = i$ then Lock := false
else waiting[$j$] := false;
until false;

• Process $i$ exits and must choose who goes next
• If one (process $j$) is waiting, process $i$ lets it proceed by setting waiting[$j$] to true; note Lock remains true.
• If none are waiting, Lock is set to false
Problems of All These

- Busy waiting; the CPU does nothing in such a way that no-one else can use it while the process is waiting
- Not easily generalizable
  - For example, Peterson's solution does not easily generalize to $n$ processes
- So look for other solutions . . .
Semaphores

• Non-negative integer variable $sem$ that has 3 allowed operations:
  • Initialization: initial value set atomically, as in
    \[ sem <- n \]
  • signal: increment value of $sem$ by 1, as in
    \[ sem <- sem + 1 \]
  • wait: block until value of $sem$ is non-zero; then decrement value by 1, as in
    \[ \textbf{while } sem = 0 \textbf{ do block} \]
    \[ sem <- sem - 1 \]
Blocking

• Each semaphore has an associated blocking (or waiting) queue
• When a process blocks, it goes into a queue
• When semaphore is non-zero, first process in queue is moved to the ready queue
• Processes normally are removed from the queue in FIFO order
Example

S1;
parbegin
  begin S2; signal(a); signal(b); end;
  begin wait(a); S3; signal(c); end;
  begin wait(b); S4; signal(d); end;
  begin S5; signal(e); end;
  begin wait(d); wait(e); S6; signal(f); end;
  begin S7; signal(g); end;
  begin wait(c); wait(f); wait(g); S8; end;
parend;
Semaphore Solution to Critical Section

• Initialize semaphore (call it \textit{mutex}) to 1
• Then \textit{wait} at the beginning of the critical section
• On exit, \textit{signal}

\begin{verbatim}
semaphore mutex <- 1;

repeat
    wait(mutex);
    // critical section
    signal(mutex);
until false;
\end{verbatim}
Process Synchronization Using Semaphores

semaphore mutex <- 0;

Process 1
repeat
...  
wait(mutex);
...  
until false;

Process 2
repeat
...  
signal(mutex);
...  
until false;
Producers-Consumers Problem

- Initialize $full$ to 0
- Initialize $empty$ to $n$ (size of buffer)
- Initialize $mutex$ to 1 – used to enforce mutual exclusion for access to the buffer
- Producer:
  - $\text{wait}(empty); \text{wait}(mutex); \text{item into buffer; signal}(mutex); \text{signal}(full)$
- Consumer:
  - $\text{wait}(full); \text{wait}(mutex); \text{item from buffer; signal}(mutex); \text{signal}(empty)$
Demonstration

• Suppose *empty* is *n*, meaning the buffer is empty
• Consumer wants an item, but blocks at *wait(full)*
• Producer wants to produce item, so at *wait(empty)*, it decrements *empty*, puts item into buffer, and signals *full* to indicate there is an item in buffer
• Now, if buffer is full, *empty* is 0 and *full* is *n*
• Producer wants to produce an item, but has to wait for buffer to have an empty spot; so it blocks on *empty*
• When consumer wants to take an item, at *wait(full)* it decrements *full*, consumes the item, and signals *empty* to indicate there is an empty space in buffer
Readers-Writers Problem

• Processes share a file
• Some processes want to read it (the *readers*)
• Others want to write it (the *writers*)
• Rules:
  • Any number of readers can access the file simultaneously
  • When a writer is accessing the file, no other process (reader or writer) can access the file
Versions

• First version: readers have priority
  • Even if a writer wants to access the file, it must wait until all readers are finished with the file and no readers want access to the file
  • Note: writers may never be able to access the file (said as “writers may starve”)

• Second version: writers have priority
  • Once a writer wants access to the file, no readers may obtain access
  • Any readers with access continue to have access
Demonstration (First Readers-Writers)

• Reader wants to read the file
  • Sets mutual exclusion
  • Adds that another reader wants to go in
  • Release mutual exclusion
  • If no other readers in critical section, wait for any writers
  • If other readers in critical section, or no writers, enter critical section
  • On exit, set mutual exclusion
  • Decrement number of readers; if last one, signal any writers they can proceed
  • Release mutual exclusion

• Summary
  • Add 1 to the number of readers in, or wanting to enter, critical section
  • If other readers in critical section, or no writers, enter critical section; otherwise, wait
  • On exit, subtract 1 from the number of readers in or wanting to enter
  • If no more readers, signal any writers
Demonstration (First Readers-Writers)

- **Writer wants to write the file**
  - Block until no readers and no other writers are in the critical section
  - Set mutual exclusion for the critical section
  - Enter
  - Release mutual exclusion

- **Summary**
  - Block until no other process is in the critical section
  - Enter the critical section
  - Unblock any waiting processes

- **Note**: mutual exclusion for critical section is *not* the same as for incrementing or decrementing the number of readers wanting to enter the critical section
Dining Philosophers Problem

• Five philosophers are dining at a circular table
• There are five plates, one in front of each philosopher
• There are five forks, one between each plate
• Philosophers alternate between thinking and using both their right and left forks to eat
• Problem: prevent starvation and deadlock
Possible Solution

• Each philosopher picks the fork on their left

```plaintext
var fork: array [0..4] of semaphore: = 1,1,1,1,1
repeat (* philosopher i *)
    wait(fork[i]);
    wait(fork[(i + 1) mod 5]);
    (* eat *)
    signal(fork[i]);
    signal(fork[(i + 1) mod 5]);
    (* think *)
until false
```
Do You See the Problem?

• Suppose all philosophers want to eat
• Each picks up their left fork ($\text{wait(fork}[i])$)
• All now want to pick up their right fork ($\text{wait(fork][(i + 1) mod 5]})$
• Oops . . . All right forks are the left forks of the philosophers to the right
• So all philosophers wait until the one to their right begins to think
• . . . Deadlock!
Problem

• Like fork/join/quit, semaphores are too low level
• Combine blocking with counting
  • Really two separate operations, and should be treated as such
• Hard to debug
  • Easy to make mistakes
  • Think of typing wait when you meant to type signal
  • Original name for wait (P), signal (V) even easier to mistype
    • P from the Dutch *passering* (“passing”)
    • V from the Dutch *verhogen* (“increase”)
    • Taken from railroad signals