Condition Locks

- `wait(condLock)` puts the thread to sleep until it is signalled. Can only be called if the thread owns `condLock`. The thread invisibly releases `condLock`
- `signal(condLock)` wakes a single thread that is waiting on the Lock. The thread must reacquire the lock again before it can continue execution.
- Broadcast releases all the waiting threads
- Make sure that the critical region is consistent at the wait since you are letting another process in
- Mesa style vs. Hoare style

Solution to Readers Writers Problem

```c
//Semaphore mutex initialized to 1
//Semaphore write initialized to 1
readers writers

wait(mutex); wait(write);
readers++;
if(readers == 1) WRITE;

wait(write);
signal(mutex); signal(write);

READ;

wait(mutex);
readers--;
if(readers == 0)
signal(write);
signal(mutex);
```

Condition Variables

- Last Time
- Condition Critical Regions
- Monitors
- Concurrency problems
- Readers Writers
- Dining Philosophers
- Frisbee Example

Last Time

- Homework Discussion
- NachOS project on class webpage
  - Partner Selection Assignment
  - Next time project overview
Dining Philosophers

Semaphore chopstick[4];

while(1) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat();
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
}

• Good enough?
• What can we do to solve the problem?

Example Synchronization Problems

Problem 5: (26 pts; 20 minutes) A mini-bus shuttles between three stops around the university circle going in the same direction. Passengers wait at their stop until the bus arrives. The bus has seats for 10 passengers. When the bus reaches a stop, the passengers going down leave the bus, and the new passengers climb on the bus if there is room (assume they leave from different doors). The bus driver waits for all the passengers to climb, before moving to the next station. You may use semaphores, locks and condition locks.

(a) (13 points) Write pseudocode to simulate this bus operation.

(b) (13 points) Assume that the bus has room for 5 standing passengers as well. A passenger sits if a seat is available, otherwise she stands. If a seat becomes available later, she sits down (however, if her station comes up first, she just leaves the bus). Write pseudocode to simulate this situation.

Problem 1, Second Midterm Fall 99

A joint bank account is being accessed by 3 of its co-owners at 3 different ATMs. The account has $10,000 in savings, and $500 in checking. One user transfers $1000 from savings to checking; another deposits $500 in checking; the third withdraws $500 from checking. The code for the three operations in the ATM controllers is:

withdraw(source, amount) {
    money[source] = money[source] - amount;
    dispense(amount); //produce bills to customer
}
deposit(destination, amount) {
    receive(amount); //mechanical receive of the envelope
    money[destination] = money[destination] + amount;
}
transfer(source, destination, amount) {
    money[destination] = money[destination] + amount;
    money[source] = money[source] - amount;
}

• (a) (10) What are the possible final values for the checking and savings accounts?
• (b) (10) Can you protect against this inconsistency? Suggest an implementation (complete serialization is unacceptable since there are so many customers using these functions); is deadlock a problem?

Other Problems

(Based on a final problem in 99) A monkey community lives on an island that has only one coconut tree. The monkeys take turn going up the tree (which can hold only 3 monkeys at a time). After each monkey grabs a coconut, it climbs down the tree to go and eat it.

(a) Simulate this synchronization problem; each monkey is an independent thread.

(b) Modify your solution if necessary to ensure that monkeys can be climbing up, or down the tree, but not both ways at the same time.

(c) The dominant monkey does not like to wait. When it arrives, it climbs up the tree ahead of any monkeys that got there before it (there still can only be three monkeys up the tree). Implement the synchronization for this problem (show the procedure for the dominant monkey thread and extend the procedure for the regular monkeys). Is starvation possible?
New Topic! CPU Scheduling

- Recall:
  - Long-term: may I come in?
  - medium-term: to swap or not to swap
  - short-term: scheduling of ready processes – our focus

- How/When: sometimes non-preemptive scheduling (no other choice) occurs
  - process switches from running to waiting state
  - process switches from running to ready (when an interrupt occurs)
  - process switches from being blocked to ready
  - process terminates

- All other scheduling is preemptive
- Scheduling problem: how do we decide what process to run next
  - What criteria should the scheduler worry about optimizing?

Potential Criteria

- Response time: minimize the amount of delay experienced by users (average? max? variance?)
- Turnaround time: minimize time between beginning and end of each job
- Waiting time: minimize the waiting time for each process
- CPU utilization: maximize the amount of work done by the CPU
- Throughput: maximize the number of processes completed per time unit
- Priority: give priorities to some processes over others
- Real-time scheduling: maximize deadlines met
- Fairness: give processes “fair” access to the CPU (how to define that?)

Process Behavior

- Alternating CPU and I/O bursts (think of your own programs)
- Most CPU bursts are short (some are long)
- CPU vs. I/O bound process classification
- CPU bound – very infrequent I/O (e.g., a numerical simulation)
- I/O bound – frequent I/O (e.g., a database application)
- Hysteresis – resist change
- other?
- Criteria can be conflicting
Shortest Job First Scheduling (SJF)

- Idea: Pick the shortest process to run next
- Provably optimal in minimizing average wait time (for non-preemptive)
- Redo the Gantt chart for the example
- Comments/Problems?

FCFS Discussion

- What do you think of this policy?
- Non-preemptive; a process continues running until blocked or terminated
- Gantt charts (space time charts) are used to illustrate scheduling

First Come First Serve

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>P4</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

SJF Discussion

- Is it fair?
- Non-preemptive, so CPU utilization is maximized
- What happens if a short process gets “stuck” behind a long process (e.g., 1msec and 10 msec processes that arrive together)
- More precisely, what is the average waiting time per process; is it minimized by FCFS?
- Consider the case where an I/O bound process and a CPU bound process are being scheduled – what happens?
- Can we do better?
Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (called quantum)
  - Quantum usually 10-100msecs
  - When this time has passed (quantum expires), the process is preempted
  - The process is added to the end of the ready queue

- What do you think?
  - What if we make the quantum (q) very large?
  - What if we make q very small?

- How does it compare to SJF and FCFS?

Discussion

- What should q be?
- Unfair to I/O bound processes – why?
  - Solution – have I/O bound processes come back to a higher priority

Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Assume the time quantum is 1 (what if 2, 5 or 10?)

- What is the average wait time, turnaround time, response time? What is the number of context switches?