NachOS Example

You've just been hired by Mother Nature to help her out with the chemical reaction to form water, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure `hReady` when it's ready to react, and each O atom invokes a procedure `oReady` when it's ready. For this problem, you are to write the code for `hReady` and `oReady`. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the procedures must call the procedure `makeWater` (which just prints out a debug message that water was made). After the `makeWater` call, two instances of `hReady` and one instance of `oReady` should return. Write the code for `hReady` and `oReady` using either semaphores or locks and condition variables for synchronization. Your solution must avoid starvation and busy-waiting.

Starting the Threads

```c
void Water() {
    Thread * t;
    int i;
    for(i=0;i<5;i++) {
        t = new Thread("Oxygen");
        t->Fork(oReady,i);
        t = new Thread("Hydrogen");
        t->Fork(hReady,i);
        t = new Thread("Hydrogen");
        t->Fork(hReady,i+5);
    }
}
```

Place this code in threadtest.cc and call it from `ThreadTest()`. Have a simple menu to test the different parts of the project; call the appropriate method depending on user selection.

Walkthrough

- Due 10/11
  - one per person – please email me team information (2 preferred, 1 is fine, 3 please talk to me)
- What is in a walkthrough? three things
  - A brief trace of the code as it is now explaining how it produces the output that it does
  - Your plan for implementing any two parts in the project
  - Answers to the questions in the walkthrough link on class page
- 2-3 pages max.

Administrivia

- Last time
  - Some synchronization problems
  - Started CPU Scheduling
- Today
  - NachOS I project overview (Another set of slides)
  - CPU Scheduling
Semaphore *Hydro = new Semaphore("H",0);
Semaphore *Oxy = new Semaphore("O",0);

void makeWater(int i) {
    printf("Oxygen Made Water \%d\n", i);
}

void oReady(int i) {
    Oxy->P();
    Oxy->P();
    makeWater(i);
    Hydro->V();
    Hydro->V();
}

void hReady(int i) {
    Oxy->V();
    Hydro->P();
    printf("Hydrogen \%d participated in making water and exited\n", i);
}

Sample Run

minating2% ./nachos -rs 100101
Oxygen Made Water 2
Hydrogen 2 participated in making water and exited
Hydrogen 7 participated in making water and exited
Oxygen Made Water 3
Hydrogen 3 participated in making water and exited
Hydrogen 8 participated in making water and exited
Oxygen Made Water 0
Hydrogen 0 participated in making water and exited
Hydrogen 5 participated in making water and exited
Oxygen Made Water 1
Hydrogen 1 participated in making water and exited
Hydrogen 6 participated in making water and exited
Oxygen Made Water 4
Hydrogen 4 participated in making water and exited
Hydrogen 9 participated in making water and exited
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

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
  - When we preemptively switch between processes, we pay for context switch
  - CPU utilization reduced, but response time also reduced

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>

- Idea: schedule processes in the order they arrive
  - Non-preemptive; a process continues running until blocked or terminated
- Gantt charts (space time charts) are used to illustrate scheduling
- What do you think of this policy?

FCFS 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., 1 msec 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?

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?

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?

SJF Discussion

- Starvation of long processes
  - Still a problem if a short process gets stuck behind a long process (any way to fix that?)
  - And just how do we know what the burst length is ahead of time?
  - Keep an exponential average of the burst length for the process: use it as estimate
  - *τ*^n+1_ n+1_ = ατ_ n_ + (1 - α)τ_ n_ * τ high, react quickly to changes (but might get fooled by a transient value)
  - * α low, react slow, but immune to transients
Discussion

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

<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>1</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

- Intuition – favor short processes, but give credit for long processes that have been waiting a long time
- Optimize the normalized turnaround value
- Define the Response Ratio as \( \frac{w + s}{s} \), where \( w \) is the wait time and \( s \) is the service time
- Non-preemptive (can we have a preemptive version? this was a homework problem last year)