Review: Semaphores

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    » This of this as the signal() operation
  - Only time can set integer directly is at initialization time

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

Review: Full Solution to Bounded Buffer

Semaphore fullBuffer = 0;  // Initially, no coke
Semaphore emptyBuffers = numBuffers;  // Initially, num empty slots
Semaphore mutex = 1;  // No one using machine

Producer(item) {
  emptyBuffers.P();  // Wait until space
  mutex.P();  // Wait until buffer free
  Enqueue(item);
  mutex.V();
  fullBuffers.V();  // Tell consumers there is more coke
}

Consumer() {
  fullBuffers.P();  // Check if there’s a coke
  mutex.P();  // Wait until machine free
  item = Dequeue();
  mutex.V();
  emptyBuffers.V();  // tell producer need more
  return item;
}
**Complete Monitor Example (with condition variable)**

- Here is an (infinite) synchronized queue

```java
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release(); // Release Lock
}

RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) {
        dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue(); // Get next item
    lock.Release(); // Release Lock
    return(item);
}
```

**Recall: Mesa vs. Hoare monitors**

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

  ```java
  while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
  - Why didn't we do this?
  if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
  ```

  - Answer: depends on the type of scheduling
    - Hoare-style (most textbooks):
      » Signaler gives lock, CPU to waiter; waiter runs immediately
      » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
    - Mesa-style (most real operating systems):
      » Signaler keeps lock and processor
      » Waiter placed on ready queue with no special priority
      » Practically, need to check condition again after wait

**Extended example: Readers/Writers Problem**

- Motivation: Consider a shared database

  - Two classes of users:
    » Readers - never modify database
    » Writers - read and modify database
  - Is using a single lock on the whole database sufficient?
    » Like to have many readers at the same time
    » Only one writer at a time

**Basic Readers/Writers Solution**

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time

- Basic structure of a solution:
  - Reader()
    Wait until no writers
    Access database
    Check out - wake up a waiting writer
  - Writer()
    Wait until no active readers or writers
    Access database
    Check out - wake up waiting readers or writer
  - State variables (Protected by a lock called "lock"):
    » int AR: Number of active readers; initially = 0
    » int WR: Number of waiting readers; initially = 0
    » int AW: Number of active writers; initially = 0
    » int WW: Number of waiting writers; initially = 0
    » Condition okToRead = NIL
    » Condition okToWrite = NIL
**Code for a Reader**

```c
Reader() {
    // First check self into system
    lock.Aquire();
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        okToRead.wait(&lock); // Sleep on cond var
        WR--; // No longer waiting
    }  
    AR++; // Now we are active!
    lock.release();  // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    lock.Aquire();
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        okToWrite.signal(); // Wake up one writer
    lock.Release();
}
```

**Code for a Writer**

```c
Writer() {
    // First check self into system
    lock.Aquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    lock.release();  // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Aquire();
    AW--; // No longer active
    if (WW > 0) { // Give priority to writers
        okToWrite.signal(); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        okToRead.broadcast(); // Wake all readers
    }
    lock.Release();
}
```

**Simulation of Readers/Writers solution**

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:
  - First, R1 comes along:
    - AR = 1, WR = 0, AW = 0, WW = 0
  - Next, R2 comes along:
    - AR = 2, WR = 0, AW = 0, WW = 0
  - Now, readers make take a while to access database
  - Situation: Locks released
  - Only AR is non-zero
- Next, W1 comes along:
  - AR = 2, WR = 0, AW = 0, WW = 0
- Can't start because of readers, so go to sleep:
  - AR = 2, WR = 0, AW = 0, WW = 1
- Finally, R3 comes along:
  - AR = 1, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  - AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and
  wakes up writer:
  - AR = 0, WW = 1
Simulation(3)

- When writer wakes up, get:
  \[ AR = 0, \ WR = 1, \ AW = 1, \ WW = 0 \]
- Then, when writer finishes:
  \[
  \begin{align*}
  &\text{if (WW > 0)} \{
  \text{// Give priority to writers}
  \text{okToWrite.signal();} \quad \text{// Wake up one writer}
  \} \text{ else if (WR > 0)} \{
  \text{// Otherwise, wake reader}
  \text{okToRead.broadcast();} \quad \text{// Wake all readers}
  \}
\]
  - Writer wakes up reader, so get:
    \[ AR = 1, \ WR = 0, \ AW = 0, \ WW = 0 \]
- When reader completes, we are finished

Questions

- Can readers starve? Consider Reader() entry code:
  \[
  \begin{align*}
  &\text{while ((AW + WW) > 0)} \{
  \text{// Is it safe to read?}
  \text{WR++;} \quad \text{// No. Writers exist}
  \text{okToRead.wait(&lock);} \quad \text{// Sleep on cond var}
  \text{WR--;} \quad \text{// No longer waiting}
  \}
  \text{AR++;} \quad \text{// Now we are active!}
  \\
  &\text{else if (WR > 0)} \{
  \text{// Otherwise, wake reader}
  \text{okToWrite.signal();} \quad \text{// Wake up one writer}
  \}
\]
- What if we erase the condition check in Reader exit?
  \[ AR--; \quad \text{// No longer active} \]
  \[
  \begin{align*}
  &\text{if (AR == 0 \&\& WW > 0)} \{
  \text{// No other active readers}
  \text{okToWrite.signal();} \quad \text{// Wake up one writer}
  \}
\]
- Further, what if we turn the signal() into broadcast()?
  \[ AR--; \quad \text{// No longer active} \]
  \[
  \begin{align*}
  &\text{okToWrite.broadcast();} \quad \text{// Wake up one writer}
  \}
\]
- Finally, what if we use only one condition variable (call it "okToContinue") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Administrivia

- Midterm coming up soon
  - Currently scheduled for Wednesday 3/11
  - Still working out the details
  - Intend this to be a 1.5-2 hour exam in 3 hour slot
- Topics will include the material from that Monday
- No class that day, extra office hours

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?
  \[
  \begin{align*}
  &\text{Wait()} \{ \text{semaphore.P();} \}
  \text{Signal()} \{ \text{semaphore.V();} \}
  \}
\]
- Does this work better?
  \[
  \begin{align*}
  &\text{Wait(lock lock)} \{
  \text{lock.Release();}
  \text{semaphore.P();}
  \text{lock.Acquire();}
  \}
  \text{Signal()} \{ \text{semaphore.V();} \}
  \}
\]
Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative - result is the same no matter what order they occur
  - Condition variables are NOT commutative

- Does this fix the problem?

  ```
  Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
  }
  ```

  ```
  Signal() {
    if semaphore queue is not empty
      semaphore.V();
  }
  ```

  - Not legal to look at contents of semaphore queue
  - There is a race condition - signaler can slip in after lock release and before waiter executes semaphore.P()

- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed

- Basic structure of monitor-based program:

  ```
  lock
  while (need to wait) {
    condvar.wait();
  }
  unlock
  ```

  ```
  do something so no need to wait
  ```

  ```
  lock
  condvar.signal();
  unlock
  ```

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know all the code paths out of a critical section

  ```
  int Rtn() {
    lock.acquire();
    ...
    if (exception) {
      lock.release();
      return errReturnCode;
    }
    ...
    lock.release();
    return OK;
  }
  ```

- Watch out for setjmp/longjmp!
  » Can cause a non-local jump out of procedure
  » In example, procedure E calls longjmp, popping stack back to procedure B
  » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

    ```
    void Rtn() {
      lock.acquire();
      ...
      DoFoo();
      ...
    lock.release();
    }
    void DoFoo() {
      ...
      if (exception) throw errException;
    }
    ```

  - Notice that an exception in DoFoo() will exit without releasing the lock
C++ Language Support for Synchronization (con’t)

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:
    ```
    void Rtn() {
      lock.acquire();
      try {
        ...  // DoFoo(); ...
      } catch (...) {  // catch exception
        lock.release(); // release lock
        throw;           // re-throw the exception
      }
      lock.release();
    }
    void DoFoo() {
      ...  // if (exception) throw errException; ...
    }
    
      » Can deallocate/free lock regardless of exit method
    ```

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:
  ```java
  class Account {
    private int balance;
    // object constructor
    public Account (int initialBalance) {
      balance = initialBalance;
    }
    public synchronized int getBalance() {
      return balance;
    }
    public synchronized void deposit(int amount) {
      balance += amount;
    }
  }
  ```
- Every object has an associated lock which gets automatically acquired and released on entry and exit from a synchronized method.

Java Language Support for Synchronization (con’t)

- Java also has synchronized statements:
  ```java
  synchronized (object) {
    ...  // DoFoo(); ...
  }
  
  - Since every Java object has an associated lock, this type of statement acquires and releases the object’s lock on entry and exit of the body
  - Works properly even with exceptions:
    ```java
    synchronized (object) {
      ...  // DoFoo(); ...
    }
    void DoFoo() {
      throw errException;
    }
    ```
  ```

Java Language Support for Synchronization (con’t 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method of block:
    » void wait(long timeout); // Wait for timeout
    » void wait(long timeout, int nanoseconds); // variant
    » void wait();
  - How to signal in a synchronized method or block:
    » void notify(); // wakes up oldest waiter
    » void notifyAll(); // like broadcast, wakes everyone
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:
    ```java
t1 = time.now();
while (!ATMRequest()) {
  wait (CHECKPERIOD);
t2 = time.new();
  if (t2 - t1 > LONG_TIME) checkMachine();
}
```  
  - Not all Java VMs equivalent!
    » Different scheduling policies, not necessarily preemptive!
Recall: Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int mylock = FREE;
Acquire(&mylock) - wait until lock is free, then grab
Release(&mylock) - Unlock, waking up anyone waiting

Acquire(int *lock) {
    disable interrupts;
    if (*lock == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        *lock = FREE;
    }
    enable interrupts;
}

Release(int *lock) {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Place on ready queue;
    } else {
        *lock = FREE;
    }
    enable interrupts;
}
```

- Really only works in kernel – why?

INIT

```c
int value = 0;
Acquire() {
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() //??
    } else {
        value = 1;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
```
Kernel Lock: Simulation

Discussion

- Notice that Scheduling here involves deciding who to take off the wait queue
  - Could do by priority, etc.
- Same type of code works for condition variables
  - The Wait queue becomes unique for each condition variable
  - Once again, transition two and from queues occurs with interrupts disabled
Recall: CPU Scheduling

- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is “fair” about fairness among users or programs?
    » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    » Time to echo a keystroke in editor
    » Time to compile a program
    » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    » Minimize overhead (for example, context-switching)
    » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    » Better average response time by making system less fair

Assumption: CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - In early systems, FCFS meant one program scheduled until done (including I/O)
    - Now, means keep CPU until thread blocks
  - Example: Process Burst Time
    | Process | Burst Time |
    |---------|------------|
    | P₁      | 24         |
    | P₂      | 3          |
    | P₃      | 3          |
  - Suppose processes arrive in the order: P₁, P₂, P₃
    - The Gantt Chart for the schedule is:

      | P₁ | P₂ | P₃ |
      |----|----|----|
      | 0  | 24 | 27 |
      | 24 | 27 | 30 |
    - Average waiting time: (0 + 24 + 27)/3 = 17
    - Average completion time: (24 + 27 + 30)/3 = 27

- Convoy effect: short process behind long process

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don’t care who is behind you, on the other hand...
- Round Robin Scheme
  - Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is q ⇒
    - Each process gets 1/n of the CPU time
    - In chunks of at most q time units
    - No process waits more than (n-1)q time units
- Performance
  - q large ⇒ FCFS
  - q small ⇒ Interleaved (really small ⇒ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:
  - Process | Burst Time |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>53</td>
</tr>
<tr>
<td>P₂</td>
<td>8</td>
</tr>
<tr>
<td>P₃</td>
<td>68</td>
</tr>
<tr>
<td>P₄</td>
<td>24</td>
</tr>
</tbody>
</table>
  - The Gantt chart is:

      | P₁ | P₂ | P₃ | P₄ |
      |----|----|----|----|
      | 0  | 20 | 28 | 48 |
      | 20 | 88 | 88 | 108|
      | 88 | 108| 112| 153|
      | 108| 153| 112| 153|
      | 153| 153| 153| 153|
  - Waiting time for P₁ = 68-20+112-88 = 72
  - Average waiting time: (68 + 0 + 3)/3 = 3
  - Average completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - Safeway: Getting milk, always stuck behind cart full of small items. Upside: get to read about space aliens!
- FCFS Scheduling (Cont.)
- Example continued:
  - Suppose that processes arrive in order: P₂, P₃, P₁
  - Now, the Gantt chart for the schedule is:

      | 0 | 3 | 6 | 30 |
      | P₂| P₃| P₁|
  - Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average completion time: (3 + 6 + 30)/3 = 13
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)
Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    » Response time suffers
  - What if infinite (∞)?
    » Get back FIFO
  - What if time slice too small?
    » Throughput suffers!

- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    » Worked ok when UNIX was used by one or two people.
    » What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    » Typical time slice today is between 10ms - 100ms
    » Typical context-switching overhead is 0.1ms - 1ms
    » Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time

- Completion Times:

<table>
<thead>
<tr>
<th>Job #</th>
<th>FIFO</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>991</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>992</td>
</tr>
</tbody>
</table>
| ...   | ...  | ...
| 9     | 900  | 999|
| 10    | 1000 | 1000|

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
- Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

<table>
<thead>
<tr>
<th>Best FCFS:</th>
<th>P₂</th>
<th>P₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>8</td>
<td>[24]</td>
</tr>
<tr>
<td>P₃</td>
<td>53</td>
<td>[68]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quantum</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td>32</td>
<td>0</td>
<td>85</td>
<td>8</td>
<td>31 1/2</td>
</tr>
<tr>
<td>Q = 1</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>Q = 4</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61 1/2</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57 1/2</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61 1/2</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66 1/2</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83 1/2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quantum</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best FCFS</td>
<td>85</td>
<td>8</td>
<td>153</td>
<td>32</td>
<td>69 1/2</td>
</tr>
<tr>
<td>Q = 1</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100 1/2</td>
</tr>
<tr>
<td>Q = 5</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99 1/2</td>
</tr>
<tr>
<td>Q = 8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95 1/2</td>
</tr>
<tr>
<td>Q = 10</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99 1/2</td>
</tr>
<tr>
<td>Q = 20</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104 1/2</td>
</tr>
<tr>
<td>Worst FCFS</td>
<td>121</td>
<td>153</td>
<td>68</td>
<td>145</td>
<td>121 1/2</td>
</tr>
</tbody>
</table>

What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time
Discussion

- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    » SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF

- Three jobs:
  - A, B: both CPU bound, run for week
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:

- Disk Utilization: 9/201 ~ 4.5%
- RR 100ms time slice: ~90% but lots of wakeups!
- RR 1ms time slice: Disk Utilization: 90%
- SRTF: Disk Utilization: 90%

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    » When you submit a job, have to say how long it will take
    » To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)
Predicting the Length of the Next CPU Burst

- **Adaptive**: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    » If program was I/O bound in past, likely in future
    » If computer behavior were random, wouldn’t help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let \( t_{n-1}, t_{n-2}, t_{n-3}, \ldots \) be previous CPU burst lengths. Estimate next burst \( t_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \)
  - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    \[ t_n = \alpha t_{n-1} + (1-\alpha)t_{n-1} \]
    with \( 0 < \alpha \leq 1 \)

Multi-Level Feedback Scheduling

- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    » Higher priority queues often considered “foreground” tasks
    - Each queue has its own scheduling algorithm
      » e.g. foreground - RR, background - FCFS
      » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    » serve all from highest priority, then next priority, etc.
  - Time slice:
    » each queue gets a certain amount of CPU time
    » e.g., 70% to highest, 20% next, 10% lowest
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job’s priority high
  - Of course, if everyone did this, wouldn’t work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors
  - Put in printf’s, ran much faster!

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    » long running jobs may never get CPU
    » In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    » What if one long-running job and 100 short-running ones?
    » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don’t get service
    » What is done in UNIX
    » This is ad hoc—what rate should you increase priorities?
    » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority→Interactive jobs suffer
Lottery Scheduling

• Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
• How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
• Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

• Assume short jobs get 10 tickets, long jobs get 1 ticket

<table>
<thead>
<tr>
<th># short jobs/ # long jobs</th>
<th>% of CPU each short jobs gets</th>
<th>% of CPU each long jobs gets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>91%</td>
<td>9%</td>
</tr>
<tr>
<td>0/2</td>
<td>N/A</td>
<td>50%</td>
</tr>
<tr>
<td>2/0</td>
<td>50%</td>
<td>N/A</td>
</tr>
<tr>
<td>10/1</td>
<td>9.9%</td>
<td>0.99%</td>
</tr>
<tr>
<td>1/10</td>
<td>50%</td>
<td>5%</td>
</tr>
</tbody>
</table>

- What if too many short jobs to give reasonable response time?
  » In UNIX, if load average is 100, hard to make progress
  » One approach: log some user out

How to Evaluate a Scheduling algorithm?

• Deterministic modeling
  - Takes a predetermined workload and compute the performance of each algorithm for that workload
• Queueing models
  - Mathematical approach for handling stochastic workloads
• Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.

Summary

• Semaphores: Like integers with restricted interface
  - Two operations:
    » P(): Wait if zero; decrement when becomes non-zero
    » V(): Increment and wake a sleeping task (if exists)
  - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
• Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
  » Three Operations: Wait(), Signal(), and Broadcast()
• Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
• FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
• Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length
Summary (2)

- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):**
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair

- **Multi-Level Feedback Scheduling:**
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

- **Lottery Scheduling:**
  - Give each thread a priority-dependent number of tokens (short tasks $\Rightarrow$ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness

- **Evaluation of mechanisms:**
  - Analytical, Queuing Theory, Simulation