Review: Condition Variables

- How do we change the `RemoveFromQueue()` routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- Condition Variable: a queue of threads waiting for something inside a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section

- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters

- Rule: Must hold lock when doing condition variable ops!
  - In Birrell paper, he says can perform `signal()` outside of lock – IGNORE HIM (this is only an optimization)

Review: Mesa vs. Hoare Monitors

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

Reader() {
  // First check self into system
  lock.Acquire();
  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.Acquire();
  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

Writer() {
  // First check self into system
  lock.Acquire();
  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.Acquire();
  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:
  
  while ((AW + WW) > 0) {
    // Is it safe to read?
    while ((AW + WW) > 0) {
      // No. Writers exist
      okToRead.wait(&lock); // Sleep on cond var
    }
    WR++; // No longer waiting
  }
  AR++; // Now we are active!

  - 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

**Simulation(2)**

- Next, W1 comes along:

  while ((AW + AR) > 0) {
    // Is it safe to write?
    while ((AW + AR) > 0) {
      // No. Active users exist
      W += 1;
      okToWrite.wait(&lock); // Sleep on cond var
    }
    WW++; // No longer waiting
    W++; // No longer waiting
  }
  AW++; // Can't start because of readers, so go to sleep:

  - AR = 2, WR = 0, AW = 0, WW = 1
  - Finally, R3 comes along:
    - AR = 2, 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:
    - if (AR == 0 && WW > 0) // No other active readers
      okToWrite.signal(); // Wake up one writer

**Questions**

- Can readers starve? Consider Reader() entry code:

  while ((AW + WW) > 0) {
    // Is it safe to read?
    while ((AW + WW) > 0) {
      // No. Writers exist
      okToRead.wait(&lock); // Sleep on cond var
    }
    WR++; // No longer waiting
  }
  AR++; // No longer active

  - Write wakes up reader, so get:
    - AR = 1, WR = 0, AW = 0, WW = 0
  - When reader completes, we are finished
**Administrivia**

- **Midterm on Wednesday 9/28 5-6:30PM**
  - I LeConte (Last name A-H) and 2050 VLSB (Last name I-Z)
  - Topics include course material through lecture 9 (today)
    » Lectures, project 1, homeworks, readings, textbook
  - Closed book, no calculators, **one double-side letter-sized page of handwritten notes**
  - Review today 6:45-8:30PM in Hearst Field Annex A1

- **No office hours on Mon 10/3 (covering CS 262A)**

- **Deadlines next week:**
  - HW2 due next Monday (10/3)
  - Project 1 code due next Wednesday (10/5)
  - Project 1 final report due next Friday (10/7)

---

**Can we Construct Monitors from Semaphores?**

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?
  ```cpp
  Wait() { semaphore.P(); }
  Signal() { semaphore.V(); }
  ```

- Does this work better?
  ```cpp
  Wait(Lock=lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
  }
  Signal() { semaphore.V(); }
  ```

---

**Construction of Monitors from Semaphores (cont'd)**

- 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?
  ```cpp
  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 (cont’d 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.

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

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

- Really only works in kernel – why?
In-Kernel Lock: Simulation

INIT

Init \text{value} = 0;

Acquire() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & (\text{value} == 1) \\
\text{put} & \text{thread on wait-queue; } \\
\text{go} & \text{to sleep(); } \text{??} \\
\text{else} & \\
\text{value} & = 1; \\
\text{enable} & \text{interrupts; }
\end{align*}

Release() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & \text{anyone on wait-queue; } \\
\text{take} & \text{thread off wait-queue; } \\
\text{Place} & \text{on ready queue; } \\
\text{else} & \\
\text{value} & = 0; \\
\text{enable} & \text{interrupts; }
\end{align*}

2/22/16 

Joseph CS162 ©UCB Spring 2016

Lec 9.25

Value: 0 
waiters owner READY

In-Kernel Lock: Simulation

INIT

Init \text{value} = 0;

Acquire() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & (\text{value} == 1) \\
\text{put} & \text{thread on wait-queue; } \\
\text{go} & \text{to sleep(); } \text{??} \\
\text{else} & \\
\text{value} & = 1; \\
\text{enable} & \text{interrupts; }
\end{align*}

Release() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & \text{anyone on wait-queue; } \\
\text{take} & \text{thread off wait-queue; } \\
\text{Place} & \text{on ready queue; } \\
\text{else} & \\
\text{value} & = 0; \\
\text{enable} & \text{interrupts; }
\end{align*}

2/22/16 

Joseph CS162 ©UCB Spring 2016

Lec 9.27

Value: 1 
waiters owner READY

In-Kernel Lock: Simulation

INIT

Init \text{value} = 0;

Acquire() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & (\text{value} == 1) \\
\text{put} & \text{thread on wait-queue; } \\
\text{go} & \text{to sleep(); } \text{??} \\
\text{else} & \\
\text{value} & = 1; \\
\text{enable} & \text{interrupts; }
\end{align*}

Release() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & \text{anyone on wait-queue; } \\
\text{take} & \text{thread off wait-queue; } \\
\text{Place} & \text{on ready queue; } \\
\text{else} & \\
\text{value} & = 0; \\
\text{enable} & \text{interrupts; }
\end{align*}

2/22/16 

Joseph CS162 ©UCB Spring 2016

Lec 9.26

Value: 1 
waiters owner READY

In-Kernel Lock: Simulation

INIT

Init \text{value} = 0;

Acquire() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & (\text{value} == 1) \\
\text{put} & \text{thread on wait-queue; } \\
\text{go} & \text{to sleep(); } \text{??} \\
\text{else} & \\
\text{value} & = 1; \\
\text{enable} & \text{interrupts; }
\end{align*}

Release() =

\begin{align*}
\text{disable} & \text{interrupts; } \\
\text{if} & \text{anyone on wait-queue; } \\
\text{take} & \text{thread off wait-queue; } \\
\text{Place} & \text{on ready queue; } \\
\text{else} & \\
\text{value} & = 0; \\
\text{enable} & \text{interrupts; }
\end{align*}

2/22/16 

Joseph CS162 ©UCB Spring 2016

Lec 9.28
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 in-kernel condition variables
  – The Wait queue becomes unique for each condition variable
  – Once again, transition to and from queues occurs with interrupts disabled

BREAK
Synchronization 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 zero or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
  » Three Operations: Wait(). Signal(). Broadcast()

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

Recall: 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

Recall: 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
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

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: 
<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>24</td>
</tr>
<tr>
<td>P_2</td>
<td>3</td>
</tr>
<tr>
<td>P_3</td>
<td>3</td>
</tr>
</tbody>
</table>
- Suppose processes arrive in the order: P_1, P_2, P_3
  The Gantt Chart for the schedule is:

```
   P_1   P_2   P_3
  0 24 27 30
```

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Average Completion time: (24 + 27 + 30)/3 = 27
- Convoy effect: short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2, P_3, P_1
    Now, the Gantt chart for the schedule is:

```
   P_2 P_3 P_1
  0 3 6 30
```

- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 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!

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 \Rightarrow
    » 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 \Rightarrow FCFS
  - q small \Rightarrow Interleaved (really small \Rightarrow 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:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>53</td>
</tr>
<tr>
<td>P_2</td>
<td>8</td>
</tr>
<tr>
<td>P_3</td>
<td>68</td>
</tr>
<tr>
<td>P_4</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>28</td>
<td>48</td>
<td>68</td>
<td>88</td>
<td>112</td>
<td>125</td>
</tr>
<tr>
<td>1</td>
<td>88</td>
<td>108</td>
<td>125</td>
<td></td>
<td>145</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>153</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Waiting time for
  - \( P_1 = (68-20) + (112-88) = 72 \)
  - \( P_2 = (20-0) = 20 \)
  - \( P_3 = (28-0) + (88-48) + (125-108) = 85 \)
  - \( P_4 = (48-0) + (108-68) = 88 \)

- Average waiting time = \( (72+20+85+88)/4 = 66\frac{1}{4} \)
- Average completion time = \( (125+28+153+112)/4 = 104\frac{1}{2} \)

- Thus, Round-Robin Pros and Cons:
  - 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 (\( \infty \))?
    » 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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>999</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

- 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>Quantum</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_4</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>32</td>
<td>103</td>
<td>85</td>
<td>8</td>
<td>31\frac{1}{4}</td>
</tr>
<tr>
<td>24</td>
<td>84</td>
<td>22</td>
<td>85</td>
<td>57</td>
<td>62</td>
</tr>
<tr>
<td>53</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61\frac{1}{4}</td>
</tr>
<tr>
<td>68</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57\frac{1}{4}</td>
</tr>
<tr>
<td>8</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>58</td>
<td>61\frac{1}{4}</td>
</tr>
<tr>
<td>153</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66\frac{1}{4}</td>
</tr>
<tr>
<td>85</td>
<td>68</td>
<td>145</td>
<td>0</td>
<td>121</td>
<td>83\frac{1}{2}</td>
</tr>
<tr>
<td>80</td>
<td>137</td>
<td>30</td>
<td>153</td>
<td>81</td>
<td>100\frac{1}{4}</td>
</tr>
<tr>
<td>153</td>
<td>135</td>
<td>28</td>
<td>153</td>
<td>82</td>
<td>99\frac{1}{4}</td>
</tr>
<tr>
<td>8</td>
<td>133</td>
<td>16</td>
<td>153</td>
<td>80</td>
<td>95\frac{1}{2}</td>
</tr>
<tr>
<td>153</td>
<td>135</td>
<td>18</td>
<td>153</td>
<td>92</td>
<td>99\frac{1}{2}</td>
</tr>
<tr>
<td>80</td>
<td>125</td>
<td>28</td>
<td>153</td>
<td>112</td>
<td>104\frac{1}{4}</td>
</tr>
<tr>
<td>153</td>
<td>121</td>
<td>68</td>
<td>145</td>
<td>121</td>
<td>121\frac{1}{4}</td>
</tr>
</tbody>
</table>
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: \( \text{wait}(), \text{signal}() \), and \( \text{broadcast}() \)

• **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