Recall: Yielding Thread

Cyan = user stack; Red = kernel stack

computePI() {
    while(TRUE) {
        ComputeNextDigit();
        thread_yield();
    }
}

run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}

Scheduler:
Later
Recall: CPU Scheduling

How to decide which thread to take off the ready queue?

Scheduling
Old scheduling assumptions

Welcome to 1970
- One program per user
- One thread per program
- Programs are independent

Simplifies the problem (E.g. "fairness" easier to understand)
Program Behavior: CPU/IO bursts

Programs alternate between CPU bursts, IO
- Scheduling decision: which CPU burst to run next

Why? Interactive programs!
- Wait for keyboard input
- Even webserver counts
Program Behavior: CPU/IO bursts
Scheduling Metrics

**Response time** *(want low)*
- What user sees: from *keypress* to *character on screen*

**Throughput** *(want high)*
- Total operations per second
- Problem: overhead (e.g. from context switching)

**Fairness**
- Many definitions
- **Conflicts** with best average throughput/response time
Outline: basic scheduling algorithms

Trivial
- First come first served (FCFS)
- Round robin (RR)
- Priority
Outline: basic scheduling algorithms

Trivial
- First come first served (FCFS)
- Round robin (RR)
- Priority
First-come, First-served (FCFS) (1)

Also "First In, First Out" (FIFO)

Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>24</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

Arrival order: A, B, then C

Response time: A = 24, B = 27, C = 30
Waiting time: A = 0, B=24, C = 27
First-come, First-served (FCFS) (2)

Response times: A = 24, B = 27, C = 30

Waiting times: A = 0, B=24, C = 27

Average response time: \( \frac{24 + 27 + 30}{3} = 27 \)

Average waiting time: \( \frac{0 + 24 + 27}{3} = 17 \)

Convoy effect: short processes behind long process
First-come, First-served (FCFS) (3)

Consider a different order:

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Waiting time: A = 6, B = 0, C = 3

Average waiting time: \((6 + 0 + 3)/3 = 3\) (other order 17)

Average response time: \((3 + 6 + 30) / 3 = 13\) (other order 27)
Outline: basic scheduling algorithms

Trivial
- First come first served (FCFS)
- Round robin (RR)
- Priority
Round Robin (RR) (1)

Give out *small* units of CPU time ("time quanta")
- 10-100 milliseconds typical

When time quantum expires, **preempt**, put to end

Each of N processes gets 1/Nth of the CPU
- If Q is time quantum,
  never wait more than \((N-1)Q\) time units to start running!

Downside:
- More context switches
- What should Q be? [Can it be too low? Too high?]
Round Robin (RR) (2)

Q = 20

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>53</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>68</td>
</tr>
<tr>
<td>D</td>
<td>24</td>
</tr>
</tbody>
</table>

Average waiting time: \((72 + 20 + 85 + 88)/4 = 66\frac{1}{4}\) (+ context switches)

Average response time: \((125+28+153+112)/4 = 104\frac{1}{4}\) (+ context switches)
Round Robin Overhead

Typical context switch: 0.1ms – 1ms

With 10ms – 100ms timeslice (Q): ~1% overhead
Question: Round Robin Quantum

If there was no overhead, *decreasing* the the time quantum Q will cause:

A. average response time to **always decrease or stay the same**

B. average response time to **always increase or stay the same**

C. average response time to **increase or decrease or stay the same**

D. Something else?
Increase Response Time

A: 1

B: 1

Q = 1

Q = 1/2

response time = \( \frac{1 + 2}{2} = \frac{3}{2} \)

response time = \( \frac{1.5 + 2}{2} = \frac{7}{4} \)
Decrease Response Time

A: 10

B: 1

Q = 10

avg response time = \( \frac{10+11}{2} = 10.5 \)

Q = 5

avg response time = \( \frac{6+11}{2} = 8.5 \)
Stay the Same

A: 1

B: 1

Q = 10

Q = 1
Round Robin (RR) (1)

Give out *small* units of CPU time ("time quanta")
- 10-100 milliseconds typical

When time quantum expires, preempt, put to end

Each of N processes gets 1/Nth of the CPU
- If Q is time quantum,
  **never wait more than \((N-1)Q\) time units!**

Downside:
- More context switches
- What should Q be? [Can it be too low? Too high?]

Maximum, not average!
Outline: basic scheduling algorithms

Trivial
- First come first served (FCFS)
- Round robin (RR)
- Priority
Strict Priority Scheduling (1)

- Something gives jobs priority
  - Usually users set explicitly

- Always run the ready thread with highest priority
  - Low priority threads might not run ever
  - Called "starvation"
Priority Scheduling (2)

Mechanism for many scheduling policies
- Just set priorities accordingly

Example: Shell higher priority than big computation
- More on this later
Trivial Scheduling: Summary

First-Come First Serve
- Simple queue ordered by arrival order
- Run until completion

Round robin
- Same as FCFS, but preempt after quantum

Priority
- Same as FCFS, but order queue by priority
- Run until completion or higher priority thread becomes ready
Logistics

**Project 1** is out

*Initial design doc due Thursday*

- Push to master
- PDF file or text file in git repository

**Sign up for design review time next week!**

Should have group assignments
Interlude: Correctness

Non-deterministism
- Scheduler can run threads in any order
- Scheduler can switch threads at any time

Makes bugfinding, testing very difficult

Solution: correctness by design
ATM Server

Multithreaded ATM server
– Correctness goal: don't lose/gain money!
ATM Server

BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}

ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}

Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}

Want to overlap disk I/O

"Solution": Multiple threads
ATM Server: Multiple Threads

BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ThreadFork(ProcessRequest, op, acctId, amount);
    }
}

ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}

Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}

Want to overlap disk I/O

"Solution": Multiple threads
ATM Server: The Race

Multiple requests running this:

```c
Deposit(acctId, amount) {
    acct = GetAccount(actId);   /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct);         /* Involves disk I/O */
}
```

May corrupt shared state:

**Thread A**

1. load r1, acct->balance
2. add r1, amount1
3. store r1, acct->balance

**Thread B**

1. load r1, acct->balance
2. add r1, amount2
3. store r1, acct->balance

Lost write
Recall: Multiprogramming

Scheduler can run threads in any order:

And with multiple cores or SMT

- Even more interleaving
- Could actually be running at the same time
Race Condition Examples (1)

What are the possible values of $x$ below?

Initially $x = y = 0$

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$</td>
<td>$y = 2$</td>
</tr>
</tbody>
</table>

Must be 1. Thread B can't interfere

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = y + 1$</td>
<td>$y = 2$</td>
</tr>
<tr>
<td></td>
<td>$y = y \times 2$</td>
</tr>
</tbody>
</table>

"Race condition": Thread A races against Thread B

1 or 3 or 5 (non-deterministic)
Race Condition Examples (2)

What are the possible values of \( x \) below?

Initially \( x = y = 0 \)

\[
\begin{array}{ll}
\text{Thread A} & \text{Thread B} \\
\hline
x = 1 & x = 2 \\
\end{array}
\]

1 or 2 (non-deterministic)

Why not 3?

- Couldn't \( x \) be written a bit at a time?
Atomic Operations

Definition: *an operation that runs to completion or not at all*
  - Need some to allow threads to work together

Example: loading or storing words
  - This is why 3 is not possible on most machines

Some *instructions* not atomic
  - e.g. double-precision floating point store
    (many platforms)
Roommates Alice and Bob want to keep their fridge stocked with milk:

<table>
<thead>
<tr>
<th>Time</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>
Definitions (1)

**Mutual exclusion**: ensuring only one thread does a particular thing at a time (one thread excludes the others)

**Critical section**: code exactly one thread can execute at once

– Result of mutual exclusion
Definitions (2)

**Lock**: object only one thread can hold at a time
- **Provides** mutual exclusion
Too Much Milk: Correctness

Correctness properties:
- At most one person buys
- At least one person buys if needed
Too Much Milk: "Solution" 1

Idea: Leave a note
   – Place before buying (a "lock")
   – Remove after buying
   – Don't try buying if there's already a note

Leaving or checking note atomic (word load/store)

```java
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove Note;
    }
}
```
Too Much Milk: "Solution" 1

**Alice**

```java
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove Note;
    }
}
```

**Bob**

```java
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove Note;
    }
}
```

Bought milk twice!
Too Much Milk: "Solution" 2

```java
leave Note;
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
  }
}
remove Note;
```

There's always a note!
But never buy milk twice.
Too Much Milk: "Solution" 3

**Alice**

Leave note Alice

if (noMilk) {
    if (noNote Bob) {
        buy milk
    }
}
Remove note Alice

**Bob**

Leave note Bob

if (noMilk) {
    if (noNote Alice) {
        buy milk
    }
}
Remove note Bob

Idea: Label the notes so Alice doesn't read hers (and vice-versa)
Too Much Milk: "Solution" 3

**Alice**

Leave note Alice

if (noMilk) {

    if (noNote Bob) {
        buy milk
    }
}

Remove note Alice

**Bob**

Leave note Bob

if (noMilk) {

    if (noNote Alice) {
        buy milk
    }
}

Remove note Bob
Too Much Milk: Solution 4

Alice
Leave note Alice
while (note Bob) {
    do nothing
}
if (noMilk) {
    buy milk
}
Remove note Alice

Bob
Leave note Bob
if (noNote Alice) {
if (noMilk) {
    Buy milk
}
}
Remove note Bob

This is a correct solution.
Too Much Milk: Solution 4 Problems

Complexity
- Proving that it works is hard
- How do you add another thread?

Busy-waiting
- Alice *consumes CPU time to wait*
Too Much Milk: Lock Primitive

Lock implementation, two **atomic** operations:
- Lock.Acquire() – wait until lock is free; then grab
- Lock.Release() – Unlock, wakeup waiters

```java
MilkLock.Acquire()
if (noMilk) {
    Buy milk
}
MilkLock.Release()
```

Critical Section

Problem: How do we write the lock?
Break
Implementing Locks: Single Core

Idea: Context switches only happen if there's an interrupt

Solution: Disable interrupts
Naive Interrupt Enable/Disable

```c
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}
```

Problem: User can **hang the system**:

```c
Lock.Acquire()
while (1) {}
```

Problem: User might want to do IO

```c
Lock.Acquire()
Read from disk
/* waits for (disabled) interrupt! */
```
Implementing Locks: Single Core

int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone waiting) {
        take a thread off queue
    } else {
        Value = FREE;
    }
    enable interrupts;
}

Idea: disable interrupts for **mutual exclusion** on accesses to **value**
Reenabling interrupts when waiting

Before on queue?
   - Release might not wakeup!

After putting the thread on the wait queue
   - Woken up by release, then switches away

Want to put it after sleep(). But – how?

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
Reenabling interrupts when waiting

```c
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Solution: run_new_thread() should do it!

Part of returning from switch()
Priority Inversion

A problem with **locks** and **priority scheduling**

Suppose Job 3 acquired a shared lock

Then Job 1 wants to acquire it

It waits...

*Until after lower-priority job 2 completes*
Solution: Priority Donation

Job 1 is waiting for Job 3 to release the lock...

Job 1 should "donate" its priority to Job 3:

Until Job 1 can "use" its priority again
Summary: Scheduling

Scheduling
− Metrics: response time, throughput, fairness

First-come first-served

Round robin:
− Fixed time quanta, return to end of line
− Tradeoff in length of quantum (switching overhead/fairness)

Priority: run highest priority first
− Mechanism for many scheduling policies
Summary: Synchronization (1)

Concurrency useful for overlapping computation and I/O...

But correctness problem:
- Arbitrary interleavings
- Could access shared resources in bad state
- Solution: careful design (not practical to test)

Building block: atomic operations
- Single processor: enable/disable interrupts
Summary: Synchronization (2)

Locks and Critical Sections
  - Only one thread at a time

Later: other synchronization constructs
RR better than FCFS? (1)

Example: 10 jobs, 100 unit CPU burst, $Q = 1$

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>

Average response time much worse under RR — *Without context switches!*
RR better than FCFS? (2)

<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 1/4</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 = 5</td>
<td>82</td>
<td>20</td>
<td>85</td>
<td>58</td>
<td>61 1/4</td>
</tr>
<tr>
<td>Q = 8</td>
<td>80</td>
<td>8</td>
<td>85</td>
<td>56</td>
<td>57 1/4</td>
</tr>
<tr>
<td>Q = 10</td>
<td>82</td>
<td>10</td>
<td>85</td>
<td>68</td>
<td>61 1/4</td>
</tr>
<tr>
<td>Q = 20</td>
<td>72</td>
<td>20</td>
<td>85</td>
<td>88</td>
<td>66 1/4</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 3/4</td>
</tr>
</tbody>
</table>
With a little help from a time machine

Goal: best FCFS

No preemption: **Shortest Job First (SJF)**
- Run whichever job has the least amount of work

Preemption: **Shortest Remaining Time First (SRTF)**
- New job (completed IO?) with shorter time replaces running one
With a little help from a time machine

**SJF/SRTF:** provably optimal (at response time)
- SJF among all non-preemptive schedulers
- SRTF among all preemptive schedulers

Key idea: remove convoy effect
- Short jobs *ahead* of long ones