Recall: Which scheduler should you use?

I care more than anything else about...

- **CPU Throughput**: First-Come First-Served
- **Average Response Time**: SRTF approximation
- **I/O Throughput**: SRTF approximation
- **Fairness – long-term CPU**: something like Linux CFS
- **Fairness – wait time for CPU**: something like RR
- **Meeting deadlines**: Earliest Deadline First
- **Favoring important users**: Strict Priority
Recall: Locks

Alice

doHomework()
watchTV()
MilkLock.Acquire()
if (noMilk) {
    buy milk
}
MilkLock.Release()

Bob

doHomework()
watchTV()
start MilkLock.Acquire()

buy milk

finish MilkLock.Acquire()
if (noMilk) {
    buy milk
}
MilkLock.Release()
Recall: Locks

Alice

doHomework()
watchTV()
MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()

Bob

doHomework()
watchTV()
start MilkLock.Acquire()
finish MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()

Critical section: only one thread can enter at a time
Recall: Locks

Alice

doHomework()
watchTV()
MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()

Bob

"Holding" lock section doesn't prevent context switches. Just stops other threads from acquiring it.
doHomework()
watchTV()
start MilkLock.Acquire()

buy milk

finish MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()
Recall: Locks

Alice

doHomework()
watchTV()
MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()

Bob

doHomework()
watchTV()
start MilkLock.Acquire()
finish MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()

Waiting threads don't consume processor time

Put back on run queue

Place self on wait queue
Recall: Disabling Interrupts

Critical sections in the kernel

On a single processor, nothing else can run
  – Primitive critical section!

Build other properties of lock on top of this
  – not excluding *everything else*
  – putting thread to sleep and letting other threads run
Recall: 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**
Multiprocessors and Locks

Disabling interrupts doesn't prevent other processor from doing anything

Solution: Hardware support for \textit{atomic operations}
Recall: 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

Some **instructions not atomic**

- e.g. double-precision floating point store (many platforms)
Outline

Hardware support for locks and spin locks

Semaphores

Monitors
Outline

Hardware support for locks and spin locks
Semaphores
Monitors
Atomic Read/Modify/Write

Recall: atomic load/store not good enough

Hardware instructions (or instruction sequences) that atomically read a value from (shared) memory and write a new value

Hardware responsible for making work in spite of caches
Read-Modify-Write Instructions

• test&set (&address) { /* most architectures */
  result = M[address];
  M[address] = 1;
  return result;
}

• swap (&address, register) { /* x86 */
  temp = M[address];
  M[address] = register;
  register = temp;
}

• compare&swap (&address, reg1, reg2) { /* 68000, x86-64 */
  if (reg1 == M[address]) {
    M[address] = reg2;
    return success;
  } else {
    return failure;
  }
}

• load-linked&store conditional(&address) {
  /* MIPS R4000, alpha */
  loop:
    ll r1, M[address];
    movi r2, 1; /* Can do arbitrary comp */
    sc r2, M[address];
    beqz r2, loop;
Locks with test&set

int value = 0; // Free

Acquire() {
    while (test&set(value)) {
        // do nothing
    }
}

Release() {
    value = 0;
}

Lock free: test&set reads 0, set value to 1, returns 0 (old value) → acquire finishes

Lock not free: test&set reads 1, sets value to 1, returns 1 (old value) → acquire **waits**
Locks with test&set: Busy waiting

int value = 0; // Free

Acquire() {
    while (test&set(value)) {
        // do nothing
    }
}

Release() {
    value = 0;
}

**Busy-waiting**: consumes CPU time while waiting
- Keeps other threads from using CPU
- Maybe even the thread holding the lock!

These are called "**spin locks**" (because they spin while busy)
Communicating Between Cores

Shared Memory Bus

Cache  Cache  Cache  Memory

CPU 1  CPU 2  CPU 3
Coherency

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>10</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>10</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>10</td>
</tr>
<tr>
<td>0x8</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>10</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td>0x8</td>
<td>30</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Coherency: Reading Normally

Message: What's 0xF0

Message: 0xF0 is 78.
Coherency

What if CPU 1 wants to write to 0x0?
Coherency: Invalidate

Message: Get rid of 0x0.

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>11</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>XX</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>XX</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>10</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td>0x8</td>
<td>30</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Coherency: Invalidate

Memory not updated yet – write-back policy.
Coherency: Snooping

What if CPU 2 wants to read 0x0 again?
Coherency: Snooping

What if CPU 2 wants to read 0x0 again?

Message: What's 0x0?

Message: 0x0 is 11.
Coherency: States

Caches need to remember if anyone else needs a copy of their items.

Simple protocol – four states:

- **Modified** – My value is more recent than memory, and I'm the only one who has it.
- **Owned** – My value is more recent than memory, but other processors have it.
- **Shared** – My value is unmodified (from owner or memory) and most recent.
- **Invalid** – My value is junk; don't use it.
Coherency: Invalidate

Message: Get rid of 0x0.

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>11</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Modified

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>XX</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Invalid

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>XX</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td>0x8</td>
<td>30</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Invalid

Cache

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>10</td>
</tr>
<tr>
<td>0x4</td>
<td>20</td>
</tr>
<tr>
<td>0x8</td>
<td>30</td>
</tr>
<tr>
<td>0xC</td>
<td>40</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Memory
Coherency: Snooping

Message: What's 0x0?

Message: 0x0 is 11.

Addr  Val
0x0  11
0xC  40
...

Owned

Cache

Addr  Val
0x0  11
0x4  20
...

Shared

Cache

Addr  Val
0x0  0
0x8  30
...

Invalid

Cache

Addr  Val
0x0  10
0x4  20
0xC  40
...

Memory
A Note on Memory Traffic

test&set requires communication!
  - Local cache needs to "own" the memory address to write to it

Means while(test&set(value)); spams the memory bus!
  - Two processors waiting: ownership flips back and forth
  - Hurts the performance of all processors
Test and Test and Set

Solution to memory traffic problem:

```c
int mylock = 0; // Free
Acquire() {
    do {
        while(mylock);   // Wait until might be free
    } while(test&set(&mylock)); // exit if get lock
}

Release() {
    mylock = 0;
}

while (mylock) stays in local cache (shared state)
  - waits for cache invalidation from lock holder
```
Recall: Disable Interrupts

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`
Locks with test&set

Use "spinlock" to build better locks

- Like we use disabling interrupts to build "proper" locks

```c
int guard = 0;
int value = FREE;

Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread();
    } else {
        value = BUSY;
        guard = 0;
    }
}

Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

Same problem with releasing lock as enabling interrupts before → need help from scheduler or spurious yield
Queues with Compare\&Swap

```c
compare\&swap (&address, reg1, reg2) {
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    } else {
        return failure;
    }
}
```

```c
addToQueue(&object) {
    do {
        // repeat until no conflict
        ld r1, M[root]  // Get ptr to current head
        st r1, M[object]  // Save link in new object
    } until (compare\&swap(&root,r1,object));
}
```
Outline

Hardware support for locks and spin locks

Semaphores

Monitors
Semaphores

Generalized lock

**Definition:** has a non-negative integer value and two operations:

- **P()** or **down** or **wait**: *atomically* wait for semaphore to become positive, then decrement it by 1
- **V()** or **up** or **signal**: *atomically* increment semaphore by 1 (waking up a waiting P() thread)

P, V from Dutch: *proberen* (test), *verhogen* (increment)
Semaphore Like Integers But...

**Cannot read/write** value directly
- Down (P)/up (V) only
- Exception: initialization

**Never negative** – something waits instead
- Two down operations can't go below 0 → some thread "wins"
Railway Analogy

Value=2
Simple Semaphore Patterns

**Mutual exclusion**: Same as lock (earlier)
  
  "Binary semaphore"

  Initial value of semaphore = 1
  
  semaphore.P();
  // Critical section goes here
  semaphore.V();

**Signaling other threads, e.g. ThreadJoin:**

  Initial value of semaphore = 0
  
  ThreadJoin() {
      semaphore.P();
  }
  ThreadFinish() {
      semaphore.V();
  }
Intuition for Semaphores

What do you need to wait for?
- Example: critical section to be finished
- Example: queue to be non-empty
- Example: array to have space for new items

What can you count that will be 0 when you need to wait?
- Ex: # of threads that can start critical section now
- Ex: # of items in queue
- Ex: # of empty spaces in array

Then: make sure semaphore operations maintain count
Higher-Level Synchronization

Want to make synchronization convenient, correct Semaphores first solution (1963)

First-class support for waiting for a condition to become true
Example: Producer/Consumer

Shared buffer (queue) – fixed size
- Producer inserts items
- Consumer removes items

Producer/consumer don't need to work in lockstep

Example: C compiler
- preprocessor → compiler → assembler → linker
Producer/Consumer Correctness

Scheduling constraints:
- Consumer waits for producer if buffer empty
- Producer waits for consumer if buffer full

Mutual exclusion constraint:
- Only one thread manipulates buffer at a time

One semaphore per constraint:
- Semaphore fullBuffers; // consumer's constraint
- Semaphore emptyBuffers; // producer's constraint
- Semaphore mutex; // mutual exclusion
Producer/Consumer Code

Semaphore fullBuffer = 0; // Initially, buffer empty
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 data.
}

Consumer() {
    fullBuffers.P(); // Check if there's an item
    mutex.P(); // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
Producer/Consumer Code

Semaphore fullBuffer = 0;  // Initially, buffer empty
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 data.
}

Consumer() {
    fullBuffers.P();  // Check if there's an item
    mutex.P();  // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();  // tell producer need more
    return item;
}
Producer/Consumer Code

Semaphore fullBuffer = 0; // Initially, buffer empty
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 data.
}

Can we do:
mutex.P() instead?

Consumer() {
    fullBuffers.P(); // Check if there's an item
    mutex.P(); // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
Producer/Consumer Code

Semaphore fullBuffer = 0; // Initially, buffer empty
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 data.
}

Consumer() {
    fullBuffers.P(); // Check if there's an item
    mutex.P(); // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // Tell producers there is more data.
    return item;
}
Producer/Consumer Code

Semaphore fullBuffer = 0; // Initially, buffer empty
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 data.
}

Consumer() {
    fullBuffers.P(); // Check if there's an item
    mutex.P(); // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}

Can we do:
emptyBuffers.V()
mutex.V()

instead?

Still correct, possibly less efficient.
Producer/Consumer: Discussion

Producer:
\[\text{emptyBuffer.P()},\ \text{fullBuffer.V()}\]

Consumer:
\[\text{fullBuffer.P()},\ \text{emptyBuffer.V()}\]

Two consumers or producers?
  - Same code!
Problems with Semaphores

Our textbook doesn't like semaphore:

- "Our view is that programming with locks and condition variables is superior to programming with semaphores."

Arguments:

Clearer to have separate constructs for
- waiting for a condition to become true and
- only allowing one thread to manipulate something at a time

Need to make sure some thread calls P() for every V()
- Other interfaces let you be sloppier (later)
Outline

Hardware support for locks and spin locks

Semaphores

Monitors
Monitors and Condition Variable

**Locks** for mutual exclusion

**Condition variables** for waiting

A **monitor** is a lock and zero or more condition variables with some associated data and operations

- Java provides this natively
- POSIX threads: provides **locks** and **condvars**, build your own
Monitor with Condition Variables

**Lock**: protects access to shared data
- **Rule**: always acquire lock while accessing

Queue of threads waiting to enter monitor
Condition variables: queue of threads waiting for something to become true inside a critical section

- **atomically** release the lock and start waiting (why?)
- another thread in the monitor will signal them
- the "something" is a function of the shared data
Condition Variables

Condition variable: queue of threads waiting inside a critical section

Operations:

- \texttt{Wait}(&lock): Atomically release lock and go to sleep. Re-acquire lock before returning.
- \texttt{Signal}(): Wake up one waiter (if there is one)
- \texttt{Broadcast}(): Wake up all waiters

Rule: Hold lock when using condition variable
Monitor Example: Queue

Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal a waiter, if any
    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);
}
Why the loop? (1)

```java
while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
}
```

and **not:**

```java
if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
}
```

When a thread is woken up by `signal()`, it is just put on the ready queue – it **might not reacquire the lock immediately!**

- Another thread could "sneak in" and empty the queue
- Consequence: need a loop
Why the loop? (2)

Couldn't we "hand-off" the lock to the signaled thread so nothing can sneak in?
- Yes. Called Hoare-style monitors.
- What many textbooks describe

Most OSs implement Mesa-style monitors
- Allow other threads to "sneak in"
- Much easier to implement
- Even easier if you allow "spurious wakeups" (returning when nothing signaled in rare cases)
  - POSIX does this
Comparing High-Level Synchronization

Semaphores can implement locks:
  - `Acquire() { semaphore.P() }`
  - `Release() { semaphore.V() }`

Monitors are a superset of locks.

Can monitors implement semaphores?
Semaphores with Monitors

Lock lock;
int Count = initial value of semaphore;
CondVar atOne;

P() {
    lock.Acquire();
    while (count == 0) {
        atOne.wait(&lock);
    }
    count--;
    lock.Release()
}

V() {
    lock.Acquire();
    count++;
    if (count == 1) {
        atOne.signal();
    }
    lock.Release()
}
Comparing High-Level Synchronization

Semaphores can implement locks:

- Acquire() { semaphore.P() }
- Release() { semaphore.V() }

Can monitors implement semaphores? Yes.

Can semaphores implement monitors?
CVs with Semaphores: Attempt 1

Attempt 1:

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

Problem:
- `Wait()` signals non-waiting threads (in the future)
- What about `broadcast()`?
CVs with Semaphores: Attempt 2

Attempt 2 to construct CV from semaphore:

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

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

Problem: "is queue empty" isn't a semaphore op

There is actually a solution
- See textbook (for Hoare scheduling)
CVs with Semaphores:

Is actually a solution!

Trick: queue of semaphores
  – Protected by semaphore acting as lock
  – Each waiting thread has a semaphore
  – Call V() on waiter's semaphore and remove it from queue
Comparing High-Level Synchronization

Semaphores can implement locks:
- \texttt{Acquire()} \{ \texttt{semaphore.P()} \}
- \texttt{Release()} \{ \texttt{semaphore.V()} \}

Monitors are a superset of locks.

Can monitors implement semaphores? Yes.

Can semaphores implement monitors? Yes.
Summary: Lock Implementation

Hardware support for locks:
- **interrupt disabling** – suitable for single-processor
- **atomic read/modify/write** – spinlocks for multiproc

Blocking threads nicely
- Move to *queue for each lock*
- Run new thread atomically (no chance to not wake up)
Summary: High-Level Sync

Semaphores – integers with restricted interface
- P() / down: Wait until non-zero then decrement
- V() / up: Increment and wake sleeping task

Monitors: Lock + Condition Variables
- Shared state determines when to wait
- Condition variables to wait "within critical section" on shared state changing
- Wait(), Signal(), Broadcast()