Synchronization

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

*If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.*
Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic
Three Reasons: 1/4

- Program execution depends on the possible interleaving of threads’ access to shared data.
  - You learned in Concurrent Computing that this is the main cause of race conditions.
  - Depending on the execution order, the result of the shared data may become unpredictable.
Three Reasons: 2/4

- Program execution can be nondeterministic.
- Interrupts can happen any time and anywhere. As a result, a thread can be switched out of the CPU by the scheduler in an unpredictable way.
  - A multithreaded program can potentially have different interleaving execution every time when it runs.
  - Jim Gray in his 1998 ACM Turing Award talk coined the term *Heisenbugs* for bugs that disappear or change behavior when you try to examine them. *Bohr bugs* are deterministic and general much easier to diagnose.
Three Reasons: 3/4

- Compilers and processor hardware can reorder instructions.
- Modern compilers and hardware reorder instructions to improve performance.
- For higher-level language statements that are “independent” of each other, compilers are free to order the execution of these statements. Only those statements that are dependent of each other are executed in the needed order. For example, \( c = a + b; \ x = c \times 100; \) will be executed in the specified order. However, \( c = a + b; \ x = m \times n; \) are not guaranteed to be executed in the specified order.
Because these two statements are independent of each other, compiler or hardware may execute the second statement prior to the first.

Suppose the order or the two statements in Thread 1 are changed. Before the value of $p$ is obtained properly, Thread 2 could start its execution. In this case, Thread 2 could use an unexpected value of $p$ to compute $q$. 
Too Much Milk Example: 1/6

Alice and Bob are sharing an apartment. Alice arrives home in the afternoon, looks in the fridge and finds that there is no milk. So, she leaves for the grocery to buy milk.

After she leaves, Bob arrives, he also finds that there is no milk and goes to buy milk.

At the end both buy milk and end up with too much milk.

<table>
<thead>
<tr>
<th>Time</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:00</td>
<td>Arrive home</td>
<td></td>
</tr>
<tr>
<td>5:05</td>
<td>Look in fridge; no milk</td>
<td></td>
</tr>
<tr>
<td>5:10</td>
<td>Leave for grocery</td>
<td></td>
</tr>
<tr>
<td>5:15</td>
<td></td>
<td>Arrive home</td>
</tr>
<tr>
<td>5:20</td>
<td></td>
<td>Look in fridge; no milk</td>
</tr>
<tr>
<td>5:25</td>
<td>Buy milk</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>5:30</td>
<td>Arrive home; put milk in fridge</td>
<td>Buy milk</td>
</tr>
<tr>
<td>5:40</td>
<td></td>
<td>Arrive home; put milk in fridge</td>
</tr>
</tbody>
</table>

Too much milk
Alice and Bob are looking for a solution to ensure that:

- Only one person buys milk, when there is no milk.
- Someone always buys milk, when there is no milk.

They will communicate by leaving (signed) notes on the door of the fridge. **Note that they do not see each other.**

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (no note) then</td>
<td>if (no note) then</td>
</tr>
<tr>
<td>if (no milk) then</td>
<td>if (no milk) then</td>
</tr>
<tr>
<td>leave note</td>
<td>leave note</td>
</tr>
<tr>
<td>buy milk</td>
<td>buy milk</td>
</tr>
<tr>
<td>remove note</td>
<td>remove note</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
</tbody>
</table>

What if Alice and Bob come home at the same time?
Too Much Milk Example: 3/6

- Each of Alice and Bob first leaves note, checks the other’s note. If no note, checks whether there is milk. If there is no milk, then busy milk. Finally, remove his/her own note.
- **Note that they do not see each other.**

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note Alice</td>
<td>leave note Bob</td>
</tr>
<tr>
<td>if (no note Bob) then</td>
<td></td>
</tr>
<tr>
<td>if (no milk) then</td>
<td></td>
</tr>
<tr>
<td>buy milk</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>remove note Alice</td>
<td>remove note Bob</td>
</tr>
</tbody>
</table>

What if Alice and Bob come home and leave note at the same time? **No milk!**
Too Much Milk Example: 4/6

Bob leaves note and repeatedly check Alice’s note until Alice’s note is not on fridge.

Once Bob finds Alice’s note is not there, he check for milk. If there is no milk, Bob buys milk.

Note that they do not see each other.

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note Alice</td>
<td>leave note Bob</td>
</tr>
<tr>
<td>if (no note Bob) then</td>
<td>while (note Alice) do</td>
</tr>
<tr>
<td>if (no milk) then</td>
<td>nothing;</td>
</tr>
<tr>
<td>buy milk</td>
<td>if (no milk) then</td>
</tr>
<tr>
<td>end if</td>
<td>buy milk</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
<tr>
<td>remove note Alice</td>
<td>remove note Bob</td>
</tr>
</tbody>
</table>

We have to assume: between the time Alice removes her note, and the time she leaves a new note next time, Bob must be able to find out that Alice’s note has been removed. Without this assume, they never buy milk. Find an execution for this scenario.
Too Much Milk Example: 5/6

- The fridge has four slots for posting notes. Alice uses $A_1$ and $A_2$, and Bob uses $B_1$ and $B_2$.

- If Alice (resp., Bob) finds that there is no note labelled $B_1$ (resp., $A_1$) on the fridge’s door, then it is Alice (resp., Bob) responsibility to buy milk.

- Otherwise, when both $A_1$ and $B_1$ are present, a decision is made according to the notes $A_2$ and $B_2$.

- If both $A_2$ and $B_2$ are present or if neither of them is present than it is Bob’s responsibility to by milk.

- Otherwise, it is Alice’s responsibility.

The six possible configurations of the notes on the fridge’s door:
Too Much Milk Example: 6/6

- The fridge has four slots for posting notes. Alice uses $A_1$ and $A_2$, and Bob uses $B_1$ and $B_2$.
- **This is a correct solution. Study and prove it.**

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note $A_1$</td>
<td>leave note $B_1$</td>
</tr>
<tr>
<td>if (B$_2$) then</td>
<td>if (no $A_2$) do</td>
</tr>
<tr>
<td>leave note $A_2$</td>
<td>leave note $B_2$</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>remove note $A_2$</td>
<td>remove note $B_2$</td>
</tr>
<tr>
<td>end if /* Alice gives priority */</td>
<td>end if /* giving to Alice */</td>
</tr>
<tr>
<td>/* to Bob in buying milk*/</td>
<td></td>
</tr>
<tr>
<td>while $B_1$ and</td>
<td>while $A_1$ and</td>
</tr>
<tr>
<td>((A$_2$ and B$_2$)) or</td>
<td>((A$_2$ and no B$_2$)) or</td>
</tr>
<tr>
<td>(no A$_2$ and no B$_2$)) do</td>
<td>(no A$_2$ and B$_2$)) do</td>
</tr>
<tr>
<td>nothing;</td>
<td>nothing;</td>
</tr>
<tr>
<td>if (no milk) then</td>
<td>if (no milk) then</td>
</tr>
<tr>
<td>buy milk</td>
<td>buy milk</td>
</tr>
<tr>
<td>end if</td>
<td>end if</td>
</tr>
<tr>
<td>remove note A1</td>
<td>remove note B1</td>
</tr>
</tbody>
</table>
Lessons

- Solution is complicated
  - “obvious” code often has bugs
- You may replace Alice and Bob with two computers and the fridge with a file.
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson’s algorithm
Definitions

**Race condition**: output of multiple threaded program that manipulates a shared resource concurrently depends on the order of operations among threads

**Mutual exclusion**: only one thread does a particular thing at a time
- **Critical section**: piece of code that only one thread can execute at once
- **Lock**: prevent someone from doing something
  - Lock before entering critical section, before accessing shared data
  - Unlock when leaving, after done accessing shared data
  - Wait if locked (all synchronization involves waiting!)
**Roadmap**

- **Concurrent apps built using shared objects.**
- **Shared objects built using synchronization variables**
- **Synchronization variables (often) built using atomic instructions**

### Shared Objects
- Bounded Buffer
- Barrier

### Synchronization Variables
- Semaphores
- Locks
- Condition Variables

### Atomic Instructions
- Test-and-Set
- Interrupt Disable

### Hardware
- Atomic and act as memory barrier

### Concurrent Applications
- Hardware Interrupts
- Multiple Processors
Locks

- `Lock::acquire`
  - Wait until lock is free, then take it
- `Lock::release`
  - Release lock, waking up anyone waiting for it

1. At most one lock holder at a time (Mutual exclusion)
2. If no one holding, acquire gets lock (Progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock
   - Need not be FIFO!

Properties of a solution to the “Critical Section” problem
Question: Why only Acquire/Release

Suppose we add a method to a lock, to ask if the lock is free. Suppose it returns true. Is the lock:

- Free?
- Busy?
- Don’t know?

when testing the return value

In security lingo, leads to a _TOCTOU_ error. “Time of check to time of use”
Lock Example: malloc/free

```c
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}

void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```
Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure.
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - **DO NOT** throw lock for someone else to release
- Never access shared data without lock
  - Danger!
- Don’t put shared objects on the stack. Why?
Will this code work? 1/2

p is a shared variable

1. if (p == NULL) {
2.     lock.acquire();
3.     if (p == NULL) {
4.         p = \texttt{newP}();
5.     }
6.     lock.release();
7. }
8. use p->field1

\texttt{newP}()\
{}
\{\
\quad p = \texttt{malloc(sizeof(p))};\
\quad p->field1 = ...\
\quad p->field2 = ...\
\quad return p;
\}

\texttt{p} is a shared variable
# Will this code work? 2/2

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if (p == NULL) {</td>
<td>1. if (p == NULL) {</td>
</tr>
<tr>
<td>2.     lock.acquire();</td>
<td>2.     lock.acquire();</td>
</tr>
<tr>
<td>3.     if (p == NULL) {</td>
<td>3.     if (p == NULL) {</td>
</tr>
<tr>
<td>4.     p = malloc(..);</td>
<td>4.     p = newP();</td>
</tr>
<tr>
<td></td>
<td>5. }</td>
</tr>
<tr>
<td></td>
<td>6.     lock.release();</td>
</tr>
<tr>
<td></td>
<td>7. }</td>
</tr>
<tr>
<td></td>
<td>8.     use p-&gt;field1</td>
</tr>
<tr>
<td>8. p-&gt;field1;</td>
<td></td>
</tr>
</tbody>
</table>

```c
lock.acquire();
if (p == NULL) {
    p = newP();
}
lock.release();
```

```c
lock.acquire();
if (p == NULL) {
    p = newP();
}
lock.release();
```
**Example Shared Object Using Locks**

**Bounded Buffer**

```c
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front%MAX];
        front++;
    }
    lock.release();
    return item;
}

tryput(item) {
    lock.acquire();
    if ((tail-front) < size) {
        buf[tail % MAX] = item;
        tail++;
    }
    lock.release();
}
```

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

- front < tail => data available
- tail = front + size => full
The Milk Problem Revisited

- Alice and Bob calls `BuyMilkIfNeeded()` to determine whether she or he should buy milk.
- Prove that
  - Only one person buys milk when there is no milk
  - Someone always buys milk when there is no milk

```java
BuyMilkIfNeeded() {
    lock.acquire();
    if (no milk) {
        buy milk;
    }
    lock.release();
}
```

Buying milk is mutually exclusive, because only one person should buy.

Both have to check “milk” in a mutually exclusive way.

If no milk, then buy it!

Can “buy milk” be moved outside of the critical section?
Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock

- Operations
  - **Wait** - atomically release lock and relinquish processor
    - Reacquire the lock when wakened
  - **Signal** - wake up a waiter, if any
  - **Broadcast** - wake up all waiters, if any

Not UNIX
wait() and signal()
Mesa vs. Hoare semantics

- **Mesa**
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor

- **Hoare**
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaler
  - Nested signals possible (i.e., cascading release)
Hoare vs. Mesa: 1/2

**Producer**
```
mutex::acquire();
if (count >= MAX)
    wait(notFull, mutex);
buf[count]='a';
count++;
signal(notEmpty);
Mutex::release();
```

**Consumer**
```
mutex::acquire();
if (count == 0)
    wait(notEmpty, mutex);
ch=buf[count];
count--;
signal(notFull);
Mutex::release;
```

Replace the *if* with a *while* for the Mesa type

What if the above code is run under the Mesa type? **Problem!!!**
## Hoare vs. Mesa: 2/2

**buffer size = 2**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_1$</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_2$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td>$P_3$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_4$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C_1$</td>
<td>2</td>
</tr>
<tr>
<td>Add 1 item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Add 1 item</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>acquire</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>wait</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>acquire</strong></td>
<td><strong>wait</strong></td>
<td><strong>acquire</strong></td>
<td></td>
<td></td>
<td><strong>acquire</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Take 1 item</strong></td>
<td><strong>signal</strong></td>
<td><strong>release</strong></td>
<td></td>
<td></td>
<td><strong>release</strong></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>Add 1 item</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>No space!</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Under Hoare, $P_3$ should immediately get the critical section.

By the time $P_3$ gets the monitor and runs again, the free spot has already been taken by $P_4$.

Under Mesa the signaler continues...
Condition Variable Design Pattern

methodThatWaits() {
    lock.acquire();
    // Read/write shared state

    while (!testSharedState()) {
        cv.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}

methodThatSignals() {
    lock.acquire();
    // Read/write shared state

    // If testSharedState now true
    cv.signal(&lock);

    // Read/write shared state
    lock.release();
}

Give up lock and reacquire!!

Assume signaler keeps lock. More later.
**Example: Bounded Buffer**

get() {
    lock.acquire();
    while (front == tail) {
        empty.wait(lock); /* Don’t know state here */
    } // Not empty; front != tail
    item = buf[front % MAX];
    front++;
    full.signal(lock); // Not full
    lock.release(); // Front <= tail
    return item;
}

put(item) {
    lock.acquire();
    while ((tail - front) == MAX) {
        full.wait(lock);
    } // Not full; tail != front+MAX

    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock); // Not empty
    lock.release(); // Front+MAX>=tail
}

Initially: front = tail = 0; MAX is buffer capacity
easy/full are condition variables

[X] [X] [ ] [ ] [ ]

Front = Tail => Empty
Tail = Front + SIZE => Full
Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - \( \text{front} \leq \text{tail} \)
  - \( \text{front} + \text{MAX} \geq \text{tail} \) (wraparound)

- These are also true on return from wait

- And at lock release

- Allows for proof of correctness

Otherwise, wrote to full buffer or read from empty buffer
methodThatWaits() {
  lock.acquire();
  // Pre-condition: State is consistent

  // Read/write shared state

  while (!testSharedState()) {
    cv.wait(&lock);
  }

  // WARNING: shared state may have changed!  But
  // testSharedState is TRUE  
  // and pre-condition is true
  // (just got the lock)

  // Read/write shared state
  lock.release();
}

methodThatSignals() {
  lock.acquire();
  // Pre-condition: State is consistent

  // Read/write shared state

  // If testSharedState is now true
  cv.signal(&lock);

  // NO WARNING: signal keeps lock

  // Read/write shared state
  lock.release();
}
Condition Variables

- **MUST** hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - **ALWAYS** hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
Condition Variables, cont’d

- When a thread is woken up from wait, it may or may not run immediately
  - \textbf{signal/broadcast} put thread on a waiting list to “re-enter” the critical section
  - When lock is released, anyone might acquire it

- \textbf{Wait \textit{MUST} be in a loop}
  
  ```
  while (needToWait()) {
    condition.Wait(lock);
  }
  ```

- \textbf{Simplifies implementation}
  - Of condition variables and locks
  - Of code that uses condition variables and locks
Design of Shared Objects

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
  - Grab lock on start to every method/procedure and release lock on finish
- If need to wait
  - while (needToWait()) { condition.Wait(lock); }
  - Do not assume when you wake up, signaler ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting
Use consistent structure

Always use locks and condition variables

Always acquire lock at beginning of procedure, release at end

Always hold lock when using a condition variable

Always wait in while loop

lock()
  . . . ops . . .
while (testState()) {sleep();}
  . . . ops . . .
release()

Still holding the lock!

lock()
  . . . ops . . .
release()
while (testState()) {sleep();}
lock()
  . . . ops . . .
release()

State may change between end of loop and lock()
Implementing Synchronization

**Approach 1**: Using memory load/store

- See too much milk solution/Peterson’s algorithm

**Approach 2:**

```cpp
Lock::acquire()
    { disable interrupts }
Lock::release()
    { enable interrupts }
```

**Thread 1**: `aLock.acquire();
    while (1==1);
aLock.release();`

Interrupts are disabled. How to regain control?

Only possible in kernel; Could never let a user process run with interrupts off!
Lock Implementation: Uniprocessor

Lock::acquire() {
    disableInterrupts(); /* mem barrier */
    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts(); /* mem barrier */
}

Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
Multiprocessor

- Interrupts turned off at individual processors
  - No instruction to turn them off on all processors simultaneously
  - Threads may be running on different processors

- Read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware

- Examples
  - Test and set, Compare and swap
  - Intel: xchg, lock prefix

- Any of these can be used for implementing locks and condition variables!
Spinlocks

A spinlock is a lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect the CPU scheduler and to implement locks

Spinlock::acquire() {
    while (testAndSet(&lockValue) == BUSY);
}

Spinlock::release() {
    lockValue = FREE;
    memorybarrier();
}

bool testAndSet(bool *flag){
    bool old=*flag;
    *flag=BUSY;
    return old;  // if FREE, return FREE
    // Next process through sees
    // and returns BUSY
}

Busy wait. Reasonable for short hold, e.g. < time for context switch

Memory operations before barrier guaranteed to be performed
What Thread Is Currently Running?

- Thread scheduler needs to find the TCB of the currently running thread
  - To suspend and switch to a new thread
  - To check if the current thread holds a lock before acquiring or releasing it

- On a uniprocessor, easy: just use a global

- On a multiprocessor, various methods:
  - Compiler dedicates a register (e.g., r31 points to TCB running on the this CPU; each CPU has its own r31)
  - If hardware has a special per-processor register, use it
  - Fixed-size stacks: put a pointer to the TCB at the bottom of its stack
    - Find it by masking the current stack pointer
Lock Implementation: Multiprocessor 1/2

Lock::acquire() {
    spinLock.acquire();
    /* Protects lock state */
    if (value == BUSY) {
        waiting.add(myTCB);

        scheduler.suspend(&spinLock);
    }
    else {
        value = BUSY;
    }
    spinLock.release();
}

Lock::release() {
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();

        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}

scheduler releases spinlock (next slide)
scheduler makes the new one ready (next slide)
To suspend a thread on a multiprocessor, we need to first disable interrupts to ensure the thread is not preempted while holding the ready list spinlock.

Now, it is safe to release the lock’s spinlock and switch to a new thread.

New running thread
An Execution Sequence

Three locks are involved: Lock, spinLock and schedSpinLock, all initialized to FREE

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Lock value</th>
<th>spinlock value</th>
<th>schedSpinLock value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock.acquire()</td>
<td>FREE</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>spinLock.acquire()</td>
<td>FREE</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>while (...)</td>
<td>FREE</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>if (value==BUSY)</td>
<td>FREE</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>value = BUSY</td>
<td>BUSY</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>spinLock.release();</td>
<td>BUSY</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
</tbody>
</table>

Thread 1 has Lock

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Lock value</th>
<th>spinlock value</th>
<th>schedSpinLock value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock.acquire()</td>
<td>BUSY</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>spinLock.acquire()</td>
<td>BUSY</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>while (...)</td>
<td>BUSY</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>if (value==BUSY)</td>
<td>BUSY</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>waiting.add(myTCB)</td>
<td>BUSY</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>scheduler.suspend(&amp;spinlock)</td>
<td>BUSY</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>disableinterrupts</td>
<td>BUSY</td>
<td>BUSY</td>
<td>FREE</td>
<td></td>
</tr>
<tr>
<td>schedSpinLock.acquire()</td>
<td>BUSY</td>
<td>BUSY</td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td>spinLock.release()</td>
<td>BUSY</td>
<td>FREE</td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td>myTCB-&gt;state = WAITING</td>
<td>BUSY</td>
<td>FREE</td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td>next = readyList.remove()</td>
<td>BUSY</td>
<td>FREE</td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td>thread_switch(myTCB, next)</td>
<td>BUSY</td>
<td>FREE</td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td>schedSpinLock.release()</td>
<td>BUSY</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
</tbody>
</table>

Other threads run … acquire, release ready list spinlock spinLock

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Lock value</th>
<th>spinlock value</th>
<th>schedSpinLock value</th>
</tr>
</thead>
<tbody>
<tr>
<td>myTCB-&gt;state = RUNNING</td>
<td>BUSY</td>
<td>FREE</td>
<td>BUSY</td>
<td></td>
</tr>
<tr>
<td>schedSpinLock.release()</td>
<td>BUSY</td>
<td>FREE</td>
<td>FREE</td>
<td></td>
</tr>
</tbody>
</table>

Other threads run and Thread 2 was switched out
Semaphores

- Semaphore has a non-negative integer value
  - \( \text{P}() \) atomically waits for value to become \( > 0 \), then decrements
  - \( \text{V}() \) atomically increments value (waking up waiter if needed)

- Semaphores are like integers except:
  - Only operations are \( \text{P} \) and \( \text{V} \)
  - Operations are atomic
    - If value is 1, two \( \text{P} \)'s will result in value 0 and one waiter

- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join
Semaphore Bounded Buffer

get()
{
    fullSlots.P();
    mutex.P();
    item = buf[front%MAX];
    front++;
    mutex.V();
    emptySlots.V();
    return item;
}

put(item)
{
    emptySlots.P();
    mutex.P();
    buf[last%MAX] = item;
    last++;
    mutex.V();
    fullSlots.V();
}

Initially: front = last = 0; MAX is buffer capacity
mutex = 1; emptySlots = MAX; fullSlots = 0;
Implementing Condition Variables using Semaphores: 1

```c
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}
signal() {
    semaphore.V();
}
```

**CV Wait():**
- Release lock;
- Wait for signal;
- Reacquire lock;

**CV Signal():**
- Awaken waiter, if there is one;
- Otherwise, nop;

Is this solution correct? No!

**What happened if a thread calls `signal()` and no one is waiting?**
With condition variables, if a thread calls `signal()` 100 times, when no one is waiting, the next `wait()` call will wait.
With the above code, the next 100 threads call `wait()` will return immediately!
Implementing Condition Variables using Semaphores: 2

```c
wait(lock)
{
    lock.release();
    semaphore.P();
    lock.acquire();
}
```

```c
signal()
{
    if (semaphore is not empty)
        semaphore.V();
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Semaphore</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(lock)</td>
<td></td>
<td>0</td>
<td>Thread 1 calls wait(lock)</td>
</tr>
<tr>
<td>release</td>
<td></td>
<td>0</td>
<td>Release the lock, switched out</td>
</tr>
<tr>
<td>lock</td>
<td></td>
<td>0</td>
<td>Thread 2 calls signal()</td>
</tr>
<tr>
<td>semaphore</td>
<td></td>
<td>0</td>
<td>Thread 2 found no waiting</td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td>0</td>
<td>Thread 2 return from signal()</td>
</tr>
<tr>
<td>P()</td>
<td></td>
<td>0</td>
<td>Thread 1 waits!</td>
</tr>
</tbody>
</table>

Thread 1 blocks, but should have been released

no way to access the internal of a semaphore
Implementing Condition Variables using Semaphores: 3

```
wait(lock)
{
    queue.Append(myTCB);
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal()
{
    if (!queue.Empty()) {
        semaphore.V();
    }
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait()</td>
<td></td>
<td>semaphore = 0</td>
<td></td>
</tr>
<tr>
<td>myTCB queued</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock released</td>
<td></td>
<td>lock is open</td>
<td></td>
</tr>
<tr>
<td>semaphore V()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>semaphore P()</td>
<td></td>
<td>semaphore = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 2 should release thread 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thread 2 releases thread 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>semaphore P()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>semaphore = 0</td>
<td>Thread 3 released</td>
</tr>
</tbody>
</table>

**Comment:**
- Because no one is waiting, this signal() should have no effect.
- But, this signal() has an impact on a later thread. Incorrect implementation.
wait(lock) {
    semaphore = new Semaphore(0); // each waiting threads
    // has its own semaphore
    queue.Append(semaphore); // queue of waiting threads
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    if (!queue.Empty()) {
        semaphore = queue.Remove();
        semaphore.V(); // wake up waiter associated
        // with semaphore
    }
}
Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in `sleep()`
The End