



CSE 451: Operating Systems, 26au

L8: Locks and Condition Variables

Instructor: Naomi Alterman

TAs:

Jonathan Trinh (Head)

Mahir Emran

Ted Guan

Jai Rungrachakanont

Soham Raut

Sagar Saby

Andrew Wood

Adapted from slides by Tom Anderson

Administrivia

- Lab 1 grades out!
- Due:
 - Lab 2.1 design doc @ 6p
 - Pset 1 @ midnight
- Out:
 - Pset 2, due 10/26

Two Threads Call Yield

Thread 1's instructions

"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state

return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

Thread 2's instructions

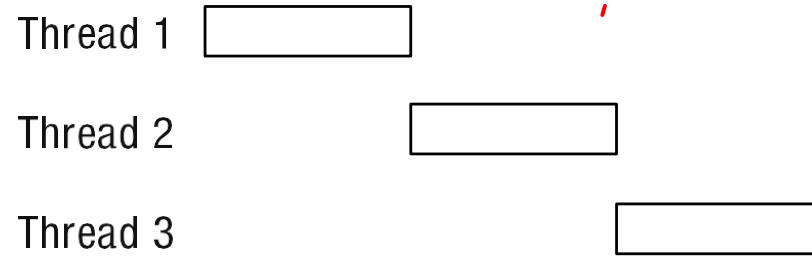
"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state

Processor's instructions

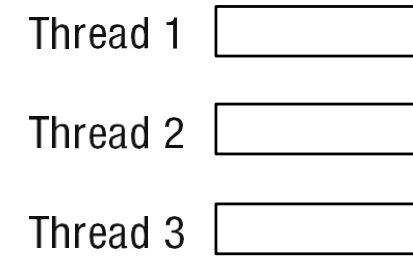
"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state
"return" from thread_switch
into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state
return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

Possible Executions

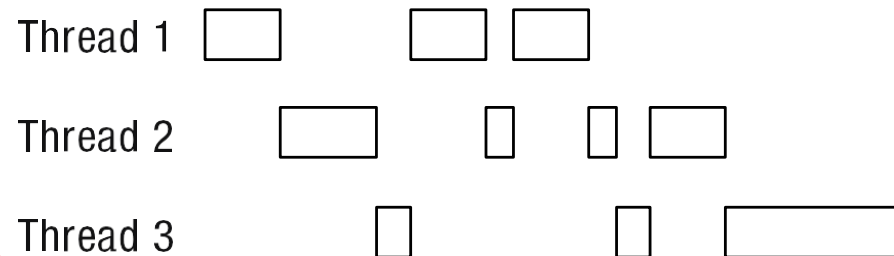
One Execution



Another Execution



Another Execution



Locks

A solution to concurrency tragedy

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 (safety)
- 2. If no one holding, acquire gets lock (liveness)
- 3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (liveness)

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
milk_lock.acquire();  
if (!milk) {  
    buy milk  
}  
milk_lock.release();
```

Lock Example: Malloc/Free

** critical
section*

```
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 Healthfully

- Lock is initially free
- • Always acquire before accessing shared data structure
 - Usually the beginning of a function
- Always release after finishing with shared data
 - • Usually the end of a function
 - Only the thread holding the lock can release it
 - (Really) bad idea to acquire a lock in one function and release it in another
- Never access shared data without lock
 - Ever ever!

Danger!! Bad! Double Checked Locking

```
if (p == NULL) {  
    lock.acquire();  
    if (p == NULL) {  
        p = newP();  
    }  
    lock.release();  
}  
use p->field1
```

```
newP() {  
    tmp = malloc(sizeof(p));  
    tmp->field1 = ...  
    tmp->field2 = ...  
    return tmp;  
}
```

Single Checked Locking

```
lock.acquire();  
if (p == NULL) {  
    p = newP();  
}  
lock.release();  
use p->field1
```

```
newP() {  
    tmp = malloc(sizeof(p));  
    tmp->field1 = ...  
    tmp->field2 = ...  
    return tmp;  
}
```

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?

Condition Variables

Dealing with liveness issues

Motivation

- Locks let us to wait for a resource to become available without busy-waiting
- But if we're waiting for a resource to *change*, that introduces its own form of busy-waiting
 - Lots of acquire->poll->release cycles crowd out other threads that could introduce the change we're waiting for!

Example: Ring Buffer

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

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

not empty?



If tryget() returns NULL, do we know the buffer is empty?

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

is there room?



If we poll tryget() in a loop, what happens to a thread calling tryput()?

Solution

- **Condition variables:** a synchronization primitive for going to sleep in a critical section until desired conditions have been met
- Operations (all atomic, all while holding a specific lock):
 - **Wait:**
 - Releases the lock, sits in a set of threads waiting for condition to occur
 - When the time comes, will be woken up and get the lock back
 - **Signal:** Releases the lock, ready one waiting thread to wake up
 - **Broadcast:** Release the lock, ready *all* waiting threads to wake up
 - but note: not *all at once*



Condition Variable Design Pattern: *Monitor*

```
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 is now true  
    cv.signal(&lock);  
  
    // Read/write shared state  
    lock.release();  
}
```

Important! Test for
“spurious wakeups”

Ring Buffer with Condition Variable

```
// Initially: front = tail = 0; lock = FREE; MAX is buffer capacity  
// empty and full are condition variables
```

```
get() {  
    lock.acquire();  
    while (front == tail) {  
        empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

empty?

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
}
```

Full?

Pre/Post Conditions

- Outlining our assumptions about the state of a concurrent program going into or out of a critical section helps both our informal design process and also allows for formal proofs of correctness
- For our example: what is state of the ring buffer on lock acquisition?
 - $\text{front} \leq \text{tail}$
 - $\text{tail} - \text{front} \leq \text{MAX}$
- These are also true on return from wait
- And at lock release

Pre/Post Conditions (in pseudocode)

```
methodThatWaits() {  
    lock.acquire();  
    // Pre-condition: State is  
    // consistent  
    while (!testSharedState()) {  
        cv.wait(&lock);  
    }  
    // WARNING: shared state may  
    // have changed! But  
    // testSharedState is TRUE and  
    // pre-condition is true  
    lock.release();  
}
```

```
methodThatSignals() {  
    lock.acquire();  
    // Pre-condition: State is consistent  
  
    // If testSharedState is now true:  
    cv.signal(&lock);  
  
    // signal keeps lock  
    lock.release();  
}
```

Ring Buffer: Example Execution

```
A get() {  
  lock.acquire();  
  while (front == tail) {  
    empty.wait(&lock);  
  }  
  item = buf[front % MAX];  
  front++;  
  full.signal(&lock);  
  lock.release();  
  return item;  
}
```

```
B put(item) {  
  lock.acquire();  
  while ((tail - front) == MAX) {  
    full.wait(&lock);  
  }  
  buf[tail % MAX] = item;  
  tail++;  
  empty.signal(&lock);  
  lock.release();  
}
```

tail	0
front	0
lock	FREE
ready list	A, B
lock wait queue	
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
  A → lock.acquire();  
  while (front == tail) {  
    empty.wait(&lock);  
  }  
  item = buf[front % MAX];  
  front++;  
  full.signal(&lock);  
  lock.release();  
  return item;  
}
```

```
put(item) {  
  B → lock.acquire();  
  while ((tail - front) == MAX) {  
    full.wait(&lock);  
  }  
  buf[tail % MAX] = item;  
  tail++;  
  empty.signal(&lock);  
  lock.release();  
}
```

tail	0
front	0
lock	HELD
ready list	A
lock wait queue	B
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
A → while (front == tail) {  
        empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

```
put(item) {  
B → lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
}
```

tail	0
front	0
lock	HELD
ready list	A
lock wait queue	B
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
  lock.acquire();  
  while (front == tail) {  
A →    empty.wait(&lock);  
  }  
  item = buf[front % MAX];  
  front++;  
  full.signal(&lock);  
  lock.release();  
  return item;  
}
```

```
put(item) {  
B →  lock.acquire();  
  while ((tail - front) == MAX) {  
    full.wait(&lock);  
  }  
  buf[tail % MAX] = item;  
  tail++;  
  empty.signal(&lock);  
  lock.release();  
}
```

tail	0
front	0
lock	FREE
ready list	B
lock wait queue	
empty queue	A
full queue	

Ring Buffer: Example Execution

```
get() {  
  lock.acquire();  
  while (front == tail) {  
A →    empty.wait(&lock);  
  }  
  item = buf[front % MAX];  
  front++;  
  full.signal(&lock);  
  lock.release();  
  return item;  
}
```

```
put(item) {  
  B → lock.acquire();  
  while ((tail - front) == MAX) {  
    full.wait(&lock);  
  }  
  buf[tail % MAX] = item;  
  tail++;  
  empty.signal(&lock);  
  lock.release();  
}
```

tail	0
front	0
lock	HELD
ready list	B
lock wait queue	
empty queue	A
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
A →    empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
B →    buf[tail % MAX] = item;  
        tail++;  
        empty.signal(&lock);  
        lock.release();  
}
```

tail	0
front	0
lock	HELD
ready list	B
lock wait queue	
empty queue	A
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
A →    empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
B →    empty.signal(&lock);  
    lock.release();  
}
```

tail	1
front	0
lock	HELD
ready list	B
lock wait queue	
empty queue	A
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
A →    empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
B →    empty.signal(&lock);  
    lock.release();  
}
```

tail	1
front	0
lock	HELD
ready list	B
lock wait queue	A
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
A →    empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
B → }
```

tail	1
front	0
lock	FREE
ready list	A, B
lock wait queue	
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
  lock.acquire();  
  A → while (front == tail) {  
    empty.wait(&lock);  
  }  
  item = buf[front % MAX];  
  front++;  
  full.signal(&lock);  
  lock.release();  
  return item;  
}
```

```
put(item) {  
  lock.acquire();  
  while ((tail - front) == MAX) {  
    full.wait(&lock);  
  }  
  buf[tail % MAX] = item;  
  tail++;  
  empty.signal(&lock);  
  lock.release();  
}
```

tail	1
front	0
lock	HELD
ready list	A
lock wait queue	
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
        empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

A

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
}
```

tail	1
front	0
lock	HELD
ready list	A
lock wait queue	
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
        empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

A



```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
}
```

tail	1
front	1
lock	HELD
ready list	A
lock wait queue	
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
        empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    return item;  
}
```

A



```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
}
```

tail	1
front	1
lock	FREE
ready list	A
lock wait queue	
empty queue	
full queue	

Ring Buffer: Example Execution

```
get() {  
    lock.acquire();  
    while (front == tail) {  
        empty.wait(&lock);  
    }  
    item = buf[front % MAX];  
    front++;  
    full.signal(&lock);  
    lock.release();  
    A → return item;  
}
```

```
put(item) {  
    lock.acquire();  
    while ((tail - front) == MAX) {  
        full.wait(&lock);  
    }  
    buf[tail % MAX] = item;  
    tail++;  
    empty.signal(&lock);  
    lock.release();  
}
```

tail	1
front	1
lock	FREE
ready list	A
lock wait queue	
empty queue	
full queue	

Moment to ponder

Does the k th call to `get()` *necessarily* return the k th item that was `put()`?



Considerations for Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
 - Condition variable is sync FOR shared state
 - ALWAYS hold lock when accessing shared state
 - Signals don't get "saved" if nobody is listening for them
 - Wait() atomically releases lock
 - If it's not atomic, we risk going into wait sleep after the condition has already become true.
- 

More considerations

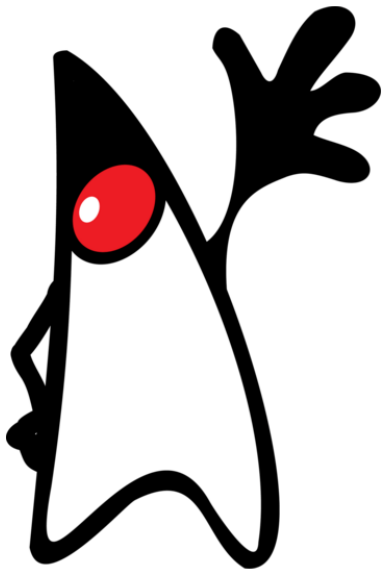
- When a thread is woken up from wait, it may not run immediately
 - Signal/broadcast just puts waiting threads back on the scheduler's ready list
 - When lock is released, *anyone* might acquire it
- Wait MUST be in a loop:

```
while (needToWait()) {  
    condition.Wait(&lock);  
}
```

This simplifies implementation:

- Of condition variables and locks
- Of code that uses condition variables and locks

Java Manual



When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

Big Picture

Thread-safe Design Practices

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
 - Grab lock on entry to every method/procedure
 - Release lock at return
- If there is a need to wait
 - `while(needToWait()) { condition.Wait(lock); }`
 - Do not assume the signal was sent any time close to when you receive it
- If you do something that might be relevant to a condition variable, send it a signal or broadcast
 - Accidentally waking threads up is low-cost
- Always leave shared state variables in a consistent state
 - When lock is released, or when waiting

Monitor

