Spin-Lock

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- Locks
    - Spinning and blocking
- Semaphore
- Readers/writer lock
- Seqlocks
- Condition variable

Why Linux synchronization ?

- **What is synchronization ?**
    - Code on multiple CPUs coordinate their operations
- No need for synchronization on early OSes, why ?
    - The CPU is only single processor
    - All kernel requests wait until completion – even disk requests
    - No possibility for two CPUs to touch same data
- Optimize kernel performance by blocking inside the kernel
    - Instead of waiting on expensive disk I/O, block and schedule another process until it completes
    - Need a **lock** to protect concurrent update to pages/inodes etc..
    - **For better CPU utilization**

Multi-processing

- **Multi-processing**
  - CPUs aren’t getting faster, just smaller
  - We can put more cores on a chip
  - The only way for software to get faster is to do more things at the same time
- **Performance scalability**
  - 1 -> 2 CPUs doubles the work: perfect scalability
  - However, most software isn’t scalable. Why ?

Coarse vs. fine-grained locking

- **Coarse-grained locking**
    - A single lock for everything
    - Idea: Before touching any shared data, grab the lock
    - Problem: unrelated operations wait on each other -> adding CPUs doesn’t improve performance
- **Fine-grained locking**
    - Many “small” locks for individual data structures
    - Idea: unrelated activities hold different locks -> adding CPUs can improve performance
    - Cost: complex to coordinate locks

How do locks work ?

- **Two key ingredients**
    - A hardware-provided atomic instruction
        - Determines who wins under contention
    - A waiting strategy for the loser(s)
- **Atomic instruction**
    - Guarantees that the entire operation is not interleaved with any other CPU
    - Intuition: The CPU ‘locks’ all of memory
        - Expensive !
    - Programmers must explicitly place atomic codes

Atomic instructions + locks

- Most lock implementations have some sort of counter
- Say initialized to 1
- To acquire the lock, use an atomic decrement
    - If someone sets the value to 0, go ahead
    - If someone gets < 0, wait
    - Atomic decrement ensures that only one CPU will decrement the value to zero
    - To release, set the value back to 1

Waiting strategies

- **Spinning**
  - Just poll the atomic counter in a busy loop
  - When it becomes 1, try the atomic decrement again
- **Blocking**
  - Create a kernel **wait queue** and go to **sleep**, **yield** the CPU to more useful work
  - Winner is responsible to wake up losers (in addition to setting lock variable to 1)
  - Create a kernel wait queue – the same thing used to wait on I/O
    - Moving to a wait queue takes you out of the scheduler's run queue

Which strategy is better ?

- **Main consideration**
    - Expected time waiting for the lock (spin) vs. time to do two context switches (yield)
    - If the lock will be held a long time (like while waiting for disk I/O)
        - Yield (waiting) makes sense
    - If the lock is only held momentarily
        - Spinning make sense

Linux spin lock

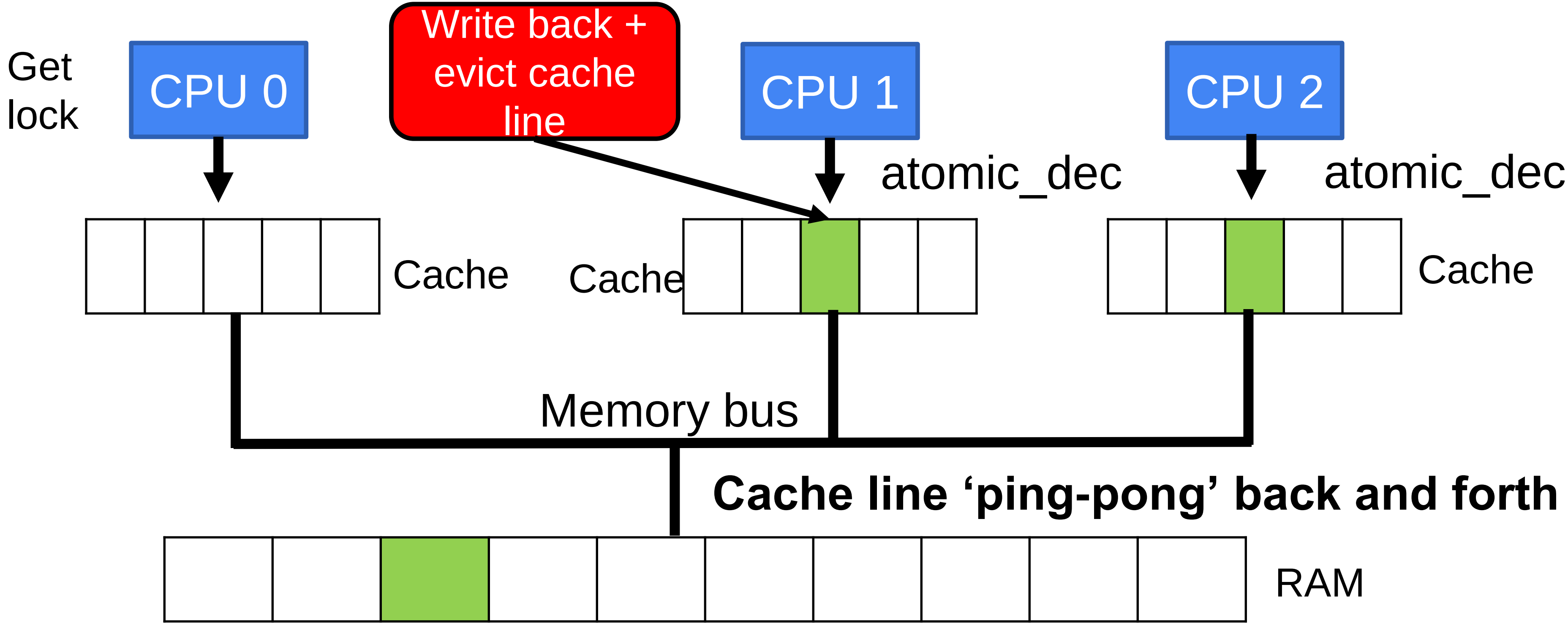
```
while (0 != atomic_dec (&lock->counter)) {
        do {
                // Pause the CPU until some coherence traffic
                // (a prerequisite for the counter changing)
                // completes
        } while (lock->counter <= 0);
}
```

Why two loops ?

- Functionally, the outer loop is sufficient
- **Problem:**
    - Attempts to write this variable invalidate it in all other caches
    - If many CPUs are waiting on this lock, the **cache line will bounce** between CPUs that are polling its value
    - **Cache line bouncing**
        - When multiple processors are trying to R/W to a same address
        - This cache line will move to other processor who is requesting
        - Then move back if the original processor again requests for the same line
    - The inner loop read-shares this cache line, allow all polling in parallel

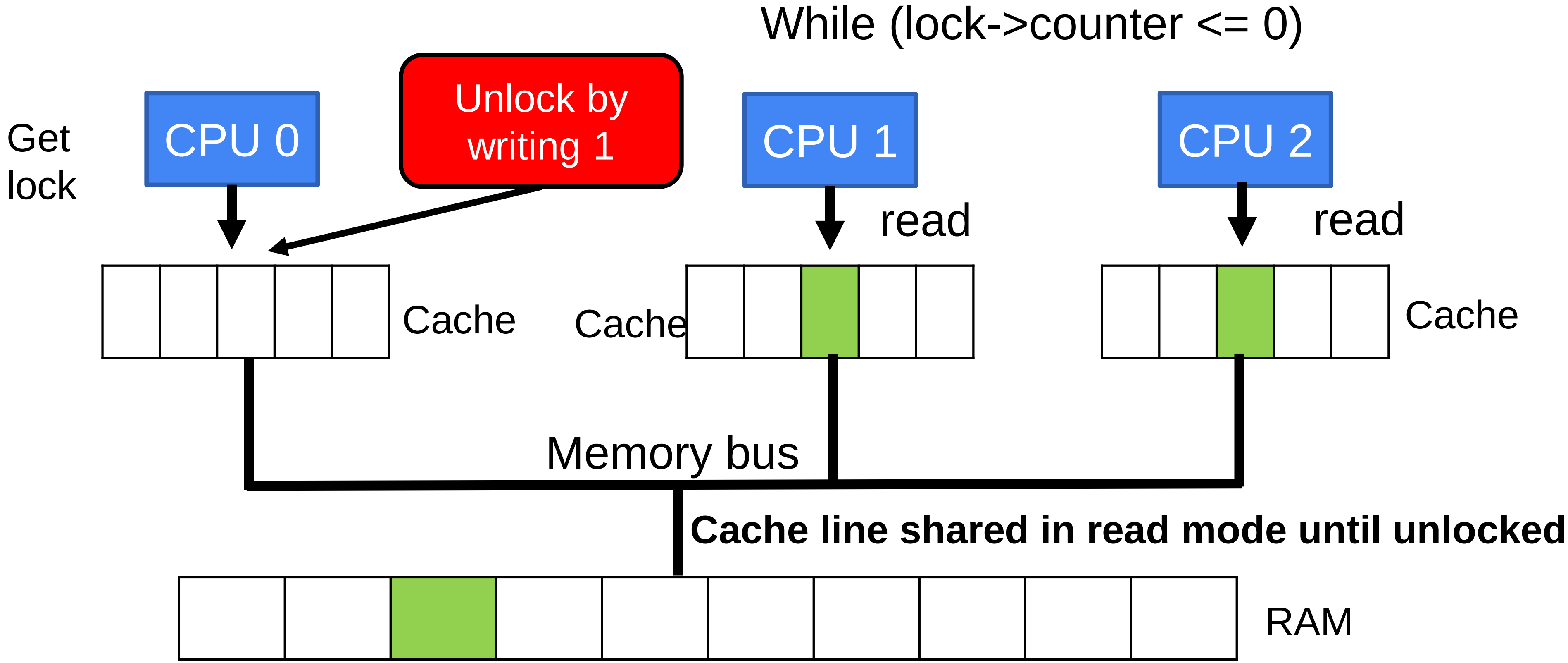
Test & Set lock

While (!atomic_dec (&lock->counter))



Cache line 'ping-pong' back and forth

Test & Set lock



Cache line shared in read mode until unlocked

Semaphore

- A semaphore is a counter that processes or threads can manipulate atomically
    - A mutex (lock) is the special case of 1 at a time -> binary semaphore
- Plus a wait queue
- Implementation
    - Similar to a spinlock, except spin loop replaced with placing oneself on a **wait queue**

Semaphore

- Operations on a semaphore
    - **P() or wait():** wait until counter > 0, then atomically decrement it
        - sem_wait(): decrement the value of the semaphore
    - **V() or signal() or post():** atomically increment counter
        - sem_post(): restore the value of the semaphore
- Counter represents the number of available resources
    - Never negative
- A semaphore whose counter is always 0 or 1 is called a **binary semaphore**
    - **This is just a lock**

Semaphore vs. mutex

- **Mutex**
  - A mutex **can be released only by the thread that had acquired it**
  - Let only one thread enter critical section -> should avoid priority inversion
  - The context switch occurs when one thread completes a certain amount of the work
- **Semaphore**
  - A binary semaphore **can be signaled by any threads** (or process)
  - Allow a number of thread enter critical section
  - Semaphore realizes the synchronization by using signals to notify other threads

Reader/writer locks

- **Problem: Share resource that is “read mostly”**
  - Enforcing strict mutual exclusion may be unacceptable
  - Want to allow arbitrary number of “readers” concurrently
  - Only want to allow “writer” if nobody else reading or writing
- **Idea**
  - In reading, let multiple readers access the data at the same time
  - Writers require mutual exclusion
  - Use the writelock semaphore to ensure that only a single writer can acquire the lock

Reader/writer locks

- When acquiring a read lock
  - The reader first acquires lock
  - Increments the readers variable to track the number of readers are inside the data structure
  - The read also acquires the write lock by calling sem_wait()
  - Releasing the lock by calling sem_post ()

```
typedef struct _rwlock_t {
    sem_t lock; // binary semaphore
    //allow ONE writer/MANY readers
    sem_t writelock;
    int readers; // #readers in critical section
} rwlock_t;
void rwlock_init (rwlock_t *rw) {
    rw-> readers = 0;
    sem_init (&rw->lock, 0, 1);
    sem_init (&rw->writelock, 0, 1);
}
void rwlock_acquire_readlock (rwlock_t *rw) {
    sem_wait (&rw->lock);
    rw->readers ++;
    // first reader gets writelock
    if (rw->readers == 1)
            sem_wait (&rw->writelock);
    sem_post (&rw->lock); }
```

Linux RW-spinlocks

- Low 24 bits count active readers
  - Unlocked: 0x01000000
  - To read lock: atomic_dec_unless (count, 0)
    - 1 reader: 0x00ffffff
    - 2 readers: 0x00fffffe
    - Readers limited to 2^24. That is a lot of CPUs !
  - 25th bits for writer
    - Readers will fail to acquire the lock until we add 0x01000000

Read/write lock issue

- What if we have a constant stream of readers and a waiting writer ?
  - The writer will starve
- How to prioritize writers over readers ?
  - Seqlocks

Seqlocks

- Explicitly favor writers, potentially starve readers
- **Idea**
  - An explicit write lock (one writer at a time)
  - Plus version number – each writer increments at beginning and end of critical section
- **Readers**
  - Check version number, read data, check again
  - If version changed, try again in a look
  - If version hasn't changed and is even, neither has data

Condition Variables

- Queue of threads waiting on some “event” inside a critical section
- A condition variable is always paired with a lock
- Operations
  - **Wait()**
    - Atomically release lock and go to sleep
    - When thread wakes up, it re-acquire the lock
  - **Signal()**
    - Wake up thread waiting on event -> no-op if nobody is waiting
  - **Broadcast()**
    - Wake up all threads waiting on event-> no-op if nobody is waiting

https://my.eng.utah.edu/~cs5460/slides/Lecture10.pdf

Condition Variable

Expected output:

```
parent: begin
child
parent: end
```

- **Condition variables**
  - Another synchronization primitive beyond locks
  - An explicit queue that threads can put themselves on when some state of execution (condition) is not as desired

```
void *child (void *arg) {
    printf (“child\n”);
    // XXX how to indicate we are
done ?
    return NULL;}
```

```
int main (int argc, char *argv[]) {
    printf (“parent: begin\n”);
    pthread_t c;
    // create child
    pthread_create(&c, NULL, child, NULL);
    // XXX how to wait for child ?
    printf (“parent: end\n”);
    return 0; }
```

How does a parent thread check the state of a child thread ? How to implement such a wait ?

Spin-based approach

- **Spin-based approach**
    - Generally work, but
    - The parent spins and waste CPU time -> inefficient
    - Why not put parent to sleep until the condition we are waiting for comes true ?

```
volatile int done = 0;

void *child (void *arg) {
    printf (“child\n”);
    done = 1;
    return NULL;
}

int main (int argc, char *argv[]) {
    printf (“parent: begin\n”);
    pthread_t c;
    pthread_create (&c, NNULL, child,
NULL);
    while (done == 0); // spin
    printf (“parent: end\n”);
    return 0;
}
```

Parent waiting for Child

```
void thr_join () {
    pthread_mutex_lock (&m);
    while (done == 0)
                pthread_cond_wait (&c, &m);
    pthread_mutex_unlock (&m);
}
int main (int argc, char *argv[]) {
    printf (“parent: begin\n”);
    pthread_t p;
    pthread_create (&p, NULL, child, NULL);
    thr_join ();
    printf (“parent: end\n”);
    return 0;
}
```

```
int done = 0;
pthread_mutex_t m =
PTHREAD_MUTEX_INIT;
pthread_cond_t c = PTHREAD_COND_INIT;

void thr_exit () {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&c);
    pthread_mutex_unlock (&m);
}

void *child (void *arg) {
    printf (“child\n”);
    thr_exit ();
    return NULL;
}
```

Parent waiting for Child

- **The first case**
  - The parent creates the child, but continue running itself
  - Immediately calls into thr_join () to wait for the child thread to complete
  - The parent acquires the lock, check if the child is done, and put itself to sleep by calling wait ()

```
void thr_join () {
    pthread_mutex_lock (&m);
    while (done == 0)
                 pthread_cond_wait (&c,
&m);
    pthread_mutex_unlock (&m);
}
int main (int argc, char *argv[]) {
    printf (“parent: begin\n”);
    pthread_t p;
    pthread_create (&p, NULL, child,
NULL);
    thr_join ();
    printf (“parent: end\n”);
    return 0;
}
```

Parent waiting for Child

- **The first case**
  - The child runs, print the message
  - Then, the child call thr_exit () to wake the parent thread
  - The child grabs the lock, sets the state variable “done”, and signals the parent to wake it up
  - Finally, the parent runs, unlock the lock, and print the “parent: end”

```
void thr_exit () {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&c);
    pthread_mutex_unlock (&m);
}

void *child (void *arg) {
    printf (“child\n”);
    thr_exit ();
    return NULL;
}
```

Parent waiting for Child

- **The second case**
  - The child runs immediately upon creation, sets “done” to 1
  - The child calls signal to wake a sleeping thread
  - The parent then runs, calls thr_join (), see that “done” is 1
  - The parent doesn’t wait and returns
  - **This approach is broken, why ?**

```
void thr_join () {
    pthread_mutex_lock (&m);
    while (done == 0)
                pthread_cond_wait (&c, &m);
    pthread_mutex_unlock (&m);
}
void thr_exit () {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&c);
    pthread_mutex_unlock (&m);
}
void *child (void *arg) {
    printf (“child\n”);
    thr_exit ();
    return NULL;
}
```

Parent waiting for Child

- Why that code is broken ?
  - The child runs immediately and calls thr_exit () immediately
  - The child will signal, but no thread falls asleep on the condition
  - When the parent runs, it calls wait and is stuck; **no thread will ever wake it**

```
void thr_join () {
    pthread_mutex_lock (&m);
    while (done == 0)
                pthread_cond_wait (&c, &m);
    pthread_mutex_unlock (&m);
}
void thr_exit () {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&c);
    pthread_mutex_unlock (&m);
}
void *child (void *arg) {
    printf (“child\n”);
    thr_exit ();
    return NULL;
}
```

Parent waiting for Child

```
void thr_join () {
     if (done == 0)
               pthread_cond_wait (&c,
&m);
}
void thr_exit () {
    done = 1;
    pthread_cond_signal (&c);
}
```

- **What’s wrong after removing the lock ?**
    - The parent calls thr_join (), then checks the value of done
    - The parent sees that it is 0 and thus try to go to sleep
    - Before the parent calls wait to sleep, the parent is interrupted, and the child runs
    - The child changes the state variable “done” to 1 and signals, but no thread is waiting, and woken
    - **When the parent runs again, it sleeps forever**

Producer/consumer (bounded buffer) problem

- **Bounded buffer problem**
  - Multiple producer and consumer threads
  - Producers generate data items and place them in a buffer
  - Consumers grab items from the buffer and consume them
  - Trouble when
    - Producer produces, but buffer is full
    - Consume consumes, but buffer is empty

```
int buffer;
int count = 0; // initially, empty
void put (int value) {
    assert (count == 0);
    count = 1;
    buffer = value;
}
void get () {
    assert (count == 1);
    count = 0;
    return buffer;
}
```

Single condition variable

- A single condition variable “cond” and associated lock “mutex”.
- If we have more than one thread, this code has two problems. What ?

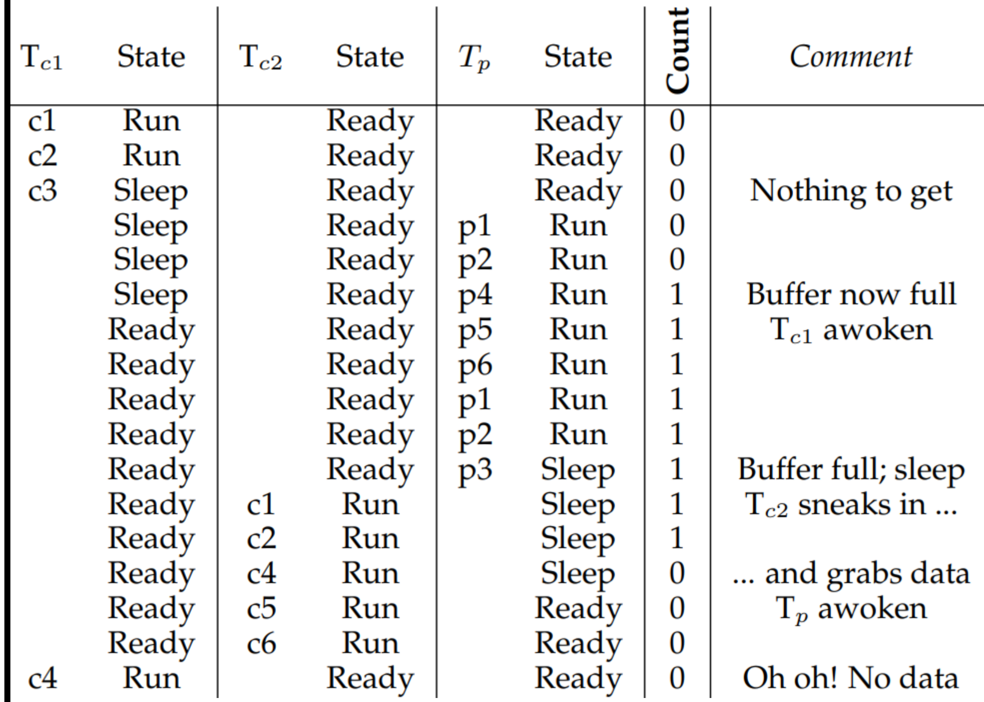
```
void *consumer (void *arg) {
 for (int i = 0; I < loops; i++) {
   pthread_mutex_lock (&mutex);            //c1
   if (count == 0)                         //c2
    pthread_cond_wait (&cond, &mutex); //c3
   get (i);                                //c4
   pthread_cond_signal (&cond);            //c5
   pthread_mutex_unlock (&mutex);      // c6
 }
}
```

```
void *producer (void *arg) {
 for (int i = 0; I < loops; i++) {
   pthread_mutex_lock (&mutex);            //p1
   if (count == 1)                         // p2
    pthread_cond_wait (&cond, &mutex); //p3
   put (i);                                //p4
   pthread_cond_signal (&cond);          // p5
   pthread_mutex_unlock (&mutex);      // p6
 }
}
```

Single condition variable

Two consumers (Tc1 and Tc2) and one producer (Tp)

1. Tc1 first runs, acquire the lock (c1), check buffer state (c2), finding that none are, wait (c3)
2. Tp runs and acquires the lock (p1), check if the buffer is full (p2), fills the buffer (p4)
3. Tp signals that a buffer has been fill. (p5), move Tc1 from sleeping to ready queue.
4. Tp continues until realizing the buffer is full, at which point it sleeps (p6, p1-p3)
5. The problem occurs: when Tc2 sneaks in and consumes the one value in the buffer
6. **No data for Tc1 when Tc1 resumes**
7. **We should avoid Tc2 sneaking in and consume the one produced value**



T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

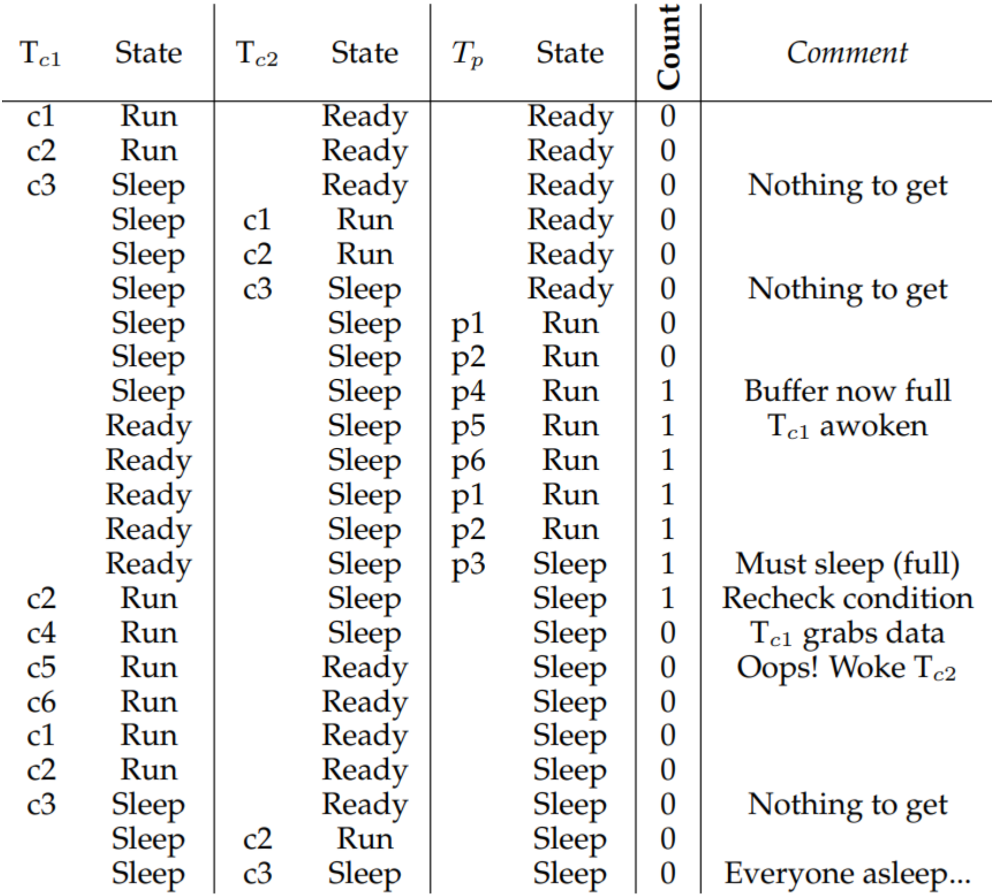
While, Not if

```
void *consumer (void *arg) {
 for (int i = 0; I < loops; i++) {
   pthread_mutex_lock (&mutex);              //c1
   while (count == 0)                                //c2
    pthread_cond_wait (&cond, &mutex); //c3
   get (i);                                                  //c4
   pthread_cond_signal (&cond);              //c5
   pthread_mutex_unlock (&mutex);        // c6
 } }
```

- Change the ‘if’ to a ‘while’
  - Consumer Tc1 wakes up
  - Immediately re-checks the state of the shared variable (c2)
  - Tc1 sleeps if the buffer is empty
  - The producer is also changed to a while (p2)
  - Using ‘while’ around conditional checks to avoid **spurious wakeup** occurs
- However, this code is still buggy after using ‘while’. Why ?
  - The buffer is full, Tc2 and Tp are sleeping and Tc1 is ready to run
  - Tc1 consumes the value (c4), then
  - Tc1 signals on the condition (c5), waking only one thread that is sleeping
  - However**, which thread should it wake ?**

While, Not if

- **Buggy code**
  - Tp and Tc2 are sleeping, which one should be waked up ? (Shared buffer is empty)
  - If Tc1 wakes up Tc2, Tc2 finds the buffer is empty (c2)
  - Then, Tc2 sleeps (c3)
  - Tp is left sleeping
  - **Thus, everyone is sleeping**



T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

Two condition variable

```
void *producer (void *arg) {
 for (int i = 0; I < loops; i++) {
   pthread_mutex_lock (&mutex);                  //p1
   while (count == MAX)                             //p2
     pthread_cond_wait (&empty, &mutex); //p3
   put (i);
//p4
   pthread_cond_signal (&fill);                   // p5
   pthread_mutex_unlock (&mutex);             // p6
 } }
```

1. A producer only sleeps if all buffers are currently filled. (p2)
2. A consumer only sleeps if all buffers are currently empty.

```
int buffer [MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0; // initially, empty
void put (int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count ++;
}
int get () {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) %MAX;
    count --;
    return tmp;
}
```

Covering condition

- **Covering condition**
  - Assume there are zero bytes free;
  - Thread Ta allocate (100), Tb asks for allocate (10). Tc calls free (50)
  - Which waiting thread (Ta or Tb) should be woken up ?
- **Lampson’s solution**
  - Using ‘pthread_cond_broadcast’ to wake up all waiting threads
  - Guarantee any threads that should be woken are negative performance impact

```
int bytesLeft = MAX_HEAP_SIZE;
void *allocate (int size) {
    pthread_mutex_lock (&m);
    while (bytesLeft < size)
       pthread_cond_wait (&c, &m);
     void *ptr = …; //get mem from heap
     bytesLeft -= size;
     pthread_mutex_unlock (&m);
     return ptr;
}
void free (void *ptr, int size) {
    pthread_mutex_lock (&m);
    bytesLeft += size;
    pthread_cond_signal (&c); // who to signal
    pthread_mutex_unlock (&m);
}
```

Conclusion

- Performance scalability vs. locking
- Fine-grained vs. coarse-grained locking
- Lock waiting strategies – spinning and yield
- Semaphore vs. mutex
- Readers/writer lock
    - Let multiple readers access the shared data at the same time
- Condition variable
    - wait(), signal(), broadcast()