Locks and Condition Variables

Questions answered in this lecture:
How can threads block instead of spin-waiting while waiting for a lock?
When should a waiting thread block and when should it spin?
How can threads enforce ordering across operations (condition variables)?
How can thread_join() be implemented?
How can condition variables be used to support producer/consumer apps?

ANNOUNCEMENTS

Exam 2 solutions posted
- Look in your handin directory for midterm1.pdf details

Project 2: Due Sunday midnight

Project 3: Shared Memory Segments – Available Monday
- New project partner if desired; your own or matched
- Linux: Using shmget() and shmat()
  - with partner
- Xv6: Implementing shmget() and shmat()
  - Alone
  - Due Wednesday 11/02

Today’s Reading: Chapter 30
TICKET LOCK IMPLEMENTATION

typedef struct __lock_t {
    int ticket;
    int turn;
} __lock_t;

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while(lock->turn != myturn); // spin
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void release(lock_t *lock) {
    lock->turn++;
}

FAA() used in textbook → conservative
Try this modification in Homework simulations

LOCK EVALUATION

How to tell if a lock implementation is good?

Fairness:
- Do processes acquire lock in same order as requested?

Performance
Two scenarios:
- low contention (fewer threads, lock usually available)
- high contention (many threads per CPU, each contending)
CPU SCHEDULER IS IGNORANT

CPU scheduler may run B instead of A even though B is waiting for A

TICKET LOCK IMPLEMENTATION

typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while(lock->turn != myturn); // spin
}

void release(lock_t *lock) {
    lock->turn++;
}

Trivial modification to improve?
**TICKET LOCK WITH YIELD()**

```c
typedef struct __lock_t {
    int ticket;
    int turn;
} __lock_t;

void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while(lock->turn != myturn)
        yield();
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}

Remember: yield() voluntarily relinquishes CPU for remainder of timeslice, but process remains READY
```

**YIELD INSTEAD OF SPIN**

![Diagram showing the difference between no yield and yield in lock acquisition and release]

- **No Yield:**
  - Process A acquires lock at 0, spins until 20, releases lock at 20, then spins until 40.
  - Process B acquires lock at 40, spins until 60, releases lock at 60, then spins until 80.
  - Process C acquires lock at 80, spins until 100, releases lock at 100, then spins until 120.
  - Process D acquires lock at 120, spins until 140, releases lock at 140, then spins until 160.

- **Yield:**
  - Process A acquires lock at 0, releases lock at 20, Acquires lock at 20, releases lock at 40, then Acquires lock at 40, releases lock at 60, Acquires lock at 60, releases lock at 80, then Acquires lock at 80, releases lock at 100, Acquires lock at 100, releases lock at 120, then Acquires lock at 120, releases lock at 140, Acquires lock at 140, releases lock at 160.

SPINLOCK PERFORMANCE

Waste…
   Without yield: \( O(\text{threads} \times \text{time_slice}) \)
   With yield: \( O(\text{threads} \times \text{context_switch}) \)

So even with yield, spinning is slow with high thread contention

Next improvement:
Block and put thread on waiting queue instead of spinning

LOCK IMPLEMENTATION:
BLOCK WHEN WAITING

Lock implementation removes waiting threads from scheduler ready queue (e.g., park() and unpark())

Scheduler runs any thread that is ready

Good separation of concerns
RUNNABLE: A, B, C, D
RUNNING: <empty>
WAITING: <empty>
Same as BLOCKED

RUNNABLE: B, C, D
RUNNING: A
WAITING: <empty>
Same as BLOCKED
RUNNABLE:  C, D, A
RUNNING:  B
WAITING:  <empty>
Same as BLOCKED
RUNNABLE: D, A
RUNNING: C
WAITING: B
Same as BLOCKED

RUNNABLE: A, C
RUNNING: D
WAITING: B
Same as BLOCKED
RUNNABLE: A, C
RUNNING: A
WAITING: B, D
Same as BLOCKED

RUNNABLE: C
RUNNING: A
WAITING: B, D
Same as BLOCKED
RUNNABLE: A
RUNNING: C
WAITING: B, D
Same as BLOCKED

RUNNABLE: C
RUNNING: A
WAITING: B, D
Same as BLOCKED
RUNNABLE: B, C
RUNNING: A
WAITING: D
Same as BLOCKED
RUNNABLE: C, A
RUNNING: B
WAITING: D
Same as BLOKED

lock    try lock (sleep)    try lock (sleep)  unlock    lock
A  B  C  D  A  C  A  B

(a) Why is guard used?
(b) Why okay to spin on guard?
(c) In release(), why not set lock=false when unpark?
(d) What is the race condition?

typedef struct {
  bool lock = false;
  bool guard = false;
  queue_t q;
} LockT;

void acquire(LockT *l) {
  while (TAS(&l->guard, true));
  if (l->lock) {
    qadd(l->q, tid);
    l->guard = false;
    park(); // blocked
  } else {
    l->lock = true;
    l->guard = false;
  }
}

void release(LockT *l) {
  while (TAS(&l->guard, true));
  if (qempty(l->q)) l->lock=false;
  else unpark(qremove(l->q));
  l->guard = false;
}
**RACE CONDITION**

**Thread 1**  (in acquire)
if (l->lock) {
    qadd(l->q, tid);
l->guard = false;
}

**Thread 2**  (in release)
while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
park(); // block

Problem: Guard not held when call park()
Unlocking thread may unpark() before other park()

**BLOCK WHEN等待: FINAL CORRECT LOCK**

**Typedef struct** {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;

```
void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        setpark(); // notify of plan
        l->guard = false;
        park(); // unless unpark()
    } else {
        l->lock = true;
        l->guard = false;
    }
}
```

```
void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
```

setpark() fixes race condition
Park() does not block if unpark() occurred after setpark()
**SPIN-WAITING VS BLOCKING**

Each approach is better under different circumstances

**Uniprocessor**
- Waiting process is scheduled --> Process holding lock isn't
- Waiting process should always relinquish processor
- Associate queue of waiters with each lock (as in previous implementation)

**Multiprocessor**
- Waiting process is scheduled --> Process holding lock might be
- Spin or block depends on how long, \( t \), before lock is released
  - Lock released quickly --> Spin-wait
  - Lock released slowly --> Block
- Quick and slow are relative to context-switch cost, \( C \)

**WHEN TO SPIN-WAIT? WHEN TO BLOCK?**

If know how long, \( t \), before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

\[
\text{t}
\]

How much wasted when block?

\[
\text{C}
\]

What is the best action when \( t < C \)?

spin-wait

When \( t > C \)?

block

Problem:
- Requires knowledge of future; too much overhead to do any special prediction
TWO-PHASE WAITING

Theory: Bound worst-case performance; ratio of actual/optimal

When does worst-possible performance occur?
   Spin for very long time $t >> C$
   Ratio: $t/C$ (unbounded)

Algorithm: Spin-wait for $C$ then block --> Factor of 2 of optimal

Two cases:
   $t < C$: optimal spin-waits for $t$; we spin-wait $t$ too
   $t > C$: optimal blocks immediately (cost of $C$); we pay spin $C$ then block
      (cost of 2 $C$); $2C / C \rightarrow 2$-competitive algorithm

Example of competitive analysis

IMPLEMENTING SYNCHRONIZATION

Build higher-level synchronization primitives in OS
   - Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

- Monitors
- Locks
- Semaphores
- Condition Variables
- Loads
- Stores
- Test&Set
- Disable Interrupts
CONDITION VARIABLES

CONCURRENCY
OBJECTIVES

Mutual exclusion (e.g., A and B don’t run at same time)
- solved with locks

Ordering (e.g., B runs after A does something)
- solved with condition variables and semaphores
**ORDERING EXAMPLE: JOIN**

```c
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);  // how to implement join()
Pthread_join(p2, NULL);
printf("main: done\n[balance: %d]\n[should: %d]\n",
       balance, max*2);
return 0;
```

**CONDITION VARIABLES**

Condition Variable: queue of waiting threads

**B** waits for a signal on CV before running
• wait(CV, …)

**A** sends signal to CV when time for **B** to run
• signal(CV, …)
**CONDITION VARIABLES**

`wait(cond_t *cv, mutex_t *lock)`
- assumes the lock is held when `wait()` is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

`signal(cond_t *cv)`
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

---

**JOIN IMPLEMENTATION: ATTEMPT 1**

Parent: 
```c
void thread_join() {
    Mutex_lock(&m);    // x
    Cond_wait(&c, &m); // y
    Mutex_unlock(&m);  // z
}
```

Child: 
```c
void thread_exit() {
    Cond_signal(&c);  // a
}
```

Example schedule:
- Parent: x y z
- Child: a

Works!
JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
  Mutex_lock(&m); // x
  Cond_wait(&c, &m); // y
  Mutex_unlock(&m); // z
}
```

Child:

```
void thread_exit() {
  Cond_signal(&c); // a
}
```

Can you construct ordering that does not work?

Example broken schedule:

- Parent: $x \rightarrow y$
- Child: $a$

Parent waits forever!

RULE OF THUMB 1

**Keep state** in addition to CV's!

CV's are used to signal threads when state changes

If state is already as needed, thread doesn't wait for a signal!
JOIN IMPLEMENTATION: ATTEMPT 2

Parent:
void thread_join() {
    Mutex_lock(&m);       // w
    if (done == 0)        // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m);    // z
}

Child:
void thread_exit() {
    done = 1;            // a
    Cond_signal(&c);     // b
}

Fixes previous broken ordering:

Parent: w x y z
Child: a b

Can you construct ordering that does not work?

Parent: w x y ... sleep forever ...
Child: a b
JOIN IMPLEMENTATION: CORRECT

Parent:
void thread_join() {
    Mutex_lock(&m); // w
    if (done == 0) // x
        Cond_wait(&c, &m); // y
    Mutex_unlock(&m); // z
}

Child:
void thread_exit() {
    Mutex_lock(&m); // a
    done = 1; // b
    Cond_signal(&c); // c
    Mutex_unlock(&m); // d
}

Parent: w x y z
Child: a b c

Use mutex to ensure no race between interacting with state and wait/signal

PRODUCER/CONSUMER PROBLEM
Pipe may have many writers and readers
- Implemented with finite-sized buffer

**Writers** add data to the buffer
- Writers have to **wait** if buffer is full

**Readers** remove data from the buffer
- Readers have to wait if buffer is empty
EXAMPLE: UNIX PIPES

write!

Buf:

start

end

EXAMPLE: UNIX PIPES

write!

Buf:

start

end
EXAMPLE: UNIX PIPES
read!

Buf:

start

end

EXAMPLE: UNIX PIPES
write!

Buf:

start

end
EXAMPLE: UNIX PIPES

read!

Buf:

start

end
EXAMPLE: UNIX PIPES

Buf:

note: readers must wait
EXAMPLE: UNIX PIPES

write!

Buf:

end

start

Buf:

end

start
EXAMPLE: UNIX PIPES

Buf:

start

end

write!

note: writers must wait
**Example: UNIX Pipes**

- Buf:
- Start
- End
- Read!

**Producer/Consumer Problem**

- **Producers** generate data (like pipe writers)
- **Consumers** grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems
- Web servers

General strategy use condition variables to:
- make producers wait when buffers are full
- make consumers wait when there is nothing to consume
PRODUCE/CONSUMER EXAMPLE

Start with easy case:
• 1 producer thread
• 1 consumer thread
• 1 shared buffer to fill/consume (max = 1)

Numfill = number of buffers currently filled
Examine slightly broken code to begin…

CONDITION VARIABLES

\textit{wait}(\text{cond}\_t *cv, \text{mutex}\_t *lock)
- assumes the lock is held when \text{wait()} is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning
- Implication:
  1. BLOCKED on condition variable
  2. \text{WAIT} to acquire lock again (separate step)

\textit{signal}(\text{cond}\_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing
numfull=0

[RUNNABLE]
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

Assume do_fill(i) increments numfull

[RUNNING]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

Assume do_get() decrements numfull
numfull=0

[RUNNABLE]
void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    Mutex_lock(&m);
    while(numfull == max)
      Cond_wait(&cond, &m);
    do_fill(i);
    Cond_signal(&cond);
    Mutex_unlock(&m);
  }
}

[RUNNING]
void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m);
    while(numfull == 0)
      Cond_wait(&cond, &m);
    int tmp = do_get();
    Cond_signal(&cond);
    Mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}
numfull = 0

[RUNNABLE]
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m);
        while (numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

[BLOCKED on CV]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

Arrow shows location when run again

numfull = 0

[RUNNING]
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m);
        while (numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

[BLOCKED on CV]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

Will producer be stuck waiting for mutex_lock()?
No, because cond_wait() releases lock for m
numfull=1

[RUNNING]
void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    Mutex_lock(&m);
    while(numfull == max)
      Cond_wait(&cond, &m);
    do_fill(i);
    Cond_signal(&cond);
    Mutex_unlock(&m);
  }
}

[BLOCKED on CV]
void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m);
    while(numfull == 0)
      Cond_wait(&cond, &m);
    int tmp = do_get();
    Cond_signal(&cond);
    Mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}

What happens to consumer?

numfull=1

[RUNNING]
void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    Mutex_lock(&m);
    while(numfull == max)
      Cond_wait(&cond, &m);
    do_fill(i);
    Cond_signal(&cond);
    Mutex_unlock(&m);
  }
}

[RUNNABLE, WAIT on MUTEX]
void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m);
    while(numfull == 0)
      Cond_wait(&cond, &m);
    int tmp = do_get();
    Cond_signal(&cond);
    Mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}
Who acquires lock next?
Could be either producer or consumer
Example gives lock to producer
What important thing happens during cond_wait()?
```c
numfull=1

[BLOCKED on CV]
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

[RUNNING: Acquires lock]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

Decrement numfull

```c
numfull=1

[BLOCKED]
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

[RUNNING]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```
void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m);
    while(numfull == 0)
      Cond_wait(&cond, &m);
    int tmp = do_get();
    Cond_signal(&cond);
    Mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}

numfull=0

[BLOCKED on CV]
void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    Mutex_lock(&m);
    while(numfull == max)
      Cond_wait(&cond, &m);
    do_fill(i);
    Cond_signal(&cond);
    Mutex_unlock(&m);
  }
}

[RUNNING]
void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m);
    while(numfull == 0)
      Cond_wait(&cond, &m);
    int tmp = do_get();
    Cond_signal(&cond);
    Mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}

What does signal() do?
numfull=0

[RUNNABLE: wait mutex] void *producer(void *arg) {
for (int i=0; i<loops; i++) {
    Mutex_lock(&m);
    while(numfull == max)
        Cond_wait(&cond, &m);
    do_fill(i);
    Cond_signal(&cond);
    Mutex_unlock(&m);
} }

[RUNNING]  void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

numfull=0

[RUNNABLE: wait mutex] void *producer(void *arg) {
for (int i=0; i<loops; i++) {
    Mutex_lock(&m);
    while(numfull == max)
        Cond_wait(&cond, &m);
    do_fill(i);
    Cond_signal(&cond);
    Mutex_unlock(&m);
} }

[RUNNING]  void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
numfull=0

[RUNNING]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

[RUNNABLE: wait mutex]
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}

numfull=0

[RUNNING]
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}

[RUNNABLE: wait mutex]
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
What does wait() do?
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        while(numfull == max) {
            Cond_wait(&cond, &m);
            do_fill(i);
            Cond_signal(&cond);
        }
        Mutex_unlock(&m);
    }
}

void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        while(numfull == 0) {
            Cond_wait(&cond, &m);
            int tmp = do_get();
            Cond_signal(&cond);
            Mutex_unlock(&m);
            printf("%d\n", tmp);
        }
    }
}
All fine with 1 producer, 1 consumer, 1 buffer
WHAT ABOUT 2 CONSUMERS?

Can you find a problematic timeline with 2 consumers (still 1 producer)?

TWO CONSUMERS: PROBLEMS

void *producer(void *arg) {
  for (int i=0; i<loops; i++) {
    Mutex_lock(&m); // p1
    while(numfull == max) // p2
      Cond_wait(&cond, &m); // p3
    do_fill(i); // p4
    Cond_signal(&cond); // p5
    Mutex_unlock(&m); // p6
  }
}

void *consumer(void *arg) {
  while(1) {
    Mutex_lock(&m); // c1
    while(numfull == 0) // c2
      Cond_wait(&cond, &m); // c3
    int tmp = do_get(); // c4
    Cond_signal(&cond); // c5
    Mutex_unlock(&m); // c6
    printf("%d\n", tmp); // c7
  }
}

Producer: c1 c2 c3
Consumer1: c1 c2 c3
Consumer2: p1 p2 p4 p5 p6 p1 p2 p3
unblocked c2 c4 c5

does last signal wake producer or consumer1?
TWO CONSUMERS: PROBLEMS

void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        while(numfull == max) // p2
            Cond_wait(&cond, &m); // p3
        do_fill(i); // p4
        Cond_signal(&cond); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m); // c1
        while(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}

Producer:
Consumer1: c1 c2 c3
Consumer2: c1 c2 c3

Want consumer2 signal() to wake producer since numbufs = 0,
but could wake consumer1

HOW TO WAKE THE RIGHT THREAD?

One solution:
    wake all the threads!
WAKING ALL WAITING THREADS

- **wait** *(cond_t *cv, mutex_t *lock)*
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning

- **signal** *(cond_t *cv)*
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

- **broadcast** *(cond_t *cv)*
  any disadvantage?
  - wake all waiting threads (if >= 1 thread is waiting)
  - if there are no waiting threads, just return, doing nothing

EXAMPLE NEED FOR BROADCAST

```c
void *allocate(int size) {
    mutex_lock(&m);
    while (bytesLeft < size)
        cond_wait(&c);
    ...
}

void free(void *ptr, int size) {
    ...
    cond_broadcast(&c)
    ...
}
```
**HOW TO WAKE THE RIGHT THREAD?**

One solution:

wake all the threads!

Better solution (usually): use separate condition variables

---

**PRODUCER/CONSUMER: TWO CVS**

```c
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i); // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0) // p2
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

Is this correct? Can you find a bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1 from CV (must still get mutex!)
3. before consumer1 runs, consumer2 runs, gets lock, grabs entry, sets numfull=0.
4. consumer2 runs, gets lock, then reads bad data 😞
Good Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!

Possible for thread B to grab lock in between signal and thread A returning from wait (before thread A gets lock)

Some implementations have “spurious wakeups”
  - May wake multiple waiting threads at signal or at any time
  - May treat signal() as broadcast()

Producer/Consumer:
Two CVS and While

```c
void *producer(void *arg) {  
    for (int i = 0; i < loops; i++) {  
        Mutex_lock(&m); // p1  
        while (numfull == max) // p2  
            Cond_wait(&empty, &m); // p3  
        do_fill(i);  // p4  
        Cond_signal(&fill); // p5  
        Mutex_unlock(&m); // p6  
    }  
}
```

```c
void *consumer(void *arg) {  
    while (1) {  
        Mutex_lock(&m);  
        while (numfull == 0)  
            Cond_wait(&fill, &m);  
        int tmp = do_get();  
        Cond_signal(&empty);  
        Mutex_unlock(&m);  
    }  
}
```

Is this correct? Can you find a bad schedule?
Correct!
  - no concurrent access to shared state
  - every time lock is acquired, assumptions are reevaluated
  - a consumer will get to run after every do_fill()
  - a producer will get to run after every do_get()
SUMMARY: RULES OF THUMB FOR CVS

Keep state in addition to CV’s

Always do wait/signal with lock held

Whenever thread wakes from waiting, recheck state