Concurrency: Signaling and Condition Variables

Questions Answered in this Lecture:

• How do we make fair locks perform better?
• How do we notify threads of that some condition has been met?
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

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

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

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

Fast when...
- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when...
- one CPU
- locks held a long time
- disadvantage: spinning is wasteful
CPU Scheduler is Ignorant

CPU scheduler may run B instead of A even though B is waiting for A
Ticket Lock with `yield()`

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

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

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

void release (lock_t *lock) {
    FAA(&lock->turn);
}
yield() Instead of spin

no yield:

yield:
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
How do we evaluate lock performance?

• **Fairness:**
  - do some processes wait more than others?
  - another way: do processes get lock in order requested?

• **Performance:** it depends...
  - Is the lock *contended*? (many threads attempting to acquire lock)
  - Or is it *uncontended*? (likely to get the lock if we request it)
Why is `yield()` useful?

How can it affect performance?

no yield:

yield:
Lock implementation: block when waiting

• Lock *implementation* removes waiting threads from the ready queue (e.g. with `park()` and `unpark()`)

• Scheduler only runs *ready* threads

• *Separates concerns* (programmer doesn’t need to deal with yield vs not/spinning issue)

• *Quiz*: where should locks be implemented?
Example

Ready: \{A, B, C, D\}
Waiting: \{}
Running: \{}
Example

Ready: \{B, C, D\}
Waiting: {} 
Running: \{A\}

lock()
Example

Ready: \{A, C, D\}
Waiting: \{\}
Running: \{B\}
Example

Ready: \{A, C, D\}
Waiting: \{B\}
Running: {}
Example

Ready: \{A, D\}
Waiting: \{B\}
Running: \{C\}

\textit{lock()}\textit{lock()}
Example

Ready: \{A, C\}
Waiting: \{B\}
Running: \{D\}
Example

Ready: \{A, C\}
Waiting: \{B, D\}
Running: {}
Example

Ready: \{C\}
Waiting: \{B, D\}
Running: \{A\}

lock() lock() lock()
Example

Ready: \{A\}
Waiting: \{B, D\}
Running: \{C\}

lock() lock() lock()
Example

Ready: \{C\}
Waiting: \{B, D\}
Running: \{A\}
Example

Ready: \{C, B, D\}
Waiting: \{\}
Running: \{A\}

\begin{itemize}
  \item A \textbf{lock()} \item B \textbf{lock()}
  \item C \textbf{lock()}
  \item A \textbf{unlock()}
\end{itemize}
Example

Ready: \{C, B, D\}
Waiting: \{\}
Running: \{A\}

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

0 10 20 30 40 50 60 70 80 100
Example

Ready: \{B, D, A\}
Waiting: \{
Running: \{C\}

\text{lock()}\text{lock()}\text{lock()}\text{unlock()}

\begin{itemize}
\item A
\item B
\item C
\item D
\item A
\item C
\item A
\item C
\end{itemize}
Example

Ready: \{D, A, C\}
Waiting: \{\}
Running: \{B\}

A
lock()
B
lock()
C
lock()
D
lock()
A
unlock()
B
lock()
What do we get?

• *Threads aren’t contending on the lock* when they shouldn’t be (mostly)

• Fewer context switches
Lock: block when waiting

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

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

void release(lock_t *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q))
        l->lock=false;
    else
        unpark(dequeue(l->q));
    l->guard = false;
}
Race!

Thread 1

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

Thread 2

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

Thread 1 calls acquire when time ≥ T1, Thread 2 calls release when time ≥ T2.

Bad state transition:

- Thread 1 acquires lock but does not enquire before releasing.

False state:

- `l->lock` is set to false.

Bad state:

- Correct execution state.

The issue is that Thread 1 acquires the lock without enquiring, leading to a false state where `l->lock` is set to false before Thread 2 can release it.
Lock: the fix

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

1. Why does this fix the race?
2. What does the scheduler need to do here?
3. Why can’t we just hold the guard when we park()?
Spin-waiting vs. Blocking

• Each approach has *tradeoffs*:

  • **Uniprocessor**
    • Waiting process is scheduled -> process holding lock isn’t
    • Waiting process should always be descheduled
    • Associate queue of waiters with each lock

  • **Multiprocessor**
    • Waiting process is scheduled -> *process holding lock might also be*
    • Spin or block depends on *how long (t) before lock is released*
    • Lock released quickly? `spin()`
    • Lock taken for long time? `block()`

  we’ll define this relative to the context switch time
Oracle

- Suppose we know how long \( t \) a lock will be held
- And suppose we make our decision to block or spin based on \( C \), the cost of a context switch

\[
\text{action} = \begin{cases} 
  t \leq C & \rightarrow \text{spin} \\
  t > C & \rightarrow \text{block}
\end{cases}
\]

BUT: we have to know the future!
Applying Bounds

- The theory: **bound our worst-case performance using actual wait time over optimal** (oracle)
- Consider:

\[
\begin{align*}
t \leq C & \quad \text{What would the oracle do?} \\
\text{What can we do (without knowing future)?} & \\
\text{We pay } 2C, \text{ because we wait an extra } C. \ 2C/C = 2 \Rightarrow 2\text{-competitive algorithm!} \\
\end{align*}
\]
Concurrency Goals

• **Mutual Exclusion**
  • Keep two threads from executing in a critical section concurrently
  • We solved this with *locks*

• **Dependent Events**
  • We want a thread to wait until some particular event has occurred
  • Or some condition has been met
  • Solved with *condition variables* and *semaphores*
“I just dropped in to see what condition my condition was in”
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);
pthread_join(p2, NULL);
printf("Main: done\n [balance: %d]\n", balance);
return 0;
```
Condition Variables

• **CV**: queue of waiting threads

• **B** waits for a signal on CV before running
  • `wait(CV, ...)`;

• **A** sends `signal()` on CV when time for **B** to run
  • `signal(CV, ...)`;
API

- **wait**(*cond_t * cv, mutex_t * lock*)
  - assumes lock is held when *wait()* is called (**why?**)
  - 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, NOP
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() {
    mutex_lock(&m); // a
    cond_signal(&c); // b
    mutex_unlock(&m); // c
}
```
Join Implementation: Attempt #1

```c
void thread_exit() {
    mutex_lock(&m);    // a
    cond_signal(&c);   // b
    mutex_unlock(&m);  // c
}

void thread_join() {
    mutex_lock(&m);    // x
    cond_wait(&c, &m); // y
    mutex_unlock(&m);  // z
}
```

Example schedule:

```
parent    x  y  z
----------
child     a  b  c
----------
```
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() {
    mutex_lock(&m);     // a
    cond_signal(&c);    // b
    mutex_unlock(&m);   // c
}
```

Example (bad) schedule:
```
parent
x
-------------
child   a   b   c
```

parent will wait forever!
Rule of Thumb #1

• *Always associate some state* with the CV
• CVs are used to signal() when state *changes*
• If state is as required, *don’t need to wait* for a signal()
Join Implementation: Attempt #2

```c
parent
void thread_join() {
    mutex_lock(&m);       // w
    if (!done)            // x
        cond_wait(&c, &m); // y
    mutex_unlock(&m);     // z
}

child
void thread_exit() {
    done = 1              // a
    cond_signal(&c);     // b
}
```
Join Implementation: Attempt #2

```c
void thread_join() {
    mutex_lock(&m);       // w
    if (!done)            // x
        cond_wait(&c, &m); // y
    mutex_unlock(&m);     // z
}

void thread_exit() {
    done = 1              // a
    cond_signal(&c);      // b
}
```

Example schedule:

<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>w x y z</td>
<td>a b</td>
</tr>
</tbody>
</table>

fixed!
Join Implementation: Attempt #2

```c
void thread_exit() {
  done = 1
  cond_signal(&c);
}
```

```c
void thread_join() {
  mutex_lock(&m);  // w
  if (!$done)      // x
    cond_wait(&c, &m); // y
  mutex_unlock(&m);  // z
}
```

Example (bad) schedule:

```
parent
  w  x  y

child
  a  b
```

sleeps forever!
Join Implementation: Attempt #3 (correct)

```c
parent
void thread_join() {
    mutex_lock(&m);    // w
    if (!done)         // 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
}
```
Join Implementation: Attempt #3 (correct)

```c

parent
void thread_join() {
    mutex_lock(&m);    // w
    if (!done)         // 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
}

Example schedule:

<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>a</td>
</tr>
<tr>
<td>x</td>
<td>b</td>
</tr>
<tr>
<td>y</td>
<td>c</td>
</tr>
<tr>
<td>z</td>
<td>d</td>
</tr>
</tbody>
</table>

fixed!
```
Signaling between threads

**Bounded Buffer (producer-consumer queue)**
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ t1: \text{put()} \]
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ \text{t1: put() } \]
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ t_2: \text{take()} \]
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ t_2: \text{take()} \]
Signaling between threads

Bounded Buffer (producer-consumer queue)
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ t2: \text{take()} \]
Signaling between threads

Bounded Buffer (producer-consumer queue)

What condition are we waiting on?
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ \text{t1: put()} \]
Signaling between threads

Bounded Buffer (producer-consumer queue)

$t1$: `put()`

What should happen?
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ t_1: \text{signal()} \]
Signaling between threads

Bounded Buffer (producer-consumer queue)

wait queue

t2: take()
Signaling between threads

Bounded Buffer (producer-consumer queue)

\[ t2: \text{take()} \]
The queue is a *circular queue*

• Also sometimes called a *ring buffer*
The queue is a *circular queue*

- Also sometimes called a *ring buffer*

**Bounded Buffer (producer-consumer queue)**

*most recent put*
The queue is a *circular queue*

- Also sometimes called a *ring buffer*

Bounded Buffer (producer-consumer queue)

We’re full
The queue is a *circular queue*

- Also sometimes called a *ring buffer*
The queue is a **circular queue**

- Also sometimes called a *ring buffer*
The queue is a *circular queue*

- Also sometimes called a *ring buffer*

![Diagram of a bounded buffer with a producer-consumer queue and a wait queue](image)

$t1$: `take()`
The queue is a *circular queue*

- Also sometimes called a *ring buffer*
The queue is a *circular queue*

- Also sometimes called a *ring buffer*
The queue is a *circular queue*

- Also sometimes called a *ring buffer*
The queue is a \textit{circular queue}

- Also sometimes called a \textit{ring buffer}
The queue is a *circular queue*

- Also sometimes called a *ring buffer*
The queue is a *circular queue*

- Also sometimes called a *ring buffer*
Bounded Buffer

- **Reads and writes** to buffer **require locking**
- when buffers are **full**, writers `wait()`
- when buffers are **empty**, readers `wait()`
- Many programming languages expose this abstraction directly
  - `chan` in Go
  - `channel` in Julia
  - Python requires a `queue()` with a `lock()`
  - Rust might use `CircularBuffer()` and `CondVar()` maybe `crossbeam_channel()`
Bounded Buffer

• **Producers** generate data
• **Consumers** take data and process it
• *Very frequent* situation encountered in systems programming
• General strategy: use CVs to notify:
  • waiting readers when data is available
  • waiting writers when slots are free
Example

• Easy case:
  • 1 consumer thread
  • 1 producer thread
  • 1 shared buffer (max slots=1)

• numfill is buffer slots currently filled
• let’s start with some code that doesn’t quite work
numfull = 0

[RUNNABLE]

```c
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]

```c
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

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);
    }
}
numfull = 0

**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);
  }
**}**
numfull = 0

[RUNNABLE]
```c
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);
    }
}
```

[SLEEPING]
```c
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]

```c
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);
    }
}
```

[SLEEPING]

```c
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]

```c
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);
    }
}
```

[SLEEPING]

```c
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]

```c
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);
    }
}
```

[SLEEPING]

```c
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 *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);
    }
}
numfull = 1

[RUNNING]

```c
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);
    }
}
```

[SLEEPING]

```c
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 *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);
    }
}
numfull = 1

[RUNNING]

```c
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]

```c
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 = 1

```c
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);
    }
}
```

```c
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 = 1

**[RUNNING]**

```c
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]**

```c
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 = 1

`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);
    }
}`
numfull = 1

**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);
  }
**}**
numfull = 1

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);
    }
}
numfull = 1

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);
  }
}
### numfull = 1

<table>
<thead>
<tr>
<th>[SLEEPING]</th>
<th>[RUNNING]</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *producer (void *arg) {</td>
<td></td>
</tr>
<tr>
<td>for (int i = 0; i &lt; loops; i++) {</td>
<td></td>
</tr>
<tr>
<td>mutex_lock(&amp;m);</td>
<td></td>
</tr>
<tr>
<td>while (numfull == max)</td>
<td></td>
</tr>
<tr>
<td>cond_wait(&amp;cond, &amp;m);</td>
<td></td>
</tr>
<tr>
<td>do_fill(i);</td>
<td></td>
</tr>
<tr>
<td>cond_signal(&amp;cond);</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock(&amp;m);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>void *consumer (void *arg) {</td>
<td></td>
</tr>
<tr>
<td>while (1) {</td>
<td></td>
</tr>
<tr>
<td>mutex_lock(&amp;m);</td>
<td></td>
</tr>
<tr>
<td>while (numfull == 0)</td>
<td></td>
</tr>
<tr>
<td>cond_wait(&amp;cond, &amp;m);</td>
<td></td>
</tr>
<tr>
<td>int tmp = do_get();</td>
<td></td>
</tr>
<tr>
<td>cond_signal(&amp;cond);</td>
<td></td>
</tr>
<tr>
<td>mutex_unlock(&amp;m);</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;%d\n&quot;, tmp);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
numfull = 0

[SLEEPING]  [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);
    }
}

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]

```c
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]

```c
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

```c
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);
    }
}
```
numfull = 0

**[RUNNABLE]**

```c
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]**

```c
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]

```c
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]

```c
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 *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);
    }
}
numfull = 0

[RUNNABLE]

```c
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]

```c
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]**

```c
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);
    }
}
```

**[SLEEPING]**

```c
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

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);
    }
}
numfull = 0

[RUNNING]

```c
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);
    }
}
```

[SLEEPING]

```c
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 = 1

[RUNNING]

```c
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);
    }
}
```

[SLEEPING]

```c
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 = 1

[RUNNING]

```c
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]

```c
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 about two consumers?

• Can you find a *bad schedule*?
```c
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);
        int tmp = do_get(); // c4
        cond_signal(&cond); // c5
        mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```
How to wake the right thread?

• One solution:
  • wake all threads!
API

• **wait**(cond_t * cv, mutex_t * lock)
  • assumes lock is held when *wait()* is called (*why?*)
  • 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, NOP

• **broadcast**(cond_t * cv)
  • wake *all* waiters
  • if no waiting threads, NOP
Summary

• **Rules of thumb**
  • Keep state in addition to CVs
  • Always `wait()` or `signal()` *with lock held*
  • Recheck state assumptions when waking up from waiting