Locks and semaphores

Johan Montelius

KTH

2019
recap, what’s the problem

: 
#include <pthread.h>

volatile int count = 0;

void *hello(void *arg) {
    for(int i = 0; i < 10; i++) {
        count++;
    }
}

int main() {
    pthread_t p1, p2;

    pthread_create(&p1, NULL, hello, NULL);
    pthread_create(&p2, NULL, hello, NULL);
    :
}
int request[2] = {0, 0};
int turn = 0;

int lock(int id) {
    request[id] = 1;
    int other = 1-id;
    turn = other;

    while(request[other] == 1 && turn == other) {}; // spin

    return 1;
}

void release(int id) {
    request[id] = 0;
}
Total Store Order

P1

a = 1

a

1

P2

b

0

0
Total Store Order

P1

\[ a = 1 \]

\[ b = 1 \]

P2

\[ a = 1 \]

\[ b = 1 \]
Total Store Order

\[
\begin{array}{c}
P1 \\
\hspace{1cm} a = 1 \\
\hspace{1cm} \text{read } b \\
0 \\
\hspace{1cm} 1 \\
\downarrow \\
\end{array}
\hspace{3cm}
\begin{array}{c}
P2 \\
0 \\
\text{read } a \\
1 \\
\downarrow \\
\end{array}
\hspace{3cm}
\begin{array}{c}
0 \\
b = 1 \\
\downarrow \\
\end{array}
\]
Total Store Order

P1

a = 1

0

read b

1

P2

0

read a

b = 1

1

0
atomic memory operations

All CPU:s provide several versions of atomic operations that both read and write to a memory element in one atomic operation. These include:

- **test-and-set**: swap i.e. read and write to a memory location, the simplest primitive.
- **fetch-and-add/and/xor/...**: update the value with a given operation, more flexible.
- **compare-and-swap**: if the memory location contains a specific value then swap.
All CPU:s provide several versions of atomic operations that both read and write to a memory element in one atomic operation.

- **test-and-set**: swap i.e. read and write to a memory location, the simplest primitive
All CPU:s provide several versions of atomic operations that both read and write to a memory element in one atomic operation.

- **test-and-set**: swap i.e. read and write to a memory location, the simplest primitive
- **fetch-and-add/and/xor/...**: update the value with a given operation, more flexible
atomic memory operations

All CPUs provide several versions of atomic operations that both read and write to a memory element in one atomic operation.

- **test-and-set**: swap i.e. read and write to a memory location, the simplest primitive
- **fetch-and-add/and/xor/...**: update the value with a given operation, more flexible
- **compare-and-swap**: if the memory location contains a specific value then swap
try to lock by swap

```c
int try(int *lock) {
    return __sync_val_compare_and_swap(lock, 0, 1);
}
```

This is using GCC extensions to C, similar extensions available in all compilers.
try to lock by swap

```c
int try(int *lock) {
    return __sync_val_compare_and_swap(lock, 0, 1);
}
```

```
pushq   %rbp
movq    %rsp, %rbp
movq    %rdi, -8(%rbp)
movq    -8(%rbp), %rdx
movl    $0, %eax
movl    $1, %ecx
lock    cmpxchgl %ecx, (%rdx)
nop
popq    %rbp
ret
```
try to lock by swap

```c
int try(int *lock) {
    return __sync_val_compare_and_swap(lock, 0, 1);
}
```

```
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq -8(%rbp), %rdx
movl $0, %eax
movl $1, %ecx
lock cmpxchgl %ecx, (%rdx)
nop
popq %rbp
ret
```

This is using GCC extensions to C, similar extensions available in all compilers.
int lock(int *lock) {

    while(try(lock) != 0) {} 

    return 1;
}

void release(int *lock) {
    *lock = 0;
}
int lock(int *lock) {  

    while(try(lock) != 0) {}  

    return 1;  
}  

void release(int *lock) {  
    *lock = 0;  
}
int global = 0;

int count = 0;

void *hello(void *name) {
    for(int i = 0; i < 10; i++) {
        lock(&global);
        count++;
        release(&global);
    }
}
We need to talk to the operating system.
We need to talk to the operating system.

```c
void lock(int *lock) {
    while (try(lock) != 0) {
        sched_yield();     // in Linux
    }
}
```
For how long should we sleep?
For how long should we sleep?
For how long should we sleep?

We would like to be woken up as the lock is released - before you go-go.
void lock(lock_t *m) {

    while(try(m->guard) != 0) {}; 

    if(m->flag == 0) {
        m->flag = 1;
        m->guard = 0;
    } else {
        queue_add(m->queue , gettid());
        m->guard = 0;
        park();
    }
}
void lock(lock_t *m) {
    while(try(m->guard) != 0) {};
    if(m->flag == 0) {
        m->flag = 1;
        m->guard = 0;
    } else {
        queue_add(m->queue, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while(try(m->guard) != 0) {};
    if(empty(m->queue)) {
        m->flag = 0;
    } else {
        unpark(dequeue(m->queue));
    }
    m->guard = 0;
}
It's not easy to get it right.

/* m->flag == 1 */
    : queue_add(m->queue, gettid());
m->guard = 0;
park();
// when I wake up the flag is set

if(empty(m->queue)) {
    m->flag = 0;
} else {
    // don’t reset the flag
    unpark(dequeue(m->queue));
}
It's not easy to get it right.

```c
/* m->flag == 1 */
    :
queue_add(m->queue, gettid());
setpark();
// if someone unparks now my park() is a noop
m->guard = 0;
park();
    if(empty(m->queue)) {
        m->flag = 0;
    } else {
        // don’t reset the flag
        unpark(dequeue(m->queue));
    }
```
Introducing futex: fast user space mutex.

- `futex_wait(mutex, val)`: suspend on the mutex if its equal to `val`.
- `futex_wake(mutex)`: wake one of the threads suspended on the mutex.

In GCC you have to call them using a `syscall()`.
Introducing futex: fast user space mutex.
Introducing futex: fast user space mutex.

- `futex_wait(mutex, val)` : suspend on the mutex if its equal to val.
- `futex_wake(mutex)` : wake one of the threads suspended on the mutex.
Introducing futex: fast user space mutex.

- futex_wait(mutex, val) : suspend on the mutex if its equal to val.
- futex_wake(mutex) : wake one of the threads suspended on the mutex

*In GCC you have to call them using a syscall()*
void lock(volatile int *lock) {
    while(try(lock) != 0) {
        // time to sleep ...
        futex_wait(lock, 1);
    }
}

void unlock(volatile int *lock) {
    *lock = 0;
    futex_wake(lock);
}
a futex lock

```c
void lock(volatile int *lock) { 
    while(try(lock) != 0) {
        // time to sleep ... 
        futex_wait(lock, 1);
    }
}

void unlock(volatile int *lock) {
    *lock = 0;
    futex_wake(lock);
}
```

Not very efficient - we want to avoid calling `futex_wait()` if no one is waiting.
```c
void lock(volatile int *lock) {
    while(try(lock) != 0) {
        // time to sleep ...
        futex_wait(lock, 1);
    }
}

void unlock(volatile int *lock) {
    *lock = 0;
    futex_wake(lock);
}
```

*Not very efficient - we want to avoid calling futex_wait() if no one is waiting.*
Using Linux futex or Sun park/unpark directly is error prone and not very portable.
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- `pthread_mutex_t` : structure that is the mutex
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- pthread_mutex_t: structure that is the mutex
- pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- `pthread_mutex_t`: structure that is the mutex
- `pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)`
- `pthread_mutex_destroy(pthread_mutex_t *mutex)`
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- `pthread_mutex_t` : structure that is the mutex
- `pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)`
- `pthread_mutex_destroy(pthread_mutex_t *mutex)`
- `pthread_mutex_lock(pthread_mutex_t *mutex)`
- `pthread_mutex_unlock(pthread_mutex_t *mutex)`
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- `pthread_mutex_t`: structure that is the mutex
- `pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)`
- `pthread_mutex_destroy(pthread_mutex_t *mutex)`
- `pthread_mutex_lock(pthread_mutex_t *mutex)`
- `pthread_mutex_unlock(pthread_mutex_t *mutex)`
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It's better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- `pthread_mutex_t`: structure that is the mutex
- `pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)`
- `pthread_mutex_destroy(pthread_mutex_t *mutex)`
- `pthread_mutex_lock(pthread_mutex_t *mutex)`
- `pthread_mutex_unlock(pthread_mutex_t *mutex)`
Using Linux futex or Sun park/unpark directly is error prone and not very portable.

It’s better to use the pthread library API, probably more efficient and definitely less problems.

Introducing pthread mutex locks:

- `pthread_mutex_t`: structure that is the mutex
- `pthread_mutex_init(pthread_mutex_t *mutex, ... *attr)`
- `pthread_mutex_destroy(pthread_mutex_t *mutex)`
- `pthread_mutex_lock(pthread_mutex_t *mutex)`
- `pthread_mutex_unlock(pthread_mutex_t *mutex)`

The lock procedure is platform specific, normally implemented as a combination of spinning and yield.
What could go wrong?

- Nothing works, will not even compile.
- Deadlock: the execution is stuck, no thread is making progress.
- Livelock: we're moving around in circles, all threads think that they are doing progress but we're stuck in a loop.
- Starvation: we're making progress but some threads are stuck waiting.
- Unfairness: we're making progress but some threads are given more of the resources.
What could go wrong?

- Nothing works, will not even compile.
What could go wrong?

- Nothing works, will not even compile.
- Deadlock: the execution is stuck, no thread is making progress.
What could go wrong?

- Nothing works, will not even compile.
- Deadlock: the execution is stuck, no thread is making progress.
- Livelock: we’re moving around in circles, all threads think that they are doing progress but we’re stuck in a loop.
What could go wrong?

- Nothing works, will not even compile.
- Deadlock: the execution is stuck, no thread is making progress.
- Livelock: we’re moving around in circles, all threads think that they are doing progress but we’re stuck in a loop.
- Starvation: we’re making progress but some threads are stuck waiting.
What could go wrong?

- Nothing works, will not even compile.
- Deadlock: the execution is stuck, no thread is making progress.
- Livelock: we’re moving around in circles, all threads think that they are doing progress but we’re stuck in a loop.
- Starvation: we’re making progress but some threads are stuck waiting.
- Unfairness: we’re making progress but some threads are given more of the resources.
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H:

M:

L:
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H: 

M: 

L:
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H: 

M: 

L: 

0 10 20 30 40 50 60 70 80 90 100 110 120
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H:

M:

L:
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H: 

M: 

L: 

takes lock
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H: takes lock
M: 
L: takes lock
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

H: 

M: 

L: 

takes lock
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

- **H**: takes lock
- **M**: 
- **L**: takes lock

suspends on lock
Assume we have a fixed priority scheduler, three processes with high (H), medium (M) and low (L) priority and one critical resource.

- **H:** takes lock
- **M:** suspends on lock
- **L:** takes lock
Some examples

- concurrent counter
- a list
- a queue
the concurrent counter

```c
struct counter_t {
    int val;
};

void incr(struct counter_t *c) {
    c->val++;
}
```
struct counter_t {
    int val;
}

void incr(struct counter_t *c) {
    c->val++;
}

struct counter_t {
    int val;
    pthread_mutex_t lock;
}

void incr(struct counter_t *c) {
    pthread_mutex_lock(c->lock);
    c->val++;
    pthread_mutex_unlock(c->lock);
}
Do the right thing

Doing the right thing often has a price.
Doing the right thing often has a price.
sloppy counter

thread 1

thread 2

Sloppy vs Speed - do the right thing.
sloppy counter
Sloppy vs Speed - do the right thing.
sloppy counter

counter

thread 1
local

thread 2
local

thread 3
local

Sloppy vs Speed - do the right thing.
sloppy counter

counter

thread 1

thread 2

thread 3

local

local

local
Sloppy vs Speed - do the right thing.
sloppy counter

0

counter

2
local

thread 1

1
local

thread 2

local

thread 3

Sloppy vs Speed - do the right thing.
sloppy counter

0

counter

2
local

thread 1

1
local

thread 2

1
local

thread 3

Sloppy vs Speed - do the right thing.
sloppy counter

![Diagram showing a counter with threads and local variables.]

- Thread 1: Counter = 3, Local = 3
- Thread 2: Counter = 1, Local = 1
- Thread 3: Counter = 1, Local = 1

Sloppy vs Speed - do the right thing.
sloppy vs speed - do the right thing.
sloppy counter

counter

thread 1

thread 2

thread 3

5 local

2 local

2 local

Sloppy vs Speed - do the right thing.
sloppy counter

counter

5

local

5

thread 1

local

2

thread 2

2

thread 3

Sloppy vs Speed - do the right thing.
sloppy counter

counter

thread 1

thread 2

thread 3
Sloppy vs Speed - do the right thing.
how about a list

*Simple solution: protect the list with one lock.*
how about a list

Simple solution: protect the list with one lock.

Concurrent solution: allow several thread to operate on the list concurrently.
Simple solution: protect the list with one lock.

Concurrent solution: allow several thread to operate on the list concurrently.

- concurrent reading: not a problem
how about a list

Simple solution: protect the list with one lock.

Concurrent solution: allow several thread to operate on the list concurrently.

- concurrent reading: not a problem
- concurrent updating: ....
Simple solution: protect the list with one lock.

Concurrent solution: allow several thread to operate on the list concurrently.

- concurrent reading: not a problem
- concurrent updating: .... hmm, how would you solve it?
What about a queue

Simple solution: protect the queue with one lock.

Concurrent solution: allow threads to add elements to the queue at the same time as other remove elements.
What about a queue

*Simple solution:* protect the queue with one lock.

*Concurrent solution:* allow threads to add elements to the queue at the same time as other remove elements.
What about a queue

*Simple solution:* protect the queue with one lock.

*Concurrent solution:* allow threads to add elements to the queue at the same time as other remove elements.
What about a queue

*Simple solution:* protect the queue with one lock.

*Concurrent solution:* allow threads to add elements to the queue at the same time as other remove elements.
What about a queue

Simple solution: protect the queue with one lock.

Concurrent solution: allow threads to add elements to the queue at the same time as other remove elements.
What about a queue

Simple solution: protect the queue with one lock.

Concurrent solution: allow threads to add elements to the queue at the same time as other remove elements.
What about a queue

*Simple solution:* protect the queue with one lock.

*Concurrent solution:* allow threads to add elements to the queue at the same time as other remove elements.
Traditionally operating systems were single threaded - the obvious solution.
Traditionally operating systems were single threaded - the obvious solution.

The first systems that operated on multi-cpu architectures used one big kernel lock to avoid any problems with concurrency.
Traditionally operating systems were single threaded - the obvious solution.

The first systems that operated on multi-cpu architectures used one big kernel lock to avoid any problems with concurrency.

An operating system that is targeting multi-core architectures will today be multi threaded and use fine grain locking to increase performance.
Traditionally operating systems were single threaded - the obvious solution.

The first systems that operated on multi-cpu architectures used one big kernel lock to avoid any problems with concurrency.

An operating system that is targeting multi-core architectures will today be multi threaded and use fine grain locking to increase performance.

How are things done in for example the JVM or Erlang?
beyond locks

The locks that we have seen are all right:

- We can take a lock and prevent others from obtaining the lock.
The locks that we have seen are all right:

- We can take a lock and prevent others from obtaining the lock.
- If someone holds the lock we will suspend execution.
The locks that we have seen are all right:

- We can take a lock and prevent others from obtaining the lock.
- If someone holds the lock we will suspend execution.
- When the lock is released we will wake up and try to grab the lock again.
The locks that we have seen are all right:

- We can take a lock and prevent others from obtaining the lock.
- If someone holds the lock we will suspend execution.
- When the lock is released we will wake up and try to grab the lock again.

We would like to suspend and only be woken up if a specified condition holds true.
What do we do now?
What do we do now?
What do we do now?
What do we do now?
Introducing pthread conditional variables: pthread_cond_t is the data structure of a conditional variable.

- pthread_cond_init(pthread_cond_t *restrict cond, ...) initializes the conditional variable.
- pthread_cond_destroy(pthread_cond_t *cond) destroys the conditional variable.
- pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) waits for a signal on the conditional variable.
- pthread_cond_signal(pthread_cond_t *cond) signals one waiting thread.
- pthread_cond_broadcast(pthread_cond_t *cond) signals all waiting threads.

The exact declarations are slightly more complicated, check the man pages.
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`

The exact declarations are slightly more complicated, check the man pages.
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`
- `pthread_cond_destroy(pthread_cond_t *cond)`
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`
- `pthread_cond_destroy(pthread_cond_t *cond)`
- `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`
- `pthread_cond_destroy(pthread_cond_t *cond)`
- `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `pthread_cond_signal(pthread_cond_t *cond)`
- `pthread_cond_broadcast(pthread_cond_t *cond)`

The exact declarations are slightly more complicated, check the man pages.
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`
- `pthread_cond_destroy(pthread_cond_t *cond)`
- `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `pthread_cond_signal(pthread_cond_t *cond)`
- `pthread_cond_broadcast(pthread_cond_t *cond)`

The exact declarations are slightly more complicated, check the man pages.
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`
- `pthread_cond_destroy(pthread_cond_t *cond)`
- `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `pthread_cond_signal(pthread_cond_t *cond)`
- `pthread_cond_broadcast(pthread_cond_t *cond)`
Introducing pthread conditional variables:

- `pthread_cond_t`: the data structure of a conditional variable
- `pthread_cond_init(pthread_cond_t *restrict cond, ...)`
- `pthread_cond_destroy(pthread_cond_t *cond)`
- `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- `pthread_cond_signal(pthread_cond_t *cond)`
- `pthread_cond_broadcast(pthread_cond_t *cond)`

*The exact declarations are slightly more complicated, check the man pages.*
A single element buffer, multiple consumers, multiple producers.

```c
int buffer;
int count = 0;
```

Let's try to make this work.
A single element buffer, multiple consumers, multiple producers.

```c
int buffer;
int count = 0;

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```
A single element buffer, multiple consumers, multiple producers.

```c
int buffer;
int count = 0;

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

Let’s try to make this work.
void produce(int val) {
    put(val);
}

int consume() {
    int val = get();
    return val;
}

this will not work
add a mutex and cond variable

```c
pthread_cond_t cond;
pthread_mutex_t mutex;
```
add a mutex and cond variable

```c
/*
 * This code uses a mutex and a condition variable to synchronize
 * between two threads: a producer and a consumer.
 */

#include <pthread.h>

int count = 0;

void* produce(void*) {
    int val = get();
    put(val);
    pthread_cond_signal(&cond);
    return NULL;
}

void* consume(void*) {
    int val = get();
    put(val);
    pthread_cond_signal(&cond);
    return NULL;
}

int main() {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_t producer, consumer;
    if (pthread_create(&producer, NULL, produce, NULL) != 0) {
        return 1;
    }
    if (pthread_create(&consumer, NULL, consume, NULL) != 0) {
        return 1;
    }

    pthread_join(producer, NULL);
    pthread_join(consumer, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    return 0;
}
```

When does this work, when does it not work?
add a mutex and cond variable

```c
pthread_cond_t cond;
pthread_mutex_t mutex;

produce(int val) {
    pthread_mutex_lock(&mutex);
    if (count == 1)
        pthread_cond_wait(&cond, &mutex);
    put(val);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

int consume() {
    pthread_mutex_lock(&mutex);
    if (count == 0)
        pthread_cond_wait(&cond, &mutex);
    int val = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return val;
}
```
add a mutex and cond variable

```
pthread_cond_t cond;
pthread_mutex_t mutex;

produce(int val) {
    pthread_mutex_lock(&mutex);
    if (count == 1)
        pthread_cond_wait(&cond, &mutex);
    put(val);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

int consume() {
    pthread_mutex_lock(&mutex);
    if (count == 0)
        pthread_cond_wait(&cond, &mutex);
    int val = get();
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return val;
}
```

When does this work, when does it not work?
If you’re signaled to wake up - it might take some time before you do wake up.
```c
pthread_cond_t filled, empty;
pthread_mutex_t mutex;
```
better

```c
pthread_cond_t filled, empty;
pthread_mutex_t mutex;

produce(int val) {
    pthread_mutex_lock(&mutex);
    while (count == 0)
        pthread_cond_wait(&empty, &mutex);
    :
    pthread_cond_signal(&filled);
    :
}

consume() {
    pthread_mutex_lock(&mutex);
    while (count == 1)
        pthread_cond_wait(&empty, &mutex);
    :
    pthread_cond_signal(&empty);
    :
}
```
```c
#include <pthread.h>

pthread_cond_t filled, empty;
pthread_mutex_t mutex;

int produce(int val) {
pthread_mutex_lock(&mutex);
while (count == 1)
    pthread_cond_wait(&empty, &mutex);
: 
pthread_cond_signal(&filled);
:
}

int consume() {
pthread_mutex_lock(&mutex);
while (count == 0)
    pthread_cond_wait(&filled, &mutex);
: 
pthread_cond_signal(&empty);
:
}
```
int buffer[MAX];
int *getp = 0;
in *putp = 0;
int count = 0;

void put(int value) {
    assert(count < MAX);
    buffer[putp] = value;
    putp = putp + 1 % MAX;
    count++;
}

int get() {
    assert(count > 0);
    int val = buffer[getp];
    getp = getp + 1 % MAX
    count--
    return val;
}
produce(int val) {
    :
    while(count == MAX)
        pthread_cond_wait(&empty, &mutex);
    :
}
produce(int val) {
    while(count == MAX)
        pthread_cond_wait(&empty, &mutex);
}

int consume() {
    while(count == 0)
        pthread_cond_wait(&filled, &mutex);
}
produce(int val) {
    :
    while(count == MAX)
        pthread_cond_wait(&empty, &mutex);
    :
}

int consume() {
    :
    while(count == 0)
        pthread_cond_wait(&filled, &mutex);
    :
}

Can we allow a producer to add an entry while another removes an entry?
atomic test and set: we need it
Where are we now?

- atomic test and set: we need it
- spin locks: simple to use but have some problems
Where are we now?

- atomic test and set: we need it
- spin locks: simple to use but have some problems
- wait and wake: avoid spinning
Where are we now?

- atomic test and set: we need it
- spin locks: simple to use but have some problems
- wait and wake: avoid spinning
- condition variables: don’t wake up if it’s not time to continue
Where are we now?

- atomic test and set: we need it
- spin locks: simple to use but have some problems
- wait and wake: avoid spinning
- condition variables: don’t wake up if it’s not time to continue

Is there more?
Semaphores

Properties of a semaphore:
- Holds a number
- Only allow threads to pass if number is above 0
- Passing threads decremented the number
- A thread can increment the number

A semaphore is a counter of resources.
Properties of a semaphore:
Properties of a semaphore:
- holds a number
Properties of a semaphore:

- holds a number
- only allow threads to pass if the number is above 0
Properties of a semaphore:

- holds a number
- only allow threads to pass if number is above 0
- passing threads decrement the number
Semaphores

Properties of a semaphore:
- holds a number
- only allow threads to pass if number is above 0
- passing threads decrement the number
- a thread can increment the number
Semaphores

Properties of a semaphore:
- holds a number
- only allow threads to pass if number is above 0
- passing threads decrement the number
- a thread can increment the number

A semaphore is a counter of resources.
POSIX semaphores

- `#include <semaphore.h>`
POSIX semaphores

- `#include <semaphore.h>`
- `sem_t`: the semaphore data structure
#include <semaphore.h>

sem_t: the semaphore data structure

sem_init(sem_t *sem, int pshared, unsigned int value): could be shared between processes
# include <semaphore.h>

- sem_t: the semaphore data structure
- sem_init(sem_t *sem, int pshared, unsigned int value): could be shared between processes
- int sem_destroy(sem_t *sem)
#include <semaphore.h>

- `sem_t`: the semaphore data structure
- `sem_init(sem_t *sem, int pshared, unsigned int value)`: could be shared between processes
- `int sem_destroy(sem_t *sem)`
- `sem_wait(sem_t *sem)`
#include <semaphore.h>

- `sem_t`: the semaphore data structure
- `sem_init(sem_t *sem, int pshared, unsigned int value)`: could be shared between processes
- `int sem_destroy(sem_t *sem)`
- `sem_wait(sem_t *sem)`
- `sem_post(sem_t *sem)`
POSIX semaphores

- `#include <semaphore.h>`
- `sem_t`: the semaphore data structure
- `sem_init(sem_t *sem, int pshared, unsigned int value)`: could be shared between processes
- `int sem_destroy(sem_t *sem)`
- `sem_wait(sem_t *sem)`
- `sem_post(sem_t *sem)`
Happy-Go-Lucky

"Vastly entertaining and very funny"

"Fresh and funny, joyful and life-affirming"

"A joy to behold"

"A delight"

Winner Best Actress

Sally Hawkins
Edith MacComack
Alex Zoppan