# Locks and semaphores

Johan Montelius

KTH

2020



## 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);
```

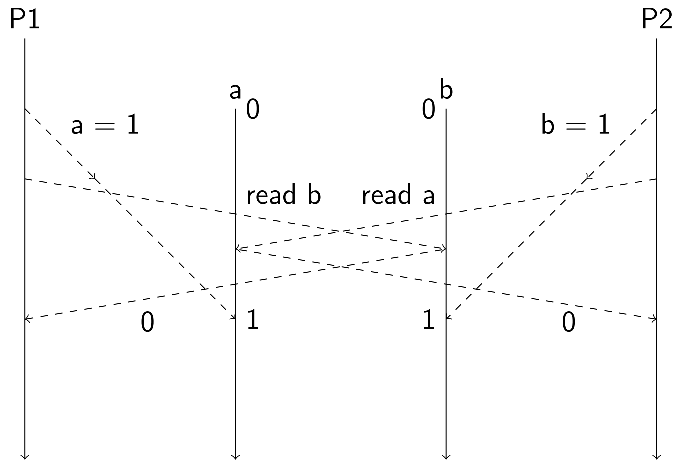


## Peterson's algorithm

```
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



## 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.

- `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

```
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)
popq     %rbp
ret
```

*This is using GCC extensions to C, similar extensions available in all compilers.*



## a spin-lock

```
int lock(int *lock) {

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

  return 1;
}

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



## finally - we're in control

```
int global = 0;

int count = 0;

void *hello(void *name) {
  for(int i = 0; i < 10; i++) {
    lock(&global);
    count++;
    release(&global);
  }
}
```

*try using* `taskset -c 1 ./spin 10000`

## spin locks



## avoid spinning

We need to talk to the operating system.

```
void lock(int *lock) {

  while(try(lock) != 0) {
    sched_yield();    // in Linux
  }

}
```



## Wham - .....

*For how long should we sleep?*

*We would like to be woken up as the lock is released - before you go-go.*



## a detour in Sun Solaris

```
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

It's not easy to 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 == 1 */                         m->flag = 0;
      :                                  } else {
queue_add(m->queue, gettid());             // don't reset the flag
setpark();                                 unpark(dequeue(m->queue));
// if somone unparks now my park() is a noop
                                         }
m->guard = 0;
park();
```



## back to Linux

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 treads suspended on the `mutex`

*In GCC you have to call them using a* `syscall()`



## a futex lock

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

*Not very efficient - we want to avoid calling* `futex_wake()` *if no one is waiting.*



## pthread 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.*

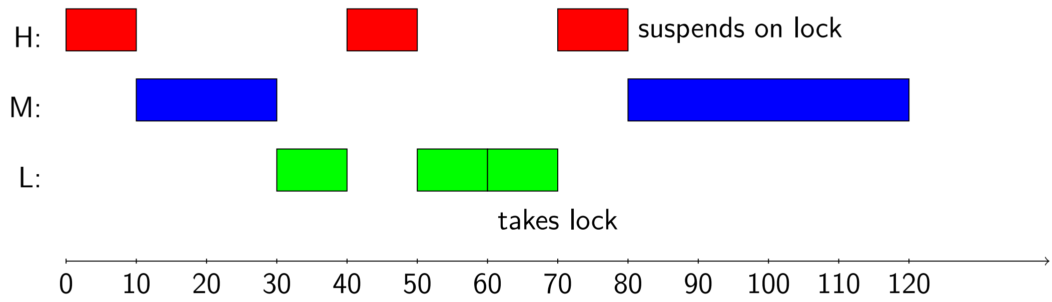
## What could go wrong?

- 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.



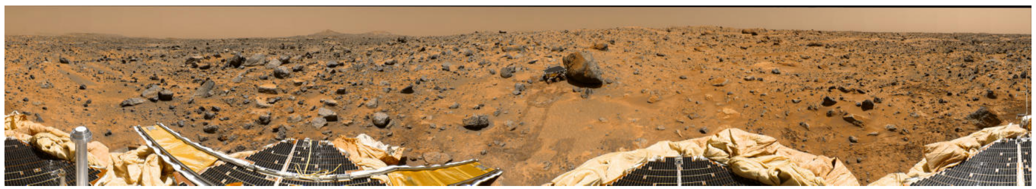
## Resources, priorities and scheduling

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






## Mars Pathfinder and Priority Inversion





## Some examples

- concurrent counter
- a list
- a queue

## the concurrent counter

```
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_lock(c->lock);
  c->val++;
  pthread_unlock(c->lock);
}
```

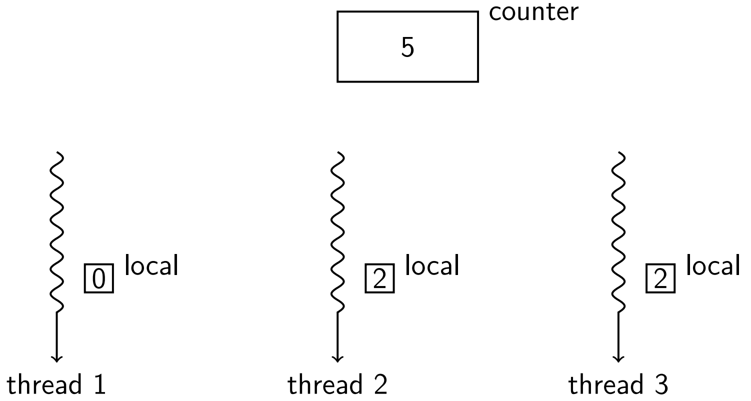


## Do the right thing

*Doing the right thing often has a price.*



## sloppy counter




*Sloppy vs Speed - do the right thing.*



## how about a list

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

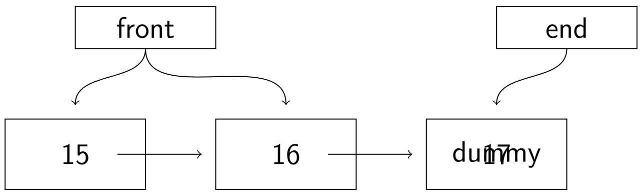
*Concurrent solution: allow several thread to operate on the list concurrently.*

- concurrent reading: not a problem
- concurrent updating: .... 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.*






## an operating system

*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

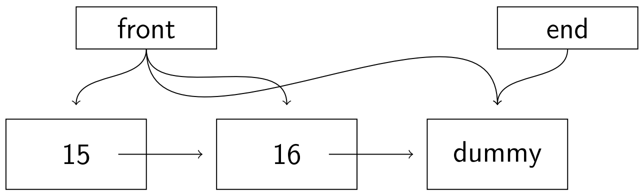
The locks that we have seen are all right:

- We can take a lock and prevent others from obtaining the lock.
- 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.



## the queue revisited




What do we do now?

## conditional variables

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.*



## the producer/consumer

A single element buffer, multiple consumers, multiple producers.

```
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.*



## this will not work

```
void produce(int val) {
    put(val);
}
```

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



## add a mutex and cond variable

```
pthread_cond_t cond;
pthread_mutex_t mutex;
```

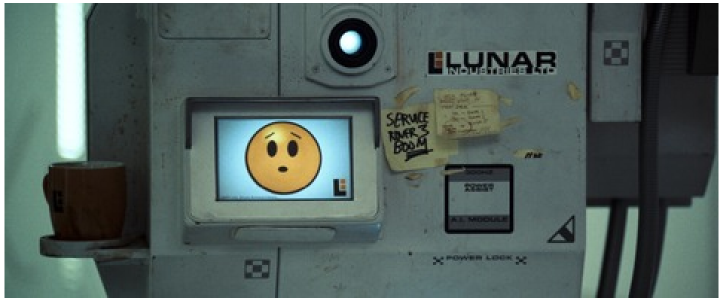
```
produce(int val) {
  pthread_mutex_lock(&mutex);
  if(count == 1)
    pthread_cond_wait(&cond, &mutex);
  put(val);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

```
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?*

## a race condition

If you're signaled to wake up - it might take some time before you do wake up.






## better

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

```
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);
  :
}
```



## a larger buffer

```
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;
}
```



## final touch

```
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?*

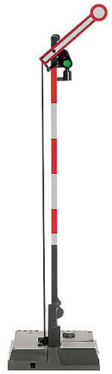
## Where are we now?

- 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 is number is above 0
- passing threads decremented the number
- a thread can increment the number

*A semaphore is a counter of resources.*



## 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)`



## Summary