Principper for Samtidighed og Styresystemer

Concurrency

René Rydhof Hansen

09 APR 2018

Next time...

XV6 hands-on exercise (v2): no lecture, no new exercises.

Learning Goals

After the last lecture, you are able to

- ... define and explain paging and how paged memory works
- ... perform simple address translation from paged (virtual) memory to physical memory
- ... explain how paged memory supports shared memory
- ... explain organisation of page tables (direct, two-level)
- ... define, explain, and discuss various page replacement algorithms and their pros and cons, including:
  - OPT (the optimal algorithm)
  - FIFO
  - Random
  - LRU
  - Clock

Learning Goals

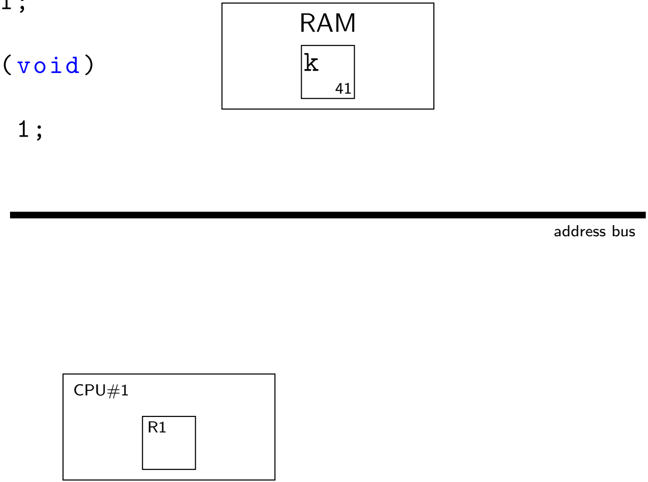
After today's lecture you

- ... can define what a race condition is
- ... can explain how mutual exclusion can be used to avoid race conditions
- ... can explain strategies for achieving and implementing mutual exclusion
- ... can define mutex, semaphore, and monitor and explain how they work and where they are useful
- ... can explain how to synchronise two (or more) threads and why it may be necessary

Race Conditions

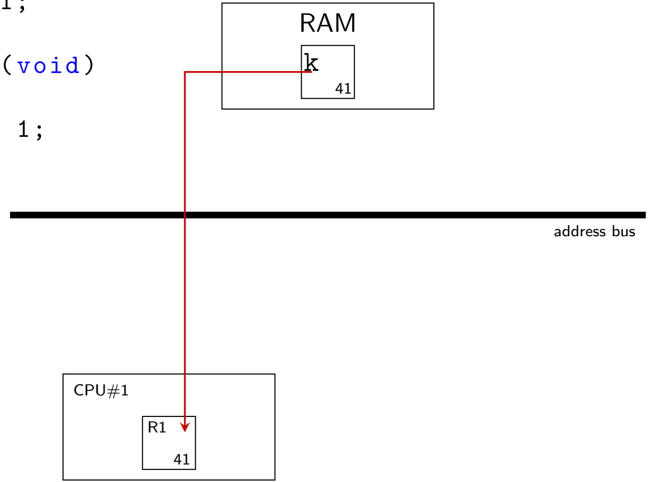
Shared Memory Communication: An Example

```
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



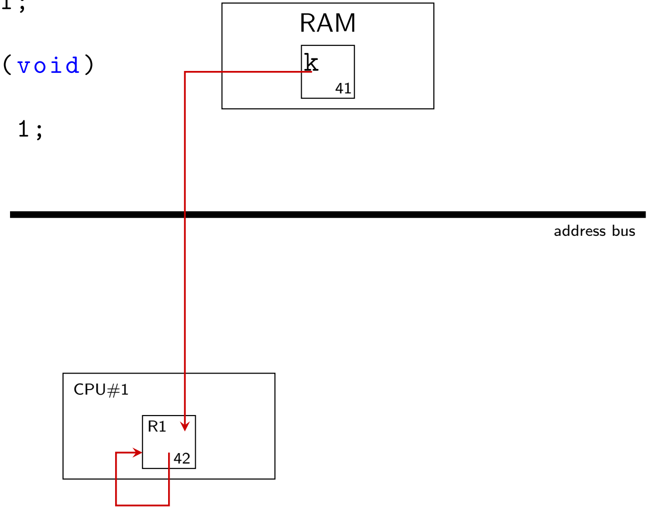
Shared Memory Communication: An Example

```
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



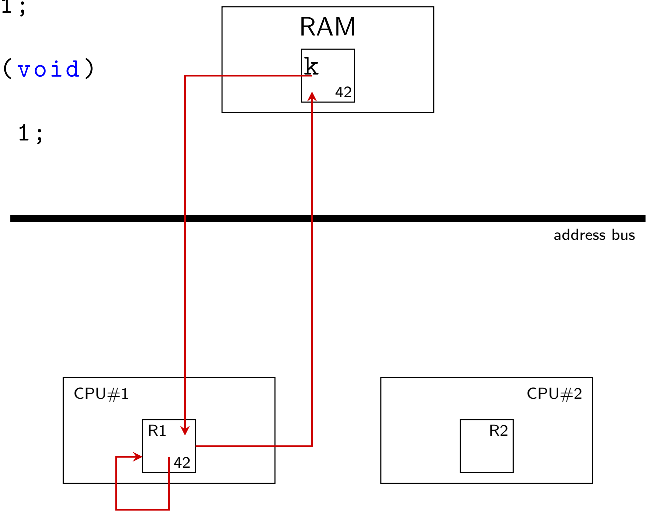
Shared Memory Communication: An Example

```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



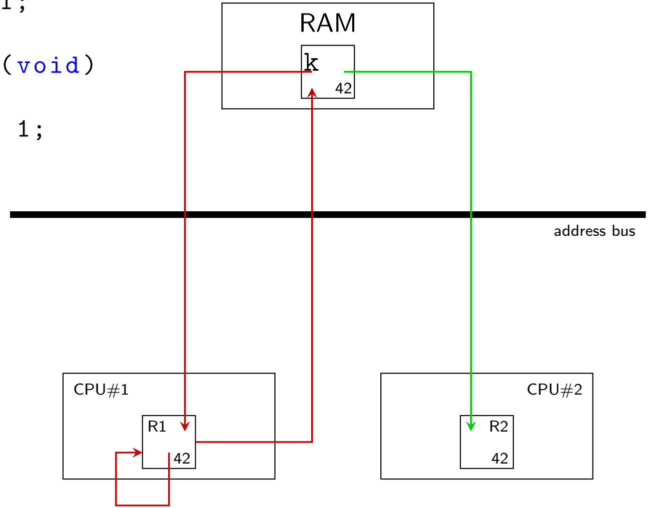
Shared Memory Communication: An Example

```
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



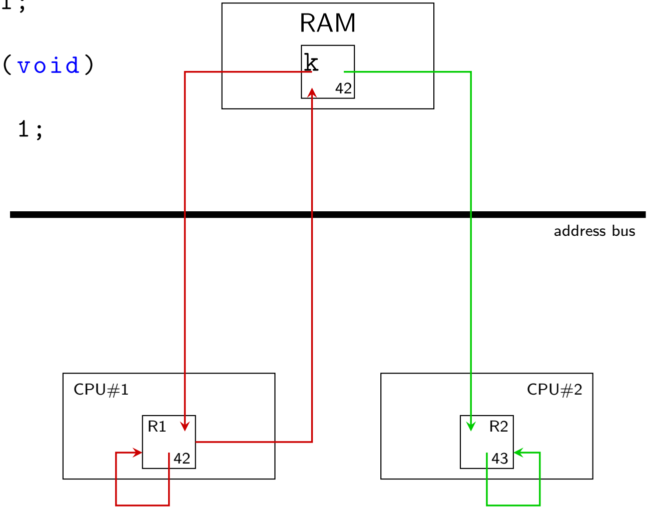
Shared Memory Communication: An Example

```
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



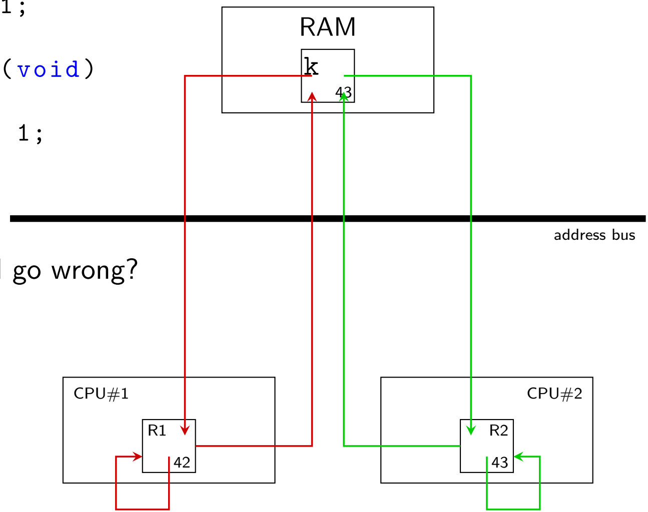
Shared Memory Communication: An Example

```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



Shared Memory Communication: An Example

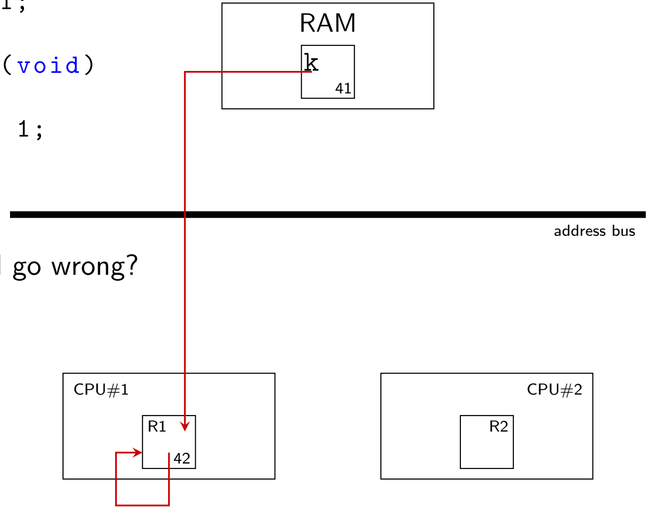
```
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



What could go wrong?

Shared Memory Communication: An Example

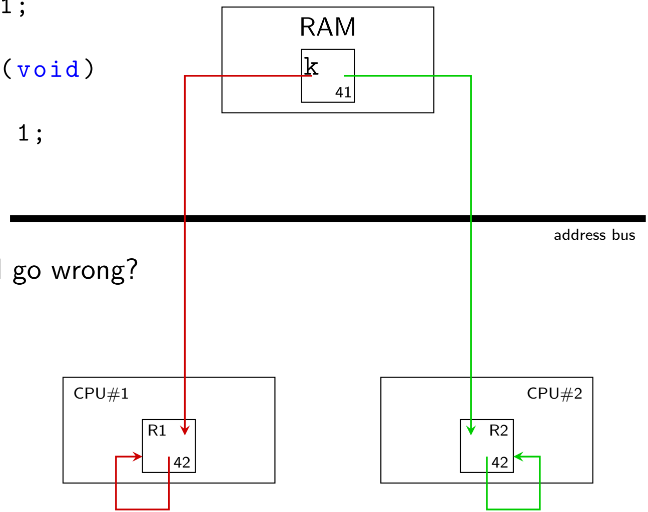
```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



What could go wrong?

Shared Memory Communication: An Example

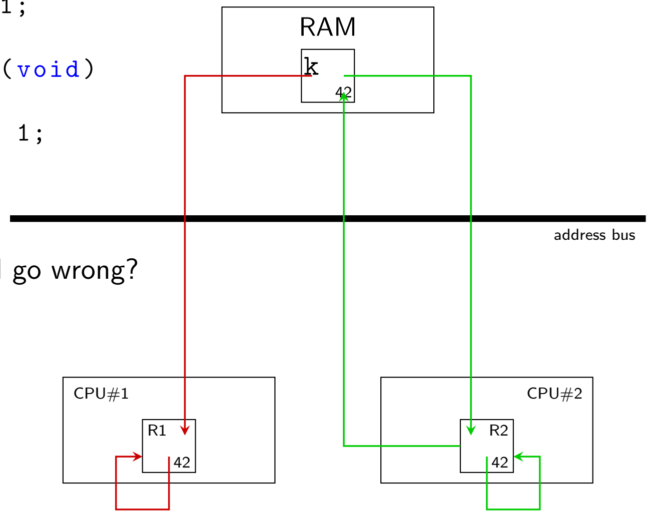
```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



What could go wrong?

Shared Memory Communication: An Example

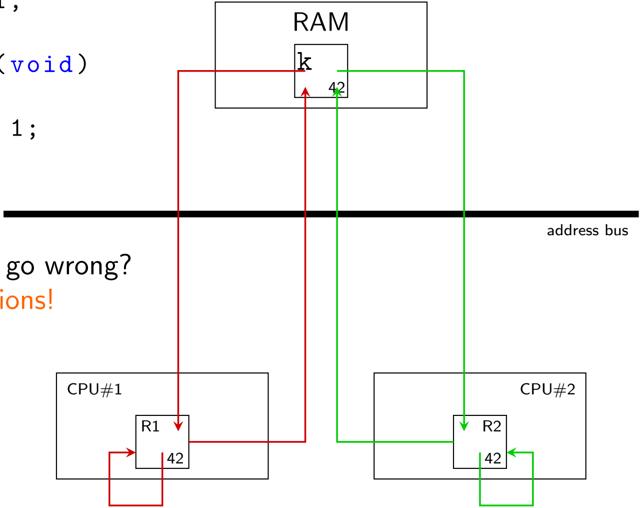
```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



What could go wrong?

Shared Memory Communication: An Example

```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



What could go wrong?

Race conditions!

Concurrency — Important Concepts

- Race condition
  - When the result of a computation depends on the relative speed of the individual threads
  - In other words: the result depends on the actual interleaving of the threads
  - Hard to debug
- Critical region (critical section)
  - Program fragment vulnerable to race conditions
  - Danish: "kritisk region"

Critical regions must be executed under mutual exclusion

- Mutual exclusion (mutex)
  - When only one thread (among many) can access a given resource or execute specific part of the program-text
  - Danish: "gensidig udelukkelse"
- Atomic
  - Event, or sequence of events, that happen(s) uninterruptedly

Ensuring Mutual Exclusion

How to ensure mutual exclusion?

Locks!

- Ensure only one thread has access to critical region
- “Locking” critical regions

Evaluation Criteria for Locks

- Should guarantee mutual exclusion (i.e., it should work)
- Should be fair
- Should be efficient

How to ensure mutual exclusion?

Solution: Disabling Interrupts

- Disable interrupts before entering critical region
- Effective... but dangerous
- Works for tiny and predictable critical regions only
- Mostly relevant for very low level code, i.e., kernel code
- Difficult to incorporate in schedulability analysis, RTSs
- What about interrupts on multi-processor systems?

Code: Mutual exclusion by interupt disabling

```
void  *thread(void *tid) {
  /* ... */
  cli();
  /* critical region */
  sti();
  /* ... */
}
```

How to ensure mutual exclusion?

Solution: Software locks

- Use “flag” to indicate entry into critical region
  1. Check if lock is open/closed, acquire lock
  2. Access resources (protected by the lock)
  3. Release lock

Code: Programming with locks

```
lock_t  lock;

void  *thread(void *tid) {
  /* ... */

  acquire_lock(&lock);
  /* critical region */
  release_lock(&lock);

  /* ... */
}
```

Implementing locks

Lock variables

```
bool flag[2] = {false, false};
```

```
void *thread0(void *tid)
{
  while(flag[1]) { }
  flag[0] = true;
  /* critical region */
  flag[0] = false;
}
```

```
void *thread1(void *tid)
{
  while(flag[0]) { }
  flag[1] = true;
  /* critical region */
  flag[1] = false;
}
```

Implementing locks

Lock variables

```
bool flag[2] = {false, false};
```

```
void *thread0(void *tid)
{
  while(flag[1]) { }
  flag[0] = true;
  /* critical region */
  flag[0] = false;
}
```

```
void *thread1(void *tid)
{
  while(flag[0]) { }
  flag[1] = true;
  /* critical region */
  flag[1] = false;
}
```

The “algorithm” is wrong!

Tricky to implement

Implementation of locks susceptible to race conditions: when the `while`-loop ends, the “competing” process can take over again

Implementing locks

A working lock implementation: Dekker's Algorithm (1965)

```
bool flag[2] = {false, false};
int turn = 0;
```

```
void *thread0(void *) {
  flag[0] = true;
  while flag[1] {
    if(turn == 1) {
      flag[0] = false;
      while(turn == 1) { }
      flag[0] = true;
    }
  }
  /* critical region */
  turn = 1;
  flag[0] = false;
}
```

```
void *thread1(void *) {
  flag[1] = true;
  while flag[0] {
    if(turn == 0) {
      flag[1] = false;
      while(turn == 0) { }
      flag[1] = true;
    }
  }
  /* critical region */
  turn = 0;
  flag[1] = false;
}
```

Implementing locks (Dekker's Algorithm)

The Good

- It works!
- Efficient when blocking is not needed

The Bad

- Not very efficient in general
- A compiler may allocate some variables in registers (thus not shared between threads)
- Difficult (impossible?) to scale to more than two processes

Implementing locks: a better implementation

Peterson's mutex-algorithm (1981)

```
bool flag[2] = {false, false};
int  turn;
 void *thread0(void *) {          void *thread1(void *) {
   flag[0] = true;                  flag[1] = true;
   turn = 1;                        turn = 0;
   while(flag[1] &&                 while(flag[0] &&
         turn == 1) { }                   turn == 0) { }
   /* critical region */            /* critical region */
   flag[0] = false;                 flag[1] = false;
 }                                }
```

The Good and the Bad

- It works!
- Scales to more processes (... but still generally inefficient)
- Guarantees fairness
- Vulnerable to compilers optimising memory access

Implementing locks

The Ugly

Memory consistency on modern hardware?

How to ensure mutual exclusion?

Hardware supported locks

- Similar to software locks, but exploit special hardware
- Atomic instructions

Code: Programming with locks

```
int  lock;

void  *thread(void *tid) {
  /* ... */

  while( locked (&lock) == 1) {}
  /* critical region */
  lock = 0;

  /* ... */
}
```

Hardware supported locks: test-and-set

The ‘test-and-set’ instruction

- Return old value and set to 1
- Executed atomically

Code: Informal semantics of test-and-set

```
int test_and_set (int *old_ptr) {
  int old = *old_ptr;
  *old_ptr = 1;
  return old;
}
```

Hardware supported locks: test-and-set

Code: Mutual exclusion using test-and-set

```c
int lock = 0;
```

```c
void *thread0(void *) {
  /* ... */
  while(t_a_s(&lock) == 1)
  {
    /* do nothing */
  }
  /* critical region */
  lock = 0;
  /* ... */
}
```

```c
void *thread1(void *) {
  /* ... */
  while(t_a_s(&lock) == 1)
  {
    /* do nothing */
  }
  /* critical region */
  lock = 0;
  /* ... */
}
```

Hardware supported locks: xchg

The 'xchg' (exchange) instruction

- Swap arguments atomically

Code: Informal semantics of xchg

```c
int xchg(int *old_ptr, int new) {
  int old = *old_ptr;
  *old_ptr = new;
  return old;
}
```

Hardware supported locks: xchg

Code: Mutual exclusion using xchg (from XV6)

```
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes...
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

Hardware supported locks: compare-and-swap

The 'compare-and-swap' instruction

- Compare old value to test value, update if equal and return old value
- Executed atomically

Code: Informal semantics of compare_and_swap

```
int compare_and_swap(int *ptr, int expected, int new) {
  int actual = *ptr;
  if (actual == expected)
    *ptr = new;
  return actual;
}
```

Hardware supported locks: compare-and-swap

Code: Mutual exclusion using compare-and-swap

```c
int lock = 0;
void *thread0(void *) {
  /* ... */
  while(c_a_s(&lock,0,1)
        == 1)
  {
    /* do nothing */
  }
  /* critical region */
  lock = 0;
  /* ... */
}
```

```c
void *thread1(void *) {
  /* ... */
  while(c_a_s(&lock,0,1)
        == 1)
  {
    /* do nothing */
  }
  /* critical region */
  lock = 0;
  /* ... */
}
```

Hardware supported locks

The 'fetch-and-add' instruction

- Atomically increment a variable and return old value

Code: Informal semantics for fetch_and_add

```
int fetch_and_add(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

Hardware supported locks

Code (from [OSTEP])

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

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

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

void unlock(lock_t *lock) {
  lock->turn = lock->turn + 1;
}
```

Hardware supported locks

- Where do these instructions come from?

Code (from XV6)

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

How to ensure mutual exclusion!

Solution summary

1. Disabling interrupts
   - Pro's: effective
   - Con's: too effective, potentially very disruptive
   - Major problem: useless for multi-processor systems
2. Software locks
   - Pro's: portable, less disruptive
   - Con's: inefficient, vulnerable to compiler optimisation
   - Major problem: may not work with (lack of) memory consistency on modern CPUs
3. Hardware locks
   - Pro's: (more) efficient
   - Con's: less portable... still inefficient(!)

Busy Waiting

Definition (Busy Wait)

A "no-op" loop waiting for a condition to be fulfilled: a loop hvor the executing thread actively waits for access to the critical region

Example (busy wait with test-and-set)

```
int mutex = 0;

void *thread0(void *tid) {

  while(test_and_set(&mutex)==1) {
    /* busy wait */
  }
  /* critical region */
  mutex = 0;
}
```

Busy Waiting

Definition (Busy Wait)

A "no-op" loop waiting for a condition to be fulfilled: a loop hvor the executing thread actively waits for access to the critical region

Example (solution: yield)

```
int mutex = 0;

void *thread0(void *tid) {

  while(test_and_set(&mutex)==1) {
    yield();
  }
  /* critical region */
  mutex = 0;
}
```

Busy Waiting

Definition (Busy Wait)

A "no-op" loop waiting for a condition to be fulfilled: a loop hvor the executing thread actively waits for access to the critical region

Example (solution: block)

```
int mutex = 0;

void *thread0(void *tid) {

  while(test_and_set(&mutex)==1) {
    /* blocking action */
  }
  /* critical region */
  mutex = 0;
}
```

A better alternative to Busy Wait: Blocking action

Idea: let another thread work while we wait

- Block the thread until access to the resource can be achieved
- Wake the thread when leaving the critical region
- Leave the hard work to the OS

Example

```
void *thread(void *tid) {

  block_until_access();
  /* critical region */
  release_ressource();

}
```

A better alternative to Busy Wait: Blocking action

Blocked thread

- Thread that does not run until it receives a signal from another thread
- Special state: blocked
- Not allocated CPU time
- OS keeps account of blocked threads and why they are blocked

Blocking action

- An action that blocks the thread until the action is completed
- System calls are (mostly) blocking actions (I/O is handled through system calls)

Semaphores

Structured high-level combination of blocking action and mutual exclusion

Programming with Mutual Exclusion

Mutexes, aka. (very) simple semaphores

Mutex

- Two states (binary): locked and unlocked
- `lock`: locks an unlocked mutex (what if mutex is already locked?)
- `unlock`: unlocks a locked mutex (what if mutex was already unlocked?)

Mutex vs. semaphore

- Mutex: `lock`/`unlock` only in same thread (not always)
- Mutex: binary semaphore

Why mutexes?

- Often specifically supported
- (More) often supported in hardware
- Can be used to implement general semaphores

Semaphores

Mutexes

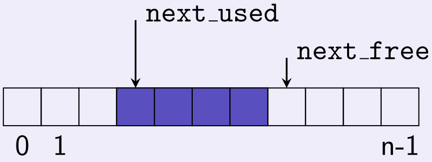
- How to handle multiple threads in the same critical region?

Semaphores

Mutexes

- How to handle multiple threads in the same critical region?

Example (Bounded buffer, producer/consumer)



Example

- Single writer
- Several readers

Semaphores

Definition (Semaphore)

- Non-negative integer variable with atomic increment and decrement operations
- `init(sem,n)`: Initialise the semaphore with value `n`
- `acquire(sem)`: Decrement semaphore; block if already zero
- `release(sem)`: Increment semaphore; wake sleeping thread (if any); what happens to the value of the semaphore?
- Semaphore (?!) must maintain threads blocked on `wait`

Properties of Semaphores

- Sometimes supported in hardware: `test-and-decrement`
- May be implemented using mutex
- Semaphore maintains (priority-)queue with blocked threads
- Order of "wake up" may result in fairness problems (example: weak semaphores in Java)

Semaphores in pthreads

Semaphore operations

- Declaration

```
sem_t sem;
```

- Initialisation (n = initial value, 0 = share among threads, not processes):

```
sem_init(&sem,0,n);
```

- Wait

```
sem_wait(&sem);
```

- Signal

```
sem_post(&sem);
```

Properties

- `sem_wait()` and `sem_post()` do not have to be in same thread
- `sem_post()` will always increase semaphore (even beyond initial value)
- Remember to `sem_post()` (often and early)

Semaphores in pthreads

Example (Controlling multiple readers)

```
#include <semaphore.h>

sem_t reader_sem;

void *read(void *tid) {
  sem_wait(&reader_sem);
  /* read something shared */
  sem_post(&reader_sem);
}

int main() {
  sem_init(&reader_sem,0,42);

  for(i = 0; i < 100; i++) pthread_create(...);
}
```

Mutual Exclusion with semaphores (mutex)

```
sem_init(&reader_sem,0,1);

void *t0(void *tid) {
  sem_wait(&reader_sem);
  /* critical region */
  sem_post(&reader_sem);
}
void *t1(void *tid) {
  sem_wait(&reader_sem);
  /* critical region */
  sem_post(&reader_sem);
}
void *t2(void *tid) {
  sem_wait(&reader_sem);
  /* critical region */
  sem_post(&reader_sem);
}
```

- Ordering is undefined
- Weak semaphores: no guarantee that `sem_wait()` ever returns (starvation)
- Strong semaphores guarantee that `sem_wait()` will return (sooner or later) unless there are no calls to `sem_post()` (fairness)
- Typical weak semaphore (mutex): `test-and-set-based`

Synchronisation

What can semaphores/mutexes be used for?

Example

```
float T;
sem_t ready;
sem_init(&ready,0,?);

void *thread0(void *tid)
{
  T = read_sensor();

}

void *thread1(void *tid)
{

  massive_computation(T);
  output();
}
```

What can semaphores/mutexes be used for?

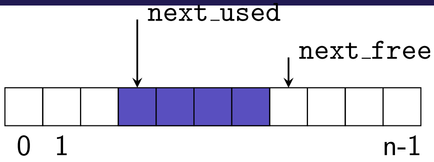
Example

```
float T;
sem_t ready;
sem_init(&ready,0,0);

void *thread0(void *tid)
{
  T = read_sensor();
  sem_post(&ready);
}

void *thread1(void *tid)
{
  sem_wait(&ready)();
  massive_computation(T);
  output();
}
```

- Ensure that `read_sensor` happens before `massive_computation`
- Synchronises the relative time of the two threads
- Mutual exclusion? ... also synchronisation!

Bounded buffer, producer/consumer

```
sem_t used, free;
int buffer[n],
  next_used = 0,
  next_free = 0;

int main() {
 sem_init(&used,0,0);
 sem_init(&free,0,n);

}
```

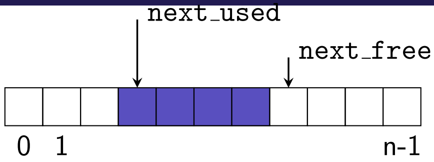



```
void *PROD() {
  while(true) {
    sem_wait(&free);

    buffer[next_free] = data;
    next_free = (next_free + 1) % n;

    sem_post(&used);
  }
}
```

```
void *CONSUMER() {
  while(true) {
    sem_wait(&used);

    data = buffer[next_used];
    next_used = (next_used + 1) % n;

    sem_post(&free);
  }
}
```

Bounded buffer, producer/consumer

```
sem_t used, free;
int buffer[n],
  next_used = 0,
  next_free = 0;

int main() {
 sem_init(&used,0,0);
 sem_init(&free,0,n);

}
```

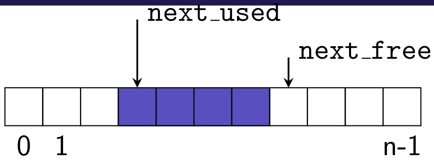



```
void *PROD() {
  while(true) {
    sem_wait(&free);

    buffer[next_free] = data;
    next_free = (next_free + 1) % n;

    sem_post(&used);
  }
}
```

```
void *CONSUMER() {
  while(true) {
    sem_wait(&used);

    data = buffer[next_used];
    next_used = (next_used + 1) % n;

    sem_post(&free);
  }
}
```

Bounded buffer, producer(s)/consumer(s)

```
sem_t used, free;
int buffer[n],
  next_used = 0,
  next_free = 0;

int main() {
 sem_init(&used,0,0);
 sem_init(&free,0,n);
 sem_init(&mutex,0,1);
}
```




```
void *PROD() {
  while(true) {
    sem_wait(&free);
    sem_wait(&mutex);
    buffer[next_free] = data;
    next_free = (next_free + 1) % n;
    sem_post(&mutex);
    sem_post(&used);
  }
}
```

```
void *CONSUMER() {
  while(true) {
    sem_wait(&used);
    sem_wait(&mutex);
    data = buffer[next_used];
    next_used = (next_used + 1) % n;
    sem_post(&mutex));
    sem_post(&free);
  }
}
```

Monitor

Problem: semaphores/mutexes low-level, easy to make mistakes

- Solution: make a language construct (called a monitor) ensuring mutual exclusion by encapsulating and ensuring atomicity of
  - Variables (only through access methods)
  - Access methods (executed under mutual exclusion)
  - Initialisation

```
monitor Counter
{
 int i = 0;

 void increment()
 {
  i++;
 }
 ...
}
```

```
void thread()
{
 for(int j = 0; j < 10000;j++)
 {
  Counter.increment();
 }
}
t1 = run(thread);
t2 = run(thread);
wait(t1);
wait(t2);
Counter.print();
```

Monitors

The compiler guarantees mutual exclusion on the entire monitor

```
monitor Counter
{
 int i = 0;

 void increment()
 {
  i++;
 }

 void decrement()
 {
  i--;
 }

 ...
}
```

```
void thread()
{
 for(int j = 0; j < 10000;j++)
 {
  Counter.increment();
  Counter.decrement();
 }
}

t1 = run(thread);
t2 = run(thread);
wait(t1);
wait(t2);
Counter.print();
```

Bounded buffer with monitors in Java

```
class BoundedBuffer {
 int buffer[], free, used, n;

 BoundedBuffer(int elements) {
  n = elements;
  buffer = new int[n];
  next_used = 0;
  next_free = 0;
  free = n;
  used = 0;
 }
synchronized int get() {
 int data;
 while(used == 0) wait();
 data = buffer[next_used];
 next_used = (next_used + 1) % n;
 free = free + 1;
 used = used - 1;
 notifyAll();
 return data;
}
 synchronized void put(int data) {
  while(free == 0) wait();
  buffer[next_free] = data;
  next_free = (next_free + 1) % n;
  free = free - 1;
  used = used + 1;
  notifyAll();
 }
}
```

- Note: `while(...) wait()` are not busy-waiting: `wait()` only returns on call of `notify()`/`notifyAll()`
- Monitors in Java: (simple) extension of objects
- Every object in Java has builtin mutex-lock
- To call `synchronized` methods thread must first acquire object lock

Bounded buffer with monitors in Java

```
void init()
{
 buffer = new BoundedBuffer(n);
}
```

```
void producer()
{
 while(true)
 {
  buffer.put(data);
 }
}
```

```
void consumer()
{
 while(true)
 {
  data = buffer.get();
 }
}
```

Monitor vs. semaphore (vs. mutex)

Monitor

- Language construct ensuring mutual exclusion
- Encapsulates critical regions
- Guaranteed by the compiler

Semaphore (mutex)

- Can be used by programmer to ensure mutual exclusion
- The `goto` of thread-programming

If structured mechanisms are available, they should be preferred (even if they are not always easier to use)

Other approaches

Condition variables

- Useful for blocked waiting on condition
- Avoids active polling

Barriers

- Useful for synchronising several (many) threads

Lock-free data structures

- Easy to use
- Efficient
- Pushes mutex to lower level