Operating Systems

Thread Synchronization Primitives

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2018

Agenda

- Week 42/43: Synchronization primitives
- Week 44: **Vacation**
- Week 45: Synchronization implementation
- Week 46: **Second Midterm Exam** + Advanced Synchronization Techniques
- Week 47: CPU Scheduling + I/O and Disks
- Week 48: RAID + File Systems

References

The content of these lectures is inspired by:

- The lecture notes of Prof. André Schiper.
- The lecture notes of Prof. David Mazières.
- The lectures notes of Arnaud Legrand.
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau

Other references:

- *Modern Operating Systems* by A. Tanenbaum
- *Operating System Concepts* by A. Silberschatz et al.

Agenda

Seen previously

Threads

- Schedulable execution context
- Multi-threaded program = multiple threads in the same process address space
- Allow a process to use several CPUs
- Allow a program to overlap I/O and computation

Implementation

- Kernel-level threads
- User-level threads
- Preemptive vs non-preemptive

Seen previously

POSIX threads API (`pthreads`) – pseudo API:

- `tid thread_create(void (*fn)(void *), void *arg);`
- `void thread_exit();`
- `void thread_join(tid thread);`

Data sharing

- Threads share the data of the enclosing process

Agenda

Motivation

Observations

- Multi-thread programming is used in many contexts.
  - It is also called concurrent programming.
- Shared memory is the inter-thread communication medium.

Is it easy to use shared memory to cooperate?

Motivation

Observations

- Multi-thread programming is used in many contexts.
  - It is also called concurrent programming.
- Shared memory is the inter-thread communication medium.

Is it easy to use shared memory to cooperate?
NO

The problem:

A set of threads executing on a shared-memory (multi-)processor is an asynchronous system.

- A thread can be preempted at any time.
- Reading/writing a data in memory incurs unpredictable delays (data in L1 cache vs page fault).

High-level goals of the lecture

- Start thinking like a concurrent programmer
- Learn to identify concurrency problems
- Learn to cooperate through shared memory
    - Synchronization
    - Communication
- Think about the correctness of an algorithm

Content of this lecture

Classical concurrent programming problems

- Mutual exclusion
- Producer-consumer

Concepts related to concurrent programming

- Critical section
- Busy waiting
- Deadlock

Synchronization primitives

- Locks
- Semaphores
- Condition variables

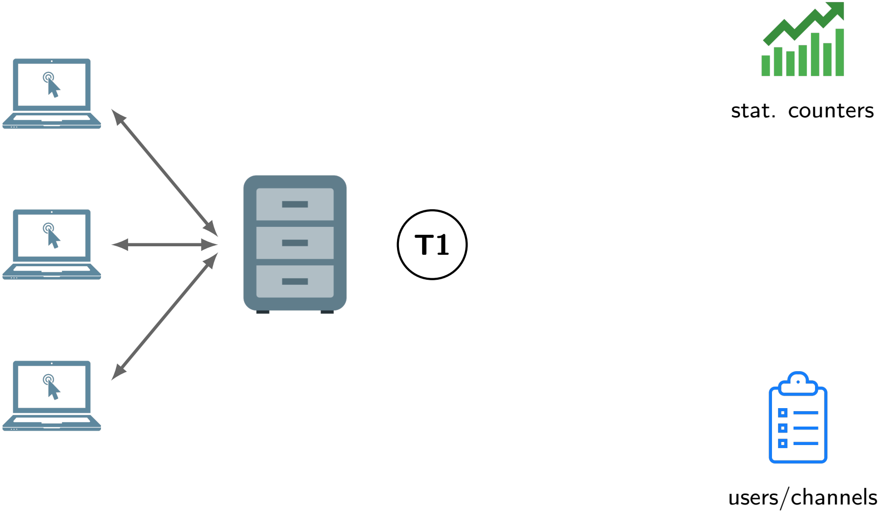
Agenda

Example: A chat server

Single-threaded version

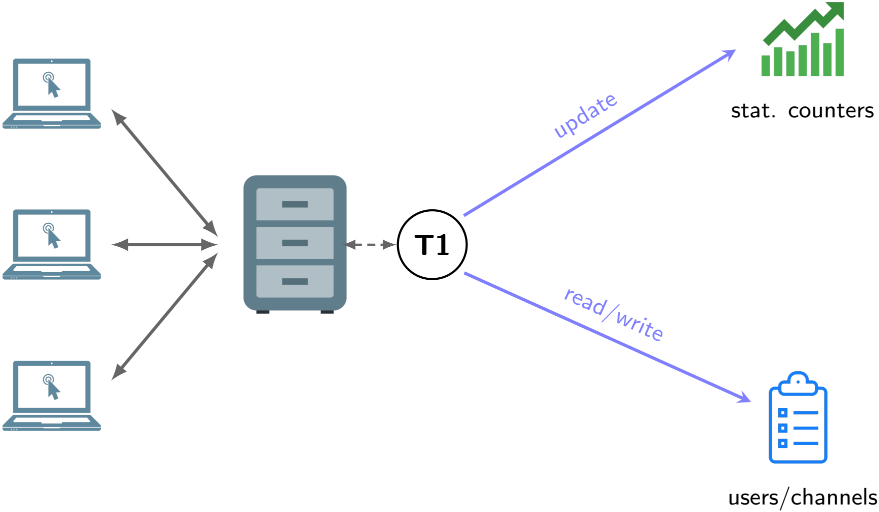
Example: A chat server

Single-threaded version



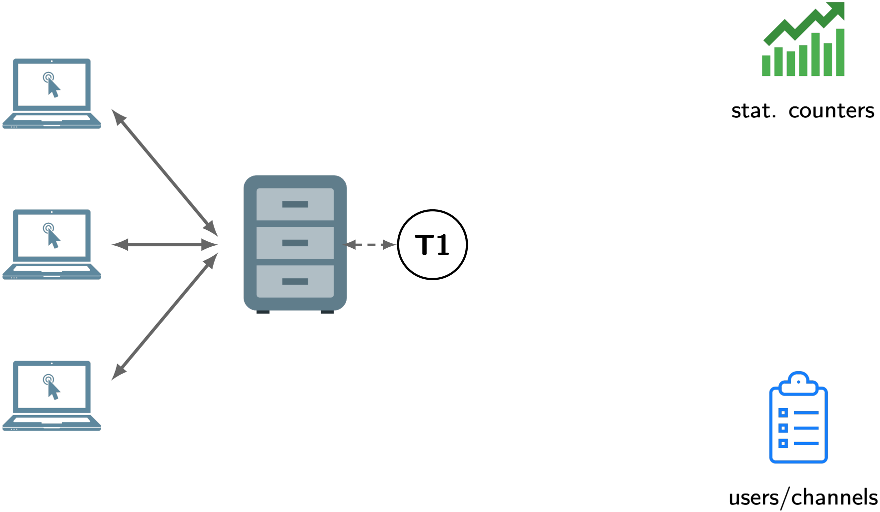
Example: A chat server

Single-threaded version



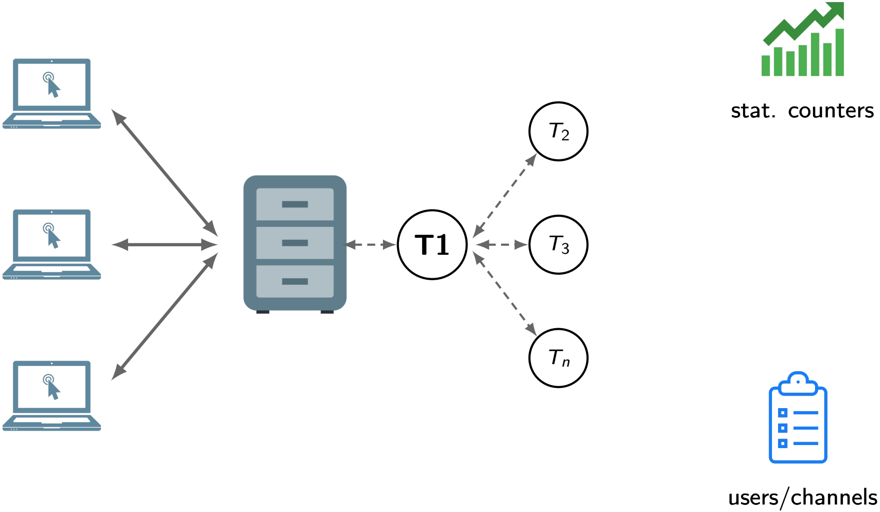
Example: A chat server

First multi-threaded version



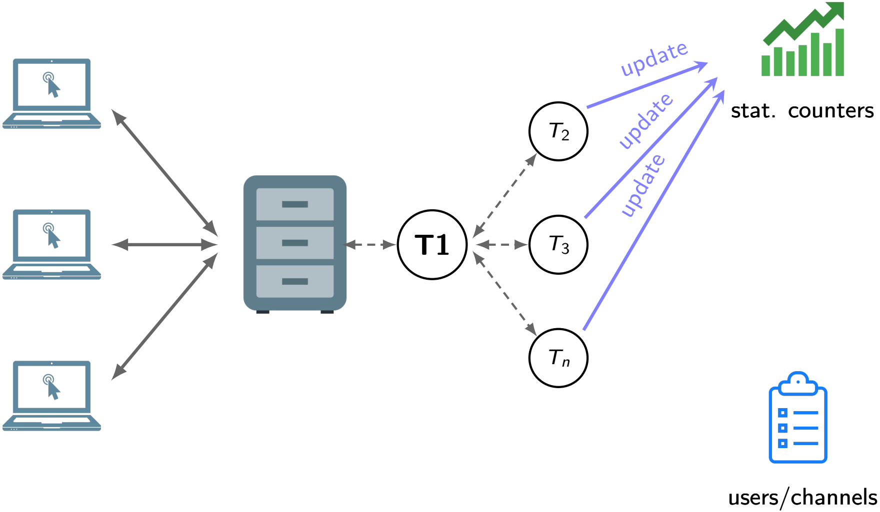
Example: A chat server

First multi-threaded version



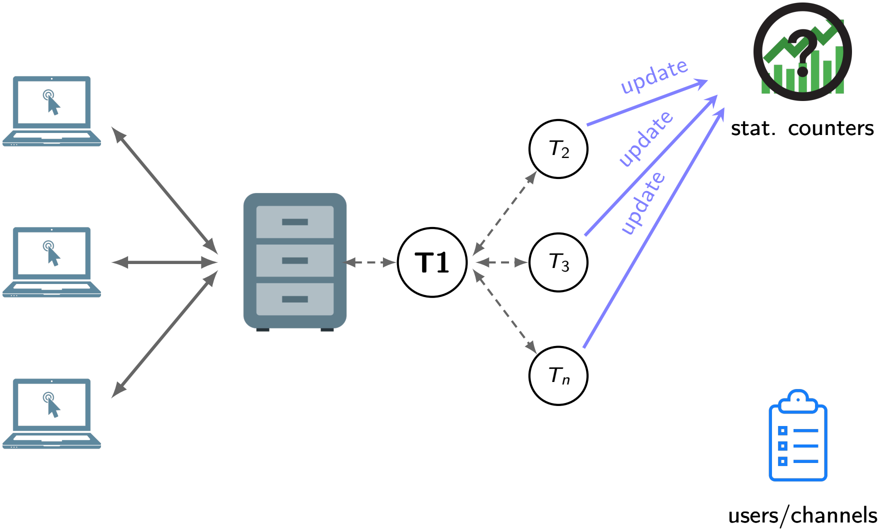
Example: A chat server

First multi-threaded version



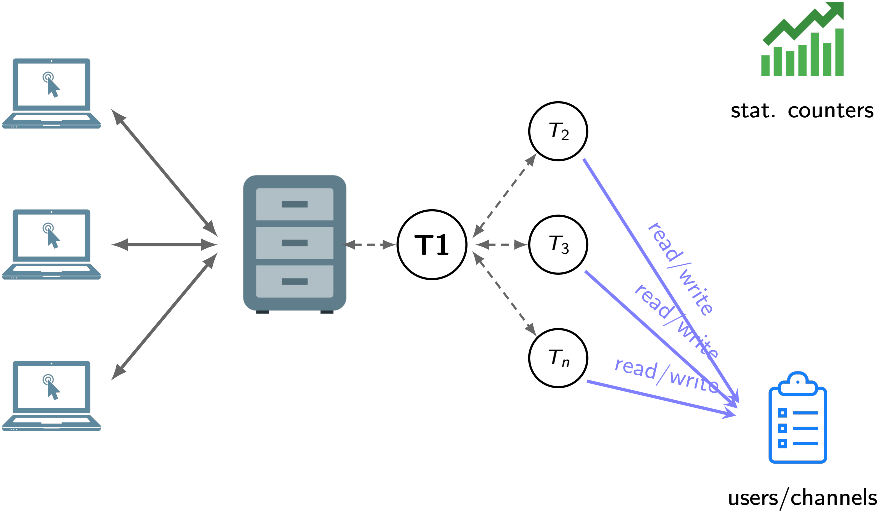
Example: A chat server

First multi-threaded version



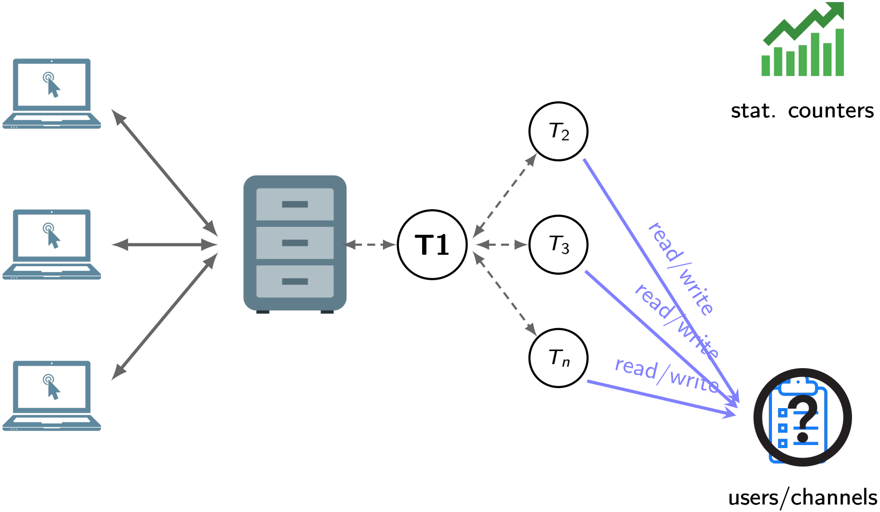
Example: A chat server

First multi-threaded version



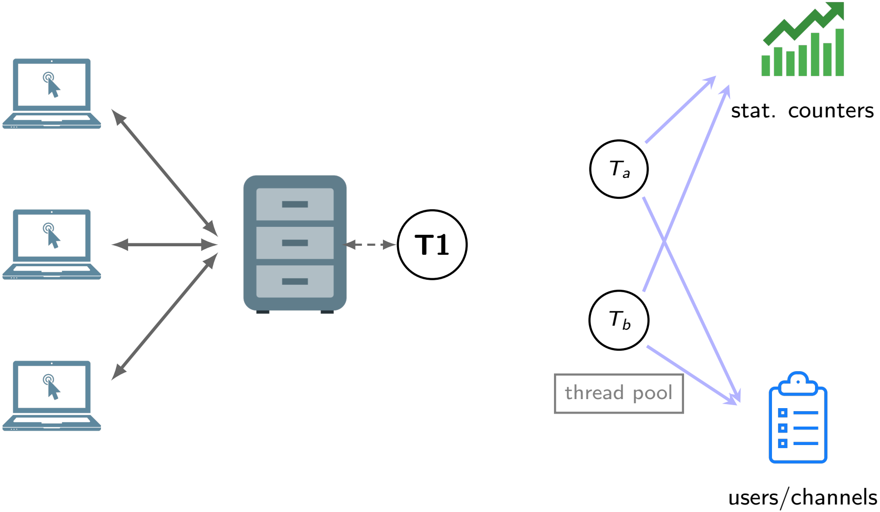
Example: A chat server

First multi-threaded version



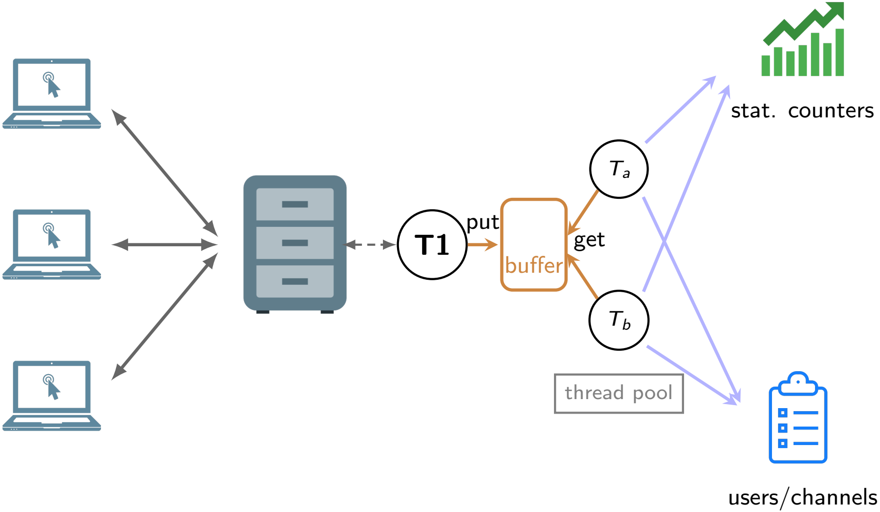
Example: A chat server

Second multi-threaded version



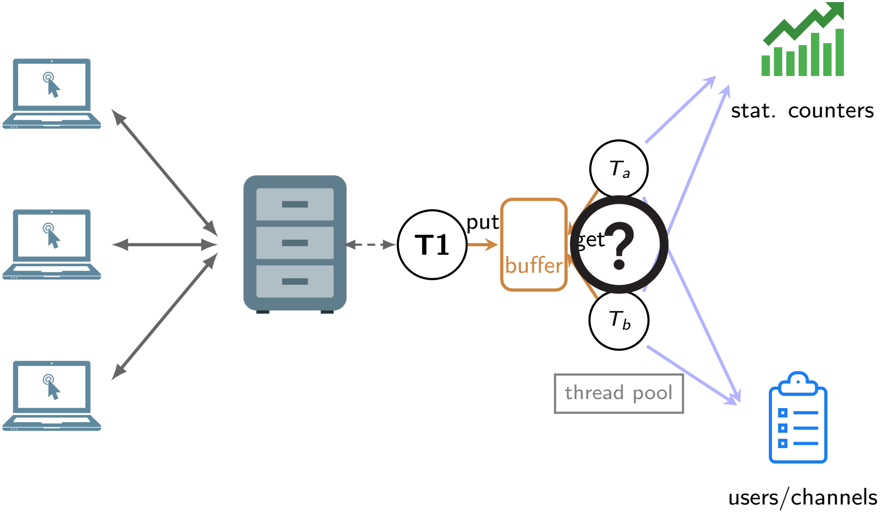
Example: A chat server

Second multi-threaded version



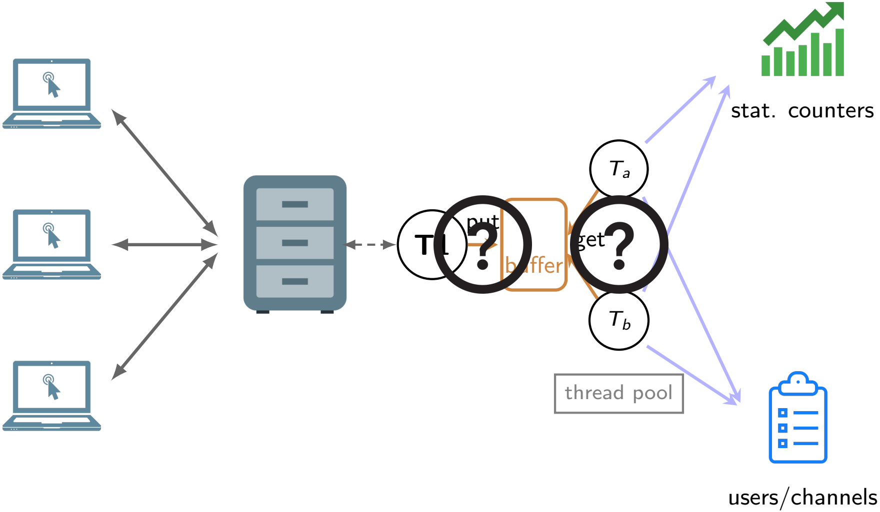
Example: A chat server

Second multi-threaded version



Example: A chat server

Second multi-threaded version



Classical problems

Synchronization

Mutual exclusion

- Avoid that multiple threads execute operations on the same data concurrently (*critical sections*)
- Example: Update data used for statistics

Classical problems

Synchronization

Mutual exclusion

- Avoid that multiple threads execute operations on the same data concurrently (*critical sections*)
- Example: Update data used for statistics

Reader-Writer

- Allow multiple readers or a single writer to access a data
- Example: Access to list of users and channels

Classical problems

Cooperation

Producer-Consumer

- Some threads *produce* some data that are *consumed* by other threads
- Example: A buffer of requests to be processed

Agenda

A shared counter

```
int count = 0;
```

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

What is the final value of `count`?

A shared counter

```
int count = 0;
```

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

What is the final value of `count`?

- A value between 2 and 20

A shared counter: Explanation

Let's have a look at the (pseudo) assembly code for `count++`:

```
mov   count, register
add   $0x1, register
mov   register, count
```

A shared counter: Explanation

Let's have a look at the (pseudo) assembly code for `count++`:

```
mov   count, register
add   $0x1, register
mov   register, count
```

A possible interleave (for one iteration on each thread)

Thread 1	Thread 2
mov count, register	
add $0x1, register	
	mov count, register
	add $0x1, register
mov register, count	
	mov register, count

A shared counter: Explanation

Let's have a look at the (pseudo) assembly code for `count++`:

```
mov   count, register
add   $0x1, register
mov   register, count
```

A possible interleave (for one iteration on each thread)

mov count, register	
add $0x1, register	
	mov count, register
	add $0x1, register
mov register, count	
	mov register, count

At the end, `count=1` :-(

A shared counter

This may happen:

- When threads execute on different processor cores
- When *preemptive* threads execute on the same core
  - A thread can be preempted at any time in this case

Critical section

Critical resource

A critical resource should not be accessed by multiple threads at the same time. It should be accessed in mutual exclusion.

Critical section

Critical resource

A critical resource should not be accessed by multiple threads at the same time. It should be accessed in mutual exclusion.

Critical section (CS)

A critical section is a part of a program code that accesses a critical resource.

Critical section: Definition of the problem

Safety

- Mutual exclusion: At most one thread can be in CS at a time

Liveness

- Progress: If no thread is currently in CS and threads are trying to access, one should eventually be able to enter the CS.
- Bounded waiting: Once a thread T starts trying to enter the CS, there is a bound on the number of times other threads get in.

Critical section: About liveness requirements

Liveness requirements are mandatory for a solution to be useful

Critical section: About liveness requirements

Liveness requirements are mandatory for a solution to be useful

Progress vs. Bounded waiting

- Progress: If no thread can enter CS, we don't have progress.
- Bounded waiting: If thread A is waiting to enter CS while B repeatedly leaves and re-enters C.S. ad infinitum, we don't have bounded waiting

Shared counter: New version

Thread 1:

```
Enter CS;
count++;
Leave CS;
```

Thread 2:

```
Enter CS;
count++;
Leave CS;
```

Shared counter: New version

Thread 1:

```
Enter CS;
count++;
Leave CS;
```

Thread 2:

```
Enter CS;
count++;
Leave CS;
```

How to implement Enter CS and Leave CS?

Implementation: First try using busy waiting

Shared variables:

```
int count=0;
int busy=0;
```

Thread 1:

```
while(busy){;}
busy=1;
count++;
busy=0;
```

Thread 2:

```
while(busy){;}
busy=1;
count++;
busy=0;
```

Exercise

Show through an example that the solution violates safety.

Exercise

Show through an example that the solution violates safety.

Thread 1	Thread 2
while(busy){;}	
	while(busy){;}
	busy = 1
busy = 1	
count++	
	count++

- The 2 threads access `count` **at the same time**.

Exercise

Show through an example that the solution violates liveness.

Exercise

Show through an example that the solution violates liveness.

Thread 1	Thread 2
while(busy){;}	
busy = 1	
count++	
	while(busy){;}
busy = 0	
while(busy){;}	
busy = 1	
	while(busy){;}
count++	
...	

- With a bad interleaving of threads, Thread 2 never gets access to count.

Synchronization primitives

To implement mutual exclusion, we need help from the hardware (and the operating system).

- Implementing mutual exclusion is the topic of next course.

Threading libraries provide synchronization primitives:

- A set of functions that allow synchronizing threads
  - Locks
  - Semaphores
  - Condition variables

Agenda

Locks

A lock provides a means to achieve mutual exclusion.

Specification

A *lock* is defined by a lock variable and two methods: `lock()` and `unlock()`.

Locks

A lock provides a means to achieve mutual exclusion.

Specification

A *lock* is defined by a lock variable and two methods: `lock()` and `unlock()`.

- A lock can be free or held

Locks

A lock provides a means to achieve mutual exclusion.

Specification

A *lock* is defined by a lock variable and two methods: `lock()` and `unlock()`.

- A lock can be free or held
- lock(): If the lock is free, the calling thread acquires the lock and enters the CS. Otherwise the thread is blocked.

Locks

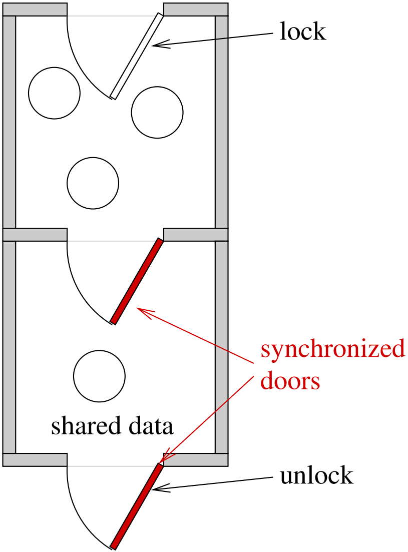
A lock provides a means to achieve mutual exclusion.

Specification

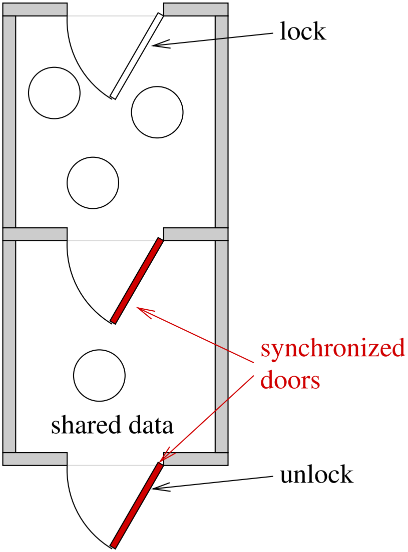
A *lock* is defined by a lock variable and two methods: `lock()` and `unlock()`.

- A lock can be free or held
- lock(): If the lock is free, the calling thread acquires the lock and enters the CS. Otherwise the thread is blocked.
- unlock(): Releases the lock. It has to be called by the thread currently holding the lock.

Locks: Analogy

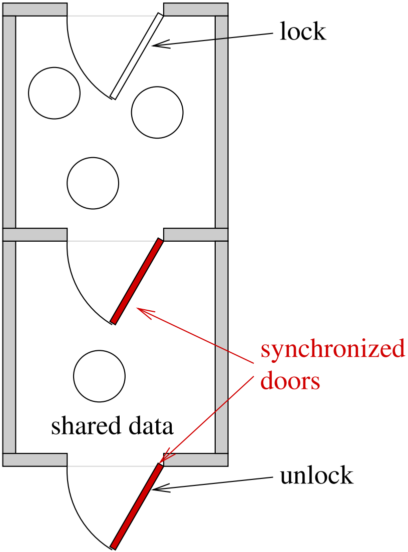


Locks: Analogy



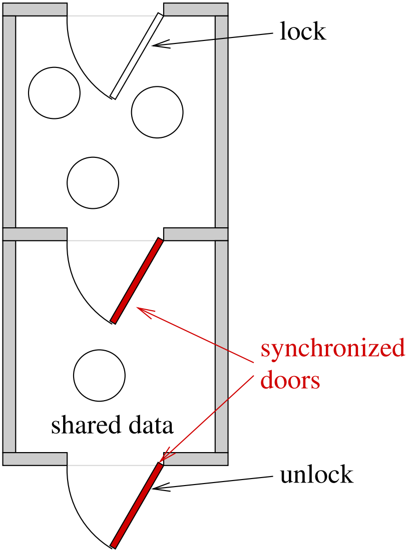
- Calling lock, a thread enters a waiting room

Locks: Analogy



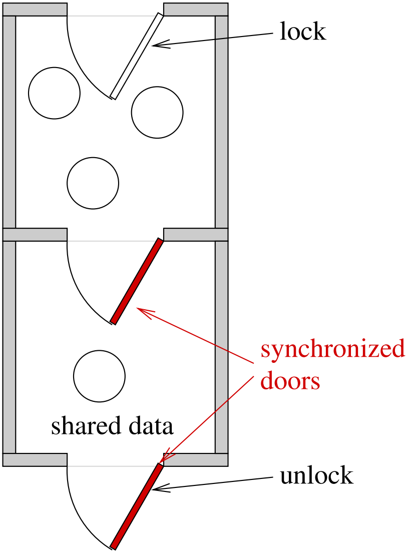
- Calling lock, a thread enters a waiting room
- A single thread can be in the *CS* room (hosting the shared data)

Locks: Analogy



- Calling lock, a thread enters a waiting room
- A single thread can be in the *CS* room (hosting the shared data)
- When the thread in the CS room calls unlock, it leaves the CS room, and lets one thread from the waiting room enter (opens the doors of the CS room)

Locks: Analogy



- Calling lock, a thread enters a waiting room
- A single thread can be in the *CS* room (hosting the shared data)
- When the thread in the CS room calls unlock, it leaves the CS room, and lets one thread from the waiting room enter (opens the doors of the CS room)
  - The doors of the CS room are initially opened.

Programming with locks

All critical data should be protected by a lock!

- Critical = accessed by more than one thread, at least one write
- Exception is initialization, before data is exposed to other threads
- This is the responsibility of the application writer

Pthread locks: Mutexes

- `mutex`: variable of type `pthread_mutex_t`
- `pthread_mutex_init(&mutex, ...)`: initialize the mutex
  - The macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize a mutex allocated statically with the default options
- `pthread_mutex_destroy(&mutex)`: destroy the mutex
- `pthread_mutex_lock(&mutex)`
- `pthread_mutex_unlock(&mutex)`
- `pthread_mutex_trylock(&mutex)`: is equivalent to lock(), except that if the mutex is held, it returns immediately with an error code

Pthread locks: Example

```
#include <pthread.h>

int count=0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;


void* thread_routine(void *arg){

      /* ... */
      pthread_mutex_lock(&count_mutex);
      count++;
      pthread_mutex_unlock(&count_mutex);
      /* ... */

}
```

Pthread locks attributes

`man pthread_mutex_lock`

Several attributes of a lock can be configured at initialization among which:

- `type`
  - `NORMAL`: Deadlock on *relock*[1]
  - `RECURSIVE`: Allows relocking. A *lock count* is implemented (as many lock() as unlock() calls required).
  - `ERRORCHECK`: Error returned on relock.
  - `DEFAULT`: Usually maps to `NORMAL`.

- `robust`: Defines what happens if a thread terminates without releasing a lock, and if a non-owner thread calls unlock().

- Other attributes are related to priority management and visibility of the lock.

[1] A thread calls lock() on a lock it already locked.

Agenda

Semaphores

- Locks ensure mutual exclusion.
- A semaphore is another mechanism that allows controlling access to shared variables but is more powerful than a lock.
- Semaphores were proposed by Dijkstra in 1968

Semaphores

A semaphore is initialized with an integer value N and can be manipulated with two operations P and V.

About the interface

- P stands for Proberen (Dutch) – try
- V stands for Verhogen (Dutch) – increment

POSIX interface

- P $\rightarrow$ `int sem_wait(sem_t *s)`
- V $\rightarrow$ `int sem_post(sem_t *s)`
  - Other interfaces call it `sem_signal()`

Semaphores

When a tread calls sem_wait():

```
N = N - 1;
if( N < 0 )
   Calling thread is blocked
```

When a tread calls sem_post():

```
N = N +1;
if( N <= 0 )
   One blocked thread is unblocked
```

About the value of N:

Semaphores

When a tread calls sem_wait():

```
N = N - 1;
if( N < 0 )
   Calling thread is blocked
```

When a tread calls sem_post():

```
N = N +1;
if( N <= 0 )
   One blocked thread is unblocked
```

About the value of N:

- If $N > 0$, N is the *capacity* of the semaphore
- if $N < 0$, N is the number of blocked threads

Mutual exclusion with semaphores

Mutual exclusion with semaphores

- Initializing a semaphore with value N can be seen as providing it with N *tokens*
- To implement critical sections, a semaphore should be initialized with $N = 1$
  - Warning: A semaphore with $N = 1$ and a lock are not equivalent

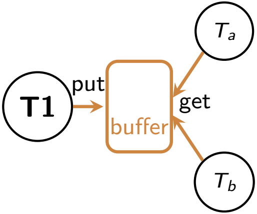
Example

```
#include <semaphore.h>

int count=0;
sem_t count_mutex;

sem_init(&count_mutex, 0, 1);
/* ... */
sem_wait(&count_mutex);
count++;
sem_post(&count_mutex);
```

Agenda

Specification of the problem

Recall



Specification

- A buffer of fixed size
- Producer threads put items into the buffer. The *put* operation blocks if the buffer is full
- Consumer threads get items from the buffer. The *get* operation blocks if the buffer is empty

Producer-Consumer

```
void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */


    while (count == BUFFER_SIZE) {

      /* Do nothing */

    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;

  }
}
```

```
void consumer (void *ignored) {
  for (;;) {

    while (count == 0) {

      /* Do nothing */

    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;


    /* consume the item in
       nextConsumed */
  }
}
```

Producer-Consumer

```
void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */


    while (count == BUFFER_SIZE) {

      /* Do nothing */

    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;

  }
}
```

```
void consumer (void *ignored) {
  for (;;) {

    while (count == 0) {

      /* Do nothing */

    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;


    /* consume the item in
       nextConsumed */
  }
}
```

Not correct: shared data are not protected

- `count` can be accessed by the prod. and the cons.
- With multiple prod./cons., concurrent accesses to `in`, `out`, `buffer`

Producer-Consumer with Locks

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE) {

      thread_yield ();

    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0) {

      thread_yield ();

    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    mutex_unlock (&mutex);

    /* consume the item in
       nextConsumed */
  }
}
```

Producer-Consumer with Locks

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE) {

      thread_yield ();

    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0) {

      thread_yield ();

    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    mutex_unlock (&mutex);

    /* consume the item in
       nextConsumed */
  }
}
```

Not correct: If a thread enters a `while` loop, all threads are blocked forever (deadlock)

- yield() does not release the lock

Producer-Consumer with Locks

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    mutex_unlock (&mutex);

    /* consume the item in
       nextConsumed */
  }
}
```

Producer-Consumer with Locks

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    mutex_unlock (&mutex);

    /* consume the item in
       nextConsumed */
  }
}
```

Correct ... but busy waiting

- **We don't want busy waiting**

About Busy Waiting

Busy waiting

Waiting for some condition to become true by repeatedly checking (spinning) the value of some variable.

Why is it bad?

- Waste of CPU cycles
  - Use CPU cycles to check the value of a variable while there is no evidence that this value has changed.
  - Follows from previous comment: Using `sleep` is still busy waiting.
- On a single processor: Wasted cycles could have been used by other threads.
- On a multi-processor: Repeatedly reading a variable that is used by other threads can slow down these threads.
  - In specific cases, with a careful design, busy waiting can be efficient.

Cooperation

Cooperation = Synchronization + Communication

- Synchronization: Imposing an order on the execution of instructions
- Communication: Exchanging information between threads

Semaphores allow cooperation between threads

Producer-Consumer with semaphores

- Initialize `fullCount` to 0 (block consumer on empty buffer)
- Initialize `emptyCount` to N (block producer when buffer full)

```
void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    sem_wait(&emptyCount);

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    /*count++;*/

    sem_post(&fullCount)
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    sem_wait(&fullCount);

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /*count--;*/

    sem_post(&emptyCount);

    /* consume the item in
       nextConsumed */
  }
}
```

Producer-Consumer with semaphores

- Initialize `fullCount` to 0 (block consumer on empty buffer)
- Initialize `emptyCount` to N (block producer when buffer full)
- An additional semaphore (initialized to 1) should be used for mutual exclusion (a lock could be used instead)

```
void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    sem_wait(&emptyCount);
    sem_wait(&mutex)
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    /*count++;*/
    sem_post(&mutex)
    sem_post(&fullCount)
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    sem_wait(&fullCount);
    sem_wait(&mutex)
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /*count--;*/
    sem_post(&mutex)
    sem_post(&emptyCount);

    /* consume the item in
       nextConsumed */
  }
}
```

Comments on semaphores

- Semaphores allow elegant solutions to some problems (producer-consumer, reader-writer)
- However they are quite error prone:
  - If you call `wait` instead of `post`, you'll have a deadlock
  - If you forget to protect parts of your code, you might violate mutual exclusion
  - You have "tokens" of different types, which may be hard to reason about

This is why other constructs have been proposed

Agenda

- Goals of the lecture
- A Multi-Threaded Application
- Mutual Exclusion
- Locks
- Semaphores
- The Producer-Consumer Problem
- Condition Variables
- Monitors
- Other synchronization problems

Condition variables (pthreads)

A condition variable is a *special shared variable*.

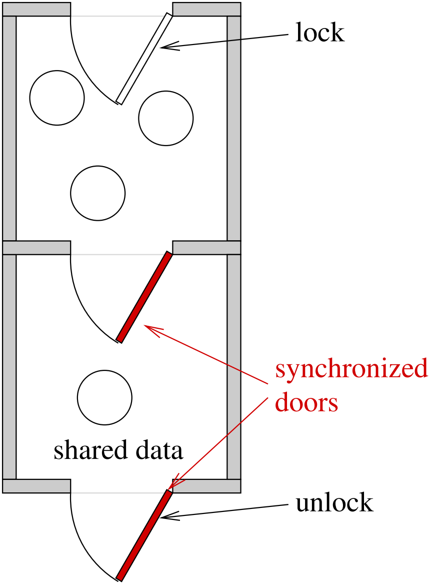
- It allows a thread to explicitly put itself to *wait*.
  - The condition variable can be seen as a container of waiting threads.
  - As such, this variable does not have a value.
- It is used together with a mutex:
  - When a thread puts itself to wait, the corresponding mutex is released.
- It is often associated to a *logical condition* (reason for this name)

Condition variables (pthreads)

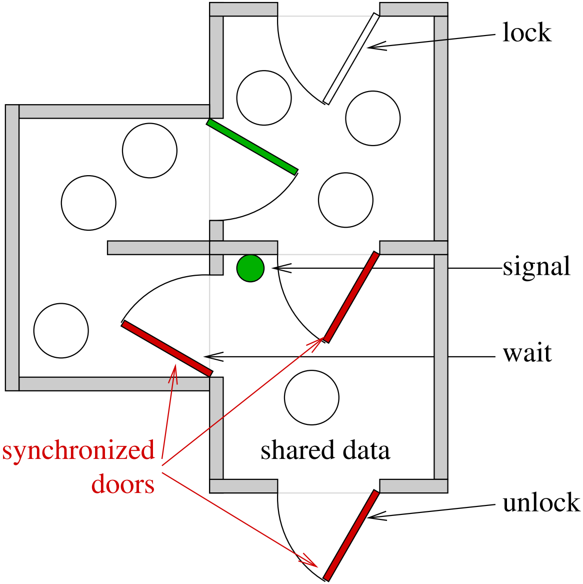
Interface

- `cond`: variable of type `pthread_cond_t`
- `pthread_cond_init(&cond, ...)`: initialize the condition
  - The macro `PTHREAD_COND_INITIALIZER` can be used to initialize a condition variable allocated statically with the default options
- `void pthread_cond_wait(&cond, &mutex)`: atomically unlock `mutex` and put the thread to wait on `cond`.
- `void pthread_cond_signal(&cond)` and `pthread_cond_broadcast(&cond)`: Wake one/all the threads waiting on `cond`.

Condition variable: Analogy



Condition variable: Analogy



On the semantic of the operations

- Calling `wait()` releases the lock similarly to `unlock()`.
- When a thread is woken up by a call to `signal()` (or `broadcast()`), it is guaranteed that at the time it returns from `wait()`, it owns the corresponding lock again.
  - However, it has to compete with other threads to acquire that lock before returning from `wait()`.
- On a call to `signal()`, any of the waiting threads might be the one that is woken up.
- Calling functions `signal()` and `broadcast()` does not require owning the lock.
  - However in most cases the lock should be held for the application logic to be correct.

Producer-Consumer with condition variables

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and
       put in nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE)
      cond_wait (&nonfull, &mutex);

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    cond_signal (&nonempty);
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0)
      cond_wait (&nonempty, &mutex);

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    cond_signal (&nonfull);
    mutex_unlock (&mutex);

    /* consume the item
       in nextConsumed */
  }
}
```

Beware: this solution does not warrant First Come First Served!

More on condition variables

Why must `cond_wait` both release mutex and sleep? Why not separate mutexes and condition variables?

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait(&nonfull);
  mutex_lock (&mutex);
}
```

More on condition variables

Why must `cond_wait` both release mutex and sleep? Why not separate mutexes and condition variables?

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait(&nonfull);
  mutex_lock (&mutex);
}
```

A thread could end up stuck waiting because of a bad interleaving

- A condition variable has no associated state

PRODUCER	CONSUMER
`while (count == BUFFER_SIZE){`	
`  mutex_unlock (&mutex);`	
	`mutex_lock (&mutex);`
	`...`
	`count--;`
	`cond_signal (&nonfull);`
`  cond_wait (&nonfull);`	
`}`	

Agenda

Goals of the lecture

A Multi-Threaded Application

Mutual Exclusion

Locks

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

Other synchronization problems

Monitors

- A monitor is a synchronization construct
- It provides synchronization mechanisms similar to mutex + condition variables. (Some people call both "monitors")

Definition

- A monitor is an object/module with a set of methods.
- Each method is executed in mutual exclusion
- Condition variables (or simply "*conditions*") are defined with the same semantic as defined previously

Comments on monitors

- Proposed by Brinch Hansen (1973) and Hoare (1974)
- Possibly less error prone than raw mutexes
- Basic synchronization mechanism in Java
- Different *flavors* depending on the semantic of signal:
  - Hoare-style: The signaled thread get immediately access to the monitor. The signaling thread waits until the signaled threads leaves the monitor.
  - Mesa-style (java): The signaling thread stays in the monitor.
- Semaphores can be implemented using monitors and monitors can be implemented using semaphores

Agenda

The Reader-Writer problem

Problem statement

- Several threads try to access the same shared data, some reading, other writing.
- Either a single writer or multiple readers can access the shared data at any time

Different flavors

- Priority to readers
- Priority to writers

The Dining Philosophers problem

Proposed by Dijkstra

Problem statement

5 philosophers spend their live alternatively thinking and eating. They sit around a circular table. The table has a big plate of rice but only 5 chopsticks, placed between each pair of philosophers. When a philosopher wants to eat, he has to peak the chopsticks on his left and on his right. Two philosophers can't use a chopstick at the same time. How to ensure that no philosopher will starve?

Goals

- Avoid deadlocks: Each philosopher holds one chopstick
- Avoid starvation: Some philosophers never eat