Operating Systems

Thread Synchronization

Thomas Ropars

thomas.ropars@imag.fr

Équipe ERODS – LIG/IM2AG/UJF

2015

Agenda

- Week 45: Synchronization: Basic Mechanisms
- Week 46: Synchronization: Implementation
- Week 47: Advanced Synchronization Techniques + IOs
- Week 48: **Second Midterm Exam** + Scheduling
- Week 49: Filesystems

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

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

Agenda

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

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
  - Hybrid threading (n:m threading)
- Preemptive vs non-preemptive

Seen previously

POSIX threads API (pthreads)

- `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

Concurrency issues

Threads are very useful in many contexts but using threads raises several challenges:

Concurrency issues

Threads are very useful in many contexts but using threads raises several challenges:

- How to synchronize (efficiently)?

Concurrency issues

Threads are very useful in many contexts but using threads raises several challenges:

- How to synchronize (efficiently)?
- How to cooperate (efficiently)?

Concurrency issues

Threads are very useful in many contexts but using threads raises several challenges:

- How to synchronize (efficiently)?
- How to cooperate (efficiently)?

Threads interact through shared memory:

- Memory in modern processors is very complex

Agenda

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

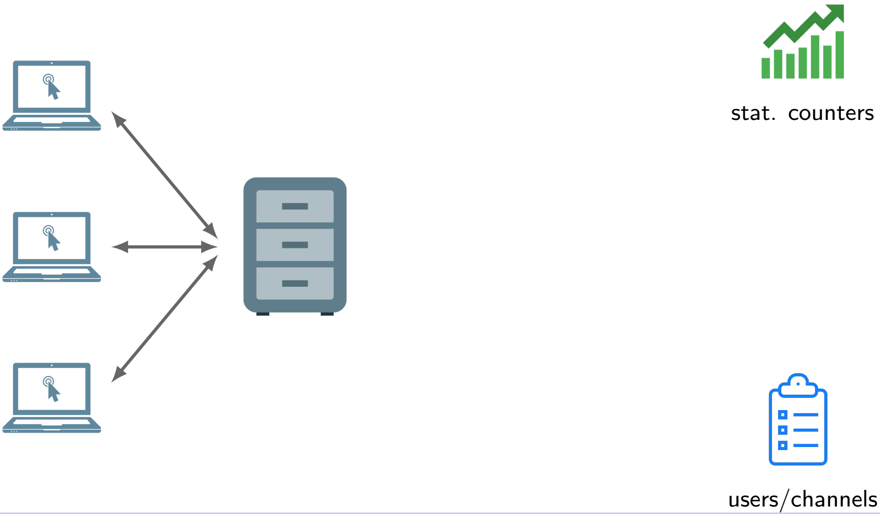
Condition Variables

Monitors

More on synchronization

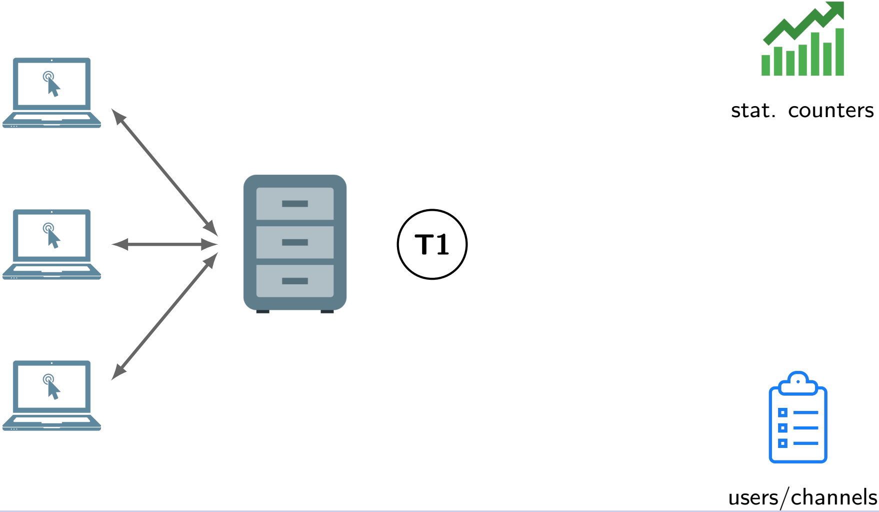
Example: A chat server

Single-threaded version



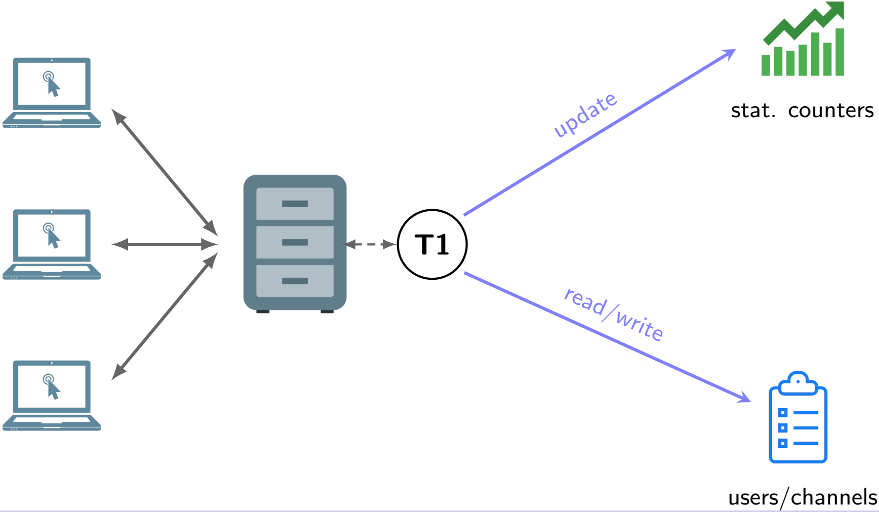
Example: A chat server

Single-threaded version



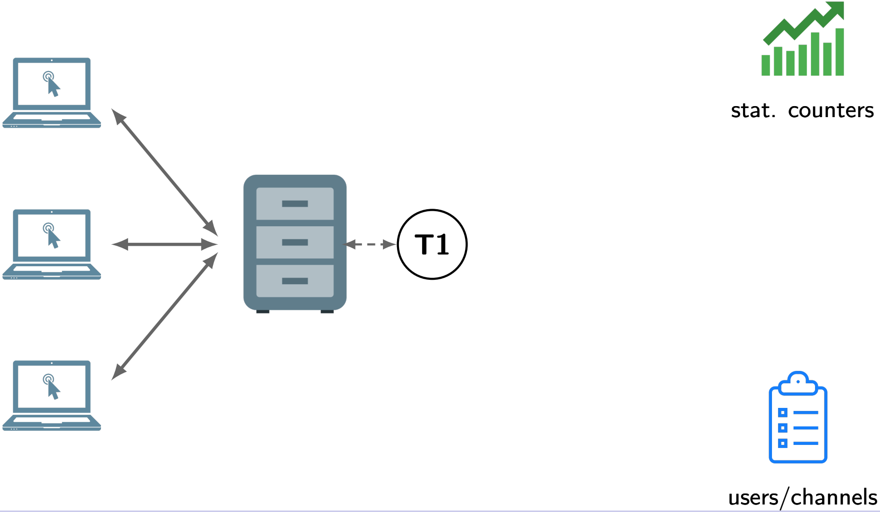
Example: A chat server

Single-threaded version



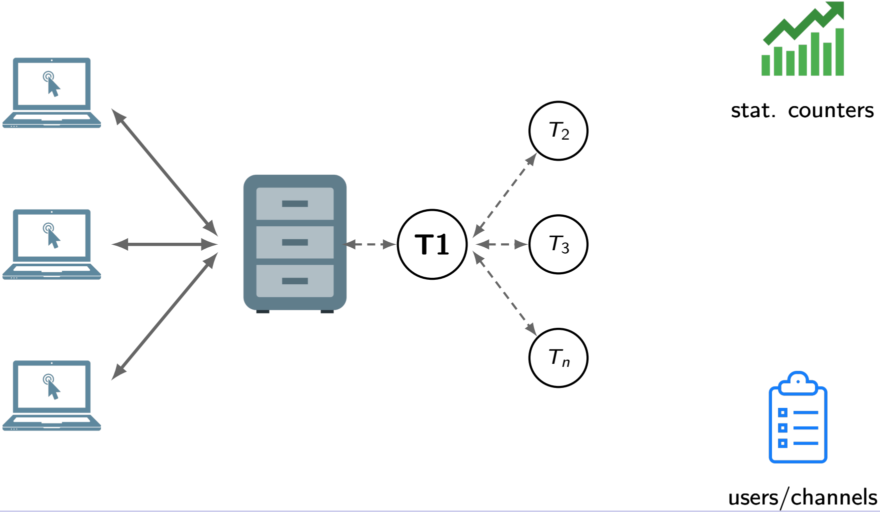
Example: A chat server

First multi-threaded version



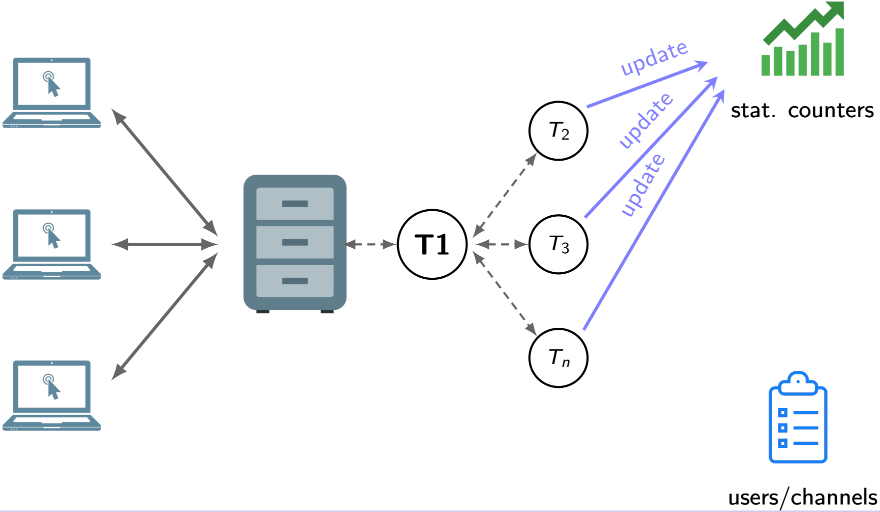
Example: A chat server

First multi-threaded version



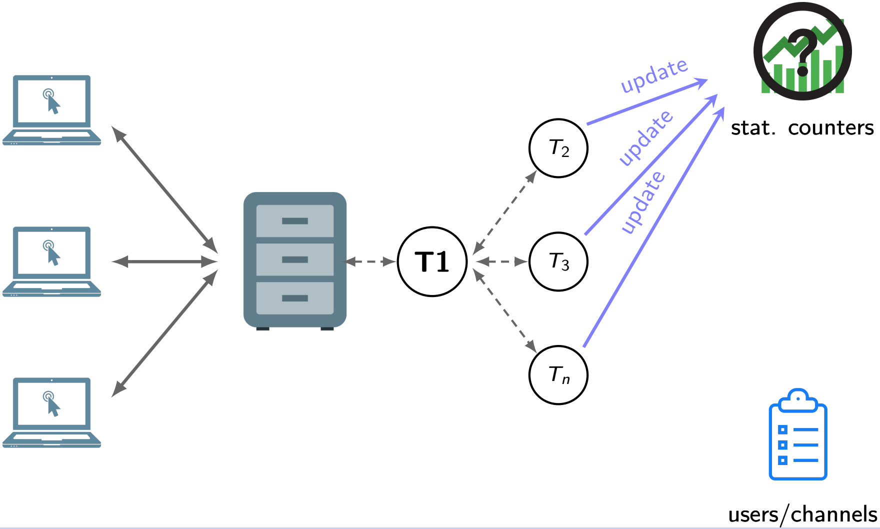
Example: A chat server

First multi-threaded version



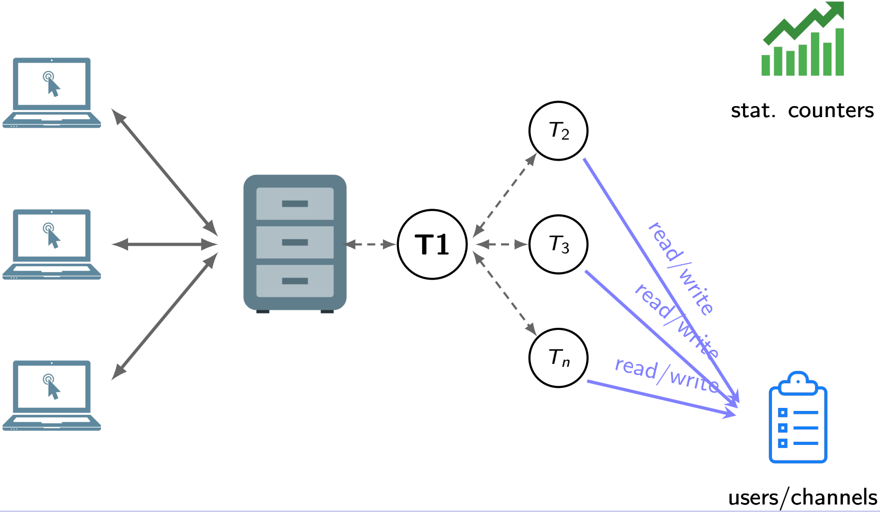
Example: A chat server

First multi-threaded version



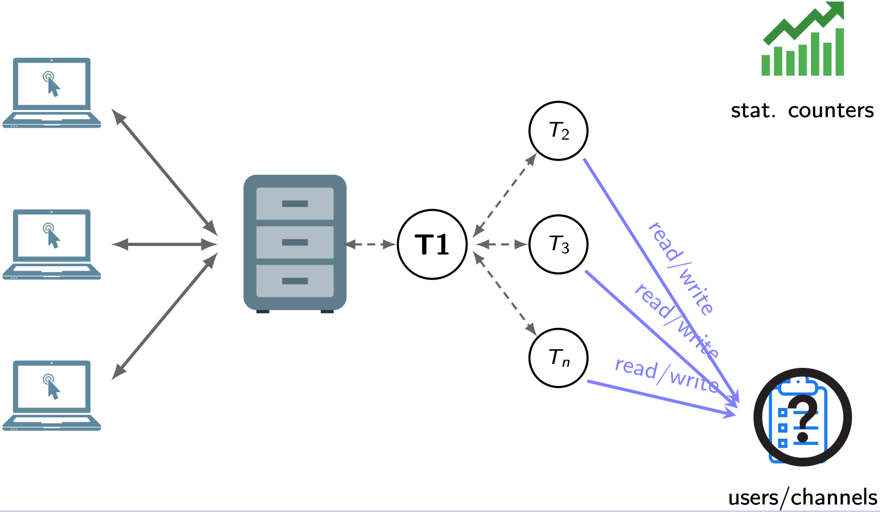
Example: A chat server

First multi-threaded version



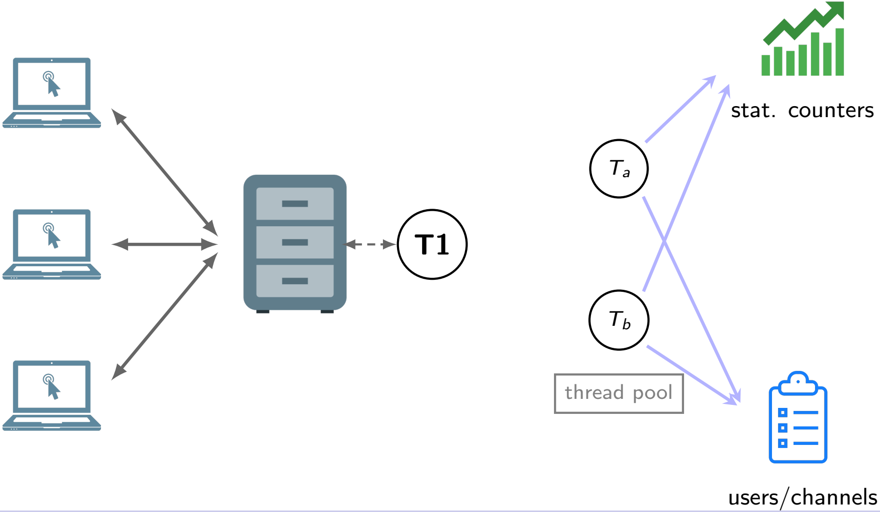
Example: A chat server

First multi-threaded version



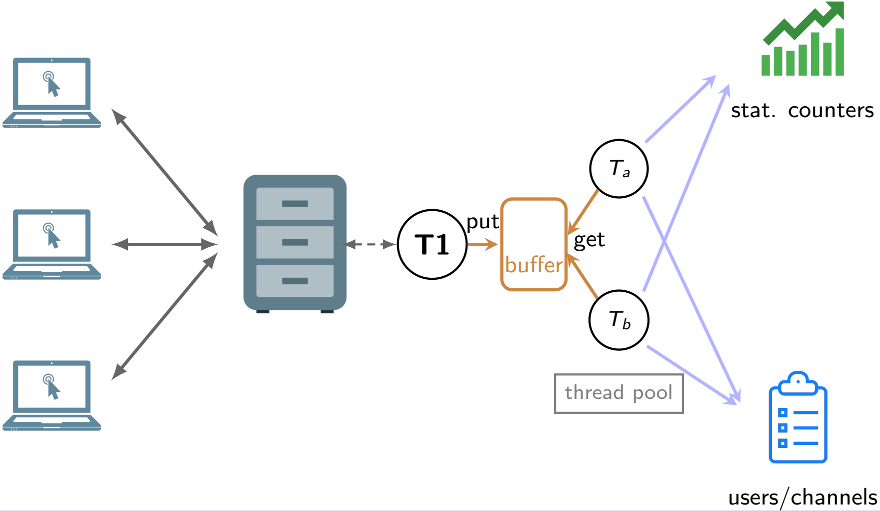
Example: A chat server

Second multi-threaded version



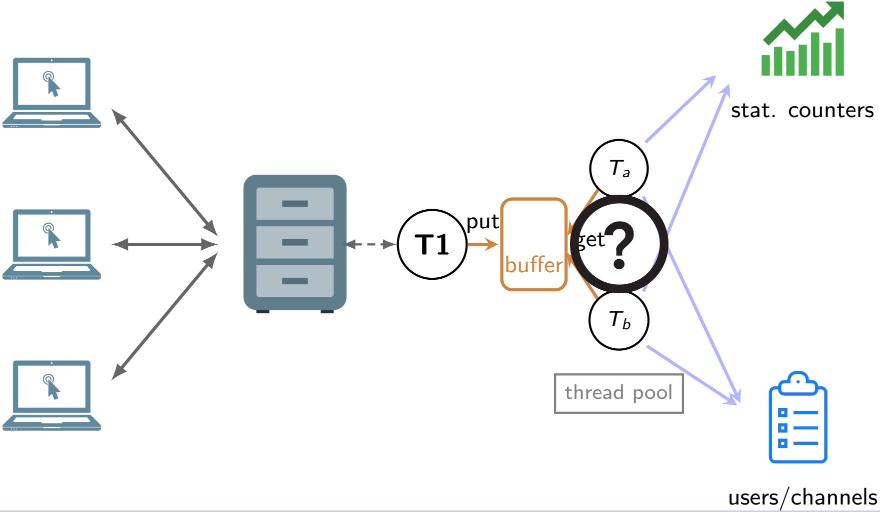
Example: A chat server

Second multi-threaded version



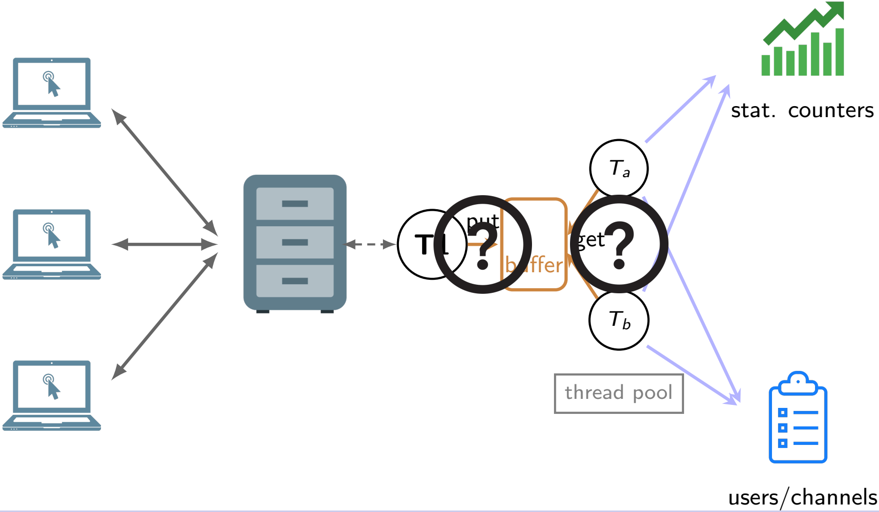
Example: A chat server

Second multi-threaded version



Example: A chat server

Second multi-threaded version



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

Producer-Consumer

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

Agenda

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

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

Final value of `count`:

- 20
- 10
- 15
- I don't know

What do you think?

A shared counter

Final value of `count`:

- 20
- 10
- 15
- I don't know

A shared counter

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

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

A shared counter

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

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

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

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 access a critical resource.

Critical section: Definition of the problem

Safety

- Mutual exclusion: Only 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;
```

Implementation: First try using busy waiting

Is the solution correct?

- The solution is correct
- The solution violates liveness
- The solution violates safety

Implementation: First try using busy waiting

Is the solution correct?

- The solution is correct
- The solution violates liveness
- The solution violates safety

CSes Implementations

Different solutions

- Peterson's algorithm
- Disable interrupts
- Locks
- Semaphores

Peterson's algorithm – Solution for 2 threads

- Assumes 2 threads T_o and T_1

Shared variables:

```
bool wants[2] = {false, false}; /*indicates which thread
                                  wants to enter the CS*/
int not_turn; /*not this thread turn*/
```

Code for thread T_i:

```
wants[i] = true;
not_turn = i;
while (wants[1-i] && not_turn == i)
  /* other thread wants in and not our turn, so loop */;
Critical_section ();
wants[i] = false;
Remainder_section ();
```

Peterson's algorithm – Correctness

Model checking

Proving the correctness of such an algorithm can be done using a model checker (*e.g.*, UPPAAL)

Discussion about correctness

Peterson's algorithm – Correctness

Model checking

Proving the correctness of such an algorithm can be done using a model checker (*e.g.*, UPPAAL)

Discussion about correctness

- Mutual exclusion: both threads in CS?
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from CS

Peterson's algorithm – Correctness

Model checking

Proving the correctness of such an algorithm can be done using a model checker (*e.g.*, UPPAAL)

Discussion about correctness

- Mutual exclusion: both threads in CS?
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from CS
- Progress
  - If T_{1-i} doesn't want CS, `wants[1-i] == false`, so T_i won't loop
  - If both threads try to enter, one thread is the `no_turn` thread

Peterson's algorithm – Correctness

Model checking

Proving the correctness of such an algorithm can be done using a model checker (*e.g.*, UPPAAL)

Discussion about correctness

- Mutual exclusion: both threads in CS?
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from CS
- Progress
  - If T_{1-i} doesn't want CS, `wants[1-i] == false`, so T_i won't loop
  - If both threads try to enter, one thread is the `no_turn` thread
- Bounded waiting
  - If T_i wants lock and T_{1-i} tries to re-enter, T_{1-i} will set `not_turn = 1 - i`, allowing T_i in

Peterson's algorithm – Limits

- Given solution works for 2 threads
- Can be generalized to *n* threads but *n* must be known in advance
- Note that the current version assumes that the memory is *sequentially consistent*. Most processors don't provide sequential consistency.
  - Discussed in another lecture.

Disabling interrupts

Basic idea

Prevent a thread from being interrupted while it is in CS

- On a single processor, if a thread is not interrupted, it will execute the CS atomically

Implementation

- Fully disabling interrupts is not a good solution (unsafe)
- Have a per-thread *Do-Not-Interrupt* (DNI) bit
  - Typical setup: periodic timer signal caught by thread scheduler
  - Scheduling decisions based on DNI bits

Limitations

- Cannot work on multi-processors
- Interrupts might get lost if disabled for a large amount of time

Locks

Specification

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

Locks

Specification

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

- A lock can be free or held

Locks

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

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

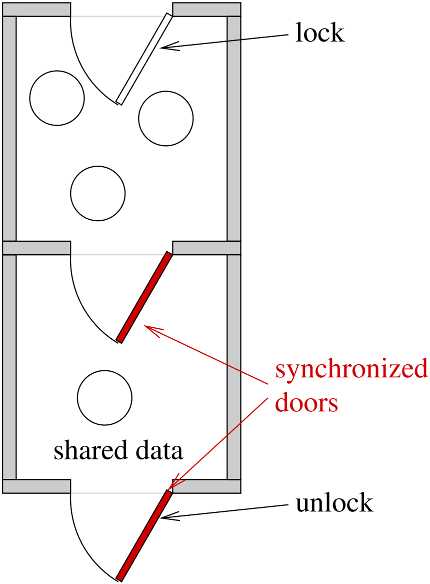
Specification

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 prevent programmers from having to write synchronization code explicitly.

Locks: Analogy



Programming with locks

All global data should be protected by a lock!

- Global = 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

Implementation of locks

User space

- Locks can be fully implemented in user space
- Use of atomic operations + busy waiting
- More details in next lecture

With the help of the kernel

- *Block* the thread: remove it from the ready list
  - In the case of a lock, put the thread into a list of threads waiting for the lock
- *Wake up* the thread: put it back into the ready list
- *Futex* system call in Linux kernel

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

Implementation based on *Futex* in Linux

Pthread locks: Example

```
#include <pthread.h>

int count=0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;

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

Agenda

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

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.

Operations semantic

Semaphores

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

Operations semantic

- P: `N = N-1`, block thread if N $<$ 0
  - P stands for Proberen (Dutch) – try

Semaphores

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

Operations semantic

- P: `N = N-1`, block thread if $N < 0$
  - P stands for Proberen (Dutch) – try
- V: `N = N+1`, wake up one thread if $N <= 0$
  - V stands for Verhogen (Dutch) – increment

Semaphores

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

Operations semantic

- P: `N = N-1`, block thread if N $<$ 0
  - P stands for Proberen (Dutch) – try
- V: `N = N+1`, wake up one thread if N $<=$ 0
  - 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()`

Mutual exclusion with semaphores

- Initializing a semaphore with value N can be seen as providing it with N *tokens*

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 $N = 1$

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

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 <pthread.h>

int count=0;
sem_t count_mutex;

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

Agenda

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

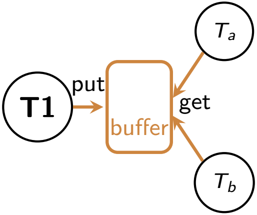
Condition Variables

Monitors

More on synchronization

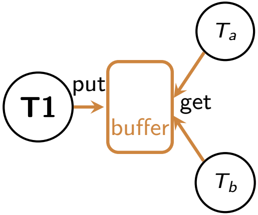
Specification of the problem

Recall



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

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)

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 (waste CPU time, slow down other threads)

Cooperation

Cooperation = Synchronization + Communication

Cooperation

Cooperation = Synchronization + Communication

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

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

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

    sem_wait(&empty);

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

    sem_post(&full)
  }
}
```

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

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

    sem_post(&empty);

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

Producer-Consumer with semaphores

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

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

    sem_wait(&empty);
    sem_wait(&mutex)
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    sem_post(&mutex)
    sem_post(&full)
  }
}
```

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

    /* 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

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

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

Condition variables (pthreads)

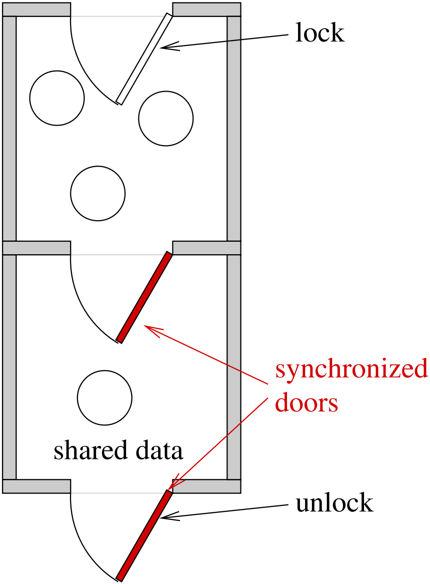
- A condition variable is a shared variable to which is associated a set of waiting threads.
- It allows a thread to explicitly put itself into a *waiting mode*.
- It is used together with a mutex: When a thread puts itself to wait, the corresponding mutex is released.
- It is often associated to a "logical condition" (hence the name)

Condition variables (pthreads)

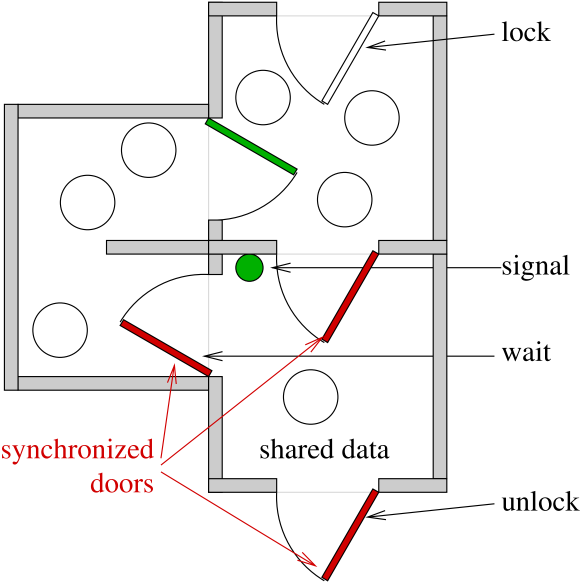
Interface

- `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



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);
    if (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);
    if (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 */
  }
}
```

Does it work with many readers and many writers?

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 */
  }
}
```

Always put a `while` around the waiting on a condition!

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 conditions 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 conditions 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

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

Agenda

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

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

Reminder: Threads

A Multi-Threaded Application

Mutual Exclusion

Semaphores

The Producer-Consumer Problem

Condition Variables

Monitors

More on synchronization

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