

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    printf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    printf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



Synchronization

Condition Variables

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

License Information

This work is licensed under the license



Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

① **BY:** Credit must be given to you, the creator.

② **NC:** Only noncommercial use of your work is permitted.

Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

③ **ND:** No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:

<https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1>

Condition Variables

❖ A condition variable is a synchronization primitive that allows threads to wait until a particular condition is true. It is used in conjunction with a mutex to protect shared data and to coordinate the execution of threads.

Condition variables provide a place for threads to rendez-vous

❖ Condition variables allow threads to wait in a race-free way for an arbitrary condition to occur

A mutex (mutual exclusion) is a lock that ensures that only one thread can access a resource at a time.

- The condition itself is protected by a mutex
- A thread must first lock the mutex to change the condition state
- Other threads will not notice the change until they acquire the mutex, because the mutex must be locked to be able to evaluate the condition

Race-Free: This means that the condition variables prevent race conditions, where the outcome depends on the sequence or timing of uncontrollable events.

❖ POSIX, C11, and C11++ have similar primitives

Thread Synchronization: When a thread changes the condition variable, other threads waiting on this condition will not notice the change until they acquire the mutex. This ensures that the condition is evaluated in a thread-safe manner.

CVs in POSIX

For more details see the reference documentation

POSIX Standard: POSIX (Portable Operating System Interface) is a family of standards specified by the IEEE for maintaining compatibility between operating systems. Pthreads is the POSIX standard for threads, providing a set of APIs for creating and managing threads.

Type\	Meaning
<code>int pthread_cond_init (...);</code>	Initializes the condition variable.
<code>int pthread_cond_destroy (...);</code>	Frees all the resources used by the condition variable. <small>This function frees all resources associated with the condition variable. It should be called when the condition variable is no longer needed.</small>
<code>int pthread_cond_signal (...);</code>	Wakes up one of any number of threads that are waiting for the specified condition variable.
<code>int pthread_cond_broadcast (...);</code>	Wakes up all theads waiting for the specified condition variable.
<code>int pthread_cond_wait (...);</code>	Blocks the calling thread amd release the mutex. A thread mush hold the mutex before calling.
<code>int pthread_cond_timedwait (...);</code>	Like wait but blocks the calling thread only until the time specified by the argument.

CVs in C

For more details see the reference documentation

The C11 standard introduced condition variables as part of the standard library for the C programming language. This provides a standardized way to use condition variables in C programs.

Type\	Meaning
<code>int cnd_init(cnd_t *cond);</code>	Initializes the condition variable pointed by cond. <small>This function initializes the condition variable pointed to by cond. It must be called before the condition variable can be used.</small>
<code>void cnd_destroy(cnd_t *cond);</code>	Frees all the resources used by cond. <small>This function frees all resources used by the condition variable pointed to by cond. It should be called when the condition variable is no longer needed.</small>
<code>void cnd_signal(cnd_t *cond);</code>	Wakes up one of any number of threads that are waiting for the specified condition variable.
<code>void cnd_broadcast(cnd_t *cond);</code>	Wakes up all threads waiting for the specified condition variable.
<code>void cnd_wait(cnd_t *cond, mtx_t *mtx);</code>	Blocks the calling thread and release the mutex. A thread must hold mtx before calling.
<code>void cnd_timedwait(cnd_t *cond, mtx_t *mtx, const struct timespec *ts);</code>	Like cnd_wait but blocks the calling thread only until the time specified by the argument ts.

CVs in C++

For more details see the reference documentation

- ❖ The C++ standard library defines
 - The class `std::condition_variable`
 - In the header `<condition_variable>`
- ❖ The library has the following member functions

Type\	Meaning
<code>wait()</code>	Takes a reference to a <code>std::unique_lock</code> that must be locked by the caller as an argument, unlocks the mutex and waits for the condition variable.
<code>notify_one()</code>	Notify a single waiting thread, mutex does not need to be held by the caller.
<code>notify_all()</code>	Notify all waiting threads, mutex does not need to be held by the caller.

The `cv.wait()` function takes a `std::unique_lock` and a predicate (a lambda function in this case). The `std::unique_lock` manages the mutex, and the predicate is a function that returns a boolean value indicating whether the condition is met. The `cv.wait()` function will block the thread until the predicate returns true.

CV Usage

We focus of the POSIX version

❖ Define and initialize a CV

```
#include <pthread.h>

pthread_cond_t cond;
pthread_mutex_t lock;
int done;
```

CV definition
(type pthread_cond_t)

CV must be used with a mutex and a condition

CV initialization

Attributes set to NULL

```
pthread_mutex_init (&m, NULL);
pthread_cond_init (&cv, NULL);
done = 0;
```

At the end, de-initialize the CV and free its memory (If allocated dynamically)

```
pthread_cond_destroy (&cv);
```

Alternative definition and initialization

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

CV Usage

❖ Signal a CV

```
1. pthread_mutex_lock (&m) ;  
2. done = 1;  
3. pthread_cond_signal (&cv) ;  
4. pthread_mutex_unlock (&m) ;
```

Signaling the CV must be protected by the thread

The function `pthread_cond_wait` is designed to unlock the mutex and put the thread into a waiting state atomically, meaning these two operations are performed as a single, uninterruptible operation. This is crucial to prevent race conditions where another thread might get a chance to lock the mutex after it's unlocked but before the thread enters the waiting state. Similarly, when a thread is awakened from a condition variable (using `pthread_cond_signal` or `pthread_cond_broadcast`), it automatically re-acquires the mutex before `pthread_cond_wait` returns. This ensures that the thread can safely proceed with its execution, knowing that it has exclusive access to the shared resources protected by the mutex. So, while the condition variable doesn't technically lock and unlock the mutex on its own, the functions associated with it (`pthread_cond_wait`, `pthread_cond_signal`, etc.) are designed to handle the mutex in a way

➤ Function “signal” is used to notify threads that a condition has been satisfied

- **thread_cond_signal** will wake up at least one thread waiting on the condition
- **pthread_cond_broadcast** will wake up all threads waiting on the condition

CV Usage

❖ Wait on a CV

```
1. pthread_mutex_lock (&m) ;  
   while (done == 0)  
       pthread_cond_wait (&cv, &m) ;  
   pthread_mutex_unlock (&m) ;
```

1. The thread obtains the mutex

- The mutex must be locked when we run cond-wait
- The mutex will be released in the epilogue

1. Lock the Mutex:

Before calling `pthread_cond_wait`, the thread must lock a mutex. This mutex is typically used to protect a shared resource or condition (like the ready variable in our example).

2. Call `pthread_cond_wait`:

When `pthread_cond_wait` is called, it atomically:

Unlocks the Mutex: This is done to allow other threads to acquire the mutex and modify the shared resource or condition. The unlocking happens atomically with the thread being placed on the condition variable's wait queue. This atomic operation is crucial because it prevents a race condition between unlocking the mutex and entering the waiting state.

Enters the Waiting State: The thread is placed into a waiting state where it will remain until it is specifically notified (via `pthread_cond_signal` or `pthread_cond_broadcast`), or until a spurious wakeup occurs.

3. Woken Up:

When the condition variable is signaled (using `pthread_cond_signal` or `pthread_cond_broadcast`), the waiting thread is awakened. However, before it can proceed with its execution, it must re-acquire the mutex it released when it entered `pthread_cond_wait`. This re-acquisition is automatic and handled internally by `pthread_cond_wait`.

4. Mutex is Re-locked:

The thread will not exit the `pthread_cond_wait` function until it successfully re-acquires the mutex. This ensures that when the thread resumes execution after waiting, it holds the mutex, just as it did when it initially called `pthread_cond_wait`.

5. Proceed with Execution:

Once the mutex is re-acquired, the thread can safely check the condition and proceed with its execution, knowing that it has exclusive access to the shared resources protected by the mutex.

CV Usage

❖ Wait on a CV

```
pthread_mutex_lock (&m);  
2. while (done == 0)  
    pthread_cond_wait (&cv, &m);  
pthread_mutex_unlock (&m);
```

2. The thread tests the predicate; if the predicate is

CV Usage

❖ Wait on a CV

pthread_cond_timedwait
has a timeout

```
pthread_mutex_lock (&m);  
while (done == 0)  
3.  pthread_cond_wait (&cv, &m);  
pthread_mutex_unlock (&m);
```

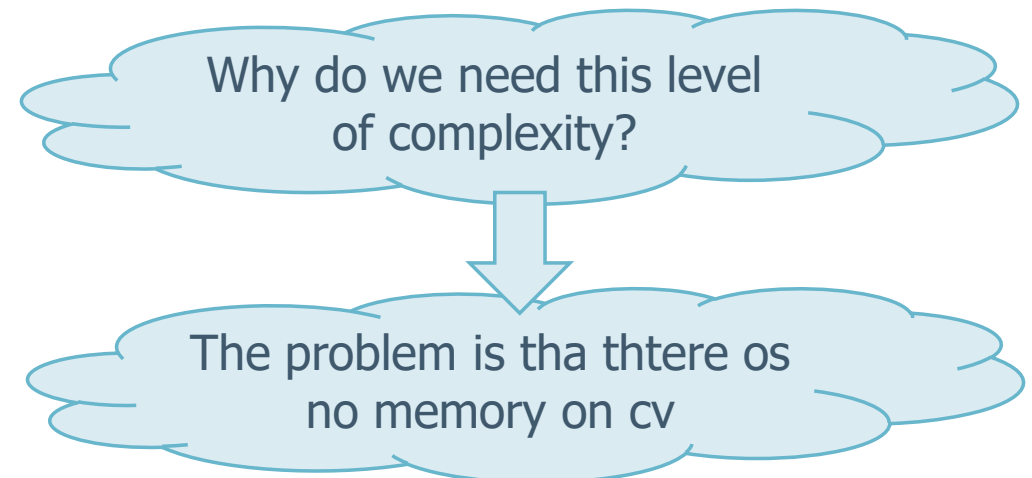
2. The thread tests the predicate; if the predicate is
3. **Satisfied**, the thread executes the wait on the CV which releases the mutex and it awaits on the condition variable
 - The mutex must be released to allow other threads to check the condition
 - When the condition variable is signaled, the thread wakes up and the predicate is checked again

CV Usage

❖ Wait on a CV

```
pthread_mutex_lock (&m);  
while (done == 0)  
    pthread_cond_wait (&cv, &m);  
4. pthread_mutex_unlock (&m);
```

2. The thread tests the predicate; if the predicate is
4. **Not satisfied**, the thread goes on and it unlock the mutex



Example

- ❖ Suppose `pthread_join` does not exist and we want to wait the termination of a thread

```
void *child(void *arg) {  
    ...  
    done = 1;  
    return NULL;  
}
```

`pthread_join` is used to wait for a thread to terminate and retrieve its exit status.
The slide suggests that if `pthread_join` does not exist, we need an alternative way to wait for the thread to finish.

```
int main(int argc, char *argv[]) {  
    pthread_t p;  
    pthread_create (&p, NULL, child, NULL);  
    pthread_join (p, &status);  
    return 0;  
}
```

No `pthread_join`

Example

Solution with spin-lock
(never use it)

❖ We can use polling

Using Polling (Inefficient Solution)

The slide discusses using polling as an alternative to pthread_join

➤ This is grossly inefficient as it wastes CPU cycles

```
void *child(void *arg) {
```

```
...
```

Child thread does some work

```
done = 1;
```

```
return NULL;
```

```
}
```

Polling involves repeatedly checking a condition in a loop.

```
int main(int argc, char *argv[]) {
```

```
pthread_t p;
```

```
pthread_create (&p, NULL, child, NULL);
```

```
while (done == 0); // spin
```

```
return 0;
```

```
}
```

Drawback: Polling is grossly inefficient as it wastes CPU cycles by continuously checking the condition.

In this example, the main thread continuously checks the done variable to see if the child thread has finished. Its the same trick that we used for conditional variables for wait

No pthread_join
(even if with
protection)

Th

Example

We can use a CV

```
void *child(void *arg) {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&cv);
    pthread_mutex_unlock (&m);
    return NULL;
}
```

```
pthread_mutex_t m = ...;
pthread_cond_t cv = ...;
int done = 0;
```

Initialization: The mutex (pthread_mutex_t m) and condition variable (pthread_cond_t cv) are initialized.

Initialization

Locks the mutex.
Sets done to 1.
Signals the condition variable to wake up the waiting thread.
Unlocks the mutex.

```
int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create (&p, NULL, child, NULL);
    pthread_mutex_lock (&m);
    while (done == 0)
        pthread_cond_wait (&cv, &m);
    pthread_mutex_unlock (&m);
    return 0;
}
```

pthread_join

Locks the mutex.
Enters a loop that waits on the condition variable until done is 1.
Unlocks the mutex after the condition is met.

Does it work?

Example

The slide provides a detailed explanation of how the condition variable mechanism works.

```
void *child(void *arg) {  
    pthread_mutex_lock (&m);  
    done = 1;  
    pthread_cond_signal (&cv);  
    pthread_mutex_unlock (&m);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    pthread_t p;  
    pthread_create (&p, NULL, child, NULL);  
    pthread_mutex_lock (&m);  
    while (done == 0)  
        pthread_cond_wait (&cv, &m);  
    pthread_mutex_unlock (&m);  
    return 0;  
}
```

The parent runs first:

1. It will acquire m, check "done", and as done=0, it will go to sleep releasing m
2. The child will run, set done to 1, signal the cv, release the mutex, and quit
3. The parent will be woken-up by the signal with the mutex locked, unlock the mutex, check cv, proceed, check "done", proceed, return

In this case the "job" is done by the **cv** on which the parent waits

Example

```

void *child(void *arg) {
    pthread_mutex_lock (&m);
    done = 1;
    pthread_cond_signal (&cv);
    pthread_mutex_unlock (&m);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create (&p, NULL, child, NULL);
    pthread_mutex_lock (&m);
    while (done == 0)
        pthread_cond_wait (&cv, &m);
    pthread_mutex_unlock (&m);
    return 0;
}

```

The child runs first:

1. It will set done to 1, signal the cv, unlock the mutex, and terminate. As there is no one waiting, the signal on cv **has no effect**
2. The parent will get to the critical section, lock the mutex, as done==1 it will go on, unlock the mutex, and terminate

Key Differences between this slide and the previous one.

1. Order of Execution:

Slide 4: The parent thread runs first, locks the mutex, checks done, and waits on the condition variable if done is 0. The child thread then sets done to 1, signals the condition variable, and releases the mutex.

Slide 5: The child thread runs first, sets done to 1, signals the condition variable, and releases the mutex. The parent thread then locks the mutex, checks done, and proceeds without waiting.

2. Effect of the Signal:

Slide 4: The signal from the child thread effectively wakes up the parent thread because the parent thread is already waiting on the condition variable.

Slide 5: The signal from the child thread has no effect because the parent thread is not yet waiting on the condition variable. The parent thread later checks done and proceeds without waiting.

3. Synchronization:

Slide 4: Proper synchronization is achieved because the parent thread waits on the condition variable, and the child thread's signal wakes it up.

Slide 5: Synchronization is still correct, but the signal from the child thread is essentially redundant because the parent thread checks done after the child thread has already set it.

Summary:

Slide 4: Demonstrates a scenario where the parent thread runs first and waits on the condition variable. The child thread's signal effectively wakes up the parent thread.

Slide 5: Demonstrates a scenario where the child thread runs first and sets done before the parent thread starts waiting. The parent thread checks done and proceeds without needing to wait.

Both slides illustrate correct use of condition variables and mutexes, but they highlight different execution orders and the impact on synchronization.

In this case the "job" is done by the variable **done** as the parent never does a wait

Example

Is the variable **done** required?

YEP TO AVOID INF LOOP

This slide questions whether the done variable is necessary and explains why it is required.

```
void *child(void *arg) {  
    pthread_mutex_lock (&m);  
    pthread_cond_signal (&cv);  
    pthread_mutex_unlock (&m);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    pthread_t p;  
    pthread_create (&p, NULL, child, NULL);  
    pthread_mutex_lock (&m);  
    pthread_cond_wait (&cv, &m);  
    pthread_mutex_unlock (&m);  
    return 0;  
}
```

The code is broken.

In fact, **iff** the child runs first:

1. It will signal the cv but as there is no one waiting, the signal has no effect
2. The parent will get to the critical section, lock the mutex, and wait on cv forever

Problem: If the child thread runs first, it will signal the condition variable, but if there is no one waiting, the signal has no effect. The parent thread will then wait on the condition variable forever.

Solution: The done variable is necessary to record the status that the threads are interested in knowing. It ensures that the parent thread can check the status and proceed accordingly.

Then, variable **done** records the status the threads are interested in knowing

Example

Is the **mutex** m required?

```
void *child(void *arg) {  
    done = 1;  
    pthread_cond_signal (&cv);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    pthread_t p;  
    pthread_create (&p, NULL, child, NULL);  
    while (done == 0)  
        pthread_cond_wait (&cv);  
    return 0;  
}
```

The code is broken.

There is a subtle **race condition**:

1. The parent runs first, and it checks done. As done==0 it is ready to go to sleep on the cond_wait, but before going on the wait the child runs
2. The child set done to 1 and it signal cv. But at this point the parent is not waiting, thus **the signal is lost**
3. The parent will go on the wait and wait forever

The mutex is required to protect the shared variable (done) and ensure that the condition variable wait and signal operations are properly synchronized.

This is not a correct implementation, because there is no mutex. Let us suppose it is correct just for the sake of the example

Using the mutex may not be always required around the signal but it is **always** required around the wait

Example

Is the **while** required or we can use a **if**?

This slide questions whether the while loop is necessary and explains why it is required.

```
void *child(void *arg) {  
    pthread_mutex_lock (&m);  
    done = 1;  
    pthread_cond_signal (&cv);  
    pthread_mutex_unlock (&m);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    pthread_t p;  
    pthread_create (&p, NULL, child, NULL);  
    pthread_mutex_lock (&m);  
    if (done == 0)  
        pthread_cond_wait (&cv, &m);  
    pthread_mutex_unlock (&m);  
    return 0;  
}
```

The code is broken.

More than one thread may be awoken, because `pthread_cond_broadcast` has been called or a race between two processors simultaneously woke two threads. The first thread locking the mutex will block all other threads. Thus, the predicate may have changed when the second thread gets the mutex. In general, whenever a CV returns, the thread should reevaluate the predicate

Solution: The while loop is necessary to re-check the condition after being awakened. This ensures that the thread only proceeds if the condition is actually met. The while loop handles spurious wakeups and ensures that the condition is re-evaluated.

Signaling a thread wakes-it up but there is **no** guarantee that when it runs the **state** will still **be the same. The while is required.**

Summary I

❖ When using a condition variable

- The mutex is used to protect the condition variable
- The mutex must be locked before waiting
- The wait will "atomically" unlock the mutex, allowing others access to the condition variable
- When the condition variable is signalled (or broadcast to) one or more of the threads on the waiting list will be woken-up and the mutex will be magically locked again for that thread

Key Points:

1. Mutex Protection:

The mutex is used to protect the condition variable and the shared resource or condition it represents. This ensures that only one thread can access or modify the shared resource at a time, preventing race conditions.

2. Mutex Locking:

The mutex must be locked before a thread can wait on the condition variable. This ensures that the thread has exclusive access to the shared resource while it checks the condition and decides whether to wait.

3. Atomic Unlocking:

When a thread calls `pthread_cond_wait`, it atomically unlocks the mutex and puts the thread into a waiting state. This atomic operation is crucial because it prevents a race condition between unlocking the mutex and entering the waiting state. Other threads can now lock the mutex and modify the shared resource or condition.

4. Signaling and Re-locking:

When the condition variable is signaled (using `pthread_cond_signal` or `pthread_cond_broadcast`), one or more threads on the waiting list are awakened. The awakened thread(s) will automatically re-lock the mutex before returning from `pthread_cond_wait`. This ensures that the thread resumes execution with exclusive access to the shared resource, allowing it to safely check the condition and proceed.

Summary II

❖ Condition variables allow a thread to notify other threads when something needs to happen

Key Points:

1. Notification Mechanism:

Condition variables allow a thread to notify other threads when a specific condition needs to be met. This notification mechanism is more efficient than polling because it avoids busy waiting.

2. Avoiding Busy Waiting:

Busy waiting involves repeatedly checking a condition in a loop, which wastes CPU cycles. Condition variables relieve the user of the burden of polling by putting the waiting thread to sleep until the condition is met.

This approach conserves CPU resources and improves the efficiency of the application.

➤ A condition variable relieves the user of the burden of polling some condition and waiting for the condition without wasting resources

➤ They avoid busy waiting

```
while (done == 0);
```



```
pthread_mutex_lock (&m);  
while (done == 0)  
    pthread_cond_wait (&cv, &m);  
pthread_mutex_unlock (&m);
```

- Used when one or more threads are waiting for a specific condition to come true

Summary III

❖ Condition variables versus semaphores

➤ Semaphores are very general and sophisticated

- They are expensive
- There are many cases in which they can do the same thing of a condition variable
 - A **condition variable** is essentially a waiting-queue and it needs a mutex
 - A **semaphore** is essentially a counter, a mutex, and a waiting queue

➤ Semaphores are used for more general synch schemes

- Used when there is a shared resource that can be available or unavailable based on some integer number of things

Exercise 01

- ❖ Only C++20 supports semaphores
 - In contrast to a mutex a semaphore is **not** bound to a thread
 - This means that the acquire and release call of a semaphore can happen on different threads
- ❖ Suppose C++20 does not exist yet
- ❖ Implement a **C++ semaphore** using a mutex and a CV

Solution with polling Never use it

Solution 01

```
struct Semaphore {  
    int count;  
    mutex m;  
    ...  
}
```

Constructor

```
Semaphore (int n) {  
    count = n;  
    return;  
}
```

At most **n**
workers in the
critical section

```
void sem_wait() {  
    while (1) {  
        while (count <= 0) {}  
        m.lock();  
        if (count <= 0) {  
            m.unlock();  
            continue;  
        }  
        count--;  
        m.unlock();  
        break;  
    }  
}
```

Polling

Polling
wait

Re-check after
acquiring the lock

If the sem cannot be
acquired, cycle (wait) again

```
void sem_signal () {  
    m.lock();  
    count++;  
    m.unlock();  
}
```

Solution with 2 mutexes **BUGGY**

Solution 02

```
struct Semaphore {  
    int count;  
    mutex m, wait;  
    ...  
}
```

Constructor

```
Semaphore (int n) {  
    count = n;  
    return;  
}
```

At most **n**
workers in the
critical section

The first mutex
is to protect the
CS, the second
one to make
threads wait

```
void sem_wait() {  
    m.lock();  
    cout--;  
    if (count < 0) {  
        m.unlock();  
        wait.lock();  
    } else {  
        m.unlock();  
    }  
}
```

Buggy because locks
have a unique owner

```
void sem_signal () {  
    m.lock();  
    count++;  
    if (count <= 0) {  
        wait.unlock();  
    }  
    m.unlock();  
}
```

Solution with a mutex
and a condition variable

Solution 03

```
#include <mutex>
#include <condition_variable>
using ...
struct Semaphore {
    int count;
    mutex m;
    condition_variable cv;
    ...
}
```

Constructor

```
Semaphore (int n) {
    count = n;
    return;
}
```

At most **n**
workers in the
critical section

```
void sem_wait() {
    unique_lock<mutex> lock(m);
    count--;
    while (count < 0) {
        cv.wait(lock);
    }
}
```

Mutex

Predicate

CV

Mutex

```
void notify( int tid ) {
    unique_lock<mutex> lock(m);
    count++;
    cv.notify_one();
}
```

Mutex

Predicate

CV

Exam of January 19,
2021

Exercise 02

- ❖ Write a C++ program in which
 - A thread **admin** initializes an integer variable **var** to 10 and then waits 3 **adder** threads
 - Each **adder** thread adds a random number between 2 and 5 to **var**
 - The program terminates when
 - All threads finish or
 - When **var** becomes equal or greater than 15
 - When the program ends the **admin** thread is awakened and prints the final value

Solution

Premises

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <fstream>

std::mutex m;
std::condition_variable adminCV;
std::condition_variable adderCV;
int var = 0;

void admin_f();
void adder_f();
```

Solution

```
int main() {  
    std::vector<std::thread> adders;  
  
    // Run admin thread  
    std::thread admin_t(admin_f);  
    for(int i=0; i<3; i++){  
        // Makes the seed different for each thread  
        srand ((unsigned)time(NULL));  
        // Run adder threads  
        adders.emplace_back(std::thread (adder_f));  
    }  
    for(auto &i: adders) {  
        i.join();  
    }  
    adminCV.notify_one();  
    admin_t.join();  
    return 0;  
}
```

Main

Run thread
admin

Wait for them

Run three
threads
adder_f

Solution

```
void admin_f () {  
    std::unique_lock<std::mutex> admin_lock{m};  
    var = 10;  
    cout << "Variable initialized to 10" << endl;  
  
    // Notify adders  
    adderCV.notify_all();  
  
    // Wait adders  
    while (var < 15)  
        adminCV.wait(admin_lock);  
  
    cout << "Variable value = " << var << endl;  
}
```

Mutex

Thread admin

Predicate

Set the condition for the adder (var=10) and wakes them (adderCV.notify_all). Then, is waits on its predicated and CV (adminCV)

CV

Mutex

Solution

```
void adder_f () {  
    std::unique_lock<std::mutex> adder_lock{m};  
  
    // Wait for initialization  
    while (var == 0) {  
        // Unlock the mutex  
        adderCV.wait(adder_lock);  
    }  
    // If var is over the threshold, notify admin and exit  
    if (var >= 15) {  
        adminCV.notify_one();  
        return;  
    } else {  
        int n = 2 + rand() % 4;  
        var += n;  
        cout << "Added = " << n << " Sum = " << var << endl;  
    }  
    return;  
}
```

Mutex

Adder threads

CV

Predicate

Mutex

Exam of January 16,
2023

Exercise 03

- ❖ Write a C++ program that operates on a vector of integers `v` managing the synchronization of the following threads
 - A thread **writer** adds a random number in the range `[1,10]` to the vector every 5 seconds
 - A thread **ui** constantly checks for user input from the console and update the global variable **command**
 - A thread **worker** executes the commands specified in the variable **command** when thread **ui** wakes it

Exercise 03

- The valid commands are the following
 - 0 terminates the program
 - 1 displays all elements in v
 - 2 displays the last element of v
 - 3 deletes all elements in v

Solution

```
#include <iostream>
#include <thread>
#include <queue>

using namespace std;

bool running = true;
int command = -1;
condition_variable cv;
mutex mx;
vector<int> vt;

... prototypes
```

Runs and waits
threads

```
int main() {
    cout << "START" << endl;
    thread t_wr(writer);
    thread t_u(ui);
    thread t_w(worker);
    t_wr.join();
    t_u.join();
    t_w.join();
    cout << "END" << endl;
}
```

Solution

The **writer** adds a random number in the range [1,10] to the vector every 5 seconds

Insert a new value in the array every 5 seconds

```
void writer() {  
    while(running) {  
        this_thread::sleep_for(chrono::milliseconds(5000));  
        unique_lock<mutex> l_w(mx);  
        vt.emplace_back(rand()%10+1);  
        l_w.unlock();  
    }  
    return;  
}
```

Add a value in the range [1,10]

Solution

```
void ui() {  
    int temp;  
    while(running) {  
        cout << "Command (0,1,2,3): " << endl;  
        cin >> temp;  
        unique_lock<mutex> l_ui(mx);  
        command = temp;  
        if(temp==0) {  
            running = false;  
        }  
        cv.notify_one();  
        l_ui.unlock();  
    }  
}
```

The ui checks for user input from the console and update the global variable command

Read user commands and update variable **command**

Solution

```
void worker() {  
    while(running) {  
        unique_lock<mutex> l_r(mx);  
        while(vt.empty() || command == -1)  
            cv.wait(l_r);  
        switch (command) {  
            case 1: cout << " ### Current elements: " << endl;  
                    for(auto &e: vt)  
                        cout << "id: " << e << endl;  
                    break;  
            case 2: cout << " ### Last element: " <<  
                    vt.back() << endl;  
                    break;  
            case 3: cout << " ### All elements removed" << endl;  
                    vt.clear();  
                    break;  
        }  
        l_r.unlock();  
    }  
}
```

The worker executes the commands specified in the variable `command` when thread `ui` wakes it

Execute commands in **command** (terminates, display, display, delete)

Exercise 04

C Implementation

- ❖ Implement a Producer-Consumer scheme with a **bounded** buffer of **size one**
 - The main thread runs NP producers and NC consumers
 - Producers and consumers communicate using a **single variable**
 - Each producer stores a predefined number of (random) integers in **buffer**
 - Each consumer displays (on standard output) a predefined number of integers, reading them from **buffer**
 - Use condition variables to perform synchronization

Solution 01

Buffer

Premises

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

Initially empty

```
pthread_cond_t  cv = PTHREAD_COND_INITIALIZER;  
pthread_mutex_t m  = PTHREAD_MUTEX_INITIALIZER;
```

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

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


Solution 01

```
void *producer(void *arg) {
    int i;
    int loops = int (args);
    for (i=0; i<loops; i++) {
        pthread_mutex_lock (&m);
        while (count==1)
            pthread_cond_wait(&cv, &m);
        enqueue (i);
        pthread_cond_signal(&cv);
        pthread_mutex_unlock(&m);
    }
}
```

Producer and Consumer
NP producers and NC
consumers

```
void *consumer(void *arg) {
    int i;
    int loops = int (args);
    for (i=0; i<loops; i++) {
        pthread_mutex_lock (&m);
        while (count==0)
            pthread_cond_wait(&cv, &m);
        int tmp = dequeue (i);
        pthread_cond_signal (&cv);
        pthread_mutex_unlock (&m);
        printf ("%d", tmp);
    }
}
```

Broken scheme
There is only **one** CV
A producer (consumer) can
wake-up another consumer
(producer)

Solution 02

```
void *producer(void *arg) {
    int i;
    int loops = int (args);
    for (i=0; i<loops; i++) {
        pthread_mutex_lock (&m);
        while (count==1)
            pthread_cond_wait(&empty, &m);
        enqueue (i);
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&m);
    }
}
```

Producer and Consumer
NP producers and NC
consumers

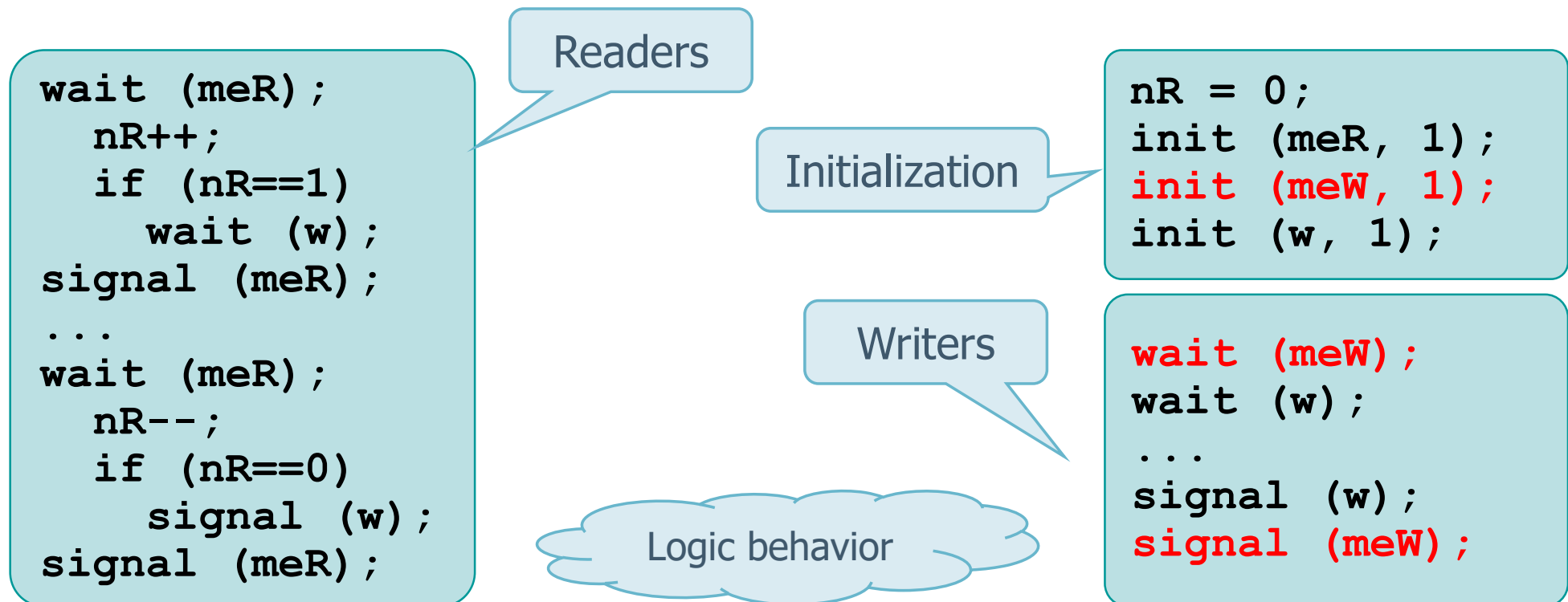
```
void *consumer(void *arg) {
    int i;
    int loops = int (args);
    for (i=0; i<loops; i++) {
        pthread_mutex_lock (&m);
        while (count==0)
            pthread_cond_wait(&full, &m);
        int tmp = dequeue (i);
        pthread_cond_signal (&empty);
        pthread_mutex_unlock (&m);
        printf ("%d", tmp);
    }
}
```

Correct scheme

Exercise 05

C Implementation

- ❖ Implement the First Reader-Writer scheme using
 - Mutexes
 - Condition Variables
 - Read-Write Locks (or shared mutexes)



Solution 01 (with mutexes)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include "pthread.h"
#include "semaphore.h"

#define N 20

typedef struct rw_s {
    int nr;
    pthread_mutex_t meR;
    pthread_mutex_t meW;
    pthread_mutex_t w;
} rw_t;

rw_t *rw;
```

With mutexes we have a 1:1 correspondence with the high-level (pseudo-code) solution

Number of concurrent readers and writers

nr, meR, meW, w
(see high-level solution)

Solution 01 (with mutexes)

```
int main (void) {  
    pthread_t th_a, th_b;  
    int i, v[N];  
    setbuf (stdout, NULL);  
    rw = (rw_t *) malloc (1 * sizeof(rw_t));  
    rw->nr = 0;  
    pthread_mutex_init (&rw->meR, NULL);  
    pthread_mutex_init (&rw->meW, NULL);  
    pthread_mutex_init (&rw->w, NULL);  
    for (i=0; i<N; i++) {  
        v[i] = i;  
        pthread_create (&th_a, NULL, reader, (void *) &v[i]);  
        pthread_create (&th_b, NULL, writer, (void *) &v[i]);  
    }  
    free (rw);  
    pthread_exit (NULL);  
}
```

Init mutex

No joins !

Run threads
(N readers, N writers)

Solution 01 (with mutexes)

```
static void *reader (void *arg) {  
    int *p = (int *) arg;  
    int i = *p;  
    pthread_mutex_lock (&rw->meR);  
    rw->nr++;  
    if (rw->nr == 1)  
        pthread_mutex_lock (&rw->w);  
    pthread_mutex_unlock (&rw->meR);  
    printf("Thread %d reading\n", i);  
    pthread_mutex_lock (&rw->meR);  
    rw->nr--;  
    if (rw->nr == 0)  
        pthread_mutex_unlock (&rw->w);  
    pthread_mutex_unlock (&rw->meR);  
    pthread_exit (NULL);  
}
```

Prologue

CS

Epilogue

```
wait (meR);  
nr++;  
if (nr==1)  
    wait (w);  
signal (meR);  
...  
wait (meR);  
nr--;  
if (nr==0)  
    signal (w);  
signal (meR);
```


Solution 01 (with mutexes)

```
static void *writer (void *arg) {  
    int *p = (int *) arg;  
    int i = *p;
```

Prologue

```
    pthread_mutex_lock (&rw->meW);  
    pthread_mutex_lock (&rw->w);  
    printf("Thread %d writing\n", i);  
    pthread_mutex_unlock (&rw->w);  
    pthread_mutex_unlock (&rw->meW);
```

CS

```
    pthread_exit (NULL);  
}
```

Epilogue

```
wait (meW);  
wait (w);  
...  
signal (w);  
signal (meW);
```

Solution 02 (with CVs)

As previous solution

Several different solutions are possible

...

```
typedef struct rw_s {  
    pthread_mutex_t lock;  
    pthread_cond_t turn;  
    int nr, nw;  
} rw_t;
```

```
rw_t *rw;
```

Lock mutex
Conditional variable for the turn
(to readers or to writers)
Condition (2 counters)

Solution 02 (with CVs)

```
int main (void) {
    pthread_t th_a, th_b;
    int i, v[N];
    setbuf (stdout, NULL);
    rw = (rw_t *) malloc (sizeof(rw_t));
    pthread_mutex_init (&rw->lock, NULL);
    pthread_cond_init (&rw->turn, NULL);
    rw->nr = rw->nw = 0;
    for (i=0; i<N; i++) {
        v[i] = i;
        pthread_create (&th_a, NULL, reader, (void *) &v[i]);
        pthread_create (&th_b, NULL, writer, (void *) &v[i]);
    }
    free (rw);
    pthread_exit (NULL);
}
```

The main program is similar to the one with mutexes

No joins !

Solution 02 (with CVs)

```
static void *reader (void *arg) {  
    int *p = (int *) arg;  
    int i = *p;  
    pthread_mutex_lock (&rw->lock);  
    while (rw->nw > 0)  
        pthread_cond_wait (&rw->turn, &rw->lock);  
    rw->nr++;  
    pthread_mutex_unlock (&rw->lock);  
    printf ("Thread %2d reading\n", i);  
    pthread_mutex_lock (&rw->lock);  
    rw->nr--;  
    if (rw->nr==0) pthread_cond_broadcast (&rw->turn);  
    pthread_mutex_unlock (&rw->lock);  
    pthread_exit (NULL);  
}
```

Prologue

```
wait (meR);  
nr++;  
if (nr==1)  
    wait (w);  
signal (meR);  
...  
wait (meR);  
nr--;  
if (nr==0)  
    signal (w);  
signal (meR);
```

CS

Epilogue

Solution 02 (with CVs)

```
static void *writer (void *arg) {  
    int *p = (int *) arg;  
    int i = *p;  
    pthread_mutex_lock (&rw->lock);  
    while (rw->nw > 0 || rw->nr > 0)  
        pthread_cond_wait (&rw->turn, &rw->lock);  
    rw->nw++;  
    pthread_mutex_unlock (&rw->lock);  
    printf ("Thread %2d writing\n", i);  
    pthread_mutex_lock (&rw->lock);  
    rw->nw--;  
    pthread_mutex_unlock (&rw->lock);  
    pthread_cond_broadcast (&rw->turn);  
    pthread_exit (NULL);  
}
```

Prologue

```
wait (meW);  
wait (w);  
...  
signal (w);  
signal (meW);
```

CS

Epilogue

Solution 03 (with reader-writer locks)

```
#define N 20
```

```
pthread_rwlock_t rw;
```

```
int main (void) {
```

```
    pthread_t th_a, th_b;
```

```
    int i, v[N];
```

```
    setbuf (stdout, NULL);
```

```
    pthread_rwlock_init (&rw, NULL);
```

```
    for (i=0; i<N; i++){
```

```
        v[i] = i;
```

```
        pthread_create (&th_a, NULL, reader, (void *) &v[i]);
```

```
        pthread_create (&th_b, NULL, writer, (void *) &v[i]);
```

```
    }
```

```
    pthread_exit(NULL);
```

```
}
```

This is all we need
with RW locks

Precedence depends on
how RW locks are
implemented

The main program is similar to
the one with mutexes

Solution 03 (with reader-writer locks)

```
static void *reader(void *arg) {
    int *p = (int *) arg;
    int i = *p;
    pthread_rwlock_rdlock (&rw);
    printf ("Thread %d reading\n", i);
    pthread_rwlock_unlock (&rw);
    pthread_exit (NULL);
}

static void *writer(void *arg) {
    int *p = (int *) arg;
    int i = *p;
    pthread_rwlock_wrlock (&rw);
    printf ("Thread %d writing\n", i);
    pthread_rwlock_unlock (&rw);
    pthread_exit (NULL);
}
```

```
wait (meR);
nR++;
if (nR==1)
    wait (w);
signal (meR);
...
wait (meR);
nR--;
if (nR==0)
    signal (w);
signal (meR);
```

```
wait (meW);
wait (w);
...
signal (w);
signal (meW);
```