

Programación Multihilo en C/C++/SDL



TPV2
Samir Genaim

Lecture Notes
Samir Genaim
TPV2 2021/22
Facultad de Informática
Universidad Complutense de Madrid

Hilos y Procesos

- ◆ Son dos maneras de implementar/usar concurrencia/parallelismo.
- ◆ Al ejecutar un programa, el sistema operativo crea un proceso que tiene (por lo menos) un hilo para ejecutar el método main.
- ◆ El proceso puede duplicarse, creando una copia "idéntica", pero no comparten nada — cada uno tiene su memoria.
- ◆ Los procesos pueden comunicar entre ellos mediante signals.
- ◆ Cada proceso puede tener varios hilos que comparten sus recursos (memoria , etc.)

Crear un Proceso

Duplica el proceso, los dos siguen ejecutando en la siguiente instrucción. Para el nuevo proceso `fork` devuelve 0, para el que ha ejecutado `fork` devuelve el identificador del nuevo proceso.

El proceso nuevo es (casi) idéntico. Tiene copia idéntica de la memoria (incluso la memoria heap). No se duplican los threads del proceso.

```
...
fork();
std::cout << "My process ID is " << getpid() << std::endl;
...
```

```
...
if (fork() == 0) {
    std::cout << "I am the new process! " << getpid() << std::endl;
} else {
    std::cout << "I am the original process! " << getpid() << std::endl;
}
...
```

¿Qué Significa “Ejecutando Simultáneamente (en Paralelo)?”?

- ◆ Esto depende de muchos factores. P. ej., si tenemos n hilos ejecutando en paralelo en una plataforma con $m \geq n$ CPUs/Cores/Threads, podemos pensar que realmente están ejecutando en paralelo, ...
- ◆ Sin embargo, normalmente tenemos hilos (en todos los programas) mas que CPUs/Cores/Threads.
- ◆ Un gestor de hilos (a nivel del sistema operativo, la maquina virtual, etc.) alterna la ejecución de los hilos, es decir, se ejecuta un poco de cada hilo ...

Librerías de threads ...

- ◆ Quizá `pthreads` es la más común, existe para sistemas operativos que siguen el estándar POSIX – `pthread` es POSIX THREADS
- ◆ Windows no tiene soporte para `pthreads` de forma nativa (es decir, al nivel del sistema operativo), tiene otras librerías con API distinto, pero hay varios wrappers que implementan la API de `pthreads` en windows.
- ◆ El estándar de C++11 ya tiene definiciones para incluir clases estándar para usar `threads`, aparte de que simplifica mucho la API, esto hace que nuestro código es portable ...

Crear un Hilo con pthread

```
#include <pthread.h>
```

```
void *foo(void *){
    std::cout << "Hello from thread" << pthread_self() << std::endl;
    return NULL;
}
```

```
int main() {
    pthread_t t1;
    pthread_t t2;
```

```
    pthread_create(&t1, NULL, foo, NULL);
    pthread_create(&t2, NULL, foo, NULL);
```

```
// DO OTHER THINGS
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
```

```
    std::cout << "Main is done!" << std::endl;
    return 0;
}
```

El método donde empieza la ejecución del hilo tiene que tener un parámetro `void*` y devolver `void*`

Un tipo que representa un hilo, un puntero a un struct con varios atributos ...

Crear un hilo y empieza a ejecutar foo en ese hilo, el main sigue ejecutando. Devuelve 0 si no hay errores ...

El hilo que ejecuta `pthread_join`, espera/bloquea hasta que el hilo del primer parámetro termine. El segundo parámetro (si no es NULL) se usa para recibir la salida del método foo.

Crear un Hilo en C++11

```
#include <thread>
```

```
void foo() {  
    std::cout << "Hello from thread " << std::this_thread::get_id() << std::endl;  
}
```

```
int main(){  
    std::thread t1(foo);  
    std::thread t2(foo);
```

```
    t1.join();  
    t2.join();
```

```
    std::cout << "Main is done!" << std::endl;  
    return 0;
```

```
}
```

A partir de c++11, hay una clase que gestiona el hilo. La implementación por detrás puede ser distinta, pero el código funciona con cualquier sistema operativo, etc.

La clase `thread` crea un hilo y empieza a ejecutar `foo` en ello, el `main` sigue ejecutando

El hilo que ejecuta `t.join()` espera hasta que el hilo `t` termine ...

Todo es más sencillo en c++11 ...

En c++11 se puede llamar a cualquier función (incluso lambda-expressions) con cualquier parámetros, mientras que en pthreads sólo a un método que recibe un sólo parámetro de tipo void* y devuelve void*

```
void foo(int x, int y) {  
    std::cout << x << ":" << y << std::endl;  
    ...  
}  
  
int main() {  
    std::thread t1(foo, 4, 5);  
    ...  
    std::thread t2([]() {  
        foo(1,5);  
    });  
}
```

```
void *foo(void *x) {  
    std::cout << *(int *)x << std::endl;  
}  
  
int main() {  
    pthread_t t1;  
    int x=4;  
    pthread_create(&t1, NULL, foo, &x);  
    ...  
}
```

Threads y SDL ...

- ◆ En SDL también quisieron desarrollar un API único para programación Multi-hilos ...
- ◆ El objetivo es que los programadores puedan hacer uso de hilos en videojuegos y que al mismo tiempo el código funciona igual en todos los sistemas operativos. También querían tener un API única para los desarrolladores de la librería SDL, para que su código Multi-hilos funcione en todos los sistemas operativos.
- ◆ En principio los conceptos son los mismos en pthread, c++11 thread y SDL threads, la diferencia es en la sintaxis ...

SDL Threads ...

```
#include <SDL.h>
```

```
int foo(void *ptr) {  
    std::cout << "Hello from thread" << SDL_ThreadID() << std::endl;  
    return 1;  
}
```

```
int main() {  
    SDL_Thread *t;  
    int threadReturnValue;  
  
    t = SDL_CreateThread(foo, "Test Thread", (void *) NULL);  
}
```

```
if (t == NULL) {  
    std::cout << "SDL_CreateThread failed: " << SDL_GetError();  
} else {  
    SDL_WaitThread(t, &threadReturnValue);  
    std::cout << "Thread returned value: " << threadReturnValue;  
}  
  
return 0;  
}
```

El main de un hilo recibe un sólo parámetro de tipo void* y devuelve int

El tipo de datos para un hilo

Crear y empezar el hilo

SDL_WaitThread es como pthread_join

¿Qué escribe este programa?

```
int c;
int foo(void *ptr) {
    for(int i=0; i<10000000; i++) c=c+1;
}
int moo(void *ptr) {
    for(int i=0; i<10000000; i++) c=c-1;
}
int main(int argc, char *argv[]) {
    SDL_Thread *t1, *t2;
    t1 = SDL_CreateThread(foo, "...", (void *) NULL);
    t2 = SDL_CreateThread(moo, "...", (void *) NULL);
    SDL_WaitThread(t1, NULL);
    SDL_WaitThread(t2, NULL);
    std::cout << c << std::endl;
}
```

c=c+1 y c=c-1 no son instrucciones atómicas, a nivel de código assembly serían:

load c	load c
push 1	push 1
add	sub
store c	store c

¿Qué escribe este programa?

```
List<int> c;  
  
int foo(void *ptr) {  
    for(int i=0; i<1000; i++)  
        c.add(i);  
}  
  
int main(int argc, char *argv[]) {  
    SDL_Thread *t1, *t2;  
  
    t1 = SDL_CreateThread(foo, ...);  
    t2 = SDL_CreateThread(foo, ...);  
  
    SDL_WaitThread(t1, NULL);  
    SDL_WaitThread(t2, NULL);  
  
    std::cout << c.size() << std::endl;  
}
```

La clase List implementa una lista enlazada ...

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio durante la carrera sólo se podrán utilizar para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución posterior (incluida la divulgación en redes sociales o en la web) sin la autorización de la persona infractora, puede vulnerar la normativa de protección de la propiedad intelectual y generar responsabilidad civil y/o penal. Si encuentra vulneraciones de estos derechos, avíselos en denuncia@ucm.es, o reporte al correo contenido@ucm.es

Sección Crítica (Critical Section)

- ★ Una sección critica es una parte del código de un programa en la cual se accede a (o modifica) un recurso compartido (estructura de datos, etc.) que no debe acceder por más de un hilo a la vez.
- ★ Para conseguir esto, se necesita un mecanismo de sincronización en la entrada y salida de la sección crítica para asegurar la utilización exclusiva del recurso (**Exclusión Mutua -- Mutual Exclusion**)
- ★ Si un hilo intenta entrar en la sección critica mientras otro lo está ejecutando, pues bloquea hasta que el primero acaba de ejecutarlo, etc.
- ★ La sección critica puede ser en distintas partes del programa (como en el ejemplo de contador)

Locks (Mutex)

```
// Pseudo code,  
//  
Lock x;  
  
void f() {  
    ...  
    lock(x);  
    ... A ...  
    unlock(x);  
    ...  
}  
  
void g() {  
    ...  
    lock(x);  
    ... B ...  
    unlock(x);  
    ...  
}
```

- ★ Es un concepto de sincronización que permite conseguir **Mutual Exclusion**
- ★ Se puede imaginar un **Lock** como se fuera una variable de tipo **Boolean**
- ★ **lock(x)**: cambia **x** a **true** si su valor es **false** y entra en la sección critica, si su valor es **true** bloquea hasta que cambie a **false** y vuelve a cambiarlo a **true** y entra en la sección critica
- ★ **unlock(x)**: cambia **x** a **false**, permitiendo a otros hilos que están esperando entrar en la sección critica
- ★ Por el momento, no es importante como implementamos **lock** y **unlock** esto lo vais a ver en programación concurrente ...

Mutex en SDL

```
SDL_mutex *mutex;
```

Declaración del Mutex

```
mutex = SDL_CreateMutex()
```

Inicialización del Mutex

```
if (SDL_LockMutex(mutex) == 0) {
```

```
// Critical section
```

```
    SDL_UnlockMutex(mutex);
```

```
} else {
```

```
    // something went wrong
```

```
}
```

```
SDL_DestroyMutex(mutex);
```

Cerrar y Abrir. La llamada a LockMutex bloquea si el mutex está cerrado hasta que lo liberan. Si devuelve algo distinto de 0 es que ha ocurrido algún error

Si no vamos a usar más el mutex hay que liberar la memoria correspondiente

Mutex sin Bloquear

```
status = SDL_LockMutexTry(mutex)
```

```
if (status == 0) {
```

```
// Critical section
```

```
SDL_UnlockMutex(mutex);
```

```
} else if (status == SDL_MUTEX_TIMEDOUT) {
```

```
// mutex not available, it is locked
```

```
} else {
```

```
// something went wrong
```

```
}
```

Se puede intentar a cerrar
el mutex sin bloquear

Usar Locks con Cuidado

```
void f() {  
    ...  
    lock(x);  
    ...  
    lock(z);  
    ...  
    unlock(z);  
    ...  
    unlock(x);  
    ...  
}
```

```
void g() {  
    ...  
    lock(z);  
    ...  
    lock(x);  
    ...  
    unlock(x);  
    ...  
    unlock(z);  
    ...  
}
```

1. T1: lock(x) succeeds
2. T2: lock(z) succeeds
3. T1: lock(z) **blocks** T1
4. T2: lock(x) **blocks** T2
5. T1 está esperando que T2 libera z y T2 está esperando que T1 libera x
6. Ningún hilo puede hacer algo — **Deadlock!**

Suponiendo que tenemos 2 locks x y z, y que un hilo T1 ejecuta f y otro hilo T2 ejecuta g, ¿cuál es el problema en este código?

Mutex en pthread

```
pthread_mutex_t mutex;
```

Declarar un Mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                      const pthread_mutexattr_t *mutexattr);
```

inicializar

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Cerrar y Abrir
el mutex

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Liberar

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Solo autorización expresa, los materiales
entregados a estudiantes por cualquier medio
durante la carrera sólo se podrán utilizar
en el estudio de la asignatura correspondiente
en la Universidad Complutense de Madrid.
La publicación o distribución posterior
(incluida la divulgación en redes sociales)
puede vulnerar la normativa de protección
de datos y/o la de propiedad intelectual y
generar responsabilidad de la persona infractora.
Si encuentra este material en otro sitio web
o reporta contenido@ucm.es

Mutex en C++11

```
#include <mutex>
```

Declarar, no hace falta inicializar

```
std::mutex x;
```

```
void x.lock();
```

Cerrar y Abrir el mutex

```
bool x.try_lock();
```

Todos los errores se avisan mediante excepciones

```
void x.unlock();
```

Variables Condicionales

Suponemos que dos funciones **f** y **g** están ejecutando en dos hilos T1 y T2. La función **f** quiere esperar hasta que cumpla una condición. La función **g** es la que satisface esta condición. ¿Cómo se puede hacer este tipo de sincronización?

```
int c;  
...  
int f(void* x) {  
    while (c == 0);  
    ...  
}  
  
int g(void* x) {  
    ...  
    c = 1;  
}  
  
int main() { ... }
```

Se puede sincronizar usando un bucle **while** en **f** para esperar hasta que la condición sea **true** — se puede usar **mutex** para el acceso a 'c'

El problema es que el método **f** consume recursos, p.ej., CPU, mientras está esperando a que cumpla la condición.

Una mejor solución sería dejar que el hilo T1 (que ejecuta **f**) duerma y despertarlo cuando cumpla la condición.

Variables Condicionales

```
int c = 0;  
  
SDL_mutex *mutex;  
SDL_cond *cond;  
  
int f(void *ptr) {  
    SDL_LockMutex(mutex);  
    while (c == 0) {  
        SDL_CondWait(cond, mutex);  
    }  
    SDL_UnlockMutex(mutex);  
    // ...  
}  
  
int g(void *ptr) {  
    sleep(50);  
    SDL_LockMutex(mutex);  
    c = 1;  
    SDL_CondSignal(cond);  
    SDL_UnlockMutex(mutex);  
    // ...  
}
```

El hilo que ejecuta la instrucción **SDL_CondWait** duerme hasta que reciba un signal a través de la variable **cond**, además, al mismo tiempo, suelta el **mutex** para que otros puedan usarlo.

Al recibir el signal se despierta, es decir vuelve de **SDL_CondWait**, pero no empieza a ejecutar hasta que consiga el **mutex** otra vez.

El hilo que modifica **c** avisa a todas los hilos que están durmiendo esperando un signal a través de **cond**.

El uso de **mutex** es para que **c==0** y **SDL_CondWait** ejecutan de manera atómica, es decir el valor de **c** no se modifica entre esas 2 instrucciones, si esto ocurre puede recibir la notificación antes de ir a dormir y así nunca se despierta

Variables Condicionales - main

```
int main(int argc, char *argv[]) {  
    SDL_Thread *t1, *t2;  
  
    mutex = SDL_CreateMutex();  
    cond = SDL_CreateCond();  
  
    t1 = SDL_CreateThread(f,...);  
    t2 = SDL_CreateThread(g,...);  
  
    SDL_WaitThread(t1, NULL);  
    SDL_WaitThread(t2, NULL);  
  
    SDL_DestroyMutex(mutex);  
    SDL_DestroyCond(cond);  
  
    return 0;  
}
```

Crear el Mutex y la variable condicional ...

Liberar la memoria, etc.

Variables Condicionales - while/if

¿Porque usamos un bucle **while** y no simplemente un **if**?

```
while ( c == 0){  
    SDL_CondWait(cond, mutex);  
}
```

```
if ( c == 0) {  
    SDL_CondWait(cond, mutex);  
}
```

Lecture Notes
Samir Genaim
TPV2 2021/22
Universidad Complutense de Madrid
Facultad de Informática

Variables Condicionales - while/if

SDL_CondSignal(cond)

Despertar uno de los hilos que están esperando a con

SDL_CondBroadcast(cond)

Despertar a todos los hilos que están esperando a con

Usamos un **while** porque un hilo puede despertarse aunque su condición **no cumple**. Esto también puede ocurrir sin llamadas a **SDL_CondBroadcast** – spurious wakeup ...

```
while ( c == 0 ) {  
    SDL_CondWait(cond, mutex);  
}
```

Producer/Consumer



- ◆ Generar y Procesar datos ...
- ◆ Partes del programa pueden generar datos ...
- ◆ Hay “Trabajadores” que procesan estos datos ...
- ◆ El procesamiento se hace en hilos distintos de los hilos de los generadores ...
- ◆ El tipo de estas datos dependen del contexto ...

Consumer

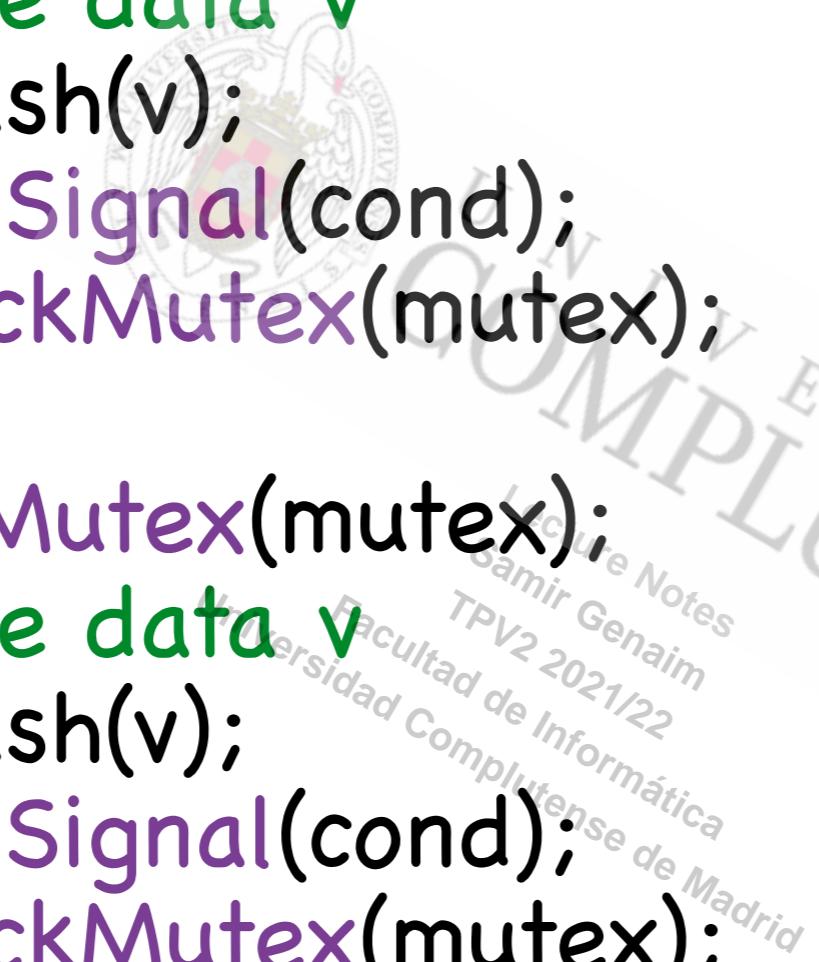
```
int consumer( ... ) {  
    while ( true ) {  
        SDL_LockMutex(mutex);  
        while (queue_.size() == 0){  
            SDL_CondWait(cond, mutex);  
        }  
        int v = queue_.front();  
        queue_.pop();  
        SDL_UnlockMutex(mutex);  
        std::cout << "Processing :" << v << std::endl;  
    }  
}
```

Los consumers duermen cuando no hay tareas y se despiertan cuando llega una tarea ...

Producer

```
int producer(...) {  
    ...  
    SDL_LockMutex(mutex);  
    // generate data v  
    queue_.push(v);  
    SDL_CondSignal(cond);  
    SDL_UnlockMutex(mutex);  
  
    ...  
    SDL_LockMutex(mutex);  
    // generate data v  
    queue_.push(v);  
    SDL_CondSignal(cond);  
    SDL_UnlockMutex(mutex);  
  
    ...  
}
```

Los generadores avisan cuando generan datos ...



Producer/Consumer

```
std::queue<int> queue_;
```

```
SDL_mutex *mutex;  
SDL_cond *cond;
```

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

```
...
```

```
    mutex = SDL_CreateMutex();
```

```
    cond = SDL_CreateCond();
```

```
    t1 = SDL_CreateThread(producer, "t1", ...);
```

```
    t2 = SDL_CreateThread(consumer, "t2", ...);
```

```
...
```

```
}
```

La variable condicional
es común ...

Salvo autorización expresa, los materiales
entregados a estudiantes por cualquier medio
durante la carrera sólo se podrán utilizar
para el estudio de la asignatura correspondiente
en la Universidad Complutense de Madrid.
La publicación o distribución posterior
(incluida la divulgación en redes sociales
o servicios de comunicación en Internet)
o servicios de comunicación en Internet)
de datos y/o la de propiedad intelectual y
generar responsabilidad de la persona infractora.
Si encuentra este material en otro sitio web,
avíselo a denunciacontenidos@ucm.es
o reportcontent@um.es

Blocking queue ...

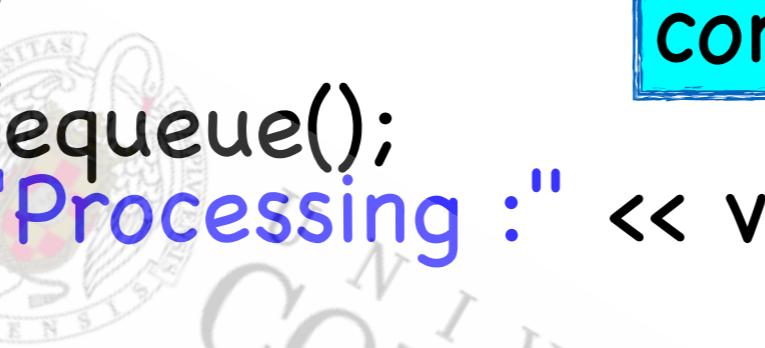
```
BlockingQueue<int> queue_;
```

```
int consumer(void *ptr) {
    int v = 0;
    while ( true ) {
        v = queue_.dequeue();
        std::cout << "Processing :" << v << std::endl;
    }
}
```

Mover todo el código de sincronización a la cola, así se puede usar sin saber nada de variables condicionales ...

```
int producer(void *ptr) {
    int v = 0;
    while ( true ) {
        std::cout << "Producing:" << v << std::endl;
        queue_.enqueue(v);
    ...
}
```

dequeue y enqueue no se ejecutan a la vez (usando mutex) y dequeue bloquea si no hay tareas ...



Blocking queue ...

```
template<typename T>
class BlockingQueue {
queue<T> queue_;
SDL_cond *cond;
SDL_mutex *mutex;

public:
    BlockingQueue() :
        queue_()
    {
        mutex = SDL_CreateMutex();
        cond = SDL_CreateCond();
    }
    virtual ~BlockingQueue() {
        SDL_DestroyMutex(mutex);
        SDL_DestroyCond(cond);
    }
    T dequeue() { ... }
    void enqueue(T& e) { ... }
};
```

dentro se usa una cola normal

Mutex y variable condicional
para sincronizar renqueue y
duque

Blocking queue ...

```
T dequeue() {  
    SDL_LockMutex(mutex);  
    while ( queue_.size() == 0 ) {  
        SDL_CondWait(cond, mutex);  
    }  
    T v = queue_.front();  
    queue_.pop();  
    SDL_UnlockMutex(mutex);  
    return v;  
}
```

```
void enqueue(T& e) {  
    SDL_LockMutex(mutex);  
    if ( queue_.size() == 0 ) {  
        SDL_CondSignal(cond);  
    }  
    queue_.push(e);  
    SDL_UnlockMutex(mutex);  
}
```

Workers ...

```
class Worker {  
public:  
    Worker();  
    virtual ~Worker();  
    void shutDown();  
    void submit(std::function<void()> f);  
private:  
    static int start(void* o);  
    void process();  
  
    BlockingQueue<std::function<void()>> tasks_;  
    SDL_Thread *t_;  
    bool stopped_;  
};
```

parar el worker

Añadir una tarea

Cola de tareas – lambda
expressions ...

El hilo que ejecuta las
tareas ...

Workers ...

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

```
    Worker w1;  
    Worker w2;
```

tarea – lambda expression ...

```
    for (int i = 0; i < 50; i++)  
        w1.submit([i](){std::cout << "Hola " << i << std::endl;});
```

```
// ....
```

```
    for (int i = 0; i < 50; i++)  
        w2.submit([i](){std::cout << "Hello " << i << std::endl;});
```

```
w1.shutdown();  
w2.shutdown();
```

```
    std::cout << "Done!" << std::endl;
```

```
}
```

Workers ...

```
void Worker::Worker() {  
    stopped_ = false;  
    t_ = SDL_CreateThread(start, "Worker", this);  
}  
  
int Worker::start(void* o){  
    static_cast<Worker*>(o)->process();  
    return 0;  
}
```

Salvo autorización expresa de los materiales entregados a estudiantes por cualquier medio durante la carrera, solo se podrán utilizar para el estudio o la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución posterior (incluida la divulgación en redes sociales o servicios de compartición en Internet) puede vulnerar la normativa de protección de datos y/o la propiedad intelectual y material en su caso, así como la legislación sobre infracción y delito de robo de ideas. Si encuentra este material en otro sitio web, o reportcontent@ucm.es. Si no tiene la autorización de la persona infractora, o avisanos a denunciacontenido@ucm.es

Workers ...

```
void Worker::process() {  
    while (!stopped_ || tasks_.size() > 0) {  
        auto &t = tasks_.dequeue();  
        t();  
    }  
}  
  
void Worker::shutDown() {  
    stopped_ = true;  
    submit([]() {}); // empty task to wake up ...  
    SDL_WaitThread(t_, NULL);  
}  
  
void Worker::submit(std::function<void()> f) {  
    tasks_.enqueue(f);  
}
```