Design Patterns in C++

Concurrency

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

April 29, 2011

Threads

- a concurrent program consists of many "flows" of executing code
- each "flow" is called **thread**
  - threads can execute in parallel (if enough processors are available) or alternate on processors depending on a *scheduling algorithm*
- a **process** is a set of threads and a (private) memory address space that contains all variables, the stacks, etc. (i.e. the program state)
  - threads belonging to the same process share the same memory
  - threads belonging to different processes can only communicate with each other through IPC (inter-process communication mechanisms, like *pipes*, *sockets*, etc,)

Mutual Exclusion Problem

- We do not know in advance the relative speed of the threads
  - hence, we do not know the order of execution of the hardware instructions
- Example: incrementing variable x
  - incrementing x is not an *atomic operation*
  - atomic behaviour can be obtained using interrupt disabling or special atomic instructions

Example 1

```
/* Shared memory */
int x;
```

```
void *threadA(void *)
{
  ...;
  x = x + 1;
  ...;
}
```

```
void *threadB(void *)
{
  ...;
  x = x + 1;
  ...;
}
```

- Bad Interleaving:

```
...
LD    R0, x     (TA)   x = 0
LD    R0, x     (TB)   x = 0
INC   R0        (TB)   x = 0
ST    x, R0     (TB)   x = 1
INC   R0        (TA)   x = 1
ST    x, R0     (TA)   x = 1
...
```

Example 2

```
// Shared object (sw resource)
class A {
    int a;
    int b;
public:
    A() : a(1), b(1) {};
    void inc() {
      a = a + 1; b = b +1;
    }
    void mult() {
      b = b * 2; a = a * 2;
    }
} obj;
```

```
void * threadA(void *)
{
     ...
     obj.inc();
     ...
}
```

```
void * threadB(void *)
{
    ...
    obj.mult();
    ...
}
```

Consistency:
After each operation, a == b

Resource in a non-consistent state!!

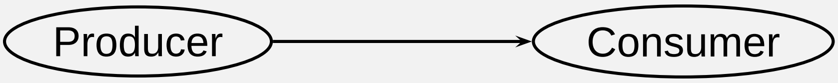
```
a = a + 1;    TA   a = 2
b = b * 2;    TB   b = 2
b = b + 1;    TA   b = 3
a = a * 2;    TB   a = 4
```


Consistency

- for any resource, we can state a set of consistency properties
  - a consistency property C_i is a boolean expression on the values of the internal variables
  - a consistency property must hold before and after each operation
  - it does not need to hold during an operation
  - if the operations are properly sequentialized, the consistency properties will always hold
- formal verification
  - let R be a resource, and let $C(R)$ be a set of consistency properties on the resource
  - $C(R) = \{C_i\}$
  - A concurrent program is correct if, for every possible interleaving of the operations on the resource, $\forall C_i \in C(R)$, C_i holds.

Producer / Consumer model

- mutual exclusion is not the only problem
  - we need a way of synchronise two or more threads
- example: producer/consumer
  - suppose we have two threads,
  - one produces some integers and sends them to another thread (PRODUCER)
  - another one takes the integer and elaborates it (CONSUMER)



Implementation with the circular array

- Suppose that the two threads have different speeds
  - for example, the producer is much faster than the consumer
  - we need to store the temporary results of the producer in some memory buffer
  - for our example, we will use the circular array structure

Producer/Consumer implementation

```
struct CA qu;
```

```
void *producer(void *)
{
  bool res;
  int data;
  while(1) {
    <obtain data>
    while (!insert(&qu, data));
  }
}
```

```
void *consumer(void *)
{
  bool res;
  int data;
  while(1) {
    while (!extract(&qu, &data));
    <use data>
  }
}
```

- Problem with this approach:
  - if the queue is full, the producer waits *actively*
  - if the queue is empty, the consumer waits *actively*

A more general approach

- we need to provide a general mechanism for synchonisation and mutual exclusion
- requirements
  - provide mutual exclusion between critical sections
    - avoid two interleaved insert operations
    - (semaphores, mutexes)
  - synchronise two threads on one condition
    - for example, block the producer when the queue is full
    - (semaphores, condition variables)

The POSIX standard

- is an IEEE standard that specifies an operating system interface
- the standard extends the C language with primitives that allow the implementation of concurrent programs
- POSIX distinguishes between the terms process and thread
  - a process is an address space with one or more threads executing in that address space
  - a thread is a single flow of control within a process
  - every process has at least one thread, the “main()” thread; its termination ends the process
  - all the threads share the same address space, and have a separate stack

The Linux pthread library

- the pthread primitives are usually implemented into a pthread library
- all the declarations of the primitives cited in these slides can be found into `sched.h`, `pthread.h` and `semaphore.h`
- use man to get online documentation
- when compiling under gcc & GNU/Linux, remember the `-lpthread` option

Thread creation

- a thread is identified by a C function, also called *body*:

```
void *my_thread(void *arg)
{
    ....
}
```

- a thread starts with the first instruction of its body
- the threads ends when the body function returns

- a thread can be created using the following primitive

```
int pthread_create( pthread_t *ID,
                    pthread_attr_t *attr,
                    void *(*body)(void *),
                    void * arg);
```

- `pthread_t` is the type that represents the thread ID
- `pthread_attr_t` is the type that represents the parameters of the thread
- `arg` is the argument passed to the thread body when it starts

Thread attributes

- thread attributes specify the characteristics of a thread
  - detach state (joinable or detached)
  - stack size and address
  - scheduling parameters (priority, ...)
- attributes must be initialized and destroyed

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- a thread can terminate itself by calling

```
void pthread_exit(void *retval);
```

- when the thread body ends after the last "}", `pthread_exit()` is called implicitly
- exception: when `main()` terminates, `exit()` is called implicitly, which terminates the whole process! (and all threads in it)

Thread joining

- each thread has a unique ID
- the thread ID of the current thread can be obtained using

```
pthread_t pthread_self(void);
```

- two thread IDs can be compared using

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

- a thread can wait the termination of another thread using

```
int pthread_join(pthread_t th,void **thread_return);
```

- it gets the return value of the thread or `PTHREAD_CANCELED` if the thread has been killed
- by default, every task must be joined
- the join frees all the internal resources (stack, registers, and so on)

Detaching

- a thread which does not need to be joined must be declared as detached.
- 2 ways:
  - the thread is created as detached using

    ```
    pthread_attr_setdetachstate(...);
    ```

  - the thread becomes detached by calling `pthread_detach()` from its body
- joining a detached thread returns an error

Killing a thread

- a thread can be killed by calling

```
int pthread_cancel(pthread_t thread);
```

- when a thread dies its data structures will be released
  - by the join primitive if the thread is joinable
  - immediately if the thread is detached
- there are two different behaviours:
  - **deferred cancellation**: when a kill request arrives to a thread, the thread does not die. The thread will die only when it will execute a primitive that is a cancellation point. This is the default behaviour of a thread.
  - **asynchronous cancellation**: when a kill request arrives to a thread, the thread dies. The programmer must ensure that all the application data structures are coherent.

Cancellation state

- the user can set the cancellation state of a thread using:

```
int pthread_setcancelstate(int state,int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

- the user can protect some regions providing destructors to be executed in case of cancellation

```
int pthread_cleanup_push(void (*routine)(void *), void *arg);
int pthread_cleanup_pop(int execute);
```

Cancellation points

- the cancellation points are primitives that can potentially block a thread; when called, if there is a kill request pending the thread will die
  - `void pthread_testcancel(void);`
  - `sem_wait`, `pthread_cond_wait`, `printf` and all the I/O primitives
  - `pthread_mutex_lock`, is NOT a cancellation point
- a complete list can be found into the POSIX Std

Cleanup handlers

- the user must guarantee that when a thread is killed, the application data remain coherent
  - the user can protect the application code by using cleanup handlers
  - a cleanup handler is an user function that cleans up the application data they are called when the thread ends and when it is killed

```
void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

- they are pushed and popped as in a stack (in LIFO order)
- if `execute` $!= 0$ the cleanup handler is called when popped

Semaphores

- a semaphore is a counter managed with a set of primitives
- it is used for
  - synchronization
  - mutual exclusion
- POSIX Semaphores can be
  - unnamed (local to a process)
  - named (shared between processed through a file descriptor)
- the `sem_t` type contains all the semaphore data structures
- initialization

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `pshared` is 0 if sem is not shared between processes
- destroying the semaphore

```
int sem_destroy(sem_t *sem)
```


Wait and post

- Wait operation:

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

  - if the counter is greater than 0, the thread decrements the counter and continues, otherwise it blocks
  - `sem_trywait` never blocks, but returns error
  - `sem_wait` is a cancellation point

```
int sem_post(sem_t *sem);
```

- if a thread is blocked, unblocks it, otherwise it increments the counter

```
int sem_getvalue(sem_t *sem,int *val);
```

- it simply returns the semaphore counter

Mutex description

- a mutex can be considered as a binary semaphore used for mutual exclusion
  - with the restriction that a mutex can be unlocked only by the thread that locked it
- mutexes also support some RT protocols
  - priority inheritance
  - priority ceiling
- mutex initialization and destruction

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


Attributes

- You must first create (and later destroy) a `mutex_attr` data structure

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- To set a protocol:

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int prot);
```

- where prot can be protocol can be `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT`, `PTHREAD_PRIO_PROTECT`
- in the last case, you need to set the ceiling:

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int c);
```

Lock and unlock

- To lock, lock without blocking and unlock:

```
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_trylock(pthread_mutex_t *m);
int pthread_mutex_unlock(pthread_mutex_t *m);
```


Condition variables

- condition variables are used to enforce synchronization between threads
  - a thread into a mutex critical section can wait on a condition variable
  - when waiting, the mutex is automatically released and locked again at wake up
  - the synchronization point must be checked into a loop!
- A condition variable has type `pthread_cond_t`, and must be initialized before its use:

```
int pthread_cond_init(pthread_cond_t *c, pthread_cond_attr *a);
```

- and destroyed when it is not used anymore

```
int pthread_cond_destroy(pthread_cond_t *c);
```

Waiting for a condition

- When we want to block a thread on a condition variable we call:

```
int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

- Every condition variable is always linked to a mutex
  - releases the mutex
  - blocks the thread on the condition variable queue
  - acquires the mutex
- Note con cancellations:
  - `pthread_mutex_lock()` is not a cancellation point, while `pthread_cond_wait()` is.
  - when a thread is killed while blocked on a condition variable, the mutex is **locked again** before dying
  - therefore, an appropriate cleanup function must be used to protect the thread from the cancellation

Signaling a condition

- To wake up a blocked thread on a condition:

```
int pthread_cond_signal(pthread_cond_t *c);
```

- to wake up all thread blocked on a condition:

```
int pthread_cond_broadcast(pthread_cond_t *c);
```

- if no thread is blocked, these functions have no effect whatsoever

Advantages of OO

- We have seen POSIX, one of many possible interfaces
  - Microsoft Windows has a completely different interface
  - In RTOS for embedded systems, the situation is actually worse as there are many different API, one for each kind of OS
- Object Oriented programming brings many advantages wrt C language
  - Achieve a higher degree of re-usability, separation of concerns, less dependencies, etc.
  - with less and cleaner code
- For example, it is possible to extend and re-use implementation by using inheritance and polymorphism
- Also, the compiler performs many additional checks
  - avoids overuse of `#define` and other pre-processor directives
  - reduces the amount of `void *` pointers
  - code is less error-prone

Independence from the platform

- One important use of the Object Oriented approach is to reduce the amount of dependencies from the underlying Operating System
  - Many different operating systems use different APIs to provide services
  - for example mutex (`pthread_mutex_t` in Posix, `CRITICALSECTION` in Windows, etc.)
  - they also have different parameters
  - However, the provided functionalities are quite similar
- We can abstract the underlying API with a unique interface
  - Our code will depend only in the common abstract APIs
  - We can select the platform API at compile time with a simple switch
- of course this can be done also in C
  - However, we would need many `#define` in the code

Boost threads

- We will study one such particular OO library that wraps threads, locks and concurrency controls in one library
  - The library is portable across many different OS
  - It is a candidate to be included in the next C++0x standard

Scoped locking

- the goal is to simplify the code for locking and unlocking mutex inside functions
  - Usually the lock is acquired at the beginning of the function and released at the end
  - however, the function may have many different return points
  - also, exceptions may be raised by other functions
- therefore, it is quite easy to forget to release the mutex

Example

- the following code contains two stupid errors

```
void myfun() {
   lock.acquire();
   ...
   if (cond1) return;

   g(); // may throw and exc.

   lock.release();
}
```

error 1: the lock is not released

error 2: an exception may be thrown, and the lock will not be released

Solution

- Use the RAII techniques (Resource Acquisition Is Initialisation)
  - The lock is wrapped inside another object called *Guard*
  - the only purpose of Guard is to guarantee that the lock is released when Guard goes out of scope
  - to do this, Guard acquires the lock in its constructor, and releases it in the destructor

```
class Guard {
    Lock &lock;
public:
    Guard(Lock &l) : lock(l) {
        lock.acquire();
    }
    ~Guard() {
        lock.release();
    }
};
```

Example, correct

```
void myfun() {
   Guard g(lock);
   ...
   if (cond1) return;

   g(); // may throw and exc.
}
```

The Guard is destructed automatically

Even when an exception is thrown

Some little problems

- Of course, the user should access the mutex only through the guard
  - in particular, she should not release the lock accessing it directly
  - if releasing the lock in the middle of the function is necessary, it may be the case to add methods `acquire` and `release` also in the `Guard` class

```
class Guard {
    Lock &lock;
    bool owner;
public:
    Guard(Lock &l) : lock(l), owner(false) {
        acquire();
    }
    void acquire() {
        if (!owner) { lock.acquire(); owner = true; }
    }
    void release() {
        if (owner) { lock.release(); owner = false; }
    }
    ~Guard() { release(); }
};
```

Deadlock

- This pattern can cause a deadlock is a function recursively calls itself
  - This can be solved putting a check into the Lock class
  - before acquiring the lock, the function check is the lock is already owned by the same thread
  - another solution is to divide interface methods (that acquire the lock) and implementation methods (which do not acquire the lock)
  - interface methods are public and can only be called from outside
  - implementation methods are private or protected, and can only be called by implementation methods
- Mutex objects should be declared *mutable* in C++, to allow const methods to acquire the lock

Configuring the lock strategy

- It may be useful to configure a class to use one of many different lock mechanisms
  - No locking at all, if the class is used by one single thread
  - a simple mutex
  - a recursive mutex to avoid self-deadlock
  - a reader-writer lock
- in any case, we would like to write the class code once and configure with different locks
- we can then apply the strategy pattern
  - Locking is a strategy that is delegated to another class

Using polymorphism

- In this case, we assume that all Lock classes belong to a hierarchy and that methods `acquire()` and `release()` are virtual methods

```
class MyClass {
    mutable Lock *lock;
public:
    MyClass(Lock *l) : lock(l) {...}

    void func() {
        Guard g(*lock);
        ...
    }
};
```


Using templates

- In this case, the type of lock is a template parameter
- of course, we need the Guard to be a template with the lock type as template parameter

```
template <class LOCK>
class MyClass {
    mutable LOCK lock;
public:
    MyClass () : lock() {}

    void func() {
        Guard<LOCK> g(lock);
        ...
    }
};
```

The Null mutex

- Here is an example of Null Mutex
- this can be used when we want to use the class for one thread only

```
class NullMutex {
public:
    NullMutex() {}
    ~NullMutex() {}
    void acquire() {}
    void release() {}
};
```


Polymorphism or template?

- We use polymorphism when we want to be flexible at run-time
- we use templates when we want to be flexible just at compile time
- therefore, polymorphism is more flexible, but errors can only be checked at run-time
- on the other end, templates are “safer” because the compiler checks everything at compile time, however, they are less flexible
- for example, when different objects of the same class need to have different locking strategies, polymorphism is more adequate (all objects will have the same type)