SECTION 06

Mutual Exclusion

© Feabhas Ltd 2012

Producer-Consumer threads

```
class Producer
{
public:
  Producer(myStack& s): sum(0), stack(s){}

protected:
  void operator()();

private:
  int sum;
  SimpleStack& stack;
};


void Producer::operator()()
{
  for(int i = 0; i < 1000; ++i)
  {
    stack.push(i);
    sum += i;
  }
  cout << "Produced: " << sum << endl;
}
```

```
class Consumer
{
public:
  Consumer(myStack& s): sum(0), stack(s){}

protected:
  void operator()();

private:
  int sum;
  SimpleStack& stack;
};


void Consumer::operator()()
{
  for(int i = 0; i < 1000; ++i)
  {
    int val = stack.pop();
    sum += val;
  }
  cout << "Consumed: " << sum << endl;
}
```

In this example we have two thread classes - a `Producer`, which creates data and inserts it onto a stack; and a `Consumer`, that retrieves data from a stack.

This pattern is very typical in embedded systems; particularly where the Producer and Consumer runs at different rates.

© Feabhas Ltd 2012

Thread-unsafe Stack class

```
class SimpleStack
{
public:
  SimpleStack();
  bool push(int val);
  int pop();

private:
  static const uint32_t size = 1000;
  int stack[size];
  uint32_t count;
};
```

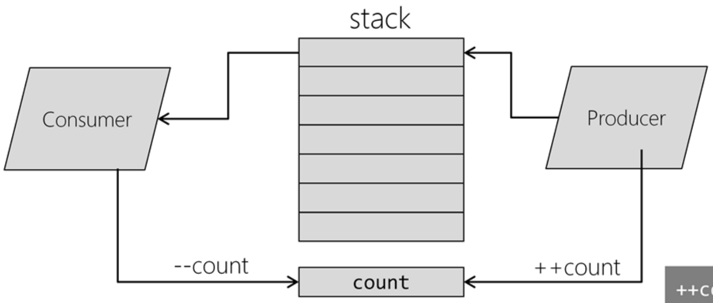
```
SimpleStack:: SimpleStack(): count(0)
{
    memset(stack, 0, sizeof(stack));
}


bool SimpleStack::push(int val)
{
  if (count < size)
  {
    stack[count++] = val;
    return true;
  }
  return false;
}


int SimpleStack::pop()
{
  if (count != 0)
  {
    int val = stack[--count];
    return val;
  }
  return -1;
}
```

Are there issues with using this class in a multi-threaded environment?

Above is a basic stack implementation. The stack is a simple array. The `count` member is used to ensure data isn't inserted onto a full stack; or read from an empty stack.

© Feabhas Ltd 2012

Problems with shared resources




```
++count
LDR   R1,[count]
ADD   R1,R1,#0x01
STR   R1,[count]

--count
LDR   R1,[count]
SUB   R1,R1,#0x01
STR   R1,[count]
```

The problem arise with two threads both trying to manipulate a commonly-shared resource.

In this example the `Producer` and `Consumer` could both attempt to adjust the count value at the same time. The OS schedules operations at the opcode level, so a context switch could occur at any point during the read-modify-write cycle.

© Feabhas Ltd 2012

std::mutex class declaration

```
class mutex
{
public:
  mutex(mutex const&) = delete;
  mutex& operator=(mutex const&) = delete;
  mutex();
  ~mutex();

  void lock();
  void unlock();
  bool try_lock();
};
```

The `std::mutex` class provides a basic mutual exclusion and synchronization facility for threads which can be used to protect shared data.

`lock()` is a blocking call which will suspend the calling thread if the mutex is unavailable (locked by another thread). When the mutex is released (with `unlock()`) any waiting thread will be scheduled.

In cases where you don't wish to block you can call `try_lock()` which will return `true` is the lock has been acquired; otherwise `false`.

Note the mutex cannot be copied.

© Feabhas Ltd 2012

Protecting the SimpleStack class

```
#include <mutex>

class SimpleStack
{
public:
  SimpleStack();
  bool push(int val);
  int pop();

private:
  static const uint32_t size = 1000;
  int stack[size];
  uint32_t count;
  std::mutex mtx;
};
```

```
bool SimpleStack::push(int val)
{
  bool retval = false;
  mtx.lock();              // LOCK
  if (count < size)
  {
    stack[count++] = val;
    retVal = true;
  }
  mtx.unlock();            // UNLOCK
  return retVal;
}

int SimpleStack::pop()
{
  int val = -1;
  mtx.lock();              // LOCK
  if (count != 0)
  {
    val = stack[--count];
  }
  mtx.unlock();            // UNLOCK
  return val;
}
```

The mutex is locked and unlocked as part of the `push()` and `pop()` functions.

© Feabhas Ltd 2012

C++11 mutual exclusion classes

class mutex;

class recursive_mutex;

class timed_mutex;

class recursive_timed_mutex;

The `std::mutex` class provides a basic mutual exclusion and synchronization facility for threads which can be used to protect shared data.

`std::recursive_mutex` is *recursive* so a thread that holds a lock on a particular instance may make further calls `lock()` or `try_lock()` to increase the lock count.

The `std::timed_mutex` class provides support for locks with timeouts on top of the basic mutual exclusion and synchronization facility provided by `std::mutex`. If a lock is already held by another thread then an attempt to acquire the lock will block until the lock can be acquired or the lock attempt times out (`try_lock_for()` or `try_lock_until()`).

© Feabhas Ltd 2012

Danger of deadlock

```
int SimpleStack::pop()
{
  mtx.lock();

  if (count != 0)
  {
    int val = stack_[--count];
    mtx.unlock();
    return val;
  }
  return -1;
}
```

One weakness is that locks and unlocks *must* be paired If an unlock is not called (e.g. exceptions, missed return path) the code will potentially deadlock.

In the example, if `count == 0` then the mutex is not unlocked!

© Feabhas Ltd 2012

std::lock_guard

```
template <class Mutex>
class lock_guard
{
public:
  typedef Mutex mutex_type;

  lock_guard();
  explicit lock_guard(mutex_type& m);
  lock_guard(mutex_type& m, adopt_lock_t);

  lock_guard(lock_guard const& ) = delete;
  lock_guard& operator=(lock_guard const& ) = delete;
};
```

In the previous example there was the potential of leaving a mutex locked accidentally. This risk can be substantially reduced by making use of the RAII / RDID model (see the section on Resource Management for more information). This technique is sometimes referred to as the Scope-Locked Idiom (See Pattern Oriented Software Architecture Volume 2, p.325 for more)

A `std::lock_guard` object locks on construction and unlocks on destruction.

© Feabhas Ltd 2012

Using a std::lock_guard

```
int SimpleStack::pop()
{
  std::lock_guard<std::mutex> guard(mtx);

  if (count != 0)
  {
    return stack[--count];
  }
  return -1;
} // UNLOCK
```

```
bool SimpleStack::push(int val)
{
  std::lock_guard<std::mutex> guard(mtx);
  if (count < sz)
  {
    stack[count++] = val;
    return true;
  }
  return false;
} // UNLOCK
```

Mutex is guaranteed to be unlocked on exit

By scoping the guard object the mutex is guaranteed to be unlocked.

It is bad practice to hold a mutex for too long. You should keep the 'locked' code as small as possible. However, the structure of your code could mean there is a lot of code between the lock and the end of the scope (function).

One solution is to put the guard in its own scope to limit its lifetime.

© Feabhas Ltd 2012

Key Points

Resources shared between two or more threads should be protected against corruption due to thread race conditions

A std::mutex class is used to abstract away from OS-specific mutual exclusion mechanisms

A std::lock_guard object can be used to ensure that Mutexes are always unlocked safely. This is known as the *scope-locked idiom*

© Feabhas Ltd 2012