

**Software Engineering**  
**CSE 335, Spring 2006**

**OO design: Concurrency and Distribution**

Stephen Wagner

Michigan State University

# Definitions

- Process: Program in execution
  - Created and managed by the operating system
  - Multiple processes execute in separate address spaces
- Thread: “Lightweight” process
  - Created and run within a process (depends on OS)
  - Multiple threads execute in shared address space
- Distributed applications with graphical user interfaces tend to be multi-threaded

# Distributed application

- **Defn:** Application comprising multiple processes, usually running on different machines over a network
- Client–server systems:
  - Server is a process
  - accessed by many client processes
- Examples
  - CSE HTTP server, accessed by any number of browser clients running anywhere in the world
  - X server

# OO and distributed applications

- OO concepts contribute to concurrent/distributed applications development
- Concurrency/synchronization model:
  - Threads enable the design of time-multiplexed *active objects*
  - *Monitors* protect concurrent access to passive objects
- Separation of interface and implementation classes:
  - Basis for “pluggable” component technologies
  - JavaBeans, EJB, ODBC
- Design patterns
  - *Proxy pattern* simplifies inter-process communication
  - Basis for distributed object middleware

# Multi-threaded programming

- C++ provides no language features for thread programming
  - Threads are supported directly in Java and C#
  - In C++, threads and thread operations are provided by standard libraries
- In Unix, standard threads library is “pthreads”
  - Short for POSIX threads
  - Include files: `/usr/include/pthread.h`
  - Link library: `libpthread.a`

## Active Object: new definition

- **Defn:** An object that runs in its own thread
- Multi-threaded applications comprise two or more active objects that manipulate one or more passive objects in shared memory
- In most object-oriented languages, you design active objects and then run them from within separate thread objects

## Design Pattern for Active Objects

```
class ActiveObjectInterface
{
    public:
        virtual bool activate() = 0;
        virtual void terminate() = 0;
};

class Thread
{
    public:
        int start(ActiveObjectInterface *);
    protected:
        ActiveObjectInterface*    activeObject;
        pthread_t                  myThread;
        static void* startMyThread(void *);
};
```

## Design pattern, active objects

```
int Thread::start(ActiveObjectInterface *ao)
{
    activeObject=ao;
    return (pthread_create(&myThread,
                          0,
                          startMyThread,
                          (void *) ao));
}
```

```
void *Thread::startMyThread(void *vptr)
{
    ActiveObjectInterface* ao=static_cast<ActiveObjectInterface *>
    while (ao->activate());
    return 0;
}
```

## Exercise

Write a short program with two threads, one of which continually emits “HELLO!”, and the other of which continually emits “GOODBYE!”.

## Answer

```
class Emitter : public ActiveObjectInterface {
    public:
        Emitter(const string & s) : word(s) {}
        bool activate() { cout << word << endl; return true; }
    protected:
        string word;
};

int main(void)
{
    Thread t;
    Emitter hello("Hello"), goodbye("Goodbye");
    t.start(&hello);
    while (goodbye.activate());
    return 0; }
```

## Question

Always a “main” thread that you need not create explicitly

**Question:** What would happen if we removed the statement `goodbye.activate()`?

## Concurrent access to shared data

- **Problem:** Multiple active objects might access the same passive object “at the same time”
  - Generally OK if the active objects are only reading data from the passive object(s)
  - But if one more active objects is modifying the data members of the shared object, then we get anomalies
- Example: two active objects trying to pull an element off of a shared queue
- To prevent these data-access anomalies requires *synchronizing* the active objects

## Example

```
class SharedQueue {
    public:
        ...
        bool pull (string &s)
        {   bool retval = q.empty();
            if (!retval) {
                s = q.back();
                q.pop();
            }
            return retval;
        }
        ...
    protected:
        queue<string> q;
};
```



# Thread synchronization

- **Definitions:**
  - A *critical section* is a region of code in which at most one thread should be allowed to execute concurrently
  - A *mutex lock* is a facility that is used to synchronize threads
    - \* One and only one thread can *own* a lock
    - \* Thread gets to own a lock by *acquiring* it
    - \* A thread will block if it attempts to acquire a lock owned by another thread
- **Design tip:** Whenever you write multi-threaded programs, you must *identify* and *protect* critical sections in your code.

## Use of mutex locks

- `pthread_mutex_t`: type use for declare a lock
- `pthread_mutex_init`: initializes a lock
- `pthread_mutex_destroy`: destroys a lock
- `pthread_mutex_lock`: acquires a lock, blocking if lock owned by another thread
- `pthread_mutex_unlock`: releases a lock, so that other threads may acquire it

```
class SharedQueue {
public:
    SharedQueue()
    { pthread_mutex_init(&lock); }
    ~SharedQueue()
    { pthread_mutex_destroy(&lock); }
    ...
    bool pull (string &s)
    { pthread_mutex_lock(&lock);
      bool retval = q.empty();
      if (!retval) {
          s = q.back();
          q.pop();
      }
      pthread_mutex_unlock(&lock);
      return retval; }
    ...
protected:
    queue<string> q;
    pthread_mutex_t lock; };
```

# Monitor synchronization

- **Defn:** A *Monitor* is an object whose methods may not be executed by multiple threads concurrently
- Example:
  - Let  $o$  be a monitor that provides the operation `void foo`
  - Suppose threads  $T_1$  and  $T_2$  invoke `o.foo()`
  - One thread (e.g.  $T_1$ ) will execute `foo` in its entirety while the other thread waits
- Most OO languages use monitor synchronization

## The monitor-object pattern

- Standard “pattern” for promoting an arbitrary C++ class into a monitor class
- Let **C** be the original class, and **M** be the (new) monitor class
  - **M** should inherit publicly from **C**
  - **M** should contain a protected data member (call it `lock`) of type `pthread_mutex_t`
  - For each public method **m** of **C**, **M** should override that method with one that acquires `lock`, invokes **C::m** and then releases `lock`

# The Proxy pattern

