An Overview of Multi-threading Support in ACE

Douglas C. Schmidt

Washington University, St. Louis

http://www.cs.wustl.edu/∼schmidt/

schmidt@cs.wustl.edu

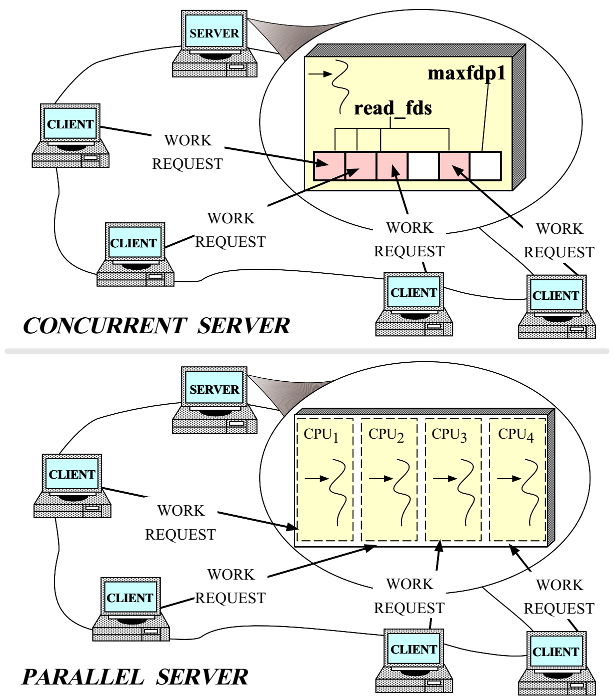
Motivation for Concurrency

- Concurrent programming is increasing relevant to:
  - *Leverage hardware/software advances*
    - *e.g.*, multi-processors and OS thread support
  - *Increase performance*
    - *e.g.*, overlap computation and communication
  - *Improve response-time*
    - *e.g.*, GUIs and network servers
  - *Simplify program structure*
    - *e.g.*, synchronous vs. asynchronous network IPC

Definitions

- *Concurrency*
  - "Logically" simultaneous processing
  - Does *not* imply multiple processing elements
- *Parallelism*
  - "Physically" simultaneous processing
  - Involves multiple processing elements and/or independent device operations
- Both *concurrency* and *parallelism* require controlled access to shared resources
  - *e.g.*, I/O devices, files, database records, in-core data structures, consoles, etc.

Concurrency vs. Parallelism



Concurrency Overview

- A thread of control is a single sequence of execution steps performed in one or more programs
  - *One program* $\rightarrow$ standalone systems
  - *More than one program* $\rightarrow$ distributed systems
- Traditional OS processes contain a single thread of control
  - This simplifies programming since a sequence of execution steps is protected from unwanted interference by other execution sequences...

Traditional Approaches to OS Concurrency

1. Device drivers and programs with signal handlers utilize a limited form of *concurrency*
   - *e.g.*, asynchronous I/O
   - Note that *concurrency* encompasses more than *multi-threading*...
2. Many existing programs utilize OS processes to provide "coarse-grained" concurrency
   - *e.g.*,
     - Client/server database applications
     - Standard network daemons like UNIX `inetd`
   - Multiple OS processes may share memory via memory mapping or shared memory and use semaphores to coordinate execution
   - The OS kernel scheduler dictates process behavior

Evaluating Traditional OS Process-based Concurrency

- Advantages
  - *Easy to keep processes from interfering*
    - A process combines *security*, *protection*, and *robustness*
- Disadvantages
  1. *Complicated to program*, *e.g.*,
     - Signal handling may be tricky
     - Shared memory may be inconvenient
  2. *Inefficient*
     - The OS kernel is involved in synchronization and process management
     - Difficult to exert fine-grained control over scheduling and priorities

Modern OS Concurrency

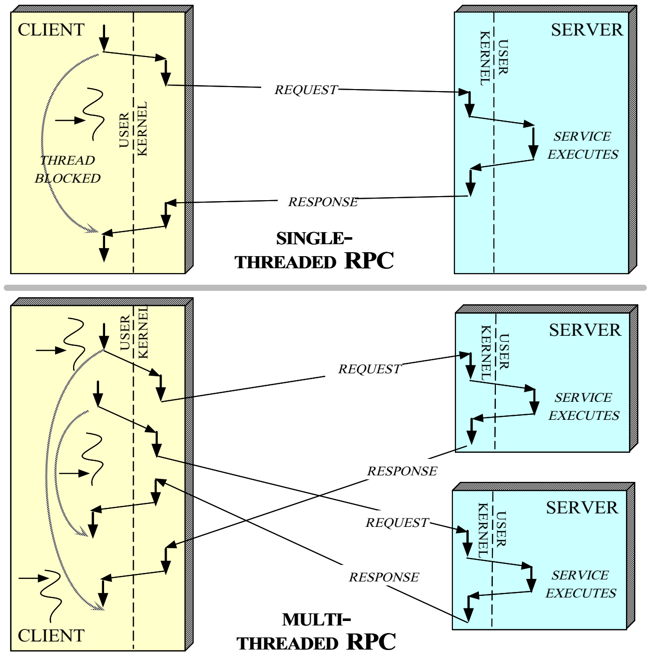
- Modern OS platforms typically provide a standard set of APIs that handle
  1. Process/thread creation and destruction
  2. Various types of process/thread synchronization and mutual exclusion
  3. Asynchronous facilities for interrupting long-running processes/threads to report errors and control program behavior
- Once the underlying concepts are mastered, it's relatively easy to learn different concurrency APIs
  - *e.g.*, traditional UNIX process operations, Solaris threads, POSIX pthreads, WIN32 threads, Java threads, etc.

Lightweight Concurrency

- Modern OSs provide lightweight mechanisms that manage and synchronize multiple threads *within* a process
  - Some systems also allow threads to synchronize *across* multiple processes

- Benefits of threads
  1. *Relatively simple and efficient to create, control, synchronize, and collaborate*
     - Threads share many process resources by default
  2. *Improve performance by overlapping computation and communication*
     - Threads may also consume less resources than processes
  3. *Improve program structure*
     - *e.g.*, compared with using asynchronous I/O

Single-threaded vs. Multi-threaded RPC



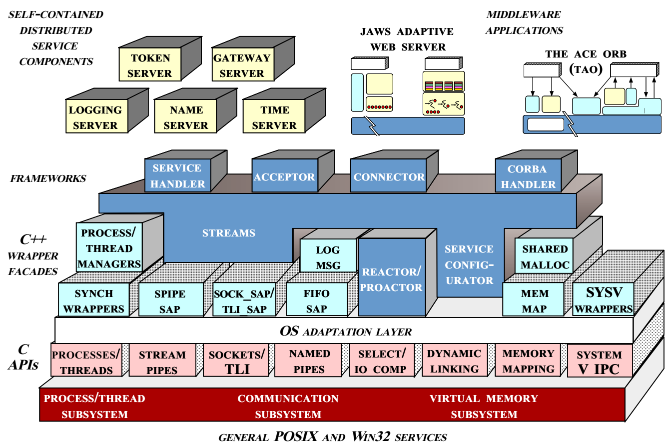
Application Threads

- Most process resources are equally accessible to all threads in a process, *e.g.*,
  - * *Virtual memory*
  - * *User permissions and access control privileges*
  - * *Open files*
  - * *Signal handlers*

- Each thread also contains unique information, *e.g.*,
  - * *Identifier*
  - * *Register set* (*e.g.*, PC and SP)
  - * *Run-time stack*
  - * *Signal mask*
  - * *Priority*
  - * *Thread-specific data* (*e.g.*, `errno`)

- Note, there is generally no MMU protection for separate threads within a single process...

The ADAPTIVE Communication Environment (ACE)

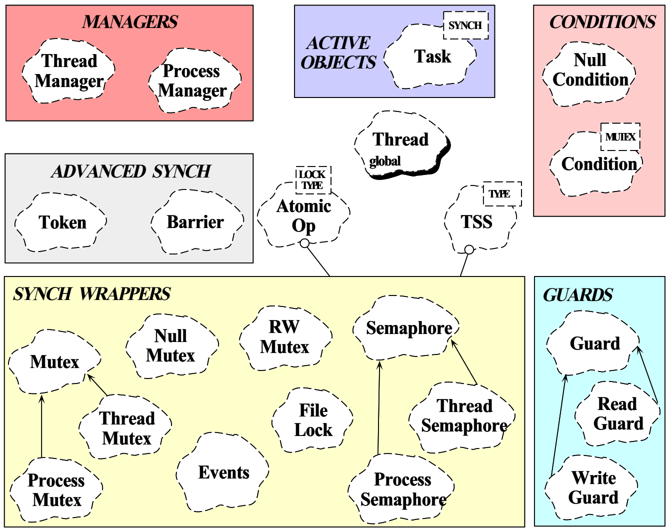


- A set of C++ wrappers, class categories, and frameworks based on patterns
  - www.cs.wustl.edu/~schmidt/ACE.html

ACE Statistics

- Core ACE frameworks and components contain 175,000 lines of C++
  - > 20 person-years of effort
- Ported to UNIX, Win32, MVS, and embedded platforms
- Large user community (ACE-users.html)
- Currently used by dozens of companies
  - *e.g.*, Siemens, Motorola, Ericsson, Kodak, Bellcore, Boeing, SAIC, StorTek ,etc.
- Supported commercially by Riverace
  - www.riverace.com/

Overview of ACE Synchronization and Concurrency Mechanisms



Threading in ACE

- One way to create threads in ACE is with the `Thread_Manager`
  - `Thread_Manager` shields applications from incompabitilities between different OS thread libraries
  - `Thread_Manager` also implements *barrier synchronization* on thread exits
- Another way is to use the ACE `Task`
  - We will cover this later...

Creating a Thread with the Thread Manager

- Thread creation is handled via the `spawn` method:

```
int spawn (ACE_THR_FUNC func,
           void *args = 0,
           long flags = THR_NEW_LWP | THR_JOINABLE,
           ACE_thread_t * = 0,
           ACE_hthread_t *t_handle = 0,
           long priority = ACE_DEFAULT_THREAD_PRIORITY,
           int grp_id = -1,
           void *stack = 0,
           size_t stack_size = 0);
```

  - `spawn` creates and starts a new thread using the `func` method specified in the call
    - Returns 0 on success and non−0 on failure
  - The identify of the thread is returned to the caller
    - A thread id is only valid within a single process
  - The caller can supply a stack or if a NULL is used the library allocates a default stack

Creating Multiple Thread with the Thread Manager

- Multiple threads can be created via the `spawn_n` method:

```
int spawn_n (size_t n,
             ACE_THR_FUNC func,
             void *args = 0,
             long flags = THR_NEW_LWP | THR_JOINABLE,
             long priority = ACE_DEFAULT_THREAD_PRIORITY,
             int grp_id = -1,
             ACE_Task_Base *task = 0,
             ACE_hthread_t thread_handles[] = 0,
             void *stack[] = 0,
             size_t stack_size[] = 0);
```

  – `spawn_n` creates and starts n new threads using the `func` method specified in the call

  – This is useful for creating thread pools

Obtaining Thread Identify with the Thread Manager

- The `thr_self` method returns the thread identifier structure of the caller

```
int thr_self (ACE_hthread_t &);
ACE_thread_t thr_self (void);
```

- The `ACE_hthread_t` and `ACE_thread_t` are the same type and value on most, but not all, OS platforms

  – *e.g.*, on Windows NT they are different...

Exiting a Thread in the Thread Manager

- The `exit` method terminates the invoking thread and sets the exit status to the specified value

```
void *exit (void *status,
            int do_thread_exit = 1);
```

  – If the thread was *not* detached, its identifier and status are retained until `join` or `wait` is called via another thread

  – If there are no remaining threads, the process exits

Joining with a Thread in the Thread Manager

- The `join` method blocks until the specified thread exits

```
int join (ACE_thread_t tid, void **status = 0);
```

  – `tid` is the thread to join with

  – If `status` is non-NULL it points to a location storing the exits status of the terminated thread

  – `join` cannot wait for detached threads, threads in other processes, or the current thread

Waiting on Threads in the Thread Manager

- The `wait` method blocks until there are no more threads running in the `Thread_Manager` or the timeout expires

```
int wait (const ACE_Time_Value *timeout = 0,
          int abandon_detached_threads = 0);
```

  - `timeout` is the timeout period

- The `wait` method is often used to implement a form of "barrier synchronization"

Synchronization Mechanisms

- Threads share resources in a process address space

- Therefore, they must use *synchronization mechanisms* to coordinate their access to shared data

- Traditional OS synchronization mechanisms are very low-level, tedious to program, error-prone, and non-portable

- ACE encapsulates these mechanisms with higher-level patterns and classes

ACE Synchronization Mechanisms

1. *Mutual exclusion* locks
   - Serialize access to a shared resource
2. *Counting semaphores*
   - Synchronize execution
3. *Readers/writer* locks
   - Serialize access to resources whose contents are searched more than changed
4. *Condition variables*
   - Used to block until shared data changes state
5. *File locks*
   - System-wide readers/write locks access by filename

ACE Synchronization Mechanism (cont'd)

1. *Guards*
   - An exception-safe scoped locking mechanism
2. *Barriers*
   - Allows threads to synchronize their completion
3. *Token*
   - Provides absolute scheduling order and simplifies multi-threaded event loop integration
4. *Task*
   - Provides higher-level "active object" semantics for concurrent applications
5. *Thread-specific storage*
   - Low-overhead, contention-free storage

ACE Mutex Overview

- The simplest type of synchronization variable is the "mutex" (mutual exclusion) lock
- Only one thread at a time may "own" a mutex lock
  - *i.e.*, used to implement "critical sections"...
- Implemented to be highly efficient, but limited in methodality
  - *e.g.*, lock/unlock operations must be "fully-bracketed"

The ACE Thread_Mutex API

- Only valid for threads in the same process

```
class ACE_Thread_Mutex
{
public:
  ACE_Thread_Mutex (LPCTSTR name = 0, void *arg = 0);
  ~ACE_Thread_Mutex (void);

  int remove (void);
  // Explicitly destroy the mutex.

  int acquire (void);
  // Acquire lock ownership (wait on queue if necessary).

  int tryacquire (void);
  // Conditionally acquire lock (i.e., don't wait
  // on queue).  Returns -1 on failure.  If we
  // "failed" because someone else already had
  // the lock, <errno> is set to <EBUSY>.

  int release (void);
  // Release lock and unblock a thread at head of queue.
  // ...
```


Programming with ACE Thread_Mutex

- Simple resource example

```
static Thread_Mutex count_mutex; // Initialized to 0
static int count;

int increment_count (void) {
      count_mutex.acquire ();
      count = count + 1; // atomic update
      count_mutex.release ();
}

int get_count (void) {
      int c;
      count_mutex.acquire ();
      c = count; // ensure memory synchronization...
      count_mutex.release ();
      return c;
}
```


ace Semaphore Overview

- Semaphores are conceptually non-negative integers that may be incremented and decremented *atomically*
- They are less efficient than mutexes, but more general
  - *e.g.*, they need not be acquired and released by the same thread
    - *i.e.*, they may be used in signal handlers or other asynchronous event notification contexts
- It is not necessary to acquire a mutex lock to use a semaphore

The ACE Thread_Semaphore API

- Only valid for threads in the same process

```
class ACE_Thread_Semaphore
{
public:
  ACE_Thread_Semaphore (u_int count = 1, LPCTSTR name = 0,
                        void * = 0, int max = 0x7FFFFFFF);
  // Initialize the semaphore, with an initial
  // value of <count>, a maximum value of <max>,
  // and unlocked by default.
  ~ACE_Thread_Semaphore (void); // Default dtor.
  int remove (void);
  // Explicitly destroy the semaphore.
  int acquire (void);
  // Block the thread until the semaphore count becomes
  // greater than 0, then decrement it.
  int acquire (ACE_Time_Value &tv);
  // Block the thread until <tv> times out or until the semaphore
  // count becomes greater than 0 (at which point it is decremented).
  int tryacquire (void);
  // Conditionally decrement the semaphore if count is greater than 0
  // (i.e., won't block).  Returns -1 on failure.  If we "failed"
  // because someone else already had the lock, <errno> is set to
  // <EBUSY>.
  int release (void);
  // Increment the semaphore by 1, potentially unblocking a waiting
  // thread.
  int release (size_t release_count);
  // Increment the semaphore by <release_count>, potentially
  // unblocking waiting threads.
  // ...
```


Programming with ACE Thread_Semaphore

- Simple producer/consumer semaphore example

```
static int rd_ptr = 0, wr_ptr = 0;
static data_t buf[BUFSIZ];
static ace_Thread_Semaphore not_empty (0);
static ace_Thread_Semaphore not_full (1);

/* Producer thread 1 */
while (work_to_do) {
    buf[wr_ptr] = produce ();
    not_full.acquire ();
    wr_ptr = (wr_ptr + 1) % BUFSIZ;
    not_empty.release ();
}

/* Consumer thread 2 */
while (work_to_do) {
    not_empty.acquire ();
    consume (buf[rd_ptr]);
    not_full.release ();
    rd_ptr = (rd_ptr + 1) % BUFSIZ;
}
```


Overview of ACE Condition Variables

- Used to "sleep/wait" until a particular condition involving shared data occurs
  - Conditions may be arbitrarily complex
- Allows more complex scheduling decisions, compared with simple mutex
  - *i.e.*, a mutex makes *other* threads wait, whereas a condition variable allows a thread to make *itself* wait for a particular condition involving shared data
  - Usually more efficient/correct than busy waiting...
- Are always used in conjunction with a mutex lock

The ACE Condition Variable API

- Only valid for threads in the same process

```
class Condition_Thread_Mutex
{
public:
  Condition_Thread_Mutex (const Thread_Mutex &);
  // Initialize the condition variable.
  ~Condition_Thread_Mutex (void);
  // Implicitly destroy the condition variable.

  int wait (Time_Value *time = 0) const;
  // Block on condition, or until time has
  // passed.  If time == 0 use blocking semantics.
  int signal (void) const;
  // Signal one waiting thread.
  int broadcast (void) const;
  // Signal *all* waiting threads.
  // ...
private:
  ACE_Thread_Mutex &mutex_; // Hold by reference...
```

Programming with ACE Condition Variables

- A particular idiom is typically associated with condition variables

```
// Global variables
static Thread_Mutex m;
static Condition_Thread_Mutex c (m);

void acquire_resources (int resources_needed)
{
      m.acquire ();
      while (condition expression is not true)
            c.wait ();

      // Atomically modify shared information...

      m.release ();
}
```

- Warning!!!! Always make sure to invoke condition variable methods while holding the associated mutex lock!!!

Programming with ACE Condition Variables (cont'd)

- Another idiom is associated with releasing resources via condition variables

```
void release_resources (int resources_released)
{
      // Acquire the lock.
      m.acquire ();

      // Atomically modify shared information here...

      c.signal ();
      // Could also use broadcast().
      m.release ();
}
```


Automating Mutex Acquisition and Release

- To ensure mutexes are locked and unlocked, we'll define a template class that acquires and releases a mutex automatically

```
template <class LOCK>
class Guard
{
public:
  Guard (LOCK &m): lock (m) { lock_.acquire (); }
  ~Guard (void) { lock_.release (); }
  // ... other methods omitted ...

private:
  LOCK &lock_;
}
```

- `Guard` uses the C++ idiom whereby a *constructor acquires a resource* and the *destructor releases the resource*

Guard Design Rationale

- Guard is parameterized by the type of **lock** since there are many different flavors of locking that benefit from the `Guard` protocol, *e.g.*,

  * *Non-recursive vs recursive mutexes*
  * *Intra-process vs inter-process mutexes*
  * *Readers/writer mutexes*
  * *Solaris and System V semaphores*
  * *File locks*
  * *Null mutex*

- In ACE, all synchronization wrappers use to Wrapper Facade and Adapter patterns to provide identical interfaces whenever possible to facilitate parameterization

Programming with Condition Variables and Guards

- Timed wait with a condition variable

```
const int TIMEOUT = 10;
static ACE_Time_Value tv;
static Thread_Mutex m;
static Condition_Thread_Mutex c;
// ...

tv.sec (time (0) + TIMEOUT);
Guard<Thread_Mutex> mon (m);

while (/* cond == FALSE */) {
    int err = c.wait (&tv);
    if (err == ETIME) {
        // handle timeout.
        return −1;
    }
}
return 0;
```


Overview of ACE Readers/Writer Mutexes

- Allow many threads simultaneous read-only access to a protected object
  - However, only a single thread may have write access to the object while excluding any readers or other writers
- Used to protect data that is read more often than written
- Must be fully bracketed (as with mutex)
- Preference is given to writers...

ACE RW_Thread_Mutex API

- Only valid for threads in the same process

```
class ACE_RW_Thread_Mutex {
public:
  ACE_RW_Thread_Mutex (LPCTSTR name = 0, void *arg = 0);
  ~ACE_RW_Thread_Mutex (void); // Default dtor.
  int remove (void);
  // Explicitly destroy a readers/writer lock.
  int acquire_read (void);
  // Acquire a read lock, but block if a writer hold the lock.
  int acquire_write (void);
  // Acquire a write lock, but block if any readers or a
  // writer hold the lock.
  int tryacquire_read (void);
  // Conditionally acquire a read lock (i.e., won't block).
  // Returns -1 on failure.  If we "failed" because someone
  // else already had the lock, <errno> is set to <EBUSY>.
  int tryacquire_write (void);
  // Conditionally acquire a write lock (i.e., won't block).
  int acquire (void);
  // Note, for interface uniformity with other
  // synchronization wrappers we include the <acquire>
  // method.  This is implemented as a write-lock to safe...
  int tryacquire (void);
  // Note, for interface uniformity with other
  // synchronization wrappers we include the <tryacquire>
  // method.  This is implemented as a write-lock to be safe...
  // Returns -1 on failure.  If we "failed" because someone else
  // already had the lock, <errno> is set to <EBUSY>.
  int release (void);
  // Unlock a readers/writer lock.
  // ...
```


Programming with ACE RW_Thread_Mutex

- Concurrent bank account program, supports multiple readers, but only 1 writer...

```
static RW_Mutex account_lock; // Initialized to 0
static float checking_balance = 100.0;
static float savings_balance = 100.0;

float get_balance (void) {
    float bal;

    account_lock.acquire_read ();
    bal = checking_balance + savings_balance;
    account_lock.release ();
    return val;
}

void transfer_checking_to_savings (float amount) {
    account_lock.acquire_write ();
    checking_balance = checking_balance − amount;
    savings_balance = savings_balance + amount;
    account_lock.release ();
}
```

Comparison of Synchronization Primitives

- Mutex locks are the most basic and most efficient in terms of time and space
  - Often based on adaptive spin-locks
- Condition variables provide a different flavor of locking than mutexes and semaphores
  - *i.e.*, blocking themselves rather than blocking others, and waiting on complex state changes
  - They are *much* less efficient than mutexes since they use sleep locks

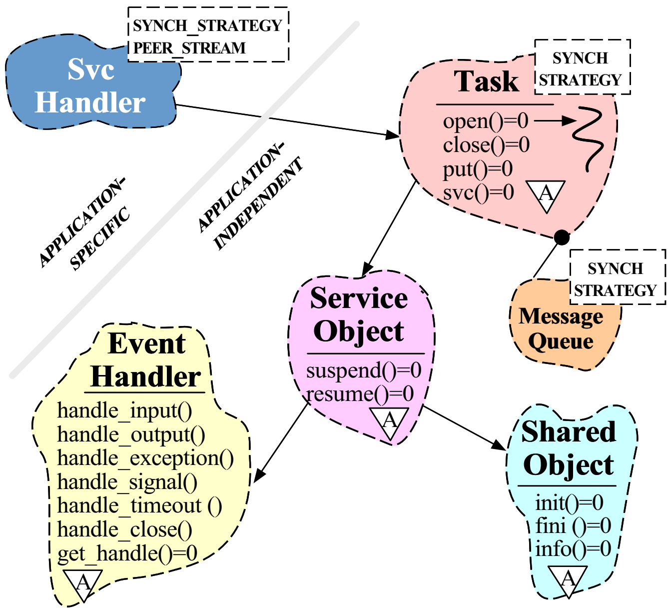
Comparison of Synchronization Primitives (cont'd)

- Semaphores use more memory than mutexes and condition variables
  - Unlike mutexes, they do not require that the original thread is also the thread to release the semaphore
    - They also allow more general "counting" behavior, as opposed to binary behavior
  - Unlike condition variables they method only on count state, rather than complex condition state
- Readers/writer locks are the most complex synchronization mechanism
  - Use at a fairly coarse-grained level

ACE Tasks

- An ACE **Task** binds a separate thread of control together with an object's data and methods
  - Multiple active objects may execute in parallel in separate lightweight or heavyweight processes
- **Task** objects communicate by passing typed messages to other **Task**s
  - Each **Task** maintains a queue of pending messages that it processes in *priority order*
- ACE **Task** are a low-level mechanism to support "active objects"

Task Inheritance Hierarchy



- Supports dynamically configured services

Task Class Public Interface

- C++ interface for message processing

  * **Tasks** can register with a **Reactor**
  * They can be dynamically linked
  * They can queue data
  * They can run as "active objects"

- *e.g.*,

```
template <class SYNCH_STRATEGY>
class Task : public Service_Object
{
public:
    // Initialization/termination hooks.
  virtual int open (void *args = 0) = 0;
  virtual int close (u_long flags = 0) = 0;

    // Hook to pass msg for immediate processing.
  virtual int put (Message_Block *,
                   Time_Value * = 0) = 0;

    // Hook run by daemon thread(s) for
    // deferred processing.
  virtual int svc (void) = 0;

    // Turn task into an active object.
  int activate (long flags, int n_threads = 1);
```


Task Class Protected Interface

- The following methods are mostly used within `put` and `svc`

```
  // Accessors to internal queue.
Message_Queue<SYNCH_STRATEGY> *msg_queue (void);
void msg_queue (Message_Queue<SYNCH_STRATEGY> *);

  // Accessors to thread manager.
Thread_Manager *thr_mgr (void);
void thr_mgr (Thread_Manager *);

  // Insert message into the message list.
int putq (Message_Block *, Time_Value *tv = 0);

  // Extract the first message from the list (blocking).
int getq (Message_Block *&mb, Time_Value *tv = 0);

  // Hook into the underlying thread library.
static void *svc_run (Task<SYNCH_STRATEGY> *);
```


OO Design Interlude

- Q: *What is the svc_run() function and why is it a static method?*

- A: OS thread spawn APIs require a C-style function as the entry point into a thread

- The `Stream` class category encapsulates the `svc_run` function within the `Task::activate` method:

```
template <class SYNCH_STRATEGY> int
Task<SYNCH_STRATEGY>::activate (long flags, int n_threads)
{
  if (thr_mgr () == NULL)
    thr_mgr (Thread_Manager::instance ());

  thr_mgr ()->spawn_n
    (n_threads, &Task<SYNCH_STRATEGY>::svc_run,
     (void *) this, flags);
}
```


OO Design Interlude (cont'd)

- `Task::svc_run` is static method used as the entry point to execute an instance of a service concurrently in its own thread

```
template <class SYNCH_STRATEGY> void *
Task<SYNCH_STRATEGY>::svc_run (Task<SYNCH_STRATEGY> *t)
{
  // Thread added to thr_mgr()
  // automatically on entry...

  // Run service handler and record return value.
  void *status = (void *) t->svc ();

  t->close (u_long (status));

  // Status becomes `return' value of thread...
  return status;

  // Thread removed from thr_mgr()
  // automatically on return...
}
```

OO Design Interlude

- Q: "How can groups of collaborating threads be managed atomically?"

- A: Develop a "thread manager" class

  – `Thread_Manager` is a collection class

    * It provides mechanisms for *suspending* and *resuming* groups of threads atomically

    * It implements *barrier synchronization* on thread exits

  – `Thread_Manager` also shields applications from incompabitilities between different OS thread libraries

    * It is integrated into ACE via the `Task::activate` method

The Active Object Pattern

- *Intent*

  – "Decouple method execution from method invocation and simplifies synchronized access to shared resources by concurrent threads"

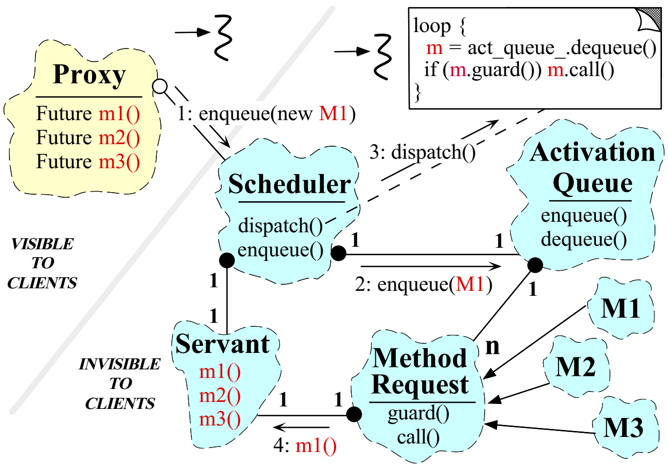
- This pattern resolves the following forces for concurrent communication software:

  – *How to allow blocking read and write operations on one endpoint that do not detract from the quality of service of other endpoints*

  – *How to serialize concurrent access to shared object state*

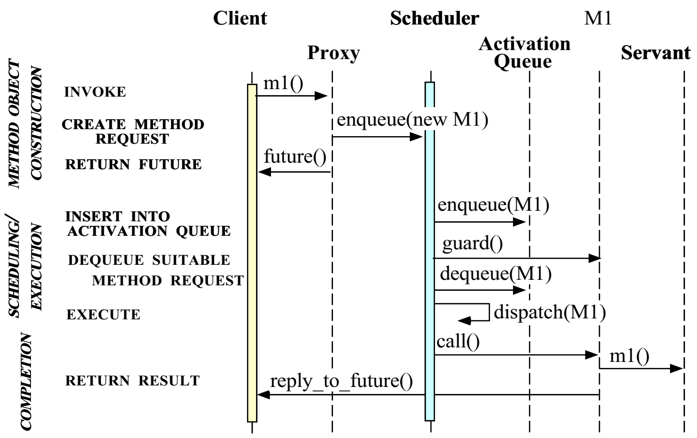
  – *How to simplify composition of independent services*

Structure of the Active Object Pattern

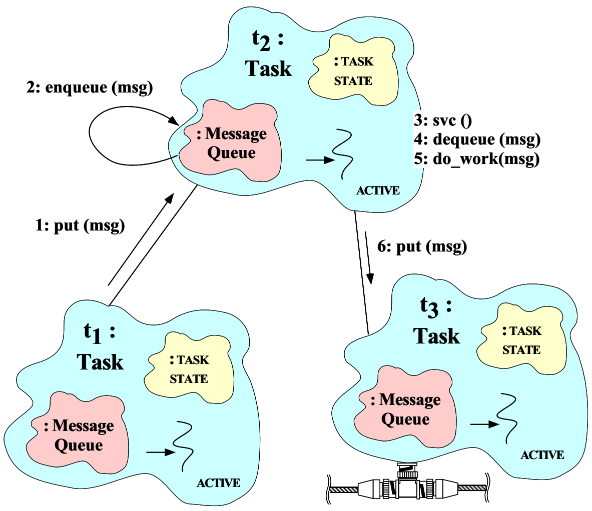


- www.cs.wustl.edu/~schmidt/Act-Obj.ps.gz

Collaboration in the Active Object Pattern



ACE Support for Active Objects

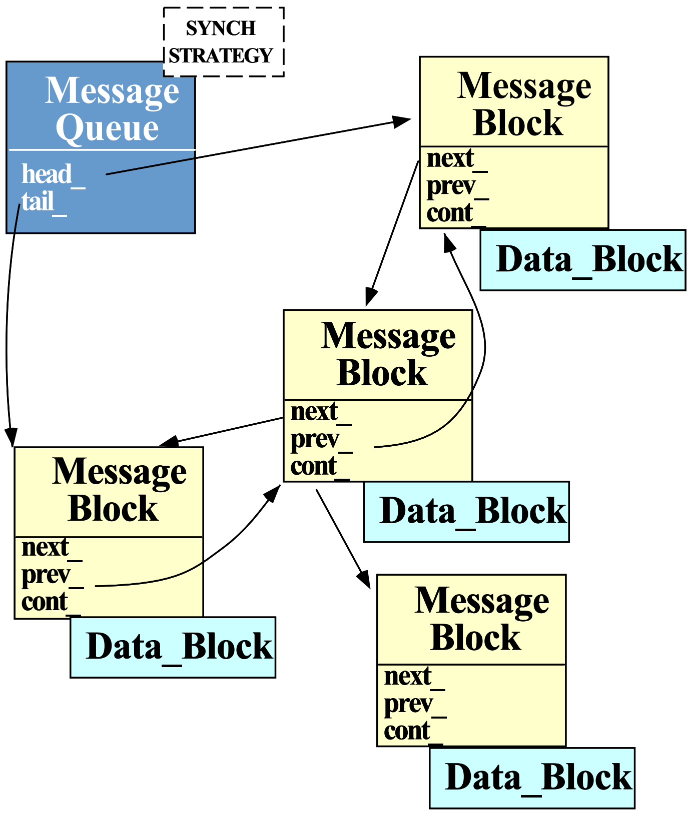


- Can implement complete Active Object pattern or lighterweight subsets

Overview of Message_Queue and Message_Block Classes

- A `Message_Queue` is composed of one or more `Message_Blocks`
  - Similar to BSD `mbufs` or SVR4 STREAMS `m_blks`
  - Goal is to enable efficient manipulation of arbitrarily-large message payloads *without* incurring unnecessary memory copying overhead
- `Message_Blocks` are linked together by `prev_` and `next_` pointers
- A `Message_Block` may also be linked to a chain of other `Message_Blocks`

Message_Queue and Message_Block Object Diagram



The Message_Block Class

- The contents of a message are represented by a Message_Block

```
class Message_Block
{
friend class Message_Queue;
public:
  Message_Block (size_t size,
                 Message_Type type = MB_DATA,
                 Message_Block *cont = 0,
                 char *data = 0,
                 Allocator *alloc = 0);
  // ...

private:
  char *base_;
    // Pointer to beginning of payload.
  Message_Block *next_;
    // Pointer to next message in the queue.
  Message_Block *prev_;
    // Pointer to previous message in the queue.
  Message_Block *cont_;
    // Pointer to next fragment in this message.
    // ...
};
```


OO Design Interlude

- Q: *What is the Allocator object in the Message_Block constructor?*

- A: *It provides extensible mechanism to control how memory is allocated and deallocated*
  - This makes it possible to switch memory management policies *without* modifying Message_Block
  - By default, the policy is to use **new** and **delete**, but it's easy to use other schemes, *e.g.*,
    * Shared memory
    * Persistent memory
    * Thread-specific memory
  - A similar technique is also used in the C++ Standard Template Library

OO Design Interlude

- Here's an example of the interfaces used in ACE
  - Note the use of the Adapter pattern to integrate third-party memory allocators

```
class Allocator {
  // ...
  virtual void *malloc (size_t nbytes) = 0;
  virtual void free (void *ptr) = 0;
};

template <class ALLOCATOR>
class Allocator_Adapter : public Allocator {
  // ...
  virtual void *malloc (size_t nbytes) {
    return allocator_.malloc (nbytes);
  }

  ALLOCATOR allocator_;
};

Allocator_Adapter<Shared_Alloc> sh_malloc;
Allocator_Adapter<New_Alloc> new_malloc;
Allocator_Adapter<Persist_Alloc> p_malloc;
Allocator_Adapter<TSS_Alloc> p_malloc;
```


The Message_Queue Class Public Interface

- A Message_Queue is a thread-safe queueing facility for Message_Blocks
  - The bulk of the locking is performed in the public methods

```
template <class SYNCH_STRATEGY>
class Message_Queue
{
public:
    // Default high and low water marks.
  enum { DEFAULT_LWM = 0, DEFAULT_HWM = 4096 };

    // Initialize a Message_Queue.
  Message_Queue (size_t hwm = DEFAULT_HWM,
                 size_t lwm = DEFAULT_LWM);

    // Check if full or empty (hold locks)
  int is_empty (void) const;
  int is_full (void) const;

    // Enqueue and dequeue Message_Block *'s.
  int enqueue_prio (Message_Block *, Time_Value *);
  int enqueue_tail (Message_Block *, Time_Value *);
  int dequeue_head (Message_Block *&, Time_Value *);
```

The Message_Queue Class Private Interface

- The bulk of the work is performed in the private methods

```
private:
    // Routines that actually do the enqueueing and
    // dequeueing (do not hold locks).
  int enqueue_prio_i (Message_Block *, Time_Value *);
  int enqueue_tail_i (Message_Block *new_item, Time_Value *
  int dequeue_head_i (Message_Block *&first_item);

    // Check the boundary conditions (do not hold locks).
  int is_empty_i (void) const;
  int is_full_i (void) const;

    // ...

    // Parameterized types for synchronization
    // primitives that control concurrent access.
    // Note use of C++ "traits"
  SYNCH_STRATEGY::MUTEX     lock_;
  SYNCH_STRATEGY::CONDITION not_empty_cond_;
  SYNCH_STRATEGY::CONDITION not_full_cond_;
};
```


The Message_Queue Class Implementation

- Uses ACE synchronization wrappers

```
template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_empty_i (void) const {
  return cur_bytes_ <= 0 && cur_count_ <= 0;
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_full_i (void) const {
  return cur_bytes_ > high_water_mark_;
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_empty (void) const {
  Guard<SYNCH_STRATEGY::MUTEX> m (lock_);
  return is_empty_i ();
}

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::is_full (void) const {
  Guard<SYNCH_STRATEGY::MUTEX> m (lock_);
  return is_full_i ();
}
```


OO Design Interlude

- Q: *How should locking be performed in an OO class?*

- A: In general, the following general pattern is useful:
  - "Public functions should lock, private functions should not lock"
    - This also helps to avoid intra-class method deadlock...
  - This is actually a variant on a common OO pattern that "public functions should check, private functions should trust"
  - Naturally, there are exceptions to this rule...

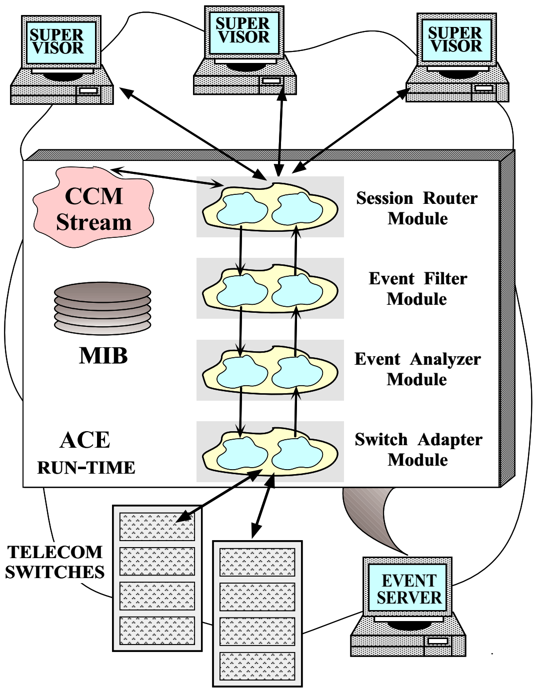

```
// Queue new item at the end of the list.

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::enqueue_tail
  (Message_Block *new_item, Time_Value *tv)
{
  Guard<SYNCH_STRATEGY::MUTEX> monitor (lock_);

  // Wait while the queue is full.

  while (is_full_i ())
    {
      // Release the lock_ and wait for timeout, signal,
      // or space becoming available in the list.
      if (not_full_cond_.wait (tv) == -1)
        return -1;
    }

  // Actually enqueue the message at the end of the list.
  enqueue_tail_i (new_item);

  // Tell blocked threads that list has a new item!
  not_empty_cond_.signal ();
}
```

```
// Dequeue the front item on the list and return it
// to the caller.

template <class SYNCH_STRATEGY> int
Message_Queue<SYNCH_STRATEGY>::dequeue_head
  (Message_Block *&first_item, Time_Value *tv)
{
  Guard<SYNCH_STRATEGY::MUTEX> monitor (lock_);

  // Wait while the queue is empty.

  while (is_empty_i ())
    {
      // Release the lock_ and wait for timeout, signal,
      // or a new message being placed in the list.
      if (not_empty_cond_.wait (tv) == -1)
        return -1;
    }

  // Actually dequeue the first message.
  dequeue_head_i (first_item);

  // Tell blocked threads that list is no longer full.
  not_full_cond_.signal ();
}
```


ACE Streams

- An ACE `Stream` allows flexible configuration of layered processing modules
- It is an implementation of the *Pipes and Filters* architectural pattern
  - This pattern provides a structure for systems that process a stream of data
  - Each processing step is encapsulated in a filter component
  - Data is passed through pipes between adjacent filters, which can be re-combined

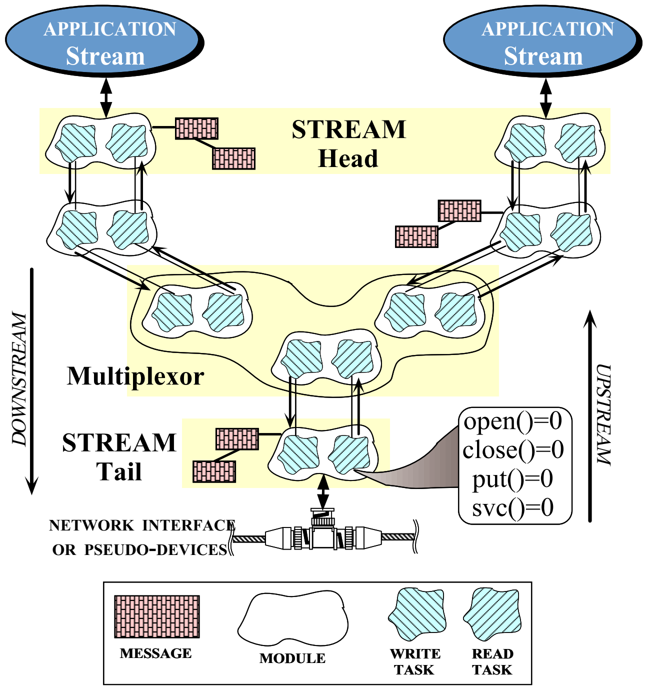
Call Center Manager Example



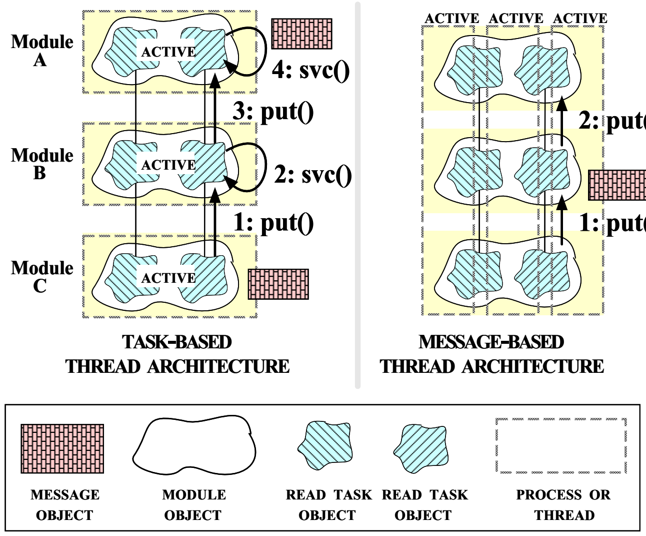
Implementing a Stream in ACE

- A `Stream` contains a stack of `Modules`
- Each `Module` contains two `Tasks`
  - *i.e.*, a *read* `Task` and a *write* `Task`
- Each `Task` contains a `Message_Queue` and a pointer to a `Thread_Manager`
  - `Tasks` can have one or more threads

Stream Class Category



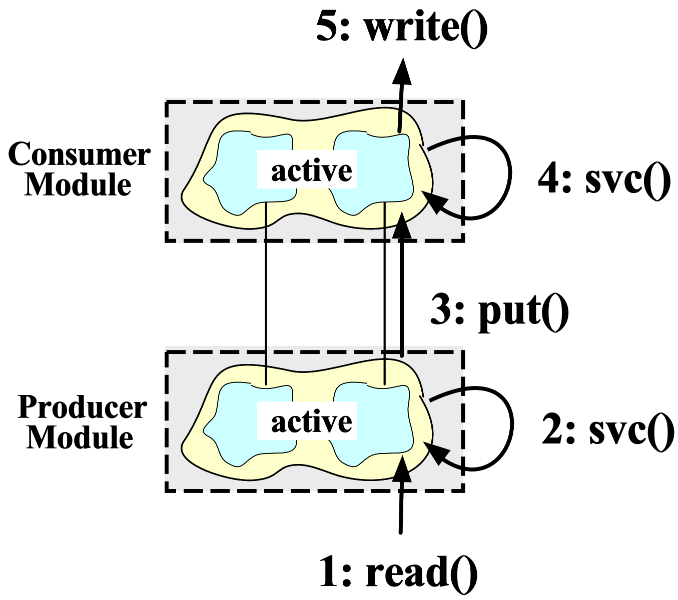
Alternative Invocation Methods



ACE Stream Example: Parallel I/O Copy

- Illustrates an implementation of the classic "bounded buffer" problem
- The program copies stdin to stdout via the use of a multi-threaded `Stream`
- In this example, the "read" `Task` is always ignored since the data flow is unidirectional

Producer and Consumer Object Interactions



Producer Interface

- *e.g.*,

```
// typedef short-hands for the templates.
typedef Stream<MT_SYNCH> MT_Stream;
typedef Module<MT_SYNCH> MT_Module;
typedef Task<MT_SYNCH> MT_Task;

// Define the Producer interface.

class Producer : public MT_Task
{
public:
    // Initialize Producer.
  virtual int open (void *)
  {
    // activate() is inherited from class Task.
    activate (THR_NEW_LWP);
  }

    // Read data from stdin and pass to consumer.
  virtual int svc (void);
  // ...
};
```



```
// Run in a separate thread.

int
Producer::svc (void)
{
  for (; ; ) {
    Message_Block *mb;

    // Allocate a new message.
    ACE_NEW_RETURN (mb, Message_Block (BUFSIZ), -1);

    // Keep reading stdin, until we reach EOF.

    ssize_t n = ACE_OS::read (STDIN,
                              mb->wr_ptr (),
                              mb->size ());
    if (n <= 0) {
      // Send a shutdown message to other thread and exit.
      mb->length (0);
      this->put_next (mb);
      break;
    }
    else {
      mb->wr_ptr (n); // Adjust write pointer.

      // Send the message to the other thread.
      this->put_next (mb);
    }
  }
  return 0;
}
```


Consumer Class Interface

- *e.g.*,

```
// Define the Consumer interface.

class Consumer : public MT_Task
{
public:
    // Initialize Consumer.
  virtual int open (void *)
  {
    // activate() is inherited from class Task.
    activate (THR_NEW_LWP);
  }

    // Enqueue the message on the Message_Queue for
    // subsequent processing in svc().
  virtual int put (Message_Block*, Time_Value* = 0)
  {
    // putq() is inherited from class Task.
    return putq (mb, tv);
  }

    // Receive message from producer and print to stdout.
  virtual int svc (void);
};
```



```
// The consumer dequeues a message from the Message_Queue,
// writes the message to the stderr stream, and deletes
// the message.  The Consumer sends a 0-sized message to
// inform the consumer to stop reading and exit.

int
Consumer::svc (void)
{
  Message_Block *mb = 0;

  // Keep looping, reading a message out of the queue,
  // until we get a message with a length == 0,
  // which informs us to quit.

  for (;;)
    {
      int result = getq (mb);

      if (result == -1) break;
      int length = mb->length ();

      if (length > 0)
        ACE_OS::write (STDOUT, mb->rd_ptr (), length);

      delete mb;

      if (length == 0) break;
    }
  return 0;
}
```

Main Driver Function

- *e.g.*,

```
int main (int argc, char *argv[])
{
  // Control hierachically-related active objects.
  MT_Stream stream;

  // Create Producer and Consumer Modules and push
  // them onto the Stream.  All processing is then
 // performed in the Stream.

  stream.push (new MT_Module ("Consumer",
                              new Consumer);
  stream.push (new MT_Module ("Producer",
                              new Producer));

  // Barrier synchronization: wait for the threads,
  // to exit, then exit ourselves.
  Thread_Manager::instance ()->wait ();
  return 0;
}
```


Evaluation of the Stream Class Category

- Structuring active objects via a `Stream` allows "interpositioning"
  - Similar to adding a filter in a UNIX pipeline

- New functionality may be added by "pushing" a new processing Module onto a `Stream`, *e.g.*,

```
stream.push (new MT_Module ("Consumer",
                            new Consumer))
stream.push (new MT_Module ("Filter",
                            new Filter));
stream.push (new MT_Module ("Producer",
                            new Producer));
```


Conclusions and Caveats

- ACE provides a wide range of portable concurrency and synchronization mechanisms

- However, some applications do not benefit directly from threads, *e.g.*,
  - CPU-bound programs on a uni-processor
  - Threads should be created for processing that lasts at least several thousand machine instructions
  - Synchronization may be expensive
    - Therefore, choose primitives carefully
  - Developer intuition is often underdeveloped...
  - Debugging is more complicated
    - *e.g.*, lack of tools