An Introduction to Programming with Threads

• Read the Birrell paper
  – excellent introductory paper
  – promotes understanding the material
  – abstract content with direct application
    • limited and rather outdated concrete technical content
Threads

• a thread is a single sequential flow of control
  – a process can have many threads and a single address space
  – threads share memory and, hence, need to cooperate to produce correct results
  – thread has thread specific data (registers, stack pointer, program counter…)
Why use threads

• Threads are useful because of real-world parallelism:
  – input/output devices (flesh or silicon) may be slow but are independent
  – distributed systems have many computing entities
  – multi-processors are becoming more common, and future platforms will all be multi-core
  – better resource sharing & utilization then processes
Thread Mechanisms

• Birrell identifies four mechanisms used in threading systems:
  – thread creation
  – mutual exclusion
  – waiting for events
  – interrupting a thread’s wait

• In most mechanisms in current use, only the first three are covered

• primitives used abstract, not derived from actual threading system or programming language!
Example Thread Primitives

- **Thread creation**
  - Thread type
  - `Fork(proc, args)` returns thread
  - `Join(thread)` returns value

- **Mutual Exclusion**
  - Mutex type
  - `Lock(mutex)`, a block-structured language construct in this lecture
Example Thread Primitives

- **Condition Variables**
  - Condition type
  - `Wait(mutex, condition)`
  - `Signal(condition)`
  - `Broadcast(condition)`

- **Fork, Wait, Signal, etc.** are not to be confused with the UNIX “fork”, “wait”, “signal”, etc. calls.
Creation Example

Thread thread1;
thread1 = Fork(safe_insert, 4);
safe_insert(6);
Join(thread1); // Optional
Mutex Example

```cpp
list<int> my_list;
Mutex m;

void safe_insert(int i) {
  Lock(m) {
    my_list.insert(i);
  }
}
```
Condition Variables

- Mutexes are used to control access to shared data
  - only one thread can execute inside a Lock clause
  - other threads who try to Lock, are blocked until the mutex is unlocked
- Condition variables are used to wait for specific events
  - free memory is getting low, wake up the garbage collector thread
  - 10,000 clock ticks have elapsed, update that window
  - new data arrived in the I/O port, process it
- Could we do the same with mutexes?
  - (think about it and we’ll get back to it)
Condition Variable Example

Mutex io_mutex;
Condition non_empty;
...
Consumer:
Lock (io_mutex) {
    while (port.empty())
        Wait(io_mutex, non_empty);
    process_data(port.first_in());
}
Producer:
Lock (io_mutex) {
    port.add_data();
    Signal(non_empty);
}
Condition Variables Semantics

- Each condition variable is associated with a single mutex
- \textit{Wait} \textit{atomically} unlocks the mutex and blocks the thread
- \textbf{Signal} awakes a blocked thread
  - the thread is awoken inside \textit{Wait}
  - tries to lock the mutex
  - when it (finally) succeeds, it returns from the \textit{Wait}
- Doesn’t this sound complex? Why do we do it?
  - the idea is that the “condition” of the condition variable depends on data protected by the mutex
Mutex io_mutex;
Condition non_empty;
...
Consumer:
Lock (io_mutex) {
    while (port.empty())
        Wait(io_mutex, non_empty);
    process_data(port.first_in());
}
Producer:
Lock (io_mutex) {
    port.add_data();
    Signal(non_empty);
}
Couldn’t We Do the Same with Plain Communication?

Mutex io_mutex;

...  

Consumer:
Lock (io_mutex) {
    while (port.empty())
        go_to_sleep(non_empty);
    process_data(port.first_in());
}

Producer:
Lock (io_mutex) {
    port.add_data();
    wake_up(non_empty);
}

• What’s wrong with this? What if we don’t lock the mutex (or unlock it before going to sleep)?
Mutexes and Condition Variables

- Mutexes and condition variables serve different purposes
  - Mutex: exclusive access
  - Condition variable: long waits

- Question: Isn’t it weird to have both mutexes and condition variables? Couldn’t a single mechanism suffice?

- Answer:
Use of Mutexes and Condition Variables

- Protect shared mutable data:

```c
void insert(int i) {
    Element *e = new Element(i);
    e->next = head;
    head = e;
}
```

- What happens if this code is run in two different threads with no mutual exclusion?
Mutex io_mutex;
Condition non_empty;

Consumer:
Lock (io_mutex) {
    while (port.empty())
        Wait(io_mutex, non_empty);
    process_data(port.first_in());
}

Producer:
Lock (io_mutex) {
    port.add_data();
    Signal(non_empty);
}

Why use \texttt{while} instead of \texttt{if}? (think of many consumers, simplicity of coding producer)
Readers/Writers Locking

Mutex counter_mutex;
Condition read_phase,
    write_phase;
int readers = 0;

**Reader:**
Lock(counter_mutex) {
    while (readers == -1)
        Wait(counter_mutex,
            read_phase);
    readers++;
}
... //read data
Lock(counter_mutex) {
    readers--;
    if (readers == 0)
        Signal(write_phase);
}

**Writer:**
Lock(counter_mutex) {
    while (readers != 0)
        Wait(counter_mutex,
            write_phase);
    readers = -1;
}
... //write data
Lock(counter_mutex) {
    readers = 0;
    Broadcast(read_phase);
    Signal(write_phase);
}
Comments on Readers/Writers Example

• Invariant: readers >= -1
• Note the use of Broadcast
• The example could be simplified by using a single condition variable for phase changes
  – less efficient, easier to get wrong
• Note that a writer signals all potential readers and one potential writer. Not all can proceed, however
  – (spurious wake-ups)
• Unnecessary lock conflicts may arise (especially for multiprocessors):
  – both readers and writers signal condition variables while still holding the corresponding mutexes
  – Broadcast wakes up many readers that will contend for a mutex
Readers/Writers Example

Reader:
Lock(mutex) {
    while (writer)
        Wait(mutex, read_phase)
    readers++;
}

… // read data

Lock(mutex) {
    readers--;
    if (readers == 0)
        Signal(write_phase);
}


Writer:
Lock(mutex) {
    while (readers != 0 || writer)
        Wait(mutex, write_phase)
    writer = true;
}

… // write data

Lock(mutex) {
    writer = false;
    if (readers == 0)
        Broadcast(read_phase);
    Signal(write_phase);
}
Avoiding Unnecessary Wake-ups

Mutex counter_mutex;
Condition read_phase, write_phase;
int readers = 0, waiting_readers = 0;

**Reader:**

Lock(counter_mutex) {
    waiting_readers++;
    while (readers == -1)
        Wait(counter_mutex, read_phase);
    waiting_readers--;
    readers++;
}

... //read data

Lock(counter_mutex) {
    readers--;
    if (readers == 0)
        Signal(write_phase);
}

**Writer:**

Lock(counter_mutex) {
    while (readers != 0)
        Wait(counter_mutex, write_phase);
    readers = -1;
}

... //write data

Lock(counter_mutex) {
    readers = 0;
    if (waiting_readers > 0)
        Broadcast(read_phase);
    else
        Signal(write_phase);
}
Problems With This Solution

• Explicit scheduling: readers always have priority
  – may lead to starvation (if there are always readers)
  – fix: make the scheduling protocol more complicated than it is now

To Do:

• Think about avoiding the problem of waking up readers that will contend for a single mutex if executed on multiple processors
Deadlocks (brief)

• We’ll talk more later… for now beware of deadlocks
• Examples:
  – A locks M1, B locks M2, A blocks on M2, B blocks on M1
  – Similar examples with condition variables and mutexes
• Techniques for avoiding deadlocks:
  – Fine grained locking
  – Two-phase locking: acquire all the locks you’ll ever need up front, release all locks if you fail to acquire any one
    • very good technique for some applications, but generally too restrictive
  – Order locks and acquire them in order (e.g., all threads first acquire M1, then M2)
• The scope of multithreading
LWPs, kernel and user threads

• kernel-level threads – supported by the kernel
  – Solaris, Linux, Windows XP/2000
  – all scheduling, synchronization, thread structures maintained in kernel
  – could write apps using kernel threads, but would have to go to kernel for everything

• user-level threads – supported by a user-level library
  – Pthreads, Java threads, Win32…
  – sched. & synch can often be done fully in user space; kernel doesn’t need to know there are many user threads
  – problem with blocking on a system call
• LightWeight Processes - LWP
  – these are “virtual CPUs”, can be multiple per process
  – the scheduler of a threads library schedules user-level threads to these virtual CPUs
  – kernel threads implement LWPs => visible to the kernel, and can be scheduled
    • sometimes LWP & kernel threads used interchangeably, but there can be kernel threads without LWPs
Multithreading models

- There are three dominant models for thread libraries, each with its own trade-offs
  - many threads on one LWP (many-to-one)
  - one thread per LWP (one-to-one)
  - many threads on many LWPs (many-to-many)

- similar models can apply on scheduling kernel threads to real CPUs
Many-to-one

• In this model, the library maps all threads to a single lightweight process

• Advantages:
  – totally portable
  – easy to do with few systems dependencies

• Disadvantages:
  – cannot take advantage of parallelism
  – may have to block for synchronous I/O
  – there is a clever technique for avoiding it

• Mainly used in language systems, portable libraries
One-to-one

• In this model, the library maps each thread to a different lightweight process

• Advantages:
  – can exploit parallelism, blocking system calls

• Disadvantages:
  – thread creation involves LWP creation
  – each thread takes up kernel resources
  – limiting the number of total threads

• Used in LinuxThreads and other systems where LWP creation is not too expensive
Many-to-many

- In this model, the library has two kinds of threads: *bound* and *unbound*
  - bound threads are mapped each to a single lightweight process
  - unbound threads *may* be mapped to the same LWP
- Probably the best of both worlds
- Used in the Solaris implementation of Pthreads (and several other Unix implementations)
High-Level Program Structure Ideas

• Boss/workers model
• Pipeline model
• Up-calls
• Keeping shared information consistent using version stamps
Thread Design Patterns

Common ways of structuring programs using threads

- **Boss/workers model**
  - boss gets assignments, dispatches tasks to workers
  - variants (thread pool, single thread per connection…)
- **Pipeline model**
  - do some work, pass partial result to next thread
- **Up-calls**
  - fast control flow transfer for layered systems
- **Version stamps**
  - technique for keeping information consistent
**Boss/Workers**

**Boss:**

```java
forever {
    get a request
    switch(request)
        case X: Fork (taskX)
        case Y: Fork (taskY)
        ...
}
```

**Worker:**

```java
    taskX();
```

- **Advantage:** simplicity
- **Disadvantage:** bound on number of workers, overhead of threads creation, contention if requests have interdependencies
- **Variants:** fixed thread pool (aka *workpile, workqueue*), producer/consumer relationship, workers determine what needs to be performed…
Pipeline

- Each thread completes portion of a task, and passes results
- like an assembly line or a processor pipeline
- Advantages: trivial synchronization, simplicity
- Disadvantages: limits degree of parallelism, throughput driven by slowest stage, handtuning needed
Up-calls

- Layered applications, e.g. network protocol stacks have top-down and bottom-up flows
- Up-calls is a technique in which you structure layers so that they can expect calls from below
- Thread pool of specialized threads in each layer
  - essentially an up-call pipeline per connection
- Advantages: best when used with fast, synchronous control flow transfer mechanisms or program structuring tool
- Disadvantages: programming becomes more complicated, synchronization required for top-down
Version Stamps

• (not a programming structure idea but useful technique for any kind of distributed environment)

• maintain “version number” for shared data
  – keep local cached copy of data
  – check versions to determine if changed