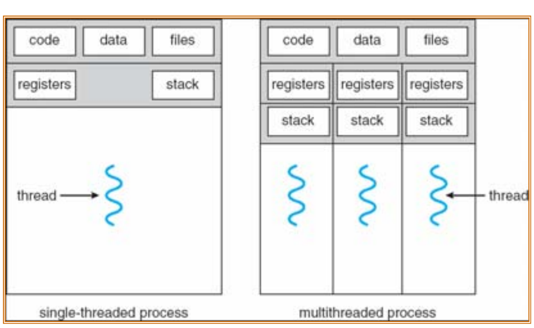
An Introduction to Programming with Threads

- 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    - limited and rather outdated concrete technical content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
  - 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

```
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
- Wait *atomically* unlocks the mutex and blocks the thread
- Signal awakes a blocked thread
  - the thread is awoken inside Wait
  - tries to lock the mutex
  - when it (finally) succeeds, it returns from the 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

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);
}
```

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:

```
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?

Using Condition Variables

```
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 while instead of 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;
  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)