An Introduction to Programming with Threads

Andrew D. Birrell
January 1989

Presented by
Micah Sherr
September 2006
But first, a quick history lesson
(from somewhat questionable sources)

• 1965: Djikstra's “processes” in the Berkeley Timesharing System
  - communication via shared variables and semaphores, shared address space

• PL/I, circa 1965
  - provided TASK function which forked a thread with a shared address space

• 1970s: UNIX processes:
  - sequential program flow and private virtual address space (heavyweight threads)

• Late 1970s: Lightweight thread support in UNIX

• 1990s: POSIX (Portable Operating System Interface for uniX) threads introduced

1989: An introduction to Programming with Threads
What's a thread?

• Thread: serial execution of code
• Concurrency achieved by simultaneously executing multiple threads
• Flavors:
  - *heavyweight*: (AKA modern-day processes) heavyweight threads have their own private resources (e.g., memory space, file handles, etc.)
  - *lightweight*: each thread has its own stack, but all threads within the same process share global address space; communication mostly through global variables
What good are they?

- Run slower jobs (e.g., disk or network accesses) in separate thread
  - assumes no data dependencies
- User Interfaces: humans are multitasking and expect the same from their computers
- Defer expensive operation (e.g., balancing tree) to period of low-CPU utilization
- Take advantage of multiple processors/cores
  - in multicore systems, threads can actually execute in parallel
  - same true for processes, but processes have greater communication overhead
A one-slide primer on how to use threads

- Explicitly create threads
  - POSIX: `pthread_create`
  - Modula-2+: `fork`

- Identify critical sections and use mutexes to ensure mutual exclusion
  - POSIX: `pthread_mutex_lock` / `pthread_mutex_unlock`
  - Modula-2+: `LOCK mutex DO ... statements ... END;`

- Optionally, wait for thread to complete
  - POSIX: `pthread_join`
  - Modula-2+: `join`

- Terminate the thread
  - POSIX: `pthread_exit`
Critical Sections

```c
int array[100] = { 0, ... , 0};
int i = 0;

void addToArray( int n ) {
    array[i] = n;
    i++;
}
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread A</th>
<th>Thread B</th>
<th>i</th>
<th>array[0-50]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>addToArray(1)</td>
<td>0</td>
<td>[1,0,0,0,0,0,...]</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>0</td>
<td>[1,0,0,0,0,0,...]</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>addToArray(2)</td>
<td>0</td>
<td>[2,0,0,0,0,0,...]</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td>0</td>
<td>[2,0,0,0,0,0,...]</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td></td>
<td>1</td>
<td>[2,0,0,0,0,0,...]</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td></td>
<td>2</td>
<td>[2,0,3,0,0,0,...]</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td></td>
<td>2</td>
<td>[2,0,3,0,0,0,...]</td>
</tr>
<tr>
<td>45</td>
<td>2</td>
<td></td>
<td>2</td>
<td>[2,0,3,0,0,0,...]</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td></td>
<td>3</td>
<td>[2,0,3,0,0,0,...]</td>
</tr>
</tbody>
</table>
```

BAD
Mutual Exclusion

- Identify *critical sections* of code and prevent multiple threads from simultaneously executing critical section
- To find critical section, identify the *invariant*
  - In previous example, \( i \) should always refer to next empty space in array
- Use *mutexes* (locks) to protect critical section

```c
void addToArray( int n ) {
    pthread_mutex_lock( &my_mutex );
    array[i] = n;
    i++;
    pthread_mutex_unlock( &my_mutex );
}
```
Deadlocks

- Each thread waits for mutex held by another process, e.g.,
  - Thread A holds mutex M1
  - Thread B holds mutex M2
  - Thread A tries to acquire M2
  - Thread B tries to acquire M1

- Many deadlock avoidance schemes
- Birrell: apply partial order to acquisition of mutexes
  - All threads that want to hold Mi...Mj simultaneously must request threads in same order
  - In above example, thread B can't acquire M2 before it gets M1, breaking circular wait

4 conditions for deadlock:
- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait
Problems with Locks

• Locks allow simple (one-thread-at-a-time) scheduling; more complicated scheduling sometimes desirable
  – e.g., wakeup threads A and B when thread C computes D.

• Locks can result in poor performance
  – inefficient processor utilization when thread that has resource stops making progress (e.g., blocks due to I/O)
Moving beyond locks: Condition Variables

- **Condition variables** allow thread to block until some event occurs
  - **Wait**: unlock mutex and wait for condition to be true
  - **Signal**: signals an event to 1+ waiting threads
  - **Broadcast**: signals an event to all waiting threads

- Condition variables always associated w/ mutex, otherwise signal may occur immediately before thread waits

- Condition variables in POSIX threads:  
  `pthread_cond_init()`, `pthread_cond_signal()`, `pthread_cond_wait()`, `pthread_cond_broadcast()`

```
PROCEDURE Consumer {
  LOCK m DO
    WHILE head=NIL DO { Wait(m,NonEmpty) }
    topElement = head;
    head = head->next;
  END
}

PROCEDURE Producer {
  LOCK m DO
    newElement->next = head;
    head = newElement;
    Signal(nonEmpty);
  END
}
```
Moving beyond locks: Alerts

- Each thread has `alert-pending` boolean
- `AlertWait`: same as `wait`, but if alert-pending is true:
  - sets alert-pending to false
  - reacquires mutex
  - raises exception – a return value that must be caught
- `Alert`: sets alert-pending to true for specified thread
- Example:
  - thread A takes user input
  - thread B provides a GUI, in which user cancels action
  - thread B sends alert to thread A
Designing a Multithreaded App

- Look for invariants – properly identify critical sections
- Avoid spurious wake-ups – don't signal threads that can't progress
- Protect against deadlock – impose partial order on locks
- Better to deadlock than produce incorrect results
Designing a Multithreaded App

- Protect against starvation
- Use alerts to allow operator to abort long computations
- Don't over-thread – imposes too high scheduling cost
- When applicable, use a fixed number of “worker threads”
Designing a Multithreaded App on a Multiprocessor Machine

- Form a chain of producer-consumer threads, called a *pipeline*
  - Thread A initiates request, enqueues it
  - Thread B reads from buffer, performs action, writes to 2\(^{nd}\) buffer
  - Thread C reads from 2\(^{nd}\) buffer, performs action, writes to 3\(^{rd}\) buffer, etc.

- Assumes threads can operate in parallel
- At best, achieves linear speed up w.r.t. number of processors
Software Support

• Threads require OS support:
  – OS must permit memory accesses to same address space from multiple threads

• Re-entrant library functions
  – Library functions must allow calls from multiple threads in parallel
Thread Implementations

- **Kernel:**
  - threads scheduled by OS
  - threads can run on separate processors

- **Userspace:**
  - threads sometimes called *fibers*
  - kernel unaware of fibers
  - efficient context-switches: kernel not involved
  - system calls block all fibers within a process

- Most modern OSes use hybrid of above
Threads vs. Processes: Current outlook

- In Windows, threads usually “cheaper” than processes
- Threads preferred for multicore machines
- GUIs – almost always threaded
- Processes required for services executed under different permissions
- Web servers – mixture of threads and processes (optimization problem)
- Programming with threads hasn't changed much since 1989