PThreads in a Nutshell

Chris Kauffman

CS 499: Spring 2016 GMU
Logistics

Today
- POSIX Threads Briefly

Reading
- Grama 7.1-9 (PThreads)
- POSIX Threads Programming Tutorial

HW4 Upcoming
- Post over the weekend soon
- Due in last week of class
- OpenMP Password Cracking
- PThreads Version
- Exploration of alternative programming models
- Maybe a sorting routine...
Threaded Programming

- OpenMP provided threads via directives (#pragma omp)
- Thread creation, execution, and cleanup all automated
- PThreads is lower-level, similar to fork() / waitpid() of IPC programming
Threads vs IPC

You can mix IPC/Threads if you hate yourself enough.

<table>
<thead>
<tr>
<th>Threads in PThreads</th>
<th>Process in IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast startup</td>
<td>Longer startup</td>
</tr>
<tr>
<td>Memory shared by default</td>
<td>Must share memory explicitly</td>
</tr>
<tr>
<td>Little protection between threads</td>
<td>Good protection between processes</td>
</tr>
</tbody>
</table>

| pthread_create() / join()         | fork() / waitpid()                  |
| Queues, Semaphores,              | Queues, Semaphores, Shared Mem      |
| Mutexes, CondVars                |                                     |

---

![Diagram](image)
Thread Creation

```c
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);

int pthread_join(pthread_t thread, void **retval);
```

- Start a thread running function `start_routine`
- `attr` may be `NULL` for default attributes
- Pass arguments `arg` to the function
- Wait for thread to finish, put return in `retval`
// Minimal example of starting a pthread, passing a
// parameter to the thread function, then waiting for it to
// finish
#include <pthread.h>
#include <stdio.h>

void *fx(void *param){
    int p=(int) param;
    p = p*2;
    return (void *) p;
}

int main(){
    pthread_t thread_1;
    pthread_create(&thread_1, NULL, fx, (void *) 42);
    int xres;
    pthread_join(thread_1, (void **) &xres);
    printf("result is: %d\n",xres);
    return 0;
}
Compilation

```c
> gcc pthreads_minimal_example.c -lpthread
pthreads_minimal_example.c: In function 'fx':
pthreads_minimal_example.c:7:9: warning: cast from pointer to integer of different size [-Wpointer-to-int-cast]
    int p=(int) param;

    ^
pthreads_minimal_example.c:9:10: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    return (void *) p;

> a.out
result is: 84
```
Things to Ask

- How much compiler support do you get with pthreads?
- How does one pass multiple arguments to a function?
- What does the parent thread do on creating a child thread?
- If multiple children are spawned, which execute?
Exercise: A Slice of the Pi

- Recall Monte-Carlo estimation of $\pi$
- Serial code: http://cs.gmu.edu/~kauffman/cs499/picalc.c
- Convert serial version to use PThreads
- How to determine # of threads, thread id
- What info to communicate threads
- How to accumulate results

```c
main()
{
    unsigned int rstate = 123456789;
    int npoints = atoi(argv[1]);
    int total_hits = 0;
    for (int i = 0; i < npoints; i++) {
        double x = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
        double y = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
        if (x*x + y*y <= 1.0){
            total_hits++;
        }
    }
    double pi_est = ((double)total_hits) / npoints * 4.0;
}
```
Recall that this problem is almost *embarassingly parallel*

Very little communication/coordination required

Speedup follows

---

4-proc desktop

```bash
> gcc picalc.c
> time a.out 100000000
npoints: 100000000
hits: 78541355
pi_est: 3.141654

real 2.35
user 2.27
sys 0.00

> gcc pthreads_picalc.c
> time -p a.out 100000000 2
npoints: 100000000
hits: 78539689
pi_est: 3.141588

real 1.36
user 2.53
sys 0.02
```
Timings on Zeus

```
zeus-1 [cs499] % gcc pthreads_picalc.c -lpthread -std=c99
zeus-1 [cs499] % time a.out 100000000 1
...
```

<table>
<thead>
<tr>
<th>#threads</th>
<th>Time (s)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.405</td>
<td>3.340</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.728</td>
<td>3.342</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.180</td>
<td>3.341</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.901</td>
<td>3.340</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.736</td>
<td>3.339</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.622</td>
<td>3.339</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.543</td>
<td>3.340</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.483</td>
<td>3.339</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.596</td>
<td>3.342</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.562</td>
<td>3.341</td>
<td></td>
</tr>
</tbody>
</table>

What kind of speedup are we getting here?
As noted in other answers, pthreads does not define a platform-independent way to retrieve an integral thread ID. This answer [http://stackoverflow.com/a/21206357/316487](http://stackoverflow.com/a/21206357/316487) gives a non-portable way which works on many BSD-based platforms.

– Bleater on Stack Overflow

```c
// Standard opaque object, non-printable??
pthread_t opaque = pthread_self();

// Non-portable, non-linux
pthread_id_np_t tid = pthread_getthreadid_np();

// Linux only
pid_t tid = syscall( __NR_gettid );
printf("Thread %d reporting for duty\n",tid);
```
**Mutual Exclusion**

- POSIX provides mutual exclusion via mutexes (mutices?), commonly referred to as locks.
- Good for thread synchronization, can also be used in IPC sync rather than semaphores/message queues

### Basic pattern

- Crate Lock variable
- Initialize Lock
- ...
- Obtain Lock
- Execute critical section
- Release Lock
- ...
- Destroy Lock

### Posix Calls

```c
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
...
pthread_mutex_lock(&lock);
total_hits++;
pthread_mutex_unlock(&lock);
...
pthread_mutex_destroy(&lock);
```
Picalc with Locks

Adjust code to use global `total_hits`, protect updates using locking

https://cs.gmu.edu/~kauffman/cs499/pthreads_picalc.c
Timing with Locks

zeus-1 [cs499] % gcc pthreads_picalc_locking.c -lpthread -std=c99
zeus-1 [cs499] % time -p a.out 100000000 1

<table>
<thead>
<tr>
<th>#threads</th>
<th>Locks</th>
<th>Lock-Free</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
</tr>
<tr>
<td>1</td>
<td>7.05</td>
<td>6.98</td>
</tr>
<tr>
<td>2</td>
<td>27.62</td>
<td>22.15</td>
</tr>
<tr>
<td>3</td>
<td>25.65</td>
<td>22.60</td>
</tr>
<tr>
<td>4</td>
<td>34.32</td>
<td>29.41</td>
</tr>
<tr>
<td>5</td>
<td>43.65</td>
<td>33.26</td>
</tr>
<tr>
<td>6</td>
<td>41.32</td>
<td>29.15</td>
</tr>
<tr>
<td>7</td>
<td>34.66</td>
<td>23.72</td>
</tr>
<tr>
<td>8</td>
<td>30.68</td>
<td>20.00</td>
</tr>
<tr>
<td>9</td>
<td>29.11</td>
<td>19.21</td>
</tr>
<tr>
<td>10</td>
<td>28.49</td>
<td>18.51</td>
</tr>
</tbody>
</table>

Why are these numbers so much worse than the lock-free version?
# Locks versus Condition Variables

- POSIX Mutexes use busy waiting - occupy CPU time while repeatedly trying to acquire the lock: *Spin Lock* or *Polling*
- Condition variables allow non-busy waiting

## Recall the Semaphore

- Check an integer value atomically
- Increment / decrement that value
- If decrementing would drop below 0, *block*, wait to be notified of non-zero value
- Built-in wait queue to notify blocked processes of changes
- Blocking does not use CPU

## CondVar ≈ Wait Queue

- Only the queue part of a semaphore
  ```c
  // Wait for signals
  pthread_cond_wait(cv, mtx);
  // Wake up waiting thread
  pthread_cond_signal(cv);
  ```
- Required: External variable/variables to indicate state
- Required: Mutex to control access to those variables
Picalc with Condvars

```c
int critical_occupied = 0;
pthread_mutex_t critical_mtx;
pthread_cond_t critical_cv;

...;
double x = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
double y = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
if (x*x + y*y <= 1.0){

// Enter the critical section
pthread_mutex_lock(&critical_mtx);
while(critical_occupied){
    pthread_cond_wait(&critical_cv, &critical_mtx);
}
critical_occupied = 1;
pthread_mutex_unlock(&critical_mtx);

// Update the state
total_hits++;

// Exit the critical section
critical_occupied = 0;
pthread_cond_signal(&critical_cv);
}
```
## Timings

<table>
<thead>
<tr>
<th>#Th</th>
<th>Condvar real</th>
<th>Condvar user</th>
<th>Condvar sys</th>
<th>Mutex real</th>
<th>Mutex user</th>
<th>Mutex sys</th>
<th>Redux real</th>
<th>Redux user</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.59</td>
<td>9.53</td>
<td>0.00</td>
<td>7.05</td>
<td>6.98</td>
<td>0.00</td>
<td>3.405</td>
<td>3.340</td>
</tr>
<tr>
<td>2</td>
<td>29.73</td>
<td>21.62</td>
<td>29.09</td>
<td>27.62</td>
<td>22.15</td>
<td>28.73</td>
<td>1.728</td>
<td>3.342</td>
</tr>
<tr>
<td>3</td>
<td>85.88</td>
<td>62.54</td>
<td>151.82</td>
<td>25.65</td>
<td>22.60</td>
<td>40.07</td>
<td>1.180</td>
<td>3.341</td>
</tr>
<tr>
<td>4</td>
<td>96.39</td>
<td>64.24</td>
<td>226.77</td>
<td>34.32</td>
<td>29.41</td>
<td>94.90</td>
<td>0.901</td>
<td>3.340</td>
</tr>
<tr>
<td>5</td>
<td>126.69</td>
<td>72.78</td>
<td>368.33</td>
<td>43.65</td>
<td>33.26</td>
<td>162.76</td>
<td>0.736</td>
<td>3.339</td>
</tr>
<tr>
<td>6</td>
<td>161.57</td>
<td>79.77</td>
<td>531.09</td>
<td>41.32</td>
<td>29.15</td>
<td>194.13</td>
<td>0.622</td>
<td>3.339</td>
</tr>
<tr>
<td>7</td>
<td>178.40</td>
<td>80.26</td>
<td>646.49</td>
<td>34.66</td>
<td>23.72</td>
<td>197.91</td>
<td>0.543</td>
<td>3.340</td>
</tr>
<tr>
<td>8</td>
<td>192.38</td>
<td>78.64</td>
<td>759.83</td>
<td>30.68</td>
<td>20.00</td>
<td>200.10</td>
<td>0.483</td>
<td>3.339</td>
</tr>
<tr>
<td>9</td>
<td>202.28</td>
<td>78.20</td>
<td>820.12</td>
<td>29.11</td>
<td>19.21</td>
<td>197.84</td>
<td>0.596</td>
<td>3.342</td>
</tr>
<tr>
<td>10</td>
<td>211.73</td>
<td>79.41</td>
<td>834.23</td>
<td>28.49</td>
<td>18.51</td>
<td>197.57</td>
<td>0.562</td>
<td>3.341</td>
</tr>
</tbody>
</table>
More Canonical Examples of Condition Variables

- Picalc is ill-suited for either Mutexes or Condition Variables to control access to the critical section of code.
- More canonical example of condvar is producer/consumer
- Examine pthreads_producer_consumer.c
John’s Solution to Mutex Problems

JohnMillerCards += 5;

- Tried several variants of lock(), trylock() schemes
- Found the following to be the most scalable
  
  ```c
  if (x*x + y*y > 1.0){
      if(pthread_mutex_trylock(&lock) == 0){
          total_miss++;
          pthread_mutex_unlock(&lock);
      }
      else{
          pthread_yield();
          pthread_mutex_lock(&lock);
          total_miss++;
          pthread_mutex_unlock(&lock);
      }
  }
  ```

- Beats the OpenMP critical version for scaling
Scaling of John’s Solutions
Take-Home

- PThreads provide threaded execution within a single program, shared memory
- Primary capability: spawn threads starting different functions
- Provide basic coordination mechanisms for mutual exclusion
- Did not cover large swaths of other facilities (message queues, thread priority and cancellation, etc.) but these exist and should be investigated should the need arise