POSIX Threads

Mark Greenstreet, CpSc 448B, 2011/12, Term 2

Lecture Outline

POSIX Threads

- Count 3's
  - Creating threads
  - Joining threads
- Communication between Threads
  - Shared Memory
  - Locks
  - Signals
- Correctness of shared memory programs
  - Bad stuff: Races, deadlock, livelock
  - Good stuff: Invariants

POSIX Threads

- POSIX threads: a library for writing parallel programs in C for shared-memory, multiprocessors (under Unix).
- Provides functions for thread creation and termination.
- Provides functions for locking (mutual exclusion).
- Provides functions for signaling between threads.

Count 3's: Design

- Given A an array of n integers.
- Let t be the intended number of worker threads.
- Create t threads
  - Each thread counts the number of 3's in a sub-array of roughly n/t elements.
  - Each thread writes its count into a separate element of a $results$ array and then terminates.
- The main thread waits for each worker thread to terminate and adds up their values to get the total number of 3's in A.

Creating a POSIX thread

pthread_create(threadId, threadAttr, thread_fn, thread_arg)

- threadId: a pointer to a pthread_t, a thread identifier;
- threadAttr: attributes for the thread – set it to NULL to get the defaults;
- threadFn: call this function to start execution of the thread;
- threadArg: the parameter to pass to threadFn.
- Corresponds to Erlang spawn(Fun, ArgList):
  - pthread_thread_create corresponds to spawn.
  - thread_fn corresponds to Fun.
  - thread_arg corresponds to ArgList.
  - threadId corresponds to the return value of spawn.
    - Why?
    - Because this is C:
      no explicit exceptions
      return value used to report errors

A thread for counting 3's

```
typedef struct {
    int *a, lo, hi;                    /* count 3's for a[lo..(hi-1)] */
    int *count;                        /* put the local count here */
} c3s_arg;

/* c3s_thread: count the number of threes in a[lo..(hi-1)] */
void *c3s_thread(void *void_arg) {
    c3s_arg *arg = (c3s_arg *)(void_arg);
    int *a = arg->a;                   /* copy arg's fields to local variables */
    int lo = arg->lo;
    int hi = arg->hi;
    int count = 0;
    for(int i = lo; i < hi; i++)       /* count */
        count += a[i] == 3;
    *(arg->count) = count;             /* save our result */
    return(NULL);                      /* that's it */
}
```

Creating Threads: Example

```
/* allocate arrays for thread IDs and per-thread counts */
pthread_t *threadId = (pthread_t *)(malloc(t*sizeof(pthread_t)));
int *counts = (int *)(malloc(t*sizeof(int)));
int oldHi = 0;

/* start threads: give each n/t values of a to work on */
for(int i = 0; i < t; i++) {
    c3s_arg *arg = (c3s_arg *)(malloc(sizeof(c3s_arg)));
    arg->a = a; arg->lo = oldHi;
    arg->hi = (((long long int)(n))*(i+1))/t;
    arg->count = &(counts[i]);
    if(pthread_create(&threadId[i], NULL, c3s_thread, arg) != 0) {
        perror("count 3's: ");
        exit(-1);
    }
    oldHi = arg->hi;
}
```

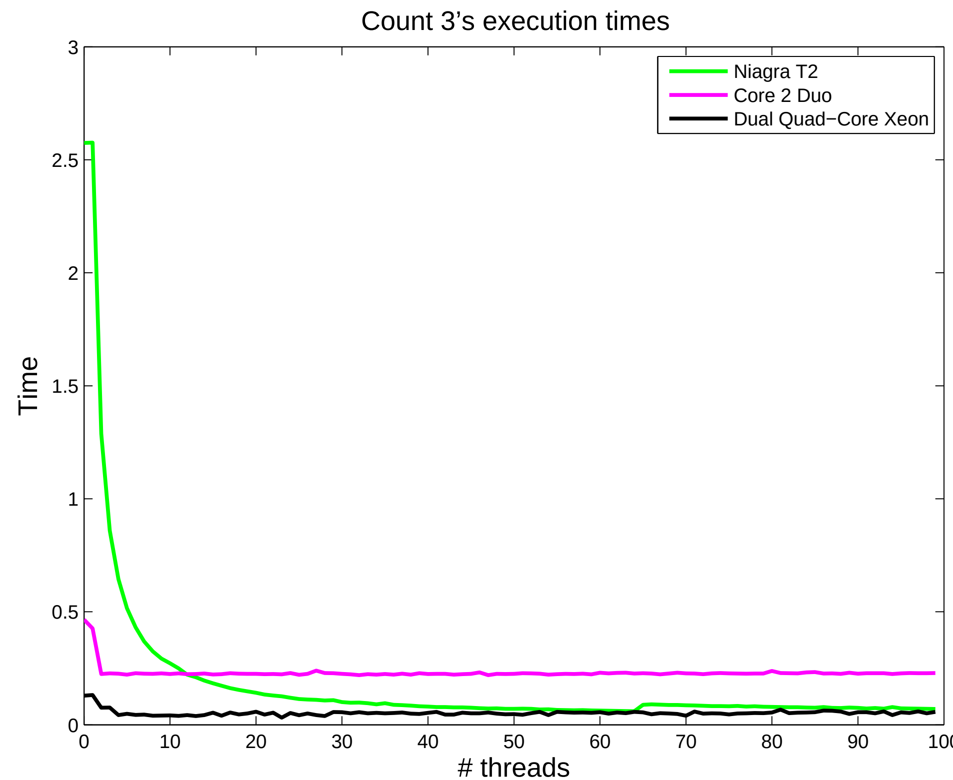
Reaping Threads

- pthread_join(threadId, void **status)
  - threadId: a pointer to a pthread_t.
    Thread join waits until the thread corresponding to threadId exits.
  - status: The exiting thread can pass a pointer back to it's parent with this. If status == NULL, then the exit value is ignored.
- pthread_exit(void *status)
  - or, the thread's top-level function can return –
    What's *status then?

Reaping Threads: Example

```
/* wait for all threads to finish */
for(i = 0; i ⊴ t; i++) {
    if(pthread_join(threadId[i], NULL) != 0) {
        perror("count 3's: ");
        exit(-2);
    }
    n3s += counts[i];
}
return(n3s);
```

Count 3’s: runtime



CPU	# cores	min. time
SUN Niagra T2	8 cores	0.0601 (64 threads)
Intel Core 2 Duo	2 cores	0.2195s (47 threads)
Intel Xeon	8 cores	0.0315s (23 threads)

Communication and Synchronization

- Example: Dekker's algorithm
- Shared Memory
- Mutexes
- Condition Variables
- Barriers

Dekker's Algorithm

Problem statement: ensure that at most one thread is in its critical section at any given time.

thread 0:

```
flag[0] = true;
while(flag[1]) {
    if(turn != 0) {
        flag[0] = false;
        while(turn != 0);
        flag[0] = true;
    }
}
critical section
turn = 1;
flag[0] = false;
```

thread 1:

```
flag[1] = true;
while(flag[0]) {
    if(turn != 1) {
        flag[1] = false;
        while(turn != 1);
        flag[1] = true;
    }
}
critical section
turn = 0;
flag[1] = false;
```

Dekker's with C-threads

```
typedef struct { /* thread parameters */
   int id, ntrials;
} dekker_args;

/* shared variables */
int flag[] = 0,0;
int count[] = 0,0;
int turn = 0;

int dekker_thread(void *void_arg) {
   ...
   for(int i = 0; i < ntrials; i++) {
      do some work;
      acquire the lock;
      critical section (includes test for inteference);
      release lock;
   }
}
```

Work, then lock

```
/* do a random amount of “work” before critical region */
r = 23*r & 0x3f; /* simple pseudo-random, range = {0 ... 63} */
for(int j = 0; j < r; j++); /* this is “work”? */

/* acquire the lock */
flag[me] = TRUE; /* indicate intention to enter critical region */
while(flag[!me]) {
    if(turn != me) {
        flag[me] = FALSE; /* give the other thread a chance */
        while(turn != me); /* spin waiting for turn */
        flag[me] = TRUE; /* try again */
    }
}
```

Critical section, then unlock

```
/* critical section */
for(int j = 0; j < 10; j++) {
    count[me] = j;
    /* check_zero reports error and dies if count[!me] != 0 */
    check_zero(count, !me, i);
}
count[me] = 0;

/* release the lock */
turn = !me;
flag[me] = 0;
```

Let's try it

```
% gcc -std=c99 dekker0.c cz.o -o d0
% d0
check_zero failed for trial 8:  a[0] = 1
% d0
check_zero failed for trial 986:  a[1] = 4
% d0
check_zero failed for trial 898:  a[1] = 4
% d0
check_zero failed for trial 10:  a[0] = 1
% ...
```

- What happened?
- Why?

Fixing the bug

```
/* acquire the lock */
flag[me] = TRUE; /* indicate intention to enter critical region */
__asm__("mfence");
while(flag[!me]) {
    if(turn != me) {
        flag[me] = FALSE;/* give the other thread a chance */
        while(turn != me);/* spin waiting for turn */
        flag[me] = TRUE;/* try again */
    __asm__("mfence");
    }
}
```

- Try again:

```
% d1
ok
% d1
ok
% d1
ok
%
```

What's mfence?

- A memory fence.
- Simple version:
  - All loads and stores issued by the processor that executes the `mfence` must complete globally before execution continues beyond the `mfence`.
- `mfence` instructions are expensive
- And in-line assembly code is painful
  - Not portable.
  - Hard to read.
  - Who wants to program in assembly?

Pthreads provides a higher-level API

- Threads communicate using shared memory.
- Mutual exclusion objects, condition variables, and barriers provide synchronization between threads.
- Pthreads functions also perform the necessary memory fences to make sure that the data is consistent between threads.
  - For changes by thread 1 to be guaranteed to be visible to thread 2: both threads must perform a pthreads synchronization action between the writes by thread 1 and the reads by thread 2.
- In other words:
  - All pthreads synchronization operations are ordered according to their logical dependencies:
  - Within a thread, the thread's actions and its pthreads calls are ordered as expected.
  - Example:
    - If thread 1 unlocks a mutex that then allows thread 2 to continue execution,
    - Then all operations performed by thread 1 before the unlock are visible to operations performed by thread 2 after it acquires the lock.

Producer-Consumer

- Problem statement:
  - The producer generates a sequence of data values: $v_1, v_2, \ldots$.
  - The consumer reads this sequence from the producer.
  - If the consumer is ready to read a value and none is available from the producer, then the consumer stalls until the a data value is available.
  - Likewise, we can implement this interface with a fixed-capacity buffer.
    - In this case, if the producer generates a value and there is no empty space available in the buffer, the producer stalls until the value can be written to the buffer.
- We'll look at an implementation using a shared, fixed-sized array as a buffer.

Producer-Consumer: try 1

```
Value buffer[n]; /* shared buffer */
int wptr, rptr; /* indices for current write and read positions */

int next(int i) { /* cyclic successor of i */
   return((i+1) % n);
}

void put(Value v) { /* called by producer */
   if(next(wptr) != rptr) {
      buffer[wptr] = v;
      wptr = next(wptr);
   } else ???
}

Value take() { /* called by consumer */
   if(rptr != wptr) {
      Value v = buffer[rptr];
      rptr = next(rptr);
      return(v);
   } else ???
}
```

Producer-Consumer: try 2

```
void put(Value v) { /* called by producer */
   while(next(wptr) == rptr); /* wait for empty space */
   buffer[wptr] = v;
   wptr = next(wptr);
}
Value take() { /* called by consumer */
   while(rptr == wptr); /* wait for data to arrive */
   Value v = buffer[rptr];
   rptr = next(rptr);
   return(v);
}
```

What’s wrong with this solution?

Condition Variables (try cond-1)

- `wait(cond);` this thread waits until a signal is sent to `cond`.
- `signal(cond);` this thread sends a signal to `cond`.

Producer-Consumer: try 3

```
Cond w_cond, r_cond; /* condition variables */

void put(Value v) { /* called by producer */
   int oldwptr = wptr;
   if(next(wptr) == rptr)
      wait(w_cond);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
}

Value take() { /* called by consumer */
   int oldrptr = rptr;
   if(rptr == wptr)
      wait(r_cond);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   return(v);
}
```

What's wrong with this solution?

Mutex Variables

- `lock(mutex);` this thread acquires a lock on `mutex`.
  - Only one thread can have the lock at a time.
  - If a thread θ_i attempts to lock a mutex that thread θ_j has already locked, then thread θ_i will block.
- `unlock(mutex);` this thread releases its lock on mutex.
  - If one or more threads are blocked trying to lock the mutex, then one of them will acquire the lock.
  - If multiple threads are waiting for the mutex, an arbitrary one gets it.
  - There is no promise or intent of first-come-first-served awarding of the mutex to waiting threads.

Producer-Consumer: try 4

```
Mutex m; /* a mutex variable */

void put(Value v) { /* called by producer */
   int oldwptr = wptr;
   lock(m);
   if(next(wptr) == rptr)
      wait(w_cond);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
   unlock(m);
}

Value take() { /* called by consumer */
   int oldrptr = rptr;
   lock(m);
   if(rptr == wptr)
      wait(r_cond);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   unlock(m);
   return(v);
}
```


What's wrong with this solution?

Condition variables and mutexes

- We need a mutex with each condition variable
  - Otherwise, we can't safely check the wait condition.
- If the thread needs to wait, then the mutex needs to but unlocked after the thread is waiting for the signal.
  - But, if the thread is waiting for a signal, then it's blocked,
  - . . . and it can't do anything.
  - In particular, it can't unlock the mutex.
- Solution: the `wait` function handles the mutex lock:
  - When the thread is suspended, `wait` unlocks the mutex.
  - When the thread is resumed, `wait` relocks the mutex.

Producer-Consumer: final solution

```
void put(Value v) { /* called by producer */
   int oldwptr = wptr;
   lock(m);
   if(next(wptr) == rptr)
      wait(w_cond, lock);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
   unlock(m);
}
Value take() { /* called by consumer */
   int oldrptr = rptr;
   lock(m);
   if(rptr == wptr)
      wait(r_cond, lock);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   unlock(m);
   return(v);
}
```

We could unlock the mutex while updating `buffer`, `rptr`, and `wptr`. Should we?

Mutexes

The mutex type: `pthread_mutex_t`

- declare and initialize a mutex:
  ```
  pthread_mutex_t my_mutex;
  pthread_mutex_init(&my_mutex, NULL);
  ```
- using a mutex:
  - `pthread_mutex_lock(&my_mutex);`
  - `pthread_mutex_unlock(&my_mutex);`
  - `pthread_mutex_trylock(&my_mutex);`
  - `pthread_mutex_destroy(&my_mutex);`
- usage:
  - Typically, a mutex is associated with a shared data structure.
  - A thread acquires the mutex before accessing the data structure.

Condition Variables

The condition variable type: `pthread_cond_t`

- declare and initialize a condition variable:
  ```
  pthread_cond_t my_cond;
  pthread_cond_init(&my_cond, NULL);
  ```
- using a condition:
  - `pthread_cond_wait(&my_cond);`
  - `pthread_cond_signal(&my_cond);`
  - `pthread_cond_broadcast(&my_cond);`
  - `pthread_cond_destroy(&my_cond);`
- condition variables and locks:

Barriers

For more information

- Lin & Snyder, chapter 6.
- https://computing.llnl.gov/tutorials/pthreads