ECE 574 – Cluster Computing
Lecture 8

Vince Weaver

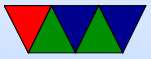
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

20 February 2025

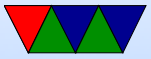
Announcements

- Homework #4 (pthreads) will be posted



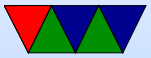
POSIX Threads (1995)

- Various interfaces:
  1. Thread management: Routines for manipulating threads – creating, detaching, joining, etc. Also for setting thread attributes.
  2. Mutexes: (mutual exclusion) – Routines for creating mutex locks.
  3. Condition variables – allow having threads wait on a lock
  4. Synchronization: lock and barrier management



POSIX Threads (pthreads)

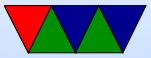
- A C interface. There are wrappers for Fortran.
- Over 100 functions, all starting with pthread_
- Involve "opaque" data structures that are passed around.
- Include pthread.h header
- Include -pthread in linker command to compiler



Pthread Programming

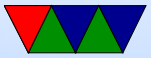
Useful links:

- `https://hpc-tutorials.llnl.gov/posix/`
- `http://www.cs.cf.ac.uk/Dave/C/node31.html`



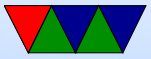
Creating Threads

- Your initial process, as per normal, only includes one thread
- `pthread_create()` creates a new thread
- You can call it anywhere, as many times as you want



pthread_create()

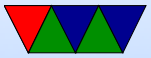
- pthread_create (thread,attr,start_routine,arg)
  - pointer to a thread object (pthread_t) which is opaque
  - an attr object (which can be NULL)
  - a start_routine which is a C function called when it starts
  - an arg argument to pass to the routine.
- Only can pass one argument. How can you pass more? pointer to a structure.
- With attributes you can set things like scheduling policies



Terminating Threads

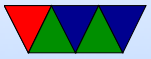
Ways to terminate threads:

- `pthread_exit()`
- Return normally from its starting routine
- another thread uses `pthread_cancel()` on it
- The entire process is terminated (by ending, or calling `exit()`, `exit_group()`, etc)



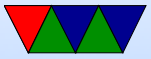
Waiting for Thread Completion

- `pthread_join()` lets a thread block until another one finishes
- The main thread can join all the children and wait until they are done before continuing.
- Argument to a join is a specific thread to wait on (so if waiting on four, have to have four calls to `pthread_join()`



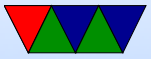
Returning Values from a Thread

- Put the result in the arg struct passed in at start
- Use the second parameter to `pthread_join()` which is a pointer to a void pointer (that gets tricky in C). This is returned by return from thread, and/or the argument to `pthread_exit()`
  - You can cast it to an int to return a single value
  - You can malloc() a struct and return a pointer to that
  - NOTE: you can't return a pointer to a local struct in the thread, as the stack will be destroyed on exit



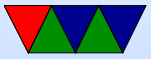
Stack Management

- Manage your own stack? Can get and set size.
- Be careful allocating too much on stack.
  Will you run out of space? OS has things like over-commit that make this less likely
- Too little stack can be issue if lots of local vars



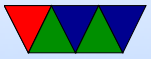
Binding threads to Cores

- Can you pick which core a thread runs on?
- Usually you can trust OS to do an OK job
- `pthread_setaffinity_np()` added recently
- Based on Linux `sched_setaffinity()` routine.
- Can also do it via command line (taskset is one way)



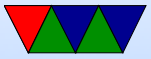
Mutexes

- Type of lock, only one thread can own it at a time.
- Can be used to avoid race conditions.



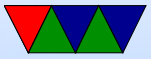
Condition Variables

- A way to avoid spinning on a mutex
- Threads can queue up waiting for lock, then be restarted once lock is freed



Example code

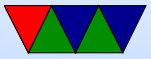
Example code is posted on course website.



Simple Pthread Example

See `pthread_simple.c`

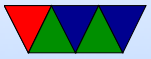
- Hard codes 10 threads
- Do they run in any specific order?



Simple Init Example

See `pthread_init.c`.

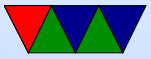
- Initializes 256MB of data. Number of threads from command line.
  Is this the most efficient way to init memory?
- Why do we have the sleep call? Note: you'd never want to write a real program using a sleep like that.
- Why errors if run on odd number?
  Be sure when splitting up problem handle remainders.



Simple Join Example

See `pthread_join.c`

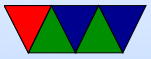
- Can use join to make the master thread wait for the others to finish.
- Second argument is return value, so can find out what thread returned when finished (or error)
- Can only join "joinable" threads (`PTHREAD_CREATE_JOINAB`
  By default all threads start out joinable



Return Value Example

See `pthread_return.c`

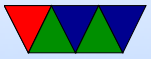
- You can return a value via `pthread_join()`



Stack Example

See `pthread_stack.c`

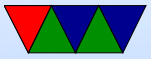
How to see how much stack is available, and how to change it if not enough.



Mutex Example

See `pthread_mutex.c` for code w/o mutex (run with a num greater than 1)
Then see `pthread_mutex2.c` for core w mutex

Creates a "thread pool" and the threads can request more work when they finish.

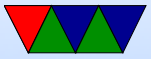


Creating Mutexes

- Can create mutexes two ways,
  - Statically, when declared

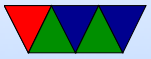
```
pthread_mutex_t our_mutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Dynamically with `pthread_mutex_init()` which allows setting mutex object attributes, attr.
- The mutex is initially unlocked.
- Can specify protocol, priority ceiling, and if it's shared/private.



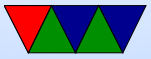
Mutex Actions

- lock – try to take lock, will block if another thread has lock
- unlock – release lock, once released another thread (including any that are waiting) can take lock
- trylock – non-blocking attempt to get lock

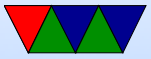


Volatile Variables

- When accessing shared variables is it ever a problem that the compiler or CPU might have the value in a register and not notice memory accesses from other threads?
- In this case should you force memory accesses while using the C `volatile` keyword?
- In theory the answer is no. When using pthreads and using locking the software stack should make sure any atomic accesses or barriers happen when they should
- You will find a lot of people will debate this though



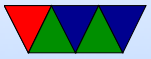
- You may run into problems if you try to open-code locks or condition variables yourself in plain C



Deadlock

When you have more than one lock, it is possible to end up nesting locks in ways that lockup a program with both threads getting stuck.

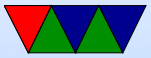
Thread 1	Thread 2
pthread_mutex_lock(&mutex1);	pthread_mutex_lock(&mutex2);
pthread_mutex_lock(&mutex2);	pthread_mutex_lock(&mutex1);



Condition Variable Example?

See `pthread_cond.c`

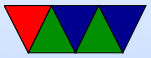
- Can have a thread start up sleeping on a lock, and wake up when signaled by another thread.



PAPI Example

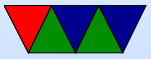
See `pthread_papi.c`

- Do a time example, like in homework 4?
- If using pthreads need to do: `PAPI_thread_init(pthread_self);`
- Will also need to do a `PAPI_register_thread()` in each thread you start



Debugging Threaded Programs

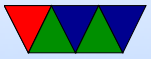
- It can be hard to debug thread and locking issues
- printf can lead to Heisenbugs
- Valgrind can help with locks (Helgrind tool)
- RR deterministic debugger



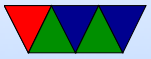
Race Conditions

```
x=0; // times we've run

x=x+1;          x=x+1;

ldr r0,x        ldr r0,x
add r0,#1       add r0,#1
str r0,x        str r0,x
```

- Shared counter address
- RMW on ARM
- Thread A reads value into reg
- Context switch happens
- Thread B reads value into reg, increments, writes out
- Context switch back to A



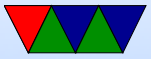
- increments value, writes out
- What happened?
- What should value be?



Critical Sections

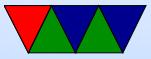
- Want mutual exclusion, only one can access structure at once

1. no two processes can be inside critical section at once
2. no assumption can be made about speed of CPU
3. no process not in critical section may block other processes
4. no process should wait forever



How to avoid

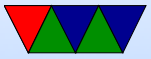
- Disable interrupts. Heavy handed, only works on single-core machines.
- Locks/mutex/semaphore



Mutex

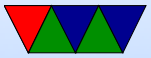
- mutex_lock: if unlocked (0), then it sets lock and returns
  if locked, returns 1, does not enter.
  what do we do if locked? Busy wait? (spinlock) reschedule (yield)?

- mutex_unlock: sets variable to zero



Semaphore

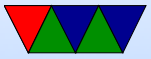
- Up/Down
- Wait in queue
- Blocking
- As lock frees, the job waiting is woken up



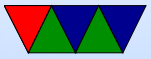
Can you write your own lock in plain C?

```
int lock=0;

try_again:
if (lock==0) {
    lock=1;  // take lock
    ....
    lock=0;  // release lock
} else {
    goto try_again;
}
```

- Problem happens between checking if lock is free and setting it taken. If another task can also pass the check before set, things break. need atomic test-and-set or similar

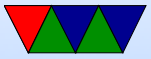


- Another issue is w/o barriers, loads and stores can be re-arranged and accesses can leak into/out-of critical section

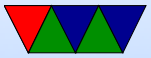


Locking Primitives

- fetch and add (bus lock for multiple cores), xadd (x86)
- test and set (atomically test value and set to 1)
- test and test and set
- compare-and-swap
  - Atomic swap instruction SWP (ARM before v6, deprecated)
  - x86 CMPXCHG
  - Does both load and store in one instruction!
  - Why bad? Longer interrupt latency (can't interrupt



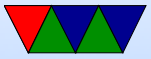
atomic op)
  - Especially bad in multi-core
- load-link/store conditional
  - Load a value from memory
  - Later store instruction to same memory address.
  - Only succeeds if no other stores to that memory location
  - in interim.
  - ldrex/strex (ARMv6 and later)
- Transactional Memory (mostly abandoned)



Locking Primitives

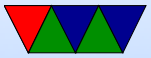
- can be shown to be equivalent

- how swap works:
  lock is 0 (free). r1=1; swap r1,lock
  now r1=0 (was free), lock=1 (in use)
  lock is 1 (not-free). r1=1, swap r1,lock
  now r1=1 (not-free), lock still==1 (in use)



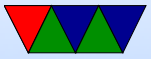
Memory Barriers

- Not a lock, but might be needed when doing locking
- Modern out-of-order processors can execute loads or stores out-of-order
- What happens a load or store bypasses a lock instruction?
- Processor Memory Ordering Models, not fun
- Technically on BCM2835 we need a memory barrier any time we switch between I/O blocks (i.e. from serial



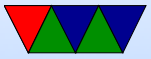
to GPIO, etc.) according to documentation, otherwise loads could return out of order

- Special assembly language instructions



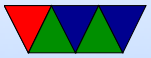
Deadlock

- Two processes both waiting for the other to finish, get stuck
- One possibility is a bad combination of locks, program gets stuck
- P1 takes Lock A. P2 takes Lock B. P1 then tries to take lock B and P2 tries to take Lock A.



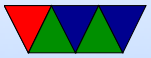
Livelock

- Processes change state, but still no forward progress.
- Two people trying to avoid each other in a hall.
- Can be harder to detect



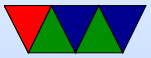
Starvation

- Not really a deadlock, but if there's a minor amount of unfairness in the locking mechanism one process might get "starved" (i.e. never get a chance to run) even though the other processes are properly taking and freeing the locks.



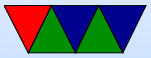
How to avoid Deadlock

- Don't write buggy code
- Pre-emption (let one of the stuck processes run anyway)
- Rollback (checkpoint occasionally)
- What to do if it happens?
  - Reboot the system
  - Kill off stuck processes



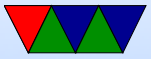
Homework #4 Preview

- We will be parallelizing the code using pthreads



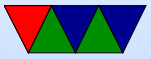
Homework #4 – Coarse Grained

- Before we calculated sobelx and sobely one after the other
- Could we run both at the same time?
- Start two threads, one for sobelx, one for sobely
- Can we launch direct into combine when one finishes? No, have to wait for both to finish first
- What is the max parallelism you can get here?

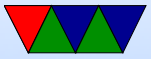


Homework #4 – Fine Grained

- Can we get more fine grained?
- Each pixel in sobel is independent
- We can split things up, if we have 16 threads, give each 1/16 of the sobel array to work on
- The hard part ends up being splitting up the work
- Be careful, have to remember to fixup at the end if not evenly divisible
- Can parallelize combine as well
- Could you start the combine on parts already done while

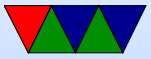


still finishing x and y? Yes, but the complexity of that might not be worth it in the end.



Homework #4 – Dividing up the Work

- This is the hard part, there are a variety of ways to do it
- The way I suggest in this case is splitting up the array into chunks. So if the image has 1024 rows and you are running with 8 threads, then start each thread and give it (1024/8) rows to work with
- Modify your sobel routine to take a start/end row
  Have your Y loop run from start to end rather than 0..ysize
- To calculate start for thread t it would be something



like:
start=(1024/8)*t; end=start+(1024/8)-1;

- This works for evenly divisible images. If it's not, the easiest way is to just set the end of the last chunk to be the total ysize rather than what you'd calculate.
- Note, with pthreads each thread needs its own copy of the command line structure. Otherwise since it's global state if you re-use it you'll have a race. The proper way to do this is use calloc() (see the example code `pthread_join.c` presented in class)