CSE 451: Operating Systems

Section 7

Data races, thread pools, project 2b

Debugging threaded programs

- `printf` is useful, but it takes time to execute—why is this potentially a problem when writing multithreaded programs?
- GDB is pthreads-aware and supports inspecting the state of running threads
  - See this site for a tutorial on interacting with threads from GDB
- If your program is crashing and you don't know why, use `ulimit -c unlimited` to have all crashing programs produce core dumps
  - Then load the core in GDB with `gdb binary core-file`

Data races

- A data race is when two threads read/write the same data concurrently
  - The C standard does not make guarantees about the state of data if there are concurrent reads/writes of it
- Some of you had data races in your test-burgers program—not good!
- Solution: protect concurrent accesses to data using a mutex

Detecting data races

- Valgrind has a tool called helgrind for detecting data races
  - Usage: `valgrind --tool=helgrind ./binary`
  - See the helgrind manual for more information
- Beyond data races, helgrind and other tools will check for problems such as:
  - Exiting a thread that holds a mutex
  - Acquiring locks in inconsistent orderings
  - Waiting on a condition variable without having acquired the corresponding mutex
  - …and many others

Thread pools

- Thread pools provide the illusion of an unlimited amount of parallel processing power, despite using a small number of threads

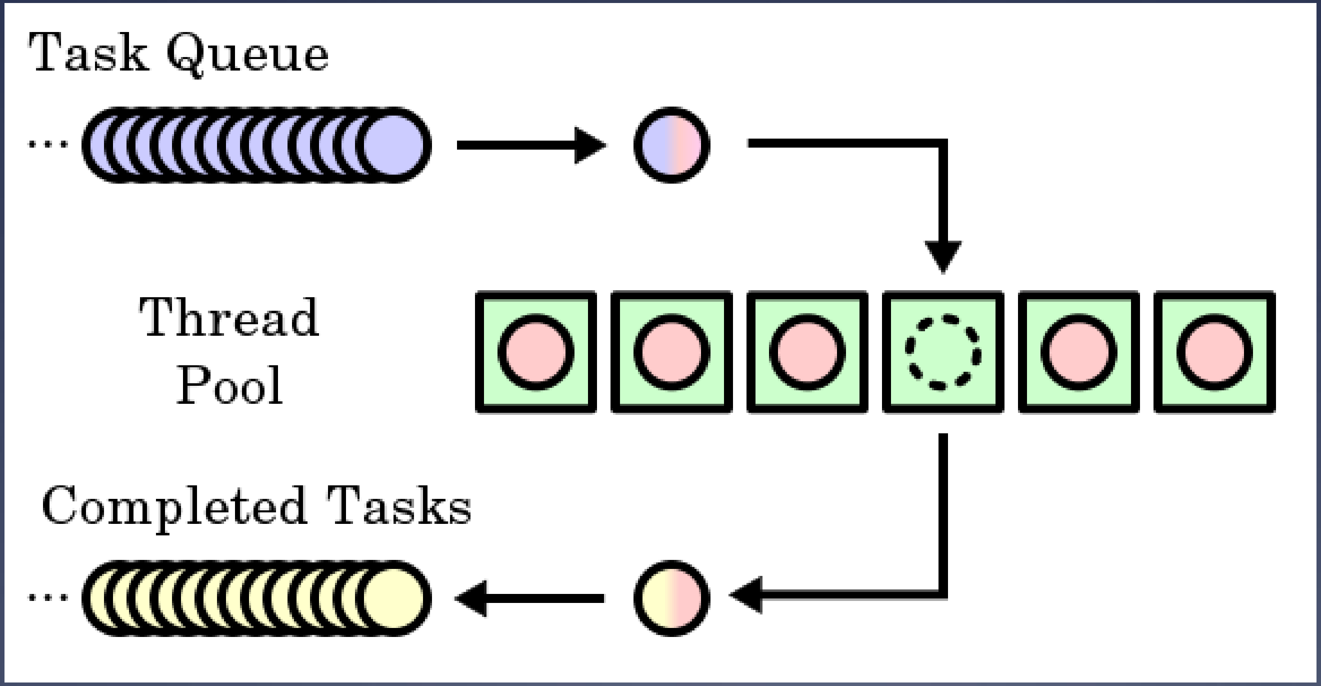

*Diagram from Wikipedia

Thread pools

* Whenever there is a new task to run, a thread from the pool processes it and then fetches the next task from the queue

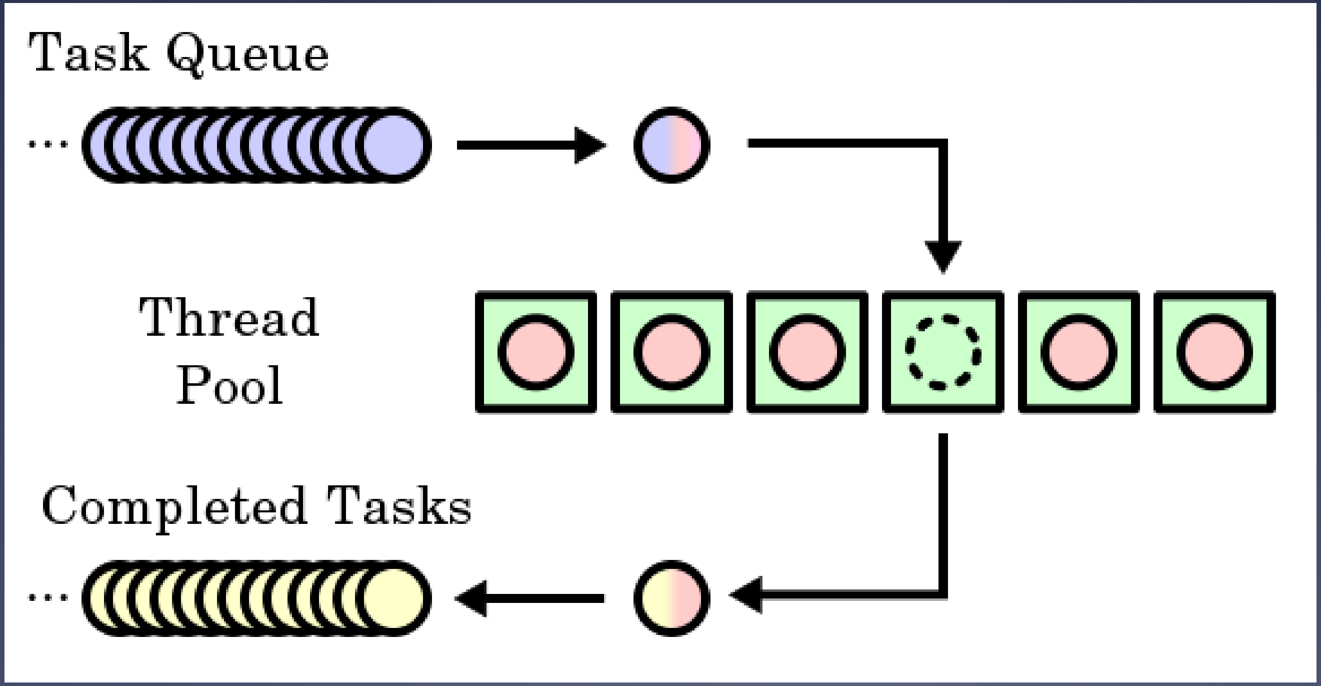

*Diagram from [Wikipedia]

Thread pool implications

- Thread pools only *simulate* an infinite number of processing threads
  - Deadlocks can occur if running threads are blocked waiting for a task that hasn't started
  - For example: launching both producers and consumers from a shared thread pool (why?)
- Thread pools save on the cost of spinning up new threads—workers are recycled

sioux thread pool

```
typedef struct {
   queue request_queue;
   sthread_cond_t request_ready;
} thread_pool;

typedef struct {
   int next_conn;
} request;


// New request arrives:
//   enqueue request, signal request_ready
// Worker threads:
//   dequeue, run handle_request(request);
```

sioux thread pool problems

- This sounds good, but what happens if the request queue grows faster than threads can process the requests?
  - Hint: it's okay to have incoming connections wait (and potentially time out) before you `accept()` them if your server is overloaded
  - The OS enforces a limit on the number of unhandled incoming connections for you—the `BACKLOG` macro in sioux_run.c determines how many

Thread pool performance

- Threads can run on separate CPU cores, but thread pool state is centralized
- Taking a work item involves locking a shared mutex, creating a central point of contention
  - If work items are quick to process, the cost of acquiring the mutex can outweigh the cost of processing the work item!
- If we know approximately how long work items take, how can we improve performance?

Thread pool performance

* Partitioning: divide work items among threads as they arrive
  * Can use a fixed scheme (simple but potentially unbalanced) or a dynamic scheme (more complex but better balanced) to distribute items
* Work stealing: threads that finish processing items in their queues steal work from other threads' queues
  * Work stealing comes up in all manner of distributed settings

Project 2b: part 4

- Make the sioux web server multithreaded
- Create a thread pool (preferrably in a separate thread_pool.[c|h])
- Use the existing connection handling code in cooperation with your thread pool
- Test using pthreads—we won't test against your sthreads implementation
- Apache Bench (`ab`) is a useful tool for measuring webserver performance, more so than the provided `webclient` tool

Project 2b: part 5

- Add preemption to the sthreads library
- One way to think about preemption safety:
  - Disable interrupts in “library” context
  - Use atomic locking in “application” context
- Does locking and unlocking a mutex occur in “library” context or “application” context?

How *not* to implement mutexes

```
sthread_user_mutex_lock(mutex)
   splx(HIGH);  // disable interrupts
   if (mutex->held) {
      enqueue(mutex->queue, current_thread);
      schedule_next_thread();
   } else {
      mutex->held = true;
   }
   splx(LOW);  // reenable interrupts
}
```

- What’s the problem here?

How *not* to implement mutexes

```
sthread_user_mutex_lock(mutex) {
  while(
      atomic_test_and_set(
           &mutex->available)) { }
}
```

- What's the problem here?

How *not* to implement mutexes

```
sthread_user_mutex_lock(mutex) {
  while(
      atomic_test_and_set(
          &mutex->available)) {
      enqueue(mutex->queue, current_thread);
      schedule_next_thread();
   }
}
```

- What's the problem here? Hint: think about preemption

How *to* implement mutexes

- Need to lock around the critical sections in the mutex functions themselves!
  - Your `struct _sthread_mutex` will likely need another member for this
- For hints, re-read lecture slides:
  - Module 7: Synchronization (slide 21 forward)
  - Module 8: Semaphores
- Similar hints apply for condition variables

Project 2b: part 6

* Writeup about webserver and thread library
* Be thorough! Make use of graphs for comparisons and provide commentary on why the results are the way they are
* As mentioned previously, the Apache Bench (`ab`) tool might be useful here as well

Disk buffers

* Both the operating system and physical disks themselves cache reads and writes
* The disk buffer is ~8-128MB on disk, while the page cache is all unused RAM (on the order of gigabytes!)
* Why bother with such a "low" amount on disk?
  * Writes often come in bursts, so this allows for saturating the speeds of both the I/O interface and the speed of physical transfer to disk
  * The OS doesn't have to care about optimizing write order for every vendor's specific hardware
  * Other thoughts?

Asynchronous IO

* Two ways of performing concurrent IO:
  * Multithreaded synchronous operations (e.g. the sioux webserver)
  * Single-threaded asynchronous operations (e.g. ???)
* How does asynchronous IO work?
  * Ask for IO to occur
  * Do some other work (potentially more IO)
  * Wait for IO to complete

Asynchronous IO

- Open files/sockets/etc. with the `O_ASYNC` flag, then use `select()` to wait until one or more file descriptors will accept a `read()` or `write()` without blocking
  - General design: loop continuously, waiting until one or more sources is ready for more processing
- POSIX also provides a set of `aio_*` functions (see `man 7 aio`) such as `aio_read` and `aio_write` to perform asynchronous IO, but these are less commonly used

Asynchronous IO

- What are the advantages and disadvantages of asynchronous IO versus synchronous IO?
- How could asynchrous IO be applied to the sioux webserver?
- Asynchronous IO can be used for event-driven programming
  - Event callbacks (e.g. button presses) in Java's AWT
  - AJAX in JavaScript

Faking record access

* What!? Ed said Unix filesystems don't allow for record access (module 15).
* "We only get `read(), write(), seek(), etc()`."
* MMAP to the rescue!
  * Map a file into memory.
  * Cast pointers to your favorite struct and act as though the file is an array of `struct awesome`.
  * Or treat as linked list or your favorite data structure.
  * Profit.