CSE 451: Operating Systems

Section 7
Address Translations, thread pools, project 2b
Midterm Grade Distribution

[Bar graph showing grade distribution with counts for grades ranging from 21 to 54]
# Midterm Grade Distribution

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>42.76</td>
</tr>
<tr>
<td>Median</td>
<td>43.0</td>
</tr>
<tr>
<td>Mode</td>
<td>41.0</td>
</tr>
<tr>
<td>Min</td>
<td>21.0</td>
</tr>
<tr>
<td>Max</td>
<td>54.0</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>6.85</td>
</tr>
</tbody>
</table>
Address Translation

- Address translation in itself is not Virtual Memory, it is taking an address and mapping to another
  - Virtual Memory is the by-product

- What does having Address Translations allow us to do?

- What are the downsides to Address Translations?
Address Translation

Processor -> Virtual Address -> Translation

Valid -> Physical Address -> Physical Memory

Invalid -> Exception

What all lives in here?
Address Translation

Processor → V.A. → TLB → V.A. (miss) → Page Table

Frame

Virtual Address

Page # | Offset

Physical Memory → Word
Address Translation

Processor → TLB

V.A. → V.A.

hit → Page Table

miss → Page Table

Frame → Frame

Offset → Offset

Virtual Address

Page #1 Page #2 Page #3 Offset

Physical Memory

Word

Phys Mem

Frame 0

Frame 1

…

Frame N

Offset
When Caching translations, we must worry about the consistency of our data

How do we deal with:

* Process Context Switching?
* Permission Change on a physical page?
* Multiprocessor machines, multiple kernel threads for a process? (Two TLBs talking about the same page tables)
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
- 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 provide the illusion of an unlimited amount of parallel processing power, despite using a small number of threads.
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
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 (preferably 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
Add preemption to the stthreads 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

```c
stthread_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

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

🔍 What’s the problem here?
How not to implement mutexes

```c
stthread_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 __stthread_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
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