Threads

Motivation: are processes all we need for useful concurrency?

Threads: Concurrency with shared memory

https://cs.wellesley.edu/~cs240/
Why do we need concurrency?

42 Years of Microprocessor Trend Data

Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten.

New plot and data collected for 2010-2017 by K. Rupp
Advantages/disadvantages of concurrent programs

**Advantages**
- More responsive
  - Interacting with IO
- Higher performance
  - Computers have multiple cores
  - Make progress when one task waits

**Disadvantages**
- New kinds of bugs
  - Race conditions
  - Deadlock
- Much more difficult to test, debug
Recall: processes create *private copies* of program state

Why might we want *shared access* to program state?
Threads: distinct execution, shared memory

- Core idea: allow shared memory, but distinct/concurrent execution

Programs are just data: what data tracks execution?

- Stack
- Heap
- Data
- Code: /usr/bin/bash

Threads need distinct stacks & registers
Threads: distinct execution, shared memory

- OS and languages generally allow processes to run two or more functions simultaneously via threading.
- The stack segment is subdivided into 1 stack per thread.
- The thread manager time slices and between threads.
- Threads often called “lightweight processes”.
- Each thread maintains its own stack, but all threads share the same text, data, and heap segments.
### Processes vs. Threads: what is shared?

<table>
<thead>
<tr>
<th></th>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Not shared (private copies)</td>
<td>Not shared (subdivided)</td>
</tr>
<tr>
<td>Registers</td>
<td>Not shared (kernel tracks)</td>
<td>Not shared (kernel tracks)</td>
</tr>
<tr>
<td>Code (instruction memory)</td>
<td>Shared</td>
<td>Shared</td>
</tr>
<tr>
<td>Heap (dynamic memory)</td>
<td>Not shared (private copies)</td>
<td>Shared</td>
</tr>
</tbody>
</table>

A thread is an independent execution sequence within a single process, with shared dynamic memory.
Processes vs. threads

**Threads**

- Easier coordination, operating on shared data
- Lower communication overhead

**Processes**

- Support for distinct programs/code (exec)
- Built-in memory protection

• Since threads have no memory protection, race conditions and deadlocks more likely
## Race condition

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = x + 1 )</td>
<td>( x = x \times 2 )</td>
</tr>
</tbody>
</table>

Assume \( x = 2 \) before this code runs.

What possible values could \( x \) have after this code runs?
ANSI C doesn't provide native support for threads.

But **pthreads**, which comes with all standard UNIX distributions, provides thread support.

- The primary **pthreads** data type is the **pthread_t**, which is a type used to manage the execution of a function within its own thread of execution.

- The **pthreads** functions we'll need: **pthread_create** and **pthread_join**.
Examine introverts!
Key points of introverts

• Introverts declares an array of six `pthread_t` handles.
• The program initializes each `pthread_t` (via `pthread_create`) by installing `recharge` as the function each `pthread_t` should execute.
• All thread routines take a `void *` and return a `void *`.
• The `pthread` thread manager's attention, and we have very little control over what choices it makes when deciding what thread to run next.
pthread_join waits

- **pthread_join** is to threads what **waitpid** is to processes.

- The main thread of execution blocks until the child threads all exit. The second argument to **pthread_join** can be used to catch a thread routine's return value.

- If we don't care to receive it, we can pass in **NULL** to ignore it.
Sharing data

• Sharing data can be complicated and dangerous in concurrent execution, but often necessary.

• Concurrent programming often makes use of specific tools to control how data is shared between threads
  • Locking/mutexes
  • Semaphores
  • Condition variables
  • Etc.
Examine robberBaronsBroken!
Something is wrong!

• How do we know?
  • Printing is out of order at the end
  • Negative value for the stash?

• Multiple threads are modifying the global variable stash

• Is it possible for two threads to evaluate stash > 0 as True with only $10000 left and then both subtract from stash?
  • Yep! Say thread A evaluates stash > 0 and then the thread manager switches to thread B before thread A subtracts the steal money from the stash.
  • Thread B executes fully bringing the stash to $0.
  • Thread A resumes execution and subtracts its $10000 bringing the total to -$10000.
  • Yikes!
A mutex is a **mutual exclusion** object.

It is a *locking* mechanism to protect shared data or critical regions of code so that only one thread can be permitted access.

Here: protect the stash so that only one robber can modify it at a given time.

We declare a mutex with `pthread_mutex_t`.

To lock a piece of code, we use `pthread_mutex_lock()`.

- When a thread tries to acquire a lock, it will either take the lock if it is not being currently used or it will wait until the lock becomes available.

To unlock a piece of code, we use `pthread_mutex_unlock()`.

- Only the thread that holds a lock can unlock it.
Examine robberBarons!