Parallel Systems
Programming Models: Processes + Threads

Chris Rossbach + Calvin Lin
CS380p
Outline for Today

• Parallel programming models
  • Processes
  • Threads
  • Fibers
  • pthreads

Acknowledgments: some materials in this lecture borrowed from or built on materials from:
  • Emmett Witchel, who borrowed them from: Kathryn McKinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger
  • Mark Silberstein, who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
  ...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
    machine.Registers[i.dst] += machine.Registers[i.src];
    break;
  }
Parallel Machines: a mental model

```c
struct machine_state{
    uint64 pc;
    uint64 Registers[16];
    uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
    ...
} machine;
while(1) {
    fetch_instruction(machine.pc);
    decode_instruction(machine.pc);
    execute_instruction(machine.pc);
}
void execute_instruction(i) {
    switch(opcode) {
    case add_rr:
        machine.Registers[i.dst] += machine.Registers[i.src];
        break;
    }
```

Programming Models for Concurrency
Programming Models for Concurrency

• Concrete model:
  • CPU(s) execute instructions sequentially
Programming Models for Concurrency

• Concrete model:
  • CPU(s) execute instructions sequentially

• Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer
Programming Models for Concurrency

• Concrete model:
  • CPU(s) execute instructions sequentially

• Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Threads/Processes
  • Message passing vs shared memory
  • Preemption vs Non-preemption
Programming Models for Concurrency

• Concrete model:
  • CPU(s) execute instructions sequentially

• Dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Threads/Processes
  • Message passing vs shared memory
  • Preemption vs Non-preemption

• Dimensions/techniques not always orthogonal
Processes and Threads

- Abstractions
- Unit of Allocation/Containment
- State
  - Where is shared state?
  - How is it accessed?
  - Is it mutable?
Processes
The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Uniprocessor: Only one program active at any instant
- Multiprocessor: two run in parallel per quantum
Threads
The Thread Model

(a) Three processes each with one thread
(b) One process with three threads
The Thread Model

<table>
<thead>
<tr>
<th><strong>Per process items</strong></th>
<th><strong>Per thread items</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
# The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread
# The Thread Model

<table>
<thead>
<tr>
<th><strong>Per process items</strong></th>
<th><strong>Per thread items</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td><strong>Decouples memory and control abstractions!</strong></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread

Each thread has its own stack

CS380P  Processes + Threads  9
The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread
- **Decouples memory and control abstractions!**
- **What is the advantage of that?**
The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread
- **Decouples memory and control abstractions**
- **What is the advantage of that?**
The Thread Model

- Items shared by all threads in a process
- Items private to each thread
- **Decouples memory and control abstraction**
- **What is the advantage of that?**

How can we share mutable state across threads?
How can we share mutable state across processes?
Using threads

Ex. How might we use threads in a word processor program?
Using threads

Ex. How might we use threads in a word processor program?
Thread Usage

A multithreaded Web server

(a) Dispatcher thread
(b) Worker thread
Where to Implement Threads:
Where to Implement Threads:

User Space

Kernel Space
Where to Implement Threads:

**User Space**

- Process
- Thread

**Kernel Space**

- User space
- Kernel space
- Run-time system
- Thread table
- Process table

A user-level threads package
Where to Implement Threads:

**User Space**

A user-level threads package

**Kernel Space**

A threads package managed by the kernel
Where to Implement Threads:

**User Space**
- Process
- Thread
- Run-time system
- Thread table
- Process table

**Kernel Space**
- Process
- Thread
- Process table
- Thread table

A user-level threads package
A threads package managed by the kernel

What are some tradeoffs between user/kernel support for threads?
Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State
Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management
Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor
Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor

• Serial
  • One at a time, no conflict
Execution Context Management

“Task” == “Flow of Control”
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor

• Serial
  • One at a time, no conflict

• Cooperative
  • Yields at well-defined points
  • E.g. wait for long-running I/O
Execution Context Management

"Task" == "Flow of Control"
"Stack" == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor

• Serial
  • One at a time, no conflict

• Cooperative
  • Yields at well-defined points
  • E.g. wait for long-running I/O

Stack Management

• Manual
  • Inherent in Cooperative
  • Changing at quiescent points

• Automatic
  • Inherent in pre-emptive
  • Downside: Hidden concurrency assumptions
Fibers
Fibers

• Cooperative tasks
  • most desirable when reasoning about concurrency
  • usually associated with event-driven programming
Fibers

• Cooperative tasks
  • most desirable when reasoning about concurrency
  • usually associated with event-driven programming

• Automatic stack management
  • most desirable when reading/maintaining code
  • Usually associated with threaded (or serial) programming
Fibers

• Cooperative tasks
  • most desirable when reasoning about concurrency
  • usually associated with event-driven programming

• Automatic stack management
  • most desirable when reading/maintaining code
  • Usually associated with threaded (or serial) programming
Threads vs Fibers
Threads vs Fibers

• Like threads, *just an abstraction* for flow of control
Threads vs Fibers

• Like threads, *just an abstraction* for flow of control

• *Lighter weight* than threads
  • In Windows, just a stack, subset of arch. registers, non-preemptive
  • stack management/impl has interplay with exceptions
  • Can be completely exception safe
Threads vs Fibers

• Like threads, *just an abstraction* for flow of control
• *Lighter weight* than threads
  • In Windows, just a stack, subset of arch. registers, non-preemptive
  • stack management/impl has interplay with exceptions
  • Can be completely exception safe
• *Takeaway*: diversity of abstractions/containers for execution flows
x86_64 Architectural Registers

- Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525
Linux x86_64 context switch excerpt

Complete fiber context switch on Unix and Windows
x86_64 Registers and Threads

Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525
x86_64 Registers and Threads

- Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525
x86_64 Registers and Fibers

- Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525
x86_64 Registers and Fibers

- Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525
x86_64 Registers and Fibers

The takeaway:
- Many abstractions for flows of control
- Different tradeoffs in overhead, flexibility
- Matters for concurrency: exercised heavily

- Register map diagram courtesy of: By Immae - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32745525
Pthreads

• POSIX standard thread model,
• Specifies the API and call semantics.
• Popular – most thread libraries are Pthreads-compatible
Preliminaries

• Include `pthread.h` in the main file
• Compile program with `-lpthread`
  • `gcc -o test test.c -lpthread`
  • may not report compilation errors otherwise but calls will fail
• Good idea to check return values on common functions
Thread creation

- **Types:** `pthread_t` – type of a thread

- **Some calls:**
  ```c
  int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void *),
                   void *arg);
  int pthread_join(pthread_t thread, void **status);
  int pthread_detach();
  void pthread_exit();
  ```

- No explicit parent/child model, except main thread holds process info
- Call `pthread_exit` in main, don’t just fall through;
- Don’t always need `pthread_join`
  - `status` = exit value returned by joinable thread
- Detached threads are those which cannot be joined (can also set this at creation)
Creating multiple threads

```c
#include <stdio.h>
#include <pthread.h>
define NUM_THREADS 4

void *hello (void *arg) {
    printf("Hello Thread\n");
}

main() {
    pthread_t tid[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, hello, NULL);

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}```
Can you find the bug here?

What is printed for myNum?

```c
void *threadFunc(void *pArg) {
    int* p = (int*)pArg;
    int myNum = *p;
    printf( "Thread number %d\n", myNum);
}

// from main():
for (int i = 0; i < numThreads; i++) {
    pthread_create(&tid[i], NULL, threadFunc, &i);
}
Pthread Mutexes

• Type: `pthread_mutex_t`

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

• Attributes: for shared mutexes/condition vars among processes, for priority inheritance, etc.
  • use defaults

• Important: Mutex scope must be visible to all threads!
Pthread Spinlock
Pthread Spinlock

• **Type:** `pthread_spinlock_t`
Pthread Spinlock

• Type: `pthread_spinlock_t`

```c
int pthread_spinlock_init(pthread_spinlock_t *lock);
```
Pthread Spinlock

• **Type:** `pthread_spinlock_t`

```c
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
```
Pthread Spinlock

• **Type:** `pthread_spinlock_t`

```c
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
```
Pthread Spinlock

• Type: `pthread_spinlock_t`

```c
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```
Pthread Spinlock

• **Type:** `pthread_spinlock_t`

```c
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```
Pthread Spinlock

- **Type:** `pthread_spinlock_t`

```c
int pthread_spinlock_init(pthread_spinlock_t *lock);
int pthread_spinlock_destroy(pthread_spinlock_t *lock);
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Wait...what’s the difference?

```c
int pthread_mutex_init(pthread_mutex_t *mutex,...);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```
Lab #1

• Basic synchronization, prefix sum
• http://www.cs.utexas.edu/~rossbach/cs380p/lab/lab1.html

• Start early!!!