Foundations:
Synchronization
Execution Abstractions
Cache Coherence

Chris Rossbach
CS378H Fall 2018
9/10/18
Today

• Questions?
• Administrivia
  • Lab 1 due Wednesday by 11am!
• Foundations
  • Basic Synchronization
  • Threads/Processes/Fibers
  • Cache coherence (maybe)

• Acknowledgments: some materials in this lecture borrowed 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)
  • Andy Tannenbaum
Faux Quiz  (answer any 2, 5 min)

• What is the maximum possible speedup of a 75% parallelizable program on 8 CPUs
• What is super-linear speedup? List two ways in which super-linear speedup can occur.
• What is the difference between strong and weak scaling?
• Define Safety, Liveness, Bounded Waiting, Failure Atomicity
• What is the difference between processes and threads?
• What’s a fiber? When and why might fibers be a better abstraction than threads?
Concurrency and Correctness

If two threads execute this program concurrently, how many different final values of X are there?

Initially, \( X == 0 \).

Thread 1

```c
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Thread 2

```c
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Answer:
A. 0  
B. 1  
C. 2  
D. More than 2
Schedules/Interleavings

Model of concurrent execution

• Interleave statements from each thread into a single thread
• If any interleaving yields incorrect results, synchronization is needed

If $X=0$ initially, $X=1$ at the end. WRONG result!
Locks fix this with Mutual Exclusion

Mutual exclusion ensures only safe interleavings

• *But it limits concurrency, and hence scalability/performance*

Is mutual exclusion a good abstraction?
Why are Locks “Hard?”

• Coarse-grain locks
  • Simple to develop
  • Easy to avoid deadlock
  • Few data races
  • Limited concurrency

• Fine-grain locks
  • Greater concurrency
  • Greater code complexity
  • Potential deadlocks
    • Not composable
  • Potential data races
    • Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}

DEADLOCK!

Thread 0
move(a, b, key1);
move(b, a, key2);

Thread 1

Correctness conditions

- Safety
  - Only one thread in the critical region

- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.

- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

Model:

```java
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

Did we get all the important conditions? 

*Hint: Anyone try last step of Lab 1 yet?*
struct machine_state{
    uint64 pc;
    uint64 Registers[16];
    uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(i) {
    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
Processes and Threads and Fibers...

- Abstractions
- Containers
- State
  - Where is shared state?
  - How is it accessed?
  - Is it mutable?
Processes

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

**Model**

**Implementation**

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>Prior to process started</td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td>CPU time used</td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td>Children’s CPU time</td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td>Time of next alarm</td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Process Address Space

- **P1**
- **0**
- **C0000000**
- **C0400000**
- **FFFFFFFFFF**

**accessible**
- **kcode**
- **kdata**
- **kbss**
- **kheap**

**related**
- **udata (1)**
- **ucode (1)**

**access**
- Possible in user mode
- Requires kernel mode

**Why relevant?**
State is shared through memory!

**Q:** How to share data across processes?

**Anyone see another issue?**

**Anyone else see another issue?**
Abstractions for Concurrency

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

When might (a) be better than (b)? Vice versa?
Could you do lab 1 with processes instead of threads?
Threads simplify sharing and reduce context overheads
The Thread Model

Per process items
- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

Per thread items
- Program counter
- Registers
- Stack
- State

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

Ex. How might we use threads in a word processor program?
Multi-threaded Webserver

A multithreaded Web server

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

**User Space**

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

**Kernel Space**

- Kernel
- 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”, but with less typing

“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

These dimensions can be orthogonal
Fibers: the Sweet Spot?

• 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

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

• *Lighter weight* than threads
  • In Windows, just a stack, subset of arch. registers, non-preemptive
  • *Not* just threads without exception support
  • 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

| ZMM0 | YMM0 | XMM0 | ZMM1 | YMM1 | XMM1 | ST(0) | MM0 | ST(1) | MM1 | AX | EAX | RAX | R12 | R11 | R8  | R9  | R10 | R13 | R14 | R15 |
|------|------|------|------|------|------|-------|-----|-------|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| ZMM2 | YMM2 | XMM2 | ZMM3 | YMM3 | XMM3 | ST(2) | MM2 | ST(3) | MM3 | BX | EBX | RBX | R12 | R11 | R9  | R10 | R11 | R13 | R14 | R15 |
| ZMM4 | YMM4 | XMM4 | ZMM5 | YMM5 | XMM5 | ST(4) | MM4 | ST(5) | MM5 | CX | ECX | RCX | R12 | R11 | R9  | R10 | R11 | R13 | R14 | R15 |
| ZMM6 | YMM6 | XMM6 | ZMM7 | YMM7 | XMM7 | ST(6) | MM6 | ST(7) | MM7 | DX | EDX | RDX | R12 | R11 | R9  | R10 | R11 | R13 | R14 | R15 |
| ZMM8 | YMM8 | XMM8 | ZMM9 | YMM9 | XMM9 | CW    | FP_IP| FP_DP | FP_CS| ESP | | | | | | | | | | | | | |
| ZMM10| YMM10| XMM10| ZMM11| YMM11| XMM11| SW    | | | | | | | | | | | | | | | | | | | |
| ZMM12| YMM12| XMM12| ZMM13| YMM13| XMM13| TW    | | | | | | | | | | | | | | | | | | | |
| ZMM14| YMM14| XMM14| ZMM15| YMM15| XMM15| FP_DS | | | | | | | | | | | | | | | | | | | |
| ZMM16| ZMM17| ZMM18| ZMM19| ZMM20| ZMM21| ZMM22| ZMM23| | | | | | | | | | | | | | |
| ZMM24| ZMM25| ZMM26| ZMM27| ZMM28| ZMM29| ZMM30| ZMM31| | | | | | | | | | | | | | |

- 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;
- When do you 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”);
}

int 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

- **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);
```
Review: mutual exclusion model

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

```c
while(1) {
  Entry section
  Critical section
  Exit section
  Non-critical section
}

Mutex, spinlock, etc. are ways to implement these.
Implementing Locks

```c
int lock_value = 0;
ext* lock = &lock_value;

Lock::Acquire() {
    while (*lock == 1)
        ; //spin
    *lock = 1;
}

Lock::Release() {
    *lock = 0;
}
```

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work
Multiprocessor Cache Coherence

Physics  |  Concurrency

\[ F = ma \sim \text{coherence} \]
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
- P2: X++
- P3: read X
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic
Cache Coherence: single-thread

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try: load lock, R0
    test R0
    bnz try
    store lock, 1
}
Cache Coherence Action Zone

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try:  load lock, R0
    test R0
    bnz try
    store lock, 1
}

SAFE!
Cache Coherence Action Zone II

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}
Read-Modify-Write (RMW)

- Implementing locks requires read-modify-write operations
- Required effect is:
  - An atomic and isolated action
    1. read memory location **AND**
    2. write a new value to the location
  - RMW is *very tricky* in multi-processors
  - Cache coherence alone doesn’t solve it

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  load lock, R0
        test R0
        bnz try
        store lock, 1
}
Essence of HW-supported RMW

```c
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try:  
    load lock, R0
    test R0
    bnz try
    store lock, 1
}
```

Make this into a single (atomic hardware instruction)
# HW Support for Read-Modify-Write (RMW)

<table>
<thead>
<tr>
<th>Test &amp; Set</th>
<th>CAS</th>
<th>Exchange, locked increment/decrement,</th>
<th>LLSC: load-linked store-conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most architectures</td>
<td>Many architectures</td>
<td>x86</td>
<td>PPC, Alpha, MIPS</td>
</tr>
</tbody>
</table>

```c
int TST(addr) {
    atomic {
        ret = *addr;
        if(!*addr)
            *addr = 1;
        return ret;
    }
}
```

```c
bool cas(addr, old, new) {
    atomic {
        if(*addr == old) {
            *addr = new;
            return true;
        }
        return false;
    }
}
```

```c
int XCHG(addr, val) {
    atomic {
        ret = *addr;
        *addr = val;
        return ret;
    }
}
```

```c
bool LLSC(addr, val) {
    ret = *addr;
    atomic {
        if(*addr == ret) {
            *addr = val;
            return true;
        }
        return false;
    }
}
```

```c
void CAS_lock(lock) {
    while(CAS(&lock, 0, 1) != true);
}
```
HW Support for RMW: LL-SC

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged

```c
void LLSC_lock(lock) {
    while(1) {
        old = load-linked(lock);
        if(old == 0 && store-cond(lock, 1))
            return;
    }
}
```
LLSC Lock Action Zone

P1

lock: 0

P2

lock: 1

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}

P2

lock(lock) {
    while(1) {
        old = ll(lock);
        if(old == 0)
            if(sc(lock, 1))
                return;
    }
}
LLSC Lock Action Zone II

P1

lock

lock: \texttt{[SM]} 0

lock: 0

P2

lock

lock: \texttt{[SL]} 0

lock: 0

lock(lock) {
  while(1) {
    old = \texttt{ll(lock)};
    if(old == 0)
      if(sc(lock, 1))
        if(sc(lock, 1))
          return;
  }
}

lock(lock) {
  while(1) {
    old = \texttt{ll(lock)};
    if(old == 0)
      if(sc(lock, 1))
        if(sc(lock, 1))
          return;
  }
}

Store conditional fails
Implementing Locks with Test&set

```cpp
int lock_value = 0;
int* lock = &lock_value;

Lock::Acquire() {
    while (test&set(lock) == 1)  //spin
}

Lock::Release() {
    *lock = 0;
}
```

What is the problem with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work
Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting

What happens to lock variable’s cache line when different cpu’s contend?

- Load can stall
- With bus-locking, lock prefix blocks *everyone*
- With CAS, LL-SC, cache line cache line “ping pongs” amongst contenders
TTS: Reducing busy wait contention

Test&Set

Lock::Acquire() {
    while (test&set(lock) == 1);
}

Lock::Release() {
    *lock = 0;
}

Busy-wait on in-memory copy

Test&Test&Set

Lock::Acquire() {
    while(1) {
        while (*lock == 1); // spin just reading
        if (test&set(lock) == 0) break;
    }
}

Lock::Release() {
    *lock = 0;
}

Busy-wait on cached copy

• What is the problem with this?
  • A. CPU usage  B. Memory usage  C. Lock::Acquire() latency
  • D. Memory bus usage  E. Does not work
Test & Test & Set with Memory Hierarchies

What happens to lock variable’s cache line when different cpu’s contend for the same lock?
What happens to lock variable’s cache line when different cpu’s contend for the same lock?
How can we improve over busy-wait?

```cpp
Lock::Acquire() {
    while(1) {
        while (*lock == 1); // spin just reading
        if (test&set(lock) == 0) break;
    }
}
```
Mutex

• Same abstraction as spinlock
• But is a “blocking” primitive
  • Lock available → same behavior
  • Lock held → yield/block
• Many ways to yield
• Simplest case of semaphore

```c
void cm3_lock(u8_t* M) {
    u8_t LockedIn = 0;
    do {
        if (__LDREXB(Mutex) == 0) {
            // unlocked: try to obtain lock
            if (__STREXB(1, Mutex)) {
                // got lock
                __CLREXB(); // remove __LDREXB() lock
                LockedIn = 1;
            } else task_yield(); // give away cpu
        } else task_yield(); // give away cpu
    } while(!LockedIn);
}
```

• Is it better to use a spinlock or mutex on a uni-processor?
• Is it better to use a spinlock or mutex on a multi-processor?
• How do you choose between spinlock/mutex on a multi-processor?
Priority Inversion

A(prio-0) → enter(l);
B(prio-100) → enter(l); → must wait.

Solution?

**Priority inheritance**: A runs at B’s priority
MARS pathfinder failure:

Other ideas?
Dekker’s Algorithm

variables
wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] = false
wants_to_enter[1] = false
turn = 0    // or 1

p0:
    wants_to_enter[0] = true
while wants_to_enter[1] {
    if turn = 0 {
        wants_to_enter[0] = false
        while turn = 0 {
            // busy wait
        }
        wants_to_enter[0] = true
    }
    // critical section
    ...
    turn = 1
    wants_to_enter[0] = false
    // remainder section
}

p1:
    wants_to_enter[1] = true
while wants_to_enter[0] {
    if turn = 1 {
        wants_to_enter[1] = false
        while turn = 1 {
            // busy wait
        }
        wants_to_enter[1] = true
    }
    // critical section
    ...
    turn = 0
    wants_to_enter[1] = false
    // remainder section

initially: c1, c2, turn = 1, 1, 1

Th.J. Dekker’s Solution
Lab #1

• Basic synchronization

• http://www.cs.utexas.edu/~rossbach/cs378/lab/lab0.html

• *Start early!!! Well...too late for that. Start soon!*
Questions?