Foundations: Synchronization Execution Abstractions

Chris Rossbach
CS378H Fall 2018
9/10/18
Today

• Questions?
• Administrivia
  • Lab 1 due sooner than you’d like
• Foundations
  • 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?
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?
Processes and Threads and Fibers...

- Abstractions
- Containers
- State
  - Where is shared state?
  - How is it accessed?
  - Is it mutable?
Programming and 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;
    }
```
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;
}
Processes

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

**Model**

**Process management**
- Registers
- Program counter
- Program status word
- Stack pointer
- Process state
- Priority
- Scheduling parameters
- Process ID
- Parent process
- Process group
- Signals
- Time when process started
- CPU time used
- Children's CPU time
- Time of next alarm

**Memory management**
- Pointer to text segment
- Pointer to data segment
- Pointer to stack segment

**File management**
- Root directory
- Working directory
- File descriptors
- User ID
- Group ID
Process Address Space

access requires kernel mode

access possible in user mode

Why relevant?
State is shared through memory!

Q: How to share data across processes?

Anyone 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

<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?*
Where to Implement Threads:

**User Space**

- Process
- Thread

**Kernel Space**

- Process
- Thread

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

A user-level threads package

A threads package managed by the kernel
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

| ZMM0 | YMM0 | XMM0 | ZMM1 | YMM1 | XMM1 | ZMM2 | YMM2 | XMM2 | ZMM3 | YMM3 | XMM3 | ZMM4 | YMM4 | XMM4 | ZMM5 | YMM5 | XMM5 | ZMM6 | YMM6 | XMM6 | ZMM7 | YMM7 | XMM7 | ZMM8 | YMM8 | XMM8 | ZMM9 | YMM9 | XMM9 | ZMM10 | YMM10 | XMM10 | ZMM11 | YMM11 | XMM11 | ZMM12 | YMM12 | XMM12 | ZMM13 | YMM13 | XMM13 | ZMM14 | YMM14 | XMM14 | ZMM15 | YMM15 | XMM15 | 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
Linux x86_64 context switch excerpt

---

Complete fiber context switch on Unix and Windows

---

8-bit registers
16-bit registers
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

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
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: 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.
  • If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i’s request is granted

while(1) {
  Critical section
  Mutex, spinlock, etc. are ways to implement
  Non-critical section
}

Did we get all the important conditions?
Why is correctness defined in terms of locks?

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider
Implementing Locks

```
int lock_value = 0;
int* 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

Completely and utterly broken. How can we fix it?
HW Support for Read-Modify-Write (RMW)

IDEA: hardware implements something like:

```c
bool rmw(addr, value) {
    atomic {
        tmp = *addr;
        newval = modify(tmp);
        *addr = newval;
    }
}
```

Why is that hard? How can we do it?

Preview of Techniques:

- Bus locking
- Single Instruction ISA extensions
  - Test&Set
  - CAS: Compare & swap
  - Exchange, locked increment, locked decrement (x86)
- Multi-instruction ISA extensions:
  - LLSC: (PowerPC, Alpha, MIPS)
  - Transactional Memory (x86, PowerPC)

More on this later…
Implementing Locks with Test&set

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

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

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

(test & set  ~= CAS  ~= LLSC)
TST: Test&set
• Reads a value from memory
• Write “1” back to memory location

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

More on this later...
Implementing Locks

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

\[ F = ma \quad \sim \quad coherence \]
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
}
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
    try:  load lock, R0
        test R0
        bnz try
    store lock, 1
}
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

```c
// (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

// (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>

### Test & Set Code

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

### CAS Code

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

### Exchange Code

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

### LLSC Code

```c
bool LLSC(addr, val) {
    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

**LLSC: load-linked store-conditional**

<table>
<thead>
<tr>
<th>PPC, Alpha, MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool LLSC(addr, val) {</td>
</tr>
<tr>
<td>ret = *addr;</td>
</tr>
<tr>
<td>atomic {</td>
</tr>
<tr>
<td>if(*addr == ret) {</td>
</tr>
<tr>
<td>*addr = val;</td>
</tr>
<tr>
<td>return true;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>return false;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

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

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

P1

lock: 0

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

P2

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

P1

P2

lock: 0

lock: SIL 0

lock: [M] 0

lock: 0

lock: [SL] 0

P1

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))
                if(sc(lock, 1))
                    return;
    }
}
Implementing Locks with Test&set

```c
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

CPU A
while (test&set (lock));
// in critical region
lock: 1
...

CPU B
while (test&set (lock));
...

Main Memory

0xF0 lock: 1
0xF4 ...

L1

L2

L1

L2

• 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

\[
\text{Lock::Acquire()} \{ \\
\quad \text{while (test\&set(lock) == 1);} \\
\}
\]

Busy-wait on in-memory copy

\[
\text{Lock::Release()} \{ \\
\quad *\text{lock} = 0; \\
\}
\]

Test&Test&Set

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

Busy-wait on cached copy

\[
\text{Lock::Release()} \{ \\
\quad *\text{lock} = 0; \\
\}
\]

• 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 & Test & Set with Memory Hierarchies

What happens to lock variable’s cache line when different cpu’s contend for the same lock?
Test & Test & Set with Memory Hierarchies

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

CPU A
// in critical region
*lock = 0

CPU B
while(*lock);
if(test&set(lock))brk;

L1
lock: 0
...

L2
lock: 0
...

Main Memory
0xF0 lock: 1
0xF4 ...

L1
lock: 0
...

L2
lock: 0
...

Wait...why all this spinning?
How can we improve over busy-wait?

```
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
        __CLREX(); // 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

data

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

Illustration:

Initially: c1, c2, turn = 1, 1, 1
- c1 = 0
- c2 = 0
- turn = 1

Process 1:
- c1 = 0
- turn = 1
- c2 = 1

Process 2:
- c2 = 0
- turn = 2
- c1 = 1

Critical section 1:
turn = 2; c1 = 1; noncritical 1

Noncritical 1:
turn = 1; c1 = 1; noncritical 1

Critical section 2:
turn = 1; c2 = 1; noncritical 2
Questions?