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?
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

```
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;
    }
}
```

```
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
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></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children's CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Process Address Space

- ustack (1)
- kcode
- kbss
- kdata
- kheap

Why relevant?

- access requires kernel mode
- access possible in user mode

0 3 GB 1 GB

C0000000  C0400000  FFFFFFFF

user (1)
user (2)
user (1)
user (1)
kheap
kbss
kdata
kcode
ucode (1)
userdata (1)

Process Address Space

Why relevant?
State is shared through memory!

access possible in user mode

access requires kernel mode

P1

0

1 GB

3 GB

FFFFFFFFF

C0000000

C0400000

ustack (1)

kheap

kbss

kdata

kcode

userdata (1)

ucode (1)

user (2)

user (2)

user (2)

user (1)

user (1)

user (1)

kernel

free
Process Address Space

Why relevant?
State is shared through memory!

Q: How to share data across processes?
Why relevant?
State is shared through memory!
Q: How to share data across processes?
Anyone see another issue?
access possible in user mode
access requires kernel mode

Process Address Space

Anyone see another issue?
Why relevant?
State is shared through memory!
Q: How to share data across processes?
Abstractions for Concurrency
Abstractions for Concurrency

(a) Three processes each with one thread
Abstractions for Concurrency

(a) Three processes each with one thread

(b) One process with three threads
Abstractions for Concurrency

(a) Three processes each with one thread

(b) One process with three threads
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?
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?
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>
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>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!*
# 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?**
Where to Implement Threads:
Where to Implement Threads:

User Space

Kernel Space
Where to Implement Threads:

**User Space**

**Kernel Space**

A user-level threads package
Where to Implement Threads:

**User Space**

- Process
- Thread

A user-level threads package

**Kernel Space**

- Process
- Thread

A threads package managed by the kernel
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
Execution Context Management

“Task” == “Flow of Control”, but with less typing
“Stack” == Task State

Task Management
Execution Context Management

“Task” == “Flow of Control”, but with less typing
“Stack” == Task State

Task Management

• Preemptive
  • Interleave on uniprocessor
  • Overlap on multiprocessor
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
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
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
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?
Fibers: the Sweet Spot?

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

Blah blah fibers
blah thread
blah...
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
  • *Not* just threads without exception support
  • 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
  • *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
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

<table>
<thead>
<tr>
<th>ZMM0</th>
<th>YMM0</th>
<th>XMM0</th>
<th>ZMM1</th>
<th>YMM1</th>
<th>XMM1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZMM2</td>
<td>YMM2</td>
<td>XMM2</td>
<td>ZMM3</td>
<td>YMM3</td>
<td>XMM3</td>
</tr>
<tr>
<td>ZMM4</td>
<td>YMM4</td>
<td>XMM4</td>
<td>ZMM5</td>
<td>YMM5</td>
<td>XMM5</td>
</tr>
<tr>
<td>ZMM6</td>
<td>YMM6</td>
<td>XMM6</td>
<td>ZMM7</td>
<td>YMM7</td>
<td>XMM7</td>
</tr>
<tr>
<td>ZMM8</td>
<td>YMM8</td>
<td>XMM8</td>
<td>ZMM9</td>
<td>YMM9</td>
<td>XMM9</td>
</tr>
<tr>
<td>ZMM10</td>
<td>YMM10</td>
<td>XMM10</td>
<td>ZMM11</td>
<td>YMM11</td>
<td>XMM11</td>
</tr>
<tr>
<td>ZMM12</td>
<td>YMM12</td>
<td>XMM12</td>
<td>ZMM13</td>
<td>YMM13</td>
<td>XMM13</td>
</tr>
<tr>
<td>ZMM14</td>
<td>YMM14</td>
<td>XMM14</td>
<td>ZMM15</td>
<td>YMM15</td>
<td>XMM15</td>
</tr>
<tr>
<td>ZMM16</td>
<td>YMM16</td>
<td>XMM16</td>
<td>ZMM17</td>
<td>YMM17</td>
<td>XMM17</td>
</tr>
<tr>
<td>ZMM24</td>
<td>YMM24</td>
<td>XMM24</td>
<td>ZMM25</td>
<td>YMM25</td>
<td>XMM25</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>ZMM0</th>
<th>YMM0</th>
<th>XMM0</th>
<th>ZMM1</th>
<th>YMM1</th>
<th>XMM1</th>
<th>ST(0)</th>
<th>MM0</th>
<th>MM1</th>
<th>RA1</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZMM2</td>
<td>YMM2</td>
<td>XMM2</td>
<td>ZMM3</td>
<td>YMM3</td>
<td>XMM3</td>
<td>ST(2)</td>
<td>MM2</td>
<td>MM3</td>
<td>RA2</td>
<td>R13</td>
</tr>
<tr>
<td>ZMM4</td>
<td>YMM4</td>
<td>XMM4</td>
<td>ZMM5</td>
<td>YMM5</td>
<td>XMM5</td>
<td>ST(4)</td>
<td>MM4</td>
<td>MM5</td>
<td>RA3</td>
<td>R14</td>
</tr>
<tr>
<td>ZMM6</td>
<td>YMM6</td>
<td>XMM6</td>
<td>ZMM7</td>
<td>YMM7</td>
<td>XMM7</td>
<td>ST(6)</td>
<td>MM6</td>
<td>MM7</td>
<td>RA4</td>
<td>R15</td>
</tr>
<tr>
<td>ZMM8</td>
<td>YMM8</td>
<td>XMM8</td>
<td>ZMM9</td>
<td>YMM9</td>
<td>XMM9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM10</td>
<td>YMM10</td>
<td>XMM10</td>
<td>ZMM11</td>
<td>YMM11</td>
<td>XMM11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM12</td>
<td>YMM12</td>
<td>XMM12</td>
<td>ZMM13</td>
<td>YMM13</td>
<td>XMM13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM14</td>
<td>YMM14</td>
<td>XMM14</td>
<td>ZMM15</td>
<td>YMM15</td>
<td>XMM15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM16</td>
<td>ZMM17</td>
<td>ZMM18</td>
<td>ZMM19</td>
<td>ZMM20</td>
<td>ZMM21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM24</td>
<td>ZMM25</td>
<td>ZMM26</td>
<td>ZMM27</td>
<td>ZMM28</td>
<td>ZMM29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZMM31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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
Pthread Mutexes

- **Type:** `pthread_mutex_t`
Pthread Mutexes

- **Type:** `pthread_mutex_t`

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
```
Pthread Mutexes

- **Type:** `pthread_mutex_t`

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
```
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);
  ```
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);
```
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);
```
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);
```
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
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);
  ```
Review: correctness conditions

while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
Review: correctness conditions

- Safety
  - Only one thread in the critical region

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

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

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

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

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

Theorem: Every property is a combination of a safety property and a liveness property.
-Bowen Alpern & Fred Schneider
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 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) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}

Mutex, spinlock, etc. are ways to implement these

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

-Bowen Alpern & Fred Schneider

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

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

-Bowen Alpern & Fred Schneider

Did we get all the important conditions?

*Why is correctness defined in terms of locks?*
Implementing Locks

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

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

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

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

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

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

```c
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)
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;
    }
}
```
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?
HW Support for Read-Modify-Write (RMW)

Preview of Techniques:

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

IDEA: hardware implements something like:

Why is that hard?
How can we do it?
HW Support for Read-Modify-Write (RMW)

Preview of Techniques:
- Bus locking

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?
HW Support for Read-Modify-Write (RMW)

Preview of Techniques:

- Bus locking
- Single Instruction ISA extensions
  - Test&Set
  - CAS: Compare & swap
  - Exchange, locked increment, locked decrement (x86)

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

IDEA: hardware implements something like:

Why is that hard? How can we do it?
HW Support for Read-Modify-Write (RMW)

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)

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

IDEA: hardware implements something like:

Why is that hard? How can we do it?
HW Support for Read-Modify-Write (RMW)

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)

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?

More on this later…
Implementing Locks with Test&set

```c
int lock_value = 0;
int* lock = &lock_value;
```
Implementing Locks with Test&set

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

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

(test & set  ~= CAS  ~= LLSC)

TST: Test&set
- Reads a value from memory
- Write “1” back to memory location
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
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 are the problem(s) with this?
- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

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

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

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

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

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

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

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

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

```c
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

\[ F = ma \]
Multiprocessor Cache Coherence

Physics | Concurrency

$F = ma \sim coherence$
Multiprocessor Cache Coherence
Multiprocessor Cache Coherence

- P1: read X
Multiprocessor Cache Coherence

• P1: read X
Multiprocessor Cache Coherence

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

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

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

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

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

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

Each cache line has a state (M, E, S, I)
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
Multiprocessor Cache Coherence

Each cache line has a state (M, E, S, I)

- Processors "snoop" bus to maintain states
- Initially $\rightarrow$ ‘I’ $\rightarrow$ Invalid
Each cache line has a state (M, E, S, I)
• Processors “snoop” bus to maintain states
• Initially $\rightarrow$ ‘I’ $\rightarrow$ Invalid
• Read one $\rightarrow$ ‘E’ $\rightarrow$ exclusive
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
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
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
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

P1

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

P1

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

P1

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

---

P1

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

// (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

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

P1

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

P2
Cache Coherence Action Zone

P1

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

P2

// (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

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

lock:

// (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
}

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

P1

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

P2

// (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
}

// (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
}

// (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

lock:

lock() {
  try:  load lock, R0
  test R0
  bnz try
  store lock, 1
}

lock:

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
}

Memory
lock: 0

P1

P2

P3

// (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

P1

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

P2

// (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

P1
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try: load lock, R0
  test R0
  bnz try
  store lock, 1
}

P2

P3

// (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
}
Cache Coherence Action Zone II

P1

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

P2

// (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  
}

// (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
}
```
P1

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

P2

// (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
}
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
}
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

```plaintext
// (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>

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

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

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

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

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

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 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;
        } else {
            return false;
        }
    }
}
```

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

- load-linked is a load that is “linked” to a subsequent store-conditional
- Store-conditional only succeeds if value from linked-load is unchanged
HW Support for RMW: LL-SC

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

| PPC, Alpha, MIPS |
|------------------|------------------|
| bool LLSC(addr, val) { |
| ret = *addr; |
| atomic { |
| if(*addr == ret) { |
| *addr = val; |
| return true; |
| } |
| return false; |
| } |

```cpp
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
lock: 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;
    }
}
LLSC Lock Action Zone

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;
    }
}
LLSC Lock Action Zone

P₁

State Data
lock: S[I] 0

lock: 0

P₂

State Data
lock: I

lock: 1

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))
        return;
  }
}
LLSC Lock Action Zone

P1
lock: S[0] 0
lock: 0

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;
  }
}

lock: 0
LLSC Lock Action Zone

\[
\begin{align*}
P_1 & \quad \text{lock: M 1} & \quad \text{lock: 0} \\
\end{align*}
\]

\[
\begin{align*}
P_2 & \quad \text{lock: 1} \\
\end{align*}
\]

\[
\begin{align*}
P_1 & \quad \text{lock: 0} \\
\end{align*}
\]

\[
\begin{align*}
\text{lock}(\text{lock}) \{ \\
\quad \text{while}(1) \{ \\
\quad \quad \text{old} = \text{ll}(\text{lock}); \\
\quad \quad \text{if}(\text{old} == 0) \\
\quad \quad \quad \text{if}(\text{sc}(\text{lock}, 1)) \\
\quad \quad \quad \quad \text{return}; \\
\quad \} \\
\} \\
\end{align*}
\]

\[
\begin{align*}
P_2 & \quad \text{lock: 1} \\
\end{align*}
\]

\[
\begin{align*}
\text{lock}(\text{lock}) \{ \\
\quad \text{while}(1) \{ \\
\quad \quad \text{old} = \text{ll}(\text{lock}); \\
\quad \quad \text{if}(\text{old} == 0) \\
\quad \quad \quad \text{if}(\text{sc}(\text{lock}, 1)) \\
\quad \quad \quad \quad \text{return}; \\
\quad \} \\
\} \\
\end{align*}
\]
LLSC Lock Action Zone II

P1
lock: 0

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: 0

P2
lock: S(L) 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))
                return;
    }
}
LLSC Lock Action Zone II

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))
                return;
    }
}
LLSC Lock Action Zone II

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

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

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))
            return;
}
}
LLSC Lock Action Zone II

**P1**

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

**lock:** 0

**P2**

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

**lock:** 1

**M**

**lock:** I
LLSC Lock Action Zone II

P1

lock: 1

P2

lock: 0

lock: M 1

lock: 1

lock: 0

lock: 0

lock: 1

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;

}

}

Store conditional fails