Foundations: Concurrency Concerns
Synchronization Basics

Chris Rossbach
CS378H Fall 2018
9/5/18
Today

• Questions?
• Administrivia
  • You’ve started Lab 1 right?
• Foundations
  • Parallelism
  • Basic Synchronization
  • Threads/Processes/Fibers, Oh my!
  • 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)
  • Mark Silberstein (who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta)
  • Andy Tannenbaum
  • Don Porter
  • me...
  • Photo source: https://img.devrant.com/devrant/rant/r_10875_uRYQF.jpg
Faux Quiz (answer any 2, 5 min)

• Who was Flynn? Why is her/his taxonomy important?
• How does domain decomposition differ from functional decomposition? Give examples of each.
• Can a SIMD parallel program use functional decomposition? Why/why not?
Who is Flynn?

Michael J. Flynn
• Emeritus at Stanford
• Proposed taxonomy in 1966 (!!)
• 30 pages of publication titles
• Founding member of SIGARCH

• (Thanks Wikipedia)
Review: Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>X AXIS: Data Streams</th>
<th>Y AXIS: Instruction Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S I S D</strong></td>
<td>Single Instruction stream</td>
</tr>
<tr>
<td></td>
<td>Single Data stream</td>
</tr>
<tr>
<td><strong>S I M D</strong></td>
<td>Single Instruction stream</td>
</tr>
<tr>
<td></td>
<td>Multiple Data stream</td>
</tr>
<tr>
<td><strong>M I S D</strong></td>
<td>Multiple Instruction stream</td>
</tr>
<tr>
<td></td>
<td>Single Data stream</td>
</tr>
<tr>
<td><strong>M I M D</strong></td>
<td>Multiple Instruction stream</td>
</tr>
<tr>
<td></td>
<td>Multiple Data stream</td>
</tr>
</tbody>
</table>
Review: Problem Partitioning

• Domain Decomposition
  • SPMD
  • Input domain
  • Output Domain
  • Both

• Functional Decomposition
  • MPMD
  • Independent Tasks
  • Pipelining
Game of Life

• Given a 2D Grid:

• $v_t(i,j) = F(v_{t-1}(\text{of all its neighbors}))$
Domain decomposition

• Each CPU gets part of the input

How could we do a functional decomposition?

Issues?
• Accessing Data
  • Can we access \( v(i+1, j) \) from CPU 0
    • …as in a “normal” serial program?
    • Shared memory? Distributed?
  • Time to access \( v(i+1,j) \) == Time to access \( v(i-1,j) \) ?
  • Scalability vs Latency
• Control
  • Can we assign one vertex per CPU?
  • Can we assign one vertex per process/logical task?
  • Task Management Overhead
• Load Balance
• Correctness
  • order of reads and writes is non-deterministic
  • synchronization is required to enforce the order
  • locks, semaphores, barriers, conditionals….
Load Balancing

• Slowest task determines performance
Granularity

- Fine-grain parallelism
  - $G$ is small
  - Good load balancing
  - Potentially high overhead
  - Hard to get correct

- Coarse-grain parallelism
  - $G$ is large
  - Load balancing is tough
  - Low overhead
  - Easier to get correct

$$G = \frac{\text{Computation}}{\text{Communication}}$$
Performance: Amdahl’s law

- Speedup is bound by serial component.
- Speedup: $\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}}$

$\text{Speedup(\#CPUs)} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{\frac{A}{\#\text{CPUs}} + (1 - A)}$
Amdahl’s law

What makes something “serial” vs. parallelizable?
Amdahl’s law

End to end time: \( \frac{X}{2} + \frac{X}{4} = \frac{3}{4}X \) seconds

What is the “speedup” in this case?

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\text{#CPUs}} + (1 - A)} = \frac{1}{\frac{.5}{2 \text{ cpus}} + (1 - .5)} = 1.333
\]
Speedup exercise

End to end time: X seconds
What is the “speedup” in this case?

\[ \text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{.75/8 + (1-.75)} = 2.91x \]
Amdahl Action Zone
Amdahl Action Zone

The diagram illustrates the speedup (S) as a function of the number of CPUs (N) for two different percentages of the program that can be parallelized: 50% and 75%. The blue line represents 50% parallelizable code, while the orange line represents 75% parallelizable code. The x-axis represents the number of CPUs, and the y-axis represents the speedup.

Key observations:
- For 50% parallelizable code, the speedup increases as more CPUs are added, reaching a plateau at around 3x speedup.
- For 75% parallelizable code, the speedup increases slightly more slowly and also reaches a plateau at around 4x speedup.

This graph demonstrates the concept of the Amdahl's Law, which states that the speedup of a program using parallel processing is limited by the sequential part of the program that cannot be parallelized.
Amdahl Action Zone
Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

- \( N = \#CPUs \), \( S = \text{serial portion} = 1 - A \)
- Amdahl's law: \( \text{Speedup}(N) = \frac{1}{\frac{A}{N} + S} \)
  - **Strong scaling**: \( \text{Speedup}(N) \) calculated given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

- Gustafson's law: \( \text{Speedup}(N) = N + (N-1) \cdot S \)
  - **Weak scaling**: \( \text{Speedup}(N) \) calculated given amount of work per CPU is fixed
  - Keep the amount of work per CPU when adding more CPUs to keep the granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence**: speedup upper bound much higher

When is Gustavson’s law a better metric?
When is Amdahl’s law a better metric?
Super-linear speedup

• Possible due to cache
• But usually just poor methodology
• Baseline: *best* serial algorithm
• Example:

  Efficient **bubble sort**
  
  • Serial: 150s
  • Parallel 40s
  • Speedup: $\frac{150}{40} = 3.75$ ?
  NO NO NO!
  
  • Serial quicksort: 30s
  • Speedup = $\frac{30}{40} = 0.75X$

Can this happen?

Why insist on best serial algorithm as baseline?
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

Thread 1

\[
\begin{align*}
tmp1 &= X; \\
tmp1 &= tmp1 + 1; \\
X &= tmp1; \\
tmp1 &= X; \\
tmp2 &= X; \\
tmp2 &= tmp2 + 1; \\
tmp1 &= tmp1 + 1; \\
X &= tmp1; \\
X &= tmp2;
\end{align*}
\]

Thread 2

\[
\begin{align*}
tmp2 &= X; \\
tmp2 &= tmp2 + 1; \\
X &= tmp2;
\end{align*}
\]

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

void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}

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?

```
void move(T s, T d, Obj key) {
    LOCK(s);
    LOCK(d);
    tmp = s.remove(key);
    d.insert(key, tmp);
    UNLOCK(d);
    UNLOCK(s);
}
```

// WITH FINE-GRAIN LOCKS
move(a, b, key1);
move(b, a, key2);

Thread 0
DEADLOCK!

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:
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?*
Foundations: Read-Modify-Write (RMW)

- Locks *implemented* using RMW instructions
  - RMW: atomic and isolated action
    1. read a memory location into a register, *AND*
    2. write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires *cache coherence hardware*. Caches snoop the memory bus.

RMW Examples:
- **TST**: *Test&set*
  - Reads a value from memory
  - Write “1” back to memory location
- **CAS**: *Compare & swap* (68000)
  - Test the value against some constant
  - If the test returns true, set value in memory to different value
  - Report the result of the test in a flag
  - if [addr] == r1 then [addr] = r2;
- **Exchange, locked increment, locked decrement** (x86)
- **LLSC**: *Load linked/store conditional* (PowerPC, Alpha, MIPS)
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;
}

(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...
Programming and 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(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?
Anyone see an issue?
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>Time when process started</td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td>CPU time used</td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td>Children’s CPU time</td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td>Time of next alarm</td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Model

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

When might (a) be better than (b)? Vice versa?
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
The Thread Model

Each thread has its own stack
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-level threads package
- A user-level threads package

**Kernel Space**
- A threads package managed by the kernel
  - A threads package managed by the kernel
Implementing Threads in the Kernel

A threads package managed by the kernel
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

* 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”);
}

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);
```

```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
Multiprocessor Cache Coherence

Physics | Concurrency

\[ F = ma \sim coherence \]
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
  bnez try
  store lock, 1
}

// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
  try: load lock, R0
  test R0
  bnez 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
    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>
</tr>
</thead>
<tbody>
<tr>
<td>Most architectures</td>
<td>Many architectures</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exchange, locked increment/decrement,</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LLSC: load-linked store-conditional</th>
</tr>
</thead>
<tbody>
<tr>
<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;
    }
}

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>

```java
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

P2
lock: 1

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: [M] 0

lock: 0

P2

lock: SIL 0

lock: 0

S[L] 1

M

lock: S[L] 0

I

Store conditional fails

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?

- 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

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

Busy-wait on in-memory copy

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

Test&Test&Set

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

Busy-wait on cached copy

```cpp
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 & 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

L1
lock: 0
...

L2
lock: 0
...

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

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

L1
lock: 0
...

L2
lock: 0
...

Wait...why all this spinning?
```
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
                __CLREX(); // remove __LDREXB() lock
                LockedIn = 1;
            }
            else task_yield(); // give away cpu
        } else task_yield(); // give away cpu
    } while(!LockedIn);
}
```
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

Th. J. Dekker's Solution

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

process 1

critical section 1;
turn = 2; c1 = 1;
noncritical 1

process 2

critical section 2;
turn = 1; c2 = 1;
noncritical 2
Lab #1

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

• Start early!!!
Questions?