Foundations:
Concurrency Concerns
Synchronization Basics

Chris Rossbach
CS378H
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?
• What is an RMW instruction? How can they be used to construct synchronization primitives? How can sync primitives be constructed without them?
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

**Y AXIS:** Instruction Streams

**X AXIS:** Data Streams

- **SISD**:
  - Single Instruction stream
  - Single Data stream

- **SIMD**:
  - Single Instruction stream
  - Multiple Data stream

- **MISD**:
  - Multiple Instruction stream
  - Single Data stream

- **MIMD**:
  - Multiple Instruction stream
  - Multiple Data stream
Review: Problem Partitioning

- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both

- Functional Decomposition
  - MPMD
  - Independent Tasks
  - Pipelining
Domain decomposition

• Each CPU gets part of the input

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....
Performance: Amdahl’s law

- Speedup is bound by serial component.
- Speedup

\[ \text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}} \]

\[ \text{Speedup}(\#\text{CPUs}) = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{A \frac{\#\text{CPUs}}{1 - A}} \]
Amdahl Action Zone
Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

- $N = \#\text{CPUs}$, $S = \text{serial portion} = 1 - A$
- 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 = 30/40 = 0.75X
Concurrent 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
Model of concurrent execution

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

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

```
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?

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

DEADLOCK!

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

Thread 1
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;

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)

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

```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
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

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 and Threads and Fibers...

- Abstractions
- Containers
- State
  - Where is shared state?
  - How is it accessed?
  - Is it mutable?
Process Address Space

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></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>
Thread Model

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

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

Each thread has its own stack
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
Using threads

Ex. How might we use threads in a word processor program?
Where to Implement Threads:

**User Space**
- A user-level threads package

**Kernel Space**
- 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

- **ZMM0**, **ZMM2**, **ZMM4**, **ZMM6**, **ZMM8**, **ZMM10**, **ZMM12**, **ZMM14**, **ZMM16**, **ZMM24**: 8-bit registers

- **ST(0)**, **MM0**, **ST(1)**, **MM1**, **ST(2)**, **MM2**, **ST(3)**, **MM3**, **ST(4)**, **MM4**, **ST(5)**, **MM5**, **ST(6)**, **MM6**, **ST(7)**, **MM7**: Floating point registers

- **CW**, **FP_IP**, **FP_DP**, **FP_C**: Floating point units

- **SW**, **FW**, **TW**: Stack pointers

- **IP_DS**: Instruction pointer

- **P_DPC**: Program counter

- **P_FP**: Floating point context

- **Reç**

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);
  ```

Wait...what's the difference?

```c
int pthread_mutex_init(pthread_mutex_t *mutex,...);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```
Review: mutual exclusion model

• Safety
  • Only one thread in the critical region

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

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

Mutex, spinlock, etc. are ways to implement these

---

*Image: Diagram illustrating mutual exclusion model with a while loop structure and critical sections.*
Multiprocessor Cache Coherence

**Physics** | **Concurrency**

\[ F = ma \sim coherence \]
Multiprocessor Cache Coherence

- P1: read X
- P2: read X
- P2: X++
- P3: read X
Each cache line has a state (M, E, S, I)
- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic
Cache Coherence: single-thread

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

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

// (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:
  
  }

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

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

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

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

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

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

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

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 II

P1

lock: [M] 0

lock: 0

P2

lock: [SIL] 0

lock: 0

Store conditional fails

P1
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      if(sc(lock, 1))
        if(sc(lock, 1))
          return;
  }
}

P2
lock(lock) {
  while(1) {
    old = ll(lock);
    if(old == 0)
      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);
}
Lock::Release() {
  *lock = 0;
}
```

Busy-wait on in-memory copy

**Test&Test&Set**

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

Busy-wait on cached copy

*What is the problem with this?*

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

What happens to lock variable’s cache line when different cpu’s contend for the same lock?
What happens to lock variable’s cache line when different cpu’s contend for the same lock?
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) \rightarrow \text{enter(l)};
B(prio-100) \rightarrow \text{enter(l)}; \rightarrow \text{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 boolans
turn : integer

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

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

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

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

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

process 1

critical section 2:
  turn = 1; c2 = 1; noncritical 2

process 2

Th. J. Dekker’s Solution
Lab #1

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

• *Start early!!!*
Questions?