

Design of Parallel and High-Performance Computing

Fall 2013

Lecture: Locks and Lock-Free

Instructor: Torsten Hoefler & Markus Püschel

TA: Timo Schneider, Arnamoy Bhattacharyya

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Administrivia

- **Intermediate (very short) presentation: Thursday 11/27 during recitation**
 - Should have first results and a real plan!
 - Time to get very quick feedback
 - Focus on:
 - What tools/programming language/parallelization scheme do you use?*
 - Which architecture? (we only offer access to Xeon Phi, you may use different)*
 - How to verify correctness of the parallelization?*
 - How to argue about performance (bounds, what to compare to?)*
 - (Somewhat) realistic use-cases and input sets?*
 - What are the key concepts employed?*
 - What are the main obstacles?*
- **Final project presentation: Monday 12/15 during last lecture**
 - Report will be due in January!
 - *Still, starting to write early is very helpful --- write - rewrite - rewrite ...*

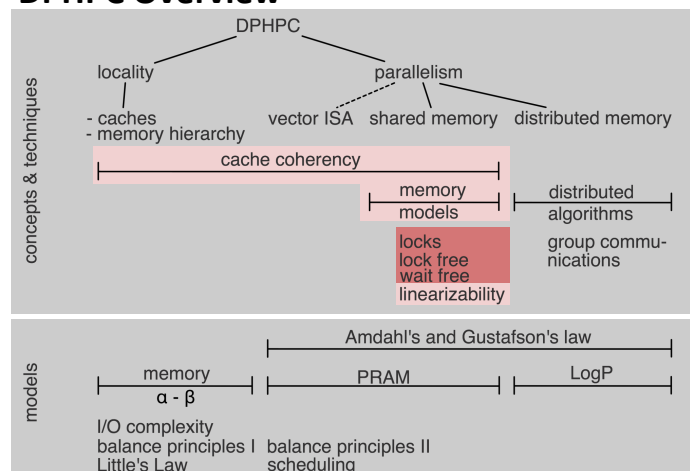
2

Review of last lecture

- **Language memory models**
 - History
 - Java/C++ overview
- **Locks**
 - Two-thread
 - Peterson
 - Many different locks, strengths and weaknesses
 - Lock options and parameters
- **Formal proof methods**
 - Correctness (mutual exclusion as condition)
 - Progress

3

DPHPC Overview



4

Goals of this lecture

- **N-thread locks!**
 - Hardware operations for concurrency control
- **More on locks (using advanced operations)**
 - Spin locks
 - Various optimized locks
- **Even more on locks (issues and extended concepts)**
 - Deadlocks, priority inversion, competitive spinning, semaphores
- **Case studies**
 - Barrier, reasoning about semantics
- **Locks in practice: a set structure**

5

Peterson in Practice ... on x86

- **Implement and run our little counter on x86**
- **100000 iterations**
 - $1.6 \cdot 10^{-6}\%$ errors
 - What is the problem?

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    while (flag[j] && victim == tid) {} // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

6

Peterson in Practice ... on x86

- Implement and run our little counter on x86

- 100000 iterations

- 1.6 · 10⁻⁶% errors
- What is the problem?
No sequential consistency for W(v) and R(flag[j])

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

7

Peterson in Practice ... on x86

- Implement and run our little counter on x86

- 100000 iterations

- 1.6 · 10⁻⁶% errors
- What is the problem?
No sequential consistency for W(v) and R(flag[j])
- Still 1.3 · 10⁻⁶%
Why?

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

8

Peterson in Practice ... on x86

- Implement and run our little counter on x86

- 100000 iterations

- 1.6 · 10⁻⁶% errors
- What is the problem?
No sequential consistency for W(v) and R(flag[j])
- Still 1.3 · 10⁻⁶%
Why?
Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm ("mfence");
    flag[tid] = 0; // I'm not interested
}
```

9

Correct Peterson Lock on x86

- Unoptimized (naïve sprinkling of mfences)

- Performance:

- No mfence
375ns
- mfence in lock
379ns
- mfence in unlock
404ns
- Two mfence
427ns (+14%)

```
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm ("mfence");
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    asm ("mfence");
    flag[tid] = 0; // I'm not interested
}
```

10

Locking for N threads

- Simple generalization of Peterson's lock, assume n levels l = 0...n-1

- Is it correct?

```
volatile int level[n] = {0,0,...,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
    for (int i = 1; i < n; i++) { //attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while ((∃k != tid) (level[k] >= i && victim[i] == tid) {});
    }
}

void unlock() {
    level[tid] = 0;
}
```

11

Filter Lock - Correctness

- Lemma: For 0 < j < n-1, there are at most n-j threads at level j!

- Intuition:

- Recursive proof (induction on j)
- By contradiction, assume n-j+1 threads at level j-1 and j
- Assume last thread to write victim
- Any other thread writes level before victim
- Last thread will stop at spin due to other thread's write

- j=n-1 is critical region

12

Locking for N threads

- **Simple generalization of Peterson's lock, assume n levels l = 0...n-1**
 - Is it starvation-free?

```
volatile int level[n] = {0,0,...,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
    for (int i = 1; i < n; i++) { //attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while ((∃k != tid) (level[k] >= i && victim[i] == tid )) {};
    }
}

void unlock() {
    level[tid] = 0;
}
```

13

Filter Lock Starvation Freedom

- **Intuition:**
 - Inductive argument over j (levels)
 - Base-case: level n-1 has one thread (not stuck)
 - Level j: assume thread is stuck
 - *Eventually, higher levels will drain (induction)*
 - *Last entering thread is victim, it will wait*
 - *Thus, only one thread can be stuck at each level*
 - *Victim can only have one value → older threads will advance!*

14

Filter Lock

- **What are the disadvantages of this lock?**

```
volatile int level[n] = {0,0,...,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level
void lock() {
    for (int i = 1; i < n; i++) { // attempt level i
        level[tid] = i;
        victim[i] = tid;
        // spin while conflicts exist
        while ((∃k != tid) (level[k] >= i && victim[i] == tid )) {};
    }
}

void unlock() {
    level[tid] = 0;
}
```

15

Lock Fairness

- **Starvation freedom provides no guarantee on how long a thread waits or if it is "passed"!**
- **To reason about fairness, we define two sections of each lock algorithm:**
 - **Doorway D** (bounded # of steps)
 - **Waiting W** (unbounded # of steps)
- **FIFO locks:**
 - If T_A finishes its doorway before T_B the $CR_A \rightarrow CR_B$
 - Implies fairness

```
void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid; // other goes first
    while (flag[j] && victim == tid) {};
}
```

16

Lamport's Bakery Algorithm (1974)

- **Is a FIFO lock (and thus fair)**
- **Each thread takes number in doorway and threads enter in the order of their number!**

```
volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
    while ((∃k != tid) (flag[k] && (label[k].k < * (label[tid],tid)))) {};
}

public void unlock() {
    flag[tid] = 0;
}
```

17

Lamport's Bakery Algorithm

- **Advantages:**
 - Elegant and correct solution
 - Starvation free, even FIFO fairness
- **Not used in practice!**
 - Why?
 - Needs to read/write N memory locations for synchronizing N threads
 - Can we do better?
 - *Using only atomic registers/memory*

18

A Lower Bound to Memory Complexity

- Theorem 5.1 in [1]: “If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes”
- So we’re doomed! Optimal locks are available and they’re fundamentally non-scalable. Or not?

[1] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. Information and Computation, 107(2):171–184, December 1993

19

Hardware Support?

- **Hardware atomic operations:**
 - Test&Set
Write const to memory while returning the old value
 - Atomic swap
Atomically exchange memory and register
 - Fetch&Op
Get value and apply operation to memory location
 - Compare&Swap
Compare two values and swap memory with register if equal
 - Load-linked/Store-Conditional LL/SC
Loads value from memory, allows operations, commits only if no other updates committed → mini-TM
 - Intel TSX (transactional synchronization extensions)
Hardware-TM (roll your own atomic operations)

20

Relative Power of Synchronization

- **Design-Problem I: Multi-core Processor**
 - Which atomic operations are useful?
- **Design-Problem II: Complex Application**
 - What atomic should I use?
- **Concept of “consensus number” C if a primitive can be used to solve the “consensus problem” in a finite number of steps (even if a threads stop)**
 - atomic registers have $C=1$ (thus locks have $C=1!$)
 - TAS, Swap, Fetch&Op have $C=2$
 - CAS, LL/SC, TM have $C=\infty$

21

Test-and-Set Locks

- **Test-and-Set semantics**
 - Memoize old value
 - Set fixed value TASval (true)
 - Return old value
- **After execution:**
 - Post-condition is a fixed (constant) value!

```
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
} // all atomic!
```

22

Test-and-Set Locks

- Assume TASval indicates “locked”
- Write something else to indicate “unlocked”
- TAS until return value is != TASval

- When will the lock be granted?
- Does this work well in practice?

```
volatile int lock = 0;

void lock() {
    while (TestAndSet(&lock) == 1);
}

void unlock() {
    lock = 0;
}
```

23

Contention

- **On x86, the XCHG instruction is used to implement TAS**
 - For experts: x86 LOCK is superfluous!
- **Cacheline is read and written**
 - Ends up in exclusive state, invalidates other copies
 - Cacheline is “thrown” around uselessly
 - High load on memory subsystem
x86 bus lock is essentially a full memory barrier ☹

```
movl $1, %eax
xchgl %eax, (%ebx)
```

24

Test-and-Test-and-Set (TATAS) Locks

- Spinning in TAS is not a good idea
- Spin on cache line in shared state
 - All threads at the same time, no cache coherency/memory traffic

Danger!

- Efficient but use with great care!
- Generalizations are dangerous

```
volatile int lock = 0;

void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}

void unlock() {
    lock = 0;
}
```

Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

1997 Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt schmidt@csc.wustl.edu Dept. of Computer Science Wash. U. St. Louis

Tim Harrison harrison@cs.wustl.edu Dept. of Computer Science Wash. U. St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3", ISBN, edited by Robert Martin, Frank Buschmann, and Dirk Rietze published by Addison-Wesley 1997.

Abstract
This paper shows how the common implementation [1] of the Singleton pattern does not work correctly in the presence of processor multi-threading or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton associated code impact the form and content of patterns used to develop concurrent software.

content of concurrency. To illustrate this, consider canonical implementations [1] of the Singleton pattern in multi-threaded environments. The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance. It normally allocates Singleton C++ programs to ensure the order of initialization of global static objects programs to not each detect and to therefore use up. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
    static Singleton* Instance(void)
    {
        if (Instance == 0)
            Instance = new Singleton();
        return Instance;
    }
};
```

Problem: Memory ordering leads to race-conditions!

Contention?

- Do TATAS locks still have contention?
- When lock is released, k threads fight for cache line ownership
 - One gets the lock, all get the CL exclusively (serially!)
 - What would be a good solution? (think "collision avoidance")

```
volatile int lock = 0;

void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}

void unlock() {
    lock = 0;
}
```

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically

- Locks granted in unpredictable order
- Starvation possible but unlikely
 - How can we make it even less likely?

```
volatile int lock = 0;

void lock() {
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}

void unlock() {
    lock = 0;
}
```

Similar to: T. Anderson: "The performance of spin lock alternatives for shared-memory multiprocessors", TPDS, Vol. 1 Issue 1, Jan 1990

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically

- Locks granted in unpredictable order
- Starvation possible but unlikely
 - Maximum waiting time makes it less likely

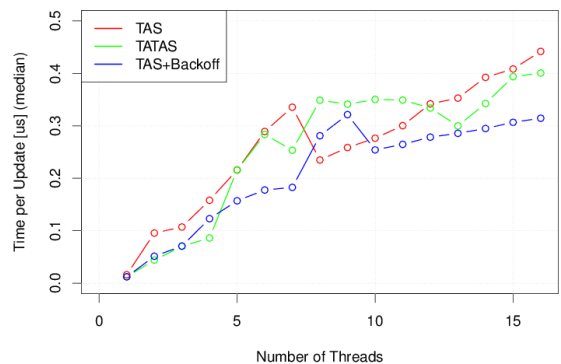
```
volatile int lock = 0;
const int maxtime=1000;

void lock() {
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time = min(time * 2, maxtime);
    }
}

void unlock() {
    lock = 0;
}
```

Similar to: T. Anderson: "The performance of spin lock alternatives for shared-memory multiprocessors", TPDS, Vol. 1 Issue 1, Jan 1990

Comparison of TAS Locks



Improvements?

- **Are TAS locks perfect?**
 - What are the two biggest issues?
 - Cache coherency traffic (contending on same location with expensive atomics)

-- or --

 - Critical section underutilization (waiting for backoff times will delay entry to CR)
- **What would be a fix for that?**
 - How is this solved at airports and shops (often at least)?
- **Queue locks -- Threads enqueue**
 - Learn from predecessor if it's their turn
 - Each threads spins at a different location
 - FIFO fairness

31

Array Queue Lock

- **Array to implement queue**
 - Tail-pointer shows next free queue position
 - Each thread spins on own location
CL padding!
 - index[] array can be put in TLS
- **So are we done now?**
 - What's wrong?
 - Synchronizing M objects requires $\Theta(NM)$ storage
 - What do we do now?

```
volatile int array[n] = {1,0,...,0};
volatile int index[n] = {0,0,...,0};
volatile int tail = 0;

void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}
```

32

CLH Lock (1993)

- **List-based (same queue principle)**
- **Discovered twice by Craig, Landin, Hagersten 1993/94**
- **2N+3M words**
 - N threads, M locks
- **Requires thread-local qnode pointer**
 - Can be hidden!

```
typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succ_blocked = 0;
    *qn = pred;
}
```

33

CLH Lock (1993)

- **Qnode objects represent thread state!**
 - succ_blocked == 1 if waiting or acquired lock
 - succ_blocked == 0 if released lock
- **List is implicit!**
 - One node per thread
 - Spin location changes
NUMA issues (cacheless)
- **Can we do better?**

```
typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succ_blocked = 0;
    *qn = pred;
}
```

34

MCS Lock (1991)

- **Make queue explicit**
 - Acquire lock by appending to queue
 - Spin on own node until locked is reset
- **Similar advantages as CLH but**
 - Only 2N + M words
 - Spinning position is fixed!
Benefits cache-less NUMA
- **What are the issues?**
 - Releasing lock spins
 - More atomics!

```
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
```

35

Lessons Learned!

- **Key Lesson:**
 - Reducing memory (coherency) traffic is most important!
 - Not always straight-forward (need to reason about CL states)
- **MCS: 2006 Dijkstra Prize in distributed computing**
 - "an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade"
 - "probably the most influential practical mutual exclusion algorithm ever"
 - "vastly superior to all previous mutual exclusion algorithms"
 - fast, fair, scalable → widely used, always compared against!

36

Time to Declare Victory?

- **Down to memory complexity of 2N+M**
 - Probably close to optimal
- **Only local spinning**
 - Several variants with low expected contention
- **But: we assumed sequential consistency ☹**
 - Reality causes trouble sometimes
 - Sprinkling memory fences may harm performance
 - Open research on minimally-synching algorithms!
Come and talk to me if you're interested

37

More Practical Optimizations

- **Let's step back to "data race"**
 - (recap) two operations A and B on the same memory cause a data race if one of them is a write ("conflicting access") and neither $A \rightarrow B$ nor $B \rightarrow A$
 - So we put conflicting accesses into a CR and lock it!
This also guarantees memory consistency in C++/Java!
- **Let's say you implement a web-based encyclopedia**
 - Consider the "average two accesses" – do they conflict?

38

Reader-Writer Locks

- **Allows multiple concurrent reads**
 - Multiple reader locks concurrently in CR
 - Guarantees mutual exclusion between writer and writer locks and reader and writer locks
- **Syntax:**
 - `read_(un)lock()`
 - `write_(un)lock()`

39

A Simple RW Lock

- **Seems efficient!?**
 - Is it? What's wrong?
 - Polling CAS!
- **Is it fair?**
 - Readers are preferred!
 - Can always delay writers (again and again and again)

```
const W = 1;
const R = 2;
volatile int lock=0; // LSB is writer flag!
```

```
void read_lock(lock_t lock) {
    AtomicAdd(lock, R);
    while(lock & W);
}
```

```
void write_lock(lock_t lock) {
    while(!CAS(lock, 0, W));
}
```

```
void read_unlock(lock_t lock) {
    AtomicAdd(lock, -R);
}
```

```
void write_unlock(lock_t lock) {
    AtomicAdd(lock, -W);
}
```

40

Fixing those Issues?

- **Polling issue:**
 - Combine with MCS lock idea of queue polling
- **Fairness:**
 - Count readers and writers

(1991) Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors

John M. Mellor-Crummey* (johnm@cs.cmu.edu)
Center for Research on Parallel Computation
Rice University, P.O. Box 1802
Houston, TX 77001-1802

Michael J. Suss* (mjsuss@cs.cmu.edu)
Computer Science Department
University of Rochester
Rochester, NY 14627

Abstract

Reader-writer synchronization relaxes the constraints of mutual exclusion to permit more than one process to request a shared object. On multiprocessors, mutual exclusion and reader-writer locks are typically subject to the scalability bottleneck. However, on shared-memory multiprocessors it is often advantageous to have processes keep track of their own readers and writers. In this paper we present reader-writer locks that combine the advantages of mutual exclusion and reader-writer locks. In this paper we present reader-writer locks that combine the advantages of mutual exclusion and reader-writer locks. In this paper we present reader-writer locks that combine the advantages of mutual exclusion and reader-writer locks.

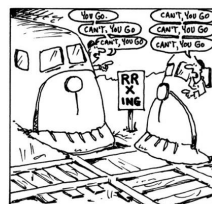
The final algorithm (Alg. 4) has a flaw that was corrected in 2003!

41

Deadlocks

- **Kansas state legislature: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."**

[according to Botkin, Harlow "A Treasury of Railroad Folklore" (pp. 381)]



What are necessary conditions for deadlock?

42

Deadlocks

- **Necessary conditions:**
 - Mutual Exclusion
 - Hold one resource, request another
 - No preemption
 - Circular wait in dependency graph
- **One condition missing will prevent deadlocks!**
 - → Different avoidance strategies (which?)

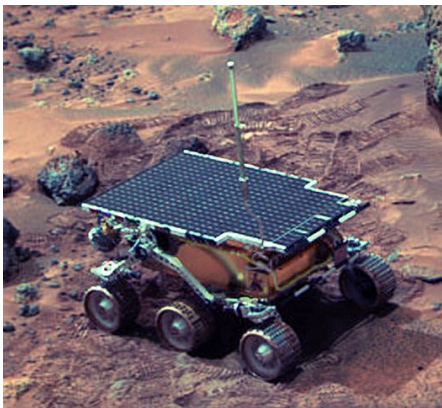
43

Issues with Spinlocks

- **Spin-locking is very wasteful**
 - The spinning thread occupies resources
 - Potentially the PE where the waiting thread wants to run → requires context switch!
- **Context switches due to**
 - Expiration of time-slices (forced)
 - Yielding the CPU

44

What is this?



45

Why is the 1997 Mars Rover in our lecture?

- **It landed, received program, and worked ... until it spuriously rebooted!**
 - → watchdog
- **Scenario (vxWorks RT OS):**
 - Single CPU
 - Two threads A,B sharing common bus, using locks
 - (independent) thread C wrote data to flash
 - Priority: A→C→B (A highest, B lowest)
 - Thread C would run into a livelock (infinite loop)
 - Thread B was preempted by C while holding lock
 - Thread A got stuck at lock ☹

[http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html]

46

Priority Inversion

- **If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!**
- **Can be fixed with the help of the OS**
 - E.g., mutex priority inheritance (temporarily boost priority of task in CR to highest priority among waiting tasks)

47

Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
 - Other threads can get them back on the queue!
- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**
- **Wait and signal are OS calls**
 - Often expensive, which one is more expensive?
Wait, because it has to perform a full context switch

48

Condition Variable Semantics

- **Hoare-style:**
 - Signaler passes lock to waiter, signaler suspended
 - Waiter runs immediately
 - Waiter passes lock back to signaler if it leaves critical section or if it waits again
- **Mesa-style (most used):**
 - Signaler keeps lock
 - Waiter simply put on run queue
 - Needs to acquire lock, may wait again