Computer Organization and Architecture

Concurrency & Parallelism

Jeongseob Ahn

Department of Software & Computer Engineering
Ajou University

Adapted slides from CIS501@Upenn

Concurrency

- Mediate multi-party access to shared resources
- Purpose: decrease response time
- Mechanism
  - Switch between different threads of control
  - Work on one thread when it can make useful progress; when it can't, suspend it and work on another thread
- E.g.,) running your clock, editor, chat at the same time on a single CPU
  - OS gives each of these programs a small time slice
  - Often slows throughput due to cost of switching contexts

Parallelism

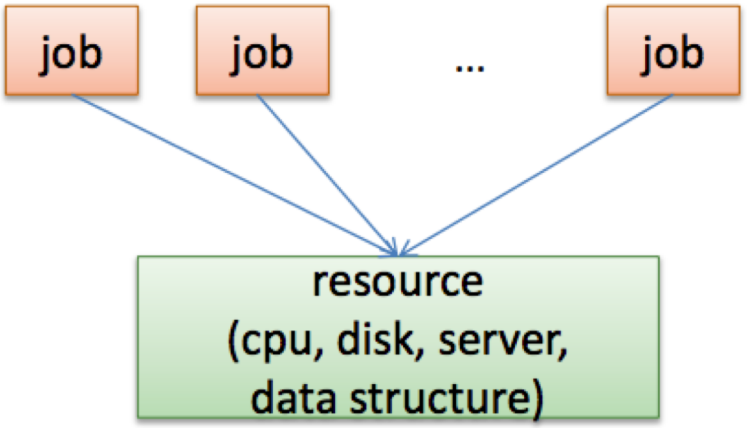
- Performs many tasks simultaneously
- Purpose: improves throughput
- Mechanism
  - Many independent computing devices
  - Decrease run time of program by utilizing multiple cores or computers
- E.g.,) running your web server handling independent requests from clients

Flavors of Parallelism

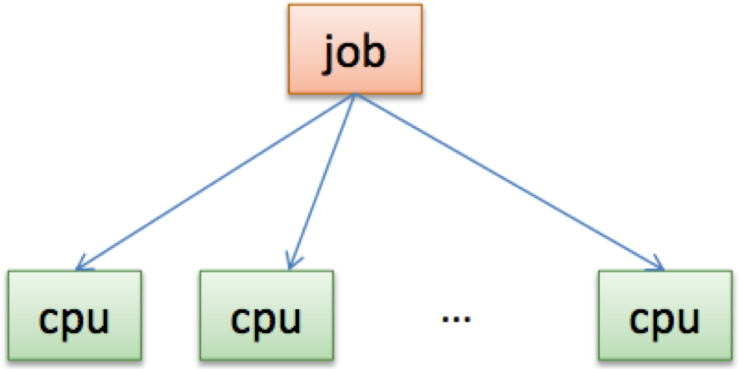
- Data parallelism
  - Same computation being performed on a lot of data
  - E.g.,) adding two vectors of numbers
- Task parallelism
  - Different computations/programs running at the same time
  - E.g.,) running web server and database
- Pipeline parallelism
  - Assembly pipeline

Concurrency vs. Parallelism

Concurrency:
Mediate/multiplex access
to shared resources

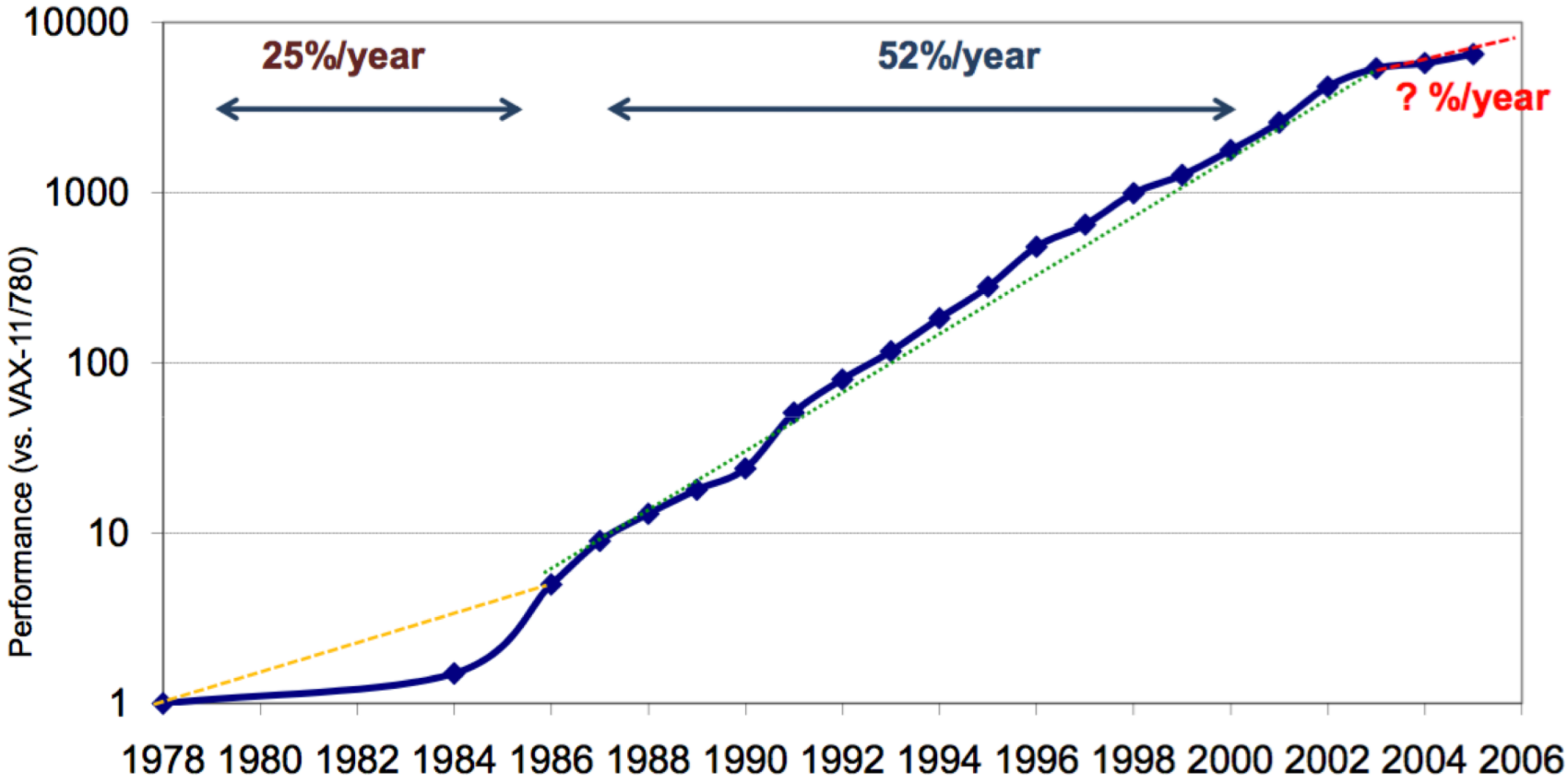


Parallelism:
Perform several independent
tasks simultaneously



Understanding Technology Trends

Uniprocessor Performance Trends



From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, October, 2006

Limits of Uniprocessors

- Era of uniprocessor improvement: mid 80s to early 2000s
  - 50% per year performance improvement
  - Faster clock frequency at every new generation of technology
    - Faster and smaller transistors
    - Deeper pipelines
  - Instruction-level Parallelism (ILP): speculative execution + superscalar
  - Improved cache hierarchy

- Limits of uniprocessors: after mid 2000s
  - Increasing power consumption started limiting microprocessor designs
    - Limits clock frequency
    - Heat dissipation problem
    - Need to reduce energy
  - Cannot keep increasing pipeline depth
  - Diminishing returns of ILP features

Communication in Multiprocessors

- How processors communication with each other?
- Two models for processor communication

- Message Passing
  - Processors can communicate by sending messages explicitly
  - Programmers need to add explicit message sending and receiving codes

- Shared Memory
  - Processors can communicate by reading from or writing to shared memory space
  - Use normal load and store instructions
  - Most commercial shared memory processors do not have separate private or shared memory → the entire address is shared among processors

Message Passing Multiprocessors

- Communication by explicit messages
- Commonly used in massively parallel systems (or large scale clusters)
  - Each node of clusters can be shared-memory MPs
  - Each node may have own OS

- **MPI (Message Passing Interface)**
  - Popular *de facto* programming standard for message passing
  - Application-level communication library

- IBM Blue Gene/L (2005)
  - 65,536 compute nodes (each node has dual processors)
  - 360 teraflops of peak performance
  - Distributed memory with message passing
  - Used MPI

MPI Example

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

if(myid == 0) {
    for(i=1;i<numprocs;i++) {
        sprintf(buff, "Hello %d! ", i);
        MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++) {
        MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
        printf("%d: %s\n", myid, buff);
    }
} else {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "reporting for duty\n");
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
}
```

Shared Memory Example: pthread

```
int arrayA [NUM_THREADS*1024]
int arrayB [NUM_THREADS*1024]
int arrayC [NUM_THREADS*1024]


void *sum(void *threadid) {
    long tid;
    tid = (long)threadid;
    for (i = tid; i < tid*1024; i++) {
        arrayC[i] = arrayA[i] + arrayB[i];
    }
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    ...
    for(t=0; t<NUM_THREADS; t++){
        pthread_create(&threads[t], NULL, psum, (void *)t);
    }
    for(t=0; t<NUM_THREADS; t++){
        pthread_join(threads[i],NULL);
    }
}
```

Shared Memory Example: OpenMP

```
int arrayA [NUM_THREADS*1024]
int arrayB [NUM_THREADS*1024]
int arrayC [NUM_THREADS*1024]


#pragma omp parallel default(none) shared(arrayA,arrayB,arrayC) private(i)
{
       #pragma omp for
       for (i=0; i<NUM_THREADS*1024; i++)
           arrayC[i] = arrayA[i] + arrayB[i];
} /*-- End of parallel region --*/
```

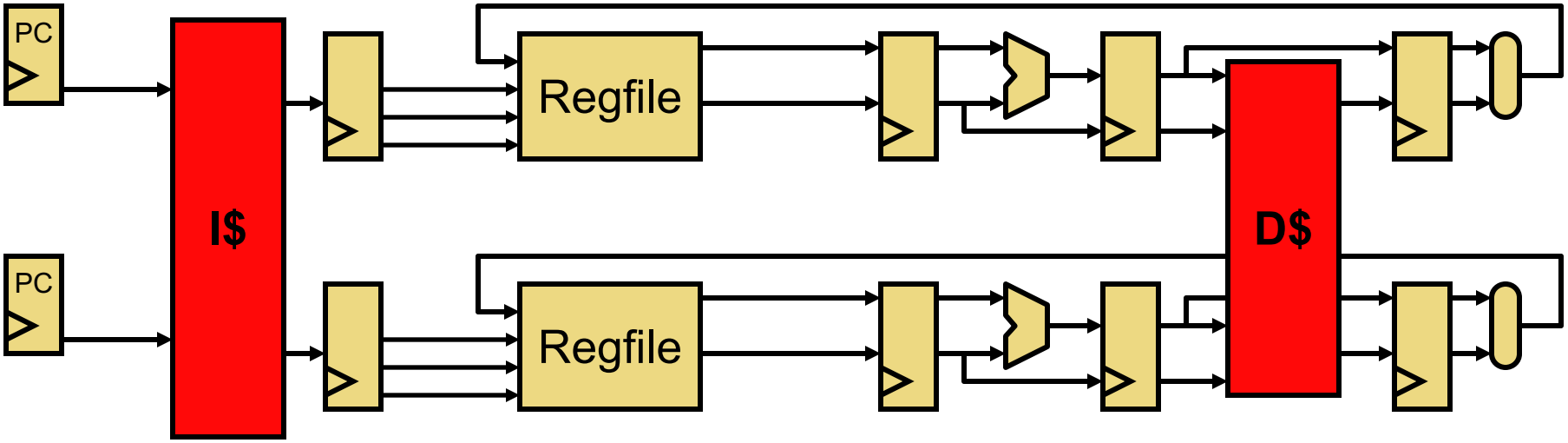
Shared Memory Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has its own stack frame for local variables
  - All memory shared, accessible by all threads
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - **Expressing parallel work via Thread-Level Parallelism (TLP)**
    - This is our focus!

Shared Memory Implementations

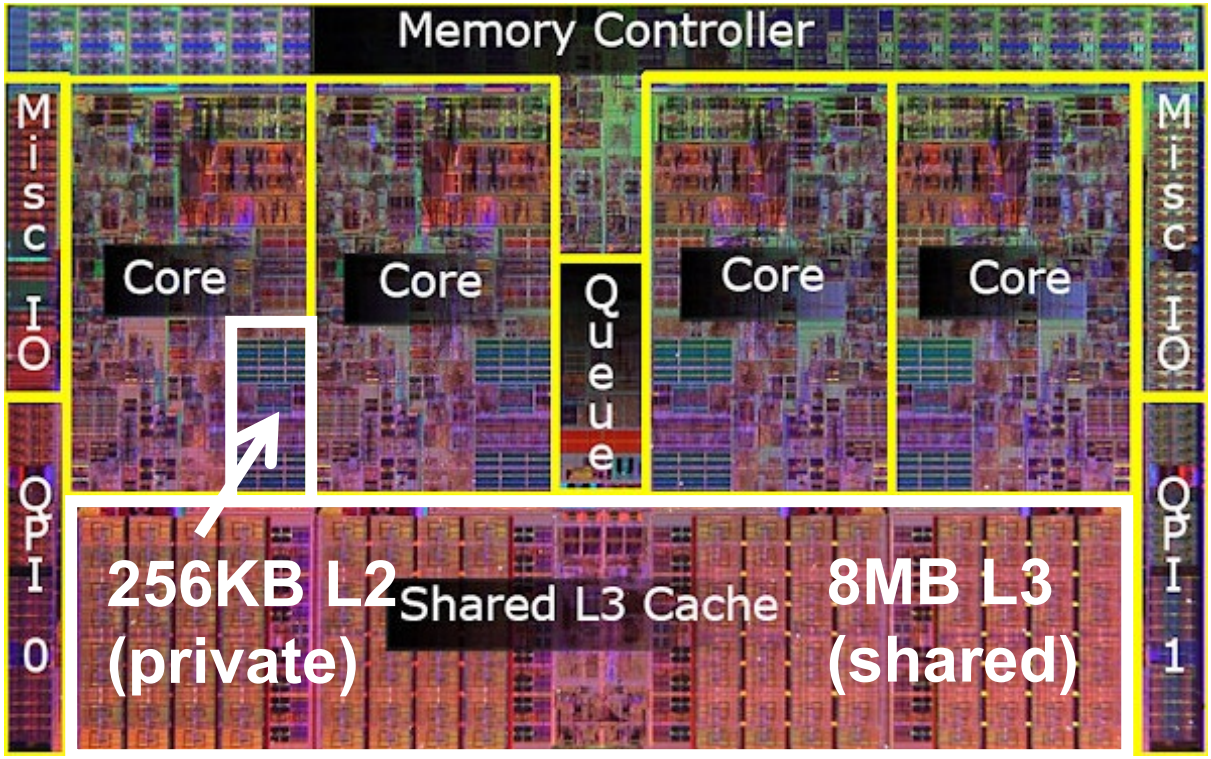
- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism
- **Multiprocessors**
  - Multiply execution resources, higher peak performance
  - Same interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores
- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model
- **All support the shared memory programming model**

Simplest Multiprocessor



- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share the caches (we'll address this bottleneck soon)
- Multiple threads execute
  - Shared memory programming model
  - Operations (loads and stores) are interleaved "at random"
  - Loads returns the value written by most recent store to location

Intel Core i7 Die Photo



- Chips today are 30–70% cache by area

Multicore & Energy

- Explicit parallelism (multicore) is highly energy efficient
- Dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel's Xscale
    - 1 GHz → 200 MHz reduces energy used by 30x
- Consider the impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th
- Assumes parallel speedup (a difficult task)
  - Subject to Ahmdal's law

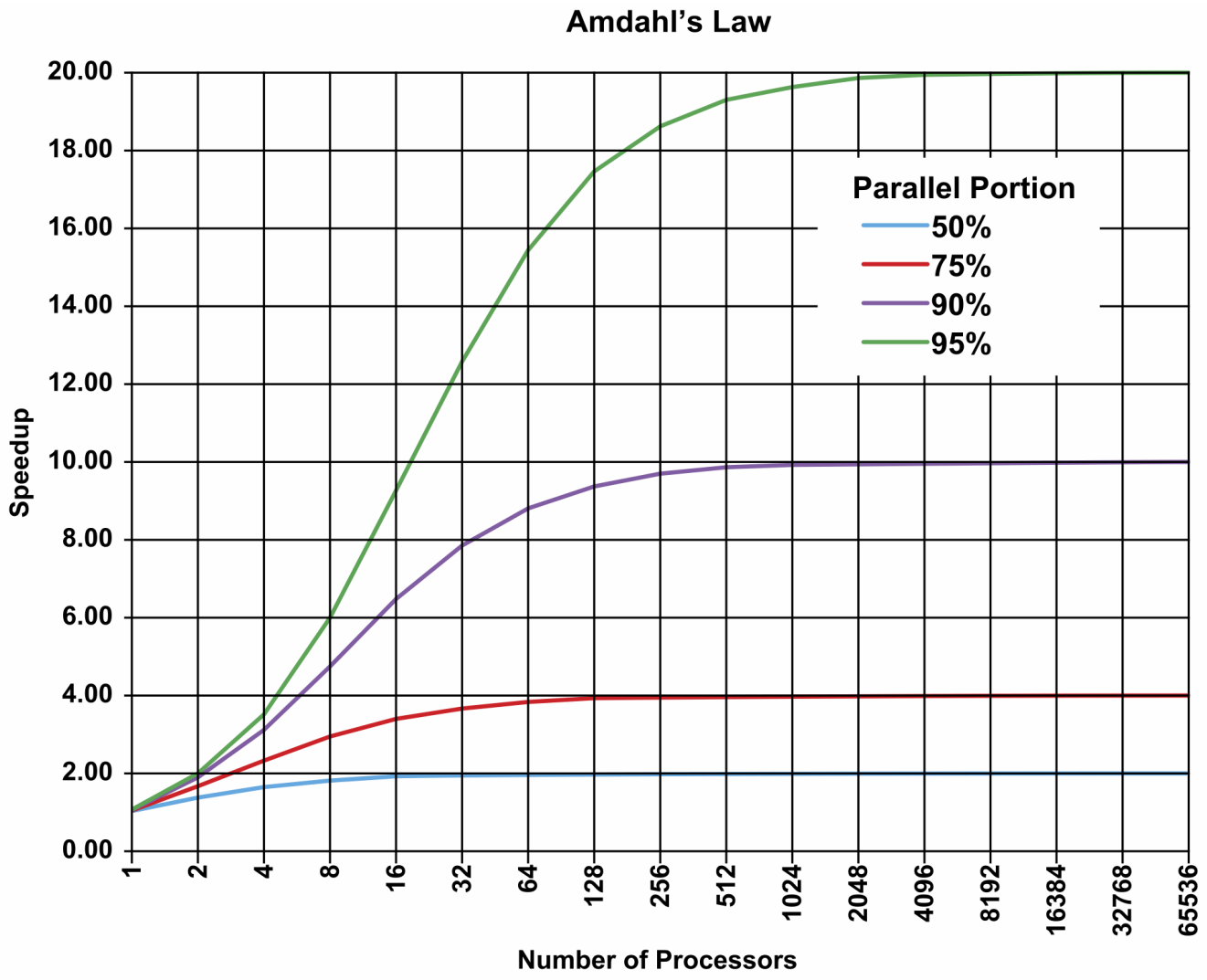
Amdahl's Law

- Restatement of the law of diminishing returns
  - Total speedup limited by non-accelerated piece
  - Analogy: drive to work & park car, walk to building

- Consider a task with a "parallel" and "serial" portion
  - What is the speedup with N cores?
  - Speedup(n, p, s) = (s+p) / (s + (p/n))
    - p is "parallel percentage", s is "serial percentage"
  - What about infinite cores?
    - Speedup(p, s) = (s+p) / s = 1 / s

- Example: can optimize 50% of program A
  - Even a "magic" optimization that makes this 50% disappear…
  - …only yields a 2X speedup

Amdahl's Law Graph



Source: Wikipedia

Parallel Programming

Parallel Programming

- One use of multiprocessors: **multiprogramming**
  - Running multiple programs with no interaction between them
  - Works great for a few cores, but what next?
- Or, programmers must **explicitly** express parallelism
  - "Coarse" parallelism beyond what the hardware can extract **implicitly**
  - Even the compiler can't extract it in most cases
- How? Several options:
  1. Call libraries that perform well-known computations in parallel
     - Example: a matrix multiply routine, etc.
  2. Add code annotations ("this loop is parallel"), OpenMP
  3. Parallel "for" loops, task-based parallelism, …
  4. Explicitly spawn "tasks", runtime/OS schedules them on the cores
- Parallel programming: key challenge in multicore revolution

Example #1: Bank Accounts

- Consider

```
struct acct_t { int balance; … };
struct acct_t accounts[MAX_ACCT];      // current balances

struct trans_t { int id; int amount; };
struct trans_t transactions[MAX_TRANS];  // debit amounts

for (i = 0; i < MAX_TRANS; i++) {
  debit(transactions[i].id, transactions[i].amount);
}

void debit(int id, int amount) {
  if (accounts[id].balance >= amount) {
    accounts[id].balance -= amount;
  }
}
```

- Can we do "debit" operations in parallel?
  - Does the order matter?

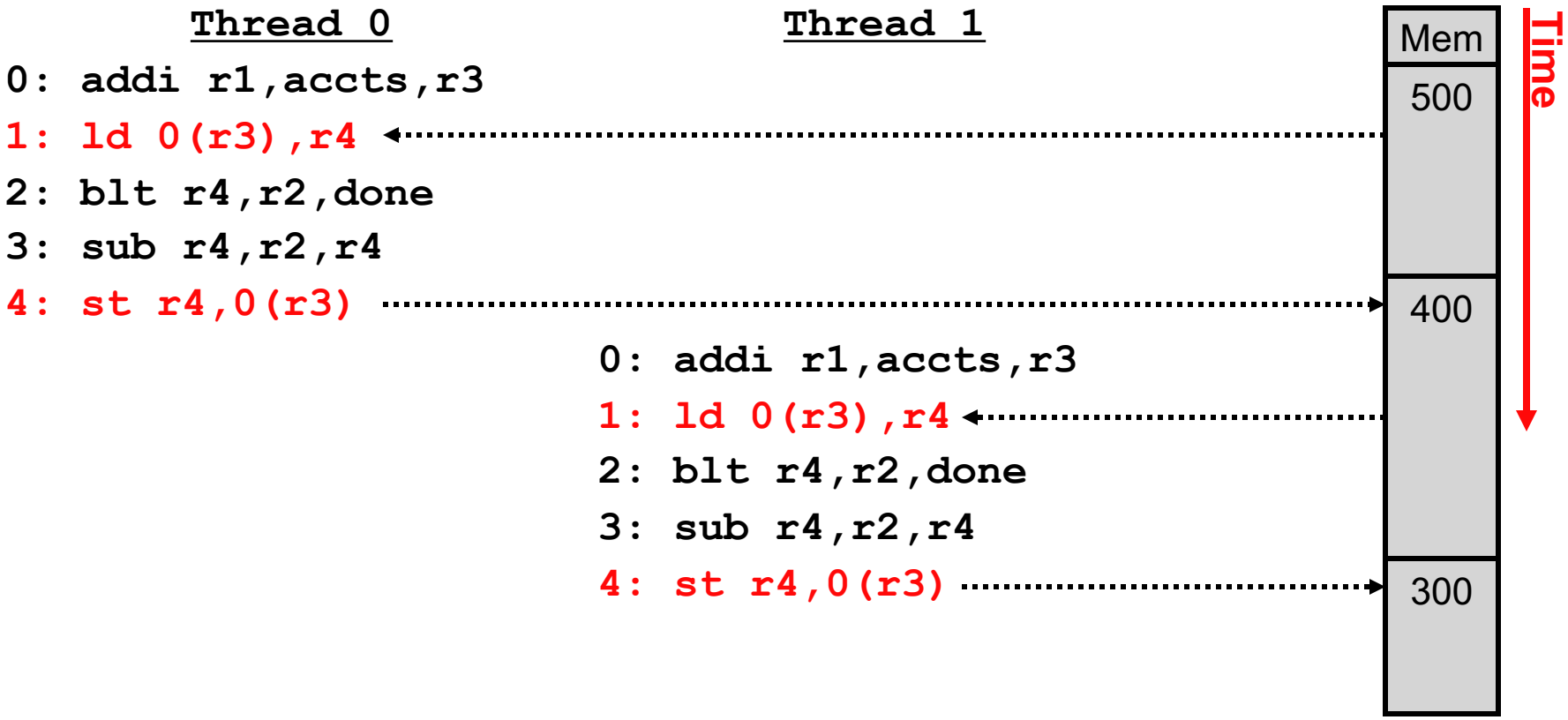
Example #1: Bank Accounts

```
struct acct_t { int bal; … };
shared struct acct_t accts[MAX_ACCT];
void debit(int id, int amt) {
  if (accts[id].bal >= amt)
  {
     accts[id].bal -= amt;
  }
}
```

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```

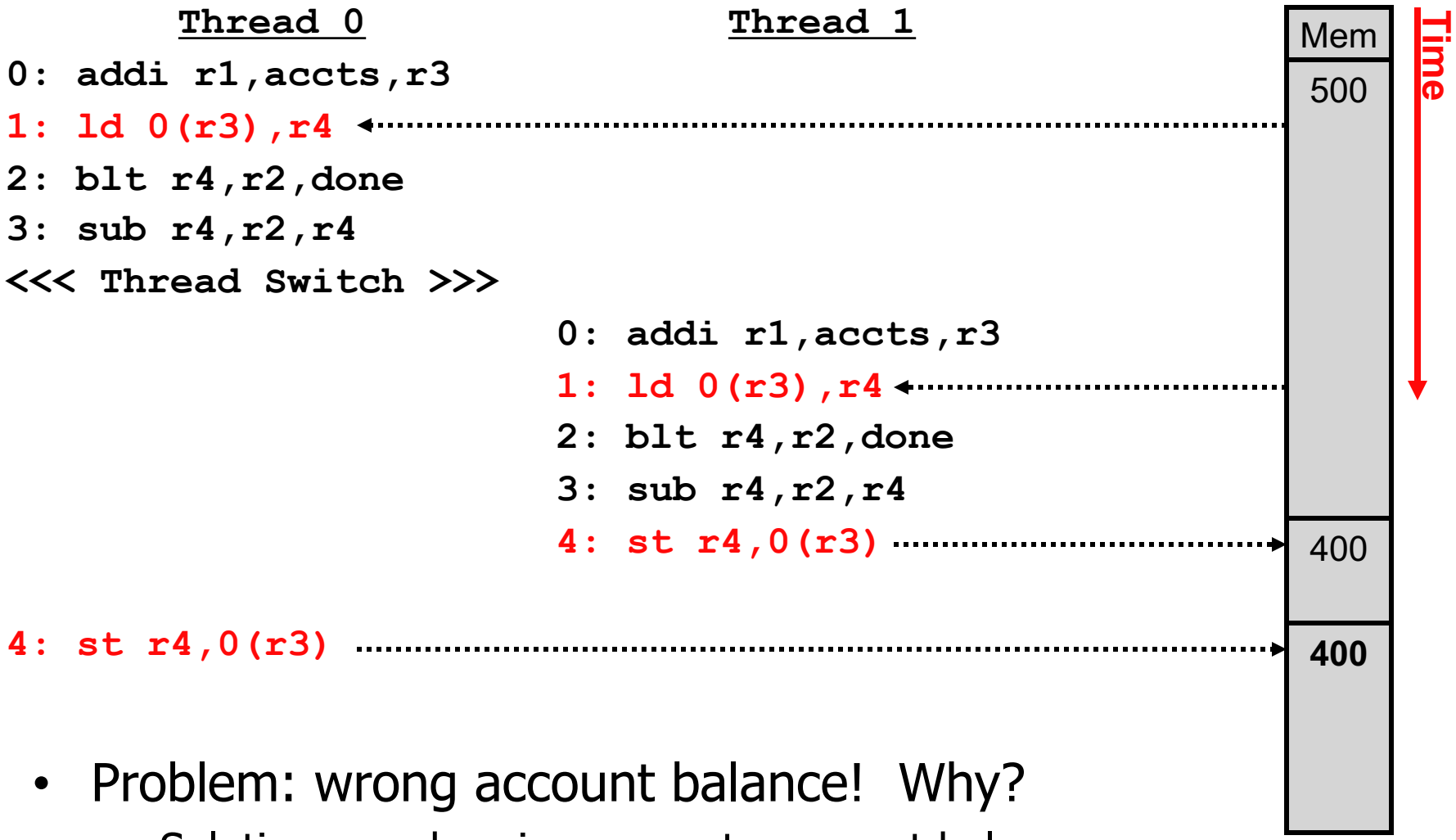
- Example of **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
  - `accts` is global and thus **shared**, can't register allocate
  - `id` and `amt` are private variables, register allocated to `r1`, `r2`
- Running example

An Example Execution



- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction executed on different processor
  - Track `accts[241].bal` (address is in `r3`)

A **Problem** Execution



- Problem: wrong account balance! Why?
  - Solution: synchronize access to account balance

Synchronization

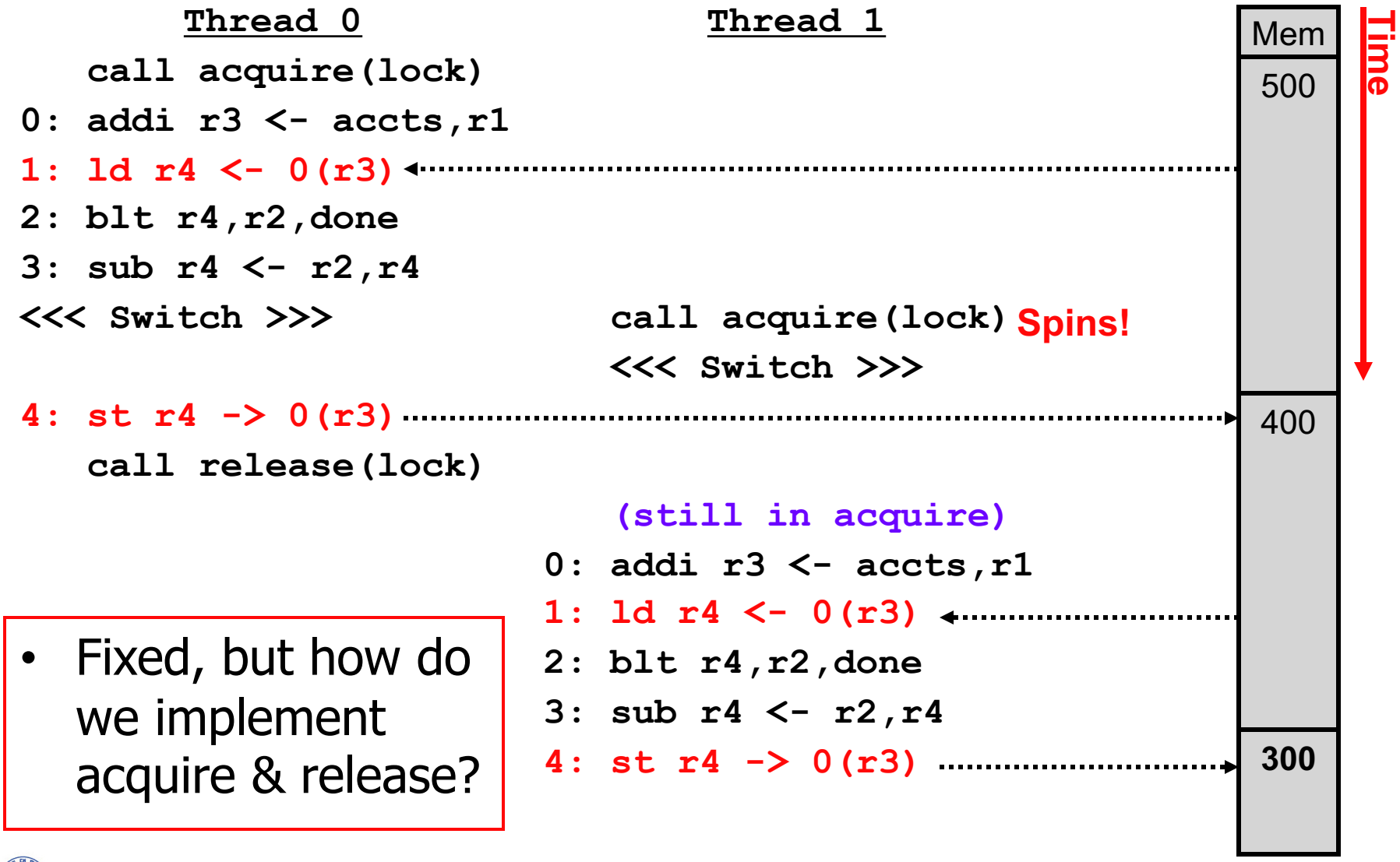
Synchronization:

- **Synchronization**: a key issue for shared memory
- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore")
  - Operations: `acquire(lock)` and `release(lock)`
  - Region between `acquire` and `release` is a **critical section**
  - Interfering `acquire` will block
- Another option: **Barrier synchronization**
  - Blocks until all threads reach barrier, used at end of "parallel_for"

```
struct acct_t { int bal; … };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
void debit(int id, int amt):
  acquire(lock);
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(lock);
```

critical section

A Synchronized Execution



- Fixed, but how do we implement acquire & release?

Strawman Lock (Incorrect)

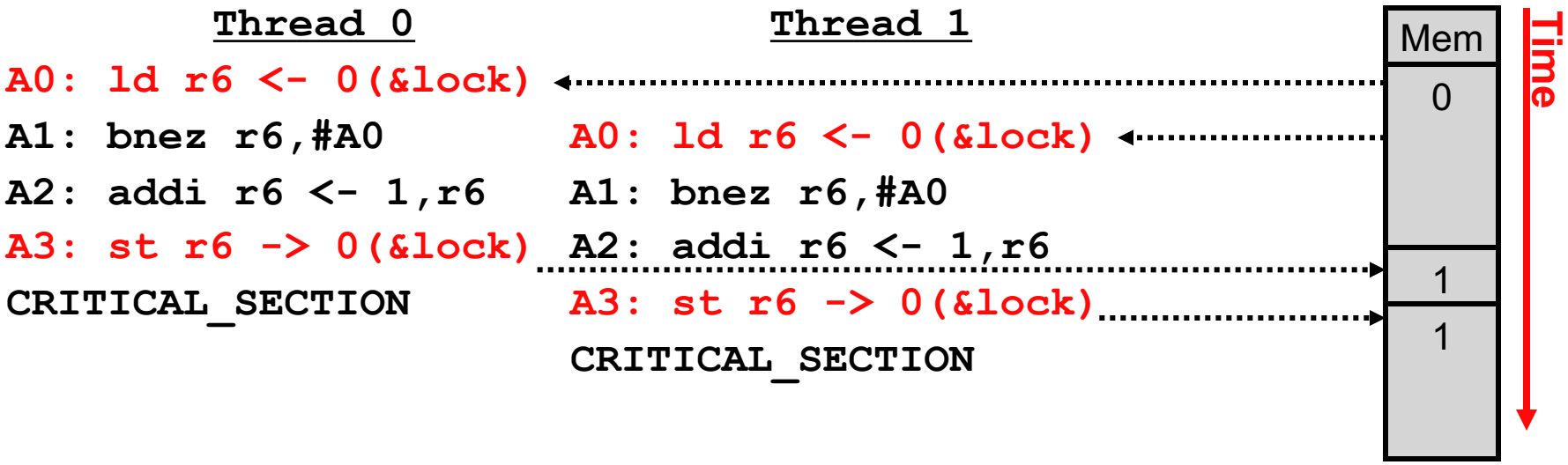
- **Spin lock**: software lock implementation
  - `acquire(lock): while (lock != 0) {} lock = 1;`
    - "Spin" while lock is 1, wait for it to turn 0

```
A0:  ld r6 <- 0(&lock)
A1:  bnez r6,A0
A2:  addi r6 <- 1,r6
A3:  st r6 -> 0(&lock)
```

  - `release(lock): lock = 0;`

```
R0:  st r0 -> 0(&lock)      // r0 holds 0
```

Incorrect Lock Implementation



- Spin lock makes intuitive sense, but doesn't actually work
  - Loads/stores of two `acquire` sequences can be interleaved
  - Lock `acquire` sequence also not atomic
  - **Same problem as before!**

- Note, `release` is trivially atomic

Correct Spin Lock: Compare and Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic compare-and-swap (CAS)**

```
cas r3 <- r1,r2,0(&lock)
```

    - Atomically executes:

```
ld r3 <- 0(&lock)
if r3 == r2:
  st r1 -> 0(&lock)
```

- New acquire sequence

```
A0: cas r3 <- 1,0,0(&lock)
A1: bnez r3,A0
```

  - If lock was initially busy (1), doesn't change it, **keep looping**
  - If lock was initially free (0), acquires it (sets it to 1), break loop

- Ensures lock held by **at most one thread**
  - Other variants: **exchange**, **compare-and-set**, **test-and-set (t&s)**, or **fetch-and-add**

RISC CAS

- `CAS`: a load+branch+store in one insn is not very "RISC"
  - Broken up into micro-ops, but then how is it made atomic?
- "Load-link" / "store-conditional" pairs
  - Atomic load/store pair

```
label:
  load-link r1 <- 0(&lock)
  // potentially other insns
  store-conditional r2 -> 0(&lock)
  branch-not-zero label   // check for failure
```

  - On **`load-link`**, processor remembers address…
    - …And looks for writes by other processors
    - If write is detected, next **`store-conditional`** will fail
      - Sets failure condition
- Used by ARM, PowerPC, MIPS, Itanium

Lock Correctness

Thread 0

```
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0
CRITICAL_SECTION
```

Thread 1

```
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0
```

+ Lock actually works…
  - Thread 1 keeps spinning

- Sometimes called a "test-and-set lock"
  - Named after the common "test-and-set" atomic instruction

Programming with Locks Is Tricky

- Multicore processors are the way of the foreseeable future
  - thread-level parallelism anointed as parallelism model of choice
  - Just one problem…
- Writing lock-based multi-threaded programs is tricky!
- More precisely:
  - Writing programs that are correct is not easy
  - Writing programs that are highly parallel is not easy
  - **Writing programs that are correct and parallel is even harder**
    - And that's the whole point, unfortunately
  - Selecting the "right" kind of lock for performance
    - Spin lock, queue lock, ticket lock, read/writer lock, etc.
  - **Locking granularity issues**

Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks**: e.g., one lock for entire database
  - \+ Easy to make correct: no chance for unintended interference
  - – Limits parallelism: no two critical sections can proceed in parallel

```
struct acct_t { int bal; … };
shared struct acct_t  accts[MAX_ACCT];
shared Lock_t lock;
void debit(int id, int amt) {
  acquire(lock);
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(lock);
}
```

Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks**: e.g., multiple locks, one per record
  - \+ Fast: critical sections (to different records) can proceed in parallel
  - – Easy to make mistakes
    - This particular example is easy
      - Requires only one lock per critical section

```
struct acct_t { int bal, Lock_t lock; …  };
shared struct acct_t  accts[MAX_ACCT];

void debit(int id, int amt) {
  acquire(accts[id].lock);
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(accts[id].lock);
}
```

- What about critical sections that require two locks?

Multiple Locks

- **Multiple locks**: e.g., acct-to-acct transfer
  - Must acquire both `id_from`, `id_to` locks
  - Running example with accts 241 and 37
  - Simultaneous transfers 241 → 37 and 37 → 241
  - Contrived… but even contrived examples must work correctly too

```
struct acct_t { int bal, Lock_t lock; …};
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
  acquire(accts[id_from].lock);
  acquire(accts[id_to].lock);
  if (accts[id_from].bal >= amt) {
     accts[id_from].bal -= amt;
     accts[id_to].bal += amt;
  }
  release(accts[id_to].lock);
  release(accts[id_from].lock);
}
```

Multiple Locks And Deadlock

Thread 0

```
id_from = 241;
id_to = 37;

acquire(accts[241].lock);
// wait to acquire lock 37
// waiting…
// still waiting…
```

Thread 1

```
id_from = 37;
id_to = 241;

acquire(accts[37].lock);
// wait to acquire lock 241
// waiting…
// …
```

FAILBlog

Multiple Locks And Deadlock

Thread 0

```
id_from = 241;
id_to = 37;

acquire(accts[241].lock);
// wait to acquire lock 37
// waiting…
// still waiting…
```

Thread 1

```
id_from = 37;
id_to = 241;

acquire(accts[37].lock);
// wait to acquire lock 241
// waiting…
// …
```

- **Deadlock**: circular wait for shared resources
  - Thread 0 has lock 241 and waits for lock 37
  - Thread 1 has lock 37 and waits for lock 241
  - Obviously this is a problem
  - The solution is …

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
  - Yet another thing to keep in mind when programming

```
struct acct_t { int bal, Lock_t lock; … };
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
  int id_first = min(id_from, id_to);
  int id_second = max(id_from, id_to);

  acquire(accts[id_first].lock);
  acquire(accts[id_second].lock);
  if (accts[id_from].bal >= amt) {
     accts[id_from].bal -= amt;
     accts[id_to].bal += amt;
  }
  release(accts[id_second].lock);
  release(accts[id_first].lock);
}
```

Correct Multiple Lock Execution

Thread 0

```
id_from = 241;
id_to = 37;
id_first = min(241,37)=37;
id_second = max(37,241)=241;

acquire(accts[37].lock);
acquire(accts[241].lock);
// do stuff
release(accts[241].lock);
release(accts[37].lock);
```

Thread 1

```
id_from = 37;
id_to = 241;
id_first = min(37,241)=37;
id_second = max(37,241)=241;

// wait to acquire lock 37
// waiting…
// …
// …
// …
acquire(accts[37].lock);
```

- Great, are we done? No

More Lock Madness

- What if…
  - Some actions (e.g., deposits, transfers) require 1 or 2 locks…
  - …and others (e.g., prepare statements) require all of them?
  - Can these proceed in parallel?
- What if…
  - There are locks for global variables (e.g., operation id counter)?
  - When should operations grab this lock?
- What if… what if… what if…

- **So lock-based programming is difficult…**
- **…wait, it gets worse**

And To Make It Worse…

- **Acquiring locks is expensive…**
  - By definition requires slow atomic instructions
    - Specifically, acquiring write permissions to the lock
  - Ordering constraints (up next) make it even slower

- **…and 99% of the time un-necessary**
  - Most concurrent actions don't actually share data
  - You pay to acquire the lock(s) for no reason

- Fixing these problem is an area of active research
  - One proposed solution "Transactional Memory"
  - Programmer uses construct: "atomic { … code … }"
    - Hardware, compiler & runtime executes the code "atomically"
    - Uses **speculation**, rolls back on conflicting accesses