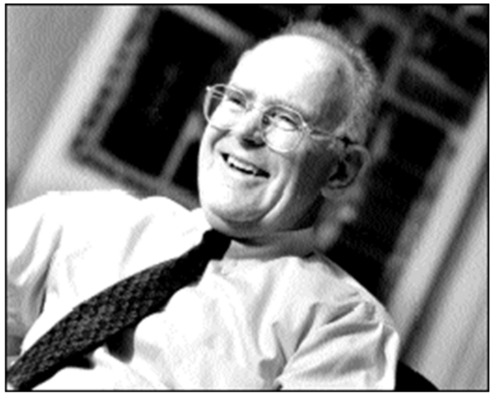
Shared-memory Programming

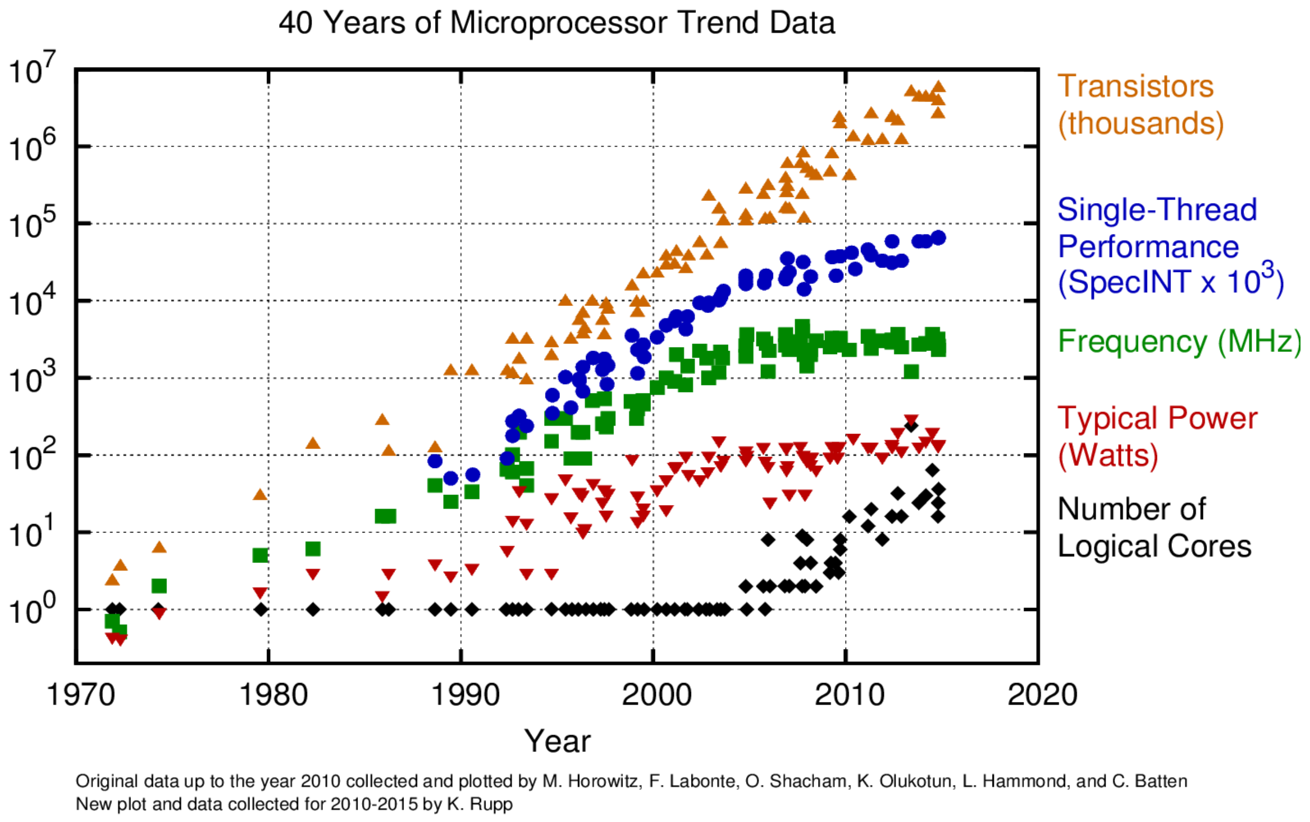
Overview

- Shared-memory
  - Architecture: chip has some number of cores (e.g., Intel Skylake has up to 18 cores depending on the model) with common memory
- Shared-memory programs
  - Application program is decomposed into a number of threads, which run on these cores
  - Each thread has its own stack, registers, PC
  - Data structures are in common memory
  - Threads communicate by reading and writing memory locations
- Programming systems: pThreads, OpenMP, Intel TBB
  - In this lecture, we will study pThreads
- Correctness and performance problems

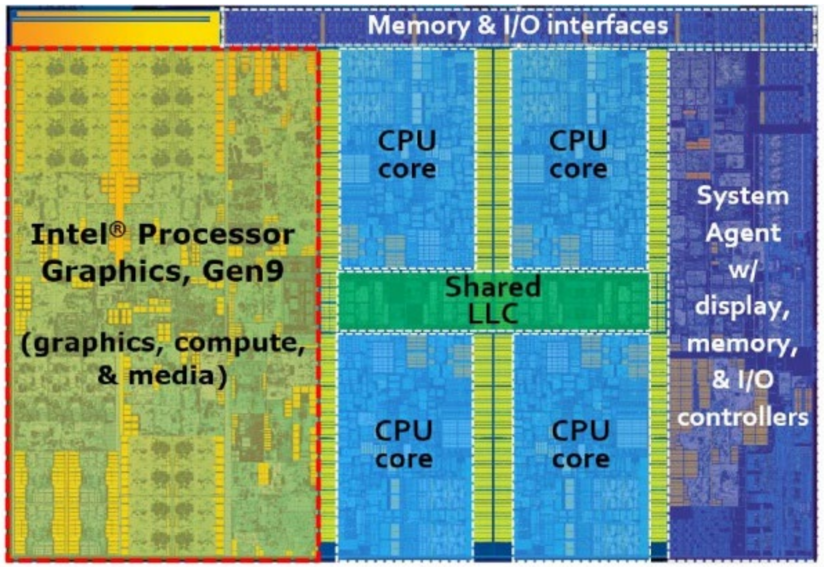
Shared-memory Architectures for Programmers

Moore's Law


Gordon Moore (Intel)

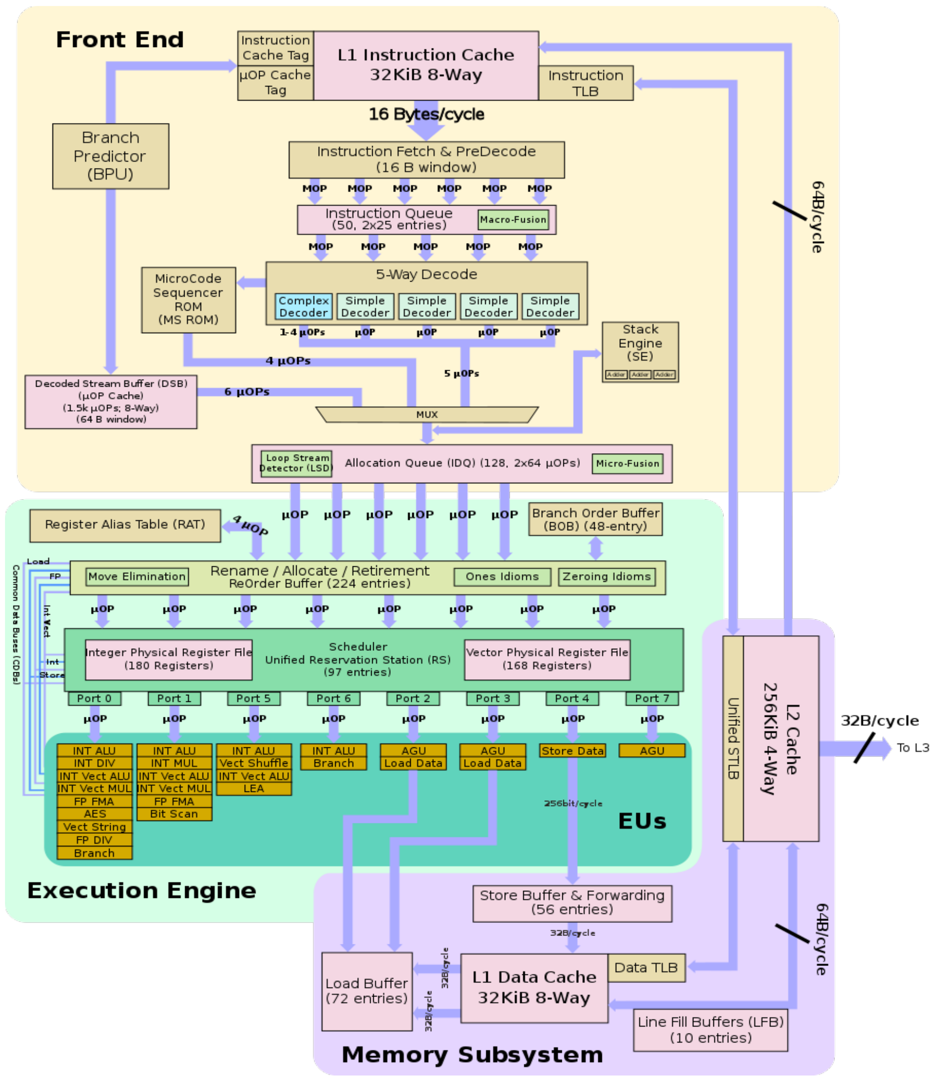


Intel Skylake chip



Memory & I/O interfaces
Intel® Processor Graphics, Gen9 (graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

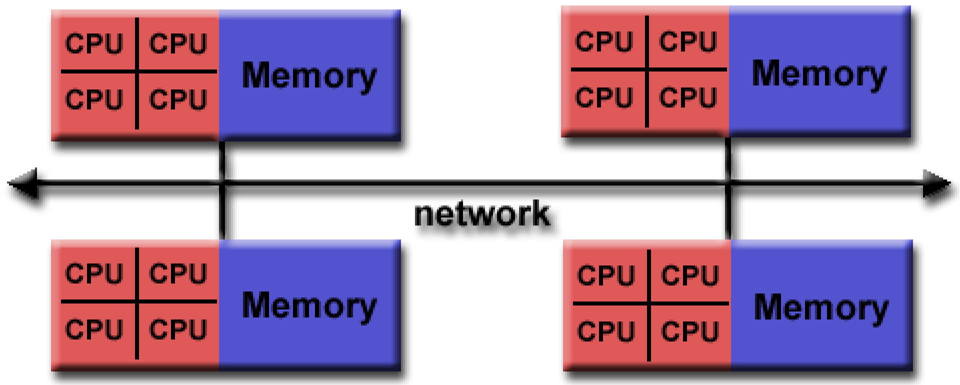
Chip



3/31/2020

Block diagram of each core[5]

Shared-memory m/c: cartoon picture



- Several multi-core chips connected by bus or network
- Single-address space for all cores but non-uniform memory access times

Typical latency numbers

From: *Latency numbers every HPC programmer should know*

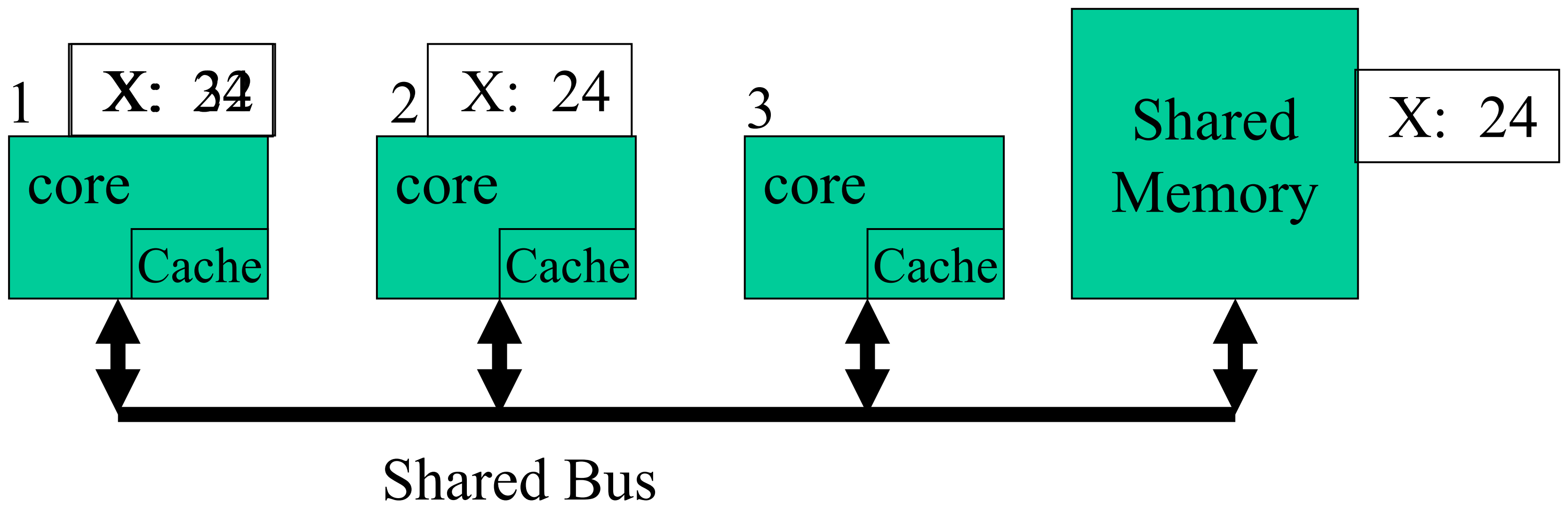
L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA operation	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
L3 cache hit	16-40 ns	40-300 cycles
256MB main memory reference	75-120 ns	TinyMemBench on
"Broadwell" E5-2690v4		
Send 4KB message between hosts	1-10 µs	MPICH on 10-100Gbps
Read 1MB sequentially from disk	5,000,000 ns	5 ms
~200MB/sec hard disk (seek time would be additional latency)		
Random Disk Access (seek+rotation)	10,000,000 ns	10 ms
Send packet CA->Netherlands->CA	150,000,000 ns	150 ms

Locality is important.

Architecture/software boundary

1. Cache coherence
   - interaction between caching and program semantics
   - we saw this in last lecture
2. Atomic instructions
   - interaction between threads
   - synchronization: coordination between threads to ensure parallel execution produces correct answers
3. Memory consistency model
   - interaction between instruction reordering within threads and program semantics

(1) Cache coherence problem



- Core 1 loads X: obtains 24 from memory and caches it
- Core 2 loads X: obtains 24 from memory and caches it
- Core 1 stores 32 to X: its locally cached copy is updated
- Core 3 loads X: what value should it get?
  - memory and core 2 think it is 24
  - core 1 thinks it is 32
- Illusion that there is a single variable X is broken

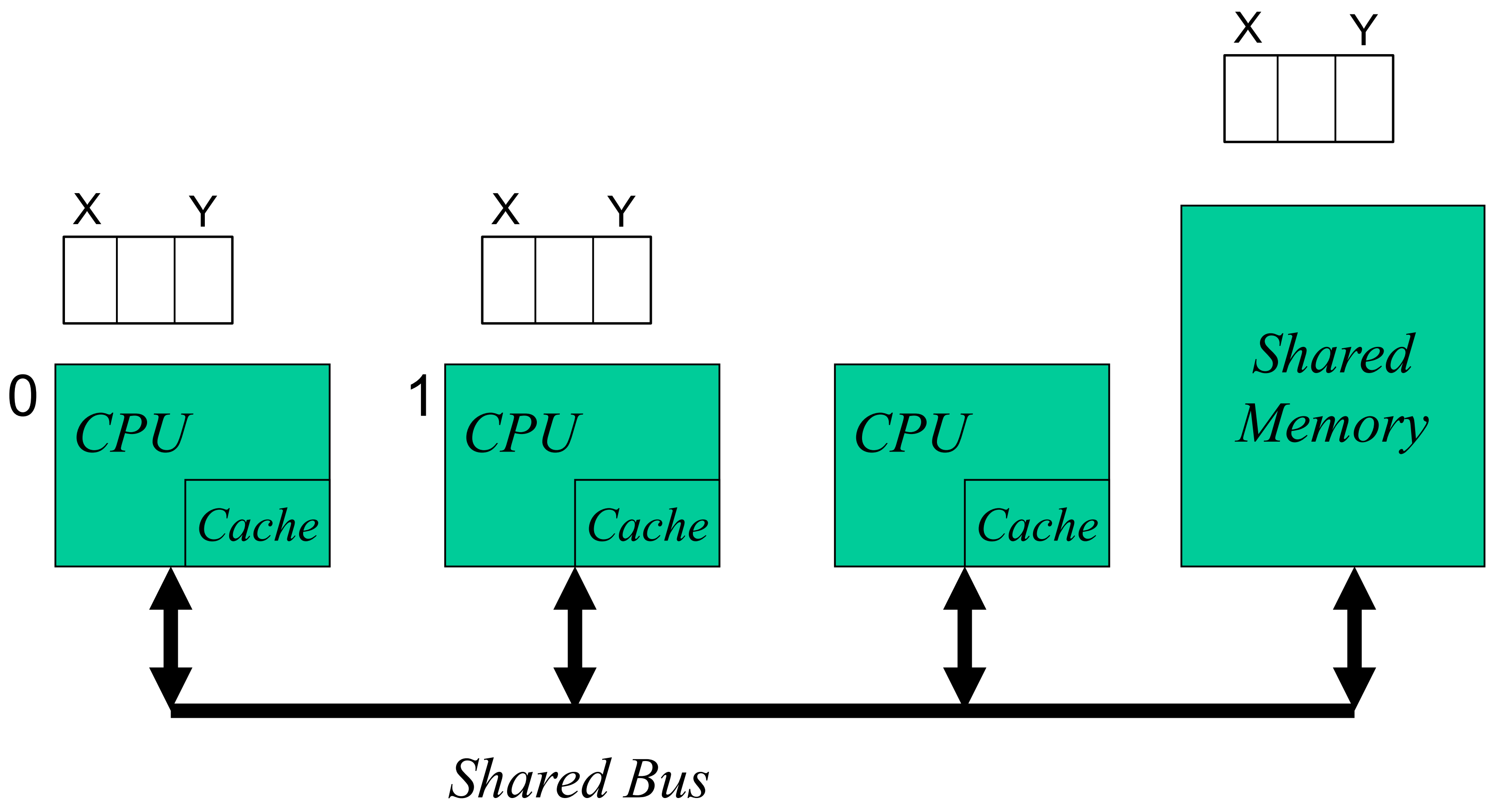
One solution

- Exclusive caching: ensure that at most one cache can have a given line at any time
- Implementation: snoopy caches
  - cache on each core ‘snoops’ (*i.e.* watches) for activity concerned with lines it has cached
  - load/store cache hit: perform operation just as in sequential machines
  - load/store cache miss:
    - perform bus cycle to obtain line
    - if some other cache has line, line is transferred to this cache and marked invalid in other cache
    - otherwise line is obtained from memory

Better solution: write-invalidate protocol

- Exclusive caching is too draconian
  - even read-only data cannot be in multiple caches
  - data written in one round that is read-only in next round cannot be in multiple caches
- Write-invalidate protocol
  - line can reside in several caches if all cores are reading from it
  - if a core wants to write to that line, line is invalidated from all other caches
- One implementation: MESI protocol
  - presented in previous lecture

False-sharing



- Core 0 reads and writes X
- Core 1 reads and writes Y
- No true sharing, but if X and Y are on the same line, there will be a lot of invalidation misses

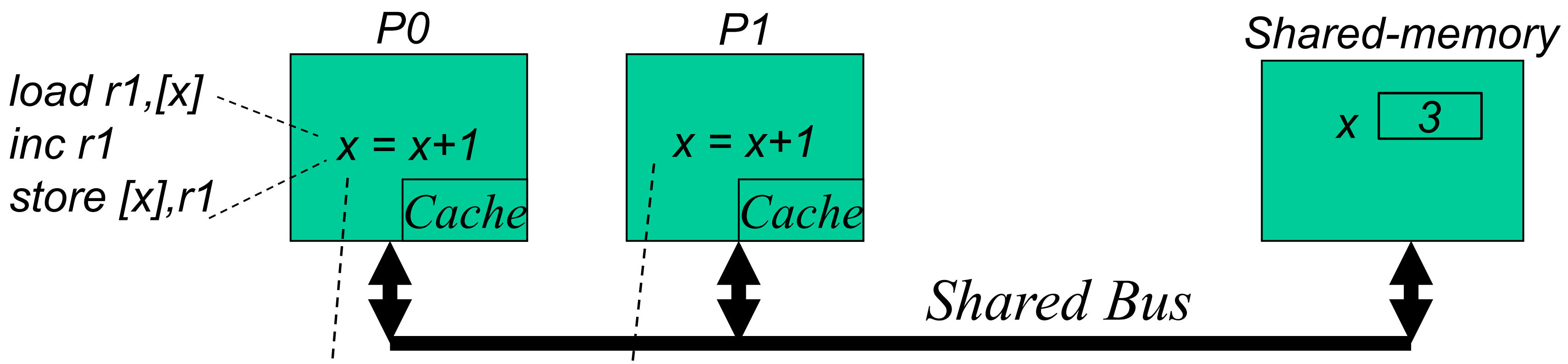
Summary

- Solution to cache-coherence:
  - snoopy caches and write-invalidate protocol
- True-sharing
  - a variable or array element is read and written by two or more cores repeatedly
- False-sharing
  - two or more cores read and write distinct variables or array elements that happen to be in the same cache line
- Sharing results in "ping-ponging" of cache lines between cores due to invalidations
  - reduces performance
  - to improve performance, try to reduce sharing of cache lines between cores

(2) Atomic instructions

- Example: sum all the elements of an array
  - core 0 adds up first half, core 1 adds up second half
  - each core adds its contribution to variable sum
- Problem: unless cores are synchronized, you get a *data-race*
    - result of read/modify/write may not be what you expect
    - final value can depend on how code is compiled and on scheduling of instructions from threads
- General problem:
  - read/modify/write must be performed atomically on a collection of variables or data structure elements

Data-race illustration



- Final value can be 4 or 5 depending on scheduling of instructions

time				
	load r1,[x]		load r1,[x]	
	inc r1		inc r1	
	store [x],r1			load r1,[x]
		load r1,[x]	store [x],r1	
		inc r1		inc r1
		store [x],r1		store [x],r1
↓	<u>x will have value 5</u>		<u>x will have value 4</u>	

Solution

- Architecture provides atomic instructions
  - small collection of read/modify/write instructions operating on ints, doubles, etc.
  - execute as though all other threads were suspended during execution of atomic instruction
  - examples:
    - swap(reg,addr)
      - swap value in memory at address addr with value in register reg
    - atomic add(reg,addr)
- Easy to modify MESI protocol to implement atomic instructions
  - like write but line is pinned in cache until instruction completes
  - no other core can steal line until instruction completes

Performance concern: contention

- Contention
  - two or more threads execute atomic instruction on given memory location simultaneously
- Correctness
  - hardware ensures atomic instructions are executed in some serial order
- Performance
  - threads simultaneously executing atomic instruction on given memory location will get serialized
- Rule of thumb
  - Uncontended atomic instruction is roughly as expensive as a write
  - Once you get contention, performance can degrade rapidly with amount of contention

Limitations of atomic instructions

- Atomic instructions give you atomicity for read/modify/write on data types like ints, floats, doubles (fit in cache line)
- Do not solve atomicity problem for updates to large amounts of data like arrays or structs
- Hardware solution: transactional memory
  - jury is still out about whether this is useful
- Software solution: locks
  - pThreads library: mutex-locks and spin-locks
  - implementation of locks uses atomic instructions

(3) Memory Consistency

- interaction between instruction reordering *within threads* and program semantics
- complicated issue: see later

pThreads library: low-level shared-memory programming

Threads

- Software analog of cores
  - Each thread has its own PC, SP, registers, and stack
  - All threads share heap and globals
- Runtime system handles mapping of threads to cores
  - if there are more threads than cores, runtime system will time-slice threads on cores
  - HPC applications: usually #threads = #cores
    - portability: number of threads is usually a runtime parameter
- Threads have two kinds of names
  - pThread name: opaque handle used by pThreads library (like social security number for people)
  - short name: usually an integer 0,1,2…(like first names for people) and used in application program to tell threads what to do or where to write their results

Thread Basics: Creation and Termination

- Program begins execution with main thread
- Creating threads:

```
int pthread_create (
    pthread_t *thread_handle,
    const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);
```

- Type (void *) is C notation for “raw address” (can point to anything)
- Thread is created and starts to execute thread_function with parameter arg, which specifies short name and other data to be passed to thread
- Thread handle: opaque handle for thread

Terminating threads

- Thread terminated when:
  - o it returns from its starting routine, or
  - o it makes a call to pthread_exit()
- Main thread
  - – exits with pthread_exit(): other threads will continue to execute
  - – otherwise other threads automatically terminated
- Cleanup:
  - – pthread_exit() routine does not close files
  - – any files opened inside the thread will remain open after the thread is terminated.

Example

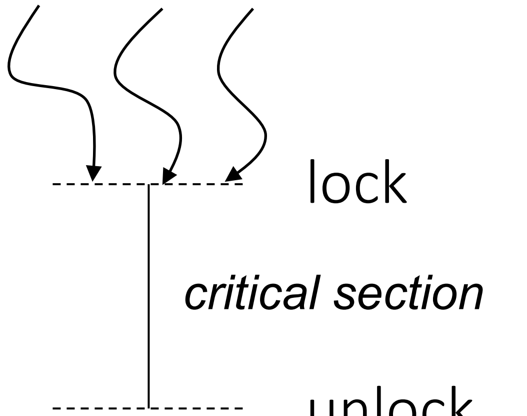
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5
int threadArg[NUM_THREADS];//parameters for threads
pthread_t handles[NUM_THREADS]; //store opaque handles for threads

void *PrintHello(void *threadIdPtr) {
   int shortId = * (int *)threadIdPtr;
   printf("\n%d: Hello World!\n", shortId);
   pthread_exit(NULL);
 }

int main(int argc, char *argv[]) {
   for(int t=0;t<NUM_THREADS;t++){
      printf("Creating thread %d\n", t);
      threadArg[t] = t;
      pthread_create(&handles[t], NULL, PrintHello, &threadArg[t]);
   }
   pthread_exit(NULL);
}
```

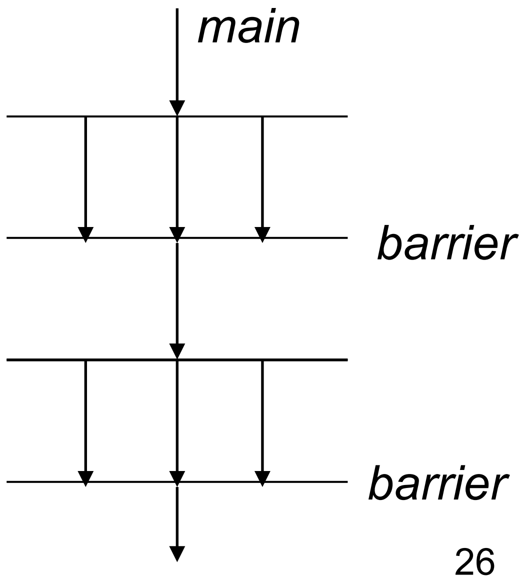
Output

```
Creating thread 0
Creating thread 1

0: Hello World!

1: Hello World!
Creating thread 2
Creating thread 3

2: Hello World!

3: Hello World!
Creating thread 4

4: Hello World!
```

Synchronization

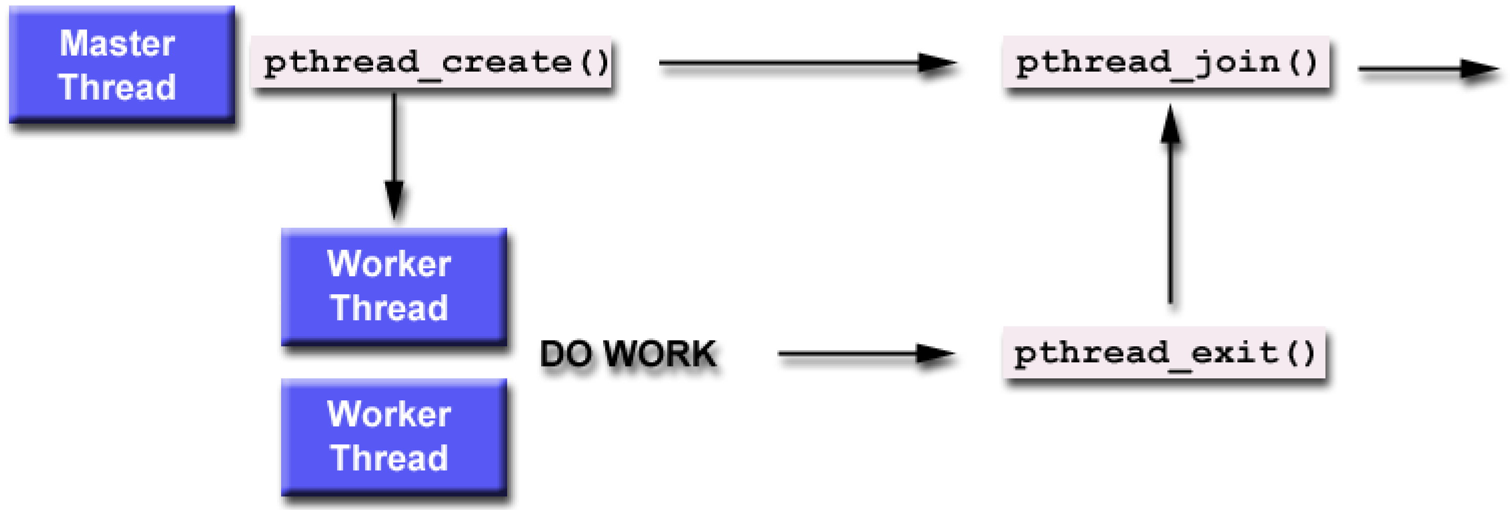
- Join:
  - block thread until some other thread terminates
- Lock:
  - used to ensure mutual exclusion: only one thread at a time can
    - access some data
    - execute some piece of code (critical section)
  - two kinds: mutexes and spin-locks
- Barrier:
  - all threads must reach barrier before any thread can move ahead





Join

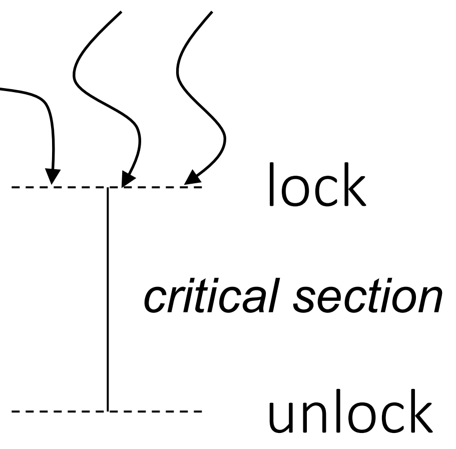
pthread_join (threadid,status)



- The pthread_join() function blocks the calling thread until the specified thread terminates.
- The programmer can obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

Critical section in code

- Portion of code that should be executed by only thread at a time
- Implementation: bracket critical section with lock/unlock
- Can be used to implement atomic updates to anything
- Coarse-grain locking
  - not the right solution for parallelism but it is a start



Mutex-locks

- Lock is implemented by
  - variable with two states: *available* or *not_available*
  - queue that can hold ids of threads waiting for the lock
- Lock acquire:
  - If lock is *available,* it is changed to *not_available,* and control returns to application program
  - If lock is *not_available*, thread is queued up at the lock, and control returns to application program only when lock is acquired by that thread
  - Key invariant: once a thread tries to acquire lock, control returns to thread only after lock has been awarded to that thread
- Lock release:
  - next thread in queue is informed it has acquired lock
- Fairness: thread that wants lock gets it even if other threads want to acquire lock unbounded number of times

Pthreads API

- Type

```
pthread_mutex_t
```

- Lock initialization

```
int pthread_mutex_init(
        pthread_mutex_t *mutex_lock,
        const pthread_mutexattr_t *lock_attr);
```

- Acquiring lock

```
int pthread_mutex_lock(
            pthread_mutex_t *mutex_lock);
```

- Releasing lock

```
int pthread_mutex_unlock (
            pthread_mutex_t *mutex_lock);
```

Spin-locks/trylocks

- Another kind of lock: spin-lock, trylock
- Lock acquire is different from mutex: if lock is available, acquire it; otherwise return a "busy" error code (EBUSY)

```
int pthread_mutex_trylock(
        pthread_mutex_t *mutex_lock);
```

- Faster than `pthread_mutex_lock` on typical systems when there is no contention since it does not have to deal with queues associated with locks

Implementing locks using swap

- Recall: swap(addr,reg)
  - swap contents of address and register atomically
- Spin-lock using swap (test-and-set spin-lock)
  - variable L has 0/1 for unlocked/locked
  - Trylock code:

```
rx ← 1;
swap(L,rx);
return rx; //if returned value = 0 you have lock else not
```

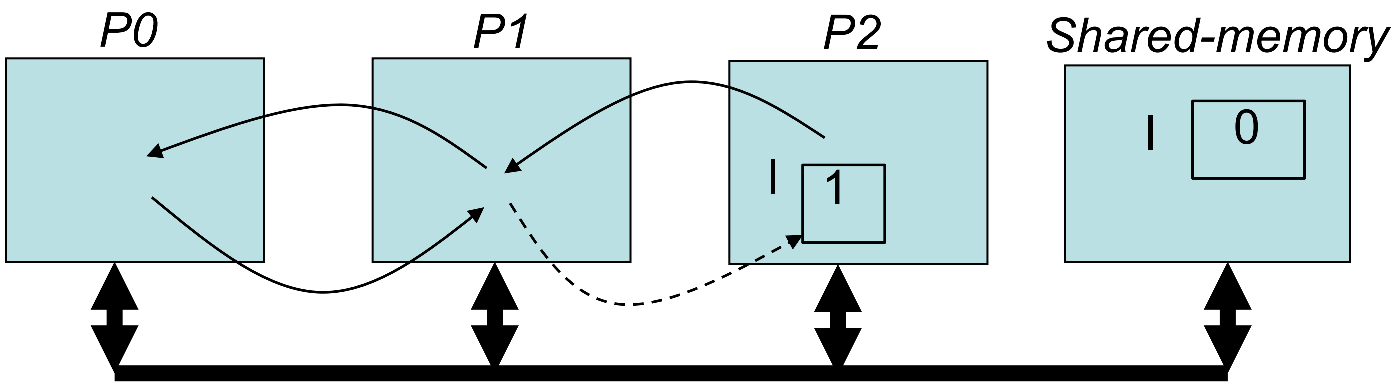
  - unlock

```
L ← 0;
```

- Problem:
  - swap must invalidate line in all caches even when lock acquire is not successful
  - if there are a lot of threads waiting for lock, busy-waiting will create a lot of bus traffic;

Busy-waiting and bus traffic

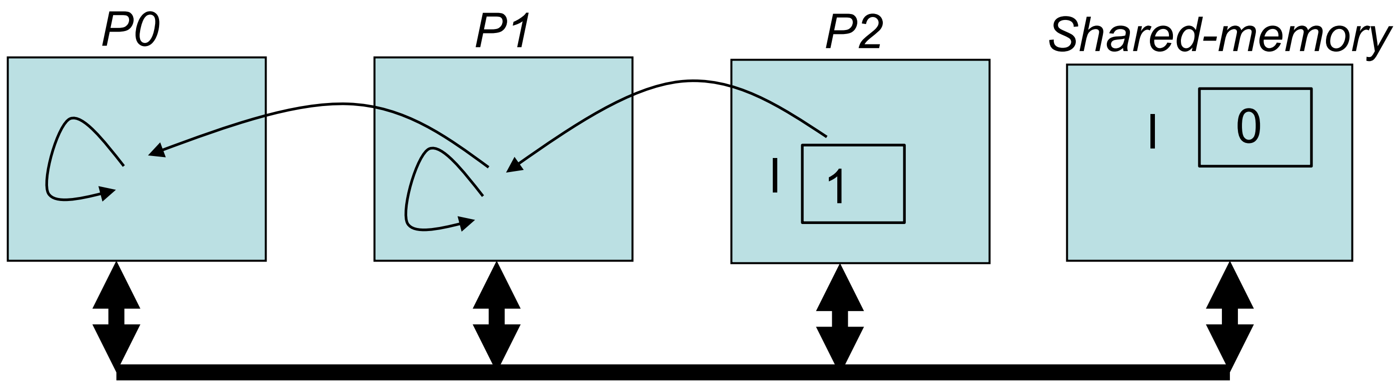
```
…..
        mov edx,1
acq: swap [l], edx
        test edx, edx
        jnz acq
…..
```



- Busy-waiting creates a lot of bus traffic
- Sequence of actions
  - all threads try to do swap
  - P2 wins and gets lock
  - P0 and P1 keep doing swap operations, invalidating line in other caches
  - P2 releases lock by writing 0 to lock
  - ….
- Solution: test-and-test-and-set
  - keep doing ordinary reads until lock is 0
  - then go into acq loop and see if you can get lock
  - if you fail, jump back to read loop

Better spin-locks: test-and-test-and-set

```
…..
        mov edx,1
 spin: mov eax, [l]
        test eax, eax
        jnz spin
        swap [l], edx
        test edx, edx
        jnz spin
…..
```



- Inner spin loop does not create bus traffic since all spinning threads spin on their local caches
- When P2 unlocks, line is invalidated from P0 and P1

Barriers

- Pthreads barrier type
  - pthread_barrier_t   varBarrier;
  - basically a struct
    - int total: initialized to # of threads to wait for
    - int count: tracks how many threads have reached barrier
    - mutex
- Initialize barrier
  - int pthread_barrier_init (&varBarrier,NULL,total);
- Waiting at barrier
  - int pthread_barrier_wait (&varBarrier);

Implementation of barriers

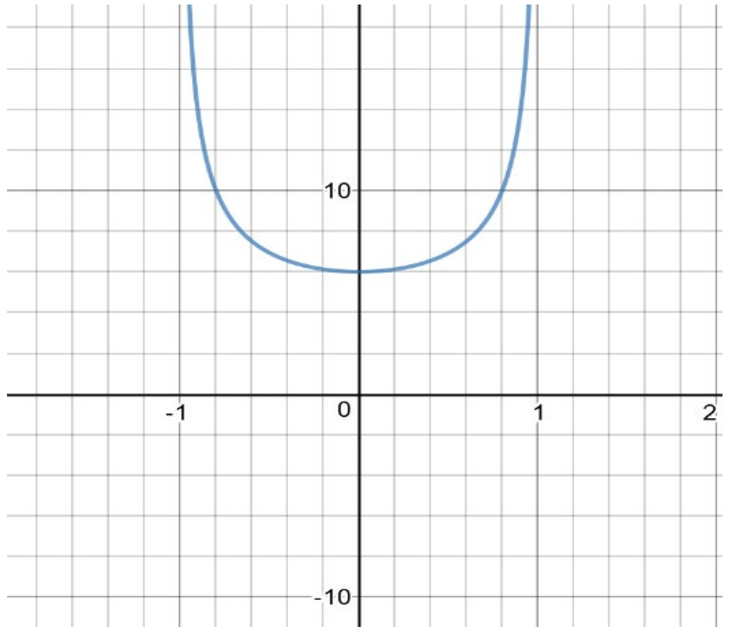
- Implemented using an atomic counter
  - Initialized to number of threads that need to arrive at barrier
  - Thread that arrives at barrier
    - decrements counter atomically
    - checks if it is the last one to arrive at barrier (counter = 0) and if so, informs other waiting threads that they can move past barrier
  - Small subtlety when barrier is within a loop

Shared-memory programming

Issues in shared-memory programming

- Performance problems
  - Load-balancing
    - Each thread must be assigned roughly same amount of work
    - Straggler problem: faster threads have to wait for slower threads at barrier
  - True and false sharing
    - Serialization bottleneck due to contention
- Correctness problems
  - Deadlocks and livelocks
    - May happen when threads need two or more locks to enter critical sections
  - Race conditions
    - Incorrectly synchronized read/modify/write
    - May result in nondeterministic output: output can depend on thread execution order

Application: numerical integration



$$f(x) = \frac{6}{\sqrt{1-x^2}}$$

- Estimate value of π using numerical integration $\int_0^{1/2} f(x)dx = \pi$
- Divide interval [0,1/2) into steps of equal size h and compute

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i * h) * h$$

Abstraction

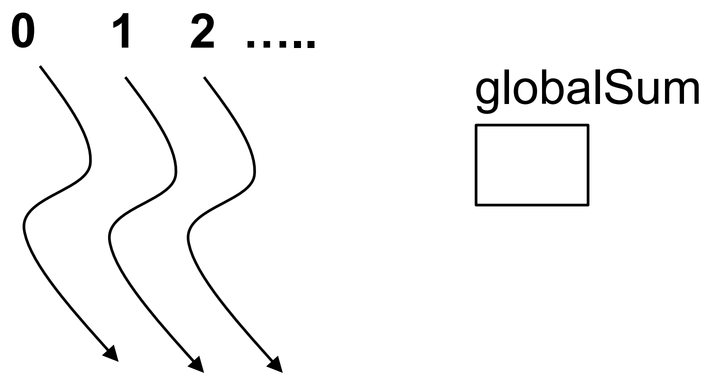
$$sum = \Sigma_{i=1}^{n} f(i)$$

- Parallelism:
  - map: function evaluations *f(i)* can be done in parallel
  - reduce: if addition is associative, *f(i)* values can be summed in parallel in O(log(n)) steps
    - we will not worry about exploiting this parallelism
- We will write several pThreads programs to illustrate the concepts we have studied

Solution (I)

- Distribution of work
  - round-robin with p threads
  - thread t computes values for i = t,t+p,t+2p..
  - load-balancing, assuming all evaluations of f take roughly same amount of time
- Single global variable globalSum
- Whenever thread computes a value, it adds it to global variable
- Preventing data races
  - use a mutex-lock

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$



Code

```
#include <pthread.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#define MAX_THREADS 512

pthread_t handles[MAX_THREADS];
int threadArg[MAX_THREADS];
double globalSum = 0.0;
pthread_mutex_t globalSum_lock;

void *compute_pi (void *);

int numPoints;
int numThreads;
double step;

double f(double x) {
  return (6.0/sqrt(1-x*x));
}
```

```
int main(int argc, char *argv[]) {

  pthread_attr_t attr;
  pthread_attr_init (&attr);

  numPoints = 100000000;
  step = 0.5/numPoints;
  numThreads = atoi(argv[1]); //number of threads is an input

  //create threads and initialize sum array
  for (int i=0; i< numThreads; i++) {
    threadArg[i] = i;
    pthread_create(& handles[i],&attr,compute_pi,& threadArg[i]);
  }

  //join with threads and add their contributions from sum array
  for (int i=0; i< numThreads; i++) {
    pthread_join(handles[i], NULL);
  }
  printf("%f\n", globalSum);
  return 0;

}
```

```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;

  for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     double value = step*f(x);
     pthread_mutex_lock(&globalSum_lock);
         globalSum = globalSum + value;  // Add to globalSum
     pthread_mutex_unlock(&globalSum_lock);
   }
```

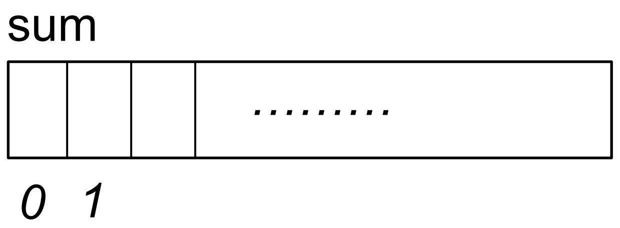
Performance

- Computation of each value added to globalSum takes little time
  - lock/add/unlock will be serial bottleneck
- We can replace critical section by atomic add
  - but atomic adds must be done serially, so serial bottleneck is still there
- In both solutions, you will also have a lot of cache line ping-ponging
- Problem: true-sharing causes serialization

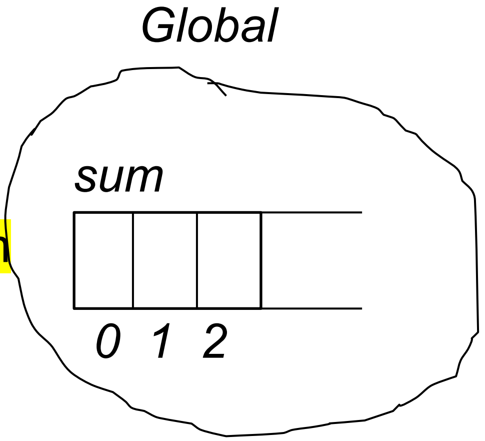
Solution (II)

- To avoid synchronization, create a global array sum
- Thread t
  - adds each value into sum[t] where sum is a global array
- Main thread joins with each worker thread and reads its contribution from sum array
- Main thread prints answer after joining with all worker threads

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i * h) * h$$



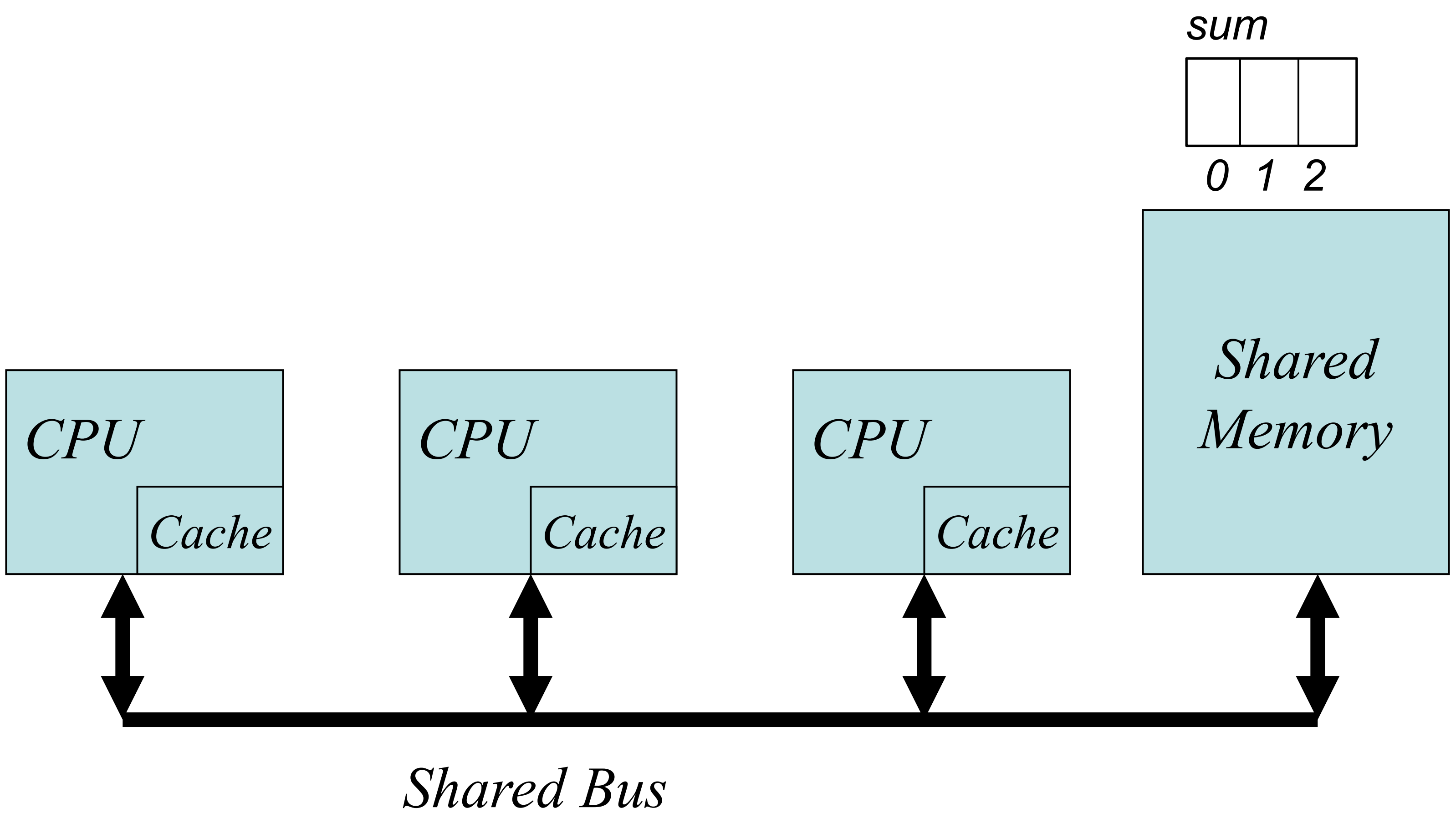
```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;
 for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     sum[myId] = sum[myId] + step*f(x);  // Add to local sum
  }
}
```



Code for main thread must add up values in sum array.

```
………
 for (int i=0; i< numThreads; i++) {
    pthread_join(handles[i], NULL);
    pi += sum[i];
 }
……..
```

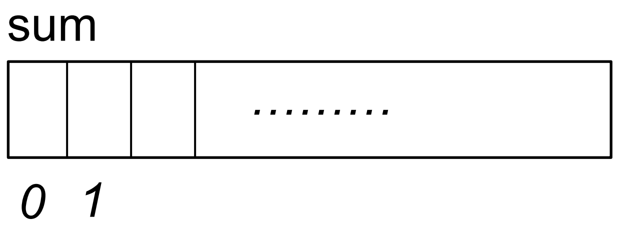
Problem: false-sharing



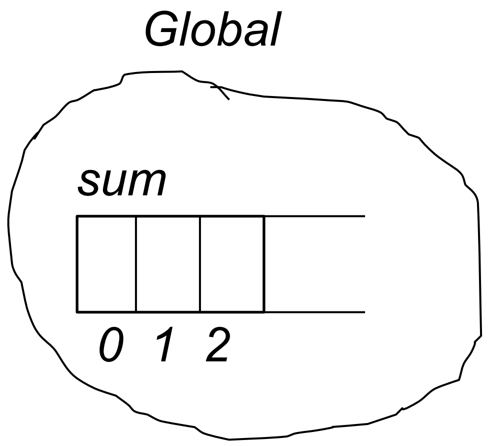
Solution (III)

- Thread t
  - computes values for i = t, t+P,t+2P,…
  - adds each value into a local variable of thread
  - when it is done, it writes the final value into sum[t]
- Main thread joins with each worker thread and reads its contribution from sum array
- Main thread prints answer after joining with all worker threads

$$\sum_{i=0}^{\frac{1}{2h}-1} f(i*h)*h$$



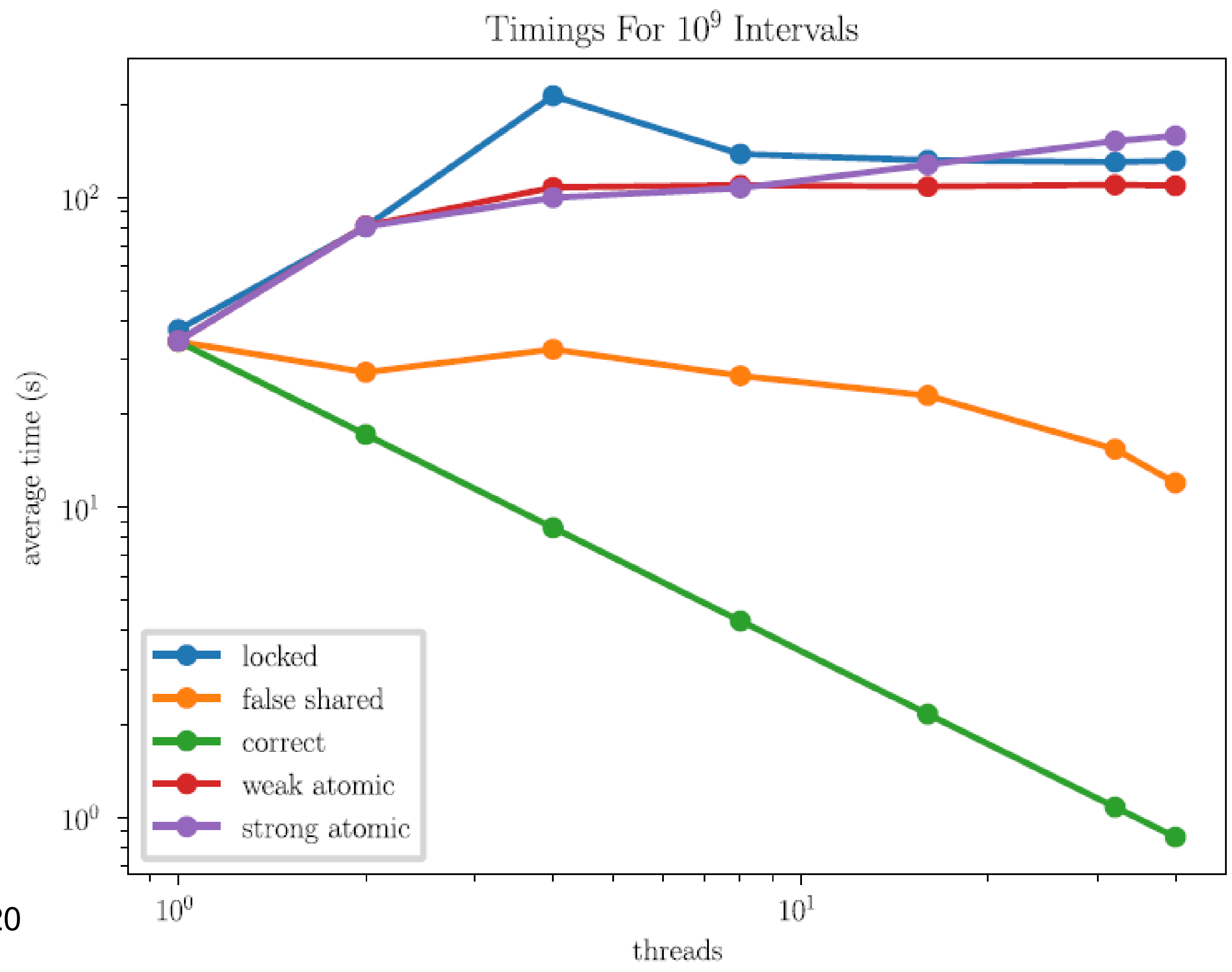
```
void *compute_pi (void *threadIdPtr) {
  int myId = *(int *)threadIdPtr;

 double mySum =0.0;
 for (int i = myId; i < numPoints; i+=numThreads) {
     double x = step * ((double) i);  // next x
     mySum = mySum + step*f(x);  // Add to local sum
  }
  sum[myId] = mySum; //write to global sum array
}
```



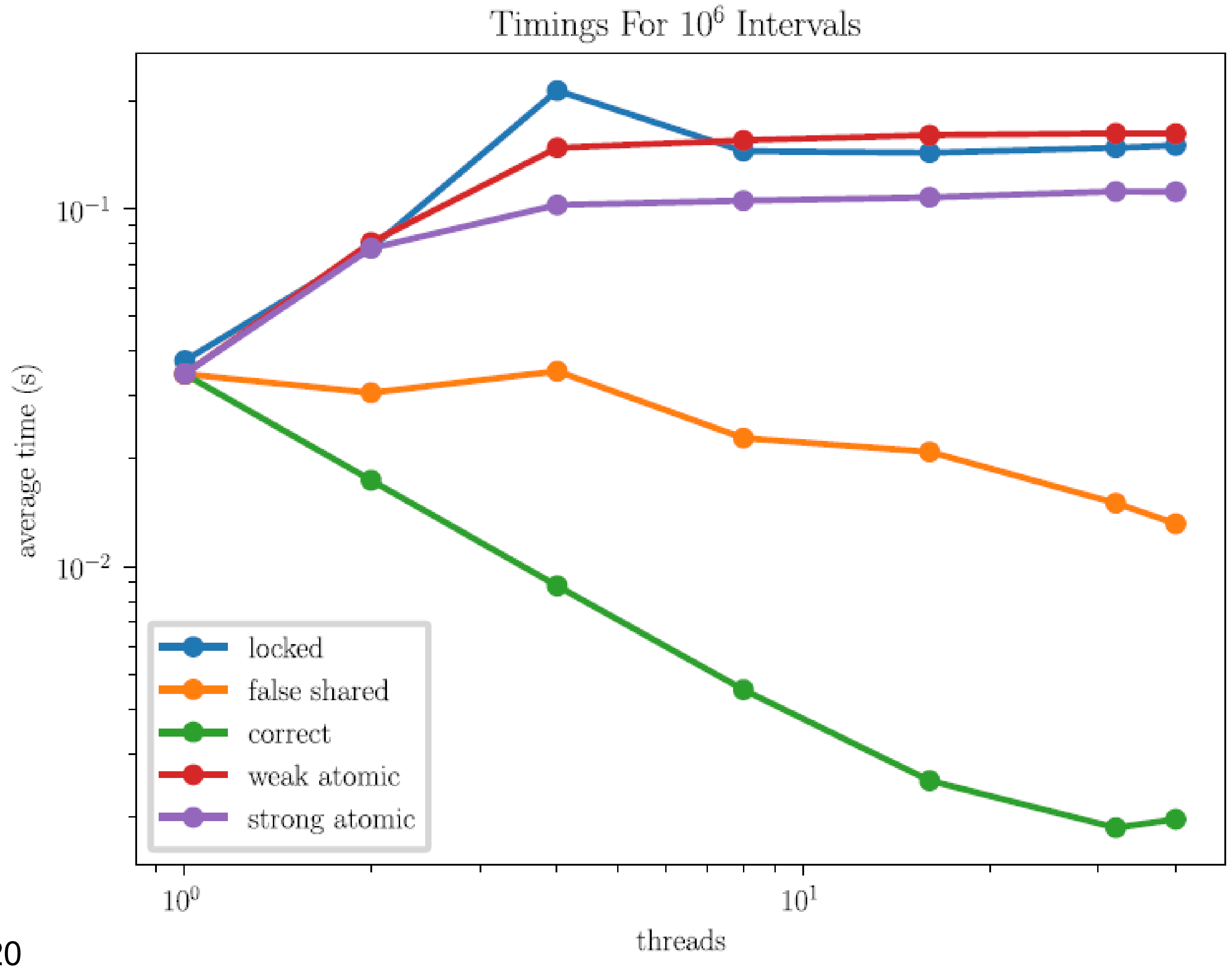
Numerical Integration Versions

- We saw three versions of program to compute pi
  - Version 1: summation in global variable
  - Version 2: summation in sum array
  - Version 3: local summation + update sum array
- Which version will perform best?
  - Version 1: true-sharing leads to many coherence misses + serialization in global variable updates
  - Version 2: false-sharing leads to many coherence misses

Performance



Performance



Correctness problems: deadlock and livelock

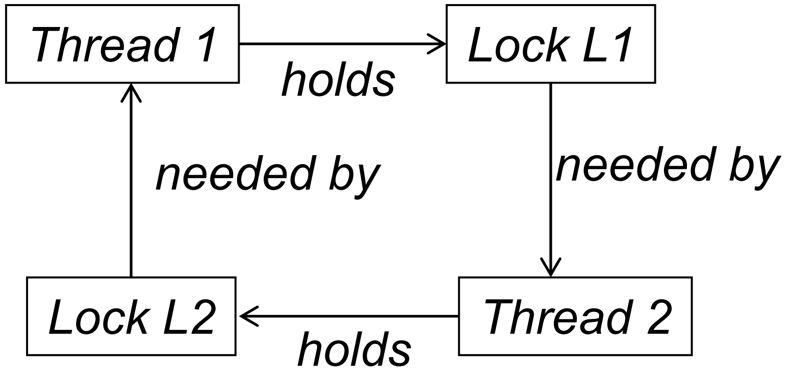
Problems with locks

- Locks are most dangerous when a thread needs to acquire multiple locks before releasing locks
- Two main problems:
  - deadlock
  - livelock
- Deadlock:
  - Threads A and B need locks L1 and l2
  - Thread A acquires L1 and wants L2
  - Thread B acquires L2 and wants L1
  - In general, there will be a cycle of threads in which each thread holds some locks and is waiting for locks held by other threads in the cycle
- Livelock:
  - may arise in some solutions to deadlock such as use of spinlocks

Deadlock

- Code snippet shows example of possible deadlock
- Subtle point:
  - deadlock may happen in some executions and not in others!
- "Deadly embrace": Dijkstra
- How do we ensure deadlocks cannot occur?

```
Thread 1:              Thread 2:
…                      …
lock(L1);              lock(L2);
lock(L2);              lock(L1);
….                     …
```



Deadlock: four conditions

- Mutual exclusion:
  - thread has exclusive control over resource it acquires
- Hold-and-wait:
  - thread acquire locks incrementally
- Circular wait:
  - there is a cycle of threads such that each thread holds one or more locks needed by the next thread in the cycle
- No pre-emption:
  - no external agency to force a thread to release locks if thread is waiting for another lock

You prevent deadlocks by ensuring that one or more of these conditions cannot arise in your program.

Prevent hold-and-wait

- Locks cannot be acquired incrementally
- One implementation:
  - single global lock to get permission to acquire locks you need
- Problem:
  - not scalable
  - conflicts with modularity and encapsulation
- You might encounter a hidden version of this problem if thread has to enter the kernel to perform some function like storage allocation
  - kernel lock is like the global-lock in our example

```
…
lock(global-lock);
lock(l1);
lock(l2);
unlock(global-lock);
…
```

Prevent circular wait

- Assign a logical total order to locks
  - (eg) name them L1,L2,L3,…
- Ensure that threads will never try to acquire a lower numbered lock while holding a higher numbered lock
  - (eg) if thread owns L3, it can try to acquire L4, L5, L6,… but it cannot try to acquire locks L1 or L2 (unless it already owns them and locks are re-entrant)
- Useful software engineering principle when you have control over the entire code base and you know what locks are required where
- However
  - easy to make mistakes
  - tension with encapsulation:
    - requires detailed knowledge of entire code base

Self-preemption

- Coding discipline:
  - Use only try-locks
  - If a thread cannot acquire a lock while it is holding other locks, it releases all locks it holds and tries again
  - Variation: OS or some other agency steps in and preempts a thread
- Problems:
  - Encapsulation
  - Livelock: threads can keep on acquiring and releasing locks without making progress because no thread ever gets all the locks it needs
  - One solution to livelock: (Ethernet) backoff: thread does not retry until some randomly chosen amount of time has passed

```
loop:
//start of lock acquires
….
if (trylock(Lj) == EBUSY) {
//unlock all locks you hold
goto loop;
}
….
endloop:

//compute with resources
//release locks
```

Summary

- Architecture
  - cache coherence
  - atomic instructions
  - memory consistency model
- The POSIX Thread API
  - creating and destroying threads
  - synchronization
    - join
    - mutual exclusion: locks and spin-locks
    - intrinsics for atomic instructions
    - barrier
- Performance:
  - minimize false and true sharing
  - keep critical sections small

ADDITIONAL MATERIAL

Spinlock example in x86

```
global main

extern printf
extern pthread_create
extern pthread_exit
extern pthread_join

section .data
            align 4
            sLock:                  dd 0        ; The lock, values are:
                                                ; 0         unlocked
                                                ; 1         locked
            tID1:                   dd 0
            tID2:                   dd 0
            fmtStr1:    db "In thread %d with ID: %02x", 0x0A, 0
            fmtStr2:    db "Result %d", 0x0A, 0

section .bss
            align 4
            result:                 resd 1
```

```
section .text
        main:                                   ; Using main since we are using gcc to link

                                                ; Call pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                                                ;                     void *(*start_routine) (void *), void *arg);

        push        dword 0                     ; Arg Four: argument pointer
        push        thread1                     ; Arg Three: Address of routine
        push        dword 0                     ; Arg Two: Attributes
        push        tID1                        ; Arg One: pointer to the thread ID
        call        pthread_create

        push        dword 0                     ; Arg Four: argument pointer
        push        thread2                     ; Arg Three: Address of routine
        push        dword 0                     ; Arg Two: Attributes
        push        tID2                        ; Arg One: pointer to the thread ID
        call        pthread_create

                                                ; Call int pthread_join(pthread_t thread, void **retval) ;
                                                ;
        push        dword 0                     ; Arg Two: retval
        push        dword [tID1]  ; Arg One: Thread ID to wait on
        call        pthread_join
        push        dword 0                     ; Arg Two: retval
        push        dword [tID2]  ; Arg One: Thread ID to wait on
        call        pthread_join

        push        dword [result]
        push        dword fmtStr2
        call        printf
        add         esp, 8                      ; Pop stack 2 times 4 bytes

        call exit
```

```
thread1:
            pause
            push        dword [tID1]
            push        dword 1
            push        dword fmtStr1
            call        printf
            add         esp, 12                 ; Pop stack 3 times 4 bytes

            call        spinLock

            mov         [result], dword 1
            call        spinUnlock

            push        dword 0                 ; Arg one: retval
            call        pthread_exit

thread2:
            pause
            push        dword [tID2]
            push        dword 2
            push        dword fmtStr1
            call        printf
            add         esp, 12                 ; Pop stack 3 times 4 bytes

            call        spinLock

            mov         [result], dword 2
            call        spinUnlock

            push        dword 0                 ; Arg one: retval
            call        pthread_exit
```

```
spinLock:
            push        ebp
            mov         ebp, esp
            mov         edx, 1                  ; Value to set sLock to
spin:       mov         eax, [sLock]  ; Check sLock
            test        eax, eax      ; If it was zero, maybe we have the lock
            jnz         spin                    ; If not try again
            ;
            ; Attempt atomic compare and exchange:
            ; if (sLock == eax):
            ;           sLock         <- edx
            ;           zero flag     <- 1
            ; else:
            ;           eax           <- edx
            ;           zero flag     <- 0
            ;
            ; If sLock is still zero then it will have the same value as eax and
            ; sLock will be set to edx which is one and therefore we aquire the
            ; lock. If the lock was acquire between the first test and the
            ; cmpxchg then eax will not be zero and we will spin again.
            ;
            lock        cmpxchg [sLock], edx   ;eax is implicit operand
            test        eax, eax
            jnz         spin
            pop         ebp
            ret
spinUnlock:
            push        ebp
            mov         ebp, esp
            mov         eax, 0
            xchg        eax, [sLock]
            pop         ebp
            ret
```

```
exit:
                                                ;
                                                ; Call exit(3) syscall
                                                ;void exit(int status)
                                                ;
          mov         ebx, 0                    ; Arg one: the status
          mov         eax, 1                    ; Syscall number:
          int         0x80
```

Summary

- Architecture
  - cache coherence
  - atomic instructions
  - memory consistency model
- The POSIX Thread API
  - creating and destroying threads
  - synchronization
    - join
    - mutual exclusion: locks and spin-locks
    - intrinsics for atomic instructions
    - barrier
- Performance:
  - minimize false and true sharing
  - keep critical sections small