

# **CSC 252: Computer Organization**

## **Spring 2021: Lecture 25**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

# Announcements

SUN 25	MON 26	TUE 27	WED 28	THU 29	FRI 30	SAT May 1
2	3	4 <b>Today</b>	5 <b>A5 Due</b>	6 <b>Last Lecture</b>	7	8
9	10	11	12 <b>Final</b>	13	14	15

# Announcements

- Final exam: **May 12, 19:15 PM -- 22:15 PM**; online.
- Past exam & Problem set: <https://www.cs.rochester.edu/courses/252/spring2021/handouts.html>
- Exam will be electronic using Gradescope, but we will send you an PDF version so that you can work offline in case
  - 1) you don't have Internet access at the exam time or
  - 2) you lose Internet access.
  - Write down the answers on a scratch paper, take pictures, and send us the pictures

# Announcements

- Open book test: any sort of paper-based product, e.g., book, **notes**, magazine, old tests.
- Exams are designed to test your ability to apply what you have learned and not your memory (though a good memory could help).
- **Nothing electronic (including laptop, cell phone, calculator, etc) other than the computer you use to take the exam.**
- **Nothing biological**, including your roommate, husband, wife, your hamster, another professor, etc.
- **“I don’t know”** gets 15% partial credit. Must erase everything else.

# Today

- From process to threads
  - Basic thread execution model
- **Multi-threading programming**
- Hardware support of threads
  - Single core
  - Multi-core
  - Cache coherence

# Shared Variables in Threaded C Programs

- One great thing about threads is that they can share same program variables.
- Question: Which variables in a threaded C program are shared?
- Intuitively, the answer is as simple as “*global variables are shared*” and “*stack variables are private*”. Not so simple in reality.

## Shared code and data

### Thread 1 (main thread)

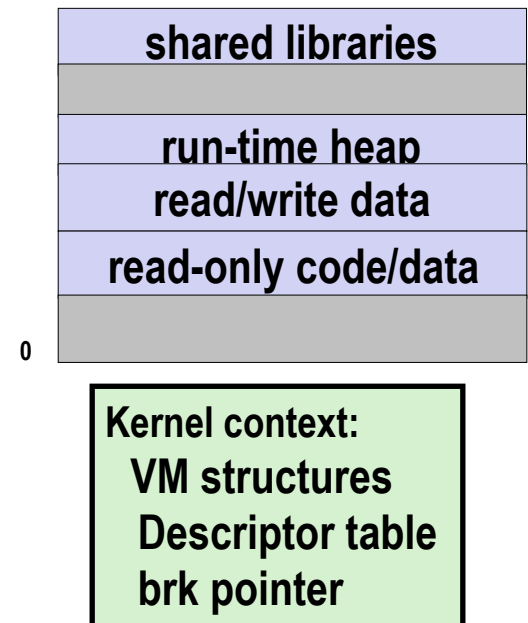
### Thread 2 (peer thread)

stack 1

stack 2

Thread 1 context:  
Data registers  
Condition codes  
SP1  
PC1

Thread 2 context:  
Data registers  
Condition codes  
SP2  
PC2



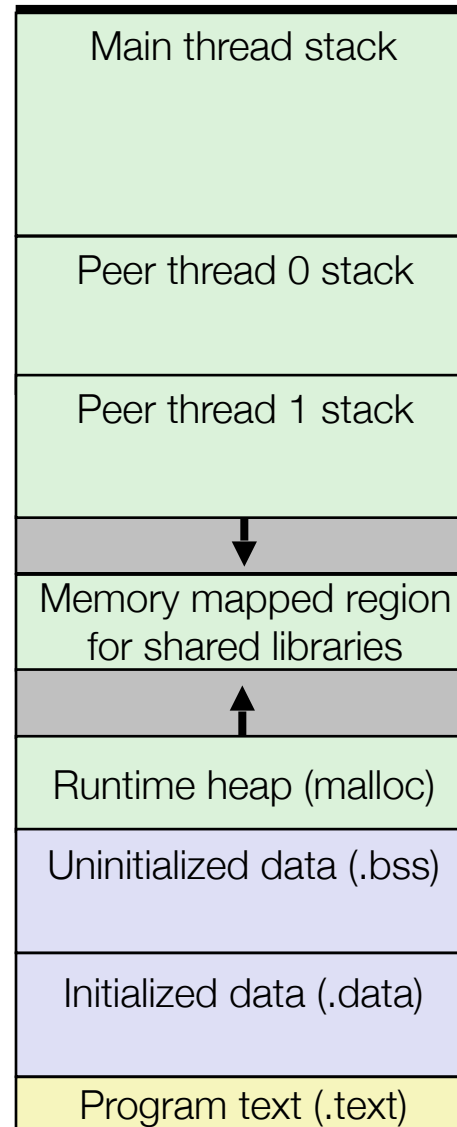
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



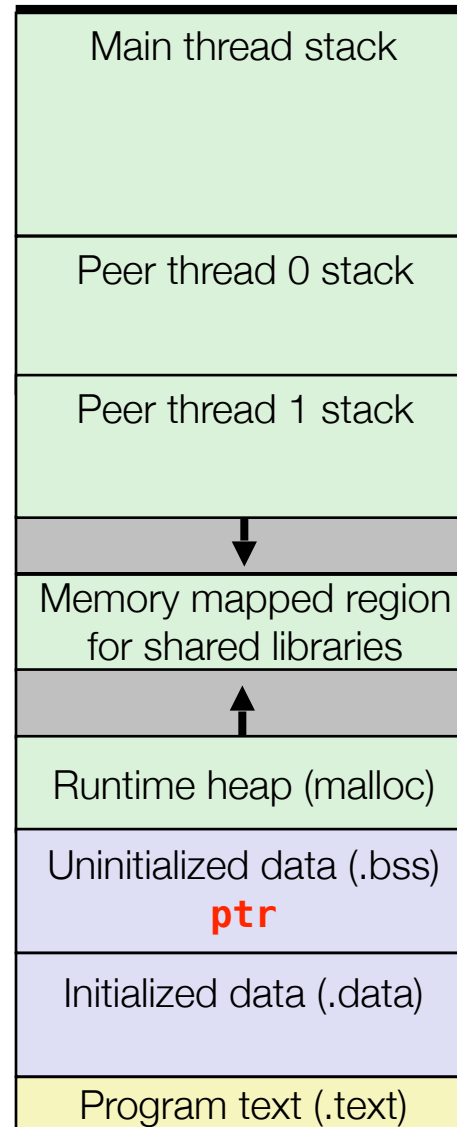
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c





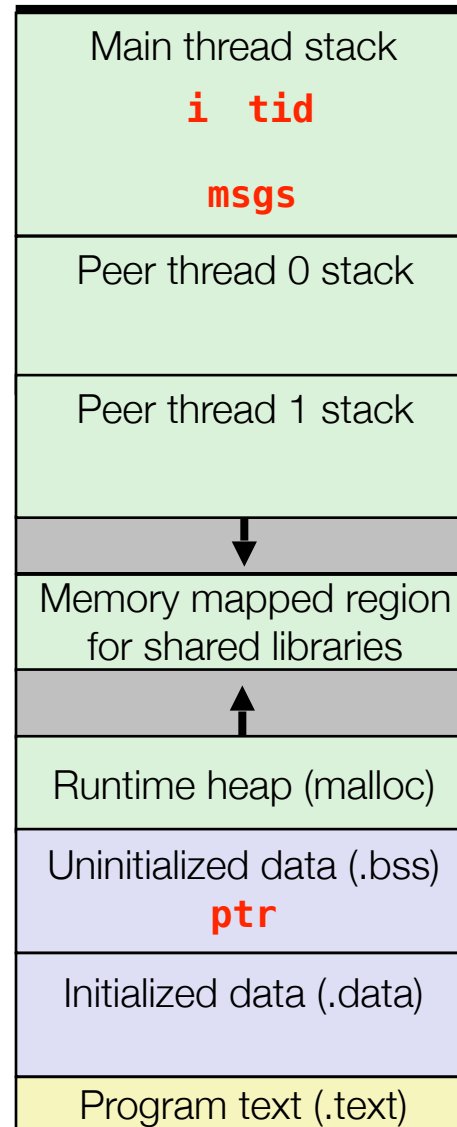
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



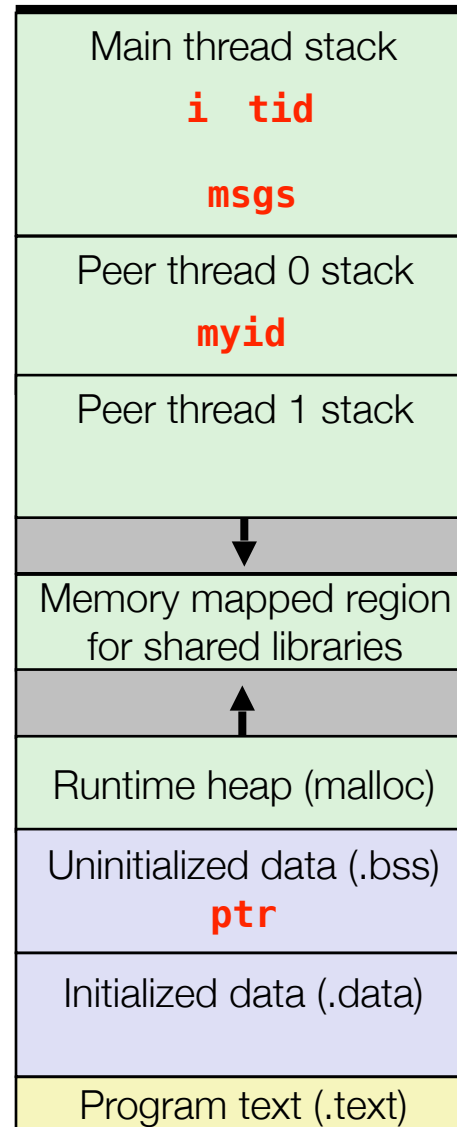
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                       NULL,
                       thread,
                       (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



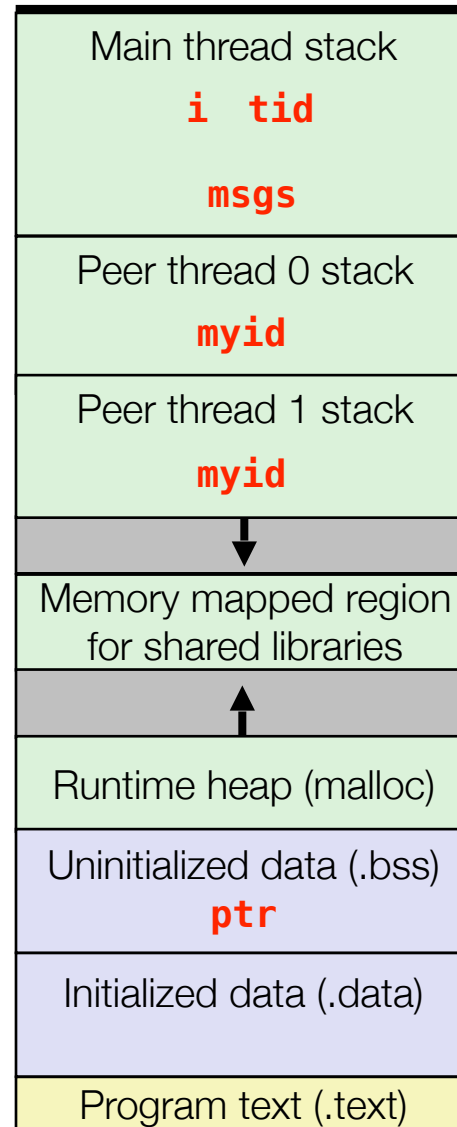
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



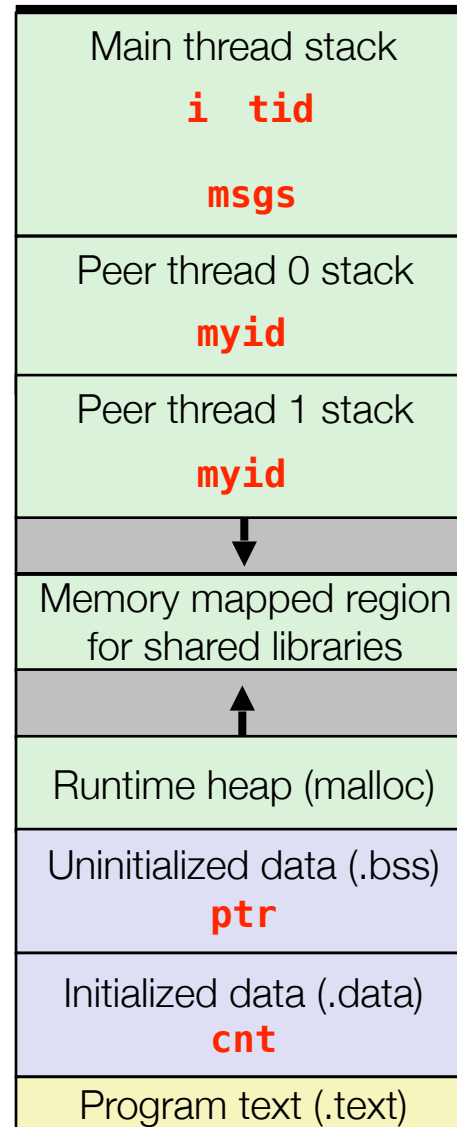
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



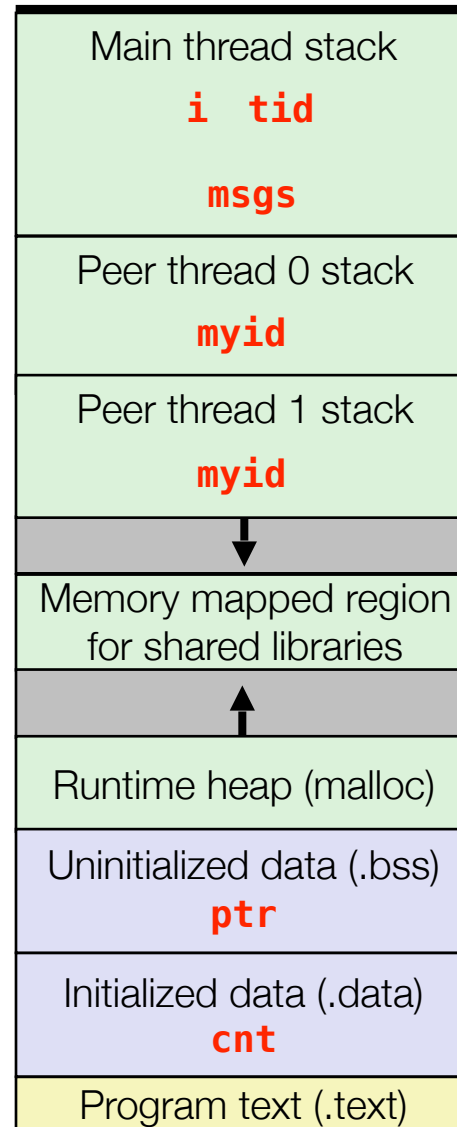
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



p0 p1 main

# Example Program to Illustrate Sharing

```

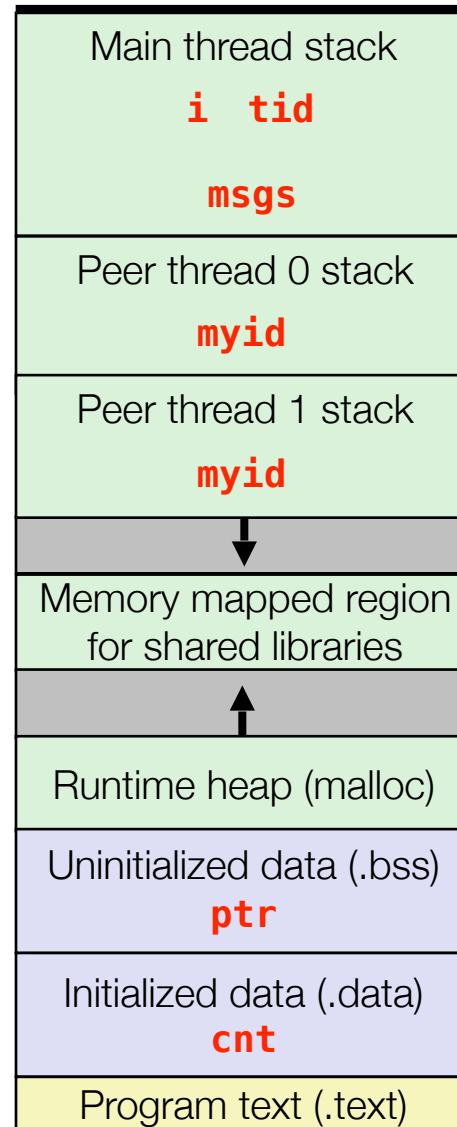
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}

```

sharing.c



p0 p1 main

p0 p1

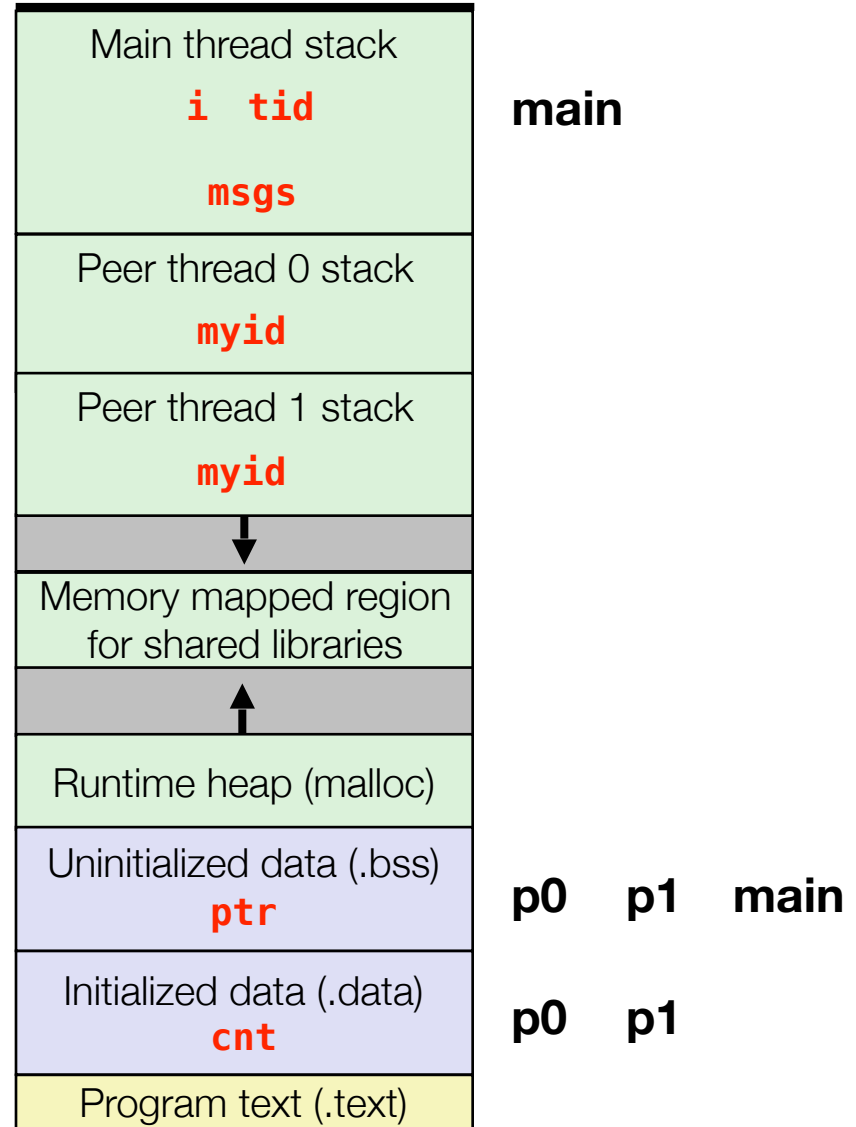
# Example Program to Illustrate Sharing

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



# Example Program to Illustrate Sharing

```

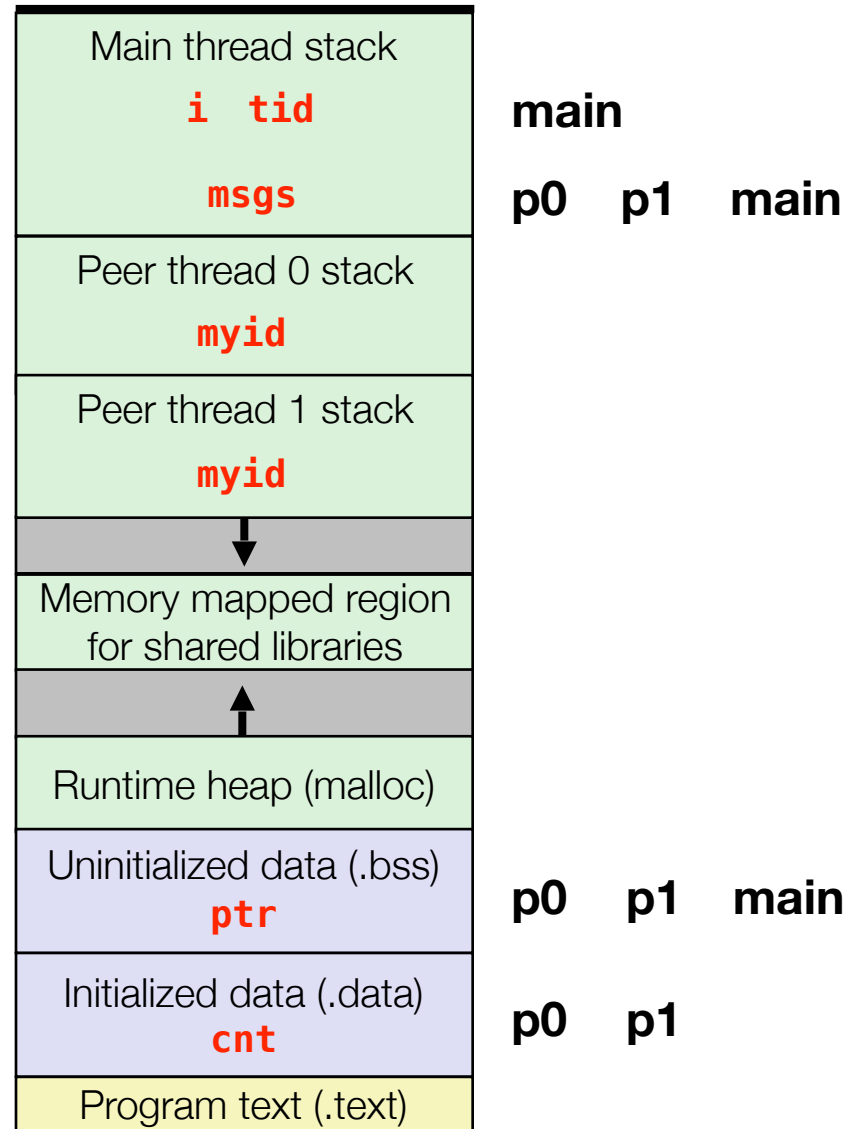
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                       NULL,
                       thread,
                       (void *)i);
    pthread_exit(NULL);
}

```

sharing.c





# Example Program to Illustrate Sharing

```

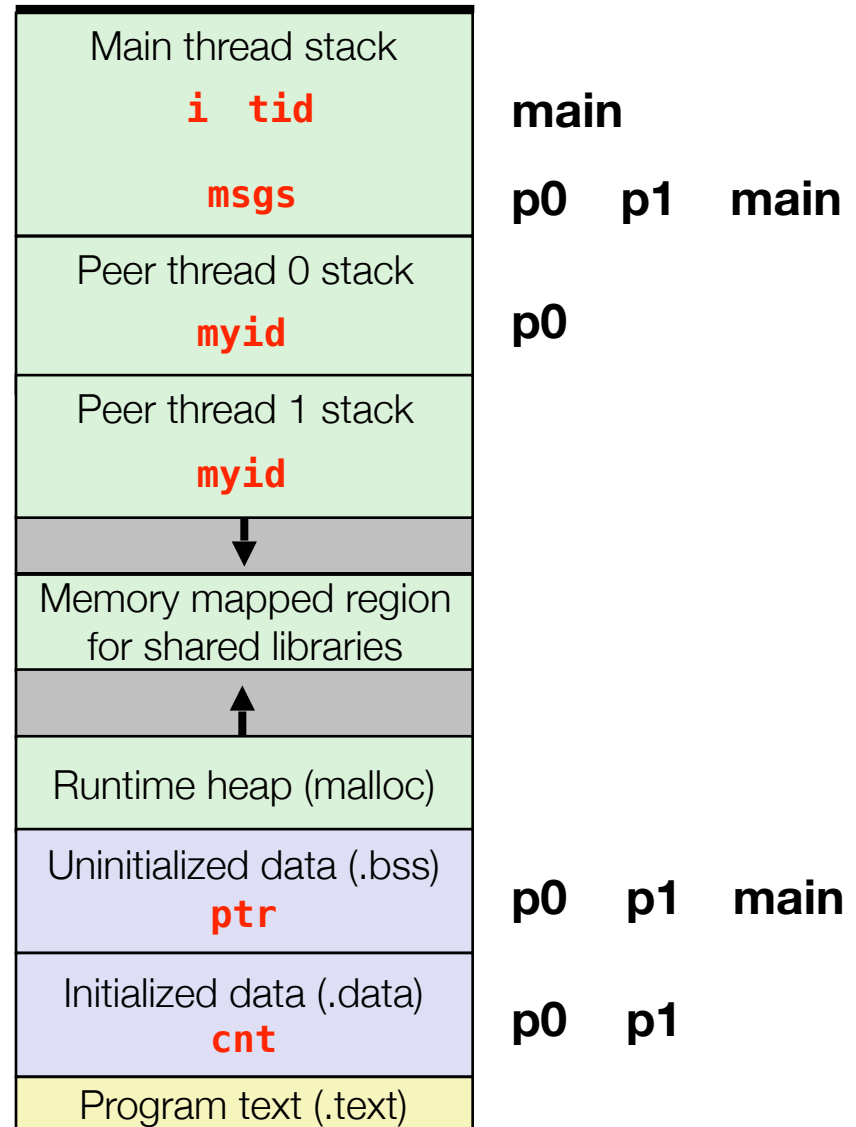
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                       NULL,
                       thread,
                       (void *)i);
    pthread_exit(NULL);
}

```

sharing.c



# Example Program to Illustrate Sharing

```

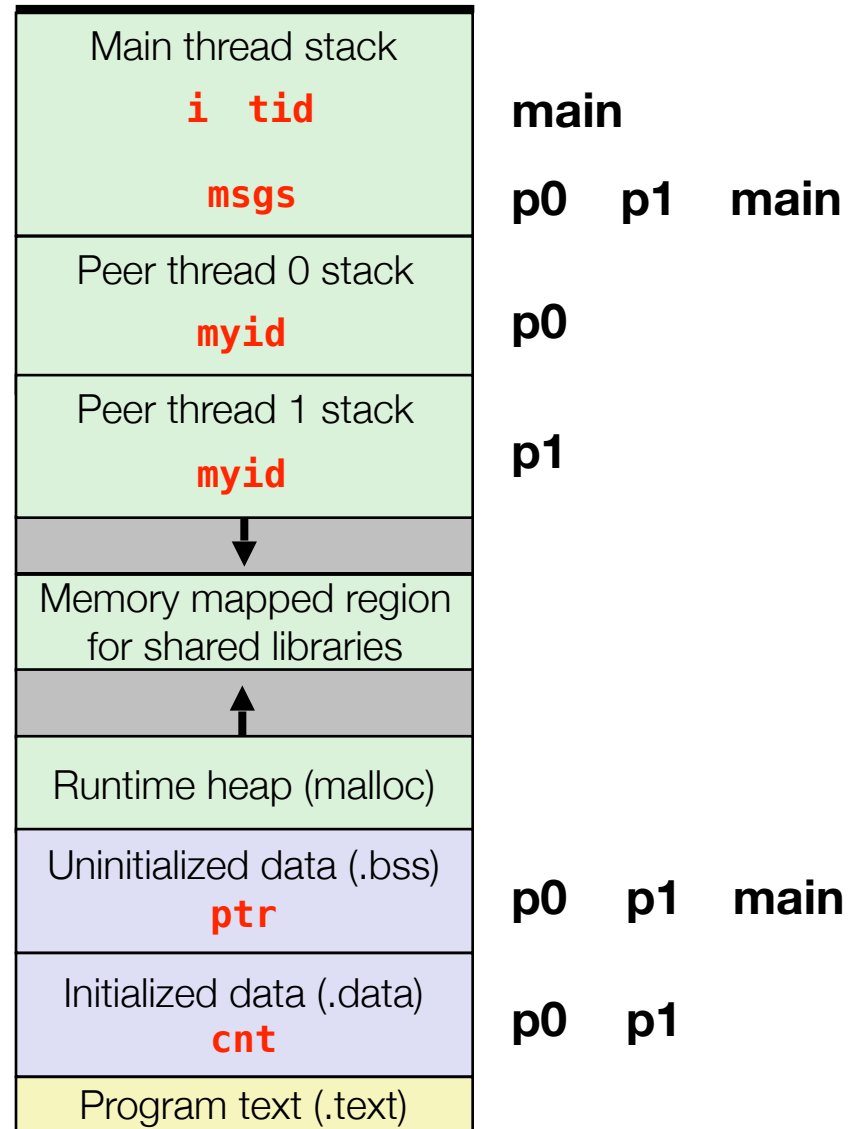
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
                      NULL,
                      thread,
                      (void *)i);
    pthread_exit(NULL);
}

```

sharing.c



# Synchronizing Threads

- Shared variables are handy...
- ...but introduce the possibility of nasty *synchronization* errors.

# Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;
    long niters = 10000;

    Pthread_create(&tid1, NULL,
                  thread, &niters);
    Pthread_create(&tid2, NULL,
                  thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * 10000))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
        *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

# Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    pthread_t tid1, tid2;
    long niters = 10000;

    Pthread_create(&tid1, NULL,
                  thread, &niters);
    Pthread_create(&tid2, NULL,
                  thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * 10000))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
        *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt
OK cnt=20000
```

```
linux> ./badcnt
BOOM! cnt=13051
```

**cnt should be 20,000.**

**What went wrong?**

# Assembly Code for Counter Loop

C code for counter loop in thread  $i$

```
for (i = 0; i < niters; i++)  
    cnt++;
```

## *Asm code for thread $i$*

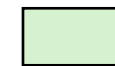
<pre>movq (%rdi), %rcx testq %rcx, %rcx jle .L2 movl \$0, %eax</pre>	}	$H_i$ : Head
<pre>.L3: movq cnt(%rip), %rdx addq \$1, %rdx movq %rdx, cnt(%rip)</pre>		
<pre>addq \$1, %rax cmpq %rcx, %rax jne .L3</pre>	}	$L_i$ : Load cnt $U_i$ : Update cnt $S_i$ : Store cnt
<pre>.L2:</pre>		
		$T_i$ : Tail

# Concurrent Execution

- **Key observation:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

i (thread)   instr<sub>i</sub>   %rdx<sub>1</sub>   %rdx<sub>2</sub>   cnt (shared)

1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
1	S <sub>1</sub>	1	-	1
2	L <sub>2</sub>	-	1	1
2	U <sub>2</sub>	-	2	1
2	S <sub>2</sub>	-	2	2



Thread 1  
critical section



Thread 2  
critical section

L <sub>i</sub>	movq	cnt(%rip), %rdx
U <sub>i</sub>	addq	\$1, %rdx
S <sub>i</sub>	movq	%rdx, cnt(%rip)

# Concurrent Execution (cont)

- A legal (feasible) but undesired ordering: two threads increment the counter, but the result is 1 instead of 2

**i (thread)   instr<sub>i</sub>   %rdx<sub>1</sub>   %rdx<sub>2</sub>   cnt (shared)**

1	L <sub>1</sub>	0	-	0
1	U <sub>1</sub>	1	-	0
2	L <sub>2</sub>	-	0	0
1	S <sub>1</sub>	1	-	1
2	U <sub>2</sub>	-	1	1
2	S <sub>2</sub>	-	1	1

<i>L<sub>i</sub></i>	<code>movq cnt(%rip), %rdx</code>
<i>U<sub>i</sub></i>	<code>addq \$1, %rdx</code>
<i>S<sub>i</sub></i>	<code>movq %rdx, cnt(%rip)</code>



# Assembly Code for Counter Loop

C code for counter loop in thread  $i$

```
for (i = 0; i < niters; i++)
    cnt++;
```

## *Asm code for thread $i$*

critical  
section  
wrt cnt

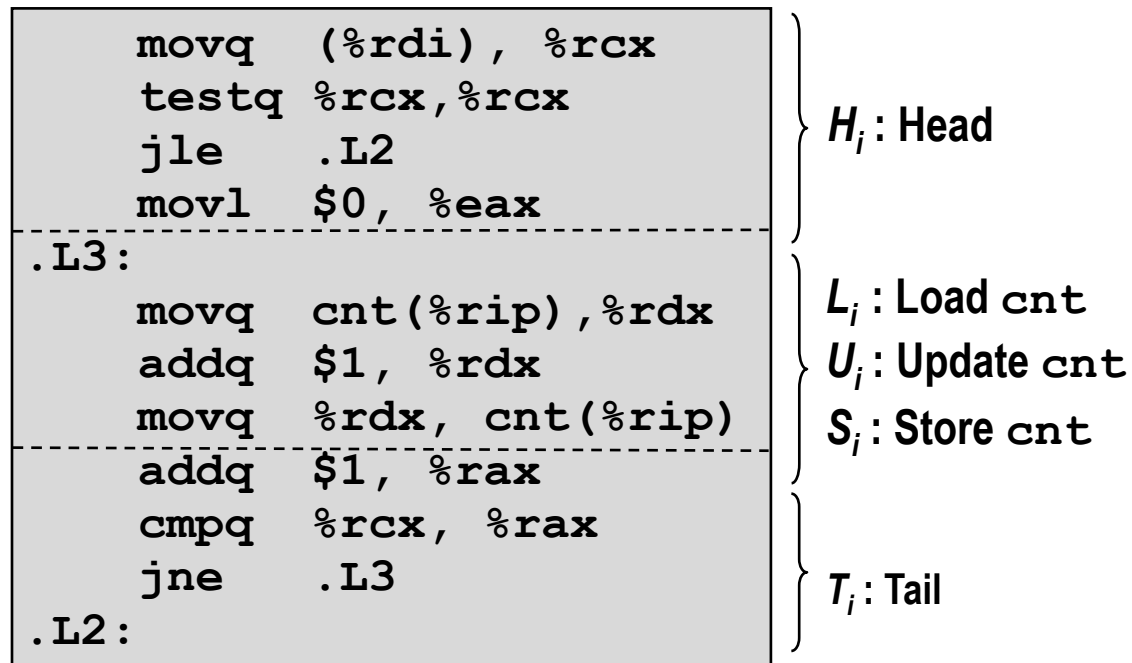


<pre>movq  (%rdi), %rcx testq %rcx,%rcx jle   .L2 movl  \$0, %eax</pre>	} $H_i$ : Head
<pre>----- .L3: movq  cnt(%rip), %rdx addq  \$1, %rdx movq  %rdx, cnt(%rip)</pre>	} $L_i$ : Load cnt $U_i$ : Update cnt $S_i$ : Store cnt
<pre>----- addq  \$1, %rax cmpq  %rcx, %rax jne   .L3</pre>	} $T_i$ : Tail
<pre>.L2:</pre>	

# Critical Section

- Code section (a sequence of instructions) where no more than one thread should be executing concurrently.
  - Critical section refers to code, but its intention is to protect data!

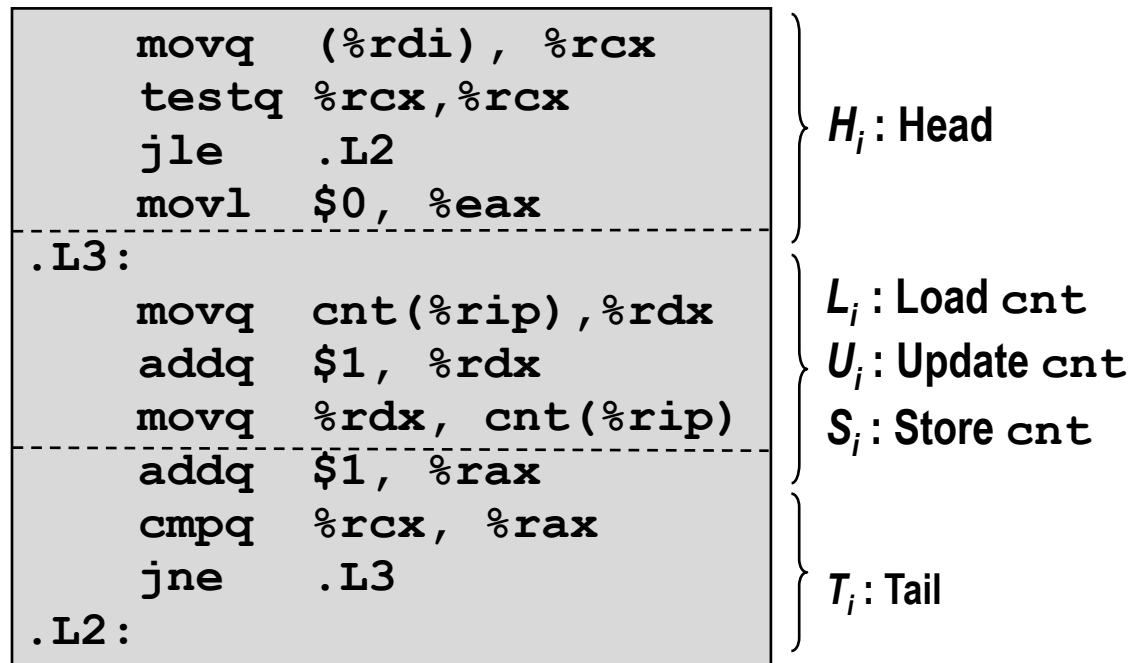
critical  
section  
wrt cnt



# Critical Section

- Code section (a sequence of instructions) where no more than one thread should be executing concurrently.
  - Critical section refers to code, but its intention is to protect data!
- Threads need to have **mutually exclusive** access to critical section. That is, the execution of the critical section must be **atomic**: instructions in a CS either are executed entirely without interruption or not executed at all.

critical  
section  
wrt cnt



# Enforcing Mutual Exclusion

- We must *coordinate/synchronize* the execution of the threads
  - i.e., need to guarantee ***mutually exclusive access*** for each critical section.
- Classic solution:
  - Semaphores/mutex (Edsger Dijkstra)
- Other approaches
  - Condition variables
  - Monitors (Java)
  - 254/258 discusses these

# Using Semaphores for Mutual Exclusion

- Basic idea:

# Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.

# Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.

# Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.



# Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
  - No more than one thread can be in the critical section at a time.

# Using Semaphores for Mutual Exclusion

- Basic idea:

- Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
- Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
- Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
- No more than one thread can be in the critical section at a time.

- Terminology

# Using Semaphores for Mutual Exclusion

- Basic idea:

- Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
- Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
- Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
- No more than one thread can be in the critical section at a time.

- Terminology

- **Binary semaphore** is also called **mutex** (i.e., the semaphore value could only be 0 or 1)

# Using Semaphores for Mutual Exclusion

- Basic idea:

- Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
- Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
- Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
- No more than one thread can be in the critical section at a time.

- Terminology

- **Binary semaphore** is also called **mutex** (i.e., the semaphore value could only be 0 or 1)
- Think of P operation as “**locking**”, and V as “**unlocking**”.

# Proper Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0; /* Counter */
sem_t mutex; /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with P and V:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

**Warning: It's orders of magnitude slower than badcnt.c.**

# Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?

```
for (i = 0; i < niters; i++) {  
    P(&mutex);  
    cnt++;  
    V(&mutex);  
}  
goodcnt.c
```

# Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.

```
for (i = 0; i < niters; i++) {  
    P(&mutex);  
    cnt++;  
    V(&mutex);  
}  
goodcnt.c
```

# Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.
  - Checking mutex value and setting its value must be an atomic unit: they either are performed entirely or not performed at all.

```
for (i = 0; i < niters; i++) {  
    P(&mutex);  
    cnt++;  
    V(&mutex);  
}  
  
goodcnt.c
```



# Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.
  - Checking mutex value and setting its value must be an atomic unit: they either are performed entirely or not performed at all.
  - on x86: the atomic test-and-set instruction.

```
for (i = 0; i < niters; i++) {  
    P(&mutex);  
    cnt++;  
    V(&mutex);  
}
```

goodcnt.c

# Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.
  - Checking mutex value and setting its value must be an atomic unit: they either are performed entirely or not performed at all.
  - on x86: the atomic test-and-set instruction.

```
for (i = 0; i < niters; i++) {  
    P(&mutex);  
    cnt++;  
    V(&mutex);  
}  
  
goodcnt.c
```

```
function Lock(boolean *lock) {  
    while (test_and_set(lock) == 1);  
}
```

# Deadlock

- Def: A process/thread is *deadlocked* if and only if it is waiting for a condition that will never be true
- General to concurrent/parallel programming (threads, processes)
- Typical Scenario
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!

# Deadlocking With Semaphores

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}

int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

**Tid[0]:**  
**P(s<sub>0</sub>);**  
**P(s<sub>1</sub>);**  
**cnt++;**  
**V(s<sub>0</sub>);**  
**V(s<sub>1</sub>);**

**Tid[1]:**  
**P(s<sub>1</sub>);**  
**P(s<sub>0</sub>);**  
**cnt++;**  
**V(s<sub>1</sub>);**  
**V(s<sub>0</sub>);**

# Avoiding Deadlock

*Acquire shared resources in same order*

```
Tid[0]:  
P(s0);  
P(s1);  
cnt++;  
V(s0);  
V(s1);
```

```
Tid[1]:  
P(s1);  
P(s0);  
cnt++;  
V(s1);  
V(s0);
```



```
Tid[0]:  
P(s0);  
P(s1);  
cnt++;  
V(s0);  
V(s1);
```

```
Tid[1]:  
P(s0);  
P(s1);  
cnt++;  
V(s1);  
V(s0);
```

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:



# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- OS decides to take the SIGCHLD interrupt and executes the handler

# Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- OS decides to take the SIGCHLD interrupt and executes the handler
- When return to parent process, **y == 20!**

# Fixing the Signal Handling Bug

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);

    exit(0);
}
```

- Block all signals before accessing a shared, global data structure.

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.
- Signal handler wants the mutex, which is acquired by the main program.

# How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.
- Signal handler wants the mutex, which is acquired by the main program.
- **Key:** signal handler is in the same process/thread as the main program. The kernel forces the handler to finish before returning to the main program.



# Summary of Multi-threading Programming

- Concurrent/parallel threads access shared variables
- Need to protect concurrent accesses to guarantee correctness
- Semaphores (e.g., mutex) provide a simple solution
- Can lead to deadlock if not careful
- Take CSC 254/258 to know more about avoiding deadlocks (and parallel programming in general)

# Thread-level Parallelism (TLP)

- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task

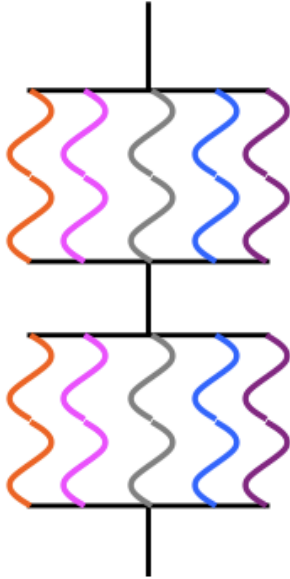
# Thread-level Parallelism (TLP)

- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task
- Example: Parallel summation of N number
  - Partition values  $1, \dots, n-1$  into  $t$  ranges,  $\lfloor n/t \rfloor$  values each range
  - Each of  $t$  threads processes one range (sub-task)
  - Sum all sub-sums in the end

# Thread-level Parallelism (TLP)

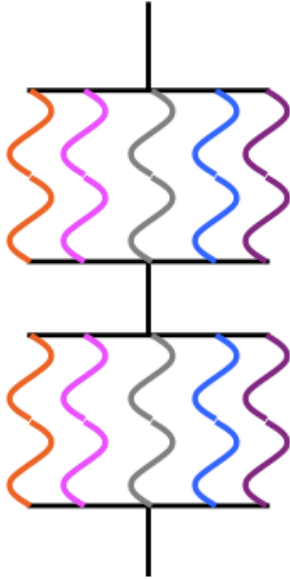
- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task
- Example: Parallel summation of N number
  - Partition values  $1, \dots, n-1$  into  $t$  ranges,  $\lfloor n/t \rfloor$  values each range
  - Each of  $t$  threads processes one range (sub-task)
  - Sum all sub-sums in the end
- Question: if you parallel you work N ways, do you always an N times speedup?

# Why the Sequential Bottleneck?



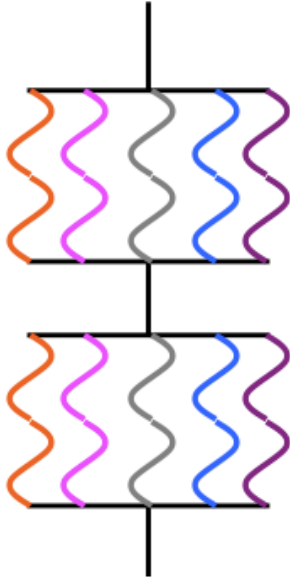
- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

# Why the Sequential Bottleneck?

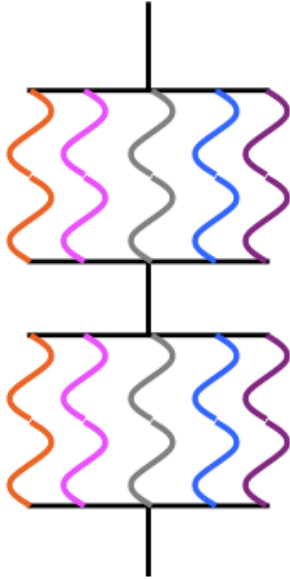


- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {  
    Compute  
    P(A)  
    Update shared data  
    V(A)  
}
```

# Why the Sequential Bottleneck?



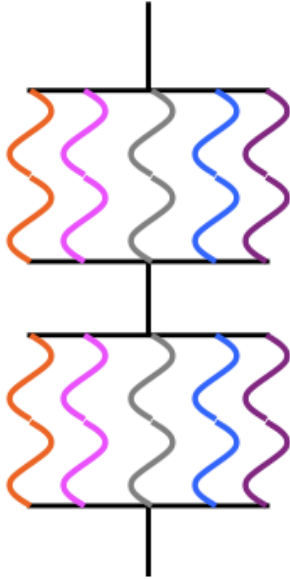
- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {  
  Compute N  
  P(A)  
  Update shared data  
  V(A)  
}
```



# Why the Sequential Bottleneck?

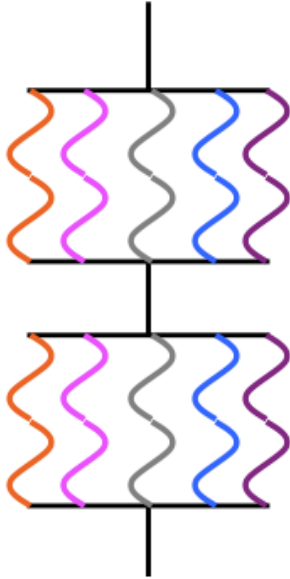


- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {  
    Compute N  
    P(A)  
    Update shared data  
    V(A) C  
}
```

# Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {
```

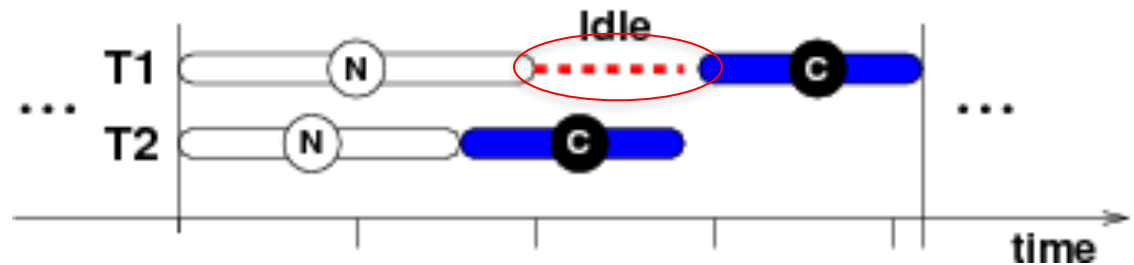
```
  Compute N
```

```
  P(A)
```

```
  Update shared data
```

```
  V(A) C
```

```
}
```



# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$1 - f$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$1 - f +$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$1 - f + \frac{f}{N}$$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$



# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - $f$ : Parallelizable fraction of a program
  - $N$ : Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable ( $f = 1$ ): Speedup =  $N$

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable (f = 1): Speedup = N
- Completely sequential (f = 0): Speedup = 1

# Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable (f = 1): Speedup = N
- Completely sequential (f = 0): Speedup = 1
- Mostly parallelizable (f = 0.9, **N = 1000**): **Speedup = 9.9**

# Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- **Hardware support of threads**
  - Single core
  - Multi-core
  - Cache coherence

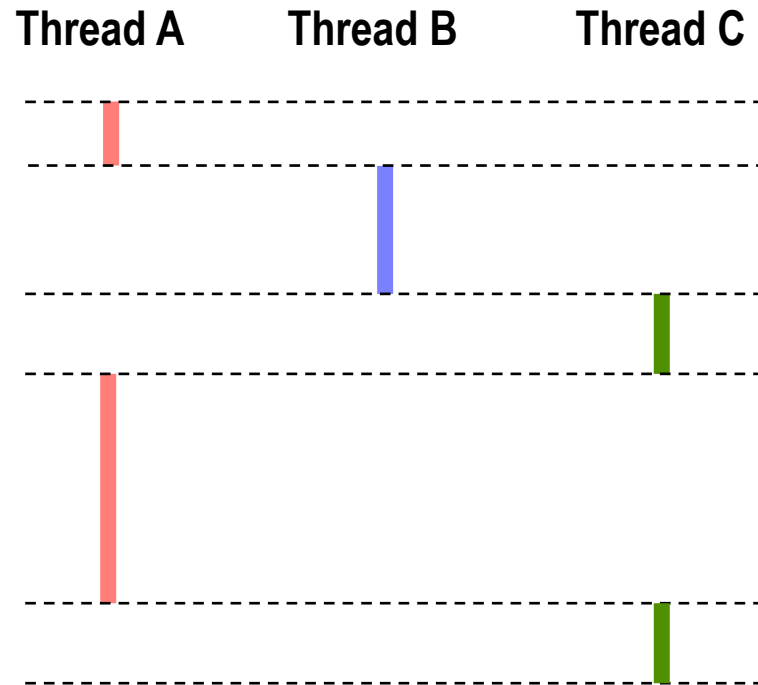
# Can A Single Core Support Multi-threading?

- Need to multiplex between different threads (time slicing)

## Sequential

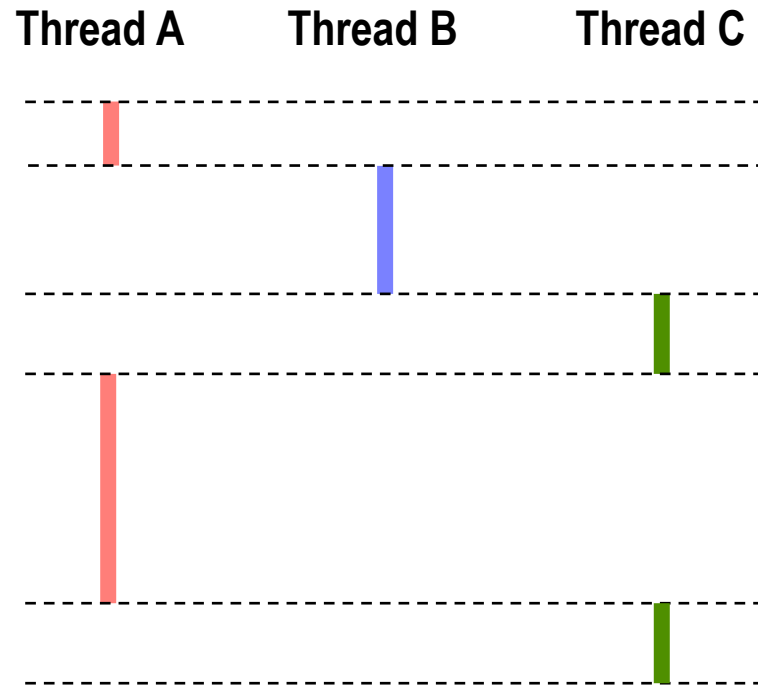


## Multi-threaded



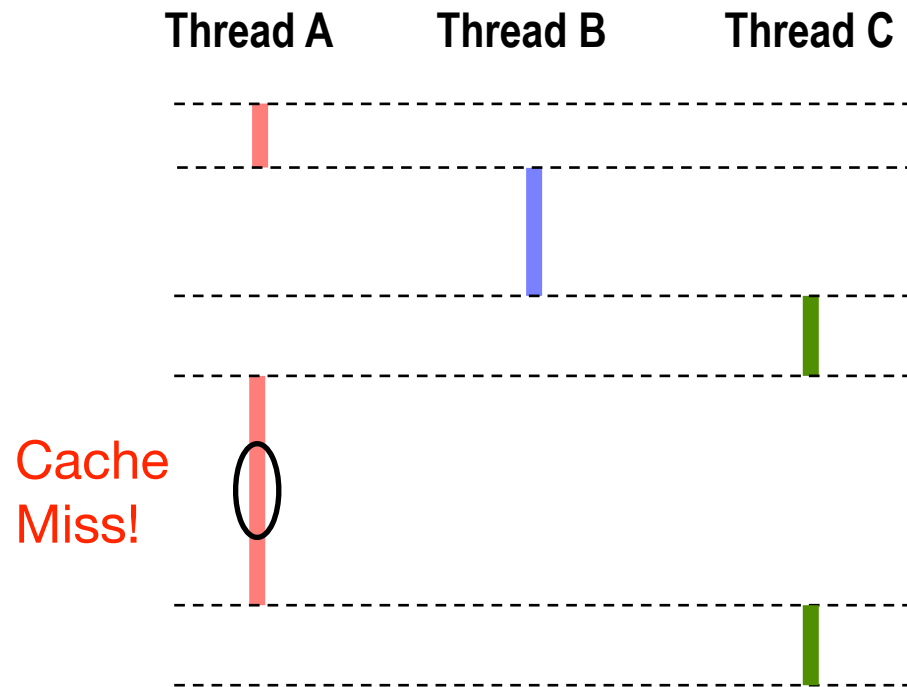
# Any benefits?

- Can single-core multi-threading provide any performance gains?



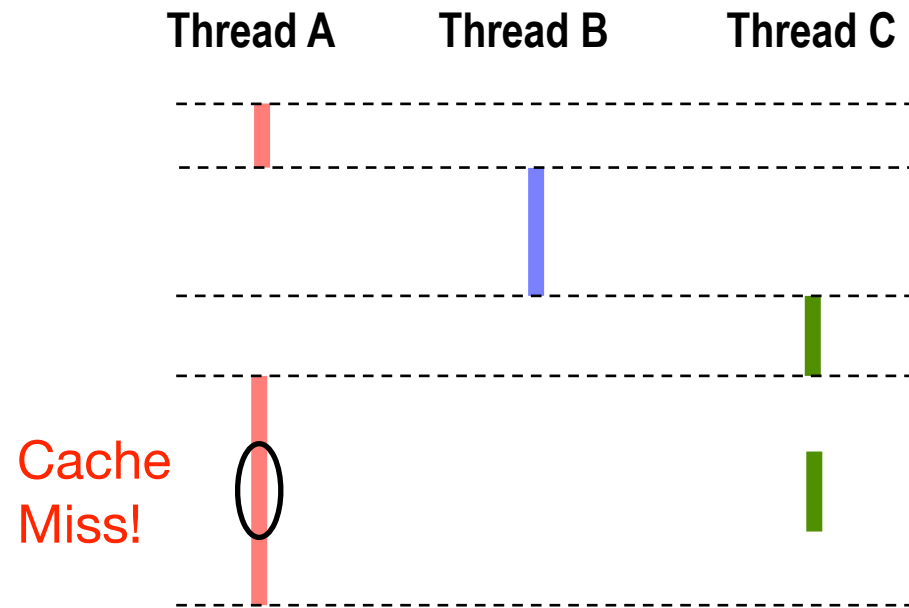
# Any benefits?

- Can single-core multi-threading provide any performance gains?



# Any benefits?

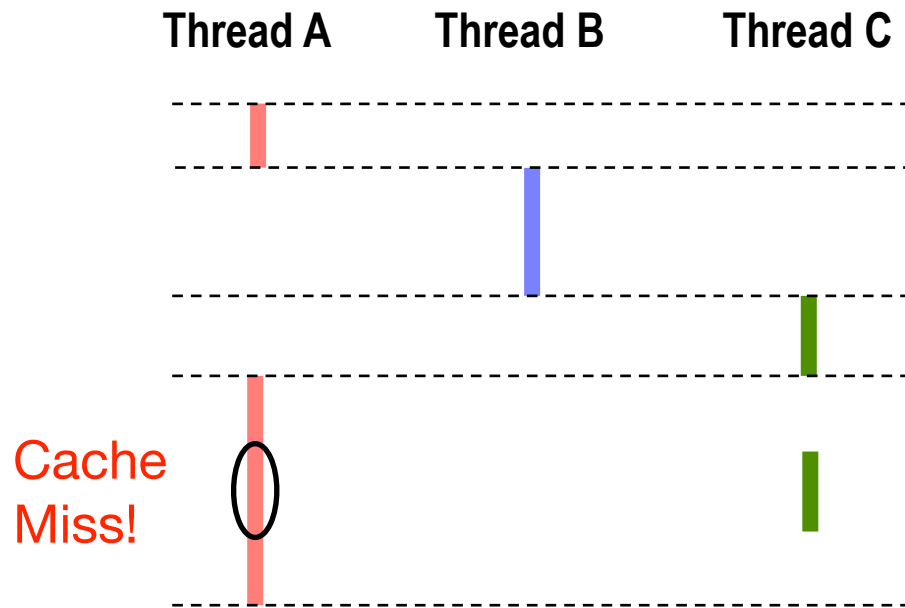
- Can single-core multi-threading provide any performance gains?





# Any benefits?

- Can single-core multi-threading provide any performance gains?
- If Thread A has a cache miss and the pipeline gets stalled, switch to Thread C. Improves the overall performance.



# When to Switch?

- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)

# When to Switch?

- **Coarse grained**
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)
- **Fine grained**
  - Cycle by cycle
  - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
  - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. The HEP machine. A seminal paper that shows that using multi-threading can avoid branch prediction.

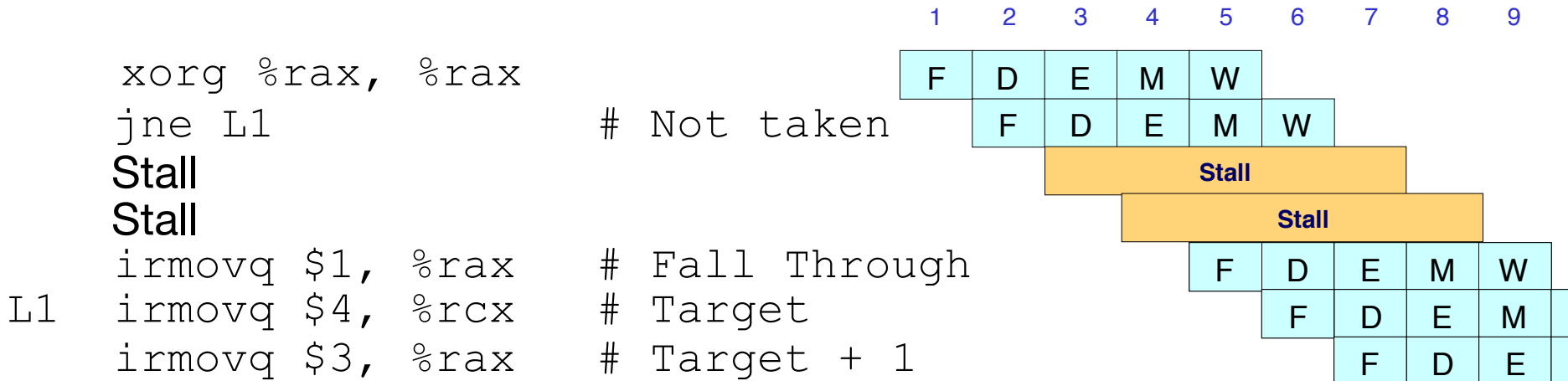
# When to Switch?

- **Coarse grained**
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)
- **Fine grained**
  - Cycle by cycle
  - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
  - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. The HEP machine. A seminal paper that shows that using multi-threading can avoid branch prediction.
- **Either way, need to save/restore thread context upon switching.**

# Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

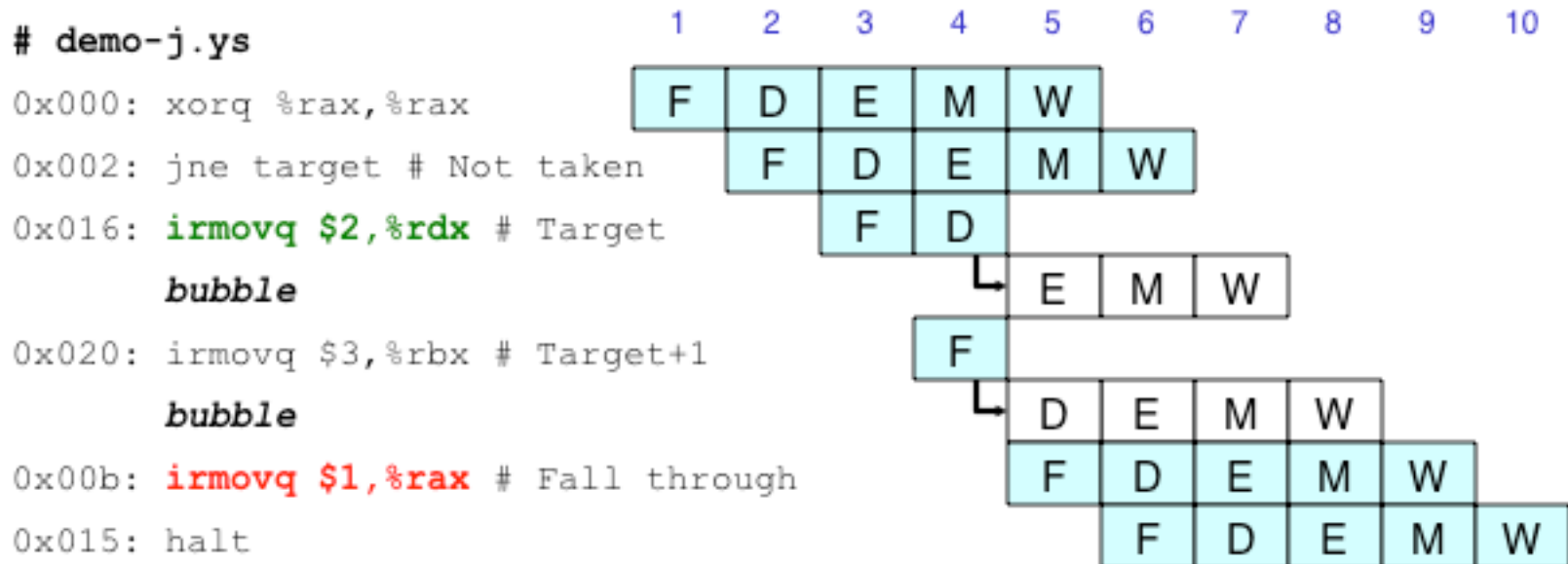
## The stalling approach



# Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

## The branch prediction approach

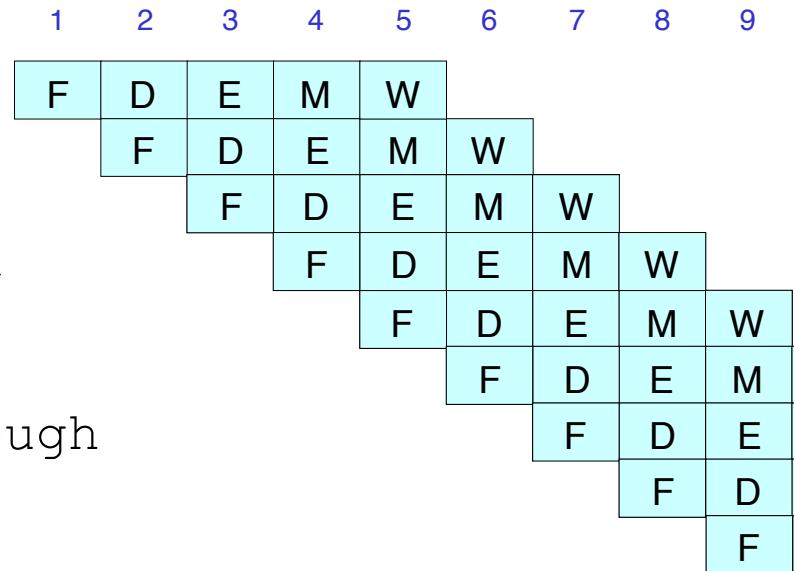


# Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

## The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1          # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
... ..
```

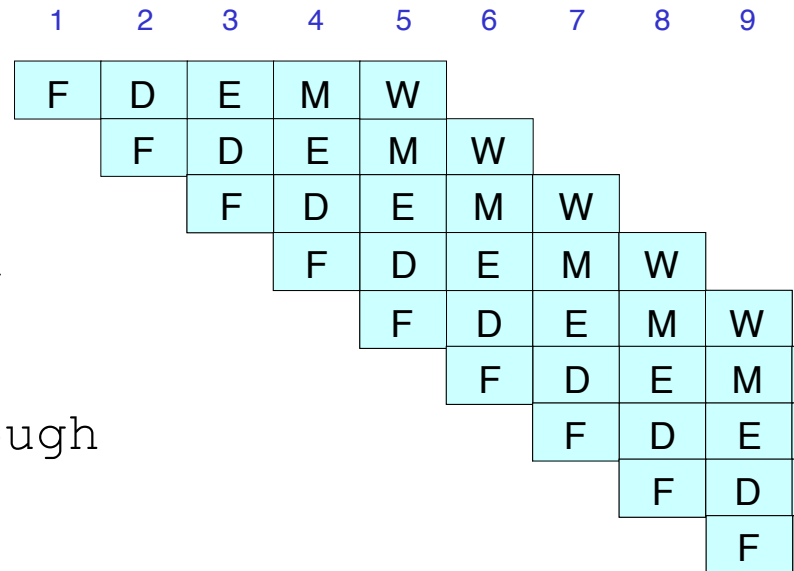


# Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!
  - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).

## The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1          # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
... ..
```



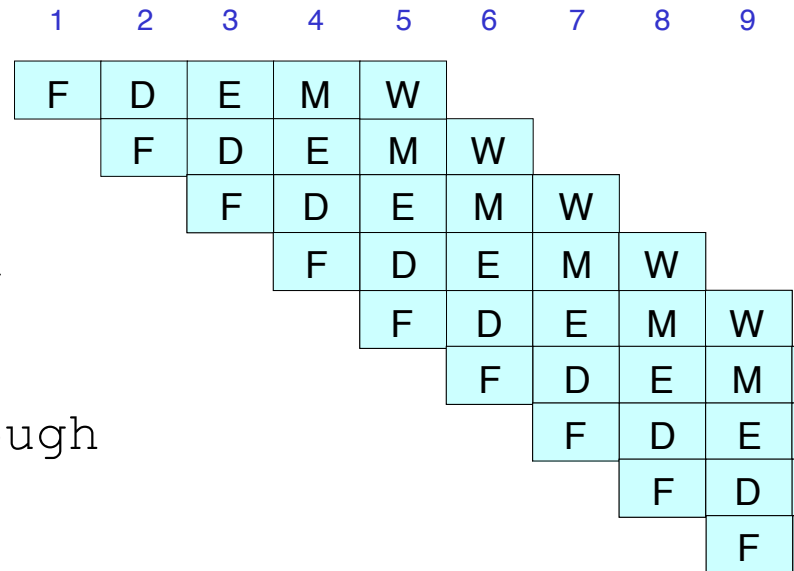


# Fine-Grained Switching

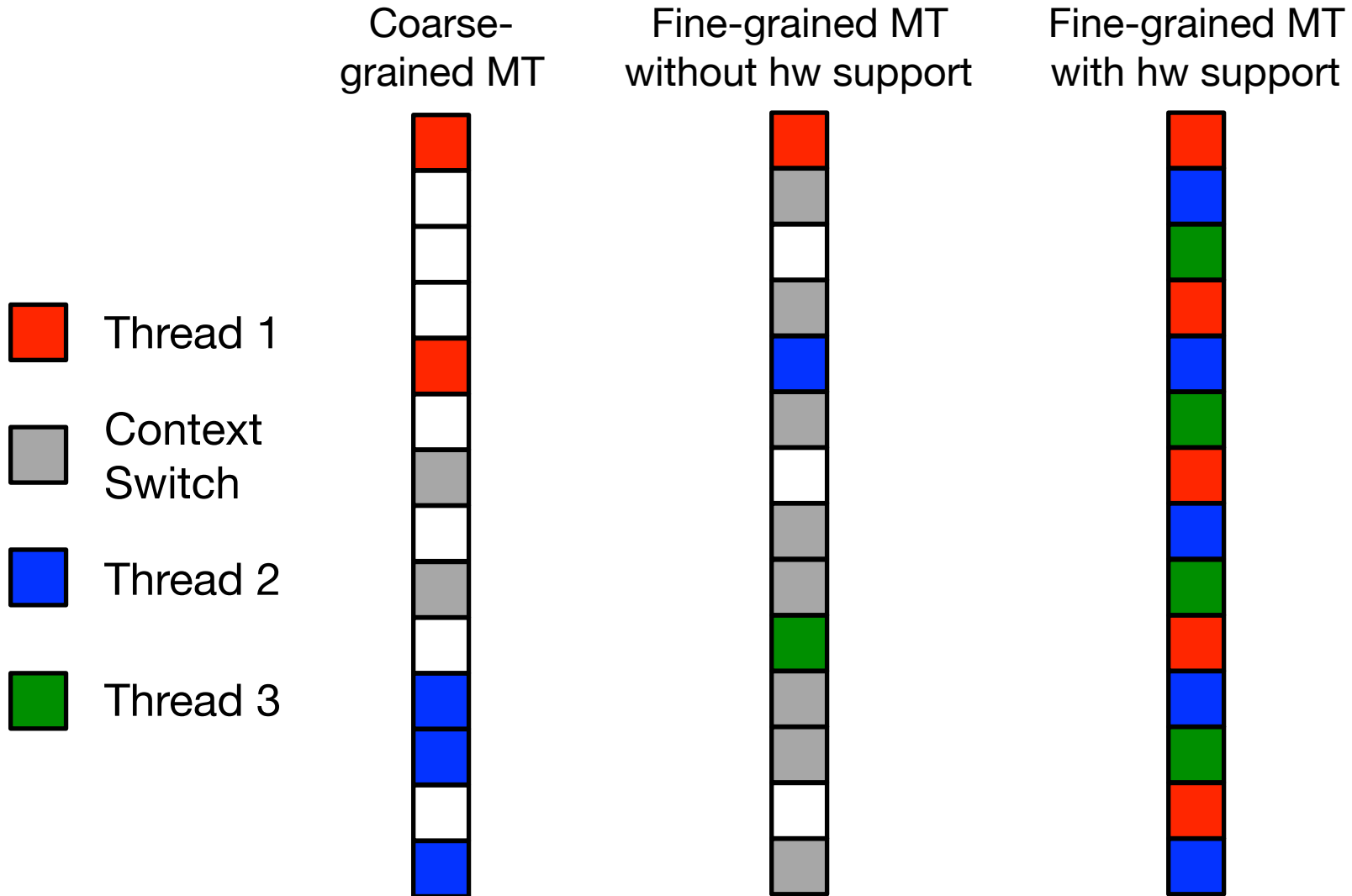
- One big bonus of fine-grained switching: no need for branch predictor!!
  - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).
  - GPUs do this (among other things). More later.

## The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1 # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
... ..
```

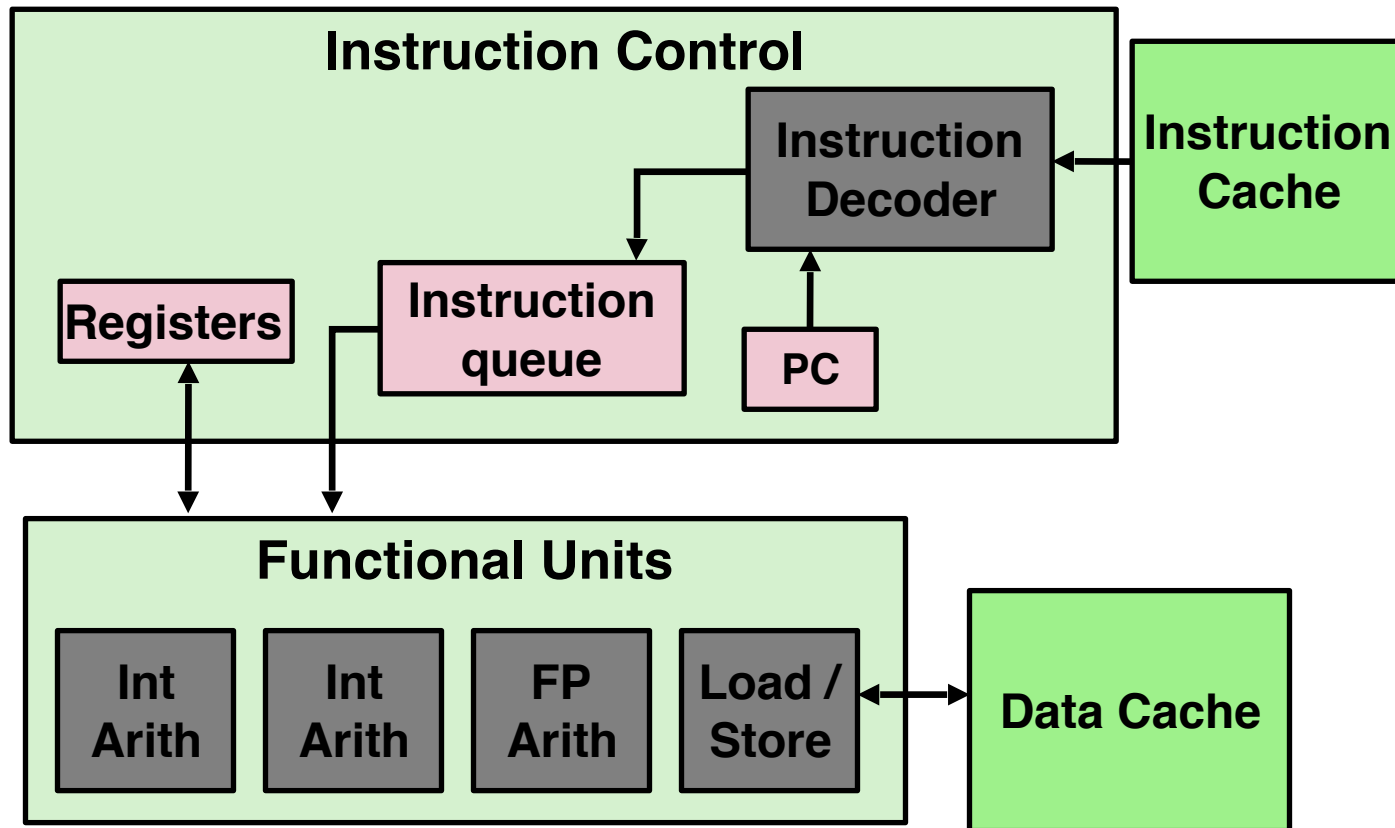


# Multi-threading Illustration (so far...)

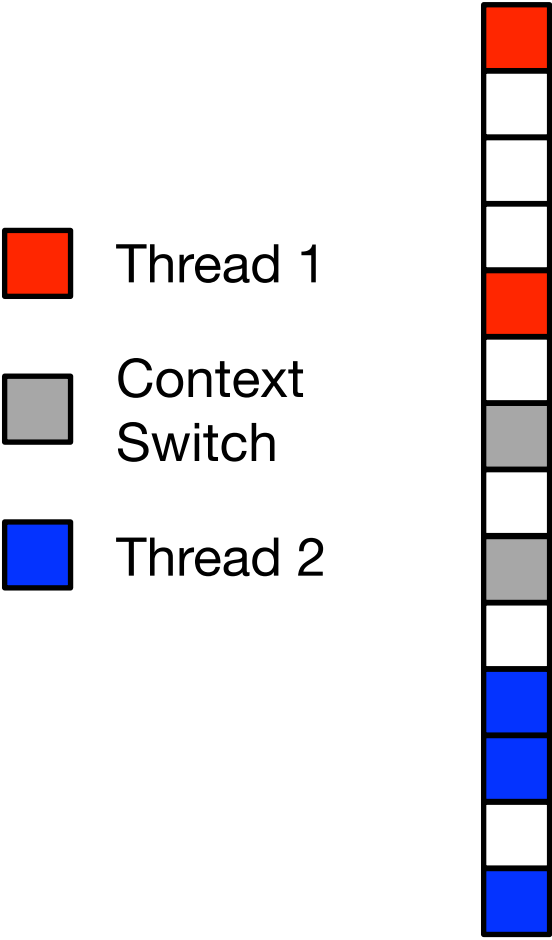


# Modern Single-Core: Superscalar

- Typically has multiple function units to allow for decoding and issuing multiple instructions at the same time
- Called “Superscalar”

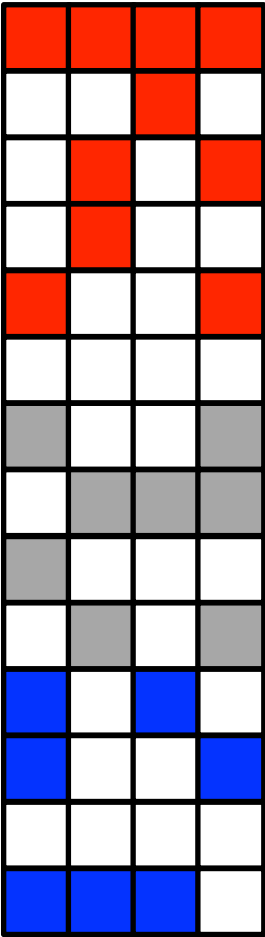


# From Scalar to Multi-Scalar Multi-threading



# From Scalar to Multi-Scalar Multi-threading

Functional Units



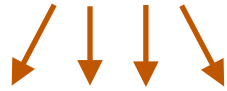
Red Thread 1

Gray Context Switch

Blue Thread 2




# From Scalar to Multi-Scalar Multi-threading

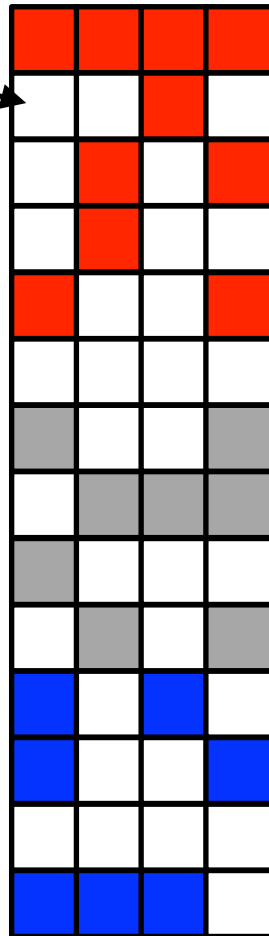
Functional Units



Why these empty slots?

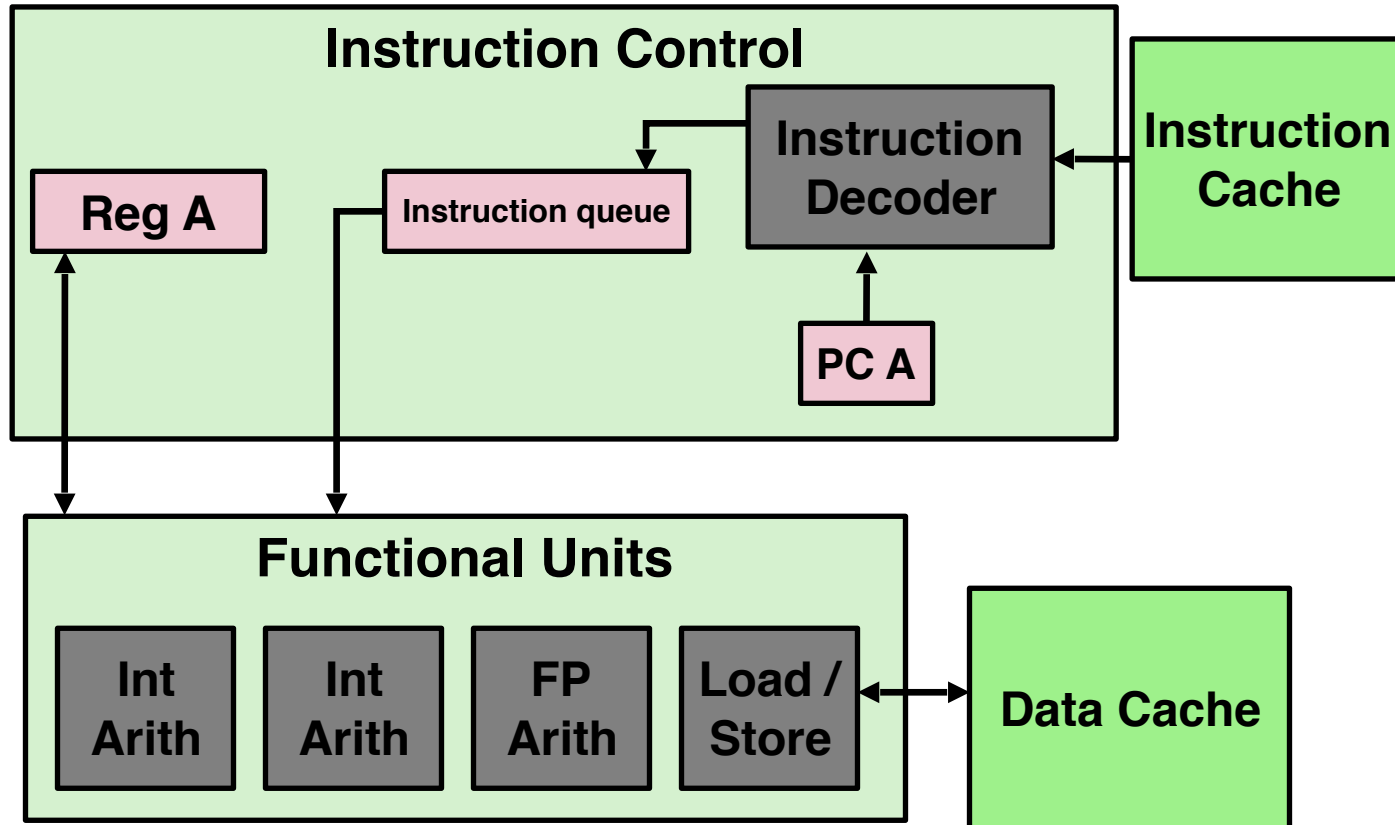


-  Thread 1
-  Context Switch
-  Thread 2



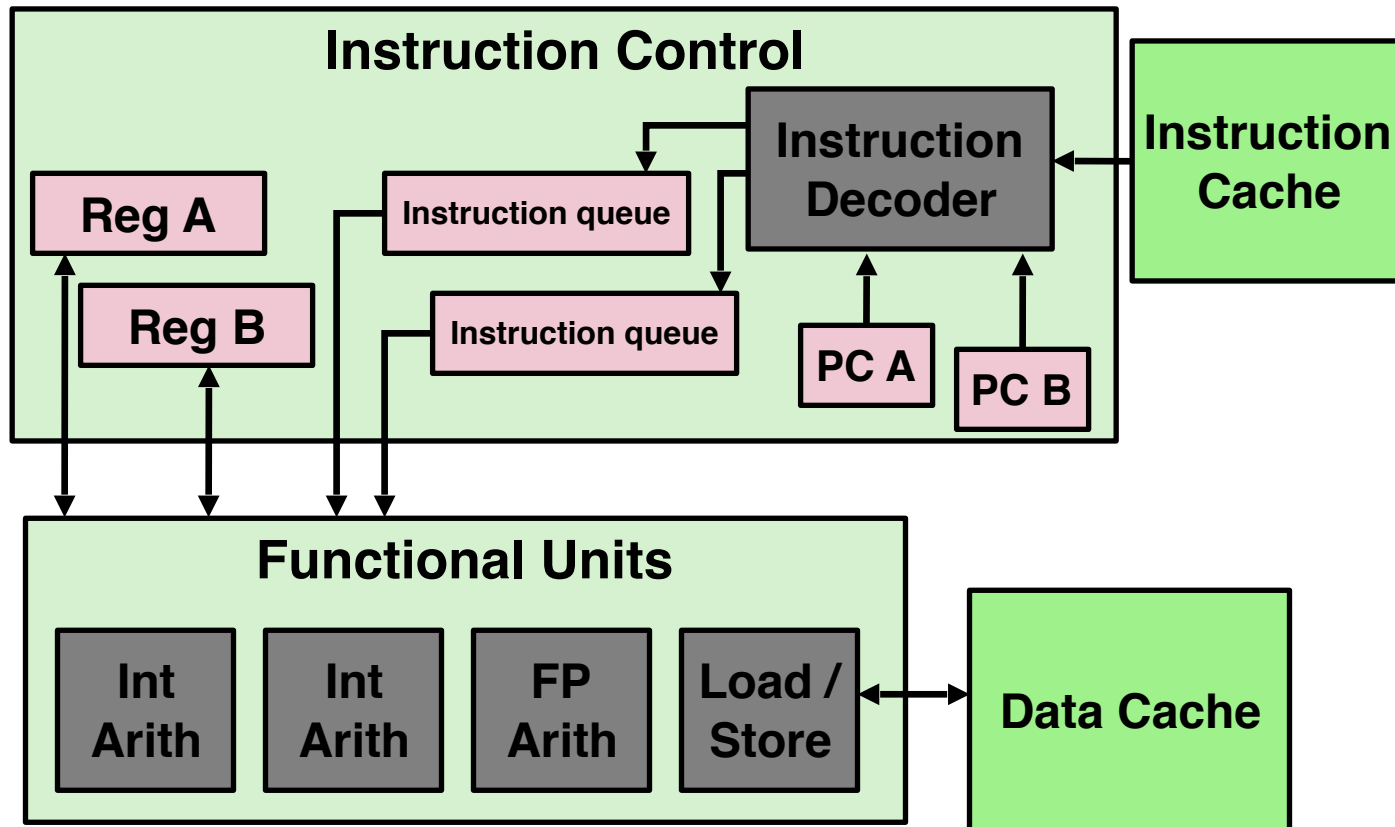
# Simultaneous Multi-Threading (SMT)

- Intel call it hyper-threading.
- Replicate enough hardware structures to process K instruction streams, i.e., threads. K copies of all registers. Share functional units.
- SMT = Superscalar + Multi-threading



# Simultaneous Multi-Threading (SMT)

- Intel call it hyper-threading.
- Replicate enough hardware structures to process K instruction streams, i.e., threads. K copies of all registers. Share functional units.
- SMT = Superscalar + Multi-threading








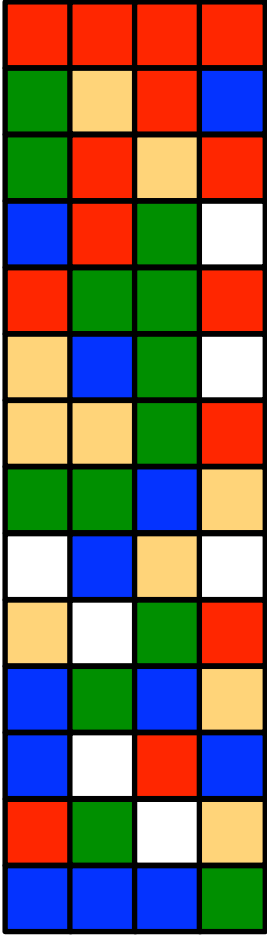
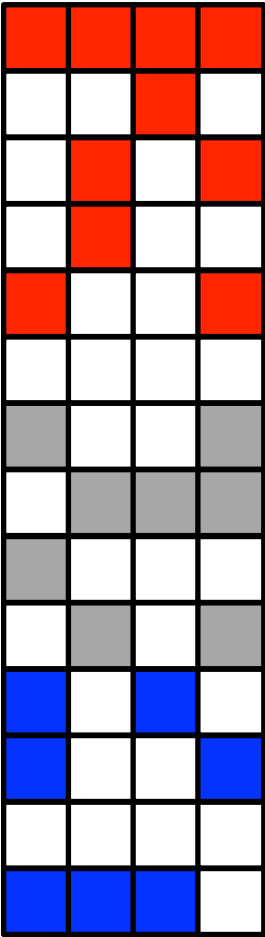


# Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT






-  Thread 1
-  Context Switch
-  Thread 2
-  Thread 3
-  Thread 4

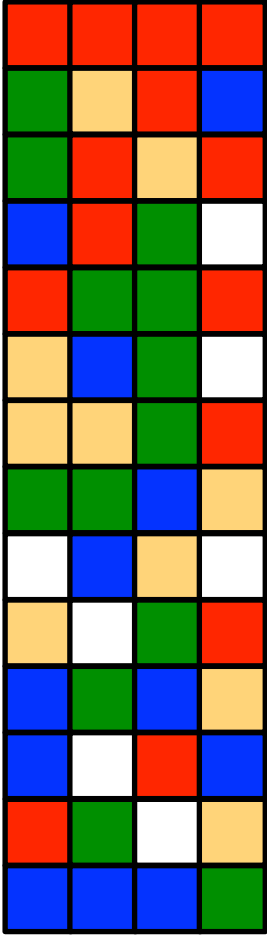
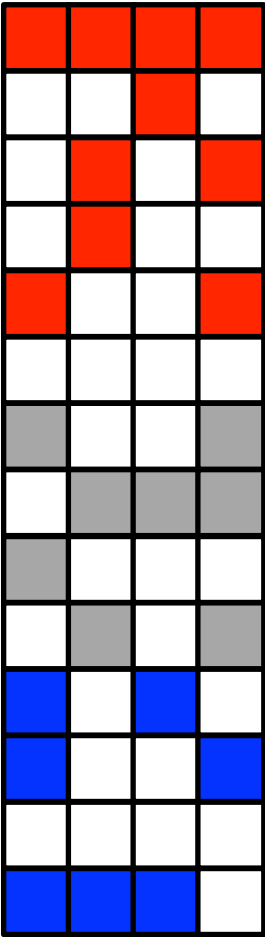


# Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT

-  Thread 1
-  Context Switch
-  Thread 2
-  Thread 3
-  Thread 4








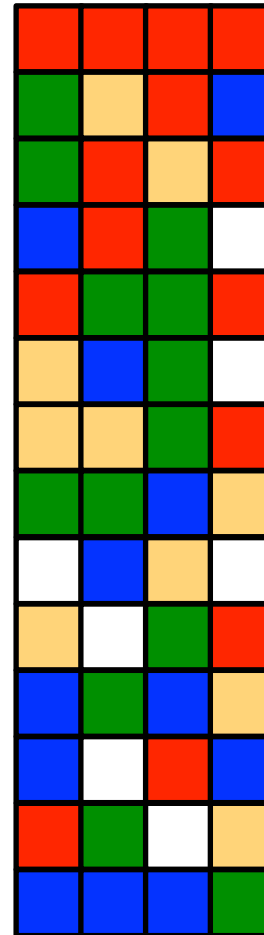
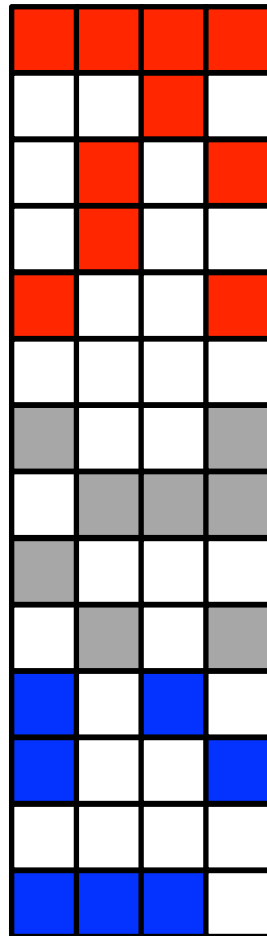
Can now make use of idle issue slots in conventional MT cores.

# Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT

-  Thread 1
-  Context Switch
-  Thread 2
-  Thread 3
-  Thread 4








Can now make use of idle issue slots in conventional MT cores.

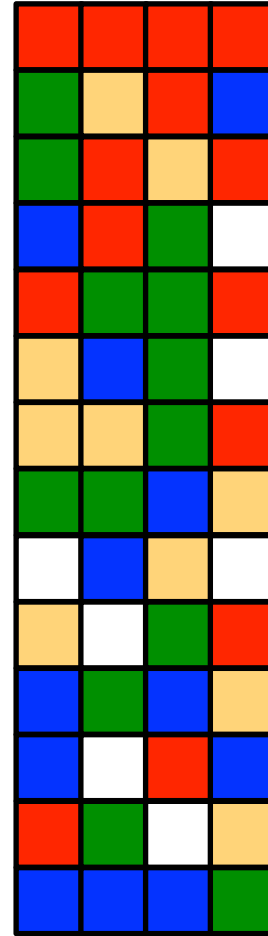
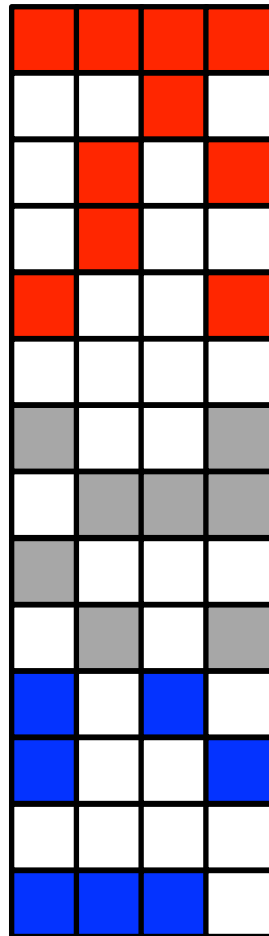
Multiple threads actually execute in parallel (even with one single core)

# Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT

-  Thread 1
-  Context Switch
-  Thread 2
-  Thread 3
-  Thread 4



Can now make use of idle issue slots in conventional MT cores.

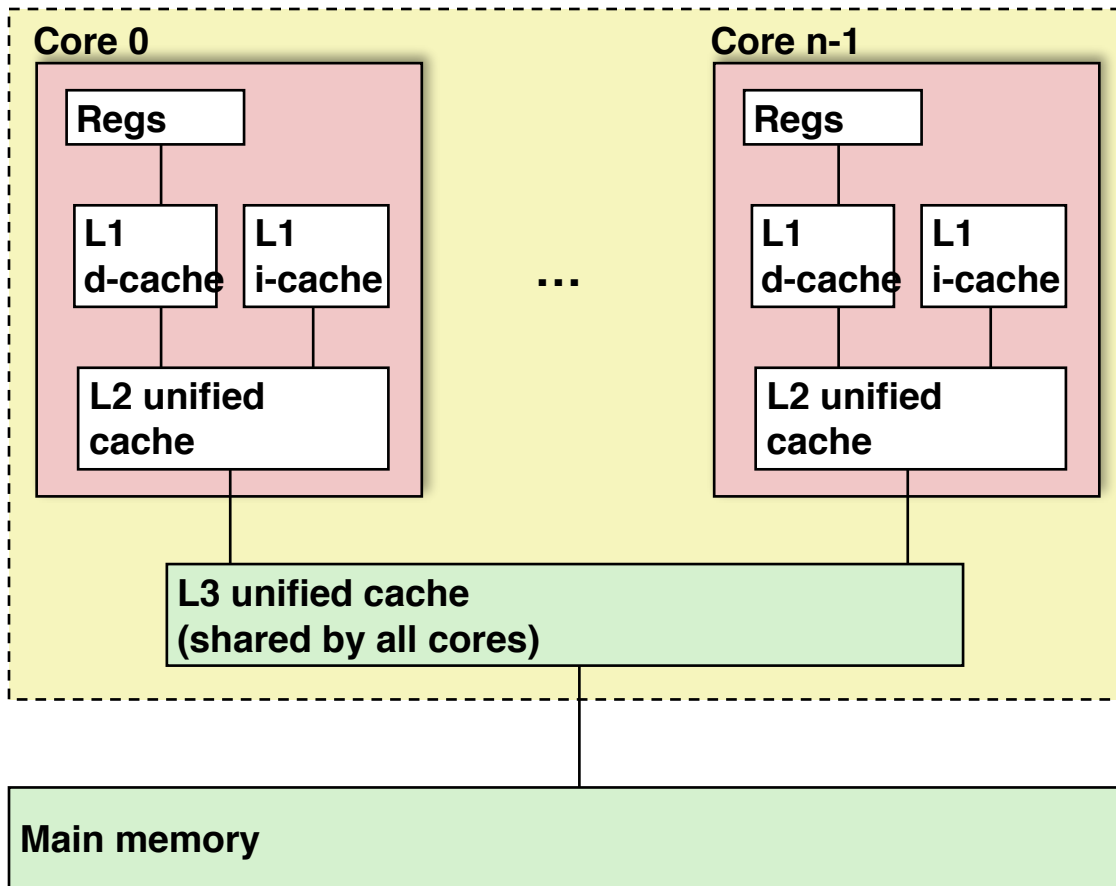
Multiple threads actually execute in parallel (even with one single core)

No/little context switch overhead

# Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- **Hardware support of threads**
  - Single core
  - **Multi-core**
  - Cache coherence

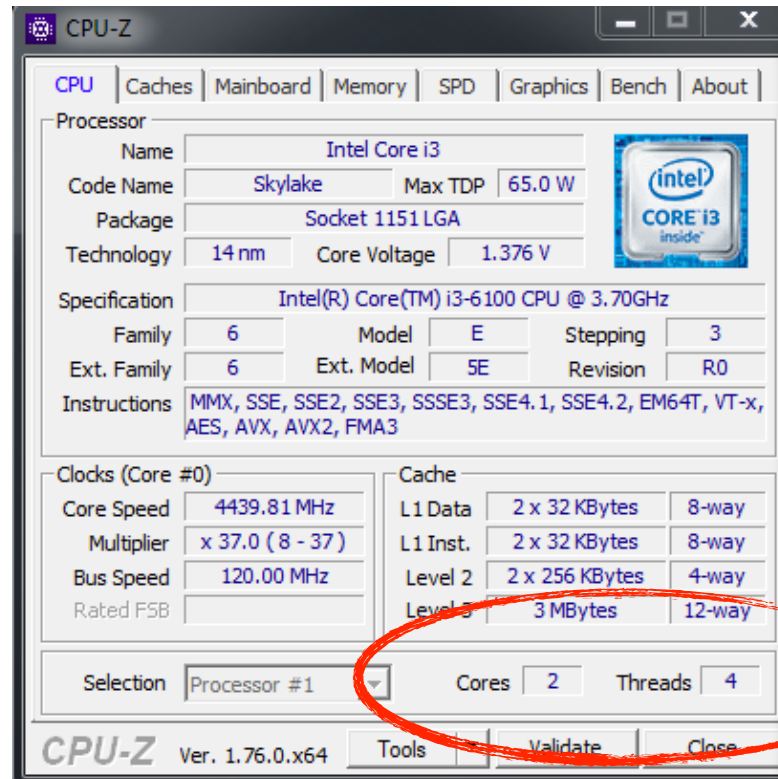
# Multi-Threading on a Multi-core Processor



- Each core can run multiple threads, mostly through coarse-grained switching.
- Fine-grained switching on conventional multi-core CPU is too costly.

# Combine Multi-core with SMT

- Common for laptop/desktop/server machine. E.g., 2 physical cores, each core has 2 hyper-threads => 4 virtual cores.
- Not for mobile processors (Hyper-threading costly to implement)

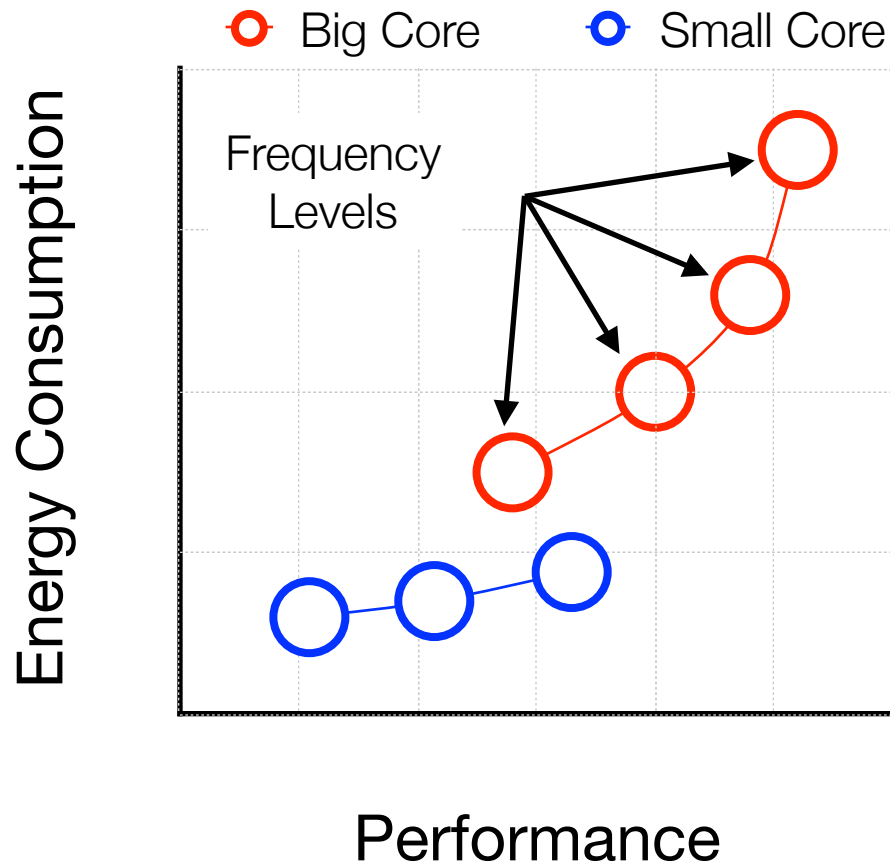


The screenshot shows the CPU-Z application window. The 'CPU' tab is selected. The processor is identified as an Intel Core i3 (Skylake) with 2 physical cores and 4 threads. The 'Cores' and 'Threads' fields at the bottom are circled in red.

Processor			
Name	Intel Core i3		
Code Name	Skylake	Max TDP	65.0 W
Package	Socket 1151 LGA		
Technology	14 nm	Core Voltage	1.376 V
Specification			
Intel(R) Core(TM) i3-6100 CPU @ 3.70GHz			
Family	6	Model	E
Ext. Family	6	Ext. Model	5E
Stepping			3
Revision	R0		
Instructions	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX, AVX2, FMA3		
Clocks (Core #0)			
Core Speed	4439.81 MHz		
Multiplier	x 37.0 ( 8 - 37 )		
Bus Speed	120.00 MHz		
Rated FSB			
Cache			
L1 Data	2 x 32 KBytes	8-way	
L1 Inst.	2 x 32 KBytes	8-way	
Level 2	2 x 256 KBytes	4-way	
Level 3	3 MBytes	12-way	
Selection	Processor #1	Cores	2
		Threads	4

# Asymmetric Multiprocessor (AMP)

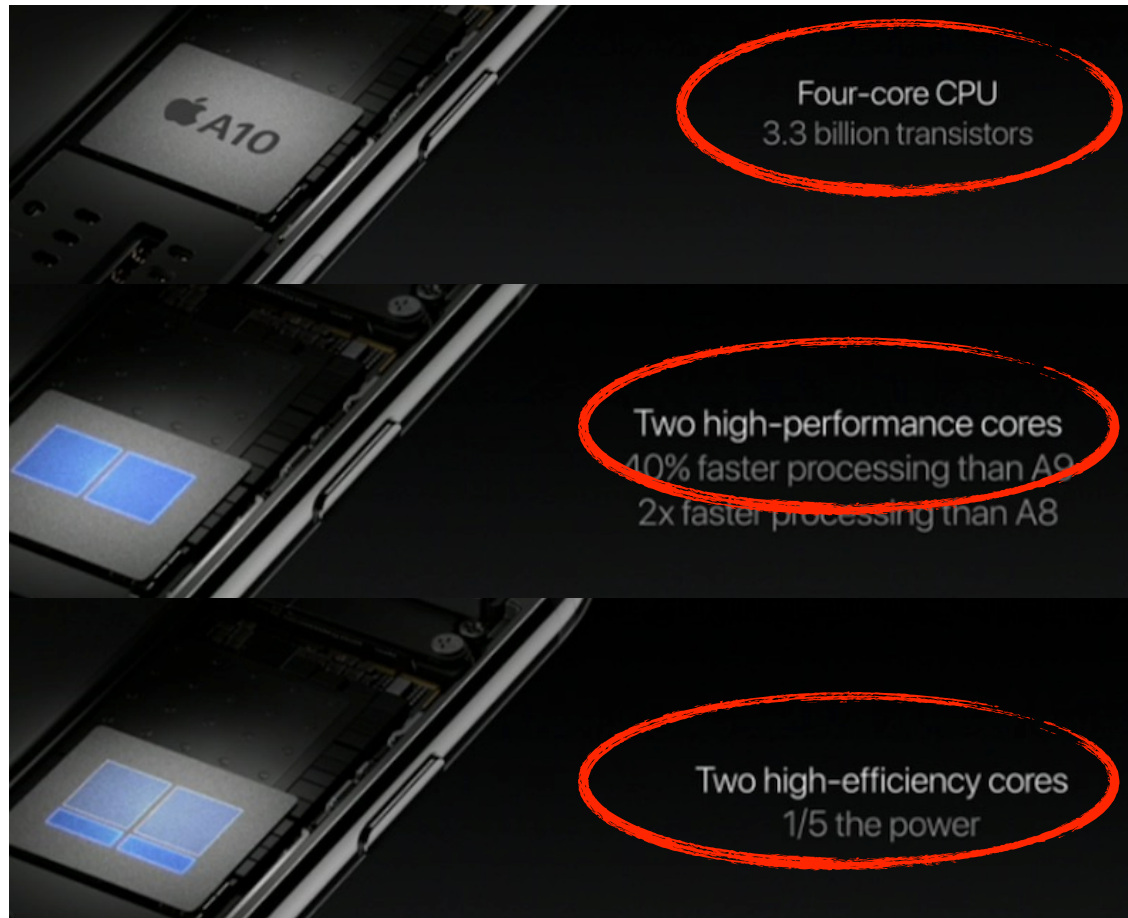
- Offer a large performance-energy trade-off space





# Asymmetric Chip-Multiprocessor (ACMP)

- Already used in commodity devices (e.g., Samsung Galaxy S6, iPhone 7)



# Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- **Hardware support of threads**
  - Single core
  - Multi-core
  - Cache coherence

# The Issue

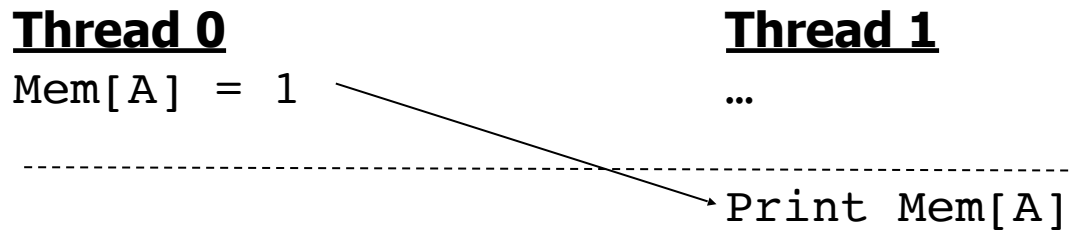
- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.

# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.

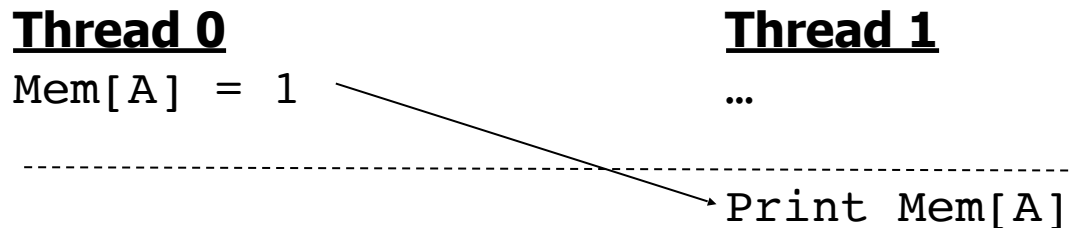
# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.



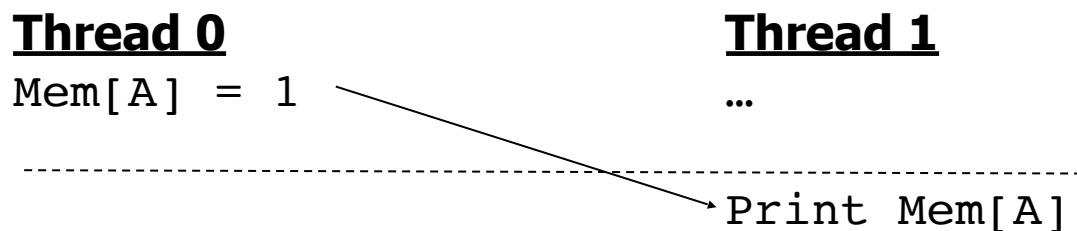
# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone



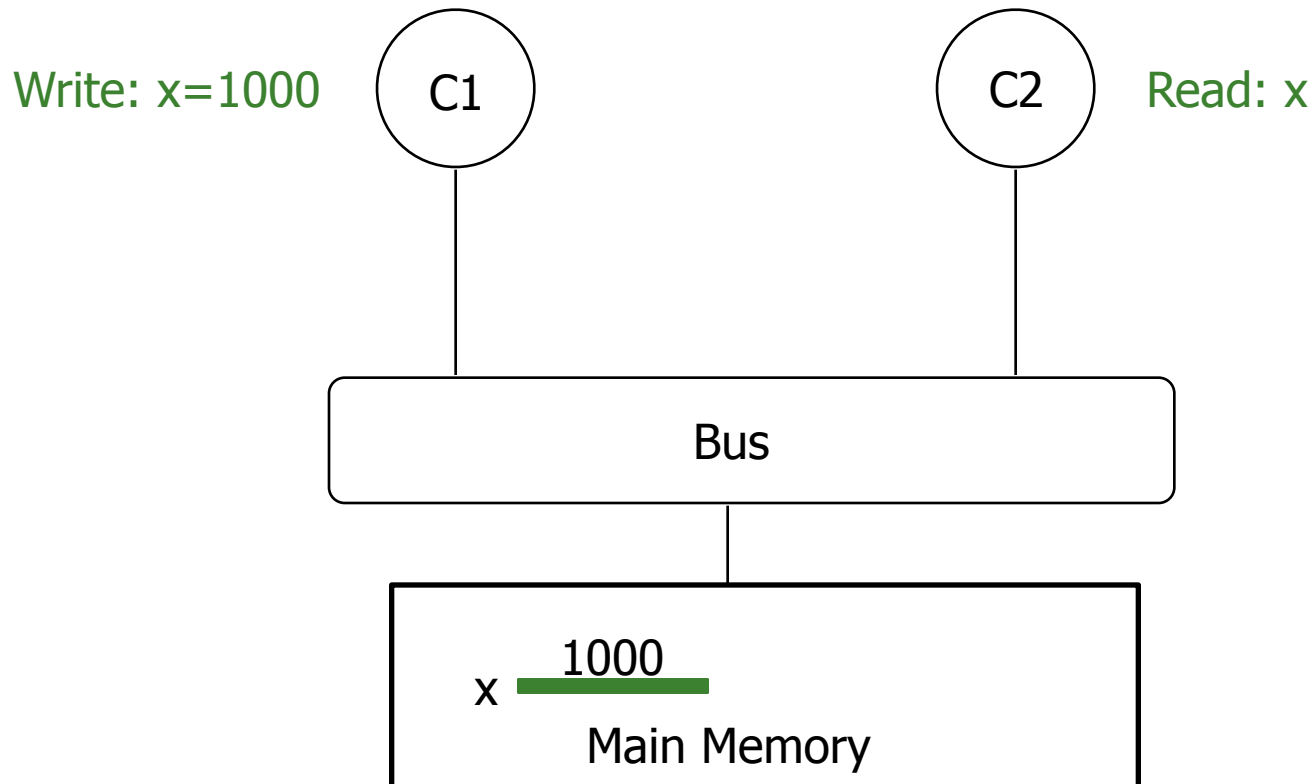
# The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone
- **Basic question:** If multiple cores access the same data, how do they ensure they all see a consistent state?



# The Issue

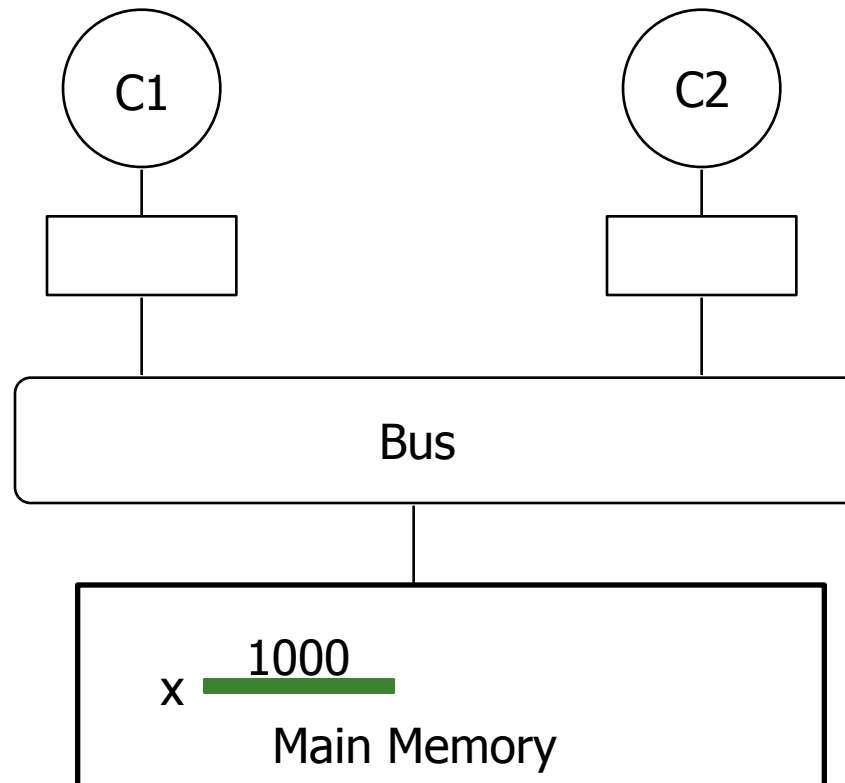
- Without cache, the issue is (theoretically) solvable by using mutex.
- ...because there is only one copy of  $x$  in the entire system. Accesses to  $x$  in memory are serialized by mutex.





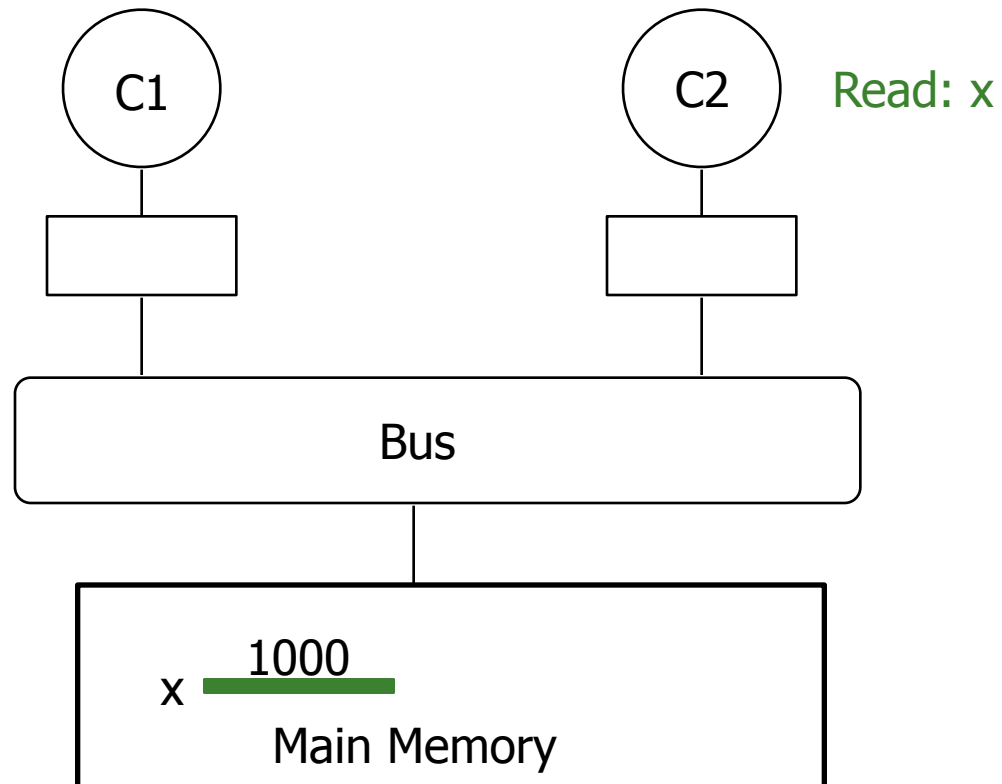
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



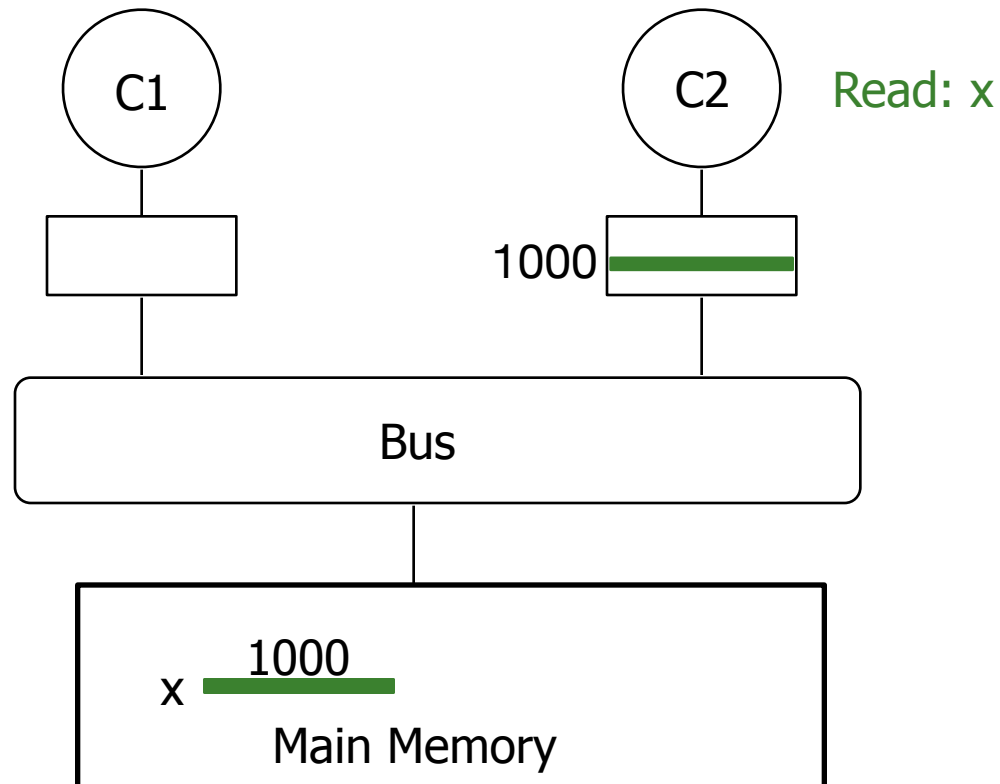
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



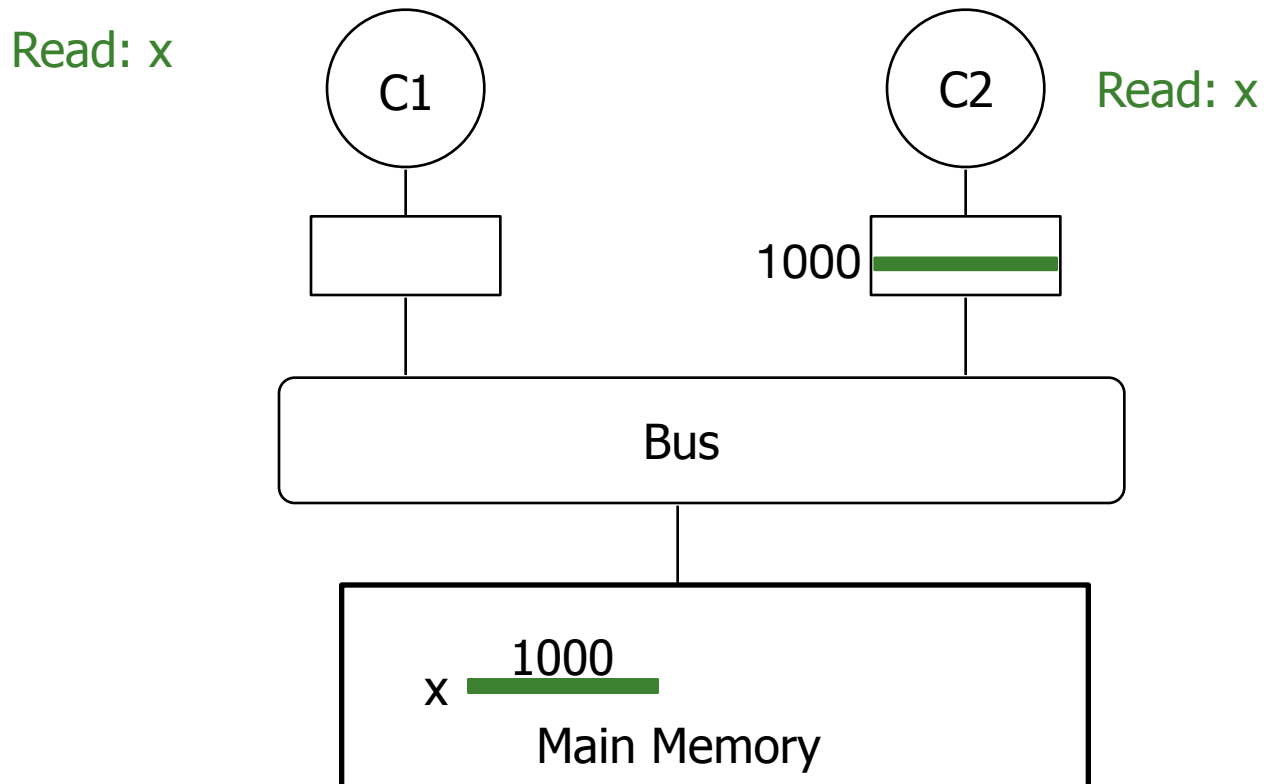
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



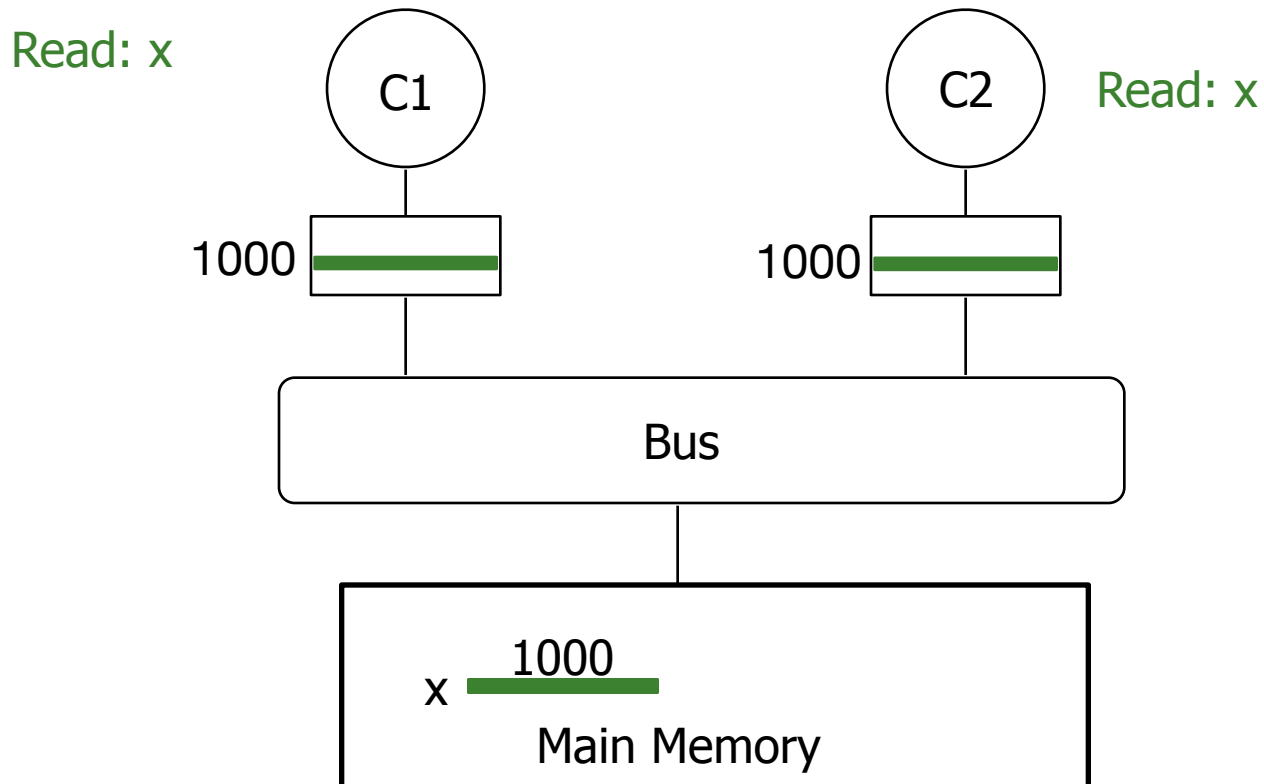
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



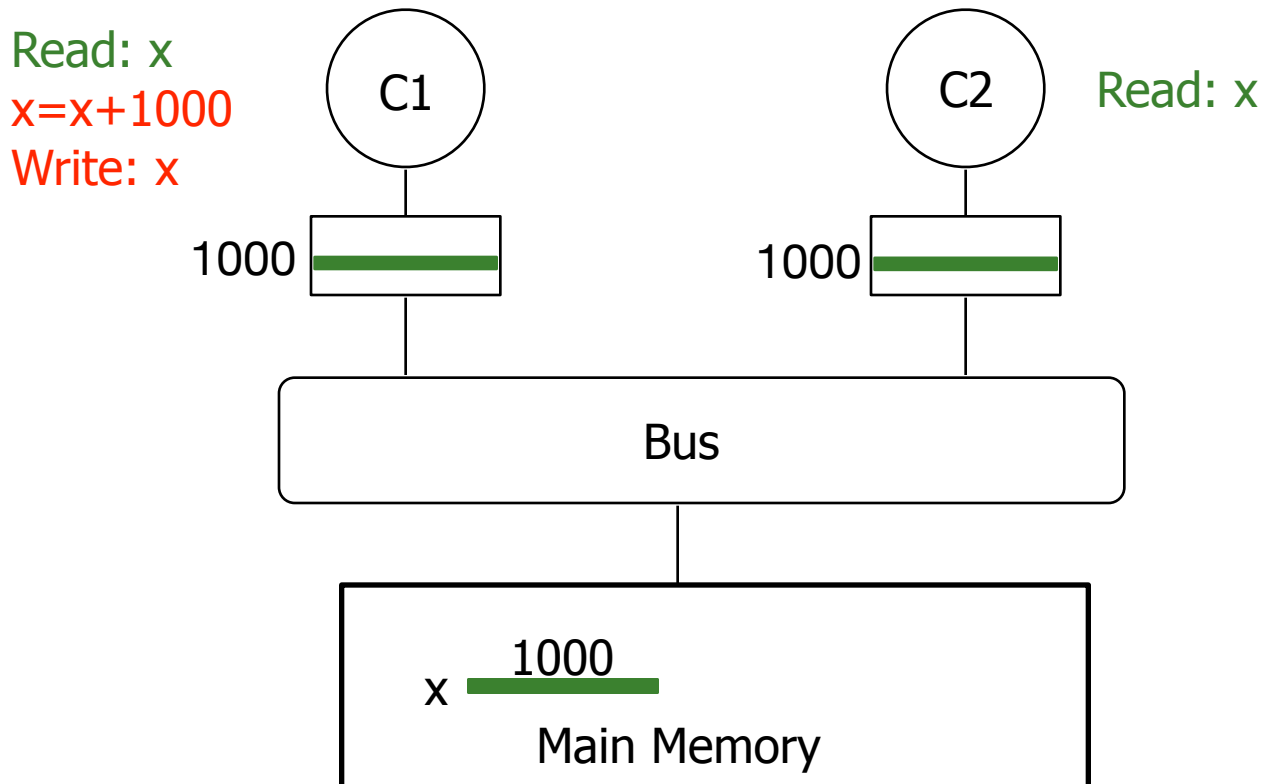
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



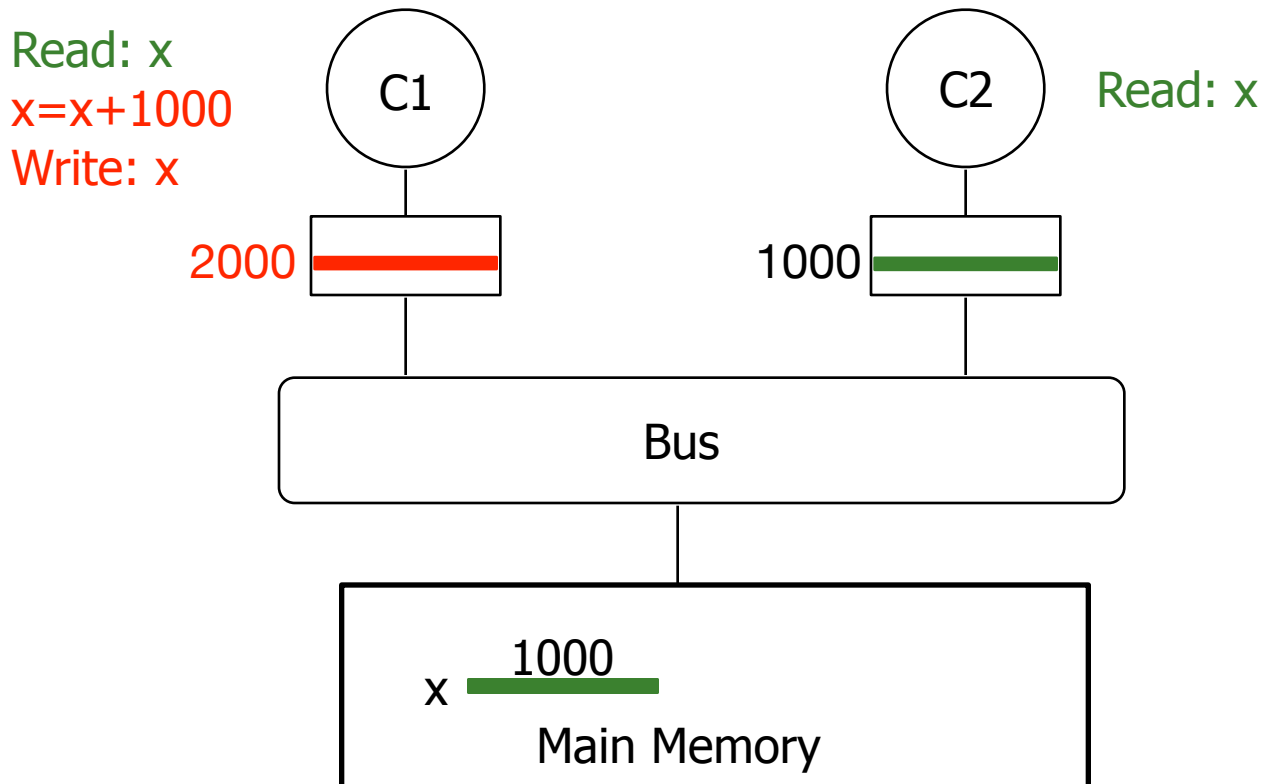
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



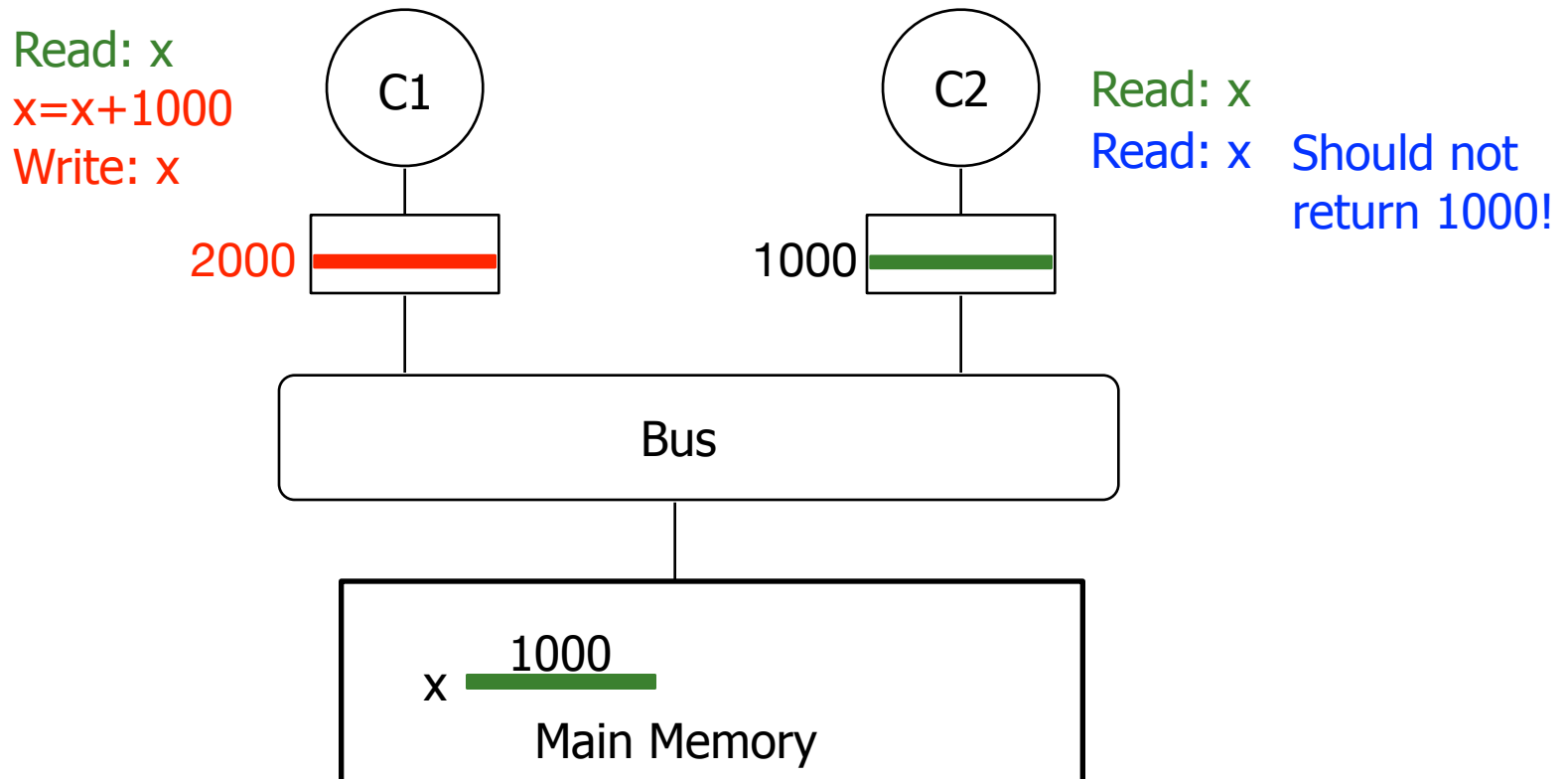
# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



# The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)





# Cache Coherence: The Idea

- **Issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.

# Cache Coherence: The Idea

- **Issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea:** ensure multiple copies have same value, i.e., *coherent*

# Cache Coherence: The Idea

- **Issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:

# Cache Coherence: The Idea

- **Issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
  - **Update:** push new value to all copies (in other caches)

# Cache Coherence: The Idea

- **Issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
  - **Update:** push new value to all copies (in other caches)
  - **Invalidate:** invalidate other copies (in other caches)

# Readings: Cache Coherence

- Most helpful

- Culler and Singh, Parallel Computer Architecture
  - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
- Patterson&Hennessy, Computer Organization and Design
  - Chapter 5.8 (pp 534 – 538 in 4<sup>th</sup> and 4<sup>th</sup> revised eds.)
- Papamarcos and Patel, “[A low-overhead coherence solution for multiprocessors with private cache memories,](#)” ISCA 1984.

- Also very useful

- Censier and Feautrier, “[A new solution to coherence problems in multicache systems,](#)” IEEE Trans. Computers, 1978.
- Goodman, “[Using cache memory to reduce processor-memory traffic,](#)” ISCA 1983.
- Laudon and Lenoski, “[The SGI Origin: a ccNUMA highly scalable server,](#)” ISCA 1997.
- Martin et al, “[Token coherence: decoupling performance and correctness,](#)” ISCA 2003.
- Baer and Wang, “[On the inclusion properties for multi-level cache hierarchies,](#)” ISCA 1988.

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?



# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.

# Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- **Key: ISA must provide cache flush/invalidate instructions**
  - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
  - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
  - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- **Classic example: TLB**
  - Hardware does not guarantee that TLBs of different core are coherent
  - ISA provides instructions for OS to flush PTEs
  - Called "TLB shutdown"