

CSC 252: Computer Organization

Spring 2018: Lecture 25

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Programming Assignment 5 grades are out**
- **Programming Assignment 6 is due soon**

Announcement

- Grades for Lab 4 and 5 are out. Talk to Yu and Amir.
- Lab 6 is out. Due 11:59pm, **Thursday, May 2.**
- Take a look at the cache and virtual memory problem sets.

21	22	23	24	25 Today	26	27
28	29	30	May 1	2 Due	3	4

Today

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

Today

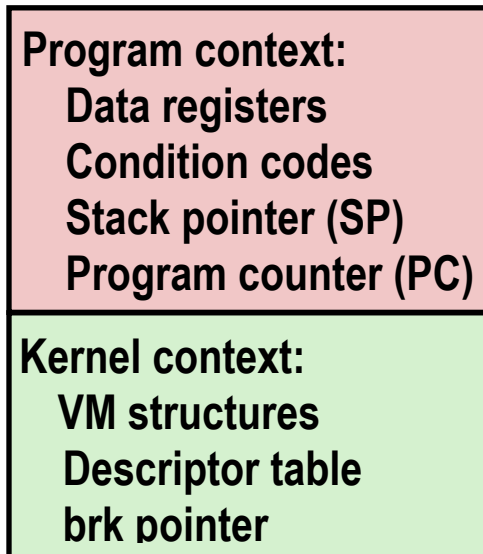
- From process to threads
 - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
 - Multi-core
 - Hyper-threading
 - Cache coherence

**We will be scratching the surface.
Take CSC 2/458 to learn more!!**

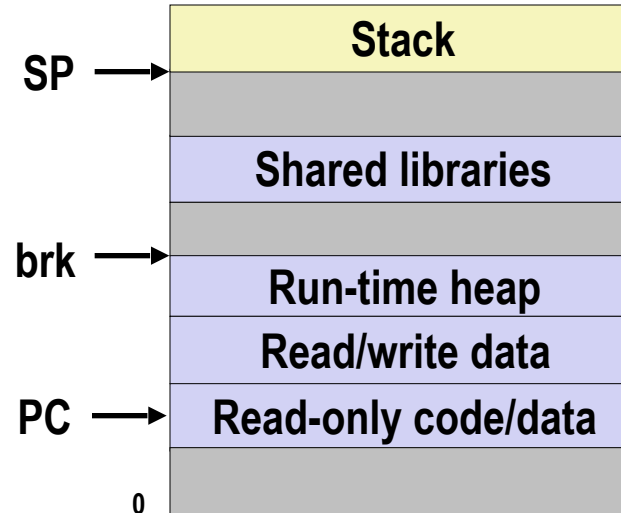
Programmers View of A Process

- Process = process context + code, data, and stack

Process context



Code, data, and stack



A Process With Multiple Threads

- Multiple threads can be associated with a process
 - Each thread has its own logical control flow
 - Each thread shares the same code, data, and kernel context
 - Each thread has its own stack for local variables
 - but not protected from other threads
 - Each thread has its own thread id (TID)

Thread 1 (main thread)

Thread 2 (peer thread)

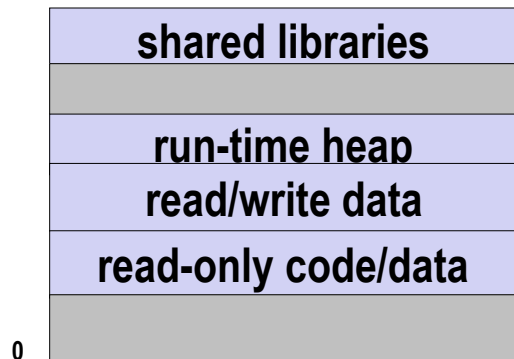
Shared code and data

stack 1

stack 2

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Thread 2 context:
Data registers
Condition codes
SP2
PC2

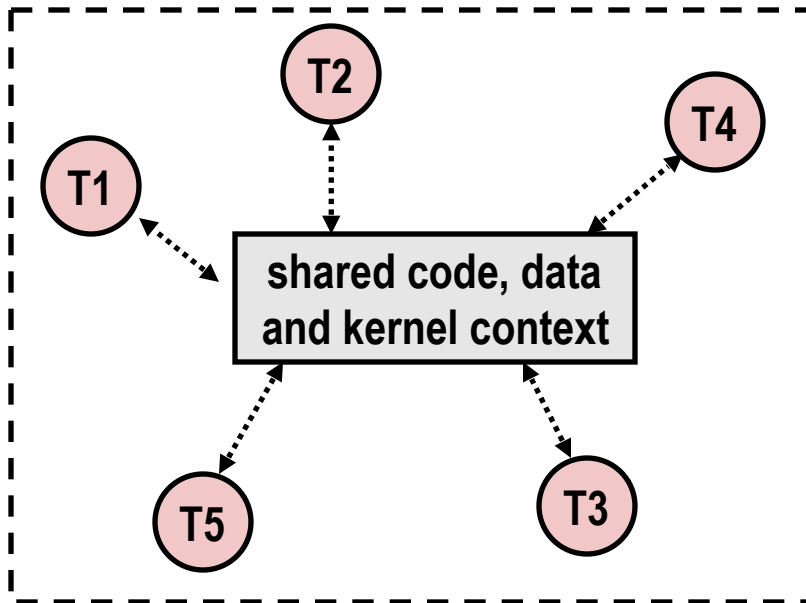


Kernel context:
VM structures
Descriptor table
brk pointer

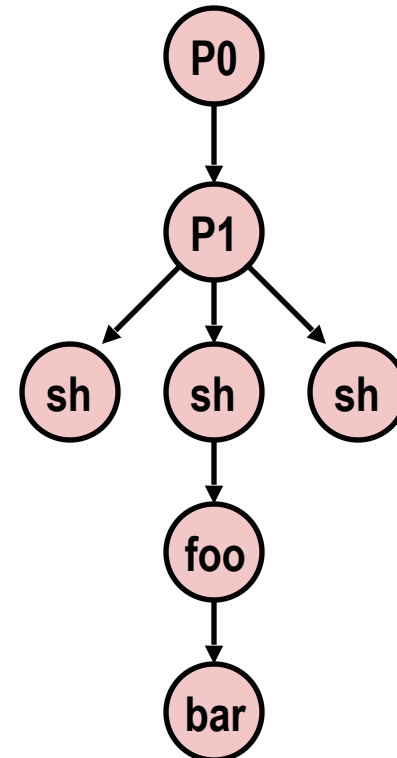
Logical View of Threads

- Threads associated with process form a pool of peers
 - Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



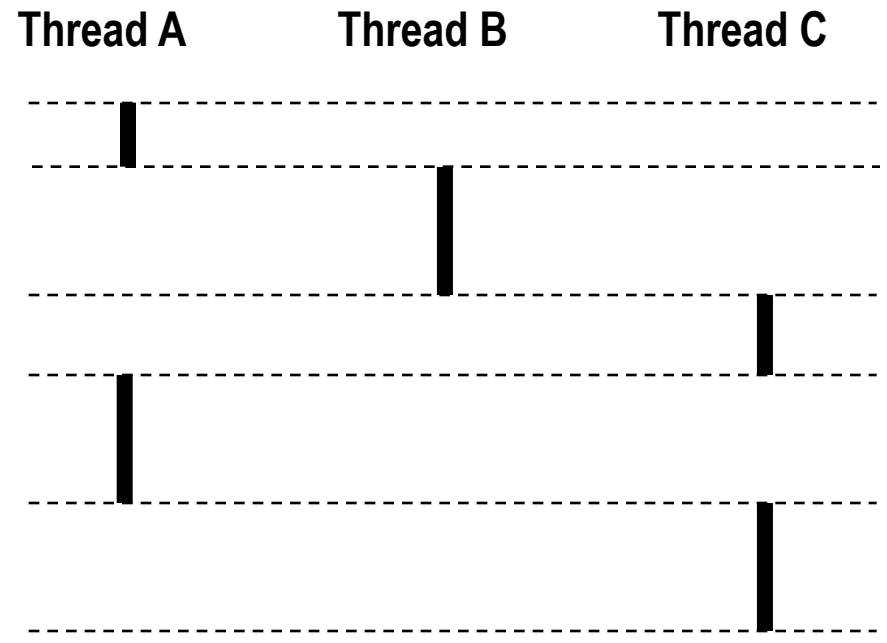
Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

- Examples:

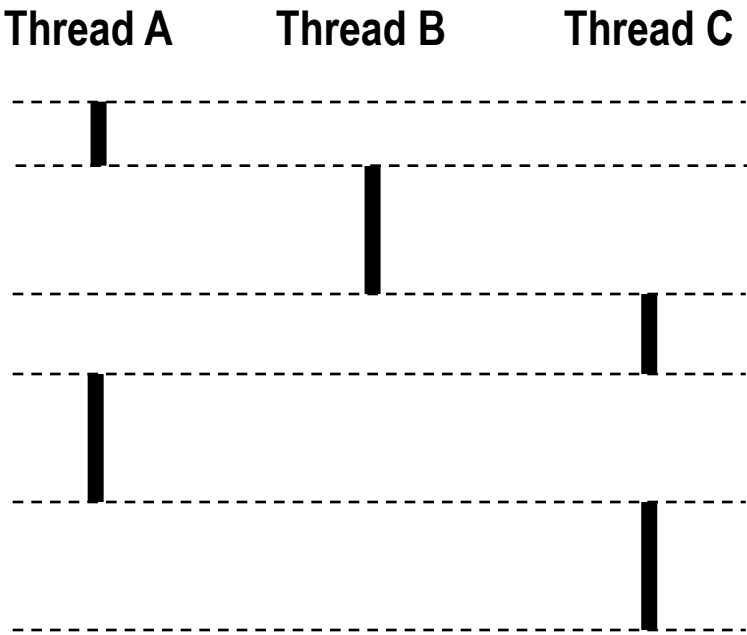
- Concurrent: A & B, A&C
- Sequential: B & C

Time

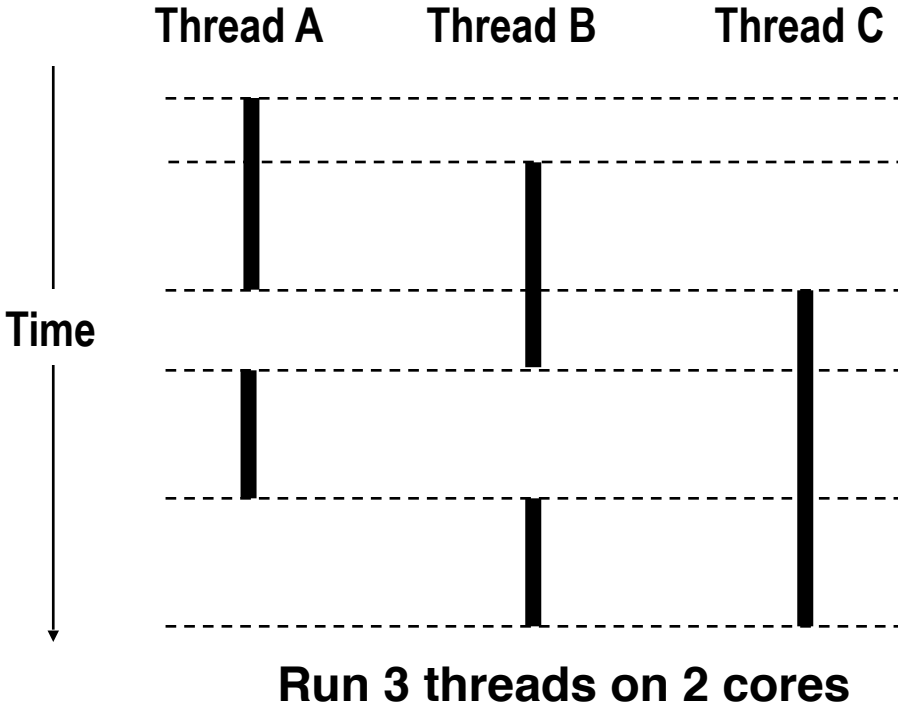


Concurrent Thread Execution

- Single Core Processor
 - Simulate parallelism by time slicing



- Multi Core Processor
 - Threads can have true parallelisms



Threads vs. Processes

- How threads and processes are similar
 - Each has its own logical control flow
 - Each can run concurrently with others (possibly on different cores)
 - Each is context switched, controlled by kernel

Threads vs. Processes

- How threads and processes are similar
 - Each has its own logical control flow
 - Each can run concurrently with others (possibly on different cores)
 - Each is context switched, controlled by kernel
- How threads and processes are different
 - Threads share all code and data (except local stacks)
 - Processes (typically) do not
 - Threads are less expensive than processes
 - Process control (creating and reaping) twice as expensive as thread control
 - Typical Linux numbers:
 - ~20K cycles to create and reap a process
 - ~10K cycles (or less) to create and reap a thread

Posix Threads (Pthreads) Interface

- *Pthreads*: Standard interface for ~60 functions that manipulate threads from C programs
 - Creating and reaping threads
 - `pthread_create()`
 - `pthread_join()`
 - Determining your thread ID
 - `pthread_self()`
 - Terminating threads
 - `pthread_cancel()`
 - `pthread_exit()`
 - `exit()` [terminates all threads] , `RET` [terminates current thread]
 - Synchronizing access to shared variables
 - `pthread_mutex_init`
 - `pthread_mutex_[un]lock`

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
hello.c
```

```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
hello.c
```

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c



```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

Thread ID

Thread attributes
(usually NULL)

```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

Thread ID

Thread attributes
(usually NULL)

Thread routine

```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

Thread ID

Thread attributes
(usually NULL)

Thread routine

Thread arguments
(void *p)

hello.c

```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

Thread ID

Thread attributes
(usually NULL)

Thread routine

Thread arguments
(void *p)

hello.c

```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

Return value
(void **p)

hello.c

Execution of Threaded “hello, world”

Main thread



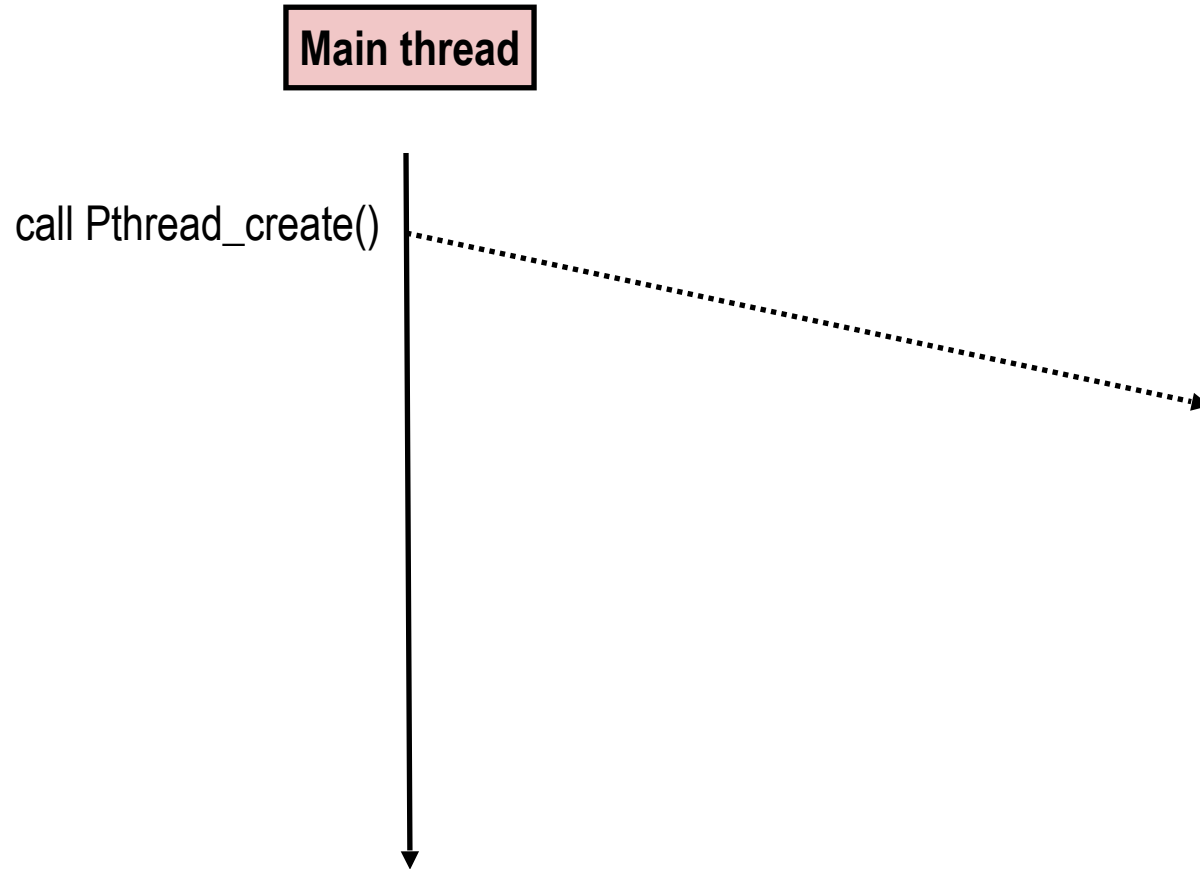
Execution of Threaded “hello, world”

Main thread

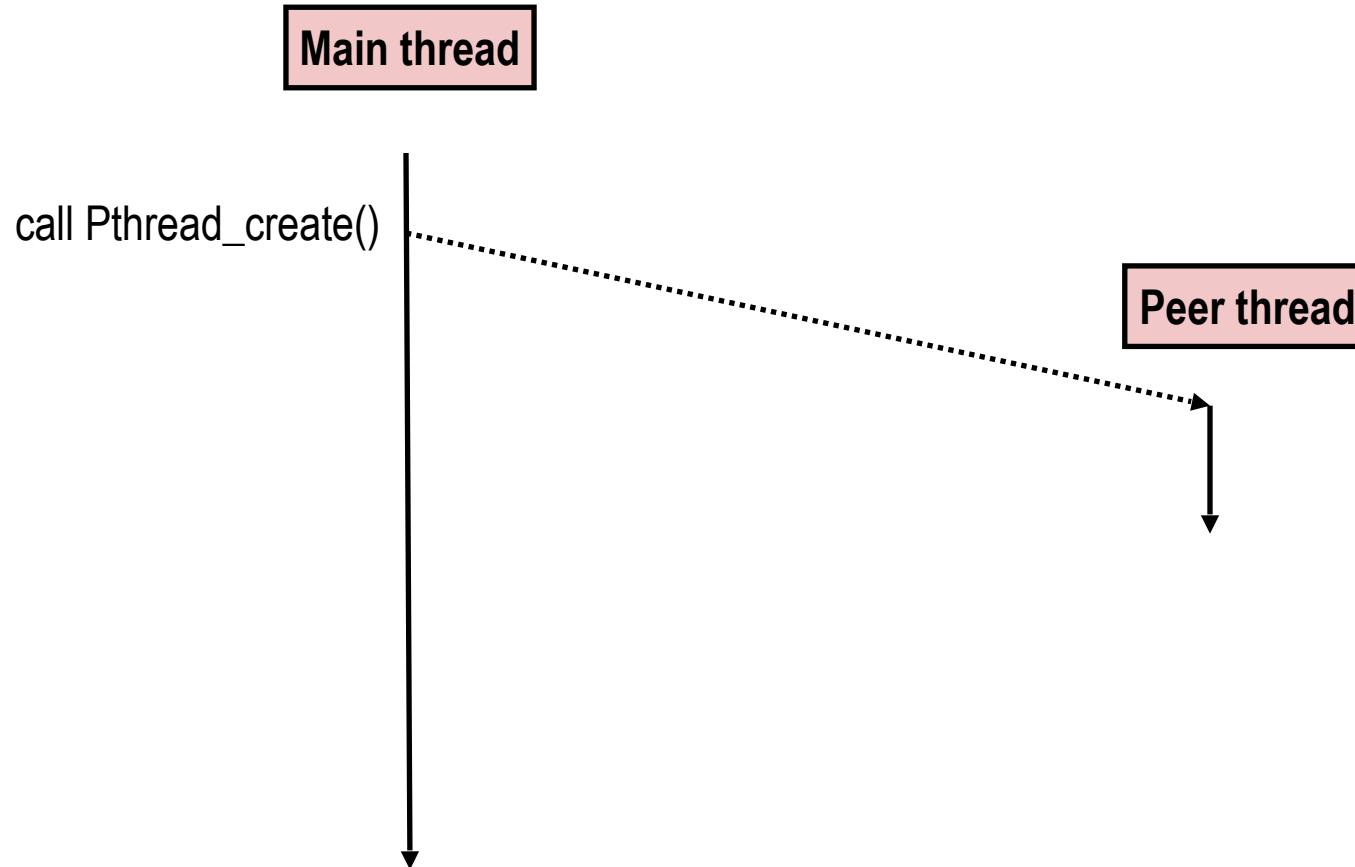
call Pthread_create()



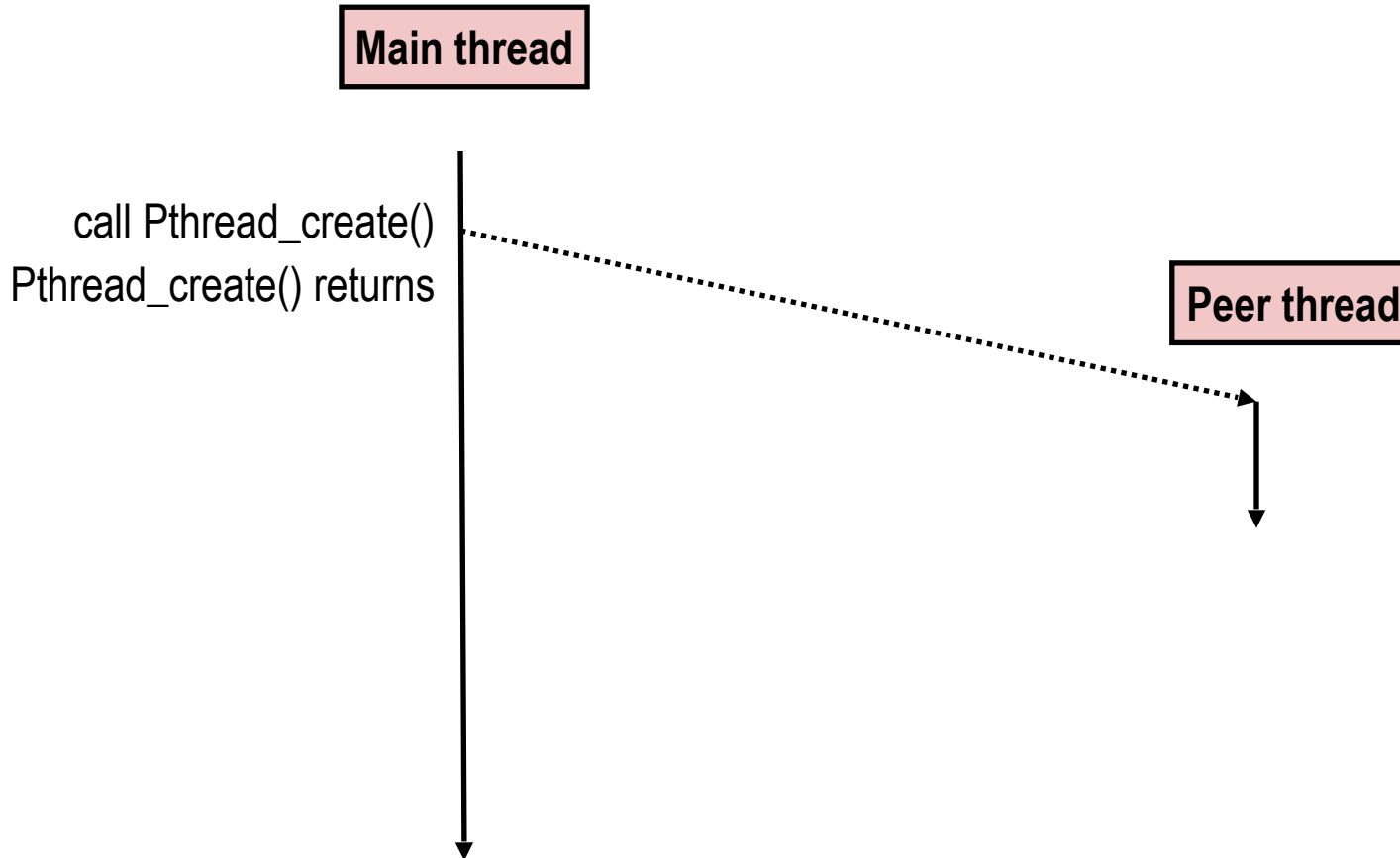
Execution of Threaded “hello, world”



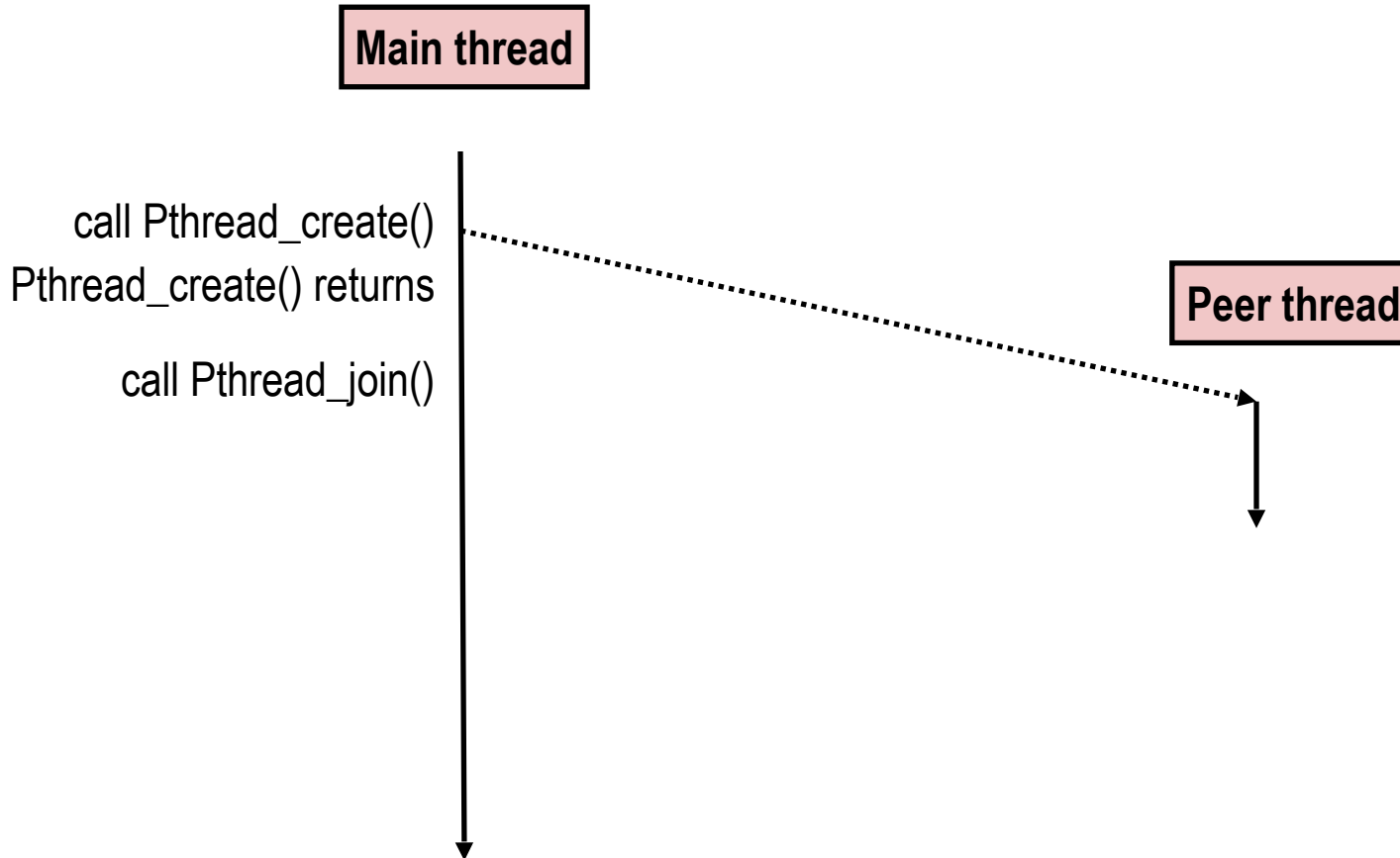
Execution of Threaded “hello, world”



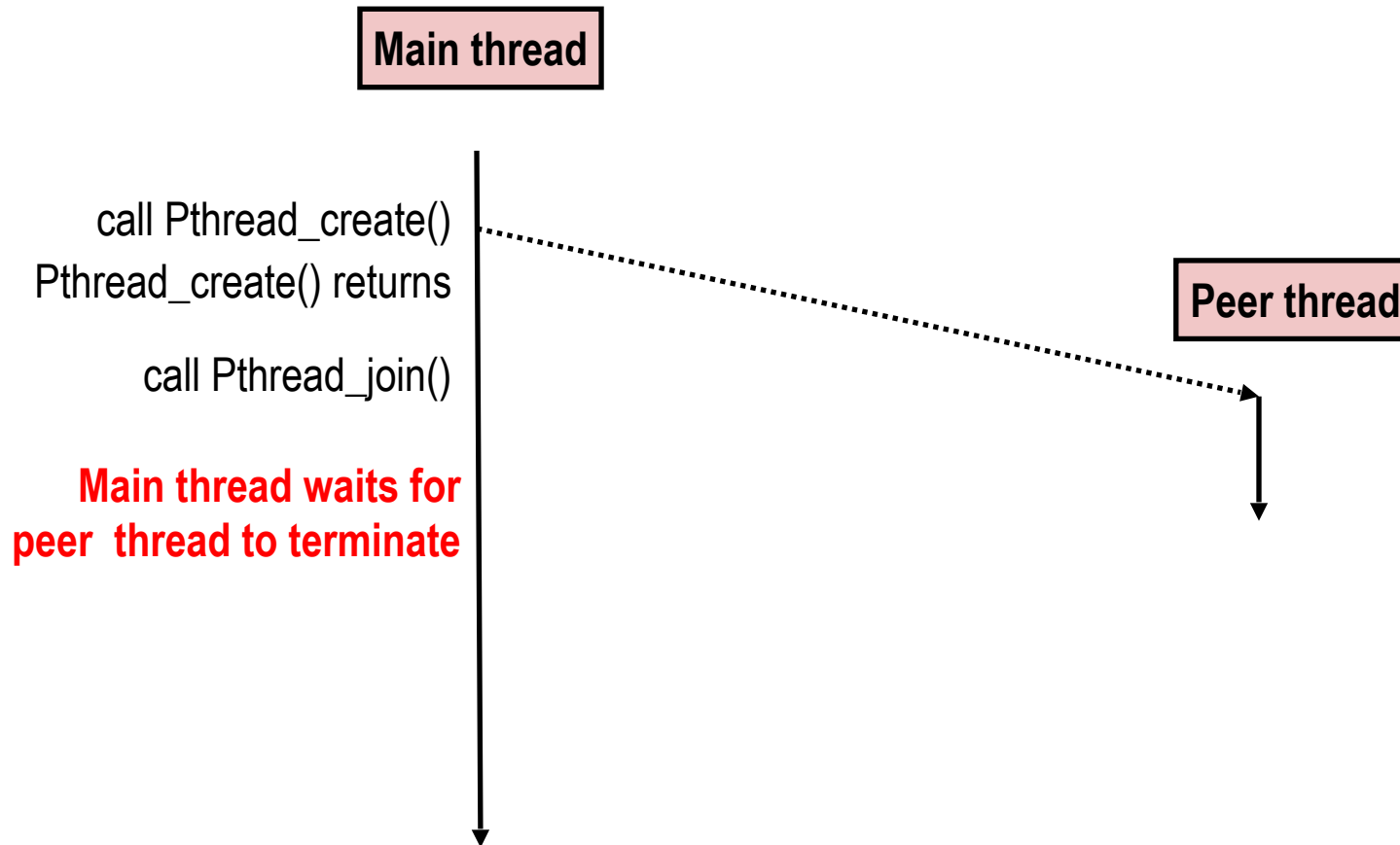
Execution of Threaded “hello, world”



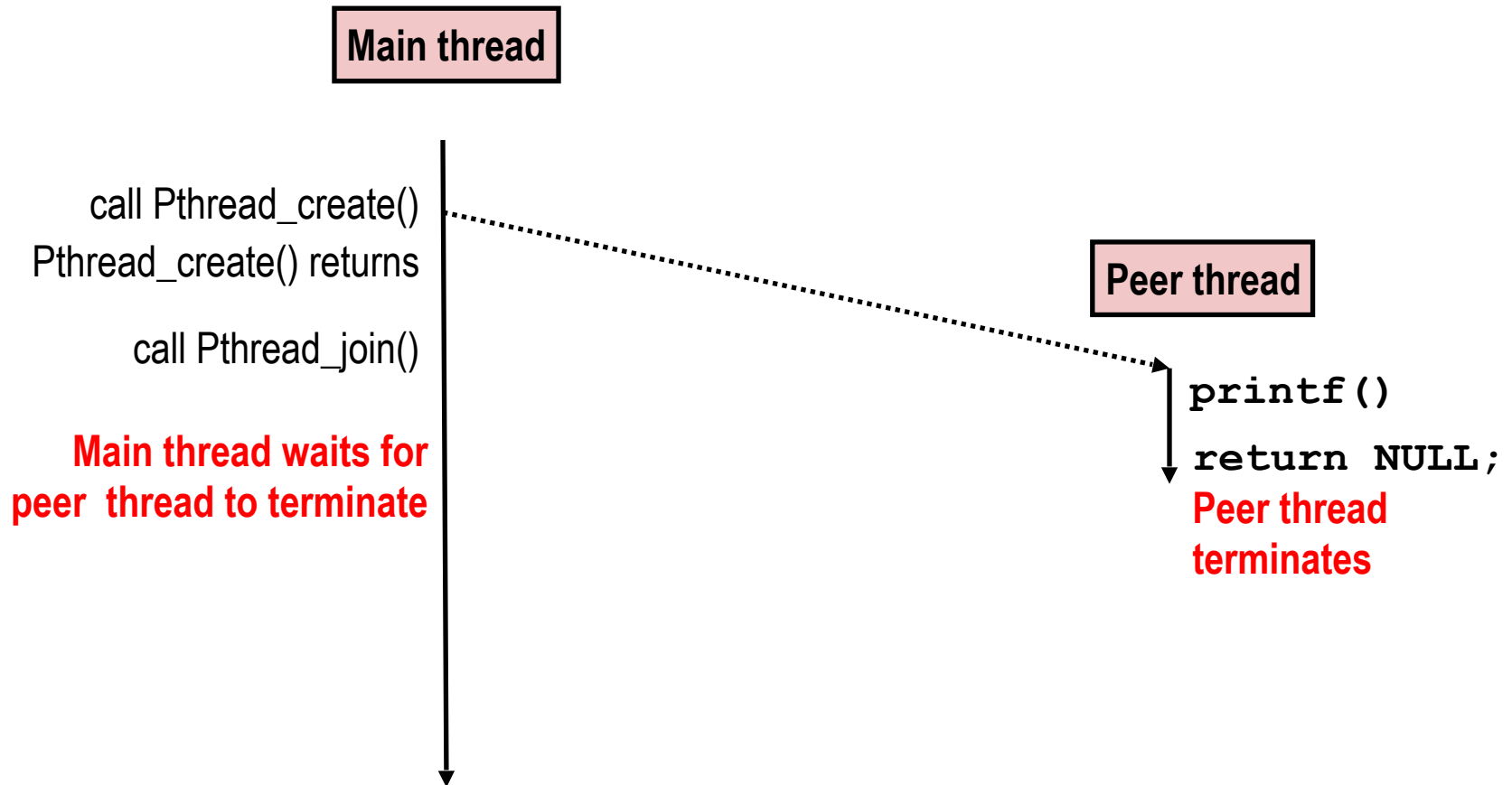
Execution of Threaded “hello, world”



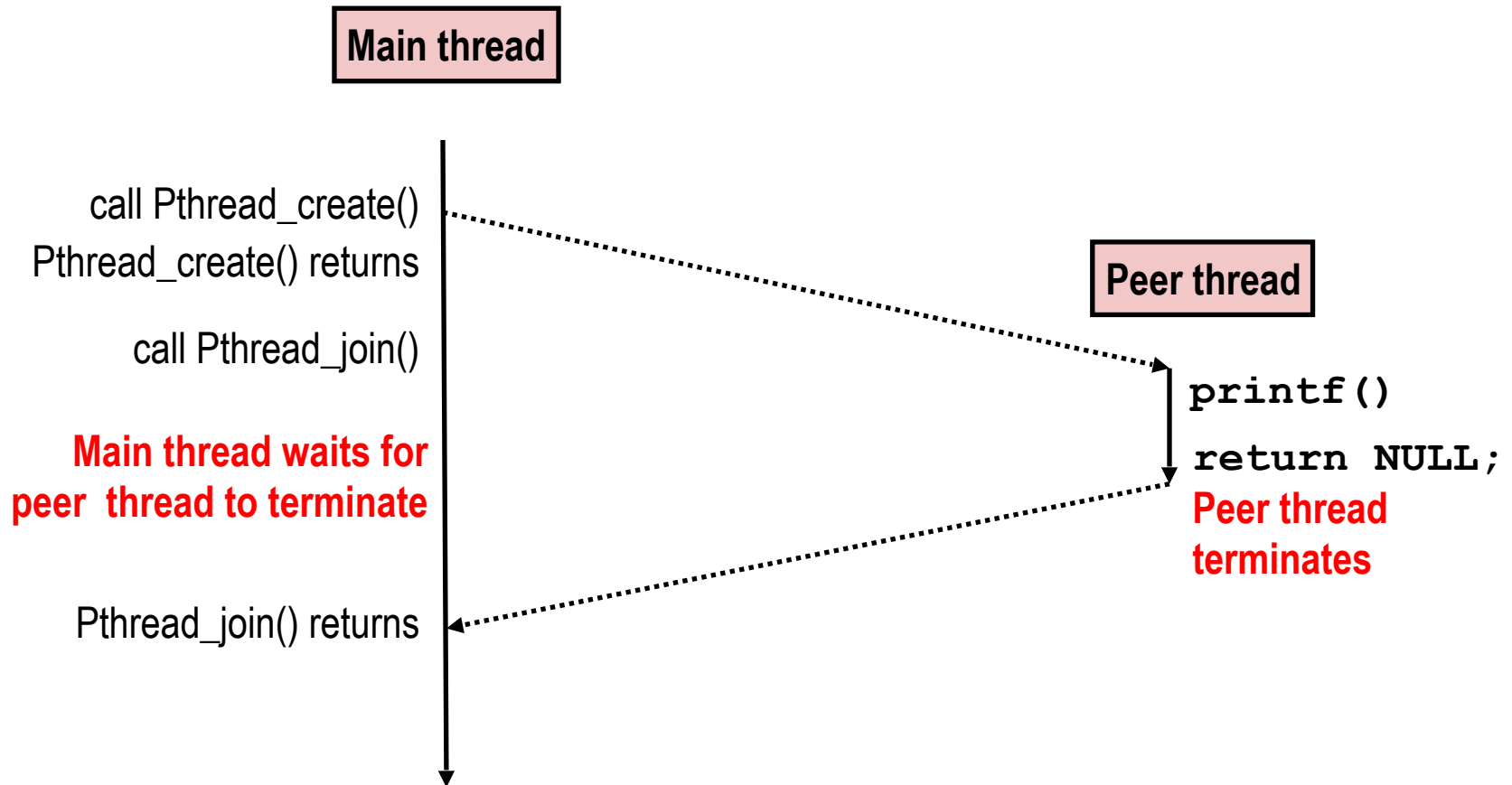
Execution of Threaded “hello, world”



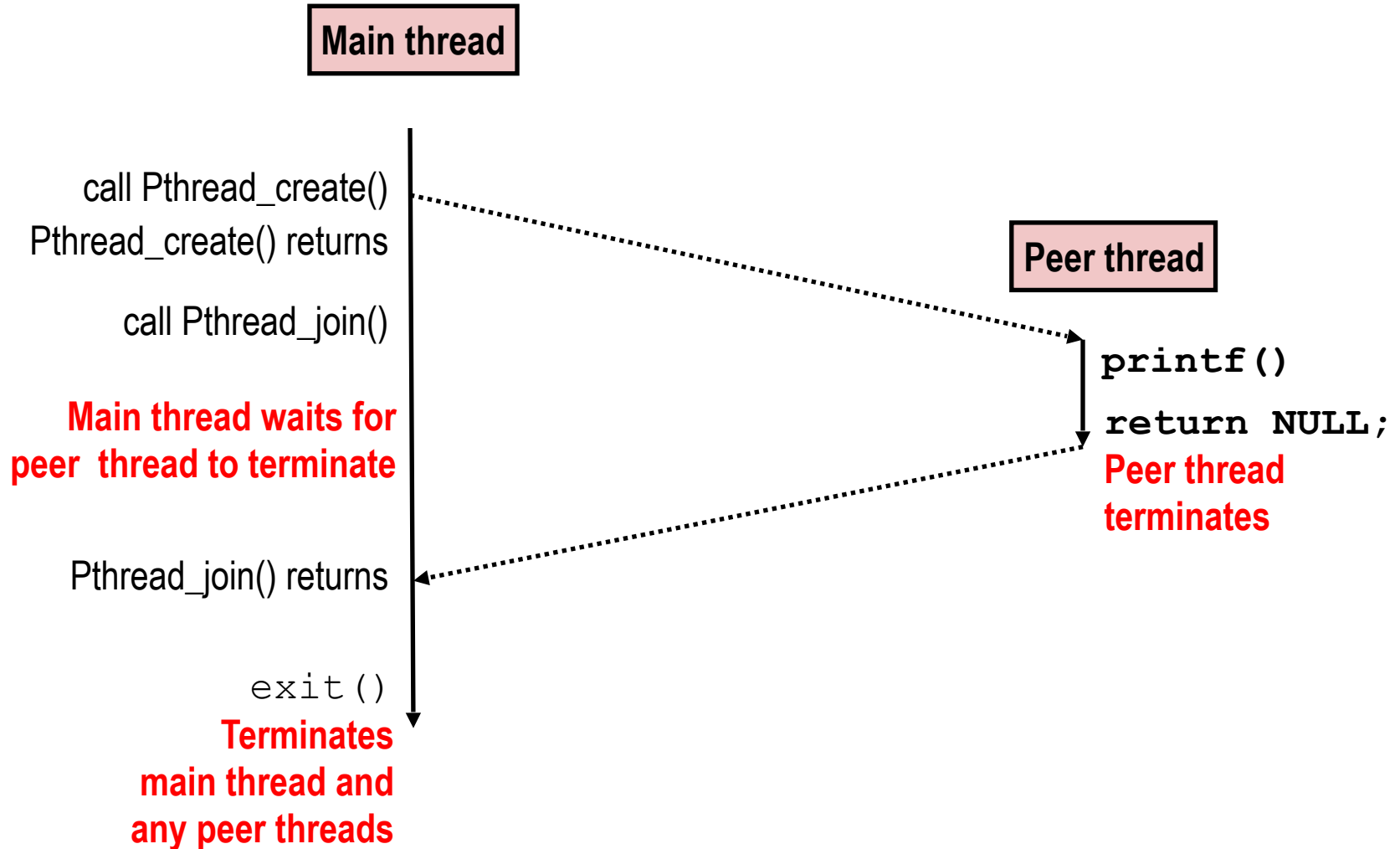
Execution of Threaded “hello, world”



Execution of Threaded “hello, world”



Execution of Threaded “hello, world”



Today

- From process to threads
 - Basic thread execution model
- **Multi-threading programming**
- Hardware support of threads
 - Multi-core
 - Hyper-threading
 - 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)

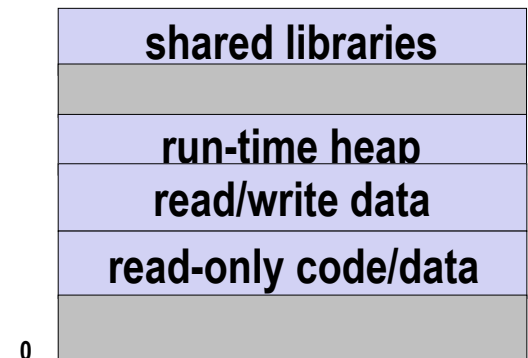
Thread 2 (peer thread)

stack 1

stack 2

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Thread 2 context:
Data registers
Condition codes
SP2
PC2



Kernel context:
VM structures
Descriptor table
brk pointer

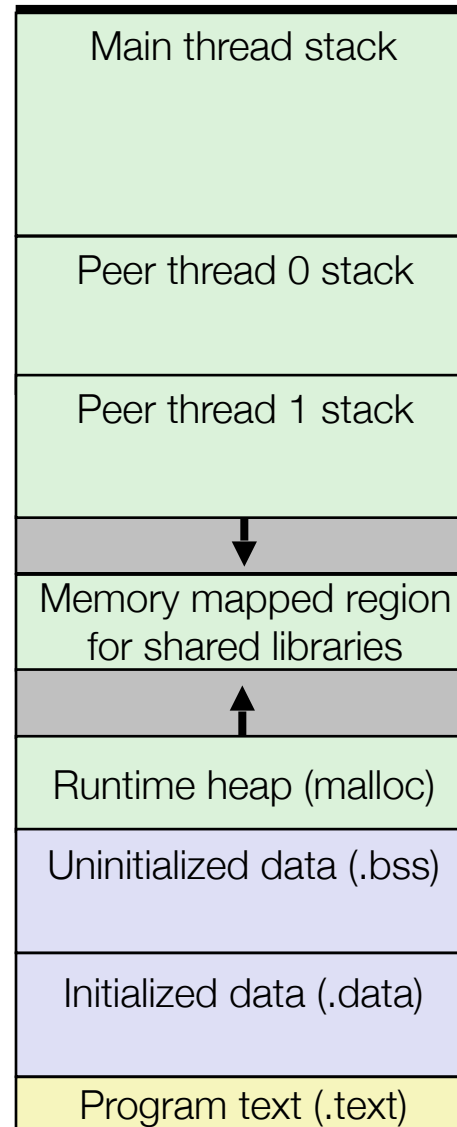
Example Program to Illustrate Sharing

```
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



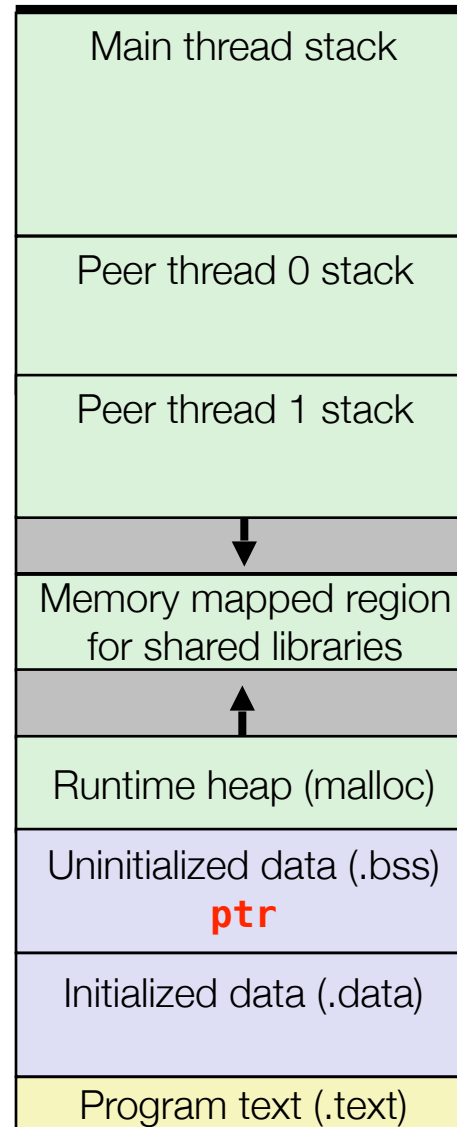
Example Program to Illustrate Sharing

```
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



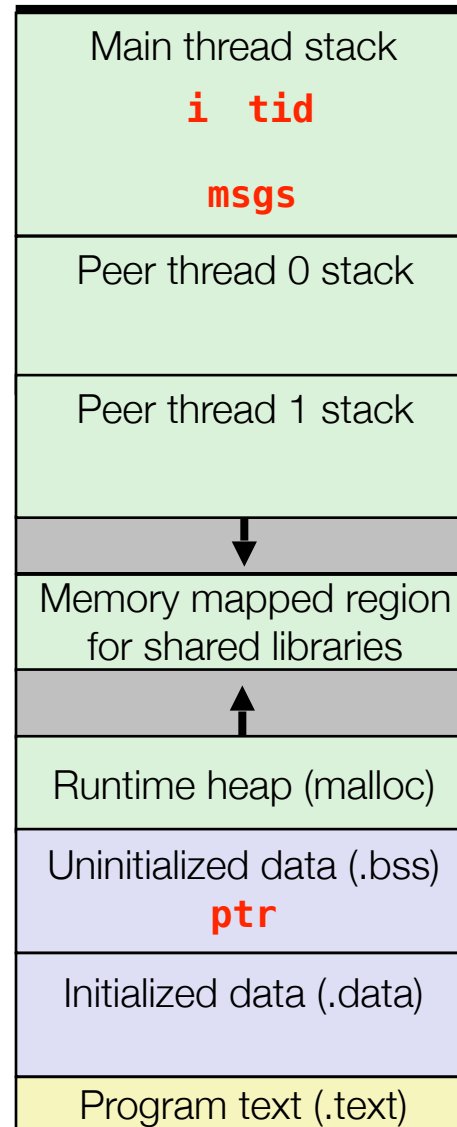
Example Program to Illustrate Sharing

```
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



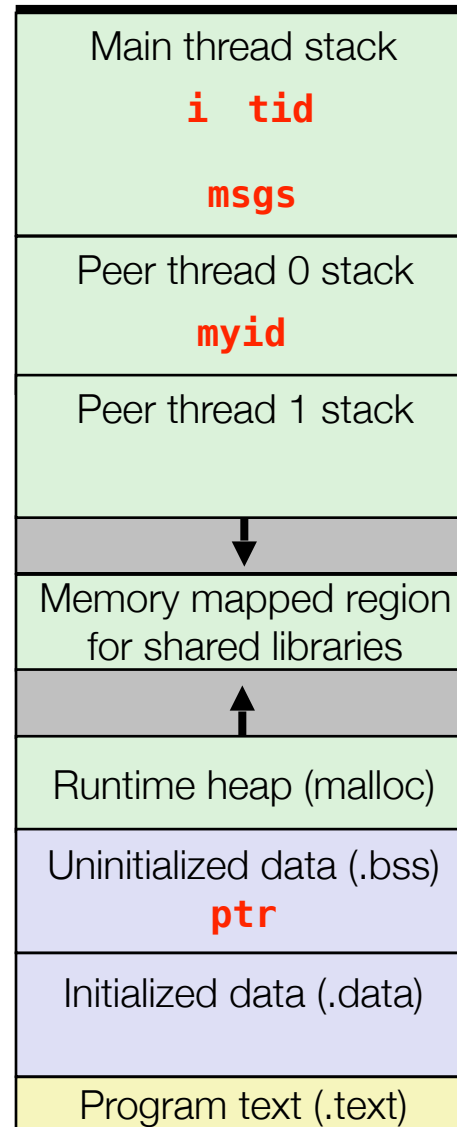
Example Program to Illustrate Sharing

```
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



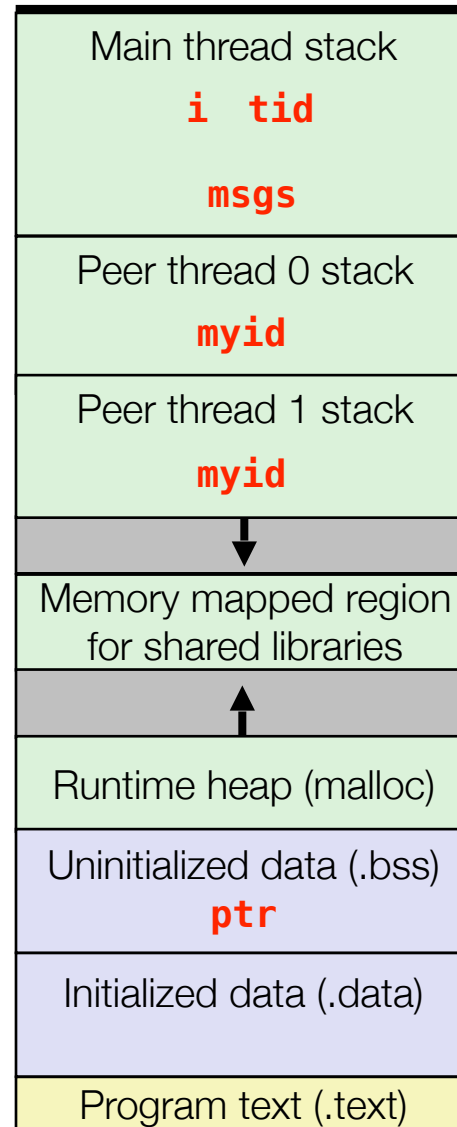
Example Program to Illustrate Sharing

```
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



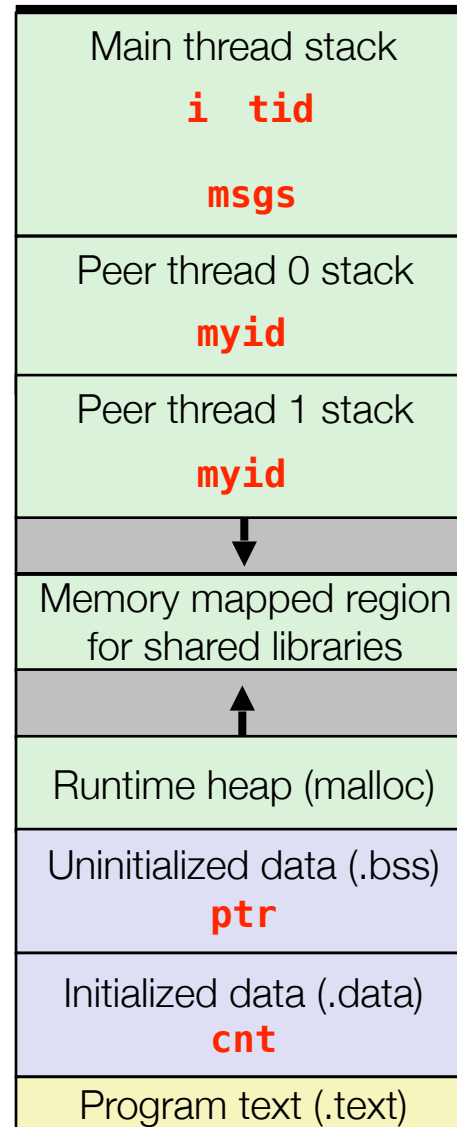
Example Program to Illustrate Sharing

```
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



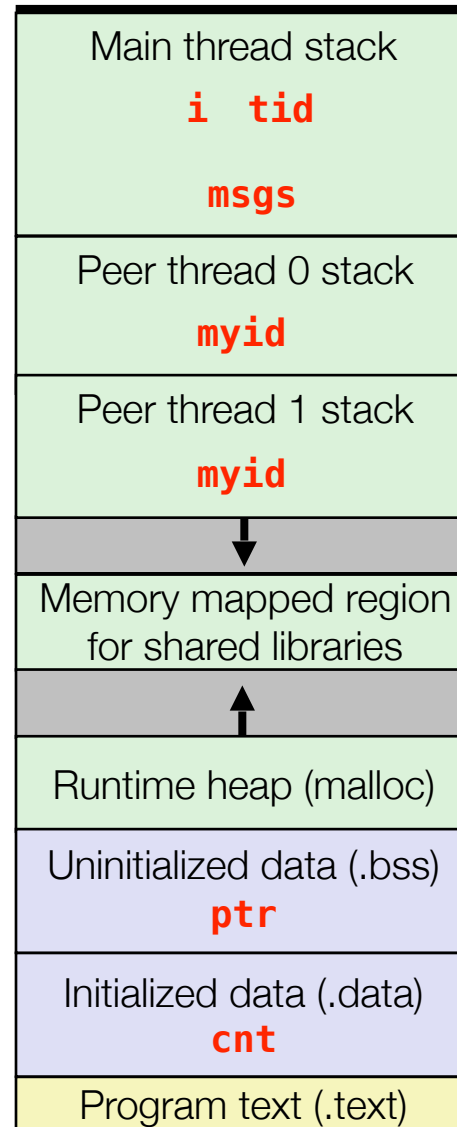
Example Program to Illustrate Sharing

```
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



main

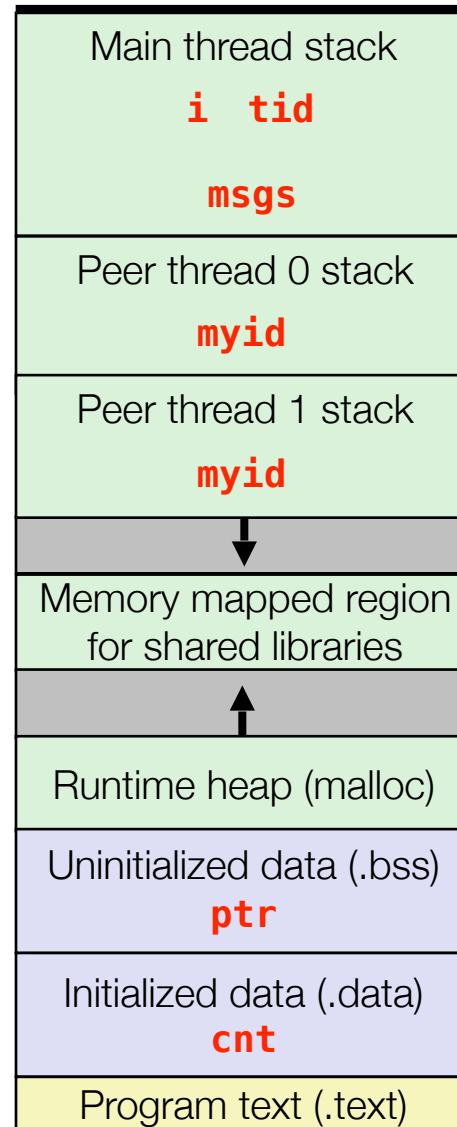
Example Program to Illustrate Sharing

```
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



main

p0 p1 main

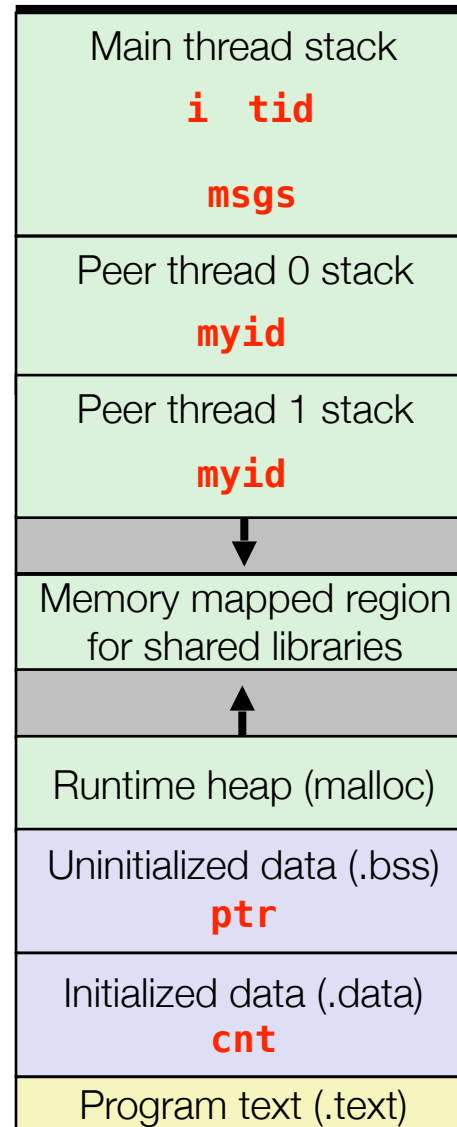
Example Program to Illustrate Sharing

```
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



main

p0 p1 main

p0

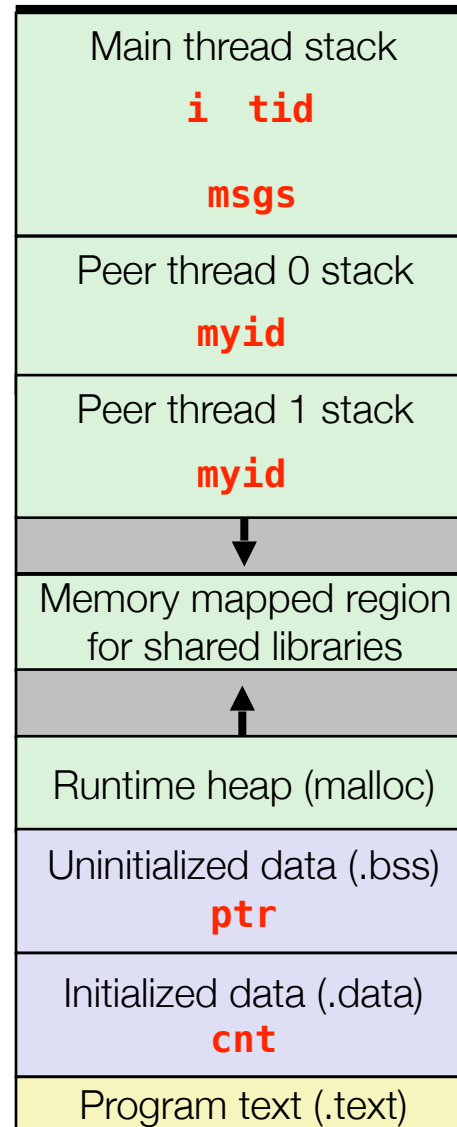
Example Program to Illustrate Sharing

```
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



main

p0 p1 main

p0

p1

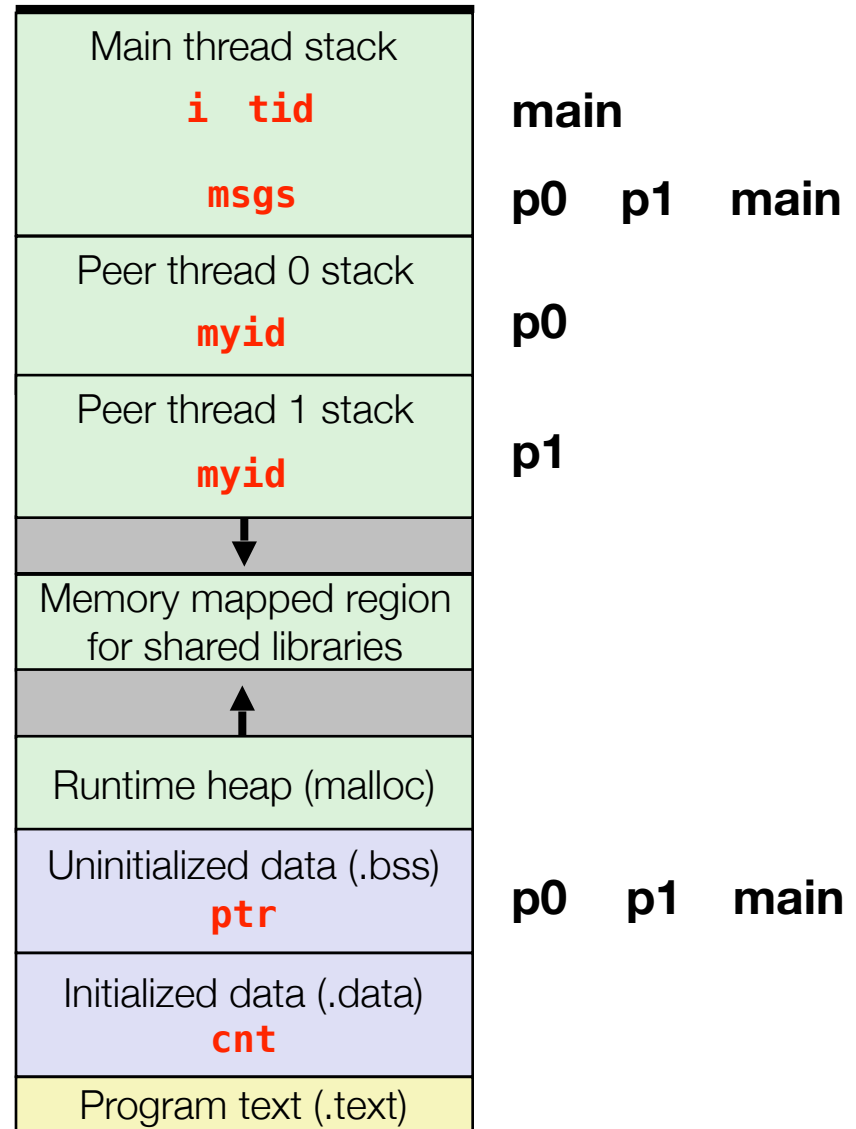
Example Program to Illustrate Sharing

```
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

```

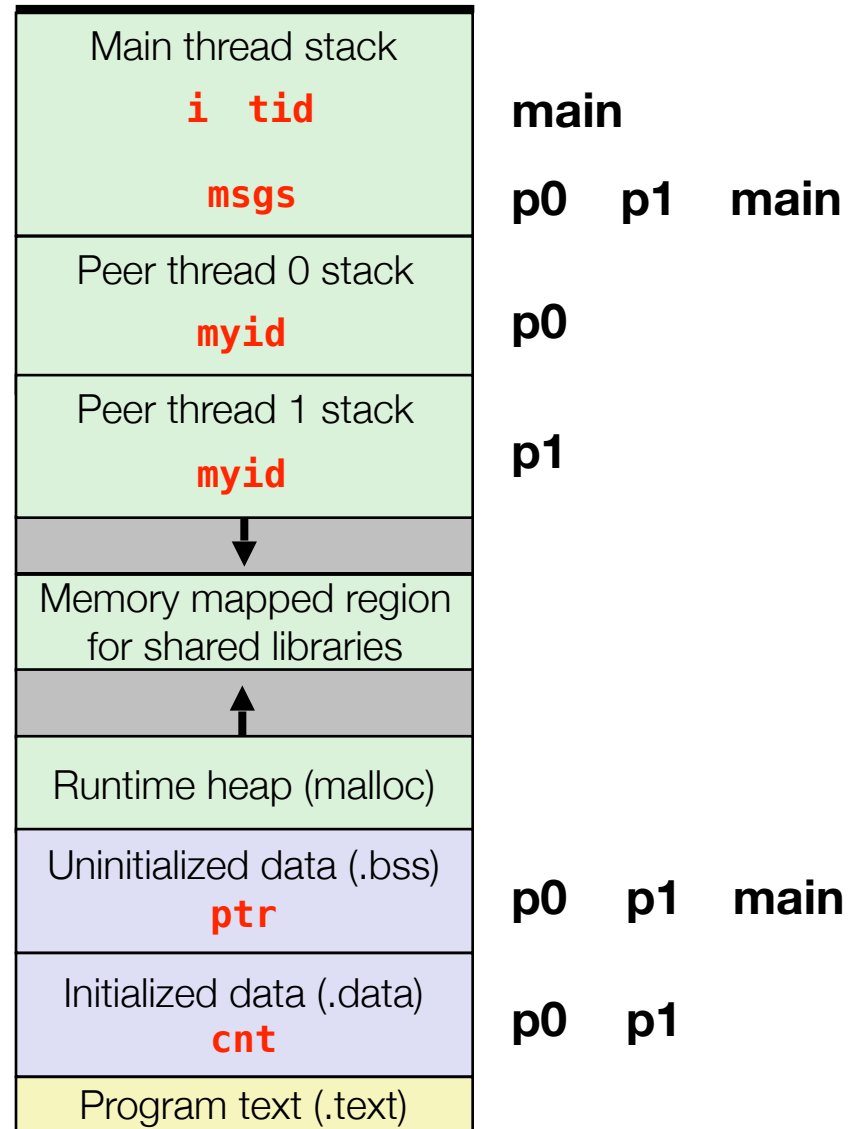
char **ptr; /* global var */

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



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

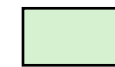
<pre>movq (%rdi), %rcx testq %rcx, %rcx jle .L2 movl \$0, %eax</pre>	} H_i : Head
<pre>.L3: movq cnt(%rip), %rdx addq \$1, %rdx movq %rdx, cnt(%rip)</pre>	
<pre>addq \$1, %rax cmpq %rcx, %rax jne .L3 .L2:</pre>	} T_i : Tail

Concurrent Execution

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

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L ₁	0	-	0
1	U ₁	1	-	0
1	S ₁	1	-	1
2	L ₂	-	1	1
2	U ₂	-	2	1
2	S ₂	-	2	2



**Thread 1
critical section**



**Thread 2
critical section**

Concurrent Execution (cont)

- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L ₁	0	-	0
1	U ₁	1	-	0
2	L ₂	-	0	0
1	S ₁	1	-	1
2	U ₂	-	1	1
2	S ₂	-	1	1

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L ₁			
2	L ₂			
2	U ₂			
2	S ₂			
1	U ₁			
1	S ₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂			
2	U₂			
2	S₂			
1	U₁			
1	S₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L ₁	0		
2	L ₂		0	
2	U ₂			
2	S ₂			
1	U ₁			
1	S ₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂			
1	U₁			
1	S₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂		1	
1	U₁			
1	S₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂		1	1
1	U₁			
1	S₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂		1	1
1	U₁	1		
1	S₁			

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂		1	1
1	U₁	1		
1	S₁	1		

Concurrent Execution (cont)

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂		1	1
1	U₁	1		
1	S₁	1		1

Concurrent Execution (cont)

- Another undesired, but legal, interleaving

i (thread) instr_i %rdx₁ %rdx₂ cnt

1	L₁	0		
2	L₂		0	
2	U₂		1	
2	S₂		1	1
1	U₁	1		
1	S₁	1		1

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

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!
- Threads need to have *mutually exclusive* access to critical section

Asm code for thread i

critical
section
wrt cnt



<code>movq (%rdi), %rcx</code>	} H_i : Head
<code>testq %rcx, %rcx</code>	
<code>jle .L2</code>	
<code>movl \$0, %eax</code>	

<code>.L3:</code>	} L_i : Load cnt
<code>movq cnt(%rip), %rdx</code>	
<code>addq \$1, %rdx</code>	} U_i : Update cnt
<code>movq %rdx, cnt(%rip)</code>	
<code>addq \$1, %rax</code>	} S_i : Store cnt
<code>cmpq %rcx, %rax</code>	
<code>jne .L3</code>	} T_i : Tail
<code>.L2:</code>	

Enforcing Mutual Exclusion

- We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.
 - i.e., need to guarantee *mutually exclusive access* for each critical section.
- Classic solution:
 - Semaphores (Edsger Dijkstra)
- Other approaches
 - Mutex and condition variables
 - Monitors (Java)

Enforcing Mutual Exclusion

- We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.
 - i.e., need to guarantee *mutually exclusive access* for each critical section.
- Classic solution:
 - Semaphores (Edsger Dijkstra)
- Other approaches
 - Mutex and condition variables
 - Monitors (Java)

Again, take CSC 2/458 to learn more!!

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

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
- 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
 - 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
 - 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
- 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
- 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.

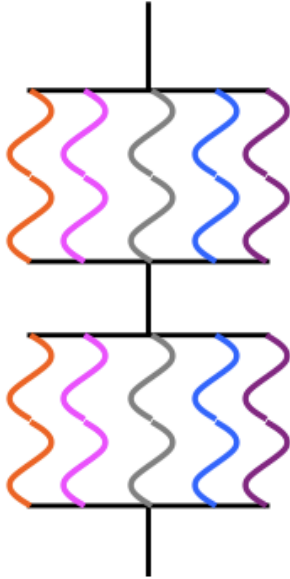
Today

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

Thread-level Parallelism (TLP)

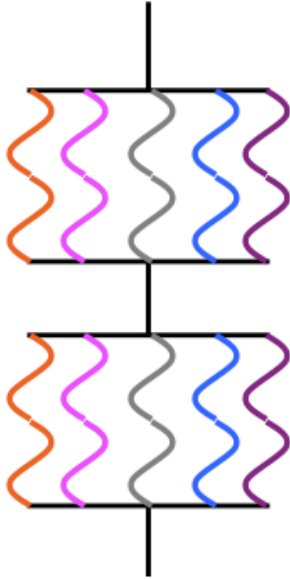
- Thread-Level Parallelism
 - Splitting a task into independent sub-tasks
 - Each thread is responsible for a sub-task
- Example: Parallel summation of N number
 - Should add up to $((n-1)*n)/2$
- Partition values $1, \dots, n-1$ into t ranges
 - $\lfloor n/t \rfloor$ values in each range
 - Each of 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?



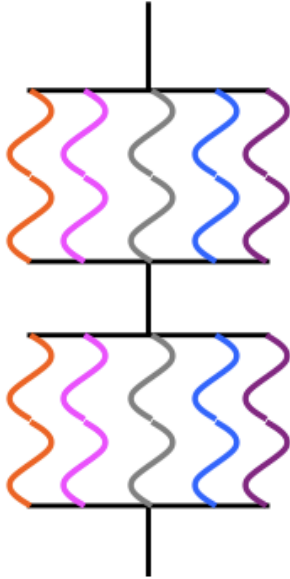
- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**

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?

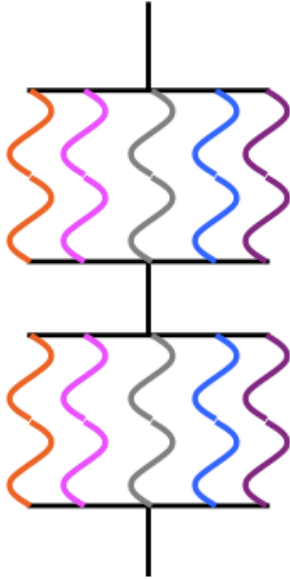


- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
 - e.g., Synchronization overhead

Each thread:

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

Why the Sequential Bottleneck?

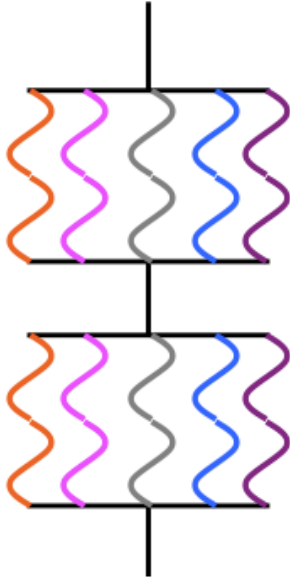


- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
 - e.g., Synchronization overhead

Each thread:

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

Why the Sequential Bottleneck?

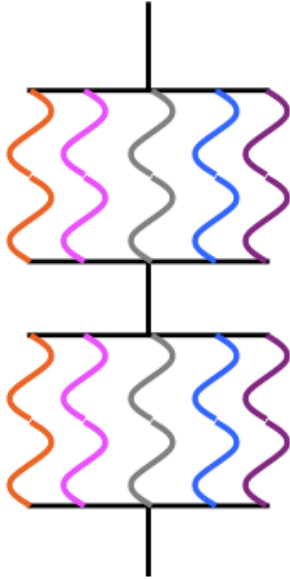


- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
 - e.g., Synchronization overhead

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 {
```

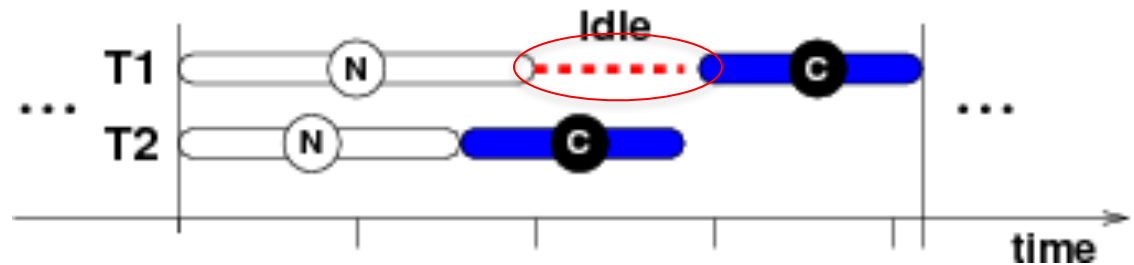
```
  Compute N
```

```
  P(A)
```

```
  Update shared data
```

```
  V(A) C
```

```
}
```



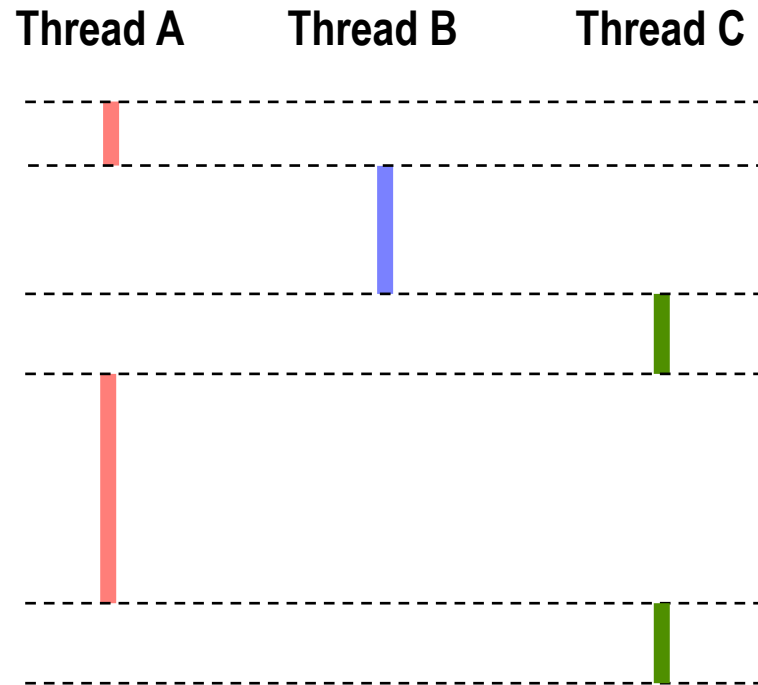
Can A Single Core Support Multi-threading?

- Need to multiplex between different threads (time slicing)

Sequential

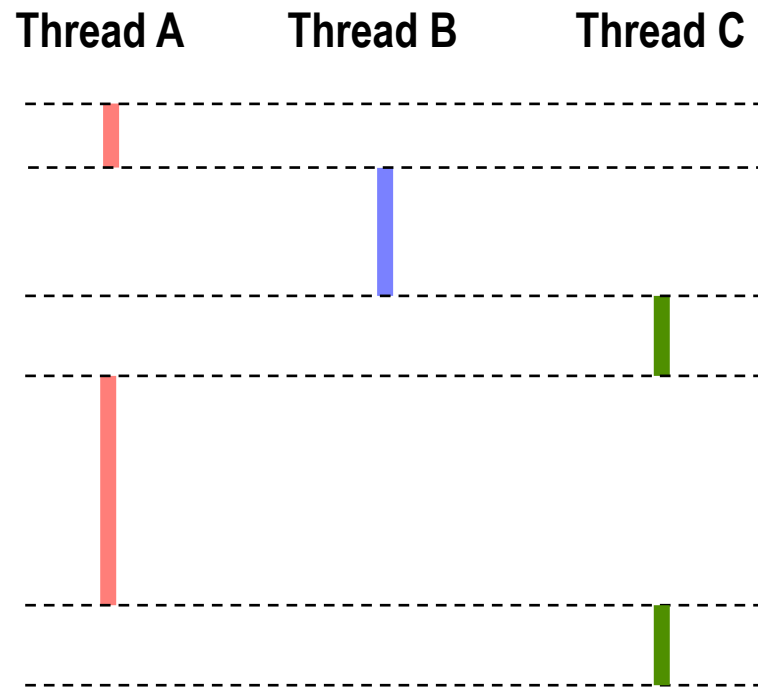


Multi-threaded



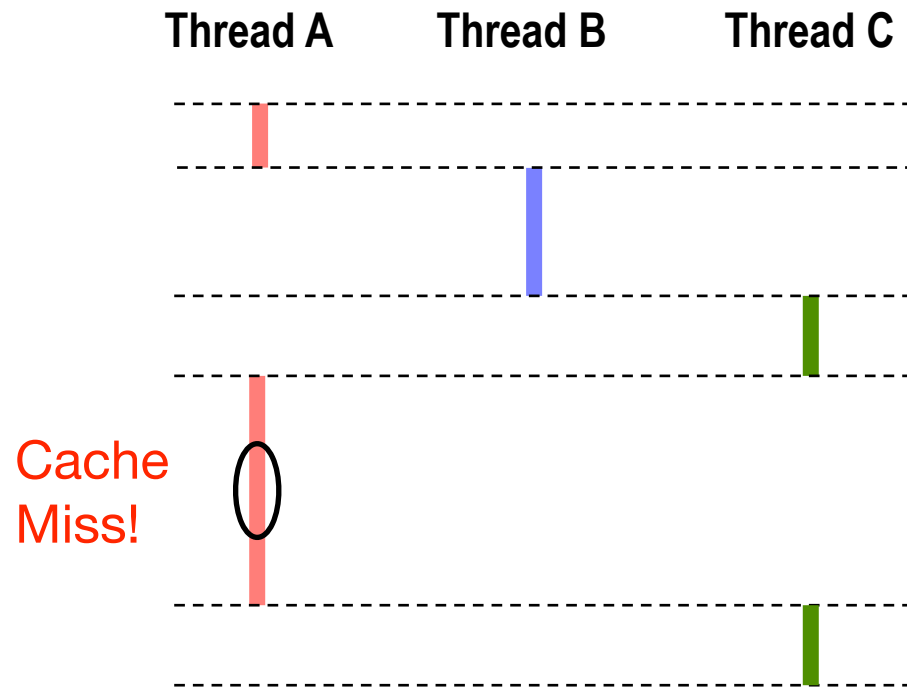
Any benefits?

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



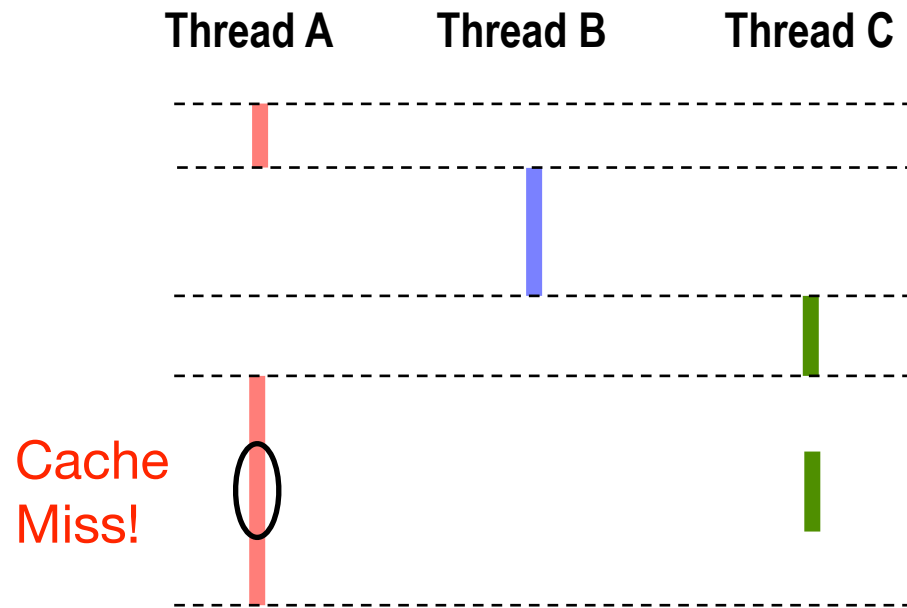
Any benefits?

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



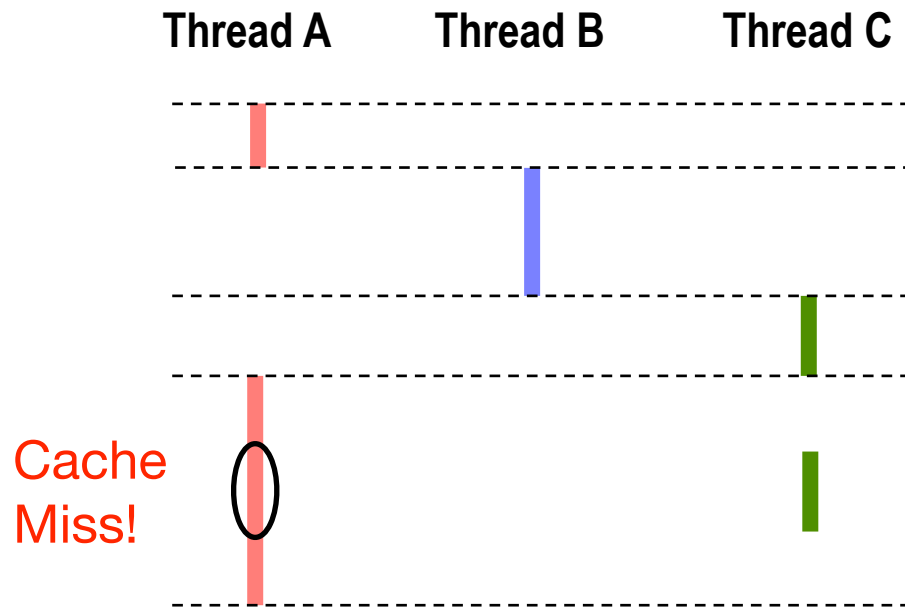
Any benefits?

- 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. Seminal paper that shows that using multi-threading can avoid branch prediction.

When to Switch?

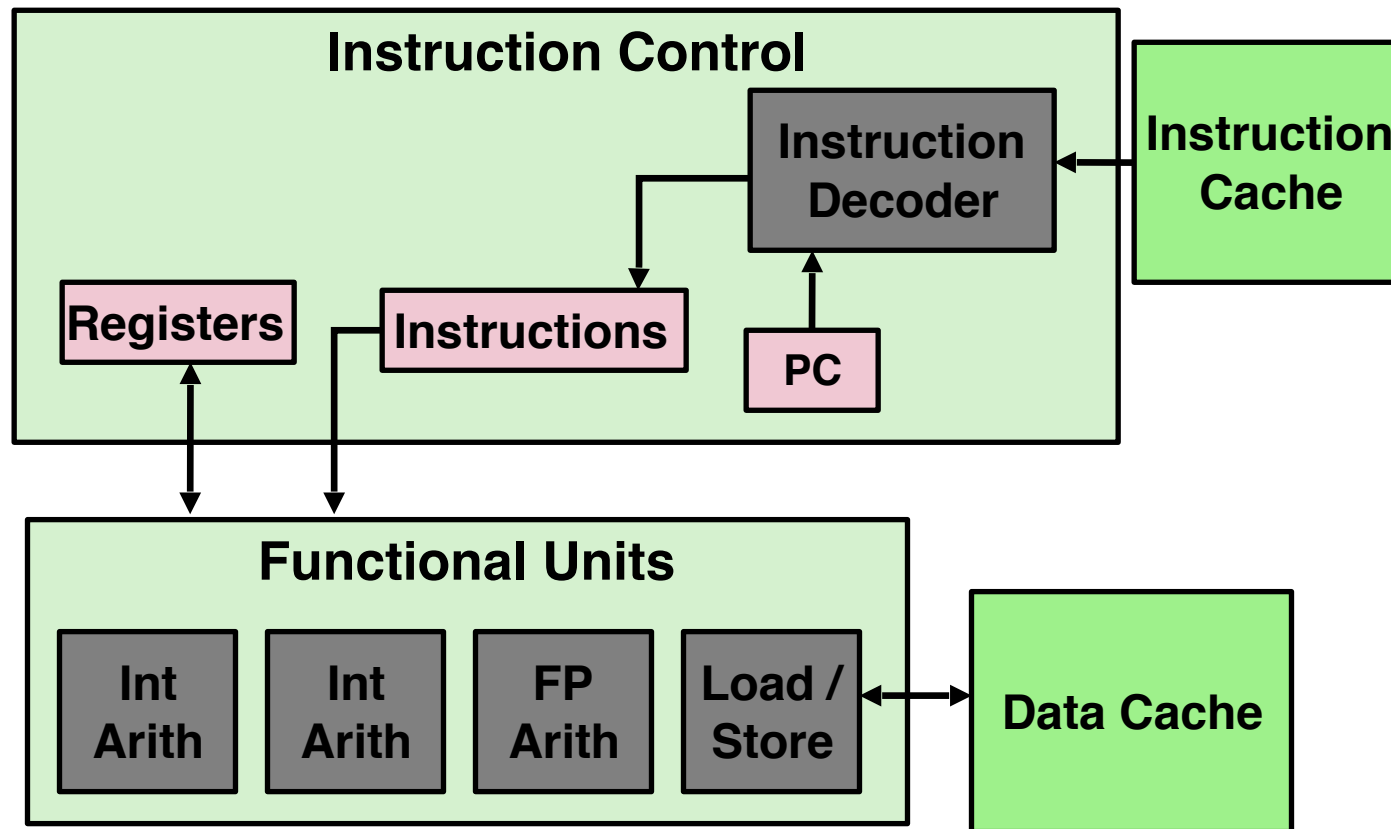
- **Coarse grained**
 - Event based, e.g., switch on L3 cache miss
 - Quantum based (every thousands of cycles)
- **Fine grained**
 - Cycle by cycle
 - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
 - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. Seminal paper that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

When to Switch?

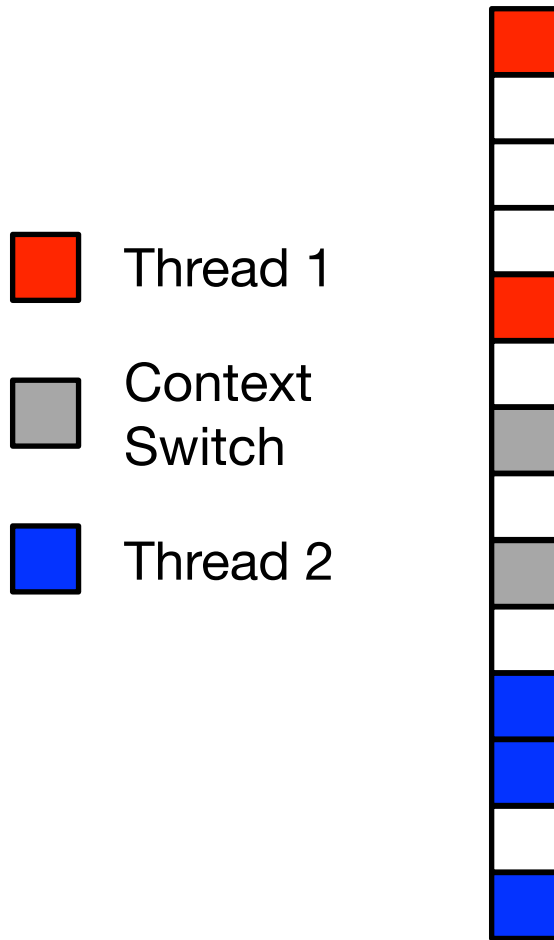
- **Coarse grained**
 - Event based, e.g., switch on L3 cache miss
 - Quantum based (every thousands of cycles)
- **Fine grained**
 - Cycle by cycle
 - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
 - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. Seminal paper that shows that using multi-threading can avoid branch prediction.
- **Either way, need to save/restore thread context upon switching**
- **One great thing about fine-grained switching: no need for branch prediction!!**

Single-Core Internals

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






Conventional Multi-threading

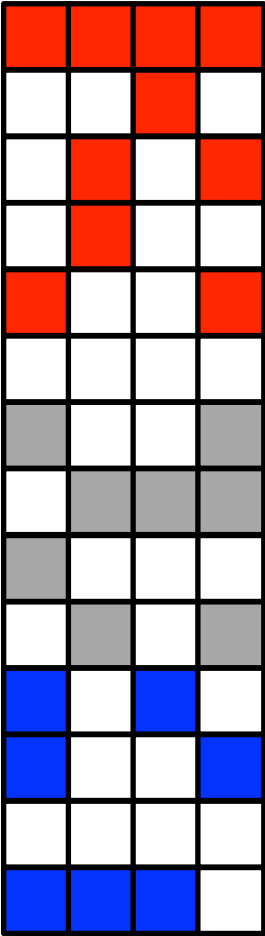


Conventional Multi-threading

Functional Units

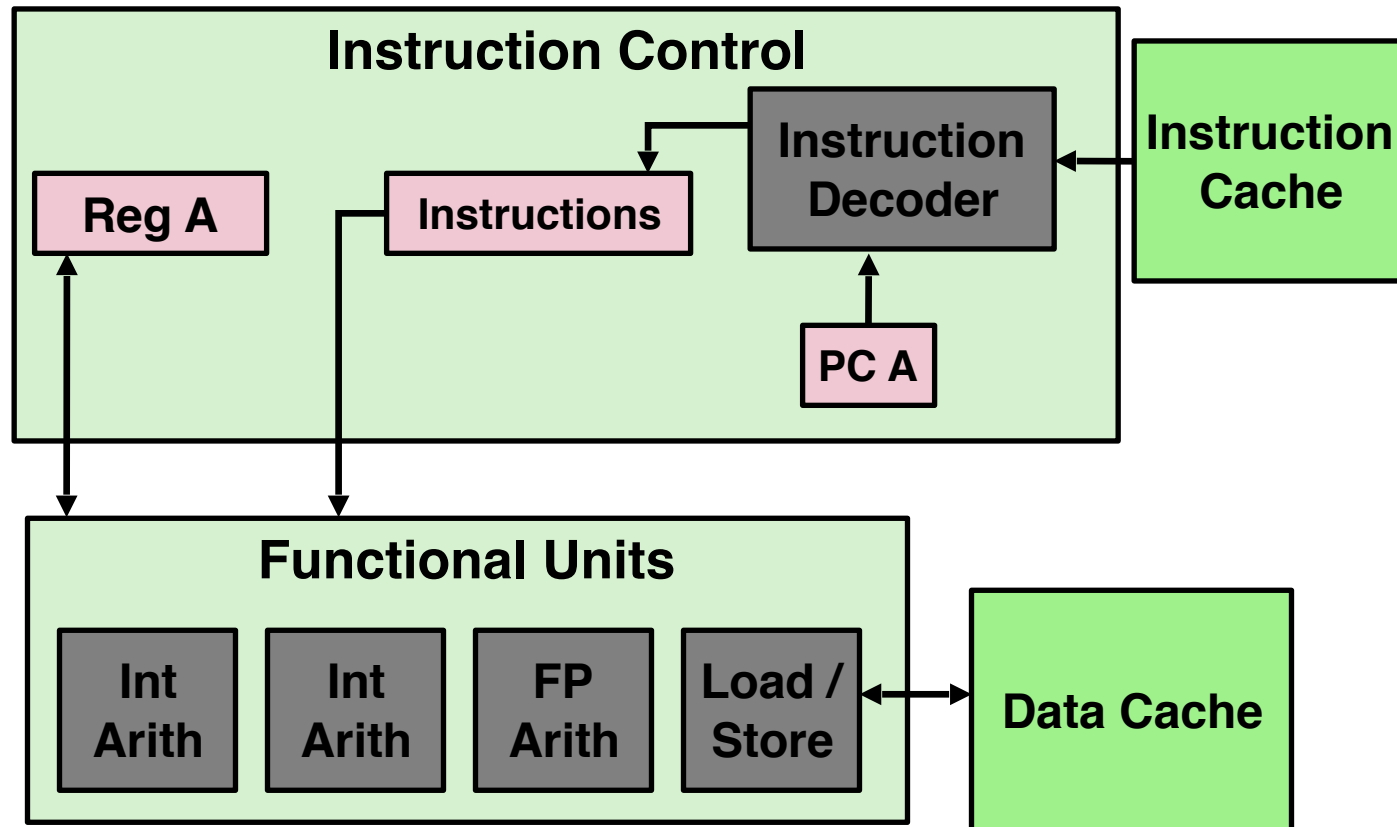


-  Thread 1
-  Context Switch
-  Thread 2



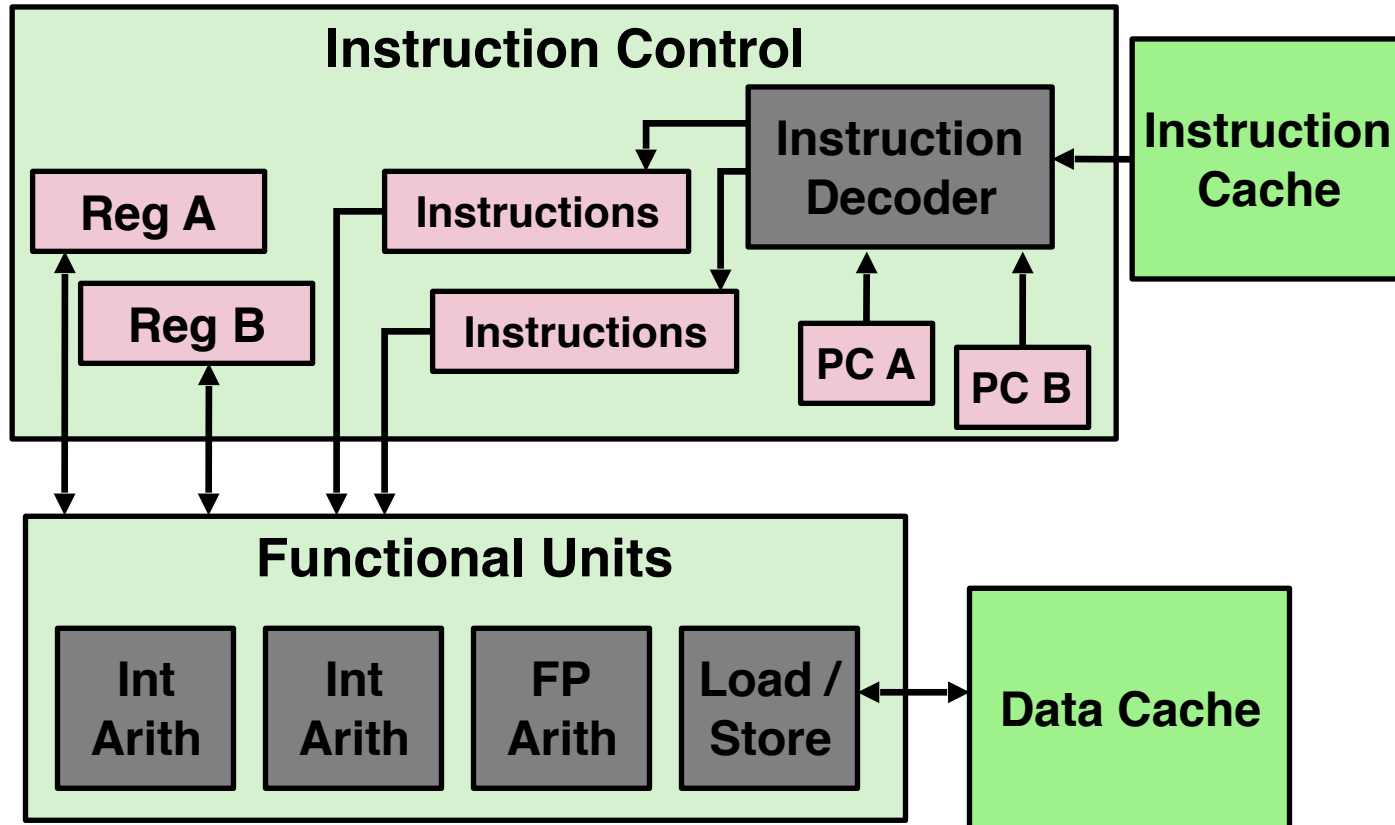
Hyper-threading

- Intel's terminology. More commonly known as: Simultaneous Multi-threading (SMT)
- Replicate enough hardware structures to process K instruction streams
- K copies of all registers. Share functional units



Hyper-threading




- Intel's terminology. More commonly known as: Simultaneous Multi-threading (SMT)
- Replicate enough hardware structures to process K instruction streams
- K copies of all registers. Share functional units

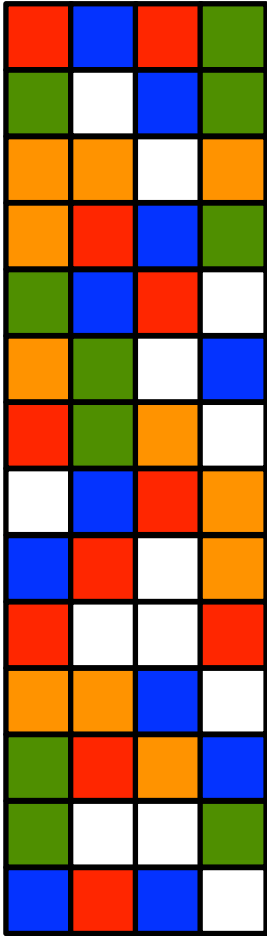
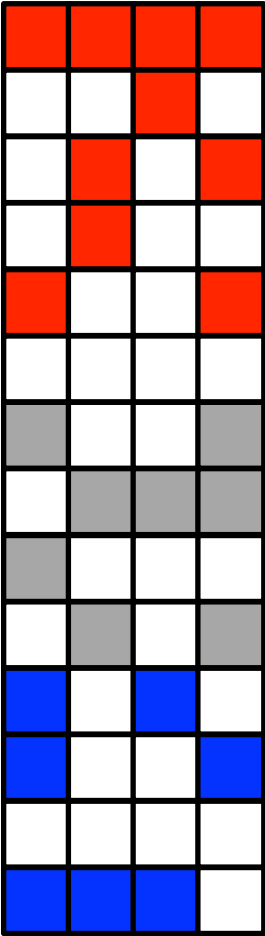


Conventional Multi-threading vs. Hyper-threading

Conventional Multi-threading

Hyper-threading




-  Thread 1
-  Context Switch
-  Thread 2

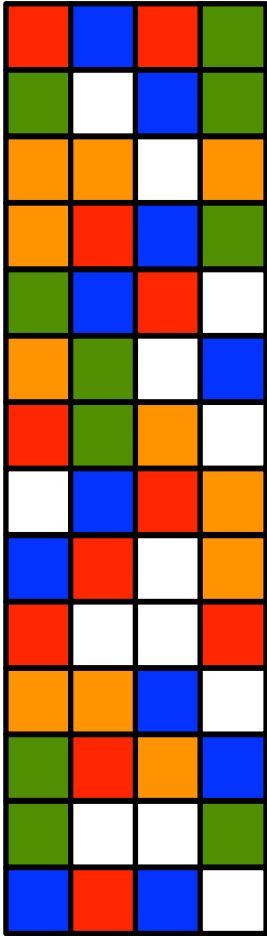
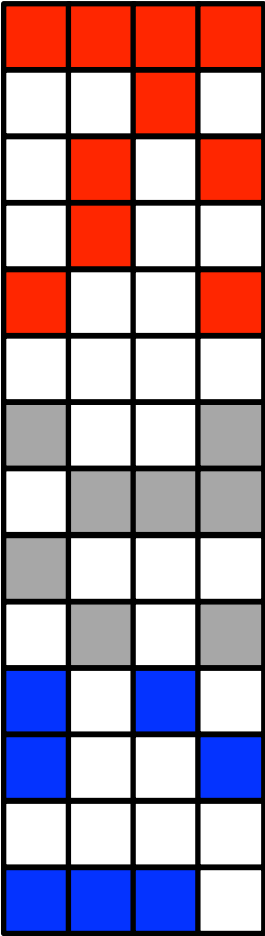


Conventional Multi-threading vs. Hyper-threading

Conventional Multi-threading

Hyper-threading

-  Thread 1
-  Context Switch
-  Thread 2






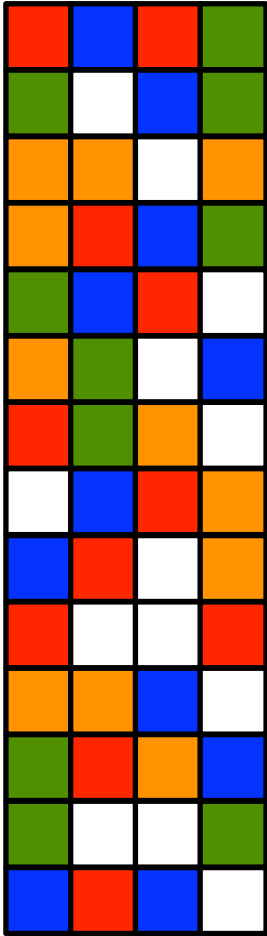
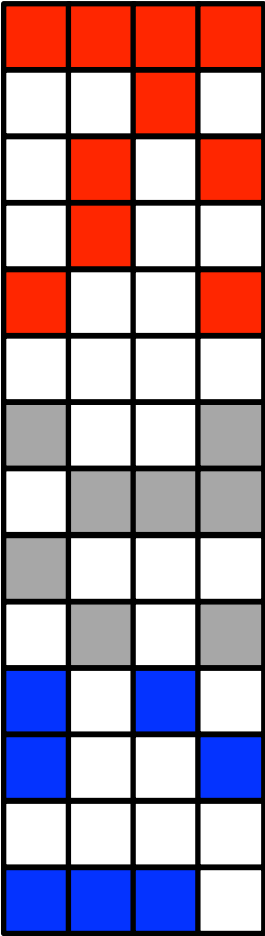
Multiple threads actually execute in parallel (even with one single core)

Conventional Multi-threading vs. Hyper-threading

Conventional Multi-threading

Hyper-threading

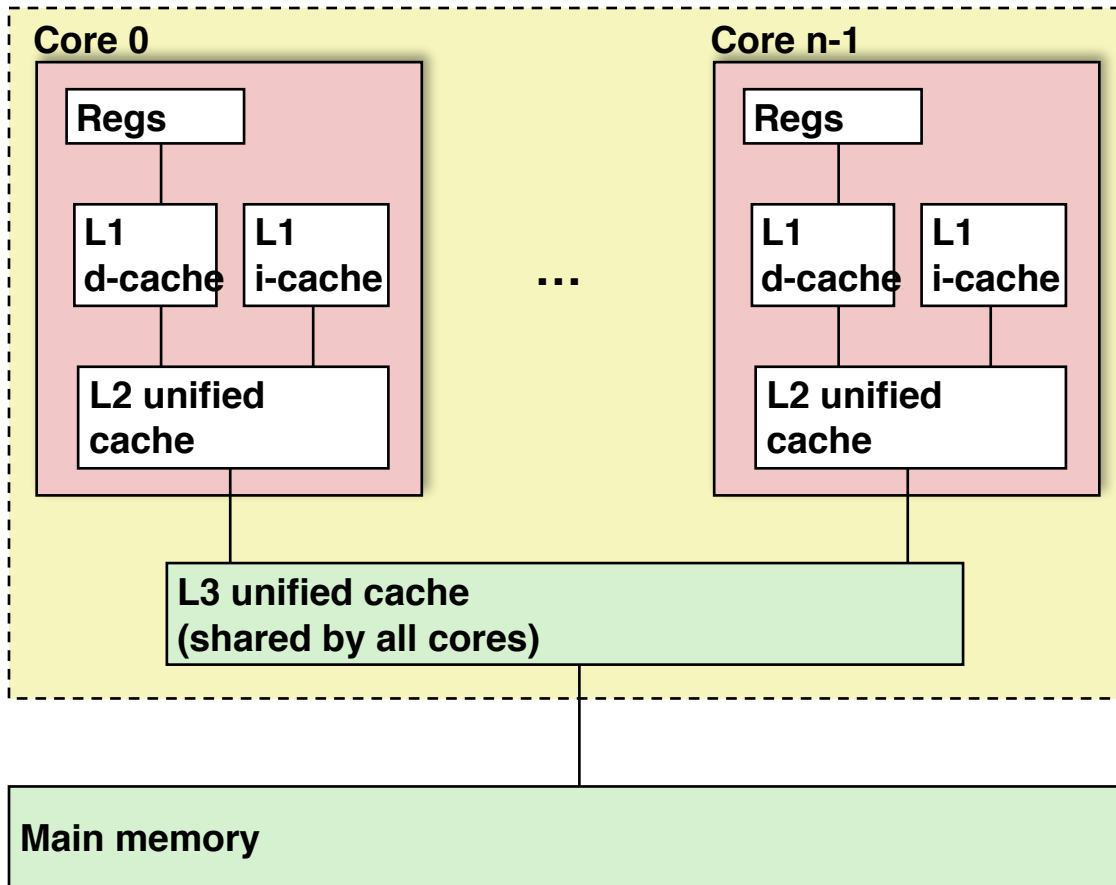
-  Thread 1
-  Context Switch
-  Thread 2



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

No/little context switch overhead

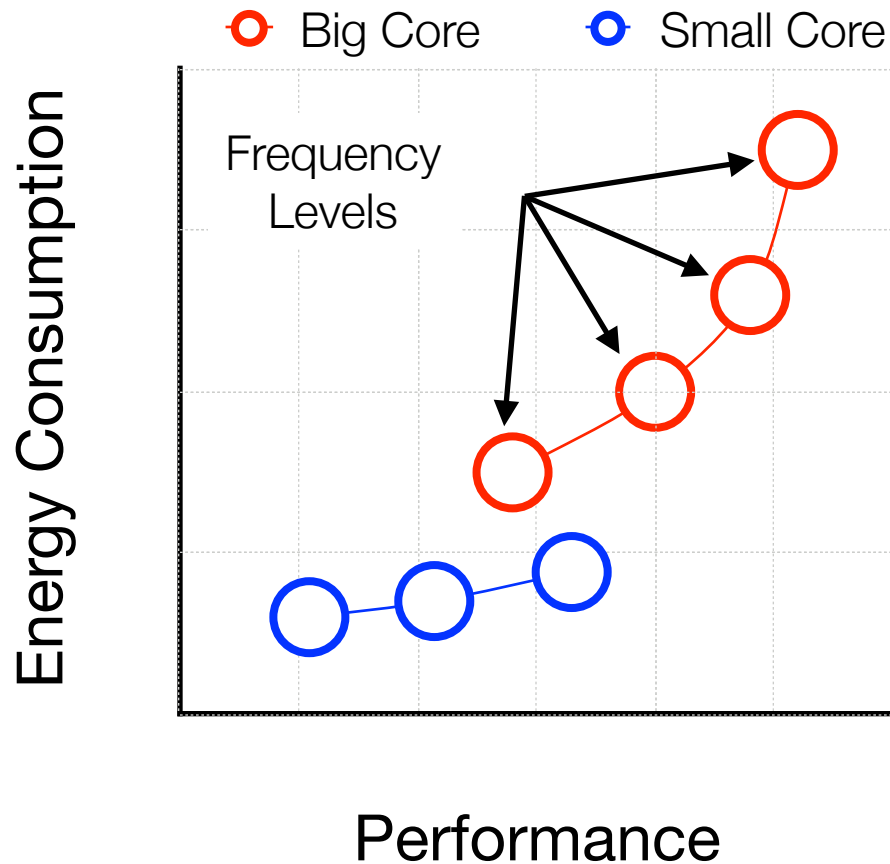
Typical Multi-core Processor



- Traditional multiprocessing: symmetric multiprocessor (SMP)
- Every core is exactly the same. Private registers, L1/L2 caches, etc.
- Share L3 (LLC) and main memory

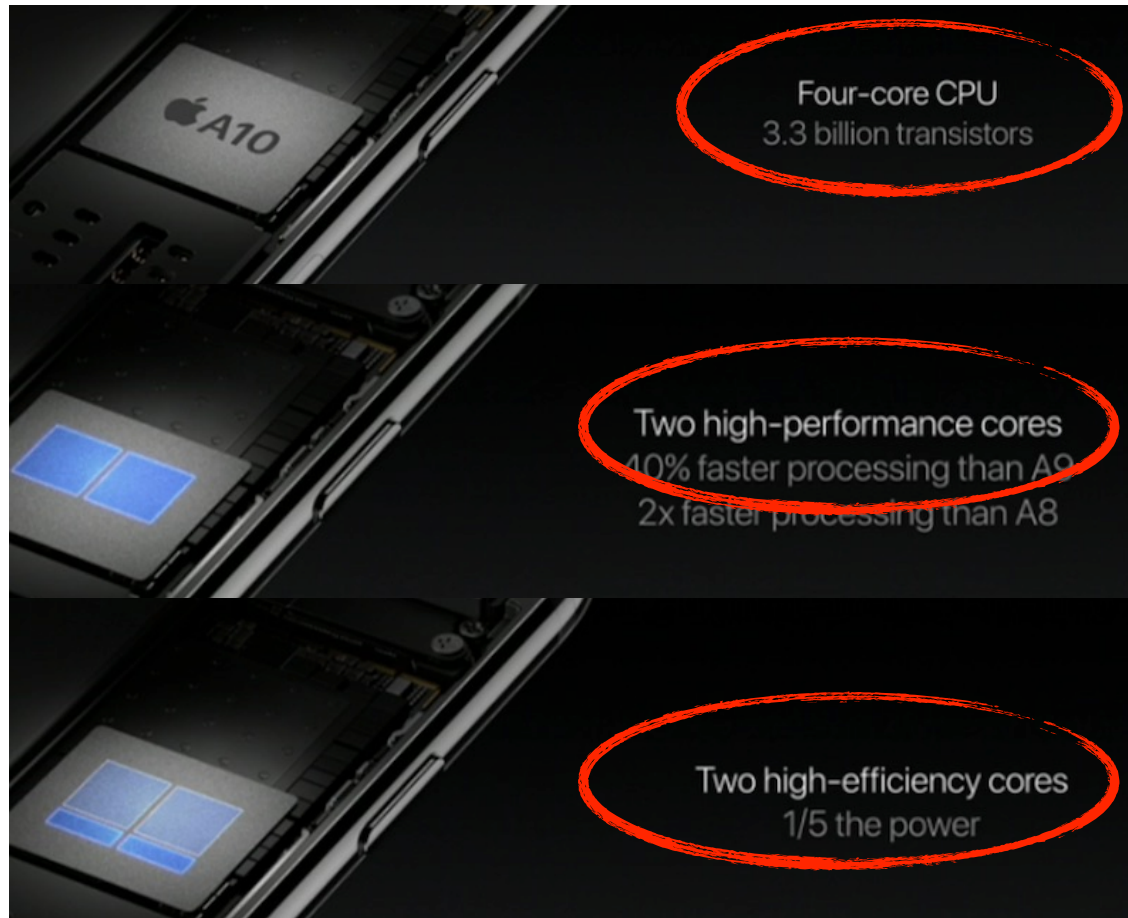
Asymmetric Multiprocessor (AMP)

- Offer a large performance-energy trade-off space



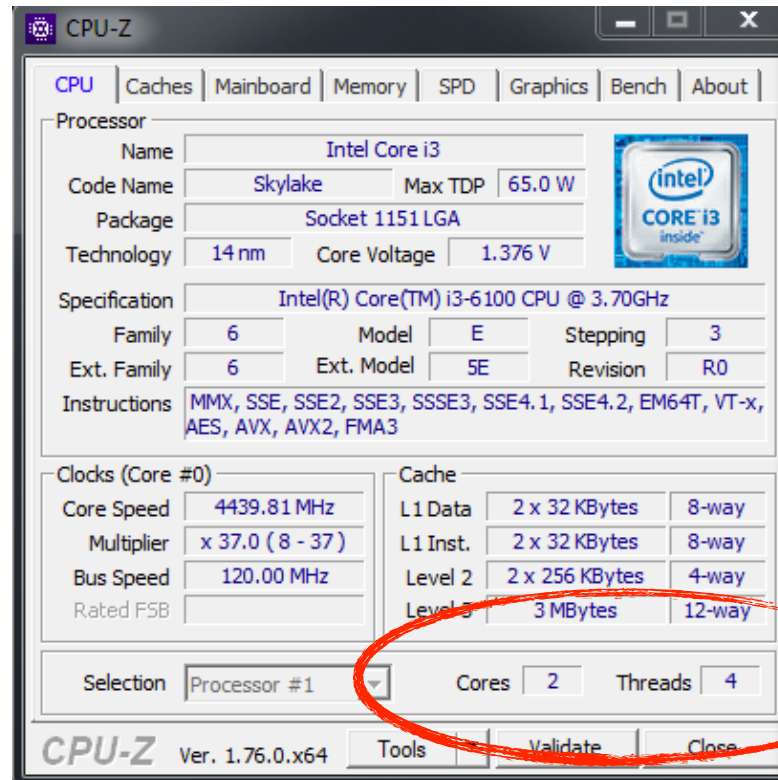
Asymmetric Chip-Multiprocessor (ACMP)

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



Combine Multi-core with Hyper-threading

- 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)



Today

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

The Issue

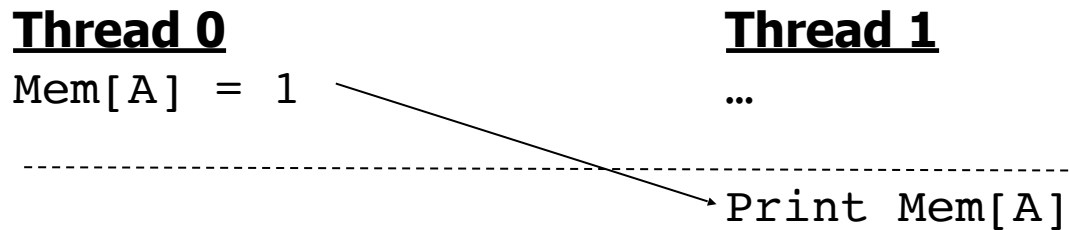
- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.

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.

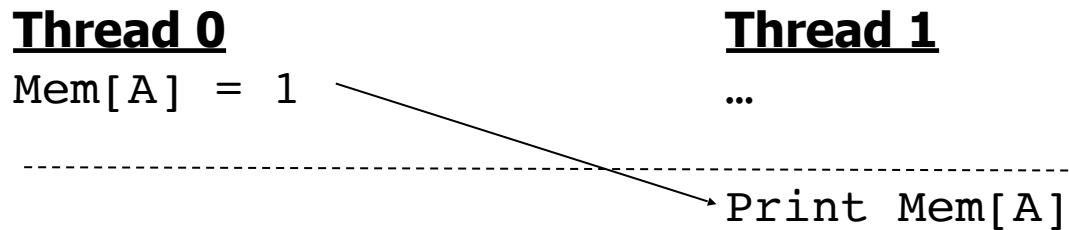
The Issue

- 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.



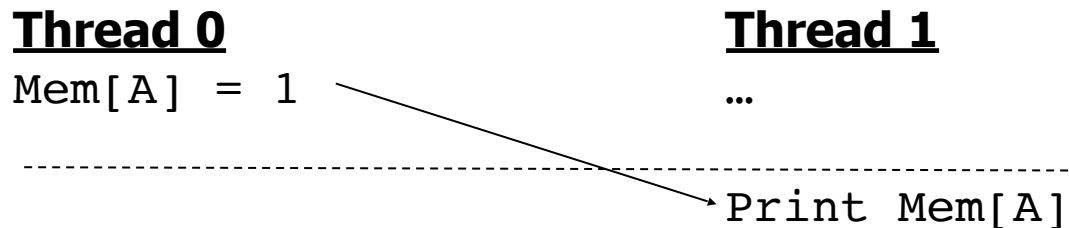
The Issue

- 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



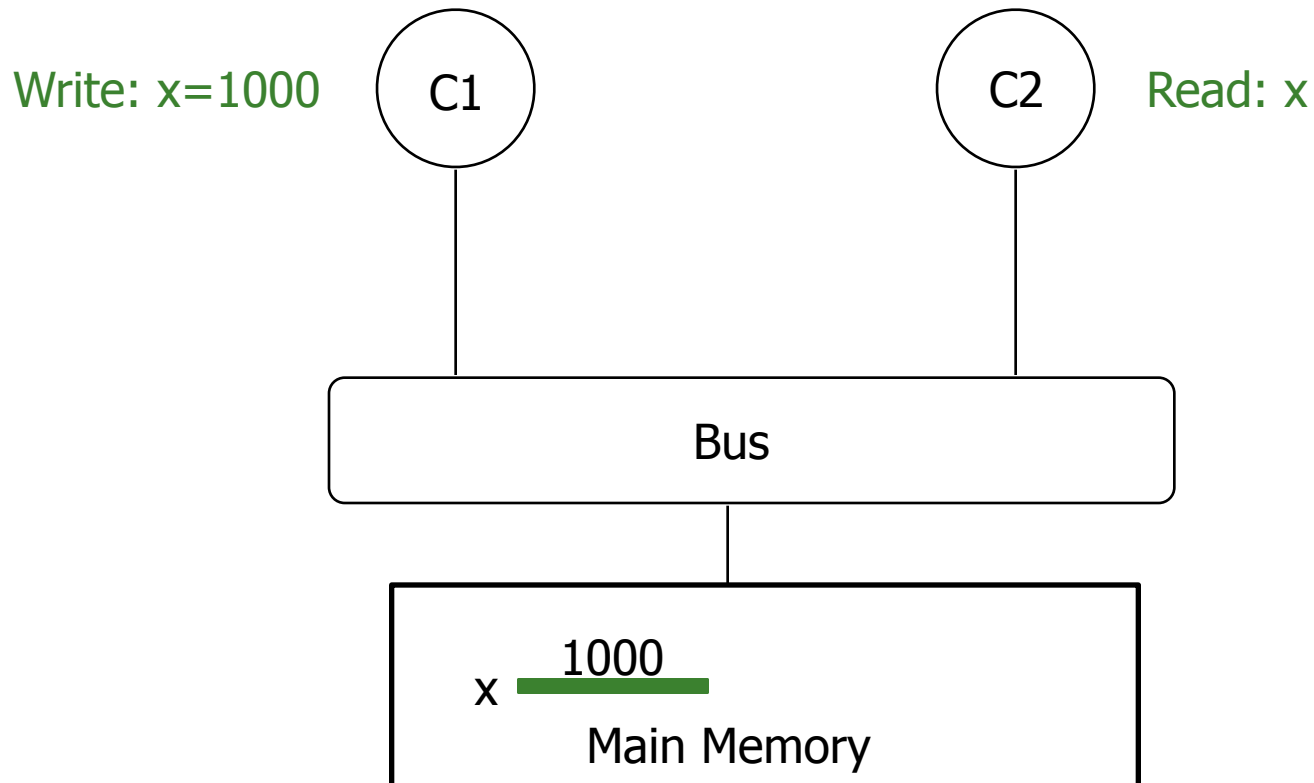
The Issue

- 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?



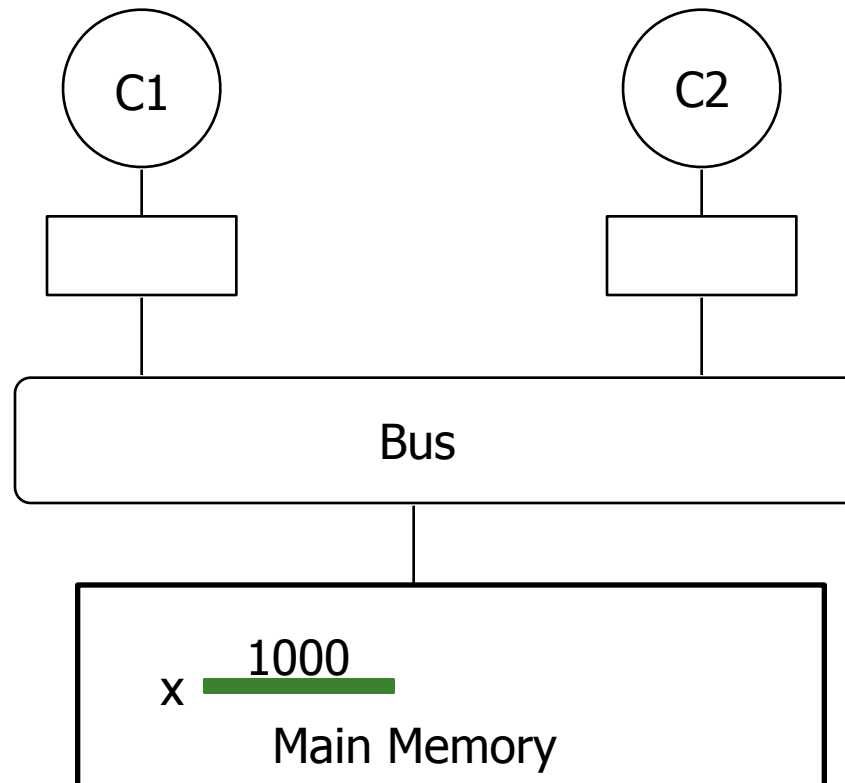
The Issue

- 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.



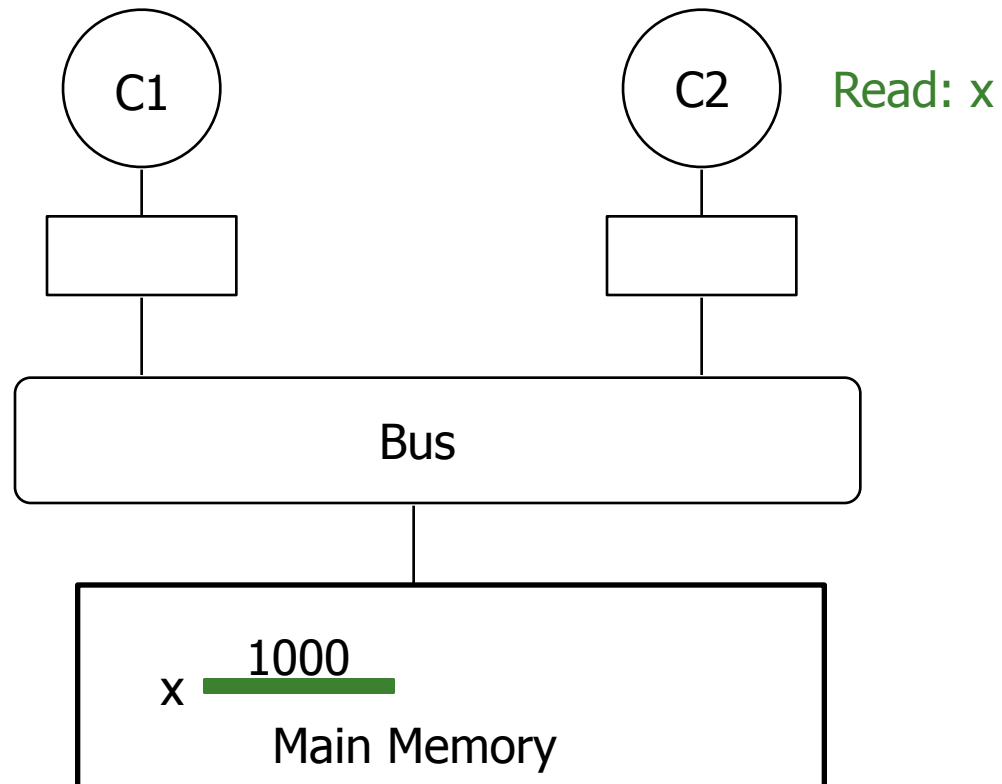
The Issue

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



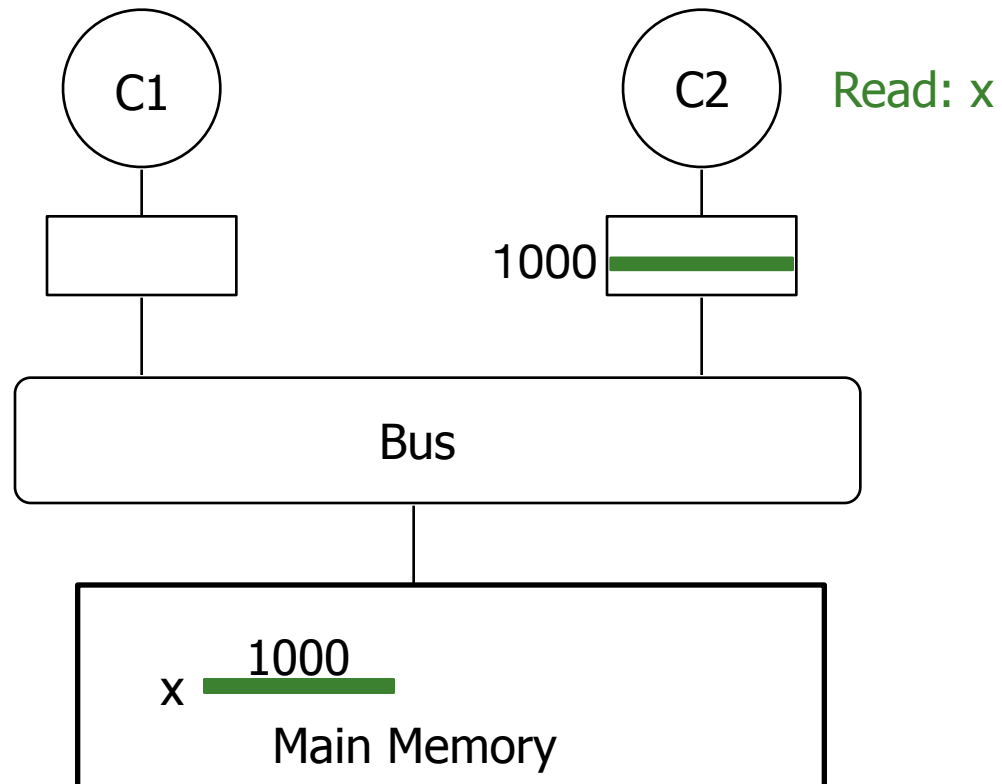
The Issue

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



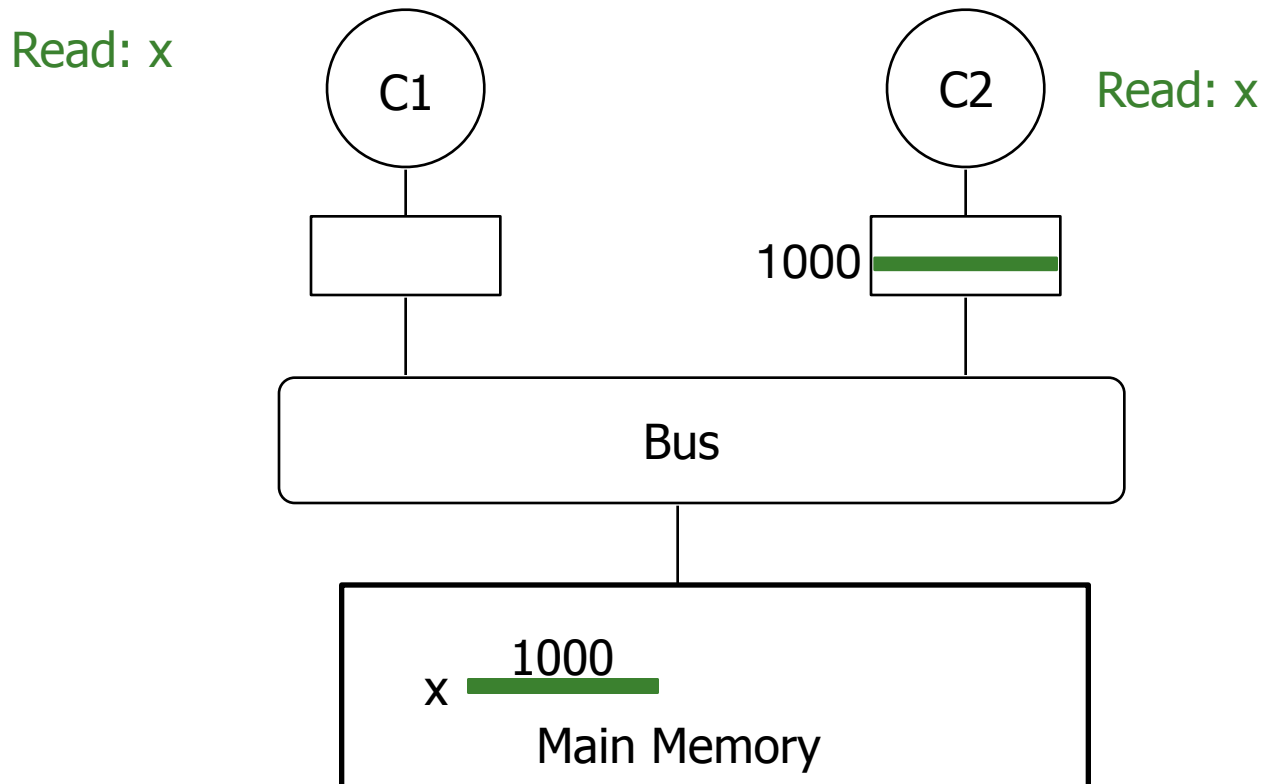
The Issue

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



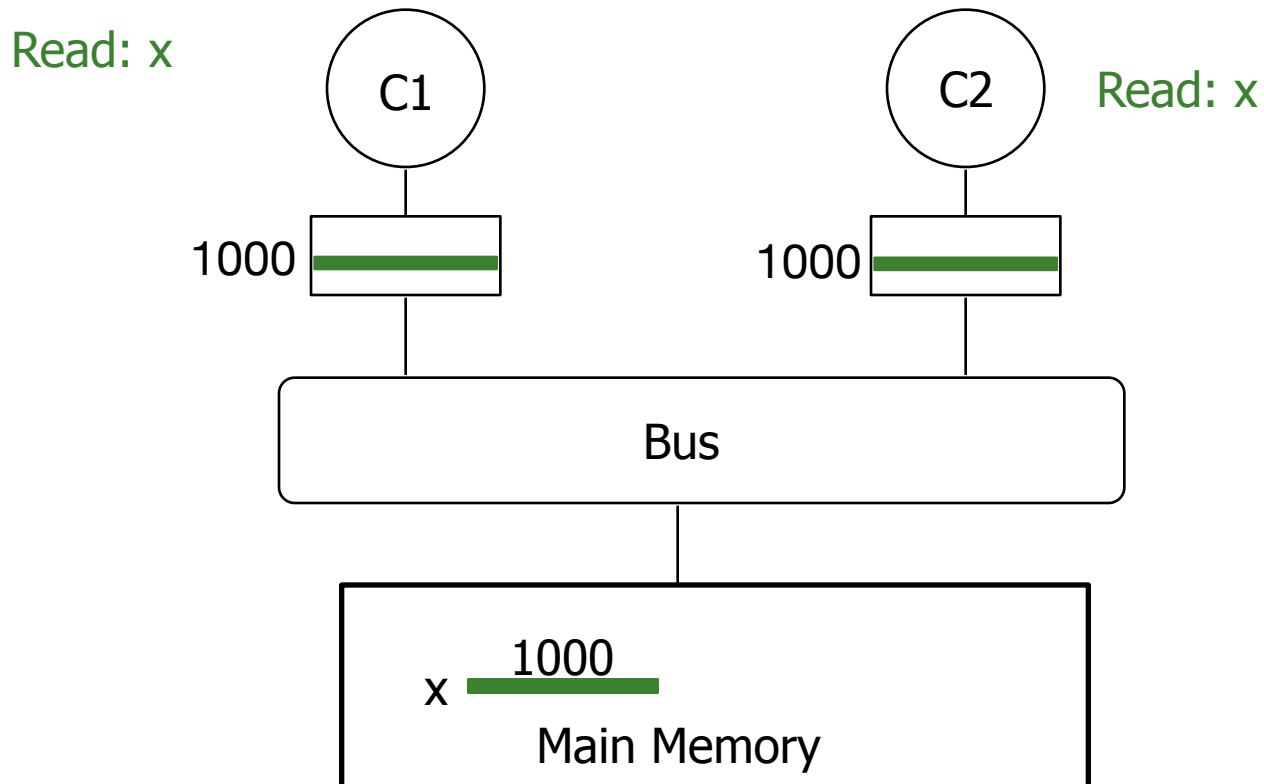
The Issue

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



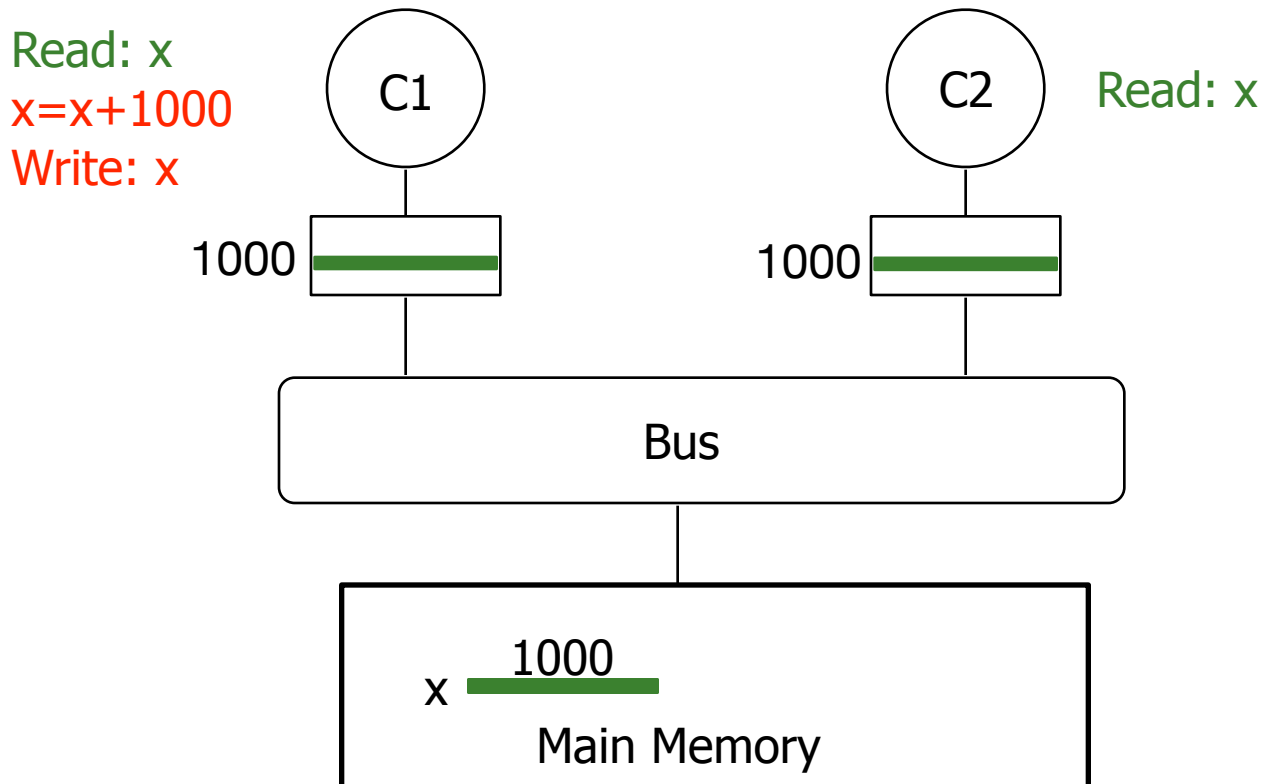
The Issue

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



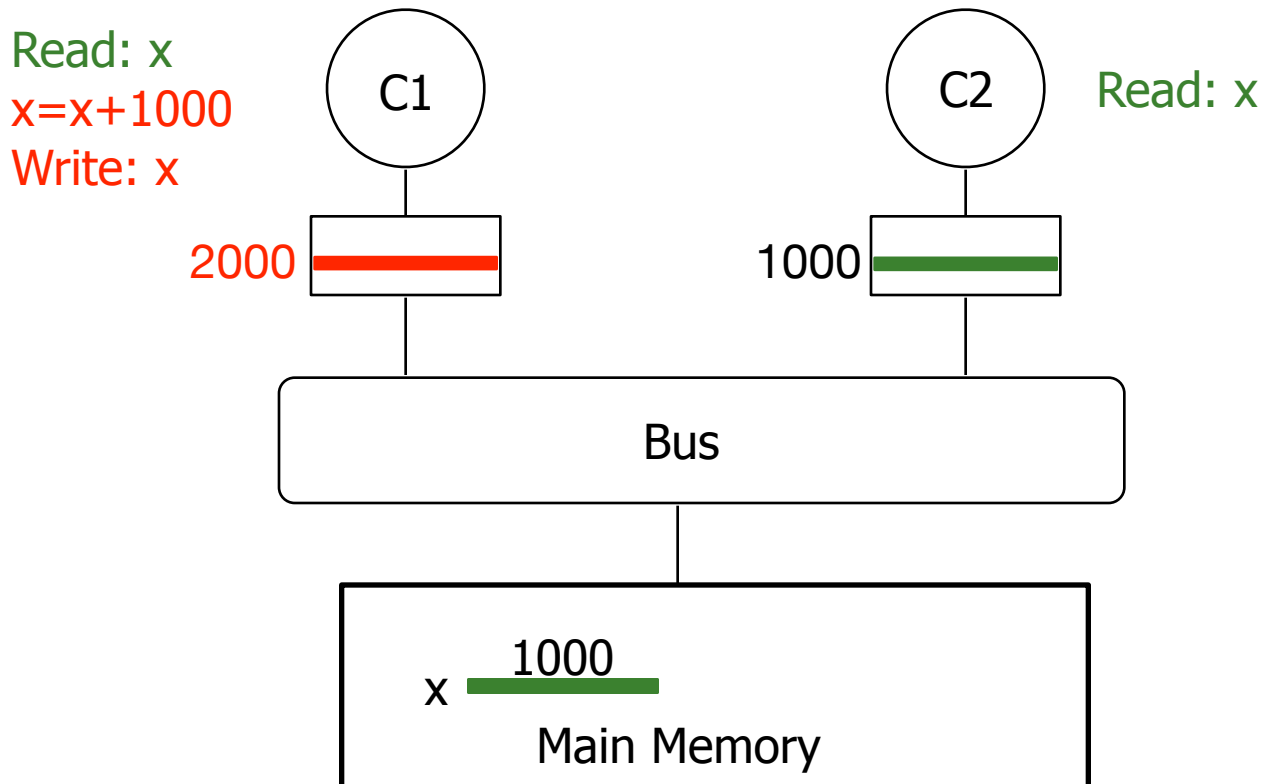
The Issue

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



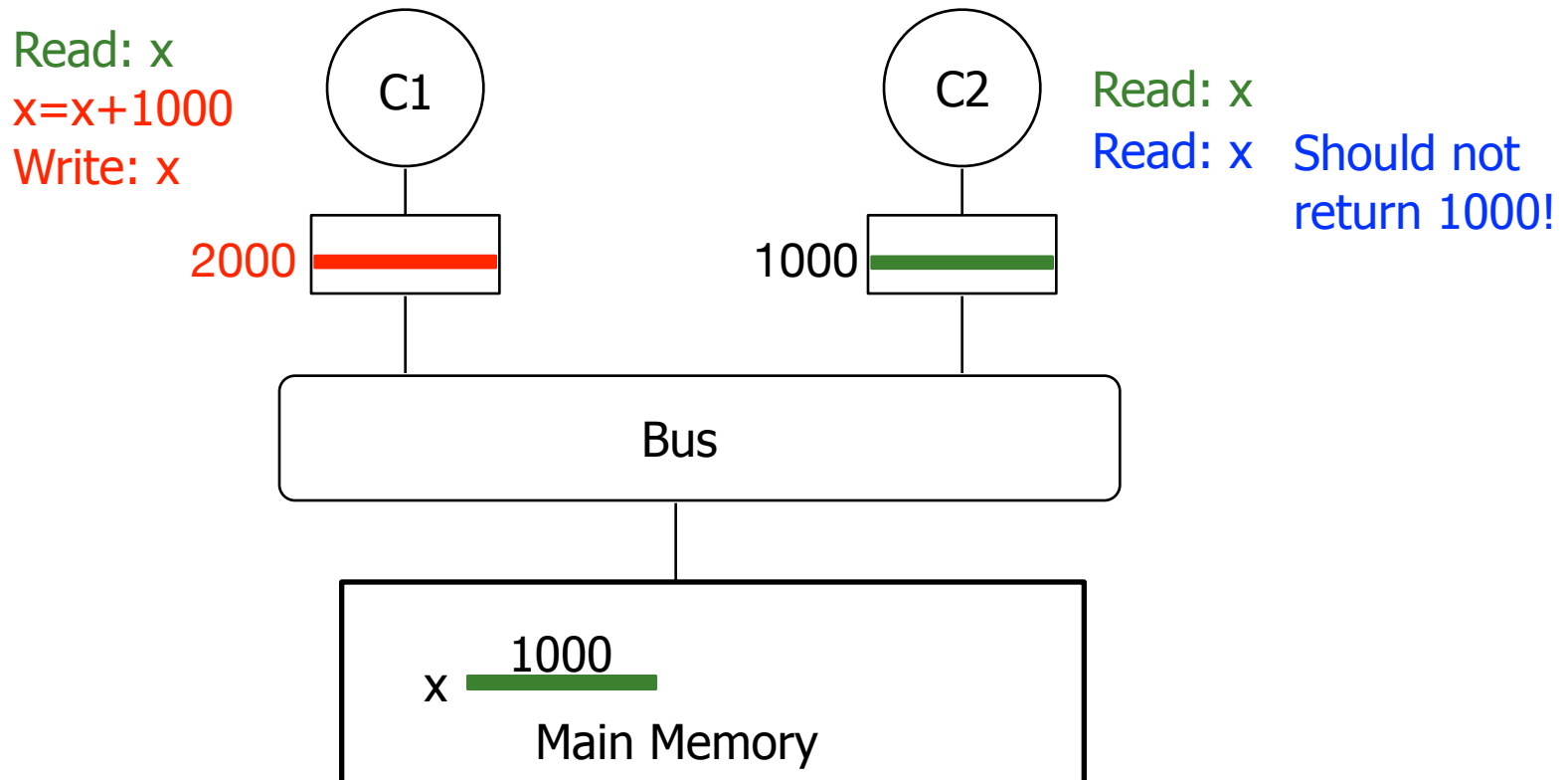
The Issue

- 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

- **Key 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

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

Cache Coherence: The Idea

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

Cache Coherence: The Idea

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

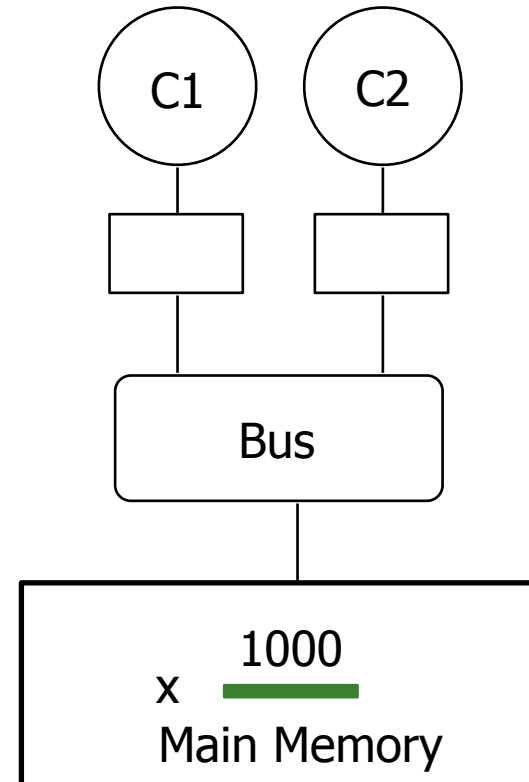
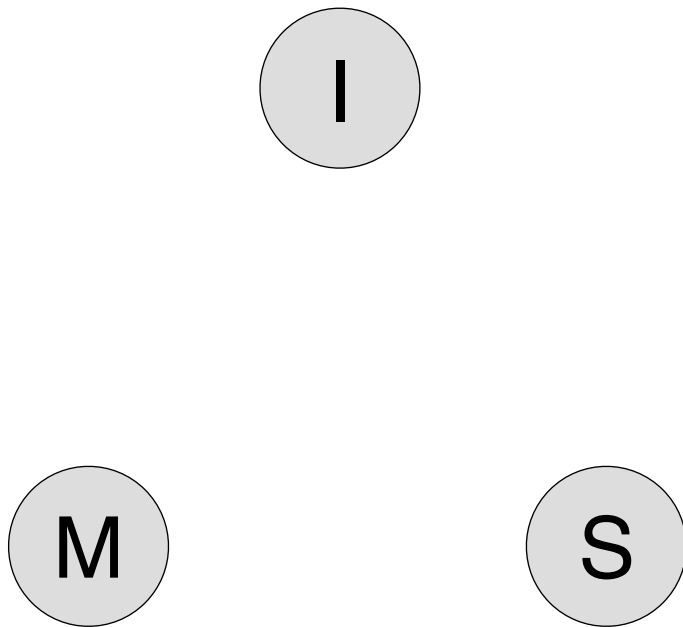
Cache Coherence: The Idea

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

Invalidate-Based Cache Coherence

Associate each cache line with 3
states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action

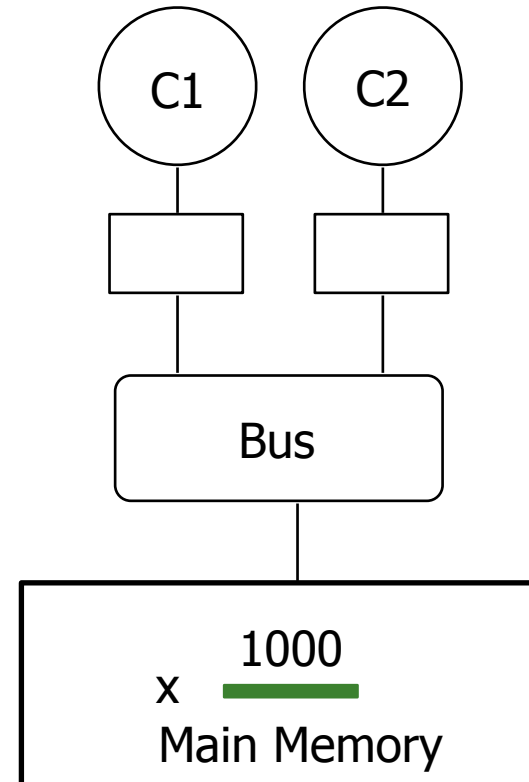
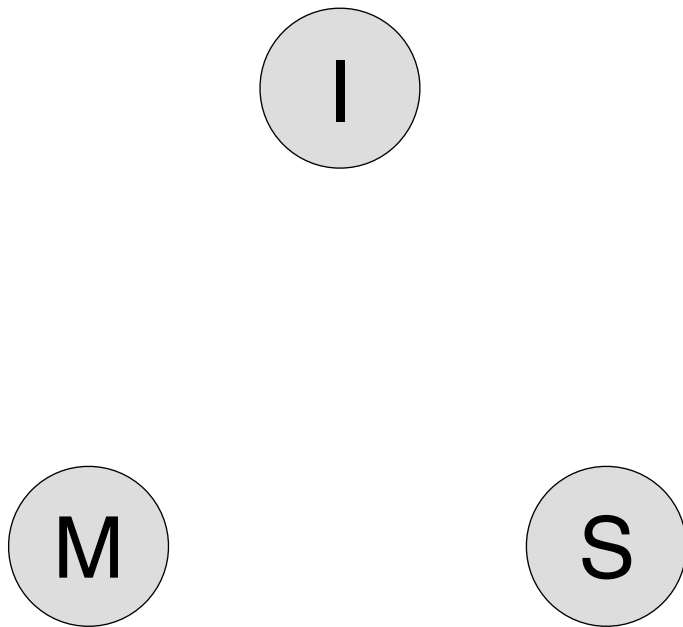


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Read: x

Below: State Transition for x in C2's cache;
Syntax: Event/Action

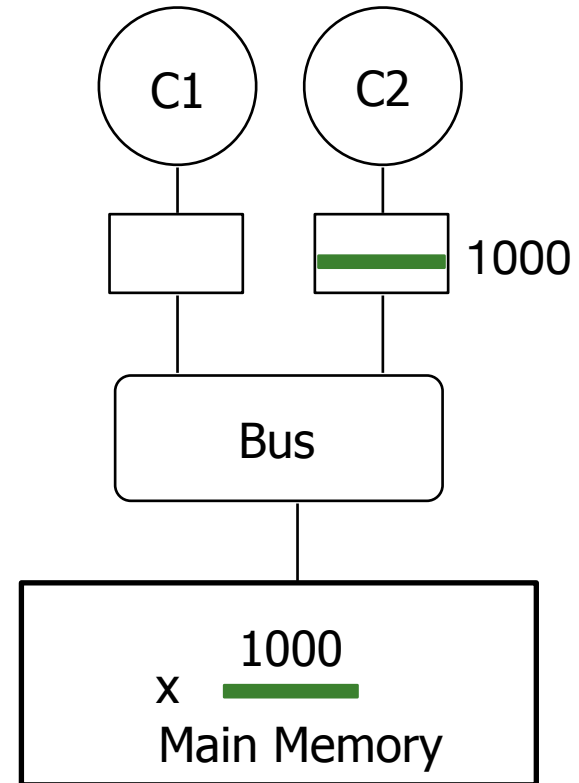
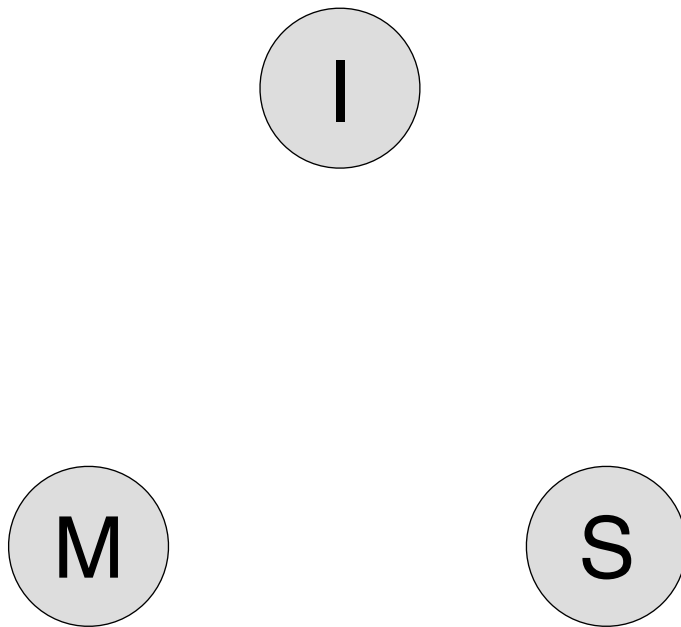


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Read: x

Below: State Transition for x in C2's cache;
Syntax: Event/Action

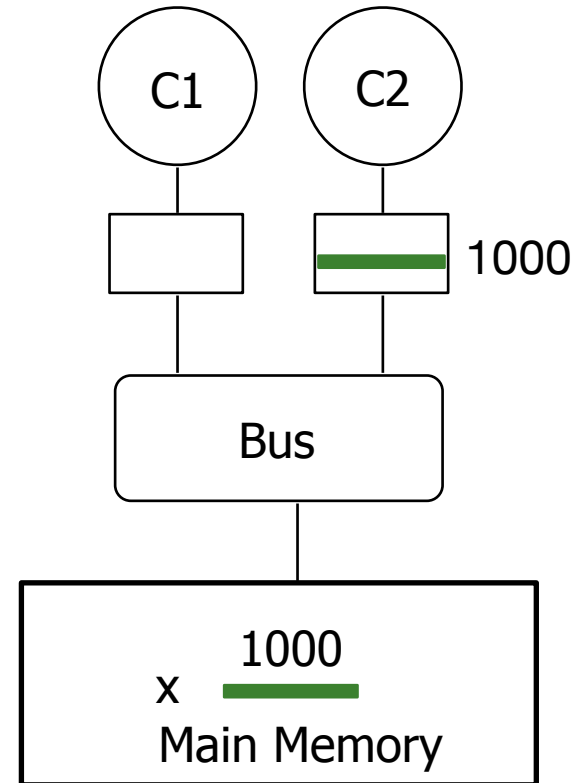
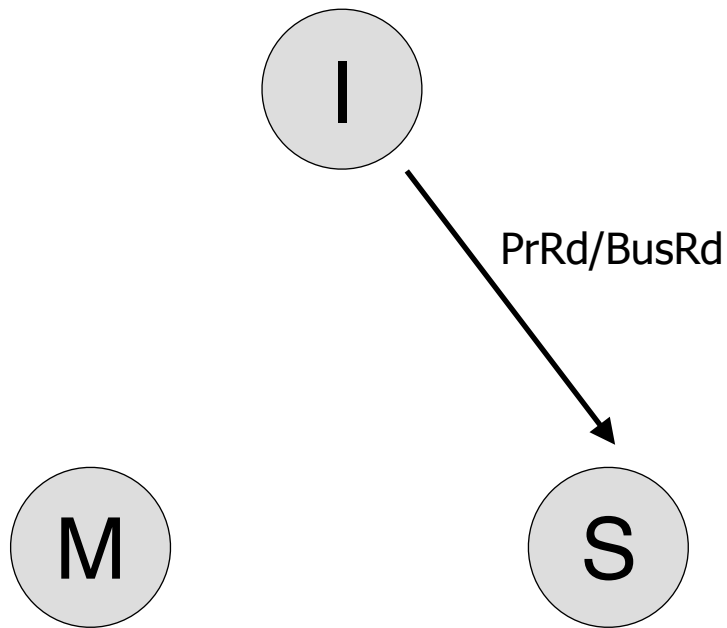


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action

Read: x

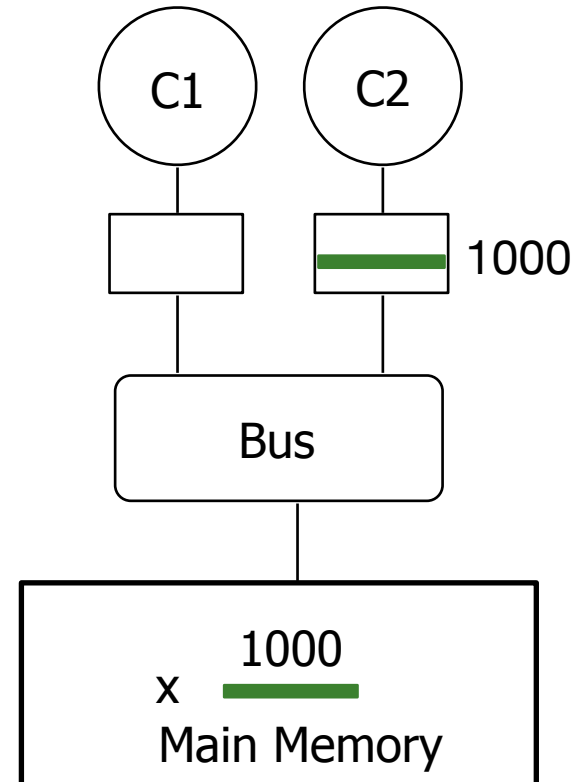
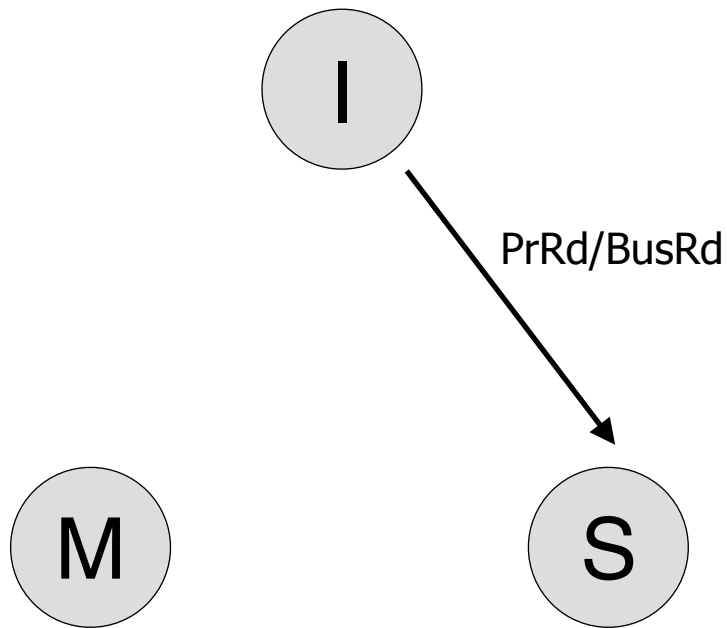


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action

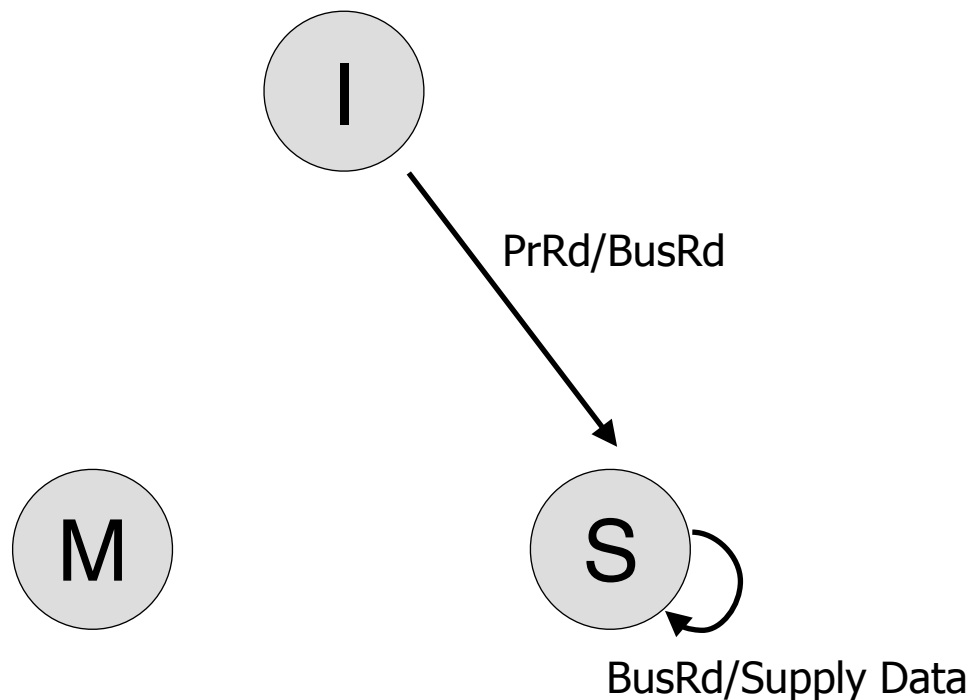
Read: x Read: x



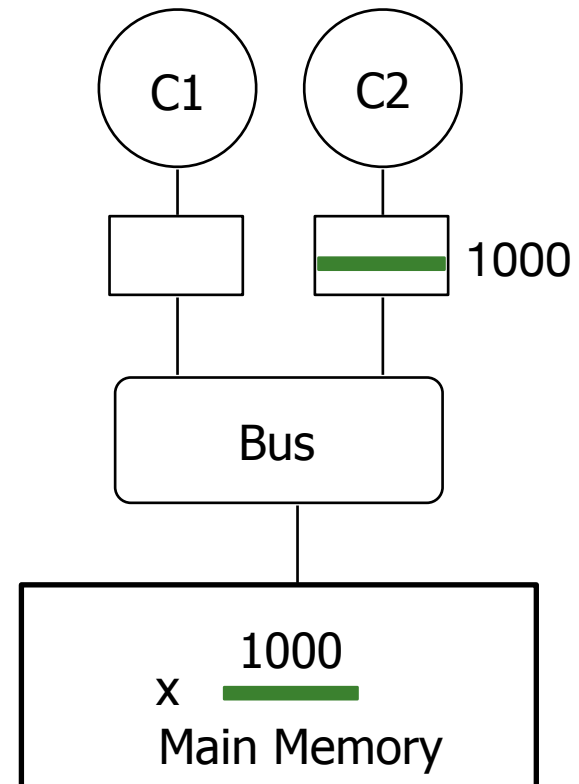
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



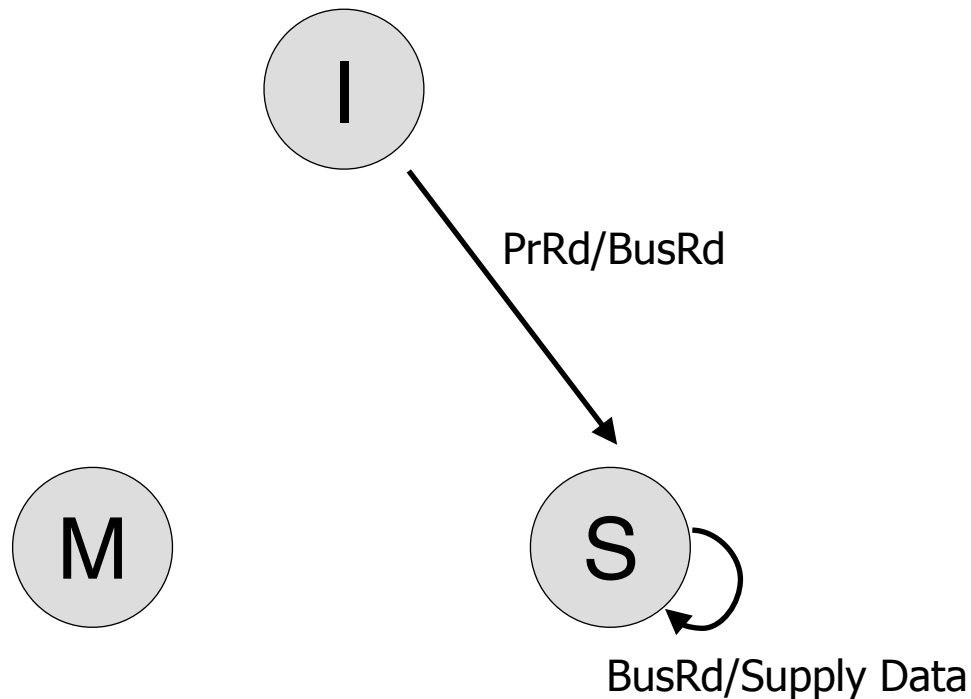
Read: x Read: x



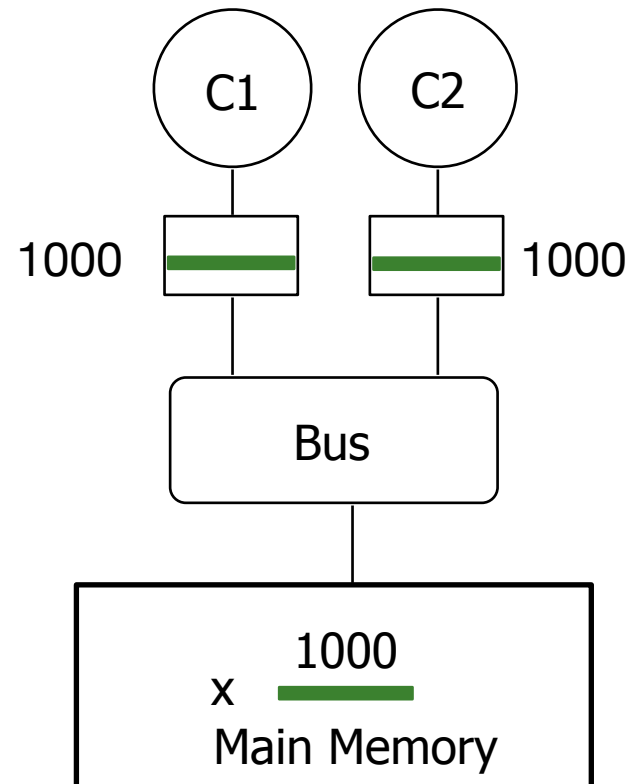
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



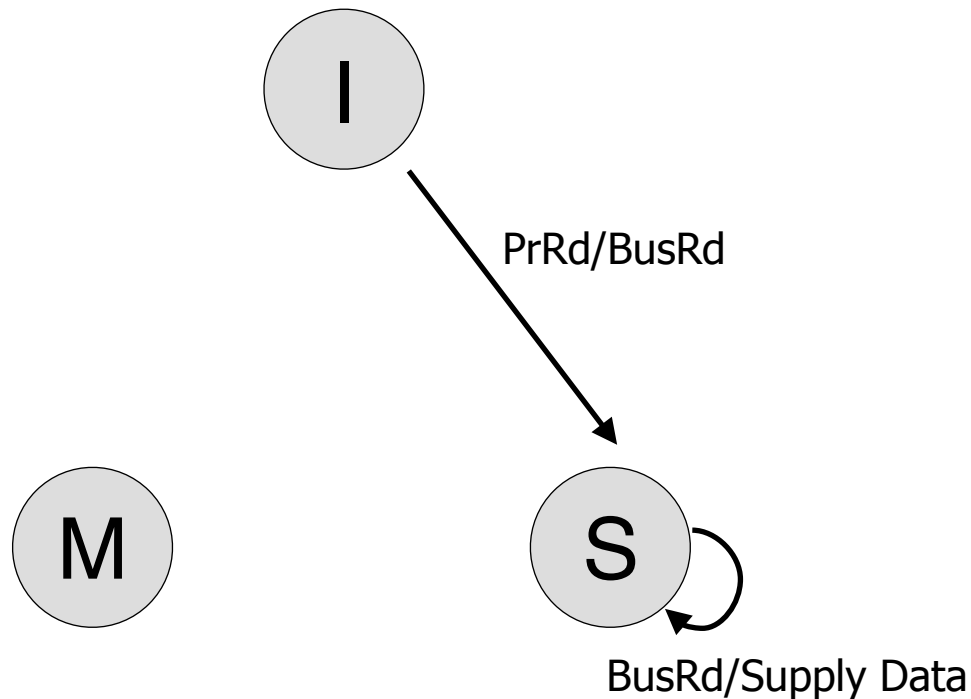
Read: x Read: x



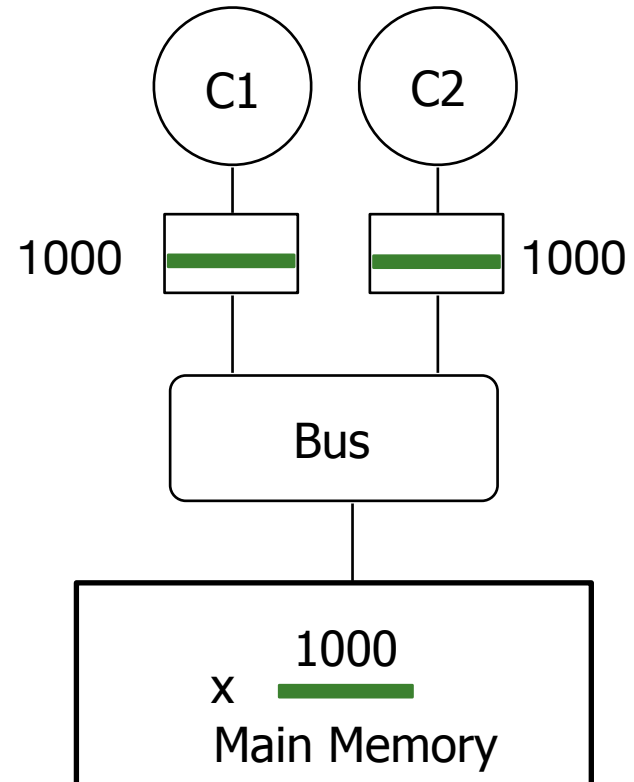
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



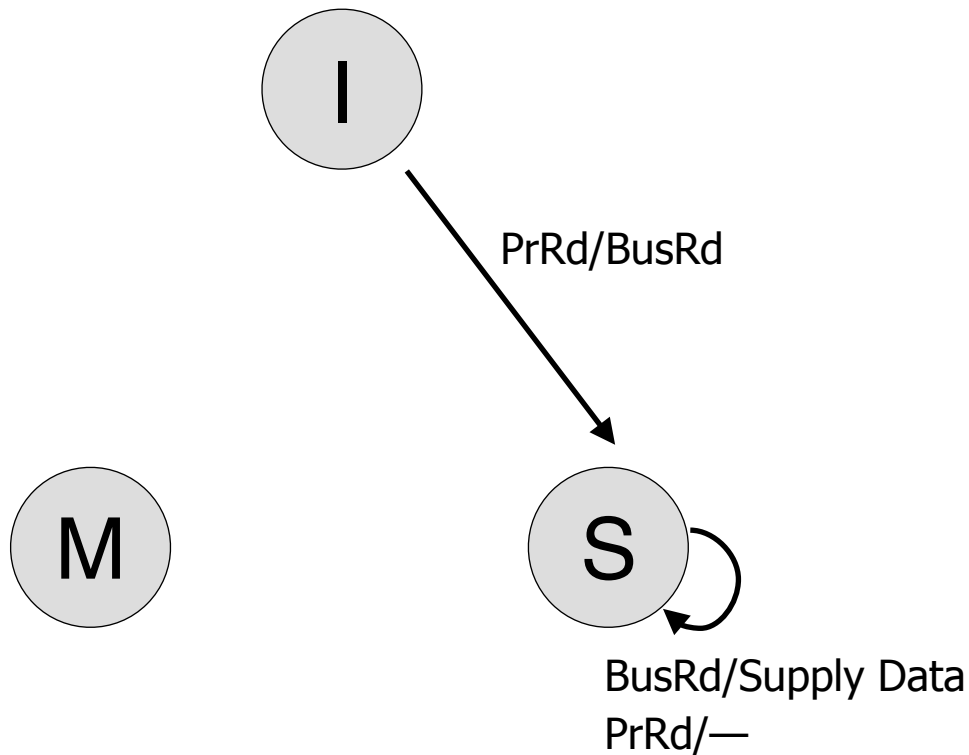
Read: x Read: x
 Read: x



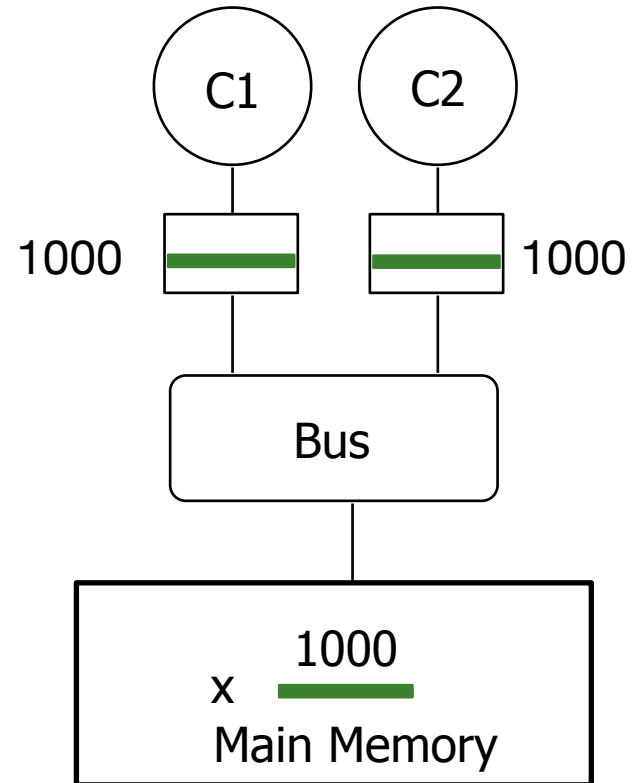
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



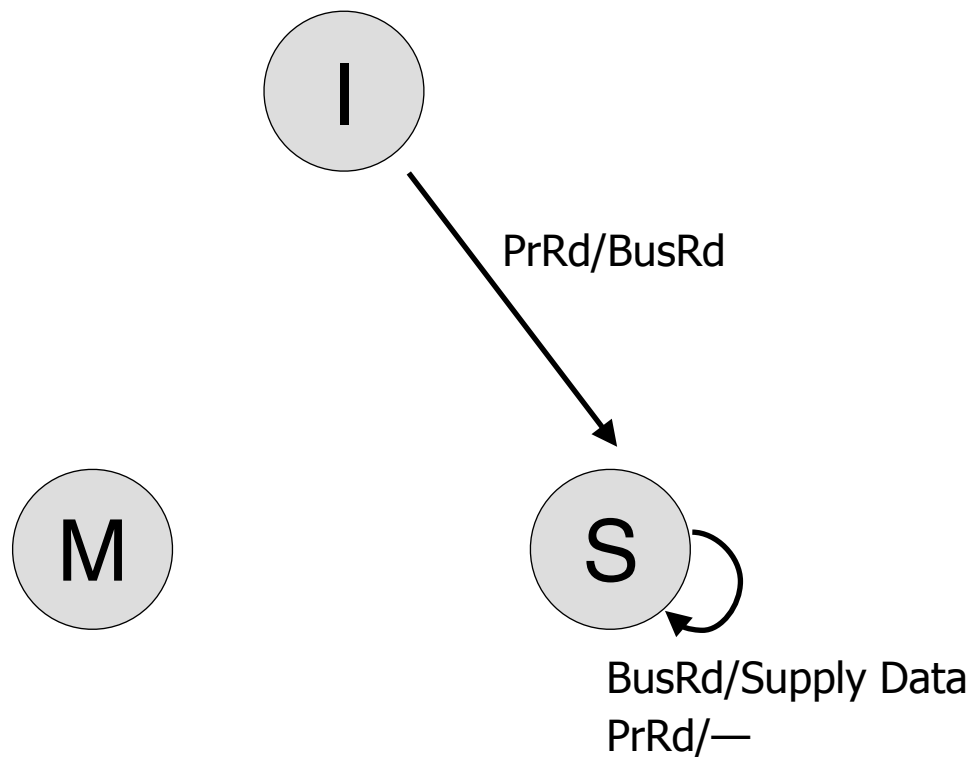
Read: x Read: x
 Read: x



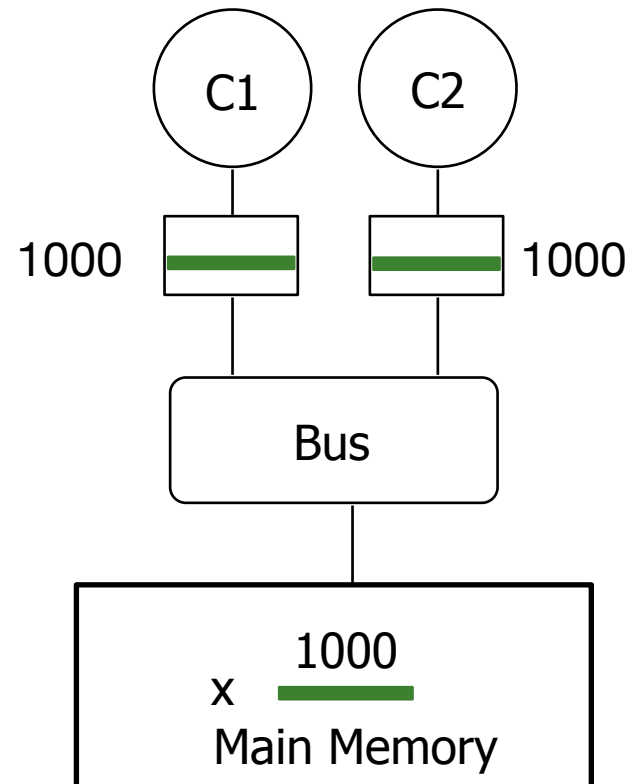
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



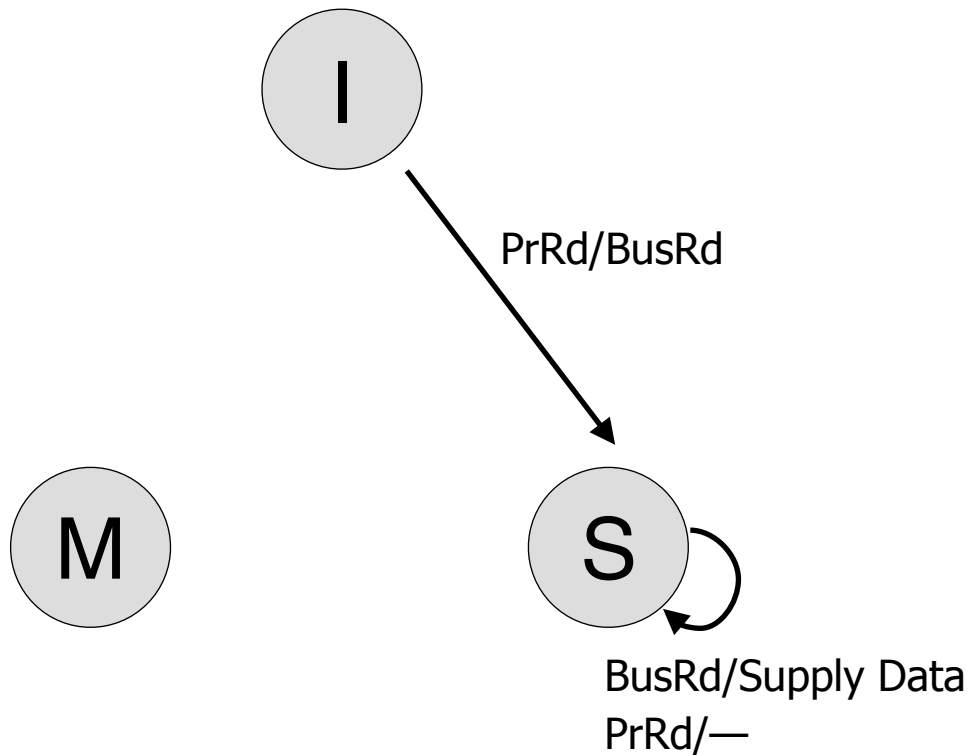
Read: x Read: x
 Read: x
 Write: x = 5000



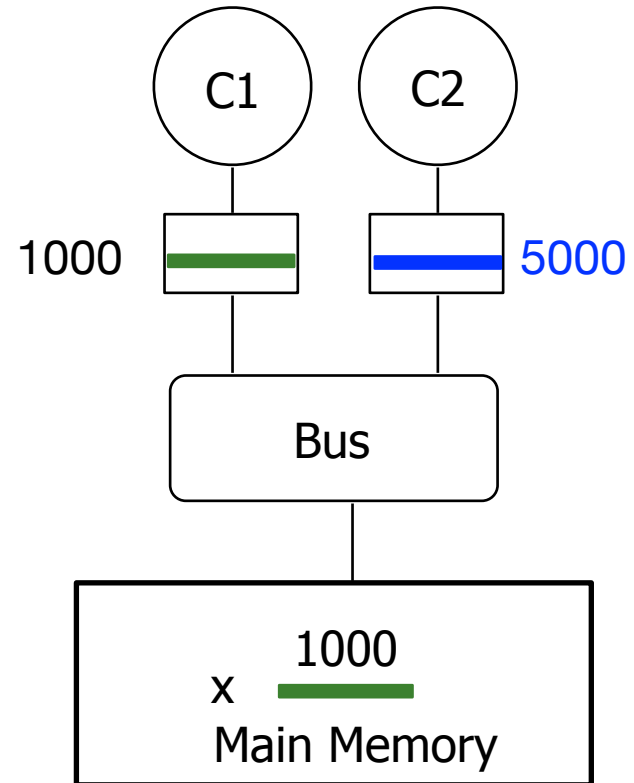
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



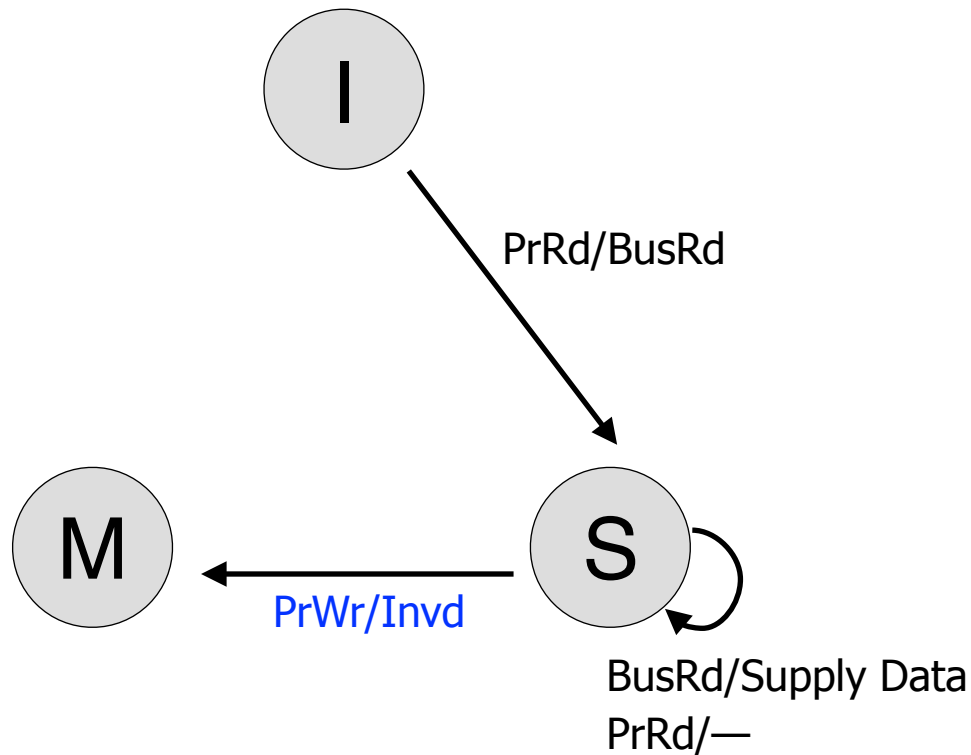
Read: x Read: x
Read: x
Write: $x = 5000$



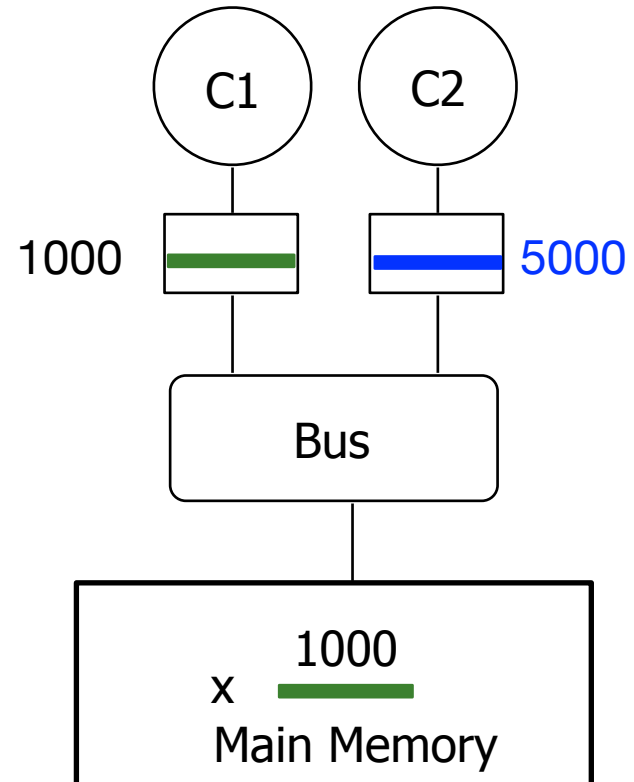
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



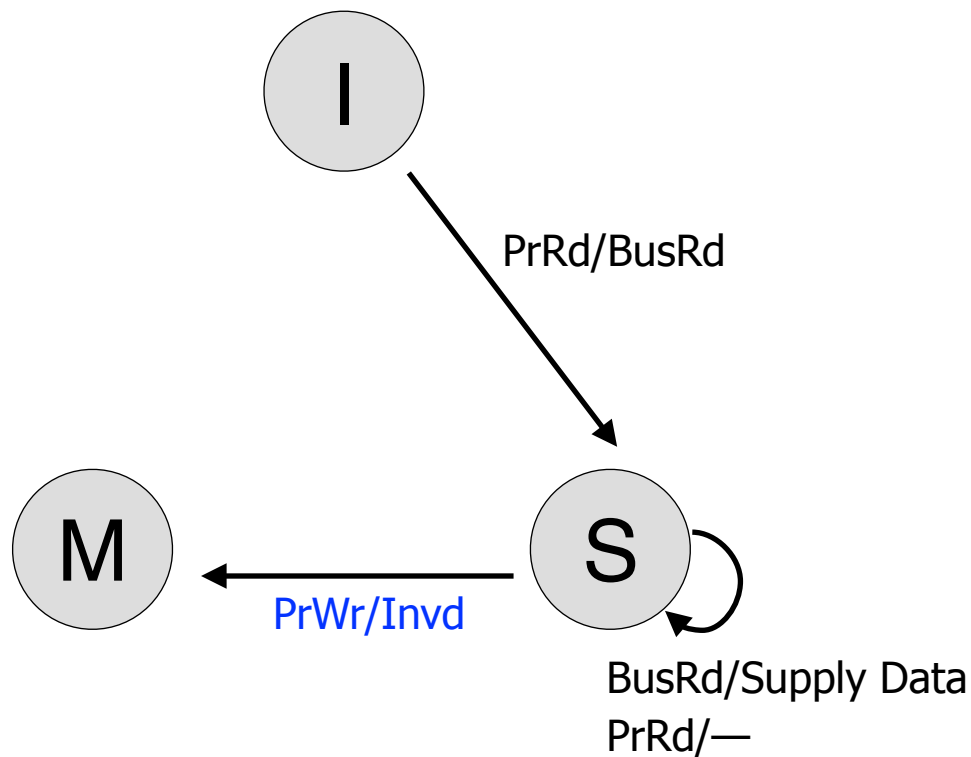
Read: x Read: x
Read: x
Write: $x = 5000$



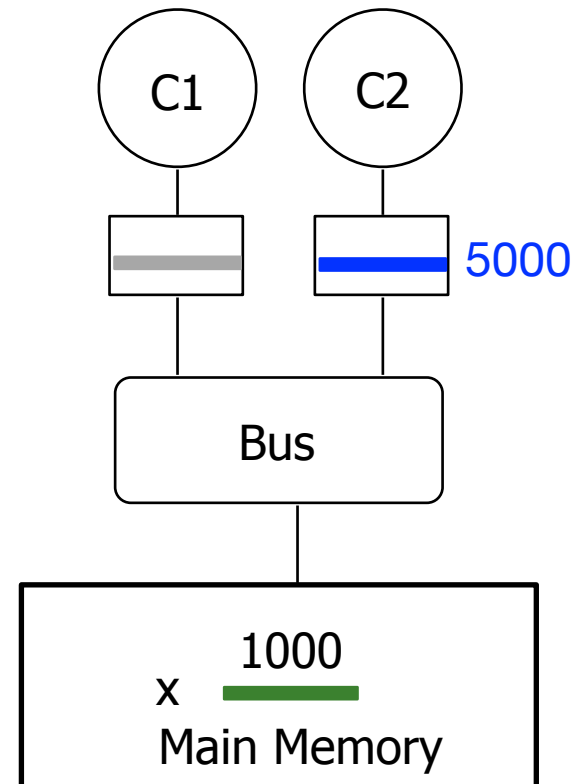
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



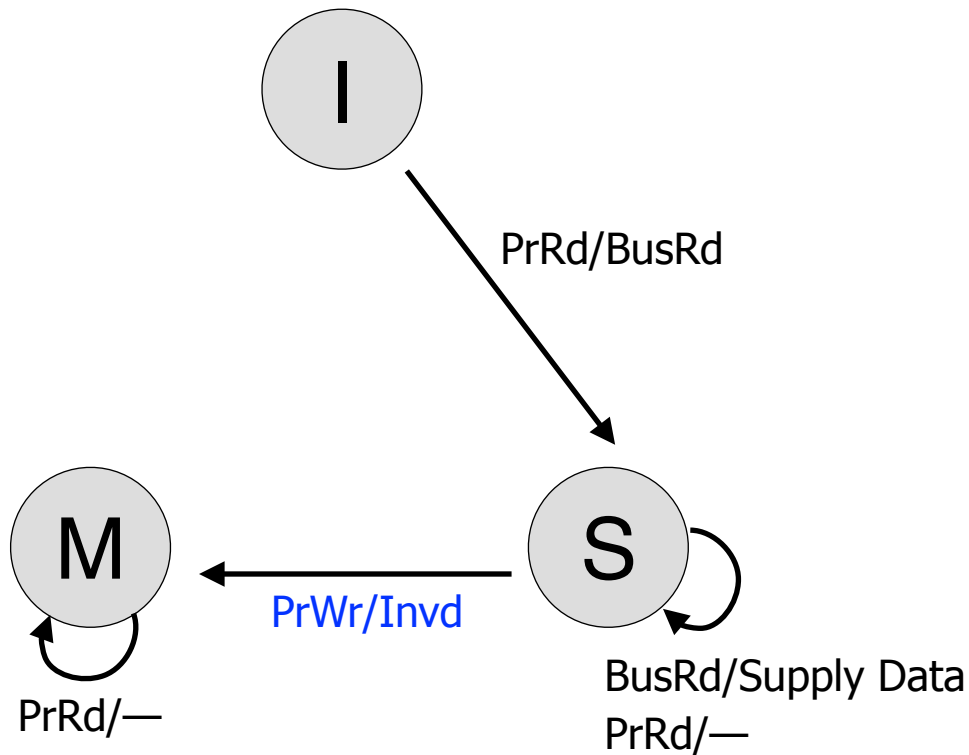
Read: x Read: x
Read: x
Write: $x = 5000$



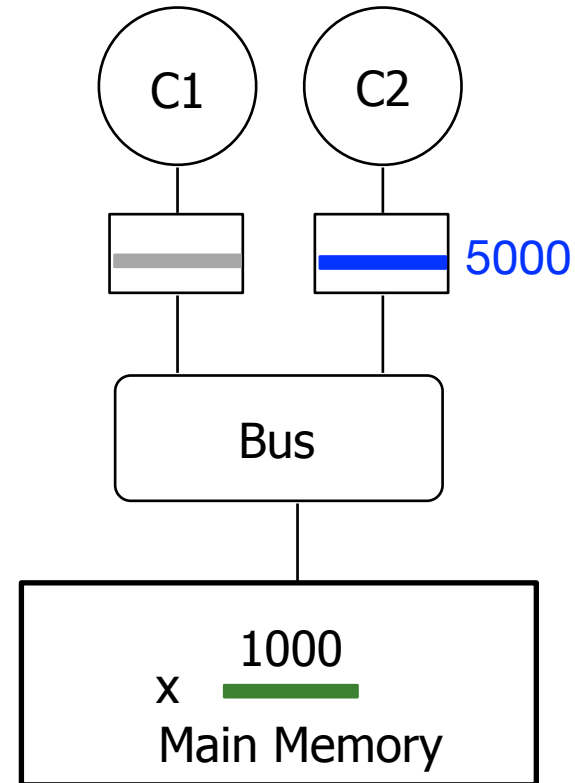
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



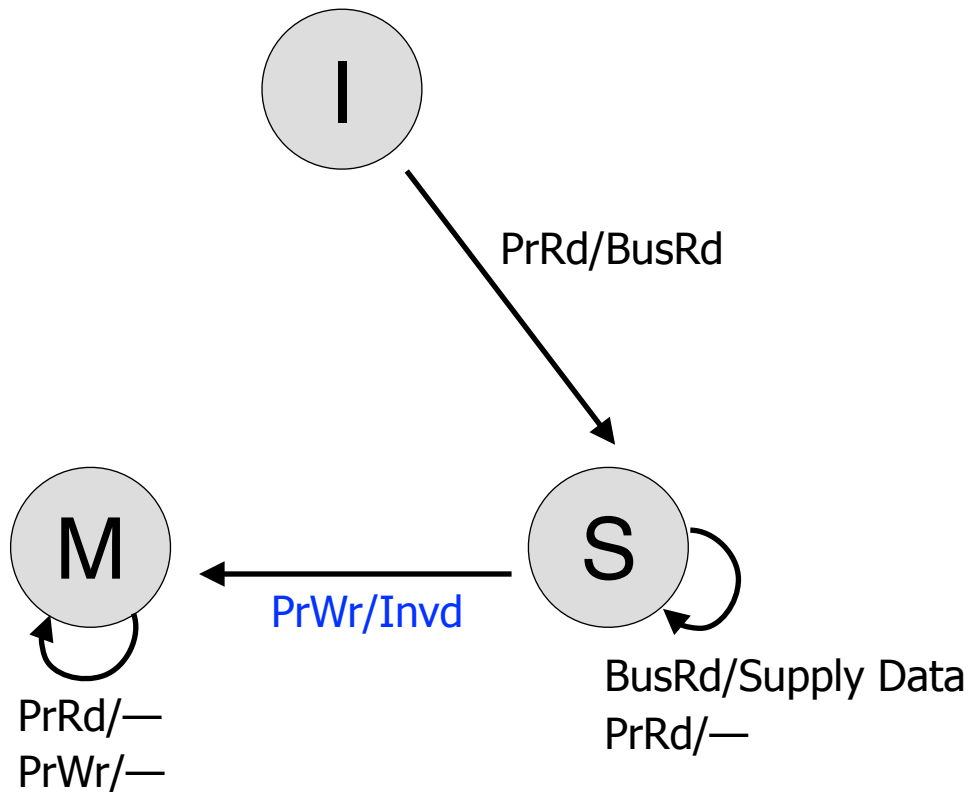
Read: x Read: x
Read: x
Write: $x = 5000$



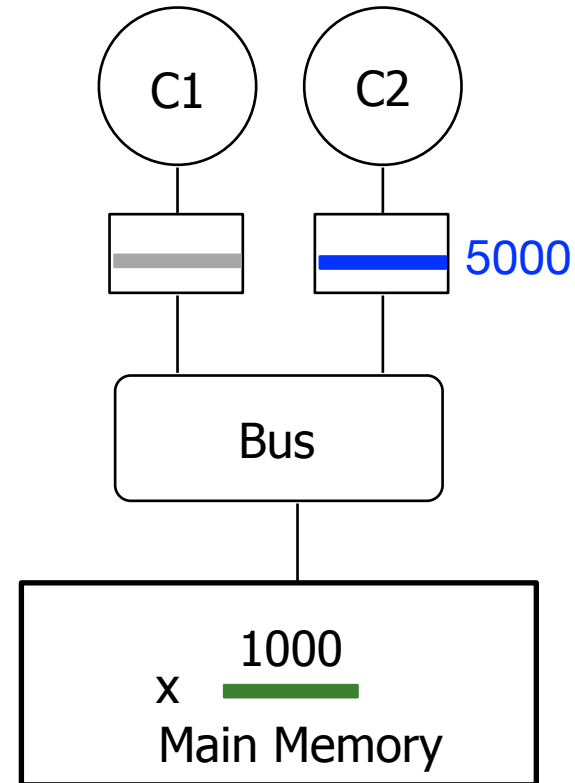
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



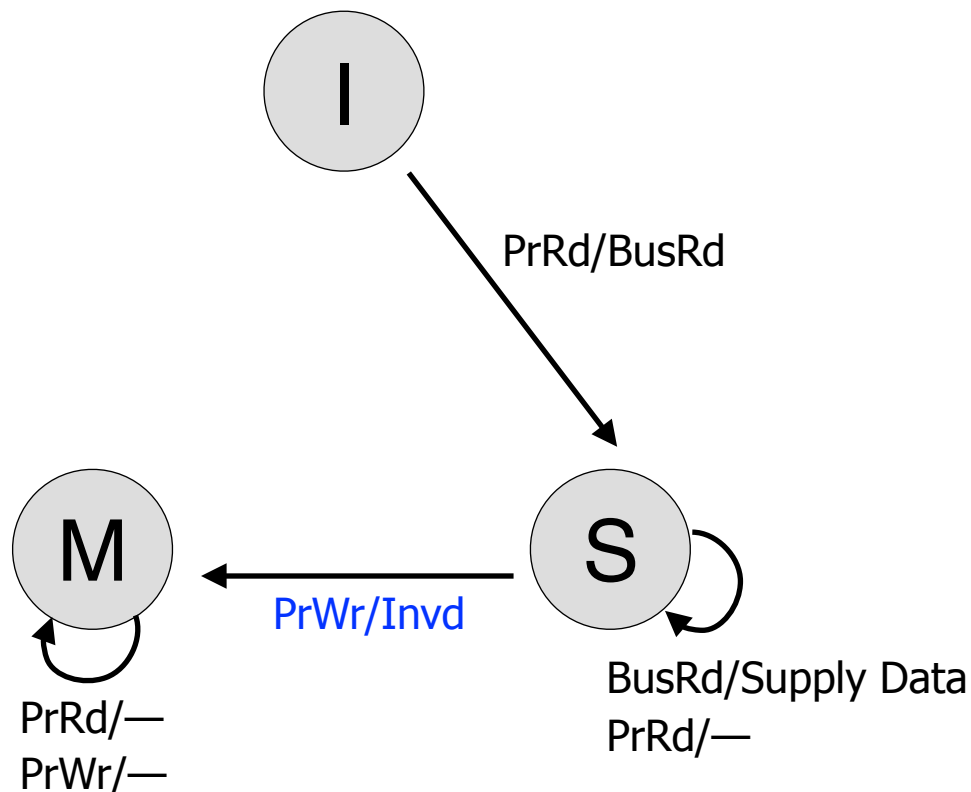
Read: x Read: x
Read: x
Write: $x = 5000$



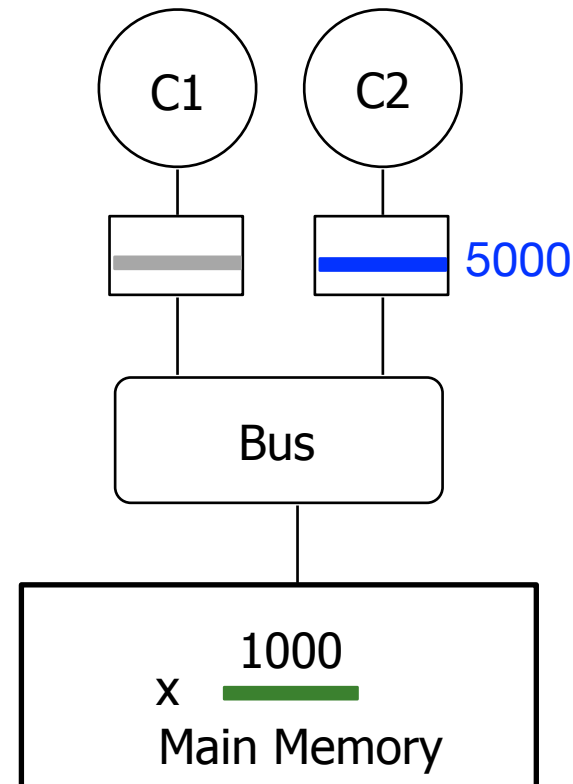
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



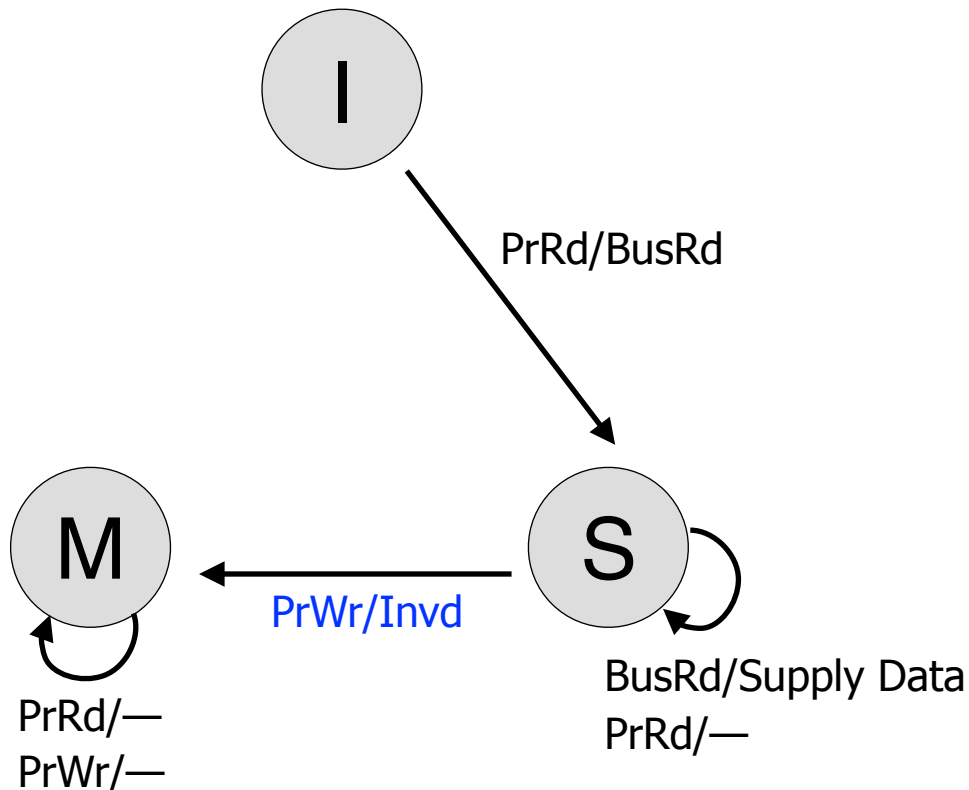
Read: x Read: x
Read: x Read: x
Write: $x = 5000$



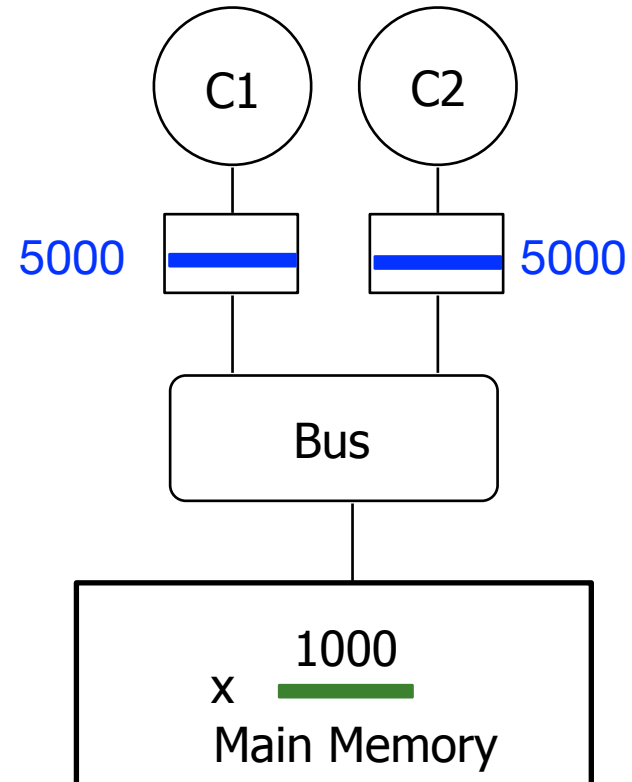
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



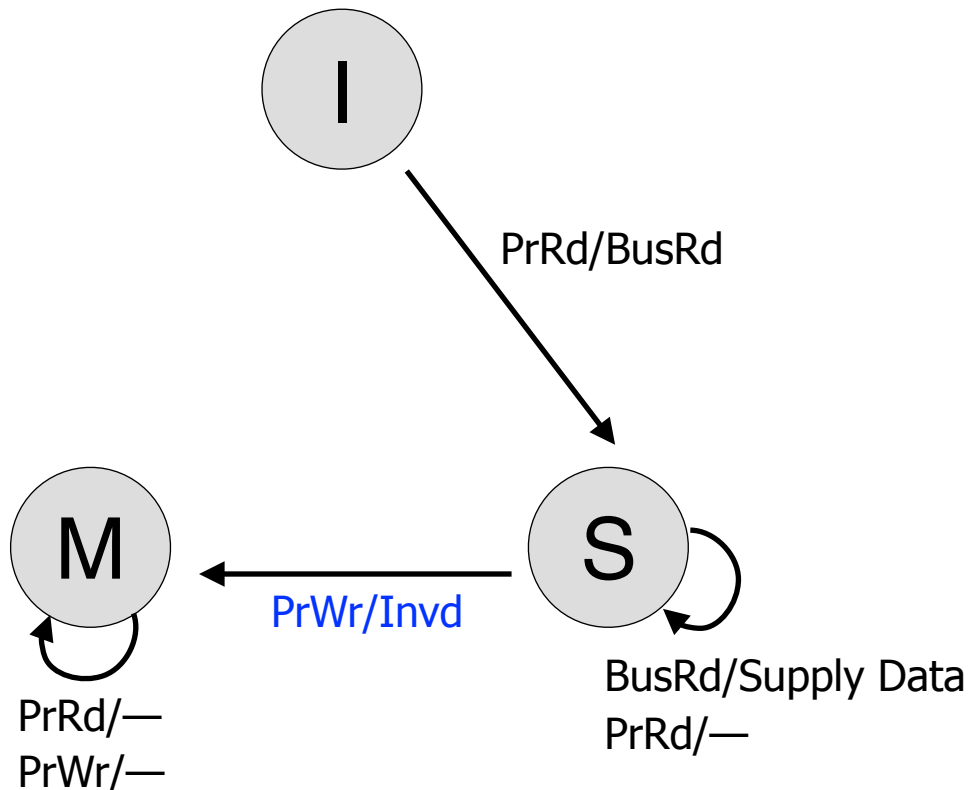
Read: x Read: x
 Read: x Read: x
 Write: $x = 5000$



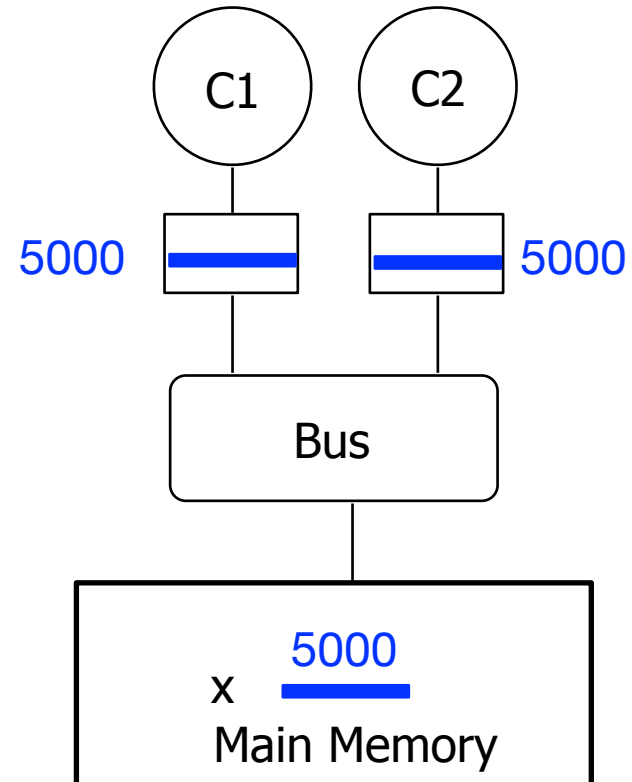
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



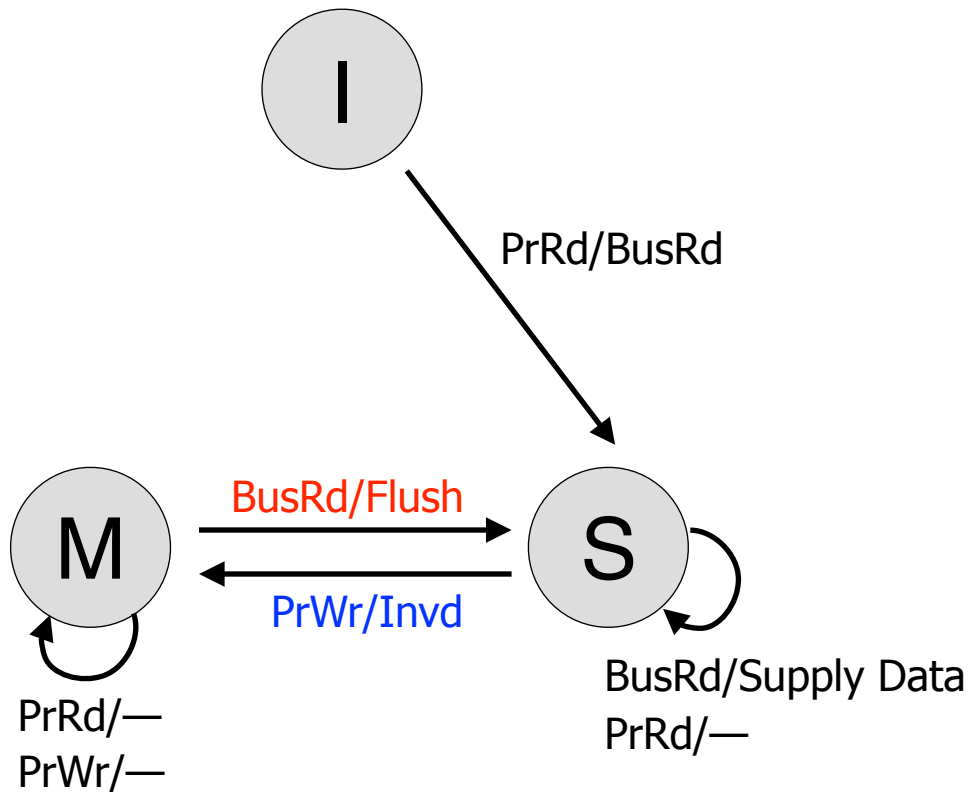
Read: x Read: x
 Read: x Read: x
 Write: $x = 5000$



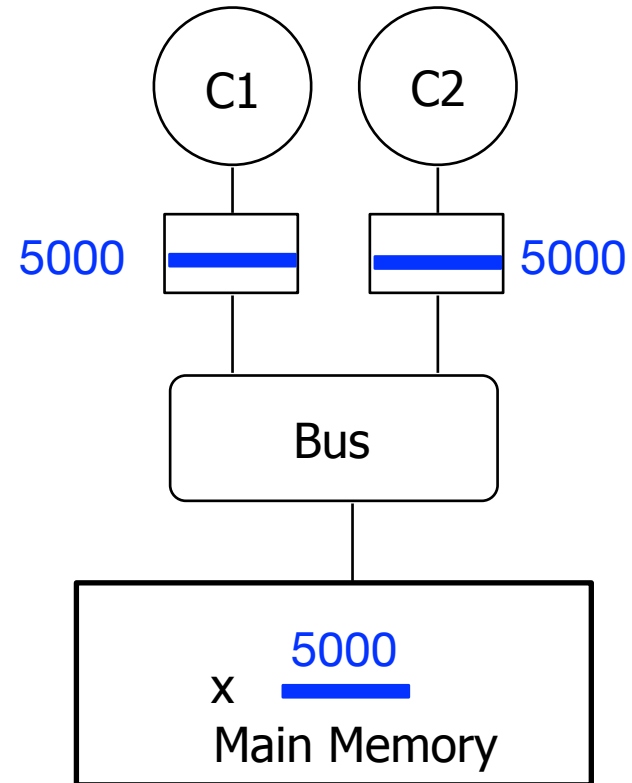
Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Below: State Transition for x in C2's cache;
Syntax: Event/Action



Read: x Read: x
 Read: x Read: x
 Write: $x = 5000$

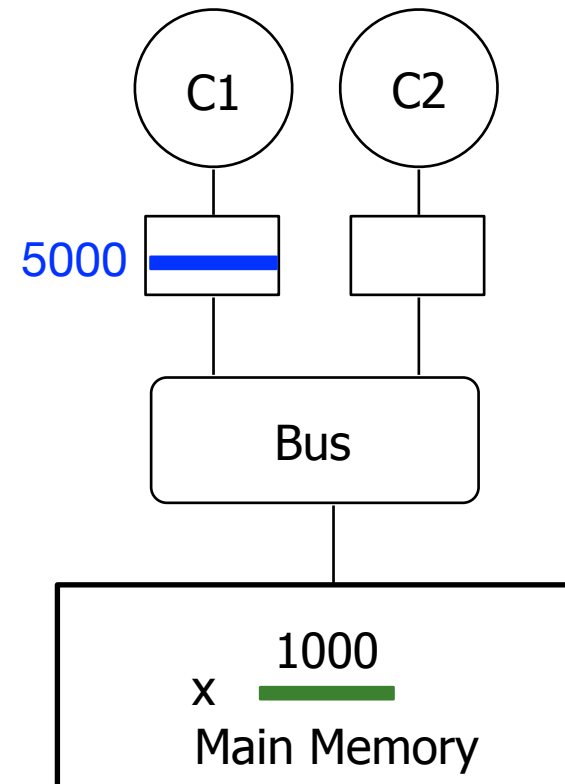
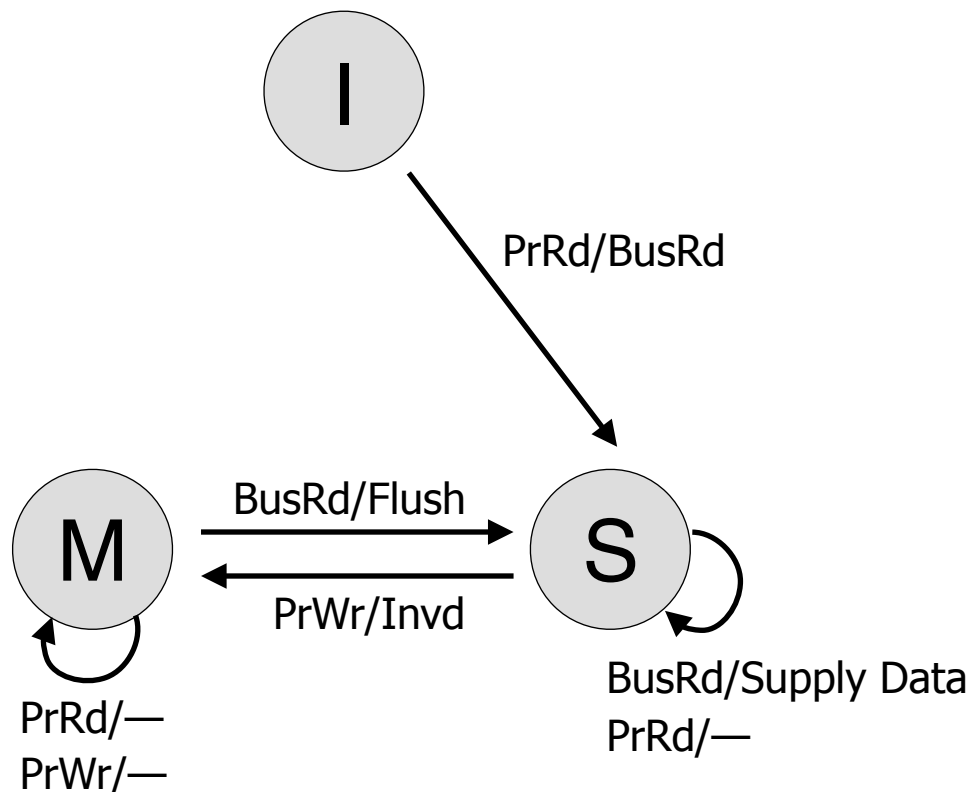


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Write: $x = 7000$

Below: State Transition for x in C2's cache;
Syntax: Event/Action

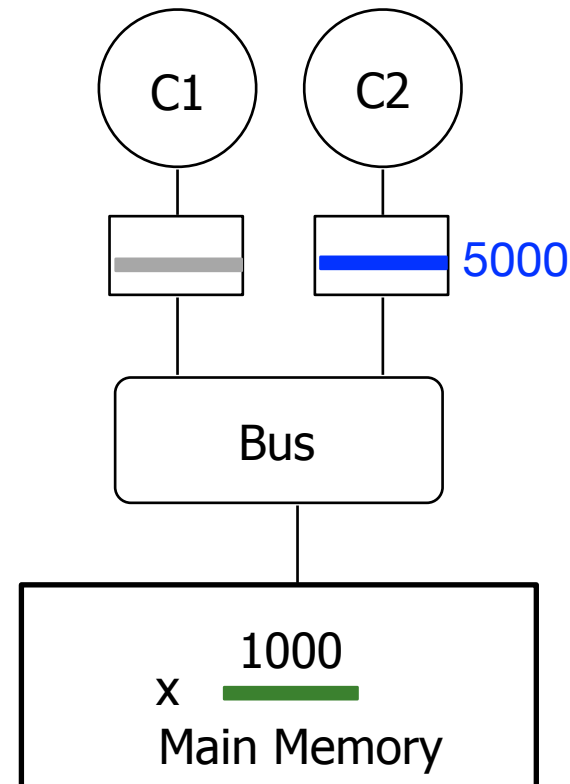
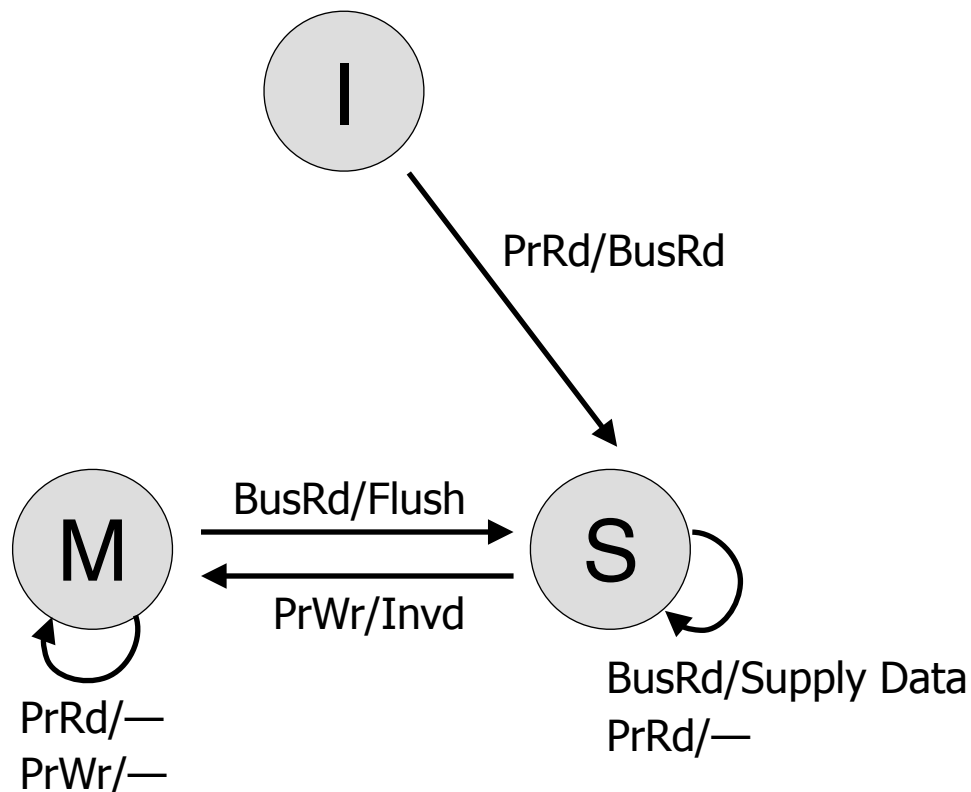


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified**, **Invalid**, **Shared**

Write: $x = 7000$

Below: State Transition for x in C2's cache;
Syntax: Event/Action

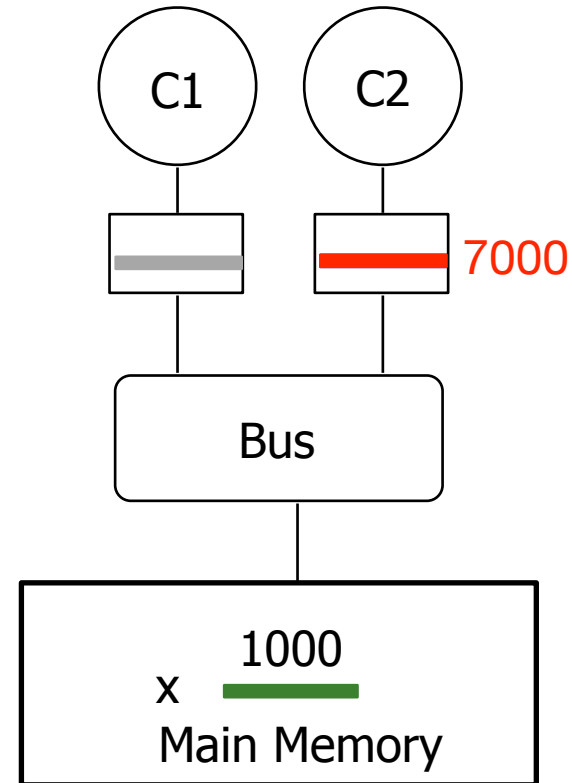
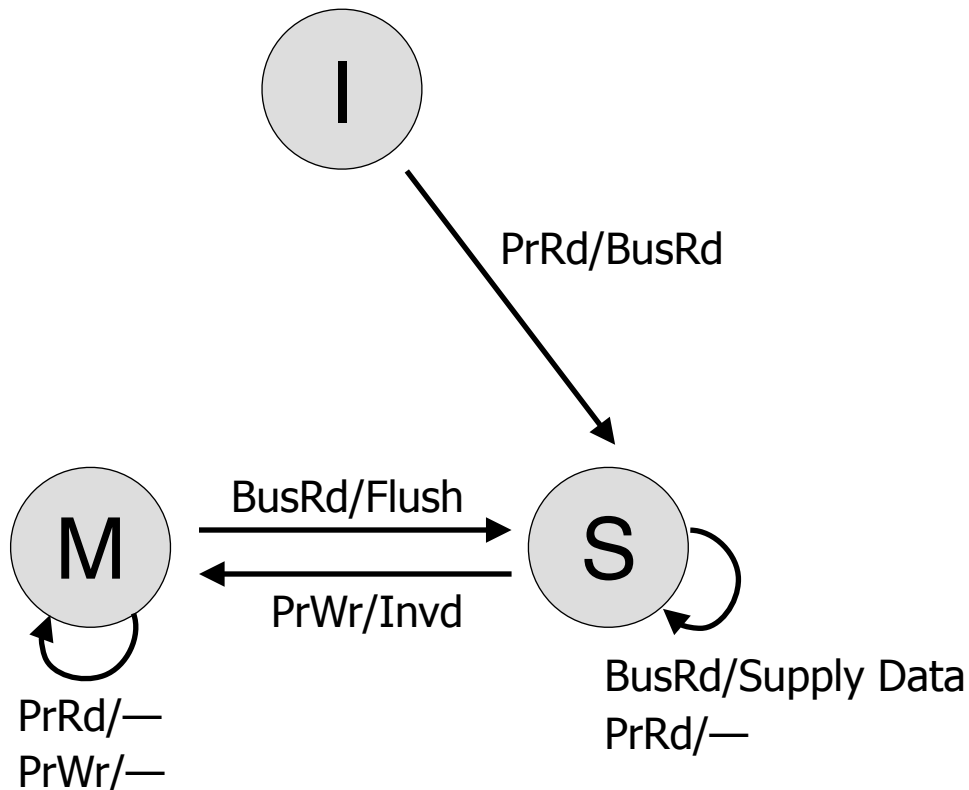


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Write: $x = 7000$

Below: State Transition for x in C2's cache;
Syntax: Event/Action

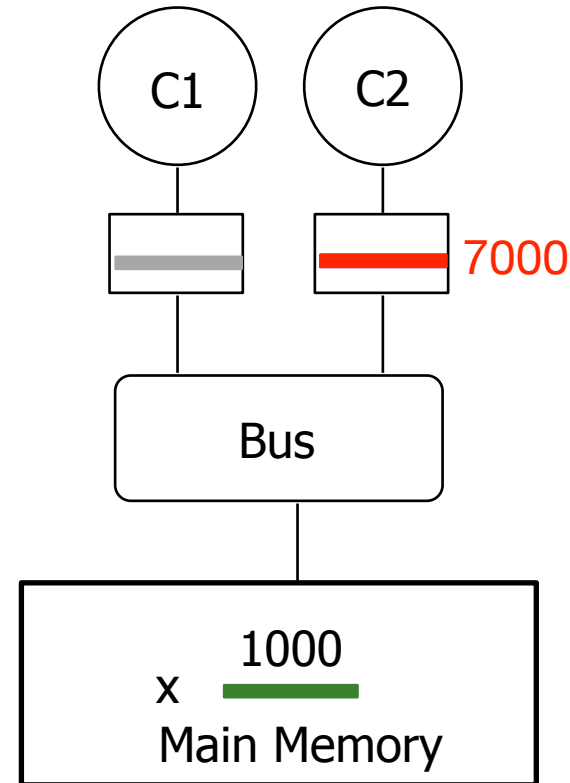
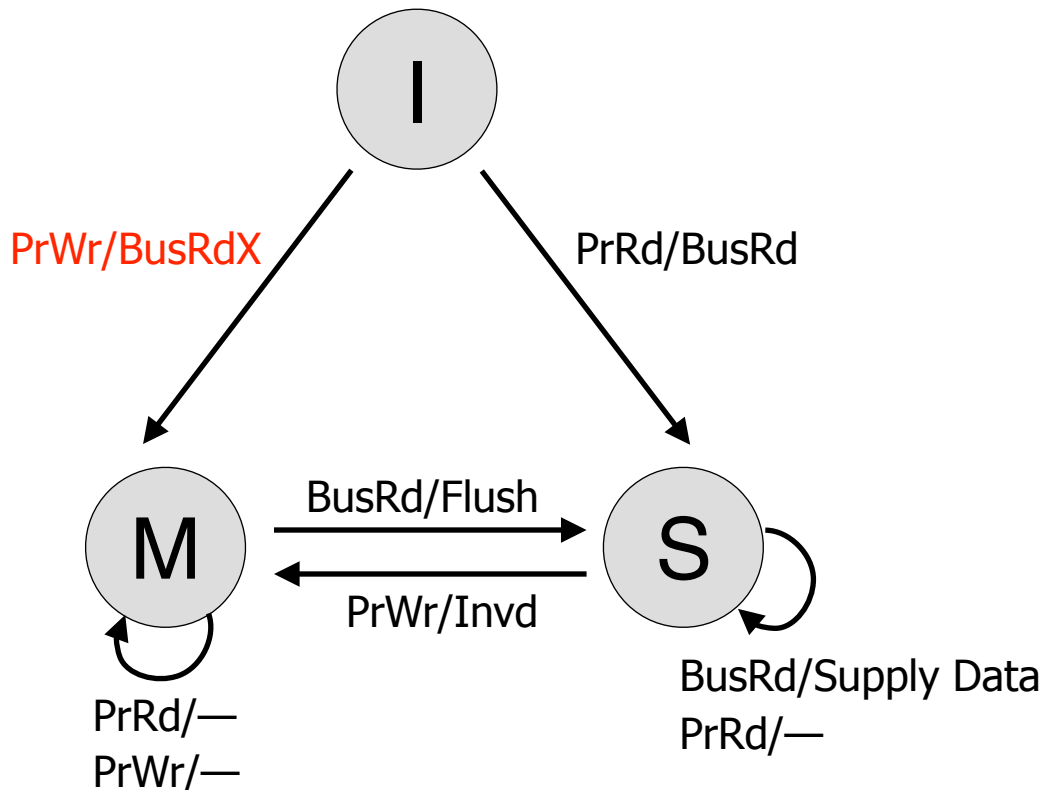


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Write: $x = 7000$

Below: State Transition for x in C2's cache;
Syntax: Event/Action

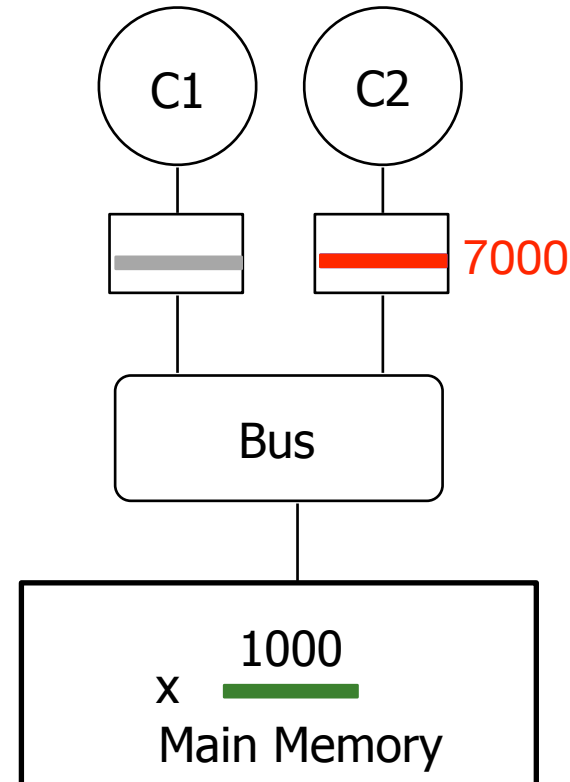
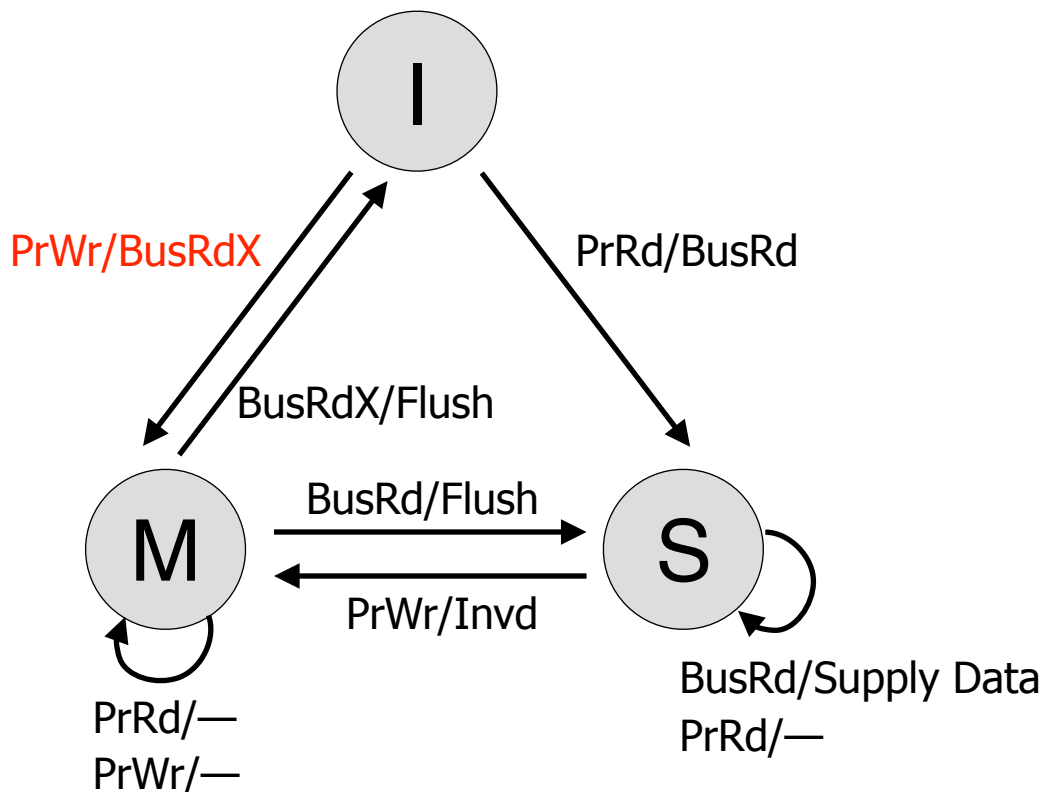


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Write: $x = 7000$

Below: State Transition for x in C2's cache;
Syntax: Event/Action

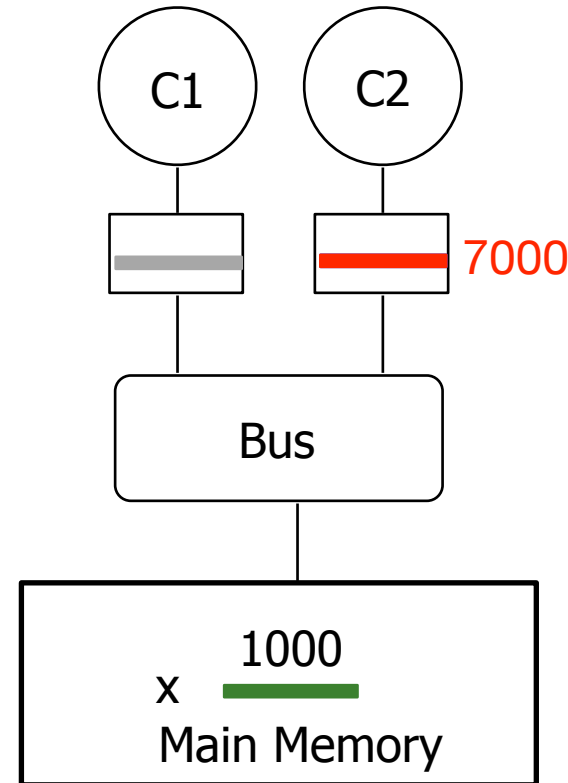
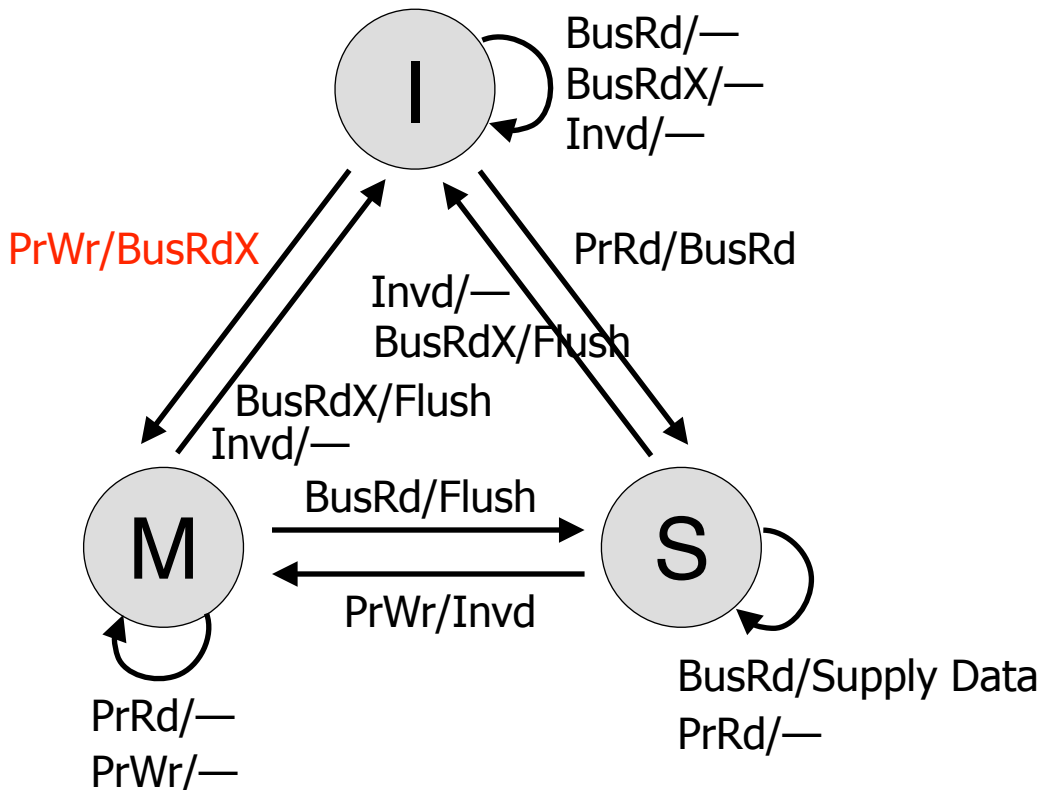


Invalidate-Based Cache Coherence

Associate each cache line with 3 states: **Modified, Invalid, Shared**

Write: $x = 7000$

Below: State Transition for x in C2's cache;
Syntax: Event/Action



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 4th and 4th 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"

Thinking in Parallel is Hard

Thinking in Parallel is Hard

Maybe Thinking is Hard