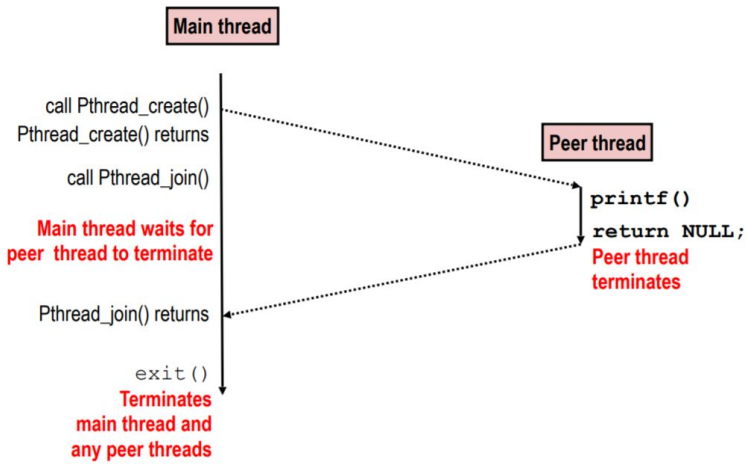
THREADS

By Sandesh Paudel

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`

Execution of Threaded "hello, world"



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

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	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

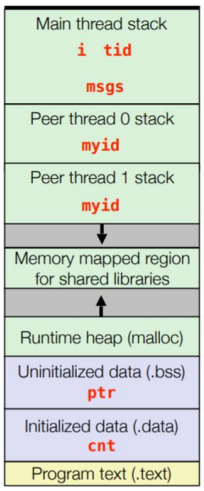
Example Program to Illustrate Sharing

```c
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



i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	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

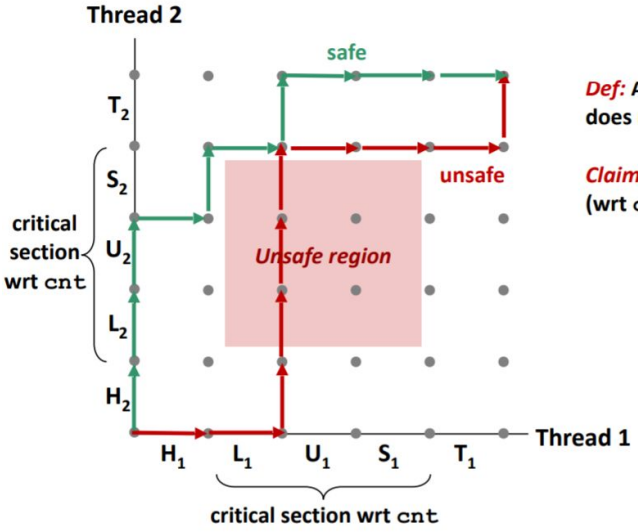
Assembly Code for Counter Loop

C code for counter loop in thread i

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

Asm code for thread i

```
    movq   (%rdi), %rcx          } Hi : Head
    testq  %rcx,%rcx             }
    jle    .L2                   }
    movl   $0, %eax              }
.L3:                             }
    movq   cnt(%rip),%rdx        } Li : Load cnt
    addq   $1, %rdx              } Ui : Update cnt
    movq   %rdx, cnt(%rip)       } Si : Store cnt
    addq   $1, %rax              }
    cmpq   %rcx, %rax            } Ti : Tail
    jne    .L3                   }
.L2:                             }
```



Def: **A trajectory is *safe* iff it does not enter any unsafe region**

Claim: **A trajectory is correct (wrt `cnt`) iff it is safe**

- Terminology:
  - *Binary semaphore*: semaphore whose value is always 0 or 1
  - *Mutex:* binary semaphore used for mutual exclusion
    - P operation: "locking" the mutex
    - V operation: "unlocking" or "releasing" the mutex
    - *"Holding"* a mutex: locked and not yet unlocked.
  - *Counting semaphore*: used as a counter for set of available resources.

Semaphores = Locks

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 */
```

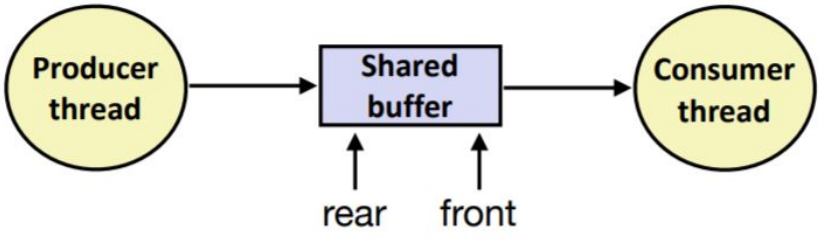
- Surround critical section with P and V:

```
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`.



Removing an item from a shared buffer:

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;

    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)]; /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */

    return item;
}
```

Inserting an item into a shared buffer:

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{

    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;  /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */

}
```

Implementation

```
Sem_init(&sp->mutex, 0, 1);
Sem_init(&sp->items, 0, 0);
Sem_init(&sp->slots, 0, n);
```

Removing an item from a shared buffer:

```
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);                          /* Wait for available item */
    P(&sp->mutex);                          /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)]; /* Remove the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->slots);                          /* Announce available slot */
    return item;
}
```

Inserting an item into a shared buffer:

```
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);                          /* Wait for available slot */
    P(&sp->mutex);                          /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;  /* Insert the item */
    V(&sp->mutex);                          /* Unlock the buffer */
    V(&sp->items);                          /* Announce available item */
}
```

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!

```
Tid[0]:          Tid[1]:
P(s0);           P(s1);
P(s1);           P(s0);
cnt++;           cnt++;
V(s0);           V(s1);
V(s1);           V(s0);
```

```
Tid[0]:          Tid[1]:
P(s0);           P(s0);
P(s1);           P(s1);
cnt++;           cnt++;
V(s0);           V(s1);
V(s1);           V(s0);
```

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 as the main program. The kernel forces the handler to finish before returning to the main program.

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)

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

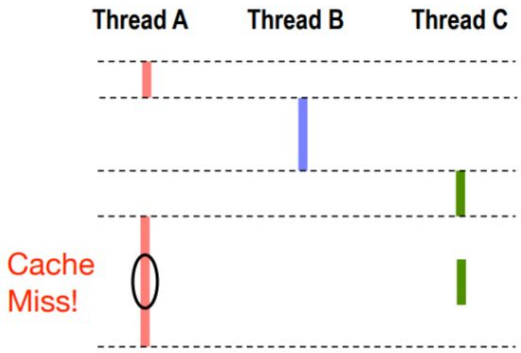
- Completely parallelizable (f = 1): Speedup = N
- Completely sequential (f = 0): Speedup = 1

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

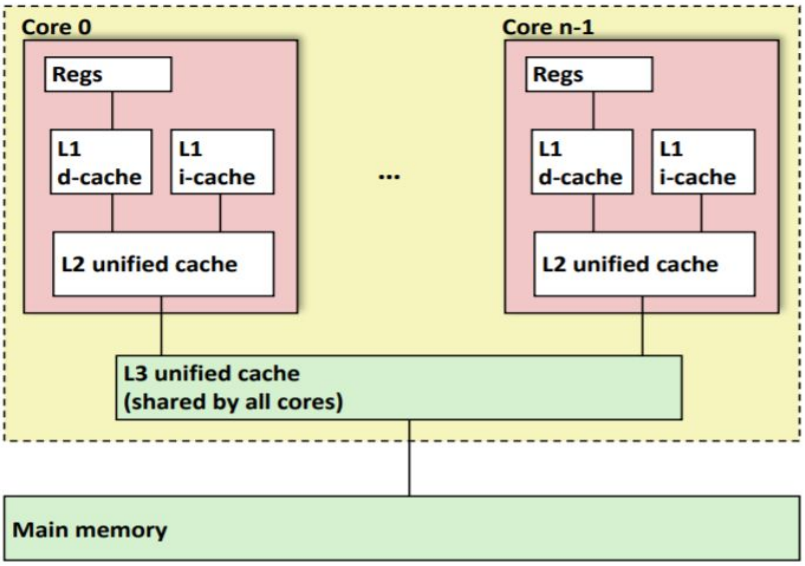
Any benefits?

- 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.



Multi-core for multi threading

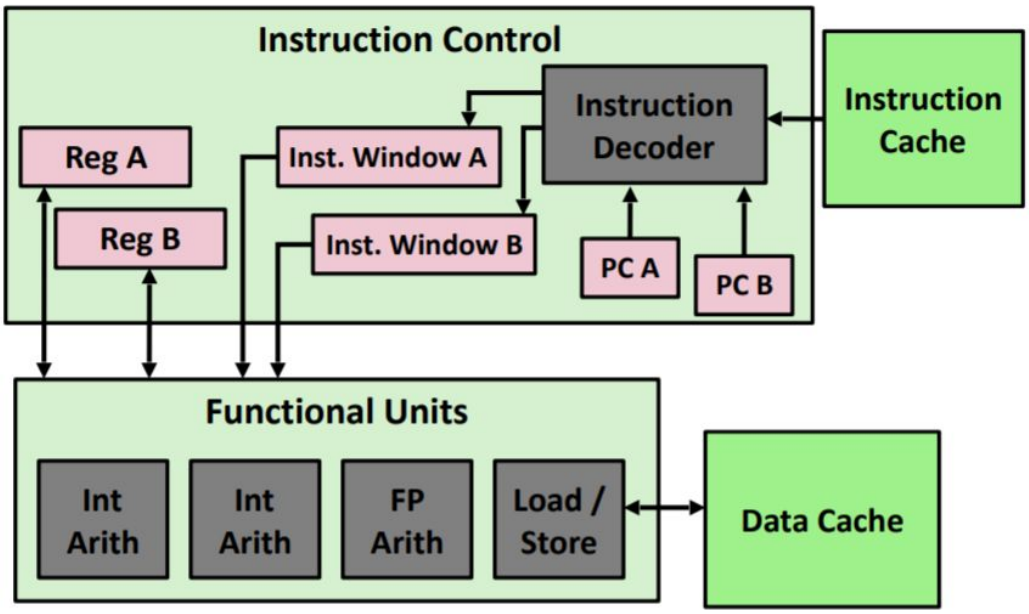
Typical Multi-core Processor



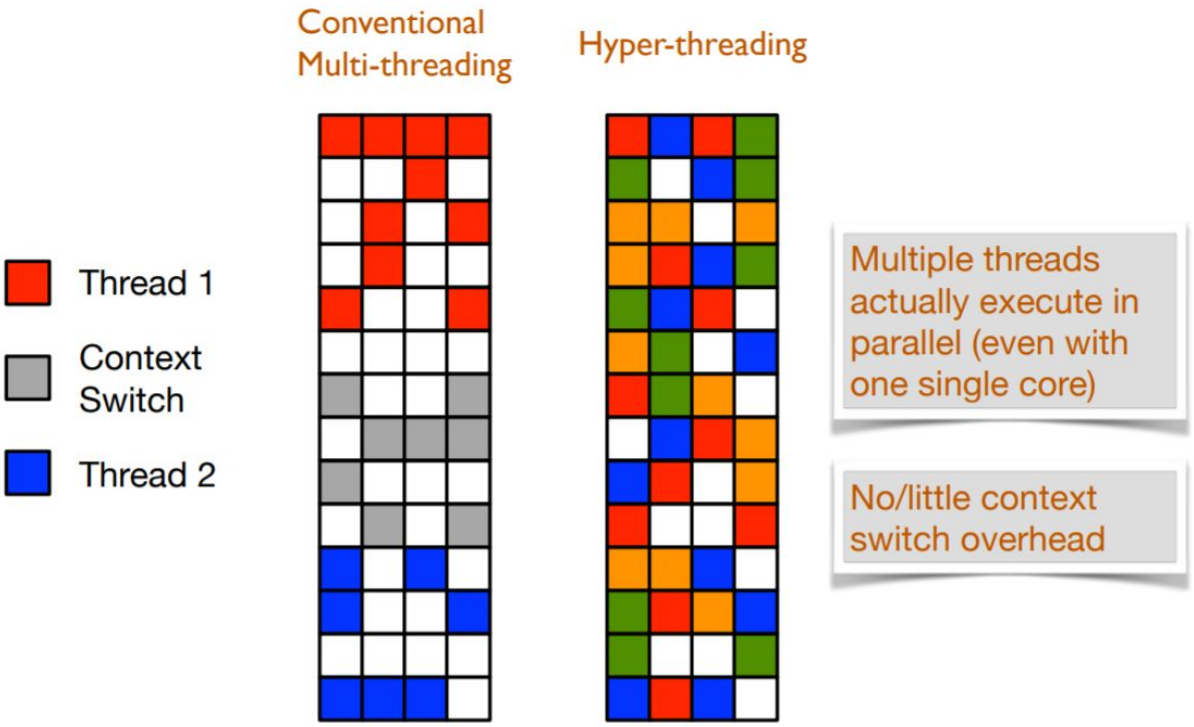
- Traditional multiprocessing: symmetric multiprocessor (SMP)
- Every core is exactly the same. Private registers, L1/L2 caches, etc.
- Share L3 (LLC) and main memory

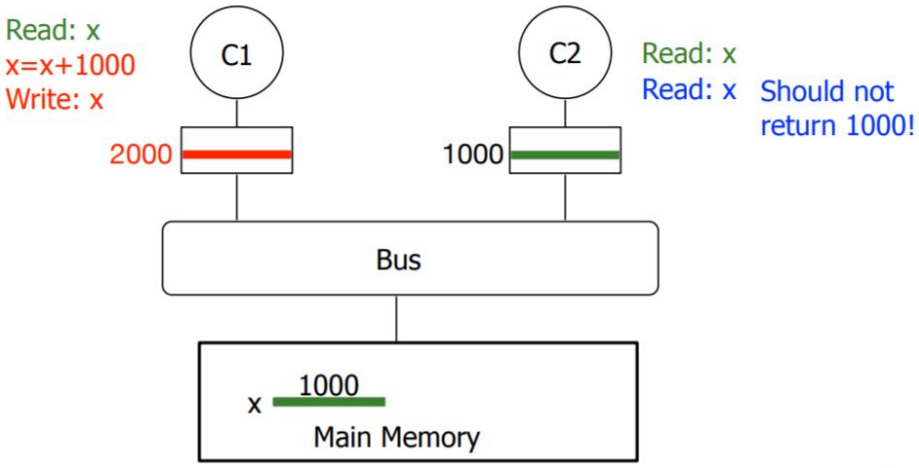
Hyper-threading

- 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



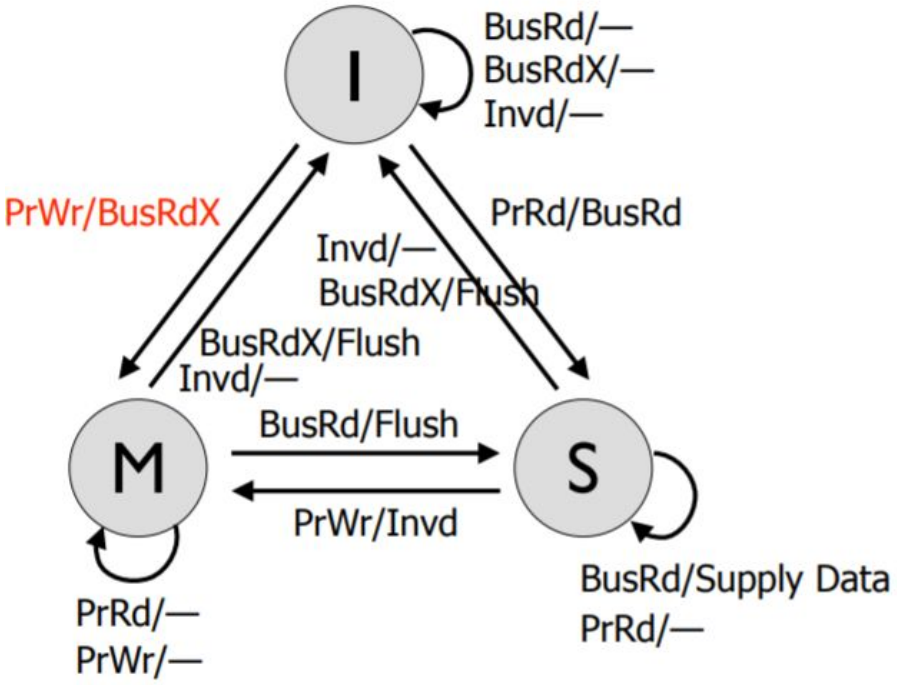


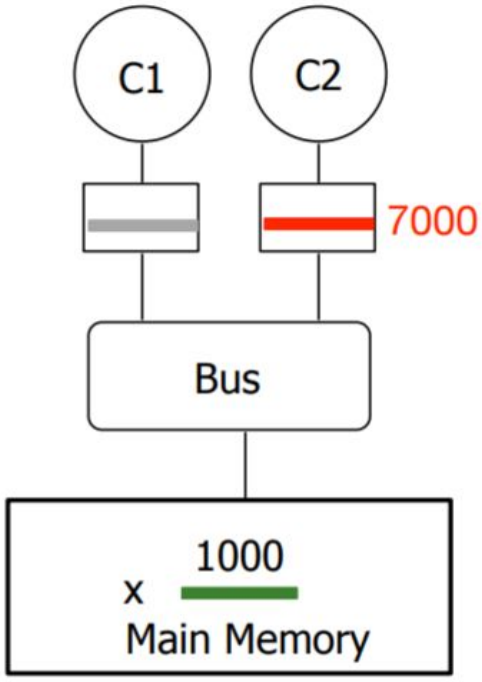
Invalidate-Based Cache Coherence

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





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 shootdown"

Take CSC 251/ECE 204 to learn more about advanced computer architecture concepts.