Multi-threading
Multi-core

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

• A5 due Dec 9th (day after last lecture)
  • Let Abhishek know about slip days
• After today, 2 lectures remain
• Projected grades posted!
<table>
<thead>
<tr>
<th>Grade</th>
<th>Score Range</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>0-65</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>65-68</td>
<td>1</td>
</tr>
<tr>
<td>C-</td>
<td>68-70</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>70-73</td>
<td>4</td>
</tr>
<tr>
<td>C+</td>
<td>73-75</td>
<td>5</td>
</tr>
<tr>
<td>B-</td>
<td>75-80</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>80-87</td>
<td>4</td>
</tr>
<tr>
<td>B+</td>
<td>87-90</td>
<td>7</td>
</tr>
<tr>
<td>A-</td>
<td>90-94</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>94-100</td>
<td>7</td>
</tr>
</tbody>
</table>
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.
The Pthreads "hello, world" Program

```c
/*
 * 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);
}
```

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

```
Main thread
```
call Pthread_create()
Execution of Threaded "hello, world"

Main thread

call Pthread_create()
Execution of Threaded “hello, world”

call Pthread_create()
Execution of Threaded “hello, world”

Main thread

call Pthread_create()
Pthread_create() returns

Peer thread
Execution of Threaded “hello, world”

Main thread

- call Pthread_create()
- Pthread_create() returns
- call Pthread_join()

Peer thread
Execution of Threaded “hello, world”

Main thread

call Pthread_create()
Pthread_create() returns

call Pthread_join()

Main thread waits for peer thread to terminate

Peer thread
Execution of Threaded “hello, world”

Main thread

call Pthread_create()
Pthread_create() returns

call Pthread_join()

Main thread waits for peer thread to terminate

Peer thread

printf()

return NULL;
Peer thread terminates
Execution of Threaded "hello, world"

Main thread

- call Pthread_create()
  - Pthread_create() returns
- call Pthread_join()

  Main thread waits for peer thread to terminate

  Peer thread

  - printf()
  - return NULL;
  - Peer thread terminates

Peer thread

Pthread_join() returns
Execution of Threaded “hello, world”

Main thread

- call Pthread_create()
- Pthread_create() returns
- call Pthread_join()
- Main thread waits for peer thread to terminate
- Pthread_join() returns
- exit()
- Terminates main thread and any peer threads

Peer thread

- printf()
- return NULL;
- Peer thread terminates
Today

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

Thread 1 (main thread)       Thread 2 (peer thread)

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

Shared code and data
- shared libraries
- run-time heap
- read/write data
- read-only code/data

Kernel context:
- VM structures
- Descriptor table
- brk pointer
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);
}
```

Memory mapped region for shared libraries

- Main thread stack
- Peer thread 0 stack
- Peer thread 1 stack
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)
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);
}
```

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);
}
```

Diagram:
- Main thread stack: 
  - `i` 
  - `tid` 
  - `msgs`
- Peer thread 0 stack
- Peer thread 1 stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- `ptr`
- Initialized data (.data)
- Program text (.text)
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);
}
```

```
Memory mapped region for shared libraries

Main thread stack
    i  tid
    msgs

Peer thread 0 stack
    myid

Peer thread 1 stack

Runtime heap (malloc)

Uninitialized data (.bss)
    ptr

Initialized data (.data)

Program text (.text)
```
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);
}
```

Memory mapped region for shared libraries

Runtime heap (malloc)

Uninitialized data (.bss)

Initialized data (.data)

Program text (.text)
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);
}
```

![Diagram of memory regions and stack frames](sharing.c)

- **Main thread stack**
  - i
  - tid
- **Peer thread 0 stack**
  - myid
- **Peer thread 1 stack**
  - myid
- **Memory mapped region for shared libraries**
- **Runtime heap (malloc)**
  - Uninitialized data (.bss)
  - ptr
- **Initialized data (.data)**
  - cnt
- **Program text (.text)**
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);
}
```

Memory mapped region
for shared libraries

Runtime heap (malloc)

Uninitialized data (.bss)

Initialized data (.data)

Program text (.text)
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);
}
```

Diagram:
- **Main thread stack**: `i` `tid` `msgs`
- **Peer thread 0 stack**: `myid`
- **Peer thread 1 stack**: `myid`
- **Memory mapped region for shared libraries**
- **Runtime heap (malloc)**
- **Uninitialized data (.bss)** `ptr` `cnt`
- **Initialized data (.data)**
- **Program text (.text)**

```c
sharing.c
```
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);
}
```

Memory mapped region for shared libraries

- Main thread stack
  - i
  - tid
  - msgs
- Peer thread 0 stack
  - myid
- Peer thread 1 stack
  - myid

Runtime heap (malloc)

- Uninitialized data (.bss)
  - ptr
- Initialized data (.data)
  - cnt

Program text (.text)
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);
}
```

---

**Diagram:**
- Main thread stack: `i tid`
- Message array: `msgs`
- Peer thread 0 stack: `myid`
- Peer thread 1 stack: `myid`
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)
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);
}
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);
}
```

```
main
| p0   | p1   | main
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer thread 0 stack</td>
</tr>
<tr>
<td>Peer thread 1 stack</td>
</tr>
<tr>
<td>Memory mapped region for shared libraries</td>
</tr>
<tr>
<td>Runtime heap (malloc)</td>
</tr>
<tr>
<td>Uninitialized data (.bss)</td>
</tr>
<tr>
<td>Initialized data (.data)</td>
</tr>
<tr>
<td>Program text (.text)</td>
</tr>
</tbody>
</table>
```

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);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);
    for (i = 0; i < niters; i++)
        cnt++;
    return NULL;
}

badcnt.c

https://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/
Improper Synchronization

```c
/* 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);
}
```

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);
    for (i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

```bash
linux> ./badcnt
OK cnt=20000
linux> ./badcnt
BOOM! cnt=13051
```

cnt should be 20,000.

What went wrong?

https://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/
Assembly Code for Counter Loop

C code for counter loop in thread i

\[
\text{for } (i = 0; i < \text{niters}; i++) \\
\quad \text{cnt}++; \\
\]

Asm code for thread i

\[
\begin{align*}
\text{movq} & \quad (\%rdi), \%rcx \\
\text{testq} & \quad \%rcx,\%rcx \\
\text{jle} & \quad .L2 \\
\text{movl} & \quad \%eax, \%eax \\
\text{L3:} & \\
\text{movq} & \quad \text{cnt}(\%rip),\%rdx \\
\text{addq} & \quad \%rdx \\
\text{L2:} & \\
\text{cmpq} & \quad \%rcx, \%rax \\
\text{jne} & \quad .L3 \\
\text{movq} & \quad \%rdx, \text{cnt}(\%rip) \\
\text{addq} & \quad \%eax \\
\text{U_i:} & \quad \text{Update cnt} \\
\text{S_i:} & \quad \text{Store cnt} \\
\text{T_i:} & \quad \text{Tail} \\
\end{align*}
\]
Concurrent Execution

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

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;2&lt;/sub&gt;</th>
<th>cnt (shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Thread 1 critical section**
- **Thread 2 critical section**

\[ \text{movq cnt(%rip),%rdx} \]
\[ \text{addq $1, %rdx} \]
\[ \text{movq %rdx, cnt(%rip)} \]
Concurrent Execution (cont)

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

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr_i</th>
<th>%rdx_1</th>
<th>%rdx_2</th>
<th>cnt (shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L_1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U_1</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L_2</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S_1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S_2</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
L_i & : \text{movq cnt(rip),%rdx} \\
U_i & : \text{addq $1, %rdx} \\
S_i & : \text{movq %rdx, cnt(rip)}
\end{align*}
\]
Assembly Code for Counter Loop

C code for counter loop in thread i

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

**Asm code for thread i**

```assembly
movq (%rdi), %rcx
getTime %rcx,%rcx
jle .L2
movl $0, %eax
.L3:
movq cnt(%rip),%rdx
addq $1, %rdx
movq %rdx, cnt(%rip)
addq $1, %rax
cmpq %rcx, %rax
jne .L3
.L2:
```

`H_i`: Head

`L_i`: Load cnt

`U_i`: Update cnt

`S_i`: Store cnt

`T_i`: Tail

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!

```
 movq (%rdi), %rcx
 testq %rcx,%rcx
 jle .L2
 movl $0, %eax

.L3:
 movq cnt(%rip),%rdx
 addq $1, %rdx
 movq %rdx, cnt(%rip)
 addq $1, %rax
 cmpq %rcx, %rax
 jne .L3

.L2:
```

- \( H_i \): Head
- \( L_i \): Load cnt
- \( U_i \): Update cnt
- \( S_i \): Store cnt
- \( T_i \): Tail
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.

```
  movq (%rdi), %rcx
  testq %rcx,%rcx
  jle   .L2
  movl  $0, %eax
  .L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
  .L2:
  
  Hi : Head
  L_i : Load cnt
  U_i : Update cnt
  S_i : Store cnt
  T_i : Tail
```
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`:

```c
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`:

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

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?

```c
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.

```c
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.

```c
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.

```c
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.

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

```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!
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);
}
Avoiding Deadlock

Acquire shared resources in same order

Tid[0]:
P(s_0);
P(s_1);
cnt++;
V(s_0);
V(s_1);

Tid[1]:
P(s_1);
P(s_0);
cnt++;
V(s_1);
V(s_0);

Tid[0]:
P(s_0);
P(s_1);
cnt++;
V(s_0);
V(s_1);

Tid[1]:
P(s_0);
P(s_1);
cnt++;
V(s_1);
V(s_0);
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.

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.
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?

This implementation will get into a deadlock.

```c
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?

- 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);
}

• 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 $1, \ldots, n-1$ 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, …, 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:

```java
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
  P(A)
  Update shared data
  V(A)
}
```
Amdahl’s Law

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

Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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)

\[
1 - f + \frac{1}{N}
\]

Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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

Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” 1967.
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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
  • 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

- 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

```
xorg %rax, %rax
jne L1 # Not taken
Stall
Stall
irmovq $1, %rax # Fall Through
L1  irmovq $4, %rcx # Target
irmovq $3, %rax # Target + 1
```
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...)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Context Switch</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Coarse-grained MT**

**Fine-grained MT without hw support**

**Fine-grained MT with hw support**
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

<table>
<thead>
<tr>
<th>Functional Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why these empty slots?</td>
</tr>
</tbody>
</table>

- **Thread 1**: Red
- **Context Switch**: Gray
- **Thread 2**: Blue
Simultaneous Multi-Threaded (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-Threaded (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

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

Thread 1
Context Switch
Thread 2
Thread 3
Thread 4
Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT

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

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)
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.

Thread 0
Mem[A] = 1

Thread 1
...

Print Mem[A]
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.

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem[A] = 1</td>
<td>Mem[A] = 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Print Mem[A]</td>
<td>Print Mem[A]</td>
</tr>
</tbody>
</table>
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?

```
Thread 0
Mem[A] = 1

Thread 1
...

Print Mem[A]
```
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)

![Diagram showing two cores (C1, C2) connected to a bus and main memory, with data read and written.]
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 \textit{cache} the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)

\[ x = x + 1000 \]

Read: \( x \)

Write: \( x \)

Should not return 1000!
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 the 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 4th and 4th revised eds.)

• Also very useful
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 shootdown”