Synchronization: Basics

15-213 / 18-213: Introduction to Computer Systems
23rd Lecture, Nov. 13, 2014

Instructors:
Greg Ganger, Greg Kesden, and Dave O’Hallaron
Today

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
Process: Traditional View

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

**Process context**

<table>
<thead>
<tr>
<th>Program context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data registers</td>
</tr>
<tr>
<td>Condition codes</td>
</tr>
<tr>
<td>Stack pointer (SP)</td>
</tr>
<tr>
<td>Program counter (PC)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM structures</td>
</tr>
<tr>
<td>Descriptor table</td>
</tr>
<tr>
<td>brk pointer</td>
</tr>
</tbody>
</table>

**Code, data, and stack**

- stack
- shared libraries
- run-time heap
- read/write data
- read-only code/data
Process: Alternative View

- Process = thread + code, data, and kernel context

**Thread**

- **Program context:**
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

**Code, data, and kernel context**

- **brk**
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

- **PC**
  - 0

**Kernel context:**

- VM structures
- Descriptor table
- brk pointer
Process with Two Threads

**Thread 1**
- **Program context:**
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

**Thread 2**
- **Program context:**
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

**Code, data, and kernel context**
- Sharing libraries
- Run-time heap
- Read/write data
- Read-only code/data

**Kernel context:**
- VM structures
- Descriptor table
- Brk pointer

**SP**
- Stack

**PC**
- Stack pointer (SP)

**Brk**
- Program counter (PC)
Threads vs. Processes

Threads and processes: similarities
- Each has its own logical control flow
- Each can run concurrently with others
- Each is scheduled and context switched by the kernel

Threads and processes: differences
- Threads share code and data, processes (typically) do not
- Threads are less expensive than processes
  - Process control (creating and reaping) is more expensive than thread control
  - Context switches for processes more expensive than for threads
Pros and Cons of Thread-Based Designs

- Easy to share data structures between threads
  - e.g., logging information, file cache
- Threads are more efficient than processes
- Unintentional sharing can introduce subtle and hard-to-reproduce errors!
Today

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
Shared Variables in Threaded C Programs

Question: Which variables in a threaded C program are shared?
- The answer is not as simple as “global variables are shared” and “stack variables are private”

Requires answers to the following questions:
- What is the memory model for threads?
- How are instances of variables mapped to memory?
- How many threads might reference each of these instances?

Def: A variable $x$ is shared if and only if multiple threads reference some instance of $x$. 
Threads Memory Model

- **Conceptual model:**
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments of the process virtual address space
    - Open files and installed handlers

- **Operationally, this model is not strictly enforced:**
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

*The mismatch between the conceptual and operation model is a source of confusion and errors*
Example Program to Illustrate Sharing

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

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);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread’s stack indirectly through global ptr variable
Mapping Variable Instances to Memory

- Global variables
  - *Def*: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- Local variables
  - *Def*: Variable declared inside function without *static* attribute
  - Each thread stack contains one instance of each local variable

- Local static variables
  - *Def*: Variable declared inside function with the *static* attribute
  - Virtual memory contains exactly one instance of any local static variable.
char **ptr; /* global var */

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);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}

Global var: 1 instance (ptr [data])

Local vars: 1 instance (i.m, msgs.m)

Local var: 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

Local static var: 1 instance (cnt [data])
Shared Variable Analysis

- Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

- Answer: A variable $x$ is shared iff multiple threads reference at least one instance of $x$. Thus:
  - ptr, cnt, and msgs are shared
  - i and myid are not shared
Today

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
badcnt.c: Improper Synchronization

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

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

    niters = atoi(argv[1]);
    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 * niters))
        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;
}
```

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

linux> ./badcnt 10000
BOOM! cnt=13051

linux>
```

cnt should equal 20,000.

What went wrong?
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**

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

- $I_i$ denotes that thread $i$ executes instruction $I$
- $\%rdx_i$ is the content of $\%rdx$ in thread $i$’s context

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr$_i$</th>
<th>$%rdx_1$</th>
<th>$%rdx_2$</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H$_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<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>1</td>
<td>S$_1$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>H$_2$</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S$_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>T$_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>T$_1$</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

Thread 1 critical section

Thread 2 critical section

**OK**
Concurrent Execution (cont)

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

<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</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H&lt;sub&gt;1&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<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>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>0</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>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!
Concurrent Execution (cont)

How about this ordering?

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instrᵢ</th>
<th>%rdx₁</th>
<th>%rdx₂</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H₁</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L₁</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>H₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L₂</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U₂</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S₂</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>U₁</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S₁</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T₁</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T₂</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!

We can analyze the behavior using a progress graph
Progress Graphs

A *progress graph* depicts the discrete *execution state space* of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible *execution state* (Inst₁, Inst₂).

E.g., (L₁, S₂) denotes state where thread 1 has completed L₁ and thread 2 has completed S₂.
A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:
H1, L1, U1, H2, L2, S1, T1, U2, S2, T2
L, U, and S form a **critical section** with respect to the shared variable `cnt`.

Instructions in critical sections (wrt to some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form **unsafe regions**.
Critical Sections and Unsafe Regions

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

Claim: A trajectory is correct (wrt cnt) iff it is safe.
Enforcing Mutual Exclusion

- **Question:** How can we guarantee a safe trajectory?

- **Answer:** We must *synchronize* the execution of the threads so that they never have an unsafe trajectory.
  - i.e., need to guarantee *mutually exclusive access* to critical regions

- **Classic solution:**
  - Semaphores (Edsger Dijkstra)

- **Other approaches (out of our scope)**
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)
Today

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
Semaphores

- **Semaphore**: non-negative global integer synchronization variable. Manipulated by $P$ and $V$ operations.

- **$P(s)$**
  - If $s$ is nonzero, then decrement $s$ by 1 and return immediately.
  - If $s$ is zero, then suspend thread until $s$ becomes nonzero and the thread is restarted by a $V$ operation.
  - After restarting, the $P$ operation decrements $s$ and returns control to the caller.

- **$V(s)$**:
  - Increment $s$ by 1.
  - If there are any threads blocked in a $P$ operation waiting for $s$ to become non-zero, then restart exactly one of those threads, which then completes its $P$ operation by decrementing $s$.

- **Semaphore invariant**: $(s >= 0)$
C Semaphores Operations

Pthreads functions:

```c
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

CS:APP wrapper functions:

```c
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```
badcnt.c: Improper Synchronization

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

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

    niters = atoi(argv[1]);
    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 * niters))
        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;
}
```

How can we fix this using semaphores?
Using Semaphores for Mutual Exclusion

Basic idea:
- Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
- Surround corresponding critical sections with $P(mutex)$ and $V(mutex)$ operations.

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.
**goodcnt.c: 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`.

```bash
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
```
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with \( P \) and \( V \) operations on semaphore \( s \) (initially set to 1)

Semaphore invariant creates a forbidden region that encloses unsafe region that cannot be entered by any trajectory.

Initially \( s = 1 \)
Summary

- Programmers need a clear model of how variables are shared by threads.

- Variables shared by multiple threads must be protected to ensure mutually exclusive access.

- Semaphores are a fundamental mechanism for enforcing mutual exclusion.