Synchronization: Basics

15-213 / 18-213: Introduction to Computer Systems
24th Lecture, Nov. 20, 2012

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

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

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

**Process context**

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

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

**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)

SP → stack

**Code, data, and kernel context**

- shared libraries
- run-time heap
- read/write data
- read-only code/data
- 0

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

brk → PC
Process with Two Threads

Thread 1

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

SP → stack

Thread 2

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

SP → stack

Code, data, and kernel context

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

0

Kernel context:
- VM structures
- Descriptor table
- brk pointer
Threads vs. Processes

- **Threads and processes: similarities**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) 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 as 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

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

int main()
{
    int 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);
}

/* thread routine */
void *thread(void *vargp)
{
    int myid = (int) vargp;
    static int cnt = 0;
    printf("[%d]: %s (svar=%d)\n",
           myid, ptr[myid], ++cnt);
}
```

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.
Mapping Variable Instances to Memory

**Global var:** 1 instance (ptr [data])

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

int main()
{
    int 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);
}
```

**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])

```c
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

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

**Local static var:** 1 instance (cnt [data])

```c
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
Improper Synchronization

```c
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
    int niters = atoi(argv[1]);
    pthread_t tid1, tid2;

    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=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

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

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

linux> ./badcnt 10000
BOOM! cnt=13051

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 < niter; i++)
    cnt++;
```

Corresponding assembly code

```assembly
    movl (%rdi),%ecx
    movl $0,%edx
    cmpl %ecx,%edx
    jge .L13

.L11:
    movl cnt(%rip),%eax
    incl %eax
    movl %eax,cnt(%rip)
    incl %edx
    cmpl %ecx,%edx
    jl .L11

.L13:
```

| Head (H<sub>i</sub>) | Load cnt (L<sub>i</sub>) | Update cnt (U<sub>i</sub>) | Store cnt (S<sub>i</sub>) | Tail (T<sub>i</sub>) |
## Concurrent Execution

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

- $l_i$ denotes that thread $i$ executes instruction $l$
- $%eax_i$ is the content of $%eax$ in thread $i$’s context

<table>
<thead>
<tr>
<th>$i$ (thread)</th>
<th>$instr_i$</th>
<th>$%eax_1$</th>
<th>$%eax_2$</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>%eax&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%eax&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>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%eax&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%eax&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></td>
</tr>
<tr>
<td>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!

- We can analyze the behavior using a progress graph
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
Critical Sections and Unsafe Regions

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:
  - $P(s)$: `[ while (s == 0) wait(); s--; ]`
    - Dutch for "Proberen" (test)
  - $V(s)$: `[ s++; ]`
    - Dutch for "Verhogen" (increment)

- OS kernel guarantees that operations between brackets `[ ]` are executed indivisibly
  - Only one $P$ or $V$ operation at a time can modify $s$.
  - When `while` loop in $P$ terminates, only that $P$ can decrement $s$

- Semaphore invariant: $(s \geq 0)$
C Semaphore 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 */
```
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
    int niters = atoi(argv[1]);
    pthread_t tid1, tid2;

    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=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int i, niters = *((int *)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 int 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 much slower than badcnt.c.
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.
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.
Threads vs. Processes (cont.)

- Processes form a tree hierarchy
- Threads form a pool of peers
  - Each thread can kill any other
  - Each thread can wait for any other thread to terminate
  - Main thread: first thread to run in a process
Posix Threads (Pthreads) Interface

- **Pthreads**: Standard interface for ~60 functions that manipulate threads from C programs
  - Threads run thread routines:
    - `void *threadroutine(void *vargp)`
  - Creating and reaping threads
    - `pthread_create(pthread_t *tid, …, func *f, void *arg)`
    - `pthread_join(pthread_t tid, void **thread_return)`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel(pthread_t tid)`
    - `pthread_exit(void *tread_return)`
    - `return` (in primary thread routine terminates the thread)
    - `exit` (terminates all threads)
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);
}

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

- **Thread attributes** (usually NULL)
- **Thread arguments** (void *p)
- **assigns return value** (void **p)