Lecture 27: Synchronization

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
A Multi-threaded process

- Each thread has its own execution context
- Threads share the same virtual address space

Thread 1

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

Shared address space

- stack 1
- stack 2
- shared libraries
- run-time heap
- read/write data
- read-only code/data

0

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

Thread 2

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

• Conceptual model:
  – Multiple threads run within the context of a single process
  – Each thread has its own separate thread execution 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
    • Open files etc.

• Operationally, this model is not strictly enforced:
  – Any thread can read and write virtual address space, including the stack of any other thread
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’s 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;
int main()
{
    int i;
    pthread_t tid[2];
    char *msgs[2] = {"Hello from foo","Hello from bar"};
    ptr = msgs;

    for (i = 0; i < 2; i++)
    {
        pthread_create(&tid[i], NULL, run,(void *)i);
        for (i = 0; i < 2; i++)
        {
            pthread_join(tid[i],NULL);
        }
    }
}

/* thread routine */
void *run(void *x)
{
    int myid = (int) x;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
badcnt.c: Improper Synchronization

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

int main(int argc, char **argv)
{
    int i, niters = atoi(argv[1]);
    pthread_t tid[2];

    for (i=0; i<2; i++)
        pthread_create(&tid[i], NULL,
                        run, (void *)niters);
    for (i=0; i<2; i++)
        pthread_join(tid[i], 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 *run(void *x)
{
    int i, niters = (int)x;
    for (i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

```bash
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

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

Corresponding assembly code

```
.L11:
    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 (Hᵢ)**
- **Load cnt (Lᵢ)**
- **Update cnt (Uᵢ)**
- **Store cnt (Sᵢ)**
- **Tail (Tᵢ)**
Concurrent Execution

- Any sequentially consistent interleaving is possible
  - $%eax_i$ 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’s critical section
Thread 2’s critical section

OK
Concurrent Execution (cont)

- **Race:** two threads increment counter, 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(_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></td>
</tr>
<tr>
<td>2</td>
<td>H(_2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L(_2)</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U(_2)</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S(_2)</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U(_1)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S(_1)</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T(_1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T(_2)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Oops!
Enforcing Mutual Exclusion

- To guarantee mutually exclusive access to a critical region, we must *synchronize* the execution of the threads.

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

- **Other approaches (not discussed in this class):**
  - Mutex and condition variables (pthreads)
  - Monitors (Java)
Semaphores

- **Semaphore**: non-negative integer synchronization variable

- **Manipulated by P and V operations:**
  - $P(s)$: [ while (s == 0) wait(); s--; ]
    - Dutch for "Proberen" (test)
    - Other names: down, wait
  - $V(s)$: [ s++; ]
    - Dutch for "Verhogen" (increment)
    - Other names: up, post

- **OS kernel guarantees that operations between brackets [ ] are executed indivisibly**
  - Only one $P$ or $V$ operation at a time can modify $s$. 
Using Semaphores for Mutual Exclusion

• **Basic idea:**

  shared variables v1, v2;
  semaphore s; /* initialized to 1*/
  ...

  Associate a semaphore with each shared variable, or a set of shared variables

  Surround critical section with P(s) and V(s)

  ...critical section reading/writing v1,v2
Using Semaphores for Mutual Exclusion

• 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.
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) */
```

**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 */
```
Improper Synchronization

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

int main(int argc, char **argv)
{
    int i, niters = atoi(argv[1]);
    pthread_t tid[2];

    for (i=0; i<2; i++)
        pthread_create(&tid[i], NULL,
                        run, (void *)niters);
    for (i=0; i<2; i++)
        pthread_join(tid[i], 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 *run(void *x)
{
    int i, niters = (int)x;
    for (i = 0; i < niters; i++)
        cnt++;
    return NULL;
}
```

How can we fix this using semaphores?
goodcnt.c: Proper Synchronization

- Define and initialize a mutex for the shared variable \texttt{cnt}:

\begin{verbatim}
volatile int cnt = 0;  /* Counter */
sem_t mutex;           /* Semaphore that protects cnt */
sem_init(&mutex, 0, 1); /* mutex = 1 */
\end{verbatim}

■ **Surround** critical section with \texttt{P} and \texttt{V}:

\begin{verbatim}
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
}
\end{verbatim}

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

Warning: It’s much slower than \texttt{badcnt.c}.
Generalization of mutual exclusion: Readers-Writers Problem

• Problem statement:
  – *Reader* threads only read shared variables
  – *Writer* threads modify shared variables
  – Writers must have exclusive access to the object
  – >1 readers can read the object concurrently

• Example: caching web server
  – >1 (reader) threads can simultaneously read pages from cache
  – Only one (writer) thread can modify cache at any time
Readers-Writers Problem

Readers:

```c
void do_read(void)
{
    am i the first reader in?
    if so, try to lock mutex to exclude writer

    /* do some reading */

    am I the last reader out?
    if so, unlock mutex
}
```

Writers:

```c
void do_write(void)
{
    try to lock mutex to exclude readers

    /* do some writing */

    unlock mutex
}
```
Solution to First Readers-Writers Problem

Readers:

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void do_read(void)
{
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
        P(&w);
    V(&mutex);

    /* do some reading */
    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
        V(&w);
    V(&mutex);
}
```

Writers:

```c
void do_write(void)
{
    P(&w);

    /* do some writing */
    V(&w);
}
```

rw1.c
Synchronize Access to Shared Resources

- **Producer/consumer pattern:**
  - Producer inserts item in buffer (waits if buffer is full)
  - Consumer removes item from buffer (waits if buffer is empty)

- **Examples**
  - Network server
    - Producer threads read from sockets, put clients’ requests in buffer
    - Consumer threads process clients’ requests
Producer-Consumer on 1-element Buffer: naïve polling

```
struct {
    int buf; /* 1 element buffer */
    int n; /* # of items in buffer*/
} shared;

shared sbuf;

void produce(int x) {
    while (sbuf.n == 1)
        sleep(1);
    sbuf.buf = x;
    sbuf.n = 1;
}

int consume() {
    int x;
    while (sbuf.n == 0)
        sleep(1);
    sbuf.n = 0;
    x = sbuf.buf;
    return x;
}

void *p_run(void *a) {
    for (i = 0; i < 5; i++)
        produce(i);
}

void *c_run(void *a) {
    for (i = 0; i < 5; i++)
        printf("consume %d",
        consume());
}

int main() {
    pthread_t pid, cid;
    pthread_create(&pid,NULL,p_run,
                   NULL);
    pthread_create(&cid,NULL,c_run,
                   NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
}
```
Producer-Consumer on 1-element Buffer

```c
struct {
    int buf; /* 1 element buffer */
    int n; /* # of items in buffer*/
    sem_t empty; /*init to 1*/
    sem_t full; /*init to 0*/
} shared;

shared sbuf;

void produce(int x) {
    /* no sleeping necessary*/
    sbuf.buf = x;
    sbuf.n = 1;
}

int consume() {
    int x;

    /* no sleeping necessary*/
    x = sbuf.buf;
    sbuf.n = 0;

    return x;
}
```
Producer-Consumer on 1-element Buffer

```c
struct {
    int buf; /* 1 element buffer */
    int n; /* # of items in buffer*/
    sem_t empty; /*init to 1*/
    sem_t full; /*init to 0*/
} shared;

shared sbuf;

void produce(int x) {
    P(&sbuf.empty);
    /* no sleeping necessary*/
    sbuf.buf = x;
    sbuf.n = 1;
    V(&sbuf.full);
}

int consume() {
    int x;
    P(&sbuf.full);
    /* no sleeping necessary*/
    x = sbuf.buf;
    sbuf.n = 0;
    V(&sbuf.empty);
    return x;
}
```
How about \( n \)-element Buffer?

- Requires a mutex and two counting semaphores:
  - \textbf{slots}: \# of available slots in buffer
  - \textbf{items}: \# of available items
  - \textbf{mutex}: coordinate mutually exclusive access to buffer state

- Because multiple producers/consumers may concurrently modify buffer state
Another Worry: Deadlock
Another Worry: Deadlock

• A process is *deadlocked* if it is waiting for a condition that will never be true.

• Typical Scenario
  – Processes 1 and 2 need 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!
int main()
{
    pthread_t tid[2];
    int i;
    sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    for (i = 0; i < 2; i++)
        pthread_create(&tid[i], NULL, count, (void*) i);
    for (i = 0; i < 2; i++)
        pthread_join(tid[i], NULL);
}

void *count(void *x)
{
    int i, id = (int) x;
    for (i = 0; i < 5; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
Avoiding Deadlock

```c
int main()
{
    pthread_t tid[2];
    int i;
    sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    for (i = 0; i < 2; i++)
        pthread_create(&tid[i], NULL, count, (void*) i);
    for (i = 0; i < 2; i++)
        pthread_join(tid[i], NULL);
}

void *count(void *x)
{
    int i, id = (int) x;
    for (i = 0; i < 5; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[0]); V(&mutex[1]);
    }
    return NULL;
}
```

Acquire resources in same order

<table>
<thead>
<tr>
<th>Tid[0]:</th>
<th>Tid[1]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(m0);</td>
<td>P(m0);</td>
</tr>
<tr>
<td>P(m1);</td>
<td>P(m1);</td>
</tr>
<tr>
<td>cnt++;</td>
<td>cnt++;</td>
</tr>
<tr>
<td>V(m0);</td>
<td>V(m0);</td>
</tr>
<tr>
<td>V(m1);</td>
<td>V(m1);</td>
</tr>
</tbody>
</table>
Thread Safety

• Can multiple threads call the same function concurrently?

• A function is thread-safe iff it produces correct results when called from multiple concurrent threads.

• 4 classes of thread-unsafe functions:
  1. Do not protect shared variables.
  2. Keep state across multiple invocations.
  3. Return a pointer to a static variable.
  4. Invoke other thread-unsafe functions.
Problem: No protection on shared variables
- Fix: Use \textit{P} and \textit{V} semaphore operations
- Issue: Synchronization operations will slow down code

```c
int cnt = 0;
    sem_t mutex; /*initialized to 1*/
void incr(int n)
{
    P(&mutex);
    for (i = 0; i < n; i++) {
        cnt++;
    }
    V(&mutex);
}
```
Thread-Unsafe Functions (class 2)

- Problem: Keep state across multiple function invocations
  - Example: Random number generator that uses static state

```c
static unsigned int next = 1;

/*! rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/*! srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```
Thread-Safe Random Number Generator

- Fix 1: synchronize operations that change static state?
- Fix 2: eliminate static state by passing state as argument

Consequence: programmer using `rand_r` must maintain seed

```c
/* rand_r - return pseudo-random integer on 0..32767 */
int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```
Thread-Unsafe Functions (class 3)

• Problem: return a pointer to a static variable
• Fix 1: Make caller pass address of variable to store result
• Fix 2. Lock-and-copy

```c
/* lock-and-copy version */
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```
Thread-Unsafe Functions (class 4)

• Invoke other thread-unsafe functions
  – Calling one thread-unsafe function makes the entire function that calls it thread-unsafe

  – Fix: Modify the function so it calls only thread-safe functions 😊
Reentrant Functions

- Def: A function is **reentrant** iff it accesses no shared variables.
  - Important subset of thread-safe functions.
    - Require no synchronization operations.
    - Only way to make a class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

<table>
<thead>
<tr>
<th>All functions</th>
<th>Thread-safe functions</th>
<th>Thread-unsafe functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reentrant functions</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread-Safe Library Functions

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe
  - Examples: malloc, free, printf, scanf

- Most system calls are thread-safe, with a few exceptions:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>3</td>
<td>asctime_r</td>
</tr>
<tr>
<td>ctime</td>
<td>3</td>
<td>ctime_r</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>3</td>
<td>gethostbyaddr_r</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>3</td>
<td>gethostbyname_r</td>
</tr>
<tr>
<td>inet_ntoa</td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime</td>
<td>3</td>
<td>localtime_r</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>rand_r</td>
</tr>
</tbody>
</table>
Conclusions

• Threads are a popular mechanism for writing concurrent programs
  – Cheaper than processes
  – Easy to share data between multiple threads
• However, the ease of sharing has a cost:
  – Easy to introduce subtle synchronization errors (races, deadlocks etc.)
  – Tread carefully with threads!
• 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 provide one solution for enforcing mutual exclusion.