Synchronization

15-213: Introduction to Computer Systems
Recitation 14: November 25, 2013

Pratik Shah (pcshah)
Section C
Topics

- News
- Shared State
- Race conditions
- Synchronization
  - Mutex
  - Semaphore
  - Readers-writers lock
- Producer-Consumer Problem
- Which locks to use when?
- Proxy lab: Cache
News

- Proxylab due on Thursday, Dec 5
- Last date to handin is Sunday, Dec 8
- Autograder available in the handout directory.
Shared state and Critical Section
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
Mapping Variable Instances to Memory

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])

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);
}

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
Race condition

- You might have experienced race conditions in shell lab!

- A race occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y.
  - Access to shared variables and data structures
  - Threads dependent on a condition
Unsafe multi-threading

```c
#include "csapp.h"

static volatile int global = 0;

int main(void) {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d", global);
    return 0;
}

void *thread(void *vargp) {
    int i;
    for (i = 0; i < 100; i++) {
        global++;
    }
    return NULL;
}
```

Output:
??
Race condition - Example

- global++;

- Think of it as:
  1. Load value of global into register
  2. Add one to register
  3. Store new value in address of global

- We don't want threads to interleave
  - 1-2-3-1-2-3

- But they might...
  - 1-2-1-2-3-3
Unsafe multi-threading

```c
#include "csapp.h"

static volatile int global = 0;

int main(void) {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d", global);
    return 0;
}

void *thread(void *vargp) {
    int i;
    for (i = 0; i < 100; i++) {
        global++;
    }
    return NULL;
}
```

Output:
Can print any integer from 2 to 200!

Shared variable:
global

Critical Section
global++
Synchronization

- Need to synchronize threads so that any critical region has at most one thread in it

- Ways to do synchronization:
  1. **Semaphore**
     - Restrictions the number of threads that can access a shared resource
  2. **Mutex**
     - Special case of semaphore that restricts access to one thread
  3. **Reader/Writer locks**
     - Multiple readers allowed
     - Single writer allowed
     - No readers allowed when writer is present
Semaphore

- Classic solution: Dijkstra's P and V operations on semaphores

- Semaphore: non-negative integer synchronization variable.
  - P(s): [ while (s == 0) wait(); s--; ]
  - V(s): [ s++; ]
  - OS guarantees that operations between brackets [ ] are executed indivisibly.
  - Only one P or V operation at a time can modify s.
  - Semaphore invariant: (s >= 0)
  - Initialize s to the number of simultaneous threads allowed
Reader/Writer locks

- Many concurrent readers
- Only one writer
- Good for data-structures that are read often
  - Like caches!
  - Search lists
    - Many readers can lookup concurrently
    - Only one writer must update the list

- Ask for either “read” or “write” permission, and the lock will wake you up when it's your turn.
- Solutions that favor reader or favor writer starve the other.
  - Eg. Favoring reader will allow readers inside critical section as long as there is at least one reader inside, even if writer is waiting.
Solution : Favors Reader

Readers:

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

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);
        /* Reading happens here */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);
        /* Writing here */
        V(&w);
    }
}
```

rw1.c
POSIX synchronization functions

- Semaphores
  - `sem_init`
  - `sem_wait`
  - `sem_post`

- Mutex
  - `pthread_mutex_init`
  - `pthread_mutex_lock`
  - `pthread_mutex_unlock`

- Read-write locks
  - `pthread_rwlock_init`
  - `pthread_rwlock_rdlock`
  - `pthread_rwlock_wrlock`
Unsafe multi-threading

```c
#include "csapp.h"

static volatile int global = 0;
static sem_t sem;

int main(void) {
    pthread_t tid1, tid2;
    sem_init(&sem, 0, 1);
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("%d", global);
    return 0;
}

void *thread(void *vargp) {
    int i;
    for (i = 0; i < 100; i++) {
        sem_wait(&sem);
        global++;
        sem_post(&sem);
    }
    return NULL;
}
```

Output:
Always prints 200
Producer-Consumer Problem

- **Classic computer science problem**
  - Often in day-to-day life as well, hence the name ;-) 

- **Common synchronization pattern:**
  - Producer waits for empty *slot*, inserts item in buffer, and notifies consumer
  - Consumer waits for *item*, removes it from buffer, and notifies producer

- **Example: Keyboard – scanf/print to display**
  - Producer: Keyboard. Keys pressed *produce* characters
  - Consumer: scanf/readline func. Reads or *consumes* those characters.
  - Shared buffer: Keyboard buffer. A finite size buffer
Producer-Consumer on 1-element Buffer

Initially: empty==1, full==0

Producer Thread

```c
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Produce item */
        item = i;
        printf("produced %d\n", item);

        /* Write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }

    return NULL;
}
```

Consumer Thread

```c
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* Read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* Consume item */
        printf("consumed %d\n", item);
    }

    return NULL;
}
```
Producer-Consumer on an $n$-element Buffer

- Requires a mutex and two counting semaphores:
  - mutex: enforces mutually exclusive access to the shared buffer
  - slots: counts the available slots in the buffer
  - items: counts the available items in the buffer

- Above primitives used to control “concurrent and safe” access to shared buffer.

- Sample Implementation in csapp.h : sbuf
#include "csapp.h"

typedef struct {
    int *buf;        /* Buffer array */
    int n;           /* Maximum number of slots */
    int front;       /* buf[(front+1)%n] is first item */
    int rear;        /* buf[rear%n] is last item */
    sem_t mutex;     /* Protects accesses to buf */
    sem_t slots;     /* Counts available slots */
    sem_t items;     /* Counts available items */
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
Producer-Consumer : n-element buffer

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

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

- What **exactly** is shared between threads?
- What type of shared state **access** is desired?
  - Mutually exclusive. Only one thread must access it.
    - **Mutex**. Example: global count.
  - More than one thread can access it concurrently or more than one instance of shared resource available:
    - **Semaphore**. Example: Multiple free slots in shared buffer
    - Multiple threads can read concurrently, only one must write at a time.
    - **Readers-writers lock**: Example: Lookup from global list, read-write from web-proxy Cache.
- What is the overhead of using particular type of locks?
  - Locking is expensive as threads must wait to acquire it.
  - How many threads (all vs few) wait and how frequent?
- Lock for **entire** resource vs Locks for **parts** of it vs **Both**
Proxy lab: Cache

- Proxy is concurrent. Cache must also be concurrent
  - Multiple threads should be able to read from it
  - Else it will be bottleneck.
- LRU policy requires updating some metadata for every access
  - E.g. LRU counter, timestamp etc
- Is reading from cache a “pure” read access?
- Good news: Strict LRU-policy rule relaxed 😊
- Design cache to allow maximum concurrency with minimum overhead.
- Ordering of acquiring and releasing locks important if multiple locks are needed.
Questions?