CSE320 System Fundamentals II

Semaphores I

YOUNGMIN KWON / TONY MIONE

Shared Variables

One of the merits of using threads is sharing variables between threads

- A variable is shared iff multiple threads reference an instance of the variable
- Sharing variables is tricky

```c
#include <pthread.h>
#include <stdio.h>
char **ptr;                             // .bss
void *thread(void *vargp) {
    long myid = (long)vargp;            // stack
    static int cnt = 1;                 // .data
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}

int main() {
    long i;
    pthread_t tid;
    char *msg[2] = {"hello", "world"};

    ptr = msg;     // copy a local msg var to a global ptr var
    for(i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void*)i);
    pthread_exit(NULL);
}
```

```
$ ./a.out
[0]: hello (cnt=2)
[1]: world (cnt=3)
```

Thread Memory Model

Each thread has its own thread context including

- Thread ID, Stack, Stack pointer, program counter, condition codes (flags), general-purpose registers

Each thread shares the rest of the process context with other threads

- Entire user virtual address space, (code, read/write data, heap, shared library code and data)
- Set of open files

Thread Memory Model

Registers are never shared

Virtual memory is always shared

Each thread has its own stack, but stack space is not protected

```c
#include <pthread.h>
#include <stdio.h>

void *thread(void *vargp) {
    int a;
    printf("%ld: &a = %p\n", pthread_self(), &a);
    pthread_exit(NULL);
    return NULL;
}

int main() {
    pthread_t tid;
    int i = 0;
    for(i = 0; i < 10; i++)
        pthread_create(&tid,
                       NULL,
                       thread,
                       NULL);
    pthread_exit(NULL);
    return 0;
}
```

```
$ ./a.out
140318187357952: &a = 0x7f9e5fb67edc
140318153787136: &a = 0x7f9e5db63edc
140318178965248: &a = 0x7f9e5f366edc
140318170572544: &a = 0x7f9e5eb65edc
140318145394432: &a = 0x7f9e5d362edc
140318137001728: &a = 0x7f9e5cb61edc
140318162179840: &a = 0x7f9e5e364edc
140318128609024: &a = 0x7f9e5c360edc
140318120216320: &a = 0x7f9e5bb5fedc
140318111823616: &a = 0x7f9e5b35eedc
```

Synchronizing Threads

Shared variables are convenient

Can introduce synchronization errors

```c
#include <pthread.h>
#include <stdio.h>

volatile long cnt = 0;

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

int main() {
    pthread_t tid1, tid2;
    long n = 100000;

    pthread_create(&tid1, NULL, thread, &n);
    pthread_create(&tid2, NULL, thread, &n);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("cnt = %ld\n", cnt);
    return 0;
}
```

Synchronizing Threads

C code for thread i

```
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

H_i: instructions at the head of the loop
L_i: load the variable cnt into %rdx
U_i: update (increment) %rdx
S_i: store updated %rdx into cnt
T_i: instructions at the tail of the loop

Synchronizing Threads

Correct Ordering

Step	Thread	Instr	$\%rdx_1$	$\%rdx_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	1	S_1	1	—	1
5	2	H_2	—	—	1
6	2	L_2	—	1	1
7	2	U_2	—	2	1
8	2	S_2	—	2	2
9	2	T_2	—	2	2
10	1	T_1	1	—	2

Synchronizing Threads

Incorrect Ordering

Step	Thread	Instr	%rdx$_1$	%rdx$_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	1	U_1	1	—	0
4	2	H_2	—	—	0
5	2	L_2	—	0	0
6	1	S_1	1	—	1
7	1	T_1	1	—	1
8	2	U_2	—	1	1
9	2	S_2	—	1	1
10	2	T_2	—	1	1

Semaphores

To synchronize different executions of threads

A semaphore, s, is a nonnegative global variable that can only be manipulated by P and V operations

- P(s): if s > 0, then decrement s and return;
  if s = 0, suspend the thread until s is increased by V from other thread. After restarting decrement s and return
- V(s): increase s by 1. If any threads are blocked by P, V restarts exactly one of these threads

Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *s,     // Semaphore
             int pshared, // 0: between threads;
                          // otherwise, between processes
             unsigned int value); // initial value

int sem_destroy(sem_t *s); //destroy the semaphore s

int sem_wait(sem_t *s);    // P(s)

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

Semaphores

Binary semaphore

- To provide mutual exclusion (mutex)
- Semaphore has values 0 and 1
- P operation is called locking
- V operation is called unlocking

Counting semaphore

- Used as a counter for a set of available resources

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
sem_t mutex;
volatile long count = 0;
void* thread(void *vargp) {
    long i, n = *((long*)vargp);
    for(i = 0; i < n; i++)
        sem_wait(&mutex),
                 count++,
                 sem_post(&mutex);
    return NULL;
}
```

```c
int main() {
    pthread_t tid1, tid2;
    long n = 100000;

    sem_init(&mutex, 0/*pshared*/, 1/*value*/);
    pthread_create(&tid1, NULL, thread, &n);
    pthread_create(&tid2, NULL, thread, &n);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    sem_destroy(&mutex);

    printf("count = %ld\n", count);
    return 0;
}
```

Semaphores

Counting Semaphores

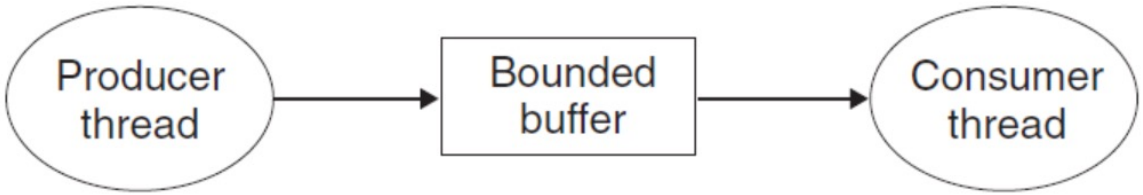
- To schedule shared resources
- Notify other threads that some program states have changed

- Producer-consumer problem
- Readers-writers problem

Producer-Consumer Problem

A producer and a consumer thread share a bounded buffer with n slots

- The producer creates items and adds them to the buffer
- The consumer removes items from the buffer and consumes (uses) them



Producer-Consumer Problem

Need a mutual exclusion to access the shared buffer

Need to schedule the access to the buffer

- If the buffer is full, the producer needs to wait
- If the buffer is empty, the consumer needs to wait

Producer-Consumer Example

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct {
    int *buf;
    int capacity, head, tail;
    sem_t mutex;
    sem_t slots;
    sem_t items;
} sbuf_t;
```

Producer-Consumer Example

```
void sbuf_init(sbuf_t* sp, int n) {
    sp->buf = (int*) calloc(n, sizeof(int));
    sp->capacity = n;
    sp->head = sp->tail = 0;
    sem_init(&sp->mutex, 0, 1);
    sem_init(&sp->slots, 0, n);
    sem_init(&sp->items, 0, 0);
}
void sbuf_deinit(sbuf_t *sp) {
    free(sp->buf);
    sem_destroy(&sp->mutex);
    sem_destroy(&sp->slots); sem_destroy(&sp->items);
}
```

Producer-Consumer Example

```
int sbuf_size(sbuf_t *sp) {
    sem_wait(&sp->mutex); // access buffer after acquiring the lock
    int n = (sp->head + sp->capacity - sp->tail) % sp->capacity;
    sem_post(&sp->mutex);
    return n;
}
```

Producer-Consumer Example

```
void sbuf_insert(sbuf_t *sp, int item) {
    sem_wait(&sp->slots); // wait while the buffer is full
    sem_wait(&sp->mutex); // access buffer after acquiring the lock
    sp->head = (sp->head + 1) % sp->capacity;
    sp->buf[sp->head] = item;
    sem_post(&sp->mutex);
    sem_post(&sp->items); // wake up consumer if it is suspended
}

int sbuf_remove(sbuf_t *sp) {
    sem_wait(&sp->items); // wait while the buffer is empty
    sem_wait(&sp->mutex); // access buffer after acquiring the lock
    sp->tail= (sp->tail + 1) % sp->capacity;
    int item = sp->buf[sp->tail];
    sem_post(&sp->mutex);
    sem_post(&sp->slots); // wake up producer if it is suspended
    return item;
}
```

Producer-Consumer Example

```
void* producer(void* vargp) {
    sbuf_t *sp = (sbuf_t*)vargp;
    int i, j;
    for(i = 0; i < 100; i++) {
        long s = 0;
        for(j = 0; j < 10000; j++)
            s += j, sbuf_insert(sp, j);
        printf("producer: sum: %ld, size: %d\n", s, sbuf_size(sp));
    }
    pthread_exit(NULL);
}
```

Producer-Consumer Example

```
void* consumer(void* vargp) {
    sbuf_t *sp = (sbuf_t*)vargp;
    int i, j;
    for(i = 0; i < 100; i++) {
        long s = 0;
        for(j = 0; j < 10000; j++)
            s += sbuf_remove(sp);
        printf("consumer: sum: %ld, size: %d\n", s, sbuf_size(sp));
    }
    pthread_exit(NULL);
}
```

Producer-Consumer Example

```
int main() {
    pthread_t tid_p, tid_c;
    sbuf_t sb;
    sbuf_init(&sb, 15000);

    pthread_create(&tid_p, NULL, producer, &sb);
    pthread_create(&tid_c, NULL, consumer, &sb);

    pthread_join(tid_p, NULL);
    pthread_join(tid_c, NULL);

    sbuf_deinit(&sb);
    return 0;
}
```

Readers-Writers Problem

A collection of concurrent threads access a shared object

- Reader: threads that only read the data
- Writer: threads that only modify the data

First readers-writers problem (favors readers)

- No readers keep waiting unless a writer has already been granted permission to update the object

Second readers-writers problem (favors writers)

- Once a writer is ready to write, it performs its operation as soon as possible.
- A reader that arrives after a writer must wait, even if the writer is already waiting

Readers-Writers Example

```
// First Readers-Writers problem
//
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

typedef struct {
    sem_t mutex;
    sem_t wlock;
    int readcount;
} rwlock;

typedef struct {
    int data;
    int copy;
    rwlock lock;
} object;
```

```
void rwlock_init(rwlock *lock) {
    sem_init(&lock->mutex, 0, 1);
    sem_init(&lock->wlock, 0, 1);
    lock->readcount = 0;
}

void rwlock_deinit(rwlock *lock) {
    sem_destroy(&lock->mutex);
    sem_destroy(&lock->wlock);
}
```

Readers-Writers Example

```
void acquire_reader_lock(rwlock *lock) {
    sem_wait(&lock->mutex);
    lock->readcount++;
    if(lock->readcount == 1)   //if this is the first reader
        sem_wait(&lock->wlock);//wait for a writer to finish or block writers
    sem_post(&lock->mutex);
}

void release_reader_lock(rwlock *lock) {
    sem_wait(&lock->mutex);
    lock->readcount--;
    if(lock->readcount == 0)   //if this is the last reader
        sem_post(&lock->wlock);//unblock any waiting writers
    sem_post(&lock->mutex);
}
```

Readers-Writers Example

```
void acquire_writer_lock(rwlock *lock) {
    sem_wait(&lock->wlock);
}

void release_writer_lock(rwlock *lock) {
    sem_post(&lock->wlock);
}
```

Readers-Writers Example

```
void* reader(void *vargp) {
    object* pobj = (object*)vargp;
    int i;
    for(i = 0; i < 10000; i++) {
        acquire_reader_lock(&pobj->lock);
        int data = pobj->data;
        int copy = pobj->copy;
        release_reader_lock(&pobj->lock);
        printf("R_%d: data: %d, copy: %d\n", i, data, copy);
    }
}
```

Readers-Writers Example

```
void* writer(void *vargp) {
    object* pobj = (object*)vargp;
    int i;
    for(i = 0; i < 10000; i++) {
        acquire_writer_lock(&pobj->lock);
        int data = pobj->data = i % 10;
        int copy = pobj->copy = pobj->data;
        release_writer_lock(&pobj->lock);
        printf("W_%d: data: %d, copy: %d\n", i, data, copy);
    }
}
```

Readers-Writers Example

```
int main() {
    pthread_t tid[3];
    object obj;
    obj.data = obj.copy = 0;
    rwlock_init(&obj.lock);

    pthread_create(tid+0, 0, reader, &obj);
    pthread_create(tid+1, 0, reader, &obj);
    pthread_create(tid+2, 0, writer, &obj);

    pthread_join(tid[0]);
    pthread_join(tid[1]);
    pthread_join(tid[2]);

    rwlock_deinit(&obj.lock);
}
```

Questions?