Synchronization: Advanced

15-213 / 18-213: Introduction to Computer Systems
25th Lecture, April 23, 2012

Instructors:
Seth Copen Goldstein, Anthony Rowe, and Greg Kesden

One Worry: Races

- A race occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```c
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}
/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Race Elimination

- Make sure don’t have unintended sharing of state

```c
/* a threaded program without the race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = malloc(sizeof(int));
        *valp = i;
        Pthread_create(&tid[i], NULL, thread, valp);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}
/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```
Today

- Races
- Locking and Deadlocks
- Producer-consumer problem
- Readers-writers problem
- Thread safety

Reminder: Semaphores

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

  - Manipulated by \( P \) and \( V \) operations:
    - \( P(s) : \) \[ \text{while} \ (s == 0) \ \text{wait}(); \ s--; \]  
      - 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 \text{while} loop in \( P \) terminates, only that \( P \) can decrement \( s \)

- Semaphore invariant: \( s >= 0 \)

Reminder: Mutual exclusion via Semaphores

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

- **Terminology:**
  - \textit{Binary semaphore}: semaphore whose value is always 0 or 1
  - \textit{Mutex}: binary semaphore used for mutual exclusion
    - \( P \) operation: "locking" the mutex
    - \( V \) operation: "unlocking" or "releasing" the mutex
    - "\textit{Holding}" a mutex: locked and not yet unlocked
  - \textit{Counting semaphore}: used as a counter for set of available resources

A Worry: Deadlock

- **Def**: A process is \textit{deadlocked} \iff it is waiting for a condition that will never be true.

- **Typical Scenario**
  - Processes 1 and 2 needs 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!
Deadlocking With Semaphores

```c
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}

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

Avoiding Deadlock

Acquire shared resources in same order

```c
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}

void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```
Today

- Races
- Deadlocks
- Producer-consumer problem
- Readers-writers problem
- Thread safety

Using Semaphores to Coordinate Access to Shared Resources

- Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true
  - Use counting semaphores to keep track of resource state.
  - Use binary semaphores to notify other threads.

- Two classic examples:
  - The Producer-Consumer Problem
  - The Readers-Writers Problem

Producer-Consumer Problem

- 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

- Examples
  - Multimedia processing:
    - Producer creates MPEG video frames, consumer renders them
  - Event-driven graphical user interfaces
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer
    - Consumer retrieves events from buffer and paints the display

Producer-Consumer on 1-element Buffer

```c
#include "csapp.h"
#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf;  /* shared var */
  sem_t full;  /* sems */
  sem_t empty;
} shared;

int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* Initialize the semaphores */
  Sem_init(&shared.empty, 0, 1);
  Sem_init(&shared.full, 0, 0);

  /* Create threads and wait */
  Pthread_create(&tid_producer, NULL, producer, NULL);
  Pthread_create(&tid_consumer, NULL, consumer, NULL);
  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  exit(0);
}
```
**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;
}
```

**Counting with Semaphores**

- Remember, it's a non-negative integer
  - So, values greater than 1 are legal
- Lets repeat thing_5() 5 times for every 3 of thing_3()

```c
int main() {
    pthread_t tid_five, tid_three;
    sem_t five;  
    Sem_init(&five, 0, 5);
    Sem_init(&three, 0, 3);
    void *five_times(void *arg);
    void *three_times(void *arg);
    /* thing_5 and thing_3 */
    #include "csapp.h"
    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);
    }
    /* thing_3 */
    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;
}
```

**Counting with semaphores (cont)**

Initially: five = 5, three = 3

```c
/* thing_5() thread */
void *five_times(void *arg) {
    int i;
    while (1) {
        for (i=0; i<5; i++) {
            /* Wait & thing_5() */
            P(&five);
            thing_5();
            V(&three);
            V(&three);
            V(&three);
        }
    }
    return NULL;
}
```

```c
/* thing_3() thread */
void *three_times(void *arg) {
    int i;
    while (1) {
        for (i=0; i<3; i++) {
            /* Wait & thing_3() */
            P(&three);
            thing_3();
            V(&five);
            V(&five);
            V(&five);
            V(&five);
        }
    }
    return NULL;
}
```

**Producer-Consumer on an n-element Buffer**

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

- Implemented using a shared buffer package called sbuf.
# sbuf Package - Declarations

```c
#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);
```

# sbuf Package - Implementation

### Initializing and deinitializing a shared buffer:

```c
/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int)); sp->n = n;  /* Buffer holds max of n items */
    sp->front = sp->rear = 0;  /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has zero items */
}

/* Clean up buffer sp */
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

### Inserting an item into a shared buffer:

```c
/* Insert item onto the rear of shared buffer sp */
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 */
}
```

### Removing an item from a shared buffer:

```c
/* Remove and return the first item from buffer sp */
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;
}
```
Sample program using sbuf

```c
void *producer(void *vargp) {
    int cnt = 0;
    while (maxcnt > 0) {
        sbuf_insert(&sbuf, cnt);
        cnt++;
        maxcnt--;
    }
    sbuf_insert(&sbuf, -1);
    pthread_exit(0);
}

void *consumer(void *vargp) {
    int sum = 0;
    while (1) {
        int val = sbuf_remove(&sbuf);
        if (val < 0) break;
        sum += val;
    }
    total = sum;
    pthread_exit(0);
}
```

Is there another way?

- One producer and one consumer

```c
/* Insert item onto the rear of shared buffer sp */
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 */
}

/* Remove and return the first item from buffer sp */
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;
}
```

Do we need locks at all?

```c
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t *sp, int item) {
    P(&sp->slots);                        /* Wait for available slot */
    sp->buf[(++sp->rear)%(sp->n)] = item; /* Insert the item */
    V(&sp->items);                        /* Announce available item */
}

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

Front and rear are not (really) shared!

```c
typedef struct {
    int *buf;
    int n;
    int front;
    int rear;
    int cnt;
} sbuf_t;

void sbuf_init(sbuf_t *sp, int n) {
    sp->n = n;
    sp->buf = calloc(sizeof(int), n);
    sp->front = 0;
    sp->rear = 0;
    sp->cnt = 0;
}
Front and rear are not (really) shared!

```c
void sbuf_insert(sbuf_t* sp, int v) {
    int next = sp->rear+1;
    if (next == sp->n) next = 0;
    while (next == sp->front) 
        pthread_yield();
    sp->buf[sp->rear] = v;
    sp->rear = next;
}

int sbuf_remove(sbuf_t* sp) {
    while (sp->front == sp->rear) 
        pthread_yield();
    int next = sp->front+1;
    if (next == sp->n) next = 0;
    int val = sp->buf[sp->front];
    sp->front = next;
    return val;
}
```

Why does this work for ONLY 1 producer and 1 consumer?

Front and rear are really shared!

```c
void sbuf_insert(sbuf_t* sp, int v) {
    int next = sp->rear+1;
    if (next == sp->n) next = 0;
    while (next == sp->front) 
        pthread_yield();
    sp->buf[sp->rear] = v;
    sp->rear = next;
}

int sbuf_remove(sbuf_t* sp) {
    while (sp->front == sp->rear) 
        pthread_yield();
    int next = sp->front+1;
    if (next == sp->n) next = 0;
    int val = sp->buf[sp->front];
    sp->front = next;
    return val;
}
```

Why does this work for ONLY 1 producer and 1 consumer?
Readers- Writers Problem

- Generalization of the mutual exclusion problem

- Problem statement:
  - Reader threads only read the object
  - Writer threads modify the object
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object

- Occurs frequently in real systems, e.g.,
  - Online airline reservation system
  - Multithreaded caching Web proxy

Variants of Readers-Writers

- First readers-writers problem (favors readers)
  - No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - A reader that arrives after a waiting writer gets priority over the writer.

- Second readers-writers problem (favors writers)
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting.

- Starvation (where a thread waits indefinitely) is possible in both cases.

Today

- Races
- Deadlocks
- Producer-consumer problem
- Readers-writers problem
- Thread safety
Solution to First Readers-Writers Problem

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);
    }
}
```

Today

- Races
- Deadlocks
- Producer-consumer problem
- Readers-writers problem
- Thread safety

Crucial concept: Thread Safety

- Functions called from a thread must be thread-safe

- Def: A function is thread-safe if it will always produce correct results when called repeatedly from multiple concurrent threads.

- Classes of thread-unsafe functions:
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions

Thread-Unsafe Functions (Class 1)

- Failing to protect shared variables
  - Fix: Use P and V semaphore operations
  - Example: goodcnt.c
  - Issue: Synchronization operations will slow down code
Thread-Unsafe Functions (Class 2)

- Relying on persistent state across multiple function invocations
  - Example: Random number generator that relies on 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

- Pass state as part of argument
  - and, thereby, eliminate static state

```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;
}
```

- Consequence: programmer using `rand_r` must maintain seed

Thread-Unsafe Functions (Class 3)

- Returning a pointer to a static variable
- Fix 1. Rewrite function so caller passes address of variable to store result
  - Requires changes in caller and callee
- Fix 2. Lock-and-copy
  - Requires simple changes in caller (and none in callee)
  - However, caller must free memory.

```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;
}
```

Warning: Some functions like `gethostbyname` require a deep copy. Use reentrant `gethostbyname_r` version instead.

Thread-Unsafe Functions (Class 4)

- Calling 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 when called by multiple threads.
  - 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`)

All functions

<table>
<thead>
<tr>
<th>Thread-safe functions</th>
<th>Thread-unsafe functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reentrant functions</strong></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 Unix 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></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>

Threads Summary

- Threads provide another mechanism for writing concurrent programs
- Threads are growing in popularity
  - Somewhat cheaper than processes
  - Easy to share data between threads
- However, the ease of sharing has a cost:
  - Easy to introduce subtle synchronization errors
  - Tread carefully with threads!
- For more info:
  - D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997

Case Study: Prethreaded Concurrent Server

- Diagram showing a pool of worker threads handling client requests, with a master thread coordinating buffer access.
- Client threads connect to the server and service requests through the master thread.
- Key components: client, worker threads, master thread, buffer.
Prethreaded Concurrent Server

```c
int main(int argc, char **argv)
{
    int i, listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;

    port = atoi(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    listenfd = Open_listenfd(port);

    for (i = 0; i < NTHREADS; i++)  /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);

    while (1) {
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
}
```

Worker thread routine:

```c
void *thread(void *vargp)
{
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}
```

```
Worker thread service routine:

```c
void echo_cnt(int connfd)
{
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;
    Pthread_once(&once, init_echo_cnt);
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        P(&mutex);
        byte_cnt += n;
        printf("thread \d received \d (\d total) bytes on fd \d\n",
               (int) pthread_self(), n, byte_cnt, connfd);
        V(&mutex);
        Rio_writen(connfd, buf, n);
    }
}
```

```
```