Synchronization: Advanced

15-213/18-213/14-513/15-513/18-613:
Introduction to Computer Systems
26th Lecture, Nov. 21, 2019
Reminder: Semaphores

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

- Manipulated by \( P \) and \( V \) operations:
  - \( P(s) \): \[ \text{while (s == 0); s--;} \]
    - Dutch for "Proberen" (test)
  - \( V(s) \): \[ s++; \]
    - Dutch for "Verhogen" (increment)

- OS kernel guarantees that operations between brackets [ ] are executed atomically
  - 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: \( (s >= 0) \)
Review: Using semaphores to protect shared resources via mutual exclusion

- Basic idea:
  - Associate a unique semaphore $mutex$, initially 1, with each shared variable (or related set of shared variables)
  - Surround each access to the shared variable(s) with $P(mutex)$ and $V(mutex)$ operations

```
mutex = 1

P(mutex)
cnt++
V(mutex)
```
Review: Using Lock for Mutual Exclusion

Basic idea:

- Mutex is special case of semaphore that only has value 0 (locked) or 1 (unlocked)
- $Lock(m)$: [ while $(m == 0)$; $m=0$; ]
- $Unlock(m)$: [ $m=1$ ]

~2x faster than using semaphore for this purpose
- And, more clearly indicates programmer’s intention

```c
mutex = 1
lock(mutex)
cnt++
unlock(mutex)
```
Review: 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 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
Review: 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.

- **The Producer-Consumer Problem**
  - Mediating interactions between processes that generate information and that then make use of that information
  - Single entry buffer implemented with two binary semaphores
    - One to control access by producer(s)
    - One to control access by consumer(s)
  - N-entry implemented with semaphores + circular buffer
Today

- Using semaphores to schedule shared resources
  - Readers-writers problem

- Other concurrency issues
  - Thread safety
  - Races
  - Deadlocks
  - Interactions between threads and signal handling
Readers-Writers Problem

Problem statement:
- Reader threads only read the object
- Writer threads modify the object (read/write access)
- 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
Readers/Writers Examples
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.**
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);
    }
}
```

Writers:

```c
void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

        V(&w);
    }
}
```

rw1.c
Readers/Writers Examples

\[ w = 0 \]
\[ \text{readcnt} = 0 \]
\[ w = 1 \]
\[ \text{readcnt} = 0 \]
\[ w = 0 \]
\[ \text{readcnt} = 2 \]
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);
    }
}
```

Arrivals: R1 R2 W1 R3
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);
    }
}
```

Arrivals: R1 R2 W1 R3

- Readcnt == 1
- W == 0
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);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0
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);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

rw1.c
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);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0
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 */
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
        /* Reading happens here */
    }
}
```

Readers:

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

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0
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);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0
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);
    }
}
```

Arrivals: R1 R2 W1 R3

Readcnt == 0
W == 1
Other Versions of Readers-Writers

- **Shortcoming of first solution**
  - Continuous stream of readers will block writers indefinitely

- **Second version**
  - Once writer comes along, blocks access to later readers
  - Series of writes could block all reads

- **FIFO implementation**
  - See rwqueue code in code directory
  - Service requests in order received
  - Threads kept in FIFO
  - Each has semaphore that enables its access to critical section
Solution to Second Readers-Writers Problem

```c
int readcnt, writecnt; // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1
void reader(void)
{
    while (1) {
        P(&r);
        P(&rmutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&rmutex);
        V(&r);

        /* Reading happens here */

        P(&rmutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&rmutex);
    }
}
```
Solution to Second Readers-Writers Problem

```c
void writer(void)
{
    while (1) {
        while (1) {
            P(&wmutex);
            writecnt++;
            if (writecnt == 1)
                P(&r);
            V(&wmutex);
            P(&w);
            /* Writing here */
            V(&w);
            P(&wmutex);
            writecnt--;
            if (writecnt == 0);
                V(&r);
                V(&wmutex);
        }
    }
}
```
Managing Readers/Writers with FIFO

**Idea**
- Read & Write requests are inserted into FIFO
- Requests handled as remove from FIFO
  - Read allowed to proceed if currently idle or processing read
  - Write allowed to proceed only when idle
- Requests inform controller when they have completed

**Fairness**
- Guarantee every request is eventually handled
Readers Writers FIFO Implementation

- Full code in rwqueue.{h,c}

```c
/* Queue data structure */
typedef struct {
    sem_t mutex;   // Mutual exclusion
    int reading_count; // Number of active readers
    int writing_count; // Number of active writers
    // FIFO queue implemented as linked list with tail
    rw_token_t *head;
    rw_token_t *tail;
} rw_queue_t;

/* Represents individual thread's position in queue */
typedef struct TOK {
    bool is_reader;
    sem_t enable;   // Enables access
    struct TOK *next; // Allows chaining as linked list
} rw_token_t;
```
Readers Writers FIFO Use

- In rwqueue-test.c

```c
/* Get write access to data and write */
void iwriter(int *buf, int v)
{
    rw_token_t tok;
    rw_queue_request_write(&q, &tok);
    /* Critical section */
    *buf = v;
    /* End of Critical Section */
    rw_queue_release(&q);
}

/* Get read access to data and read */
int ireader(int *buf)
{
    rw_token_t tok;
    rw_queue_request_read(&q, &tok);
    /* Critical section */
    int v = *buf;
    /* End of Critical section */
    rw_queue_release(&q);
    return v;
}
```
Library Reader/Writer Lock

- **Data type** `pthread_rwlock_t`

- **Operations**
  - Acquire read lock
    ```c
    Pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
    ```
  - Acquire write lock
    ```c
    Pthread_rwlock_wrlock(pthread_rwlock_t *rwlock)
    ```
  - Release (either) lock
    ```c
    Pthread_rwlock_unlock(pthread_rwlock_t *rwlock)
    ```

- **Observation**
  - Library must be used correctly!
    - Up to programmer to decide what requires read access and what requires write access
Today

- Using semaphores to schedule shared resources
  - Readers-writers problem
- Other concurrency issues
  - Races
  - Deadlocks
  - Thread safety
  - Interactions between threads and signal handling
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(int argc, char** argv) {
    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);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

race.c
Data Race
Race Elimination

- Don’t share state
  - E.g., use malloc to generate separate copy of argument for each thread

- Use synchronization primitives to control access to shared state
Today

- Using semaphores to schedule shared resources
  - Producer-consumer problem

- Other concurrency issues
  - Races
  - Deadlocks
  - Thread safety
  - Interactions between threads and signal handling
A Worry: Deadlock

- **Def:** A process is *deadlocked* iff 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!
Deadlocking With Semaphores

```c
int main(int argc, char** argv)
{
    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);
    return 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;
}
```
Deadlock Visualized in Progress Graph

Locking introduces the potential for deadlock: waiting for a condition that will never be true.

Any trajectory that enters the deadlock region will eventually reach the deadlock state, waiting for either $S_0$ or $S_1$ to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate fact: deadlock is often nondeterministic (race).

$S_0 = S_1 = 1$
Deadlock
Avoiding Deadlock

Acquire shared resources in same order

```c
int main(int argc, char** argv)
{
    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);
    return 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;
}
```

Tid[0]:
- P(s₀);
- P(s₁);
- cnt++;
- V(s₀);
- V(s₁);

Tid[1]:
- P(s₀);
- P(s₁);
- cnt++;
- V(s₁);
- V(s₀);
Avoided Deadlock in Progress Graph

- Thread 0
  - $P(s_0)$
  - $V(s_0)$
  - $P(s_1)$
  - $V(s_1)$

- Thread 1
  - $V(s_0)$
  - $V(s_1)$
  - $P(s_0)$

Forbidden region for $s_0$

Forbidden region for $s_1$

- $s_0 = s_1 = 1$

No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial
Demonstration

- See program deadlock.c
- 100 threads, each acquiring same two locks
- Risky mode
  - Even numbered threads request locks in opposite order of odd-numbered ones
- Safe mode
  - All threads acquire locks in same order
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/10968
Today

- **Using semaphores to schedule shared resources**
  - Readers-writers problem

- **Other concurrency issues**
  - Races
  - Deadlocks
  - Thread safety
  - Interactions between threads and signal handling
Crucial concept: Thread Safety

Functions called from a thread must be thread-safe.

Def: A function is thread-safe iff 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 (or mutex)
  - 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
  - Requires changes in caller and callee

- 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
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    sprintf(buf, "%d", x);
    return buf;
}

char *lc_itoa(int x, char *dest)
{
    P(&mutex);
    strcpy(dest, itoa(x));
    V(&mutex);
    return dest;
}
```
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`)
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><code>asctime</code></td>
<td>3</td>
<td><code>asctime_r</code></td>
</tr>
<tr>
<td><code>ctime</code></td>
<td>3</td>
<td><code>ctime_r</code></td>
</tr>
<tr>
<td><code>gethostbyaddr</code></td>
<td>3</td>
<td><code>gethostbyaddr_r</code></td>
</tr>
<tr>
<td><code>gethostbyname</code></td>
<td>3</td>
<td><code>gethostbyname_r</code></td>
</tr>
<tr>
<td><code>inet_ntoa</code></td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td><code>localtime</code></td>
<td>3</td>
<td><code>localtime_r</code></td>
</tr>
<tr>
<td><code>rand</code></td>
<td>2</td>
<td><code>rand_r</code></td>
</tr>
</tbody>
</table>
Today

- Using semaphores to schedule shared resources
  - Readers-writers problem

- Other concurrency issues
  - Races
  - Deadlocks
  - Thread safety
  - Interactions between threads and signal handling
Signal Handling Review

**Action**

- Signal can occur at any point in program execution
  - Unless signal is blocked
- Signal handler runs within same thread
- Must run to completion and then return to regular program execution
Threads / Signals Interactions

- Many library functions use lock-and-copy for thread safety
  - Because they have hidden state
  - malloc
    - Free lists
  - fprintf, printf, puts
    - So that outputs from multiple threads don’t interleave
  - sprintf
    - Not officially asynch-signal-safe, but seems to be OK
- OK for handler that doesn’t use these library functions
Bad Thread / Signal Interactions

- What if:
  - Signal received while library function holds lock
  - Handler calls same (or related) library function

- Deadlock!
  - Signal handler cannot proceed until it gets lock
  - Main program cannot proceed until handler completes

- Key Point
  - Threads employ symmetric concurrency
  - Signal handling is asymmetric
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