Synchronization: Advanced

CS351: Systems Programming
Day 27: Nov. 29, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Review: Semaphores

- **Semaphore**: non-negative global integer synchronization variable. Manipulated by $P$ and $V$ operations.

- $P(s)$
  - If $s$ is nonzero, then decrement $s$ by 1 and return immediately.
  - If $s$ is zero, then suspend thread until $s$ becomes nonzero and the thread is restarted by a $V$ operation.
  - After restarting, the $P$ operation decrements $s$ and returns control to the caller.

- $V(s)$:
  - Increment $s$ by 1.
  - If there are any threads blocked in a $P$ operation waiting for $s$ to become non-zero, then restart exactly one of those threads, which then completes its $P$ operation by decrementing $s$.

- **Semaphore invariant**: $(s \geq 0)$
Review: Using semaphores to protect shared resources via mutual exclusion

Basic idea:

- Associate a unique semaphore \textit{mutex}, initially 1, with each shared variable (or related set of shared variables)
- Surround each access to the shared variable(s) with \textit{P(mutex)} and \textit{V(mutex)} operations

\begin{verbatim}
mutex = 1
P(mutex)
cnt++
V(mutex)
\end{verbatim}
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 and to notify other threads
  - Use mutex to protect access to resource

- **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 an $n$-element Buffer

- Requires a mutex and two counting semaphores:
  - `mutex`: enforces mutually exclusive access to 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);
```

/* 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 0 items */
}

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

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;
}
```
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**
Solution to First Readers-Writers Problem

Readers:

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

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

        /* Critical section */
        /* Reading happens */
        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

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

Readers:  Writers:  rw1.c
Putting It All Together: Prethreaded Concurrent Server

- Main thread
  - Accept connections
  - Insert descriptors
- Buffer
  - Remove descriptors
- Worker thread
- Pool of worker threads
- Service client
- Client
- Worker thread
  - Service client
  - Remove descriptors
Prethreaded Concurrent Server

```c
sbuf_t sbuf; /* Shared buffer of connected descriptors */

int main(int argc, char **argv)
{
    int i, listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    for (i = 0; i < NTHREADS; i++) /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
}
```

- echoserverpreter_pre.c
Worker thread routine:

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

Prethreaded Concurrent Server
Prethreaded Concurrent Server

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);
    }
}
```

Worker thread service routine: `echo_cnt.c`
Prethreaded Concurrent Server

**echo_cnt initialization routine:**

```c
static int byte_cnt;    /* Byte counter */
static sem_t mutex;    /* and the mutex that protects it */

static void init_echo_cnt(void)
{
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
```

*echo_cnt.c*
Real-world example

**Making Break-ups Less Painful: Source-level Support for Transforming Legacy Software into a Network of Tasks**

Nik Sultana  
University of Pennsylvania

Achala Rao  
University of Pennsylvania

Zihao Jin  
Tsinghua University

Pardis Pashakhanloo  
University of Pennsylvania

Henry Zhu  
University of Pennsylvania

Ke Zhong  
Shanghai Jiao Tong University

Boon Thau Loo  
University of Pennsylvania

**ABSTRACT**

"Breaking up" software into a dataflow network of tasks can improve availability and performance by exploiting the flexibility of the resulting graph, more granular resource use, hardware concurrency and modern interconnects. Decomposing legacy systems in this manner is difficult and ad hoc however, raising such challenges as weaker consistency and potential data races. Thus it is difficult to build on battle-tested legacy systems.

We propose a paradigm and supporting tools for developers to recognize task-level modularity opportunities in software. We use the Apache web server as an example of legacy software to test our ideas. This is a stepping stone towards realizing a vision where automated decision-support tools assist in the decomposition of systems to improve the reuse of components, meet performance targets or exploit new hardware devices and topologies.

**CCS CONCEPTS**
- Computer systems organization → Maintainability and maintainance
- Software and its engineering → Extra-functional properties, Software post-development issues

**KEYWORDS**
program analysis; program transformation; distributed systems

**ACM Reference Format:**

---

- **Apache pre-threaded webserver “MPM”**

- **Specializing threads, forming a pipeline, client partitioning.**

---

**Figure 1:** Transforming the subroutine call/return control-flow paradigm to one based on enqueue/dequeue over channels between threads. We must analyze programs to ensure that sufficient context is passed from one thread to the next, that the transformation will not produce name clashes or type errors, and to avoid introducing data races between threads (e.g., if one
Another use example

https://gitlab.com/niksu/hashtray

---

### About

`libhashtray` provides an implementation of cuckoo hashing, and can provide wrappers to use third-party hash tables using the same interface.

The latter is useful for applications that want to use one or more of these hashtable implementations simultaneously.

### Version

1.0

### Downloading

[gitlab](https://gitlab.com/niksu/hashtray)

### Building

Running `make headers` and `make libhashtray.a` generates the outputs for development and linking.

The included tests and example code is compiled using `make tests`. Specific tests can be compiled using the appropriate target, and an extensive debug mode can be used by prepending a flag, e.g., `DEBUGGING=1 make hashtray_multiprocess`.

### Using
Another use example

```c
struct idxs {
    HASHTRAY(key_t) idx[CHOICES];
};

static HASHTRAY(key_t) alt_idx(HASHTRAY(key_t) idx, HASHTRAY(key_t) key) {
    static struct idxs idxs_of_DATA_TYPE(HASHTRAY(data_t) data);

    struct entry {
        bool clear;
        HASHTRAY(key_t) key;
        HASHTRAY(value_t) value;
    };

    struct cell {
        struct entry entry[NUM_CELL_ENTRIES];
    };

    struct HASHTRAY(table) {
        struct cell cell[TABLE_SIZE];
        #ifdef MULTITHREADED
            pthread_mutex_t lock[TABLE_SIZE];
        #endif // MULTITHREADED
        #ifdef MULTIPROCESS
            sem_t * lock[TABLE_SIZE];
        #endif // MULTIPROCESS
    };

    #ifdef REMEMBER_LOSS
        struct overflow_t {
            struct entry entry[NUM_OVERFLOW_ENTRIES];
        } overflow;
    #endif

    #ifdef HASHTRAY_ASSUME
        struct HASHTRAY_ASSUME table {
            struct cell cell[TABLE_SIZE];
            #ifdef MULTITHREADED
                pthread_mutex_t lock[TABLE_SIZE];
            #endif // MULTITHREADED
            #ifdef MULTIPROCESS
                sem_t * lock[TABLE_SIZE];
            #endif // MULTIPROCESS
            #ifdef REMEMBER_LOSS
                struct overflow_t overflow;
            #endif
        };
    #endif

    assert((int)result.idx[i] >= 0);
    assert((int)result.idx[i] < TABLE_SIZE);
}
#endif // HASHTRAY_ASSUME
return result;
}

static inline void
unlock_index(struct HASHTRAY(table) * t, int table_idx) {
    int error;
    #if !defined(MULTITHREADED) && !defined(MULTIPROCESS)
        // Do nothing
    #elif defined(MULTITHREADED) && defined(MULTIPROCESS)
        #error Simultaneous MULTITHREADED and MULTIPROCESS not supported.
    #elif defined(MULTITHREADED)
        error = pthread_mutex_unlock(&t->lock[table_idx]);
    #ifdef HASHTRAY_ASSUME
        assert(!error); // FIXME check when !HASHTRAY_ASSUME
    #endif // HASHTRAY_ASSUME
    #elif defined(MULTIPROCESS)
        error = sem_post(t->lock[table_idx]);
    #ifdef HASHTRAY_ASSUME
        assert(!error); // FIXME check when !HASHTRAY_ASSUME
    #endif // HASHTRAY_ASSUME
    #endif
}

#end

static inline void
lock_index(struct HASHTRAY(table) * t, int table_idx) {
    int error;
    #if !defined(MULTITHREADED) && !defined(MULTIPROCESS)
        // Do nothing
    #elif defined(MULTITHREADED) && defined(MULTIPROCESS)
        #error Simultaneous MULTITHREADED and MULTIPROCESS not supported.
    #elif defined(MULTITHREADED)
        error = pthread_mutex_lock(&t->lock[table_idx]);
    #ifdef HASHTRAY_ASSUME
        assert(!error); // FIXME check when !HASHTRAY_ASSUME
    #endif // HASHTRAY_ASSUME
    #elif defined(MULTIPROCESS)
        error = sem_wait(t->lock[table_idx]);
    #ifdef HASHTRAY_ASSUME
        assert(!error); // FIXME check when !HASHTRAY_ASSUME
    #endif // HASHTRAY_ASSUME
    #endif
}

#endif // HASHTRAY_ASSUME
```

https://gitlab.com/niksu/hashtray
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
  - 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 global 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;
}
```
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>Reentrant functions</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><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>
Summary

- Concurrency provides more flexibility and resource utilization.
  - *Prethreading*: creating pools of threads to lower start-up overhead.
- But it is difficult to reason about concurrent logic flows.
- We use synchronization to manage access to shared resources.
- *Critical sections* of code access and use these resources.
- *Semaphores*: provide abstraction for synchronization. Can be used for *mutual exclusion*.
- Risks:
  - Races
  - Deadlocks
- Thread-safety and re-entrancy – likely to be encountered in other courses.
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- The form is anonymous.
  (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- [https://forms.gle/qoeEbBuTYXo5FiU1A](https://forms.gle/qoeEbBuTYXo5FiU1A)
- I’ll remind about this at each lecture.
Course Evaluation Survey

- Course-level evaluation (vs lecture-level)
- Your feedback is important!
- The survey is anonymous.
- You’ll receive an email with the survey link.
Extra slides
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;
}
```

N threads are sharing \( i \)
Race Illustration

```c
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
```

Race between increment of `i` in main thread and deref of `vargp` in peer thread:
- If deref happens while `i = 0`, then OK
- Otherwise, peer thread gets wrong id value
Could this race really occur?

Main thread

```c
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL, thread,&i);
}
```

Peer thread

```c
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    int i = *((int *)vargp);
    save_value(i);
    return NULL;
}
```

Race Test

- If no race, then each thread would get different value of `i`
- Set of saved values would consist of one copy each of 0 through 99
Experimental Results

No Race

Single core laptop

Multicore server

The race can really happen!
Race Elimination

/* Threaded program without the race */

int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    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;
}

Avoid unintended sharing of state
Another 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()
{
    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;
}
```

Tests:

- Tid[0]: P(s0); P(s1); cnt++; V(s0); V(s1);
- Tid[1]: P(s1); P(s0); cnt++; V(s1); V(s0);
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).
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;
}
```
Avoided Deadlock in Progress Graph

No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Forbidden region for $s_0$

Forbidden region for $s_1$