Lecture 25: “Concurrent Programming”

John P. Shen & Zhiyi Yu
November 30, 2016

Required Reading Assignment:
• Chapter 12 of CS:APP (3rd edition) by Randy Bryant & Dave O’Hallaron.
A. Concurrency Approaches
- Process Based
- Event Based
- Thread Based

B. Thread Synchronization
- Semaphores (P & V operations)
- Producer-Consumer Problem
- Readers-Writers Problem
- Thread Safety, Races, Deadlocks
Concurrent Programming is Hard!

- Classical problem classes of concurrent programs:
  - **Races**: outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - **Deadlock**: improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - **Livelock / Starvation / Fairness**: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line

- Many aspects of concurrent programming are beyond the scope of our course..
  - but, not all 😊
  - We’ll cover some of these aspects in the next two lectures.
Iterative Servers

Iterative servers process one request at a time

Client 1
- connect
- write
- call read
- ret read
- close

Server
- accept
- read
- write
- read
- close

Client 2
- connect
- write
- call read
- ret read

Wait for server to finish with Client 1
Where Does Second Client Block?

- **Second client attempts to connect to iterative server**

  - Call to connect returns
    - Even though connection not yet accepted
    - Server side TCP manager queues request
    - Feature known as “TCP listen backlog”

  - Call to `rio_writen` returns
    - Server side TCP manager buffers input data

  - Call to `rio_readlineb` blocks
    - Server hasn’t written anything for it to read yet.
Fundamental Flaw of Iterative Servers

Solution: use *concurrent servers* instead

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.
Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

1. Process-based
   - Kernel automatically interleaves multiple logical flows
   - Each flow has its own private address space

2. Event-based
   - Programmer manually interleaves multiple logical flows
   - All flows share the same address space
   - Uses technique called *I/O multiplexing*.

3. Thread-based
   - Kernel automatically interleaves multiple logical flows
   - Each flow shares the same address space
   - Hybrid of process-based and event-based.
Approach #1: Process-based Servers

- Spawn separate process for each client

```
client 1
  call connect
  call fgets

User goes out to lunch

Client 1 blocks waiting for user to type in data

child 1
  call read

Child blocks waiting for data from Client 1

fork

server
  call accept
  ret accept

fork

child 2
  call fgets
  write
  call read
  ret read
  close

write
  call read
  close

client 2
  call connect
```

Process-Based Concurrent Echo Server

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);   /* Child services client */
            Close(connfd);  /* Child closes connection with client */
            exit(0);        /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

echoserverp.c
Process-Based Concurrent Echo Server (cont)

- Reap all zombie children

```c
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

echoserverp.c
Concurrent Server: accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling `connect`

3. Server returns `connfd` from `accept`. Forks child to handle client. Connection is now established between `clientfd` and `connfd`
Process-based Server Execution Model

- Each client handled by independent child process
- No shared state between them
- Both parent & child have copies of listenfd and connfd
  - Parent must close connfd
  - Child should close listenfd
Issues with Process-based Servers

- Listening server process must reap zombie children
  - to avoid fatal memory leak
- Parent process must close its copy of connfd
  - Kernel keeps reference count for each socket/open file
  - After fork, \( \text{refcnt}(\text{connfd}) = 2 \)
  - Connection will not be closed until \( \text{refcnt}(\text{connfd}) = 0 \)
Pros and Cons of Process-based Servers

- + Handle multiple connections concurrently
- + Clean sharing model
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- + Simple and straightforward
- – Additional overhead for process control
- – Nontrivial to share data between processes
  - Requires IPC (interprocess communication) mechanisms
    - FIFO’s (named pipes), System V shared memory and semaphores
Approach #2: Event-based Servers

- Server maintains set of active connections
  - Array of connfd’s

- Repeat:
  - Determine which descriptors (connfd’s or listenfd) have pending inputs
    - e.g., using `select` or `epoll` functions
    - arrival of pending input is an event
  - If listenfd has input, then accept connection
    - and add new connfd to array
  - Service all connfd’s with pending inputs

- Details for select-based server in book
I/O Multiplexed Event Processing

Active Descriptors

<table>
<thead>
<tr>
<th>connfd's</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>12</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Pending Inputs

<table>
<thead>
<tr>
<th>connfd's</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>-1</td>
</tr>
</tbody>
</table>

Active Descriptors

<table>
<thead>
<tr>
<th>connfd's</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>listenfd = 3</td>
</tr>
</tbody>
</table>

Pending Inputs

<table>
<thead>
<tr>
<th>connfd's</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>listenfd = 3</td>
</tr>
</tbody>
</table>
Pros and Cons of Event-based Servers

- One logical control flow and address space.
- Can single-step with a debugger.
- No process or thread control overhead.
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado

- Significantly more complex to code than process- or thread-based designs.
- Hard to provide fine-grained concurrency
  - E.g., how to deal with partial HTTP request headers
- Cannot take advantage of multi-core
  - Single thread of control
Approach #3: Thread-based Servers

- Very similar to approach #1 (process-based)
  - ...but using threads instead of processes
Traditional View of a Process

- **Process** = process context + code, data, and stack
Alternate View of a Process

- Process = thread + code, data, and kernel context

Thread (main thread)
- Stack
- Thread context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

Code, data, and kernel context
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data
- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer
A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)

Thread 1 (main thread)  Thread 2 (peer thread)

- stack 1
- stack 2

Thread 1 context:
  - Data registers
  - Condition codes
  - SP1
  - PC1

Thread 2 context:
  - Data registers
  - Condition codes
  - SP2
  - PC2

Shared code and data

- shared libraries
- run-time heap
- read/write data
- read-only code/data

Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer
Logical View of Threads

- Threads associated with process form a pool of peers
  - Unlike processes which form a tree hierarchy
Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time.
- Otherwise, they are sequential.

Examples:
- Concurrent: A & B, A&C
- Sequential: B & C

```
Thread A     Thread B     Thread C
Time
```

11/30/2016 (©J.P. Shen & Zhiyi Yu)
Concurrent Thread Execution

- **Single Core Processor**
  - Simulate parallelism by time slicing

- **Multi-Core Processor**
  - Can have true parallelism

Run 3 threads on 2 cores
**Threads vs. Processes**

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched

- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread
Posix Threads (Pthreads) Interface

- **Pthreads**: Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads], `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`
The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
  pthread_t tid;
  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

void *thread(void *vargp) /* thread routine */
{
  printf("Hello, world!\n");
  return NULL;
}
```

Thread ID

Thread attributes (usually NULL)

Thread routine

Thread arguments (void *p)

Return value (void **p)
Execution of Threaded “hello, world”

Main thread

call Pthread_create()
Pthread_create() returns

call Pthread_join()

Main thread waits for peer thread to terminate

Pthread_join() returns

exit()
Terminates main thread and any peer threads

Peer thread

printf()
return NULL;
Peer thread terminates
Thread-Based Concurrent Echo Server

```c
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                          (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

echoserver.c

- `malloc` of connected descriptor necessary to avoid deadly race (later)
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}

- Run thread in “detached” mode.
  - Runs independently of other threads
  - Reaped automatically (by kernel) when it terminates
- Free storage allocated to hold connfd.
- Close connfd (important!)
Thread-based Server Execution Model

- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables
Issues With Thread-Based Servers

- **Must run “detached” to avoid memory leak**
  - At any point in time, a thread is either *joinable* or *detached*
    - *Joinable* thread can be reaped and killed by other threads
      - must be reaped (with `pthread_join`) to free memory resources
    - *Detached* thread cannot be reaped or killed by other threads
      - resources are automatically reaped on termination
  - Default state is joinable
    - use `pthread_detach(pthread_self())` to make detached

- **Must be careful to avoid unintended sharing**
  - For example, passing pointer to main thread’s stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`

- **All functions called by a thread must be thread-safe**
**Pros and Cons of Thread-Based Designs**

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**

- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!
Summary: Approaches to Concurrency

- **Process-based**
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients

- **Event-based**
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core

- **Thread-based**
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable
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”

- Def: A variable $x$ is shared if and only if multiple threads reference some instance of $x$.

- 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?
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*
Example Program to Illustrate Sharing

```c
char **ptr; /* global var */

int main()
{
    long 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);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread’s stack indirectly through global ptr variable

sharing.c
Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - Each thread stack contains one instance of each local variable

- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - Virtual memory contains exactly one instance of any local static variable.
Mapping Variable Instances to Memory

**Global var:** 1 instance (ptr [data])

**Local vars:** 1 instance (i.m, msgs.m)

```c
char **ptr; /* global var */

int main()
{
    long 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);
}
```

**Local var:** 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);

    return NULL;
}
```

**Local static var:** 1 instance (cnt [data])

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);

    return NULL;
}
```

Carnegie Mellon University
Shared Variable Analysis

- Which variables are shared?

<table>
<thead>
<tr>
<th>Variable</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
Synchronizing Threads

- Shared variables are handy...

- ...but introduce the possibility of nasty synchronization errors.
badcnt.c: Improper Synchronization

/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    pthread_create(&tid1, NULL, thread, &niters);
    pthread_create(&tid2, NULL, thread, &niters);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *(long *)vargp;

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}

linux> ./badcnt 10000
OK cnt=20000

linux> ./badcnt 10000
BOOM! cnt=13051

What went wrong?

cnt should equal 20,000.
Assembly Code for Counter Loop

C code for counter loop in thread i

```c
for (i = 0; i < niters; i++)
    cnt++;
```

**Asm code for thread i**

```
.L2:
    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
    H_i : Head
.L1:
    L_i : Load cnt
    U_i : Update cnt
    S_i : Store cnt
    T_i : Tail
```
**Concurrent Execution**

- **Key idea:** In general, any sequentially consistent interleaving is possible, but some give an unexpected result!
  - $i_i$ denotes that thread $i$ executes instruction $i$
  - $\%rdx_i$ is the content of $\%rdx$ in thread $i$’s context

<table>
<thead>
<tr>
<th>$i$ (thread)</th>
<th>instr$_i$</th>
<th>$%rdx_1$</th>
<th>$%rdx_2$</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$H_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$L_1$</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$U_1$</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$S_1$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$H_2$</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$L_2$</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$U_2$</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$S_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$T_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>$T_1$</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

Thread 1 critical section

Thread 2 critical section

OK
Concurrent Execution (cont)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;2&lt;/sub&gt;</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H&lt;sub&gt;1&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Oops!
Concurrent Execution (cont)

- How about this ordering?

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr&lt;sub&gt;i&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;1&lt;/sub&gt;</th>
<th>%rdx&lt;sub&gt;2&lt;/sub&gt;</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>H&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L&lt;sub&gt;2&lt;/sub&gt;</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S&lt;sub&gt;2&lt;/sub&gt;</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T&lt;sub&gt;2&lt;/sub&gt;</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Oops!

- We can analyze the behavior using a progress graph
A progress graph depicts the discrete execution state space of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible execution state \((\text{Inst}_1, \text{Inst}_2)\).

E.g., \((L_1, S_2)\) denotes state where thread 1 has completed \(L_1\) and thread 2 has completed \(S_2\).
A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2
Critical Sections and Unsafe Regions

L, U, and S form a critical section with respect to the shared variable \( \text{cnt} \).

Instructions in critical sections (wrt some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form unsafe regions.
Def: A trajectory is safe iff it does not enter any unsafe region

Claim: A trajectory is correct (wrt cnt) iff it is safe
Enforcing Mutual Exclusion

- **Question:** How can we guarantee a safe trajectory?

- **Answer:** We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.
  - i.e., need to guarantee *mutually exclusive access* for each critical section.

- **Classic solution:**
  - Semaphores (Edsger Dijkstra)

- **Other approaches (out of our scope):**
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)
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.
    - Test and decrement operations occur atomically (indivisibly)
  - 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.
    - Increment operation occurs atomically
  - 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 >= 0)
C Semaphore Operations

Pthreads functions:

```c
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

CS:APP wrapper functions:

```c
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```
badcnt.c: Improper Synchronization

/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
pthread_t tid1, tid2;

    niters = atoi(argv[1]);
Pthread_create(&tid1, NULL, thread, &niters);
Pthread_create(&tid2, NULL, thread, &niters);
Pthread_join(tid1, NULL);
Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
exit(0);
}

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

How can we fix this using semaphores?
Using Semaphores for Mutual Exclusion

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

- **Terminology:**
  - *Binary semaphore*: semaphore whose value is always 0 or 1
  - *Mutex*: binary semaphore used for mutual exclusion
    - P operation: “locking” the mutex
    - V operation: “unlocking” or “releasing” the mutex
    - “Holding” a mutex: locked and not yet unlocked.
  - *Counting semaphore*: used as a counter for set of available resources.
**goodcnt.c: Proper Synchronization**

- Define and initialize a mutex for the shared variable `cnt`:

  ```c
  volatile long cnt = 0; /* Counter */
  sem_t mutex; /* Semaphore that protects cnt */
  
  Sem_init(&mutex, 0, 1); /* mutex = 1 */
  ```

- **Surround** critical section with `P` and `V`:

  ```c
  for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
  }
  ```

Warning: It’s orders of magnitude slower than `badcnt.c`. 

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
```
Why Mutexes Work

Provide mutually exclusive access to shared variable by surrounding critical section with \( P \) and \( V \) operations on semaphore \( s \) (initially set to 1)

Semaphore invariant creates a forbidden region that encloses unsafe region and that cannot be entered by any trajectory.
Summary

- Programmers need a clear model of how variables are shared by threads.

- Variables shared by multiple threads must be protected to ensure mutually exclusive access.

- Semaphores are a fundamental mechanism for enforcing mutual exclusion.
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:
  - \texttt{mutex}: enforces mutually exclusive access to the buffer
  - \texttt{slots}: counts the available slots in the buffer
  - \texttt{items}: counts the available items in the buffer

- Implemented using a shared buffer package called \texttt{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);

sbuf.h
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 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 */
}
```

`sbuf.c`
sbuf Package - Implementation

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;
}
```

sbuf.c
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);
    }
}
```

rw1.c
Putting It All Together: Prethreaded Concurrent Server

Client → Master thread (Accept connections) → Buffer → Worker thread (Insert descriptors)
Worker thread (Remove descriptors) → Service client
Pool of worker threads

Client → Worker thread (Service client)

...
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_listenedfd(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 */
    }
}
```

echoserver_pre.c
Prethreaded Concurrent Server

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);
    }
}
```

`echoserververt_pre.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
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);
    }
}
```

echo_cnt.c
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 😊
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>
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.c**

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

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

norace.c
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 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**

```
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;
}
```

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

Tid[1]:
- P(s₁);
- P(s₀);
- cnt++;
- V(s₁);
- V(s₀);
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

\( s_0 = s_1 = 1 \)
Lecture 26: “Thread Level Parallelism”

John P. Shen & Zhiyi Yu
December 5, 2016

Required Reading Assignment:
• Chapter 12 of CS:APP (3rd edition) by Randy Bryant & Dave O’Hallaron.