

COMP2310/COMP6310

Systems, Networks, & Concurrency

Convener: Prof John Taylor

Synchronization: Advanced

Acknowledgement of material: With changes suited to ANU needs, the slides are obtained from Carnegie Mellon University: <https://www.cs.cmu.edu/~213/>

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 *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)
```

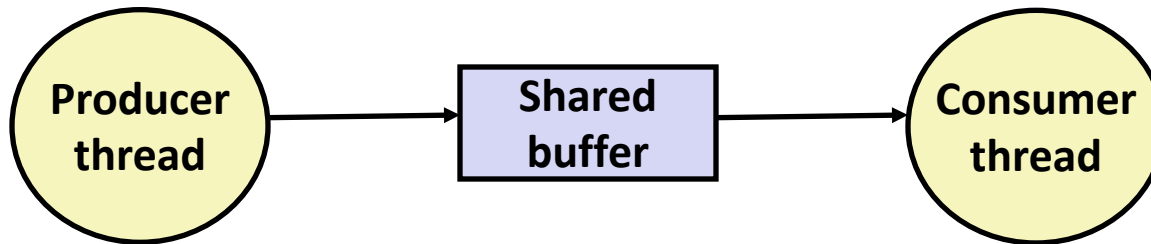
```
cnt++
```

```
V(mutex)
```

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 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

```
#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:

```
/* 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.c

sbuf Package - Implementation

Inserting an item into a shared buffer:

```
/* 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:

```
/* 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:

```
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:

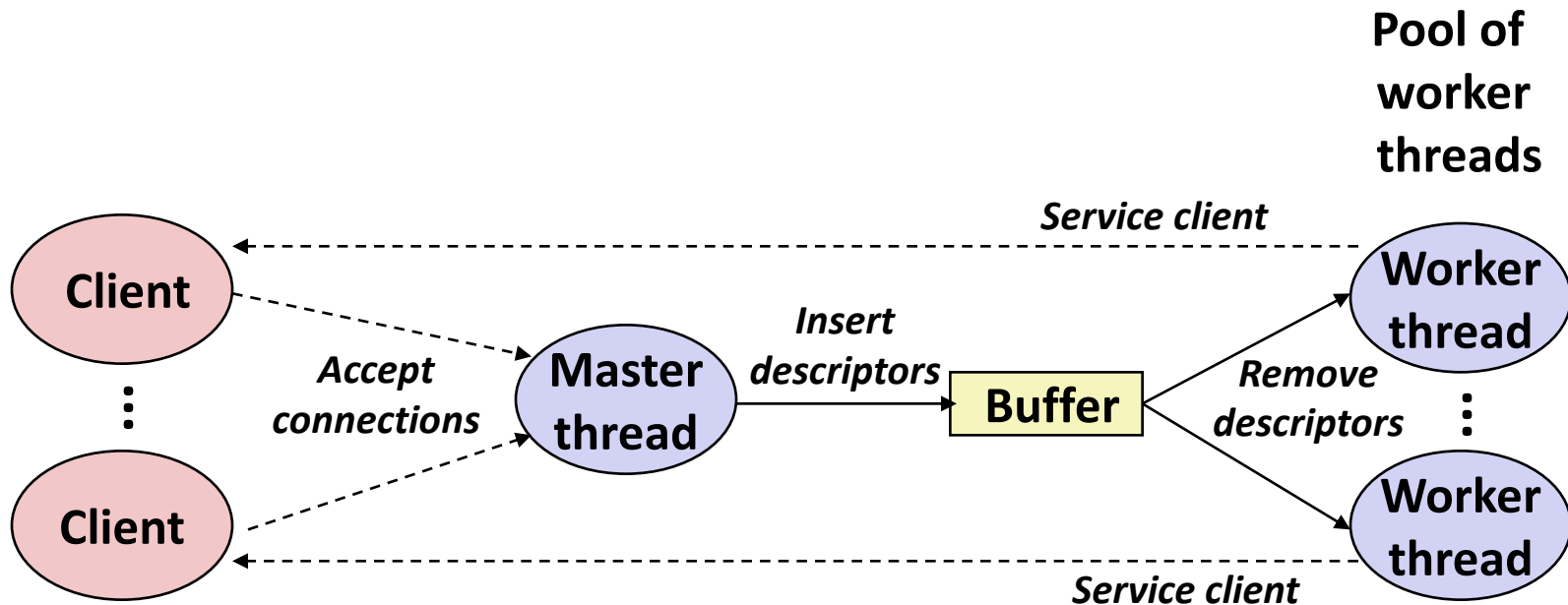
```
void writer(void)
{
    while (1) {
        P(&w);

        /* Critical section */
        /* Writing happens */

        V(&w);
    }
}
```

rw1.c

Putting It All Together: Prethreaded Concurrent Server



Prethreaded Concurrent Server

```
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 */
    }
}
```

echoservt_pre.c

Prethreaded Concurrent Server

Worker thread routine:

```
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);
    }
}
```

echoservt_pre.c

Prethreaded Concurrent Server

echo_cnt initialization routine:

```
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

Prethreaded Concurrent Server

Worker thread service routine:

```
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

```
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

```
/* 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.

```
/* 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)

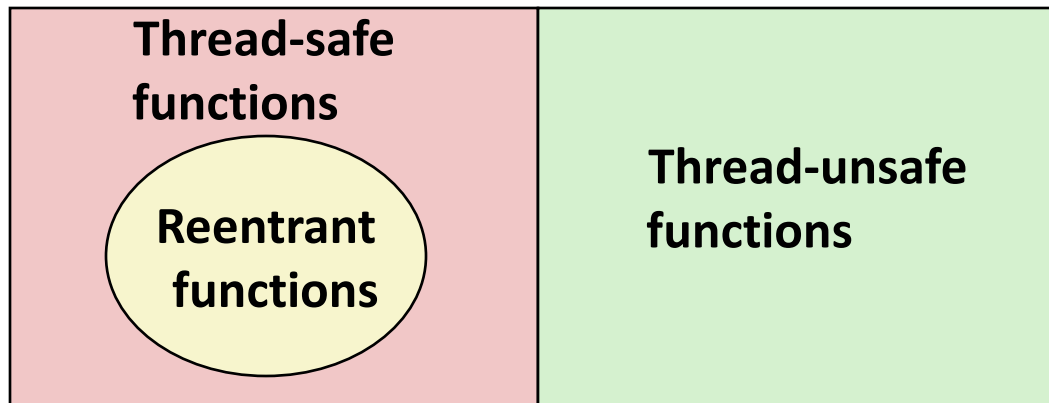
■ Calling thread-unsafe functions

- 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



Thread-Safe Library Functions

- All functions in the Standard C Library (at the back of the K&R C text) are thread-safe
 - Examples: `malloc`, `free`, `printf`, `scanf`
- Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function	Class	Reentrant version
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(none)
<code>localtime</code>	3	<code>localtime_r</code>
<code>rand</code>	2	<code>rand_r</code>

One worry: Races

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

```
/* 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);
}

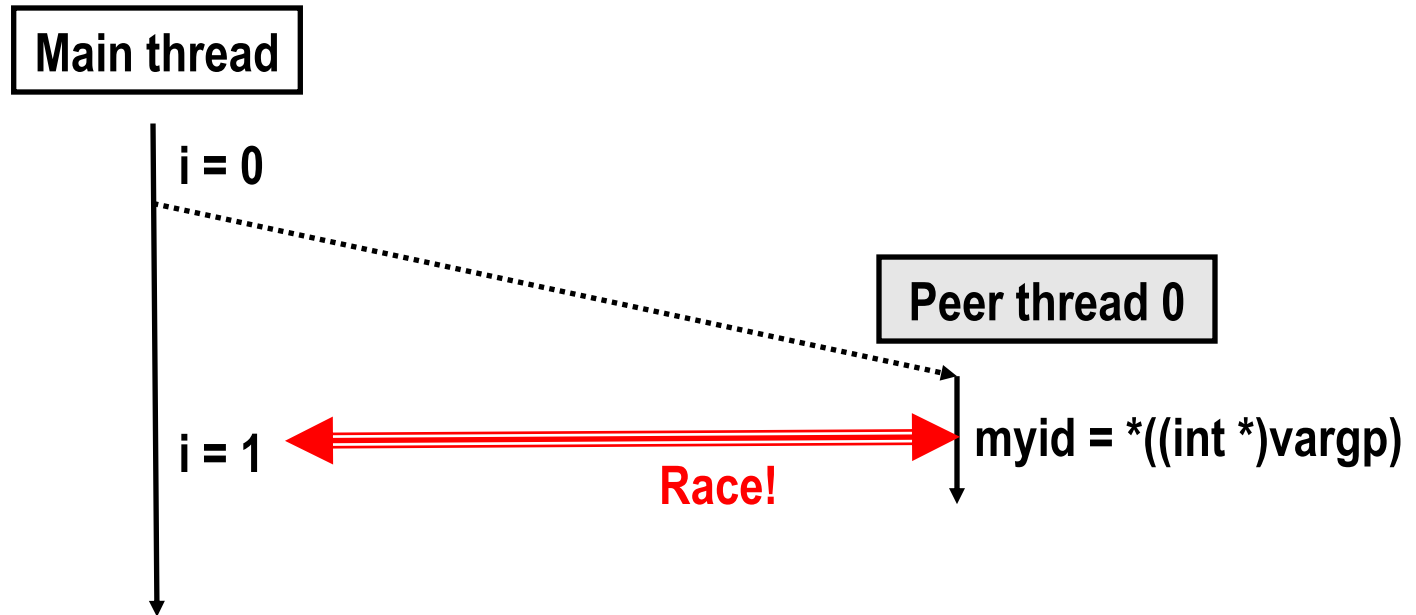
/* Thread routine */
void *thread(void *vargp)
{
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

N threads are sharing i

race.c

Race Illustration

```
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

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

Peer thread

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

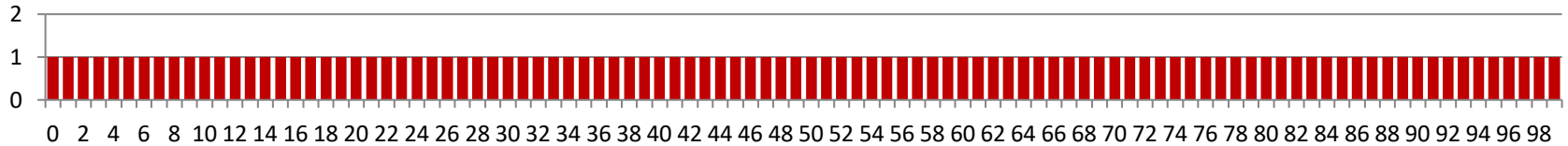
race.c

■ Race Test

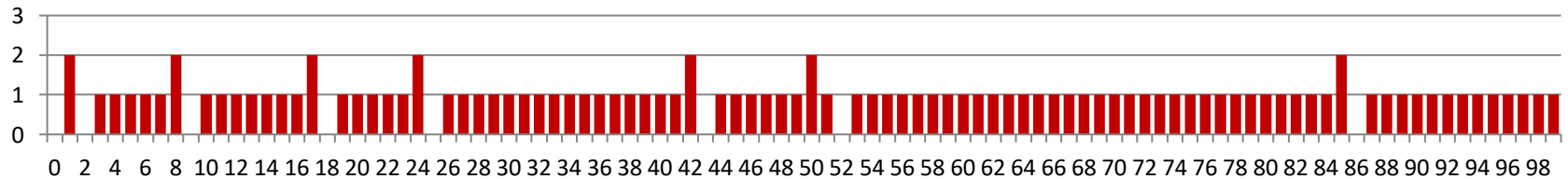
- If no race, then each thread would get different value of *i*
- Set of saved values would consist of one copy each of 0 through 99

Experimental Results

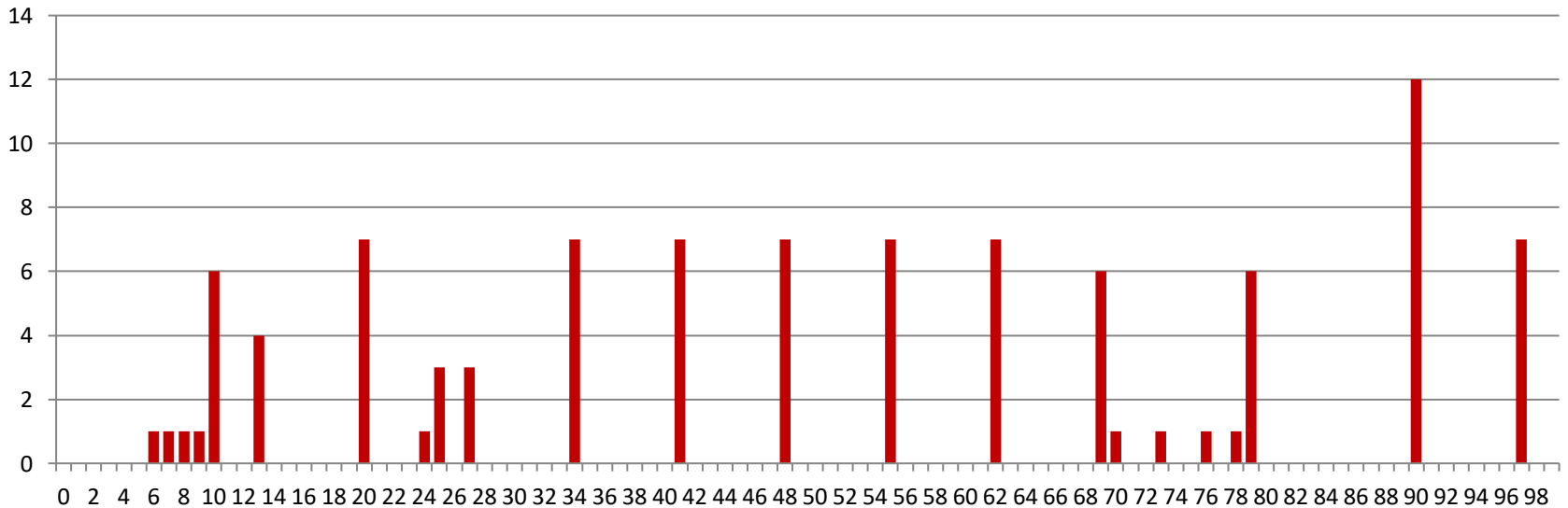
No Race



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 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

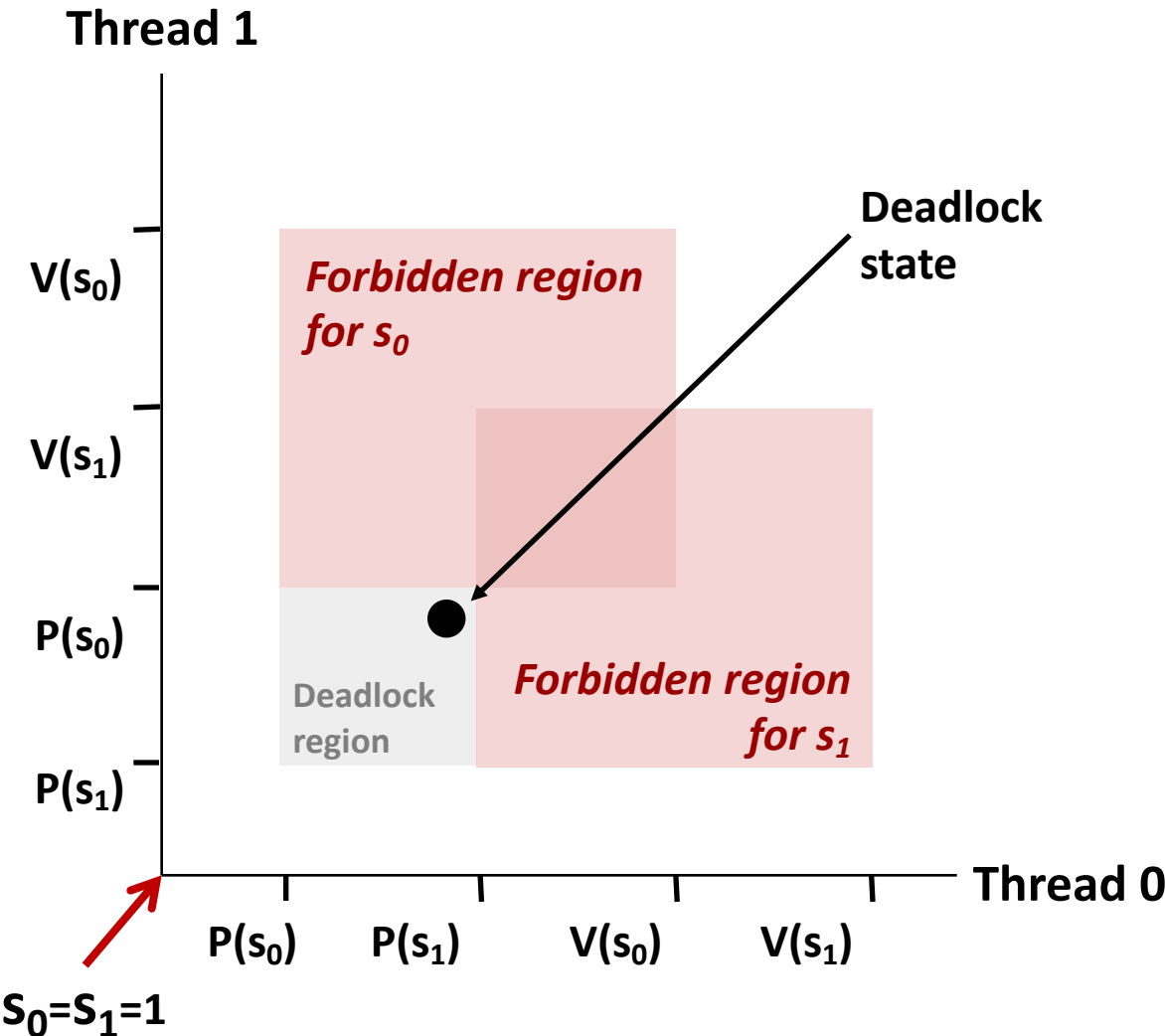
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], 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);

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

```
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);
}
```

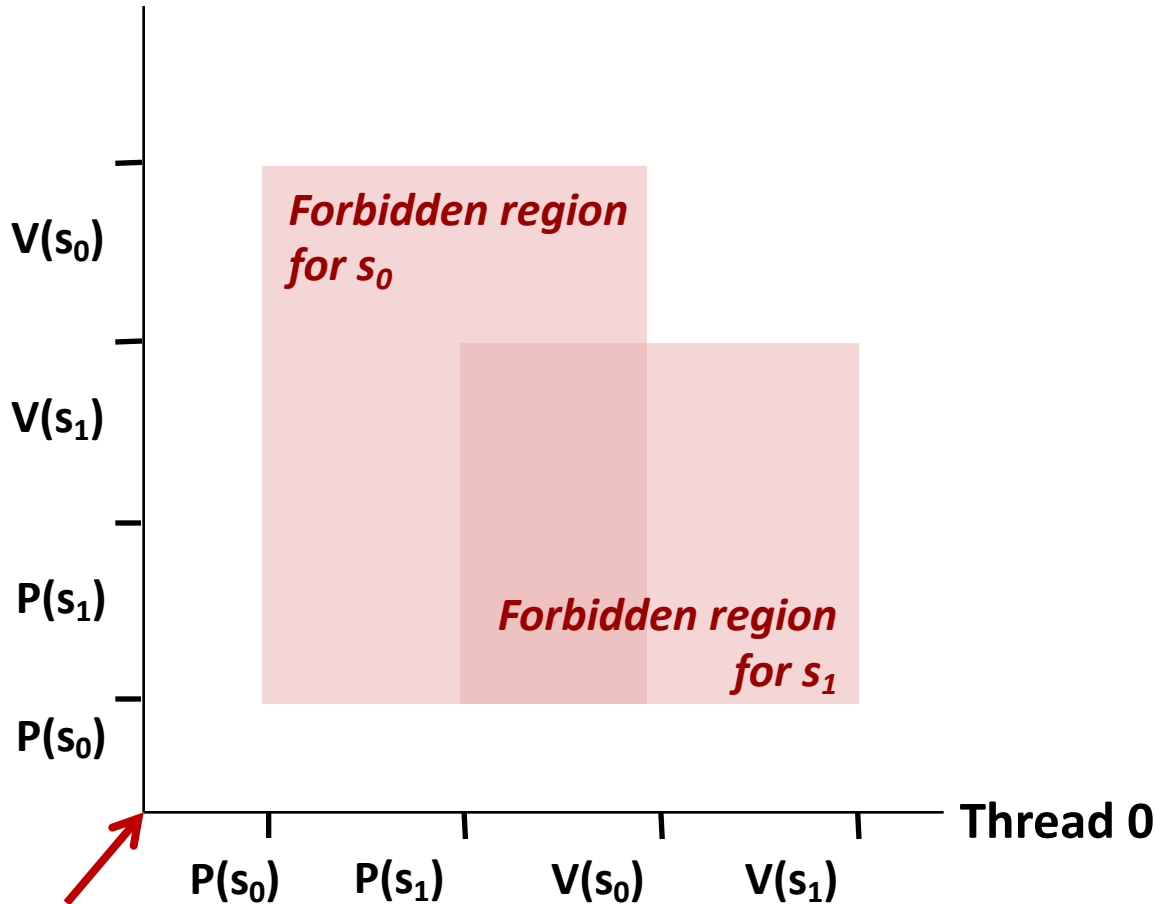
```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

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 1



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

$S_0 = S_1 = 1$