CS 4400

Computer Systems

LECTURE 24

Semaphores

Concurrency issues

```c
/* badcnt.c */
#include "csapp.h"

#define NITERS 200000000
void* count(void* arg);

unsigned int cnt = 0;     /* shared counter variable */

int main() {
  pthread_t tid1, tid2;

  Pthread_create(&tid1, NULL, count, NULL);
  Pthread_create(&tid2, NULL, count, NULL);
  Pthread_join(tid1, NULL);
  Pthread_join(tid2, NULL);

  if(cnt != (unsigned)NITERS*2)
    printf("BOOM! cnt=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}

/* thread routine */
void* count(void* arg) {
  int i;
  for(i = 0; i < NITERS; i++)
    cnt++;
  return NULL;
}
```

```
unix> ./badcnt
BOOM! ctr=278125352

unix> ./badcnt
BOOM! ctr=271726247

unix> ./badcnt
BOOM! ctr=276537330
```

Example: Shared Variable `cnt`

C code for thread i

```
for (i=0; i<NITERS; i++)
    ctr++;
```

Asm code for thread i

```
.L9:
    movl -4(%ebp),%eax
    cmpl $99999999,%eax
    jle .L12
    jmp .L10
------------------------------
.L12:
    movl ctr,%eax
    leal 1(%eax),%edx
    movl %edx,ctr
------------------------------
.L11:
    movl -4(%ebp),%eax
    leal 1(%eax),%edx
    movl %edx,-4(%ebp)
    jmp .L9
.L10:
```

H_i : Head

L_i : Load `ctr`

U_i : Update `ctr`

S_i : Store `ctr`

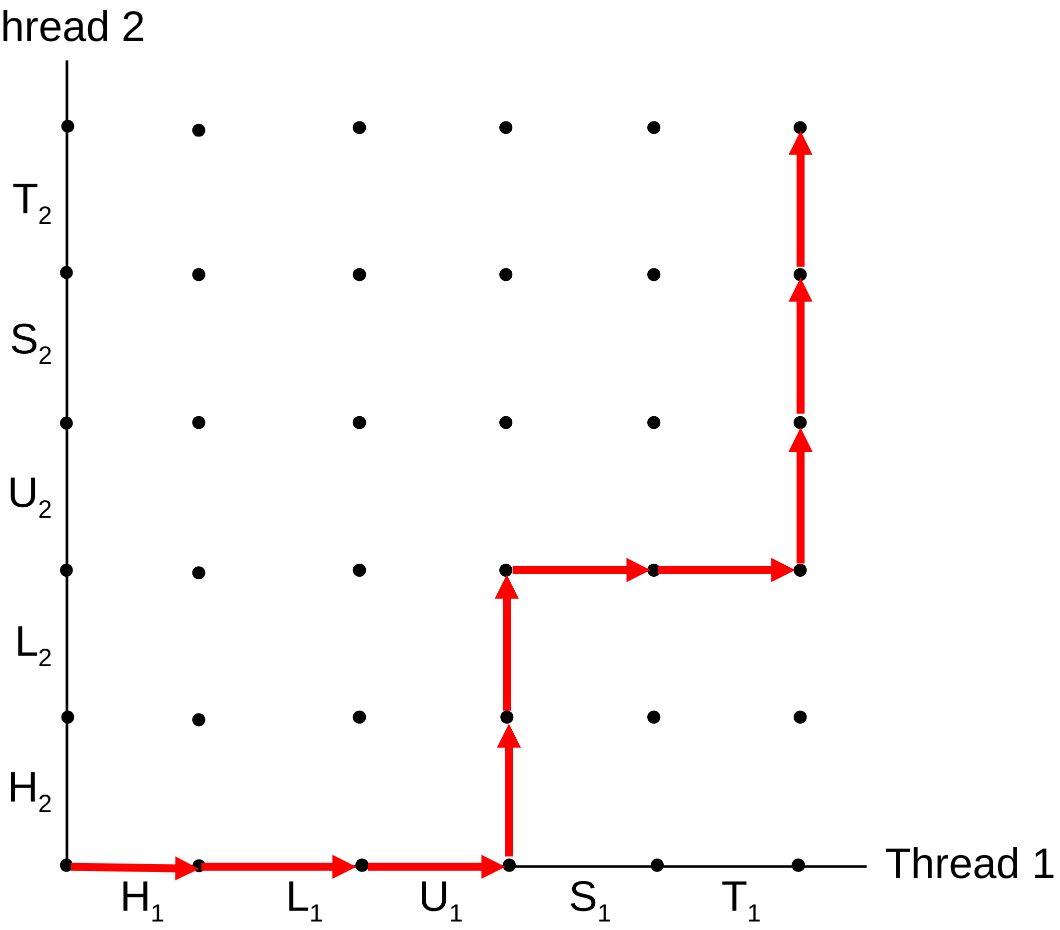
T_i : Tail

- H_i and T_i manipulate only local stack variables.
- L_i U_i and S_i manipulate the shared counter variable.

Process Graph

Models the execution of n threads as a trajectory through an *n*-dimensional Cartesian space.

- each axis *k* shows the progress of thread *k*
- each point $(I_1, I_2, \ldots, I_n)$ represents the state where thread *k* has completed instruction I_k
- the trajectory corresponds to the ordering of instructions



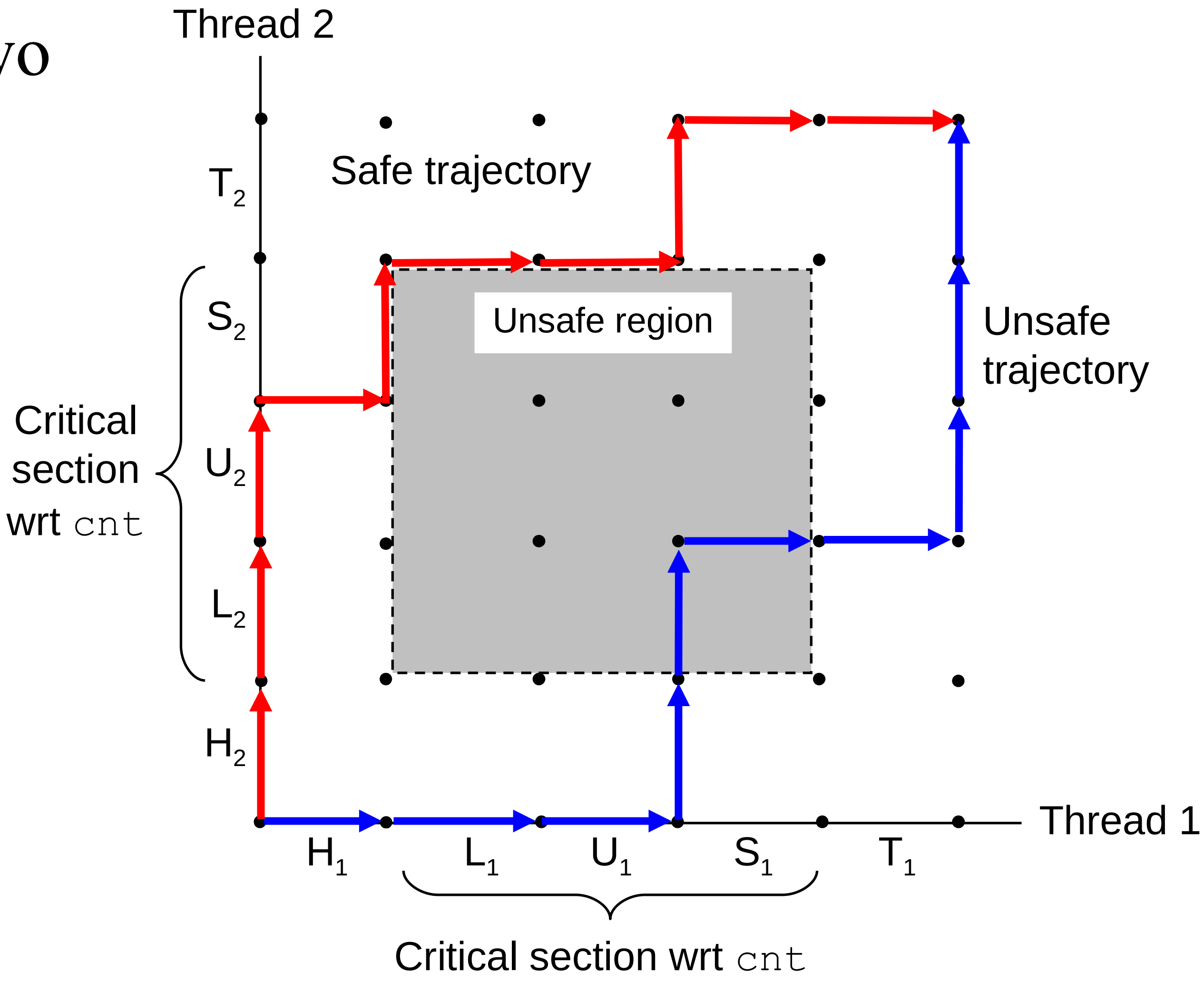
Critical Section

- Instructions L_i, U_i, and, S_i constitute a *critical section* for thread i.

- The intersection of two critical sections is an *unsafe region*.

- A *safe trajectory* skirts the unsafe region.

- An *unsafe trajectory* touches any part of the unsafe region.



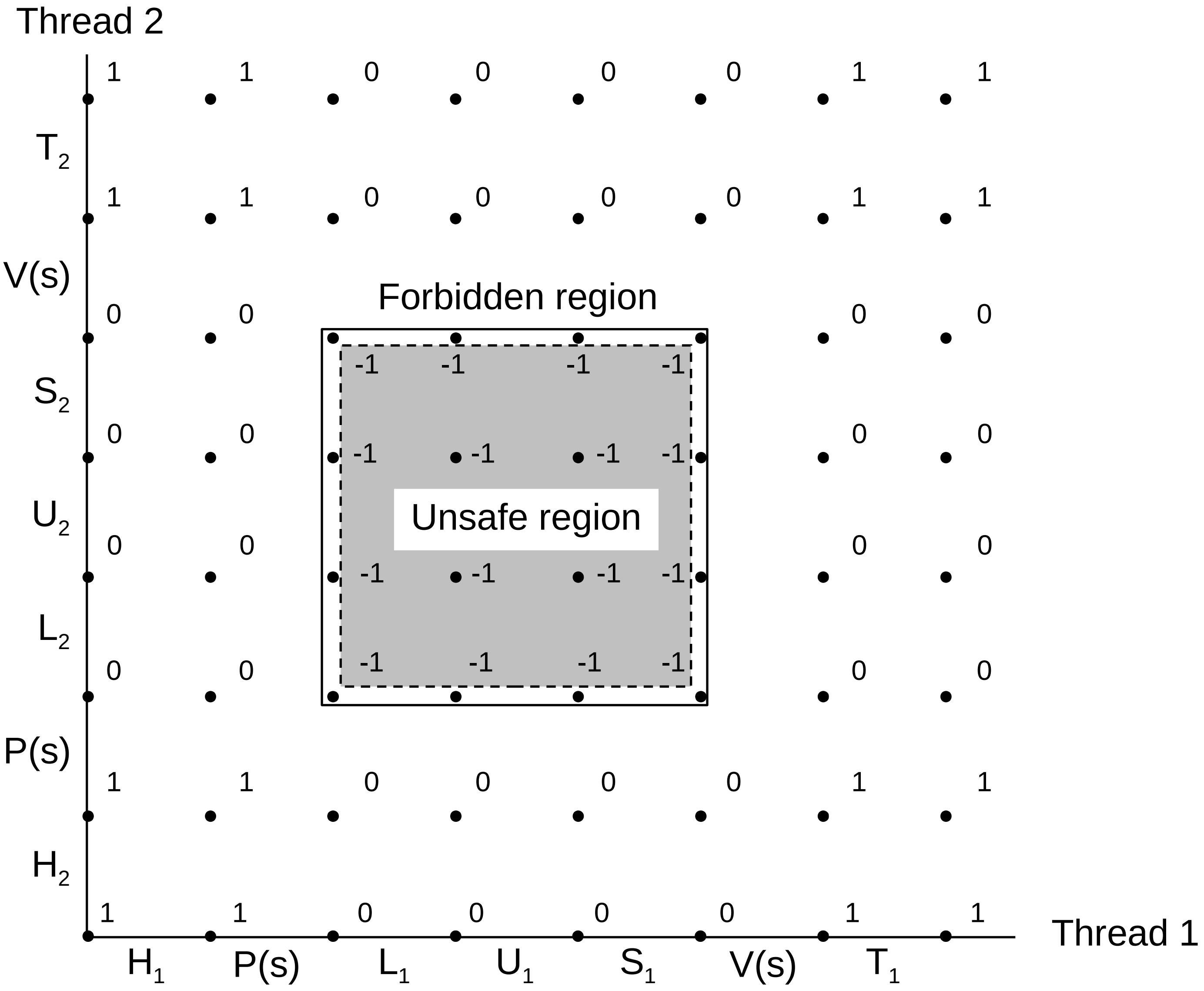
Semaphore

- A global variable $s \geq 0$ that can only be manipulated using one of two operations: *P* and *V*.
- $P(s)$
  - if $s != 0$, s-- and return (occurs indivisibly)
  - if $s == 0$, suspend the process until s becomes nonzero (process is restarted by a *V* operation), after restarting s-- and return
- $V(s)$
  - s++ and check to see if any processes are blocked in a *P* operation waiting for s to become nonzero (restarts exactly one of such processes)
  - increment occurs indivisibly

Binary Semaphores

- Associate a semaphore s (initially 1) with each shared variable (or related set of shared variables).
- Surround the corresponding critical section with $P(s)$ and $V(s)$ operations.
- Binary—value of s is always 0 or 1.
- The semaphore operations ensure *mutually exclusive access* to the critical region.
- Where should $P(s)$ and $V(s)$ be placed in our `cnt` example?

Example: Binary Semaphore



Initially
s = 1

Posix Semaphores

- Functions for manipulating semaphores.

```
int sem_init(sem_t* sem, 0, unsigned int value);
int sem_wait(sem_t* sem);   /* P(s) */
int sem_post(sem_t* sem);   /* V(s) */

void P(sem_t* sem);         /* wrapper */
void V(sem_t* sem);         /* wrapper */
```

- *Example*:

```
sem_t mutex; /* semaphore to synch cnt access */
sem_init(&mutex, 0, 1);               /* init mutex */
...
for(i = 0; i < NITERS; i++) {
  P(&mutex);                       /* protect shared */
  cnt++;                             /* variable cnt */
  V(&mutex);
}
```

Producer-Consumer Model

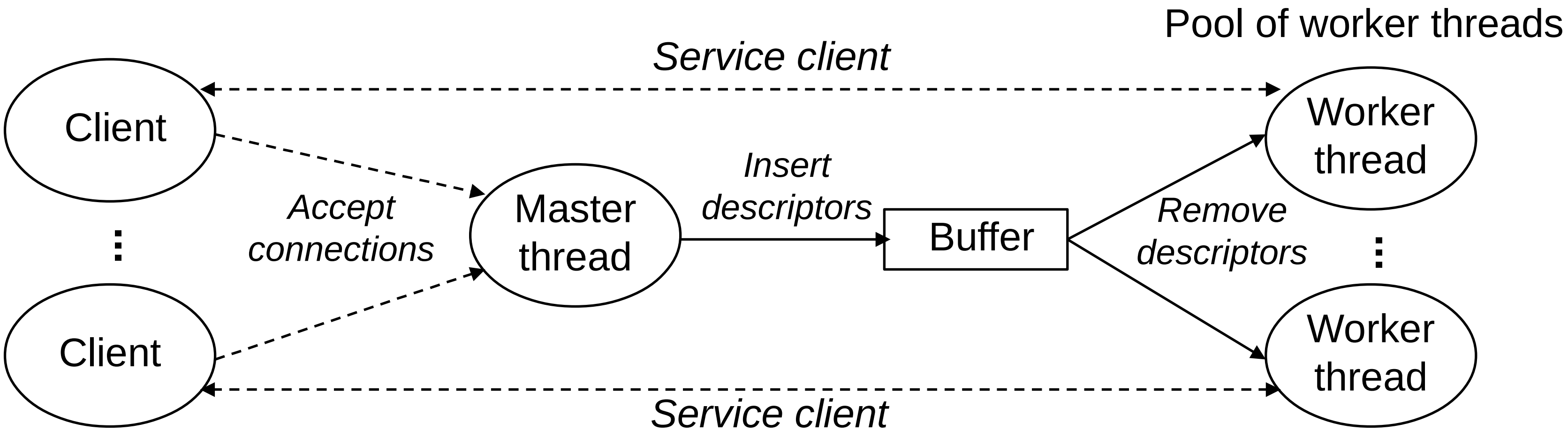
- Producer and consumer threads share a bounded buffer, with *n* slots.
  - producer thread adds items to the buffer
  - consumer thread retrieves items from the buffer
- Must guarantee mutually-exclusive access to the buffer, and that the producer/consumer cannot access the buffer if it is full/empty.

```
typedef struct {
  int* buf;    /* Buffer array */
  int n;       /* Max # of slots */
  int front;   /* buf[(front+1)%n] is 1st 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;
```

```
/* 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 data items */
}

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

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

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

Prethreading

- Recall that our concurrent echo server creates a new thread for each client, incurring significant overhead.
- Another solution includes a main thread (the server) and *n* worker threads.
  - main thread accepts connection requests from clients and puts each connection descriptor in a shared buffer
  - each worker thread repeatedly removes a descriptor from the buffer, services the client, and waits for the next descriptor



```
/* echoservert_pre.c – a prethreaded concurrent echo server */
sbuf_t sbuf; /* shared buffer of connected descriptors */

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

  if (argc != 2)
    /* ERROR, QUIT */
  port = atoi(argv[1]);
  sbuf_init(&sbuf, 16);
  listenfd = Open_listenfd(port);

  for(i = 0; i < 4; i++)  /* Create worker threads */
    Pthread_create(&tid, NULL, thread, NULL);

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

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

```
/* A thread-safe version of echo that counts the total number
   of bytes received from clients. */

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;
}

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);
  }
}
```

Other Concurrency Issues

- We've looked at techniques for mutual exclusion and producer-consumer synchronization, a small part of concurrency programming.
- Synchronization is a fundamentally difficult problem that raises issues that do not arise in sequential programs.
- What follows is a sample of the issues programmers must be aware of when writing concurrent programs.
- Presented in the context of threads, the issues exist whenever concurrent flows manipulate shared resources.

Thread Safety

- A function is *thread-safe* iff it always produces correct results when called repeatedly from multiple concurrent threads.
  - a function that is not thread-safe is called *thread-unsafe*
- Four (non-disjoint) 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

Class 1: Shared Variables

-
  ```
  /* thread-unsafe routine */
  void* count(void* arg) {
    int i;
    for(i = 0; i < NITERS; i++)
      cnt++;
    return NULL;
  }
  ```

- To make thread-safe, protect the shared variable with synchronization operations.

- *Pro*: No changes in the calling program required.

- *Con*: Synchronization operations will slow down the function.

Class 2: Keeps State Across Calls

-
  ```
  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;
  }
  ```
- Calling `rand` repeatedly from a single thread is correct.
  - What can happen if it is called from multiple threads?
- To make thread-safe, we must rely on the caller to pass state information via arguments.
  - forces a change in the code of the calling routine
  - potentially 100s of call sites, a difficult and error-prone change

Class 3: Returns Pointer to Static

- Some functions compute a result in a local `static` variable and return a pointer to that variable.
  - results being used by one thread may be silently overwritten by another thread
- To make thread-safe, require the caller to pass the address of the variable in which to store the result.
  - removes shared variable, requires change in calling code
- Another option is the *lock-and-copy* technique.
  - associate a mutex with the thread-unsafe function
  - especially useful when the thread-unsafe function is impossible to modify (e.g., it is linked from a library)

Lock-and-Copy

- At each call site:
  - dynamically allocate memory for the result
  - lock the mutex
  - call the thread-unsafe function
  - copy the result returned by the function to this memory
  - unlock the mutex
-
```
struct hostent* gethostbyname_ts(char* hostname) {
   struct hostent *sharedp, *unsharedp;

   unsharedp = Malloc(sizeof(struct hostent)); /* dyn mem */
   P(&mutex);                                     /* lock mutex */
   sharedp = gethostbyname(hostname); /* thread-unsafe fn */
   *unsharedp = *sharedp;        /* copy to private struct */
   V(&mutex);                                   /* unlock mutex */
   return unsharedp;
}
```

Class 4: Calls Thread-Unsafe

- If function *f* calls thread-unsafe function *g*, *f* may or may not also be thread-unsafe.
- If *g* keeps state across multiple invocations, then *f* is also thread-unsafe.
    - only solution is to rewrite *g*
- If *g* does not protect shared variables or returns a pointer to a static variable, *f* may still be thread-safe.
    - solution is to protect call to *g* with a mutex (like previous example)

Reentrancy

- Reentrant functions do not reference any shared data when they are called by multiple threads.
- The set of reentrant functions is a proper subset of the thread-safe functions.
  - due to the lack of synchronization ops, reentrant functions are typically more efficient that non-reentrant thread-safe functions
- The only way to convert a Class 2 thread-unsafe function into a thread-safe one is to rewrite it to be reentrant.

```
/* rand_r - a reentrant pseudo-random integer generator */
int rand_r(unsigned int* nextp) {
  *nextp = *nextp * 1103515245 + 12345;
  return (unsigned int)(*nextp / 65536) % 32768;
}
```

Determining Reentrancy

- *Explicitly reentrant*—all function arguments are passed by value and all data references are to local automatic stack variables.
- *Implicitly reentrant*—allows some parameters in an otherwise explicitly-reentrant function to be pointers.
  - thus, it is a reentrant function only if the calling threads are careful to pass pointers to non-shared data
  - *example*: function `rand_r`
- Why is function `gethotstbyname_ts` thread-safe, but not reentrant?

Races

- A *race* occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y.
- Threaded programs must work correctly for any feasible trajectory.
  - Often programmers assume that threads will take a particular trajectory through the execution state space.

Example: Race

```
void* thread(void* vargp);

int main() {
  pthread_t tid[4];
  int i;

  for(i = 0; i < 4; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
  for(i = 0; i < 4; 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;
}
```

```
unix> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3

unix> ./race
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

Example: No Race

```
void* thread(void* vargp);

int main() {
  pthread_t tid[4];
  int i, *ptr;

  for(i = 0; i < 4; i++) {
    ptr = Malloc(sizeof(int));
    *ptr = i;
    Pthread_create(&tid[i], NULL, thread, ptr);
    /* why not call free here? */
  }
  for(i = 0; i < 4; 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;
}
```

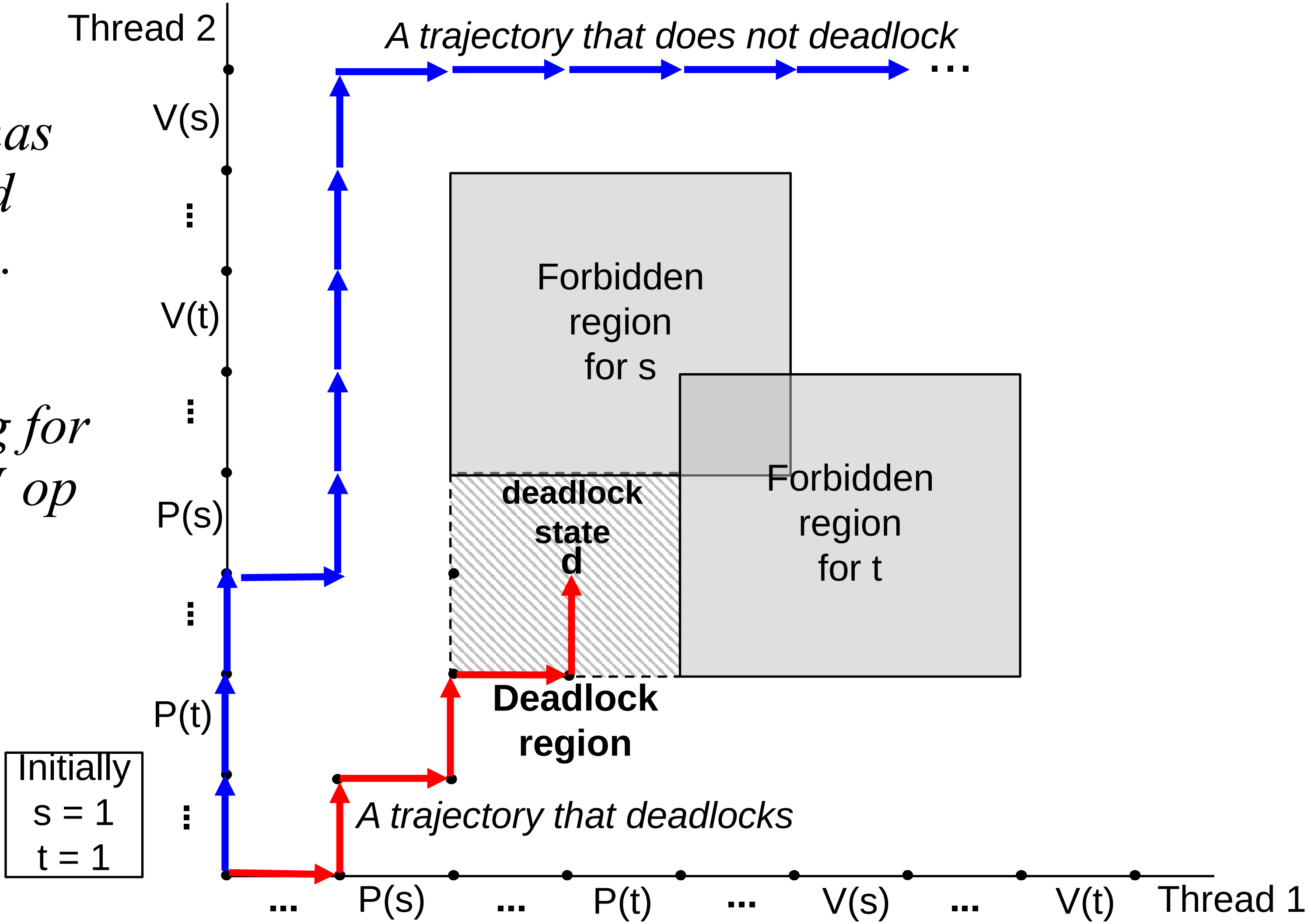
```
unix> ./norace
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

Deadlock

A run-time error where a collection of threads are blocked, waiting for a condition that will never be true.

The programmer has incorrectly ordered the semaphore ops.

In state d*, each program is waiting for the other to do a* V *op that won't occur.*



Avoiding Deadlock

- Deadlock is difficult to predict in a program.
  - some trajectories will skirt the deadlock region
  - others will be trapped by it
- When binary semaphores are used for mutual exclusion, a simple rule can be applied.
  - A program is deadlock-free if, for each pair of mutexes (*s*, t) in the program, each thread that holds both s and t simultaneously locks them in the same order.
- In our example, lock *s* first then *t*, in each thread.

Exercise: Deadlock

- *Initially*: `s = 1`, `t = 0`.

Thread 1:	*Thread 2*:
`P(s);`	`P(s);`
`V(s);`	`V(s);`
`P(t);`	`P(t);`
`V(t);`	`V(t);`

- Does this program deadlock? Always?
- If so, what simple change to the initial semaphore values will eliminate the potential for deadlock?

Summary

- A concurrent program consists of a collection of logical flows that overlap in time.
  - via processes—scheduled by the kernel, separate address space
  - via threads—scheduled by the kernel, shared address space
- *P* and *V* operations on semaphores help to synchronize concurrent accesses to shared data.
  - provides mutually exclusive access to shared data
  - schedules access to shared buffers in producer-consumer programs
- Difficult concurrency issues:
  - thread safety, reentrant functions, races, deadlocks