Interaction and Communication between Programs

Synchronization in Concurrent Programs

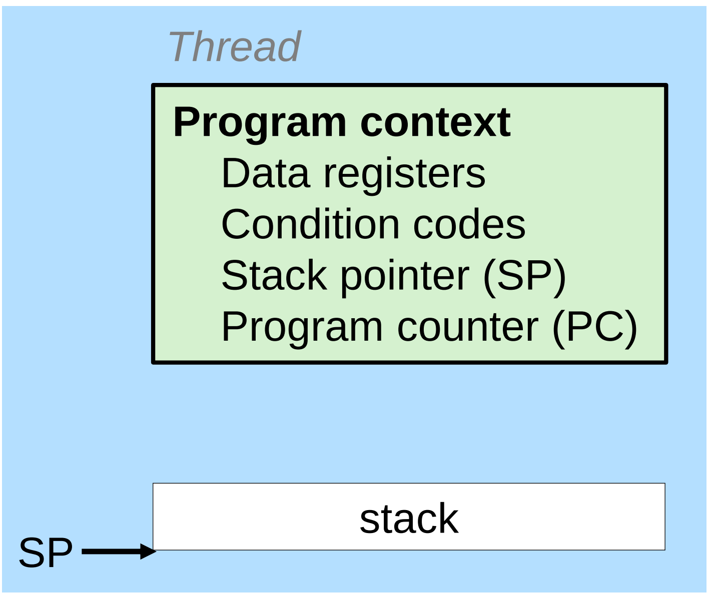
Synchronization in Concurrent Programs

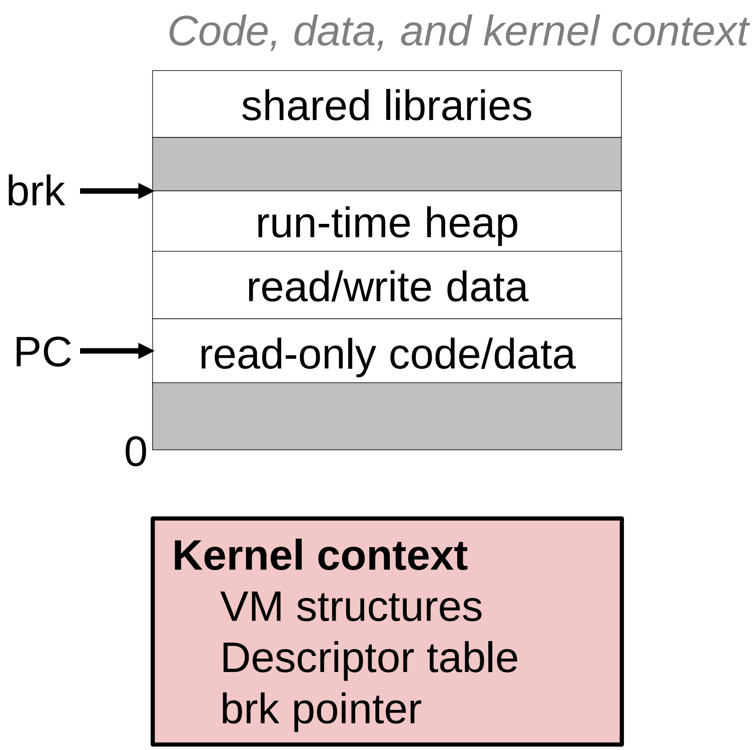
- **Threads review**
- Sharing
- Mutual exclusion
- Semaphores
- Synchronization
  - Producer-consumer problem
  - Readers-writers problem
  - Thread safety
  - Races
  - Deadlocks

Acknowledgement: slides based on the cs:app2e material

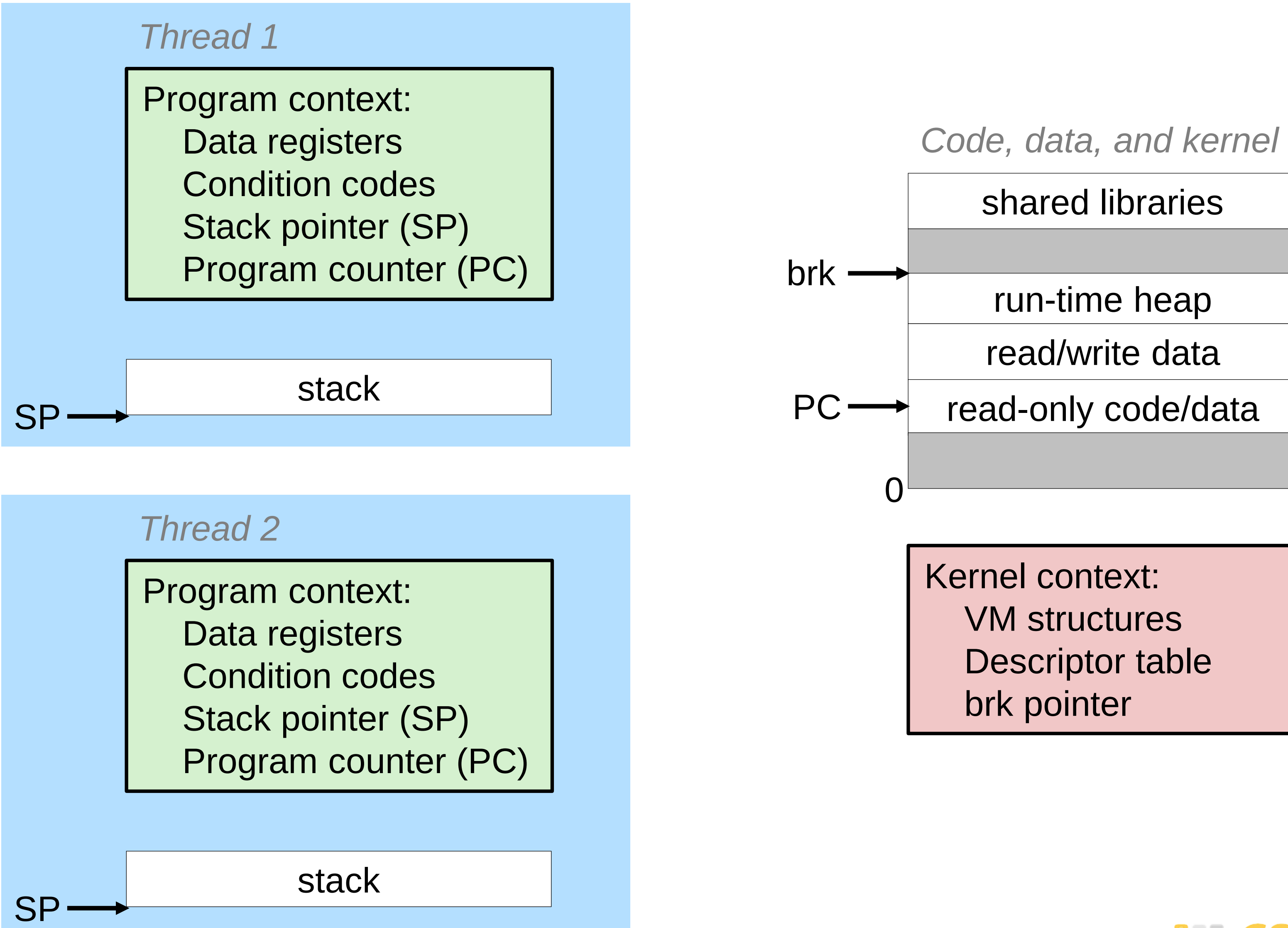
Process: Single Thread View

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





Process with Two Threads



Threads vs. Processes

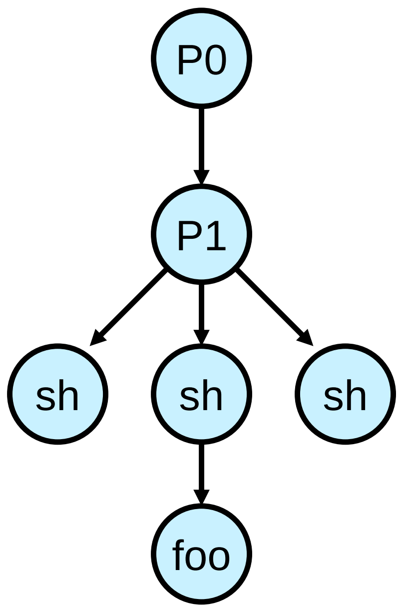
- Threads and processes: similarities
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) by the kernel

- Threads and processes: differences
  - Threads share code and data, processes (typically) do not
  - Threads are less expensive than processes
    - Process control (creating and reaping) is more expensive as thread control
    - Context switches for processes more expensive than for threads

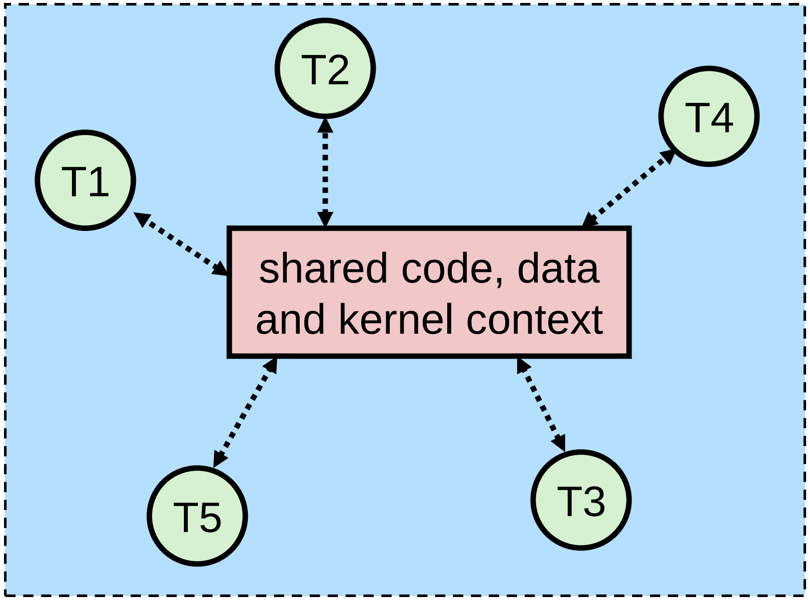
Threads vs. Processes (cont.)

- Processes form a tree hierarchy
- Threads form a pool of peers
  - Each thread can kill any other
  - Each thread can wait for any other thread to terminate
  - Main thread: first thread to run in a process

Process hierarchy



Thread pool



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`
    - `pthread_cond_init`
    - `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

```
/*
 * 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);
}
```

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

Thread attributes (usually `NULL`)

Thread arguments (`void *p`)

return value (`void **p`)

Pros and Cons of Thread-Based Designs

+ Easy to share data structures between thread
  - 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!

Synchronization in Concurrent Programs

- Threads review
- **Sharing**
- Mutual exclusion
- Semaphores
- Synchronization
  - Producer-consumer problem
  - Readers-writers problem
  - Thread safety
  - Races
  - Deadlocks

CSE 컴퓨터공학부
Department of Computer Science & Engineering

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*”

- 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?

- ***Definition***:
  A variable `x` is shared if and only if multiple threads reference some instance of `x`.

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

→ This mismatch between the conceptual and operation model is a source of confusion and errors

Example Program to Illustrate Sharing

```
char **ptr;  /* global */

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

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int) vargp;
    static int cnt = 0;

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

Peer threads reference main thread's stack indirectly through global `ptr` *variable*

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`)

```
char **ptr;  /* global */

int main()
{
    int 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]
)

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

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

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

Shared Variable Analysis

- Which variables are shared?

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
ptr	yes	yes	yes
cnt	no	yes	yes
i.m	yes	no	no
msgs.m	yes	yes	yes
myid.p0	no	yes	no
myid.p1	no	no	yes

- 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

Synchronization in Concurrent Programs

- Threads review
- Sharing
- **Mutual exclusion**
- Semaphores
- Synchronization
  - Producer-consumer problem
  - Readers-writers problem
  - Thread safety
  - Races
  - Deadlocks

badcnt.c: Improper Synchronization

```
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
  int niters = atoi(argv[1]);
  pthread_t tid1, tid2;

  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=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}
```

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

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

  return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

`cnt` should equal 20,000.

What went wrong?

Assembly Code for Counter Loop

C code for counter loop in thread i

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

Corresponding assembly code

```
        movl (%rdi),%ecx
        movl $0,%edx
        cmpl %ecx,%edx
        jge .L13
.L11:
        movl cnt(%rip),%eax
        incl %eax
        movl %eax,cnt(%rip)
        incl %edx
        cmpl %ecx,%edx
        jl .L11
.L13:
```

Head (H_i)

Load `cnt` (L_i)
Update `cnt` (U_i)
Store `cnt` (S_i)

Tail (T_i)

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
  - `%eax`$_i$ is the content of `%eax` in thread i's context

i (thread)	instr$_i$	%eax$_1$	%eax$_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

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

`i` (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

Concurrent Execution (cont)

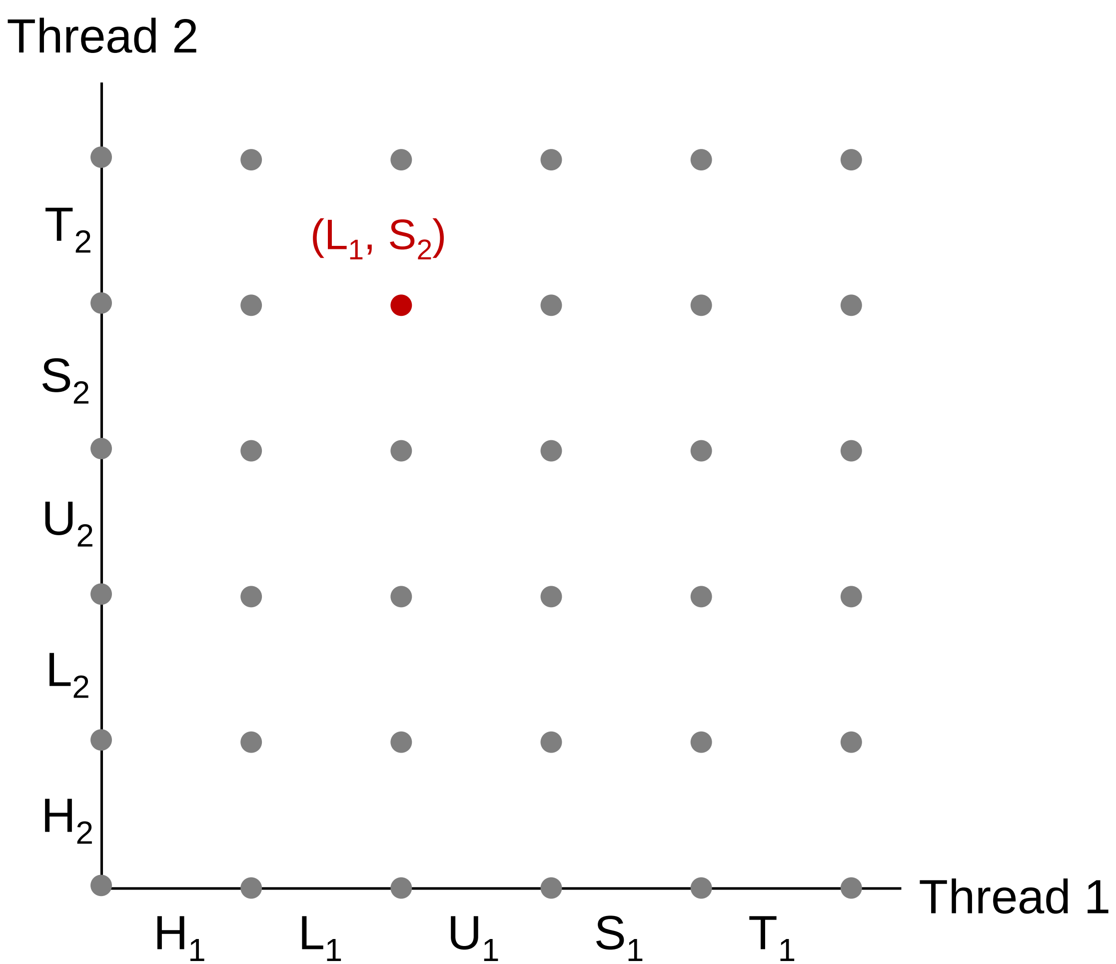
- How about this ordering?

`i` (thread)	`instr`$_i$	`%eax`$_1$	`%eax`$_2$	`cnt`
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			
2	T_2			1

Oops!

- We can analyze the behavior using a progress graph

Progress Graphs



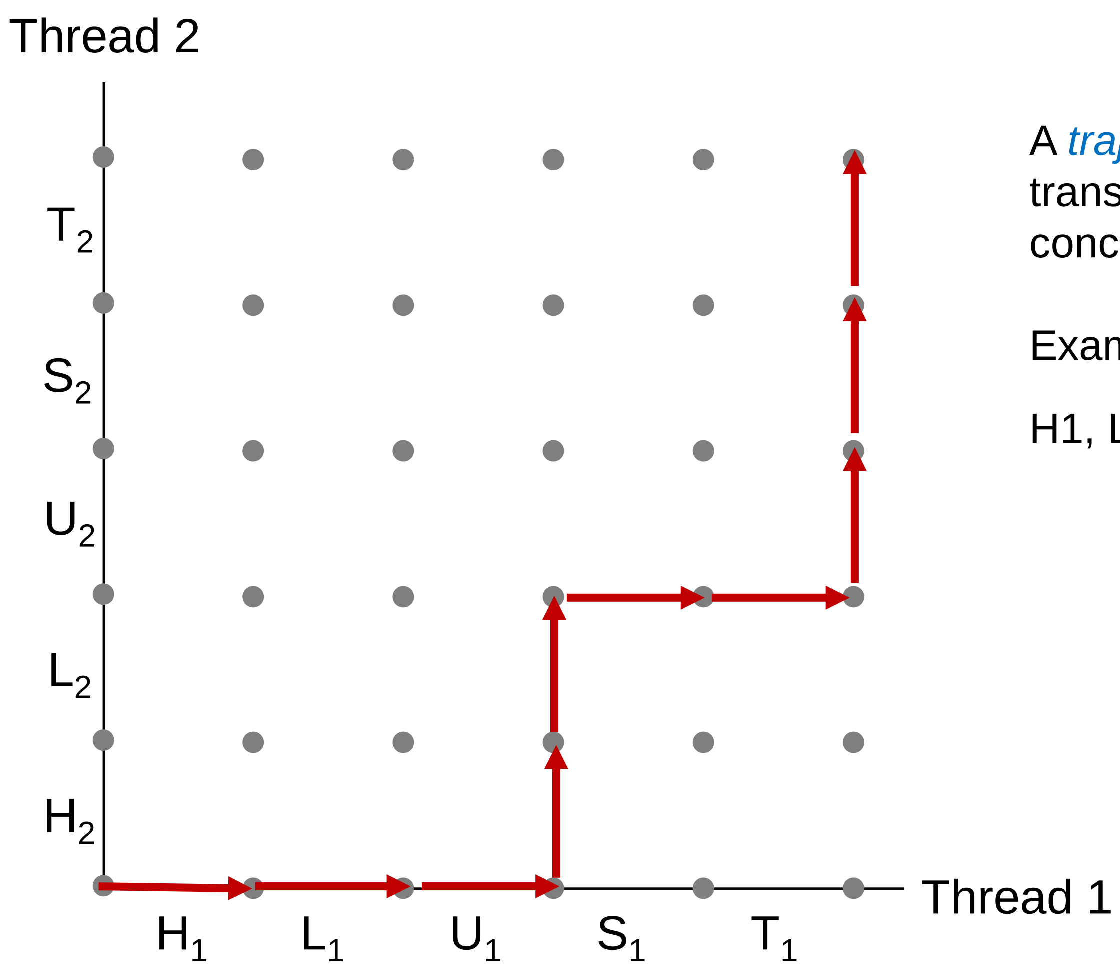
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

Each point corresponds to a possible *execution state* ($Inst_1$, $Inst_2$).

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs

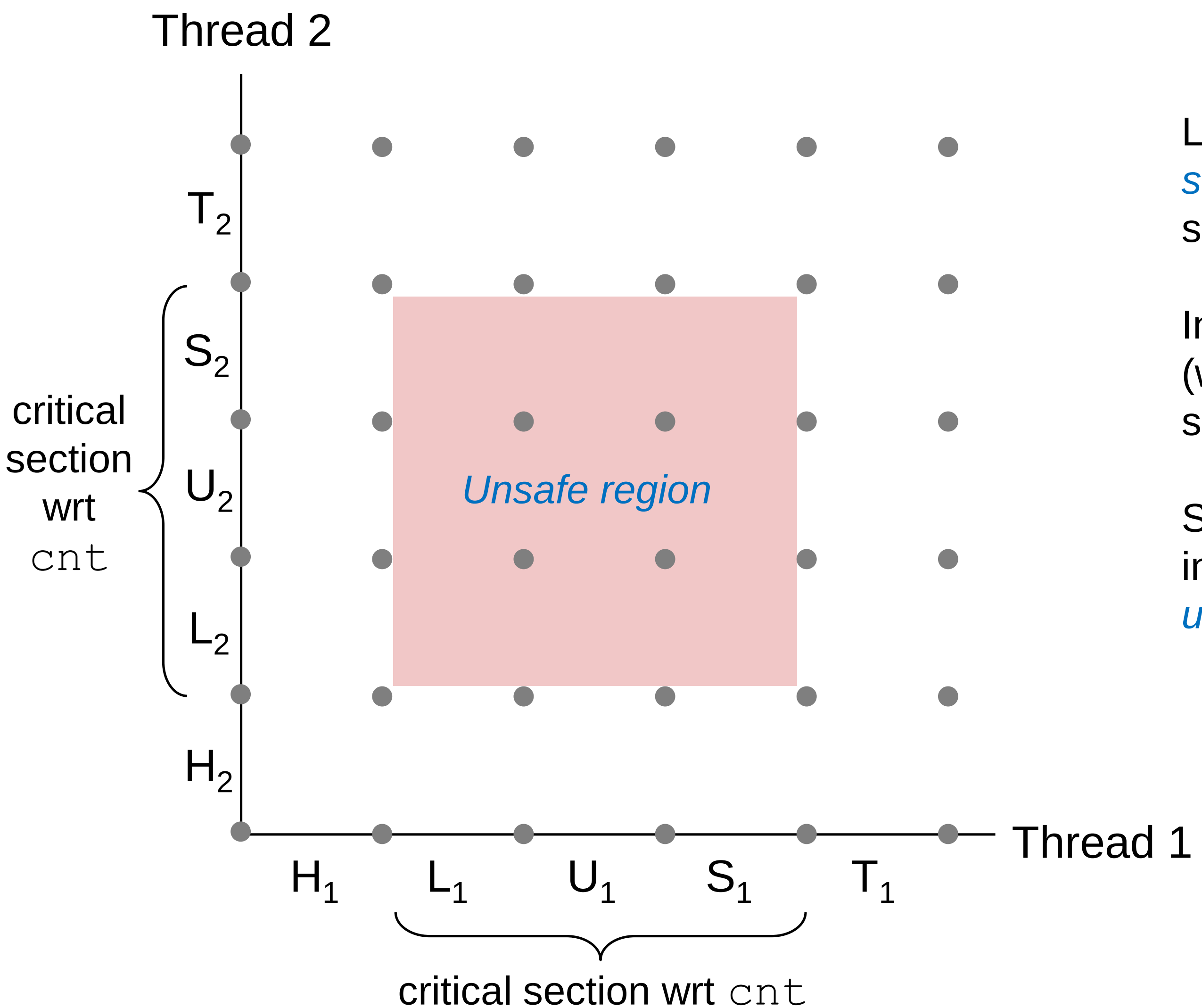


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

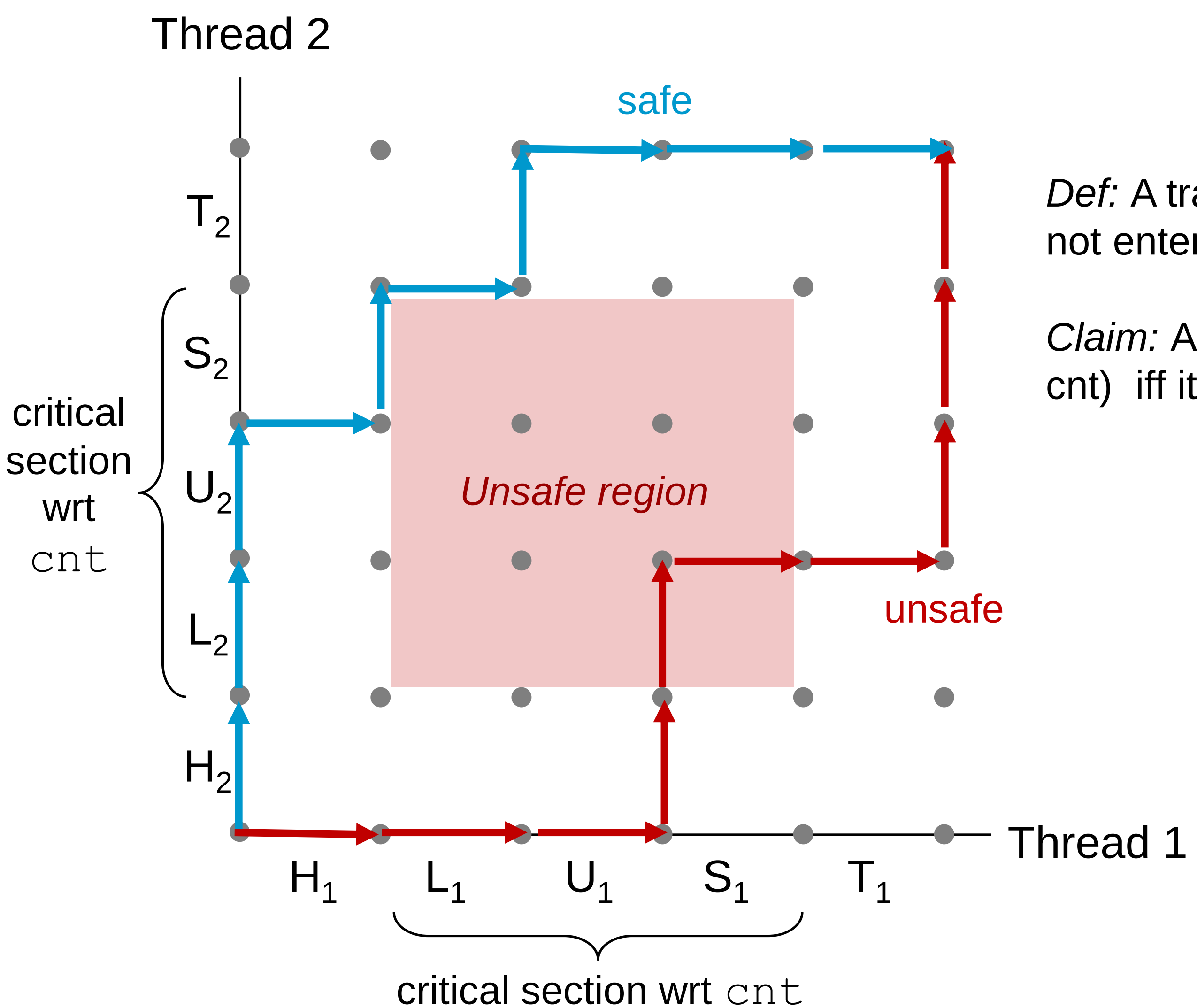


L, U, and S form a *critical section* with respect to the shared variable cnt

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

Sets of states where such interleaving occurs form *unsafe regions*

Critical Sections and 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 never have an unsafe trajectory.
  - i.e., need to guarantee mutually exclusive access to critical regions

- Classic solution:
  - Semaphores (Edsger Dijkstra)

- Other approaches (out of our scope)
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Synchronization in Concurrent Programs

- Threads review
- Sharing
- Mutual exclusion
- **Semaphores**
- Synchronization
  - Producer-consumer problem
  - Readers-writers problem
  - Thread safety
  - Races
  - Deadlocks

Semaphores

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

- Manipulated by *P* and *V* operations:
  - *P(s)*: `[  while (s == 0) wait(); s--; ]`   "test" / "wait"
  - *V(s)*: `[  s++; ]`   "increment" / "post"

- *OS kernel guarantees that operations between brackets [ ] are executed indivisibly*
  - Only one *P* or *V* operation at a time can modify s.
  - When while loop in P terminates, only that P can decrement s

- *Semaphore invariant*: (s >= 0)

C Semaphore Operations

- POSIX Pthreads semaphore functions

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

- CA:APP wrapper functions

```
#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

```
volatile int cnt = 0; /* global */

int main(int argc, char **argv)
{
  int niters = atoi(argv[1]);
  pthread_t tid1, tid2;

  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=%d\n", cnt);
  else
    printf("OK cnt=%d\n", cnt);
  exit(0);
}
```

```
/* Thread routine */
void *thread(void *vargp)
{
  int i, niters = *((int *)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:

```
volatile int cnt = 0;     /* Counter */
sem_t mutex;              /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1);   /* mutex = 1 */
```

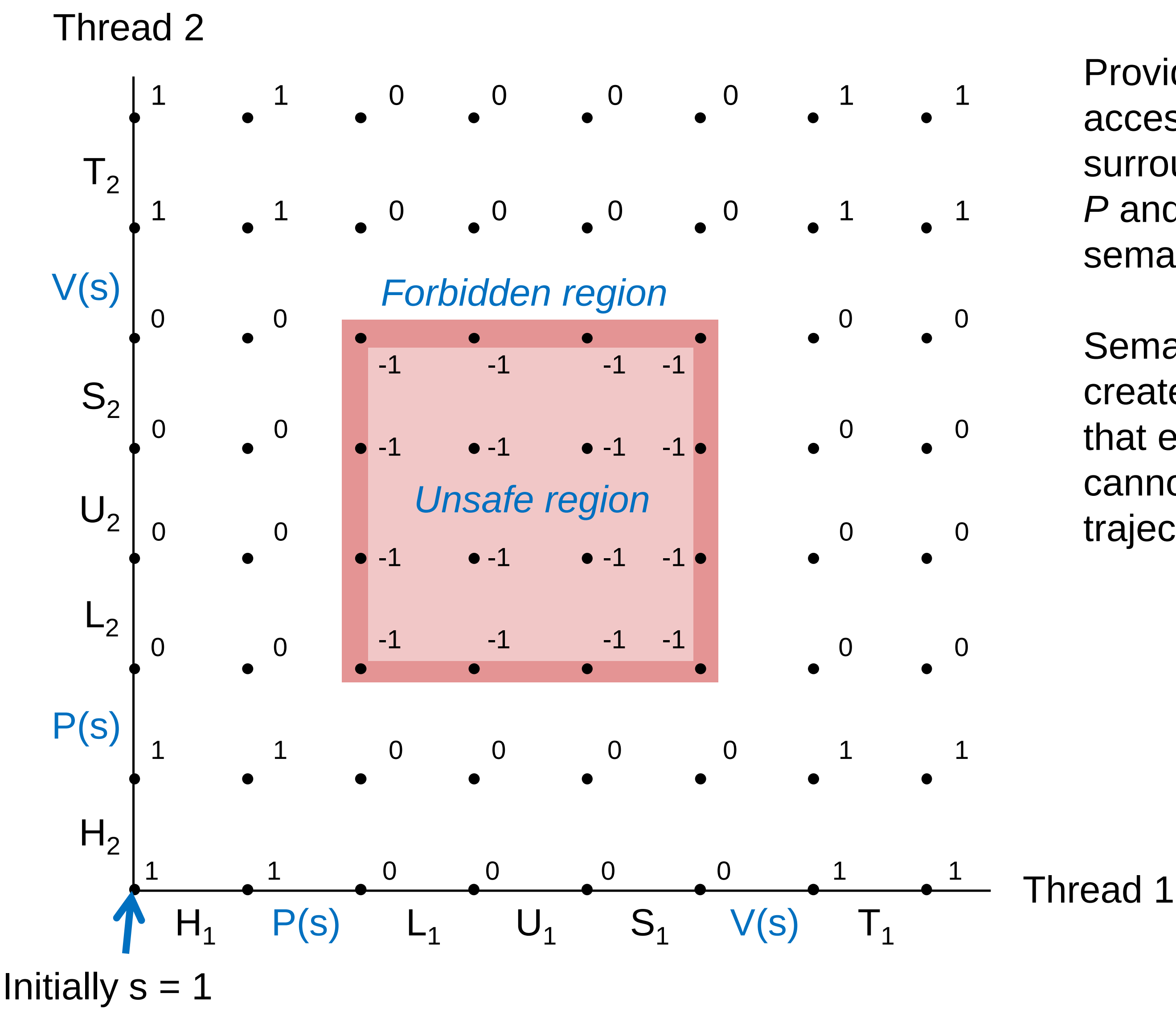
- Surround critical section with *P* and *V*:

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

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Observation: much slower than `badcnt.c`.

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 that cannot be entered by any trajectory.

Synchronization in Concurrent Programs

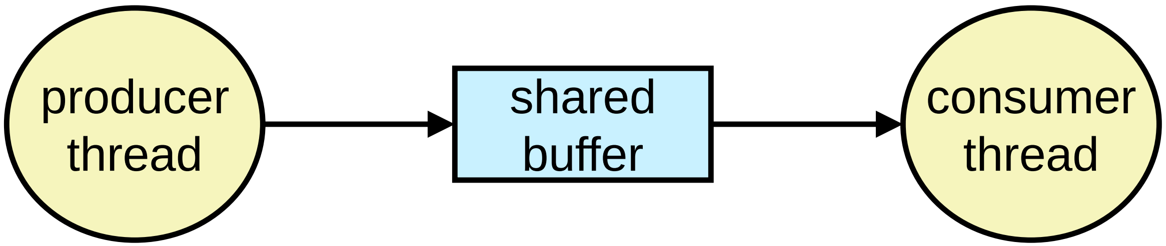
- Threads review
- Sharing
- Mutual exclusion
- Semaphores
- **Synchronization**
  - **Producer-consumer problem**
  - Readers-writers problem
  - Thread safety
  - Races
  - Deadlocks

Using Semaphores to Schedule Access to Shared Resources

- Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true
  - Use counting semaphores to keep track of resource state.
  - Use binary semaphores to notify other threads.

- 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 1-element Buffer

```
#include “csapp.h”

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} shared;

…
```

```
…

int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* Initialize the semaphores */
  Sem_init(&shared.empty, 0, 1);
  Sem_init(&shared.full,  0, 0);

  /* Create threads and wait */
  Pthread_create(&tid_producer, NULL,
                 producer, NULL);
  Pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  exit(0);
}
```

Producer-Consumer on 1-element Buffer

- Initially: `empty==1, full==0`

Producer Thread

```
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* Write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return NULL;
}
```

Consumer Thread

```
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* Read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* Consume item */
    printf("consumed %d\n", item);
  }
  return NULL;
}
```

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

- Data structure and interface

```
#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 zero 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
```

Synchronization in Concurrent Programs

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
- **Synchronization**
  - Producer-consumer problem
  - **Readers-writers problem**
  - Thread safety
  - Races
  - Deadlocks

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.
- Third readers-writers problem (equal priority to both)
  - Under the assumption of a FIFO wake-up sequence for semaphores
- *Starvation* (where a thread waits indefinitely) is possible in both cases.

Solution to First Readers-Writers Problem

- Reader code

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

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

    /* Reading happens here */

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

- Writer code

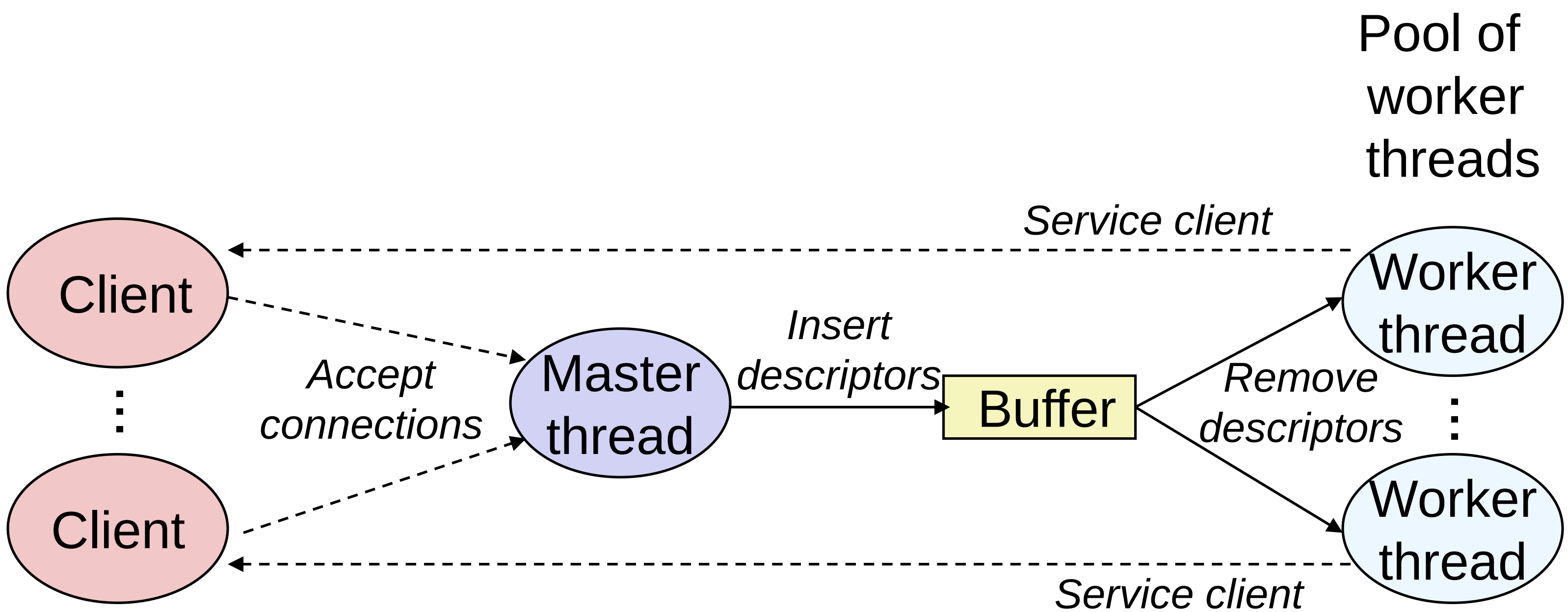
```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Case Study: Prethreaded Concurrent Server



Prethreaded Concurrent Server

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

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

    port = atoi(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    listenfd = Open_listenfd(port);

    for (i = 0; i < NTHREADS; 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 */
    }
}
```

echoservert_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
                                            buffer */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}                                               echoservert_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

Synchronization in Concurrent Programs

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
- **Synchronization**
  - Readers-writers problem
  - Producer-consumer problem
  - **Thread safety**
  - Races
  - Deadlocks

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 semaphore operations to protect parts that manipulate shared variables
  - 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

- Fix: Pass state as part of argument
  - and, thereby, eliminate static 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;
}
```

Warning: Some functions like `gethostbyname` require a *deep copy.* Use reentrant *`gethostbyname_r`* version instead.

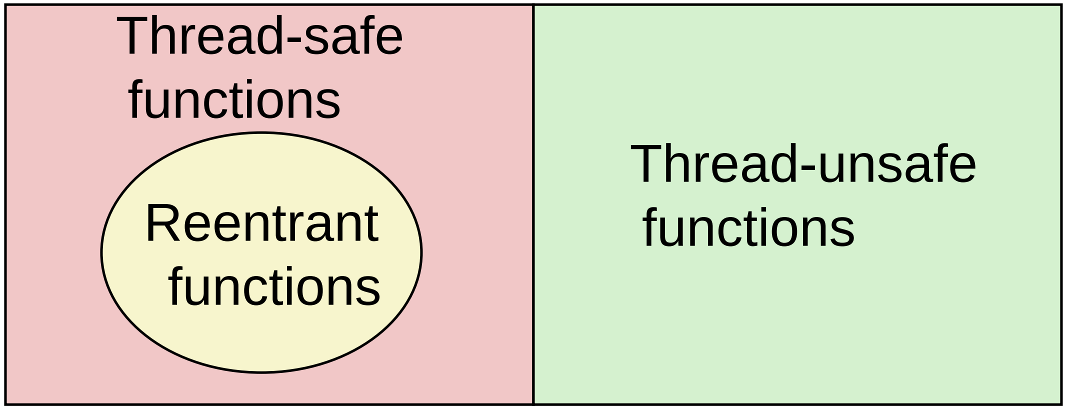
Thread-Unsafe Functions (Class 4)

- Calling thread-unsafe functions
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix 1: Modify the function so it calls only thread-safe functions
  - Fix 2: Protect the call site and resulting shared data with a mutex iff the callee is of class 1 or 3.

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 reetnrant (e.g., rand_r)

All functions



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 important exceptions:

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

Synchronization in Concurrent Programs

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
- **Synchronization**
  - Readers-writers problem
  - Producer-consumer problem
  - Thread safety
  - **Races**
  - Deadlocks

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;
}
```

race.c

Race Elimination

- Make sure don't have unintended sharing of state

```
/* a threaded program without the race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = malloc(sizeof(int));
        *valp = i;
        Pthread_create(&tid[i], NULL, thread, valp);
    }
    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;
}
```

norace.c

Synchronization in Concurrent Programs

- Threads review
- Sharing
- Mutual exclusion
- Semaphores
- **Synchronization**
  - Readers-writers problem
  - Producer-consumer problem
  - Thread safety
  - Races
  - **Deadlocks**

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!

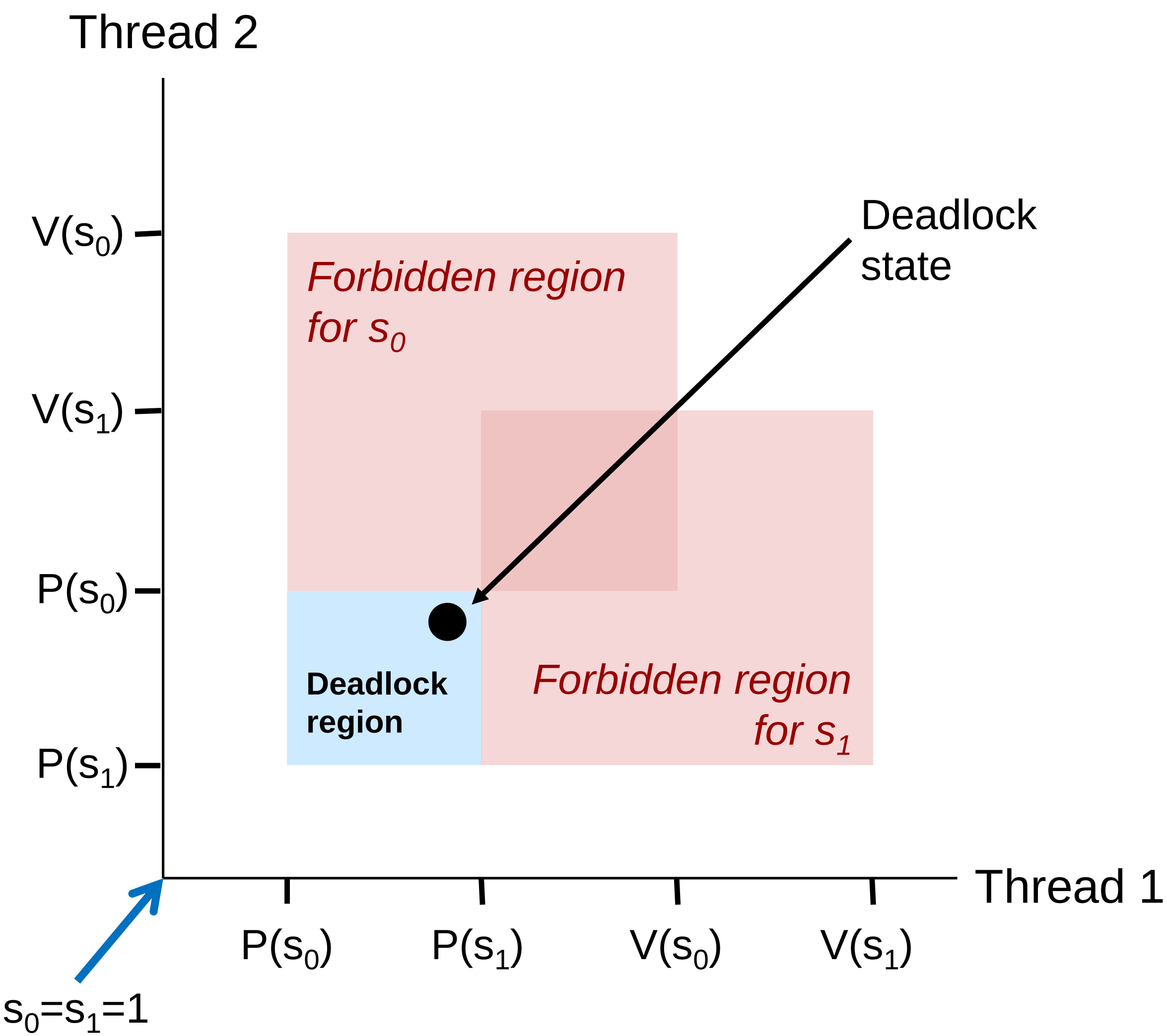
Deadlocking With Semaphores

```
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]:	Tid[1]:
P(s_0);	P(s_1);
P(s_1);	P(s_0);
cnt++;	cnt++;
V(s_0);	V(s_1);
V(s_1);	V(s_0);

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

Avoiding Deadlocks: Acquire shared resources *in the same order*

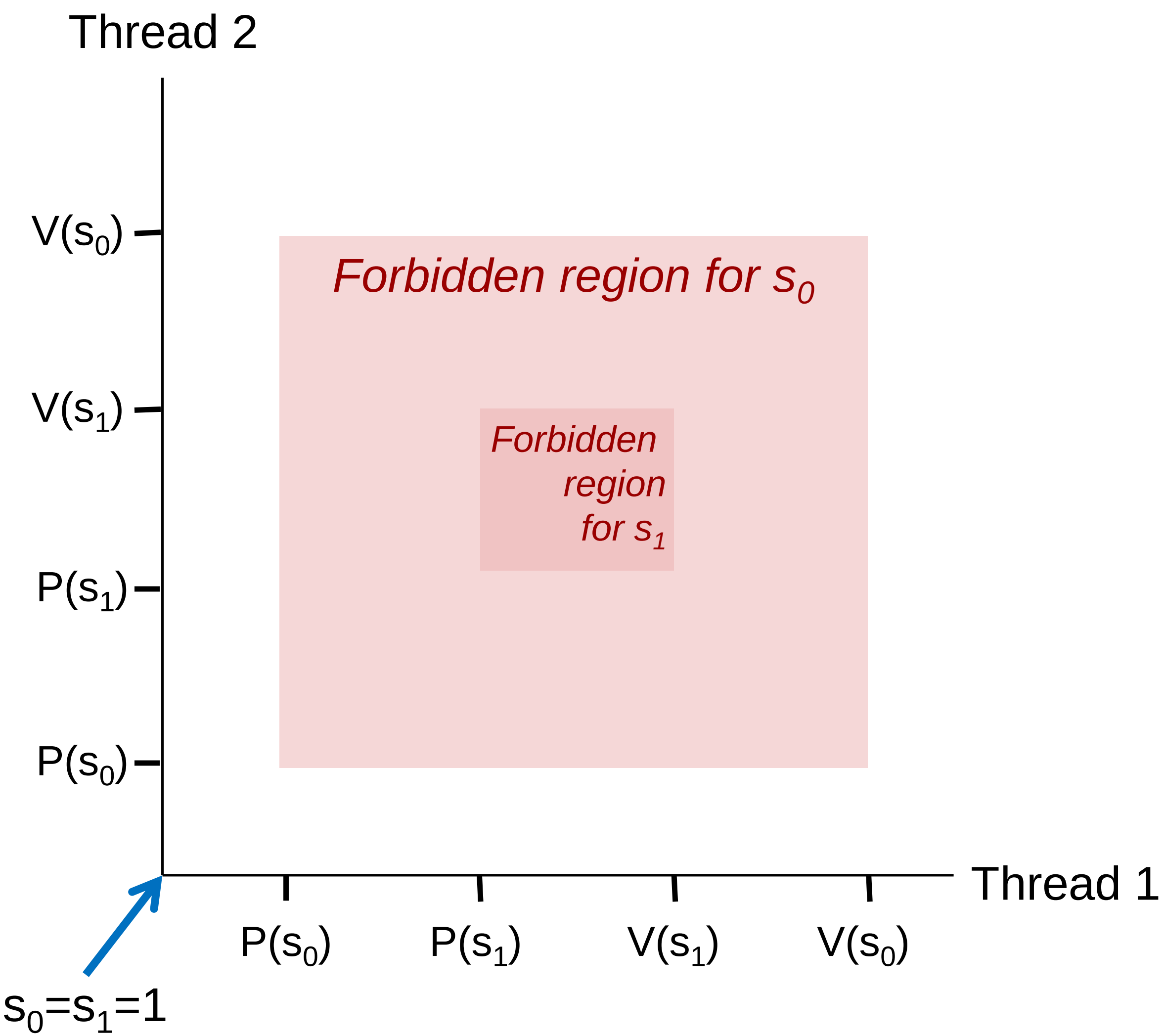
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1],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

Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_0)$;
$P(s_1)$;	$P(s_1)$;
cnt++;	cnt++;
$V(s_1)$;	$V(s_1)$;
$V(s_0)$;	$V(s_0)$;

what about the release order?

Avoiding Deadlock in Progress Graph



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released?

Synchronization Summary

- Threads provide another mechanism for writing concurrent programs
- Threads are popular
  - Somewhat cheaper than processes
  - Easy to share data between threads
- However, the ease of sharing has a cost
  - Easy to introduce subtle synchronization errors

- 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.

- For more info
  - D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997