Synchronization: Advanced

15-213 / 18-213: Introduction to Computer Systems
25th Lecture, July 31, 2018

Instructors:

Abi Kim

Reminder: Semaphores

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

Manipulated by *P* and *V* operations:

- *P(s):* [`while (s == 0) wait(); s--;`]
  - Dutch for "Proberen" (test)
- *V(s):* [`s++;`]
  - Dutch for "Verhogen" (increment)

OS kernel guarantees that operations between brackets [ ] are executed atomically

- 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)*

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)
```

Today

- **Using semaphores to schedule shared resources**
  - Producer-consumer problem
  - Readers-writers problem
- **Other concurrency issues**
  - Thread safety
  - Races
  - Deadlocks

Using Semaphores to Coordinate Access to Shared Resources

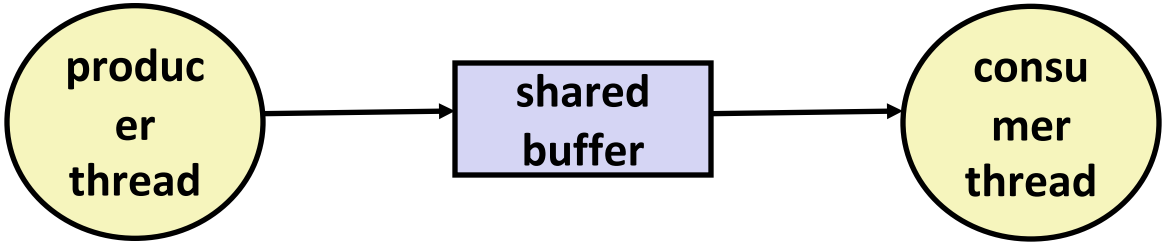
Basic idea: Thread uses a semaphore operation to notify another thread that some condition has become true

- Use counting semaphores to keep track of resource state.
- Use binary semaphores to notify other threads.

Two classic examples:

- The Producer-Consumer Problem
- The Readers-Writers Problem

Producer-Consumer Problem



Common synchronization pattern:

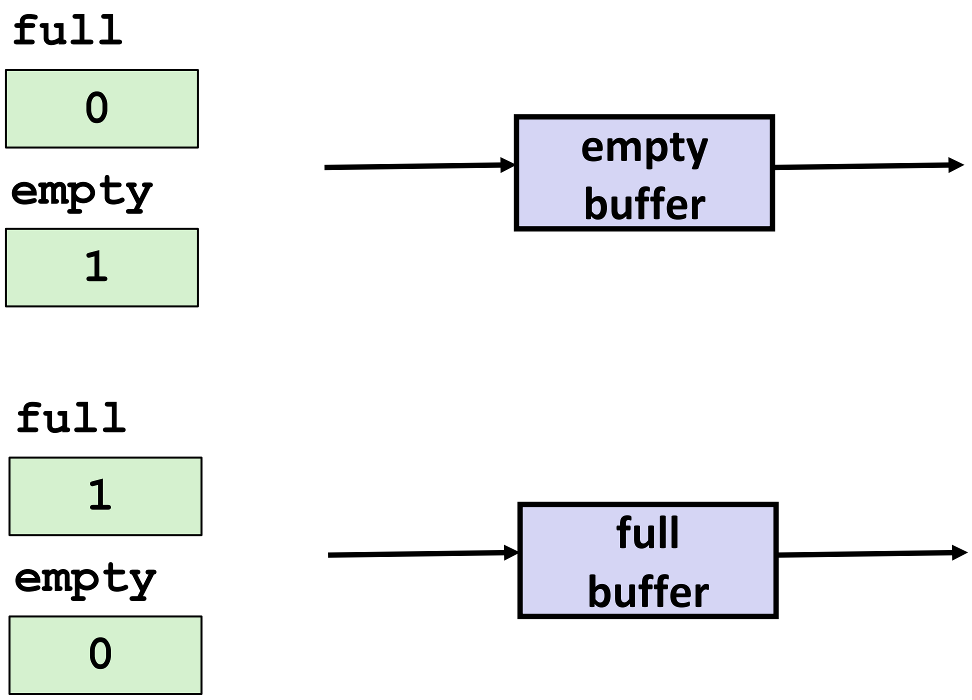
- Producer waits for empty ***slot***, inserts item in buffer, and notifies consumer
- Consumer waits for ***item***, removes it from buffer, and notifies producer

Examples

- Multimedia processing:
  - Producer creates 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

- **Maintain two semaphores: `full` + `empty`**



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(int argc, char** argv) {
  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);

  return 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

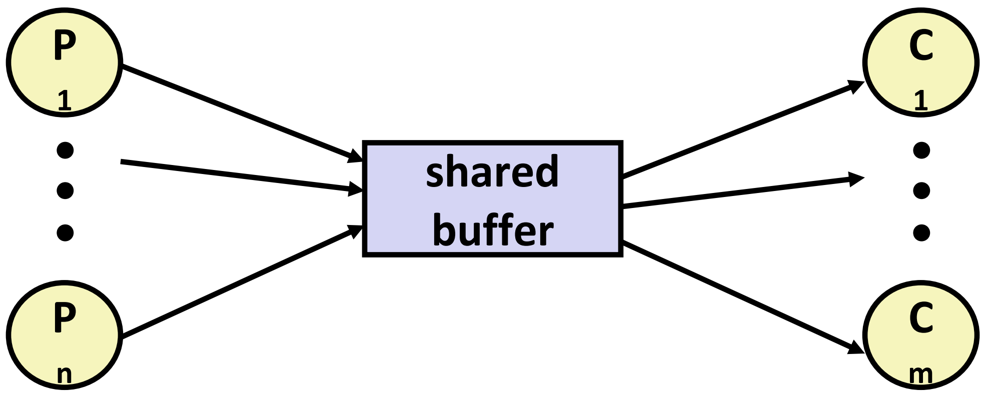
```
void *consumer(void *arg) {
  int i, item;

  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;
}
```

Why 2 Semaphores for 1-Entry Buffer?

- Consider multiple producers & multiple consumers



- Producers will contend with each to get `empty`
- Consumers will contend with each other to get `full`

Producers

```
P(&shared.empty);
shared.buf = item;
V(&shared.full);
```

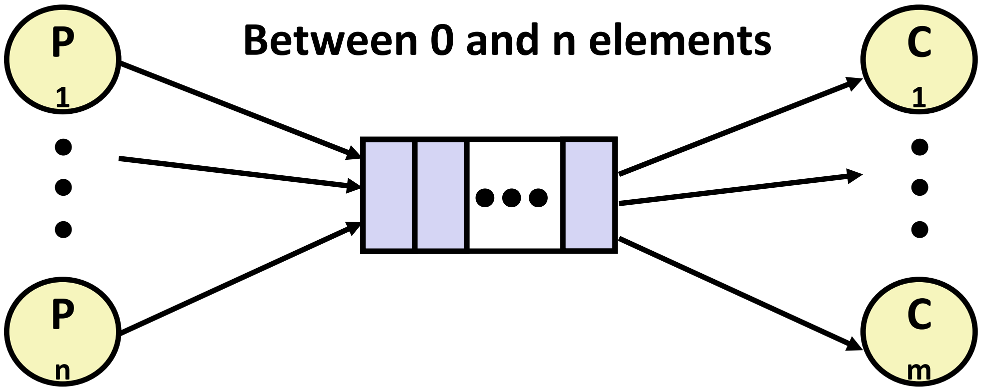
`empty`

`full`

Consumers

```
P(&shared.full);
item = shared.buf;
V(&shared.empty);
```

Producer-Consumer on an *n*-element Buffer



Implemented using a shared buffer package called `sbuf`.

Circular Buffer (n = 10)

Store elements in array of size n

items: number of elements in buffer

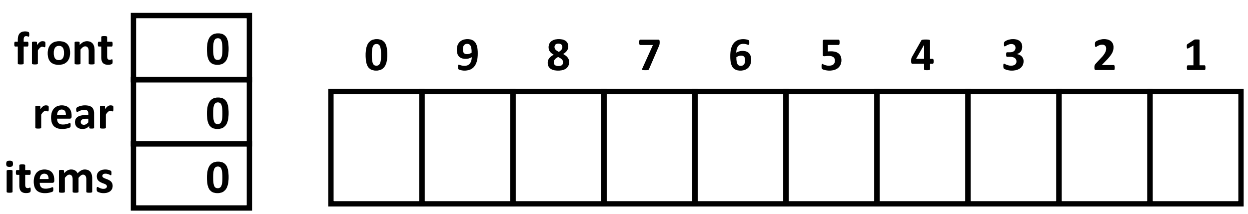
Empty buffer:

- front = rear

Nonempty buffer

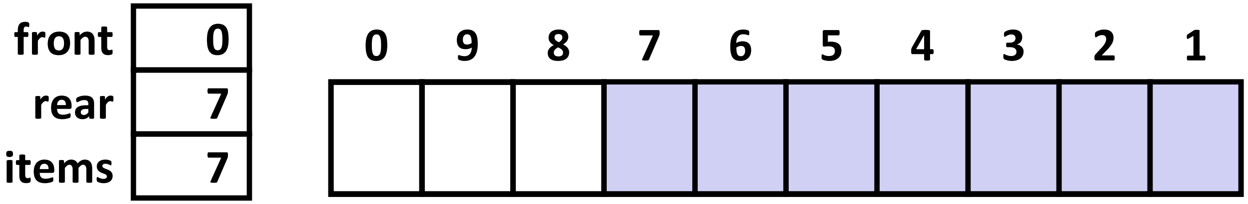
- rear: index of most recently inserted element
- front: index of next element to remove – 1 (mod n)

Initially:



Circular Buffer Operation (n = 10)

Insert 7 elements



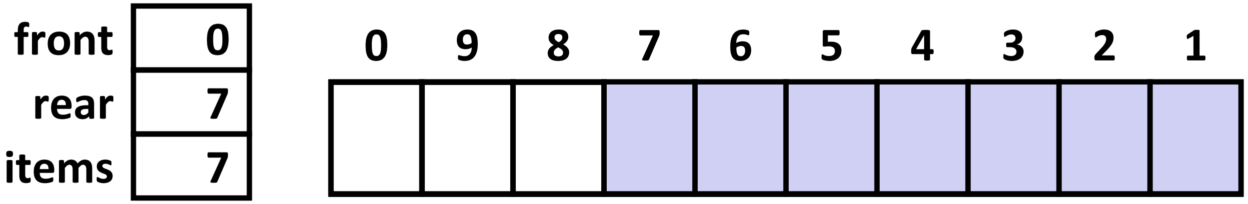
Remove 5 elements

Insert 6 elements

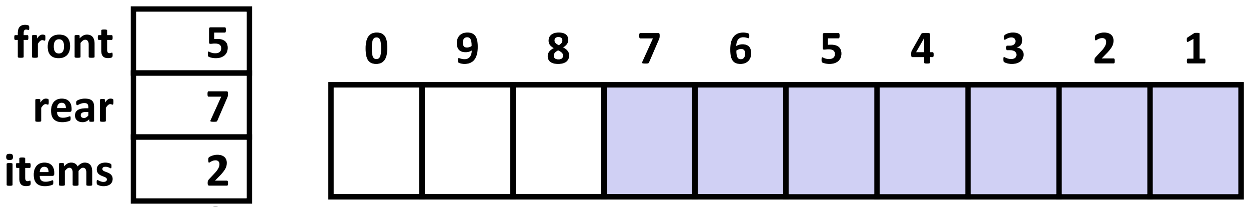
Remove 8 elements

Circular Buffer Operation (n = 10)

Insert 7 elements



Remove 5 elements

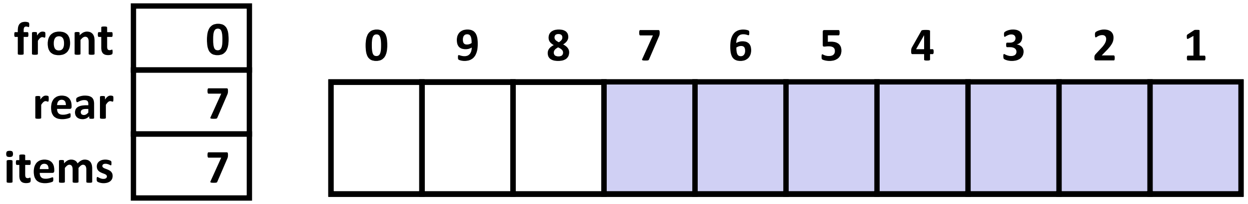


Insert 6 elements

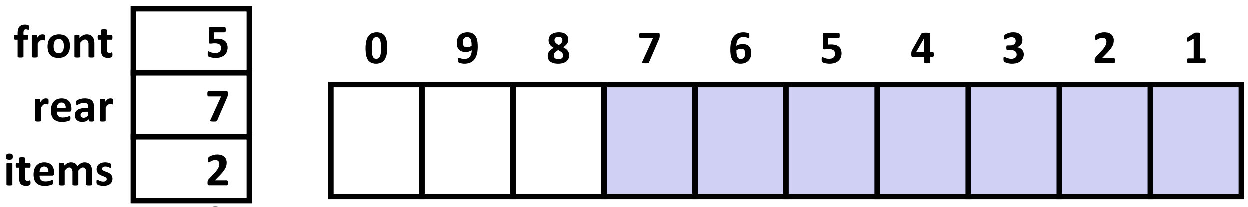
Remove 8 elements

Circular Buffer Operation (n = 10)

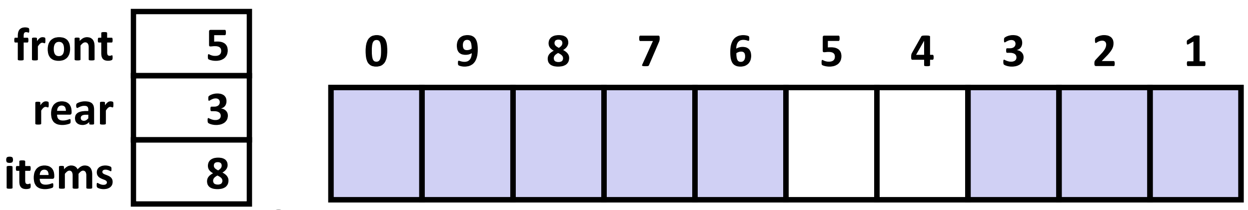
Insert 7 elements



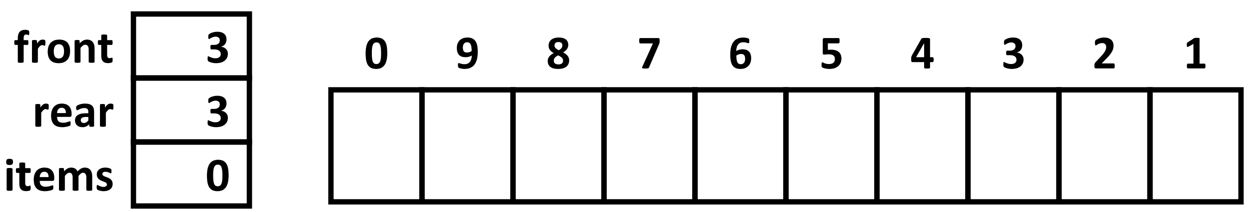
Remove 5 elements



Insert 6 elements

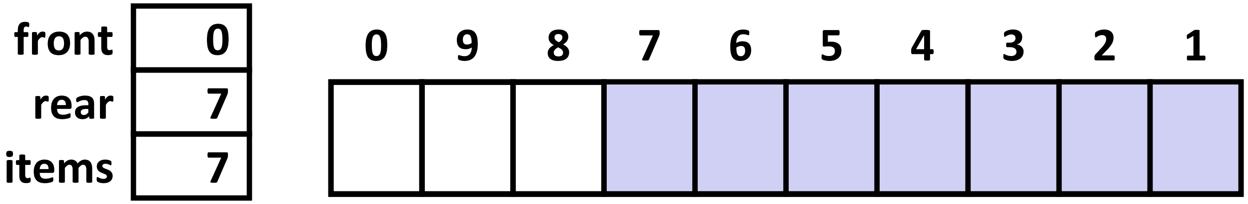


Remove 8 elements

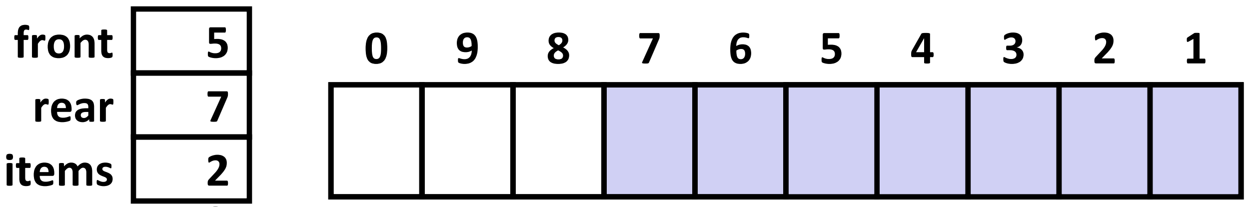


Circular Buffer Operation (n = 10)

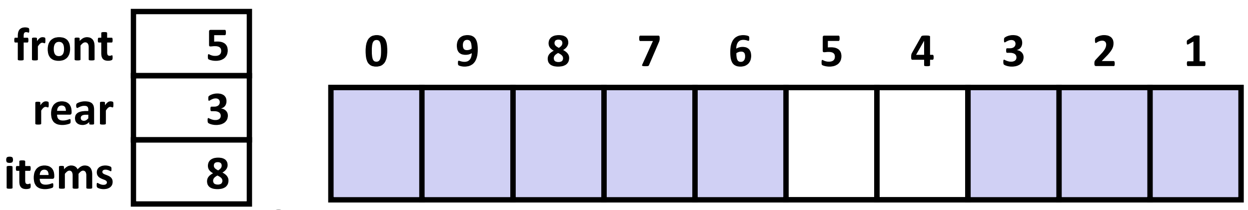
Insert 7 elements



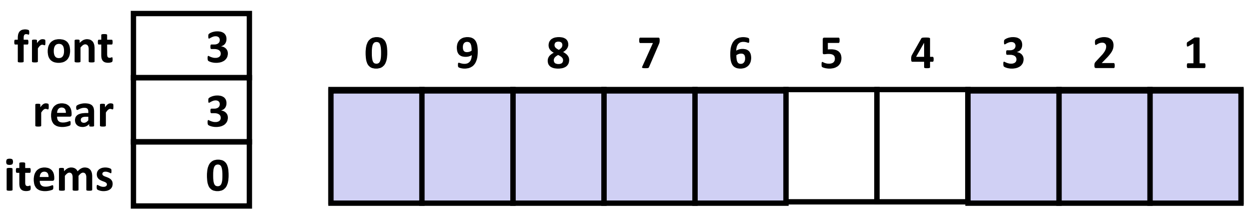
Remove 5 elements



Insert 6 elements



Remove 8 elements



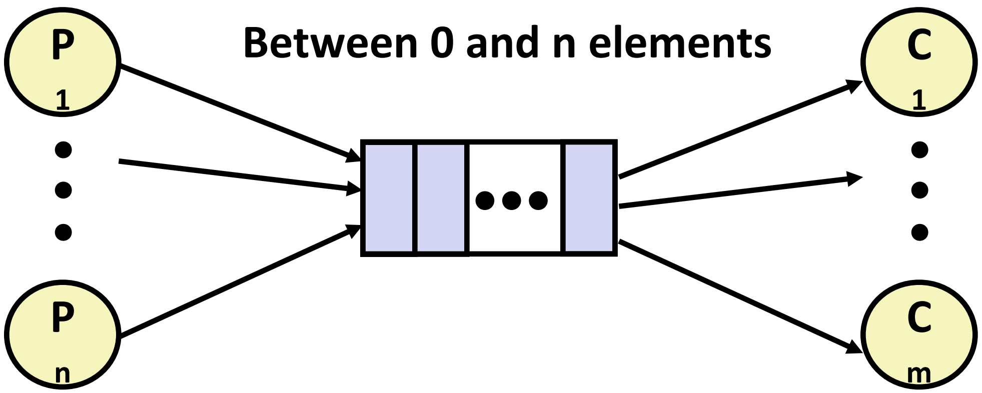
Sequential Circular Buffer Code

```
init(int v)
{
    items = front = rear = 0;
}
```

```
insert(int v)
{
    if (items >= n)
        error();
    if (++rear >= n) rear = 0;
    buf[rear] = v;
    items++;
}
```

```
int remove()
{
    if (items == 0)
        error();
    if (++front >= n) front = 0;
    int v = buf[front];
    items--;
    return v;
}
```

Producer-Consumer on an *n*-element Buffer



Requires a mutex and two counting semaphores:

- `mutex`: enforces mutually exclusive access to the buffer and counters
- `slots`: counts the available slots in the buffer
- `items`: counts the available items in the buffer

Makes use of general semaphores

- Will range in value from 0 to n

sbuf Package - Declarations

```
#include "csapp.h"

typedef struct {
    int *buf;       /* Buffer array                   */
    int n;          /* Maximum number of slots        */
    int front;      /* buf[front+1 (mod n)] is first item */
    int rear;       /* buf[rear]   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         */
    if (++sp->rear >= sp->n)        /* Increment index (mod n) */
        sp->rear = 0;
    sp->buf[sp->rear] = item;       /* Insert the item         */
    V(&sp->mutex);                  /* Unlock the buffer       */
    V(&sp->items);                  /* Announce available item */
}
```

```
insert(int v)
{
   if (items >= n)
       error();
   if (++rear >= n) rear = 0;
   buf[rear] = v;
   items++;
}
```

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         */
    if (++sp->front >= sp->n)   /* Increment index (mod n) */
        sp->front = 0;
    item = sp->buf[sp->front];  /* Remove the item         */
    V(&sp->mutex);              /* Unlock the buffer       */
    V(&sp->slots);              /* Announce available slot */
    return item;
}
```

sbuf.c

```
int remove()
{
   if (items == 0) error();
   if (++front >= n) front = 0;
   int v = buf[front];
   items--;
   return v;
}
```

Demonstration

See program produce-consume.c in code directory

10-entry shared circular buffer

5 producers

- Agent i generates numbers from 20*i to 20*i – 1.
- Puts them in buffer

5 consumers

- Each retrieves 20 elements from buffer

Main program

- Makes sure each value between 0 and 99 retrieved once

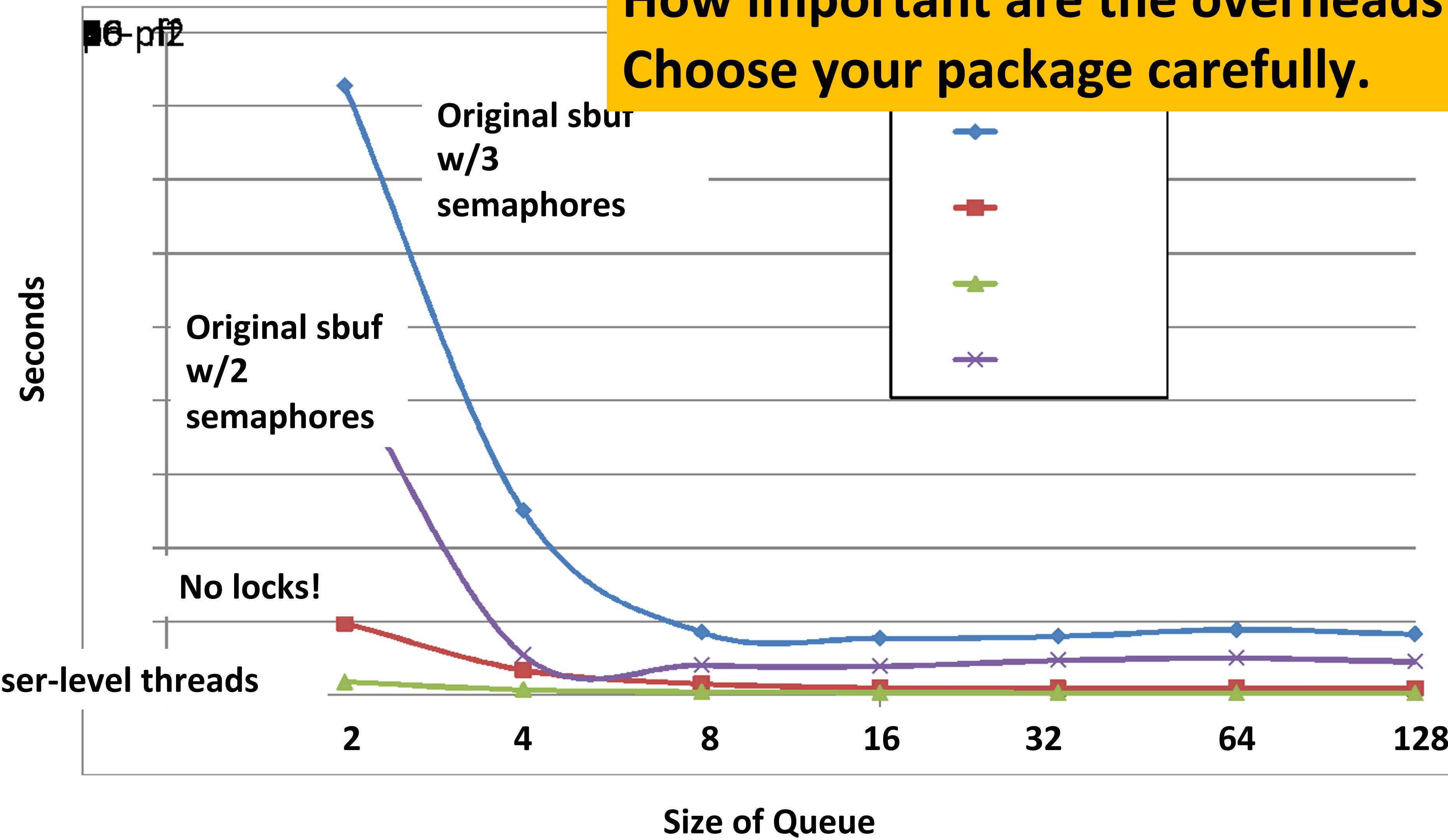
Sample program using sbuf

```
void *
producer(void *vargp)
{
  int cnt = 0;
  while (maxcnt > 0) {
    sbuf_insert(&sbuf, cnt);
    cnt++;
    maxcnt--;
  }
  sbuf_insert(&sbuf, -1);
  pthread_exit(0);
}
```

```
void *
consumer(void *vargp)
{
  int sum = 0;
  while (1) {
    int val = sbuf_remove(&sbuf);
    if (val < 0) break;
    sum += val;
  }
  total = sum;
  pthread_exit(0);
}
```

Timing

Where are the overheads?
How important are the overheads?
Choose your package carefully.



Today

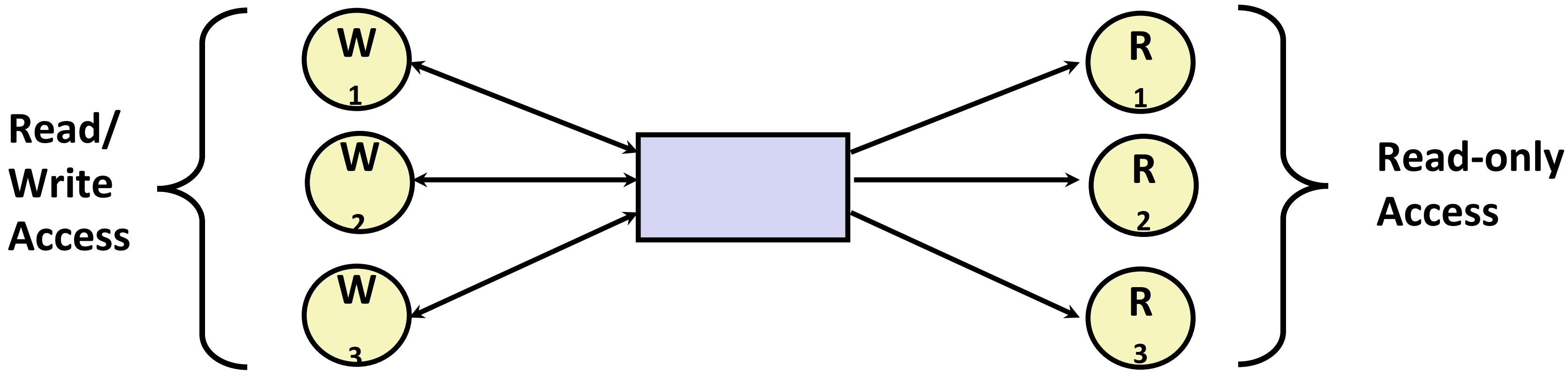
Using semaphores to schedule shared resources

- Producer-consumer problem
- **Readers-writers problem**

Other concurrency issues

- Thread safety
- Races
- Deadlocks

Readers-Writers Problem



Problem statement:

- *Reader* threads only read the object
- *Writer* threads modify the object (read/write access)
- 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; /* 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);
  }
}
```

Writers

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Solution to First Readers-Writers Problem

Readers:

```
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);
  }
}
```

Writers

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Solution to First Readers-Writers Problem

Readers:

```
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);

R1 →  /* Reading happens here */

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

Writers

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

Solution to First Readers-Writers Problem

Readers:

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

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
R2 →  if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

R1 →  /* Reading happens here */

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

Writers

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

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2

W == 0

Solution to First Readers-Writers Problem

Readers:

```
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);
R2 →
R1 →/* Reading happens here */

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

Writers

```
void writer(void)
{
  while (1) {   ← W1
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

Solution to First Readers-Writers Problem

Readers:

```
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);

R2 →  /* Reading happens here */

      P(&mutex);
      readcnt--;
      if (readcnt == 0) /* Last out */
         V(&w);
      V(&mutex);
R1 → }
}
```

Writers

```
void writer(void)
{
   while (1) {     ← W1
      P(&w);

      /* Writing here */

      V(&w);
   }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

Solution to First Readers-Writers Problem

Readers:

```
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);
  }
}
```

R3 → if (readcnt == 1)

R2 → before the second P(&mutex)

R1 → closing } of while loop

Writers

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

    /* Writing here */

    V(&w);
  }
}
```

W1 → P(&w);

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

Solution to First Readers-Writers Problem

Readers:

```
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);
  }
}
```

R3 →

R2 →

Writers

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

    /* Writing here */

    V(&w);
  }
}
```

← W1

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 1

W == 0

Solution to First Readers-Writers Problem

Readers:

```
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);
R3 →V(&mutex);
  }
}
```

Writers

```
void writer(void)
{
  while (1) {
    P(&w);          ← W1

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 0
W == 1

Demonstration

See program read-write.c

100 agents

- ~20% are writers. They write their ID to global variable
- Rest are readers. They read the global variable

Today

Using semaphores to schedule shared resources

- Producer-consumer problem
- Readers-writers problem

Other concurrency issues

- **Races**
- Deadlocks
- Thread safety

One Worry: Races

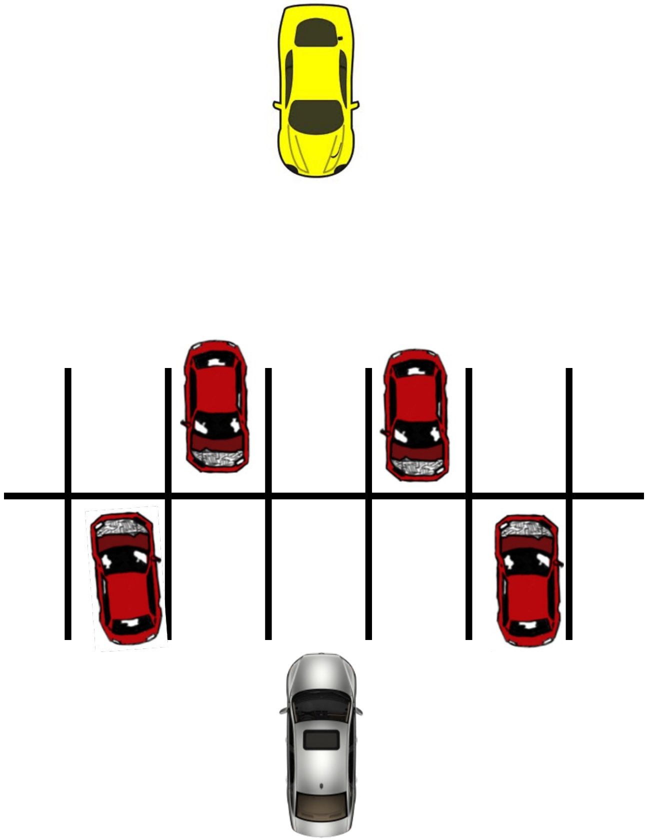
- A ***race*** occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```
/* a threaded program with a race */
int main(int argc, char** argv) {
    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);
    return 0;
}

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

race.c

Data Race



Race Elimination

Make sure don't have unintended sharing of state

```
/* a threaded program without the race */
int main(int argc, char** argv) {
    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);
    return 0;
}

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

norace.c

Today

Using semaphores to schedule shared resources
- Producer-consumer problem
- Readers-writers problem

Other concurrency issues
- Races
- **Deadlocks**
- Thread safety

A 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!

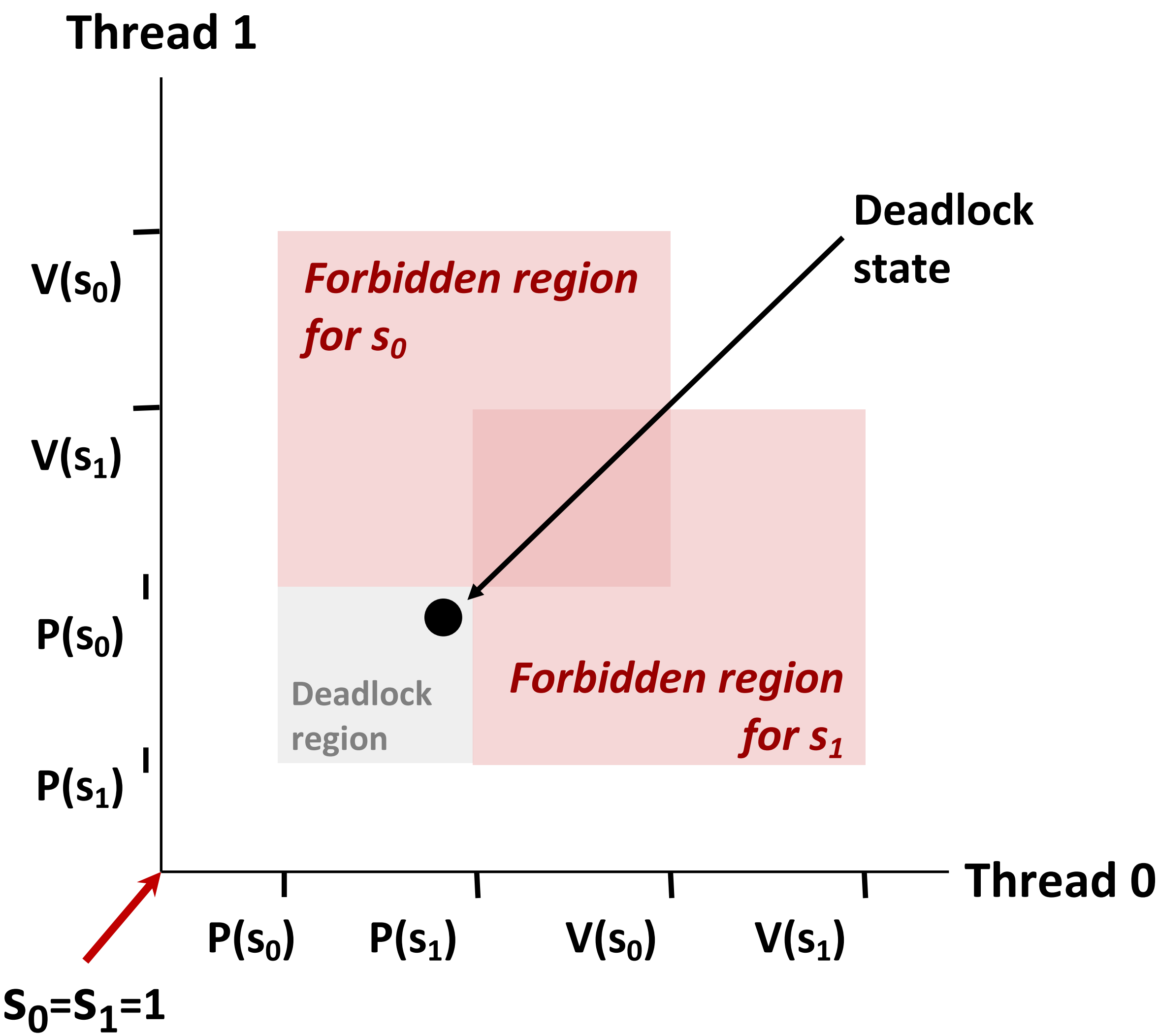
Deadlocking With Semaphores

```
int main(int argc, char** argv)
{
    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);
    return 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



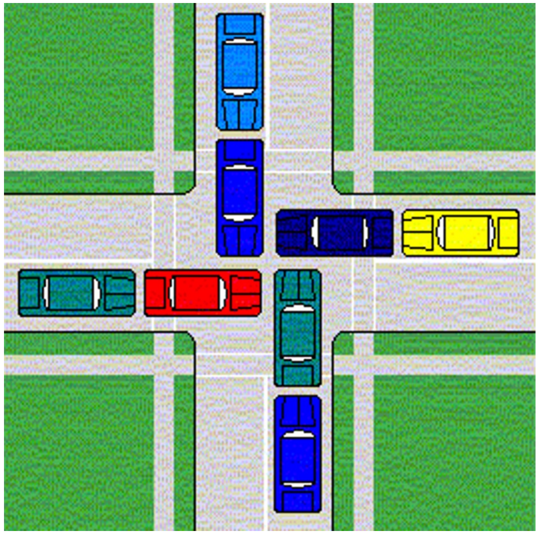
Locking introduces the potential for ***deadlock:*** waiting for a condition that will never be true

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 (race)

Deadlock



Avoiding Deadlock

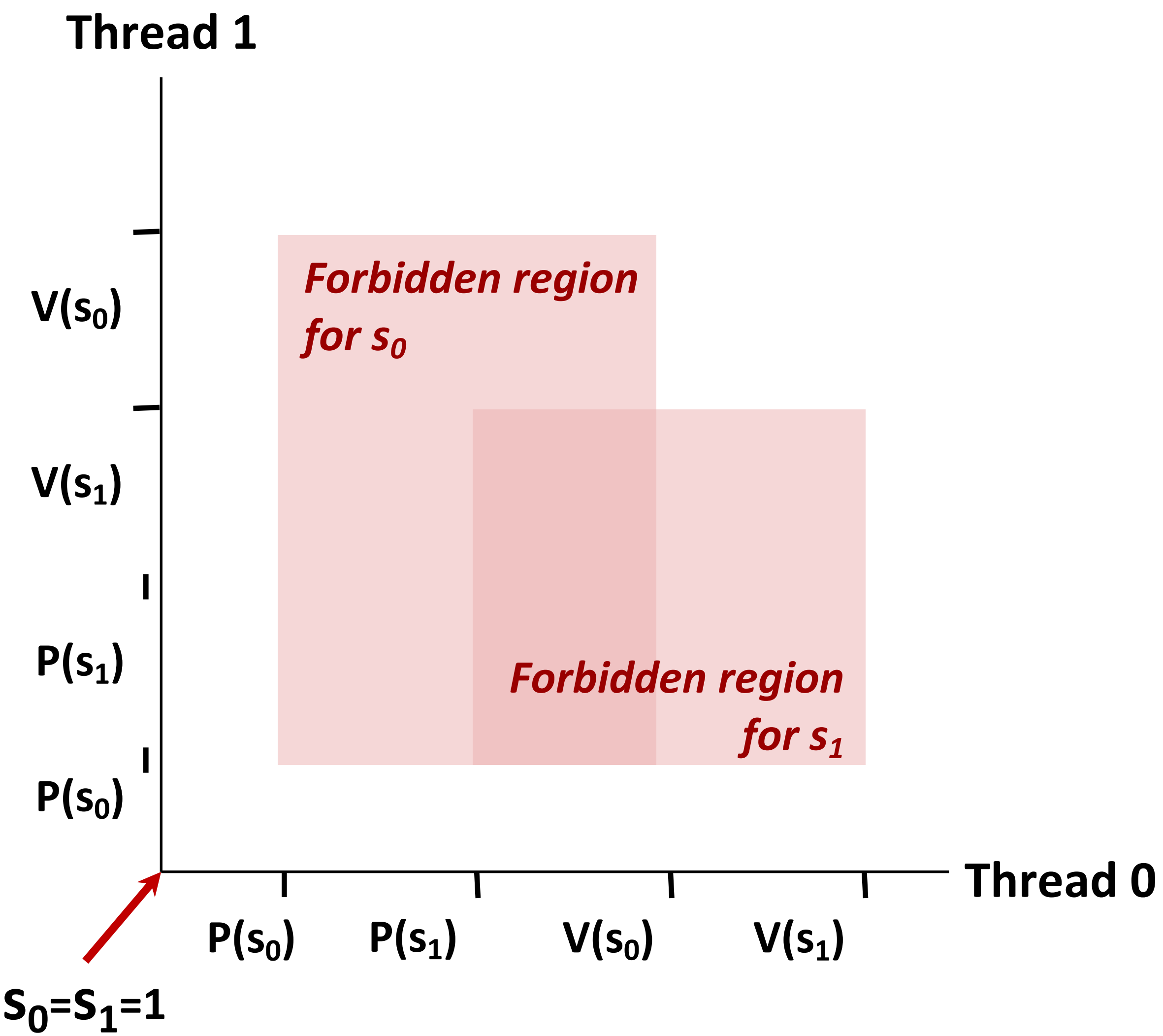
Acquire shared resources in same order

```
int main(int argc, char** argv)
{
    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);
    return 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]:	Tid[1]:
P(s_0);	P(s_0);
P(s_1);	P(s_1);
cnt++;	cnt++;
V(s_0);	V(s_1);
V(s_1);	V(s_0);

Avoided Deadlock in Progress Graph



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Demonstration

- **See program deadlock.c**
- **100 threads, each acquiring same two locks**
- **Risky mode**
  - Even numbered threads request locks in opposite order of odd-numbered ones
- **Safe mode**
  - All threads acquire locks in same order

Today

Using semaphores to schedule shared resources

- Producer-consumer problem
- Readers-writers problem

Other concurrency issues

- Races
- Deadlocks
- **Thread safety**

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 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.

```
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    sprintf(buf, "%d", x);
    return buf;
}
```

```
char *lc_itoa(int x, char *dest)
{
    P(&mutex);
    strcpy(dest, itoa(x));
    V(&mutex);
    return dest;
}
```

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

Thread-Unsafe Functions (Class 4)

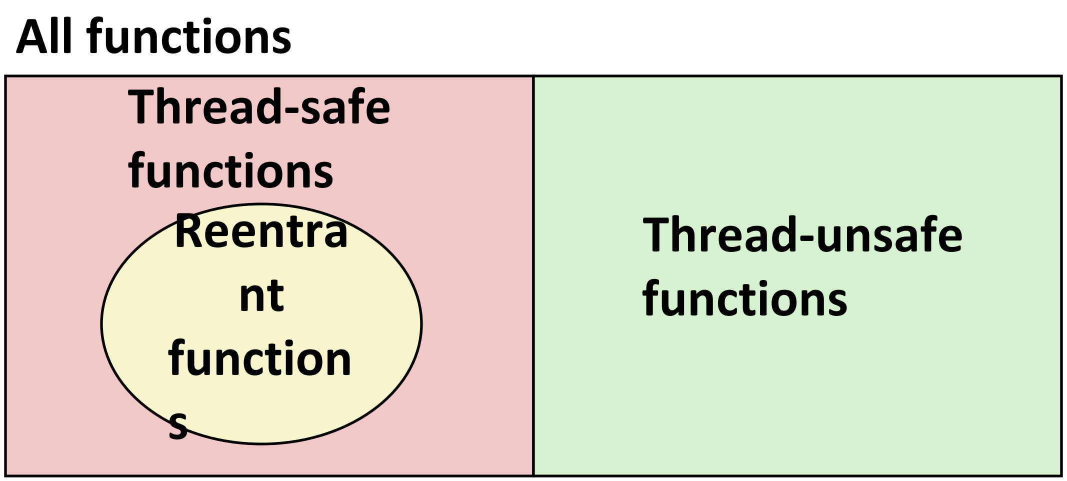
Calling thread-unsafe functions

- Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
- Fix: 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 reetnrant (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:**

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`

Threads Summary

Threads provide another mechanism for writing concurrent programs

Threads are growing in popularity

- Somewhat cheaper than processes
- Easy to share data between threads

However, the ease of sharing has a cost:

- Easy to introduce subtle synchronization errors
- Tread carefully with threads!

For more info:

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