Operating Systems

Process Synchronization

Session 5

A Biswas

Process Synchronization

Process Synchronization

- **The Critical-Section Problem**
- **Peterson's Solution**
- **Bakery Algorithm**
- **Synchronization Hardware**
- **Semaphores**
- **Classic Problems of synchronization**

Critical-Section Problem

Producer-Consumer Problem:

Producer Process: produces information

Consumer Process: consumes the information

- ***unbounded-buffer*** **places no practical limit on the size of the buffer**
- ***bounded-buffer*** **assumes that there is a fixed buffer size**

Critical-Section Problem

Shared data:

```
#define BUFFER_SIZE 10
typedef struct
{
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
```

Critical-Section Problem

Bounded-Buffer:

Producer:

```
while (true)
 {
     /*  produce an item and
     put in nextProduced  */

    while (count == BUFFER_SIZE)
             ; // do nothing
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Consumer:

```
while (true)
{
      while (count == 0)
               ; // do nothing
      nextConsumed =  buffer[out];
      out = (out + 1) %
BUFFER_SIZE;
      count--;

          /*  consume the item in
          nextConsumed */
}
```

Count is used by both producer and consumer process

Critical-Section Problem

Race Condition:

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Critical-Section Problem

Each process is designed with an entry section followed by the critical section of that process. The exit section which follows the critical section is meant for relinquishing the right to modify the shared data. In the entry section, the process acquires the permission to modify the shared data. It is obvious that when a process executes in the CS, no other process will execute in their critical section

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

Every process will have its Own critical section involving the shared data/variable. It is not necessary that the critical section is similar for all processes, its only required that each critical section involves the shared data/variable.

Solution to the Critical-Section Problem

Solution must satisfy the following three conditions:

1. Mutual Exclusion - **If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.**

1. Progress - **If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.**

1. Bounded Waiting - **A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.**

- **Assume that each process executes at a nonzero speed**
- **No assumption concerning relative speed of the N processes**

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

- The two processes share two variables:
  - int turn;
  - Boolean flag[2]

- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section.

flag[i] = true implies that process P_i is ready!

Peterson's Solution

<u>Algorithm for Process P_i</u>

```
while (true)
{

        flag[i] = TRUE;
        turn = j;

        while ( flag[j]  &&  turn == j);

              CRITICAL SECTION

        flag[i] = FALSE;

                REMAINDER SECTION

}
```

<u>Algorithm for Process P_j</u>

```
while (true)
{

        flag[j] = TRUE;
        turn = i;

        while ( flag[i]  &&  turn == i);

              CRITICAL SECTION

        flag[j] = FALSE;

                REMAINDER SECTION

}
```

Peterson's Solution

Algorithm for Process P_1

```
while (true)
{
          flag[1] = TRUE;
          turn = 2;

          while ( flag[2]  &&  turn == 2);

              CRITICAL SECTION

          flag[1] = FALSE;

               REMAINDER SECTION

}
```

Algorithm for Process P_2

```
while (true)
{
          flag[2] = TRUE;
          turn = 1;

          while ( flag[1]  &&  turn == 1);

              CRITICAL SECTION

          flag[2] = FALSE;

               REMAINDER SECTION

}
```

1. Mutual Exclusion 2. Progress 3. Bounded Waiting

Peterson's Solution

Mutual Exclusion is preserved:

Let us assume P_i execute in its critical section.

P_i can only execute in its CS only if

either flag[j] = false or turn = I

If P_1 and P_2 could not have successfully executed their while statement because the value of turn can either be 1 or 2 but cannot be both.

Hence, only one process can execute the while statement successfully.

Say P_1 executes it successfully.

P_2 will spin on the while loop as long as the P_1 executes its CS.

Thus, mutual exclusion is preserved.

Multiple-Process Solutions

Bakery Algorithm:

Critical section for n processes:

- **Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.**

- **If processes Pi and Pj receive the same number,**
  if i < j
  Pi is served first;
  else
  Pj is served first.

- **The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...**

Multiple-Process Solutions

Bakery Algorithm:

- **Notation: lexicographical order (ticket #, process id #)**
  - **(a,b) < (c,d) if a < c or if a = c and b < d**
  - **max(a_0, . . . , $a_{(n-1)}$) is a number, k , such that k >= a_i for i = 0, . . . , n − 1**
- **Shared data**

  var **choosing:** array **[0..n−1]** of **boolean;**

  number: array **[0..n−1]** of **integer;**

- **Data structures are initialized to false and**

Multiple-Process Solutions

Bakery Algorithm:

```
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n − 1])+1;
    choosing[i] := false;
    for j := 0 to n − 1
        do begin
            while choosing[j] do no-op;
            while number[j] != 0
                and (number[j],j) < (number[i], i) do no-op;
        end;

        critical section

    number[i] := 0;

        remainder section

until false;
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndndSet Instruction

- **Definition:**

```
boolean TestAndSet (boolean *target)
 {
     boolean rv = *target;
     *target = TRUE;
     return rv:
 }
```

If two testAndSet instructions are simultaneously each on a different CPU, they are sequentially in some arbitrary order.

Solution using TestAndndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
while (true)
{
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

Solution using TestAndndSet

Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution:

```
while (true)
{
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

Swap Instruction

- **Definition:**

```
void Swap (boolean *a, boolean *b)
 {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key.
- Solution:

```
while (true)
{
        key = TRUE;
        while ( key == TRUE)
               Swap (&lock, &key );

                   //    critical section

        lock = FALSE;

                   //      remainder section

 }
```

Solution using Swap

- **Definition:**

```
  void Swap (boolean *a, boolean *b)
  {
        boolean temp = *a;
        *a = *b;
        *b = temp:
  }
```

- Shared Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key.
- **Solution:**

```
while (true)
{
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section

        lock = FALSE;

                //    remainder section

}
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S)

```
{
    while S <= 0;  // no-op
      S--;
}
```

  - signal (S)

```
{
  S++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

    Critical Section

    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Spin lock

- Software solution and the semaphore definition given here involves busy waiting.
- This type of semaphore which involves busy waiting is called spinlock.
- A spinlock avoids context switch hence useful in case of short critical section.
- Useful in multiprocessors, a thread busy-waits on a resource on one processor while another process uses the resource on a different processor.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

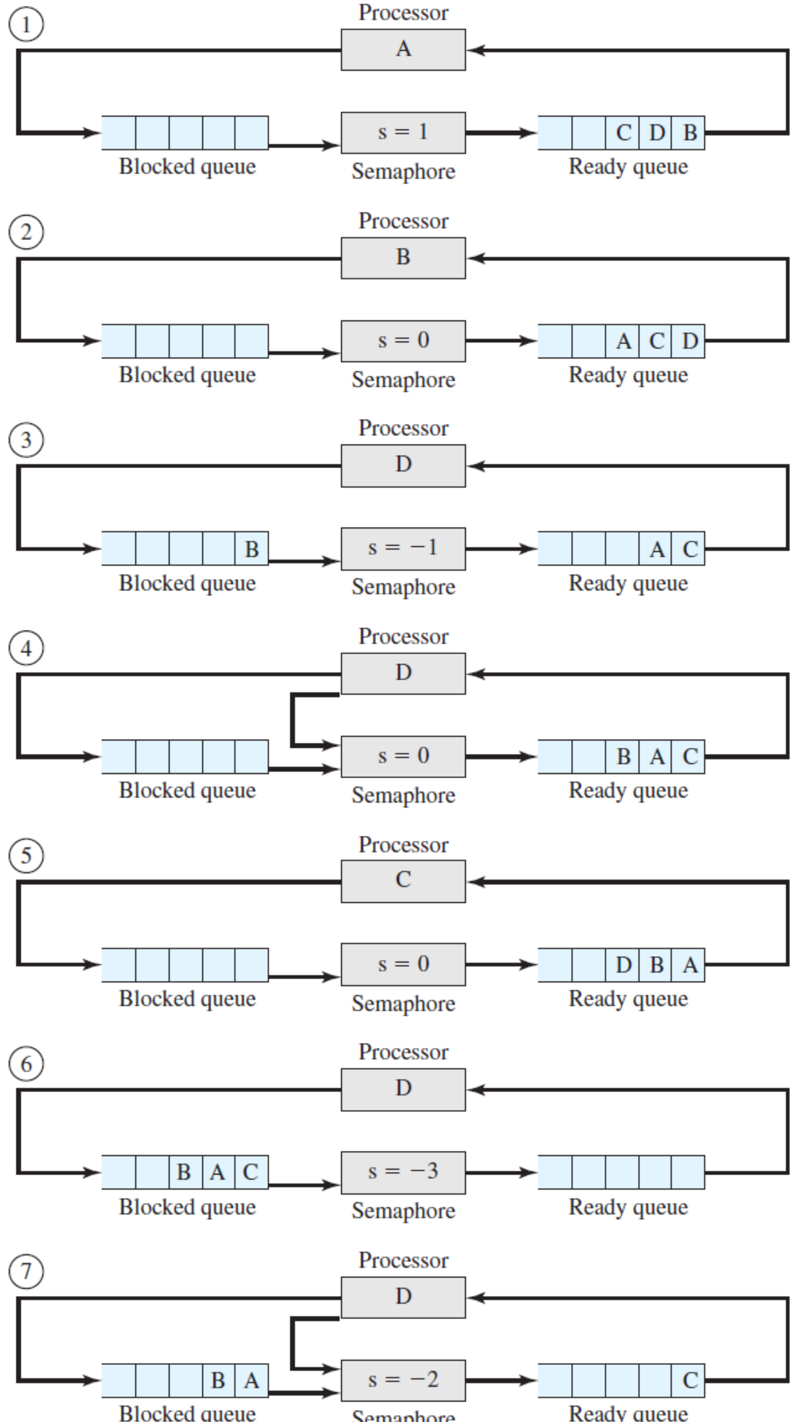
```
wait (S)
{
    value--;
    if (value < 0)
    {
        add this process to waiting queue
        block();
    }
}
```

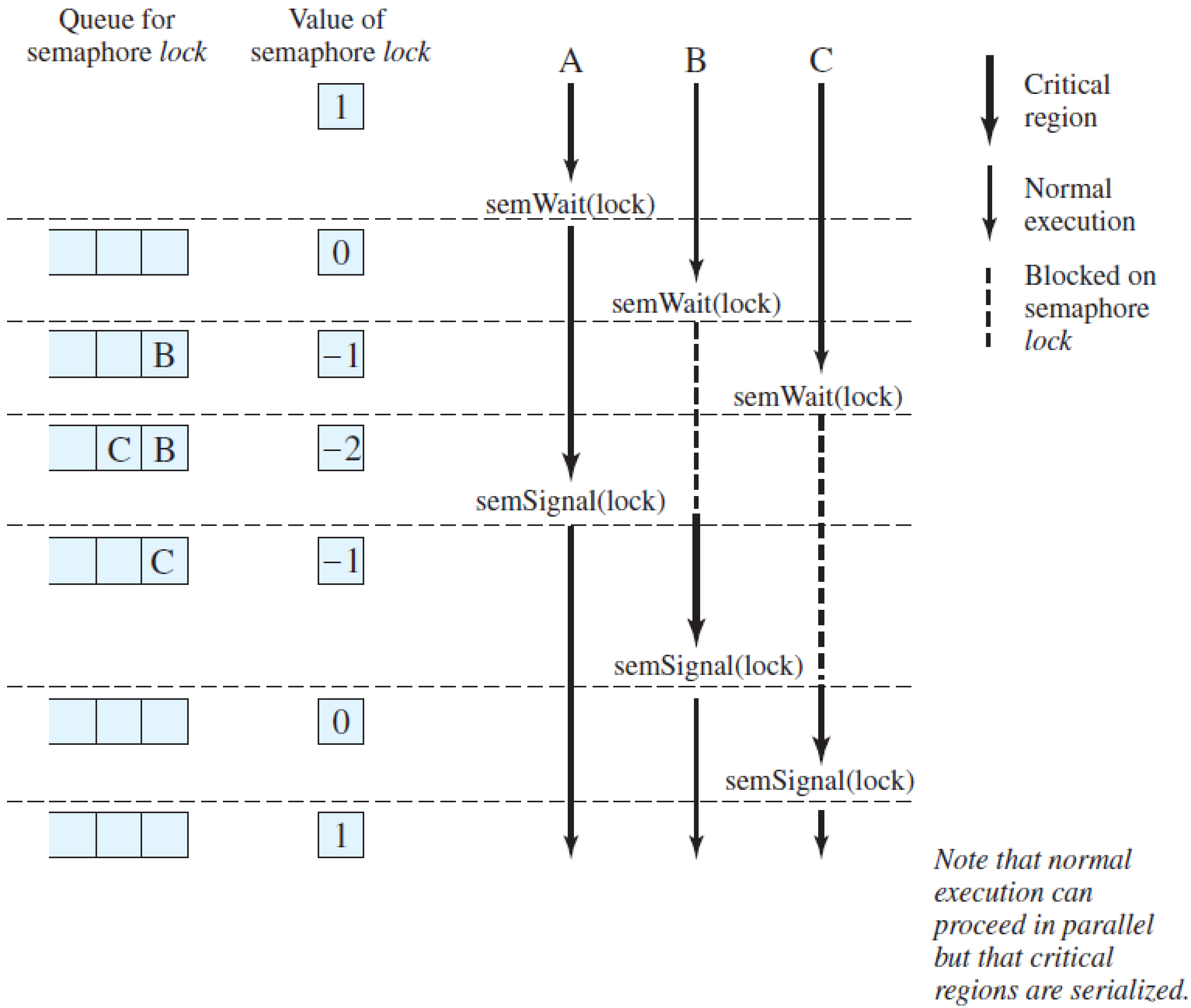
- Implementation of signal:

```
Signal (S)
{
    value++;
    if (value <= 0) {
        remove a process P from waiting queue
        wakeup(P);  }
}
```

Binary Semaphore Definitions

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
           s.value = zero;
      else {
                 /* place this process in s.queue */;
                 /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
           s.value = one;
      else {
                 /* remove a process P from s.queue */;
                 /* place process P on ready list */;
      }
}
```


1
Processor
A
s = 1
C D B
Blocked queue
Semaphore
Ready queue
2
Processor
B
s = 0
A C D
Blocked queue
Semaphore
Ready queue
3
Processor
D
B
s = −1
A C
Blocked queue
Semaphore
Ready queue
4
Processor
D
s = 0
B A C
Blocked queue
Semaphore
Ready queue
5
Processor
C
s = 0
D B A
Blocked queue
Semaphore
Ready queue
6
Processor
D
B A C
s = −3
Blocked queue
Semaphore
Ready queue
7
Processor
D
B A
s = −2
C
Blocked queue
Semaphore
Ready queue



Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
−1
semWait(lock)
C B
−2
semSignal(lock)
C
−1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore *lock*
Note that normal execution can proceed in parallel but that critical regions are serialized.

Deadlock and Starvation

- **Deadlock** – two or or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- ***N*** **buffers, each can hold one item**
- **Semaphore mutex initialized to the value 1**
- **Semaphore full initialized to the value 0**
- **Semaphore empty initialized to the value N.**

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true)  {

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

Bounded Buffer Problem (Cont.)

- **The structure of the consumer process**

```
while (true)
{
        wait (full);
        wait (mutex);

                //  remove an item from  buffer

        signal (mutex);
        signal (empty);

         //  consume the removed item
 }
```

Bounded Buffer Problem (Cont.)

Producer:

```
while (true)
{

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the buffer

    signal (mutex);
    signal (full);
}
```

Consumer:

```
while (true)
{
        wait (full);
        wait (mutex);
        //remove an item from buffer
        signal (mutex);
        signal (empty);
            //  consume the removed item
}
```

```c
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
```

```c
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
```

Solution to the Infinite-Buffer Producer/Consumer Problem

```
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
```

```
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
```

Solution to the Infinite-Buffer Producer/Consumer Problem

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.

- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- **The structure of a writer process**

```
while (true)
{
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
  }
```

Readers-Writers Problem (Cont.)

- **The structure of a reader process**

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex);

                // reading is performed

         wait (mutex) ;
         readcount  - - ;
         if (readcount  == 0)  signal (wrt) ;
         signal (mutex) ;
}
```

Readers-Writers Problem (Cont.)

Writer process

```
while (true)
{
    wait (wrt) ;

    //    writing is performed

    signal (wrt) ;
}
```

Reader process

```
while (true)
{
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
            wait (wrt) ;
    signal (mutex)

    // reading is performed

    wait (mutex) ;
    readcount  - - ;
    if (readcount  == 0)
            signal (wrt) ;
    signal (mutex) ;
}
```

Writers have priority:

No new readers are allowed once at least one writer has declared the desire to write.

Writers have priority:

- A semaphore **rsem** that inhibits all readers while there is at least one writer desiring access to the data area
- A variable **writecount** that controls the setting of rsem
- A semaphore **y** that controls the updating of writecount

Readers/Writers w/ Writers Priority (Using Semaphores)

Reader:

```
P(mutex);
if (AW+WW > 0)
    WR++;
else {
    V(OKToRead);
    AR++;
}
V(mutex);
P(OKToRead);

read database

P(mutex);
AR– –;
if (AR == 0 &&
    WW > 0) {
    V(OKToWrite);
    AW++; WW– –;
}
V(mutex);
```

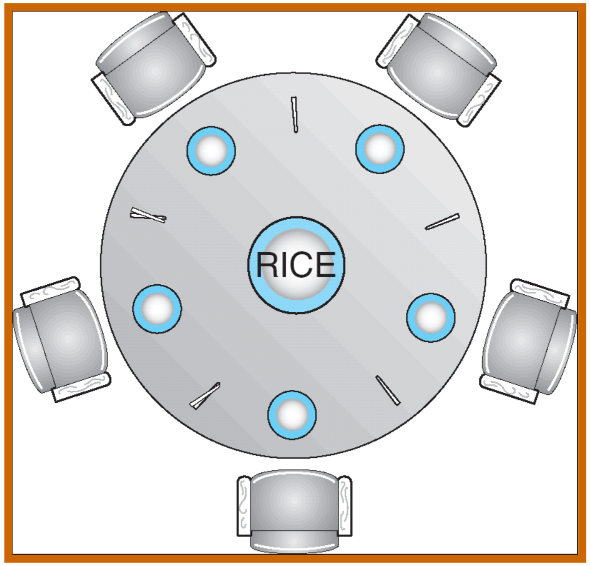
Writer:

```
P(mutex);
if (AW+AR > 0)
    WW++;
else {
    V(OKToWrite);
    AW++;
}
V(mutex);
P(OKToWrite);

write database

P(mutex);
AW– –;
if (WW > 0) {
    V(OKToWrite);
    AW++; WW– –;
} else if (WR > 0) {
    V(OKToRead);
    AR++;  WR– –;
}
V(mutex);
```

Notes on R/W w/ Writers Priority (Using Semaphores)

- Reader:
  - If there are active or waiting writers, this reader has to wait (writers have priority)
  - Otherwise, this reader can read (possibly along with other readers)
  - When the last reader finishes, if there are waiting writers, it must wake one up
- Writer:
  - If there are active readers or writers, this writer has to wait (everyone has to finish before writer can update database)
  - Otherwise, this writer can write (and has exclusive access to database)
  - When the writer finishes,
    - (first choice) if there are waiting writers, it must wake one up (writers have priority)
    - (second choice) if there are waiting readers, it must wake one up

Dining-Philosophers Problem



- **Shared data**
  - **Bowl of rice (data set)**
  - **Semaphore chopstick [5] initialized to 1**

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
while (true)
{
     wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

               //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

           //  think
}
```

End of Session 5

Shared Variable in Threaded Programs

Multiple threads can share the same program variables.

```
#include "csapp.h"
#define N 2

char **ptr; /* global variable */

void *thread(void *vargp);

int main()
{
        int i;
        pthread_t tid;
        char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
        };

        ptr = msgs;

        for (i = 0; i < N; i++)
                pthread_create(&tid, NULL, thread, (void *)i);
        pthread_exit(NULL);
}

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

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

- **unix> ./sharing**
- **[0]: Hello from foo (cnt=1)**
- **[1]: Hello from bar (cnt=2)**

Threads Memory Model

A pool of concurrent threads runs in the context of a process.

Each thread has its own separate thread context, which includes a thread ID, stack, stack pointer, program counter, condition codes, and general purpose register values.

Threads Memory Model

Each thread shares the rest of the process context with the other threads. This includes the entire user virtual address space, which consists of read-only text (code), read/write data, the heap, and any shared library code and data areas.

Threads Memory Model

The threads also share the same set of open files and the same set of installed signal handlers.

In an operational sense, it is impossible for one thread to read or write the register values of another thread.

Mapping Variables to Memory

C variables in threaded programs are mapped to virtual memory according to their storage classes.

Global variables.

A *global variable* is any variable declared outside of a function. At run-time, the read/write area of virtual memory contains exactly one instance of each global variable that can be referenced by any thread.

Global variables.

For example, the global ptr variable in line 4 has one run-time instance in the read/write area of virtual memory. When there is only one instance of a variable, we will denote the instance by simply using the variable name, in this case ptr.

Mapping Variables to Memory

Local automatic variables.

A *local automatic variable* is one that is declared inside a function without the static attribute. At run-time, each thread's stack contains its own instances of any local automatic variables. This is true even if multiple threads execute the same thread routine.

Local automatic variables.

For example, there is one instance of the local variable tid, and it resides on the stack of the main thread. We will denote this instance as tid.m. As another example, there are two instances of the local variable myid, one instance on the stack of peer thread 0, and the other on the stack of peer thread 1. We will denote these instances as myid.p0 and myid.p1 respectively.

<u>Mapping Variables to Memory</u>

Local static variables.

A *local static variable* is one that is declared inside a function with the static attribute. As with global variables, the read/write area of virtual memory contains exactly one instance of each local static variable declared in a program.

For example, even though each peer thread in our example program declares cnt in line 27, at runtime there is only one instance of cnt residing in the read/write area of virtual memory. Each peer thread reads and writes this instance.

Mapping Variables to Memory

Shared Variables

A variable v is shared if and only if one of its instances is referenced by more than one thread.

For example, variable cnt in our example program is shared because it has only one run-time instance, and this instance is referenced by both peer threads.

On the other hand, myid is not shared because each of its two instances is referenced by exactly one thread.

However, it is important to realize that local automatic variables such as msgs can also be shared.

Progress Graphs

A *progress graph* models the execution of n concurrent threads as a trajectory through an n-dimensional Cartesian space.

Each axis k corresponds to the progress of thread k.

Each point (I1; I2; : : : ; In) represents the state where thread k, (k = 1; : : : ; n) has completed instruction Ik.

The origin of the graph corresponds to the *initial state* where none of the threads has yet completed an instruction.

```
#include "csapp.h"

#define NITERS 100000000

void *count(void *arg);

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

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;
}
```

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: The block of instructions at the head of the loop.
- L_i: The instruction that loads the shared variable `cnt` into register `%eax`$_i$, where `%eax`$_i$ deno
  value of register `%eax` in thread i.
- U_i: The instruction that updates (increments) `%eax`$_i$.
- S_i: The instruction that stores the updated value of `%eax`$_i$ back to the shared variable `cnt`.
- T_i: The block of instructions at the tail of the loop.

```
unix> ./badcnt
BOOM! ctr=198841183

unix> ./badcnt
BOOM! ctr=198261801

unix> ./badcnt
BOOM! ctr=198269672
```

Instructions can be interleaved in any order, so long as the instructions for each thread execute in program order.

For example, the ordering
H1;H2; L1; L2; U1; U2; S1; S2; T1; T2

is sequentially consistent, while the ordering

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

is not sequentially consistent because U1 executes before L1. Unfortunately not all sequentially consistent orderings are created equal. Some will produce correct results, but others will not, and there is no way for us to predict whether the operating system will choose a correct ordering for our threads.

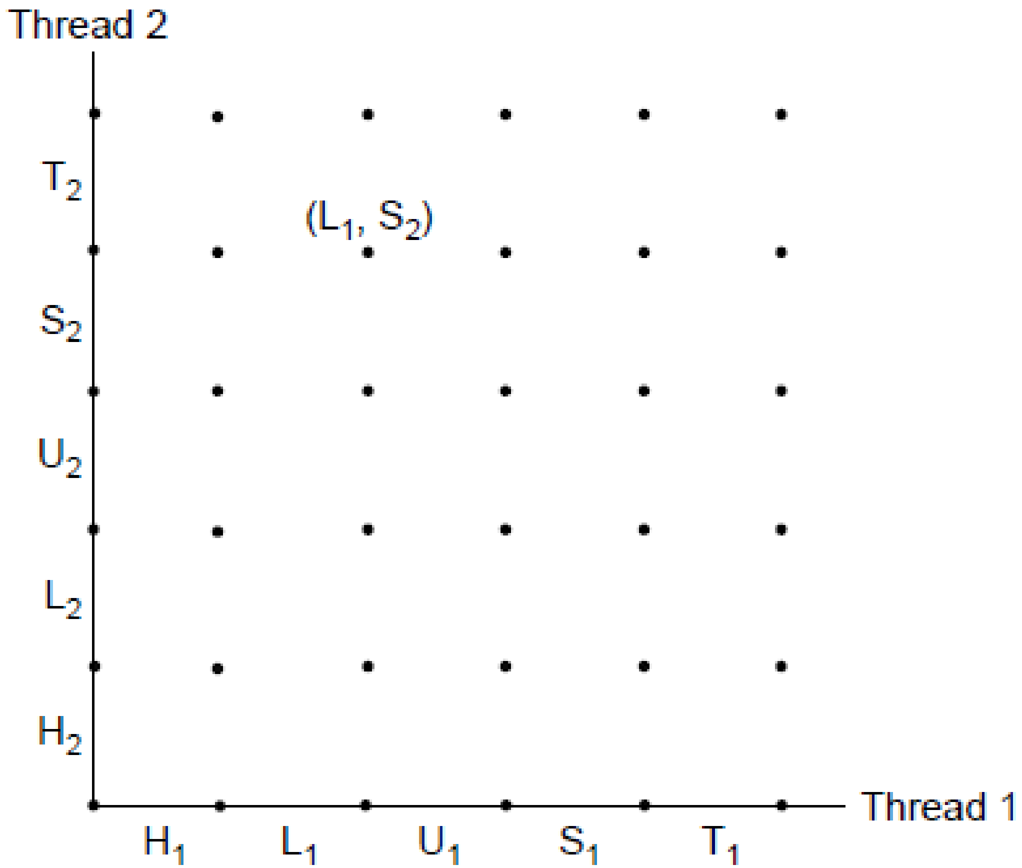


Figure 11.11: **Progress graph for the first loop iteration of** `badcnt.c`**.**

The horizontal axis corresponds to thread 1,
the vertical axis to thread 2.
Point (L1; S2) corresponds to the state where thread 1 has completed L1 and thread 2 has completed S2.

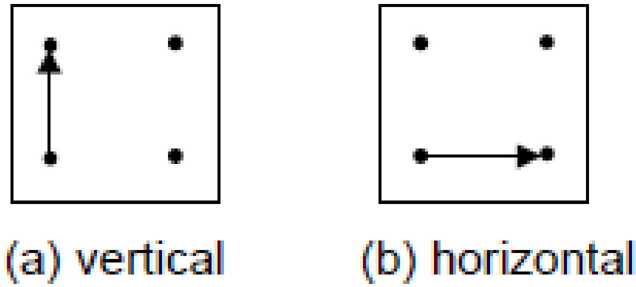


Figure 11.12: **Legal transitions in a progress graph.**

For the single-processor systems that we are concerned about, where instructions complete one at a time in sequentially-consistent order, legal transitions move to the right (an instruction in thread 1 completes) or up (an instruction in thread 2 completes).

Programs never run backwards, so transitions that move down or to the left are not legal.

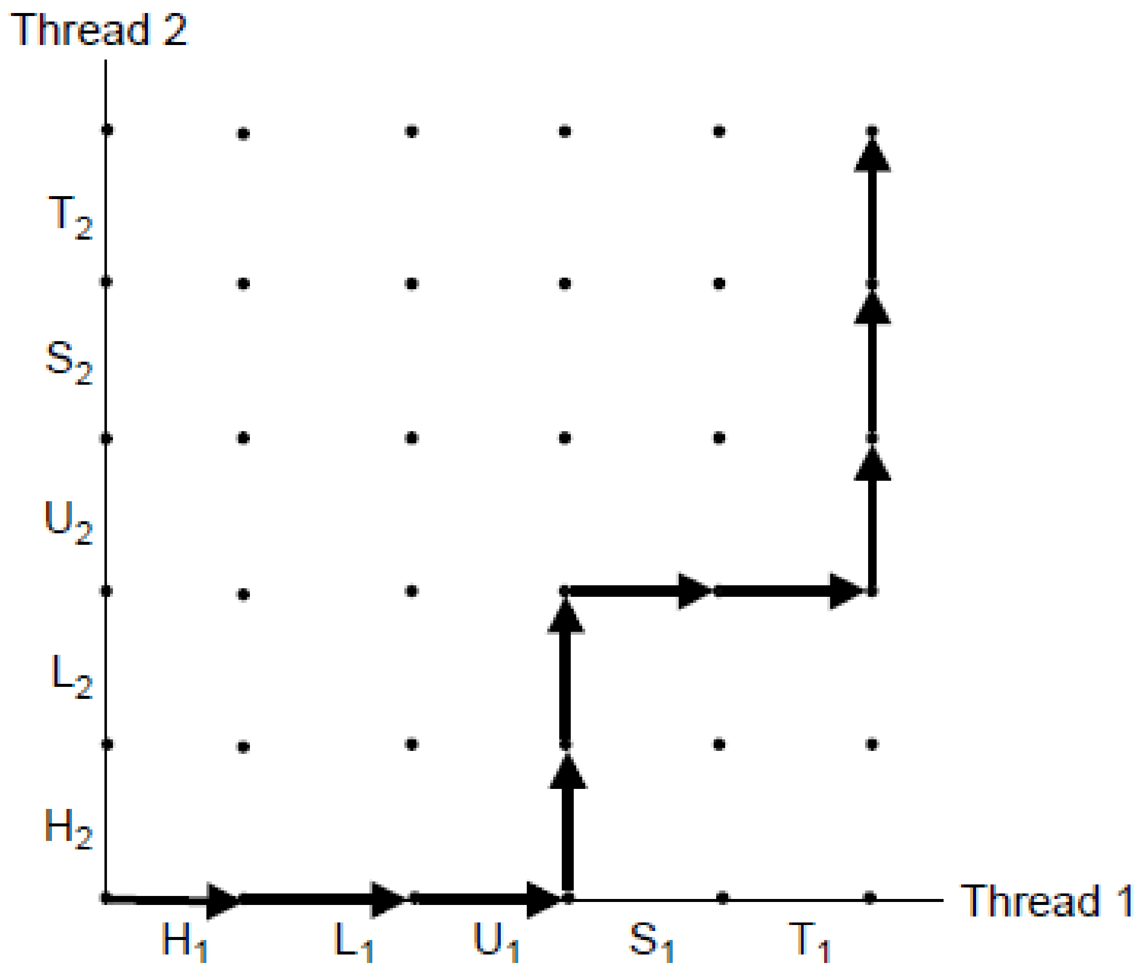


Figure 11.13: **An example trajectory.**

$$H_1, L_1, U_1, H_2, L_2, S_1, T_1, U_2, S_2, T_2.$$

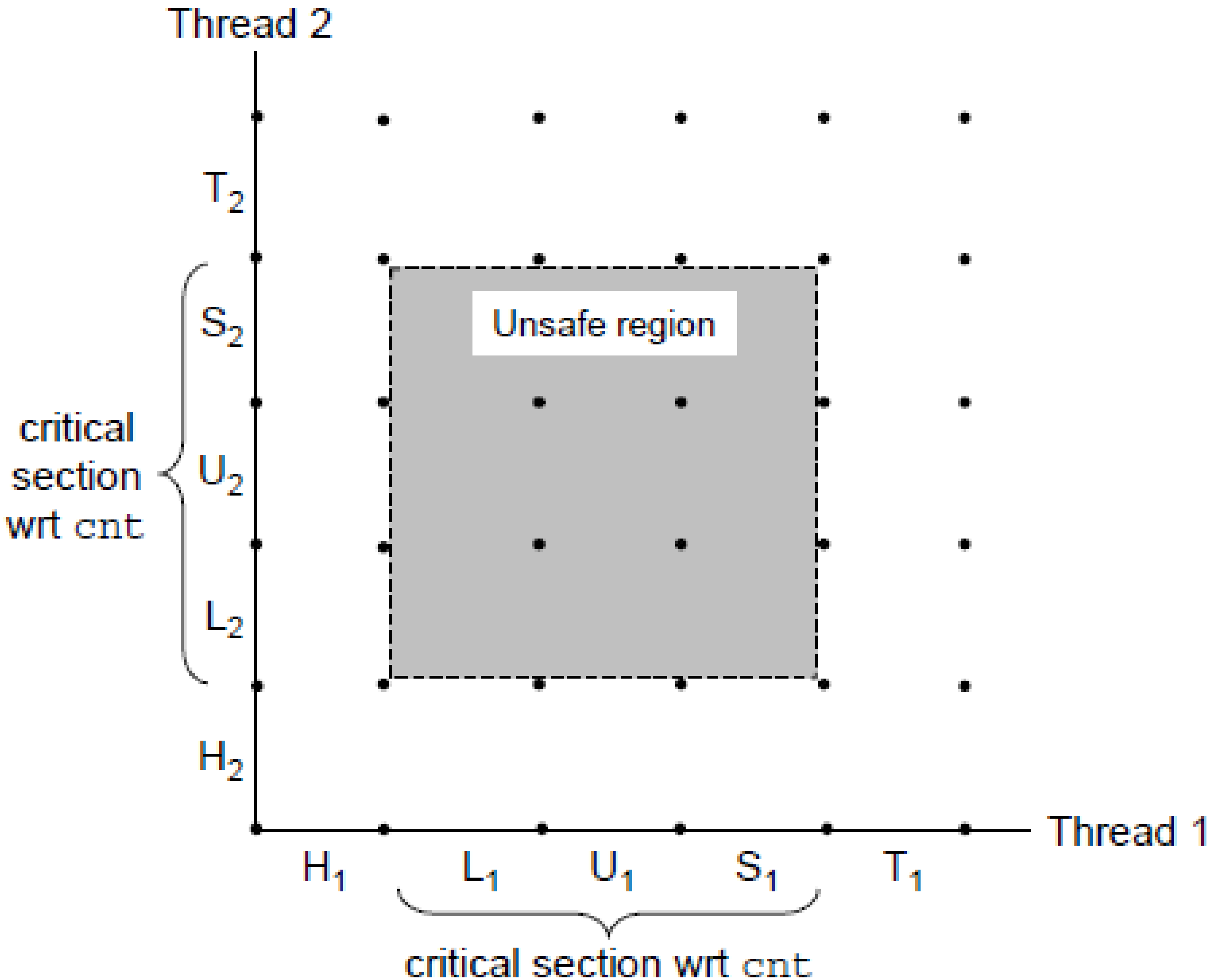


Figure 11.14: **Critical sections and unsafe regions.**

For thread i, the instructions (Li; Ui; Si) that manipulate the contents of the shared variable cnt constitute a *critical section* (with respect to shared variable cnt) that should not be interleaved with the critical section of the other thread.

The intersection of the two critical sections defines a region of the state space known as an *unsafe region*.

Figure 11.14 shows the unsafe region for the variable cnt. Notice that the unsafe region abuts, but does not include, the states along its perimeter. For example, states (H1;H2) and (S1; U2) abut the unsafe region, but are not a part of it.

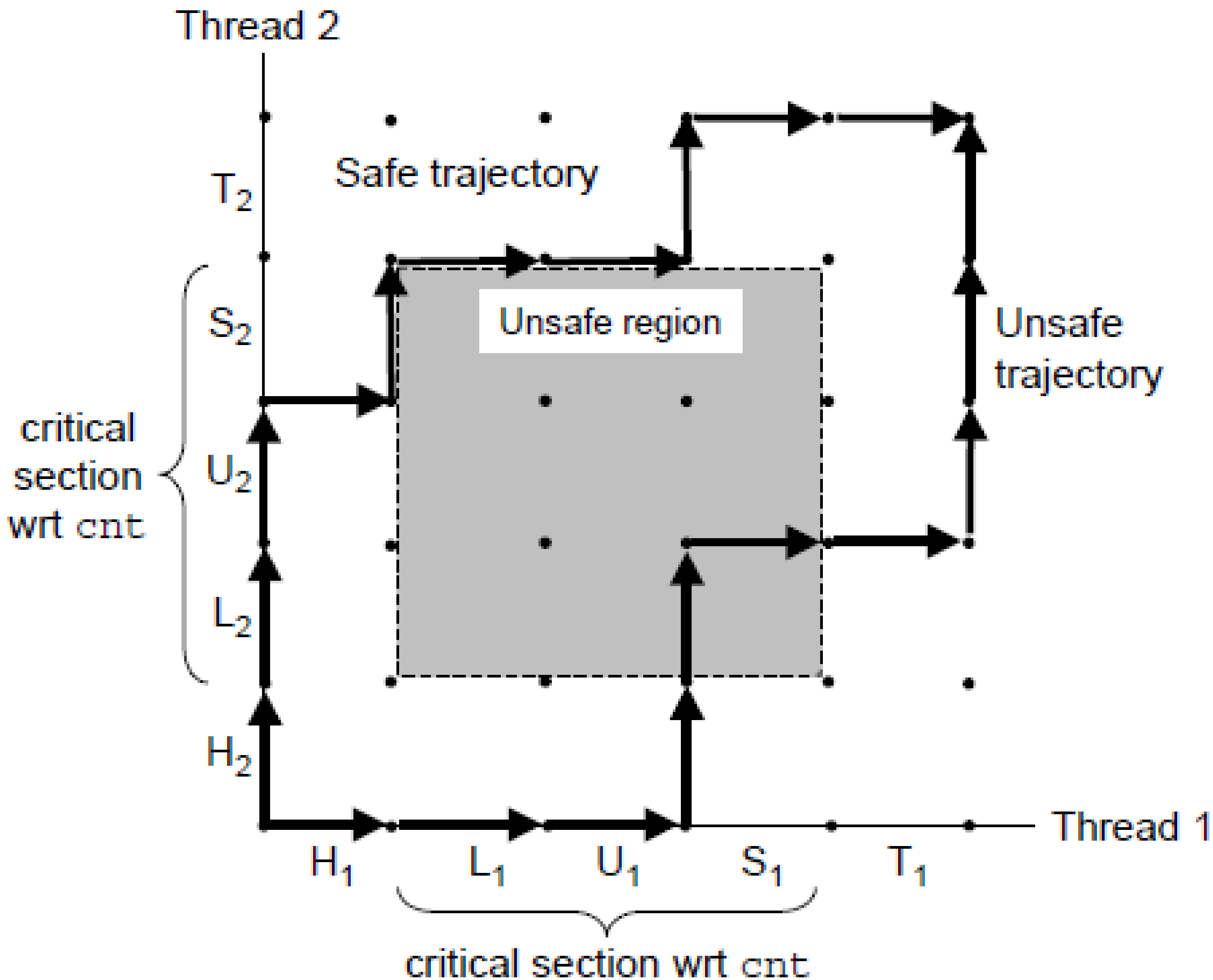
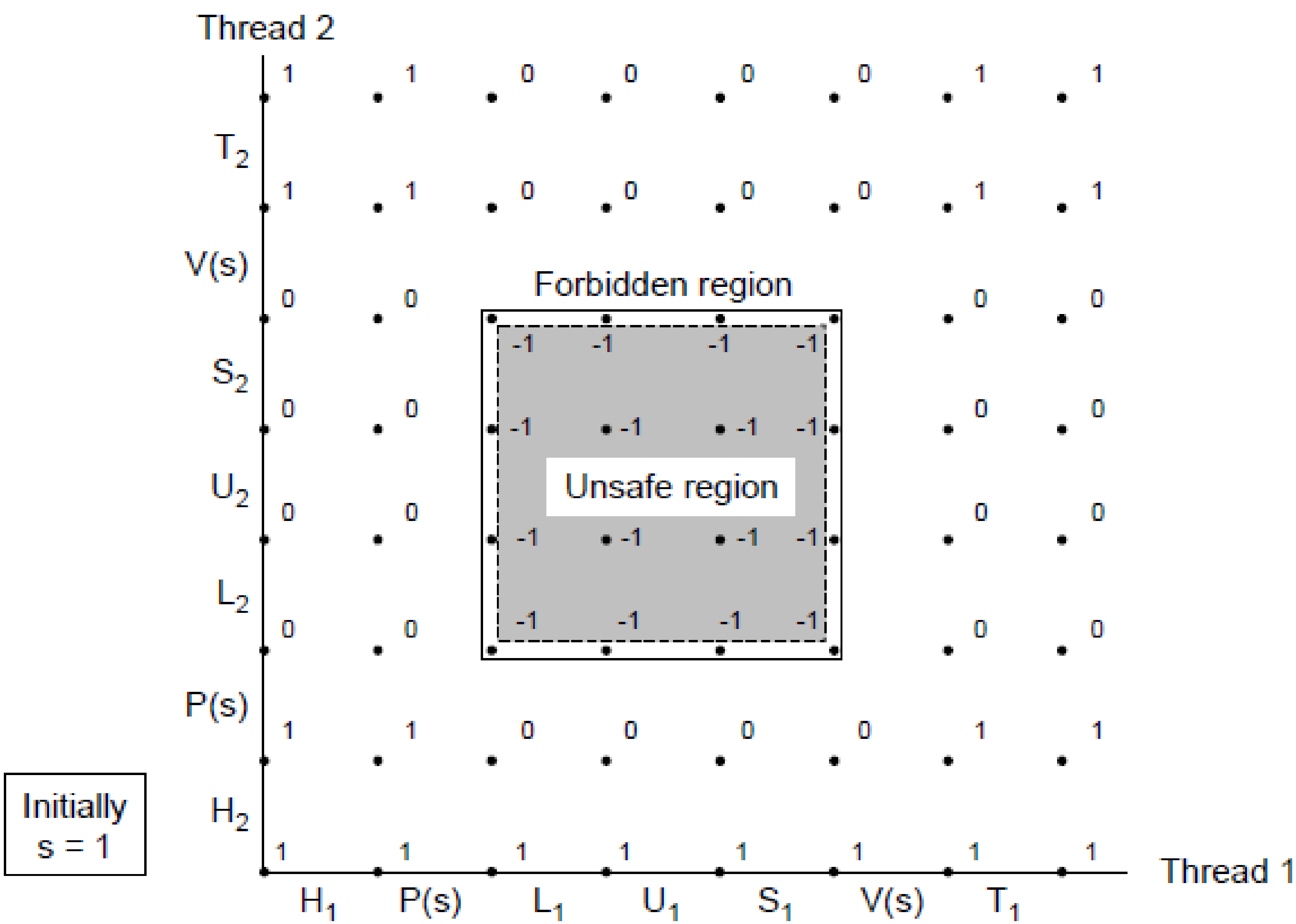
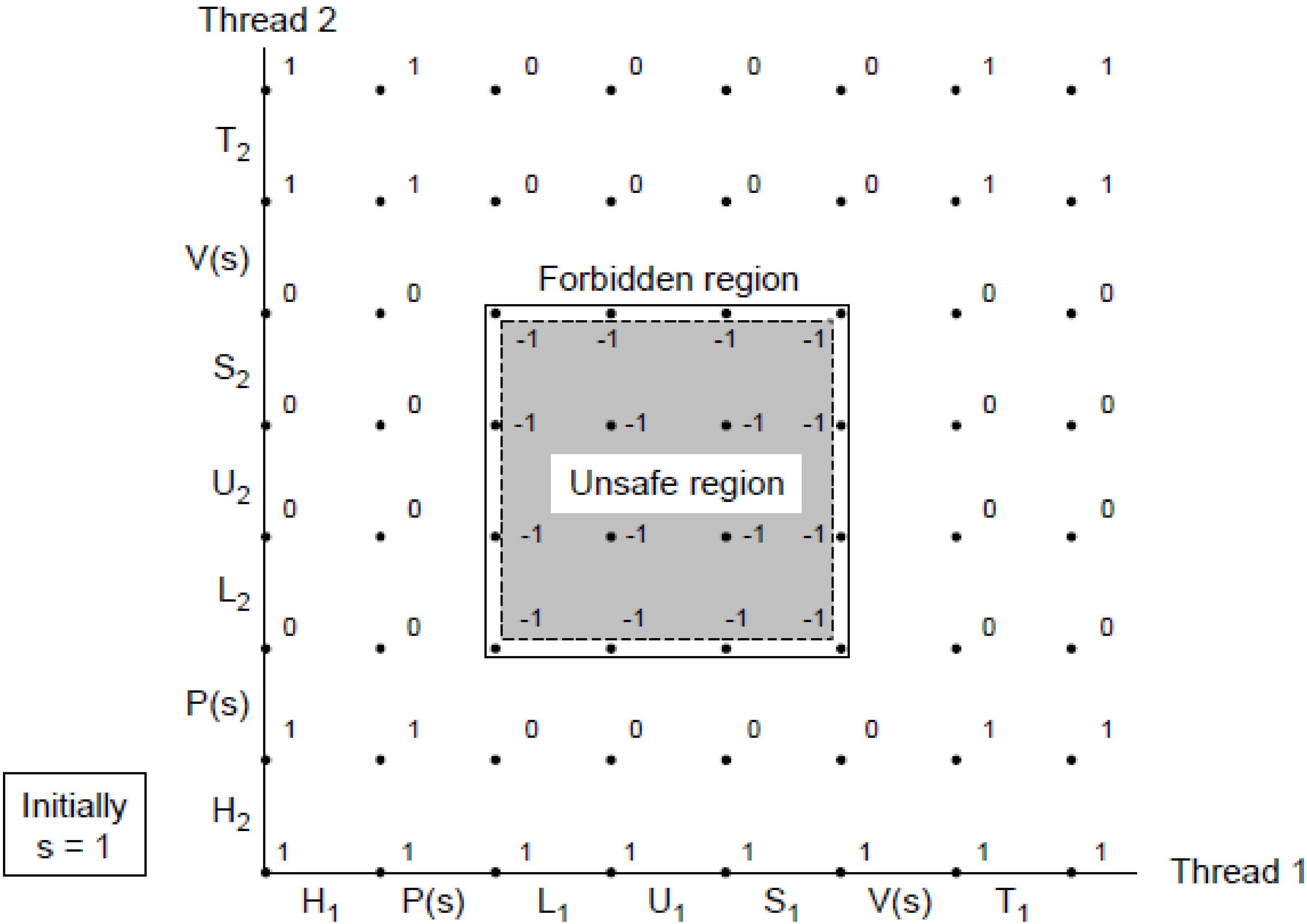


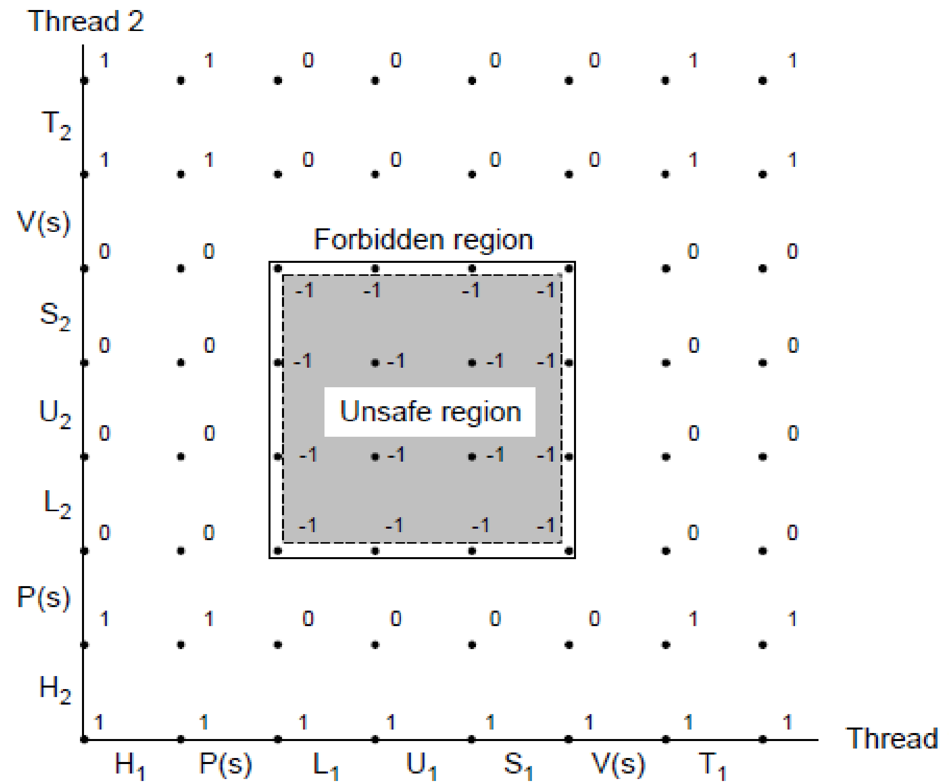
Figure 11.15: **Safe and unsafe trajectories.**

The basic idea is to associate a semaphore s, initially 1, with each shared variable (or related set of shared variables) and then surround the corresponding critical section with P(s) and V (s) operations.





Thread 2
T2
V(s)
S2
U2
L2
P(s)
H2
Forbidden region
Unsafe region
Initially
s = 1
H1
P(s)
L1
U1
S1
V(s)
T1
Thread 1



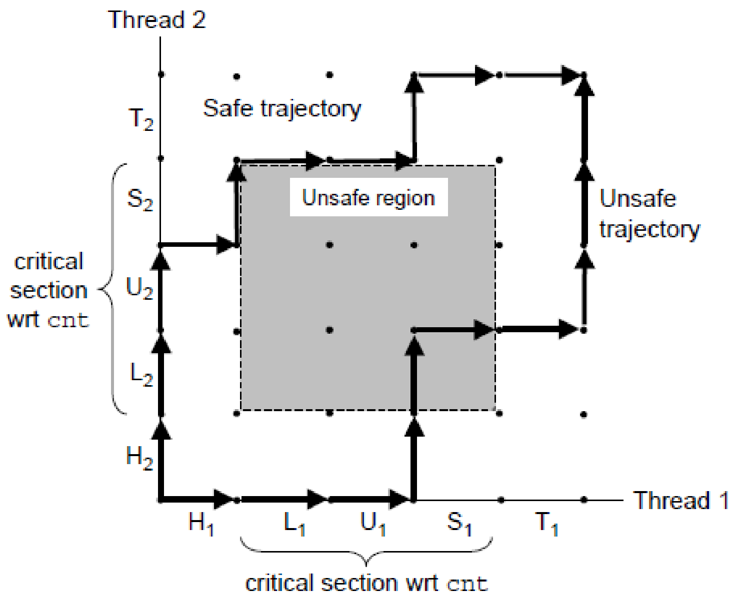


Figure 11.15: **Safe and unsafe trajectories.**

In the figure, each state is labeled with the value of semaphore s in that state.

The crucial idea is that this combination of P and V operations creates a collection of states, called a *forbidden region*, where s < 0.

Because of the semaphore invariant, no feasible trajectory can include one of the states in the forbidden region.

And since the forbidden region completely encloses the unsafe region, no feasible trajectory can touch any part of the unsafe region.

Thus, every feasible trajectory is safe, and regardless of the ordering of the instructions at runtime, the program correctly increments the counter.

A primary aim of an operating system is to share a computer installation among many programs making unpredictable demands upon its resources.

Designer should try to construct separate schedulers for each class of resource.

A primary task of its designer is therefore to construct resource allocation (or scheduling) algorithms for resources of various kinds (main store, drum store, magnetic tape handlers, consoles, etc.).

Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources.

Such a collection of associated data and procedures is known as a *monitor.*

Monitor

A monitor is an object intended to be used safely by more than one thread.

Monitor

Its methods are executed with mutual exclusion. At each point in time, at most one thread may be executing any of its methods.

Monitor

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Monitor

Monitors also have a mechanism for signaling other threads that such conditions have been met.

Mutual Exclusion

While a thread is executing a method of a monitor, it is said to *occupy* the monitor.

Mutual exclusion property: *at each point in time, at most one thread may occupy the monitor*.

Upon calling one of the methods, a thread must wait until no thread is executing any of the monitor's methods before starting execution of its method.

In a simple implementation, mutual exclusion can be implemented by the compiler equipping each monitor object with a private lock, often in the form of a semaphore.
This lock is initially unlocked, is locked at the start of each public method, and is unlocked at each return from each public method.

```
monitor class Account
{
        private int balance := 0
        invariant balance >= 0

        public method boolean withdraw(int amount)
        {
                if amount < 0 then error "Amount may not be negative"
                else if balance < amount then return false
                else { balance := balance - amount ; return true }
        }

        public method deposit(int amount)
        {
                if amount < 0 then error "Amount may not be negative"
                else balance := balance + amount
        }
}
```

Waiting and Signaling

For many applications, mutual exclusion is not enough.

Threads attempting an operation may need to wait until some assertion *P* holds true.

A busy waiting loop

while not (*P*) do skip

will not work, as mutual exclusion will prevent any other thread from entering the monitor to make the condition true.

Waiting and Signaling

Condition variables:

A condition variable is a queue of threads, associated with a monitor, upon which a thread may wait for some assertion to become true.

Thus each condition variable *c* is associated with some assertion *Pc*.

While a thread is waiting upon a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state.

In most types of monitors, these other threads may signal the condition variable *c* to indicate that assertion *Pc* is true.

Two main operations on conditions variables:
wait c is called by a thread that needs to wait until the assertion *Pc* to be true before proceeding.
signal c (sometimes written as notify c) is called by a thread to indicate that the assertion *Pc* is true.

```
monitor class Semaphore
{
        private int s := 0
        invariant s >= 0
        private Condition sIsPositive /* associated with s > 0 */

        public method P()
        {
                if s = 0 then wait sIsPositive assert s > 0 s := s - 1
        }

        public method V()
        {
                s := s + 1 assert s > 0
                signal sIsPositive
        }
}
```

A thread that tries to decrement must wait until the integer is positive. We use a condition variable sIsPositive with an associated assertion of *PsIsPositive* = ($s > 0$)

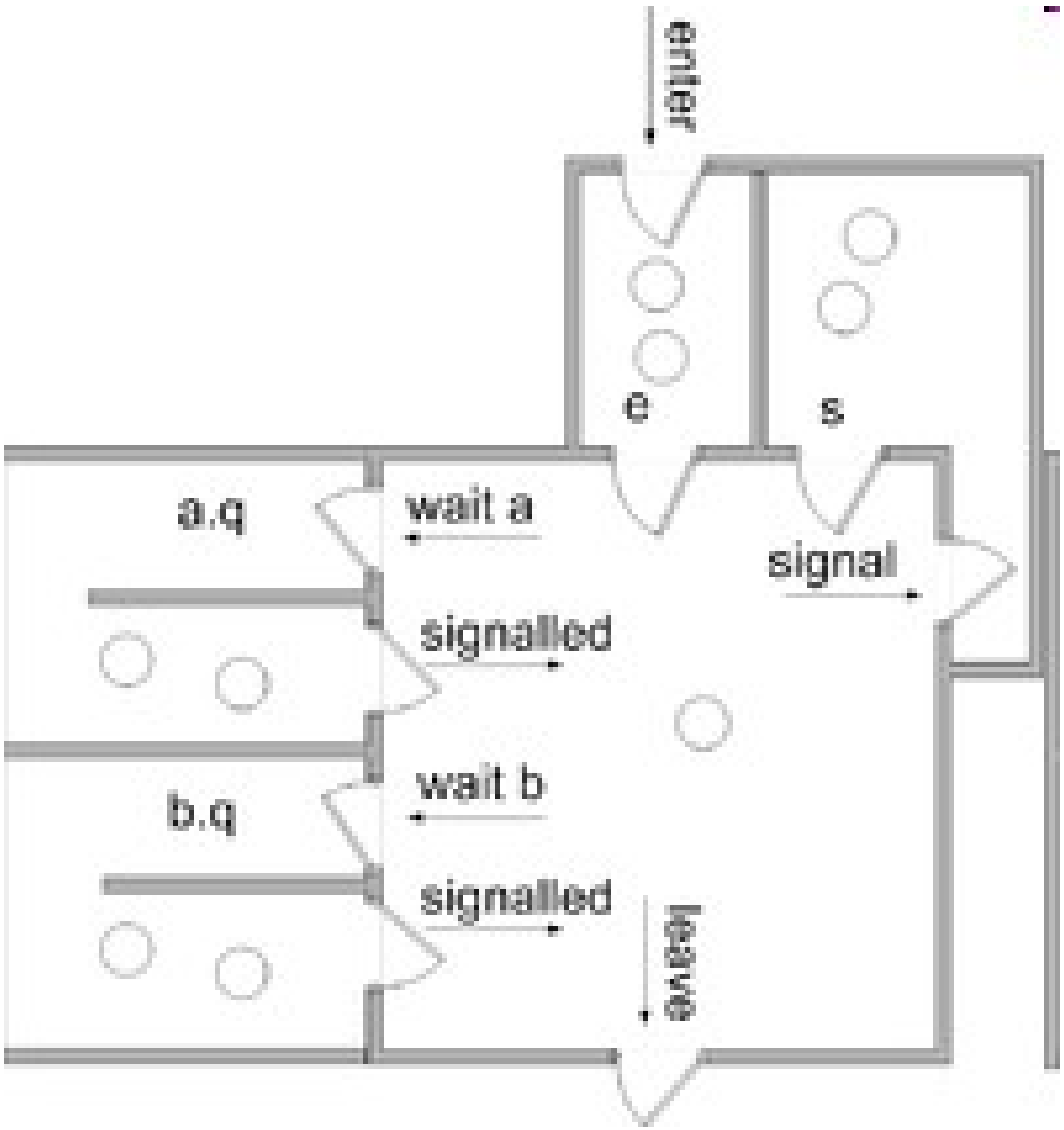
When a signal happens on a condition that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor:

the thread that signals and
any one of the threads that is waiting.

In order that at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

Blocking condition variables give priority to a signaled thread.

Nonblocking condition variables give priority to the signaling thread.


enter
e
s
a.q
wait a
signal
signalled
wait b
b.q
signalled
leave

enter the monitor:
  enter the method
  if the monitor is locked
    add this thread to e block this thread
  else
    lock the monitor

leave the monitor:
  schedule
  return from the method

wait *c* :
  add this thread to *c*.q
  schedule
  block this thread

schedule :
  if there is a thread on s
    select and remove one thread from s and restart it
    (this thread will occupy the monitor next)
  else if there is a thread on e
    select and remove one thread from e and restart it
    (this thread will occupy the monitor next)
  else
    unlock the monitor
    (the monitor will become unoccupied)

signal *c* :
  if there is a thread waiting on *c*.q
    select and remove one such thread t from *c*.q
    (t is called "the signaled thread")
    add this thread to s restart t
    (so t will occupy the monitor next)
    block this thread

With a blocking condition variable, the signaling thread must wait outside the monitor (at least) until the signaled thread relinquishes occupancy of the monitor by either returning or by again waiting on a condition.

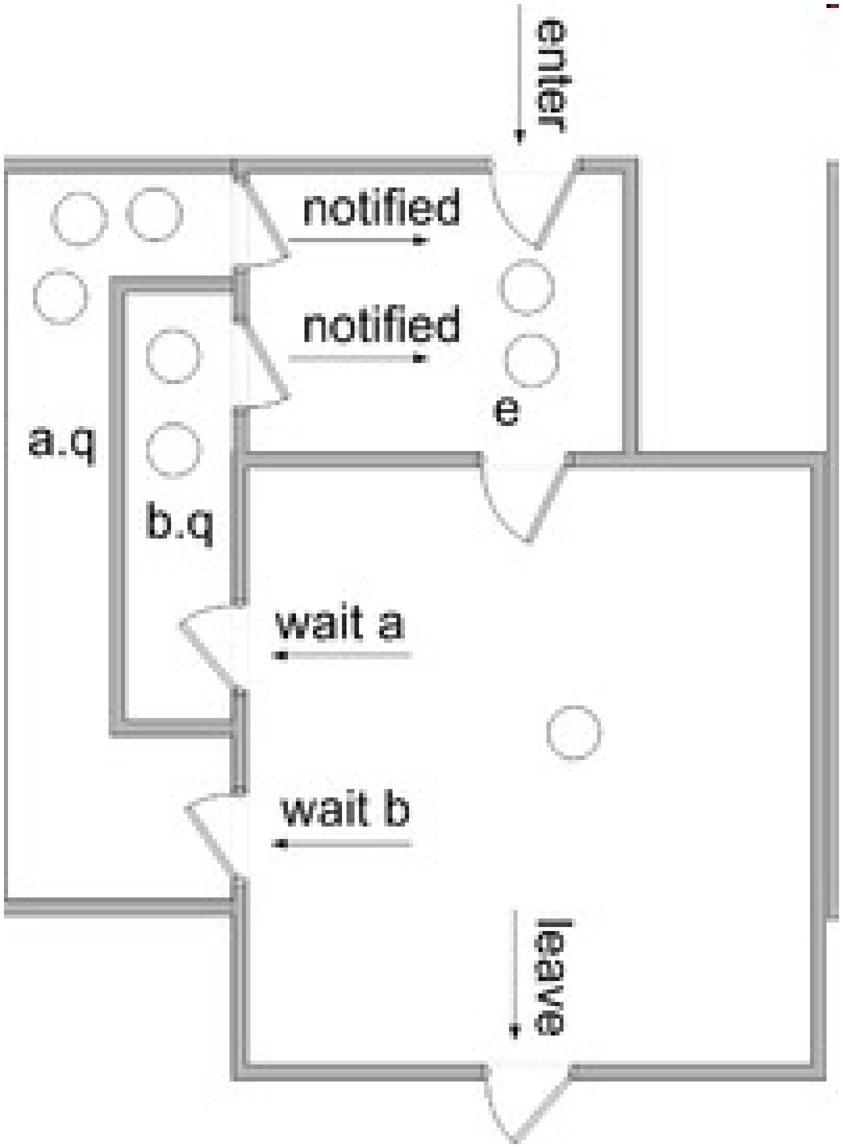
We assume there are two queues of threads associated with each monitor object

e is the entrance queue

s is a queue of threads that have signaled.

In addition we assume that for each condition *c*, there is a queue

c.q, which is a queue for threads waiting on condition *c*



With *nonblocking condition variables,*
signaling does not cause the signaling thread to lose occupancy of the monitor. Instead the signaled threads are moved to the e queue. Signaling thread continues.
There is no need for the s queue.

enter the monitor:

enter the method
if the monitor is locked
    add this thread to e
    block this thread
else
    lock the monitor

leave the monitor:

schedule
return from the method

wait *c* :

add this thread to *c*.q
schedule
block this thread

notify *c* :

if there is a thread waiting on *c*.q
select and remove one thread t from *c*.q
(t is called "the notified thread")
move t to e

notify all *c* :

move all threads waiting on *c*.q to e

schedule :

if there is a thread on e
    select and remove one thread from e and restart it
else
    unlock the monitor

Monitors

A monitor is an object that contains both
the data and
the procedures
needed to perform allocation of a particular of a serially reusable shared resources.

A thread calls a monitor entry routine to access a resource.

Only one thread at a time is allowed to enter the monitor.

Other threads are made to wait at the monitor boundary.

Data inside a monitor may be
i) global to all routines inside the monitor, or
ii) local to a specific routine

Monitors

Data inside a monitor may be

i) global to all routines inside the monitor, or

ii) local to a specific routine

Monitor data is accessible only within the monitor.

Threads outside the monitor cannot access the monitor data.

This is a kind of information hiding.

Monitors

A thread –

a) calls a monitor entry routine

b) if no other threads inside the monitor

c) the thread acquires a lock on the monitor and enters it

d) else

e) the thread is made to wait until the lock is released by the other thread

Monitors

Finally, a thread having the lock of the monitor calls the monitor entry routine to release the resource.

Monitor entry routine calls signal to allow one of the waiting threads to enter the monitor and acquire the resource.

If there is no waiting thread then signal has no effect,
monitor recaptures the resource.

Monitors : Condition variables

Condition variables:

A thread inside a monitor uses a condition variable to wait on a condition outside the monitor.

A monitor associates a separate condition variable with each distinct situation that might cause a thread to have to wait.

Operations:

wait(condition variable)

signal(condition variable)

Monitors : Condition variables

Condition variables … :

Every condition variable has an associated Queue.

A thread calls wait on a c.v. is placed in the queue of the c.v.

→While in queue the thread is waiting outside the monitor.
(so that another thread may enter the monitor to signal.)

A thread calls signal on a c.v. causes a thread waiting in the queue to be removed and reenter the monitor.

→FIFO queue is maintained most often.

Monitors : Condition variables

Condition variables … :

Signal-and-exit monitor:
- a thread immediately exits the monitor upon signaling.

Signal-and-continue monitor:
- signals that the monitor will soon be available
- still keeps the lock until the thread exits
  - a) calling a wait on another c.v.
  - b) after executing some code in the monitor

Monitors : Resource Allocation

Resource Allocation:

```
// Fig. 6.1: Resource allocator monitor

// monitor initialization (performed only once)
boolean inUse = false; // simple state variable
Condition available; // condition variable

// request resource
monitorEntry void getResource()
{
    if ( inUse ) // is resource in use?
    {
        wait( available ); // wait until available is signaled
    } // end if

    inUse = true; // indicate resource is now in use

} // end getResource

// return resource
monitorEntry void returnResource()
{
    inUse = false; // indicate resource is not in use
    signal( available ); // signal a waiting thread to proceed

} // end returnResource
```

Monitors : Circular buffer

```
// Fig. 6.2: Circular buffer monitor

char circularBuffer[] = new char[ BUFFER_SIZE ]; // buffer
int writerPosition = 0; // next slot to write to
int readerPosition = 0; // next slot to read from
int occupiedSlots = 0;  // number of slots with data
Condition hasData;      // condition variable
Condition hasSpace;     // condition variable

// monitor entry called by producer to write data
monitorEntry void putChar( char slotData )
{
   // wait on condition variable hasSpace if buffer is full
   if ( occupiedSlots == BUFFER_SIZE )
   {
      wait( hasSpace ); // wait until hasSpace is signaled
   } // end if

   // write character to buffer
   circularBuffer[ writerPosition ] = slotData;
   ++occupiedSlots; // one more slot has data
   writerPosition = (writerPosition + 1) % BUFFER_SIZE;
   signal( hasData ); // signal that data is available
} // end putChar

```

Monitors : Circular buffer

```
// monitor entry called by consumer to read data
monitorEntry void getChar( outputParameter slotData )
{
   // wait on condition variable hasData if the buffer is empty
   if ( occupiedSlots == 0 )
   {
      wait( hasData ); // wait until hasData is signaled
   } // end if

   // read character from buffer into output parameter slotData
   slotData = circularBuffer[ readPosition ];
   occupiedSlots--; // one fewer slots has data
   readerPosition = (readerPosition + 1) % BUFFER_SIZE;
   signal( hasSpace ); // signal that character has been read
} // end getChar
```

Monitors : Readers Writers Problem

```
// Fig. 6.3: Readers/writers problem

int readers = 0; // number of readers
boolean writeLock = false; // true if a writer is writing
Condition canWrite; // condition variable
Condition canRead; // condition variable

// monitor entry called before performing read
monitorEntry void beginRead()
{
   // wait outside monitor if writer is currently writing or if
   // writers are currently waiting to write
   if ( writeLock || queue( canWrite ) )
   {
      wait( canRead ); // wait until reading is allowed
   } // end if

   ++readers; // there is another reader

   signal( canRead ); // allow waiting readers to proceed
} // end beginRead

```

Monitors : Readers Writers Problem

```
// monitor entry called after reading
monitorEntry void endRead()
{
    --readers; // there are one fewer readers

    // if no more readers are reading, allow a writer to write
    if ( readers == 0 )
    {
        signal ( canWrite ); // allow a writer to proceed
    } // end if

} // end endRead

// monitor entry called before performing write
monitorEntry void beginWrite()
{
    // wait if readers are reading or if a writer is writing
    if ( readers > 0 || writeLock )
    {
        wait( canWrite ); // wait until writing is allowed
    } // end if

```

Monitors : Readers Writers Problem

```
    writeLock = true; // lock out all readers and writers
} // end beginWrite

// monitor entry called after performing write
monitorEntry void endWrite()
{
    writeLock = false; // release lock

    // if a reader is waiting to enter, signal a reader
    if ( queue( canRead ) )
    {
        signal( canRead ); // cascade in waiting readers
    } // end if
    else // signal a writer if no readers are waiting
    {
        signal( canWrite ); // one waiting writer can proceed
    } // end else

} // end endWrite
```

Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```java
// Fig. 6.4: SynchronizedBuffer.java
// SynchronizedBuffer synchronizes access to a shared integer.

public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer
   private int occupiedBuffers = 0; // counts occupied buffers

   // place value into buffer
   public synchronized void set( int value )
   {
      // for display, get name of thread that called this method
      String name = Thread.currentThread().getName();

      // while no empty buffers, place thread in waiting state
      while ( occupiedBuffers == 1 )
      {
         // output thread and buffer information, then wait
         try
         {
            System.err.println( name + " tries to write." );
            displayState( "Buffer full. " + name + " waits." );
            wait(); // wait until buffer is empty
         } // end try

```

Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```
            // if waiting thread interrupted, print stack trace
            catch ( InterruptedException exception )
            {
               exception.printStackTrace();
            } // end catch

         } // end while

         buffer = value; // set new buffer value

         // indicate producer cannot store another value
         // until consumer retrieves current buffer value
         ++occupiedBuffers;

         displayState( name + " writes " + buffer );

         notify(); // tell waiting thread to enter ready state
      } // end method set; releases lock on SynchronizedBuffer

```

Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```java

    // return value from buffer
    public synchronized int get()
    {
       // for display, get name of thread that called this method
       String name = Thread.currentThread().getName();

       // while no data to read, place thread in waiting state
       while ( occupiedBuffers == 0 )
       {
          // output thread and buffer information, then wait
          try
          {
             System.err.println( name + " tries to read." );
            displayState( "Buffer empty. " + name + " waits." );
```

```java
             wait();// wait until buffer contains new values
          } // end try

          // if waiting thread interrupted, print stack trace
          catch ( InterruptedException exception )
          {
             exception.printStackTrace();
          } // end catch

       } // end while

```

Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```java
         // indicate that producer can store another value
         // because consumer just retrieved buffer value
         --occupiedBuffers;

         displayState( name + " reads " + buffer );

         notify(); // tell waiting thread to become ready

         return buffer;
      } // end method get; releases lock on SynchronizedBuffer

      // display current operation and buffer state
      public void displayState( String operation )
      {
         StringBuffer outputLine = new StringBuffer( operation );
         outputLine.setLength( 40 );
         outputLine.append( buffer + "\t\t" + occupiedBuffers );
         System.err.println( outputLine );
         System.err.println();
      } // end method displayState

   } // end class SynchronizedBuffer
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```java
// Fig. 6.5: SharedBufferTest2.java
// SharedBufferTest2creates producer and consumer threads.

public class SharedBufferTest2
{
   public static void main( String [] args )
   {
      // create shared object used by threads
      SynchronizedBuffer sharedLocation = new SynchronizedBuffer();

      // Display column heads for output
      StringBuffer columnHeads =
         new StringBuffer( "Operation" );
      columnHeads.setLength( 40 );
      columnHeads.append( "Buffer\t\tOccupied Count" );
      System.err.println( columnHeads );
      System.err.println();
      sharedLocation.displayState( "Initial State" );

      // create producer and consumer objects
      Producer producer = new Producer( sharedLocation );
      Consumer consumer = new Consumer( sharedLocation );

      producer.start(); // start producer thread
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```
         consumer.start(); // start consumer thread

      } // end main

   } // end class SharedBufferTest2
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Sample Output 1:

Operation	Buffer	Occupied Count
Initial State	-1	0
Consumer tries to read. Buffer empty. Consumer waits.	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Consumer tries to read. Buffer empty. Consumer waits.	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```
Consumer reads 3                          3                0

Consumer tries to read.
Buffer empty. Consumer waits.             3                0

Producer writes 4                         4                1

Consumer reads 4                          4                0
Producer done producing.
Terminating Producer.

Consumer read values totaling: 10.
Terminating Consumer.
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Sample Output 2:

Operation	Buffer	Occupied Count
Initial State	-1	0
Consumer tries to read. Buffer empty. Consumer waits.	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Producer tries to write. Buffer full. Producer waits.	2	1
Consumer reads 2	2	0
Producer writes 3	3	1

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```
Consumer reads 3                         3              0

Producer writes 4                        4              1

Producer done producing.
Terminating Producer.
Consumer reads 4                         4              0

Consumer read values totaling: 10.
Terminating Consumer.
```

Sample Output 3:

```
Operation                               Buffer         Occupied Count

Initial State                            -1             0

Producer writes 1                        1              1
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

Sample Output 3 (Cont.):

```
Operation                               Buffer          Occupied Count

Initial State                            -1               0

Producer writes 1                        1                1

Consumer reads 1                         1                0

Producer writes 2                        2                1

Consumer reads 2                         2                0

Producer writes 3                        3                1

Consumer reads 3                         3                0

Producer writes 4                        4                1

Producer done producing.
Terminating Producer.
```

6.4 Java Multithreading Case Study, Part III: Producer/Consumer Relationship in Java

```
Consumer reads 4                                    4                0


Consumer read values totaling: 10.
Terminating Consumer.
```