Operating Systems: Internals and Design Principles

Chapter 5 Concurrency: Mutual Exclusion and Synchronization

Seventh Edition

By William Stallings

Operating Systems:
Internals and Design Principles

" Designing correct routines for controlling concurrent activities proved to be one of the most difficult aspects of systems programming. The ad hoc techniques used by programmers of early multiprogramming and real-time systems were always vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred. The errors are particularly difficult to locate, since the precise conditions under which they appear are very hard to reproduce."

—THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY,

MIT Press, 1980

Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing

Concurrency

- **Concurrency refers to any form of interaction among processes or threads**
  - concurrency is a fundamental part of O/S design
  - concurrency includes
    - communication among processes/threads
    - sharing of, and competition for system resources
    - cooperative processing of shared data
    - synchronization of process/thread activities
    - organized CPU scheduling
    - solving deadlock and starvation problems

Concurrency

- **Concurrency arises in the same way at different levels of execution streams**
  - **multiprogramming**—interaction between multiple processes running on one CPU (pseudoparallelism)
  - **multithreading**—interaction between multiple threads running in one process
  - **multiprocessors**—interaction between multiple CPUs running multiple processes/threads (real parallelism)
  - **multicomputers**—interaction between multiple computers running distributed processes/threads

  →*the principles of concurrency are basically the same in all of these categories (possible differences will be pointed out)*

Concurrency Arises in Three Different Contexts:

Multiple Applications

- invented to allow processing time to be shared among active applications

Structured Applications

- extension of modular design and structured programming

Operating System Structure

- OS themselves implemented as a set of processes or threads

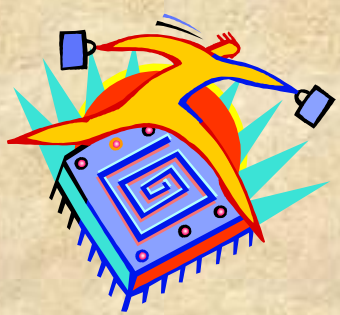
Concurrency

Key Terms

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Principles of Concurrency

- Interleaving and overlapping
  - can be viewed as examples of concurrent processing
  - both present the same problems
- Uniprocessor – the relative speed of execution of processes cannot be predicted
  - depends on activities of other processes
  - the way the OS handles interrupts
  - scheduling policies of the OS



Difficulties of Concurrency

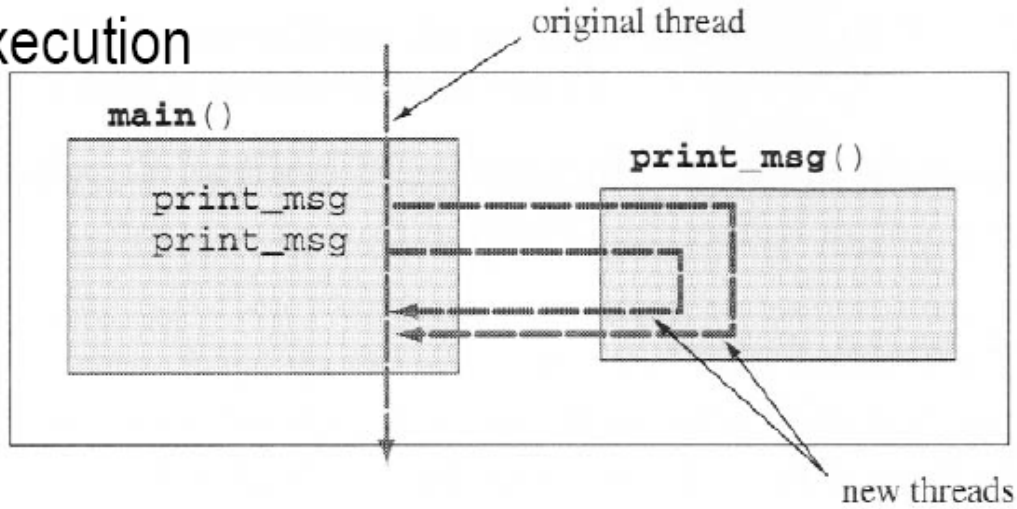
- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - the "loser" of the race is the process that updates last and will determine the final value of the variable

Race Conditions and Critical Regions

- **Inconsequential race condition in the shopping scenario**
  - ✓ there is a "race condition" if the outcome depends on the order of the execution



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

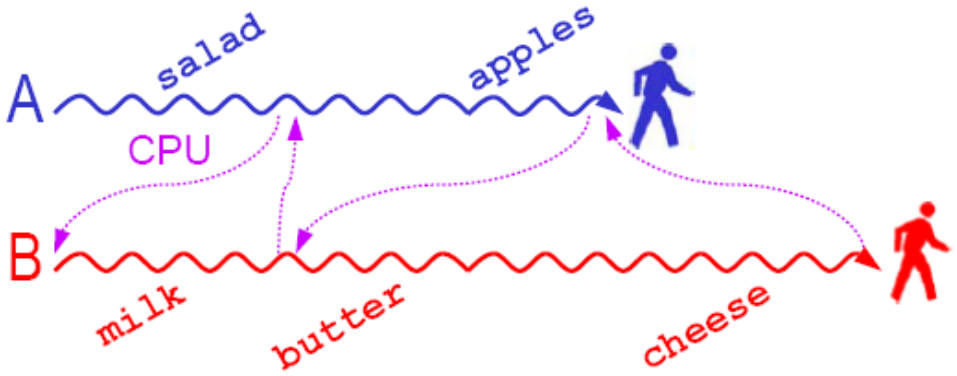
```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

Multithreaded shopping diagram and possible outputs

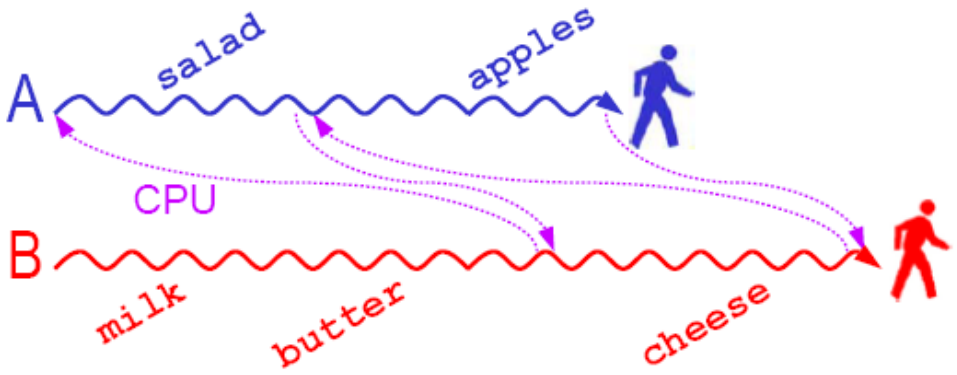
Race Conditions and Critical Regions

- **Inconsequential race condition in the shopping scenario**
  - ✓ the outcome depends on the CPU scheduling or "interleaving" of the threads (separately, each thread always does the same thing)

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```




```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

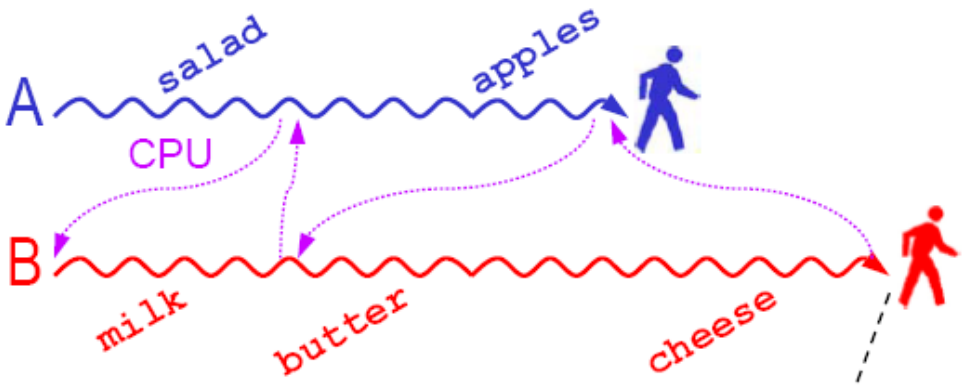


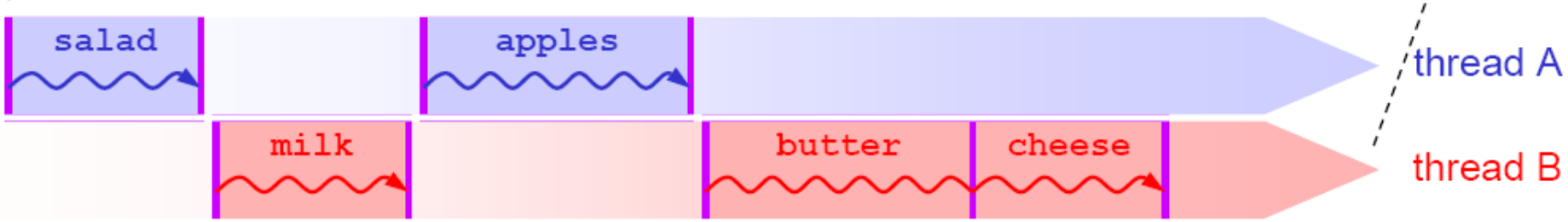
Race Conditions and Critical Regions

➢ Inconsequential race condition in the shopping scenario

- ✓ the CPU switches from one process/thread to another, possibly on the basis of a preemptive clock mechanism

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

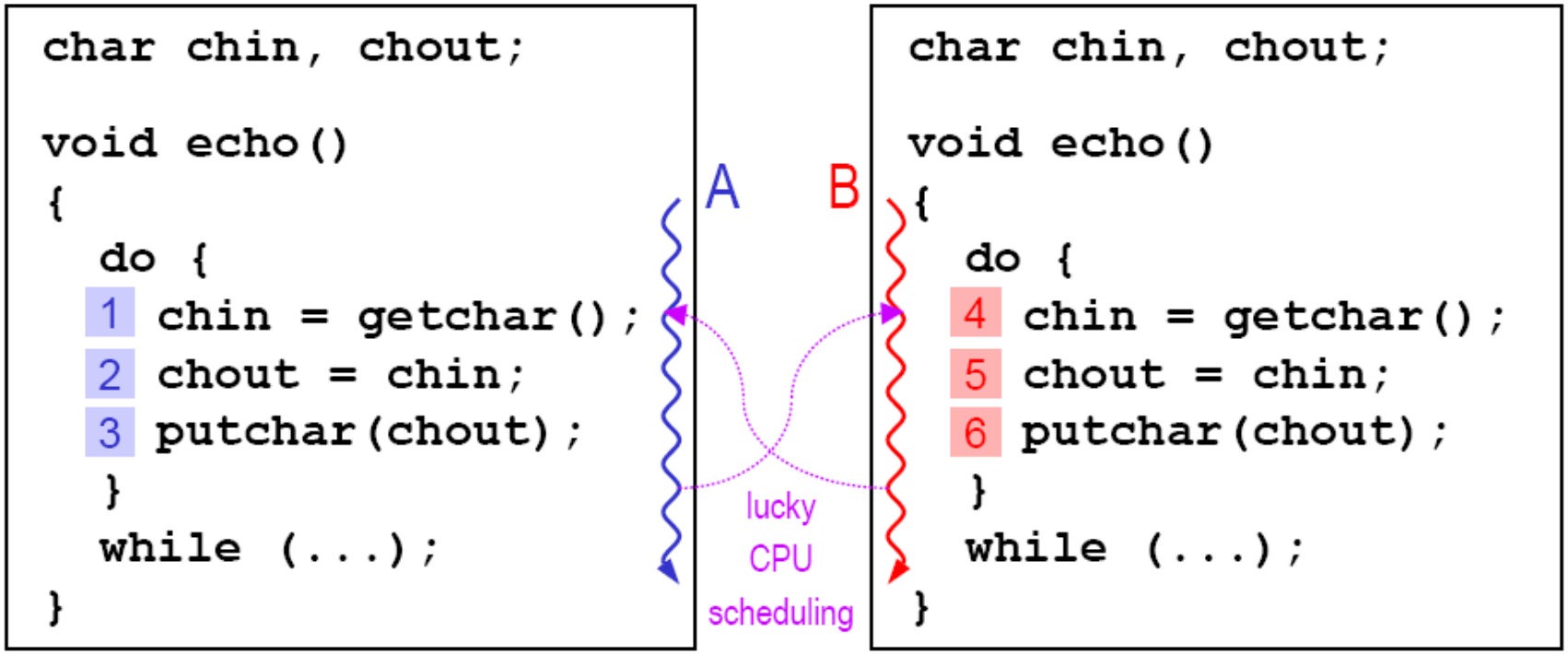




Thread view expanded in real execution time

Race Conditions and Critical Regions

- **Consequential race conditions in I/O & variable sharing**




```
> ./echo
Hello world!
Hello world!
```

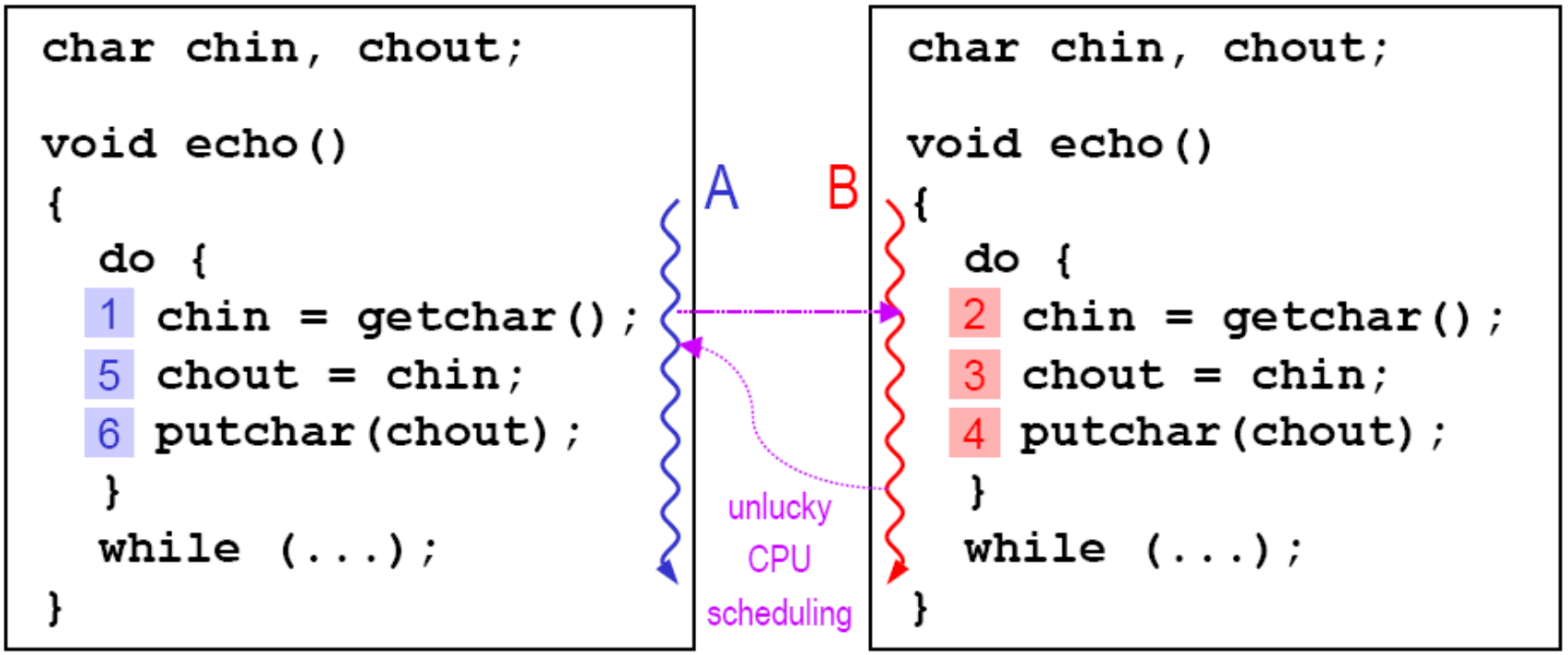
Single-threaded echo

```
> ./echo
Hello world!
Hello world!
```

Multithreaded echo (lucky)

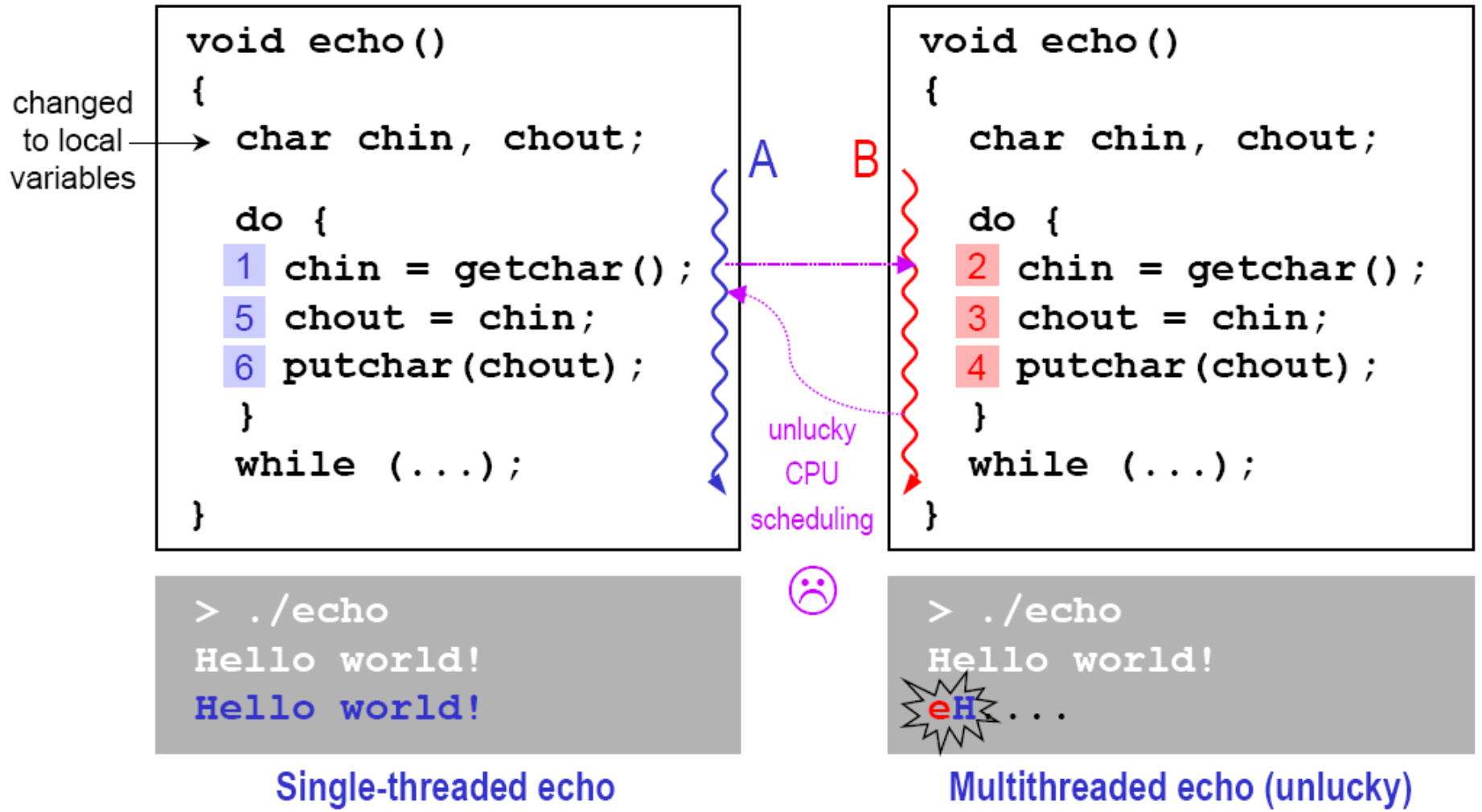
Race Conditions and Critical Regions

- **Consequential race conditions in I/O & variable sharing**



Race Conditions and Critical Regions

➢ Consequential race conditions in I/O & variable sharing



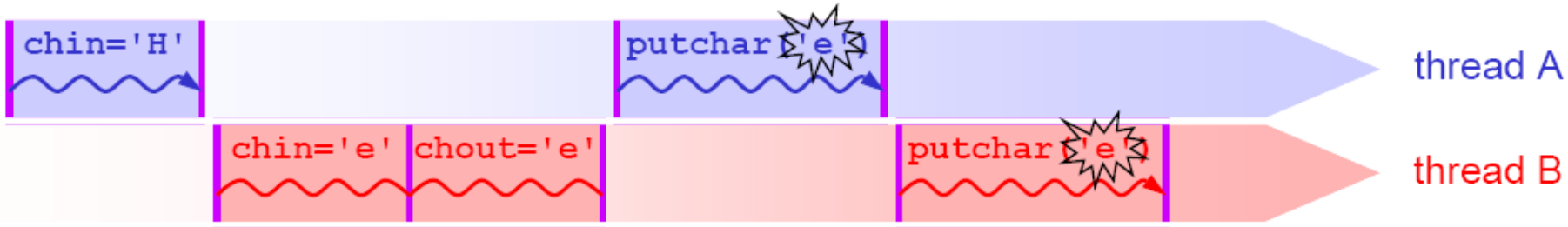
Race Conditions and Critical Regions

- **Consequential race conditions in I/O & variable sharing**
  - ✓ note that, in this case, replacing the global variables with local variables did not solve the problem
  - ✓ we actually had two race conditions here:
    - one race condition in the shared variables and the order of value assignment
    - another race condition in the shared output stream: which thread is going to write to output first (this race persisted even after making the variables local to each thread)
  - → *generally, problematic race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)*

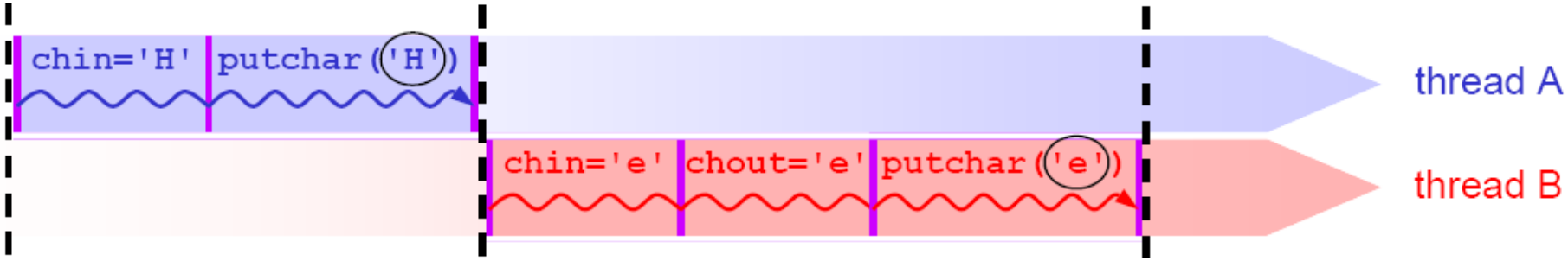
Race Conditions and Critical Regions

How to avoid race conditions?

- ✓ find a way to keep the instructions together
- ✓ this means actually. . . reverting from too much interleaving and going back to "indivisible" blocks of execution!!



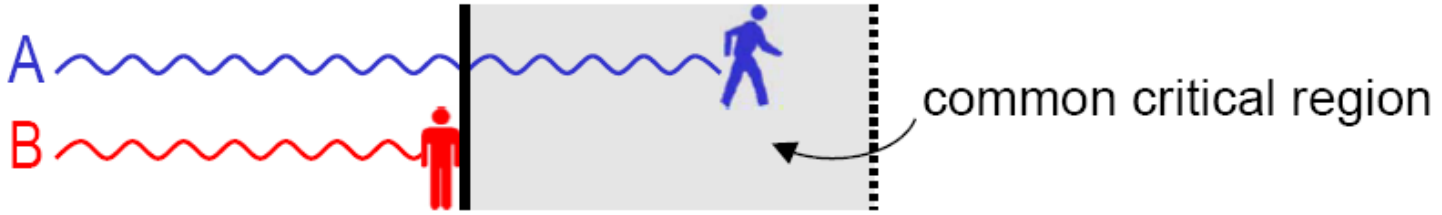
(a) too much interleaving may create race conditions



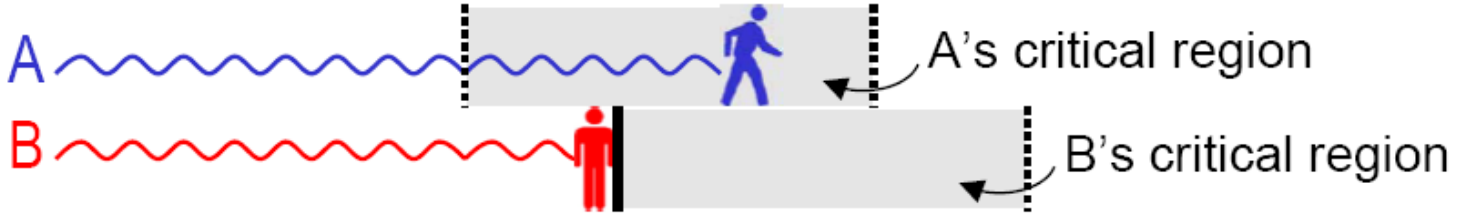
(b) keeping "indivisible" blocks of execution avoids race conditions

Race Conditions and Critical Regions

- **The "indivisible" execution blocks are critical regions**
  - ✓ a critical region is a section of code that may be executed by only one process or thread at a time



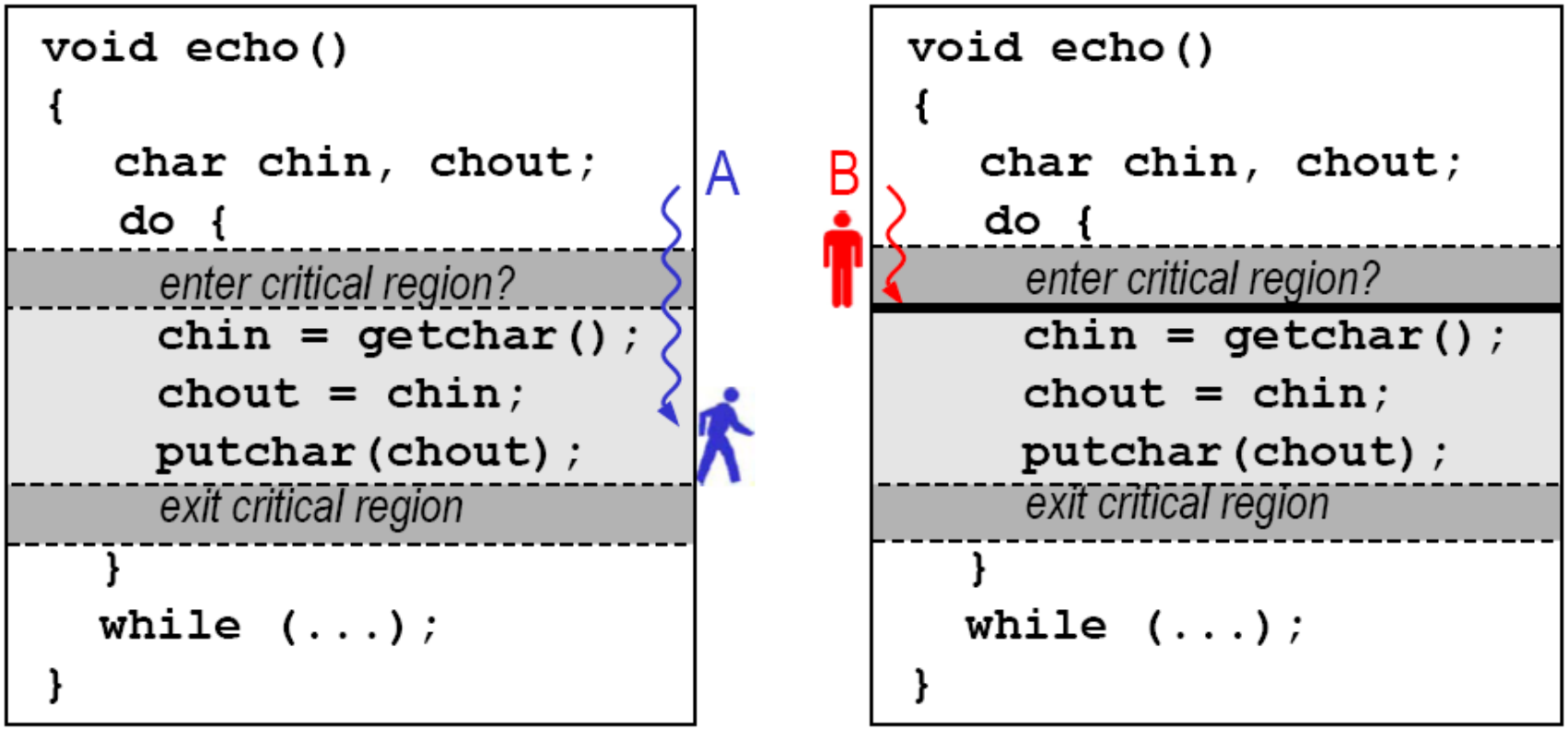
  - ✓ although it is not necessarily the same region of memory or section of program in both processes



  - → *but physically different or not, what matters is that these regions cannot be interleaved or executed in parallel (pseudo or real)*

Race Conditions and Critical Regions

- **We need mutual exclusion from critical regions**
  - ✓ critical regions can be protected from concurrent access by padding them with entrance and exit gates (we'll see how later): a thread must try to check in, then it must check out

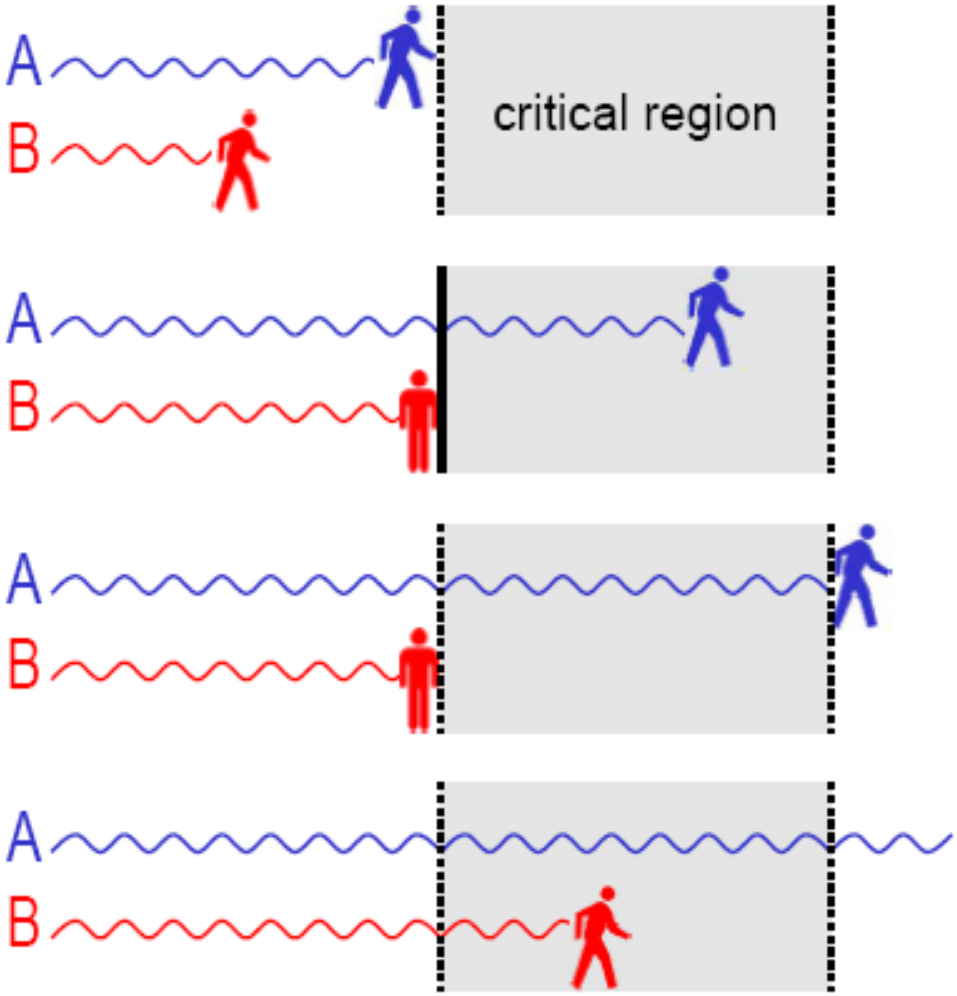


Mutual Exclusion by Busy Waiting

- **Desired effect: mutual exclusion from the critical region**

1. thread A reaches the gate to the critical region (CR) before B
2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)
3. thread A exits CR; thread B can now enter
4. thread B enters CR

HOW is this achieved??



Mutual Exclusion

```
        PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
        /* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

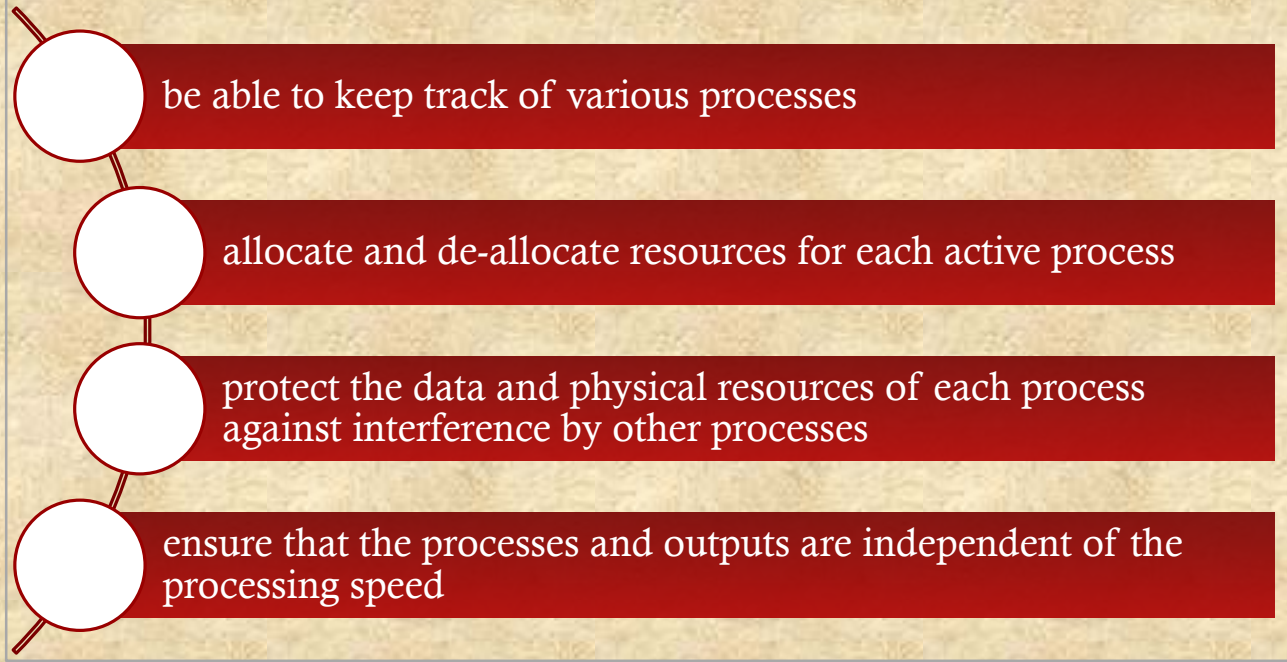
. . .

```
        /* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

Figure 5.1 Illustration of Mutual Exclusion

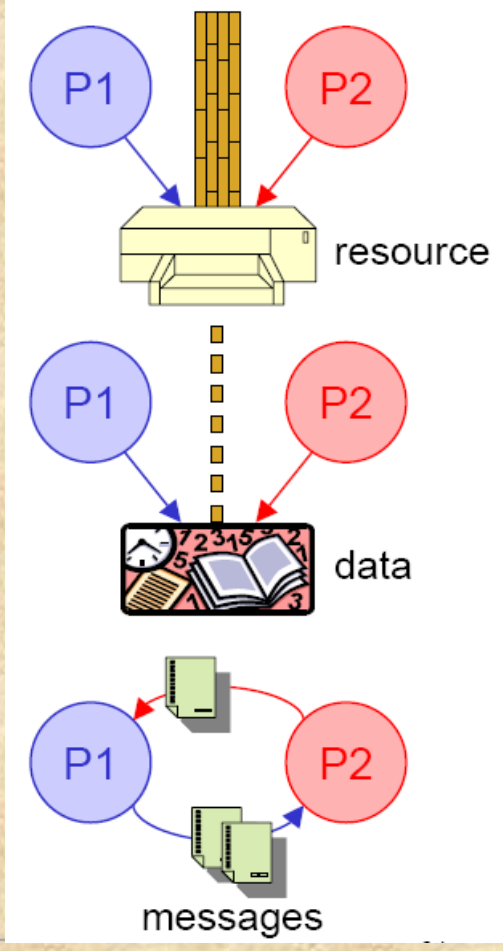
Operating System Concerns

- Design and management issues raised by the existence of concurrency:
  - The OS must:



Concurrency

- **Whether processes or threads: three basic interactions**
  - **processes unaware of each other**—they must use shared resources independently, without interfering, and leave them intact for the others
  - **processes indirectly aware of each other**—they work on common data and build some result together via the data ("stigmergy" in biology)
  - **processes directly aware of each other**—they cooperate by communicating, e.g., exchanging messages



Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- **the need for mutual exclusion**
- **deadlock**
- **starvation**

Cooperation Among Processes by Sharing

- Writing must be mutually exclusive
- Critical sections are used to provide data integrity

Cooperation Among Processes by Communication

- Messages are passed
  - Mutual exclusion is not a control requirement
- Possible to have deadlock
  - Each process waiting for a message from the other process
- Possible to have starvation
  - Two processes sending message to each other while another process waits for a message

Race Conditions and Critical Regions

Chart of mutual exclusion

1. **mutual exclusion inside** — only one process at a time may be allowed in a critical region

2. **no exclusion outside** — a process stalled in a *non*critical region may not exclude other processes from their critical regions

3. **no indefinite occupation** — a critical region may be only occupied for a finite amount of time

Race Conditions and Critical Regions

Chart of mutual exclusion (cont'd)

4. **no indefinite delay when barred** — a process may be only excluded for a finite amount of time (no deadlock or starvation)

5. **no delay when about to enter** — a critical region free of access may be entered immediately by a process

6. **nondeterministic scheduling** — no assumption should be made about the relative speeds of processes

Requirements for Mutual Exclusion

- Must be enforced
- A process that halts must do so without interfering with other processes
- No deadlock or starvation

- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

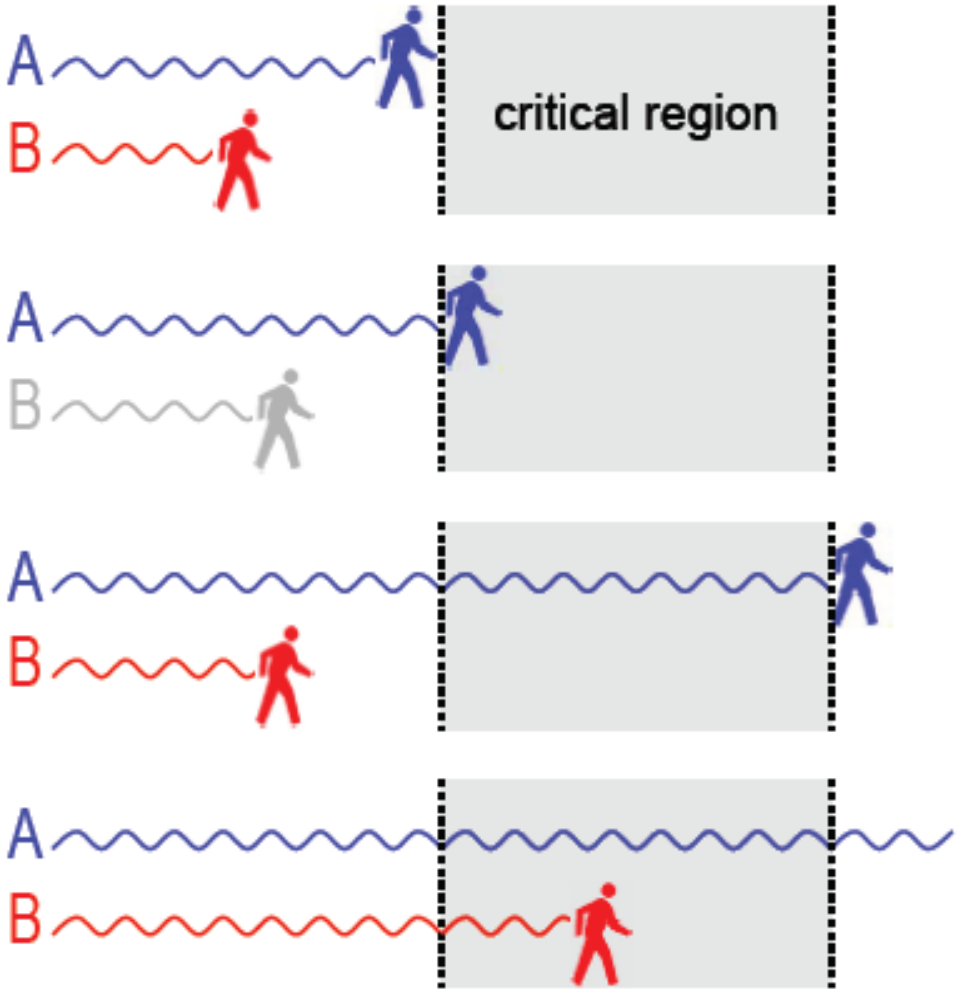
Disabling Interrupts

```
While (true) {
   /*  preamble – not in critical section   */
   /*  disable interrupts  */
   /*  critical section */
   /*  enable interrupts */
   /*  remainder – not in critical section  */
}
```

Mutual Exclusion by Busy Waiting

- **Implementation 0 — disabling hardware interrupts**

1. thread A reaches the gate to the critical region (CR) before B
2. as soon as A enters CR, it disables all interrupts, thus B cannot be scheduled
3. as soon as A exits CR, it reenables interrupts; B can be scheduled again
4. thread B enters CR



Mutual Exclusion by Busy Waiting

- **Implementation 0 — ~~disabling hardware interrupts~~** 👎
  - ✓ it works, but is foolish
  - ✓ what guarantees that the user process is going to ever exit the critical region?
  - ✓ meawhile, the CPU cannot interleave any other task, even unrelated to this race condition
  - ✓ the critical region becomes one *physically* indivisible block, not logically
  - ✓ also, this is not working in multi-processors

```
void echo()
{
   char chin, chout;
   do {
      disable hardware interrupts
      chin = getchar();
      chout = chin;
      putchar(chout);
      reenable hardware interrupts
   }
   while (...);
}
```

Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - – uniprocessor system
  - – disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - – the efficiency of execution could be noticeably degraded
  - – this approach will not work in a multiprocessor architecture

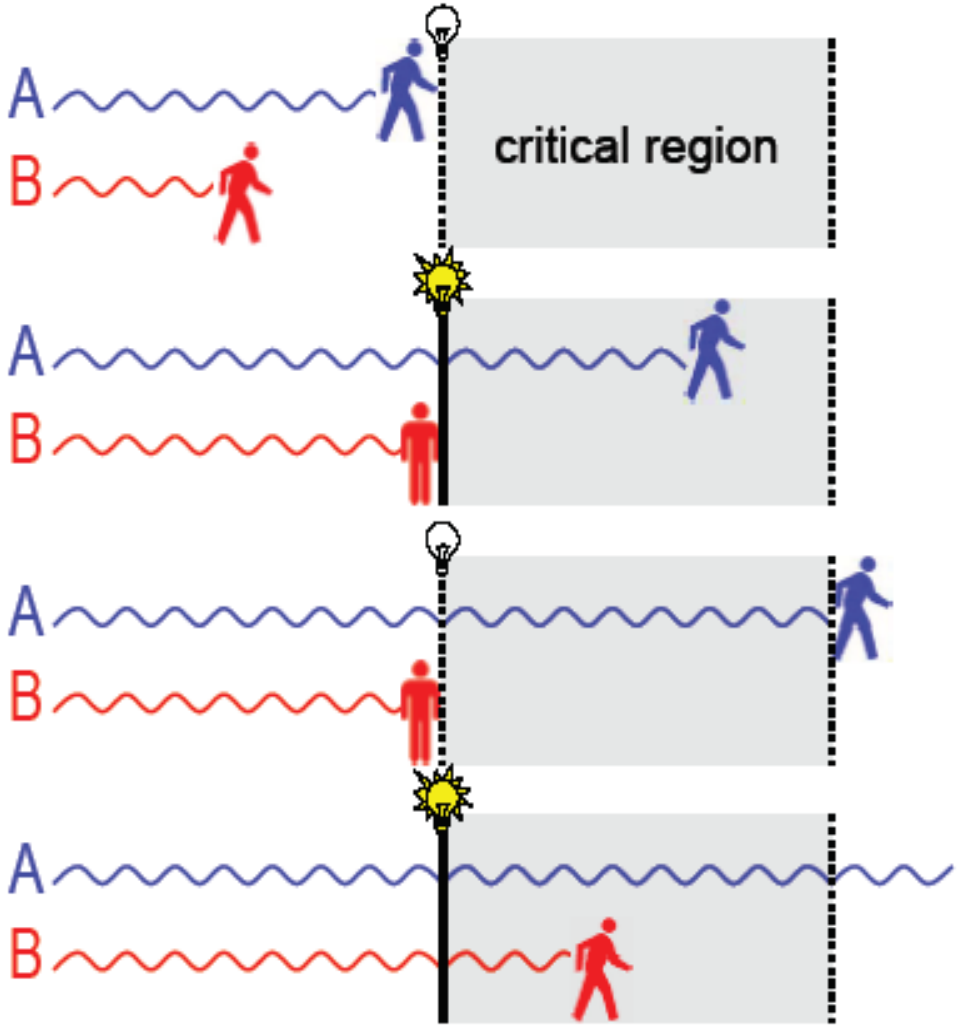
Mutual Exclusion: Using Simple Lock Variable

```
// before critical region

while (lock == TRUE);

lock = TRUE;

// inside the critical region

lock = FALSE;

// after critical region
```

Mutual Exclusion by Busy Waiting

- **Implementation 1 — simple lock variable**

1. thread A reaches CR and finds a lock at 0, which means that A can enter
2. thread A sets the lock to 1 and enters CR, which prevents B from entering
3. thread A exits CR and resets lock to 0; thread B can now enter
4. thread B sets the lock to 1 and enters CR



Mutual Exclusion by Busy Waiting

➢ Implementation 1 — simple lock variable

- ✓ the "lock" is a shared variable
- ✓ entering the critical region means testing and then setting the lock
- ✓ exiting means resetting the lock

```
while (lock);
   /* do nothing: loop */
lock = TRUE;
```

```
lock = FALSE;
```

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```

Mutual Exclusion by Busy Waiting

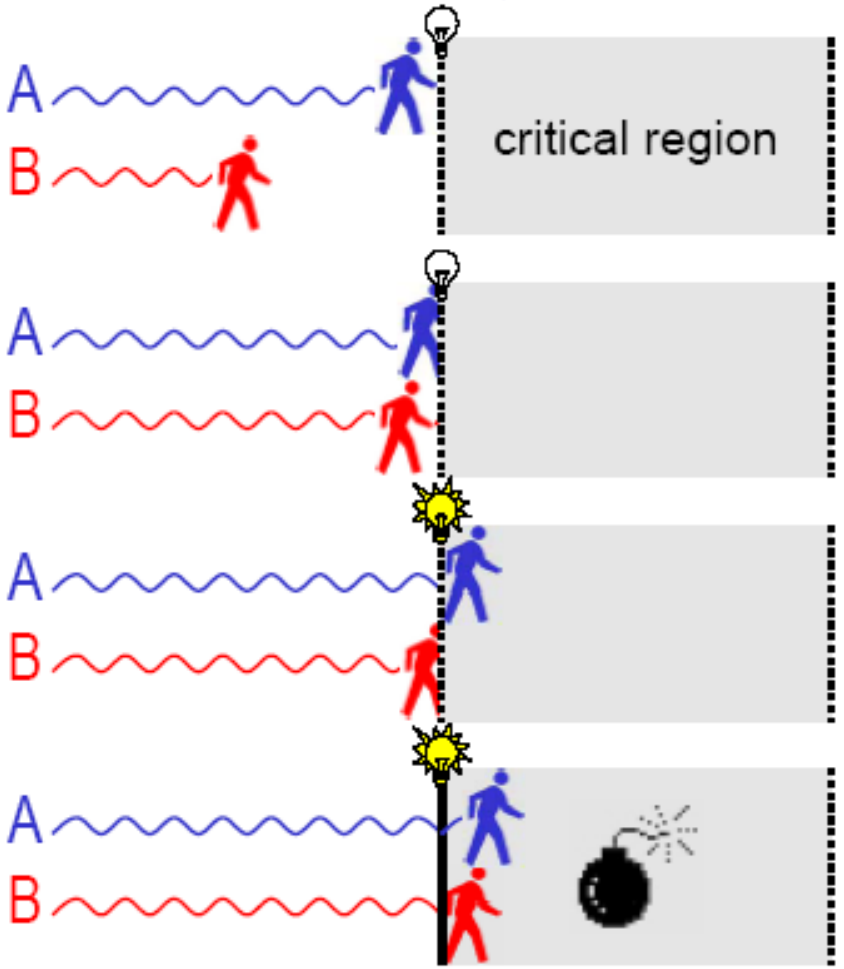
- **Implementation 1 — ~~simple lock variable~~** 👎

1. thread A reaches CR and finds a lock at 0, which means that A can enter

1.1 but before A can set the lock to 1, B reaches CR and finds the lock is 0, too

1.2 A sets the lock to 1 and enters CR but cannot prevent the fact that . . .

1.3 . . . B is going to set the lock to 1 and enter CR, too



Mutual Exclusion by Busy Waiting

- **Implementation 1 — ~~simple lock variable~~** 👎
  - ✓ suffers from the very flaw we want to avoid: a race condition
  - ✓ the problem comes from the small gap between testing that the lock is off and setting the lock

```
while (lock);   lock = TRUE;
```

  - ✓ it may happen that the other thread gets scheduled exactly inbetween these two actions (falls in the gap)
  - ✓ so they both find the lock off and then they both set it and enter

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Compares value of a variable to a constant and changes value in one single machine-level instruction

Mutual Exclusion by Busy Waiting

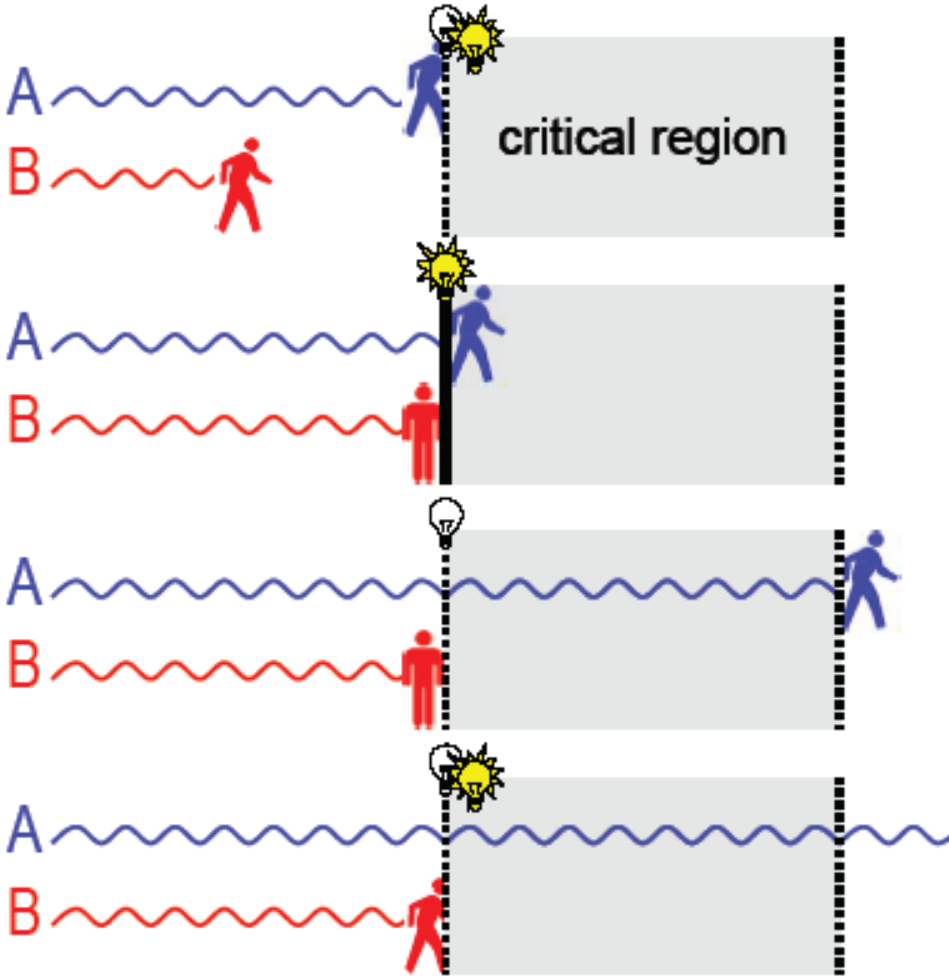
- **Implementation 2 — "indivisible" lock variable**

1. thread A reaches CR and finds the lock at 0 *and* sets it in one shot, then enters

1.1' even if B comes right behind A, it will find that the lock is already at 1

2. thread A exits CR, then resets lock to 0

3. thread B finds the lock at 0 and sets it to 1 in one shot, just before entering CR



Mutual Exclusion by Busy Waiting

- **Implementation 2 — "indivisible" lock variable**
  - ✓ the indivisibility of the "test-lock-and-set-lock" operation can be implemented with the hardware instruction **TSL**

```
enter_region:
    TSL REGISTER,LOCK | copy lock to register and set lock to 1
    CMP REGISTER,#0   | was lock zero?
    JNE enter_region  | if it was non zero, lock was set, so loop
    RET               | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0      | store a 0 in lock
    RET               | return to caller
```

```
void echo()
{
   char chin, chout;
   do {
      test-and-set-lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      set lock off
   }
   while (...);
}
```

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Mutual Exclusion by Busy Waiting

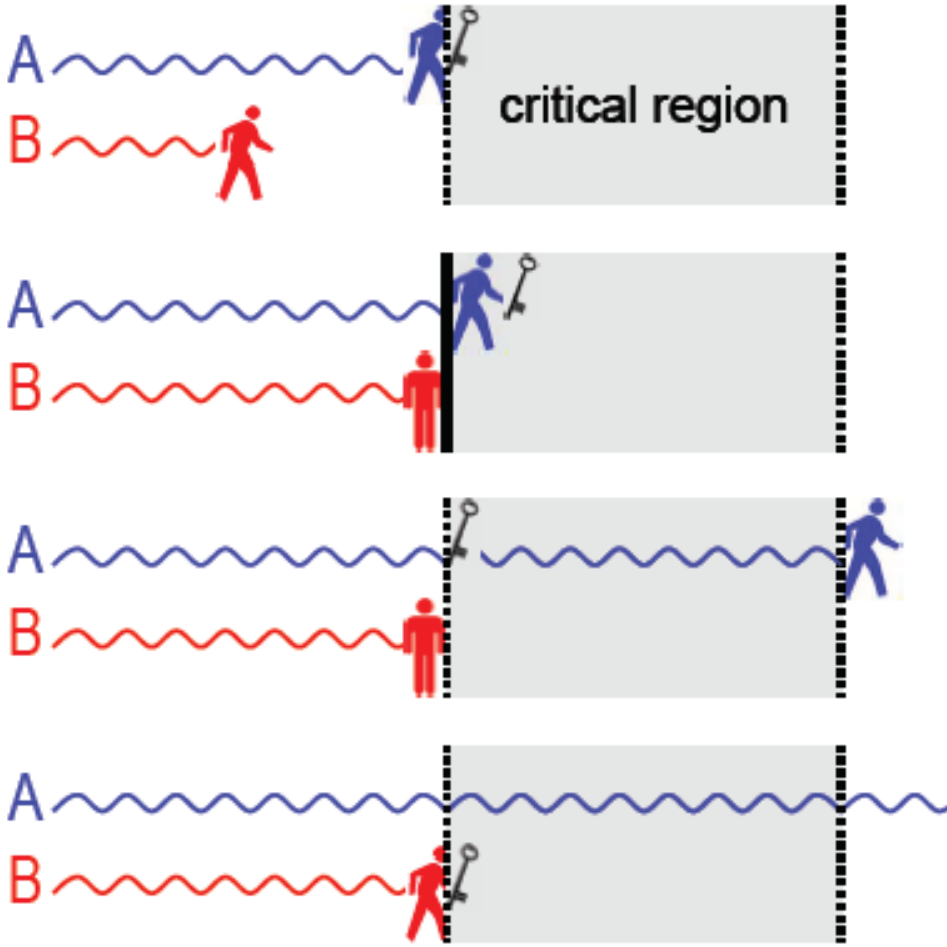
- **Implementation 2 — "indivisible" lock ⇔ one key**

1. thread A reaches CR and finds a key *and* takes it

1.1' even if B comes right behind A, it will not find a key

2. thread A exits CR and puts the key back in place

3. thread B finds the key and takes it, just before entering CR



Mutual Exclusion by Busy Waiting

➢ Implementation 2 — "indivisible" lock ⇔ one key

- ✓ "holding" a unique object, like a key, is an equivalent metaphor for "test-and-set"
- ✓ this is similar to the "speaker's baton" in some assemblies: only one person can hold it at a time
- ✓ holding is an indivisible action: you see it and grab it in one shot
- ✓ after you are done, you release the object, so another process can hold on to it

```
void echo()
{
   char chin, chout;
   do {
      take key and run
      chin = getchar();
      chout = chin;
      putchar(chout);
      return key
   }
   while (...);
}
```

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Test&Set Instruction
    - a **compare** is made between a memory value and a 0
    - if the values are the same, set the memory value to 1 and return TRUE
    - Otherwise, don't change value, return False
    - carried out atomically

Mutual Exclusion: Hardware Support

- Test and Set Instruction Logical Behavior

```
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

Remember: this entire function is performed in its entirety as a single operation

Test and Set Instruction

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
     while (true)
     {
          while (!testset (bolt))
             /* do nothing */;
          /* critical section */;
          bolt = 0;
          /* remainder */
     }
}
void main()
{
     bolt = 0;
     parbegin (P(1), P(2), . . . ,P(n));

}
```

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Compare&Swap Instruction
    - also called a "compare and exchange instruction"
    - a **compare** is made between a memory value and a test value
    - if the values are the same a **swap** occurs
    - carried out atomically

Mutual Exclusion: Hardware Support

- Compare&Swap Instruction

```
int compare_and_swap (int *word, int testval, int newval)
{
   int oldval;
   oldval = *word;
   if (oldval == testval) *word = newval;
   return oldval;
}
```

Remember: this entire function is performed in its entirety as a single operation

Compare and Swap Instruction

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

(a) Compare and swap instruction

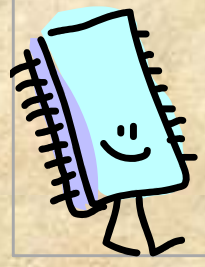


Figure 5.2 Hardware Support for Mutual Exclusion

Mutual Exclusion: Hardware Support

- Exchange instruction

```
void exchange (int register, int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Remember: this entire function is performed in its entirety as a single operation

Exchange Instruction

```
    /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

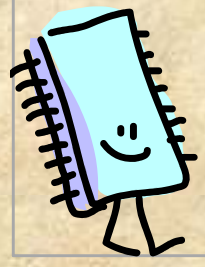


Figure 5.2 Hardware Support for Mutual Exclusion

Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Special Machine Instruction: Disadvantages

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock is possible
  - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

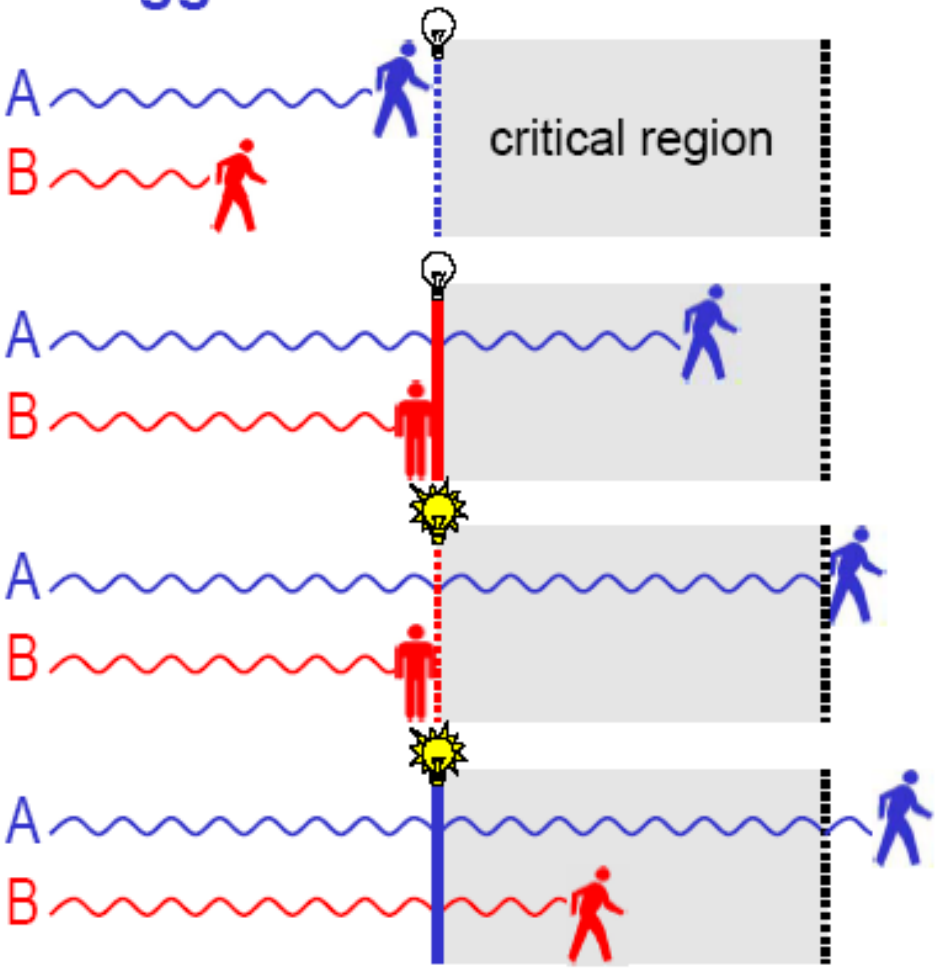
Mutual Exclusion: Without Hardware Support

- If there are no special machine instructions, then we must devise a solution that uses only variables

Mutual Exclusion by Busy Waiting

- **Implementation 3 — no-TSL toggle for two threads**

1. thread A reaches CR, finds a lock at 0, and enters without changing the lock
2. however, the lock has an opposite meaning for B: "off" means do not enter
3. only when A exits CR does it change the lock to 1; thread B can now enter
4. thread B enters CR: it will reset it to 0 for A after exiting



Mutual Exclusion by Busy Waiting

➢ Implementation 3 — no-TSL toggle for two threads

- ✓ the "toggle lock" is a shared variable used for strict alternation
- ✓ here, entering the critical region means only testing the toggle: it must be at 0 for A, and 1 for B
- ✓ exiting means switching the toggle: A sets it to 1, and B to 0

```
bool toggle = FALSE;

void echo()
{
    char chin, chout;
    do {
        test toggle
        chin = getchar();
        chout = chin;
        putchar(chout);
        switch toggle
    }
    while (...);
}
```

A's code	B's code
`while (toggle); /* loop */`	`while (!toggle); /* loop */`
`toggle = TRUE;`	`toggle = FALSE;`

Mutual Exclusion by Busy Waiting

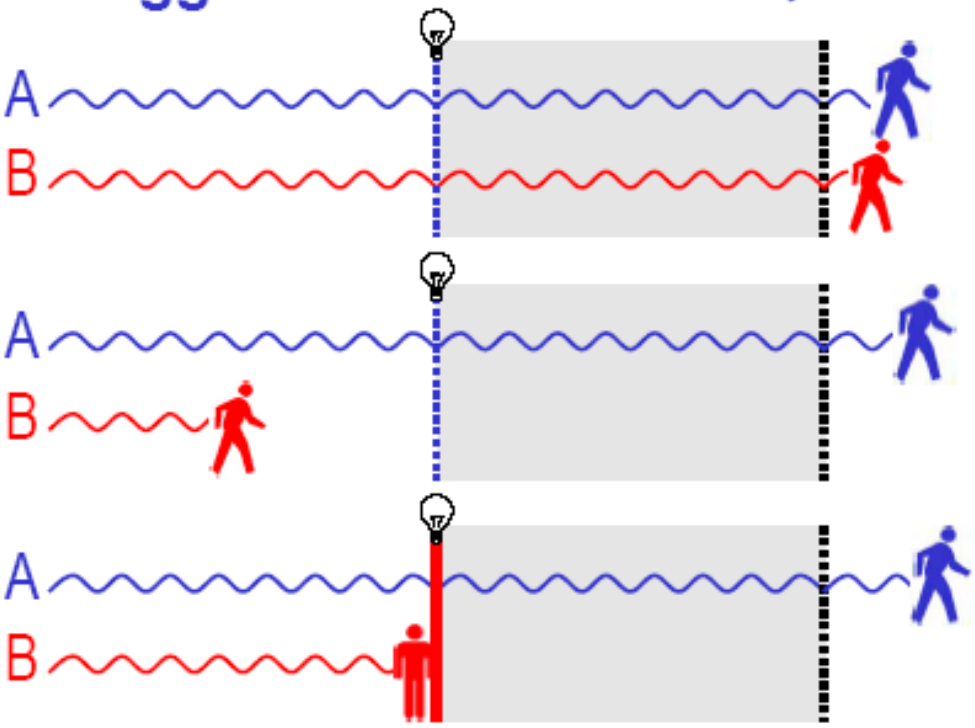
- **Implementation 3 — ~~no-TSL toggle for two threads~~** 👎

5. thread B exits CR and switches the lock back to 0 to allow A to enter next

5.1 but scheduling happens to make B faster than A and come back to the gate first

5.2 as long as A is still busy or interrupted in its noncritical region, B is barred access to its CR

→ *this violates item 2. of the chart of mutual exclusion*

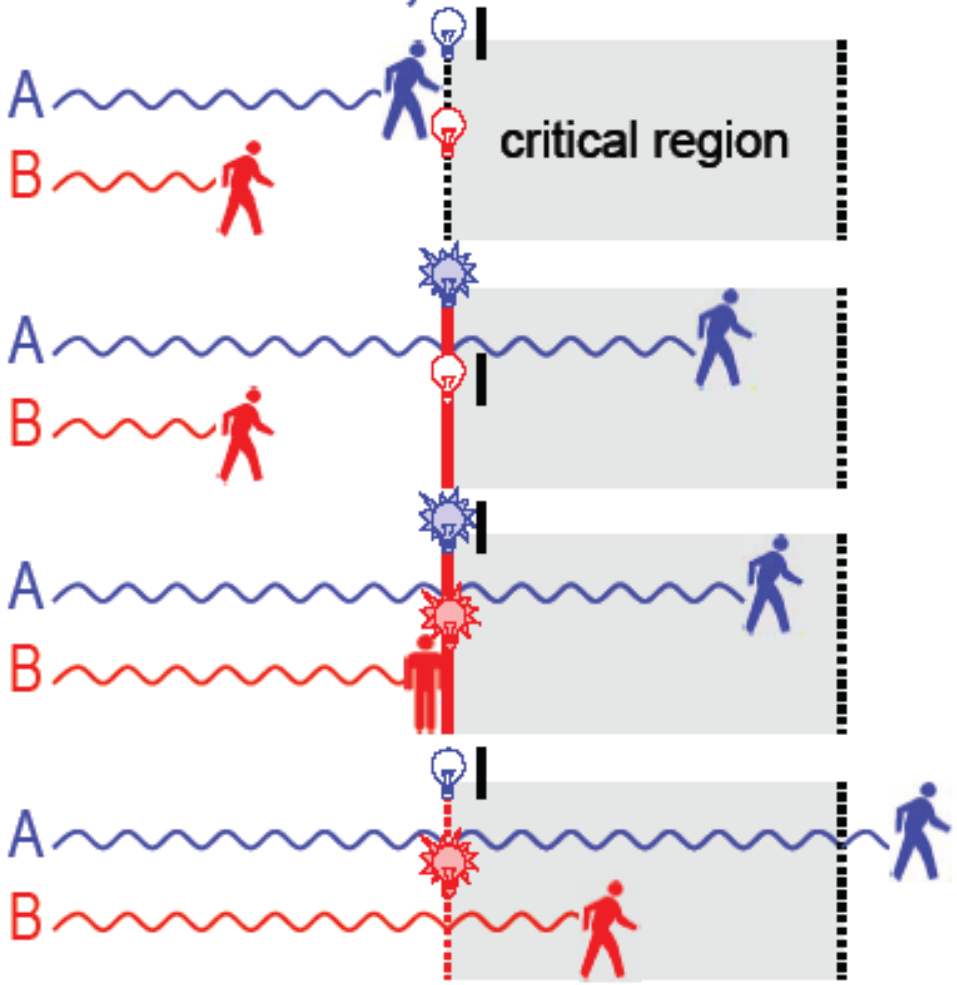


→ *this implementation avoids TSL by splitting test & set and putting them in enter & exit; nice try... but flawed!*

Mutual Exclusion by Busy Waiting

- **Implementation 4 — Peterson's no-TSL, no-alternation**

1. A and B each have their own lock; an extra toggle is also masking either lock
2. A arrives first, sets its lock, pushes the mask to the other lock and may enter
3. then, B also sets its lock & pushes the mask, but must wait until A's lock is reset
4. A exits the CR and resets its lock; B may now enter



Mutual Exclusion by Busy Waiting

➢ Implementation 4 — Peterson's no-TSL, no-alternation

- ✓ the mask & two locks are shared
- ✓ entering means: setting one's lock, pushing the mask and tetsing the other's combination
- ✓ exiting means resetting the lock

A's code

```
lock[A] = TRUE;
mask = B;
while (lock[B] &&
       mask == B);
   /* loop */
```

```
lock[A] = FALSE;
```

B's code

```
lock[B] = TRUE;
mask = A;
while (lock[A] &&
       mask == A);
   /* loop */
```

```
lock[B] = FALSE;
```

```
bool lock[2];
int mask;
int A = 0, B = 1;
void echo()
{
   char chin, chout;
   do {
      set lock, push mask, and test
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```

Mutual Exclusion by Busy Waiting

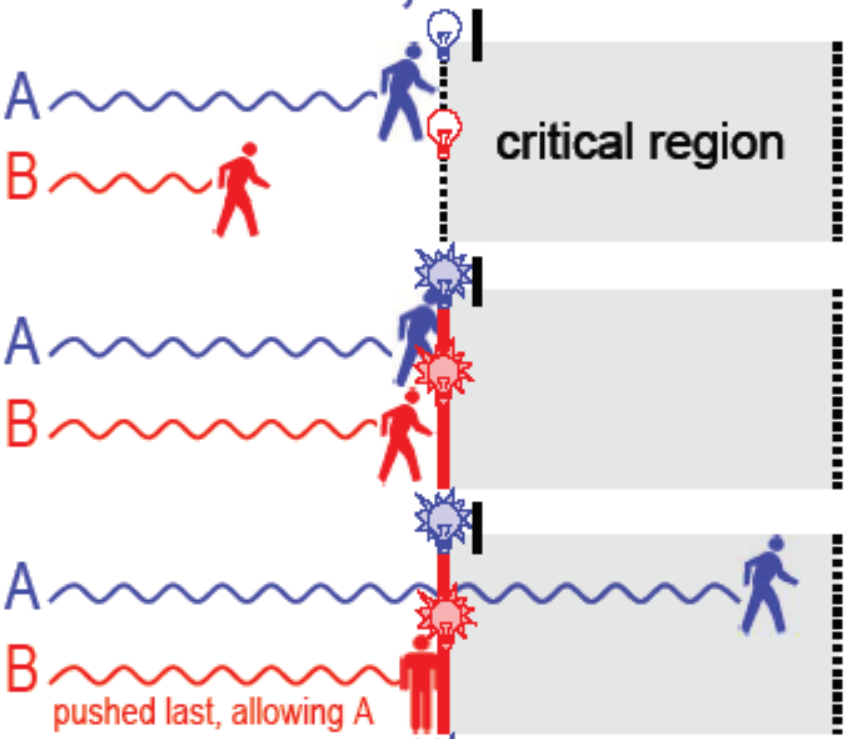
- **Implementation 4 — Peterson's no-TSL, no-alternation**

1. A and B each have their own lock; an extra toggle is also masking either lock

2.1 A is interrupted between setting the lock & pushing the mask; B sets its lock

2.2 now, both A and B race to push the mask: whoever does it last will allow the other one inside CR

→ *mutual exclusion holds!! (no bad race condition)*



Mutual Exclusion by Busy Waiting

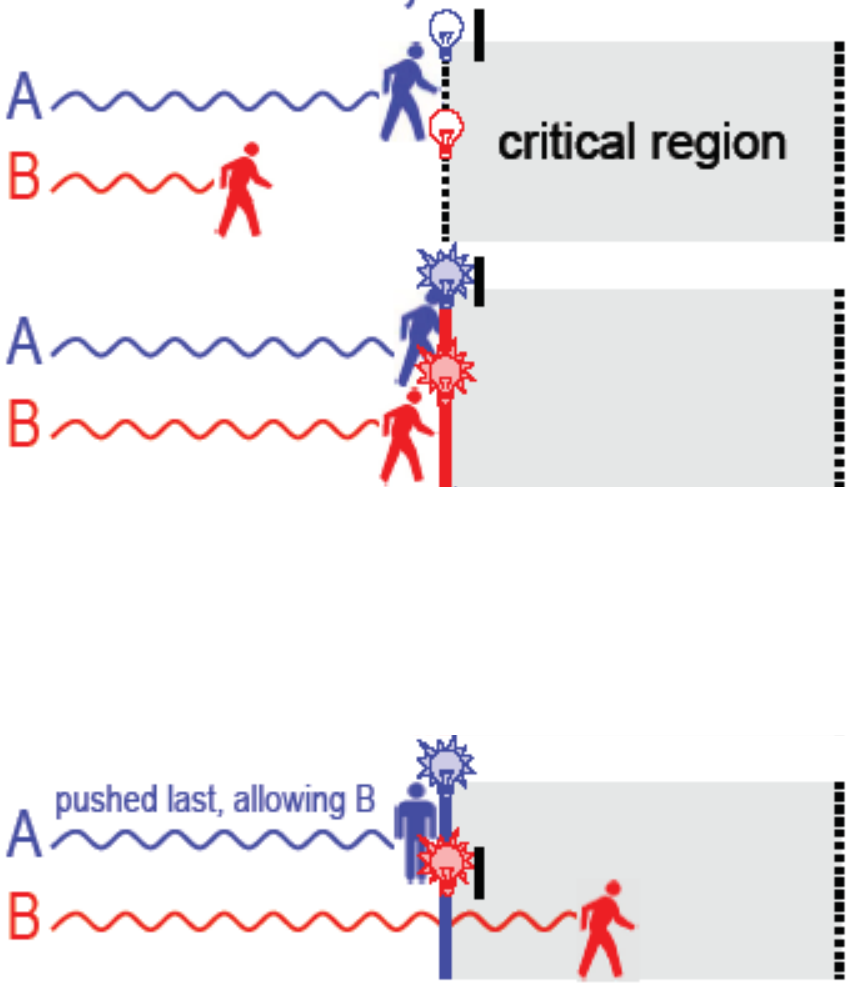
- **Implementation 4 — Peterson's no-TSL, no-alternation**

1. A and B each have their own lock; an extra toggle is also masking either lock

2.1 A is interrupted between setting the lock & pushing the mask; B sets its lock

2.2 now, both A and B race to push the mask: whoever does it last will allow the other one inside CR

→ *mutual exclusion holds!! (no bad race condition)*



Mutual Exclusion by Busy Waiting

- **Summary of these implementations of mutual exclusion**
  - ✓ **Impl. 0 — disabling hardware interrupts**
    - 👎 NO: race condition avoided, but can crash the system!
  - ✓ **Impl. 1 — simple lock variable (unprotected)**
    - 👎 NO: still suffers from race condition
  - ✓ **Impl. 2 — indivisible lock variable (TSL)**
    - 👍 YES: works, but requires hardware

    this will be the basis for "mutexes"
  - ✓ **Impl. 3 — no-TSL toggle for two threads**
    - 👎 NO: race condition avoided inside, but lockup outside
  - ✓ **Impl. 4 — Peterson's no-TSL, no-alternation**
    - 👍 YES: works in software, but processing overhead

Mutual Exclusion by Busy Waiting

- **Problem: all implementations (1-4) rely on busy waiting**
  - ✓ "busy waiting" means that the process/thread continuously executes a tight loop until some condition changes
  - ✓ busy waiting is bad:
    - **waste of CPU time** — the busy process is not doing anything useful, yet remains "Ready" instead of "Blocked"
    - **paradox of inversed priority** — by looping indefinitely, a higher-priority process B may starve a lower-priority process A, thus preventing A from exiting CR and . . . liberating B! (B is working against its own interest)

→ *we need for the waiting process to block, not keep idling*

Common Concurrency Mechanisms

`Semaphore`	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Mutual Exclusion & Synchronization – Semaphores and Mutexes

➢ Implementation 2' — indivisible blocking lock = mutex

- ✓ a mutex is a safe lock variable with blocking, instead of tight looping
- ✓ if **TSL** returns 1, then voluntarily yield the CPU to another thread

```
mutex_lock:
    TSL REGISTER,MUTEX  | copy mutex to register and set mutex to 1
    CMP REGISTER,#0     | was mutex zero?
    JZE ok              | if it was zero, mutex was unlocked, so return
    CALL thread_yield   | mutex is busy; schedule another thread
    JMP mutex_lock      | try again later
ok: RET                 | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0       | store a 0 in mutex
    RET                 | return to caller
```

```
void echo()
{
    char chin, chout;
    do {
        test-and-set-lock or BLOCK
        chin = getchar();
        chout = chin;
        putchar(chout);
        set lock off
    }
    while (...);
}
```

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Mutual Exclusion & Synchronization – Semaphores and Mutexes

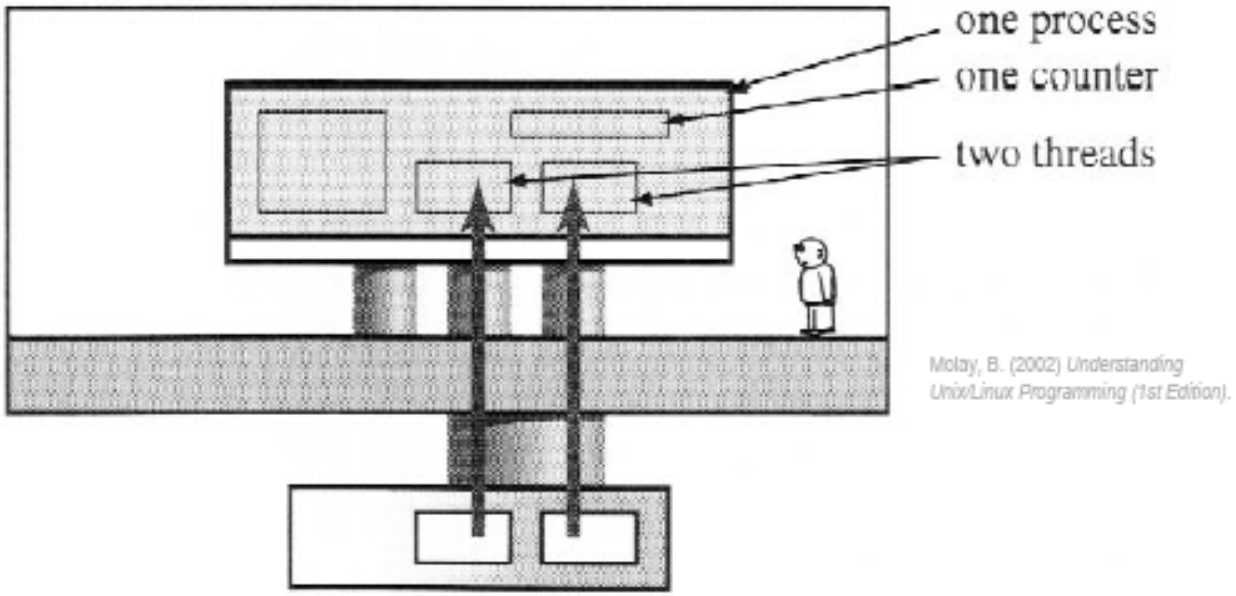
Difference between busy waiting and blocked

- ✓ in busy waiting, the PC is always looping (increment & jump back)
- ✓ it can be preemptively interrupted but will loop again tightly whenever rescheduled → *tight polling*

- ✓ when blocked, the process's PC stalls after executing a "yield" call
- ✓ either the process is only timed out, thus it is "Ready" to loop-and-yield again → *sparse polling*
- ✓ or it is truly "Blocked" and put in event queue → *condition waiting*

Mutual Exclusion & Synchronization – Semaphores and Mutexes

- **Illustration of mutex use: shared word counter**
  - ✓ we want to count the total number of words in 2 files
  - ✓ we use 1 global counter variable and 2 threads: each thread reads from a different file and increments the shared counter



A common counter for two threads

Mutual Exclusion & Synchronization – Semaphores and Mutexes

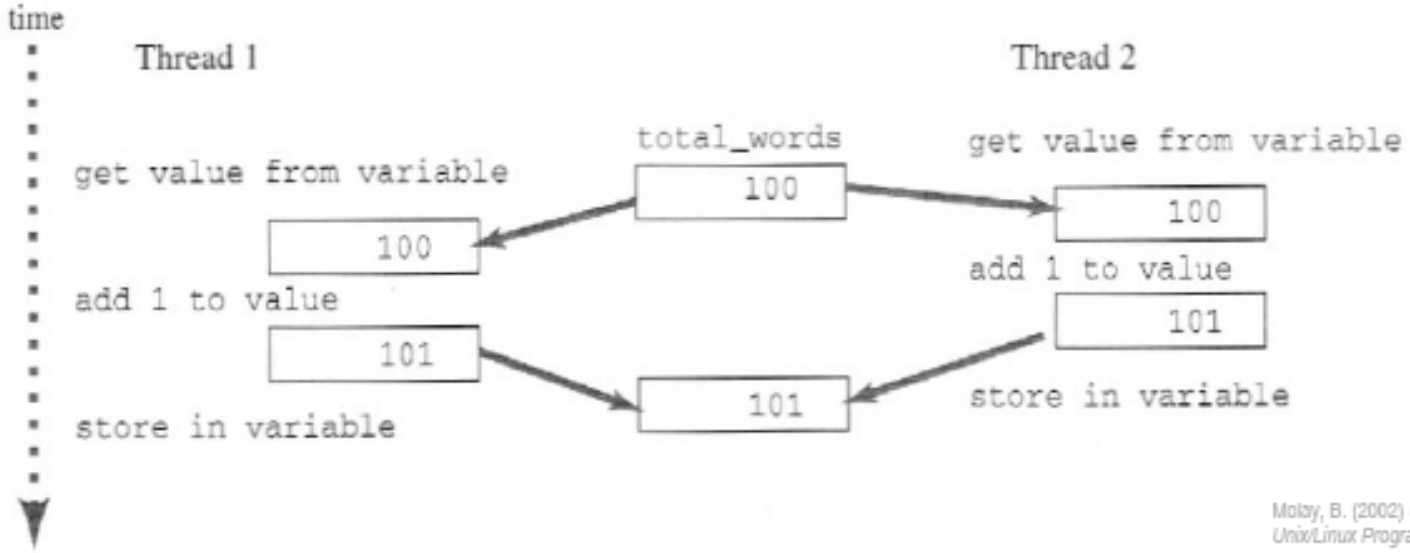
```
int total_words;

void main(...)
{
    ...declare, initialize...
    pthread_create(&th1, NULL, count_words, (void *)filename1);
    pthread_create(&th2, NULL, count_words, (void *)filename2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("total words = %d", total_words);
}

void *count_words(void *filename)
{
    ...open file...
    while (...get next char...) {
        if (...char is not alphanum & previous char is alphanum...) {
            total_words++;
        }
        ......
```

`total_words = total_words + 1;`
is not necessarily atomic! (depends on machine code and stage of execution)

Multithreaded shared counter with possible race condition

Mutual Exclusion & Synchronization – Semaphores and Mutexes

- **A race condition can occur when incrementing counter**
  - ✓ if not atomic, the increment block of thread 1, "get1-add1" may be interleaved with the increment block of thread 2, "get2-add2" to produce "get1-get2-add1-add2" or "get1-get2-add2-add1"
  - → *this results in missing one count*



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

Two threads race to increment the counter

Mutual Exclusion & Synchronization – Semaphores and Mutexes

```
int total_words;
pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;

void main(int ac, char *av[])
{
    ...declare, initialize...
    pthread_create(&th1, NULL, count_words, (void *)filename1);
    pthread_create(&th2, NULL, count_words, (void *)filename2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("total words = %d", total_words);
}

void *count_words(void *filename)
{
    ...open file...
    while (...get next char...) {
        if (...char is not alphanum & previous char is alphanum...) {
            pthread_mutex_lock(&counter_lock);
            total_words++;
            pthread_mutex_unlock(&counter_lock);
        }
        ......
```

protect the critical region with mutual exclusion

Mulithreaded shared counter with mutex protection

Mutual Exclusion & Synchronization - Mutexes

System calls for thread exclusion with mutexes

- ✓ `err = pthread_mutex_lock(pthread_mutex_t *m)`

  locks the specified mutex

  - if the mutex is unlocked, it becomes locked and owned by the calling thread
  - if the mutex is already locked by another thread, the calling thread is blocked until the mutex is unlocked

- ✓ `err = pthread_mutex_unlock(pthread_mutex_t *m)`

  releases the lock on the specified mutex

  - if there are threads blocked on the specified mutex, one of them will acquire the lock to the mutex

Semaphore

A variable that has an integer value upon which only three operations are defined:

There is no way to inspect or manipulate semaphores other than these three operations

1) **May be initialized to a nonnegative integer value**
2) **The `semWait` operation decrements the value**
3) **The `semSignal` operation increments the value**

Semaphores

- Special variable called a semaphore is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent
- Wait and signal operations cannot be interrupted
- Queue is used to hold processes waiting on the semaphore

Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

A Metaphor

Original Notation

P = wait

V = signal

- Paula and Victor (P and V) are working in a restaurant:
  - Paula handles new arrivals: her main concern is to prevent people from entering the restaurant when all tables are busy.
  - Victor handles departures: his main concern is to notify people waiting to enter the restaurant that there is a table available.
- The semaphore represents the ***number of available tables***; it is initialized with the total number of tables in restaurant.

A Metaphor

Original Notation

P = wait

V = signal

- When people wish to enter the restaurant, they wait for Paula to direct them:
  - If the semaphore is **> 0**, she let them in and decrements the semaphore.
  - Otherwise, she directs them to the waiting area.
- When people leave, they tell Victor:
  - Victor increments the semaphore and checks the waiting area: if there is anyone in there, he lets one group in and decrements the semaphore.

A Metaphor

Original Notation

P = wait

V = signal

- Paula and Victor have worked long enough together so that they don't interfere with each other.
- There are still several possible problems:
  - What if somebody sneaks in the restaurant and bypasses Paula?
    - *Paula will let a group of people in when all tables are busy.*
  - What if people forget to tell Victor they are leaving?
    - *Their table will never be reassigned.*
- Paula and Victor must trust the restaurant patrons.

Mutual Exclusion & Synchronization - Semaphores

- **Semaphores are used for signaling between processes**
  - ✓ semaphores can be used for **mutual exclusion**
  - ✓ binary semaphores are the same as mutexes
  - ✓ integer semaphores can be used to allow more than one process inside a critical region; generally:
    - the positive value of an integer semaphore corresponds to a maximum number of processes allowed concurrently inside a critical region
    - the negative value of an integer semaphore corresponds to the number of processes currently waiting in the queue
  - ✓ binary and integer semaphores can also be used for **synchronization**

Semaphore Primitives

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

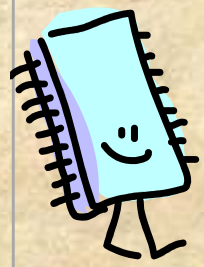
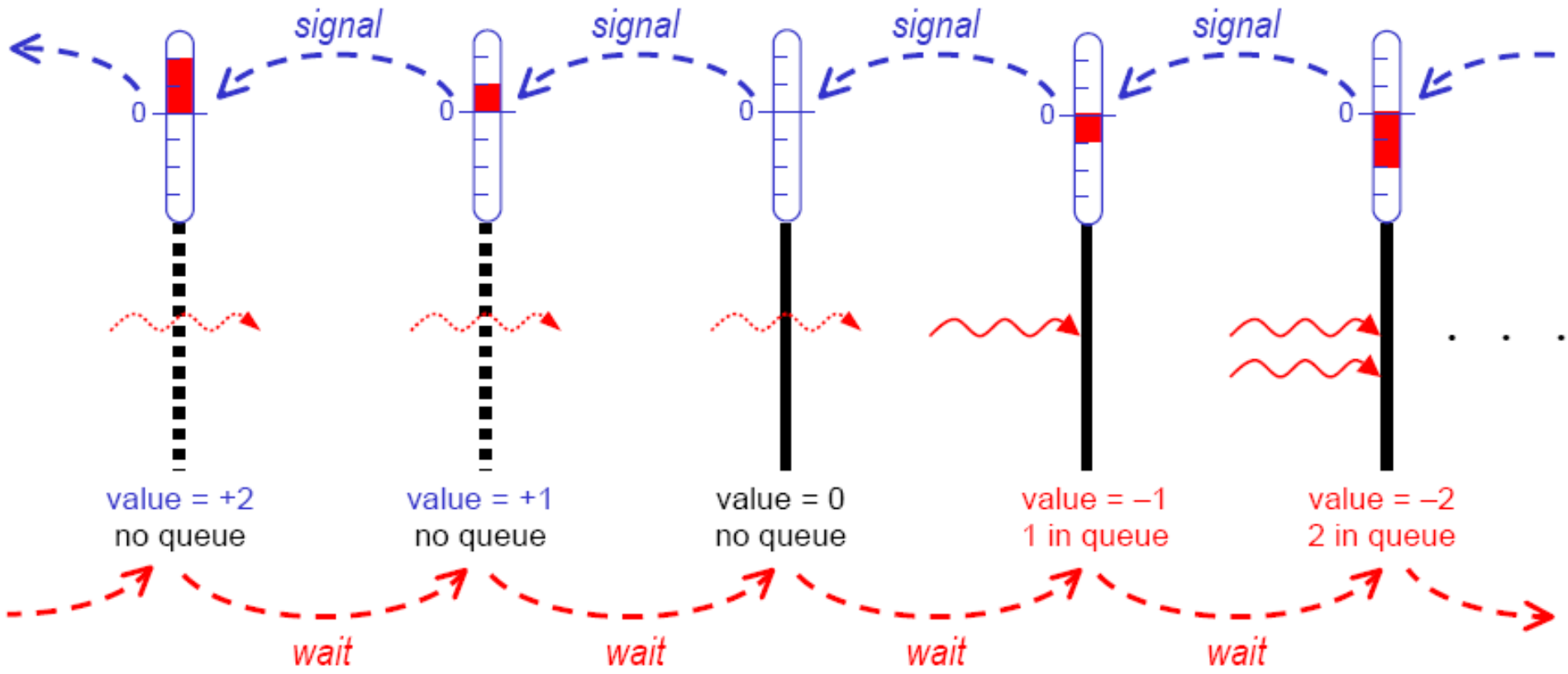


Figure 5.3 A Definition of Semaphore Primitives

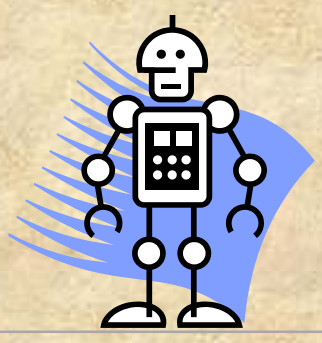
Mutual Exclusion & Synchronization - Semaphores

- **Integer semaphore ⇔ "thermometer"**



Implementation of Semaphores

- Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives
- Can be implemented in hardware or firmware
- Software schemes such as Dekker's or Peterson's algorithms can be used
- Use one of the hardware-supported schemes for mutual exclusion



Implementing Semaphore Primitives

```
void semWait(semaphore s)
{
    while (!testset(s.flag))
        /* do nothing */;
    s.count--;
    if (s.count < 0)
    {
        place this process in s.queue;
        block this process (must also set s.flag to 0)
    }
    else
        s.flag = 0;
}

void semSignal(semaphore s)
{
    while (!testset(s.flag))
        /* do nothing */;
    s.count++;
    if (s.count <= 0)
    {
        remove a process P from s.queue;
        place process P on ready list
    }
    s.flag = 0;
}
```

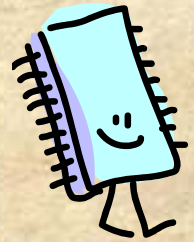
Implementing Semaphore Primitives

```
void semWait(semaphore s)
{
        inhibit interrupts;
        s.count--;
        if (s.count < 0)
        {
                place this process in s.queue;
                block this process and allow interrupts
        }
        else
            allow interrupts;
}

void semSignal(semaphore s)
{
        inhibit interrupts;
        s.count++;
        if (s.count <= 0)
        {
                remove a process P from s.queue;
                place process P on ready list
        }
        allow interrupts;
}
```

Binary Semaphore Primitives

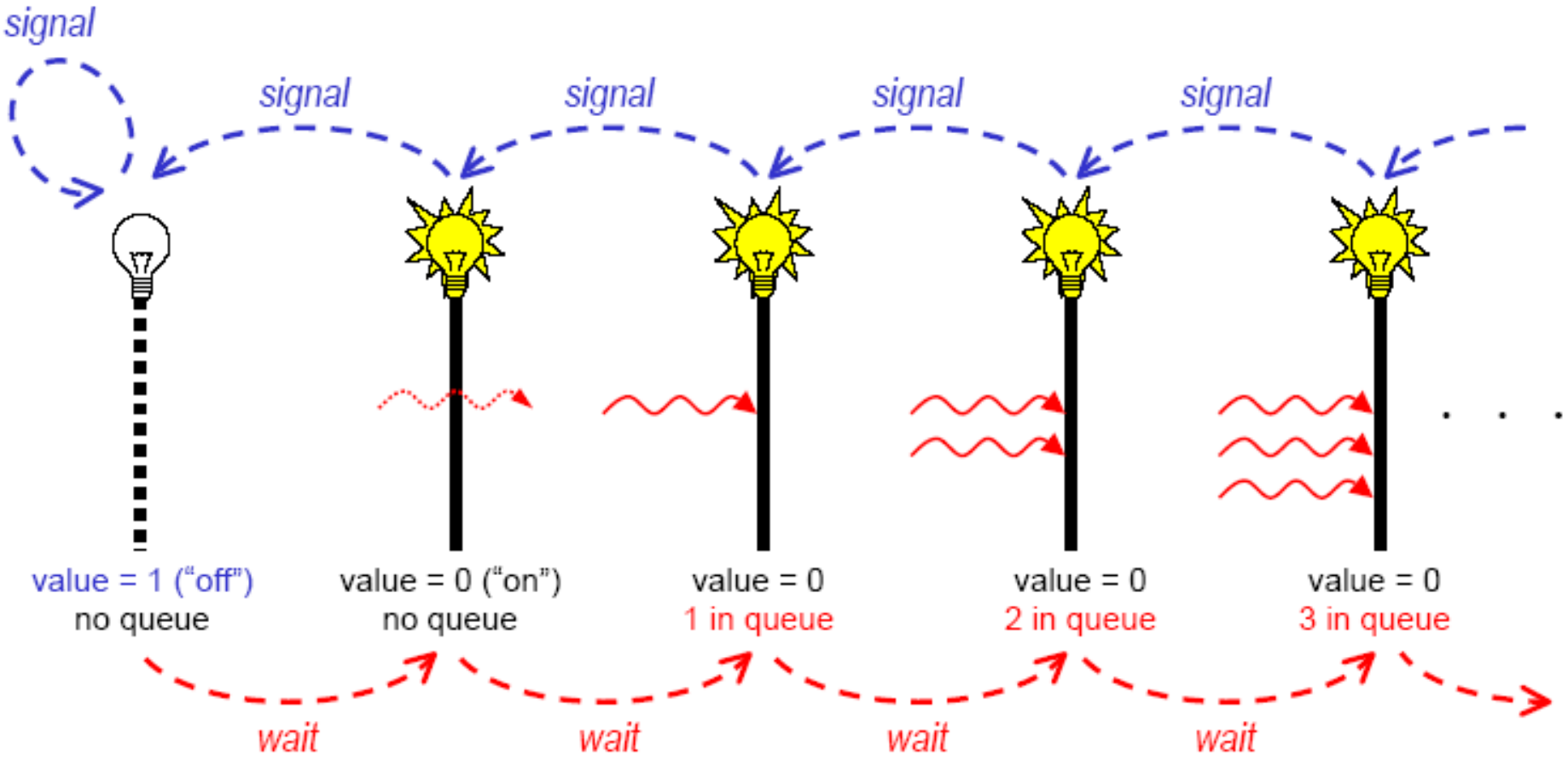
```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore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

Figure 5.4 A Definition of Binary Semaphore Primitives



Mutual Exclusion & Synchronization - Mutexes

- **Binary semaphore ⇔ mutex**



Binary Semaphores

```
wait(s)
{
    while (!testset(s.flag))
    /* do nothing */;
    if (s.count == 1)
            s.count--;
    else
    {
            place this process in s.queue;
            block this process (must also set
    s.flag to 0)
    }
    s.flag = 0;
}
```

```
signal(s)
{
    while (!testset(s.flag))
    /* do nothing */;
    s.count = 1;
    if (s.queue is not empty)
    {
            remove a process P from s.queue;
            place process P on ready list
            s.count = 0;
    }
    s.flag = 0;
}
```

Mutual Exclusion & Synchronization - Mutexes

- **Binary semaphore ⇔ mutex**
  - ✓ a binary semaphore is a variable that has a value 0 or 1
  - ✓ a **wait** operation attempts to decrement the semaphore
    - 1 → 0 and goes through; 0 → blocks
  - ✓ a **signal** operation attempts to increment the semaphore
    - 1 → 1, no change; 0 → unblocks or becomes 1

Strong/Weak Semaphores

😐 A queue is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified

Example of Semaphore Mechanism

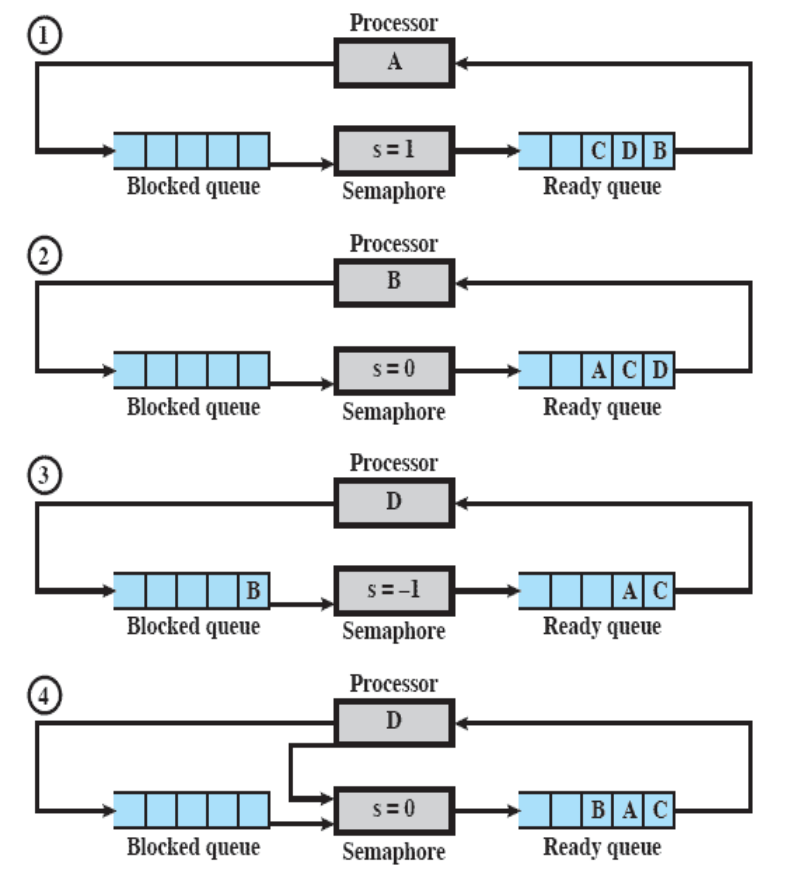


Figure 5.5 Example of Semaphore Mechanism

Mutual Exclusion

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

Shared Data Protected by a Semaphore

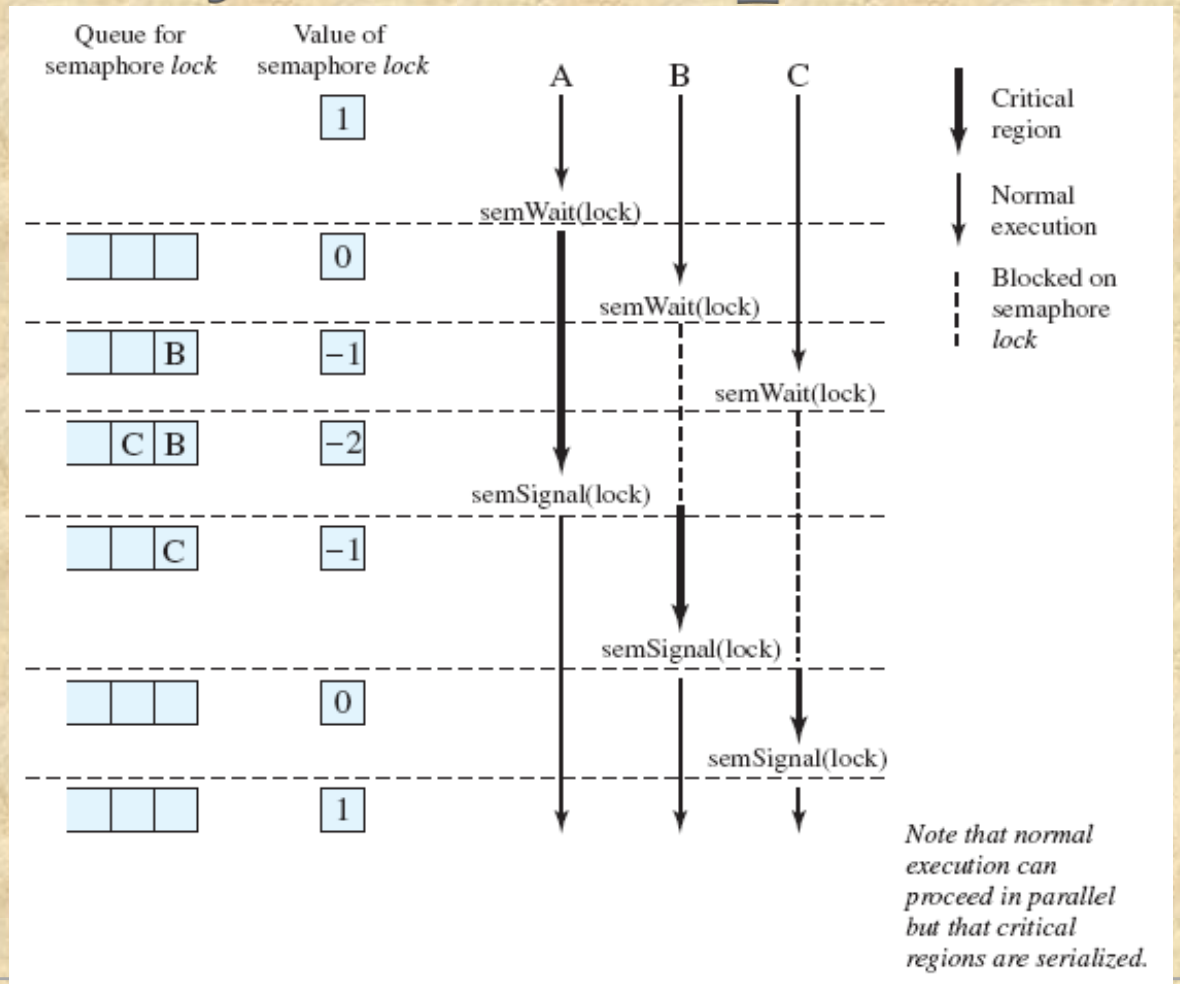


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

Producer/Consumer Problem

General Situation:

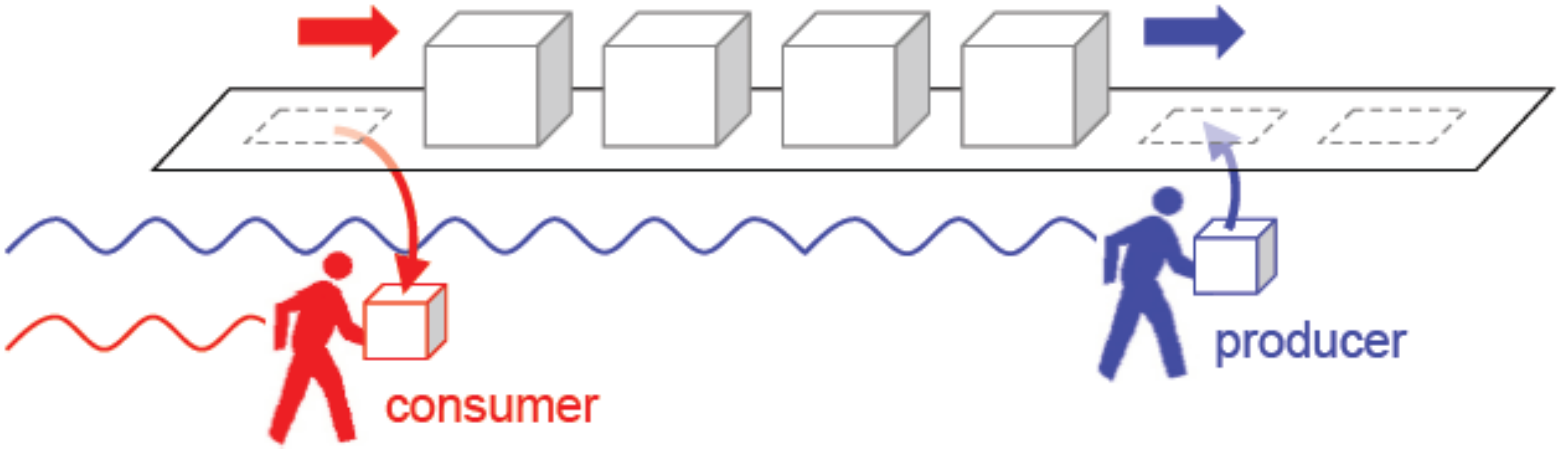
- one or more producers are generating data and placing these in a buffer
- a single consumer is taking items out of the buffer one at time
- only one producer or consumer may access the buffer at any one time

The Problem:

- ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

Mutual Exclusion & Synchronization - Mutexes

- **Real-world mutex use: the producer/consumer problem**
  - ✓ **producer** — generates data items and places them in a buffer
  - ✓ **consumer** — takes the items out of the buffer to use them
  - ✓ example 1: a print program produces characters that are consumed by a printer
  - ✓ example 2: an assembler produces object modules that are consumed by a loader



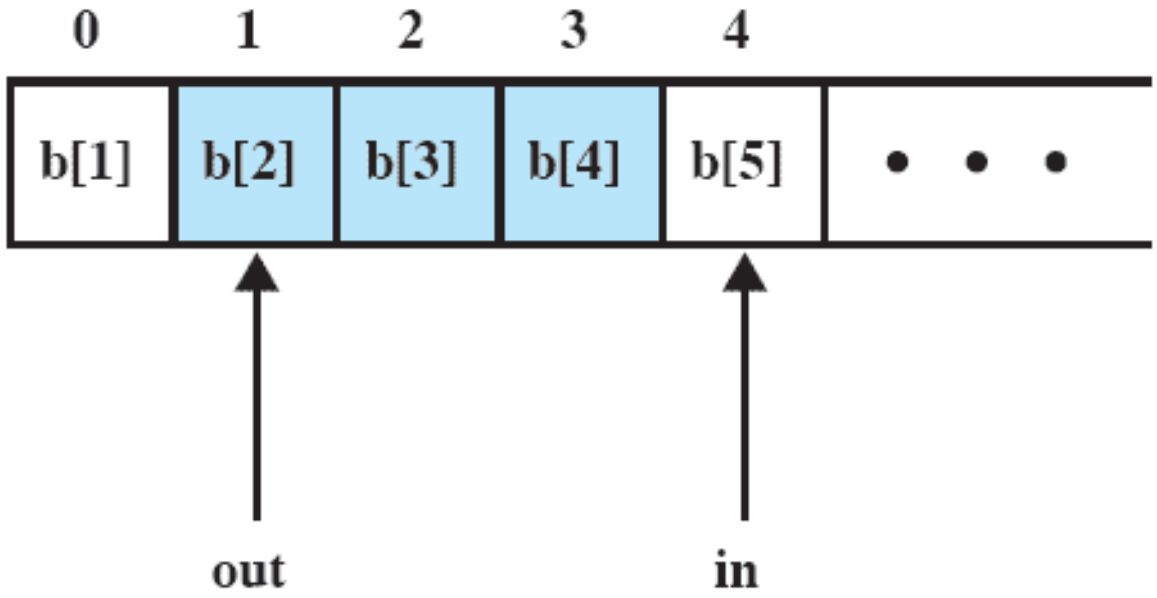
Producer

```
producer:
while (true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

Consumer

```
consumer:
while (true) {
 while (in <= out)
    /*do  nothing */;
 w = b[out];
 out++;
 /* consume item w */
}
```

Buffer Structure

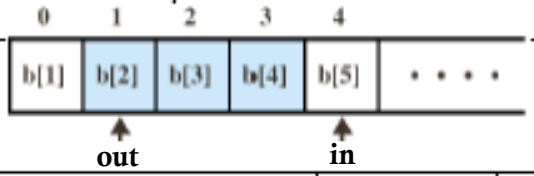


Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Mutual Exclusion & Synchronization – Mutexes Only

- **Unbounded buffer, 1 producer, 1 consumer**
  - ✓ `in` modified only by producer and `out` only by consumer
  - ✓ no race condition; no need for mutexes, just a while loop



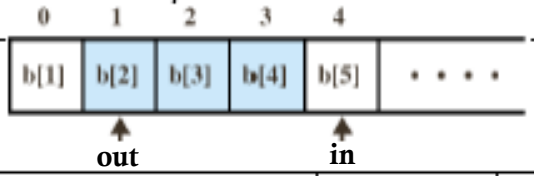

```
item[] b;
int in, out;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

```
void consumer()
{
  while (true) {
    while (out == in);

    item = b[out];
    out++;

    consume(item);
  }
}
```

Mutual Exclusion & Synchronization – Mutexes Only

- **Unbounded buffer, 1 producer, *N* consumers**
  - ✓ `in` modified only by producer and `out` only by consumer
  - ~~✓ no race condition; no need for mutexes, just a while loop~~



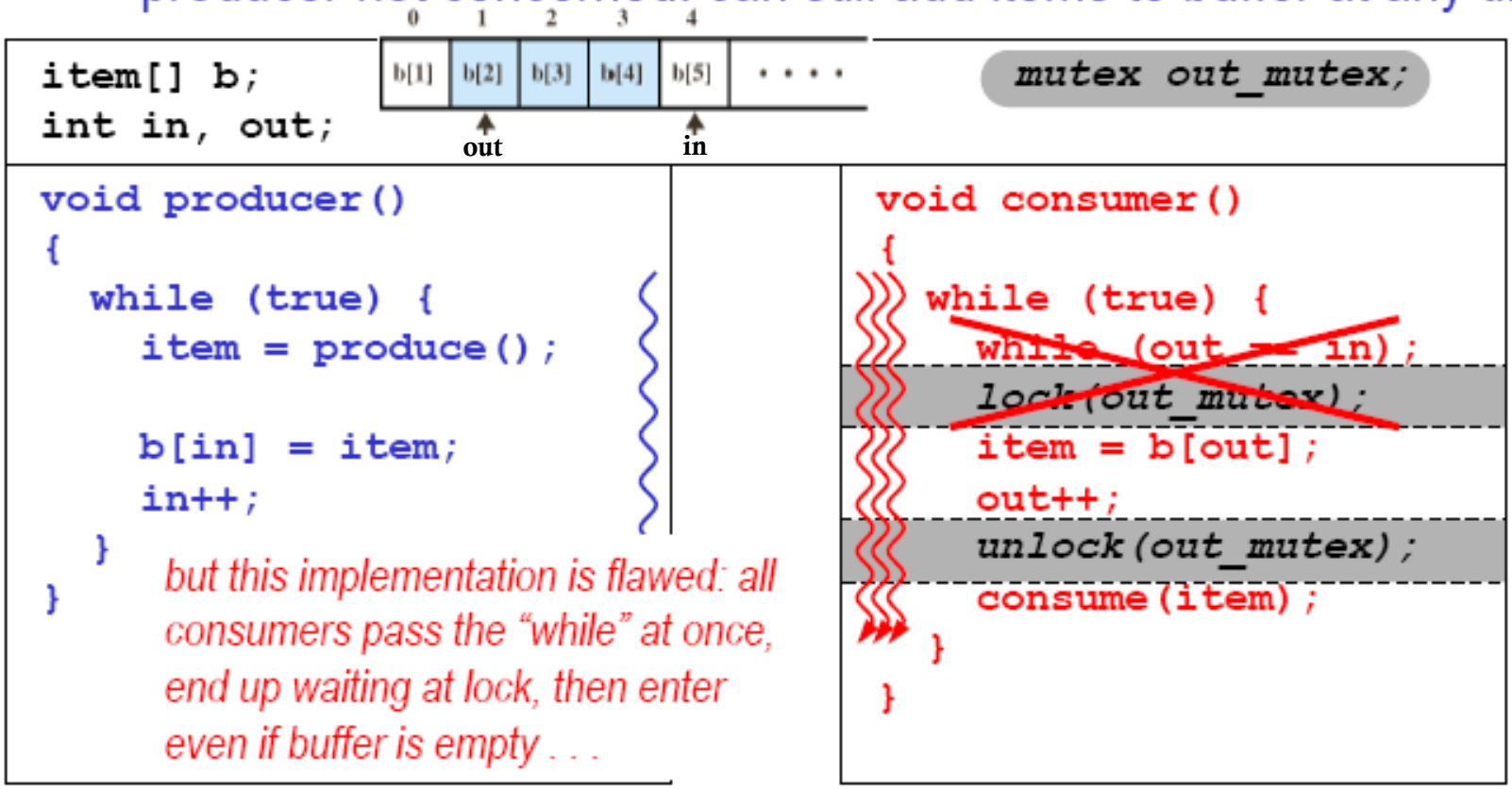

```
item[] b;
int in, out;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

```
void consumer()
{
  while (true) {
    while (out == in);
    lock(out_mutex);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

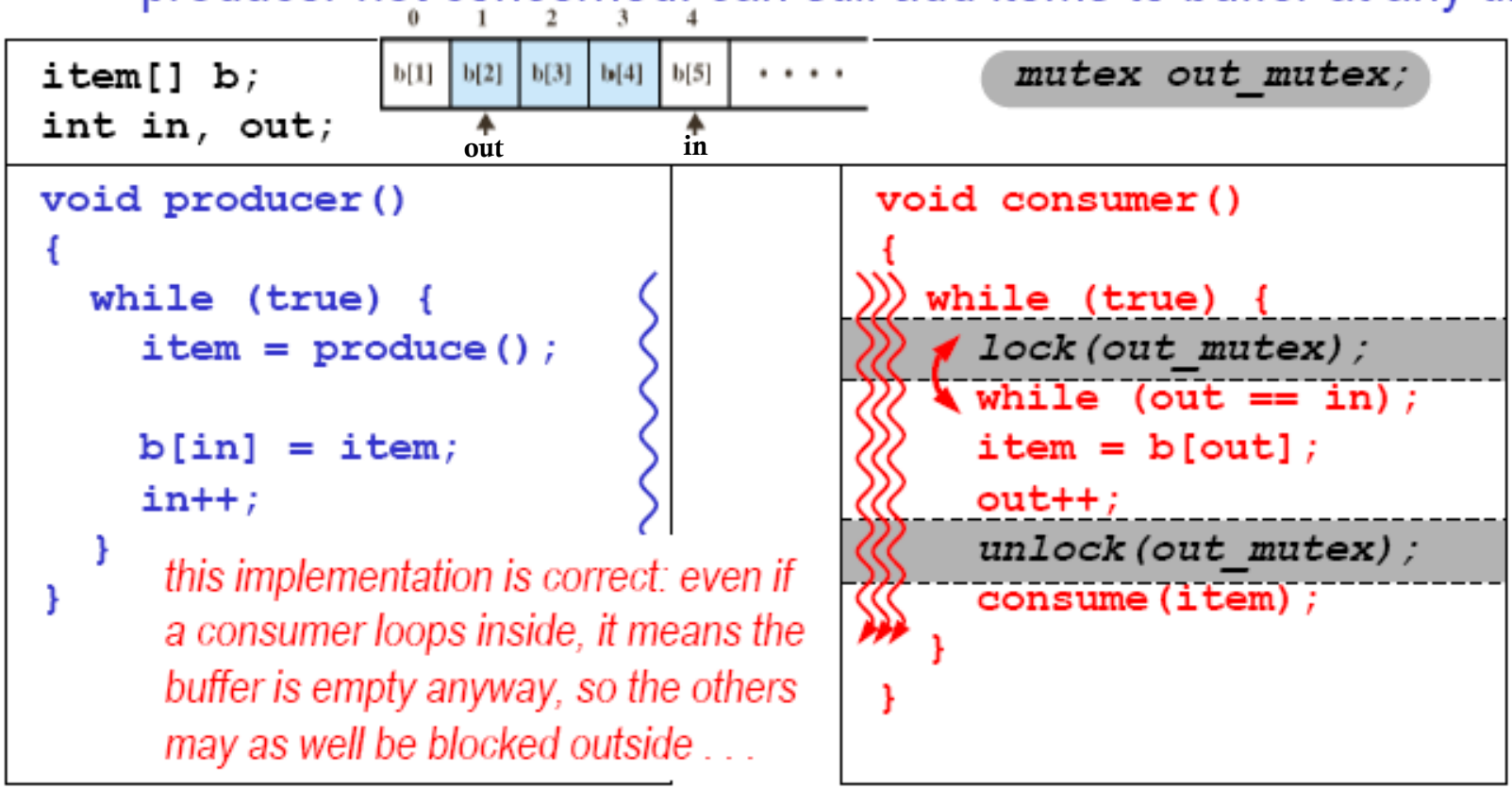
Mutual Exclusion & Synchronization – Mutexes Only

- **Unbounded buffer, 1 producer, *N* consumers**
  - ✓ `out` shared by all consumers → mutex among consumers
  - ✓ producer not concerned: can still add items to buffer at any time



Mutual Exclusion & Synchronization – Mutexes Only

- **Unbounded buffer, 1 producer, *N* consumers**
  - ✓ `out` shared by all consumers → mutex among consumers
  - ✓ producer not concerned: can still add items to buffer at any time



Mutual Exclusion & Synchronization – Mutexes Only

- **Unbounded buffer, *N* producers, *N* consumers**
  - ✓ `in` shared by all producers → other mutex among producers
  - ✓ consumers and producers still (relatively) independent


```
item[] b;
int in, out;
```

```
mutex out_mutex;
mutex in_mutex;
```

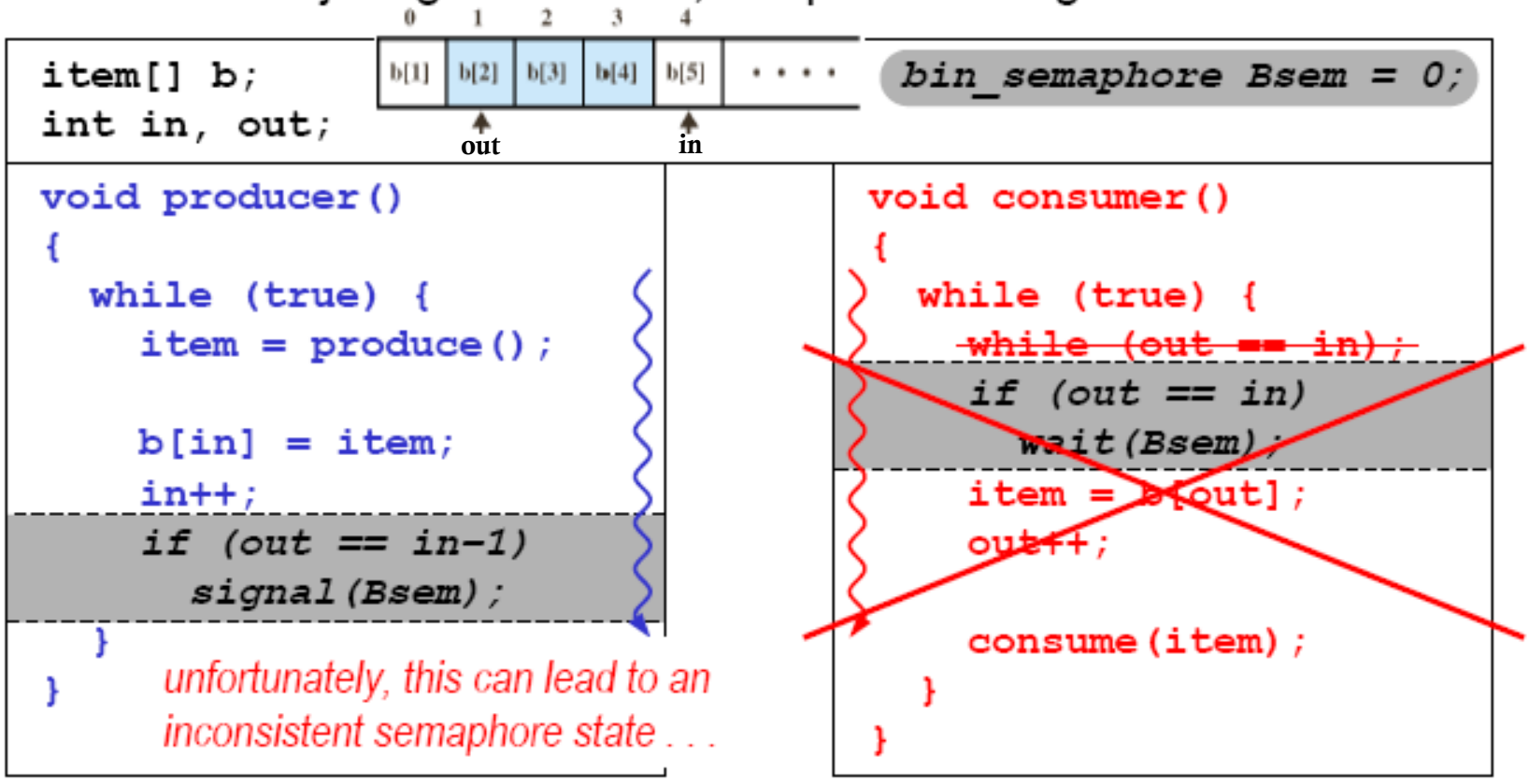
```
void producer()
{
  while (true) {
    item = produce();
    lock(in_mutex);
    b[in] = item;
    in++;
    unlock(in_mutex);
  }
}
```

```
void consumer()
{
  while (true) {
    lock(out_mutex);
    while (out == in);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

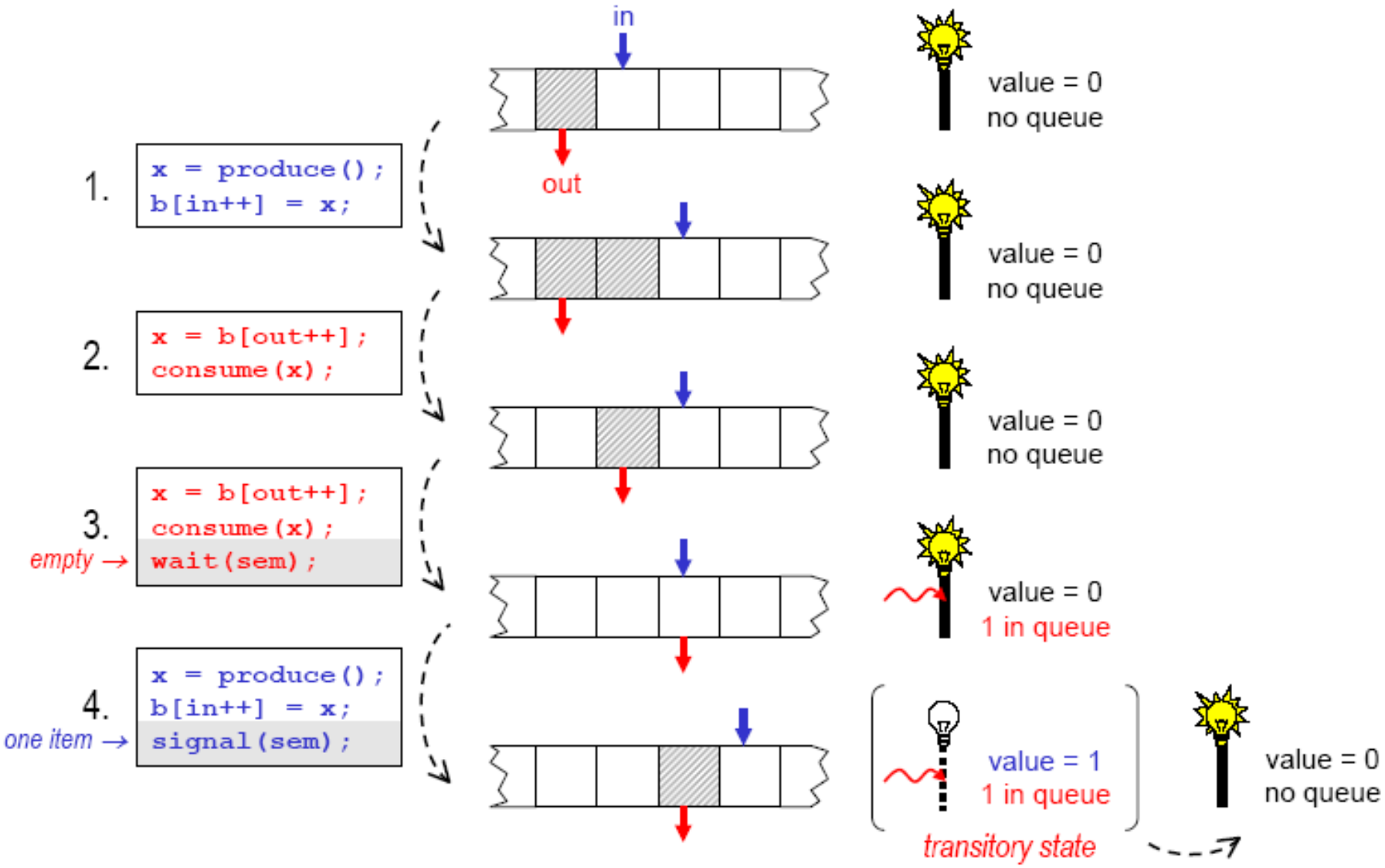
still correct, but in all cases the consumers are busy waiting . . .

Mutual Exclusion & Synchronization – Mutexes and Binary Semaphores

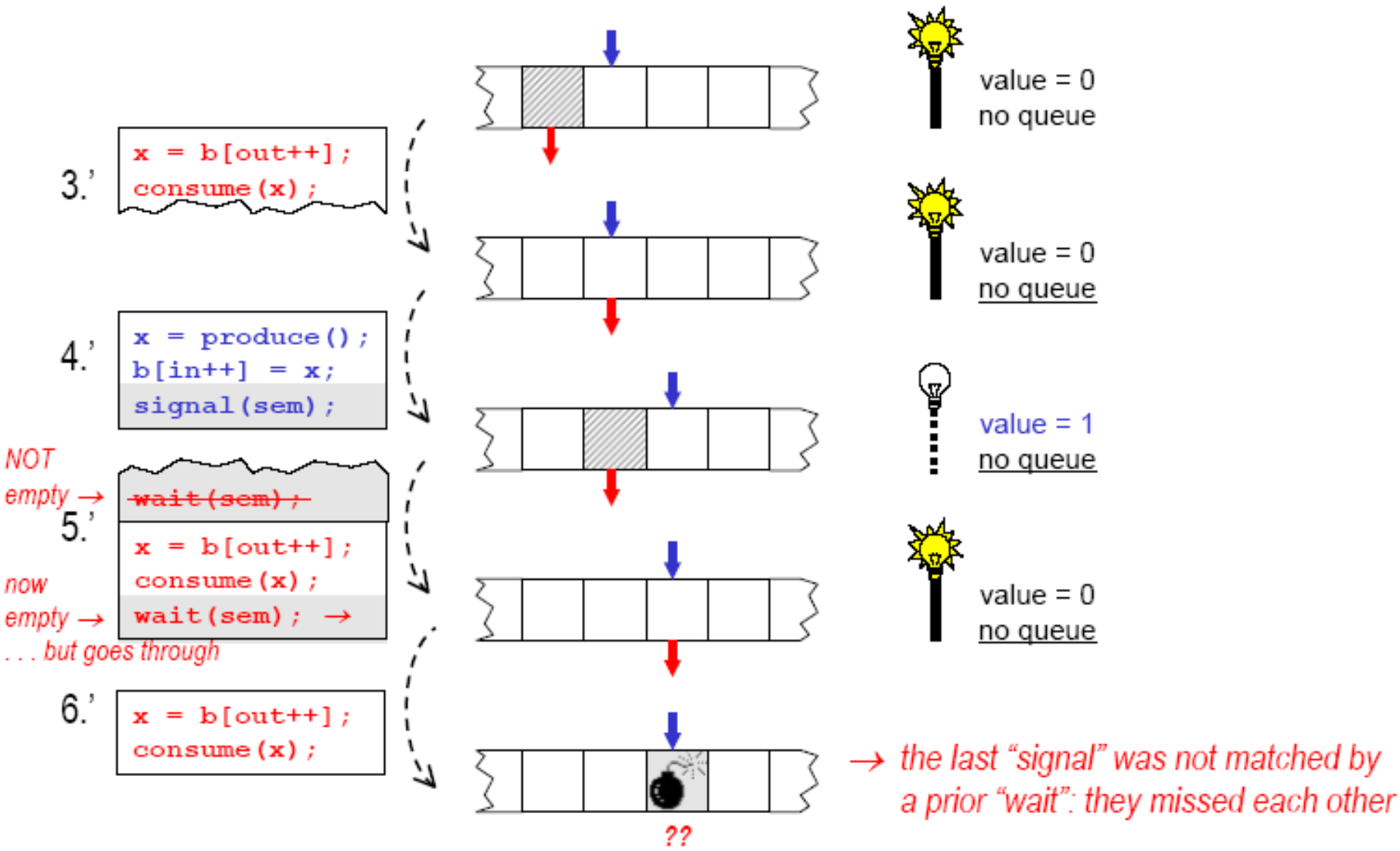
- **Unbounded buffer, 1 producer, 1 consumer with sync**
  - ✓ if buffer is empty, the consumer waits on a semaphore
  - ✓ if buffer just got one item, the producer signals to the consumer



Mutual Exclusion & Synchronization – Mutexes and Binary Semaphores

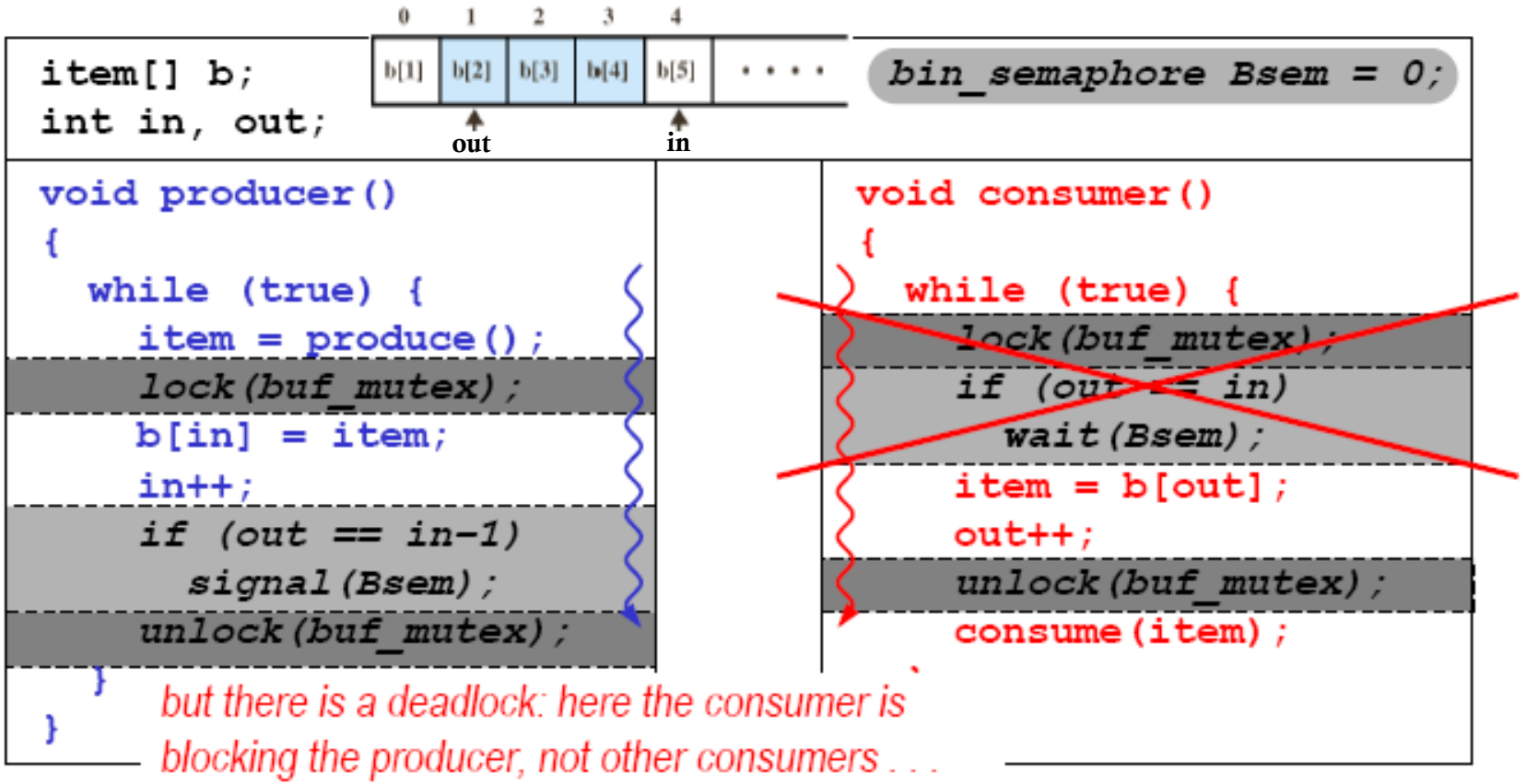


Mutual Exclusion & Synchronization – Mutexes and Binary Semaphores



Mutual Exclusion & Synchronization – Mutexes and Binary Semaphores

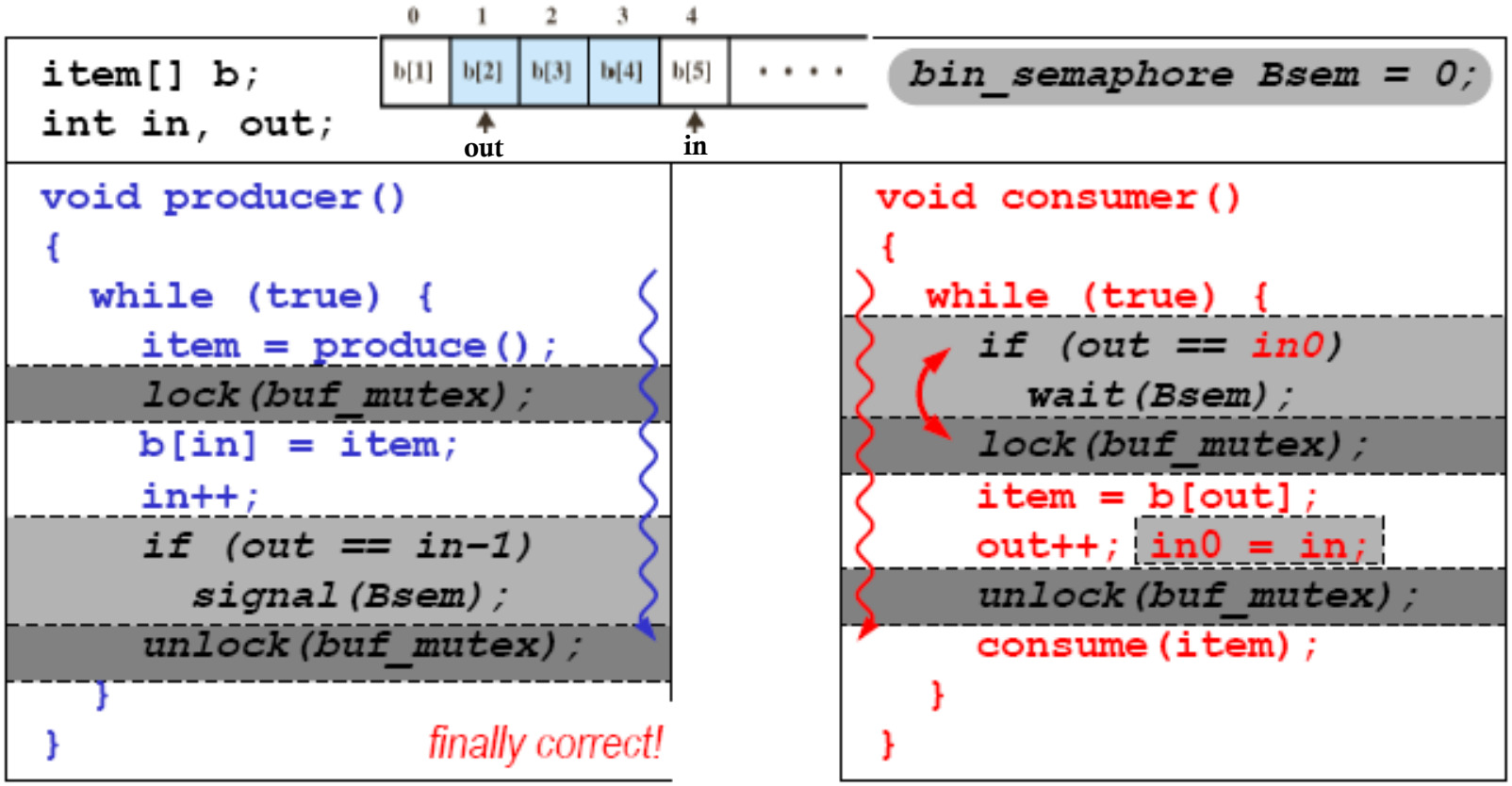
- **Unbounded buffer, 1 producer, 1 consumer with sync**
  - ✓ we need to create critical areas to keep "consuming" and "checking the semaphore" together



Mutual Exclusion & Synchronization – Mutexes and Binary Semaphores

➢ Unbounded buffer, 1 producer, 1 consumer with sync

✓ the consumer needs to remember the current state of `in` & `out`, so it can exit the CR before checking the semaphore



Incorrect Solution

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

produce() generates an item

append() puts the item in the buffer

take() removes an item from the buffer

consume() uses the item

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Possible scenario

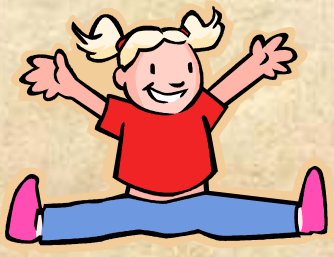
Table 5.4 Possible Scenario for the Program of Figure 5.9

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

NOTE: White areas represent the critical section controlled by semaphore s.

Note: White areas represent the critical section controlled by semaphore s.

Correct Solution



```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
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);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Solution Using Semaphores

```
/* program  producerconsumer */
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();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Semaphore n acts as a counter of the number of items in the buffer

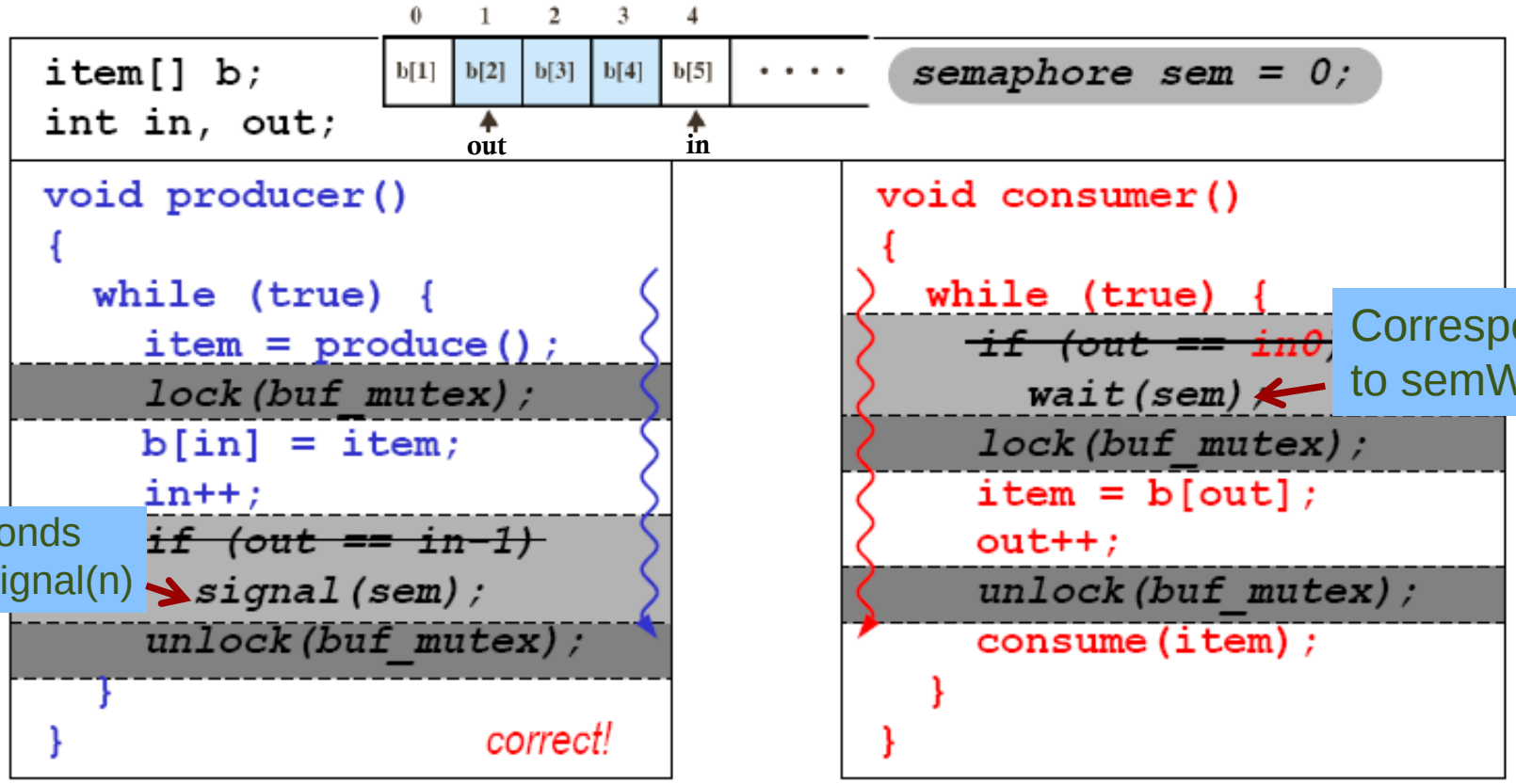
Semaphore s enforces mutual exclusion to the critical section (it could just as well be a binary semaphore (mutex)

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

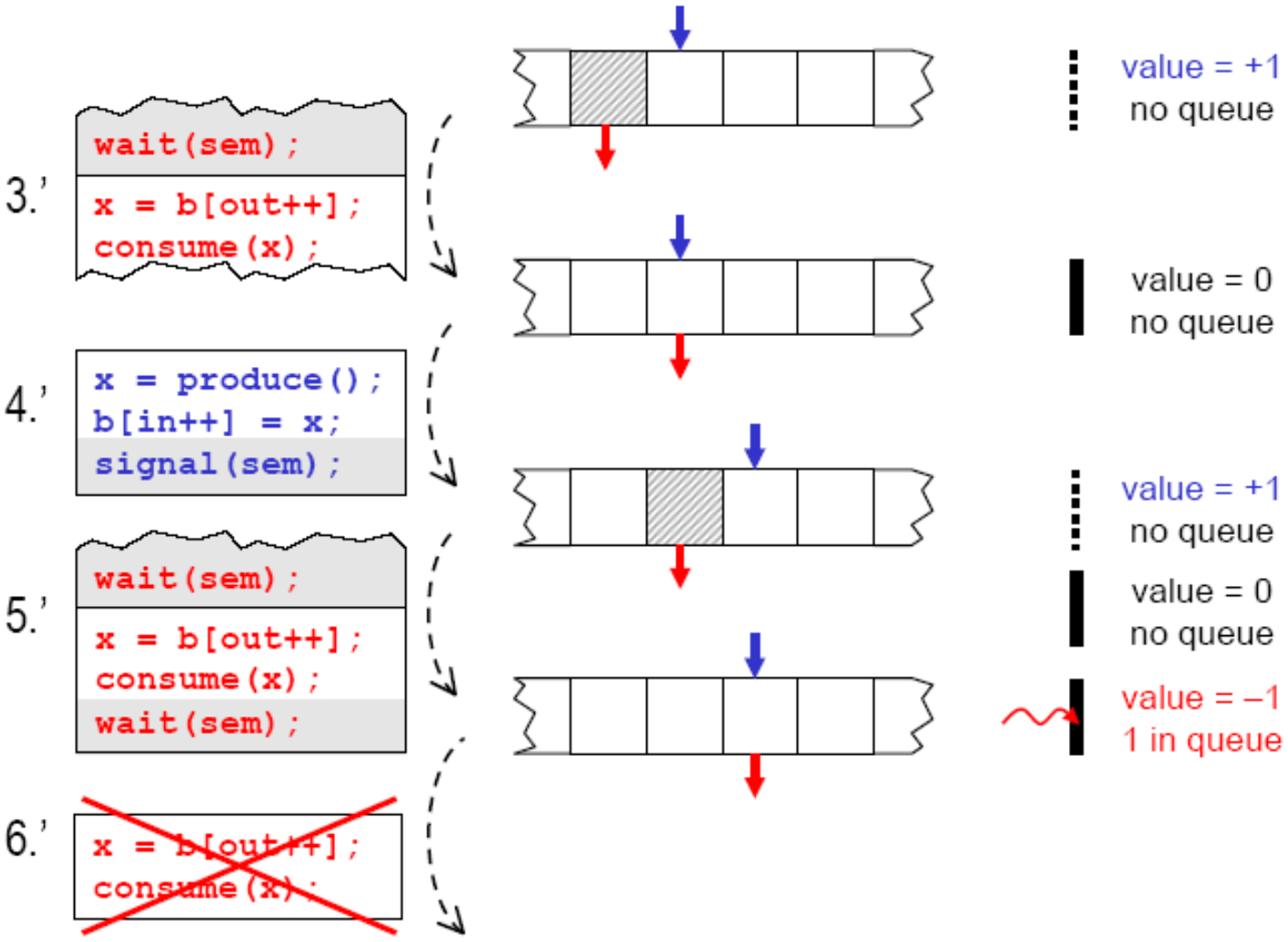
Mutual Exclusion & Synchronization - Semaphores

Producer/consumer with an integer semaphore

- ✓ no need for a condition: the semaphore itself keeps track of the size of the buffer



Mutual Exclusion & Synchronization - Semaphores



the consumer is blocked, as it should be; the producer may proceed . . .

Finite Circular Buffer

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

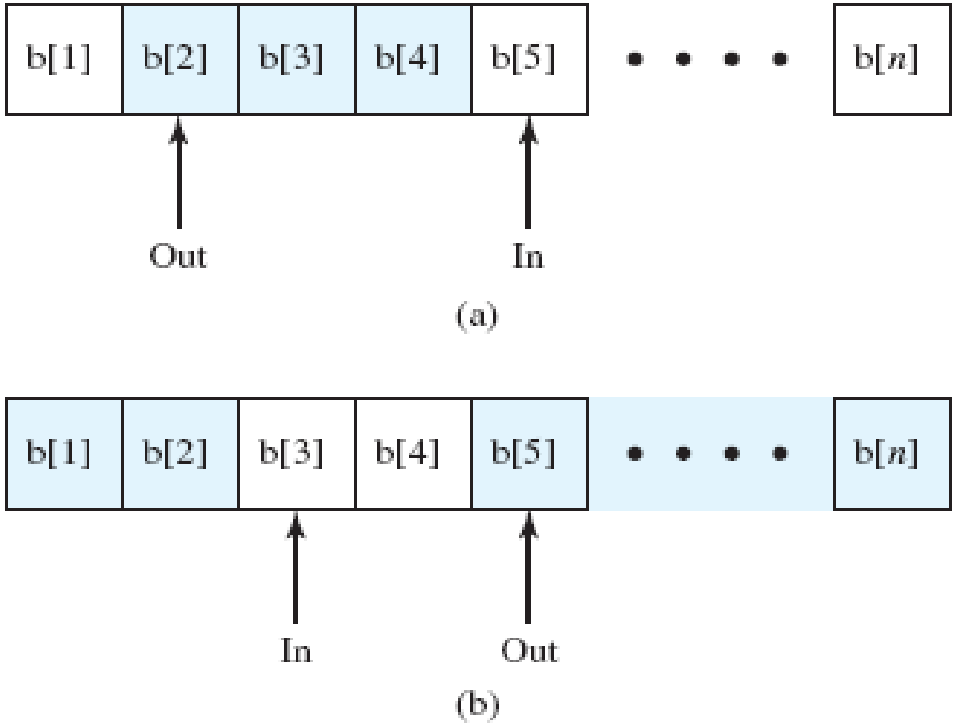


Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

Producer with Circular Buffer

producer:

```
while (true) {
   /* produce item v */
   while ((in + 1) % n == out)
         /* do nothing */;
   b[in] = v;
   in = (in + 1) % n
}
```

Consumer with Circular Buffer

consumer:

```
while (true) {
   while (in == out)
         /* do nothing */;
   w = b[out];
   out = (out + 1) % n;
   /* consume item w */
}
```

Solution Using Semaphores

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

e tracks # empty spaces

n tacks # used spaces

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

Limitations of Semaphores

- No abstraction and modularity
  - a process that uses a semaphore has to know which other processes use the semaphore, and how these processes use the semaphore
  - a process cannot be written in isolation
- Consider sequencing between three processes
  - P1, P2, P3, P1, P2, P3, …

P1	P2	P3
Wait(sem1);	**Wait(sem2);**	**Wait(sem3);**
// do stuff	**// do stuff**	**// do stuff**
Signal(sem2);	**Signal(sem3);**	**Signal(sem1);**

- What happens if there are only two processes?
- What happens if you want to use this solution for four processes?

Limitations of Semaphores

- Very easy to write incorrect code
  - changing the order of P (wait) and V (signal)
    - can violate mutual exclusion requirements

      Signal(mutex); CODE; Wait(mutex); instead of

      Wait(mutex); CODE; Signal(mutex);
    - can cause deadlock

      Wait(seq); instead of

      Signal(seq);
  - similar problems with omission
- Extremely difficult to verify programs for correctness
  - Need for still higher-level synchronization abstractions!

Monitors

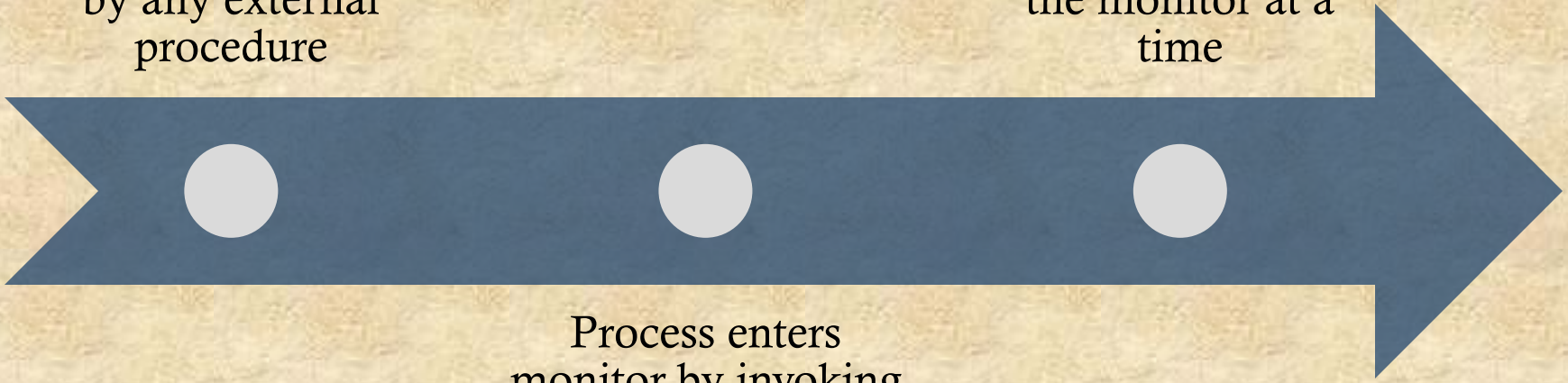
- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure

Process enters monitor by invoking one of its procedures

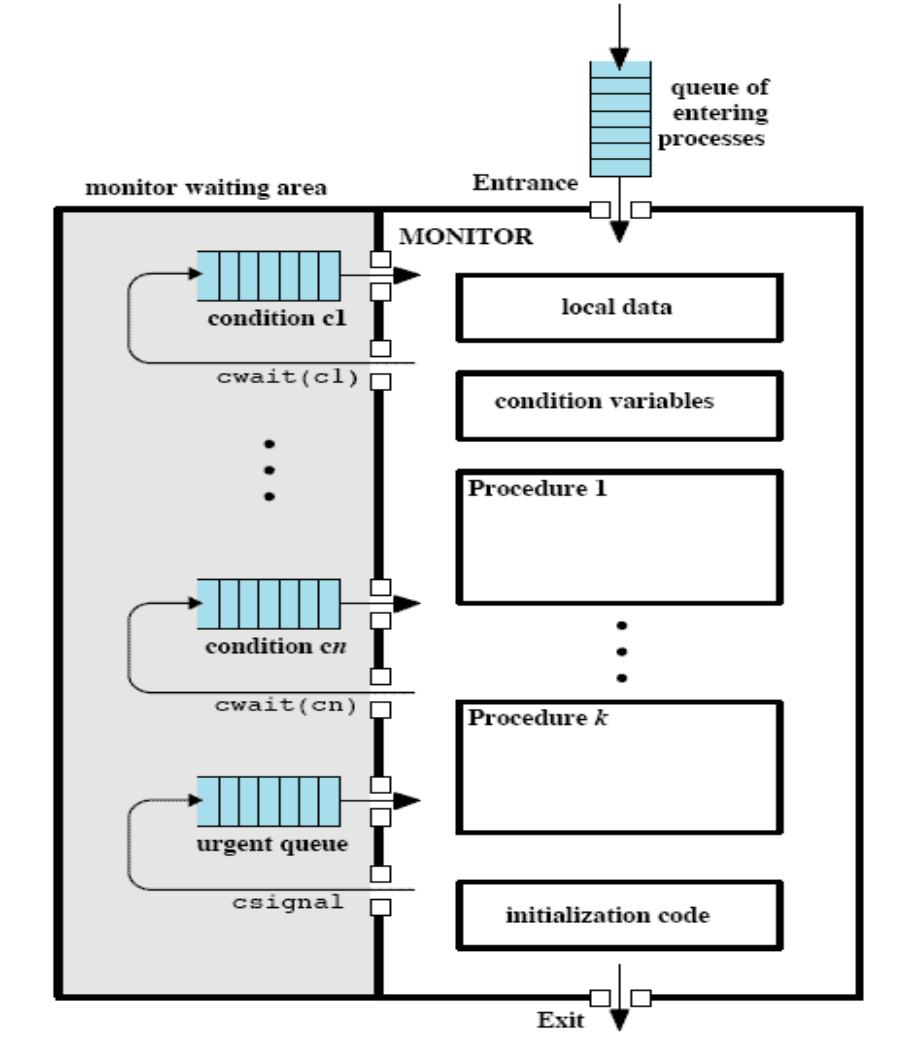
Only one process may be executing in the monitor at a time



Synchronization

- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are operated on by two functions:
    - cwait(c): suspend execution of the calling process on condition c
    - csignal(c): resume execution of some process blocked after a cwait on the same condition

Structure of a Monitor



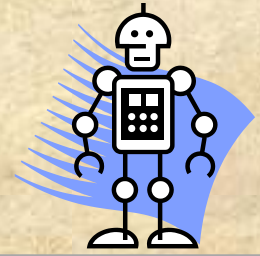


Figure 5.15 Structure of a Monitor

Waiting in the Monitor

- Note that the semantics of executing a *wait* in the monitor is that several processes can be waiting "inside" the monitor at any given time but only one is executing
  - wait queues are internal to the monitor
  - there can be multiple wait queues

Waiting in the Monitor

- Who executes after a signal operation? (say P signals Q)
  - Mesa semantics: signaler P continues
    - Q is enabled but gets its turn only after P either leaves or executes a *wait*
  - Hoare semantics: signalee Q continues and P waits/exits
    - logically natural since the condition that enabled Q might no longer be true when Q eventually executes
      - P needs to wait for Q to exit the monitor
  - require that the *signal* be the last statement in the procedure
    - advocated by Brinch Hansen (Concurrent Pascal)
    - easy to implement but less powerful than the other two

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                        /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     csignal(notfull);                         /* resume any waiting producer */
}
{                                                          /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Problem Solution Using a Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                        /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     csignal(notfull);                         /* resume any waiting producer */
}
{                                                          /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Bounded Buffer Monitor

```
void append (char x)
{
     while(count == N) cwait(notfull);    /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;                                  /* one more item in buffer */
     cnotify(notempty);                      /* notify any waiting consumer */
}

void take (char x)
{
     while(count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     cnotify(notfull);                       /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor

Condition Variables

- **Synchronization**
  - ✓ processes can also **cooperate** by means of simple signals, without defining a "critical region"
  - ✓ like mutexes: instead of looping, a process can block in some place until it receives a specific **signal** from the other process

Condition Variables work similarly to the condition variables of a Monitor, but there is no monitor, just the condition variables

Message Passing

- When processes interact with one another two fundamental requirements must be satisfied:

synchronization	communication
• to enforce mutual exclusion	• to exchange information

- Message Passing is one approach to providing both of these functions
  - works with distributed systems *and* shared memory multiprocessor and uniprocessor systems

Message Passing

- The actual function is normally provided in the form of a pair of primitives:

  send (destination, message)

  receive (source, message)

- A process sends information in the form of a *message* to another process designated by a *destination*
- A process receives information by executing the `receive` primitive, indicating the *source* and the *message*

Message Passing

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queuing Discipline
- FIFO
- Priority

Table 5.5 Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization

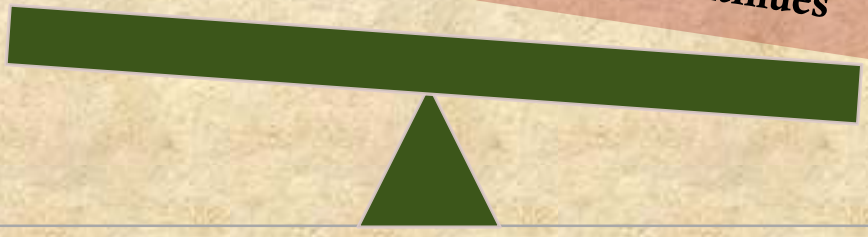
Communication of a message between two processes implies synchronization between the two

the receiver cannot receive a message until it has been sent by another process

When a receive primitive is executed in a process there are two possibilities:

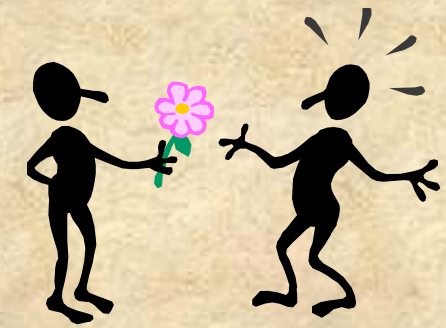
if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive

if a message has previously been sent the message is received and execution continues



Blocking Send, Blocking Receive

- Both sender and receiver are blocked until the message is delivered
- Sometimes referred to as a *rendezvous*
- Allows for tight synchronization between processes



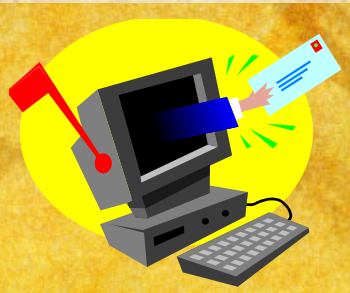
Nonblocking Send

Nonblocking send, blocking receive

- sender continues on but receiver is blocked until the requested message arrives
- most useful combination
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait

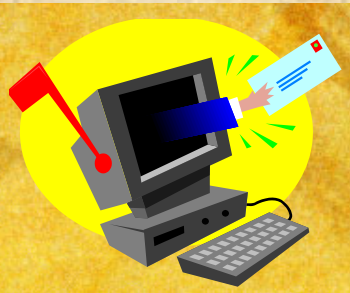


Addressing

- Schemes for specifying processes in `send` and `receive` primitives fall into two categories:

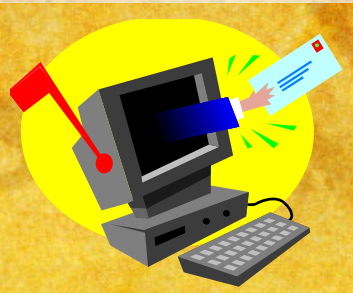
Direct addressing

Indirect addressing



Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed



Indirect Addressing

Messages are sent to a shared data structure consisting of queues that can temporarily hold messages

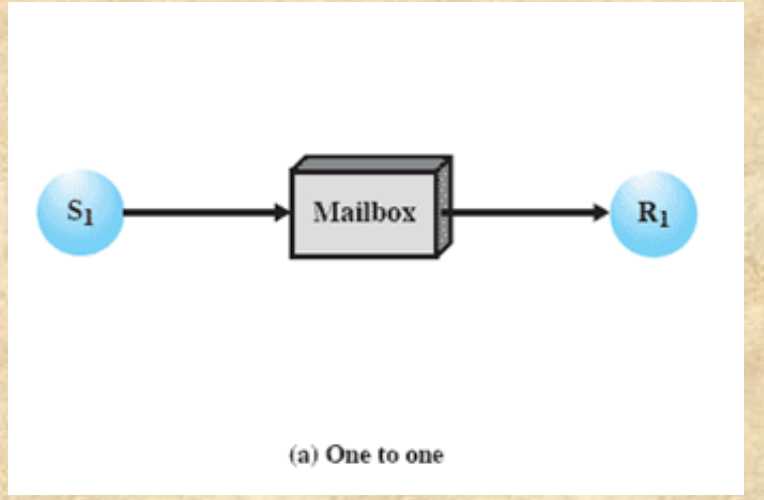
Queues are referred to as *mailboxes*

One process sends a message to the mailbox and the other process picks up the message from the mailbox

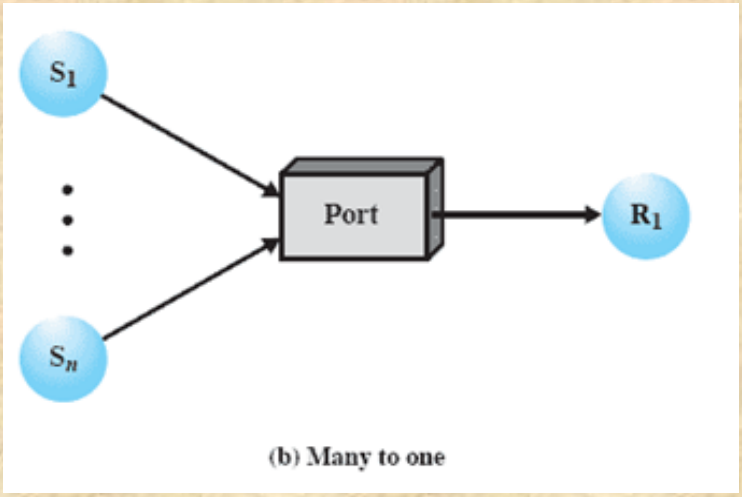
Allows for greater flexibility in the use of messages

Indirect Process

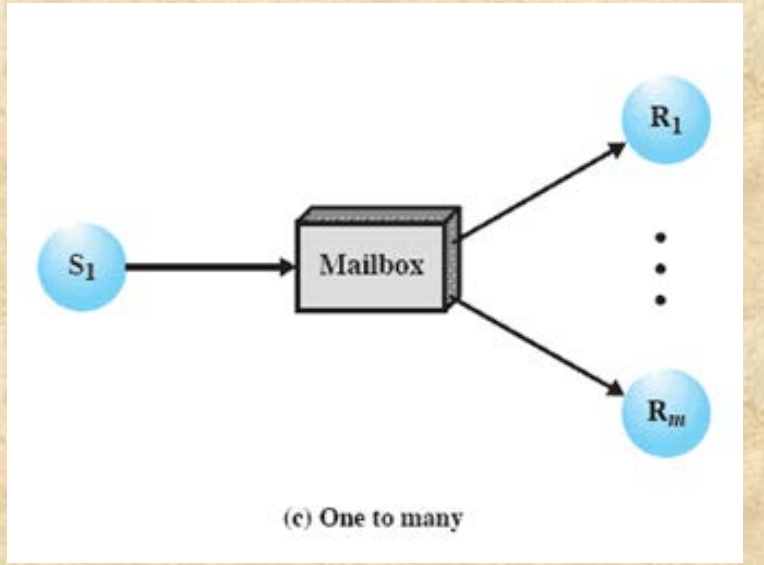
Communication



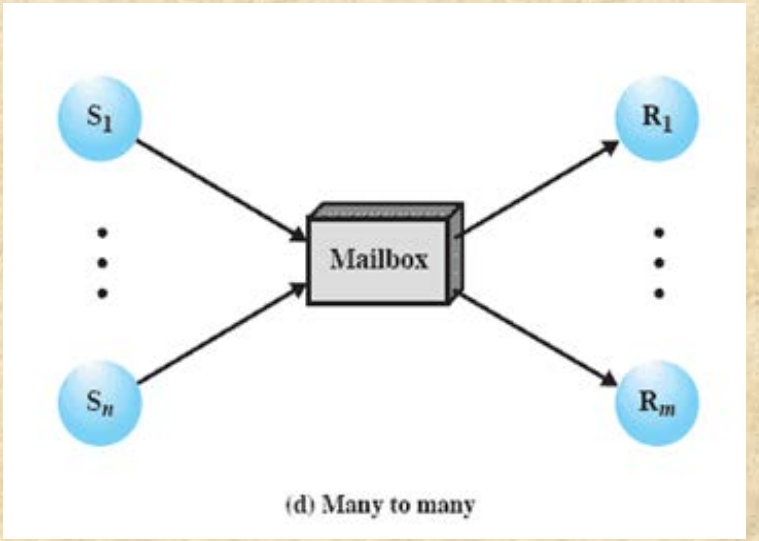
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

General Message Format

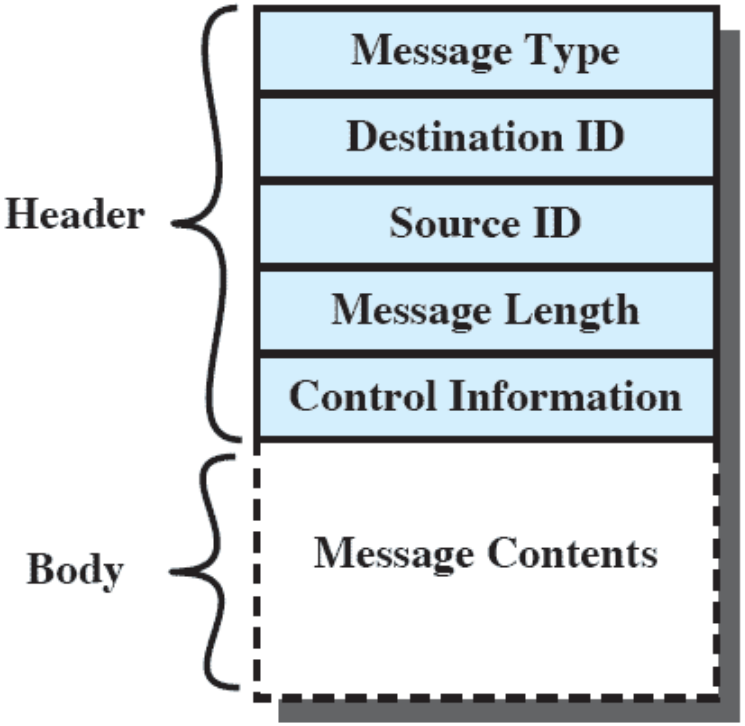


Figure 5.19 General Message Format

Mutual Exclusion

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true) {
     receive (box, msg);
     /* critical section   */;
     send (box, msg);
     /* remainder   */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

Message Passing Example

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce, null);
    parbegin (producer, consumer);
}
```

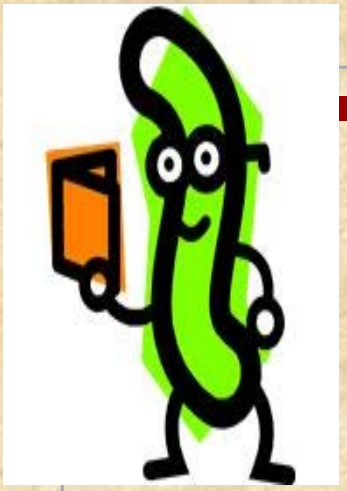
Figure 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Readers/Writers Problem

- A data area is shared among many processes
  - some processes only read the data area, (readers) and some only write to the data area (writers)
- Conditions that must be satisfied:
  1. any number of readers may simultaneously read the file
  2. only one writer at a time may write to the file
  3. if a writer is writing to the file, no reader may read it

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Readers Have Priority



Readers Have Priority

- Semaphore *wsem* enforces mutual exclusion on access to the file
- Semaphore *x* enforces mutual exclusion on accesses to the global variable *readcount*
- Once one reader has access, other readers can access the file without having to acquire the *wsem* semaphore

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority

A Problem

- Once one reader has access, other readers can access the file without having to acquire the *wsem* semaphore
- Can lead to writer starvation

Writers Have Priority

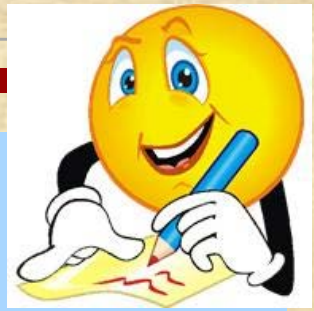
- Semaphores
  - *wsem* enforces mutual exclusion on access to the file
  - rsem inhibits all readers when there is at least one writer trying to access the file
  - *x* enforces mutual exclusion on accesses to the global variable *readcount*
  - *y* enforces mutual exclusion on accesses to the global variable *writecount*
  - *z* enforces mutual exclusion on the semaphore *rsem*
- Only one reader can be queued up on *rsem* at any time
- Ensures that writers queueing up on *rsem* will have a fair shake

Solution Writers Have Priority

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1) semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
```

Figure 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority

Solution Writers Have Priority



```
void writer ()
{
    while (true) {
     semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority

State of the Process Queues

Readers only in the system	•*wsem* set •no queues
Writers only in the system	•*wsem* and *rsem* set •writers queue on *wsem*
Both readers and writers with read first	•*wsem* set by reader •*rsem* set by writer •all writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*
Both readers and writers with write first	•*wsem* set by writer •*rsem* set by writer •writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*

Table 5.6 State of the Process Queues for Program of Figure 5.23

Message Passing

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Figure 5.24 A Solution to the Readers/Writers Problem Using Message Passing

Message Passing

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
 }
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive (writerequest, msg);
               writer_id = msg.id;
               count = count – 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

- If count > 0
  - No writer is waiting and there may or may not be readers active
  - Service all “finished” messages first to clear active readers
  - Then service write requests and then read requests
- If count = 0
  - Only one write request is outstanding
  - Allow write to proceed and wait for “finished” message
- If count < 0
  - A writer has made a request and is being made to wait to clear all active readers
  - Only service “finished” messages until all active readers are cleared
  - Then allow write to proceed

Figure 5.24 A Solution to the Readers/Writers Problem Using Message Passing

Summary

Messages

- Useful for the enforcement of mutual exclusion discipline

Operating system themes are:

- Multiprogramming, multiprocessing, distributed processing
- Fundamental to these themes is concurrency
  - issues of conflict resolution and cooperation arise

Mutual Exclusion

- Condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time
- One approach involves the use of special purpose machine instructions

Semaphores

- Used for signaling among processes and can be readily used to enforce a mutual exclusion discipline

Summary

- Principles of concurrency
  - Race condition
  - OS concerns
  - Process interaction
  - Requirements for mutual exclusion
- Mutual exclusion: hardware support
  - Interrupt disabling
  - Special machine instructions
- Semaphores
  - Mutual exclusion
  - Producer/consumer problem
  - Implementation of semaphores
- Monitors
  - Monitor with signal
  - Alternate model of monitors with notify and broadcast
- Message passing
  - Synchronization
  - Addressing
  - Message format
  - Queueing discipline
  - Mutual exclusion
- Readers/writers problem
  - Readers have priority
  - Writers have priority