Chapter 2: Process Synchronization

2.4
Process Synchronization

GV: Nguyễn Thị Thanh Vân

Outline

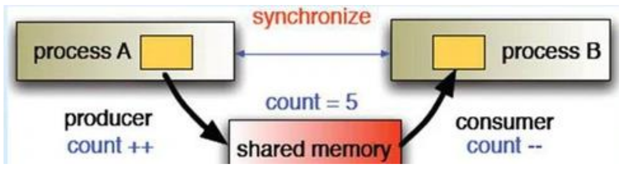
- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

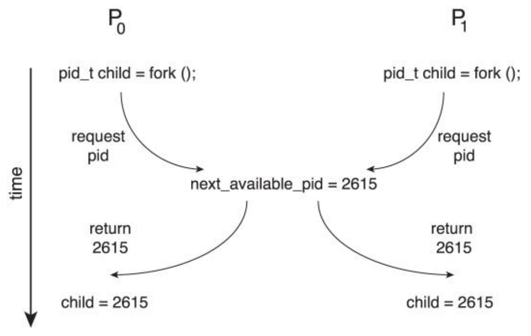
Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.



Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Critical Section Problem

- process synchronization = critical-section problem
- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process ***P**$_i$*

```
do {
     entry section
          critical section
     exit section
          remainder section
} while (true);
```

- Two general approaches are used to handle critical sections in OS:
  - preemptive kernel: allows a process to be preempted while it is running in kernel mode.
  - nonpreemptive kernel: does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks,or voluntarily yields control of the CPU.

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process ***P**$_i$* is executing in its critical section, then no other processes can be executing in their critical sections
2. **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 process that will enter the critical section next cannot be postponed indefinitely
3. **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

Classification of solutions

- Interrupt-based Solution
- Software Solution
  - Software Solution 1
  - Software Solution 2
  - Peterson’s Algorithm
- Hardware support for synchronization
  - Memory barriers
  - Special hardware instructions
    - **Test-and-Set** instruction
    - **Compare-and-Swap** instruction
- Mutex lock
- Semaphores
- Monitors
- Liveness

Interrupt-based Solution

- The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified.
- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Software Solution 1

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn;`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- Initially **`turn = 0`**
- Algorithm for Process *P_i*

```
do {
          while (turn != i);
          critical section
          turn = j;
          remainder section
} while (1);
```

Correctness of the Software Solution1

- Mutual exclusion is preserved

  P_i enters critical section if and only if:

  `turn = i`

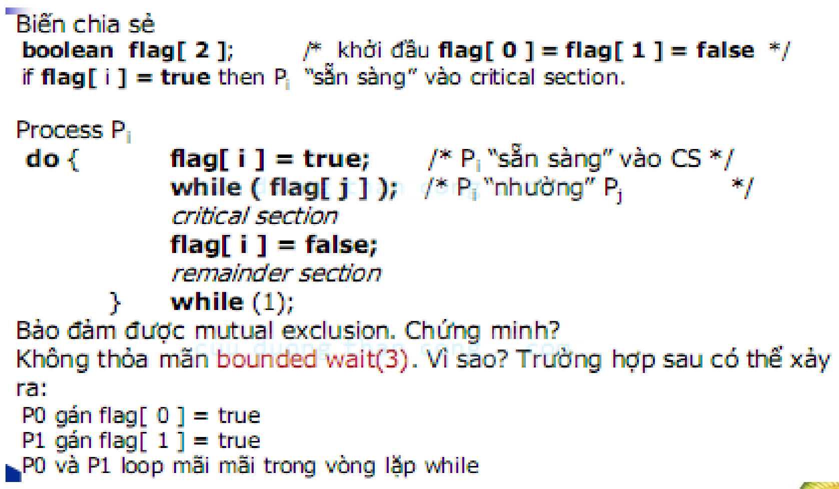
  and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
  - If Process 1 wants to enter the critical section and Process 2 is not interested in entering the critical section, can Process 1 enter?

```
Process P0:
do
   while (turn != 0);
   critical section
   turn := 1;
   remainder section
while (1);
```

```
Process P1:
do
   while (turn != 1);
   critical section
   turn := 0;
   remainder section
while (1);
```

- What about the Bounded-waiting requirement?

Software Solution 2


Biến chia sẻ

```
boolean  flag[ 2 ];        /*  khởi đầu flag[ 0 ] = flag[ 1 ] = false  */
if flag[ i ] = true then Pi  "sẵn sàng" vào critical section.
```

Process P_i

```
do {        flag[ i ] = true;       /* Pi "sẵn sàng" vào CS */
            while ( flag[ j ] );    /* Pi "nhường" Pj           */
            critical section
            flag[ i ] = false;
            remainder section
    }       while (1);
```

Bảo đảm được mutual exclusion. Chứng minh?
Không thỏa mãn bounded wait(3). Vì sao? Trường hợp sau có thể xảy ra:

P0 gán flag[0] = true
P1 gán flag[1] = true
P0 và P1 loop mãi mãi trong vòng lặp while

Software Solution -- Peterson's Algorithm

- Peterson's solution:
  - software-based solution to the critical-section problem
  - Involves designing software that addresses the requirements of
    - mutual exclusion,
    - progress,
    - and bounded waiting
  - is not guaranteed to work on modern computer architecture
- Two process solution:
  - alternate execution between their critical sections and remainder sections,
  - And share two variables:
    - `int turn;`
    - `boolean flag[2]`

Algorithm for Process P_i

- 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!

```
while (true){

        flag[i] = true; /*Pi ready */
        turn = j;    /*preemptive Pj */
        while (flag[j] && turn = = j)
               ;

           /* critical section */

        flag[i] = false;

        /* remainder section */
 }
```

Peterson's Algorithm

```
while (true){       /* 0 wants in */
       flag[0] = true;
       turn = 1;     /* 0 gives a change to 1 */
       while (flag[1] && turn = = 1)
              ;
          /* critical section */
        /*  0 no longer wants in  */
       flag[0] = false;
       /* remainder section */
 }
```

```
while (true){       /* 1 wants in */
       flag[1] = true;
       turn = 0;     /* 1 gives a change to 0 */
       while (flag[0] && turn = = 0)
              ;
          /* critical section */
        /*  1 no longer wants in  */
       flag[1] = false;
       /* remainder section */
 }
```

Correctness of Peterson's Solution

- Three CS requirement are met:
  1. Mutual exclusion is preserved

     P$_i$ enters CS only if: either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

1. Prove: Pi and Pj are same time if: flag[i] = flag[j] == true and turn == i. => cannot occur

2,3. Prove: Note that a process can be prevented from entering the CS only if it is stuck in the while loop with the condition **`flag[i] = true and turn = j`**, this loop is the only one possible.

- If Pj is not ready to CS, then flag[j] == false, and Pi can enter its CS.
- If Pj has set flag[j] = true and is also executing in its while statement, then either turn == i or j.
  - If turn == i,then Pi will enter the CS.
  - If turn == j, then Pj will enter the CS. (Pi continue)
- However, once Pj exits its CS, it will reset flag[j] =false, allowing Pi to enter its CS.
- If Pj resets flag[j] to true, it must also set turn to i.
- Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the CS **(progress)** after at most one entry by Pj **(bounded waiting**)

Peterson's Solution

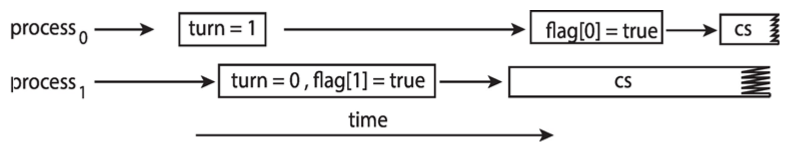
- Solution works for 2 process.
- What about modifying it to handle 10 processes?
- Solution requires **busy waiting**
  - Processes waste CPU cycles to ask if they can enter the critical section
- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is OK as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:
```
boolean flag = false;
int x = 0;
```
- Thread 1 performs
```
while (!flag)
  ;
print x
```
- Thread 2 performs
```
x = 100;
flag = true
```
- What is the expected output (Thread1)? `100`
- However, since the variables **`flag`** and **`x`** are independent of each other, the instructions: (reorder the instructions for Thread 2)
```
flag = true;
x = 100;
```
- If this occurs, the output may be (Thread1): `0`

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution: reordered 2 lines
```
while (true){
       flag[i] = true; /*Pi ready */
       turn = j;       /*preemptive Pj */
       while (flag[j] && turn = = j)
              ;
          /* critical section */
       flag[i] = false;
       /* remainder section */
 }
```



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**

Hardware Support for Synchronization

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
    - Is this practical?
  - Generally, too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at **three** forms of hardware support:
  - Memory Barriers
  - Hardware instructions
  - Atomic Variables

Memory Barrier Instructions

- A **memory barrier** instruction:
  - is used to ensure that all loads and stores instructions are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that
  - the store operations are completed in memory and visible to other processors before future load or store operations are performed.
- Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion

Memory Barrier Example

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
  while (!flag)
    memory_barrier();
  print x
```

- Thread 2 now performs

```
  x = 100;
  memory_barrier();
  flag = true
```

- For Thread 1 we are guaranteed that that the value of **`flag`** is loaded before the value of **`x`**.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment **`flag`**.

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

The test_and_set Instruction

- Definition

```
boolean test_and_set (boolean *target)
        {
           boolean rv = *target;
           *target = true;
           return rv:
        }
```

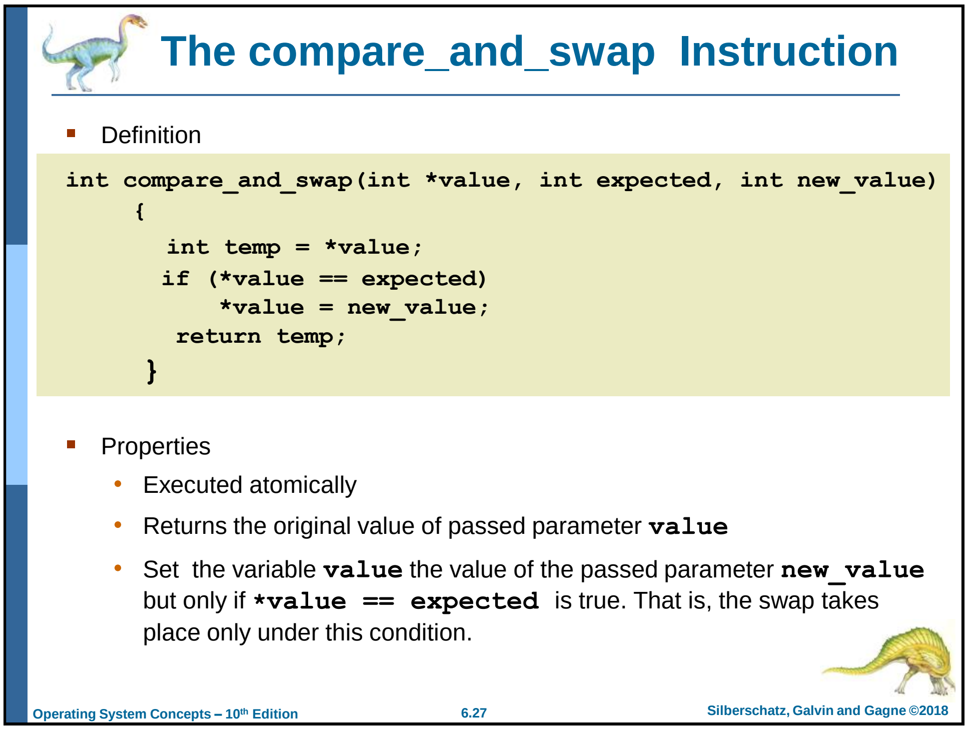
- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**

Solution Using test_and_set()

- Shared Boolean variable **`lock`**, initialized to **`false`**
- Solution:

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

           /* critical section */
       lock = false;
           /* remainder section */
   } while (true);
```

- Based on busy waiting
- Does it solve the critical-section problem?
  - Mutual exclusion
  - bounded wait



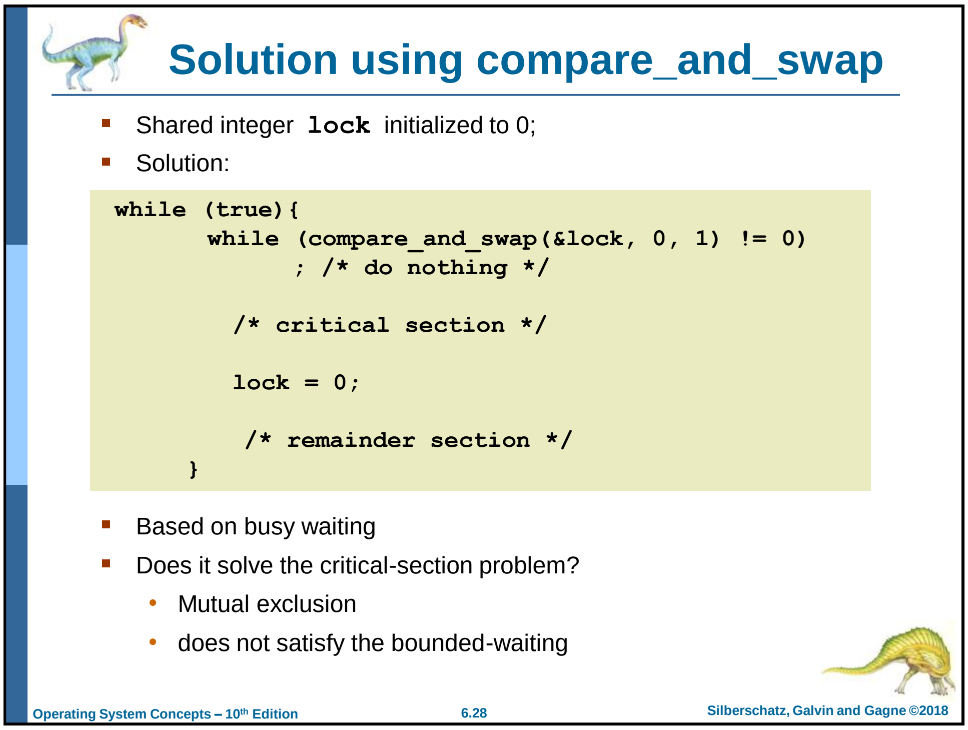
The compare_and_swap Instruction

- Definition

```
int compare_and_swap(int *value, int expected, int new_value)
     {
        int temp = *value;
       if (*value == expected)
           *value = new_value;
        return temp;
      }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Operating System Concepts – 10th Edition 6.27 Silberschatz, Galvin and Gagne ©2018



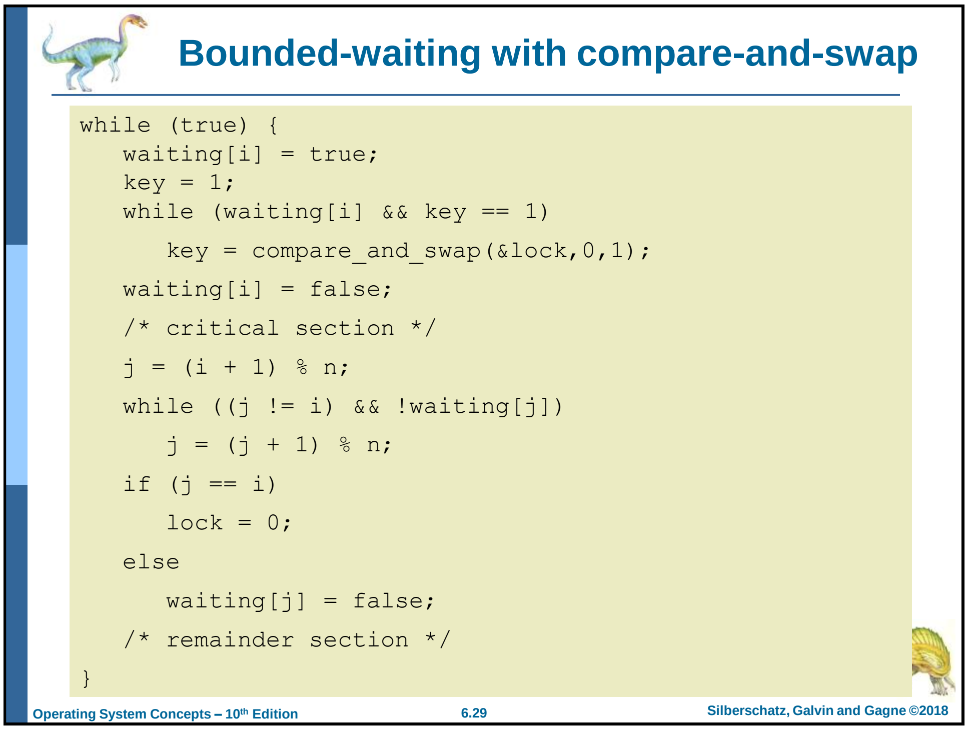
Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

```
while (true){
       while (compare_and_swap(&lock, 0, 1) != 0)
             ; /* do nothing */

         /* critical section */

         lock = 0;

          /* remainder section */
      }
```

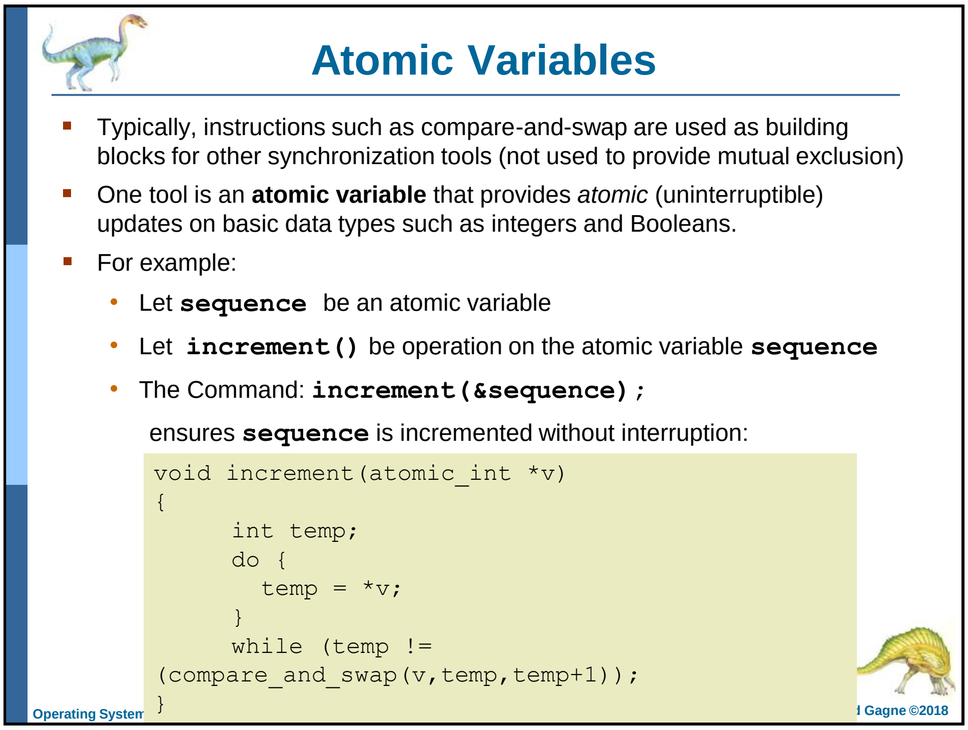
- Based on busy waiting
- Does it solve the critical-section problem?
  - Mutual exclusion
  - does not satisfy the bounded-waiting

Operating System Concepts – 10th Edition 6.28 Silberschatz, Galvin and Gagne ©2018



Bounded-waiting with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```



Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools (not used to provide mutual exclusion)
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and Booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command: **`increment(&sequence);`**

  ensures **`sequence`** is incremented without interruption:

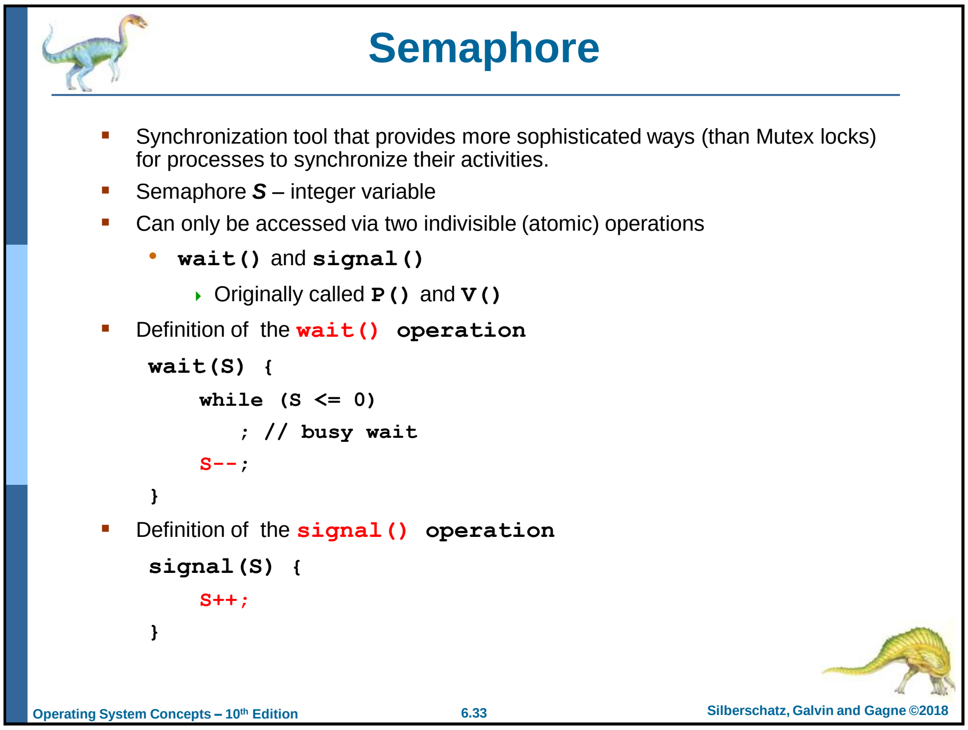
```
void increment(atomic_int *v)
{
      int temp;
      do {
         temp = *v;
      }
      while (temp !=
(compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Two operations:
  - **`acquire()`** a lock
  - **`release()`** a lock
- The **`acquire()`** and **`release()`** are executed atomically
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {

       acquire lock

          critical section

       release lock

remainder section
}
```



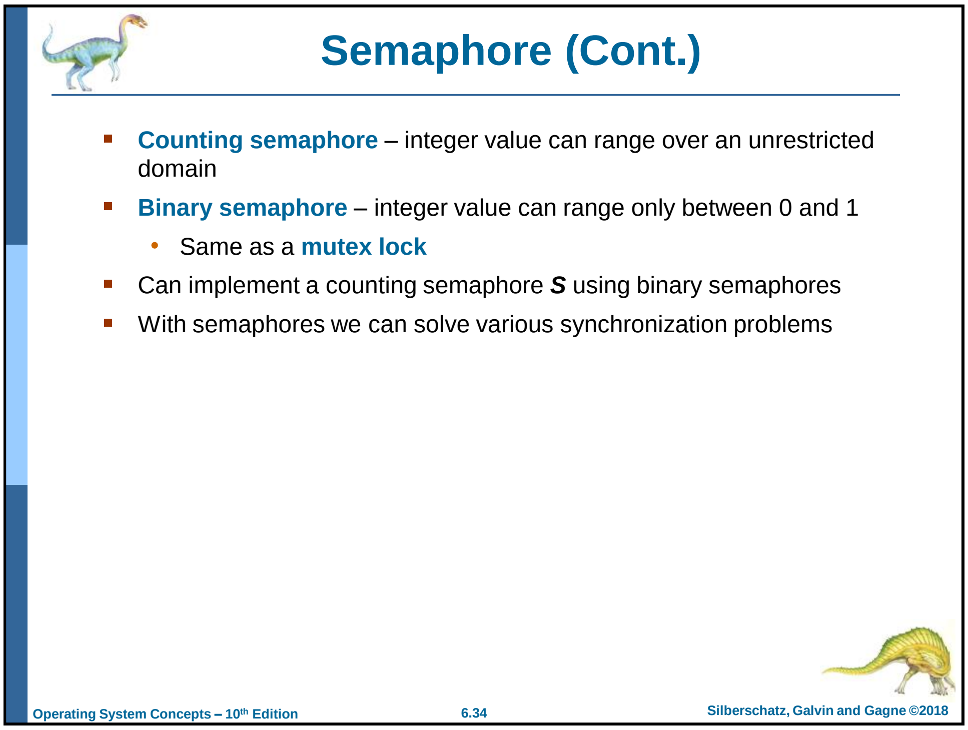
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()`** **`operation`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal()`** **`operation`**

```
signal(S) {
    S++;
}
```



Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** using binary semaphores
- With semaphores we can solve various synchronization problems

Semaphore Usage Example (Cont.)

- Solution to the CS Problem
  - Create a semaphore “**mutex**” initialized to 1
  - Code:

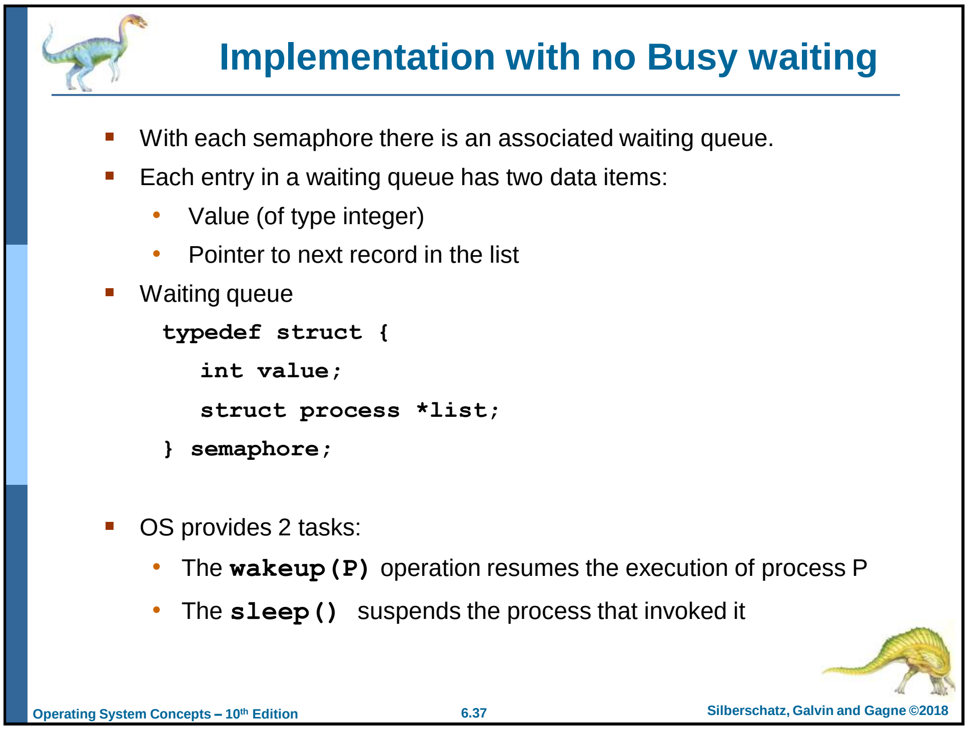
```
wait(mutex);
   CS
signal(mutex);
```

- Consider ***P_1*** and ***P_2*** that with two statements ***S_1*** and ***S_2*** and the requirement that ***S_1*** to happen before ***S_2***
  - Create a semaphore “**synch**” initialized to 0
  - Code: to synchronization

```
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Can be implemented using any of the critical sections solutions - 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 for “regular” applications, where the application may spend lots of time in critical sections this is not a good solution

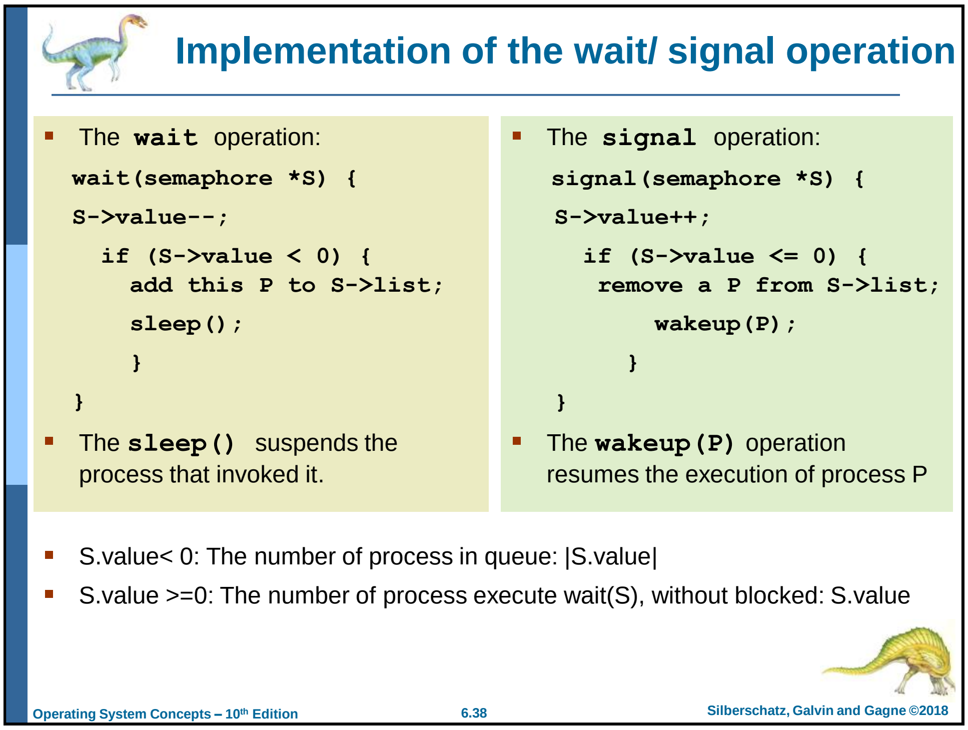


Implementation with no Busy waiting

- 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
- Waiting queue

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

- OS provides 2 tasks:
  - The **`wakeup(P)`** operation resumes the execution of process P
  - The **`sleep()`** suspends the process that invoked it



Implementation of the wait/ signal operation

- The **`wait`** operation:

```
wait(semaphore *S) {
S->value--;
   if (S->value < 0) {
      add this P to S->list;
      sleep();
      }
}
```

- The **`sleep()`** suspends the process that invoked it.

- The **`signal`** operation:

```
signal(semaphore *S) {
S->value++;
   if (S->value <= 0) {
    remove a P from S->list;
        wakeup(P);
      }
  }
```

- The **`wakeup(P)`** operation resumes the execution of process P

- S.value< 0: The number of process in queue: |S.value|
- S.value >=0: The number of process execute wait(S), without blocked: S.value

Problems with Semaphores

- Incorrect use of semaphore operations:

```
signal(mutex)
...
critical section
...
wait(mutex)
```

```
wait(mutex)
...
critical section
...
wait(mutex)
```

```
signal(mutex)
...
critical section
...
signal(mutex)
```

  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**

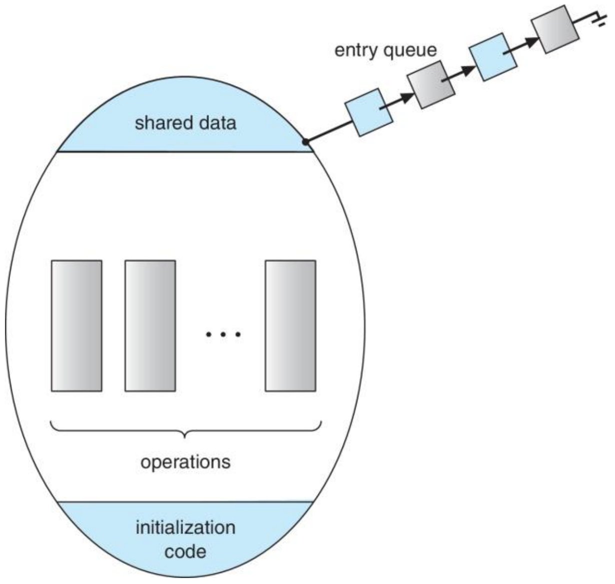
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.
- Solution: introduce high-level programming constructs

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

   initialization code (…) { … }
}
```

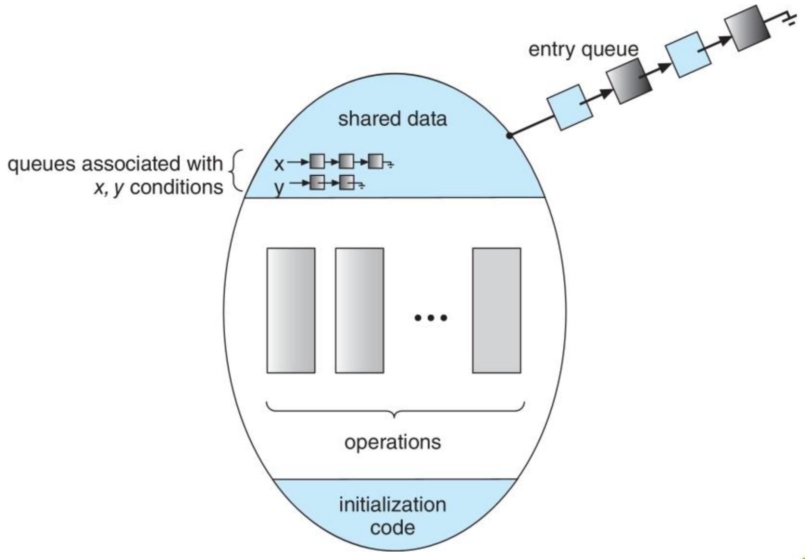
Schematic view of a Monitor



Condition Variables

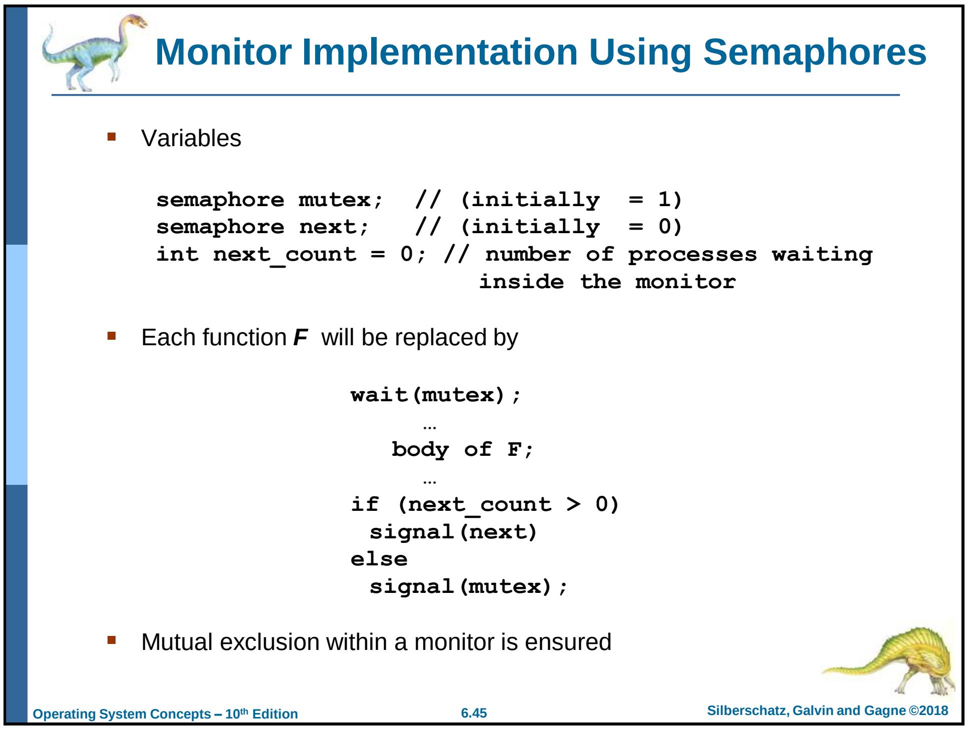
- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java



Monitor Implementation Using Semaphores

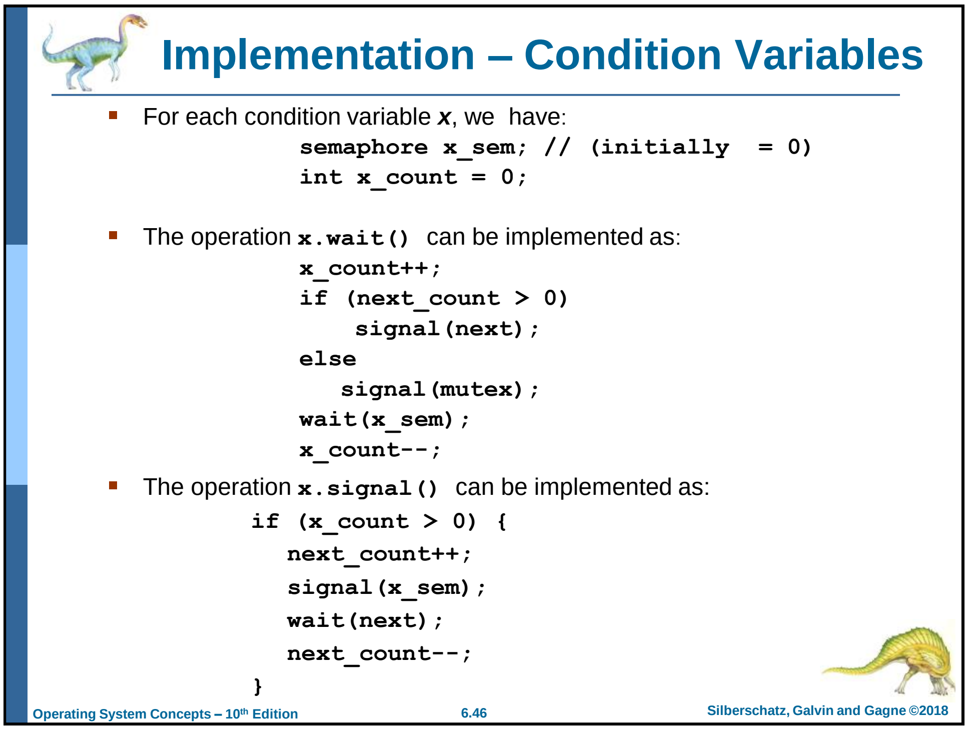
- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting
                       inside the monitor
```

- Each function ***F*** will be replaced by

```
wait(mutex);
     …
   body of F;
     …
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

- Mutual exclusion within a monitor is ensured



Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation `x.wait()` can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

- The operation `x.signal()` can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes are queued on condition variable **`x`**, and **`x.signal()`** is executed, which process should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form **`x.wait(c)`**
  - Where **`c`** is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Single Resource allocation Example

- Allocate a single resource among competing processes using priority numbers that specify the maximum amount of time a process plans to use the resource

```
R.acquire(t);
     ...
  access the resurce;
     ...

R.release;
```

- Where R is an instance of type **`ResourceAllocator`** (shown in the next slides)

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = true;
   }
   void release() {
            busy = FALSE;
            x.signal();
   }
    initialization code() {
    busy = false;
   }
}
```

- Usage:

```
acquire
 ...
release
```

- Incorrect use of monitor operations
  - **release()** … **acquire()**
  - **acquire()** … **acquire())**
  - Omitting of **acquire()** and/or **release()**

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

Deadlock

- **Deadlock** – two 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

```
        P0                  P1
     wait(S);            wait(Q);
     wait(Q);            wait(S);
      ...                  ...
     signal(S);          signal(Q);
     signal(Q);          signal(S);
```

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q) – it is blocked, it must wait until P_1 executes signal(Q)
- However, P_1 is waiting – wait(S) is blocked, until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.

Other Forms of Deadlock

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- Solved via **priority-inheritance protocol**

Chapter 2: Process Synchronization

Synchronization Examples

GV: Nguyễn Thị Thanh Vân

Outline

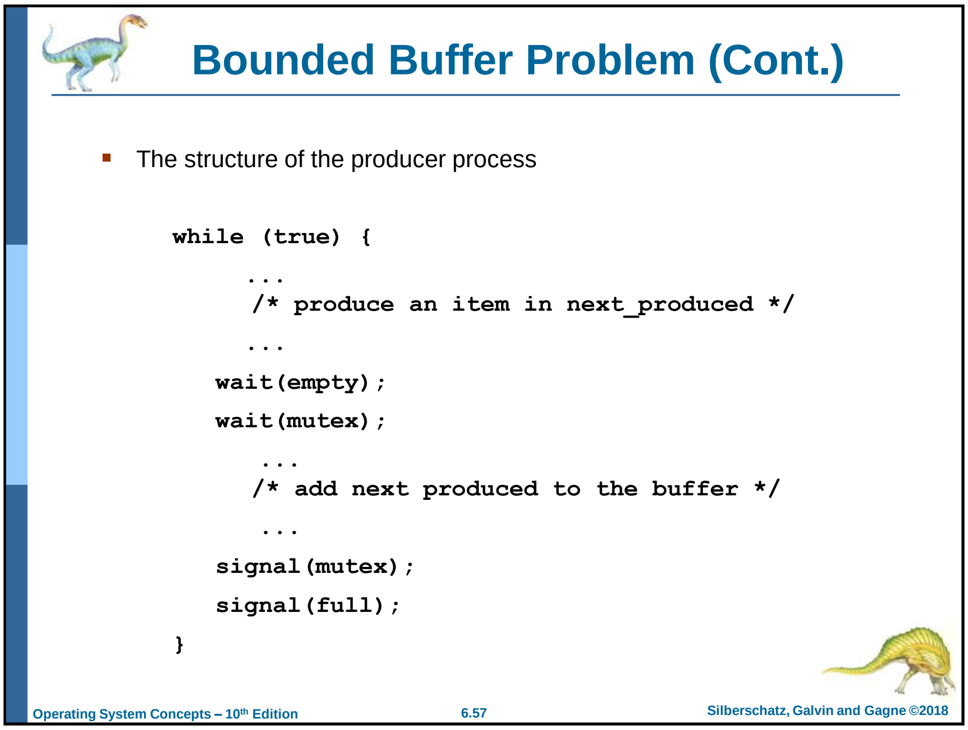
- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain and dining-philosophers synchronization problems

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

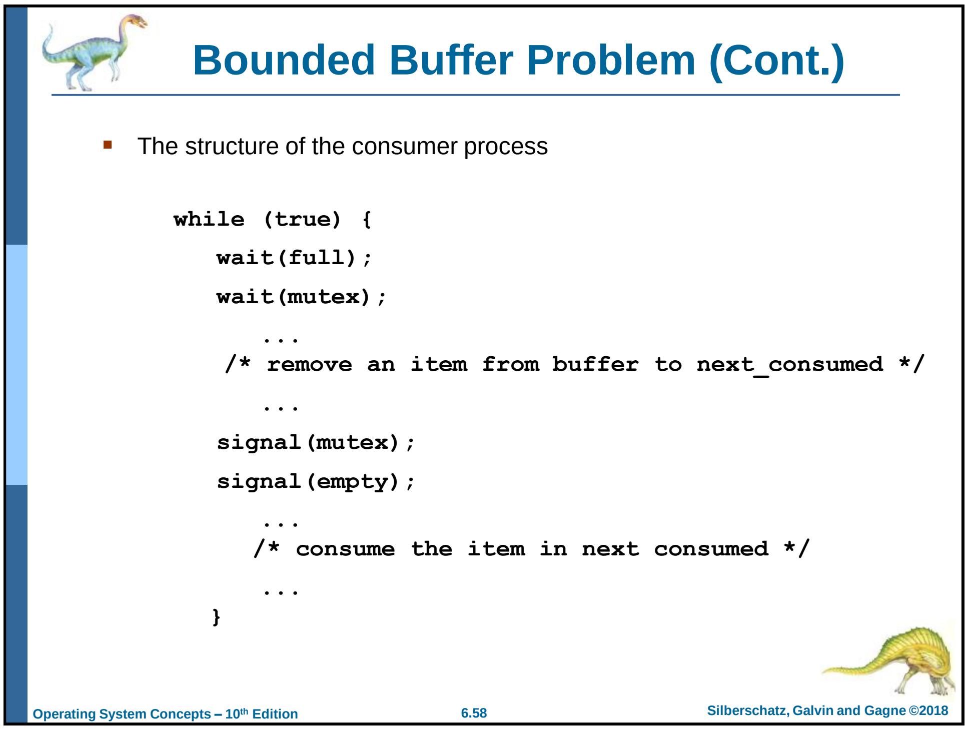
- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
       ...
        /* produce an item in next_produced */
       ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
}
```



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

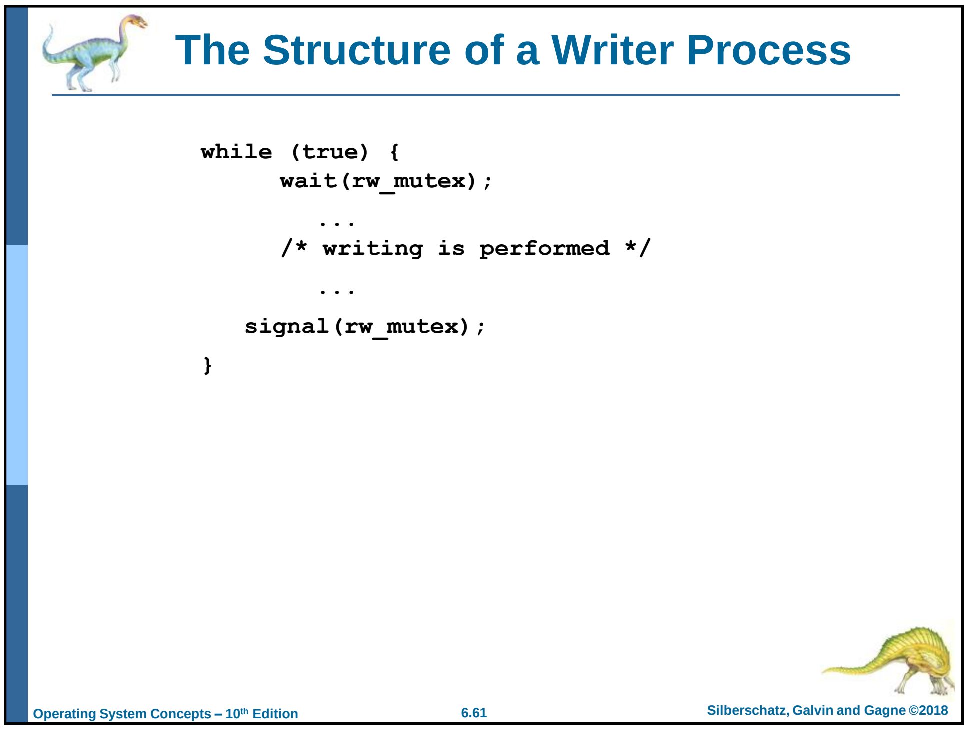
```
while (true) {
    wait(full);
    wait(mutex);
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
        /* consume the item in next consumed */
        ...
   }
```

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
- Several variations of how readers and writers are considered – all involve some form of priorities

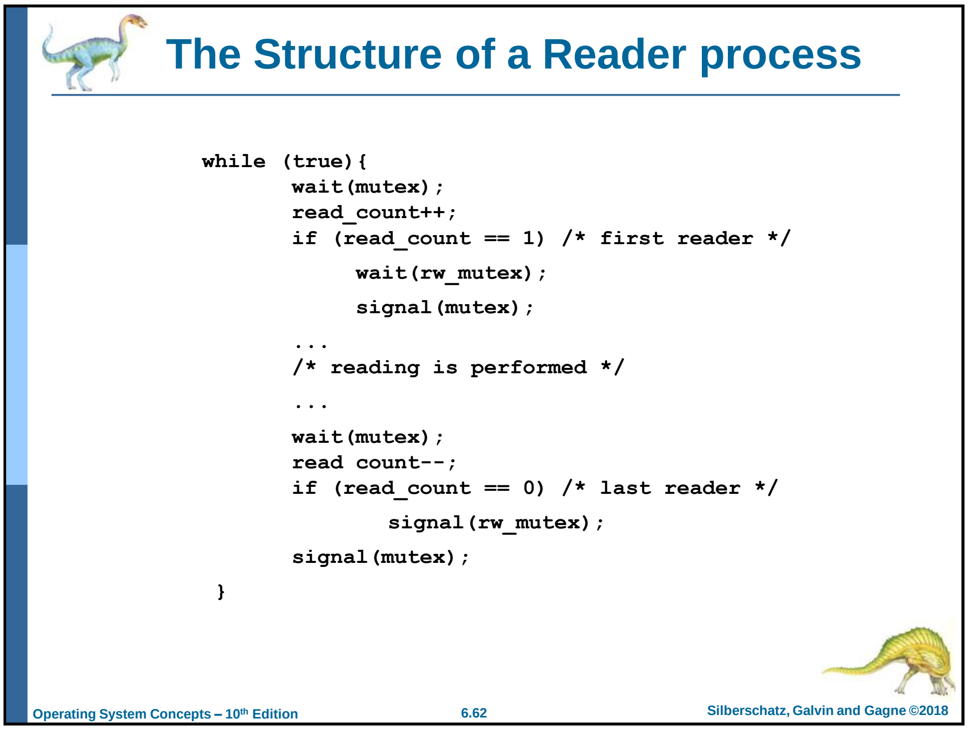
Readers-Writers: Shared Data

- Data set
- Semaphore **`rw_mutex`** initialized to 1
- Semaphore **`mutex`** initialized to 1
- Integer **`read_count`** initialized to 0



The Structure of a Writer Process

```
while (true) {
      wait(rw_mutex);
         ...
      /* writing is performed */
         ...
   signal(rw_mutex);
}
```



The Structure of a Reader process

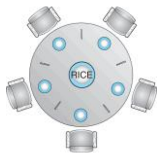
```
while (true){
       wait(mutex);
       read_count++;
       if (read_count == 1) /* first reader */
            wait(rw_mutex);
            signal(mutex);
       ...
       /* reading is performed */
       ...
       wait(mutex);
       read count--;
       if (read_count == 0) /* last reader */
               signal(rw_mutex);
       signal(mutex);
 }
```

Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
- Both the first and second may result in starvation, leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- N philosophers’ sit at a round table with a bowel of rice in the middle.



- They spend their lives alternating between thinking and eating.
- They do not interact with their neighbors.
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both chopsticks to eat, then release both when done
- In the case of 5 philosophers, the shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Semaphore Solution to Dining-Philosophers

- The structure of Philosopher *i*:

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

      /* eat for awhile */

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

      /* think for awhile */

}
```

- What is the problem with this algorithm?

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
                    // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }
```

Solution to Dining Philosophers (Cont.)

```
      void test (int i) {
               if ((state[(i + 4) % 5] != EATING) &&
               (state[i] == HUNGRY) &&
               (state[(i + 1) % 5] != EATING) ) {
                    state[i] = EATING ;
                self[i].signal () ;
               }
       }

           initialization_code() {
              for (int i = 0; i < 5; i++)
              state[i] = THINKING;
            }
    }
```

Solution to Dining Philosophers (Cont.)

- Each philosopher "i" invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

     /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

End of Chapter 2
2.4