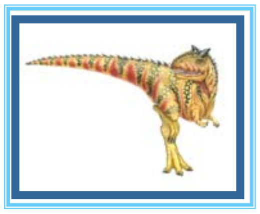
Chapter 6: Process Synchronization



(slides improved by R. Doemer, 04/22/10 – 04/30/10)

Chapter 6: Process Synchronization

- Background
- The Producer-Consumer Problem
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
  - Pthread Synchronization
- Atomic Transactions

(slide modified by R. Doemer, 04/29/10)

Objectives

- To introduce the critical-section problem,
  whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions
  of the critical-section problem
- To introduce the concept of an atomic transaction
  and describe mechanisms to ensure atomicity

(slide modified by R. Doemer, 04/22/10)

Background

- Concurrent execution of processes or threads
  creates situations of **non-determinism**!
  - CPU scheduling by operating system often (!)
    yields *non-deterministic* order of execution
    of concurrent program instructions
  - e.g. thread may be preempted at any time (!)
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms
  to ensure the *orderly* execution of cooperating processes

(slide modified by R. Doemer, 04/22/10)

Producer-Consumer Problem

- Paradigm for cooperating processes
  - **Producer** process produces information that is consumed by a **consumer** process
- Buffered communication
  - *Unbounded-buffer* places no practical limit on the size of the buffer
  - *Bounded-buffer* assumes that there is a fixed buffer size
- EECS111 Notes:
  - We will discuss the postponed *bounded buffer* implementation from chapter 3 here as Version 1
  - We will improve this implementation then (in chapter 6) using better efficiency and synchronization as Version 2 and 3

(slide inserted from chapter 3 and modified by R. Doemer, 04/27/10)

Producer-Consumer Problem

- Bounded buffer implementation *(Version 1)*
  - Data in shared memory

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

item buffer[BUFFER_SIZE]; /* circular buffer */
int in = 0;        /* index of next free position */
int out = 0;       /* index of first full position */
```

(slide inserted from chapter 3 and modified by R. Doemer, 04/23/10)

Producer-Consumer Problem

- Producer implementation *(Version 1)*
  - Produce an item, wait for buffer space, store in buffer

```
item nextProduced;

while (true) {
      /* produce an item and put in nextProduced */
      while (((in + 1) % BUFFER_SIZE) == out)
            ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
}
```

(slide modified by R. Doemer, 04/23/10)

Producer-Consumer Problem

- Consumer implementation *(Version 1)*
  - Wait for an item available, load it from buffer, consume it

```
item nextConsumed;

while (true) {
      while (in == out)
            ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      /* consume the item in nextConsumed */
}
```

(slide modified by R. Doemer, 04/23/10)

Producer-Consumer Problem

- Discussion on Implementation of Version 1
  - Data in shared memory
    - **`buffer[], in, out`**
  - Busy waiting in both producer and consumer
    - Empty loops
  - Is this a valid / safe implementation?
    - Variable **`in`** only modified by producer
    - Variable **`out`** only modified by consumer
    - Writing an integer to memory should be atomic
  - Is this an efficient implementation?
    - (a) Space: At most **`BUFFER_SIZE-1`** items can be stored
    - (b) Time: We'll investigate that in Assignment 3…

(slide modified by R. Doemer, 04/23/10)

Producer-Consumer Problem

- Bounded buffer implementation *(Version 2)*
  - Data in shared memory

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

item buffer[BUFFER_SIZE]; /* circular buffer */
int in = 0;        /* index of next free position */
int out = 0;       /* index of first full position */
int counter = 0;   /* number of items in buffer */
```

(slide modified by R. Doemer, 04/23/10)

Producer-Consumer Problem

- Producer implementation *(Version 2)*
  - Produce an item, wait for buffer space, store in buffer

```
item nextProduced;

while (true) {
      /* produce an item and put in nextProduced */
      while (counter == BUFFER_SIZE)
            ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

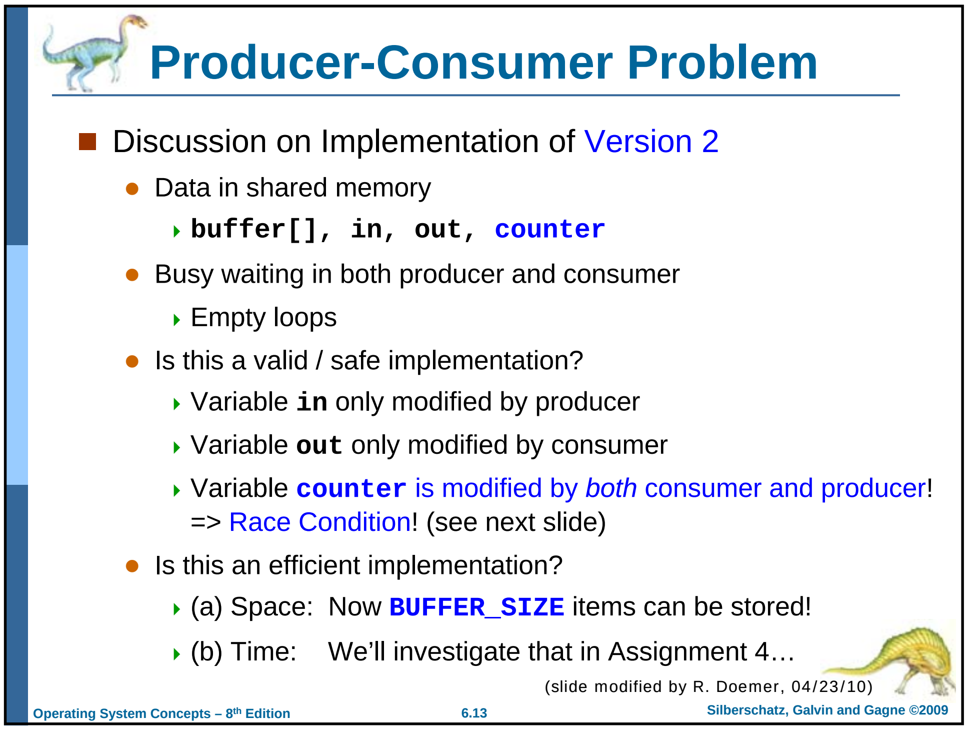
(slide modified by R. Doemer, 04/23/10)

Producer-Consumer Problem

- Consumer implementation *(Version 2)*
  - Wait for an item available, load it from buffer, consume it

```
item nextConsumed;

while (true) {
      while (counter == 0)
            ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* consume the item in nextConsumed */
}
```

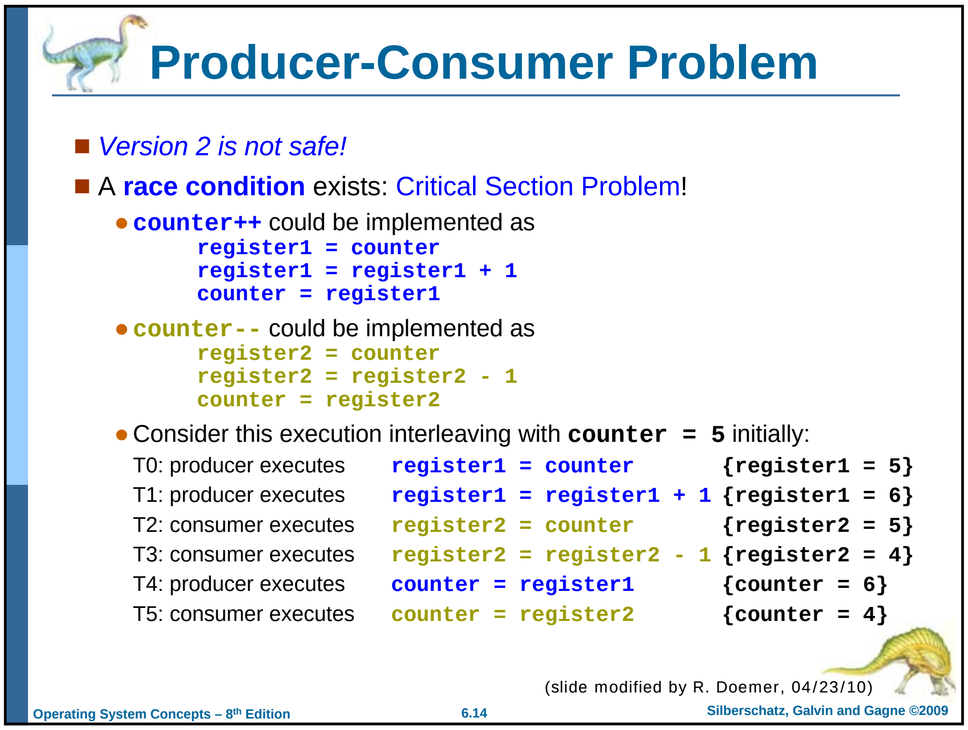
(slide modified by R. Doemer, 04/23/10)



Producer-Consumer Problem

- Discussion on Implementation of Version 2
  - Data in shared memory
    - **`buffer[], in, out, counter`**
  - Busy waiting in both producer and consumer
    - Empty loops
  - Is this a valid / safe implementation?
    - Variable **`in`** only modified by producer
    - Variable **`out`** only modified by consumer
    - Variable **`counter`** is modified by *both* consumer and producer!
      => Race Condition! (see next slide)
  - Is this an efficient implementation?
    - (a) Space: Now **`BUFFER_SIZE`** items can be stored!
    - (b) Time: We'll investigate that in Assignment 4…

(slide modified by R. Doemer, 04/23/10)



Producer-Consumer Problem

- *Version 2 is not safe!*
- A **race condition** exists: Critical Section Problem!
  - **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

  - **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

  - Consider this execution interleaving with **`counter = 5`** initially:

T0: producer executes	`register1 = counter`	`{register1 = 5}`
T1: producer executes	`register1 = register1 + 1`	`{register1 = 6}`
T2: consumer executes	`register2 = counter`	`{register2 = 5}`
T3: consumer executes	`register2 = register2 - 1`	`{register2 = 4}`
T4: producer executes	`counter = register1`	`{counter = 6}`
T5: consumer executes	`counter = register2`	`{counter = 4}`

(slide modified by R. Doemer, 04/23/10)

Critical Section Problem

- Critical section
  - Segment of code where multiple processes manipulate shared data
- Mutual exclusion
  - While one process is executing in its critical section,
    no other process is to be allowed to execute in its critical section
  - Processes must ask for permission to enter critical section
- Structure of a critical section for a typical process

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

(slide added by R. Doemer, 04/27/10)

Solution to Critical-Section Problem

Three requirements:

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

(slide modified by R. Doemer, 04/27/10)

Early Example: Peterson's Solution

- Two process *software-based* solution
  - No guarantee to work on modern processors (out-of-order execution!)
- Assume that the LOAD and STORE instructions are atomic; that is, these cannot be interrupted.
- The two processes P_i and P_j share two variables:
  - int turn;
  - boolean flag[2];
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section: flag[i] = true implies that process P_i is ready!

(slide modified by R. Doemer, 04/27/10)

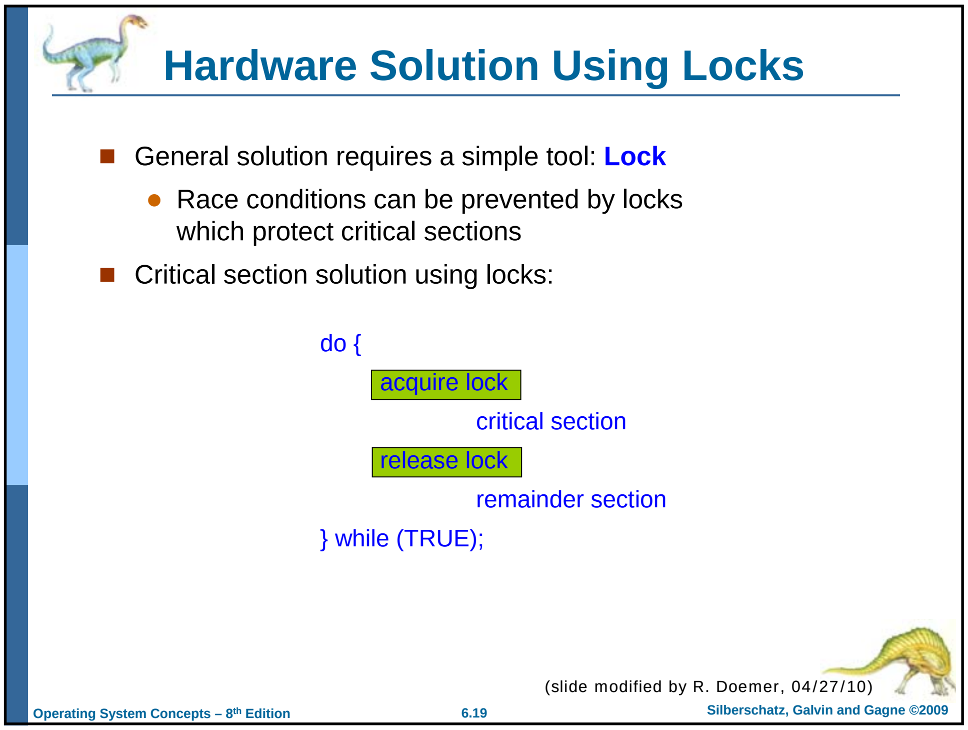
Early Example: Peterson's Solution

- Algorithm for Process P_i for

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = FALSE;
                remainder section
} while (TRUE);
```

- Mutual exclusion is preserved
- Progress requirement is satisfied
- Bounded waiting requirement is met

(slide modified by R. Doemer, 04/27/10)

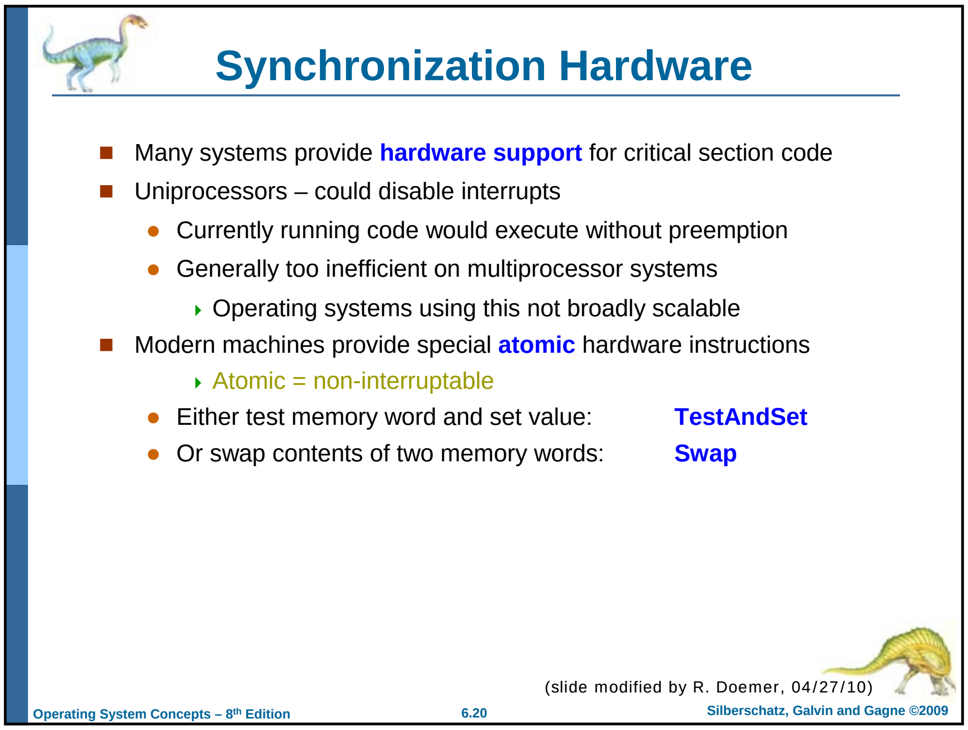


Hardware Solution Using Locks

- General solution requires a simple tool: **Lock**
  - Race conditions can be prevented by locks which protect critical sections
- Critical section solution using locks:

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

(slide modified by R. Doemer, 04/27/10)



Synchronization Hardware

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

(slide modified by R. Doemer, 04/27/10)

TestAndSet Instruction

- Definition:

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

(slide fixed by R. Doemer, 01/07/09)

Critical Section Solution using TestAndSet

- Shared boolean variable lock indicates whether or not someone is in the critical section
- Solution:

```
boolean lock = FALSE;
do {
        while ( TestAndSet (&lock) )
                ;   // do nothing

        //    critical section

        lock = FALSE;

        //      remainder section

} while (TRUE);
```

(slide modified by R. Doemer, 04/27/10)

Swap Instruction

- Definition:

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

(slide modified by R. Doemer, 04/28/10)

Critical Section Solution using Swap

- Shared Boolean variable lock indicates whether or not someone is in the critical section
- Each process has a local Boolean variable key
- Solution:

```
 boolean lock = FALSE;
 do {
        key = TRUE;
        while (key == TRUE)
                Swap (&lock, &key );

        //    critical section

        lock = FALSE;

        //    remainder section

 } while (TRUE);
```

(slide modified by R. Doemer, 04/28/10)

Critical Section Solutions with Bounded Waiting

- The previous two solutions (with **TestAndSet** and **Swap**)
  - satisfy the mutual-exclusion and progress requirements
  - do *not* satisfy the bounded-waiting requirement
- Bounded-waiting Mutual Exclusion with **TestandSet**():

```
do {    waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
        // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```

(slide modified by R. Doemer, 04/28/10)

Semaphores

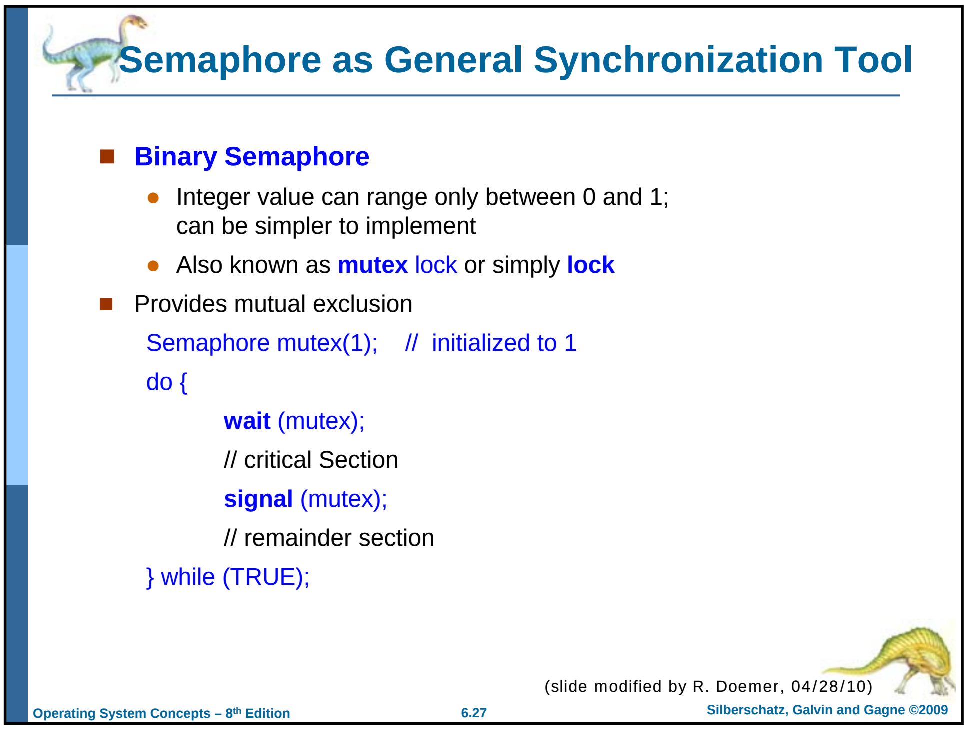
- General synchronization tool that does not require busy waiting
- **Semaphore**
  - Integer variable S
  - Two *atomic* operations: **wait**() and **signal**()
  - Originally called P() and V()
  - Less complicated than previous schemes
- **Definition** of a Semaphore S (using busy waiting aka. **spinlock**):
  - **wait** (S) {

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

  - **signal** (S) {

```
        S++;
    }
```

(slide modified by R. Doemer, 04/28/10)

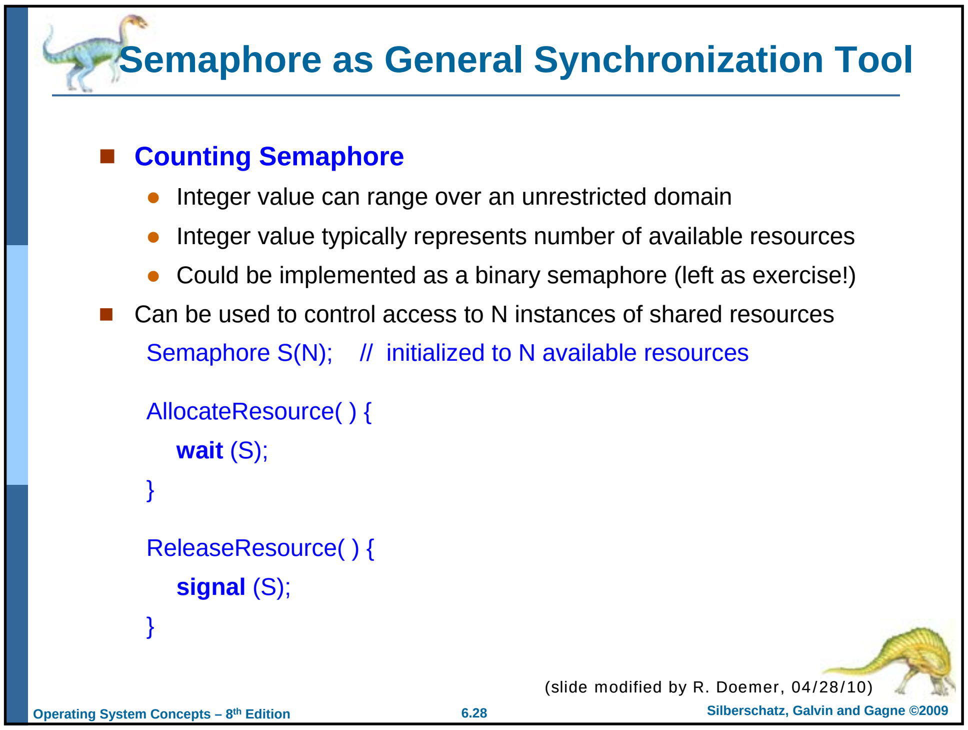


Semaphore as General Synchronization Tool

- **Binary Semaphore**
  - Integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex** lock or simply **lock**
- Provides mutual exclusion

```
Semaphore mutex(1);    //  initialized to 1
do {
        wait (mutex);
        // critical Section
        signal (mutex);
        // remainder section
} while (TRUE);
```

(slide modified by R. Doemer, 04/28/10)



Semaphore as General Synchronization Tool

- **Counting Semaphore**
  - Integer value can range over an unrestricted domain
  - Integer value typically represents number of available resources
  - Could be implemented as a binary semaphore (left as exercise!)
- Can be used to control access to N instances of shared resources

```
Semaphore S(N);    //  initialized to N available resources

AllocateResource( ) {
    wait (S);
}

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

(slide modified by R. Doemer, 04/28/10)

Semaphore as General Synchronization Tool

- **Signaling Semaphore**
  - Integer value initialized to 0
  - Integer value represents a flag for inter-process signaling
- Can be used to let a process Pi wait for another concurrent process Pj
  - Statements1( ) of Pj will be executed before Statements2( ) of Pi

```
Semaphore S(0);    //  initialized to 0

Process Pi:   wait (S);
              Statements2( );
              …

Process Pj:   Statements1( );
              signal (S);
              …
```

(slide modified by R. Doemer, 04/29/10)

Semaphore Implementation

- Must guarantee that no two processes can execute the code of wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the *critical section problem* where the wait and signal code are placed in the critical section.
  - Multi-processor systems often use **spinlock** (busy waiting) in critical section implementation
    - Implementation code is short
    - Little busy waiting if critical section is rarely occupied
    - Avoids context-switch if wait() and signal() are executed on different processors
  - Generally spinlocks are *not* a good solution because applications may spend lots of time in critical sections!

(slide modified by R. Doemer, 04/28/10)

Semaphore Implementation

- Implementation without Busy Waiting
  - With each semaphore, there is an associated waiting queue
  - Each entry in a waiting queue has two data items:
    - value – integer for semaphore value
    - list – queue of waiting processes
  - Two operations:
    - block – place the process invoking the operation on the appropriate waiting queue
    - wakeup – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

- Implementation without Busy Waiting

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

Semaphore Caveats

- **Deadlock**
  - Two or more processes are waiting indefinitely for an event that can be created only by one of the waiting processes
  - Example: 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);
```

- **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 a higher-priority process

(slide modified by R. Doemer, 04/28/10)

Classical Problems of Synchronization

- Classic examples of a large class of concurrency-control problems:
- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- We will use semaphores for synchronization here

(slide modified by R. Doemer, 04/29/10)

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
  - used as lock for mutual exclusive access
- Semaphore full initialized to the value 0
  - indicates number of full buffers
- Semaphore empty initialized to the value N.
  - indicates number of empty buffers

(slide modified by R. Doemer, 04/29/10)

Bounded-Buffer Problem

- The structure of the producer process

```
do {
        //   produce an item

        wait (empty);
        wait (mutex);

        //  add the item to the buffer

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

(slide modified by R. Doemer, 04/29/10)

Bounded-Buffer Problem

- The structure of the consumer process

```
do {
            wait (full);
            wait (mutex);

            //  remove an item from the buffer

            signal (mutex);
            signal (empty);

            //  consume the item

    } while (TRUE);
```

(slide modified by R. Doemer, 04/29/10)

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Integer readcount initialized to 0
    - Number of current readers
  - Semaphore mutex initialized to 1
    - Mutual exclusion to access readcount
  - Semaphore wrt initialized to 1
    - Mutual exclusion for writers (and first and last reader)

(slide modified by R. Doemer, 04/29/10)

Readers-Writers Problem

- The structure of a writer process

```
do {
        wait (wrt);

        // writing is performed

        signal (wrt) ;
   } while (TRUE);
```

(slide modified by R. Doemer, 04/29/10)

Readers-Writers Problem

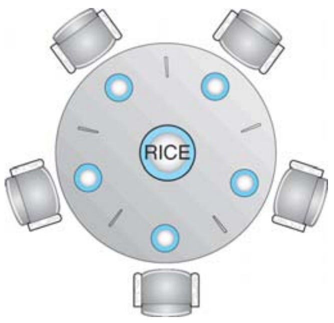
- The structure of a reader process

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

         // reading is performed

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

(slide modified by R. Doemer, 04/29/10)

Dining-Philosophers Problem

- Story
  - Dining philosophers spend their life
    - Thinking
    - Eating
  - Five sit at a table,
    bowl of rice in the middle,
    one chopstick to their left and right
  - Other than sharing a chopstick
    with their neighbor,
    they do not interact with each other
- Problem
  - Philosopher can only eat when no neighbor is eating
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



(slide modified by R. Doemer, 04/29/10)

Dining-Philosophers Problem

- The structure of philosopher i:

```
do  {
            wait ( chopstick[ i ] );
            wait ( chopstick[ (i + 1) % 5] );

            //  eat

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

            //  think

    } while (TRUE);
```

- Note: Only mutual exclusion is solved here.

(slide modified by R. Doemer, 04/29/10)

Dining-Philosophers Problem

- Deadlock problem
  - All philosophers pick up first chopstick at the same time
- Deadlock solutions
  - Allow only 4 philosophers at the table for 5
  - Pick both chopsticks at the same time (add a critical section)
  - Asymmetric solution:
    - Odd philosophers pick up left chopstick first
    - Even philosophers pick up right chopstick first
- Starvation problem
  - Deadlock free does not mean starvation free!
  - Left as an exercise!

(slide added by R. Doemer, 04/29/10)

Monitors

- Programmer's problems with semaphores
  - Frequent incorrect use of semaphore operations:
    - signal (mutex) …. wait (mutex)
    - wait (mutex) … wait (mutex)
  - Frequent omitting
    - of wait (S)
    - or signal (S)
    - or both!
- **Monitors** offer a solution (in the programming language!) that relieves the programmer of the above problems
  - Basically, the compiler automatically inserts the mutex and its handling!

(slide modified by R. Doemer, 04/30/10)

Monitors

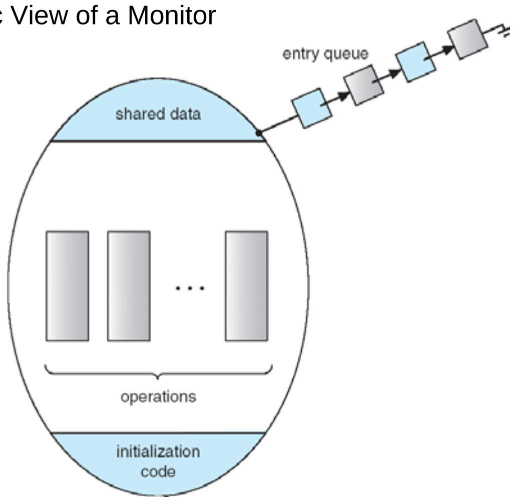
- **Monitor**
  - A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract Data Type (ADT)
  - Only one process may be active within the monitor at any time
  - Shared variables can only be accessed through local procedures

```
monitor monitor-name
{
   // shared variable declarations

   procedure P1 (…) { … }
            …
   procedure Pn (…) { … }

   initialization(…) { … }
}
```

(slide modified by R. Doemer, 04/30/10)

Monitors

- Schematic View of a Monitor



(slide modified by R. Doemer, 04/29/10)

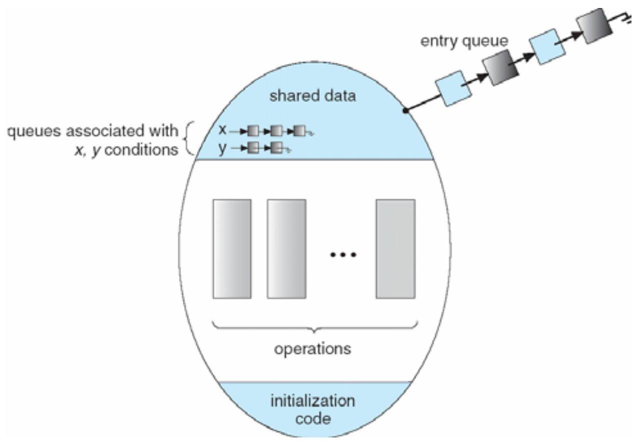
Condition Variables in Monitor

- Monitor construct defined so far is not yet powerful enough to solve general synchronization problems
- **Condition Variables** are needed in the monitor to pass control from one process to another
  - condition x;
- Two operations exist on a condition variable:
  - x.wait()
    - a process that invokes the operation is suspended
    - in turn, another process may enter the monitor
  - x.signal()
    - resumes *one* of the processes that invoked x.wait()
    - if no process is waiting, signaling has no effect
- Note: Many implementations also offer x.broadcast() which will allow all waiting processes to resume (one after another)

(slide modified by R. Doemer, 04/30/10)

Condition Variables in Monitor

- Schematic View of a Monitor with Condition Variables



(slide modified by R. Doemer, 04/29/10)

Condition Variables in Monitor

- Example:
  - Process Q suspends in monitor on condition x
    - Q: x.wait ()
  - Process P enters monitor and signals condition x
    - P: x.signal ()
  - Now, both processes can conceptually continue their execution.
  - However, only one may be active in the monitor at any time!
  - Choice between two possibilities:
    1. P waits until Q leaves the monitor (or waits for another condition)
       - Called "signal and wait" (aka. *"Hoare-style"*)
    2. Q waits until P leaves the monitor (or waits for another condition)
       - Called "signal and continue" (aka. *"Mesa-style"*)
       - This is implemented by Pthreads and Nachos condition variables!

(slide modified by R. Doemer, 04/30/10)

Monitor Solution to Dining Philosophers

- Deadlock-free solution to dining-philosophers problem
  - Philosopher picks up chopsticks when both are available
  - Each philosopher i invokes the monitor operations pickup() and putdown() in the following sequence:
    - DiningPhilosophers.pickup(i);
    - EAT
    - DiningPhilosophers.putdown(i);
  - We use the following to describe the state of each philosopher
    - enum { THINKING; HUNGRY, EATING) state [5];
  - We also use the following condition variables in which a philosopher can wait when hungry but no chopsticks are available
    - condition self [5];

(slide modified by R. Doemer, 04/29/10)

Monitor Solution to Dining Philosophers

- Deadlock-free solution to dining-philosophers problem

```
monitor DP
{
        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((i + 4) % 5);
        test((i + 1) % 5);
        }
…
```

(slide modified by R. Doemer, 04/29/10)

Monitor Solution to Dining Philosophers

- Deadlock-free solution to dining-philosophers problem

```
…
        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;
        }
}
```

- Note: This is not a starvation-free solution! (left as an exercise)

(slide modified by R. Doemer, 04/29/10)

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads API

(slide modified by R. Doemer, 04/29/10)

Pthreads Synchronization

- Pthreads API is OS-independent (portable)
- It provides:
  - mutex locks
  - condition variables (*"Mesa-style"*)
- Non-portable extensions include:
  - read-write locks
  - spin locks
  - semaphores

(slide modified by R. Doemer, 04/30/10)

Pthreads Synchronization

- **Pthreads Mutex Objects**
  - Act as binary semaphores initialized to 1
  - Implement mutual exclusion for a critical region
- Creating a pthread mutex:

```
pthread_mutex_t mutex = PHTREAD_MUTEX_INITIALIZER;
```

  - Equivalent to: Semaphore S(1);
- Locking a pthread mutex: (entering a critical region / monitor)

```
pthread_mutex_lock(&mutex);
```

  - Equivalent to: wait(S);
- Unlocking a pthread mutex: (leaving a critical region/ monitor)

```
pthread_mutex_unlock(&mutex);
```

  - Equivalent to: signal(S);

(slide added by R. Doemer, 04/29/10)

Pthreads Synchronization

- **Pthreads Condition Variables**
  - Act as condition variables in a monitor implemented by a pthread mutex
  - Allow to pass control predictably from one thread to another
- Creating a pthread condition variable:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

  - Equivalent to: condition x;
- Waiting on a pthread condition: (atomically releases/re-acquires the mutex)

```
pthread_cond_wait(&cond, &mutex);
```

  - Equivalent to: x.wait();
  - Monitor mutex must be held by the calling thread and implicitly will be reacquired by the thread upon return
- Signaling a pthread condition: (signals a waiting thread and continues)

```
pthread_cond_signal(&cond);
```

  - Equivalent to: x.signal();
  - Monitor mutex must be held by the calling thread in order to achieve predictable scheduling behavior

(slide added by R. Doemer, 04/30/10)

End of Chapter 6