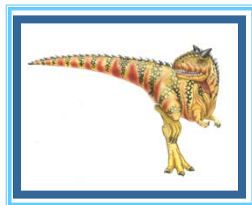
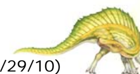
Chapter 6: Process Synchronization



(slides selected/modified by R. Doemer, 01/12/11)

Chapter 6: Process Synchronization

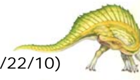
- Background
- The Producer-Consumer Problem
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
  - Pthread Synchronization
- Atomic Transactions



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

Background

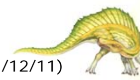
- Concurrent execution of processes or threads creates situations of **non-determinism**!
  - CPU scheduling by operating system often (!) yields *non-deterministic* order of execution of concurrent program instructions
  - e.g. thread may be preempted at any time (!)
- Concurrent access to shared data 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 Example

- Paradigm for cooperating processes
  - **Producer** process produces information that is consumed by a **consumer** process
- Buffered communication
  - *Bounded-buffer* assumes that there is a fixed buffer size
  - Both consumer and producer access shared data

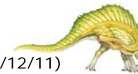


(slide inserted from chapter 3 and modified by R. Doemer, 01/12/11)

Producer-Consumer Example

- Bounded buffer implementation
  - 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 */
```

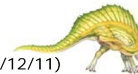


(slide modified by R. Doemer, 01/12/11)

Producer-Consumer Example

- Producer implementation
  - 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, 01/12/11)

Producer-Consumer Example

- Consumer implementation
  - 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, 01/12/11)

Producer-Consumer Example

- Discussion on Implementation
  - 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)

(slide modified by R. Doemer, 01/12/11)

Producer-Consumer Example

- *Implementation 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, 01/12/11)

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

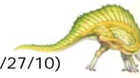
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

(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

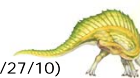


(slide modified by R. Doemer, 04/27/10)

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:

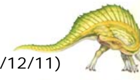
```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```



(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, 01/12/11)

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)

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++;
    }
    ```

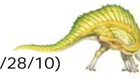
(slide modified by R. Doemer, 04/28/10)

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)

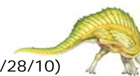


Semaphore as General Synchronization Tool

- **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)

Monitors

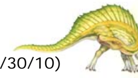
- Programmer's problems with semaphores
  - Frequent incorrect use of semaphore operations:
    - signal (mutex) …. wait (mutex)
    - wait (mutex) … wait (mutex)
  - Frequent omitting
    - of wait (S)
    - or signal (S)
    - or both!
- **Monitors** offer a solution (in the programming language!) 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

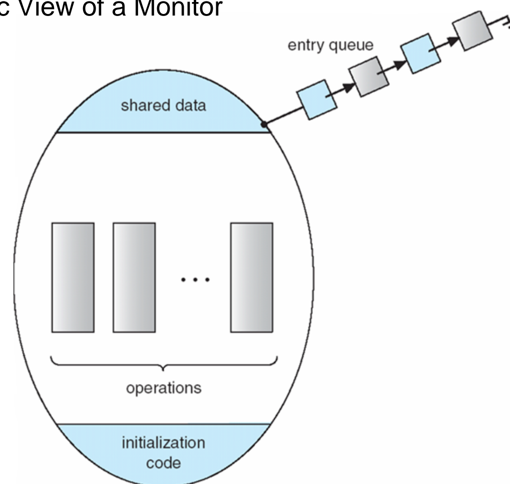
```
monitor monitor-name
{
  // shared variable declarations

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

  initialization(…) { … }
}
```

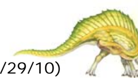


(slide modified by R. Doemer, 04/30/10)

Monitors

- Schematic View of a Monitor

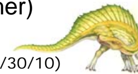




(slide modified by R. Doemer, 04/29/10)

Condition Variables in Monitor

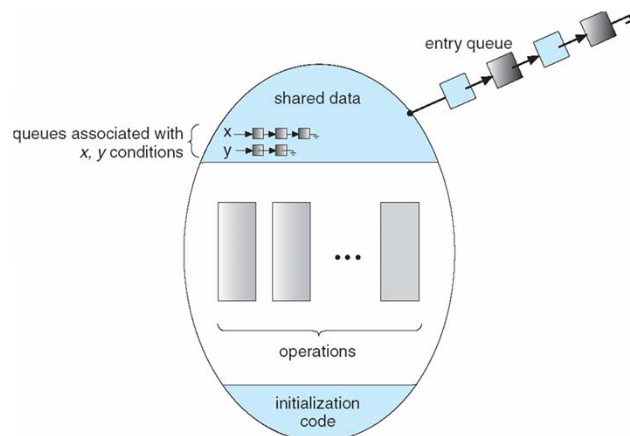
- Monitor construct defined so far is not yet powerful enough to solve general synchronization problems
- **Condition Variables** are needed in the monitor to pass control from one process to another
  - condition x;
- Two operations exist on a condition variable:
  - x.wait()
    - a process that invokes 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)

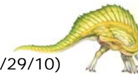


(slide modified by R. Doemer, 04/30/10)

Condition Variables in Monitor

- Schematic View of a Monitor with Condition Variables

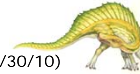




(slide modified by R. Doemer, 04/29/10)

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)

End of Chapter 6