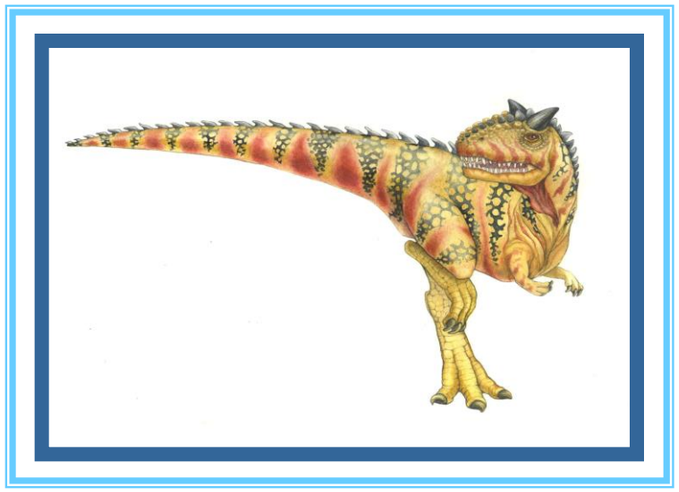
Chapter 5: Process Synchronization



Silberschatz, Galvin and Gagne ©2013



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples





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 examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems





Process Synchronization

- A cooperating process is one that can affect or be affected by other processes executing in the system.

- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.





Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency (Examples)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- **Illustration of the problem:**

  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.





Producer

- The code for the producer process can be modified as follows:

```
while (true) {
   /* produce an item in next produced */

   while (counter == BUFFER_SIZE) ;
         /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```





Consumer

The code for the consumer process can be modified as follows:

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

Note: Although the producer and consumer routines shown above and in previous slide are correct separately, they may not function correctly when executed concurrently.





Race Condition

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

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

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

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

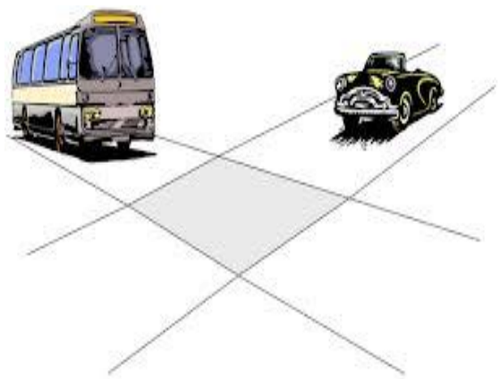
- Consider this execution interleaving with "counter = 5" initially: concurrent execution of both producer and consumer.

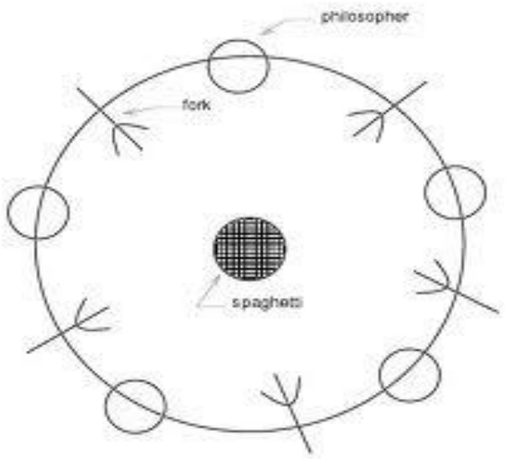
S0: producer execute **register1 = counter** {register1 = 5}
S1: producer execute **register1 = register1 + 1** {register1 = 6}
S2: consumer execute **register2 = counter** {register2 = 5}
S3: consumer execute **register2 = register2 – 1** {register2 = 4}
S4: producer execute **counter = register1**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

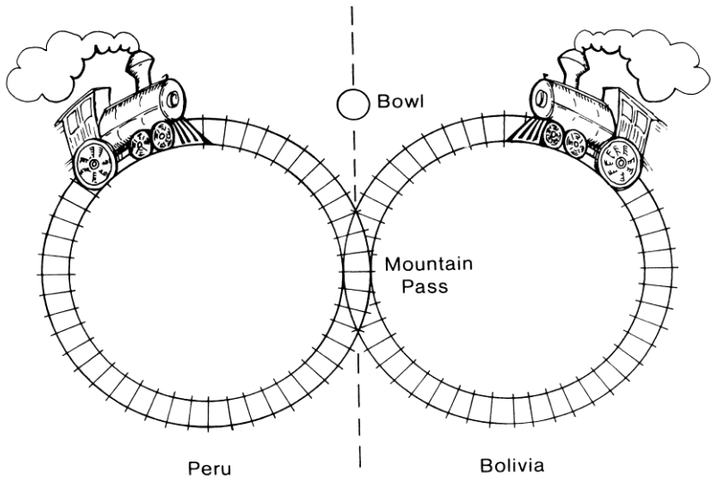


Critical Section Problem

- We begin our consideration of process synchronization by discussing the so-called critical-section problem.
- Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections at the same time.











Critical Section Problem (cont)

- The ***critical-section problem*** is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.
- The general structure of a typical process *Pi* is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code. See the Text book., Chapter 5. page 203





Critical Section

- General structure of process p_i is

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





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 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

- Two approaches depending on if kernel is preemptive or non-preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
- Under non_preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the **process** either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows.





Peterson's Solution

- It is a good algorithmic description of solving the problem, Peterson's solution is restricted to two processes.
- Two process solution only, no guarantees that Peterson's solution will work correctly in some architectures.
- Assume that the **`load`** and **`store`** instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready.
- What is the disadvantage of Peterson's solution to the critical section solution?





Peterson's Solution

P_0

i = 0, j = 1

```
while (true) {
   flag[i] = true;
   turn = j;
   while (flag[j] &&
          turn == j)
     ; // busy wait
   // critical section
   // …
   flag[i] = false;
   // remainder section
   // …
}
```

P_1

i = 1, j = 0

```
while (true) {
   flag[i] = true;
   turn = j;
   while (flag[j] &&
          turn == j)
     ; // busy wait
   // critical section
   // …
   flag[i] = false;
   // remainder section
   // …
}
```





Peterson's Solution

entry section

exit section

```
while ( true ) {
   flag[i] = true;
   turn = j;
   while ( flag[j] && turn == j )
      ; // busy wait
   // critical section
   // …
   flag[i] = false;
   // remainder section
   // …
}
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met





Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- 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-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Kindly, See the text book





Solution to Critical-section Problem Using Locks

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





Solution using test_and_set()

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing   P1*/

   /* critical section */

      p0, lock = false;

   /* remainder section */
} while (true);
```





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;
}
```





Solution using compare_and_swap

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

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


do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */   here  lock =1
      /* critical section */
   lock = 0;
      /* remainder section */
} while (true);
```





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
- Product critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not

- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions

- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock** تدوير القفل





acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true; // for the other processes
}
```

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





Semaphore

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

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





Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Then a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

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





Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

  what is the solution?





Semaphore Implementation with no Busy waiting

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

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





Semaphore Implementation with no Busy waiting (Cont.)

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





getAndSet vs. Semaphores

```
private AtomicBoolean mutex;
…
while ( mutex.getAndSet( true ) )
  Thread.yield();

// critical section

mutex.set( false );

// remainder section
…
```

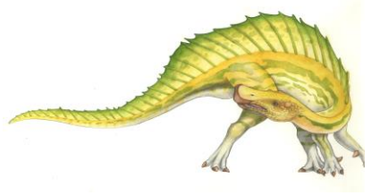
```
Semaphore sem = new Semaphore(1);

sem.acquire();

   // critical section

sem.release();

   // remainder section
```





Busy waiting

```
acquire() {
    while (value <= 0)
       ; // no-op
    value--;
}

release() {
    value++;
}
```

© CourseSmart





Busy waiting vs. Blocking

```
acquire() {
    while (value <= 0)
       ; // no-op
    value--;
}

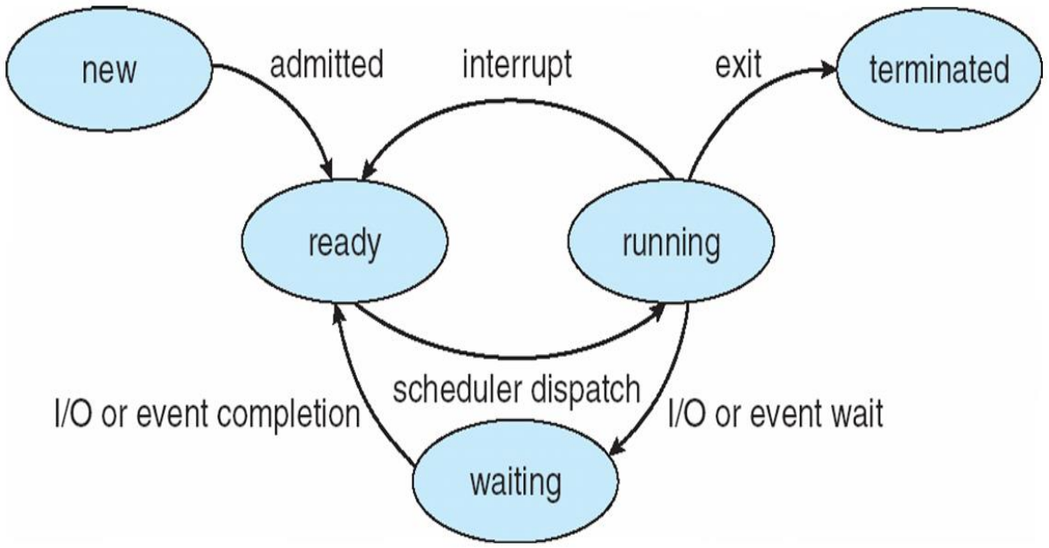
release() {
    value++;
}
```

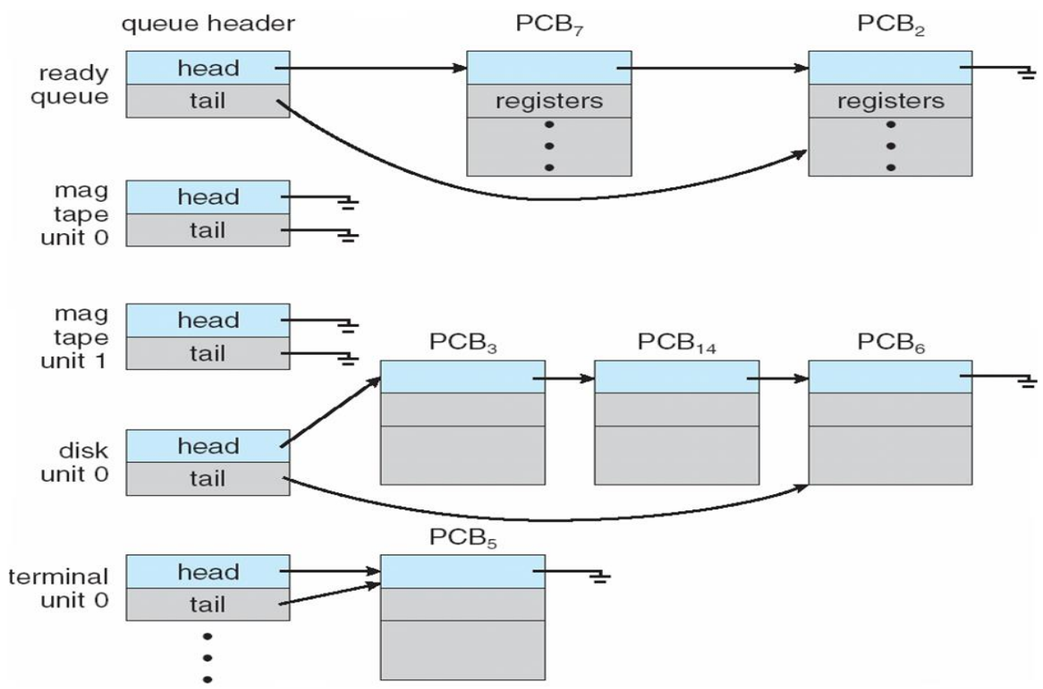
```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block();
    }
}

release(){
    value++;
    if (value <= 0) {
       remove a process P from list
       wakeup(P);
    }
```



Blocking & Waking up






```
acquire(){
    value--;
    if (value < 0) {
        add this process to list
        block();
    }
}

release(){
    value++;
    if (value <= 0) {
        remove a process P from list
        wakeup(P);
    }
}
```



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 treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0
  - Memory locations
  - Shared File





Classic Synchronization Problems

- Representative of large class of problems
- Used to test prospective solutions
- Examples:
  - Bounded Buffer Problem
  - Readers and Writers Problem
  - Dining Philosophers Problem





Readers-Writers Problem (Cont.)

- The structure of a writer process

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





Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
   wait(mutex);
   read count++;
   if (read_count == 1)
      wait(rw_mutex);    //recursive call
   signal(mutex);

   ...
   /* reading is performed */
   ...

   wait(mutex);
   read count--;
   if (read_count == 0)
      signal(rw_mutex); // function call
   signal(mutex);
} while (true);
```





Readers-Writers Problem Variations

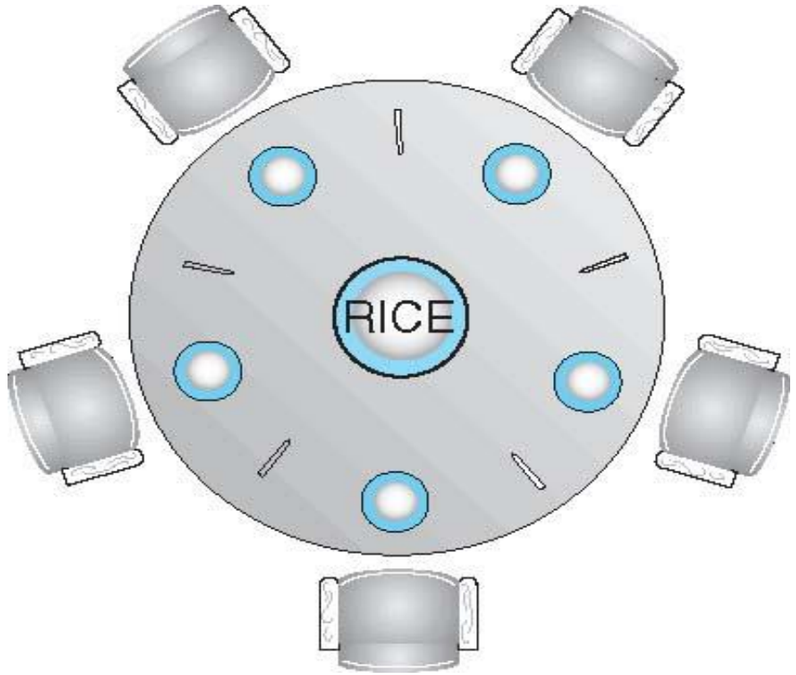
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

الاختلاف الأول





Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1





Dining-Philosophers Problem Algorithm

- 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);
```

- What is the problem with this algorithm?





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);    // left
             test((i + 1) % 5);    // right
         }
```





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





Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads





Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted // none preemptive
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)





Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption



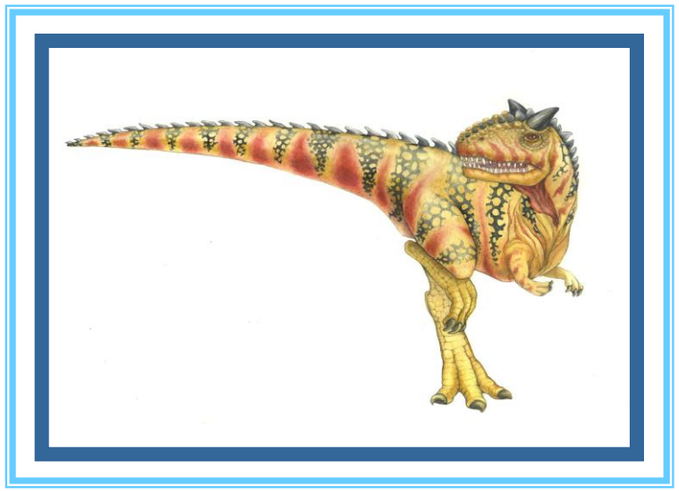


Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage



End of Chapter 5



Silberschatz, Galvin and Gagne ©2013