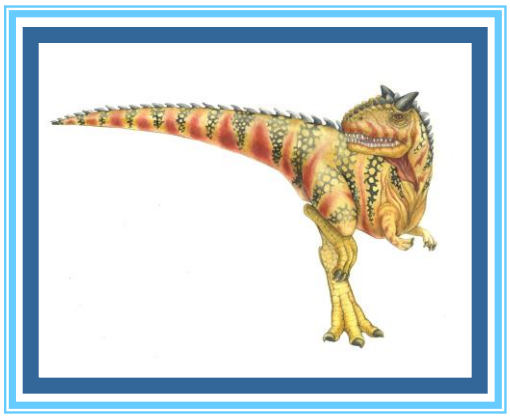
Chapter 7: Synchronization Examples

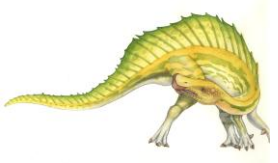


Silberschatz, Galvin and Gagne ©2018



Outline

- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain the dining-philosophers synchronization problem
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX and Java can be used to solve process 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

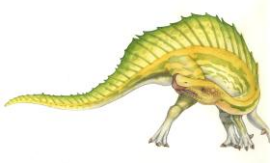


Troy University Resources



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);
}
```



Troy University Resources



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

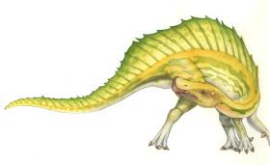
```
while (true) {
    wait(full);
    wait(mutex);
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
       /* consume the item in next consumed */
        ...
   }
```





Readers-Writers Problem

- Multiple readers may access the shared data concurrently.
- A writer needs exclusive access: it cannot overlap another writer or any reader.
- A policy must decide when readers and writers enter if both are waiting.
- The following semaphore solution gives readers preference and may starve writers.

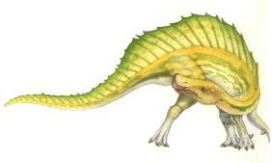


Troy University Resources



Readers-Writers Problem (Cont.)

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





Readers-Writers Problem (Cont.)

- The structure of a writer process

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



Troy University Resources



Readers-Writers Problem (Cont.)

- 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 Policies

Policy	Admission rule	Possible problem
Reader preference	A new reader may join active readers.	A stream of readers can starve writers.
Writer preference	Queued writers prevent new readers from entering.	A stream of writers can starve readers.
Fair admission	Queue requests or bound overtaking.	Fairness depends on the lock implementation and scheduler.

A reader-writer lock supplies an API. Its documented policy determines the actual fairness guarantees.

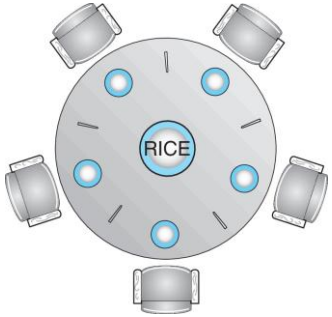


Troy University Resources

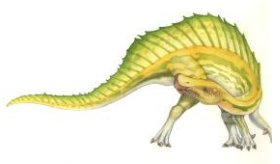


Dining-Philosophers Problem

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



- They spend their lives alternating 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 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





Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher i :

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

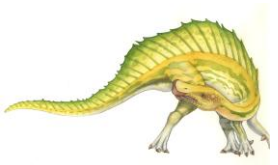
     /* eat for awhile */

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

     /* think for awhile */

}
```

- What is the problem with this algorithm?

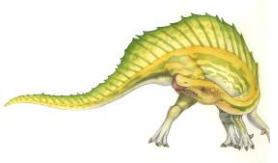


Troy University Resources



Dining Philosophers: Avoiding Deadlock

- Resource order: number the chopsticks and always acquire the smaller-numbered one first.
- Limit admission: allow at most N - 1 philosophers to try acquiring chopsticks at once.
- Use a monitor that grants permission only when both neighbors are not eating.
- These designs break the deadlock cycle. Starvation freedom still requires a fair admission policy.





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);
          while (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);
   }
```



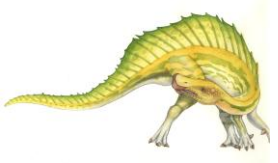
Troy University Resources



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.)

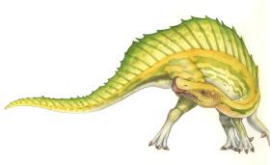
- Each philosopher "i" invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

/** EAT **/

DiningPhilosophers.putdown(i);
```

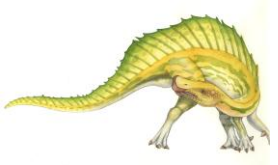
- No deadlock, but starvation is possible





Kernel Synchronization: Windows

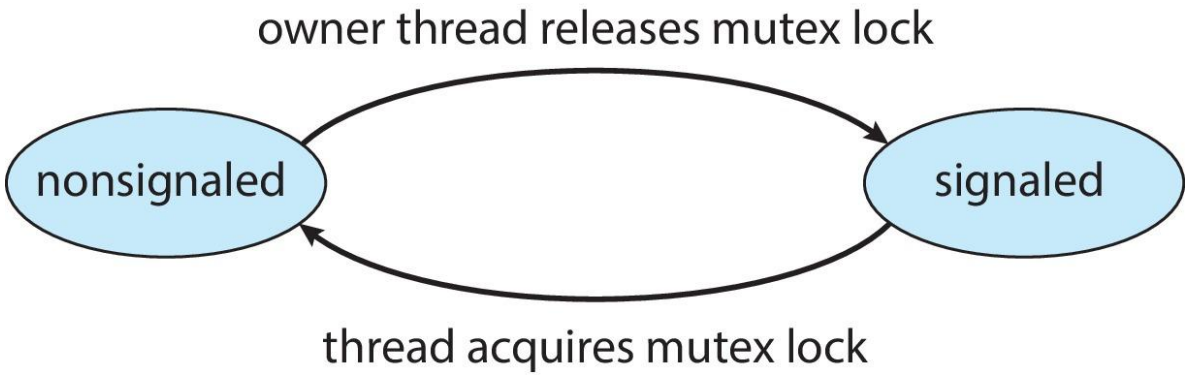
- Kernel spinlock APIs raise IRQL as required to protect short critical sections. They also synchronize access from other CPUs.
- At DISPATCH_LEVEL a thread must not perform a blocking wait or access pageable data.
- Dispatcher objects include mutexes, semaphores, events, and timers. Threads can wait for their signaled state.
- An event retains signaled state according to its reset mode. A condition variable has different wakeup semantics.





Kernel Synchronization - Windows

- Mutex dispatcher object

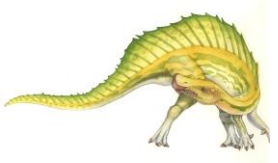






Kernel Synchronization: Linux

- Linux provides atomic operations, mutexes, semaphores, spinlocks, reader-writer locks, and other mechanisms.
- Kernel preemption is configurable. “Linux 2.6 and later is fully preemptive” is not a universal rule.
- On a non-PREEMPT_RT kernel, spinlock_t disables preemption while held. Interrupt-safe variants also manage local interrupt state.
- PREEMPT_RT changes spinlock_t to a sleeping, priority-inheriting implementation. raw_spinlock_t retains strict spinning behavior.





Linux Atomic Integer Operations

- atomic_t provides atomic operations on an integer.
- The illustrated sequence sets counter to 5 and ends with value = 12.
- Atomicity of one counter does not by itself protect a larger shared invariant.

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

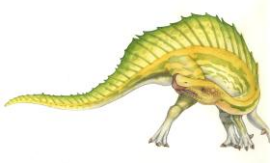
Memory ordering depends on the operation. An atomic read or set is not a general memory barrier.





Solaris Adaptive Mutexes

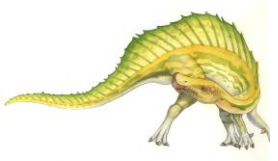
- Solaris uses adaptive mutexes for many short kernel critical sections.
- If the owner is currently running on another CPU, a contender can spin briefly.
- If the owner is not running, the contender blocks so the scheduler can run useful work.
- This adapts the waiting strategy to the owner's execution state. It is not simply a semaphore renamed as a spinlock.





Solaris Turnstiles

- Solaris uses turnstiles to manage threads waiting for adaptive mutexes and reader-writer locks.
- The turnstile records waiters and supports priority inheritance for the lock owner.
- Turnstile structures are associated with contended locks through the kernel's allocation and lending scheme.
- The goal is to avoid delaying a high-priority waiter behind unrelated medium-priority work.

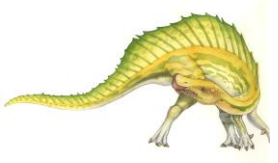


Troy University Resources



POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS





POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

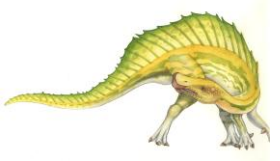
/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

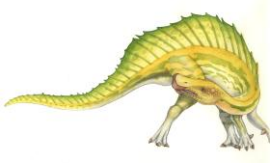
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```





Named and Unnamed POSIX Semaphores

- A named semaphore is opened by name with sem_open(). Unrelated processes can open the same name.
- An unnamed semaphore is initialized with sem_init(). With pshared == 0, threads in one process share it.
- With nonzero pshared, an unnamed semaphore in shared memory can be shared by unrelated processes if the platform supports it.
- Check API and platform support. The same memory location must refer to the same shared semaphore object.





POSIX Named Semaphores

Create or open a named semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

Processes use the same name to open the same semaphore.
Acquire and release a permit:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Use "/SEM" as the POSIX name. Check SEM_FAILED and handle EINTR from sem_wait().



Troy University Resources



POSIX Unnamed Semaphores

Initialize an unnamed semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

Acquire and release a permit:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

pshared = 0 shares between threads. Use nonzero pshared and shared storage for interprocess use.





POSIX Condition Variables

A POSIX condition variable works with a mutex that protects the shared predicate. Initialize both objects:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

- pthread_cond_wait() atomically releases the mutex and waits.
- It reacquires the mutex before returning.
- Check the predicate in a while loop because wakeups can be spurious.





Waiting and Signaling a Predicate

Wait for a == b while holding the mutex:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

Change the predicate and signal while holding the same mutex:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

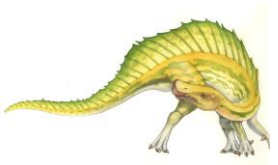
The condition variable does not store permits. Recheck the predicate after every return from the wait.





Java Synchronization

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables

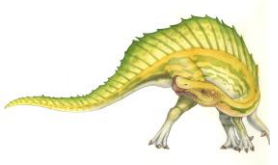


Troy University Resources



Java Monitors

- Each Java object has an associated intrinsic monitor lock. synchronized code acquires that lock automatically.
- An instance synchronized method locks its receiver. A static synchronized method locks its Class object.
- The lock is reentrant and is released when the synchronized block or method exits, including exceptional exit.
- Only one thread owns a given monitor lock at a time. Different object monitors are separate locks.





Bounded Buffer – Java Synchronization

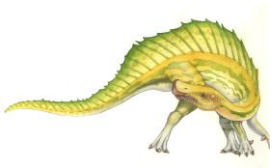
```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

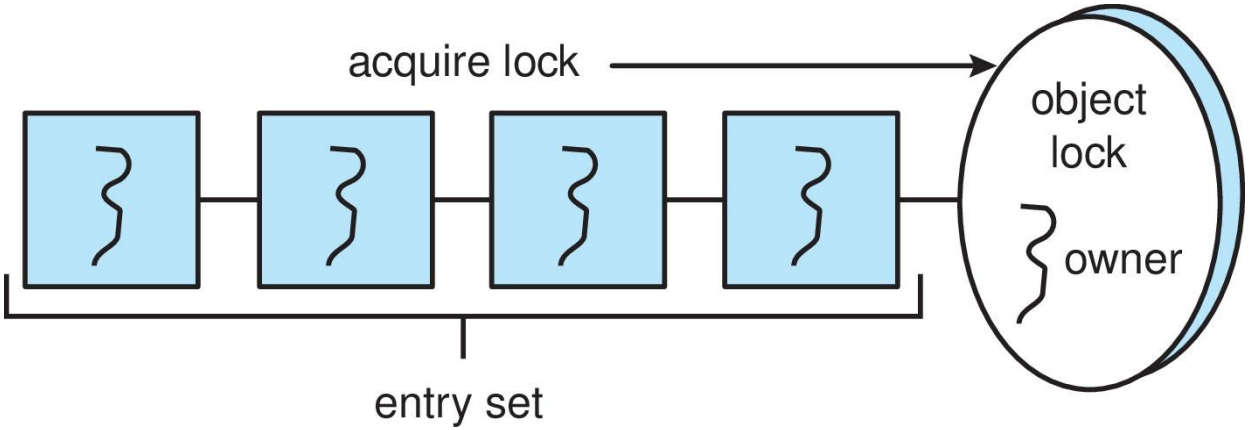


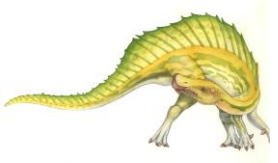
Troy University Resources



Java Synchronization

- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:

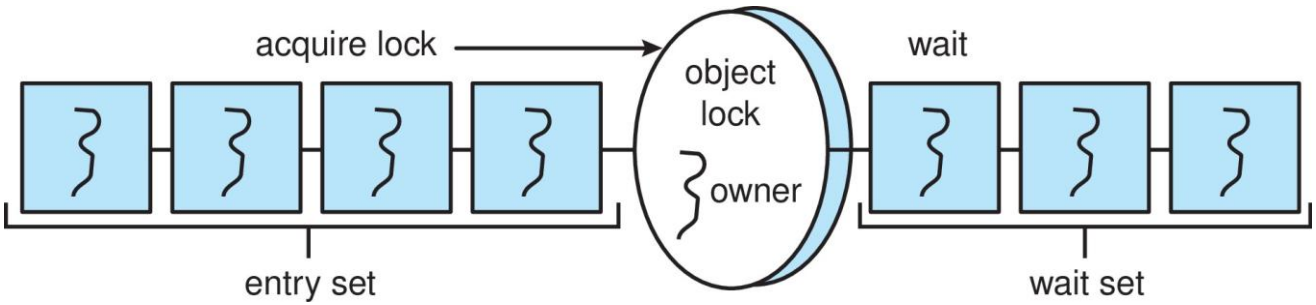




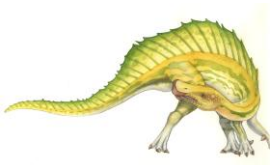


Java Object.wait()

- Call wait() while holding the object's monitor.
- wait() releases that monitor and places the thread in the object's wait set.
- An untimed wait uses Thread.State.WAITING. BLOCKED means waiting to acquire a monitor lock.
- Before wait() returns, the thread must reacquire the monitor.



Use a while loop around wait() to recheck the predicate.





Java Notification

- notify() selects an arbitrary waiter on the same object. notifyAll() wakes all its waiters.
- The notifying thread continues while it holds the monitor. Notification does not transfer ownership immediately.
- An awakened thread competes to reacquire the monitor, then rechecks the predicate in a while loop.
- For a bounded buffer with producers and consumers sharing one wait set, notifyAll() avoids waking only an ineligible class of waiter.





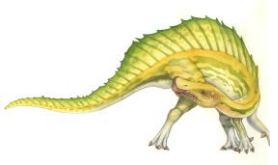
Java Bounded Buffer: insert()

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   count++;

   notify();
}
```

Use notifyAll() for mixed producers and consumers. Propagate InterruptedException.





Java Bounded Buffer: remove()

```
/* Consumers call this method */
public synchronized E remove() {
   E item;

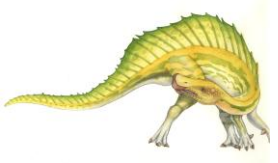
   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

Use notifyAll() for mixed waiters. Propagate InterruptedException.





Java Reentrant Locks

- Similar to mutex locks
- The **`finally`** clause ensures the lock will be released in case an exception occurs in the **`try`** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```





Java Semaphores

A Semaphore counts available permits:

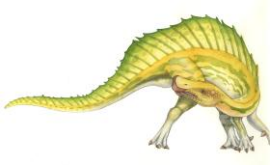
```
Semaphore(int value);
```

Acquire a permit before using the protected resource:

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

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Enter try only after acquire() succeeds. Release that permit in finally. Propagate interruption.



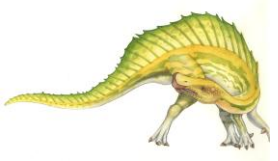


Java Lock Conditions

Create a Condition from its associated ReentrantLock:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- Hold the lock when calling await(), signal(), or signalAll().
- await() releases the lock while waiting and reacquires it before returning.
- Use a while loop to test the predicate. Always unlock in finally.





Java Conditions: Turn-Taking State

- Five threads use turn to select whose work can proceed.
- Each condition belongs to the same ReentrantLock.
- In doWork(), wait with while (threadNumber != turn), because await() may wake spuriously.
- Propagate interruption. Update turn before signaling the next thread, and unlock in finally.

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```





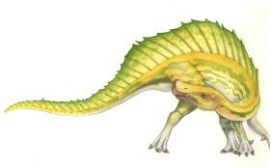
Java Condition Variables

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled.
       */
      if (threadNumber != turn)
         condVars[threadNumber].await();

      /**
       * Do some work for awhile ...
       */

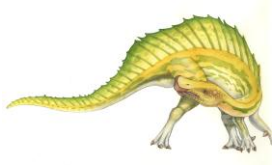
      /**
       * Now signal to the next thread.
       */
      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```





Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages





Transactional Memory

A mutex can make the updates in a critical section execute with mutual exclusion:

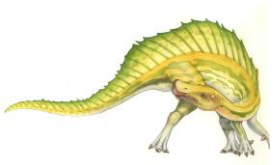
```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

A memory transaction commits its reads and writes atomically. The atomic block below is conceptual pseudocode, not standard C syntax.

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```



Troy University Resources

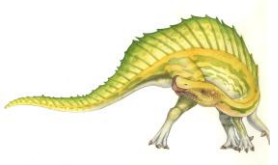


OpenMP critical

All accesses that need mutual exclusion for count must use the same named critical region or an equivalent protocol.

```
void update(int value) {
    #pragma omp critical(counter_update)
    {
        count += value;
    }
}
```

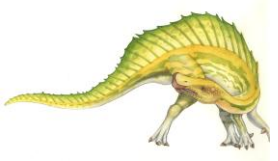
critical serializes matching regions among threads in the contention group. For one simple increment, omp atomic can also apply.





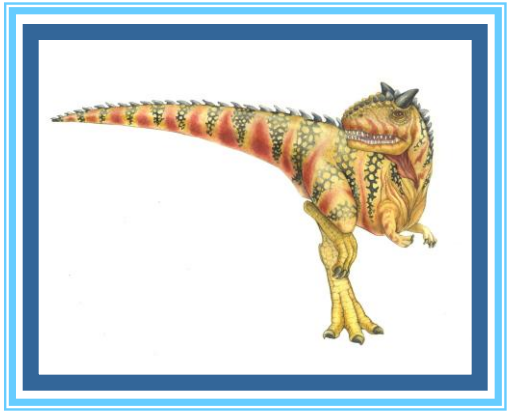
Functional Programming and Shared State

- Immutable data reduces the need to coordinate writes to shared objects.
- Pure functions compute results without modifying shared state.
- Erlang emphasizes isolated processes and message passing. Scala supports functional programming alongside mutable and object-oriented code.
- A functional language can still expose shared mutable state. Its presence still requires a correct synchronization design.



Troy University Resources

End of Chapter 7



Silberschatz, Galvin and Gagne ©2018