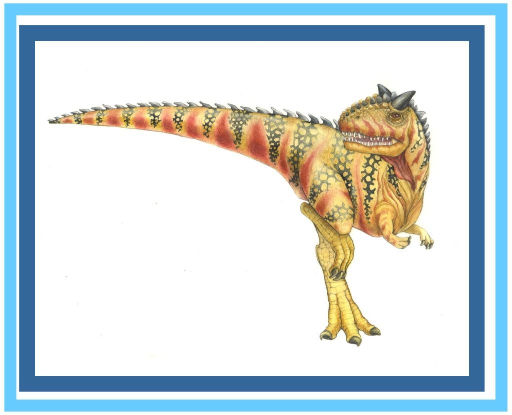
Chapter 6: Process Synchronization



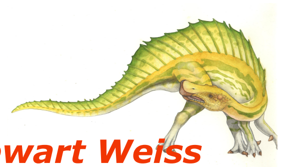
Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Synchronization Examples

modified by Stewart Weiss

Objectives

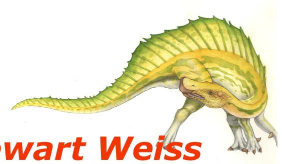
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem



modified by Stewart Weiss

Background

- Concurrent access to shared data may result in data inconsistency (we will show this soon)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The producer-consumer problem will demonstrate what the issues are.
- Recall –
  - producers fill buffers that are "consumed" by consumers;
  - producers and consumers run asynchronously, concurrently, and in perpetuity;
  - buffers are used to "smooth" the variation in running speed, so that the two types of processes do not need to run in lockstep;
  - the problem of interest is the bounded buffer problem – we have a finite number of buffers.



modified by Stewart Weiss

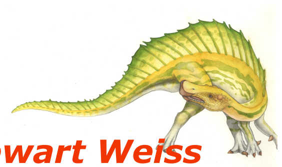
Producer-Consumer Problem, Revisited

- The solution in Chapter 3 worked by leaving one buffer empty all of the time.
- Here we develop a solution that uses all the buffers. We can do so by having an integer `count` that keeps track of the number of full buffers.
- Initially, `count` 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

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

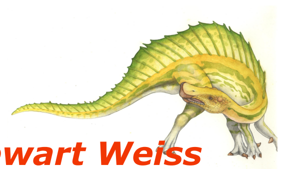
- The producer sits in the loop called BUSY WAIT until the count is decremented by the consumer. It then fills `buffer[in]`, increments `in` and `count`.



modified by Stewart Weiss

Consumer

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



modified by Stewart Weiss

Race Conditions

- The ordinary implementation of an assignment instruction such as

```
count = count + 1;
```

is

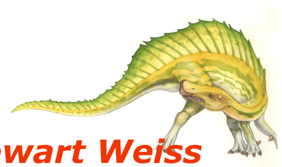
```
register1 = count              MOV count, R1
register1 = register1 + 1      INC  R1
count = register1              MOV R1, count
```

- Similarly, the implementation of

```
count = count - 1;
```

is

```
register2 = count              MOV count, R2
register2 = register2 – 1      DEC R2
count = register2              MOV R2, count
```

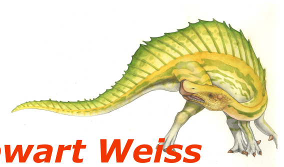


Race Conditions

- Suppose that count = 5 initially and that the processes are time sliced onto the processor so that these machine instructions get executed in the following sequence. (You can also imagine that there are two processors and they execute on the separate processors and execute the instructions in this order.)

S0: producer executes	register1 = count	{register1 = 5}
S1: producer executes	register1 = register1 + 1	{register1 = 6}
S2: consumer executes	register2 = count	{register2 = 5}
S3: consumer executes	register2 = register2 - 1	{register2 = 4}
S4: producer executes	count = register1	{count = 6 }
S5: consumer executes	count = register2	{count = 4}

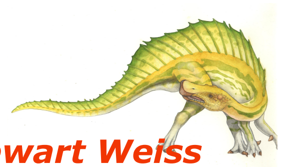
- The result is that, even though the producer incremented the counter and the consumer decremented it, so that the final value should remain 5, it is not. It is count = 4. If we switched instruction S4 and S5, the final value would be count = 6.
- ***The final value depends on the order in which the instructions are executed!!!***
- This is called a ***race condition.***



modified by Stewart Weiss

Race Conditions

- Race conditions must be eliminated in the concurrent execution of cooperating processes.
- To prevent them, we must ensure that only one process at a time updates the value of the shared variable.
- The rest of this chapter deals with ways of preventing race conditions in both hardware and software.



Critical Sections

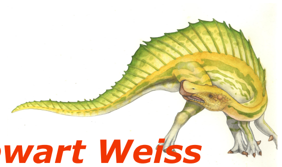
- A section of executable code in a process, that refers to a resource (such as a variable) that is shared by one or more other processes is called a ***critical section*** (***cs***) with respect to that resource.
- Example – in producer-consumer code, in the producer,

  ```
  count = count + 1
  ```

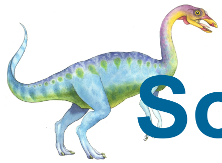
  is a critical section wrt `count`, and

  ```
  count = count -1
  ```

  is a critical section wrt `count`.
- Processes must execute their cs's in mutual exclusion: only one process at a time should be allowed to execute in its critical section with respect to a resource
- The ***Critical Section problem*** is the problem of designing a protocol that processes can use to cooperate in the sharing of a resource.



modified by Stewart Weiss

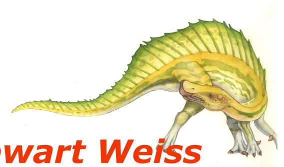


Solutions to Critical-Section Problem

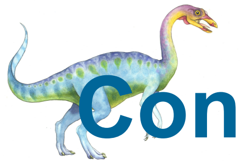
- A protocol for the CS problem has a general form. Each process can be viewed as being in one of 4 possible sections of code:

entry section:	***request permission to enter CS***
critical section:	***execute CS***
exit section:	***indicate completion of CS***
remainder section:	***non-CS code***

- This indicates that a process must request access to CS, then enter the CS, then indicate leaving CS and then do stuff that is not in the CS. It might repeat this indefinitely.
- ***Henceforth CS always refers to same resource***

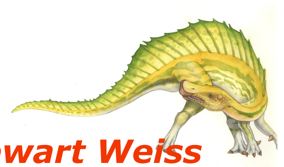


modified by Stewart Weiss



Constraints on Solutions to CS Problem

1. ***Mutual Exclusion*** - If a process 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 trying to enter their critical section, then only those processes not in their remainder sections can participate in the decision about which enters CS next, and this decision must be made in finite time.
3. ***Bounded Waiting*** - There is a bound on the number of times that other processes enter their critical sections after a process has made a request to enter its critical section and before that request is granted (fairness property)

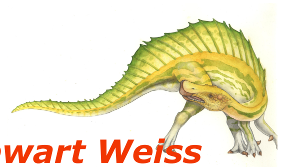


modified by Stewart Weiss

Further Assumptions

- All processes make progress – execute at nonzero speed
- Processes may execute at unpredictable speed and in no relation to the speeds of other processes.
- Machine instructions to fetch and store from memory and to evaluate a variable are atomic.
- Processes never halt in their CS's.
- The priorities of processes are arbitrary and unconstrained.
- Every process is in a loop of the form

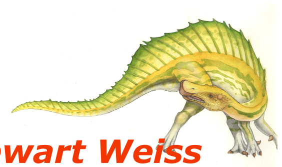
```
loop {
    entry section
    CS
    exit section
    remainder section
}
```



modified by Stewart Weiss

Race Conditions in Kernel

- In the kernel – system calls that open files write to system open file table, so access to table is a race
- In the kernel – system calls to allocate and deallocate memory are race conditions
- *In general, all kernel tables that can be accessed by system calls are potential sources of race conditions*
- If a kernel is nonpreemptive, meaning kernel processes are never preempted, it can have NO race conditions, since ony one kernel process executes at a time.
- Preemptive kernels can have races and must be designed to prevent them.



modified by Stewart Weiss

False Software Solution

- Consider this code for two concurrent processes: assume turn = 0 to start

```
P1: loop                                P2: loop
  1      while turn != 0 do;             5      while turn != 1 do;
  2      CS;                             6      CS;
  3      turn = 1;                       7      turn = 0;
  4     NCS;                             8     NCS;
    end loop                                end loop
```

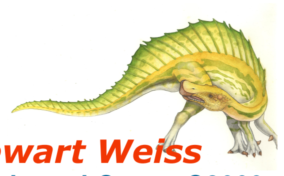
- Is it a solution?
- Violates ***progress***: a process in its entry section may be forced to wait there even though other process is not in its CS (in its NCS)

  e.g. P2 in entry, P1 in NCS, turn == 0 because P2 just left CS

  sequence is ...1,2,3,5,6,7,4,8,5 (turn == 0)
- The CS is like a key to an apartment; the person who has it locked the door, went on vacation, and the person who needs the key cannot get in.

Synchronization Hardware

- There are software solutions, but they are slow, complex.
- Hardware support for critical section code is faster
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
    - process on different processor can still access shared resource even if interrupts are disabled, so all processors would have to have interrupts disabled – inefficient and slow
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - test&set: test memory word and set value
  - swap: swap contents of two memory words



modified by Stewart Weiss

Solution to Critical-section Problem Using Locks

- Conceptually, we need a lock that can be unlocked and locked

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

TestAndSet Instruction

- Test and Set can be used to implement a lock;
- Definition of hardware functionality as a software function:

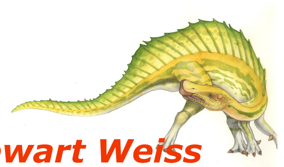
```
boolean TestAndSet (boolean *target)
{
     boolean rv = *target;
     *target = TRUE;
     return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
 loop {
       while ( TestAndSet (&lock ))
             /* do nothing */;
       criticalsection;
       lock = FALSE;
        remainder section
   }
```

Satisfies mutual exclusion and progress, but not bounded waiting. Why?



modified by Stewart Weiss

Swap Instruction

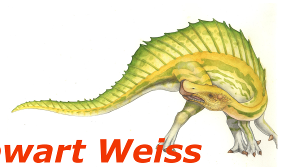
- Another atomic hardware operation is Swap.
- Definition:

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

Solution using Swap

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

```
 loop {
     key = TRUE;
     while ( key == TRUE)
         Swap (&lock, &key );
     critical section
     lock = FALSE;
     remainder section
    }
```



modified by Stewart Weiss

Bounded-waiting Mutual Exclusion with TestAndSet()

- The previous faux-solution using test-and-set did not satisfy bounded waiting. The next one is a bona-fide solution, satisfying all three constraints, and works for an arbitrary number of processes, not just two.
- Assume n is number of processes.
- It uses the two global objects:

```
boolean waiting[n];
boolean lock;
```

- Initially everything is set to false.
-

modified by Stewart Weiss

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
     waiting[i] = TRUE;
     key = TRUE;
     while (waiting[i] && key)
         key = TestAndSet(&lock); // entry section
     waiting[i] = FALSE;
     critical section
     j = (i + 1) % n;
     while ((j != i) && !waiting[j]) //exit section
          j = (j + 1) % n;           // find  j s.t. waiting[j] true
     if (j == i)                      // waiting[j] false for all j
         lock = FALSE;
     else                             // found a waiting Pj
         waiting[j] = FALSE;         // make false so waiting Pj can
      remainder section              // enter

} while (TRUE);
```

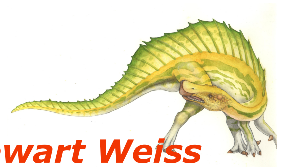
modified by Stewart Weiss

Proof of Correctness

- To prove that a potential solution is correct, we have to prove that mutual exclusion, progress, and bounded waiting are all satisfied.
- Some observations:
  - waiting[i] is true only if Pi is in entry section
  - once Pi gets CS, waiting[i] is false
  - only a process in its CS can change waiting[j] to false for some other process, and only one waiting[j] can be false at a time

Proof of Correctness: Mutual Exclusion

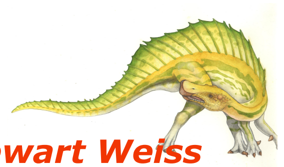
- Mutual exclusion is satisfied – suppose process `i` is in the CS.
- This can be argued inductively on how many processes have entered the CS so far.
- Base case is 0. Suppose no processes have ever entered CS. Only one will acquire `lock` because `TestAndSet` sets lock to true, so the first process to get `lock` and enter CS "locks the door behind it".
- Assume it is true for N and suppose that in the N+1st time, there are multiple processes competing to enter the CS. If `lock` is false, because last process left CS and set `lock` to false. In this case only one process gets `lock` and gets into CS.
- If `lock` is true, the only way that another process `j` can get into the CS is if `waiting[j]` is false. But if process `j` is in its entry section, it set `waiting[j]` to true before starting. When a process exits CS it sets `waiting[j]` to false for at most one process `j`, so at most one process will enter the CS, proving it is true in this case also.



modified by Stewart Weiss

Proof of Correctness: Progress

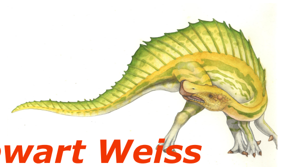
- ***Progress*** is satisfied – process `i` gets into the CS only if `lock` is false or `waiting[i]` is false.
- Suppose one or more processes are trying to get into the CS. There are two cases: either no process has ever entered the CS or at least one has.
- In the first case, `lock` is false and the first process to do `TestAndSet` gets into the CS.
- In the second case, the last process in the CS set `lock` to false or set `waiting[j]` to false for some `j` that is trying to enter the CS (why?), so exactly one of the processes in the entry section will get into the CS and the decision is determined by processes in the entry or exit sections alone.



modified by Stewart Weiss

Proof of Correctness: Bounded-waiting

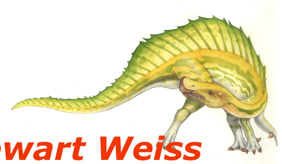
- ***Bounded waiting*** is satisfied – suppose process **`k`** is trying to enter the CS. Need to show that there is a bound on the number of processes that get into the CS before **`k`**.
- Each time a process **`i`** enters and exits CS, it sets **`waiting[j]`** to false for a process that is in the entry section, in the order **`[i+1,i+2,...,n-1,0,1,2,...,i-1]`**. I.e., it sets **`waiting[j]`** to false for first **`j`** it finds in this sequence for which **`waiting[j]`** is true.
- Since each process that enters searches in this same order, eventually, process **`k`** moves closer to the beginning of this sequence and eventually is the first for which **`waiting[j]`** is true. It then gets CS. ***So it waits at most n-1 times.***



Semaphores

- Problem with previous solutions is busy-waiting: spinning in a loop on CPU waiting to enter CS – wastes CPU cycles, is not easily generalized, and prone to errors in programming also
- Semaphores overcome these deficiencies
- Semaphore *S* – integer variable with two ***atomic operations***
  - wait() and signal()
  - originally called P() and V()

- `wait(S)` is equivalent to: `{ wait until S > 0;  S--; }` executed atomically
- `signal(S)` is equivalent to: `{ S++; }` executed atomically

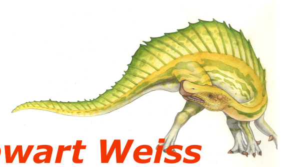


Using Semaphores for CS Problem

- Two kinds of semaphores: ***counting*** and ***binary***.
- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - can be simpler to implement
  - are also known as ***mutex locks***
- Can implement a counting semaphore S using a binary semaphore
- Semaphores provide mutual exclusion for n-process CS problem:

```
Semaphore mutex = 1;
loop {
    wait (mutex);
    Critical Section
    signal (mutex);
    remainder section
};
```

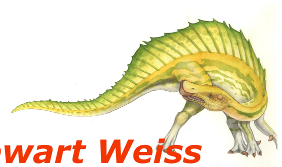
- Does it satisfy progress and bounded waiting?



Using Semaphores for Synchronization

- Suppose P1 qnd P2 are processes and that S1 in P1 must be executed before S2 in P2. a semaphore can synchronize them by setting it to 0 initially:

```
                   Semaphore synch = 0;
P1: {                                     P2: {
     S1;                                       wait(synch);
     signal (synch);                           S2;
     .
};
```



Semaphore Implementation

- An actual implementation of semaphores must not use busy waiting and spinlocks. It is inefficient to busy-wait on a spinlock
- Instead it should block processes that issue wait until S > 0, and unblock them when they acquire the semaphore.
- Must also 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.

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.)

- Implementation of wait:

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

- Implementation of signal:

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

- ***This implementation does not satisfy progress; bounded waiting depends on the queue discipline for the S-list.***

modified by Stewart Weiss

Correct Semaphore Implementation

- An implementation of semaphores that satisfies all three conditions is possible. It requires small busy-waits on two different spin-locks. It forces an alternation of P() and V() operations so that V() do not starve. The implementation is shown in the next two slides.
- Global variables:

```
boolean sem_lock = FALSE;
boolean delay    = TRUE;
Semaphore   S    = N; // # of processes allowed in CS
```

Correct Semaphore Implementation

- Implementation of wait:

```
wait(semaphore *S) {
    int sem;
    disable interrupts
    while (TestAndSet(sem_lock)  /*do nothing*/;
    S->value--;
    if (S->value < 0) {
        sem_lock = FALSE;
        while (TestAndSet(delay)) /*do nothing*/;
        put process on S->list
    }
    sem_lock = FALSE;
    enable interrupts;
}
```

modified by Stewart Weiss

Correct Semaphore Implementation

- Implementation of signal:

```
signal(semaphore *S) {
    int sem;
    disable interrupts
    while (TestAndSet(sem_lock)  /*do nothing*/;
    S->value++;
    if (S->value <= 0) {
        delay = FALSE;
        remove a  process from S->list
        enable interrupts;
    } else {
        sem_lock = FALSE;
        enable interrupts;
    }
}
```

modified by Stewart Weiss

Deadlock

- ***Deadlock*** is defined as a state of a set of processes in which two or more processes are waiting for an event that can only be caused by one of the waiting processes.
- This implies that all processes involved in the deadlock will wait forever.
- 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);
```

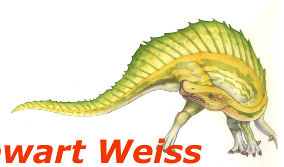
- If P0 gets S and then P1 gets Q, then P0 will wait for Q and P1 will wait for S, so they are deadlocked.

modified by Stewart Weiss

Starvation and Priority Inversion

- Starvation – is also called ***indefinite blocking***. This occurs when a process that is otherwise capable of running never acquires a resource for which it waits, even though the resource is available intermittently.
- This can happen, for example, when a low priority process remains in a queue waiting for a resource, but higher priority processes keep arriving and acquiring the resource ahead of it. The low priority process remains in the queue indefinitely.

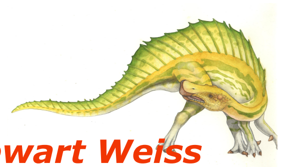
- Priority Inversion - This occurs when a lower priority process prevents a high priority process from acquiring a resource.
- Consider this scenario: 3 processes, L, M, H with priorities in the order L < M < H. L holds a resource R needed by H. L is preempted by M on processor. M runs, preventing H from getting R. If a stream of processes with priority M repeatedly preempt L, H may never acquire R.
- Solution is priority inheritance: low priority process temporarily gets priority of higher priority process if one is waiting for resource.



modified by Stewart Weiss

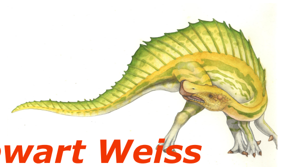
Classical Problems of Synchronization

- Bounded-Buffer Producer Consumer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem (skipping this one)



Bounded-Buffer Problem

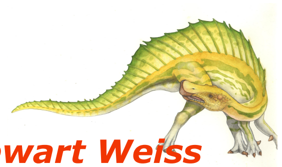
- *N* buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value N.
- **`full`** is a semaphore that counts number of full buffers, and **`empty`** is a semaphore that counts empty buffers. **`mutex`** is a semephore that provides mutual exclusion on the access to the buffer itself



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do  {
     //   produce an item in nextp
     wait (empty);
     wait (mutex);
     //  add the item to the  buffer
     signal (mutex);
     signal (full);
     } while (TRUE);
```



modified by Stewart Weiss

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

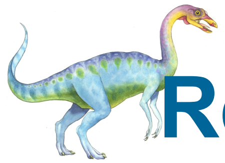
```
do   {
     wait (full);
     wait (mutex);
     // remove  item from buffer into nextc
     signal (mutex);
     signal (empty);
     // consume item in nextc
     } while (TRUE);
```

modified by Stewart Weiss

Readers-Writers Problem

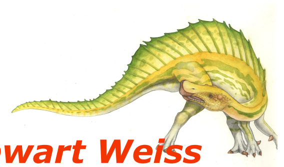
- A data set is shared among a number of concurrent processes.
  - Some only read the data, called ***readers***
  - Some can read and write the data, called ***writers***
- Problem – how to allow multiple readers to read at the same time, and only allow one single writer to access the shared data at a time. Called the ***readers/writers problem***

modified by Stewart Weiss

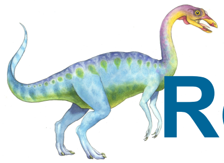


Readers-Writers Problem Variations

- There are various versions of this problem, depending on amount of preference given to readers versus writers. The possible situations are:
  - If a reader is reading and a reader arrives, and no writer is waiting, the reader immediately reads, so there is no question in this case.
  - If a reader is reading and a writer arrives (and thus has to wait), should any readers that arrive afterward wait until the writer writes, or should arriving readers keep entering, so that the writer has to wait until no readers are waiting?
  - If there is a writer writing and a reader and writer are both waiting, which gets to go first?

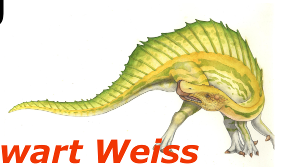


modified by Stewart Weiss



Readers-Writers Problem Variations

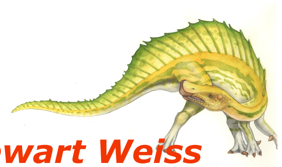
- Solutions:
- ***Writer Priority***
  - Waiting writers have priority over waiting readers, regardless of which is currently accessing the data. (This implies that a waiting writer starts as soon as the reader or writer currently accessing the data finishes, and that arriving readers wait until no active writers or waiting writers remain.)
- ***Weak Reader Priority***
  - If a reader is reading the data, waiting readers have priority over waiting writers. (This implies that a writer has to wait until no readers are reading or waiting.) If a writer is writing, no preference is given to either readers or writers.
- ***Strong Reader Priority***
  - No matter which kind of process is accessing the data, waiting readers have priority over waiting writers and arriving writers wait until no active readers remain.



modified by Stewart Weiss

Readers-Writers and Starvation

- All of these variations of the readers-writers problems can result in starvation:
- Writer Priority
  - A steady stream of writers can starve a reader.
- Weak Reader Priority:
  - A steady stream of readers can starve a writer.
- Strong Reader Priority
  - A steady stream of readers can starve a writer.
- There are variations that are starvation-free.



Weak Reader Priority Solution

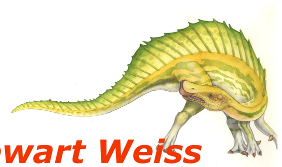
- A solution using semaphores to the weak reader priority version of the problem
- Idea: readers allow readers to share, but set a semaphore that blocks writers only.
- Shared Data:
  - Data set
  - Semaphore **`mutex`** initialized to 1, controls access to readcount
  - Semaphore **`wrt`** initialized to 1, controls access to data
  - Integer **`readcount`** initialized to 0

modified by Stewart Weiss

Writer Process Structure

- The structure of a writer process

```
do {
    wait (wrt) ;
    // write data
    signal (wrt) ;
} while (TRUE);
```



modified by Stewart Weiss

Reader Process Structure

- The structure of a reader process

```
do {
     wait (mutex) ;       // mutex to update readcount
     readcount ++ ;
     if (readcount == 1) // first reader
          wait (wrt);     // block all writers
     signal (mutex);     // release mutex
     // read data
     wait (mutex) ;       // mutex to update readcount
     readcount--;
     if(readcount  == 0) // last reader
          signal (wrt) ;  // allow writer to enter
     signal (mutex) ;    // release mutex
 } while (TRUE);
```

modified by Stewart Weiss

Weak Reader Priority Observations

- The first reader issues the wait on `wrt`, so that no writers can enter. Subsequent readers skip this step and therefore are not blocked on `wrt`.
- In this solution, a steady stream of readers may block a writer.
- If a reader and writer are both waiting on semaphore `wrt` and the writing writer finishes and signals `wrt`, either the reader or writer may get access next, depending on how the semaphore is implemented.

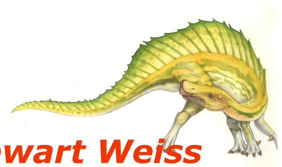
Reader-Writer Locks

- The solutions to the readers-writers problem have been generalized to create read-writer locks in some systems.
- A reader-writer lock can be used to control access to shared data. A process requests the lock, specifying whether it is read or write access. If read access, other processes are permitted to acquire the lock for reading. If write-access it waits until no readers or writers are active and then acquires access for writing.
- Useful when there are many readers and few writers.
- Java provides reader-writer locks in its concurrent library.

modified by Stewart Weiss

Alternatives to Semaphores

- Semaphores are a low-level primitive for synchronization; their correct use requires programmer discipline.
- Can be used incorrectly – e.g. reversing the order of signal and wait, or creating deadlock situations we saw in earlier slide.
- Higher level constructs for synchronization exist in some programming languages
- Monitors are one such construct (but will not be covered here)
- See documentation for Java, which has them



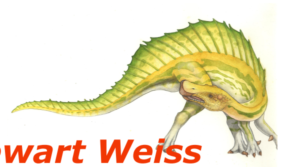
modified by Stewart Weiss

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Within the kernel itself, four different mechanisms are used:
  - Uses adaptive mutexes for efficiency when protecting data from short code segments
  - Uses condition variables and readers-writers locks when longer sections of code need access to data
  - Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Solaris Adaptive Mutexes

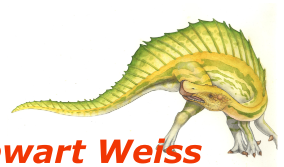
- An ***adaptive mutex*** is initially like a semaphore implemented as a spinlock.
  - If a thread tries to acquire the mutex but it is already locked by another thread that is running, the thread busy-waits in the spinlock.
  - If the thread holding the mutex is not running, the second thread blocks instead and is awakened by the release of the mutex.
  - On a single processor system, the thread holding the mutex cannot be running, so the second thread always blocks.
- Adaptive mutexes are only used to protect short critical sections (< few hundred instructions) because spin-waiting is not efficient for longer than that.
- For longer code sections, condition variables and real semaphores (with blocking) are used.



modified by Stewart Weiss

Solaris Turnstiles

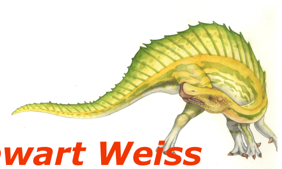
- A ***turnstile*** is a queue of kernel threads blocked on an adaptive-mutex or read-writer lock, but rather than being attached to the lock initially, it is attached to a kernel thread.
  - When a kernel thread blocks on a lock, its turnstile turns into a queue for the lock itself.
  - All threads that block on that lock are queued on that turnstile.
  - When the initial kernel thread is released, it acquires a new turnstile from the pool of free turnstiles.
  - Solaris prevents priority inversion using a priority inheritance protocol – if lower priority thread holds a lock on which a higher priority thread blocks, the low priority thread gets the priority of the higher one.
- Same locking methods are available to user threads, but not priority inheritance on turnstiles.



modified by Stewart Weiss

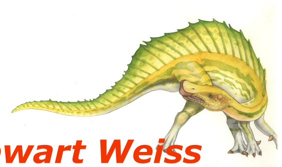
Linux Synchronization

- Linux prior to 2.6 was non-preemptive.
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
  - Uses spinlocks in the kernel for multiprocessor machines; does not use spinlocks on single processor machines -- instead disables and enables preemption for single processor machines within kernel.
- Linux provides:
  - semaphores for long critical sections
  - spin locks for short ones



Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks



modified by Stewart Weiss

End of Chapter 6