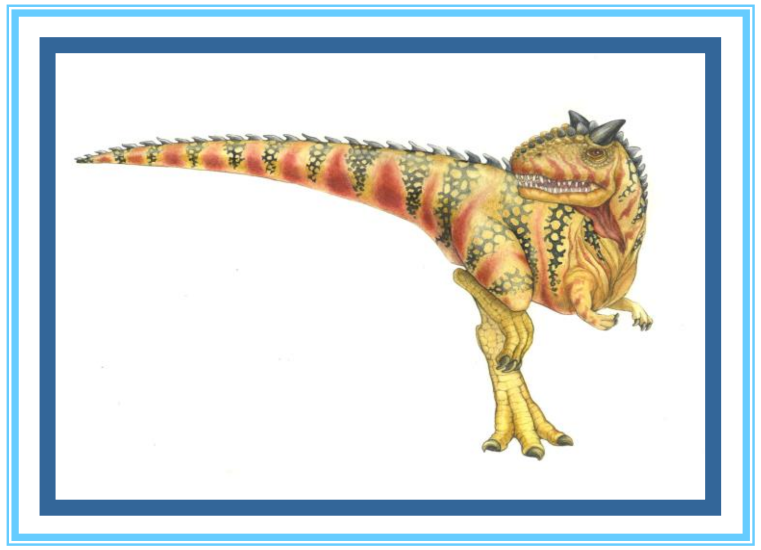
Chapter 6: Process Synchronization



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

Chapter 6: Process Synchronization

Background	
The Critical-Section Problem	临界区问题
Peterson’s Solution	
Synchronization Hardware	硬件同步
Mutex Locks	互斥锁
Semaphores	信号量
Classic Problems of Synchronization	
Monitors	管程
Synchronization Examples	
Alternative Approaches	

BACKGROUND

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency 不一致
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background

- 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

```
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

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

Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

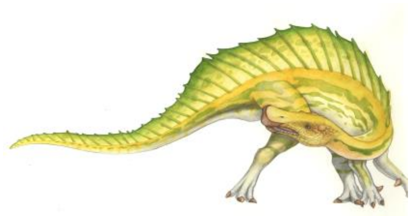
```
register2 = counter
register2 = register2 - 1
counter = register2
```

Race Condition

- Consider this execution interleaving with "count = 5" initially:

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



Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** *临界区问题* is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

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 互斥** - If process $\boldsymbol{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress 前进** - 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

Solution to Critical-Section Problem

3. **Bounded Waiting 有限等待** - 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

Solution to Critical-Section Problem

- 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

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- 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!

Algorithm for Process P_i

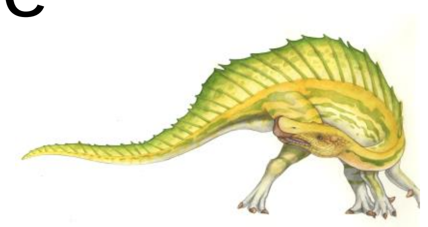
```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

- 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



Synchronization Hardware

- Modern machines provide special atomic hardware instructions
    - **Atomic 原子** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

- Definition:

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

Solution using test_and_set()

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

```
do {
   while (test_and_set(&lock))
         ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

swap Instruction

- 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:

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

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    …
```

Bounded-waiting Mutual Exclusion with test_and_set

```
       /* critical section */
      j = (i + 1) % n;
      while ((j != i) && !waiting[j])
         j = (j + 1) % n;
      if (j == i)
         lock = false;
      else
         waiting[j] = false;
      /* remainder section */
   } while (true);
```

SEMAPHORE

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

Mutex Locks

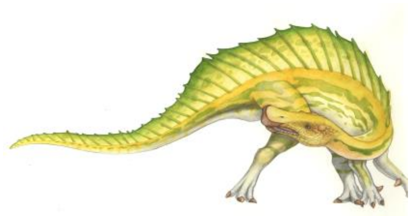
- Product critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock 自旋锁**

acquire() and release()

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

release() {
    available = true;
}
```

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



Semaphore

- **Binary semaphore 二进制信号量** – integer value can range only between 0 and 1
  - Then a **mutex lock**
- **Counting semaphore 计数信号量** – integer value can range over an unrestricted domain
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems

Semaphore

- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**

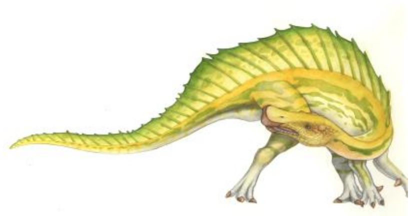
- Can only be accessed via two indivisible (atomic) operations

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

Semaphore Usage

- 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 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

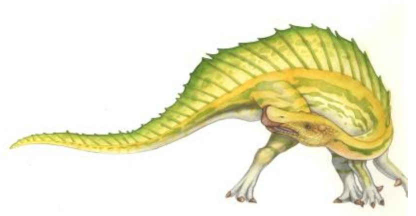
Semaphore Implementation with no Busy waiting (Cont.)

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

Semaphore Implementation with no Busy waiting (Cont.)

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

DEADLOCK AND STARVATION



Deadlock and Starvation

- **Deadlock 死锁** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- 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);
```

Deadlock and Starvation

- **Starvation** – **indefinite blocking 无限期阻塞**
  - A process may never be removed from the semaphore queue in which it is suspended

CLASSICAL PROBLEM OF SYNCHRONIZATION

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem 有限缓冲问题
  - Readers and Writers Problem 读者-写者问题
  - Dining-Philosophers Problem 哲学家进餐问题

BOUNDED BUFFER PROBLEM

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

```
do {
   /* produce an item in next_produced */
   wait(empty);
   wait(mutex);
   /* add next produced to the buffer */
   signal(mutex);
   signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

READERS-WRITERS PROBLEM

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

Readers-Writers Problem Variations

- Several variations of how readers and writers are treated – all involve priorities
- ***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

Readers-Writers Problem (cont.)

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

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

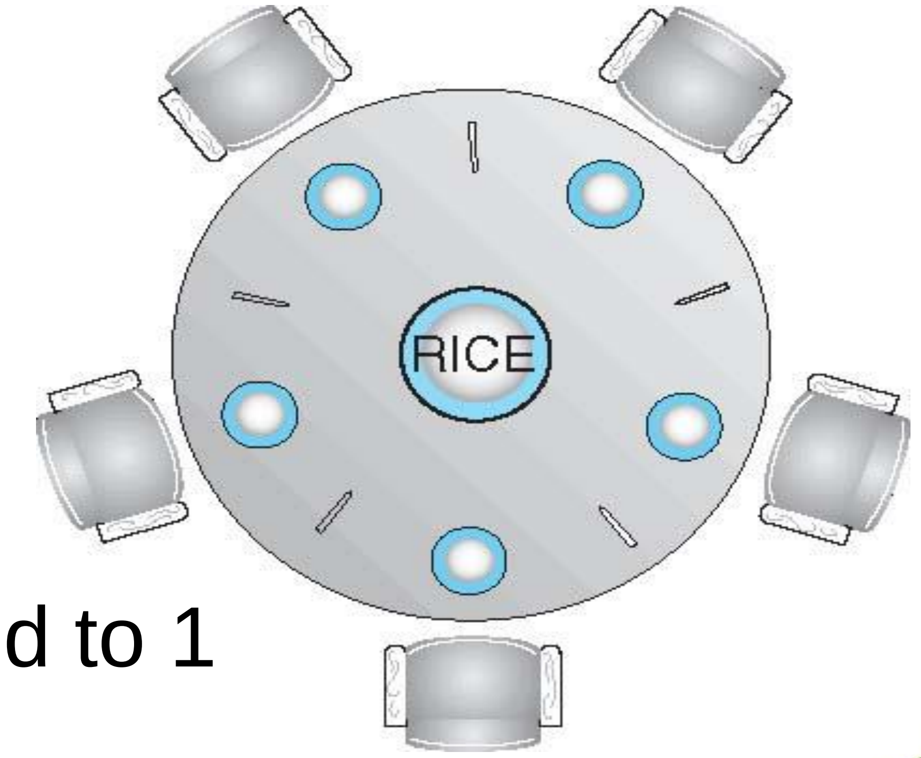
```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
     wait(wrt);
   signal(mutex);
          ...
/* reading is performed */

     ...
   wait(mutex);
   read_count--;
   if (read_count == 0)
     signal(wrt);
   signal(mutex);
} while (true);
```

DINING-PHILOSOPHERS PROBLEM

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?

MONITORS AND CONDITIONAL VARIABLES

Monitors

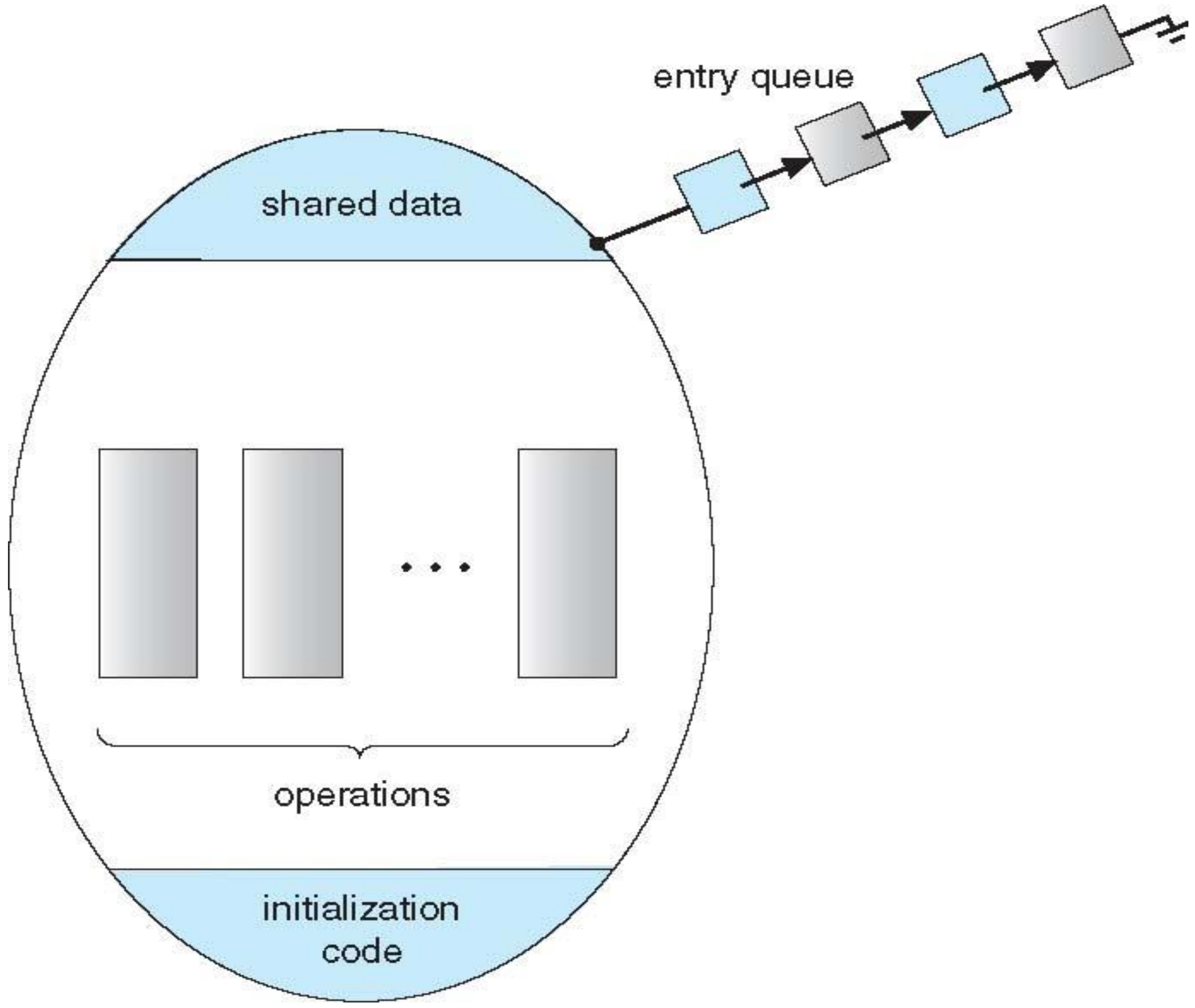
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

Monitors

- But not powerful enough to model some synchronization schemes

```
monitor monitor-name{
  // shared variable declarations
  procedure P1 (…) { …. }
  procedure Pn (…) {……}
  Initialization code (…) { … }
}
```

Schematic 示意 view of a Monitor



Solution to Dining Philosophers (Cont.)

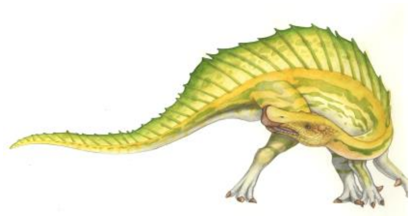
- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

EAT

DiningPhilosophers.putdown(i);
```

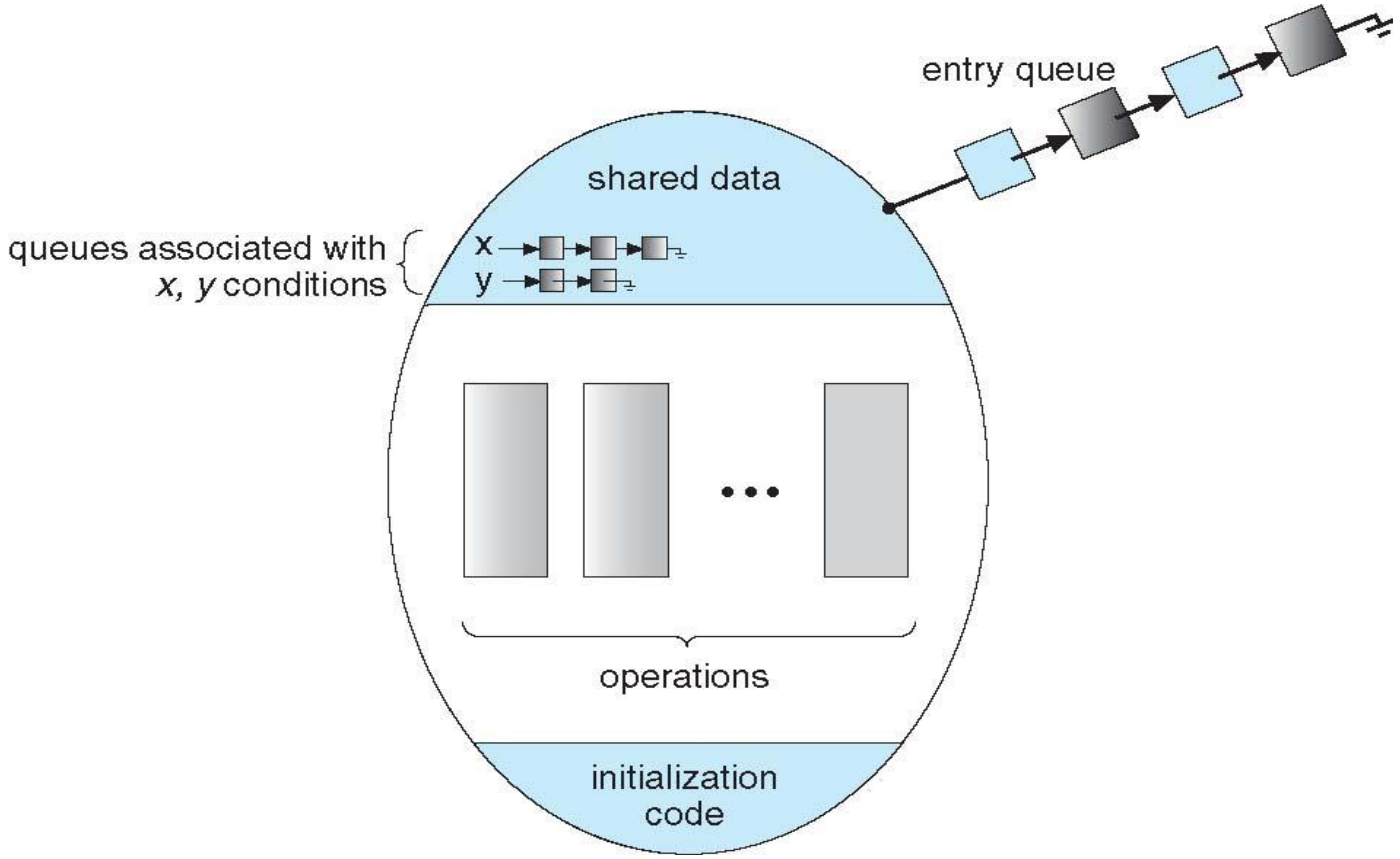
- No deadlock, but starvation is possible



Condition Variables

- `condition x, y;`
- Two operations on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

Monitor with Condition Variables



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;
  }
    …
```

Solution to Dining Philosophers (cont.)

```
void putdown (int i) {
   state[i] = THINKING;
   // test left and right neighbors
   test((i + 4) % 5);
   test((i + 1) % 5);
}
   …
```

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 () ;
  }
}
 …
```

Solution to Dining Philosophers (Cont.)

```
initialization_code() {
  for (int i = 0; i < 5; i++)
    state[i] = THINKING;
  }
}
```

Condition Variables Choices

- If process P invokes **`x.signal()`** , with Q in **`x.wait()`** state, what should happen next?
  - If Q is resumed, then P must wait

Condition Variables Choices

- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Implementation Using Semaphores

- Variables

```
        semaphore mutex;  // (initially  = 1)
        semaphore next;   // (initially  = 0)
        int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
                wait(mutex);
                   …
                body of F;
                   …
                if (next_count > 0)
                   signal(next)
                else
                   signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation (Cont.)

- The operation **`x.signal`** can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait 条件等待** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
  boolean busy;
  condition x;
  void acquire(int time) {
    if (busy)
    x.wait(time);
    busy = TRUE;
  }
  …
```

```
  void release() {
    busy = FALSE;
    x.signal();
  }

  initialization code() {
    busy = FALSE;
  }
}
```

SYNCHRONIZATION EXAMPLES

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes 适应互斥** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

Solaris Synchronization

- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles 十字转门** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

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

Windows XP Synchronization

- Also provides **dispatcher objects 调度对象** 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

Pthreads Synchronization

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

ATOMIC TRANSACTION

Atomic Transactions 原子业务

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures

System Model

- **Transaction** - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of **read** and **write** operations
  - Terminated by **commit** 提交 (transaction successful) or **abort** 撤销 (transaction failed) operation
  - Aborted transaction must be **rolled back** 回退 to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is **write-ahead logging**
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - **Undo(T_i)** restores value of all data updated by T_i
  - **Redo(T_i)** sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be **idempotent 幂等**
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, **undo(T_i)**
  - If log contains <T_i starts> and <T_i commits>, **redo(T_i)**

Checkpoints

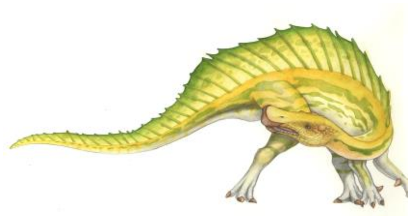
- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the 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

Concurrent Transactions

- Must be equivalent to serial execution – **serializability 串行化**
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- **Concurrency-control algorithms 并发控制算法** provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules



Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

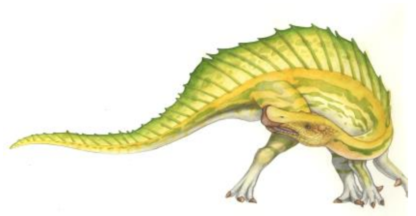
- **Nonserial schedule** allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - **Conflict** if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is **conflict serializable**

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - **Shared** – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - **Exclusive** – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

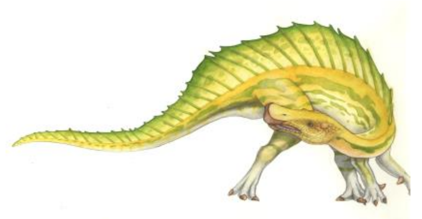


Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – **timestamp-ordering**
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j



Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed

Timestamp-based Protocol Implementation

- **Timestamp-ordering protocol** assures any conflicting **read** and **write** executed in timestamp order
- Suppose Ti executes **read(Q)**
  - If $TS(T_i) <$ W-timestamp(Q), Ti needs to read value of Q that was already overwritten
    - **read** operation rejected and T_i rolled back
  - If $TS(T_i) \geq$ W-timestamp(Q)
    - **read** executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If TS(T_i) < R-timestamp(Q), value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - **Write** operation rejected, T_i rolled back
  - If TS(T_i) < W-timestamp(Q), T_i attempting to write obsolete value of Q
    - **Write** operation rejected and T_i rolled back
  - Otherwise, **write** executed

Timestamp-ordering Protocol

- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

The Sleeping Barber Problem

- N chairs
- 1 barber
- The barber: Sleep / Cut hair
- The customers: Awake the barber / Cut hair / Wait / Leave

End of Chapter 6