Chapter 6: Process Synchronization

Module 6: Process Synchronization

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

Background: Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
    - Consumer may have to wait
    - Producer can always produce new item
  - *bounded-buffer* assumes that there is a fixed buffer size

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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 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

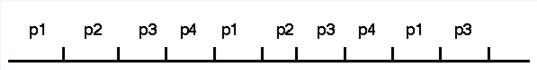
```
while (true) {

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


Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
        nextConsumed =  buffer[out];
         out = (out + 1) % BUFFER_SIZE;
         count--;

                    /*  consume the item in nextConsumed
 }
```

Processus concurrents

- Les processus sont exécutés en parallèle



- Les commutations sont indépendantes du programme des processus
- On **ne peut pas (et doit pas)** faire d'hypothèse sur l'ordre relatif des exécutions
- Seuls comptent
  - L'ordre d'exécution interne d'un processus
  - Les relations logiques entre les processus (synchronisation)

Race Condition

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:

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

Incorrect state

- Counter == 4 whereas 5 buffers are full
- Reversing statement S4 and S5 also gives an incorrect state
  - Counter == 6
- Both processes are allowed to manipulate the variable counter concurrently
- If the outcome of the execution depends on the particular order in which the access take place by several processes is called a **race condition**

Sections critiques et actions atomiques

- Comment protéger les accès aux variables partagées
  - Assurer qu'un ensemble d'opérations sont exécuté de manière indivisible (atomique)

processus p1

A1
```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2

A2
```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

- Si A1 et A2 sont atomiques
  - Les seules exécutions possibles sont
    - A1; A2 ou A2; A1
- Section critique
  - Un ensemble d'opérations qui ne doit pas être exécuté de façon concurrente
- Exclusion mutuelle
  - Permettre un accès exclusif à un ensemble d'instructions

Section critique

déclaration et initialisation de variables communes

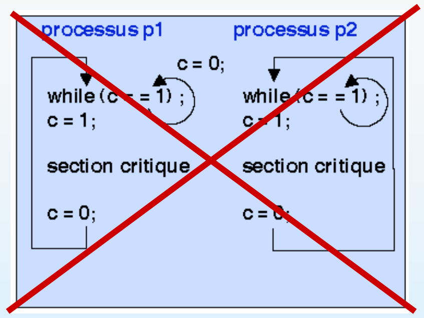
processus p1	processus p2
...	...
entrée en section critique	entrée en section critique
section critique	section critique
sortie de section critique	sortie de section critique
...	...

- Les opérations « entrée en section critique » et « sortie de section critique » doivent garantir l’exclusion mutuelle

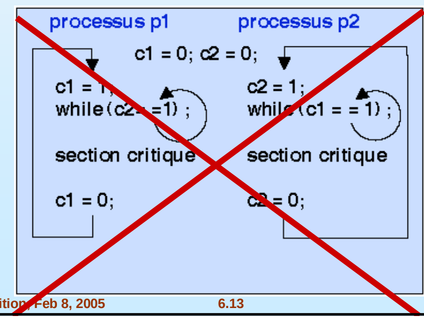
Réalisation d’une section critique

- Attente active
  - Le processus qui attend la section critique boucle sur le test d’entrée
    - Méthode très inefficace
    - Fonctionne s’il n’y a qu’un processeurs
    - Utiliser parfois en mode noyau pour de très courtes sections
- Primitives spéciales
  - Elles doivent être atomiques

Réalisation d'une section critique



Try again …



Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - **No assumption** concerning relative speed of the N processes

Preemptive / non preemptive

- Non preemptive kernel
  - Does not allow a process running in kernel mode to be preempted
  - A kernel mode process will run until
    - it exits kernel mode
    - Blocks
    - Yield the CPU
  - → free from RACE condition on kernel data structure
- Preemptive kernel
  - Allow a process to be preempted while it is running in kernel mode
  - → should be carefully designed
  - Especially difficult to design on SMP architecture
    - 2 kernel mode process could run simultaneously

Why anyone would favor preemptive over non preemptive?

- Preemptive → more suitable for real time
  - "real" time process is able to preempt a process currently running in the kernel
  - Preemptive kernel may be more responsive
    - Less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the CPU to waiting processes
  - XP/2000 are non preemptive
  - > Linux 2.6 preemptive as Solaris and IRIX

Peterson's Solution

- Software based solution
- Two process solution P_0 and P_1
- 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

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

        // CRITICAL SECTION

        flag[i] = FALSE;

        // REMAINDER SECTION

  }
```

- Process P_i first set flag to TRUE
- And set turn to j
- → asserting that if the other process whishes to enter the Critical Section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time
- Only one of these assignment will last

Proof

- Mutual exclusion is preserved
- Progress requirement is satisfied
- Bounded waiting time is met

Proof

- Mutual exclusion is preserved
  - P_i enters its critical section only if flag[j]==false OR turn==i
  - If both processes are in critical then
    - Flag[0]==flag[1]==TRUE
  - P_0 and P1 can not have successfully executed their while at the same time
    - Since turn is either 0 or 1 but not both
  - P_i did and P_j not
  - At that time flag[j]==true AND turn==j
    - This condition will persist as long as P_j is in its critical section
- MUTUAL exclusion is preserved

Proof

- Progress requirement is satisfied && Bounded waiting time is met
  - P_i can be prevented from entering only if
    - Flag[j]==true and turn==j (while loop condition / only loop)
  - If P_j is not ready to enter
    - Flag[j]==false and P_i can enter
  - If P_j has set flag[j] to TRUE and it is in the loop then
    - Either turn == I or turn == j
  - If turn == i the P_i will enter
  - If turn == j then P_j will enter
- Once P_j exists
  - It will set reset flag[j] to true AND set turn to I
- Since P_i does not change the value of turn during the loop
  - P_i will enter (progress) after at most one entry by P_j (bounded waiting)

Synchronization Hardware

- We need a "**lock**"

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
    - Disabling interrupts is time consuming in MultiProc
    - Bad effect on a system clock updated by interrupts

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

TestAndSet Instruction

- Definition:

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


Solution using TestAndSet

- Shared boolean variable lock., initialized to false.

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

 }
```

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.

```
while (true) {
            key = TRUE;
            while ( key == TRUE)
                   Swap (&lock, &key );

                      //    critical section

             lock = FALSE;

                     //     remainder section

     }
```

Comments

- Algorithms satisfy the mutual-exclusion
- But do not satisfy the bounded-waiting time

Bounded waiting mutual exclusion with TestAndSet

- Shared boolean variable key and boolean array waiting

```
do {
   waiting[i] = TRUE
   key = TRUE
   while (waiting[i] && key)
        key = TestAndSet(&lock)
   waiting[i] = FALSE
   // 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

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V() ***(Puis-je ? / Vas-y !)***
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - ```
    wait (S) {
         while S <= 0
              ; // no-op
         S--;
    }
    ```
  - ```
    signal (S) {
         S++;
    }
    ```
- **When one process modifies the semaphore, no other process can simultaneously modify the same semaphore value**

Semaphore as General Synchronization Tool

- 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
  - Also known as mutex locks → lock that provides mutual exclusion
- Can use binary semaphore to deal with critical section
- Provides mutual exclusion

```
Semaphore S;    // initialized to 1
wait (S);
          Critical Section
signal (S);
```

Semaphore as General Synchronization Tool

- Counting semaphore used to control access to a given resource consisting of a finite number of instance
  - Semaphore is initialized to the number of resource
  - To use a resource, P should perform a wait()
  - To release a resource perform a signal()
  - Semaphore == 0 → all resources are used
- Synchronization
  - P_1 with statement S_1 and P_2 with statement S_2
  - S_2 be executed only after S_1 as completed

```
Semaphore synch initialized to 0
S_1;
Signal(synch)

Wait(synch)
S_2
```


Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Busy waiting wastes CPU cycles
- **Spinlock** semaphore == the process spins while waiting for the lock
  - Spinlock advantage == no context switch
  - Useful when the lock are expected to be held for short time
- 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

- To overcome the need of spinlock
  - → modify the wait and signal semaphore operations
- 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

```
Typedef struct {
        int value;
        struct process *list;
} semaphore
```

- 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 waiting queue S->list
            block();
        }
    }
```

- Implementation of signal:

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

Deadlock and Starvation

- Deadlock – 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

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

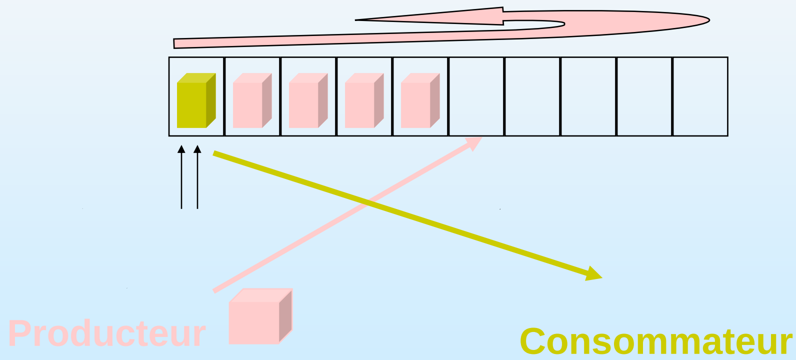
- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Producteurs / consommateurs

Tampon de messages géré circulairement



Problématique

- Plusieurs producteurs / plusieurs consommateurs
- Les producteurs remplissent un buffer
- Consommateurs le vident
- Problème
  - Une donnée doit être lue une seule fois
  - Une donnée ne doit pas être écrasée avant d’avoir été lue
  - Une case non remplie ne peux être lue

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

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);

                //  remove an item from  buffer

        signal (mutex);
        signal (empty);

                //  consume the removed item

}
```


Bounded Buffer Problem (Cont.)

- Symmetry between the producer and consumer
- Producer produces “full” buffer for the consumer
- Consumer produces “empty” buffer fir the producer

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.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

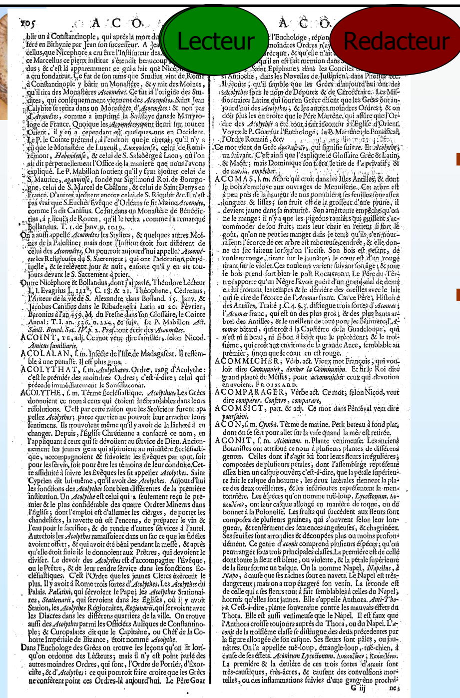
Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

                // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)  signal (wrt) ;
        signal (mutex) ;
}
```


Readers-Writers Problem (Cont.)

- If a writer is in the Critical section and N reader are waiting
  - → One reader is queued on WRT
  - → N - 1 readers are queued on MUTEX
- If a writer executes signal(WRT), we may resume the execution of
  - → either the waiting readers
  - Or a single waiting writers

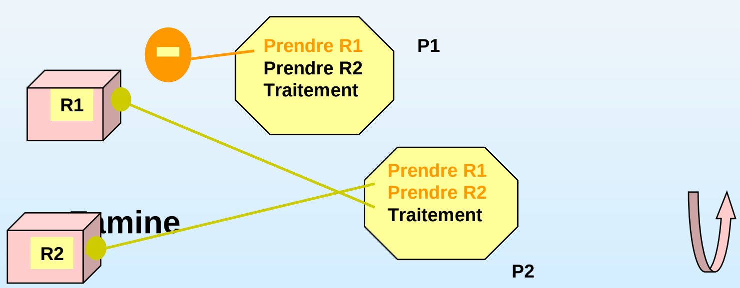


Scénario

- Le rédacteur doit attendre que tous les lecteurs aient fini
  - Mais comme le nombre de lecteurs n'est pas bornés…
- Un algorithme doit assurer une certaine équité
  - Eviter les cas de famine

Coalition

- **Ensemble de n processus monopolisant les ressources au détriment de p autres processus**



Une solution plus juste

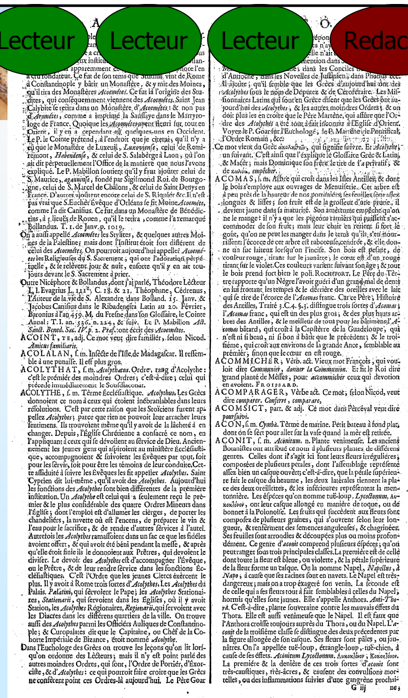
```
Init(MutexLecture, 1)
Init(LectEcr, 1)
Init(Ecriture, 1)
nblect = 0
```

```
// Lecteur

DébutLecture {
   P(LecEcr);
   P(MutexLecteur);
   nbLect := nbLect + 1;
   If (nbLect = 1) then P(Ecriture);
   V(MutexLecteur);
   V(LecEcr);
}

FinLecture {
   P(MutexLecteur);
   nbLect := nbLect - 1;
   if (nbLect = 0) then V(Ecriture);
   V(MutexLecteur);
}
```

```
// Redacteurs

DébutEcriture {
   P(LecEcr);
   P(Ecriture);
}

FinEcriture {
   V(Ecriture);
   V(LectEcr);
}
```


Un resultat plus juste

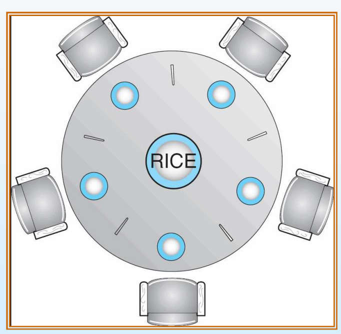
Lecteur Lecteur Lecteur Redacteur



Dîner des philosophes

- Problème
  - Quelques philosophes se retrouvent pour manger
    - Ils sont installés autour d'une table ronde
  - Un philosophe a besoin de deux baquettes pour manger
  - L'activité d'un philosophe consiste en :
    - Penser
    - Manger

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

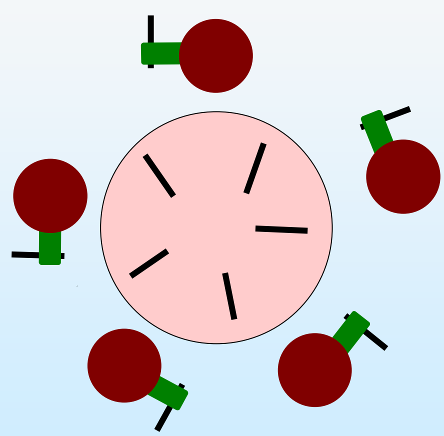
Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

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

              //  eat

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

             //  think

}
```


Problème

- Tous les philosophes peuvent détenir une baquette
- Pas d'autre baquette disponible
  - Aucun philosophe ne libérera sa baquette tant qu'il n'aura pas mangé
  - Tous les philosophes sont bloqués



Solutions

- At most 4 philosophers to be sitting simultaneously at the table
- Allow a philosopher to the chopsticks only if both chopsticks are available (in a critical section)
- Use asymmetric solution
  - Odd philosopher picks up first the left chopstick and then the right one.
- Any satisfactory solution MUST guard against the possibility that one philosopher will stave to death
- Deadlock free does not imply no starvation

Problems with Semaphores

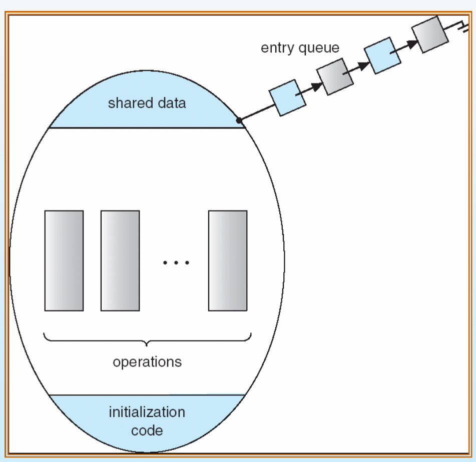
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

    procedure Pn (…) {……}

     Initialization code ( ….) { … }
             …
    }
}
```

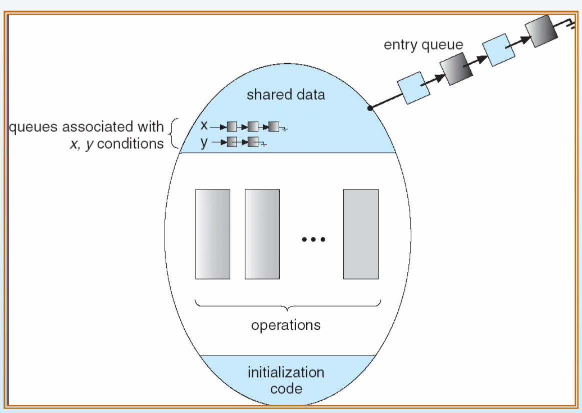

Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
  {
     enum { THINKING; HUNGRY, EATING) state [5] ;
     condition self [5];

     void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING) self [i].wait;
     }

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

      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:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```


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

- 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

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
     next-count++;
     signal(x-sem);
     wait(next);
     next-count--;
}
```

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 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

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

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

Atomic Transactions

- 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
- 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 (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back 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
  - 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
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms 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-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-tiimestamp(Q), T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

End of Chapter 6