Chapter 6 Synchronization

Process Synchronization

- Why Synchronization?
  - To ensure data consistency for concurrent access to shared data!
- Contents:
  - Various mechanisms to ensure the orderly execution of cooperating processes

Process Synchronization

- A Consumer-Producer Example

- Producer

```
while (1) {
    while (counter == BUFFER_SIZE)
        ;
    produce an item in nextp;
    ….
    buffer[in] = nextp;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

- Consumer:

```
while (1) {
    while (counter == 0)
        ;
    nextc = buffer[out];
    out = (out +1) % BUFFER_SIZE;
    counter--;
    consume an item in nextc;
}
```

Process Synchronization

- counter++ vs counter—

```
r1 = counter       r2 = counter
r1 = r1 + 1        r2 = r2 - 1
counter = r1       counter = r2
```

- Initially, let counter = 5.

1. P: r1 = counter
2. P: r1 = r1 + 1
3. C: r2 = counter
4. C: r2 = r2 – 1
5. P: counter = r1
6. C: counter = r2

⇒ A Race Condition!

Process Synchronization

- A Race Condition:
  - A situation where the outcome of the execution depends on the particular order of process scheduling.
- The Critical-Section Problem:
  - Design a protocol that processes can use to cooperate.
    - Each process has a segment of code, called a <u>critical section</u>, whose execution must be <u>mutually exclusive</u>.

Process Synchronization

- A General Structure for the Critical-Section Problem

```
                          do {
permission request  →       entry section;
                              critical section;
exit notification   →       exit section;
                              remainder section;
                          } while (1);
```

The Critical-Section Problem

- Three Requirements

1. Mutual Exclusion
   a. Only one process can be in its critical section.
2. Progress
   a. Only processes not in their remainder section can decide which will enter its critical section.
   b. The selection cannot be postponed indefinitely.
3. Bounded Waiting
   a. A waiting process only waits for a bounded number of processes to enter their critical sections.

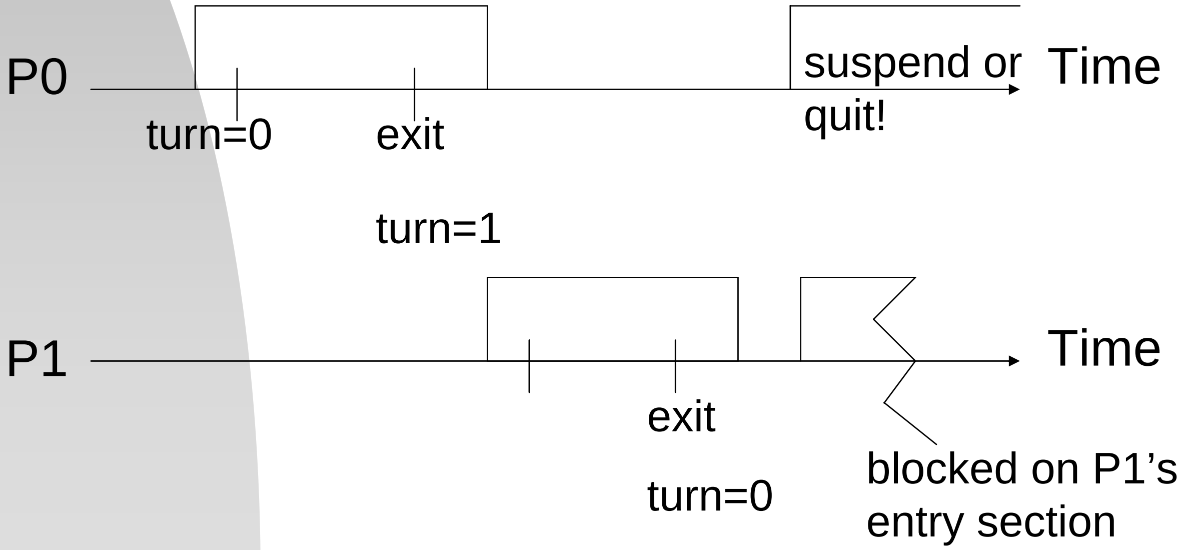
The Critical-Section Problem – Peterson's Solution

- Notation
  - Processes Pi and Pj, where j=1-i;
- Assumption
  - Every basic machine-language instruction is atomic.
- Algorithm 1
  - Idea: Remember which process is allowed to enter its critical section, That is, process i can enter its critical section if turn = i.

```
do {
   while (turn != i) ;
   critical section
   turn=j;
   remainder section
} while (1);
```

The Critical-Section Problem – Peterson’s Solution

- Algorithm 1 fails the progress requirement:



P0

turn=0

exit

turn=1

suspend or quit!

Time

P1

exit

turn=0

blocked on P1’s entry section

Time

The Critical-Section Problem – Peterson’s Solution

- Algorithm 2
  - Idea: Remember the state of each process.
  - flag[i]==true → Pi is ready to enter its critical section.
  - Algorithm 2 fails the progress requirement when flag[0]==flag[1]==true;
    - the exact timing of the two processes?

Initially, flag[0]=flag[1]=false

```
do {
    flag[i]=true;
    while (flag[j]) ;
    critical section
    flag[i]=false;
    remainder section
}  while (1);
```

* The switching of “flag[i]=true” and “while (flag[j]);”.

The Critical-Section Problem – Peterson’s Solution

- Algorithm 3
  - Idea: Combine the ideas of Algorithms 1 and 2
  - When (flag[i] && turn=i), Pj must wait.
  - Initially, flag[0]=flag[1]=false, and turn = 0 or 1

```
do {
    flag[i]=true;
    turn=j;
    while (flag[j] && turn==j) ;
    critical section
    flag[i]=false;
    remainder section
}  while (1);
```

The Critical-Section Problem – Peterson’s Solution

- Properties of Algorithm 3
  - Mutual Exclusion
    - The eventual value of *turn* determines which process enters the critical section.
  - Progress
    - A process can only be stuck in the while loop, and the process which can keep it waiting must be in its critical sections.
  - Bounded Waiting
    - Each process wait at most one entry by the other process.

The Critical-Section Problem – A Multiple-Process Solution

- Bakery Algorithm
  - Originally designed for distributed systems
  - Processes which are ready to enter their critical section must take a number and wait till the number becomes the lowest.
  - int number[i]: Pi’s number if it is nonzero.
  - boolean choosing[i]: Pi is taking a number.

The Critical-Section Problem – A Multiple-Process Solution

```
do {
      choosing[i]=true;
      number[i]=max(number[0], …number[n-1])+1;
      choosing[i]=false;
      for (j=0; j < n; j++)
         while choosing[j] ;
         while (number[j] != 0 && (number[j],j)<(number[i],i)) ;
      critical section
      number[i]=0;
      remainder section
  } while (1);
```

- An observation: If Pi is in its critical section, and Pk (k != i) has already chosen its number[k], then (number[i],i) < (number[k],k).

Synchronization Hardware

- Motivation:
  - Hardware features make programming easier and improve system efficiency.
- Approach:
  - Disable Interrupt → No Preemption
    - Infeasible in multiprocessor environment where message passing is used.
    - Potential impacts on interrupt-driven system clocks.
  - Atomic Hardware Instructions
    - Test-and-set, Swap, etc.

Synchronization Hardware

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target=true;
    return rv;
}
```

```
do {
    while (TestAndSet(lock)) ;
      critical section
    lock=false;
      remainder section
} while (1);
```

Synchronization Hardware

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

```
do {
      key=true;
      while (key == true)
         Swap(lock, key);
        critical section
      lock=false;
        remainder section
} while (1);
```

Synchronization Hardware

```
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) && (not waiting[j])
                  j= (j+1) % n;
      If (j=i) lock=false;
      else waiting[j]=false;
      remainder section
} while (1);
```

- Mutual Exclusion
  - Pass if key == F or waiting[i] == F
- Progress
  - Exit process sends a process in.
- Bounded Waiting
  - Wait at most n-1 times
- Atomic TestAndSet is hard to implement in a multiprocessor environment.

Semaphores

- Motivation:
    - A high-level solution for more complex problems.
- Semaphore
    - A variable S only accessible by two atomic operations:

```
wait(S) {        /* P */
    while (S <= 0) ;
    S—;
}
```

```
signal(S) {     /* V */
    S++;
}
```

•Indivisibility for “(S<=0)”, “S—”, and “S++”

Semaphores – Usages

- Critical Sections

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```

- Precedence Enforcement

```
P1:
   S1;
   signal(synch);

P2:
   wait(synch);
   S2;
```

Semaphores

- Implementation
  - Spinlock – A Busy-Waiting Semaphore
    - “while (S <= 0)” causes the wasting of CPU cycles!
    - Advantage:
      - When locks are held for a short time, spinlocks are useful since no context switching is involved.
  - Semaphores with Block-Waiting
    - No busy waiting from the entry to the critical section!

Semaphores

- Semaphores with Block Waiting

```
typedef struct {
        int value;
        struct process *L;
} semaphore ;
```

```
void wait(semaphore S) {
        S.value--;
        if (S.value < 0) {
            add this process to S.L;
            block();
        }
}
```

```
void signal(semaphore S);
        S.value++;
        if (S.value <= 0) {
            remove a process P form S.L;
                wakeup(P);
        }
}
```

* |S.value| = the # of waiting processes if S.value < 0.

Semaphores

- The queueing strategy can be arbitrary, but there is a restriction for the bounded-waiting requirement.
- Mutual exclusion in wait() & signal()
  - Uniprocessor Environments
    - Interrupt Disabling
    - TestAndSet, Swap
    - Software Methods, e.g., the Bakery Algorithm, in Section 7.2
  - Multiprocessor Environments
- Remarks: Busy-waiting is limited to only the critical sections of the wait() & signal()!

Deadlocks and Starvation

- Deadlock
  - A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

```
P0:  wait(S);          P1: wait(Q);
     wait(Q);              wait(S);
       …                     …
     signal(S);            signal(Q);
     signal(Q);            signal(S);
```

- Starvation (or Indefinite Blocking)
  - E.g., a LIFO queue

Binary Semaphore

- Binary Semaphores versus Counting Semaphores
  - The value ranges from 0 to 1→ easy implementation!

<u>wait(S)</u>

```
wait(S1);    /* protect C */
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

<u>signal(S)</u>

```
wait(S1);
C++;
if (C <= 0)
    signal (S2); /* wakeup */
else
    signal (S1);
```

* S1 & S2: binary semaphores

Classical Synchronization Problems – The Bounded Buffer

<u>Producer:</u>

```
do {
        produce an item in nextp;
        …….
        wait(empty);  /* control buffer availability */
        wait(mutex);  /* mutual exclusion */
        ……
        add nextp to buffer;
        signal(mutex);
        signal(full);  /* increase item counts */
} while (1);
```

Initialized to *n* ⇨ wait(empty)

Initialized to *1* ⇨ wait(mutex)

Initialized to *0* ⇨ signal(full)

Classical Synchronization Problems – The Bounded Buffer

Consumer:

```
                        do {
Initialized to 0  ⇨          wait(full);  /* control buffer availability */
Initialized to 1  ⇨          wait(mutex);  /* mutual exclusion */
                             …….
                             remove an item from buffer to nextp;
                             ……
                             signal(mutex);
Initialized to n  ⇨          signal(empty);  /* increase item counts */
                             consume nextp;
                        } while (1);
```

Classical Synchronization Problems – Readers and Writers

- The Basic Assumption:
  - Readers: shared locks
  - Writers: exclusive locks
- The first reader-writers problem
  - No readers will be kept waiting unless a writer has already obtained permission to use the shared object → potential hazard to writers!
- The second reader-writers problem:
  - Once a writer is ready, it performs its write asap! → potential hazard to readers!

Classical Synchronization Problems – Readers and Writers

First R/W Solution ⇨

```
semaphore wrt, mutex;
  (initialized to 1);
int readcount=0;
Writer:
    wait(wrt);
    ……
    writing is performed
    ……
    signal(wrt)
```

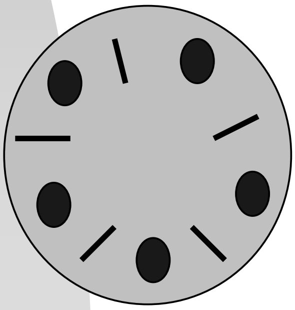
Queueing mechanism ➡ wait(wrt);

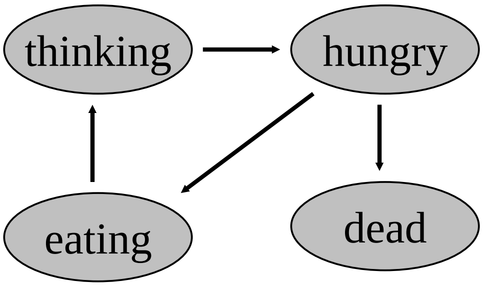
```
Reader:
    wait(mutex);
    readcount++;
    if (readcount == 1)
            wait(wrt);
    signal(mutex);
    …… reading ……
    wait(mutex);
    readcount--;
    if (readcount== 0)
            signal(wrt);
    signal(mutex);
```

Which is awaken? ➢ signal(mutex);

Classical Synchronization Problems – Dining-Philosophers

- Each philosopher must pick up one chopstick beside him/her at a time
- When two chopsticks are picked up, the philosopher can eat.





Classical Synchronization Problems – Dining-Philosophers

```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i + 1) % 5 ]);
        … eat …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        …think …
} while (1);
```

Classical Synchronization Problems – Dining-Philosophers

- Deadlock or Starvation?!
- Solutions to Deadlocks:
  - At most four philosophers appear.
  - Pick up two chopsticks “simultaneously”.
  - Order their behaviors, e.g., odds pick up their right one first, and evens pick up their left one first.
- Solutions to Starvation:
  - No philosopher will starve to death.
    - A deadlock could happen??

Critical Regions

- Motivation:
  - Various programming errors in using low-level constructs,e.g., semaphores
    - Interchange the order of wait and signal operations
    - Miss some waits or signals
    - Replace waits with signals
    - etc
- The needs of high-level language constructs to reduce the possibility of errors!

Critical Regions

- Region v when B do S;
  - Variable v – shared among processes and only accessible in the region

```
struct buffer {
   item pool[n];
   int count, in, out;
};
```

  - B – condition
    - count < 0
  - S – statements

Example: Mutual Exclusion

region *v* when (true) *S1*;

region *v* when (true) *S2*;

Critical Regions – Consumer-Producer

```
struct buffer {
        item pool[n];
        int count, in, out;
};
```

Producer:

```
region buffer when
(count < n) {
     pool[in] = nextp;
     in = (in + 1) % n;
     count++;
}
```

Consumer:

```
region buffer when
(count > 0) {
     nextc = pool[out];
     out = (out + 1) % n;
     count--;
}
```

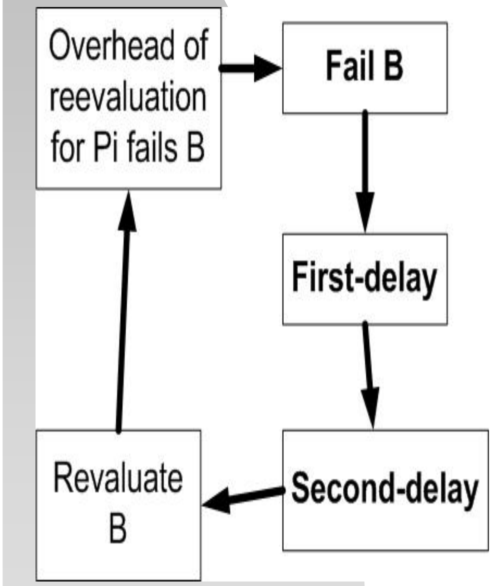
Critical Regions – Implementation by Semaphores

Region **x** when B do S;

```
/* to protect the region */
semaphore mutex;
/* to (re-)test B */
semaphore first-delay;
int first-count=0;
/* to retest B */
semaphore second-delay;
int second-count=0;
```

```
wait(mutex);                              ○ ●
   while (!B) {                             ○ ●
        /* fail B */
        first-count++;
        if (second-count > 0)
           /* try other processes waiting
              on second-delay */
          signal(second-delay);
          else signal(mutex);                ○ ●
        /* block itself on first-delay */
        wait(first-delay);                   ○ ●
```

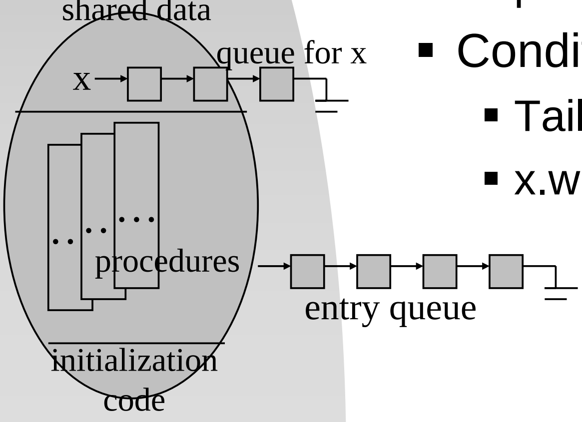
Critical Regions – Implementation by Semaphores



```
        first-count--;
        second-count++;
        if (first-count > 0)
           signal(first-delay);
           else signal(second-delay);
        /* block itself on first-delay */
        wait(second-delay);
        second-count--;
}
S;
if (first-count > 0)
        signal(first-delay);
else if (second-count > 0)
        signal(second-delay);
else signal(mutex);
```

Monitor

- Components
  - Variables – monitor state
  - Procedures
    - Only access local variables or formal parameters
  - Condition variables
    - Tailor-made sync
    - x.wait() or x.signal



```
monitor name {
   variable declaration
  void proc1(…) {
  }
 …
  void procn(…) {
  }
}
```

Monitor

- Semantics of signal & wait
  - x.signal() resumes one suspended process. If there is none, no effect is imposed.
  - *P* x.signal() a suspended process *Q*
    - *P* either waits until *Q* leaves the monitor or waits for another condition
    - *Q* either waits until *P* leaves the monitor, or waits for another condition.

Monitor – Dining-Philosophers

```
Pi:
  dp.pickup(i);
  … eat …
  dp.putdown(i);
```

```
monitor dp {
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i) {
          stat[i]=hungry;
          test(i);
          if (stat[i] != eating)
             self[i].wait;
    }
    void putdown(int i) {
          stat[i] = thinking;
          test((i+4) % 5);
          test((i + 1) % 5);
    }
```

Monitor – Dining-Philosophers

No deadlock!
But starvation could occur!

```
void test(int i) {
    if (stat[(i+4) % 5]) != eating &&
        stat[i] == hungry &&
        state[(i+1) % 5] != eating) {
           stat[i] = eating;
           self[i].signal();
    }
}
void init() {
    for (int i=0; i < 5; i++)
           state[i] = thinking;
}
```

* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2005.

Monitor – Implementation by Semaphores

- Semaphores
  - *mutex* – to protect the monitor
  - *next* – being initialized to zero, on which processes may suspend themselves
    - *nextcount*
- For each external function *F*

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

* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2005.

Monitor – Implementation by Semaphores

- For every condition *x*
  - A semaphore *x-sem*
  - An integer variable *x-count*
  - Implementation of x.wait() and x.signal :

- x.wait()

```
x-count++;
if (next-count > 0)
      signal(next);
else signal(mutex);
wait(x-sem);
x-count--;
```

- x.signal

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

* x.wait() and x.signal() are invoked within a monitor.

Monitor

- Process-Resumption Order
  - Queuing mechanisms for a monitor and its condition variables.
  - A solution:

    x.wait(c);

    - where the expression *c* is evaluated to determine its process's resumption order.

```
R.acquire(t);
 …
access the resource;
R.release;
```

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

Monitor

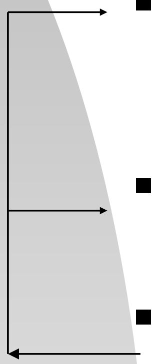
- Concerns:
  - Processes may access resources without consulting the monitor.
  - Processes may never release resources.
  - Processes may release resources which they never requested.
  - Process may even request resources twice.

Monitor

- Remark: Whether the monitor is correctly used?

  => Requirements for correct computations
  - Processes always make their calls on the monitor in correct order.
  - No uncooperative process can access resource directly without using the access protocols.
- Note: Scheduling behavior should consult the built-in monitor scheduling algorithm if resource access RPC are built inside the monitor.

Synchronization – Solaris



- Semaphores and Condition Variables
- Adaptive Mutex
  - Spin-locking if the lock-holding thread is running; otherwise, blocking is used.
- Readers-Writers Locks
  - Expensive in implementations.
- Turnstile
  - A queue structure containing threads blocked on a lock.
  - Priority inversion → priority inheritance protocol for kernel threads

---

Synchronization – Windows XP

- General Mechanism
  - Spin-locking for short code segments in a multiprocessor platform.
  - Interrupt disabling when access to global variables is done in a uniprocessor platform.
- Dispatcher Object
  - State: signaled or non-signaled
  - *Mutex* – select one process from its waiting queue to the ready queue.
  - *Events* – select all processes waiting for the event.

Synchronization – Linux

- Preemptive Kernel After Version 2.6
  - Spin-locking for short code segments in a multiprocessor platform.
  - Interrupt disabling and enabling in a uniprocessor platform.
    - preempt_disable() and preempt_enable()
    - Preempt_count

Synchronization – Pthreads

- General Mechanism
  - Mutex locks – mutual exclusion
  - Condition variables – Monitor
  - Read-write locks
- Extensions
  - POSIX SEM extension: semaphores
  - Spinlocks – portability?

Atomic Transactions

- Why Atomic Transactions?
  - Critical sections ensure mutual exclusion in data sharing, but the relationship between critical sections might also be meaningful!

  → Atomic Transactions

- Operating systems can be viewed as manipulators of data!

Atomic Transactions – System Model

- Transaction – a logical unit of computation
  - A sequence of read and write operations followed by a commit or an abort.
- Beyond "critical sections"
  1. Atomicity: All or Nothing
     - An aborted transaction must be *rolled back*.
     - The effect of a committed transaction must persist and be imposed as a logical unit of operations.

Atomic Transactions – System Model

2. Serializability:
   - The order of transaction executions must be equivalent to a serial schedule.

T0	T1
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Two operations O_i & O_j conflict if
1. Access the same object
2. One of them is write

Atomic Transactions – System Model

- Conflict Serializable:
  - *S* is conflict serializable if *S* can be transformed into a serial schedule by swapping nonconflicting operations.

T0	T1
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

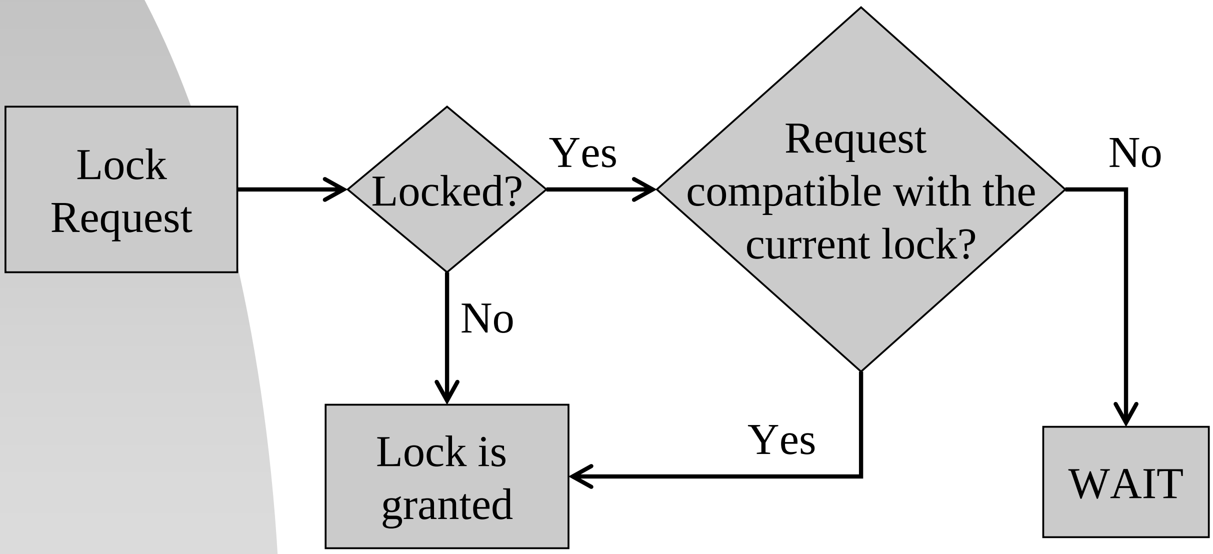
→

T0	T1
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Atomic Transactions – Concurrency Control

- Locking Protocols
  - Lock modes (A general approach!)
    - 1. Shared-Mode: “Reads”.
    - 2. Exclusive-Mode: “Reads” & “Writes“
  - General Rule
    - A transaction must receive a lock of an appropriate mode of an object before it accesses the object. The lock may not be released until the last access of the object is done.

Atomic Transactions – Concurrency Control

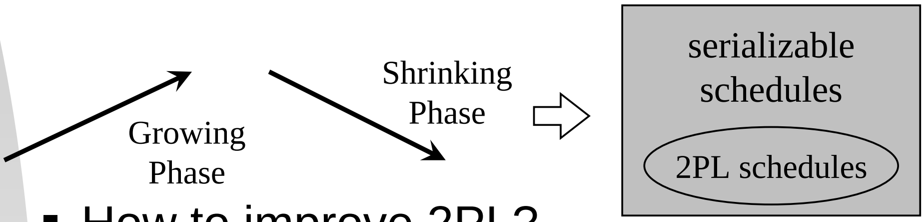


Atomic Transactions – Concurrency Control

- When to release locks w/o violating serializability

  R0(A) W0(A) R1(A) R1(B) R0(B) W0(B)

- Two-Phase Locking Protocol (2PL) – Not Deadlock-Free



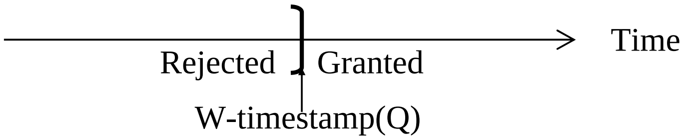
- How to improve 2PL?
  - Semantics, Order of Data, Access Pattern, etc.

Atomic Transactions – Concurrency Control

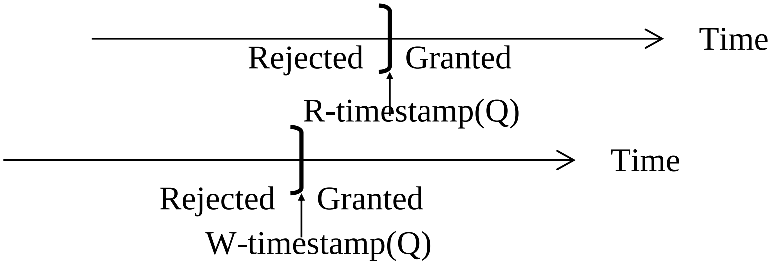
- Timestamp-Based Protocols
  - A time stamp for each transaction TS(T_i)
    - Determine transactions' order in a schedule in advance!
  - A General Approach:
    - TS(T_i) – System Clock or Logical Counter
      - Unique?
    - Scheduling Scheme – deadlock-free & serializable
      - $W-timestamp(Q) = Max_{T_i - W(Q)}(TS(T_i))$
      - $R-timestamp(Q) = Max_{T_i - R(Q)}(TS(T_i))$

Atomic Transactions – Concurrency Control

- R(Q) requested by T_i → check TS(T_i) !



- W(Q) requested by T_i → check TS(T_i) !



- Rejected transactions are rolled back and restated with a new time stamp.

Failure Recovery – A Way to Achieve Atomicity

- Failures of Volatile and Nonvolatile Storages!
  - Volatile Storage: Memory and Cache
  - Nonvolatile Storage: Disks, Magnetic Tape, etc.
  - Stable Storage: Storage which never fail.
- Log-Based Recovery
  - Write-Ahead Logging
    - Log Records

< Ti starts >

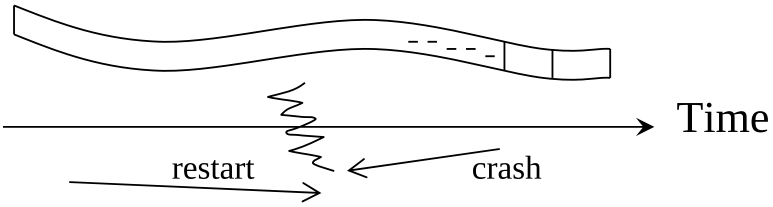
< Ti commits >

< Ti aborts >

< Ti, Data-Item-Name, Old-Value, New-Value>

Failure Recovery

- Two Basic Recovery Procedures:



  - undo(Ti): restore data updated by Ti
  - redo(Ti): reset data updated by Ti
- Operations must be idempotent!
- Recover the system when a failure occurs:
  - “Redo” committed transactions, and “undo” aborted transactions.

Failure Recovery

- Why Checkpointing?
  - The needs to scan and rerun all log entries to redo committed transactions.
- CheckPoint
  - Output all log records, Output DB, and Write <check point> to stable storage!
  - Commit: A Force Write Procedure