Process Synchronization

Mehdi Kargahi
School of ECE
University of Tehran
Summer 2016

Producer-Consumer (Bounded 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

- Producer

$$\begin{aligned}&register_1 = \texttt{counter}\\&register_1 = register_1 + 1\\&\texttt{counter} = register_1\end{aligned}$$

- Consumer

$$\begin{aligned}&register_2 = \texttt{counter}\\&register_2 = register_2 - 1\\&\texttt{counter} = register_2\end{aligned}$$

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

When CPU Scheduling Occurs

1. Running process switches to wait state
   - I/O request
   - Waiting for termination of one of the child processes
2. Running process switches to ready state (timer interrupt)
3. Waiting process switches to ready state (I/O completion)
4. Running process terminates

Under non-preemptive scheduling: 1 and 4 are legal scheduling points

Under preemptive scheduling: 1 through 4 are legal Scheduling points

Critical Sections

- Structure of a Typical Process

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

- Requirements for Solving critical-section problem:
  - Mutual exclusion
  - Progress
  - Bounded waiting

Two-Process Solutions

- Algorithm 1
  - Shared: int turn;

P_i:

```
while (TRUE) {
    while (turn!=i);
    critical-section
    turn = 1-i;
    remainder-section
}
```

Mutual exclusion yes

Progress no

Bounded waiting yes

Two-Process Solutions

- Algorithm 2
  - Shared: boolean flag[2];

```
Pi:
flag[i] = FALSE;
do {
    flag[i] = TRUE;
    while (flag[j]);
   critical-section
   flag[i] = FALSE;
   remainder-section
} while (TRUE);
```

Mutual exclusion yes

Progress no

Bounded waiting yes

Peterson's Solution

```
do {

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Eisenberg & McGuire Solution

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

```
do {
   while (TRUE) {
      flag[i] = want_in;
      j = turn;

      while (j != i) {
         if (flag[j] != idle) {
            j = turn;
         else
            j = (j + 1) % n;
      }

      flag[i] = in_cs;
      j = 0;

      while ( (j < n) && (j == i || flag[j] != in_cs) )
         j++;

      if ( (j >= n) && (turn == i || flag[turn] == idle) )
         break;
   }

      // critical section

   j = (turn + 1) % n;

   while (flag[j] == idle)
      j = (j + 1) % n;

   turn = j;
   flag[i] = idle;

      // remainder section
}while (TRUE);
```

Synchronization Hardware

- Atomic instructions: TestAndSet

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

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Synchronization Hardware

- Atomic instructions: Swap

```
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 (TRUE);
```

Synchronization Hardware

- Atomic instructions: Compare_and_Swap

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Synchronization Hardware

- Spin lock
  - Methods that use busy waiting
- n-process solution with bounded-waiting

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

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) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Mutex Locks

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

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

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

Semaphores

- A semaphore S is an integer variable
  - Two atomic operations

```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphore Usage

- Critical-section

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

- Process synchronization (<u>Counting vs. Binary</u> Semaphore)

```
S1;
signal(synch);
```

```
wait(synch);
S2;
```

- *Problems*: Spin-lock & bounded-waiting

A Better Definition

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

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

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

Semantics

- The value of semaphore specifies the number of processes waiting on that semaphore
- It is assumed that wait and signal are executed atomically, while this is not simply valid
- *Method 1*: disabling interrupts to prevent interleaving of wait and signal operations on a semaphore
  - In multiprocessor systems, interrupts on all processors should be disabled resulting in quite low performance
- *Method 2*: using one of the previous critical-section methods (busy waiting)➔ More proper for MP systems
  - Wait and signal have a few instructions resulting in short critical sections➔ Busy waiting is not problematic

Deadlocks

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

- What is the difference between deadlock and starvation?

Priority Inversion

- PI occurs in systems with more than two priorities. So one solution is to have at most two priority levels.
- PIP: Priority-Inheritance Protocol
- PI and the Mars Pathfinder

Classical Problems of Synchronization

- The bounded-buffer problem

Producer

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

Consumer

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

The Readers-Writers Problem

```
semaphore mutex, wrt;
int readcount;
```

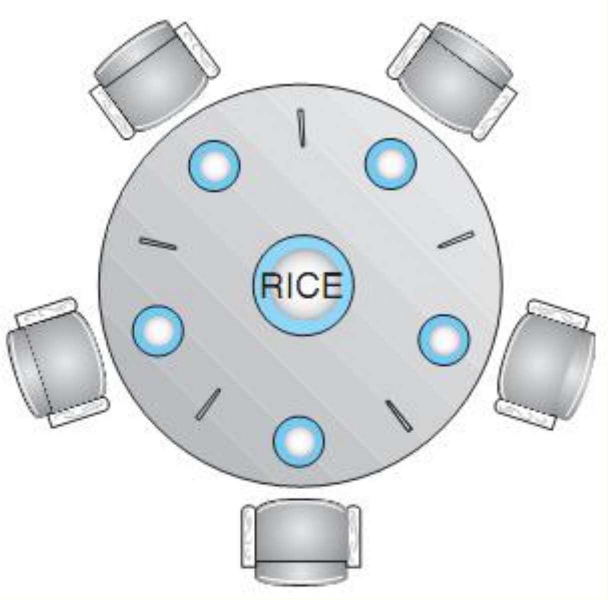
Writer

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

Reader

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

The Dining Philosophers Problem



The Dining Philosophers Problem

```
semaphore chopstick[5];
```

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

- Problem?
- Three solutions !!
- Is a deadlock-free solution also starvation-free?

Monitors

- Problems with semaphores (programming faults):

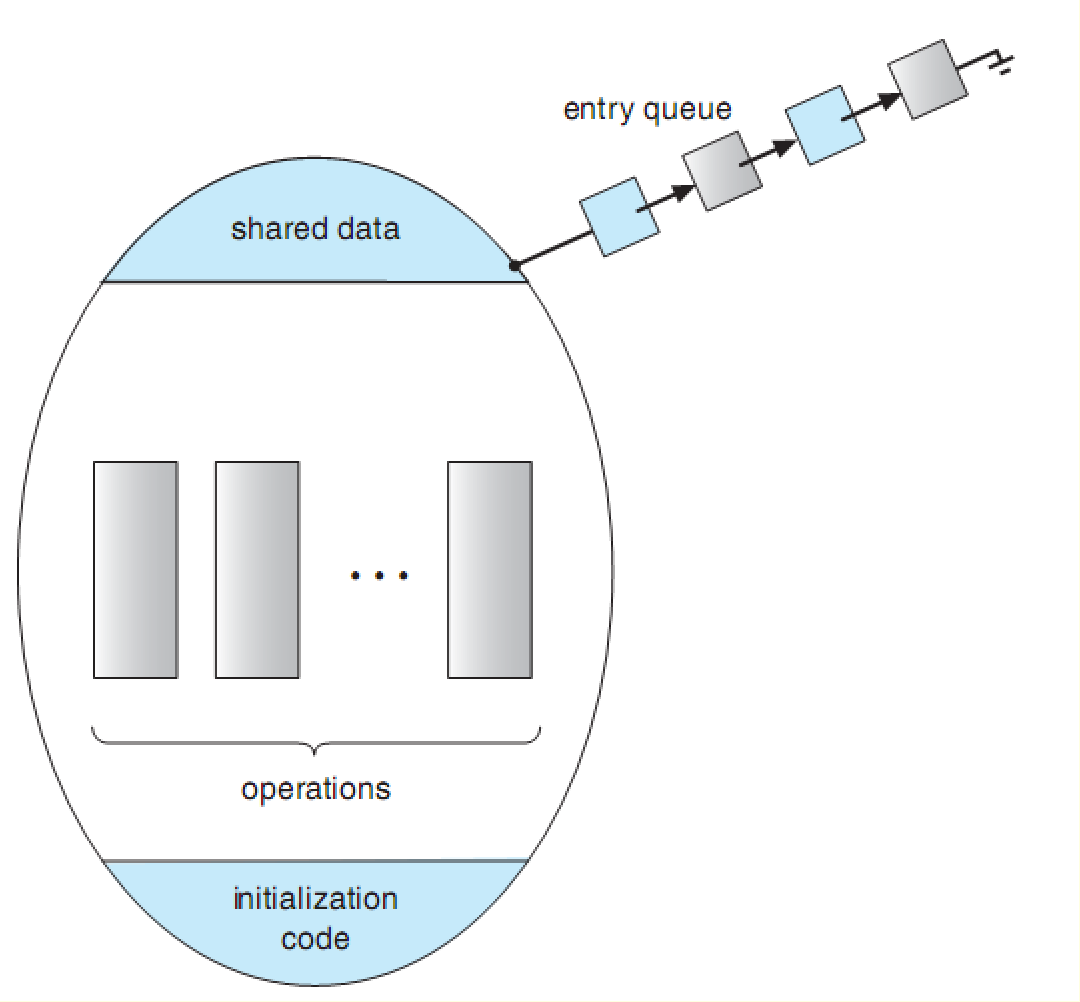
```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

Syntax of a Monitor

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }


         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

A Monitor Scheme

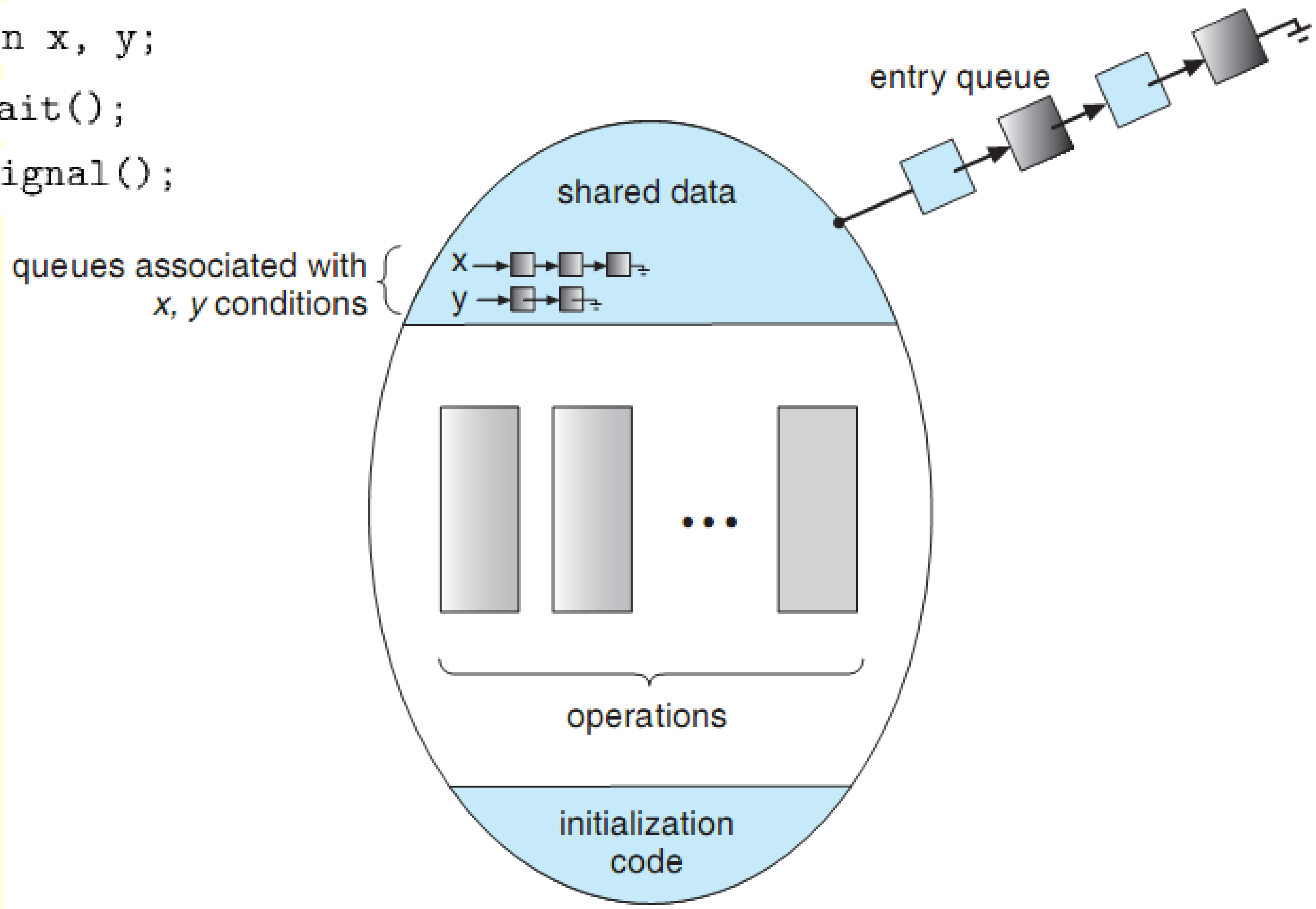


Monitor with Condition Variables

```
condition x, y;
```

```
x.wait();
```

```
x.signal();
```



When x.signal() is Invoked by P

- Two possibilities exist:

1. **Signal and wait**. P either waits until Q leaves the monitor or waits for another condition.
2. **Signal and continue**. Q either waits until P leaves the monitor or waits for another condition.

Dining-Philosophers Solution using Monitors

```
enum {thinking, hungry, eating} state[5];
```

```
condition self[5];
```

```
dp.pickup(i);
   ...
   eat
   ...
dp.putdown(i);
```

Dining-Philosophers Solution using Monitors

```
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((i + 4) % 5);
     test((i + 1) % 5);
  }

  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;
   }
}
```

Implementing a Monitor using Semaphores

- External procedure F is replaced by

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

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--;
}
```

Resuming Processes within a Monitor

- x.wait(c), where c is called a priority number

```
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;
   }
}
```

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

Similar Mechanisms

- Down: wait
- Up: signal

Problems with Monitors

- A process might access a resource without first gaining access permission
- A process might never release a resource
- A process might attempt to release a resource that it never requested
- A process might request the same resource twice

Atomic Transactions

- Transaction
  - A collection of instructions that performs a single logical function
- A transaction may **commit** or **abort**
- When a transaction is aborted, the state of the data accessed by it is restored to what it was before executing that transaction, i.e., it is **rolled back**
- The execution of a transaction should be assumed as **atomic**

Different Types of Storage

- Volatile storage
- Non-Volatile storage
- Stable storage

Log-Based Recovery

- Log is stored on the stable storage
- Write-ahead log
  - Every log precedes a single write operation by
    - Transaction name
    - Data item name
    - Old value
    - New value
  - Special log records
    - *<T_i starts>*
    - *<T_i commits>*

Recovery Algorithm

- *Undo*(T_i): restores the value of all data updates by T_i to the old values
- *Redo*(T_i): sets the value of all data updates by T_i to the new values
- If T_i aborts, we need to execute *undo*(T_i)
- After a system failure
  - Transaction T_i needs to be undone if the log contains the $< T_i$ `starts`$>$ record but does not contain the $< T_i$ `commits`$>$ record.
  - Transaction T_i needs to be redone if the log contains both the $< T_i$ `starts`$>$ and the $< T_i$ `commits`$>$ records.

Concurrent Atomic Transactions

- Serializability
  - Restrictive
- Serial vs. non-serial schedules

Serial

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

Non-serial but serializable

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

- Conflicting operations: O_i and O_j are conflicting if they access the same data item and at least one of them is a *write* operation

Concurrent Atomic Transactions

- A conflict serializable schedule
  - If a schedule S can be transformed to a serial schedule S' by a series of swaps of non-conflicting operations

Locking Protocol

- Shared vs. exclusive lock
- Two-phase locking (2PL) protocol
  - **Growing phase**. A transaction may obtain locks but may not release any lock.
  - **Shrinking phase**. A transaction may release locks but may not obtain any new locks.
- Properties
  - Ensures conflict Serializability
  - It is not deadlock-free
  - There may be conflict-serializable schedules that cannot be obtained by 2PL
    - Additional information on transactions may improve 2PL

Timestamp-Based Protocols

- Each transaction T_i is assigned a timestamp in an ascending order of time by the system

  - Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
  - Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned.

- Each data item Q has two timestamps

  W-timestamp(Q) denotes the largest timestamp of any transaction that successfully executed `write`(Q).

  R-timestamp(Q) denotes the largest timestamp of any transaction that successfully executed `read`(Q).

Timestamp-Based Protocols

- If transaction T_i issues *read*(Q)
  - If $TS(T_i) <$ W-timestamp(), then T_i needs to read a value of Q that was already overwritten. Hence, the `read` operation is rejected, and T_i is rolled back.
  - If $TS(T_i) \geq$ W-timestamp(Q), then the `read` operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and $TS(T_i)$.
- If transaction T_i issues *write*(Q)
  - If $TS(T_i) <$ R-timestamp(Q), then the value of Q that T_i is producing was needed previously and T_i assumed that this value would never be produced. Hence, the `write` operation is rejected, and T_i is rolled back.
  - If $TS(T_i) <$ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this `write` operation is rejected, and T_i is rolled back.
  - Otherwise, the `write` operation is executed.
- Each transaction that is rolled is restarted with a new timestamp

Timestamp-Based Protocols

- Properties
  - Deadlock-free: no transaction ever waits
  - Not stronger than 2PL and vice versa

Approaches for Multicore Processors

- Problems if we use classic locks
  - Increased risk of race condition
  - Increased risk of deadlock
  - Low scalability because of contention between very large number of threads
- Solutions that better scale for large number of cores to support designing thread-safe concurrent applications
  - In programming languages
  - In hardware