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
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting (Cont.)

```
wait (S) {
        value--;
        if (value < 0) {
              add this process to waiting queue
              block();  }
      }

Signal (S) {
        value++;
        if (value <= 0) {
              remove a process P from the waiting queue
             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.

Solution to Dining Philosophers 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 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;
    }
}
```

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

6.9: Atomic Transactions

- Introduce notions of databases into operating systems
  - Challenge is that some of these operations are “heavy” and not necessarily fast
- Transaction:
  - A collection of operations that performs a single logical function. For example, transferring money from your checking account to savings account will be one single transaction
  - Transactions are atomic with all are nothing semantics
    - Committed transactions means, all the operations went through
    - Aborted transactions means, none of them went through
    - You cannot be in a state where the money came out of your checking account but didn’t go into savings accounts
    - When a transaction aborts, we roll back

Storage states

- Storage to implement transactions:
  - Volatile storage: Does not survive system crash
  - Nonvolatile storage: Survives system crashes
  - Stable storage: Information is “never” lost. Uses nonvolatile storage and replication
- Log-based recovery:
  - Write-ahead logging, where we write all operations into a log in stable storage
    - <transaction name, data item name, old value, new value>
  - Transaction is made up of
    - <Ti, starts> set of transaction logs <Ti, commit>
    - If both starts and commit is there, then the transaction is committed. Else, it is rolled back
    - Logs are idempotent, you can apply it again and again in the same order without side effects

Checkpoints

- Logs keep growing. After every failure, we'd have to go back and replay the log. This can be time consuming.
- Checkpoint frequently
  - Output all log records currently in volatile storage onto stable storage
  - Output all modified data residing in volatile storage to the stable storage
  - Output a log record <checkpoint> into stable storage
- On failure, search backwards till we hit the first checkpoint. The first transaction start from the checkpoint (going back) is the start of replay

Serializability

- Transactions can be concurrent. Such concurrency may cause problems depending on the interleaving of the transactions. We introduce stricter notions of this phenomenon in order to predict system behavior
- Schedule is an execution sequence
- Serial schedule: Schedule where two concurrent transactions follow one after the other
  - For two transactions T1, T2: serial schedule is T1 then T2 or T2 then T1. For n transactions, we have n! choices, all of which is valid
  - Serial schedule cannot fully utilize the system resources and so we want to relax the schedule: non-serial schedule

Conflict

- We define a schedule to be in conflict if they both operate on the same data item and one of the operations is a write
- If there is no conflict, the schedule can be swapped.
- If after non-conflicting swaps we reach a serial schedule, then that schedule is called conflict serializable

Read(A)	
Write(A)	
Read(B)	
Write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Serial schedule

Read(A)	
Write(A)	
	read(A)
	write(A)
Read(B)	
Write(B)	
	read(B)
	write(B)

Conflict serializable schedule