PROCESS COORDINATION II

SOME PROCESS COORDINATION PROBLEMS

THE BOUNDED BUFFER (PRODUCER / CONSUMER) PROBLEM:

- This is the same producer / consumer problem as before. But now we'll do it with signals and waits. Remember: a **wait decreases** its argument and a **signal increases** its argument.

```
        INITIALIZE: mutex = 1;   empty = n;   full = 0;

producer:
        repeat
                /* produce an item in nextp */
                wait (empty);                   /* Do action     */
                wait (mutex);                   /* Buffer guard*/
                /* add nextp to buffer  */
                signal (mutex);
                signal (full);
        until ( false );

consumer:
        repeat
                wait (full);
                wait (mutex);
                /* remove an item from buffer to nextc */
                signal (mutex);
                signal (empty);
                /* consume an item in nextc */
        until ( false );
```

THE READERS/WRITERS PROBLEM:

- This is the same as the Producer / Consumer problem except - we now can have many concurrent readers and one exclusive writer.
- There are **shared** (for the readers) and **exclusive** (for the writer) locks.
- Two possible (contradictory) guidelines can be used:

a) No reader is kept waiting unless a writer holds the lock (the readers have precedence).

b) If a writer is waiting for access, no new reader gains access (writer has precedence).

- (NOTE: starvation can occur on either of these rules if they are followed rigorously.)

<<< This code is FIGURES 6.12, 6.13 >>>

```
        var     mutex,  wrt:   semaphore;
                readcount:     integer;

 Writer:
        repeat
                wait( wrt );
                /*   writing is performed    */
                signal(  wrt  );
        until( false );

 Reader:
    repeat
        wait( mutex );                        /* Allow 1 reader in entry*/
           readcount = readcount + 1;
           if readcount == 1  then  wait(wrt );    /* 1st reader locks writer */
        signal( mutex );

        /*   reading is performed  */

        wait( mutex );
           readcount = readcount - 1;
           if readcount == 0  then  signal(wrt );   /*last reader frees writer */
        signal( mutex );
     until( false );
```

THE DINING PHILOSOPHERS PROBLEM:

- 5 philosophers with 5 chopsticks sit around a circular table. They each want to eat at random times and must pick up the chopsticks on their right and on their left.

- Clearly deadlock is rampant (and starvation possible.)

- Several solutions are possible:

  a) Allow only 4 philosophers to be hungry at a time.

  b) Allow pickup only if both chopsticks are available. (Done in critical section)

  c) Odd # philosopher always picks up left chopstick 1st, even # philosopher always picks up right chopstick 1^{st}.

CRITICAL REGIONS:

- High Level synchronization construct implemented in a programming language.
- A shared variable v of type T, is declared as: **var** v; **shared** T
- Variable v is accessed only inside a statement: **region** v **when** B **do** S

where B is a Boolean expression.

- While statement S is being executed, no other process can access variable v.
- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression B is evaluated.
- If B is true, statement S is executed.
- If it is false, the process is delayed until B is true and no other process is in the region associated with v.

EXAMPLE: Bounded Buffer:

Shared variables declared as:

```
var     buffer:   shared record
                  pool:      array [ 0.. n - 1]      of  item;
                  count,  in,  out:                   integer;
        end;
```

Producer process inserts **nextp** into the shared buffer:

```
region   buffer   when  count  <  n
   do  begin
        pool[in]  =  nextp;
        in        =  in  +  1  mod n;
        count     =  count  + 1;
   end;
```

Consumer process removes an item from the shared buffer and puts it in nextc.

```
region   buffer   when  count  >  n
   do  begin
        nextc  =  pool[out];
        out     =  out  +  1  mod n;
        count   =  count  - 1;
   end;
```

SO, HOW IS SYNCHRONIZATION REALLY USED:

- The book discusses Solaris 2 as an example, but other operating systems work this way as well.
- Spin locks are used around critical sections that should be held only a short time. This is determined by:
  a) Is the lock holder currently running?
  b) Have we already spun for a while?
  c) Spin for some time and then cause reschedule. (This is very common because it's deterministic.)
- Long held locks (those held across a process reschedule or during a disk access) always cause a reschedule / sleep.

Atomic Transactions

- **Transaction** – a program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed.
- Must preserve atomicity despite possibility of failure.
- We are concerned here with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.
- We will look at several common uses of atomic transactions – situations where atomicity is required.

Log-Based Recovery

- **Write-ahead log** - all updates are recorded on the log, which is kept in stable storage; log has following fields:

  a) transaction name

  b) data item name; old value; new value

- The log has a record of < Ti **starts** >, and either < Ti **commits** > if the transactions commits, or < Ti **aborts** > if the transaction aborts.

- Recovery algorithm uses two procedures:

  undo(Ti) - restores value of all data updated by transaction Ti to the old values. It is invoked if the log contains record < Ti **starts** >, but not <Ti **commits** >.

  redo(Ti) - sets value of all data updated by transaction Ti to the new values. It is invoked if the log contains both < Ti **starts** > and < Ti **commits**>.

Checkpoints - reduce recovery overhead

- Output all log records currently residing in volatile storage onto stable storage.

- Output all modified data residing in volatile storage to stable storage.

- Output log record < **checkpoint** > onto stable storage.

- Recovery routine examines log to determine the most recent transaction Ti that started executing before the most recent checkpoint took place.

  Search log backward for first < **checkpoint** > record.
  Find subsequent < Ti start > record.

- **redo** and **undo** operations need to be applied to only transaction Ti and all transactions Tj that started executing after transaction Ti.

Concurrent Atomic Transactions

Serial schedule - the transactions are executed sequentially in some order.

Example of a serial schedule in which T0 is followed by T1 :

T0	T1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Conflicting operations - Oi and Oj **conflict** if they access the same data item, and at least one of these operations is a **write** operation.

Conflict serializable schedule - schedule that can be transformed into a serial schedule by a series of swaps of nonconflicting operations.

Example of a concurrent serializable schedule:

T0	T1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

- **Locking protocol** governs how locks are acquired and released; data item can be locked in following modes:

  Shared: If Ti has obtained a shared-mode lock on data item Q, then Ti can read this item, but it cannot write Q.

  Exclusive: If Ti has obtained an exclusive- mode lock on data item Q, then Ti can both read and w rite Q.

Two-phase locking 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.

- The two-phase locking protocol ensures conflict serializability, but does not ensure freedom from deadlock.

- **Timestamp-ordering** scheme - transaction ordering protocol for determining serializability order.

  a) With each transaction Ti in the system, associate a unique fixed timestamp, denoted by TS(Ti).

  b) If Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj).

- Implement by assigning two timestamp values to each data item Q.

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

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

- Example of a schedule possible under the timestamp protocol:

T2	T3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- There are schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa.

- The timestamp-ordering protocol ensures conflict serializability; conflicting operations are processed in timestamp order.