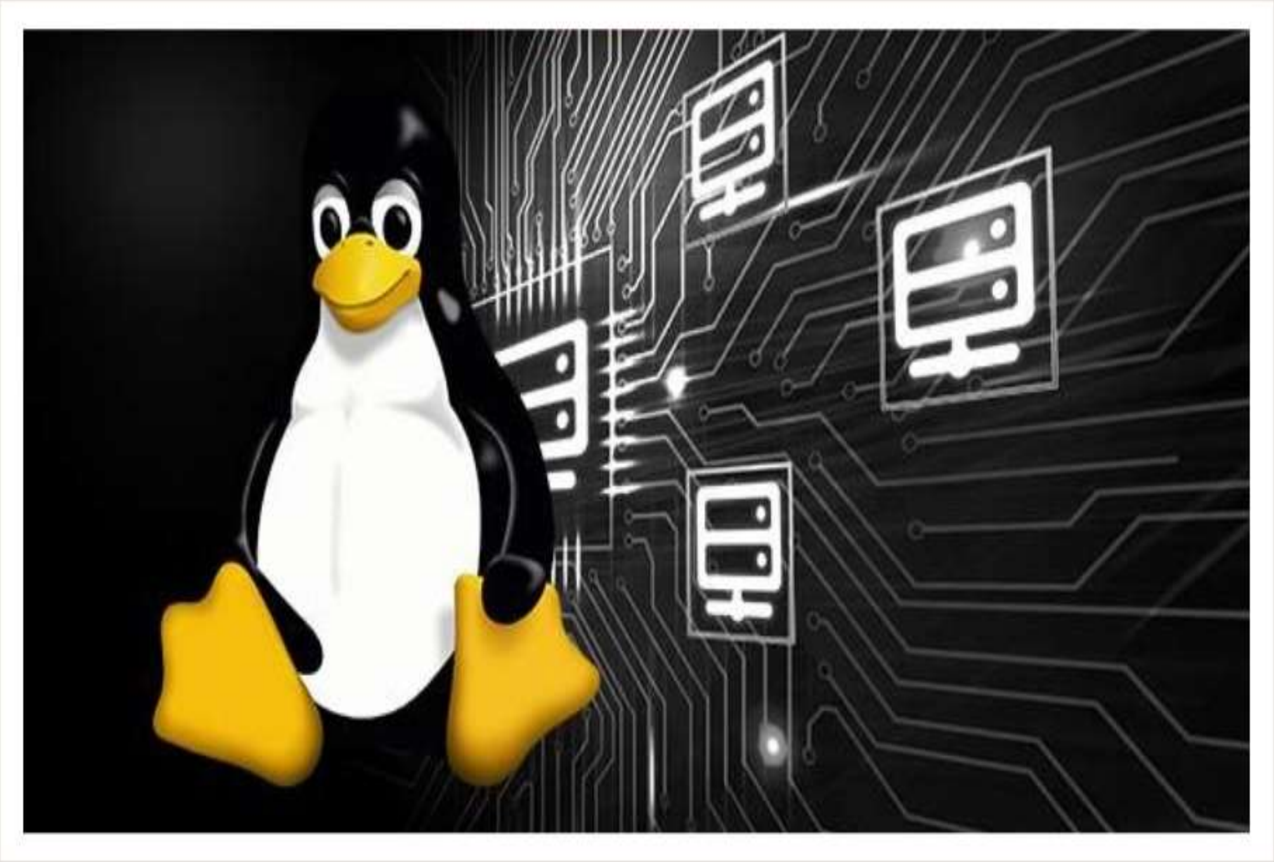


Operating systems

By
I Ravindra kumar, B.Tech, M.Tech,(Ph.D.)
Assistant professor,
Dept of CSE, VNR VJIET

Inter process communication (Cooperating processes)

- A cooperating process
  - one that can affect or be affected by other processes executing in the system.
  - can directly share a logical address space (that is, both code and data) or
  - can be allowed to share data only through files or messages.
- Messages
- Shared Data

Bounded buffer Problem

our solution allows at most BUFFER.SIZE - 1 items in the buffer at the same time

Producer

```
while (true)
{
/* produce an item in nextProduced */
 while (counter ==BUFFER.SIZE)
; /* do nothing */
buffer[in]= nextProduced;
in = (in + 1) %BUFFER-SIZE;
counter++;
}
```

Consumer

```
while (true)
{
   while (counter == 0)
   ; /* do nothing */
   nextConsumed=buffer[out];
out= (out + 1)%BUFFER_SIZE;
    counter--;
 /* consume the item in
 nextConsumed */
}
```

One such interleaving is

T0: Producer execute register1=counter {register1=5}
T1: Producer execute register1=register1+1 {register1=6}
T2: Consumer execute register2=counter {register2=5}
T3: Consumer execute register2=register2-1 {register2=4}
T4: Producer execute counter=register1 {counter=6}
T5: Consumer execute counter=register2 {counter=4}

Race condition:

Where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place

Critical Section

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion:**

   If process *Pi 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded Waiting:**

   There exists a bound 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.

Two-Process Solutions/Peterson's Solution

- Applicable to only two processes at a time
- The processes are numbered ***Po and P1.***
- When presenting *Pi, we use Pj to denote the other process;*

  that is, $j == 1 - i$

Algorithm:

- let the processes share a common integer variable turn
- initialized to 0 (or 1). If turn == i, then process ***Pi is allowed to execute in its*** critical section

Boolean flag[2];
int turn;
Initially flag [0] = flag [1] = false

```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (1);
```

int turn;

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

```
do{
Flag[0]=True
Turn=1;
While(flag[1]
   &&turn==1);
//Critical section
Flag[0]==False;
}while(1);
```

```
do{
Flag[1]=True
Turn=0;
While(flag[0]
   &&turn==0);
//Critical section
Flag[1]==False;
}while(1);
```

Synchronization Hardware

```
boolean TestAndSet(boo1ean &target) {
        boolean rv = target;
        target = true;
        return rv;}
```

```
do {

    while (TestAndSet(lock)) ;

        critical section

    lock = false;

        remainder section

} while(1);
```

Mutual-exclusion implementation with **TestAndSet**.

```
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);
```

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

    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 (1);
```

Bounded-waiting mutual exclusion with TestAndSet.

Semaphores

- semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal.

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

Mutual-exclusion implementation with semaphores.

do {

wait (mutex) ;

critical section

signal (mutex) ;

remainder section

} **while (1);**

Problem is Busy waiting

typedef struct {

int value ;

struct process *L;

} semaphore;

- Each semaphore has an integer value and a list of processes.
- When a process must wait on a semaphore, it is added to the list of processes.
- A **signal** operation removes one process from the list of waiting processes and awakens that process

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

- The **signal semaphore operation can now be defined as**

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

```
P0                P1

wait(S);          wait(Q);
wait(Q);          wait(S);

signal(S);        signal(Q);
signal(Q);        signal(S);
```

- Deadlock:
- Starvation:
  - a situation where processes wait indefinitely within the semaphore.
  - if we add and remove processes from the list associated with a semaphore in LIFO order.

Classic Problems of Synchronization

The Bounded-Buffer Problem :

- The **mutex semaphore** provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The **empty and full** semaphores count the number of empty and full buffers, respectively.

```
do {
    ...
  produce an item in nextp
    ...
  wait(empty) ;
  wait(mutex) ;
    ...
  add nextp to buffer
    ...
  signal(mutex);
  signal(full) ;
} while (1);
```

```
do {
  wait(full) ;
  wait(mutex) ;
    ...
  remove an item from buffer to nextc
    ...
  signal(mutex) ;
  signal(empty) ;
    ...
  consume the item in nextc
    ...
} while (1);
```

The Readers- Writers Problem

semaphore mutex, wrt;
int readcount;

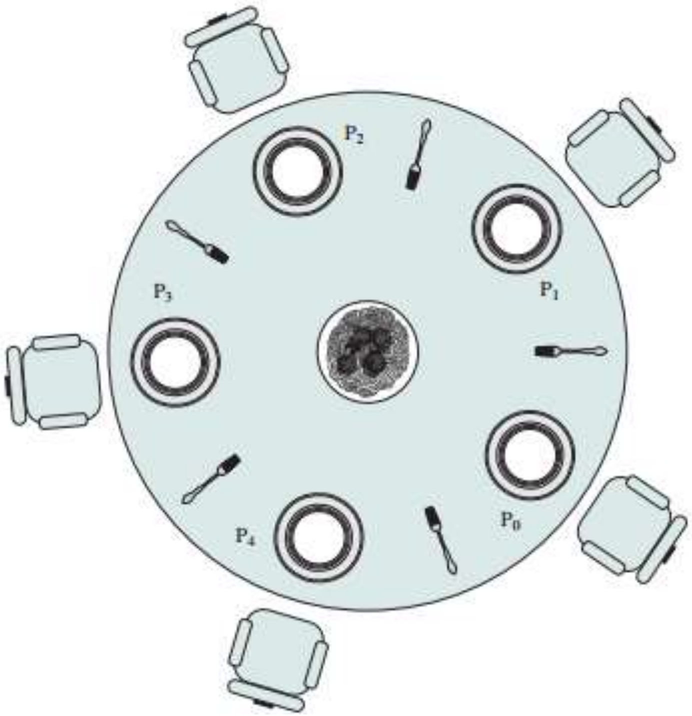
```
wait(wrt);
    ...
  writing is performed
    ...
signal(wrt);
```

The structure of a writer process.

```
wait(mutex) ;
readcount++;
if (readcount == 1)
   wait(wrt);
signal(mutex) ;
   ...
   reading is performed
   ...
wait(mutex) ;
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex) ;
```

The structure of a reader process.

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(1);
```

The structure of philosopher i.

Solution

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution;
  - an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Monitors

semaphore: but only if programmers use them properly

A monitor is essentially a class

- all data is private
- restrictions
  - only one method within any given monitor object may be active at the same time.
  - monitor methods may only access the shared data within the monitor and any data passed to them as parameters.

    I.e. they cannot access any data external to the monitor.

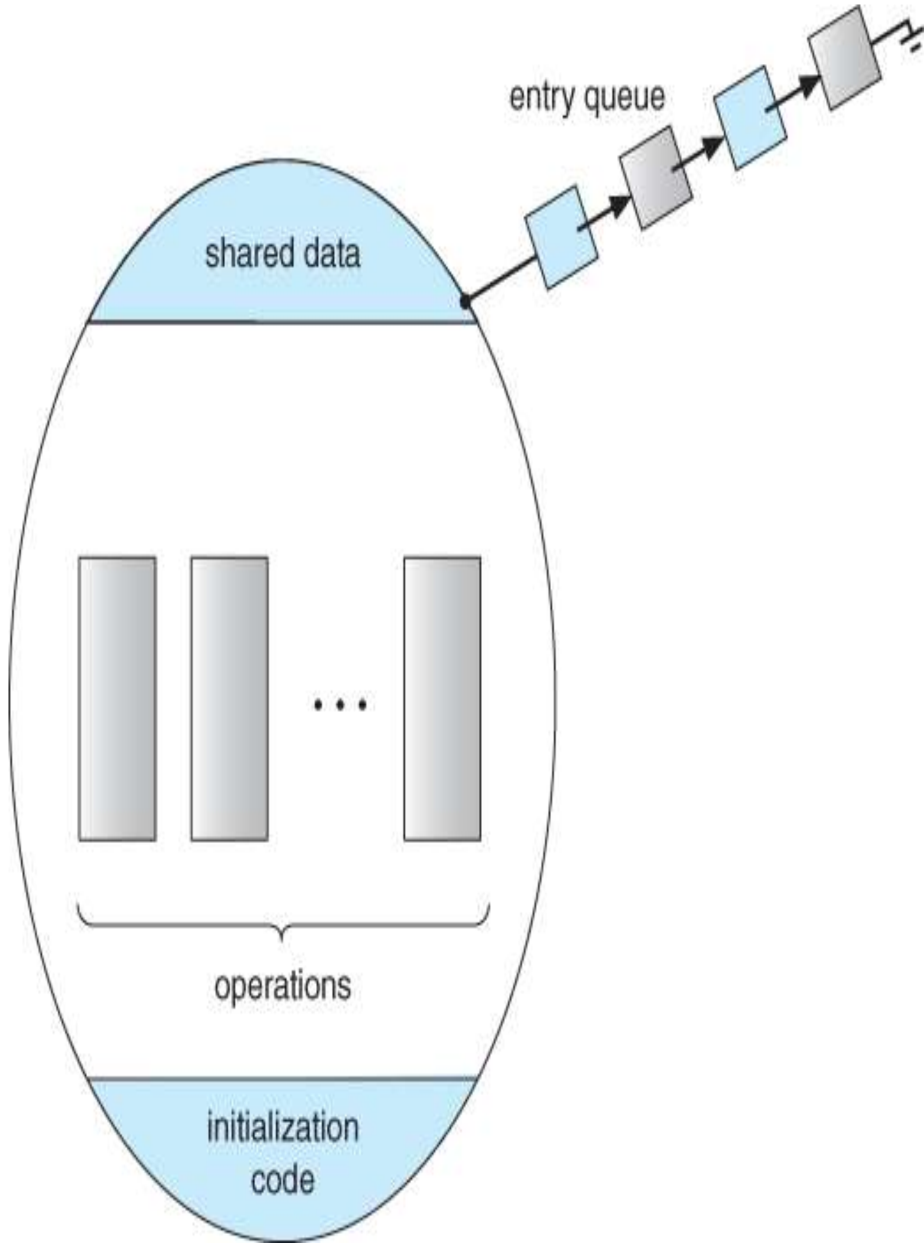
```
monitor monitor name
{
   // shared variable declarations

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

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

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

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



Condition:

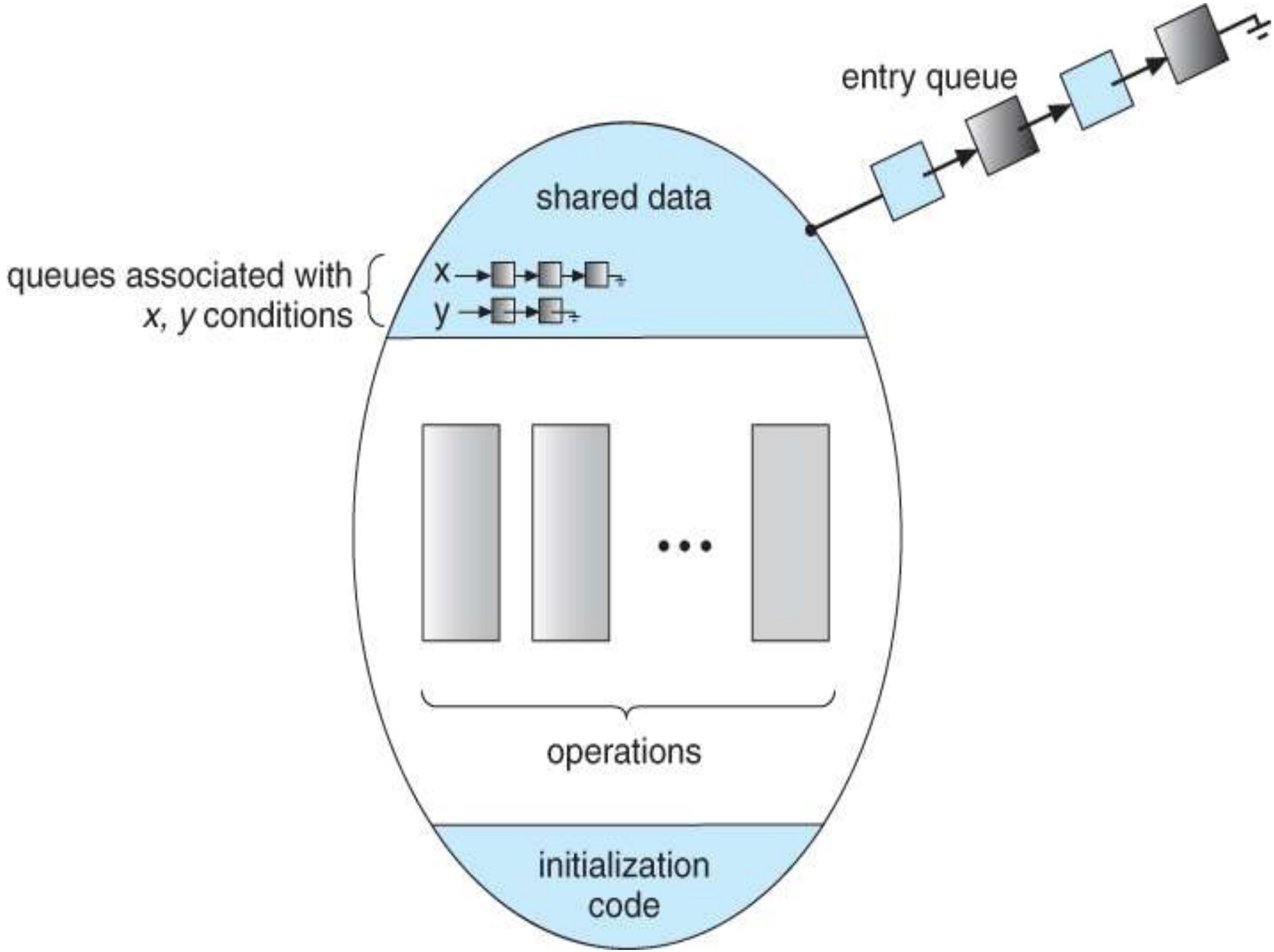
A variable of type condition has only two legal operations,

wait and signal

I.e. if X was defined as type condition, then legal operations would be

X.wait( ) and X.signal( )

- The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.
- The signal process does nothing if there are no processes waiting on that condition


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

If process P within the monitor issues a signal that would wake up process Q also within the monitor,
then there would be two processes running simultaneously within the monitor,
violating the exclusion requirement.

Two possible solutions to this dilemma:

- Signal and wait -

  When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

- Signal and continue -

  When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

```
monitor DiningPhilosophers
{ 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;
    }

}
```

Synchronization Examples

- **Synchronization in Solaris**
- To control access to critical sections, Solaris provides adaptive mutexes, condition variables, semaphores, reader-writer locks, and turnstiles.
- If the data are locked and therefore already in use, the adaptive mutex does one of two things.
- If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available, because the thread holding the lock is likely to finish soon.
- adaptive-mutex method to protect only data that are accessed by short code segments.
- That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, spin waiting will be exceedingly inefficient.
- Readers-writers locks are relatively expensive to implement, so again they are used on only long sections of code.
- Solaris uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader-writer lock.
- A turnstile is a queue structure containing threads blocked on a lock
- The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object itself.
- Subsequent threads blocking on the lock will be added to this turnstile

- To prevent a priority inversion, turnstiles are organized according to a priority inheritance protocol
- This means that if a lowerpriority thread currently holds a lock that a higher-priority thread is blocked on, the thread with the lower priority will temporarily inherit the priority of the higher-priority thread.

Synchronization in Windows 2000:

- On a multiprocessor system, Windows 2000 protects access to global resources using spinlocks
- the kernel only uses spinlocks only to protect short code segments
- for thread synchronization outside of the kernel, Windows 2000 provides dispatcher objects.
- Using a dispatcher object, a thread can synchronize according to several different mechanisms including mutexes, semaphores, and events.
- Shared data can be protected by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished.
- Events are a synchronization mechanism that may be used much as are condition variables; that is, they may notify a waiting thread when a desired condition occurs.

- Dispatcher objects may be in either a signaled or nonsignaled state.
- A signaled state indicates that an object is available and a thread will not block when acquiring the object.
- A nonsignaled state indicates that an object is not available and that a thread will block when attempting to acquire the object
- When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting and the thread is placed in a waiting queue for that object.
- When the state for the dispatcher object moves to signaled, the kernel checks if there are any threads waiting on the object.
- If so, the kernel moves one-or possibly more-threads from the waiting state to the ready state where they can resume executing.
- Let us use a mutex lock as an illustrating example of dispatcher objects and thread states.
- If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object.
- When the mutex moves to the signaled state (the result of another thread releasing the lock on the mutex), the thread waiting on the mutex will:

  1. Be moved from the wait to the ready state,

  2. Acquire the mutex lock.

Atomic Transactions

- The mutual exclusion of critical sections ensures that the critical sections are executed atomically.
- Databases are concerned with the storage and retrieval of data, and with the consistency of the data.
- System Model:
- collection of instructions (or operations) that performs a single logical function is called a transaction.
- A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system
- A transaction is a program unit that accesses and possibly updates various data items that may reside on the disk within some files.
- From our point of view, a transaction is simply a sequence of read and write operations, terminated by either a commit operation or an abort operation.
- A commit operation signifies that the transaction has terminated its execution successfully, whereas an abort operation signifies that the transaction had to cease its normal execution due to some logical error.
- A terminated transaction that has completed its execution successfully is committed; otherwise, it is aborted

- the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing.
- We say that such a transaction has been rolled back.
- Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.
- Volatile Storage: Information residing in volatile storage does not usually survive system crashes
- Nonvolatile Storage: Information residing in nonvolatile storage usually survives system crashes
- Stable Storage: Information residing in stable storage is never lost

- **Log-Based Recovery :**
- The most widely used method for achieving this form of recording is write ahead logging.
- The system maintains, on stable storage, a data structure called the log.
- Each log record describes a single operation of a transaction write, and has the following fields:
- Transaction Name: The unique name of the transaction that performed the write operation
- Data Item Name: The unique name of the data item
- written Old Value: The value of the data item prior to the write operation
- New Value: The value that the data item will have after the writ
- Before a transaction Ti starts its execution, the record < Ti starts> is written to the log.
- During its execution, any write operation by Ti is preceded by the writing of the appropriate new record to the log.
- When Ti commits, the record < Ti commits> is written to the log
- we cannot allow the actual update to a data item to take place before the corresponding log record is written out to stable storage.
- We therefore require that, prior to a write(X) operation being executed, the log records corresponding to X be written onto stable storage

- Note the performance penalty inherent in this system. Two physical writes are required for every logical write requested
- The recovery algorithm uses two procedures:
- undo(Ti), which restores the value of all data updated by transaction Ti to the old values
- redo(Ti), which sets the value of all data updated by transaction Ti to the new values
- This classification of transactions is accomplished as follows:
- Transaction Ti needs to be undone if the log contains the < Ti starts> record, but does not contain the < Ti commits> record.
- Transaction Ti needs to be redone if the log contains both the < Ti starts> and the < Ti commits> records.

- Checkpoints :
- When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone
- There are two major drawbacks to this approach:

  1. The searching process is time-consuming.

  2. Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need modify.
- the system periodically performs checkpoints that require the following sequence of actions to take place:

  1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.

  2. Output all modified data residing in volatile storage to the stable storage.

  3. Output a log record onto stable storage.

- The recovery operations that are required are as follows:
- a For all transactions Tk in T such that the record < Tk commits, appears in the log, execute redo(Tk).
- a For all transactions Tk in T that have no < Tk commits> record in the log, execute undo(Tk).

Concurrent Atomic Transactions

- each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions executed serially in some arbitrary order. This property called serializability
- All transactions share a common semaphore mutex, which is initialized to 1.
- When a transaction starts executing, its first action is to execute wait(mutex).
- After the transaction either commits or aborts, it executes signal(mutex).

- Serializability:
- Consider a system with two data items A and B that are both read and written by two transactions To and TI.
- Suppose that these transactions are executed atomically in the order To followed by TI.
- This execution sequence, which is called a schedule,
- A schedule where each transaction is executed atomically is called a serial schedule.
- if we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial.
- A nonserial schedule does not necessarily imply that the resulting execution is incorrect (that is, is not equivalent to a serial schedule).

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

Schedule 1: A serial schedule in which T_0 is followed by T_1.

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

Schedule 2: A concurrent serializable schedule.

- if a schedule S can be transformed into a serial schedule S' by a series of swaps of non conflicting operations, we say that a schedule S is conflict serializable.
- Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1.

- Locking Protocol:
- two modes:
- Shared: If a transaction Ti has obtained a shared-mode lock (denoted by S) on data item Q, then Ti can read this item, but cannot write Q.
- Exclusive: If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on data item Q, then Ti can both read and write Q.
- One protocol that ensures serializability is the two-phase locking protocol.
- This protocol requires that each transaction issue lock and unlock requests in two phases:
- 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.
- Initially, a transaction is in the growing phase. The transaction acquires locks as needed.
- Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

- Timestamp-Based Protocols:
- each transaction Ti in the system, we associate a unique fixed time stamp, denoted by TS(Ti).
- This timestamp is assigned by the system before the transaction Ti starts execution.
- two simple methods for implementing this scheme:
- Use the value of the system clock as the timestamp;
- Use a logical counter as the timestamp;
- To implement this scheme, we associate with each data item Q two timestamp values:
- W-timestamp(Q), which denotes the largest timestamp of any transaction that executed writ e(Q) successfully
- R-timestamp(Q), which denotes the largest timestamp of any transaction that executed read(Q) successfully

- The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- This protocol operates as follows:
- Suppose that transaction Ti issues read(Q):
- If TS(Ti) < W-timestamp(), then this state implies that Ti needs to read a value of Q that was already overwritten.
- Hence, the read operation is rejected, and Ti is rolled back.
- If TS(Ti) > W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).
- Suppose that transaction Ti issues write(Q):
- If TS(Ti) < R-timestamp(Q), then this state implies that the value of Q that Ti is producing was needed previously and Ti assumed that this value would never be produced.
- Hence, the write operation is rejected, and Ti is rolled back.
- If TS(Ti) < W-timestamp(Q), then this state implies that Ti is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and Ti is rolled back.
- Otherwise, the write operation is executed

Thank YOU