```
Bankers Algo
C = Claim matrix, A = Allo matrix,
R = Resource vector, A = available vector
    [3 2 2]      [1 0 0]       [2 2 2]
C = [6 1 3]  A = [6 1 2]  C-A = [0 0 1]  R = [9 3 6]
    [3 1 4]      [2 1 1]       [1 0 3]
    [4 2 2]      [0 0 2]       [4 2 0]

V = R - A = [9 3 6] - [9 2 5] = [0 1 1]

C-A_row <= V, True, then V = V + A_row
Safe State -> V = R

C-A_1: [2 2 2] > [0 1 1] so fail
C-A_2: [0 0 1] < [0 1 1] so, new V = [0 1 1] + [6 1 2] = [6 2 3]
C-A_3: [1 0 3] < [6 2 3] so, new V = [6 2 3] + [2 1 1] = [8 3 4]
C-A_4: [4 2 0] < [8 3 4] so, new V = [8 3 4] + [0 0 2] = [8 3 6]

Since not all process were finished, go back to failed PL:
C-A_1: [2 2 2] < [8 3 6], so new V = [8 3 6] + [1 0 0] = [9 3 6]

R = V ? TRUE !
```

```
WOLFEY'S SOLUTION
#include <cmath>
#include <fcntl.h>
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

static pthread_mutex_t bsem;
static pthread_cond_t rincon = PTHREAD_COND_INITIALIZER;
static pthread_cond_t castro = PTHREAD_COND_INITIALIZER;

bool rinconGo = true;

bool isRincon(char *member) { return strcmp(member, "RINCON") == 0; }

void outputMember(char *member) {
  std::cout << member << " member inside the house" << std::endl;
  std::cout << member << " member leaving the house" << std::endl;
}

void *access_one_at_a_time(void *family_void_ptr) {
  char *member = (char *)family_void_ptr;
  bool am_i_rincon = isRincon(member);

  pthread_mutex_lock(&bsem);
  if (rinconGo != am_i_rincon) {
    pthread_cond_wait(am_i_rincon ? &rincon : &castro, &bsem);
  }

  outputMember(member);
  rinconGo = !rinconGo;
  pthread_cond_signal(am_i_rincon ? &castro : &rincon);
  pthread_mutex_unlock(&bsem);

  return NULL;
}

int main() {
  int nmembers;
  std::cin >> nmembers;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
  pthread_t *tid = new pthread_t[nmembers];
  char **family = new char *[nmembers];
  for (int i = 0; i < nmembers; i++) {
    family[i] = new char[20];
    strcpy(family[i], (i % 2 ? "CASTRO" : "RINCON"));
  }

  for (int i = 0; i < nmembers; i++) {

    pthread_create(&tid[i], NULL, access_one_at_a_time, family[i]);
    // Write the code to call pthread_create
  }

  // Wait for the other threads to finish.
  for (int i = 0; i < nmembers; i++)
    pthread_join(tid[i], NULL);

  for (int i = 0; i < nmembers; i++)
    delete[] family[i];
  delete[] family;
  delete[] tid;
  return 0;
}
```

CHAPTER 5 MUTUAL EXCLUSION

The central themes of operating system design are all concerned with the management of processes and threads. - true
It is possible in a single-processor system to not only interleave the execution of multiple processes but also to overlap them. - false
As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes. - true
Race condition is a situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work. - false
The sharing of main memory among processes is useful to permit efficient and close interaction among processes because such sharing does not lead to any problems. - false
When processes cooperate by communication, the various processes participate in a common effort that links all of the processes. - true
Atomicity guarantees isolation from concurrent processes. - true
Concurrent processes do not come into conflict with each other when they are competing for the use of the same resource. - false
Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. - true
The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes. - true
A process that is waiting for access to a critical section does not consume processor time. - false
The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. - true
It is possible for one process to lock the mutex and for another process to unlock it. - false
One of the most common problems faced in concurrent processing is the producer/consumer problem. - true
Processes need to be synchronized to enforce mutual exclusion. - true
The management of multiple processes within a uniprocessor system is _____ .

A) multiprogramming
B) structured applications
C) distributed processing
D) multiprocessing
A situation in which a runnable process is overlooked indefinitely by the scheduler, although it is able to proceed, is _____ .

A) mutual exclusion
B) deadlock
C) starvation
D) livelock -
The requirement that when one process is in a critical section that accesses shared resources,
no other process may be in a critical section that accesses any of those shared resources is _____ .

A) critical section
B) livelock
C) mutual exclusion
D) atomic operation
A means for two processes to exchange information is with the use of _____ .

A) spinlocks
B) event flags
C) condition variables
D) messages
A semaphore that does not specify the order in which processes are removed from the queue is a _____ semaphore.

A) weak
B) general
C) strong
D) binary
A _____ occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

A) atomic operation
B) race condition
C) livelock
D) deadlock
A _____ is an integer value used for signaling among processes.

A) semaphore
B) message
C) mutex
D) atomic operation -

_____ is when the sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state.

A) Critical section
B) Mutual exclusion
C) Atomic operation
D) Starvation
_____ are memory words used as a synchronization mechanism.

RINCON'S PRACTICE EXAM SOLUTION
```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

static pthread_mutex_t bsem;
static pthread_cond_t rincon = PTHREAD_COND_INITIALIZER;
static pthread_cond_t castro = PTHREAD_COND_INITIALIZER;
static char turn[] = "RINCON";
static bool busy =false;

void *access_one_at_a_time(void *family_void_ptr)
{

                pthread_mutex_lock(&bsem);
                char fam[20];
                strcpy(fam,(char *) family_void_ptr);
                while (busy == true || strcmp(fam,turn)!=0)
  {
    if(strcmp(fam,"RINCON")==0)
                                pthread_cond_wait(&rincon, &bsem);
    else
      pthread_cond_wait(&castro, &bsem);
  }
  busy = true;
                std::cout << fam << " member inside the house\n";
                pthread_mutex_unlock(&bsem);

  usleep(100);

                pthread_mutex_lock(&bsem);
                std::cout << fam << " member leaving the house\n";
  busy = false;
                if (strcmp(turn,"RINCON") == 0)
  {
    strcpy(turn,"CASTRO");
                                pthread_cond_signal(&castro);
  }
  else
  {
    strcpy(turn,"RINCON");
    pthread_cond_signal(&rincon);
  }
                pthread_mutex_unlock(&bsem);
                return NULL;
}

int main()
{
  int nmembers;
  std::cin >> nmembers;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
                pthread_t *tid= new pthread_t[nmembers];
                char **family=new char*[nmembers];
  for(int i=0;i<nmembers;i++)
    family[i]=new char[20];
                for(int i=0;i<nmembers;i++)
                {
                        if(i%2 == 0)
                                                strcpy(family[i],"RINCON");
                        else
                                                strcpy(family[i],"CASTRO");
                  if(pthread_create(&tid[i], NULL, access_one_at_a_time,(void *)family[i]))
                                {
                                                fprintf(stderr, "Error creating thread\n");
                                                return 1;
                                }

                }
                // Wait for the other threads to finish.
                for (int i = 0; i < nmembers; i++)
                pthread_join(tid[i], NULL);
  for(int i=0;i<nmembers;i++)
    delete [] family[i];
  delete [] family;
  delete [] tid;
                return 0;
}
```

A) Semaphores
B) Event flags
C) Counting semaphores
D) Mailboxes - B. Event flags
The term _____ refers to a technique in which a process can do nothing until it gets permission to enter its critical section but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance.

A) spin waiting
B) general semaphore
C) critical resource
D) message passing
A _____ is a data type that is used to block a process or thread until a particular condition is true.

A) deadlock
B) general semaphore
C) condition variable
D) mutex -
A semaphore whose definition includes the policy that the process that has been blocked the longest is released from the queue first is called a _____ semaphore.

A) general
B) strong
C) weak
D) counting
The _____ is a programming language construct that provides equivalent functionality to that of semaphores and is easier to control.

A) atomic operation
B) coroutine
C) critical section
D) monitor
Probably the most useful combination, _____ allows a process to send one or more messages to a variety of destinations as quickly as possible.

A) blocking send, blocking receive
B) nonblocking send, blocking receive
C) nonblocking send, nonblocking receive
D) blocking send, nonblocking receive
A _____ relationship allows multiple server processes to provide concurrent service to multiple clients.

A) many-to-many
B) one-to-many
C) many-to-one
D) one-to-one –

RINCON'S SAMPLE CODE OF CONDITION VARIABLE

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define NTHREADS 5
#define FAMILYNAME "RINCON"


static pthread_mutex_t bsem;
static int members;
static pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

void *access_house(void *family_void_ptr)
{
                pthread_mutex_lock(&bsem);
                char fam[20];
                strcpy(fam,(char *) family_void_ptr);
                printf("%s member arrives to the house\n", fam);
                if (strcmp(fam,FAMILYNAME)!=0)
                                pthread_cond_wait(&empty, &bsem);
                members++;
                printf("%s member inside the house\n", fam);
                pthread_mutex_unlock(&bsem);

                sleep(5);

                pthread_mutex_lock(&bsem);
                printf("%s member leaving the house\n", fam);
                members--;
                if (strcmp(fam,FAMILYNAME) == 0 && members == 0)
                                pthread_cond_broadcast(&empty);
                pthread_mutex_unlock(&bsem);
                return NULL;
}

int main()
{

                pthread_t tid[NTHREADS];
                static int members=0;
                char family[NTHREADS][20];
                pthread_mutex_init(&bsem, NULL); // Initialize access to 1

                for(int i=0;i<NTHREADS;i++)
                {
                                if(i%2 == 0)
                                                strcpy((char *) &family[i],"RINCON");
                                else
                                                strcpy((char *)&family[i],"CASTRO");
                                if(pthread_create(&tid[i], NULL, access_house,(void *)&family[i]))
                                {
                                                fprintf(stderr, "Error creating thread\n");
                                                return 1;

                                }
                                sleep(1);
                }
                // Wait for the other threads to finish.
                for (int i = 0; i < NTHREADS; i++)
                pthread_join(tid[i], NULL);
                return 0;
}
```

CHAPTER 6 DEADLOCK AND STARVATION
Both process interleaving and process overlapping are examples of concurrent processes and both present the same basic problems. - True
T / F - Concurrency issues are a concern on multiprocessor systems, but do not impact uniprocessor systems. - False, impact both types of systems.
The following requirement(s) must be met by any facility or capability that is to provide support for mutual exclusion: - 1. Only one process at a time can be allowed

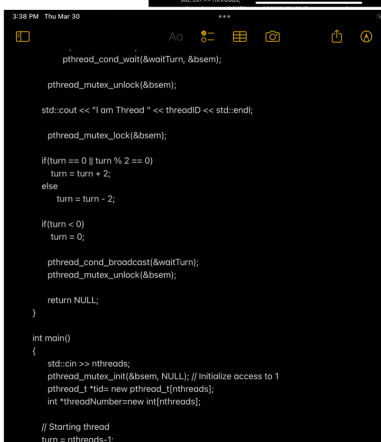
into a critical code section
2. A process remains in its critical code section
for a finite time only
3. No assumptions can be made about relative
process speeds
In a uniprocessor system, mutual exclusion can be guaranteed by: - Disabling interrupts
The situation where Process 1 (P1) holds Resource 1 (R1), while P2 holds R2, and P1 needs R2 to complete and P2 needs R1 to complete is referred to as _____ . - deadlock
When only one process is allowed in its critical code section at a time, then _____ is enforced. - mutual exclusion
In multiprocessor configurations, special machine instructions that carry out two actions in a single instruction cycle are said to do so _____ . - atomically
The technique in which a process can do nothing until it gets permission to enter its critical section but continues to test the appropriate variable to gain entrance is called _____ . - busy waiting
A semaphore that does not specify the order in which processes are removed from the queue is called a: - Weak semaphore
Weak semaphores guarantee freedom from starvation, but strong semaphores do not. - False, opposite is true.
a finite circular buffer and an infinite buffer are two ways to implement a data storage area for the classic Producer / Consumer Problem. - False, only finite buffer
The major difficulty with semaphores is that wait and signal operations may be scattered throughout a program and it is difficult to see the overall effect of these operations on the semaphores they affect. - True
The classic concurrency problem that involves multiple readers that can read from shared data area when no single writer is exclusively writing to it is the _____ problem. - Readers / Writers
All deadlocks involve conflicting needs for resources by: - Two or more processes
An example of a consumable resource is the following: - Messages
T / F - Although deadlock can potentially exist without it, the condition known as Circular Wait is required for deadlock to actually take place. - I dont have this sheet, but the possibility of deadlock exists without circular wait, but the existence of deadlock means there is a circular wait, so I guess its true?
T / F - The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is minimized. - False
T / F - The Deadlock Avoidance approach to solving the deadlock problem allows the three necessary conditions for deadlock to exist. - True
T / F - In the Resource Allocation Denial approach to Deadlock Avoidance, a safe state is defined as one in which all potential process sequences do not result in a deadlock. - true
T / F - Although each strategy that deals with deadlocks has its advantages and disadvantages, the best solution to the problem is to choose one and stick with it. - False, a combination is best
The Dining Philosophar's Problem is a standard test case for evaluating approaches to implementing - Synchronization
A programming-language construct that provides equivalent functionality to that of a semaphore and that is easier to control - Monitor
Two fundamental requirements that must be satisfied for process interaction: - Synchronization and Communication
The problem where a data area is shared among many processes, some processes only read the data area, some only write to the area. - Readers/Writers Problem
3 Conditions to satisfy the Readers/Writers Problem - 1. Multiple readers may read the file at once. 2. Only one writer at a time may write. 3. If a writer is writing to the file, no reader may read it.
The condition where only one process may use a resource at a time. - Mutual Exclusion
The condition where a process may hold allocated resources while awaiting assignment of other resources - Hold-and-wait
The condition where no resource can be forcibly removed from a process that is holding it. - No Pre-emption
3 conditions for possible deadlock: - Mutual Exclusion, Hold-and-wait, No Pre-emption
The condition in which a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain. - Circular Wait
Three approaches for dealing with deadlock: - Prevention, Avoidance, and Detection
This approach to dealing with deadlock eliminates one of more of the conditions required for deadlock to be possible: - Deadlock Prevention
Method of deadlock prevention in which all three necessary conditions are prevented from occuring at once: - Indirect Deadlock Prevention, examples: Hold-and-wait - require a process request all of its required resources at once time
Method of deadlock prevention in which circular waits are prevented - Direct Deadlock Prevention, example: Define linear order of resource types Given (R1, R2) and (Type R1), Given R2) Q requests R1 is not possible
Problems with deadlock avoidance: - 1. Inefficient use of resources. 2. Inefficient execution of processes.
This approach to dealing with deadlock makes dynamic decisions whether the current resource allocation request will, if granted, potentially lead to deadlock - Deadlock Avoidance
This method of deadlock avoidance does not start a process if its demands might lead to deadlock - Process Initiation Denial - A process is only started if the maximum claim of all current processes plus those of the new process can be met. • Not optimal!!!
This method of deadlock avoidance does not grant an incremental resource request to a process if this allocation might lead to deadlock. - Resource Allocation Denial - Banker's algorithm
Deadlock Avoidance Advantages: • It is not necessary to preempt and rollback prkocesses, as in deadlock detection
• It is less restrictive than deadlock
prevention
Deadlock Avoidance Restrictions: • Maximum resource requirement must be
stated in advance • Processes under consideration must be
independent and with no synchronization
requirements
• There must be a fixed number of
resources to allocate
• No process may exit while holding
resources

A deadlock avoidance mechanism requires knowledge of future process requests. True
When a thread calls a signal over a condition variable, if there is no waiting thread on the signaled condition variable, this signal is lost. True
In message passing, a solution based on mailboxes uses direct addressing. False
A disadvantage of the deadlock algorithm is that frequent checks consume considerable processor time. True
In deadlock avoidance, the solution is executed after assigning the resources to a process. False
Peterson's algorithm is a hardware-based solution to guarantee mutual exclusion. False
Select the concurrency mechanism that is a hardware solution. Exchange Instruction
Deadlock Prevention Approach: Requesting all resources at once.
Select the option that is not a requirement for mutual exclusion: Using the relative process speeds or the number of processes as parameters to guarantee mutual exclusion.
In the deadlock detection algorithm, if all processes are marked, then: No deadlock was detected.
Select the option that is not a recovery strategy of the deadlock detection algorithm: None of the above.

Critical Section is the lock and unlock pairing, so if there's a
pthread_mutex_lock
// shared resources go here;
pthread_mutex_unlock
--> Then this is one critical section, and the value(s) that changes in between the critical section is the shared resources.

USING CONDITION TO PRINT IN CERTAIN ORDER

```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

static pthread_mutex_t bsem;
static pthread_cond_t waitTurn = PTHREAD_COND_INITIALIZER;
static int order;

void *print_in_reverse_order(void *void_ptr_argv)
{

                pthread_mutex_lock(&bsem);
                int *index = (int *) void_ptr_argv;
                while((*index) != order)
                  pthread_cond_wait(&waitTurn, &bsem);

                  std::cout<< "I am Thread " << (*index) << std::endl;
                  order--;
                  pthread_cond_broadcast(&waitTurn);
                  pthread_mutex_unlock(&bsem);
                return NULL;
}

int main()
{
  int nthreads;
  std::cin >> nthreads;
  pthread_mutex_init(&bsem, NULL); // Initialize access to 1
  order = nthreads - 1;
                pthread_t *tid= new pthread_t[nthreads];
                int number[nthreads];
                int *threadNumber=new int[nthreads];
  for(int i=0;i<nthreads;i++){
    number[i]= i;
    if(pthread_create(&tid[i], NULL, print_in_reverse_order,&number[i]))
                                {
                                                fprintf(stderr, "Error creating thread\n");
                                                return 1;

                                }

  }
                // Wait for the other threads to finish.
                for (int i = 0; i < nthreads; i++)
                pthread_join(tid[i], NULL);

  delete [] threadNumber;
  delete [] tid;
                return 0;
}
```



Pipes a pipe is a technique for passing information from one program process to another. Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only.
pros and cons of pipes Pros:
-Simple
-Flexible
-Efficient Communication
Cons:
-No way to open an already existing pipe. This makes it impossible for two arbitrary processes to share the same pipe, unless the pipe was created by a common ancestor process
procedure for using shared memory - find a key
-use shmget() to allocate a shared memory
-use shmat() to attach a shared memory to an address space
-use shmdt() to detach a shared memory from an address space
-use shmctl() to deallocate a shared memory
Message passing Communicating between multiple processors by explicitly sending and receiving information.
requires:
-One send
send(addr, msg,length);
-One receive(addr,msg,length);
Advantages of message passing - Very general
- senders and receivers can be on different machines
-relatively secure
-receiver can inspect the messages it has received before processing them
Disadvantages of message passing -Hard to use
-every data transfer REQUIRES send() and receive()
-Receiving process MUST expect the send()
- might require forking a special thread
shared memory two or more processes share a part of their address space
Advantages of shared memory - Fast and easy to use
BUT
-concurrent accesses to the shared data can result into small disasters
-MUST synchronize access to shared data
Disadvantages of shared memory Not a general solution
-sender and receiver MUST be on same machine
Less secure
-Processes can directly access a part of the address space of other processes
Defining issues of message passing-Direct/indirect communication
-Blocking/non blocking primitives
-exception handling
-quality of service
direct communication -send and receive system calls ALWAYS specify the process to destination or source
-no intermediary between sender and receiver
what must a process know when executing the receive call of direct communication? process must know the identity of all processes that are likely to send a message
-this is a very bad solution for servers as servers have to answer request from arbitrary processes
indirect communication send and receive must specify an intermediary entity as destination or source
Mailbox created by the kernel at request of user process as an intermediary entity for indirect communication
mailbox calls send(mailbox, msg, size);
recv(mailbox,msg,size);
sending and receiving of mailbox-different processes can send messages to same mailbox
-a process can receive message from another process without having to know it
- a process can wait for messages coming from different senders and will answer the first message received
mailboxes can be PRIVATE
-attached to a specific process (Ex, cellphone)
PUBLIC
-attached to system objects (ex. homephone)
Private mailbox -Process that requested its creation and its children are the only process that can receive messages through the mailbox are that process and its children
-Cease to exist when the process that requested the creation terminates
-also called PORTS
Public mailbox -shared by all the processes having the right to receive messages through it
-works best when all processes are on same machine
-survive termination of the process that requested the creation
Blocking send -the sender is blocked until the message is received
does not return until the receiving process has received the message
-no buffer needed
ex. when you call someone that doesnt have a voicemail set up
blocking receive-the receiver is blocked until a message is available
-does not return until a message has been received
ex. waiting by phone for an important messages or standing by mailbox all day waiting for mailman
Non-blocking send the sending process sends the message and resumes operation
-returns as soon as message has been accepted by delivery for OS
Non-blocking receive Receiver returns after retrieving either a valid message or null
ex. checking whether your mail has arrived or not
busy wait simulates a block receive with a non blocking receive within a loop
simulating blocking sends with two non-blocking sends -sender sends message and requests an acknowledgement(ACK)
-sender waits for ACK from receiver using a blocking receive
-receiver sends ACK
when it comes to blocking/non blocking, we prefer -Indirect naming
-Non blocking sends
sender does not care what happens after message gets sent
-Blocking receives
receiver needs data to continue
Buffering -required by non blocking primitives to let the OS store messages that have been sent but not yet received somewhere
Bounded capacity- when the buffer can take only a certain amount of messages
unlimited capacity when the buffer can receive unlimited amount of capacity
-buffers are theoretically unlimited capacity
Blocking and receive does not go well with direct communication Processes cannot wait for messages from several sources without using special parallel programming constructs:
Dijkstra's alternative command
using blocking receives with direct naming: a bad combination
does not allow the receiving process to recieve any messages from any process except for the one that's specified
exception condition handling- Must specify what to done if one of the two processes dies
-important whenever two processes are on two different machines
must handle:
-host failures
-network partitions
Duckle of access when sender and receiver are on different machines -can be lost, corrupted or duplicated
-arrive out of sequence
BUT can still decide to provide reliable message delivery
positive acknowledgement (ACK) -Best technique for providing reliable delivery of messages
-Destination sends an acknowledgement message for every message correctly delivered
-damaged messages are ignored
- sender resends any messages that has not been acknowledged within a fixed time frame
rule of acknowledgement Receiver must acknowledge the second time a message has been sent otherwise it would be resent ONE MORE TIME.
acknowledge any message that does not need to be resent
datagrams messages are sent one at a time
-some messages can be lost, others duplicated or arrive out of sequence
Virtual Circuits -ordered sequence of messages
-connection oriented service
-Require setting up a virtual connection BEFORE sending data
-costlier than datagrams
-best for transmitting large amounts of data that requires sending several messages
ex. FTP, HTTP
streams -ordered sequence of bytes
-message boundaries are ignored
-similar to virtual circuits

-do not preserve message boundaries
-receiver sees a seamless stream of bytes
-record boundaries dont count
-messages boundaries should not count
reliable datagrams unsent until they acknowledged
unreliable datagrams -messages are not acknowledged
-works well when message request a reply
-reply is implicit acknowledgment of a message
reason to request is when it might take a long time to reply to it
User Datagram Protocol (UDP) -Best known datagram protocol
-Provides an unreliable datagram service
-best for short interactions (one request and one reply)
Transmission Control Protocol (TCP) -best known stream protocol
-provides a reliable stream service
-heavy weight
requires three messages (packets) to establish a virtual connection
Datagrams vs Stream
Datagrams:
-unreliable
-not ordered
-lightweight
-deliver messages
ex. UDP
Stream:
-reliable
-ordered
-heavyweight
-stream oriented
ex. TCP
busy wait -a process synchronization technique in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution.
Remote Procedure Call (RPC) - A protocol that enables a process on one computer to call a process on another computer.
-Apply to client-server mode of computation
advantages of RPC - -Hides all details of message passing
Programmer can focus on the logic of her application
-Provides a higher level of abstraction
-Extends a well known programming model
-Anybody that can use procedures and function can quickly learn to use remote procedure calls
Disadvantages of RPC - -The illusion is not perfect
-RPCs do not always behave exactly like regular procedure calls
-Client server do not share the same address space
-Programmer must remain aware of these subtle and not so subtle differences
user program - contains the user code
calls the user stub
- rpc(xyz,args, &result);
Appears to call the server procedure
user stub - Procedure generated by RPC package:
-Packs arguments into request message and performs required data conversions (argument marshaling)
-sends request message
-waits for server's reply message
-unpacks the results and performs required data conversions (argument unmarshaling)
argument unmarshaling - Unpacks results and performs required data conversions
the server stub - Generic server generated by RPC package:
-waits for client requests
-unpacks requests argument and performs required data conversions
-calls appropriate server procedure
-packs the results into reply message and performs required data conversions
-send reply message
argument marshaling - Packs arguments into request message and performs required data conversions
server procedure - -Procedure called by the server stub
-written by the user
-Does the actual processing of user requests
Differences with regular PC - -Client and server processes do NOT share the same address space
-Client and server can be on different machines
-Must handle partial failures
No shared address space - means:
No global variables
-Cannot pass addresses
-Cannot pass arguments by reference
-Cannot pass dynamic data structures through pointers
The solution - RPC can pass arguments by value and result
-Pass the current value of the argument to the remote procedure
-Copy the returned value in the user program
-Not the same as passing arguments by reference
Passing dynamic types - -you can NOT pass dynamic data structures through pointers
-must send a copy of the data structure
-For a linked in list:
send array with elements of linked list plus unpacking instructions
little endians - start at the LEAST significant byte
ex- Intel's 80x86 including pentium
Big endians - start at the MOST significant byte
ex. Sun's SPARC and most RISC processors
Solution to architecture considerations - -Define a network order and convert all numerical variables to that order
-use htonl family of functions
-Same as requiring all air traffic control communications to be in english
Detecting partial failures - -The client must detect server failures
-Can send "Are you alive?" messages to the server at fixed time intervals
Handling partial executions - -Client must deal with possibility that the server could have crashed after having partially executed the request
ex. ATM machine using the bank computer
was the account debited or not?
first solution to handling partial executions - -Ignore the problem and always resubmit requests that have not been answered
Some requests may be executed more than once
-Will work if requests are idempotent
Executing them several times has the same effect as executing them exactly once
Technique is used by all RPCs in the Sun Microsystems' Network File System (NFS)
first solution to handling partial executions - -Attach a serial number to each request
-Server can detect replays of requests it has previously received and refuse to execute them
-At most once semantics
cheap but not perfect
some requests could end up being partially executed
third solution of handling partial execution : Use a transaction mechanism
-Guarantees that each request will either be fully executed or have no effect
All or nothing semantics
-Best and costliest solution
-use it in all financial transactions
SUN RPC - -Developed by sun microsystems
-used to implement their network file system
MSRPC (Microsoft RPC) - -Proprietary version of the DCE/RPC protocol
-Was used in the Distributed Component Object Model(DCOM)
SOAP - -Exchanges XML- based messages
-Runs on the top of HTTP:
-very portable
-very verbose

JSON-RPC - uses Javascript object notation (JSON)
multiple processes - Operating system design is concerned with the management of processes and threads:
-multiprogramming
-multiprocessing
-Distributed processing
multiple applications - invented to allow processing time to be shared among active applications
Structured applications - an extension of the principles of modular design and structured programming.
some applications can be effectively programmed as a set of concurrent processes.
operating system structure - OS themselves implemented as a set of processes or threads
atomic operation - A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
Critical Section - A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock - a situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something
livelock - A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
Mutual Exclusion - The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition - -Occurs when multiple processes or threads read and write data items
-The final result depends on the order of execution
The "loser" of the race is the process that updates last and will determine the final value of the variable
starvation - A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.
Principles of concurrency - Interleaving and overlapping
- can be viewed as examples of concurrent processing
- both present the same problems

Uniprocessor: the relative speed of execution of processes cannot be predicted
- depends on the activities of other processes
- the way the OS handles interrupts
- scheduling policies of the OS
Difficulties of Concurrency: - Sharing of global resources
-Difficult for the OS to manage the allocation of resources optimally
-Difficult to locate programming errors as results are not deterministic and reproducible
Operating system concerns: - -Design and management issues raised by the existence of concurrency:
-The OS must:
be able to keep track of various processes
-allocate and deallocate resources for each active process
-protect the data and physical resources of each process against interference by other processes
-ensure that processes and outputs are independent of the processing speed

resource competition - Concurrent processes come into conflict when they are competing for use of the same resource

In case of the competing processes three control problems must be faced:
-the need for mutual exclusion
-deadlock
-starvation
requirements for mutual exclusion - -Must be enforced
-A process that halts must do so without interfering with other processes
-No deadlock or starvation
-A process must not be denied access to a critical section when there is no other process using it
-No assumptions are made about relative process speeds or number of processes
-A process remains inside its critical section for a finite time only
Interrupt Disabling - -uniprocessor system
-disabling interrupts guarantees mutual exclusion
Disadvantages of Interrupt Disabling - -the efficiency of execution could be noticeably degraded
-this approach will not work in a multiprocessor architecture
Advantages of special machine instruction
- -Applicable to any number of processes on either a single processor or multiple processors sharing main memory
-Simple and easy to verify
-Can be used to support multiple critical sections; each critical sections can be defined by its own variable
Disadvantages of special machine instruction :
- -Busy waiting is employed, thus while a process is waiting for access to a critical section to continues to consume processor time
-starvation is possible when a process leaves a critical section and more than one process is waiting
-deadlock is possible
semaphore - a variable that has an integer value upon which only three operations are defined -> there is no way to inspect or manipulate sempahores other than these three operations
may be initialized to a nonnegative integer value
semWait operation - decrements the value
semSignal operation - increments the value
consequences of semaphore- 1. There is no way to know before a process decrements a semaphore whether it will block or not
2. There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently
3. You dont know whether another process is waiting so then umber of unblocked processes may be zero or one
strong semaphores - the process that has been blocked the longest is released from the queue first
weak semaphores - the order in which processes are removed from the queue is not specified
mutex - used to hold processes waiting on the semaphore
Producer-Consumer Problem - General situation
-one or more producers are generating data and placing these in a buffer
- a single consumer is taking items out of the buffer one at a time
-only one producer or consumer may access the buffer at any one time
The problem
-ensure that the producer cant add data into full buffer and consumer cant remove data from an empty buffer
Implementation of Semaphores - -Imporative that the semWait and semSignal operations be implemented as atomic primitives
-can be implemented in hardware or firmware
-Software schemes such as Dekker's or Peterson's algorithms can be used
-Use one of the hardware-supported schemes for mutual exclusion
Monitors - - programming language construct that provides equivalent functionality to that of sempahores and is easier to control
-Implemented in Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3 and Java
-Has also been implemented as a program library
- Software module been consisting of one or more procedures, an initialization sequence and local data
Monitor Characteristics - -Local data variables are accessible only by the monitor's procedures and not by any external procedure
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time
synchronization - -Achieved by the use of condition variables that are contained within the monitor and accessible within the monitor
Condition variable functions –
-cwait
csignal
cwait(c) - suspend execution of calling process on condition c
csignal(c) - resume execution of some process blocked after a cwait on the same condition
Fundamental requirements of message passing - synchronization and communication
synchronization of message passing - to enforce mutual exclusion
communication of message passing - to exchange information
Blocking send, Blocking receive - -Both the sender and receiver are blocked until the message is delivered;
-this is sometimes referred to as a rendezvous .
- This combination allows for tight synchronization between processes.
Nonblocking send, blocking receive - Sender continues on but receiver is blocked until the requested message arrives,
-most useful combination,
sends one or more messages to a variety of destinations as quickly as possible,
example -- a service process that exists to provide a service or resource to other processes
Nonblocking send, nonblocking receive - neither party is required to wait
Direct Addressing - The send primitive includes a specific identifier of the destination process
The receive primitive can be handled in one of two ways:

-require that the process explicitly designate a sending process
effective for cooperating concurrent processes
-implicit addressing
source parameter of the receive primitive possesses a value returned when the receive operation has been performed
Indirect Addressing - 1.Messages are sent to a shared data structure consisting of queues that can temporarily hold messages
2.Queues are referred to as mailboxes,
3. One Process sends a message to the mailbox and the other process picks up the message from the mailbox, 4.Allows for greater flexibility in the use of messages.
Reader/Writer Problem - -A data area is shared among many processes
-some processes only read the data area (readers)
and some only write to the data area (writers)
Conditions that must be satisfied in Readers/Writers Problem –
1. any number of readers may simultaneously read the file
2. only one writer at a time may write to the file
3. if a writer is writing to the file, no reader may read it

DEADLOCK AND STARVATION
Deadlock - -The permanent blocking of a set of processes that either communicate for system resources or communicate with each other
-A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
-no efficient solution
-Permanent
Resource Categories reusable and consumable
Reusable Resource -Can be safely used by only one process and is not depleted by that use
ex. processors, I/O channels, main and secondary memory, devices and data structures such as files, databases and semaphores
Consumable resource One that can be created (produced) and destroyed (consumed)
-ex Interrupts, signals, messages, and information. inside I/O buffers
Deadlock Detection Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

Conditions for deadlock
Mutual exclusion- only one process may use a resource at a time
Hold and wait- a process may hold allocated resources while awaiting assignment of others
No pre-emption- no resource can be forcibly removed from a process holding it
Circular wait- a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
Deadlock Prevention - adopt a policy that eliminates one of the conditions
Deadlock Avoidance - make the appropriate dynamic choices based on the current state of resource allocation
requires knowledge of future process requests
Deadlock Detection - attempt to detect the presence of deadlock and take action to recover
Deadlock prevention strategy Design a system in such a way that the possibility of deadlock is excluded
two methods: indirect and direct
Direct Deadlock Prevention - prevent the occurrence of a circular wait
Indirect Deadlock Prevention prevent the occurrence of one of the three necessary conditions
Conditions of Deadlock prevention
Mutual Exclusion- if access to a resource requires mutual exclusion then it must be supported by the OS
Hold and wait require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
No preemption- if a process holding certain resources is denied a further request, then process must release its original resources and request them again.
OS may preempt the second process and require it to release its resources
Circular Wait define a linear ordering of resource types
Approaches to Deadlock Avoidance - Resource Allocation Denial - do not grant an incremental resource request to a process if this allocation might lead to deadlock
Process Initiation Denial - do not start a process if its demands might lead to deadlock
Resource Allocation denial -
AKA banker's algorithm
-State of the system reflects the current allocation of resources to processes
Safe state is one in which there is at least one sequence of resources allocations to processes that does not result in a deadlock
unsafe state is a state that is not safe
Deadlock avoidance formula
if (C-A) <= available vector -> allocation + available,

resource vector must equal available vector at end if safe state
Deadlock Detection formula
set work to equal available vector
if Q <= Work -> allocation + work
mark each process after finishing it
Advantages of deadlock avoidance
-It is not necessary to preempt and rollback processes, as in deadlock detection
-It is less restrictive than deadlock preventtion
Restrictions of Deadlock Avoidance
-Maximum resource requirement for each process must be stated in advance
-Processes under consideration must be independent and with no synchronization
-There must be a fixed number of resources to allocate
-No process may exit while holding resources
Differences between Deadlock detection and prevention strategies
Deadlock prevention strategies are very conservative, they limit access to resources by imposing restrictions on processes
Deadlock detection strategies do the opposite, resources are granted whenever possible
Advantages of Deadlock detection
-it leads to early detection
-the algorithm is relatively simple
Disadvantages of Deadlock prevention
frequent checks consume considerable processor time
Recovery strategies
- abort all deadlocked processes
- back up each deadlocked process to some previously defined checkpoint and restart all processes
- successively abort deadlocked processes until deadlock no longer exists
- successively preempt resources until deadlock no longer exists

Dining Philosophers Problem The idea that resources must be shared among several processes in a deadlock and starvation free manner
(non mutual exclusion)