*Concurrency Mutual Exclusion and Synchronization
***OS design is concerned with the mangement of processes and threads**: -Multiprogramming, -Multiprocessing, -Distributed Processing
***Concurrency Arises in Three Different Contexts**:
1) Multiple Applications: invested to allow processing time to be shared among active applications
2) Structured Applications: extension of modular design and structured programming
3) Operating System Structure: OS themselves implemented as a set of processes or threads

Concurrency Key Terms:
***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 instructions 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.(log files useful if interrupted).
***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.
NOTE ** With a mutex semaphore (semaphore initialized to 1), everything between semwait() and semSignal() is critical.
Also, When inside of a monitor the critical section will be member functions. Also, Everything between a pthread_mutex_lock(&bsem) and pthread_mutex_unlock(&bsem) will be critical. Also, between sem_wait and sem_post
***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) w/o 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** - A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution. 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** - 1) Interleaving and overlapping *can be viewed as examples of concurrent processing, *both present the same problem
2) Uniprocessor - the relative speed of execution of processes cannot be predicted - *depends on activities of other processes
*the way the OS handles interrupts, *Scheduling policies of the OS
***Difficulties of Concurrency** - 1) sharing of global resources is fraught with peril, 2) Difficult for the OS to manage the allocation of resources optimally
3) 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:
1) be able to keep track of various processes.
2) allocate and de-allocate resources for each active process
3) protet the data and physical resources of each process against interference by other processes.
4) ensure that the processes and outputs are independent of the processing speed.

*** Process Interactions:**
- 1. Degree of Awareness: Processes unaware of each ohter
  Relationship: Competition
  Inclуence that One Process has on the other: - Results of one process independent of the action of others - Timing of process may be affected
  Potential Control Problems: - Mutual exclusion - Deadlock (renewable resource) - Starvation
- 2. Degree of Awareness: Processes Indirectly aware of each other (e.g. shared object)
  Relationship: Cooperation by Sharing
  Incluence that One Process has on the other: - Results of one process may depend on information obtained from others - Timing of process may be affected
  Potential Control Problems: - Mutual exclusion - Deadlock - Starvation - Data Coherence
- 3. Degree of Awareness: Processes directly aware of each other (have communication primitives available to them)
  Relationship: Cooperation by Communication
  Incluence that One Process has on the other: - Results of one process may depend on information obtained from others - Timing of process may be affected
  Potential Control Problems: - Deadlock - Starvation

***Resource Competition** - Concurrent processes come into conflict when they are competing for use of the same resource *for example: I/O devices, memory, processor time, clock
- In the case of competing processes three control problems must be faced: 1) the need for mutual exclusion 2) deadlock 3) starvation
***Requirements for mutual exclusion** - 1) Must be enforced 2) A process that halts must do so without interfering with other processes 3) No deadlock or starvation
3) A process must not be denied access to a critical section when there is no other process using it. 4) No assumptions are made about relative process speeds or number of processes. 5) A process remains inside its critical section for a finite time only.

***Peterson's Algorithm** - Algorithm to obtain mutual exclusion, has an issue related to using busy wait.

FIRST ATTEMPT: Assumption: Only one access to a memory location can be made at a time. Global memory location "turn" is reserved for shared variable

```
Process 0                           Process 1
{                                  {
    while (turn != 0)                   while (turn != 1)
    /*do nothing*/;                     /*do nothing*/;
    /*CS*/                              /*CS*/
    turn = 1;                           turn = 0;
}                                  }
```

Guarantees Mutual Exclusion. Has two problems: Processes must strictly alternate in their use of their CS; pace is dictated by the slower process. If one process fails, the other one is permanently blocked; whether in CS or not.

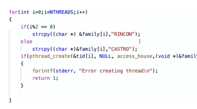
SECOND ATTEMPT: Need state information about both processes. flag[0] for P0 and flag[1] for P1(Boolean vector flag; when one fails, the other can still access CS)
Each process may examine the other's flag, but may not alter it...

```
enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};
Process 0                           Process 1
{                                  {
    while (flag[1])                     while (flag[0])
    /*do nothing*/;                     /*do nothing*/;
    flag[0]=TRUE;                       flag[1]=TRUE;
    /*CS*/                              /*CS*/
    flag[0]=TRUE;                       flag[1]=TRUE;
}                                  }
```

Does not Guarantees Mutual Exclusion. A process can change its state after the other process has checked it, but before the other process can enter into critical section.

THIRD ATTEMPT: Need state information about both processes. flag[0] for P0 and flag[1] for P1□(Boolean vector flag; when one fails, the other can still access CS)
Each process may examine the other's flag, but may not alter it...

```
enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};
Process 0                           Process 1
{                                  {
    flag[0]=TRUE;                       flag[1]=TRUE;
    while (flag[1])                     while (flag[0])
    /*do nothing*/;                     /*do nothing*/;
    /*CS*/                              /*CS*/
    flag[0]=TRUE;                       flag[1]=TRUE;
}                                  }
```

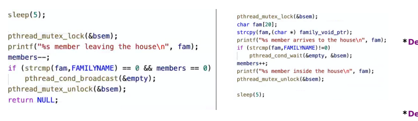


```
for(int i=0;i<NTHREADS;i++)
{
    if(i%2 == 0)
        strcpy((char *) &family[i],"RINCON");
    else
        strcpy((char *)&family[i],"CASTRO");
    if(pthread_create(&tid[i], NULL, access_house,(void *)&family[i]))
    {
        fprintf(stderr, "Error creating thread\n");
        return 1;
  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

}
```

If both processes set their flags to TRUE at the same time, then they are in a loop for ever. A process sets its flag without knowing other process's status!!

FOURTH ATTEMPT: Need state information about both processes. flag[0] for P0 and flag[1] for P1□(Boolean vector flag; when one fails, the other can still access CS)
Each process may examine the other's flag, but may not alter it...

```
enum boolean {FALSE=0; TRUE=1};
boolean flag[2] = {FALSE, FALSE};

Process 0                           Process 1
{                                  {
    flag[0]=TRUE;                       flag[1]=TRUE;
    while (flag[1]) {                   while (flag[0]) {
        flag[0] = FALSE                     flag[1] = FALSE
        /*delay*/                           /*delay*/
        flag[0] = TRUE                      flag[1] = TRUE
    }                                   }
/*CS*/                             /*CS*/
flag[0]=FALSE;                     flag[1]=FALSE;
}                                  }
```



```
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;
```

```
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);
```

The above sequences could be extended indefinitely. Neither process could get into CS It is not a deadlock! It is a livelock! Any alteration in relative speeds of processes could make one process enter into CS.

CORRECT SOLUTION: Need to observe the state of both processes, which process has the right to insist on entering into CS.

```
boolean flag[2];
int turn;

Process 0                           Process 1
{                                  {
    flag[0]=TRUE;                       flag[1]=TRUE;
    turn = 1;                           turn = 0;
    while (flag[1] && turn==1) {        while (flag[0] && turn==0) {
        /*do nothing*/                      /*do nothing*/
    }                                   }
/*CS*/                             /*CS*/
flag[0]=FALSE;                     flag[1]=FALSE;
}                                  }
```

Mutual Exclusion: Hardware Support:
> Interrupt Disabling: - Tniprocessor system - Tisabling interrupts guarantees mutual exclusion
> Disadvantages: - The efficiency of execution could be noticeably degraded - This approach will not work in a multiprocessor architecture (we cannot garantee mutual exclusion, all processes are disabled, they may use resources)
> Special Machine Instructions: Compare&Swap Instruction also called a "compare and exchange instruction" a compare is made between a memory value and a test value if the values are the same a swap occurs carried out atomically.

*** Special Machine Instruction: Advantages:** - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Disadvantages: -Busy-waiting is employed, thus while a process is waiting for access to a critical section it 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** - An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic (cannot be interrupted):
1) **Initialize** - may be initialized to a nonnegative integer value.
2) **Decrement** (semWait)- operation may result in the blocking of a process. Decrements the value. When a semWait is performed on a semaphore with value of 0, process will be suspended.
3) **Increment** (semSignal) - operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore. Increments the value.
*A variable that has an integer upon which only three operations are defined
*There is no way to inspect or manipulate semaphores other than these three operations listed.
***Consequences** - There is no way to know before a process decrements a semaphore whether it will block or not.
- There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently.
- You don't know whether another process is waiting so the number of unblocked processes may be zero or one.

Semaphore Primitives

```
struct semaphore {
    int count;
    queueType queue;
}
void semWait(sempahore s){
    s.count--;
    if(s.count < 0){
        /* place this process in s.queue */
        /* block this process */
    }
}
void semSignal(semaphore s){
    s.count++;
    if(s.count <= 0){
        /* remove a process P from s.queue */
        /* place process P on ready list */
    }
}
```

Binary Semaphore Primitives

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s){
    if(s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */
        /* block this process */
    }
}
void semSignamB(semaphore s){
    if(s.queue is empty())
        s.value = one;
    else {
        /* remore a process P from s.queue */
        /* place process P on ready list */
    }
}
```

Mutual Exclusion

```
/* program mutual exclusion */
const int n = /* number of processes */
semaphore s = 1;
void P (int i) {
    while (true) {
        semWait(s);
        /* critical section */
        semSignal(s);
    }
}

void main() {
    parbegin(P(1), P(2),...,P(n));
}
```

Producer Consumer Correct Solution

```
/* program producer consumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer() {
    while(true){
        produce();
        semWaitB(s);
        append();
        n++;
        if(n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer() {
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWait(s);
        take();
        n--;
        m=n;
        semSignalB(s);
        consume();
        if (m == 0) semWaitB(delay);
    }
}
void main(){
    n=0;
    parbegin (producer, consumer);
}
```

Cleaner Solution

```
/* program producer consumer */
semaphore n = 0, s = 1;
void producer() {
    while(true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(s);
    }
}
void consumer() {
    while(true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main() {
    parbegin(producer, consumer);
}
```

Finite Circular Buffer

```
/* program bounded buffer */
const int sizeofbuffer = /* buffer size */
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer() {
    while(true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer() {
    while(true) { semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main() {
    parbegin(producer, consumer);
}
```

Pipes
*Create pipe: system call pipe(), #include <unistd.h>, int pipe(int fd[2])
    [0] standard input
    [1] standard output
    [2] standard error
    ----
    [3] pipe read
    [4] pipe write
*unidirectional communication buffer with two file descriptors:
    fd[0] = read
    fd[1] = write
*Data write and read on a first-in-first-out base. No external or permanent name, and can only be accessed through two file descriptors
*The pipe can only be used by the process that created it and its descendants (i.e., child & grandchild processes)
*Pipe Commands:
    -close(fd): closes a file descriptor
    -dup(newfd) and dup2(newfd, oldfd): duplicates a file descriptor. Creates a copy of a given file descriptor. The new descriptor does not behave like a copy, but like an alias of the old one.
    -Read: not necessarily atomic: may read less bytes. Blocking: if no data, but write dile descriptor still opens. If empty, and all file scriptors for the write end are clsoed, read sees end-of-file, returns 0.
    -Write: Atomic for at most PIPE_BUF (512,4k,or 64k). Blocking: if buffer is full, and read file descriptors open. When all descriptors referring to the read end of a pipe are closed cause a SIGPIPE signal for the calling process.
    *Pros: simple, flexible, efficient communication
    *Cons: No way to open an already existing pipe. This makes it impossible for two arbitrary proceses to share the same pipe, unless the pipe was created by a common ancestor processes.
        - The only way we can use pipes, is when we have the relationship between parent and child process, and that is a restriction.

        - dup(fd[1])   Puts fd[1] in the first available position in the array of descriptors, if you close the standard output like: close(1) then dup(fd[1]) will replace the standard output, so you can use printf()
        - dup2(fd[1], 1)   close 1 and duplicate it with fd[1], replace standard output with fd[1]

Interprocess Communication - Shared Memory

*The parent and child processes are run in seperate address spaces
*A shared memory segment is a piece of memory that can be allocated and attached to an address space. Thus, processes that have this memory segment attached will have access to it.
    -race conditions can occur
*Commands:
    shmget() - to allocate a shared memory
    shmat() - to attach a shared memory to an address space
    shmdt() - to detach a shared memory from an address space
    shmctl() - to deallocate a shared memory.

> Shared Memory Segments:

Procedure for using shared memory along with commands:
Find a key. Unix uses this key for identifying shared memory segments.

How do we allow different processes to know where this shared memory segments is? By using Keys
Keys are fixed values.

All of this provides a way to share information between parent and child without using pipes

```
        shm_id = shmget( key_t key /* identity key */, int size /* memory size, use sizeof(int) */, int flag /* creation, use IPC_CREAT | 0666 */ );    // it creates a new shared memory segment but if the key
already exists it will get that segment to use it in another process

        shm_ptr = shmat( int shm_id /* ID from shmget() */, char *ptr /* Use NULL here*/, int flag /* Use 0 here */ );// returns pointer to the memory, needs to be casted to a datatype
        shmdt( shm_ptr );                                    // After a shared memory is detached, it is still there. You can re-attach and use it again
        shmctl( shm_is, IPC_RMID, NULL );                      // After a shared memory is removed, it no longer exists
```

*Keys - keys are global entities. If other processes know your key, they can access your shared memory.
    -ftok(): command to generate a key for you. key_t = ftok(char *path, int ID);

****LAST LECTURE****
Deadlock and Starvation
~Resource Categories
***Reusable resource**
  -Can be safely used by only one process at a time and it is not depleted by that use
    -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)
  -interrupts, signals, messages, and information
  -in i/o buffers
  NOTE: normally this type of resource has a time stamp.
***Ostrich Algorithm** - A strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named for the ostrich effect which is defined as "to stick one's head in
***Memory Request & deadlock example**
  -Space is available for allocation of 200k bytes, and the following sequence of events occur:

  -Deadlock occurs if both processes progress to their second request.
    *in this example, the ostrich algroithm will allow for the user to press 'cntrl + c' to forcefully stop the execution allowing for the other program to run.
***Consumable Resources Deadlock example**
  -Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process.

```
P1              P2
Recieve(P2)     Recieve(P1)
Send(P2,M1)         Send(P1,M2)
```

  -Deadlock occurs if the Receive is blocking.

***Deadlock Detection, Prevention, and Avoidance:**

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and p revention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

***Conditions for Deadlock**
1) Mutual exclusion - Only one process may use a resource at a time.
2) Hold-and-Wait - A process may hold allocated resources while awaiting assignment of others.
3) No Pre-emption - No resource can be forcibly removed from a process holding it
4) 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 (consequence of previous 3 conditions)
***Dealing with Deadlock**
-Three general approaches exist for dealing with dealock
  1) Prevent Deadlock - adopt a policy that eliminates one of the conditions
    *design a system in such a way that the possibility of deadlock is excluded.
    *two main methods: 1) Indirect - prevent the occurence of one of the three necessary conditions. 2) direct - prevent the occurence of a circular wait.
  2) Avoid Deadlock - make the appropriate dynamic choices based on the current state of resource allocation
  3) Detect Deadlock - attempt to detect the presence of deadlock and take action to recover
***Deadlock Condition Prevention**
1) Mutual Exclusion - if access to resource requires mutual exclusion then it must be supported by the OS
2) 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 granded simultaneously
  *may lead to starvation
3) No Preemption - if a process holding certain resources is denied a further request, that process must release its original resources and request them again.
  *OS may preempt the second process and require it to release its resources (will abort and may cause starvation)
4) Circular Wait - define a linear ordering of resource types.
***Deadlock Avoidance** - A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. requires knowledge of future process
  Two Approaches: 1) Resource allocation denial - do not grant an incremental resource request to a process if this allocation might lead to deadlock
  2) Process Initiation denial - do not start a process if its demands might lead to deadlock.
***Resource Allocation Denial** - referred to as the 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 resource allocations to processes that does not result in a deadlock
  -Unsafe state is a state that is not safe
  *Banker's Algorithm is the algorithm ran to test all permutations to determine if there is a safe state or not.
  ***QUESTION** : "which process can we run based off of our available vector?"
  Instuctions to complete this execution:
    Step 1) V = [0, 1 ,1] the only process that can run is (C-A) P2 [0, 0 , 1]   * Rule: C-A <= vector V
    Step 2) V[0, 1, 1] - (C-A)[0, 0 ,1] = V[0, 1, 0]
    Step 3) V[0, 1, 0] changes matrix A to [6, 1, 3].   * matrix C - new vector V = [6, 1, 3]
    Step 4) when row in matrix A == matrix C, all resources are there to finish execution. Release resources.
    Step 5) V = [0, 1, 0] + [6, 1 , 3] = [6, 2, 3]
    *after steps 1-5**
    Step 6) V[6,2,3] ***repeat steps for all other processes until finished to show that there is no deadlock***
  A safe state is when your vector V == vector R

***Binary semaphore** - A semaphore that takes on only the values 0 and 1.
***Mutex** - Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (value to 1). Initialized to 1. NOTE* binary_semaphore s= 1, is a mutex semaphore because it is initialized to 1.
***Condition Variable** - a data type that is used to block a process or thread until a particular condition is true.
***Monitor** - A programming language construct that encapsulates variables (like a class), access procedures and initialization code within an abstract data type. The monitors variable may only be accessed via its access procedures and only one process may be actively acessing the monitor at any one time. The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it.
***Event flags** - A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocking until all of the required bits are set (AND) or until at least one of the bits is set (OR)
***Mailboxes/Messages** - A means for two processes to exchange information and that may be used for synchronization
***Spinlocks** - Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.
***Strong semaphores** - The process that has been blocked the longest is released from the queue first (FIFO)
***Weak Semaphores** - The order in which processes are removed from the queue is not specified
***Producer/Consumer Problem**
  General Solution:
    *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 can't add data into full buffer and consumer can't remove data from an empty buffer.
***Monitors** - Programming language construct that provides equivalent functionality to that of semaphores and is easier to control. Guarantees mutual exclusion.
  - Implemented in a number of programming languages (Pascal, Pascal-plus, Modula-2/3, java)
  - Has been implemented as a program library
  - Software module consisting of one or more procedures, an initialization sequence, and local data.
  *system guarentees that when a process is accessing the monitor and its method(s), no other process can access, guarenteeing mutual exclusion.
***Monitor Characteristics**:
  *Local data variables are accessible only by the monitor's procedures and not by any external procesdure.
  data members private, member functions are public and used to changed values.
  *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 only within the monitor.
  -Condition variables are operated on by two functions:
    *cwait(c): suspend execution of the calling process on condition c.
    *csignal(c): resume execution of some process blocked after a cwait on the same condiiton
  NOTE* the difference between the semephore and a condition variable is that if a signal is sent to all processes waiting for the buffer to not be full, assume there is no processes waiting for that particular condition. This means that signmal will be lost. With a condition variable it will do a wait over that condition, this will allow you to get out of the monitor (monitor waiting area) and allow other processes to enter the monitor and do a procedure over that conditon.
***Deadlock Detection Algorithms** - A check for deadlock can be made as frequently as each resource requests or, less frequently, depending on how likely it is for a deadlock to occur
  *Advantages: -leads to early detection, -the algorithm is relatively simple
  *Disadvantage: -frequent checks consume considerable processor time

Deadlock Algorithm
step 1) Go to Allocation matrix A and find all zeros (not deadlocked). **mark p4 off**
step 2) Find a row in matrix Q that is <= to Allocation Vector. *P3 in above example **mark p3 off allocation matrix A**
step 3) V = [0,0,0,0,1] + A[0,0,0,1,0] = v[0,0,0,1,1]
step 4) find a row in matrix Q that is <= v[0,0,0,1,1] (NONE WILL WORK, DEADLOCK FOR ONES NOT MARKED & STOP EXECUTION)

***Recover Strategies:** -Abort all deadlocked process, -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.
*Deadlock Detection Algorithms - A check for deadlock can be made as frequently as each resource requests or, less frequently, depending on how likely it is for a deadlock to occur
  *Advantages: -leads to early detection, -the algorithm is relatively simple
  *Disadvantage: -frequent checks consume considerable processor time

Deadlock Algorithm
step 1) Go to Allocation matrix A and find all zeros (not deadlocked). **mark p4 off**
step 2) Find a row in matrix Q that is <= to Allocation Vector. *P3 in above example **mark p3 off allocation matrix A**
step 3) V = [0,0,0,0,1] + A[0,0,0,1,0] = v[0,0,0,1,1]
step 4) find a row in matrix Q that is <= v[0,0,0,1,1] (NONE WILL WORK, DEADLOCK FOR ONES NOT MARKED & STOP EXECUTION)

***Recover Strategies:** -Abort all deadlocked process, -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.

*Interprocess Commuication (sender/reciever can be on different comps. Relatively secure.)
Types of IPC: -Message passing -Shared memory
*Message passings - Processes that want to exchange data and recieve messages
-Any message exchange requires: 1) send(addr, msg, length); 2) recieve(addr, msg, length);
*Advantages: - Very general: Sender and receivers can be on different machines - Relatively secure: Receiver can inspect the messages it has received before processing them.
*Disadvantages - Hard to use: Every data transfer requires a send() and a recieve(). Recieving process must expect the send(), might require forking a special thread.
*Shared memory - two or more processes share a part of their address space.
*Avantages: - Fast and easy to use: The data are there but – Some concurrent accesses to the shared data can result into small disasters – Must synchronize access to shared data
*Disadvantages: • Not a general solution: – Sender and receivers must be on the same machine • Less secure: – Processes can directly access a part of the address space of other processes
*Message Passing Issues: - Direct/Indirect communication – Blocking/Non-blocking primitives – Exception handling – Quality of service: • Unreliable/reliable datagrams • Virtual circuits, streams
*Direct Communication - Send and receive system calls always specify processes as destination or source: (Most basic solution because there is no intermediary between sender and receiver)
-send(process, msg, length)
-receive(process, msg, &length)
*process executing the receive call must know the identity of all processes likely to send messages. (bad solution for servers) - servers have to answer requests from arbitrary processes.
*Indirect Communication - send receive primitives now specify an intermediary entity as destination or source: the mailbox.
-send(mailbox, msg, size);
-recieve(mailbox, msg, &size);
*__mailbox__ is a system object created by the kernel at the request of a user process. Can be 1) **private**: attached to a specific process 2) **Public**: system objects.
Private mailboxes: Process that requested its creation and its children are the only processes that can receive messages through the mailbox are the process and its children
: Cease to exist when the process taht requested its creation (and all its children) terminate. Often called ports. Example: BSD sockets.
Public mailboxes: Owned by the system, shared by all the processes having the right to receive messaged through it.
: Survive the termination of the process that requested their creation
: work best when all processes are on the same machine. Example: system V UNIX message queues
*Different processes can send messages to the same mailbox
-A process can recieve messages from processes it does not know anything about.
-A process can wait for messages coming from different senders, will answer the first message it receives.
*Blocking primitives
- **blocking send** - does not return until the receiving process has received the message. -no buffering is needed, -Analogous to what is happening when you call somebody who does not have voice mail
- **blocking receive** - does not return until a message has been received. -like waiting by the phone for an important message or staying all day by your mailbox waiting for the mail carrier.
*Non-blocking primitives
-A **non-blocking send** returns as soon as the message has been accepted for delivery by the OS
-Assumes that the OS can store the message in a buffer. (like mailing a letter: once the letter is dropped in the mailbox, we are done. The mailbox will hold your letter until a postal employee takes it)
*Can simulate a blocking send with two non-blocking sends and a blocking recieve. 1) sender sends a message and requests an acknowledgement ACK. 2) Sender waits for ACK from receiver using a blocking receive
- A **non-blocking receive** returns as soon as it has either retrieved a message or learned that the mailbox is empty (like checking whether your mail has arrived or not)
*A non-blocking recieve can simulate a blocking recieve with a loop.
do{code = receive(mbox,msg,size); sleep(1); }while(code == EMPTY_MBOX); -This is known as a **busy wait**
***The standard Choice (the correct combination)**
-inderect naming
-non-blocking sends (- Sender does not care about what happens once the message is sent - Similar to UNIX delayed writes)
-blocking receives (Receiver needs the data to continue)
*Buffering : Non-blocking primitives require buffering to let OS store somewhere messages that have been sent but not yet received.
The buffers can have: *Bounded capacity: Refuse to receive messages when the buffer is full
*****Theroretically unlimited capactiy
Blocking receive does NOT go well with direct communication (an explosive combination) – Processes cannot wait for messages from several sources without using special parallel programming constructs: • Dijkstra's alternative command
*****using blocking receives with direct naming does not allow the receiving process to receive any messages from any process but the one it has specified.
*Exception condition handling : Must specify what to do if one of the two processes dies. Especially important whenever the two processes are on two different machines. must handle: 1) host failure 2) network partitions
*Quality of Service : • When sender and receiver are on different machines, messages– Can be lost, corrupted or duplicated – Arrive out of sequence • Can still decide to provide reliable message delivery
*Positive Acknowledgments: Basic technique for providing delivery of messages
*Destination process sends an acknowledgment ACK for every message that was correctly delivered.
*Damaged messages are ignored, sender resends any message that has not been ACK within a fixed time frame.
***RULE:** Acknowledge any message that does not need to be resent!

***Classes of service:**
*Datagrams - messages are send one at a time • Each message is sent individually:
*some messages can be lost, other duplicated or arrive out of sequence. Equivalent of a conventional letter.
*Reliable datagrams: resent until they are acknowledged.
*Unreliable datagrams = **UDP**,
- Messages are not acknowledged. Works well when message requests a reply, reply is **Implicit ACK.** • User Datagram Protocol: • Best known datagram protocol • Provides an unreliable datagram service
– Messages can be lost, duplicated or arrive out of sequence • Best for short interactions – One request and one reply.
- Sole reason to ACK a request is when it might take a long time to reply to it.
*Virtual circuits - Ordered sequence of messages, connection-oriented service.
*Establish a logical connection between the sender and the receiver.
*Messages are guaranteed to arrive in sequence without lost messages or duplicated messages. (Analogous to the words of a phone conversation).
• Require setting up a virtual connection before sending any data
– Costlier than datagrams
• Best for transmitting large amounts of data that require sending several messages
– File transfer protocol (FTP)
– Hypertext transfer protocol (HTTP)
*Streams - Like virtual circuits - Ordered sequence of bytes, Message boundaries are ignored. (Not messages, we are sending a stream of bytes, the receiver needs to know they are receiving the same amount of bytes, and they represent a particular datatype)
• Do not preserve message boundaries:
– Receiver sees a seamless stream of bytes
• Offspring of UNIX philosophy
– Record boundaries do not count
– Message boundaries should not count

*TCP - Transmission Control Protocl, Best known stream protocol.
-Provides a reliable stream service
-Heavyweight *requires three messages (packets) to establish a virtual connection
*Datagram vs Streams:
Datagrams : *Unreliable *Not ordered *Lightweight *Deliver messages (UDP)
Streams : *Reliable *Ordered *Heavyweight *Stream-oriented (TCP)

*REMOTE PROCEDURE CALLS
-Apply to client-server model of computation
-A typical client-server interaction:

```
        send_req(args);              rcv_req(&args);
                                     process(args, &results);
                                     send_reply(results);
        rcv_reply(&results);
```

-System takes care of all message passing details.
*RPC Advantages
-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 model of programming
-Anybody that cna use procedures and function can quickly learn to use remote procedure calls.
*RPC Disadvantages
-The illusion is not perfect
-RPCs do not always behave exactly like regular procedure calls
-Client and server do not share the same address space.
-Programmer must remain aware of these subtle and not so subtle differences
*RPC explained from audio lecture
-The user program creates (calls) a user stub (system generated), the stub handles the communication process. The server creates a stub which gets passed to the server procedure (server procedure does all the calculations)
-what we see as a programmer is the user program and the server procedure (All IPC between the client and server are hidden).
*The User Program:
-contains the user code, -calls the user stub: rpc(xyz, args, &results), appears to call the server procedure.
*The 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 results and performs required data conversion (argument unmarshaling)
*The Server stub:
-Generic server generated by RPC package:
-Waits for client requests
-Unpacks request arguments and performs required data conversions
-Calls appropriate server procedure
-Packs results into reply message and performs required data conversions
-sends reply message
*The Server Procedure (basically executing the function): -Procedure called by the server stub, -written by the user, -Does the actual processing of user requests
*Differences with regular Procedural call
- 1) Client and server processes do not share the same address space. 2) Client and server can be on different machines. 3) Must handle partial failures.
*No Shared address space : 1) No global variables, 2) Cannot pass addresses, which means you Cannot pass arguments by reference, You 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:
-Cannot pass dynamic data structures through pointers
-Must send a copy of data structure
-For a linked list: -send array with elements of linked list plus unpacking instructions. You re-create the linked list on the server side. The Header identifies linked list of 4 elements [LL][4][A][B][C][D]
*Architecture consdierations
-The machine representation of floating point numbers and byte ordering conventions can be different:
-Little endians start at the least significant byte: Intel's 80x86 pentium
-Big-endians start at the most significant byte: Sun's SPARC and most RISC processors
The solution: • Define a network order and convert all numerical variables to that order – Use hton family of functions – Same as requiring all air traffic control communications to be in English
*Detecting partial failures: The client must detect server failures.
*Handling partial executions: -Client must deal with the possibility that the server could have crashed after having partially executed the request
-Solution(s): 1) Ignore the problem and always resubmit requests that have not been answered (some requests may be executed more than once)
1) Will work if all requests are idempotent - Executing them several times has the same effect as executing them exactly once
-example of idempotent requests: 1) reading n bytes from a fixed location. 2) Writing n bytes starting at a fixed location 2) checking an account balance over and over
2) Attach a serial number to the request: servers 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 being partially executed
3) Use a transaction mechanism: -Guarantees that each requests will either be fully executed or have no effect (All or nothing semantics, best and costliest solution)

Chapter 05
Message Passing:
When processes interact with one another two fundamental requirements must be satisfied: - synchronization (to enforce mutual exclusion) - communication (to exchange information)
Message Passing is one approach to providing both of these functions (works with distributed systems and shared memory multiprocessor and uniprocessor systems)

The actual function is normally provided in the form of a pair of primitives: send (destination, message) / receive (source, message) -> A process sends information in the form of a message to another process designated by a destination
A process receives information by executing the receive primitive, indicating the source and the message

Synchronization: > Communication of a message between two processes implies synchronization between the two: the receiver cannot receive a message until it has been sent by another process
>When a receive primitive is executed in a process there are two possibilities: - if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive
if a message has previously been sent the message is received and execution continues

Blocking Send, Blocking Receive: - Both sender and receiver are blocked until the message is delivered - Sometimes referred to as a rendezvous - Allows for tight synchronization between processes
Nonblocking Send:
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
Addressing: Schemes for specifying processes in send and receive primitives fall into two categories: - Direct addressing - Indirect addressing

Direct Addressing: - Send primitive includes a specific identifier of the destination process - Receive primitive can be handled in one of two ways: 1. require that the process explicitly designate a sending process *effective for cooperating concurrent processes 2. implicit addressing *source parameter of the receive primitive possesses a value returned when the receive operation has been performed
Indirect Addressing: Messages are sent to a shared data structure consisting of queues that can temporarily hold messages -> Queues are referred to as mailboxes -> One process sends a message to the mailbox and the other process picks up the message from the mailbox -> Allows for greater flexibility in the use of messages
Readers/Writers 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: 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 Avoidance Advantages: - It is not necessary to preempt and rollback processes, as in deadlock detection - It is less restrictive than deadlock prevention.**
* **Deadlock Avoidance Restrictions: - Maximum resource requirement for each process 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
* **Deadlock Strategies: - Deadlock prevention strategies are very conservative (limit access to resources by imposing restrictions on processes) -Deadlock detection strategies do the opposite (resource requests are granted whenever possible)**

Deadlock detection

$$Q = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \rightarrow \text{Request matrix } Q = C - A$$

C → claim matrix

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow \text{Allocation matrix}$$

$V = [1, 1, 0] \rightarrow$ available vector

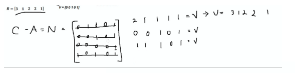
Steps to get R:
1.) To get Resource R. Add all columns of matrix A.
$[2 \quad 3 \quad 2]$
2.) Add that to the available vector V
(sum of columns A) + V = R
$[3 \quad 4 \quad 2] = R \rightarrow$ Resource Vector.

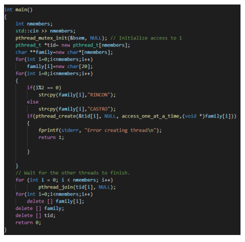
Steps to find deadlock (Banker's algorithm)

1) Need matrix = C − A
Q = C − A

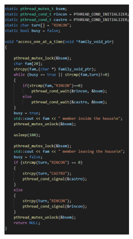
Rule: 1.) claim matrix ≤ R vector
2.) matrix Q ≤ V vector

Rows ≤ are safe
Rows > are not safe and may lead to a deadlock

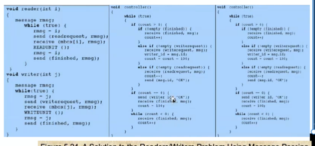




```
int main()
{
    int nmembers;
    std::cin >> nmembers;
    pthread_mutex_init(&bsem, NULL); // Initialize access to 1
    pthread_t *tid= new pthread_t[nmembers];
    char **family=new char*[nmembers];
    for(int i=0;i<nmembers;i++)
        family[i]=new char[20];
    for(int i=0;i<nmembers;i++)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



```
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;
}
```

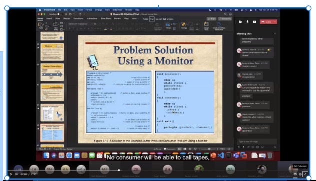


```
void reader(int i)
{
    message rmsg;
        while (true) {
            rmsg = i;
            send (readrequest, rmsg);
            receive (mbox[i], rmsg);
            READUNIT ();
            rmsg = i;
            send (finished, rmsg);
        }
}
void writer(int j)
{
    message rmsg;
    while(true) {
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void    controller()
{
        while (true)
        {
            if (count > 0) {
                if (!empty (finished)) {
                    receive (finished, msg);
                    count++;
                }
                else if (!empty (writerequest)) {
                    receive (writerequest, msg);
                    writer_id = msg.id;
                    count = count - 100;
                }
                else if (!empty (readrequest)) {
                    receive (readrequest, msg);
                    count--;
                    send (msg.id, "OK");
                }
            }
            if (count == 0) {
                send (writer_id, "OK");
                receive (finished, msg);
                count = 100;
            }
            while (count < 0) {
                receive (finished, msg);
                count++;
            }
        }
}
```

```
void    controller()
{
        while (true)
        {
            if (count > 0) {
                if (!empty (finished)) {
                    receive (finished, msg);
                    count++;
                }
                else if (!empty (writerequest)) {
                    receive (writerequest, msg);
                    writer_id = msg.id;
                    count = count - 100;
                }
                else if (!empty (readrequest)) {
                    receive (readrequest, msg);
                    count--;
                    send (msg.id, "OK");
                }
            }
            if (count == 0) {
                send (writer_id, "OK");
                receive (finished, msg);
                count = 100;
            }
            while (count < 0) {
                receive (finished, msg);
                count++;
            }
        }
}
```

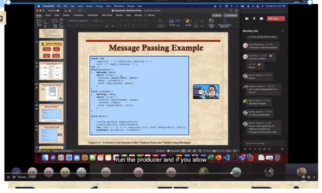
Figure 5.24 A Solution to the Readers/Writers Problem Using Message Passing



Problem Solution Using a Monitor

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

No consumer will be able to call tapes,



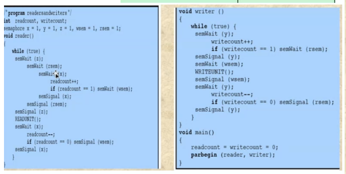
Message Passing Example

Figure 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

run the producer and if you allow

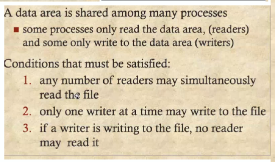
Readers only in the system	•*wsem* set •no queues
Writers only in the system	•*wsem* and *rsem* set •writers queue on *wsem*
Both readers and writers with read first	•*wsem* set by reader •*rsem* set by writer •all writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*
Both readers and writers with write first	•*wsem* set by writer •*rsem* set by writer •writers queue on *wsem* •one reader queues on *rsem* •other readers queue on *z*

•other readers queue on *z*



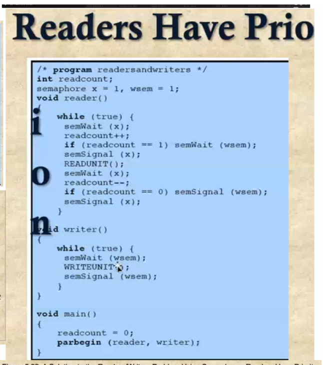
```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
   while (true) {
    semWait (z);
        semWait (rsem);
            semWait (x);
                readcount++;
                if (readcount == 1) semWait (wsem);
            semSignal (x);
        semSignal (rsem);
    semSignal (z);
    READUNIT();
    semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
    semSignal (x);
   }
}
```

```
void writer ()
{
    while (true) {
      semWait (y);
          writecount++;
          if (writecount == 1) semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
          writecount--;
          if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```



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)

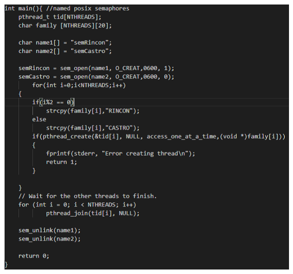
Conditions that must be satisfied:
1. any number of readers may simultaneously read the file
2. only one writer at a time may write to the file
3. if a writer is writing to the file, no reader may read it



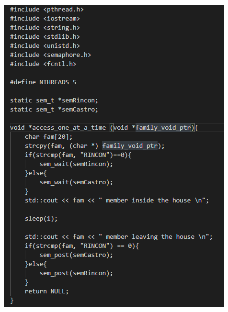
Readers Have Prio

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
     semWait (x);
     readcount++;
     if (readcount == 1) semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0) semSignal (wsem);
     semSignal (x);
    }
}
void writer()
{
    while (true) {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

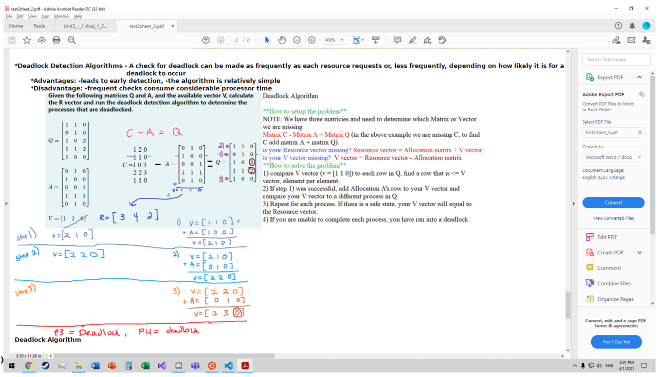
Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority



```
int main(){ //named posix semaphores
    pthread_t tid[NTHREADS];
    char family [NTHREADS][20];

    char name1[] = "semRincon";
    char name2[] = "semCastro";

    semRincon = sem_open(name1, O_CREAT,0600, 1);
    semCastro = sem_open(name2, O_CREAT,0600, 0);
        for(int i=0;i<NTHREADS;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 < NTHREADS; i++)
            pthread_join(tid[i], NULL);

    sem_unlink(name1);
    sem_unlink(name2);

    return 0;
}
```



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

#define NTHREADS 5

static sem_t *semRincon;
static sem_t *semCastro;

void *access_one_at_a_time (void *family_void_ptr){
    char fam[20];
    strcpy(fam, (char *) family_void_ptr);
    if(strcmp(fam, "RINCON")==0){
        sem_wait(semRincon);
    }else{
        sem_wait(semCastro);
    }
    std::cout << fam << " member inside the house \n";

    sleep(1);

    std::cout << fam << " member leaving the house \n";
    if(strcmp(fam, "RINCON") == 0){
        sem_post(semCastro);
    }else{
        sem_post(semRincon);
    }
    return NULL;
}
```



test2sheet_2.pdf - Adobe Acrobat Reader DC (32-bit)

***Deadlock Detection Algorithms - A check for deadlock can be made as frequently as each resource requests or, less frequently, depending on how likely it is for a deadlock to occur**
***Advantages: -leads to early detection, -the algorithm is relatively simple**
***Disadvantage: -frequent checks consume considerable processor time**

Given the following matrices Q and A, and the available vector V, calculate the R vector and run the deadlock detection algorithm to determine the processes that are deadlocked.

$$Q = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$V = [1 \ 1 \ \cancel{0}]$ $R = [3 \ 4 \ 2]$

C − A = Q

$$C = \begin{matrix} 1\,2\,0 \\ 1\,1\,0 \\ 1\,0\,3 \\ 2\,2\,3 \\ 1\,1\,0 \end{matrix} - A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} = Q = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & \boxed{2} \\ 1 & 1 & \boxed{2} \\ 1 & 0 & 0 \end{bmatrix}$$

step 1) v=[2 1 0] 1) V=[1 1 0] + A=[1 0 0] = V=[2 1 0]
step 2) v=[2 2 0] 2) V=[2 1 0] + A=[0 1 0] = V=[2 2 0]
step 3) 3) V=[2 2 0] + A=[0 1 0] = V=[2 3 $\boxed{0}$]

P3 = Deadlock, P4 = deadlock

Deadlock Algorithm

Deadlock Algorithm

How to setup the problem
NOTE: We have three matricies and need to determine which Matrix or Vector we are missing
Matrix C - Matrix A = Matrix Q (in the above example we are missing C, to find C add matrix A + matrix Q).
is your Resource vector missing? Resource vector = Allocation matrix + V vector
is your V vector missing? V vector = Resource vector - Allocation matrix
How to solve the problem
1) compare V vector (v = [1 1 0]) to each row in Q, find a row that is <= V vector, element per element.
2) If step 1) was successful, add Allocation A's row to your V vector and compare your V vector to a different process in Q.
3) Repeat for each process. If there is a safe state, your V vector will equal to the Resource vector.
4) If you are unable to complete each process, you have ran into a deadlock.