When using condition variables, signal(c) suspends the execution of the calling process on condition c **(False).** In message passing, a solution based on mailboxes uses direct addressing **(False).** All mutex semaphores are binary semaphores **(True).** A semaphore is a reusable resource **(True).** A deadlock avoidance mechanism requires knowledge of future process requests **(True).** Concurrency is possible in Uniprocessor systems **(True).** Peterson's algorithm is a hardware-based solution to guarantee mutual exclusion **(False)**. A disadvantage of the deadlock detection algorithm is that frequent checks consume considerable processor time **(True).** A banker's algorithm is a deadlock prevention mechanism **(False).** A deadlock occurs when two processes request the same resources in the same order at the same time **(False).** When using condition variables, signal(c) suspends the execution of the calling process on condition c **(False).** The Banker's algorithm is a deadlock prevention mechanism **(False)**. In deadlock avoidance, the solution is executed after assigning the resources to a process.**(False).** 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).** When executing concurrent processes, the result of these processes are deterministic and reproducible. **(False).**

1)The OS needs to be concerned about cooperation by sharing when the processes are: **a. Indirectly aware of each other** b. Unaware of each other c. Directly aware of each other d. None of the above. 2) Select the matrix of the Banker's algorithm that is equal to the matrix Q of the deadlock detection algorithm: a. Matrix A b. Matrix Q **c. Matrix C - A** d. None of the above. 3) In the deadlock detection algorithm, if all processes are marked, then: a. All processes are deadlocked **b. No deadlock was detected** c. The algorithm has not started its execution d. None of the above. 4) Select the option that is a disadvantage of special machine instructions **a. Busy-waiting is employed** b. Will not work in multiprocessor systems c. Disabling the interrupts is a hard task for the programmer d. None of the above 5) A situation in which a runnable process is overlooked indefinitely by the scheduler is: a. Mutual Exclusion b. Deadlock c. Livelock d. Racing condition **e. None of the above** 6) Select the option that is not a condition for a deadlock a. Mutual Exclusion b. Hold-and-Wait c. No Pre-emption d. Circular Wait **e. None of the above** 7) Select the option that is not a requirement for mutual exclusion a. No deadlock or starvation **b. Using the relative process speeds or the number of processes as parameters to guarantee mutual exclusion.** c. A process remains inside its critical section for a finite time only. d. A process that halts must do so without interfering with other processes. e. None of the above 8) Select the option that is not a recovery strategy of the deadlock detection algorithm a. Abort all deadlocked processes b. successively abort deadlocked processes until deadlock no longer exists c. Successively preempt resources until deadlock no longer exists **d. None of the above** 9) In message passing select the combination that is referred to as rendezvous a. Non-blocking send and blocking receive b. Blocking send and Non-Blocking receive c. Non-blocking send and Non-Blocking receive **d. Blocking send and blocking receive** 10) Select the value that you must use to initialize a mutex semaphore a. 0 b. N **c. 1** d. None of the above 11) Select the option that is a deadlock prevention approach **a. Requesting all resources at once** b. Banker's algorithm c. Detection algorithm d. Ostrich Algorithm e. None of the above 12) Select the concurrency mechanism that is a hardware solution a. Semaphores **b. Exchange Instruction** c. Peterson's Algorithm d. None of the above 13) Select the method that must be part of a program based on monitors to solve the dining philosophers problem a. Eat() **b. Release_forks()** c. Think() d. None of the above

Deadlock Detection Algo
Q = C-A (its already given)
R = V+A
*** mark a row in A if it has all 0's**
Compare Q to V

Banker's Algo
R = V+A
Compare C-A to V
Update A matrix after a process is in safe state (* mark the row)
Add A to V
Maintain a safe state queue <p2,p0…>

3. **A restaurant has a single employee taking orders and has three seats for its customers. The employee can only serve one customer at a time and each seat can only accommodate one customer at a time. Complete the following function template in a way that guarantees that customers will never have to wait for a seat while holding the food they have just purchased.**

```
semaphore seats = 3;
semaphore employee = 1;
void customer () {
        semWait(&seats);
        semWait(&employee);
        order_food();
        semSignal(&employee);
        eat();
        semSignal(&seats);
} // customer
```

a) Set the initial values of the semaphores **(5 points).**
b) Select the missing instructions after **order_food()** and **eat()** from the following list **(15 points)**:

1. semSignal(&seats);
2. semWait(&employee);
3. semWait(&seats);
4. semSignal(&employee);

Complete the following C++ program to guarantee that only one person at a time will be in the house, alternating between a Rincon family member and a Castro family member (starting with a Rincon family member). Your program will receive from STDIN the number of people (npeople). The number of Rincon family members is ceil(npeople / 2) and the number of Castro family members is npeople - the number of Rincon family members.

For npeople = 5, the number of Rincon family members is 3 and the number of Castro family members is 2.

```
RINCON member inside the house
RINCON member leaving the house
CASTRO member inside the house
CASTRO member leaving the house
RINCON member inside the house
RINCON member leaving the house
CASTRO member inside the house
CASTRO member leaving the house
RINCON member inside the house
RINCON member leaving the house
```

The following code comes from a prev exam. I was told "test to see if this switch from (turn > myTurn to turn == myTurn) works b/c thats how the prof showed me"

```
#include <pthread.h>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
using namespace std;

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

void *print_in_reverse_order(void *void_ptr_argv)
{
    int myTurn = *(int *) void_ptr_argv;
    // As long as its not my turn... WAIT!
    pthread_mutex_lock(&bsem);
    while (turn > myTurn) {
        pthread_cond_wait(&waitTurn, &bsem);
    }
    pthread_mutex_unlock(&bsem);

    // If it's my turn, print and and decrement turn.
    pthread_mutex_lock(&bsem);
    cout << "I am Thread " << myTurn << endl;
    turn = turn - 1;
    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
     pthread_t *tid= new pthread_t[nthreads];
    int *threadNumber=new int[nthreads];
    turn = nthreads - 1;

    for(int i=0;i<nthreads;i++)
    {
        threadNumber[i] = i;
        if (pthread_create(&tid[i], NULL, print_in_reverse_order, &threadNumber[i])) {
            cerr << "Failed to create thread";
            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;
}
```

```
inting the threads in reverse order
ide <pthread.h>
ide <iostream>
ide <string>
 namespace std;

: pthread_mutex_t bsem;
: pthread_cond_t waitTurn = PTHREAD_COND_INITIALIZER;
: int turn;

'print_in_reverse_order(void *void_ptr_argv) {
 threadID = *((int *)void_ptr_argv);
'ead_mutex_lock(&bsem);
le (threadID!=turn) {
:hread_cond_wait(&waitTurn, &bsem);

'ead_mutex_unlock(&bsem);
: << "I am thread " << threadID << endl;
'ead_mutex_lock(&bsem);
ı = turn - 1;
'ead_cond_broadcast(&waitTurn);
'ead_mutex_unlock(&bsem);
ırn nullptr;


ıin() {
 nthreads = 5;
'ead_mutex_init(&bsem, nullptr);
'ead_t *tid = new pthread_t[nthreads];
 *threadNumber = new int[nthreads];
ı = nthreads - 1;
 (int i = 0; i < nthreads; i++) {
ıreadNumber[i] = i;
:hread_create(&tid[i], nullptr, print_in_reverse_order, &threadNumber[i]);


 (int i = 0; i < nthreads; ++i) {
:hread_join(tid[i], nullptr);

:te[] threadNumber;
:te[] tid;
ırn 0;

'//////  output ////////////
:hread 4
:hread 3
:hread 2
:hread 1
:hread 0
```

```
#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;
 }
```

Chapter 5 – KEY TERMS

Atomic Operation – A function or action implemented as a sequence of one or more instructions that appears to 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 process.
Critical Section – A section of code within a process that requires access to shared resources and 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 other to do something.
Livelock – A situation in which two or more processes continuously change their states in response to changes in the other process without doing any useful.
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 access any of these 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.
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 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

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.

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.

Mutual Exclusion: Hardware Support

Interrupt Disabling
uniprocessor system.
disabling interrupts guarantees mutual exclusion.
Disadvantages:
the efficiency of execution could be noticeably degraded.
this approach will not work in a multiprocessor architecture.
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.

Special Machine Instruction: 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

A variable that has an integer value upon which only three operations are defined:
May be initialized to a nonnegative integer value.
The **semWait** operation decrements the value.
The **semSignal** operation increments the value.
There is no way to inspect or manipulate semaphores other than these three operations

Semaphore 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.

Strong/Weak Semaphores

A queue is used to hold processes waiting on the semaphore.
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 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 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.

Implementation of Semaphores

Imperative 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 semaphores and is easier to control.
Implemented in a number of programming languages
including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java.
Has also been implemented as a program library.
Software module 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 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 condition

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.

Message Passing II

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 II

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

Chapter 5 – KEY TERMS CONTINUED…

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:
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

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:
Any number of readers may simultaneously read the file
Only one writer at a time may write to the file
If a writer is writing to the file, no reader may read it.

Summary

Messages: Useful for the enforcement of mutual exclusion discipline.
Operating system themes are: Multiprogramming, multiprocessing, distributed processing
Fundamental to these themes is concurrency
Issues of conflict resolution and cooperation arise.
Mutual Exclusion: Condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time
One approach involves the use of special purpose machine instructions.
Semaphores: Used for signaling among processes and can be readily used to enforce a mutual exclusion discipline.

Chapter 6 – KEY TERMS

Deadlock

The permanent blocking of a set of processes that either compete 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
Permanent
No efficient solution.

Resource Categories

Reusable: can be safely used by only one process at a time and 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: one that can be created (produced) and destroyed (consumed)
interrupts, signals, messages, and information
in I/O buffers

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.

Dealing with Deadlock

Three general approaches exist for dealing with deadlock
Prevent Deadlock: adopt a policy that eliminates one of the conditions.
Avoid Deadlock: make the appropriate dynamic choices based on the current state of resource allocation.
Detect Deadlock: 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 main methods:
Indirect:
prevent the occurrence of one of the three necessary conditions
Direct:
prevent the occurrence of a circular wait.
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, that 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.

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 request.
Two Approaches to Deadlock Avoidance:
Resource Allocation Denial: do not grant an incremental resource request to a process if this allocation might lead to deadlock. 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
Process Initiation Denial: do not start a process if its demands might lead to deadlock.

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

Deadline Detection Algorithms

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
Advantages:
it leads to early detection
the algorithm is relatively simple
Disadvantage:
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

No two philosophers can use the same fork at the same time (mutual exclusion)
No philosopher must starve to death (avoid deadlock and starvation)

PIPE

A circular buffer allowing two processes to communicate on the producer–consumer model. Thus, it is a first-in-first-out queue, written by one process and read by another. In some systems, the pipe is generalized to allow any item in the queue to be selected for consumption. Two types: Name, Unnamed.

Atomic Operations II

Atomic operations execute without interruption and without interference
Simplest of the approaches to kernel synchronization
Two types:
Integer Operations: operate on an integer variable, counters.
Bitmap Operations: operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable.

Chapter 6 – KEY TERMS CONTINUED…

Spinlocks

Most common technique for protecting a critical section in Linux
Can only be acquired by one thread at a time
any other thread will keep trying (spinning) until it can acquire the lock
Built on an integer location in memory that is checked by each thread before it enters its critical section
Effective in situations where the wait time for acquiring a lock is expected to be very short
Disadvantage:
locked-out threads continue to execute in a busy-waiting mode.

Synchronization Primitives

Mutual Exclusion (Mutex) Locks, Condition Variables, Semaphores, Readers w/ Rider Locks.

Critical Sections

Similar mechanism to mutex except that critical sections can be used only by the threads of a single process
If the system is a multiprocessor, the code will attempt to acquire a spin-lock
as a last resort, if the spinlock cannot be acquired, a dispatcher object is used to block the thread so that the kernel can dispatch another thread onto the processor.

Summary

Deadlock:
the blocking of a set of processes that either compete for system resources or communicate with each other
blockage is permanent unless OS takes action
may involve reusable or consumable resources
Consumable - destroyed when acquired by a process
Reusable - not depleted/destroyed by use
Dealing with deadlock:
Prevention – guarantees that deadlock will not occur
Detection – OS checks for deadlock and takes action
Avoidance – analyzes each new resource request

CRITICAL SECTION:

```
pthread_mutex_init(&bsem, NULL); //Init bsem to 1
static int members = 0;

void *access_house(void *family_void_ptr){

    pthread_mutex_lock(&bsem);
    /* CRITICAL SECTION I */
    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++; // SHARED RESOURCE
    printf("%s member inside the house\n", fam);
    /* CRITICAL SECTION I END */
    pthread_mutex_unlock(&bsem);

    sleep(5);

    pthread_mutex_lock(&bsem);
    /* CRITICAL SECTION II */
    printf("%s member leaving the house \n", fam);
    members--; // SHARED RESOURCE
    if(strcmp(fam, FAMILYNAME) == 0 && members == 0)
        pthread_cond_broadcast(&empty);
    /*CRITICAL SECTION II END */
    pthread_mutex_unlock(&bsem);

    return NULL;
}
```

NOTICE:
pthread_mutex_lock(&mutex);
/* critical section */
pthread_mutex_unlock(&mutex);

MORE INFO:

In the proposed solution of lock variables. Which of the following does it address: Scheduler independent & Allows Progress. **What are some of the problems with lock variables as a solution?** Either does not work at all or depends on a scheduler. Starvation is possible. Require busy waiting. Require busy waiting. **How does Dekker/Petersen's Solution function with regards to solving the critical section problem?** Utilizes a ready flag and a turn indicator variable. Whichever process reaches the critical section first waits for the other to go by setting the flag for the other. **In the Dekker/Petersen's solution, which of the following has it addressed?** Mutual exclusion**,** Scheduler independent, Allows progress, Starvation free. **What are some of the problems with Dekker/Petersen's solution?** Only works for 2 processes. Requires busy waiting. Assumption: writes and reads are atomic. **What does it mean for a read or write to be performed atomically?** An atomic operation, is one that is un-interruptible. **What atomic operation(s) are supported on many CPU's?** test and set. If the CPU is busy, it cannot execute its operation, otherwise it does. **In the Test and Set solution, which of the following has it addressed?** Mutual Exclusion, Scheduler Independent, Allows progress. **What is a lock?** Mutual exclusion mechanism to protect critical sections. **How does a lock address the concurrency problem?** Prevents more than one thread from executing a certain bit of code (eg: code to modify a variable). **What are the two fundamental operations for locks?** Lock and Unlock. **What is a SpinLock?** a lock mechanism that requires a thread to spin in a loop testing a condition of some sort (waiting for another thread to unlock). **What are some possible problems with spin locks?** Starvation is possible, Busy waiting. **What is a solution to the busy waiting problem?** Suspend / Resume Cycle. **How does Sleep/Wake or Suspend/Resume differ from a busy waiting scenario?** It differs in that the onus is put on the thread in the critical section to notify the waiting (and suspended) thread when it has completed its work and the critical section can now be entered. **What are some implementation issues with the Suspend/Resume Cycle?** If many threads are waiting / suspended, they must be ordered in some way. **What are 3 abstractions used for mutual exclusion?** Locks (mutex), Semaphores, Monitors. **What are the 4 instructions in Pthreads mutex API? pthread_mutex_init() pthread_mutex_destroy(), pthread_mutex_lock() pthread_mutex_unlock(). What information is contained in a semaphore?** # of pending wakeups, # of sleeping process. **What are the two atomic operations of a semaphore?** Acquire & Release. **What are the two types of semaphores?** binary: can be at most 1, counting: Initialized to a positive number, can be decremented until 0 where it locks. Lets a specified number of processes access a resource. **What are some possible issues with implementing semaphores?** The internal lock on the semaphore can still cause a spin lock when a thread is trying to acquire the semaphore. Deadlocks are still possible. **What are the advantages of a Semaphore?** Don't have to increment/decrement counters. Don't have to check for boundary cases. Single call required. Same abstraction for different synchronization problems. **How can we mitigate the issues associated with semaphores?** Implementing a safe queue to queue processes waiting to acquire a semaphore. Must ensure the semaphore queue is mutexed. **How do monitors differ from locks or semaphores?** Monitors are language specific. Monitors are implemented with locks, semaphores, or other mechanisms. **What will likely occur if two or more threads try to acquire semaphores incorrectly?** A deadlock. **What are the four conditions necessary to create a deadlock?** Mutual Exclusion**,** Hold and Wait. No preemption, Circular Wait. **What are the 4 methods of handling Deadlocks?** Prevention, Avoidance, Do nothing, Detection and Recovery. **What is a RAG and what is its purpose?** Resource Allocation Graph, Used to reason about deadlocks.

Quiz 2 Review

What is the main objective of the deadlock prevention techniques?
Adopt a policy at the design level to eliminate one of the conditions for deadlock.
Explain why in the deadlock detection algorithm, you must mark each process that has a row in the allocation matrix of all zeros?
Because a process without resources cannot be deadlocked.
Does disabling the interrupts guarantee mutual exclusion in multiprocessors? Explain your answer.
Disabling interrupts will not prevent other processes from executing on a different
processor and accessing the critical section at the same time.
How does a monitor guarantee mutual exclusion?
The data of the monitor is only accessible thru its methods and only one process can
call these methods at a particular time.
What is the difference between a livelock and a deadlock?
Livelock = A situation in which two or more processes continuously change their states without doing any useful work.
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.
What is the major disadvantage of the special machine instructions?
Busy wait.
Select the resource that is NOT reusable:
Messages
Select the condition for a deadlock that guarantees that no resource can be forcibly removed from a process holding it:
No preemption
Select the instruction from the correct solution of Peterson's algorithm that represents a busy wait:
while(flag[1] && turn==1);
Select the policy used by a weak semaphore to release the processes in the queue after a signalSem:
Not specified
Choose the most useful combination when selecting the primitives in the message passing solution:
Nonblocking send, Blocking receive
It is the condition in which multiple processes try to get access to a shared resource at the same time:
Racing Condition
Select the type of semaphore that is normally used to create a critical section:
Mutex
The OS needs to be concerned about competition for resources when the processes are:
Unaware of each other
Select the method that must be part of a program based on monitors to solve for the producer-consumer problem:
take_from_buffer();
Select the primitive that is NOT an atomic operation:
a. waitSem()
b. signalSem()
c. Special Machine Instructions
d. None of the above
A restaurant has a single employee taking orders and has three seats for its customers.
The employee can only serve one customer at a time and each seat can only
accommodate one customer at a time. Complete the following function template in a way
that guarantees that customers will never have to wait for a seat while holding the food
they have just purchased.
a) Set the initial values of the semaphores
(5 points).
b) Select the missing instructions after
order_food() and eat() from the
following list (15 points):
1. semSignal(&seats);
2. semWait(&employee);
3. semWait(&seats);
4. semSignal(&employee);

```
semaphore seats = 3;
semaphore employee = 1;
void customer () {
semWait(&seats);
semWait(&employee);
order_food();
semSignal(&employee);
eat();
semSignal(&seats);
} // customer
```

MORE INFO:

Preventing a process / thread from holding more than one resource at a time would be one way to prevent deadlocks, are there any issues with this method? Yes, It requires inefficient implementation, and also does not account scenarios where more than one resource is required by a process. **To prevent hold and wait, we may try and allocate all the resources a process needs at the start of execution. What are some possible issues with this method?** Have to know up front what resources are necessary, Starvation is possible, low utilization of resources, lots of holding. **What is another option to try and prevent hold and wait that is similar to ensuring processes have all their resources at the beginning of execution? This method suffers from the same downfalls as normal preemption.** Processes may not own any resources when requesting resources. eg: have to request all resources needed in batches, no holding. **Ensuring that a process releases all of its resources if it is not able to acquire the resources needed to execute is a method of ensuring?** Preemption to prevent deadlocks. **What are some of the issues associated with ensuring resource release by processes that were not able to acquire the necessary resources?** Some resources should not be pre-empted. Starvation is possible. Under utilization of resources. **What is one method to try and prevent circular wait separate from direct preemption**? Ensure resources are only able to be required in a predetermined order (R1, R2, etc). **What are the pros and cons of ensuring resource acquisition order?** Pro: Easy to implement. Con: Code must be written to maintain the order. **What are the three types of states that one must be aware of in order to avoid deadlocks from occurring?** Safe state: requesting a resource will not risk a deadlock. Unsafe state: deadlock is possible. Deadlock: deadlock has occurred. **What is an issue associated with attempting to detect if a deadlock has occurred?** Algorithms to detect a deadlock can be expensive. **What are the three main options for handling a deadlock once it has been detected?** Notify the user, Terminate the process or process tree associated, Preempt all associated resources. **What are the 5 main goals of effective thread / process synchronization?** Avoid destructive inference, Avoid deadlock, Avoid starvation, Avoid scheduler reliance, Maximize concurrency.

MORE INFO:
Deadlock Detection Algorithm

1). Mark each process that has a row in the allocation matrix of all zeros.
2.) Initialize a temporary vector W equal to the available vector.
3.) Find an index i such that the process i is currently unmarked and the ith row of Q is less than or equal to W. If not such is found, terminate the algorithm.
4.) If such a row is found, mark process i and add the corresponding row of the allocation matrix to W.

Banker's Algorithm

KEY:
A = Allocation
C = MAX
C – A = NEED
R = RESOURCES AVAILABLE
V = R – A

EXAMPLE:
R = [3 1 2 2 1]
A = [3 1 1 2 0]
V = [0 0 1 0 1]

$$C = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccccc} R_1 & R_2 & R_3 & R_4 & R_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array} \qquad A = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} \begin{array}{c} \begin{array}{ccccc} R_1 & R_2 & R_3 & R_4 & R_5 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad R = [3 \; 1 \; 2 \; 2 \; 1]$$

V = [0 0 1 0 1] *SAFE STATE*

A:
1 0 1 1 0
1 1 0 0 0
1 0 0 1 0
0 0 0 0 0
\+ --------------
3 1 1 2 0

C – A: (Need)
0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

AVAILABLE (V)/(Work)
0 0 1 0 1
1 1 1 0 1
2 1 2 1 1
3 1 2 2 1

Solution: Need <= Work, if so - > Work = Work + Allocation

P1 = 0 1 0 0 1 <= 0 0 1 0 1, False
P2 = 0 0 1 0 1 <= 0 0 1 0 1, True -> Work = 0 0 1 0 1 + 1 1 0 0 0 = 1 1 1 0 1
P1 = 0 1 0 0 1 <= 1 1 1 0 1, True -> Work = 1 1 1 0 1 + 1 0 1 1 0 = 2 1 2 1 1
P3 = 0 0 0 0 1 <= 2 1 2 1 1, True -> Work = 2 1 2 1 1 + 1 0 0 1 0 = 3 1 2 2 1
P4 = 1 0 1 0 1 <= 3 1 2 2 1, True

Safe State = <P2, P1, P3, P4> or <P2, P3, P4, P1>

MORE INFO

How might direct avoidance of deadlocks be implemented with regards to the states? User specifies max resource, request/release order when process runs. System checks if it can run safely before executing. **What are some disadvantages to programming to directly avoid deadlocks?** Low utilization, Poor user experience. **What is the most basic algorithm for determining deadlock safety? What are its downfalls?** Resource Allocation Graph (RAG), Does not work when resources have multiple instances. **What is a more general algorithm for determining deadlock safety?** The bankers algorithm**. In what method of handling deadlocks would we allow a deadlock to occur and then carry out some method of handling the deadlock?** Detection and Recovery. **In the detection and recovery methodology, how would a deadlock be detected?** Use method such as RAG or Bankers algorithm to repeatedly check if cycles have occurred or there is an over commitment of resources.