- A forked process may share a Run-Time Stack with its parent. False.
- A forked process may share code space with its parent. True.
- A forked process shares its parent's data space at all times. False.
- A thread may share a File Descriptor Table with its parent. True.
- A thread may share a program counter with its parent. False.
- A thread may share code space with its parent. True.
- A thread may share data space with its parent. True.
- An acceptable solution for implementing mutual exclusion is to let the users disable interrupts before entering a critical section and enable them after leaving the critical section. False.
- Assume that you have globally defined. Show the correct way to do the the following.

```
struct Shared {
char Name[10];
int Value; } S1, S2;

thread_t tid;
void * RtnValue;
void * FooBar(void * arg);
```

1. Store "THREAD 1" in S1's Name field: `strcpy( S1.Name, "THREAD 1" );`
2. Pass S1 in the following call: `pthread_create(&tid,NULL,FooBar, void *S1);`
3. From within the function FooBar, print the Name that was passed inside the struct: `printf("The Name Received = %s\n", ((struct Shared*)arg)->Name);`
4. store 2 times the thread id into the Value field of arg: `((struct Shared*)arg)->Value = 2 * pthread_self()`
5. return the struct via a thr_exit: `pthread_exit( arg );`
6. Have main retrieve the returned value: Use the variable RtnValue since you don't know which thread has exit-ed. Even though `pthread_join()` can be used to wait for a specific thread; there is no way to wait for "any" thread by using the Pthreads library in Linux.
7. Have main print which thread exit-ed and the integer returned: Same.

- Can you build a system where mutual exclusion condition is eliminated? No. It is a resource constraint and most resources (if not all) require it.
- Can you suggest a strategy that can achieve the desired result in the previous question, i.e. no circular wait can occur in the system Assign a number to each resource and require that, at any moment, a process can only request a resource whose number is higher than that of any resource it is currently holding.
- Context switch means a process switching from a "blocked state" to "ready state". False
- Create a variable named bar which stores the output of the 'users' command. `bar=$(users)`
- Determine the output for these commands: variable="Dennis" echo "$variable"? `Dennis`
- Determine the output for these commands: variable="Dennis" echo "$variable"? `$variable`
- Different threads within a process may specify different dispositions for a specific interrupt True
- Even if mutual exclusion is not enforced on a critical section, results of multiple execution is deterministic (i.e. it produces the same results each time it runs). False..
- Explain under what circumstances a spinlock might be less efficient than the alternative. In uniprocessor systems, processes can use CPU one at a time. If a process is "busy waiting" for a long time, then a spinlock might be less efficient.
- Explain under what circumstances a spinlock might be more efficient than the alternative. 1. If you have more processor power than you need (e.g. in multiprocessor systems). Spinlock do not require context switch when a process must wait on a lock, and therefore it is more efficient. 2. when the locks are expected to be held for a short time, a spinlock might be more efficient (saves the context switch time, therefore, preferred).
- Give an example event for each transition to occur. (You need to give at least five example events). {create process}: user starts a new process or a program issues a `fork()` or a `pthread_create()` call. {cpu available}: the process currently occupying CPU stops running and the process in front of the READY queue is scheduled to run {time expires}: currently running process uses up its alloted time slot for this turn (timer interrupt). {event wait}: the process issues an I/O read {event completed}: DMA controller interrupts CPU signalling the completion of an I/O read {exit system}: process terminates or segmentation fault.
- Give an example of something that a process might do to initiate a voluntary context switch. Relate this answer to the Process State graph. When a process initiates an I/O Read/Write or event `wait()`. In the Process State graph, the transition is from Running State to `BLOCKED(Waiting)` State.
- Give an example of something that would cause a process to experience an involuntary context switch. Relate this answer to the Process State graph. When a process is interrupted by an exernal source such as a timer. In the Process State graph, the transition is from Running to Ready State.
- Mach OS is a microkernal. True.
- Give two reasons why we should not give the users the power to disable/enable interrupts in a multiprogrammed computer system. (i) they may (unintentionally) forget to enable them which means all (hardware & software) interrupts will be ignored. (ii) they may intentionally abuse this power and let their processes dominate the system resources.
- Given

```
int X[10];
some appropriate function MyFun
pthread_t Tid;
```

1. Write the call to `pthread_create()`, which executes `MyFun`, which accepts the array `X` as an argument. `pthread_create(&Tid, NULL, MyFun, X)`
2. How can the code segment "if(i < y) cout << foobar(i,y);" be made to behave as if it were atomic? Answer the question by recoding the segment.

```
pthread_mutex_t m=1;
pthread_mutex_lock( &m );
if( i < y ) cout << foobar( i, y
  );
pthread_mutex_unlock( &m );
```

- How does a thread acquire RAM that is NOT shared with other threads in the task? Define local variables and use them only in this thread scope.
- How does a user process get privileged operations performed? The hardware allows privileged instructions to be executed only in supervisor mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction, but rather treats the instruction as illegal and traps to the OS and automatically switches to supervisor mode. When a user program does a system call, the switch to the supervisor mode is done automatically by the OS and the priviledged instructions that are needed by the system routine can be executed. Since the OS code is trusted, there is no harm done by giving access to privilidged instructions this way.
- I've got a variable named `flibble` which contains the string `tom Jerry harry`. Write a command which stores just `tom` into a new variable. `new=$(echo $flibble | cut -f1 -d' ')`
- If a process uses up its allocated time slot, a timer interrupt occurs and the process is placed in a BLOCKED Queue. False. It is placed in the READY queue.
- If mutual exclusion is not enforced in accessing a critical section, a deadlock is guaranteed to occur. False.
- If the shared resources are numbered 1 through N and a process can only ask for resources that are numbered higher than that of any resource that it currently holds, then deadlock can never happen. True.
- If there is no mutual exclusion condition for any resource in the system, then there is no possibility for deadlock. True.
- If we constrain the resource requests in such a way that no cycle can occur in the process-resource graph, then deadlock can be prevented permanently. True.
- In Producer/Consumer problem access to shared buffer must be done in a critical section, but access to "in" and "out" pointers doesn't need to be done inside a critical section. False. "in" and "out" are shared variables. They need to be accessed inside a critical section.
- In Readers/Writers problem, it is possible to write code which is functionally correct but may lead to the starvation of writers. However, it is impossible to write the code such that readers may starve instead of writers. False. It is possible to write such code.
- In a uniprocessor environment, threads waiting for a lock always sleep rather than spin. Why is this true (and reasonable)? In a uniprocessor system, only one thread can run at a time. If a thread is spinning for a lock, it is doing nothing but wasting CPU cycles while waiting for the lock to become available. Instead, it is better that they sleep (in the blocked queue) while waiting so that others can use the CPU. When the lock becomes available, OS will wake them up.
- In dining philosophers problem, which senario may lead to a deadlock situation? If the simulation code is written in such a way that each philosopher first acquires the right fork and then attempts to acquire the left fork, a deadlock may occur. Because it is possible that philosophers may enter a circular wait situation in which each one holds the fork on the right and tries to get the fork on the left (which will never happen).
- In dining philosophers problem, why examining and acquiring a fork is done inside a critical section? Explain by giving an example what may go wrong if critical section is not used. Each fork is shared by two neighboring philosophers. If it is not handled inside a critical section, both philosophers may think that they acquired the same fork and start eating using the same fork. This situation leads to an incorrect simulation.
- In the specification of `pthread_join()` in Linux, there is no way to wait for "any" thread (i.e. the call should specify a particular `thread_id` to join) True. Q. In the "zombie" state, the process no longer exists but it leaves a record for its parent process to collect. True.
- LINUX is designed as a monolithic kernel. True. But, with a new twist for expanding OS functionality: LINUX supports dynamically installable modules. A module can be compiled and installed on a running version of the kernel. This is accomplished by providing system calls to install/remove modules.
- Many modern O.S.s use a microkernel design. What does that mean? Microkernel is a small privileged OS core that provides process scheduling, memory management, and communication services and relies on other processes to perform some of the functions traditionally associated with the operating system kernel. It removes all nonessential components from the kernel, and implements them as system and user-level programs. The result is a smaller kernel. Amoeba, Chorus, Mach, and Windows/NT use microkernel design approach.
- Multiprogramming (having more programs in RAM simultaneously) decreases total CPU efficiency (in comparison to Uniprogramming or Batch processing). FALSE. It increases CPU efficiency (see question above)
- Name at least two important differences between a POSIX thread created through a `pthread_create()` call and a child process created through a `fork()` call. 1) Threads share data space and file descriptors with the other threads and the parent process while forked process doesn't 2) a forked process has a seperate and unique PID and hence a process control block (PCB) while the threads created by a process uses their parent's PCB. 3) it takes less time to create a new thread, less time to switch between two threads within the same process, less time to terminate a thread 4) Thread Library provides more control over the execution of concurrent threads through system calls such as `pthread_yield()`, `pthread_suspend()`, `pthread_kill()`, and `pthread_continue()`
- One of the solutions proposed for handling the mutual exclusion problem relies on the knowledge of relative speeds of processes/processors. False. No solution should rely on such assumptions
- The word 'mutex' is short for mutual exclusion.
- UNIX kernel is designed as a microkernel False. UNIX is a monolithic kernel, meaning that the process, memory, device, and file managers are all implemented in a single software module.
- Unlike time sharing systems, the principal objective of batch processing systems is to minimize response time. False. Principal objective for time sharing is to minimize response time. Whereas, the principal objective for batch processing is to maximize processor use.
- What are the 3 possible dispositions that a process may specify with respect to a signal (interrupt)? 1) ignore signal; 2) run the default signal handler provided by the OS; 3) catch the signal and run the user's signal handler.
- What are the important differences between a unix `fork()` and `pthread_create()`? `fork()`: child process has the separate data space with parent process, but they have the same code space. `pthread_create()`: threads share data space, code space, and os resources, but they have the unique thread ID, register state, stack and priority, PC counter.
- What are the main goals today in the design of operating systems? Convenience for user, efficient utilization of the computer resources (CPU, memory, I/O devices), and expandibility.
- What are the necessary conditions for a deadlock to exist. a) mutual exclusion; b) hold and wait; c) no preemption d) circular wait.
- What does `#!` signify in a shell script? It's a special notation which tells the shell, "After this mark, read the name of the script interpreter i want to use."
- What does it mean to say "`rand()` call is not MT-safe"? MT-Safe : MultiThread-Safe This means that the behavior of the function `rand()` is not stable when two threads try to use it at the same time. To get around this problem, we use MT-safe interfaces (such as `rand_r()`). However, `rand_r()` doesn't exist on some multithreaded operating systems (e.g. LINUX), therefore, you need to execute such calls inside a Critical Section to make sure that only one thread can call the function at any one time.

```
pthread_mutex_t rand_mutex;
pthread_mutex_lock( &rand_mutex )
  ;
int a = rand() % 5;
pthread_mutex_unlock( &rand_mutex
  );
```

- What does it mean to say that "`pthread_mutex_lock(...)` is a blocking call"? If the mutex lock requested by this call is held by another process, the called will be blocked until the lock is released by the current owner (via executing a `pthread_mutex_unlock()` call)
- What does it mean to say that a library or a module is MT-Safe? MT-Safe: MultiThread-Safe which means that the behavior of the function is stable when two threads try to use it at the same time. In other words, a library/module protects its global and static data with locks and can provide a reasonable amount of concurrency.
- What does the command `du` do? figures out disk usage (size of a directory)
- What function is used from within a process to send a signal to a process? `kill()`
- What information is saved and restored during a context switch? Context switch requires saving the state of the old process and loading the saved state for the new process. Process state minimally includes current contents of registers, program counter, stack pointer, file descriptors, etc.
- What is a context switch? Switching the CPU to another process.
- What is a critical section i.e. what makes a section "critical"? In an asynchronous procedure of a computer program, a part that can not be executed simultaneously with an associated critical section of another asynchronous procedure. In general, a code segment in which shared variables, shared file descriptors (shared resources) are accessed is considered a critical section.
- What is a spinlock? When a process is in its critical section, any other process that tries to enter the same CS must loop continuously in the entry code of the CS until the first one gets out. The spinlock is the most common technique used for protecting a CS in Linux. It is easy to implement but has the disadvantage that locked-out threads continue to execute in a busy-waiting mode. Thus spinlocks are most effective in situations where the wait-time is expected to be very short.

```
spin_lock(&lock) /* Critical
  Section */ spin_unlock(&lock)
```

- What is an alternative to spinlocking? Put the process in a wait queue, so it doesn't waste CPU cycles and allow it to sleep until the block is released.
- What is an atomic operation? An operation that can not have it's execution suspended until it is fully completed. Generally, it is a single operation that can not be interrupted or divided into smaller operations.
- What is dual mode operation? User mode vs. supervisor mode.
- What is the difference between `pthread_mutex_lock` and `pthread_mutex_trylock`? `pthread_mutex_lock` is a blocking call. If we try to lock a mutex that is already locked by some other thread, `pthread_mutex_lock` blocks until the mutex is unlocked. But `pthread_mutex_trylock` is a nonblocking function that returns if the mutex is already locked.
- What is the difference between starvation and deadlock? STARVATION: A condition in which a process is indefinitely delayed because other processes are always given preference. DEADLOCK: An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar state.
- What is the effect of executing `pthread_yield()`? `pthread_yield()`: stop executing the caller thread and yield the CPU to another thread
- What is the required function prototype for the function `MyFun()`? `void* MyFun( void* X )`
- What was the original purpose/goal in the design of operating systems? To increase the efficiency of hardware.
- What would you expect to see (what instructions) surrounding the 'critical section' code?

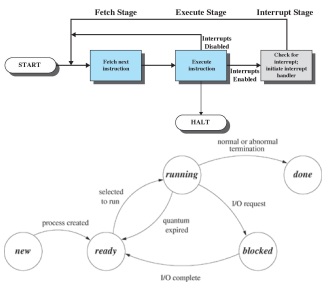
```
pthread_mutex_t mp;
pthread_mutex_lock( &mp );
// CRITICAL SECTION
pthread_mutex_unlock( &mp );
```

- When a process resumes execution after returning from a `fork()`, how can it tell if it is the original process or the new one? By the process ID which is returned by the `fork()` call. If it is equal to 0, it is the new one (child process); if it is a non-zero positive value, it is the original process.
- Which of the following strategies are used for deadlock prevention (circle all that apply)? A. Processes request all of the resources they need at once. They either get it all or get nothing and wait until they are all available. B. If a process needs to acquire a new resource, it must first release all resources it holds, then reacquire all it needs C. Remove the mutual exclusion condition on all of the resources D. Do not let any process hold more than 2 resources at any time. E. Resources are numbered sequentially in a total order. A process X can only ask for Rj if Rj > Ri ∀Ri that X is currently holding. A, B, E.
- Which of the following would not necessarily cause a process to be interrupted? (a) Division by zero (b) reference outside user's memory space (c) page fault (d) accessing cache memory (e) end of time slice (f) none of the above d.
- While DMA (Direct Memory Access) is taking place, processor is free to do other things. The processor is only involved at the beginning and end of the DMA transfer. True.
- Who are the 2 individuals generally credited with the invention of C/Unix? Ken Thompson and Dennis Ritchie.
- Why are context switches considered undesirable (to be minimized) by OS designers? Because context switches waste a considerable amount of CPU time when they save and load the state of the processes.
- Why does a machine need dual mode operation? To ensure proper operation, we must protect the operating system and all the programs and their data from any malfunctioning program. The protection is accomplished by designating some of the machine instructions that may cause harm as "privileged" instructions. Dual mode operations can protect the OS from errant users, and the users from one another. It also prevents the abuse of privileged instructions (such as 'interrupt enable/disable') by the user programs.
- Write a command which lists the processes that are owned by or contain the word foo in their name. `ps -ef | grep foo`
- Write a command which reads from the keyboard and stores it in the variable kbinput `read kbinput`
- Write a loop which reads the file 'foobar' into the variable 'lineIN' one line at a time `cat foobar | while read lineIN (do ... done)`
- Write a shell statement which divide $\frac{3}{2}$ and stores the answer in $average `average=$(echo '3 / 2' | bc)`
- Write a shell statement which first adds the numbers 5 and 7 and stores the result in $sum. `sum=$(echo '5 + 7' | bc)`
- Write an if statement which compares $variable to the integer 5 to see if it is equal `if [ $variable -eq 5 ]`
- Write the command which throws output and errors away while running the script foo.sh. `foo.sh >& /dev/null`
- Write the name of the system call for obtaining the process id of a process in UNIX? `getpid()`
- Write the name of the system call for obtaining the process id of a process' parent in UNIX? `getppid()`
- If I issue the command `someprogram.sh textfile` how do i access `textfile` as a variable? `$1`
- whats wrong with this command: `PATH='cellardoor'? $PATH` is a reserved system variable
- In a single processor system, there is no real multitasking. CPU time is shared among running processes. When the time slice for a running process expires, a new process has to be loaded for execution. Switching from one process or thread to another is called context switch. Process context switch involves saving and restoring process state information including program counter, CPU registers and process control block which is a relatively expensive (in terms of CPU time) operation. Similarly, thread context switch involves pushing all thread CPU registers and program counter to the thread private stack and saving the stack pointer. Thread context switch compared to process context switch is relatively cheap and fast as it only involves saving and restoring CPU registers.
- A semaphore, s, is a nonnegative integer variable that can only be changed or tested by these two atomic (indivisible/uninterruptable) functions:

```
P(s) : [while(s == 0) {wait}; s=s-1;]
V(s) : [s = s + 1;]
```

```
P0() { ...
P(mutex);
balance += amount;
V(mutex);
... }
```

```
int pthread_create(pthread_t *
  thread, const pthread_attr_t *
  attr, void *(*start_routine) (
  void *), void *arg);
attr = { scope, detachstate,
  stackaddr, stacksize,
  inheritsched, schedpolicy } ||
  NULL
```



Kernel a portion of the operating system that includes the most heavily used portions of software. Generally the kernel is maintained permanently in main memory. The kernel runs in a privileged mode and responds to calls from processes and interrupts from devices.

Critical Section in an asynchronous procedure of a computer program a part that cannot be executed simultaneously with an associated critical section of another asynchronous procedure

Preemption reclaiming a resource from a process before the process has finished using it

Concurrent pertaining to processes or threads that take place within a common interval of time during which they may have to alternately share common resources.

Main Memory memory that is internal to the computer system- is program addressable and can be loaded into registers for subsequent execution of processing

Time Sharing the concurrent use of a device by a number of users

Privileged Instruction an instruction that can be executed only in a specific mode usually by a supervisory program

Nonprivileged State an execution context that does not allow sensitive hardware instructions to be executed such as the interrupt disable and I/O instructions

Mutual Exclusion a condition in which there is a set of processes only one of which is able to access a given resource or perform a given function at any time. See critical section.

Starvation a condition in which a process in indefinitely delayed because other processes are always given preference Symmetric Multiprocessing (SMP) a form of multiprocessing that allows the operating system to execute on any available processor or on several available processors simultaneously

Shell the portion of the operating system that interprets interactive user commands and job control language commands. It functions as an interface between the user and the operating system.

Interrupt Handler a routine generally part of the operating system. When an interrupt occurs control is transferred to the corresponding interrupt handler which takes some action in response to the condition that caused the interrupt.

Batch Processing pertaining to the technique of executing a set of computer programs such that each is completed before the next program of the set is started

Secondary Memory memory located outside the computer system itself including disk and tape

Download source on my Github.