1. **What are CPU modes?**

CPU modes are modes that the CPU executes under. The two common modes are user and supervisor mode. User mode is where user programs are executed. Supervisor mode is where the OS runs. Privileged instructions, or those that could harm the OS, are run under supervisor mode.

1. **What is dual-mode execution?**

Dual-mode execution refers to the two modes that a CPU can run under: user and supervisor.

1. **What is a privileged instruction? Why are they needed?**

A privileged instruction is a special instruction that could do harm to the OS. They are needed so the OS can perform actions, but they are protected from the user so the user doesn’t harm the OS.

1. **What is an atomic instruction? Why are they needed? What would happen if multiple CPUs/cores execute their atomic instructions?**

An atomic instruction is one that executes as one uninterruptable unit, without interleaving or interrupts. When an atomic instruction executes, all other instructions are suspended until the atomic instruction finishes. They are needed because they guarantee that an instruction will finish and that the expected results are produced. If multiple CPUs execute atomic instructions at the same time, they will be run sequentially, though the order in which they run can’t be specified.

1. **What is an interrupt and what is a trap?**

An interrupt is an event that requires the attention of the OS. They can be generated by hardware or software. Examples include I/O completion, a key press, dividing by zero, and accessing memory outside the bounds of a program. A trap is an interrupt that is generated by software.

1. **Which one of the following events is an interrupt/trap? Why?**
   1. **Real time clock goes off**

An interrupt because it is generated by hardware.

* 1. **A keypress**

An interrupt because it is generated by hardware.

* 1. **A segmentation fault**

A trap because it is generated by software.

* 1. **A modem dial-up call**

An interrupt because it is generated by hardware.

* 1. **Floating-point exception**

A trap because it is generated by software.

* 1. **Accessing an area not belonging to your program**

A trap because it is generated by software.

* 1. **The completion of an I/O**

An interrupt because it is generated by hardware.

* 1. **Accessing a page in virtual memory that is not in physical memory**

A trap because it is caused by software.

* 1. **A system call**

A trap because it is caused by software.

* 1. **A memory parity error**

An interrupt because it is caused by hardware.

1. **What does it mean to be interrupt-driven?**

Interrupt-driven means the OS is only active when an interrupt has occurred.

1. **Explain the steps of the hardware and operating system when an interrupt occurs.**

1. The current executing process is suspended.

2. Control is transferred to the OS.

3. The OS examines the interrupt and called the specific interrupt handler.

4. Control is transferred back to the process and it resumes.

1. **What is a system call?**

A system call is an operation made available by the OS.

1. **Why is an interval timer needed?**

An interval timer is needed to prevent a program from becoming a CPU hog. When the timer reaches 0, it sends an interrupt to the program so the OS can regain control and decide it the program should continue to execute.

1. **What is a process?**

A process is a program in execution. It has a process ID, state, program counter, register set, code section, data section, and other OS resources attached to it.

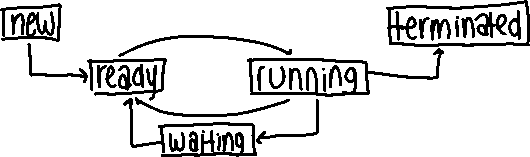
1. **How is process space allocated?**

|  |
| --- |
| Stack – holds local variables |
| Heap – holds dynamic memory allocations |
| Data – holds global data |
| Text/Code – holds program code |

1. **What are process states?**

The process states are new, ready, running, waiting, and terminated. They describe what part of execution a process is currently in.

1. **Draw the process state diagram and elaborate on every state and transition.**



New: the process is being created and resources are being allocated.

Ready: the process has everything it needs to run besides a CPU.

Running: the process is executing.

Waiting: the process is waiting for some event to occur.

Terminated: the process has finished execution.

New Ready: the process has been admitted and is waiting for a CPU.

Ready Running: the scheduler has picked the process and given it CPU control.

Running Ready: an interrupt has occurred.

Running Waiting: the process is waiting for an event to occur.

Waiting Ready: the event has completed, and the process has everything but the CPU.

Running Terminated: the process exits.

1. **What is a PCB? What information should be stored in a PCB?**

A PCB is a process control block. It stores the process ID, process state, program counter, register set, scheduling information, memory limits, list of open files, and other process information.

1. **Can a process assume the system scheduling policy in order to run properly? Why?**

A process can’t assume anything about the scheduling policy in order to run properly because scheduling policy is system dependent and an interrupt or interleaving could occur at any time.

1. **What is a CPU scheduler?**

The CPU scheduler looks through processes in the ready queue and chooses one to allocate CPU time to so the process can execute.

1. **What is the context of a process?**

The context of a process is the runtime information about a process. It includes the process ID, state, register values, program counter, and other file and memory management information for a process.

1. **What is a context switch? Show all needed steps in a context switch.**

A context switch occurs when the CPU scheduler picks a process to run and must allocate the CPU to it. To switch from process A to B, the CPU

1. suspends A

2. activates the CPU scheduler

3. saves the context of A

3. loads the context of B

4. resumes B from the value of its program counter

1. **What is a thread or lightweight process?**

A thread is a basic unit of CPU execution created by a process. It has a thread ID, program counter, register set, and stack but shares the same code and data sections with other threads of the same process.

1. **What are the resources needed to run a thread?**

A thread needs a thread ID, program counter, register set, and stack.

1. **Why are threads cheaper than processes?**

Threads are cheaper than processes because they don’t consume as many system resources which makes it easier to perform context switches between them.

1. **What are the major benefits of using threads?**

The major benefits of using threads are responsiveness, economy, resource sharing, and the utilization of a multiprocessor architecture.

1. **What is a user-level thread?**

A user thread is a thread supported at the user level. The kernel has no knowledge of its existence, which makes it more efficient than kernel threads, but also means that if the containing processes is blocked, all user threads of that process will be blocked. A thread library supports operations such as creation, termination, joining, and scheduling.

1. **What is a kernel supported thread?**

A kernel thread is a thread supported by the kernel, so the kernel is responsible for operations such as creation, termination, joining, and scheduling. It is less efficient than user threads due to system overhead but blocking one kernel thread doesn’t cause peer threads to be blocked.

1. **Describe the major advantages and disadvantages of using user-level and kernel supported threads.**

The advantage of user threads is that they are more efficient, but the disadvantage is that if the containing process is blocked then all user threads will be blocked. The advantage of kernel threads is that if one is blocked, other kernel threads can still run, but the disadvantage is that they are less efficient due to system overhead.

1. **What are the thread models?**

One-to-One: A process had one thread that is associated with one kernel thread. This is the traditional Unix model. Alternatively, one user thread is mapped to one kernel thread. If the kernel thread is blocked, the associated user thread will be blocked.

Many-to-One: Many user threads are mapped to one kernel thread. When the kernel thread is blocked, all associated user threads will be blocked.

Many-to-Many: Many user threads are mapped to multiple kernel threads. Unless all kernel threads are blocked, at least one user thread can run.

1. **How are threads of a process scheduled in each model?**

One-to-One: Scheduling is done by the kernel thread scheduler.

Many-to-One: Scheduling is done by the kernel thread scheduler.

Many-to-Many: Scheduler is done dynamically by the user thread scheduler.

1. **What are the two commonly used thread cancellation methods?**

The two thread cancellation methods are asynchronous and deferred. Asynchronous means that the target thread, or the thread to be cancelled, is terminated immediately. Deferred means that the target thread periodically checks if it should terminate which allows it to terminate in an orderly way.

1. **What is thread-safe and why is it important?**

A library that is thread-safe is one that can be used by multiple threads properly. This is important because when multiple threads are running concurrently, nondeterministic results could occur, and a thread-safe library guarantees certain operations will perform in certain ways.

1. **What is a coroutine?**

A coroutine has multiple exit and entry points so the next call to the routine resumes at the instruction following the previous exit point.

1. **What is a fiber?**

A fiber is a lightweight thread that has a stack, subset of registers, and data but shares other resources with other fibers of the same thread.

1. **What is the relationship between a thread and its fibers?**

A thread is the parent of its fibers.

1. **Why is the use of fibers “cheaper” than threads?**

Fibers are cheaper than threads because they share resources amongst peer fibers.

1. **Why is synchronization needed?**

Synchronization is needed because when multiple processes are running concurrently and operating on the same variable, nondeterministic results can occur since the execution flow could be switched in the middle of execution.

1. **Define race conditions.**

A race condition occurs when two or more processes access a shared resource concurrently and the outcome depends on the order that access takes place.

1. **Does program 1 or program 2 have race conditions?**

Program 1 doesn’t have race conditions since they are no shared resources, but Program 2 could have race conditions since there are shared resources.

1. **What is a critical section?**

A critical section is the section of code where access to a shared resource takes place.

1. **What is mutual exclusion?**

Mutual exclusion means that only one process can execute its critical section at a time.

1. **What are the three conditions that must be met to design a good solution to the critical section problem?**

The three conditions are mutual exclusion, progress, and bounded waiting.

1. **What are the differences between progress and bounded waiting?**

Progress states that if no processes are executing their critical section and a process wishes to enter, then only those processes that are waiting to enter their critical sections can compete, no other processes may influence the decision, and the decision must occur in a finite amount of time. Bounded waiting states that once a process has requested to enter its critical section and before it is granted permission to, there is a bound on how many other processes may enter their critical sections. Progress is different from bounded waiting because progress guarantees that a decision on who can enter will be made in finite time, not that a process will enter its critical section. However, bounded waiting does guarantee that a process will enter its critical section eventually.

1. **What are the differences between finite and bounded?**

Finite means that any number could be written down while bounded means a number can be at most equal to the bound.

1. **Suppose the following processes are running concurrently and x is a variable in shared memory initialized to 0. Assume x must be loaded into a register before being incremented. What are all the possible values for x after both processes have finished their loop?**



|  |  |  |
| --- | --- | --- |
| Process A | Process B | x |
| x=x+1 |  | 1 |
| x=x+1 |  | 2 |
| x=x+1 |  | 3 |
| x=x+1 |  | 4 |
| x=x+1 |  | 5 |
|  | x=x+1 | 6 |
|  | x=x+1 | 7 |
|  | x=x+1 | 8 |
|  | x=x+1 | 9 |
|  | x=x+1 | 10 |

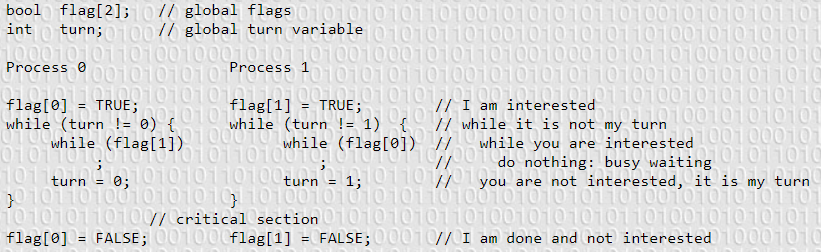
|  |  |  |
| --- | --- | --- |
| Process A | Process B | x |
|  | LOAD | 0 |
|  | ADD | 0 |
| x=x+1 |  | 1 |
| x=x+1 |  | 2 |
| x=x+1 |  | 3 |
| x=x+1 |  | 4 |
| x=x+1 |  | 5 |
|  | SAVE | 1 |
|  | x=x+1 | 2 |
|  | x=x+1 | 3 |
|  | x=x+1 | 4 |
|  | x=x+1 | 5 |

x could equal 2 to 10 inclusive.

1. **Consider a banking system with two functions: deposit(amount) and withdraw(amount). Assume a shared bank account exists between a husband and wife and concurrently the husband calls withdraw() the wife calls the deposit(). Describe how a race condition is possible and what might be to prevent it.**

A race condition is possible because a shared resource, the bank account, is accessed concurrently since the wife and husband perform operations at the same time, and the outcome depends on the order of execution, so if the husband’s process runs first the account might not have enough to withdraw from and if the wife’s process runs first then the results in the bank account will be higher than the husband expected. This could be prevented by adding entry and exit protocols to the calls to withdraw and deposit since both functions perform operations in the critical section of the code. These instructions could also be made atomic so that no interleaving or interruption could occur.

1. **Suppose we have two processes and two shared variables and turn as shown below.**
   1. **Are there any race conditions? Will turn = 0 in P0 and turn = 1 in P1 cause a race condition?**
   2. **Is it possible that both processes be in their critical sections at the same time?**



There aren’t race conditions because it is assumed that updating a variable is atomic.

It is possible for both processes to be in their critical sections at the same time.

.

1. **Which conditions do the first attempt violate and why? Use step-by-step execution to explain your answer.**

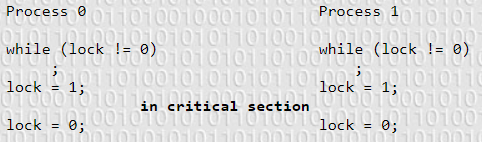
The first attempt violates progress because the attempt forces the processes to run in an alternating way. If Pi sets turn to j when it exits its critical section and then doesn’t re-enter for a while, Pj can only enter once, since it will set turn back to i and Pi won’t enter again to reset j. So, an outside process has influence on the decision of which process can enter its critical section which violates progress.

|  |  |  |
| --- | --- | --- |
| Pi | Pj | turn |
| turn = j |  | j |
|  | while(turn!=j) is false | j |
|  | Critical section | j |
|  | turn = i | i |
|  | while(turn!=j) is true | i |

1. **Which conditions do the second attempt violate and why? Use step-by-step execution to explain your answer.**

The second attempt violates progress and bounded waiting. Progress is violated because if both P0 and P1 set their flags to true at the same time, both will loop in the while loop for an infinite amount of time, and no decision on who can enter the critical section can be made in finite time. Bounded waiting is violated because if P0 exits and P1 fails to detect the change in the value of flag[0], then P0 can enter its critical section again. This can be repeated multiple times and therefore there is no bound on how many times P0 enters its critical section before P1 can enter.

1. **Consider the following solution to the critical section problem. The shared variable lock has an initial value of 0. Show step-by-step that this solution doesn’t satisfy the mutual exclusion condition. Does the solution satisfy progress and bounded waiting?**



|  |  |  |
| --- | --- | --- |
| Process 0 | Process 1 | lock |
| while(lock!=0) TRUE |  |  |
|  | lock = 0 | 0 |
| lock = 1 |  | 1 |
|  | while(lock!=0) FALSE | 1 |
|  | lock = 1 | 1 |
| In CS | In CS | 1 |

If P0 is in its critical section, then lock = 1.

If P1 is in its critical section, then lock = 1.

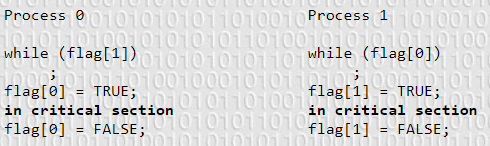
Assume both are in the critical section, then lock = 1.

This fails mutual exclusion because both processes could be in the critical section at the same time.

The solution satisfies progress because…

This doesn’t satisfy bounded waiting because if P0 sets lock = 0 on exit, it’s while loop condition would be false, so it could re-enter its critical section. This could repeat multiple times so there isn’t a bound on how many times P0 enters its critical section before P1 can.

1. **Consider the following solution to the critical section problem. The shared variables in flag are initialized to false. Show step-by-step that this solution doesn’t satisfy the mutual exclusion condition. Does this satisfy progress and bounded waiting?**



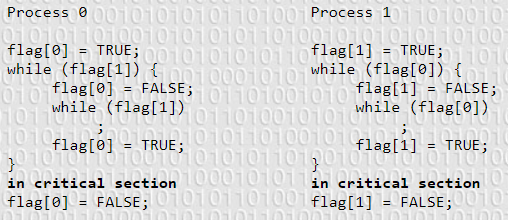
|  |  |  |  |
| --- | --- | --- | --- |
| Process 0 | Process 1 | flag[0] | flag[1] |
|  | Flag[1]=FALSE |  | FALSE |
| Flag[0] = FALSE |  | FALSE | FALSE |
| While(flag[1]) = FALSE |  | FALSE | FALSE |
|  | While(flag[0]) = FALSE | FALSE | FALSE |
| Flag[0] = TRUE |  | TRUE | FALSE |
|  | Flag[1]=TRUE | TRUE | TRUE |
| In CS | In CS | TRUE | TRUE |

This doesn’t satisfy mutual exclusion because if both flag[0] and flag[1] are false then both processes can update their flags to equal true and then enter their critical sections.

This satisfies progress because …

This doesn’t satisfy bounded waiting because if both flags are false, P0 could pass its while loop, update its flag, enter the critical section, and exit resetting flag[0] to false. If P1 never detects a change in flag[0], then P0 could continue to execute in its critical section and therefore there isn’t a bound on how many times P0 can enter its critical section before P1 can.

1. **Consider the following solution to the critical section problem. The shared variables in flag are initialized to false. Does this solution satisfy mutual exclusion, progress, and bounded waiting?**



If P0 is in its critical section, then flag[1] = FALSE, flag[0] = TRUE.

If P1 is in its critical section, then flag[0] = FALSE, flag[1] = TRUE.

Assume both are in their critical sections. Then,

Flag[1] = FALSE, flag[1] = TRUE

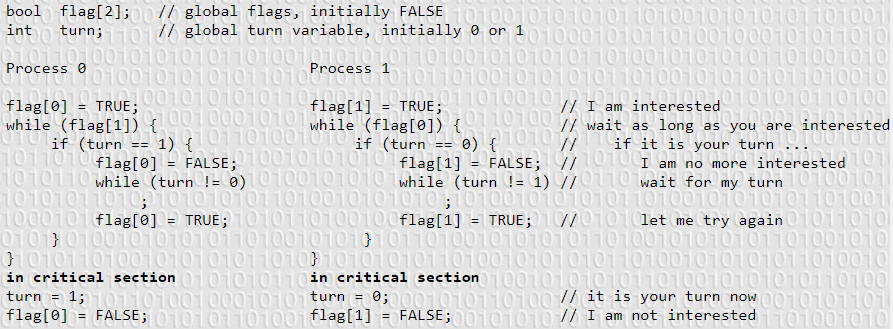
Flag[0] = TRUE, flag[0] = FALSE

This is absurd since a variable can’t hold more than one value so mutual exclusion holds.

This doesn’t satisfy progress because if both processes set their flags = TRUE at the same time, then both processes will be stuck in the while loop and a decision on who can enter their critical section can’t be made in a finite amount of time.

This \_\_\_ bounded waiting because …

1. **Suppose we have two processes and two shared variables flag and turn as shown below.**
   1. **Show this solution satisfies mutual exclusion.**
   2. **Show this solution satisfies progress.**
   3. **Show this solution satisfies bounded waiting. What is the bound?**



If P0 is in its critical section, then flag[1] = FALSE, flag[0] = TRUE.

If P1 is in its critical section, then flag[0] = FALSE, flag[1] = TRUE.

Assume both processes are in their critical sections. Then,

Flag[0] = FALSE and flag[0] = TRUE

Flag[1] = FALSE and flag[1] = TRUE

This is absurd since a variable can’t hold more than one value so mutual exclusion holds.

This satisfies progress because if both processes set their flags to TRUE at the same time, they will both enter their while loops. If the turn variable is not set for the process, then that process will unset its flag, wait for the turn to be set then reset its flag. This allows the other process to enter its critical section since turn will be set for that process and the process can skip into its critical section. Therefore, a decision is made in finite time with no other processes influencing it.

This satisfies bounded waiting because if both processes are waiting to enter their critical sections, one will yield execution of the critical section to the other, and when that process is done, the turn will be changed, and the flag will be reset so the other process can enter its critical section. Therefore, there is a bound of 1 turn for how long a process needs to wait until it can enter its critical section.