Assignment 2

By: Ian Edington
Student ID: 3236986
Date: July 21st, 2017

1. Define short-term scheduler and long-term scheduler, and explain the main differences between them.

Both a short-term and long-term scheduler is involved in deciding when to provide resources to a particular process. The long-term scheduler determines what processes should be allowed to contend for resources. If a system has a new job queue, the long-term scheduler decides which processes are allowed into the ready queue. This has long term implications since a processes will then contend for resources until it completes. This favours a long-term scheduler that makes accurate predictions about available computer resources. It is usually seen on servers or batch processing systems.

The short-term scheduler on the other hand decides which process in the ready queue will receive CPU time. This scheduler is on every computer system, and is fired very often. In contrast to the long-term scheduler the short-term schedulers decision generally has short term consequences since processes usually change quickly in a CPU. This favours a short-term scheduler with a short runtime in order to decrease overhead.

2. Explain the concept of a context switch.

The processes of switching between processes is the basis of multi-programming, which has been shown to increase resource utilization and speed up computer systems. One reason for the increase in productivity is that the CPU doesn't need to wait for a process that has requested IO. When switching from one process to another a processor needs to save the state of the currently running process in order to be able to pick up where it left off the next time it runs the process. The context switch is where the running process is halted, all state associated with the running process is saved in it's Process Control Block, a new process has it's state loaded into the CPU from it's PCB, and the new process starts executing in the CPU. All the times spent in context switching is overhead since the CPU does no useful work while switching.

3. Explain the terms at most once and exactly once, and indicate how these terms relate to remote procedure calls.

"At most once" means that a certain action is guaranteed not to execute more than one time. It might execute zero times or one time but not more than one time.

"Exactly once" means that an action is guaranteed to execute exactly one time. It is not possible for the program to move forward without the action happening and it is impossible for the action to have been done more than one time.

With remote procedure calls (RPC) this concept gives us guarantees about the state of another machine. When an RPC is made over a network there is no guarantee that the RPC will make it to the server. Likewise, there is no guarantee that the acknowledgement (ACK) will make it back to the client. A simple solution to this problem is for the client to wait for a response and if it doesn't receive one within a reasonable time delay to send the RPC again. However, if both RPC make it to the server, the action could be performed twice. This is where the "at most once" and "exactly once" guarantees are useful.

4. Identify and briefly explain each of the four major categories of benefits of multi-threaded programming.

Responsiveness: When a process is waiting for IO or running a long CPU burst, it can be interrupted to give other processes a chance to make progress. This allows systems to react quickly to human input, and more effectively use resources.

Resource sharing: A process can only share information with other processes through shared memory and message passing. Since threads are all part of the same process, the code, data and files are shared between all the threads in the process automatically.

Economy (efficiency): Because of sharing the code, data and files in memory, both making a new thread and context switching between threads is much more efficient than the with processes.

Scalability: A single process with multiple threads can run across all core of a computer, vs a process with a single thread can only run on a single core.

5. Briefly describe the benefits and challenges for multi-threaded programming that are presented by multi-core systems.

Because of the thermal and physical constraints on processor, processors have largely reached their limit in terms of clock speed. The largest future gains in processor performance is in multi-core systems.

The benefit of a multi-threaded program on a multi-core systems is that two or more of a processes threads can be run in parallel. If a program is well designed it can increase it's productivity by a factor of how much of the program is possible to run in parallel. However, it can be very difficult to create a multi-threaded program that functions with out errors. Tasks should be split into large enough chunks that the benefit of parallelism outweighs the cost of thread creation and management, and small enough that it will split the work evenly across all available cores. Any data that is shared between threads has the possibility of becoming corrupted and resulting in unforeseen consequences. Finally, testing and debugging is much more difficult since there isn't a single line of execution to examine but multiple lines split across different cores and memory areas.

6. Define coarse-grained multi-threading and fine-grained multi-threading, and explain their differences.

Coarse-grained and fine-grained multi-threading refer to scheduling algorithms implemented in hardware to switch between the threads in a CPU.

Coarse-grained threading uses an approach similar to cooperative programming in an OS scheduler. One thread runs until it encounters an instruction that will result in a wait (ie. fetch from memory), and then will switch to the next thread.

Fine-grained threading uses an approach similar to preemptive scheduling, where the CPU switches between the two threads at a given time interval.

7. Explain process starvation and how aging can be used to prevent it.

A processes is said to be starving when it doesn't receive the resources it requires to make progress in a given amount of time. If a process continues to starve it might never execute. Since the process will continue to be in memory, and if given the required resources will continue from where it left of, process starvation needs to be defined based on the system requirements. For a batch processing system it might be fine for a process to run every 3 days, where as in a real-time system that same process might be said to be starving.

Processes starvation can occur anywhere there is a priority queue that determines resource allocation. Aging prevents process starvation by increasing the priority of processes that have spent a certain amount of time waiting for a given resource. Any type of aging eliminates starvation since processes will eventually become a high priority task.

8. How does the dispatcher determine the order of thread execution in Windows?

The dispatcher in Windows uses a preemptive priority queue scheduler, weighted based on recent CPU and IO bursts, and limited by a time quantum, in order to determine execution order. There exist variable and fixed priority classes and a process is given a class and a relative priority. If the process is part of the fixed priority class (real-time) then it will always execute before any other class and it's relative priority is fixed within it's class. If the process is part of the variable priority class if can move up and down in relative priority. If it exceeds it's time quantum it will move down in relative priority and if it just returned from an IO burst it will move up in priority.

9. Define critical section, and explain two general approaches for handling critical sections in operating systems.

A critical section is a area of code where a thread modifies a shared data structure. Critical sections only exist in programs that run concurrently since only in concurrent systems can two threads modifying a single data structure. There are two main ways of dealing with critical section, One way to handle critical sections is to switch the processor into a single threaded mode and disable interrupts. This guarantees that the section will not be modified by both threads, however, it has significant overhead that increases with every core in the processor. The other solution is to set a lock on the data in order signal to other threads that the area is currently being modified. A second thread that tries to acquire a lock for the data will be forced to wait until the lock is available.

10. Describe the dining-philosophers problem, and explain how it relates to operating systems.

The dining-philosophers are a group of philosophers, typically five, who site together around a table, and spend their time thinking or eating. When a philosopher decides to eat, they first pick up the chopstick on the left, then the one on the right. However, at each place setting there is only one chopstick between each philosopher, meaning that if the philosopher to either their left or right already has the chopstick, the philosopher must wait for them to be finished eating. The problem arises when all the philosophers decide to eat at the same time. All five of them reach for the left chopstick. Once they have their left chopstick firmly in hand, they all reach for the right chopstick. However, they all find that the right chopstick is being held by the person to their right, and will wait in a circle forever for the person on the right to finish with the chopstick. This is a deadlock.

Deadlocks are seen very often in multi-threaded programs, and are one of the most difficult problems to diagnose and fix in a program.

11. Define the two-phase locking protocol.

Two-phase locking is a specialized type of lock used when more complicated locking logic is required, usually involving multiple lock types on a single object. When a lock has already been acquired by a thread, subsequent requests are examined to determine if that type of lock can be granted, based on the locking protocol.

Read-write locking is the most common form of two-phase locking. The protocol allows many threads to read at the same time, but ensures only one thread has access to the data when it is being written. This is useful when an object is often read and not often written.

12. Describe how an adaptive mutex functions.

An adaptive mutex is an ingenious way to get the benefit of a mutex will mitigating many of the negative aspects. This is accomplished by using multiple implementations of a mutex depending on the best solution at the time.

When a lock fails, it determines the best course of action. For a multiprocessor system, it will spin if the lock is held by a running thread, and will sleep if the lock is held by a sleeping thread. For a single processor system, it will always sleep since the holding thread is definitely asleep. This results in a lock that acts optimally for short code segments.

13. Describe a scenario in which the use of a reader-writer lock is more appropriate than using another synchronization tool, such as a semaphore.

Reader-writer locks have a higher implementation and maintenance cost, however, they allow multiple threads to read the data at the same time. A reader-writer lock should we used when the benefit of reading a piece of data concurrently outweighs the cost of maintaining a more complicated lock. The most compelling example for reader-writer locks is with databases running on many cores, where a piece of data might be accessed by many thread at a time, but can only be written by one at a time. In this case a read-

write lock would outperform a semaphore many times over by allowing all the read operations to happen concurrently, and only the write transactions would need to happen sequentially.

14. What is the difference between deadlock prevention and deadlock avoidance?

Deadlock prevention stops deadlocks from happening by putting restrictions on how locks are acquired. This work without additional information about the system but requires changes in how programs use locks, and can leads to lower resource utilization. Whereas, deadlock avoidance uses additional information about the processes to determine when a deadlock is possible. Given the state of the system and information about which processes might need what resources it is possible to calculate if giving one additional resource could lead to a deadlock. This has the benefit of higher resource utilization and less programmer consideration. However, the algorithm to determine if the state is safe runs in quadratic time and needs to be run every time a resource is request.

15. Describe a wait-for graph, and explain how it detects deadlock.

A wait-for-graph represents which processes are waiting or which other processes. Using this graph it is possible to detect when a system is in deadlock by determining if the graph has any strongly connected components. Every-time a resource is allocated or deallocated the wait-for-graph needs to be updated, which cost a constant time per resource allocation. Since the algorithm for finding strongly connected components runs in linear time (Tarjan, R. E. 1972) this is an efficient way to find if a deadlock exists. However, it is limited in that it only works with resources that single instances.

16. Describe how a safe state ensures that deadlock will be avoided.

If a system is in a safe state it is impossible for a cycle to exists in the resource dependency. For a system to be in a safe state means there there is no sequence of process execution that will result in a deadlock. This is because the resource allocation algorithm doesn't allow resources to be taken that might cause a wait for cycle. Two algorithms exist to determine if a system is in a safe state, the resource-allocation-graph algorithm and the banker's algorithm. However, both these algorithms run in quadratic time and must be run every time a resource is allocated.