CS3510: Operating Systems I

Finals: Autumn 2021

Srivatsan T – CS20BTECH11062

1. Since there is no DMA or Direct Memory access in which the device controller transfers a block of memory from device to main memory or vice versa. DMA happens without the intervention of CPU which is free to switch to a new process and complete it the block is getting transferred and CPU can resume this process once the DMA is done transferring the block and an interrupt in signalled. This is the basic essence of multiprogramming since the CPU can now efficiently handle more than one process concurrently.

Now since DMA is unavailable, the CPU has to be involved in transferring of data from the device to main memory or vice versa. Also, the data transfer occurs byte by byte and not as a block, so it involves a lot more time. Since the occupied with the process even when there is no progress happening in it (Since it's waiting for data), the CPU now can't focus on other tasks as it could with DMA. Thus, a lot of multiprogramming and the efficient completion of multiple tasks by the CPU is lost when DMA is not present in the implementation.

2. The average time to access a word from a machine with a cache, DRAM and a disk is as follows.

Avg. time = Probability that it's gotten from cache * Time to access from cache +
Probability that it's gotten from DRAM * Time to access from DRAM +

Probability that it's gotten from disk * Time to access from disk.

Hit rate signifies the rate that the searched word is available in the storage searched.

Given Cache hit rate = 95%. So, Probability that we get the word from Cache = 0.95

Given DRAM hit rate = 99%. But we move to DRAM only when the cache doesn't contain the word searched.

So, Probability that we get the word from DRAM = Miss rate of Cache * Hit rate of DRAM

= 1- Hit rate of cache * Hit rate of DRAM

= (1 – 0.95) * 0.99

= 0.0495

The disk always contains the word needed but we move to disk only when the word is not available in both cache and DRAM.

So, Probability that we get the word from disk = Miss rate of Cache * Miss rate of DRAM * 1

(Since the disk always contains the word so probability that it's found is 1)

= (1- Hit rate of cache) * (1 - Hit rate of DRAM)

= (1 – 0.95) * (1 – 0.99)

= (0.05) * (0.01)

= 0.0005

Avg. Time = 0.95 * Time for Cache access + 0.0495 * Time for DRAM access + 0.0005 * Time for disk access

= 0.95 * 1ns + 0.0495 * 10ns + 0.0005 * 10ms (Since 1ms = 1000000 ns)

= 0.95 + 0.495 + 5000 ns

= 5001.445 ns

= 5.001445 microseconds is the average time taken to access a word.

3. 30% of the code is given to be serial. We know from Amdahl's Law that the speedup a process is given by

Speedup <= 1/ (S + (1-S)/N)

where N represents the number of processing cores and S represents the fraction of serial portions.

So, the maximum speed up for the process is given by 1/ (S + (1-S)/N).

In this case 30% of the code is serial so S = 30/100 = 3/10 = 0.3

a. N = 4
   Speedup = 1 / (0.3 + 0.7/4)
   = 4 / (1.2+0.7)
   = 4 / (1.9)
   = 2.105
b. N = 8
   Speedup = 1 / (0.3 + 0.7/8)
   = 8 / (2.4+0.7)
   = 8 / (3.1)
   = 2.580

So the speedup achieved using 4 processors is roughly 2.1 times and using 8 processors is 2.5 times.

4.

a.

The server is single threaded. It means there can only be one request that can be handled at a time. So, when the process is waiting to get data from disk, the thread sleeps, and process is halted. Let's find the average time taken for a request to get processed completely.

Given Disk operation necessary 1 / 3 of the times so,

Probability that the request in cache = 2 / 3 &

Probability that disk operation is necessary = 1 / 3

Avg. Time = Probability that request is in cache * time if it's in cache +

Probability that disk operation necessary * time is disk operation is done

= (2 / 3) * 12 + (1 / 3) * (12 + 75)

= 8 + 29 milliseconds

= 37 milliseconds

So, one request to the single threaded server takes 37 ms on an average so it implies that the server handles 1 / 37 requests in 1 millisecond or 1000 / 37 requests in 1 second.

Number of requests handled in 1 sec = 27.02 (or) 27.02 requests/second.

b.

The server is multi-threaded, and another thread executes when one thread sleeps. It means multiple requests can be handled (one by each thread) and can be executed in parallel since another thread executes whenever a thread is sleeping, as is the case when a thread is waiting for disk to output the data required.

We notice that 1 / 12 requests can be serviced in 1ms if the data is in cache which happens 2/3 of the time. And in the remaining 1/3 of the time we complete 1/87 request in 1ms and along with it we trigger another thread to complete a request. So the average number of requests handled can be written as

Avg requests/ms = 2 / 3 (1 / 12) + 1 / 3 (1 / 87 + Avg requests/ms)

So solving this we get Avg requests per second = 100.57 which is significantly higher than the single threaded model.

5.

a. Asynchronous Cancellation of threads mean that the thread responsible for cancellation of other threads executes the cancellation of target thread right after the cancellation was invoked. The thread cancelled might have been in a process and is abruptly stopped from its execution.

Consider a situation in which a process is attempting to populate an array with values it generates by executing a calculation and the process is to stop once the number of array entries exceed a certain number (say 100).

Now the process is divided into multiple threads which share the same memory (the array) and all of them insert values into the shared memory space. Now when the array size exceeds 100 all the threads are cancelled asynchronously. The case might arise that the thread was trying to write into the array in the moment it was cancelled so it might result in the array containing incomplete or junk entries.

Another important problem with threads sharing common memory space is the fact that the OS only reclaims a part of the memory dedicated for the thread once it's destroyed. This might result in not freeing a data resource that is common to and is used by other system processes.

b. Deferred cancellation of threads mean that the threads get deleted only when the cancellation command is received for the target thread, and it has reached a cancellation point which acts as a check point which enables cancellation of a particular thread.

The implementation of Cancellation points is through the blocking system calls in POSIX and standard C libraries. The blocking system calls is what prevents the thread from safely getting cancelled. So, the returning of such system calls mean that the thread is no longer blocked and can be cancelled safely and can serve as valid cancellation points for the threads. E.g., read() system call blocks the thread while getting an input. So, the return of this system call implies that the data is read and the thread is now safe to be cancelled. Now we can call functions like *pthread testcancel()* can be called which tests if such system calls are running and cancels a thread if they're not.

6.

a.

Situation : Child processes are deleted automatically when the parent is deleted.

Modern web browsers like Google Chrome uses a child process to handle each tab opened by the user. Each process then proceeds to execute the instructions from the user for each given tab. Now when the user commands to close the browser , it simulates the termination of parent process. Since every single tab must be closed whenever the browser is closed, all the child processes corresponding to each tab must be terminated (cascade termination) before the parent process can be stopped. In this example having an implementation such that Child processes are deleted automatically when the parent is deleted is beneficial.

b.

Example of situation in which deletion of parent process should specifically not result in deletion of its children processes. Consider a situation in which the parent process creates a child which acts as a checker for parent process and makes sure that the parent process has a smooth termination and doesn't crash. This might be important when the parent process is very critical, and the result of its termination must be sent to other parts of the program.

If the parent process crashes, the child process checking on it must keep running to output the event and completes its task rather than it getting closed as a result of its parent's termination.

7. One such situation in which a single threaded server might be better is when the underlying server is computationally incapable of handling multiple requests and managing threads which might lead to the server getting crashed whenever there are a lot of requests made simultaneously.

Another reason would be to improve the safety of the server which forces us to have only one user getting serviced by the server at a time. For example, consider a web server that provides each of its users with a unique user ID and a password. Such a server might want to have only one user getting an ID and password at a time since the presence of multiple users in multiple threads might decrease the security of such a server since the other threads might try to read other user's details (by using various packet sniffing techniques which in this case is unethical) which would result in a breach of privacy.

So since the server imposes that only one user gets this request serviced at a time, the server works better as a single threaded implementation.