Operating System Exam 2

7th December 2021

Tanmay Garg CS20BTECH11063

Q1. On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

Sol

- DMA or direct memory access is used to improve performance when transferring data between I/O and buffer instead of using an interrupt driven system
- After the process or task sets up all buffers, counters and pointers for the I/O device, the device controller transfers an entire block of data directly to or from the device and main memory, with no intervention by the CPU
- In high end systems, switches are used instead of bus architecture, this helps multiple components talk to each other concurrently instead of waiting for cycles on a shared bus
- In the case of multiprogramming, since more than one process occurs at the same time by switching between the processes
- Since multiple switching and data transfer needs to occur many times during a certain time interval
- In that case DMA will help smooth things out very efficiently as now the Processor does not need to create multiple interrupts interrupt each time a switch needs to occur between the processes
- It can now create a single interrupt and the entire block will be transferred and the process can continue smoothly
- This gives CPU the flexibility to carry out more than one type of task during an I/O task
- If there is no DMA, then the only I/O tasks would be carried out more significantly leading to no advantage of having a multiprogramming system

Q2. Consider a computer system that has cache memory, main memory (RAM) and disk, and an operating system that uses virtual memory. Assume that it takes 1 nsec to access a word from the cache, 10 nsec to access a word from the RAM, and 10 ms to access a word from the disk. If the cache hit rate is 95% and main memory hit rate (after a cache miss) is 99%, what is the average time to access a word?

Sol

Let us

- Cache
  - Time = 1 ns
  - Hit = 0.95
  - Miss = 0.05
- RAM
  - Time = 10 ns
  - Hit = 0.0495
  - Miss = 0.0005
- Disk
  - Time = 10 ms =$10 * 10^6$ ns
  - Hit = 0.0005

Average Time:

$$= 0.95 * 1 + 0.0495 * 10 + 0.0005 * 10 * 10^6$$

$$= 0.95 + 0.495 + 5000$$

$$= 5{,}001.445\ ns$$

Q3. An application has 30% of code that is inherently serial. Theoretically, what will its maximum speedup be using Amdahl's law if it is run on a multicore system with (a) four processors and (b) Eight processors.

Sol

S = 0.3

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

(a) 4 Processors

N = 4

Max speedup = $\frac{1}{0.3+\frac{0.7}{4}} \approx 2.105$

(b) 8 Processors

N = 8

Max speedup = $\frac{1}{0.3+\frac{0.7}{8}} \approx 2.581$

Q4. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 12 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle (a) if the server is single threaded (b) if it is multithreaded assuming that whenever a thread is stuck waiting for data another thread executes?

Sol

(a) Single Threaded
- Thread would have to sleep while it waits for some disk operation
- So, all processes combine to become a sequential process, ie, each process occurs one by one
- So, the thread which do not need any disk operation would still have to wait and be executed in sequential order
    - Average time taken to execute a single threaded process
        - $12 * \frac{2}{3} + (12 + 75) * \frac{1}{3} = 37$ msec
    - Request Handled per sec is
        - $\frac{1}{average\ time} = \frac{1000}{37} \approx 27$

(b) Multi-Threaded
- We can create a thread pool in the case for having multiple threads which we can take as 3
- Let the first thread read data from Disk to store in the cache
- Once this is done, now all the other threads can occur simultaneously in the CPU on different threads
    - Average time taken to execute a single threaded process
        - $(12 + 75) * \frac{1}{3} = 29$ msec
    - Request Handled per sec is
        - $\frac{1}{average\ time} = \frac{1000}{29} \approx 34$

Q5. Linux supports two modes of cancellation: Asynchronous & Deferred. (a) Explain the difficulty with asynchronous cancellation in threads using an example. [3 pts] (b) Deferred cancellation is ensured by cancellation points. When a thread reaches a cancellation point, it gets terminated. How do you think are the cancellation points are internally implemented [4 pts]

Sol

(a) Asynchronous Cancellation
- One thread immediately terminates the target thread
- The problem occurs in 2 cases
  - Resources are allocated to a cancelled thread
  - Thread is cancelled while being in the middle of updating certain values which are shared by different threads
- The Operating System tries to reclaim system resources from a cancelled thread but is not able to do so completely
- Hence, this may not free a system wide necessary resource
- This can leave data in an undefined state leading to fatal crashes
- If the cancellation is called when the thread is performing a heavy operation could cause the program to enter into a state of confusion where the operation had to cancelled midway leading to resource allocation problems and registers and other values being in an undefined state

(b) Deferred Cancellation
- Target thread checks periodically if it should terminate or not on its own gracefully in the case of Deferred Cancellation
- Cancellation occurs only when the thread reaches a safe point, or the cancellation point at which the thread can terminate
- The thread then performs a check on this point to determine if it can be cancelled safely or not
- The read() and write() system calls are cancellation points that allow cancelling a thread that is blocked while awaiting input from read() or output from write()
- In order to establish a cancellation point, pthread_testcancel() function must be invoked
- This function creates a cancellation point inside the calling thread, so a thread which is executing a code that contains no cancellation point will respond to cancellation request
- If a cancellation request is found to be pending, then the function will not return, and the thread will terminate immediately
- Otherwise, the thread will continue to run
- After terminating, a clean up function must be called to free that memory which was originally allocated to the cancelled thread, and this helps in freeing up resources just before the thread was terminated

Q6. In some systems, a spawned process is destroyed automatically when its parent is destroyed; in other systems, spawned processes proceed independently of their parents, and the destruction of a parent has no effect on its children. (a) Give an example where this situation is beneficial a spawned (child) process is destroyed automatically when its parent is destroyed**. [3 pts] (b) Give an example of a situation in which destroying a parent should specifically not result in the destruction of its children. [3 pts]**

Sol

(a) Spawned process destroyed automatically when its parent is destroyed
- Let us say there is a situation when the parent process encounters an undefined behavior while accessing a restricted memory section or doing illegal operations
- Now the child process which might have taken a copy of those values could lead to disastrous situation where the child process could also encounter such a situation and it could also be the case that the child process might update a certain section of memory with that undefined value
- This could lead to fatal consequences such as the entire program crashing or updating a file in an illegal manner leading to the file getting corrupted
- Hence, if a parent process is terminated due to such illegal operations, then all of its child process should also be terminated to avoid such consequences

(b) Destroying a parent should not lead to destroying a child process
- There can be a situation where the parent process has done some operations on a set of variables
- Now the process that spawned a child process, which will operate on that same variable to give some output to other process
- Since the parent process is now completed and does not need to wait for the child process to be completed, it can terminate itself
- When this occurs, the child process can now in itself become a parent process and spawn off other child processes to work on a set of variables that each of them performs operations independently
- For eg, in the case of compressing a particular file, the parent process can prepare the file for compression and its child process can begin the compression process
- So, the parent process can now be terminated and its child process can continue as the parent process required large resources to function, this released resource can now be used by child process or by any other process to further continue their work

Q7. In the class, we described a multithreaded Web server, showing why it is better than a singlethreaded server and a finite-state machine server. Are there any circumstances in which a singlethreaded server might be better? Give an example. [4 pts]

Sol

- The answer should depend on the type of processor system it has
- If the system is a uniprocessor type system, then creating additional threads is a computational expensive process
- Moreover, the context switching between the threads will lead to a significantly reduced performance and efficiency
- In the case of a multiprocessor system, creating additional threads is a very light task and overhead is very minimal
- Several similar type of tasks can be grouped together and can be performed efficiently to improve performance
- In the case when the entire server needs to do operations which only occur within the CPU, then in that case, having multiple threads would unnecessarily complicate a simple task which can be executed in a sequential fashion than making it happen over multiple threads and creating extra CPU overhead
- Generally, servers have a fixed amount of load so having single threaded structure can easily help predict when the server might crash
- Moreover, if a server crashes at any point in time, then it is easier to find faults in the case of a single threaded procedure than in a multi-threaded procedure
- Debugging a multi-threaded program becomes very complicated as each thread would have to paused in the middle and be checked individually for all the variables in correspondence to different threads which are also paused
- Let us say that the entire process that needs to occur, occurs only within the CPU, then if we have some data for a specific field such as name or phone numbers then each of the fields take only finite amount of characters and storing for thousand or more than that for such fields will not take a huge space and can be accessed easily