**Jay Vicory - CS575 BHA: Operating Systems: *Homework #2***

3) In general, assembly language tends to have higher performance than compiled code. This is not always true; however, since the resultant efficiency of the assembly language code is programmer dependent and dependent on the capabilities of the compiler for higher level langauges. In other words, assembly language coding executes faster. Since interrupts are common especially for scheduling systems and since context switching is costly, it makes sense to optimize routines that execute in the interrupt context.

Some CPU registers needed in the interrupt context may also have protected access requiring the execution of a privilege elevation op-code not generally built by compilers.

4) The kernel is, in essence, another process that has it's own local variables, data, program counter, and so forth. Likewise, the kernel also has it's own operating stack.

6) Figure 2-11(a) illustrates several independent processes. Each process operates within it's own data space and address space. Even if each process were operating on the same dataset, the dataset would likely be a copy located in an address space unique to the particular process. As a result, each process context switch would likely generate a cache miss causing the cache to be refreshed relative to the current process. Each subsequent context switch would likewise thrash the cache. The cache, in this example, is only using within the scope of the current process. The longer the process is executing, the more efficient the cache in such a system.

Although a disk access would seem to be a relatively long event, for a set of processes accessing a slow storage device, a blocking access would cause an immediate context switch. Each process would cause a cache refresh and shortly thereafter would context switch when blocked on IO. This action could "thrash" the cache.

35a) Q= infinity. Since the Quanta is infinite, a context switch may occur as soon as the process blocks on I/O or an interrupt occurs. The Quanta size indicates no actual time slicing. Cpu efficiency = [ T / (S + T) ] \* 100 %.

35b) For Q > T, the time slicing scheduler does not cause a time slice context switch before the process blocks on IO. As a result, the process never time slices and is similar to case (a). Cpu efficiency = [ T / (S + T) ] \* 100 %.

35c) S < Q < T. Since Q is less than T, the process will time slice prior to blocking on IO. Cpu efficiency =

Number of task switches = [T/Q].

Time for task switching = S \* [ T/ Q]

Efficiency = ( T / S[ T/ Q] + T ) \* 100 %.

35d) Q = S.

Efficiency = ( T / {Q[ T/ Q] + T} ) \* 100 % = (T / 2T ) \* 100%.

However this could be problematic for a hard-real time system where the quanta would need to include any processing overhead (including time for the scheduler). Where the quanta is the same as the switching time, this may cause a problem for time slicing systems that rely on hardware based timers. This is common on modern operating systems where the "tick" timer is an interrupt based on a time interval. For such a system where Q=S, the process would never execute beyond the initial execution since the during the context switch (after time S), another context switch would immediately occur due to the time slice quanta. As a result, the actual processes would never execute post the initial execution.

35e) Q ~= 0. The quanta is very small relative to T and S.

Efficiency = ( T / S[ T/0] + T ) \* 100 %. Divide by Zero (or almost zero). As a result the S term is very large leading to efficiency -> 0.

39)

40) aT0 + (1 - a)T1. Pg. 158.

a = 1/2

estimates: 40,20,40,15

t0/8 + t1/8 + t2/4 + t3/2 = 40/8 + 20/8 + 40/4 + 15/2 = 5 + 2.5 + 10 + 7.5 = 25

44) The process controls the execution of it's threads, when threads are spawned, and when threads terminate (policy) while the kernel schedules execution of each process by a mechanism unknown (and not controlled) by the process. The kernel controls the mechanism of the system overall. The applications (processes) handles execution policy by using threads.

Extra 1) b > a > 0 where the priority of the process while waiting changes at rate a while running the process priority changes at rate b.

b increases at a greater rate than a. This indicates that the scheduler is running is first in / first out or first come / first served (FCFS).

Extra 2) a < b < 0. b is less than zero and a is less than b. A waiting process will decrease in priority as it waits more than if it is running. A new or "younger" process will always execute first. Last in first out. LIFO.

Extra 3) a = b = 0. All processes have the same priority. Round robin in time slicing system.

Extra 4) a > b = 0. Running processes go to the lowest priority. Waiting process increase in priority. New processes at least priority. Round robin.

Extra 5) a < b = 0. Waiting processes decrease in priority. New processes = greatest priority. Running processes increase in priority. Last in first out (LIFO).

Extra 6) a = b > 0. Old processes have higher priority than new processes. Waiting and running priorities increase at same rate. First in first out (FIFO) or First Come First Served.

Extra 7) a = b < 0. New processes have higher priority than old processes. Last in first out (LIFO).