**Problem Set 1** Matt Kaiser

1. **Suppose we have a round-robin scheduler in the OS kernel of a preemptively time sliced system, and three programs in its ready queue: P1, P2, and P3. P1 takes 10 seconds to complete, P2 takes 8.5 seconds, and P3 takes 3 seconds. Suppose the time slice is 2 seconds, and the overhead for OS context switching is 0.2 seconds. If P1 starts executing first at time 0, when do each of the programs finish executing? Draw a timeline of execution. What is the percentage overhead due to context switching? Assume that a process that finishes early transfers control back to the OS scheduler.**

**Execution Timeline:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Seconds | 2 | 4.2 | 6.4 | 8.6 | 10.8 | 12 | 14.2 | 16.4 | 18.6 | 20.8 | 23 | 23.7 |
| P1🡪 | | |  | 🡪 | | |  | 🡪 | | | 🡪 | | | 🡪 | |\\ |  |
| P2 | 🡪 | | |  | 🡪 | | |  | 🡪 | | | 🡪 | | | 🡪 | |\\ |
| P3 |  | 🡪 | | |  | 🡪 | |\\ |  |  |  |  |  |  |

#the | lines and times at the top correlate stop times of a program not the start time, the 🡪 arrows indicate the start of a program, and the |\\ represents the finishing of a program

* P1 finishes after 23 seconds
* P2 finishes after 23.7 seconds
* P3 finishes after 12 seconds
* Total execution time is 23.7 seconds
* There were 11 context switches
* 0.2/2 = 10% of overhead is due to context switching

1. **Repeat above for a batch mode multiprogrammed OS. Which system, preemptively multitasked or batch mode multiprogrammed finished execution of all three programs the fastest? Under what conditions might the reverse be true?**

**Execution Timeline:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Seconds |  | 10 |  | 18.5 | 21.5 |
| P1 🡪 |  | |\\ |  |  |  |
| P2 |  | 🡪 |  | |\\ |  |
| P3 |  |  |  | 🡪 | |\\ |

#the | lines and times at the top correlate stop times of a program not the start time, the 🡪 arrows indicate the start of a program, and the |\\ represents the finishing of a program

* P1 finishes after 10 seconds
* P1 finishes after 18.5 seconds
* P1 finishes after 21.5 seconds
* The batch mode multiprogrammed os finished first for execution for all three programs because there was no context switching time
* The opposite would be true if the CPU was held up with I/O bound jobs, the CPU would be held up on the jobs and no progress would be made

1. **Explain the differences between the four different kinds of exceptions found in an OS exception table. What is a software interrupt, and how does that differ from a hardware interrupt?**

|  |  |
| --- | --- |
| Exceptions | Differences |
| Trap | Internal exception used as a software interrupt for a system call, control is given to the kernel by the app, and always returns the next instruction |
| Fault | Exception for potentially recoverable error like a stack overflow or page fault |
| Interrupt | Exception due to I/O task completion |
| Abort | Exception for error with no fix, no recovery, often caused by hardware failure |

Software interrupt vs Hardware interrupt:

Software interrupts are usually used as instructions in the software’s instruction set, this just causes a context switch to an interrupt handler. Hardware interrupts force the processor to save the current state of execution and calls execution of an interrupt handler similarly to software interrupts process.

1. **What role does the jump table play in executing a system call?**

System calls are mostly static to a set of functions. When we call functions, and retrieve the necessary pointer to either call a function or jump to native machine code depending on what type of table is used by the function. An example would be the simple\_add and helloworld calls i.e.(syscall(326)) we wrote for the first homework.

1. **Explain in what way overlapping I/O with CPU processing is advantageous. Explain two ways that I/O can be overlapped with CPU execution and how they are each an improvement over not overlapping I/O with the CPU.**

Overlapping I/O with CPU processing is advantageous because it allows the CPU to stay busy. Rather than idling the CPU can run other tasks, when a task is completed by the I/O an interrupt signal is sent letting the CPU know that it is ready to process another task. Two examples being interrupt driven and DMA based I/O. Interrupt driven I/O begins with the CPU setting up an interrupt handler. Next it works in parallel with I/O data transfer to perform necessary tasks. After the I/O data transfer is finished the CPU is interrupted and let known that it should complete the rest of the work to complete the I/O. Direct memory access based I/O works similarly to interrupt driven I/O but it adds the feature that its controller manages data transfer between device registers and memory.

1. **Describe each step of a write() operation to disk, from the application first calling a write() through the OS processing the write() to the final return from the write() call upon completion of the disk operation. Assume interrupt driven I/O operation. You may draw and label a figure if you'd like. Your answer should include components such as the device controller, interrupt handler, device handler, device driver, and any other OS components you deem appropriate to add.**

Steps of a write() operation:

* System call order of operations:
  + CPU calls the write driver
  + CPU checks for device availability
  + CPU sends interrupt request
  + Kernel starts to execute the write
  + The devices controller updates the status to the write driver
  + The write is added to the systems request queue
  + Control is given back to the CPU after the write has been completed
* While device is available:
* The handler for the device is executed
* Data is pulled from the devices controller
* Resolved data is sent back to the CPU
* While the device is unavailable:
* Interrupt is sent out when the device is later available
* The CPU checks what device has sent out that interrupt
* Steps 3-5 are the step from the section while device is available in the same order as previously presented.