CS 1550: Homework 1

Zach Sadler
zps6@pitt.edu

September 16, 2013

Problem 1

Problem 1a

An interrupt-driven I/O produces an interrupt for every single byte of data to be transferred. In addition, the interrupt-driven I/O requires the CPU to be able to perform the necessary overheard of setting up buffers, pointers, and counters for this data transfer. This is terribly inefficient. Instead, a DMA is a physical device that can handle the overhead itself of fetching the necessary buffers and pointers, then generates only a single interrupt when it has fetched the entire block of data. In addition, the DMA does not require the processor to complete its task so the processor can concurrently perform other operations.

Problem 1b

Consider what would happen if the DMA were interrupted during data transfer by the processor accessing the same data. The CPU is designed to be interrupted, but the DMA does not have the ability to be interrupted or have its data corrupted and then recover. Thus if the DMA is not given the highest priority then an interruption by the processor would cause it to fail irreparably. This is undesirable, so we give the DMA highest priority.

Problem 1c

$$\frac{9600 \text{ bits}}{1 \text{ second}} = \frac{9600 \text{ bits}}{1 \text{ second}} \cdot \frac{1 \text{ byte}}{8 \text{ bits}}$$

$$= \frac{1200 \text{ bytes}}{1 \text{ second}}$$

Thus the DMA can access $1,200$ bytes in one second, and the CPU can fetch $1,000,000$ instructions per second. Thus the CPU is slowed down every

$$\frac{\frac{1 \text{ second}}{1200 \text{ bytes}}}{\frac{1 \text{ second}}{1000000 \text{ instructions}}} = 833$$

cycles. So a total of $100/833 = .12\%$ slowdown caused by the DMA, which is very little, especially compared to an interrupt-driven form instead of the DMA.

Problem 2

Problem 2a

Interrupts and time-sharing go hand in hand. Often times during an interactive program, the CPU is waiting on the user for input, or outputting to the user's viewing device (like an LCD screen). During this time the CPU is inactive, merely waiting for an external device to send it information (or for that device to display information to the user). Instead of merely accomplishing nothing during this time, the CPU can switch to other tasks and complete those. When the hardware device has finished or received new input from the user, it sends a physical signal to the CPU telling it that it has new information. In this way, the CPU is able to share time among a number of processes while waiting for interrupts from the hardware devices.

Problem 2b

A trap is a software-generated interrupt. Think of an exception, like divide by zero, or a segfault. In this case code in the user space attempted some illegal procedure and so a hardware bit is set to signify this error case.

Alternatively, traps can be used by a user space program to request access to a kernel operation- this is called a System Call and is often done by library functions. Because the user does not have access to specific driver functions and the like, it must signify through a trap that it would like the kernel to handle an operation for it.

Problem 2c

A trap is a specific example of an interrupt. Whereas an interrupt can be generated by all sorts of hardware, software, and timers, a trap is specifically generated by user-mode code. Interrupts are used constantly to determine when to switch processes, when to perform I/O with devices, and all sorts of operations, but traps have a more specific purpose. A trap occurs typically only when user code would like access to protected locations in the kernel or devices. A user program is able to generate a hardware interrupt by executing a system call and then passing parameters to the kernel for it to complete the work.
To put it simply, a trap is a kind of interrupt that the user creates to enable the kernel to take over and perform some action for the user. Interrupts, however, can be generated by hardware, the CPU itself, and external devices, and are much more common than traps.

Problem 2d

For this question we have to assume that the CPU at some point knows the current time. Whether it was when the computer was created, when Linux was invented, or some arbitrary epoch, it must have some value to correspond to a fixed point in time. From there, the computer can simply run timers for some fixed length and then count the number of interrupts since that point. For example, if the timer goes off every 100ms then we can count the number of 100ms intervals and add that to the fixed point, then have our answer.

Problem 3

Problem 3a

With this distinction, we can seperate access to certain critical features like hardware and other processes' address spaces and limit each individual process' ability to manipulate these features. This is crucially important, so that a poor programmer in CS 0401 doesn't set their monitor on fire when they try to output text or break their printer when they print a document (an important security feature in my opinion).
In addition, trying to access the correct file in a hard drive is an arduous task and a user could screw up and access two adjacent files or a file he does not have permission to read. Speaking of which, we need to be able to limit a process to only access its specific parts of memory allocated to it.
Distinguishing two modes of access to components of the hardware and memory (user and kernel mode) allow us to do all of this and more.

Problem 3b

System calls allow us to perform the complicated and involved process of interacting with hardware and external devices, and treat these devices as if they were logical rather than physical. I have no idea how to locate the correct file in a 64gb Solid State Hard-drive, but I know that between the device controller and the device drivers, the operating system can take care of this process for me.
System calls abstract a lot of work from the user and ensure that the people who made the device are the ones who write the code for working with it (ideally, at least). System calls let me perform operations that can be fatal or corrupting to my computer if I mess them up, with no fear of destroying my hardware.
The actual code which interacts with the devices and does the operations is located in the kernel (or a loadable kernel module), so the user must trap into the kernel through a system call in order to perform these operations. Man, it seems like all these questions are really coming together aren't they! So as a user, I set some parameters to say what I want from the device then pass it through a system call (or trap) to enter kernel mode and let the kernel handle these risky operations for me.

Problem 3c

A, C, and D should be only allowed in kernel mode.
Letting the user disable all interrupts would allow it to have full control over the CPU and never be preempted by other processes.
Setting the time of day clock could disrupt timers and reduce the integrity of timestamps.
Changing the memory map sounds like a horrible thing that would ruin everything. Lets not let users do that.

Problem 4

Problem 4a

I/O_1	P_1	I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n

Table 1: One Job

I/O_1	P_1	I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n	
	I/O_1	P_1	I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n

Table 2: Two Jobs

I/O_1	P_1			I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n			
	I/O_1	P_1			I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n		
		I/O_1	P_1			I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n	
			I/O_1	P_1			I/O_2	P_2	$\cdots$	$\cdots$	I/O_n	P_n

Table 3: Four Jobs

So for one job, the turnaround time is simply TN, by definitions of T and N.
For two jobs, the turnaround time is $T \cdot N + \frac{T}{2} \to TN$ since we can run the second job completely overlapped with the first job, except with a single $\frac{T}{2}$ period of overlap at the end.
For four jobs, the turnaround time is $2TN + \frac{T}{2} \to 2TN$, since it takes $2TN$ time for the first job to finish, plus $\frac{T}{2}$ for the final job to finish at the end.

For one job, the throughput is $1/n$ by definition.
For two jobs, the throughput is $2/n$ since as n increases we are running two jobs simultaneously at all points except for the very first half time interval and the final half time interval, which become much less important as $n \to \infty$.
For four jobs, the throughput is $2/n$ since as n increases we are running two jobs simultaneously at all points except for the very first time interval and the final time interval, which become much less important as $n \to \infty$.

For one job, the processor utilization is $1/2$ since we use the processor exactly half the time.
For two jobs, the processor utilization $\to 1$ since we use the processor all the time except for the very first half time interval and final half time interval.
For four jobs, the processor utilization $\to 1$ since we use the processor for all but the first half time interval and final time interval.

Problem 4b

I/O_1	P_1	P_1	I/O_1	I/O_2	P_2	P_2	I/O_2	$\cdots$	I/O_n	P_n	P_n	I/O_n

Table 4: One Job

I/O_1	P_1	P_1	I/O_1	I/O_2	P_2	P_2	I/O_2	$\cdots$	I/O_n	P_n	P_n	I/O_n	
	I/O_1		P_1	P_1	I/O_1	I/O_2	P_2	P_2	$\cdots$	I/O_n	P_n	P_n	I/O_n

Table 5: Two Jobs

I/O_1	P_1	P_1	I/O_1			I/O_2			P_2	P_2	I/O_2		
	I/O_1		P_1	P_1	I/O_1			I/O_2			P_2	P_2	I/O
		I/O_1			P_1	P_1	I/O_1					I/O_2	P_2
				I/O_1			P_1	P_1	I/O_1				

Table 6: Four Jobs

So for one job, the turnaround time is simply TN, by definitions of T and N.

For two jobs, the turnaround time is $T \cdot N + \frac{T}{2} \rightarrow TN$ since we can run the second job completely overlapped with the first job, except with a single $\frac{T}{2}$ period of overlap at the end.
For four jobs, the turnaround time is $2TN + \frac{T}{2} \rightarrow 2TN$, since it takes $2TN$ time for the first job to finish, plus $\frac{T}{2}$ for the final job to finish at the end.

For one job, the throughput is $1/n$ by definition.
For two jobs, the throughput is $2/n$ since as n increases we are running two jobs simultaneously at all points except for the very first half time interval and the final half time interval, which become much less important as $n \rightarrow \infty$.
For four jobs, the throughput is $2/n$ since as n increases we are running two jobs simultaneously at all points except for the very first time interval and the final time interval, which become much less important as $n \rightarrow \infty$.

For one job, the processor utilization is $1/2$ since we use the processor exactly half the time.
For two jobs, the processor utilization $\rightarrow 1$ since we use the processor all the time except for the very first half time interval and final half time interval.
For four jobs, the processor utilization $\rightarrow 1$ since we use the processor for all but the first half time interval and final time interval.