CPSC 457 – Assignment 2

Lamess Kharfan
10150607
TUT T02

Question 1

a) Define DMA
Direct Memory Access, or DMA, is a special piece of hardware on most modern systems which is used for bulk data movement between the controller and memory with less interrupts to the CPU. Typically, DMA is combined with slow devices/controllers that have limited memory because interrupts generated by them can interrupt the CPU too frequently and can take up too many CPU cycles while saving/resuming the original operation. Instead of the data being sent directly to the CPU, the DMA collects the entire block of data directly and only generates one interrupt per block when the operation has been completed. DMA helps in this way so that the CPU can do other useful things.

b) Define multiprogramming
Multiprogramming allows for multiple jobs to be loaded on different memory partitions so the CPU can execute on one job while it waits for I/O on the others, and in this way the CPU is not sitting idle while it waits on I/O from a job, it is allocated a different one to execute on instead. Multiprogramming speeds up the rate at which programs run, as they can run in parallel instead of consecutively.

c) Would the concept of multiprogramming be practical on a computer that does not support DMA? Why or why not?
No, the concept of multiprogramming would not be practical on a computer that does not support DMA because with multiple programming running at the time, we have multiple programs requesting I/O from slow devices with a small amount of memory that would be sending frequent interrupts to the CPU. DMA is required for the bulk movement of data to minimize the number of interrupts so that the CPU can be doing useful work while waiting for I/O, like executing instructions for some program, rather than getting interrupted often to act on interrupts generated by I/O.

Question 2

a) Describe how a wrapped system call, e.g. read() in libc, invokes the actual system call in the kernel.
Wrapped system calls, such as read(), are provided in libraries like libc to be used by the user applications to invoke the actual system call in the kernel. They are higher level APIs that can be used by user applications. It does so by first having the user application call the wrapped system call, then passes the libraries implementation of the system call to the “System Call Interface” which transfer user mode in the application to kernel mode. Kernel mode then looks up the system call in the system call table via the index associated with the call and executes the kernel’s implementation of the system call. The kernel then, lastly, passes the results of the call back to user mode through the system call interface.

b) Is it essential that a wrapper of a system call is named the same as the underlying system call?
No, it is not essential that a wrapper of a system call is named the same as the underlying system call. For example, printf() is a higher level API from the standard C library that uses the underlying system call write() after doing some formatting to the string given to it. Although the wrapper and underlying system call have different names, but the underlying system call is still able to be used by the wrapper.

Question 3

a) Is it possible for a process to go from blocking state to running state?
No, it is not possible for a process to go from blocking state to running state. When a process is in the blocked state, it is waiting for some event to occur, such as I/O. Once the event has occurred, the process cannot begin running on the CPU immediately. The process must go to the ready state first in order to be scheduled to run on the CPU, and then can transition to the running state when it is the processes turn.

b) What about going from ready state to blocking state?
No, it is not possible for a process to go from ready state to blocking state. When a process is in the ready state, it means it is in queue to be executed by the CPU and is not waiting for any events to occur. A process is in the blocking state if it is waiting for an event occur, such as I/O. Thus, a process cannot go from ready and not waiting for any events to suddenly waiting for an even to occur in the blocking state. In order for the process to go into the blocking state, it must go from the ready state to the running state first, and then to the blocking state when it needs to wait for an event.

Question 4 - What is a context switch? Describe the actions taken, including the information to be saved/restored, during a context switch?

Context switching allows for the illusion of sharing a single CPU among many processes and is essential for any multitasking OS. If the number of CPU's is less than the number of process, context switching is required to implement multitasking. The OS will maintain a context, or state, for each process in the form of PCB. When the OS needs to switch from process A to process B, it will save the state of process A in the PCB of A, and restores the saved state of process B from the PCB of B. The information saved and restored during a context switch includes the CPU registers, data, heap, and stack, which are part of each processes PCB.

Question 7
Results of time utility:

	time ./scan.sh pdf 4	**time ./scan pdf 4**
real	0m0.377s	0m0.241s
user	0m0.063s	0m0.000s
sys	0m0.172s	0m0.047s

Results of strace -c on scan.sh:

#strace -c ./scan.sh pdf 4
./Modern-Operating-Systems.pdf 6323630
./A1/report.pdf 142619
./Tutorials/report.pdf 68404
Total size: 6534653

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.080185	16037	5	1	wait4
0.00	0.000000	0	7		read
0.00	0.000000	0	8		open
0.00	0.000000	0	20	6	close
0.00	0.000000	0	2		stat
0.00	0.000000	0	7		fstat
0.00	0.000000	0	4		lseek
0.00	0.000000	0	15		mmap
0.00	0.000000	0	8		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	16		brk
0.00	0.000000	0	10		rt_sigaction
0.00	0.000000	0	19		rt_sigprocmask
0.00	0.000000	0	1		rt_sigreturn
0.00	0.000000	0	1	1	ioctl
0.00	0.000000	0	5	5	access
0.00	0.000000	0	3		pipe
0.00	0.000000	0	1		dup2
0.00	0.000000	0	1		getpid
0.00	0.000000	0	4		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		uname
0.00	0.000000	0	3	1	fcntl
0.00	0.000000	0	1		gettimeofday
0.00	0.000000	0	2		getrlimit
0.00	0.000000	0	1		sysinfo
0.00	0.000000	0	1		getuid
0.00	0.000000	0	1		getgid
0.00	0.000000	0	1		geteuid
0.00	0.000000	0	1		getegid
0.00	0.000000	0	1		getppid
0.00	0.000000	0	1		getpgrp
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		time
100.00	0.080185		155	14	total

Results of strace -c on scan.c:

#strace -c ./scan pdf 4
./Modern-Operating-Systems.pdf : 6323630
./A1/report.pdf : 142619
./Tutorials/report.pdf : 68404
Total size: 6534653 bytes.

% time	seconds	usecs/call	calls	errors	syscall
100.00	0.000142	142	1		wait4
0.00	0.000000	0	3		read
0.00	0.000000	0	4		write
0.00	0.000000	0	2		open
0.00	0.000000	0	4		close
0.00	0.000000	0	3		stat
0.00	0.000000	0	4		fstat
0.00	0.000000	0	8		mmap
0.00	0.000000	0	4		mprotect
0.00	0.000000	0	1		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	1		ioctl
0.00	0.000000	0	3	3	access
0.00	0.000000	0	1		clone
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		fcntl
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		pipe2
100.00	0.000142		46	3	total

Results Explanation:

The C program, scan.c, is faster than to the bash script, scan.sh. This is because the C program makes much fewer system calls compared to the bash script. As we can see in output of the strace -c utility for both programs, the list of different system calls is much longer for scan.sh, and the total amount of system calls is 155, which is above triple the amount scan.c makes, which is 46. Also, from the results of the time utility, we know scan.c is faster than scan.sh because the amount of time spent is sys is greater for the bash script, and so it is spending more time in kernel mode and spends zero time in user mode. The C program, however, utilizes both user mode and kernel mode, and spends less time in kernel mode. For these reasons, and as shown by the output of strace -c and time, scan.c is faster than scan.sh.