**CPSC 457**

**Spring 2018**

**Assignment 2**

**3rd June 2018**

**Siddharth Kataria**

A1.

(a) DMA:

Direct Memory Access or DMA is a special piece of hardware which is used for movement of bulk data such as disk I/O. Using the DMA, a device controller can transfer entire blocks of data directly to the main memory without any involvements by the CPU. The DMA thus allows certain hardware subsystems to access main system memory, i.e., Random-Access-Memory, independent of the CPU.

(b) Multiprogramming:

When each program has a (virtually) separate program counter and is executing concurrently. In reality, the switching of execution of each process so that it *appears* that the programs are running at the same time. The process of multiprogramming thus allows for multiple jobs to be loaded onto memory partitions so as to let the CPU not be sitting idle whilst doing I/O.

(c) Multiprogramming without DMA:

Since the purpose of DMA is to move bulk data to minimize interrupts. Thus, letting the CPU do more as it waits for the I/O. So, multiprogramming would not be practical as when multiple programs do run concurrently requesting I/O would result in sending frequent interrupts to the CPU consequently slowing it down.

A2.

(a) Wrapped system call invoking a system call in kernel:

Functions provided by the library of C like read() are called wrapped system calls to give the user applications the ability to invoke a system call in kernel by using this higher-level API. The user application first calls this wrapped system call. Secondly, this library implemented system call is passed to the system call interface where the application now switches from user mode to kernel mode. Once in kernel mode, the system searches for the system call in its system call table using the system call indices and then executes the same system call in kernel before returning back the results of the call back to user mode.

(b) Wrapper for a system call and underlying system call:

In systems, it is not important that the wrapper for a system call is the named the same as for the call it invokes in kernel, i.e., the underlying system call.

E.g. printf() is wrapper for the system call write().

A3.

(a) Processes in blocking state and running state:

For a process that is in the blocking state, it is not possible for the process to switch to running state. The process must be ‘unlocked’ first: triggered by an I/O event, to move to the ready queue in the ‘ready state’. Now the process is ready to be executed by the CPU (scheduler dispatch) to switch to the ‘running state’.

(b) Processes in ready state and blocked state:

For a process that is in the ready state, it is not possible for it to switch to the blocked state. Only processes that are being executed by the CPU: in the running state, can be interrupted to switch to a blocked state. As a process is in the ready state, it waits on I/O events to occur.

A4.

Context Switching:

If the number of CPUs is less than the number of processes, context switching is implemented for multitasking on the CPU; it is a vital feature for multitasking on an operating system. Context Switching is taken place in kernel mode and during it the data of one of the process, which is a register in the CPU is saved into a Process Control Block (PCB, a data structure in kernel) while another process is loaded from another register in the CPU from another Process Control Block. Registers play an essential role in context switching.

A6.

strace -c calls:

(For the scan.sh *Bash* scrpit)

[siddharth.kataria@zone08-ed Assignment 2]$ strace -c ./scan.sh docx 3

strace: exec: Exec format error

% time seconds usecs/call calls errors syscall

------ ----------- ----------- --------- --------- ----------------

57.05 0.000255 255 1 1 execve

34.23 0.000153 25 6 6 open

4.70 0.000021 21 1 write

2.24 0.000010 10 1 fstat

1.79 0.000008 8 1 getpid

------ ----------- ----------- --------- --------- ----------------

100.00 0.000447 10 7 total

(For the scan.c *C code* for the same desired outcome)

[siddharth.kataria@zone08-ed Assignment 2]$ strace -c ./scan docx 3

./report.docx : 14187

./test12.docx : 4247

./test1.docx : 4077

Total size: 22511 bytes.

% time seconds usecs/call calls errors syscall

------ ----------- ----------- --------- --------- ----------------

100.00 0.002487 2487 1 wait4

0.00 0.000000 0 3 read

0.00 0.000000 0 4 write

0.00 0.000000 0 10 8 open

0.00 0.000000 0 4 close

0.00 0.000000 0 7 3 stat

0.00 0.000000 0 4 fstat

0.00 0.000000 0 5 mmap

0.00 0.000000 0 4 mprotect

0.00 0.000000 0 1 munmap

0.00 0.000000 0 4 brk

0.00 0.000000 0 1 1 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 0.002487 53 12 total

time calls:

(For scan.sh)

[siddharth.kataria@zone08-ed Assignment 2]$ time ./scan.sh docx 3

./report.docx 14187

./test12.docx 4247

./test1.docx 4077

Total size: 22511

real 0m0.015s

user 0m0.009s

sys 0m0.011s

(For scan.c)

[siddharth.kataria@zone08-ed Assignment 2]$ time ./scan docx 3

./report.docx : 14187

./test12.docx : 4247

./test1.docx : 4077

Total size: 22511 bytes.

real 0m0.011s

user 0m0.004s

sys 0m0.005s

As we can see, the bash file makes less amount of system calls than the c file but takes more time doing it. The c file also spends less time in user mode and kernel mode than the bash file. This is because the bash file directly jumps into kernel to look for such files, sort them and print the out while the C code takes time in declaring memory beforehand and error handling has to be taken care of.