Marc-Andre Fichtel - 30014709
CPSC 457 - Fall 2017

Assignment 2

Due Friday, Oct 13, 2017 at 11:59pm.
Individual assignment. No group work allowed.
Weight: 6% of the final grade.

Question 1:

DMA, short for Direct Memory Access, is used for bulk data movement (i.e. during disk I/O). It is achieved with a special piece of hardware on most modern systems, and works by letting the device controller transfer an entire block of data directly to main memory, without the need for CPU intervention. In this way, only a single interrupt is needed per block of data in order to let the device driver know that the transfer operation is complete. DMA helps speed up slow computers by allowing the CPU to do other things. It can further be used on fast devices that have the potential of overwhelming the CPU.
Multiprogramming describes how computers with a limited number of CPUs can simulate running many processes simultaneously. This is done by allocating a specific amount of time on the CPU for each process in a queue. If the running process is waiting for I/O, another process is run in the meantime. Each process may have one of three states: Running, Ready, and Blocked.

On a machine that does not support DMA, each piece of data sent to main memory generates an interrupt. Depending on the amount of data sent to main memory, this could potentially block the CPU for long times. Multiprogramming would be practical for this scenario: If the data movement operation is done through a process, this process could be appended to the process queue, thus allowing the CPU to work on other processes and keep the computer responsive. The drawback of this is an increase in time it takes to complete the data movement process.

Question 2:

The standard C library provides access to many OS system calls, such as read(). It is a system call wrapper that invokes the read system call. A user application written in C may call the read() function provided by the standard C library. This will call the associated system call in the kernel via the System Call Interface (a table mapping each system call to a unique identifier number). From there, the result is returned via the standard C library to the application.

There often is a strong correlation between a system call wrapper function and its associated system call within the kernel, but it is not essential that their names are the same, because the system call wrapper and the associated system call are not the same. While this does increase readability, system calls' primary feature is that they are optimized, and thus usually written in Assembly. So human-readability is not the first concern of system calls. An example of this is how printf() in the standard C library first performs some formatting and then invokes the write system call.

Question 3:

It is not possible for a process to go from the Blocked (i.e. because it is waiting for some event to occur, such as I/O) to the Running state. It would not make sense to allow this transition. Only a process in the Ready state may transition to the Running state via a scheduler dispatch. A process in the Blocked state must first be unblocked to then transition to the Ready state.

It is also not possible for a process to go from the Ready to the Blocked state. If a process is Ready, then it is not being run currently. For it to become Blocked, some event such as I/O must occur. But if the process is not Running, then no such event can occur for this process, thus this transition is not allowed.

Question 4:

A context switch is an essential feature of multitasking OSes. It creates the illusion of a single CPU (or a limited number of CPUs) being shared among many processes. If the number of processes outnumbers that of the CPUs, then context switching is needed to implement multitasking. It works as follows: The OS maintains a context (a.k.a. state) for each process, usually in the form of a PCB (Process Control Block). This PCB includes information such as process state, program counter, CPU registers, any much more. When the OS then switches from some process A to another process B, A's state is saved in A's PCB, and B's state is restored from B's PCB. A context switch occurs in kernel mode when a process exceeds its allotted time slice, when the process voluntarily yields (i.e. by sleeping), or due to other events such as I/O.

Question 7:

The following are the outputs of the `time` and `strace -c` commands used on the programs made in Questions 5 and 6 (run on my personal laptop running Ubuntu 16.04.1):

```
time ./scan.sh jpg 3                    time ./scan jpg 3
./OW/pmetallica.jpg 393651              ./OW/pmetallica.jpg 393651
./NW/balbopilosum.jpg 324217            ./NW/balbopilosum.jpg 324217
./NW/gbb.jpg 307789                     ./NW/gbb.jpg 307789
Total: 1025657                          Total: 1025657

real    0m0.006s                        real    0m0.003s
user    0m0.000s                        user    0m0.000s
sys     0m0.000s                        sys     0m0.000s
```

strace -c ./scan.sh jpg 3

```
./OW/pmetallica.jpg 393651
./NW/balbopilosum.jpg 324217
./NW/gbb.jpg 307789
Total: 1025657
% time      seconds           usecs/call  calls       errors      syscall
------------------------------------------------------------------------------
-
  0.00      0.000000          0           6                       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           3                       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           0                       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           5                 1     wait4
  0.00      0.000000          0           1                       uname
  0.00      0.000000          0           3                 1     fcntl
  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
```

```
-----------------------------------------------------------------------------
-
100.00      0.000000                     155              14    total
```

```
strace -c ./scan jpg 3
./OW/pmetallica.jpg 393651
./NW/balbopilosum.jpg 324217
./NW/gbb.jpg 307789
Total: 1025657
% time      seconds          usecs/call  calls       errors      syscall
-----------------------------------------------------------------------------
-
  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           15                      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           3                3      access
  0.00      0.000000         0           1                       clone
  0.00      0.000000         0           1                       execve
  0.00      0.000000         0           1                       wait4
  0.00      0.000000         0           1                       fcntl
  0.00      0.000000         0           1                       arch_prctl
  0.00      0.000000         0           1                       pipe2
-----------------------------------------------------------------------------
-
100.00      0.000000                     57               3     total
```

There do not appear to be great differences between the outputs of the `time` command runs between the two programs. While `time scan.sh jpg 3` seemed to produce lower values more often than `time ./scan jpg 3`, consecutive runs of both often produced identical outputs, so there does not seem to be a difference between the runtimes of the two programs. This is to be expected despite the differences in implementation, as both programs perform the same task. The slightly higher runtime of `time scan.sh jpg 3` in some cases, however, is likely due to the higher number of system calls (and thus higher number of call errors) made, as can be seen in the output of `strace -c scan.sh jpg 3` (32 system calls made) compared to that of `strace -c ./scan jpg 3` (17 system calls made).