University of Calgary

CPSC 457 - Principles of Operating Systems

Professor: Dr. Pavol Federl

Assignment #2

Evan Loughlin

Student ID: 00503393

Tutorial Section: T04

TA: Sina Keshvadi

February 12, 2018

Q1

a) Define "DMA".
DMA stands for "Direct Memory Access". DMA is the concept of allowing certain hardware subsystems (such as I/O) to access main system memory (RAM), independent of the CPU. This frees the CPU from needing to perform constant operations to monitor, read from, and control I/O devices. The main computer's CPU sets up the DMA chip by telling it the device and memory addresses involved, how many bytes to transfer, and the direction. The DMA chip is then allowed to perform its task. Once the DMA chip is complete, it causes an interrupt to notify this to the CPU. The feature is useful at any time whereby a CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer.

b) Define "multiprogramming".
Multiprogramming is a form of parallel computing, whereby a single processor's work is split between two or more tasks. This allows multiple programs or processes to run simultaneously. The concept was heavily popularized by third generation operating systems (1965-1980), where it was recognized that a considerable amount of a processor's time was being wasted in the time spent waiting for I/O operations to finish. The solution was to partition memory into several pieces, with a different job in each partition. When a sufficient number of jobs were being processed simultaneously, this allowed the processor to remain busy nearly 100% of the time.

c) Would the concept of multiprogramming be practical on a computer that does not support DMA? Why or why not?
Yes, very much so. This is because the DMA feature is used heavily to take over much of the I/O programming and computing tasks that the CPU would normally be required to do on devices. With Direct Memory Access, the CPU simply needs to instruct the DMA chip of the memory addresses, number of bytes, and direction of transfer along the bus. Then, the DMA chip will complete its task, allowing the CPU to perform other tasks until the DMA chip sends an interrupt to the CPU notifying that its job has been completed. So, without the DMA capability on a computer, multiprogramming would be even more useful and practical. This is because I/O devices can often be relatively slow. Multiprogramming allows for other programs and processes to be computed in the wasted time of waiting for such I/O devices.

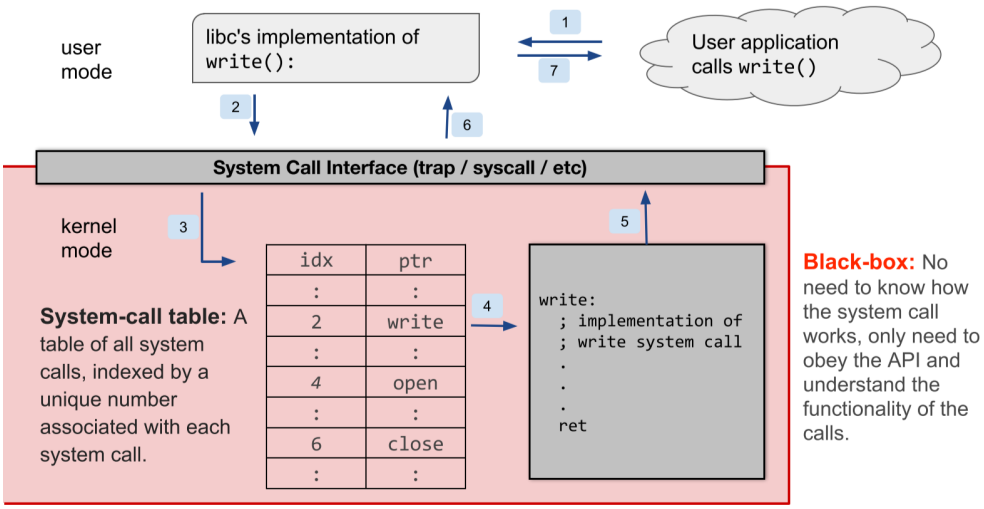
Q2

a) Describe how a wrapped system call, e.g. `read()` in libc, invokes the actual system call in the kernel.

The general way in which a wrapped system call invokes an actual system call, is as follows:

1. The user application calls a wrapped system call, or an API (application programming interface), such as write(), which will then refer to the library's (such as libc) implementation of write().
2. The library's implementation of write() uses a trap to access the kernel mode of the OS.
3. Once in Kernel Mode, a System Call Table is used to look up the index of the system call.

4. The system call is executed by the CPU.
5. The result is computed, and returned to User Mode.
6. The result is returned to the library's implementation.
7. The result is returned to the user's application which called the wrapped system call in the first place.



b) Is it essential that a wrapper of a system call is named the same as the underlying system call?

No. One of the intentions of wrapped system calls is to introduce a level of abstraction that allows users or programmers to utilize system calls more accessibly. From the programmer's perspective, the system call is a black box. There's no need for the programmer to know how the system call works, but simply to obey the rules of the API which calls it. An example of a wrapped system call that uses the same name of a system call is `write()` for `write`. However, an example of a differently named wrapped system call is `printf()` in C, which first performs some formatting prior to using the `write` system call.

Q3

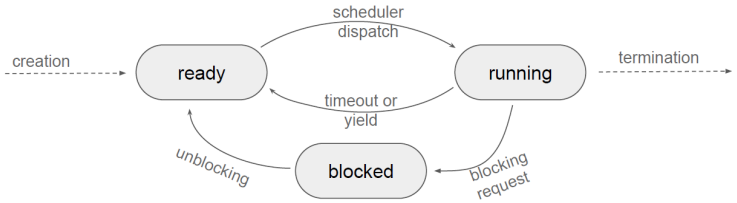
a) Is it possible for a process to go from a blocking state to a running state?

No. It is only possible for the process to go into the running state when it is in the ready state. If a process is in a blocking state, this means that it is unable to run until some external event happens (typically that the process is requiring some input). Once this input has been provided, the process can be unblocked, meaning it can go into the ready state. But it cannot go directly into the running state from blocking, because it's up to the scheduler to dictate how it wishes to allocate processes that are "ready", and the time they are in "running" state. If a process could go from "blocking" directly to "running", then it would immediately interfere and disrupt whatever

process the CPU was currently running at the time, which would not be good. Processes must wait patiently for their turn to be selected by the scheduler.

b) What about going from a ready state to a blocking state?

No. This is also not possible, because a process can only go into the blocking state when the operating system discovers that a process cannot continue until some other event happens. If the process is in the ready state, that means that the CPU is not currently executing instructions for it; and it's inherently not possible for the operating system to suddenly "know" that a process requires some external event to happen if it's not being processed. This is why it can only go into the "blocking" state when it's in the "running" state.



Q4

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

A context switch is the process of saving (and storing) the state of a process or of a thread, such that it is able to be restored; meaning that execution can be resumed from that exact point, at some later time. This fundamental concept allows for a single CPU to manage multiple processes simultaneously.

If a process is to be put into the ready state or the blocking state from the running state, its information need to be saved (pushed onto the stack). This information includes the program counter, program status word, and sometimes one or more registers. Thus, when the other process or interrupt is complete, this information can be popped off the stack, so that the CPU can return to where it left off.

Q5

Refer to `scan.sh` file submitted via D2L.

Q6

Refer to `scan.c` file submitted via D2L.

Q7

Run ‘‘`strace -c`’’ and ‘‘`time`’’ on your bash script from Q5 and your C/C++ program from Q6 and compare the results. Output of these commands are as follows:

strace -c

```
evan.loughlin@zone33-wa:~/Desktop/CPSC457/tut/A2_practice$ strace -c ./scan.sh txt 100
./romeo-and-juliet.txt  138298
./simple.txt    182
./a2temp.txt    87
./Contact.txt   24
Total size: 138591
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 18.30    0.000176           3        52        31 open
 16.63    0.000160          40         4           clone
  9.15    0.000088           3        33         6 close
  8.32    0.000080          10         8           mprotect
  6.96    0.000067           3        23           mmap
  5.72    0.000055           3        20           fstat
  5.51    0.000053           4        15         3 stat
  5.41    0.000052           3        19           rt_sigprocmask
  4.16    0.000040           3        16           rt_sigaction
  3.33    0.000032          32         1           munmap
  3.33    0.000032           6         5         1 wait4
  2.29    0.000022           3         8           read
  1.98    0.000019           6         3           pipe
  1.77    0.000017           6         3         2 ioctl
  1.35    0.000013           3         4           brk
  1.04    0.000010           3         3           lseek
  0.73    0.000007           2         3         1 fcntl
  0.62    0.000006           3         2           prlimit64
  0.42    0.000004           4         1           dup2
  0.42    0.000004           2         2           getpid
  0.42    0.000004           4         1           sysinfo
  0.31    0.000003           3         1           rt_sigreturn
  0.31    0.000003           3         1           uname
  0.31    0.000003           3         1           getuid
  0.31    0.000003           3         1           getegid
  0.31    0.000003           3         1           getpgrp
  0.21    0.000002           2         1           getgid
  0.21    0.000002           2         1           geteuid
  0.21    0.000002           2         1           getppid
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000962                   237        45 total
```

```
evan.loughlin@zone33-wa:~/Desktop/CPSC457/tut/A2_practice$ strace -c ./scan txt 100
./romeo-and-juliet.txt  138298
./simple.txt    182
./a2temp.txt    63
./Contact.txt   24
Total size: 138567
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 83.20    0.005036        5036         1           wait4
  5.15    0.000312          11        29        23 open
  2.86    0.000173          12        14           mmap
  2.48    0.000150          15        10           mprotect
  1.50    0.000091          91         1           clone
  0.78    0.000047           6         8         3 stat
  0.69    0.000042           7         6           close
  0.58    0.000035           6         6           read
  0.51    0.000031           5         6           fstat
  0.50    0.000030           6         5           write
  0.41    0.000025          25         1           execve
  0.36    0.000022          22         1           munmap
  0.31    0.000019           5         4           rt_sigaction
  0.25    0.000015           5         3           brk
  0.23    0.000014          14         1         1 access
  0.10    0.000006           6         1           arch_prctl
  0.08    0.000005           3         2           rt_sigprocmask
------ ----------- ----------- --------- --------- ----------------
100.00    0.006053                    99        27 total
```

time

```
evan.loughlin@zone33-wa:~/Desktop/CPSC457/tut/A2_practice$ time ./scan.sh txt 100
./romeo-and-juliet.txt  138298
./simple.txt    182
./a2temp.txt    63
./Contact.txt   24
Total size: 138567

real    0m0.007s
user    0m0.005s
sys     0m0.006s
evan.loughlin@zone33-wa:~/Desktop/CPSC457/tut/A2_practice$ time ./scan txt 100
./romeo-and-juliet.txt  138298
./simple.txt    182
./a2temp.txt    63
./Contact.txt   24
Total size: 138567

real    0m0.017s
user    0m0.004s
sys     0m0.010s
```

The outputs shown below demonstrate some key features; notably that the C++ implementation tended to require more time, although less overall calls. This could likely be the result of more efficiently designed algorithms from the Linux BASH program, compared to my C++ implementation which included an inefficient sorting algorithm. Whereas the bash file included many short calls to the system, my C++ implementation included only a single large system call, which can be seen above.

Q8

Refer to `sum.c` file submitted via D2L.