CPSC 457 — Principles of Operating Systems
ASSIGNMENT 1

Name: Haohu Shen
UCID: 30063099

1. **Written question (4 marks)**

   Consider a CPU with a 5 stage instruction cycle. For every instruction, the five stages take 3ns, 6ns, 1ns, 10ns and 5ns, respectively. Answer the following:

   (a) How many instructions per second can this CPU execute on average if the stages are not parallelized?

   Solution. If the stages are not parallelized, that is, the instructions are executed sequentially, then for each instruction, suppose it takes t_{total} to execute, then

   $$\begin{aligned} t_{total} &= t_{stage\_1} + t_{stage\_2} + t_{stage\_3} + t_{stage\_4} + t_{stage\_5} \\ &= 3 \text{ ns} + 6 \text{ ns} + 1 \text{ ns} + 10 \text{ ns} + 5 \text{ ns} \\ &= 25 \text{ ns} \\ &= 0.000000025 \text{ second} \end{aligned}$$

   Thus, if we suppose on average the CPU executes S instructions per second, then

   $$\begin{aligned} S &= 1 \text{ instructions}/t_{total} \\ &= 1 \text{ instructions}/0.000000025 \text{ second} \\ &= 40000000 \text{ instructions/second} \end{aligned}$$

   Thus, this CPU can execute 40000000 instructions per second on average if the stages are not parallelized.

   (b) How many instructions per second can this CPU execute on average if all stages are operating in parallel?

   Solution. If the stages are parallelized, that is, the CPU pipelines the operations, then the first instruction will take 25ns to execute, but over the long run, since all units of 5 stages are working independently and in parallel, the time of an instruction cost relies on the slowest stage, thus we can suppose for each instruction it takes $t_{parallel}$ to execute, then

   $$\begin{aligned} t_{parallel} &= \max\left(t_{stage\_1}, t_{stage\_2}, t_{stage\_3}, t_{stage\_4}, t_{stage\_5}\right) \\ &= \max\left(3 \text{ ns}, 6 \text{ ns}, 1 \text{ ns}, 10 \text{ ns}, 5 \text{ ns}\right) \\ &= 10 \text{ ns} \\ &= 0.00000001 \text{ second} \end{aligned}$$

   Thus, if we suppose on average the CPU executes $S_{parallel}$ instructions per second, then

   $$\begin{aligned} S_{parallel} &= 1 \text{ instructions}/t_{parallel} \\ &= 1 \text{ instructions}/0.00000001 \text{ second} \\ &= 100000000 \text{ instructions/second} \end{aligned}$$

Thus, this CPU can execute 100000000 instructions per second on average if all stages are operating in parallel.

2. **Written question (4 marks)**

   (a) from a company's perspective;

   Solution. Virtual machines can help a company to appropriately allocate its resources in a more effective way and improve its financial situation. For example, companies can reduce the needs and the cost on buying many small servers by deploying numerous virtual machines on few big servers, which will improve the average rate of utilization of a server as well. On the other hand, buying less servers means the company can also limit the cost on the maintenance, technical support, power consumption, needs of air conditioners. Moreover, for companies that have high demands on cloud computations or game rendering, using virtual machines could also help them lower the requirement of spaces on building their server farms.

   (b) from a programmer's/developer's perspective;

   Solution. Since it is possible to run multiple operation systems using multiple virtual machines on a single device, virtual machines bring a lot of convenience for developers when they need to test the same product on different platforms but have limited hardware resources. Meanwhile, since different virtual machines are isolated from each other, it is safe to run conflicting applications on different virtual machines and a VM can be quickly built or configured if it is broken. Moreover, for security engineers, a VM is a perfect platform to learn or implement reverse-engineer an unknown computer virus, also the same environment for testing the virus is able to be restored by creating VM snapshots if the guest OS is contaminated. Furthermore, Java programmers and Lua programmers use virtual machines in their jobs since the source code of Java or Lua must be firstly compiled to bytecode by JVM or Lua VM and the bytecode will be interpreted by JVM or Lua VM, which makes Java and Lua portable on different hardware.

   (c) from a regular user's perspective;

   Solution. Since a virtual machine can help isolate the guest OS and the host OS, it shares some characteristics from a sandbox, which will help a regular user to reduce the possibility of leakage of sensitive data and being infected by malware. For example, a regular user can set up a VM with *Whonix*, which integrates *Tor* toolchains into an OS, to protect themselves from privacy leakage while accessing web services. On the other hand, virtual machines also provide a platform for a regular user to taste different operating systems before installing them on the real machine, which flattens the learning curve of switching to a different OS.

   (d) from a system administrator's perspective.

   Solution. Using virtual machines saves a system administrator a lot of time since it is much easier to manage system backups and monitor the system usage on multiple VMs in a single big server instead of multiple servers. Meanwhile, virtual machines give a system administrator much flexibility on the allocation of hardware resources to process tasks with different priorities.

3. **Written question (4 marks)**

(a) Define interrupts.

Solution. An interrupt is a signal generated by the controller in the hardware (such as the controller in the I/O device) as an interrupt request to the CPU to let it know that some important event happens. Once the CPU accepts the request, it will suspend the current state and turn to kernel mode to perform a pre-defined routine called Interrupt Service Routine. If the interrupt is handled, the CPU will restore the state, switch back to the user mode and execute the next instruction.

(b) Define traps.

Solution. A trap is an intentional interrupt generated by software and it is caused by an exception such as segmentation fault. It is also a method to switch from user mode to kernel mode then execute a per-defined routine(operating systems take advantage of it to invoke system calls). If a trap is handled, it will return to user mode and resume the original operations.

(c) What are some key differences between hardware interrupts and traps?

Solution.

- A hardware interrupt is generated asynchronously due to external devices such as disk controller, programmable interval timer, while a trap is generated synchronously by the CPU's control unit due to internal events such as an exception happens in a user process or system calls.
- A hardware interrupt is generated by unknown event such that the time of the event is unknown and unpredictable, while a trap occurs as a result of execution of a machine instruction.
- On the other hand, both a hardware interrupt and a trap share some features, they both store the current state of the CPU before switching to the kernel mode, then execute a kernel pre-defined routine. Also, they both return to the user mode and continue to execute the original operation.

(d) Why are interrupts handled in kernel mode instead of user mode?

Solution. Because in user mode accessing hardware is limited and must be done by calling APIs (wrappers of system calls) given, while the hardware can be communicated without any restrictions in kernel mode. For a hardware interrupt, it must directly send a signal to CPU without restrictions; for a software interrupt, such as an exception, it will access the hardware by invoking APIs. Also, a routine pre-defined by OS must be executed by both type of interrupts, which requires interrupts must be handled in kernel mode.

4. **Written question (6 marks)**

(a) What are the outputs of the *time* commands? Copy/paste this from the terminal output to your report.

Solution. The output of the first command is:

```
haohu.shen@csx:~/a1$ time ./simple_wc < a-tale-of-two-cities.txt
  16272  138883  804335

real  0m1.146s
```

```
user  0m0.297s
sys   0m0.841s
```

The output of the second command is:

```
haohu.shen@csx:~/a1$ time wc < a-tale-of-two-cities.txt
 16272 138883 804335

real  0m0.017s
user  0m0.016s
sys   0m0.000s
```

(b) How much time did the C++ program and *wc* spend in the kernel mode and user mode, respectively?

Solution. From the output above, we can see that *wc* spent 0.000s (less than 0.001s but not actually 0s) in the kernel mode and spent 0.016s in the user mode, while the C++ program spent 0.841s in the kernel mode and spent 0.297s in the user mode.

(c) Why is the *wc* program faster than the C++ program?

Solution. In order to benchmark *wc* and *simple_wc.cpp*, we firstly compile *simple_wc.cpp* as *simple_wc*, then use *strace* utility with the $-c$ command line option to test these 2 programs with the same text file and we have the outputs below:

```
haohu.shen@csx:~/a1$ strace -c ./simple_wc < a-tale-of-two-cities.txt
  16272  138883 804335
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    9.432204          11    804340           read
  0.00    0.000061          61         1           write
  0.00    0.000033           3        10           mprotect
  0.00    0.000019          19         1           munmap
  0.00    0.000013           4         3           brk
  0.00    0.000009           1         6           fstat
  0.00    0.000000           0         5           close
  0.00    0.000000           0        14           mmap
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         5           openat
------ ----------- ----------- --------- --------- ----------------
100.00    9.432339              804388         1 total
```

```
haohu.shen@csx:~/a1$ strace -c wc < a-tale-of-two-cities.txt
 16272 138883 804335
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 47.56    0.000253           7        36        19 openat
 18.23    0.000097           5        18           mmap
 14.29    0.000076           4        19           fstat
 12.59    0.000067           3        20           close
```

```
  6.58    0.000035           0        54           read
  0.75    0.000004           4         1           fadvise64
  0.00    0.000000           0         1           write
  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           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000532                   161        20 total
```

Thus we can see the reason why *wc* program runs faster than the C++ program is that *wc* only makes 161 system calls in 0.000532 seconds, while the C++ program makes 804388 system calls where almost 100.00% of the time was spent on *read* in 9.432339 seconds. More specifically, from the 31st line of *simple_wc.cpp* we can see that the routine executes *read(int, void *, size_t)* every time to process a character from the standard input until EOF is reached, which makes a large amount of system calls.

6. **Written question (2 marks)**

(a) Is your program from Q5 faster than *simple_wc.cpp*? Why do you think that is?

Solution. My program from Q5 is faster than *simple_wc.cpp*. And the outputs of *strace -c* and *time* on my program with the text file for testing, which is the same as that in the previous question, is shown below:

```
haohu.shen@csx:~/a1$ strace -c ./myWc < a-tale-of-two-cities.txt
  16272  138883 804335
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 59.22    0.000562          93         6           read
 13.80    0.000131           9        14           mmap
 10.54    0.000100          10        10           mprotect
  5.80    0.000055          11         5           openat
  2.53    0.000024           4         5           close
  2.21    0.000021           3         6           fstat
  1.69    0.000016          16         1           munmap
  1.58    0.000015           5         3           brk
  1.05    0.000010          10         1           arch_prctl
  0.84    0.000008           8         1         1 access
  0.74    0.000007           7         1           execve
  0.00    0.000000           0         1           write
------ ----------- ----------- --------- --------- ----------------
100.00    0.000949                    54         1 total
```

```
haohu.shen@csx:~/a1$ time ./myWc < a-tale-of-two-cities.txt
  16272  138883 804335

real  0m0.011s
```

```
user  0m0.007s
sys   0m0.002s
```

Thus, we can see it is way faster than *simple_wc.cpp* in kernel mode, user mode and elapsed real time from *time* and it only makes 54 system calls totally in 0.000949 seconds. More specifically, from the source code of *myWc.cpp* you can see a 1 MiB buffer of *char* type is initialized in the data segment and when the input from standard is processed by making a system call using *read(int, void *, size_t)* to read up to 1MiB characters into the buffer array at a time, which will significantly reduce the number of making system calls.

(b) Is your program faster or slower than *wc* and why?

Solution. From the output in 4(a) we can see that base on the same text file given, my program runs faster and spent less than half of time that *wc* takes in user-mode but slower in kernel mode. On the other hand, my program makes 54 system calls in 0.000949 seconds, which is slower than *wc* that makes 161 system calls only in 0.000532 seconds. Base on the implementation of *wc* from *Github Repo of GNU Coreutils* The reasons why my program is faster in user mode but slower in kernel mode may include:

- From the source code given, the application *wc* does a lot of things before parsing the text given, including setting the locale, parsing and verifying the arguments given, checking the type of the file given, which takes extra time and thus slower than my program in user mode since I have not considered other complicated scenarios in my program.
- When *wc* is parsing, it uses buffer with GNU Coreutils' own wrapper *size_t safe_rw (int, void const *, size_t)* to avoid directly calling *read(int, void *, size_t)* in a smart way, as well as using the page size and some other tricks. All of these help it take advantage of the kernel and thus has fewer time in the kernel mode. On the other hand, the compiler also optimized the code of *wc.c* when appropriate flags were given.