CPSC 457 – Spring 2019
Assignment 1 – Tutorial 1

Eric Austin – 30037742

May 17, 2019

1

1.1

If the stages are not parallelized, then each instruction takes $6+4+2=12$ns. Given that there are 1 billion nanoseconds per second, the CPU can execute $1,000,000,000/12 = 8.333... * 10^7 \approx 83,333,333$ instructions per second.

1.2

If the stages are parallelized, then the execution is bottlenecked by the fetch stage since the decode and execute can run during this memory access. This takes 6ns, so the CPU can execute $1,000,000,000/6 = 1.666... * 10^8 \approx 166,666,667$ instructions per second.

2

2.1

One benefit of using Virtual Machines from a company's perspective is the lower cost / saving of money. Rather than buy hardware for each of the different servers that would have to be set up to run different types of Operating Systems, simply buy one big server that can run VMs for each OS.

2.2

One benefit of using Virtual Machines from a developer / programmer's perspective is the ability to test a program for portability (ie the ability to run on multiple Operating systems) from one machine. Rather than having to own multiple machines running different OSs and installing the program on each for testing, the programmer can simply boot up the different OSs on one machine using a VM and test from there.

2.3

One benefit of using Virtual Machines from a regular user's perspective is the extra security that can be gained from running things in a VM. For example, a user could set up a VM to open a suspect email attachment. If the attachment contains a virus or malicious software, the user can just shut down and delete the VM without risking an attack on his or her actual system.

2.4

One benefit of using Virtual Machines from a system administrator's perspective is the ease of setting up or shutting down servers. If each server must be set up individually on separate hardwar, a new server would involve the SA physically going to the server room to configure the hardware and start the server. Using VMs, the SA can set up a new server with some clicks and typing from the comfort of his or her office.

3

3.1

An interrupt is a mechanism to let the CPU know that something important has happened and to 'interrupt' the current activity to handle the important event. An interrupt is generated by hardware, eg. the disk drive interrupts the CPU to let it know requested data has finished being read.

3.2

A trap is similar to an interrupt in that it is a mechanism to let the CPU know that something important has happened and to 'interrupt' the current activity to handle the event, however the source of a trap is software rather than hardware. The CPU is interrupting itself since the source of a trap is the execution of an instruction, eg. a system call.

3.3

Some key differences:

- Hardware interrupts are external events delivered to the CPU from I/O devices, timers, user input, etc. while traps are internal events triggered by the CPU's execution of an instruction like a system call.
- Hardware interrupts are asynchronous with the current activity of the CPU while traps are synchronous with the current activity of the CPU.
- Hardware interrupts are unpredictable (don't know when a key will be pressed or a disk drive finished reading/writing), while traps are the result of a machine instruction (loaded into memory).

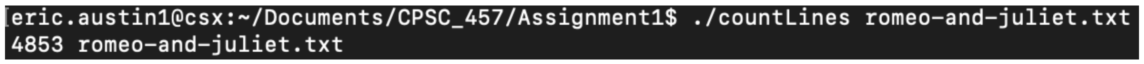
3.4

The Operating System acts as a layer between application programs and the hardware of the computer. Application programs run in user mode and are not able to communicate directly with I/O or external devices. So when a hardware interrupt is generated by some device, the CPU must switch into kernel mode so that the OS can handle the interrupt. Similarly, if an application needs to interact with an I/O device (read from file, print to screen, etc.), it needs to make a system call to the OS via a trap (software interrupt) so that the OS can communicate with the I/O device and the CPU must be in kernel mode.
Whether it is hardware interrupting the CPU or software making a system call to interact with hardware, the interrupt must be handled by the OS and the instructions / operations to do so can only be done in kernel mode, not user mode.

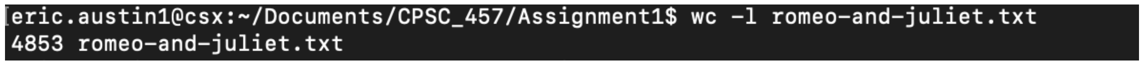
4

4.1

Output of running programs:



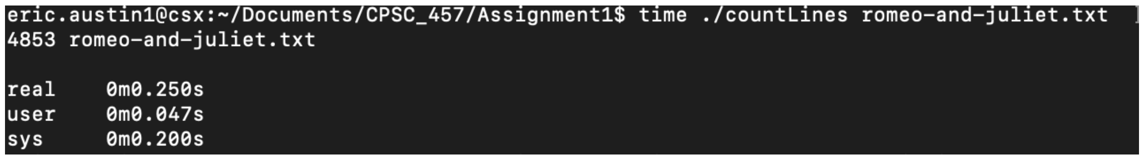
```
[eric.austin1@csx:~/Documents/CPSC_457/Assignment1$ ./countLines romeo-and-juliet.txt
4853 romeo-and-juliet.txt
```



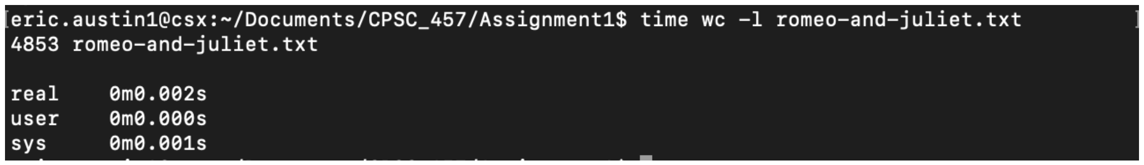
```
[eric.austin1@csx:~/Documents/CPSC_457/Assignment1$ wc -l romeo-and-juliet.txt
4853 romeo-and-juliet.txt
```

4.2

Output of timing programs:



```
eric.austin1@csx:~/Documents/CPSC_457/Assignment1$ time ./countLines romeo-and-juliet.txt  ]
4853 romeo-and-juliet.txt

real    0m0.250s
user    0m0.047s
sys     0m0.200s
```



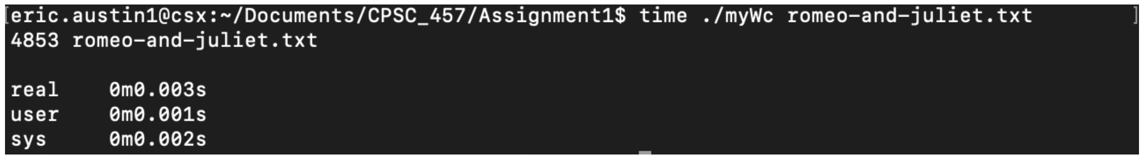
```
[eric.austin1@csx:~/Documents/CPSC_457/Assignment1$ time wc -l romeo-and-juliet.txt        ]
4853 romeo-and-juliet.txt

real    0m0.002s
user    0m0.000s
sys     0m0.001s
```

The countLines program spends 0.200 seconds in kernel mode and 0.047 seconds in user mode while the wc -l only spends 0.001 seconds in kernel mode and 0.000 seconds in user mode. The wc -l program is a couple of orders of magnitude faster than countLines.

4.3

The wc program is much faster than countLines.cpp because countLines reads data from the text file one char (byte) at a time. The romeo-and-juliet.txt file is 178,983 bytes, so we are making 178,983 I/O system calls to read from the file. Since I/O is slow, all of these I/O operations slow the program down.

5



```
[eric.austin1@csx:~/Documents/CPSC_457/Assignment1$ time ./myWc romeo-and-juliet.txt        ]
4853 romeo-and-juliet.txt

real    0m0.003s
user    0m0.001s
sys     0m0.002s
```

The improved myWc spends 0.002 seconds in kernel mode and 0.001 seconds in user mode. This performance is much closer to wc -l than the original readLines.cpp because rather than read one char (byte) at a time we are now reading 1000 chars (bytes) each time, reducing the number of I/O operations greatly.