# CS 374M
# Programming Assignment 1

Ved Khandekar
20d170019@iitb.ac.in
Department of Mechanical Engineering

February 5, 2022

## Question 1

(a) The output shows that there are 8 processor cores in the system. Actually the CPU has only 4 processor cores, but since hyper-threading is enabled each physical core is seen as two "virtual cores" by the OS. Since the output is quite large, only the one processor's entry is shown below.

```
$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 165
model name  : Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz
stepping    : 2
microcode   : 0xea
cpu MHz     : 800.025
cache size  : 8192 KB
physical id : 0
siblings    : 8
core id     : 0
cpu cores   : 4
apicid      : 0
initial apicid: 0
fpu           : yes
fpu_exception : yes
cpuid level   : 22
wp            : yes
flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
   cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
    syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon
   pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf
   pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma
    cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
   tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3
   dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs ibpb
   stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid
   ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
```

```
   mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec
   xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify
   hwp_act_window hwp_epp pku ospke md_clear flush_l1d
   arch_capabilities
vmx flags   : vnmi preemption_timer posted_intr invvpid
   ept_x_only ept_ad ept_1gb flexpriority apicv tsc_offset vtpr
   mtf vapic ept vpid unrestricted_guest vapic_reg vid ple pml
   ept_mode_based_exec
bugs        : spectre_v1 spectre_v2 spec_store_bypass swapgs
   itlb_multihit
bogomips    : 4999.90
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

(b) The following table lists the operating frequency of each core.

| CPU Core | Frequency (MHz) |
|---|---|
| cpu0 | 2500.000 |
| cpu1 | 2500.000 |
| cpu2 | 2500.000 |
| cpu3 | 2500.000 |
| cpu4 | 2500.000 |
| cpu5 | 2500.000 |
| cpu6 | 800.090 |
| cpu7 | 800.018 |

(c) We can check `/proc/meminfo` to find the installed and free physical memory.

```
$ cat /proc/meminfo | head
MemTotal:         7955676 kB
MemFree:          1723008 kB
MemAvailable:     5048280 kB
Buffers:           128056 kB
Cached:           3833508 kB
SwapCached:             0 kB
Active:           2625796 kB
Inactive:         2660112 kB
Active(anon):        5864 kB
Inactive(anon):   1787292 kB
```

From the above output we find that total installed physical memory is 79,55,676 kB. In order to find "free" memory, we must add `MemFree` and `MemAvailable`, since this will give the total memory that can be used before swapping begins. Thus amount of free memory available is 67,71,288 kB.

(d) We can check the output of `/proc/stat` to find the number of forks and context switches. In my case, the output is as follows.

```
[ved@fedora report]$ cat /proc/stat
cpu  263983 2668 69159 3985513 42220 48447 9688 0 0 0
cpu0 25850 368 8885 2011639 29672 21847 2838 0 0 0
cpu1 31625 509 12561 281841 1938 2021 1121 0 0 0
cpu2 34862 375 8405 282210 898 1951 960 0 0 0
```

```
cpu3 35060 302 8279 280197 2218 1963 1073 0 0 0
cpu4 32752 338 7443 281709 2490 9741 824 0 0 0
cpu5 36276 228 7840 282361 2108 1780 880 0 0 0
cpu6 32215 246 7486 282502 2013 6449 841 0 0 0
cpu7 35340 298 8258 283050 879 2692 1145 0 0 0
...
ctxt 58698050
btime 1644048843
processes 16324
procs_running 1
procs_blocked 0
softirq 12678965 1029030 1900692 77 232437 77959 28 28772 5092297
    0 4317673
```

From this output we see that 5,86,98,050 context switches have happened across all CPU cores. Moreover at most 16,324 forks have been made. The "processes" line gives the number of processes and threads created, which includes (but is not limited to) those created by calls to the `fork()` and `clone()` system calls.

# Question 2

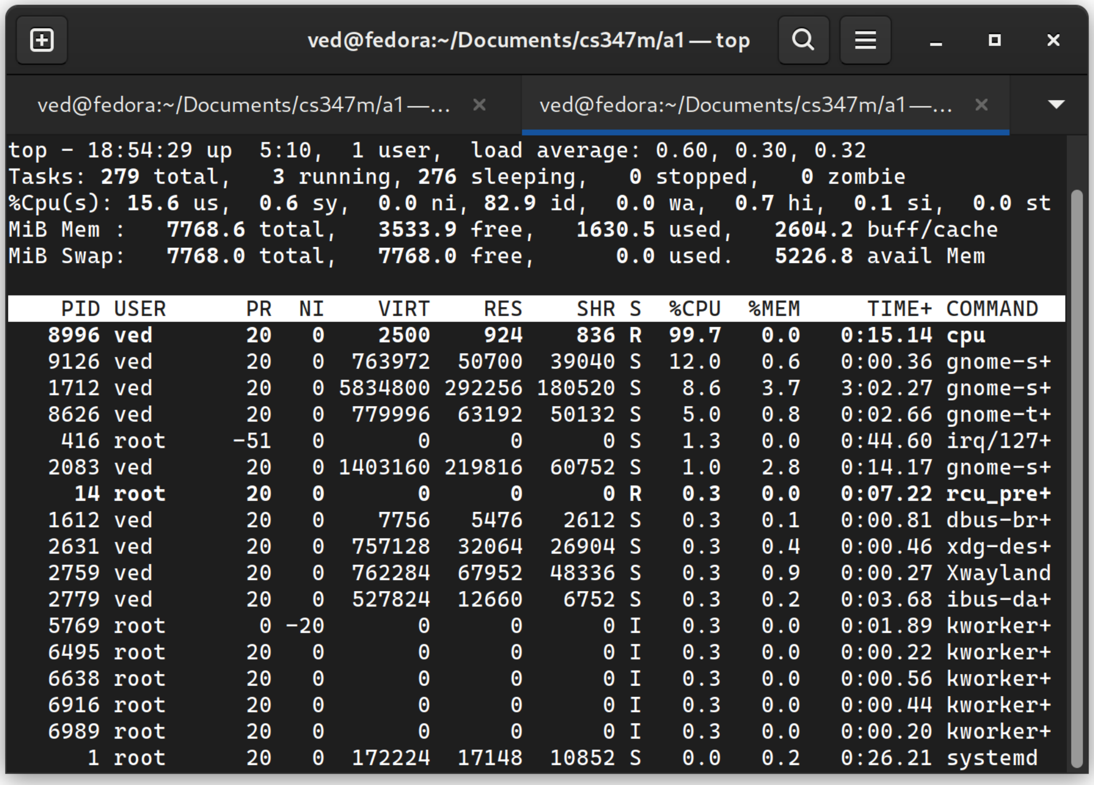


Figure 1: Output generated by `top`

(a) The PID is 8996.

(b) The program used 99.7% of the CPU and 0.0% memory.

(c) In the "S" column of the output by `top`, we can see the state of the process. For `cpu`, this column has the value "R," which means that this process is either in the running or runnable state.

# Question 3

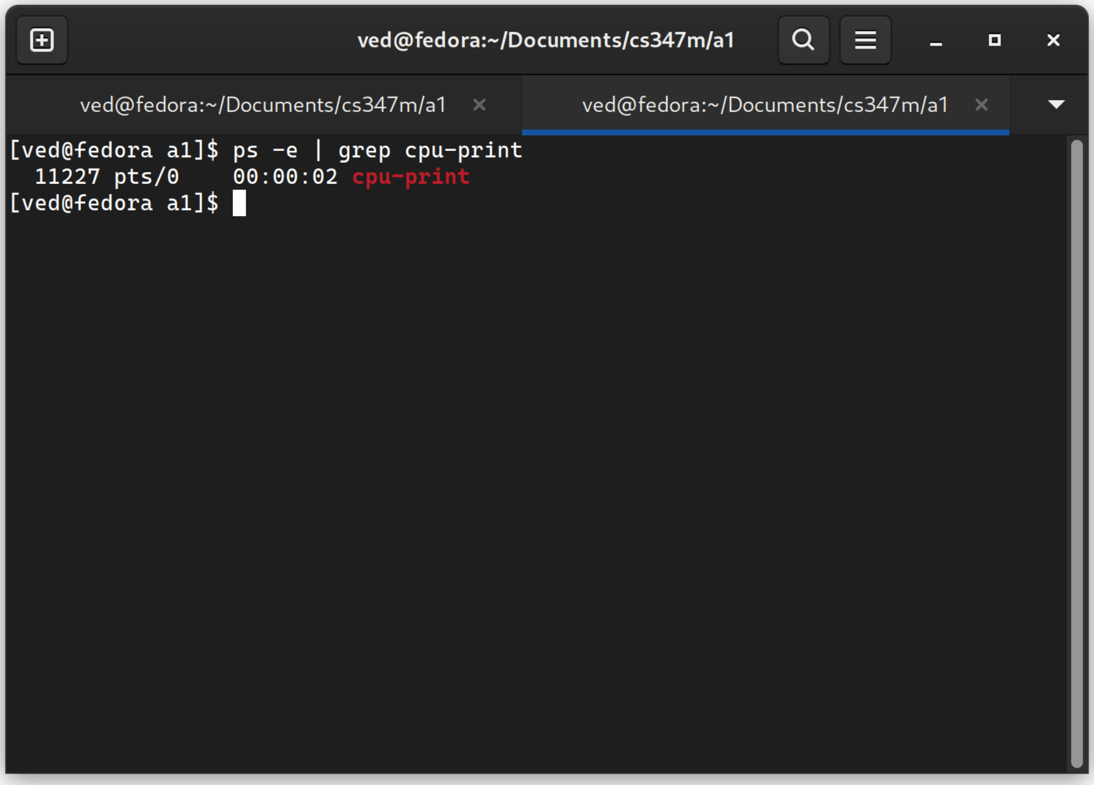


Figure 2: Output generated by `ps`

(a) The PID is 11227.

(b) We can use `pstree` to get the parents of the `cpu-print` along with the PIDs in the following way. Note that the PID has changed because I had stopped `cpu-print` and started it again.

```
[ved@fedora a1]$ ps -e | grep cpu-print
  12616 pts/0    00:00:04 cpu-print
[ved@fedora a1]$ pstree -p -s 12616
systemd(1)
   `---systemd(1569)
      `---gnome-terminal-(8626)
         `---bash(8644)
            `---cpu-print(12616)
```

(c) We have the following information from `/proc`.

```
[ved@fedora ~]$ ./cpu-print > /tmp/tmp.txt &
[1] 18341
[ved@fedora ~]$ ls /proc/18341/fd
0  1  2
[ved@fedora ~]$ file /proc/18341/fd/*
/proc/18341/fd/0: symbolic link to /dev/pts/0
/proc/18341/fd/1: symbolic link to /tmp/tmp.txt
/proc/18341/fd/2: symbolic link to /dev/pts/0
[ved@fedora ~]$ tty
/dev/pts/0
```

Here we see that `/proc/18341/fd/0` and `/proc/18341/fd/2` both point to `/dev/pts/0`. That is, `stdin` and `stderr` for the process is the terminal on which the command was run. But `stdout` is `/tmp/tmp.txt`, the file which we have specified while running the command. Thus the shell implements redirection by changing the standard IO locations for the process.

# Question 4

(a) The exact order in which the child processes are called is not deterministic since it is up to the scheduler to decide which process to run just after `fork()`. In my run, I got the following output which has 15 lines.

```
[ved@fedora a1]$ ./fork4
child 0
child 1
child 2
child 3
child 1
child 2
child 2
child 2
child 3
child 3
child 3
child 3
child 3
child 3
child 3
```

Although we cannot predict the output, we can say for sure that the output will have exactly 15 lines in every run. This can be found by rewriting the program in the way as shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(int argc, void *argv) {
  // only one process is running

  int i = 0;
  int ret = fork();
  // two processes are running now
  if (ret == 0) {
    printf("child %d\n", i);
  }

  i = 1;
  int ret = fork();
  // each of the two processes forks, now there are four
   processes
  if (ret == 0) {
    printf("child %d\n", i);
  }

  i = 2;
  int ret = fork();
  // now 4 * 2 = 8 processes
  if (ret == 0) {
    printf("child %d\n", i);
  }
```

```
    i = 3;
    int ret = fork();
    // finally 8 * 2 = 16 processes
    if (ret == 0) {
      printf("child %d\n", i);
    }
  }
```

There are 16 processes, of which one is the ultimate parent. Since only child prints to the console, there are $16 - 1 = 15$ lines of output. Moreover "child 0" will be printed once, but "child 1" will be printed twice. This is because there are two child processes which will have $i = 1$: the first child and first child's child. By induction, "child 2" and "child 3" will be printed four and eight times respectively.

(b) The program with suitable `wait()` statements is given below.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
  for(int i = 0; i < 4; i++) {
    int ret = fork();
    if (ret < 0) {
      fprintf(stderr, "fork failed!\n");
      exit(EXIT_FAILURE);
    } else if (ret == 0) {
      printf("child %d with pid %d\n", i, (int) getpid());
    } else {
      wait(NULL);
      printf("parent: reaped child %d\n", ret);
    }
  }

  return 0;
}
```

By having a **wait()** in the parent after every **fork()**, it is ensured that all of child, grandchild, great-grandchild etc. processes are reaped. The output after the above change is as follows.

```
[ved@fedora a1]$ ./fork4wait
child 0 with pid 15354
child 1 with pid 15355
child 2 with pid 15356
child 3 with pid 15357
parent: reaped child 15357
parent: reaped child 15356
child 3 with pid 15358
parent: reaped child 15358
parent: reaped child 15355
child 2 with pid 15359
child 3 with pid 15360
```

```
parent: reaped child 15360
parent: reaped child 15359
child 3 with pid 15361
parent: reaped child 15361
parent: reaped child 15354
child 1 with pid 15362
child 2 with pid 15363
child 3 with pid 15364
parent: reaped child 15364
parent: reaped child 15363
child 3 with pid 15365
parent: reaped child 15365
parent: reaped child 15362
child 2 with pid 15366
child 3 with pid 15367
parent: reaped child 15367
parent: reaped child 15366
child 3 with pid 15368
parent: reaped child 15368
```