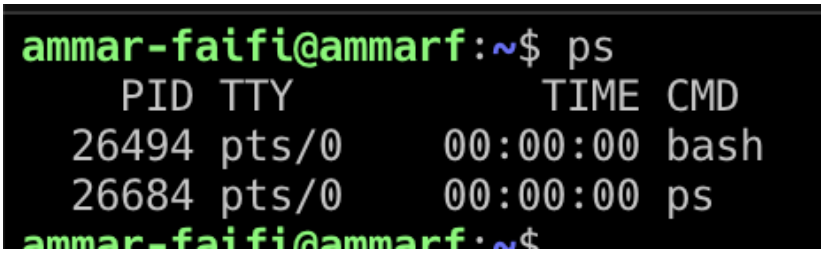
Homework 2 - Operating Systems (ICS431)

Alfaifi, Ammar

1 Commands

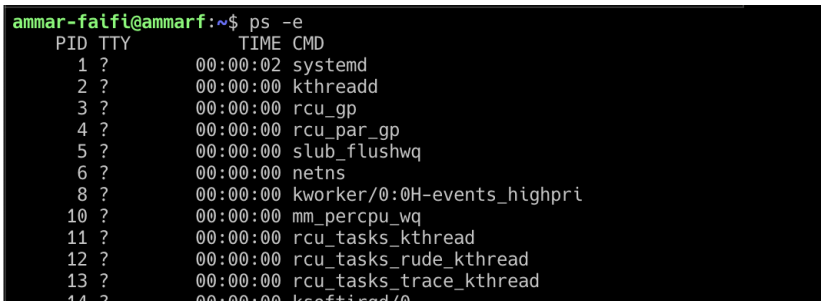
1.1 `ps`

This shows processes status. without any options, it will show you a list of processes running in your current terminal session.



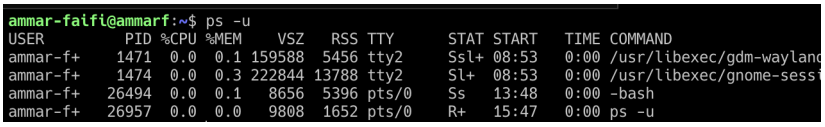
1.2 `ps -e`

show all processes, not just those related to the current terminal session



1.3 `ps -u`

will show only the processes started by the current user.



1.4 `ps -el`

command will show the complete list of processes in a long format. It will include the process ID, parent process ID, user ID, group ID, virtual memory size, resident set size, CPU usage, start time, terminal, and command.

```
ammar-faifi@ammarf:~$ ps -el
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       0  0  80   0 - 42097 -      ?        00:00:02 systemd
1 S     0       2       0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 I     0       3       2  0  60 -20 -     0 -      ?        00:00:00 rcu_gp
1 I     0       4       2  0  60 -20 -     0 -      ?        00:00:00 rcu_par_gp
1 I     0       5       2  0  60 -20 -     0 -      ?        00:00:00 slub_flushwq
1 I     0       6       2  0  60 -20 -     0 -      ?        00:00:00 netns
1 I     0       8       2  0  60 -20 -     0 -      ?        00:00:00 kworker/0:0H-events_highpri
1 I     0      10       2  0  60 -20 -     0 -      ?        00:00:00 mm_percpu_wq
1 I     0      11       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_kthread
1 I     0      12       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_rude_kthread
1 I     0      13       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks_trace_kthread
1 S     0      14       2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd/0
1 I     0      15       2  0  80   0 -     0 -      ?        00:00:07 rcu_preempt
```

2 Zombie Processes

This code is to simulate a zombie process.

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

int main() {
  pid_t pid;
  pid = fork();

  if (pid < 0) {
    fprintf(stderr, "Fork failed.\n");
    exit(1);
  } else if (pid == 0) {
    printf("This is the child process.\n");
    exit(0);
  } else {
    printf("This is the parent process. Child's PID %d\n", pid);
    sleep(10);
    waitpid(pid, NULL, 0); // wait for child process to exit
    printf("Parent process exiting.\n");
    exit(0);
  }

  return 0;
}
```

From the follwing screenshots, the process with PID of 29296 is the `bash` itself. The running C program has PID of 33017, its parent is the `bash`. When `fork()` is called, a duplicate process from the parent (PID:33017) is created with PID of 33020. This child's PID is returned in the parent process, as seen in the first line in the second figure. When the child is finished but the parent is not, the child's process (33020) becomes a zombie process (see the status `Z`).

```
ammar-faifi@ammarf:~/Documents$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   29296   29295  0  80   0 -  2262 do_wai pts/0    00:00:00 bash
0 R  1000   32943   29296  0  80   0 -  2452 -      pts/0    00:00:00 ps
ammar-faifi@ammarf:~/Documents$ ./a.out &
[1] 33017
This is the parent process. Child's PID 33020
This is the child process.
ammar-faifi@ammarf:~/Documents$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   29296   29295  0  80   0 -  2262 do_wai pts/0    00:00:00 bash
0 S  1000   33017   29296  0  80   0 -   547 hrtime pts/0    00:00:00 a.out
1 Z  1000   33020   33017  0  80   0 -     0 -      pts/0    00:00:00 a.out <defunct>
0 R  1000   33092   29296  0  80   0 -  2452 -      pts/0    00:00:00 ps
ammar-faifi@ammarf:~/Documents$ Parent process exiting.
ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   29296   29295  0  80   0 -  2262 do_wai pts/0    00:00:00 bash
0 R  1000   33166   29296  0  80   0 -  2452 -      pts/0    00:00:00 ps
[1]+  Done                    ./a.out
```

3 Multithreaded Program

The codes in C is

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *print_primes(void *arg) {
  int num = *(int *)arg;
  int i, j, flag;
  for (i = 2; i <= num; i++) {
    flag = 1;
    for (j = 2; j <= i / 2; j++) {
      if (i % j == 0) {
        flag = 0;
        break;
      }
    }
    if (flag == 1) {
      printf("%d ", i);
    }
  }
  printf("\n");
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  int num = atoi(argv[1]);

  pthread_t tid;
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  pthread_create(&tid, &attr, print_primes, &num);
  pthread_join(tid, NULL);

  return 0;
}
```

In the following screenshot, I ran the program with an input value of 100. Next I ran it with very large number so I can watch its thread using `ps -lfL` command. I use `>` to pipe the `stdout` to the virtual device `/dev/null` to discard any output, then I use `&` to run it in background.

From the output of `ps`, there are to processes with same PID which indicated it's a multithreaded process. We see also the value of the thread ID (LWP). Also the CPU utilization (C) is large in the thread with value of 93.

```
ammar-faifi@ammarf:~/Documents$ ./prime 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
ammar-faifi@ammarf:~/Documents$ ./prime 100000000 > /dev/null &
[1] 38065
ammar-faifi@ammarf:~/Documents$ ps -lfL
F S UID          PID    PPID     LWP  C NLWP PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S ammar-f+   29296   29295   29296  0    1  80   0 -  2262 do_wai 16:08 pts/0    00:00:00 -bash
0 S ammar-f+   38065   29296   38065  0    2  80   0 - 18995 futex_ 18:55 pts/0    00:00:00 ./prime 100000000
1 R ammar-f+   38065   29296   38067 93    2  80   0 - 18995 -      18:55 pts/0    00:00:02 ./prime 100000000
0 R ammar-f+   38140   29296   38140  0    1  80   0 -  2452 -      18:55 pts/0    00:00:00 ps -lfL
ammar-faifi@ammarf:~/Documents$ kill %1
[1]+  Terminated              ./prime 100000000 > /dev/null
ammar-faifi@ammarf:~/Documents$ ps
    PID TTY          TIME CMD
  29296 pts/0    00:00:00 bash
  38287 pts/0    00:00:00 ps
```

4 Proc files

1. By the commnad `cat cpuinfo`, it shows 4 CPU cores in my machine.

```
ammar-faifi@ammarf:/proc$ cat cpuinfo
processor       : 0
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3 a
simddp sha512 asimdfhm dit uscat ilrcpc flagm sb paca pacg dcpodp flagm2 frint
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0

processor       : 1
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3 a
simddp sha512 asimdfhm dit uscat ilrcpc flagm sb paca pacg dcpodp flagm2 frint
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0

processor       : 2
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3 a
simddp sha512 asimdfhm dit uscat ilrcpc flagm sb paca pacg dcpodp flagm2 frint
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0

processor       : 3
BogoMIPS        : 48.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm jscvt fcma lrcpc dcpop sha3 a
simddp sha512 asimdfhm dit uscat ilrcpc flagm sb paca pacg dcpodp flagm2 frint
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x000
CPU revision    : 0
```

2. From `cat meminfo`, I have 4002372 kB (∼ 4 GB), this is true because I'm running VM in my Mac and I specify 4 GB to the Ubuntu VM. And available of 287492 kB (287.49 MB).

```
ammar-faifi@ammarf:/proc$ cat meminfo
MemTotal:        4002372 kB
MemFree:          287492 kB
MemAvailable:    3057608 kB
Buffers:          163580 kB
Cached:          2526304 kB
SwapCached:           92 kB
Active:           995716 kB
Inactive:        2130972 kB
Active(anon):       2208 kB
Inactive(anon):   493472 kB
Active(file):     993508 kB
Inactive(file):  1637500 kB
Unevictable:       61436 kB
Mlocked:           26192 kB
SwapTotal:       2077692 kB
SwapFree:        2076908 kB
Zswap:                 0 kB
Zswapped:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:        498220 kB
Mapped:           232952 kB
Shmem:             58164 kB
KReclaimable:     322592 kB
Slab:             435388 kB
SReclaimable:     322592 kB
SUnreclaim:       112796 kB
KernelStack:        7232 kB
PageTables:        14208 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     4078876 kB
Committed_AS:    3603988 kB
VmallocTotal:   133143592960 kB
VmallocUsed:       27568 kB
VmallocChunk:          0 kB
Percpu:             3472 kB
HardwareCorrupted:     0 kB
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
ShmemPmdMapped:        0 kB
FileHugePages:         0 kB
FilePmdMapped:         0 kB
CmaTotal:          32768 kB
CmaFree:            2144 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
```

3. From running `cat stat`, then from the line field `ctxt`, It had 5417758 context switches.

```
ammar-faifi@ammarf:/proc$ cat stat
cpu  67149 686 25138 15467377 3354 0 190 0 0 0
cpu0 15037 144 6206 3868406 740 0 72 0 0 0
cpu1 16376 162 6346 3867500 794 0 19 0 0 0
cpu2 16404 180 6194 3867466 854 0 27 0 0 0
cpu3 19332 199 6390 3864004 965 0 71 0 0 0
intr 3904624 0 76320 993156 0 0 0 0 0 0 0 2544663 0 0 0
0 0 0 2743 62360 0 0 0 0 0 27081 34672 30004 33469 0 0 0
ctxt 5417758
btime 1679129575
processes 42381
procs_running 1
procs_blocked 0
softirq 3709491 2357 798097 5 74961 4115 0 63112 1266475
```

4. From the same output I see that my system had forked 42381 processes.

5 Shell Program in C

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

#define MAX_COMMAND_LENGTH 100
#define MAX_ARGUMENTS 10

int main() {
  char command[MAX_COMMAND_LENGTH];
  char *arguments[MAX_ARGUMENTS];
  pid_t pid;
  int status;
  printf("Enter commands (type 'exit' to quit): \n");
  while (1) {
    printf("201855360> ");
    fgets(command, MAX_COMMAND_LENGTH, stdin);
    // Remove newline character
    command[strlen(command) - 1] = '\0';
    if (strcmp(command, "exit") == 0) {
      printf("Exiting shell program...\n");
      break;
    }
    // Parse command-line arguments
    char *token;
    int i = 0;
    token = strtok(command, " ");
    while (token != NULL) {
      arguments[i++] = token;
      token = strtok(NULL, " ");
    }
    arguments[i] = NULL; // Set last element to NULL for execvp
    pid = fork();
    if (pid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
    } else if (pid == 0) {
      // Child process
      if (execvp(arguments[0], arguments) == -1) {
        perror("execvp");
        exit(EXIT_FAILURE);
      }
    } else {
      // Parent process
      if (waitpid(pid, &status, 0) == -1) {
        perror("waitpid");
        exit(EXIT_FAILURE);
      }
      printf("Terminated: ");
      if (WIFEXITED(status)) {
        printf("Normally\n");
        printf("Exit status: %d\n", WEXITSTATUS(status));
      } else if (WIFSIGNALED(status)) {
        printf("Due to signal\n");
        printf("Signal number: %d\n", WTERMSIG(status));
      }
    }
  }
  return 0;
}
```

Images for running different commands

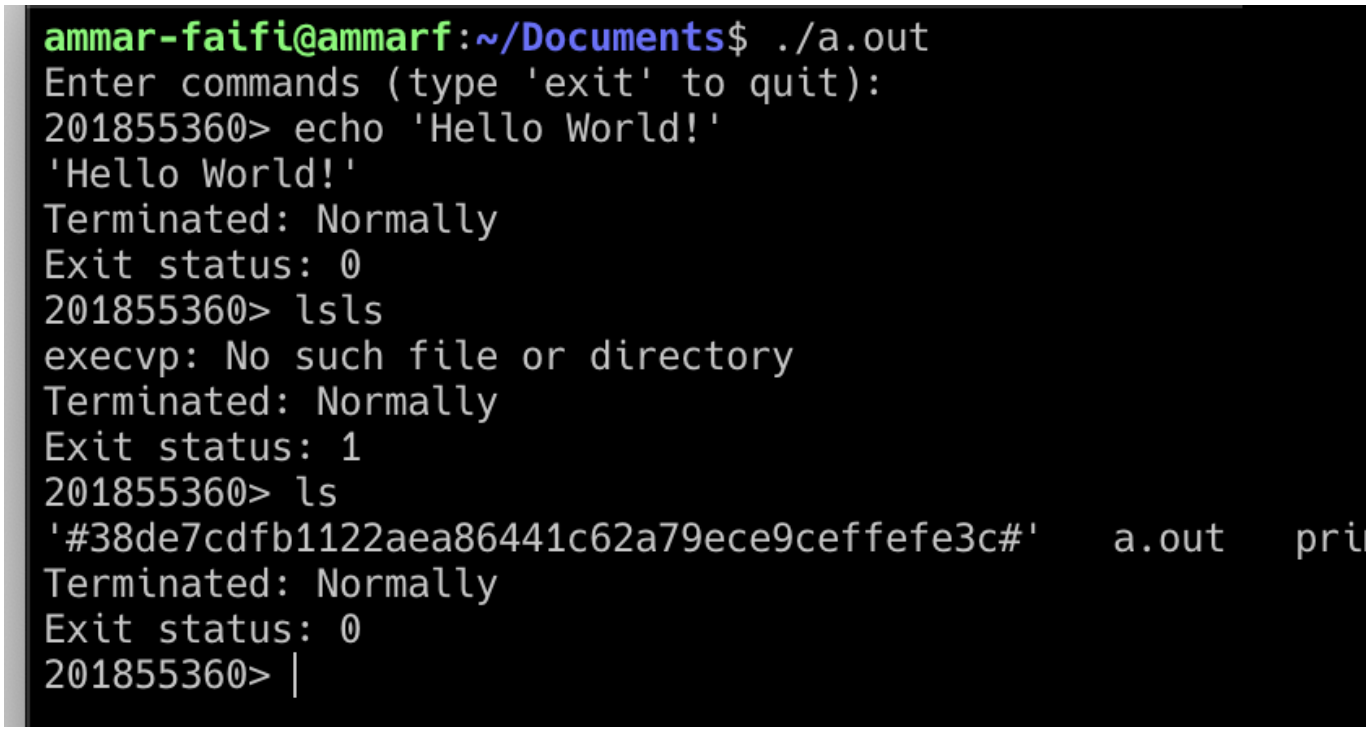


Figure 1: A screenshot for the ran program.

6 N Threads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define N 100
#define RANGE 10

int A[N], B[N], C[N];

void *sum(void *arg) {
  int i = *(int *)arg;
  C[i] = A[i] + B[i];
  return NULL;
}

int main() {
  srandom(time(NULL)); // randomize seed

  // initialize A and B
  for (int i = 0; i < N; i++) {
    A[i] = random() % RANGE;
    B[i] = random() % RANGE;
  }

  // create N threads to compute C[i] = A[i] + B[i]
  pthread_t threads[N];
  for (int i = 0; i < N; i++) {
    pthread_create(&threads[i], NULL, sum, &i);
  }

  // wait for all threads to finish
  for (int i = 0; i < N; i++) {
    pthread_join(threads[i], NULL);
  }

  // print A, B, and C
  printf("A = [");
  for (int i = 0; i < N; i++) {
    printf("%d ", A[i]);
  }
  printf("]\n");

  printf("B = [");
  for (int i = 0; i < N; i++) {
    printf("%d ", B[i]);
  }
  printf("]\n");

  printf("C = [");
  for (int i = 0; i < N; i++) {
    printf("%d ", C[i]);
  }
  printf("]\n");

  return 0;
}
```

The following figure shows the output of executing the program.

```
ammar-faifi@ammarf:~/Documents$ ./threads.o
A = [6 5 5 0 4 7 5 6 4 1 9 5 3 5 1 8 2 9 5 0 8 2 9 5 0 0 1 1 3 1 1 5 7 3 6 5 2 2 1 5 5 2 3 5 9 3 1 8 5 4 9 2 2 3 4 6 9 2 2 2 7
 8 8 9 1 1 7 4 0 4 1 7 1 5 0 8 9 8 4 5 2 0 2 1 7 0 8 3 1 6 0 9 5 8 6 2 3 1 1 2 ]
B = [5 8 9 0 7 9 2 9 9 5 3 9 5 4 5 9 5 8 1 1 7 4 9 3 6 9 6 6 6 4 1 5 4 4 3 6 0 1 7 1 7 8 3 7 8 2 1 8 3 3 2 1 4 3 0 1 8 3 9 3 7
 8 6 6 3 2 3 9 9 6 0 0 9 5 9 7 8 7 7 8 7 0 6 4 6 0 0 1 8 2 7 9 9 2 4 1 4 7 4 0 ]
C = [11 13 14 0 11 16 7 15 13 6 12 14 8 9 6 17 7 17 6 1 15 6 18 8 6 9 7 7 9 5 2 10 11 7 9 11 2 3 8 6 12 10 6 12 17 5 2 16 8 7
11 3 6 6 4 7 17 5 11 5 14 16 14 15 4 3 10 13 9 10 1 7 10 10 9 15 17 15 11 13 9 0 8 5 13 0 8 4 9 8 7 18 14 10 10 3 7 8 5 2 ]
```

Figure 2: A screenshot for the ran program.

7 Theoretical Part

1. When a kernel context-switches between processes, it saves the current process state, including program counter, registers, and stack pointer, into the current process's PCB. It then loads the saved state of the next process to be executed from its PCB into the CPU's registers and program counter. The kernel also updates the memory management unit to switch the virtual memory mappings to the next process's memory space.

2. Ordinary pipes are more suitable in situations where a simple communication channel is required between two processes executing on the same machine. For example, a parent process might create a child process and want to pass data to the child's standard input. Named pipes are more suitable in situations where multiple processes need to communicate with each other, possibly across different machines. For example, in a distributed system, several processes might need to communicate over a network using a named pipe.

3. Yes, it is possible to have concurrency without parallelism. Concurrency refers to the ability of multiple tasks to be executed simultaneously, while parallelism refers to the actual simultaneous execution of those tasks on multiple CPUs or cores. For example, a single-core CPU can execute multiple threads in a concurrent manner, but they will not execute in parallel.

4. When a thread is created, it uses fewer resources than when a process is created because it shares the same memory space as the parent process. The resources used include a program counter, a stack, and registers. When a process is created, it requires its own memory space, including a separate virtual address space and memory allocation for code, data, and stack.

5. (a) CPU utilization and response time can conflict in situations where a process with a high CPU utilization is causing other processes to wait for CPU time, resulting in longer response times.

   (b) I/O device utilization and CPU utilization can conflict in situations where a process with high I/O utilization is blocking the CPU from executing other processes, resulting in reduced CPU utilization.

6. (a) First-Come-First-Serve (FCFS) scheduling algorithm discriminates against short processes because long processes can monopolize the CPU, causing short processes to wait longer.

   (b) Round-Robin (RR) scheduling algorithm discriminates in favour of short processes because each process is given a time slice, and short processes can complete their work within a single time slice.

   (c) Multilevel feedback queues scheduling algorithm discriminates in favour of short processes because it assigns shorter time slices to processes with higher priority.