Unit 1 – Introduction

The following sections show the implementation of small C programs that illustrate the main four features of an operating system: CPU virtualization, memory virtualization, concurrency, and persistence. These programs contain their respective main source file, and use a common source file `common.c` with auxiliary functions declared in header file `common.h`.

Auxiliary functions

Create a new folder named intro using command `mkdir`, and place files `common.c` and `common.h` in it, with the following content. These files contain functions `getTime`, which returns the current time in number of seconds since 1970, and `spin`, which makes the program stay in a loop until the given amount of seconds has elapsed.

File `common.h`

```
#ifndef COMMON_H
#define COMMON_H

double getTime();
void spin(int how_long);

#endif
```

File `common.c`

```
#include <stdlib.h>
#include <sys/time.h>
#include <sys/stat.h>

#include "common.h"


double getTime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double) t.tv_sec + (double) t.tv_usec / 1e6;
}


void spin(int how_long)
{
    double t = getTime();
    while ((getTime() - t) < (double) how_long);
}
```

CPU virtualization

In the same directory as the previous files, add file `cpu.c` with the code shown below. This program takes one command-line argument and repeatedly prints it forever, waiting one second between iterations.

File `cpu.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int main(int argc, char *argv[])
{
    // Check valid arguments
    if (argc != 2)
    {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }

    // Print first argument forever
    while (1)
    {
        printf("%s\n", argv[1]);
        spin(1);
    }

    // Unreachable
    return 0;
}
```

You can compile this program using command `gcc cpu.c common.c -o cpu`. This command compiles the code in `cpu.c` and `common.c`, and produces output file `cpu`, an executable file that can be run with command `./cpu`, followed by the required string argument. Run the program and observe the output it generates. You can stop its execution by pressing `Control+C`.

Next, run the program in the background by adding the `&` symbol at the end of the command line:

```
prompt> ./cpu hello &
[1] 4379
```

By running the program in the background, you regain control over the shell, which allows you to continue typing commands while the program continues its execution. The number that is printed right after pressing `Enter` represents the *pid* (process identifier) of the new process running executable file `cpu`. You can use this number to send a signal to the child process to terminate its execution. Specifically, we can send signal code 9 (`SIGKILL`) with the following command:

```
prompt> kill -9 4379
[1]+  Killed                  ./cpu hello
```

Now run four instances of the same program in four different background processes, using the following command:

```
prompt> ./cpu a & ./cpu b & ./cpu c & ./cpu d &
```

You can easily send a `SIGKILL` signal to all these processes in the background by running command `killall cpu`. If you have a machine with four processor cores, each process will probably be executed by a different core. But even if you have less cores or processors than the number of running processes, you will still observe the same output. This effect is called *CPU virtualization*, and is one of the main purposes of an OS.

Memory virtualization

Write the following program and store it in file `mem.c`. This program reads an initial integer value from the command line, assigns it to a global variable named value, and increments it forever, waiting one second between iterations. You can compile this program with command `gcc mem.c common.c -o mem`.

File `mem.c`

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "common.h"

int value;

int main(int argc, char **argv)
{
    // Check syntax
    if (argc != 2)
    {
        fprintf(stderr, "usage: mem <value>\n");
        exit(1);
    }

    // Make p point to global variable and assign initial value
    int *p = &value;
    *p = atoi(argv[1]);
    printf("(pid:%d) address of p: %p, value of p: %d\n",
             (int) getpid(), p, *p);


    // Increment content of p forever
    while (1)
    {
        spin(1);
        *p = *p + 1;
        printf("(pid:%d) value of p: %d\n", getpid(), *p);
    }

    // Unreachable
    return 0;
}
```

Now run two instances of this program in two child processes using the following command:

```
prompt> ./mem 10 & ./mem 20 &
```

You can observe that the memory address used to store the global variable value is identical for both processes. However, the fact that both processes increment independent values proves that they are actually accessing different physical memory locations. This is due to a second major role of an OS, referred to as *memory virtualization*.

Concurrency

The following program spawns two child threads, each of which enters a loop running for the number of iterations specified in the first command-line argument. In each iteration, each thread increments a global counter by one. At the end of the execution, the final value of the counter is expected to be twice the number provided in the argument. You can compile this program with `gcc threads.c -o threads -lpthread`.

File `threads.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

// Global counter
volatile int counter = 0;

// Number of iterations
int loops;

// Thread function
void *worker(void *arg)
{
     int i;
     for (i = 0; i < loops; i++)
          counter++;
}


// Main program
int main(int argc, char **argv)
{
     // Check syntax
     if (argc != 2)
     {
          fprintf(stderr, "usage: threads <loops>\n");
          exit(1);
     }

     // Read number of iterations from argument
     loops = atoi(argv[1]);

     // Show initial value for counter
     printf("Initial value: %d\n", counter);

     // Spawn two threads
     pthread_t p1, p2;
     pthread_create(&p1, NULL, worker, NULL);
     pthread_create(&p2, NULL, worker, NULL);

     // Wait for threads to finish
     pthread_join(p1, NULL);
     pthread_join(p2, NULL);

     // Show final value for counter
     printf("Final value: %d\n", counter);
```

```
    return 0;
}
```

Run the program several times, increasing the number of iterations for the loops, as given in the first argument. After a certain number of iterations, you will observe strange effects:

```
prompt> ./threads 100000
Initial value: 0
Final value: 200000

prompt> ./threads 1000000
Initial value: 0
Final value: 2000000

prompt> ./threads 10000000
Initial value: 0
Final value: 15095029              <-- What??

prompt> ./threads 10000000
Initial value: 0
Final value: 17488257              <-- What????
```

This strange, non-deterministic result is a symptom of a race condition, typically occurring when concurrency is not dealt with properly. The OS provides a set of tools to manage concurrency, and solve this problem.

Persistence

Permanent storage is abstracted by an operating system using a standard interface that involves file names, file descriptors, and input/output operations. The program below writes string `Hello world` into a file named `/tmp/file`, which will be available after the program finishes execution, and even after the system is shut down and restarted. You can compile this program with command `gcc io.c -o io`.

File `io.c`

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>
#include <string.h>

int main(int argc, char **argv)
{
    // Open a file named '/tmp/file'
    int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, 0660);

    // Create buffer containing string 'Hello world'
    char *buffer = "hello world\n";

    // Write it into a file
    int count = write(fd, buffer, strlen(buffer));
    printf("%d bytes written\n", count);

    // Flush file state and close it
    fsync(fd);
    close(fd);
}
```

When the program finishes, you can check that file `/tmp/file` actually contains the printed string:

```
prompt> cat /tmp/file
hello world
```