Student Name: Shashank P
Student ID: 200010048

CS314 Operating Systems Lab
Lab 2

Part 1

In this part, we are supposed to write a C program to print **Hello World** but each character has to be printed by a different process. This can be achieved using the system call **fork**. On execution of my code the following steps occur:

- The variable hello_world is initialized to an array of characters.
- Until the end of the string we run a loop
- In a single iteration, it prints a character and increments the counter.
- It also prints the current Process ID.
- Sleep is called for a random number of seconds.
- Then **fork** is called. The child continues the iterations while the parent breaks out.

The flow, code and output are shown below.

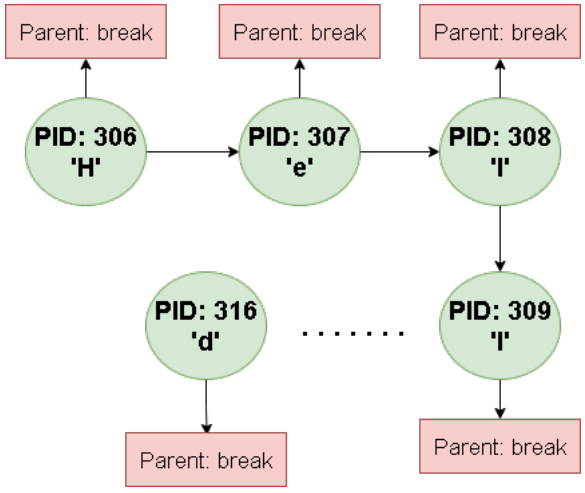

Figure 1: Part-1: Flow

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

int main()
{
    int i = 0;
    char hello[] = "Hello World";
    while(hello[i] != '\0'){
        printf("character: %c, PID: %d\n", hello[i++], getpid());
        sleep(rand()%4 + 1);
        int child_pid = fork();
        if(child_pid>0) break;
    }
    return 0;
}
```

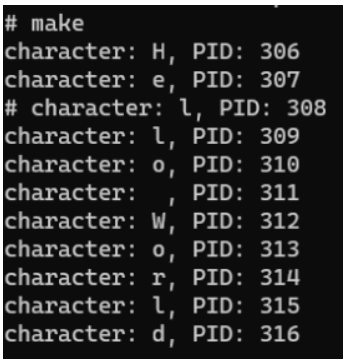

Figure 2: Part-1: Output

Part 2

Part 2 was to create three object files which could be recursively called to operate on a value and produce output. The files were **twice.c**, **half.c** and **square.c** which perform their respective operations. These operations had to also be done in a single process. Therefore I used the **execvp** system call.

- Execution can be started using any of the three operations.
- The program reads all the command line arguments.
- It performs the self-defined operation on the value present in the argument.
- We then perform the **execvp** system call.
- The arguments include all the filenames except the first one, which has already been performed.
- The new value after the operation is also passed.

The code for **twice** operation and output is shown below:

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

int main(int argc, char* argv[]){
    if(argc >= 2){
        int value = atoi(argv[argc-1]);
        value *= 2;
        printf("Twice: Current process id: %d, Current result: %d\n", getpid(), value);
        char val[12];
        sprintf(val, "%d", value);
        argv[argc-1] = val;
        argv = &argv[1];
        if(argc>2) execvp(argv[0], argv);
    }
    return 0;
}
```

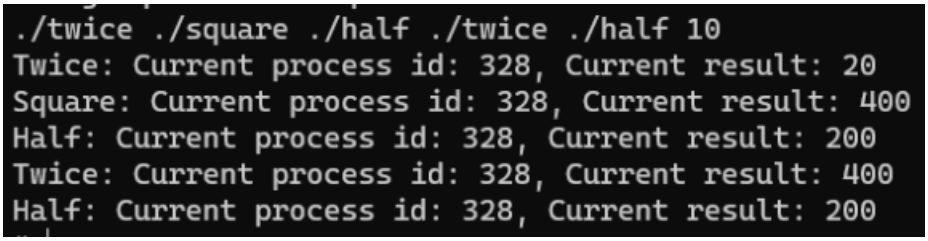

Figure 3: Part-2: Output

Part 3

In this part, we had to modify the **minix** kernel source code to print:

- 'Minix: PID *pid* created' whenever a new process is created.
- 'Minix: PID *pid* exited' whenever a process is exited.

To achieve this goal I have modified the following files:

- The **get_free_pid** function in **minix/servers/pm/utility.c** to add process creation log.
- The **cleanup** function in **minix/servers/pm/forexit.c** process exit log.

The changes performed are shown below:

```
33  pid_t get_free_pid()
34  {
35    static pid_t next_pid = INIT_PID + 1;         /* next pid to be assigned */
36    register struct mproc *rmp;                   /* check process table */
37    int t;                                        /* zero if pid still free */
38
39    /* Find a free pid for the child and put it in the table. */
40    do {
41          t = 0;
42          next_pid = (next_pid < NR_PIDS ? next_pid + 1 : INIT_PID + 1);
43          for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
44                  if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
45                          t = 1;
46                          break;
47                  }
48    } while (t);                                  /* 't' = 0 means pid free */
49 +  printf("Minix: PID %d created\n", next_pid);
50    return(next_pid);
51  }
```

Figure 4: Part-3: Create Process

```
720   static void cleanup(rmp)
721   register struct mproc *rmp;     /* tells which process is exiting */
722   {
723     /* Release the process table entry and reinitialize some field. */
724 +   printf("Minix: PID %d exited\n", rm->mp_pid);
725     rmp->mp_pid = 0;
726     rmp->mp_flags = 0;
727     rmp->mp_child_utime = 0;
```

Figure 5: Part-3: Exit Process

On building the kernel and rebooting, the updated kernel is booted up. A sample command run is shown below.

Order of creation and exit

Here I have run a simple process of compiling a C program. We can see all the processes that get created and exited. We can observe the following.

- One process can give rise to multiple processes. Ex. Process 259 creates process 260
- The parent exits only after the child exits, i.e., after cleaning up the child. Ex. Process 259 exits only after Process 260 & 261 exits.

These observations are expected behaviour in an Operating System.

```
# clang hello.c
Minix: PID 259 created
Minix: PID 260 created
Minix: PID 260 exited
Minix: PID 261 created
Minix: PID 261 exited
Minix: PID 259 exited
```

Figure 6: Part-3: Output