UPPSALA
UNIVERSITET

Report for 1DT086 and 1DT032

Lab 3: Processes and Multicores

Group 33

Nafi Uz Zaman Md Tahseen Anam Nasir Uddin Ahmed

October 4, 2019

1 Task 1: Creating Some Processes

1.1 Answer to Question 1.1

A code containing `fork(); fork(); fork(); fork();` will create 15 new processes. A program containing n number(s) of `fork();` calls will create

$$(2)^n - 1$$

new processes. When we use `fork();` a new child process is created, and then both the parent and the child processes start executing the the next line of code in the program. Considering the code below:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main() {
5      fork();
6      fork();
7      fork();
8      fork();
9      return 0;
10 }
```

- The first call to `fork();` on line 5 creates a new child process. So now we have 2 processes running.
- These 2 processes then call the `fork();` on line 6, which creates 2 new processes. Now there are 4 processes in total.
- These 4 processes then call the `fork();` on line 7, which creates 4 new processes. Now there are 8 processes in total.
- These 8 processes then call the `fork();` on line 8, which creates 8 new processes. Now there are 16 processes in total.

Therefore, there are 15 *new* processes created in the program but in total there are 16 processes including the process `main()`.

1.2 Answer to Question 1.2

In the first code snippet the parent process calls `fork();` a new child process is created and a value is stored in the variable `pid`. If the call to `fork()` returns `0` then it means the call returns to the newly created child process. But for the parent process, the value of `pid` will be the process ID of the newly created child process. So, after creating a new child process, the child process will execute everything that are inside the `if` part of the condition and the parent process will execute everything that are inside `else` part of the condition.

In both code snippets, two new child processes get created.

For the first code snippet, The newly created child process creates another child process.

In the second code snippet the newly created child process does not create another child process. This time, the parent process creates another child process.

1.3 Listing

Listing 1.1: Task 1

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

#include "led_matrix.h"

int rand_range(int min_n, int max_n){

  return rand() % (max_n + 1 - min_n) + min_n;
}

int sleep_ms(int milliseconds){

  if(usleep(1000* milliseconds) == -1){
    return -1;
  }
}

int main(){

  int pid;
  uint16_t color;
  int red_val,green_val,blue_val;

  srand(time(NULL));

  if(open_led_matrix() == -1){
    return -1;
  }

  pid = fork();

  if(pid == 0){
    red_val = rand_range(0, 255);
    green_val = rand_range(0, 255);
    blue_val = rand_range(0, 255);
    color = make_rgb565_color(red_val, green_val, blue_val);
    if(usleep(2000000) == -1){
      return -1;
    }
    set_led(0,0,color);
    printf("I'm the child! Lighting LED at (0,0)\n");
  }else{
    red_val = rand_range(0, 255);
    green_val = rand_range(0, 255);
    blue_val = rand_range(0, 255);
    color = make_rgb565_color(red_val, green_val, blue_val);
    set_led(0,1,color);
    printf("I'm the parent! Lighting LED at (0,1)\n");
  }
  wait(NULL);
  if(usleep(2000000) == -1){
    return -1;
  }
```

```
58    clear_leds();
59    if (close_led_matrix() == -1){
60      return -1;
61    }
62
63    return 0;
64
65  }
```

2 Task 2: Processes and Scheduling

We have observed the number of sockets, cores and threads in our raspberry pi using the `lscpu` command. According to our observation, we have found the following information:

- Number of socekts = 1
- Cores per socket = 4
- Threads per core = 1

So, we can see that there are 4 threads in our raspberry pi, 1 in each core. So, 4 processes can run concurrently.

2.1 Answer to Question 2.1

As we run the program, first one child process gets created and it calls the **run_child** function. So, all the 8 leds of the first row of the 8 by 8 led matrix get light up one after another. The parent process waits until the child process gets terminated. It then clears the led matrix and again creates two child processes. So, now all the 8 leds of the first two rows get light up concurrently. In each row, the leds get light up one after another. We have observed a synchronization between the leds. The first leds of the first two rows get light up simultaneously, then the second leds, then the third and thus upto the last leds. Then the two child processes get terminated, parent process clears the led matrix and again creates three child processes. Thus it continues for upto 8 child processes.

We have seen the synchronization of the leds upto 4 rows. This happens because 4 processes can run concurrently in our raspberry pi. But when we try to light up leds of more than four rows by creating more than four child processes, we have observed that the leds light up in a scattered way. It happens because all the processes are demanding more resources than the cpu can provide at an instance. So, often a fair amount of time gets allocated for all processes, sometimes higher priority processes gets more portion of the cpu than lower priority processes. Before a process finishes its execution, another process is assigned the cpu. Because of all these things, the leds light up in a scattered manner.

2.2 Answer to Question 2.2

Here, we have used the `nice()` function. `nice()` function assigns priority to a process. A niceness of -20 is highest priority and 19 is considered as the lowest priority. As a child process get created, a nice value is assigned to them by calling the `nice()` function and passing the nice value as a parameter. Then the child process calls the `run_child()` function. Because of this, the first child gets the highest priority with a nice value 0, then the second child, then third and forth and so on up to the eighth child.

We have seen the synchronization of the leds upto 4 rows just as before. But now using the nice value, we can also observe synchronization of the leds between more than 4 rows. So, what we can observe now is for example, the first led of the fifth row lights up only after the first row finishes lighting up all its leds. It ensures that `(n+1)th` led of a lower priority row does not light up before `nth` led of a higher priority row lights up. This creates a nice synchronization between the leds of different rows.

2.3 Listing

Listing 2.1: Task 2

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

#include "led_matrix.h"

uint16_t color;

int rand_range(int min_n, int max_n){

  return rand() % (max_n + 1 - min_n) + min_n;
}

void sleep_ms(int milliseconds){

  usleep(1000* milliseconds);
}

void pointless_calculation () {
  int amount_of_pointlessness = 100000000;
  int x = 0;
  for ( int i = 0; i < amount_of_pointlessness ; i++) {
    x+=i;
  }
}

void run_child (int n) {
  int row = n;
  int i;
  int red_val, green_val, blue_val;
  for (i = 0; i < 8;i++) {
    pointless_calculation();
    red_val = rand_range(0, 255);
    green_val = rand_range(0, 255);
    blue_val = rand_range(0, 255);
    color = make_rgb565_color(red_val, green_val, blue_val);
```

```
    set_led(row,i,color);
  }
}

int main(){

  int pid;
  int i,j;

  srand(time(NULL));

  if(open_led_matrix() == -1){
    return -1;
  }

  for(i=1; i <= 8; i++){
    for(j=0; j <= (i-1); j++){
      pid = fork();
      if(pid == 0){
    nice(j);
    run_child(j);
    exit(0);
      }
    }
    for(j=0;j <= (i-1); j++){
      wait(NULL);
    }
    if(usleep(1000000) == -1){
      return -1;
    }
    clear_leds();

  }

  if (close_led_matrix() == -1){
    return -1;
  }

  return 0;

}
```

3 Task 3: Write a simple shell

3.1 Listing

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

int main(){

  int pid;
  char input[100];

  while(1)
  {
    pid=-1;
    printf("myshell > ");
    scanf(" %s", input);

    if (strcmp(input,"ls")==0){
```

```
21        pid = fork();
22
23        if(pid==0){
24      if(execl("/bin/ls","ls",(char *) NULL)==-1){
25        return -1;
26      }
27
28        }
29      }
30      else if (strcmp(input,"pstree")==0){
31        pid = fork();
32
33        if(pid==0){
34      if(execl("/usr/bin/pstree","pstree",(char *) NULL)==-1){
35        return -1;
36      }
37
38        }
39      }
40      else if (strcmp(input,"htop")==0){
41        pid = fork();
42
43        if(pid==0){
44      if(execl("/usr/bin/htop","htop",(char *) NULL)==-1){
45        return -1;
46      }
47
48        }
49      }
50      else if (strcmp(input,"ifconfig")==0){
51        pid = fork();
52
53        if(pid==0){
54      if(execl("/sbin/ifconfig","ifconfig",(char *) NULL)==-1){
55        return -1;
56      }
57
58        }
59      }
60      else if (strcmp(input,"quit")==0){
61        break;
62      }
63      else printf("Command unknown");
64
65      printf("\n");
66      wait(NULL);
67
68    }
69
70    return 0;
71  }
```

UPPSALA
UNIVERSITET

Figure 1: The university seal, from around the year 1600.