Write-Up OS Assignment 1

Sakshat Sachdeva

September 2024

Introduction

The following is the Write-Up for the OS Assignment 1. The Assignment consisted of four questions.

Question 1

Write a program where the parent process has 7 children. Each child generates 4 random numbers from 1 to 100, calculates their mean, prints it in a new line and returns. The parent should wait for the children to finish and then return. $[10pts]$

Solution 1

We first use the $rand()$ function to generate random numbers.

We define the function separately and use the following expression to define numbers between 0-100:

$$(rand()\%100) + 1$$

We also created a $mean\_cal()$ function which calculates the mean of these random numbers and returns it.

In the $int\ main()$, we forked 7 times, let the 7 child processes run, and made the parent wait for these 7 child processes before exiting.

Question 2

Perform a binary search on a 16-element sorted array by passing half the array to a child process. The parent should define the array, print it and ask the user to input a target value. The parent will compare the target value to the mid element, call fork, and and the child will operate on half of the array. Similarly, the next child will operate on 1/4 of the array and so on. The child process that finds the target value should print its index in the terminal. If the target doesn't exist in the array, print -1. Each parent should wait for their child to finish before returning. [25*pts*]

Solution 2

We rewrite the binary search algorithm, but in the recursive, call, we use the fork to call the child process. The forking is first verified and then the recursive call is made under that child process. Then, the parent enters the else block and waits for the child process to wrap up.

The final output is displayed, that is either the *index* or -1.

Question 3

For this question, create a folder q3 which should contain the following files:

1. date.c: Contains a simple implementation of the date command. This program should print the current date in at least three different formats. The user can pass options/flags to specify the desired format (e.g., -u for UTC, -r for RFC 2822 format). If no flag is passed, it should print the default date and time format, similar to the linux date command.
2. cal.c: Contains a simple implementation of the cal command using Zeller's congruence. The program should take month and year as arguments and calculate the first day of the month using the formula for the Gregorian calendar. If the month and year are not provided, it should print an error message. Use Zeller's congruence to determine the first day of the given month and construct the rest of the month.
3. uptime.c: Contains a simple implementation of the uptime command. This program should print the system's uptime using the sysinfo() function. If no additional options are provided, print the uptime in hours, minutes, and seconds. Add an error message if there are any issues retrieving the uptime.
4. main.c: This program will run all the executables of the above programs as child processes using the fork-wait-exec sequence. First, create 3 child processes using fork. Then, inside the child processes, call one of the exec functions to execute the above programs. Each child will run one program.

The parent process will wait for all child processes to finish before returning.
5. Makefile: This makefile should build the first 3 programs (date, cal, uptime) before building the main program. Your files should be correctly compiled using this Makefile. The output of compiling a C file should be an executable file of the same name. For example: date.c should be compiled to date in Linux. [$40pts$]

Solution 3

We first start with the $1st$ part, that is to create $date.c$: We define three functions:

$$void\ utc()$$

$$void\ rfc()$$

$$void\ default\_time()$$

We use $< time.h >$ header's $time\_t's\ struct\ tm$ which we create through derefrencing the $GMTime$ built-in $struct$ in the header.

We repeat the same for $rfc()\ and\ default\_time()$.

We pass them in the main function depending upon the user's input being $-u$ or $-r$.

For the $2nd$ part, that is to create $cal.c$: We define two functions:

$$f\_d\ (find\ date)$$

$$f\_n\_d\ (find\ number\ of\ days)$$

.

The equation $int\ h = (q+\ (13*\ (m+1)5)+k+(k/4)+(j/4)+(5*j))\%\ 7$ helps in calculating what day lies on which date.

For the $3rd$ part, that is to create $uptime.c$: We dereference the $sysinfo\ struct$ and do basic mathematical operation to get the exact Hours, Minutes and Seconds.

For the 4th part, that is to create *main.c* which executes *date.c*, *uptime.c*, *cal.c*: We fork each individual process and execute the files using the

$$execlp()$$

function.
We then go ahead and $wait()$ for 3 processes since we know that only three processes are running.

For the 5th part, we create a 'MakeFile': We give the variable CC the value of gcc and add the programs under the variable *all*. Then, we just compile the functions one by one and then call the main file which passes the upper three compiled files into the *main* function.

Question 4

Q4. Write a c program to simulate the different CPU scheduling algorithms. Four scheduling algorithms should be implemented in the program: [25*pts*]
1. First In First Out (FIFO)
2. Shortest Job First (SJF)
3. Shortest Remaining Time First (SRTF)
4. Round Robin (RR)
The program should accept input for n processes (n¿4), where each process has: Process ID: A unique identifier for the process.
Arrival Time: The time at which the process arrives in the ready queue.
Burst Time: The total time required by the process for execution.
Time Quantum (TQ): The value of TQ should only be used in RR.
For each scheduling algorithm, the program should:
Calculate and display the order of execution of processes.
Calculate and display the scheduling policy's Average Response Time and Average Turnaround Time.

Solution 4

We have to replicate the process of FIFO, SJF, SRTF and RR.
We start by defining a struct:

```
typedef struct {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int response_time;
    int turnaround_time;
    int time_quantum;
} Process;
```

The above struct contains Process ID, Arrival Time, Burst Time, Remaining Time, Response Time, Turn Around Time and Time Quantum.

The struct took several reiterations to arrive to. Now in each individual function, we pass the process struct and the number of processes.

Each function has a *helper* function inside it, which makes the task way easier. In case of *RR*, we take a special input for the *Time Quantum*

In SJF, we first sort the jobs by there order. This allows us to take the shortest job first. Since it is non-premptive in nature, we can go ahead and set a set pattern that does not change.
In RR, the order is actually how many iterations each process took as it displays it while continuing to slice off Time Quantum from the Burst time.

In the end, we lay out the Average Response and Turnaround Time as well as the order in which the process executes.