Universidad Autónoma de Guadalajara

C programming Language

STUDENT: Erick Mathew García Sánchez

CAREER: Software Engineering and Data Mining

COURSE: Operating Systems

REGISTRATION NUMBER: 2818812

TEACHER: Agustín Villarreal

DATE: 13 – 09 – 25

A) Bookstores:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <time.h>
  #include <string.h>
  #include <sys/time.h>
```

1. Functions such as fork(), execvp(), getpid()
2. For wait() and macros related to process states
3. Time-related functions
4. String manipulation functions, not used directly here
5. For gettimeofday() functions and more accurate

B) Command definition

```
int main()
{
    char *commands[][3] =
    {
        {"ls", "-1", NULL},
        {"date", NULL, NULL},
        {"whoami", NULL, NULL},
        {"pwd", NULL, NULL},
        {"ps", "aux", NULL}

    };
```

An Array of arrays of character pointers is declared; each sub-array contains a command and its arguments to execute. The las element is always NULL, required by execvp to mark the end of arguments.

c) variables to control processes and times

```
int num_commands = 5;
pid_t child_pids[num_commands];

struct timeval start_total, end_total;
```

num_commands: Number of commands to be executed (in this case 5).

child_pids: Array to store the process identifiers (PIDs) of the children.

start_total and end_total: variables to measure the total execution time using gettimeofday().

d) Measure total time at start and display information

```
gettimeofday(&start_total, NULL);

printf("--- Process Monitor ---\n");
printf("Parent PID: %d\n\n", getpid());
```

gettimeofday obtains the current time with microsecond precision. A header and the parent process PID are printed.

e) create child processes with fork()

```
printf("Creating child processes...\n");

for (int i = 0; i < num_commands; i++)
{
    pid_t rc = fork();
```

A loop is created to generate a child for each command. fork() duplicates the current process, returning:

- < 0 if it fails
- 0 in the child process
- PID of the child in the parent process

f) Error handling and execution in child processes

```
if (rc < 0)
{
    fprintf(stderr, "Fork failed for command %d\n", i);
    exit(1);
}else if (rc == 0){
    printf("Child %d (PID: %d) executing: %s\n", i, getpid(), commands[i][0]);

    execvp(commands[i][0], commands[i]);

    fprintf(stderr, "Exec failed for command: %s\n", commands[i][0]);
    exit(1);
```

If fork() fails, an error is output and the program terminates. In the child process (rc == 0), an output is generated indicating that a command is being executed. The command is executed with execvp(). If execvp() fails, an error message is output and the child process terminates to prevent further execution of faulty code.

g) store child PIDs in the parent process

```
}else{
    child_pids[i] = rc;
    printf("Parent created child %d with PID: %d\n", i, rc);
}
```

in the parent, store the child's PID in the array. Print confirmation of the child's creation.

h) wait for the children to finish and obtain their status

```
for (int i = 0; i < num_commands; i++)
{
    int status;
    pid_t wc = wait(&status);

    if (wc == -1){
        fprintf(stderr, "wait failed\n");
        continue;
```

Use a loop to wait for all children to finish with wait(). wait() returns the PID of the finished child ant its status. If wait() fails (-1), an error is reported and the program continues.

i) Find the index of the finished child

```
int child_index = -1;
for (int j = 0; j < num_commands; j++)
{
    if (child_pids[j] == wc)
    {
        child_index = j;
        break;
    }
}
```

Search the child_pids array to find out which child finished (by its PID). This allows you to relate the PID to the corresponding index and command.

j) Display execution result for each child

```
if (WIFEXITED(status))
{
    printf("Child %d (PID: %d) completed with exit code: %d\n", child_index, wc, WEXITSTATUS(status));

}else{
    printf("Child %d (PID: %d) terminated abnormally\n", child_index, wc);
}
```

WIFEXITED(status): checks whether the child terminated normally.

WEXITSTATUS(status): obtains the exit code of the child.

If the child terminated abnormally, this is reported.

k) Measure total execution time and display it

```
gettimeofday(&end_total, NULL);
double total_time = (end_total.tv_sec - start_total.tv_sec) + (end_total.tv_usec - start_total.tv_usec) / 1000000.0;
printf("\n total execution time: %.3f seconds\n", total_time);

return 0;
```

The time is captured when all child processes are finished. The difference in seconds is calculated, considering seconds and microseconds. The total execution time of the parent program is printed.