ICS 2305: Systems Programming Assignment 1 (CAT 1): SCT211-0221/2018: Peter Kibuchi

Created	@October 16, 2023 5:15 AM
Last Updated	@October 16, 2023 7:55 AM
Tags	3.1 Assignments

1. **Write a C program that prints the process ID, priorities and parent ID of all programs currently in the RAM.**

   To accomplish this task, we can use the `ps` command to retrieve information about running processes. The `ps` command provides details about the processes running on your system.

   Below we use the `popen` function to execute the `ps` command and then parse its output to obtain the process ID, priority, and parent ID for all processes:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    // Open a pipe to execute the "ps" command and read its output
    FILE *pipe = popen("ps -e -o pid,pri,ppid", "r");
    if (!pipe) {
        perror("popen");
        exit(EXIT_FAILURE);
    }

    // Print the header
    printf("PID\\tPriority\\tParent PID\\n");

    // Read and print process information
    char line[256];
    while (fgets(line, sizeof(line), pipe)) {
        if (line[0] >= '0' && line[0] <= '9') {  // Check if the line starts with a d
igit (process ID)
            int pid, priority, ppid;
            if (sscanf(line, "%d %d %d", &pid, &priority, &ppid) == 3) {
                printf("%d\\t%d\\t\\t%d\\n", pid, priority, ppid);
            }
        }
    }

    // Close the pipe
    pclose(pipe);

    return 0;
}
```

   In this program:

   1. We use `popen` to execute the `ps` command and read its output.

2. We print a header to label the columns.
3. We parse each line of the `ps` output and extract the process ID (`pid`), priority (`pri`), and parent process ID (`ppid`).
4. We use the `sscanf` function to extract the values.
5. If the line starts with a digit (indicating a process entry), we print the process information.

2. **Create a Shell Script called “NYONGA”. While the Shell is open, write a program in C that kills the open shell Script.**

   To create a shell script called "NYONGA" and write a C program that kills the open shell script when executed, we can follow these steps:

   a. Create the "NYONGA" shell script (i.e., "NYONGA.sh") with the following content:

```
#!/bin/bash

# This is the NYONGA shell script.
# It will keep running as long as the shell is open.

echo "NYONGA shell script is running."

# Keep the script running indefinitely
while true; do
  sleep 1
done
```

   Make the shell script executable by running: `chmod +x NYONGA.sh`

   b. Next, create a C program (e.g., "kill_NYONGA.c") that kills the open "NYONGA" shell script when executed. Here's the C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

int main() {
    // Find the process ID (PID) of the "NYONGA" shell script
    FILE *pidfile = popen("pgrep -o -f 'NYONGA.sh'", "r");
    if (pidfile == NULL) {
        perror("popen");
        exit(1);
    }

    char pid_str[16];
    if (fgets(pid_str, sizeof(pid_str), pidfile) != NULL) {
        // Convert the PID string to an integer
        int nyonga_pid = atoi(pid_str);

        // Kill the "NYONGA" shell script
        if (nyonga_pid > 0) {
            printf("Killing NYONGA (PID: %d)\\n", nyonga_pid);
            kill(nyonga_pid, SIGTERM);
        } else {
```

```
            printf("NYONGA is not running.\\n");
        }
    } else {
        printf("NYONGA is not running.\\n");
    }

    pclose(pidfile);
    return 0;
}
```

c. Run the "NYONGA" shell script in one terminal window, and the compile & run the C program in another.

The "NYONGA" script will run indefinitely, printing "NYONGA shell script is running."

The C program will find the PID of the "NYONGA" script and send a `SIGTERM` signal to terminate it. You'll see the message "Killing NYONGA (PID: ...)" when it successfully kills the script.

3. **Write a C program that uses the fork() API to create a child process.**

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t child_pid;

    // Fork a child process
    child_pid = fork();

    if (child_pid == -1) {
        // Fork failed
        perror("fork");
        return 1;
    }

    if (child_pid == 0) {
        // This code runs in the child process
        printf("Child process: PID = %d, Parent PID = %d\\n", getpid(), getppid());
    } else {
        // This code runs in the parent process
        printf("Parent process: PID = %d, Child PID = %d\\n", getpid(), child_pid);
    }

    return 0;
}
```

In this program:

1. We include the necessary headers, including `<stdio.h>` and `<unistd.h>`.
2. We declare a variable `child_pid` of type `pid_t`, which is used to store the process ID of the child process.
3. We call the `fork()` function, which creates a new process by duplicating the existing process. The return value of `fork()` is used to differentiate between the

parent and child processes.

4. In the parent process, `fork()` returns the process ID (PID) of the newly created child process. In the child process, `fork()` returns 0.
5. We check the return value of `fork()` to determine whether we are in the parent or child process and print the appropriate information.

When we run this program, it will create a child process, and both the parent and child processes will print their respective process IDs (PID) and parent process IDs (PPID). The parent process will display the child's PID, and the child process will display the parent's PID.

4. **Describe in prose how waitpid() and wait() works. Write a C program that uses waitpid() and wait() APIs to get the termination status of a child in the parent.**

   `wait()` and `waitpid()` are system calls in Unix-like operating systems used for handling child processes. These calls are used by the parent process to wait for the termination of its child processes and retrieve their termination status. Here's how they work:

   1. `wait()`:
      - `wait()` is a simple way to wait for the termination of a child process.
      - When the parent process calls `wait()`, it will block until any of its child processes terminates.
      - It returns the PID of the terminated child process.
      - The parent can then use `WIFEXITED(status)` to check if the child terminated normally, `WEXITSTATUS(status)` to get the exit status of the child, and `WIFSIGNALED(status)` to check if the child terminated due to a signal.
   2. `waitpid()`:
      - `waitpid()` is more flexible than `wait()` because it allows the parent process to specify which child process it wants to wait for based on the PID.
      - The parent process can use `waitpid()` to wait for a specific child, all children, or any child that matches certain criteria.
      - It can also be used with options like `WNOHANG` to make it non-blocking.
      - `waitpid()` returns the PID of the terminated child process or -1 in case of an error.
      - The parent can use `WIFEXITED(status)`, `WEXITSTATUS(status)`, and `WIFSIGNALED(status)` in the same way as with `wait()` to retrieve information about the child's termination.

   Here's a C program that demonstrates how to use `waitpid()` and `wait()` to get the termination status of a child in the parent:

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

```
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t child_pid;
    int status;

    child_pid = fork(); // Create a child process

    if (child_pid < 0) {
        perror("fork");
        exit(1);
    } else if (child_pid == 0) {
        // Child process
        printf("Child process is running...\\n");
        sleep(3); // Simulate some work
        exit(42);  // Exit with a status code
    } else {
        // Parent process
        printf("Parent process is waiting for the child...\\n");

        // Using waitpid to wait for the specific child
        if (waitpid(child_pid, &status, 0) == -1) {
            perror("waitpid");
            exit(1);
        }

        if (WIFEXITED(status)) {
            printf("Child process exited with status: %d\\n", WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("Child process terminated by signal: %d\\n", WTERMSIG(status));
        }
    }

    return 0;
}
```

In this program, the parent process creates a child, waits for the child's termination using `waitpid()`, and then checks the termination status of the child process. The child process simulates some work and exits with status code 42. The parent process uses `WIFEXITED` and `WIFSIGNALED` to determine how the child process terminated.

5. **In Linux tools such as `sysstat`, `sar` and `Nmon` can be used to monitor CPU performance. We want to graph various CPU usages. We want to capture the CPU usage for few minutes and graph it. Write a C program that incorporates a simple Graphical User Interface that graphs CPU usage in real time.**

   Creating a graphical user interface (GUI) for real-time CPU usage monitoring and graphing in C is a more complex task, typically achieved using GUI libraries such as GTK, Qt, or SDL.

   Below I provide a simplified example using the GTK library. (One has to have the GTK development libraries installed on their system to compile and run the program.)

```
#include <gtk/gtk.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sysinfo.h>
```

```
// Structure to hold CPU usage data
typedef struct {
    double usage;
} CPUData;

// Function to update CPU usage data
void update_cpu_data(CPUData *cpu_data) {
    struct sysinfo info;
    if (sysinfo(&info) == 0) {
        // Calculate CPU usage as a percentage
        cpu_data->usage = 100.0 - (100.0 * info.loads[0] / info.procs);
    }
}

// Callback to draw the CPU usage graph
gboolean draw_cpu_graph(GtkWidget *widget, cairo_t *cr, gpointer data) {
    CPUData *cpu_data = (CPUData *)data;
    int width, height;
    gtk_widget_get_size_request(widget, &width, &height);

    // Draw the background
    cairo_set_source_rgb(cr, 0.9, 0.9, 0.9);
    cairo_rectangle(cr, 0, 0, width, height);
    cairo_fill(cr);

    // Draw the CPU usage graph
    cairo_set_source_rgb(cr, 0.0, 0.5, 0.0);
    cairo_set_line_width(cr, 2.0);
    cairo_move_to(cr, 0, height);
    cairo_line_to(cr, width * cpu_data->usage / 100.0, height);
    cairo_stroke(cr);

    return TRUE;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *drawing_area;
    CPUData cpu_data;

    // Initialize GTK
    gtk_init(&argc, &argv);

    // Create the main window
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "CPU Usage Monitor");
    gtk_window_set_default_size(GTK_WINDOW(window), 400, 100);

    // Create a drawing area
    drawing_area = gtk_drawing_area_new();
    gtk_container_add(GTK_CONTAINER(window), drawing_area);

    // Connect the "draw" signal to the draw_cpu_graph function
    g_signal_connect(G_OBJECT(drawing_area), "draw", G_CALLBACK(draw_cpu_graph), &cpu
_data);

    // Update the CPU usage data every second
    g_timeout_add(1000, (GSourceFunc)update_cpu_data, &cpu_data);

    // Show all widgets
    gtk_widget_show_all(window);

    // Start the GTK main loop
    gtk_main();

    return 0;
}
```

This program creates a simple GTK-based GUI window that displays a real-time graph of CPU usage as a green bar that grows horizontally. The `update_cpu_data` function calculates the CPU usage as a percentage, and the `draw_cpu_graph` function is responsible for drawing the graph. The main loop continuously updates the graph every second.

6. **Describe users defined signals with two working examples.**

   User-defined signals, also known as "custom signals," allow processes in a Unix-like operating system to communicate with each other. These signals can be used to notify a process about specific events or conditions. Custom signals are defined by their numbers, starting from `SIGUSR1` and `SIGUSR2`. Here are two succinct working examples of user-defined signals:

   Example 1: Sending and Handling SIGUSR1

   In this example, we create a C program that sends a SIGUSR1 signal from one process to another, and the receiving process handles the signal.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

// Handler function for SIGUSR1
void sigusr1_handler(int signum) {
    printf("Received SIGUSR1\\n");
}

int main() {
    // Register the SIGUSR1 signal handler
    signal(SIGUSR1, sigusr1_handler);

    printf("Send a SIGUSR1 signal to this process using 'kill -SIGUSR1 <pid>'\\n");

    // Keep the process running
    while (1) {
        sleep(1);
    }

    return 0;
}
```

   Compile the program and run it. We can send a SIGUSR1 signal to the process using the `kill` command like this:

```
# Get the PID of the process
ps aux | grep your_program_name

# Send SIGUSR1 to the process
kill -SIGUSR1 <PID>
```

   When we send the SIGUSR1 signal, the process will print "Received SIGUSR1."

   Example 2: Sending and Handling SIGUSR2

   In this example, we'll create a C program that sends a SIGUSR2 signal to itself and handles the signal.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

// Handler function for SIGUSR2
void sigusr2_handler(int signum) {
    printf("Received SIGUSR2\\n");
}

int main() {
    // Register the SIGUSR2 signal handler
    signal(SIGUSR2, sigusr2_handler);

    printf("Send a SIGUSR2 signal to this process using 'kill -SIGUSR2 <pid>'\\n");

    // Send a SIGUSR2 signal to the current process
    raise(SIGUSR2);

    return 0;
}
```

Compile the program and run it. The program sends a SIGUSR2 signal to itself using `raise(SIGUSR2)` and then handles the signal. When we run this program, it will print "Received SIGUSR2."

7. **Write a Program that creates a text file called JUJU in drive C and create a signal that deletes the file 5 seconds after the creation and reports on prompt.**

   Here's a basic outline of how you might achieve this using a C program and a shell script:

   1. Create a C program to create the file "JUJU.txt" in a specified directory (e.g., "C:/") and print a message indicating that the file has been created.

```
#include <stdio.h>

int main() {
    FILE *file = fopen("C:/JUJU.txt", "w");
    if (file) {
        fprintf(file, "This is the JUJU file.");
        fclose(file);
        printf("File 'JUJU.txt' created in C:/.\\n");
    } else {
        printf("Failed to create the file.\\n");
    }
    return 0;
}
```

   2. Compile and run the C program to create the file. This program creates the "JUJU.txt" file on the C:/ drive.
   3. Create a shell script (e.g., "delete_JUJU.sh") that uses the `at` command to schedule the deletion of the file after a specific delay. Save it in the same directory as the C program.

```
#!/bin/bash
```

```
echo "rm C:/JUJU.txt" | at now + 5 minutes
```

4. Make the shell script executable:

```
chmod +x delete_JUJU.sh
```

5. Run the shell script:

```
./delete_JUJU.sh
```

The shell script schedules the file deletion command to be executed 5 minutes after it is called. You should see a prompt that confirms the job has been scheduled.

This is a basic example, and you may need to adapt it to your specific requirements and the operating system you are using. It's important to handle file deletion carefully, especially when working with system directories like the root of the C drive.