Shenggui Jin

110984504

CSE 376 HW4 Final Project

Client-Server Unix Domain Socket Job Submission System

1. Introduction

This project is composed of 2 programs that interact each other using a special protocol via a unix domain pipe. The client is able to send to the server any bash command or program runnable by execvp(3) lib call. The server program is able to listen for client request and returns response to the client based on the "action_code" provided by the client. The server then create a different thread to handle the request and put the job in a job list while main thread is listening for more connection.

2. System and Environment

-**System**: The program was tested and successfully run on linux Ubuntu 18.04.3 LTS

-**Environment:** gcc 7.5.0 / ld 2.30

3. How to Use

By "make", you get "client" and "server" executables. Both have -h flag that tells you the usage of the program.

Server:

```
Usage: ./server -[idh] -[cjm] [optarg]
-i  Interactive mode
      Can enter commands runnable by execvp
      Special commands:
         'exit' exit the server
         'list' list all jobs
         'resource' list resource of all jobs
         'clearlist' clear exited or killed jobs
-j [Max_Job_Number]  <- an int
      Allow Max_Job_Number of jobs running at the same time
      Exited and Killed jobs are still considered running if they are not cleared
      Therefore, to add new job. Client has to clear job list with action_code 2
      The threshold of Max_Job_Number is 10. Anything above 10 will be set to 10
-m [mem_limit]  <- an int
      Specify mem_limit for each job
-c [cpu_limit]  <- an int
      Specify cpu_limit for each job
-h Print Usage
-d Print Debug Message
```

Client:

Usage: ./client -a 1 -c "command" or ./client -a [2-6] -[hd] -[j] [optarg]
-a [0-6]
0 No Action
1 Submit
2 Delete a job/jobs
3 List a job/jobs
4 Stdout of a job
5 Stderr of a job
6 Resource of a job/jobs
-c "command" (Surround with " ", e.g. "echo hello")
Must be used with -a 1
-j [job number]
For action 2, 3, 4, 5 and 6
For action 2, 3 and 6, if no job/ incorrect job number is provided,
all jobs will be printed by default.
-h Print Usage
-d Print Debug Message
Note: command example: “pwd”, “ls”, “echo hello”, “kill -19 pid”, anything runnable by execvp

4. Program Design

Handshaking stage:

The server and clients are communicating through a local pipe socket file called socket_pipe. This file will be created automatically if there is none. When server is running, it will be put into sleep mode by "accept" and wait for connection. Client at this moment can get input from command line, parse it and put the data in a struct called "request" in "message.h" and try to connect to the server. Once a connection is established, server will wake up and create a separate, detached thread to handle the client request. Once connected, the client will send the bytes of the struct request it previous stored to the server. Server then returns the message of whether it gets the packet or not back to the client. For example, if the number of jobs in the job list on the server side is full. An error message will be returned to the client. Client then closes the connection. (Client doesn't need to wait for the job to finish).

Protocol struct:

```
struct request{
    int action_code;
    int job_id;
    char command[MAX_COMMAND_LEN];
};
```

Processing stage:

If the action_code of request is not "Submit" a new job, server can return the result immediately back to client. Else, the server will parse the command in the request packet, fork a child process, setrlimit based on command line, dup2 stdout and stderr tot the previously opened pipes and execute command using execvp.

Handling Job and Return Status:

Return status are basically obtained by wait(2) family functions. All job are stored in a double linked list. Each job node has a job_id, pid, running status, return status, and fields for printing stdout, stderr and resources. Using double linked list in the way that newly added job is always inserted headnode.next and we can access the head of the list job by calling headnode.prev. This make adding and listing much faster.

Job node struct:

```
struct joblist{
    struct joblist *prev;
    int jobnumber;
    char *name;
    int pid;
    int status;
    int retval;
    char stdout[1000];
    char stderr[1000];
    struct rusage usage;
    struct joblist *next;
};
```

Handling stdout/stderr:

Since client doesn't wait for the job to finish, the server needs to store the output of stdout and stderr somewhere so client can get it next time. To do so, server will pipe(2) and dup2(2) the child process to retrieve stdout and stderr to be stored in a buffer of 1000 bytes. Each job in the joblist is associated with two such buffers, 1 for stdout, 1 for stderr. If client asks for these next time, server can send them directly to the client.

Handling Print Resources:

All paused/killed/exited jobs can output a usage struct using wait4(2) syscall. The problem is the running processes. To solve this problem, the program temporarily sends SIGSTOP to running process, obtain the usage struct and then sends SIGCONT to continue the processes.

Handling Limit Resources:

Server can limit the resources of each child processes with -c (CPU) and -m (Mem) flag when program runs. These value will be applied to all child processes using setrlimit(2) syscall.

Handling Server Side Resources Checking:

User can call interactive mode of server using flag -i to start a interactive shell. This shell is a seperate thread from the main thread that handles connection and jobs. By inputing special command "list" or "resource", user can track the resource and status of each job currently in the job list and kill/stop them with "kill -SIGNUM pid" just like terminal.

Handling Kill/Stop/Priority:

Both client and server can execvp(3) "kill"/"renice" process using like terminal.