SMASH Job Server

Overview

To create a job server, which clients may connect to and submit jobs. Jobs will be given resource limits to adhere to (maximum CPU time and maximum memory usage), as well as a priority level that they should run at. If a client's submitted job exceeds these limits, it should be terminated without prejudice.

The client and server shall communicate over a UNIX domain socket. The server shall allow for multiple clients to connect and submit jobs and commands, such that each client is only able to access data and jobs associated with themselves. The server shall provide asynchronous notifications of status updates for any of their clients' submitted jobs.

Server Program

The server program should conform to the following command line options:

`-f socketfile`: Specifies the socket file to use for the server, or `.cse376hw4.socket` if this option is not specified. `-d`: Enables debugging output. `-n maxjobs`: Maximum number of jobs the server can concurrently run, or `INT_MAX` if this option is not specified.

After parsing any command line options supplied by the user, the server install any required signal handlers (at least for `SIGINT`, `SIGTERM`, `SIGCHLD`, and `SIGUSR1`) before creating a UNIX domain socket using `socket(2)` and specifying `AF_UNIX`. The program shall then `bind(2)` to the file descriptor of the socket and `listen(2)` for up to `1024` connections.

The server shall then enter the main loop of the program, wherein it will use `pselect(2)` to check the status of all open file descriptors, using `accept(2)` to accept new connections and using the function `server_handle_client()` to handle communications from a particular client.

`server_handle_client()` This function shall be specified to handle any transmissions from a client to the server. The function shall first call `recv_pkt()` on the appropriate file descriptor, using the results returned from this call to determine how to further process the request.

The server may then use the results of `recv_pkt()` to determine the packet type received, take appropriate action (enqueue a new job, process a results request, etc.), then use `send_pkt()` to communicate the result of the client request back to the client.

Signal Handling The signal handler installed by the server shall do naught except set a flag denoting what kind of signal was received and return. The program shall only take action on these signals during the main loop, wherein it shall first mask all signals and call a function `handle_all_signals()` (which will actually perform the necessary operations based on the signals received) before once again unmasking all signals.

The following signals shall be caught within the signal handler and handled within the `handle_all_signals()` function: - `SIGINT`/`SIGTERM`: specifies that the user wishes to shut down the server. This should cleanly exit the server, freeing any necessary resources, disconnecting any connected clients, removing any files created, and ending any jobs currently running. - `SIGCHLD`: specifies that one or more of the server's child processes changed state. If the client who owns this job is currently connected, the client shall be notified of the change in job state. If a child process

has ended, the server shall attempt to start any other jobs currently not already started, until the server reaches the upper limit of maximum allowed jobs. - `SIGUSR1`: specifies that the user wishes to toggle debugging output. If debugging output is currently disabled, this signal enables it. If debugging output is currently enabled, this signal disables it.

Clients The server shall store a list of all known clients. Each client shall be identified by a unique name, and a client record shall contain at least the following information: a file descriptor associated with the client, the name of the client, a flag denoting whether the client is connected or not, and a linked list of all jobs associated with the client.

Connections Upon a new connection, the server shall keep a record of the connection within a linked list and the client associated with it. The connection structure shall be defined as containing a file descriptor, a pointer to the `client_t` representing the specific client, and a link to the next connection in the list.

The first packet received by the server from a newly connected client shall be a `LOGIN` packet with a specified username for the client. If a client with the specified username already exists and is not currenty connected, the server will log the client in and the client may continue. If a client with the specified username already exists and is currently connceted, the server shall refuse the login request and disconnect the client. If no record of a client exists with the specified username, then the server shall create and maintain a new client record for the client.

When a client disconnects, the server shall remove the corresponding entry from the connections list for the server. Note that the client structure associated with the disconnected client shall **NOT** be freed.

Jobs A job record will represent one job submission from a client. It must contain at least the following information:

```
typedef struct job_s
{
    client_t *owner;     /* pointer to the client which owns this job */
    uint32_t status;     /* the state of the job */
    uint32_t exitcode;   /* the exitcode of the job */
    uint32_t pgid;       /* process id of the job */
    uint32_t jobid;      /* the id of the job, unique per client */
    uint32_t maxmem;     /* maximum memory (in bytes) */
    uint32_t maxcpu;     /* maximum cpu time (in seconds) */
    int32_t priority;    /* priority level (niceness) of the job */
    char **envp;         /* environment variables for the job */
    struct job_s *next;  /* next job for client */
    struct job_s *snext; /* next job in list of all jobs on server */
} job_t;
```

In addition, a job must be in of the following states: - `NEW` : denotes this job is freshly submitted, and has not executed or been canceled yet - `RUNNING` : denotes this job is currently executing - `SUSPENDED` : denotes that the execution of this job is currently stopped - `EXITED` : denotes that this job completed, exited, and was `wait(2)`'d for - `ABORTED` : denotes that the execution of this job was

aborted due to some signal - `CANCELED` : denotes that the job has been canceled, and is currently in a stage wherein it is expected to be `wait(2)`'d for

The server shall redirect standard output and standard error of any executed job to files that the client can later request the contents of. These files shall be named `username_timeofsubmission.out` for standard output and `username_timeofsubmission.err` for standard error.

Jobs will be limited to an upper bound on resource usage for memory and cpu time using `setrlimit(2)` and the appropriate flags for `RLIMIT_CPU` and `RLIMIT_AS`. The priority level of a job will be set using `setpriority(2)`.

Upon termination of child processes, the server shall use `wait4(2)` to reap any necessary zombie processes. `wait4(2)` should be used, since it populates a `struct rusage` for the reaped process. This structure will contain the resource usages of the reaped process, and examining it can help the server and client determine the reason for the termination of the child, in the case that the child went over its resource limits.

Jobs shall be stored on the server in a job list of all jobs known to the server. In addition, each client record shall contain a list of all jobs associated with that client.

Client Program

The client program should conform to the following command line options:

`-f socketfile`: Use the specified filename as the UNIX domain socket, or `.cse376hw4.socket` if this option is not specified. `-d`: Enables debugging output. `-c`: Specify a command to be executed. Must be combined with the `-u` flag. If this flag is specified, the client shall *ONLY* log in, execute the command, disconnect, and terminate cleanly. `-u`: Specify the user to log in as. If this is not specified, then the client shall prompt the user for a username upon startup.

In addition, the client should support the following commands: - `submit [max_cpu] [max_mem] [pri] [cmd]`: Submit a new job to the server, with the specified resource limitations given by max_cpu and max_mem, running at priority pri - `list`: List all jobs for client - `stdout [jobid]`: Get the standard output results of the specified completed job - `stderr [jobid]`: Get the standard error results of the specified completed job - `status [jobid]`: Get the status of the job with the specified id - `kill [jobid]`: Terminates the job with the specified id - `stop [jobid]`: Stops the job with the specified id - `resume [jobid]`: Resumes a stopped job with the specified id - `pri [jobid] [priority]`: Adjust the priority level of a job - `expunge [jobid]`: Removes the specified job from the client's list of jobs - `help`: Displays this list of commands - `quit`: Disconnect and close the client

If the client is running in interactive mode (`-c` is **NOT** specified), then the client shall enter into an input loop (similar to a shell) and be able to receive asynchronous notifications about job updates from the server. The client shall make use of a `client_handle_server()` function, wherein it will handle and resolve any communications from the server which the client did not initiate. Note that the server should only be sending `JOB_UPDATE` packets to the client without a prior client request.

Protocol

The protocol used for communication between server and client shall be packet-based. The following packet types are defined for use: - `ACK`: acknowledgement of command successfully received/processed - `NACK`: command was unsuccessfully received/processed - `LOGIN`: should be first packet sent by client

on connection. Followed by length of username, then username. Expects either an `ACK` or `NACK` response. - `JOB_SUBMIT`: client wants to submit a new job. Followed by a `submission_t`. Expects either a `NACK` or `JOB_SUBMIT_SUCCESS` response. - `JOB_STATUS`: client wants to know status of a job. Followed by the client job id. Expects either a `NACK` or `JOB_STATUS` response. - `JOB_SIGNAL`: client wants to kill/stop/start a job. Followed by a `signal_t`. Expects either a `NACK` or `ACK` response. - `JOB_SET_PRI`: client wants to change the priority of a job. Followed by a `priority_t`. Expects either a `NACK` or `ACK` response. - `JOB_GET_STDOUT`: client wants the standard output of a job. Followed by the client job id. Expects either a `NACK` or `JOB_RESULTS` response. - `JOB_GET_STDERR`: client wants the standard error of a job. Followed by the client job id. Expects either a `NACK` or `JOB_RESULTS` response. - `JOB_LIST_ALL`: client wants a list of all their jobs. Expects either a `NACK` or `JOB_LIST_ALL_RESP` response. - `JOB_EXPUNGE`: client wants to remove a job from their joblist. Followed by the client job id. Expects either a `NACK` or `ACK` response. - `JOB_UPDATE`: sent by server to client when status a job changes. Followed by an `update_t`. No response. - `JOB_SUBMIT_SUCCESS`: server response to `JOB_SUBMIT` when job was successfully submitted to server (server should send a `NACK` on error). - `JOB_RESULTS`: sent by server to client, packet contains results of a job (server should send a `NACK` on error). - `JOB_STATUS_RESP`: sent by server to client as a response to a client's `JOB_STATUS` request - `JOB_LIST_ALL_RESP`: sent by server to client as a response to a client's `JOB_LIST_ALL` request

The transmission of these packets and implementation of their protocols shall be achieved by the following functions: - `int send_pkt(int fd, int packet_type, void *payload)` where `fd` is the file descriptor to write the packet to, `packet_type` is the type of packet being written, and `payload` is a pointer to the payload being written. This function shall return `0` on success and `-errno` on error. - `int recv_pkt(int fd, void **payload)` where `fd` is the file descriptor to read from and `payload` is a pointer to a pointer denoting where to store the received data. This function shall return the packet type which was received on success and `-errno` on error.

The following structures are defined for transmissions of these packet types, and the program shall use at least these structures for transmission of information between server and client:

```
typedef struct submission_s
{   /* for JOB_SUBMIT requests */
    uint32_t maxcpu;
    uint32_t maxmem;
    int32_t priority;

    uint32_t cmdlen;
    char *cmdline;

    uint32_t envpc;
    char **envp;
} submission_t;

typedef struct status_s
{   /* for JOB_STATUS requests */
    uint32_t status;
    int32_t exitcode;

    uint32_t maxcpu;
```

```
    uint32_t maxmem;
    int32_t priority;

    struct rusage ru;
} status_t;

typedef struct update_s
{   /* for JOB_UPDATE notifications */
    uint32_t jobid;
    uint32_t status;
} update_t;

typedef struct listing_s
{   /* for response to JOB_LIST_ALL requests,
       as a JOB_LIST_ALL_RESP response */
    uint32_t jobid;
    uint32_t left;
    uint32_t cmdlen;
    char *cmdline;
    uint32_t status;
    int32_t exitcode;

    struct listing_s *next;
} listing_t;

typedef struct priority_s
{   /* for JOB_SET_PRI requests */
    uint32_t jobid;
    int32_t priority;
} priority_t;

typedef struct signal_s
{   /* for JOB_SIGNAL requests */
    uint32_t jobid;
    uint32_t signal;
} signal_t;

typedef struct results_s
{   /* for response to JOB_GET_STDOUT and JOB_GET_STDERR requests,
       as a JOB_RESULTS response */
    uint32_t length;
    char*    results;
} results_t;
```