Assignment #2

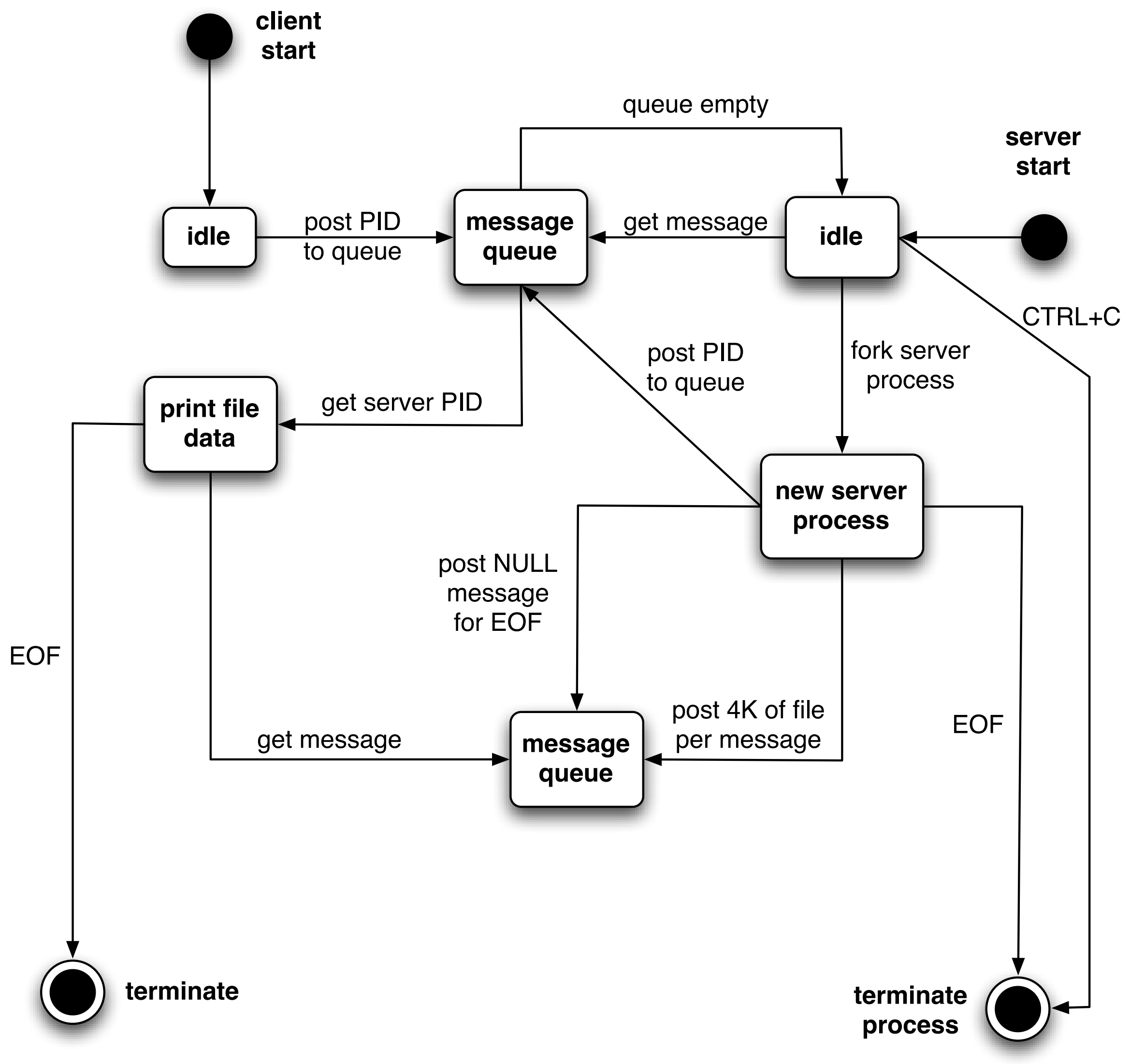
Message Queue File Server

Steffen L. Norgren
A00683006

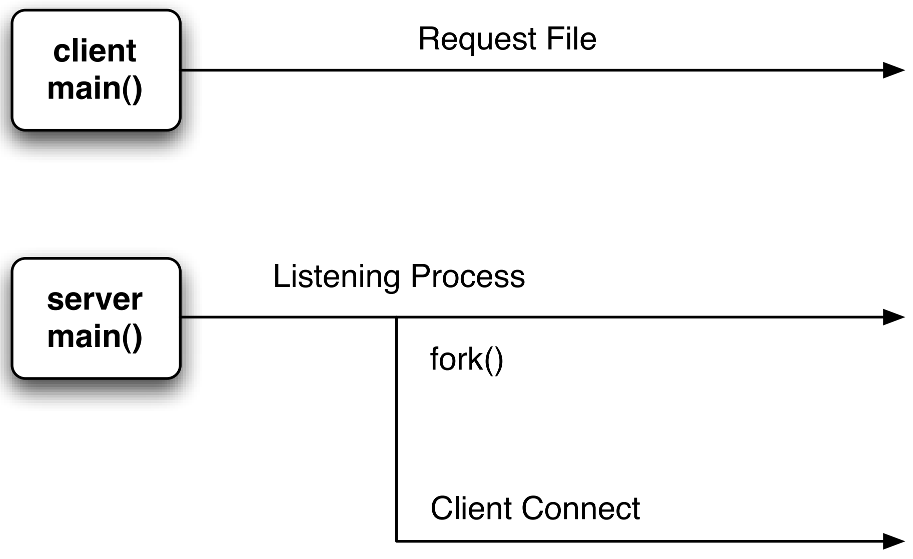
COMP 4981 • BCIT • February 12, 2009

DESIGN

General Overview



The way this multi-client message-queue file server works is that upon starting up the server goes into a loop, reading the message queue, looking for a client to post its PID to the queue. When a client is started up, it will post its PID to the queue and the server will respond by forking and then posting its new PID to the queue. This allows the client and server to put their respective PIDs in the message type. That way messages never get misdirected because the server is only retrieving messages from the client's PID and vice-versa.



The client process consists of a single process and never forks. However, upon requesting a file from the server, the server's main listening process will fork and serve the client's request. Upon completion of the request, the forked server will terminate.

Client - Pseudo-Code

```
while (TRUE) {
    pid_client = getpid();

    // Send PID, file name and priority to queue
    mesg_send(pid_client, file_name, priority);

    mesg_recv(server_pid, data);

    if (data == NULL) {
            // the file does not exist
            exit();
    }
    else {
            while (TRUE) { // loop until NULL
                  if (data == NULL) {
                        // file is complete
                  }
                  else { // print the file data
                        printf(data);
                  }
            }
    }
}
```

Message Handler

```
void mesg_send(char * data, int type, int priority) {
    struct message;

    // create message struct and assign values

    key = ftok(".", 'a');
    qid = msgget(key, 0666 | IPC_CREAT)

    msgsnd(qid, &message, length, 0);
}

int mesg_recv(int type, char * data) {
    struct message;

    key = ftok(".", 'a');
    qid = msgget(key, 0666 | IPC_CREAT)

    msgrcv(qid, message, length, type, 0);

    // retrieve message struct contents

    return priority;
}
```

Server - Pseudo-Code

```
pid_master = getpid();

while (TRUE) {
    connect_idle()
}

void connect_idle() {
    mesg_recv(1, data);

    client_pid = data;

    pid_child = fork();

    if (getpid() == pid_master) {
            return; // go back to idle loop
    }
    else {
            serve_client(client_pid);
    }
}

void serve_client(client_pid) {
    buffer[MAX_DATA];

    mesg_send(getpid(), 2); // send PID to client

    priority = mesg_recv(pid_client, data); // get file and assign pri-
ority

    fp = fopen(file, "r");

    while (!EOF) {
            // seek through the file 4K at a time and post to local
buffer

            // Send the data from the buffer
            mesg_send(pid_client, data);
    }
    // send null to signify end of file
    mesg_send(pid_client, NULL);

    exit(); // terminate process
}
```

TESTING

Tests & Results

Testing for this project was done by creating a large text file that contains number sequences. This was easily done via a shell script that is included with the project, *mktest.sh*. This scrip creates files varying in size from just over 4KB to 1GB.

The client program outputs all of the file to stdout and everything else to stderr. That way you can type **./client testData 0 > outData** and then run **diff testData outData** to see if there is any difference between the input and output files.

This was very handy throughout debugging and has resulted in an error-free file-transmission within the final program.

Additionally, since the program is entirely command-line driven, you can type **time ./client testData 0-x** (X being a very large number) and get the real time the system spent retrieving the data.

This program has been tested to work with text files up to 2GB in size. However, there is a bug that causes the server to produce a “file not found” error if you attempt to request a file that beings with a null terminator.