Michael Russo - mr880
Criag Sirota - cms631

Overview:

Our project took a great deal of direction from Beej's guide to network programming. https://beej.us/guide/bgnet/html/multi/syscalls.html#socket. This was important in order to ensure the network programming worked beyond a local host, which was an early issue we ran into.

Client.c

Our client program operates fairly standardly as far as socket programming goes with some changes in how we spawn two threads after connecting to a server socket. When a connection is made inside of main we spawn..

1. A thread that reads in messages from the server
2. A thread that sends messages to the server

The application of the first thread (reads in from server) is fairly simple once we figured out that we could make a read system call the argument of a while loop where each iteration is a reading in of the server writes. Two reads trigger immediate action on the client side. Both exit the client safely.

1. "Quitting.."
2. "** Server disconnected **"

The first is a normal quit command, as entered by the user on the client side. This input reaches the server, which sends back "Quitting" after doing some housekeeping on the server side, prompting our condition to safely exit the client via the exit_func( ). The other is in the case of a server shutdown while the client is still running, which performs the same function as receiving the "Quitting.." server write.

exit_func( ) is used in our signal system call, which we initialize in our main function. Mentioned above, it is used to safely exit our client. It does this by making sure to close the client's socket and performing an exit(0) call.

Functions Used:

void exit_func( )
void* send_user_commands(void* fd)
void* outputFromServer(void* fd)
void* get_in_addr(struct sockaddr *sa)

Server.c

Our server program begins by setting a SIGINT signal for a safe disconnect and immediately after spawns a thread. This thread is passed in the port number given to it by the user via the command line (argv[1]). From here, the socket server is pretty boilerplate until we leave the

program running in a while loop, waiting to accept new client sockets and spawning new threads to begin the client service.

It it's the first time a socket is accepted inside the while loop specified above, we display on the server our column headers, "Account, Balance, InSession" and spawn a thread that sets an alarm. This alarm is how we update our server's data.

```
void* set_alarm()
{
      signal(SIGALRM, print_accounts);
      //schedule the first alarm
      alarm(15);

      while(1)
            pause();

      pthread_exit(0);
}

void print_accounts()
{
      PAUSE = 1;
      system("clear");
      printf("\t\t\t\tBank-System\n");
      printf("Account Name\t\tCurrent Ballance\t\tIn Service\n");
      struct Account* temp = NULL;
      temp = head;

      while(temp != NULL)
      {
            printf("%s\t\t\t$%.2f\t\t\t\t%d\n", temp->name,
temp->balance, temp->inSession);
            temp = temp->next;
      }
      //sleep(5);
      PAUSE = 0;
      alarm(15);

}
```

Taking a closer look, we use semaphores (PAUSE) to isolate the call to print out our collected data and then set the alarm for 15 seconds. Upon the alarm thread's creation, we create our client handler's thread.

Client handler does the account creation and services. Here in order to maintain synchronization, we use mutex locks when we need to access and manipulate data that was stored in the heap.

Two examples of mutex locks being used for synchronization.

```
void deposit(double amount, char* name)
{
	struct Account* temp = NULL;
	temp = head;
	//printf("NAME: %s\n", name);
	while(temp != NULL)
	{
		if(strcmp(temp->name, name) == 0)
			break;

		temp = temp->next;
	}
	pthread_mutex_lock(&lock);
	temp->balance += amount;
	pthread_mutex_unlock(&lock);
	printf("Deposited Money\n");
}

double withdraw(double amount, char* name)
{
	struct Account* temp = NULL;
	temp = head;
	//printf("NAME: %s\n", name);
	while(temp != NULL)
	{
		if(strcmp(temp->name, name) == 0)
			break;

		temp = temp->next;
	}
	if((temp->balance - amount) < 0)
		return -1.1;
	pthread_mutex_lock(&lock);
	temp->balance -= amount;
	pthread_mutex_unlock(&lock);
	return temp->balance;
}
```

Once we are finished on the server side, we use a signal interrupt from the keyboard (SIGINT) in order to shut things down on the server side. On top of safely shutting things down on the server side, the server sends a message to all open clients prompting their individual shutdown processes as well.

Inside server_handler...

```
newsockfd = accept(sockfd, (struct sockaddr *) &their_addr,
&addr_size);

...

struct socket* new_socket = (struct socket*)malloc(sizeof(struct
socket));
new_socket->fd = newsockfd;
new_socket->next = sockhead;
sockhead = new_socket;
```

… we accept connections from the client and get a file descriptor specific to the client. This file descriptor is saved after accepting into a linked list. We use this linked list to iterate through in our signal interrupt handler (disconnected).

cntrl + C calls disconnected…

```
void disconnected(){

      char buffer[255];
      bzero(buffer, 255);
      printf("Server is disconnecting...\n");

      struct socket* tempsock;
      tempsock = sockhead;

      while(tempsock != NULL)
      {
            send(tempsock->fd, "** Server disconnected **",25,0);
            bzero(buffer, 255);
            tempsock = tempsock->next;
      }
      free(head);
      free(sockhead);
      exit(0);
}
```

On the client side....

```
void* outputFromServer(void* fd)
{
      …
      while(recv(newfd, buff, sizeof(buff), 0) > 0)
      {
            if(strncmp(buff, "** Server disconnected **", 25) == 0)
            {
                  printf("** Server disconnected **\n");
                  exit_func();
            }

            …
      }
      …
}
```

If the client catches this message, it safely shuts down just like it would if the same call was made on the server side.