CS39002: Operating Systems Lab
Spring 2012

Assignment 4
Due: February 16, 2012, 1 pm

In this assignment, you write a multi-process message-passing utility. The messages are stored in a shared-memory segment. A static library is created which provides a set of functions to send/receive messages; every user process links with this library and uses these functions for passing messages to other processes. Every access (read or write) to this shared-memory segment is to be guarded by semaphores to effect mutual exclusion. Named processes can transfer messages among each other using this interface.

Data Structures

A *message* consists of the following four fields:

1. A sender name (a string of maximum length 10)
2. A recipient name (a string of maximum length 10)
3. A sending time (in whatever format you would like to store)
4. A text message (a string of maximum length 100)

A *registered process* consists of the following two fields:

1. A name (a string of maximum length 10)
2. A process ID

The Controller Process

This process creates a shared-memory segment (henceforth referred to as the *message* segment) for storing a maximum of 100 messages. It also creates another shared-memory segment (the *registry segment*) for storing an array of at most 10 registered processes. Finally, the controller process creates two semaphores for the mutually exclusive accesses of the two segments. The shared-memory segments and the semaphores should have keys based upon the `ftok()` call on some predetermined file and/or directory names.

After this initial book-keeping task, the controller process keeps on periodically checking the message segment at regular intervals (like every one minute). In this checking phase, the controller process deletes any message that has been left unread for a period of five minutes or longer (you can set this time to any value for your testing).

Note that since messages can be added or deleted at any time, you will have holes in your message segment. So a field needs to be kept with each element of the message segment to indicate if it is free, or if a valid message is stored in that element currently.

The Static Library

Write a static library (like `libmympl.a`) and an associated header file (like `mympl.h`) to build a message-passing API (application programming interface). Any user process performs message-related tasks by invoking these API functions only. The static library should consist of the following functions.

A) **Registration:** Initially, each user process needs to self-register itself in order to send and receive messages. The process involves sending its formal name to an API function with the following prototype.

```
int register ( char *procname );
```

If the table holding the list of registered users in the registry segment is not full, `procname` and the PID of the calling process are stored in a free entry of the table, and 1 is returned. If the table is full, the registration is unsuccessful, nothing is stored, and –1 is returned. If another process has already registered the same name, –2 is returned. If registration is unsuccessful for any other reason, –3 is returned.

B) **Sending messages:** When a process with name *P* wants to send some message *m* to a process with name *Q* (note that *P* and *Q* are string names, NOT PIDs), *P* invokes the following API function with the two input values: the recipient's name *Q*, and a text message *m*.

```
int sendmessage ( char *recptname, char *textmsg );
```

The function first checks if the sender is registered or not (how does it know who the sender is?). If the sender is not registered, the function fails, and returns –1. Otherwise, it checks whether the message segment has an empty slot (start checking from the start to find the first free slot) to accommodate this new message. If not, the send-message request fails again, and –2 is returned. Otherwise, *P*, *Q*, *m,* and the current system time *t* are inserted in an empty slot in the message segment, and 1 is returned.

During sending a message to another process *Q*, it is not necessary that *Q* is already a registered process name. A process with the name *Q* may join later to read messages sent before its registration (unless the controller process deletes them because of timeout).

C) **Receiving message lists:** When a process with name *P* wants to see the list of all messages waiting for it, it makes an API function call that *prints* the list of message headers (consisting of only the sender names and the times of sending).

```
int listmessages ( );
```

The function returns the number of messages waiting for the process on success, and –1 on failure. Note that if no message is waiting for the process, it returns 0, not –1. This function just shows the list, no messages are deleted from the message segment.

D) **Reading messages:** If a process *P* wants to read one message waiting for it, it invokes the following API call.

```
int readmesssage ( message *buffer );
```

The function copies the *oldest* message waiting for *P* to the local buffer passed as the argument. The return value is 1 if the copy is successful, 0 if no message is waiting for *P*, and –1 on failure. The API call also deletes the copied message from the message segment.

E) **Deregistration:** Before a user process *P* quits, it needs to make an API function call in order to delete *P* along with its PID from the registry segment entry. The API call would be as follows. It returns 1 on success, –1 on failure.

```
int deregister ( );
```

Note that a new process with a different PID may reuse the name *P*. Messages for *P* may still be there in the message segment, which can be read by this new process.

User Processes

Each user process uses only the API calls without bothering about the implementation details (so it links with the library you created). First, the process registers itself to the messaging system using the `register()` call (it will ask the user for a name to use). If the call fails, the user process quits. Otherwise, it enters a loop to send and/or receive messages interactively (that is, as and so long as the user demands). This should be through a text-based menu which gives options to (i) send a message, (ii) list messages waiting for the process name, (iii) read a message, and (iv) quit. If quit is seleted, the process should call `deregister()`, and then quit. For sending a message, the recipient name and the message will be input by the user. For list/receive, the output should be shown on the display.

Submit the following files in a tarball:

1. A header file for the static library: `mympl.h`
2. The archivable source code (without `main()`) for the static library: `mympl.c`
3. A C file (with `main()`) for the controller process: `controller.c`
4. A C file (with `main()`) for a user process: `msgclient.c`
5. A `Makefile` for compiling your source codes properly. It should create the static library `libmympl.a`, and two executable files: `controller` and `msgclient`.
