Project #3
Message Passing Facility (MF Lib)

Assigned: Mar 26, 2024 Document version: 1.2
Due date: April 20, 2024

- You will develop the project in C/Linux. Submitted projects will be tested in Ubuntu 22.04 Linux 64-bit.
- You are not allowed to share this project and/or its solution.
- *Objectives/keywords*: Learn and practice with synchronization, memory management, memory addressing, virtual addresses, semaphores, shared memory, message passing, library development.

In this project, you will develop a message passing facility (MF) that enables applications to send messages to each other via one or more message queues. The facility will be implemented as a library that will be linked to applications (processes) interested in message passing. Internally, shared memory will be utilized to implement the message queuing abstraction. Your library will also handle synchronization among multiple processes that will be exchanging messages via message queues stored in shared memory. The library will be named as **libmf** (MF library).

1. Library Interface (API)

You will implement the following functions in your library. You are not allowed to change the function prototypes defined here. Internally, in your library, you can define and use additional functions. The ones below are the **interface functions**, the functions that can be called by applications (API functions).

Some of these functions (`mf_init()` and `mf_destroy()`) will be called by a special first program that will be started to create the facility (i.e., to create and initialize a shared memory segment). The name of this special program will be **mfserver**. You will develop it. The *rest of the functions* below will be called by applications (**processes**) that would like to use the messaging facility.

1. `int` **`mf_init`**`()`. The **mfserver** program will invoke this function (not applications). Its purpose is to establish (create) a shared memory area (region) where message queues and their associated messages will be stored, and accessed by the library. This function handles all essential initialization tasks, including the creation of required synchronization objects such as semaphores. Parameters such as the shared memory's name and size will be obtained from a configuration file (`CONFIG_FILENAME`). Upon successful completion, the function will return 0 (`MF_SUCCESS`); otherwise, it will return -1 (`MF_ERROR`).
2. `int` **`mf_destroy`**`()`. This function will be invoked by the mfserver program during termination. Its purpose is to perform all necessary deallocation and cleanup tasks. This includes removing the shared memory and all the

synchronization objects to ensure a clean system state. Upon successful completion, the function will return 0; otherwise, it will return -1.

3. `int` **`mf_connect`**`()`. This function will be called by each application (process) intending to utilize the MF library for message-based communication. It will read the configuration file CONFIG_FILENAME to retrieve the shared memory's name and size. Subsequently, it will perform the required initialization for the process. Upon successful initialization, the function will return 0; otherwise, it will return -1.

4. `int` **`mf_disconnect`**`()`. This function will be invoked by an application (process) that no longer requires the messaging library. The library will remove this process from the list of active processes utilizing the library. Upon successful disconnection, the function will return 0; otherwise, it will return -1.

5. `int` **`mf_create`**`(char *mqname, int mqsize)`. This function creates a new message queue and initializes the necessary information for it. It allocates sufficient *contiguous* space, as specified by the `mqsize` parameter, for the message queue, which is initially empty. Additionally, the function assigns a unique message queue identifier (qid) to the newly created queue. Each message queue within the library is identified by a unique identifier. Upon successful completion, the function returns 0; otherwise, it returns -1.

6. `int` **`mf_remove`**`(char *mqname)`. This function removes a message queue. It deallocates the space in the shared memory used by the message queue. The message queue is associated with a reference count: each time the queue is opened, the reference count increases; each time it's closed, the count decreases. Therefore, the message queue will only be removed if the reference count reaches 0; otherwise, it remains intact. Upon successful removal, the function returns 0; otherwise, it returns -1.

7. `int` **`mf_open`**`(char *mqname)`. This function opens a message queue for sending or receiving messages. If successful, it returns the queue ID (qid) of the message queue. Otherwise, it returns -1.

8. `int` **`mf_close`**`(int qid)`. This function closes a message queue. After closure, access to the queue by the application should not be allowed. The reference count of the message queue will be decremented by 1. Upon success, the function returns 0; otherwise, it returns -1.

9. `int` **`mf_send`**`(int qid, void *bufptr, int datalen)`. This function sends a message to the message queue. It can block the caller until space is available in the queue. The message, obtained from the memory space pointed to by `bufptr`, is copied to the message queue buffer in the shared memory of the library. The size of the message, indicating how many bytes need to be copied from the application into the library, is specified by the `datalen` parameter.

   A message is simply a sequence of bytes (characters), and its interpretation depends on the application. The library may also store additional information about the message, such as its size.

   If there is insufficient space to store the message in the shared memory buffer, the calling process will be blocked, and the function will not return immediately. The process will be awakened later when sufficient space

becomes available in the message queue buffer. At that point, the message will be copied to the shared memory buffer, and the function will return.

Upon successful completion, the function returns 0; otherwise, it returns −1.

10. `int` **`mf_recv`**`(int qid, void *bufptr, int bufsize)`. This function retrieves a message from the message queue, blocking the caller if no message is available for removal. Messages are removed in FIFO order, adhering to the queue principle. The retrieved message (data) is copied to the memory space in the application pointed to by `bufptr`. The size of the data (message) is internally stored in the library buffer alongside the data, allowing the library to determine the number of bytes to copy to the application buffer space pointed to by `bufptr`. After copying, the message is deleted from the library's shared memory buffer.

    The `bufsize` parameter of this function specifies the size of the application buffer where the incoming message is to be stored; it does not represent the length of the incoming message. The `bufsize` parameter value (i.e., application buffer size) must be larger or equal to `MAXDATALEN` to ensure sufficient space in the application buffer for any incoming message.

    Note that a message is received as a whole, not partially. Upon successful retrieval, the function returns the size of the received message. Otherwise, it returns −1.

11. `int` **`mf_print`**`()`. This function prints information about the state of the shared memory and message queues. It serves for testing and debugging purposes.

You will implement the MF library with the specified interface (API) and behavior. An application wishing to utilize the library will include the `mf.h` header file and link with the library using the `-l` option of the gcc compiler (as `-lmf`).

2. Starting the Facility

A server program, named `mfserver`, will use the `mf_init()` function to create a shared memory region and initialize it. The server must remain running in the background (mostly sleeping) until termination, so that it can perform a cleanup when it is terminated. Termination can be achieved through the kill command or by pressing `Ctrl-C` or `Ctrl-D` at the command line if the server is running in the foreground. A signal handler within `mfserver` will manage termination requests, calling the `mf_destroy()` function for necessary deallocation and cleanup.

During the period between `mf_init()` and `mf_destroy()`, the server will not engage in extensive activity. It may simply sleep for most of the time. The following can be a partial implementation of the `mfserver`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
```

```
#include "mf.h"

//
// write the signal handler function
// it will call mf_destroy()
//

int
main(int argc, char *argv[])
{
    printf ("mfserver pid=%d\n", (int) getpid());
    // register the signal handler function
    mf_init(); // will read the config file
    // do some initialization if needed
    while (1)
        sleep(1000);
    exit(0);
}
```

3. Configuration

A configuration file with name `CONFIG_FILENAME` (which is to be defined in `mf.h`) will store some of the parameters of the MF library. They will be the configurable (changeable) parameters of the library. These are:

1. `SHMEM_NAME`: The name of the shared memory region (i.e., segment) to use.
2. `SHMEM_SIZE`: The size of the shared memory region to use.
3. `MAX_MSGS_IN_QUEUE`: The maximum number of *messages* (data items) allowed in a message queue.
4. `MAX_QUEUES_IN_SHMEM`: The maximum number of *message queues* allowed in the shared memory).

The non-configurable parameters of the MF library are the following. They are to be defined in `mf.h` file.

1. `MIN_DATALEN`: The minimum size that a message can have.
2. `MAX_DATALEN`: The maximum size that a message can have.
3. `MAX_MQNAMESIZE`: The maximum size of a message queue name.
4. `MIN_MQSIZE`: The minimum size of a message queue - in KB.
5. `MAX_MQSIZE`: The maximum size of a message queue - in KB.
6. `MIN_SHMEMSIZE`: The minimum size of the shared memory segment - in MB.
7. `MAX_SHMEMSIZE`: The maximum size of the shared memory segment - in MB.
8. `CONFIG_FILENAME`: The name of the configuration file (an ascii text file).

Information about the possible values of these parameters will be included in the **Clarifications** page.

4. Memory Management

Your library will manage the shared memory segment with the memory management scheme that you design. For each message queue, a *contiguous* space (called

a buffer) should be allocated in the shared memory. It will be a circular buffer, with messages placed one after another in the allocated space. Similarly, messages will be removed in the order they are placed into the queue, implementing a FIFO queue. Dynamic allocation for messages inside a message queue space will not be used. Instead, a new message (item) will simply be added to the end of the other items in the allocated space (buffer). Besides the data for an item, additional information about the item, such as its size, can be stored.

The size of the message queue is specified by the application calling `mf_create()`. Consequently, space must be allocated from shared memory for message queues of various sizes. This presents a dynamic storage allocation problem, which can be addressed using various methods such as the hole-list method, bitmap method, buddy-system allocation method, and others.

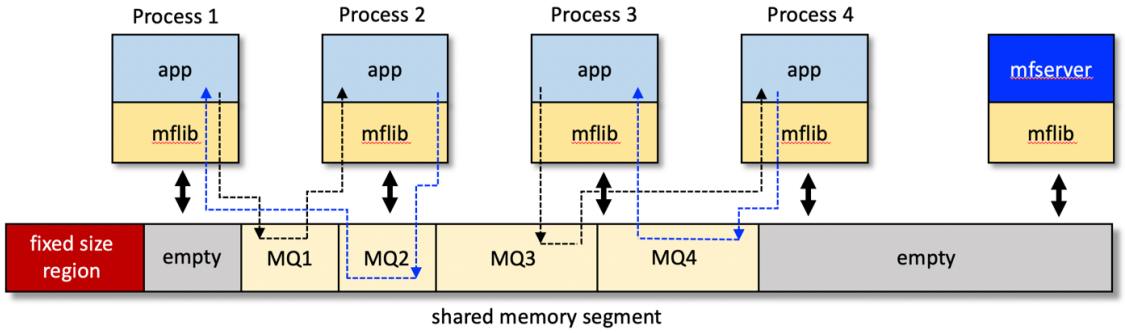

Figure 1: Figure shows the runtime structure where multiple processes are linked with the MF library. Some message queues are created. They are are sitting in the shared memory (each message queue is allocated contiguous space from the shared memory). The library manages the shared memory. The `mfserver` process is responsible from creation and initialization of the shared memory. In this example, processes P1 and P2 are communicating with message queues 1 and 2. Processes P3 and P4 are communicating with message queues 3 and 4.

A fixed portion of your shared memory can be allocated for storing management information and structures. For instance, names of semaphores can be stored in this portion. Additionally, details such as the number of queues created, various configuration parameters, and even specific information about each message queue can be stored here. These are just examples, and you will determine the specifics of your memory management scheme and decide what information to store where in the shared memory.

5. Synchronization

Access to the shared memory and message queues should be **synchronized** using POSIX named **semaphores** or POSIX mutex and condition variables. It's essential to investigate the suitability of each option. Unlike synchronizing threads, we'll be synchronizing processes that do not typically share address space. Hence, the mutex and condition variables used must be made shared among multiple processes.

For each message queue you need to use a separate lock (or semaphore) so that we can also achieve maximum *concurrency* possible. For instance, if one process pair is using message queue 1, another process pair should be able to simultaneously use message queue 2. This level of concurrency cannot be achieved with a single lock for all message queues. Therefore, each message queue should have its own lock.

6. Application(s)

Applications linked with the MF library begin by calling the `mf_connect()` function, initializing the library for their use.

Below is a sample application utilizing the library. It involves two processes: the main process (P1) and a child process (P2). Both processes initially call `mf_connect()` to connect and initialize the library. The main process is tasked with creating a message queue, while the child process (P2) waits until the queue is created. Subsequently, both processes open the queue, with the parent sending messages to it and the child receiving messages from it. Finally, the parent removes the queue.

This example illustrates the basic usage of the MF library for message-based inter-process communication.

```
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include "mf.h"

#define COUNT 5
char *semname1 = "/semaphore1";
char *semname2 = "/semaphore2";
sem_t *sem1, *sem2;
char *mqname1 = "/msgqueue1";

int
main(int argc, char **argv)
{
    int ret,  i, qid;

    sem1 = sem_open(semname1, O_CREAT, 0666, 0); // init sem
    sem2 = sem_open(semname2, O_CREAT, 0666, 0); // init sem

    ret = fork();
    if (ret > 0) {
        // parent process - P1
        char *bufptr = (char *) malloc (MAX_DATALEN);
        sem1 = sem_open(semname1, 0);
        sem2 = sem_open(semname2, 0);
```

```
        mf_connect();
        mf_create (mqname1, 16); //  create mq;  16 KB
        qid = mf_open(mqname1);
        sem_post (sem1);

        for (i = 0; i < COUNT; ++i) {
            sprintf (bufptr, "%s-%d", "MessageData", i);
            mf_send (qid, (void *) bufptr, strlen(bufptr)+1);
        }
        free (bufptr);
        mf_close(qid);
        mf_disconnect();
        sem_wait(sem2);
        mf_remove(mqname1);    // remove mq
    }
    else if (ret == 0) {
        // child process - P2
        char *bufptr = (char *) malloc (MAX_DATALEN);
        sem1 = sem_open(semname1, 0);
        sem2 = sem_open(semname2, 0);
        sem_wait (sem1);
        mf_connect();
        qid = mf_open(mqname1);
        for (i = 0; i < COUNT; ++i) {
            mf_recv (qid, (void *) bufptr, MAX_DATALEN);
            printf ("%s\n", bufptr);
        }
        free (bufptr);
        mf_close(qid);
        mf_disconnect();
        sem_post(sem2);
    }
    return (0);
}
```

7. Additional Information and Constraints

1. Note that message boundaries will be preserved. This means that if the sender has sent a message of size N (N bytes sent), the receiver will receive it entirely in a single `mf_recv()` call (N bytes received).
2. Internally, ensure that spaces (structures) are allocated in multiples of 4 bytes to avoid alignment problems. For instance, when allocating space in the queue for a message sent from a process, ensure that the space allocated is a multiple of 4 bytes, even if the process intends to send only a single byte. Also make sure that integer fields in a structure start at addresses (offsets) that are multiples of 4. This alignment practice prevents potential alignment-related issues at run time.
3. It's important to note that storing pointers (C pointer values) in shared memory is not advisable as their references may not hold significance across different processes. A byte in the shared memory region may have different

C pointer values in different processes. Therefore, use relative addressing (relative to the beginning of the shared memory region) mechanisms like offsets. For instance, referring to the first byte of shared memory with address (offset) 0 provides a consistent reference point across processes without relying on absolute memory addresses (C pointer values).

8. Tips, Clarifications, Additional Requirements

1. Tips, clarifications, additional requirements about the project will be added to the following *web-page*: `https://www.cs.bilkent.edu.tr/~korpe/courses/cs342spring2024/dokuwiki/doku.php?id=p3-s24`
2. The page above will also be linked from course's Moodle page (under project document). Please check this project web-page regularly to follow the clarifications.
3. This page may include additional requirements or specifications. If there is a conflict between a requirement stated here and requirement stated in the Clarifications page, you need to follow the one in the Clarifications page. Clarification page requirements will supersede the requirements here (if any).
4. Further *useful information*, *tips*, and *explanations* on this page will help you in developing the project. It will be a dynamic, living page, that may be updated (based on your questions as well) until the project deadline.
5. An initial project skeleton (including a `Makefile`) is added to github at the following address. You can start with this. `https://github.com/korpeoglu/cs342spring2024-p3`

9. Experiments and Report

You will design and conduct some performance experiments. You will then interpret them. You will put your results, interpretations, and conclusions in a report.pdf file. You will include this file as part of your uploaded package. Experiments and the related report will be **20% of the project grade**.

10. Submission

Submission will be the same as the previous projects.

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source file(s). We should be able to compile your program(s) by just typing `make`. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312` and `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

`tar cvf 21404312-214052104.tar 21404312-214052104`
Then they will gzip (compress) the tar file as follows:
`gzip 21404312-214052104.tar`
In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

References

[1] Overview of POSIX Semaphores. `https://man7.org/linux/man-pages/man7/sem_overview.7.html`

[2] Overview of POSIX Shared Memory. `https://man7.org/linux/man-pages/man7/shm_overview.7.html`.

[3] In this project you will not use threads, but still you can use mutex and condition variables if you can share them between multiple processes. See if you can (if you want to use mutex/condition variables) instead of semaphores). `https://man7.org/linux/man-pages/man7/pthreads.7.html`