CSCI 4061: Introduction to Operating Systems
FALL 2016
Assignment 3: Asynchrony

Due: November 21st 11:55pm, you may work in a group of 2 or 3.

Purpose:

In this assignment, you will become more familiar with the issues that surround dynamic memory management: dynamic memory allocation and deallocation. Understanding these issues is important in designing memory-efficient run-time systems, an important task of the systems programming.
You will make use of Unix library/system calls for memory management: `malloc,free,memcpy`.
You will also measure the performance of your dynamic memory manager and show (hopefully) how we can outperform the Unix heap management routines! In addition, you will also learn about interrupt-driven programming (via alarm signals) and separately compiled functions.

Description:

1. **Dynamic Memory Management Functions**

   Dynamic memory allocation and deallocation in Unix is achieved via the `malloc` family of system calls along with `free`.
   However, for programs that wish to perform a great deal of allocation and deallocation, this introduces a lot of overhead due to the expense of making library/system calls.
   In addition, in some environments `malloc` may not be *thread-safe* or *signal-handler-safe*, meaning that dynamic memory allocation may not work correctly if called by threads or within signal handlers because of race conditions in the heap management routines.

To combat these problems, you decide to write your own dynamic memory manager and make it available to applications that wish to utilize dynamic memory in a more convenient and efficient manner. Your memory manager will "manage" a pool of dynamic memory divided into a fixed number of fixed-size *chunks*. For example, the pool of dynamic memory might be 100 chunks of 64 bytes (a total of 6400 bytes). Since most applications will want memory chunks of some size corresponding to a data structure type, this is a reasonable restriction.

Here is the interface of your memory manager. You must implement the following functions of the memory manager:

```
int mm_init (mm_t *mm, int how_many_chunks, int chunk_size);
//allocate all memory, returns -1 on failure

void *mm_get (mm_t *mm);
//get a chunk of memory (pointer to void), NULL on failure

void mm_put(mm_t *mm, void* chunk);
//give back 'chunk' to the memory manager, don't free it though!

void mm_release (mm_t *mm);
//release all memory back to the system
```

Note that the `mm_init` function should allocate ALL of the memory ahead of time that the manager will use for `mm_get` and `mm_put`.

This creates an upper bound to the total amount of memory that this `mm_t` can give out, which is not *truly* dynamic memory allocation, but is close enough for our purposes.

The idea is that an application would declare a variable of type `mm_t` for *each*

separate dynamic data structure they wished to manage. The main program would then initialize this variable by calling `mm_init`. After that, calls could be made from threads, signal handlers, or any other functions, to get or put memory chunks as needed by the program.

Prior to defining any of these functions you will need to define the type `mm_t`. You will need to keep track of status of memory chunks (free or taken). Within `mm_t`, you are not allowed to use arrays of a fixed-size length. (You cannot assume anything about the maximum number of chunks.)

We have provided the basic framework of these functions for you. We have left code comments for you to fill in your code in specific areas of these files. The file `mm.c` will contain your implementation of the memory manager routines and a timer that you can use.

The file `mm.h` contains the definition of `mm_t`, which you must come up with. You will compile your memory manager as a separately compiled object file (without a main program).
To do this type: `gcc -o mm.o -c mm.c`.

This should produce `mm.o` (assuming there are no errors). This file will be "linked in" with several main programs as described in the next section. For successful linking, you should include the `mm.h` header file at the top of the C files that use these memory management functions.

2. Using the Memory Manager

Part A

Is your Memory Manager more efficient than native `malloc`/`free`? To test this, create an MM instance of a given size (use 1,000,000 objects of size 64 bytes).

Perform 1,000,000 `mm_get`'s followed by 1,000,000 `mm_put`'s and put a timer around this code. For an example of a Unix timer, look at the example we provided in `timer_example()` in `mm.c`. Now, compare this performance against the time to perform 1,000,000 `malloc`'s of size 64 bytes followed by 1,000,000 `free`'s.

I will be impressed if you can defeat the native calls (you should be able to!). For the comparison, write two separate main programs, `main_malloc.c` and `main_mm.c` that work as described above. For `main_mm.c`, you would compile it as:

```
gcc -o main_mm main_mm.c mm.o.
```

Don't forget to include the correct headers in your main program for successful linking.

Part B

Now you will use your memory manager in an application. As stated in the introduction, the use of `malloc` in signal handlers may be a problem, so it may be useful to use your memory manager within a signal handler to return allocated memory.

Now you will implement a simple interrupt- driven message-handling capability using signals. Your application is going to be `sender-receiver` pair where the `sender` sends the messages to the `receiver` and the `receiver` prints the messages on `STDOUT`. However, the message is sent to the `receiver` in the form of packets (fixed-size message fragments). When all of the packets have arrived at the receiver, the message can be assembled in the receiver and printed out. Assembled means that all of the packets can be put together in one big memory-contiguous variable (Hint: `malloc,memcpy` will be useful here).

To model network communication to the receiver, packets will sent asynchronously whenever an alarm goes off (`SIGALRM`) in the `sender`. The communication of

packets between the sender and receiver will be via message-queues. The receiver should not block or poll the queue. Instead, the sender will send SIGIO to the receiver whenever it puts a new packet on the queue. You may need to sleep between separate message transmissions to make sure that the receiver has time to pick up all the packets of the current message before any new ones arrive for the next message.

We will provide the code that creates packets for the sender. The sender needs to know the pid of the receiver to send SIGIO signal to the receiver, hence, at the beginning the receiver will send its pid (call getpid) to the sender on the queue that is set up by the sender. The sender will need the pid when it starts up.
The receiver will have to have a handler for SIGIO which reads the packets from the queue, and stores them in a chunk of memory obtained by the Memory Manager. Once all the packets have arrived for a given message, the receiver will assemble them in a message and print it. You will be manipulating pointers quite a bit including some elementary pointer arithmetic. We have provided a code-shell (packet_sender.c and packet_receiver.c).

packet_sender and packet_receiver take k as the command line argument which is the number of messages sender is going to send to the receiver. Make sure that the value of k is same for both sender and receiver while starting the processes. We are not going to handle the errors where different values of k are passed to the sender and receiver. We will assume the behavior is undefined in that case.
The sender will send the packets out of order to the receiver. Check get_packet() in packet_sender.c which returns a random packet for the given message. Hence, the receiver needs to assemble the message in proper order while storing it in the memory. For example, if the receiver gets the message in following order: “bbb”, “ccc”, “aaa”, then it needs to assemble the message as “aaabbbccc”.

Here are the set of changes you need to do:

In packet_sender.c:

1. Set up a message queue for sending the packets to the `sender`.
2. Read the `receiver's pid` from the queue.
3. Set up timer and handler for `SIGALRM` to send packets to the receiver on a regular interval. You can use the `get_packet` function to create and return a packet. The packet contains the number of packets in that message. In the handler for `SIGALRM`, you send the packet to the `receiver` via the message queue. After sending you will have to send `SIGIO` to the receiver informing it that there is a new packet on the queue.

In packet_receiver.c:

1. Get the message queue set up by the `sender`. Use the same `key` as used by the `sender`.
2. Send the `pid` to `sender` via the queue
3. Write a signal handler for `SIGIO` that reads the packet from the message queue. When a packet arrives (to the handler), you must buffer it somewhere. Hint: your memory manager will be useful here. Each packet carries with it some data, a packet number (0, 1, ...) and the total number of packets for the message. Once you have copied the packet data into the memory provided by the memory manager, you will store it within the `message_t` structure. You might have to block other signals when the handler starts executing as reading might get interrupted due to other signals.
4. Once all packets for a given message are received, allocate the completed memory-contiguous message (you can use `malloc` here since you are not in the handler) and assemble all of the stored packets from the `message_t` structure into this memory. Hint: you will use `memcpy` but you copy all of the packets. Pointer arithmetic will be needed. Print the message once assembled.
5. Remove the queue and release the memory manager.

Do not forget to handle the return types of various system calls and check for errors. Check the comments in `packet.h` for the details of different data structures that can be used for communication between `receiver` and `sender`.

To compile:

gcc -o packet_sender packet_sender.c mm.o
gcc -o packet_receiver packet_receiver.c mm.o

First run the packet_sender and then the packet_receiver.

Useful System calls:
msgget, msgrcv, msgsend, sigaction, setitimer, kill, signal, getpid.

Useful commands:
ipcs -q: to check the status of the message queue created by the process.
ipcrm -q <msqid>: to kill any queue that is not being used

Sample Output for : ipcs -q <msquid>

------ Message Queues --------

key	msqid	owner	perms	used-bytes	messages
0x00000fdd	131072	xyz	666	0	0

Sample Output for Receiver and Sender:
Sender:
$ ./packet_sender 2
Waiting for receiver pid.
Got pid : 31746
==========
Sending Message: 1
Number of packets in current message: 3
Sending packet: bbb
Sending packet: ccc

Sending packet: aaa
==========
Sending Message: 2
Number of packets in current message: 4
Sending packet: aaa
Sending packet: ddd
Sending packet: ccc
Sending packet: bbb

Receiver:
./packet_receiver 2
Sending pid: 31746
GOT IT: message=aaabbbccc
GOT IT: message=aaabbbcccddd

OPTIONAL:
Handle `SIGINT` in `receiver` and `sender` to clean up the resources (especially the message queue) before exiting. You will have to handle errors for the other process if you remove the message queue from one process. For example, if the `receiver` is killed and removes the queue then `sender` will start failing as it is not able to write to the queue. Handle this as gracefully as possible.

3. Deliverables

1. Files containing your code
2. A README file
3. A makefile that will compile your code and produce the proper binary executables.

All files should be submitted using the Moodle. This is your official submission that we will be graded. Please do not send your deliverables to the TAs. Also, the future submissions under the same homework title overwrite the previous submissions. Only the most recent submission is graded.

4. Documentation

You must include a **README file** which describes your program. It must contain:

1. How to compile your program
2. How to use the program from the shell (syntax)
3. What exactly your program does (briefly)
4. Work Distribution (Important)

The README file should not be too long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion after reading the README. Within your code you should write appropriate comments, although you don't need to comment every line of your code.

At the top of your README file and main C source file please include the following comment:

/* CSCI4061 F2016 Assignment 3
*** login: cselabs login name (login used to submit)**
*** date: mm/dd/yy**
*** section: <one digit number>**
*** name: full name1, full name2, full name3 (for partner(s))**
*** id: id for first name, id for second name, id for third name */**

Grading:

5% README file – tell us who did what
20% Documentation within code, coding and style
(indentations, readability of code, use of defined constants rather than numbers)
75% Test cases

1. Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.

2. Please make sure that your program works on the CSELabs machines e.g., KH 4-250 (cselkh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

3. We will use the GCC version installed on the CSELabs machines(i.e. 5.4.0) to compile your code. Make sure your code compiles and run on CSELabs.
4. You need to make your own test case as test cases. Thus, please make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification.