Project 3

- Due Dec 8 by 11:59pm
- Points 18

CMPSC 473 Project 3: Virtual Memory Manager

(due Dec 8 11:59PM thru Github - NO EXTENSIONS WILL BE GIVEN!)

This page will continue to be updated. Please check back periodically for updates and announcements.

Please direct all your project-related questions/clarifications to the TAs through Canvas Email or Discussions.

Academic Integrity Statement:

The University defines academic integrity as the pursuit of scholarly activity in an open, honest and responsible manner. All students should act with personal integrity, respect other students' dignity, rights and property, and help create and maintain an environment in which all can succeed through the fruits of their efforts (refer to Senate Policy 49-20. Dishonesty of any kind will not be tolerated in this course. Dishonesty includes, but is not limited to, cheating, plagiarizing, fabricating information or citations, facilitating acts of academic dishonesty by others, having unauthorized possession of examinations, submitting work of another person or work previously used without informing the instructor, or tampering with the academic work of other students. Students who are found to be dishonest will receive academic sanctions and will be reported to the University's Office of Student Conduct for possible further disciplinary sanctions (refer to Senate Policy G-9). The Academic Integrity policy for the Department of Computer Science and Engineering for programming projects can be explicitly found at ***EECS Academic Integrity Policy*** ***(https://www.eecs.psu.edu/students/resources/EECS-CSE-Academic-Integrity.aspx)*** *. Please go through it carefully and do ask the instructor for any clarifications if something is not clear.*

There should be no exchange/dissemination of code, or collaborative working on the projects across teams! Any sharing of code (both the giver and the taker), or using code from sites on the Internet, use of AI Tools, and Tutoring services/sites, or consulting others (except the instructor and TAs) to do your projects, will be considered as academic dishonesty, leading to serious consequences. Ensure that your code is NOT visible to anyone outside your team. For any violation, you will be given a score of 0 for that project apart from being reported to the concerned authorities! A repeat violation will have more serious consequences including the assignment of a "F" grade for the course.

Note that all team members (when working in a team) will be held equally accountable and responsible for the integrity of the entire project, even if you have worked on only a part.

Summary

In this project, you will learn about virtual/physical memory and signals. This project requires you to implement a virtual memory manager which performs **paging**.

Deadline

December 8, 2025, 11:59 PM

Goals

- Learn how to manage the virtual memory
- Figure out how signals and signal handlers work in Linux
- Properly implement paging based on page replacement policies

Overview

You will implement a library (libmemmanager) to manage virtual memory on limited physical memory by using an access control mechanism and two page replacement policies. The virtual address space (size) and the number of physical pages is given to you via the *mm_init()* function. Based on the input file, the program will access (read/write) the virtual address space. When accesses are made, (i) you need to figure out whether the corresponding virtual page is in physical memory, (ii) if it is, allow the access to go through, (iii) else, allocate a frame for this virtual page in physical memory (and consequently evict some other page as per the page replacement policy if the physical memory is full). You need to fault the access if the page is not in physical memory and handle the page faults properly.

Hardware Model

This machine is a single CPU system. Only one thread will read/write to the virtual memory.

Memory Protection and SIGSEGV

As explained above, the program will read/write to the given virtual memory range, based on the input file using normal loads/stores instructions available in the hardware as in a normal program. The program will NOT CALL ANY EXPLICIT FUNCTION (like read(...) or write(...)) to do these operations. Below is an example:

Read	Write

`mm_init(policy, vm_ptr, vm_size, NUM_FRAMES, PAGE_SIZE, stats);`
`...`
`...`
`int read_value = vm_ptr[read_page_number*PAGE_SIZE];`
	`mm_init(policy, vm_ptr, vm_size, NUM_FRAMES, PAGE_SIZE, stats);`
	`...`
	`...`
	`vm_ptr[write_page_number*PAGE_SIZE] = write_value;`

On the left, you can see that a value is read (from *vm_ptr* region) and stored into *read_value*. On the right, you can see that the *write_value* is written (to the *vm_ptr* region).
Again, it is not calling functions like read(...)/write(...). Instead, it is just reading/writing to a virtual address in C.

Q. How will you *know* when a virtual page is read/written if no explicit function calls are made?
HINT. You can protect (e.g., make it inaccessible, read-only, read/write) a given virtual memory range by using *mprotect()*.
If you set *mprotect()* on the *vm_ptr* region properly, the Linux kernel will generate the SIGSEGV signal for the program, according to your *mprotect()* access flags. When a signal is generated, the signal handler is executed. The default SIGSEGV handler will normally kill the program. However, you can register your own signal handler by using *sigaction()*. Inside your signal handler, you can emulate your page replacement algorithm.

Initially, all virtual pages should have read/write permissions turned off, so that you catch any access to any of those pages. Once you decide to bring in a page, you will *mprotect()* it with appropriate permissions based on the type of access. You have to determine this first in your segfault handler. Subsequently, you can give it read-only permission if it is read access, and read-write permission if it is a write access. (This is not always the case. Check *Page Replacement Policies*.) When you evict a virtual page, you will *mprotect()* it with none, so that any future access to it will raise a SIGSEGV, and you will record it as a page fault.

IMPORTANT: Your implementation should use the minimal number of SIGSEGVs to implement the required page replacement functionality. You should NOT keep faulting on every page and every time! Else, there will be a penalty when grading.

It is very important to read and understand the *mprotect()* and *sigaction()* manual, which can be found in the Linux manual (**https://man7.org/linux/man-pages/ (https://man7.org/linux/man-pages/index.html)**)

Page Replacement Policies

If the physical frames are full, you need to evict and replace a physical frame. As discussed in class, there are multiple ways to do this. In this project, you will implement two replacement policies, explained below.

FIFO (First-In-First-Out) Replacement:

- Evict the oldest virtual page (among the currently resident pages in physical memory) brought in to the physical memory. You will have to protect the pages that are NOT in physical memory to catch accesses to these page and record page faults.

Third Chance Replacement:

- Maintain a circular linked list, with the head pointer pointing to the next potential candidate page for eviction (as discussed in class)
- Maintain a reference bit and a modified bit for each page in physical memory
- When looking for the next candidate page for eviction, if by any chance, the page pointed by the head pointer has its reference bit on (=1), the head pointer resets this bit (=0), and moves to the next element in the circular list and retries. When the head finds a candidate with a zero reference bit, it becomes a candidate for replacement as per the second chance page replacement algorithm.
- **However**, in our third chance algorithm, you will need to give such a page a third chance if it has been modified (modified bit=1) since we assume pages requiring write-back will incur higher overheads for replacement.
- Example:
  A physical page under the clock head could be in one of the following 3 states: (a) R=0, M=0; (b) R=1, M=0; or (c) R=1, M=1
  - In case (a), the page can be immediately evicted.
  - In case (b), you should give it a second chance, i.e. reset the R bit, and if the next time the clock head comes to that page its bits indicate state (a), then replace it.
  - In case (c), you should give it a third chance, i.e. reset the R bit in the 1st pass, and even in the 2nd pass that the head comes to that page, you should skip it. Caution! You may be tempted to reset the M bit. However, if you do that, note that you will not know whether this page needs to be written back to disk (write_back parameter) later on at the time of replacement. Only the third time, should it be replaced (and written back). However, note it is possible that between the 2nd pass and the 3rd pass, the R bit could again change to 1 in which case it will again get skipped in the 3rd and 4th pass and would get evicted only in the 5th pass (as long as there is no further reference before then).

Notes/Hints on Third Chance Replacement:

- The only way for you to emulate this is by taking SIGSEGV signals appropriately.
  (For example, assume the page is read-only, and you set that page to *mprotect(..., PROT_READ)*. If the program reads that page, you will NOT get a

SIGSEGV signal (and therefore the signal handler will not execute) since it does NOT violate the access flag set by *mprotect()*. Consequently, you can't set the reference bit even though it is a read access.)

- Therefore, for a page in physical memory for which the reference bit is 0, you may still require to *mprotect()* to take a fault on either a read or write to that page. In such a case, a read will turn on the reference bit, and a write will turn on both the reference and modified bits. For a page in physical memory for which the reference bit is on, you would need to *mprotect()* it in read-only mode until the first write to it (to update the modified bit). Turning off reference bits would imply doing appropriate *mprotect()*s.
- Note: you should minimize the number of SIGSEGVs. Check *Additional Implementation Requirements/Restrictions* below.

APIs to Implement

In interface.h, the policy type is defined. You should implement the interface in a file called *interface.c*, whose semantic is described below.

```
// Policy type
enum policy_type
{
    MM_FIFO = 1,  // FIFO Replacement Policy
    MM_THIRD = 2, // Third Chance Replacement Policy
};
```

The above *enum* represents the policy type.

```
// APIs
void mm_init(enum policy_type policy, void *vm, int vm_size, int n_frames, int page_size, struct mm_stats *stats);
void mm_finish();
```

The mm_init() function initializes your memory manager. The stats pointer is the pointer you should pass when you call the mm_logger() function (below). Other arguments are self explanatory. *void *vm* points to the virtual memory where you will manage using your memory manager. The mm_finish() function is called once at the end to notify your memory manager that the program is done using the memory manager, and it should free all memory and data structures.

Your memory manager (libmemmanager) should call mm_logger() function (below), to log the events.

```
 /*
  * virt_page   : Page number of the virtual page being referenced
  * fault_type  : Indicates what caused SIGSEGV.
  *               0 - Read access to a non-present page
  *               1 - Write access to a non-present page
  *               2 - Write access to a currently Read-only page
  *               3 - Track a "read" reference to the page that has Read and/or Write permissions on.
  *               4 - Track a "write" reference to the page that has Read-Write permissions on.
  * evicted_page: Virtual page number that is evicted. (-1 in case of no eviction)
  * write_back  : 1 indicates evicted page needs writing back to disk, 0 otherwise
  * phy_addr    : Represents the physical address (frame_number concatenated with the offset)
  */
```

```
void mm_logger(struct mm_stats *stats, int virt_page, int fault_type, int evicted_page, int write_back, unsigned int phy_addr);
```

This function is already implemented in *logger.c*. Therefore, you do not have to implement it. Instead, you have to call this function from your memory manager accordingly. As you can see in *main.c*, the values you passed to this function eventually gets printed to the output file.
Note: *fault_type*=2 only requires a permission change, while 3 and 4 would not incur a fault in a *real hardware* but are needed in your emulation (memory manager) just to set the reference bit (note that if the reference bit was already one, a signal should NOT have been raised!)

You can implement additional functions (helper functions) and have additional variables and structures in *vmm.c/h*, but note that the *main.c* will call only *mm_init/finish()*. Your memory manager (SIGSEGV signal handler) should call *mm_logger()*.

Input File Format

An input to the program is a text file which has the following format:

```
Operation [Virtual page number] [Offset] [Value]
Operation [Virtual page number] [Offset] [Value]
Operation [Virtual page number] [Offset] [Value]
...
```

where,

- **Operation:** either **"read"** (for read operation) or **"write"** (for write operation)
- **Virtual page number:** the referenced virtual page number
- **Offset:** the offset within the referenced virtual page (in 4 bytes)

- **Value:**
  If the operation is **"read",** it is 0 (unused)
  If the operation is **"write",** this field will be the value to be written

Sample Input and Output

① Simple Case

```
read 1 8 0
write 2 16 12
read 0 8 0
read 3 8 0
write 4 16 12
```

can be interpreted as:
- Read virtual page 1, offset 8*4=32
- Write 12 to virtual page 2, offset 16*4=64
- Read virtual page 0, offset 8*4=32
- Read virtual page 3, offset 8*4=32
- Write 12 to virtual page 4, offset 16*4=64

Following is the MM_FIFO output of the above sample input file (assuming 4 frames):

MM_FIFO

```
Page Size: 4096
Num Frames: 4
type virt-page evicted-virt-page write-back phy-addr
0    1         -1                0          0x0020
1    2         -1                0          0x1040
0    0         -1                0          0x2020
0    3         -1                0          0x3020
1    4         1                 0          0x0040
```

- Since the frames are initially empty, the first read to page 1 will generate a fault (type=0). There will be no evicted virtual pages. The physical address of the access is 0*page_size+32=0x20.
- The write to page 2 will generate a fault (type=1). There will be no evicted virtual pages. The physical address of the access is 1*page_size+64=4160=0x1040.
- The read to page 0 will generate a fault (type=0). There will be no evicted virtual pages. The physical address of the access is 2*page_size+32=8224=0x2020.
- The read to page 3 will generate a fault (type=0). There will be no evicted virtual pages. The physical address of the access is 3*page_size+32=12320=0x3020.
- The write to page 4 will generate a fault (type=1). Since we have no more frames, we need to evict/replace! According to MM_FIFO policy, we select virtual page 1 (physical frame 0) to evict. Therefore, the physical address of this access is 0*page_size+64=0x40.
* NOTE: the above explanation is for illustrative purposes only. There are more types as explained in the API section. Also, see *Page Replacement Policies* for more details.

② Same input, different outputs

```
write 7 8 0
write 2 16 200
read 3 64 0
read 5 10 0
read 6 400 0
write 1 50 23
```

can be interpreted as:
- Write 0 to virtual page 7, offset 8*4=32
- Write 200 to virtual page 2, offset 16*4=64
- Read virtual page 3, offset 256
- Read virtual page 5, offset 40
- Read virtual page 6, offset 1600
- Write 23 to virtual page 1, offset 200

Following are the outputs of the above sample input file (assuming 3 frames):

MM_FIFO

```
Page Size: 4096
Num Frames: 3
type virt-page evicted-virt-page write-back phy-addr
1    7         -1                0          0x0020
1    2         -1                0          0x1040
0    3         -1                0          0x2100
0    5         7                 1          0x0028
0    6         2                 1          0x1640
1    1         3                 0          0x20c8
```

MM_THIRD

```
Page Size: 4096
Num Frames: 3
type virt-page evicted-virt-page write-back phy-addr
1    7         -1                0          0x0020
1    2         -1                0          0x1040
0    3         -1                0          0x2100
0    5         3                 0          0x2028
0    6         7                 1          0x0640
1    1         2                 1          0x10c8
```

- For MM_FIFO, the output is straightforward. Since we have 3 physical frames, the fourth access, which is the access to page 5 will evict/replace the virtual page 7 (physical frame 0), according to the FIFO policy.
- For MM_THIRD, when the fourth access happens, you will evict virtual page 3 (physical frame 2). According to the third chance policy, since virtual page 7 and 2 has a third chance, virtual page 3 will be evicted. There is no write-back, since the initial access to the virtual page 3 was a read access. The subsequent accesses to virtual page 6 and 1 will evict the pages 7 and 2 accordingly, but the write-back is 1. Since you wrote to page 6 and 1 initially, you should write-back.
*Note: The above explanation is for illustrative purposes only. Please follow the page replacement policy.

③ Different fault type

```
read 0 8 0
read 1 4 0
read 2 8 0
write 3 4 5
write 0 4 12
write 1 8 12
write 2 4 12
write 4 4 8
write 1 8 12
read 2 4 0
```

can be interpreted as:
- Read to virtual page 0, offset 32
- Read to virtual page 1, offset 16
- ... and so forth

Following are the outputs of the above sample input file (assuming 4 frames):

MM_FIFO

```
Page Size: 4096
Num Frames: 4
type virt-page evicted-virt-page write-back phy-addr
0     0       -1                0          0x0020
0     1       -1                0          0x1010
0     2       -1                0          0x2020
1     3       -1                0          0x3010
2     0       -1                0          0x0010
2     1       -1                0          0x1020
2     2       -1                0          0x2010
1     4       0                 1          0x0010
```

MM_THIRD

```
Page Size: 4096
Num Frames: 4
type virt-page evicted-virt-page write-back phy-addr
0    0         -1                0          0x0020
0    1         -1                0          0x1010
0    2         -1                0          0x2020
1    3         -1                0          0x3010
2    0         -1                0          0x0010
2    1         -1                0          0x1020
2    2         -1                0          0x2010
1    4         0                 1          0x0010
4    1         -1                0          0x1020
3    2         -1                0          0x2010
```

You can now see that the number of lines of the output file is also different.
- For MM_FIFO, the output is straightforward. Note the *type=2*. Virtual pages 0, 1, 2 were initially read-only. Therefore, write accesses to virtual pages 0, 1, 2 will trigger a fault, and the result is logged as *type=2*. (Check the *API* description section.) The 8th access will trigger a replacement, which will evict virtual page 0. After this replacement, subsequent accesses to page 1 and 2 doesn't fault, and therefore there is no output.
- For MM_THIRD, even though the last two accesses (virtual page 1 and 2) had enough permissions, the signal handler was executed because we needed to mark the reference/modified bit(s), according to the third chance replacement policy. The *type* is also logged accordingly.
*Note: The above explanation is for illustrative purposes only. Please follow the page replacement policy.

④ Input file number of operations != output file number of logs

```
write 1 400 0
read 1 400 0
read 1 3 0
write 1 1009 0
read 1 333 0
read 8 899 0
write 8 234 17
read 5 1007 0
read 7 234 0
write 1 501 22
read 8 778 0
read 1 540 0
```

- The interpretation is straightforward now.

Following are the outputs of the above sample input (check Num Frames):

MM_FIFO

```
Page Size: 4096
Num Frames: 3
type virt-page evicted-virt-page write-back phy-addr
1    1         -1                0          0x0640
0    8         -1                0          0x1e0c
2    8         -1                0          0x13a8
0    5         -1                0          0x2fbc
0    7         1                 1          0x03a8
1    1         8                 1          0x17d4
0    8         5                 0          0x2c28
```

MM_THIRD

```
Page Size: 4096
Num Frames: 3
type virt-page evicted-virt-page write-back phy-addr
1    1         -1                0          0x0640
0    8         -1                0          0x1e0c
2    8         -1                0          0x13a8
0    5         -1                0          0x2fbc
0    7         5                 0          0x23a8
4    1         -1                0          0x07d4
3    8         -1                0          0x1c28
```

- The first access to virtual page 1 will trigger a fault. You can see that subsequent virtual page 1 accesses don't fault for both replacement policies. Therefore, the input file number of operations != the output file number of logs. Also, note that the number of logs may differ between MM_FIFO and MM_THIRD (check case ③ again).
- The first access to virtual page 8 (which is the 6th access overall) generates a fault (*type=0*). However, the subsequent second access to virtual page 8 (which is the 7th access overall) generates a fault again, but this time, it is *type=2*. Go over the API section to check what this means.
- For MM_FIFO, the first evicted virtual page is virtual page 1, according to the FIFO policy.
- For MM_THIRD, the first evicted virtual page is virtual page 5, according to the third chance policy.

Additional Implementation Requirements/Restrictions

- You CANNOT use any time/timer functionality provided by the operating system (e.g., C time(), sleep(), jiffies, pthread_cond_timedwait(), timespec, clock_gettime() etc.).
- You CANNOT use outside libraries.
- Your program must cleanly exit without errors within a timeout. If your program executes longer than this timeout, it is assumed to have some bug (and therefore failed the given test case).
- The page size will be the page size of the machine/operating system, which is 4096 bytes (check *main.c* file to learn how it is obtained).
- You should minimize the number of SIGSEGVs raised/handled inside your memory manager. Therefore, you can NOT simply set *mprotect(..., PROT_NONE)* for every virtual page and raise a SIGSEGV every access, which will execute your signal handler on every access. Instead, set the correct *mprotect()* of the pages so the SIGSEGVs are minimized for the page replacement algorithm.

SIGSEGV Handler Implementation Hints

The signal handler that you write to catch the SIGSEGVs before implementing the page replacement, must first determine:
(i) the address of the access that raised the fault, and
(ii) whether the fault raised originated from a read operation or a write operation.

If you assign your signal handler via *(*sa_handler)(int)*, you can NOT get the signal/context information inside your handler. Therefore, you should specify SA_SIGINFO in *sa_flags* and assign your signal handler via *(*sa_sigaction)(int, siginfo_t *, void *)*. From the Linux *sigaction()* manual:

```
If SA_SIGINFO is specified in sa_flags, then sa_sigaction
       (instead of sa_handler) specifies the signal-handling function
       for signum.  This function receives three arguments, ...
```

The following is a brief summary of the manual (you should also read the manual):
The signal handler registered via *sigaction()* - *sa_sigaction* has three parameters passed to it. The second argument (*siginfo_t *info*) and the third argument (*void *ucontext*) will point to the information about the signal and context of the process where the signal was raised.

```
void handler(int sig, siginfo_t *info, void *ucontext) {
... // Your signal handler implementation
}
```

- The *siginfo_t* structure contains information necessary to get the address where the fault was raised.
- The *ucontext* contains the dump of the CPU register state when the fault (signal) was raised. Accessing a specific location within the *ucontext* structure should help you determine whether the fault was raised due to a read or a write operation. Therefore, (1) inspect `ucontext_t` and `mcontext_t` (link below), and (2) locate the general registers. Among the general registers, (3) locate the error register. x86 architectures export the error information onto the error register (REG_ERR) upon a page-fault with specific error codes. Checking the error codes will let you know whether the fault was raised due to a read or a write operation.

I. The *siginfo_t* is additionally explained in the sigaction() Linux manual.
Link: **Linux manual: sigaction** **(https://man7.org/linux/man-pages/man2/sigaction.2.html)**

II. **(https://man7.org/linux/man-pages/man2/sigaction.2.html)** The *ucontext_t* is additionally explained (defined) in glibc (GNU C Library), which is the /usr/include/sys/ucontext.h file.
Link: **glibc: ucontext.h** **(https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/x86/sys/ucontext.h;h=7a3938cbcdaa348027e7cddf4e3540b1a1d8c187;hb=c758a6861537815c759cba2018a3b1a**

III. **(https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/x86/sys/ucontext.h;h=7a3938cbcdaa348027e7cddf4e3540b1a1d8c187;hb=c758a6861537815c759cba2018a3b1a**
Error register information (error codes) are defined and explained in the error code enum in Linux kernel code.
Link: **Linux: trap_pf.h** **(https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/trap_pf.h)**

*Note: Since you assigned your own personal SIGSEGV signal handler, the normal segmentation faults (from malloc()ed memory error, invalid accesses, etc.) will also execute *your* signal handler, instead of the *default* SIGSEGV signal handler (which normally kills the program and prints *Segmentation fault (core dumped)* to the terminal). Therefore, if you make a memory access error on your part, the program may not be killed but just hang (or do actions based on your SIGSEGV signal handler). It is hard to debug these cases. Please use *gdb*, set breakpoints, and try to go over line by line.

Do's and Don'ts

- Fully (re)reading/understanding this document, the template code, understanding textbooks/slides, and reading the appropriate manuals are part of this project.
- Try to go through *main.c* and understand what it does. You can see that `./tester <allocation-policy> <num-frames> <input-filename>` is how you run the program.
- Check how and what is logged to the output file.
- Go through the lecture slides and the textbook again and understand how virtual memory and paging work.
- Check the *sample_input* and *sample_output*. Go over the cases and understand the output.
- You are STRONGLY SUGGESTED to first write a small, simple, separate standalone program which will generate SIGSEGV fault, and write a signal handler to catch this fault. Additionally, investigate how to determine if the raised fault is from a read/write to a page. Ensure that you understand how to write signal handlers and leverage *mprotect()* before proceeding. Check the *mprotect()* Linux manual.
- Implement the MM_FIFO replacement policy first.
- Note that your memory manager must work with varying number of physical frames (passed on via *mm_init()*).
- Create your own corner-case inputs to test your virtual memory manager.
- A page usually means a *virtual* page, and a frame usually means a *physical* frame. However, sometimes they are used interchangeably; some text/textbooks/documents may use the term physical page, which means physical frame. It is important to understand the context.

Teamwork and Submission Instructions

- You can work in teams (at most 2 members per team), or individually, for the project. You are free to choose your partner and it does not need to be the same as in earlier projects. But if either of you choose to drop out of your team for any reason at any time, each of you will be individually responsible for implementing and demonstrating the entire project and writing the project report by the designated deadline. (Please notify the TAs if this happens.)
- Even though you will work in pairs, each of you should be fully familiar with the entire code and be prepared to answer a range of questions related to the entire project. It will be a good idea to work together at least during the initial design of the code. You should also ensure that there is a fair division of work between the members. Remember that you both are owning up to the entire project that is being submitted, and you both are owning up to what works, what doesn't and the integrity of the entire project (not just your piece)!
- Feel free to change teams from the prior projects. (It does not need to be the same from Project 1 or 2.)
- One team member should create a team using the following invitation. The other team member should join the team (if you would like to work in teams).
Github invitation link: **https://classroom.github.com/a/k7VGyOGj** **(https://classroom.github.com/a/k7VGyOGj)**
- All the code-bases will be tested on the W135 Linux Lab servers located in Westgate. Regardless of where/how you develop/debug your code (laptop linux, Linux VMs, etc.), you need to ensure you test and run them in W135 servers for being graded!
- Additional announcements would be sent out in the future to get your team information and the GitHub repository. Stay tuned!

Submission Guidelines

- Ensure you commit your code often using the `git commit` and the `git push` commands (similar to how you had done P0~P2).
- You should write a brief report (`report.pdf` of around 2 pages) explaining the design of your memory allocator, any implementation choices/restrictions, any functions not implemented, things learned, distribution of the workload within the team, etc.
- The project (your final memory allocator implementation, including the `report.pdf`) must be pushed to the repository before the posted deadline. The last commit and push that you do prior to the deadline will be taken as your final submission.
- You may make multiple branches, but make sure you push your code into the main branch before the posted deadline. We will only clone from the main branch.
- Double-check and see if your submission is properly pushed by accessing your repository in GitHub.com
- NO EXTENSIONS WILL BE ENTERTAINED. ENSURE YOU PUSH YOUR CODE AND REPORT TO THE GITHUB REPO BEFORE THE DEADLINE.

Project Grading

- We will clone the last push you made before the deadline, on the W135 server. We will type *make* to compile your program. We will not fix any compile errors.
- We have additional surprise test cases (they may be large, max. 1M requests) and use them as input to your virtual memory manager.
- We will change the *num_frames* to different values (in addition to the *sample_output*).
- After your virtual memory manager produces all outputs, we will use the *diff* command to see any differences from our output. (i.e., Your output must match our results.)
- We will run the same test cases multiple times and see if you get the same result. This is to check if there are any bugs in the code. There is a timeout of 3 seconds for each execution for the sample test cases. If your allocator runs longer than that, we will assume it has a bug, and the corresponding test case will fail. (The timeout will increase accordingly for longer/larger surprise test cases.)
- Count how many SIGSEGV signals are raised. Check *Additional Implementation Requirements* section.
- Read the *report.pdf* and see how your team attempted/implemented the virtual memory manager.
- Check if *main.c* or *Makefile* was changed or not. Do NOT change the *main.c* file and the top-level *Makefile* file.
- Check if you have followed the project descriptions, and check if you have used any restricted functions such as time. (Check *Additional Implementation Requirements* section)
- Have a short meeting with your team to demonstrate and ask about the project, the code, the project report, and the functionality. For example, we may ask you to find, explain, and/or modify portions of code in the meeting.

Questions

For questions, please use the 'Discussions' tab of Canvas or use office hours. Please direct all project-related questions to the TAs. Please do NOT copy/paste code and post it in Discussions.