Computer Systems and Networks

Prof. Forsyth

Project 3 - Virtual Memory

Due: **October 21st 2025**

1 Introduction

Read the entire document before starting. There are critical pieces of information and hints along the way.

In this project, you will be implementing part of a virtual memory system simulator. You have been given a simulator which is missing some critical parts. You will be responsible for implementing these parts. Detailed instructions are in the files to guide you along the way. If you are having trouble, we **strongly suggest** that you take the time to read about the material from the book and class notes.

Warning: In past semesters, this project has caused students a lot of trouble due to the sheer amount of concepts to understand. It is important that you not just understand the concept of virtual memory, but also have a good understanding of C and how to use GDB for debugging. So, start early. If you are struggling with writing the code, then step back and review the concepts. Be sure to start early, ask Piazza questions, and visit us in office hours for extra help!

This project is divided into 10 problems. The files that you will be modifying are the following:

- `address_splitting.h` - Break down a virtual address into its components.
- `mmu.c` - Initialize any system- and memory access-related bookkeeping.
- `proc.c` - Initialize any process-related bookkeeping.
- `page_fault.c` - Implement the page fault handler.
- `page_replacement.c` - Implement the frame eviction algorithms FIFO and Approximate LRU.
- `stats.c` - Calculate the Average Memory Access Time (AMAT)

It will be a good idea to peek into following the files:

- `mmu.h` - Defines the structures used by mmu.c
- `proc.h` - Defines the structures used by proc.c
- `pagesim.h` - Defines simulation parameters as well as global structures.
- `pagesim.c` - Reads a trace file of memory operations and calls each operation's corresponding function implemented in `proc.c`.
- `stats.h` - Defines parameters that can be used when calculating AMAT.
- `swap.c` - Initializes functions to support a queue that are used in `swapops.c`
- `swapops.c` - Initializes functions to keep track of the frames swapped out to physical memory. Discussed in section 8
- `types.h` - Defines different types that are used throughout the simulation
- `util.c` - Initializes a random function used for the random replacement algorithm

2 Memory Organization Background

The simulator simulates a system with 20-bit **byte-addressable** of physical memory (20-bit physical address space). Throughout the simulator, you can access physical memory through the global variable `uint8_t mem[]` (an array of bytes called "mem"). You have access to, and will manage, the entirety of physical memory.

The system has a 24-bit virtual address space and memory is divided into 16KB pages.

Like a real computer, your page tables and paging data structures live in physical memory too! Conveniently, both a page table and the frame table fit in a single page frame in memory and so you'll want to dedicate some

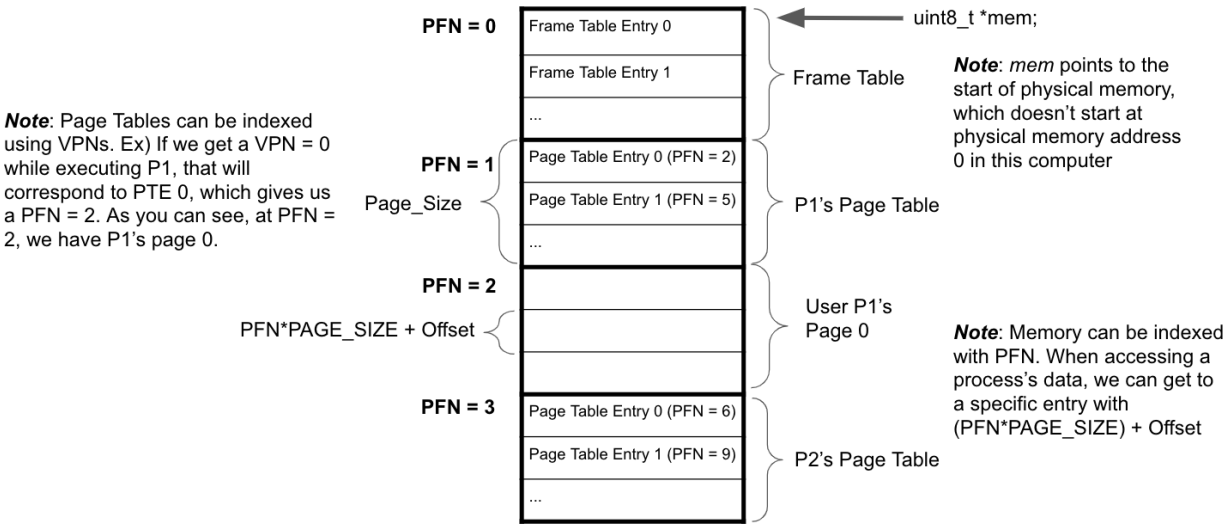

Figure 1: Organization of physical memory showing frames and frame entries.

page frames for storing this data. **You are responsible for placing and initializing these structures in memory.**

Note: Since user data page frames and operating system page frames such as the frame table and page tables coexist in the same physical memory, we must have some way to differentiate between the two, and keep user pages from replacing system pages.

For this project, we will take a simple approach: Every page frame has a "protected" bit in its frame table entry, which is set to "1" for system frames and "0" for user frames. In other words, we'll set the protected bit to 1 for the frames holding paging system meta-data and 0 for the page frames holding user data pages.

3 Address Splitting

In most modern operating systems, user programs access memory using virtual addresses. The hardware and the operating system work together to turn the virtual address into a physical address, which can then be used to address into physical memory. The first step of this process is to translate the virtual address into two parts: the higher order bits for the VPN, and the lower bits for the page offset.

Implement the `vaddr_vpn` and `vaddr_offset` functions in `address_splitting`.

These will be used to split a virtual address into its corresponding page number and page offset. You will need to use the parameters for the virtual memory system defined in `pagesim.h` (`PAGE_SIZE`, `MEM_SIZE`, etc.).

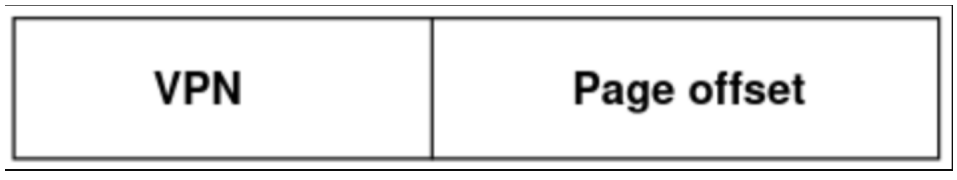

Figure 2: Virtual addresses containing VPN and offset

4 Initialization

When the simulation starts, it will need to first set up the frame table (sometimes known as a "reverse lookup table").

Implement the `system_init()` function in `mmu.c`.

For simplicity, we always place the frame table in physical frame 0. You need to initialize the `frame_table` pointer to the start of this frame. Don't forget to mark the first entry (which corresponds to the frame holding the frame table) of the frame table as "protected" because we will **never evict the frame table**. Then, during your page replacement, you will need to make sure that you never choose a protected frame as your victim.

After setting up the frame table, we will need to set up a page table at the start of each process.

Implement the `proc_init()` function in `proc.c`.

Since processes can start and stop any time during your computer's lifetime, we must be a little more sophisticated in choosing which frames to place their page tables. For now, we won't worry about the logistics of choosing a frame–just call the `free_frame` function you'll write later in `page_replacement.c`. Then make the appropriate flags for that frame table entry. (HINT: Do we ever want to evict the frame containing the page table while the process is running?) After picking the frame for a process's page table, remember to update the ptbr in the process pcb struct.

You may add any global variables or helper functions you deem necessary.

Each frame contains `PAGE_SIZE` bytes of data, therefore to access the start of the i-th frame in memory, you can use `mem + (i * PAGE_SIZE)`.

5 Context Switches and the Page Table Base Register

As you know, every process has its own page table. When the processor needs to perform a page lookup, it must know which page table to look in. This is where the *page table base register* (PTBR) comes in.

In the simulator, you can access the page table base register through the global variable `pfn_t PTBR`.

Implement the `context_switch` function in `proc.c`.

Your job is to update the PTBR to refer to the new process's page table. This function will be very simple.

Going forward, pay close attention to the type of the PTBR. The PTBR holds a physical frame number (PFN), not a virtual address. Think about why this must be.

6 Reading and Writing Memory

The ability to allocate physical frames is useless if we cannot read or write to them. **In this section, you will add functionality to the simulator to allow it to make read and write memory references on behalf of the simulated processes.** The simulator will use pre-recorded lists of memory references captured (traced) from the execution of real processes to simulate the memory references in each process. Because processes operate on a virtual memory space, it is necessary to first translate a virtual address supplied by a process into its corresponding physical address, which then will be used to access the location in physical memory. This is accomplished using the page table, which contains all of a process's mappings from virtual addresses to physical addresses.

As previously mentioned in Section 3, when running a user process, all addresses from the CPU are virtual and must be translated. Do note that for this project, we assume that when the operating system is running (i.e. the CPU is in system mode), address translation is disabled and all memory addresses referenced by the CPU will be treated as physical addresses. This is why the operating system itself has no page table!

Implement the `mem_access` function in `mmu.c`.

You will need to use the passed-in virtual address to find the correct page table entry and the offset within the corresponding page. **HINT:** Use the virtual address splitting functions that you wrote earlier in the project.

Once you have identified the correct page table entry, you must use this to find the corresponding physical frame and its physical address in memory, and then perform the read or write at the proper location within the page. (Remember that the simulator's physical memory is represented by the `mem` array and its subscripts are the physical memory addresses).

Keep in mind that not all entries in a process's page table have necessarily been mapped. Entries not yet mapped are marked as invalid, and an attempt to access an invalid address should generate a page fault. You will write the `page_fault()` function in the next section, so for now just assume that it has successfully allocated a page for that address after it returns.

After we are sure the correct page is mapped in for the requested virtual address, we want to make sure the frame is marked as referenced in the frame table. This is used later by the page replacement algorithms.

Make sure to mark the containing page as "dirty" in the process's page table on a write. These bits will be used later when deciding on what pages should be evicted first, and if an evicted page needs to be written to the disk to preserve its content.

7 Eviction and Replacement

Recall that when a CPU encounters an invalid VPN to PFN mapping in the page table, the OS allocates a new frame for the page by either finding an empty frame or evicting a page from a frame that is in use. In this section, you will be implementing a page fault and replacement mechanism.

Implement the function `page_fault()` in `page_fault.c`. A page fault occurs when the CPU attempts to translate a virtual address, but finds no valid entry in the page table for the VPN.

Our page fault handler will provide the CPU with a valid frame filled either with its previously evicted data or with zeros (meaning it was never written to before).

To resolve the page fault, you must do the following:

1. Get the page table entry for the faulting virtual address.
2. Using `free_frame()` get the PFN of a new frame.
3. Check to see if there is an existing swap entry for the faulting page table entry. **More on swap in section 8**.
4. If a swap entry exists, then use `swap_read()` to read in the saved frame into the new frame, otherwise clear the new frame.
5. Update the mapping from VPN to the new PFN in the current process' page table.
6. Update the flags in the corresponding frame table entry.

Next, we will turn our attention to the eviction process in `page_replacement.c`.

When we allocate frames for our pages, they needed to have come from somewhere. Throughout the whole project, you have been using `free_frame()` without implementing it. Now, it is finally time to implement `free_frame()`.

Implement `free_frame()` in `page_replacement.c`.

`free_frame()` will return the frame selected from `select_victim_frame()` (discussed in section 10). If the victim frame is already mapped, you will effectively evict it. This would mean updating both the page and frame table entries with the appropriate flags (recall what `mem_access()` looked for to check for evicted

pages). If the evicted frame is dirty, then you will need to `swap_write()` (discussed in section 8) it into the swap space and clear the dirty bit.

8 Swap Space

If the evicted page is dirty, you will need to write its contents to disk. **The area of the disk that stores the evicted pages is called the swap space.** Swap space effectively extends the main memory (RAM) of your system. If physical memory is full, the operating system kicks some frames to the hard disk to accommodate others. When the "swapped" frames are needed again, they are restored from the disk into physical memory. Without the swapping mechanism, when the system runs out of RAM and we start evicting physical frames, we lose the data stored in these frames, and the process whose pages were originally mapped to the evicted frames loses its data forever. Therefore, upon selecting a victim, we need to make sure that its data is swapped out to disk and restored when needed.

Recall that during your page fault handler, you must fill the newly provided frame with any data that was previously written to this page that was evicted to the swap space. To do this, we have provided the method `swap_exists()` that checks if the faulting page was swapped out to disk previously. If it has, then you need to restore it, using `swap_read()` If the faulting page has not been swapped previously, then you need to zero out the freed frame to prevent the current process from potentially reading the memory of some other process.

To write the contents of a victim's page to disk, we provide a method called `swap_write()`, where you can pass in a pointer to the victim's page table entry and a pointer to the frame in memory. This will simulate swapping the page to disk.

9 Finishing a Process

If a process finishes, we don't want it to hold onto any of the frames that it was using. We should unmap any frames so that other processes can use them (remember not to unmap pages that are now being used by other processes). Also: If the process is no longer executing, can we release the page table?

As part of cleaning up a process, you will need to also free any swap entries that have been mapped to pages. You can use `swap_free()` to accomplish this.

Implement the function `proc_cleanup()` in `proc.c`.

10 Better Victim Selection

In section 7, we relied on the `select_victim_frame()` function to tell us which frame to choose as our "victim".

We have provided you with a default, inefficient page replacement algorithm that randomly selects a page to be evicted. The simulator can run this replacement strategy out-of-the-box so that you can test the other parts of your code without having to write a page replacement algorithm. Run the simulator with `-rrandom` to use the random algorithm.

10.1 FIFO Algorithm

Of course, we can do better than random replacement. Implement the **First In First Out** algorithm. When looking for a victim, we simply choose the oldest page that was entered in. Look at section 8.3.3 in the textbook for more information on this. You can use the global `oldest_pfn` to keep track of this.

Once you have implemented the FIFO algorithm, you will be able to run the simulator with the `-rfifo` argument to use the algorithm as your page replacement strategy.

Remember again that if the protected bit is set, it should never be chosen as a victim frame.

10.2 Approximate LRU Algorithm

After implementing the FIFO algorithm, implement the **Approximate Least Recently Used (LRU)** algorithm. Your approximate LRU algorithm should choose the least recently mapped frame table entry. Familiarize yourself with textbook section 8.3.4.2 (Approximate LRU #2) before attempting this part. For the purposes of this project, we have provided a simulator "daemon" that wakes up every few memory accesses and calls `daemon_update()`. **Make sure you implement this method before writing the algorithm.** A real daemon would wake up on a time-based schedule, however, this implementation creates more consistent results for the autograder.

Hint: Remember what the bitwise operators do and how they are used.

Once you have implemented the Approximate LRU algorithm, you will be able to run the simulator with the `-rlru` argument to use the algorithm as your page replacement strategy.

After writing your stats function in section 11, compare the performance of the three algorithms. What do you observe?

Remember again that if the protected bit is set, it should never be chosen as a victim frame.

11 Computing AMAT

In the **final section** of this project, you will be computing some statistics.

- `accesses` - The total number of accesses to the memory system
- `page_faults` - Accesses that resulted in a page fault
- `writebacks` - How many times you wrote to disk
- `AMAT` - The average memory access time of the memory system

We will give you some numbers that are necessary to calculate the AMAT:

- `MEMORY_ACCESS_TIME` - The time taken to access memory **SET BY SIMULATOR**
- `DISK_PAGE_READ_TIME` - The time taken to read a page from the disk **SET BY SIMULATOR**
- `DISK_PAGE_WRITE_TIME` - The time taken to write to disk **SET BY SIMULATOR**

You will need to implement the `compute_stats()` function in `stats.c`

12 Simulator Process Diagram

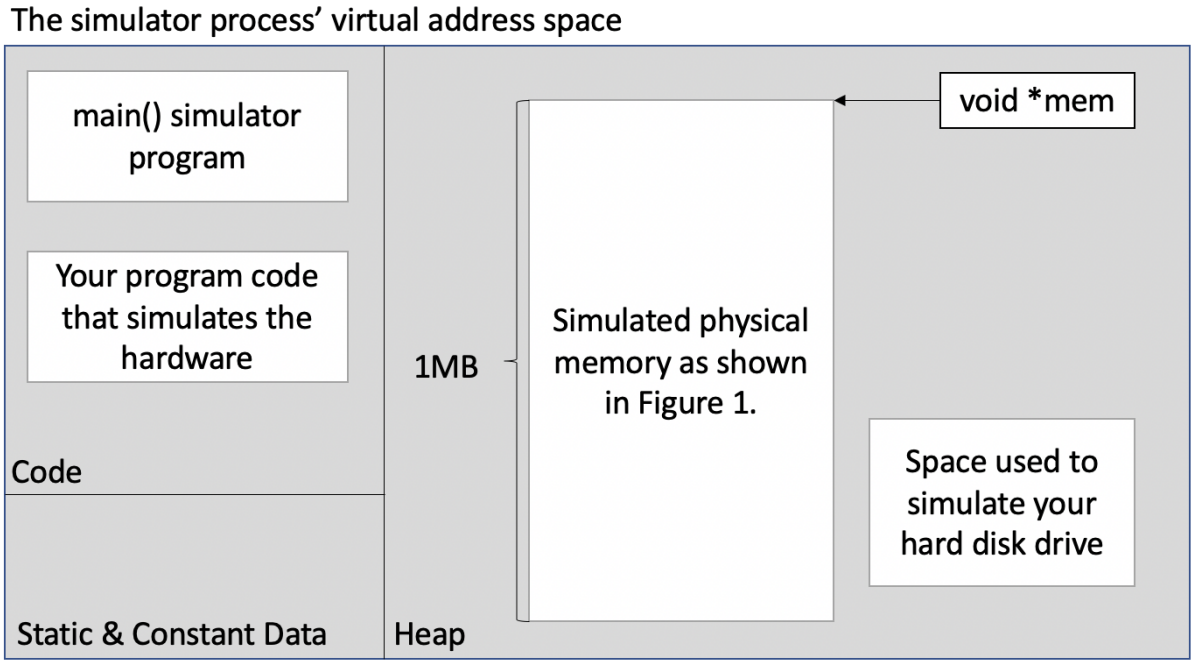

Figure 3: This diagram gives a general overview of how the simulator works.

13 How to Run / Debug Your Code

13.1 Environment

Your code will need to compile under Ubuntu 20.04 LTS. You can develop on whatever environment you prefer, so long as your code also works in Ubuntu 20.04 (which we will use to grade your projects). **Non-compiling solutions will receive a 0. Make sure your code compiles with no warnings.** We highly recommend using the setup available on Canvas. If you are having trouble with this, ask a TA about it in office hours.

13.2 Compiling and Running

We have provided a Makefile that will run gcc for you. To compile your code with no optimizations (which you should do while developing, it will make debugging easier) and test with the "random" algorithm, run:

```
$ make
$./vm-sim -i traces/<trace>.trace -rrandom
```

Once your FIFO algorithm has been implemented, you can run the program with the `-rfifo` argument in order to test. For example, you should run:

```
$ make
$./vm-sim -i traces/<trace>.trace -rfifo
```

Once your LRU algorithm has been implemented, you can run the program with the `-rlru` argument in order to test. For example, you should run:

```
$ make
$./vm-sim -i traces/<trace>.trace -rlru
```

We highly recommend starting with "`simple.trace`." This will allow you to test the core functionality of your virtual memory simulator without worrying about context switches or write backs, as this trace contains neither.

13.3 Inputting Trace Lines

The sim allows to run with the -s flag that will allow you to input the commands in the trace files from the command line. Run:

```
$ make
$./vm-sim -s -rrandom
```

The commands from the trace files come in 3 forms.

Start Process: START <PID>
Stop Process: STOP <PID>
Memory Access: <PID> <r/w> <Address> <Value>

A simple example that starts a process, saves and then reads some data, and ends the process would look like:

```
$ make
$./vm-sim -s -rrandom
Input trace lines.
START 3
        0: PID 3 started
3 w d84f78 7
        1:  3 w 0xd84f78 <- 07
3 r d84f78 0
        2:  3 r 0xd84f78 -> 07
STOP 3
        3: PID 3 stopped
```

You will need to control-c to quit the simulation.

13.4 Corruption Checker

One challenge of working with any memory-management system is that your system can easily corrupt its own data structures if it misbehaves! Such corruption issues can easily hide until many cycles later, when they manifest as seemingly unrelated crashes later.

To help with detecting these issues, we've included a "corruption check" mode that aggressively verifies your data structures after every cycle. To use the corruption checker, run the simulator with the `-c` argument:

```
$./vm-sim -c -i traces/<trace>.trace -r<algorithm>
```

13.5 Debugging Tips with GDB

If your program is crashing or misbehaving, you can use GDB to locate the bug. GDB is a command line interface that will allow you to set breakpoints, step through your code, see variable values, and identify segfaults. There are tons of online guides, click here (http://condor.depaul.edu/glancast/373class/docs/gdb.html) for one.

13.5.1 Compiling with Debugging Information

To compile with debugging information, you must build the program with `make debug`:

```
$ make clean
$ make debug
```

13.5.2 Starting GDB

To start your program in gdb, run:

```
$ gdb ./vm-sim
```

13.5.3 Setting Breakpoints

You may find it useful to set a breakpoint inside the main loop of the simulator to debug specific simulator commands in your implementation. You can do this either by finding the line number inside `pagesim.c` and breaking there:

```
$ (gdb) break pagesim.c:53 ! set breakpoint at call to system_init
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for breakpoint)
$ (gdb) s   ! step into the function call
```

...or by using the actual function name being called from the main loop:

```
$ (gdb) break sim_cmd      ! set breakpoint at call to sim_cmd
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for breakpoint)
$ (gdb) s   ! step into the function call
```

13.5.4 Stepping Through Code

Once execution pauses at a breakpoint, you can step through your code using the `step` command. For example:

```
$ (gdb) s   ! step into the function call
```

13.5.5 Examining Memory

Sometimes, you may want to examine a large area of memory. To do this in GDB, you can use the `x` command (short for examine). For example, to examine the first 24 bytes of the frame table, we could do the following:

```
$ (gdb) x/24xb frame_table
0x1004000aa: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000b2: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000ba: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

The exact command syntax is x/[**Length**][**Format**] [**Address expression**].

13.5.6 Backtracing

If your program crashes, you can use backtracing to see the call stack at the point of the crash. For example:

```
$ (gdb) backtrace
```

13.5.7 Using the Corruption Checker

If you use the corruption checker, you can set a breakpoint on `panic()` and use a backtrace to discover the context in which the panic occurred:

```
$ (gdb) break panic
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for GDB to stop at the breakpoint)
$ (gdb) backtrace
$ (gdb) frame N    ! where N is the frame number you want to examine
```

Feel free to ask about gdb and how to use it in office hours and on Piazza. Do not ask a TA or post on Piazza about a segfault without first running your program through GDB.

13.6 Verifying Your Solution

On execution, the simulator will output data read/write values. To check against our solutions, run

```
$ ./vm-sim -i traces/<trace>.trace -r<algorithm> > my_output.log
$ diff my_output.log outputs/<trace>.log
```

The second half of the output file name includes the type of replacement algorithm that should be run when comparing the output. Ex. astar-random.log should be compared with the output from using random replacement algorithm ($-rrandom$) as shown below.

```
$ ./vm-sim -i traces/astar-random.trace -rrandom > my_output.log
$ diff my_output.log outputs/astar-random.log
```

You **MUST** implement the FIFO algorithm in order to test against all the *-fifo.log output files.

NOTE: To get full credit you must completely match the TA generated outputs for each trace.

14 How to Submit

Submit all the files in the student-src/ directory:

- address_splitting.h
- mmu.c
- page_fault.c
- page_replacement.c
- proc.c
- stats.c

Always re-download your assignment from Canvas after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you may receive a 0 regardless of what is on your machine.

This project will be demoed. In order to receive full credit, you must sign up for a demo slot and complete the demo. We will announce when demo times are released.

15 Debugging Tips with GDB

If your program is crashing or misbehaving, you can use GDB to locate the bug. GDB is a command line interface that will allow you to set breakpoints, step through your code, see variable values, and identify segfaults. Below are some helpful debugging tips with GDB:

15.0.1 Compiling with Debugging Information

To compile with debugging information, you must build the program with `make debug`:

```
$ make clean
$ make debug
```

15.0.2 Starting GDB

To start your program in gdb, run:

```
$ gdb ./vm-sim
```

15.0.3 Setting Breakpoints

You may find it useful to set a breakpoint inside the main loop of the simulator to debug specific simulator commands in your implementation. You can do this either by finding the line number inside `pagesim.c` and breaking there:

```
$ (gdb) break pagesim.c:53 ! set breakpoint at call to system_init
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for breakpoint)
$ (gdb) s   ! step into the function call
```

...or by using the actual function name being called from the main loop:

```
$ (gdb) break sim_cmd      ! set breakpoint at call to sim_cmd
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for breakpoint)
$ (gdb) s   ! step into the function call
```

15.0.4 Stepping Through Code

Once execution pauses at a breakpoint, you can step through your code using the `step` command. For example:

```
$ (gdb) s   ! step into the function call
```

15.0.5 Examining Memory

Sometimes, you may want to examine a large area of memory. To do this in GDB, you can use the `x` command (short for examine). For example, to examine the first 24 bytes of the frame table, we could do the following:

```
$ (gdb) x/24xb frame_table
0x1004000aa: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000b2: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000ba: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

15.0.6 Backtracing

If your program crashes, you can use backtracing to see the call stack at the point of the crash. For example:

```
$ (gdb) backtrace
```

15.0.7 Using the Corruption Checker

If you use the corruption checker, you can set a breakpoint on `panic()` and use a backtrace to discover the context in which the panic occurred:

```
$ (gdb) break panic
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for GDB to stop at the breakpoint)
$ (gdb) backtrace
$ (gdb) frame N    ! where N is the frame number you want to examine
```

Remember: Always run your program through GDB before asking for help with a segfault.

These are just a few basic GDB commands to get you started. Feel free to explore more commands and consult GDB documentation for advanced debugging techniques.