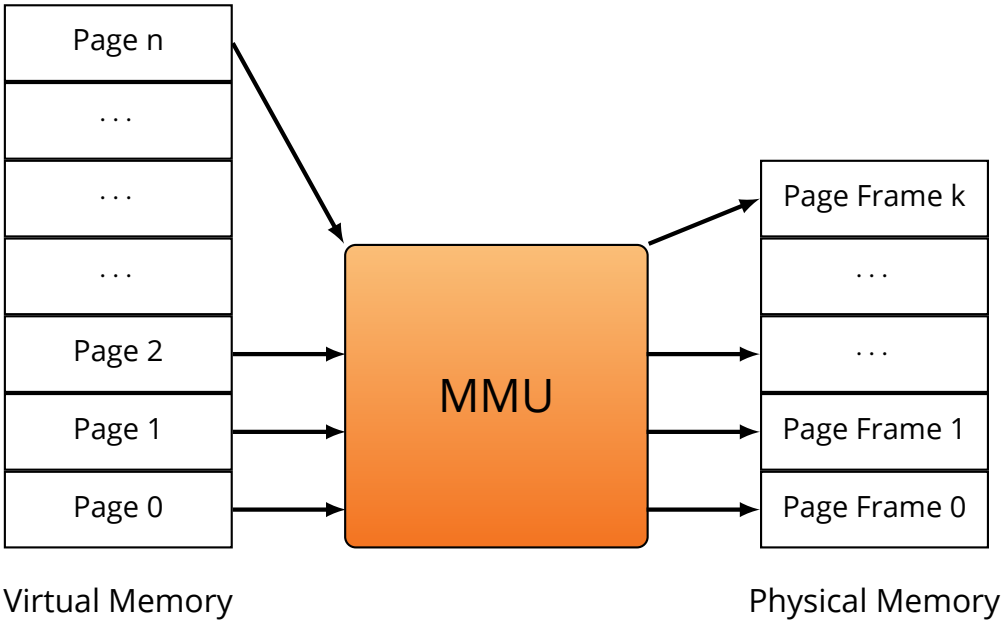
Operating Systems 202.1.3031

Spring 2022/2023 Assignment 3

Memory Management

Responsible Teaching Assistants:
Ido Ben-Yair and Hedi Zisling



Ben-Gurion University
of the Negev

Contents

1 Introduction

Welcome to the world of memory management!

Memory management is one of the key features of every operating system. In this assignment, we will examine how xv6 handles memory and extend it by implementing paging. This will allow xv6 to store parts of a process' memory in secondary storage, such as a hard disk or an SSD. To help you get started, we will first provide a brief overview of the memory management facilities available in xv6. We strongly suggest you read this section while examining the relevant xv6 files (vm.c, memlayout.h, kalloc.c, etc.) and documentation.

1.1 Overview of memory management in xv6

Memory in xv6 is managed in 4096 ($= 2^{12}$) bytes long pages (and frames). Each process has its page table that maps virtual user space addresses to physical addresses, while all processes share the *kernel page table*. In xv6 RISC-V, only the bottom 39 bits of a 64-bit virtual address (VA) are used, using a 3-level paging mechanism:

- The first 9 bits are the PTE (Page Table Entry) index into the process page table,
- The second 9 bits are the PTE index into the second-level page table,
- Another 9 bits for the index in the last-level page table,
- The last 12 bits in the VA are the offset into the physical page.

Looking at the page table data structure, each PTE is 64 bits (8 bytes) long, and each page table contains 512 PTEs. In each PTE, the first 10 bits are reserved and not used, followed by 44 bits which are the page frame number (physical page number), and 10 flag bits (see figure 1). The xv6 kernel uses direct mapping, that is, a kernel VA is also its physical address (PA) (note that this does not override the paging system). Direct mapping simplifies kernel code when handling memory. There are however some exceptions: The trampoline page is at the top of the VA space (`MAXVA`), and just below it are the kernel stack pages for each process. Each kernel stack page is followed

by a guard page, which serves as a barrier between the different stack pages. The kernel memory starts at `KERNBASE`, and apart from the above exceptions, continues until `PHYSTOP` (see figure 2).

While in principle each process' virtual memory is limited by `MAXVA` (256GB), in practice, runtime allocations in xv6 occur between the kernel data segment and `PHYSTOP`. There exists a double mapping - when a process asks the kernel to allocate memory, the memory is allocated from the kernel free list using `kalloc()`. Thus, you can access the memory from the process page directory after the allocation, and from the kernel using its direct memory mapping.

The location of the trampoline page is the same in both kernel space and user space. However, in user space the process trapframe is located directly below the trampoline page for that process. The process `text` segment starts at VA 0, followed by the data segment, a guard page, and the process stack.

Good luck and have fun!

2 Submission Instructions

- Make sure your code compiles without errors and warnings and runs properly!
- We recommend that you comment your code and explain your choices, if needed. This would also be helpful for the discussion with the graders.
- You should submit your code as a single `.tar.gz` or `.zip` file.
- We advise you to work with git and commit your changes regularly. This will make it easier for you to track your progress, collaborate and maintain a working version of the code to compare to when things go wrong.
- Theoretical questions in this assignment are **not for submission**, but you are advised to understand the answers to these questions for the grading sessions.
- Submission is allowed **only in pairs** and **only via Moodle**. Email submissions will not be accepted.
- Before submitting, run the following command in your xv6 directory:

  ```
  $ make clean
  ```

  This will remove all compiled files as well as the `obj` directory.
- Help with the assignment and git will be given during office hours. Please email us in advance.

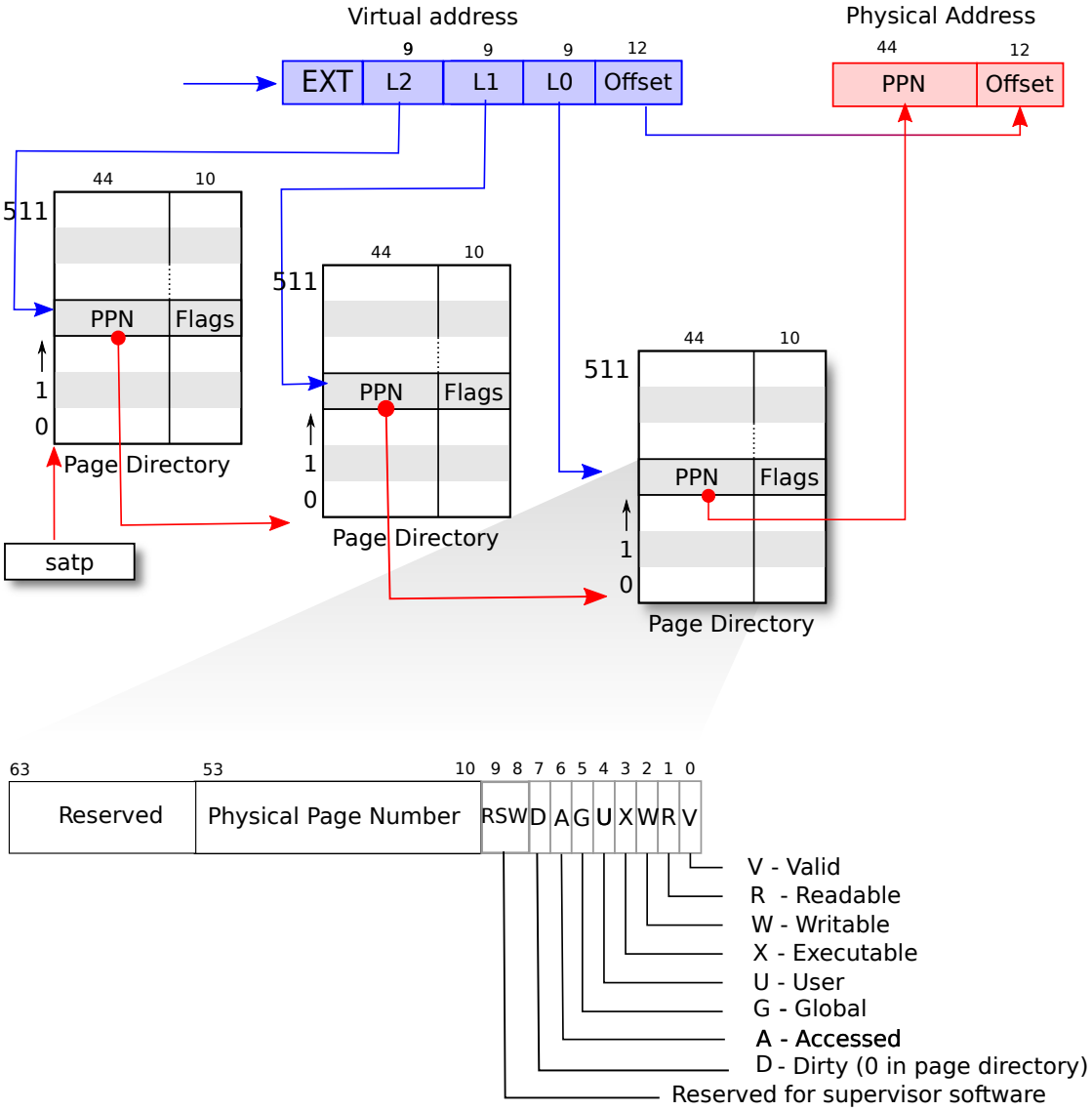


Figure 1: Page table structure in RISC-V xv6, from the xv6 book

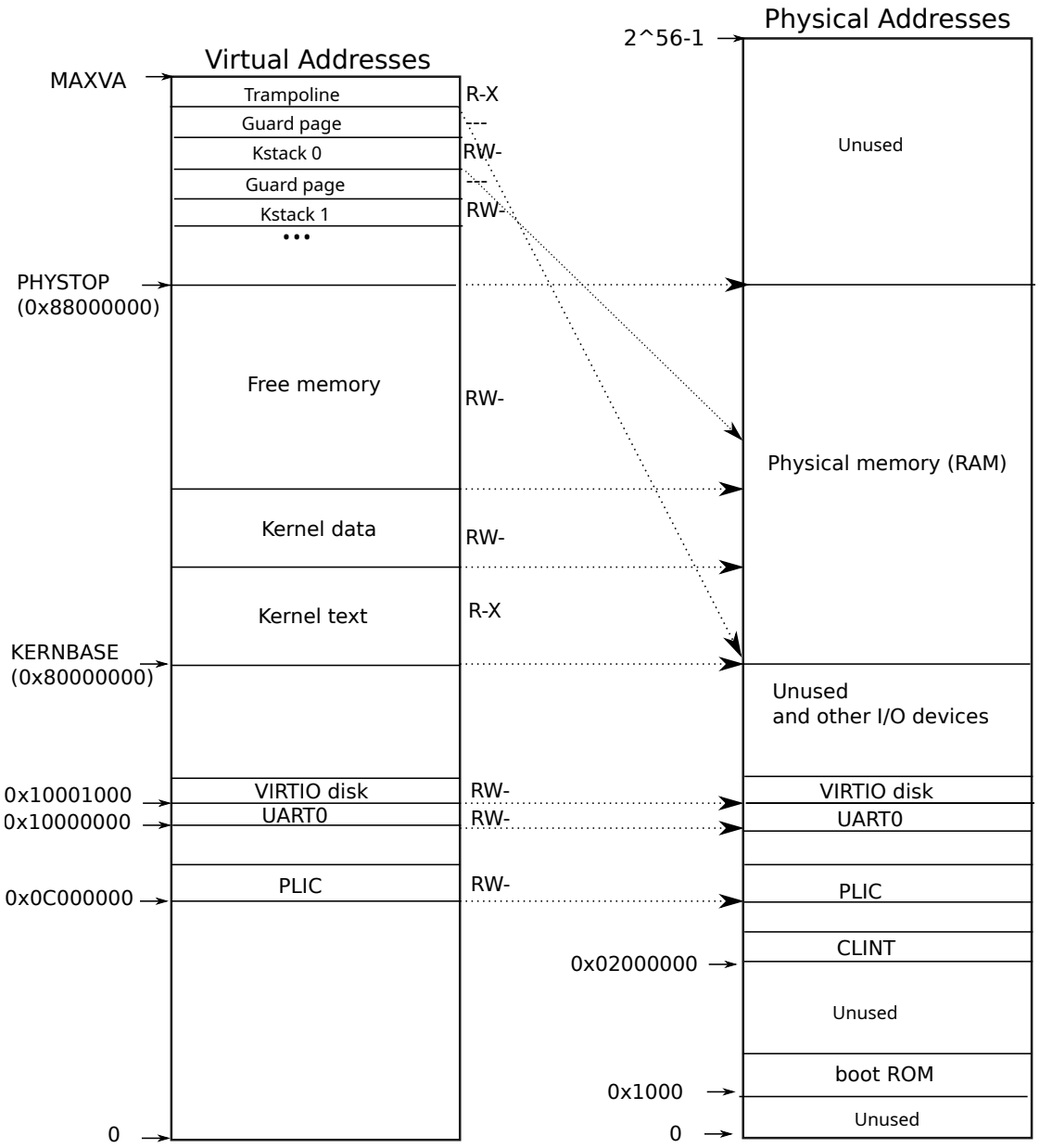


Figure 2: On the left, xv6's kernel address space. RWX refers to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see. Taken from the xv6 book.

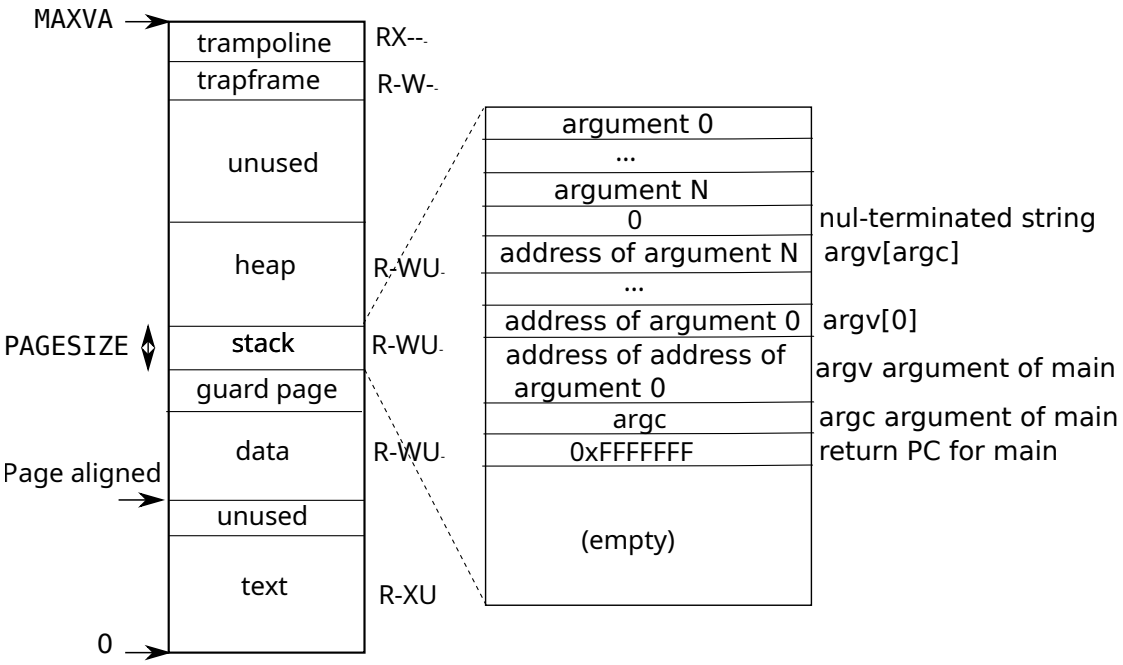


Figure 3: Structure of the virtual address space for a process in RISC-V xv6, from the xv6 book

3 Task 1: Userspace Stack Memory Allocator

First, let's discuss memory management in userspace.

As we saw in the first assignment, `malloc()` and `free()` are implemented in the "C library" entirely in user space. In xv6, `free()` does not actually release memory back to the kernel, but only adds it to a free list. Thus, memory is never actually released to be reused until the process exits.

In this task, you will implement a user space memory allocator that functions like a stack: it will assume that buffers are allocated and freed in LIFO order. That is — if the user program allocates blocks A, B, and C, then it must free first C, then B, then A. When the allocator needs more memory, it will request another page from the kernel using the `sbrk()` system call. However, unlike the `malloc()` implementation in xv6, your allocator will enforce the LIFO assumption: when the user asks to free a buffer, the last allocated buffer will be freed and the stack will be rolled back to the last allocated buffer. If a page boundary is crossed during a free operation, the page must be freed using a call to `sbrk()` with a negative argument. Since pages in xv6 are 4096 bytes long, and we don't want to deal with multiple page crossings in one push or pop operation, the maximum size of a push/alloc operation will be **512 bytes**.

To understand how to implement this allocator, examine `malloc()` and `free()` in `umalloc.c`, as well as the implementation of `sbrk()` in `sysproc.c`. Read these functions carefully, and make sure you understand how they work. Macros like `PGSIZE`, `PGROUNDUP` and `PGROUNDDOWN` will also be useful. Use them by including `kernel/riscv.h`.

Note: this stack is not the same as the one used by the CPU/compiler to keep track of function calls, nor is it related to the `p->kstack` field in the process structure. It's just a stack data structure to be used by applications.

WARNING

Do not use or call `malloc()` and `free()` in a process that uses your stack memory allocator. They will not play well together due to the stateful nature of the `sbrk()` interface, which will require the different allocators to be aware of each other.

3.1 The stack memory allocator API

The stack memory allocator will be used by the user application through the following API, exposed through the new `ustack.h` header file:

1. `void* ustack_malloc(uint len);`

   Allocates a buffer of length `len` onto the stack and returns a pointer to the beginning of the buffer.

   If `len` is larger than the maximum allowed size of an allocation operation, the function should return `-1`. If a call to `sbrk()` fails, the function should return `-1`.

   Note: the "heap" area does not yet exist at the first call to `ustack_malloc()`. Refer to the implementation of `malloc()` to see how to handle this case.

2. `int ustack_free(void);`

   Frees the last allocated buffer and pops it from the stack. After this call completes, any pointers to the popped buffer are invalid and must not be used.

   Should return `-1` on error, and the length of the freed buffer on success. If the stack is empty, the function should return `-1`. When the last allocated buffer is freed, all pages allocated by `sbrk()` should be released back to the kernel. A call to `sbrk(-PGSIZE)` will never fail and you may assume this in your code.

These functions will be written in the new `ustack.c` file. You are free to use any data structures you wish, as long as they are implemented in user space.

Both functions must execute in **constant time**, must **not** use `malloc()` or `free()`, and must **not** use any system calls other than `sbrk()`. To avoid corruption of your data structures, **check for errors before modifying any data structures!** Also, just as there is no initialization function for `malloc()`, no initialization function is allowed here! Finally, no changes to the kernel are allowed for this task.

3.2 Grading

Our grading program will allocate and free buffers of different sizes and verify that the stack abstraction works as expected. It will also test the failure modes defined in the API above. Furthermore, it will test the exact number of page allocations and deallocations performed by your allocator, so make sure to allocate only when necessary and only in units of 1 page, deallocate when required, and avoid leaving holes in your memory. Use constants provided by the OS and `sizeof`, not hard-coded values. Make sure to handle all possible failure cases!

Task 1 – Implement the stack memory allocator

1. Read and understand the implementations of `malloc()` and `free()` in `umalloc.c`.
2. Create a new header file `ustack.h` and declare the API described above as well as any constants you need.
3. Implement the stack memory allocator in `ustack.c`.
4. Add `ustack.c` to `ULIB` in `Makefile`.
5. Make sure neither `ustack.c` nor `ustack.h` expose any implementation details to the user.

For submission: `ustack.c`, `ustack.h`, and the modified `Makefile`. You may also include a test program in your submission, but it is not required and will not be executed or graded.

4 Task 2: Swapping Pages Out and In

An important feature lacking in xv6 is the ability to swap out pages to a backing store, such as a hard disk or an SSD. That is — at each moment in time, all process memory is held in physical memory. In this task, you will implement a paging system for xv6, which can take pages out of physical memory and store them on disk. In addition, the system will retrieve pages back to memory when needed. In our system, the kernel will implement a paging policy for each process separately, as opposed to a global pool of pages for all processes. To keep things simple, we have provided a file system interface to manipulate files from kernel code, in which swapped out memory pages will be stored. This file system interface is described in the next section.

4.1 The file system interface

To complete this assigment, you are given a library for creating, writing, reading, and deleting swap files. This library was added to `kernel/fs.c`, and `struct proc` was modified to hold a pointer to the swap file, `swapFile`. This file, named `"/.swap<pid>"` where `<pid>` is the PID of the process, will hold the swapped-out memory.

Review the following functions and understand how to use them:

1. ```
   int createSwapFile(struct proc *p);
   ```

   Creates a new swap file for a given process `p`.
   Requires `p->pid` to be initialized correctly.

   Returns 0 on success and -1 on error.

2. ```
   int readFromSwapFile(struct proc *p, char* buffer,
                        uint fileOffset, uint size);
   ```

   Reads `size` bytes into `buffer` starting from `fileOffset` in the swap file for given process `p`.
   Note that if `fileOffset == n`, the file size should be at least `n`, and it will read `size` bytes or until the end of the file, if it is smaller than `fileOffset + size`.

   Returns the number of bytes read on success, and -1 on error.

3. `int writeToSwapFile(struct proc *p, char* buffer, uint fileOffset, uint size);`

   Writes `size` bytes from `buffer` from `fileOffset` in the swap file for given process `p`.
   Note that if `fileOffset == n`, the file size should be at least `n`.

   Returns the number of bytes written on success, and -1 on error.

4. `int removeSwapFile(struct proc *p)`

   Deletes the swap file for given process `p`.
   Requires `p->pid` to be initialized correctly.

   Returns 0 on success and -1 on error.

4.2 Swapping out – storing memory pages into files

Next, we detail some restrictions on processes in our new paging system. At any given time, a process should have no more than `MAX_PSYC_PAGES=16` pages in physical memory. In addition, a process may not use more than `MAX_TOTAL_PAGES=32` pages. Whenever a process exceeds `MAX_PSYC_PAGES`, the kernel must select as many pages as needed to restore the limitation, and move them to the swap file for this process.

To keep track of which pages are in the swap file for a process and where they are located in that file (i.e., paging metadata), you should enrich the PCB with a paging metadata structure. We leave the design of this data structure to you. Be sure to swap only the process' private user memory pages (leaves). If you don't know what this means, please find the information in the xv6 book. When swapped out, don't forget to free a page's physical memory. Lastly, remember xv6's double memory mapping: a page's physical address is its "virtual address" when accessing it from kernel space.

Whenever a page is moved to the paging file, it should be marked in the process' page table entry as *not present*. This is done by clearing the *valid flag* (`PTE_V`). A cleared *present* flag does not imply that a page was swapped out, since there could be other reasons for this flag being cleared. For this reason, we will use one of the unused flag bits (see figure 1) in the PTE to indicate that the page was swapped out.

Add the following line to `riscv.h`:

```
#define PTE_PG (1L << 9) // Swapped out
```

Now, whenever you move a page to the secondary storage, set this flag.

4.3 Swapping in – retrieving pages on demand

While executing, a process may require swapped-out data. Whenever the MMU fails to access the required page, it generates a trap. Use `r_scause()` to determine the reason for the trap, which should be either 13 or 15 for a page fault, and `r_stval()` to determine the faulting address. Use this address to identify the page. Check its PTE to determine if this page has been swapped out or if this is just a plain old segmentation fault. If the page resides in the swap file, allocate a new physical page, copy its data from the file, and map the page back into the page table. After returning from the trap frame to user space, the process retries executing the faulting instruction and should not generate a page fault if your handling of the page fault was correct. Don't forget to check if the current process is already using `MAX_PSYC_PAGES`. If so, another page should be swapped out. The decision as to which page should be selected for swapping out is the subject of Task 3. For now, you can select a relevant page as you see fit.

4.4 Changes to existing functionality

Our new functionality also affects other parts of xv6. You should modify the existing code to work properly with our paging system. Specifically, make sure that `fork()` and `exec()` still work properly. The forked process should have its own swap file whose initial content is identical to the parent's file. Upon termination, the kernel should delete the swap file, and properly free the process' pages which reside in the physical memory. To avoid issues, the shell and init process should be **disregarded by our paging system**.

IMPORTANT

After completing this task, some of the tests in `usertests` might not work. This is because some use more than 32 memory pages.
This is OK.

Task 2 – Add paging to the xv6 kernel

1. Apply the changes from `paging.patch` to your xv6 kernel.
2. Use the provided functions to implement a paging system when a page fault occurs. Page faults cause traps, which are handled (you know it!) by the trap handlers in `trap.c`.

For submission: Modified kernel code that implements the paging system.

4.5 Notes and optional further reading

- The size restrictions are necessary since xv6's file system can generate only small size files (up to 17 pages).
- You may assume the maximal file size of an executable (ELF file) is smaller than 13 pages. This is because `exec()` allocates a maximum of 15 pages, of which 2 serve a special purpose. To support larger ELF files, you would have to create a temporary swap file, and we don't want to deal with this issue in this assignment.
- There are good reasons for not writing to files from within kernel code, but in this assignment, we ignore these. Some of these good reasons appear here.
- Real-world operating systems often use a swap *partition* instead of a swap file. A brief discussion of the pros and cons is available here.
- You may wonder what happens if the user attempts to access a swapped-out page, since we never removed it from the TLB, so the TLB might still hold a reference to its old mapping. xv6 refreshes the TLB when moving between user space and kernel space, using the `sfence.vma` instruction, which is part of the RISC-V instruction set. This is inefficient because it will generate a page fault for every page the process might try to access — even if the page is still in memory — but it is the simplest way to ensure that the TLB is always up-to-date and mappings are never stale.
- Finally, if you really want to understand VM in a real operating system (as of 2007, at least), you are invited to read Mel Gorman's compherensive book Understanding the Linux Virtual Memory Manager.

5 Task 3: Page Replacement Algorithms

Now that you have a paging framework, an important question remains to be answered: which pages should be swapped out? As seen in class, there are numerous alternatives to selecting which pages should be swapped out. In our case, controlling which policy is executed will be done with the aid of a compile-time flag, `SWAP_ALGO`.

For this assignment, we will implement only a few simple page replacement algorithms:

1. NOT FREQUENTLY USED + AGING (`SWAP_ALGO=NFUA`):

   For each page, maintain a counter. When a page is accessed (check the status of the `PTE_A`), the counter is shifted right by one bit, and then the digit 1 is added to the most significant bit (i.e., the leftmost bit). If a page was never accessed, the counter is just shifted right by one bit (i.e., a leading zero should appear in the MSB). The page with the lowest counter should be swapped out. When a page is created or loaded into memory, reset its counter to 0.

2. LEAST ACCESSED PAGE + AGING (`SWAP_ALGO=LAPA`):

   Keep track of a counter that is shifted similarly to the previous algorithm, but the page with the smallest number of 1s will be swapped out. If there are several such pages, the one with the lowest counter should be removed. When a page is created or loaded into memory, reset its counter to `0xFFFFFFFF`.

3. SECOND CHANCE FIFO (`SWAP_ALGO=SCFIFO`):

   As learned in class, according to the order in which the pages were created and the status of the `PTE_A` flag of the page table entry (accessed/reference bit).

4. DISABLED (`SWAP_ALGO=NONE`): No paging is done, and the system should behave exactly like the original xv6.

The AGING algorithms should update their data structures every time a process returns to the scheduler function. Don't forget to clear the `PTE_A` flag!

Complete the following task to implement the page replacement algorithms:

Task 3 – Implement page replacement algorithms

1. Modify the Makefile to support `SWAP_ALGO` – a macro that selects the appropriate page replacement scheme during compilation. For example, the following line should invoke the xv6 build with `SCFIFO` scheduling:

   ```
   $ make qemu SWAP_ALGO=SCFIFO
   ```

   If the `SWAP_ALGO` macro is omitted, Second Chance FIFO should be used as default. You can do that by adding the following code to `Makefile`:

   ```
   ifndef SWAP_ALGO
       SWAP_ALGO:=SCFIFO
   endif
   ```

2. Pass the value of `SWAP_ALGO` to the compiler as a command line argument. You can do that by adding the following line to the `Makefile` where appropriate:

   ```
   CFLAGS += -DSWAP_ALGO=$(SWAP_ALGO)
   ```

   `SWAP_ALGO` should now be available in the code as a macro.

   Tip: you should read about preprocessor directives in C (e.g., `#define`, `#ifdef` and `#if`).

3. Implement each of the 3 page replacement algorithms as well as the ability to disable paging.

   It is up to you to decide which data structures are required for each algorithm.

For submission: Modified kernel code that implements the page replacement algorithms and modified `Makefile`. You may also include a test program in your submission, but it is not required and will not be executed or graded.