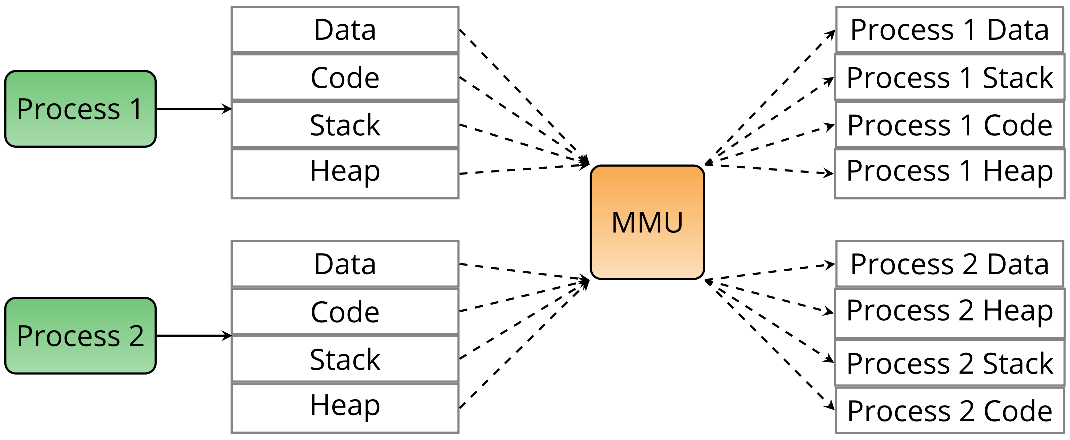
Operating Systems 202.1.3031

Spring 2025 Assignment 3

Memory Management



Ben-Gurion University of the Negev

Contents

1 Introduction

Welcome to the third assignment in our operating systems journey!

In this assignment, you'll take a deeper dive into how `xv6` handles userspace memory by extending it to support shared memory between processes. You'll use this new mechanism to build a simple multi-process logging system. Along the way, you'll apply atomic operations to ensure multiple processes can safely write to a shared logging buffer.

This assignment builds on many of the core ideas you've encountered so far — especially virtual memory, page tables, and process management. It will help solidify your understanding of:

- How virtual memory abstracts physical memory
- How page tables control memory mapping and access
- How processes interact with their virtual memory space and with shared memory

This assignment is a bit more involved than the previous ones — but take it step by step, and you'll learn a lot along the way. You've got this!

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.
- Submit a **single copy** of xv6 containing all of the changes you made, in 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.

3 Task 1: Memory Sharing

We'll start off by implementing memory sharing in xv6. Shared memory is a memory segment that can be accessed by multiple processes. Real-world uses include interprocess communication (IPC) and shared data structures. Android makes extensive use of shared memory for IPC between applications and system services, most notably in the Binder IPC framework and the UI compositing subsystem, SurfaceFlinger. Our shared memory system will have a simple API. Given a virtual pointer in a process' address space, the system will map as many pages as needed to the address space of another process.

We will now describe the API functions to support shared memory:

Implement a kernel function that, given a virtual address in the source process, maps the corresponding physical pages to a destination process:

```
uint64 map_shared_pages(struct proc* src_proc,
                        struct proc* dst_proc,
                        uint64 src_va, uint64 size);
```

where `src_proc` is the source process, `dst_proc` is the destination process, `src_va` is the virtual address in the source process, and `size` is the number of bytes to map. The function should return the virtual address in the destination process where that corresponds to the source address in the source process, which we will eventually provide to a user process by a system call. It is important to note that `src_va` is a virtual address in the source process. Our function, however, needs to map the corresponding **physical** pages to the destination process. Use the existing functions in `vm.c` to accomplish this, namely `walk()` and `mappages()`.

When implementing `map_shared_pages()`, pay attention to the following:

- **Recall** that the source address might not be page-aligned, but mappings are only done in page-sized chunks. So you must map the correct number of pages to cover the desired size, and return the virtual address that resides in the first mapped page in the destination process, with **the correct offset** from the start of the page. This may result in mapping slightly more memory than requested, but this is acceptable as there is no way around it in a page-based memory management system.

- **Check** for correctness: verify that `walk()` returns a valid page table entry (PTE) for the source address, and that the requested mapping is valid (`PTE_V`) and user-accessible (`PTE_U`).
- **Maintain** the correct size of the process address space in the `sz` field of the `proc` structure. Otherwise, if the process exits, the kernel will attempt to release the wrong amount of memory.
- **Set** the correct permissions for the pages in the destination process — copy the flags from the source mapping and combine them with a new flag to indicate that the mapping is not owned by the destination process (i.e., shared): add a new flag to `riscv.h`:

  ```
  #define PTE_S (1L << 8) // shared page
  ```

  Bit number 8 in the PTE is unused by the hardware and can be used for this purpose, as discussed in class.
- **Use** the `PGROUNDDOWN` and `PGROUNDUP` macros where needed to map from the beginning of a page, and to only change the size of the processes by a multiple of the page size. Other useful macros you will need to use are `PGSIZE`, `PTE2PA`, and `PTE_FLAGS`.

To decide what the new virtual address should be in the destination process, **read the code** for `uvmalloc()` in `vm.c` to understand how it assigns new addresses. Since the destination process may allocate memory in addition to these mappings, a real system would need a more complex mechanism to manage the virtual address space of the destination process. For this assignment, let's ignore these complexities and assume the destination process does not mix mallocs with shared memory.

Implement the corresponding function to unmap the shared memory from the destination process:

```
uint64 unmap_shared_pages(struct proc* p, uint64 addr, uint64 size);
```

The same concerns as noted above apply here as well. Use the `uvmunmap()` function to unmap the pages from the destination process. Note the difference in arguments compared to `mappages()`. Don't forget to update the `sz` field of the process to reflect the new size of the address space! Return 0 on success and -1 on failure, for example if the requested mapping does not exist or isn't a shared mapping.

Finally, we need to teach the kernel to only free physical pages if they were originally allocated by the process being deleted. When a process exits, the kernel will free all of its pages via the mappings in the page table. This is because xv6 was not built with shared memory in mind, so it assumes that all pages are owned by the process that created them. This is not the case for a process that has shared memory mappings. Therefore, the default code in the kernel will free the same pages multiple times, which is bad.

Modify the `uvmunmap()` function to only free physical pages if the mapping is owned by the process being deleted. Note that while the *mappings* must be deleted from the page table in both cases, the *physical pages* should only be freed when the pages are owned by the process being deleted! Weird things will happen if the mappings are not cleared properly. Try it for fun, and see!

To test the shared memory code, **expose** the functions to userspace by adding the following system calls:

```
uint64 sys_map_shared_pages(void);
uint64 sys_unmap_shared_pages(void);
```

Their arguments and return values are up to you.

Test your implementation with a userspace program that uses shared memory. Implement the test program in `shmem_test.c`:

1. Create a shared memory mapping from a parent process to a child process.
2. Write the string "Hello daddy" in the child and print it in the parent.

3. Unmap the shared memory in the child process and show that `malloc()` can now allocate memory in the child process properly.
4. Print the size of the child process before the mapping, after the mapping, after the unmapping and after the call to `malloc()`.
5. Show that the mappings can be cleaned up properly by calling the unmap function and print the size after the unmapping. It should be the same as before the mapping.
6. Include a flag to disable the unmapping in the child process. This will allow you to test the case where the child process exits without unmapping the pages. In this case, the kernel should not free these pages, and the parent process should still be able to access them.

The test should be able to run more than once without rebooting the system, which means your memory mapping code must leave the system in a valid state.

Task 1 – Implement memory sharing in xv6

1. Add the `map_shared_pages()` and `unmap_shared_pages()` functions to the kernel to map shared memory from one process to another.
2. Expose the memory sharing functionality to userspace by adding the `sys_map_shared_pages()` and `sys_unmap_shared_pages()` system calls.
3. Show that your implementation works with the test program `shmem_test.c`.

For submission: Modified kernel code that implements the shared memory system and `shmem_test.c`.

Questions – not for submission

1. How does the kernel manage the virtual address space of a process?
2. How does the kernel manage the physical memory?
3. How are virtual addresses mapped to physical ones?
4. How is memory sharing done? Why do we need to look up the physical address of the source page?
5. Why might the virtual address for the same physical memory be different in different processes?
6. Can we map the same physical page to multiple virtual addresses in the same process?
7. Can we map multiple physical pages to the same virtual address?
8. How does `uvmalloc()` assign new addresses to a process?
9. How does `mappages()` map pages into a process' address space?
10. How does `uvmunmap()` delete pages from a process' address space? What does the `do_free` parameter do?
11. What is the purpose of the `PTE_S` flag? Can you think of a way to implement shared memory without the `PTE_S` flag?
12. What do the `PTE_V` and `PTE_U` flags mean? What about the other flags?
13. What happens if we don't deallocate the physical pages when a process exits?
14. What happens if we deallocate the pages in both processes or in the wrong process?
15. Why do we need to keep track of the size of the process' address space in the `sz` field?
16. What do the macros `PGROUNDDOWN` and `PGROUNDUP` do? Why are they needed?
17. What do the macros `PTE2PA`, `PA2PTE` and `PTE_FLAGS` do?

4 Task 2: Multi-process Logging

In this task, we will implement a logging system that allows multiple processes to write messages to a shared memory buffer. This builds upon the shared memory system from the previous task and the test code in `shmem_test.c`.

The logging system uses a shared memory buffer accessible by all processes. As in the test program, begin by forking multiple child processes before allocating the shared buffer. Once all child processes are created and have access to their shared memory pointer, the parent process maps the shared buffer into each child using the `map_shared_pages()` function.

To avoid conflicts when multiple child processes write to the buffer simultaneously, we'll define a simple logging protocol using atomic operations. Each child is assigned a unique index by the parent and attempts to write log messages by first claiming a segment of the shared buffer.

Each message begins with a 32-bit header that encodes two values:

`uint16 child index`	`uint16 message length`

A zero-value header indicates a free segment. When a child encounters a zero header, it attempts to write its own header to claim the following `N` bytes (where `N` is the message length, without the null terminator) for its message. If the header is non-zero, the child skips ahead to the next potential header location and tries again.

The parent process reads messages by scanning the buffer. It reads a header, and if it's non-zero, it reads and prints the message, then continues scanning.

Atomic header writes and other implementation details

To ensure header writes are atomic and not partially updated, we use the `__sync_val_compare_and_swap` operation. This function atomically compares the value at a memory location with a given value. If equal, it replaces it with a new header and returns the old value. Regardless of the outcome, it returns the original value, making it ideal for our protocol.

Note that this operation requires 4-byte alignment. To maintain alignment after reading or writing a message, all processes must advance the address by `N + 4` bytes (the message plus header), then align to the next 4-byte boundary using:

```
addr = (addr + 3) & ~3;
```

This ensures the address is properly aligned for the next header write or read.

If a process detects that its current reading or writing address exceeds the size of the shared buffer (i.e., the starting address + `PGSIZE`), it should stop processing and exit. Avoid complex error handling — just ensure the program doesn't crash due to an invalid memory access.

Implement the logging system in `log_test.c`. Test it with at least 4 child processes. Avoid complex error handling and focus on ensuring the system works correctly in the "happy path". You do not have to free or unmap the buffer when the parent process exits, as the kernel will clean it up (if you implemented the unmapping correctly in Task 1).

Include the child process index in the string written to the buffer by each child. When printing the messages, the parent should also print the child index so we can verify that the messages are delivered correctly and match their process of origin. You may use messages of the same length, but your code must support messages of different lengths and it may not assume a fixed length.

You may add a mechanism to increase fairness between child processes, but this is not required. Make sure at least one child exceeds the buffer size to test the error handling.

Approach guidance

This task may seem intimidating at first, especially if you haven't worked with low-level systems or shared memory before—but don't worry! The key is to break it down into manageable steps. Start by reviewing how shared memory is allocated and mapped in the previous task. Then, tackle one part at a time: fork the child processes, set up the shared buffer, and implement the message-writing protocol. Use print statements generously to observe what each process is doing, and insert short sleep calls to reduce console overlap and make the output easier to follow. Avoid submitting debug sleep calls or print statements though.

As a suggestion, you can start by implementing the logging system in a single process, before extending it to multiple processes. This will allow you to test the reading and writing of messages in a single process before adding the complexity of multiple processes.

Contrary to what you may have been shown so far, C is a fairly high-level language, and the compiler is allowed to make many tranformations in the name of optimization. This means your C code is essentially just a set of guidelines for the compiler to follow. For this reason, the code may not always behave as you expect. If you encounter a situation like this, first of all don't panic — this is a great learning opportunity. Try to understand what your code is asking the compiler to do (not what you think it's doing). More often than not, the bug is in the code you wrote rather than the code you intended to write. However, this may not always be the case. The C language has many pitfalls, nuances and caveats that can lead to unexpected behavior. If something truly doesn't make sense, try to first formulate what is happening, what you expected to happen, and try to come up with a theory you can test. If nothing else, this will help you formulate your question when asking for help, whether by searching the internet or asking your friends, an AI, or of course, us.

Pay close attention to memory alignment and buffer boundaries - but don't get discouraged. Mistakes are part of the process, and debugging is where much of the real learning happens. Take your time, test often, and don't hesitate to ask for help!

Task 2 — multi-process logging

1. Implement the logging system in `log_test.c`.
2. Fork at least 4 child processes and map the shared buffer.
3. Use the `__sync_val_compare_and_swap` function to implement atomic header writes.
4. Child processes should write messages to the shared buffer using the headers.
5. The parent process should read messages from the shared buffer and print them to the console.
6. Ensure that the logging works correctly and that messages do not overlap or get corrupted.
7. Ensure that the program does not crash or access invalid memory.

For submission: `log_test.c` containing the logging system and changes to the Makefile to compile it.

Questions — not for submission

1. What is the goal of the logging system, and why is shared memory a suitable mechanism for this task?
2. What information is stored in the header of each log entry, and how is this information used by both child and parent processes?
3. How does a child process find and claim space in the shared buffer before writing a message? What mechanism prevents multiple children from writing to the same location?
4. Why are atomic operations necessary for writing headers? What kinds of bugs or inconsistencies might occur if writes were not atomic?
5. Under what conditions does a child process stop attempting to write to the shared buffer? How does it detect that it has reached the end of the usable memory?
6. How does the parent process know where one message ends and the next begins?
7. What should happen if a child process attempts to write beyond the end of the shared buffer?
8. What is the maximum length of a message that can be logged, and how is this determined?
9. How many child processes can this system support at a maximum and why?
10. Consider our logging system as a synchronization system. Does it guarantee mutual exclusion? Can it deadlock? Can there be starvation? What are your observations?
11. Have you encountered any bugs or unexpected behavior while implementing the logging system? If so, how did you resolve them?
12. How would you extend this system to allow multiple readers as well as multiple writers?

5 Conclusion

Operating systems can be tricky — full of weird bugs, low-level details, and concepts that aren't always easy to visualize. We know this class hasn't been simple. But you've learned a great deal about how computers actually work behind the scenes, and many of the concepts you've learned in this class are applicable to other areas of computer science as well.

We hope these assignments helped make some of those ideas feel real (and maybe even a little fun?). If nothing else, you've now had the joy of debugging shared memory across multiple processes - an experience you're unlikely to forget anytime soon.

This is the final assignment for the course. Congrats on getting through it all, and best of luck on the final exam. You've earned a break (and maybe a nap).