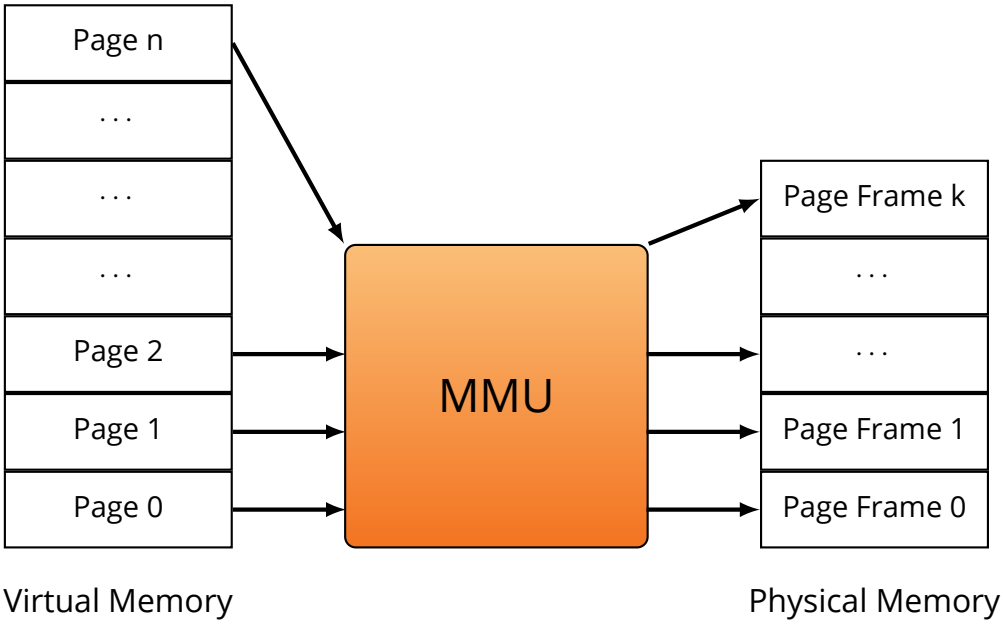
# Operating Systems 202.1.3031

## Spring 2024 Assignment 3

Memory Management

**Responsible Teaching Assistants:**
**Ido Ben-Yair and Pan Eyal**




Ben-Gurion University
of the Negev

# Contents

**1 Introduction** **2**

**2 Submission Instructions** **3**

**3 Task 1: Memory Sharing** **4**

**4 Task 2: Userspace Cryptographic Service** **9**

**5 Conclusion** **15**

# 1 Introduction

Memory management is one of the key features of every modern operating system. In this assignment, we will examine how xv6 handles userspace memory and extend it by implementing a shared memory system, which we will utilize to implement a cryptographic service for our OS.

This service is intended to enable user processes to encrypt or decrypt messages by calling what appears to the process as a regular system call, but in reality engages a more complex mechanism that involves the kernel, a cryptographic server process, and the shared memory system. Designs like this add complexity to the system, but also have several advantages over naive designs, such as:

- Enhanced security and reliability by isolating the cryptographic code from both the kernel and user processes.
- Simplified development by breaking down the system into more manageable parts and moving complex logic out of the kernel.
- Easier extensibility by allowing the cryptographic server to be replaced or upgraded without changing the kernel.
- The ability to provide inputs to the system that the user process may not have access to, such as inputting a key to the cryptographic server via a secure user interface or smart card. See macOS's Keychain Services, Windows' Data Protection API, or Android's Keystore for examples.

We will implement a simple cryptographic service using memory sharing, while simultaneously ensuring that the kernel remains free of any cryptographic code and complicated state management (unlike what we did in the previous assignment). Our system will consist of three parts: the capability to share memory pages between processes so that both could read and write to the same memory, a request queue and a userspace cryptographic server. Data pages from a client process will be shared temporarily with the server for the duration of the operation.

**On a personal note, we know this assignment is a bit more complex than the previous ones, but we believe you can do it and learn a lot in the process!**

# 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 solution as a **single copy of xv6 with all tasks included and ready to compile with `make qemu`**.
- Before submitting, run the following command in your xv6 directory:

  ```
  $ make clean
  ```

  This will remove all compiled files as well as the `obj` directory.
- Submit as a single `.tar.gz` or `.zip` file. `.rar` files will not be accepted.
- We advise you to use git to maintain a working version of your code as you work on the assignment. This will also make it easier for you to collaborate and to compare versions 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.
- Help with the assignment 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.

**Start** by cloning or downloading the assignment skeleton code from the following git repository:
https://github.com/BGU-CS-OS/os242-assignment3-dist

This code is based on the xv6 codebase, which you should be familiar with by now. The changes to the code will be more relevant to the next task, but try to compile it and you'll see that three functions are missing. We will now implement these functions and test them with two userspace programs.

**Implement** a kernel function that, given a virtual address in one process, maps the corresponding physical pages to another 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 function `uvmunmap()` to unmap the pages from the destination process. Note its arguments that are different 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 delete physical pages if they were originally allocated by the process being deleted. When a process exits, the kernel will delete 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 delete the same pages multiple times, which is bad.

**Modify** the `uvmunmap()` function to only delete 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 deleted 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 two userspace programs that use shared memory:

1. `shmem_test1.c`: Allocate memory in a parent process. Write the string "Hello child", share it from the parent to the child, and print it in the child. The parent should wait for the child to finish before exiting. The mappings should be cleaned up properly by the kernel in this test **without explicitly calling unmap**.

2. `shmem_test2.c`: Create a shared memory mapping from a parent process to a child process. Write the string "Hello daddy" in the child and print it in the parent. Unmap the shared memory in the child process and show that `malloc()` can now allocate memory in the child process properly. Print the size of the child process before the mapping, after the mapping, after the unmapping and after the call to `malloc()`. The mappings should be cleaned up properly and the size after the unmapping should be the same as before the mapping.

**Both tests should be able to run one after the other without rebooting the system, which means your 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 implementations work by implementing the test programs `shmem_test1.c` and `shmem_test2.c`.

**For submission:** Modified kernel code that implements the shared memory system and `shmem_test1.c` and `shmem_test2.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 would 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?
18. Why is accessing user-provided data in the kernel dangerous?

# 4 Task 2: Userspace Cryptographic Service

We will now add a cryptographic service to our system, which can be called by a user process to request a cryptographic operation to be performed on its memory in an asynchronous manner.

We've provided you with a system call to request a cryptographic operation:

```
int crypto_op(struct crypto_op* op, uint64 size);
```

It requests a cryptographic operation by adding a request to a queue in the kernel, which is also provided. This queue is used by the server process to process requests from clients, such as `crypto_cli.c`. Read the code in `crypto_cli.c` to understand how this system call is used. Part of this exercise is to develop the ability to understand and use an existing system, which is a crucial skill in software development.

In this task, you will implement the cryptographic server in `crypto_srv.c`. This userspace process is launched directly by the kernel, like `init`. It will process data using the shared memory segments implemented in Task 1. A process like this, that has no direct interaction with the user, is often called a *daemon* in Unix terminology.

To fulfill a request, the server needs to ask the kernel to map the shared memory of a request into its address space. This is done calling a new system call, whose implementation is provided to you:

```
int take_shared_memory_request(void** addr, uint64* size);
```

which maps the shared memory of the request into the server's address space and removes it from the queue. It returns 0 on success and -1 on failure, and passes the virtual address where the new memory was mapped and size of the shared memory segment to the caller via the pointer arguments. Yes, `void**` is a pointer to a pointer. This is not a mistake, it is a way to return a pointer to the server process in userspace.

At this point, the server can access the shared memory and process the request. It knows nothing about the management of the request queue or about shared memory. This is good, because that is not the server's job. For security, when the server process starts, check that the PID is 2 (the kernel will launch it with this PID). Exit if the PID is not 2, e.g., when the server process is started from the shell.

One last call is used to remove the shared memory mapping from the server process:

```
int remove_shared_memory_request(void* addr, uint64 size);
```

This call removes the shared memory mapping associated with the virtual address from the calling process' address space,

Finally, we can implement the cryptographic server. It will accept encryption and decryption requests from other processes in the following format, which is constructed by the client in the shared memory segment:

| uint8 type | uint8 state | uint64 key_size | uint64 data_size |
|---|---|---|---|
| char key[key_size] | | | |
| char data[data_size] | | | |

where `type` is `CRYPTO_OP_TYPE_ENCRYPT` or `CRYPTO_OP_TYPE_DECRYPT`, `state` is `CRYPTO_OP_STATE_INIT` at first, then changes to `CRYPTO_OP_STATE_DONE` when the server is done processing the request. `key_size` is the size of the key in bytes, and `data_size` is the size of the data in bytes. See the implmentation of the client side in `crypto_cli.c` to see how the request structure is created, filled out and acknowledged. Hopefully, the need to map multiple pages is now clear.

The server should call the `take_shared_memory_request()` system call in a loop. When a request is received, the server should use the pointer returned by the system call to access the shared memory and process the request. Use the XOR operation, which is simple and reversible, for both encryption and decryption. As a reminder, encrypting and decrypting with XOR is done by applying the XOR operation bytewise between the key and the data, where the key is repeated as needed to match the length of the data. Use the key to encrypt or decrypt the data in-place.

Due to the memory being shared with the original process, the changes will be visible to the original process as well. After processing the request, the server should set the `state` field to `CRYPTO_OP_STATE_DONE` to indicate that the request has been processed and request the shared memory to be unmapped using the `remove_shared_memory_request()` system call. Since this is user-provided data, check that the sizes are within reasonable limits. When first receiving the request, check that the `state` field is set to `CRYPTO_OP_STATE_INIT` and that `type` is set to either `CRYPTO_OP_TYPE_ENCRYPT`

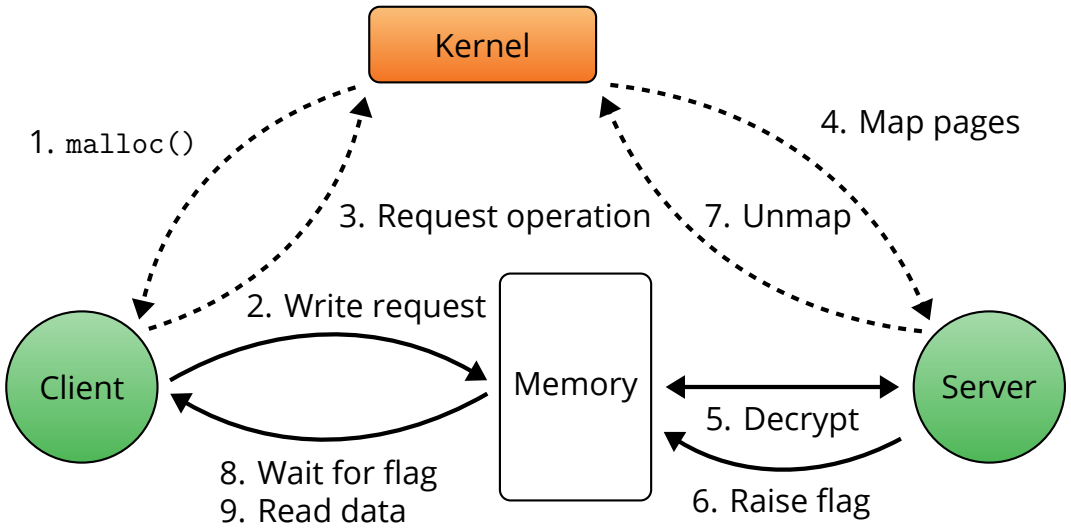
or `CRYPTO_OP_TYPE_DECRYPT`. If an error is detected, change the `state` field to `CRYPTO_OP_STATE_ERROR` to indicate to the client that the request was invalid. This kind of signaling inside the mode of communication, sometimes called *in-band signaling*, is common in communication protocols and interfaces between hardware and software. The opposite is *out-of-band signaling*, where a separate channel is used to signal the state of the communication, is used often as well.

As a final note, while the changes to the memory will eventually be visible to the original process, it does not happen immediately due to memory coherency and caching. Furthermore, memory operations can be reordered by the compiler and CPU, which can cause changes to the `state` field to be visible before the data is written. Since we want to use memory to prevent the client process from accessing the data before the processing is complete, we need to establish a *happens-before* relationship between the data writes and the change to the `state` field. These properties can be achieved by using the following statement in the server process **before any change** to the `state` field:

```
asm volatile ("fence rw,rw" : : : "memory");
```

This is not a full synchronization mechanism, all it does is enforce the order of a single write operation to memory. In this case, we do not need full synchronization between the client and server. This is also a common pattern in modern software and hardware design, especially in a multithreaded and multicore environment.

The complete flow of a decryption request is shown below:




The approach of constructing a generic mechanism in the kernel while breaking off logic into userspace processes, is a common design pattern in operating systems and a recent ongoing trend in modern OS design, notably recent Linux and macOS releases.

If your server works, you should see a message that makes sense.

While reading the client code, note that the client does not wait for the server to finish processing. Normally we would make the client sleep until processing is done, but since this is a long-running operation, the client can do other useful work. Many real-world systems use this principle of *asynchronous programming*, where the client can continue to work while waiting for some task to finish. The most common example is the web browser, which can continue to render the page while waiting for the server to respond to a request, or the node.js framework which uses asynchronous operations as a core principle. In our case, "useful work" amounts to simply burning CPU cycles waiting for the server to finish processing. If you want to simulate more useful work, you can add a loop that prints some messages while waiting. Do not submit this.

The setup where a process repeatedly checks to see if some event has occured rather than being notified about it is called *polling*.

## Task 2 – Implement the cryptographic server

1. Implement the cryptographic server in `crypto_srv.c` that processes requests from the shared memory queue.
2. Check the PID of the server process and exit if it is not 2.
3. Use the provided system calls `take_shared_memory_request()` and `remove_shared_memory_request()` to obtain and release cryptographic operation requests in an infinite loop.
4. Check the request for validity and process it using the XOR operation.
5. Set the `state` field to `CRYPTO_OP_STATE_DONE` when the request is processed.
6. Don't forget to use the `fence` instruction before changing the state of the request!
7. When the client prints the decrypted message, you are done.

**For submission:** The implementation of the cryptographic server in `crypto_srv.c`.

## Questions – not for submission

1. What is the purpose of the cryptographic server in our system?
2. How does the client communicate with the server? How is the request constructed?
3. How does the server process the request?
4. How does the server signal the client that the request has been processed?
5. How does the client know when the server has finished processing the request?
6. What does the server do when there are no requests in the queue? Is this a problem?
7. How does the queue mechanism work? Why do we need it?
8. Is the client aware of the queue mechanism or the existence of the server?
9. Is the kernel aware of the cryptographic operations being performed?
10. Does the server know where or how the request was created and delivered?
11. How would we use this system to build another similar service?
12. Why do we need the fence instruction before changing the state of the request? What does it do?
13. How would we signal completion out-of-band instead of in-band?
14. If we wanted to make the client sleep until the request is complete like we did in previous assignments, how would we do it?

# 5 Conclusion

To conclude, after you put together all of the pieces of this assignment, your operating system should have a service designed to let processes request encryption and decryption of data. Of course, the clients are unaware of the details, as all they see is their memory being modified magically.

As noted in the introduction, the kernel only participates in this system by providing the shared memory machinery and the queue, while having no knowledge of the request-response protocol or the details of the cryptography. Most importantly, it does not access any data or keys, which is a good security practice.

This ends the assignments for this year's OS course. We hope you enjoyed them and learned a lot about operating systems and how they provide the services we rely on.