# Homework 5: Shrimpy shared memory

In this assignment we are going to implement a small form of shared memory between parent and child processes called Shrimpy.

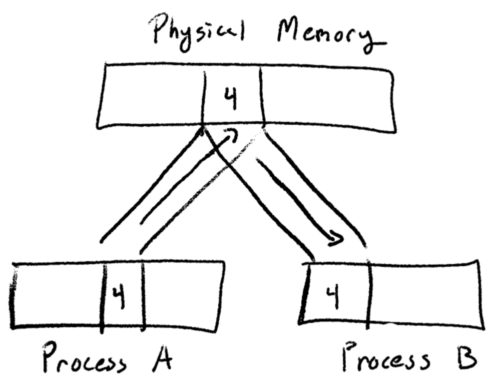
Objectives:

- Understand the concept of shared memory and its implementation in xv6.
- Understand how xv6 uses page tables in memory management.
- Implement a shared memory feature using paging mechanisms.

This lab does *not* depend on the scheduling lab.

## Shared Memory

Processes often need a way of communicating with each other. This is called interprocess communication (IPC). The most performant method of IPC is shared memory. In a shared memory IPC scheme, the same physical memory pages are mapped into two or more processes. It is the responsibility of the operating system to facilitate this by appropriately manipulating the process page tables.



When Process A writes to the shared memory page, the data is "instantly" available to Process B. This is faster than other forms of IPC because there are no system calls required to transfer the data, only the paging hardware is involved.

Both Process A and Process B are allowed to read from and write to the shared memory page.

## Shared Memory on Unix-like systems

There are multiple ways to setup shared memory. One of the simpler approaches is to use `shmget`, `shmat`, and `shmdt`.

The `shmget` function is used to either create a new shared memory segment or access an existing one. Its prototype is:

```
int shmget(key_t key, size_t size, int shmflg);
```

Parameters:

- key: This is a unique identifier for the shared memory segment. Processes that wish to share memory must use the same key.
- size: This is the size of the shared memory segment in bytes. When creating a new segment, this specifies the size of the segment.
- shmflg: This argument specifies flags and permissions for the shared memory segment. Commonly used flags include IPC_CREAT (to create a new segment) and IPC_EXCL (which causes shmget to fail if the segment already exists). Permissions are specified using the same format as for file permissions.

The shmget function returns the shared memory identifier for the segment. If an error occurs, it returns -1.

The `shmat` function is used to attach a shared memory segment to the address space of the calling process. Its prototype is:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Parameters:

- shmid: This is the identifier for the shared memory segment, as returned by shmget.
- shmaddr: This specifies the desired address of the shared memory segment in the process's address space. Normally, this should be NULL, and the system will choose a suitable address.
- shmflg: This argument can be used to specify various flags, such as SHM_RDONLY to attach the segment for read-only access.

The shmat function returns a pointer to the shared memory segment. If an error occurs, it returns (void *) -1.

The shmdt function is used to detach a shared memory segment from the address space of the calling process. Its prototype is:

```
int shmdt(const void *shmaddr);
```

The shmdt function returns 0 on success and -1 on error.

Here is an example of how this API can be used

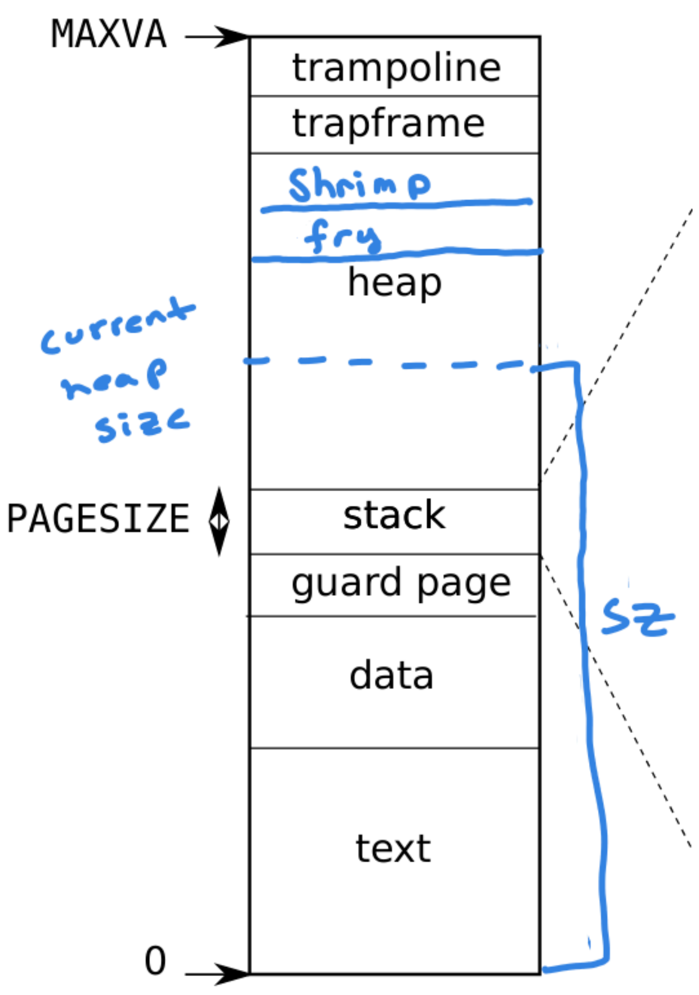
```
#include <sys/ipc.h>  // Provides definitions for the key_t type and the IPC_* constants
#include <sys/shm.h>  // Provides the System V shared memory API, including
shmget(), shmat(), shmdt(), and shmctl()
#include <stdio.h>    // Standard C library for input/output operations
#include <string.h>   // Provides string manipulation functions
#include <unistd.h>   // Provides standard Unix/Linux system calls like fork() and sleep()
#include <sys/wait.h> // Provides wait() function for child process management

#define SHM_KEY 1234    // The unique identifier for the shared memory segment
#define SHM_SIZE 1024   // The size of the shared memory segment

int main() {
    int shmid; // Shared memory ID
    char *shared_mem; // Pointer to the shared memory segment
    pid_t pid; // Process ID

    // Create a shared memory segment.
    // shmget() returns the ID of the shared memory segment.
    // IPC_CREAT | 0666 specifies that the shared memory segment should be created if
it does not exist,
    // and it should be readable and writable by the user, group, and others.
    shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid < 0) {
        perror("shmget"); // If shmget() fails, print an error message and exit
        return 1;
    }

    // Attach the shared memory segment to our address space.
    // shmat() returns a pointer to the shared memory segment.
    shared_mem = shmat(shmid, NULL, 0);
    if (shared_mem == (char*) -1) {
        perror("shmat"); // If shmat() fails, print an error message and exit
        return 1;
    }

    // Fork a new process
    pid = fork();
    if (pid < 0) {
        perror("fork"); // If fork() fails, print an error message and exit
        return 1;
    }

    if (pid == 0) {  // Child process
        // Write a message to the shared memory
        strncpy(shared_mem, "Hello from child!", SHM_SIZE);
    } else {  // Parent process
        wait(NULL); // Wait for the child process to exit
        // Read the message from the shared memory
        printf("Message from child: %s\n", shared_mem);
    }

    // Detach the shared memory segment from our address space
    if (shmdt(shared_mem) == -1) {
        perror("shmdt"); // If shmdt() fails, print an error message and exit
        return 1;
    }

    return 0;  // Exit successfully
}
```

## Shrimpy Design

We will create a much simpler form of shared memory for xv6 called "Shrimpy." The Shrimpy shared memory scheme allows for sharing a single page of memory between parent and child processes only.

All addresses described below are page-aligned.

- Let **fry** denote a well-known virtual address. This is constant for all processes.
- Let **shrimp** denote a well-known virtual address. This is constant for all processes.
- Let **PHY(P, VA)** denote the physical address corresponding to the virtual address VA in the page table of process P.
- Every process *P* has two pages of shared memory.
    - One page is shared with the parent of *P*. This page is mapped at the **shrimp** virtual address, and is called the shrimp page for *P*. If the process does not have a parent, then behavior upon accessing this page is undefined.
    - One page is shared with the children of *P*. All children use the same physical page. This page is mapped at the **fry** virtual address, and is called the fry page for *P*.
    - These are single pages. It is not possible to grow or shrink the amount of shared memory.
- The shrimp and fry pages are mapped at well-known addresses. See address space diagram below.
- The fry page is allocated during process creation time.
- When process *P* forks, creating child *NP*.
- When a process *P* forks and creates a child process NP, **PHY(P, fry)** is mapped to **PHY(NP, shrimp)** in the page table for *NP*.



**Requirements for grading**

The following items are required.

- You must place the shrimp and fry pages at the virtual address described above.
- You must demonstrate that the shared memory feature is functioning correctly.
- You must deallocate memory when a process exits. Demonstrating that shared memory works without appropriately freeing the shared pages will receive partial credit.

## Shrimpy Implementation Hints

The following instructions are meant to be helpful, but are not necessarily requirements. Some items overlap with requirements in the previous section.

## Defining `SHRIMP` and `FRY` well-known addresses

Shrimpy is implemented as a modification of the process address space. Two pages are mapped in at well-known locations, i.e. these locations are the same for all processes.

- Modify `memlayout.h` to add preprocesor directives defining virtual address for the `SHRIMP` and `FRY` pages. Place these pages directly below the trampoline and trap frame pages.

## Physical page allocation

When a process is created, a physical page needs to be allocated for the fry page. It is not necessary to allocate a new physical page for the shrimp page, which was allocated when the parent was created. The `allocproc` function is a reasonable place to implement allocation for a process.

- Use `kalloc` to allocate physical memory for the fry page.

The virtual addresses are well-known, but this physical addresses will change from process to process. Therefore it is necessary to keep track of these physical addresses. ... but where to save this return value from `kalloc`?

- Modify `allocproc` to initialize the physical address of the fry page using the return value of `kalloc`.
- Hint: There is an example nearby that initializes the trapframe page.

## Mapping the fry page

The fry page needs to be mapped into the process address space.

The proc_pagetable function in xv6 is used to create a new page table for a process. We accomplish this by adding an entry in the page table for the newly created process. This entry maps the fry page to the fry virtual address. We will not be touching with the shrimp page yet.

### `proc_pagetable`

The proc_pagetable function is used by allocproc to create a new page table for a process. It takes a single parameter: a pointer to a process `struct proc`.

The function returns a pointer to the page table for a newly created process. If we want to add mappings for every process, then proc_pagetable is a natural place to do this.

### `mappages`

The `mappages` function can be used to map physical pages into the process. It takes five parameters

1. A pointer to a page table to update.
2. A virtual address `va` to start mapping at. This address must be page-aligned, i.e. a multiple of the page size in bytes. This is a virtual address in the process address space that we want to map to a physical address.
3. The size of the memory region to map (in bytes).
4. A physical a physical address `pa` to map to the virtual address `va`. This address must be page-aligned.
5. Permissions flags for the mapping.

`mappages` maps a range of virtual addresses to physical addresses. The range is 'size' bytes, and each mapping has permissions specified by 'flags'. The function updates the page table to reflect these mappings. It's important to note that both 'va' and 'pa' must be page-aligned and 'size' must be a multiple of the page size.

- In `proc_pagetable` use `mappages` to map physical pages into the process virtual address space. The page should be readable, writable, and user-accessible.

At this point, if you were to try to boot xv6, you would get a kernel panic. This is because the fry page is still marked as valid in the page table when the process exits. The xv6 kernel has a sanity check when it is freeing memory that will cause a panic if a page is marked as valid when it is freed.

- Unmap the fry page in `proc_freepagetable`. Read the code to understand the parameters to the `uvmunmap` function.

## Mapping the shrimp page

All processes (except init) have a parent process. When a process forks, the child process shares a page with its parent process. When a new child process is created by `fork` in `proc.c`, the fry page of the parent process becomes the shrimp page of the child process.

Fork copies the entire process memory, including the page table. In fork,

1. Remove the mapping for the 'FRY' page from the page table of the newly created child process. This mapping is a copy from the parent, but it's no longer relevant for the child.
2. Create a new mapping for the fry page in the child's page table. This mapping should point to the physical memory that was allocated for the child's 'FRY' page. Use the mappages function for this, and ensure the page is mapped with read, write, and user permissions (PTE_R | PTE_W | PTE_U)."

These pages should be mapped with flags that are appropriate for user memory. Look at the user memory functions for examples.

- Unmap the existing fry page in the new child process. This was copied from the parent.
- Map the newly allocated fry page for the child process.
- Unmap the shrimp page for the child process. This was copied from the parent.
- Map the fry page from the parent process as the shrimp page for the child.

### Free

When a process terminates, you need to deallocate the memory associated with its fry page. There is a flag to `uvmunmap` that determines if a page is unmapped and freed or just unmapped.

- Free the fry page
- Don't free the shrimp page, as that is still being used by the parent.

### Test it

Write a userspace xv6 program that demonstrates shared memory between a parent and child process. Here's a high-level description of what it needs to do:

1. The parent process retrieves a pointer to its own shared memory space.
2. The parent process writes a specific value into this shared memory space. This can just be a number.
3. The parent process prints out the value in the shared memory space to confirm it was written successfully.
4. The parent process creates a child process using fork.
5. In the child process:

- The child process retrieves a pointer to its shared memory space. This shared memory space should be the same physical memory that the parent process used.
- The child process prints out the value in the shared memory space to confirm that it sees the same value that the parent wrote.

## Deliverables

1. Create a `hw5-shrimpy` branch that includes your code.
2. Push your branch to GitHub and open a pull request.