# Memory Mapping with mmap System Call

OS Lab UG TAs

December 14, 2024

## 1 Overview

In this section, we have two main tasks. First, we define the `mmap` system call and its semantics, which allow us to expand the virtual address space of a process without allocating any pages. Second, we handle traps to ensure the process accesses the correct memory before allocating the pages on demand.

## 2 Defining the Syscall

Similar to *Part-A*, we make changes to the following files: `user.h`, `usys.s`, `syscall.h`, and `syscall.c` to define the `mmap` system call. Since these changes are almost identical to those in *Part-A*, they will not be elaborated here. However, the changes in `sysproc.c` differ, as this is where the actual syscall is implemented. We will base our implementation on how the `sbrk` system call is implemented.

### 2.1 Understanding the `sbrk` System Call

#### 2.1.1 Changes in `sysproc.c`

The `sbrk` system call takes an argument specifying the amount by which to grow the process's address space, and this argument cannot be negative. Inside `sysproc.c`, the argument can be accessed using the `argint` function. The system call then calls `growproc`.

#### 2.1.2 `Growproc`

The `growproc` function is defined in `proc.c` and returns the address of the end of the process's address space before `sbrk` was called. It then calls `allocuvm` to allocate additional pages, followed by `switchuvm`, which updates the process's page directory.

#### 2.1.3 `Allocuvm`

The `allocuvm` function allocates physical frames to extend the process's address space. It takes as arguments the page directory, the old size of the process, and the new size. If the new size is smaller than the old size, `deallocuvm` is called. Otherwise, the function allocates as many physical frames as needed:

1. A new frame is allocated using `char *mem = kalloc();`, where `kalloc` returns a free physical frame if available.
2. The newly allocated frame is cleared using `memset(mem, 0, PGSIZE)`.
3. The frame is mapped to the process's address space using `mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W | PTE_U)`. The `PTE_U` flag ensures user-space accessibility, and `PTE_W` allows write access. The `mappages` function also ensures that the page is marked present by setting the `PTE_P` flag.

Finally, `allocuvm` returns the new size, indicating the operation was successful.

### 2.2 Implementing the `mmap` Syscall

For the `mmap` system call, we expand the process's virtual address space without allocating physical memory. This can be done entirely within `sysproc.c`, as we don't need to allocate physical frames. We acquire the memory size to be added, which we'll call `allocMem`, using the `argint` function. We ensure the value is positive and a multiple of the page size. The current process size is stored, and the size is then increased by `allocMem`. The function returns the old process size.

# 3 Trap Handling

Simply defining the system call is not enough for `mmap` to work. When a process attempts to access the newly allocated memory, a trap will occur because no valid pages (present, user-accessible, and optionally writable) are assigned to those pages. The trap handler needs to verify that the memory being accessed was allocated by `mmap`. We do this by ensuring that the accessed address is smaller than the process's size.

## 3.1 Modifications Inside `trap.c`

We acquire the trap-causing address using `rcr2()`. We then check if the address is accessible by the user by ensuring that its value is less than the size of the process. If the condition is met, we invoke the `allocPage` function, defined in `defs.h` and implemented in `vm.c`. This function is responsible for allocating physical frames, which can only be done in `vm.c`.

## 3.2 Modifications in `vm.c`

We implement the `allocPage` function as follows:

```
void allocPage(int addr, pde_t* pgdir) {
  char *mem = kalloc();
  if(mem == 0){
    panic("Memory");
  }
  memset(mem, 0, PGSIZE);
  if(mappages(pgdir, (char*)addr, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0){
    kfree(mem);
    panic("mappages");
    return;
  }
}
```

The steps performed in this function are:

1. Memory is allocated using `kalloc`.
2. The allocated frame is cleared using `memset`.
3. The page table mapping is created using `mappages`.

Finally, either `switchuvm(myproc())` is called at the end of `allocPage`, or it can be placed at the end of `trap.c`.