CS179F: Projects in Operating System

Memory mapped files

Emiliano De Cristofaro and Lian Gao

Virtual Address Space

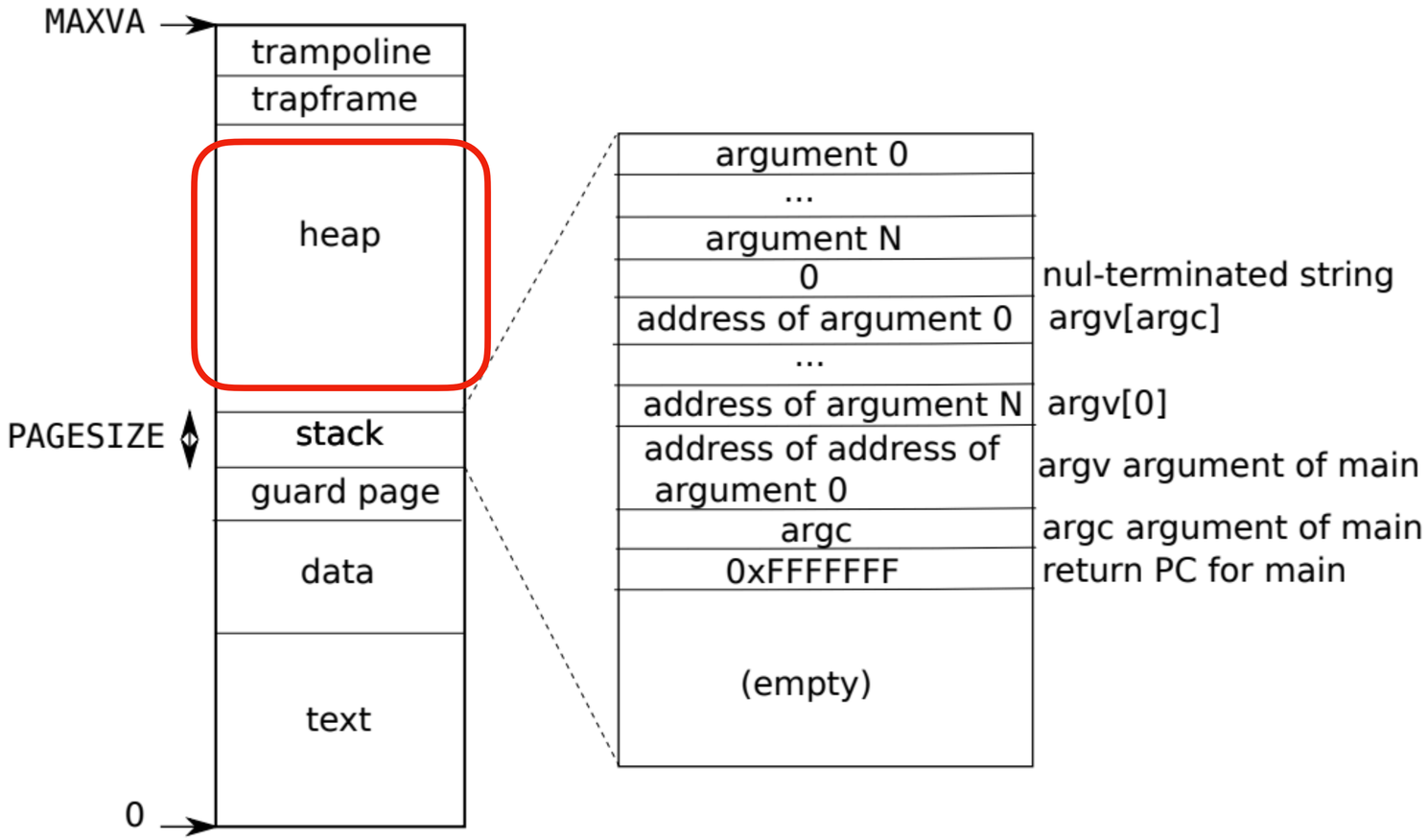


Figure 3.4: A process's user address space, with its initial stack.

Virtual Memory Allocation

- `sbrk()` - a single memory region
- `mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - Multiple (segments) of memory regions
- `munmap(void *addr, size_t length)`
  - Can unmap partially of a region

Virtual Address Space Management

- Question: how to keep track of memory regions?
- VMA: virtual memory area
  - What metadata is necessary?
  - `mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`

Virtual Address Space Management

- Question: how to keep track of memory regions?
- VMA: virtual memory area
  - Q1: what metadata is necessary?
    - [start, end) - `[addr, addr + length)`
    - permissions - `prot`
    - behaviors - `flags`
    - file object, offset - `fd, offset`

Virtual Address Space Management

- Q2: how to manage vmas (e.g., how to find a vma)?
  - Array (fixed size)
  - Linked list

Virtual Address Space Management

- Q3: how to handle lazy allocation (when a page fault happens, how do we know it's due to lazy allocation)?
  - How to improve the performance?
    - Heap (sorted tree) - red-black tree

Virtual Address Space Management

- Q4: how to allocate vma?
  - Find an unallocated virtual space region that is large enough
    - Recall heap allocation
- Q5: how to handle munmap?
  - Shrink, split, or free

Virtual Address Space Management

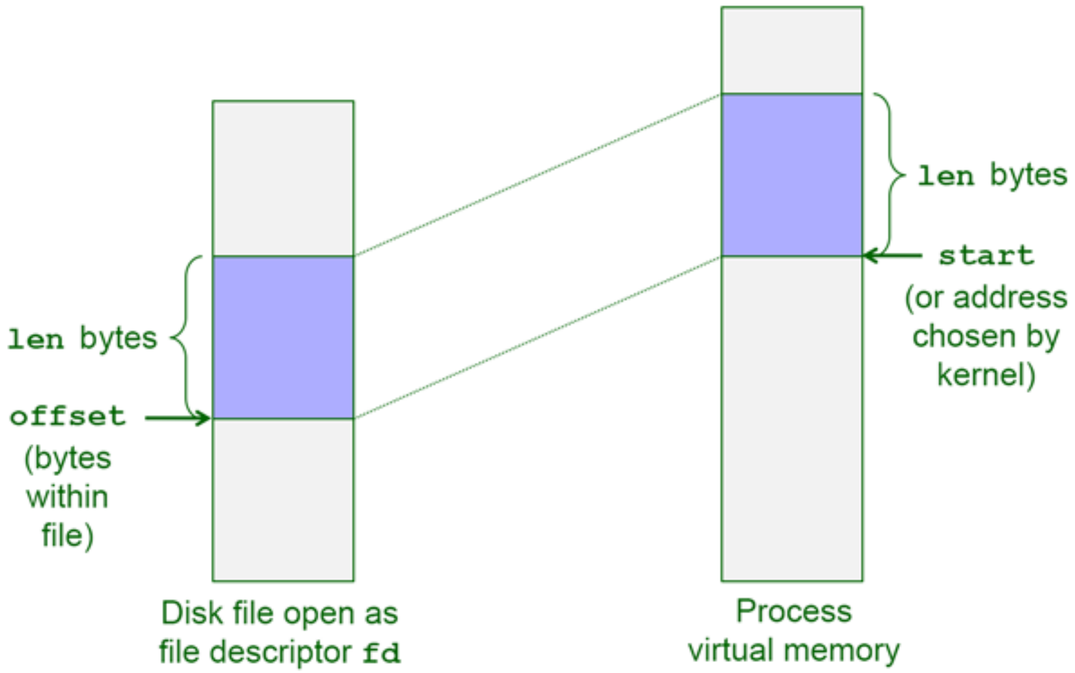
- Q6: how to handle a memory mapped file?
  - SHARED vs PRIVATE
  - offset?
  - lazy allocation
  - buffer?

Step 0: Pull the code for Lab 5

- In your xv6-riscv code base, run
  - $ git fetch
  - $ git checkout mmap
- Make sure this checkout to this branch new branch, and then
  - $ make clean
  - $ make qemu
- Read lab instructions carefully
  - https://github.com/emidec/cs179f-fall23/blob/xv6-riscv-fall23/doc/lab5.md
  - You will use the knowledge/skills you learned in Lab2 and Lab4
- Important dates
  - 12/15/2023 1:59PM (NO EXTENSIONS)

mmap() and munmap() system call

```
void *mmap(void *start, size_t len,
           int prot, int flags, int fd, off_t offset)
```



Reference: https://www.clear.rice.edu/comp321/html/laboratories/lab10/

- mmap() maps files (or devices) into virtual memory of the calling process where **`start`** specify the starting address of the mapping and **`len`** specifies the length.
- In this lab, we assume that:
  - **`start`** will always be zero, meaning that the kernel will decide the address at which to map the file
  - **`prot`** indicates if the memory mapped should be readable and(or) writable, it can be **`PROT_WRITE, PROT_READ`** or both.
  - **`flags`** can be
    - **`MAP_SHARED:`** modifications should be written back
    - **`MAP_PRIVATE`**: modifications should not be written back
  - **`offset`** always be 0. The mapping always starts from the beginning of the file.
- **`munmap(void* start, size_t len)`**
  - Remove the mapping. If **`MAP_SHARED`**, write back the modifications
  - You can assume that it will either unmap at the start, or at the end, or the whole region (but not punch a hole in the middle of a region).
- Return value:
  - On success, **`mmap`** returns the pointer to the mapped area, on error, it returns -1
  - On success, **`munmap`** returns 0, on error, returns -1

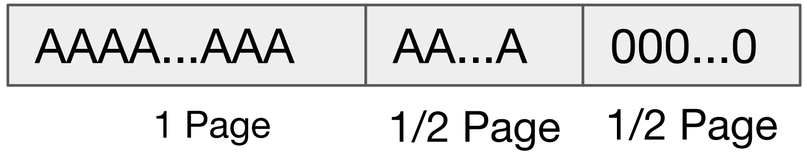
Your Goal

You should implement enough mmap and munmap functionality to make the `mmaptest` test program work.

If `mmaptest` doesn't use a mmap feature, you don't need to implement that feature.

mmaptest walkthrough

Test File




```
 99   makefile(f);
100   if ((fd = open(f, O_RDONLY)) == -1)
101     err("open");
```

Test 1: Basic checks

```
char *p = mmap(0, PGSIZE*2, PROT_READ, MAP_PRIVATE, fd, 0);
if (p == MAP_FAILED)
  err("mmap (1)");
_v1(p);
if (munmap(p, PGSIZE*2) == -1)
  err("munmap (1)");
```

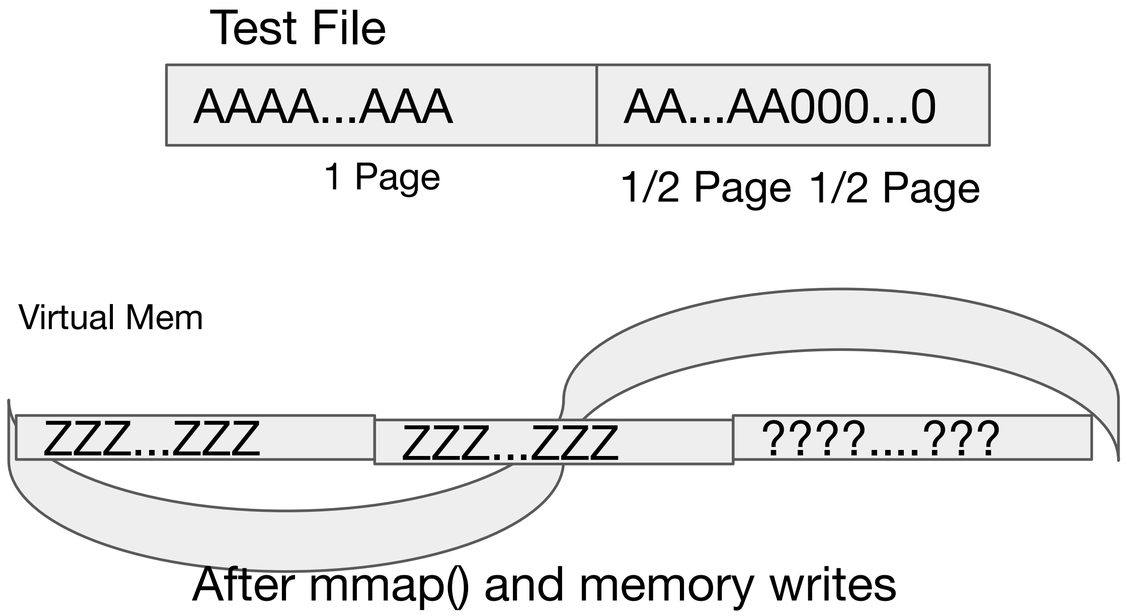
Test 2: Private writable mapping for a file opened read-only is allowed

```
// should be able to map file opened read-only with private writable
// mapping
p = mmap(0, PGSIZE*2, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (p == MAP_FAILED)
  err("mmap (2)");
if (close(fd) == -1)
  err("close");
_v1(p);
for (i = 0; i < PGSIZE*2; i++)
  p[i] = 'Z';
if (munmap(p, PGSIZE*2) == -1)
  err("munmap (2)");
```

Test 3: Reject a shared readable/writable mapping of a file opened read-only

```
// check that mmap doesn't allow read/write mapping of a
// file opened read-only.
if ((fd = open(f, O_RDONLY)) == -1)
  err("open");
p = mmap(0, PGSIZE*3, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (p != MAP_FAILED)
  err("mmap call should have failed");
if (close(fd) == -1)
  err("close");
```

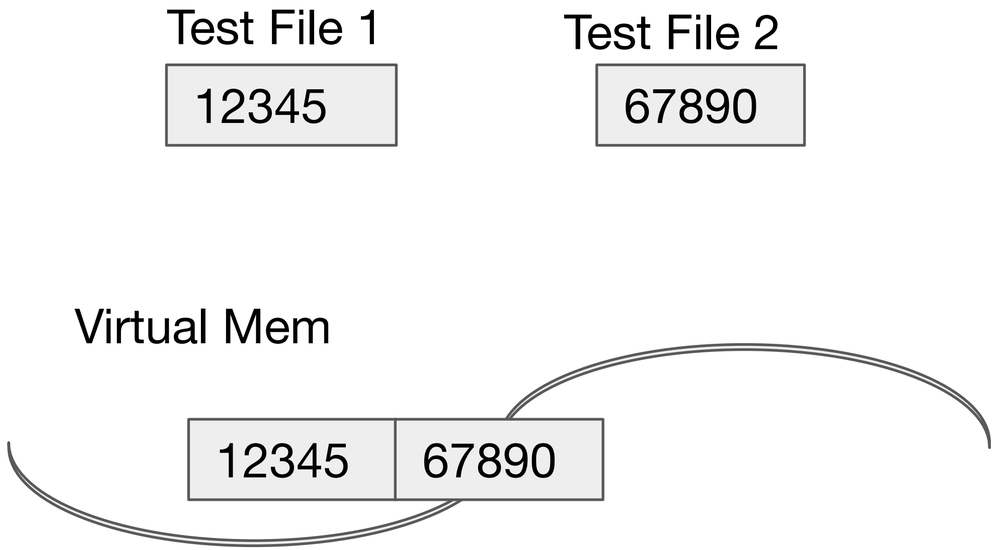
mmaptest walkthrough



Test 4: munmap

```
147   // check that mmap does allow read/write mapping of a
148   // file opened read/write.
149   if ((fd = open(f, O_RDWR)) == -1)
150     err("open");
151   p = mmap(0, PGSIZE*3, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
152   if (p == MAP_FAILED)
153     err("mmap (3)");
154   if (close(fd) == -1)
155     err("close");
156
157   // check that the mapping still works after close(fd).
158   _v1(p);
159
160   // write the mapped memory.
161   for (i = 0; i < PGSIZE*2; i++)
162     p[i] = 'Z';
163
164   // unmap just the first two of three pages of mapped memory.
165   if (munmap(p, PGSIZE*2) == -1)
166     err("munmap (3)");
167
168   // check that the writes to the mapped memory were
169   // written to the file.
170   if ((fd = open(f, O_RDWR)) == -1)
171     err("open");
172   for (i = 0; i < PGSIZE + (PGSIZE/2); i++){
173     char b;
174     if (read(fd, &b, 1) != 1)
175       err("read (1)");
176     if (b != 'Z')
177       err("file does not contain modifications");
178   }
179   if (close(fd) == -1)
180     err("close");
181
182   // unmap the rest of the mapped memory.
183   if (munmap(p+PGSIZE*2, PGSIZE) == -1)
184     err("munmap (4)");
```

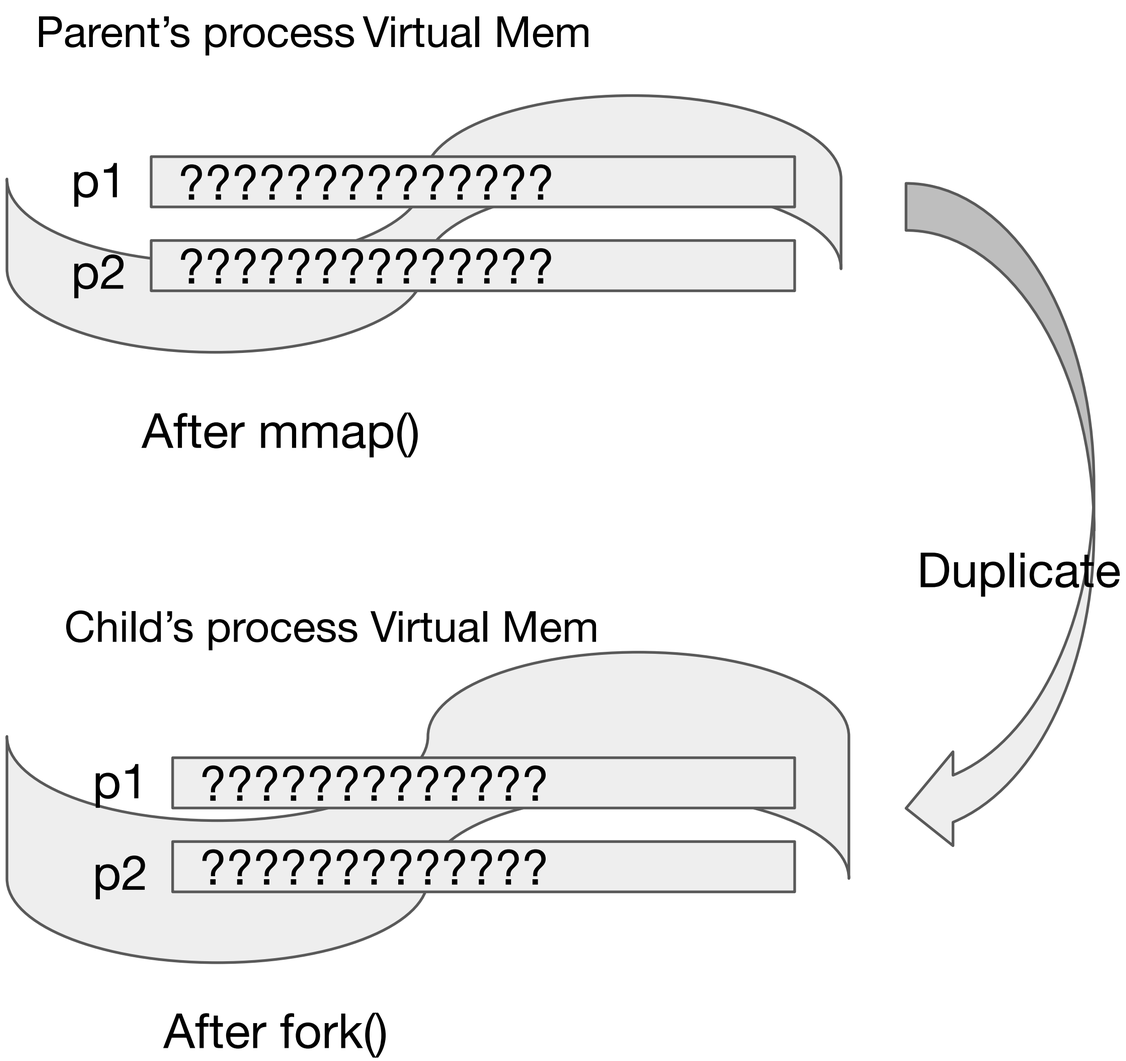
mmaptest walkthrough



Test 5: map two files

```
186   //
187   // mmap two files at the same time.
188   //
189   int fd1;
190   if((fd1 = open("mmap1", O_RDWR|O_CREATE)) < 0)
191     err("open mmap1");
192   if(write(fd1, "12345", 5) != 5)
193     err("write mmap1");
194   char *p1 = mmap(0, PGSIZE, PROT_READ, MAP_PRIVATE, fd1, 0);
195   if(p1 == MAP_FAILED)
196     err("mmap mmap1");
197   close(fd1);
198   unlink("mmap1");
199
200   int fd2;
201   if((fd2 = open("mmap2", O_RDWR|O_CREATE)) < 0)
202     err("open mmap2");
203   if(write(fd2, "67890", 5) != 5)
204     err("write mmap2");
205   char *p2 = mmap(0, PGSIZE, PROT_READ, MAP_PRIVATE, fd2, 0);
206   if(p2 == MAP_FAILED)
207     err("mmap mmap2");
208   close(fd2);
209   unlink("mmap2");
210
211   if(memcmp(p1, "12345", 5) != 0)
212     err("mmap1 mismatch");
213   if(memcmp(p2, "67890", 5) != 0)
214     err("mmap2 mismatch");
215
216   munmap(p1, PGSIZE);
217   if(memcmp(p2, "67890", 5) != 0)
218     err("mmap2 mismatch (2)");
219   munmap(p2, PGSIZE);
220
221   printf("mmap_test OK\n");
222 }
```

mmaptest walkthrough



Test 6: forktest

```
225 // mmap a file, then fork.
226 // check that the child sees the mapped file.
227 //
228 void
229 fork_test(void)
230 {
231   int fd;
232   int pid;
233   const char * const f = "mmap.dur";
234
235   printf("fork_test starting\n");
236   testname = "fork_test";
237
238   // mmap the file twice.
239   makefile(f);
240   if ((fd = open(f, O_RDONLY)) == -1)
241     err("open");
242   unlink(f);
243   char *p1 = mmap(0, PGSIZE*2, PROT_READ, MAP_SHARED, fd, 0);
244   if (p1 == MAP_FAILED)
245     err("mmap (4)");
246   char *p2 = mmap(0, PGSIZE*2, PROT_READ, MAP_SHARED, fd, 0);
247   if (p2 == MAP_FAILED)
248     err("mmap (5)");
249
250   // read just 2nd page.
251   if(*(p1+PGSIZE) != 'A')
252     err("fork mismatch (1)");
253
254   if((pid = fork()) < 0)
255     err("fork");
256   if (pid == 0) {
257     _v1(p1);
258     munmap(p1, PGSIZE); // just the first page
259     exit(0); // tell the parent that the mapping looks OK.
260   }
261
262   int status = -1;
263   wait(&status);
264
265   if(status != 0){
266     printf("fork_test failed\n");
267     exit(1);
268   }
269
270   // check that the parent's mappings are still there.
271   _v1(p1);
272   _v1(p2);
273
274   printf("fork_test OK\n");
275 }
276
```

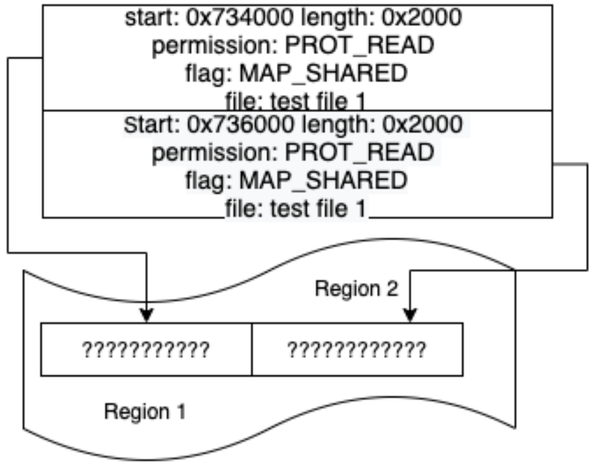
Design (1/2)

- You must use lazy allocation (lab 2). That is, mmap should not allocate physical memory or read the file. Instead, do that in page fault handling code in (or called by) usertrap. Reasons: 1) performance 2) mapping large file (larger than physical memory)
- Keep track of what **`mmap`** has mapped for each process. Define a structure (referred as **`VMA`** in the following discussions) recording address, length, permission, file, etc. for a virtual memory region created
- In **`mmap()`**, find an unused region in the process's address space in which to map the file, and add a **`VMA`** to the process's table of mapped regions. The VMA should contain a pointer to a struct file for the file being mapped; mmap should increase the file's reference count so that the structure doesn't disappear when the file is closed

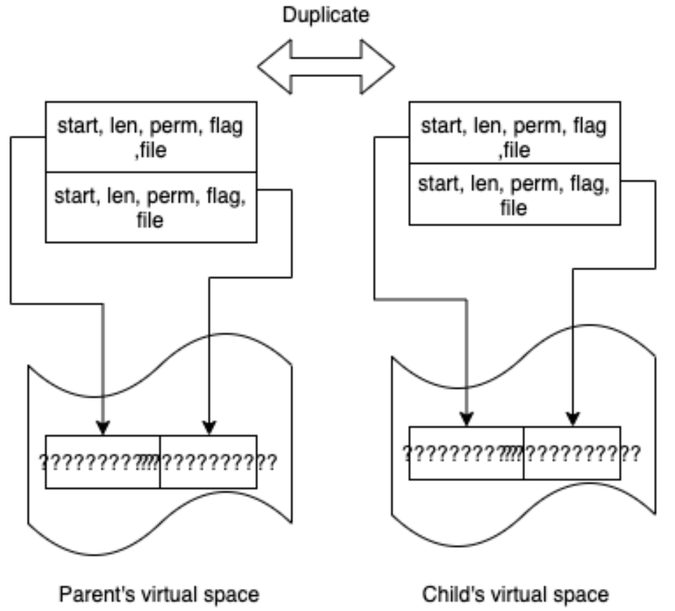
Design (2/2)

- In **`usertrap()`**, read 1 page of the relevant file into that page, and map it into the user address space
- Implement **`munmap()`**: find the **`VMA`** for the address range and unmap the specified pages. If **`munmap`** removes all pages of a previous mmap, it should decrement the reference count of the corresponding struct file. If an unmapped page has been modified and the file is mapped **`MAP_SHARED`**, write the page back to the file.
- Modify **`exit()`** to unmap the process's mapped regions as if munmap had been called
- Modify **`fork()`** to ensure that the child has the same mapped regions as the parent. Don't forget to increment the reference count for a **`VMA`**'s struct file

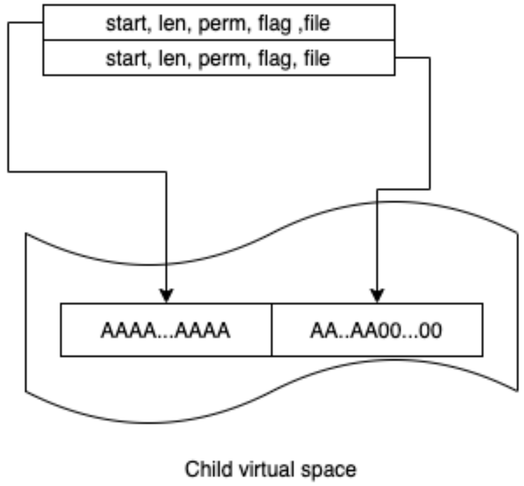
Running example based on `fork_test()`



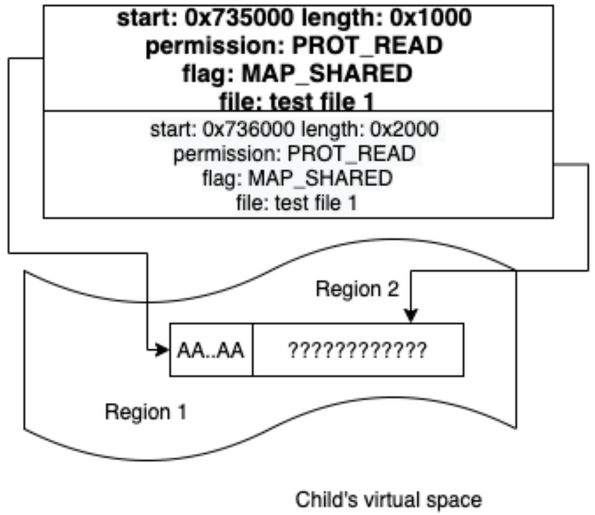
After call mmap()



After call fork()



After usertrap()



After munmap() for the first page

Implementation step 0: Add flags/definitions for the syscalls

- Add **`$U_mmaptest`** in **`Makefile`** to add mmaptest test.
- Add **`entry("mmap"); entry("munmap");`** in **`user/usys.pl`**
- Add the following function definitions in **`user/user.h`**
  - **`void* mmap(void*, unsigned int, int, int, int, unsigned int);`**
  - **`int munmap(void*, unsigned int);`**
- Add the following definitions in **`kernel/syscall.h`**
  - **`#define SYS_mmap    23`**
  - **`#define SYS_munmap    24`**
- Add the following mapping in **`kernel/syscall.c`**
  - **`[SYS_mmap]     sys_mmap,`**
  - **`[SYS_munmap]     sys_munmap,`**

Implementation step 0: Add flags/definitions for the syscalls

- Add the following definitions in **kernel/syscall.c**
  - `extern uint64 sys_mmap(void);`
  - `extern uint64 sys_munmap(void);`
- Add the following definitions in `kernel/fcntl.h`
  - `#define PROT_READ 0x000`
  - `#define PROT_WRITE 0x001`
  - `#define MAP_PRIVATE 0x000`
  - `#define MAP_SHARED 0x001`
- Your implementations can go to kernel/sysfile.c
  - `uint64_t mmap() { // your implementation }`
  - `Uint64_t munmap() { // your implementation }`

Implementation: defining and allocating structure vma

- Define your **`VMA`** structure (recording start, length, permission, flags and file for each mapped memory range). You can define it in kernel/proc.h

- Add a table in struct proc of all the VMAs for a process, for example
  - **`struct vma* vma_table[4];  //vma regions`**

- Since the xv6 kernel doesn't have a memory allocator in the kernel, it's OK to declare a fixed-size array of VMAs and allocate from that array as needed. A size of 16 should be sufficient
  - Hint: look at how file structure is allocated in **`kernel/file.h`**

Implementation: mmap()

- Check and respond to different permissions, flags.
- Allocate a **`vma`**
- Update the **`vma`** with addr, length, file, permissions, etc.
- Add the **`vma`** to the process's vma table
- Call **`filedup`** to increase the reference count of the file

Implementation: usertrap()

- Given a faulty address, check if it belongs to a VMA region, and find the VMA region if it does
- Deciding the offset in the file at which the data copy should start by looking at the faulty address and the VMA's start address
- Allocate a physical page
- Call **`readi()`** to transfer the data from file to the physical page
- Map the physical map to user's virtual memory space by calling **`mappagers()`**

Implementation: munmap() exit() and fork()

- Find the VMA for the address range and unmap the specified pages (hint: use **`uvmunmap()`**).
- If all pages are removed, decrease the reference count of the corresponding struct file (hint: look at **`fileclose`**)
- If an unmapped page has been modified and the file is mapped **`MAP_SHARED`**, write the page back to the file (hint: look at **`filewrite()`**)
- Modify **`exit()`**, which removes all mapped memory regions for a process
- Modify **`fork()`** to ensure that the child has the same mapped regions as the parent. Don't forget to increment the reference count for a VMA's **`struct file`**
- You will probably need to modify **`uvmcopy() and uvmunmap()`** in a way similar to what you've done in Lab2.

Self-Assessment

```
$ make qemu-gdb
(3.6s)
  mmaptest: mmap_test: OK
  mmaptest: fork_test: OK
usertests:
$ make qemu-gdb
OK (31.7s)
Score: 100/100
```

- You can run each test individually
  - $ make qemu
  - $ mmaptest
  - $ usertests
- Or use
  - $ make grade