Xv6 Virtual Memory and Sharing
A Team Project

• Either individual
• Or find a team member to work together
• At most 2 members in a team
• Due: Mar. 30, 5:00PM
Shared Memory

P1 virtual memory

Physical memory

P2 virtual memory
What You Do

• Implement a system call called shmgetat
• Allocates a shared memory segment

SYNOPSIS
Void *shmgetat(int key, int num_pages)

• Combination of the two Linux system calls: shmget() followed by shmat()
• if processes call **shmgetat()** with the same **key** for the first argument, then they will share the specified number of physical pages
• Using different keys in different calls to **shmgetat()** corresponds to different physical pages
More Description

• **shmgetat** returns the virtual address of the shared pages to the caller, so the process can read/write them

• **shmgetat** should map the shared physical pages to the next available virtual pages, starting at the high end of that process' address space

• 0x80000000 and above is reserved for the kernel
Example

• when a process calls `shmgetat(0, 1)`:  
  – The OS should map 1 physical page into the virtual address space of the caller, starting at the very high end of the address space  
  – These pages should be zero-filled initially
• If another process then calls `shmgetat(0, ANY_VALUE)`:  
  – this process should also get that same 1 page mapped into its virtual address space (possibly at a different virtual address)  
  – Second argument is ignored if the key has been already existing
• The two processes can then each read and write to this page and thus communicate
• A third or fourth process calls `shmgetat(0, ANY_VALUE)`  
  – Same page mapped into their address spaces as well
Another example

• Another key is used, `shmgetat(1, 3)`
  – This corresponds to a new shared region

• if any other process calls `shmgetat(1, 3)`
  – OS will map 3 (new) physical pages into the address space of the calling process
  – Associate these 3 pages with key value 1
  – Subsequent calls that use `key=1` will map these three pages into the calling process' address space
Another System Call (Optional)

- **int shm_refcount(int key)**
  - Returns, for a particular key, how many processes currently are sharing the associated pages
  - If a process exits, you need to decrement this reference count
  - If the reference count for a key goes to 0, then all state associated with those pages should be freed

- **if the reference count for a key goes to 0, call shmgetat(key, SOME_VALUE)**
  - This call will be treated as a new shared segment with a new number of shared physical pages
  - the pages appropriately initialized to zero
Desirable Features/Tips

- Same process can call shmgetat more than once
- Keys are globally visible, not for a particular process
- When a fork is called, every shared region has to be accessible to the new process **without** it needing to call shmgetat()
- Shared regions have to be mapped starting from the very **end** of calling process's address space
User Process Code Example/References

- http://eng.utah.edu/~cs5460/OS-s14/xv6_shared_memory.html
- http://pages.cs.wisc.edu/~dusseau/Classes/CS537/Fall2016/Projects/P3/p3b.html
About Grading

• Implemented shared memory (80%)
• Implemented key (rest 20%)
• These two features are optional:
  – Same process can call shmgetat more than once
  – When a fork is called, every shared region has to be accessible to the new process **without** it needing to call shmgetat()
More words about requirements

• As long as you can show a working version that is able to share pages and identify shared pages by keys, you’re good

• For example, the following items are not required:
  – Read/Write Synchronization between processes
  – Dynamically allocate pages (you can allocate a page pool, and use them when in need)
  – Reclaiming shared pages
  – A bug free implementation
  – Solve efficiency issues
  – Other corner cases
Xv6 memory Management

- Translate a kernel VA to a PA
  - V2P(a) (((uint) (a)) - KERNBASE)
- Translate a PA to a kernel VA
  - P2V(a) (((void *) (a)) + KERNBASE)
One way to do page sharing?

- Kernel Pages (Virtual Addresses) $\geq 0x80000000$ (KERNBASE)
- Request a page `kalloc()`
- Return kernel virtual address
- `Shmgetat(k, num)`
- Return address `addr1`
- Return address `addr2`
- Change $*addr1 = x$
- Physical Memory (start from 0)
- Read $*add2$ out
- $addr1 == addr2$?
- $*addr2 == x$?
Work Items for sharing pages

Initialization
• Define a global variable SharedMem[PAGES]
• Allocate pages \((kalloc)\) using a function ShareMemInit() in vm.c
• Assign the virtual addresses to SharedMem[PAGES] during booting
• Call ShareMemInit() during booting in \texttt{main.c}

Add System Call \texttt{shmgetat}(key, num)
• Change these files: defs.h, syscall.c, syscall.h, sysproc.c, user.h
• Implement it in vm.c

Implementation
• Start from top of virtual address in user space (0x80000000)
• The first virtual address should be the starting point of the page: \(0x80000000\text{-PAGE}\text{SIZE}(4K, 0x1000) = 0x7fffff000\)
• Walk through the page table and map virtual addresses to their physical addresses (walkpgdir and mappages functions)
• Physical addresses can be obtained from SharedMem[PAGES] (V2P function)
One way to implement key

<table>
<thead>
<tr>
<th>index</th>
<th>Virtual Addr</th>
<th>KN[KEYS]</th>
<th>Total Num</th>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8dfbb000</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>8dfba000</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>8dfb9000</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8dfaa000</td>
<td>3</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>8dfc8000</td>
<td>4</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>8dffe000</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8dfbe000</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8dfbf000</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8dfc0000</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8dfc1000</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Work items to enable keys

- Define a global variable g_k and a global array KN[KEYS] to track which pages have been shared.
- Manipulate g_k and KN to map their items to allocated memory pages ShareMem[PAGES].
- Key  Num  KN[key]
  
  0   3   3
  1   2   5
  2   1   6
  3   1   7

  KN[key] keeps the total number of pages up to now shared by different processes, the values can be used as indices to track which items in ShareMem[PAGES] have been shared. For example, key 3 has one item shared, which is corresponding to index 6 in ShareMem[PAGES].
Testing

Test Page Sharing

• Write a user program in directory userspace to call the system call
• Fork() is needed to test the shared memory between processes
• One process assigns a value to a location in the shared memory and another process read it out, the two values should be the same

Test Key Identification

• Process 1 uses key 0
• Process 2 uses key 1 and writes to the shared page X
• process 3 uses key 1 and read the shared page value, which should be X as well
• You can assume key will be used in a continuous way
One Testing Scenario

P1
*p = 0
*x = 100
Fork()
WAIT()
P2
*p = Shmgetat(0, 2)

P2
Fork()
WAIT()
P3
*p = Shmgetat(0, 2)
p = Shmgetat(1, 3)
Fork()
WAIT()
p = Shmgetat(1, 3)
P3
*p = 40
*x = 140
Printf(*p) = ?
Printf(*x) = ?

Exit(): needs to make sure shared pages are not freed!!
*p = 0
*x = 100
int p = 0;
int* test;
int t = 100;
int* x = &t;

if(fork() == 0){
    if(fork() == 0){
        //Current limitation: key needs to be used continuously, 0 first, then 1, 2...
        p = shmget(0, 3);
        test = (int*)p;
        *test = 30;
        *x = 130;
        printf(1, "Grand Child: %x, %d, %x, %d\n", test, *test, x, *x);
    }
    else{
        wait();
        p = shmget(1, 2);
        printf(1, "Share key 1, 2\n");
        test = (int*)p;
        *test = 40;
        *x = 140;
        printf(1, "Child: %x, %d, %x, %d\n", test, *test, x, *x);
    }
}
else{
    wait();
    p = shmget(1, 4);
    printf(1, "Share key 1 \n");
    test = (int*)p;
    printf(1, "Parent: %x, %d, %x, %d\n", test, *test, x, *x);
}
exit();
Usage of Functions

• V2P(a) (((uint) (a)) – KERNBASE)
  – Translate a kernel VA to a PA
• P2V(a) (((void *) (a)) + KERNBASE)
  – Translate a PA to a kernel VA
• Char* kalloc(void)
  – Allocate one 4096-byte page of physical memory
  – Returns a pointer that the kernel can use
  – Returns 0 if the memory cannot be allocated
• static int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
  – Create PTEs for virtual addresses starting at va that refer to physical addresses starting at pa
  – va and size might not be page-aligned
  – Perm: PTE_U, PTE_W, PTE_P...
More Functions

- **walkpgdir(pde_t *pgdir, const void *va, int alloc)**
  - Search a PTE or create a new PTE
  - If allocate == 0, return the address of the PTE in page table pgdir that corresponds to virtual address va
  - If alloc!=0, create a new page and initialize a new PTE based on va, set the new PTE with permission PTE_P, PTE_W and PTE_U

- **Int deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)**
  - Deallocate user pages to bring the process size from oldsz to newsz
  - **You need to modify this function to make sure shared pages won’t be freed when one process exits**

- **pde_t* copyuvm(pde_t *pgdir, uint sz)**
  - Given a parent process's page table, create a copy of it for a child
  - only copies all contents from its parent address space, such as stack and heap, to a child address space
  - Generates a new PD and PTs but doesn't copy PTE tries, remapping all shared pages needs to be done
  - Modify it to make sure a child can inherit shared pages from parent, **optional to modify**
Run some code if we have time

• Show a testing user program
• Good video for the page sharing part, it won’t waste your time
  – https://www.youtube.com/watch?v=YPfa75AGyGE