Part 2: System calls to share a memory page

Written by Xue Zhang A20484478, and Xiaoxu Li A20522966

1. Introduction

Page tables determine what memory addresses mean, and what parts of physical memory can be accessed. They allow xv6 to isolate different process's address spaces and to multiplex them onto a single physical memory.

The task of part 2 is to implement a pair of system calls: GetSharedPage() and FreeSharedPage() that will allow two programs (two processes) to share pages. Here I explain my idea to implement these two system calls in detail.

2. Paging hardware

As Figure 3.2 shows, a RISC-V CPU translates a virtual address into a physical in three steps. A page table is stored in physical memory as a three-level tree. The root of the tree is a 4096-byte page-table page that contains 512 PTEs, which contain the physical addresses for page-table pages in the next level of the tree. Each of those pages contains 512 PTEs for the final level in the tree. The paging hardware uses the top 9 bits of the 27 bits to select a PTE in the root page-table page, the middle 9 bits to select a PTE in a page table page in the next level of the tree, and the bottom 9 bits to select the final PTE.

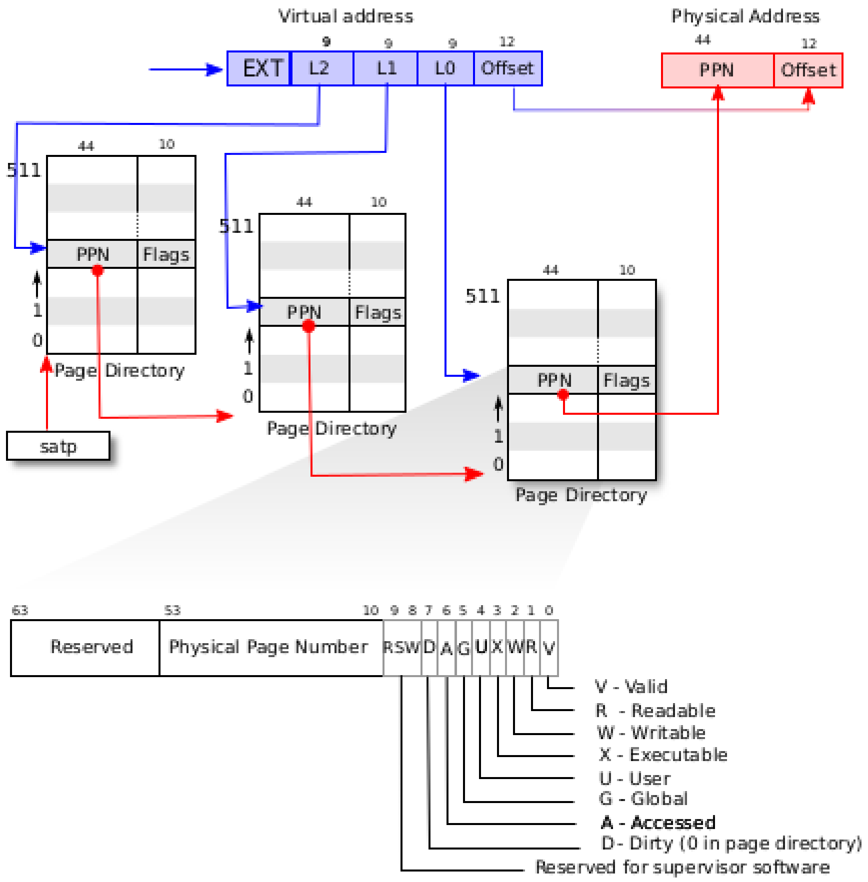
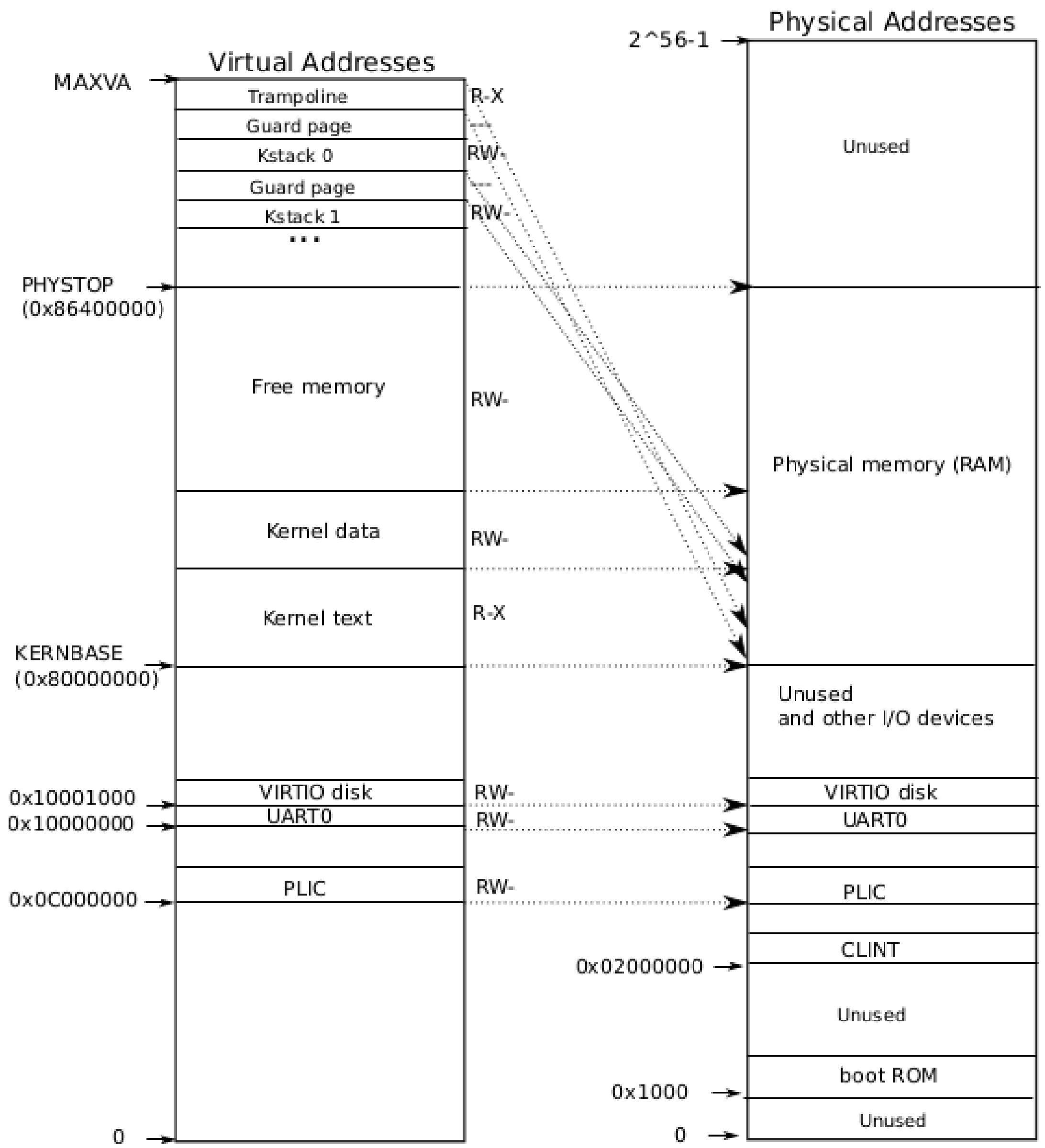


Figure 3.2: RISC-V address translation details.

3. Memory layout

The following picture shows how this layout maps kernel virtual addresses to physical addresses.



From the memory layout, we could see clearly that kernel is directed map to physical memory, which is useful for the kernel control the whole system.

4. Code for implementation

My design for the functions of `getsharedpage` and `freesharedpage` are based on the following ideas.

4.1 Record the share page information for every process

Since one process may call the `getsharedpage` several times, I used a `struct shared_page_mapping` to record the key and its corresponding virtual address. For each process, I assumed that the maximum call times is 32.

```c
// record mapped shared pages
struct shared_page_mapping {
  int key;
  void *va;
};

// Per-process state
struct proc {
    ...
  struct shared_page_mapping shm[32]; // process shared pages
};
```

4.2 Create keys and shared pages mappings.

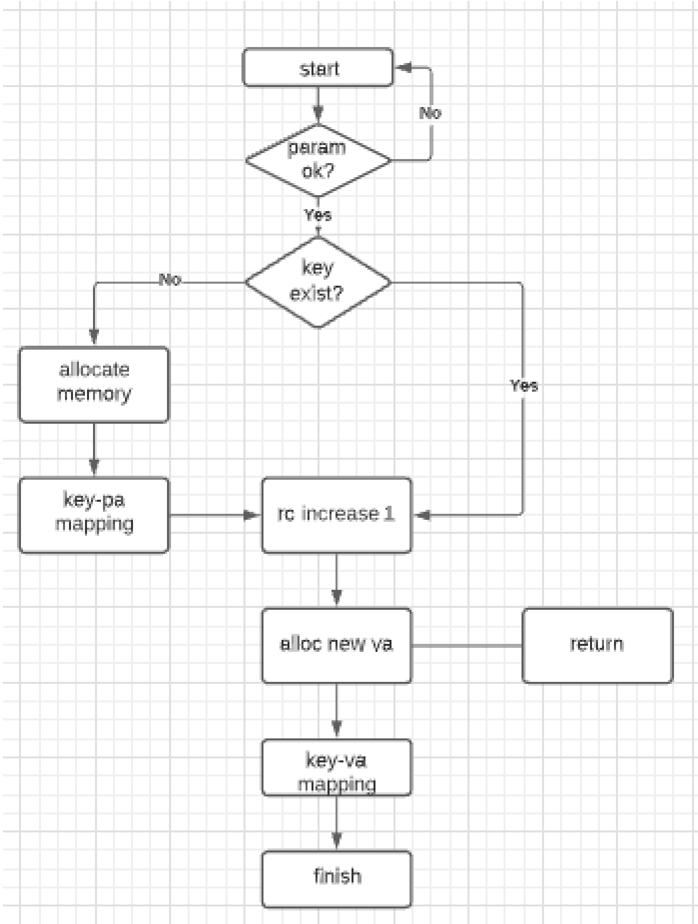
Although we record key for every process, the key is a global variable. Every process could use it. So in the virtual memory, I use an `struct shm_region` to record the mapping between keys and shared pages. Considering the requirement of freeing shared page, I add an attribute `rc` to record how many processes reference to the shared pages. An array `physical_pages` uses to record the map between the key and the physical address.

```c
struct shm_region {
  bool valid;
  int rc;
  int len;
  uint64 physical_pages[MAX_REGION_SIZE];
};
```

Every time, when we call the `getsharedpage`, we system will return an virtual address which points to a shared page region. Creating virtual addresses and physical addresses mappings, the XV6 system calls `mappages` functions, which installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. The bits of PTE tells the system several information about read, write and valid.

```c
// Map region <key> into process <p> at virtual addres <addr>
void map_shm_region(int key, struct proc *p, void *addr) {
  for (int k = 0; k < regions[key].len; k++) {
    mappages(p->pagetable, (uint64)(addr + (k*PGSIZE)), PGSIZE,
regions[key].physical_pages[k], PTE_V|PTE_W|PTE_U|PTE_R);
  }
}
```

4.3 Implementation of the `getsharedpage`




```
void *
getsharedpage(int key, int len)
{
  uint64 mem;
  // check parameter
  if(key < 0 || key > 32)
    return (void *)0;
  struct proc *p = myproc();
  // Allocate pages in the appropriate regions' pysical pages
  if(!regions[key].valid) {
    for(int j = 0; j < len; j++) {
      if((mem = (uint64)kalloc()) == 0)
        return (void *)-1;
      memset((void *)mem, 0, PGSIZE); // fill in zero in this page
      regions[key].physical_pages[j] = mem; // Save new page
      //printf("save memory success.\n");
    }
    regions[key].valid = 1;
    regions[key].len = len;
    regions[key].rc = 0;
  }
  regions[key].rc += 1;

  // Find the index in the process
  int shind = -1;
  for (int x = 0; x < 32; x++) {
    if (p->shm[x].key == -1) {
      shind = x;
      break;
    }
  }
```

```
  if (shind == -1)
    return (void*)0;

  //printf("shind= %d\n", shind);

  // Get the lowest virtual address space currently allocated
  void *va = (void*)KERNBASE-PGSIZE;
  for (int x = 0; x < 32; x++) {
    if (p->shm[x].key != -1 && (uint64)(va) > (uint64)(p->shm[x].va)) {
      va = p->shm[x].va;
    }
  }

  // Get va of new mapped pages
  va = (void*)va - (len*PGSIZE);
  p->shm[shind].va = va;
  p->shm[shind].key = key;


  p->shm[shind].va = (void*)va;
  p->shm[shind].key = key;

  // Map them in memory
  map_shm_region(key, p, (void*)va);

  //printf("map success.\n");

  return (void*) va;
}
```

5. Code for implementation `freesharedpage`

There are three operations we need to when it comes to `freesharedpage`. The first one is to clear shared memory data structure, which makes the shared page mapping to go back to initialize state; the last one is to decrease the reference count, freeing if unused.

The second one is the most important operation in implementation. It is to clear page table entries for all pages in the process. We could use `walk` to find the PTE. `walk` descends the 3-level page table 9 bits at the time. It uses each level 9 bits of virtual address to find the PTE of either the next-level page table or the final page. If the PTE isn't valid, then the required page hasn't yet been allocated; if the `alloc` argument is set, walk allocates a new page-table page and puts its physical address in the PTE. It returns the address of the PTE in the lowest layer in the tree.

```
// dealing with freeing of shared pages.
int
freesharedpage(int key)
{
  // Clear shared memory data structure
  struct proc *p = myproc();
  void *va = 0;
  for (int i = 0; i < 32; i++) {
    if (p->shm[i].key == key) {
      va = p->shm[i].va;
      p->shm[i].key = -1;
      p->shm[i].va = 0;
      break;
```

```c
    }
  }
  if(va == 0)
    return -1;

  // Clear page table entries for all pages in the process
  struct shm_region* reg = &regions[key];
  for(int i = 0; i < reg->len; i++) {
    pte_t* pte = walk(p->pagetable, (uint64)va + i*PGSIZE, 0);
    if(pte == 0) {
      return -1;
    }
    *pte = 0;
  }

  // Decrease the refcount, freeing if unused.
  reg->rc--;
  if(reg->rc == 0) {
    regions[key].valid = 0;
    regions[key].rc = 0;
    for(int i = 0; i < regions[key].len; i++)
      kfree((void*)(regions[key].physical_pages[i]));
    regions[key].len = 0;
  }

  return 0;
}
```