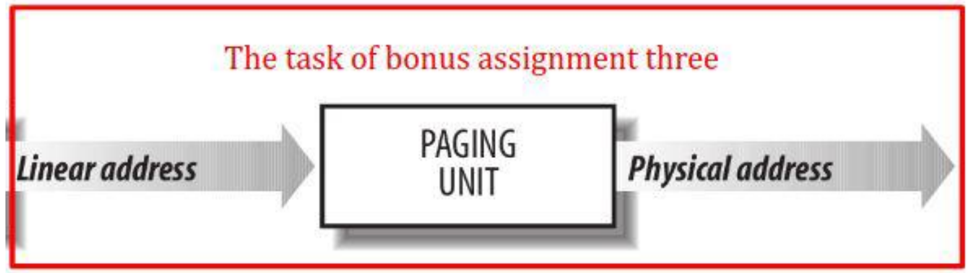
Bonus Assignment Four

XUE JIN

jinxue@cse.cuhk.edu.hk

Overview



Task:

- Implement a simple simulator of translating processes’ linear address (virtual address) to physical address using paging technique.

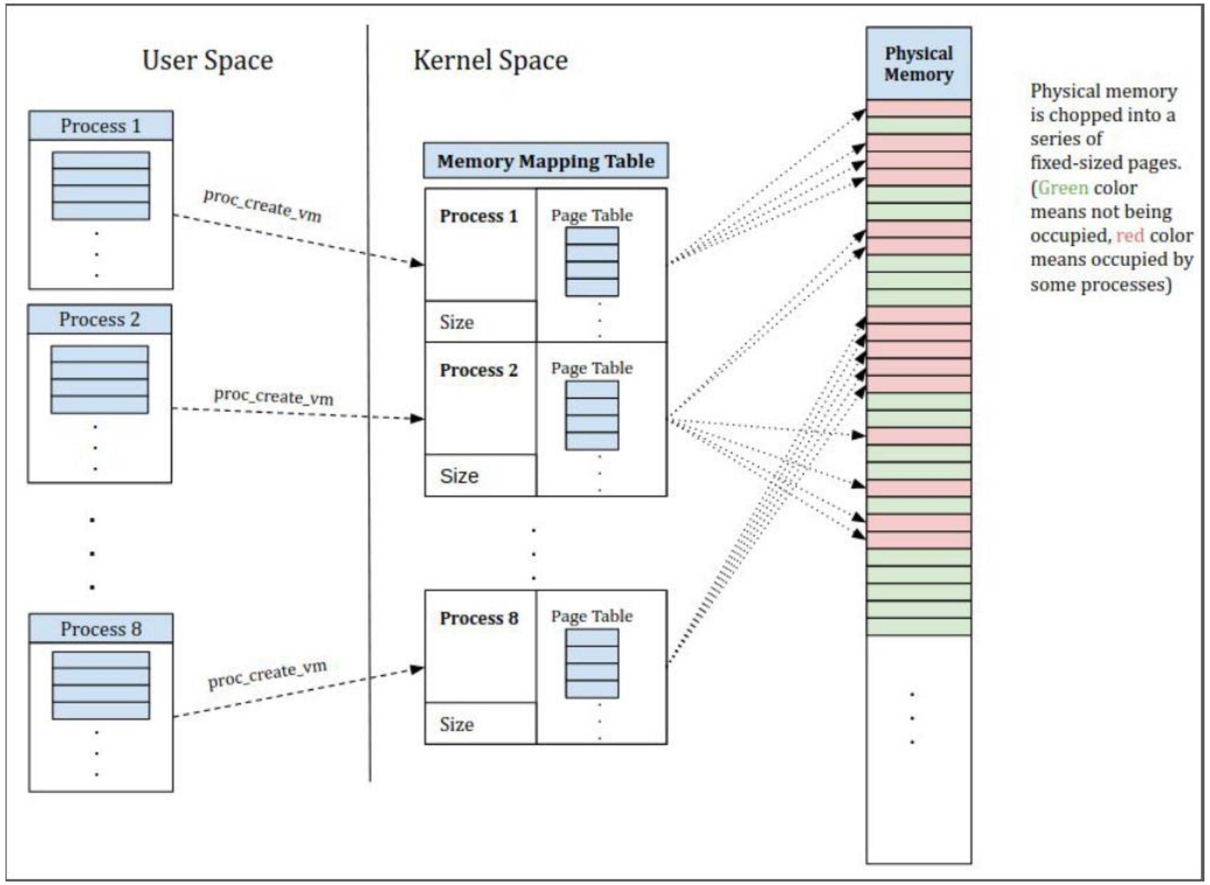
Overview

User Space

- User can create (proc_create_vm) process with specified virtual memory size through the API.
- User can read/write (vm_read/vm_write) the process' virtual memory through API.
- User can terminate (proc_exit_vm) process through the API.

Kernel Space

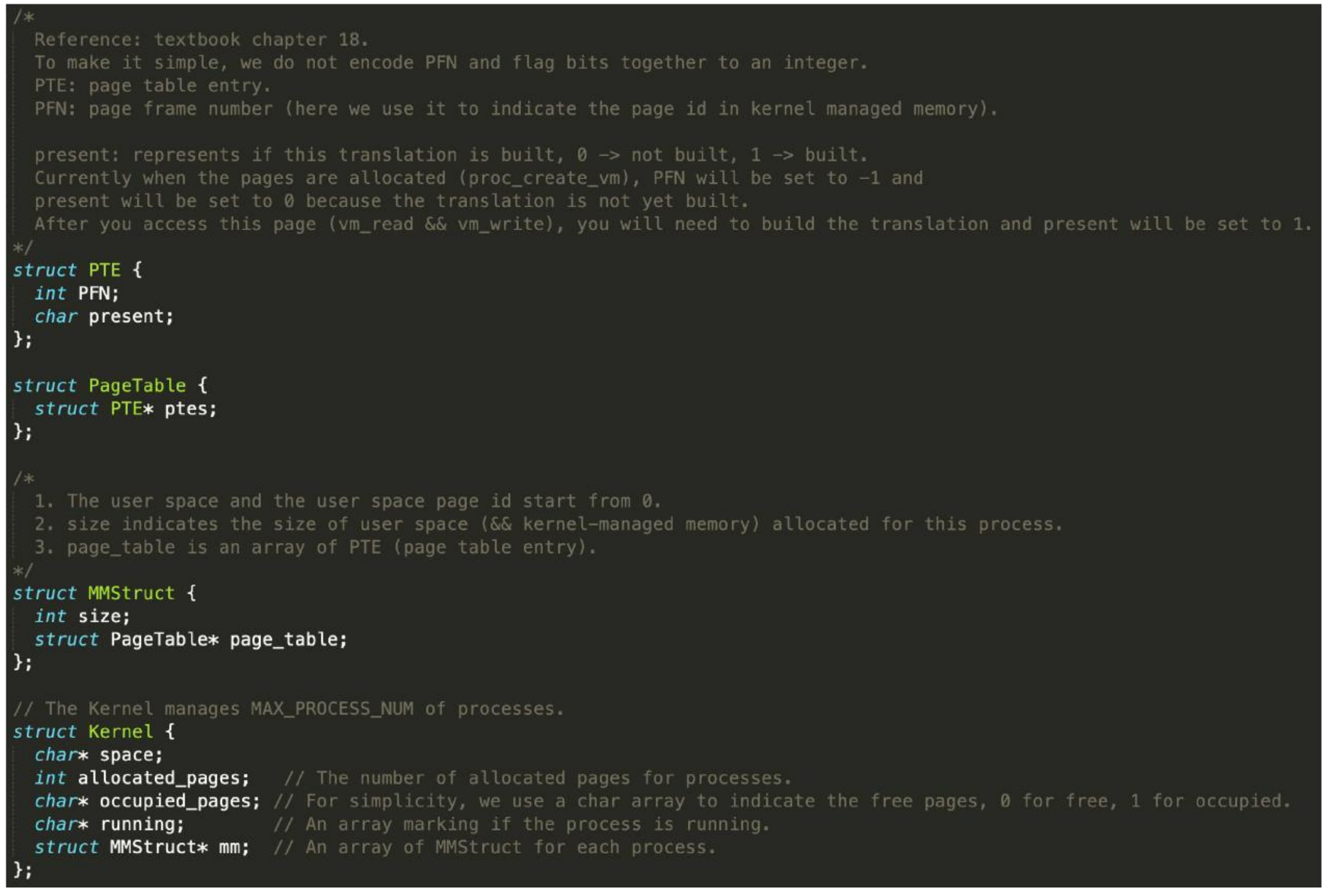
- The kernel simulator maintains a page table for each process and a shared physical memory.
- The kernel simulator handles the mappings from processes' virtual memory to physical memory.



Data Structures

Kernel

- **space**: the physical memory.
- **allocated_pages**: Total number of allocated pages for all running processes.
- **occupied_pages**: char array (size is the number of kernel pages) to indicate the free pages, 0 -> free, 1 -> occupied.
- **running**: an array marking if the corresponding process is running.
- **mm**: an array of page tables.
  - Array of **MMStruct**

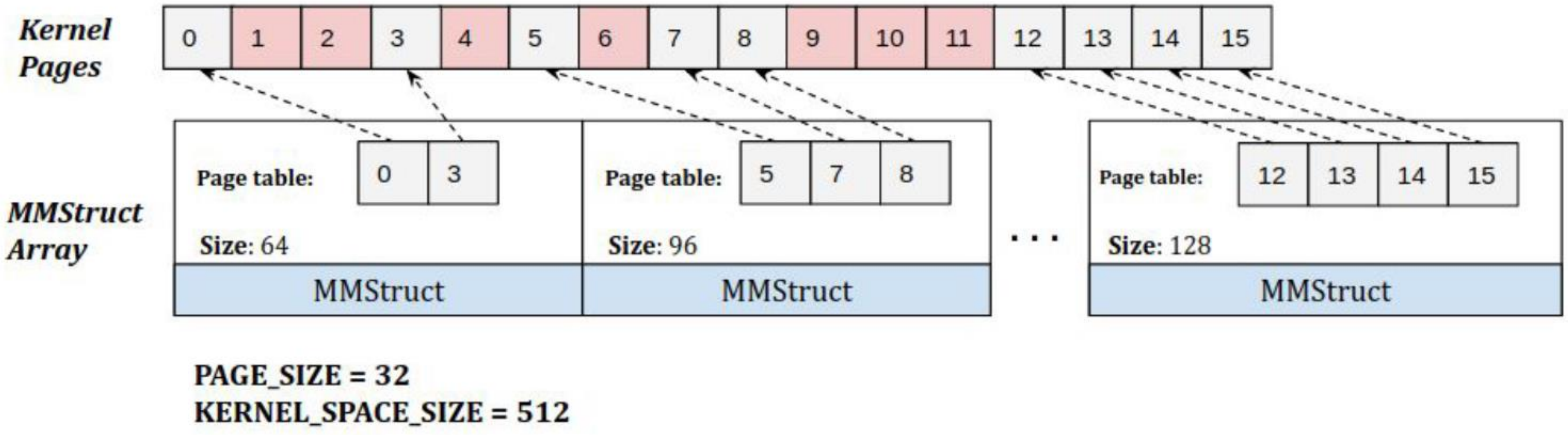


```c
/*
  Reference: textbook chapter 18.
  To make it simple, we do not encode PFN and flag bits together to an integer.
  PTE: page table entry.
  PFN: page frame number (here we use it to indicate the page id in kernel managed memory).

  present: represents if this translation is built, 0 -> not built, 1 -> built.
  Currently when the pages are allocated (proc_create_vm), PFN will be set to -1 and
  present will be set to 0 because the translation is not yet built.
  After you access this page (vm_read && vm_write), you will need to build the translation and present will be set to 1.
*/
struct PTE {
  int PFN;
  char present;
};

struct PageTable {
  struct PTE* ptes;
};

/*
  1. The user space and the user space page id start from 0.
  2. size indicates the size of user space (&& kernel-managed memory) allocated for this process.
  3. page_table is an array of PTE (page table entry).
*/
struct MMStruct {
  int size;
  struct PageTable* page_table;
};

// The Kernel manages MAX_PROCESS_NUM of processes.
struct Kernel {
  char* space;
  int allocated_pages;   // The number of allocated pages for processes.
  char* occupied_pages; // For simplicity, we use a char array to indicate the free pages, 0 for free, 1 for occupied.
  char* running;        // An array marking if the process is running.
  struct MMStruct* mm;  // An array of MMStruct for each process.
};
```

Data Structures (Cont'd)

MMStruct:

- ***size*** determines the number of allocated pages.
- ***page table*** contains an array of PTEs.



Implementation

Task: Totally 4 functions (API) in the kernel.c need to be implemented.

1. int proc_create_vm(struct Kernel* kernel, int size)

1.1. Check if a free process slot exists (check the running, the slot will be the pid returned).

1.2. Check if there's enough free space (check allocated_pages).

1.3. Allocate the space for page_table (the size of it depends on the pages you needed. e.g. if size=33 and PAGE_SIZE=32, then you need 2 pages) and update allocated_pages.

1.4. The mapping to kernel-managed memory is not built up, all the PFN should be set to -1 and present byte to 0 (PTE) and set the corresponding element in running to 1.

1.5. Return the pid if success, -1 if failure.

Implementation (Cont’d)

2. int vm_read(struct Kernel* kernel, int pid, char* addr, int size, char* buf)

2.1. Check if the reading range is out-of-bounds.

2.2. If the pages in the range [addr, addr+size) of the user space of that process are not present, you should firstly map them to the free kernel-managed memory pages (first fit policy: scan from the beginning).

3. int vm_write(struct Kernel* kernel, int pid, char* addr, int size, char* buf)

4. int proc_exit_vm(struct Kernel* kernel, int pid)

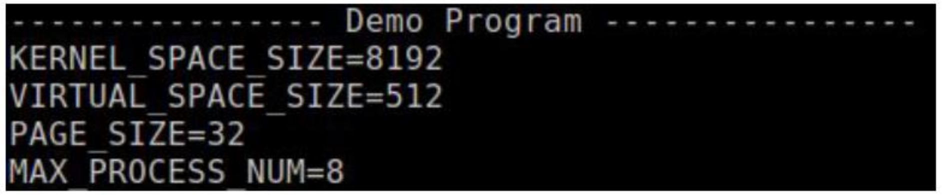
4.1. Reset the corresponding pages in occupied_pages to 0.

4.2. Release the page_table in the corresponding MMStruct and set to NULL. Return 0 when success, -1 when failure.

Demo

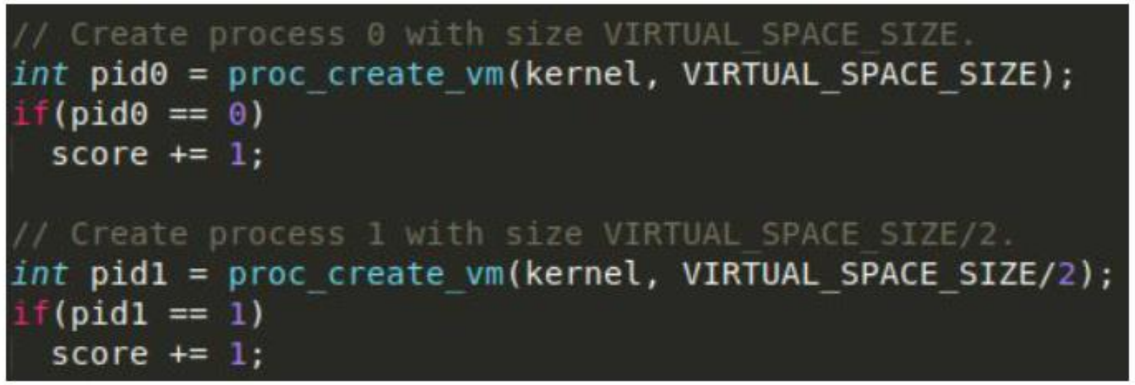
You can find the demo program in *demo.c*

Initial parameters

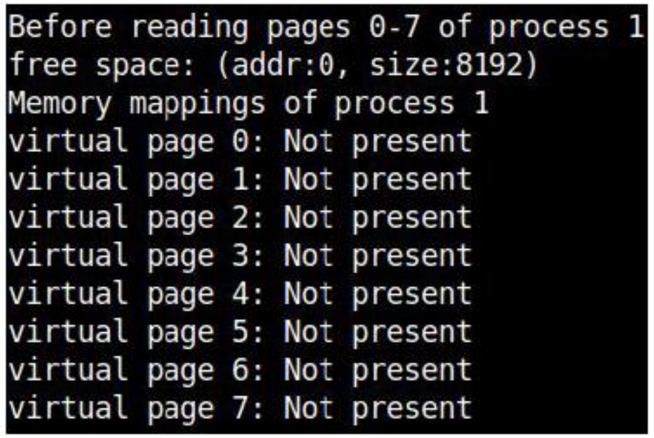


```
---------------- Demo Program ----------------
KERNEL_SPACE_SIZE=8192
VIRTUAL_SPACE_SIZE=512
PAGE_SIZE=32
MAX_PROCESS_NUM=8
```

Create process 0 and 1



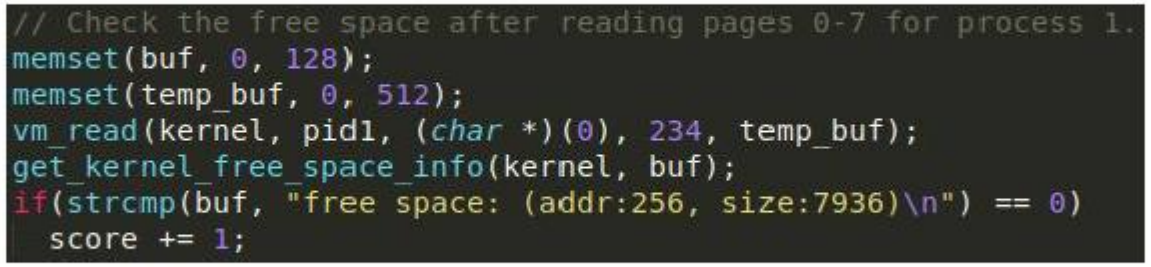
```
// Create process 0 with size VIRTUAL_SPACE_SIZE.
int pid0 = proc_create_vm(kernel, VIRTUAL_SPACE_SIZE);
if(pid0 == 0)
  score += 1;

// Create process 1 with size VIRTUAL_SPACE_SIZE/2.
int pid1 = proc_create_vm(kernel, VIRTUAL_SPACE_SIZE/2);
if(pid1 == 1)
  score += 1;
```



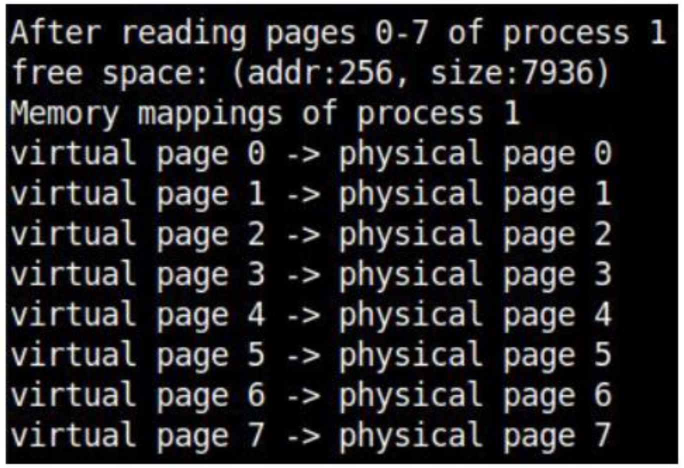
```
Before reading pages 0-7 of process 1
free space: (addr:0, size:8192)
Memory mappings of process 1
virtual page 0: Not present
virtual page 1: Not present
virtual page 2: Not present
virtual page 3: Not present
virtual page 4: Not present
virtual page 5: Not present
virtual page 6: Not present
virtual page 7: Not present
```

Demo (Cont’d)

Read page 0-7 of process 1

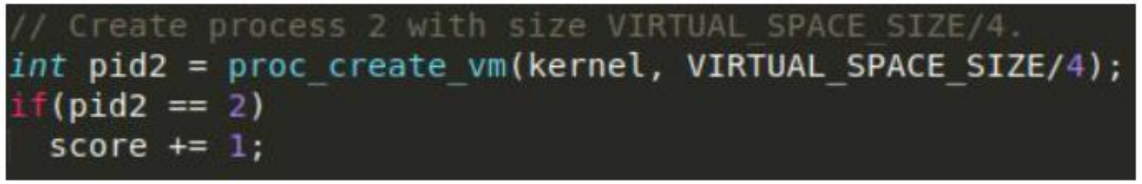


```
// Check the free space after reading pages 0-7 for process 1.
memset(buf, 0, 128);
memset(temp_buf, 0, 512);
vm_read(kernel, pid1, (char *)(0), 234, temp_buf);
get_kernel_free_space_info(kernel, buf);
if(strcmp(buf, "free space: (addr:256, size:7936)\n") == 0)
  score += 1;
```

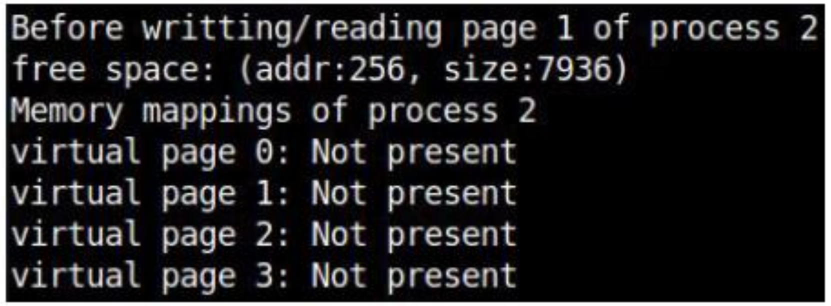


```
After reading pages 0-7 of process 1
free space: (addr:256, size:7936)
Memory mappings of process 1
virtual page 0 -> physical page 0
virtual page 1 -> physical page 1
virtual page 2 -> physical page 2
virtual page 3 -> physical page 3
virtual page 4 -> physical page 4
virtual page 5 -> physical page 5
virtual page 6 -> physical page 6
virtual page 7 -> physical page 7
```

Create process 2



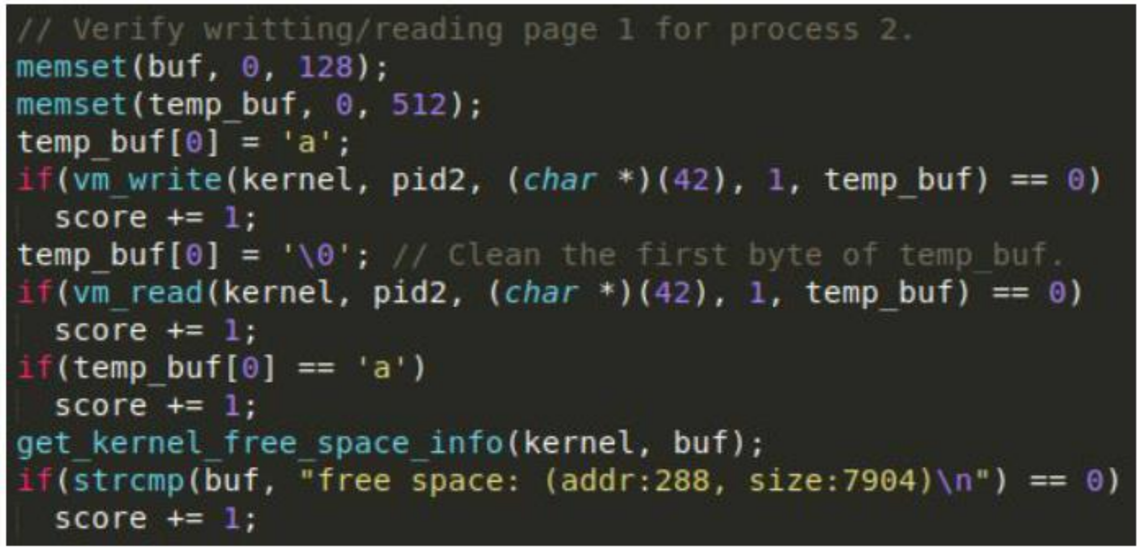
```
// Create process 2 with size VIRTUAL_SPACE_SIZE/4.
int pid2 = proc_create_vm(kernel, VIRTUAL_SPACE_SIZE/4);
if(pid2 == 2)
  score += 1;
```



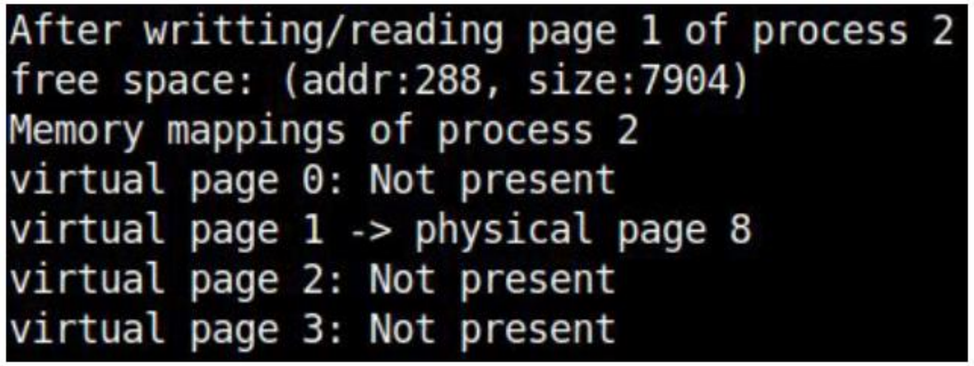
```
Before writting/reading page 1 of process 2
free space: (addr:256, size:7936)
Memory mappings of process 2
virtual page 0: Not present
virtual page 1: Not present
virtual page 2: Not present
virtual page 3: Not present
```

Demo (Cont’d)

Write and read the page 1 of process 2



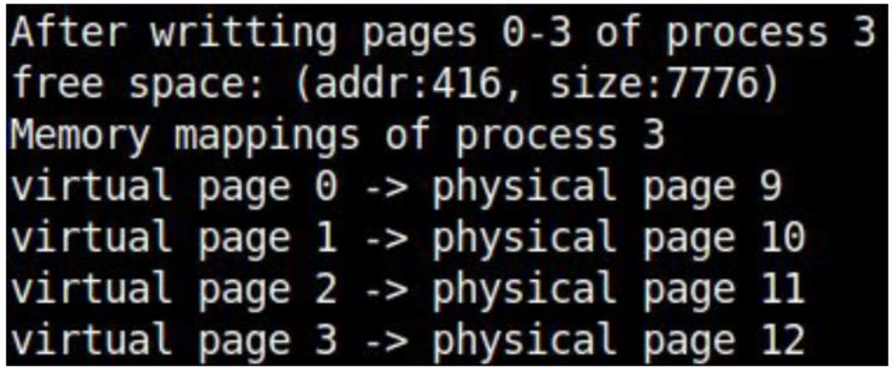
```
// Verify writting/reading page 1 for process 2.
memset(buf, 0, 128);
memset(temp_buf, 0, 512);
temp_buf[0] = 'a';
if(vm_write(kernel, pid2, (char *)(42), 1, temp_buf) == 0)
  score += 1;
temp_buf[0] = '\0'; // Clean the first byte of temp_buf.
if(vm_read(kernel, pid2, (char *)(42), 1, temp_buf) == 0)
  score += 1;
if(temp_buf[0] == 'a')
  score += 1;
get_kernel_free_space_info(kernel, buf);
if(strcmp(buf, "free space: (addr:288, size:7904)\n") == 0)
  score += 1;
```



```
After writting/reading page 1 of process 2
free space: (addr:288, size:7904)
Memory mappings of process 2
virtual page 0: Not present
virtual page 1 -> physical page 8
virtual page 2: Not present
virtual page 3: Not present
```

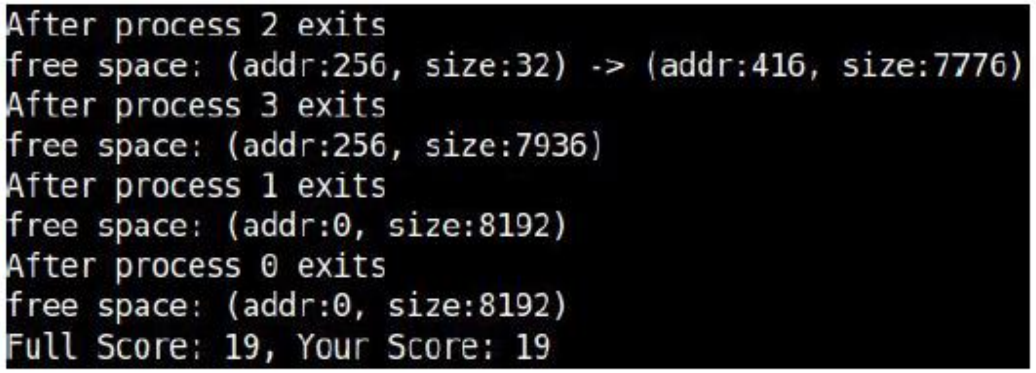
Demo (Cont'd)

Create process 3 and write page 0-3 of process 3



```
After writting pages 0-3 of process 3
free space: (addr:416, size:7776)
Memory mappings of process 3
virtual page 0 -> physical page 9
virtual page 1 -> physical page 10
virtual page 2 -> physical page 11
virtual page 3 -> physical page 12
```

Exit



```
After process 2 exits
free space: (addr:256, size:32) -> (addr:416, size:7776)
After process 3 exits
free space: (addr:256, size:7936)
After process 1 exits
free space: (addr:0, size:8192)
After process 0 exits
free space: (addr:0, size:8192)
Full Score: 19, Your Score: 19
```

Submission (Deadline: 23:59, May 8, 2022)

After you finish bonus assignment four, please only submit the kernel.c to Blackboard.