CSE 120 - Discussion 6

May 13, 2019

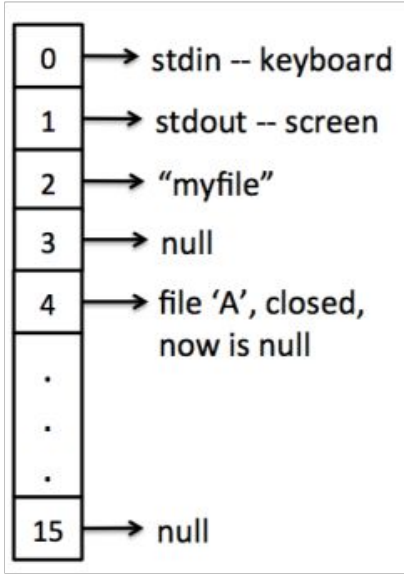
Overview of Project 2

1. Implement file handling system calls
    - creat, open, read, write, close, unlink
2. Support multiprogramming at the user level
    - Memory management for multiple processes
3. Implement process control system calls
    - exec, join, exit

Implementing file handling system calls - Recap

File Descriptors

- File Descriptor - used by program to refer to a file
- File Descriptor Table :
  Mapping between fd and OpenFile for each process
  - creat() and open() add an entry to the table, return the fd to the user program
  - read() and write() use this fd
  - close() deletes an entry from the table



Implementation tips

- Data structure for the File Table
- Use methods from machine.stubFileSystem.java
  - creat() and open() - check methods in ThreadedKernel.fileSystem
  - unlink() - check methods in ThreadedKernel.fileSystem
  - close() - file.close()
  - read() - read from file to buffer, place contents read onto user specified buffer in virtual memory. Use file.read()
  - write() - read from user specified buffer to local buffer, write contents of local buffer to file. Use file.write()

Support multiprogramming at the user level

Physical Memory

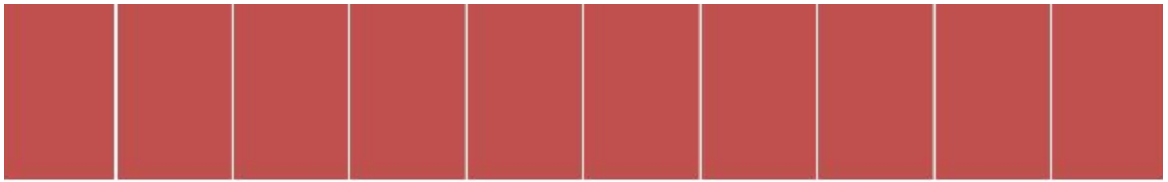
- Memory is split into pages of fixed size
- User processes do not see physical memory at all
- They only see a virtual address space

10 pages of physical memory:

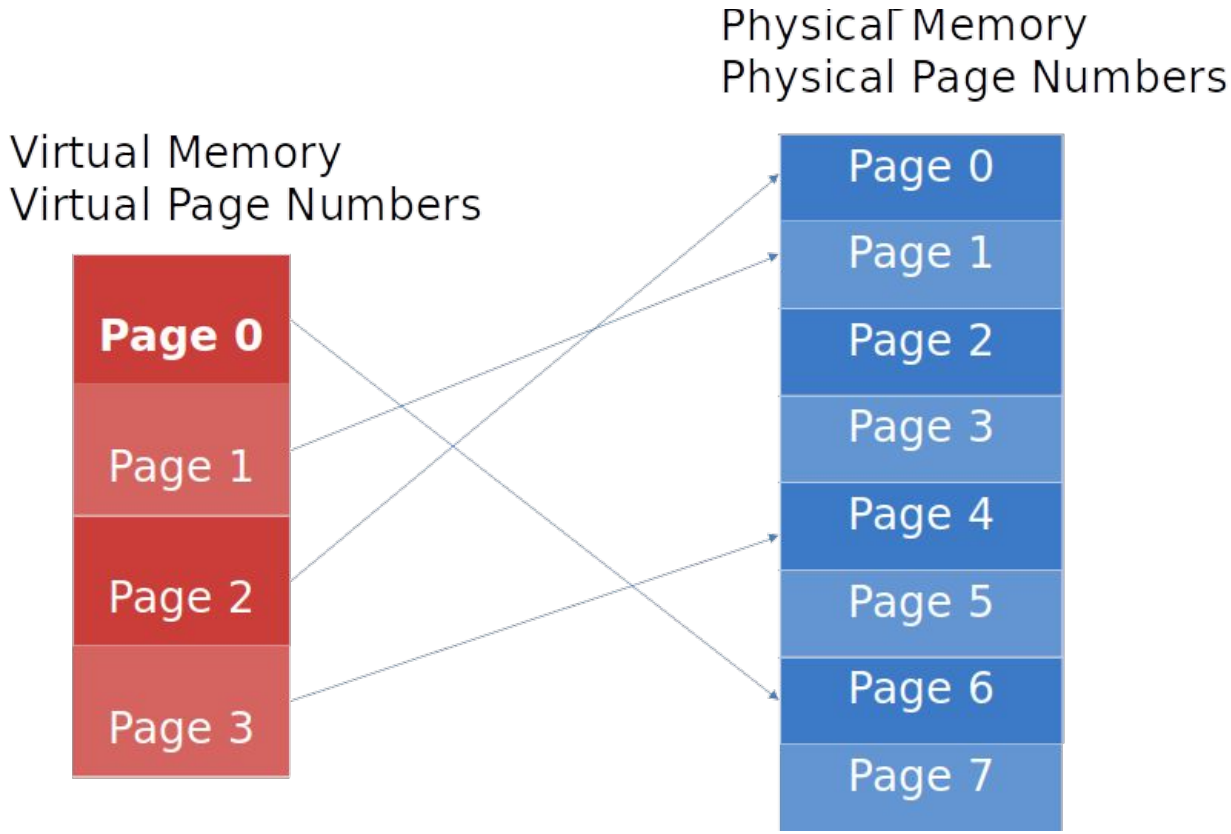
Virtual Memory

- Gives each process the illusion of a private, contiguous address space
- Virtual memory is also split into pages of fixed size

10 pages of virtual memory:



Virtual to physical mapping



Address Translation

- Virtual address = virtual page number(vpn) + offset
- Get corresponding physical page number from the Page Table
- Physical address = physical page number(ppn) + offset

Page Table:

VPN	PPN
0	6
1	1
2	0
3	4

Current Implementation

- Several processes can run on a uniprocessor
- Each process should have a separate address space

- Currently, all physical pages are given to the first user process and vpn = ppn

```
public UserProcess() {
 int numPhysPages = Machine.processor().getNumPhysPages();
 pageTable = new TranslationEntry[numPhysPages];
 for (int i = 0; i < numPhysPages; i++)
       pageTable[i] = new TranslationEntry(i, i, true,
                                           false, false, false);
}
```

What do we need to do?

- Memory allocation
  - Allocate physical pages to multiple processes
  - Keep track of free physical pages
- Implement a simple page table
  - Translate virtual page number to physical page number
- Modify readVirtualMemory() and writeVirtualMemory() to support multiple processes

Memory allocation

- How many physical pages are there in total?
  - Machine.processor().getNumPhysPages()
- How many pages are allocated to a process?
  - UserProcess.java : numPages
  - load() : numPages includes
    - Coff section pages
    - 8 stack pages
    - 1 page reserved for arguments

Memory allocation

- How do we track free/available physical pages?
- What data structure can we use to maintain free pages?
- Where do we declare this data structure?
  - UserProcess or UserKernel? Why?
- Do we need mutual exclusion on this data structure?

Page Table Implementation

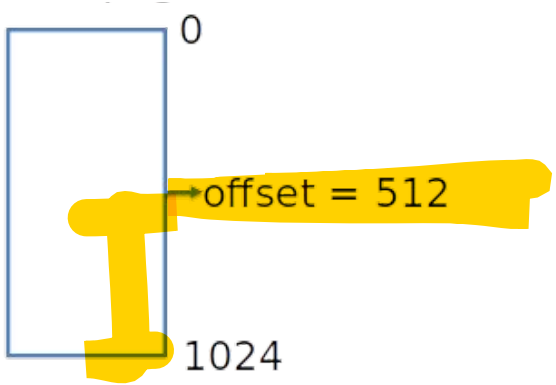
- Each page table entry (TranslationEntry) maps a vpn to a ppn
- How many entries in one process's page table?
  - numPages
- Understand other entries of the TranslationEntries fields
- When to allocate physical pages and fill up the page table?
  - In loadSections()
  - Reclaim pages in unloadSections() accordingly
- When to set bits to read only?
  - Determine which sections are read only using Coff.isReadOnly()

Translation : vaddr to paddr

- Get vpn from vaddr
  - Processor.pageFromAddress(vaddr)
- Get page offset from vaddr
  - Processor.offsetFromAddress(vaddr)
- Get ppn from the page table entry at vpn
- Compute physical address
  - pageSize x ppn + pageOffset

Page boundary

- What is the maximum size you can read each time?
  - Once we obtain the paddr, can we still read 1024 bytes from this physical page?



Process Management System Calls

The exec() system call

exec(char *file, int argc, char *argv[])

- Create new child process for *file
- argc - number of arguments
- Char *argv[] - array of pointers to the arguments
- Return processID of child
- processID : globally unique positive integer

Implementing exec()

handleExec(coffName, argc, argv)

- Read coffName from virtual address
- Read argvs from virtual address
- Execute user program in a new child process - newUserProcess()
- Save details of child in parent and details of parent in child
- Return child PID

The join() system call

join(int pid, int *status)

- pid - child’s pid
- Suspend execution until child completes
  - Condition variable?
- Get child status and write it to *status
- Return status of join

Implementing join()

handleJoin(childID, *status)

- Sleep on child
- Get and set child status
- Status is virtual address

The exit system call

exit(int status)

- Terminate the current process
- Close all file descriptors
- Free all memory
- Save the status to parent
  - jon() will need it
- Awake the parent process
  - join() would have put it to sleep

Implementing exit()

handleExit(status)

- Close all files in file table
- Delete all memory by calling UnloadSections()
- Close the coff by calling coff.close()
- If it has a parent process, save the status for parent
- Wake up parent if sleeping
- In case of last process, call kernel.kernel.terminate()
- Close KThread by calling KThread.finish()