Project 3: Virtual Memory

Prof. Jin-Soo Kim (jinsookim@skku.edu)
T.A. – Jong-Sung Lee (leitia@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Introduction (1)

- Paging in the x86 architecture

*32 bits aligned onto a 4-KByte boundary.
Current Pintos VM implementation

- Use paging
- Page size: 4KB
- Each process has its own page tables
  - The page directory is allocated when the process is created (pagedir_create() @ userprog/pagedir.c)
  - (struct thread *) t->pagedir points to the page directory (load() @ userprog/process.c)
  - The (secondary) page tables are dynamically created if necessary (lookup_page() @ userprog/pagedir.c)
  - For kernel region, processes have the same mapping (PHYS_BASE ~ 0xffffffff)
Introduction (3)

- Current Pintos VM implementation (cont’d)
  
  - No demand paging
    - When a process is created, all the contents of code and data segments are read into the physical memory (load_segment() @ userprog/process.c)
  
  - Fixed stack size
    - Only one stack page is allocated to each process (setup_stack() @ userprog/process.c)
Project 3 Overview

- **Requirements**
  - Lazy loading (or demand paging)
  - Swapping in/out pages from/to swap disk
  - Dynamic stack growth
  - Memory mapped files
Lazy Loading (1)

**Why?**
- An executable file holds code and data images
- A process will not need all the pages immediately

**How to?**
- Use the executable file as the backing store
  - Only when a page is needed at run time, load the corresponding code/data page into the physical memory
  - Loaded pages will have valid PTEs
- Handling page faults
  - Accesses to not-yet-loaded pages will cause page faults
  - Find the corresponding location in the executable file
  - Read in the page from the executable file
  - Setup the corresponding PTE
Lazy Loading (2)

- Loading code/data from the executable file
  - In load_segment() @ userprog/process.c
  - Each page is filled with data using “page_zero_bytes” and “page_read_bytes”
    - \( \text{page_zero_bytes} + \text{page_read_bytes} = \text{PGSIZE} \)
  - All zeroed page (\( \text{page_zero_bytes} == \text{PGSIZE} \))
    - Allocate a new page and initialize it with zeroes
  - Full code/data page (\( \text{page_read_bytes} == \text{PGSIZE} \))
    - Allocate a new page and read its contents from the executable file
  - Partial page (\( 0 < \text{page_read_bytes} < \text{PGSIZE} \))
    - Read \( \text{page_read_bytes} \) from the executable file and fill the rest of the page with zeroes
Lazy Loading (3)

- The supplemental page table
  - The page table with additional data about each page
  - Main purposes
    - On a page fault, find out what data should be there for the faulted virtual page
    - On a process termination, decide what resources to free
  - Possible organizations
    - Per-segment
    - Per-page
  - Implementation
    - You can use any data structure for the supplemental page table.
    - `<hash.h>` will be useful (`lib/kernel/hash.[ch]`)
Lazy Loading (4)

- Strategy

1. Get the information on the faulted page
2. Allocate a new frame
3. Read from the executable file
4. Update information
5. Install the new page

Page fault handler

The supplemental page table

VA

x86 page table

Page fault

1. Get the information on the faulted page
2. Allocate a new frame
3. Read from the executable file
4. Update information
5. Install the new page

The supplemental page table
Swapping (1)

- **Why?**
  - You may run out of the physical memory
  - Your program’s memory footprint can be larger than the physical memory size

- **How to?**
  - Find a victim page in the physical memory
  - Swap out the victim page to the swap disk
  - Extend your supplemental page table to indicate the victim page has been swapped out
  - When the page is accessed later, swap in the page from the swap disk to the physical memory
**Swapping (2)**

### Swap disk

- Use the following command to create an 4 MB swap disk in the `vm/build` directory
  
  \[
  $ \text{pintos-mkdisk swap.dsk 4}
  \]

- Alternatively, you can tell Pintos to use a temporary 4-MB swap disk for a single run with `--swap-size=4`
  - Used during “make check”

- A swap disk consists of swap slots
  - A swap slot is a continuous, page-size region of disk space on the swap disk
Swapping (3)

- Accessing swap disk
  - Use the disk interface in devices/block.h
    - A size of a disk sector is 512 bytes
    - You can read or write one sector at a time

```c
struct block *block_get_role (enum block_type);
block_sector_t block_size (struct block *);
void block_read (struct block *, block_sector_t, void *);
void block_write (struct block *, block_sector_t, const void *);
```
Swapping (4)

- Managing swap slots
  - Pick an unused swap slot for evicting a page from its from to the swap disk
  - Free a swap slot when its page is read back or the process is terminated
  - Allocate lazily, i.e., only when they are actually required by eviction

- The swap table
  - The swap table tracks in-use and free swap slots
  - `<bitmap.h>` will be useful (`lib/kernel/bitmap.[ch]`)
Swapping (5)

- Page replacement policy

  - You should implement a global page replacement algorithm that approximates LRU
    - Do not use FIFO or RANDOM
    - The “second chance” or “clock” algorithm is OK
    - Bonus if you implement your own page replacement policy better than the “second chance” algorithm

  - Get/Clear Accessed and Dirty bits in the PTE
    - pagedir_is_dirty(), pagedir_set_dirty()
    - pagedir_is_accessed(), pagedir_set_accessed()

  - Other processes should be able to run while you are performing I/O due to page faults
    - Some synchronization effort will be required
Swapping (6)

- **The frame table**
  - Allows efficient implementation of eviction policy
  - One entry for each frame that contains a user page
    - Each entry contains a pointer to the page, if any, that currently occupies it, and other data of your choice
  - Use the frame table while you choose a victim page to evict when no frames are free
  - Code pages can be shared among those processes created from the same executable file (optional)
Swapping (7)

- **User pool vs. kernel pool**
  - The physical memory is divided into the user pool and the kernel pool
    - Running out of pages in the user pool just causes user programs to page
    - Running out of pages in the kernel pool means a disaster
    - The size of the user pool can be limited (\(-u1\) option)
  - The frames used for user pages should be obtained from the “user pool”
    - By calling `palloc_get_page (PAL_USER)`
**Swapping (8)**

- **Frame allocation**
  - On top of the current page allocator (threads/palloc.c)
    - `palloc_get_page()`, `palloc_free_page()`
  - If there are free frames in the user pool, allocate one by calling `palloc_get_page()`
  - If none is free
    - Choose a victim page using your page replacement policy
    - Remove references to the frame from any page table that refers to it
    - If the frame is modified, write the page to the file system or to the swap disk
    - Return the frame
Stack Growth (1)

Growing the stack segment

- Allocate additional pages as necessary
- Devise a heuristic that attempts to distinguish stack accesses from other accesses
  - Bug if a program writes to the stack below the stack pointer
  - However, in x86, it is possible to fault 4 ~ 32 bytes below the stack pointer
- You may impose some absolute limit on stack size
- The first stack page need not be allocated lazily
  - The page is initialized with the command line arguments
- All stack pages should be candidates for eviction
  - An evicted stack page should be written to swap
Stack Growth (2)

How to obtain the user stack pointer?

• You need the current value of the user program’s stack pointer on page fault
  – Compare it with the faulted address
• When the page fault occurred in the user mode
  – Use (struct intr_frame *) f->esp
• When the page fault occurred in the kernel mode
  – struct intr_frame is not saved by the processor
  – (struct intr_frame *) f->esp yields an undefined value
  – Save esp into struct thread on the initial transition from user to kernel mode
Memory Mapped Files (1)

- Example
  - Writes the contents of a file to the console

```c
#include <stdio.h>
#include <syscall.h>
int main (int argc, char *argv[])
{
    void *data = (void *) 0x10000000;

    int fd = open (argv[1]);
    mapi_d_t map = mmap (fd, data);
    write (1, data, filesize(fd));
    munmap (map);
    return 0;
}
```
Memory Mapped Files (2)

- System calls to implement

```c
mapid_t mmap (int fd, void *addr);
void munmap (mapid_t mapping);
```

- `mmap()` fails if
  - `fd` is 0, 1, 2
  - The file has a length of zero bytes
  - `addr` is 0
  - `addr` is not page-aligned
  - The range of pages mapped overlaps any existing set of mapped pages

- All mappings are implicitly unmapped when a process exits
Memory Mapped Files (3)

- Managing mapped files
  - Lazily load pages in mmap regions
    - For the final mapped page, set the bytes beyond the end of the file to zero
  - Use the mmap’d file itself as backing store for mapping
    - All pages written to by the process are written back to the file
  - Closing or removing a file does not unmap any of its mappings
    - Once created, a mapping is valid until munmap() is called or the process exits
Summary (1)

Pages

- Code page (clean)
- Data page (clean/dirty)
- Stack page (dirty)
- mmaped page (clean/dirty)
Summary (2)

- When you attach a new frame,
  - It may be just initialized to zero
  - It may be read from a file
  - It may be read from a swap slot

- When you evict a frame,
  - It may be just dropped
  - It may be swapped out to a swap slot
  - It may be written to a file
Tips (1)

- Suggested order of implementation
  - Lazy loading
    - Modify `load_segment()` and `page_fault()`
    - Construct the supplemental page table
    - You should be able to run all user programs of Project 2
  
  - Frame allocation/deallocation layer
    - Add a new interface that can allocate or free a frame
    - Construct the frame table as you allocate a new frame
    - Assume there is enough physical memory
      - No eviction is necessary
    - You should be able to run all user programs of Project 2
Tips (2)

- Suggested order of implementation (cont’d)
  - Page replacement policy
    - Develop your own page replacement policy
    - Need to interact with the supplemental page table and the frame table
    - First, try to evict read-only pages and make sure it has no problem
    - And then, implement the swap table and test your code to access the swap disk
    - Finally, implement the full-fledged page replacement policy
  - Stack growth
    - Extend your page fault handler
  - Memory mapped files
No files in the vm directory

- You should add your files in the directory
- The Pintos documentation says...

```
vm/frame.c   |  162 ++++++++  
vm/frame.h   |   23 +        
vm/page.c    | 297 ++++++++++++  
vm/page.h    |   50 ++       
vm/swap.c    |   85 +++      
vm/swap.h    |     11       
```

- Adding your own source files (src/Makefile.build)
Submission

- **Due**
  - December 8, 11:59PM
  - Fill out the design document and save it with PDF format (TeamName.pdf)
  - Tar and gzip your Pintos source codes
    
    ```
    $ cd pintos
    $(cd src/vm; make clean)
    $ tar cvzf TeamName.tar.gz src
    ```