Pintos Task 3: Virtual Memory

Eva Kalyvianaki
Mark Rutland

Imperial College London

24 February 2011

Introduction

Goal

Remove the space limitations of main memory

Tasks

i. Lazy-loading of executables: load pages on demand
ii. Stack growth
iii. Memory-mapped files
iv. Swapping

Getting Started

Does Task 2 need to be complete?

- Completion of Task 2 is essential for Task 3
- Virtual memory is an extension to user program handling
- A good understanding of threads and processes in pintos is required

Testing

- tests for Task 3
  - Including Task 2 tests
- Tests take a long time to run
  - Running individual tests will allow you to work faster

Important Files

Files to modify:

i. threads/thread.h
   - Add supplemental page table to `struct thread`

ii. userprog/process.c
   - Handles loading, execution and exit of processes
   - Handles stack setup

iii. userprog/syscall.c
   - Handles all syscalls
   - Checks validity of process-supplied data

iv. userprog/exception.c
   - Handles segmentation faults
   - Some lazy-loading code goes here

Important Files

Warning

Some features you are to add should not be placed in existing files, as they will require a large amount of additional code.

Files to create:

i. Frame table
   - vm/frame.c ?

ii. Supplemental page table
   - vm/page.c ?

iii. Swap table
   - vm/swap.c ?

iv. Memory map table
   - vm/mmap.c ?

Important Files

Files with useful functions

i. `<bitmap.h>`
   - In-kernel bitmap implementation

ii. `<hash.h>`
   - Generic hash table implementation

iii. `"devices/block.c"`
   - Provies sector-based access to block devices

iv. `"threads/vaddr.h"`
   - `is_user_vaddr()` used to validate user provided pointers
   - `pg_ofs()` used to find the offest of a pointer within a page
   - `pg_round_down()` rounds a pointer down to the nearest page boundary

v. `"userprog/pagedir.c"`
   - Functions for querying and altering the page table

Virtual Memory in Pintos

Kernel addresses

- Due to the way x86 virtual memory is implemented, it is not possible to address memory by physical address whilst in protected mode
- You will not need to use physical addresses directly, kernel functions take **kernel addresses**
- Pintos maps every physical address to a **kernel address** using a fixed offset (`PHYS_BASE`)

Page Faults in Pintos

- Handled in `page_fault` in userprog/exception.c
- Currently not very useful:
  - Kills user processes on any page fault

Project Requirements

- Frame allocator
- Lazy-loading of executables
- Dynamic stack allocation
- Memory mapped files
- Frame/swap reclamation on exit
- Swapping
- Eviction policy

Frame Allocator + Eviction Policy

What?

- After a number of allocations, physical frames will be exhausted
- Need a way of tracking allocations to enable swapping

How?

- Build a frame table
- Contains pointers to pages and other data of your choise
- Record which thread owns a frame
- Most important operation is to obtain an used frame
- Modify calls to `palloc_get_page()` and `palloc_free_page()`

Warning

Many processes may need to allocate or free pages simultaneously. Carefully consider how you will handle concurrent access.

Segment setup

Current process setup

- Executable is loaded into process memory by `load_segment()`
- Stack segment is ignored
- Stack is set up at a fixed address

So...?

- The entire executable must be loaded before it is run
- Data which may never be used is read into memory

Solution?

- Use the supplemental page table
- Load in segments on demand, using the page-fault handler

Stack growth

The problem

- Allocating a small stack (as Pintos currently does) prohibits use of large stack objects
- Allocating a large stack wastes space for processes which do not use it

The solution

- Grow the stack dynamically, as page faults occur
- Identify stack accesses by looking at the `esp` of the `struct intr_frame`

Stack accesses

- As with any other page, the stack can be swapped out.
- Set an absolute limit on stack size, 8 MB.

Memory-mapped files

Idea

- Map files into process address space
- On page fault, load file data into page
- When swapping, write file pages back to disk

Mapping files

- Need to connect mapids to actual files (a table?)
- Multiple processes can map a single file (share the page?)

Warning(s)

- Maps should remain if a file is closed or removed (use `file_reopen`)
- All mapped files should be flushed when a process exits