# Virtual Memory

## Introduction

By now you should have some familiarity with the inner workings of Pintos.You will build this assignment on top of the code we provided on FTP.

You will continue to handle Pintos disks and file systems the same way you did in the previous assignment. (Try to get lock before you call filesystem's code)

## Source Code Overview

You will work in the ‘vm’ directory for this project.You will probably be encountering just a few files for the first time:

- ‘devices/block.h/c’

  Provides sector-based read and write access to block device. You will use this interface to access the swap partition as a block device.

- 'pgae.h/c'

  Data struct of a page, sometimes called a virtual page, is a continuous region of virtual memory 4,096 bytes (the page size) in length.

a 32-bit virtual address can be divided into a 20-bit page number and a 12-bit page offset (or just offset), like this:

```
31                12 11         0
+--------------------+----------+
|   Page Number      |Offset    |
+--------------------+----------+
```

- 'frame.h/c'

  Data struct of a kernel frame, sometimes called a physical frame or a page frame, is a continuous region of physical memory.

- 'swap.h/swap.c'

Data strict of a swap slot. A swap slot is a continuous, page-size region of disk space in the swap partition.

# Managing the Frame Table

The frame table contains one entry for each frame that contains a user page. Each entry in the frame table contains a pointer to the page, if any, that currently occupies it, and other data of your choice. The frame table allows Pintos to efficiently implement an eviction policy, by choosing a page to evict when no frames are free.

The most important operation on the frame table is obtaining an unused frame. This is easy when a frame is free. When none is free, a frame must be made free by evicting some page from its frame.

If no frame can be evicted without allocating a swap slot, but swap is full, panic the kernel. Real OSes apply a wide range of policies to recover from or prevent such situations, but these policies are beyond the scope of this project.

The process of eviction comprises roughly the following steps:

1. Choose a frame to evict, using your page replacement algorithm. The “accessed” and “dirty” bits in the page table, described below, will come in handy.
2. Remove references to the frame from any page table that refers to it.
3. If necessary, write the page to the file system or to swap.

# Accessed and Dirty Bits

80x86 hardware provides some assistance for implementing page replacement algorithms, through a pair of bits in the page table entry (PTE) for each page. On any read or write to a page, the CPU sets the accessed bit to 1 in the page’s PTE, and on any write, the CPU sets the dirty bit to 1. The CPU never resets these bits to 0, but the OS may do so.

Functions to work with accessed and dirty bits:

```
bool pagedir_is_dirty (uint32_t *pd, const void *page)
bool pagedir_is_accessed (uint32_t *pd, const void *page)
```

Returns true if page directory pd contains a page table entry for page that is marked dirty (or accessed). Otherwise, returns false.

```
void pagedir_set_dirty (uint32_t *pd, const void *page, bool value)
void pagedir_set_accessed (uint32_t *pd, const void *page, bool value)
```

If page directory pd has a page table entry for page, then its dirty (or accessed) bit is set to value.

## Inspection and Updates Pages

These functions examine or update the mappings from pages to frames encapsulated by a page table. They work on both active and inactive page tables (that is, those for running and suspended processes), flushing the TLB as necessary.

```
bool pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)
```

Adds to pd a mapping from user page upage to the frame identified by kernel virtual address kpage. If writable is true, the page is mapped read/write; otherwise, it is mapped read-only.

User page `upage` must not already be mapped in `pd`.

Returns true if successful, false on failure. Failure will occur if additional memory required for the page table cannot be obtained.

```
void* pagedir_get_page (uint32_t *pd, const void *uaddr)
```

Looks up the frame mapped to uaddr in pd. Returns the kernel virtual address for that frame, if uaddr is mapped, or a null pointer if it is not.

```
void pagedir_clear_page (uint32_t *pd, void *page)
```

Marks page “not present” in pd. Later accesses to the page will fault.

**E.g** you have a kernel frame 'struct frame *f', and you can call

```
pagedir_is_accessed (f->used_process->pagedir, f->uvpage->uvaddr))
```

to determine whether a address's access bit is 0 or 1.

And call

```
pagedir_clear_page(f->used_process->pagedir, f->uvpage->uvaddr)
```

to clear a page content.

# Requirements

Implement a global page replacement algorithm that approximates LRU on top of our provided code in FTP. Your algorithm should perform at least as well as the simple variant of the “second chance” or “clock” algorithm.

The function prototype is

```
void *frame_alloc_evict(enum palloc_flags flags)
```

which is decleaded in 'src/vm/frame.c'.

You should download and **fill** the **DESIGNDOC** and archive your code(**the whole pintos folder**) and DESIGNDOC into a **zip** file **named sid*nameInPinYin*vid.zip(e.g 12330441*zhangsan*v0.zip)** and upload to the ftp.