PAGING AND SWAPPING

- In lecture: address translation in a static partitioning scheme known as “paging”
  - Main memory divided into fixed-size pages that are allocated to programs and tracked using page tables
- An entire program’s address space may not fit into main memory (may be too large, or other programs may need the space)
- Instead, some of it is stored in swap space (on disk) until needed
  - The CPU can’t access data on disk directly, so it has to be swapped into memory before use
  - If memory is full when the data is swapped in, we evict a page by swapping it out to disk.
ASSIGNMENT 4

SUMMARY
ASSIGNMENT 4

- Complete a demand paging implementation by writing the following functions:
  - lpage_fault – handles a page fault
  - lpage_evict – evicts a page from physical memory
  - page_replace – contains page replacement policy (at first random, then sequential)

- Don’t tackle these functions one at a time; add layers of functionality across all functions
Much of the VM system is already provided

Main machine-independent files are src/kern/vm/
*

Machine-dependent parts are in src/kern/arch/
mips/vm/{vm.c, coremap.c}
MIPS MMU
MIPS R2000 Memory Structure (used in OS/161)

- Page size: 4kB (4096 bytes)
- Virtual address: 32 bits (12 bits for offset)
- TLB: 64 entries; each entry is 64 bits long
  - Stores virtual address for lookup (key), and copy of the PTE
  - Software-managed (by the OS)
  - Entries are read/written as two 32-bit words: entryhi and entrylo
TLB Entries

- See src/kern/arch/mips/include/tlb.h

- High-order word
  - Virtual page number for lookup (TLBHI_VPAGE) : 20 bits (mask 0xffff000)
  - Also has 6 bits for hardware PID; unused in OS/161

- Low-order word
  - Physical page number (TLBLO_PPAGE) : 20 bits
  - Also has 4 status bits, and 8 unused bits
    - Eg: V for “valid”, D for “dirty” (“writable”/”referenced”)
**TLB Interface**

- 4 functions defined to allow the OS to access the TLB
  - TLB_Random – load an entry into a random slot in the TLB
  - TLB_Write – load an entry into a specified slot in the TLB
  - TLB_Read – read an entry from a specified slot in the TLB
  - TLB_Probe – check if the TLB has an entry for a virtual page

- Read comments in src/kern/arch/mips/include/tlb.h

- Do not use these functions directly to access TLB!
  - See coremap.c for a better alternative
TLBs and Address Translation

- Hardware checks all entries of the TLB in parallel (associative cache)
- 3 cases:
  - TLB hit: entry is found
  - TLB miss: entry is not found; causes a TLB miss exception in src/kern/arch/mips/locore/trap.c
    - Causes EX_TLBS for stores (writes)
    - Causes EX_TLBL for loads (reads)
  - Protection fault: user is trying to write to read-only memory (ie: D bit is 0); causes a TLB modify exception in src/kern/arch/mips/locore/trap.c
    - Causes EX_MOD
HANDLING TLB EXCEPTIONS

- `vm_fault` is called from `mips_trap` in `trap.c` for any TLB exception.

- **On a TLB miss:**
  - Look up page in the page table
    - Implemented in `src/kern/vm/addrspace`: `as_fault`
  - Choose an entry in the TLB to replace it
    - In `src/kern/arch/mips/vm/coremap.c`: `tlb_replace`
  - Update TLB entry with PTE from page table
    - In `src/kern/arch/mips/vm/coremap.c`: `mmu_map`
OS/161 PAGE TABLES
Because TLB is software-managed, OS can use any page table format it wants.

OS/161 paging uses **virtual memory objects**

- `struct vmobj` defined in `src/kern/include/vmprivate.h`
- A VM object defines a **region** of an address space `addrspace` (created by `as_define_region`)
- A VM object contains a base virtual address and an array of pages

```
struct vm_object {
    struct array *vmo_lpages;
    vaddr_t vmo_base;
    size_t vmo_lower_redzone;
};
```

This is a paging-segmentation hybrid.
Each VM object has an array of logical pages (lpages), one for each virtual page in the region.

```c
struct lpage {
    paddr_t lp_paddr;
    off_t lp_swapaddr;
    struct spinlock lp_spinlock;
};
```

lpage stores where the page is in memory (lp_addr), and where the page is stored in swap when not in main memory (lp_swapaddr)
More on lpages

- If the page is not in main memory, lp_addr is INVALID_PADDR
- If no swap space has been allocated for it, lp_swapaddr is INVALID_SWAPADDR
- Low bits of lp_addr are used to hold flags:
  - LPF_DIRTY is set if the page has been modified
- See src/kern/include/vmprivate.h, and src/kern/vm/lpage.c for lpage functions
PAGE FAULTS
PAGE FAULTS

- **Minor fault**: TLB does not contain a requested PTE
  - Find the Page Table Entry for it and insert the new mapping into the TLB and the coremap (coremap? more on this later...)
  - See coremap.c for a function you can use! (mmu_map)

- **Major fault**: the desired page is not in main memory (it’s either in swap space, or hasn’t been created yet)
  - How do we know if it’s a major fault?
    - lp_addr field of the lpage struct will tell you
    - lp_addr is INVALID_PADDR if the page is not in memory
MORE ON MAJOR FAULTS

- Major fault: desired page is not in memory
  - Page hasn’t been created yet
    - A new page is allocated to the process and initialized (zero-filled) in src/kern/vm/addrspace.c: as_fault
  - Page is in swap
    - We need to swap the page into memory from swap space
    - Need a page of physical memory for the page

- How do we keep track of pages of physical memory? (so we know what we can use...)
  - Coremap!
COREMAP
COREMAP

- We need to track if/how physical pages are being used
- The coremap data structure has one entry per page frame (physical page)
- It’s a “reverse page table”
  - It allows you to use the physical address to find the logical page that inhabits it (NULL if empty)
  - Has bit flags that indicate if pages are kernel pages, pinned (busy), etc.
THE COREMAP AND LPAGES

- Each lpage entry is a logical piece of memory
  - That page may be in memory
  - It may also be in swap (on disk)
  - Each lpage points to the location of its data

- The coremap maps physical memory usage
  - When you need physical memory, consult the coremap to see what memory is free
  - Each entry points to an lpage
On a major fault, you need to swap a page into memory

Need an empty page frame. If memory is full, we need to evict a page to swap.

Two parts to eviction
  - Selecting the page
    - Different algorithms implement different heuristics
  - Replacing the victim page in memory
SELECTING THE PAGE

- Functions involved in eviction:
  - lpage_evict
  - page_replace

- When a physical page is needed, coremap_alloc_one_page(lp, dopin) is called
  - This in turn calls do_page_replace if there is no free page

- In page_replace, the page replacement algorithm (random or sequential) chooses which page is to be evicted
REPLACING THE VICTIM

- Actually replacing the victim in memory
  - Updating the victim’s PTE to show that it is in swap
  - Copying it to disk (iff it is dirty)
  - Evicting (/invalidating) victim’s PTE from the TLB
  - Loading the new page into memory
  - Updating the new page’s PTE and inserting it into the TLB
PAGE REPLACEMENT ALGORITHMS

- First implement a random page replacement algorithm
  - Configure the kernel with ASST4-RAND
- Complete the implementation of A4 (swapping, etc.) with random page replacement before moving on!
- Then implement the sequential
  - Configure the kernel with ASST4-NORAND
- The only difference between ASST4-RAND and ASST4-NORAND is in coremap.c: page_replace