Administrivia

• Recently updated class mailing list
  - If you didn’t get “Second test of class mailing list,” contact cs240c-staff.

• Clarification on “double counting” policy
  - Your class project may coincide with your research or another class project
  - In general this is fine, even encouraged
  - But you must tell me (and advisor or instructor of the other class) in advance to make sure

• I had to move my office hours
  - Now Monday 10am-12pm
Multiprogramming on physical memory

- Makes it hard to allocate space contiguously
  - Convenient for stack, large data structures, etc.
- Need fault isolation between processes
  - (Even Microsoft now seems to believe this...) 
- Processes can consume more than available memory
  - Dormant processes (wating for event) still have core images
Solution: Address Spaces

- Give each program its own address space
- Only privileged software can manipulate mappings
- Isolation is natural
  - Can’t even name other processes’ memory
Alternatives

- **Segmentation**
  - Part of each memory reference implicit in segment register
    \[
    \text{segreg} \leftarrow \langle \text{offset}, \text{limit} \rangle
    \]
  - By loading segment register code can be relocated
  - Can enforce protection by restricting segment register loads

- **Language-level protection (Java)**
  - Single address space for different modules
  - Language enforces isolation

- **Software fault isolation**
  - Instrument compiler output
  - Checks before every store operation prevents modules from trashing each other
Paging

- Divide memory up into small “pages”
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set “dirty” bit
  - Control caching of page
x86 Paging (review)

- Paging enabled by bits in `%cr0`
- Normally 4KB pages
- `%cr3`: points to 4KB page directory
- Page directory: 1024 PDEs (page directory entries)
  - Each contains physical address of a page table
- Page table: 1024 PTEs (page table entries)
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MB of Virtual mem
- INVLPG instruction invalidates page translation
  - Must tell hardware when page table modified
*32 bits aligned onto a 4–KByte boundary

1024 PDE × 1024 PTE = 2²⁰ Pages
x86 Paging Extensions

- **PSE: Page size extensions**
  - Setting bit 7 in PDE make a 4MB translation (no PT)

- **PAE Page address extensions**
  - New 64-bit PTE format allows 36 bits of physical address
  - Page tables, directories have only 512 entries
  - Use 4-entry Page-Directory-Pointer Table to regain 2 lost bits
  - PDE bit 7 allows 2MB translation

- **Long mode PAE**
  - In Long mode, pointers are 64-bits
  - Extends PAE to map 48 bits of virtual address
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Sign Extend</th>
<th>Page−Map Level−4 offset (PML4)</th>
<th>Page Directory−Pointer Offset</th>
<th>Page Directory Offset</th>
<th>Page−Table Offset</th>
<th>Physical−Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48 47 39 38</td>
<td>30 29 21 12 11 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Why are aren’t upper 16 bits of VA used?**
Very different MMU: MIPS

- Hardware has 64-entry TLB
  - References to addresses not in TLB trap to kernel

- Each TLB entry has the following fields:
  Virtual page, Pid, Page frame, NC, D, V, Global

- Kernel itself unpaged
  - All of physical memory contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses

- User TLB fault handler very efficient
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables

- OS is free to choose page table format!
64-bit address spaces

- Straight hierarchical page tables not efficient
- **Solution 1: Guarded page tables [Liedtke]**
  - Omit intermediary tables with only one entry
  - Add predicate in high level tables, stating the only virtual address range mapped underneath + # bits to skip

- **Solution 2: Hashed page tables**
  - Store Virtual $\rightarrow$ Physical translations in hash table
  - Table size proportional to physical memory
  - Clustering makes this more efficient
Superpages

• How should OS make use of “large” mappings
  - x86 has 2/4MB pages that might be useful
  - Alpha has even more choices: 8KB, 64KB, 512KB, 4MB

• Sometimes more pages in L2 cache than TLB entries
  - Don’t want costly TLB misses going to main memory

• Transparent superpage support [Navarro]
  - “Reserve” appropriate physical pages if possible
  - Promote contiguous pages to superpages
  - Does complicate evicting (esp. dirty pages) – demote
OS policy choices (besides page table)

- **Page replacement**
  - Optimal – Least soon to be used (impossible)
  - Least recently used (hard to implement)
  - Random
  - Not recently used

- **Direct-mapped physical caches**
  - Virtual → Physical mapping can affect performance
  - Applications can conflict with each other or themselves
  - Scientific applications benefit if consecutive virtual pages to not conflict in the cache
  - Many other applications do better with random mapping
Paging in day-to-day use

- Demand paging
- Growing the stack
- BSS page allocation
- Shared text
- Shared libraries
- Shared memory
- Copy-on-write (fork, mmap, etc.)
- Q: Which pages should have global bit set on x86?
Example memory layout

- 4 Gig
- 0x0f000000
- USTACKTOP
- First 256MB physical memory
  - kernel text & most data
  - mapped kernel data
  - Invalid Memory
  - user stack
- [mmaped regions]
- break point
- 0
- heap
- BSS
- program data
- program text (read-only)
- Invalid Memory
Early VM system calls

- OS keeps “Breakpoint” – top of heap
  - Memory regions between breakpoint & stack fault
- char *brk (const char *addr);
  - Set and return new value of breakpoint
- char *sbrk (int incr);
  - Increment value of the breakpoint & return old value
- Can implement malloc in terms of sbrk
  - But hard to “give back” physical memory to system
More VM system calls

- `void *mmap (void *addr, size_t len, int prot,
              int flags, int fd, off_t offset)`
  - `prot`: OR of `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`
  - `flags`: shared/private, ...

- `int munmap(void *addr, size_t len)`
  - Removes memory-mapped object

- `int mprotect(void *addr, size_t len, int prot)`
  - Changes protection on pages to or of `PROT_`...

- `int mincore(void *addr, size_t len, char *vec)`
  - Returns in `vec` which pages present
Catching page faults

```
struct sigaction {
    union {
        /* signal handler */
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    };
    sigset_t sa_mask; /* signal mask to apply */
    int sa_flags;
};

int sigaction (int sig, const struct sigaction *act, 
               struct sigaction *oact)
```

- Can specify function to run on SIGSEGV
Example: OpenBSD/i386 siginfo

```c
struct sigcontext {
    int sc_gs;       int sc_fs;       int sc_es;       int sc_ds;
    int sc edi;      int sc esi;      int sc ebp;      int sc ebx;
    int sc edx;      int sc ecx;      int sc eax;

    int sc eip;      int sc cs;       /* instruction pointer */
    int sc eflags;   /* condition codes, etc. */
    int sc esp;      int sc ss;       /* stack pointer */

    int sc onstack;  /* sigstack state to restore */
    int sc mask;     /* signal mask to restore */

    int sc trapno;
    int sc err;
};
```