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 (waiting 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 read or write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set “accessed” and “dirty” bits
  - Control page execute permission separately from read/write
  - Control caching of page
x86 Paging

- Paging enabled by bits in a control register (%cr0)
  - Only privileged OS code can manipulate control registers

- 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

- INVLPAG instruction invalidates page translation
  - Must tell hardware when page table modified
*32 bits aligned onto a 4-KByte boundary
**x86 Paging Extensions**

- **PSE: Page size extensions**
  - Setting bit 7 in PDE makes 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
- 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!
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
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;                  /* signal mask to apply */
};

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;
};
```