Details of paging

The user-level perspective

Case study: 4.4 BSD
Some complications of paging

- **What happens to available memory?**
  - Some physical memory tied up by kernel VM structures
  - E.g., page tables, page metadata

- **What happens to user/kernel crossings?**
  - More crossings into kernel
  - Pointers in syscall arguments must be checked
    (can’t just kill process if page not present—might need to page in)

- **What happens to IPC?**
  - Must change hardware address space
  - Increases TLB misses
  - Context switch flushes TLB entirely on old x86 machines
    (But not on MIPS… Why?)
Some complications of paging

- **What happens to available memory?**
  - Some physical memory tied up by kernel VM structures
  - E.g., page tables, page metadata

- **What happens to user/kernel crossings?**
  - More crossings into kernel
  - Pointers in syscall arguments must be checked
    (can’t just kill process if page not present—might need to page in)

- **What happens to IPC?**
  - Must change hardware address space
  - Increases TLB misses
  - Context switch flushes TLB entirely on old x86 machines
    (But not on MIPS…Why? MIPS tags TLB entries with PID)
64-bit address spaces

- Recall x86-64 only has 48-bit virtual address space
- What if you want a 64-bit virtual address space?
  - Straight hierarchical page tables not efficient
  - But software TLBs (like MIPS) allow other possibilities

Solution 1: Hashed page tables
- Store Virtual → Physical translations in hash table
- Table size proportional to physical memory
- Clustering makes this more efficient [Talluri]

Solution 2: 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
1. Details of paging

2. The user-level perspective

3. Case study: 4.4 BSD
Recall typical virtual address space

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>heap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|        |        |        |        | **breakpoint**

- Dynamically allocated memory goes in heap
- **Top of heap called** breakpoint
  - Addresses between breakpoint and stack all invalid
Early VM system calls

• OS keeps “Breakpoint” – top of heap
  - Memory regions between breakpoint & stack fault on access

  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
### Memory mapped files

<table>
<thead>
<tr>
<th>Memory Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Uninitialized data (bss)</td>
</tr>
<tr>
<td>Initialized data</td>
</tr>
<tr>
<td>Read-only data</td>
</tr>
<tr>
<td>Code (text)</td>
</tr>
</tbody>
</table>

- **Other memory objects between heap and stack**
mmap system call

- void *mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)
  - Map file specified by fd at virtual address addr
  - If addr is NULL, let kernel choose the address

- prot – protection of region
  - PROT_EXEC – executable
  - PROT_READ – readable
  - PROT_WRITE – writable
  - PROT_NONE – inaccessible

- flags
  - MAP_ANON – anonymous memory (fd should be -1)
  - MAP_PRIVATE – modifications are private
  - MAP_SHARED – modifications seen by everyone
More VM system calls

- int msync(void *addr, size_t len, int flags);
  - Flush changes of mmapped file to backing store
- 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
- int madvise(void *addr, size_t len, int advice);
  - Advises the OS regarding the memory behavior
    - MADV_FREE – Kernel can discard the memory
    - MADV_WILLNEED – Will need the memory soon
    - MADV_DONTNEED – Kernel can swap the memory
    - MADV_NORMAL, MADV_SEQUENTIAL, MADV_RANDOM – Hint access pattern
Exposing page faults

- **Signals** are a mechanism to receive notifications from the kernel
- You can think of these as userspace exceptions

```c
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;
};
```

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

- Can specify function to run on **SIGSEGV**
  (Unix signal raised on invalid memory access)
Example: OpenBSD/i386 siginfo

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;
};

- Linux uses ucontext_t – same idea, just uses nested structures that won’t all fit on one slide
VM tricks at user level

- **Combination of** `mprotect/sigaction very powerful`
  - Can use OS VM tricks in user-level programs [Appel&Li]
  - E.g., fault, unprotect page, return from signal handler

- **Technique used in object-oriented databases**
  - Bring in objects on demand
  - Keep track of which objects may be dirty
  - Manage memory as a cache for much larger object DB

- **Other interesting applications**
  - Useful for some garbage collection algorithms
  - Snapshot processes (copy on write)
1. Details of paging

2. The user-level perspective

3. Case study: 4.4 BSD
• Each process has a *vmspace* structure containing
  - *vm_map* – machine-independent virtual address space
  - *vm_pmap* – machine-dependent data structures
  - statistics – e.g. for syscalls like *getrusage()*

• *vm_map* is a linked list of *vm_map_entry* structs
  - *vm_map_entry* covers contiguous virtual memory
  - points to *vm_object* struct

• *vm_object* is source of data
  - e.g. vnode object for memory mapped file
  - points to list of *vm_page* structs (one per mapped page)
  - *shadow objects* point to other objects for copy on write
4.4 BSD VM data structures
Pmap (machine-dependent) layer

- Pmap layer holds architecture-specific VM code
- VM layer invokes pmap layer
  - On page faults to install mappings
  - To protect or unmap pages
  - To ask for dirty/accessed bits
- Pmap layer is lazy and can discard mappings
  - No need to notify VM layer
  - Process will fault and VM layer must reinstall mapping
- Pmap handles restrictions imposed by cache
Example uses

- **vm_map_entry structs for a process**
  - r/o text segment → file object
  - r/w data segment → shadow object → file object
  - r/w stack → anonymous object

- **New vm_map_entry objects after a fork:**
  - Share text segment directly (read-only)
  - Share data through two new shadow objects (must share pre-fork but not post-fork changes)
  - Share stack through two new shadow objects

- **Must discard/collapse superfluous shadows**
  - E.g., when child process exits
What happens on a fault?

- Traverse `vm_map_entry` list to get appropriate entry
  - No entry? Protection violation? Send process a SIGSEGV
- Traverse list of [shadow] objects
- For each object, traverse `vm_page` structs
- Found a `vm_page` for this object?
  - If first `vm_object` in chain, map page
  - If read fault, install page read only
  - Else if write fault, install copy of page
- Else get page from object
  - Page in from file, zero-fill new page, etc.
Paging in day-to-day use

- Demand paging
  - Read pages from \texttt{vm\_object} of executable file

- Copy-on-write (\texttt{fork}, \texttt{mmap}, etc.)
  - Use shadow objects

- Growing the stack, BSS page allocation
  - A bit like copy-on-write for \texttt{/dev/zero}
  - Can have a single read-only zero page for reading
  - Special-case write handling with pre-zeroed pages

- Shared text, shared libraries
  - Share \texttt{vm\_object} (shadow will be empty where read-only)

- Shared memory
  - Two processes \texttt{mmap} same file, have same \texttt{vm\_object} (no shadow)