Address spaces and memory management

• Review of processes
  – Process = one or more threads in an address space
  – Thread = stream of executing instructions
  – Address space = memory space used by threads

• Address space
  – All the memory space the process can use as it runs
  – Hardware interface: one memory of “small” size, shared between processes
  – OS abstraction: each process has its own, larger memory

• Abstractions provided by address spaces
  – Address independence: same numeric address can be used in different address spaces (i.e., different processes), yet remain logically distinct
  – Protection: one process can’t access data in another process’s address space (actually controlled sharing)
  – Virtual memory: an address space can be larger than the machine’s physical memory

Uni-programming

• 1 process runs at a time (viz. one process occupies memory at a time)
• Always load process into same spot in memory (with some space reserved for OS)

<table>
<thead>
<tr>
<th>fffff</th>
<th>operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>80000</td>
<td></td>
</tr>
<tr>
<td>7ffff</td>
<td>user process</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td></td>
</tr>
</tbody>
</table>

• Problems?
Multi-programming and address translation

- Multi-programming: more than 1 process in memory at a time
  - Need to support address translation
  - Need to support protection
- Must translate addresses issued by a process, so they don’t conflict with addresses issued by other processes
  - Static address translation: translate addresses before execution (i.e., translation doesn’t change during execution)
  - Dynamic address translation: translate addresses during execution (i.e., translation can change during execution)
- Is it possible to run two processes at the same time and provide address independence with only static address translation?

- Does static address translation achieve the other address space abstractions?

- Achieving all the address space abstractions requires doing some work on each memory reference
Dynamic address translation

- Translate every memory reference from virtual address to physical address
  - virtual address: an address issued by the user process
  - physical address: an address used to access the physical memory

- Translation enforces protection
  - One process can’t refer to another process’s address space

- Translation enables virtual memory
  - A virtual address only needs to be in physical memory when it’s being accessed
  - Change translations on the fly; different virtual addresses occupy the same physical memory

Address translation

- Many ways to implement translator

- Tradeoffs
  - Flexibility (sharing, growth, virtual memory)
  - Size of data needed to support translation
  - Speed of translation
Base and bounds

- Load each process into contiguous region of physical memory. Prevent each process from accessing data outside its region
  
  ```
  if (virtual address > bound) {
      trap to kernel; kill process (core dump)
  } else {
      physical address = virtual address + base
  }
  ```

- Process has illusion of dedicated memory [0, bound)

Base and bounds

- Similar to linker-loader, but also protects processes from each other
- Only kernel can change translation data (base and bounds)
- What does it mean to change address spaces?
  - Changing data used to translate addresses (base and bounds registers)

Base and bounds

- What to do when address space grows?

- Low hardware cost (2 registers, 1 adder, 1 comparator)
- Low overhead (add and compare on each memory reference)
Base and bounds

- A single address space can’t be larger than physical memory
  - But sum of all address spaces can be larger than physical memory
    - Swap current address space out to disk; swap address space for new process in
- Can’t share part of an address space between processes

External fragmentation

- Processes come and go, leaving a mishmash of available memory regions
  - This is called “external fragmentation”: wasted memory between allocated regions

Process 1 start: 100 KB (physical addresses 0-99 KB)
Process 2 start: 200 KB (physical addresses 100-299 KB)
Process 3 start: 300 KB (physical addresses 300-599 KB)
Process 4 start: 400 KB (physical addresses 600-999 KB)

Process 3 exits: physical addresses 300-599 KB now free
Process 5 starts: 100 KB (physical addresses 300-399 KB)
Process 1 exits: physical 0-99 KB now free

Process 6 start: 300 KB. No contiguous space large enough.
- May need to copy memory region to new location
Base and bounds

- Hard to grow address space

Segmentation

- Segment: a region of contiguous memory (contiguous in physical memory and in virtual address space)
- Base and bounds used a single segment. We can generalize this to multiple segments, described by a table of base and bounds pairs

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Base</th>
<th>Bounds</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4000</td>
<td>700</td>
<td>code segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>500</td>
<td>data segment</td>
</tr>
<tr>
<td>2</td>
<td>n/a</td>
<td>n/a</td>
<td>unused</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>1000</td>
<td>stack segment</td>
</tr>
</tbody>
</table>

- In segmentation, a virtual address takes the form: (segment #, offset)
- Many ways to specify the segment #
  - High bits of address
  - Special register
  - Implicit to instruction opcode
Not all virtual addresses are **valid**
- Valid means that the region is part of the process’s address space
- Invalid means the virtual address is illegal to access. Accessing an invalid virtual address causes a trap to OS (usually resulting in core dump)
- E.g., no valid data in segment 2; no valid data in segment 1 above 4ff)

Protection: different segments can have different protection
- E.g., code is usually read only (allows instruction fetch, load)
- E.g., data is usually read/write (allows instruction fetch, load, store)
- Protection in base and bounds?

What must be changed on a context switch?
Segmentation

- Works well for sparse address spaces. Regions can grow independently.
- Easy to share segments, without sharing entire address space
- But complex memory allocation

- Can a single address space be larger than physical memory?

Paging

- Allocate physical memory in fixed-size units (called pages)
  - Fixed-size unit is easy to allocate
  - Any free physical page can store any virtual page
- Virtual address
  - Virtual page # (high bits of address, e.g., bits 31-12)
  - Offset (low bits of address, e.g., bits 11-0, for 4 KB page size)
- Translation data is the page table

<table>
<thead>
<tr>
<th>Virtual page #</th>
<th>Physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>invalid</td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
<tr>
<td>1048575</td>
<td>invalid</td>
</tr>
</tbody>
</table>
Paging

• **Translation process**

```java
if (virtual page is invalid or non-resident or protected) {
    trap to OS fault handler
} else {
    physical page # = pageTable[virtual page #].physPageNum
}
```

• What must be changed on a context switch?

• Each virtual page can be in physical memory or “paged out” to disk

---

Paging

• How does processor know that a virtual page is not in physical memory?

• Like segments, pages can have different protections (e.g., read, write, execute)
Valid versus resident

• Valid means a virtual page is legal for the process to access. It is (usually) an error for the process to access an invalid page.
  – Who makes a virtual page valid/invalid?

  – Why would a process want one of its virtual pages to be invalid?

• Resident means a virtual page is in physical memory. It is not an error for a process to access a non-resident page.
  – Who makes a virtual page resident/non-resident?

Paging

• Page size
  – What happens if page size is really small?

  – What happens if page size is really big?

  – Typically a compromise, e.g., 4 KB or 8 KB. Some architectures support multiple page sizes.
• How well does paging handle two growing regions?
Paging

• Simple memory allocation
• Flexible sharing
• Easy to grow address space

• But large page tables

• Summary
  – Base and bound: unit of translation and swapping is an entire address space
  – Segmentation: unit of translation and swapping is a segment (a few per address space)
  – Paging: unit of translation and swapping/paging is a page

• How to modify paging to reduce space needed for translation data?

Multi-level paging

• Standard page table is a simple array. Multi-level paging generalizes this into a tree.
• E.g., two-level page table
  – Index into level 1 page table using virtual address bits 31-22
  – Index into level 2 page table using virtual address bits 21-12
  – Page offset: bits 11-0 (4 KB page)
  – What information is stored in the level 1 page table?
  – What information is stored in the level 2 page table?

• How does this allow the translation data to take less space?
### Multi-level paging

#### Level 1 Page Table

<table>
<thead>
<tr>
<th>virtual address bits 21-12</th>
<th>physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Level 2 Page Tables

<table>
<thead>
<tr>
<th>virtual address bits 21-12</th>
<th>physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- How to share memory when using multi-level page tables?
- What must be changed on a context switch?
- Space efficient for sparse address spaces
- Easy memory allocation
- Flexible sharing
- But two extra lookups per memory reference
Translation lookaside buffer (TLB)

- Translation when using paging involves 1 or more additional memory references. How to speed up the translation process?

- TLB caches PTEs
  - If TLB contains the entry you’re looking for, you can skip all the translation steps
  - On TLB miss, get the PTE, store in the TLB, then restart the instruction

- Does this change what happens on a context switch?

Replacement policies

- Which page to evict when you need a free page?
  - Goal: minimize page faults

- Random
- FIFO
  - Replace page that was brought into memory the longest time ago
  - But this may replace pages that continue to be frequently used
- OPT
  - Replace page that won’t be used for the longest time in the future
  - This minimizes misses, but requires knowledge of the future
Replacement policies

• LRU (least recently used)
  – Use past reference pattern to predict future (temporal locality)
  – Assumes past is a mirror image of the future. If page hasn’t been used for a long time in the past, it probably won’t be used for a long time in the future.
  – Approximates OPT
  – But still hard to implement exactly
  – Can we make LRU easier to implement by approximating it?

Clock replacement algorithm

• Most MMUs maintain a “referenced” bit for each resident page
  – Set by MMU when page is read or written
  – Can be cleared by OS
• Why maintain reference bit in hardware?
• How to use reference bit to identify old pages?
• How to do work incrementally, rather than all at once?
Clock replacement algorithm

• Arrange resident pages around a clock

• Algorithm
  – Consider page pointed to by clock hand
  – If not referenced, page hasn’t been accessed since last sweep. Evict.
  – If referenced, page has been referenced since last sweep. What to do?

  – Can this infinite loop? What if all pages have been referenced since last sweep?

  – New pages put behind the clock hand and marked referenced

Eviction

• What to do with page when it’s evicted?

• Why not write page to disk on every store?
Eviction optimizations

• While evicted page is being written to disk, the page being brought into memory must wait
  – How might you optimize the eviction system (but don’t use these for Project 3)?

Page table contents

• Physical page number
  – Written by OS; read by MMU
• Resident: is virtual page in physical memory
  – Written by OS; read by MMU
• Protection (readable, writable)
  – Written by OS; read by MMU
• Dirty: has virtual page been written since dirty bit was last cleared?
  – Written by OS and MMU; read by OS
• Referenced: has virtual page been read or written since reference bit was last cleared?
  – Written by OS and MMU; read by OS

• Does hardware page table need to store disk block # for non-resident virtual pages?
MMU algorithm

- MMU does work on each memory reference (load, store)
  
  if (virtual page is non-resident or protected) {
    trap to OS fault handler
    retry access
  } else {
    physical page # = pageTable[virtual page #].physPageNum
    pageTable[virtual page #].referenced = true
    if (access is write) {
      pageTable[virtual page #].dirty = true
    }
    access physical memory
  }

- Project 3 infrastructure implements (a subset of) the MMU algorithm

Software dirty and referenced bits

- Do we need MMU to maintain dirty bit?

- Do we need MMU to maintain referenced bit?
User and kernel address spaces

- You can think of (most of) the kernel as a privileged process, with its own address space

Where is translation data stored?

- Page tables for user and kernel address spaces could be stored in physical memory, i.e., accessed via physical addresses

- Page tables for user address spaces could be stored in kernel's virtual address space

  - Benefits?
  - Project 3: translation data for user address spaces stored in kernel virtual address space; kernel virtual address space managed by infrastructure (Linux)
Kernel versus user address spaces

• Can you evict the kernel’s virtual pages?

• How can kernel access specific physical memory addresses (e.g., to refer to translation data)?
  – Kernel can issue untranslated address (bypass MMU)
  – Kernel can map physical memory into a portion of its address space (vm_physmem in Project 3)

How does kernel access user’s address space?

• Kernel can manually translate a user virtual address to a physical address, then access the physical address

• Can map kernel address space into every process’s address space

| fffff | operating system |
| 80000 |               |
| 7ffff | user process   |
| 00000 |               |

  – Trap to kernel doesn’t change address spaces; it just allows computer to access both OS and user parts of that address space
Kernel versus user mode

- Who sets up data used by translator?

- CPU must distinguish between kernel and user instructions
  - Access physical memory
  - Issue privileged instructions
- How does CPU know kernel is running?
  - Mode bit (kernel mode or user mode)
- Recap of protection
  - Protect address spaces via translation. How to protect translation data?
  - Distinguish between kernel and user mode. How to protect mode bit?

Switching from user process into kernel

- Who can change mode bit?

- What causes a switch from user process to kernel?
System calls

• From C++ program to kernel
  – C++ program calls `cin`
  – `cin` calls `read()`
  – `read()` executes assembly-language instruction `syscall`
  – `syscall` traps to kernel at pre-specified location
  – kernel’s syscall handler receives trap and calls kernel’s `read()`

• Trap to kernel
  – Set mode bit to kernel
  – Save registers, PC, SP
  – Change SP to kernel stack
  – Change to kernel’s address space
  – Jump to exception handler

Passing arguments to system calls

• Can store arguments in registers or memory
  – Which address space holds the arguments?

• Kernel must carefully check validity of arguments
Interrupt vector table

- Switching from user to kernel mode is safe(r) because control can only be transferred to certain locations
- Interrupt vector table stores these locations
- Who can modify interrupt vector table?
- Why is this easier than controlling access to mode bit?
- What are administrative processes (e.g., processes owned by root)?

Process creation

- Steps
  - Allocate process control block
  - Initialize translation data for new address space
  - Read program image from executable into memory
  - Initialize registers
  - Set mode bit to “user”
  - Jump to start of program
- Need hardware support for last steps (e.g., “return from interrupt” instruction)
- Switching from kernel to user process (e.g., after system call) is same as last few steps
Multi-process issues

• How to allocate physical memory between processes?
  – Fairness versus efficiency

• Global replacement
  – Can evict pages from this process or other processes

• Local replacement
  – Can evict pages only from this process
  – Must still determine how many pages to allocate to this process

Thrashing

• What would happen if many large processes all actively used their entire address space?

• Performance degrades rapidly as miss rate increases
  – Average access time = hit rate * hit time + miss rate * miss time
  – E.g., hit time = .0001 ms; miss time = 10 ms
    • Average access time (100% hit rate) = .0001 ms
    • Average access time (1% hit rate) =
    • Average access time (10% hit rate) =

• Solutions to thrashing
Working set

• What do we mean by “actively using”?  
• Working set = all pages used in last T seconds  
  – Larger working set $\Rightarrow$ process needs more physical memory to run well (i.e., avoid thrashing)  
• Sum of all working sets should fit in memory  
  – Only run subset of processes that fit in memory  

• How to measure size of working set?

Case study: Unix process creation

• Two steps  
  – fork: create new process with one thread. Address space of new process is copy of parent process’s address space.  
  – exec: replace new process’s address space with data from program executable; start new program  
  – Separating process creation into two steps allows parent to pass information to child  
• Problems
Implementing a shell

- Text and graphical shells provide the main interface to users
- How to write a shell program?