CSCI-GA.2250-001
Operating Systems
Lecture 7: Memory Management III

Hubertus Franke
frankeh@cs.nyu.edu
Summary of Paging

• Virtual address space bigger than physical memory
• **Mapping** virtual address to physical address
• Virtual address space divided into fixed-size units called **pages**
• Physical address space divided into fixed-size units called pages **frames**
• Virtual address space of a process can be non-contiguous in physical address space
Paging

MMU

OS Involvement

The CPU sends virtual addresses to the MMU

The MMU sends physical addresses to the memory
Paging

- MMU
- OS Involvement

**Diagram:**
- CPU sends virtual addresses to the MMU
- The MMU sends physical addresses to the memory
- Memory management unit

**Question:**
What is the role of OS Involvement in Paging?
OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs
OS Involvement With Paging

• When a new process is created
  – Determine how large the program and data will be (initially)
  – Create page table
  – Allocate space in memory for page table
  – Record info about page table and swap area in process table

• When a process is scheduled for execution
• When process exits
• When page fault occurs
OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs
OS Involvement With Paging

• When a new process is created
• When a process is scheduled for execution
  – MMU reset for the process
  – TLB flushed
  – Process table made current
• When process exits
• When page fault occurs
OS Involvement With Paging

• When a new process is created
• When a process is scheduled for execution
• When process exits
  – OS releases the process page table
  – Frees its pages and disk space
• When page fault occurs
OS Involvement With Paging

• When a new process is created
• When a process is scheduled for execution
• When process exits
• When page fault occurs
Page Fault Handling

1. The hardware:
   - Saves program counter
   - Traps to kernel

2. An assembly routine saves general registers and calls OS

3. OS tried to discover which virtual page is needed

4. OS checks address validation and protection and assign a page frame (page replacement may be needed)
Page Fault Handling

5. If page frame selected is dirty
   - Page scheduled to transfer to disk
   - Frame marked as busy
   - OS suspends the current process
   - Context switch takes place

6. Once the page frame is clean
   - OS looks up disk address where needed page is
   - OS schedules a disk operation
   - Faulting process still suspended

7. When disk interrupts indicates page has arrived
   - OS updates page table
Page Fault Handling

8. Faulting instruction is backed up to its original state before page fault and PC is reset to point to it.

9. Process is scheduled for execution and OS returns to the assembly routine.

10. The routine reloads registers and other state information and returns to user space.
Instruction Backup At Page Fault

- In order to restart the instruction, the OS needs to know where the first byte of the instruction is.

```
MOVE.L #6(A1), 2(A0)
```

<table>
<thead>
<tr>
<th></th>
<th>Opcode</th>
<th>First operand</th>
<th>Second operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>MOVE</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

16 Bits

The value of the PC at the time of trap depends on which operand faulted and the CPU microcode.
Instruction Backup At Page Fault

- In order to restart the instruction, the OS needs to know where the first byte of the instruction is.

MOVE.L #6(A1), 2(A0)

<table>
<thead>
<tr>
<th>16 Bits</th>
<th>Opcode</th>
<th>First operand</th>
<th>Second operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>MOVE</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

The value of the PC at the time of trap depends on which operand faulted and the CPU microcode.

The CPU designers usually saves PC in some internal register for the OS.
Interesting Scenario: Virtual Memory & I/O Interaction

- Process issues a syscall to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a change the page containing the buffer be removed from memory.
- The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page!
Interesting Scenario: Virtual Memory & I/O Interaction

• Process issues a syscall to read a file into a buffer
• Process suspended while waiting for I/O
• New process starts executing
• This other process gets a page fault
• If paging algorithm is global there is a chance the page containing the buffer be removed from memory.
• The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page!

One solution: Locking (pinning) pages engaged in I/O so that they will not be removed.
Backing Store

- Swap area: does not a normal file system on it.
- Associated with each process the disk address of its swap area; store in the process table
- Before process starts swap area must be initialized
  - One way: copy all process image into swap area [static swap area]
  - Another way: don’t copy anything and let the process swap out [dynamic]
- Instead of disk partition, one or more preallocated files within the normal file system can be used [Windows uses this approach.]
Backing Store

(a) Static Swap Area

(b) Dynamic
Page Fault Handler

Figure 9-5. The flow diagram of the Page Fault handler
Memory Mappings

• Each process consists of many memory areas:
  – Aka:
    • segments
    • regions
    • VMAs virtual memory areas.
  – Ex: Heap, stack, code, data, ronly-data, etc.

• Each has different characteristics
  – Protection (executable, rw, rdonly )
  – Fixed, can grow (up or down) [ heap, stack ]

• Each process can have 10s-100s of these.
Example: emacs VMAs

```
06110000-001a3000 r-xp 00000000 08:01 267740 /usr/lib/libgdk-x11-2.0.so.0.2000.1
061a3000-001a5000 r--p 00093000 08:01 267740 /usr/lib/libgdk-x11-2.0.so.0.2000.1
061a5000-001a6000 rw-p 00095000 08:01 267740 /usr/lib/libgdk-x11-2.0.so.0.2000.1
061a6000-001bf000 r-xp 00000000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
061bf000-001c0000 ---p 00190000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
061c0000-001c1000 r--p 00190000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
061c1000-001c2000 rw-p 001a0000 08:01 267488 /usr/lib/libatk-1.0.so.0.3009.1
061c2000-001cc000 r-xp 00000000 08:01 265243 /usr/lib/libpangocairo-1.0.so.0.2800.0
061cc000-001cd000 r--p 00099000 08:01 265243 /usr/lib/libpangocairo-1.0.so.0.2800.0
061cd000-001ce000 rw-p 0009a000 08:01 265243 /usr/lib/libpangocairo-1.0.so.0.2800.0
061ce000-001d1000 r-xp 00000000 08:01 267773 /usr/lib/libgmodule-2.0.so.0.2400.1
061d1000-001d2000 r--p 00002000 08:01 267773 /usr/lib/libgmodule-2.0.so.0.2400.1
061d2000-001d3000 rw-p 00003000 08:01 267773 /usr/lib/libgmodule-2.0.so.0.2400.1
061d3000-001e8000 r-xp 00000000 08:01 267367 /usr/lib/libICE.so.0.6.3.0
061e8000-001e9000 r--p 00140000 08:01 267367 /usr/lib/libICE.so.0.6.3.0
061e9000-001ea000 rw-p 00150000 08:01 267367 /usr/lib/libICE.so.0.6.3.0
061ea000-001ec000 rw-p 00000000 00:00 0
061ec000-001fb000 r-xp 00000000 08:01 267433 /usr/lib/libXpm.so.0.4.11.0
061fb000-001fc000 r--p 00000000 08:01 267433 /usr/lib/libXpm.so.0.4.11.0
061fc000-001fd000 rw-p 00000000 00:00 0
075fc000-b763b000 r--p 00000000 08:01 395228 /usr/lib/locale/en_US.utf8/LC_TYPE
b763b000-b763c000 r--p 00000000 08:01 395233 /usr/lib/locale/en_US.utf8/LC_NUMERIC
b763c000-b763d000 r--p 00000000 08:01 404948 /usr/lib/locale/en_US.utf8/LC_TIME
b763d000-b775b000 r--p 00000000 08:01 395227 /usr/lib/locale/en_US.utf8/LC_COLLATE
b775b000-b776c000 r--p 00000000 00:06 0
b776c000-b776d000 r--p 00000000 08:01 404949 /usr/lib/locale/en_US.utf8/LC_MONETARY
b776d000-b776e000 r--p 00000000 08:01 404950 /usr/lib/locale/en_US.utf8/LC_MESSAGES/SYS_LC_MESSAGES
b776e000-b776f000 r--p 00000000 08:01 395442 /usr/lib/locale/en_US.utf8/LC_PAPER
b776f000-b7770000 r--p 00000000 08:01 395102 /usr/lib/locale/en_US.utf8/LC_NAME
b7770000-b7771000 r--p 00000000 08:01 404951 /usr/lib/locale/en_US.utf8/LC_ADDRESS
b7771000-b7772000 r--p 00000000 08:01 404952 /usr/lib/locale/en_US.utf8/LC_TELEPHONE
b7772000-b7773000 r--p 00000000 08:01 395529 /usr/lib/locale/en_US.utf8/LC_MEASUREMENT
b7773000-b777a000 r--p 00000000 08:01 269322 /usr/lib/gconv/gconv-modules.cache
b777a000-b777b000 r--p 00000000 08:01 404953 /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
b777b000-b777d000 rw-p 00000000 00:00 0 [stack]
```
More on memory regions

**Anonymous memory:**
- Swap space required

**Memory mapped files:**
- Typically **no** swap space required
- The file is the swap space

### NAME
mmap, munmap - map or unmap files or devices into memory

### SYNOPSIS
```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

### DESCRIPTION
mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.
Organization of Memory Regions

- Cells are by default non-overlapping
  - Called VMA (Virtual Memory Areas)
- Organized as AVL trees
- Identify in $O(\log N)$ time during pgfault
  - $\text{Pgfault}(\text{vaddr}) \rightarrow \text{VMA}$
- Rebalanced when VMA is added or deleted
Real Life Program: A Compiler

Virtual address space

Call stack

Address space allocated to the parse tree

Parse tree

Space currently being used by the parse tree

Constant table

Symbol table has bumped into the source text table

Free

Source text
Wouldn’t it be much easier to have separate address spaces for each type?
Segmentation

- Provide the machine with many completely independent address spaces.
- Each segment consists of linear sequence of addresses.
- Different segments can have different lengths.
- Segment length may change during execution.
- A segment is a logical entity.
- Programmer is aware of segments.
One dimensional address space

Segmented memory
External Fragmentation (Checkerboarding)
<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection</td>
</tr>
</tbody>
</table>
Segmentation with Paging

• It may be difficult to keep full segments in memory at once
• Paging segments can help
• Example from real life
  – MULTICS
  – Intel Pentium
MULTICS

• MULtiplexed Information & Computing Service (MULTICS)
• Time Sharing OS (1965 - 2000)
• Provides each program with up to $2^{18}$ segments
• Each segment up to 65,536 words (word size is 36 bits in this system)
• Each program has a segment table with one descriptor per table
MULTICS

- Segment descriptor contains indication whether segment is in memory or not
- If segment is in memory, the descriptor contains 18-bit pointer to its page table
MULTICS

Virtual address
MULTICS

MULTICS virtual address

Segment number

Page number
Offset

Descriptor segment
Page table
Page

Segment number
Descriptor number
Page number
Offset

Word
The Intel Pentium

- 16K independent segments
- Each can hold up to 1 billion 32-bit word
- The heart of the system consists of two tables:
  - LDT (Local Descriptor Table): each program has its own LDT
  - GDT (Global Descriptor Table): for system segments
- To access a segment, a Pentium program first loads a selector for that segment into one of the machine's six segment registers
The Intel Pentium

- At the time the selector is loaded into a segment register, the corresponding descriptor is fetched from LDT or GDT and stored in microprogram registers.
The Intel Pentium

Address = (Segment, Offset)

Selector

Descriptor

- Base address
- Limit
- Other fields

Offset

32-Bit linear address
The Intel Pentium

Each running program has a page directory.
4 level PgTable
Conclusions

• We are done with Chapter 3
• Main goal
  – Provide CPU with illusion of large and fast memory
• Constraints
  – Speed
  – Cost
  – Protection
  – Transparency
  – Efficiency
• Memory management
  – Paging
  – Segmentation
  – Paged segments