CMSC412 Discussion

11/07/2012

(note: based on Daozheng's slides from last semester)
Overview

• Project 4
  • using virtual memory instead of segmentation
    • set up for this in Part I
  • userseg.c -> uservm.c
  • demand paging and paging to disk
Project 4

• Part II
  • User Memory Mapping
  • Demand Paging and Paging to Disk
  • Recommendations:
    • Start the coding from copying functionality from userseg.c to uservm.c
    • Add pageability later
Part II – User Memory Mapping

- Kernel Memory:
  - Start of kernel memory: 0x00000000

- User Memory:
  - Start of User memory (unmapped page): 0x80000000
  - Text segment usually load here: (segment->startAddress)
  - Initial stack on the top of this page: 0xFFFF F000
  - Argument in this page: 0xFFFFFFFF
Copy From User & Copy To User

- bool Copy_From_User(void *destInKernel, ulong_t srcInUser, ulong_t bufSize)
- bool Copy_To_User(ulong_t destInUser, void *srcInKernel, ulong_t bufSize)

- given from/to addresses + size, copy data around
- key difference: address translation
- page may be paged out to disk (can worry about later)
Copy_From_User & Copy_To_User

• “User pages may need to be paged in from disk before being accessed.”

• Page_Fault_Handler

• “Before you touch (read or write) any data in a user page, **disable the PAGE_PAGEABLE bit**.”

• Section “Copy Data Between Kernel and User Memory”
Copy_From_User & Copy_To_User

• May want to write a claim_a_page function which does the following
  • Given a linear address in user space
    • Allocate a page for this linear address if it is not present
    • Disable the PAGE_PAGEABLE bit to make the page not able to be paged out to disk
    • Need to access user context’s page directory

• May want to write an unclaim_a_page function which enables the PAGE_PAGEABLE bit
Copy_From_User & Copy_To_User (cont.)

• “Be very careful with race conditions in reading a page from disk. Kernel code must always assume that if the struct Page for a page of memory has the PAGE_PAGEABLE bit set, IT CAN BE STOLEN AT ANY TIME. The only exception is if interrupts are disabled; because no other process can run, the page is guaranteed not to be stolen.”

• You may choose to do this outside the claim_a_page or the unclaim_a_page functions.
  
  • In case multiple pages need to be claimed or unclaimed, an atomic section needs to be enable only once.
  
  • Note: an atomic section is not enabled when copying the data
Copy_From_User & Copy_To_User (cont.)

• All pages should be claimed before data copying starts.
• All pages should be unclaimed after data copying finishes.
Create_User_Context

- Linear memory space is identical for all processes now
- Base address is always 0x8000 0000
- Size is always 0x8000 0000
- User context’s page directory (pageDir) is used in paging to validate and map user memory accesses to physical memory.
Destroy_User_Context

• Free stuff
• Before: free malloc'd memory
• Now: free pages
Switch_To_Address_Space

- Spec:
  - "You will also need to add code to switch the PDBR (cr3) register as part of a context switch. For this, in Switch_To_Address_Space you should add a call to Set_PDBR (provided for you in lowlevel.asm), after you load the LDT. You will use the pageDir field in the User_Context structure that will store the address of the process's page directory."
Read the Source!
Load_User_Program Overview

- Find maximum virtual address
- Determine size for argument block
- Determine size needed for memory block (to run process)
- Create User_Context(size)
- Load segment data into memory
- Format argument block
- Fill in code entry point
- Fill in addresses of argument block and stack
Load_User_Program Overview

• Find maximum virtual address
• Determine size for argument block
• Determine size needed for memory block (to run process)
• Create User_Context(size)
• Load segment data into memory
• Format argument block
• Fill in code entry point
• Fill in addresses of argument block and stack
Load_User_Program Overview

- Find maximum virtual address
- Determine size for argument block
- Determine size needed for memory block (to run process)
- Create User_Context(size)
- Load segment data into memory
- Format argument block
- Fill in code entry point
- Fill in addresses of argument block and stack
Load_User_Program Overview

- Determine size for argument block
- Create User_Context
  - also need to initialize pageDir
- Load segment data into memory
  - bit more complicated since not malloc'd
  - and now everything is at specific locations
- Format argument block
  - again, everything is at specific location
- Fill in code entry point
- Fill in addresses of argument block and stack
  - again, specific locations (Note: "userContext stuff" is in user (logical) addresses)
Load_User_Program

- Allocate page directory for a user process (Alloc_Page)
  - userContext->pageDir
    - Contain entries to address the kernel memory
    - Copy entries value of the bottom half of the kernel page directory (0-2GB for kernel space)
- Set the PDBR to be the newly allocate page directory (Why?)
- Allocate pages for segment data and copy values (Alloc_Pageable_Page)
  - Is it similar to what Copy_To_User is doing?
  - Claim all pages for the segment data, copy the data, and then unclaim all pages (claim_a_page & unclaim_a_page)
Load_User_Program (cont.)

• Allocate the page for argument block and stack and format argument block (Alloc_Pageable_Page)
  • Claim the page argument block, format the block, unclaim the page (claim_a_page & unclaim_a_page)
  • Which page to claim? (figure on slide 4)
• Allocate the initial page for the stack
• Update argBlockAddr and stackPointerAddr
• Set the PDBR back to be the original page directory (Why?)
Page Fault Error Codes

Interrupt 14—Page-Fault Exception (#PF) (Continued)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>P</strong></td>
</tr>
</tbody>
</table>

**P**
- 0  The fault was caused by a non-present page.
- 1  The fault was caused by a page-level protection violation.

**W/R**
- 0  The access causing the fault was a read.
- 1  The access causing the fault was a write.

**U/S**
- 0  The access causing the fault originated when the processor was executing in supervisor mode.
- 1  The access causing the fault originated when the processor was executing in user mode.

**RSVD**
- 0  The fault was not caused by reserved bit violation.
- 1  The fault was caused by reserved bits set to 1 in a page directory.

Figure 5-7. Page-Fault Error Code
Part II – Demand Paging

• Two valid fault conditions
  • The fault is within one page of current stack limit
  • The page is on disk
• Otherwise
  • Process termination
• Test: use rec.c to trigger a fault (memory pressure by stack expansion)
Within One Page of Current Stack Limit

• May want to create a new stack limit field
• Initialize it to be proper value in `Load_User_Program`
• Detect the within-one-page condition in `Page_Fault_Handler` and allocate the new stack page
• Update this field accordingly
Within One Page of Current Stack Limit

User Memory

Fault at an address in this page

Allocate a new page for stack
Pages are on Disk

• Condition to check: kernelInfo == KINFO_PAGE_ON_DISK
• Allocate a new page (Alloc_Pageable_Page)
• Read the contents of the indicated block of space in the paging file into the allocated page (Read_From_Paging_File)
• Update the relevant page table entry
• Free the page-sized chunk of disk space in the paging file (Free_Space_On_Paging_File)
Paging to Disk

- Allocate_Pageable_Page
  - has codes to page out a page
  - Find_Space_On_Paging_File & Write_To_Paging_File
- Find your own way to manage paging file.
  - Write Init_Paging to initialize paging file structure
    - Call it in main.c (which location?)
  - Paging_Device (Get_Paging_Device)
    - The block device for paging file, the 1\textsuperscript{st} disk block number, the number of disk blocks
  - To read and write the paging files
    - Block_Read and Block_Write
    - Initial blockNum for a page
      - \([\text{the 1}\text{st block number}] + 8*\text{pageTable->pageBaseAddr}\)
      - Need to ensure a page cannot be stolen (why?)
Paging files

• How many pages are on disk?
• Disk read and write is block by block

startSector

1 page = 8 consecutive disk blocks

A block
Pseudo-LRU

• Optional implementation
• LRU in theory: in textbook (somewhere)
• http://en.wikipedia.org/wiki/Page_replacement_algorithm