Page Fault Handling with Swapping

1. MMU (TLB)
2. Page fault
3.1 (if phys mem full) swap out a victim page to disk
3.2 Update PTE of victim pg, flush TLB
4.1 Swap in access page from disk
4.2 Update PTE of access pg
5. Resume faulting intr
Page fault handling (cont)

• On a page fault
  ♦ Find an unused phy page. If no unused, find a used phy. page (policy on which used one to pick in future lecture)
  ♦ If the phy. page is used
    » If it has been modified (how to know?), write it to disk
    » Invalidate its current PTE and TLB entry (how?)
  ♦ Load the new page from disk
  ♦ Update the faulting PTE and its TLB entry (how?)
  ♦ Restart the faulting instruction

• Supporting data structure that an OS uses
  ♦ For speed: A list of unused physical pages (more later)
  ♦ Data structure to map a phy. page to its pid(s) and virtual address(es)
  ♦ Data structure to remember where a swapped out page is on disk
Address Translation Redux

• We started this topic with the high-level problem of translating virtual addresses into physical addresses
• We’ve covered all of the pieces
  ♦ Virtual and physical addresses
  ♦ Virtual pages and physical page frames
  ♦ Multi-level page tables and page table entries (PTEs)
  ♦ TLBs
  ♦ On-demand allocation and swapping
• Now let’s put it together, bottom to top
TLB Misses

- At this point, two other things can happen
  1. TLB does not have this virtual address
  2. Mapping in TLB, but memory access violates protection bits or the invalid bit is set
- We’ll consider each in turn
Reloading the TLB

• If the TLB does not have mapping, two possibilities:
  MMU loads PTE from page table in memory (a page table walk)
    » Hardware managed TLB, OS not involved in this step

• When TLB has PTE, it restarts translation
  ♦ Common case is that the PTE refers to a valid page in memory
    » Hardware just reads PTE from the page table and loads it into TLB
  ♦ Uncommon case is that TLB faults again on PTE because of PTE protection/valid bits (e.g., page is invalid (not in memory))
    » Becomes a page fault…
Page Faults

- PTE can indicate the type of a page fault
  - Read/write/execute – operation not permitted on page
  - Invalid – page not in physical memory
- Traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to user process, or use for other purposes (e.g., copy on write, more next time)
  - Invalid
    - Page not in physical memory because this is the first access
      - OS allocates physical frame and sets up the PTE (and flush TLB)
    - Page not in physical memory because it has been swapped out
      - Finds an empty frame in physical memory (if none, need to swap out something first), reads the page from disk, sets up the PTE to point to the new physical frame (and flush TLB)
Memory Allocation

- Virtual memory allocation
- Physical memory allocation
- Who performs these allocations?
- How are the allocations done?
Virtual memory allocation: two general forms

- **Stack**
  - Restricted
  - Simple and efficient
  - Easy to implement

- **Heap**
  - More general
  - Less efficient
  - More difficult to implement
Heap organization

• Allocation & freeing are unpredictable
  ♦ For arbitrary, complex data structures

• Memory consists of allocated areas and free areas (holes) → lots of holes inevitable

• Fragmentation problem
  ♦ solution: keep # of holes small, size large
Heap organization

- **Fragmentation**: inefficient use of memory due to holes too small
  - What happens in stack?

- Typically, heap allocation uses a *free list (or tree)* of holes
- Allocation algorithms differ in how to manage the free list
Two system calls for allocating heap virtual memory

• **brk**
  - Grow heap by certain size

• **mmap**
  - With MAP_ANONYMOUS flag
  - Allocate a chunk of virtual memory of certain size, the starting virtual memory address can be anywhere (unless specifies MAP_FIXED)

• Who calls these system calls?
Implementation of virtual memory allocation

• Usually uses some tree data structure to keep track of free/used spaces
  ♦ Linux vma (virtual memory area) tree
  ♦ Vma tree also contains information about which allocated range has what permission
  ♦ Need some algorithm to find a hole (free space) that’s big enough (and would lead to less fragmentation)
Implementation of physical memory allocation

• Do we need any special algorithm for allocating physical memory (if using paging)?

• How do we keep track of free physical page frames?
  ♦ Bit map
  ♦ Linked list
malloc, brk, and physical memory allocation

- Who calls malloc?
- What happens at malloc time?

- What is brk?
- Who calls brk?
- What happens at brk time?

- When is physical memory allocated?
malloc and \textit{brk} / \textit{mmap}
Reclamation

• When can dynamically-allocated memory be freed?
  ♦ Easy if a chunk is used in one place
  ♦ Hard when a chunk is shared
  ♦ Sharing is indicated by presence of pointers to the data

• Reference counting
  ♦ Keep track of the number of outstanding pointers to each chunk of memory
  ♦ When this goes to 0, free the memory
Advanced Functionality

- Now we’re going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on write
Sharing

• Private virtual address spaces protect applications from each other
  ♦ Usually exactly what we want
• But this makes it difficult to share data (have to copy)
  ♦ Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
• We can use shared memory to allow processes to share data using direct memory references
  ♦ Both processes see updates to the shared memory segment
    » Process B can immediately read an update by process A
  ♦ How are we going to coordinate access to shared data?
Sharing (2)

- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection bits
  - Must update both PTEs when page becomes invalid

- How to destroy a virtual address space without affecting the other address space that shares data with it?
  - Reference count

- How to swap out/in a shared page?
  - Link all PTEs
  - Operation on all entries
Isolation: No Sharing

Virtual Address Space #1

Physical Memory

Virtual Address Space #2
Sharing Pages

PTEs Point to Same Physical Page
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use copy-on-write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
- How does this help fork()?
Copy on Write: Before Fork

- Parent Virtual Address Space
- Physical Memory
Copy on Write: Fork

Parent Virtual Address Space

Physical Memory

Child Virtual Address Space

Read-Only Mappings

Read-Only Mappings
Copy on Write: On A Write

Now Read-Write & Private

Physical Memory

Parent Virtual Address Space

Child Virtual Address Space
Final lecture on memory management:

• Goals of memory management
  ♦ To provide a convenient abstraction for programming
  ♦ To allocate scarce memory resources among competing processes to maximize performance with minimal overhead

• Mechanisms
  ♦ Physical and virtual addressing
  ♦ Techniques: Partitioning, paging, segmentation
  ♦ Page table management, TLBs
  ♦ Allocation, sharing

• Policies
  ♦ Page replacement algorithms
Next time...

- CPU Scheduling (will be in final)
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open, read into buffer, operate on buffer, …”
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address $base + N$ refers to offset $N$ in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    » Another use of the dirty bit in PTE
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);

See NOTES for information on feature test macro requirements.

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 (which must be greater than 0).

If addr is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If addr is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see MAP_ANONYMOUS below), are initialized using Length bytes starting at offset offset in the file (or other object) referred to by the
MapViewOfFile function

Maps a view of a file mapping into the address space of a calling process.

To specify a suggested base address for the view, use the MapViewOfFileEx function. However, this practice is not recommended.

Syntax

```cpp
LPVOID WINAPI MapViewOfFile(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap
);
```
Virtual Address Space

- Pages of file mapped one-to-one and contiguous into virtual pages in the address space

File on Disk
Mapped Files

- Pages do not have to be contiguous in physical memory
- Not all pages have to be in physical memory at once
Mapped Files (2)

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  - Virtual address space not backed by “real” files also called Anonymous VM

- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying

- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)
One last thing...

- Not in exam, but nice to know and very interesting
Kernel Address Space

• Wait…how does the OS virtual address space work?
• We have talked about it as a separate address space
• But it is typically implemented as an extension of the user-level process address space
  ♦ The bottom portion is for the user-level process
  ♦ The top portion is for the operating system/kernel
  ♦ VMS, early Unix: user 2GB, kernel 2GB (32-bit)
  ♦ Linux, Windows: user 3GB, kernel 1GB (32-bit)
Process Address Space

Address space used by process

Stack

Heap

Static Data (Data Segment)

Code (Text Segment)
Kernel Address Space

Address space used by process

Address space used by kernel

Trap to kernel

Same in all page tables
Kernel Address Space

• When CPU is in user mode, a process can only access the user-level portion
• When CPU is in kernel/privileged mode, the OS can access the entire region
• This arrangement is very convenient for the OS
  ♦ The OS can access any memory in the user-level portion of the current process (e.g., copying system call arguments)
  ♦ But the OS region is protected from the process
• As a result, the OS is mapped into every process
  ♦ The upper portion of every process address space is the OS
  ♦ Context switching effectively just switches the bottom portion
• This works well until Meltdown (mitigation: kernel page-table isolation KPTI) read more: here (Meltdown and KPTI not in exam)