CSE 120
Principles of Operating Systems

Fall 2021

Lecture 12: Swapping, Memory Allocation, Page Replacement

Yiying Zhang
Administrivia

- No class this Thursday
- Project 2 due date extended to next Friday
- Work on project 2!
Translation Lookaside Buffer (TLB)

Virtual address

VPN | offset

VPN | PFN | ...
VPN | PFN | ...

TLB

Miss

Real page table

VPN | PFN | ...

Hit

PFN | offset

Physical address
Hardware vs. software controlled

**Hardware approach**
- Efficient – TLB misses handled by hardware
- OS intervention is required only in case of page fault
- Page structure prescribed by MMU hardware -- rigid

**Software approach**
- Less efficient -- TLB misses are handled by software
- MMU hardware very simple, permitting larger, faster TLB
- OS designer has complete flexibility in choice of MM data structure
Virtual Memory Implementation

- Virtual memory is typically implemented via demand paging.
- **Demand paging:**
  - Load memory pages (from storage or initially allocated) “on demand”
  - Paging with swapping, e.g., physical pages are swapped in and out of memory
What happens at virtual memory allocation time and access time?

• What happens at virtual memory allocation time?
  ♦ If demand paging (on-demand allocation) is used, the OS allocates a virtual address (more later today) and establishes a PTE with no PFN and with invalid bit set

• What happens when the virtual address is first accessed?
  ♦ The OS should allocate physical memory for it
  ♦ How to capture the first write to a virtual page?
    » e.g. want to trap into page fault handler
      ▪ Use valid bit
  ♦ In page fault handler handler, check if the virtual page is allocated (and access permitted)
    » If not, segmentation fault
    » Else allocate physical page and update PTE
What happens when main memory is not big enough?

• Processes running on a machine have collectively used more memory than what the physical main memory has.

• Some memory pages need to be put to storage (swap out)

• What happens when a swapped out page is accessed?
  ♦ Need to swap in the page
  ♦ How to detect that a swapped out page is accessed?
Page Fault Handling in demand paging

1. MMU (TLB)
2. Page fault
3. Swap out a victim page to disk
4. Update PTE of victim pg, flush TLB
5. Swap in access page from disk
6. Update PTE of access pg
7. 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 next 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
  - Demand paging
- Now let’s put it together, bottom to top.
The Common Case

- The compiler compiles source code into binaries (containing memory instructions)
- OS loads the executable (a.out) into memory and starts its execution
- Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it, virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning the physical page frame and protection bits for this address
  3. TLB validates that the protection bits allows reads (in this example)
  4. MMU combines the PFN and offset into a physical address
  5. MMU then reads from that physical address, returns value to CPU
- Note: The above execution is all done by the hardware
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:
  1. MMU loads PTE from page table in memory (a page table walk)
     » Hardware managed TLB, OS not involved in this step
  2. Trap to the OS
     » Software managed TLB, OS intervenes at this point

♦ A machine will only support one method or the other (all modern computers have hardware-managed TLB)

• 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

- TLB 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)
  - 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
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
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}

Architecture:

- Application
  - Allocator (libc)
    - malloc()
    - free()
    - realloc()
    - calloc()
  - brk()
  - mmap()
  - munmap()

- Virtual Memory
  - Heap
    - brk()
  - Mappings
- Physical Memory
  - MMU
  - page fault
  - lookup

Process Address Space
Memory Management

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

• **Policies**
  - Page replacement algorithms
Next time...

- Move on to storage systems
- Read Chapters 37, 39
Backup Slides
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 file descriptor fd.
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**

```c++
LPVOID WINAPI MapViewOfFile(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap
);
```
Mapped Files

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

Virtual Address Space #1

Physical Memory

Virtual Address Space #2

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

Read-Only Mappings

Child Virtual Address Space
Copy on Write: On A Write

Parent Virtual Address Space

Physical Memory

Now Read-Write & Private

Child Virtual Address Space
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)

User Address Space (3GB)

0x00000000

0xBFFFFFFF
Kernel Address Space

Address space used by process

Stack

Heap

Static Data (Data Segment)

Code (Text Segment)

Address space used by kernel

Stack

Heap

Static Data (Data Segment)

Code (Text Segment)

OS Code, Data, Heap, Drivers, ...

User Address Space (3GB)

User Address Space (3GB)

Kernel Address Space (1GB)

Same in all page tables

Trap to kernel

Kernel Address Space

CSE 120 – Lecture 10 – Paging
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)
Belady's anomaly states that it is possible to have more page faults when increasing the number of page frames while using FIFO method of frame management. Laszlo Belady demonstrated this in 1970. Previously, it was believed that an increase in the number of page frames would always provide the same number or fewer page faults.
Example

Page Requests
321032432104
**Example (Page Faults in Red)**

<table>
<thead>
<tr>
<th>Page Requests – 3 frames</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Frame 2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Frame 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example (Page Faults in Red)

- Page Requests – 4 frames
- Frame 1
- Frame 2
- Frame 3
- Frame 4
Ideal curve of # of page faults v.s. # of physical pages
FIFO illustrating Belady’s anomaly
Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    » So that it faults less
    » But not always (FIFO, Belady’s Anomaly)
  - If the fault rate is below a low threshold, take away memory
    » Should fault more
    » But not always

- Hard for PFF to distinguish between changes in locality and changes in size of working set
Key observation

- Locality in memory references
  - Spatial and temporal

- Want to keep a set of pages in memory that would avoid a lot of page faults
  - "Hot" pages

- Can we formalize it?
Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 60s
- Definition
  - \( WS(t,w) = \{\text{all the pages that were referenced in the time interval } (t, t-w)\} \)
  - \( t \): time, \( w \): working set window (measured in page refs)
- A page is in the working set (WS) only if it was referenced in the last \( w \) references
Working Set Size

- Working set size is the number of unique pages in the WS
  - The number of pages referenced in the interval (t, t-w)
- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger
- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter w that determines a working set with few faults
  - Denning: Don’t run a process unless working set is in memory
Working Sets in the Real World

Working set size

transition, stable
Working Set Problems

• Problems
  ♦ How do we determine $w$?
  ♦ How do we know when the working set changes?

• Too hard to answer
  ♦ So, working set is not used in practice as a page replacement algorithm

• However, it is still used as an abstraction
  ♦ The intuition is still valid
  ♦ When people ask, “How much memory does Firefox need?”, they are in effect asking for the size of Firefox’s working set
The BIG picture

- We’ve talked about single evictions
- Most computers are multiprogrammed
  - Single eviction decision still needed
  - New concern – processes compete for resources
  - How to be “fair enough” and achieve good overall throughput
Possible replacement strategies

- Global replacement:
  - All pages from all processes are lumped into a single replacement pool
  - Most flexibility, least (performance) isolation

- Local replacement
  - Per-process replacement:
    » Each process has a separate pool of pages
  - Per-user replacement:
    » Lump all processes for a given user into a single pool

- In local replacement, must have a mechanism for (slowly) changing the allocations to each pool
Improving CPU utilization in multiprogramming

• In multiprogramming, when OS sees the CPU utilization is low,
  ♦ It thinks most processes are waiting for I/O
  ♦ it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  ♦ It adds/loads another process to the system
When there are not enough page frames

- Suppose many processes are making frequent references to 50 pages, memory has 49.
- Assuming LRU:
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out.
- Btw, what is the optimal strategy here?
  - MRU
- What is the average memory access time?
- The system is spending most of its time paging!
- The progress of programs makes it look like “memory access is as slow as disk”, rather than “disk being as fast as memory”
Thrashing

- Thrashing
  - When most of the time is spent by the OS in paging data back and forth from disk
  - Little time spent doing useful work (making progress)
  - In this situation, the system is overcommitted
Thrashing can lead to vicious cycle

• If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  ♦ low CPU utilization
  ♦ OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  ♦ another process added to the system
  ♦ page fault rate goes even higher
Thrashing (Cont.)

![Graph showing CPU utilization vs. degree of multiprogramming. The graph indicates a pattern of thrashing with increased CPU utilization as the degree of multiprogramming increases, peaking and then decreasing.](image-url)
What causes thrashing?

• The system does not know it has taken more work than it can handle

• What do humans do when thrashing?
  ♦ Dropping or degrading a course if taking too many than you can handle 😁
Intuitively, what to do about thrashing?

• If a single process’s locality too large for memory, what can OS do?
  ♦ e.g., pin most data (hotter data) in memory, sacrifice the rest

• If the problem arises from the sum of several processes?
  ♦ Figure out how much memory each process needs – “locality”
  ♦ What can we do?
    » Can limit effects of thrashing using local replacement
    » Or, bring a process’ working set before running it
    » Or, wait till there is enough memory for a process’s need