Sandeep K. S. Gupta
Arizona State University
17th lecture
-Virtual Memory(2)
(Slides adapted from CSAPP)
Announcement

- HW 4 is posted
  - Due: Fri. Nov 11, 11:59 pm
  - Assignment:
    - Reading assignments: Read chapter 8 of CSAPP book (process control and signals)
    - Programming assignment: implement your own Linux shell program
Previous class summary

- Virtual memory
  - VM as a memory management tool
  - VM address translation

- This class:
  - VM for Intel Core i7 Systems
  - Memory mapping
  - Dynamic memory allocation
Agenda

- Intel Core i7 Memory System
  - Address translation
  - Page table
- Virtual Memory of a Linux Process
  - Linux page fault handling
- Memory mapping
- Dynamic memory allocation (basic concepts)
  - The malloc Package
  - Fragmentation
  - Implementation issues
Intel Core i7 Memory System

Processor package

Core x4

- Registers
- Instruction fetch
  - L1 d-cache: 32 KB, 8-way
  - L1 i-cache: 32 KB, 8-way
  - L2 unified cache: 256 KB, 8-way
  - L1 d-TLB: 64 entries, 4-way
  - L1 i-TLB: 128 entries, 4-way
  - L2 unified TLB: 512 entries, 4-way
  - QuickPath interconnect: 4 links @ 25.6 GB/s each

- MMU (addr translation)

- DDR3 Memory controller: 3 x 64 bit @ 10.66 GB/s, 32 GB/s total (shared by all cores)

- Main memory

To other cores
To I/O bridge
Review of Symbols

- **Basic Parameters**
  - $N = 2^n$: Number of addresses in virtual address space
  - $M = 2^m$: Number of addresses in physical address space
  - $P = 2^p$: Page size (bytes)

- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number

- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN**: Physical page number
  - **CO**: Byte offset within cache line
  - **CI**: Cache index
  - **CT**: Cache tag
End-to-end Core i7 Address Translation

Virtual address (VA)

CPU

VPN

VPO

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

CR3

Page tables

PTE

PTE

PTE

PTE

L1 d-cache

(64 sets, 8 lines/set)

L1 hit

L1 miss

Physical address (PA)

32/64

Result

L2, L3, and main memory

40

12

CT

CI

CO

L1 d-cache

Page tables

CPU

VPN

VPO

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

CR3

Page tables

PTE

PTE

PTE

PTE

L1 d-cache

(64 sets, 8 lines/set)
Core i7 Level 1-3 Page Table Entries

| 63 | 62 | 52 | 51 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| XD | Unused | Page table physical base address | Unused | G | PS | A | CD | WT | U/S | R/W | P=1 |

Available for OS (page table location on disk)  
P=0

Each entry references a 4K child page table

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

WT: Write-through or write-back cache policy for the child page table.

CD: Caching disabled or enabled for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

G: Global page (don’t evict from TLB on task switch)

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
## Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>Page physical base address</th>
<th>Unused</th>
<th>G</th>
<th>D</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P=0</td>
<td>Available for OS (page location on disk)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Each entry references a 4K child page

- **P**: Child page is present in memory (1) or not (0)
- **R/W**: Read-only or read-write access permission for child page
- **U/S**: User or supervisor mode access
- **WT**: Write-through or write-back cache policy for this page
- **CD**: Cache disabled (1) or enabled (0)
- **A**: Reference bit (set by MMU on reads and writes, cleared by software)
- **D**: Dirty bit (set by MMU on writes, cleared by software)
- **G**: Global page (don’t evict from TLB on task switch)

**Page physical base address**: 40 most significant bits of physical page address (forces pages to be 4KB aligned)
Core i7 Page Table Translation

CR3
Physical address of L1 PT

VPN 1  VPN 2  VPN 3  VPN 4  VPO

L1 PT
Page global directory

L2 PT
Page upper directory

L3 PT
Page middle directory

L4 PT
Page table

L1 PTE
L2 PTE
L3 PTE
L4 PTE

VPN 1  VPN 2  VPN 3  VPN 4  VPO

512 GB region per entry
1 GB region per entry
2 MB region per entry
4 KB region per entry

40 40 40 40

PPN

CEN591 Fall 2011
Cute Trick for Speeding Up L1 Access

Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible
Virtual Memory of a Linux Process

- **Different for each process**
  - Process-specific data structs (ptables, task and mm structs, kernel stack)
- **Identical for each process**
  - Physical memory
  - Kernel code and data
  - User stack
  - Memory mapped region for shared libraries
  - Runtime heap (malloc)
  - Uninitialized data (.bss)
  - Initialized data (.data)
  - Program text (.text)

- **Kernel virtual memory**
- **Process virtual memory**

- **Process virtual memory**
  - Process virtual memory
  - Identical for each process
    - Physical memory
    - Kernel code and data
    - User stack
    - Memory mapped region for shared libraries
    - Runtime heap (malloc)
    - Uninitialized data (.bss)
    - Initialized data (.data)
    - Program text (.text)

- **Different for each process**
  - Process-specific data structs (ptables, task and mm structs, kernel stack)
Linux Organizes VM as Collection of “Areas”

- **pgd**: 
  - Page global directory address 
  - Points to L1 page table

- **vm_prot**: 
  - Read/write permissions for this area

- **vm_flags** 
  - Pages shared with other processes or private to this process

```
Linux Organizes VM as Collection of “Areas”

- **pgd**: 
  - Page global directory address 
  - Points to L1 page table

- **vm_prot**: 
  - Read/write permissions for this area

- **vm_flags** 
  - Pages shared with other processes or private to this process
```
Linux Page Fault Handling

**Process virtual memory**

- **shared libraries**
- **data**
- **text**

- **vm_area_struct**
  - `vm_end`
  - `vm_start`
  - `vm_prot`
  - `vm_flags`
  - `vm_next`

1. **read**
   - Segmentation fault: accessing a non-existing page

2. **write**
   - Protection exception: e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

3. **read**
   - Normal page fault

CEN591 Fall 2011
Memory Mapping

- VM areas initialized by associating them with disk objects.
  - Process is known as memory mapping.

- Area can be backed by (i.e., get its initial values from):
  - **Regular file** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - **Anonymous file** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (**demand-zero page**)
    - Once the page is written to (**dirtied**), it is like any other page

- Dirty pages are copied back and forth between memory and a special **swap file**.
Demand paging

- **Key point:** no virtual pages are copied into physical memory until they are referenced!
  - Known as *demand paging*

- Crucial for time and space efficiency
Sharing Revisited: Shared Objects

- Process 1 maps the shared object.
Sharing Revisited: Shared Objects

- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.
Sharing Revisited: Private Copy-on-write (COW) Objects

- Two processes mapping a private copy-on-write (COW) object.
- Area flagged as private copy-on-write.
- PTEs in private areas are flagged as read-only.

Diagram:
- Process 1 virtual memory
- Physical memory
- Process 2 virtual memory
- Private copy-on-write object
- Private copy-on-write area
Sharing Revisited: Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!
The `fork` Function Revisited

- VM and memory mapping explain how `fork` provides private address space for each process.

- To create virtual address for new new process
  - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW

- On return, each process has exact copy of virtual memory

- Subsequent writes create new pages using COW mechanism.
The `execve` Function Revisited

- To load and run a new program `a.out` in the current process using `execve`:
  - Free `vm_area_struct`'s and page tables for old areas
  - Create `vm_area_struct`'s and page tables for new areas
    - Programs and initialized data backed by object files.
    - `.bss` and stack backed by anonymous files.
  - Set PC to entry point in `.text`
    - Linux will fault in code and data pages as needed.
User-Level Memory Mapping

```c
void *mmap(void *start, int len,
            int prot, int flags, int fd, int offset)
```

- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
  - `start`: may be 0 for “pick an address”
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be `start`)
User-Level Memory Mapping

```c
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

Process virtual memory

Disk file specified by file descriptor fd

len bytes

(start (or address chosen by kernel))

len bytes

offset (bytes)
Using `mmap` to Copy Files

- Copying without transferring data to user space.

```c
#include "csapp.h"

/*
* mmapcopy - uses mmap to copy
* file fd to stdout
*/
void mmapcopy(int fd, int size)
{
    /* Ptr to mem-mapped VM area */
    char *bufp;

    bufp = Mmap(NULL, size,
                 PROT_READ,
                 MAP_PRIVATE, fd, 0);
    Write(1, bufp, size);
    return;
}

/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmdline arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }

    /* Copy the input arg to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

Copying without transferring data to user space.
Dynamic Memory Allocation: Basic Concepts
Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.

- For data structures whose size is only known at runtime.

Dynamic memory allocators manage an area of process virtual memory known as the *heap*. 

![Dynamic Memory Allocation Diagram]

- **Application**
- **Dynamic Memory Allocator**
- **Heap**
- **User stack**
- **Heap (via malloc)**
- **Uninitialized data (.bss)**
- **Initialized data (.data)**
- **Program text (.text)**

Top of heap (`brk` ptr)
Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized blocks, which are either allocated or free

- Types of allocators
  - Explicit allocator: application allocates and frees space
    - E.g., malloc and free in C
  - Implicit allocator: application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp

- Will discuss simple explicit memory allocation today
# The malloc Package

```c
#include <stdlib.h>

void *malloc(size_t size)

- **Successful:**
  - Returns a pointer to a memory block of at least \texttt{size} bytes (typically) aligned to 8-byte boundary
  - If \texttt{size} == 0, returns NULL
- **Unsuccessful:** returns NULL (0) and sets \texttt{errno}

void free(void *p)

- Returns the block pointed at by \texttt{p} to pool of available memory
- \texttt{p} must come from a previous call to \texttt{malloc} or \texttt{realloc}

Other functions

- \texttt{calloc}: Version of \texttt{malloc} that initializes allocated block to zero.
- \texttt{realloc}: Changes the size of a previously allocated block.
- \texttt{sbrk}: Used internally by allocators to grow or shrink the heap
```
void foo(int n, int m) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;

    /* Return p to the heap */
    free(p);
}
Assumptions Made in This Lecture

- Memory is word addressed (each word can hold a pointer)
Allocation Example

\[
\begin{align*}
p_1 &= \text{malloc}(4) \\
p_2 &= \text{malloc}(5) \\
p_3 &= \text{malloc}(6) \\
\text{free}(p_2) \\
p_4 &= \text{malloc}(2)
\end{align*}
\]
Constraints

- **Applications**
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - *free* request must be to a `malloc`'d block

- **Allocators**
  - Can’t control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
    - *i.e.*, can’t reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8 byte alignment for GNU `malloc (libc malloc)` on Linux boxes
  - Can manipulate and modify only free memory
  - Can’t move the allocated blocks once they are `malloc`'d
    - *i.e.*, compaction is not allowed
Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - \( R_0, R_1, \ldots, R_k, \ldots, R_{n-1} \)

- Goals: maximize throughput and peak memory utilization
  - These goals are often conflicting

- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second
Performance Goal: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - \( R_0, R_1, \ldots, R_k, \ldots, R_{n-1} \)

- **Def:** Aggregate payload \( P_k \)
  - `malloc(p)` results in a block with a **payload** of \( p \) bytes
  - After request \( R_k \) has completed, the **aggregate payload** \( P_k \) is the sum of currently allocated payloads

- **Def:** Current heap size \( H_k \)
  - Assume \( H_k \) is monotonically nondecreasing
    - i.e., heap only grows when allocator uses `sbrk`

- **Def:** Peak memory utilization after \( k \) requests
  - \( U_k = (\max_{i<k} P_i) / H_k \)
Fragmentation

- Poor memory utilization caused by fragmentation
  - *internal* fragmentation
  - *external* fragmentation
Internal Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than block size.

- Caused by:
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions (e.g., to return a big block to satisfy a small request)

- Depends only on the pattern of *previous* requests
  - Thus, easy to measure
External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough.

- Depends on the pattern of future requests.

```
    p1 = malloc(4)
    p2 = malloc(5)
    p3 = malloc(6)
    free(p2)
    p4 = malloc(6)
```

`Oops! (what would happen now?)`
Implementation Issues

- How do we know how much memory to free given just a pointer?

- How do we keep track of the free blocks?

- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?

- How do we pick a block to use for allocation -- many might fit?

- How do we reinsert freed block?
Knowing How Much to Free

- **Standard method**
  - Keep the length of a block in the word preceding the block.
    - This word is often called the *header field* or *header*
  - Requires an extra word for every allocated block

```
p0 = malloc(4)
```

```
free(p0)
```
Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

```
```

- Method 2: *Explicit list* among the free blocks using pointers

```
```

- Method 3: *Segregated free list*
  - Different free lists for different size classes

- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key