CSC 252: Computer Organization
Spring 2018: Lecture 23

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:
• Programming Assignment 6 is out
Announcement

• Programming Assignment 6 is out
  • Trivia: 12:00pm, **Thursday, April 19**.
  • Main assignment: 11:59pm, **Monday, April 30**.
• Thursday office hours moved to today.

### Calendar

- **17th April**: Trivia
- **30th April**: Due
Today

• Case study: Core i7/Linux memory system
• Memory mapping
• Dynamic memory allocation
Intel Core i7 Memory System

Processor package

Core x4

Registers

L1 d-cache
32 KB, 8-way

L1 i-cache
32 KB, 8-way

L2 unified cache
256 KB, 8-way

Instruction fetch

L1 d-TLB
64 entries, 4-way

L1 i-TLB
128 entries, 4-way

MMU
(address translation)

L2 unified TLB
512 entries, 4-way

QuickPath interconnect
4 links @ 25.6 GB/s each

L3 unified cache
8 MB, 16-way
(shared by all cores)

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
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

36

12

VPN VPO
End-to-End Core i7 Address Translation

```
CPU

Virtual address (VA)

36 12

VPN VPO

32 4

TLBT TLBI
```
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN VPO

TLBT TLBI

L1 TLB (16 sets, 4 entries/set)
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

36

12

VPN

VPO

32

4

TLBT

TLBI

TLB
hit

L1 TLB (16 sets, 4 entries/set)

40

PPN
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

PPN

CR3

PTE

Page tables

VPN

VPO

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)
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

PTE

Page tables

PPN

PPO

Page tables
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN

VPO

TLBT

TLBI

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

PPN

PPO

Physical address (PA)
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN VPO

TLB

TLBT TLBI

TLB hit

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

CR3

PTE PTE PTE PTE

Page tables

PPN PPO

CT CI CO

Physical address (PA)
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

PTE

Page tables

CR3

L1 d-cache

(64 sets, 8 lines/set)

PPN

PPO

Physical address (PA)

CT

CI

CO
End-to-End Core i7 Address Translation

CPU

Virtual address (VA)

VPN
VPO

TLBT
TLBI

TLB hit

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1
VPN2
VPN3
VPN4

9
9
9
9

PTE

Page tables

CR3

L1 hit

L1 d-cache
(64 sets, 8 lines/set)

32/64

Result

PPN
PPO

L1

Physical address (PA)

CT
Cl
CO

32/64

36
12

32
4

PTE

Page tables
End-to-End Core i7 Address Translation

Virtual address (VA)

L1 TLB hit

TLB hit

L1 d-cache (64 sets, 8 lines/set)

L1 miss

CPU

VPN

VPO

TLBT

TLBI

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

VPN

VPO

TLBT

TLBI

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

CPU

VPN

VPO

TLBT

TLBI

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

CPU

VPN

VPO

TLBT

TLBI

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables

CPU

VPN

VPO

TLBT

TLBI

VPN1

VPN2

VPN3

VPN4

CR3

PTE

Page tables
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

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

40
40
40
40

VPN 1
VPN 2
VPN 3
VPN 4
VPO

PPN
PPO

Virtual address
Offset into physical and virtual page
Physical address

Virtual Address Space of a Linux Process

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

**Process virtual memory**

- **Kernel virtual memory**

- **Process-­‐specific data structs (ptables, task and mm structs, kernel stack)**

- **Physical memory**

- **Kernel code and data**

- **Memory mapped region for shared libraries**

- **User stack**

- **Runtime heap (malloc)**

- **Uninitialized data (.bss)**

- **Initialized data (.data)**

- **Program text (.text)**

- **Process virtual memory**

- **Kernel virtual memory**
Linux Organizes VM as Collection of “Areas”

- Areas are also known as “segments”
- pgd:
  - Page global directory address (L1 PT)
- 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

vm_area_struct

- vm_end
- vm_start
- vm_prot
- vm_flags

vm_next

vm_area_struct

- vm_end
- vm_start
- vm_prot
- vm_flags

vm_next

vm_area_struct

- vm_end
- vm_start
- vm_prot
- vm_flags
- vm_next
Linux Page Fault Handling

- `vm_area_struct`
  - `vm_end`
  - `vm_start`
  - `vm_prot`
  - `vm_flags`

Process virtual memory

- `shared libraries`
- `data`
- `text`
Linux Page Fault Handling

Segmentation fault: accessing a non-existing page
Linux Page Fault Handling

Segmentation fault: accessing a non-existing page
Linux Page Fault Handling

1. Segmentation fault: accessing a non-existing page

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

Segmentation fault: accessing a non-existing page

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

Segmentation fault: accessing a non-existing page

Normal page fault

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

- Case study: Core i7/Linux memory system
- Memory mapping
- Dynamic memory allocation
Memory Mapping

• Memory mapping is the mechanism that Linux uses to initialize VM areas (standard across modern OSes)
Memory Mapping

• Memory mapping is the mechanism that Linux uses to initialize VM areas (standard across modern OSes)
• VM area initial data can come from:
  • Some file on disk (e.g., an compiled executable)
  • Initialized to 0
Memory Mapping

• Memory mapping is the mechanism that Linux uses to initialize VM areas (standard across modern OSes)
• VM area initial data can come from:
  • Some file on disk (e.g., an compiled executable)
  • Initialized to 0
• VM 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 (i.e., nothing)
    • First fault will allocate a physical page full of 0's (demand-zero page)
Memory Mapping

- Memory mapping is the mechanism that Linux uses to initialize VM areas (standard across modern OSes)

- VM area initial data can come from:
  - Some file on disk (e.g., an compiled executable)
    - Initialized to 0
  - VM 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 (i.e., nothing)
      - First fault will allocate a physical page full of 0's (demand-zero page)

- Dirty pages are copied back and forth between memory and a special swap file.
Memory Mapping

• Memory mapping is the mechanism that Linux uses to initialize VM areas (standard across modern OSes)

• VM area initial data can come from:
  • Some file on disk (e.g., an compiled executable)
  • Initialized to 0

• VM 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 (i.e., nothing)
    • First fault will allocate a physical page full of 0's (demand-zero page)

• Dirty pages are copied back and forth between memory and a special swap file.

• Memory map provides a clean interface to initialize VM data
Memory Mapping For Sharing

- Multiple processes often share data
  - Different processes that run the same code (e.g., shell)
  - Different processes linked to the same standard libraries
  - Different processes share the same file
- It is wasteful to create exact copies of the share object
- Memory mapping allow us to easily share objects
  - Different VM pages point to the same physical page/object
Sharing Revisited: Shared Objects

- Process 1 maps the shared object.

- The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.
Sharing Revisited: Shared Objects

• Process 2 maps the shared object.

• The kernel remembers that the object (backed by a unique file) is mapped by Proc. 1 to some physical pages.

• Now when Proc. 2 wants to access the same object, the kernel can simply point the PTEs of Proc. 2 to the already-mapped physical pages.
The Problem...

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2
The Problem...

- What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2
- Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.
The Problem…

• What if Proc. 1 now wants to modify the shared object, but doesn’t want the modification to be visible to Proc. 2
• Simplest solution: always create duplicate copies of shared objects at the cost of wasting space. Not ideal.
• Idea: Copy-on-write (COW)
  • First pretend that both processes will share the objects without modifying them. If modification happens, create separate copies.
Private Copy-on-write (COW) Objects

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

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

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

[Diagram showing the structure of process virtual memory with task_struct, mm_struct, vm_area_struct, and various fields like mm, pgd, mmap, vm_end, vm_start, vm_prot, vm_flags, vm_next, shared libraries, data, and text sections.]
Private Copy-on-write (COW) Objects

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

- Instruction writing to private page triggers page (protection) fault.

Diagram:
- Process 1 virtual memory
- Physical memory
- Process 2 virtual memory
- Copy-on-write
- Private copy-on-write object
- Write to private COW page
Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it’s a COW object.
Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it’s a COW object.
- Handler then creates new R/W page.
Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it’s a COW object.
- Handler then creates new R/W page.
Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it’s a COW object
- Handler then creates new R/W page.
- Instruction restarts upon handler return.
Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers page (protection) fault.
- Handler checks the area protection, and sees that it’s a COW object.
- Handler then 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 (in the PTE) of each page in both processes as read-only
  - Flag each area in both processes as private COW (in `vm_area_struct`)

- 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("a.out", NULL, NULL)`
- Free `vm_area_struct` and page tables for old process
- Create `vm_area_struct` 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

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

void *mmap(void *start, int len, int prot, int flags, int fd, int offset)
Example: Using \texttt{mmap} to Copy Files

- Copying a file to stdout without transferring data to user space
  - i.e., no file data is copied to user stack

```c
#include "csapp.h"

void mmapcopy(int fd, int size) {
    /* Ptr to memory mapped 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 cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }

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

• Case study: Core i7/Linux memory system
• Memory mapping
• Dynamic memory allocation
  • Basic concepts
  • Implicit free lists
Dynamic Memory Allocation

• Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.

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

![Diagram of memory regions]

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

Top of heap (brk ptr)
# The `malloc/free` Functions

```c
#include <stdlib.h>
void *malloc(size_t size)
```

- **Successful:**
  - Returns a pointer to a memory block of at least `size` bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If `size == 0`, returns NULL
- **Unsuccessful:** returns NULL (0) and sets `errno`

```c
void *malloc(size_t size)
```
The `malloc/free` Functions

```c
#include <stdlib.h>

void *malloc(size_t size)
    • Successful:
        • Returns a pointer to a memory block of at least `size` bytes
          aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
        • If `size == 0`, returns NULL
    • Unsuccessful: returns NULL (0) and sets `errno`

void free(void *p)
    • Returns the block pointed at by `p` to pool of available memory
    • `p` must come from a previous call to `malloc` or `realloc`
```
The `malloc/free` Functions

```c
#include <stdlib.h>

void *malloc(size_t size)

• Successful:
  • Returns a pointer to a memory block of at least `size` bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  • If `size == 0`, returns NULL
• Unsuccessful: returns NULL (0) and sets `errno`

void free(void *p)

• Returns the block pointed at by `p` to pool of available memory
• `p` must come from a previous call to `malloc` or `realloc`

Other functions
• `calloc`: Version of `malloc` that initializes allocated block to zero.
• `realloc`: Changes the size of a previously allocated block.
• `sbrk`: Used internally by allocators to grow or shrink the heap
```
malloc Example

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
    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 allocated block to the heap */
    free(p);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.

• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.

• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```c
int* foo(int n) {
    int i, *p;
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) exit(0);
    for (i=0; i<n; i++)
        p[i] = i;
    return p;
}

void bar() {
    int *p = foo(5);
    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```c
int* foo(int n) {
    int i, *p;
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) exit(0);
    for (i=0; i<n; i++)
        p[i] = i;
    return p;
}

void bar() {
    int *p = foo(5);
    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!

```c
int* foo(int n) {
    int i, *p;
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) exit(0);
    for (i=0; i<n; i++)
        p[i] = i;
    return p;
}

void bar() {
    int *p = foo(5);
    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

- Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
- More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```c
int* foo(int n) {
    int i, *p;
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) exit(0);
    for (i=0; i<n; i++)
        p[i] = i;
    return p;
}

void bar() {
    int *p = foo(5);
    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```c
int* foo(int n) {
    int i, *p;
    p = (int*) malloc(n * sizeof(int));
    if (p == NULL) exit(0);
    for (i=0; i<n; i++)
        p[i] = i;
    return p;
}
void bar() {
    int *p = foo(5);
    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```c
int* foo() {
    int i;
    int p[5];

    for (i=0; i<5; i++)
        p[i] = i;

    return p;
}

void bar() {
    int *p = foo();

    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!

```c
int* foo() {
    int i;
    int p[5];
    for (i=0; i<5; i++)
        p[i] = i;
    return p;
}

void bar() {
    int *p = foo();
    printf("%d\n", p[0]);
}
```

Diagram:

```
bar Stack
   p

foo Stack
   p i

Heap (via malloc)
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!

```c
int* foo() {
    int i;
    int p[5];

    for (i=0; i<5; i++)
        p[i] = i;

    return p;
}

void bar() {
    int *p = foo();

    printf("%d\n", p[0]);
}
```
Why Do We Need Dynamic Allocation?

• Some data structures’ size is only known at runtime. Statically allocating the space would be a waste.
• More importantly: access data across function calls. Variables on stack are destroyed when the function returns!!!
Dynamic Memory Allocation

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

• Blocks that are no longer used should be free-ed to save space

• Assumptions Made in This Lecture
  • Memory is word addressed
  • Words are int-sized
Dynamic Memory Allocation

• Types of allocators
  • *Explicit allocator*: application (i.e., programmer) 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, JavaScript, Python, etc...

• Will discuss simple explicit memory allocation today
Allocation Example

\[ p_1 = \text{malloc}(4) \]

\[ p_2 = \text{malloc}(5) \]

\[ p_3 = \text{malloc}(6) \]

\[ \text{free}(p_2) \]

\[ p_4 = \text{malloc}(2) \]
Allocation Example

\begin{itemize}
  \item p1 = malloc(4)
  \item p2 = malloc(5)
  \item p3 = malloc(6)
  \item free(p2)
  \item p4 = malloc(2)
\end{itemize}
Allocation Example

\[ p_1 = \text{malloc}(4) \]

\[ p_2 = \text{malloc}(5) \]

\[ p_3 = \text{malloc}(6) \]

\[ \text{free}(p_2) \]

\[ p_4 = \text{malloc}(2) \]
Allocation Example

p1 = malloc(4)  

p2 = malloc(5)  

p3 = malloc(6)  

free(p2)  

p4 = malloc(2)
Allocation Example

\begin{itemize}
  \item \texttt{p1 = malloc(4)}
  \item \texttt{p2 = malloc(5)}
  \item \texttt{p3 = malloc(6)}
  \item \texttt{free(p2)}
  \item \texttt{p4 = malloc(2)}
\end{itemize}
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 place allocated blocks only in free memory
  • Must align blocks so they satisfy all alignment requirements
    • 8-byte (x86) or 16-byte (x86-64) alignment 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
External Fragmentation

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

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

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

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

Oops! (what would happen now?)
External Fragmentation

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

```c
p1 = malloc(4)  
```

```c
p2 = malloc(5)  
```

```c
p3 = malloc(6)  
```

```c
free(p2)  
```

```c
p4 = malloc(6)  
```

*Oops! (what would happen now?)*

• Depends on the pattern of future requests
Key Issues in Dynamic Memory Allocation

• Free:
  • How do we know how much memory to free given just a pointer?
  • How do we keep track of the free blocks?
  • How do we reinsert freed block?

• Allocation:
  • 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?
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

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

```c
free(p0)
```
Internal Fragmentation

• For a given block, *internal fragmentation* occurs if payload is smaller than block size
Internal Fragmentation

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

![Diagram of block with internal fragmentation]

• 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)
Keeping Track of Free Blocks

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

![Diagram of free blocks with implicit list]
Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks
  
  ![Diagram of implicit list]

- Method 2: *Explicit list* among the free blocks using pointers
  
  ![Diagram of explicit list]
Keeping Track of Free Blocks

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

```
  5   4   6   2
```

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

```
  5   4   6   2
```

• Method 3: *Segregated free list*
  • Different free lists for different size classes
Keeping Track of Free Blocks

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

![ Implicit list diagram ]

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

![ Explicit list diagram ]

• 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
Today

• Case study: Core i7/Linux memory system
• Memory mapping
• Dynamic memory allocation
  • Basic concepts
  • Implicit free lists
Implicit List

• For each block we need both size and allocation status
  • Could store this information in two words: wasteful!
Implicit List

• For each block we need both size and allocation status
  • Could store this information in two words: wasteful!

```
Size: block size
Payload: application data (allocated blocks only)
```

```
a = 1: Allocated block
a = 0: Free block
```

```
5  4   6  2
```

```
Size   a   Payload   Optional padding
```
Implicit List

• For each block we need both size and allocation status
  • Could store this information in two words: wasteful!
• Standard trick
  • If blocks are aligned, some low-order address bits are always 0
  • Instead of storing an always-0 bit, use it as a allocated/free flag
  • When reading size word, must mask out this bit
Implicit List

• For each block we need both size and allocation status
  • Could store this information in two words: wasteful!

• Standard trick
  • If blocks are aligned, some low-order address bits are always 0
  • Instead of storing an always-0 bit, use it as a allocated/free flag
  • When reading size word, must mask out this bit

Format of allocated and free blocks

<table>
<thead>
<tr>
<th>Size</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Optional padding</td>
<td></td>
</tr>
</tbody>
</table>

- a = 1: Allocated block
- a = 0: Free block
- Size: block size
- Payload: application data (allocated blocks only)