Introduction

Course Schedule
- Lecture: Monday 4:15 – 5:00 p.m. (RZ F21)
- Recitation: TBD
- Assignments: in teams, time to be arranged individually

Contact
- Stefan Freudenberger, RZ H6, stefan.freudenberger@inf.ethz.ch
- TBD

Resources
Structure of the Course

- **Projects**
  - Topics related to operating systems
  - Practical focus

- **Organization**
  - Programming assignments (labs)
  - Teams (max 2 students) are allowed
  - Final grade based on the labs

- **Environment**
  - Unix/C
# Course schedule (tentative)

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-02-22</td>
<td>Memory Allocation</td>
<td></td>
</tr>
<tr>
<td>2010-03-01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-03-08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-03-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-03-22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-03-29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-04-05</td>
<td>Easter Monday: holiday</td>
<td></td>
</tr>
<tr>
<td>2010-04-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-04-19</td>
<td>Sechseläuten: holiday</td>
<td></td>
</tr>
<tr>
<td>2010-04-26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-05-03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-05-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-05-17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2010-05-24</td>
<td>Whit Monday: holiday</td>
<td></td>
</tr>
<tr>
<td>2010-05-31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suggested Literature

- R. E. Bryant and D. O’Hallaron
  *Computer Systems — A Programmer’s Perspective*
Project 1: Memory Allocation

- Summary
  - Implement a memory allocator: malloc, free, realloc

- Goals
  - Learn how to manage storage allocation
  - Good exercise to master pointers

- Environment
  - Unix like system
  - C

- Resources
The Heap

Area of a process’s (virtual) memory that is dynamically allocated

- Example (Unix)
  - Grows upwards
  - The\(\text{brk}\) pointer marks the top of the heap
Memory Management

The memory manager (user space) maintains a process’s heap: a collection of blocks (contiguous chunks of memory) which are either allocated or free

- Allocation is explicit (e.g., malloc in C or new in Java)
-Deallocation can be:
  - explicit (e.g., free in C, delete in C++)
  - implicit (garbage collection in Java, .NET)

Issues

- The memory manager needs to:
  - Distinguish block boundaries
  - Distinguish between free and allocated blocks
Memory Allocation: Strategies

- **first-fit**: use the first free block which is big enough
- **best-fit**: take smallest fitting block
- **worst-fit**: take biggest available block
- **quick-fit**: best-fit but multiple free-lists (one per block size)
Memory Allocation: Bitmaps

The simplest (but often slowest) way to manage memory blocks is to mark in a bitmap if a block is free or allocated

- **Space**
  - For a 4GB memory space and 4KB words a 128MB table is necessary to store the free/allocated information

- **Speed**
  - Slow scan to find a free block
Memory Allocation: Dynamic Data Structures

- Free blocks are stored in a dynamic data structure
  - For example, ordered list, ordered tree, ...
  - Lists are usually ordered by allocation address to allow a faster merging of free blocks
Memory Allocation: Dynamic Data Structures

- Free blocks are stored in a dynamic data structure
  - For example, ordered list, ordered tree, ...
  - Lists are usually ordered by allocation address to allow a faster merging of free blocks
    - When two adjacent blocks are both free, they are coalesced into a single block
      - Initially:
        ![Diagram of initial state]
      - After free:
        ![Diagram after being marked as free]
      - After coalescing:
        ![Diagram after coalescing]
Memory Allocation: Lists

- **Space**
  - Link information as well as some flags (i.e., free or allocated) has to be stored in the block

- **Speed**
  - Depending on the chosen data structure operations can be implemented efficiently
Memory Allocation: Space Utilization

- External fragmentation
  - Unused space among the blocks – depends on the allocation strategy

- Internal fragmentation
  - Unused space inside the blocks – depends on the possible block sizes (granularity)

- Data structures
  - Space used to maintain data structures cannot be allocated (bitmaps, pointers, flags, …)
Memory Allocation: Buddy System

- **An example of allocation strategy:**
  - Each block has a size of $2^k$
  - The system maintains a list for each size
- **Block allocation:**
  - Find a block which fits
  - If necessary split a larger block, and put the remaining part in the correct list
- **Free:**
  - Mark the block as free
  - If is possible merge with a free neighbor

- The buddy system is used in several operating systems (Windows, Linux, Oberon, …).
Memory Allocation: Buddy System (example)

- **Free:**
  - Initial:
    - 8
    - 8
    - 16

- **Free list:**
  - 8
  - 16
  - 32
Memory Allocation: Buddy System (example)

- **Free:**
  - Initial:
    - 8
    - 8
    - 16
  - Free:
    - 8
    - 8
    - 16

- Free list:
  - 8
  - 8
  - 16
  - 32
Memory Allocation: Buddy System (example)

- **Free:**
  - Initial:
    - 8
    - 8
    - 16
  - Free:
    - 8
    - 8
    - 16
  - Coalesce:
    - 16
    - 16

- **Free list:**
  - 8
  - 16
  - 16
  - 32
Memory Allocation: Buddy System (example)

- Free:
  - Initial:
    - 8
    - 8
    - 16
  - Free:
    - 8
    - 8
    - 16
  - Coalesce:
    - 16
    - 16
  - Coalesce again:
    - 32

- Free list:
  - 8
  - 16
  - 32
Memory Allocation: Buddy System (example)

- Allocate(8):
  - Initial:
    - Free list:
      - 32

- Free list:
  - 8
  - 16
  - 32
Memory Allocation: Buddy System (example)

- Allocate(8):
  - Initial:
    - Split free block:

  

- Free list:
  - 8
  - 16
  - 32
Memory Allocation: Buddy System (example)

- Allocate(8):
  - Initial: 32
  - Split free block: 16 16
  - Split free block again: 8 8 16

Free list:

- 8
- 16
- 32
Memory Allocation: Buddy System (example)

- Allocate(8):
  - Initial:
    - Memory allocation diagram showing a block of size 32
  - Split free block:
    - Memory allocation diagram showing blocks of sizes 16 and 16
  - Split free block again:
    - Memory allocation diagram showing blocks of sizes 8, 8, and 16
  - Allocate:
    - Memory allocation diagram showing blocks of sizes 8, 8, and 16

Free list:
- 8
- 16
- 32
Memory Allocation: Slab Allocator

- Disadvantages of the buddy system:
  - **Internal fragmentation** (max 50%)
  - **Bad distribution** of the block addresses (bad caches performance)

- Solaris introduced the concept of a **slab allocator**:
  - Based on the idea that processes are likely to request a lot of objects of the same size: these objects can be **kept** and **reused**
  - Slabs are collections of objects of the same size
  - Sizes (and addresses) are not **geometrically** distributed

- Used by AmigaOS (4), Linux (≥ 2.2) and Solaris (≥ 2.4)

Lab: Task

- Write a dynamic memory allocator, i.e., implement:
  - `malloc`
  - `free`
  - `realloc`

- Goals of the lab:
  - Understand how a real memory allocator works
  - Evaluate different design strategies
  - A nice exercise for low level C programming
Lab: Task

- **void** mm_malloc(size_t size)
  - Returns a pointer to a memory block of at least size bytes
  - 8 byte aligned
  - If no allocation is possible returns **NULL**

- **void** mm_free(void* ptr)
  - Frees the block pointed to by ptr
  - If ptr is not the first address of an allocated block the behavior is undefined
Lab: Task

- \texttt{void* mm realloc(void* ptr, size_t size)}
  - If \texttt{ptr} is \texttt{NULL}: \texttt{mm malloc(size)}
  - If \texttt{size} is 0: the allocated memory block is reduced in size to zero bytes and a non-\texttt{NULL} pointer is returned (which cannot be directly dereferenced, since it points to no allocated memory, but it can be used in future calls to \texttt{realloc} and \texttt{free})(this is the c99 definition)
  - Returns a pointer to a new block of \texttt{size} bytes and with the same content as the block pointed by \texttt{ptr} (up to the minimum of the old and new sizes)
Lab: Framework Provided

- We provide one of the simplest possible implementations:
  - `malloc`: sequentially allocate a block increasing the top of heap pointer (`brk`)
  - `free`: do nothing
  - `realloc`: allocate a new block with the requested size

Although this solution is formally correct and very fast, it has a very bad space utilization.
Lab: Design

First, think about the design of your memory allocator:

- Choose the allocation strategy: first fit, best fit, …
- Choose a data structure(s): ordered list, binary tree, …
- Think on how to mark the blocks: in general how to store the size of the block, the free tag and the eventual pointers to manage the data structure (e.g., next pointer for a list, left, right for tree, …)
- Think about the API of the functions to manage your structure (flexibility)
- Choose a time to perform coalescing (merging):
  - immediate (when putting a block on the free list)
  - deferred (e.g., when allocation fails)
Lab: Design

- How to choose the correct strategy?
  - Analyze the needs (space, speed, predictability)
  - Analyze the environment (experimentally)
  - Test different strategies