



Lecture 5: Memory Management

CREATING THE NEXT®

Administrivia

- Assignment and Sheet 1 due date postponed to Sep 13th
- Assignment and Sheet 2 will be released on Sep 9th
- Come in-person to office hours
- Additional office hours posted on sign up sheet

Today's Agenda

Recap

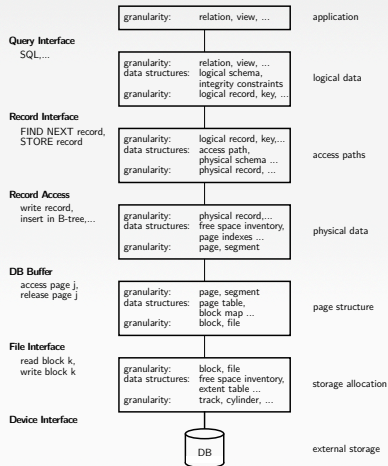
Dynamic Memory Management

Segments

System Catalog

Recap

Layered Architecture

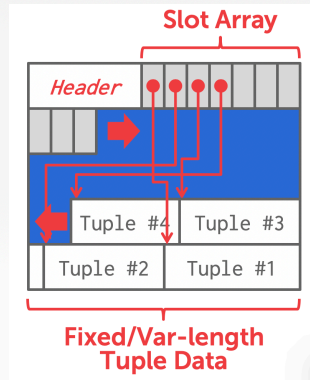


Database System Architectures

- Disk-Centric Database System
 - ▶ The DBMS assumes that the primary storage location of the database is HDD.
- Memory-Centric Database System
 - ▶ The DBMS assumes that the primary storage location of the database is DRAM.

Slotted Pages

- The most common page layout scheme is called slotted pages.
- The **slot array** maps "slots" to the tuples' starting position offsets.
- The header keeps track of:
 - ▶ The number of used slots
 - ▶ The offset of the starting location of the last slot used.



Dynamic Memory Management

Virtual Address Space

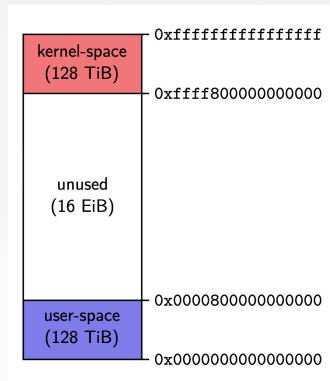
Each Linux process runs within its own virtual address space

- The kernel pretends that each process has access to a (huge) continuous range of addresses (≈ 256 TiB on x86-64)
- Virtual addresses are mapped to physical addresses by the kernel using page tables and the memory management unit (MMU)
- Greatly simplifies memory management code in the kernel and improves security due to memory isolation
- Allows for useful “tricks” such as memory-mapping files

Virtual Address Space

The kernel also uses virtual memory

- Part of the address space has to be reserved for kernel memory
- This kernel-space memory is mapped to the same physical address for each process
- Access to this memory is restricted
- Most of the address space is unused
- MMUs on x86-64 platforms only support 48 bit pointers at the moment



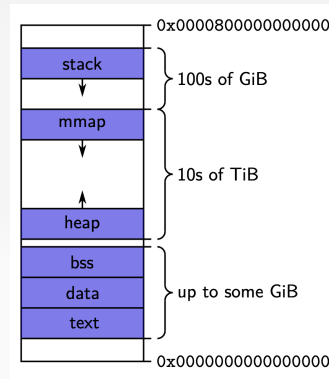
Virtual Address Space

User-space memory is organized in segments:

- Stack segment
- Memory mapping segment
- Heap segment
- BSS, data and text segments

Segments grow over time

- Stack and memory mapping segments usually grow down (i.e. addresses decrease)
- Heap segment usually grows up (i.e. addresses increase)



Stack Segment

Stack memory is typically used for objects with automatic storage duration

- The compiler can statically decide when allocations and deallocations must happen
- The memory layout is known at compile-time
- Allows for highly optimized code (allocations and deallocations simply increase/decrease a pointer)

Fast, but inflexible memory

- The stack grows and shrinks as functions push and pop local variables
- Stack variables only exist while the function that created them is running
- No dynamic data structures are possible (trees, graphs, *e.t.c.*)

Stack Segment

All variables are allocated using stack memory

```
include <stdio.h>
double multiplyByTwo (double input) {
    double twice = input * 2.0; return twice;
}

int main (int argc, char *argv[]){
    int age = 30; double salary = 12345.67;
    double myList[3] = {1.2, 2.3, 3.4};
    printf("double your salary is %.3f\n", multiplyByTwo(salary));
    return 0;
}
```

Heap Segment

The heap is typically used for objects with dynamic storage duration

- The programmer must explicitly manage allocations and deallocations
- Allows for more flexible memory management

Disadvantages

- Performance impact of heap-based memory allocator
- Memory fragmentation
- Dynamic memory allocation is error-prone
 - ▶ Memory leaks
 - ▶ Double free (deallocation)
 - ▶ Make use of debugging tools! ([GDB](#), [Valgrind](#), [ASAN](#))

Heap Segment

All variables are allocated using heap memory

```
include <stdio.h>
include <stdlib.h>
double *multiplyByTwo (double *input) {
    double *twice = malloc(sizeof(double));
    *twice = *input * 2.0; return twice;
}

int main (int argc, char *argv[]) {
    int *age = malloc(sizeof(int)); *age = 30;
    double *salary = malloc(sizeof(double)); *salary = 12345.67;
    double *twiceSalary = multiplyByTwo(salary);
    printf("double your salary is %.3f\n", *twiceSalary);
    free(age); free(salary); free(twiceSalary);
    return 0;
}
```

Dynamic Memory Management in C++

C++ provides several mechanisms for dynamic memory management

- Through new and delete expressions (discouraged)
- Through the C functions malloc and free (discouraged)
- Through smart pointers and ownership semantics (preferred)

Mechanisms give control over the storage duration and possibly lifetime of objects

- Level of control varies by method
- In all cases: manual intervention required

Dynamic Memory Management in C++

Key functions and features

- `std::memcpy` : copies bytes between non-overlapping memory regions
- `std::memmove` : copies bytes between possibly overlapping memory region
- `std::unique_ptr`: assumes unique ownership of another C++ object through a pointer

Dynamic Memory Management in C++

Key functions and features

- copy semantics: Assignment and construction of classes typically employ copy semantics
- move semantics: Move constructors/assignment operators typically “steal” the resource of the argument

```
struct A {  
    A(const A& other);  
    A(A&& other);  
};  
  
int main() {  
    A a1;  
    A a2(a1); // calls copy constructor  
    A a3(std::move(a1)); // calls move constructor  
}
```

Memory Mapping Files

POSIX defines the function `mmap()` in the header `<sys/mman.h>` which can be used to manage the virtual address space of a process.

```
void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- Arguments have different meaning depending on flags
- On error, the special value `MAP_FAILED` is returned
- If a pointer is returned successfully, it must be freed with `munmap()`

```
int munmap(void* addr, size_t length)
```

- `addr` must be a value returned from `mmap()`
- `length` must be the same value passed to `mmap()`
- `munmap()` should be called to follow the

Resource Acquisition Is Initialization (RAII) principle

Memory Mapping Files

One use case for `mmap()` is to map the contents of a file into the virtual memory. To map a file, the arguments are used as follows:

```
void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- `addr`: hint for the kernel which address to use, should be `nullptr`
- `length`: length of the returned memory mapping (usually multiple of page size)
- `prot`: determines how the mapped pages may be accessed and is a combination (with bitwise or) of the following flags:
 - ▶ `PROT_EXEC`: pages may be executed
 - ▶ `PROT_READ`: pages may be read
 - ▶ `PROT_WRITE`: pages may be written
 - ▶ `PROT_NONE`: pages may not be accessed

Memory Mapping Files

One use case for `mmap()` is to map the contents of a file into the virtual memory. To map a file, the arguments are used as follows:

`void*` `mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)`

- `flags`: should be either `MAP_SHARED` (changes to the mapped memory are written to the file) or `MAP_PRIVATE` (changes are not written to the file)
- `fd`: descriptor of an opened file
- `offset`: Offset into the file where the mapping should start (multiple of page size)

Memory Mapping Files

Example of reading integers from file /tmp/ints:

- Note: This assumes that integers are written in binary format to the file!
- Using `mmap()` to read from large files is often faster than using `read()`
- This is because with `mmap()` data is directly read from and written to the file without copying it to a buffer first

```
int fd = open(`/tmp/ints', O_RDONLY);  
void* mappedFile= mmap(nullptr, 4096, PROT_READ, MAP_SHARED, fd, 0);  
int* fileInts= static_cast<int*>(mappedFile);  
for (int i = 0; i < 1024; ++i)  
    std::cout<< fileInts[i] << std::endl;  
munmap(mappedFile, 4096);  
close(fd)
```

Using mmap for Memory Allocation

mmap() can also be used to allocate memory by not associating it with a file.

- flags must be MAP_PRIVATE | MAP_ANONYMOUS
- fd must be -1; offset must be 0
- Used by malloc() internally
- Should be used manually only to allocate large regions of memory (e.g., several MBs)

Example of allocating 100 MiB of memory:

```
void* mem = mmap(NULL, 100 * (1ull << 20),  
    PROT_READ | PROT_WRITE,  
    MAP_PRIVATE | MAP_ANONYMOUS,  
    -1, 0);
```

```
// [...]  
munmap(mem, 100 * (1ull << 20));
```

Tuple Layout

- A tuple is essentially a sequence of bytes.
- The DBMS needs a way to keep track of individual tuples.
- Each tuple is assigned a unique record identifier: TID.

```
std::vector<char> tuple_data;
```

```
struct TID {  
    explicit TID(uint64_t raw_value);  
    TID(uint64_t page, uint16_t slot);  
    /// The TID could, for instance, look like the following:  
    /// - 48 bit page id  
    /// - 16 bit slot id  
    uint64_t value;  
};
```


Tuple Schema

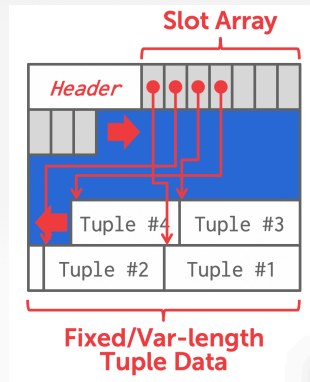
- It's the job of the DBMS to interpret those bytes into attribute types and values.

```
std::vector<schema::Table> tables{
    schema::Table(
        "customer",
        {
            schema::Column("c_custkey", schema::Type::Integer()),
            schema::Column("c_name", schema::Type::Varchar(25)),
            schema::Column("c_address", schema::Type::Varchar(40)),
            schema::Column("c_acctbal", schema::Type::Numeric(12, 2)),
        }
    );
};
```

```
auto schema = std::make_unique<schema::Schema>(std::move(tables));
```

Page Layout

- The most common page layout scheme is called slotted pages.
- The **slot array** maps "slots" to the tuples' starting position offsets.
- The header keeps track of:
 - ▶ The number of used slots
 - ▶ The offset of the starting location of the last slot used.



Page Layout

- The header keeps track of:
 - ▶ The number of used slots
 - ▶ The offset of the starting location of the last slot used.

```
struct SlottedPage {  
    struct Header {  
        // Constructor  
        explicit Header(char *_buffer_frame, uint32_t page_size);  
        /// overall page id  
        uint64_t overall_page_id;  
        /// location of the page in memory  
        char *_buffer_frame;  
        /// Number of currently used slots  
        uint16_t slot_count;  
        /// Lower end of the data  
        uint32_t data_start;  
    };  
};
```

Page Layout

- The slot array maps "slots" to the tuples' starting position offsets.

```
struct SlottedPage {  
    ...  
    struct Slot {  
        Slot() = default;  
        /// The slot value  
        uint64_t value;  
    };  
    /// Constructor.  
    explicit SlottedPage(char *buffer_frame, uint32_t page_size);  
    /// Slot helper functions  
    TID addSlot(uint32_t size);  
    void setSlot(uint16_t slotId, uint64_t value);  
    Slot getSlot(uint16_t slotId);  
};  
/// Slot array  
auto *slots = reinterpret_cast<Slot *>(header.buffer_frame + sizeof(header));
```

Segments

Segments

While page granularity is fine for I/O, it is somewhat unwieldy

- most data structures within a DBMS span multiple pages
- convenient to treat these as one entity: segment
- relations, indexes, free space inventory (FSI), *e.t.c.*
- each logical DBMS structure is managed as a segment

Conceptually similar to file (but supports non-linear ordering of data).

Segments

A segment offers a virtual address space within the DBMS

- can allocate and release new pages
- can iterate over all pages
- can drop the whole segment
- offers a non-linear address space

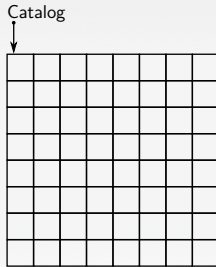
Greatly simplifies the logic of higher layers.

Segments

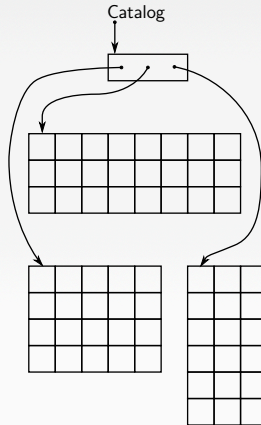
Example: pages from R1 | pages from R2 | pages from R1

- Dropping relation R2 → hole in the segment
- New pages from R1 may be inserted into the hole
- Logical insertion order of R1 does not match the physical storage order in segment
- Need ORDER BY to guarantee logical ordering

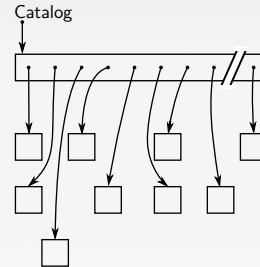
Disk Block Mapping



static file-mapping



dynamic extent-mapping



dynamic block-mapping

Disk Block Mapping

All approaches have pros and cons:

- ❶ static file-mapping
 - ▶ very simple, low overhead
 - ▶ resizing is difficult
- ❷ dynamic block-mapping
 - ▶ maximum flexibility
 - ▶ administrative overhead, additional indirection
- ❸ dynamic extent-mapping
 - ▶ can handle growth
 - ▶ slight overhead

In most cases extent-based mapping is preferable.

Disk Block Mapping

The units of database space allocation are **disk blocks, extents, and segments.**

- A disk block is the smallest unit of data used by a database.
- An extent is a logical unit of database storage space allocation made up of a number of **contiguous** disk blocks.
- One or more extents in turn make up a segment.
- When the existing space in a segment is completely used, the DBMS allocates a new extent for the segment.

Disk Block Mapping

A segment is a set of extents that contains all the data for a specific logical storage structure within a tablespace.

- For each table, the DBMS allocates one or more extents to form that table's data segment
- For each index, the DBMS allocates one or more extents to form its index segment.

Disk Block Mapping

Dynamic extent-mapping:

- grows by adding a new extent
- should grow exponentially (*e.g.*, factor 1.25)
- exponential growth bounds the number of extents
- reduces both complexity and space consumption
- and helps with sequential I/O! Why?

Segment Types

Segments can be classified into types

- public vs. private (*e.g.*, list of segments) // visibility to the user
- permanent (*e.g.*, relation) vs. temporary (*e.g.*, intermediate output of a relational operator in the query plan)
- automatic vs. manual
- with recovery vs. without recovery

Differ in complexity and required effort.

Private Segments

Most DBMS will need at least two private segments:

- segment inventory
 - ▶ keeps track of all disk blocks allocated to segments
 - ▶ keeps extent lists or page tables or ...
- free space inventory (FSI)
 - ▶ keeps track of free pages
 - ▶ maintains bitmaps or free extents or ...

Public Segments

Public segments built upon these low-level private segments.

Common high-level segments:

- schema
- relations
- temporary segments (created and discarded on demand)
- ...

Slotted Page Segment

Slotted Page Segment

```
class SPSegment : public buzzdb::Segment {
public:
    /// Constructor
    SPSegment(uint16_t segment_id, BufferManager &buffer_manager,
              SchemaSegment &schema, FSISegment &fsi);
    /// Allocate a new record.
    TID allocate(uint32_t record_size);
    /// Read the data of the record into a buffer.
    uint32_t read(TID tid, std::byte *record, uint32_t capacity) const;
    /// Write a record.
    uint32_t write(TID tid, std::byte *record, uint32_t record_size);
    /// Resize a record.
    void resize(TID tid, uint32_t new_size);
    /// Removes the record from the slotted page
    void erase(TID tid);
};
```

Slotted Page Segment

Slotted Page Segment

```
class SPSegment : public buzzdb::Segment {  
    ...  
protected:  
    /// Schema segment  
    SchemaSegment &schema;  
    /// Free space inventory  
    FSISegment &fsi;  
};
```

System Catalog

System Catalog

- A DBMS stores meta-data about databases in its internal catalog.
 - ▶ List of tables, columns, indexes, views
 - ▶ List of users, permissions
 - ▶ Internal statistics (e.g., disk reads, storage space allocation)
- Almost every DBMS stores their catalog as a private database.
 - ▶ Wrap object abstraction around tuples.
 - ▶ Specialized code for “bootstrapping” catalog tables. Why?

System Catalog

- You can query the DBMS's INFORMATION_SCHEMA database to get info.
 - ▶ ANSI standard set of read-only views that provide info about all of the tables, views, columns, and procedures in a database
 - ▶ DBMSs also have non-standard shortcuts to retrieve this information.

Accessing Table Schema

SQL Fiddle: [Link](#)

- **Task:** List all the tables in the database.

--- SQL 92

```
SELECT * FROM INFORMATION_SCHEMA.TABLES  
WHERE table_schema = 'public';
```

--- PostgreSQL

\d

--- MySQL

```
SHOW TABLES;
```

--- SQLite

```
.tables;
```

Accessing Table Schema

- **Task:** List all the columns in the students table.

--- SQL 92

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS  
WHERE table_name = 'students';
```

--- PostgreSQL

```
\d student
```

--- MySQL

```
DESCRIBE student;
```

--- SQLite

```
.schema student;
```

Conclusion

- The units of database space allocation are disk blocks, extents, and segments
- A DBMS stores meta-data about databases in its internal catalog

Next Class

- Data Representation