Lab 7: Memory Management

Innopolis University
Course of Operating Systems

Recap

- What is a signal in Linux?

Recap

- What is a signal in Linux?
  - A software interrupt delivered to process that represents an event that may require some response from the process or thread.

Recap

- What is a signal in Linux?
  - A software interrupt delivered to process that represents an event that may require some response from the process or thread.
- Name a few signals that are common to see in Linux systems.

Recap

- What is a signal in Linux?
  - A software interrupt delivered to process that represents an event that may require some response from the process or thread.
- Name a few signals that are common to see in Linux systems.
  - For instance: SIGNINT, SIGKILL, SIGSTOP, SIGCONT, SIGALRM, SIGTERM, SIGUSR1, SIGUSR2

Recap

- What does it mean for process scheduling to be preemptive?

Recap

- What does it mean for process scheduling to be preemptive?
  - Preemptive scheduling allocates CPU resources for a limited time only, so process can be put back into scheduling queue.

Recap

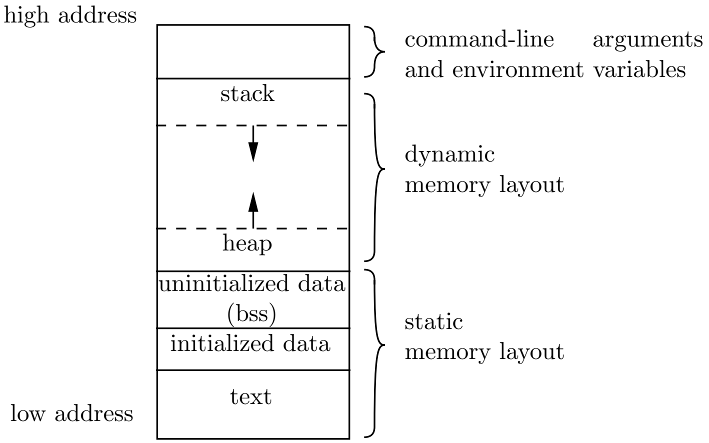
- What does it mean for process scheduling to be preemptive?
  - Preemptive scheduling allocates CPU resources for a limited time only, so process can be put back into scheduling queue.
- Name a few ways to do process scheduling

Recap

- What does it mean for process scheduling to be preemptive?
  - Preemptive scheduling allocates CPU resources for a limited time only, so process can be put back into scheduling queue.
- Name a few ways to do process scheduling
  - For example: First in First out, Round Robin, Shortest job next

Memory Layout of C Program

An object or executable file in Linux consists of several sections shown as follows.



Text or Code Segment

- Contains machine code of the compiled program. The text segment of an executable/object file is often read-only segment that prevents a program from being accidentally modified.

Initialized Data or Data Segment

- Stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand

Uninitialized Data or .bss Segment

- Stores all uninitialized global, static, and external variables (declared with extern keyword)

Stack Segment

- Stores all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over

Heap Segment

- Part of RAM where dynamically allocated variables are stored. In C language dynamic memory allocation is done by using **malloc** and **calloc** functions.

Memory Layout of C Program (Example)

Example:

- Write a C program **ex1.c** which includes only a main function as follows: `int main(){return 0;}`
- Produce the object file of the program by running “gcc -c ex1.c -o ex1.o” then the executable file by running “gcc ex1.c -o ex1.out”.
- Using **size** shell command, check the size of memory segments of the program for both **ex1.o** and **ex1.out** files . Discuss the difference.
- Add to the program, an integer global variable **g_vari** with initial value 1, another integer global variable **g_varu** without an initial value and calculate the size of segments again. Discuss the difference. Which memory segments are changed? why?
- Declare local variables in the main function and check the size of segments again.

Multiple ways to store program data

- Static global data
  - Fixed size at compile-time.
  - Entire lifetime of the program.
  - Portion is read-only (e.g. string literals).
  - Example (`array` variable).
- Stack-allocatd data
  - Local/temporary variables
  - known lifetime (deallocated on `return`)
  - Examples (`tmp` and `local_array` variables).
- Dynamic (heap allocated) data
  - Size known at runtime.
  - Lifetime known at runtime.
  - Example (`dyn` variable).

```
int array[1024];

int* foo(int n){
    int tmp;
    int local_array[n];

    int* dyn = (int*)
        malloc(n*sizeof(
        int));
    return dyn;
}
```

Pointers revision (1/3)

- Each variable represents an address in memory and a value.
- Address: &variable = address of variable
- A pointer is a variable that “points” to the block of memory that a variable represent

Pointers revision (2/3)

- Declaration: *data_type *pointer_name;*
- Example:

```
char x = 'a';
char *ptr = &x; // ptr points to a char x
```

- Pointers are integer variables themselves, so can have pointer to pointers:

```
char **ptr;
```

Pointers revision (3/3)

- Dereferencing = Using Addresses

```
int x = 5;
int *ptr = &x;
// Access x via ptr, and changes it to 6
*ptr = 6;
// Will print 6 now
printf("%d", x);
```

Why use pointers?

- Pass-by-reference rather than value

```
void sample_func(char* str_input);
```

- Manipulate memory effectively
- Useful for arrays (Array in C - a pointer and a length)

malloc()

- void *malloc(size_t size)
  Example:

  ```
  int array[10]; // the same as
  int *array = malloc(10*sizeof(int));
  ```

- malloc() does not initialize the array; this means that **the array may contain random or unexpected values**

calloc()

- void *calloc(size_t nmemb, size_t size);
- The calloc() function allocates space for an array of items and **initializes the memory to zeros**

realloc()

- void *realloc(void *ptr, size_t size);
- The realloc() function changes the size of the object **pointed to by *ptr*** to the **size specified by *size***

free()

- void free(void *ptr)
- Releases memory allocated by malloc(), calloc() or realloc()

```
int *myStuff = malloc(20 * sizeof(int));
if (myStuff != NULL)
    {
        /* more statements here */
        /* time to release myStuff */ free( myStuff );
    }
```

Dynamic memory allocation (Example)

Example:

- Write a C program that dynamically allocates memory for an array of N integers, fills the array with incremental values starting from 0, prints the array and deallocates the memory. The program should prompt the user to enter N before allocating the memory.

brk & sbrk

- The program break **(brk)** identifies the end of the heap segment.
- We can allocate more space from the memory (increase the size of heap segment) using function **brk()** and **sbrk()**. They are defined as follows:
  - `#include "unistd.h"`
  - `int brk(void *addr);`
  - `void *sbrk(intptr_t increment);`
- **brk()** system call assigns the value of *addr* to the ending of the heap segment. **sbrk()** increases the heap space of the program by *increment* bytes.
- The present location of the program break can be found by calling **sbrk()** with just a raise of 0.
- **Note:** The **brk** and **sbrk** functions are historical curiosities left over from earlier days before the advent of virtual memory management. Historically, **malloc()** worked using **sbrk()** or **brk()**, but many modern allocators use **mmap** nowadays.

brk & sbrk

Example:

- Write a C program which contains only empty main function and print the end address of the data segment.
- Modify the program to dynamically allocate memory for an array of N integers, fill the array with incremental values starting from 0, print the array and deallocate the memory. The program should prompt the user to enter N before allocating the memory.
- Double the size of the heap segment using **brk** system call.
- What is the maximum size for the program's data segment? While running the program, check the file */proc/[pid]/limits*. You can change the limits of the process using **prlimit** command.
- **Note:** The hard limit is the maximum value that is allowed for the soft limit. Any changes to the hard limit require root access. The soft limit is the value that Linux uses to limit the system resources for running processes. The soft limit cannot be greater than the hard limit.

Dynamic memory allocators

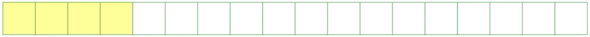
- A dynamic memory allocator manages an area of the process's memory known as the heap.
- An allocator divides and maintains the heap as a collection of various-sized blocks. Each block is a contiguous chunk of memory that is either allocated or free.
- Allocator usually requests *pages* in the heap region. The virtual memory hardware and OS kernel allocate these pages to the process. Page is the smallest unit of data for memory management in the virtual memory of the OS.
- Application objects are typically smaller than *pages*, so the allocator manages blocks within pages
- By definition, an allocated block has been reserved by the application for use, and a free block is available for allocation. An allocated block remains allocated until it is free. The same is true for the free block.

Dynamic Memory allocation

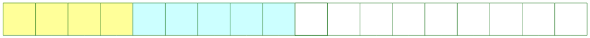
- For faster access, a memory block should be aligned, and usually by the size of the machine word (`size_t` size). On a 32-bit system `size_t` will take 32 bits (4 bytes), on a 64-bit one 64 bits (8 bytes).
- “aligned” allocations will be in sizes that are multiples of the word size.
- Applications/user programs can issue arbitrary sequence of `malloc` and `free` requests. They must never access memory not currently allocated. They must never free memory not currently allocated.
- Allocators cannot control the number or size of allocated blocks. They must respond immediately to `malloc`. They must allocate blocks from free memory. They must align blocks so that they satify the alignment requirements. They cannot move the allocated blocks so defragemntation is not allowed, otherwise this could break your pointers.

Dynamic memory allocation

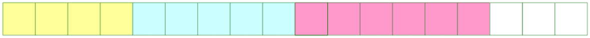
```
p1 = malloc(32)
```



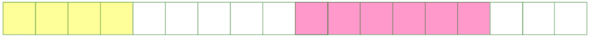
```
p2 = malloc(40)
```



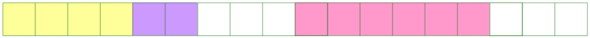
```
p3 = malloc(48)
```



```
free(p2)
```



```
p4 = malloc(16)
```



Each block is an 8-byte word (64-bit system). Colored blocks are allocated and others are free.

Dynamic memory allocation

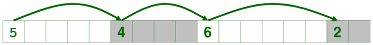
Some performance goals of allocators:

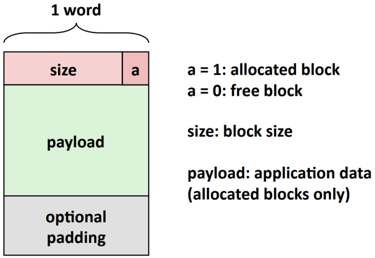
- Goal: Given some sequence of `malloc` and `free` requests $R_0, R_1, ..., R_k, ..., R_{n-1}$, maximize **throughput** and **peak memory utilization**.
- **Throughput:** number of requests per unit time. For example, if 5000 malloc calls and 5000 free calls completed in 10 seconds, then throughput is $1,000$ operations/second.
- **Aggregate payload P_k:** is the sum of currently allocated payloads after request R_k has completed.
- **Current heap size H_k:** Assume that H_k is monotonically increasing. We can increase the size of the heap using `sbrk`.
- **Peak memory utilization U_k:** is the maximum aggregate payload $max(P_i)$ divided by current heap size H_k after k requests for all $i <= k$.

Dynamic memory allocation

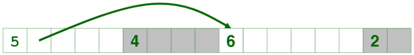
There are several methods to implement dynamic allocators.

- **Implicit free list:** For each block we need to store the block size and whether it is allocated or not.





- **Explicit free list:** The pointers only within the free blocks.



- **Segregated free list:** Different free lists for different size ranges.
- etc...

Implicit Free List: Finding a Free Block

There are different algorithms to find a free block for the required size.

- **First fit:** Search list from beginning, choose first free block that fits the required size.
  - Can take linear time in total number of blocks (allocated/free).
  - Can cause "splinters" (small free blocks) at beginning of list.
- **Best fit:** Choose the free block with the closest size that fits the required size (requires complete search of the list).
  - Keeps fragments small – usually helps fragmentation
  - Will typically run slower than first-fit
- **Worst fit:** it locates the largest available free block so that the block left will be big enough to be useful. It is the reverse of best fit.

Exercise 1 (1/4)

- You are tasked with implementing a simple memory allocator simulation in C.
- Write a program **allocator.c** which creates an unsigned **integer** array for storing 10^7 integers (maximum memory size) to serve as memory cells. On startup, all cells are initialized to zeros (0) (i.e. all memory cells are free).
- Implement the following functions:
  1. `void allocate_<algo>(adrs, size)`:
     - This function should find a contiguous free cells from the array for storing at least n integers where $n =$`size`, then it should set these cells to the value `adrs`.
     - `adrs` will act as an identifier for reserved cells. It should be a non-zero unsigned integer and does not represent an index in the array since the indices of allocated cells should be determined by the allocator at run-time and based on the selected algorithm and status of the memory.
     - `size` can take values between 1 and 10^7.

Exercise 1 (2/4)

2. `void clear(adrs)`:
   - This function searches for memory cells that hold the given address (`adrs`) and frees it.
   - `adrs` should be a non-zero unsigned integer.

- **Note: for simplicity, we consider here that `adrs` are only identifiers for the allocated memory cells which can be used by `clear` queries.**
- You are required to implement each of the following: First Fit, Best Fit, Worst Fit, to use as the algorithm for the allocate function. At the end, you should have 3 allocate functions, one function for each algorithm.
- Your program should accept input from a file named `queries.txt`. The file will contain a list of queries, with each query on a new line. The end of the file will be indicated by the line “end”.

Exercise 1 (3/4)

- The queries in the file can be one of two types:
  - `allocate adrs size`: Allocate memory of the specified `size` for the given `adrs`.
  - `clear adrs`: Clear the memory space associated with the given `adrs`.
- **Note:** we assume that the file **queries.txt** always contains valid input (e.g. there is no clear query for an address not allocated). You just need to check if the file exists.
- For each of the three allocation algorithms, you should reset the memory array (free all memory cells), execute all the queries from `queries.txt` file, and then print the throughput (performance metric). Throughput in this context simply refers to the number of queries executed divided by the total allocation time.

Exercise 1 (4/4)

- Example for `queries.txt`:

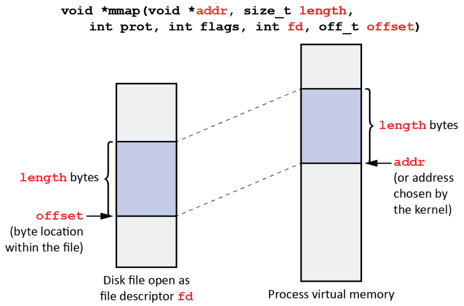
```
allocate 1001 512
allocate 2002 256
clear 1001
end
```

- You can find a bigger sample for `queries.txt` from here.
- You should submit **allocator.c**. You also need to compare the performance of the algorithms (assuming that they are receiving the same input queries) and add the performance results and your findings to **ex1.txt**. Submit **ex1.txt** too.
- The total allocation time starts from the very first query and ends after executing the last query. You can use the function **clock()** from **time.h**.

mmap

- The `mmap()` function is used for mapping between a process address space and either files or devices. When a file is mapped to a process address space, the file can be accessed like an array in the program.
- We do not necessarily need a file to use **mmap** and instead we can use "anonymous mappings". Anonymous mappings are not backed by any file on disk (so nothing is transferred from disk) and the requested memory is initialized to zero. Anonymous mapping is quite useful for building allocators as **mmap** provides a convenient interface with which we can request memory directly from the operating system.
- For using `mmap()`, you need to include the header file `<sys/mman.h>`.
- The **malloc()** implementation uses **sbrk** for small allocations and **mmap** for larger ones.

mmap



void *mmap(void *addr, size_t length,
int prot, int flags, int fd, off_t offset)
length bytes
offset
(byte location within the file)
Disk file open as file descriptor fd
length bytes
addr
(or address chosen by the kernel)
Process virtual memory

mmap function

Note: The file should be mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining bytes in the partial page at the end of the mapping are zeroed when mapped, and modifications to that region are not written out to the file.

mmap (Example)

Example:

- Using mmap, allocate 100MiB space (char pointer) from the memory with read/write permissions and not shared with other processes.
- Initialize the pointer cells with random values using rand function.
- Return the location of numbers which are divisible by 11.
- unmap the allocated region.
- Repeat the steps above using **sbrk** function instead of **mmap**.

Exercise 2 (1/2)

- In this exercise, you should use `mmap` system call to a big text file **text.txt** in chunks.
- This exercise requires generating a relatively large file of size *500MiB*.
- The content of the text file **text.txt** should be generated from "/dev/random". The file "/dev/random" is a special character file that generates pseudorandom numbers.
- Write the program **ex2.c** to perform the following tasks using memory mapping.
  - Create the empty file **text.txt**. Open the file "/dev/random", and read a character c at a time.
  - If the generated character c is a printable character (use **isprint** function from **ctype.h** to check it), then we should add it to the file **text.txt**. Otherwise, we ignore the character and generate a new character.

Exercise 2 (2/2)

- You need to add a new line to **text.txt** after adding 1024 characters (max line length). You should continue adding characters till you get a file of size *500 MiB= 500*1024KiB.*
- After finishing the process above, you would get a relatively large file in your file system.
- Open the whole file **text.txt** in chunks where the chunk size is *1024th multiple of the page size in your system.* You can get the page size in C as follows: (if page size is *4KiB* then chunk size is 4MiB)

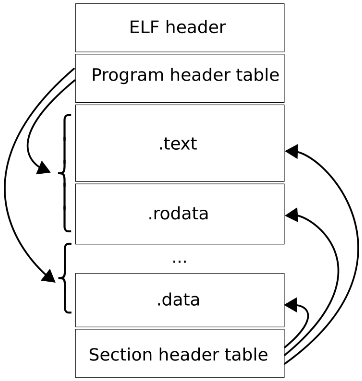
```
#include <unistd.h>
long sz = sysconf(_SC_PAGESIZE);
```

- Count the capital letters in the mapped chuncks. Print the total number of the capital letters in the file to `stdout`.
- Replace the capital letters with lowercase letters in the file.
- `unmap` the mapped memory.

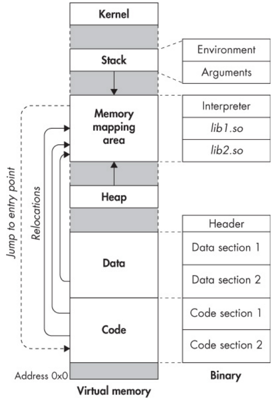
- Submit `ex2.c`

Binary file layout in Linux

Executable and Linkable Format (ELF), is the default binary format on Linux-based systems. (check this link for more info)



ELF file in disk



ELF loaded in memory

End of lab 7