TCSS 422: OPERATING SYSTEMS

Address Spaces and the Memory API

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Introduction to memory virtualization
- The address space
- Goals of OS memory virtualization
- Memory API
- Common memory errors

MEMORY VIRTUALIZATION

- What is memory virtualization?
- This is not "virtual" memory,
  - Classic use of disk space as additional RAM
  - When available RAM was low
  - Less common recently

MEMORY VIRTUALIZATION - 2

- Presentation of system memory to each process
- Appears as if each process can access the entire machine's address space
- Each process's view of memory is isolated from others
- Everyone has their own sandbox

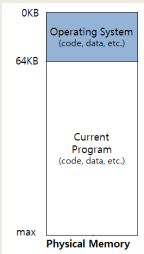
Process A Process B Process C

MOTIVATION FOR MEMORY VIRTUALIZATION

- Easier to program
  - Programs don't need to understand special memory models
- Abstraction enables sophisticated approaches to manage and share memory among processes
- Isolation
  - From other processes: easier to code
- Protection
  - From other processes
  - From programmer error (segmentation fault)

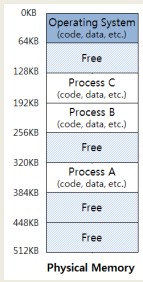
EARLY MEMORY MANAGEMENT

- Load one process at a time into memory
- Poor memory utilization
- Little abstraction



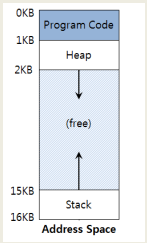
MULTIPROGRAMMING WITH SHARED MEMORY

- Later machines supported running multiple processes
- Swap out processes during I/O waits to increase system utilization and efficiency
- Swap entire memory of a process to disk for context switch
- Too slow, especially for large processes
- Solution→
  - Leave processes in memory
- Need to protect from errant memory accesses in a multiprocessing environment



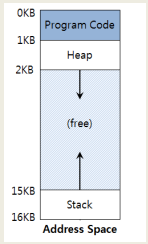
ADDRESS SPACE - 3

- Program code
  - Static size
- Heap and stack
  - Dynamic size
  - Grow and shrink during program execution
  - Placed at opposite ends
- Addresses are virtual
  - They must be physically mapped by the OS



ADDRESS SPACE

- Easy-to-use abstraction of physical memory for a process
- Main elements:
  - Program code
  - Stack
  - Heap
- Example: 16KB address space



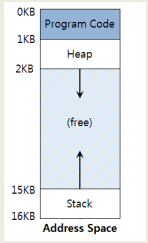
VIRTUAL ADDRESSING

- Every address is virtual
  - OS translates virtual to physical addresses

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

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```


ADDRESS SPACE - 2

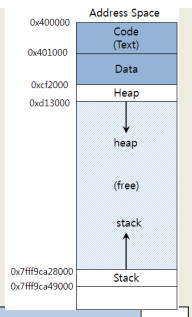
- Code
  - Program code
- Stack
  - Program counter (PC)
  - Local variables
  - Parameter variables
  - Return values (for functions)
- Heap
  - Dynamic storage
  - Malloc() new()



VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:

```
location of code: 0x400686
location of heap: 0x1129420
location of stack: 0x7ffe040d77e4
```



GOALS OF OS MEMORY VIRTUALIZATION

- Transparency
  - Memory shouldn't appear virtualized to the program
  - OS multiplexes memory among different jobs behind the scenes
- Protection
  - Isolation among processes
  - OS itself must be isolated
  - One program should not be able to affect another (or the OS)

MALLOC

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocates memory on the heap
- size_t unsigned integer (must be +)
- size size of memory allocation in bytes
- Returns
- SUCCESS: A void * to a memory address
- FAIL: NULL
- sizeof() often used to ask the system how large a given datatype or struct is

GOALS - 2

- Efficiency
  - Time
    - Performance: virtualization must be fast
  - Space
    - Virtualization must not waste space
    - Consider data structures for organizing memory
    - Hardware support TLB: Translation Lookaside Buffer
- Goals considered when evaluation memory virtualization schemes

SIZEOF()

- Not safe to assume data type sizes using different compilers, systems
- Dynamic array of 10 ints
- Static array of 10 ints

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

```
int x[10];
printf("%d\n", sizeof(x));
```

```
40
```


THE MEMORY API

FREE()

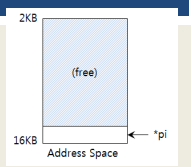
```
#include <stdlib.h>

void free(void* ptr)
```

- Free memory allocated with malloc()
- Provide: (void *) ptr to malloc'd memory
- Returns: nothing

VIRTUAL ADDRESS SPACE

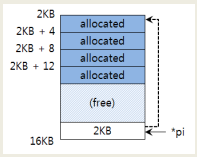
```
int *pi; // local variable
```

- Pointer is a local variable on the stack



- Malloc returns space on the heap

```
pi = (int *)malloc(sizeof(int)* 4);
```

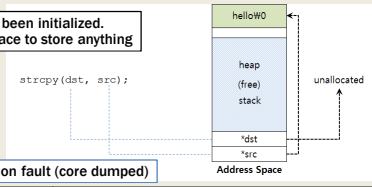


FORGETTING TO MALLOC

- C is not Java
- When forgetting to maloc:

```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```

dst has not been initialized.
It has no place to store anything

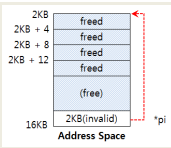


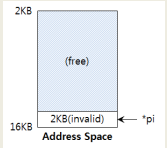
Segmentation fault (core dumped)

VIRTUAL ADDRESS SPACE - 2

- Releases heap space pointed to by the pointer on the stack

```
free(pi);
```

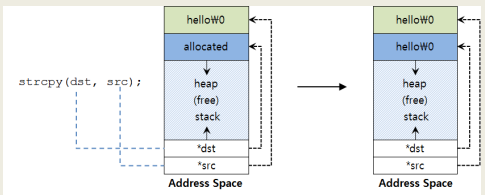




CORRECTION

```
char *src = "hello";  //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);     //work properly
```

- Why do we malloc length + 1 ?



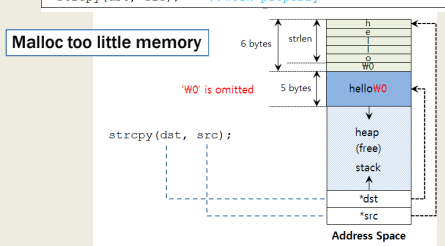
COMMON MEMORY ERRORS

- Forgetting to malloc memory
- Unterminated string
- Uninitialized memory
- Memory leak
- Dangling pointer

UNTERMINATED STRING

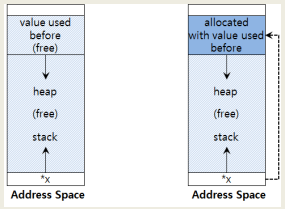
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```

Malloc too little memory



FORGETTING TO INITIALIZE

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```




```
#include<stdio.h>

int * set_magic_number_a()
{
  int a =53247;
  return &a;
}

void set_magic_number_b()
{
  int b = 11111;
}

int main()
{
  int * x = NULL;
  x = set_magic_number_a();
  printf("The magic number is=%d\n",*x);
  set_magic_number_b();
  printf("The magic number is=%d\n",*x);
  return 0;
}
```

What will this code do?

Output:

```
$ ./pointer_error
The magic number is=53247
The magic number is=11111
```

We have not changed *x but the value has changed!!

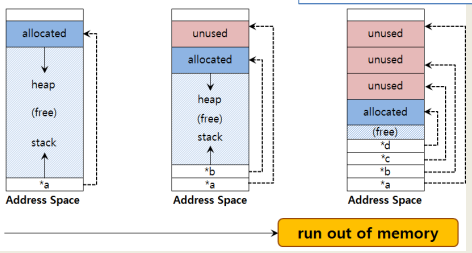
Why?

28

MEMORY LEAK

unused : unused, but not freed

Program runs out of memory and eventually dies...



DANGLING POINTER (1/2)

- Dangling pointers arise when a variable referred (a) goes "out of scope", and it's memory is destroyed/overwritten (by b) without modifying the value of the pointer (*x).
- The pointer still points to the original memory location of the deallocated memory (a), which has now been reclaimed for (b).


```
#include<stdio.h>

int * set_magic_number_a()
{
  int a =53247;
  return &a;
}

void set_magic_number_b()
{
  int b = 11111;
}

int main()
{
  int * x = NULL;
  x = set_magic_number_a();
  printf("The magic number is=%d\n",*x);
  set_magic_number_b();
  printf("The magic number is=%d\n",*x);
  return 0;
}
```

What will this code do?

27

DANGLING POINTER (2/2)

- Fortunately in the case, a compiler warning is generated:

```
$ g++ -o pointer_error -std=c++0x pointer_error.cpp

pointer_error.cpp: In function 'int*
set_magic_number_a()':
pointer_error.cpp:6:7: warning: address of local
variable 'a' returned [enabled by default]
```

- This is a common mistake - - - accidentally referring to addresses that have gone "out of scope"

CALLOC()

```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate "C"lear memory on the heap
- Calloc wipes memory in advance of use…
- `size_t num` : number of blocks to allocate
- `size_t size` : size of each block(in bytes)

- Calloc() prevents…

```
char *dest = malloc(20);
printf("dest string=%s\n", dest);

dest string=��F
```


SYSTEM CALLS

- brk(), sbrk()
  - Used to change data segment size (the end of the heap)
  - Don't use these
- Mmap(), munmap()
  - Can be used to create an extra independent "heap" of memory for a user program
  - See man page

REALLOC()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Resize an existing memory allocation
- Returned pointer may be same address, or a new address
  - New if memory allocation must move
- `void *ptr`: Pointer to memory block allocated with malloc, calloc, or realloc
- `size_t size`: New size for the memory block(in bytes)

QUESTIONS

DOUBLE FREE

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```

- Can't deallocate twice
- Second call core dumps