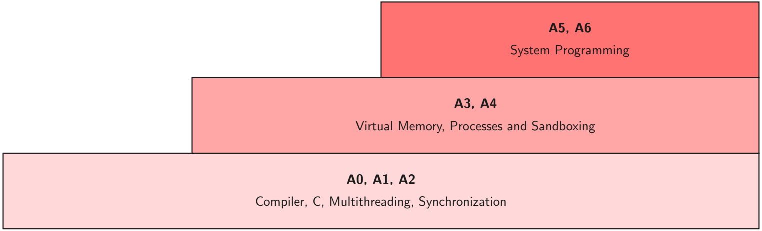
System Level Programming

Daniel Gruss

2023-04-23

Course Overview



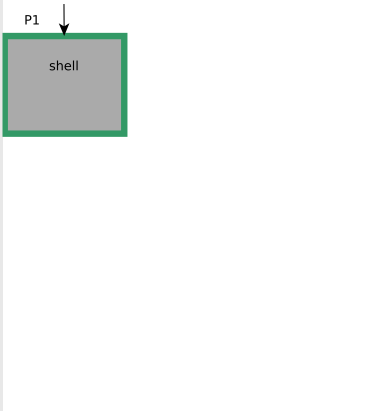
A4 - Interprocess Communication

Shared Memory

Code

```
    //shell stuff
```

Image

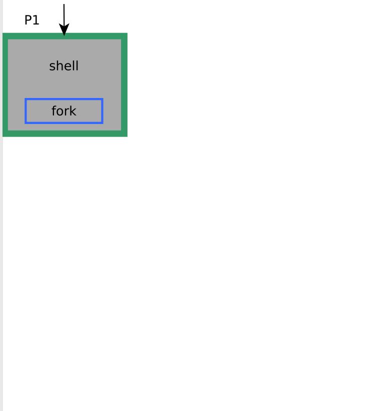


Shared Memory

Code

```
//shell stuff

pid_t pid = fork();
```

Image

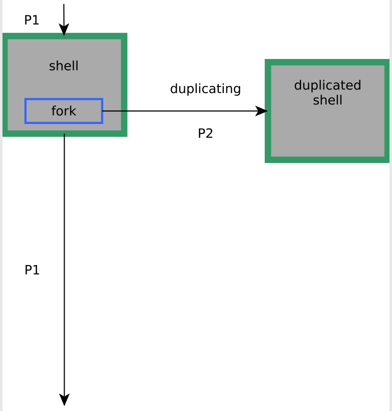


Shared Memory

Code

```
    //shell stuff

        pid_t pid = fork();
        if(pid == 0)
```

Image

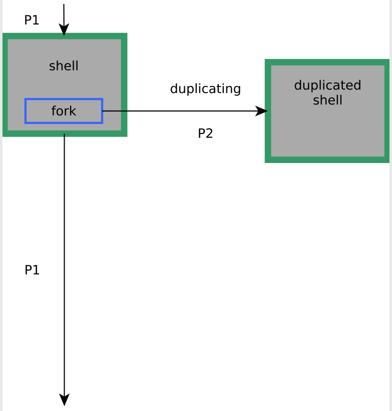


Shared Memory

Code

```c
    //shell stuff

        pid_t pid = fork();
        if(pid == 0)
        {
                const char* args[] = {"~/
                    "};
        }
        else
        {
                //do further shell stuff
        }
```

Image

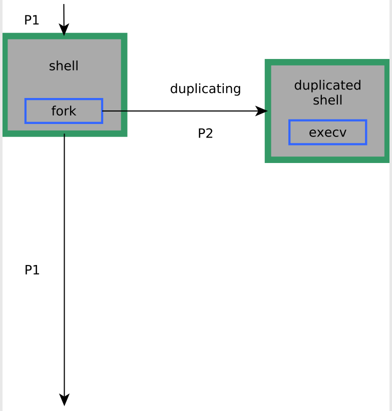


Shared Memory

Code

```
//shell stuff

pid_t pid = fork();
if(pid == 0)
{
  const char* args[] = {"~/"};
  execv("/bin/ls", args);
}
else
{
  //do further shell stuff
}
```

Image

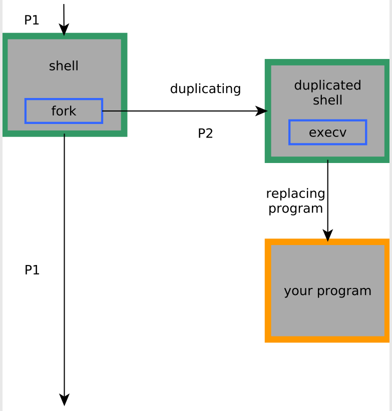


Shared Memory

Code

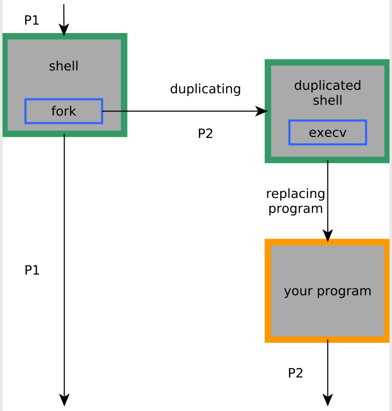
```c
//shell stuff

pid_t pid = fork();
if(pid == 0)
{
  const char* args[] = {"~/"};
  execv("/bin/ls", args);
}
else
{
  //do further shell stuff
}
```

Image



Shared Memory

Image

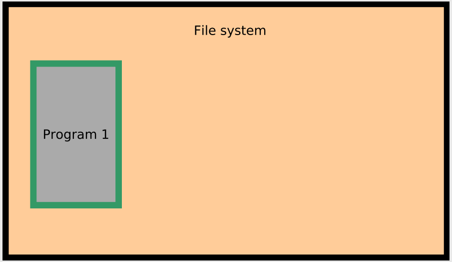


Shared Memory

Code

```
/* just the start of the main */
```

Image

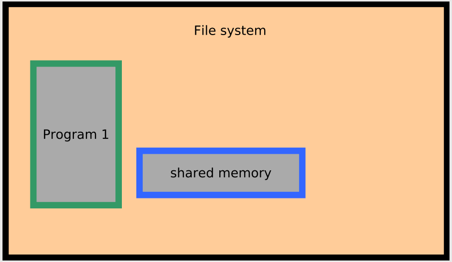


Shared Memory

Code

```c
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644);
```

Image

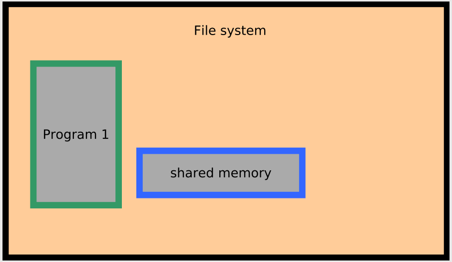


Shared Memory

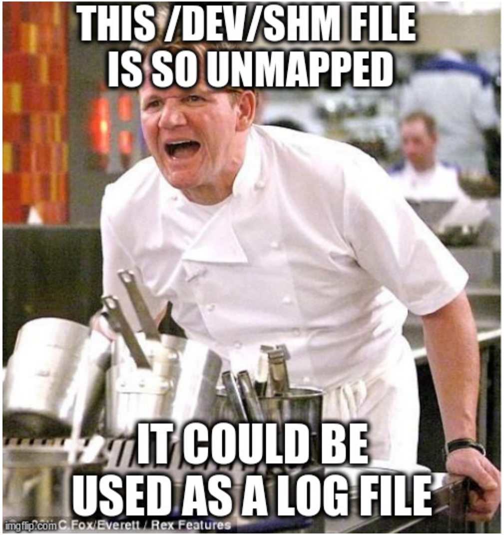
Code

```
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644);

/* enlarge the shared memory object */
ftruncate(fd, 1000);
```

Image



Shared Memory



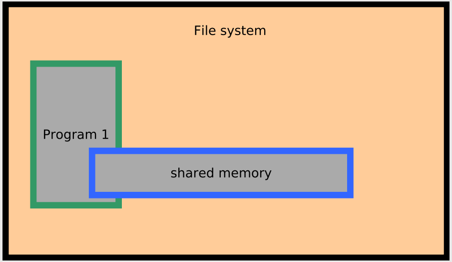
Shared Memory

Shared Memory

Code

```c
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644);

/* enlarge the shared memory object */
ftruncate(fd, 1000);

/* now map the shared object */
char* ptr = (char*) mmap(NULL, 1000,
    PROT_READ | PROT_WRITE, MAP_SHARED, fd,
    0);
```

Image

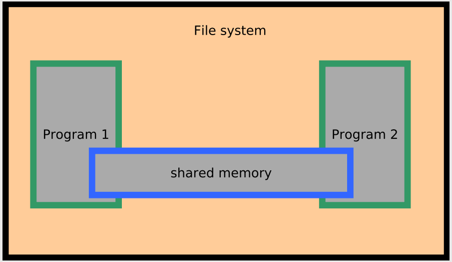


Shared Memory

Code

```c
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644);

/* enlarge the shared memory object */
ftruncate(fd, 1000);

/* now map the shared object */
char* ptr = (char*) mmap(NULL, 1000,
    PROT_READ | PROT_WRITE, MAP_SHARED, fd,
    0);

/* fork the process */
pid_t pid = fork();
```

Image



A5 - malloc/free - Task Description

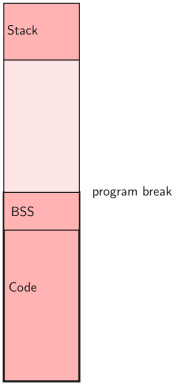
- Write your own implementation of malloc/free
- `void *malloc(size_t size);`
- `void free(void *ptr);`
- Write them in C++ with classes!!
- The malloc/free functions manage the Heap area and give a program the ability to request memory areas of a given size and free those areas if they are not needed anymore
- You can reuse this code in OS A2

A5 - malloc/free - Introduction

```
int inputsize = 200;
int* buffer = malloc(inputsize*sizeof(int));
memcpy(buffer,input,inputsize)
//do something very important
free(buffer);
```

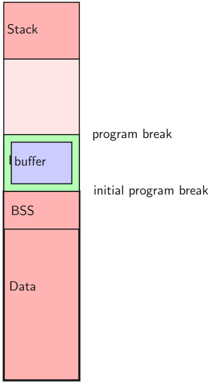
- Where in the memory is this buffer area?
- How can it be increased/decreased at runtime?

Memory of a process



- Virtual Memory Space
- Code: Segment for the binary code
- BSS: part of Data Segment;
  global/static variables with known size at compiletime
- Program break shows end of Data Segment
- Program break can be increased/decerased

Heap



- Program break increased
- Heap = between end of BSS Segment and program break
- Memory addresses below program break can be used by the program
- Let's use this area for our buffer

brk/sbrk

- OS offers syscalls brk and sbrk to change the program break of the own process
- `void* sbrk(intptr_t increment);`
- `sbrk(inc)` increments the break by inc bytes
- Returns the address of the previous program break
- `sbrk(0)` returns current location of the break

Why not just:

```c
void *malloc(size_t size){
  return sbrk(size)
}
```

Because ...

```c
while(1){
  void* t = malloc(100);
  //do anything
  free(t);
}
```

It's not that easy, but not much harder

What do we want

- Efficient usage of memory
- Reuse of freed memory areas
- Avoid fragmentation of Heap Segment

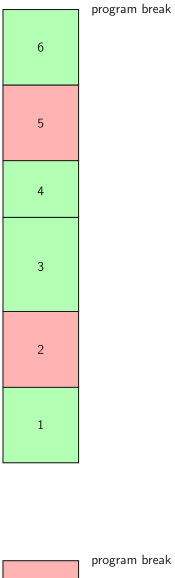
How?

- Decrease program break if possible
- Merge freed memory areas
- Split large free memory areas to the needed size

A5 - malloc/free

Decrease program break if possible

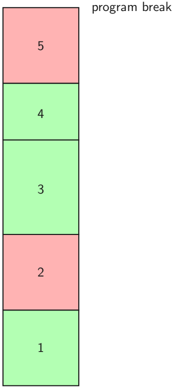
- If there is free memory area just below the break
- Size of this memory area



A5 - malloc/free

Merge free memory areas

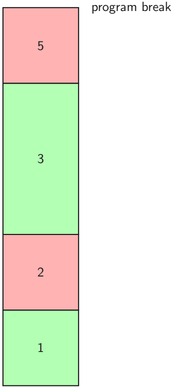
- Only possible to merge with next or previous area
- We have to know the size, location and state of the areas



A5 - malloc/free

Reuse free areas/split large free memory areas to the needed size

- Search for a free memory area larger/equal than needed size
- Split to right size
- State of all memory areas and their location
- Size of the area to split



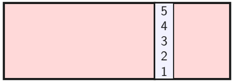
What do we have to know about the memory areas

- Is the memory area free?
- How large is the memory area?
- Location of the memory area?

Think about a structure which allows you to organise the Heap Segment

Errors you should detect

- Double free
- Out of memory
  - sbrk returns 0
- Buffer overflow / memory corruption
  - Special value at begin of every memory area
  - Check if first word == special value



A5 - Task Summary

- Consider a structure to organize the memory areas
- Decrease program break if possible
- Avoid heap fragmentation
  - Merge free neighboring memory areas
  - Split large free memory areas to the needed size
- Detect overflows, double frees and out of mem
- Your implementation has to be POSIX compliant (manpage)

A5 - Hints

- Pointer arithmetics: `int* p; p+5;` – addr in `p` is increased by `5*sizeof(int)`
- How many bytes does a pointer need? use typedef mempos in malloc.h
- Double-Linked-List of memory areas
- Mempos address = "valid addr"; `int* i = (int*) address; *i = 100;`
- Be careful to test the right malloc implementation ;)

Down the rabbit hole: Underneath x86 Linux C programs

General structure

How does a C program "work"?

- Control starts at main
- Certain functions pass control to operating system, e.g. printf has the OS write something to "standard output"
- When main returns, the program terminates gracefully
- Certain errors kill the program forcefully, e.g. with a "Segmentation fault"

C Standard Library

How does printf "work"?

- Format string parsing, argument extraction, construct final string $\rightarrow$ trivial
- write final string to stdout filedescriptor
- write, in turn, makes a **system call** (syscall) with the appropriate syscall number
- The syscall transfers control to the operating system, which executes the write on the user program's behalf

C program start and termination

How is main called and return handled?

- Operating system does not actually run main
- Execution starts at the entry point address, where the standard library start function is located
- Initializes standard library, obtains program arguments, calls main
- After main, `exit` is called with the return value of main
- `exit` performs a syscall that terminates the program gracefully

Odds and ends

- For C++ programs, initialization and deinitialization of global objects also has to happen before/after main, respectively
- Disassembly of a program: `objdump -d`
- Some interesting info (entry point address, sections, ...): `readelf -a`
- What symbols are visible in your program: `nm`
- Which shared libraries are loaded: `ldd`

What's in a C program?

- Compiler produces object files for your code
- Linker takes your object files and links it with standard library objects
- `gcc -nostdlib` → "nothing" works anymore
- Provide your own standard library!

Header files and objects

- #include <stdio.h> still works, despite `-nostdlib`!
- Yes, but linking fails: `undefined reference to 'printf'`
- When compiling printf (...) , the compiler produces something like: call printf
- The linker takes all object files, assigns ("arbitrary") addresses to all functions
- Then, all references to printf are replaced by that address

Virtual Memory

Why can the linker assign static addresses to symbols? Virtual Memory!

You'll learn about that in OS ;)

32bit Calling conventions

Brief overview

- cdecl: "Standard" calling convention gcc uses for C programs
- syscall (not the OS/2 one): How syscalls are called
- fastcall, thiscall, pascal, ...: For other operating systems, languages, compilers, ...

We will now look at cdecl and syscall.

cdecl

How do 32bit functions work?

- There is a stack somewhere in memory
- The register `esp` points to the top of the stack
- Assembly instructions `push` and `pop` use and modify `esp`
- Another register, `ebp` points to the beginning of the current "stack frame"
- Each call of each function opens a new "stack frame", i.e. `ebp` is moved to the top of the stack
- How to restore the old ebp when the function returns? Save it on the stack!
- Local variables and parameters are always referenced relative to `ebp`!

Example: function

Consider:

```c
int myfunc(int i)
{
    return 2*i;
}
```

This produces the following assembly:

```
pushl %ebp
movl %esp, %ebp

movl 8(%ebp), %eax
addl %eax, %eax

popl %ebp
ret
```

cdecl

How does the call work?

- Function refers to parameters on the stack
- So we will have to push them on the stack (right to left)
- `call` function
- Return value is then in `eax`
- Remove parameters from stack again ("caller cleanup")
- Except for floating point values, but we won't cover that here

Example: call

```
myfunc(1);
```

This produces the following assembly:

```
subl  $4, %esp
movl  $1, (%esp)
call  myfunc
addl  $0, %esp
```

syscall

How does a system call work?

- Put all parameters into registers
- Request an interrupt
- The interrupt handler will run in kernel mode and use values from registers
- Return value is then again in `eax`
- What happens in kernel mode? You will find out in **Operating Systems**!