Operating systems. Lecture 9

Michał Goliński

2018-11-27

Introduction

Recall

- Reading and writing wiles in the C/C++ standard libraries
- System calls managing processes (`fork`, `exec` etc.)

Plan for today

- `fork` – one more example
- allocating memory – system and library calls
- signals

`fork` again

One more example

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    printf("1");
    fork();
    printf("2");
    return 0;
}
```

Allocating memory with system calls

Heap and stack

We already talked about virtual memory. Recall: the operating system gives memory pages to a running process by adding them in the process's page table. The address space has two large areas: heap (which occupies the low memory addresses and grows upwards when requested) and stack (which occupies high addresses and grows automatically downwards as needed).

`brk` and `sbrk`

The `brk` system call allows to grow the upper limit of the heap – so-called system break. We will not concern ourselves with the precise definition as we are not supposed to use it directly. Heap is contiguous in virtual address space. It is possible (although unlikely on 64-bits) that the call will fail even though free memory is still available.

`sbrk` allows to move the program break relatively to its current position.

Mapping files – `mmap`

The `mmap` system call allows to map files into process address space. Mapping is like a window through which file contents can be viewed (or changed). Apart from `mmap` no other calls `read` or `write`) are not necessary. Contents of the memory will be written to the file on `munmap`. The call has the following signature:

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```

The possible parameters values differ heavily between operating systems.

Parameters

- `addr` is a hint where in the address space the mapping should occur. The operating system can disregard this hint.
- `len` is the amount of memory that should be mapped.
- `prot` is the permission for the newly created region. It is either `PROT_NONE` or a logical combination of `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`.

Parameters cont.

- `flags` is either `MAP_SHARED` or `MAP_PRIVATE`. A shared map changes the underlying object and the changes are visible to others. A change in a private mapping does not change the underlying file and changes are not visible to other processes. There are other multiple flags differing by the OS.

Parameters cont.

- `fildes` is a descriptor of an open file that should be mapped. It can be -1 if no specific underlying file is requested.
- `off` is the offset in the file at which the mapping should start. For performance it is required to be a multiple of the page size.

`mmap` and `fork`

Memory mappings are preserved through `fork`. This means that a map with `MAP_SHARED` creates a memory region that can be used as an interprocess communication. `MAP_PRIVATE` uses the copy-on-write strategy.

`MAP_ANONYMOUS`

If the `flags` contains `MAP_ANONYMOUS` then no specific file should be mapped. Rather, a new memory area should be created in the process's virtual address space. In practice this gives a way to extend the address space – allocate new memory for a process.

Example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdint.h>

uint64_t hash(uint64_t x) {
    x = (x ^ (x >> 30)) * UINT64_C(0xbf58476d1ce4e5b9);
    x = (x ^ (x >> 27)) * UINT64_C(0x94d049bb133111eb);
    x = x ^ (x >> 31);
    return x;
}

int main(int argc, char *argv[]) {
    int fd = open("/tmp/mmap.bin", O_RDWR | O_CREAT | O_TRUNC,
     ↪  S_IRWXU);

    int size = 1024*1024;
    size_t len = sizeof(uint64_t)*size;
    lseek(fd, len - 1, SEEK_SET);
    write(fd, "", 1);

    uint64_t *data = (uint64_t *) mmap(0, len, PROT_READ |
     ↪  PROT_WRITE, MAP_SHARED, fd, 0);
```

```
    data[size-1] = hash(size-1);
    for (int i = size - 2; i >= 0; --i) {
        data[i] = hash(data[i+1]);
    }

    munmap(data, len);
    close(fd);
    return 0;
}
```

`munmap`

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

This call unmaps the indicated memory region. All subsequent references to this memory area give a segmentation fault.

ASLR

It would seem natural to assume that a program memory looks the same each time the process is started. This is usually not the case: for security reasons the address space is randomized on each run of the program. This is called ASLR – address space layout randomization.

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

int main(void) {
    printf("%p\n", sbrk(0));
    return 0;
}
```

Allocating memory with library calls

Rationale

Details of these system calls differ between platforms (even between different versions of Linux) and hence gives a potentially fragile code. User programs, barring special needs, should not use the system calls directly. Every language gives better tools for memory allocation.

C – `malloc`

The C standard library gives us the `malloc` function:

```
#include <stdlib.h>
void *malloc(size_t size);
```

When called, `malloc` reserves *size* bytes of memory and returns a pointer to the newly allocated region. This does not necessarily mean that it performs a system call: the library request memory from the system in larger chunks and gives them to the application in smaller amounts (mechanism is similar to buffering).

In principle small amounts tend to be satisfied with `brk` calls and large amounts of memory with `mmap` calls.

`realloc`

The `realloc` function is closely related with `malloc`:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

It takes an existing region that was allocated by `malloc` (referenced by the pointer `ptr`) and changes it size to *size* bytes. It **may need to change** the region's place in memory, but then the relevant data will be copied.

`free`

It is the process's duty to give the memory back to the OS if it is not longer needed. To tell the library that a region allocated with `malloc` *et el.* is no longer needed and can be reused or given back to the OS, one should call `free`:

```
#include <stdlib.h>
void free(void *ptr);
```

Failing to call `free` is safe, but leads to a *memory leak*.

Memory leak

When a program fails to call `free` but a memory region is no longer used we are just wasting memory. If the pointer that points to the memory region is lost (overwritten etc.) without freeing it before that, then we say that a *memory leak* has occurred. Leaking large amounts of memory is problematic as the only way to regain the memory is to restart the program. Memory leaks are hard to find as often the `malloc` and `free` have to be done in very distant parts of code.

Tools like `valgrind` can help find memory leaks.

Automatic heap management

The necessity to `free` unused memory regions is a heavy burden on the programmer if the application is not so small. Many so-called managed languages (Java, C#, Python) use a *garbage collector* that looks for unused memory areas and deallocates them automatically. This sacrifices some performance for programmer's convenience. Garbage collectors are also available for C.

C++ – `new`

In C++ one uses the `new` operator that allocates objects in the heap. This is a bit more than `malloc` does: not only is the memory region reserved, but also the **constructor** for the new object needs to be called. This is a special functions that bootstraps the object into a usable state. Calling `new` might call `malloc` under the hood, but not necessarily. The operator returns a pointer to the newly created object.

One can also allocate a *dynamic array* of objects using `new`.

`delete`

To deallocate an object created by `new` one should use the `delete` operator passing the pointer to an object. If an array was allocated using new, then it has to be deallocated using `delete[]`.

Example (C)

We will write a program that asks for numbers and prints them back in the reverse order.

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

int main() {
    int n;
    scanf("%d", &n);
    int *p = calloc(n, sizeof(int));
    for (int i = 0; i < n; ++i)
        scanf("%d", p+i);
    for (int i = n-1; i >= 0; --i)
        printf("%d\n", p[i]);
    free(p);
    return 0;
}
```

Example (C++)

The same program in C++ using new:

```
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

int main() {
    int n;
    cin >> n;
    int *p = new int[n];
    for (int i = 0; i < n; ++i)
        cin >> p[i];
    for (int i = n-1; i >= 0; --i)
        cout << p[i] << endl;
    delete[] p;
    return 0;
}
```

Example (C++ – vector)

The C++ library contains also the STL – Standard Template Library. Using this library we can write or program without explicit memory management:

```
#include <iostream>
#include <vector>

using std::cin;
using std::cout;
using std::endl;
using std::vector;

int main() {
    int n;
    cin >> n;
    vector<int> p;
    for (int i = 0; i < n; ++i) {
        int m;
        cin >> m;
        p.push_back(m);
    }
    for (int i = n-1; i >= 0; --i)
        cout << p[i] << endl;
```

```
    return 0;
}
```

Example (C++ – full STL)

The C++ library contains also the STL – Standard Template Library. Using this library we can write or program without explicit memory management:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using std::cin;
using std::cout;
using std::endl;
using std::vector;
using std::copy_n;
using std::istream_iterator;
using std::ostream_iterator;
using std::back_inserter;

int main() {
    int n;
    cin >> n;
    vector<int> p;
    std::copy_n(std::istream_iterator<int>(cin), n,
 ↪  std::back_inserter(p));
    std::copy(p.crbegin(), p.crend(),
 ↪  std::ostream_iterator<int>(cout,"\n"));
    return 0;
}
```

Signals

Introduction

Signals are a simple IPC (inter-process communication) mechanism present in all the UNIX systems. Signals are represented by small integers. When a signal is delivered to a process the process normal execution is paused and a designated *signal handler* is run. For most signals the process can change the handler, but for some this is not possible.

Most important signals

Some signals and their default action:

- `TERM` – kills the process
- `KILL` – kills the process
- `INT` – keyboard interrupt (Ctrl+C, kills the process
- `STOP` – stop the process (Ctrl+Z)
- `CONT` – continue the stopped process
- `SEGV` – segmentation fault, kills the process

All but `KILL` and `STOP` can be ignored or caught and handled.

`kill`

The `kill` command sends a given signal to the process with a given ID. By default the `TERM` signal is sent for which the default action is to end the process, but this can be overridden.

`killall`

The `killall` command sends a signal to all processes with a given name. Otherwise works like `kill`.

The `signal` function

To change how a signal is handled we can use the (now discouraged) `signal` function:

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

This admittedly scary looking prototype says that signal is function that takes 2 arguments:

- an `int`
- a pointer to a function that takes `int` as an argument and returns nothing

The return value is a pointer to a function taking `int` as an argument and returning nothing.

Arguments and return value

The first argument is the signal which handler we wish to change. The second argument is the new handler or `SIG_DFL` (default behaviour for the signal) or `SIG_IGN` (ignore the signal).

The function returns a pointer to the previous signal handler (pointer to a function).

Signal handler

A signal handler is function that takes an `int` argument and returns nothing. It will have the signal number passed as an argument, this way a single handler can handle multiple signals.

Reentrancy

It is possible that a signal handler is interrupted by another signal. This means that he function can be interrupted and its execution can begin from the start. If a function behaves properly in such a situation it is called reentrant.

All signal handlers need to be programmed in a reentrant way.

Example

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sighandler(int signum) {
    fflush(stdout);
    printf("received signal %d\n", signum);
}

int main(void) {
  signal(SIGINT, sighandler);
  for(;;) {
      sleep(3600);
  }
  return 0;
}
```