

Lecture 2

OS external Concepts and Interface

In the last lecture we learned that multiple programs are loaded in the computer at the same time, but the computer run one of them at a time.

We look at issues arised from an application programmer's perspective.

References:

- [OSC] 3.2 (OS services), 4.5 (IPC).
- [ULK] pp. 7–19 (OS concepts), 28–33 (Process/Memory management), 249–254 (Signals), 524–528 (Pipes), 533–534 (FIFOs).
- Unix man pages: fork, waitpid, exit, kill, execve, open, read, write, ioctl, brk, mmap, sigaction; Solaris man pages: shm_open, shm_unlink.

POS(0230A)

POS(0230A)-2.1

Program loading

Programs loading:

1. **Read the program** from the disk to the memory.
2. Set up **parameters** and **environment variables**.
3. **Jump (Pass control)** to the entry point of the program.

Unix: Done by *execXX* system call. (Try “man *exec1*”). E.g.,

```
exec1("/usr/bin/ls", "ls", "-l", NULL);
```

load */usr/bin/ls*, give it two arguments *ls* (the program name) and *-l* (its parameters). *Won't return* unless there is an error.

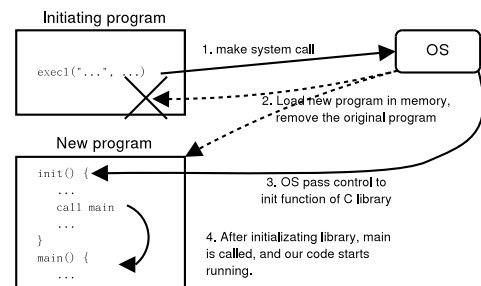
Entry point: the *init()* function. In C: it does some initialization (e.g., open files) and then call *main()* with the parameters and env-var list.

We don't set env-var here. To setup env-var, one would use *execle()* instead.

POS(0230A)-2.2

Schematics of Unix program loading

The whole process is something like this:



The old program disappear altogether once a new program is loaded. We usually say that the program load into the memory of the old one, although in fact old memory is freed and new memory is allocated.

POS(0230A)-2.3

Process creation

But what if we want the old program to still run? Then we are **creating** a new process—after that, 2 processes run.

Unix: process creation is done by a separate call, *fork()*. Intuitively:

1. A new virtual machine (i.e., process) is created. It is called the **child** process (the original one called the **parent**). A process identifier (PID) is assigned for the child.

Each process has a unique PID, which is a small integer identifying it.

2. All properties of the calling process is copied over to the new process, so that they seem to run the same program.

3. For the parent, return value is set to the PID of the child. For the child, return value is set to 0.

So the program can watch the return value to know whether it is the child.

4. Both processes continue to run (probably at different “speed!”).

POS(0230A)-2.4

Waiting for a child

- Suppose we want to execute a new program, and after the program finishes we want to know whether the program successfully done its work.

- Usually we want the parent program not to proceed until that happens.

- In other words we want to use *waitpid()*:

1. Stop the process until a particular child process finishes;

2. Get the “exit status” of *pid*, which is the return value of the *main()* function.

- If we are interested in all children, we use *wait()* instead, which completes if *any* child is finished and return the PID of the finished child.

POS(0230A)-2.5

Example program

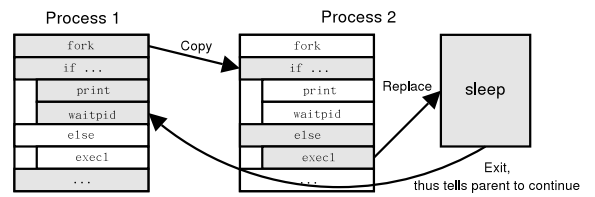
```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <sys/types.h>
int main() {
    pid_t p = fork();
    if (p != 0) { // Parent, p = child id
        int retval;
        printf("Started process. Waiting...\n");
        waitpid(p, &retval, 0);
        printf("Done, returned %d.\n", retval);
    } else // Child: fork returned 0
        execl("/bin/sleep", "sleep", "5", NULL);
    return 0;
}
```

To understand this fully, try to change the program and understand the result!

Memory is copied: new process can use all data of the old one, but any modification will not be seen in the old process.

Schematics of Fork/Exec sequence

It gets messy, since fork does not follow "normal" logic of a program:



After fork, **both** processes run **concurrently**. However, the **instructions executed can be different**, because the return value of `fork()` is different. The concurrency is implemented by switching CPU from time to time.

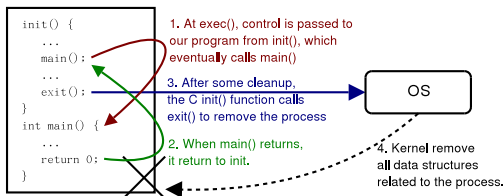
Process 1 calls `waitpid()`, **suspending itself** until Process 2 terminates. I.e., until sleep finishes. After `execl()`, process 2 is not running the original program.

Process removal

In Unix, a process is removed in **two scenarios**:

- The program executes `exit()`, telling the OS that it is done.
- The kernel or another process sends it a **signal** that remove it.

The typical "return from main" actually happens like this:



So even normal exit results in a kernel call.

The process tree

- When process is created by `fork()`, a child is created.
- So all processes form a "family tree". Each process has a parent. A process can get the ID of its parent by `getppid()`.
- A parent should wait for its children and get its exit status. Otherwise the PID of the children remain unusable and some resources is kept by the OS, just in case the parent suddenly decide to ask for the exit status. The dead process is said to be **zombie**.
- There is of course an exception: the root of family tree has no parent. That is special anyway, because that is always the first process of the system not created using `fork()`.
- What if a parent `exit()`? In Unix, the process is **reparented** to a specific process, usually of PID 0 or 1, to keep the tree structure. After that, the OS won't expect anybody to `wait()` for its children.

Memory management

- After `execl()` is executed, the **old memory** of the process is **deallocated**, and **some new memory is allocated**.
For the program code, global variables and stack ("auto" variables).
- Memory is organized into disjoint **segments**, so that they can grow independently. Program code and global variables are in the **text segment**, while stack is in the **stack segment**.
- Processes can call `brk()` to **enlarge or shrink** the text segment. Size of stack segment is automatically adjusted without programmer's help.
- It can also call `mmap()` to allocate **new memory segments**. The `mmap()` call also allow disk files to be read like memory.
When allocating large objects, better to use `mmap()` than `brk()`.
- Most program would simply use `malloc()` or `new` to allocate memory, which eventually call `brk()` or `mmap()`.

Example

Let's allocate some memory and see the addresses. Similar address means the same memory segment.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/fcntl.h>

// Allocated on program loading
int size;

int main() {
    // Allocated in stack
    int i, fd;
    char *mem, *map;

    // See their addresses
    size = getpagesize();
    printf("size at %p\n", &size);
    printf("i at %p\n", &i);

    // New memory segment
    mem = mmap(0, size, PROT_READ|PROT_WRITE,
               MAP_PRIVATE|MAP_ANON, 0, 0);
    printf("mem at %p\n", mem);
    for (i = 0; i < size; ++i)
        mem[i] = 'a';
    printf("memory filled with 'a'\n");

    // Reading file like memory
    fd = open("/etc/passwd", O_RDONLY);
    map = mmap(0, size, PROT_READ,
               MAP_PRIVATE, fd, 0);
    printf("map at %p\n", map);
    for (i = 0; i < size; ++i)
        printf("%c", map[i]);
    printf("\n");
    return 0;
}
```

Virtual vs. physical memory

- Suppose we run the program of the last slide **twice** at the **same** time. This results in two different **processes**.
- The OS allocates two different parts of memory, one for each of them. The 2 processes thus get 2 sets of **addresses** for the variables.
- Or is it? If we really the program twice, we will notice that **the addresses allocated are exactly the same!**
- Why? Remember that a process is a **virtual** machine, so memory of processes are also **virtual** rather than “real”!
- So the memory addresses of the two processes are **unrelated**, even though they have the same addresses.
- When a “virtual CPU” runs, the virtual addresses are **translated** to **physical** memory addresses, which is different for the two processes.
Would it be slow? No, but... let's not talk about implementation yet.

POS(0230A)-2.12

Shared memory

- We have seen that normally, memory of processes are **independent**: even if two virtual memory addresses in two different processes are the same, they are actually different memory.
- But what if I **really** want processes to share some memory, e.g. for **Communication** among processes? (IPC, **I**nter-**P**rocess **C**ommunication.)
- A process can ask for memory that is **shared** with other processes. If two processes share the same memory, they can talk to each other.
NB: they need not share the same virtual address.
- Unix: you can make a **shared mmap** to achieve it.
- The shared mapping may or may not be backed by a disk file .
How to address it without a disk file? It is sometimes okay that it isn't addressed at all! See example.

POS(0230A)-2.13

Example code

Here the parent and child reads and writes the memory at the same time.

```
#include <sys/types.h>           // Fork out new process
#include <sys/mman.h>           char *id = "parent";
#include <unistd.h>             pid_t pid = fork();
#include <stdio.h>              if (pid
                                id = "child";

int main() {
    // Allocated shared memory
    int size = getpagesize(), i;
    // Note the MAP_SHARED
    char *map = mmap(0, size,
                    PROT_READ|PROT_WRITE,
                    MAP_SHARED|MAP_ANON,
                    0, 0);
    // Both processes writes by "++"
    for (i = 0; i < 5; ++i) {
        printf("%s: %d\n", id, ++map[0]);
        // Sleep to give the other process
        // a chance to run
        sleep(1);
    }
}
```

What if the processes are not parent and child, so we can't use the `fork()` trick to get the same memory? Then we would open a file that is not a disk file for mapping. See the man page of `shm_open()`.

POS(0230A)-2.14

Filesystem

- Computers have secondary storage to keep data permanently.
- The OS organizes these device into **filesystems** to make access more convenient. This involves grouping data into **files** and **directories**, and gives names to each so that one can easily address them.
The filesystem serves another purpose, as we will see.
- Processes can ask the OS to access files for them:
 - `open()`, `close()`: start and end accessing a file . A “file descriptor” is returned so that you can access the files .
 - `read()`, `write()`: reading and writing data.
 - `rename()`, `unlink()`: changing names and removing file .
 - `lseek()`: allow random rather than sequential access.
 - ...

POS(0230A)-2.15

Example code

```
#include <stdio.h>
#include <unistd.h>
#include <sys/fcntl.h>
int main() {
    char c[1024];
    int fd = open("/etc/passwd", O_RDONLY);
    if (fd == -1) { /* handle error */ }
    int size = read(fd, c, 1024), i;
    /* print the first 1024 chars of the file */
    for (i = 0; i < size; ++i) printf("%c", c[i]);
    close(fd);
    return 0;
}
```

`fd` is the file descriptor. It is a **small integer** to identify the opened file .

In real programs, we usually use `ofstream` (C++) or `FILE*` (C) instead to be more portable. Each of them stores a file descriptor internally.

POS(0230A)-2.16

Manipulating file descriptors

- File descriptor 0, 1 and 2 are usually used for **standard input, standard output and standard error** respectively.
I.e., `fd[0]` points to the file for standard input, etc.
- Not significant to the kernel, but for the C library, `printf()` prints to `fd 1` and `scanf()` read `fd 0`; the symbols `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO` are defined to be 0, 1 and 2; and the file streams `stdin`, `stdout` and `stderr` are predefined to use those 3 fds.
- There are system calls `dup()` and `dup2()` that allow you to **duplicate a file descriptor**, i.e., create a new file table entry that points to an already opened file entry.
- This is handy if you want to redirect the input and output of `scanf()` and `printf()`. Indeed, shell redirections (`<` and `>`) are implemented by `dup2()`.

POS(0230A)-2.17

Accessing I/O devices

Unix allows users to **do I/O like files**. E.g., the sound device is available at the file `/dev/dsp`. To read sound for 1 second:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/fcntl.h>
int main() {
    char buf[8000];
    int fd = open("/dev/sound/dsp", O_RDONLY);
    if (fd == -1) { printf("Error\n"); exit(1); /* handle error */ }
    read(fd, buf, 8000);
    /* do whatever we want to buf */
    close(fd);
    return 0;
}
```

Indeed similar to file reading, right?

The read function **blocks**, i.e., stops the running process until it completes. Other processes continue to use the CPU.

POS(0230A)-2.18

Using the device file

- Device files are very similar to regular files: you can use `open()`, `read()`, `write()`, `close()`, and perhaps `seek()` on them.
- But we still need more operations... basically, to **control** the device.
- E.g., if we want to read data from the `/dev/dsp` device (i.e., reading sound samples), we want to set the sampling rate, etc.
- So there is one more system call: `ioctl()`. It is a “catch-all” system call: everything not done by regular calls are done using `ioctl()`.
- **Each device driver defines a set of `ioctl` operations it supports**, and assign numbers to them. E.g., the CDROM device driver recognizes the operation `0x5301` to pause a CD, `0x5309` to eject a CD, etc. See `ioctl_list(2)` for a complete list.

POS(0230A)-2.19

Example: ejecting a CD

The following program ejects a CD from the CD-ROM drive.

```
#include <sys/fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main() {
    int cd_fd = open("/dev/cdrom", O_RDONLY);
    if (cd_fd == -1)
        exit(1);
    if (ioctl(cd_fd, CDROMEJECT) == -1) // This is 0x5309
        exit(2);
}
```

But this is ugly... it is difficult to know the list of available `ioctls`, and difficult to do the `ioctl` from command line.

Linux started switching to use the filesystem instead. Imagine you can eject a CD by just typing `echo 1 > /dev/cdrom/eject`, or know what operations it support by just typing `ls /dev/cdrom/`.

POS(0230A)-2.20

Pipes

There is another way to do IPC... a process can **send** a message, while another process **receive** the message—**Message Passing**.

Pipes is such a mechanism, again using file descriptors and `read()` and `write()` to transfer messages.

- Pipes have two “ends”: a **read** end and a **write** end. Things written to the write end will be sent to the read end.
- Pipes are accessed through normal Unix `read()` and `write()` system calls, so some **file descriptors** are connected to the pipe.
- Pipes have a buffer of `PIPE_BUF` (usually, 4096) bytes. If buffer is exceeded, the calling process must wait.
- Messages smaller than `PIPE_BUF` are guaranteed to arrive without splitting. But messages do merge.

POS(0230A)-2.21

Getting hold of pipes

Whenever we run something like “`ls | less`”, we use pipes. The two processes for `ls` and `less` communicate through pipes.

Now let's see how to use them in our programs.

- A pipe can be created by the `pipe()` system call. This results into **two** new file descriptors, attached to the read and write ends. How to return 2 fds? The user must give `pipe()` an array of 2 integers to store them.

There is no way to refer to such a pipe if you don't already hold an fd. So it is called **unnamed pipes**.

- Another way: to create a file system entry that **corresponds** to a pipe, and use normal `open()` calls to get a file descriptor. The entry is created with the `mknod` command or `mknod()` system call.

We call them **named pipes**, or **FIFOs**.

POS(0230A)-2.22

Pipes example

To achieve something like “`ls | less`” in our program:

```
#include <unistd.h>
#include <sys/types.h>
int main() {
    pid_t p;
    int filedes[2]; /* fd's of the pipes, 0 is in, 1 is out */
    pipe(filedes); /* create unnamed pipe */
    p = fork();
    if (p == 0) { /* child for a.out */
        close(filedes[0]); /* close in part */
        dup2(filedes[1], 1); /* replace stdout by pipe */
        exec("/bin/ls", "ls", NULL);
    } else { /* parent for less */
        close(filedes[1]); /* close out part */
        dup2(filedes[0], 0); /* replace stdin by pipe */
        exec("/usr/bin/less", "less", NULL);
    }
}
```

POS(0230A)-2.23

FIFO example

In a shell script...

```
mknod testpipe p
ls -l > testpipe &
less < testpipe
rm testpipe
```

Note: the & character asks the shell to run ls in "background", so that you can continue to type the less command.

In a C program:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
    pid_t p;
    int fd;
    mknod("testpipe", 0666, S_IFIFO);
    p = fork();
    if (p == 0) {
        fd = open("testpipe", O_WRONLY);
        dup2(fd, 1);
        execl("/bin/ls", "ls", NULL);
    } else {
        fd = open("testpipe", O_RDONLY);
        dup2(fd, 0);
        execl("/usr/bin/less", "less", NULL);
    }
}
```

POS(0230A)-2.24

Signals: asynchronous IPC

- So far, the IPC mechanisms we know are **synchronous**, i.e., the receiver has to run instruction to **check** for and **perform** the communication. E.g., read the memory, receive the message, etc.
- Sometimes this is not the best. E.g., the communication **rarely** occurs, or it needs to **wait** for a real long time for communication to happen. Usual example: you probably don't want having to explicitly wait the user to press Control-C or Control-Z in your program to stop or suspend it.
- The CPU also has a similar problem: it doesn't want having to repeatedly wait for I/O, so it has something called "**interrupts**" for devices to communicate **asynchronously** with the CPU.
- In our program we cannot use interrupts. But we have something very similar: **signals**. We can consider them **virtual interrupts**. Normal interrupts are handled by the OS kernel, processes can't work on them.
- Like interrupts, signals are **numbered**, to carry different meaning.

POS(0230A)-2.25

All the different signals

Some system generated signals:

Number	Symbol	Event
2	SIGINT	User type Ctrl-C to terminate the program.
4	SIGILL	Program runs an illegal CPU instruction.
11	SIGSEGV	Segmentation fault: access memory incorrectly.
18	SIGTSTP	User type Ctrl-Z to temporarily suspend the program.
25	SIGWINCH	Terminal size changed.

For full list, see the man page `signal(7)`.

Processes can generate any of these signals using `kill()`. A number of signals are reserved just for these purpose.

Number	Symbol	Usual meaning
9	SIGKILL	User request the program be terminated immediately.
30	SIGUSR1	User-defined

E.g., type `kill -9 <pid>` to kill a process immediately.

POS(0230A)-2.26

Trapping signals

Why many types of signals? So that you can establish **signal handlers** to handle each type of signals differently, using the system call `sigaction()`.

E.g., if you don't want your program to be killed by Ctrl-C:

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void int_handler(int signo) { /* To be executed during SIGINT */
    printf("Don't kill me!\n");
}
int main() { /* This works for most signals, but not SIGSTOP and SIGKILL */
    int to_sleep = 100;
    struct sigaction act;
    sigaction(SIGINT, 0, &act); /* Get the old signal handler */
    act.sa_handler = int_handler;
    sigaction(SIGINT, &act, 0); /* Set the new signal handler */
    while (to_sleep) to_sleep = sleep(to_sleep);
    return 0;
}
```

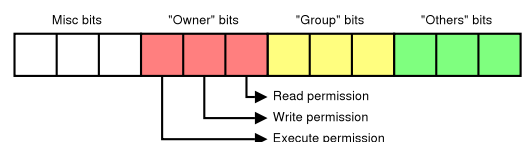
POS(0230A)-2.27

User identification

- In any multi-user OS, there are methods to **identify different users**. Even if the OS is supposed to be used by a single user, it is useful to have user identification—e.g., to prevent virus from modifying system files.
- In Unix, every user has a **user ID**, or **UID**. Each user belongs to one or more **groups**. Groups are identified by a **GID**.
- Each process, file ,directory, shared memory, etc., will be tagged with a UID and a GID: the **owner** of the process or resources.
- Each file ,shared memory, etc., has a **mode**, telling whether the owner and other users can read, write or "execute" the resource.
- The mode is usually represented in 3-digit **octal** numbers. The 3 digits represents permission for the user, the group and other people.
- The mode is determined when you create the resource (e.g., create the file with `open()`), and may be modified later (e.g., using `chmod()`).

POS(0230A)-2.28

Interpreting Unix permission



E.g., if your file has the permission of the octal number 754...

- you can read, write and execute the file (7 means 111);
- other group members can only read and execute it (5 means 101);
- non-group members can only read it (4 means 100).

POS(0230A)-2.29

Default permission

What will happen if you don't specify the permissions?

There is a default, or actually "inverse" of the default, stored for each process. It is called the **umask**. It is usually 022 or 077.

This is to make sure that normal processes will not give others too much permission accidentally.

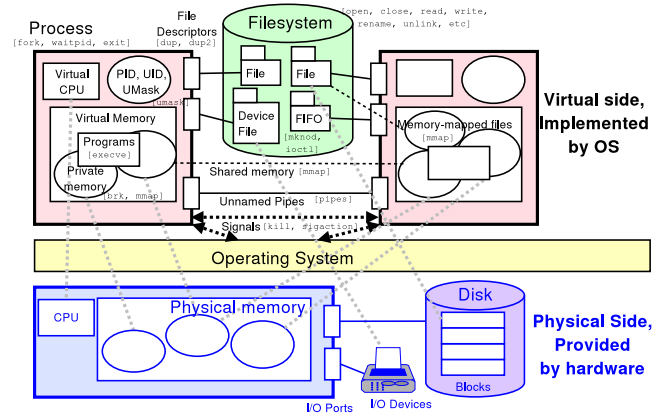
A program that wishes to create a group and world readable file would thus do something like the following:

```
// Initial creation of file
int fd = open("myfile", O_RDWR|O_CREAT|O_EXCL, 0644);
if (fd == -1) {
    // handle error, e.g., file already exists
}
// The file may be too restrictive. Force it to be 644
fchmod(fd, 0644);
```

Alternatively, use `umask()` to set `umask` to 0 before creating the file .

POS(0230A)-2.30

Revision



POS(0230A)-2.31

Watching the calls

- It would be convenient if the OS allows itself to be "tapped into", by **emitting a message somewhere whenever a system call is made**.
- In Linux, you can use a program called `strace` to ask the OS to do exactly that.
In Solaris: use `truss` instead.
- So to know what `ls` does, just say `strace ls`, and you will see all messages coming out telling you how the program uses the OS to list the directory, find the width of the screen and output the results.
Or use `strace ls > outfile` so that normal output won't go to terminal.
- There is another program that can intercept library calls (called `ltrace`), which may also be useful if you want to see what other programs do.

POS(0230A)-2.32