Lecture 2

OS external Concepts and Interface

In this lecture we will learn the basic concepts that modern operating systems are built upon.

References:

- [OSC] Section 3.1, 3.2
- [ULK] Basic OS Concepts (pp. 7–12)
- Unix man pages (section 2): fork, waitpid, exit, kill, execve, open, read, write, brk, mmap, shmget, shmat, shmctl.

Our focus in this lecture

- In this course, we will go into the OS, study the theory there, and understand how it provides system services.
- But various parts of the OS are heavily **inter-dependent**. It is impossible to understand one part of the OS completely without an overall picture.
- So in this lecture we will see a few **key services** provided by a popular OS (Unix), and how a program can access them (the **API**, application program interface to the system).
  After this lecture, you should be able to **use** most OS service, although we still don't know how it is **implemented**.
- We will quickly introduce all the basic concepts within the operating system, including **process**, **virtual memory**, **time-sharing**, **inter-process communication**, **I/O interface** and **file-system**
- For now, just focus on **what they are**, not **how they can be achieved** by the OS. The latter question will be addressed in the upcoming weeks.

Program loading

The earliest use of an OS is to load programs. This involves:

- **Reading the program** from the disk (or other storage device) to the memory, since computers can only run programs in memory.
- **Passing control** to the **entry point** of the program. I.e., setting the instruction pointer to the address of the entry point.
- In most modern OS, some **parameters** may be passed to the new programs as arguments, as well as a list of **environment variables**.

Unix: Done by the `execXX` "calls". (Try "man execl".) E.g.,

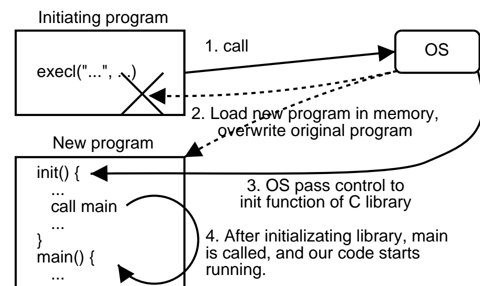
```
execl("/usr/bin/ls", "ls", "-l", NULL);
```

load `/usr/bin/ls`, give it two arguments `ls` (the program name) and `-l` (its option). *Never returns* 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 argument and env-var list.

Schematics of Unix program loading

The whole process is something like this:



The **old program no longer runs** 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.

Multi-programming

Once we allow multiple programs to remain in memory, we can run the same program for multiple times. We call each a **process**.

That is, **a process is (roughly) an instance of a running program**.

The program loading interface is not sufficient:

- The original program become non-existent (or is stopped).
- The new program cannot get data easily from the old one.

What we need: for a process to be able to

- **create** a new process **without stopping** the currently running one.
- **wait** for the **completion** of a process, **if desired**.
- **allow part of its data** to be used by the newly created one.

Unix Interface (Multi-programming)

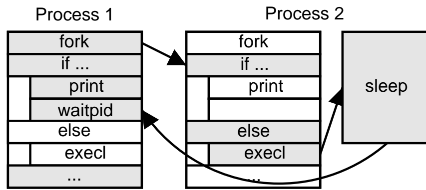
In Unix, `fork` creates a new process **running exactly the same program** as the current process, but get different return value from `fork`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
    pid_t p;
    if ((p = fork())) {         // 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);
}
```

Memory is copied: new process can use all data of the old one.

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 might be implemented by switching CPU from time to time.

Process 1 calls *waitpid*(), **suspending itself** until Process 2 terminates.

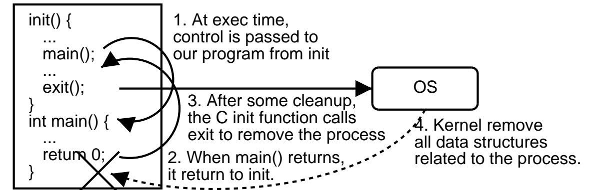
I.e., until sleep terminates. After the execl call, process 2 is no longer running the original program.

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.

Consequence (Multi-programming)

- **Processes** and **programs** are two independent concepts in a multi-programming OS.
- There can be more than one process running the same program (e.g., the same command issued twice, or a program executes a `fork()` call without `execXX()` call).
- The same process can be used for multiple programs (e.g., a program issues an `execXX()` call, the process is used for both programs, one after the other).
- The OS considers only processes when **allocating resources**, e.g., when selecting jobs to execute, when allocating memory, etc.

The OS use the abstraction of a process for keeping track of **resources allocation**.

Memory management

The memory of a computer serves multiple processes. The OS decides which process should use which part of the memory.

- Memory is allocated when a program is **loaded**, to hold the **program itself** and the **data associated** with the program.
- Memory is also allocated when a running process **requests** for it.
- Apart from allocating and deallocating memory, most OS can also **"swap" infrequently used memory contents** to disk temporarily, so that more memory are available for other processes.
- When swapped memory is subsequently needed, the OS **transparently load** it back from disk.
  Possibily swapping some other parts of memory to disk temporarily

Some OS allows **disk accesses** to be done like memory access. This is also serviced through the memory management system of the OS.

Unix interface

- After *execl*() is executed, the **old memory** of the process is **deallocated**, and a **new piece of memory is allocated**.
- The process can then use a part of memory containing the **global variables** of the program (which is the "associated data").
- It can also use a **stack**, storing local variables and return addresses of function calls.
- *fork*() causes memory of a process to be **copied** to the new one.
- Processes can call *brk*() to **enlarge or shrink** the memory area for global data. It can also call *mmap*() to allocate new memory regions. The *mmap* call also allow disk files to be read like memory.
  Most program would simply use function like *malloc*(), or language features like **new**, which eventually use these OS services.
- In a user program, there is **no call to deal with swapping**. (Why?)

Example

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

int size;        // Allocated on program loading

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

    // Note the different place of allocation
    size = getpagesize();
    printf("size at %x\n", &size);
    printf("i at %x\n", &i);
```

```
    // New memory region
    mem = mmap(0, size, PROT_READ|PROT_WRITE,
          MAP_PRIVATE|MAP_ANON, 0, 0);
    printf("mem at %x\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 %x\n", map);
    for (i = 0; i < size; ++i)
          printf("%c", map[i]);
    printf("\n");
}
```


Consequence

- A user program refers to its memory (*variables)* by an address that is **the same throughout the execution** of the program.
- The **OS want to move memory around** for different purposes, like swapping or mapping memory.
- We thus distinguish the concepts of **physical memory** as seen by the OS, and the **logical memory** as seen by the program.
- The logical memory of a program need not be completely stored in the physical memory all the time.
- The OS needs to **detect** that a program is **accessing missing memory**, and loads it back to physical memory accordingly.
- An we will see later, this requires some support from the CPU.
  Memory management unit (MMU) translates logical addresses to physical ones.

Filesystem

- Computers are equipped with various kinds of secondary storage devices that can be used to store 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.
- It also tries to **speed up** accesses, e.g., by maintaining **memory caches** so that data recently read from disks doesn't need to be read again, scheduling accesses to minimize disk head movement, etc.
- Most user processes don't directly use the disk, but instead use it through the file system.
- The OS **provides calls to access files** in the file system, e.g., to **create**, **delete**, **open**, **close**, **rename**, **read** and **write** files.

Unix interface

In Unix, files are given **pathnames** like "/etc/passwd".

A process must "open" to obtain a "file descriptor" before using a file . Then it can use "read" and "write" to access the file .

```
#include <stdio.h>
#include <sys/fcntl.h>
int main() {
      char c[1024];
      int size, fd;
      fd = open("/etc/passwd", O_RDONLY);
      size = read(fd, c, 1024);
      while (size > 0) {
            int i;
            for (i = 0; i < size; ++i) printf("%c", c[i]);
            size = read(fd, c, 1024);
      }
}
```


Overlapped CPU and I/O

- There is little use of a computer if it can't do I/O.
- I/O devices are usually the slowest thing in the computer. E.g., keyboard events occurs at the speed of human, modems usually send signals at 57600 bytes per seconds (while some CPU operates at 1GHz).
- How to bridge the gap?
  - **Wait** for it **in a loop** (**while** a keystroke is not received **do** nothing): **waste the CPU**.
  - **Continue** to work on other things, **stops once a while** to check the device: **difficult to program** correctly; sluggish I/O.
  - We want a mechanism to keep the CPU busy doing other things, while allowing programs to respond to events quickly.
  - The mechanism should also be convenient to use.

Unix Interface (Overlapped CPU and I/O)

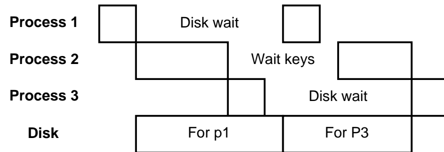
Unix allows users to **do I/O like files**. E.g., the sound device is available at the file `/dev/dsp`. To read sound for 1 second:

```
#include <stdio.h>
#include <sys/fcntl.h>
int main() {
      char buf[8000];
      int fd = open("/dev/dsp", O_RDONLY);
      if (fd ≥ 0) {
            read(fd, buf, 8000);
            close(fd);
      } else
            printf("Error\n");
}
```

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

Consequence (Overlapped CPU and I/O)

- User programs **do not perform I/O directly**. They do so through some **abstractions** (e.g., files) provided by the OS.
- The hardware (the devices) has some mechanism to **tell the CPU** that it has **completed a task**. This is used by the OS to construct the abstraction. The user programs are usually ignorant about it.
- Processes **share the CPU**, so that CPU and I/O of different processes overlaps. This improves the performance of the overall system.



Inter-Process communication (IPC)

- How to let different processes to communicate?
  There is one easy way: **through the file system**. But that is **slow** to transfer significant amount of data as files ,since I/O is involved.
- The OS provides abstractions like **shared memory** and **message queues** for processes to communicate directly.
- Share memory works by having **two or more process being able to read and write the same memory**.
- Message passing works by having one process **sends** some data to a "mailbox" in the OS kernel and another **retrieves** it.
- There are also means for **asychronous** communication (receiver runs no code to wait for it), e.g., by **signals**.
- When a signal is received, the execution of a process is **interrupted** temporarily, until the **signal handler** returns.

Unix interface

There are many IPC interfaces in Unix. One example: shared memory.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/shm.h>
int main() {
        char *shmptr;
        int i, shmid = shmget(IPC_PRIVATE,
           1024, IPC_CREAT|0700);
        printf("shmid: %d\n", shmid);
        shmptr = shmat(shmid, 0, 0);
        for (i = 0; i < 100; ++i) {
                *shmptr = (char)i;
                sleep(1);
        }
        shmctl(shmid, IPC_RMID, 0);
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/shm.h>
int main() {
        char *shmptr;
        int i;
        int shmid;
        printf("shmid? ");
        scanf("%d", &shmid);
        shmptr = shmat(shmid, 0, 0);
        for (i = 0; i < 100; ++i) {
                printf("%d\n", *shmptr);
                sleep(1);
        }
}
```


User identification

- In any multi-user OS, there are methods to **identify different users**. Lacking this causes major insecurity and inconvenience.
  Windows 98 is not a multi-user OS, and doesn't differentiate users. Consequently, any user has the ability to corrupt the whole system, e.g., by overwriting a system file .(Great for viruses)
- Differentiation between users involves:
  - Associate **owners** for each process and resources.
  - Owners **specify what operations others are allowed to perform**.
  - **Accesses** are allowed **only through the OS**.
  - The OS **allows only the specified operations**.
  - The OS **protect itself** from being tempered with.

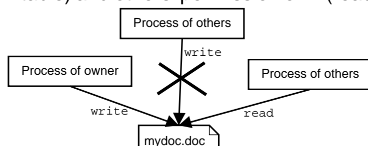
Unix interface

In Unix, every user is given a "**uid**" (user identifier), which is a "small" integer, i.e., less than 65535.

Every **process carries an uid**, which can be fetched with `getuid`.

Every **resources** to be protected **carries an owner**. E.g., each file ,each process, each piece of shared memory, etc. has an owner.

The **owner specifies the permissions of the resources** separately for himself and for others' access. E.g., a file can have user's permission of 'rw' (readable, writable) and others' permission of 'r' (readable).



Summary, Exercise

- We just have a very quick look at what an OS provides for the programs running in a computer.
- Briefly, an OS provide concepts like process, memory system, file system, I/O abstraction, interprocess communication and user identification and protection.
- In the remaining weeks, we turn into each abstraction, finding the different ways in which they are implemented and consider their trade-offs.
- We also study problems they create and how to solve those problems.

Exercise

- Read the manual pages of the C function calls that we have mentioned.