Operating Systems

System Calls

Outline

- **Motivation & basics**
- Process control
- File management
- Concluding example

Motivation

- Problem – multiple processes created by several vendors need to share the same set of computer resources
  - How can I write stuff to the disk and not interfere with all other processes
  - How can I separate stuff meant for me received on the network from stuff that belongs to other processes?
  - How can I ensure there is no CPU hog that takes the CPU and does not share?
- A process is ***not supposed*** to access the hardware.
  - It will call a "supervisor" to use the hardware
- The OS = (The "supervisor") is in charge of managing the resources
- This is strictly enforced ("protected mode") for good reasons:
  - Can jeopardize other processes running.
  - Cause physical damage to devices.
  - Alter system behavior.
- The system call is the mechanism that provides a safe mechanism to ***request*** specific kernel operations.

A side

- If multiple processes run on the same hardware (Hypervisor!) we have “Super Supervisor” this is called Hypervisor (Hyper = Above)
- A Hypervisor treats Oss like OS treats processes.
- This is beyond our scope

System Call - Definition

- What is a System Call?
  - An ***interface*** between a ***user application*** and a ***service*** provided by the operating system
- System call interface – see next slide
- Separate ASM instruction
  - Call – brunch to a function (return with ret)
  - Syscall – brunch to the OS (return with sysret)
    - Increase permission level
    - (intel Ring 3->Ring 0, ARM EL0->EL1)
    - There is a syscall table in the kernel with addresses of functions based on syscall number

Last kernel interface is interrupts

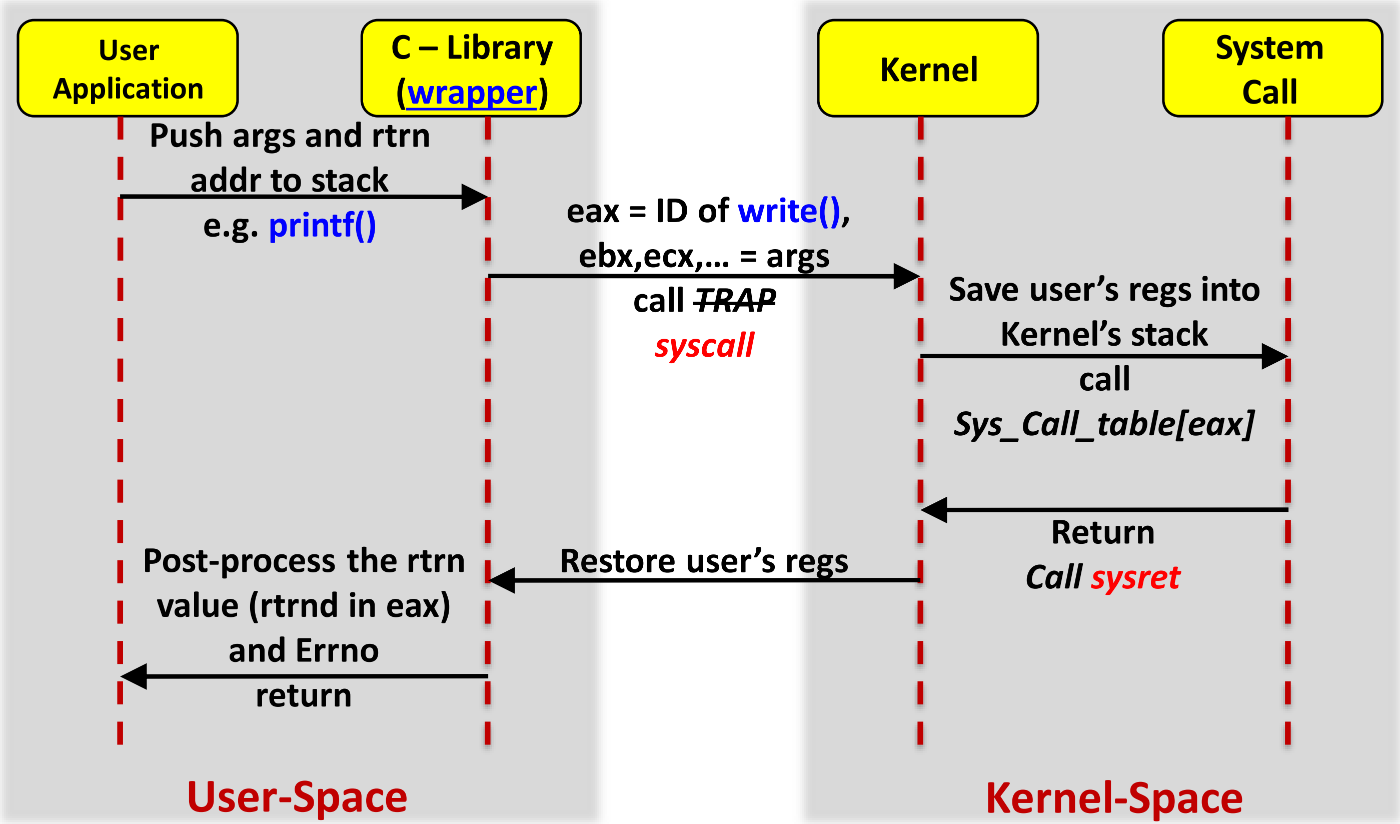
- Something (event) that requires immediate attention
- Return with iret
- Mostly beyond scope

System Calls - Categories

- System calls can be roughly grouped into ***five*** major categories:
  - Process control (e.g. create/terminate process)
  - File management (e.g. read, write)
  - Device management (e.g. logically attach a device)
  - Information maintenance (e.g. set time or date)
  - Communications (e.g. send messages)

System Calls - Interface

- Calls are usually made with C/C++ library functions



Convention

- When I describe a command you can type on the command line I will use (1) e.g. gcc(1) ; ls(1) etc.
- When I describe a syscall I will use (2) e.g. open(2) ; socket(2) ; listen (2) etc.
- When I describe a standard library call I will use (3) e.g. printf(3) ; srand (3) etc.
- This follows man (1) standard

Man(1)

- Short for manual
- Nothing mesogenic
- You can get instructions on how to use virtually any UNIX command

Outline

- Motivation & basics
- **Process control**
  - **Creating a new process**
  - Waiting for a process
  - Running a script / command
  - Error report
- File management
- Concluding example

Process Control: fork ()

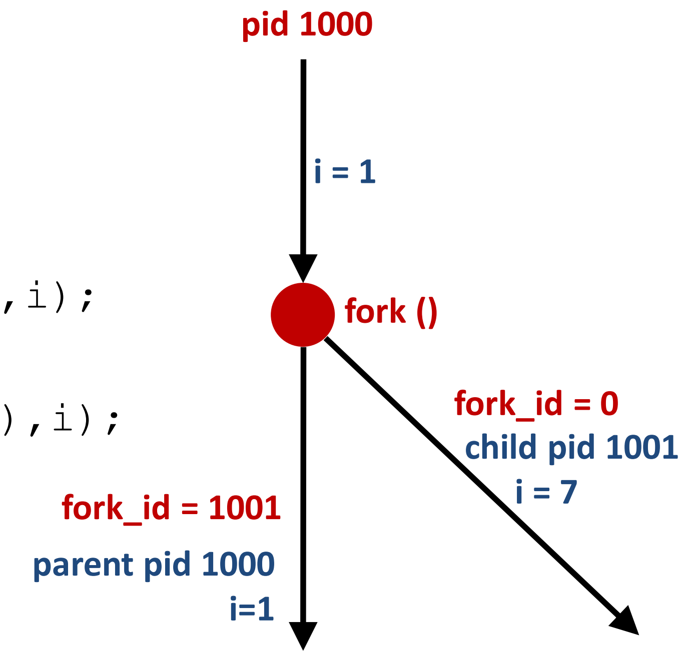
- `pid_t fork(void);`
  - Creates a new process, which is an exact duplicate of the caller
    - Including all file descriptors, registers, instruction pointer, etc
  - Both child and parent resume from after the fork() command
    - and go on their separate ways
  - fork() returns
    - The child’s pid, if called by the parent
    - 0, if called by the child
  - A child process can get his parent pid using getppid()

fork () and Copy On Write: motivation

- When fork is invoked, the parent's information should be copied to its child
- May be wasteful if the child will not need all this information
  - To avoid such situations, use ***Copy On Write*** (COW).

fork(): Example 1

```
int i = 1;
printf("my process pid is %d\n",getpid());
fork_id = fork();
if (fork_id == -1) {
       perror ("Cannot fork\n"); exit
(EXIT_FAILURE);  }
else if (fork_id == 0){
       i=7;
       printf("child pid %d, i=%d\n",getpid(),i);
}else
       printf("parent pid %d, i=%d\n",getpid(),i);
return 0;
```




```
Output:
my process pid is 1000
child pid 1001, i=7
parent pid 1000, i=1
```

```
Is this the only possible output?
How can we force the output to be deterministic?
```

How many times hello world is printed

```
Int main(int argc, char * argv[]) {
        printf("hello world");
        fork();
        printf("\n");
        fflush();
}
```

Answer – Usually TWO

- Though the standard does not specify (= allow implementation to choose) most stdlib C implementations flush the output buffer only on endofline.
- The printf function moves “Hello world” to the output buffer.
- Then calling fork and flushing will cause both parent and child to flush resulting in two printings.

Outline

- Motivation & basics
- **Process control**
  - Creating a new process
  - **Waiting for a process**
  - Running a script / command
  - Error report
- File management
- Concluding example

Zombies

- When a process ends, the memory and resources associated with it are ***de-allocated***.
- However, the entry for that process is ***not*** removed from its parent's process table.
  - This allows the parent to ***collect*** the child's exit status.
- When this data is not collected by the parent the child is called a "***zombie***".
  - Such a leak is usually not worrisome in itself. Actually, in some (rare) situations, a zombie is actually ***desired*** – e.g., for preventing the creation of another child process with the same PID.
  - However, the existence of unplanned zombie is a good indicator for problems to come.

Detecting and collecting zombies

- A Zombie can be **collected** by the parent process with the **`wait()`** system call.
    - See next slide
- Zombies can be **detected** with **`ps -el`** (marked with ‘Z’).

wait(), waitpid(), waitid()

- **`wait()`** – wait for a change in the status of *any* of the children
  - wait – ie, suspend execution of the calling process
  - status: the process terminated / was stopped / was resumed.
  - Once the status of a process is collected, that process is removed from the process table of the collecting process.
- **`waitpid(), waitid() :`** A finer control then wait(), e.g.
  - Wait for a specific process
  - Wait for any one from a group of processes.
- Detailed documentation

Outline

- Motivation & basics
- **Process control**
  - Creating a new process
  - Waiting for a process
  - **Running a script / command**
  - Error report
- File management
- Concluding example

Running another file: exec ()

- `int execv(char const *path, char const *argv[]);`
- `Variants: int execve(), execvp(), execvl(), …`

- A family of C-library functions, which replace current process image with a new process image (text, data, stack, etc.).
  - Since no new process is created, PID remains the same.

- **`exec()`** functions ***do not*** return to the calling process unless an error occurred.
  - in which case -1 is returned and **`errno`** is set with a special value.

Why?

- Why do I need to duplicate myself and then call exec to replace myself when I want to start a new process?
- Answer 1 – not so bad due to CoW
- Answer 2 – That’s just the way it is!
- I am not discussing what could happen. I discuss how things work!

What about system(3)

- System(3) is just a library function that calls
- If (!(fork()) execve(…);
- wait();
- You can’t start processes without calling fork(2)!
- (or clone(2))

Outline

- Motivation & basics
- **Process control**
  - Creating a new process
  - Waiting for a process
  - Running a script / command
  - **Error report**
- File management
- Concluding example

errno

- A system variable, which is set by system calls in the event of an error
- Usually i*ndicates what went wrong*
  - However, "*a function that succeeds is allowed to change errno*" (Linux' manual)
  - The existence or an error is indicated by the function's return value
    - Usually -1 indicates an error
- Frequently a macro.
  - E.g. EACCES (permission denied), EAGAIN (rsrc temporarily unavailable).
- **errno** is *thread local* and *thread-safe*, meaning that setting it in one thread does not affect its value in any other thread.
- Use perror(3) to report errors
- Be wary of mistakes such as:
  - ```
    If (call() == -1){
            printf("failed…");
            if (errno == …)
    }
    ```
- Code defensively! Use **errno** often!

What's the problem?
errno may have been changed by *printf()*
    ```

Process control - example 2

```
int main(int argc, char **argv){
        …
        while(true){
                type_prompt();
                read_command(command, params);
                pid = fork();
                if (pid<0){         //fork failed
                        if (errno == EAGAIN) {
                                perror("fork:");
                                coutinue;
                        }
                        else {
                                … //handle other possible errors
                        }
                }
                if (pid>0)          //parent
                        wait(&status);
                else                //child
                        execvp(command,parmas);
        }
}
```

Outline

- Motivation & basics
- Process control
- **File management**
    - **File descriptors**
    - Basic operations: Open, close, lseek, duplicate
    - Example file
- Concluding example

File descriptors

- In POSIX operating systems, files are accessed via a ***file descriptor***
  - In Windows: *file handle*.
- File descriptors can refer to files, directories, sockets and a few more data objects.
- A file descriptor is an integer specifying the index of an entry in the ***file descriptor table***.
  - A file descriptor table is held by each process, and contains details of all open files.
  - The following is an example of such a table:

FD	Name	Other information
0	Standard Input (stdin)	…
1	Standard Output (stdout)	…
2	Standard Error (stderr)	…

Outline

- Motivation & basics
- Process control
- **File management**
  - File descriptors
  - **Basic operations: Open, close, lseek, duplicate**
  - Example file
- Concluding example

open(2) and close(2) of a file

- `int open(const char *pathname, int flags);`
- `int open(const char *pathname, int flags, mode_t mode);`
  - Returns a file descriptor for a given pathname.
    - This file descriptor will be used in subsequent system calls (according to the flags and mode).
  - Flags define the ***access mode***:
    - O_RDONLY (read only)
    - O_WRONLY (write only)
    - O_RDRW (read write).

- `Int close(int fd);`
  - Closes a file descriptor so it no longer refers to a file.
  - Returns 0 on success or -1 in case of a failure (**`errno`** is set).

Moving within a file: lseek(2)

- `off_t lseek(int fildes, off_t offset, int whence);`
  - Repositions the location within the file according to the directive **`whence`**
  - **`whence`** can be set to
    - **`SEEK_SET`** → move directly to offset
    - **`SEEK_CUR`** → move to current+offset
    - **`SEEK_END`** → move to end+offset
  - Positioning the offset beyond file end is allowed. This does not change the size of the file.
    - Writing to a file beyond its end results in a "hole" filled with '\0' characters (null bytes).
  - Returns the location as measured in bytes from the beginning of the file, or -1 in case of error (and sets **`errno`**).

dup(2)

- **`int dup(int oldfd);`**
- **`int dup2(int oldfd, int newfd);`**
  - Duplicates the file descriptor **`oldfd`**.
  - After a successful **`dup`** command is executed the old and new file descriptors may be used interchangeably.
  - They refer to the same open file descriptions and thus share information such as offset and status.
    - That means that using **`lseek`** on one will also affect the other!
    - They do not share descriptor flags (**`FD_CLOEXEC`**).
  - **`Dup`** uses the lowest numbered unused file descriptor, and **`dup2`** uses **`newfd`** .
    - closing current **`newfd`** if necessary
  - Returns the new file descriptor, or -1 in case of an error (and sets **`errno`**).

File descriptors – Example 3 (file: fd.c)

```
fileFD = open("file.txt"…);
/* closes file handle 1, which is stdout.*/
close(1);
/* will create another file handle. File handle
1 is free, so it will be allocated. */
fd = dup(fileFD);
/* don't need this descriptor anymore.*/
close(fileFD);
printf("this did not go to stdout");
```

What is the output?

0	stdin	…
~~1~~	~~stdout~~	~~…~~
2	stderr	…
3	file.txt	…

0	stdin	…
1	file.txt	…
2	stderr	…

File Management – Example 4

```
#define…
…
#define RW_BLOCK 10

int main(int argc, char **argv){
        int fdsrc, fddst;
        ssize_t readBytes, wroteBytes;
        char *buf[RW_BLOCK];
        char *source = argv[1];
        char *dest = argv[2];

        fdsrc=open(source,O_RDONLY);
        if (fdsrc<0){
                perror("eERROR while trying to open source fil:");
                exit(-1);
        }
        fddst=open(dest,O_RDWR|O_CREAT|O_ , 0666);
        if (fddst<0){
                perror("ERROR while trying to open destination file:");
                exit(-2);
        }
```

perror() produces a message on the standard error output describing the last error encountered during a call to a system call. Use with care: the message is not cleared when non-erroneous calls are made.

exit(2) system call

Bitwise OR: open for both *reading* and *writing*. If the file does not exist create it and always start at 0.

File Management – Example 4 (Cont’)

```
    lseek(fddst,20,SEEK_SET);
    do{
            readBytes=read(fdsrc, buf, RW_BLOCK);
            if(readBytes<0){
                    if(errno == EIO){
                            printf("I/O errors detected, aborting.\n");
                            exit(-10);
                    }
                    exit (-11);
            }
            wroteBytes=write(fddst, buf, readBytes);
            if (wroteBytes<RW_BLOCK)
                    if (errno == EDQUOT)
                            printf("ERROR: out of quota.\n");
                    else if (errno == ENOSPC)
                            printf("ERROR: not enough disk space.\n");
    } while(readBytes>0);
    lseek(fddst,0,SEEK_SET);
    write(fddst,"\\*WRITE START*\\\n",19);
    close(fddst);
    close(fdsrc);
    return 0;
}
```

Start writing at offset 20. If the file is opened with ***hexedit***, the first 20 bytes will be 00.

Using errno directly.

Adding an extra comment at the beginning of the file.

Outline

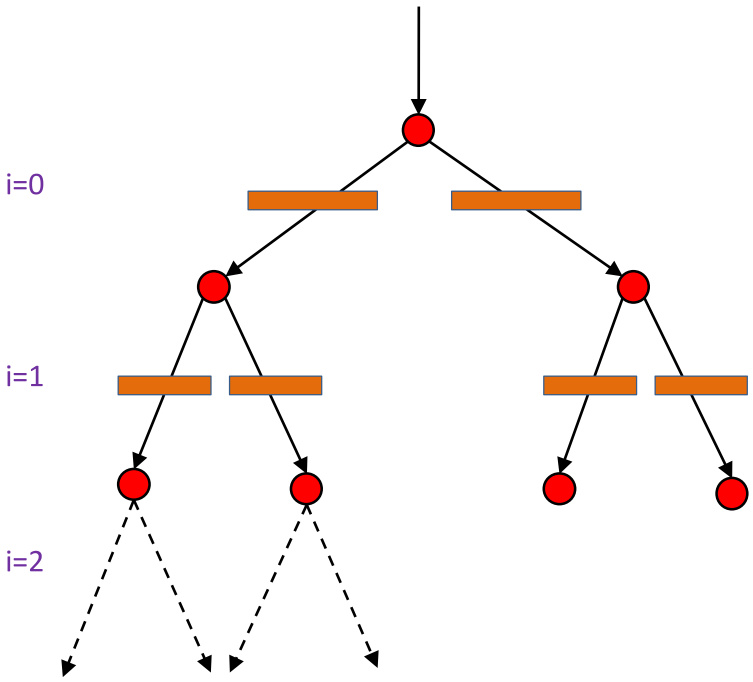
- Motivation & basics
- Process control
- **File management**
  - File descriptors
  - Basic operations: Open, close, lseek, duplicate
  - **See example file: child_dad_fd.c**
- Concluding example

Outline

- Motivation & basics
- Process control
- File management
- **Concluding examples**

Concluding examples: example 5.1

```c
int main(int argc, char **argv)
{
        int i;
        for (i=0; i<10; i++){
                fork();
                printf("Hello\n");
        }
        return 0;
}
```

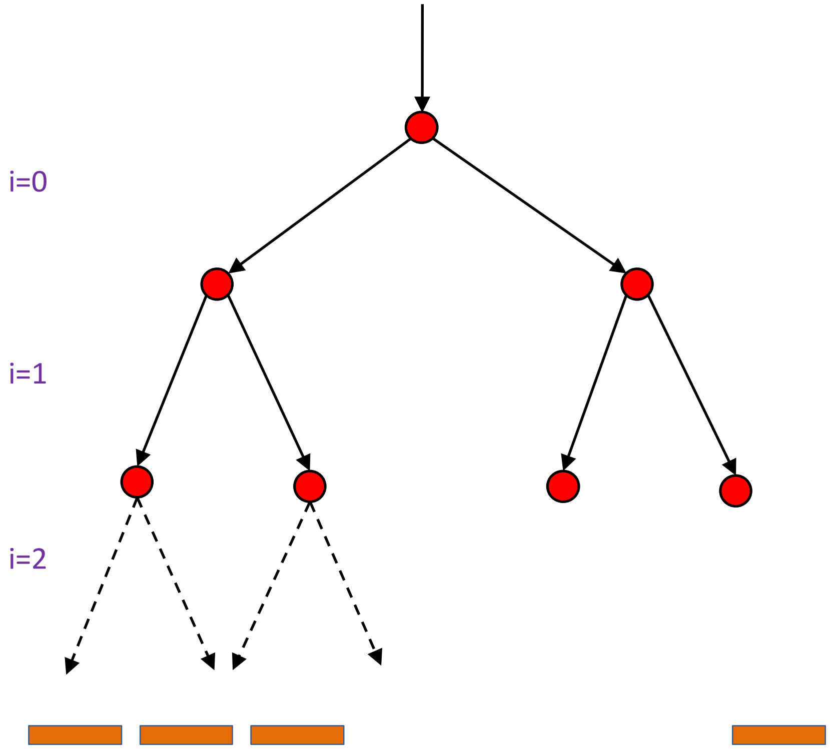


$$\sum_{i=0}^{9} 2^{i+1} = 2046$$

How many lines of "Hello" will be printed in the following example?

Concluding examples: example 5.2

```
int main(int argc, char **argv)
{
        int i;
        for (i=0; i<10; i++)
                fork();
        printf("Hello\n");
        return 0;
}
```



$$2^i = 1024$$

How many lines of "Hello" will be printed in the following example?