Operating Systems

Courtesy BGU-CSE and Dr. Itamar Cohen

Tutorial 1 – System Calls

Ben-Gurion University of the Negev
Communication Systems Engineering Department

Outline

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

Motivation

- A process is ***not supposed*** to access the kernel.
  - It can’t access the kernel memory or functions.

- 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 mechanism provides a safe mechanism to ***request*** specific kernel operations.

System Call - Definition

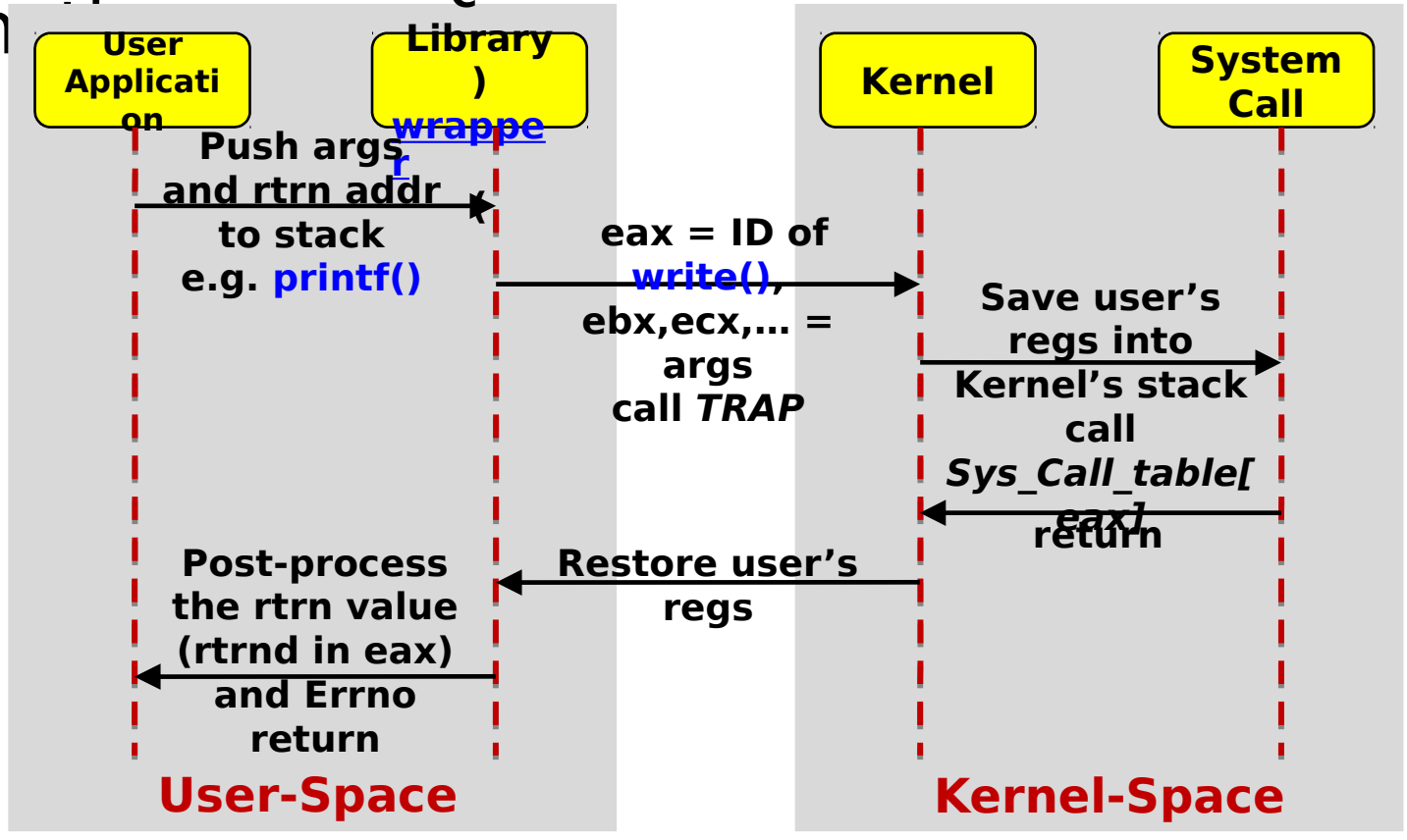
- What is a System Call?
  - An ***interface*** between a ***user application*** and a ***service*** provided by the operating system (or kernel).
- System call interface – see next slide
  - More info

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 fun[illegible] c-



Outline

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

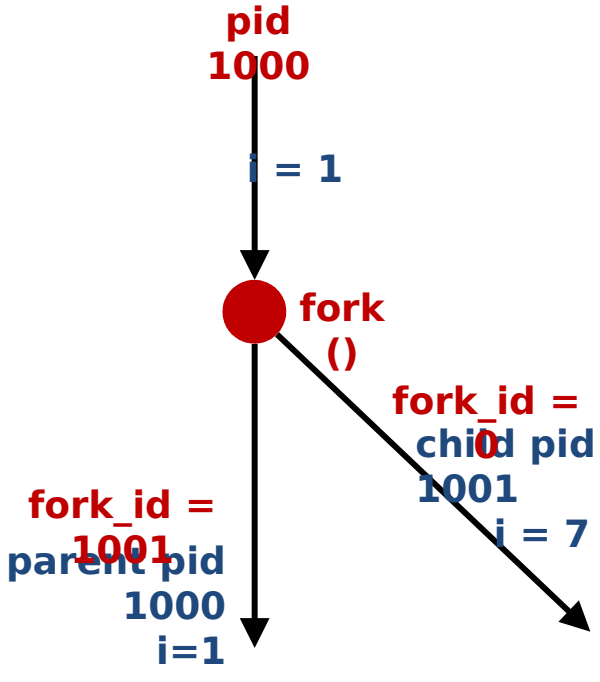
- When fork is invoked, the parent’s information should be copied to its child
- May be wasteful if the child will not need all this information
  - To avoid such situations, use ***Copy On Write*** (COW).

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?

Outline

- Motivation & basics
- **Process control**
  - Creating a new process
  - **Waiting for a process**
  - Running a script / command
  - Error report
- 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.
  -

Detecting and collecting zombies

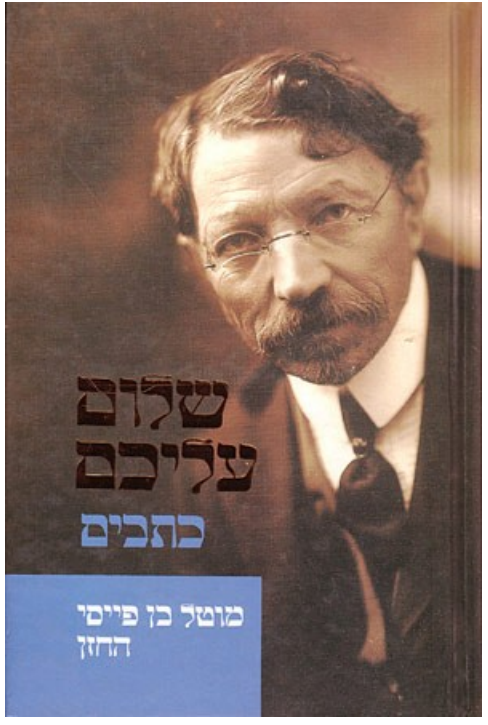
- A Zombie can be **collected** by the parent process with the **`wait()`** system call.
  - See next slide
- Zombies can be **detected** with **`ps -el`** (marked with ‘Z’).

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

Orphans

- When the parent process dies before its child, the child becomes an ***orphan*** process
- The orphans are “adopted” by the **`init`** system’s process (PID 1)
  - *aka* **reparenting**
- Sometimes orphans are generated intentionally, in order to serve as *daemon* process, e.g.
  - Printer daemon
  - *sshd* – responsible for accepting *secure shell* connections



Outline

- Motivation & basics
- **Process control**
  - Creating a new process
  - Waiting for a process
  - **Running a script / command**
  - Error report
- 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.

Outline

- Motivation & basics
- **Process control**
  - Creating a new process
  - Waiting for a process
  - Running a script / command
  - **Error report**
- 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.
- Be wary of mistakes such as:

```
If (call() == -1){
    printf("failed…");
    if (errno == …)
}
```

- Code defensively! Use **errno**

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

Process control - example 2

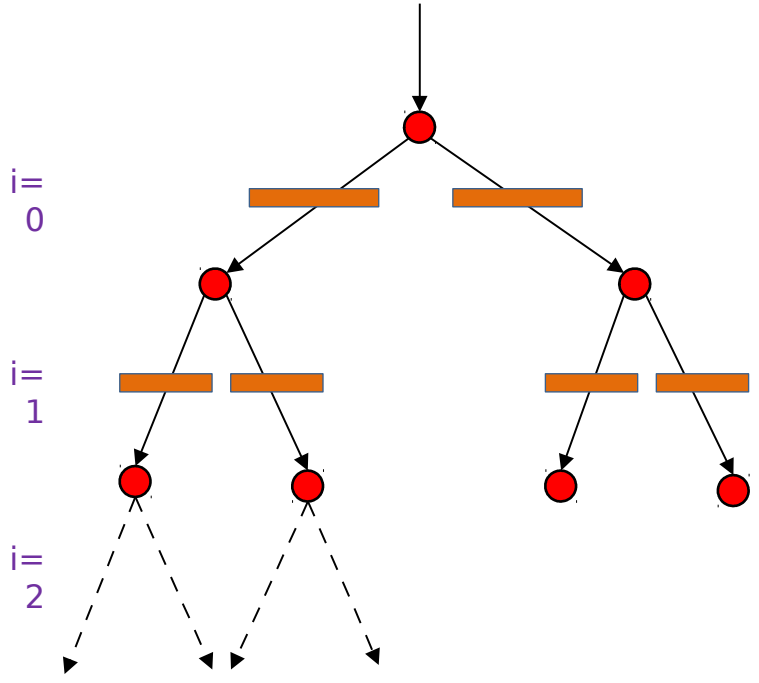
```
int main(int argc, char **argv){
  …
  while(true){
  type_prompt();
  read_command(command, params);
  pid = fork();
  if (pid<0){  //fork failed
  if (errno == EAGAIN) {
  printf("ERROR can't allocate memory\n");
  coutinue;
  }
  else {
  … //handle other possible errors
  }
  }
  if (pid>0)   //parent
  wait(&status);
  else   //child
  execvp(command,parmas);
  }
}
```

Outline

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

Concluding examples: example 5.1

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

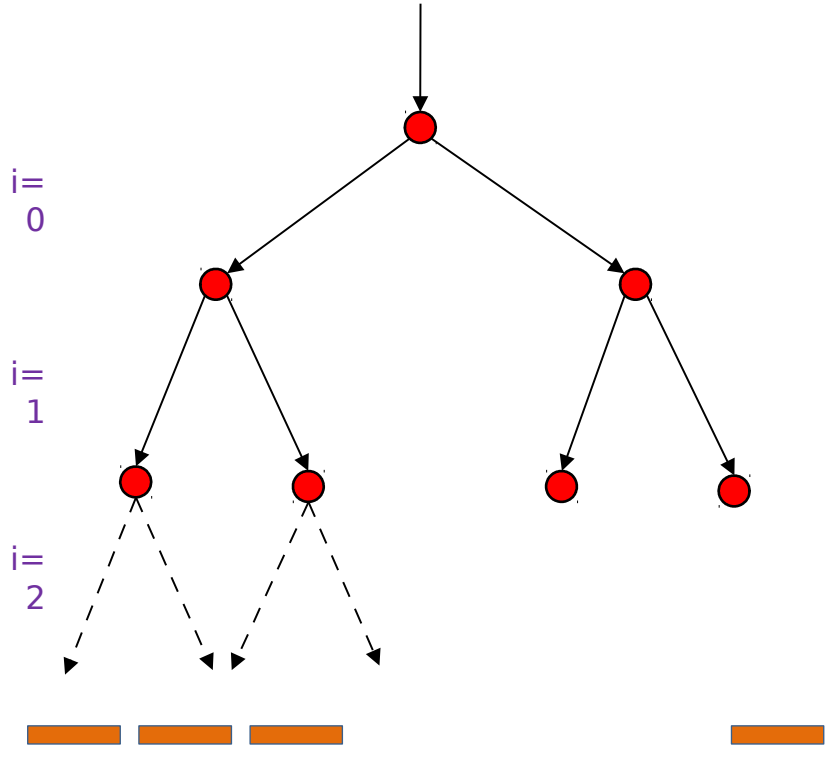


$$\sum_{i=0}^{9} 2^{i+1} = 2046$$

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;
}
```



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

$2^i = 2^{10} = 1024$