INF113: Manipulating Processes

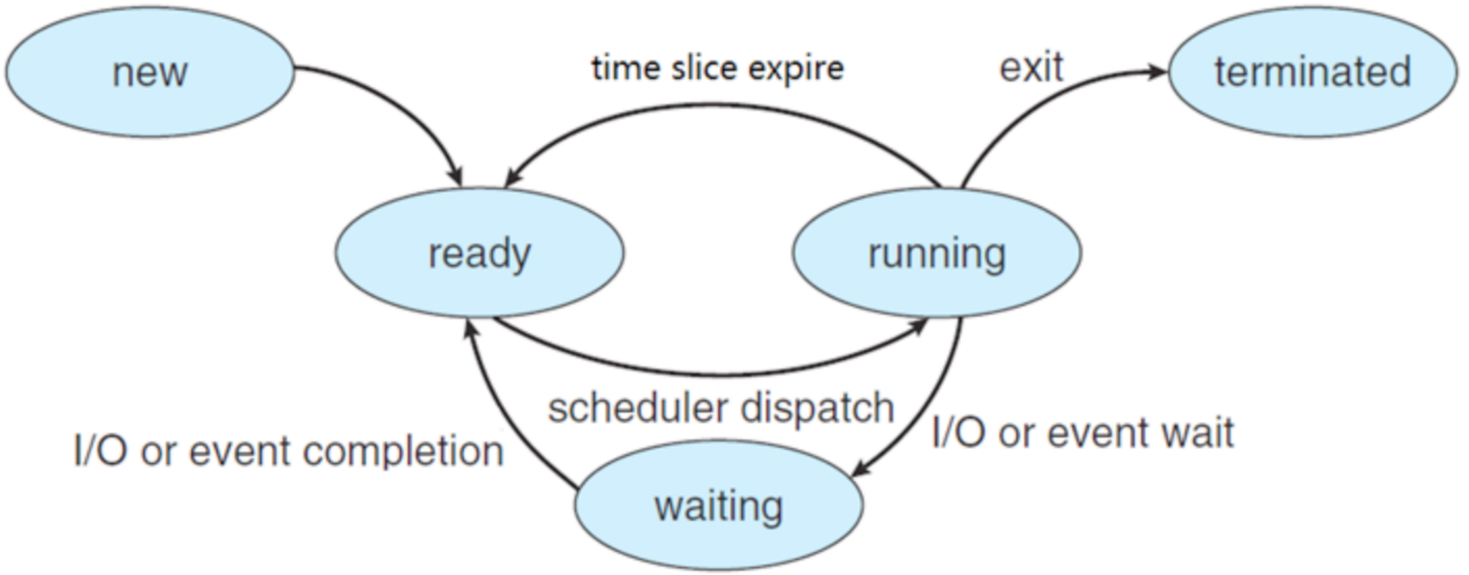
Kirill Simonov

05.09.2025

Process states

- Process actions:
  - create
  - destroy
  - stop/resume
  - status
  - wait for finish

  available to the OS and via syscall



Common process states in Linux:

```
D: uninterruptible sleep
R: running or runnable
S: interruptible sleep (waiting for an
event to complete)
T: stopped by job control
Z: ‘‘zombie’’ (terminated but not reaped)
```

Process tools

- `ps` gives the list of processes

```
ps aux #list all processes
```

- `top` and `htop` provide interactive monitoring
- `/proc`—a virtual filesystem to work with processes

```
cat /proc/cpuinfo
cat /proc/639560/cmdline
```

- with `kill` we can send signals to process with a known `pid`
  - `killall`—a wrapper to address processes by name
  - `pkill`—a wrapper to address processes by an expression

```
kill -SIGTERM 639560
```

Process signals

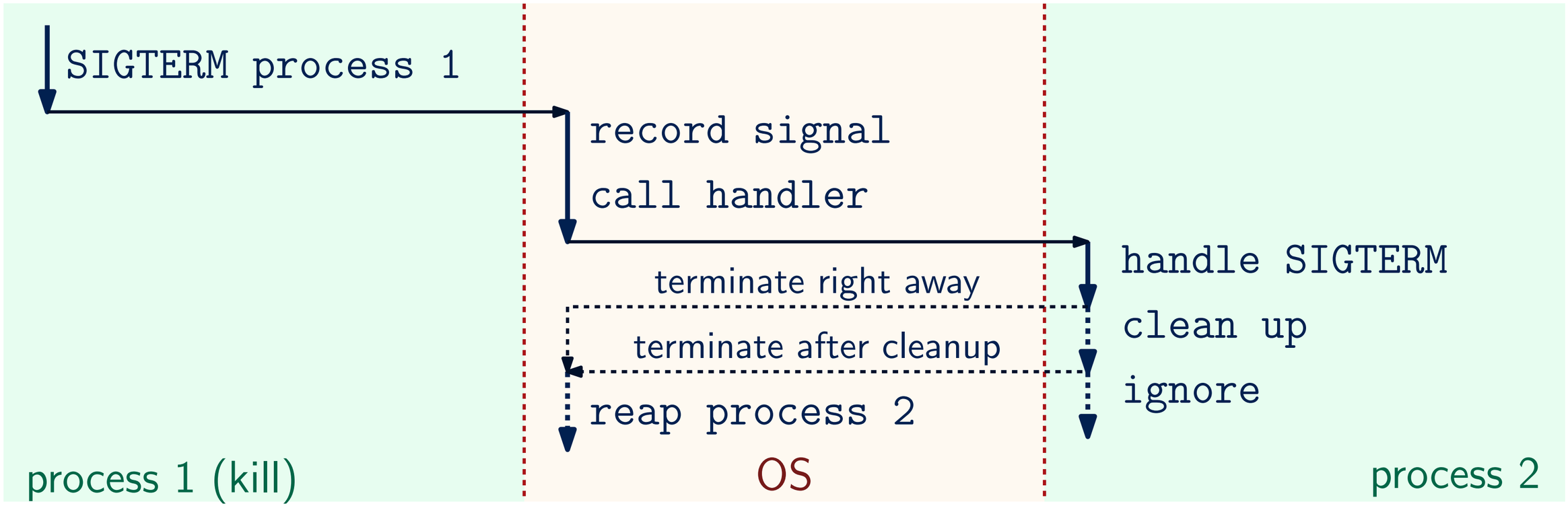
OS communicates with processes via **signals** Hotkeys:

- `SIGTERM` asks the process to terminate nicely
- `SIGINT` asks to return to prompt, often same as `SIGTERM` `Ctrl+C`
- `SIGKILL` terminates the process forcefully
- `SIGSTOP` stops the process until explicitly resumed `Ctrl+Z`
- `SIGCONT` resumes the process call `fg` utility
- Others communicate the type of error, alarms, children status, ...

```
man 7 signal
```

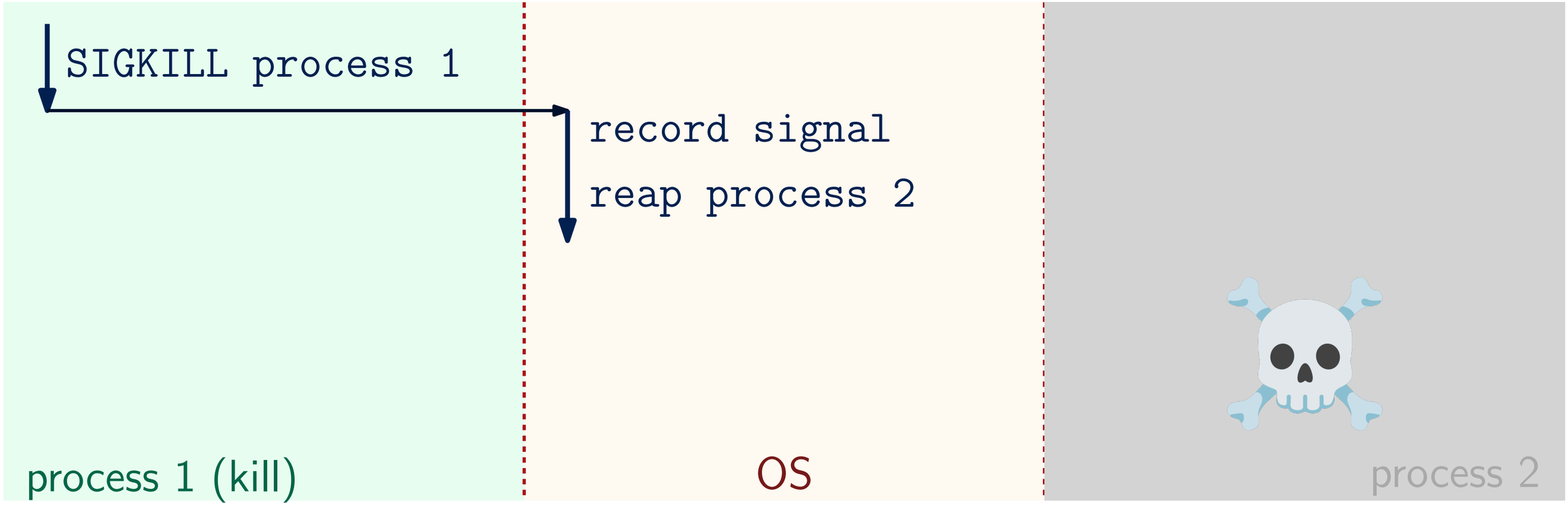
Handling signals

- `SIGTERM` and `SIGINT` can be handled or ignored by the process
  – OS, upon interrupt, calls the designated handler within the program



Handling signals

- `SIGTERM` and `SIGINT` can be handled or ignored by the process
  – OS, upon interrupt, calls the designated handler within the program
- `SIGKILL` and `SIGSTOP` cannot be handled
  – OS terminates/stops immediately



Fork: making children

- `fork` makes an exact copy of the process
  - Same memory and registers
  - Meaning also same code
  - Even same current instruction

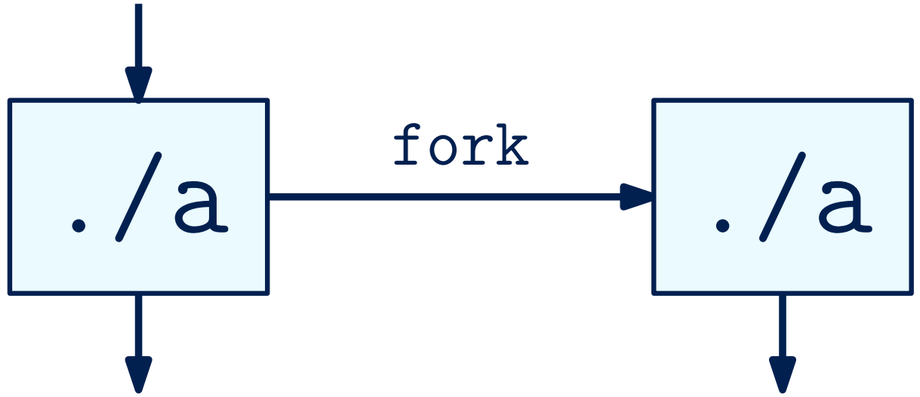
```
#include <unistd.h>

int main() {
    int rc = fork();   pid
    if (rc == 0) {
        //child
    } else {
 →      //parent
    }
    return 0;
}
```

- The only difference is the return value of `fork()`:
  - Parent: `pid` of the child
  - Child: `0`

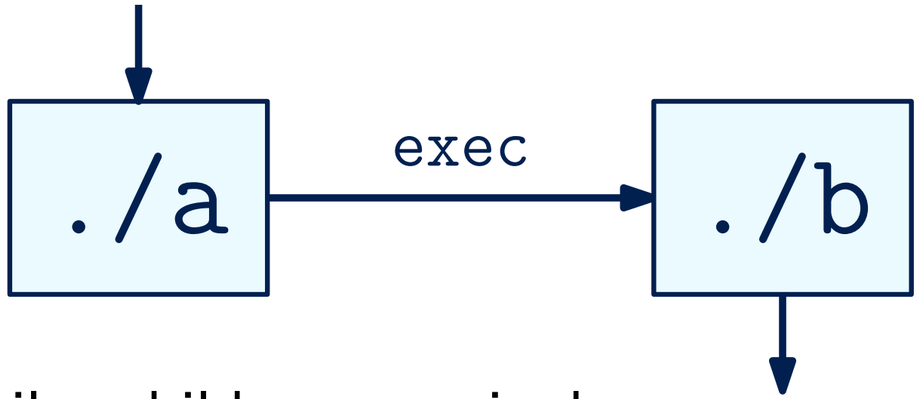
```
#include <unistd.h>

int main() {
    int rc = fork();    0
    if (rc == 0) {
 →      //child
    } else {
        //parent
    }
    return 0;
}
```

Create and control

- `fork`—create a new process, (almost) a copy of the current one



- `exec`—replace program in the current process

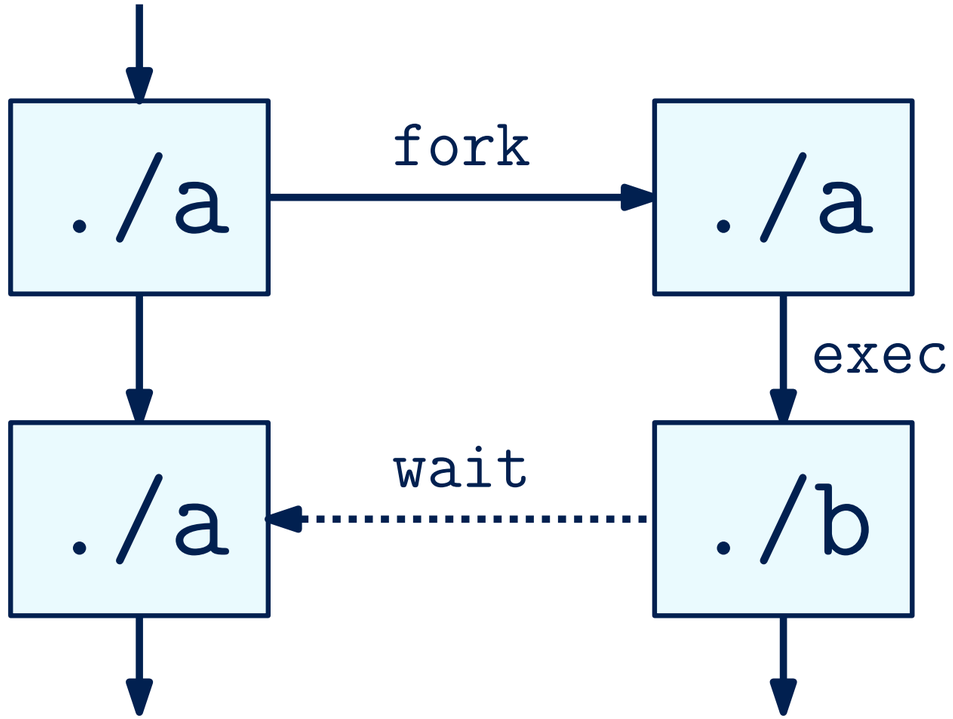


- `wait`—wait until a child process is done

Fork and exec

- `fork` and `exec` form a combo to start any new process
- Example: `./a` is `bash`, `./b` is `ps > p.out`

```c
#include <unistd.h>

int main() {
    int rc = fork();        ps > p.out
    if (rc == 0) {
        //preparation       set stdout
        //code              to p.out
        exec();             run ps
    } else {
        //parent            terminal waits
        //continues         and runs
    }
    return 0;
}
```



Process tree

- Processes form a hierarchy
  - Children inherit properties of the parent
  - Parents can (somewhat) influence children
    e.g., wait on them

```
$ ./a & #forks and runs ./b
$ ./c
```

- Properties:
  - Limits
  - User
  - Current directory
  - System variables, e.g., `$PATH`
  - ...