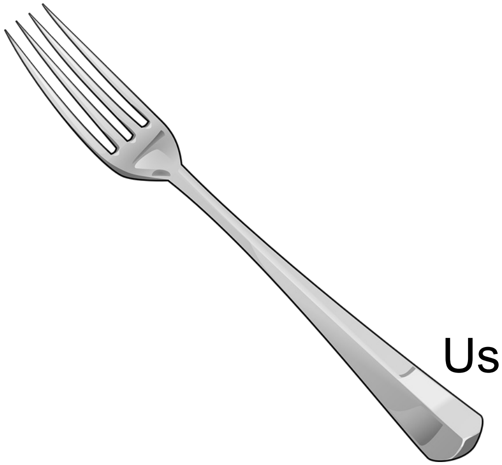
Using Fork and Pipe

ECE 650
Methods & Tools for Software Engineering (MTSE)
Fall 2023

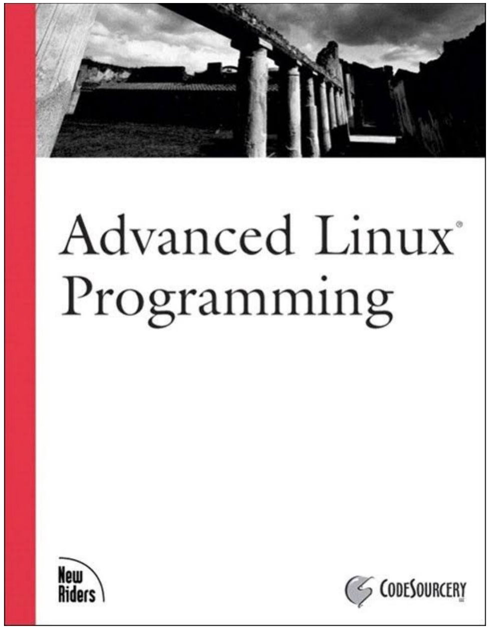
Presented by
Dr. Albert Wasef



Used by permission from Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

Additional Information



Advanced Linux Programming

- Chapter 2.1 (Interacting with Execution Environment)
- Chapter 3 (Processes)
- Chapter 5.4 (Pipes)

The book is available from the links below

https://github.com/MentorEmbedded/advancedlinuxprogramming/blob/gh-pages/alp-folder/advanced-linux-programming.pdf

https://github.com/MentorEmbedded/advancedlinuxprogramming/tree/gh-pages

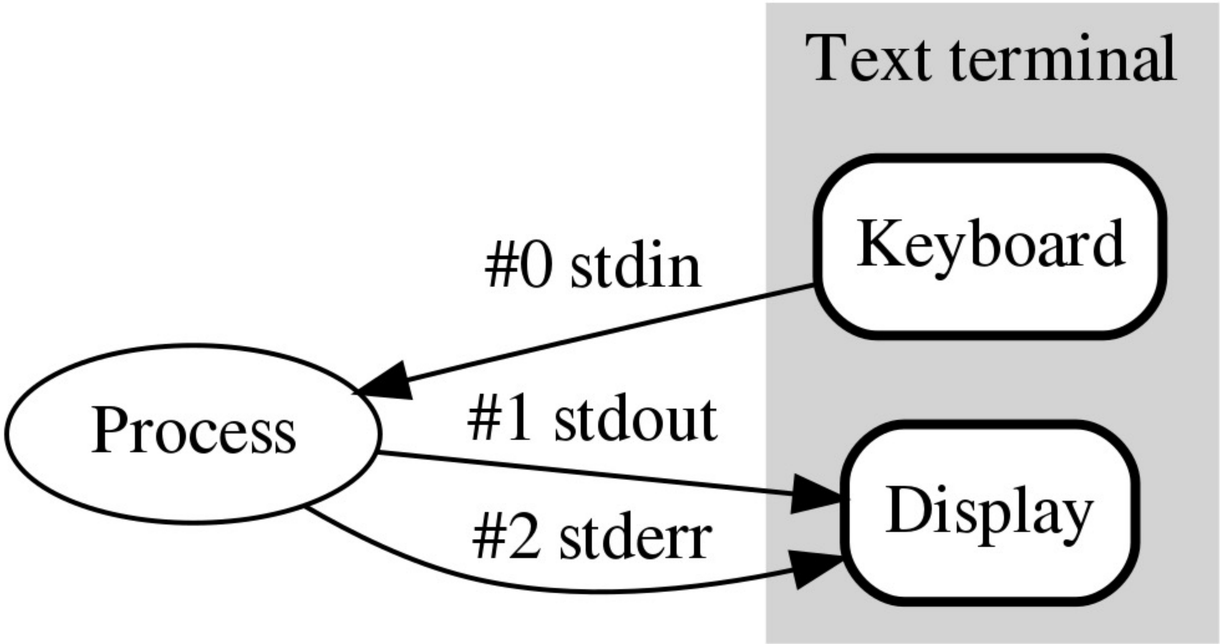
PROCESS

What is a “Process”?

- What is a process:
  - *“A running instance of a program”*
  - *Examples:*
    - *Each of the two instances of Chrome*
    - *The shell and the ls command executed, each is a process*

- Advanced programmers use multiple processes to
  - Do several tasks at once
  - Increase robustness (one process fails, other still running)
  - Make use of already-existing processes

Standard input, output, and error



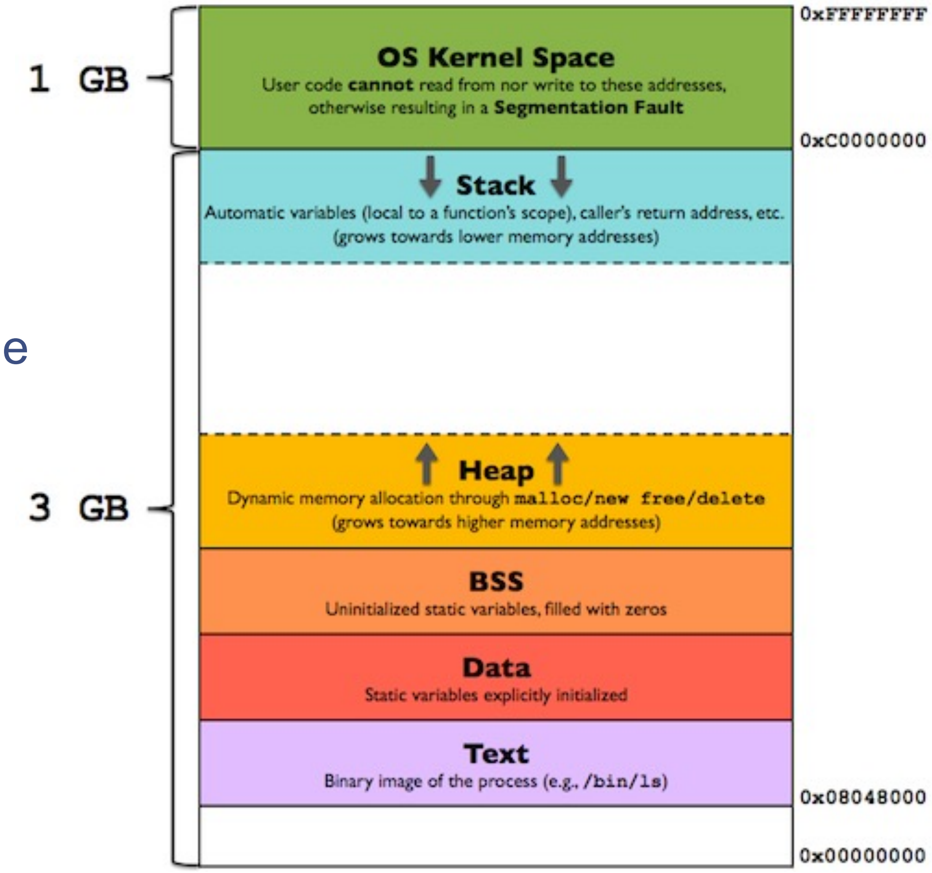
https://en.wikipedia.org/wiki/File:Stdstreams-notitle.svg

- Let's change stdin, stdout, and stderr

The “Guts” of a Process!

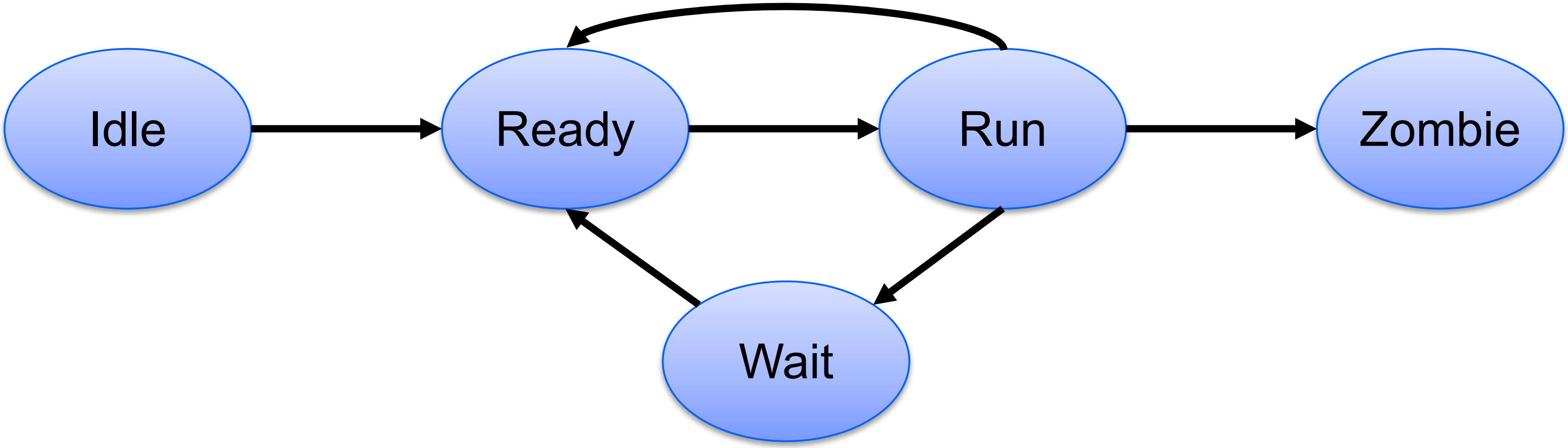
Process of a 32bit architecture

- The main components of a process:
  - An executable piece of code (a program)
  - Data that is input or output by the program
  - Execution context (information about the program needed by OS)


1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

Life Cycle of a (Unix) Process



Idle	state when the process is creating
Ready	ready to run
Run	executing
Wait	waiting for resources (CPU, disk, network, etc.)
Zombie	ended, waiting to be collected

https://www.cs.miami.edu/home/burt/learning/Csc521.111/notes/process-life-cycle.html

Let’s Dissect a Process!

- Windows:
  - Task manager

- Unix-like (Mac and Linux):
  - In the terminal type:
    - `ps` or `top` or `htop`
    - `ps -f` for full details

System Calls: using OS provided services

System calls (a.k.a., **syscalls**) is the programmatic way for a program (in user-space) to request services from an operating system (in kernel-space)

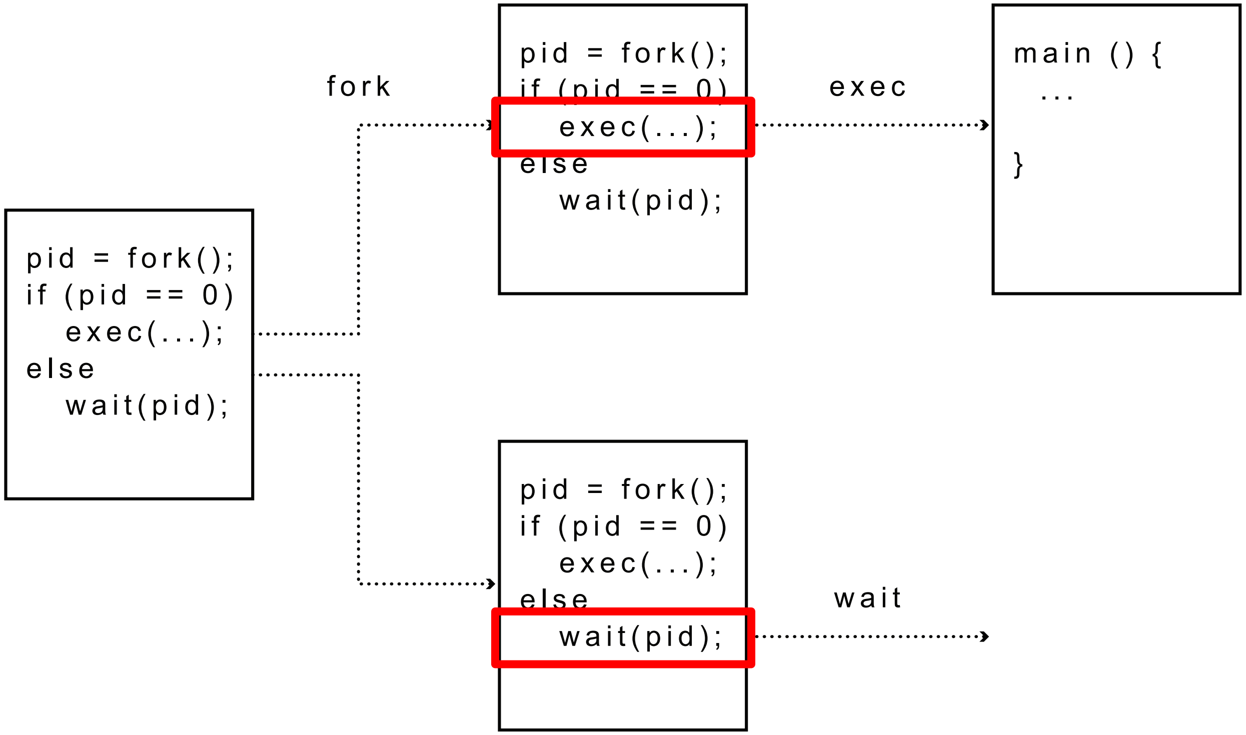
In code, a syscall looks like a function calls

- however, unlike a regular call it transfer control to OS
- therefore, syscalls are **independent** of the programming language
  - e.g., syscall fork() works the same in Python, C, C++, ...
- but **depend** on the OS
  - e.g., fork() works differently on Windows, Linux, OSX

Syscalls are used to access OS managed resources

- processes, files, memory, devices, …

UNIX Process Management



```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

```
main () {
  ...
}
```

```
pid = fork();
if (pid == 0)
   exec(...);
else
   wait(pid);
```

https://ece.uwaterloo.ca/~smzahedi/crs/se350/slides/03-management.pdf

UNIX Process Management System Calls

`fork()`

- Create a copy of current process and start it as a child

`execv() / execl() / …`

- Load an executable into the current process and run it

`wait() / waitpid()`

- Wait for a child process to finish

`kill()`

- Send a signal (e.g., `SIGTERM`, `SIGKILL`, `SIGINT`) to another process

The Parent of a Process

- Each process (with some exceptions) has a parent process (indicated by `ppid` – parent process identifier)

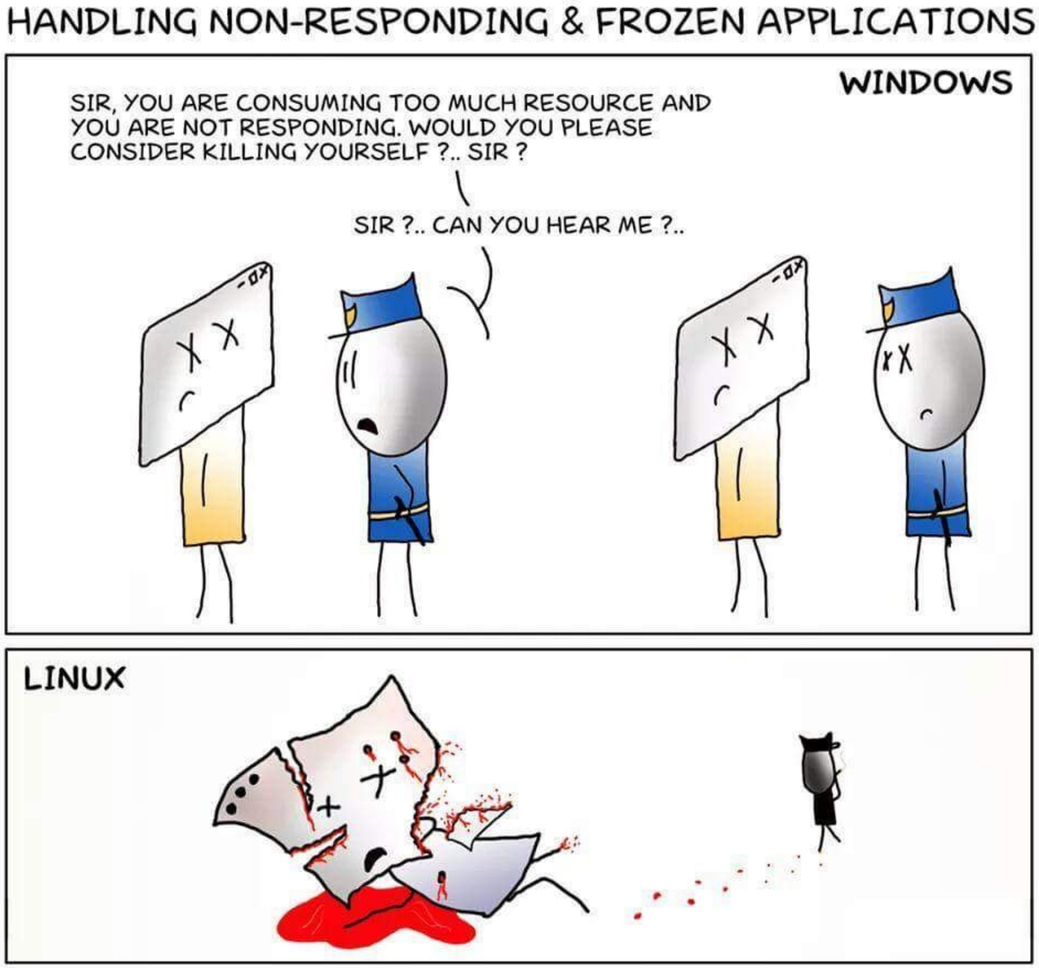
- Can we get this information within a program?
  - YES!
  - Use `getpid()` and `getppid()` `libc` functions defined in `unistd.h`

```
// c++ -o main main.cpp
#include <unistd.h>
#include <iostream>

int main(void) {
  std::cout << "my pid: " << getpid()
            << " ppid: " << getppid() << "\n";
}
```

KILLing a Process!

- Run `kill` in the terminal (run kill with -KILL)


HANDLING NON-RESPONDING & FROZEN APPLICATIONS
WINDOWS
SIR, YOU ARE CONSUMING TOO MUCH RESOURCE AND YOU ARE NOT RESPONDING. WOULD YOU PLEASE CONSIDER KILLING YOURSELF ?.. SIR ?
SIR ?.. CAN YOU HEAR ME ?..
LINUX

https://www.reddit.com/r/linuxmasterrace/comments/3y42qz/killing_a_non responding_process/

Creating a Process - fork() system call

Forks an execution of the process

- after a call to `fork()`, a new process is created (called child)
- the original process (called parent) continues to execute concurrently
- in the parent, `fork()` returns the process `id` of the child that was created
- in the child, `fork()` return `0` to indicate that this is a child process
- The parent and child are independent

Man(ual) Page

- `man 2 fork`

exec() – executing a program in a process

`exec()` series of functions are used to start another program in the current process

- after a call to `exec()` the current process is replaced with the image of the specified program
- different versions allow for different ways to pass command line arguments and environment settings
- `int execv(const char *file, char *const argv[ ])`
  - `file` is a path to an executable
  - `argv` is an array of arguments. By convention, `argv[0]` is the name of the program being executed

Man page

- `man 3 exec`

kill() – sending a signal

A process can send a signal to any other process

- usually the parent process sends signals to its children
- `int kill(pid_t pid, int sig)`
  - send a signal `sig` to a process `pid`
- useful signal: `SIGTERM`
  - asks a process to terminate

When a parent process exits, the children processes are terminated

It’s a good practice to kill and wait for children to terminate before exiting

Man page

- `man 2 kill`

Signals

- A special message sent to a process
- Signals are asynchronous
- Different types of signals (defined in `signum.h`)
  - `SIGTERM`: Termination
  - `SIGINT`: Terminal interrupt (Ctrl+C)
  - `SIGKILL`: Kill (can't be caught or ignored)
  - `SIGBUS`: BUS error
  - `SIGSEGV`: Invalid memory segment access
  - `SIGPIPE`: Write on a pipe with no reader, Broken pipe
  - `SIGSTOP`: Stop executing (can't be caught or ignored)
- Handling a signal:
  - Default *disposition*
  - Signal handler procedure
- Sending signal from one process to another process (`SIGTERM`, `SIGKILL`)

waitpid() – Waiting for a child

A parent process can wait for a child process to terminate

- `pid_t waitpid(pid_t pid, int *status, int options)`
  - block until the process with the specified `pid` terminates
  - the return code from the terminating process is placed in `status`
  - `options` control whether the function blocks or not
    - 0 is a good choice for options

Man page

- `man 2 waitpid`

pipe() and dup2() – Inter-Process Communication

`pipe()` creates a ONE directional pipe

- two file descriptors: one to write to and one to read from the pipe
- a process can use the pipe by itself, but this is unusual
- typically, a parent process creates a pipe and shares it with a child, or between multiple children
- some processes read from it, and some write to it
  - there can be multiple writers and multiple readers
    - although multiple writers is more common

`dup2()` duplicates a file descriptor

- used to redirect standard input, standard output, and standard error to a pipe (or another file)
- `STDOUT_FILENO` is the number of the standard output

Man pages

- `man 2 pipe`
- `man 2 dup2`

getopt() – processing CLI options

flag with an argument

positional argument

CLI – comr flag inte

```
$ foo -s -t 10 bar.txt baz.txt
```

At a start of the program, main(argc, argv) is called, where

- `argc` is the number of CLI arguments
- `argv` is an array of 0 terminated strings for arguments
  - e.g., argv[0] is “foo”, argv[1] is “-s”, argv[2] is “-t”, argv[2] is “10”, …

`getopt()` is a library function to parse CLI arguments

- `getopt(argc, argv, “st:”)`
- input: arguments and a string describing desired format
- output: returns the next argument and an option value
- see example in `using_getopt.cpp`

/dev/urandom – Really Random Numbers

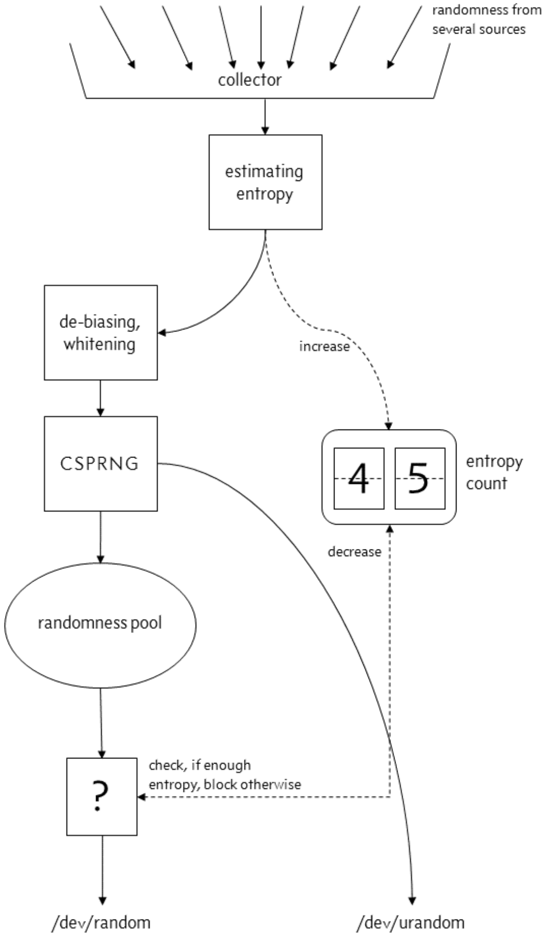
`/dev/urandom` is a special file (device) that provides supply of “truly” random numbers

”infinite size file” – every read returns a new random value

To get a random value, read a byte/word from the file

see `using_rand.cpp` for an example

Have to use it for Assignment 3!



https://www.2uo.de/myths-about-urandom/