ECE353 Lectures

Hanhee Lee

February 10, 2025

Contents

Summary: Lecture slides have all the content, lectures are just a summary of the slides, cheatsheet is a summary of the lectures. On second thoughts, I'm just going to make a cheatsheet for this course as the LaTeX lecture slides is sufficient and I should focus on the cheatsheet.

1 Review

1.1 Converting Between Binary, Hexadecimal, and Decimal

Process:

1. **Binary to Decimal:**
   (a) Write down the binary number.
   (b) Assign place values, starting from 2^0 on the rightmost digit.
   (c) Multiply each binary digit by its corresponding power of 2.
   (d) Add all the results together to get the decimal equivalent.
2. **Decimal to Binary:**
   (a) Divide the decimal number by 2.
   (b) Record the remainder (0 or 1).
   (c) Repeat the division process with the quotient until the quotient is 0.
   (d) Write the remainders in reverse order to obtain the binary equivalent.
3. **Binary to Hexadecimal:**
   (a) Group the binary number into groups of 4 digits, starting from the right. Add leading zeros if necessary.
   (b) Convert each 4-digit binary group to its hexadecimal equivalent using the binary-to-hex mapping (e.g., $0000 = 0$, $0001 = 1$, $1110 = E$).
   (c) Combine the hexadecimal digits to get the hexadecimal equivalent.
4. **Hexadecimal to Binary:**
   (a) Write down each hexadecimal digit.
   (b) Replace each hexadecimal digit with its 4-bit binary equivalent.
   (c) Combine the binary groups to get the binary equivalent.
5. **Decimal to Hexadecimal:**
   (a) Divide the decimal number by 16.
   (b) Record the remainder as a hexadecimal digit (0–9 or A–F).
   (c) Repeat the division process with the quotient until the quotient is 0.
   (d) Write the remainders in reverse order to obtain the hexadecimal equivalent.
6. **Hexadecimal to Decimal:**
   (a) Write down the hexadecimal number.
   (b) Assign place values, starting from 16^0 on the rightmost digit.
   (c) Multiply each hexadecimal digit by its corresponding power of 16, converting any letters (A–F) to decimal values (A=10, B=11, etc.).
   (d) Add all the results together to get the decimal equivalent.

1.2 Little-endian and Big-endian

Definition:

- **Little-endian:** In the little-endian format, the least significant byte (LSB) of a multi-byte data value is stored at the lowest memory address, and the most significant byte (MSB) is stored at the highest memory address.
- **Big-endian:** In the big-endian format, the most significant byte (MSB) of a multi-byte data value is stored at the lowest memory address, and the least significant byte (LSB) is stored at the highest memory address.

Example:

- For example, the hexadecimal value `0x12345678` would be stored in memory as:

```
78 56 34 12
```

- For example, the hexadecimal value `0x12345678` would be stored in memory as:

```
12 34 56 78
```

1.3 Memory

Summary: Table, int*, &a, int**a, *a, int[5], etc.

2 Why Systems Software? Kernels

Summary:

2.1 Useful Terminal Commands

Summary:
- `./hello-world-linux-aarch64` to run hello world.
- `readelf -a <FILE>` to see the ELF header.
- `strace <PROGRAM>` to trace all the system calls a process makes on Linux.

2.2 Three OS Concepts

Definition:
1. **Virtualization:** Share one resource by mimicking multiple independent copies.
2. **Concurrency:** Handle multiple things happening at the same time.
3. **Persistence:** Retain data consistency even without power.

2.3 OS Manages Resources

Definition: Insert picture.

2.4 Program

Definition: A file containing all the instructions and data required to run.

2.5 Process:

Definition: An instance of running a program.

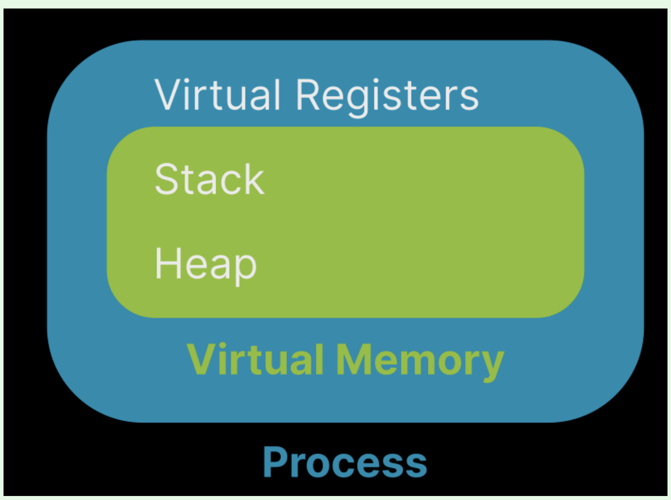

Figure 1: Process

2.5.1 Basic Requirements for a Process

Definition: Insert picture w/ virtual memory.

2.6 Process (Abstraction)

2.6.1 Static

Definition: Only able to use the global variable in the current C file.

2.6.2 Motivation for Virtualization

Motivation: How to run two different programs at the same time? Insert code.
- Was the address of local the same b/w 2 processes? Different address in physical memory b/w different processes.
- Was the address of global the same b/w 2 processes? Same address in physical memory b/w different processes, but uses virtual memory.
- What else may be needed for a process?

Warning: Local variables are stored on the stack.

2.6.3 Does the OS allocate different stacks for each process?

Definition: The stacks for each process need to be in physical memory. One option is the operating system just allocates any unused memory for the stack.
-

2.6.4 What about global variables?

Definition: The compiler needs to pick an address (random) for each variable when you compile.
- What if we had a global registry of addresses? Impossible (too much space and know memory addresses ahead of time).

Summary:
- The kernel is the part of the operating system (OS) that interacts with hardware (it runs in kernel mode).
- System calls are the interface between user and kernel mode:
  - Every program must use this interface!
- File format and instructions to define a simple "Hello world" (in 168 bytes):
  - Difference between API and ABI.
  - How to explore system calls.
- Different kernel architectures shift how much code runs in kernel mode.

FAQ:
- What is difference b/w printf and write?

2.7 File Descriptor (Abstraction)

Motivation: Since our processes are independent, we need an explicit way to transfer data.

Definition:
1. **IPC:** Inter-process communication is transferring data b/w two processes.

2. **File Descriptor:** A resource that users may either read bytes from or write bytes to (identified by an index stored in a process).
   - e.g. File or terminal.
   - e.g. 0 is standard input, 1 is standard output, and 2 is standard error.

2.8 System Calls

Definition: System calls are the interface b/w user and kernel mode.

2.8.1 System Calls Make Requests to the Operating System

Definition:

```c
ssize_t write(int fd, const void *buf, size_t count);
```

- Description: writes bytes from a byte array to a file descriptor
  - fd: the file descriptor
  - buf: the address of the start of the byte array (called a buffer)
  - count: how many bytes to write from the buffer

```c
void exit_group(int status);
```

- Description: exits the current process and sets an exit status code
  - status: the exit status code (0–255)

Example: Hypothetical "Hello World" Program

```c
void _start(void) {
    write(1, "Hello world\n", 12);
    exit_group(0);
}
```

Warning: System calls uses registers, while C is stack based.

2.9 API Tells You What and ABI Tells You How

Definition:
- Application Programming Interface (API) abstracts the details and describes the arguments and return value of a function.
- Application Binary Interface (ABI) specifies the details, specifically how to pass arguments and where the return value is.

2.10 Magic

Definition: The "magic bytes" refer to the first 4 bytes of a file that uniquely identify the file format.

2.10.1 Programs on Linux Use the ELF File Format

Definition: Executable and Linkable Format (ELF) specifies both executables and libraries.
- Always starts with the 4 bytes: 0x7F 0x45 0x4C 0x46 or with ASCII encoding: DEL 'E' 'L' 'F'

Example: Hello World ELF File

1. **168 Byte Program:**

- Tells the OS to load the entire executable file into memory at address `0x10000`.
- The file header is 64 bytes, and the "program header" is 56 bytes (120 bytes total).
- The next 36 bytes are instructions, followed by 12 bytes for the string:
  - `"Hello world\n"`
  - Instructions start at `0x10078` (`0x78` is 120).
  - The string (data) starts at `0x1009C` (`0x9C` is 156).

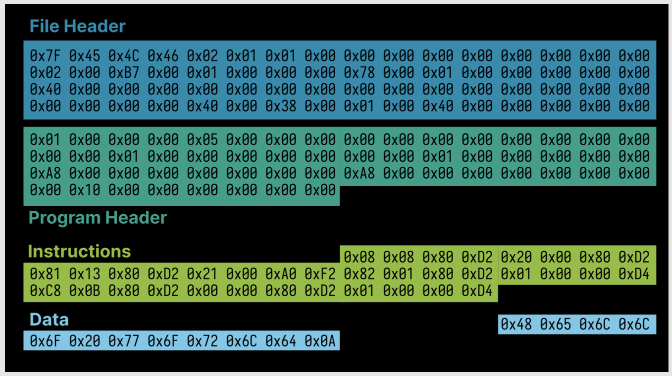

Figure 2: ELF File Division

2. **C Program:** Takes 500 bytes.
3. **Python Program:** Takes 2000 bytes.
4. **Java Program:** Takes 2000000 bytes.

2.11 Kernel

Definition: Kernel is a core part of the operating system that interacts with hardware that runs in kernel mode.

2.11.1 Kernel Mode

Definition: Kernel mode is a privilege level on your CPU that gives access to more instructions.

2.11.2 Levels of Privelege

Definition:

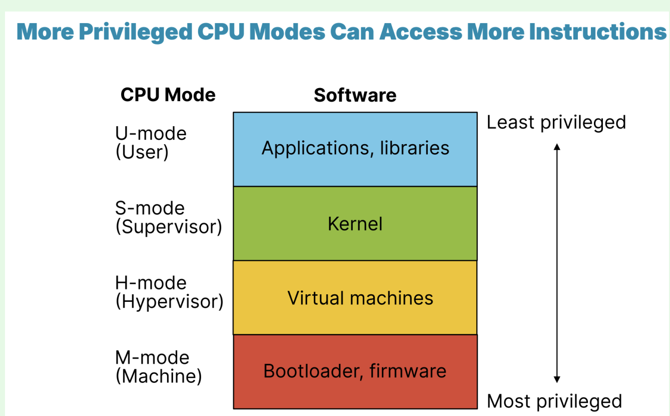

Figure 3: Levels of Privelege

2.11.3 System Calls Transition Between User and Kernel Mode

Definition:

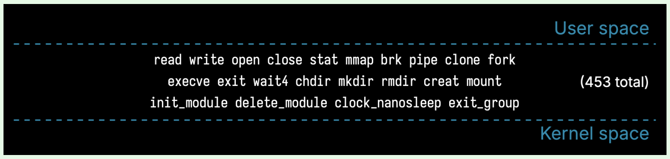

Figure 4: System Calls Transition

2.11.4 Different Tpyes of Kernel Architectures

Definition:

- **Monolithic Kernel:** All the services are in the kernel.

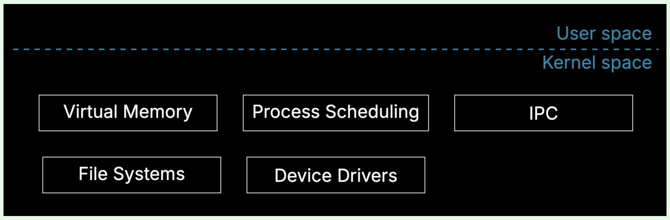

Figure 5: Monolithic Kernel

- **Microkernel:** Only the essential services are in the kernel.

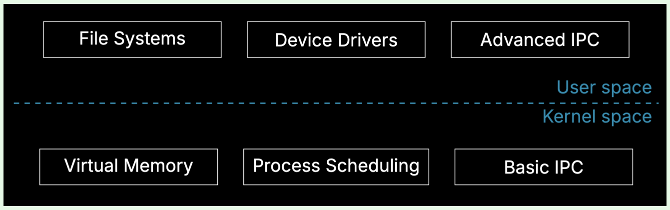

Figure 6: Microkernel

- **Hybrid Kernel:** A mix of monolithic and microkernel.
- **Nanokernel and picokernel:** Even smaller services than microkernel.

Warning: Short answer question.

3 Libraries

Summary:
- `ldd <executable>`: Shows which dynamic libraries are used by the executable.
- `-Db_sanitize=address`: Add the flag to Meson to detect memory leaks, which was built into the compiler, but you have to recompile.
- `valgrind <executable>`: Detect memory leaks from malloc and free.
  - **Warning:** GNU C library (libc.so) may allocate memroy for its own uses, so it doesn't bother to free.
- Know the high-level rules of ABI changes without API changes.

FAQ:

3.1 What is an Operating System?

Definition: An operating system consists of a kernel and libraries required for your application.

Warning: OS have different libraries for different applications.

3.2 Normal Compilation in C

Definition:

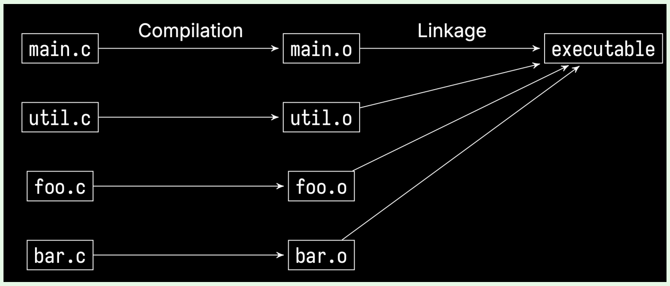

Figure 7:

Notes: Object files (.o) are ELF files with code for functions.

3.3 Static Libraries and Dynamic Libraries

3.3.1 Static Libraries

Definition:

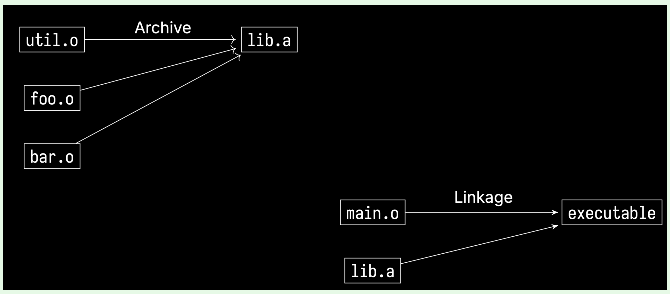

Figure 8:

- Static libraries are included at link time.

Notes:
- Put all .o files into a single .a file (i.e. library), then link the library with the application.

3.3.2 Dynamic Libraries

Motivation: C standard library (.so) is a dynamic library that is a collection of .o files containing funciton defintiions.

Definition:

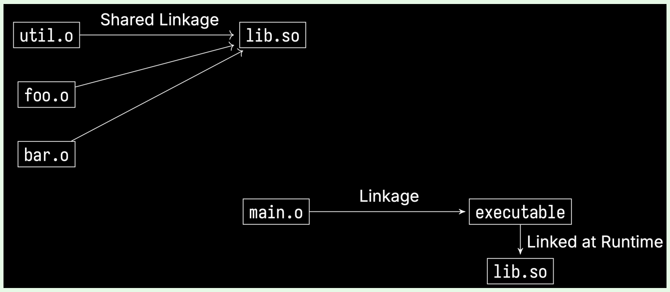

Figure 9:

- Dynamic libraries are included at runtime.
- Dynamic libraries are for reusable code. Multiple applications can use the same library.

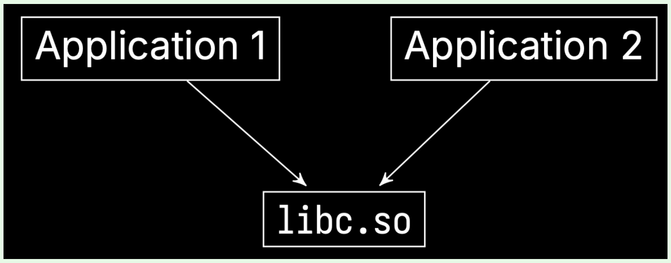

Figure 10:

  - The operating system only loads `libc.so` in memory once, and shares it.
- Dynamic libraries allow easier debugging.
  - Control dynamic linking with environment variables `LD_LIBRARY_PATH` and `LD_PRELOAD`.

3.3.3 Comparison of Static and Dynamic Libraries

Notes:
- **Static:** Statically linking basically copies the `.o` files directly into the executable
  - Statically linking prevents re-using libraries (commonly used libraries have many duplicates)
  - Any updates to a static library requires the executable to be **recompiled**
- **Dynamic:** The executable has a reference to the dynamic library
  - Dynamic libraries updates can break executables.
    - A dynamic library update may subtly change the ABI causing a crash.

Example:

1. **Dynamic Library ABI Changes:**
   - **Given:** Consider the following in a dynamic library:
     - A `struct` with multiple fields corresponding to a specific data layout (C ABI).
     - An executable accesses the fields of the `struct` used by a dynamic library.
   - **Problem:** Now, if a dynamic library reorders the fields:
     - The executable uses the old offsets and is now wrong.
   - **Note:** This is OK if the dynamic library never exposes the fields of a `struct`.
2. **C Uses a Consistent ABI for Structs:**
   - `struct`s are laid out in memory with the fields matching the declaration order.
   - C compilers ensure the ABI (Application Binary Interface) of `struct`s is consistent for an architecture.
   - Consider the following structures:
     - Library v1:

```c
struct point {
    int y;
    int x;
};

```

     - Library v2:

```c
struct point {
    int x;
    int y;
};

```

   - For v1, the `x` field is offset by 4 bytes from the start of `struct point`'s base.
   - For v2, it is offset by 0 bytes, and this difference will cause problems.
3. **After Compilation the Translation Differs for Each Version:**

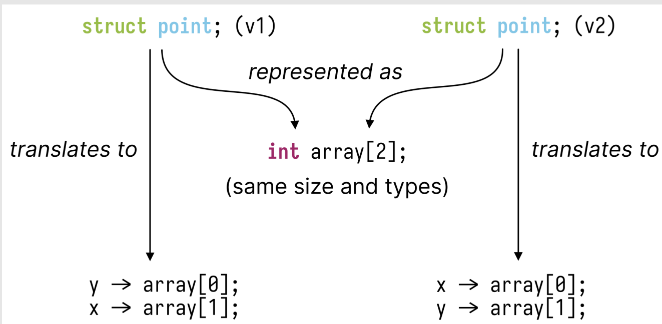

Figure 11:

4. **Point API Has 4 Functions:**

libpoint.so

```c
struct point *point_create(int x, int y);
int point_get_x(struct point *p);
int point_get_y(struct point *p);
void point_destroy(struct point *p);
```

Figure 12:

5. **ABI Stable Code Should Always Print "1,2" for Both Lines:**

```c
#include <stdio.h>
#include <stdlib.h>

#include "point.h"

int main(void) {
  struct point *p = point_create(1, 2);

  printf("point (x, y) = %d, %d (using library)\n",
         point_get_x(p), point_get_y(p));

  printf("point (x, y) = %d, %d (using struct)\n", p->x, p->y);

  point_destroy(p);
  return 0;
}
```

Figure 13:

6. **Mismatched Versions Don't Agree on the Location of X and Y:**

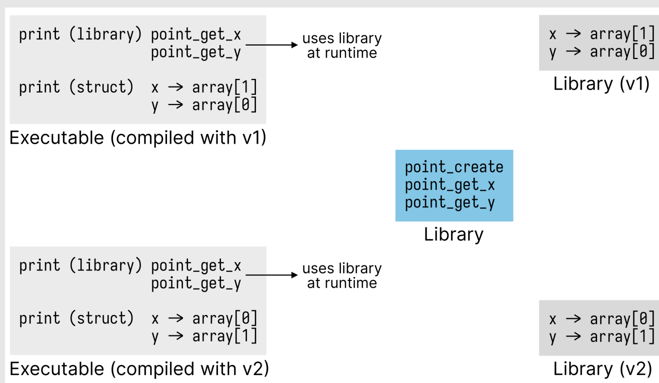

Figure 14:

7. **Takeaways:** The definition of struct point in both libraries is different. Order of x and y change.
   - Our code works correctly if the compiled and linked versions match. If you expose a struct it becomes part of your ABI.
   - A proper stable ABI would hide the struct from point.h

3.4 Semantic Versioning

Definition: Given a version number `MAJOR.MINOR.PATCH`, increment the:
- **MAJOR** version when you make incompatible API/**ABI** changes.
- **MINOR** version when you add functionality in a backwards-compatible manner.
- **PATCH** version when you make backwards-compatible bug fixes.

3.5 System Calls are Rare in C

Definition: Most system calls have corresponding function calls in C, but may:
- Set `errno`
- Buffer reads and writes (reduce the number of system calls)
- Simplify interfaces (function combines two system calls)
- Add new features

3.5.1 C exit

Definition:
- **System Call Exit (or exit_group):** Program stops at that point.
- **C Exit:** Feature to register functions to call on program exit (e.g. `atexit`).
  - i.e. Runs the function when the program exits.

Example:

```c
#include <stdio.h>
#include <stdlib.h>

void fini(void) {
    puts("Do fini");
}

int main(void) {
    atexit(fini);
    puts("Do main");
    return 0;
}
```

- Return 0 is the same as calling `exit(0)`.

4 Processes

Summary:
- User space (applications, libraries) vs. kernel space (OS)
  - User space to kernel space: system calls
  - Libraries: Performs the system calls in this course.
- Open file descriptors: 0,1,2 go to Terminal (stdin, stdout, stderr)
- Process: Virtual registers, virtual memory (stacks, heap), open file descriptors.

FAQ:
- What does the terminal do with the open file descriptors?
- What does the file 1, file 2, and terminal do in the initial summary?
- Slide 3: What is PCB? Contains all the info about the process.
- Slide 3: What is a PID? Keeps track of a running process.
- Slide 4: What do the different componetns of the process state diagram mean? REWATCH
  - Created: Process is created.
  - Ready: Process is ready to run.
  - Running: Process is running, but it goes back to ready because it is waiting for something.
  - Blocked: Process is blocked, which goes to
- Slide 5: What is the proc filesystem?
  - The "proc" filesystem is a special filesystem in Unix-like operating systems that presents information about processes and other system information in a hierarchical file-like structure.
  - It is commonly mounted at '/proc'.
  - It allows users and applications to access kernel information in a structured and readable way.
- Slide 7: What is parent and child process?
  - Parent process: The original process that creates a new process.
  - Child process: The new process created by the parent process.
  - The child process is a copy of the parent process, but with a unique process ID.
- Slide 8: How to distinguish between parent and child process? 0 is returned in the child process, and the process ID of the child process is returned in the parent process.
- Slide 9: What is POSIX system? POSIX (Portable Operating System Interface) is a family of standards specified by the IEEE for maintaining compatibility between operating systems.
- Slide 9: What is man? is a command to display documentation, and use number.
- Slide 8: What happens when you call fork? Only the parent process calls the fork, and the child process is created.
- Slide 8: Will parent run first then child? No, since there two independent processes, the order of execution is not guaranteed.

4.1 Process: Adding onto the Process

Definition:

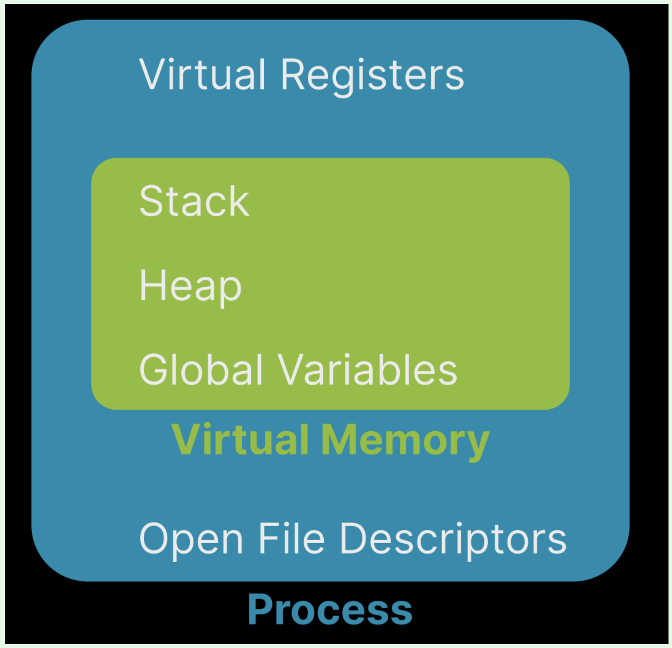

Figure 15: Process

4.2 Process Control Block (PCB)

Definition: Contains all information.
- Process state
- CPU registers
- Scheduling information
- Memory management information
- I/O status information
- Any other type of accounting information

Warning: Each process gets a unique process ID (pid) to keep track of it.

Example: In Linux, this is the `task_struct` structure.

4.3 Process State Diagram

Definition:

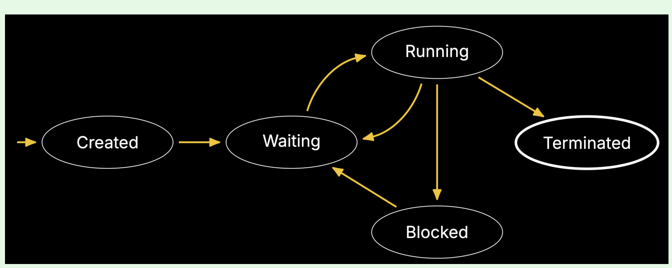

Figure 16: Process State Diagram

Notes: Waiting $\iff$ Ready.

4.4 proc Filesystem

Definition: Read process state using the "proc" file system.

4.5 Create processes from scratch

4.5.1 Fork

Definition: `fork()` creates a new process, a copy of the current one.

```
int fork(void);
```

- Returns the process ID of the newly created child process:
  - -1: on failure
  - 0: in the child process
  - >0: in the parent process

4.6 Clone processes