CS 351: Systems Programming — Reference Material
Illinois Institute of Technology

Karl Hallsby

Last Edited: November 6, 2020

Contents

List of Theorems **iv**

1 C Programming **1**
- 1.1 Memory Allocation . . . 1
  - 1.1.1 `malloc` . . . 1
  - 1.1.2 `calloc` . . . 1
  - 1.1.3 `realloc` . . . 1
  - 1.1.4 `free` . . . 1

2 Processes **2**
- 2.1 Prevent Process Disruption . . . 2
- 2.2 Required Hardware . . . 3
- 2.3 Process Scheduling . . . 3
  - 2.3.1 Priority Scheduling . . . 3
- 2.4 Exceptional Control Flow . . . 3
  - 2.4.1 Synchronous Exceptions . . . 4
    - 2.4.1.1 Traps . . . 4
    - 2.4.1.2 Faults . . . 4
    - 2.4.1.3 Aborts . . . 4
  - 2.4.2 Asynchronous Exceptions . . . 4

3 Process Management **5**
- 3.1 Making Processes, `fork` . . . 5
  - 3.1.1 Using Processes . . . 5
  - 3.1.2 `fork` Fails . . . 7
  - 3.1.3 `fork` Bomb . . . 7
- 3.2 Terminating Processes, `exit` . . . 7
  - 3.2.1 `atexit` . . . 8
  - 3.2.2 Zombie Processes . . . 8
- 3.3 Getting Values from Processes, `wait` . . . 8
  - 3.3.1 Synchronization Mechanism . . . 8
- 3.4 Changing the Running Program, `exec` . . . 8
- 3.5 Signals . . . 10
  - 3.5.1 Signal Lifecycle . . . 12
  - 3.5.2 Process Groups . . . 13
  - 3.5.3 Registering Signal Handlers . . . 13
  - 3.5.4 Adjusting Signal Masks . . . 13

4 Input/Output (I/O) **13**
4.1 I/O Devices . . . 15
4.2 Filesystem . . . 15
4.3 Files . . . 16
4.4 System-Level I/O API . . . 17
4.4.1 Why up to `nbytes`? . . . 17
4.4.2 Non-Blocking Reads . . . 18
4.5 Buffering . . . 18
4.5.1 Benefits . . . 18
4.6 File I/O API . . . 18
4.7 Generalizing . . . 18
4.7.1 Easing Your Life . . . 19
4.7.1.1 Stream Objects . . . 19
4.8 Performing Both Input and Output . . . 19
4.9 Summary . . . 19

5 Inter-Process Communication **19**
5.1 Mechanisms . . . 19
5.1.1 Signals . . . 20
5.1.2 (Regular) Files . . . 20
5.1.3 Shared Memory . . . 20
5.1.3.1 API . . . 20
5.1.3.2 Synchronizing Shared Memory . . . 20
5.1.4 Pipes . . . 20
5.1.4.1 Named Pipes . . . 21
5.1.4.2 Unnamed Pipes . . . 21
5.1.5 File Locks and Semaphores . . . 21
5.1.5.1 File Locks . . . 21
5.1.5.2 Semaphores . . . 22
5.1.5.3 Spinlocks . . . 23
5.1.6 Sockets . . . 23
5.2 Challenges . . . 23
5.2.1 Link/Endpoint Creation . . . 23
5.2.2 Data Transmission . . . 23
5.2.3 Data Synchronization . . . 23

6 Sockets **23**
6.1 Communication Protocols . . . 23
6.1.1 Physical . . . 24
6.1.2 Data Link . . . 24
6.1.3 Network . . . 24
6.1.4 Transport Layer . . . 24
6.1.5 Session and Presentation . . . 24
6.1.6 Application . . . 24
6.2 Middleware Protocols . . . 24
6.3 Sockets . . . 24
6.3.1 Message-Oriented *Transient* Communication . . . 25
6.3.2 Berkeley Socket . . . 25
6.3.2.1 Socket Usage . . . 25
6.3.3 Message-Oriented *Persistent* Communication . . . 25
6.4 MPI (Message Passing Interface) . . . 25
6.5 Remote Procedure Calls . . . 28
6.5.1 Design Issues . . . 28
6.5.2 Steps . . . 29
6.5.3 Marshaling . . . 29

A Computer Components **30**
A.1 Central Processing Unit . . . 30
A.1.1 Registers . . . 30
A.1.2 Program Counter . . . 30
A.1.3 Arithmetic Logic Unit . . . 30
A.1.4 Cache . . . 30
A.2 Memory . . . 30
A.2.1 Stack . . . 30
A.2.2 Heap . . . 32
A.3 Disk . . . 32
A.4 Fetch-Execute Cycle . . . 32

List of Theorems

1 Defn (Process) . . . 2
2 Defn (Program) . . . 2
3 Defn (Process Control Block) . . . 2
4 Defn (Syscall) . . . 2
5 Defn (Context Switch) . . . 3
6 Defn (Process Scheduler) . . . 3
7 Defn (Starvation) . . . 3
8 Defn (Exceptional Control Flow) . . . 3
9 Defn (Synchronous Exception) . . . 4
10 Defn (Trap) . . . 4
11 Defn (Fault) . . . 4
12 Defn (Abort) . . . 4
13 Defn (Asynchronous Exception) . . . 4
14 Defn (Interrupt Handler) . . . 4
15 Defn (Zombie Process) . . . 8
16 Defn (Reap) . . . 8
17 Defn (Signal) . . . 10
18 Defn (Reentrant) . . . 12
19 Defn (Bitmap) . . . 13
20 Defn (Process Group) . . . 13
21 Defn (Signal Handler) . . . 13
22 Defn (Block Device) . . . 15
23 Defn (Character Device) . . . 15
24 Defn (File) . . . 16
25 Defn (`inode`) . . . 16
26 Defn (Hard Link) . . . 16
27 Defn (Symlink) . . . 16
28 Defn (`vnode`) . . . 16
29 Defn (File Description) . . . 16
30 Defn (File Descriptor) . . . 16
31 Defn (Buffering) . . . 18
32 Defn (Inter-Process Communication) . . . 19
33 Defn (Pipe) . . . 20
34 Defn (Named Pipe) . . . 21
35 Defn (Unnamed Pipe) . . . 21
36 Defn (Lock) . . . 21
37 Defn (Mutex) . . . 22
38 Defn (Semaphore) . . . 22
39 Defn (Socket) . . . 23
A.1.1 Defn (Central Processing Unit) . . . 30
A.1.2 Defn (Register) . . . 30
A.1.3 Defn (Program Counter) . . . 30
A.2.1 Defn (Memory) . . . 30
A.2.2 Defn (Volatile) . . . 30
A.2.3 Defn (Call Stack) . . . 30
A.2.4 Defn (Stack Frame) . . . 30
A.2.5 Defn (Dynamic Link) . . . 31
A.2.6 Defn (Local Variable) . . . 31
A.2.7 Defn (Temporary Variable) . . . 31
A.2.8 Defn (Static Link) . . . 31
A.2.9 Defn (Function Argument) . . . 31
A.2.10 Defn (Return Address) . . . 32
A.2.11 Defn (Garbage Collection) . . . 32
A.2.12 Defn (Frame Pointer) . . . 32
A.2.13 Defn (Stack Pointer) . . . 32
A.2.14 Defn (Class Descriptor) . . . 32
A.2.15 Defn (Heap) . . . 32
A.3.1 Defn (Non-Volatile) . . . 32

List of Listings

1 Exceptional Control Flow Example . . . 4
2 `pid` Definition and `fork()` Declaration . . . 5
3 `fork()` Usage . . . 6
4 `fork()` Usage . . . 6
5 Using `fork()`, Performing Separate Actions . . . 6
6 Post- `fork` , Child Finishes First . . . 7
7 Post- `fork` , Parent Finishes First . . . 7
8 Using `errno` to get Error Return Codes . . . 7
9 `fork()` Bomb . . . 7
10 `wait()` Usage . . . 9
11 Using `wait()` as a Synchronization Tool . . . 9
12 `exec()` Usage . . . 10
13 Using `fork()` and `exec()` . . . 11
14 Using Signals . . . 11
15 Registering Signal Handlers . . . 14
16 Using `sigprocmask` . . . 14
17 `server.c` as an Example for Server-side Socket Programming . . . 26
18 `client.c` as an Example for Client-side Socket Programming . . . 27

1 C Programming

C is one of the lowest "high level" languages you can use today. It provides very minimal abstractions from hardware and assembly code, but allows you to relatively good typechecked code.

1.1 Memory Allocation

Because C is a language that does not provide many abstractions, it also requires the programmer to remember and manage their memory usage. So, **YOU** must be the one to manage the memory, there is **NO** built-in garbage collector for you to use.

Memory allocation is done on the heap of the program's execution space in memory. When you allocate memory in your program, you are actually requesting the operating system to give you the memory you want.

1.1.1 `malloc`

This is the simplest function of all possible memory allocation functions. `malloc`:

- Takes one argument:
  1. The number of bytes to allocate.
- Returns a **POINTER** to the front of the allocated memory.

`malloc` ***DOES NOT*** initialize memory, so it will be garbage.

1.1.2 `calloc`

This is quite similar to malloc. `calloc`:

- Takes 2 arguments:
  1. The number of spaces to allocate, for example the number of elements in an array.
  2. The number of bytes to allocate, for the type being stored.
- Returns a **POINTER** to the front of the allocated memory.

`calloc` ***ZEROS*** memory, so this does have a slight performance penalty.

1.1.3 `realloc`

`realloc` is used to **REALLOCATE** an existing memory location.

- Takes 2 arguments:
  1. The pointer to the memory location previously allocated with either `malloc` or `calloc`.
  2. The amount of memory to reallocate, in bytes.
- If the `NULL` pointer is passed to `realloc`, it will behave exactly like `malloc`.
- Returns a **POINTER** to the front of the reallocated memory

1.1.4 `free`

`free` is used to free memory that was previously allocated, removing from the programming space entirely.

- Takes 1 argument:
  1. A pointer to the memory to be deallocated.
- Returns `void`.

2 Processes

Processes are the fundamental unit of computation within an operating system.

Defn 1 (Process)**.** A *process* is a Program in execution. A process carries out the computation that we specify. A process contains:

- Code (`text`) of your program.
- Runtime data (Global, local, dynamic variables)
- Registers:
  - Program Counter (`PC`)
  - Stack Pointer (`SP`)
  - Frame Pointer (`FP`)
- Process Control Block

Defn 2 (Program)**.** A *program* is the binary image stored at some file location on the storage medium. The program is read into memory, and then is used to start a Process that runs that program.

Processes require both a **predictable** and **logical** control flow. This means:

- The Process must start somewhere, typically defined to be `main`.
- Nothing can disrupt a program mid-execution.
  - This is further discussed in Section 2.1.

2.1 Prevent Process Disruption

The easiest way to prevent a Process from having its control flow being interrupted is for the process to "own" the CPU for the entire duration of the process's execution. However, this means:

- No other process can run on this core
- This prevents efficient multi-Process/multitasking systems
- Malicious or poorly written program can "take over" the CPU
- An idle process (for example, waiting for user input) will underutilize the CPU

For the operating system to simulate this seamless logical control flow, we use all of the information used to make a Process, and need a Process Control Block.

Defn 3 (Process Control Block)**.** The *Process Control Block* (*PCB*) contains additional metadata about a Process. This includes:

- Process ID (`PID`)
- CPU Usage
- Memory Usage
- Pending Syscalls

The Process Control Block is sued to allow Processes to be interrupted, saved, and moved off a core. This allows the operating system to schedule processes according to some algorithm.

Defn 4 (Syscall)**.** A *syscall*, short for a *system call* is a way for a user-level Process to perform some computation that the operating system kernel restricts. Some common syscalls are:

- Opening a file
- Reading a file
- Writing a file
- Closing a file
- Creating a process
- Changing the process's binary
- Reading from the network
- Writing to the network

2.2 Required Hardware

Interrupting the execution of a Process requires some hardware support to be possible and efficient. We need 3 things:

1. A hardware mechanism to periodically interrupt the current Process to change execution to the operating system.
   - This is usually the *periodic clock interrupt*.
2. An operating system procedure to decide which processes to run, and in which order.
   - This is the operating system's Process Scheduler.
3. A routine for seamlessly switching between processes.
   - This is called a Context Switch.
   - Relatively speaking, these are expensive to perform.
   - **These are external to a Process's logical control flow**.
   - This forms part of the process of Exceptional Control Flow.
   - A Context Switch makes no guarantee about if and/or when this Process will start running again.
   - A Context Switch is the only way to invoke Syscalls.

To schedule Processes onto one of possibly many CPUs, there are programs called Process Schedulers.

Every time the Process Scheduler schedules a new Process, or when a process needs to perform a Syscall, or a kernel-level exception occurs, a Context Switch is made.

Defn 5 (Context Switch)**.** A *context switch* is the process of interrupting a Process, saving its current state, and scheduling something else to run on that CPU. This is done whenever a Syscall is made, and happens sometimes during a process's lifetime.

Context Switches form a core part of the Exceptional Control Flow.

2.3 Process Scheduling

Process scheduling is the process by which a process is put "scheduled" onto a particular CPU, according to some criteria. There are any number of scheduling algorithms, each of which optimizes for certain cases, and may yield a different order of process execution.

Defn 6 (Process Scheduler)**.** The *process scheduler* is an operating system component that chooses which Process to run next. It chooses this according to some algorithm, each of which might change the order of process execution.

One such Process Scheduler is the Priority Scheduling algorithm.

2.3.1 Priority Scheduling

Priority scheduling involves placing a priority on every Process that can be scheduled. Then, the process with the highest priority is chosen first, working our way down to the lowest priority. This is, partly, the setup most modern operating systems take today.

However, priority scheduling creates new issues that must be dealt with. One of these is Starvation.

Defn 7 (Starvation)**.** *Starvation* is when something that needs a resource to function does not receive this resource.

In this case, a lower-priority Process can experience Starvation if only higher-priority processes are present, and continually steal CPU execution time.

2.4 Exceptional Control Flow

To illustrate the power of Exceptional Control Flow, we will use an example piece of code, Listing 1.

Defn 8 (Exceptional Control Flow)**.** *Exceptional control flow* is designated by the fact that most of the computation involved is done in response to exceptions or special events. Which one depends on what "exceptional" means in that circumstance.

If there was an exception of any kind during the execution of the `while`-loop, then the process would be interrupted by the kernel, and Context Switched out. From there, the exception would be handled (if possible), and then the operating system would continue execution (if possible). While the execution of the Process was non-sequential, the process **is not aware** of this fact. **ALL** of the process's state is saved **before** the context switch, so that when the process is switched back in, it continues executing from the **same** spot as before.

There are 2 kinds of exceptions in operating systems today:

1. Synchronous Exceptions
2. Asynchronous Exceptions

```c
#include <stdio.h>

int main() {
	while (1) {
		printf("Hello world!\n");
	}
	return 0;
}
```

Listing 1: Exceptional Control Flow Example

2.4.1 Synchronous Exceptions

Defn 9 (Synchronous Exception)**.** A *synchronous exception* is one that is caused by the **currently executing** Process. These are usually things that the process wants to bring to attention, or to have handled.

2.4.1.1 Traps

Defn 10 (Trap)**.** A *trap* is a Synchronous Exception caused **by** the currently executing Process. A trap is caused by the process making a Syscall.

Remark 10.1 (Interaction with Scheduling). A Trap will make the Process that made the Syscall Context Switch out of execution. The Process Scheduler makes **no** guarantees **when** the process will run again.

2.4.1.2 Faults

Defn 11 (Fault)**.** A *fault* is an unintentional failure in the Process and may or may not be recoverable.

A short list of common Faults includes:

- Segmentation Fault, Unrecoverable
- Protection Fault, may or may not be recoverable
- Page Fault, Recoverable
- Divide-by-zero, Possibly Recoverable

It's possible to recover from a Fault by having the fault handler fix the problem.
In addition, there may be a Context Switch after this action.

Unrecoverable : **Will** be a Context Switch, as this Process terminates.
Recoverable : **May** be a Context Switch, depending on the Process Scheduler.

2.4.1.3 Aborts

Defn 12 (Abort)**.** An *abort* is an **unintentional** and **unrecoverable** error. The Process is terminated by the operating system. If too many errors accumulate, the OS might terminate itself.

2.4.2 Asynchronous Exceptions

Defn 13 (Asynchronous Exception)**.** An *asynchronous exception*, typically called an *interrupt* is one caused by events **external** to the current Process. On older keyboards that used the PS/2 interface, pressing a key on the keyboard would generate an asynchronous exception. On modern computers, pressing `Ctrl+C` will produce a `SIGKILL` signal, which is another asynchronous exception.

These are typically associated with specific hardware pins on the CPU. A check for Asynchronous Exceptions is performed after **every** CPU cycle. These types of exceptions are handled with/by interrupt handlers.

Defn 14 (Interrupt Handler)**.** An *interrupt handler* takes the current state of the system and the Asynchronous Exception that was just raised, and handles the exception, then returns to another Process.
The steps involved are:

1. Save Context

2. Load OS Context
3. Execute the Interrupt Handler
4. Load context for the next process given by the scheduler
5. Return to the next process

Remark 14.1. These handlers are fairly lightweight, but having more and more interrupts **will** affect performance.

3 Process Management

In almost all modern systems today, there are many, many Processes running "simultaneously". That is in quotes because if you have multiple cores/Central Processing Units in a single package, you actually *can* run multiple processes at once. However, we choose to limit our discussion to single core packages, to simplify our discussions and remove a whole class of issues.

By default, there is only one Process running at the start of a computer's execution. We need ways to make more processes, change what Programs these processes are executing, and a way to wait for these processes to finish and pick up after them.

3.1 Making Processes, `fork`

`fork` creates a **copy** of the current Process. This is our *only* method of creating new processes. The child process is nearly an **exact** duplicate of the parent process, where only some process metadata in the Process Control Block is different. The function prototype for `fork` is shown in Listing 2.

After a `fork`, the parent and child share the same:

- Registers:
  - Program Counter `PC`. The child starts **at the same place in the program as the parent**.
  - Stack Pointer, `SP`
  - Frame Pointer, `FP`
- Open Files

```c
#include <unistd.h>

/* typedef int pid_t */

pid_t fork();
/* Makes a system call to trap to the OS.
 * This requests the OS to create a new process.
 * This is mostly a duplicate of the original. */
```

Listing 2: `pid` Definition and `fork()` Declaration

`fork` returns **twice**.

- Once to the parent Process, with the `PID` of the child (> 0).
- Once to the child process, with the return code of `fork`. A returned value of `0` indicates success; it is a sentinel value.

`pid_t` is a system-wide unique Process IDentifier (`PID`). It is `typedef`-ed from an integer, so normal integer arithmetic and rules apply.

Listing 3 code will print `Hello World!` twice, but in no particular order. The reason that we don't have garbage being printed out to the screen is because the `STDOUT` stream has a lock associated with it, only allowing one Process to use the screen at a time.

Listing 4 will print `Hello World!` four times, but in no particular order. The main parent Process has 2 children, and the parent's **first** child makes another child.

3.1.1 Using Processes

There is usually a split in the logical control flow between the parent and child, making them take different actions. This is possible because the parent and child Processes receive different return values. A simple example of this is shown in Listing 5.

The results from Listing 5 executing is that both print statements are executed. But, you are not guaranteed the order in which they execute. Some orders that exist are:

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

int main(void) {
	fork();
	printf("Hello World!\n");
	return 0;
}
```

Listing 3: `fork()` Usage

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

int main(void) {
	fork();
	fork();
	printf("Hello World!\n");
	return 0;
}
```

Listing 4: `fork()` Usage

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

void fork0() {
	int pid = fork();
	if (pid == 0) {
		printf("Hello from child\n");
	} else {
		printf("Hello from parent\n");
	}

	return;
}

int main(void) {
	fork0();
	return 0;
}
```

Listing 5: Using `fork()`, Performing Separate Actions

- Child prints first, seen in Listing 6.

```
$ ./a.out
Hello from child
Hello from parent
```

Listing 6: Post- `fork` , Child Finishes First

- Parent prints first, seen in Listing 7.

```
$ ./a.out
Hello from parent
Hello from child
```

Listing 7: Post- `fork` , Parent Finishes First

- Child and parent print at the same time. But, there is a lock for the screen, blocking multiple Processes from printing out at the same time. This lock is what makes the output text appear in order.

3.1.2 `fork` Fails

`fork` , like most other Syscalls will return `-1` on a failure. The global variable `errno` is populated with the cause of the failure To access `errno`, refer to Listing 8.

```c
#include <errno.h>

extern int errno;
```

Listing 8: Using `errno` to get Error Return Codes

3.1.3 `fork` Bomb

A fork bomb just generates new Processes as fast as possible, overloading the system. A simplistic one is shown in Listing 9.

```c
#include <unistd.h>

int main() {
	while(1)
		fork();
	return 0;
}
```

Listing 9: `fork()` Bomb

3.2 Terminating Processes, `exit`

There are several possible ways for a Process to terminate.

- The simplest way to terminate a Process is for the main process to `return`. If we are being pedantic, the compiler actually implicitly inserts an `exit` in this case, making all possible exits from a process use `exit`.
- The `exit` Syscall.
  - This exits immediately
  - This may prevent a normal `return`

The standard UNIX **convention** is that exit status `0` is success, and any other value is some error code.

3.2.1 `atexit`

`int atexit (void (*fn)())` is a unique function. It registers a function that will be called after a Program has had `exit` called on it, but before it fully exits. This registration is achieved by passing a pointer to the function that shoudl be run. The registartion must happen some time before the `exit`. There is no particular place this registration **MUST** happen though.

In addition, these handlers are inherited by child Processes.

3.2.2 Zombie Processes

All processes become Zombie Process eventually, awaiting to be Reaped.

Defn 15 (Zombie Process)**.** A *zombie process* is one that is "dead", because it finished its execution, but is still tracked by the OS, because the parent has not used/Reaped/`wait`-ed them yet. This means:

- The `PID` remains in-use.
- The child Process's `exit` status can be queried.

ALL terminating/terminated processes turn into zombies.

Remark 15.1 (Processes Responsible for Reaping). All processes are responsible for Reaping their own (immediate) children. If a program has 2 forks, the child of the child is **not** reaped by the original parent.

Remark 15.2 (Orphaned Process). If a Process is completely *orphaned*, it transfers ownership to `PID = 1`, which will then Reap it.

If the parent Process did **NOT** Reap its children, the only way to remove these Zombie Processes is by **killing** the parent processes. Note that this is **not** the same as terminating the process.

Defn 16 (Reap)**.** To *reap* a Process is to clean up after the process. This means closing any files, freeing any resources used, then reading the `exit` code from the child (the one being reaped), and freeing the process itself.

Typically, this is done with the `wait` Syscall.

3.3 Getting Values from Processes, `wait`

The `wait` Syscall is the one that allows parent Processes to receive values back from their children. It is only called by a process with >= 1 children.

The `wait` Syscall:

1. Waits (if needed) for a child to terminate, and returns the `exit` status of the child. This informs the parent:
   - Termination cause
   - Normal/abnormal termination
   - Some macros are defined to find out the exit status of a process. There are **MANY** more than the ones below, I just listed a couple.

     `WIFEXITED(status)`: Did the process `exit` normally?

     `WEXITSTATUS(status)`: What was the `exit` status of the child?
2. Reaps the zombified child. If the number of Zombie Processes is ≥ 1, and no specific one was given to `wait`, then `wait` picks a child.
3. Returns the reaped child's `PID` and exit status via pointer (if non-`NULL`)

If `wait` is called by a Process with no children, `wait` returns `-1` and populates `errno` with an appropriate error code.

How to use `wait` is shown in Listing 10.

3.3.1 Synchronization Mechanism

`wait` also functions as a synchronization mechanism. If a parent Process `wait`-s for the child to finish, this synchronizes things between the parent and the child. An example, in code, is shown in Listing 11.

3.4 Changing the Running Program, `exec`

`exec` is almost never used directly. Instead, its family of syscalls is used, which all provide some amount of abstraction from the base `exec` call.

All of these are front-ends to `exec`.

1. `execl`

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
        pid_t cpid;
        if (fork() == 0) {
                exit(0); /* Child -> Zombie */
        }
        else {
                cpid = wait(NULL); /* Reaping Parent */
        }

        printf("Parent pid = %d\n", getpid());
        printf("Child pid  = %d\n", cpid);
        while(1);
}
```

Listing 10: `wait()` Usage

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

void fork9() {
        if (fork() == 0) {
                printf("HC: hello from child\n");
        } else {
                printf("HP: hello form parent\n");
                wait(NULL);
                printf("CT: child has terminated\n");
        }
        printf("Child is dying. Bye\n");
}

int main() {
        fork9();
        return 0;
}
```

Listing 11: Using `wait()` as a Synchronization Tool

2. `execlp`
3. `execv`
4. `execvp`
5. `execve`

The variations in the families are denoted by the last letters in the function.

l: Arguments passed as list of strings to `main()`.

v: Arguments passed as array of strings to `main()`.

p: Path(s) to search for running program.

e: Environment (Environment variables and other state) specified by the caller.

Each of these can be mixed to some extent. The only constant between all of these is that the first argument, the name of the file to execute.

All of these execute a **new Program** within the **current Process context**, meaning **NO** new `PID` is given. When called, **`exec` never returns**, because it immediately starts the execution of the new program. This is because the binary image is replaced in-place.

How to use `exec` is shown in Listing 12.

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

int main() {
        execl("/bin/echo", "/bin/echo",
              "hello", "world", (void *)0);
        /* Everything below the execl becomes unreachable code, as the new
         * program REPLACES the original binary. */
        printf("Done exec-ing...\n");
        return 0;
}
```

Results:

```
$ ./a.out
hello world
```

Listing 12: `exec()` Usage

`exec` is a strong complement to `fork`, because we can make a new Process with `fork`, then change the new child to a new program with `exec`. An example of this, in code, is shown in Listing 13.

Results:

```
$ ./a.out
-rwxr-xr-x 1 ... a.out
-rwxr-xr-x 1 ... demo.c
Command completed
```

3.5 Signals

One way to pass information between Processes that are not constantly, directly communicating is through the use of Signals.

Defn 17 (Signal)**.** *Signal*s are messages delivered by the kernel to the user Processes. These can occur in the cases of:

- In response to OS events (segfault)
- The request of another process

Signals are delivered by a Signal Handler function in the **receiving process**.

```c
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
        if (fork() == 0) {
                execl("/bin/ls", "/bin/ls", "-l", (void *)0);
                exit(0);
        }
        wait(NULL);
        printf("Command Completed\n");
        return 0;
}
```

Listing 13: Using `fork()` and `exec()`

```c
#include <unistd.h>
#include <sys/wait.h>

int main() {
        int stat; /* stat is for the status of the child. */
        pid_t pid;
        if ((pid = fork()) == 0)
                while(1); /* Child goes into infinite while-loop. */
        else {
                kill(pid, SIGINT); /* Parent INTERRUPTs child with SIGINT*/
                wait(&stat);
                /* Signal handler in the parent to handle how a child handled a signal. */
                if (WIFSIGNALED(stat))
                        psignal(WTERMSIG(stat), "Child term due to:");
        }

        return 0;
}
```

Listing 14: Using Signals

An example of how Signals can be used is shown in Listing 14.
It can be useful to send a signal to multiple processes at once, in which case, the signal can be sent to a Process Group.

1. Signals can be delivered at *any time*
   - Interrupt **anything NONATOMIC**
   - Problematic if using global variables
   - Thus, minimize the use of global variables and their use in Signal Handlers
     - If global variables are needed, use data that can be read/written atomically.
2. A Signal Handler may execute in overlapping fashion (even with itself).
   - It does this when there are multiple signals to handle
   - Try to create separate Signal Handlers for different Signals.
     - Otherwise, signal handlers MUST be *Reentrant*
3. Execution of Signal Handlers for *separate* signals may overlap.
   - Any functions these Signal Handlers call may overlap as well
   - Thus, keep signal handlers simple
   - Minimize calls to other functions
4. Race conditions can be caused by this concurrency because we cannot predict when:
   - A child terminates
   - A Signal will arrive
   - We need to ensure that certain sequences *cannot be interrupted*

Defn 18 (Reentrant)**.** A *reentrant* program or function is one that is able to be called, repeatedly, while already executing.

3.5.1 Signal Lifecycle

1. Sending a signal to a process or a Process Group.
   - The `void kill(pid_t pid, int sig)` function is an example of a function that sends a Signal.
   - Give the process with `pid` the Signal `sig`.
   - There is a list of signals with names and values. The actual list is **MUCH** longer, but a few are shown below.

```
SIGHUP: Terminate process, Terminal line hangup
SIGINT: Terminate process, Interrupt program.
SIGQUIT: Create core image/dump, quits program.
```

2. Registering a Signal Handler for a given Signal.
   - Some Signals cannot be caught by the Process.
   - The function `sig_t signal(int sig, sig_t func)` registers a function (`func`) to a particular signal (`sig`).
   - Children inherit their parent's signal handlers after a `fork`.
   - Children lose their parent's signal handlers when they `exec` to another Program.
3. Delivering a Signal to a Process (done by kernel).
   - 2 Bitmaps per Process
     (a) Pending
     (b) Blocked
   - There is no queue or counter for signals, as this functionality is not supported by a Bitmap
     - However, they are dealt with in a particular order
     - The order is from higher number to lower number, lower to higher priority (31 -¿ 0)
   - Some Signals cannot be delivered/blocked (`SIGKILL`, and others)
   - Newly `fork`-ed child inherits the parent's blocked bitmap, but pending vector is empty.
4. Designing a Signal Handler.
   - If the same signal is received while that signal handler is running
     - Nothing special happens.
     - The handler is already running, and when it finishes, the value in the Bitmap will be zeroed out, indicating completion.
   - If we receive a higher priority (lower-number) signal while handling a lower-priority (higher-number) one:
     - Preempt the lower priority handler with the higher one.
     - The higher priority will interrupt the higher priority and be run.

– If it is possible, the lower-priority signal will be handled after the higher priority one is complete.

Defn 19 (Bitmap)**.** A *bitmap* of *bitmask* is a data structure of finite size where each component element can either be a `0` or a `1`. These are used for keeping track of information about existence. They are done this way, because we can very easily check for the existence of something by performing a bitwise operation.

The kernel uses these Signal Bitmaps before a Process starts/resumes. Before resuming a process, the kernel computes `~pending & blocked`

- Remember that this is bitwise NOT (`~`) and bitwise AND (`&`).
- The result of this operation determines which signals get delivered to the process before the process begins regular execution.

The Signal Bitmaps are held in **kernel memory**.

3.5.2 Process Groups

Defn 20 (Process Group)**.** A *process group* is a way to logically group several Processes together. Child processes `fork`-ed inherit their process group ID `pgid` from their parent. This leads to the following properties:

(i) The founder of the group becomes the group leader.
(ii) The group leader is the Process where `pid == pgid` .
(iii) A Process can become a group leader by `setpgrp` .
(iv) The whole Process Group interacts with the Signals.
(v) If `kill` is given a negative value, it will kill the corresponding process group.

Remark. For most Processes started from a shell, they inherit the shell's Process Group ID. This can be changed through the `setpgrp` function. If this is done, any subsequent child processes started by this one will inherit that new `pgid`.

3.5.3 Registering Signal Handlers

Before we run our program, many Signal Handlers are registered to handle certain Signals in certain ways. However, we can choose to override them, or define new ones, if we so choose.

Defn 21 (Signal Handler)**.** A *signal handler* is a function that has been registered to handle one particular signal. These can be used to override some default handlers, but there are some handler functions, like the one for `SIGTERM` that are not allowed to be changed.

An example of the registration of a Signal Handler is shown in Listing 15.

3.5.4 Adjusting Signal Masks

If we want to modify the Signal state Bitmaps, we use the `sigprocmask` function.

`sigprocmask` has the following prototype: `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)` . The returned `int` is either `0`, for success, or `-1` for failure. The three parameters are:

`how` How to deal with the signals. There are three options here, defined as preprocessor constants in `signal.h`.

- `SIG_BLOCK` The set of blocked Signals is the union of the current set and the `set` argument.
- `SIG_UNBLOCK` The Signals in `set` are removed from the current set of blocked signals.
- `SIG_SETMASK` The set of blocked signals is set to the argument `set`.

`*set` The set of Signals to block that should be applied to the current set.

`*oldset` The previous set of blocked Signals. You do not **need** to pass something in here, but if you are interested in getting the signal mask at some point, you will want to have a variable to assign here.

By using `sigprocmask`, you are explicitly telling the system you want to block handling a particular set of Signals, until you unblock that particular set of signals.

4 Input/Output (I/O)

In UNIX, I/O devices include:

- Disk
- Terminal

```c
#include <unistd.h>
#include <sys/wait.h>

int main() {
	int stat; /* stat is for the status of the child. */
	pid_t pid;
	if ((pid = fork()) == 0)
		while(1); /* Child goes into infinite while-loop. */
	else {
		kill(pid, SIGINT); /* Parent INTERRUPTs child with SIGINT*/
		wait(&stat);
		/* Signal handler in the parent to handle how a child handled a signal. */
		if (WIFSIGNALED(stat))
			psignal(WTERMSIG(stat), "Child term due to:");
	}

	return 0;
}
```

Results:

```
$ ./a.out
^CRelaying SIGINT to child
Child Dying
```

Listing 15: Registering Signal Handlers

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

int main(void) {
	sigset_t mask;
	sigemptyset(&mask);
	sigaddset(&mask, SIGINT);
	sigaddset(&mask, SIGALRM);

/* Block the set of signals specified in `mask'
 * Do not collect the previous set of blocked signal by passing `NULL'*/
	sigprocmask(SIG_BLOCK, &mask, NULL);

	return 0;
}
```

Listing 16: Using `sigprocmask`

- Shared Memory
- Printer
- Network

This is mostly because UNIX made the design decision to try to view every component of a system as a File.

Due to the variety of I/O devices that need to be supported, there are a vast number of different mechanisms for using these devices. But, there are a few common mechanisms, requirements, and activities:

- Read/Write Ops
- Metadata:
  - Name
  - Position
  - Directory Name
  - Creation Date
  - Last Access Date
  - IP Address
  - MAC Address
  - TCP Packet Sequence Number
- Robustness
- Thread-safety

There are few general concerns that we need to have about the idea of viewing everything as a File.

- How are I/O endpoints represented?
  - File Descriptor
- How do we perform I/O?
  - Byte at a time
  - Give a chunk of memory and later check that the requested I/O completed?
- How do we perform I/O *efficiently*?
  - Efficiency depends on what we define efficient to be. Essentially, what are we optimizing for?

4.1 I/O Devices

There are 2 major types of I/O devices on UNIX systems:

1. Block Devices
2. Character Devices

Defn 22 (Block Device)**.** A *block device* is an I/O device that accesses and stores data in fixed-sized blocks. Typically, this also means they have fixed total size as well. This means they support seeking through their contents and random access for parts of their contents.

Some typical devices classified as this are:

- Disk
- Memory

Defn 23 (Character Device)**.** A *character device* is an I/O device that access and receives data as a stream. This means it receives "characters" as a stream, one-by-one. There is no support for seeking or random access of the stream, because we are getting the data as soon as it is being given.

Some typical devices in this category are:

- Network
- Mouse
- Keyboard

4.2 Filesystem

The filesystem acts as a namespace for devices, and allows for the efficient storage of data on Block Devices. A typical file system consists of two types of files:

1. *Regular files* consist of ASCII or binary data
   - Directories

2. *Special Files* may represent:
    - In-memory structures
    - Sockets
    - Raw Devices

4.3 Files

Defn 24 (File)**.** A *file* is an operating system abstraction over some other data. It allows us to interact with many different file systems and devices over a variety of protocols in a abstract and concise way. A file can be accessed by using a *fully qualified path.*

The only thing each file **MUST** have is a unique `inode`.

Defn 25 (`inode`)**.** The *inode* is a filesystem-unique number (Typically, there is one filesystem per device, so this is typically a per-device-unique number). The inode tracks:

- Ownership
- Permissions
- Size
- Type
- Location
- Number of Hard Links

Defn 26 (Hard Link)**.** A *hard link* is a link between `inode`s. Thus, they each point to the same data, and must have the same name. When one of the links is deleted, the total count for that inode decreases. Once the inode reaches `0` hard links, it is removed (deleted) from the system.

Defn 27 (Symlink)**.** A *symlink* (*symbolic link*, *soft link*) is a link between Files. Thus, the link points to the file, which then points to the data. The link is not required to have the same name as the original file. However, if the file that the symlink is pointing to is deleted, then we are left with a dangling pointer.

However, the items discussed above are held strictly in storage, hard drive, meaning they aren't good for regular use, as they are too slow.

Remark. Barring you doing something like setting up a portion of your RAM as a "hard drive", allowing for ridiculous performance.

Thus, we load and open an `inode` into Memory, and call this copy a `vnode`.

Defn 28 (`vnode`)**.** A *vnode* is a copy of the `inode` of the current file. Every currently open file has a **single** `vnode`.

Now that we have a way to efficiently access a file in-Memory, using the `vnode` structure, how can we "open" the same File multiple times? This is handled by the File Description structure.

Defn 29 (File Description)**.** The *file description* allows the kernel to track which Process has opened which File and trace them back to their `vnode`. For **each process**, the kernel maintains a table of pointers to its open file structures. This table points to each file descriptor, which then point to the backing `vnode`s.

The file description tracks:

- Position
- Access Mode
- Pointer to the backing vnode

There is an open file description for **each occurrence** of a File's opening.

However, all of the structures discussed above are in **kernel memory**, meaning user Processes cannot interact with them. This is where the File Descriptor comes in.

Defn 30 (File Descriptor)**.** The *file descriptor* is the index of this file in the File Description table. There are always three file descriptors defined at the beginning of a Program's execution.

1. FD `0` is STanDard INput (`STDIN`)
2. FD `1` is STanDard OUTput (`STDOUT`)
3. FD `2` is StanDard ERRor (`STDERR`)

After opening a file, **all** file operations are performed using File Descriptors. This obscures kernel I/O and filesystem implementation details from the user, allowing for an elegant and abstract I/O API.

4.4 System-Level I/O API

Input and output is one of the basic operations that a program of any use will have to do. Below, there is a list of common Syscalls for performing these operations. These Syscalls are the lowest-level I/O calls we can make for files

`int open(const char *path, int oflag, ...)` Opens a File.

- Loads `vnode` for File at `path`
- `oflag` is a bitwise OR of `O_RDONLY` , `O_WONLY` , ...
- Creates and initializes a new File Description in the table
- Returns the first unused File Descriptor available
- If you open the same file twice, then you get a new file descriptor (a new file description is made), but it points to the same `vnode`
- Process inherits parent's open files across a `fork`
- Process retains them after an `exec`
- Parent and child share:
  - File position
  - File Access mode
- Sharing this file description allows for coordinating between separate process
- You can mirror this inside of a single process by using the `dup` Syscall.

`int fstat(int fd, struct stat *buf)` Query for file metadata

- `struct stat fstat.st_ino` Get inode number
- `struct stat fstat.st_size` Get file size
- `struct stat fstat.st_nlink` Get number of hard links

`int dup(int fd)` Duplicate the given File Descriptor and return a new file descriptor that points to the same File.

`int dup2(int fd1, int fd2)` Duplicate `fd1` such that `fd2` writes to the same `vnode`.

- We can use this to change the File Descriptor in use, allowing us to write to 2 locations at once.

`int close(int fd)` Close this File Descriptor.

- Delete the File Descriptor and deallocate the File Description.
- Once all File Descriptions of that point to this `vnode` are closed, the vnode is freed and the File is really closed.

`off_t lseek(int fd, off_t offset, int whence)`

`ssize_t read(int fd, void *buf, size_t nbytes)` • Read up to `nbytes` from `fd` into the buffer `buf`.

- Blocks until at least one byte is available.
- Returns the number of bytes **actually** read.

`ssize_t write(int fd, void *buf, size_t nbytes)` • Write up to `nbytes` into the open file at `fd` from `buf`.

- Returns the number of bytes **actually** written.

4.4.1 Why up to `nbytes`?

The `nbytes` parameters in `read` and `write` are necessary because the kernel attempts to maximize performance and minimize throughput. By knowing these terms, it makes it easier to schedule disk reads and writes, and allows for more efficient kernel buffering.

Each of the items in the list below are each valid reasons to have to specify the number of bytes to manipulate.

- `read`
  - `EOF`
  - Unreadable `fd`, for example when the disk is failing.
  - If the file is slow, this allows reading, a quick return to the Process ASAP and allow it to decide to read again or do something else.
  - Interrupt, for example from a Context Switch.
- `write`
  - Out of space, for example when the file is full, or when there is no more storage space on the backing media.
  - Unwritable `fd`
  - Slow File
  - Interrupt

4.4.2 Non-Blocking Reads

Non-blocking reads are just like the other `read` operation, but it returns to the calling process immediately.

4.5 Buffering

Defn 31 (Buffering). *Buffering* means the system will read more bytes than we (the programmer) actually requested, and will place all of the gathered data into a separate *backing buffer*.

How it works:

1. The user requests an amount of data
2. The kernel fetches it, plus some extra
3. The extra is not presented to the user, but is stored in the kernel
4. If the user asks for the next part, which is in the extra part, the read returns immediately.

4.5.1 Benefits

Your computer has many levels of buffers.

- User-level buffers
- Kernel-level buffers
- There are also lower-level buffers/caches. They are present in:
  - The CPU
  - The storage controller
  - The disk itself

By having many levels of buffers, we can create the illusion of great speeds, depending on the workload. This way, we can operate on a buffer in-memory, rather than having to go to disk constantly. That also has the added benefit of avoiding extra Syscalls.

4.6 File I/O API

`FILE* fopen(const char *filename, const char *mode)` Open the file specified by `filename` in the mode specified by `mode`.

`FILE* fdopen(int fd, const char *mode)` Open the file described by the File Descriptor in the mode specified by `mode`.

`int fclose(FILE *stream)` Close the file (object) stream that `stream` points to.

`int fseek(FILE *stream, long offset, int whence)` Seek within the file stream specified by `stream`, according to the `offset`, from `whence`.

`size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream)` Read `nitems`, each of size `size`, from `stream`, and store the contents where `ptr` points to.

`size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *stream)` Write `nitems`, each of size `size`, from `ptr` to `stream`.

`int fprintf(FILE *stream, const char *format, ...)` Write a character stream specified by `format`, with the dots being additional parameters as specified by `format`, to `stream`.

`int fscanf(FILE *stream, const char *format, ...)` Read a character stream specified by `format`, with the dots being additional parameters as specified by `format`, to `stream`.

`char* fgets(char *str, int size, FILE *stream)` Get `size` characters from `stream` and store it where `str` points to.

4.7 Generalizing

An example of a higher-level I/O function that performs many of the tasks we have discussed here is `printf`. It handles the:

1. File Descriptor, `STDOUT`
   - Way to access the file in the user-level code.
2. The Backing Buffer
   - Significantly reduces the cost of accessing stored information, because using memory access
3. The number of unused bytes
4. Pointer to the next byte

4.7.1 Easing Your Life

While `open` and `read` are the lowest level of I/O Syscall, and are available to you, they are also somewhat hard to work with. So, there are a variety of syscalls that wrap `open` for you, performing many of the actions we discussed earlier. Some examples are:

- `fclose`
- `fflush`
- `fgets`

All of these examples are from the system-library (which is really from `glibc`) `stdio.h`.

4.7.1.1 Stream Objects The functions from `stdio.h` operate on *stream* objects, which are wrappers around a file descriptor and the associated buffer. This means stream buffers can absorb multiple writes before flushing to the underlying file. However, a flush would occur when:

- The buffer is full
- Normal Process Termination
- A newline is encountered. In a line-buffered stream, for example, printing to the screen.
- Explicitly, with the `fflush` function.

However, this means that if the data is **not** flushed from the buffer, then the data is lost. This could be a dangerous operation because the programmer might expect the data to be flushed/written immediately.

4.8 Performing Both Input and Output

Performing both input and output at the same time to the same backing buffer is undefined int he C compiler, however, the C standard **does** specify this.

- ISO C99 standard 7.19.5.3 Paragraph 6
- Output shall not be followed by input without an intervening call to the `fflush` function.
- Input shall not be directly followed by output without an intervening call to a file positioning function.

4.9 Summary

In summary,

- Buffered `<stdio.h>` functions help minimize system overhead and simplify I/O
- Use `<stdio.h>` whenever possible.
- Beware of glitches when it comes to saving buffered data for later use.
- Don't mix buffered and unbuffered I/O, as it is undefined.
- `stdio.h` type of I/O is not appropriate for some devices (Network)
  - Use a low-level, robust I/O for network communications instead.

5 Inter-Process Communication

The OS kernel is great at *isolating* Processes from each other, but allowing processes to communicate with each other makes them more useful. Allowing them to communicate enables the processes to exchange data and interact dynamically.

However, this separation is done to make programming easier. If the OS were to **not** isolate each process

- Any and every Process could read and/or write to any other process's memory space.
- Thus, any process's memory integrity would not be guaranteed
- In effect, this would make any process's control flow unpredictable.

Because the kernel enforces isolation, we need the assistance of the kernel to complete any Inter-Process Communication. Two processes must explicitly request the kernel to allow them to communicate.

Defn 32 (Inter-Process Communication)**.** *Inter-Process Communication* (*IPC*) is the act of two or more Processes communicating with one another. There are variety of mechanisms for allowing this, explored in Section 5.1.

5.1 Mechanisms

There are a variety of mechanisms for Processes to interact and communicate with each other. Predictably, each one of these has an intended use, has certain benefits, and has certain drawbacks.

5.1.1 Signals

Signals were discussed in more depth in Section 3.5. But, in the case of Inter-Process Communication, signals are a very limited form of communication, as the signal sends a very well-predefined message.

5.1.2 (Regular) Files

It is always possible to save the information to a regular file on the system and then have other Processes read from and write to this file to communicate. This does have its place, but for many small reads and writes, the overhead of writing to the storage medium and using the file system will cause greater slow-downs. The slow-down incurred will typically be 1–2 orders of magnitude slower than memory.

This is very good for *static* Inter-Process Communication, such as configuration files. But files are not typically considered as a regular mechanism for dynamic Inter-Process Communication.

5.1.3 Shared Memory

Processes can share memory regions between each other. This allows for very fast access, with no direct limitation on the way the information is written and read form the shared area. However, this lack of uniformity (and potential problems with atomicity) can lead to major headaches.

5.1.3.1 API In the API below, **YOU**, as the programmer, must explicitly remove this shared memory. Memory allocated through the functions below are **NOT** deallocated upon process termination, which can lead to a permanent memory leak. This is intended because the operating system doesn't necessarily know when a set of Processes is done with a chunk of shared memory.

`int shm_open(const char *name, int oflag, mode_t mode)` Open a named region of memory to share between Processes.

`int shm_unlink(const char *name)` Unlink a chunk of shared memory from the current Process's memory map.

5.1.3.2 Synchronizing Shared Memory Shared memory is dangerous, because we have no protection from another Process writing over what we had just written. Thus, we need to synchronize access to this memory.

This can be done with Signals, in the following steps:

1. Writer sends signal to reader
2. Reader reads from memory
3. Reader signals back to writer that it is done
4. Writer removes the shared memory

But, the operating system cannot queue signals, so certain operations cannot be queued. In addition, signals do have some overhead, so passing many signals around as a synchronization mechanism can be costly.

5.1.4 Pipes

These are similar to the shell pipe `|`.

Defn 33 (Pipe)**.** *Pipe*s are a data structure and idea that allow us to implement Inter-Process Communication. A pipe behaves like a queue data structure, with one Process writing to the pipe, and another (not necessarily the same one) reading from it. A pipe uses explicit `send`, `receive`, `read`, and `write` functions to utilize the pipe, making it easier to figure out what process is doing what at what time. However, only 2 processes can use a pipe at a time, making it difficult to go from one process to many different processes.

There are 2 kinds of pipes:

1. Named Pipe
2. Unnamed Pipe

Using Pipes allows us to more easily implement correct Inter-Process Communication functionality. In addition, there is no need to go to the file system, so there are no file system performance implications. Technically Pipes are implemented using Shared Memory, so you get memory-access speeds.

5.1.4.1 Named Pipes

Defn 34 (Named Pipe)**.** A *named pipe* has some key differences compared to a Unnamed Pipe. Both of these types exist in the UNIX world. These differences are:

- It has a specific name which can be given to it by the programmer, which corresponds to where it is located on the filesystem. Named pipe is referred to through this name only by the reader and writer. All instances of a named pipe share the same pipe name.
- A named pipe can be used for communication between two unnamed process as well. Processes of different ancestry can share data through a named pipe.
- A named pipe exists in the file system. After input/output has been performed by the sharing Processes, the pipe still exists in the file system independently of the process, and can be used for communcation between some other processes.
- Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in case of a distributed system.
- A named pipe can have multiple process communicating through it, like multiple clients connected to one server.

In addition to these, one Process will block a `read` operation until another process connects to the `write` end of the pipe.

Named Pipe API

`int mkfifo(const char *path, mode_t perms)` Create a named pipe at the location specified by `path`, with permissions `perms`, and returns that pipe's File Descriptor.

`close(int fd)` Close the pipe.

5.1.4.2 Unnamed Pipes

Defn 35 (Unnamed Pipe)**.** An *unnamed pipe*, sometimes called an *anonymous pipe* has some key differences compared to a Named Pipe. Both of these types exist in the UNIX world. The differences are:

- Unnamed pipes are not given a name. It is accessible through two file descriptors that are created through the function `pipe(fd[2])` , where `fd[1]` signifies the **write** File Descriptor, and `fd[0]` describes the **read** File Descriptor.
- An unnamed pipe is only used for communication between a child and it's parent process.
- An unnamed pipe vanishes as soon as it is closed, or one of the Process (parent or child) completes execution.
- Unnamed pipes are always local; they cannot be used for communication over a network.
- An unnamed pipe is a one-way Pipe that typically transfers data between a parent process and a child process.

Unnamed Pipe API

`int pipe(int fds[2])` Create an unnamed pipe, which can be referenced by the File Descriptors `fds`.

`fds[0]` The read end of the unnamed pipe.

`fds[1]` The write end of the unnamed pipe.

`close(int fd)` Close the given file descriptor. All instances of this pipe will need to be closed on all ends to have the pipe deallocated.

5.1.5 File Locks and Semaphores

These two synchronization mechanisms are used to make concurrent systems predictable.

Some mechanisms that are similar to this that we have already used are:

`wait` But, this has a limited ability to do *things*.

`kill` **and** `signal` A poor way to communicate, because we cannot queue and cannot handle multiple Signals simultaneously.

Pipes The synchronization here is actually implicit, and allows for blocking calls to get information on both ends of the pipe. From the end-user perspective, this simplifies things, because the kernel is in charge of keeping the pipe synchronized. However, we are limited to the byte stream interface of the pipe, meaning we cannot read arbitrarily within the pipe.

5.1.5.1 File Locks File Locks control concurrent access/modification of shared files.

Defn 36 (Lock)**.** A *lock* allows **only one** Process to enter the portion of code that is locked. While a thread holds this lock no other Process can execute on this code portion.

Remark 36.1 (Binary Semaphore)*.* Locks can be represented as *binary Semaphores*.

These are one of the most common synchronization mechanisms, but they are not the best from a performance perspective.

- Concurrently reading a file from multiple processes is allowed.
- Concurrently modifying a file is not allowed. This can have ugly consequences.
- A file lock prevents other Processes from using a file.
- Locks are **NOT** preserved across a `fork`.
- There is also **NO** assurance that the filesystem supports file locking, as Linux makes it advisory.
- Mandatory locking is possible, but the filesystem must have been designed with that in mind.
- In general, these are not designed for general-purpose synchronization.

Defn 37 (Mutex)**.** A *mutex* (short for ***mut**ual **ex**clusion*) is the same as a Lock, **but it can be system wide (shared by multiple processes)**. A mutex lock protects critical regions and prevents race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

The `acquire()` function acquires the lock, preventing any other Process from using the thing the lock protects. Likewise, the `release()` function releases the lock, allowing another Process take acquire the lock and use the resource it protects. To perform its function correctly, the lock's `acquire()` and `release()` functions must be atomic.

When a Process and/or Process attempts `acquire()` the lock, while it is already owned by someone else, it is put in a `WAITING` state.

File Lock API

`int fcntl(int fd, int cmd, struct flock)` Create a file lock for the given File Descriptor.

- `cmd` is an enumeration of `F_GETLK`, `F_SETLK`, `F_SETLKW`.
  - `F_GETLK` Test acquisition of the lock.
  - `F_SETLK` Acquire the lock.
  - `F_SETLKW` Release the lock.
- `struct flock ...` is the type of lock and how to get the lock.

5.1.5.2 Semaphores

Semaphores control shared memory's access and modification. Essentially, they allow m of n Processes to acquire a resource. If this is supposed to be a mutually exclusive resource, then $m = 1$, and this is called a Mutex.

Semaphores control the order in which Processes run. They are typically associated with a counter. They also require that the wait and release process be atomic, especially when working with the counter.

Defn 38 (Semaphore)**.** A *semaphore* regulates the number of things (Processes or Processs) performing operations on something (Shared Resource). Functionally, this is the same as a Mutex but allows x Processes/Processs to enter or use the resource at a time. This allows for limits on the number of CPU, I/O or RAM intensive tasks running at the same time.

A semaphore can only be interacted with through 2 operations `wait()` and `signal()`. `wait()` is similar to a Mutex's `acquire()` function and the `signal()` function is similar to the Mutex's `release()` function. Here, if a Process or Process `wait`s, if there is more of the resource, then the requester gets the resource, and the internal count of the semaphore is decremented. If a Process `signal`s, then it is done with the resource, and the requester loses access to the resources, and the internal count of the semaphore is incremented. Again, these manipulations **MUST** be atomic.

Remark 38.1 (Confusion with Mutexes)*.* Technically, you can create a Semaphore that acts like a Mutex by giving it a binary value. However, this is typically in poor programming taste, because while both function similarly, the Semaphore is for signaling the amount of a resource available and the Mutex is for signaling if code is capable of execution.

Remark 38.2 (Correct Use of Semaphores)*.* The correct use of a Semaphore is for signaling from one task to another. A Mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects. By contrast, tasks that use Semaphores either signal or wait, not both.

Semaphore API

`sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)` Create a semaphore with `name`, the flags `oflag`, with the mode `mode`, and an initial value of `value`. The semaphore object is returned from the call.

`int sem_wait(sem_t *sem)` Decrements `sem`'s backing counter. If the counter is already `0`, then this call blocks the calling Process.

`int sem_post(sem_t *sem)` Icrements `sem`'s backing counter. If the counter was `0` and becomes greater than `0`, and another Process is currently waiting on `sem`, then it one will be woken up and allowed to lock the semaphore.

5.1.5.3 Spinlocks

A spinlock is the process of busy-polling a resource until its available. This is a highly-responsive way to allocate, but wastes a lot of CPU time.

5.1.6 Sockets

Sockets are mainly used for network communication. However, they can be used on the local computer too.

Defn 39 (Socket)**.** A `socket` is a way of connecting two nodes on a network or Processes to communicate with each other. One socket listens, while another socket reaches out to the other to form a connection.

Network sockets have a high overhead due to the software-defined network stacks.

Almost all modern computers use this today, except for High Performance Computing, which uses their own hardware solutions to reduce latency.

5.2 Challenges

5.2.1 Link/Endpoint Creation

Some common issues with creating a link and/or endpoint are:

- Naming the endpoint
- Looking up the endpoint
- Need a registry to keep track of this information

5.2.2 Data Transmission

Data transmission has many questions to answer for it to be effective. These include:

- Unidirectional or bidirectional?
- Single-sender or multi-sender and/or single-receiver or multi-receiver?
- Speed of the transmission medium/link?
- Capacity of the transmission medium?
- Message packetizing? How does the message stream get converted to packets?
- How is the transmission routed?

5.2.3 Data Synchronization

For the data to be of any use to anyone/anything, must it arrive in a certain order? These kinds of questions define the data synchronization problem of Inter-Process Communication, and some more are included below:

- What is the behavior when there are multiple senders and/or receivers?
- What is the control required to synchronize?
  - Is it done implicitly?
  - Does it need to be done explicitly?
  - Is there **ANY** synchronization?

6 Sockets

6.1 Communication Protocols

- Protocols are agreement and rules on communication
- These can be connection-oriented or connectionless
- These protocols are typically built with a layered architecture-
  1. Physical
  2. Data Link
  3. Network
  4. Transport
  5. Session
  6. Presentation
  7. Application
- These messages build off each other by wrapping the higher-level protocol in a lower one.

6.1.1 Physical

- How to encode 0s and 1s
- What voltages are used
- How long does a bit need to be signaled
- What does the cable, plug, antenna look like?

6.1.2 Data Link

- How big is a frame
- Can I detect an error
- What marks an end of a frame
- How do I control access to a shared channel (Flow Control)

6.1.3 Network

- How to route packets
- Congestion control algorithm
  - Traffic Shaping
  - Flow Specifications
  - Bandwidth reservation
- Accounting
- Fragment or combine packets

6.1.4 Transport Layer

- How to order messages and detect duplicates
- Error detection
- Retransmission
- Connection-oriented vs. Connectionless

6.1.5 Session and Presentation

-

6.1.6 Application

- What marks the subject field
- How to represent cursor movements
- Services
  - SMTP
  - FTP
  - HTTP
  - SNMP
  - NFS
  - NTP
  - NNTP

6.2 Middleware Protocols

6.3 Sockets

In general, networking is done with message-oriented communication. This is typically done with packets, which is a byte oriented **stream** of data. Packets are used because of:

- Persistent
- Synchronicity
- These packets are routed through potentially multiple computers before reaching a destination
- You buffer the packets you receive to handle the data as it comes.

Sockets are **not** limited to just inter-computer network communication. They can also be used within the same computer. For example:

- Emacs (Emacs server + `emacsclient`) is written to be usable over a socket on the local computer
- Remote communication happens for email, web browsing, etc.

Server doesn't need to `fork` unless they want to communicate with multiple clients. Threads can also be used. The client doesn't need to `fork` unless they want to communicate with multiple servers simultaneously. Just like the server, threads can be used for this instead of Processes.

6.3.1 Message-Oriented *Transient* Communication

6.3.2 Berkeley Socket

The typically definition of a socket follows the definition of a *Berkeley socket*.

Primitive	Meaning
Sockets	Communication End point. When programming, it looks like File Descriptor.
Bind	Attach a local address to a socket. A single computer might have multiple IP addresses/host adapters. This tells the Process which network interface to listen on.
Listen	Announce willingness to accept connections. Essentially a blocking call until a connection comes into the system.
Accept	Block caller until a connection request arrives. Have received notification you have received a connection and you allow it (Server-side).
Connection	Actively attempt to establish a connection. (Client-end).
Send	Send some data over the connection.
Receive	Receive some data over the connection.
Close	Release this connection.

Table 6.1: Berkeley Socket Primitives

6.3.2.1 Socket Usage To start using sockets, you must distinguish and denote one computer a client and another a server. Servers behave similarly to the way we are used to running programs, but these Processes are typically very long-running. The usual lifecycle for a socket consists of:

1. Socket
2. Bind
3. Listen on a specific port. Now blocks.
   - If the parent forks, then the parent can stay here and accept new connections.
4. Accept
5. Send/Receive
6. Close (Usually client is the one that closes the connection)
7. Wait until a new connection. (Sometimes if the connection is closed, the process ends)

On the other hand, clients behave as shorter-running processes, compared to their server brethren. They go through these steps:

1. Connect
2. Send/Receive
3. Close

6.3.3 Message-Oriented *Persistent* Communication

6.4 MPI (Message Passing Interface)

- Berkeley sockets are designed for general-purpose network communication
  - Simple send/receive primitives
  - General-purpose protocol stacks such as TCP/IP
  - These primitives are very heavy, because of the all the work that goes into a network stack
- Abstraction of these kinds of sockets are not suitable for other protocols in clusters of workstations or massively parallel systems.
  - New, more advanced, primitives are required to make these communications as fast as possible.

```c
/* Server side C/C++ program to demonstrate socket programming. */
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080
int main(int argc, char const *argv[]) {
        int server_fd, new_socket, valread;
        struct sockaddr_in address;
        int opt = 1;
        int addrlen = sizeof(address);
        char buffer[1024] = {0};
        char *hello = "Hello from Server";

        /* Create socket file descriptor. */
        if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
                perror("socket failed");
                exit(EXIT_FAILURE);
        }

        /* Forcefully attach socket to the PORT port. */
        /* Use TCP/IP */
        if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT< &opt, sizeof(opt))) {
                perror("setsockopt");
                exit(EXIT_FAILURE);
        }
        address.sin_family = AF_INET;
        address.sin_addr.s_addr = INADDR_ANY; /* Allow this port to listen on ALL IP Addresses */
        address.sin_port = htons(PORT);

        /* Forcefully attach socket to the PORT port. */
        if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
                perror("bind failure");
                exit(EXIT_FAILURE);
        }
        /* fork here, child continues through this, and parent will stay on listen. */
        if (listen(server_fd, 3) < 0) {
                perror("listen");
                exit(EXIT_FAILURE);
        }
        /* Socket accepts from client. */
        if ((new_socket = accept(server_fd, (struct sockaddr*)&address, (socklen_t*)&addrlen)) < 0) {
                perror("accept");
                exit(EXIT_FAILURE);
        }
        /* Socket Receives data. */
        valread = recv(new_socket, buffer, 1024, 0);
        printf("%s\n", buffer);
        /* Socket sends some data back. */
        /* Have to be careful of architectural differences between 2 communicating machines. */
        send(new_socket, hello, strlen(hello), 0);
        printf("Hello message sent\n");
        return 0;
}
```

Listing 17: `server.c` as an Example for Server-side Socket Programming

```c
/* Client side C/C++ program to demonstrate socket programming. */
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[]) {
        int sock = 0, valread;
        struct sockaddr_in serv_addr;
        char *hello = "Hello from client";
        char buffer[1024] = {0};
        if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        }
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(PORT);

        // Convert IPv4 and IPv6 from text to binary.
        if(inet_pton(AF_NET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
                perror("\nInvalid address/Address not supported.\n");
                return -1;
        }

        /* connect does eventually timeout. */
        if(connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
                perror("\nConnection Failed \n");
                return -1;
        }

        send(sock, hello, strlen(hello), 0);
        printf("Hello message sent\n");
        valread = recv(sock, buffer, 1024);
        printf("%s\n", buffer);
        return 0;
}
```

Listing 18: `client.c` as an Example for Client-side Socket Programming

- However, over time, there have become a large number of incompatible proprietary libraries and protocols.
  - This demands a standard interface to be used, called Message Passing Interface (MPI)
- The MPI Reference is available online at `https://www.mcs.anl.gov/mpi/`
- MPI is:
  - Hardware independent
  - Primarily for highly parallel applications
  - Transient communication
  - Applications that require **MUCH** lower latency in comparison to sockets.
- Key idea here is that communication is done between groups of processes
  - Each endpoint is a `(groupID, processID)` pair
- Supports most forms of communication

Primitive	Meaning
`MPI_bsend`	Append outgoing message
`MPI_...`	

6.5 Remote Procedure Calls

The question here is: How do we make "distributed computing look like traditional computing"?

- The standard Client-Server protocol provides usable mechanisms for services in distributed systems
  - But, these require explicit communication, and require an explicit Send-Receive Paradigm.
- Can we use procedure calls to do this?
  - The goal here is to make a remote procedure call look like a local procedure call.
- In distributed system, the callee may be a completely different system from the one executing.

6.5.1 Design Issues

- Parameter passing
  - Local:
    1. Parameters passed on the stack before jumping elsewhere
    2. The stack holds the parameters and the possible local variables until you finish the call.
    3. Parameters can be call-by-value or call-by-reference
  - Remote:
    1. Simulate parameter passing with *Stubs and Marshaling
       * Client makes procedure call to client stub
       * Server written as a standard procedure
       * Stubs take care of packaging arguments and sending messages
       * The packaging is called marshaling
       * Stub compiler generates stubs automatically from specifications in an *Interface Definition Language* (IDL)
- Binding
- Reliability
  - How to handle failures
  - Message loss
  - Client crash
  - Server crash
- Performance and implementation issues
- Exception handling
- Interface definition

6.5.2 Steps

1. Client procedure calls client stub in normal way
2. Client stub build message and calls local OS (Marshaling)
3. Client's OS sends message to remote OS (Actual socket send happens here)
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server (Can also call multiple functions, need to determine the parameters passed).
6. Server does work and returns results to the server-side stub
7. Server stub marshals the message and calls the local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks the result and return to the client

6.5.3 Marshaling

A Computer Components

A.1 Central Processing Unit

Defn A.1.1 (Central Processing Unit)**.** The *Central Processing Unit*, *CPU*, is a chip that performs all actions in the computer. It calculates mathematical and logical values and acts based on them. It has several components built onto it, and can be thought of as the "brain" of the computer.

The design of a CPU determines some of the functionality it has. Therefore, more specialized processors can be made for special tasks, and more general processors can be built to handle a wide variety of calculations.

A.1.1 Registers

Defn A.1.2 (Register)**.** A *register* is a data storage mechanism built directly onto the Central Processing Unit. It is several hundred times faster than the system Memory. Registers are generally used when the currently running program is performing calculations. Since they are so fast, they are used as both source and destination operands in instructions.

Remark A.1.2.1. Depending on the Central Processing Unit architecture, there may be cases when Registers behave slightly differently between processors. This is something that can only be found by checking the Central Processing Unit manufacturer's documentation.

A.1.2 Program Counter

Defn A.1.3 (Program Counter)**.** The *program counter* is a Register that contains the value for the memory address of the next executing instruction. It does **NOT** hold the currently executing instruction's address in memory because that instruction is already in the Central Processing Unit. This keeps track of where the program is in execution and which instruction comes next.

A.1.3 Arithmetic Logic Unit

A.1.4 Cache

A.2 Memory

Defn A.2.1 (Memory)**.** *Memory*, or *RAM* (*Random Access Memory*), is a Volatile data storage mechanism. It is directly connected to the Central Processing Unit. This is the location that the Central Processing Unit writes to when it cannot or should not store something in the Central Processing Unit's Registers.

Remark A.2.1.1 (Volatility). Memory is volatile because each of the cells is a small capacitor. In between the clock cycles on the Central Processing Unit and Memory, the capacitors discharge. On the clock cycle, the capacitors are refreshed with electrical power, which does one of 2 things:

1. Keep the data bits the same, 1 to 1.
2. Update the data bits from 0 to 1.

Defn A.2.2 (Volatile)**.** If a data storage mechanism is called *volatile*, it means that once the storage mechanism loses power, the data is lost. This is in contrast to Non-Volatile data storage mechanisms.

A.2.1 Stack

Defn A.2.3 (Call Stack)**.** The *call stack* is an imaginary construct that resembles the traditional stack data structure. It is a way to visualize and organize the way memory is used during the execution of a program and its function/methods. It is filled with Stack Frames. This **does not** hold the code that is used in the function, rather it is everything that is needed for the function to be able to run.

Defn A.2.4 (Stack Frame)**.** A *stack frame*, *activation frame*, or *activation record* are objects that represent necessary portions of a function. These include:

- A Dynamic Link
- Local Variables
- Temporary Variables
- Static Links
- Function Arguments
- Return Address

Additionally, there are 2 registers used as pointers to move around and interact with the stack frame.

1. `FP` is in register `%rbp`. It is the Frame Pointer.
2. `SP` is in register `%rsp`. It is the Stack Pointer.

Defn A.2.5 (Dynamic Link)**.** The *dynamic link* or *dynlink* is a memory address pointer and sits at the bottom of a Stack Frame. It is a pointer back to the previous function's dynamic link. This ensures that any function can find its parent/calling function.

The dynamic link also serves as a means to access any variable that might be needed by this function. To access any variable in *this* function, you can subtract a byte multiple that you need to access the proper value. To access any variable in the calling function, you can add a byte multiple that would correspond to the proper variable.

In the `x86_ 64` architecture, instruction set, and convention that we used, the register `%rbp` was the dynamic link.

Remark A.2.5.1. Note: Due to the conventions we used, when accessing arguments passed to the function, we treated them as local variables, just further down in the call stack. This also means that we need to skip over the Return Address block in memory.

Remark A.2.5.2. Because we used the `%rbp` register to store our current Dynamic Link's address, the dynamic link might also be called the *base pointer*.

Defn A.2.6 (Local Variable)**.** *Local variables* are handled very simply. They get an appropriate amount of memory allocated to them on the stack, and that is it.

There is no way to give a variable a name in assembly. (Usually. Depends on the architecture and instruction set). However, there is no way to name something in memory. But, because the size of all the objects is known at compile-time, allocating the proper amount of memory required by each variable is possible.

Remark A.2.6.1. This holds true for strongly-typed, static, compiled languages, like Java, C, C++, etc. However, Python is slightly different in this regard, and handles it differently.

Defn A.2.7 (Temporary Variable)**.** A *temporary variable* is one that is allocated on this function's Stack Frame while calculating values. Once the calculations are completed, these values are deallocated. These temporary variables can also point to objects on the Heap. When the function has finished running, then these values are deallocated, along with all other Local Variables in use.

For example, since assembly-level addition only allows for 2 operands, but in general, addition can have more than 2 operands in use, there needs to be a way to store the value used in the addition. While we can accumulate and use that value in the addition, the values being added together are *not* modified.

Defn A.2.8 (Static Link)**.** The *static link* is an implicit argument, meaning it ***ALWAYS*** gets pushed onto the stack as a Function Argument, when appropriate.

Appropriate in this context could mean several things:

- When an object is alive, and when it is being acted on by a function.
  - In this case, the static link points to an instance of a class on the Heap.
- When a language allows for nested function declarations.
  - Then the static link points to the Dynamic Link of the outer function
  - This allows access to the outer function's Local Variables like normal, and allows us to go back later.

Defn A.2.9 (Function Argument)**.** *Function arguments* are handled very simply. If a function call takes an argument, then the argument is calculated, and then that argument is pushed onto the stack in *this* Stack Frame.

Remark A.2.9.1. If more than one argument is passed to a function, there are 2 ways to push values onto the Stack Frame:

1. In the order they are passed to the function
   - Say a function with 3 arguments is called, then the stack would have arguments in this order
     (a) argument0 (Lowest memory address)
     (b) argument1
     (c) argument2 (Highest memory address)
2. In reverse order
   - Say the same function is called with the same 3 arguments, then the stack would have arguments in this order
     (a) argument2 (Lowest memory address)
     (b) argument1
     (c) argument0 (Highest memory address)

When the values that were passed need to be accessed, and if the memory sizes of things are known at compile time, then we can calculate how far down we need to go in the stack to find the value. This is done by adding a positive value to the `%rbp`

Defn A.2.10 (Return Address)**.** The *return address* is used by the `%rip` register. It is calculated and pushed onto the Stack Frame stack when the `CALL` macro is used. It is the thing that allows us to jump around in the code from the `text` area of our program.

Remark A.2.10.1. The `%rip` register is the *register instruction pointer*. It holds the value of the *next* instruction to execute. Technically, it is the Program Counter's value.

Defn A.2.11 (Garbage Collection)**.** *Garbage collection* is the act of deallocating objects that may still be on the heap and organizing the heap. Since the heap allocates continuous "blocks" of memory required by an object, the heap may have the necessary memory to allocate an object, but in discontinuous locations.

Defn A.2.12 (Frame Pointer)**.** The *frame pointer* is a pointer that ***ALWAYS*** points to the current function's Dynamic Link. The frame pointer is commonly abbreviated as *FP*. This is the thing that allows us to access Local Variables, Function Arguments, and everything else inside of the Stack Frame. The value is held in the `%rbp` register.

Defn A.2.13 (Stack Pointer)**.** The *stack pointer* is a pointer that ***ALWAYS*** points to the top of the current Stack Frame. The stack pointer is commonly abbreviated as *SP*. This value is held in the `%rsp` register. This is the pointer that allows us to push and pop onto this function's Stack Frame.

Defn A.2.14 (Class Descriptor)**.** The *class descriptor* is a portion of memory set aside for the methods that are in an object.

A.2.2 Heap

Defn A.2.15 (Heap)**.** The *heap* is a memory organization construct that helps visualize the way memory is used during the execution of a program. It is ***SIGNIFICANTLY*** larger than the Call Stack.

In an object-oriented language, this is where instances of classes exist. These objects are allocated on a first-come first-serve basis. When the object is allocated, the amount of memory that the object requires is allocated ***in a continuous block***. The problem is that if the heap has been extensively used, then there might be enough total memory required to allocate an object, but because it is not in a continuous block, the object cannot be allocated.

This makes deallocation more complex, because when they are deallocated, the program might still view the memory as in-use. Additionally, because of the continuous block allocation nature of the heap, it must be organized every once in a while. This is called Garbage Collection.

Remark A.2.15.1. In an object-oriented language, like Java, when objects are stored on the heap, **ONLY** their class fields are stored there. The class methods are stored elsewhere, in the Class Descriptor.

A.3 Disk

Defn A.3.1 (Non-Volatile)**.** If a data storage mechanism is called *non-volatile*, it means that once the storage device loses power, the data is still safely stored. This is in contrast to Volatile data storage mechanisms.

A.4 Fetch-Execute Cycle