The Go Teaching Operating System

Mechanical Sympathy for Software Developers

mumblingdrunkard

22. April 2022

Contents

List of Listings

Chapter 1

Introduction

Welcome to *The Go Teaching Operating System* (gotos). While this book concerns operating system (OS) concepts, it is not a book to teach you everything about operating systems. It should be supplemented with other texts. This book is more about *a practical approach* to learning OS concepts through exercises and lab assignments.

Knowledge of computer architecture should not be a requirement for Gotos , but it will help you understand the low-level details quicker. The necessary computer architecture will be covered as you progress through the book and should line up nicely with the concepts as they are introduced.

Some important words:

- *I*, the author
- *You*, the reader
- *We*, both of the above

What this book is:

A comprehensive guide to get you from A to Z in gotos, building a working operating system with a POSIX environment for user-mode applications. Introduces important concepts of os-development

What should you know?:

Required knowledge: Go, git, reading.

Recommended knowledge: C, computer architecture, systems programming, basic algorithms and data structures.

Chapter 2

The processor

Before we even get into the nitty-gritty details of operating system development, we have to cover some basics of computer architecture. The goal is to teach you just enough about the processor to understand how it helps the operating system accomplish its goals.

I will frequently switch between explaining parts of the processor and the operating system, hoping you'll have a good understanding of the "what, why, and how" by the end. Sometimes, *what* we do in the operating system won't make much sense without knowing *how* they happen on the processor. Similarly, parts of the processor won't make sense without knowing *why* they exist for the benefit of the operating system.

To access the base code for this chapter: `git checkout 02-the-processor`.

2.1 What a processor is

Before we delve into the details of operating systems, we should cover what a processor is. A processor is, as its name implies, something that processes. This definition is somewhat unhelpful, however, as it doesn't say anything about how it does the processing.

Most, if not all, processors use *registers* and *instructions.* A register is a small space where the processor can keep certain information. This information is represented as a series of 1s and 0s; it could be a number, a letter, or anything else.

The processor uses instructions to do things with the data stored in registers. An instruction could be to add the contents of two registers and store it in a third register. A different instruction might be to place (load) a value into a register. By performing instructions in a specified order, the processor can do anything from playing Pong to browsing the internet. We call this sequence of instructions a *program*. In listing 2.1, you can see a very simple program that loads values 17 and 25 into register `t0` and `t1`, adds them together, and stores the value in register `a0`. The register names are actually just numbers. E.g. registers `t0`, `t1`, and `a0`, are registers number 5, 6, and 10 respectively.

```
li   t0, 17       # load 17 into t0
li   t1, 25       # load 25 into t1
add  a0, t0, t1   # add t0 and t1, store result in a0
```

Listing 2.1: A simple program that computes the answer to life, the universe, and everything.

The language used here is called assembly, a human-readable form of machine code. This assembly code is specifically for a RISC-V processor and assembles to 3 RISC-V instructions. This program may look a little confusing so in listing 2.2 you can see a rough equivalent in Go.

```
var register  [32]uint32
register[5]  = 17                             // li   t0, 17
register[6]  = 25                             // li   t1, 2
register[10] = register[5] + register[6] // add  a0, t0, t1
```

Listing 2.2: Go ”pseudocode” illustrating instructions acting on registers.

2.2 Memory and addressing

A RISC-V processor has 31 registers, a very large amount compared to other processor architectures which often have just 16 or fewer. In the grand scheme of things, however, it is very little storage. A 32-bit RISC-V processor has 31 32-bit registers which is only 992 bits or 96 bytes of storage.[1] For this reason, processors also use *random access memory* (RAM) to store data that may not be immediately included in any calculation, but can be loaded in and out of registers on demand. The “random access” means that different parts of memory need not be accessed in any specific order.

Memory is structured as an *addressable, ordered sequence of bytes*. If you’ve previously done some programming, you have likely used arrays before. Arrays are addressable collections of data, usually accessed with square brackets like so: `array[12]`, which would give you the element with index 12. Thus, memory can be seen as an *array of bytes*. The `uint8` type in Go can be used to store a byte.

You can get individual bytes from this array by using different addresses. For our array example, the address is simply the index, 12 in our case. Bytes can be interpreted alone or several bytes together may form larger pieces of data. You

[1]There are usually 8 bits in a byte, but some very old systems used different lengths of a byte. Virtually all modern systems use the 8-bit byte however.

can combine four bytes into a 32-bit unsigned integer for example. In Go we can do it as illustrated in listing 2.3. When printing, we can represent the number as a series of hexadecimal digits.

Hexadecimal is specially suited for displaying binary numbers, as clusters of 4 bits can be represented by a single hexadecimal digit.

```
bytes [4]uint8{33, 67, 101, 135}
number := binary.LittleEndian.Uint32(bytes[:])
fmt.Printf("%08X\n", number)
```

Listing 2.3: Four bytes can be interpreted as a uint32 in Go.

Memory ordering

In the example code, you can see that we have used little endian encoding. Little endian means that when combining the bytes, we take the last byte as the *most significant byte*.
The bytes in the example can be written as hexadecimal values `0x21`, `0x43`, `0x65`, and `0x87`. When combined using little endian ordering, the final value becomes `0x87654321`. Had we used big endian, the result would be `0x21436587`.
In this book and throughout this project, we will almost exclusively be using little endian as this is the default for RISC-V processors.

How about when we are not just reading bytes and want larger types? Well, just read the next $X - 1$ bytes where X is the number of bytes in the type you want to read. In 2.4 you can see how this would look, converting the 4 bytes from address 8 to 11 (inclusive) into an unsigned integer, still in little endian encoding. Can you find the value of the variables?

```
bytes := [12]uint8{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
n0 := binary.LittleEndian.Uint32(bytes[0:0+4])
n1 := binary.BigEndian.Uint16(bytes[4:4+2])
n2 := binary.LittleEndian.Uint16(bytes[6:6+2])
n3 := binary.LittleEndian.Uint32(bytes[8:8+4])
```

Listing 2.4: We can address several bytes at a time and interpret/combine them into larger types.

2.3 Stored programs

When assembly language is assembled into machine code, the resulting machine code is a binary file. The assembly code from the previous section compiles into the binary data that you can see in listing 2.5.

```
93 02 10 01     # li   t0, 17
13 03 90 01     # li   t1, 25
33 85 62 00     # add  a0, t0, t1
```

Listing 2.5: Simple program assembled to a binary, represented as hexadecimal digits. Two digits is one byte.

Since we can represent instructions as a sequence of bytes, we can also store them in memory. In listing 2.6 we have done precisely this. First, we create a system (a virtual computer essentially) with a single core, then we write our program to its memory, starting from address 0. Each instruction in RISC-V is 32 bits, so in total we store 96 bits – or 12 bytes – to memory.[2]

We have just loaded three instructions, so it makes sense that our for-loop will run three times and then stop. Inside the loop, we *step* and *dump*. A step means reading an instruction, executing it, then stopping, leaving the processor ready to fetch and execute the next instruction in the next cycle. Dumping will print the data contained in the processor's registers. Note that dumping registers in Gotos will only print the registers that contain non-zero values.

```go
func main() {
    sys := system.NewSystem(1)

    sys.Memory().WriteRaw(0, []uint8{
        0x93, 0x02, 0x10, 0x01,
        0x13, 0x03, 0x90, 0x01,
        0x33, 0x85, 0x62, 0x00,
    })

    for i := 0; i < 3; i++ {
        sys.StepAndDump()
    }
}
```

Listing 2.6: Storing a program to system memory, then running it.

You should try this code for yourself. To do this, you can navigate to the git repository you cloned and checkout `02-the-processor`. Execute the code using `go run main.go`. Observe how the registers change after each instruction. Example output after the first instruction is shown in listing 2.7.

After all the instructions have finished, you should end up with values 17, 25, and 42 in registers 5, 6, and 10 respectively.

```
> === Register dump for core 0 ===
> pc: 4
> Integer registers
```

[2]Some RISC-V processors will also support 16-bit *compressed* instructions.

```
> [05]: 00000011 = 17
```

Listing 2.7: Output after the first instruction is executed. Observe that register 5 contains value 17.

2.4 The program counter

The processor has what is called a fetch-execute cycle where it reads the next instruction and executes it. This happens billions of times a second on modern processors.

How does the processor know which instruction it should fetch and execute next? It uses the *program counter*! I actually think that this is a confusing term as it doesn't really count anything. We will therefore use another term: the *instruction pointer*. These terms will be used to mean the same thing.

The instruction pointer is a special register that holds the address of the next instruction to be executed. It points to the next instruction, hence the name. In the example we used in the previous section, the pointer started at 0 and incremented by 4 in each run of the for-loop.

Let's try to start the program at a different location, say address 12. In listing 2.8, notice that we now have to manually set the instruction pointer using `SetCSR(cpu.Csr_MEPC, 12)` before telling the system to perform a boot routine. These lines ensure that the instruction pointer is set to 12 before execution starts.[3]

```
sys.Memory().WriteRaw(12, []uint8{
    0x93, 0x02, 0x10, 0x01,
    0x13, 0x03, 0x90, 0x01,
    0x33, 0x85, 0x62, 0x00,
})
sys.Cores[0].SetCSR(cpu.Csr_MEPC, 12)
sys.Cores[0].UnsafeBoot()
```

Listing 2.8: Starting a program at a different location in memory.

2.4.1 Branches and jumps

Most instructions will increment the instruction pointer when they execute successfully. However, some special instructions will set the instruction pointer to an arbitrary value or increment/decrement it by an arbitrary amount. These instructions are called *branches* and *jumps* and are important for controlling the flow of your program.

[3]`UnsafeBoot` only enters and exits machine mode, but the side effect of exiting machine mode is that the instruction pointer is set to the value stored in the **mepc** register that we write to. This was not needed earlier when the program counter defaulted to 0.

Branches are conditional, meaning that they only change the instruction pointer if some condition is true. Otherwise, they don't do anything and the instruction pointer is incremented by 4 as normal.

Jumps, on the other hand, are unconditional; they always change the instruction pointer. Branches are used for things like `if`-statements, while jumps are most notably used for entering and returning from functions.

An example of a branch being used is shown in listing 2.9. Note that `_start:` and `loop:` are just *labels*. They don't do anything in the assembled program, but they act as "handles" for the rest of our assembly-code to refer to. In the `blt` instruction, we compare registers `t0` and `t1` and if some condition is satisfied we jump to `loop`. Jumping to `loop` is the same thing as jumping to the address of the `addi` instruction, which is the same thing as setting the instruction pointer to the address of `addi`.

```
_start:
  li   t0, 0        # set counter t0 to 0
  li   t1, 42       # set target t1 to 42
loop:
  addi t0, t0, 1    # increment counter by 1
  blt  t0, t1, loop # go back to loop if t0 < t1
```

Listing 2.9: A program that branches

Roughly equivalent code in Go is shown in listing 2.10. Note that this is a pseudo-code-like example intended to give familiar syntax to an unfamiliar language.

```
var register [32]uint32
register[5] := 0
register[6] := 42

for {
    register[5] += 1
    if register[5] < register[6] {
        continue // run the loop again
    } else {
        break // leave the loop
    }
}
```

Listing 2.10: How a branch works.

You can load the assembled program by using the code from listing 2.11. You should modify the code in `main.go`. Pay special attention to how the `pc` (program counter/instruction pointer) changes throughout the execution of the program. Instructions are noted with their address as they appear in memory. E.g. when `pc` is 8, the next step will fetch and execute the `addi` instruction. Try to predict the next register dump based on the value of `pc`.

```
sys.Memory().WriteRaw(0, []uint8{
    0x93, 0x02, 0x00, 0x00, // 0 li   t0, 0
    0x13, 0x03, 0xa0, 0x02, // 4 li   t1, 42
    0x93, 0x82, 0x12, 0x00, // 8 addi t0, t0, 1
    0xe3, 0xce, 0x62, 0xfe, // C blt  t0, t1, ?
})

for i := 0; i < 20; i++ {
    sys.StepAndDump()
}
```

Listing 2.11: Loading the branching program.

In the rest of the book, the term *program counter* will be used instead of *instruction pointer* simply because that is the term used in Gotos in general. Just know that both terms refer to the same thing.

2.5 Ending your program

So far, we've only run programs for a given number of cycles. This is pretty unintuitive as we have to know exactly how many cycles each program can possibly take. It would be much nicer if the programs could just tell us when they're done and we could move on to the next task automatically. We'll get to what the "next task" is in the next chapter.

As it turns out, ending your program is easier said than done and we will have to jump through a few hoops.

2.5.1 Limited direct execution

There is no special instruction to tell the processor that we would please like to stop executing now. There's a good reason for this as it would be a bad experience if your web-browser could shut down your computer. Stopping the processor is what we call a *privileged* operation.

Instead, the processor implements what is called *limited direct execution* and is the first piece of the processor that directly helps the operating system achieve one of its goals. We want programs to run as fast as possible, but we may not want to let all programs execute all possible instructions. Therefore, the processor has *privilege levels* such as *user-mode* and *kernel-mode*. Kernel-mode has access to all instructions, even those that would turn off the system. User-mode, on the other hand, only has access to a select few instructions. Our programs only run in user-mode.

Accessing I/O is usually a privileged operation and so is turning off or halting the processor. Unprivileged operations are simple things like: adding numbers, setting the program counter, or writing to select locations in memory.

If you're very observant, a thought may have occured to you: "Hello, World!"

requires access to I/O. How can our program print things to the screen if it can only run in the unprivileged user-mode?

2.5.2 The processor trap

Traps are events where a program is interrupted so the *kernel* – a fancy word for operating system – can perform some specified task. A trap happens by interrupting the execution of the current program and switching to a *trap handler*. Eventually, control is (usually) returned to the program. A trap could be performed for several reasons. One such reason is a program that attempts to execute an unsupported or non-existent instruction.

The current approach of gotos is to simply stop the processor when a trap happens and print the reason for the trap, so let's try causing a trap by executing an instruction that doesn't exist. The simplest such instruction is just `0x00000000` and is what we have used in listing 2.12. Notice that we've used `sys.Run()` now which will cause the processor to run until it *halts*.[4]

```
sys.Memory().WriteRaw(0, []uint8{
    ...
    0x00, 0x00, 0x00, 0x00,
})

sys.Run() // run until the processor stops
```

Listing 2.12: Stopping the processor by using an illegal instruction, causing a trap.

```
> [core 0]: Illegal Instruction. **mtval** : 00000000
```

Huzzah! The processor stopped!

2.6 A better way

A program that attempts to perform an illegal instruction would likely crash on a real system. Not exactly a graceful exit. If your program contains an illegal instruction, it is most certainly an accident. Unless we actually want our program to crash, we'll have to come up with a better solution.

A much better approach is to use the `ecall` instruction from RISC-V which very intentionally causes a trap. An `ecall` (*environment call*) is a *call* – or request – to the environment (the operating system in this case) to perform some action. Let's add an `ecall` instruction at the end of our program:

If we run the code now we get a much friendlier output:

```
> [core 0]: ECALL from User Mode
```

[4]**Halt**, *intransitive verb*, To stop; pause.

```
sys.Memory().WriteRaw(0, []uint8{
    ...
    0x73, 0x00, 0x00, 0x00, // ecall
})
```

Listing 2.13: Intentionally causing a trap by using the `ecall` instruction.

No message about illegal instructions accompanied by scary numbers. Just a message that we have performed an `ecall` intsruction. For now, the processor just stops when it encounters this instruction, but let's do something else.

2.6.1 Handling `ecall`

In `system/trap_handler.go`, you can see where we handle the `ecall` trap. The specific method is `handleEcallUMode(...)`. Try printing a different message.

Add a file `system/syscall.go` with the contents from listing 2.14 and modify method `handleEcallUMode()` to invoke the `syscall()` method instead of halting.

```
// new file `system/syscall.go`
package system

import (
    "fmt"
    "gotos/cpu"
)

func (s *System) syscall(c *cpu.Core) {
    number := c.GetIRegister(cpu.Reg_A0)
    switch number {
    case 1: // `exit` system call
        fmt.Println("Program exited")
        c.HaltIfRunning() // stop the processor
    }
}

// modify in `system/trap_handler.go`
func (s *System) handleEcallUMode(c *cpu.Core) {
    s.syscall(c)
}
```

Listing 2.14: Special code to handle the environment call instruction.

Congratulations! You have just implemented your first *system call*! A system call consists of a call number and often some arguments. To use a system call, place the call number in register `a0`, then perform an `ecall` instruction. Let's do that.

2.6.2 Using `exit` from the program

We're really just missing one line here; we should put the system call number in register `a0` before `ecall` as shown in listing 2.15. The system call number for `exit` is 1. We have successfully implemented and invoked our first system call! Running the program now should give you a very friendly output:

```
> Program exited
```

```go
sys.Memory().WriteRaw(0, []uint8{
    0x93, 0x02, 0x10, 0x01,
    0x13, 0x03, 0x90, 0x01,
    0x33, 0x85, 0x62, 0x00,
    0x13, 0x05, 0x10, 0x00, // li    a0, 1
    0x73, 0x00, 0x00, 0x00, // ecall
})
```

Listing 2.15: Using `exit` in our program.

2.7 Returning control

Just to get the hang of it, let's add another system call: `sayhello`. It should have the call number 513 and, unlike `exit`, it should not stop the processor. See listing 2.16 for the updated `syscall.go` contents.

```go
func (s *System) syscall(c *cpu.Core) {
    number := c.GetIRegister(cpu.Reg_A0)
    switch number {
    case 1:   // `exit` system call
        fmt.Println("Program exited")
        c.HaltIfRunning() // stop the processor
    case 513: // `sayhello` system call
        fmt.Println("Hello!")
    }
}
```

Listing 2.16: Adding another system call.

Let's invoke our new system call just before we invoke `exit`. We do this by loading 513 into register `a0`, then performing an `ecall` as shown in listing 2.17. If you run the program as is, you'll notice something strange:

```
> Hello!
> Hello!
> Hello!
> Hello!
...
```

```
sys.Memory().WriteRaw(0, []uint8{
    ...
    0x13, 0x05, 0x10, 0x20, // li    a0, 513
    0x73, 0x00, 0x00, 0x00, // ecall
    0x13, 0x05, 0x10, 0x00, // li    a0, 1
    0x73, 0x00, 0x00, 0x00, // ecall
})
```

Listing 2.17: Calling `sayhello` before `exit`.

The message `Hello!` keeps printing with no end in sight; it seems our program has gotten stuck on invoking `sayhello()`! (Use `Ctrl + C` to kill the program on Mac or Linux. It might work on Windows too, but if it doesn't, just close whatever you are running the program in.) What went wrong?

The `ecall` instruction, like branches and jumps, is special when it interacts with the instruction pointer. It is special in that it causes a trap and it is special in that it *does not update the instruction pointer* for the running program.

Exceptions are exceptional

When a trap happens, it is an exception and you can guarantee that the instruction pointer will not be automatically updated or incremented. It is assumed that the operating system handling the trap will either return control at the proper address or perhaps even start running a different program. In short: you need to set it yourself.

This was not a problem when we were using the `exit` syscall as we stopped the processor, but after the `sayhello()` call, the operating system returns control to the program at the same `ecall` instruction, causing a loop.

Instead, we should return control to the program at the instruction *after* `ecall`. To do this, we will have to manually update the instruction pointer before we leave the `sayhello()` function.

To update the instruction pointer, we first have to get the address of the `ecall` instruction that caused the trap. Luckily it is not too difficult: when a trap occurs, the processor will store the address of the instruction that caused the trap in a special *control status register* (CSR). You have already used one such register though you may not be aware. `Csr_MHARTID` is a register that stores the ID of the processor. The address in question will be stored in the CSR named **mepc** (*machine exception program counter*) and can be accessed using `c.GetCSR(cpu.Csr_MEPC)`.

Before returning from `sayhello`, we should set the instruction pointer to the address of the next instruction, which is 4 bytes after the trapped instruction. See listing 2.18 for the corrected version of `sayhello`. When control is returned to user-mode, the instruction pointer will be set to the value stored in **mepc**. By setting this value, we can control where the processor continues execution after our trap handler returns. We want to return control at the instruction after `ecall` so we set it to the address four bytes after the current one.

```
    case 513: // `sayhello` system call
        fmt.Println("Hello!")
        cameFrom := c.GetCSR(cpu.Csr_MEPC)
        c.SetCSR(cpu.Csr_MEPC, cameFrom + 4)
```

Listing 2.18: Properly returning from a system call.

Finally! The output after fixing `sayhello` looks correct when running.

```
> Hello!
> Program exited
```

Chapter summary

In this chapter we have covered basic processor architecture.

A *register* is a small storage space for data that the processor is currently using. *Instructions* make the processor do things to the data stored in registers. Instructions are represented as a series of bytes and are stored in *memory*. Memory is *addressable* storage for data that doesn't immediately fit in registers. Memory is addressed at the byte-level, though multiple bytes can be combined to form larger pieces of data.

The *fetch-execute cycle* is the procedure of reading an instruction from memory and executing it. The *instruction pointer* is a special register that keeps track of which instruction should be fetched in the next cycle. Some special instructions will change the instruction pointer to an arbitrary value; these are called jumps or branches.

Traps are special exceptions or events that need to be handled by the operating system. Traps can be caused on accident by including an illegal instruction in your program, on purpose through `ecall`. *System calls* can be implemented with the help of an intentional trap. System calls allow a program to interact with the operating system to perform actions that the program may not have permissions for. Accessing I/O is an example of such an operation.

In the next few chapters we will explore how the operating system uses these concepts and mechanisms to share the processor between several programs.

Exercises

Exercise 2.1 (*). Take the below code and figure out the final result placed in register `a0`.

```
li   t0, 17
li   t1, 25
mul  t0, t0, t1
li   t1, 5
sub  a0, t0, t1
```

Exercise 2.2 (*). Do some research about assembly language and figure out what the below code is doing. Do you recognise the algorithm?

```
start:
  li     t0, 0
  li     t1, 1
  li     t3, 200
loop:
  add    t2, t0, t1
  mv     t0, t1
  mv     t1, t2
  blt    t2, t3, loop
  mv     a0, t2
```

Exercise 2.3 (*). Take the code from listing 2.4 and figure out what the final value of the numbers will be.

Hint: Convert the numbers to two-digit hexadecimal numbers and reorder them afterwards. Two hexadecimal digits perfectly represent a single byte.

Exercise 2.4 (*). How many cycles does the program from 2.11 take until it's finished? (Finished meaning it would continue past the `blt` instruction.) 20 cycles was not enough.

Exercise 2.5. Modify the program from 2.11 to call `exit()` when it's done.

Exercise 2.6 (*). Modify the `exit()` system call to print a result value. The result value should be placed in register `a1` before the `ecall` instruction is run. `0x93, 0x05, 0xa0, 0x02,` will place 42 in register `a1`.

Exercise 2.7. Add a system call `add` with call number 514. `add` should take the values stored in registers `a1` and `a2` and place the sum in `a0`. `0x13, 0x05, 0x20, 0x20` will place 514 in register `a0`.

Part I

VIRTUALISATION

Chapter 3

Processes I

A central goal of the operating system is managing and sharing resources between multiple programs. The processor is one of these resources. Operating systems provide the illusion that each program has its own processor to run on. The program doesn't really care that it is sharing time on the processor with other programs, only that it eventually gets to do work.

Take the microwave in your flat for example. You're probably sharing it with multiple other students who are struggling to make ends meet. You likely only have the one microwave, but there are multiple hungry students and you all have frozen dishes that you would like to cook.[1] In this metaphor, the microwave is the processor, while you and your flatmates are programs. Everyone has things they would like to do with the microwave and some specific settings they need to set to properly accomplish the task.

As only one item fits in the microwave at a time, you all have to share it. If the microwave is busy cooking Ada's popcorn, Brian will have to wait to cook his hotpocket. While it is annoying to wait for a long time, you all mostly act like you have your own microwave. You have successfully virtualised the microwave.

In this chapter we will explore mechanisms for sharing the processor as well as simple queue management. We will also further abuse the microwave metaphor. To access the base code for this chapter, switch to branch `03-processes-i`.

3.1 The process

While programs are sequences of instructions, usually stored in memory, a *process* can be viewed as the combination of both a program and some *state*. The state is essentially all data that is **not** the program. The only state we worry about

[1]One important distinction between people and programs is that programs don't get particularly grumpy when they don't get to use the microwave to heat their dinner.

for now is the state of the registers and of the program counter introduced in the previous chapter.

Thus, we have our process. We represent a process using the *process control block* (PCB). This is a struct that contains important information about a process. In listing 3.1 you can see the process control block struct we will use in this chapter and it is defined in `system/pcb.go`. It contains an identifierss, the program counter, and the registers.

```
type PCB struct {
    ID        int
    PC        uint32
    Registers [32]uint32
}
```

Listing 3.1: The process control block in Gotos.

3.1.1 Creating a process

When creating a process, we need two things: the program and the process control block. Let's create a `Load(p PCB)` method that

3.1.2 Starting a process

3.2 The process queue

3.3 The context switch