### 1. Introduksjon

#### 1.1 Oppgavetekst

* Ledestjernen gjennom hele oppgaveløsningen: «Oppgaven går ut på å lage et system som ...»

#### 1. 2 Bakgrunn

* Sammenhengen utstyret/programmet/systemet skal benyttes – motivasjon
* Alt som studenten ikke har gjort selv
* Teori

### 2. Konstruksjon (*alt.*Materialer og metoder)

* Alt studenten(e) har gjort selv
* En redegjørelse for konstruksjon/programmering/løsningsmetode nøyaktig som den foreligger
* Skjema, kretskort
* Blokkdiagram
* Flytdiagram
* Forklaring av program på overordnet nivå
* Beskrivelse av eksperimentelle oppsett (*uten* resultater)

### 3. Resultat

* Objektive, målbare resultater
* Ytelse
* Presisjon
* Feilrater
* Kompatibilitet
* s-1 (alt som kan måles pr sekund)

### 4. Diskusjon

* Med utgangspunkt i oppgaveteksten (1.1), sammenlign med resultatene (3)
* Mulige alternative løsninger (som ikke ble valgt)
* Forslag til forbedringer
* Konklusjon (hvis den er mer enn et avsnitt, kan den bli kap 5)

### Vedlegg

* Referanseliste
* Økonomisk oversikt (prototyp/volumproduksjon)
* Datablad (hvis disse ikke finnes lett tilgjengelig på nett)
* Programlisting (hele prosjektstrukturen som .7z)
* Prosjektplan

#### 1.1 Oppgavetekst

This task is about creating a simulation of a rudimentary computer through software, for the purpose of teaching new students about some operating system aspects such as process scheduling and memory management. To further improve the ease of learning the program also needs a way to visualize these aspects in the form of a graphical user interface. The program simulates a CPU’s fetch-decode-execute cycle while integrating the said OS-level features process scheduling and memory management.

This will hopefully be used in the Operating Systems and System Programming course, to add to the learning experience. As currently in the course there is no way of visualizing the inner workings of how the system behaves. There will be some simplifications done, where deemed necessary. It would also be optimal if the students are able to code some modules to be used in the program, such as a scheduling policies, as part of some assignment. To fulfill this requirement the program needs to be somewhat modular and also be coded in GO.

The simulation is built around instructions, which are low-level operations a the system can perform. They are the simplest actions the system can perform, and things like memory and scheduling just enhances what type of instuctions the system can do. One such instruction can be to simply add a value to an address in memory. In a real system, everything, even the operating system itself is handles through these instructions. The operating system is just another program made up of instructions that the CPU runs, like any other program. But in this simulation, that part is not modeled in this way.

So things like the operating system, scheduling, and memory management do exist in the code, but they aren't simulated using instructions. They are implemented with regular functions written in higher-level logic to support the simulation.

#### 1.2 Technology stack

Written in Go.

Fyne ( Front-End)

Accumulator-based architecture

Modules:

- Cpu

- Memory

- OS

- Scheduler

- Dashboard (Fyne Part)

#### 1. 3 Bakgrunn

Now to explain how instructions work. All instructions involve what is called registers, there a multiple, and differ based on architecture. These registers are a small, super fast pieces of memory located directly in the CPU. They are used for holding various information, often for the immediate crucial data the CPU needs for it to run an instruction. One of the most common instruction architecture used in modern CPU’s are x86. This is a general-purpose register architecture, which uses multiple general-purpose registers for storing data, hence the name. An example of the add instruction on such a architecture is as follows:

“ADD EAX, EBX “

Where the first word “ADD” is for the type of instruction. This part is often called the opcode, and is usually represented in either HEX or Binary. The two other parts “EAX” and “EBX” are what is called operands, and is what is being worked on, depending on the type of instruction. In this case, these are the names of general-purpose registers. The instruction “ADD”, adds the value from the second register “EBX” to the register “EAX”. EAX + EBX -> EAX.

This is just one example of an instruction on the x86 architecture. To keep things even simpler, the simulation does not implement the x86 architecture, instead it uses an accumulator-based architecture.

For the add instruction, the premise is the same. But instead of having two operands, there is only one. Instead of explicitly mention which register to add to (“EAX”), it always use a special register called the Accumulator, so there is only one operand. It was mentioned “EBX” the second operand is a register, but it is possible for that operand to be a memory location or a immediate value instead. Where a memory location is often written like this [EBX]. The brackets indicate the value stored inside is a pointer to a memory location. And simply a number when using a immeditae value. However in the accumulator architecture method using a register does not exist as in x86, so the only options are an immediate number or memory address.

If using a immediate number the instruction would look like this:

“ADD 5“

Where the value 5 is added to the accumulator. This makes the instructions simpler, as it requires only writing one operand. Which hopefully makes it simpler to understand.

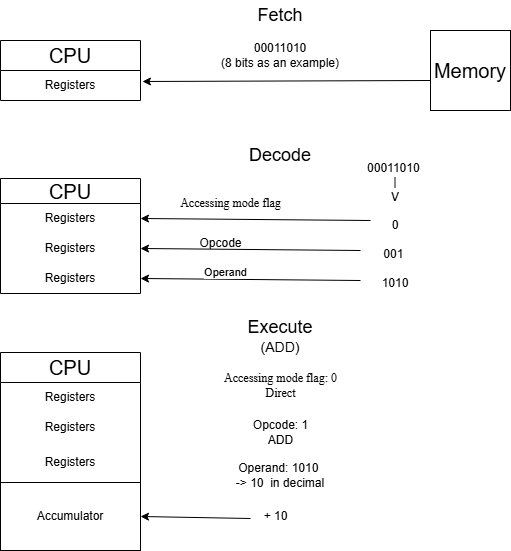
These instructions however is not written using the Latin alphabet inside the code. As all data is written in bits, the instructions are also written as such. That means each of the different parts, opcode, operands and also different flags that are not touched upon quite yet, are represented by a part of the full bit length each. This is also a place where this simulation differ. The x86 architecture uses different bit lengths for the instructions. This is makes the instructions more compact and faster to read, but makes it unnecessarily complex.

As such in this system, the instruction bit length is the same for all instructions. There is also only one flag, representing the previous example of accessing a memory location or using an immediate number. If the bit is 1, it accesses memory, and 0 represents an immediate number . The instructions are made of 64 bits, where the first bit is the accessing mode flag, the next 31 bits are for the opcode, and the last 32 bits are for the operand.

This implementation however, introduces an obvious flaw. There is a static max length of bits for the different parts. Now the accessing mode flag, is a non problem, however the operand is not. The bit length of the operand is always 32 bits. This can represent up to the number 4,294,967,295, which is quite large, but a limitation regardless. As for the opcode, this is also not a problem as even in modern system does not come close to the available bit length, and this simulation keep it even simpler, having only a few instructions. The limitation of the operand is however deemed necessary trade-off, to make it simple.

1. Konstruksjon

The Program is organized into two main parts, the operating system and the dashboard. The OS hosts the rest of the underlying logic, like the CPU, Memory, Scheduler, MMU, Process Controller, and the Process table. These are made as separate structs, which the OS holds a pointer for, inside the main OS struct. While the dashboard, is the logic required to make the GUI, which both show and allow some user inputs to operate on some data. Like creating processes and choosing which scheduler to use.



CPU Fetch-Decode-Execute cycle:

As previously stated is the basis of this simulation the CPU instructions, which are basic operations, which are executed by the CPU. This is done in what is called a Fetch-Decode-Execute cycle. First the CPU “fetches” the instruction from memory and stored in the appropriate registers. After this step, the instruction is decoded so it can be executed. As mentioned the instruction is composed of many bits, in this program 64 bits, but the different bits has different meanings. This means it has to do something “decode” it into separate parts. The last step in this cycle, is the execute step, which actually executes the instruction. In the example of the instruction “ADD”, this is where the CPU actually adds the value to the accumulator. This whole process is one cycle, and only one instruction is executed per cycle.

To understand how this was implemented, first we look at the registers:

- Program Counter, which points to the next instruction, in the code section in memory.

- Accumulator, which the logical operations work on.

- Instruction Register, holds the instruction in use.

- Memory Address Register, which holds the address for the memory.

- Memory Data Register, which hold either the instruction or data from memory.

- Stack Pointer, which points to the next memory location in the stack.

The CPU use these registers, to store temporary data used for handling the instructions. First the Program Counter (PC)(often called instruction counter), which is used to point towards the next instruction in the memory. The value stored is an address, and is incremented to the address for the next instruction in line. In a system like x86, this would mean it increments by various lengths, as mentioned it has a variable instruction length. In this simulation the length of each instruction is always the same, 64 bits. This makes this process simpler, as it only needs to increase the program counter to the next 64 bits each time.

There is a slight change in how this is implemented. Each memory address or location, can only hold so many bits, this is called the “Word size”. And this simulation has a word size of only 32 bits. The decision to use 32 bits instead of 64 bits, was the thought that it would be simpler to use. As 64 bit numbers, would be very large, and not as easy to look at. 32 bits were a middle ground where it would not be too large that might make it harder to use, and not too small that it should not be a hindrance.

Now there is actually a downside, which this simulation does account for, but in a real system would be an issue. Is that if you only use 1 bit of those 32 bits, to store lets say the number 1, now there would be a waste of 31 bits. This is bad in practice because it wastes a lot of space, however as this simulation is for student to learn more of how it is stored, and how to access such systems. And not what is stored inside the addresses, it was deemed as not a concern. It would still be more optimal, but the difficulty it would add, would be to far of a cost. There is also the aspect of showing such information, and this would make viewing such memory locations be harder to understand. In a real system, the word size would be 1 byte or 8 bits. If the information you want to store is larger than this, the data expand to multiple memory locations. It would make knowing which memory locations are for which data much more difficult, so instead with this simulation one memory location, is a complete data bit string.

Now back to the slight implementation difference talked about earlier, the instruction size is 64 bits. This is too large for storing in one memory location, which only hold 32 bits. This is where the technique for data to fill multiple memory locations comes in. Each instructions is stored in two of said memory locations, totalling 64 bits. Since the instruction is split into 3 parts - addressing mode flag (1 bit), opcode(31 bits) and operand (32 bits). And is neatly split where the first two parts is a 32 bit length size, and the operand is 32 bits. Viewing this information is not as tedious, as the first part is for the addressing mode flag and the opcode. The last is solely for the operand, which makes viewing the information user friendly.

This means at the fetching stage, the CPU is fetching from two memory location for each instruction.

What is store inside the program counter?

The value stored is the virtual address not the actual address for the memory location mentioned earlier. This is to keep the valuable information of how things are stored from each process, making in more secure, and harder for a process to access memory locations not allocated to it. For this to function something needs to translate the the virtual address to a physical one. This job is handled by the Memory Management System (MMU), and this method of securing memory, can be used by other memory strategies, not only paging which this simulation use.

So the CPU has to first ask the MMU to translate the virtual address stored inside the program counter, and the resulting translated physical address is then stored in what is called the Memory Address Register (MAR). This register holds the physical address, which the CPU is going to access next. The address stored could the translated address from the operand if the accessing mode is 1. It is not used solely for the program counter physical address. After this steps the CPU retrieves the instruction stored at the address inside the MAR. This retrieved instruction is temporary stored at the Memory Data Register (MDR). This register holds the retrieved data from memory, as such it doesn’t need to hold an instruction, but could also be an address, or value. Now as mentioned the CPU needs to fetch two memory locations to attain the 64 instruction bits. The first 32 bits was attained from the fetching the address stored in the program counter. So it also fetches from the next memory location holding the last 32 bits of the instruction.

The next step is to store this instruction to a register called Instruction register (IR), which holds the current running instruction, used for the other stages - decode and execute. The value stored in this implementation (in a real system everything is always stored in bits) are an instruction struct. Which has the three fields - addressing mode flag, opcode and operand. This ends the fetching stage, and the CPU sets the program counter to the next instruction to be run, and proceeds to the decoding stage.

At the decoding stage the CPU interprets the instruction. In this implementation the CPU first look at the addressing mode flag bit. This addressing mode flag indicates whether it needs to access the memory as mentioned. If the addressing mode flag is set to 0, it interprets the operand as an actual value to be used at the execute step. The value gets stored into the Memory Data Register (MDR), hold data that is being transferred to or from memory, and the step is done. If the addressing mode flag is 1, it skips this step, as this would mean the fetched data from memory is an address. The addressing mode flag being set to 1, indicates that the operand is a virtual address to be used to access memory, and the value stored at this memory location is what is stored inside the MDR to be used for the later operation, e.g ADD. However this memory accessing is also done in the execution stage, so the decoding stage is only to decipher the addressing mode flag. The reason why this is the only step is that in the fetching stage, the CPU already split the instruction, and stored these as a struct.

The next and last stage is the execution stage, where the instruction is executed. First it checks if it needs to access the memory, from the decoding stage. If it does, it does so first. Then the CPU proceeds to execute the opcode stored in the IR. These opcodes are mapped to operation functions stored inside the CPU. So it looks for the corresponding function for the opcode, and runs the function. This concludes the fetch-decode-execute cycle.

Operations:

ADD:

Adds the **specified value** to the accumulator.

OldAccumulator + Value = NewAccumulator

SUB:

Subtract the **specified value** from the accumulators.

OldAccumulator - Value = NewAccumulator

STORE :

Stores the current value in the accumulator, to the **specified value (Address)**.

LOAD

Retrieves the value from the **specified value (Address)**, and sets the accumulator to it.

JUMP:

Sets the Program Counter to the **specified value**.

PUSH:

Stores the current value in the accumulator, to the address of the Stack Pointer.

POP:

Removes the value at the address of the Stack Pointer, and sets the accumulator to it.

Note: The specified value can be directly from the operand, or from a stored memory address. Remember the Addressing mode flag.

Scheduling:

A combination of these instructions are combined into what is called a process, and these processes is what is being scheduled by a scheduler.

The scheduler is responsible for scheduling these processes to be run on the CPU. When there are more than one process are ready to run on the CPU, the scheduler has to decide which takes priority. The simplest method is what is called FIFO(First in, First out), which is a queue system, where the first in line will be run next. There is a host of scheduling policies with its own benefits, drawbacks and problems, which will be touched upon later. First comes an explanation over how the scheduler is made. Inside the OS struct there is a scheduler interface. This is how modularity of policies are done. Where you can input any scheduling policy inside the OS struct, as long as it fulfills the required interface requirements.

These are as follows:

AddProcess: This function adds a process to the ready queue(s) (Depends on policy).

GetNextProcess: This function returns the next process to be run based on the policy implemented.

GetRunningProcess: This function returns the current running process.

As long as the scheduling policy implements these, you can swap between any policy.

------------------------------

1. OS (Operating System)

- The entity of that govern the other entities, like cpu, memory, scheduler, process controller.

- Initialize the required components:

- CPU

- Memory

- ProcessTable

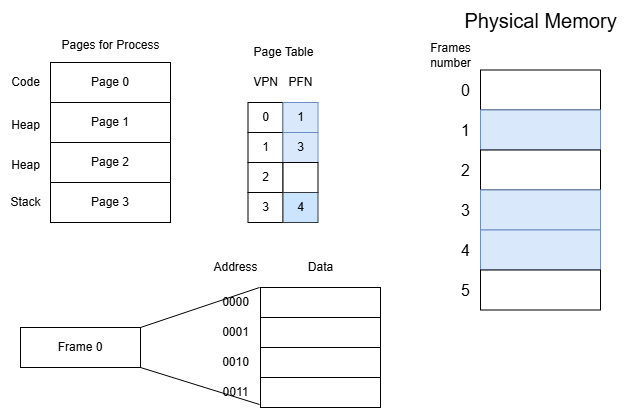
- CPU Controller

- Scheduler

- FreeList

- Start/Stop/Pause/Resumes the simulation.

- Schedules processes.

Figure 2.2 - Memory

1. Memory (RAM)

- Stores code, stack and heap, of various processes.

- Accessible by the CPU via the MMU.

Each process needs to have access to the system memory for storing data. How this is done, is determined by the memory management system. The system can give access to all of the available memory and call it a day, but that would cause various issues. If there are other processes that wants to also use the memory, they would write over the others data, if both have access to all of the memory. This is not ideal, so we need to give each process a “private” memory space, they are able to use.

There are many methods to achieve this, and this simulation uses paging, as mentioned earlier. Some other methods are: Segmentation, Contiguous Memory Allocation. They all have pros and cons, but paging is what is most commonly used in modern systems like x86, so that is what this simulation will use.

The paging system allocates what is called pages for each process. Where each page has a fixed size address space, that can be used for storing data. The actual physical memory is split into these fixed size address spaces, but when talking about the physical memory these are called frames, not pages. They are the same size, and is each page can be connected to one frame. The reason why this distinction is made, is to know where in the system we are talking about, as pages are the processes memory, and frames are the actual physical memory.

Now before we talked about using the MMU to translate virtual addresses to physical ones. This also requires to translate the page number to a frame number. We will talk later about the MMU process, but we will just touch upon the page table. Each process has a separate page table, which is a list of page to frames translations. See figure 2.2.

Each page, also called Virtual Page Number (VPN), has a corresponding Page Frame Number (PFN).

If you look closer at “Pages for process”, you can see a forth page “Page 3”, which is not in the page table. This is not an error. The processes is allocated a set of amount of pages when created, this not mean however, that each of those pages have been allocated a corresponding frame. They are just available in the future if needed by the process. This is to reduce wasted space in the physical memory. The above figure, only allocates four pages to the process, but what if that number was in the tenths, to allow for a lot of memory if needed later in the process lifespan. At creation it might only use a few, which would mean a large number of actual physical frames are kept empty, but unavailable for other processes. This is the wasted space we do not want to happen.

When using paging, the process also dedicate the pages for different segments. There are three of these and are called code, heap, and stack. The code segment is for the programs code, it is where all the instructions are stored. The heap is for dynamic memory allocation, global variables and such. And lastly the stack is for temporary / local variables in functions. Both the code and heap segment is accessed by using addresses, whereas the stack is only accessed by the “Pop” and “Push” instructions. Where the address used is determined by the stack pointer, which points to the top of the stack.

There are some other key characteristics for the stack, that differentiate it from the code an heap, it grows in the opposite direction. Both the code and heap grows from the top. See Figure 2.2 at frame 0. The top memory location is at address 0000, then the next is at 0001 etc. For both the code and heap they will use the top available address (if needing to write/store to a new address), when writing to those segments. This is where the stack is different. The stack starts at the lowest memory location, I.e 0011, when writing to a new location, and grows up. Now another way of looking at it, is that the code and heap starts using the lowest address possible, meanwhile the stack starts using the largest address possible.

Now how is the memory actually implemented, well simply it is a slice that holds slices [ ][ ]. Where the first slice holds the list of frames, and the frames are slices themselves where they hold memory locations or addresses. With this solution there is a need for two numbers, to access a particular value. The first is what has been mentioned earlier, the frame number. The second is what is called an offset.

As mentioned every physical frame has a number of accessible memory locations, the offset is used to differentiate these. It is the “offset” from the top of the frame, and are numbered like this {0, 1, 2 ….}. So the an offset of 2, is the third memory location from the top, as the numbering starts from 0.

How the CPU accesses the memory will be talked about later.

3. MMU (Memory Management Unit)

- Translates virtual addresses to physical addresses.

The main and only job of the MMU is to translate a virtual address to a physical one, as stated before. Since the memory management system in use is paging, the MMU needs to have access to the page table we mentioned earlier, to do the translation. It only has access to one page table at a time, for the current running process in the CPU. As mentioned, each process has a separate page table.

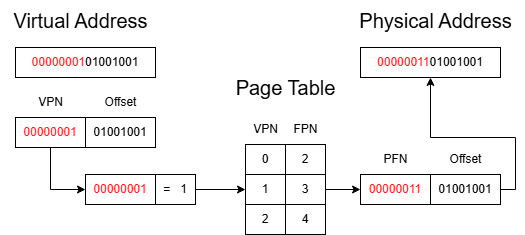
A memory address consists of two parts: The VPN/PFN and the offset. 

Figure 2.3 - Memory Address Translation

The MMU split the virtual address, which consist of the VPN and offset, into variables. Then the MMU looks at the page table, to find the corresponding frame (PFN) for that VPN. It then combines the new PFN it got, with the old offset, to create a new memory address for the physical memory. Where the only difference between the two addresses are the left side of an address representing the VPN or PFN. It keeps the same offset, as both the VPN and PFN has the same number of memory locations.

There is also some error handling that the MMU does when translating. The MMU checks if the Page Table Entries (PTE) is valid. There are actually more information that PTE holds than just the corresponding frame number as previously told. There is also other fields used for various tasks. In x86, one such field is read and write, which as the names might imply, indicate whether the you are allowed to either write or read to that page. The are many others but the only field used in this simulation is the “Valid” flag. Which indicate whether the entry is valid or not. There are two cases in which it can be invalid. The first one being if there are no corresponding frame for that page. In this case, using that page table entry, will result in an error, as the new physical address will either have the frame number as 0, or simply panic. Both cases are bad, as such the valid field is a useful method of preventing this.

The other case the flag can be invalid, is when the page is a guard page. There are two such pages, and those are for separating the memory segments: Code, Heap and Stack. Lets say the CPU is looking for the next instruction to be run, but it has already run the last instruction in the code segment(can be multiple pages). How would the CPU know the data in heap is not a valid instruction? It can’t. Well not as easily. The solution is to add a guard page. The guard page sits between the code and heap, so when the CPU tries to go past the code segment to look for a new instruction it won’t go into the heap, but rather to the guard page first. This tells the MMU that the CPU has gone to an invalid page, and the OS can handle the response, what that may be.

The other guard page sits between the heap and stack for the same purpose. Remember that the stack grows up, so that can also try to “hit” the heap.

1. Memory Controller

- Reads from memory.

- Writes to memory.

The memory controllers job is to bridge the CPU to the Memory. It is this module that actually reads or writes to the memory. The CPU has asked the MMU to translate a virtual address and has been given physical address. This physical address will be used to access the physical memory. Remember that the physical memory is a slice of slices [ ][ ], and the PFN and offset is attained by splitting the physical address. That is what this module does. It splits the physical address, into a PFN and offset, and uses these to access a particular memory location. [VPN][Offset]

6. Process Controller

- Create a new process.

- Control the allocation of memory space.

- Store instructions

This module does not exist in any form modern system, specifically x86. The functions this module executes are all handles by the OS, in x86. But to try and separate some of the OS’s tasks, so it makes simpler to both understand but also edit. A decision to create a new module to manage these tasks. The OS is still responsible for actually using this module, it is not a separate entity. It is used for gathering a subset of the OS’s functions, and put them in a new struct.

The module’s job is to control processes, hence the name. It is this module that does the process creation. It takes in a list of instruction which can either come from a file that the user can input, or from a creation process inside the program itself. When creating a new process from within the program, it is also saved as a file, that can be used later. It first gives the process a virtual memory, and then inputs these instructions into the code segment for that process.

The second job of this module is to allocate new memory frames for a process. As mentioned earlier, a page table entry for a particular process might not have a corresponding frame. If the process needs to use this entry, the process controller can allocate a frame for that entry. It keeps track of the available frames using what is called a “freelist”, which is simply is a list of available frames. The way it is implemented in this simulation, is that the freelist is a slice which holds bools. And the index correspond to a specific frame, so if the bool for a particular frame is true, it would mean it is available and not in use by another process.

It does not keep track of which process has which frame, and only keeps track if a frame is available or not. When giving a particular frame, the process inputs this frame number into the page table. So if a process wants to deallocate a frame for whatever reason, it needs to only give the frame numbers it wants gone, it the freelist can update the bools for the frames to true again.

1. OS

The largest module is the OS, it holds a pointer to every other module, and manages the other modules. It is the brain of the operation if you will. The main function of the OS in this simulation is to do what is called a context switch. The OS holds a list of all created processes, and is the one who gives the CPU the processes to be run. This is what is called a context switch, it switches between processes. The word context comes from the fact switches the registers that the CPU currently holds, to different values that comes from another process, it switches contexts. From one process context to another.

The way it works, is that something has triggered a context switch, it could be from the OS itself or elsewhere. Now the OS needs to schedule a new process to be run on the CPU, but it can’t just swap to a new process and call it a day. If the current running process was not completed when triggering the context switch, it will need to be run at a later time. It would not be optimal if the process needs to start from the beginning, so there has to be some mechanism to allow the process to be run from the same place, when entering the CPU a second time.

To to this, the OS stores the current registers in the CPU, as these holds all the information needed to be run from the same conditions from the first time. These registers are stored inside the Process Control Block (PCB) of a process. A process are just a collection of instructions, but the OS sees the processes as PCBs, where they hold valuable information about the process, like the page table. The process themselves don’t hold the page table, nor do they know it exist. The page table sits inside the PCB, and the OS connects the PCB to a process. The PCB holds other information like the state of the process. Is it runnning, blocked or terminated, etc?

When doing a context switch after storing the registers of the current running process, the OS will then ask the scheduler to give it the next process to be run. The OS does not decide the next process to run, this logic sits inside the scheduler. When the OS has attained the next process it can begin the next step. Which is to replace the page table in the MMU. As mentioned the MMU has access to the page table for the running process, and it is the job of the OS to give this access. After this the OS does the actual switching of the processes in the CPU. It does so by replacing all the registers by the new ones stored inside the PCB of the next process to be run.

The last step is for the OS to decide to queue up the process that was just being run. If the OS does decide to requeue the process, it sets the status as ready, indicating it is ready to be run, and asks the scheduler to add it to the queue.