NTNU – Trondheim
Norwegian University of
Science and Technology

X86_64 Assembly language

Assembly: Yet Another Programming Language?

- From the hallowed pages of the Merriam-Webster Online Dictionary:
  - "Compile":
    - To compose out of materials from other documents
    - To collect and edit into a volume
    - To build up gradually
    - To run (as a program) through a compiler
  - "Assemble":
    - To bring together (as in a particular place or for a particular purpose)
    - To fit together the parts of
  - The Compiler edits things, discovers, synthesizes language information
  - The Assembler substitutes text for numbers
    - Corollary: The Assembler is dumb as a brick™
    - Let us stop here

Names for things

- 'x86' is generally bandied about to talk about
  - A line of processor families
  - The instructions they support
  - The common design of systems they are often found inside
- IA-32 was a little more specific
  - That's a specification of a bunch of binary sequences, and what they (ostensibly) do to a slab of transistors
  - It's an Instruction Set Architecture (ISA), complete with symbolic names for all the binary sequences
  - Technically, an assembler could use different names, but there is no point in renaming the entire instruction set when it already has perfectly good ones

So, IA-32 begat IA-64, then?

- Well, *kind of*.
- IA-32 was backwards compatible right back to the pleistocene era, so it retained many Very Interesting design decisions
- IA-64 struck a blow for elegance, throwing legacy to the wind.
- This gave us the *Itanium* and *Itanium2* processors, with
  - Remarkable throughput
  - Remarkably delayed design and production
  - Remarkable price tags
  - Remarkably few customers
- They *have* found niches in the server market, three people eagerly await the revolution.

- We don't like to talk about it.

In the meantime

- While IA-xx was busy reinventing itself, the commercial value of corny legacy design was recognized elsewhere, spawning a 64-bit architecture initially called “Hammer”.
- If you have a “regular PC” (or Mac) these days, it's most likely a descendant of that design, which is colloquially called 'x86_64', in order not to hurt any feelings.
- If you look a little closer, some people also call it 'amd64', which gives away what kind of feelings we're not hurting.

- We don't like to talk about that either.

Why the name salad?

- By now, That Company have regained enough of an edge to decide what ISA extensions should look like again, and taken the opportunity to re-re-baptize the whole enchilada as “Intel® 64”
- I'm not going to say “Intel® 64” very often, the ® is hard to pronounce
  - So x86_64 it is, even though that may also refer to chips, systems
  - We only need to talk about the ISA, so it'll be ok
- I'm mentioning all this for reference, you may come across any and all of these names in different places
  - At least it will suffice until the next one is called “Frankenstein ♬ 128” or something like that

Is this really useful?

...well, apart from in compiler construction?

- Courses you have been through may have indicated that
  - Compilers produce better low-level code than humans
  - Assembly code is hard to write and maintain
  - Executable code isn't human readable
  - *et cetera*
- There is enough truth in this to say it, but still:
  - Compilers must adhere to language semantics, humans can see shortcuts compilers can not take
  - Assembly code can interface with other languages, and be inserted where it counts (keeping it short and semi-readable)
  - Executable code is simple, just very tedious to deal with
  - *et cetera*

Hence, the <u>motivation</u>

- Picking up a smattering of assembly is good for
  - Knowing roughly how your code in any language ends up
  - Being able to insert it by hand on special occasions
  - Writing specialized code generators for particular problems
  - Finding it out when a compiler you use has a bug
  - Succeeding at job interviews which ask about it (yes, actually)
  - Unifying theories of quantum mechanics and gravity
  - Recovering lost socks from the washing machine

- Even if you never write another compiler, this is a useful takeaway

Once more

- A process! (Figure pinched from McCusick&Neville-Neil, 2005)

In memory

On disk

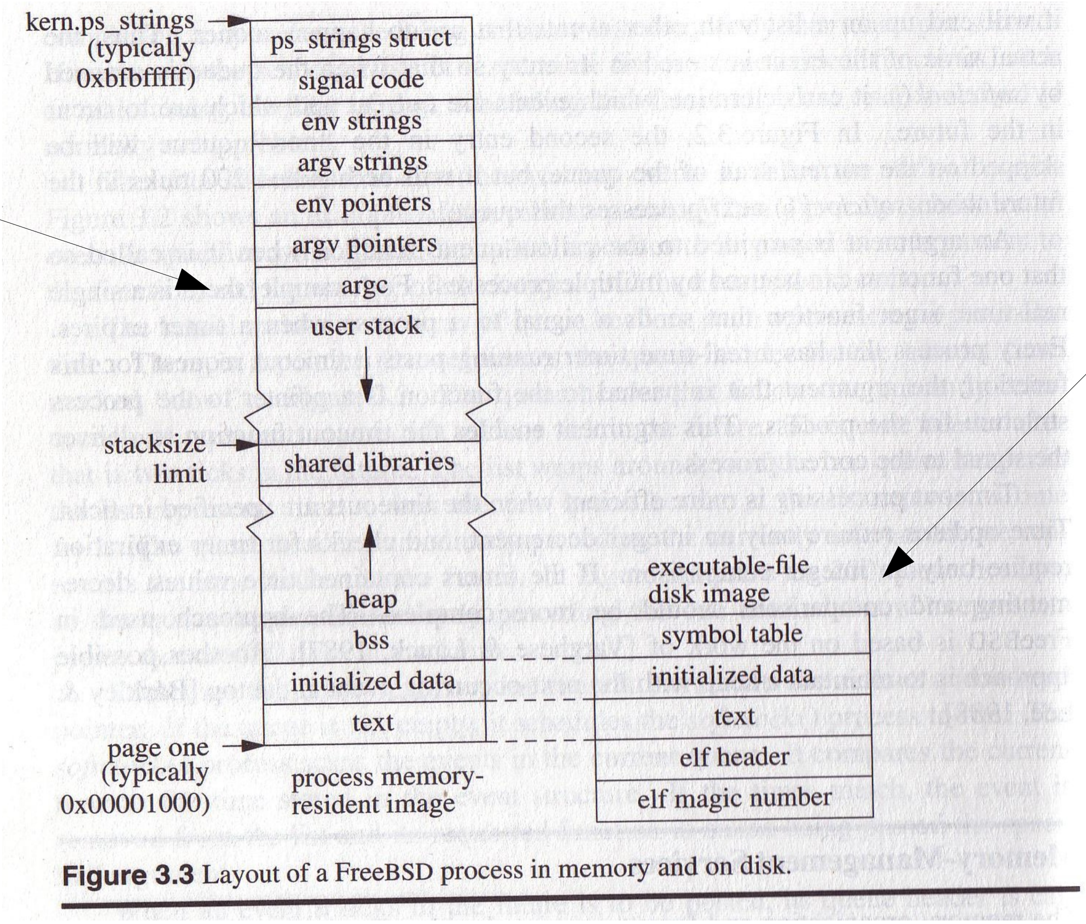
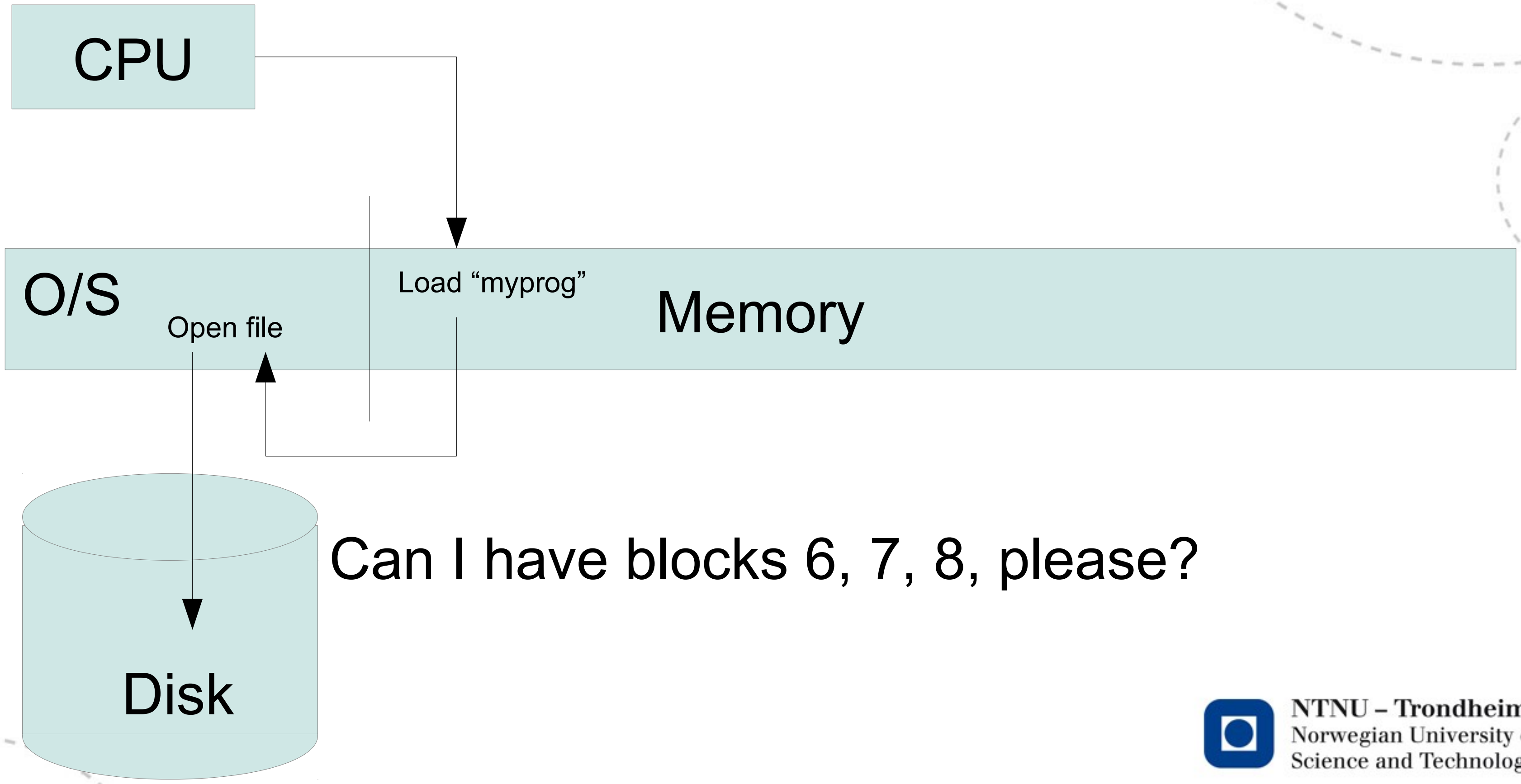


Figure 3.3 Layout of a FreeBSD process in memory and on disk.

NTNU – Trondheim
Norwegian University of
Science and Technology

The actions of the loader

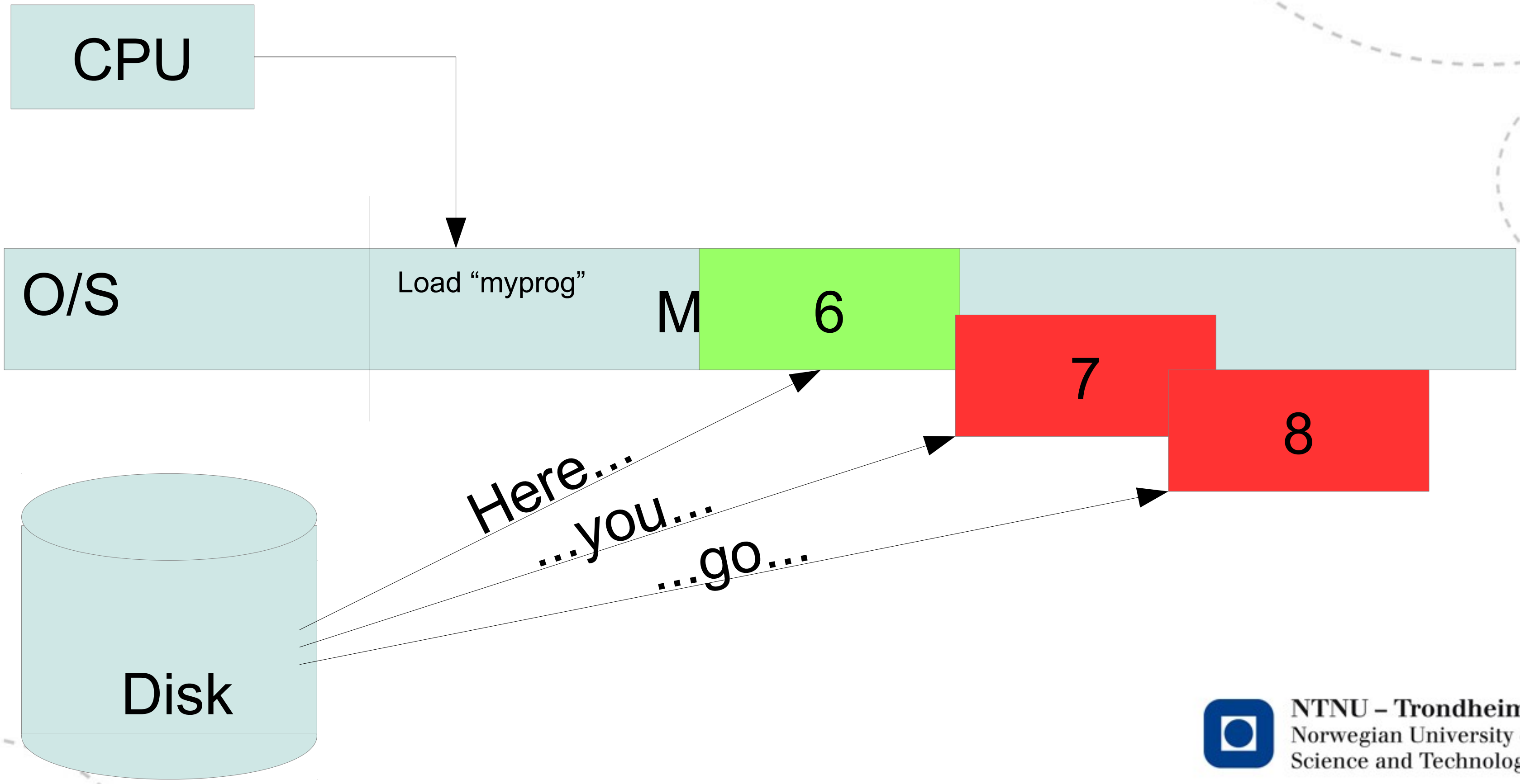
(in principle)



Can I have blocks 6, 7, 8, please?

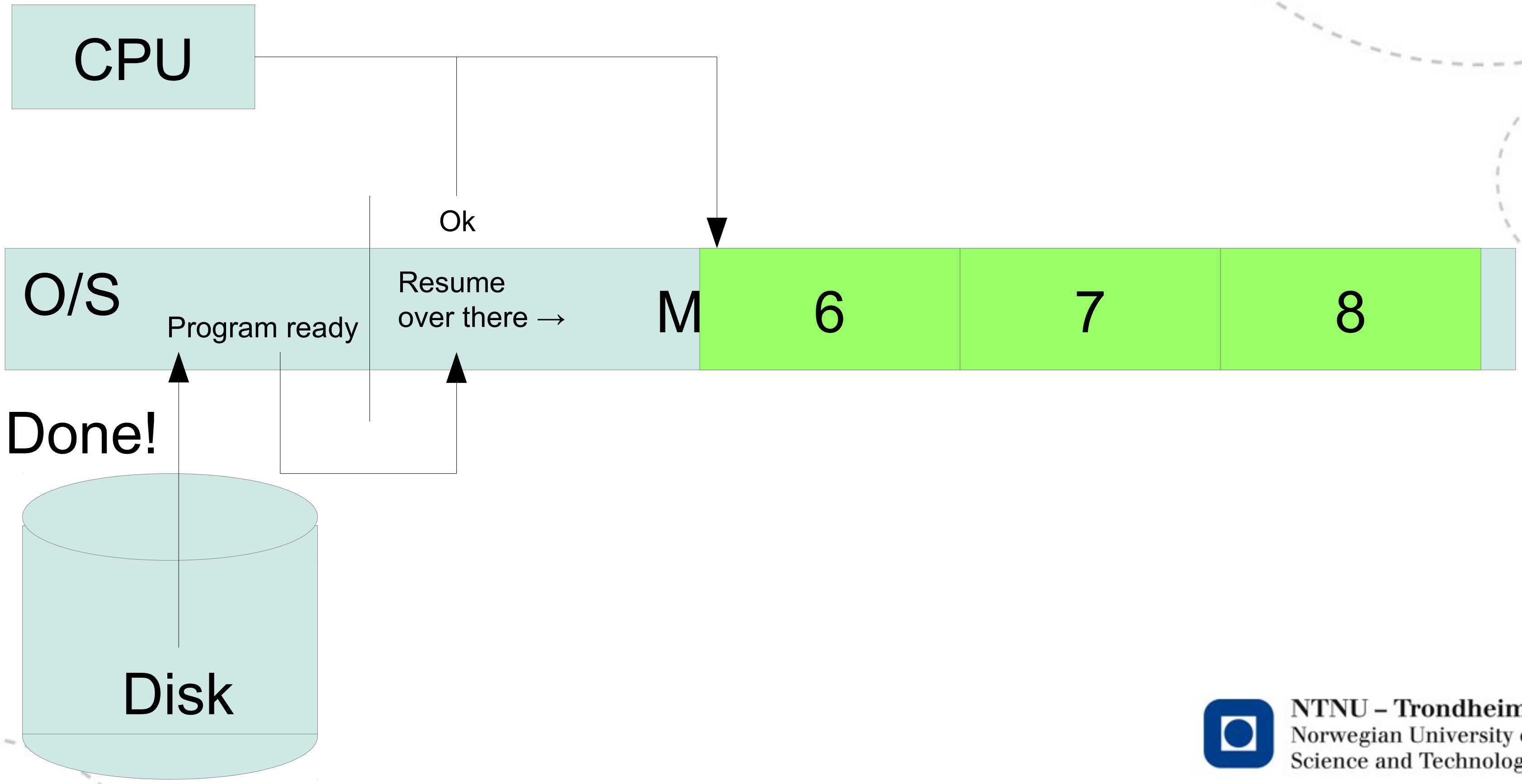
The actions of the loader

(in principle)

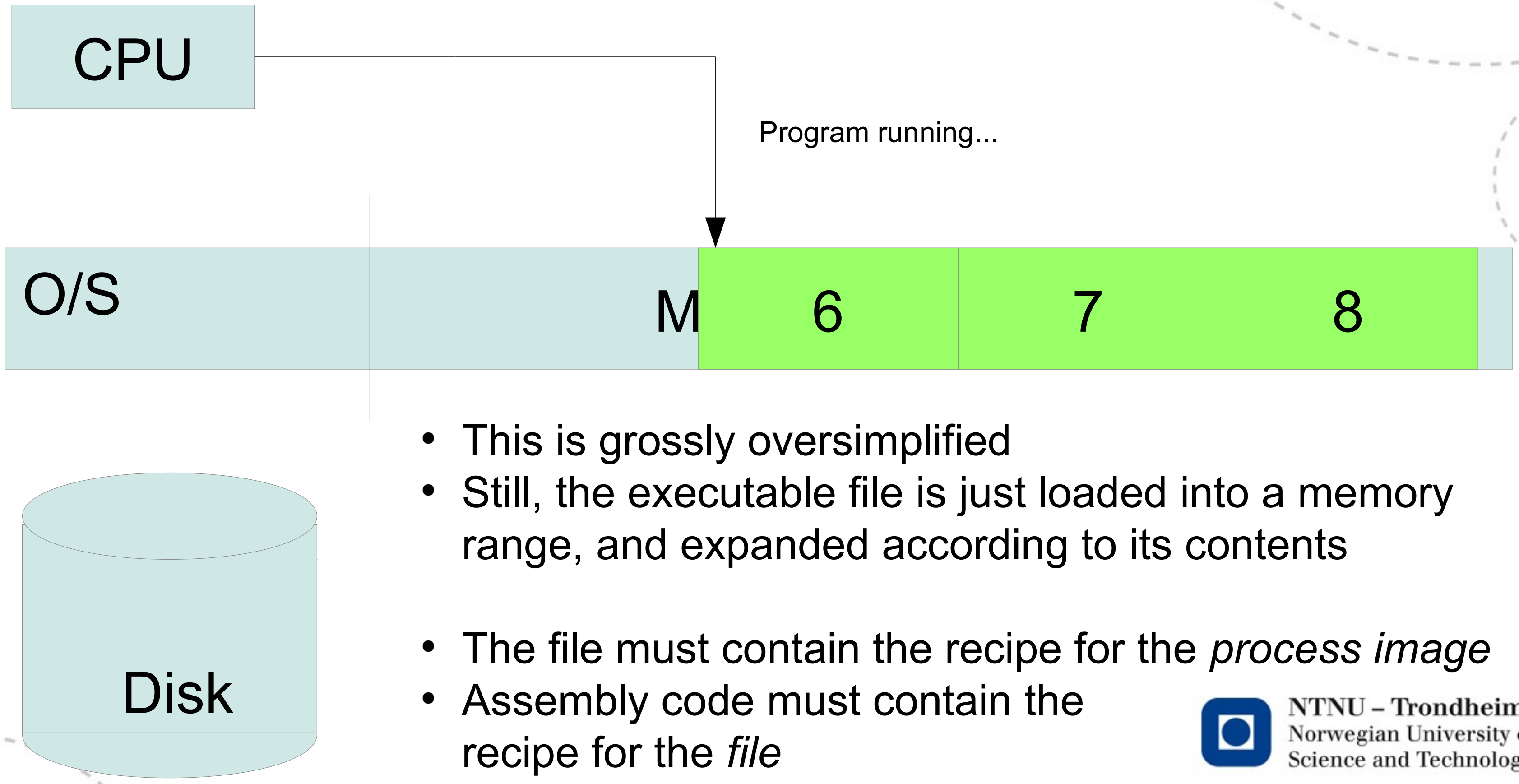


The actions of the loader

(in principle)



The actions of the loader
(in principle)



- This is grossly oversimplified
- Still, the executable file is just loaded into a memory range, and expanded according to its contents

- The file must contain the recipe for the *process image*
- Assembly code must contain the recipe for the *file*

The main parts

- The 2 parts which map directly are the data and text segments of the file
- Where they begin and end is directly evident in the code:

  .section .data ← This opens the data section

  .section .text ← This opens the… oh, I'm sure you can read

- The data section contains data (wow!) that can be
  - specified raw (if necessary), or
  - translated from friendly directives like ".string", which saves you from looking up ASCII table values and such.
- The text section is filled with instructions generated from symbolic names for operations that the CPU supports
  - This saves you from looking up op-code table values, hand-calculating how many bytes apart things are stored, and a few other tasks that are as exciting as reading a telephone directory.

Naming locations

- It is convenient to be able to talk about locations in the file by symbolic names, instead of counting bytes
- The alternative is to re-calculate all subsequent addresses every time you insert something. If you tried it, programming with line numbers is a gentle cousin to this type of pain.
- If you start a line with, *e.g.*

  ```
  farfegnugen:
  ```

  the assembler will treat this label as an integer, but calculate wherever-it-is for you.

A pointer to a string

```
.section .data
hello: .string “Hello, there!\n”
```

- Even though it doesn’t contain a program, this code already employs the assembler for several tasks:
  - Defining a data segment
  - Turning “Hello, there!\n” into the array [72, 101, 108, …
  - Tracking how many bytes came before it, and calling this offset ‘hello’ throughout the rest of this assembly
  - This should explain why we want a table of all strings in PS4

Memory is hierarchical

- Before dissecting instructions, we must know what they do.
  - They manipulate memory.
  - That’s it, really.
- Memory design is a trade-off between size, speed, power usage, production process, and a bunch of other things you can learn about elsewhere.
- Small is fast and expensive, big is slow and cheap.
- There are successively bigger/slower levels:
  - Registers
  - 1,2,3 levels of cache memory
  - DRAM memory
  - Swap file on disk
- When programming, we only see registers and “memory”, but the same address can be housed in any of these parts
- For computationally demanding programs, the way they use memory determines how fast they run, so the Dragon mentions it.

The registers

- There are 8 “general purpose” registers inherited and extended from IA-32:

– RAX	(Result accumulator)		
– RBX	(Array base pointer)		
– RCX	(Counter)		
– RDX	(Data destination pointer)		
– RSI	(String op. Source)	←	Yes, 2 in 8 are mildly text related
– RDI	(String op. Destination)	←	I am not making this up
– RSP	(Stack pointer)		
– RBP	(Frame pointer)		

(All have special roles for certain instructions, so the generality of their purpose is disputable, but they are still called that.)

- There are even less general ones with dedicated instructions
- There are *really* more than 8, but those are managed transparently, programs use only these 8 names

NTNU – Trondheim
Norwegian University of
Science and Technology

The registers, pt. II

- There is also
  - R8
  - R9
  - R10
  - …
  - R15
- Coupled with the previous 8, this makes an oddly half-consistent naming scheme.
- Presumably, the designers ran out of fancy side effects.

The rest of memory

- The style of the specification takes a cue from CISC (Complex Instruction Set Computers), providing hundreds of operations that do all sorts of useful things.
  (One highlight of convenience is the RSQRTPS instruction, which computes 4 reciprocal square roots in parallel…)
- This is legacy, complex things are reduced to sequences of more economical micro-operations for the execution core, but we don't get to see it.
- As part of that legacy, it's not an explicit load/store design – memory is accessed by each instruction supporting a large set of addressing modes instead:
  - Register (straight-up number value)
  - Register indirect (value in memory, pointer in register)
  - Register indirect + offset (same, offset by constant)
  - (Instruction-pointer relative, which I won't talk about)
- Main memory can be either source or destination, but only one at a time

Moving data

- This is the most basic of operations, and makes good examples of addressing modes:
- `movq $3, %rax /* Copy constant 3 to RAX */`
- `movq %rax, %rbx /* Copy contents of RAX to RBX */`
- `movq %rax, (%rsp) /* Copy contents of RAX to addr. RSP */`
- `movq 8(%rbp), %rbx /* Copy contents of adr RBP+8 to RBX */`

NTNU – Trondheim
Norwegian University of Science and Technology

Why mov**Q**?

- q means address as 'quadword', which is 64 bits (8 bytes)
- l would be 'longword' (32 bits / 4 bytes)
- w is 'word' (16 bits / 2 bytes)
- b is 'byte' (8 bits / 1 byte)
- This is an artifact of AT&T assembler syntax, not of the instruction set
- Therefore, you won't see them in the processor manual, but it doesn't take a lot of imagination to add them
- We'll only work in quadwords, just add 'q', and you'll do fine

Add, subtract, increment, decrement

- addq op1, op2

  adds numbers together in 2nd op. Addressing modes as per the move instruction
- subq op1, op2

  wins no awards for subtlety, it subtracts op1 from op2
- incq op1

  is a simple increment
- decq op1

  is the matching decrement

  (Bonus points: how would you translate a=b++ and a=++b, respectively?)

NTNU – Trondheim
Norwegian University of Science and Technology

Integer multiply

- That should be a natural continuation: mulq op1,op2 right?
- I wish.
- There are a few variations available, but all suffer from the same issue: the product of two quadwords can be disturbingly much larger than a quadword
- Therefore, this instruction employs seriously funky mojo by using RAX and RDX together, with high-order bits in RDX and low-order ones in RAX
- This is denoted RDX:RAX

Integer multiply

- A reasonably humane way to get the integer multiply to work, is to
  - Load RAX with one of the operands
  - Extend it to RDX:RAX
  - Use the instruction format 'imulq op1' which takes RDX:RAX as its other operand implicitly, and stores the result there as well
  - Use RAX as the result, merrily assuming that it didn't overflow

“Extend RAX”?

- Yes – integers are stored in 2-s complement, so negatives are a little funny
- That is, you find a negative number by flipping all the bits of the positive number, and adding 1
  - 00000001 is 1, so
  - 111111111 is -1
- For brevity, pretend that RDX, RAX are 8-bit registers that extend to 16:

00000000	00000001	“1” (16-bit)
	11111111	“-1” (8-bit)
00000000	11111111	“255” (16-bit)

NTNU – Trondheim
Norwegian University of
Science and Technology

Extending RAX to RDX:RAX

- The contents of RDX when used as high-order bits should depend on the sign of RAX, but it won't know just because you have a number in RAX
- Just zeroing it isn't going to deal with negative numbers
- We *could* compare RAX to 0 and set RDX accordingly, but it's a hassle
- The 'cqo' instruction takes no operands, but does precisely this sign-dependent extension in one go

  (and obviously knocks out any RDX value doing it)

Integer division

- The 'idivq' instruction works in the same way, that is, “divide RDX:RAX by op1”
- The answer won't overflow, though
- Instead, what we get is
  - RAX contains the quotient (nearest smaller integer multiple)
  - RDX contains the remainder

  when it completes

  (Except for div. by 0, which raises an exception for obvious reasons)

NTNU – Trondheim
Norwegian University of
Science and Technology

Arithmetic shift

- A similar split applies to high/low bytes of old register names
  - *"There are odd limitations accessing the byte registers due to coding issues in the REX opcode prefix used for the new registers"*
    (Intel's own words, Chris Lomont)
- Shift values don't need high values
  - 64 is the length of a register
  - A byte of shift value will do
- sarq %cl, %rax ← shift RAX right by byte-value in RCX
- salq %cl, %rax ← shift RAX left by byte-value in RCX

Pushing and popping

- pushq op1
  - Subtracts 8 from RSP register (q is 8 bytes)
  - Writes op1 into the resulting address (%rsp)
  - You can do this yourself, but it's nice to have an instruction
- popq op1
  - Writes (%rsp) into op1
  - Adds 8 to the RSP register
  - You can do this too, but it's nice to have an instruction
  - If you no longer care for the contents of the stack, it's easier to just add to the RSP register
  - addq $24, %rsp will take out top 3 quadwords on stack

Calling functions

- A function call goes something like
  - Caller pushes any values it wants to keep
  - Caller arranges parameters
  - 'call <label>' implicitly pushes return address, *i.e.* where the instruction pointer should come back on return
  - Callee starts by pushing contents of RBP right after return address
  - Callee sets its own RBP to the contents of the RSP
  - Callee runs, putting result in RAX
  - Callee restores RSP to its RBP (which points at caller RBP)
  - 'ret' instruction implicitly pops return address into instruction counter
  - Caller's execution is restored, can now recover register values and result of call in RAX

Eccentricities of the call convention

- Where IA-32 demanded all arguments on stack, x86_64 expects the first few arguments in designated registers, before they start spilling onto stack
- We'll only need the convention for integers:
  - First 6 arguments go in RDI, RSI, RDX, RCX, r8, r9
  - Any further arguments should be pushed on stack
- Remember order reversal (think of printf)
- System calls require 128-bit / 16-byte stack alignment
  - Mind this if you've pushed an odd number of quadwords

Control flow

- jmp <label> is an unconditional jump to the address computed from your label
- cmpq op1, op2 compares the 2 ops, and sets a special status register to reflect the outcome
- j[cc] <label> conditionally jumps based on the contents of that register, and typically follows the comparison instruction. [cc] can be many things:
  - jne is jump-non-equal
  - jge is jump on greater-or-equal
  - jz, jnz is jump on equal to zero, non-equal-to-zero
  - *et cetera, et cetera*

Constants

- Constants are prefixed with $
- That is what says “treat this as a constant”
  - It's not needed when an address is expected by the operation, such as with call, jump, and friends
  - It IS needed when addresses are treated as data: if you want to push the address of labelled data element 'hello', the instruction is pushq $hello

Exporting labels

- Labels which should be visible in the object code are declared initially with directive

  ```
  .globl <label>
  ```
- This makes symbols visible to C and friends
- If we export 'main', the C compiler back end will mistake our assembly from the result of a C program, set up execution to start there, and link in the C standard lib. to play with.
- Things can be done more generally, but this is easy
- Corollaries:
  - Global variables and functions translate to labels which are exported
  - Static declarations translate to labels which are not exported

NTNU – Trondheim
Norwegian University of Science and Technology

A complete example

```
.globl main
.section .data
hello:
    .string "Hello, world! %ld\n"
.section .text
main:
    pushq %rbp
    movq %rsp, %rbp
    movq $42, %rsi
    movq $hello,%rdi
    call printf
    leave
    ret
```

This will assemble and link, just put it in a file with the .s suffix and send to gcc, it will know what to do

There is much much more

...but this is pretty much what we will need

- Manals can be found at

  https://software.intel.com/en-us/articles/intel-sdm

  instruction set reference is volumes 2A-2D

- Hopefully, hello world is a decent starting point for experiments
- Complete instruction/processor reference manuals are free for download at the link
- They're a bit long and boring to *read*, but work OK for a reference