A Bare Minimal Computer for Everyone

L. Chaloyard, F. de Dinechin, M.P. Escudié, L. Morel

lucas.chaloyard@insa-lyon.fr

CitiLab - Phénix

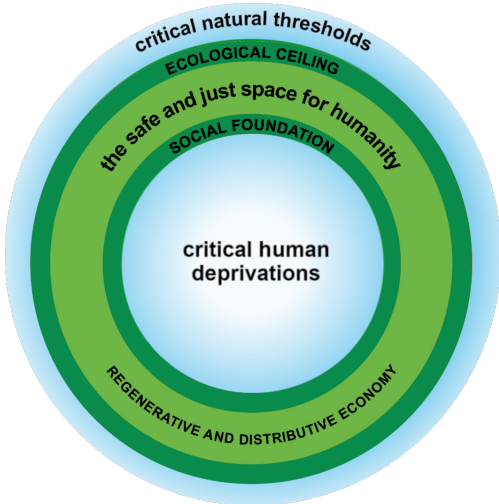
INSA Lyon

Undone Computer Science
February 7, 2024

Presentation layout

1. Social setting in which we consider our work:
   Planetary limits, social foundations and the doughnut economics
2. Properties that this setting puts on digital systems:
   Resiliency and conviviality
3. Our current object of study:
   Tiny operating system and its kind of "minimal" programming language
4. Our desired properties and a couple of research directions:
   Rethink the shape of our tools and their usage

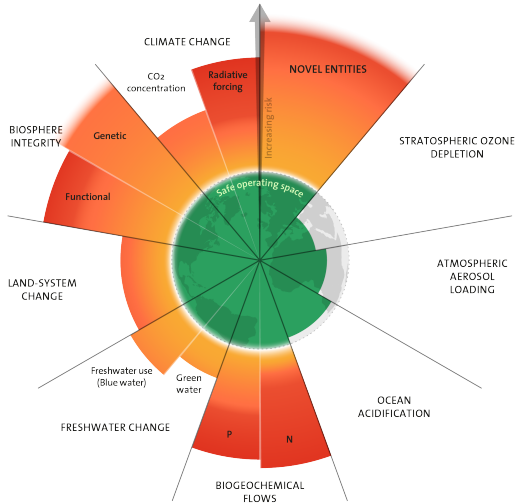
Context: Doughnut economics (1)



Credits: Kate Raworth - A Safe and Just space for Humanity (2012)

Context: Doughnut economics (2)

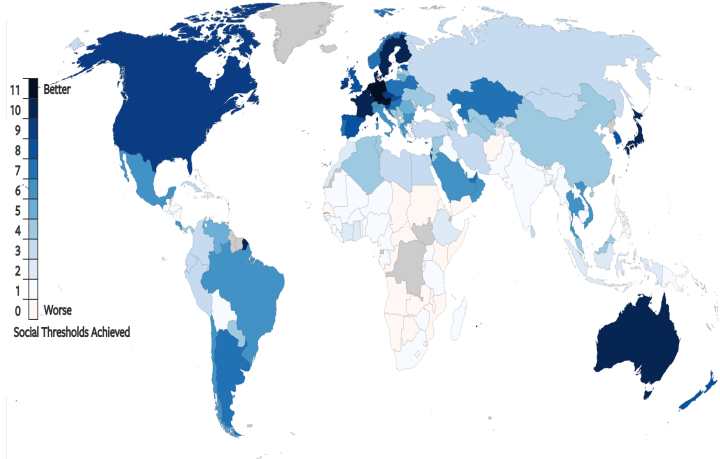
- Six boundaries assessed and crossed
- **Can we make digital tools fitting below the ecological ceiling ?**



Credits: Wang-Erlandsson et al. (2022) Stockholm Resilience Center

Context: Doughnut economics (2)

- No life essentials that is fully accessible (data from 2011*)
- Can we make digital tools fitting below the ecological ceiling **and used to achieve the social foundations ?**

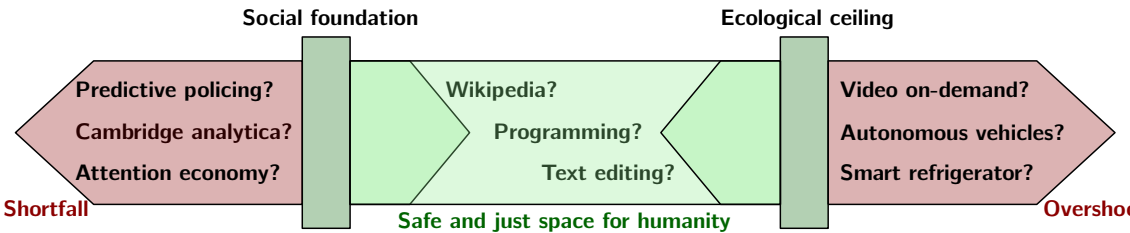


Credits: O'Neill, L. Fanning, F.Lamb, K. Steinberger (2018) - A good life for all within planetary boundaries

Context: Digital technologies

What about digital technologies ?

- Became a radical monopoly
- Used as an accelerator to a lot of human activities
- Has good and bad applications that needs to be discussed



How could we build a personal computer fitting in the doughnut ?

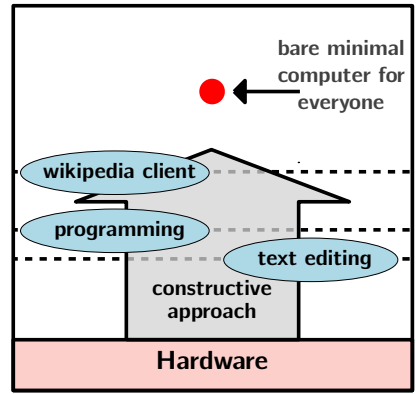
Goal: A computer for everyone fitting in the doughnut

Complexity

Quantity of dependencies needed to make, maintain and use the tool

Our approach :

- (Co-)constructive : Build until we are satisfied
  - Reducing complexity by building from a minimal
  - Building for an ethical value
  - But when are we satisfied ? (red dot)



$\Longrightarrow$ **What use cases should our bare minimal tool be used for ?**

$\Longrightarrow$ **What should a bare minimal tool look like ?**

Bare minimal computer for everyone

Two target values for the *bare minimal* computer *for everyone* : **resiliency** and **conviviality**

- Resiliency is the capacity of a socio-technical system to restore a reasonable level of social foundations after a change
- The convivial[1] structure of a digital tool still has to be defined, but according to Illich it has to protect three essentials values:
  - Survival
  - Justice : Equal possibilities and control for everyone over the tool outputs
  - Self-defined work : Similar amount of needed effort and equal control for everyone over the tool usage

We will focus on **resiliency**, **justice** and **self-defined work**

[1] Ivan Illich - Tools for Conviviality (1973)

Resilient and convivial computer (1)

What about the hardware ?

Hypothesis: sustainable hardware might be possible
Goal: frame this sustainable hardware

Several hardware specifications have an impact on the shape of the tool:

- 64-bit? 32-bit? 16-bit? 8-bit?
- RISC? CISC? VLIW? Dataflow?
- 1KB? 1MB? 1GB? 1TB?

Minimal computers: RPi 0, One Laptop per Child, “vintage” computers

We will frame the needed hardware with the software we wish to run

Resilient and convivial computer (2)

Which software bricks are we studying?

- Programming languages
- Compiler, interpreters and virtual machine
- **Operating systems** “as en Extended Machine[1]”

Several “operating systems” seemed interesting:

- Portable OSes: Thoth, InfernoOS, HelenOS, NetBSD
- RTOSes: FreeRTOS, Contiki, TinyOS, Zephyr, Riot
- Virtual Machines: Java/JVM, SectorLISP/LISP, **DuskOS/FORTH**

It has interesting properties for our definitions of resiliency and conviviality

[1] Andrew S. Tanenbaum - Modern Operating System 3rd edition (2007)

DuskOS: Case study (1)

We observed multiples interesting technical properties of DuskOS:

- The entire system and its design could fit in one brain
  $\Longrightarrow$ It can be entirely **understood** by its user
- The **portability** effort seemed reasonable even for one person
  $\Longrightarrow$ Allowing it to adapt to a change of hardware more easily
- Once DuskOS is live, it is capable of being **self-sufficient**
  $\Longrightarrow$ DuskOS will not be affected by a change of external softwares and its user need nothing else to use it

We will look into how DuskOS brings the understandability, portability and self-sufficiency properties

DuskOS kezako ?

DuskOS[1] is

- An “operating system” developed by Virgil Dupras
- 32-bit Forth environment
- Running on ARM, i386 and include a POSIX C VM
- Currently capable of running a FAT16 filesystem, a text editor, a (sub-)C compiler, ...
- Very small memory footprint, 180KB of RAM on a PC running in TUI mode with a text editor and the C compiler loaded

Also it is the big brother of Collapse OS[2]

[1] https://sr.ht/~vdupras/duskos/
[2] http://collapseos.org/

DuskOS: Case study (2)

Three DuskOS features contribute to the technical properties we identified:

- FORTH: a portable and minimalist language
- DuskOS HAL: a minimal assembler for a virtual stack machine
- DuskOS replication: cross-assembling

FORTH: a portable and minimalist language

What is FORTH ?

- (Family of) stack programming languages and an interactive environment
- Conceived by Charles H. Moore “released” 1968, for astronomical and spatial apps
- Capable of fitting a developing environment in a **restrained memory space**

3 distinct characteristics :

- No grammar, only names separated by spaces
- Those names are "**words**" contained in a "**dictionary**"
- Stack language : *2 3 + 4 * $\rightarrow$ 20*

FORTH, take a look at the beast !

```
        ( stack-before-execution -- stack-after-execution )
[REPL]> 1 333 22 3max ( -- 333 )
[REPL]> 33 -          ( 333 -- 300 )
[REPL]> .             ( 300 -- )
[REPL]> ‘‘50’’
```

FORTH allows its user to define new “word” and so to extent the system, using the “**:**” word, and definition is closed by “**;**”

```
: 3max ( a b c -- max(a,b,c) )
  2dup > if drop else nip then ( a b c -- a max(b,c) )
  2dup > if drop else nip then ; ( a max(b,c) -- max(a,b,c) )
```

That’s what we’ll call “compilation” in FORTH

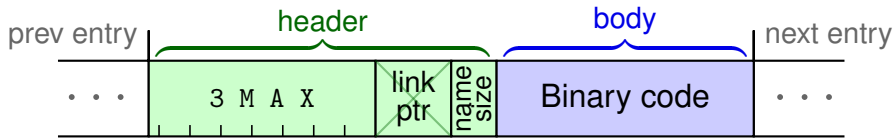
How does it work ?

FORTH Core Engine and Dictionary

FORTH Core Engine is the part of software allowing to:

- Add a new word to the dictionary (compilation, assembling)
- Find and execute a word (interpretation)

A structure called **dictionary** (a linked list) keep tracks of FORTH words available to the system



Execute a word = call to the first address of its body

$\Longrightarrow$ Core Engine mainly consists in manipulating the dictionary (a linked list)
$\Longrightarrow$ Making the core engine small, so **understandable** and **portable**

DuskOS: HAL (1)

The HAL (Harmonized Assembly Layer) is defined as :

> *a set of words implemented by all DuskOS kernels which have the same semantics and compile native code that has consistent results on all architectures.*

There is two primary usage of this HAL in DuskOS

- Assembling its own bootstrap code, allowing DuskOS to become a usable system
- Generate binary code in a cross-arch manner, making the (sub-)C compiler fully arch-independent

DuskOS: HAL (2)

The HAL is an assembler for a kind of abstract (or virtual) stack-machine implemented when porting DuskOS on a new arch.

That abstract machine can be described with the following info :

- **2 stacks** : Parameter Stack (PS) and Return Stack (RS)
- **4 registers** : W (Top of stack), A, PSP, RSP.
- **3 kinds of operands** : registers, immediates, memory addresses
- **Instructions takes one or none operands** (W is supposed as default destination but can also be used as source).

$\Longrightarrow$ Makes DuskOS kernel bigger but allows several parts of DuskOS of being arch-independent, making it more **portable**.

DuskOS: Kernel

DuskOS kernel can be divided in 3 parts :

- FORTH Core Engine (seen above)
- Harmonized Assembly Layer (seen above)
- Arch-specific code (bootstrap, configuration, ..)

On the ARM port, this is equivalent to 1000 lines of code.

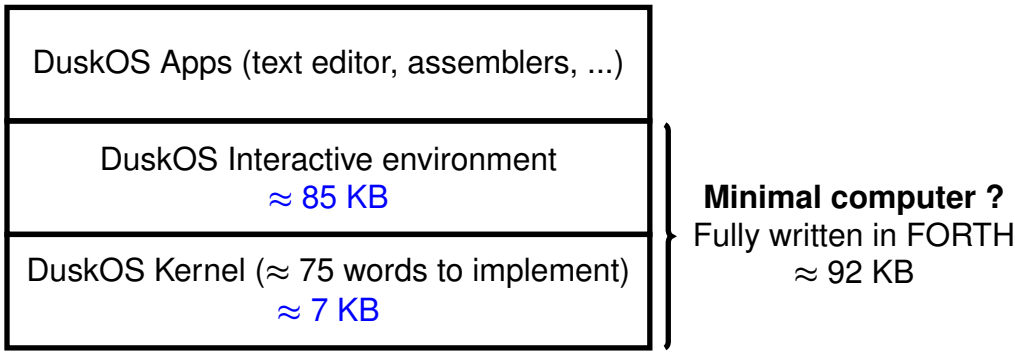
DuskOS kernel can be small thanks to :

- FORTH very **minimalist** approach.
- The HAL making several system's layers **arch-independent**.

$\Longrightarrow$ Allows DuskOS to be easily **ported** and **understood** (*by a software engineer at least*).

DuskOS: Global architecture

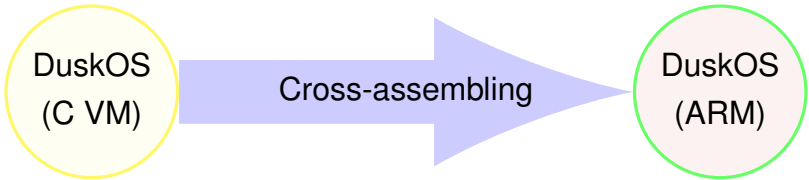
DuskOS live-system could be divided in those 3 parts :



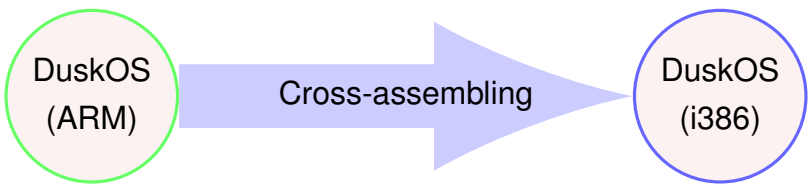
How do I generate a DuskOS image ?

DuskOS: Cross-assembling mechanism (1)

$\Longrightarrow$ **Bootstrap**: DuskOS C VM cross-assemble for new architecture



$\Longrightarrow$ DuskOS ARM can now live on its own and generate new images



DuskOS: Cross-assembling mechanism (2)

```
xcode @,
  ax HAL16B i) test, L2 abs>rel jnz,
  ax HAL8B i) test, L2 abs>rel jz,
  forward! ax $8a00 i) or,
  lblregulwr absjmp,
```

How does it work ?

- Creation in memory of a cross-dictionary
- "**xcode**" allows to assemble a word going in the cross-dictionary
- Copy of the cross-dictionary in the new DuskOS binary image
- Wrapping with some boot code, here is a **working DuskOS image**

$\Longrightarrow$ Allows DuskOS to build images of itself, and so to be **self-sufficient**

DuskOS: Case study (3)

We talked about three DuskOS features:

- FORTH: a portable and minimalist language
- DuskOS HAL: a minimal assembler for a virtual stack machine
- DuskOS replication: cross-assembling

They contribute to three technical properties we were interested in:
Portability, self-sufficiency and understandability

What those technical properties bring to our three essential values ?

DuskOS: Case study (4)

Properties \ Values	**Self-defined work**	**Justice**	**Resiliency**
Portability	Can use it without worrying about hardware	Everyone can port it on what she wants	Adapt easily to a change of hardware
Self-sufficiency	User doesn't have to use external tools	Having one DuskOS live offers every possibilities	Can maintain itself without external help
Understandability	Easily modifiable	Everyone is able to access and use the tool	The system is more easily adapted

DuskOS: Case study (4)

Properties \ Values	**Self-defined work**	**Justice**	**Resiliency**
Portability	Can use it without worrying about hardware	Everyone can port it on what she wants	Adapt easily to a change of hardware
Self-sufficiency	User doesn't have to use external tools	Having one DuskOS live offers every possibilities	Can maintain itself without external help
Understandability	Easily modifiable	Everyone is able to access and use the tool	The system is more easily adapted

DuskOS: Case study (4)

Properties \ Values	**Self-defined work**	**Justice**	**Resiliency**
Portability	Can use it without worrying about hardware	Everyone can port it on what she wants	Adapt easily to a change of hardware
Self-sufficiency	User doesn't have to use external tools	Having one DuskOS live offers every possibilities	Can maintain itself without external help
Understandability	Easily modifiable	Everyone is able to access and use the tool	The system is more easily adapted

What's next?

DuskOS is capable of a few of what we thought of as **desirable applications** (text editing, programming), but not all (wikipedia client for example)

That's why we might want to keep (co-)building DuskOS, by adding new software bricks, which would allow new desirable use of this tool

Here is a few examples of software bricks that could be interesting to add:

- Concurrency
- Memory protection
- Language typing
- ...

Should they be added and in which shape ?

Do they bring anything to the table ?

Properties \ Values	**Self-defined work**	**Justice**	**Resiliency**
Concurrency model	**??**	**??**	**??**
Memory protection	**??**	**??**	**??**
Language typing	**??**	**??**	**??**

We are not trying to decide alone, so give us your insights !

WIP: Abstract concurrency ?

If we decide that concurrency is desirable, a concurrency model could be a new tool, allowing users to make a better use of their personal computer

Adding a concurrency model would raise several questions:

- Could it be implemented with the HAL only ?
- What should the execution model be like ?
- What should the programming model be like ?

We're still exploring the different kinds of concurrency, to identify one that would "fit in the doughnut"

Two research directions

To summarize our approach :

Give the tool a proper shape

Our technical tools help us shape our world, so in order to change the current shape of our world, we have to rethink our tools

Question our needs and usage

Our needs can be answered by other means than the technological ones, using digital technologies should be a decision, not a default response

Both of them have to be tackled from a trans-disciplinary angle

Conclusion

Why is that Undone Science ?

- Recognize “economical growth” as a dogma we wont take part in
- Low or non-profit possible from this kind of research
- Trans-disciplinary research
- Challenge the “innovation is the solution”

A new methodology to design technical tools ?

- Matrix crossing technical properties and human values
- “If a technical property isn’t filling any box, should it be added ?”
- Refine our thinking by crossing with Ethical matrix[1](stakeholders / principles) and Max-Neef matrix[2](needs / existential categories)

[1]Ben Mepham - Ethical Matrix Manual

[2]Manfred Max-Neef’s Fundamental human needs

Thank you for listening !

Any questions ?