Introduction

Amandyk Kartbayev
KBTU- 2021/2022

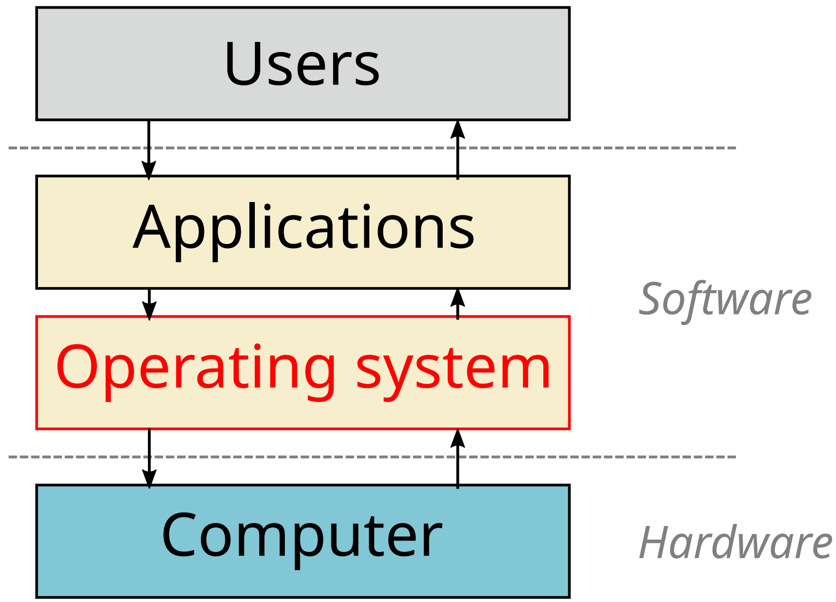
Operating system definition

Used to be a bit blurry

- Everything that was shipped by the operating system vendor
- But prone to abuse (e.g., US vs Microsoft, 98)

(Somewhat) clearer definition

An operating system (OS) is the layer of software that manages a computer's resources for its users and their applications.



Computer organization

Types of computers

Desktop computer

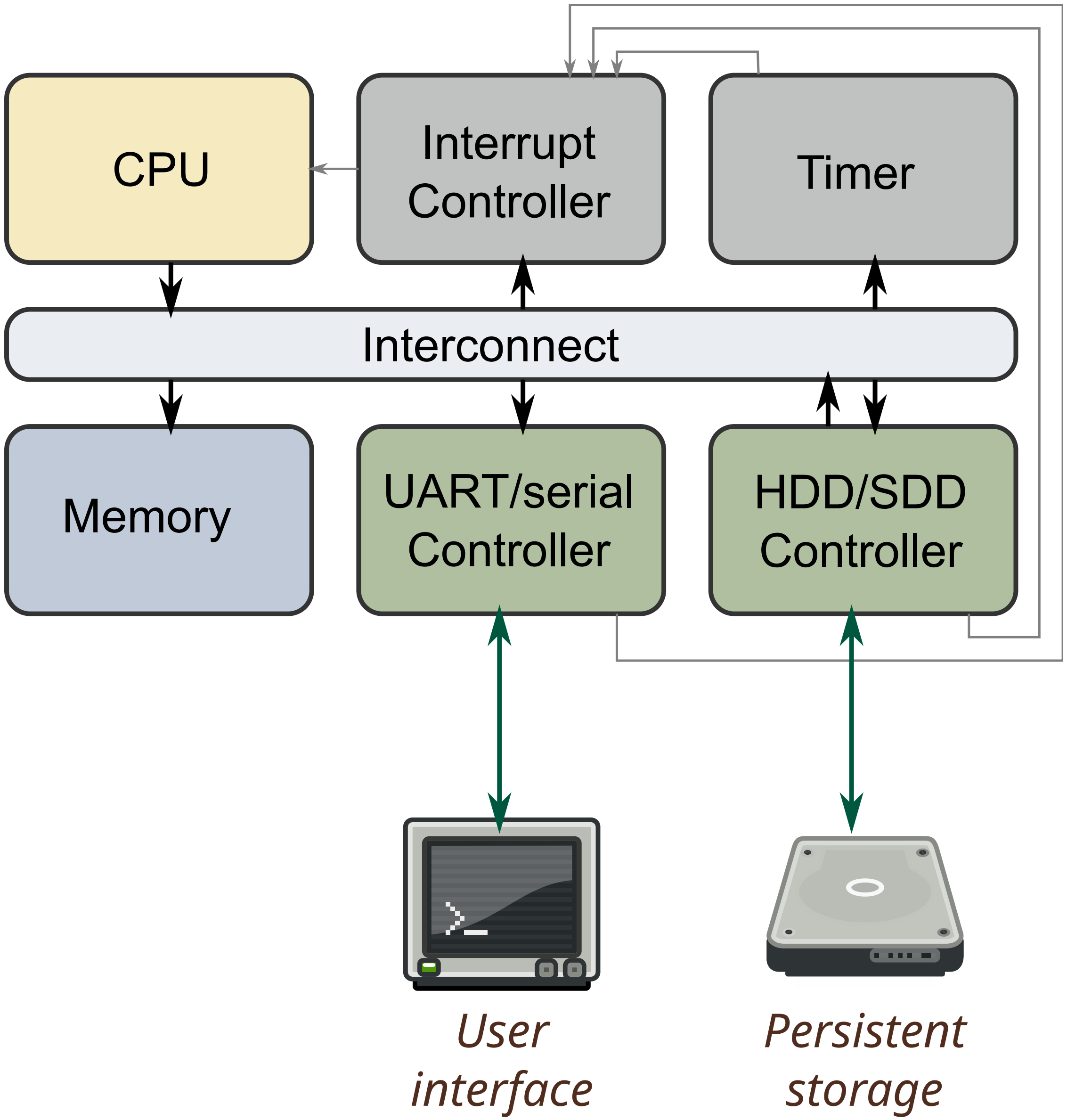
Blade server (by Dmitry Nosachev - CC BY-SA 4.0)

Internet of Things (by Miiicihiaieil Hieinizilieir, CC BY-SA 4.0)

Car computer

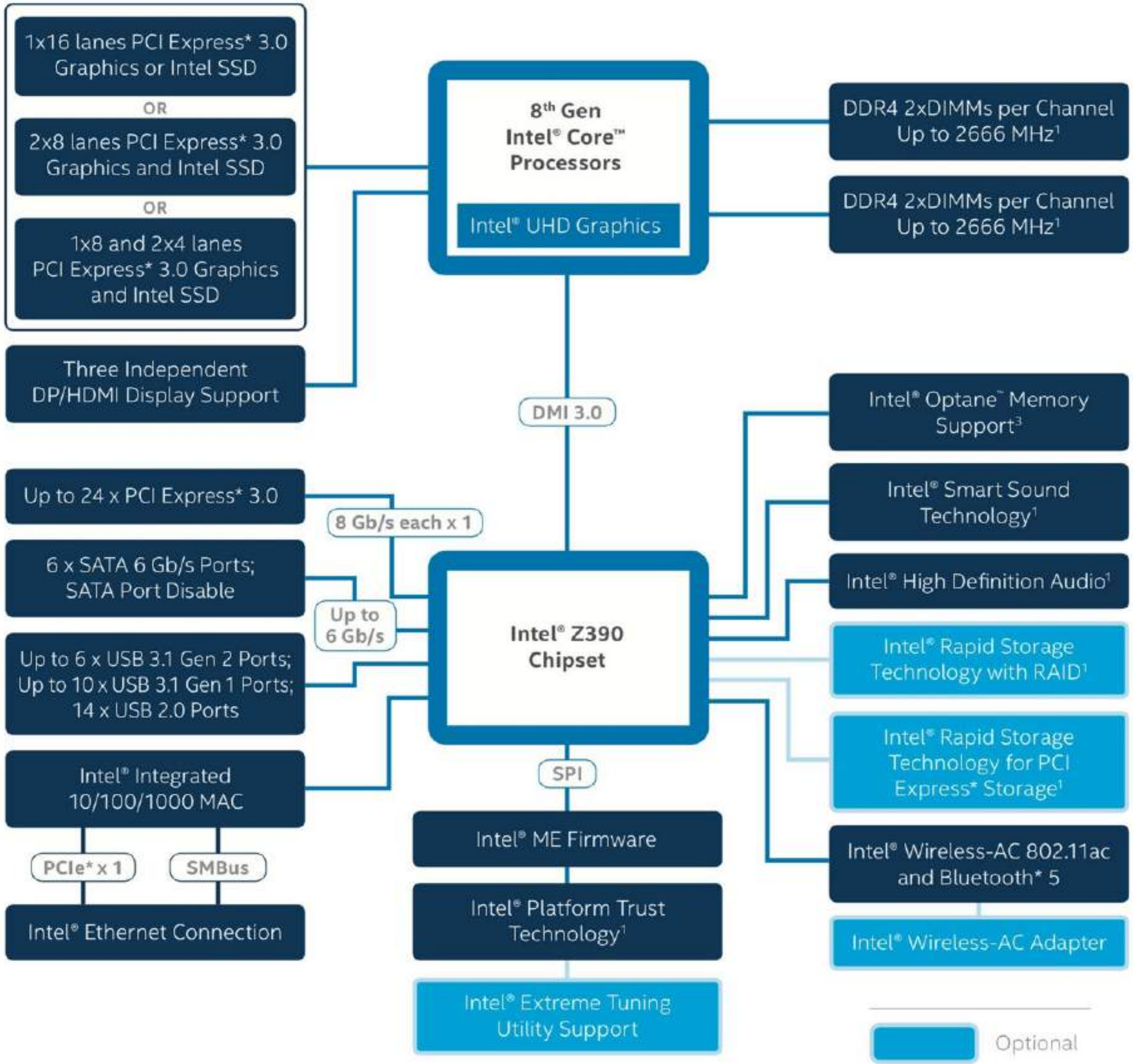
Computer organization

The bare minimum



Computer organization

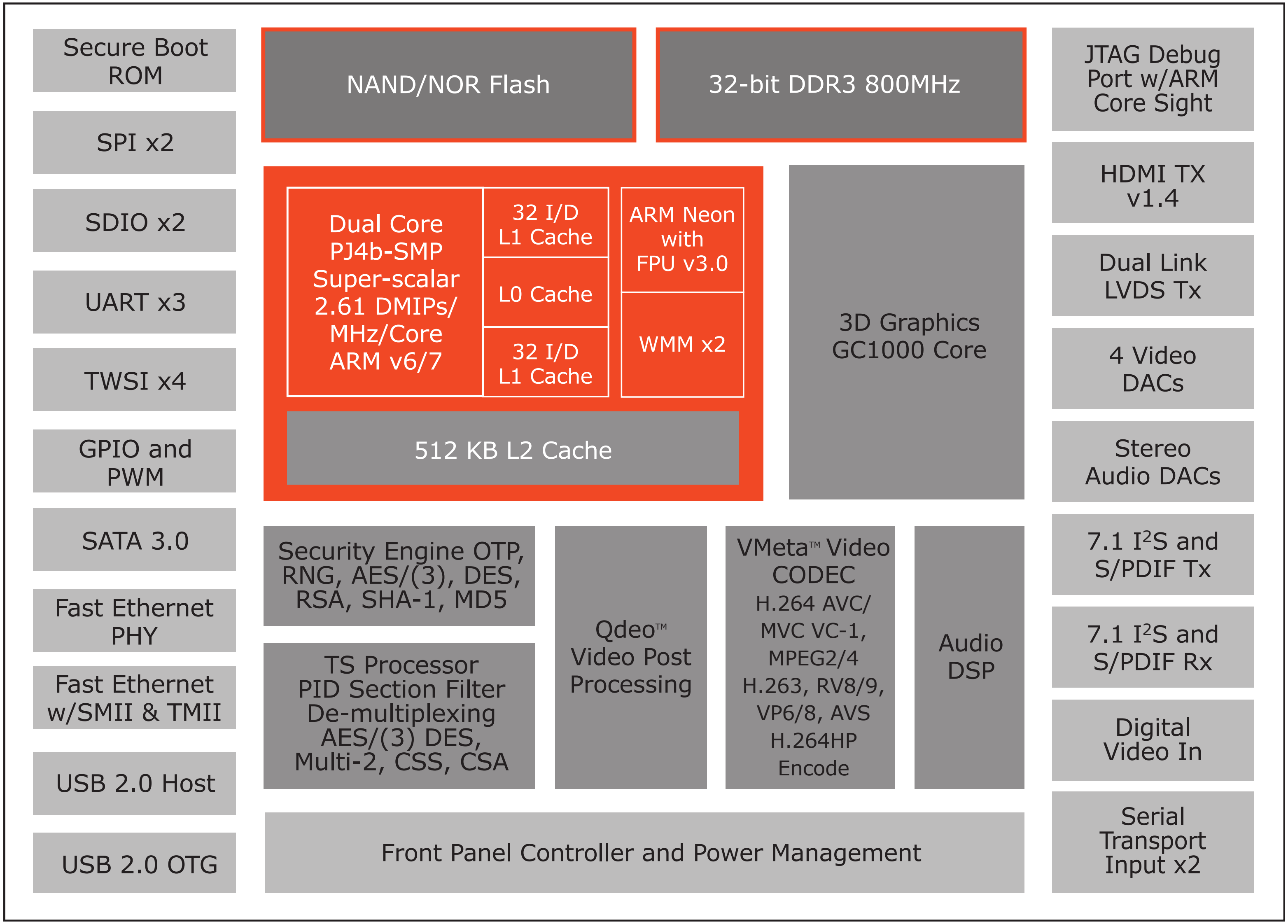
General-purpose computing



Intel Chipset Z390 (2018)

Computer organization

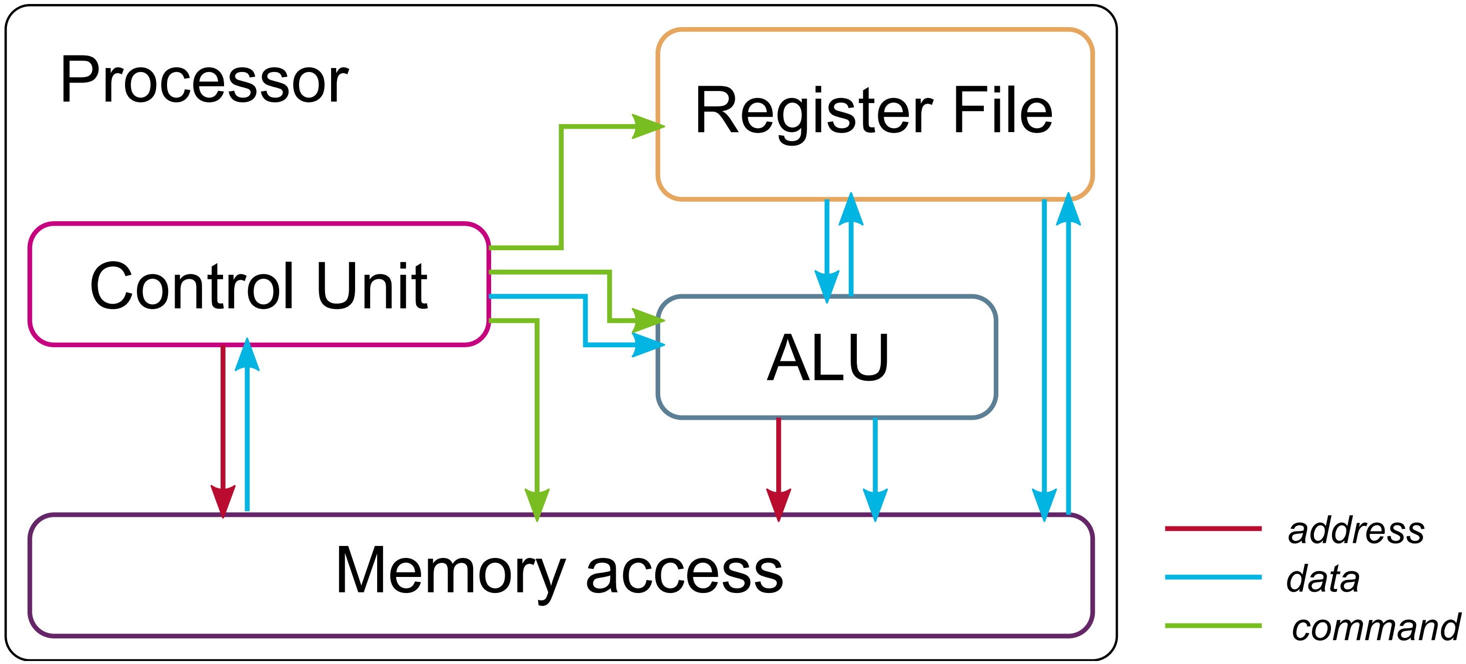
Embedded computing



Marvell Armada 1500 (similar to SoC found in Google Chromecast)

Hardware components

Processor

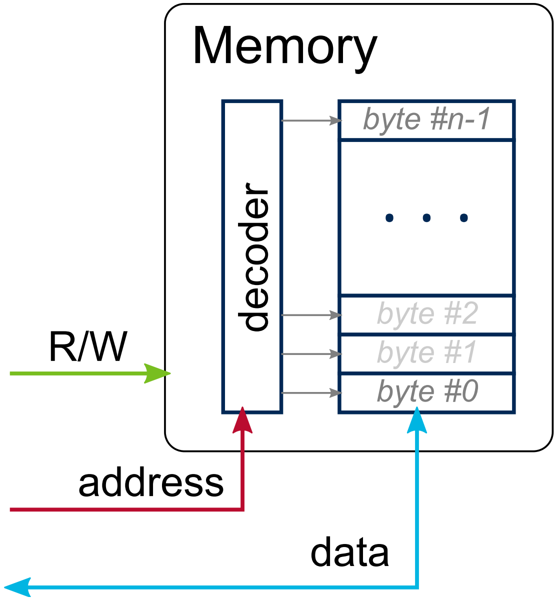


- Fetches instructions from memory, decodes them, and executes them
- Characterized by an *Instruction Set* (e.g., `i686`, `x86_64`, `AArch64`)
  - Access to memory, arithmetic/logical operations, control flow
- Contains a set of *registers*
  - General-purpose registers, program counter, status register

Hardware components

Memory

- Set of **addressable** bytes that can hold numerical values



- Organized in a hierarchy of layers (with various access times)

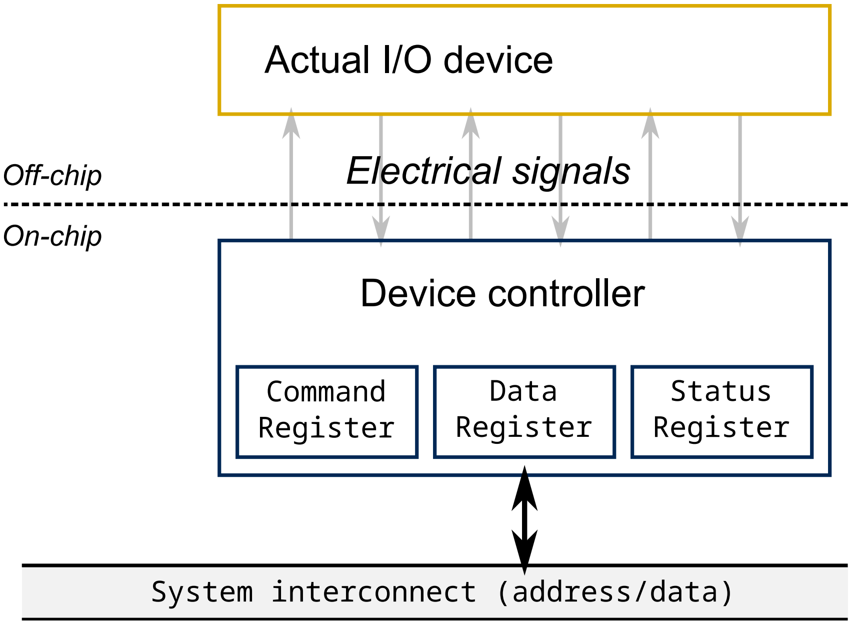
Memory layer	**Access time**	**Accessibility**
CPU registers	Immediate	Via CPU instructions
CPU cache	Few cycles	Not addressable (transparent)
RAM/Main memory	Few hundred cycles	Addressable from CPU
Second-level storage	Few thousand cycles	Indirectly addressable (see later in course)

Hardware components

I/O Devices

- Controllers connected to system interconnect
- Devices connected to controllers

- The controller provides an interface for accessing the device resources/functionalities
  - Via device registers



- Memory-mapped access
  - Device registers mapped into memory address space
  - Accessible through regular memory instructions

- Port-mapped access
  - Device registers mapped into special I/O space
  - Accessible through special I/O instructions

Hardware components

Interconnects (buses, networks)

- Transfer data between components
- Characterized by various features, speed, bandwidth, etc.

CPU to memory buses

- Cache bus (aka Back side bus in Intel lingo)
- Memory bus (aka Front side bus in Intel lingo)
  - Now implemented by AMD HyperTransport, or Intel QuickPath Interconnect

CPU to device buses

Bus	Created	Bandwidth	Type
ISA	1981	~8 MiB/s	Expansion
IDE	1986	~8 MiB/s	Mass-storage
PCI	1992	~133 MiB/s	Expansion
AGP	1997	~266 MiB/s	Video card
SATA 1.0	2000	~150 MiB/s	Mass-storage
PCI-e 1.x	2003	~250 MiB/s per lane	Expansion/Video card
PCI-e 5.x	**2019**	**~8 GiB/s per lane**	**Expansion/Video card**

OS definition, part II

Roles

In order to manage a computer's resources, an OS needs to play various roles:

- Referee
- Illusionist
- Glue

Design principles

A well-constructed OS needs to achieve various design goals:

- Reliability
- Security
- Portability
- Performance

OS roles

Referee

Manage the **resource sharing** between applications

- Resource allocation (e.g., CPU, memory, I/O devices)
- Isolation (e.g., fault isolation)
- Communication (e.g., safe communication)

Illusionist

Abstraction of hardware via **resource virtualization**

- Mask scarcity of physical resources
- Mask potential hardware failure

Glue

Set of **common services** to applications

- Hardware abstraction
- Filesystem, message passing, memory sharing

OS design principles

Reliability

- OS does exactly what it is designed to do
- Related to *availability*: OS is always usable

Security

- OS is secure if cannot be compromised by a malicious attacker
- Related to *privacy*: data is only accessible by authorized users
- Enforcement mechanisms vs security policies

Portability

- OS provides the same abstractions regardless of the underlying hardware
    - Applications
    - System libraries
    - Kernel

> *Binary compatibility is* ***so*** *important that I do not want to have anything to do with kernel developers who don't understand that importance. [...] The* ***only*** *reason for an OS kernel existing in the first place is to serve user-space.*
> *Linus Torvalds, LKML, 2012*

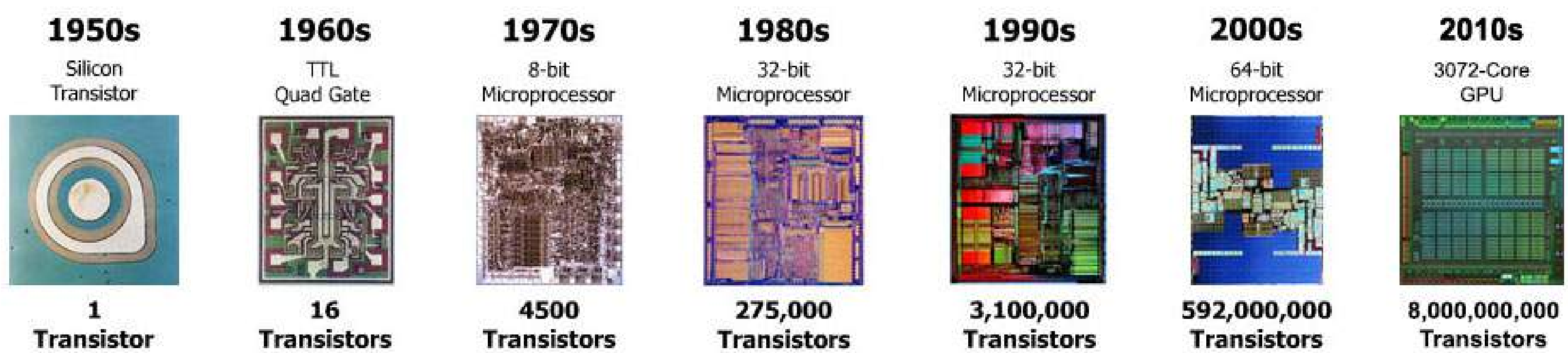
Performance

- Overhead
- Fairness
- Latency
- Throughput
- Predictability

OS history

Three major phases

- Hardware is very expensive (1955-65)
- Hardware slowly becomes affordable (1965-80)
- Hardware becomes dirt cheap (1980-present)



OS history

Hardware is very expensive (1955-65)

- Introduction of the transistor in the mid-50s
- Expensive mainframes, operated by humans
- Read from punch card, run job, print result
- Batch systems in order to better serialize jobs

IBM 7090, circa 1960

Software

- OS is simply a runtime *library* (common I/O functions)
- Applications have full access to hardware

OS history

Hardware slowly becomes affordable (1965-80)

- Use of integrated circuits
- Mainframe computers
  - E.g., IBM System/360

- Personal computers
  - E.g., DEC PDP-11

Software

- IBM OS/360
  - Multiprogramming, memory protection
- Multics
  - Timesharing, dynamic linking, security, hierarchical file-system
- UNIX, BSD/SystemV variants
  - C language
- POSIX

OS history

Hardware becomes dirt cheap (1980-present)

- Continuation of Moore's law
- Personal computers
  - Command line interfaces
  - GUI
- Pervasive computers
  - Desktops, laptops
  - Smartphones, tablets
  - Embedded systems
  - Data centers

OS history

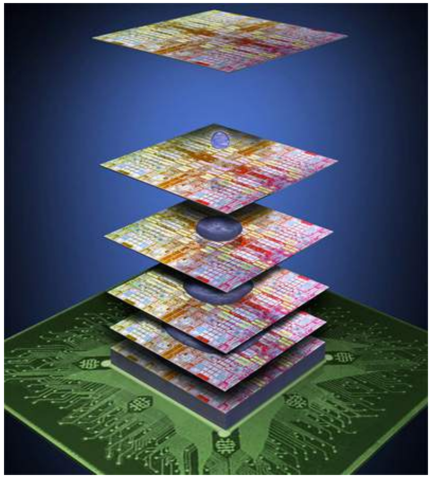
Progression over time

Metric	1981	1997	2014	Factor(2014/1981)
CPU performance (MIPS)	1	200	2500	2.5 K
CPU/computer	1	1	10+	10+
MIPS cost	$100K	$25	$0.20	500 K
DRAM (MiB)/$	$500	$0.5	$0.001	500 K
Disk (GiB)/$	$333	$0.14	$0.00004	8 M
WAN (bps)	300	256K	20M	60 K
LAN (bps)	10M	100M	10G	1000+
Ratio users to computers	100:1	1:1	1:several	100+

Numbers taken from OSPP textbook

Future of OSes

End of Moore's law



- Make better use of the same area
  - Deep redesign of processors?
- 3-D stacking
  - Multiprocessors
  - Multi- and many-cores
- Quantum computing

From the very small... to the super big

- Power-efficient IoT devices
  - Smart home, smart city, smart [blank]
- Giant data centers
  - Very large-scale storage

Heterogeneity

- Different processors on same chip
  - E.g., ARM big.LITTLE, Intel Hybrid Technology
- Specialized computing accelerators
  - GP-GPUs, FPGAs, AI accelerators

Manual approach

Code example

main.c

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

#include "fact.h"

int main(int argc, char **argv)
{
    int n;
    if (argc < 2) {
        fprintf(stderr,
                "Usage: myfact number\n");
        exit(1);
    }
    n = atoi(argv[1]);
    printf("fact(%d) = %d\n",
           n, fact(n));
    return 0;
}
```

main.c

fact.h

```c
#ifndef FACT_H_
#define FACT_H_

int fact(int n);

#endif /* FACT_H_ */
```

fact.h

fact.c

```c
#include "fact.h"

int fact(int n) {
    if (n == 0)
        return 1;
    return n * fact(n - 1);
}
```

fact.c

README.md

```markdown
# Overview

This program computes the
factorial of a number
```

README.md

Manual approach

Compilation

```
$ gcc -Wall -Wextra -Werror -c -o fact.o fact.c
$ gcc -Wall -Wextra -Werror -c -o main.o main.c
$ gcc -Wall -Wextra -Werror -o myfact main.o fact.o
```

```
$ ./myfact
Usage: myfact number
$ ./myfact 5
fact(5) = 120
```

```
$ pandoc -o README.html README.md
$ firefox README.html
```

On the long run...

Now, what if:

- fact.c changes? main.c changes? fact.h changes?
- I want to change the compilation options?
- I want to recompile this code on another computer?
- I want to share this code?

Solution is to **automate the build process**!

Introduction

Definition

A *Makefile* is a file containing a set of rules used with the *make* build automation tool.

The two following commands are equivalent:

```
$ ls
Makefile ...
$ make
$ make -f Makefile
```

The set of Makefile rules usually represents the various steps to follow in order to build a program: it's the building *recipe*.

Introduction

Anatomy of a rule

```
target: [list of prerequisites]
[ <tab> command ]
```

- For `target` to be generated, the prerequisites must all exists (or be generated if necessary)
- `target` is generated by executing the specified `command`
- `target` is generated only if it does not exist, or if one of the prerequisites is more recent
    - Prevents from building everything each time, but only what is necessary

Commenting

- Lines prefixed with `#` are not evaluated

    ```
    # This is a comment
    ```

Version 0.1

Basic rules

```
myfact: main.o fact.o
    gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
    gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
    gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
    pandoc -o README.html README.md
```

Makefile_v0.1

```
$ make
gcc -c -o main.o main.c
gcc -c -o fact.o fact.c
gcc -o myfact main.o fact.o
```

```
$ make README.html
pandoc -o README.html README.md
```

Version 0.1

`all` rule

```
all: myfact README.html

myfact: main.o fact.o
	gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
	gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
	gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
	pandoc -o README.html README.md
```

Makefile_v0.1

```
$ make
gcc -c -o main.o main.c
gcc -c -o fact.o fact.c
gcc -o myfact main.o fact.o
pandoc -o README.html README.md
```

Version 0.1

`clean` rule

```
all: myfact README.html
...
clean:
	rm -f myfact README.html main.o fact.o
```

Makefile_v0.1

```
$ make
gcc -c -o main.o main.c
gcc -c -o fact.o fact.c
gcc -o myfact main.o fact.o
pandoc -o README.html README.md
```

```
$ make clean
rm -f myfact README.html main.o fact.o
```

Version 0.1

A first and basic Makefile

```
all: myfact README.html

myfact: main.o fact.o
	gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
	gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
	gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
	pandoc -o README.html README.md

clean:
	rm -f myfact README.html main.o fact.o
```

Makefile_v0.1

- Was good enough for Project #1
  - *(No need to generate html out of markdown --pandoc is not installed on CSIF, and also it's just for the example)*

Version 1.0

How to avoid redundancy...?

A good programmer is a lazy programmer!

```
all: myfact README.html

myfact: main.o fact.o
	gcc -Wall -Wextra -Werror -o myfact main.o fact.o

main.o: main.c fact.h
	gcc -Wall -Wextra -Werror -c -o main.o main.c

fact.o: fact.c fact.h
	gcc -Wall -Wextra -Werror -c -o fact.o fact.c

README.html: README.md
	pandoc -o README.html README.md

clean:
	rm -f myfact README.html main.o fact.o
```

Makefile_v0.1

Version 1.0

Automatic variables in commands

- `$@`: replaced by name of target
- `$<`: replaced by name of **first** prerequisite
- `$^`: replaced by names of **all** prerequisites

```
all: myfact README.html

myfact: main.o fact.o
	gcc -Wall -Wextra -Werror -o $@ $^

main.o: main.c fact.h
	gcc -Wall -Wextra -Werror -c -o $@ $<

fact.o: fact.c fact.h
	gcc -Wall -Wextra -Werror -c -o $@ $<

README.html: README.md
	pandoc -o $@ $<

clean:
	rm -f myfact README.html main.o fact.o
```

Version 1.0

Pattern rules

A pattern rule `%.o: %.c` says how to generate *any* file `<file>.o` from another file `<file>.c`.

```
all: myfact README.html

myfact: main.o fact.o
	gcc -Wall -Wextra -Werror -o $@ $^

%.o: %.c fact.h
	gcc -Wall -Wextra -Werror -c -o $@ $<

%.html: %.md
	pandoc -o $@ $<

clean:
	rm -f myfact README.html main.o fact.o
```

Version 1.0

Variables

```
CC       := gcc
CFLAGS   := -Wall -Wextra -Werror
CFLAGS   += -g
PANDOC   := pandoc

all: myfact README.html

myfact: main.o fact.o
	$(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
	$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
	$(PANDOC) -o $@ $<

clean:
	rm -f myfact README.html \
	      main.o fact.o
```

```
$ make
gcc -Wall -Wextra -Werror -g -c -o main.o main.c
gcc -Wall -Wextra -Werror -g -c -o fact.o fact.c
gcc -Wall -Wextra -Werror -g -o myfact main.o fact.o
pandoc -o README.html README.md
```

Version 2.0

More variables

```
targets := myfact README.html
objs    := main.o fact.o

CC      := gcc
CFLAGS  := -Wall -Wextra -Werror
CFLAGS  += -g
PANDOC  := pandoc

all: $(targets)

myfact: $(objs)
	$(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
	$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
	$(PANDOC) -o $@ $<

clean:
	rm -f $(targets) $(objs)
```

```
$ make
gcc -Wall -Wextra -Werror -g -c -o main.o main.c
gcc -Wall -Wextra -Werror -g -c -o fact.o fact.c
gcc -Wall -Wextra -Werror -g -o myfact main.o fact.o
pandoc -o README.html README.md
```

Version 2.0

Nice output

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make clean
CLEAN
```

```
...

myfact: $(objs)
	@echo "CC   $@"
	@$(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
	@echo "CC   $@"
	@$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
	@echo "MD   $@"
	@$(PANDOC) -o $@ $<

clean:
	@echo "CLEAN"
	@rm -f $(targets) $(objs)
```

- In case of debug, how can we still see the commands that are executed?

Version 3.0

Conditional variables

```
...

ifneq ($(V),1)
Q = @
endif

myfact: $(objs)
	@echo "CC    $@"
	$(Q)$(CC) $(CFLAGS) -o $@ $^

%.o: %.c fact.h
	@echo "CC    $@"
	$(Q)$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
	@echo "MD    $@"
	$(Q)$(PANDOC) -o $@ $<

clean:
	@echo "CLEAN"
	$(Q)rm -f $(targets) $(objs)
```

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make V=1
CC main.o
gcc -Wall -Wextra -Werror -g -c -o main.o main.c
CC fact.o
gcc -Wall -Wextra -Werror -g -c -o fact.o fact.c
CC myfact
gcc -Wall -Wextra -Werror -g -o myfact main.o fac
t.o
MD README.html
pandoc -o README.html README.md
```

Version 3.0

Generic rules vs dependency tracking

Non-generic rule

```
%.o: %.c fact.h
	@echo "CC   $@"
	$(Q)$(CC) $(CFLAGS) -c -o $@ $<
```

Generic rule

```
%.o: %.c
	@echo "CC $@"
	$(Q)$(CC) $(CFLAGS) -c -o $@ $<
```

Makefile_v3.0

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make
make: Nothing to be done for 'all'.
```

```
$ touch fact.h
$ make
make: Nothing to be done for 'all'.
```

- How can we preserve the generic rule but also have accurate dependency tracking?

Version 3.0

Rule composition

```
%.o: %.c
	@echo "CC $@"
	$(Q)$(CC) $(CFLAGS) -c -o $@ $<
```

Makefile_v3.0

```
main.o: main.c fact.h
fact.o: fact.c fact.h
```

- How can we have these additional rules be generated automatically and included in the Makefile?

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
```

```
$ make
make: Nothing to be done for 'all'.
```

```
$ touch fact.h
$ make
CC main.o
CC fact.o
CC myfact
```

Version 3.0

Use GCC for dependency tracking

```c
#include <stdio.h>
#include "fact.h"

int fact(int n) {
    if (n == 0)
        return 1;
    return n * fact(n - 1);
}
```

```
$ gcc -Wall -Wextra -Werror -MMD -c -o fact.o fact.c
```

```
$ cat fact.d
fact.o: fact.c fact.h
```

Version 3.0

Dependency tracking Makefile integration

```
targets := myfact README.html
objs    := main.o fact.o
...
CFLAGS  := -Wall -Wextra -Werror -MMD
...
all: $(targets)

# Dep tracking *must* be below the 'all' rule
deps := $(patsubst %.o,%.d,$(objs))
-include $(deps)
...
%.o: %.c
	@echo "CC $@"
	$(Q)$(CC) $(CFLAGS) -c -o $@ $<
...
clean:
	@echo "clean"
	$(Q)rm -f $(targets) $(objs) $(deps)
```

Makefile_v3.0

- `$(deps)` will be computed from `$(obj)` into `main.d fact.d`
- Prefix `-` ignores inclusion errors

Version 3.0

First run

- Dependency files don't exist but make won't complain
- GCC generates them

```
$ ls *.d
```

```
$ make
CC main.o
CC fact.o
CC myfact
MD README.html
$ ls *.d
main.d fact.d
```

```
$ cat main.d
main.o: main.c fact.h
$ cat fact.d
fact.o: fact.c fact.h
```

Following runs

- Dependency files are included by the Makefile
- They are used to compose the generic rule for object generation

```
$ make
make: Nothing to be done for 'all'
$ touch fact.h
$ make
CC main.o
CC fact.o
CC myfact
```

Final Makefile

```
targets := myfact README.html
objs    := main.o fact.o

CC      := gcc
CFLAGS  := -Wall -Wextra -Werror -MMD
CFLAGS  += -g
PANDOC  := pandoc

ifneq ($(V),1)
Q = @
endif

all: $(targets)

# Dep tracking *must* be below the 'all' rule
deps := $(patsubst %.o,%.d,$(objs))
-include $(deps)

myfact: $(objs)
	@echo "CC $@"
	$(Q)$(CC) $(CFLAGS) -o $@ $^

%.o: %.c
	@echo "CC $@"
	$(Q)$(CC) $(CFLAGS) -c -o $@ $<

%.html: %.md
	@echo "MD $@"
	$(Q)$(PANDOC) -o $@ $<

clean:
	@echo "clean"
	$(Q)rm -f $(targets) $(objs) $(deps)
```

Makefile_v3.0

GDB

GNU project

- Started by Richard Stallman in 1983
- Free software, mass collaboration project in response to proprietary UNIX
    - Copyleft license: GNU GPL
    - User programs: text editor (Emacs), compiler (GCC toolchain), debugger (GDB), and various utilities (ls, grep, awk, make, etc.)
    - Kernel: GNU Hurd

GDB

- GNU DeBugger
- Supports many languages
    - Including C and C++
- Inspection of program during execution
    - Execution flow
    - Data
- Helps finding errors like *segmentation fault*
- Read the fully-detailed manual: https://sourceware.org/gdb/current/onlinedocs/gdb/

GDB usage

Compilation flags

- Canonical compilation command line:

```
$ gcc [cflags] -o <output> <input>
```

- Optimize for speed (`-O2`)

```
$ gcc -Wall -Werror -O2 -o myprogram main.c
```

- Enable debugging support (`-g`)

```
$ gcc -Wall -Werror -g myprogram main.c
```

  - Not recommended to use debugging along with optimizations
    - No optimization option is equivalent to `-O0`

GDB usage

Makefile digression

- During development, very useful to be able to debug your program
- For production, probably better to disable the debug support and activate all possible optimization support
  - Reduce size of the executable (can easily be by 50%!)
  - Increase performance (can also be by 50%!)

Makefile automation

```
ifeq ($(D),1)
CFLAGS    += -g   # Enable debugging
else
CFLAGS    += -O2  # Enable optimization
endif
```

Building mode

```
$ make D=1    # compile with debug support and no optimization
```

```
$ make        # compile with optimizations (production)
```

- Probably want to use `make clean` when changing building mode

GDB usage

Starting GDB

- Start GDB, specify the program to debug

```
$ gdb
...
(gdb) file myprogram
Reading symbols from myprogram...done.
(gdb)
```

- Or, start GDB with the program to debug as argument

```
$ gdb myprogram
...
Reading symbols from myprogram...done.
(gdb)
```

Running the program

- Without any argument:

```
(gdb) run
```

- With arguments:

```
(gdb) run argv1 argv2...
```

GDB usage

Interactive help

- GDB offers an interactive shell
  - History management
  - Auto-complete (with TAB)

In order to discover what you can do, just ask:

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
...

(gdb) help breakpoints
Making program stop at certain points.
List of commands:
awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified location
...

(gdb) help break
Set breakpoint at specified location.
break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]
...
```

GDB usage

Possible scenarios

1. Program doesn't have bugs:
    - It will run fine until completion

    ```
    $ ./myprogram
    I worked, hurray!
    ```

2. *Best*-case scenario, regarding bugs:
    - *Segmentation fault*

    ```
    $ ./myprogram
    segmentation fault (core dumped) ./myprogram
    ```

3. *Worst*-case scenario:
    - Doesn't crash but wrong result

    ```
    $ ./myprogram
    I work��, ��rray!
    ```

    - Bugs that don't trigger any segmentation fault
    - In this case, you'll probably have to spend more time...

Segmentation faults

Example #1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t foo_len (const char *s)
{
    return strlen(s);
}

int main (int argc, char *argv[])
{
    char *a = NULL;

    printf ("size of a = %d\n", foo_len(a));

    return 0;
}
```

Execution

```
$ ./strlen-test
zsh: segmentation fault (core dumped)  ./strlen-test
```

Segmentation faults

Run with GDB

- *(After compiling the code with `-g`)*

```
$ gdb ./strlen-test
(gdb) run
Starting program: /home/joel/tmp/test/strlen-test

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7abc446 in strlen () from /usr/lib/libc.so.6
(gdb)
```

Backtrace

- First thing to do when getting a *segfault*:
  - Understand what is the sequence of calls that brought us there

```
(gdb) backtrace     # use just 'bt'
#0  0x00007ffff7abc446 in strlen () from /usr/lib/libc.so.6
#1  0x000000000040055e in foo_len (s=0x0) at strlen-test.c:7
#2  0x0000000000400583 in main (argc=1, argv=0x7fffffffd788) at strlen-test.c:14
```

Investigate

- `foo_len()` is supposed to receive a pointer
- Here it receives `0` (aka `NULL`)
- Looks like this `NULL` pointer probably gets dereferenced in `strlen()`...

Segmentation faults

Fix...

- Here, the problem is fairly obvious

```
size_t foo_len (const char *s)
{
    return strlen(s);
}

int main (int argc, char *argv[])
{
     char *a = "This is a valid string";

    printf ("size of a = %d\n", foo_len(a));

    return 0;
}
```

And, celebrate!

```
$ ./strlen-test
size of a = 22
```

Segmentation faults

Better fix

- Prevent the same bug from happening again

```
size_t foo_len (const char *s)
{
    assert(s && "String cannot be NULL here!");
    return strlen(s);
}

int main (int argc, char *argv[])
{
    char *a = NULL;

    printf ("size of a = %d\n", foo_len(a));

    return 0;
}
```

```
$ ./strlen-test
strlen-test: strlen-test.c:8: foo_len:
    Assertion `s && "String cannot be NULL here!"' failed.
```

Segmentation faults

Example #2

```
#include<stdio.h>
#include<stdlib.h>

const static int len = 10;

int main(void)
{
    int *tab;
    unsigned int i;

    tab = malloc(len * sizeof(int));

    for (i = len - 1; i >= 0; i--)
        tab[i] = i;

    free(tab);
    return 0;
}
```

Execution

```
$ ./tablen-test
segmentation fault (core dumped)  ./tablen-test
```

Segmentation faults

Run GDB

```
$ gdb ./tablen-test
(gdb) run
Starting program: /home/joel/tmp/test/tablen-test

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400535 in main () at tablen-test.c:14
14              tab[i] = i;
```

Backtrace

```
(gdb) bt
#0  0x0000000000400535 in main () at tablen-test.c:14
```

- Except that here, it's not much of help...

Inspect variables

- Display index `i` so that we know which index in the array was being accessed:

  ```
  (gdb) print i
  $1 = 4294967295
  ```

Segmentation faults

Fix...

- Problem is a case of overflow
  - An `unsigned int` type automatically wraps from 0 to 4294967295

```
#include<stdio.h>
#include<stdlib.h>

const static int len = 10;

int main(void)
{
    int *tab;
    int i;

    tab = malloc(len * sizeof(int));

    for (i = len - 1; i >= 0; i--)
        tab[i] = i;

    free(tab);
    return 0;
}
```

Tracking bugs

Behavior bugs

- Behavioral bugs more complicated to find because program doesn't crash
- It's just that the output is wrong

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    int i;
    char str[] = "Tracking bugs is my passion";

    printf("Before: %s\n", str);

    for (i = 0; i < strlen(str) - 1; i++)
        str[i] = toupper(str[i]);

    printf("After: %s\n", str);

    return 0;
}
```

Execution

```
Before: Tracking bugs is my passion
After: TRACKING BUGS IS MY PASSIOn
```

Tracking bugs

Setting breakpoints

- Stop the program during the execution at a designated point
- Set as many breakpoints as necessary
- GDB will always stop the execution when reaching them

Breaking at exact location in code

```
(gdb) break string-test.c:13
Breakpoint 1 at 0x4005ef: file string-test.c, line 13.

(gdb) r
Starting program: /home/joel/tmp/test/string-test
Before: Tracking bugs is my passion

Breakpoint 1, main () at string-test.c:13
13              str[i] = toupper(str[i]);
```

Tracking bugs

Breaking at a particular function

```
(gdb) b main
Breakpoint 1 at 0x40059f: file string-test.c, line 8.

(gdb) r
Starting program: /home/joel/tmp/test/string-test

Breakpoint 1, main () at string-test.c:8
8           char str[] = "Tracking bugs is my passion";
```

Breaking only if condition is satisfied

```
(gdb) b string-test.c:13 if i == 5
Breakpoint 1 at 0x4005ef: file string-test.c, line 13.

(gdb) r
Starting program: /home/joel/tmp/test/string-test
Before: Tracking bugs is my passion

Breakpoint 1, main () at string-test.c:13
13              str[i] = toupper(str[i]);

(gdb) print i
$1 = 5
```

Tracking bugs

Dealing with breakpoints

- Set at least one breakpoint before running the program
  - Otherwise the program will run until completion
- Once the program stops and the gdb shell is available, a few options:
  1. Continue the execution until hitting the same or another breakpoint

     ```
     (gdb) continue       # or just 'c'
     ```

  2. Execute only the next line of code and break again

     ```
     (gdb) step           # or just 's'
     ```

     Careful, `step` enters function calls
  3. Jump over function calls

     ```
     (gdb) next           # or just 'n'
     ```

Tip: typing `<enter>` in the interactive GDB shell repeats the last command

Tracking bugs

Printing variables

```
int a = 2;
char b = 'x';
int *c = &a;
char *s = "A string";
                        // <= breaking here
...
```

- Inspect the value of all your variables with command `print`

Default

- By default, prints variables according to their type

```
(gdb) print a
$1 = 2
(gdb) p b
$2 = 120 'x'
(gdb) p c
$3 = (int *) 0x7fffffffd65c
(gdb) p s
$4 = 0x40070b "A string"
```

Tweak

- Can tweak both the way `print` prints and what it prints

```
(gdb) print /x a
$1 = 0x2
(gdb) p /c b+2
$2 = 122 'z'
(gdb) p *c
$3 = 2
(gdb) p s[0]
$4 = 65 'A'
```

Tracking bugs

Printing data structures

```
struct entry {
    int     key;
    char    *name;
} obj = {
    .key    = 2,
    .name   = "toto",
};

struct entry *e = &obj;
```

- With `print`, you can access the pointer and the object it's pointing to:

```
(gdb) print e
$1 = (struct entry *) 0x7fffffffd640
(gdb) print &obj
$2 = (struct entry *) 0x7fffffffd640
(gdb) p *e
$3 = {key = 2, name = 0x400734 "toto"}
(gdb) p e->key
$4 = 2
(gdb) p obj.name
$5 = 0x400734 "toto"
```

Misc

Setting watchpoint

- Breakpoints are for interrupting the execution flow at a specific location
- Watchpoints are for interrupting the program when a variable is modified

```
(gdb) watch i
Hardware watchpoint 2: i

(gdb) c
Continuing.
Before: Tracking bugs is my passion

Hardware watchpoint 2: i

Old value = 0
New value = 1
0x0000000000400612 in main () at string-test.c:12
12          for (i = 0; i < strlen(str) - 1; i++)
```

Misc

Other useful commands

- `finish`
  - Runs until the current function is finished
- `until`
  - When executed in a loop, continues the execution until the loop ends
- `info breakpoints`
  - Shows informations about all declared breakpoints

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040059f in main at string-test.c:
8
breakpoint already hit 1 time
2       hw watchpoint  keep y                      i
breakpoint already hit 2 times
```

- `delete`
  - Deletes a breakpoint

Valgrind

Example

```c
#include <stdlib.h>

void f(void)
{
        int *x = malloc(10 * sizeof(int));
        x[10] = 0;
}

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

Valgrind

Run

```
$ valgrind --leak-check=full ./valgrind_example
...
==31134== Invalid write of size 4
==31134==    at 0x108668: f (in /home/joel/work/ecs150/slides/tuto_gdb/code/valgrind_example)
==31134==    by 0x108679: main (in /home/joel/work/ecs150/slides/tuto_gdb/code/valgrind_example)
==31134==  Address 0x51f0068 is 0 bytes after a block of size 40 alloc'd
==31134==    at 0x4C2CEDF: malloc (vg_replace_malloc.c:299)
==31134==    by 0x10865B: f (in /home/joel/work/ecs150/slides/tuto_gdb/code/valgrind_example)
==31134==    by 0x108679: main (in /home/joel/work/ecs150/slides/tuto_gdb/code/valgrind_example)
==31134==
==31134==
==31134== HEAP SUMMARY:
==31134==     in use at exit: 40 bytes in 1 blocks
==31134==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==31134==
==31134== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31134==    at 0x4C2CEDF: malloc (vg_replace_malloc.c:299)
==31134==    by 0x10865B: f (in /home/joel/work/ecs150/slides/tuto_gdb/code/valgrind_example)
==31134==    by 0x108679: main (in /home/joel/work/ecs150/slides/tuto_gdb/code/valgrind_example)
==31134==
==31134== LEAK SUMMARY:
==31134==    definitely lost: 40 bytes in 1 blocks
==31134==    indirectly lost: 0 bytes in 0 blocks
==31134==      possibly lost: 0 bytes in 0 blocks
==31134==    still reachable: 0 bytes in 0 blocks
==31134==         suppressed: 0 bytes in 0 blocks
==31134==
==31134== For counts of detected and suppressed errors, rerun with: -v
==31134== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

C preprocessor: Macros

Sequential values

Example

```c
#define ST_READY    0
#define ST_RUNNING  1
#define ST_BLOCKED  2

void show_state(struct uthread_tcb *tcb)
{
    printf("Thread %d: ", tcb->tid);
    switch(tcb->state) {
        case ST_READY:
            printf("Ready\n");
        case ST_RUNNING:
            printf("Running\n");
        case ST_BLOCKED:
            printf("Blocked\n");
    }
}
```

After preprocessing

```
$ cpp file.c
...
```

```c
...
void show_state(struct uthread_tcb *tcb)
{
    printf("Thread %d: ", tcb->tid);
    switch(tcb->state) {
        case 0:
            printf("Ready\n");
        case 1:
            printf("Running\n");
        case 2:
            printf("Blocked\n");
    }
}
```

C preprocessor: Macros

Versus `enum`

```
#define ST_READY    0
#define ST_RUNNING  1
#define ST_BLOCKED  2
```

```
enum state {
    ST_READY,
    ST_RUNNING,
    ST_BLOCKED,
};
```

Pros

- Autonumbering
- If used in a switch-case, warn if switch-case isn't complete
- Debugger gets access to names

Cons

- No control on the integer type the compiler chooses
  - *(possible only in C++: `enum state : char { ... };`)*

C preprocessor: Macros

Arbitrary values

Example

```c
#define HZ 100

#define TIMER_REG1 0x0
#define TIMER_REG2 0x4
#define TIMER_REG3 0x8

#define TIMER_REG1_MASK 0x8000FFFF

void configure_timer(int *timer_addr, int val)
{
    *(timer_addr + TIMER_REG1) = val & TIMER_REG1_MASK;
}
```

Pros

- Recommended when names matter more than values
- And values are somewhat arbitrary (e.g., not in strict sequence)
  - Otherwise prefer `enum`

C preprocessor: Macros

Code replacement

Example

```
#define twice(x) 2 * x

void f(int a)
{
    int b = twice(a);
}
```

After preprocessing

```
void f(int a)
{
    int b = 2 * a;
}
```

Pitfall #1

```
int c = twice(a + 1) * 3;
```

```
int c = 2 * a + 1 * 3;
```

- What's the value of `c`? Is it the value meant to be assigned?

Solution

- Always surround your arguments with parenthesis!

```
#define twice(x) (2 * (x))
```

```
int c = (2 * (a + 1)) * 3;
```

C preprocessor: Macros

Pitfall #2

```
#define die_perror(x)    \
    perror(x);           \
    exit(1);

void f(void)
{
    int *a;

    a = malloc(10 * sizeof(int));
    if (a == NULL)
        die_perror("malloc");
    ...
}
```

- Any issues?

After preprocessing

```
#define die_perror(x)    \
    perror(x);           \
    exit(1);

void f(void)
{
    int *a;

    a = malloc(10 * sizeof(int));
    if (a == NULL)
        perror("malloc");
        exit(1);
        ;
    ...
}
```

- Misleading, and logically incorrect

C preprocessor: Macros

Pitfall #2 (cont'd)

```c
#define die_perror(x)    \
{                        \
    perror(x);           \
    exit(1);             \
}

void f(void)
{
    int *a;

    a = malloc(10 * sizeof(int));
    if (a == NULL)
        die_perror("malloc");
    ...
}
```

- Does this work better?

```c
#define die_perror(x)    \
{                        \
    perror(x);           \
    exit(1);             \
}

void f(void)
{
    int *a;

    a = malloc(10 * sizeof(int));
    if (a == NULL)
        die_perror("malloc")
    ...
}
```

- Works but have to purposefully omit the semi-colon which is counter-intuitive

C preprocessor: Macros

Pitfall #2 (cont'd)

Solution

```
#define die_perror(x)   \
do {                    \
    perror(x);          \
    exit(1);            \
} while(0)

void f(void)
{
    int *a;

    a = malloc(10 * sizeof(int));
    if (a == NULL)
        die_perror("malloc");
    ...
}
```

- Surround your multiple statement code with `do { ... } while(0)`

C preprocessor: Macros

Pitfall #3

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))

void f(int a, int b)
{
    int c = MAX(a, b);
}
```

```
int d = MAX(++a, b);
```

```
int d = ((++a) > (b) ? (++a) : (b))
```

- What's the value of `d`? Is it the value that was meant to be assigned?

Solution

```
#define MAX(x,y)                    \
({                                  \
    typeof(x) _x = (x);             \
    typeof(y) _y = (y);             \
    (_x) > (_y) ? (_x) : (_y);      \
})
```

- `({` to start a compound statement within an expression
- `typeof(variable)` replaced by type of `variable`

C preprocessor: Macros

Return value

```c
void f(void)
{
    int b = ({int c = 2; printf("lol\n"); c;});
    printf("%d\n", b);
}
```

- Last statement of expression is expression's result

Macro example

```c
#define xmalloc(size)        \
({                           \
    void *x = malloc(size); \
    if (!x) {                \
        perror("malloc");    \
        exit(1);             \
    }                        \
    x;                       \
})

void f(void)
{
    int *a = xmalloc(10 * sizeof(int));
    ...
}
```

- Better to have `xmalloc` declared as a `static inline` (see next slide)

C preprocessor: Macros

Macro vs `static inline`

`static inline`

```
static inline void* xmalloc(size_t size)
{
    void *x = malloc(size);
    if (!x) {
        perror("malloc");
        exit(1);
    }
    return x;
}
```

Pros

- Type checking
- Actual function

Cons

- Inlining is purely advisory (compiler is free to ignore it)

Macro

```
#define WARN_IF(exp)                      \
do {                                      \
  if (exp)                                \
    fprintf(stderr,                       \
            "Warning: " #exp "\n");       \
} while(0)

...
WARN_IF(myptr == NULL);
```

Pros

- No type checking (generic)
- Possibility to stringify arguments

Cons

- No type checking (or more difficult to implement)
- Difficult to debug
- Harder to edit when multi-line

Multi-talented `printf()`

101

```
    printf("Hello world!\n");
```

printf_examples.c

```
$ ./a.out
Hello world!
```

201

```
    printf("%s from %c to Z, in %d minutes!\n", "printf", 'A', 45);
```

printf_examples.c

```
$ ./a.out
printf from A to Z, in 45 minutes!
```

pow(101, 2)

```
    int i;
    printf("\b%n", &i);
```

printf_examples.c

```
    printf("%s\bD is \033[1;31m#%d\033[0m!\n", "UCB", i);
```

printf_examples.c

```
$ ./a.out
 UCD is #1!
```

`printf()`: an odyssey

Fortran I

Special statement for building formatting descriptions:

```
    WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA

601 FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
   +        13H SQUARE UNITS)
```

- (Approximate) translation in C:

```
printf(" A= %5d  B= %5d  C= %5d  AREA= %10.2f SQUARE UNITS", a, b, c, area);
```

BCPL

Printing and formatting are merged into a single statement:

```
WRITEF("%I2-QUEENS PROBLEM HAS %I5 SOLUTIONS*N", NUMQUEENS, COUNT)
```

- (Approximate) translation in C:

```
printf("%2d-queens problem has %5d solutions\n", numqueens, count);
```

`printf()`: an odyssey

C

```
printf("Hello %s, you are %d years old\n", name, age);
```

Trickle-down string formatting

Unix `printf`

```
$ printf "%s, stop lying; you're not %d\!\n" Bob 21
Bob, stop lying; you're not 21!
```

Other languages...

awk, C++, Objective C, D, F#, G (LabVIEW), GNU MathProg, GNU Octave, Go, Haskell, J, Java (since version 1.5) and JVM languages (Clojure, Scala), Lua (string.format), Maple, MATLAB, Max (via the sprintf object), Mythryl, PARI/GP, Perl, PHP, Python (via % operator), R, Red/System, Ruby, Tcl (via format command), Transact-SQL (via xp_sprintf), Vala.

Why `printf()`?

Output

```
$ cowsay "I love lectures about printf\!"
 _______________________________
< I love lectures about printf! >
 -------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
$ cowthink -f vader "printf, I'm your father"
 _________________________
( printf, I'm your father )
 -------------------------
        o    ,-^-.
         o   !oYo!
          o /./=\.\______
               ##        )\/\
                ||-----w||
                ||      ||

               Cowth Vader
```

```
$ du /bin/* | sort -rn
20672   /bin/js52
19164   /bin/inkscape
19136   /bin/inkview
18720   /bin/node
18684   /bin/clementine
17452   /bin/mariabackup
17112   /bin/mysqld
16432   /bin/mysql_client_test_embedded
16332   /bin/mysql_embedded
16232   /bin/mysqltest_embedded
7600    /bin/gdb
...
```

Introspection

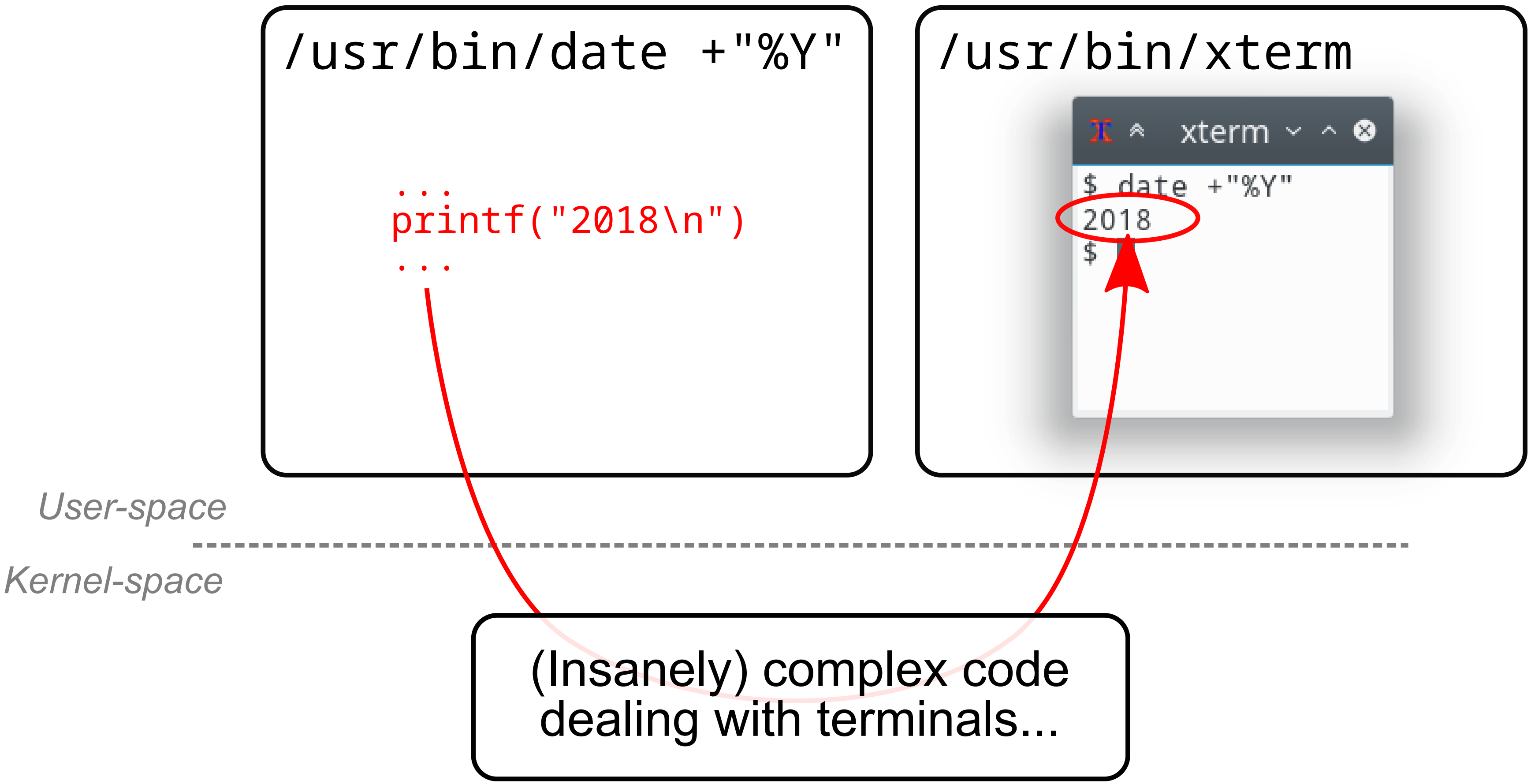
```
Booting kernel from Legacy Image at 20080000 ...
    Image Name:   Linux-2.6.37
    Image Type:   ARM Linux Kernel Image (uncompressed)
    Data Size:    1256880 Bytes = 1.2 MiB
    Load Address: 20008000
    Entry Point:  20008000
    Verifying Checksum ... OK
    Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Linux version 2.6.37 (nkinar at matilda) (gcc version 4.3.5 (Buildro
ot 2011.02) ) #3 Sat Apr 2 17:28:21 CST 2011
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00053177
CPU: VIVT data cache, VIVT instruction cache
Machine: Atmel AT91SAM9RL-EK
Memory policy: ECC disabled, Data cache writeback
Clocks: CPU 200 MHz, master 100 MHz, main 12.000 MHz
Built 1 zonelists in Zone order, mobility grouping on.
Total pages: 16256
Kernel command line: console=ttyS0,115200
mtdparts=flash:10M(kernel),100M(root),-(storage) rw rootfstype=ubifs
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 64MB = 64MB total
Memory: 62348k/62348k available, 3188k reserved, 0K highmem
...
```

Tell me where you `printf()`!

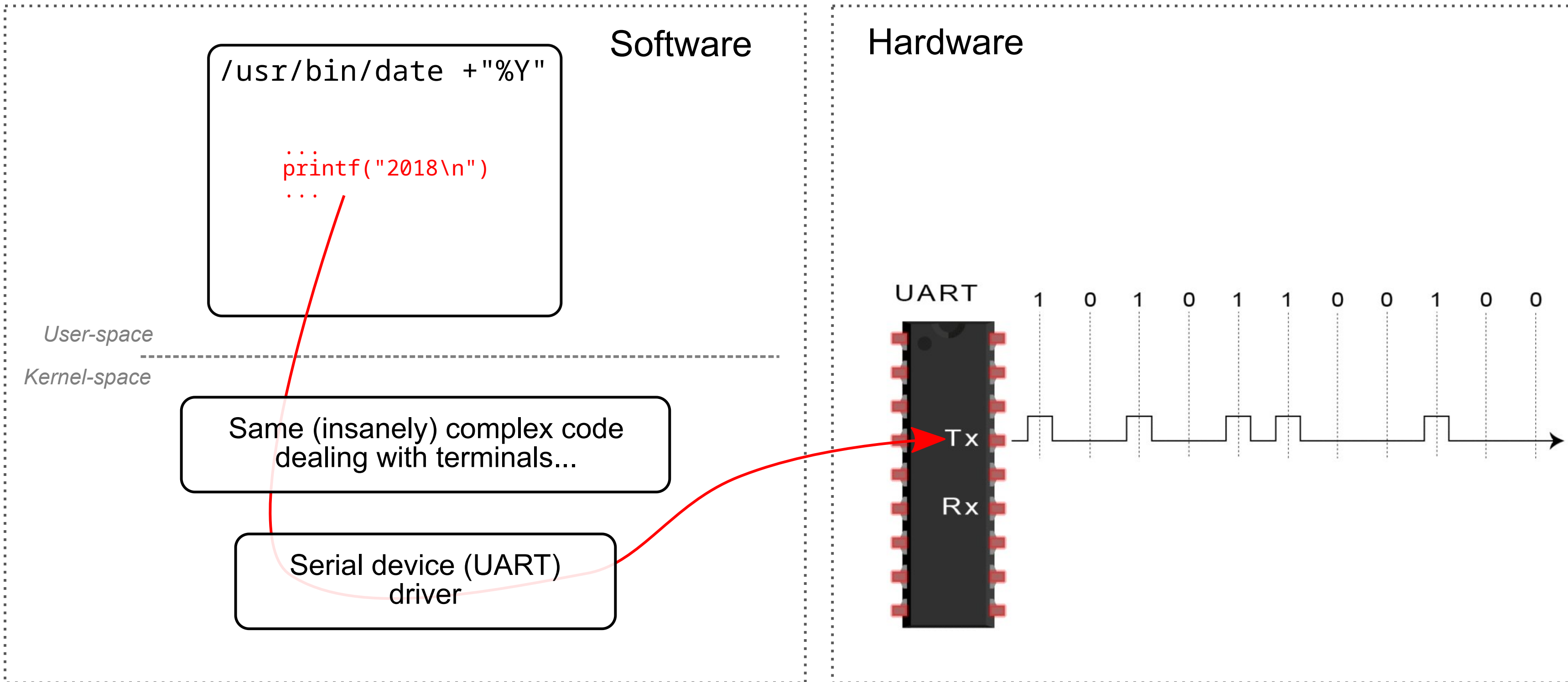
Typical GNU/Linux computer



Ingo Molnár (Linux kernel core developer): "The tty layer is one of the very few pieces of kernel code that scares the hell out of me :-)"

Tell me where you `printf()`!

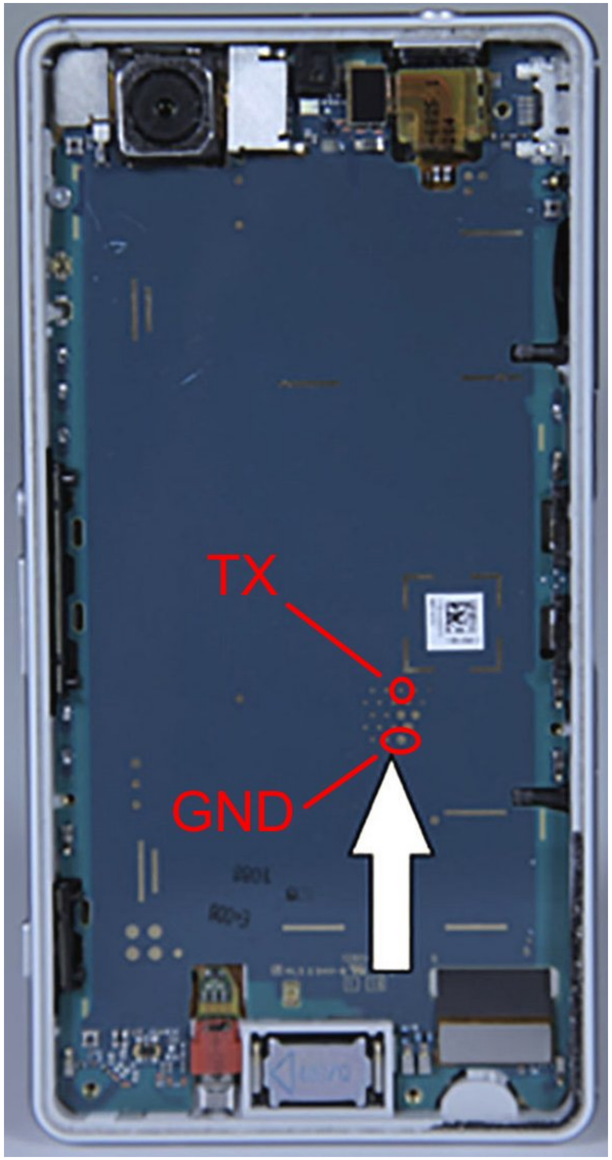
Embedded systems or when display is not available



- Characters are sent one by one over a serial port

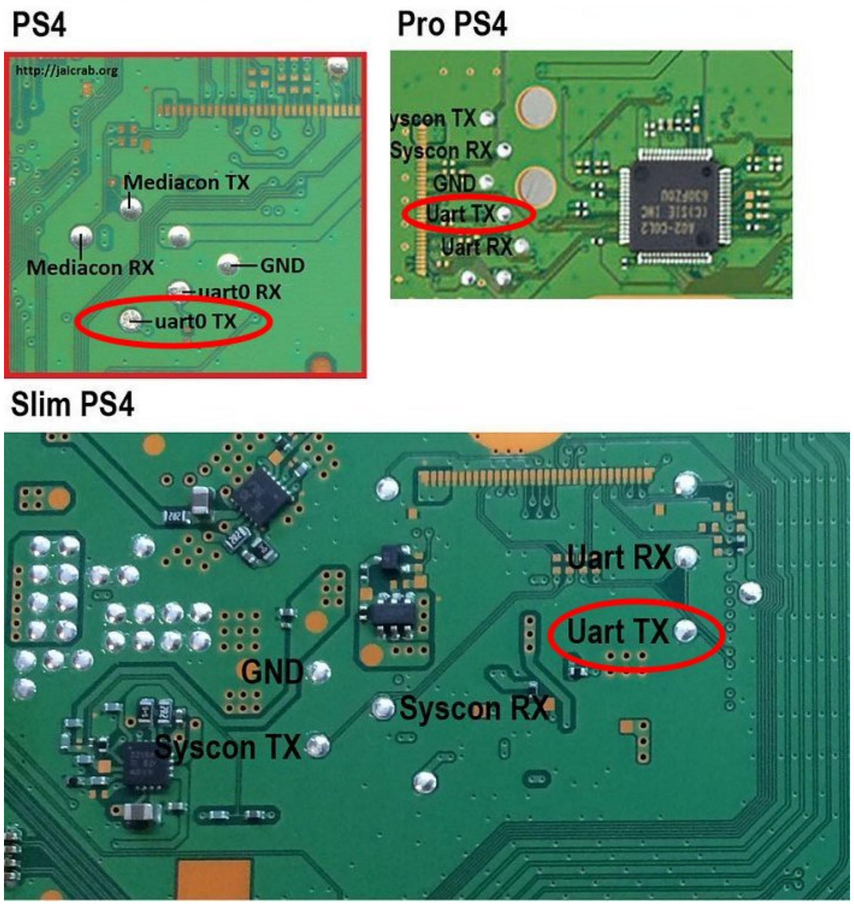
A serial port on every board?

My own phone!

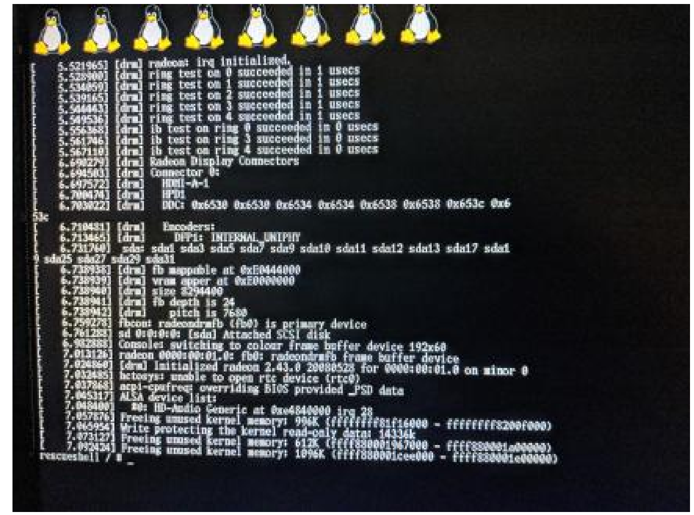


A serial port on every board?

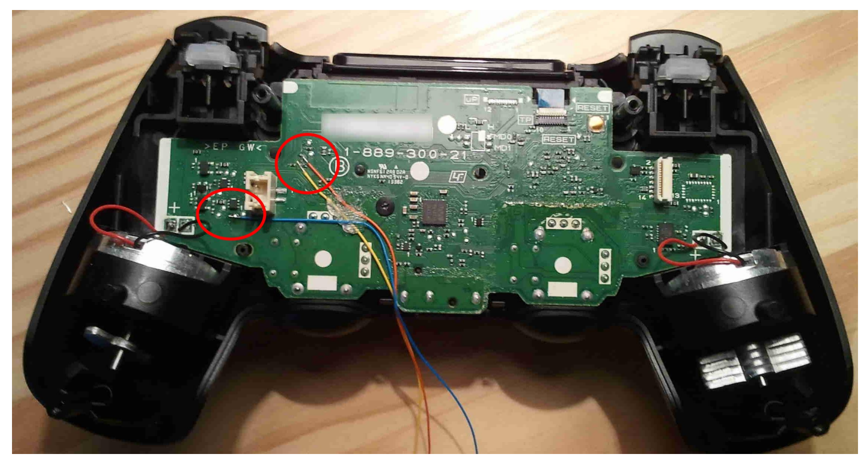
Playstation 4



Linux on PS4



PS4 controller



How to print?

`putchar()`: the cornerstone

```
/*
 * Code for IBM-PC x86: write character to COM1 serial port
 */
void putchar(char ch)
{
  /* Wait until the Transmitter Holding Register (THR) is empty. */
  while ((inb(COM1_PORT+COM_LSR) & LSR_THRE) == 0);

  /* Then output the character to the THR */
  outb(ch, COM1_PORT+COM_THR);
}
```

*Copied from NuttX RTOS

How to print?

No formatting is simply `puts()`

```c
void my_printf(char *fmt)
{
    while (*fmt)
        putchar(*fmt++);
}

int main(void)
{
    /* printf 101 */
    my_printf("Hello world!\n");
}
```

printf_puts.c

```
$ ./a.out
Hello world!
```

How to print?

With formatting...

Input:

```
my_printf("%s from %c to Z, in %d minutes!\n", "printf", 'A', 45);
```

Output:

```
printf from A to Z, in 45 minutes!\n
```

- For each placeholder, need to retrieve next parameter
- Depending on placeholder:
  - '%s': get string
  - '%c': get character
  - '%d': **convert** integer into characters

Variadic functions

Example: function prototype

```c
#include <stdarg.h>    /* Macro/function definitions for variadic functions */
#include <stdio.h>

/*
 * Sum a variable number of integers
 * @count: the number of integer parameters
 *
 * Receive @count integers, sum them up and return the result
 */
int sum_ints(int count, ...)
{
...
}

int main(void)
{
    int res;

    /* sum up 3 integers: 10, 20 and 30 */
    printf("Sum is %d\n", sum_ints(3, 10, 20, 30));

    /* sum up 5 integers: 10, 20, 30, 40, 50 */
    printf("Sum is %d\n", sum_ints(5, 10, 20, 30, 40, 50));

    return 0;
}
```

variadic_fcn.c

Variadic functions

Example: function implementation

```
#include <stdarg.h>    /* Macro/function definitions for variadic functions */
...
int sum_ints(int count, ...)
{
    va_list ap;
    int i, sum = 0;

    va_start(ap, count);                /* Init variable parameter list */

    for (i = 0; i < count; i++) {
        int n = va_arg(ap, int);        /* Get next integer parameter */
        sum += n;
    }

    va_end(ap);                            /* Clean up parameter list */

    return sum;
}
```

variadic_fcn.c

`printf()`: a simple implementation

Demo time: let's code!

`printf()`: a simple implementation

Before

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

void my_printf(char *fmt, ...)
{
    /* ??? */
}

int main(void)
{
    /* printf 101 */
    my_printf("Hello world!\n");

    /* printf 201: basic placeholders */
    my_printf("%s from %c to Z, in %d minutes!\n",
              "printf", 'A', 0);

    return 0;
}
```

printf_demo_start.c

printf(): a simple implementation

After

```c
void my_printf(char *fmt, ...)
{
    va_list ap;
    char c;
    int d;
    char *s;

    va_start(ap, fmt);

    while (*fmt) {
        /* Check if character is '%' */
        if (*fmt != '%') {
            /* If not, display without no processing and
             * continue to the next character
             */
            putchar(*fmt++);
            continue;
        }

        /* Skip '%' and get to the placeholder */
        fmt++;
        /* Distinguish different placeholders */
        switch(*fmt) {
            case 'c':
                /* Get character from arguments */
                c = va_arg(ap, int);
                putchar(c);
                break;
            case 's':
                /* Get string pointer from arguments */
                s = va_arg(ap, char*);
                /* Display each char from the string */
                while (*s) putchar(*s++);
                break;
...
```

printf_demo_end.c

```c
            case 'd':
                /* Get integer from arguments */
                d = va_arg(ap, int);
                /* Translate integer into string
                 * (via ASCII conversion) */
                char buf[10], *end = buf;
                do {
                    *end = (d % 10) + '0';
                    end++;
                    d /= 10;
                } while (d);
                /* Display string */
                while (end != buf) {
                    putchar(*--end);
                }
                break;
        }
        /* Skip placeholder and continue */
        fmt++;
    }

    va_end(ap);
}

int main(void)
{
    /* printf 101 */
    my_printf("Hello world!\n");

    /* printf 201: basic placeholders */
    my_printf("%s from %c to Z, in %d minutes!\n",
              "printf", 'A', 45);

    return 0;
}
```

printf_demo_end.c

```
printf("Thanks!");
```