the murgiahack experiment

Gianluca Guida
<glguida@gmail.com>

Bruxelles, 30 January 2016

mh / introduction

MH is a microkernel and a set of libraries

early stage

BSD license

~~*portable but not ported*~~

Original (and still valid) goals:

management of privileged system resources

basic HW platform management

base for experiments with hardware and software

mh / base for experiments

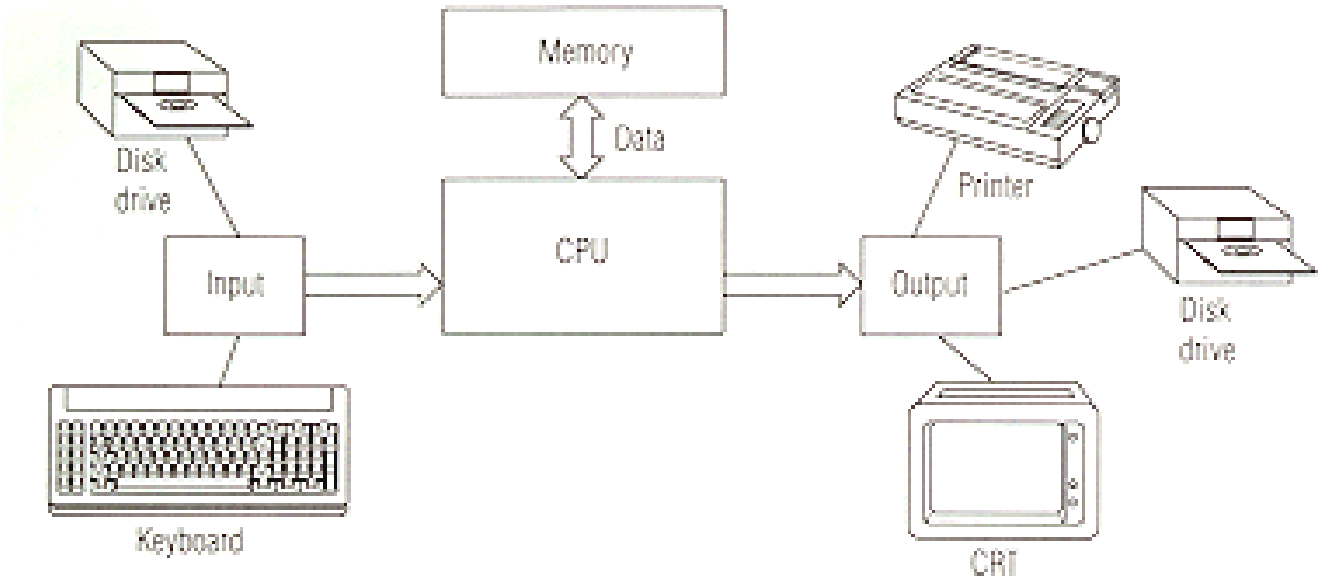
A system for quick experimenting should be easy to tinker and think with.

Should be:

- **Flexible.** *Entities in the system should be able to be used as building blocks to build complex systems – e.g., pipes in UNIX*
- **Sensible and clear entities.** *Simple objects with simple interactions help to reason with the system – e.g., pipes vs grant tables*
- **Practical.** *The system should not be suffocated by excess of ideology in the actual implementation – we* are *engineers.*

mh / microcomputer I

Meet the microcomputer:



mh / microcomputer II

- Its **flexibility** has allowed the microcomputer to *evolve* into really complex current hardware.

- When you do *not* look at the details, the abstractions are **clear and sensible**:

  CPU, bus, memory, I/O device

Its ease to experiment with it is probably the reason why many of us in this room – the *oldest* of us? – started programming.

mh / the questions

Could we use these abstractions to gain software flexibility while keeping the system practical and clear?

Would such a system be actually useful?

mh / not new I

OS kernels whose interface is similar and closer to the hardware aren't new.

- *exokernel*s do it to safely multiplex the hardware (no abstraction paradigm)
- *hypervisors* expose either an almost-identical or identical to hardware interface to provide virtual machines (paravirtualization or full virtualization)

Both of these interfaces are inherently non-portable.

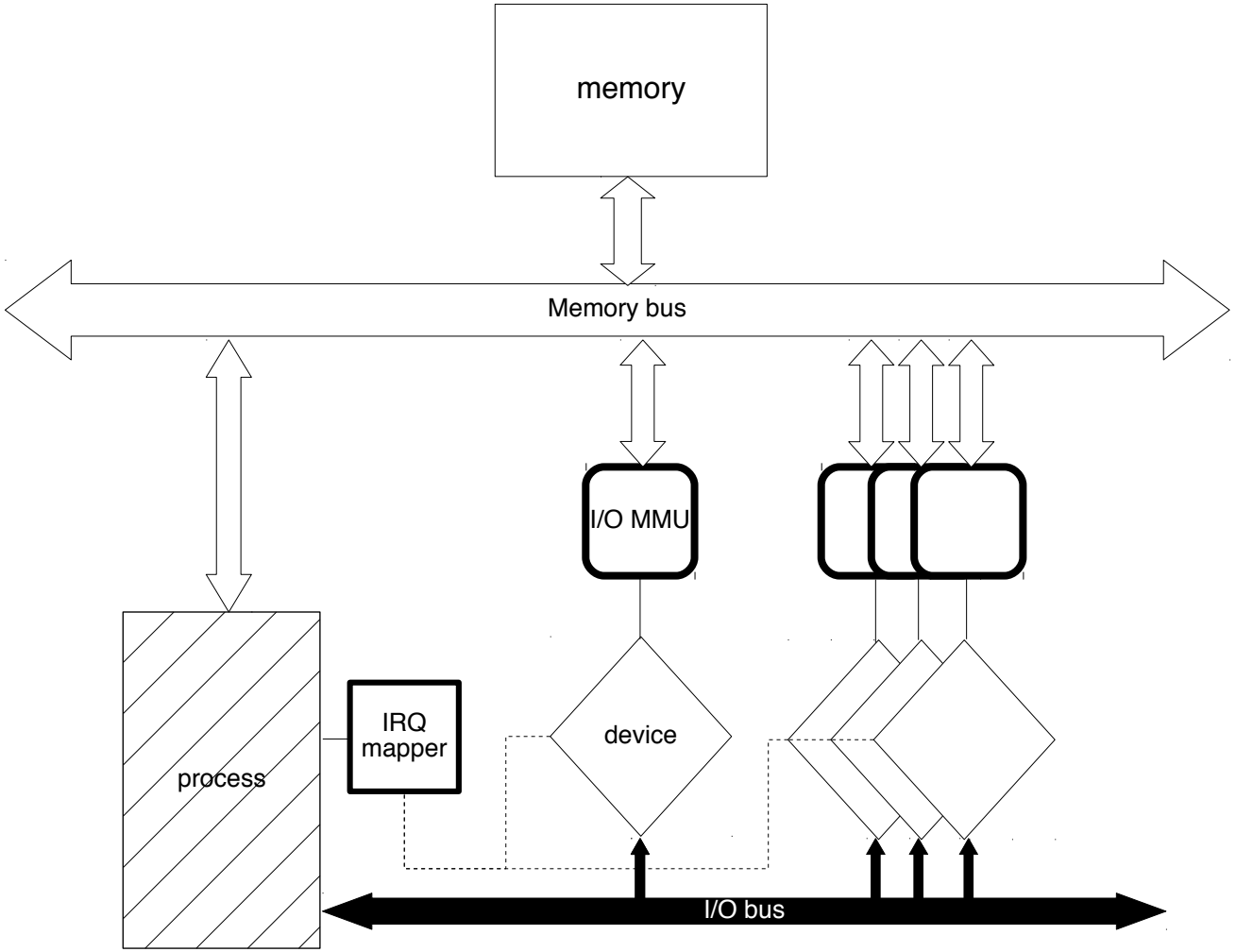
- *exokernels* do not want to lie to the user
- *hypervisors* are built to avoiding emulation and to use native architecture as much as possible for efficiency and performance

mh / not new II

- *Another* operating system whose interface is inspired by hardware: early UNIX
  - UNIX in essence exposes a terminal, a disk and interrupts (*signals*) to every process (which are CPUs)
  - There are other concepts to this hardware-centric view: *fork(), pids, uids, gids.* These help create a collaborative system where resource can be protected and shared at the same time

What happens if we take this interpretation of the UNIX approach and we lower the exposed hardware to the level of CPU, bus, I/O device?

mh / the process model I



mh / the process model II

Each process has **memory** and a **bus.**

Memory:

- note the absence of the MMU. It is *physical* memory and cannot be mapped twice in the process address space
- The allocation is negotiated with the microkernel: syscall to map a new page in the address space
- Pagefault exceptions delivered to the process, making it possible to allocate memory at run time

mh / the process model III

Process bus:

- A process *plugs* a device selected from the devices registered to the system. *open()*
- A process can *unplug* a device at any time. *close()*
- A process may map a device IRQ to a process interrupt. i*rqmap()*
- A process can give access to physical memory to a device by programming a per-device IO-MMU that creates a shared I/O memory address space. *export(dev, va, ioaddr)*
- A process can send and receive data in the I/O bus space to the device's I/O port. *in(), out()*

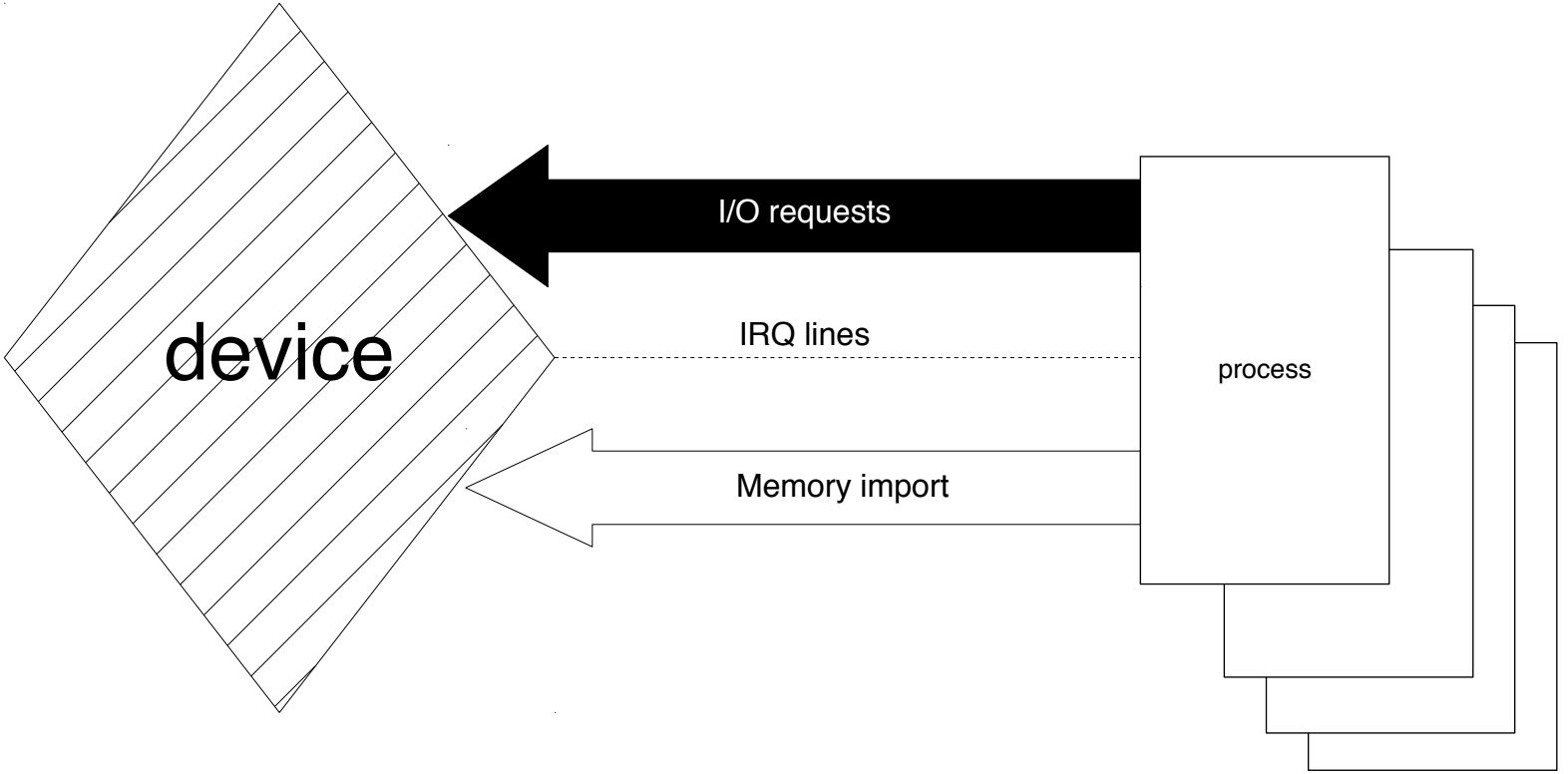
Remarkably, no MMIO.

mh / the process model IV

Process control

- Interrupt and Exceptions:
  - Exceptions and interrupts can be delivered to a specific IP / stack pointer, specified by the process.
  - Special syscall, *sys_iret(),* to restore back to the IP/Stack pointer specified in the stack.
  - A process can enable and disable interrupt delivery, *sys_cli()*, *sys_sti()*
  - A process can enter a halted state and be awaken by the next interrupt. *sys_wait()*

mh / the device model I



mh / the device model II

Every process can be a device

- A process *registers* itself as a device by calling *creat()*, specifying a name ID (64 bit), a vendor ID and a device ID
  - Using DEC RAD50 we can pack 12 case insensitive characters in a 64 bit ID
- When a device process dies the device is *unregistered* on all the buses where it is plugged.
- A device process receive *I/O notifications* from clients through an interrupt specified in *creat()*

mh / the device model III

Device operations:

- *Poll* from the request queues from the device. *poll()*
- *Raise* process specific IRQ line. *irq(id, irq)*
- Access the process I/O memory address space, by *importing* process memory from the I/O MMU. *import(id, ioaddr, va)*
- Signaling the end of I/O processing to the process, which will raise the special EIOIRQ (#0) to the process. *eio(id)*

mh / actual examples I

- *Too* simple device loop:

```
cfg.nameid = 500;
cfg.vendorid = 0xf00ffa;
cfg.deviceid = 1;
sys_creat(&cfg, interrupt_number, mode);

while (1) {
    unsigned id;
    struct sys_poll_ior ior;

    sys_wait();
    id = sys_poll(&ior);
    printf("I/O port %x,  val %x\n", ior.port, ior.val);
    sys_irq(id, 3);
    sys_eio(id);
}
```

mh / actual examples II

- Process code:

```
size_t exported_mem_size = 1024;
struct sys_creat_cfg cfg;
int desc, ret;
void *p;

p = drex_mmap(NULL, exported_mem_size,
                  PROT_READ|PROT_WRITE, MAP_ANON, -1, 0);
desc = sys_open(500);
sys_mapirq(desc, 0, 5);
sys_export(desc, p, 0);
sys_readcfg(desc, &cfg);
printf("cfg: %llx %lx %lx\n",
       cfg.nameid, cfg.vendorid, cfg.deviceid);
sys_out(desc, 10, 255);
sys_wait();
printf("IRQ received!\n");
```

mh / all the rest I

We have seen how we lowered the abstraction interfaces of UNIX in the murgiahack microkernel. What about the rest?

The approach is to keep things as close as possible to UNIX.

- *Fork()*
  - The *fork()* syscall creates a readonly copy of the process, the copying is made and mapped inside the process exception handler. Memory is still reference counted in the microkernel, so last process to write gets the original copy.
  - Child does not inherit devices in the parent bus. But they can be plugged in the device the same way in UNIX-like systems we modify descriptors of a child after a fork().

mh / all the rest II

- Resource ownership and permission checks
  - Unlike many microkernels, we keep the original UNIX scheme of UID/GID.
  - Each process inherit UID/GID from parents and follow the same semantics of a POSIX system (effective, saved-set-uid)
  - Each device is created with a *devmode_t* flag, which exactly like a *mode_t* in a UNIX creat() tells the kernel who can plug the device to its bus.

mh / the exec problem

- *exec()* can be *easily* implemented in a library
  - loading elf and mapping memory it's *easy!*

but:

- *exec()* is much more than that in UNIX
  - the SETUID/SETGID mechanism.

setuid is a trusted mechanism to *increase* permission, which is a privileged resource. Kernel must be aware.

setuid associated to a file, a concept we don't have and we don't want to have.

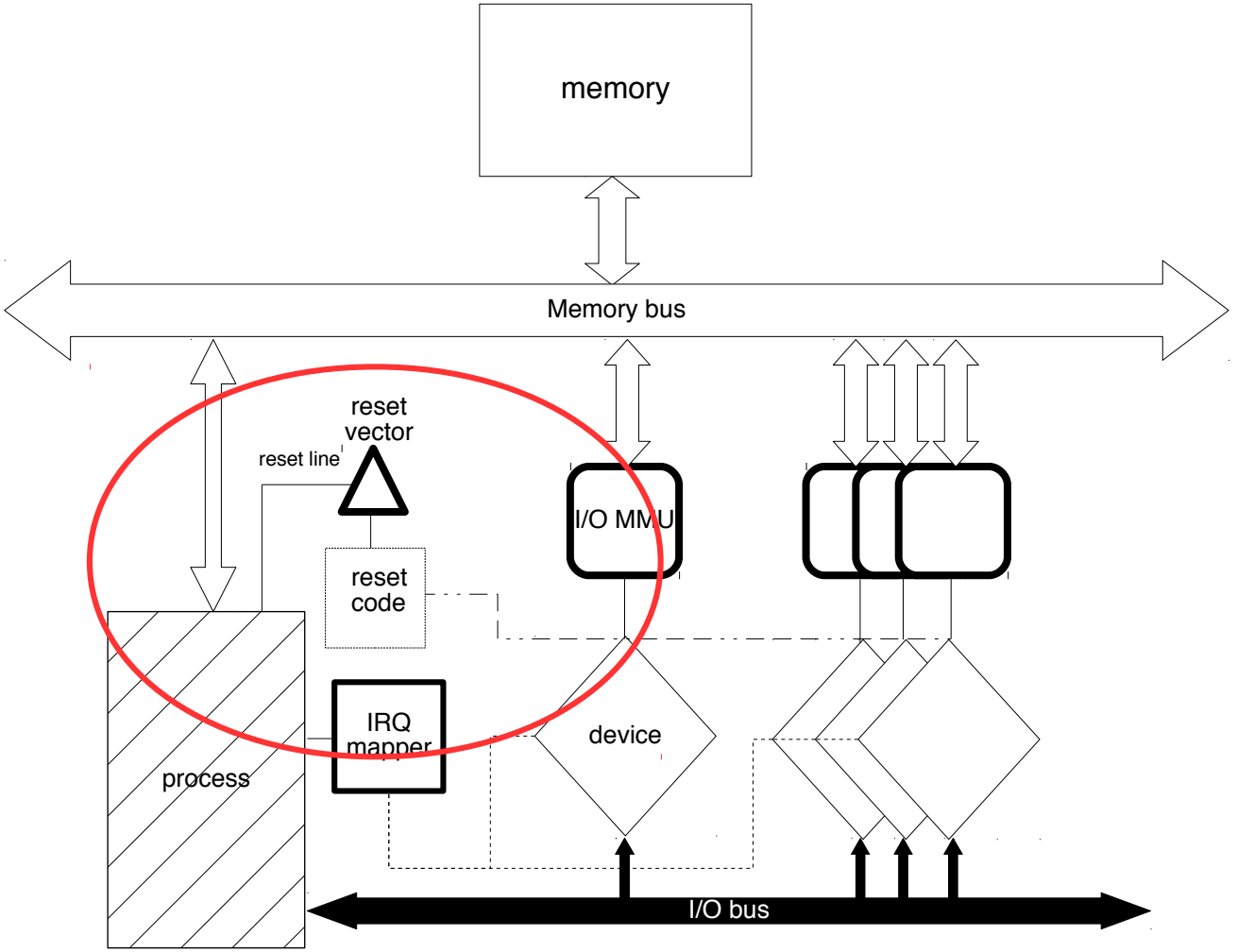
mh / the exec solution!

What is *exec()?*

- A way to execute foreign and protected code
  - “Hey, that's BIOS code!”

- Destroys the current state of the process, while keeping external resources almost intact (CLOEXEC apart)
  - “Hey, that's a CPU RESET!”

mh / RESET I


memory
Memory bus
reset
vector
reset line
reset
code
I/O MMU
IRQ
mapper
process
device
I/O bus

mh / RESET II

- Process can *give permission* to a device to write to its RESET vector.
- Device *writes* to the process RESET vector both memory and associated UID/GID.

- Anytime it desires, a process can call *reset()* to execute code in the reset vector at the UID/GID specified, while honoring the saved-set-uid interface of UNIX.

- A process can of course write to its own reset vector.
  - Interesting way to implement *restartable servers!*

mh / the libraries

- libdrex: a simple libOS that implements basic UNIX commands. [Not there yet]
- libdirtio: Dirt I/O is a protocol almost identical to virtio (of KVM and lguest) that let us have a standard way to discover devices and communicate with them. [Almost there]
  - Only difference: address shared are *I/O MMU* addresses and not *guest* address.
- NetBSD libc and headers in userspace.
- rump kernels support is easy and is very natural to the mh architecture.

mh / last slide

- Exciting project to work on
- The environment it creates is really fun and nice to play with
- Lot of things to do!

Code at http://github.com/glguida/mh

Questions?