Universität Bamberg

Operating Systems Engineering

Lecture 4: OS Interfaces and protection

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

OS Interfaces

Universität Bamberg

- We saw **system calls** as a mechanism to…
  - request services from the OS
  - control access to OS functions
    - check if a process is allowed to execute a given syscall
    - check if the parameters are valid
  - protect the OS from erroneous or malicious accesses by applications
    - read/write kernel memory
    - call kernel functions directly

- Let’s take a detailed look at the **interface** between kernel and application
  - Which functions are usually provided, which type of parameters?

Role of the Operating System

Universität Bamberg

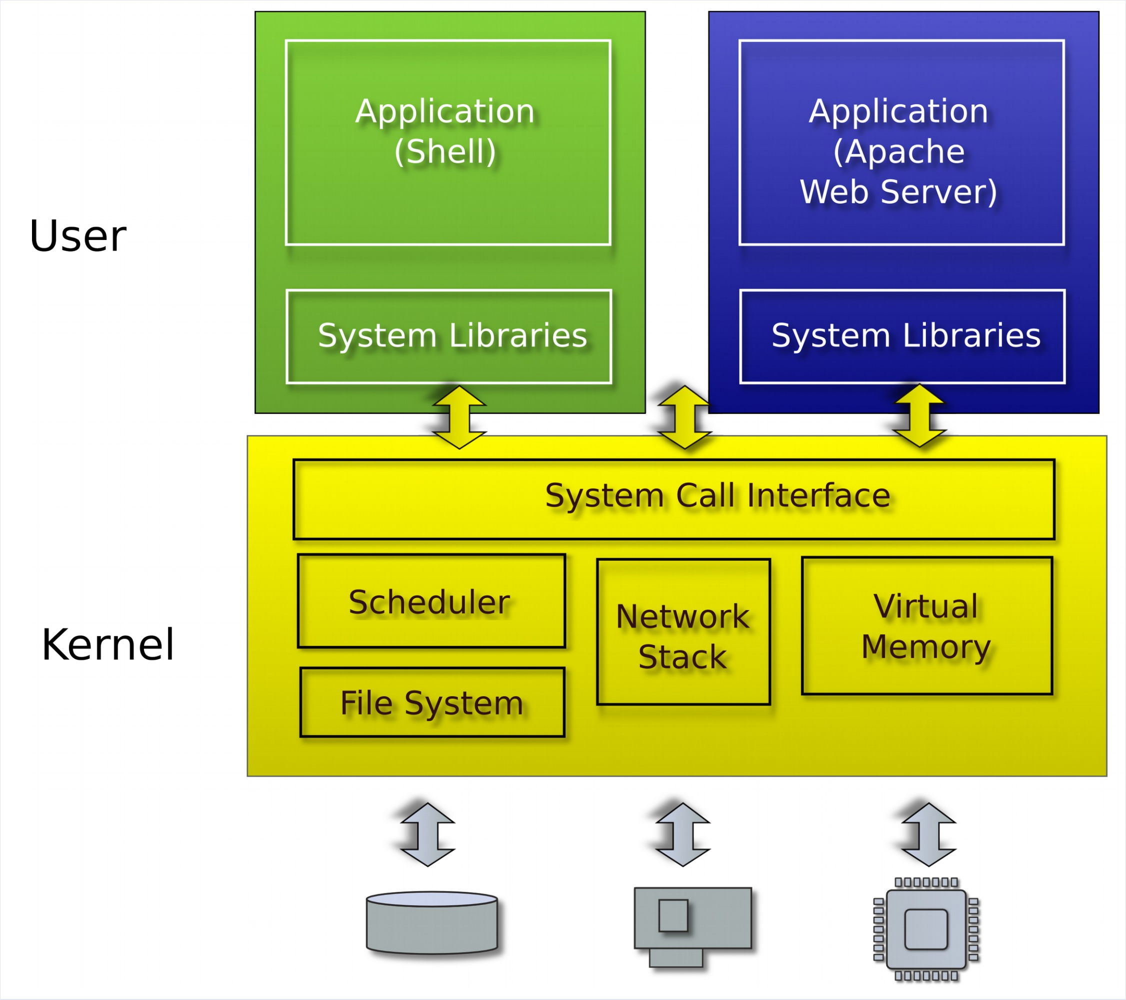
- Share hardware across multiple processes
  - Illusion of private CPU, private memory
- Abstract hardware
  - Hide details of specific hardware devices
- Provide services
  - **Serve as a library for applications** – our approach so far
- Security
  - Isolation of processes
  - Controlled ways to communicate (in a secure manner)

Syscalls in a typical Unix system

Universität Bamberg

System calls…

- provide user to kernel communication
- effectively an invocation of a kernel function
- system calls implement the **interface of the OS**



What system calls do we need?

Universität Bamberg

- **System calls are the abstract interface to the hardware of a computer and other resources**

- **Typical Unix system calls can be grouped into:**
  - **Processes**
    - Creating, exiting, waiting, terminating
  - **Memory**
    - Allocation, deallocation
  - **Files and folders**
    - Opening, reading, writing, closing
  - **Inter-process communication**
    - Pipes, shared memory

Syscalls in xv6

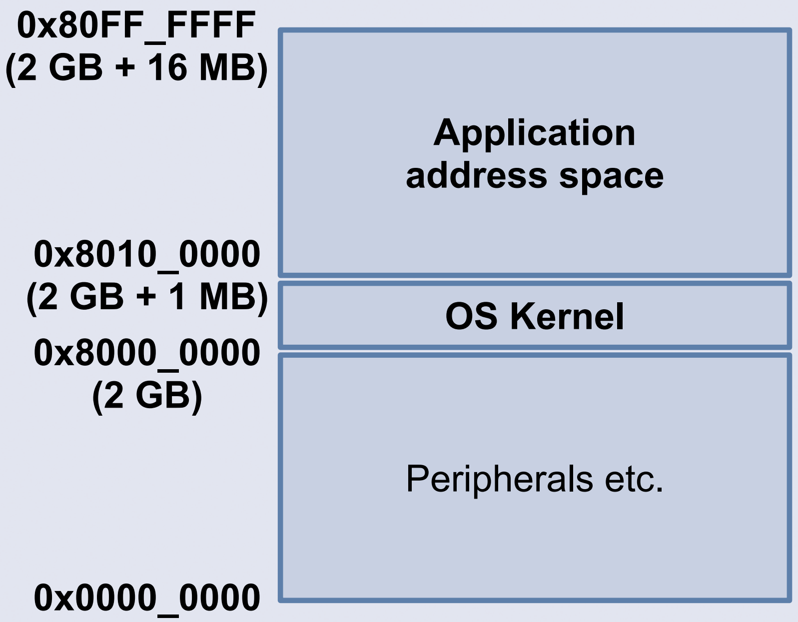
Universität Bamberg

- xv6 implements a small set of useful system calls
  - Real Unix systems have many more (but include the xv6 ones)

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Address spaces in xv6

- **Separate *address spaces* for kernel and applications**
  - The OS kernel and the application(s) use disjunct ranges of addresses for their code and data segments
  - On 32-bit systems, often the upper 1 GB of the 4 GB address range is reserved for the kernel



- We currently only have physical memory addresses available, so we start the kernel at the start of the RAM (0x8000_0000) and give it 1 **M**B of address space
- The application can use the rest (here: 15 MB out of 16)

Realizing separate address spaces

Universität Bamberg

- **How can we separate address spaces of kernel and apps?**
  - Currently, kernel and application code are compiled together
  - Only one text and data segment generated by the linker for kernel and application code…
    - …since the linker does not know which parts of the code belong to the kernel and which to the application

- ***We have to give the linker a hint!***
  - Tell the linker to move code and data of the application to a different address range
  - We still compile kernel and application together into one binary for now

Realizing separate address spaces

Universität Bamberg

- ***We have to give the linker a hint!***
  - with the **attribute** gcc extension, we can assign code and data to another section than the standard .text/.data/.bss:

```
int __attribute__ ((section (".usertext")))
main(void);
```

  - the linker script provides addresses for the new sections, e.g.:

```
/* user address space */
. = 0x80100000;
.usertext : {
    *(.usertext .usertext.*)
    . = ALIGN(0x1000);
    PROVIDE(eutext = .);
}
```

Realizing separate address spaces

- Result:

```
 $ riscv64-unknown-elf-nm hello | sort
[…]
000000008000047e T setup
0000000080001070 D uart0
0000000080001080 B stack0
0000000080100000 T syscall
0000000080100036 T main
```

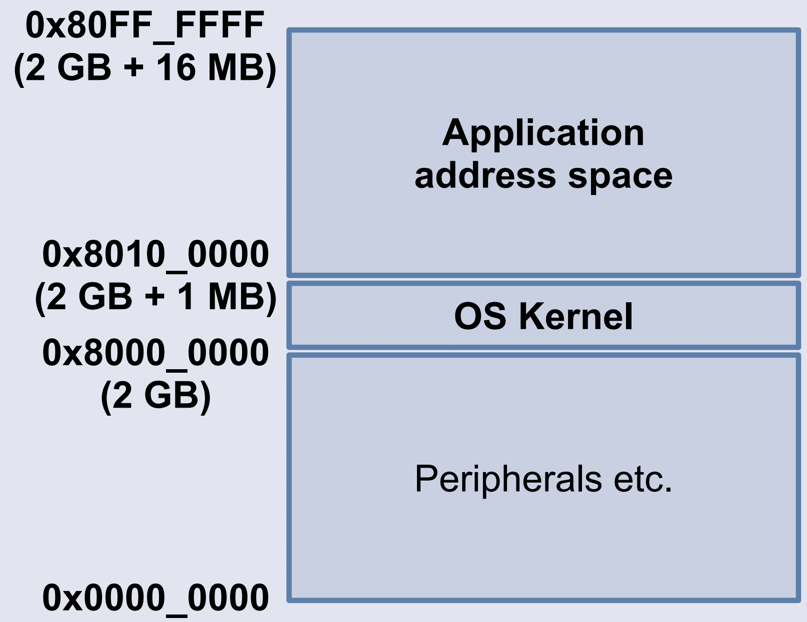
- The usermode functions (main and the "syscall" stub) are now in their own address range
- Global variables also have to be annotated to be stored in their own .userdata/.userbss sections (and the linker script extended)
- ***Unsolved: kernel and user mode still use the same stack!***

Protecting the OS via PMP


```
int *p = (int *)0x8000_1234; // in kernel space
*p = 0xdeadc0de;             // overwrite!
```

- **How can we protect the OS against this error (or attack)?**

- Use Physical Memory Protection (PMP) to make the kernel address space inaccessible from user space
- Also, direct access to the memory region below 0x8000_0000 should be prohibited
  - Otherwise, an application could directly manipulate devices!



Physical memory protection (again)

Universität Bamberg

- ***PMP is always active*** in S and U mode (off in M), by default all memory access is prohibited (*enforced in qemu 7 and in real hardware*)

- ***So far, we allowed user mode to access all memory directly*** (in the setup function)

```
// configure Physical Memory Protection to
// give user mode access to all of physical memory.
w_pmpaddr0(0x3fffffffffffull);
w_pmpcfg0(0xf);
```

- ***Idea:*** restrict the accessible address range for user space!

Physical memory protection CSRs

Universität Bamberg

- ***How? Yet another set of CSRs***
  - Up to 64 different memory segments configurable
- PMP address register **pmpaddr**x encodes bits 55–2 of a 56-bit physical address
  - Address related to memory segment x
- PMP configuration register **pmpcfg**x encodes access permissions for segment x

63 54	53 0
0 (**WARL**)	address[55:2] (**WARL**)
10	54

Figure 3.34: PMP address register format, RV64.

	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
`pmpcfg0`	pmp7cfg	pmp6cfg	pmp5cfg	pmp4cfg	pmp3cfg	pmp2cfg	pmp1cfg	pmp0cfg
	8	8	8	8	8	8	8	8

	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
`pmpcfg2`	pmp15cfg	pmp14cfg	pmp13cfg	pmp12cfg	pmp11cfg	pmp10cfg	pmp9cfg	pmp8cfg
	8	8	8	8	8	8	8	8

⋮

	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
`pmpcfg14`	pmp63cfg	pmp62cfg	pmp61cfg	pmp60cfg	pmp59cfg	pmp58cfg	pmp57cfg	pmp56cfg
	8	8	8	8	8	8	8	8

Figure 3.32: RV64 PMP configuration CSR layout.

Physical memory protection CSRs

- PMP configuration register **pmpcfg**x encode access permissions for segment x: eight bits per segment
  - Permissions for the segment: **R**ead, **W**rite, e**X**ecute
  - **A**: address matching
  - **L**: indicates that the PMP entry is locked, i.e., writes to the configuration register and associated address registers are ignored
    - *Locked PMP entries remain locked until reset*

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

Table 3.10: Encoding of A field in PMP configuration registers.

7	6–5	4–3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)
1	2	2	1	1	1

Figure 3.35: PMP configuration register format.

Physical memory protection (again)

Universität Bamberg

- ***What do we actually configure here?***

```
// configure Physical Memory Protection to
// give user mode access to all of physical memory.
w_pmpaddr0(0x3fffffffffffffull);
w_pmpcfg0(0xf);
```

63 54	53 0
0 (**WARL**)	address[55:2] (**WARL**)
10	54

Figure 3.34: PMP address register format, RV64.

- pmpaddr0: 54 "1" bits in bit 53…0

7	6 5	4 3	2	1	0
L (**WARL**)	0 (**WARL**)	A (**WARL**)	X (**WARL**)	W (**WARL**)	R (**WARL**)
1	2	2	1	1	1

Figure 3.35: PMP configuration register format.

- pmpcfg0:
  bits 3–0 = "1", rest "0"
  - R, W, X permissions
  - TOR (top of range) mode

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

Table 3.10: Encoding of A field in PMP configuration registers.

Physical memory protection in action

- ***What do we have configure to protect the kernel?***

- ***No permission (R,W,X all = 0)*** for 0x0000_0000 – 0x7fff_ffff (I/O)
- ***No permission (R,W,X all = 0)*** for 0x8000_0000 – 0x80ff_ffff (kernel)
- ***All permissions (R,W,X = 1)*** for 0x8010_0000 – 0x80FF_FFFF (app code/data)

- **Configure three ranges with TOR mode:**
  - range 0 (pmpaddr0/pmpcfg0) starts (implicitly) at 0, ends at 0x7fff_ffff
  - range 1 (pmpaddr1/pmpcfg1) starts after the end of range 0, ends at 0x800f_ffff
  - range 2 (pmpaddr2/pmpcfg2) starts after the end of range 1, ends at 0x80ff_ffff

- Homework :)

Protecting the OS from nasty apps…

Universität Bamberg

- **How can we protect the OS from accesses by user space?**
  - Use physical memory protection (PMP)

- ***Is this protection sufficient?***

- **What could happen in this case?**
  - Example: `write` system call:

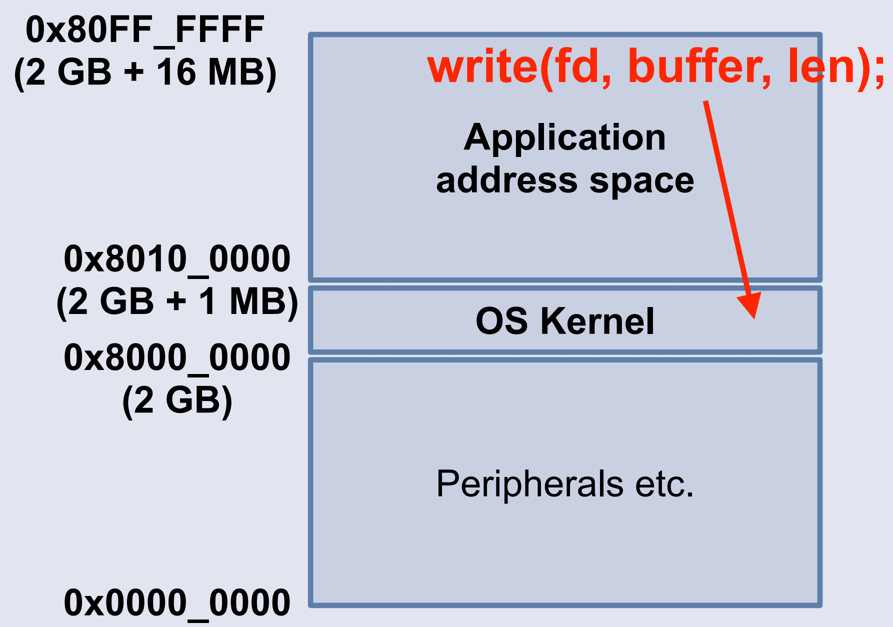
```
write(int fd, void* buffer, len_t length);
```

  - The parameter "`buffer`" passes a pointer to the address data should be written to
  - ***What if*** `buffer` ***points to the kernel address space?***

Protecting the OS via PMP


```
write(int fd, void* buffer, len_t length);
```

- ***What if*** `buffer` ***points to the kernel address space?***
  - Without protection, the kernel code implementing the `write` syscall would overwrite code or data in the kernel address space!
  - This can result in effects from no effect at all (overwriting unused memory) to crashes or security holes (overwriting process permission data)



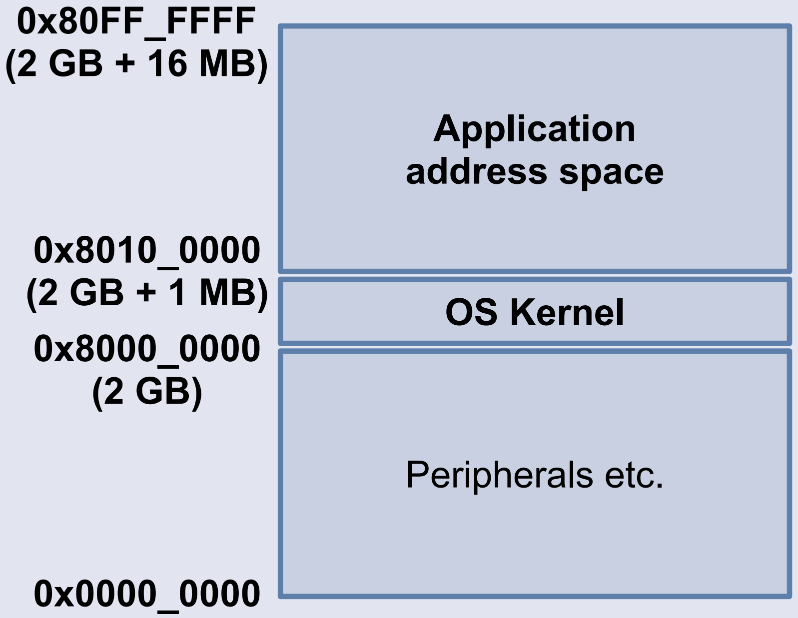
Protecting the OS via PMP


```
write(int fd, void* buffer, len_t length);
```

- How can we protect the OS against this attack?

- Check for a valid address in software – (*overhead!*):
```
(addr >= 0x8010_0000 &&
 addr <= 0x80FF_FFFF)
```

- PMP does not help here
    - ***Why?***



Switching processes

Universität Bamberg

- **With a separate address space for processes…**
  - we can now try to ***exchange*** the process in memory!

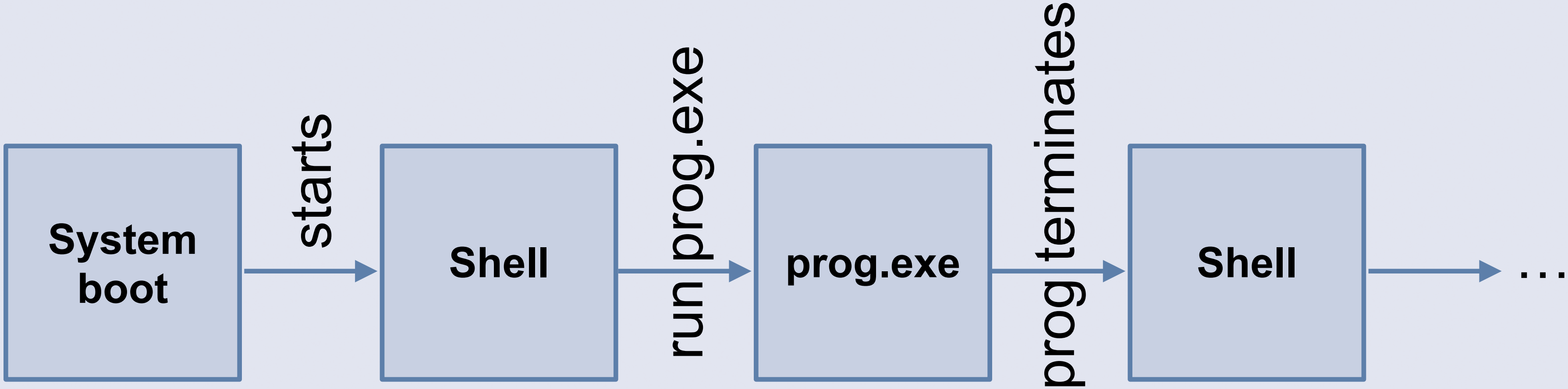
- ***Idea:*** simple MS-DOS-style operation
  - Only one process in memory at a time
  - This can start another process
  - Only when this process terminates, another one can be run
- ***Special treatment*** for the command interpreter (shell)
  - On MS-DOS, the shell (command.com) does not stay in memory when a program is started
  - Instead, the kernel loads it automatically again when a program terminates[1]

1 It's a bit more complex than this – details at https://retrocomputing.stackexchange.com/questions/306/how-does-the-command-com-shell-work-with-ms-dos

Switching processes

Universität Bamberg

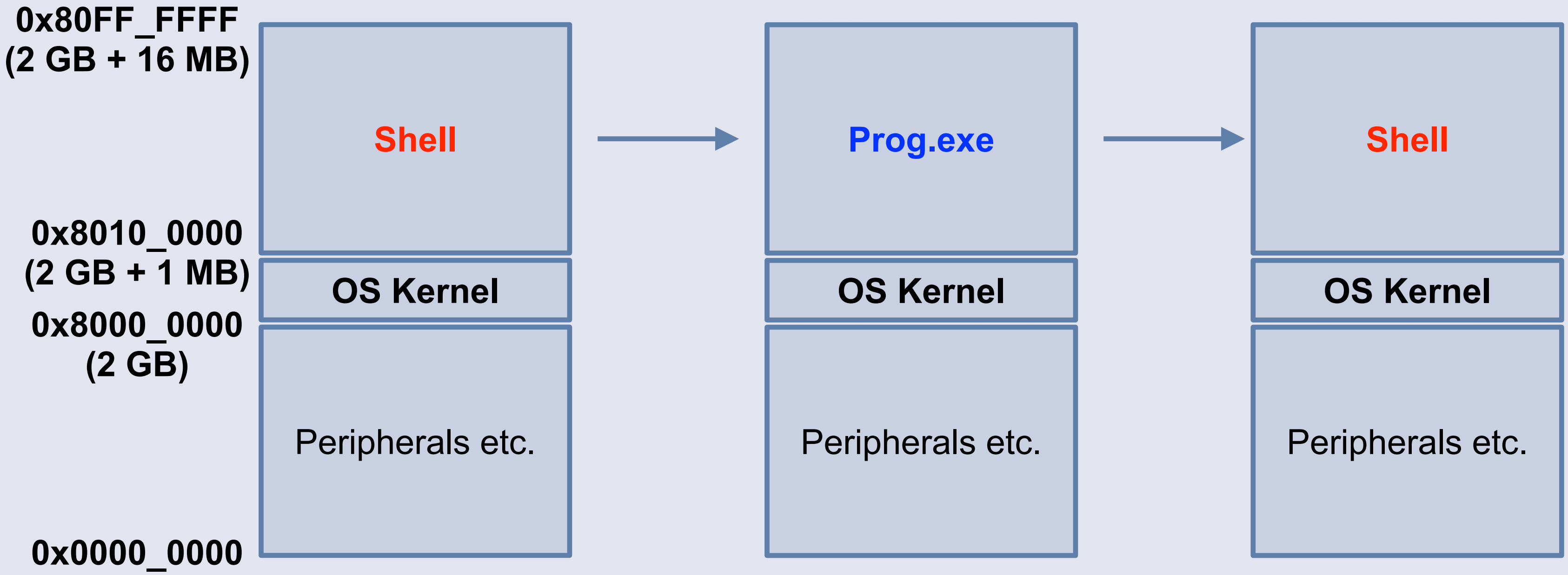
- **With a separate address space for processes…**
  - we can now easily ***exchange*** the process in memory!



- The OS kernel initially starts a given first process (shell)
- The shell then accepts commands
  - e.g. `run prog.exe` loads and runs the given program
  - to do this, the shell asks the kernel to ***execute*** the new program

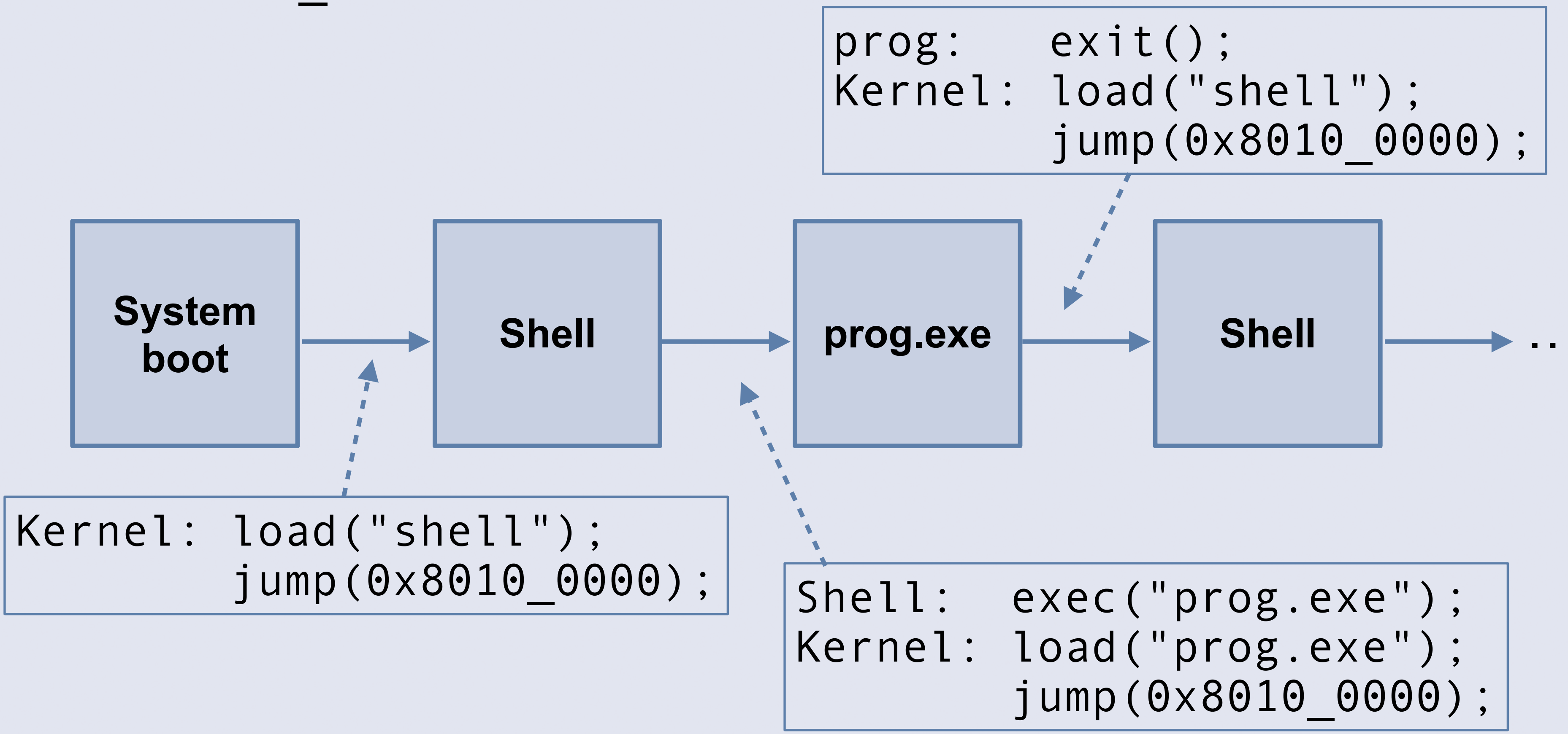
Switching processes: memory view

- **The OS has to replace the application memory range with the new application**
  - Kernel code and data remains in its memory range



Managing process switching

- **With a separate address space for processes…**
  - we could try to load different programs into memory at 0x8010_0000 one after the other



Houston, we have a problem…

Universität Bamberg

- **All applications would now have to be linked to start at address 0x8010_0000**
  - This is not (easily) possible using a single binary
    - Compile applications separately into executables
    - Use a linker script that starts at this address
      - No more need for separate .usertext/.userdata etc.
  - Compile separate binaries for different applications
    - e.g. all DOS (.COM) executables always start at address 0x100 and are more or less a simple dump of the program as is appears in memory
    - We can achieve a similar effect using `objcopy -O binary`
- Unix (and DOS/Windows EXE formats) use a more complex executable format
  - See lecture 2 for a description of the ELF format

Where do the apps come from?

- **Apps would require a *file system* to be loaded…**
  - we don't have one yet (will discuss this later)

- So far, we can "cheat" a bit
  - Keep all code of applications in separate arrays in memory (sort of a RAM disk)
  - Copy code over to address 0x8010_0000 as required

- As long as we ***run all processes to completion***, we don't have to care about…
  - Protecting their data, bss and stack
  - Remembering where a process terminated
  - ***…we'll take care of this soon…***

Conclusion

Universität Bamberg

- **We are slowly adding functionality to our OS**
  - Protection of the kernel
  - Starting processes

- What's missing here
  - switching between processes in memory

- …however, we are still quite a bit away from a Unix-like system
  - Multitasking is still missing
  - All processes run to completion

- **Next step:** make switching between processes work and then implement ***cooperative multitasking***

References

1. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, Document Version 20211203
2. SiFive, *RISC-V Security Architecture Introduction*, https://sifive-china.oss-cn-zhangjiakou.aliyuncs.com/西安珠海杭州合肥ppt/04%20hujin%20RISC-V%20Security%20Architecture%20Introduction_4%20City.pdf