User process

- Advanced debugging technique
- User process
- Programs and file system
- Create the first user process
- Induction to system call

- Advanced debugging technique

Suggestions for debugging

- Always enable –Wall & -Werror.
- Insert as many assertions as possible.
- Use printk to output debug information.
- Use GDB to make the execution flow clear.

- Interrupts & exceptions make the execution flow more complex.
  - Can you find the last instruction before interrupt/exception is triggered?

Assertion in Lab2

- check the status of IF bit in EFLAGS

```
#define INTR assert(read_eflags() & IF_MASK)
#define NOINTR assert(~read_eflags() & IF_MASK)
```

- insert them in your code
- consistency
  - NOINTR when in critical region or during interrupt
  - INTR otherwise

Trap 3 – use locking in interrupts

```
void timer_handler() {
    // ...
    NOINTR;
    V(sem);
    NOINTR;
    // ...
}
```

```
void V(Sem *s) {
    lock(); NOINTR;
    // ...
    unlock();
}
```

should not sti()

Display the dying state

- How to display the state when an exception is triggered?
- All information is already saved in the stack!
  - tf->eip
  - tf->eax, tf->ebx...
- display the content in the stack
  - printk("%x %x %x %x %x %x")
- display stack trace in gdb
  - use "bt"

Mysterious reboot

http://wiki.osdev.org/Triple_Fault

- Why mysterious reboot?
  - triple fault
- fault -> exception
- fault during exception -> double fault
- fault during double fault -> triple fault
- triple fault -> reboot

Mysterious reboot (cont.)

- How to catch the state before mysterious reboot?

```
-no-reboot
    Exit instead of rebooting.

-no-shutdown
    Don't exit QEMU on guest shutdown, but instead only stop the emulation.  This allows
    for instance switching to monitor to commit changes to the disk image.
```

- User process

User process

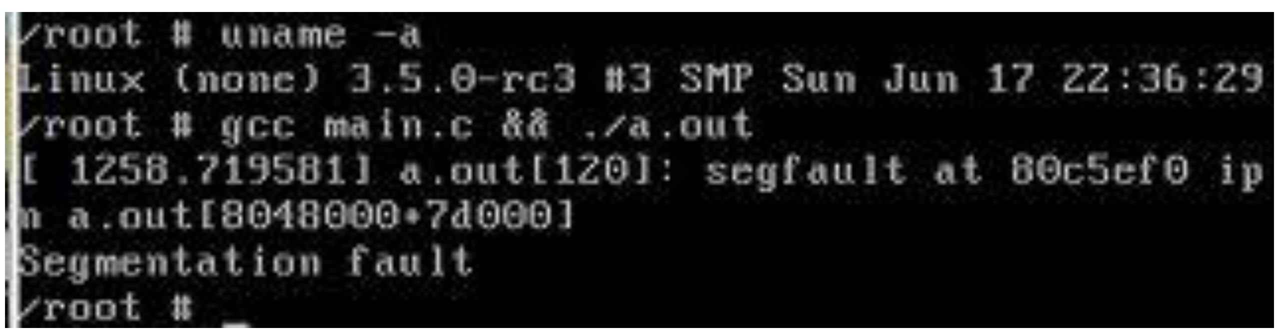
- a process running in user mode
  - lower privilege level
  - obey lots of restrictions
    - IA-32 memory/interrupt protection
  - can do nothing except "computing"
- Why?
  - OS should not let a user process do everything it want to.
  - Only legal actions are allowed.

Address space

- Kernel threads share the same address space with kernel.
  - every kernel thread can access “current”
  - can even modify other’s data
- A user process has its address space.
  - a legal “field” where it can perform legal actions
    - code, data, stack
  - presented by page directory and page tables

Address space (cont.)

- A process trying to access illegal "field" will be killed.

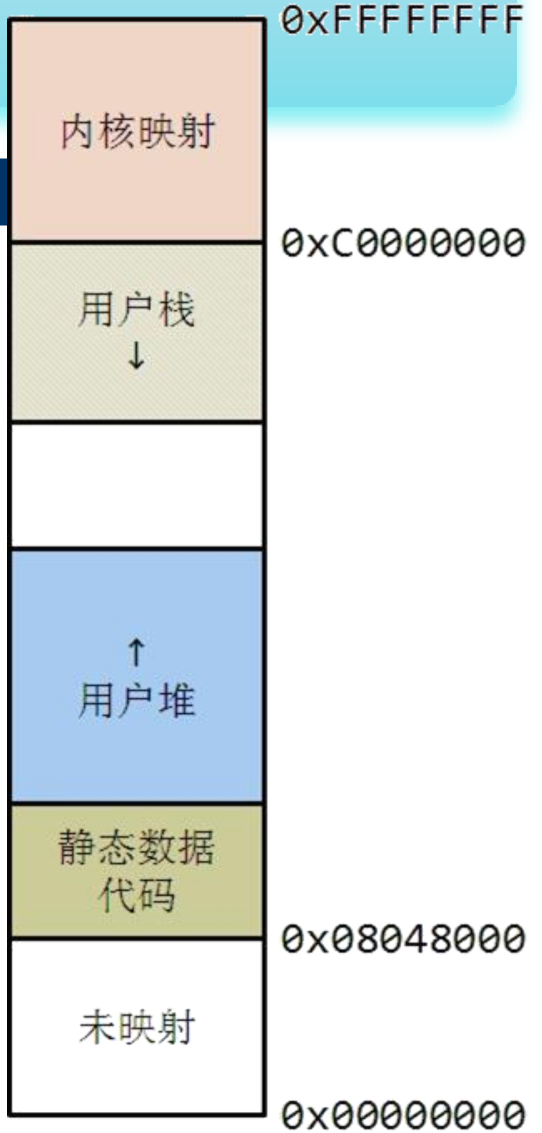


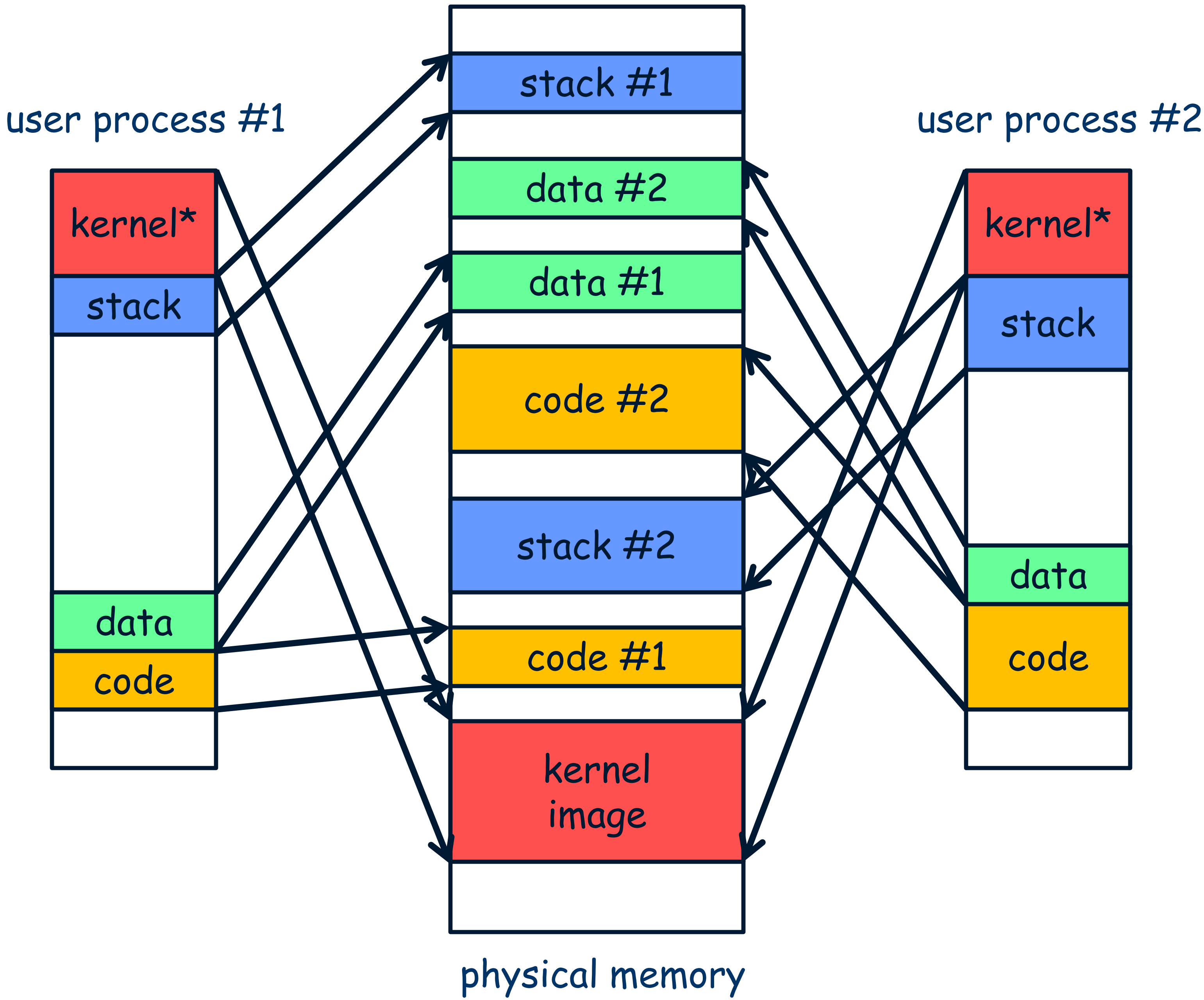
Memory layout

- In Nanos, the memory layout of user process is the same as the one in Linux
  - Linker in Linux will put everything in the right place.
  - the same layout for every user process
- The same VA of different processes maps to different PAs.
  - how to implement this?

Memory layout (cont.)

- code starts from 0x08048000
  - why not 0x00000000?
- data start next to codes
- heap starts above data
  - grows up
- above 0xc0000000 is kernel*
- user stack starts below kernel*
  - grows down
  - what about the stack in PCB?




user process #1
kernel*
stack
data
code
stack #1
data #2
data #1
code #2
stack #2
code #1
kernel
image
user process #2
kernel*
stack
data
code
physical memory

Why use kernel*?

- Both user processes and kernel threads have kernel* in their address space.
- For user processes
  - kernel* makes it possible to access GDT & IDT
    - without kernel*, user processes in Nanos will crash once interrupt comes
- For kernel threads
  - they see the same kernel image as user processes do
    - pointers passed from kernel* can be used directly

- Programs and file system

Program

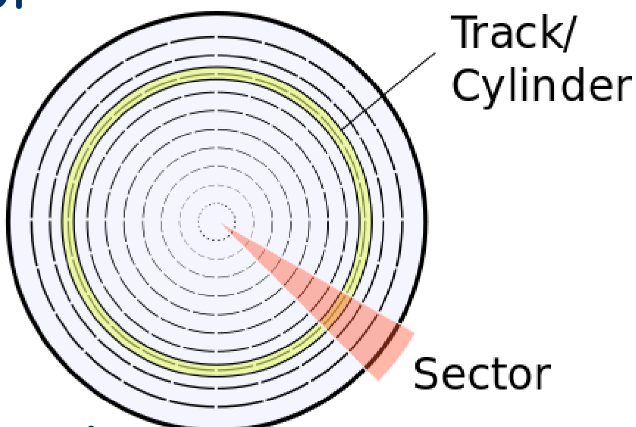
- When kernel initializes itself, there is no user process.
- Where do user processes come from?
  - from programs
- Where do programs come from?
  - from storages, such as disk!
  - They are all executable files!

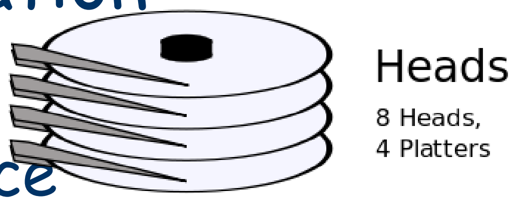
Cross-compilation

- cannot compile a program under Linux by default settings
  - the linker will try to access the system library in Linux, which does not exist in Nanos
- use the flags that we compile the kernel
  - the same CFLAGS & LDFLAGS
  - see Makefile for reference

What is a disk?

- for disk controller
  - CHS addressing to find a sector
  - 512 bytes for each sector
- for IDE
  - LBA addressing
    - logical block addressing
- IDE performs the transformation
  - $A = (c \cdot N_{\text{heads}} + h) \cdot N_{\text{sectors}} + (s - 1)$
  - provide a view of sector sequence





Accessing disk

- We must load the program from disk to memory.
- We already have IDE driver.
- But for simplicity, we put programs in memory.
  - Memory used as disk is called RAMDISK.
  - easier to initialize, easier to debug

Disk as byte sequence

- Memory has more fine-grained units.
  - Sector sequence is not a good way to model RAMDISK.
- from sequence to byte sequence
  - number each byte in disk
    - $A = (c \cdot N_{\text{heads}} + h) \cdot N_{\text{sectors}} + (s - 1)$
    - B = A * 512 + offset_in_sector
- It is convenience to implement RAMDISK from the view of byte sequence.
  - uint8_t disk[NR_DISK_SIZE];
  - but we must provide a high-level view for user process

File system

- manage the mapping from files to bytes
  - FS converts file operations into read/write requests to IDE/RAMDISK.
- For now, a simplified FS is sufficient:
  - File names are integers.
    - 0, 1, 2, 3…
  - Files are mix-length.
    - for example, 128KB each
  - Files are located sequentially on disk.

File system (cont.)

128KB 256KB 384KB

file #0	file #1	file #2	……

- trick for initialization with RAMDISK

```
static uint8_t file[NR_MAX_FILE][NR_FILE_SIZE] = {
        {0x12, 0x34, 0x56, 0x78},          // the first file '0'
        {"Hello World!\n"},                // the second file '1'
        {0x7f, 0x45, 0x4c, 0x46, ...}      // the third file '2'
};
static uint8_t *disk = (void *)file;
```

- in Lab 4, you will extend this simplified FS to a complex one

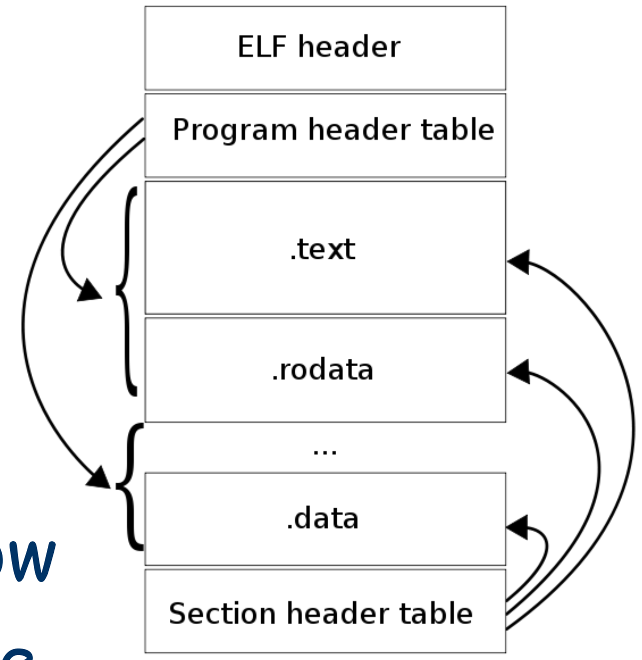
Executable file

- An executable file should contain essentials of a program.
  - code, rodata, data…
  - Where is stack?
- Only piling the essentials up to form a file is not a feasible solution.
  - You can not tell how much code there is.
- A way to organize the essentials is needed.
  - Different ways constitute different file formats.

ELF file format

- ELF header
- program headers
- section headers
- body of the binary file

- Use *readelf* command to show information about an ELF file.



ELF header

- global information about the ELF file

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0xc01037d0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          47456 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         9
  Section header string table index: 6
```

Section header

- sections from the view of linking

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        c0100000 001000 008fda 00 WAX  0   0 16
  [ 2] .rodata           PROGBITS        c0108fe0 009fe0 001909 00   A  0   0 32
  [ 3] .data             PROGBITS        c010a8ec 00b8ec 000018 00  WA  0   0  4
  [ 4] .bss              NOBITS          c010b000 00b904 b50928 00  WA  0   0 40
96
  [ 5] .comment          PROGBITS        00000000 00b904 00001c 01  MS  0   0  1
  [ 6] .shstrtab         STRTAB          00000000 00b920 00003d 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 00bac8 001630 10      8 130  4
  [ 8] .strtab           STRTAB          00000000 00d0f8 000da0 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Section header (cont.)

- .text
  - binary code
- .rodata
  - read only data, such as constant, string
- .data
  - initialized global/static variables
- .bss
  - Block Started by Symbol
  - non-initialized global/static variables

Section header (cont.)

- What is the different between .data and .bss?
  - int a = 1234;
    - the initial value should be recorded in the ELF file
  - int b [1024*1024];
    - the array will occupy 4MB of memory
    - but the ELF file is far less than 4MB
    - for data in .bss, only address and size are needed
    - the initial value of non-initialized variables is 0 by default
- Where are local variables?

Program header

- segments from the view of run-time

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0xc0100000 0xc0100000 0x0a904 0xb5b928 RWE 0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
```

- This is actually what we should focus on:
  - Offset – relative location in the file
  - VirtAddr – VA which this segment should be loaded
  - FileSiz
  - MemSiz

Parsing an ELF file

- In fact, bootblock has done this.

```
/* The binary is in ELF format (please search the Internet).
   0x8000 is just a scratch address. Anywhere would be fine. */
elf = (struct ELFHeader*)0x8000;

/* Read the first 4096 bytes into memory.
   The first several bytes is the ELF header. */
readseg((unsigned char*)elf, 4096, 0);

/* Load each program segment */
ph = (struct ProgramHeader*)((char *)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph ++) {
    pa = (unsigned char*)(ph->paddr - KOFFSET); /* physical address */
    readseg(pa, ph->filesz, ph->off); /* load from disk */
    for (i = pa + ph->filesz; i < pa + ph->memsz; *i ++ = 0);
}
```

- Create the first user process

Create a process

- allocate a PCB
- create address space
- load the program to the right memory position
- initialize PCB and user stack
- put the process in the ready queue

- simplification: let user processes run in ring0
  - ignore details about stack for now
  - we will put them in ring3 in the future

Create a process (cont.)

- This is rather complex, it involves
  - process management
  - memory management
  - file management
- We'd better divide them into different servers.
  - PM for process management
  - MM for memory management
  - FM for file management
  - In Nanos, they run as kernel threads.

PM

- create the first user process
  - "0" for now
  - "/bin/sh" in Lab4
- communicate with MM and FM to finish this task
  - synchronization is necessary
    - how to implement?
  - pay attention to dead locks
- handle process-related system calls
  - explain in the future

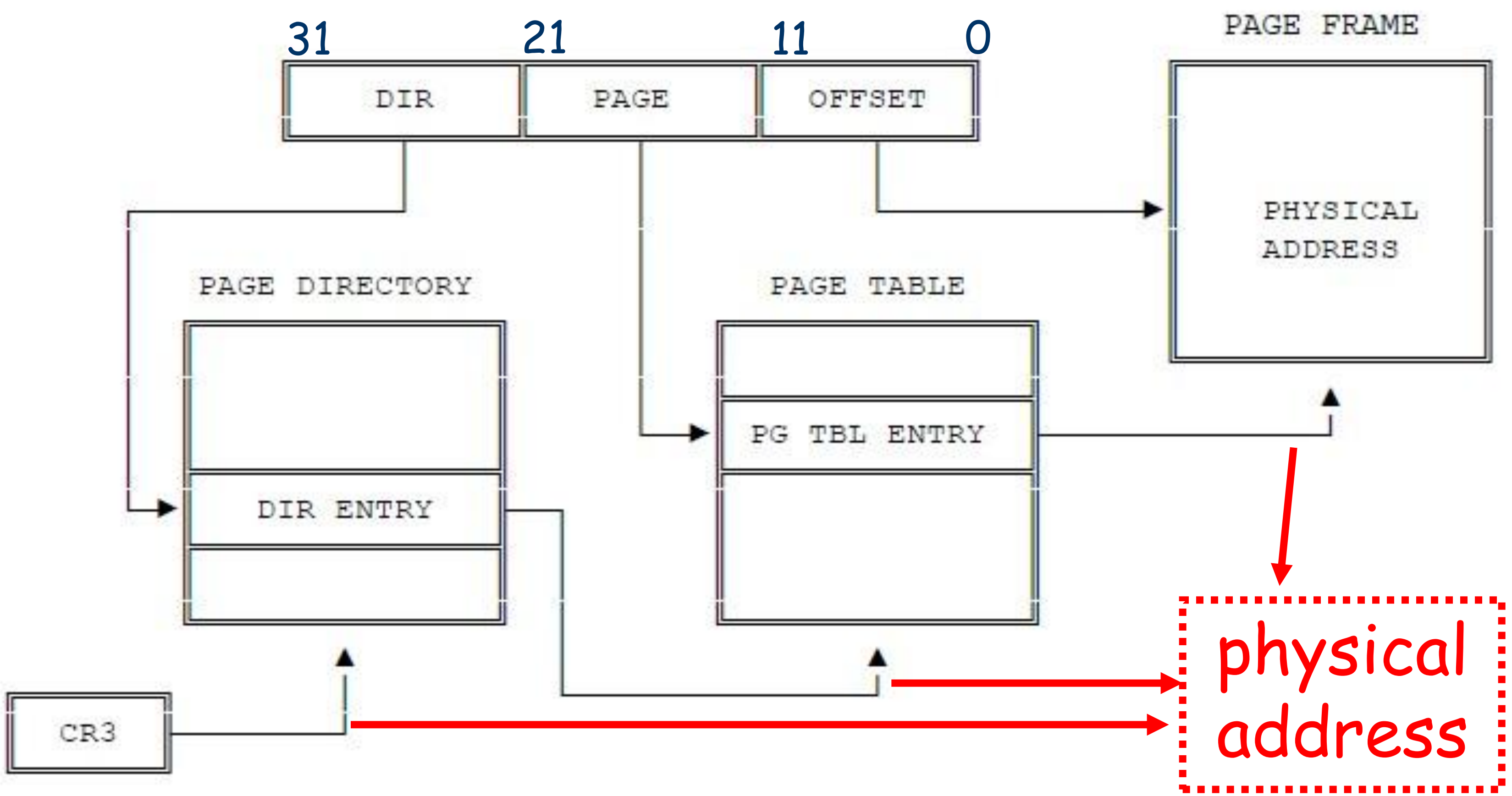
MM

- page allocation
  - MM organizes available physical memory into pages (4KB per page)
- Nanos has physical memory of 128MB (default setting of qemu)
  - lower 16MB for kernel , the rest for user process
  - rest = (128 MB – 16 MB) / 4KB = 27684 pages
  - MM should record which pages are already allocated
    - bitmap is OK

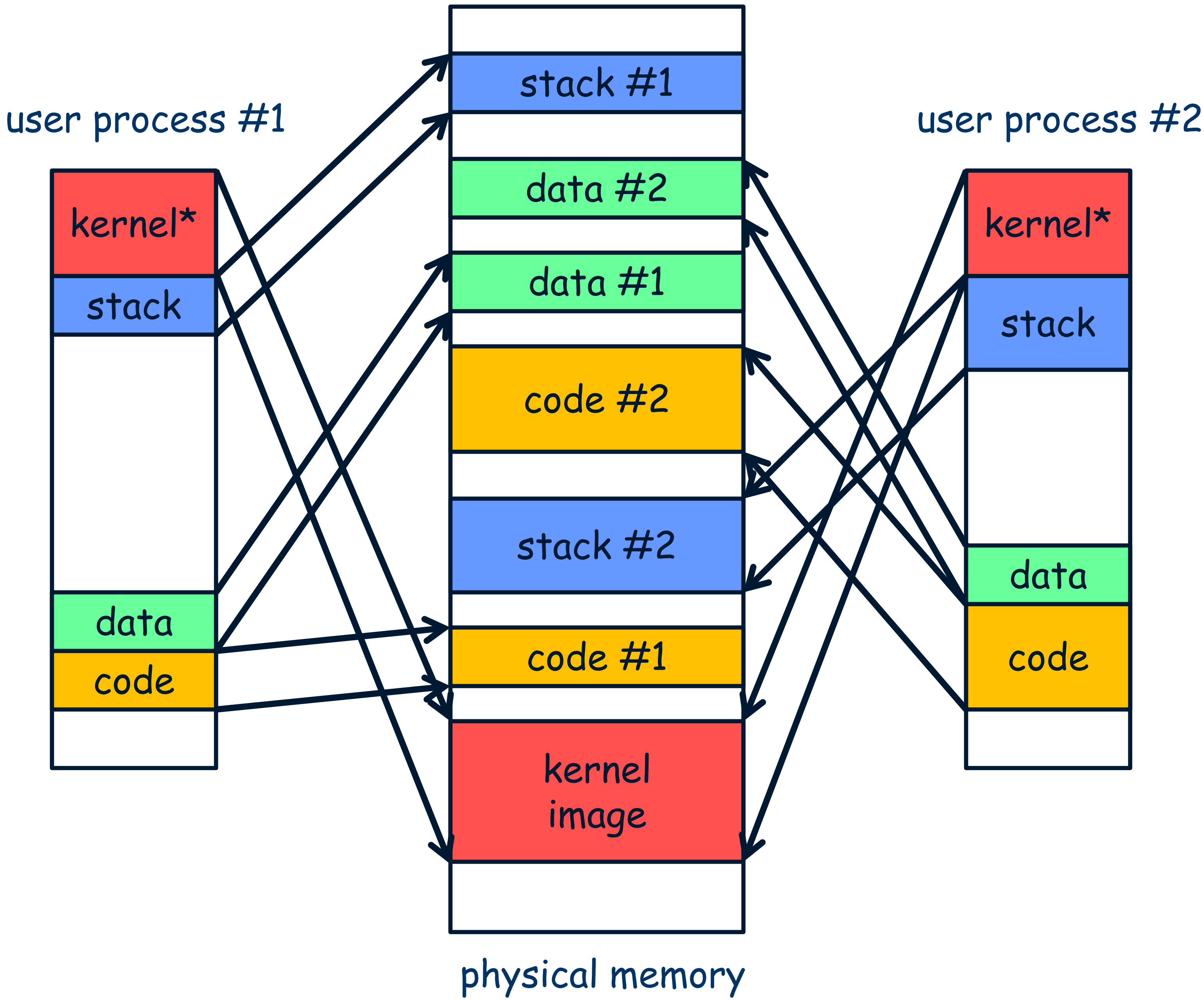
MM (cont.)

- create address space
  - allocate pages for code and data
  - allocate one page for user stack
  - map the memory above 0xc0000000 to kernel
- create address space = fill in the pdir & ptable
  - to make virtual memory map to the right place
  - The address fields in pde, pte and CR3 should be PHYSICAL address!
    - why?
- CR3 should be switched during context switch.

MM (cont.)

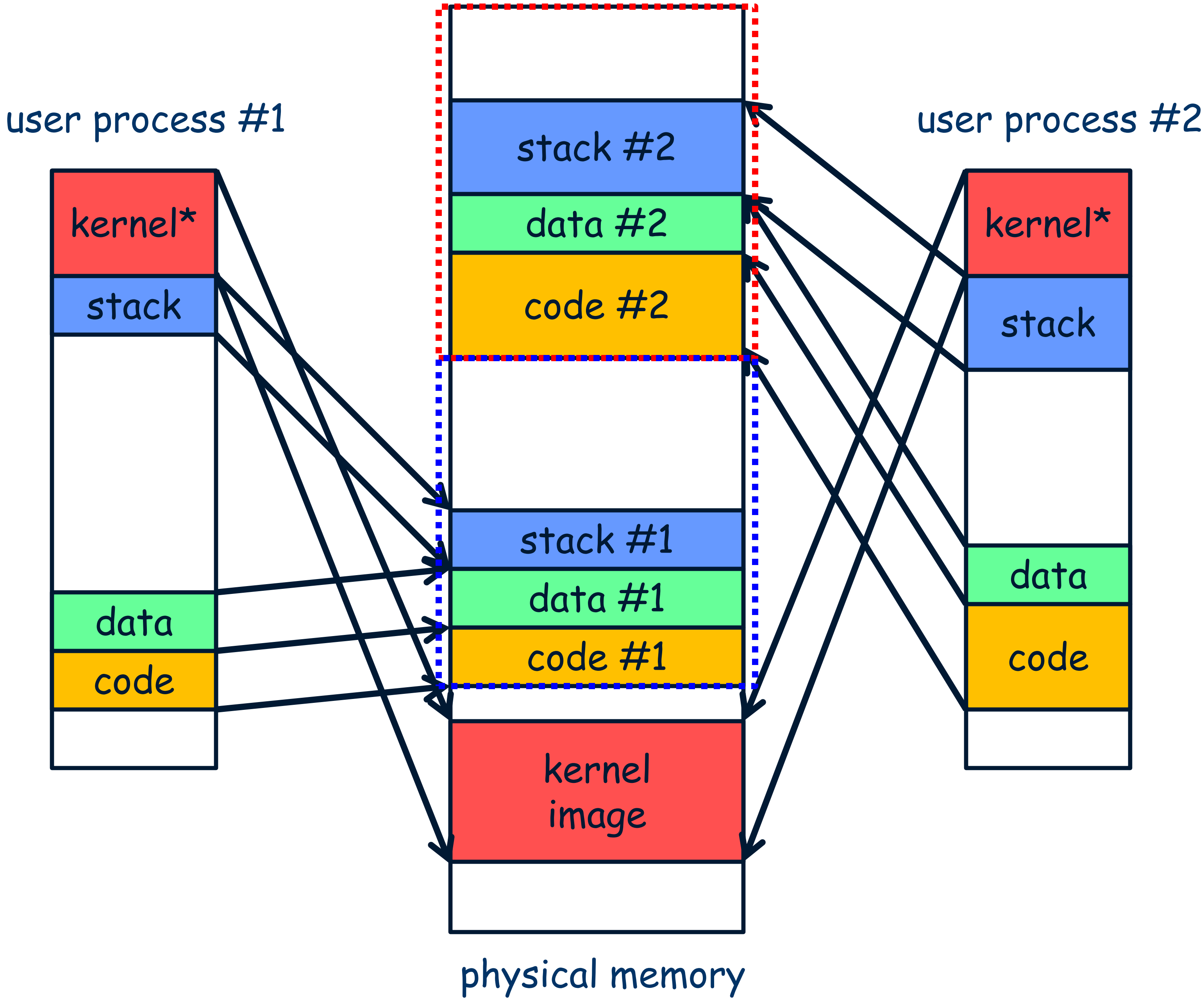


31
21
11
0
DIR
PAGE
OFFSET
PAGE FRAME
PHYSICAL ADDRESS
PAGE DIRECTORY
PAGE TABLE
PG TBL ENTRY
DIR ENTRY
CR3
physical address


user process #1
kernel*
stack
data
code
stack #1
data #2
data #1
code #2
stack #2
code #1
kernel
image
physical memory
user process #2
kernel*
stack
data
code

Simplification – segmentation

- Segmentation is easier than paging to implement an the level of OS memory management.
- Let's make it simpler further.
  - static segmentation
  - associate each process with a fixed piece of memory segment
- MM becomes much easier to implement.
- We will upgrade to paging in the future.



user process #1
kernel*
stack
data
code
stack #2
data #2
code #2
stack #1
data #1
code #1
kernel
image
physical memory
user process #2
kernel*
stack
data
code

FM

- implement the simplified FS
  - file names are integers
  - map file names to bytes
- provide read operation
  - do_read(filename, buf, offset, len)
  - read *len* bytes starting from the *offset*th byte of *filename* into *buf*
  - communicate with RAMDISK

- Introduction to System call

Legal demands

- Legal demands must be satisfied.
  - printf
  - open a file
  - fork
- All legal service should be provided by kernel.
- For the sake of safety, a user process can trap into kernel only by system calls.

In the real world

View from user process

- set arguments
  - eax – system call ID
  - ebx – argument #1
  - ecx – argument #2
  - edx, esi, edi
  - CANNOT flush esp & ebp!
- int $0x80
  - trap into kernel, wait for return
    - similar to RPC (remote procedure call)
  - Return value is stored in eax.

View from user process (cont.)

- example:

```
movl $4, %eax
movl $1, %ebx
movl $output, %ecx
movl len, %edx
int $0x80
```

View from kernel

- Kernel catches an exception.
  - should be treated specially
- All information is saved in TrapFrame by *asm_do_irq*.
- Kernel dispatches system calls according to the system call ID.
- System calls are handled by appropriate servers.

Dispatch system calls

```
// dispatch the system call task
void do_syscall(struct TrapFrame *tf){
    int id = tf->eax;

    switch(id){
    case SYS_exit:      syscall_exit(tf); break;
    case SYS_fork:      syscall_fork(tf); break;
    case SYS_exec:      syscall_exec(tf); break;
    case SYS_sleep:     syscall_sleep(tf); break;
    ......

    default:    panic("Unknown system call type!");
    }
}
```

First system call

- Without system calls, user process even cannot output a message.
- To test your first user process, you have to implement your first system call.
  - *puts()* is the simplest one, *printf()* is recommended
  - after trapping into kernel, call *printk()*
  - in Linux, they use *write()* to output to screen
    - we will implement *write()* in Lab 4
    - for now, just use *printk()* for simplicity

Lab3 is partially out!

- the first stage
  - implement RAMDISK and FM
- the second stage
  - create your first process
- to be continue...
- Have fun!