Thread management

- Recap – context switch
- Thread management
- Parallel programming with threads

Supplement

- What happen when nested interrupts come, if "current state" is stored at a fixed place?
- Suppose there is such a CPU architecture. When interrupts come, "current state" will be stored at a fixed place.

Supplement

We are in general thread.

interrupt 1 comes

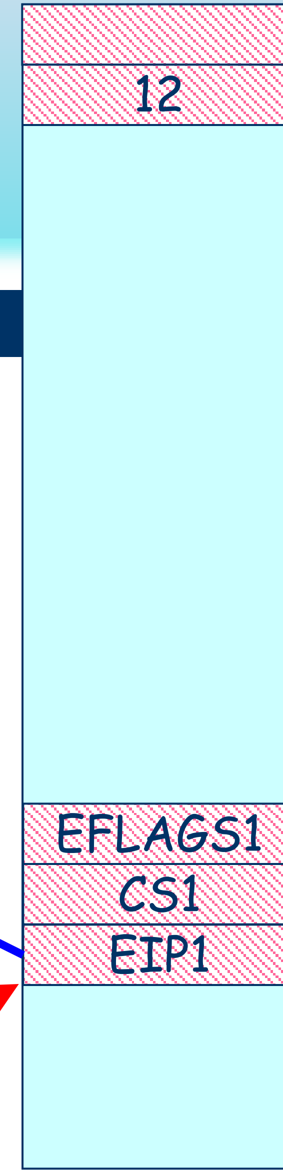
```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
}
```

12

Supplement

We are in general thread.

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
}
```



"current state" is always stored here

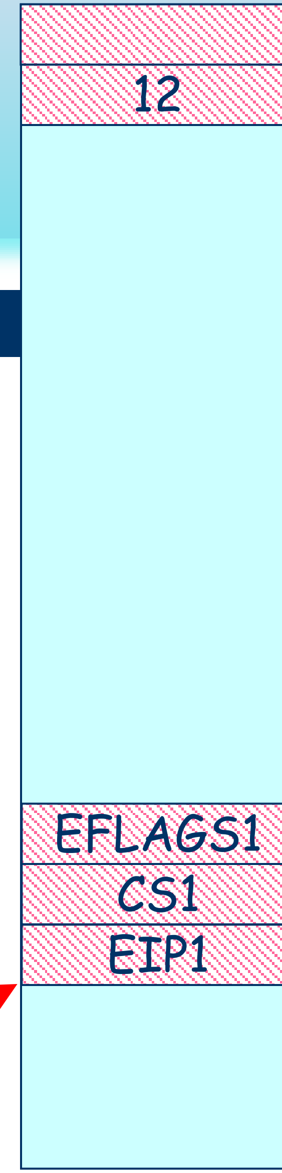
Supplement

We are in interrupt 1.

```
# 中断和异常处理函数的入口
# 中断处理函数会在IDT中为相应的中断/异常设置处理程序
# 中断/异常的行为参见i386手册
.globl vec0; vec0: pushl $0; jmp asm_do_irq
.globl vec1; vec1: pushl $1; jmp asm_do_irq
.globl vec2; vec2: pushl $2; jmp asm_do_irq
.globl vec3; vec3: pushl $3; jmp asm_do_irq
.globl vec4; vec4: pushl $4; jmp asm_do_irq
.globl vec5; vec5: pushl $5; jmp asm_do_irq
.globl vec6; vec6: pushl $6; jmp asm_do_irq
.globl vec7; vec7: pushl $7; jmp asm_do_irq
.globl vec8; vec8: pushl $8; jmp asm_do_irq
.globl vec9; vec9: pushl $9; jmp asm_do_irq
.globl vec10; vec10: pushl $10; jmp asm_do_irq
.globl vec11; vec11: pushl $11; jmp asm_do_irq
.globl vec12; vec12: pushl $12; jmp asm_do_irq
.globl vec13; vec13: pushl $13; jmp asm_do_irq

.globl irq0; irq0: pushl $1000; jmp asm_do_irq

.globl irq_empty; irq_empty: pushl $-1; jmp asm_do_irq
```



Supplement

We are in interrupt 1.

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
```

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                    __FUNCTION__, __LINE__);
```

“current state” is always stored here

Supplement

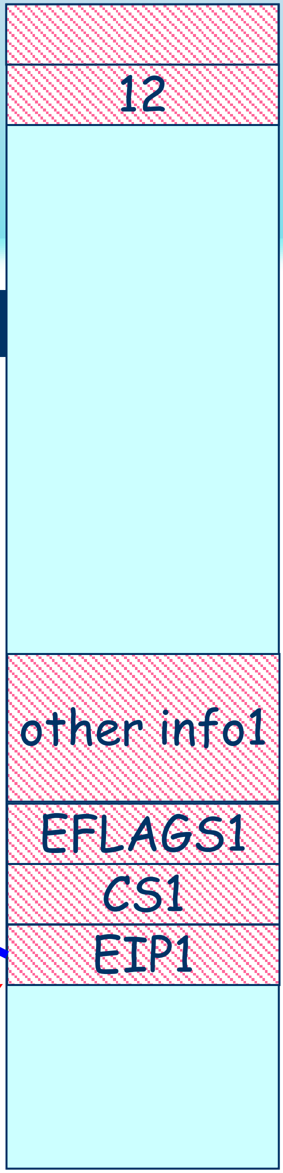
We are in interrupt 1.

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
```

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                        FUNCTION , LINE );
```

interrupt 2 comes

"current state" is always stored here



Supplement

We are in interrupt 1.

the "current state" of interrupt 1 is flushed !!!!

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
```

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                    __FUNCTION__, __LINE__);
```

"current state" is always stored here

Supplement

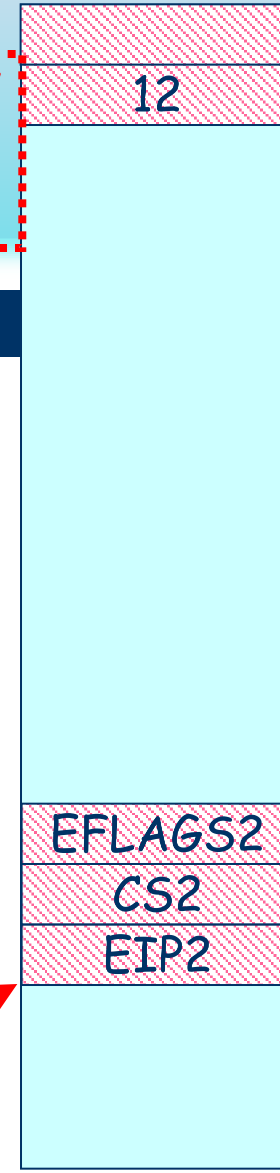
the "current state" of interrupt 1 is flushed !!!!

We are in interrupt 2.

```
# 中断和异常处理函数的入口
# 中断处理函数会在IDT中为相应的中断/异常设置处理程序
# 中断/异常的行为参见i386手册
.globl vec0; vec0: pushl $0; jmp asm_do_irq
.globl vec1; vec1: pushl $1; jmp asm_do_irq
.globl vec2; vec2: pushl $2; jmp asm_do_irq
.globl vec3; vec3: pushl $3; jmp asm_do_irq
.globl vec4; vec4: pushl $4; jmp asm_do_irq
.globl vec5; vec5: pushl $5; jmp asm_do_irq
.globl vec6; vec6: pushl $6; jmp asm_do_irq
.globl vec7; vec7: pushl $7; jmp asm_do_irq
.globl vec8; vec8: pushl $8; jmp asm_do_irq
.globl vec9; vec9: pushl $9; jmp asm_do_irq
.globl vec10; vec10: pushl $10; jmp asm_do_irq
.globl vec11; vec11: pushl $11; jmp asm_do_irq
.globl vec12; vec12: pushl $12; jmp asm_do_irq
.globl vec13; vec13: pushl $13; jmp asm_do_irq

.globl irq0; irq0: pushl $1000; jmp asm_do_irq

.globl irq_empty; irq_empty: pushl $-1; jmp asm_do_irq
```



Supplement

We are in interrupt 2.

the "current state" of interrupt 1 is flushed !!!!

```
while(1){
        x = (x + 1) % 100;
        if(x == 0){
                printk("a");
        }
```

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                    FUNCTION , LINE );
```

"current state" is always stored here

Supplement

returning from interrupt 2 is OK

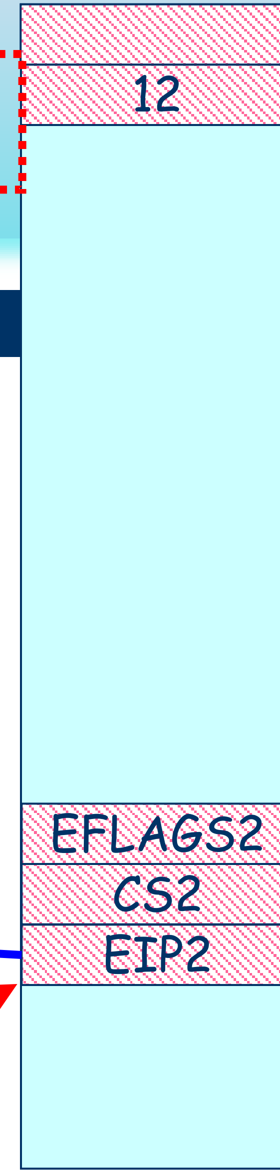
We are in interrupt 2.

```
.globl asm_do_irq
.extern irq_handle
asm_do_irq:
    pushal

    pushl %esp          # ???
    call irq_handle
    addl $4, %esp

    popal
    addl $4, %esp
    iret
```

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                    __FUNCTION__, __LINE__);
```



"current state" is always stored here

Supplement

returning from interrupt 2 is OK

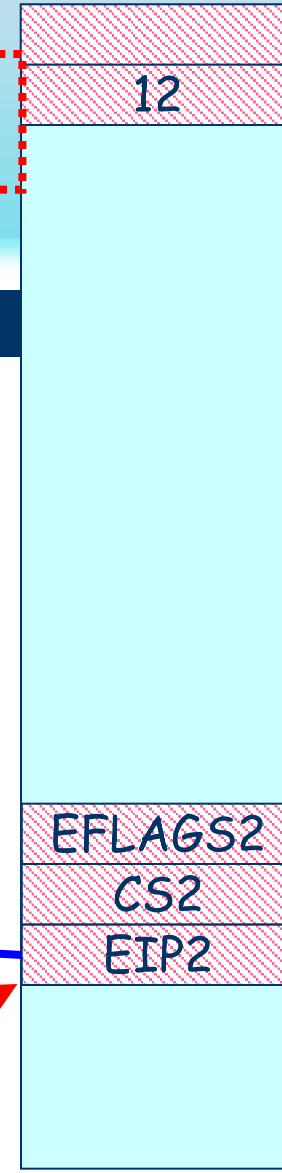
We are in interrupt 1.

```
.globl asm_do_irq
.extern irq_handle
asm_do_irq:
    pushal

    pushl %esp          # ???
    call irq_handle
    addl $4, %esp

    popal
    addl $4, %esp
    iret
```

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                    FUNCTION  ,   LINE  );
```



“current state” is always stored here

Supplement

We are in interrupt 1.

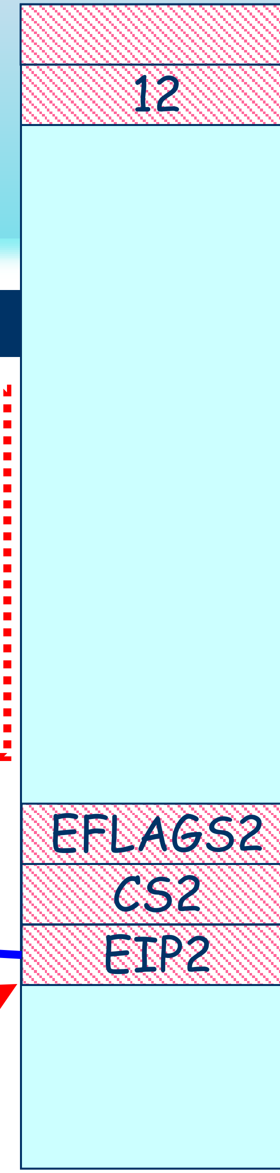
```
.globl asm_do_irq
.extern irq_handle
asm_do_irq:
    pushal

    pushl %esp          # ???
    call irq_handle
    addl $4, %esp

    popal
    addl $4, %esp
    iret
```

Now it is ready to return from interrupt 1 to the general thread, but EIP1 is flushed by EIP2 !!!

```
void
irq_handle(struct TrapFrame *tf) {
    if(tf->irq < 1000) {
        if(tf->irq == -1) {
            printk("%s, %d: Unhandled exception!\n",
                    __FUNCTION__, __LINE__);
```



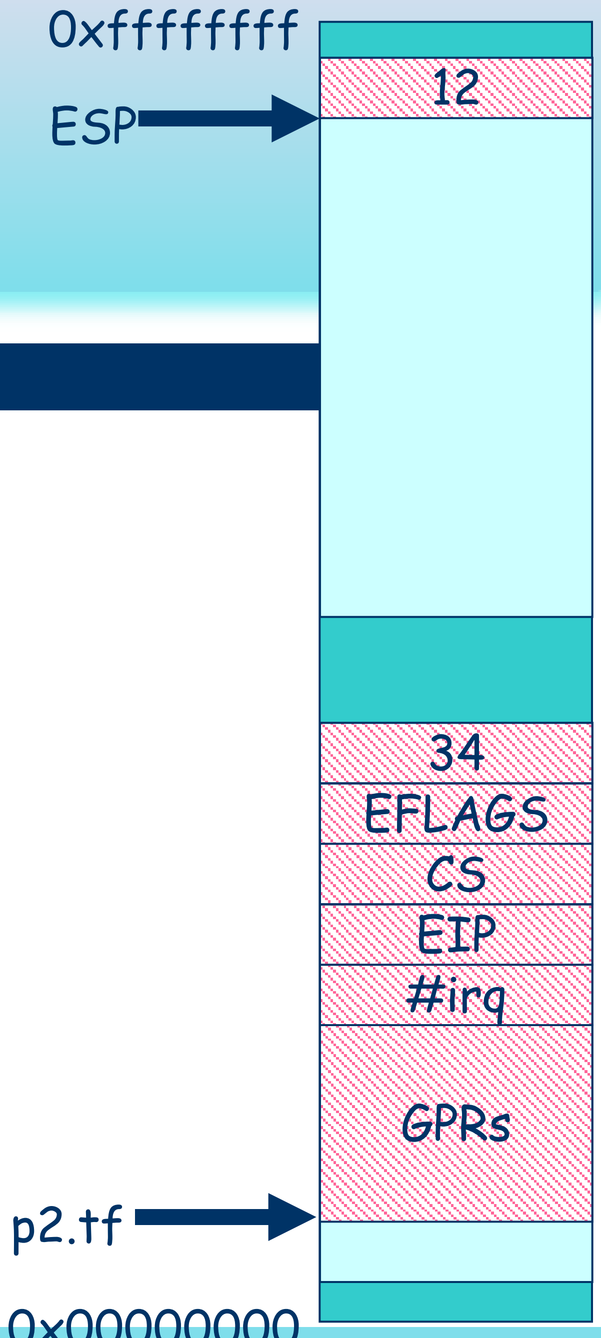
“current state” is always stored here

It gets stuck in irq_handle and asm_do_irq.

- Recap - context switch

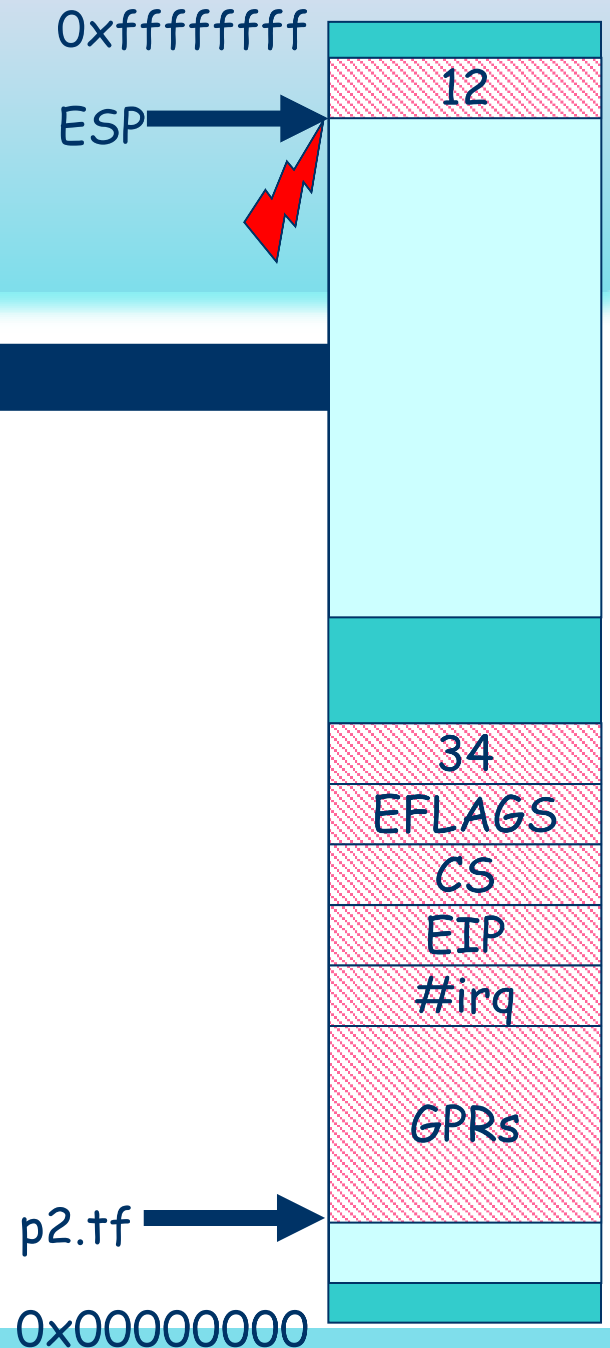
Context switch (cont.)

- process #1 is running...



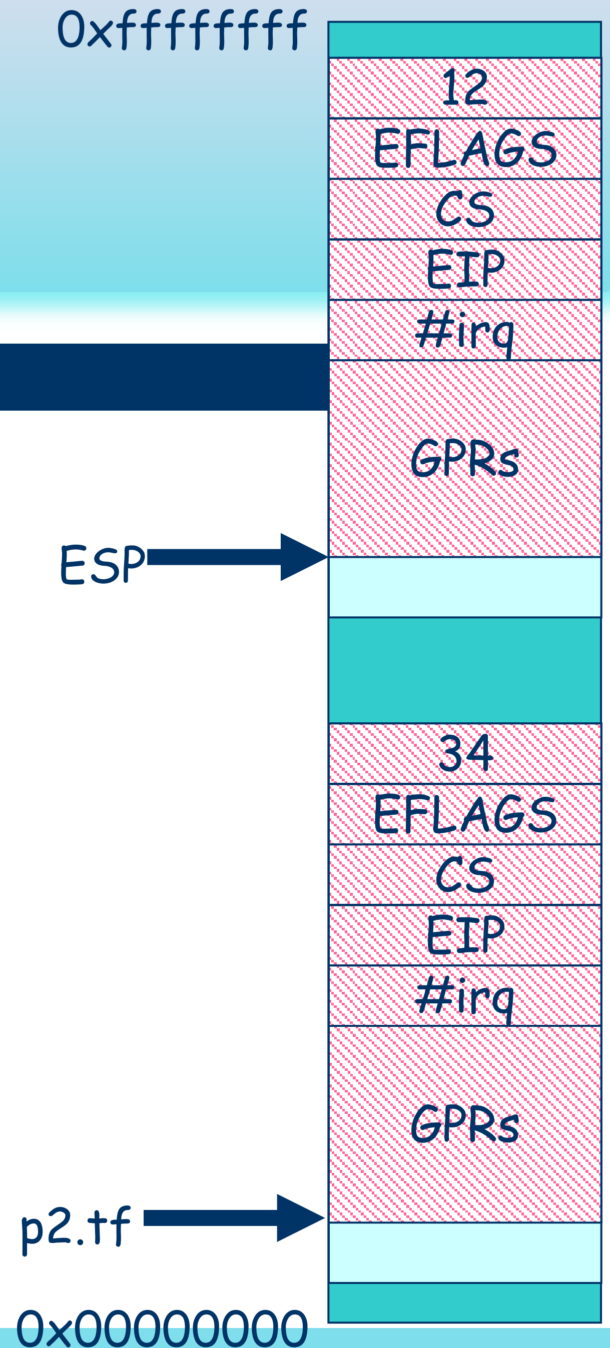
Context switch (cont.)

- process #1 is running…
- time's up



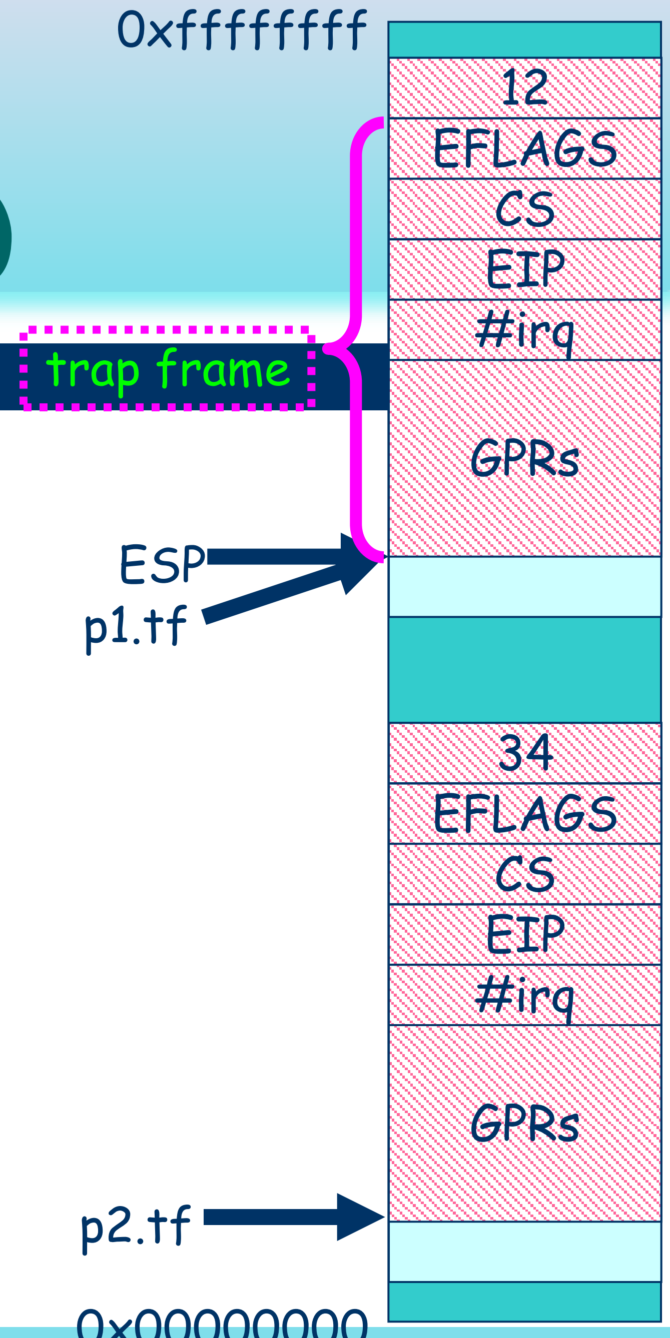
Context switch (cont.)

- process #1 is running...
- time's up
- save current state



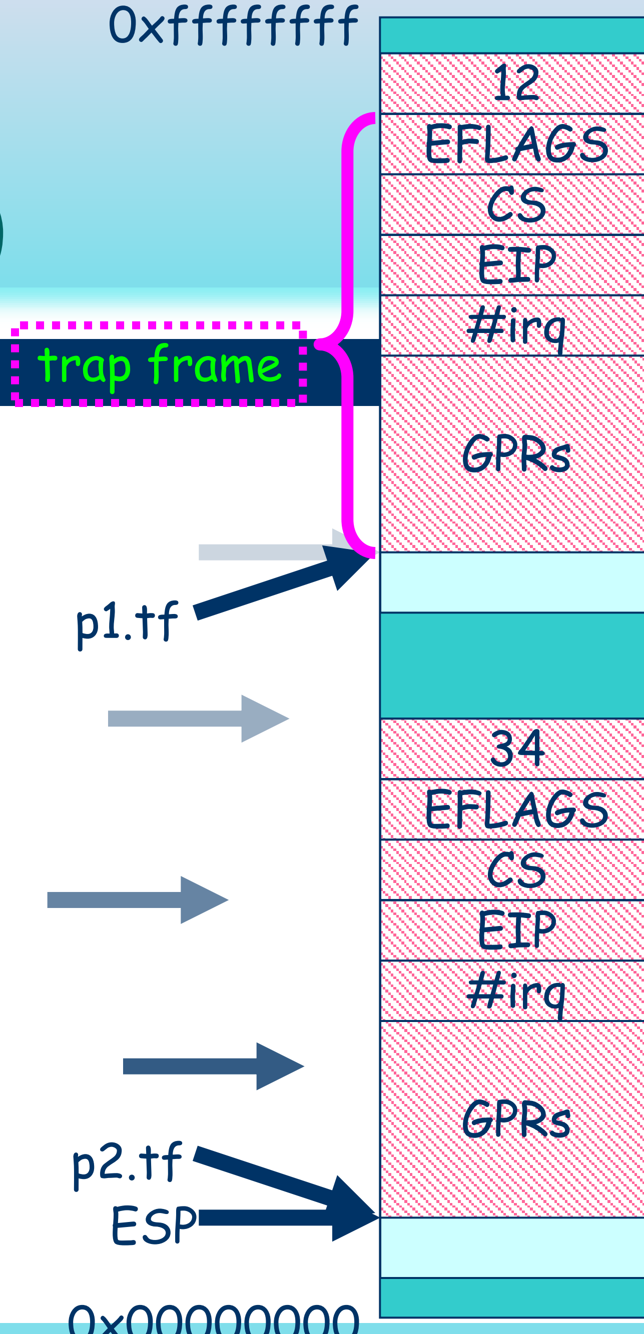
Context switch (cont.)

- process #1 is running...
- time's up
- save current state
- save the top of #1's stack



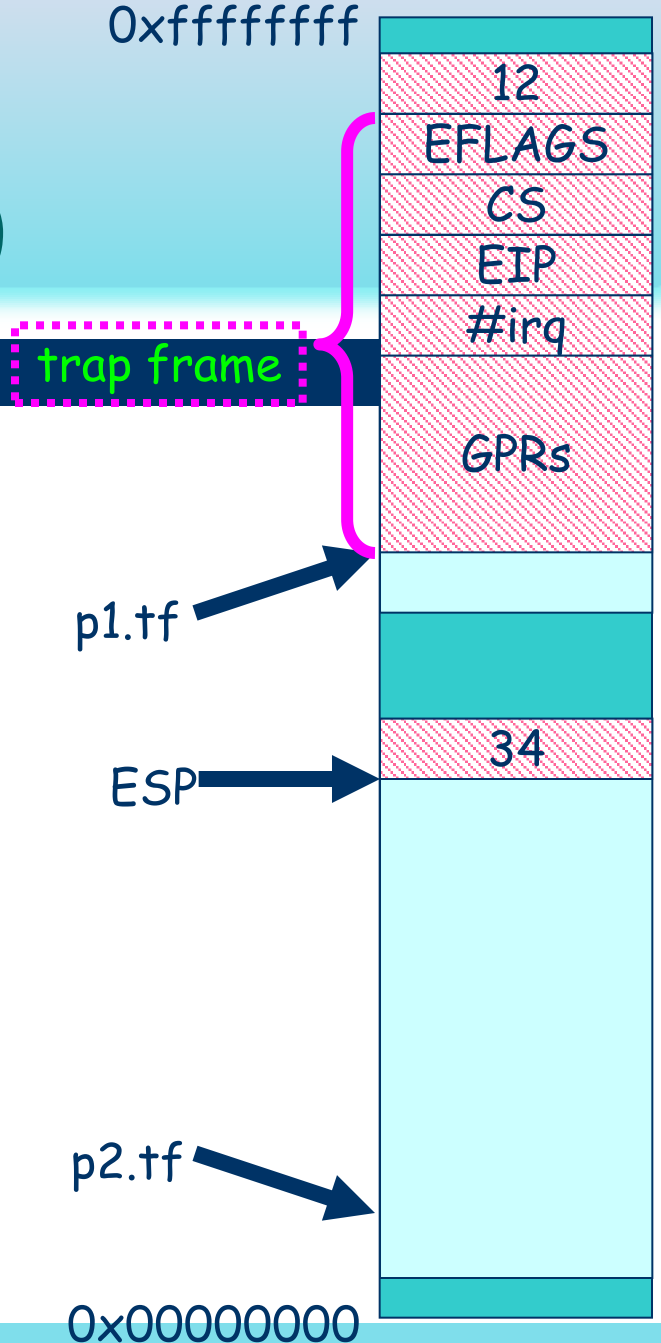
Context switch (cont.)

- process #1 is running...
- time's up
- save current state
- save the top of #1's stack
- switch %ESP to process #2



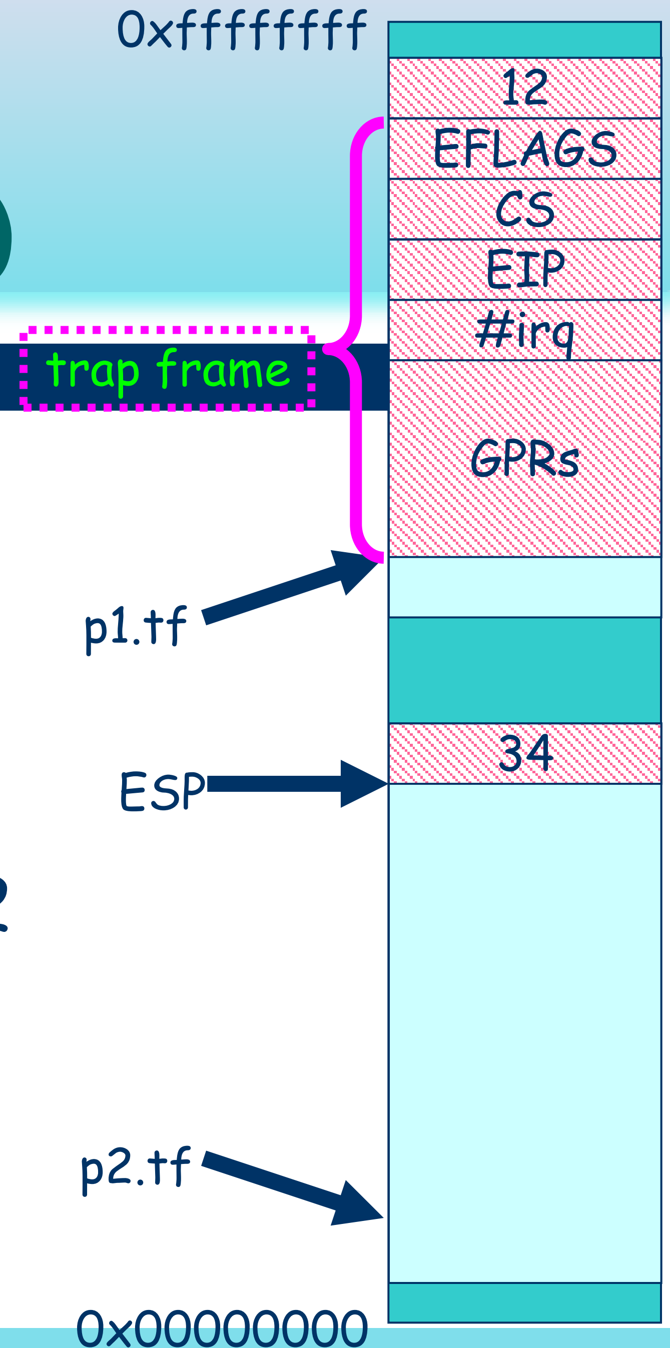
Context switch (cont.)

- process #1 is running...
- time's up
- save current state
- save the top of #1's stack
- switch %ESP to process #2
- restore current state



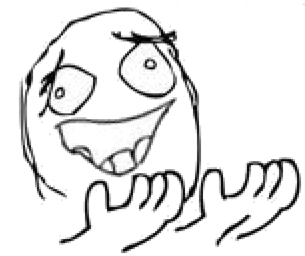
Context switch (cont.)

- process #1 is running...
- time's up
- save current state
- save the top of #1's stack
- switch %ESP to process #2
- restore current state
- process #2 is running...



Implementation Hint 1

- How many of you have
  - triggered unexpected exceptions?
  - got stuck in mysterious reboot?
  - suffered from qemu crashing?
- Hint: The machine is always right!!!
  - How to find the victim instruction?



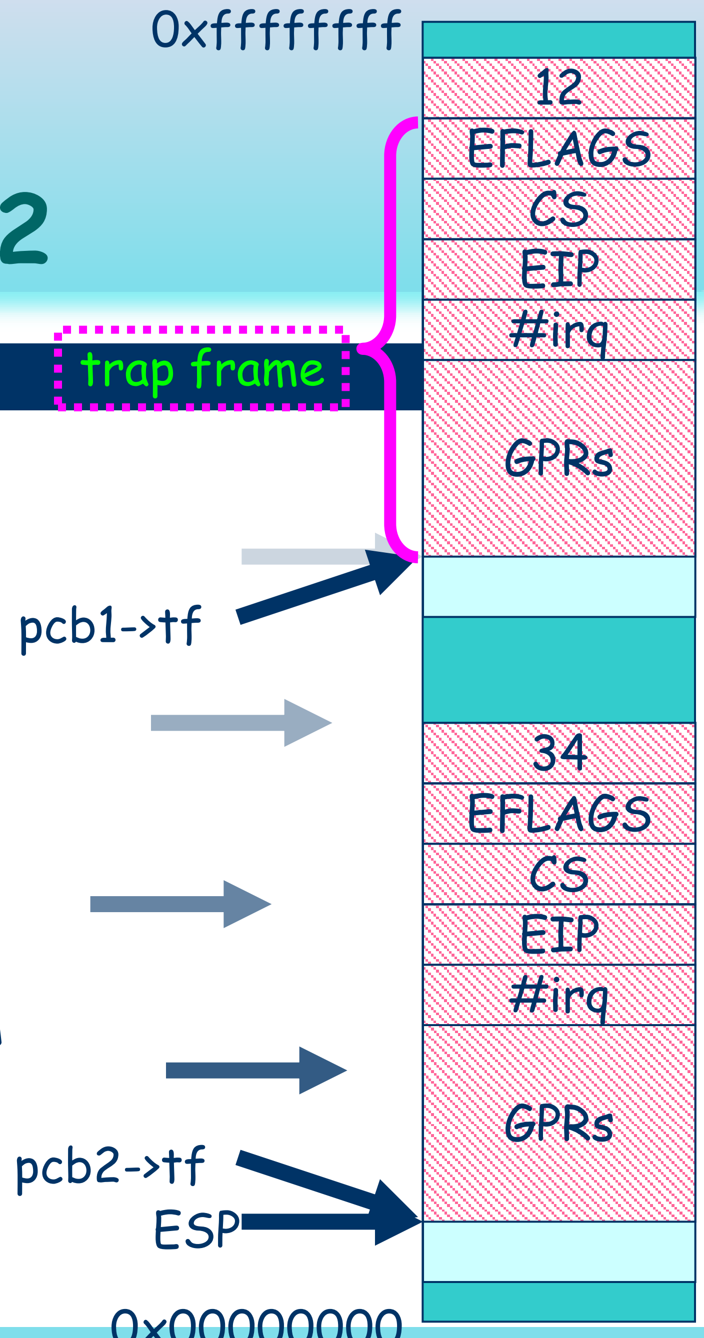
```
system panic in function "irq_handle", line 49, file "src/kernel/irq/irq_handle.
c":
Unexpected exception #14
Hint: The machine is always right! For more details about exception #14, see
i386 Manual
```

Implementation Hint 2

- initialize "current state"

```
TrapFrame *tf = (TrapFrame *)(pcb->
          kstack + KSTACK_SIZE) - 1;
tf->eflags = ...;
......
```

- What is type casting from the view of machine?



Implementation Hint 3

- KISS principle
  - Keep It Simple, Stupid
- Example: first schedule policy

```
current = (current == pcb_A ? pcb_B : pcb_A);
```

- Why KISS?

KISS or not KISS

如果觉得这种哲学描述太抽象的话，原文中有一个关于Unix中断处理的例子，非常生动。一位MIT的教授一直困恼于Syscall处理时间过长出现中断时如何保护用户进程某些状态，从而让用户进程能继续执行。他问新泽西人，Unix是怎么处理这个问题。新泽西人说，Unix只支持大多数Syscall处理时间较短的情况，如果时间太长出现中断Syscall不能完成，那就会返回一个错误码，让用户重新调用Syscall。但MIT人不喜欢这个解决方案，因为这不是“正确的做法”。

施教授讲了他博士后期间的一个故事。一次他的任务是纯化一个蛋白。两天下来，虽然纯化了，但是产量只有20%。

他不好意思地对导师说，“产率很低，我计划继续优化蛋白的纯化方法，提高产率”。

但导师反问：“你为什么想提高产率？已有的蛋白不够你做初步的结晶实验吗？”

他回敬：“我有足够的蛋白做结晶筛选，但我需要优化产率以得到更多的蛋白。”

导师不客气地打断：“不对。产率够高了，你的时间比产率重要。请尽快开始结晶。”

实践最后证明他导师的建议是对的。

KISS in software engineering

- Build a prototype as soon as possible.
  - Making the system runnable is MORE IMPORTANT than any other issue.
    - elegant algorithms
    - full functions
    - efficiency
- This is one of the precepts of the Unix Philosophy.

KISS & Testing

- Break down the code into small units
  - according different functions
  - decoupling your code
- Perform unit tests as early as possible.
  - write code for function A -> test it ->
  - write code for function B -> test it ->
  - …
- unit test -> component test -> system test

KISS & Testing (cont.)

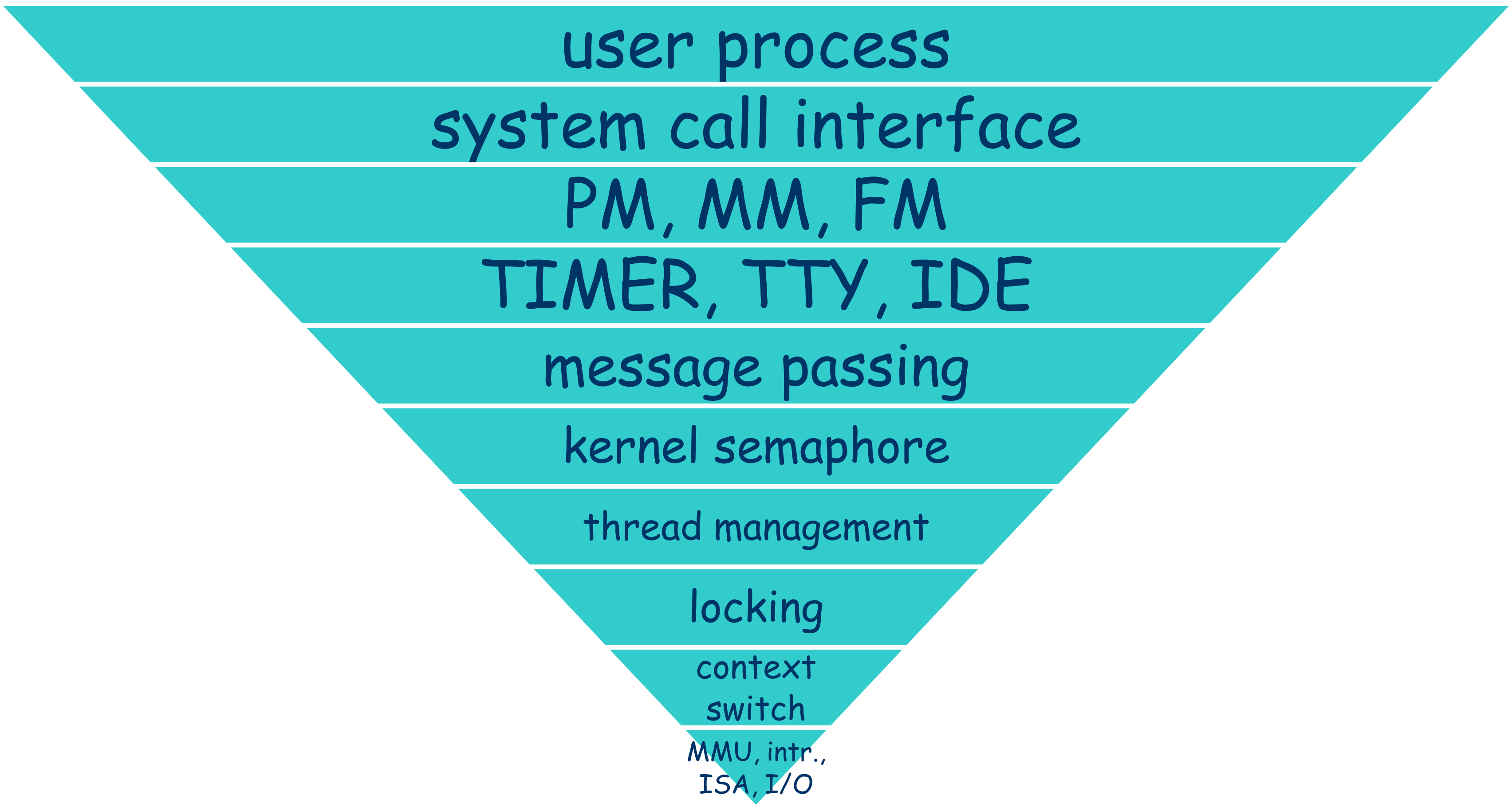
- Take Lab1 as an example
  - write code for stack switch -> test it ->
  - write code for creating kernel threads -> test it
    - PCBs are predefined.
    - take the simplest schedule policy
  - wirte code for allocating PCBs -> test it
  - write code for ready queue and round robin scheduling -> test it
  - write code for sleep() and wakeup() -> test it

KISS & Testing (cont.)

- If your code is very buggy, a large system test will not give you any useful information.
  - errors are everywhere
  - still need to break down the code into small units
  - usually more cost to debug
- Do NOT write a large piece of code without unit tests.
  - Build and test step by step.

- Thread management

The architecture of Nanos



Thread management

- thread creation
  - create a kernel thread
  - you have implemented it!
- thread sleep
  - block the current thread
- thread wake-up
  - resume a blocking thread
- NO thread destruction
  - kernel threads in Nanos (drivers, servers) will not terminate.

What do they mean?

- thread sleep = do not schedule this thread + do not execute this thread
- thread wake-up = make the thread ready to schedule

- We need something to distinguish ready threads and not-ready threads.
  - A state flag is easy to start with.
    - But you will find it inconvenient to code in the future.

Using list head

- List head is a data structure of cyclic doubly linked list.
  - efficient
  - flexible
  - easy to use and maintain
    - see include/adt/list.h

Using list head (cont.)

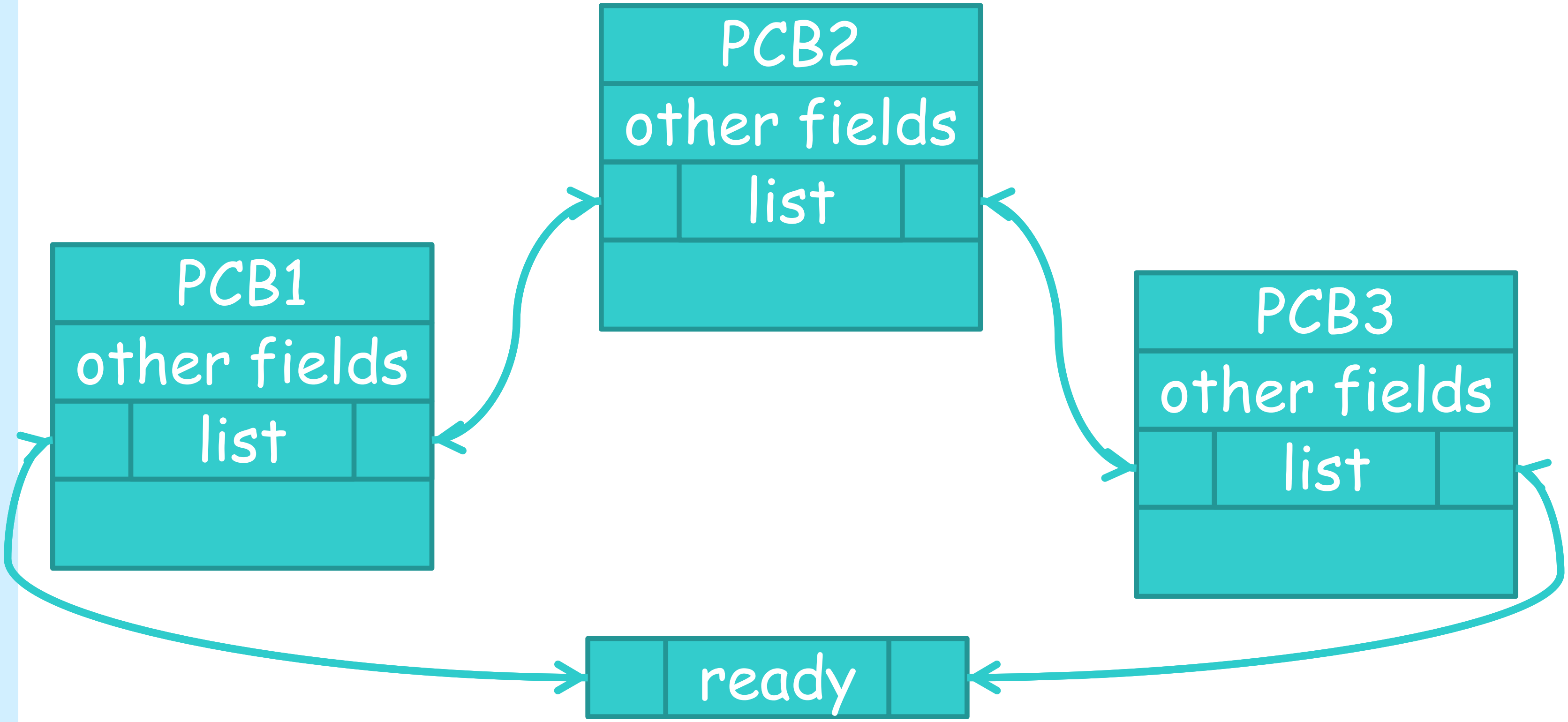
- single head

- embed list head in other structure

```
struct PCB {
        // other fields
        ListHead list;
};
```

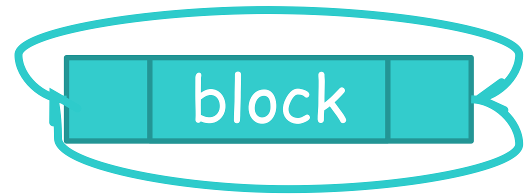
Using list head (cont.)

- linked list



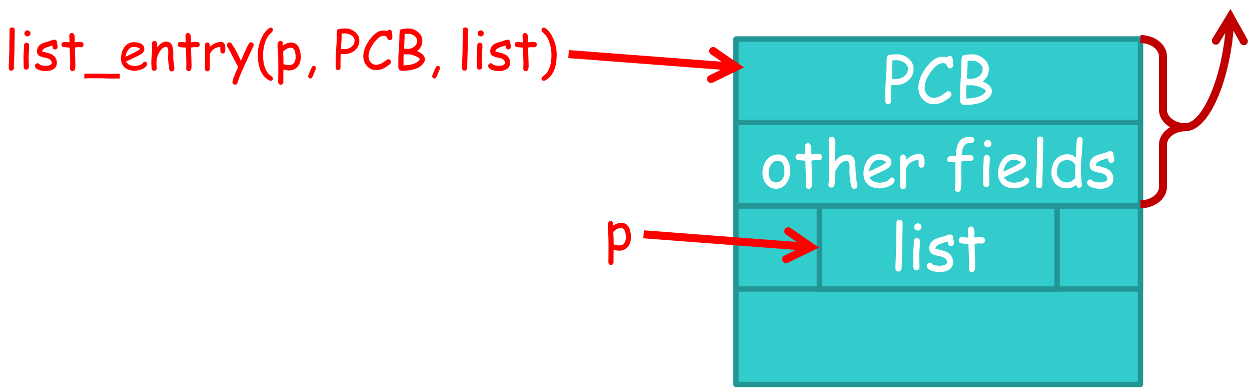
Using list head (cont.)

- empty linked list



- find the start address of the structure

```
#define list_entry(ptr, type, member) \
  ((type*)((char*)(ptr) - (int)(&((type*)0)->member)))
```

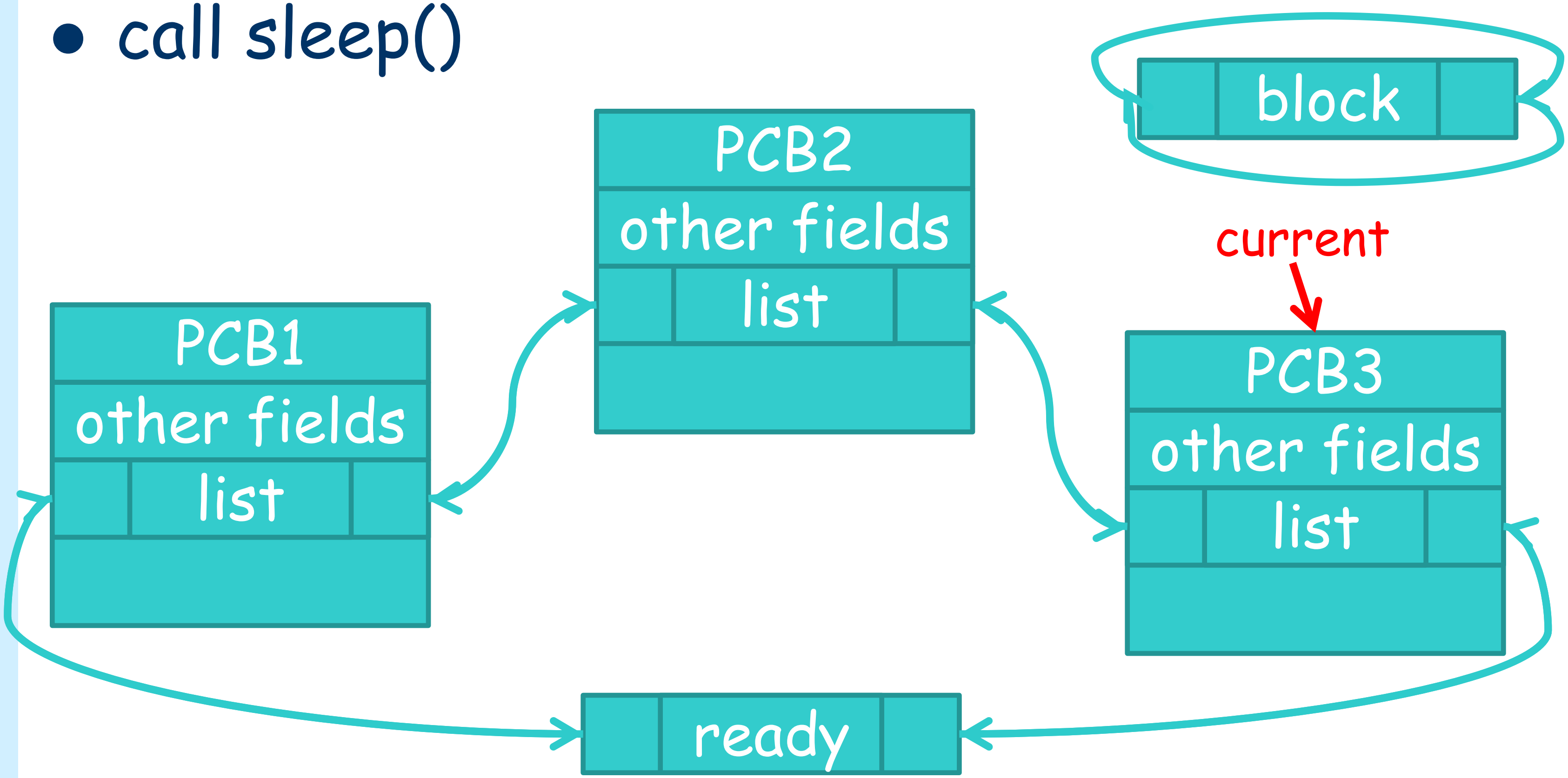


Thread sleep

- thread sleep = do not schedule this thread + do not execute this thread
- do not schedule this thread = put the thread into blocking queue
  - always choose ready threads to schedule
- do not execute this thread = ?

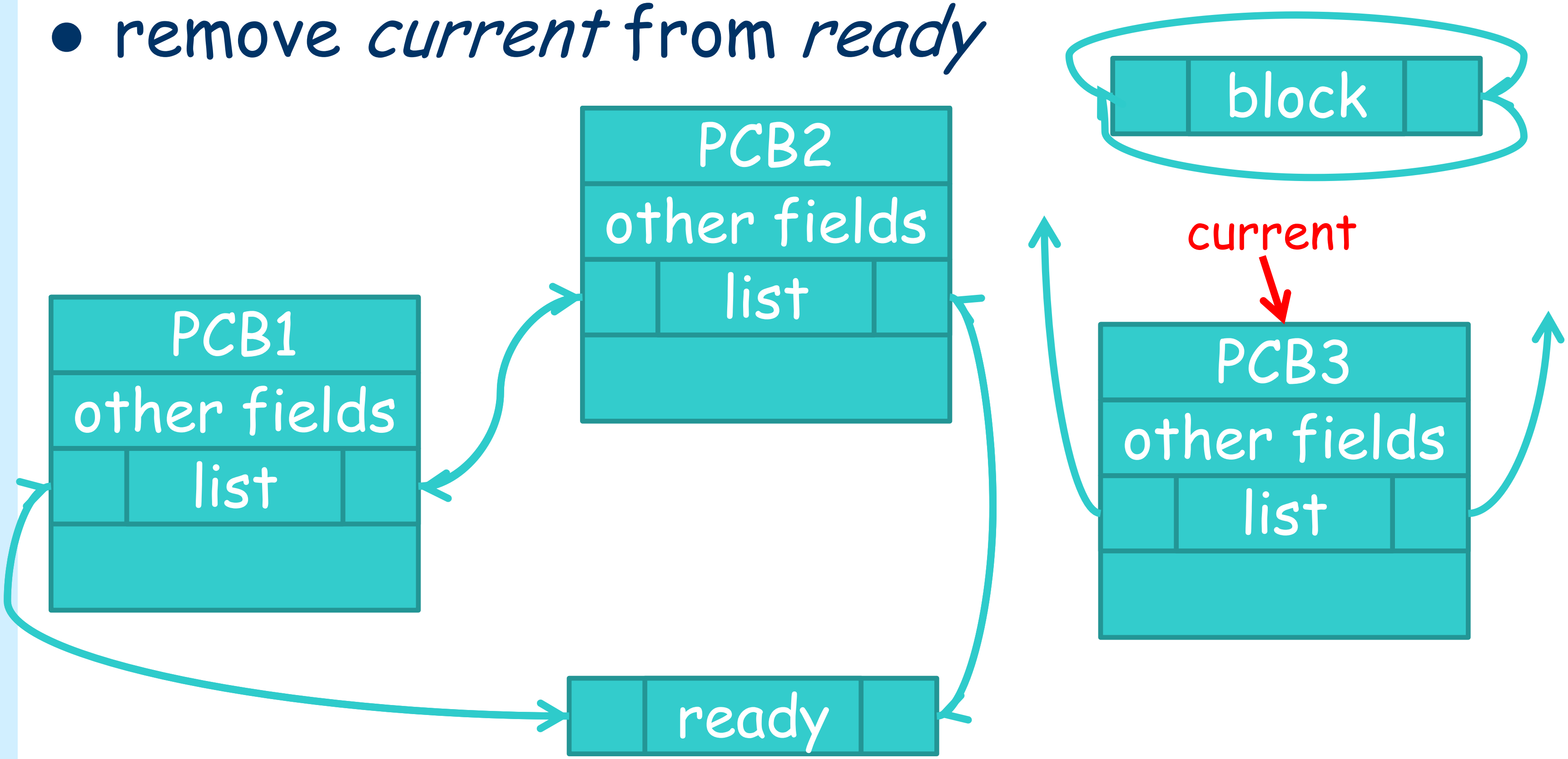
Implement thread sleep

- call sleep()



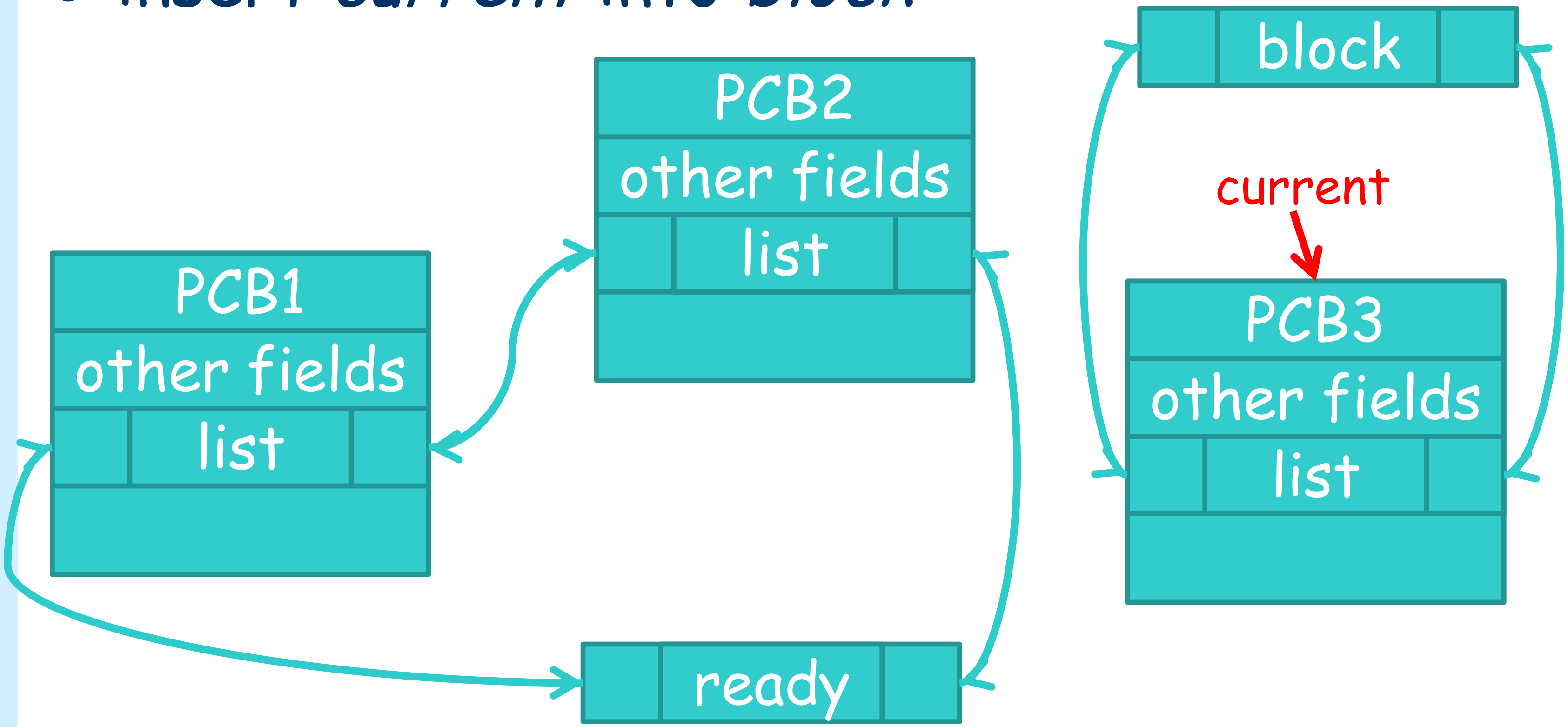
Implement thread sleep (cont.)

- remove *current* from *ready*



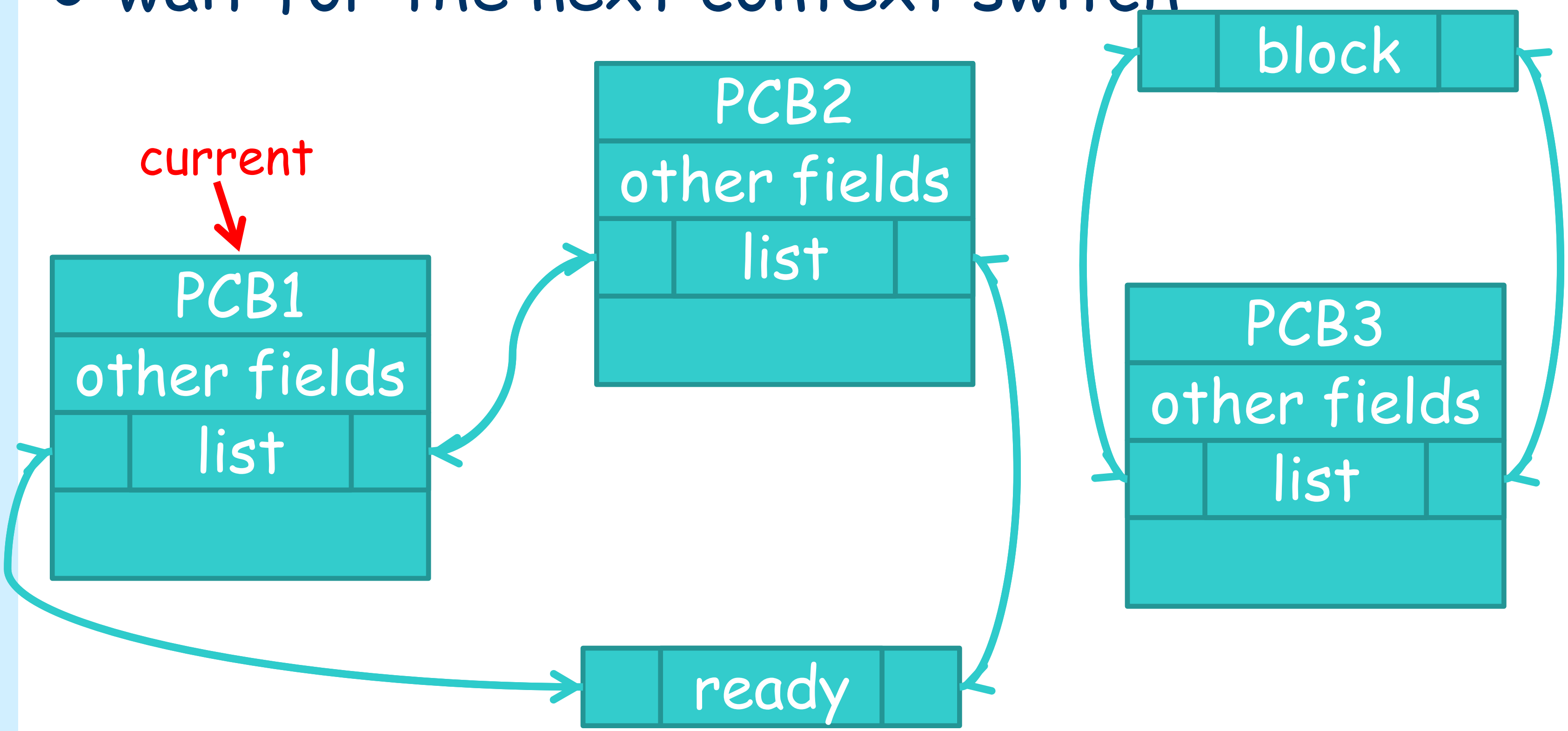
Implement thread sleep (cont.)

- insert *current* into *block*



Implement thread sleep (cont.)

- wait for the next context switch



Wait for context switch

- After insert the thread into blocking queue, the thread should not be executed any longer.
  - cannot let the thread return from sleep()
- Waiting for the next interrupt is OK.
  - interrupt comes = trigger context switch = schedule other threads

```
void sleep() {
// insert the current thread into blocking queue
        wait_intr();
};
```

```
sleep();
printk("a");
```

Any better idea?

- The next interrupt is unpredictable.
  - For a 100Hz timer, it costs 5ms on average to wait for the next timer interrupt.
  - But for a 2GHz CPU, this is a looooooooong time.
    - 10,000,000 CPU cycles!!!

- Can we trigger context switch manually?

Using exceptions

- We can manually trigger an exception to make the CPU jump to the interrupt processing code.
  - volatile int x = *(int *)0x87654321;
- Any safe exception?
  - asm volatile ("int $14");
- Any safe exception number?
  - asm volatile ("int $0x80");

System call

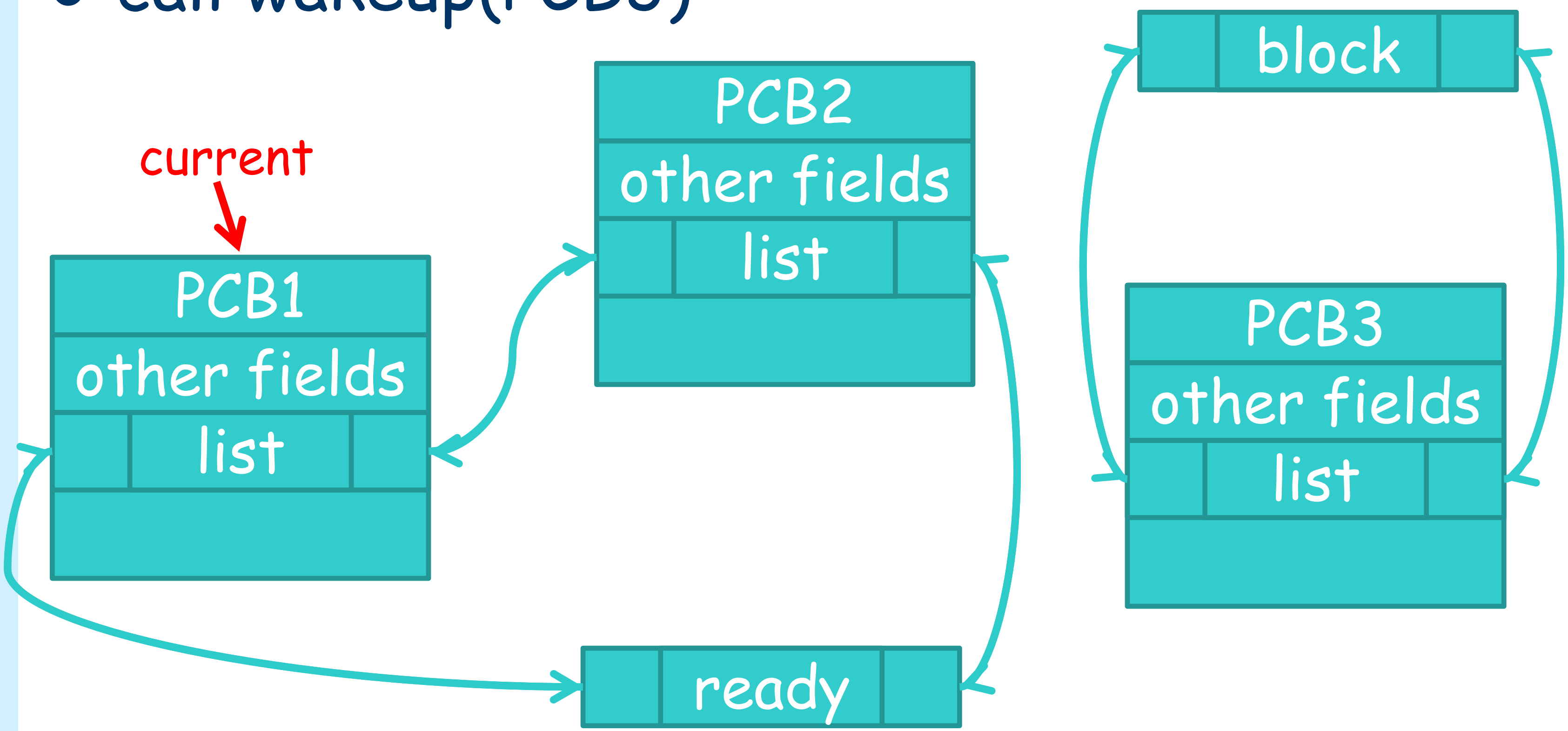
- asm volatile ("int $0x80");
- This is exactly the trap instruction used by user processes to request service from OS.
- But now there is no service avaliable.
  - We use system call here as a safe exception to perform context switch immediately.

Thread wake-up

- thread wake-up = make the thread ready to schedule
- It is easy and straightforward.
  - just put the blocking thread into ready queue

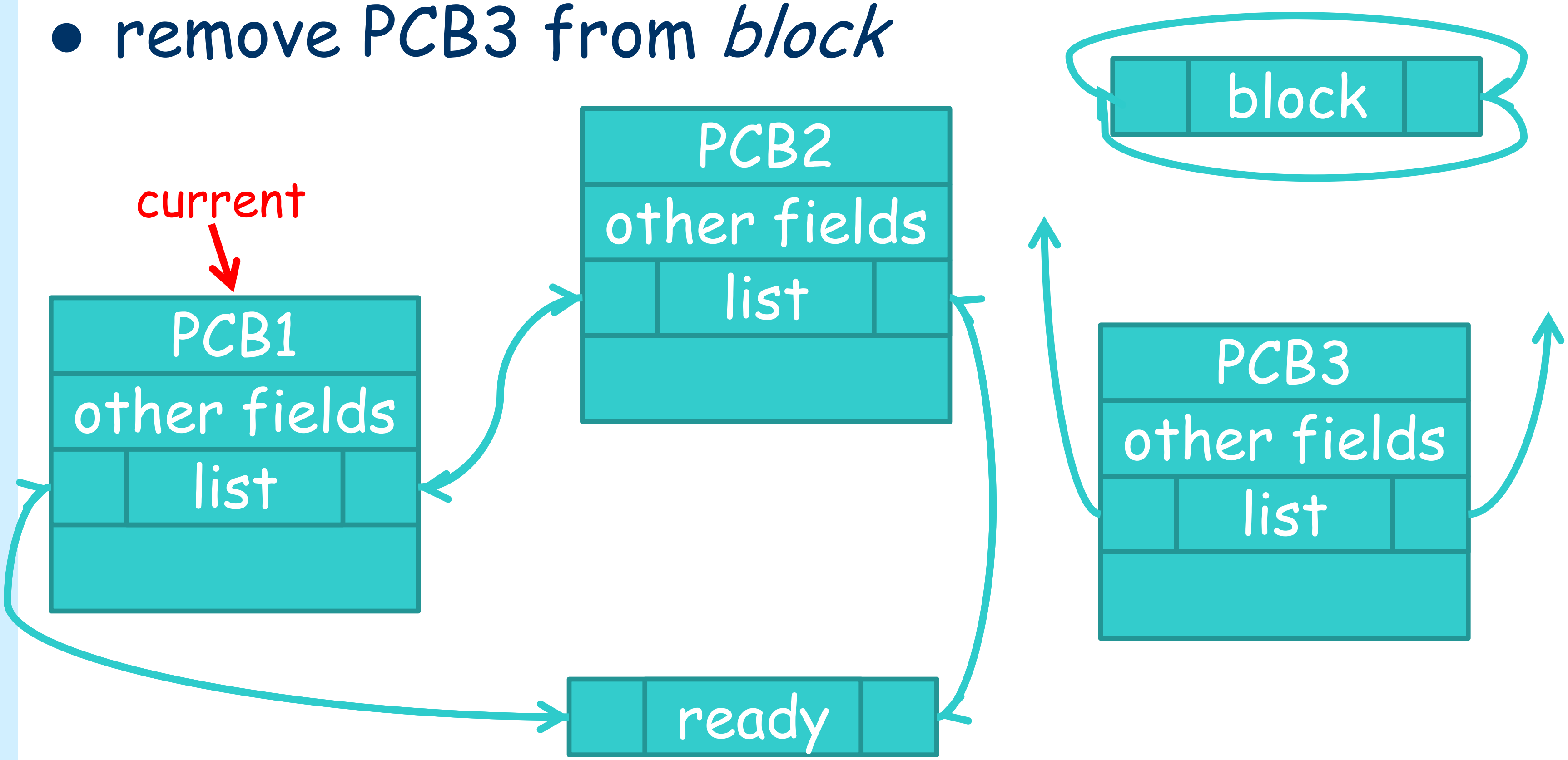
Implement thread wake-up

- call wakeup(PCB3)



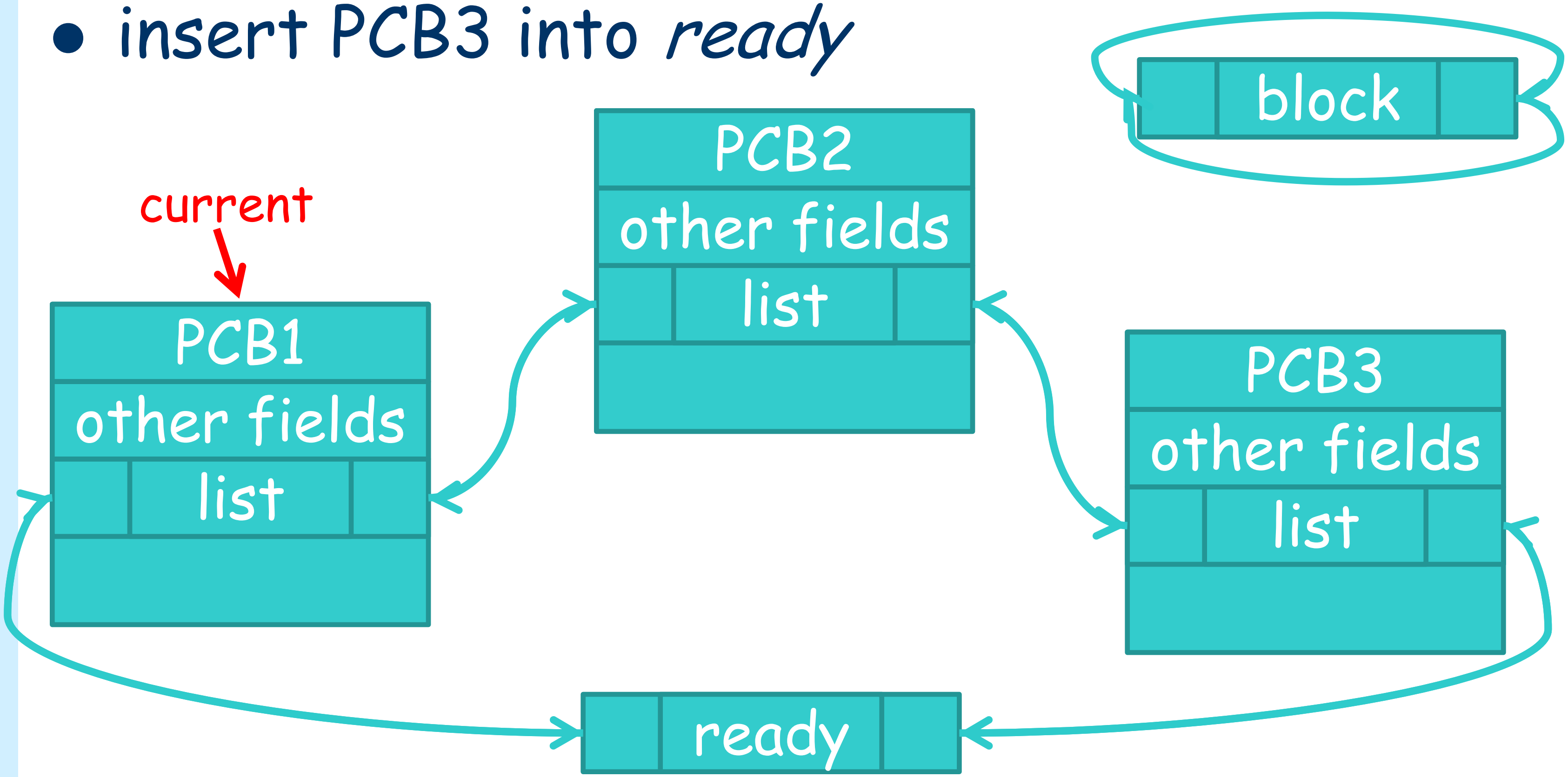
Implement thread wake-up (cont.)

- remove PCB3 from *block*



Implement thread wake-up (cont.)

- insert PCB3 into *ready*



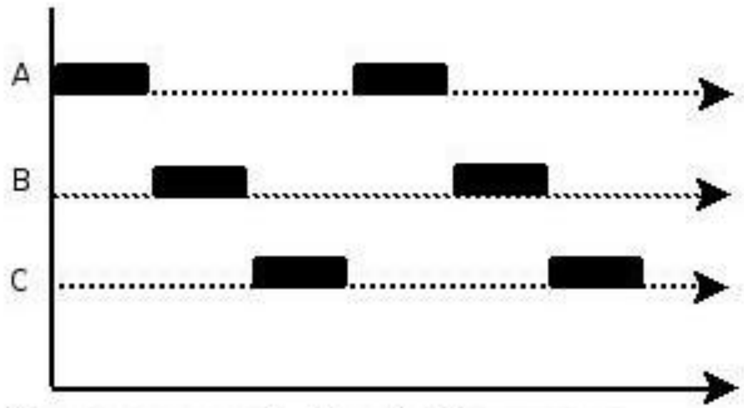
Summary

- thread sleep
  - put the current thread into blocking queue
  - use system call (trap instruction) to trigger context switch
- thread wake-up
  - put the blocking thread into ready queue
- context switch= interrupt & exception driven stack switch
- New problems are arisen.
  - This is the topic of next week.

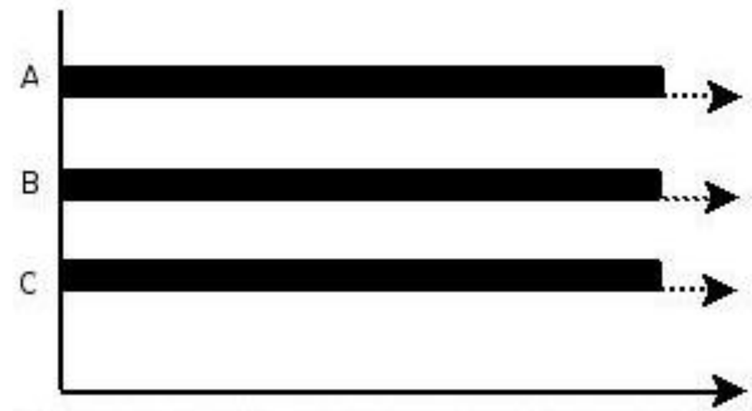
- Parallel programming with threads

Concurrency and Parallelism

- Parallelism needs hardware support.



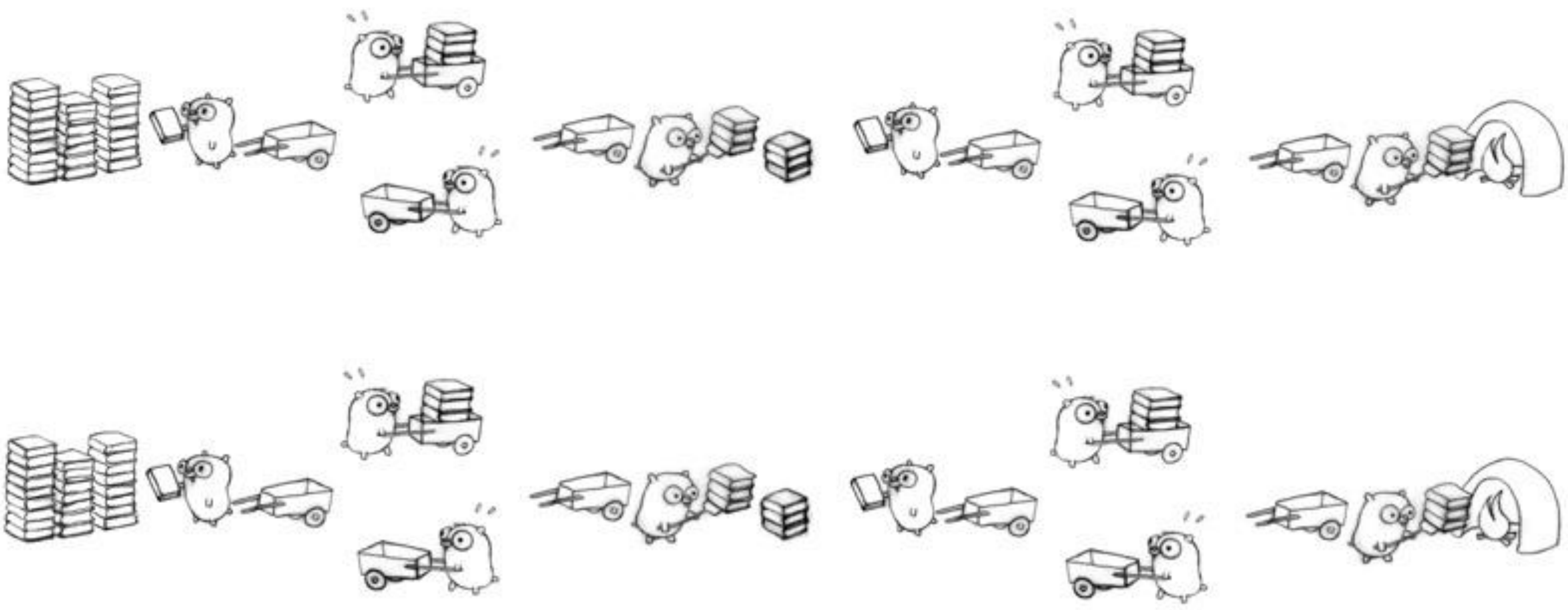
Concurrency : 1. Single Processor
2. logically simultaneous processing



Parallelism : 1. Multiprocessores, Multicore
2. Physically simultaneous processing

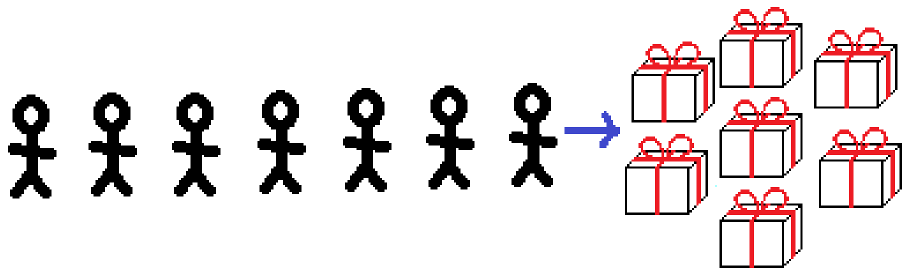
Concurrency and Parallelism (cont.)

http://concur.rspace.googlecode.com/hg/talk/concur.html

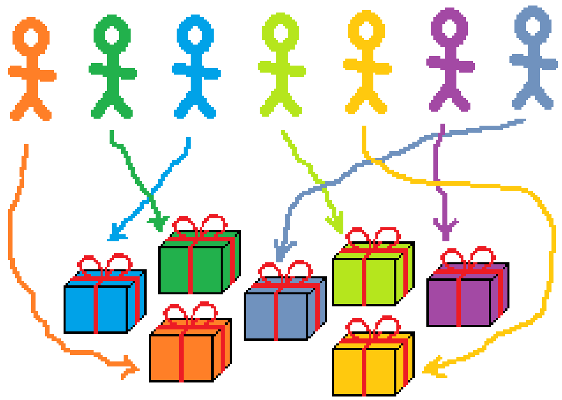


Concurrency and Parallelism (cont.)

http://www.yosefk.com/blog/parallelism-and-concurrency-need-different-tools.html



Concurrency: 7 kids queueing for presents from 1 heap



Parallelism: 7 kids getting 7 labeled presents, no queue

Vector addition

- using one thread

```
#define N 10000000
int a[N], b[N], c[N];
void add() {
        int i;
        for(i = 0; i < N; i ++) {
                c[i] = a[i] + b[i];
        }
}
```

Vector addition (cont.)

- using k threads

```
#define N 10000000
#define NR_THREAD k
#define NUM_PER_THREAD   (N / NR_THREAD)
int a[N], b[N], c[N];
void add(int tid) {
        int i, start = NUM_PER_THREAD * tid;
        for(i = 0; i < NUM_PER_THREAD; i ++) {
                c[start + i] = a[start + i] + b[start + i];
        }
}
```

How to create threads with arguments?

- How to implement create_kthread() to pass arguments to kernel threads?

```
for(i = 0; i < NR_THREAD; i ++) {
        create_kthread(add, i);
}
```

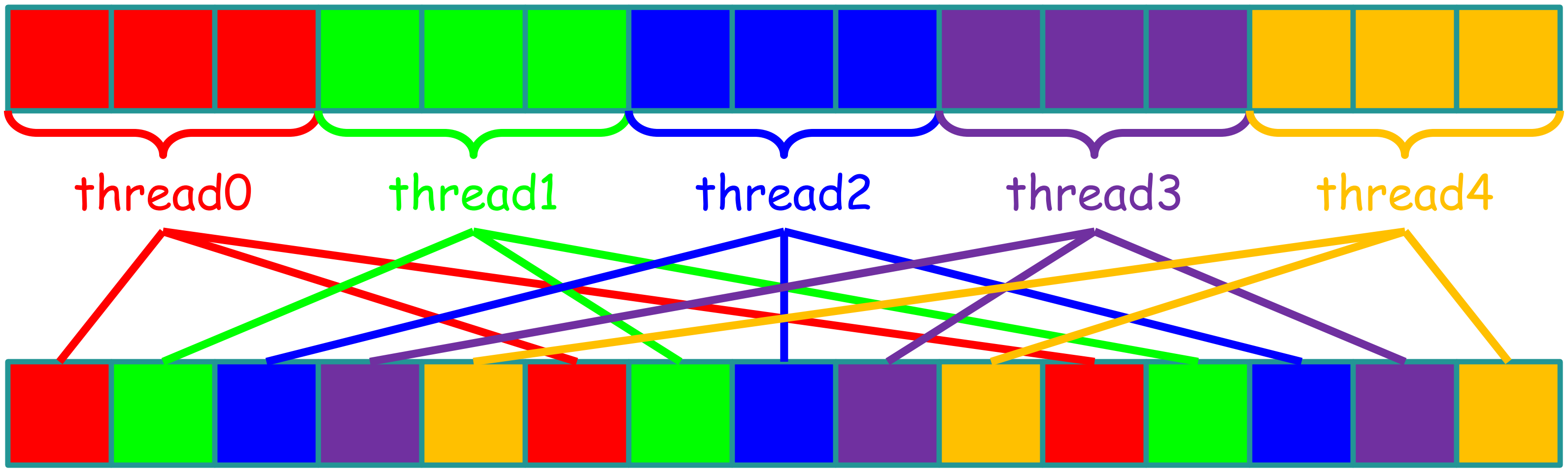
Which is faster?

```
void add(int tid) {
        int i, start = NUM_PER_THREAD * tid;
        for(i = 0; i < NUM_PER_THREAD; i ++) {
                c[start + i] = a[start + i] + b[start + i];
        }
}
```

```
void add(int tid) {
        int i, index = tid;
        for(i = 0; i < NUM_PER_THREAD; i ++) {
                c[index] = a[index] + b[index];
                index += NR_THREAD;
        }
}
```

Which is faster? (cont.)

- spatial locality
- DRAM burst
- memory coalescing



More about parallel programming

- Can you write a parallel program to perform matrix multiplication?
  - Can you make it faster?
    - tile algorithms
- Heterogeneous Parallel Programming
  - parallelize with different architecture
  - https://www.coursera.org/course/hetero
- Our world is parallel, but why it is difficult to parallelize the programms?

Lab1 is completely out!

- the second stage
  - implement sleep() and wakeup()
  - create 4 kernel thread to print "abcdabcd…"
- the third stage
  - answer 10 qeustions
  - some of them are VERY difficult
  - prepare yourself
- Have fun!