Lab2 feedback

- Lab2 feedback
- A guide to create the first process

- Lab2 feedback

Disable interrupt

- Disable what? (P.154, i386 manual)
- Exception is unmaskable.

The IF (interrupt-enable flag) controls the acceptance of external interrupts signalled via the INTR pin. When IF=0, INTR interrupts are inhibited; when IF=1, INTR interrupts are enabled. As with the other flag bits, the processor clears IF in response to a RESET signal. The instructions CLI and STI alter the setting of IF.

CLI (Clear Interrupt-Enable Flag) and STI (Set Interrupt-Enable Flag) explicitly alter IF (bit 9 in the flag register). These instructions may be executed only if CPL ≤ IOPL. A protection exception occurs if they are executed when CPL > IOPL.

The IF is also affected implicitly by the following operations:

- The instruction PUSHF stores all flags, including IF, in the stack where they can be examined.

Message passing

- protecting critical regions in send() & receive()
  - use lock() & unlock()
  - CANNOT use P(mutex) & V(mutex)
- blocking in receive()
  - use P(nr_msg)
- receive an unwanted message
  - A: receive(B, &m);
  - C: send(A, &m);
  - P() again until the wanted message comes
    - pay attention to the consistency between nr_msg.token and the number of message in the queue

Device drivers

- Top half may send messages in interrupt context.
  - unlock() should NOT enable interrupts.
- Use assertion to check your implementation for locking.

Interrupt and critical region (1)

危机四伏的中断

中断不仅是引发不同线程之间数据冒险的罪魁祸首，就连不同的中断之间也可能会引发数据冒险，更极端的，相同的中断也可能会出现临界区问题，真是万恶的根源啊！不过你能想明白这是什么原因吗?

为了避免这些问题的出现，Nanos做了简化：所有外部中断都是不可嵌套的．这样，你就不需要对中断处理过程中的临界区进行额外的保护．

- TTY registers two top halves.
  - send_keymsg() & send_updatemsg()
- Both of them will send message to TTY.
- There will be critical regions if interrupt is enabled during interrupt context.

Semaphore with negetive tokens

负数的信号量
回顾我们在课堂上讨论的信号量，token的值都是非负的．有一种信号量的token可以取负数，你知道这种信号量有什么好处吗?

- When token < 0, it records the number of waiting process.
- Nothing else special.

Interrupt and critical region (2)

危机四伏的中断(2)(这个问题有难度)
我们不应该在中断处理过程中使用P操作，否则可能会引起致命的问题．你能想明白这个问题吗?

- This is a tricky question.

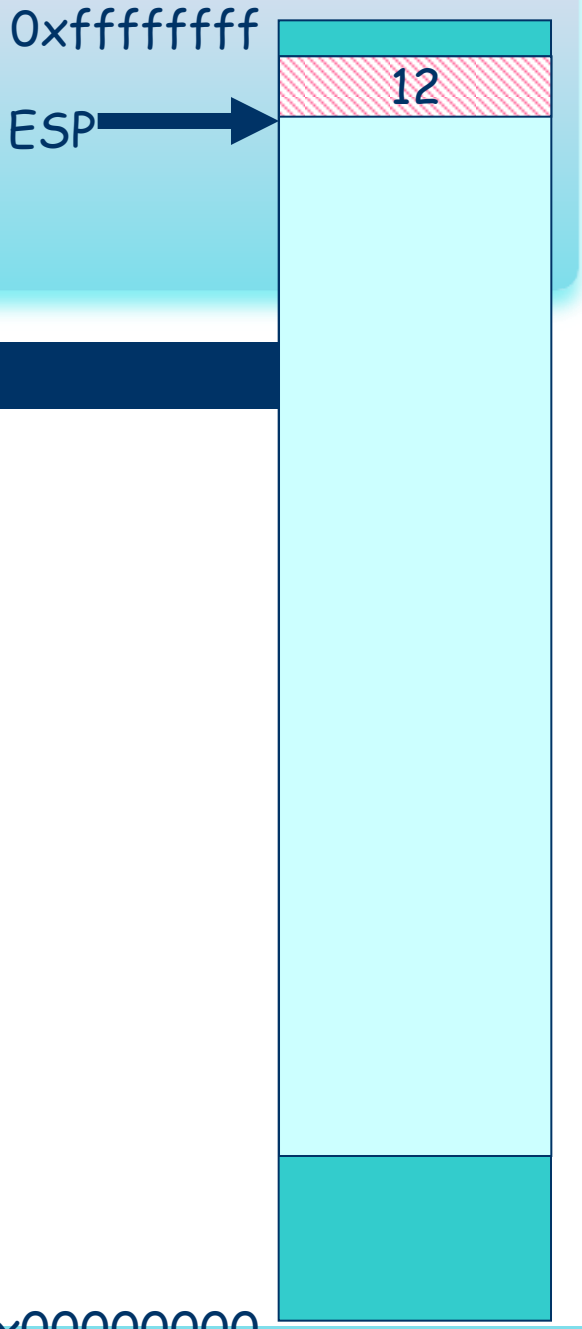
Interrupt context

- Interrupt context is not associated with any process.
  - does not have its own resource
- When interrupt comes, its handler will "borrow" resources from the interrupted process(let's call it A) to exetute.
  - save A's current state in asm_do_irq()
  - handler is executed with A's stack

Problems

- If the handler is blocked, so does A.
  - When the handler is waken up is unpredictable.
  - A cannot execute before the handler finishes.
  - What if A is a driver or server?
    - The system may be unresponsive.
- another problem:
  - The handler is not associated with any PCB.
  - How to wake up the handler?

The fatal problem



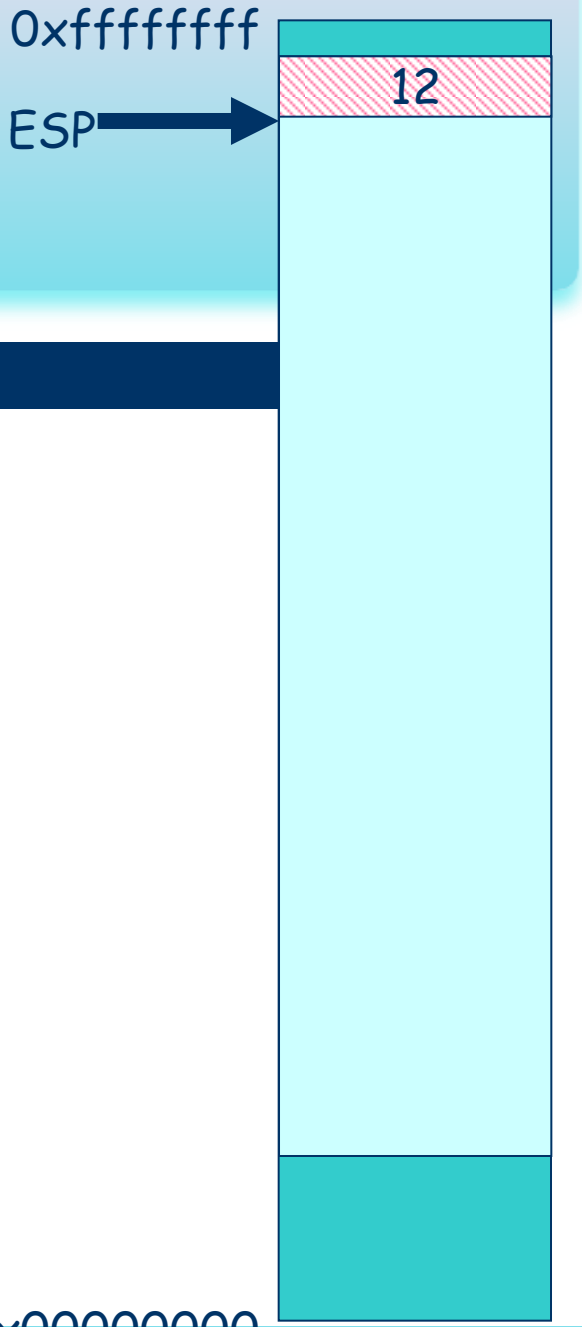

```
void ttyd() {
    // ...
    receive();
    // ...
}
```

```
void receive() {
    P(mutex);
    // ...
    V(mutex);
}
```

```
void send_keymsg() {
    // ...
    send(TTY);
    // ...
}
```

```
void send() {
    P(mutex);
    // ...
    V(mutex);
}
```

The fatal problem



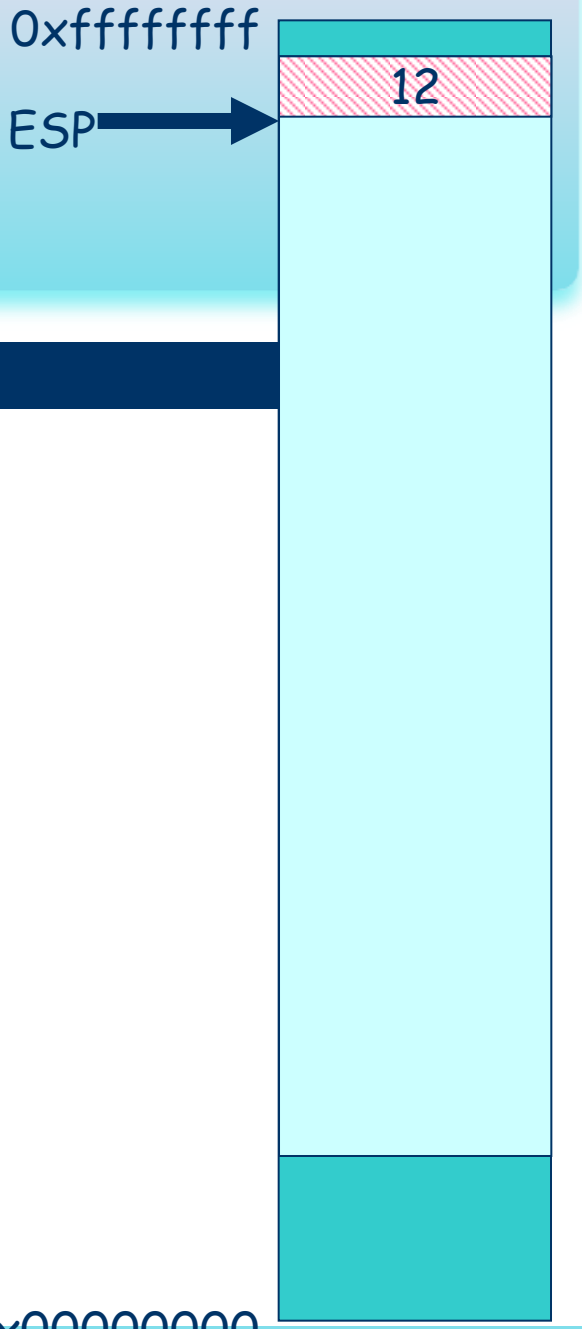

```
void ttyd() {
    // ...
    receive();
    // ...
}
```

```
void receive() {
    P(mutex);
    // ...
    V(mutex);
}
```

```
void send_keymsg() {
    // ...
    send(TTY);
    // ...
}
```

```
void send() {
    P(mutex);
    // ...
    V(mutex);
}
```

The fatal problem



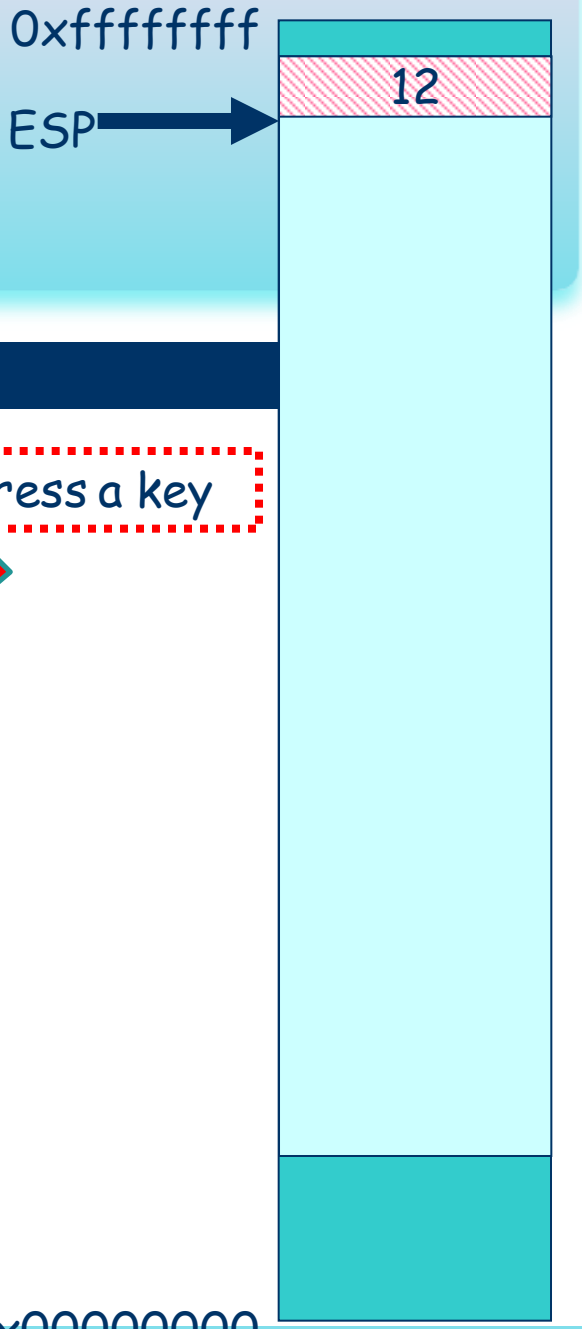

```
void ttyd() {
    // ...
    receive();
    // ...
}
```

```
void receive() {
    P(mutex);
    // ...
    V(mutex);
}
```

```
void send_keymsg() {
    // ...
    send(TTY);
    // ...
}
```

```
void send() {
    P(mutex);
    // ...
    V(mutex);
}
```

The fatal problem




```
void ttyd() {
    // ...
    receive();
    // ...
}
```

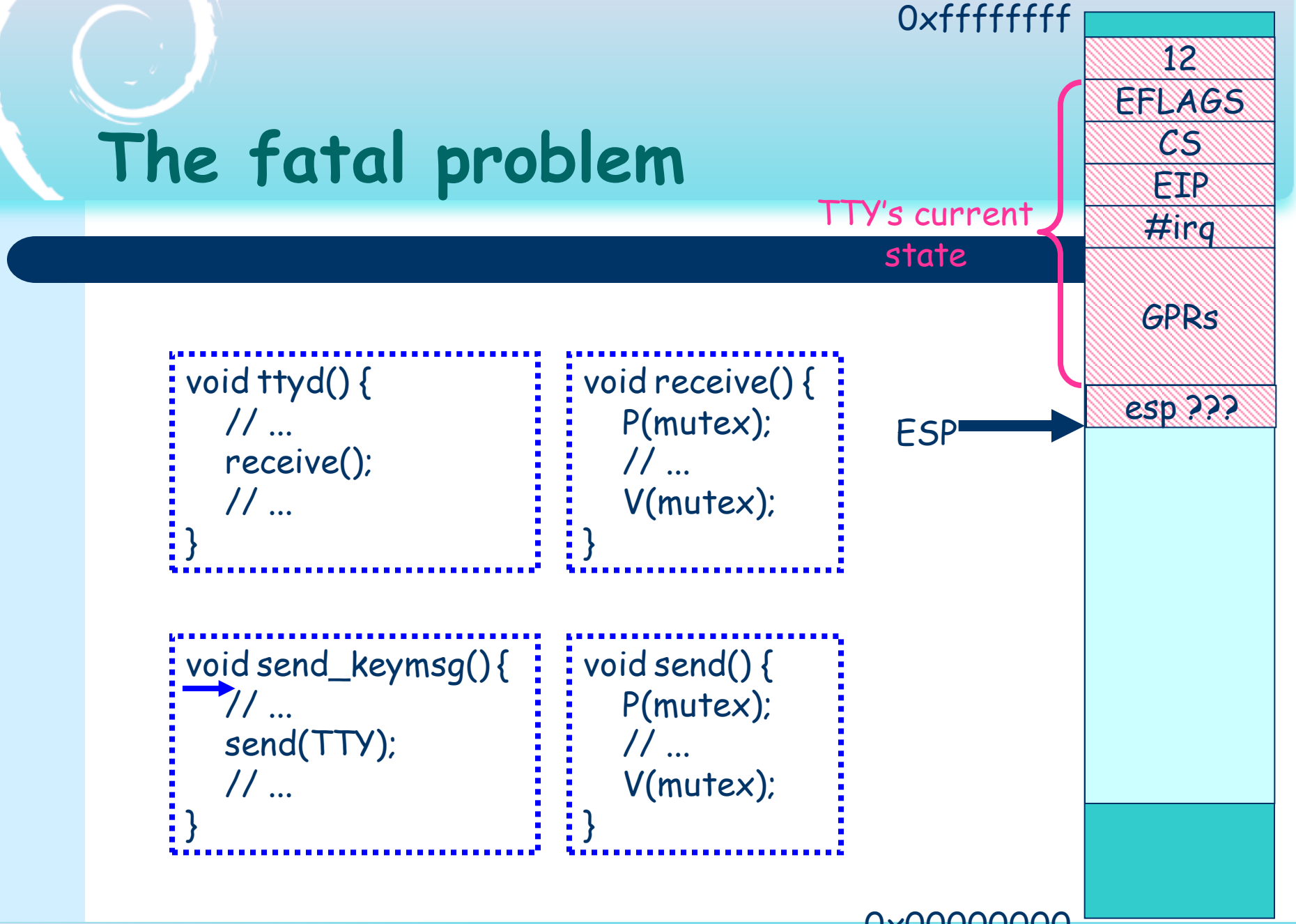
```
void receive() {
    P(mutex);
    // ...
    V(mutex);
}
```

press a key

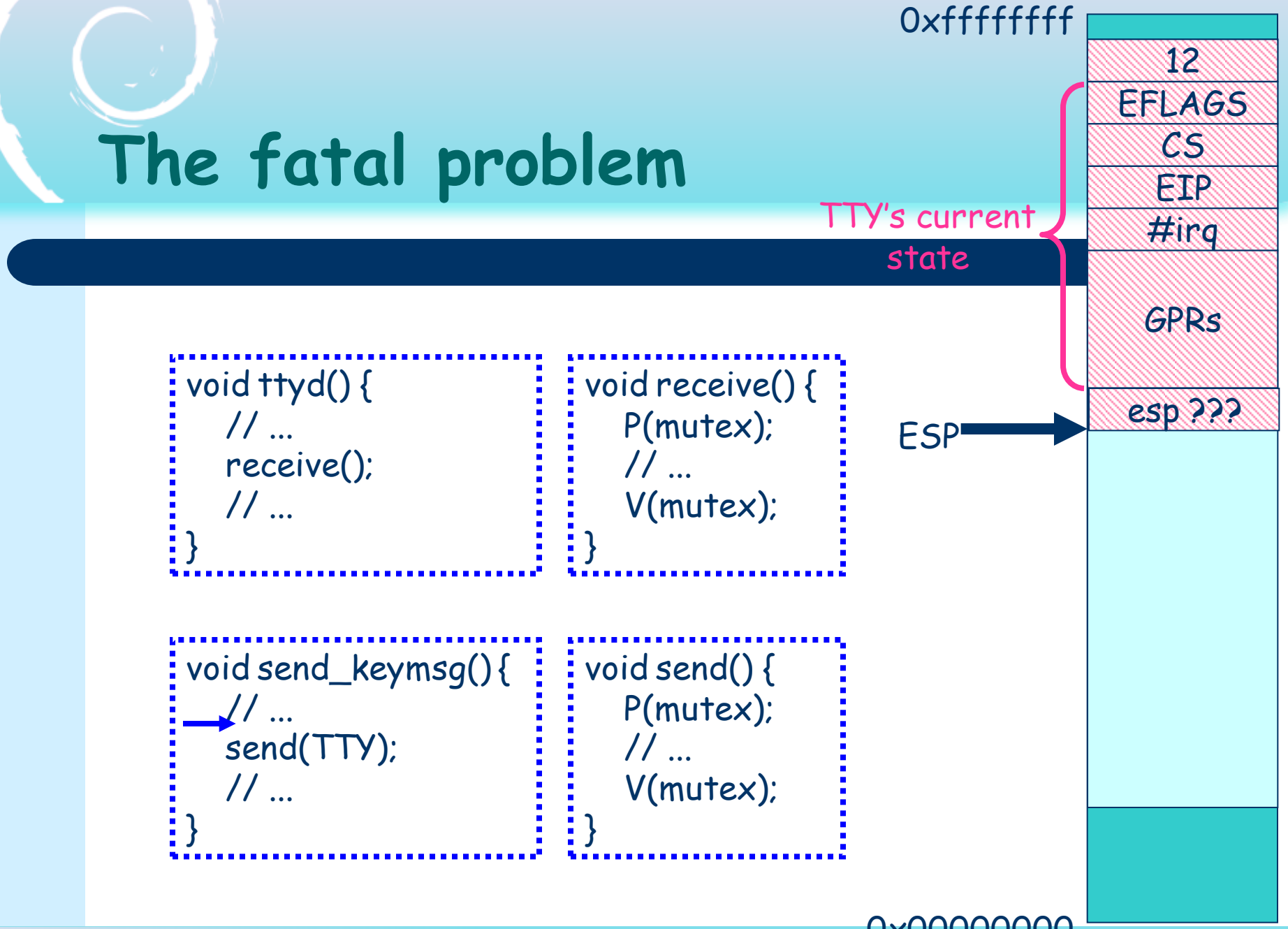
```
void send_keymsg() {
    // ...
    send(TTY);
    // ...
}
```

```
void send() {
    P(mutex);
    // ...
    V(mutex);
}
```

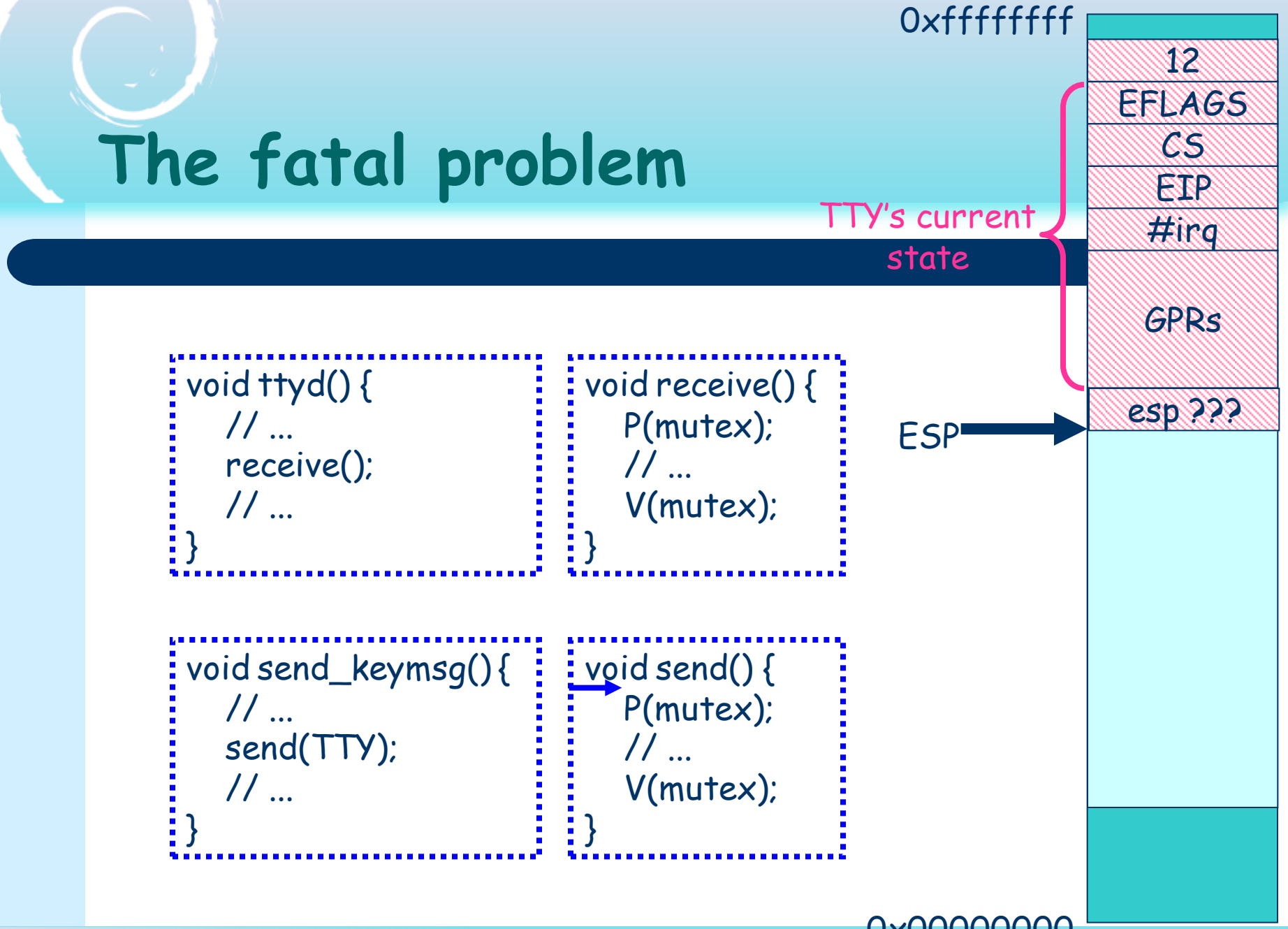
The fatal problem



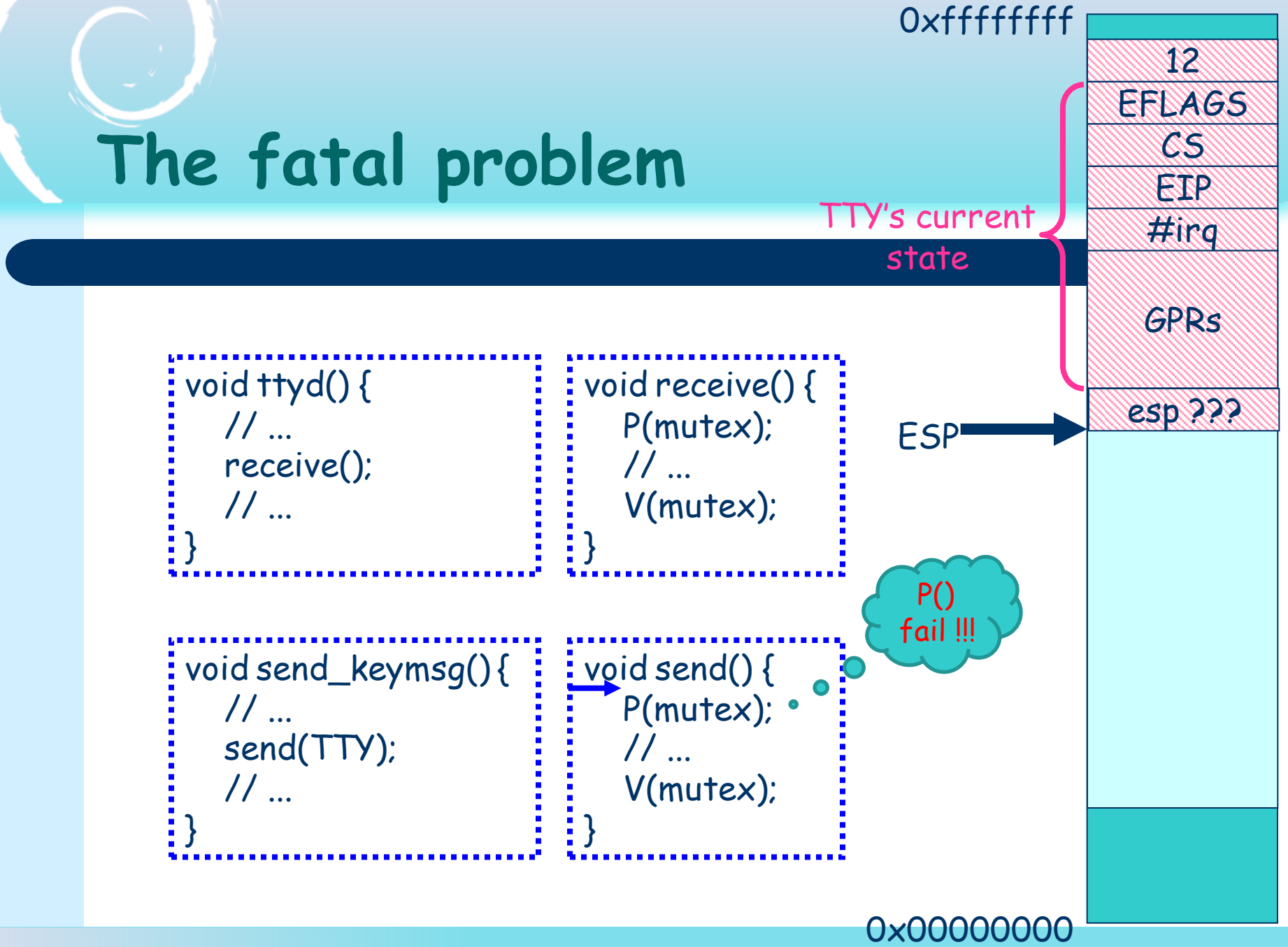
The fatal problem



The fatal problem

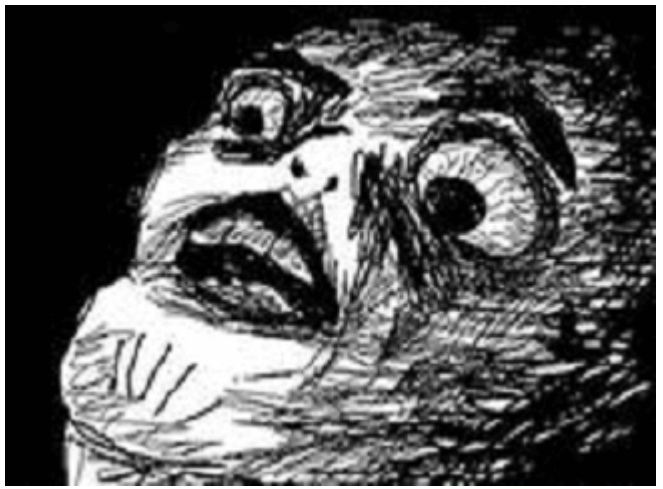


The fatal problem



The fatal problem

- Both TTY and the handler get stuck.
  - The handler sleeps until TTY releases the mutex.
  - TTY waits the handler to return from the interrupt context.
- Deadlock happens!!!



Solution

- Interrupt is unpredictable.
- So does A's state when interrupt comes.
  - process context
  - holding resources, especially those shared with interrupt handlers
- solution: NEVER sleep in interrupt context
  - http://stackoverflow.com/questions/1053572/why-kernel-code-thread-executing-in-interrupt-context-cannot-sleep/

Peterson algorithm

Peterson算法的正确性(这个问题有难度)

理论课上提到了Peterson算法是自旋锁的一种正确做法. 是的, 但这仅仅是算法层面的正确. 如果把书上的Peterson算法直接放到现代的多处理器系统上, 却有可能发生致命的错误! 你能想明白为什么吗?

嗯... 这个问题确实有难度, 给你一些提示吧: 你已经在组成原理课上学习到了cache, 流水线, VLIW, 超标量, 多发射.....

答案在这里 http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/

这个问题再次说明了一个道理: 很多理论上十分容易的问题, 真正实践起来会变得十分困难.

- memory ordering

Synchronous message passing

完全同步的消息机制

有的操作系统(例如Minix)提供的消息机制是完全同步的，发送方执行send()的时候，如果接收方没有阻塞在receive()上，发送方也会被阻塞，直到接收方执行receive()，才能完成消息的传递．这种做法的好处是不会发生消息队列溢出的情况，甚至不需要消息队列的存在！你能想明白为什么吗?但它的缺点是效率不高，而且很容易产生死锁．你能举出一个在半同步消息机制下可以正常运行，但在完全同步消息机制下产生死锁的例子吗?

- A: send(B)
- B: send(A)

random and urandom

random和urandom

细心的你会发现，我们在刚才的命令中使用的是urandom设备，而不是random. 如果你使用random，你可能不会马上看到输出. 你知道这是为什么吗? (Hint: man一下就有答案了)

- man 4 random

When read, the /dev/random device will only return random bytes within the estimated number of bits of noise in the entropy pool. /dev/random should be suitable for uses that need very high quality randomness such as one-time pad or key generation. When the entropy pool is empty, reads from /dev/random will block until additional environmental noise is gathered.

A read from the /dev/urandom device will not block waiting for more entropy. As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver. Knowledge of how to do this is not available in the current unclassified literature, but it is theoretically possible that such an attack may exist. If this is a concern in your application, use /dev/random instead.

Using reqst_pid

为什么要使用reqst_pid?
这两个接口函数都需要提供reqst_pid参数，你知道为什么需要这个参数吗?

- The message sender may only responsible for forwarding requests.
  - The address of buf may not be provided by the message sender.
- But why drivers need to know reqst_pid?

Top half and bottom half

顶半处理和底半处理
这种中断处理机制的目的是尽可能快地从中断上下文返回，并且尽可能地把中断处理的真正工作往后延迟．如今的大部分操作系统也采取这种中断处理机制，你知道其中的原因吗?

- The interrupted process will be blocked until the handler finishes.
- A short top half will increase the concurrency of the system.

Interrupt and critical region (3)

危机四伏的中断(3)(这个问题有难度)
我们不能使用由信号量实现的互斥锁来解决中断嵌套问题，你知道为什么吗?

- use P(mutex) to protect critical regions with nested interrupt
  = P(mutex) may fail in interrupt context
  = potential deadlock

Interrupt and critical region (4)

危机四伏的中断(4)(这个问题有难度)
在SMP环境下，如果仅仅使用基于原子指令的锁机制，还是不能解决中断嵌套问题，你知道为什么吗？应该如何正确地解决它？

- use atomic instructions to protect critical regions with nested interrupt
  = interrupt will come during critical regions
  = atomic instruction may fail in interrupt context
  = potential deadlock

Synchronicity of message passing

消息机制的天然同步特性
我们已经实现了消息机制了，仔细思考实现的过程，天然同步的特性究竟从何而来?

- mutex send()
  - critical region protection
- serial receive()
  - requests are handled one by one

Implement timer

- Use ListHead to maintain the active timers.
  - initialize the active timer with cnt = second * HZ
- Register a top half to capture timer interrupts.
  - At each timer interrupt, cnt-- for each active timer.
  - Send a message to the requestor if a timer expires.

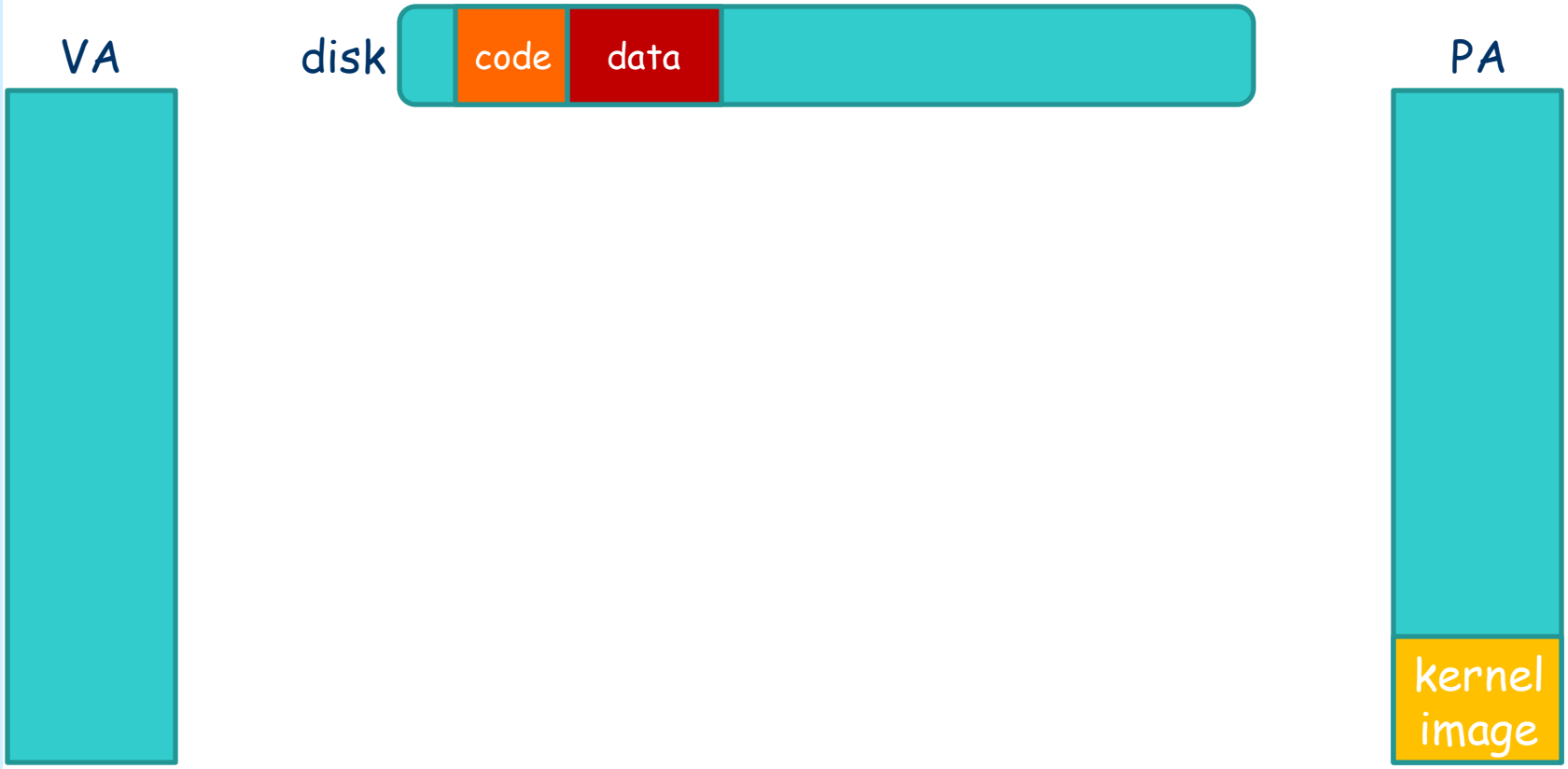
- A guide to create the first process

Preperation

- The binary executable is stored in disk.
- The program header table is found by parsing the ELF header.

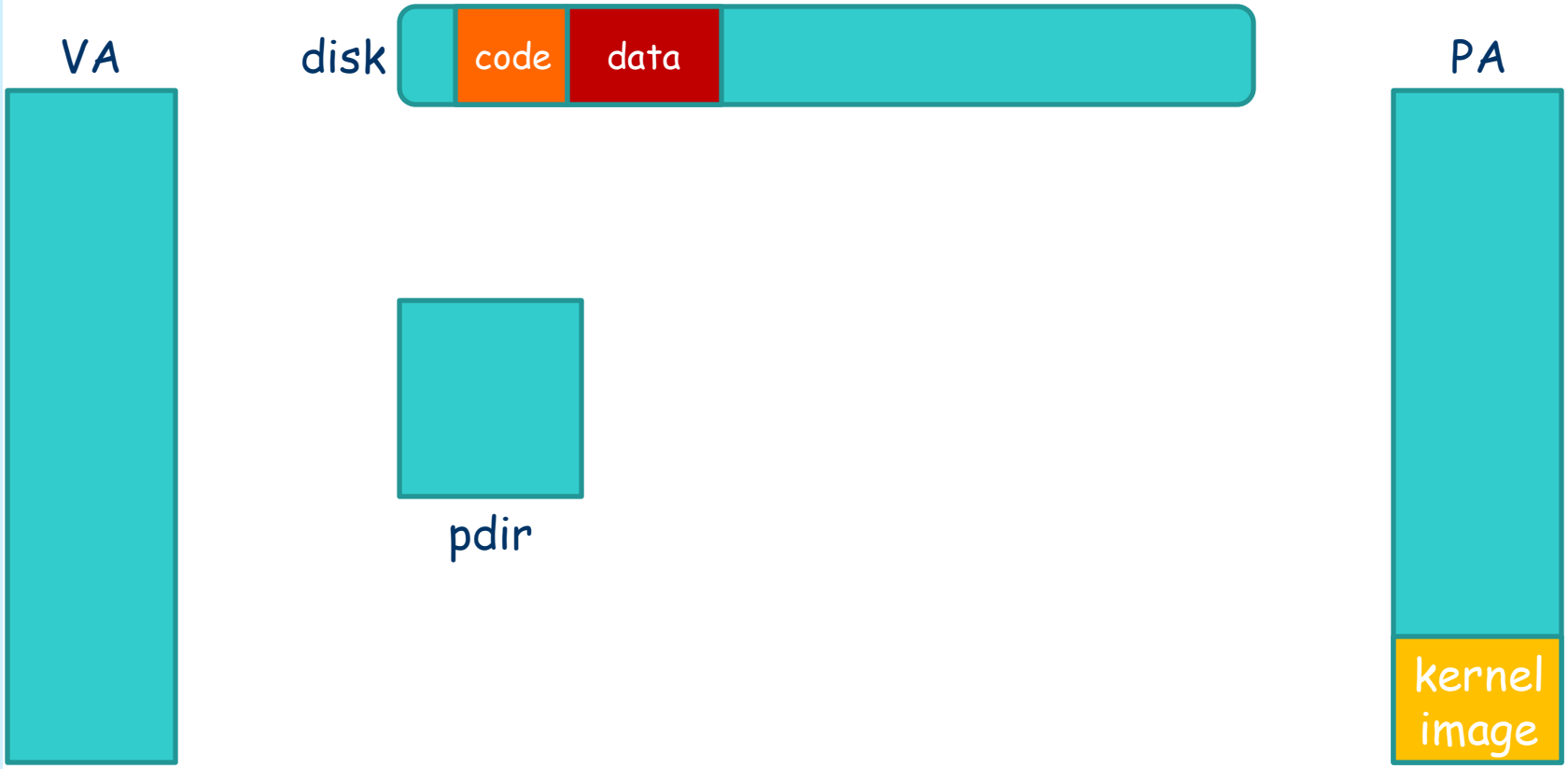
Start

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



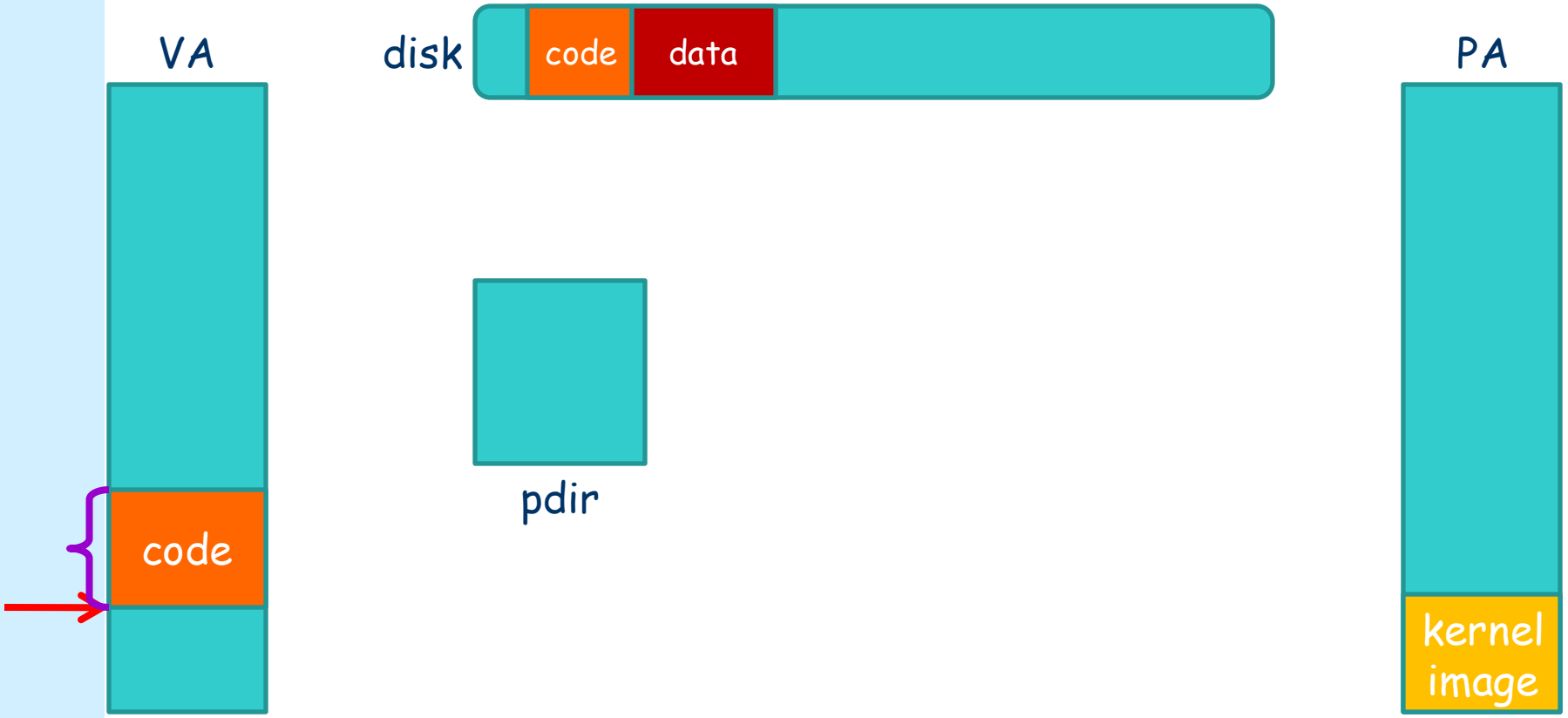
Allocate a page directory

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



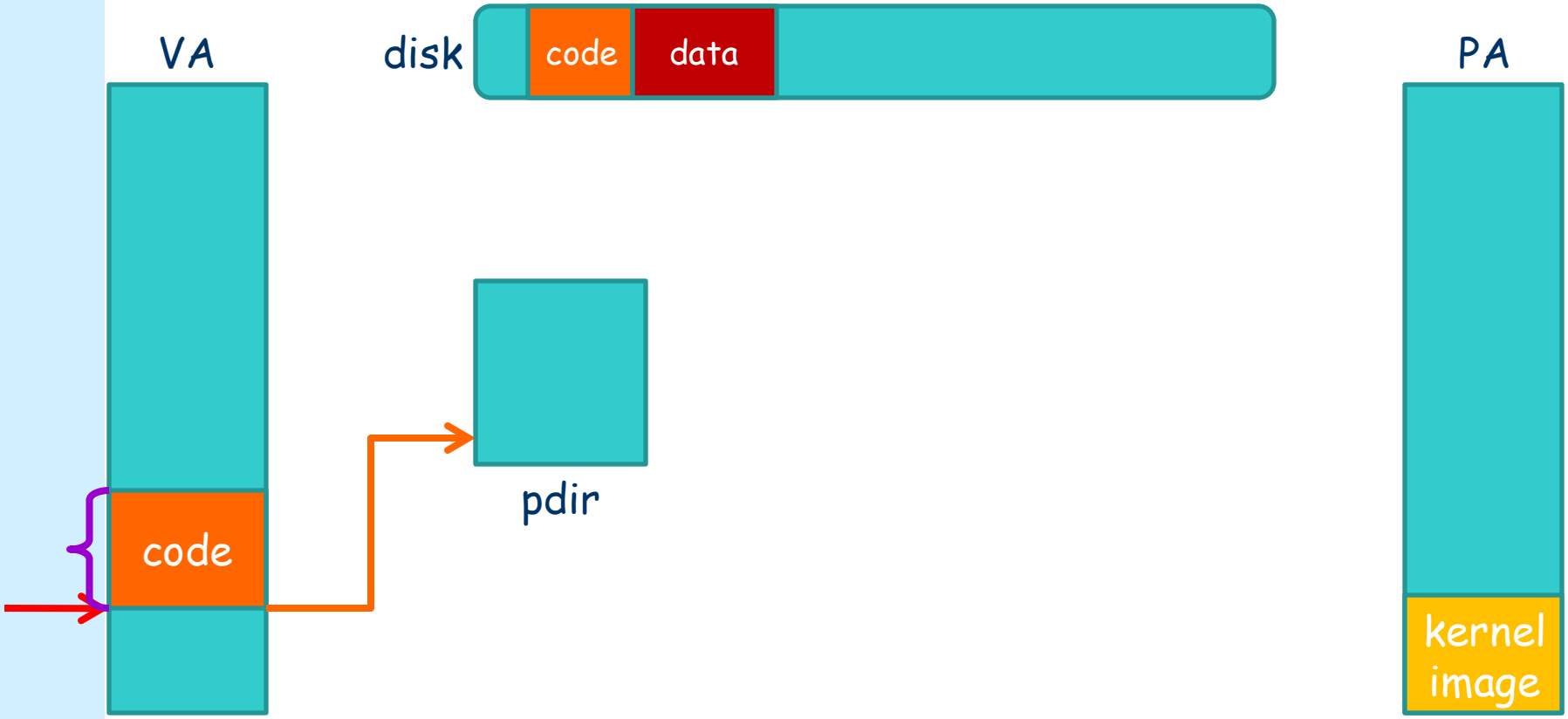
Find VA for code

```
Type           Offset   VirtAddr   PhysAddr   FileSiz  MemSiz   Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954  0x02954  R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000  0x15be0  RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000  0x00000  RWE 0x10
```



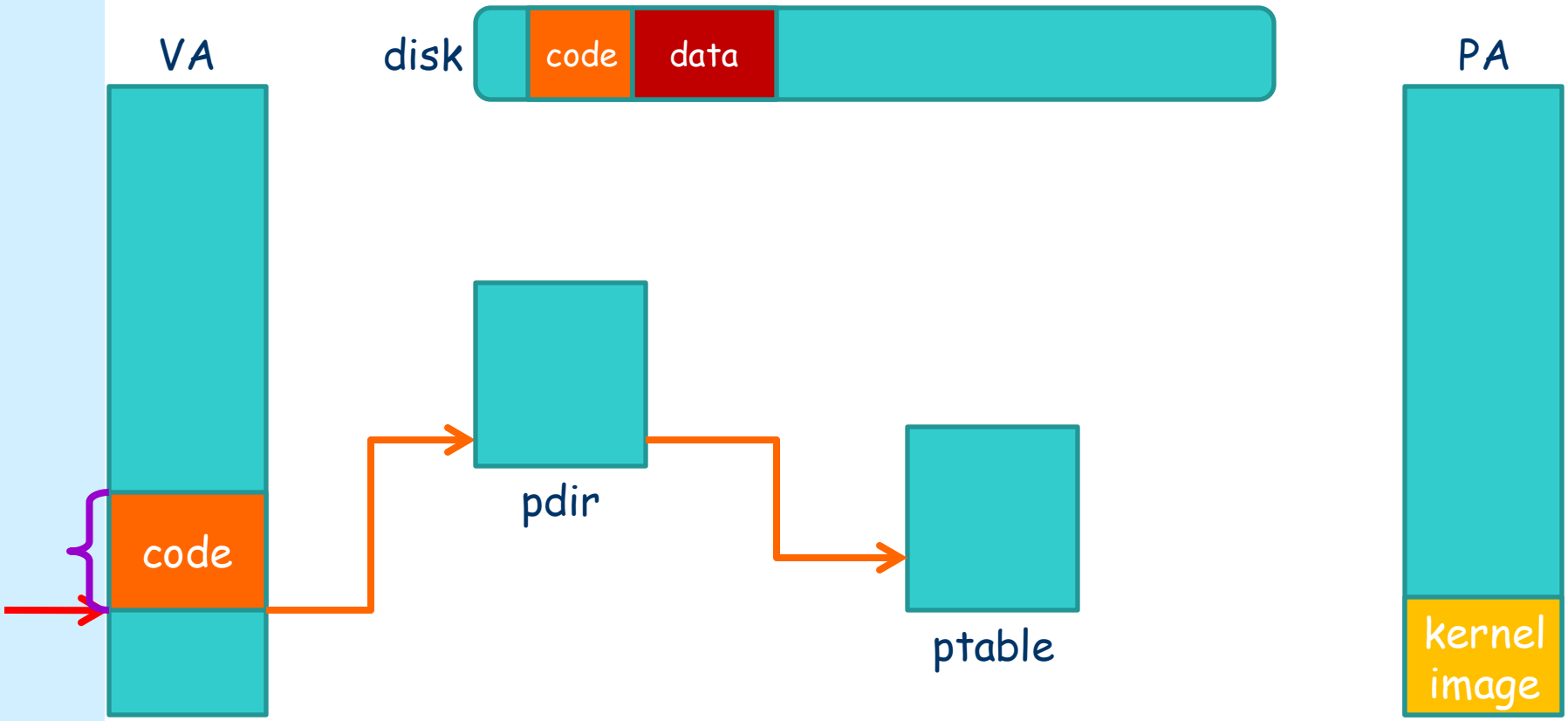
Allocate pages for code

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x08048000	0x08048000	0x02954	0x02954	R E	0x1000
LOAD	0x003000	0x0804b000	0x0804b000	0x00000	0x15be0	RW	0x1000
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10



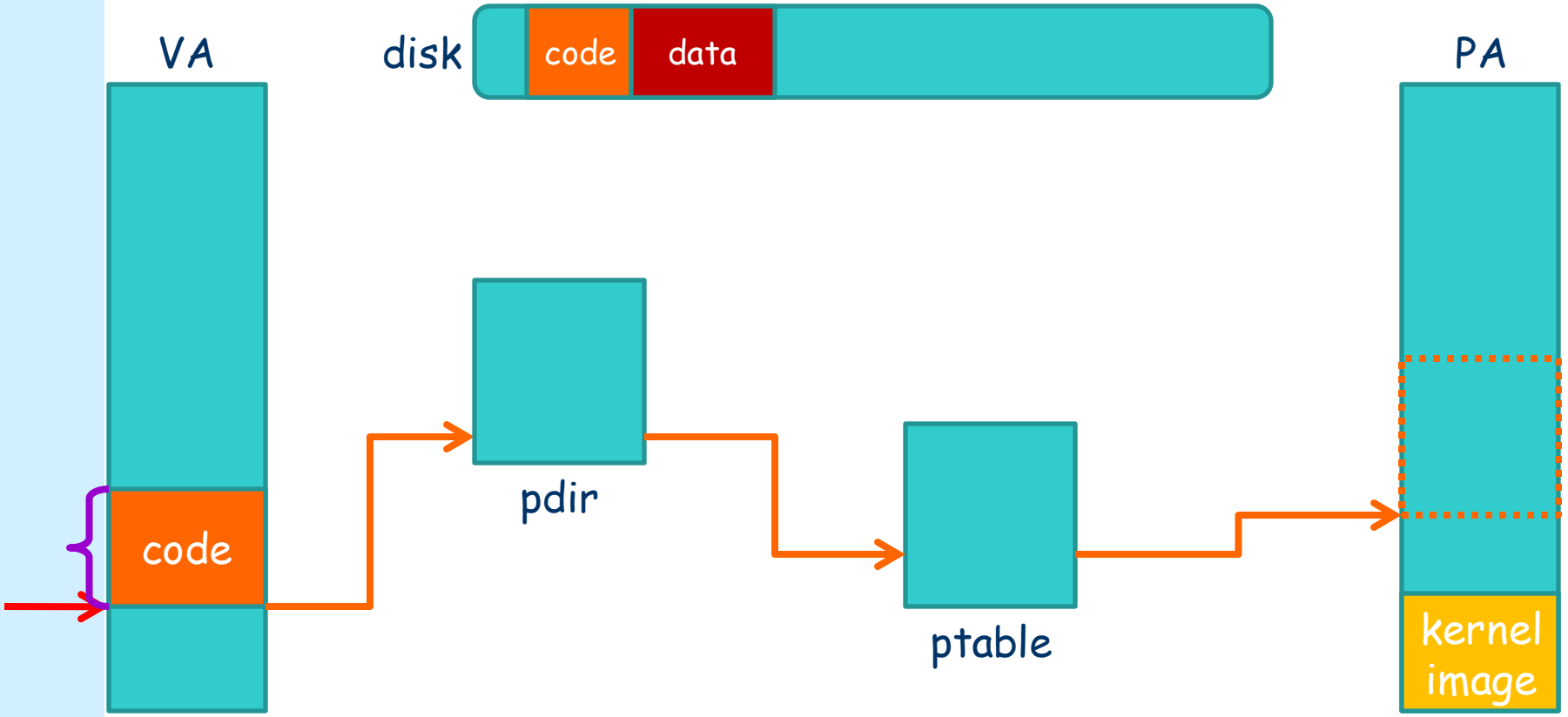
Allocate pages for code

```
Type           Offset   VirtAddr   PhysAddr   FileSiz  MemSiz   Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954  0x02954  R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000  0x15be0  RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000  0x00000  RWE 0x10
```



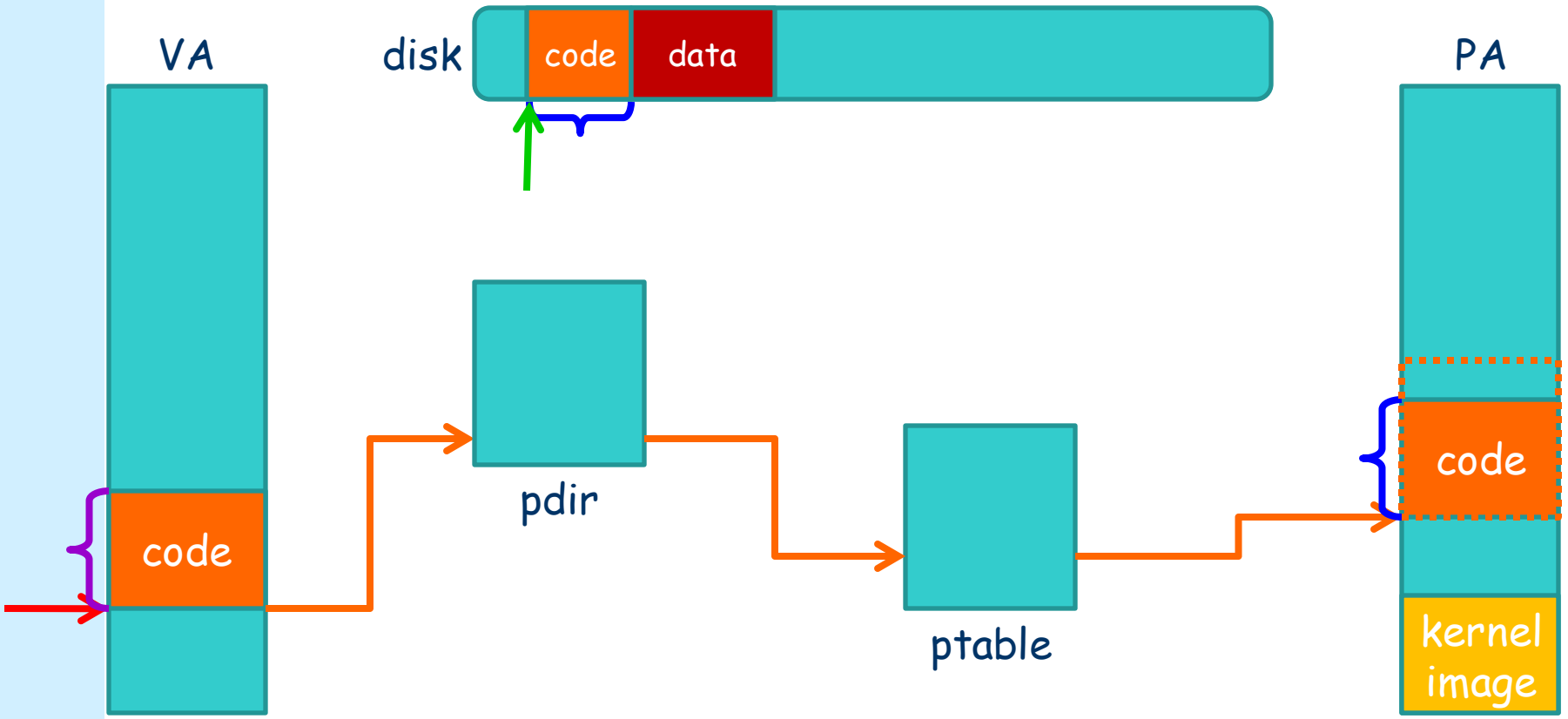
Allocate pages for code

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



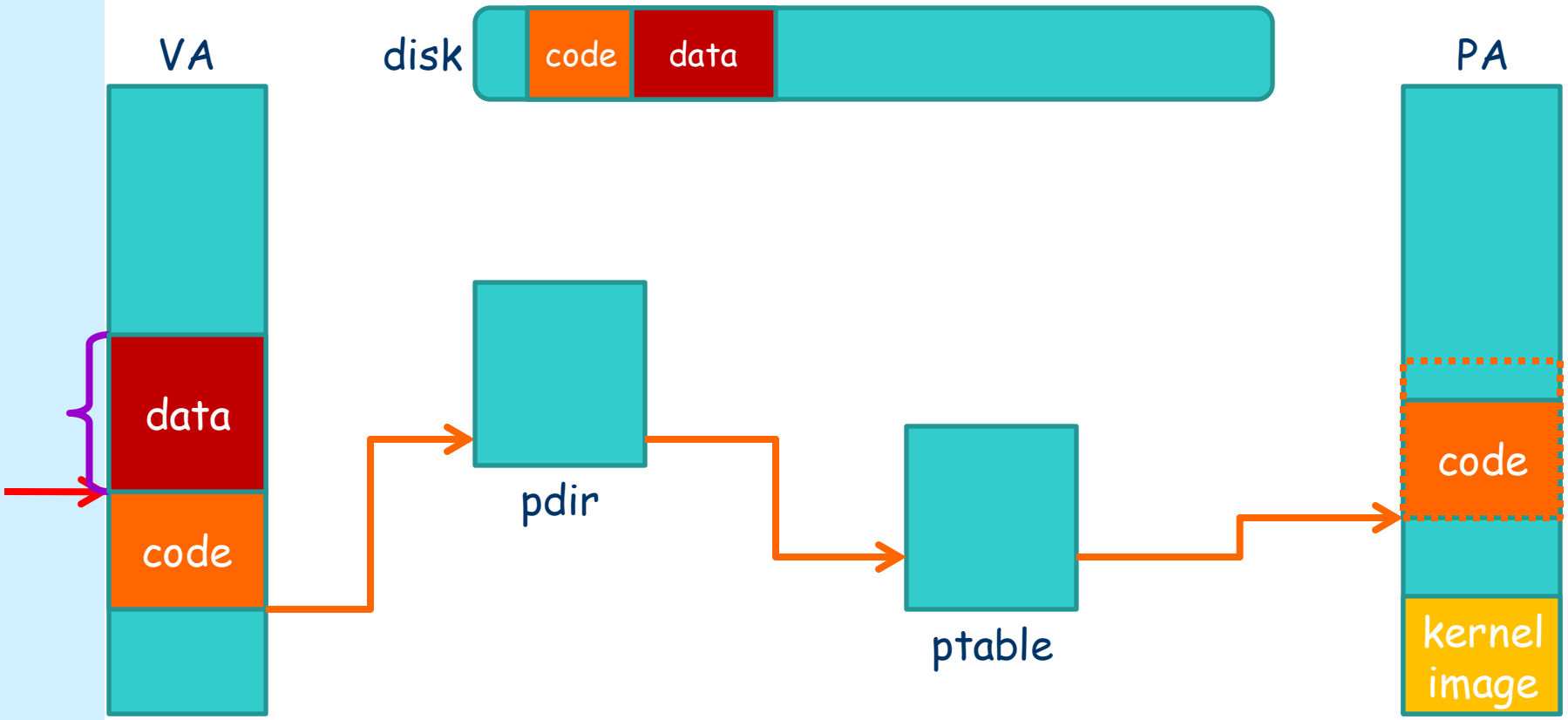
Load code into memory

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



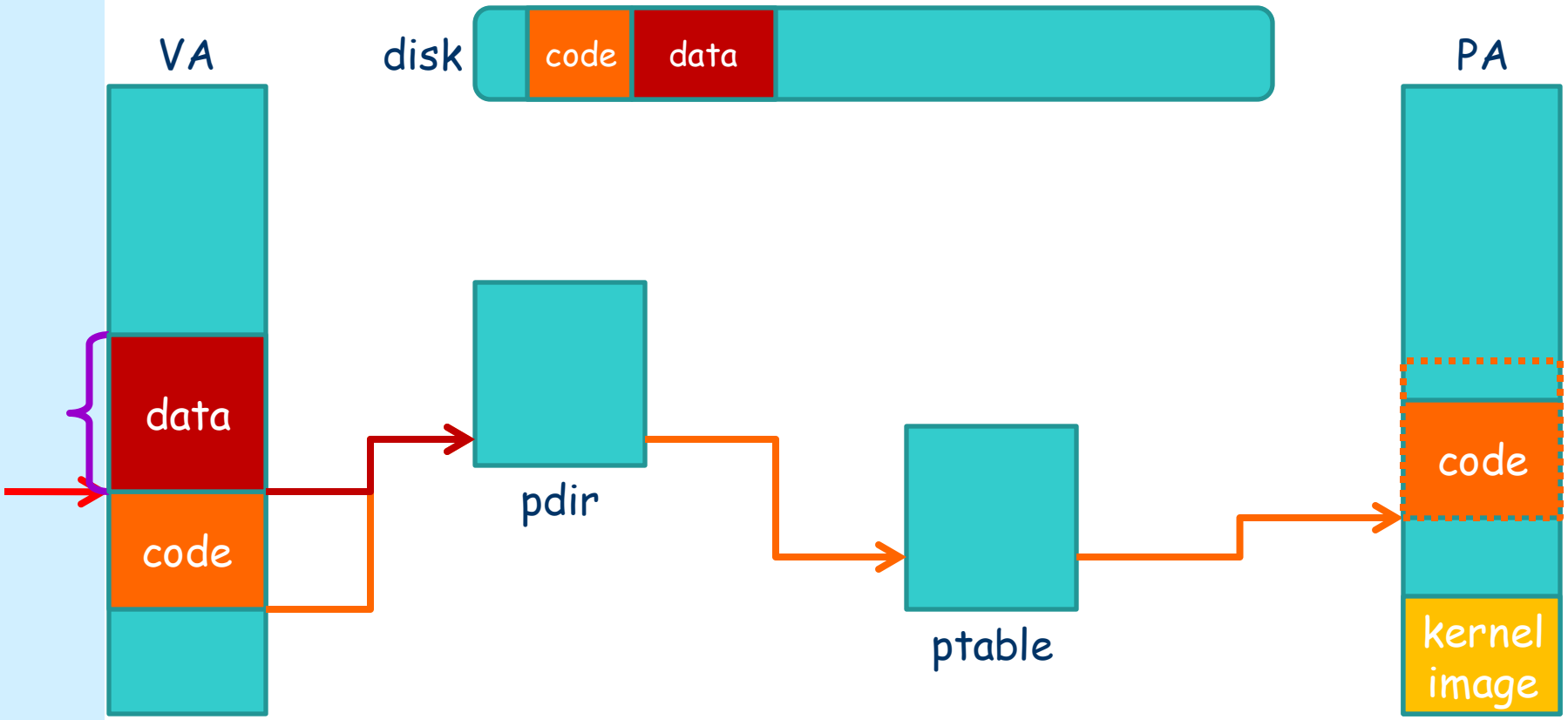
Find VA for data

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



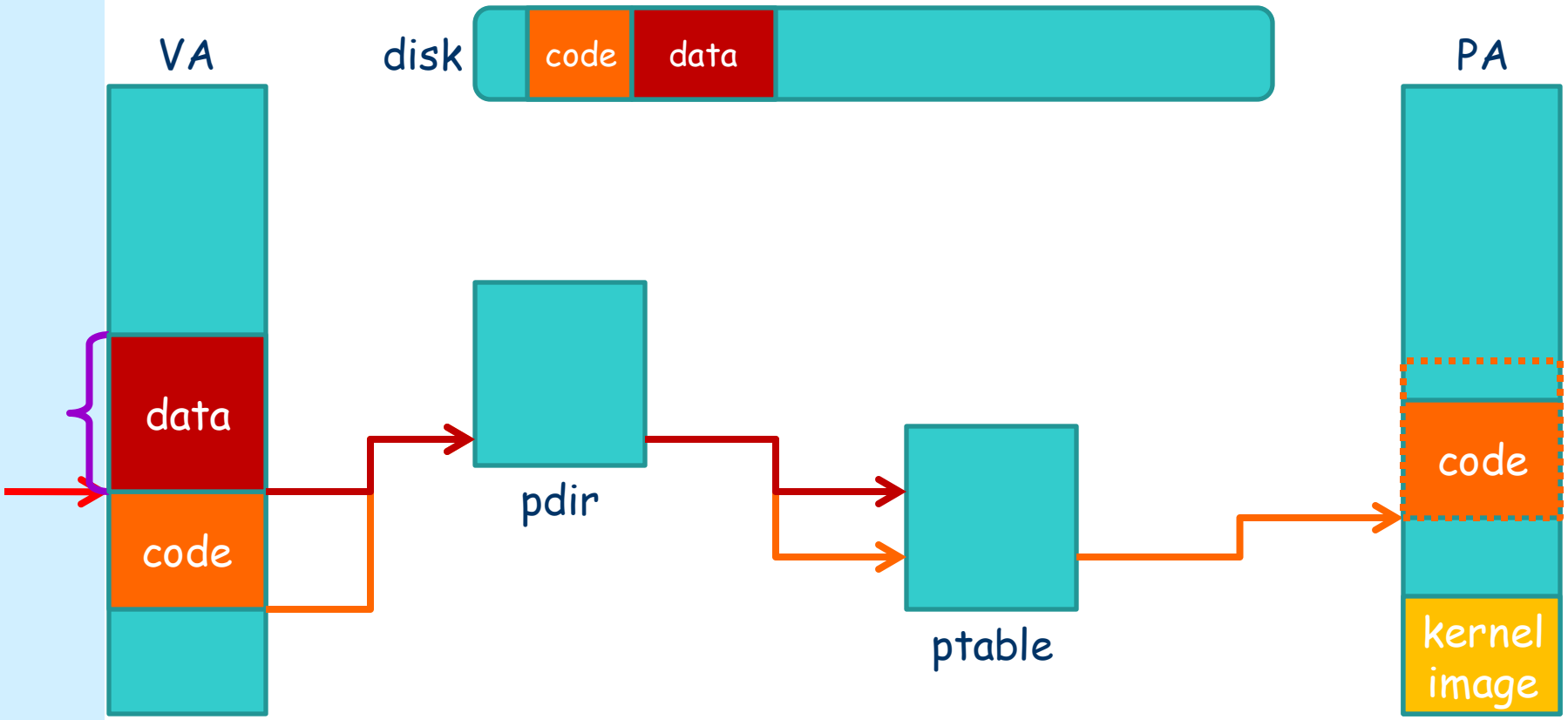
Allocate pages for data

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



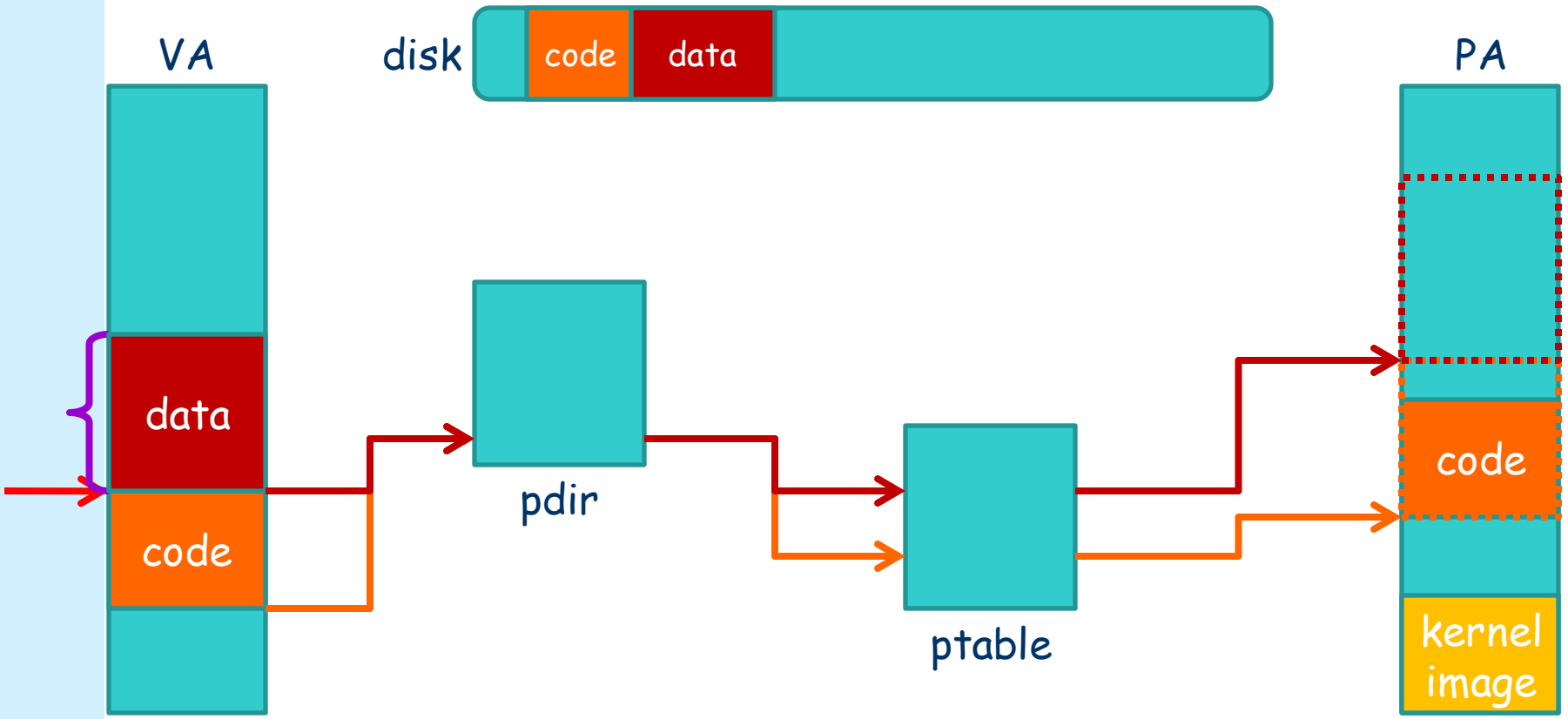
Allocate pages for data

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



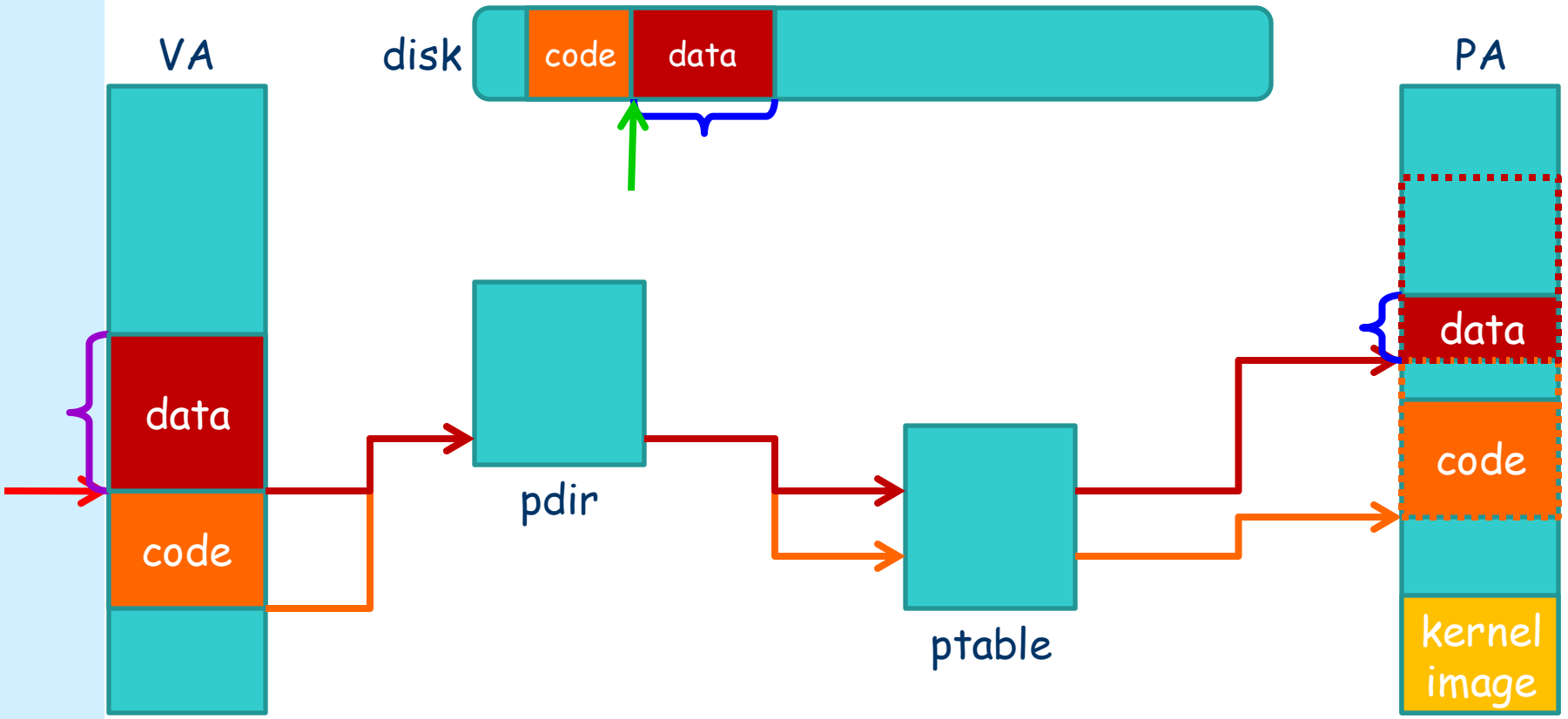
Allocate pages for data

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



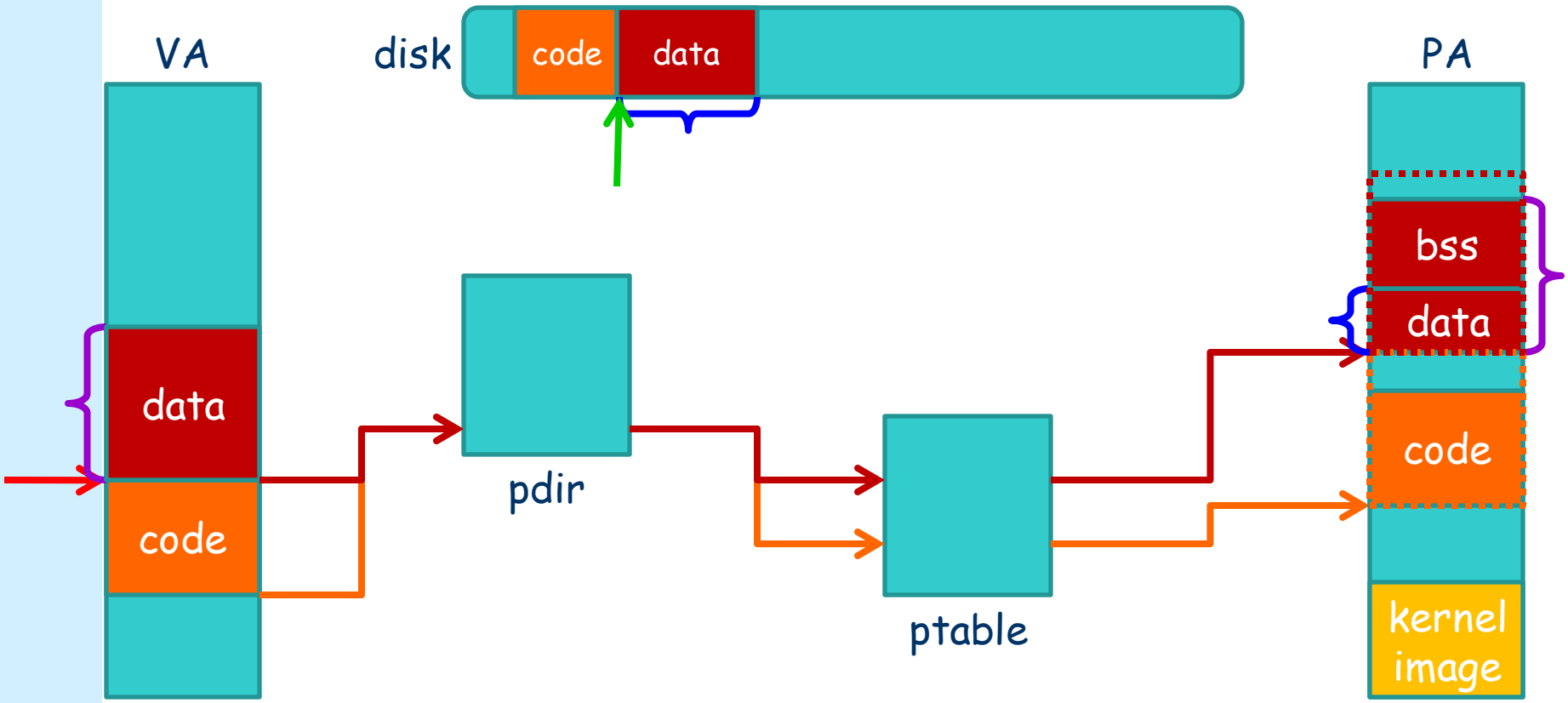
Load data into memory

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



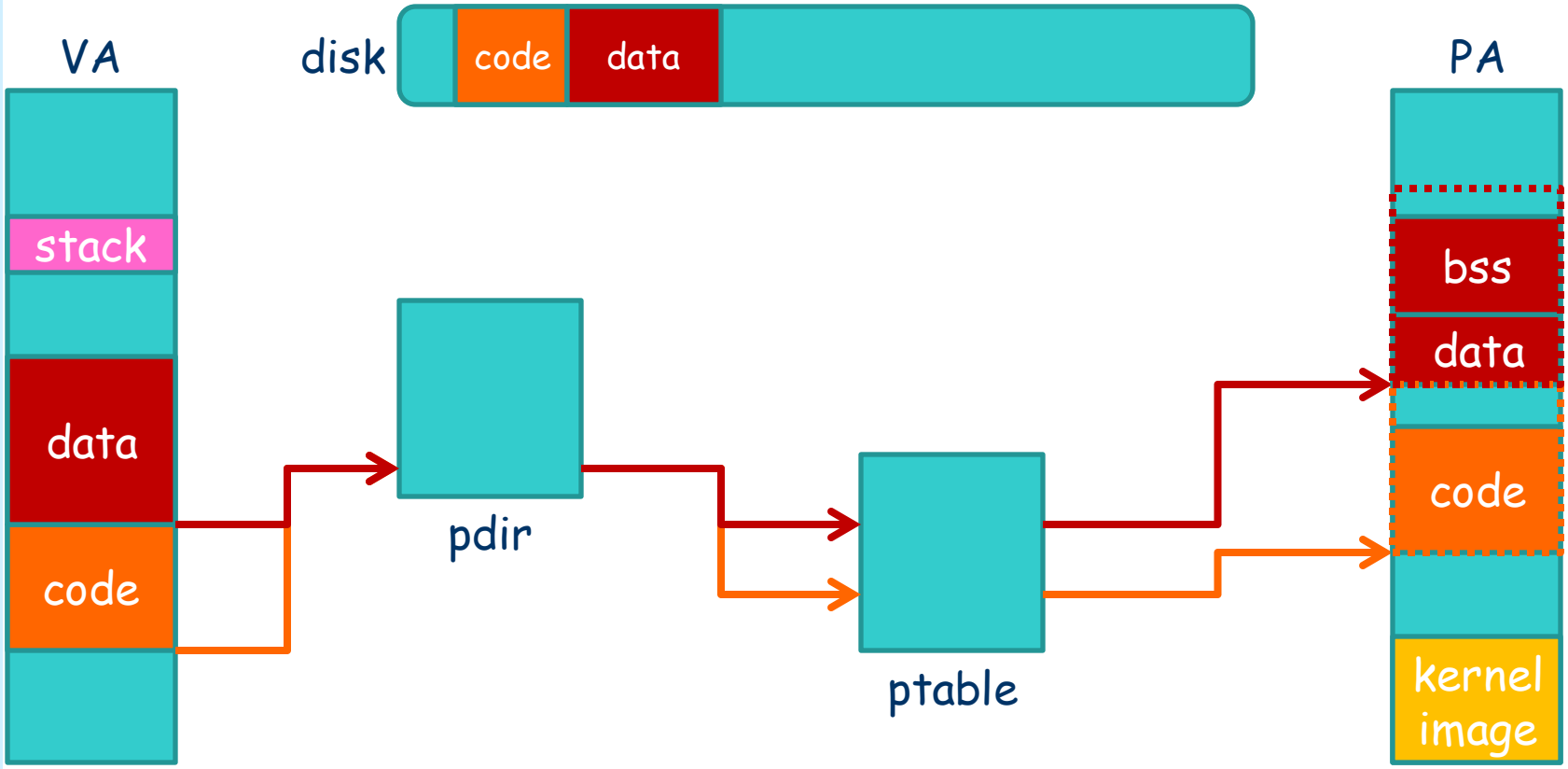
Initializa bss segment

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



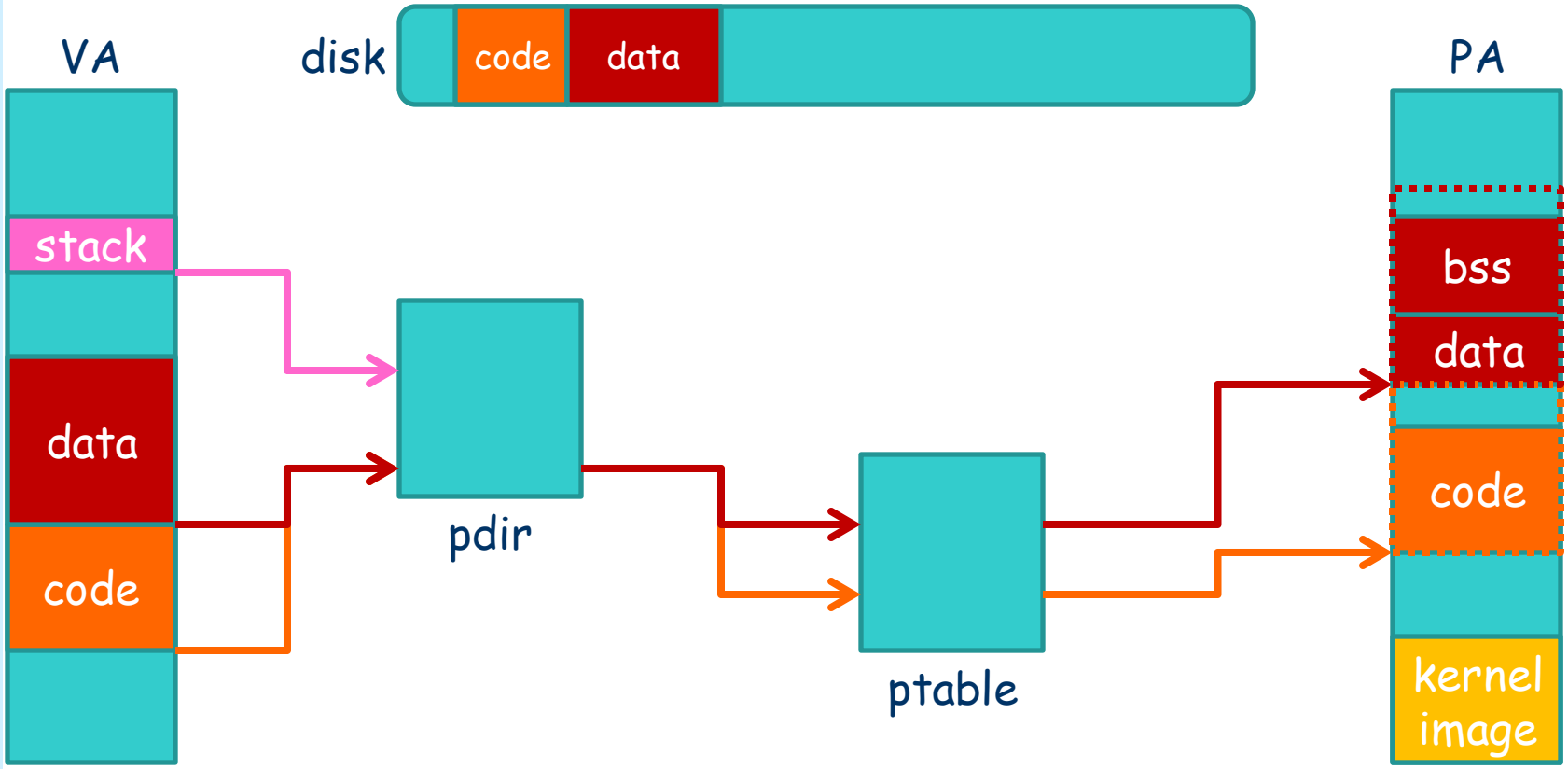
Allocate one page for user stack

```
Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD          0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD          0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



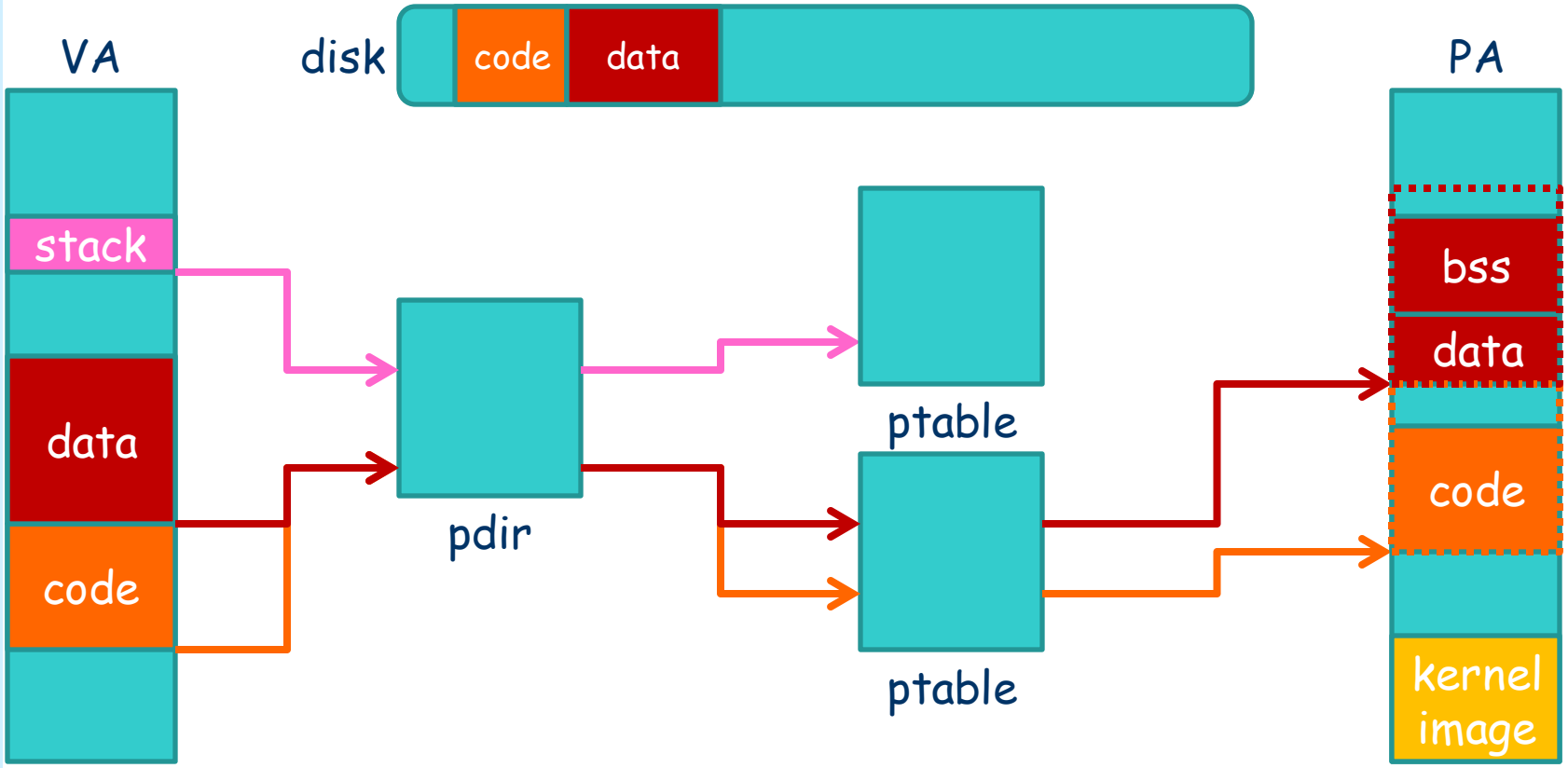
Allocate one page for user stack

```
Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD          0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD          0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



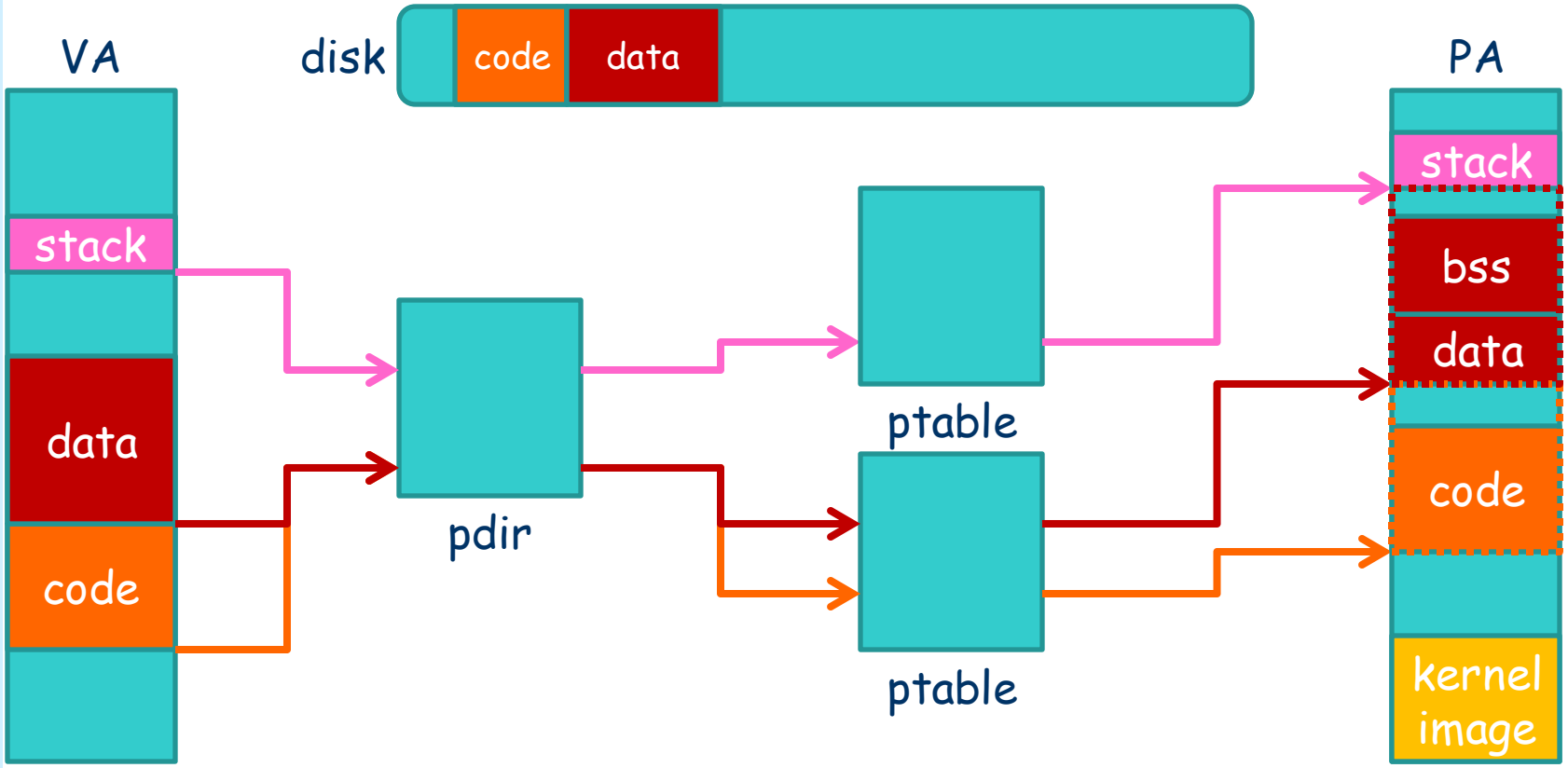
Allocate one page for user stack

```
Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD          0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD          0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



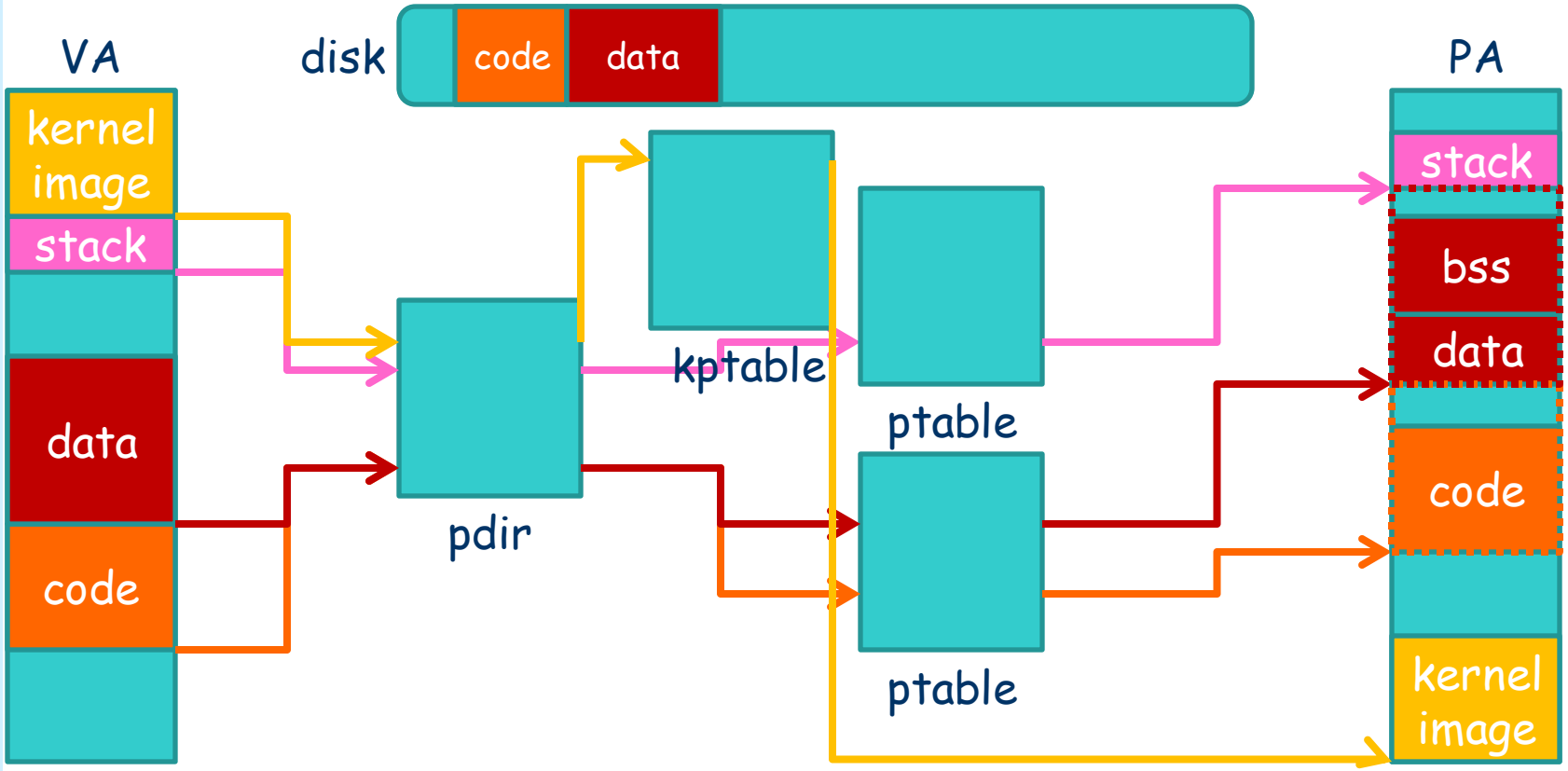
Allocate one page for user stack

```
Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD          0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD          0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



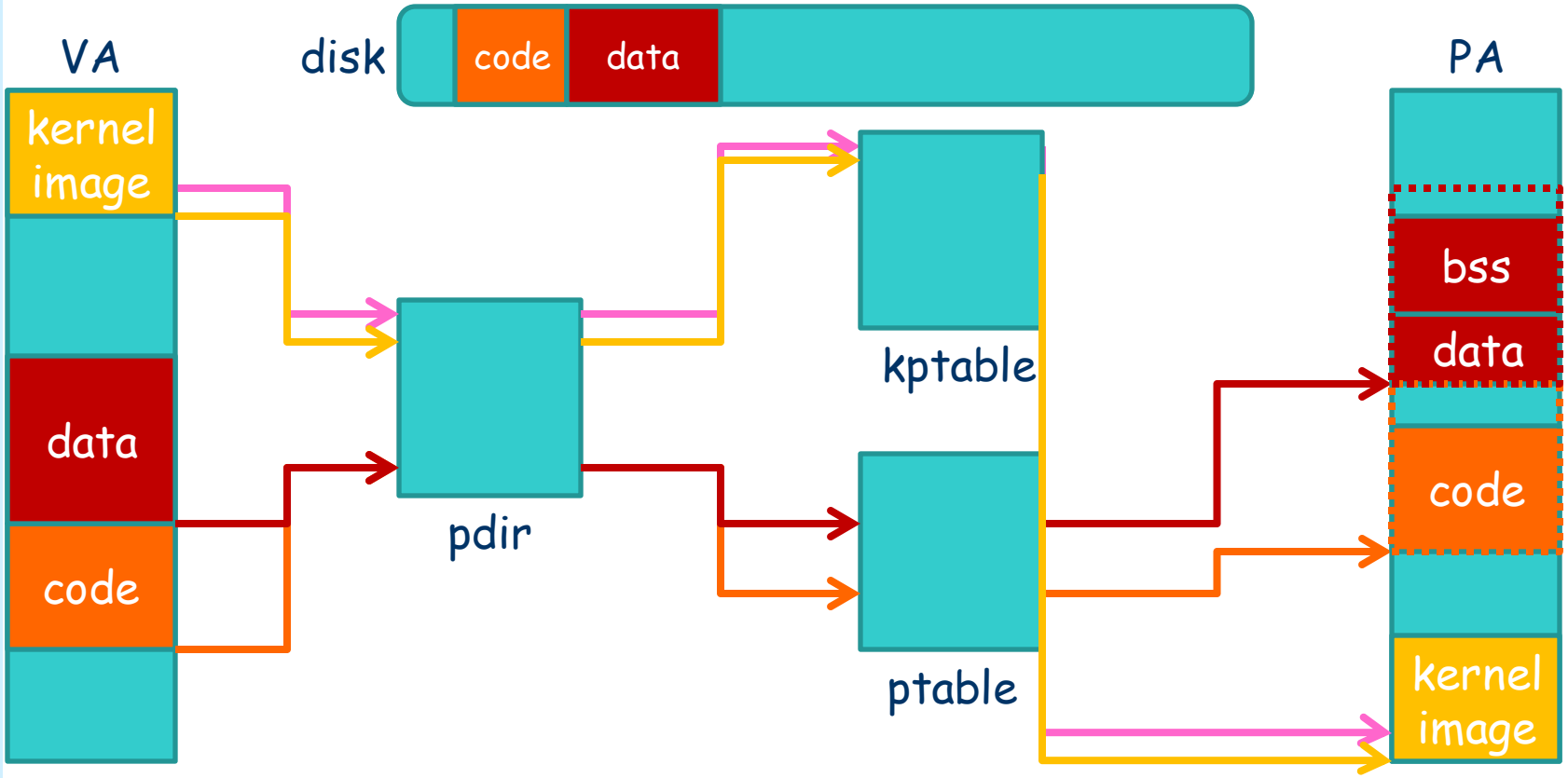
Create mapping to kernel image

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



Simplication: use kernel stack

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x08048000 0x08048000 0x02954 0x02954 R E 0x1000
LOAD           0x003000 0x0804b000 0x0804b000 0x00000 0x15be0 RW  0x1000
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```



Simplication: use kernel stack

- user process runs in ring0 → use kernel stack with no privilege exceptions
- initialize the current state in kernel stack
  - exactly what you have done in Lab1
- remaining work
  - initialize PCB
  - put the user process into ready queue
  - When the scheduler chooses it, your first process is running!
- Have fun in Lab3!