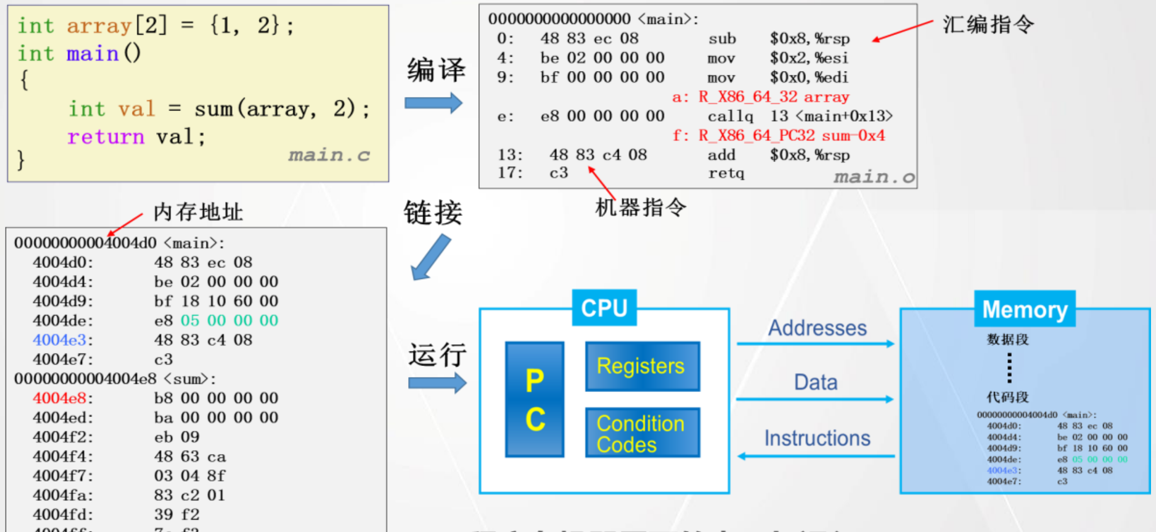
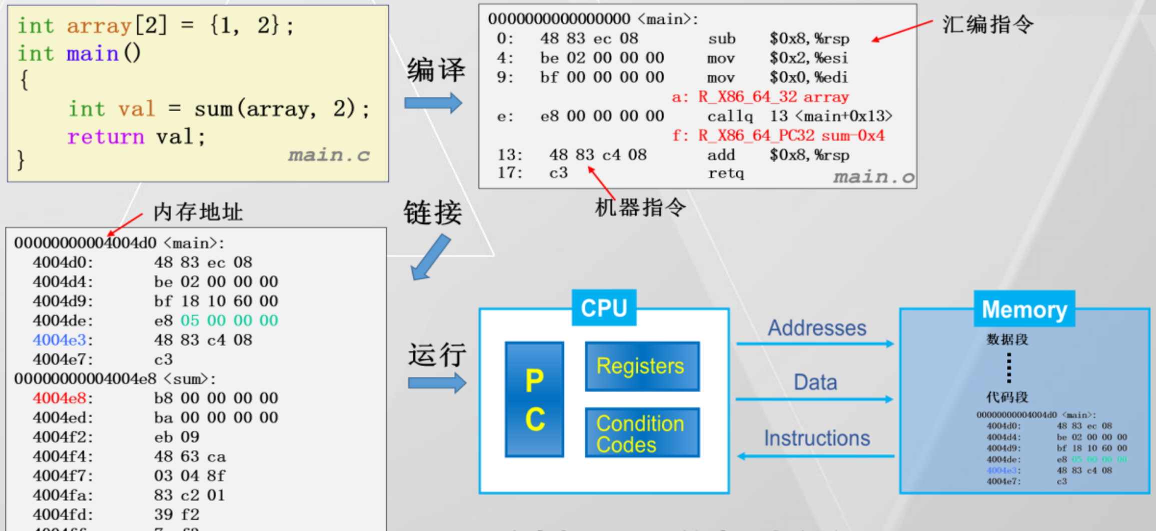
汇编语言程序设计

第15讲 异常与MIPS32异常处理



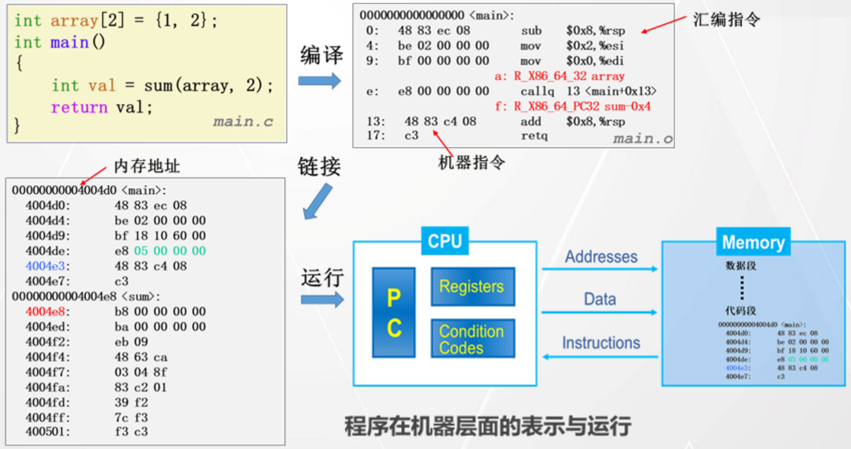
程序在机器层面的表示与运行


程序在机器层面的表示与运行

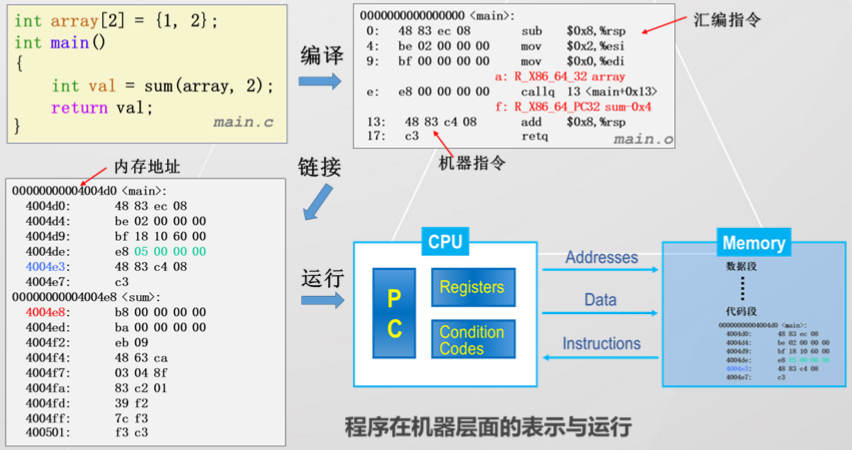
（看上去）一个程序占据了一个处理器以及一块完整的内存空间，但是实际情况要远为复杂！ 怎么做到的？

关系到计算机系统的两个重要概念：

- 虚存——在有限物理内存前提下设计出连续的、相互独立的虚拟内存
- 异常——各个任务切换的重要机制（当然异常还有很多其他作用）



程序在机器层面的表示与运行



程序在机器层面的表示与运行

（看上去）一个程序占据了一个处理器以及一块完整的内存空间。但是实际情况要远为复杂!

怎么做到的?

关系到计算机系统的两个重要概念：

- **虚存——在有限物理内存前提下设计出连续的、相互独立的虚拟内存**
- **异常——各个任务切换的重要机制（当然异常还有很多其他作用）**

与进程密切相关的系统调用，如fork等充分利用了虚存特性

目录

异常(Exception)

在程序运行过程中，某些打断程序正常运行流程的、且会引起运行态改变（从用户态到核心态）的事件

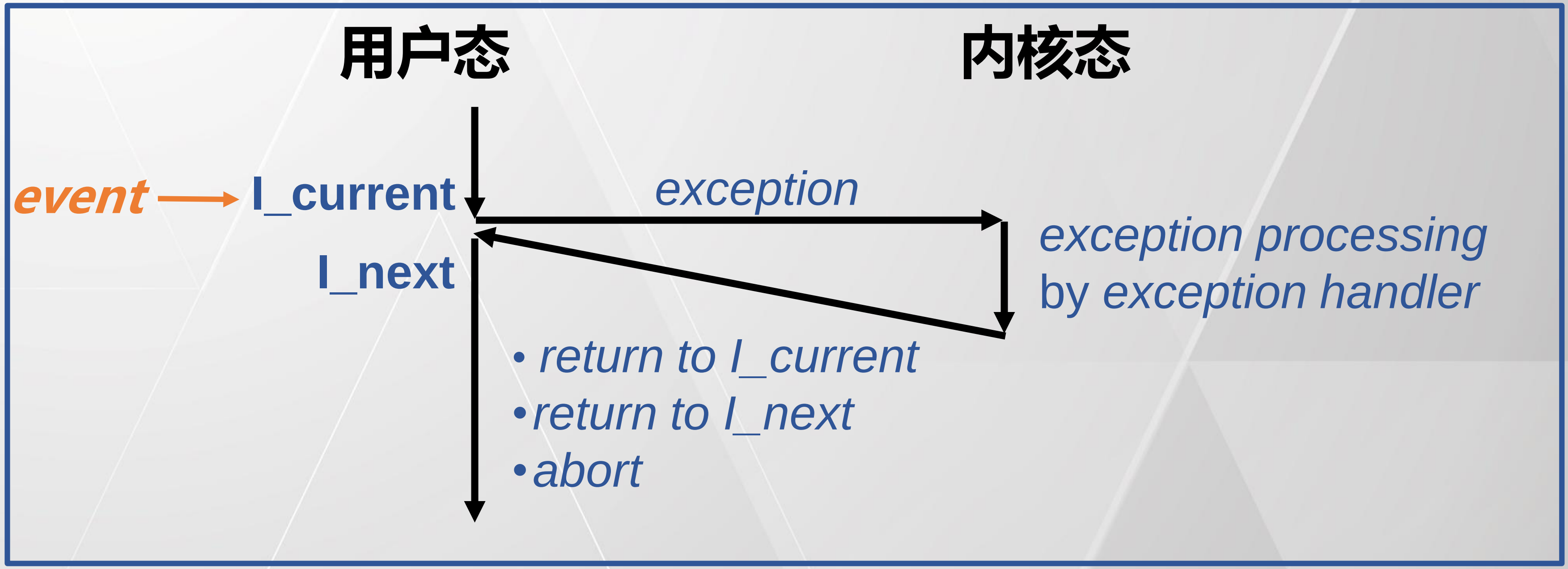
- 示例：
  - 网卡收到数据 / 磁盘数据读取完成
  - （除法）指令除以零
  - 用户在键盘上按Ctrl-C
  - 系统定时器到期

分为两类

1、同步异常

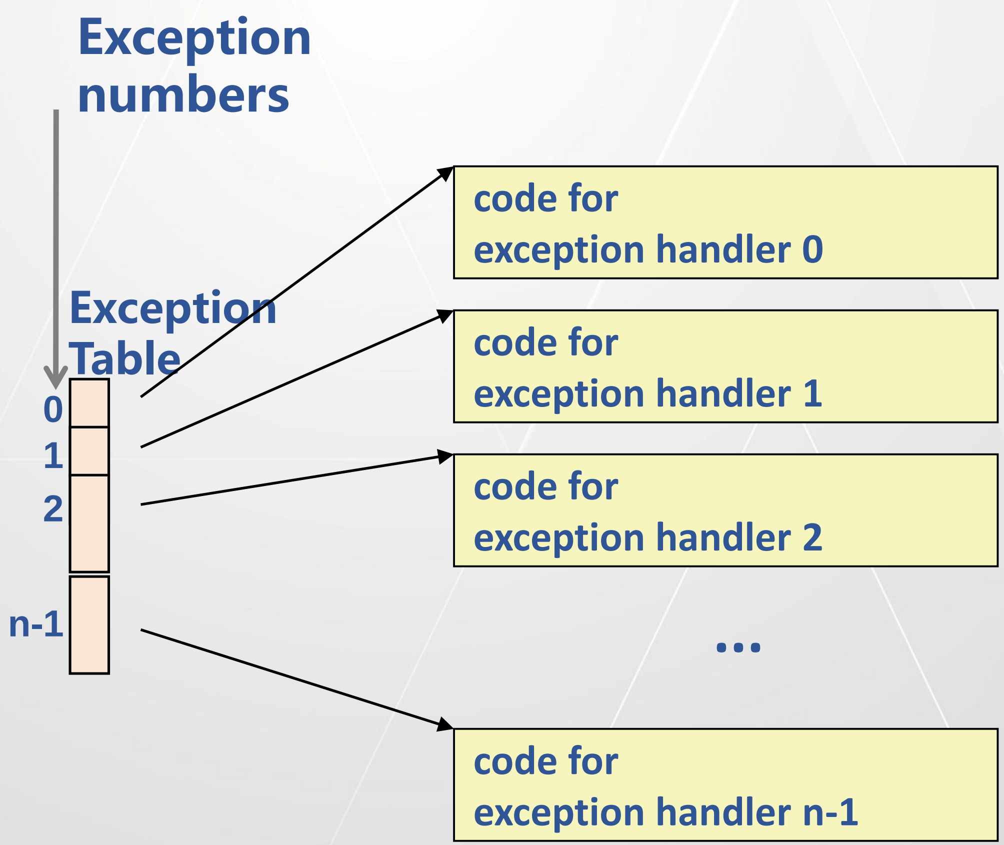
2、异步异常

异常



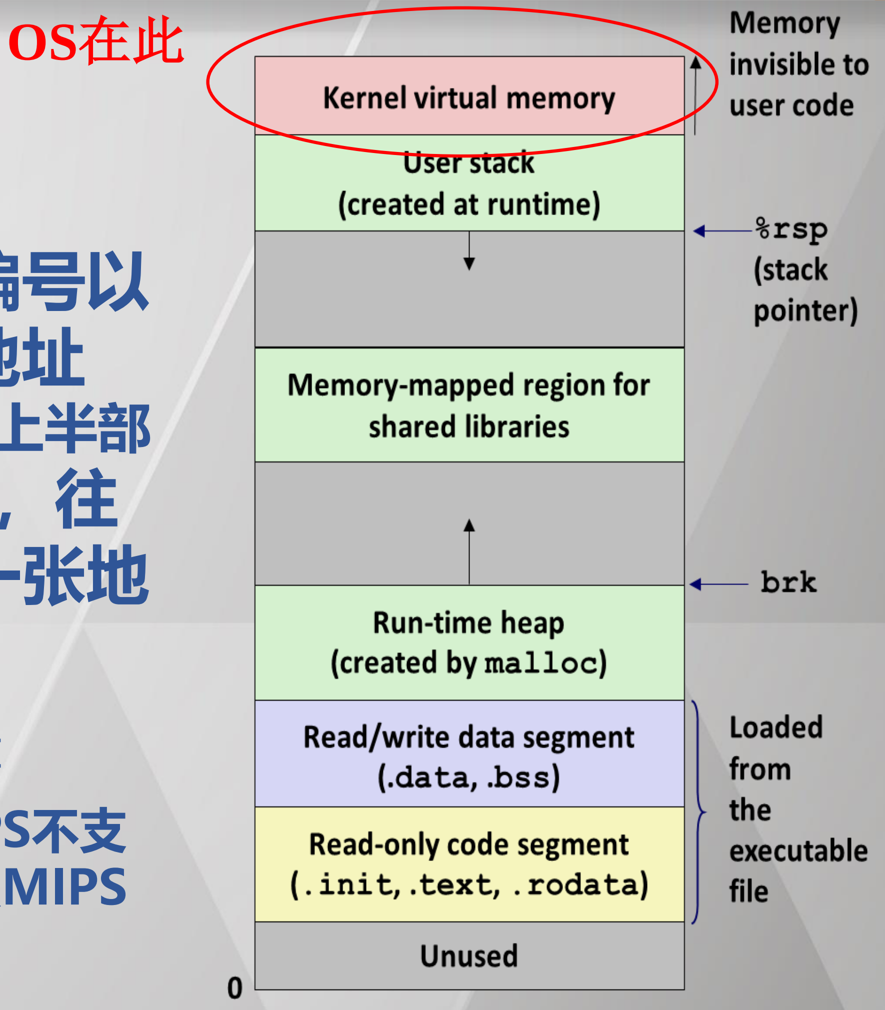
用户态
内核态
event
I_current
I_next
exception
exception processing
by exception handler
• return to I_current
•return to I_next
•abort

异常处理向量（Vectors）



每类异常都有其编号以及异常处理入口地址（位于虚存空间的“上半部分”，即内核空间），往往在内存中构成一张地址表

- 所以称之为向量
- 但是早期的MIPS不支持这种模式，从MIPS 4Ke开始引入



同步异常

由指令执行引起的，分为三类

一、陷入（Traps）

程序“故意”引起的

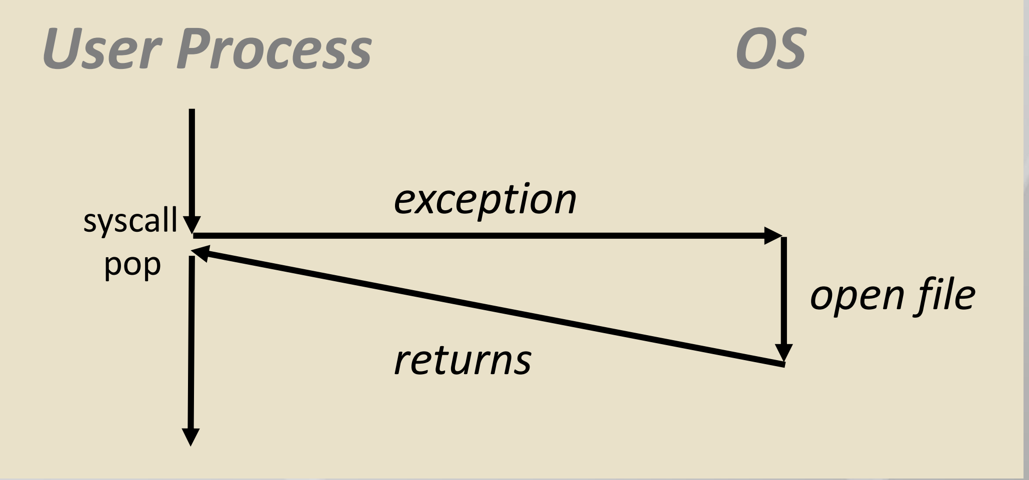
如: 系统调用（system call）、断点（breakpoint）、Trap指令

返回到下一条指令

Trap Example: Opening File

Call Code = 2	SYS_open	Open a file.
		rdi = address of NULL terminated file name rsi = file status flags (typically 0 RDONLY)
	If unsuccessful, returns negative value. If successful, returns file descriptor.	

```
0804d070 <__libc_open>:
 . . .
 804d082:       0f 05                    syscall
 804d084:       5b                       pop    %ebx
 . . .
```



二、Faults

程序“无意”引起的、但是可恢复

如: 页缺失

(recoverable)

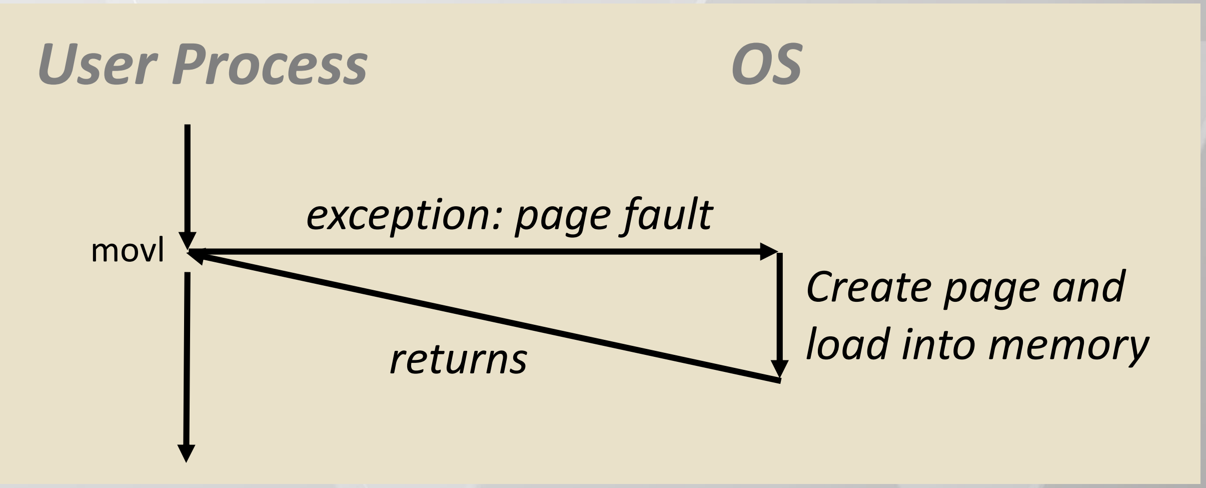
重新执行当前指令

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

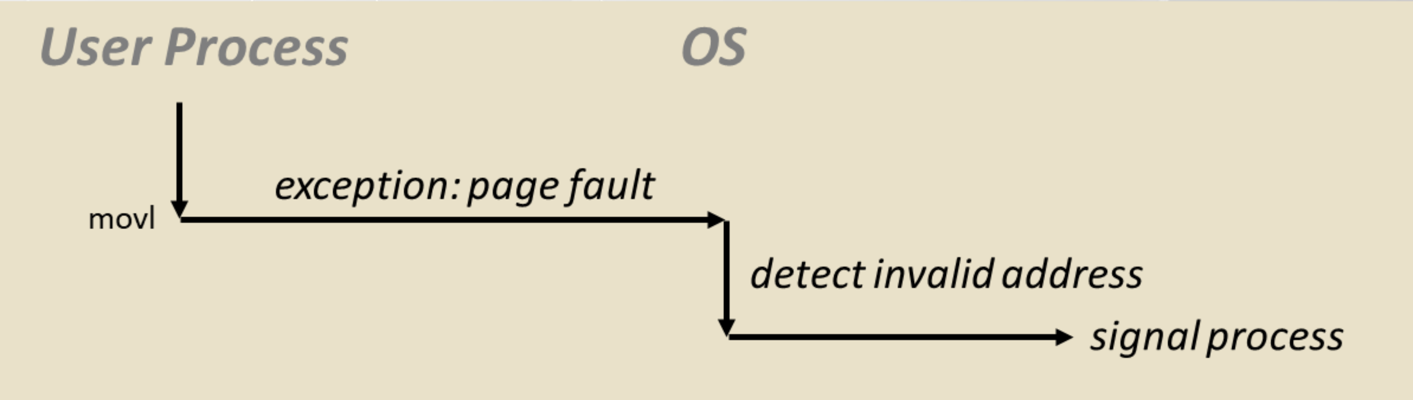
```
80483b7: c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example-2: Invalid Mem Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```



"segmentation fault"

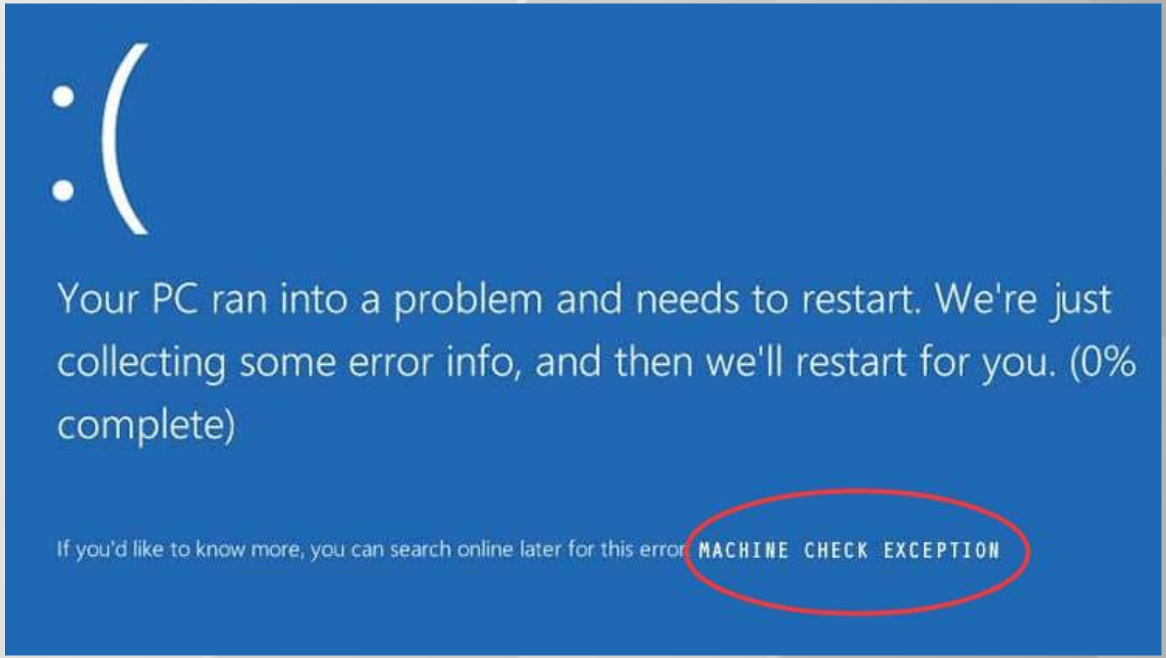
三、Aborts

程序“无意”引起的、不可恢复

如: parity error （奇偶校验错），machine check

程序退出

Aborts Example: 蓝屏~~



异步异常（中断）

由“外部事件”引起，往往是外设触发

如：IO中断（网卡收到数据 / 磁盘数据读取完成等）、Hard Reset、Soft Reset

处理完后返回到“下一条”指令

对MIPS32而言，在中断发生时，如果指令已经完成了MEM阶段的操作，则保证该指令执行完毕。反之，则丢弃流水线对这条指令的工作

精确异常处理（现代处理器一般都支持，以MIPS为例）

精确异常指的就是： 一般情况下，产生异常的指令之前的指令都应执行完毕；该指令之后的则都不处理

1. 需要精确记录异常的位置（指令）
    - Branch Delay Slot发生的异常如何处理？
2. 需要取消后续指令
3. 需要正确恢复执行

该指令本身如何处理?

取决于异常类型

目录

进程概念

- **定义: 一个进程就是程序的一个运行实例**
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

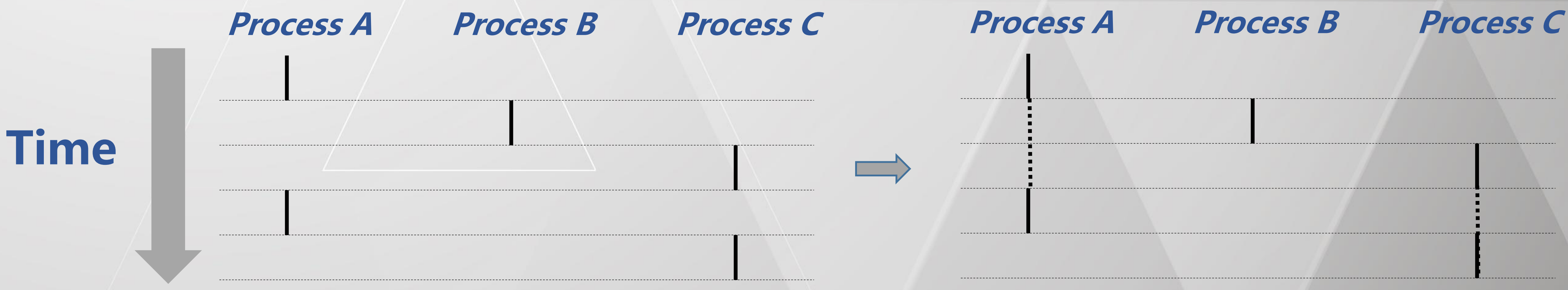
- **进程包含有两个关键抽象:**
  - Each seems to have exclusive use of the CPU
  - Each seems to have exclusive use of main memory (Private virtual address space)

  进程的二要素

- **如何实现的?**
  - Process executions interleaved (multitasking) or run on separate cores
  - Address spaces managed by virtual memory system

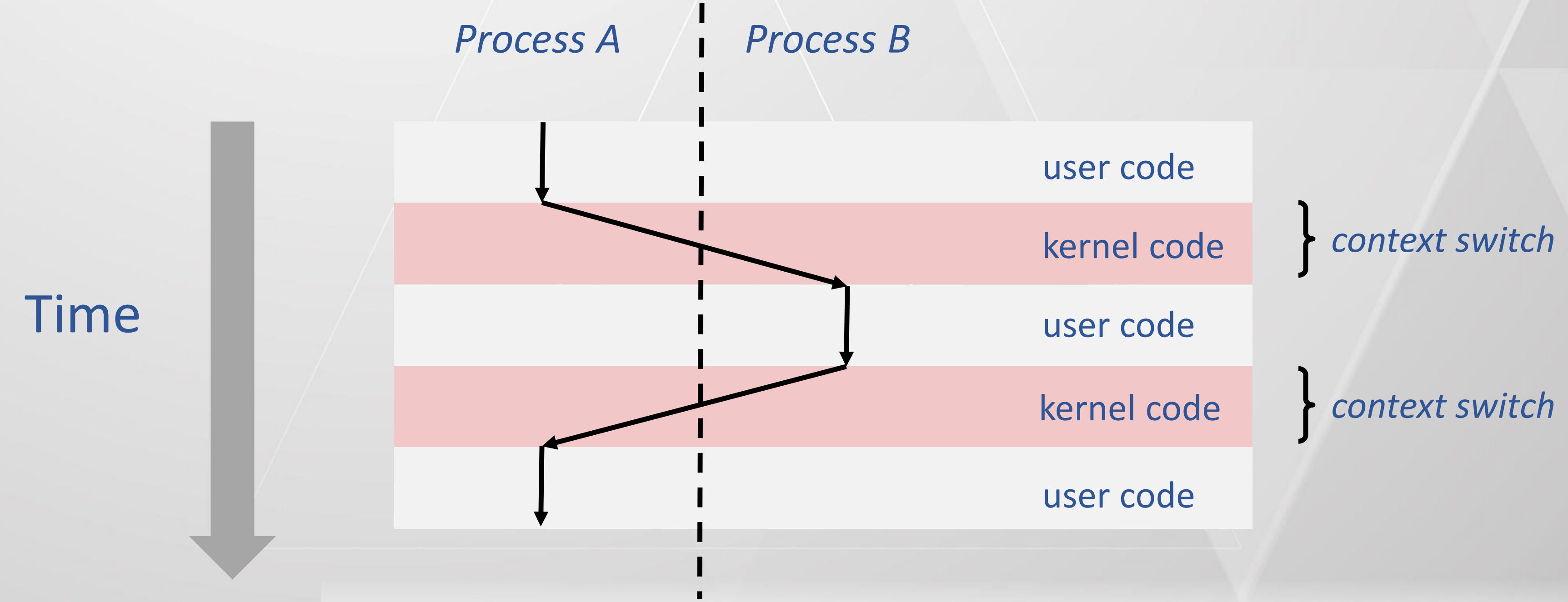
并发进程

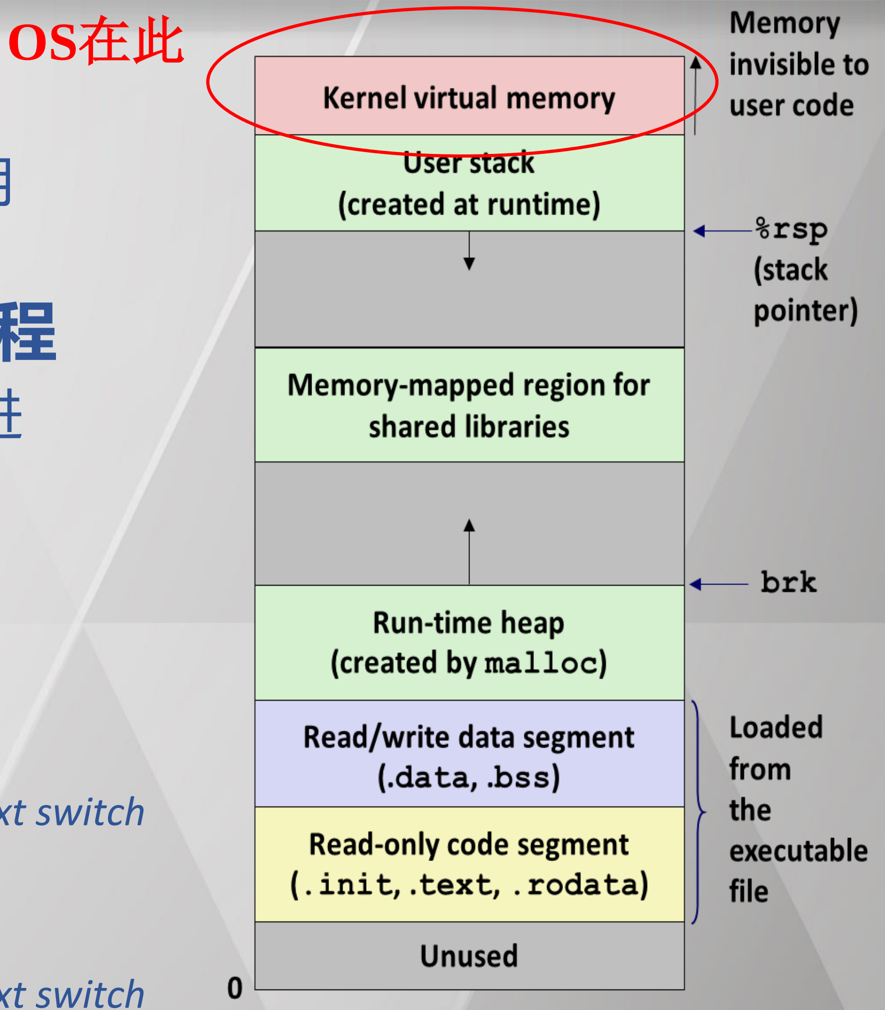
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



进程切换（Context Switching，也叫进程上下文切换）

- **进程由称为内核的操作系统代码管理**
  - 重要提示：内核不是一个独立的进程，而是作为某个用户进程的一部分运行的
- **通过上下文切换，从一个进程切换到到另一个进程**
  - 切换的时机——异常：可能是系统调用引起的，或者进程运行时间片用完（通过时钟中断）





创建进程（fork）

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process，即克隆当前进程~)
  - returns 0 to the child process
  - returns child's **pid** to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- **Fork is interesting (and often confusing) because it is called *once* but returns *twice***

关于fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from parent
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

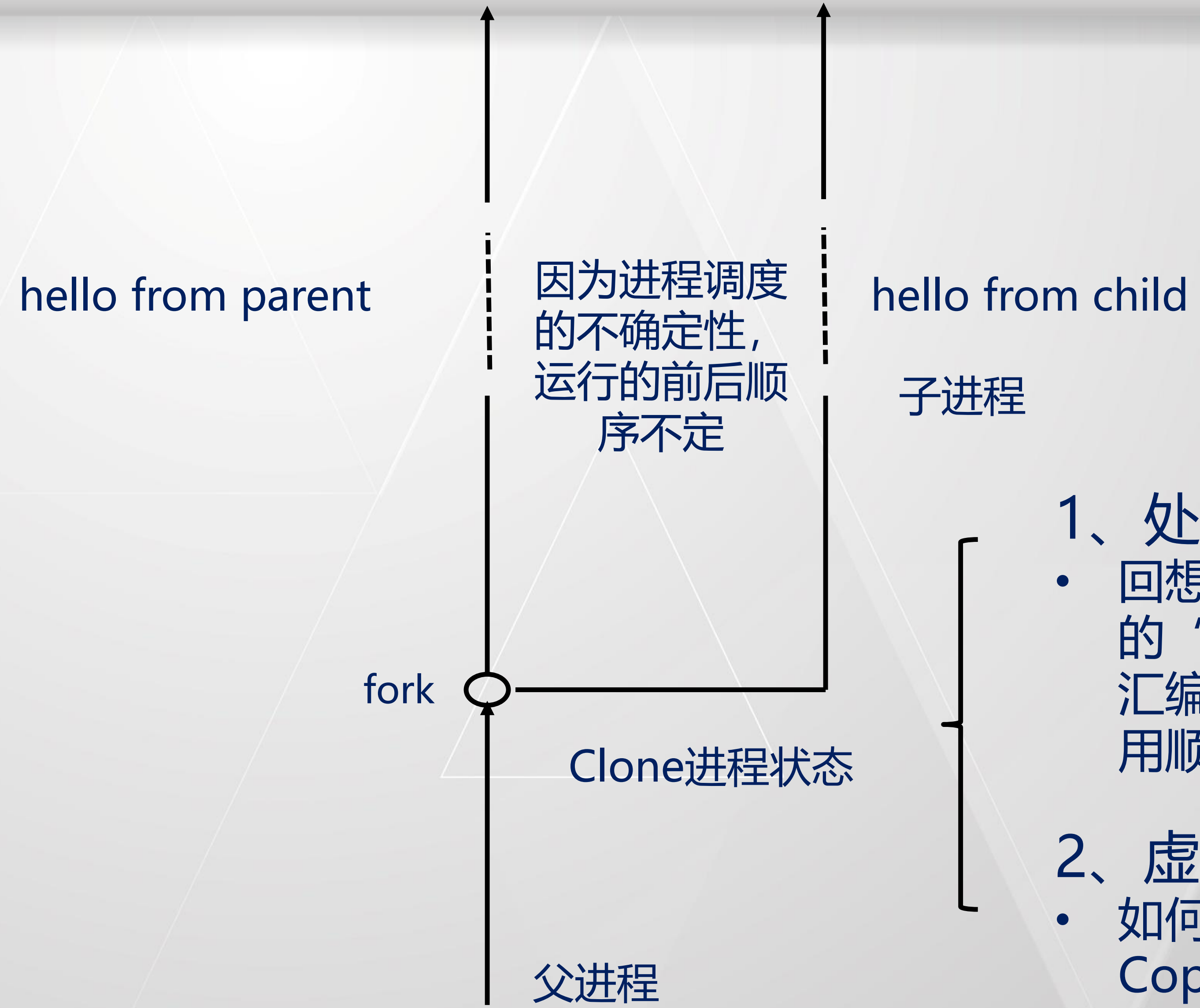
```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from child
```

Which one is first?

关于fork



1、处理器状态

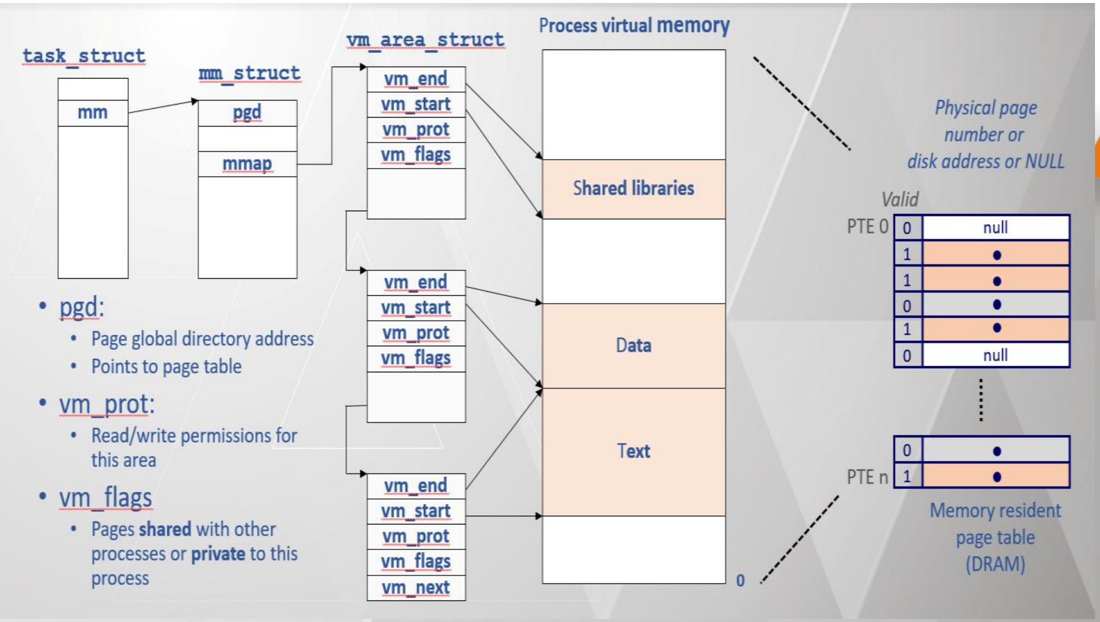
- 回想在“汇编语言”的函数调用部分所讲的“利用汇编以及对运行栈的理解来编写汇编过程打破‘过程返回的顺序恰好与调用顺序相反’这一惯例”这一示例

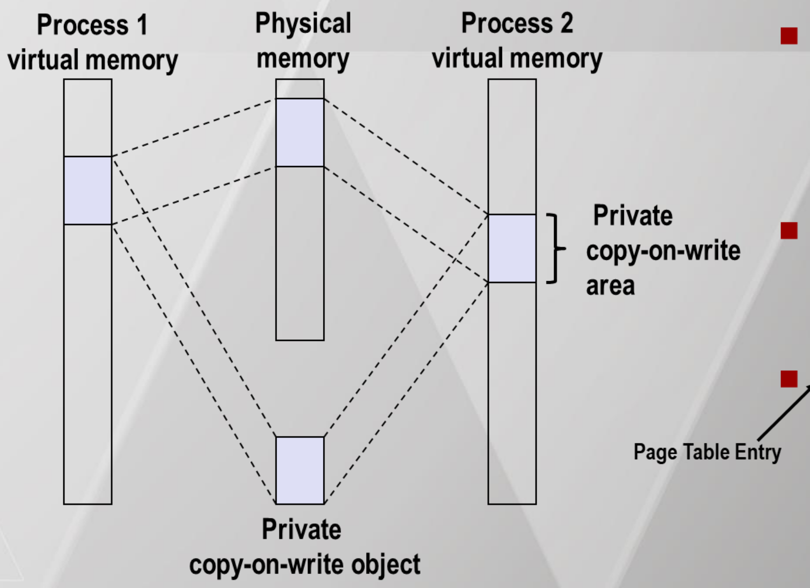
2、虚存

- 如何高效的复制？涉及到VM的特性：Copy-on-write

关于fork (补充知识)

- 上一讲中的VM和memory mapping可以解释fork如何为每个进程高效的提供私有地址空间
  - To create virtual address for new process
    - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
    - Flag each page in both processes as read-only
    - Flag each `vm_area_struct` in both processes as private COW
  - On return, each process has exact copy of virtual memory
  - Subsequent writes create new pages using COW mechanism.





- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private *copy-on-write*
- PTEs in private areas are flagged as read-only

Fork Example #1

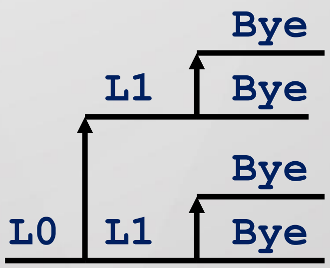
- **Parent and child both run same code**
  - Distinguish parent from child by return value from `fork`
- **Start with same state, but each has private copy**

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

- Both parent and child can continue forking

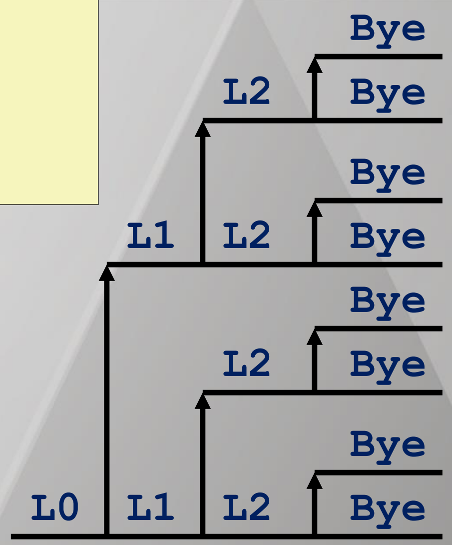
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



#3

- Both parent and child can continue forking

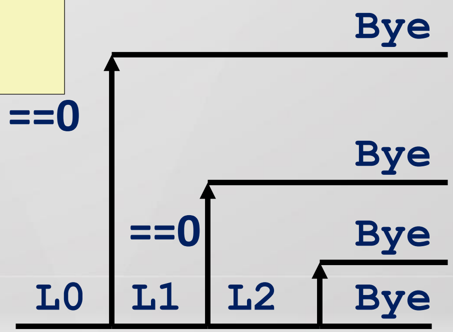
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

- Both parent and child can continue forking

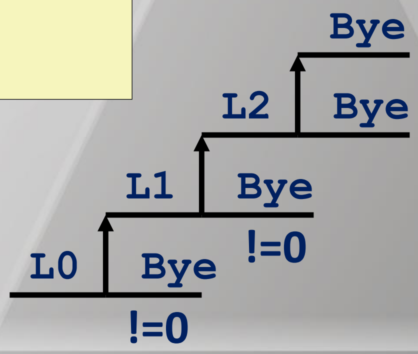
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
          printf("L2\n");
          fork();
       }
    }
    printf("Bye\n");
}
```



#5

- Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
          printf("L2\n");
          fork();
       }
    }
    printf("Bye\n");
}
```

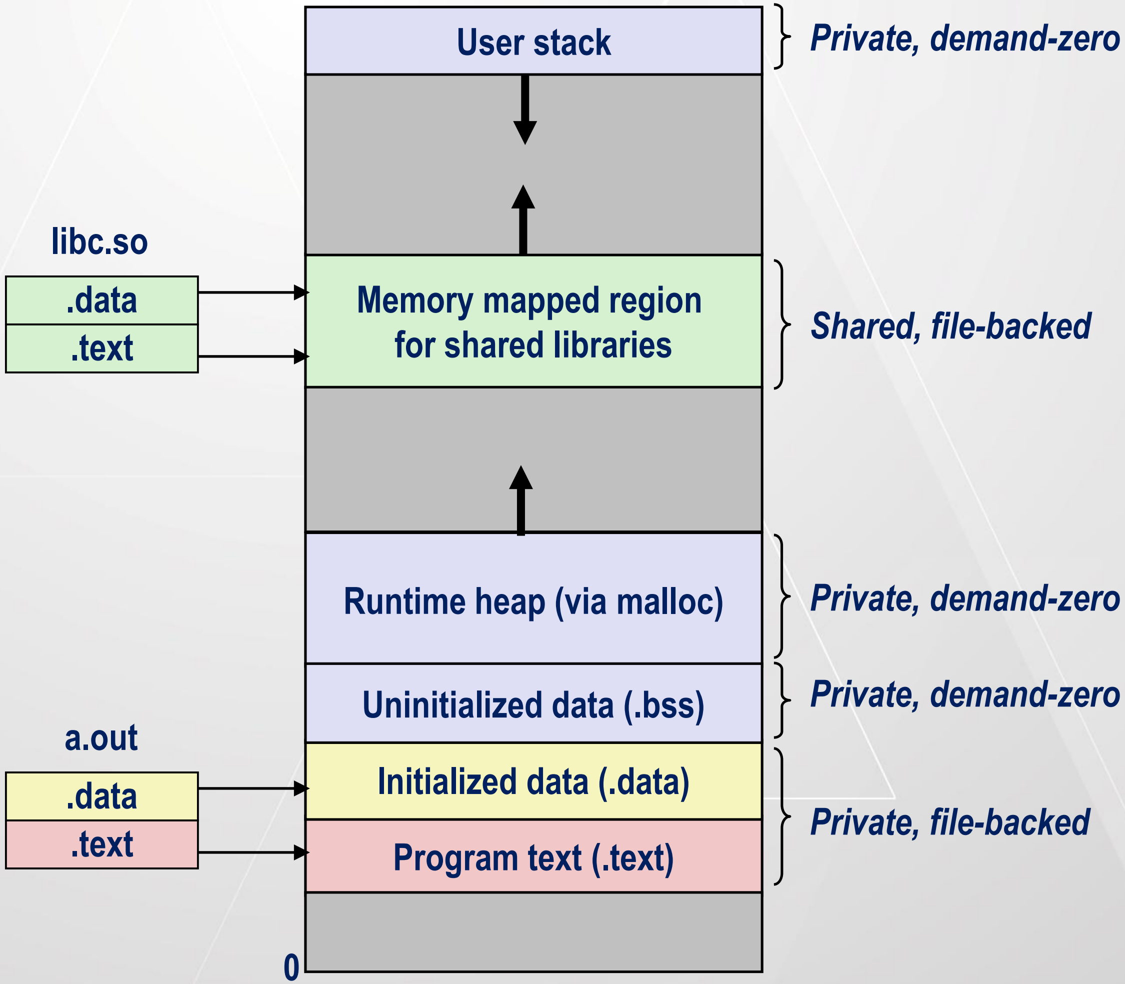


execve（） 执行特定程序

- ```
  int execve(
     char *filename,
     char *argv[],
     char *envp[]
  )
  ```
- **Loads and runs in current process:**
  - Executable **`filename`**
  - With argument list **`argv`**
  - And environment variable list **`envp`**
- **Does not return (unless error)**
- **Overwrites code, data, and stack**
  - keeps pid (进程号)
- **Environment variables:**
  - "name=value" strings
  - `getenv` and `putenv`

```
if ((pid = fork()) == 0) {
  if (execve(argv[0],argv,environ) < 0) {
        printf("Command not found.\n");
        exit(0);
    }
}
```
```

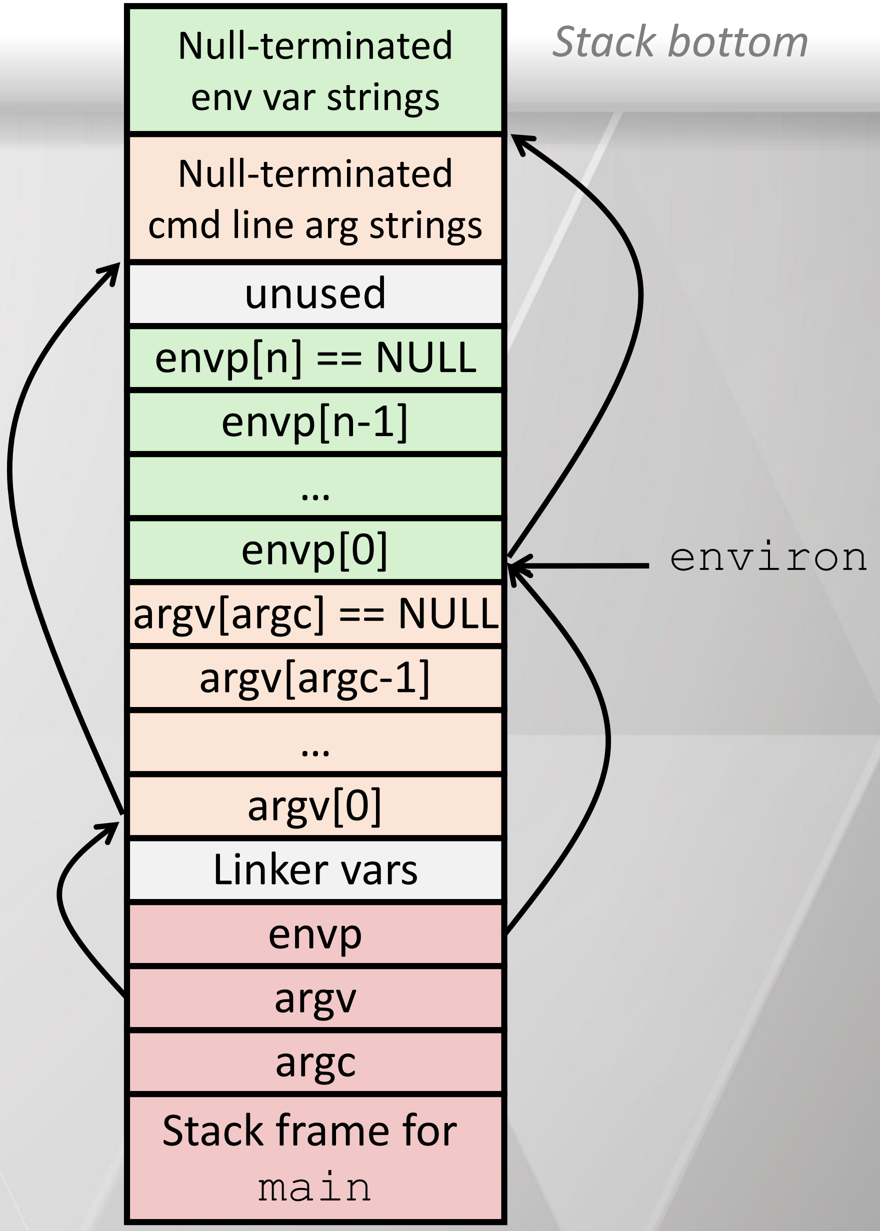
execve（） 执行特定程序



- To load and run a new program `a.out` in the current process using `execve`:
- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files .
- Set PC to entry point in `.text`
  - Linux will (page) fault in code and data pages as needed.

main函数的参数

回顾X86汇编 / MIPS32下的参数传递示例



exit：终止当前进程

- void exit(int status)
  - exits a process
    - Normally return with status 0
  - atexit() registers functions to be executed upon exit

status相当于进程正常通过main函数返回退出时，return的那个数值

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies （僵尸进程）

- **Idea**
  - 进程终止后仍会消耗系统资源
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead

- **Reaping**
  - 由父进程对终止的子进程进行reap
  - 父进程获得子进程的退出状态信息
  - 内核释放子进程的剩余资源

- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then child will be reaped by init process
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombies 示例

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- ***ps*** **shows child process as “defunct”**
- **Killing parent allows child to be reaped by** ***OS***

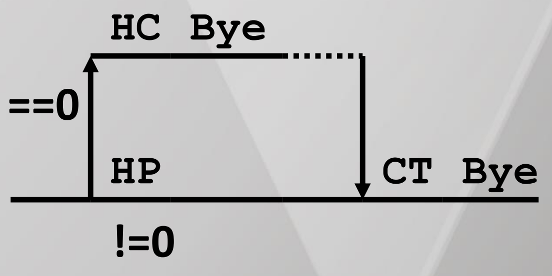
Reap子进程

wait: Synchronizing with Children

- `int wait(int *child_status)`
  - 挂起当前进程直到它的某个子进程终止
    - 返回值是终止的子进程的pid
  - if **`child_status != NULL`**, then the object it points to will be set to a status indicating why the child process terminated (获得子进程终止的原因，正常终止的话是子进程的退出状态值)
    - void exit(int status)

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}
```



wait() 示例

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));//退出值的低8位
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

Reap子进程之二：针对特定子进程

- `waitpid(pid, &status, options)`
  - suspends current process until specific process terminates
  - various options (Google ~~)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

```
int main() {
	if (fork() == 0)
	 {
		if (fork() == 0)
			printf("3");
		else
		 {
			pid_t pid; int status;
			if ((pid = wait(&status)) > 0)
				printf("4");
		}
	}
	else {
		if (fork() == 0)
		{
			printf("1");
			exit(0);
		}
		printf("2");
	}
	printf("0");
	return 0;
}
```

Out of the 5 outputs listed below, circle only the valid outputs of this program. Assume that all processes run to normal completion.

A. 2030401 B. 1234000 C. 2300140
D. 2034012 E. 3200410

wait调用：阻塞调用进程，直到它的一个子进程退出

```c
int counter = 0;
int main()
{
	int i;
	for (i = 0; i < 2; i ++){
		fork();
		counter ++;
		printf("counter = %d\n", counter);
	}
	printf("counter = %d\n", counter);
	return 0;
}
```

A. How many times would the value of counter be printed?
B. What is the value of counter printed in the first line?
C. What is the value of counter printed in the last line?

目录

MIPS32下的异常种类

异常	描述
Reset	由SI_ColdReset信号引起
Soft Reset	由SI_Reset信号引起
DSS	EJTAG调试单步异常
DINT	EJTAG调试中断异常，由外部的EJ_DINT输入引起，或由设置ECR寄存器中的EjtagBrk位引起
NMI	由SI_NMI信号引起(不可屏蔽中断)
Machine Check	TLB写操作与一个存在的表项冲突
Interrupt	由未被屏蔽的中断信号引起
DIB	EJTAG调试硬件指令断点匹配

异常	描述
WATCH	访问地址与观察寄存器中的地址匹配（取指时）
Deferred Watch	延迟的观察
AdEL	指令地址对齐错误，或用户模式下访问核心地址空间
TLBL	指令TLB缺失，指令TLB 无效（有效位为0）
IBE	取指时总线错误
DBp	EJTAG断点（执行SDBBP指令）
Sys	执行SYSCALL指令
Bp	执行BREAK指令
CpU	对一个未使能的协处理器执行协处理器指令

异常	描述
RI	执行保留指令
Ov	算术指令溢出
Tr	执行陷入指令（陷入条件为真时）
DDBL / DDBS	EJTAG数据地址断点（只对地址有效），或Store指令的EJTAG数据值断点（对地址和值有效）
WATCH	访问地址与观察寄存器中的地址匹配（访问数据时）
AdEL	读数据地址对齐错误，或用户模式下读核心地址空间数据
AdES	写数据地址对齐错误，或用户模式下写核心地址空间数据
TLBL	读数据时TLB缺失，或TLB无效（有效位为0）
TLBS	写数据时TLB缺失，或TLB无效（有效位为0）
TLB Mod	写TLB错误（写使能位为0）
DBE	读/写数据时总线错误
DDBL	EJTAG数据硬件断点与读指令读出的数据匹配

MIPS异常处理基本过程

保存现场

- 在异常程序入口，硬件只记录了被打断程序的很少量信息（EPC记录异常处理返回地址；Cause寄存器记录异常原因，其BD位记录branch delay slot信息；Status寄存器的EXL位被置1），同时需要保留相关的寄存器等值使得异常处理程序能够执行
  - k0 、k1寄存器保留给异常处理使用

判断不同的异常

- 查询Cause寄存器，根据其不同的异常原因来进行不同的处理

构造异常处理的内存空间

- 需要保存通用寄存器等

处理异常 ……

返回

- 恢复保存的寄存器，清零Cause寄存器，将Status寄存器的相关位 置1以开中断

异常返回

一般而言，异常处理代码工作在核心态，而被中断的程序是在用户态，所以异常返回意味着状态转换

这个转换与指令返回必须“同时”完成，即用一条指令完成。

为什么？

ERET指令

- 返回EPC指向的地址
- Status寄存器修改(EXL位置为0)

中断

中断是异步发生的，是来自处理器外部的I/O设备的信号引发的

硬件中断不是由任何一条专门的指令造成的

I/O设备，比如网络适配器，磁盘控制器等通过处理器芯片上的一个引脚发送信号，并将中断号放到系统总线上，用来触发中断，这个异常号标识了引起中断的设备

填空题 1分

设置

页缺失（page fault）是 [填空1] 异常，系统调用 [填空2] 异常。

正常使用填空题需3.0以上版本雨课堂

作答

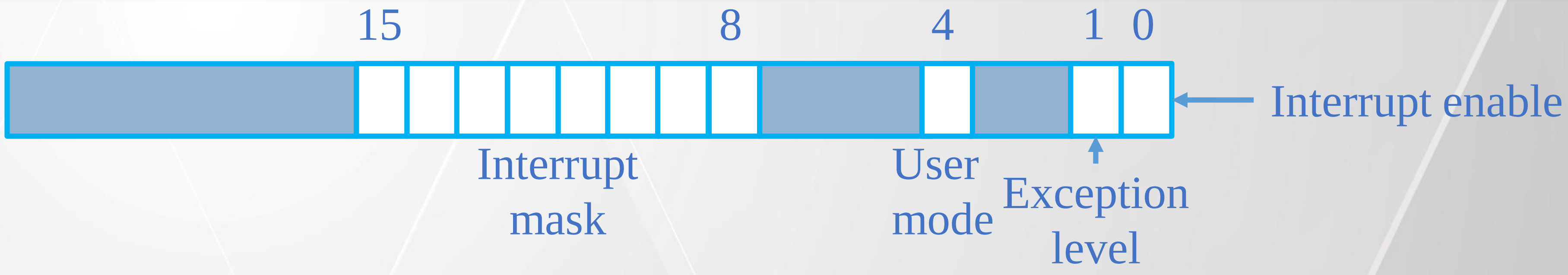
SPIM模拟器支持的异常处理流程

SPIM实现了部分CP0寄存器

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer (the counter increments every other clock.)
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

mfc0 与mtc0 可以访问这些寄存器

Status寄存器



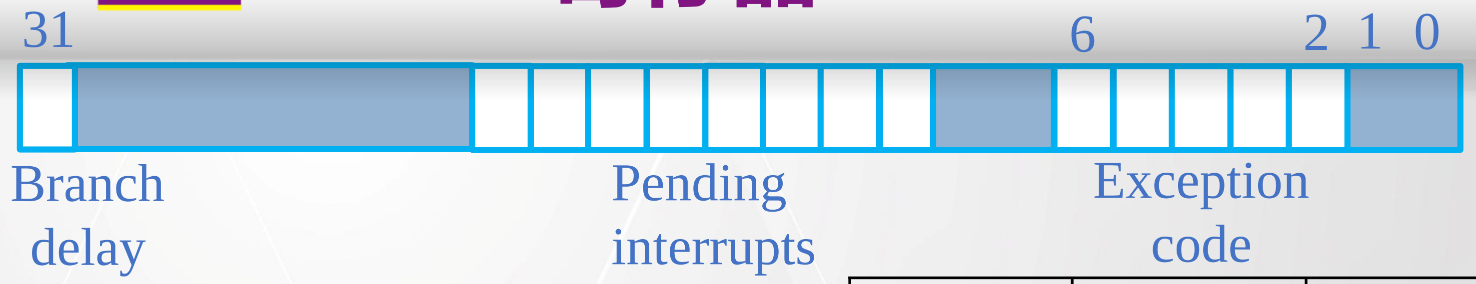
mask位为0：disable相应的中断（6个外部中断、2个内部中断）

user mode位为0表示运行于内核态；否则为用户态 （模拟器中固定为1）

exception level位 （EXL）：平时为0；当异常发生后被硬件置为1（此时屏蔽了中断处理，即阻止一个正在被处理的异常被中断），异常处理完毕后再被软件置为0（即ERET指令）

interrupt enable位：表示中断处理被禁止（0）或使能（1）

Cause寄存器



Branch delay位表示是否是在delay slot中的指令发生了异常

Interrupt pending中的某位为1，表示相应的中断发生（即pending，需要处理）

Exception code表示具体的异常原因（如右图所示）

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

异常处理实例

触发读数据地址不对齐异常（AdEL）

- 定位导致异常的指令（EPC 或者EPC+4，取决于Cause中的相关位）
- 解码该指令，取得异常数据地址并处理（默认异常处理入口地址：0x80000180）
- 如何返回?

如果位于Branch Delay Slot中的指令触发异常，EPC则被设为该Branch指令的地址

SPIM的异常处理程序

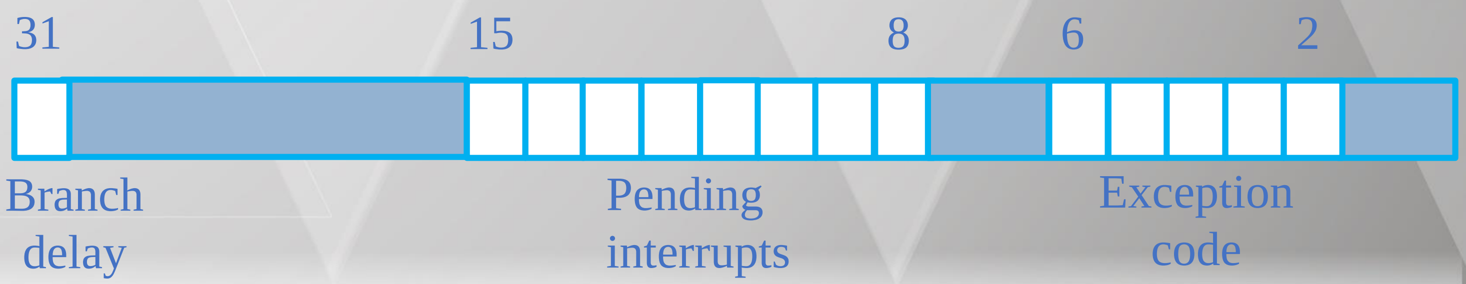
```
.kdata
__m1_: .asciiz "  Exception "
__m2_: .asciiz " occurred and ignored\n"
__e0_: .asciiz "  [Interrupt] "
__e1_: .asciiz "  [TLB]"
__e2_: .asciiz "  [TLB]"
__e3_: .asciiz "  [TLB]"
__e4_: .asciiz "  [Address error in inst/data fetch] "
__e5_: .asciiz "  [Address error in store] "
__e6_: .asciiz "  [Bad instruction address] "
__e7_: .asciiz "  [Bad data address] "
__e8_: .asciiz "  [Error in syscall] "
__e9_: .asciiz "  [Breakpoint] "
__e10_: .asciiz "  [Reserved instruction] "
__e11_: .asciiz ""
__e12_: .asciiz "  [Arithmetic overflow] "
__e13_: .asciiz "  [Trap] "
__e14_: .asciiz ""
__e15_: .asciiz "  [Floating point] "
__e16_: .asciiz ""
__e17_: .asciiz ""
__e18_: .asciiz "  [Coproc 2]"
__e19_: .asciiz ""
__e20_: .asciiz ""
__e21_: .asciiz ""
__e22_: .asciiz "  [MDMX]"
__e23_: .asciiz "  [Watch]"
__e24_: .asciiz "  [Machine check]"
__e25_: .asciiz ""
__e26_: .asciiz ""
__e27_: .asciiz ""
__e28_: .asciiz ""
__e29_: .asciiz ""
__e30_: .asciiz "  [Cache]"
__e31_: .asciiz ""
__excp:
.word __e0_, __e1_, __e2_, __e3_, __e4_, __e5_,
__e6_, __e7_, __e8_, __e9_
.word __e10_, __e11_, __e12_, __e13_, __e14_,
__e15_, __e16_, __e17_, __e18_,
.word __e19_, __e20_, __e21_, __e22_, __e23_,
__e24_, __e25_, __e26_, __e27_,
.word __e28_, __e29_, __e30_, __e31_
save1: .word 0
save2: .word 0
```

```
# This is the exception handler code that the processor runs when
# an exception occurs. It only prints some information about the
# exception, but can server as a model of how to write a handler.
#
# Because we are running in the kernel, we can use $k0/$k1 without
# saving their old values.

.set noat
move $k1, $at # Save $at
.set at
sw $v0, save1    # Not re-entrant and we can't trust $sp
sw $a0, save2    # But we need to use these registers

mfc0 $k0, $13  # Cause register

# Print information about exception.
#
li $v0, 4       # syscall 4 (print_str)
la $a0, __m1_
syscall
```



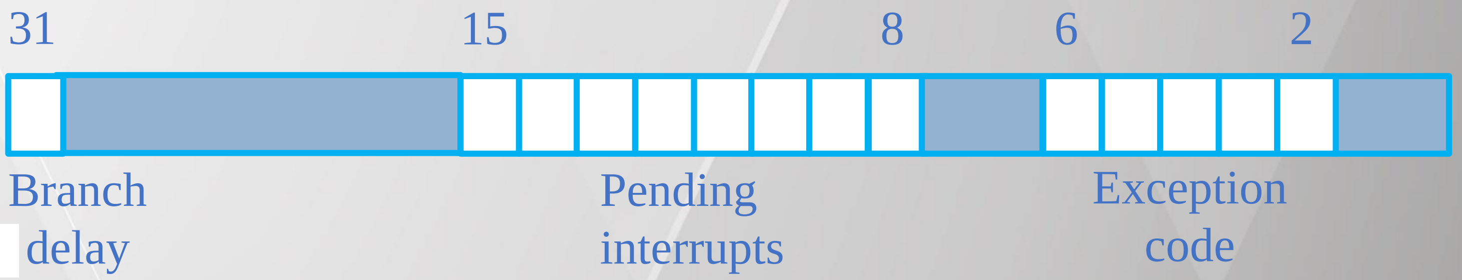
```
li $v0, 1          # syscall 1 (print_int)
srl $a0, $k0, 2   # Extract ExcCode Field
andi $a0, $a0, 0x1f
syscall

li $v0, 4          # syscall 4 (print_str)
andi $a0, $k0, 0x3c    # 0x3c = 11 1100
lw $a0, __excp($a0)
nop
syscall

bne $k0, 0x18, ok_pc
nop                # Bad PC exception (IBE) requires special checks

mfc0 $a0, $14            # EPC
andi $a0, $a0, 0x3     # Is EPC word-aligned?
beq $a0, 0, ok_pc
nop

li $v0, 10               # Exit on really bad PC
syscall
```



```
ok_pc:
li   $v0, 4# syscall 4 (print_str)
la   $a0, __m2_
syscall

srl  $a0, $k0, 2# Extract ExcCode Field
andi $a0, $a0, 0x1f
bne  $a0, 0, ret# 0 means exception was an interrupt
nop

# Interrupt-specific code goes here!
# Don't skip instruction at EPC since it has not executed.

ret:
# Return from (non-interrupt) exception.
mfc0  $k0, $14     # Bump EPC register
addiu $k0, $k0 4   # Skip faulting instruction
                   # (Need to handle delayed branch case here)
mtc0  $k0, $14
```

```
# Restore states
lw $v0, save1   #Restore registers
lw $a0, save2

.set noat
move $at, $k1 #Restore $at
.set at

mtc0 $0, $13  #Clear Cause register

mfc0 $k0, $12 #Set Status register
ori $k0, 0x1  #Interrupts enabled
mtc0 $k0, $12

eret
nop
```