第八章 进程与异常控制流

进程与进程的上下文切换

异常和中断的响应和处理

Linux中的进程控制

Linux中的信号与非本地跳转

进程与异常控制流

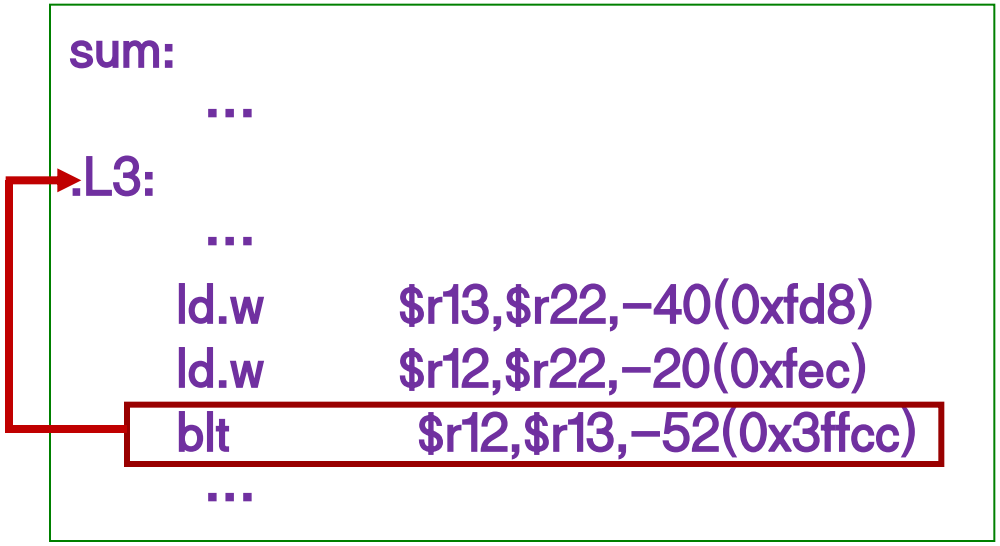
- 主要教学目标
  - 了解程序执行过程中正常的控制流和异常控制流的区别
  - 了解在较低层次上如何实现异常控制流
  - 初步理解硬件如何和操作系统协调工作，从而为将来理解和掌握操作系统核心内容打下良好基础。
- 主要教学内容
  - CPU控制流、异常控制流
  - 进程和进程上下文切换
  - 异常和中断的响应和处理
  - Linux中的进程控制
  - Linux中的信号与非本地跳转

进程与异常控制流

- 分以下四个部分介绍
  - 第一讲：进程与进程的上下文切换
    - 程序和进程的概念
    - 进程的逻辑控制流
    - 进程与进程的上下文切换
  - 第二讲：异常和中断
    - 异常和中断的基本概念、异常和中断的响应和处理
  - 第三讲：LoongArch+Linux下的异常/中断机制
  - 第四讲：Linux中的进程控制与信号处理
    - 进程的创建、休眠和终止
    - 进程ID的获取和子进程的回收
    - Linux中的信号处理机制
    - 非本地跳转处理

回顾：LA32/LA64跳转指令举例

```
int sum(int a[ ], int len)
{
   int  i,  s = 0;
   for (i = 0; i <= len–1; i++)
        s += a[i];
   return s;
}
```



程序的正常执行顺序有哪两种?

(1) 按顺序取下一条指令执行

(2) 通过BL/B/Bxx/JIRL等指令跳转到目标地址处执行

CPU所执行的**指令的地址序列**称为**CPU的控制流**，通过上述两种方式得到的控制流为**正常控制流**。

异常控制流

- CPU会因为遇到内部异常或外部中断等原因而打断程序的正常控制流，转去执行操作系统提供的针对这些特殊事件的处理程序。
- 由于某些特殊情况引起用户程序的正常执行被打断所形成的意外控制流称为异常控制流（Exceptional Control of Flow, ECF）。
- 异常控制流的形成原因:
  - 内部异常（缺页、越权、越级、整除0、溢出等）
  - 外部中断（Ctrl-C、打印缺纸、DMA结束等）

  } 发生在硬件层

  - 进程的上下文切换（发生在操作系统层）
  - 一个进程直接发送信号给另一个进程（发生在应用软件层）

“程序”和“进程”

程序（program）指按某种方式组合形成的代码和数据集合，代码即是机器指令序列，因而程序是一种静态概念。

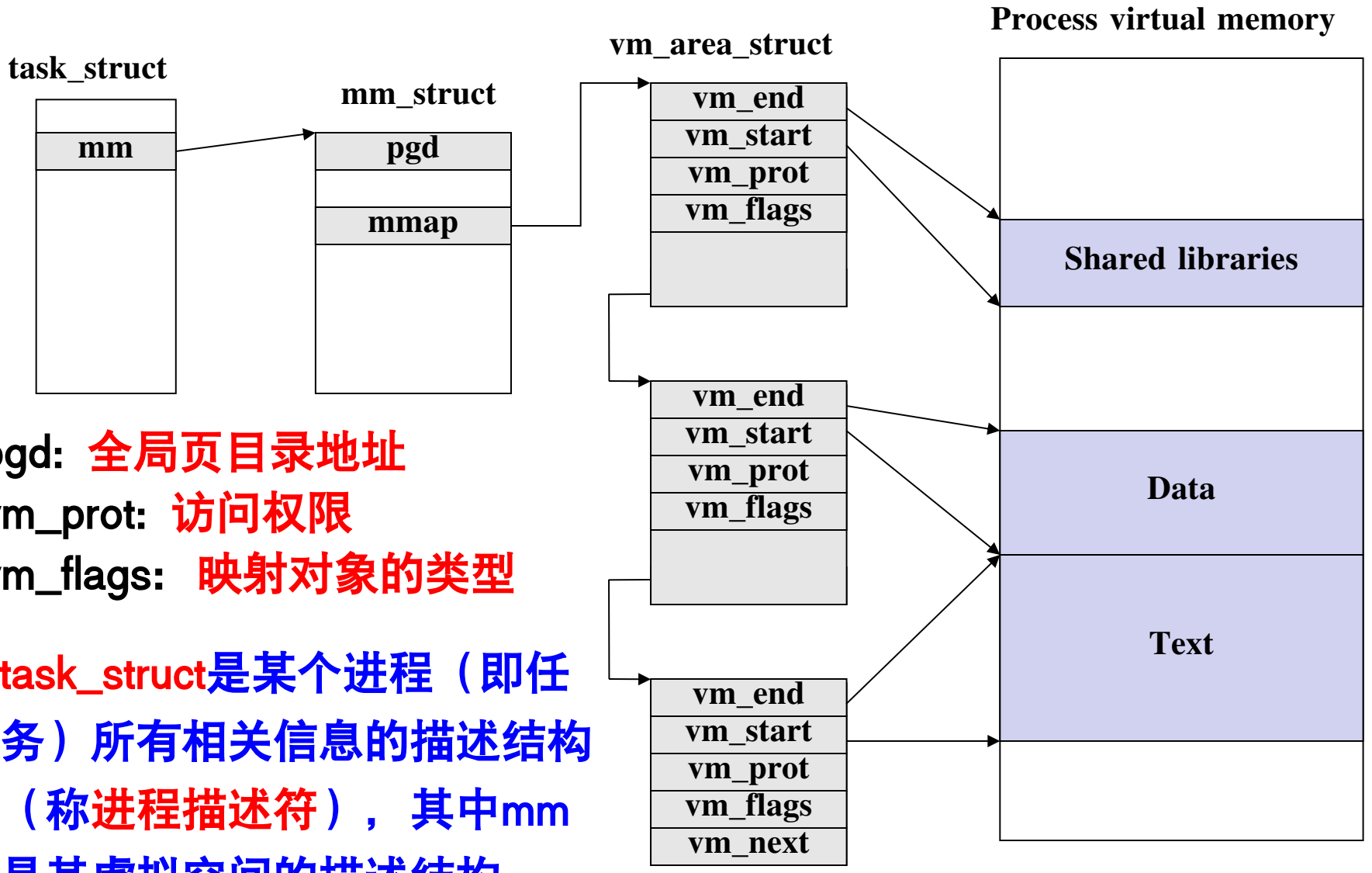
进程（ process）指程序的一次运行过程。更确切说，进程是具有独立功能的一个程序关于某个数据集合的一次运行活动，因而进程具有动态含义 。同一个程序处理不同的数据就是不同的进程

- 进程是OS对CPU执行的程序的运行过程的一种抽象。进程有自己的生命周期，它由于任务的启动而创建，随着任务的完成（或终止）而消亡，它所占用的资源也随着进程的终止而释放。
- 一个可执行目标文件（即程序）可被加载执行多次，也即，一个程序可能对应多个不同的进程。
  - 例如，用word程序编辑一个文档时，相应的用户进程就是winword.exe，如果多次启动同一个word程序，就得到多个winword.exe进程，处理不同的数据。

进程的概念

- 操作系统（管理任务）以外的都属于“用户”的任务。
- 计算机处理的所有“用户”的任务由进程完成。
- 为强调进程完成的是用户的任务，通常将进程称为用户进程。
- 计算机系统中的任务通常就是指进程。例如，
  - Linux内核中通常把进程称为任务，每个进程主要通过一个称为进程描述符（process descriptor）的结构来描述，其结构类型定义为task_struct，包含了一个进程的所有信息。
  - 所有进程通过一个双向循环链表实现的任务列表（task list）来描述，任务列表中每个元素是一个进程描述符。
  - x86中的任务状态段（TSS）、任务门（task gate）等概念中所称的任务，实际上也是指进程。

回顾：Linux将虚存空间组织成“区域”的集合



pgd: 全局页目录地址

vm_prot: 访问权限

vm_flags: 映射对象的类型

task_struct是某个进程（即任务）所有相关信息的描述结构（称进程描述符），其中mm是其虚拟空间的描述结构

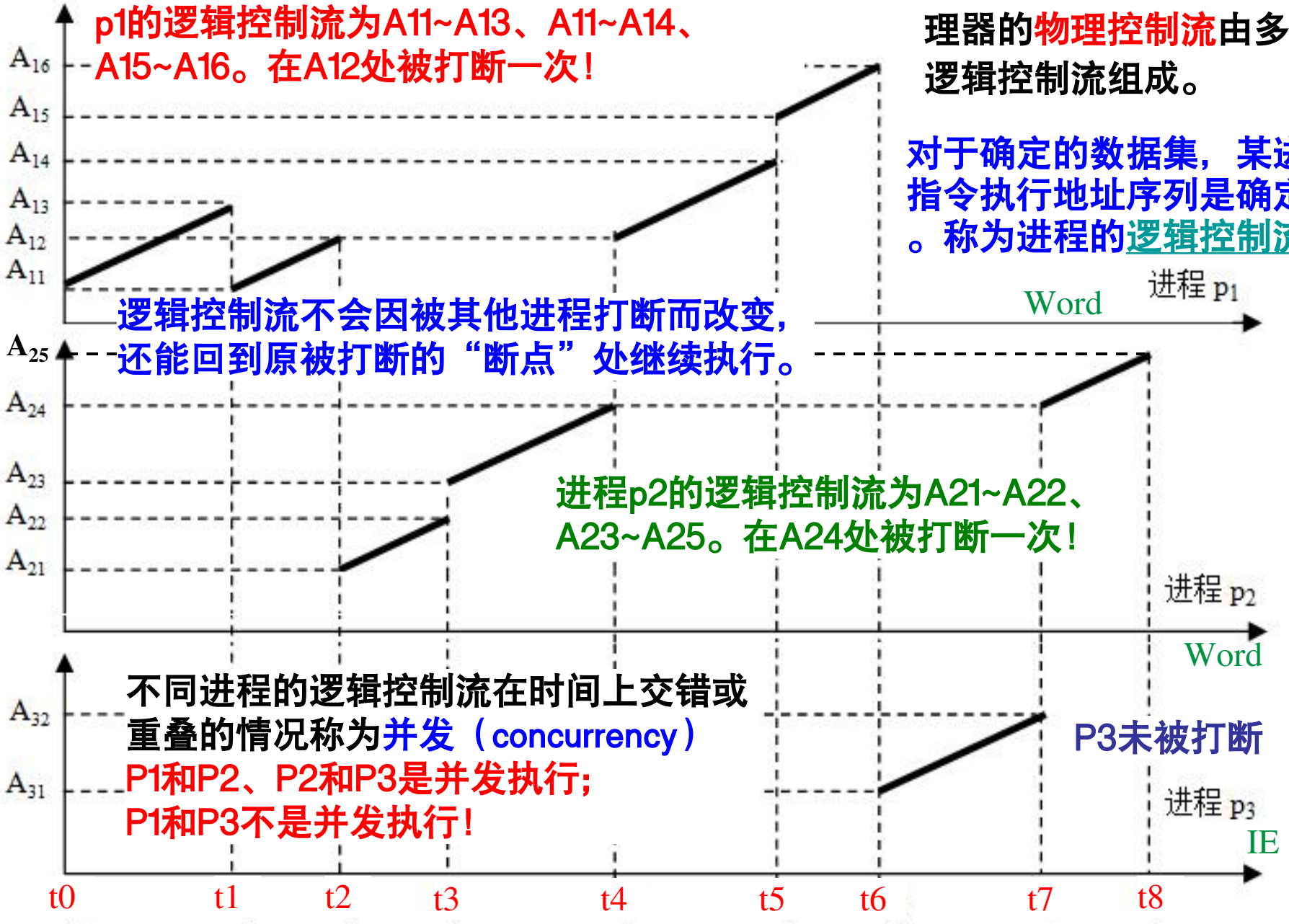
引入“进程”的好处

- “进程”的引入为应用程序提供了以下两方面的抽象:
  - 一个独立的逻辑控制流
    - 每个进程拥有一个独立的逻辑控制流，使得程序员以为自己的程序在执行过程中独占使用处理器
  - 一个私有的虚拟地址空间
    - 每个进程拥有一个私有的虚拟地址空间，使得程序员以为自己的程序在执行过程中独占使用存储器
- 进程”的引入简化了程序员的编程以及语言处理系统的处理，即简化了编程、编译、链接、共享和加载等整个过程。

逻辑控制流

对于单处理器系统，进程会轮流使用处理器，即处理器的物理控制流由多个逻辑控制流组成。

对于确定的数据集，某进程指令执行地址序列是确定的。称为进程的逻辑控制流。



“进程” 与“上下文切换”

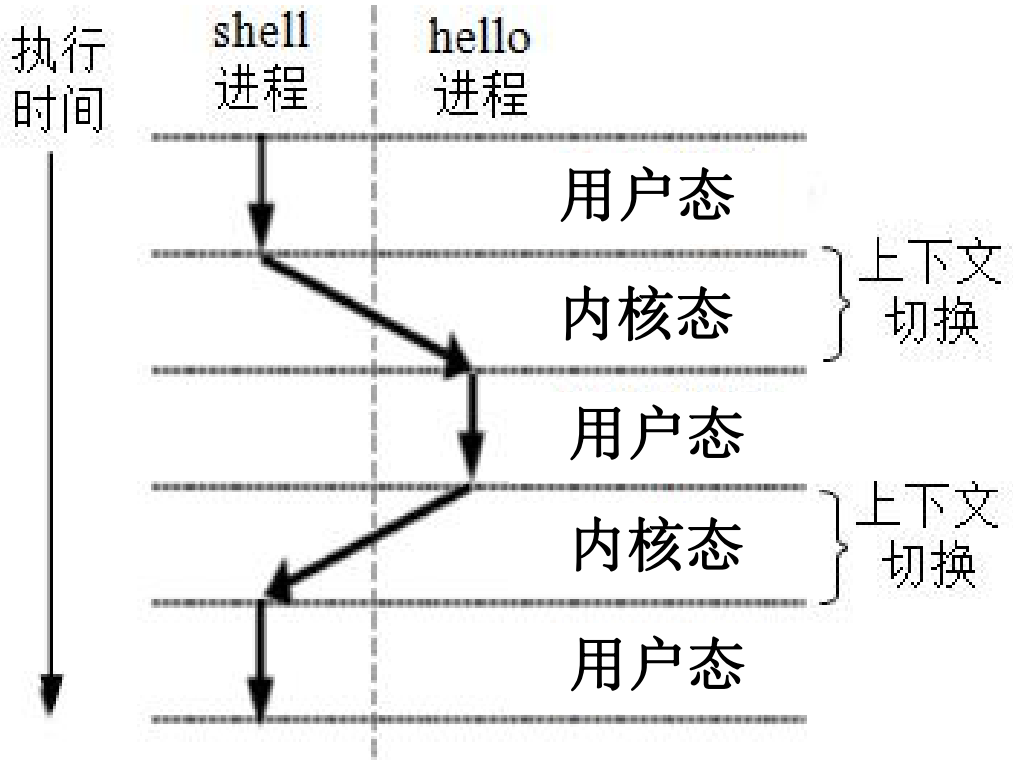
OS通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为进程的上下文切换（context switching）

```
$./hello
hello，world
$
```

“$”是shell命令行提示符，说明正在运行shell进程。

在一个进程的生命周期中，可能会有其他不同进程在处理器上交替运行！

感觉到的运行时间比真实执行时间要长！



处理器调度等事件会引起用户进程正常执行被打断，因而形成异常控制流。进程的上下文切换机制很好地解决了这类异常控制流，实现了从一个进程安全切换到另一个进程执行的过程。

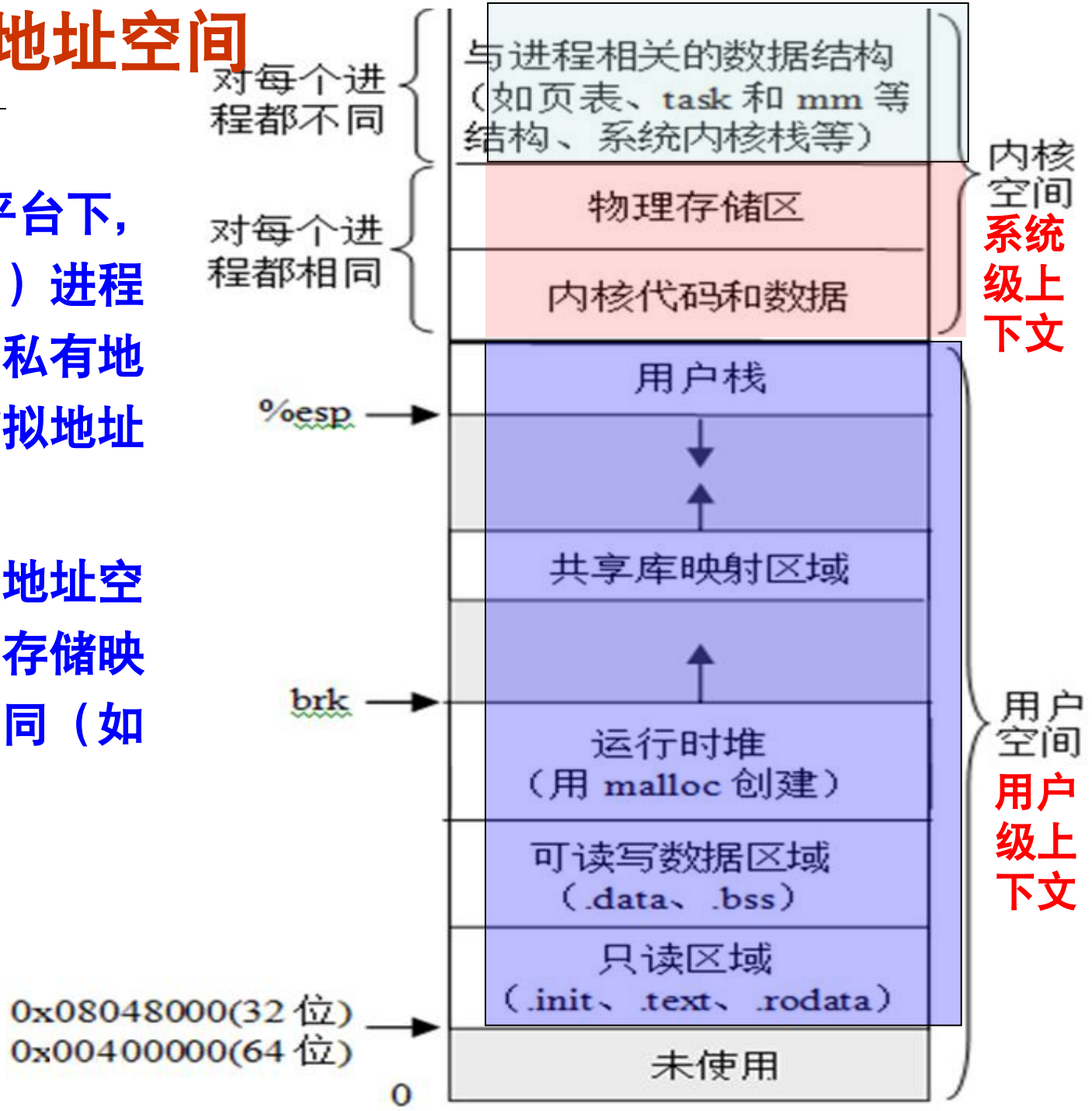
“进程”的“上下文”

- 进程的物理实体（代码和数据等）和支持进程运行的环境合称为**进程的上下文**。
- 由进程的程序块、数据块、运行时的堆和用户栈（两者通称为用户堆栈）等组成的用户空间信息被称为**用户级上下文**;
- 由进程标识信息、进程现场信息、进程控制信息和系统内核栈等组成的内核空间信息被称为**系统级上下文**;
- 处理器中各寄存器的内容被称为**寄存器上下文**（也称**硬件上下文**），即进程的现场信息。
- 在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文中的现场信息位置。
- 用户级上下文地址空间和系统级上下文地址空间一起构成了**一个进程的整个存储器映像**

进程的存储器映像

回顾：进程地址空间

- x86+Linux平台下，每个（用户）进程具有独立的私有地址空间（虚拟地址空间）
- 每个进程的地址空间划分（即存储映像）布局相同（如右图）



进程与异常控制流

- 分以下四个部分介绍
  - 第一讲：进程与进程的上下文切换
    - 程序和进程的概念
    - 进程的逻辑控制流
    - 进程与进程的上下文切换
  - 第二讲：异常和中断
    - 异常和中断的基本概念、异常和中断的响应和处理
  - 第三讲：LoongArch+Linux下的异常/中断机制
  - 第四讲：Linux中的进程控制与信号
    - 进程的创建、休眠和终止
    - 进程ID的获取和子进程的回收
    - Linux中的信号处理机制
    - 非本地跳转处理

回顾：LA页表映射模式下的TLB访问

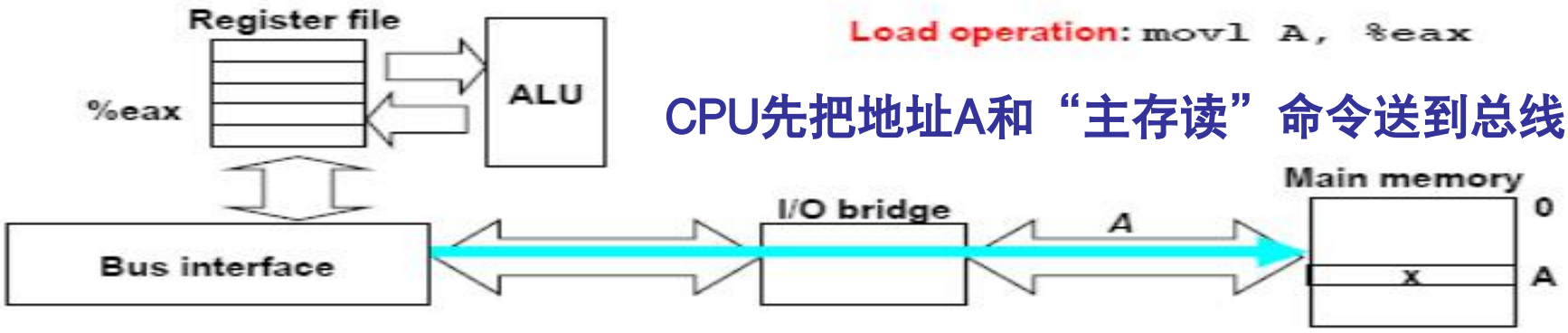
4）基于TLB的虚实地址转换过程中的异常事件

MMU通过TLB进行虚实地址转换过程中，若没找到匹配的TLB表项，或者尽管有匹配的TLB表项，但其中的页表项无效（V=0）或访问权限和特权级等不相符，则会触发异常，从而转到操作系统内核或其它监管程序进行异常处理

- MTLB和STLB都miss时，触发TLB重填（TLBR）异常
- 若取指令、Load取数、Store存数的地址转换过程发生TLB命中但V=0，则分别触发取指令页无效（PIF）、Load页无效（PIL）和Store页无效（PIS）三种异常
- 若TLB命中且对应V=1，但其特权级不合规，则触发页特权级不合规（PPI）异常页特权级不合规条件：页表项RPLV=0且当前特权级（CSR.CRMD.PLV）大于页表项PLV值，或者 页表项RPLV=1且当前特权级不等于页表项PLV值
- 若Store操作时TLB命中且对应V=1、特权级合规，但在当前特权级为PLV3或当前特权级不是PLV3但对应特权级的CSR.MISC.DWPL=0（未开启禁止写允许检查功能）的前提条件下，页表项D=0，则触发页修改（PME）异常
- 当Load操作时TLB命中且对应V=1、特权级合规，但该页表项中NR=1，将触发页不可读（PNR）异常
- 当取指令操作时TLB命中且对应V=1、特权级合规，但该页表项中NX=1，将触发页不可执行（PNX）异常

回顾：x86指令“movl 8(%ebp), %eax”操作过程

由8(%ebp)得到主存地址A的过程较复杂，涉及MMU、TLB、页表等许多重要概念！



- IA-32中，执行“movl 8(%ebp)，%eax” 中取数操作的大致过程如下：
  - 若CPL>DPL则越级，否则计算有效地址EA=R[ebp]+0×0+8
  - 通过段寄存器找到段描述符以获得段基址，线性地址LA=段基址+EA
  - 若“LA>段限”则越界，否则将LA转换为主存地址A
    - 若访问TLB命中则地址转换得到A；否则处理TLB缺失（硬件/OS）
    - 若缺页或越权(R/W不符)则调出OS内核；否则地址转换得到A
    - 根据A先到Cache中找，若命中则取出A在Cache中的副本
    - 若Cache不命中，则再到主存取A所在主存块送对应Cache行

打断程序正常执行的事件

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被中止的程序处（断点）继续执行。
- 程序执行被 “中断” 的事件（在硬件层面）有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件

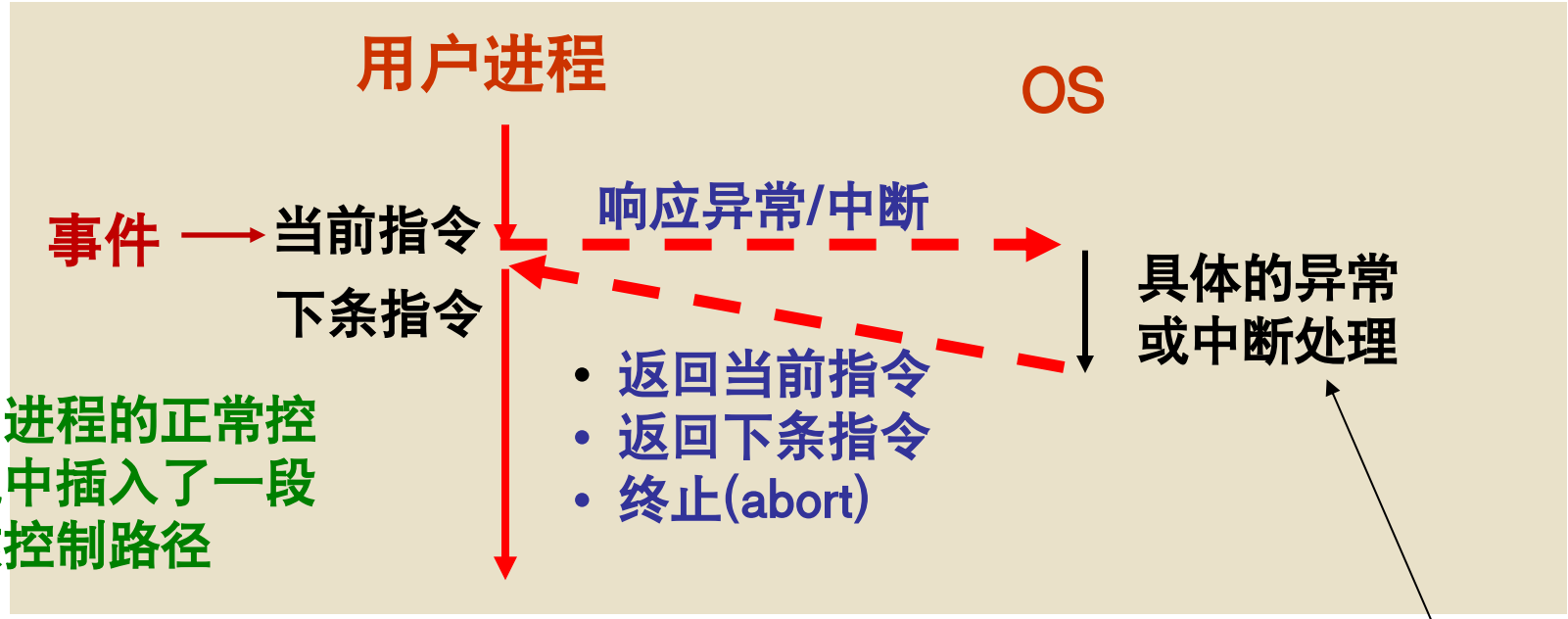
    按发生原因分为硬故障中断和程序性中断两类

    硬故障中断：硬件线路故障等

    程序性中断：执行某条指令时发生的“例外(Exception)”事件，如溢出、缺页、越界、越权、越级、非法指令、除数为0、堆/栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。

异常和中断的处理

- 发生异常(exception)和中断(interrupt)事件后，系统将进入OS内核态对相应事件进行处理，即改变处理器状态（用户态→内核态）



中断或异常处理执行的代码不是一个进程，而是“内核控制路径”，它代表异常或中断发生时正在运行的当前进程在内核态执行一个独立的指令序列。内核控制路径比进程更“轻”，其上下文信息比进程上下文信息少得多。而上下文切换后CPU执行的是另一个用户进程。

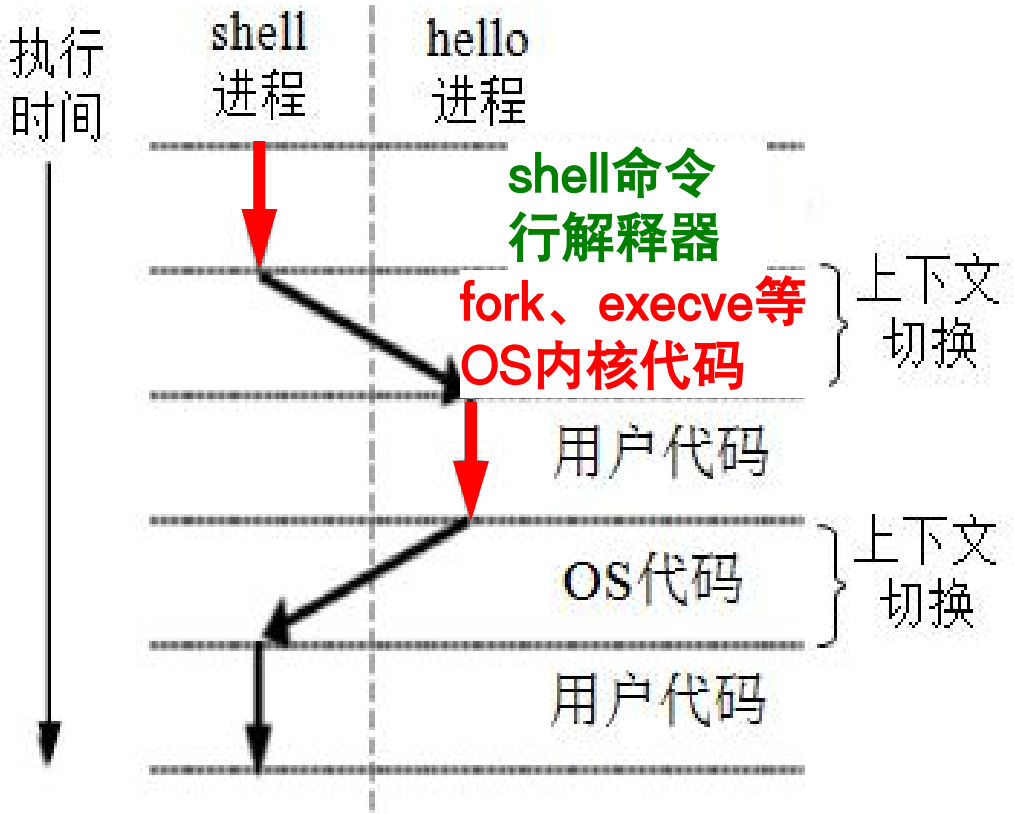
回顾：“进程” 与“上下文切换”

OS通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为进程的上下文切换（context switching）

```
$./hello
hello，world
$
```

“$”是shell命令行提示符，说明正在运行shell进程。

上下文切换后，是另一个用户进程！



处理器调度等事件会引起用户进程正常执行被打断，因而形成异常控制流。进程的上下文切换机制很好地解决了这类异常控制流，实现了从一个进程安全切换到另一个进程执行的过程。

异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)：执行指令引起的异常事件，如溢出、非法指令、缺页、访问越权等。 “断点”为发生故障指令的地址

自陷(Trap)：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用（执行访管指令）等。是一种自愿中断。

“断点”为自陷指令下条指令地址

终止(Abort)：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点”是什么？ 随便！

思考1：自陷处理完成后回到哪条指令执行？ 回到下条指令

思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法指令、内存保护错等：终止当前进程。

“断点”：异常处理结束后回到原来被“中断”的程序执行时的起始指令。

异常举例—页故障

“页故障”事件何时发现？如何发现？

执行每条指令都要访存（取指令、取操作数、存结果）

在保护模式下，每次访存都要进行逻辑地址向物理地址转换

在地址转换过程中会发现是否发生了“页故障”！

“页故障”事件是软件发现的还是硬件发现的？

逻辑地址向物理地址的转换由硬件（MMU）实现，故“页故障”事件由硬件发现。所有异常和中断事件都由硬件检测发现！

- 以下几种情况都会发生“页故障”
  - 缺页：页表项有效(存在)位为0 ⟵ 可通过读磁盘恢复故障
  - 地址越界：地址大于最大界限
  - 访问越级或越权（保护违例）：

  } 不可恢复，称为“段故障（segmentation fault）”

    - 越级：用户进程访问内核数据（CPL=3 / DPL=0）
    - 越权：读写权限不相符（如对只读段进行了写操作）

异常举例—页故障

假设在LA64+Linux系统中一个C语言源程序 P 如下：

```
1   int  a[1000];
2   int  x;
3   main( ) {
4       a[10]=1;
5       a[1000]=3;
6       a[10000]=4;
7   }
```

正常的控制流为

…、1200006a4、 1200006a8、 1200006ac 、…

可能的异常控制流是什么？

假设编译、汇编和链接后，第4、5行源代码对应的指令序列如下：

```
1    1200006a4:   1c00010c      pcaddu12i $r12, 8(0x8)
2    1200006a8:   28e6318c      ld.d      $r12, $r12, -1652(0x98c)
3    1200006ac:   0280040d      addi.w    $r13, $r0, 1(0x1)
4    1200006b0:   2980a18d      st.w      $r13, $r12, 40(0x28)
5    1200006b4:   1c00010c      pcaddu12i $r12, 8(0x8)
6    1200006b8:   28e5f18c      ld.d      $r12, $r12, -1668(0x97c)
7    1200006bc:   02800c0d      addi.w    $r13, $r0, 3(0x3)
8    1200006c0:   250fa18d      stptr.w   $r13, $r12, 4000(0xfa0
9    1200006c4:   1c00010d      pcaddu12i $r13, 8(0x8)
10   1200006c8:   28e5b1ad      ld.d      $r13, $r13, -1684(0x96c)
11   1200006cc:   1400014c      lu12i.w   $r12, 10(0xa)
12   1200006d0:   0010b1ac      add.d     $r12, $r13, $r12
13   1200006d4:   0280100d      addi.w    $r13, $r0, 4(0x4)
14   1200006d8:   29b1018d      st.w      $r13, $r12, -960(0xc40)
```

异常举例—页故障

假定采用分页虚拟存储管理方式，页大小为16KB。若在运行该程序对应的进程P时，系统中没有其他进程在运行，则上述14条指令的取指令操作是否会发生缺页异常？若进程P运行时在虚拟地址0x1 2000 8030开始的8个单元中存放的是0x0000 0001 2000 8060（即数组a的起始虚拟地址0x0000 0001 2000 8060存放在0x1 2000 8030开始的8字节中），则在执行上述14条指令中的Load/Store操作对应的页表遍历过程中，哪些指令会发生页故障？哪些页故障是可恢复的？哪些是不可恢复的？

解：这14条指令的取指令操作都不会发生缺页异常。因为这些指令执行前已经随前面某条指令一起装入了主存，且不会被调出主存。

第2行Load指令取数操作页表遍历过程中会发生页故障，具体故障类型是缺页，可恢复。操作数地址为0x1 2000 06a4+0x8000+SEXT(0x98c) = 0x1 2000 8030.

第4行Store指令存数操作页表遍历过程中不会发生页故障。因为第2行ld.d指令执行后r12的内容为0x0000 00001 2000 8060,第4行st.t指令的访问地址为0x1 2000 8060+0x28=0x1 2000 8088，与第2行指令的访问地址x1 2000 8030位于同一页，所以在这条指令执行前，该页已被装入主存。

第6行Load指令取数操作页表遍历过程中不会发生页故障，因为该操作访问的地址为0x0000 0001 2000 06b4 + 0x0000 0000 0000 8000 + 0xffff ffff ffff f97c = x10000 0001 2000 8030，该地址在第2行指令执行时被访问过，故不会发生页故障。

同理，第8行Store指令和第10行Load指令的执行都不会发生页故障。

第14行Store指令存数操作页表遍历过程中可能发生页故障，且不可恢复。数据地址为0x1 2000 8060 + 0xa000 + SEXT(c40) = 0x1 2001 1ca0，该地址偏离数组a所在页首地址0x1 2000 8000达0x1 2001 1ca0 − 0x1 2000 8000 = 0x9ca0个单元，偏离了2个页面，可能超出了可读写数据区范围，因而可能发生地址越界或访问越权

异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。 “断点”为发生故障指令的地址

自陷(Trap)：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用（执行访管指令）等。是一种自愿中断。
“断点”为自陷指令下条指令地址

终止(Abort)：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点”是什么？ 随便！

思考1：自陷(陷阱、陷入)处理完成后回到哪条指令执行？ 回到下条指令

思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

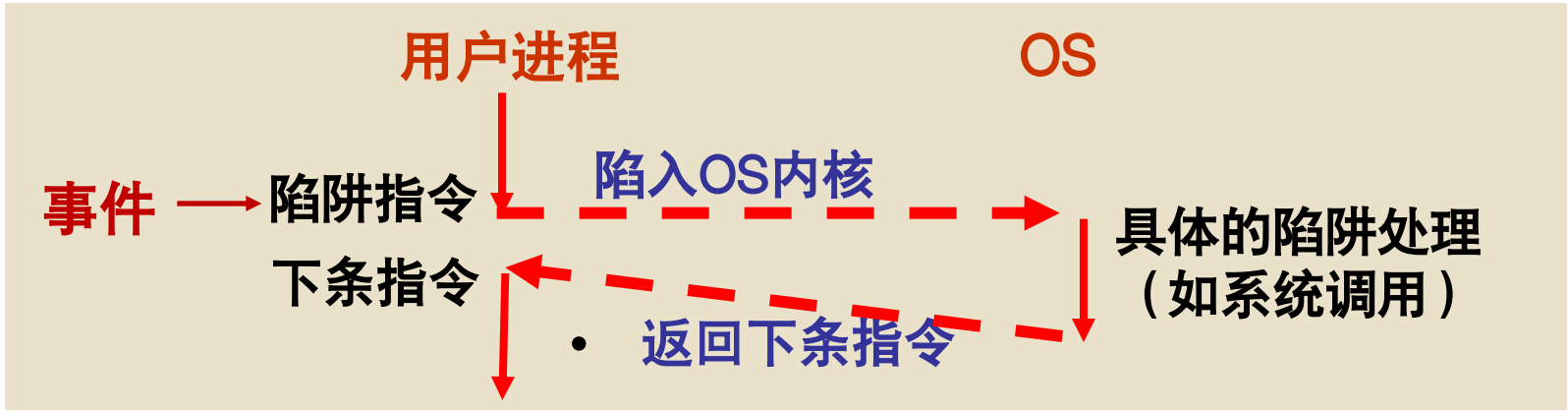
缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法操作、内存保护错等：终止当前进程。

- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！

陷阱（Trap）异常

- 陷阱也称自陷或陷入，执行陷阱指令（自陷指令/访管指令）时，CPU调出特定程序进行相应处理，处理结束后返回到陷阱指令下一条指令执行。



- 陷阱的作用之一是在用户和内核之间提供一个像过程一样的接口，这个接口称为系统调用，用户程序利用这个接口可方便地使用操作系统内核提供的一些服务。操作系统给每个服务编一个号，称为系统调用号。例如，Linux系统调用fork、read和execve的调用号分别是1、3和11。
- IA-32中的 int 和 sysenter 等指令、LoongArch中的 syscall和break等指令都属于陷阱指令（相当于“地雷”）。
- 陷阱指令异常称为编程异常（programmed exception），这些指令包括x86中的 INT n、int 3、into（溢出检查）、bound（地址越界检查）等

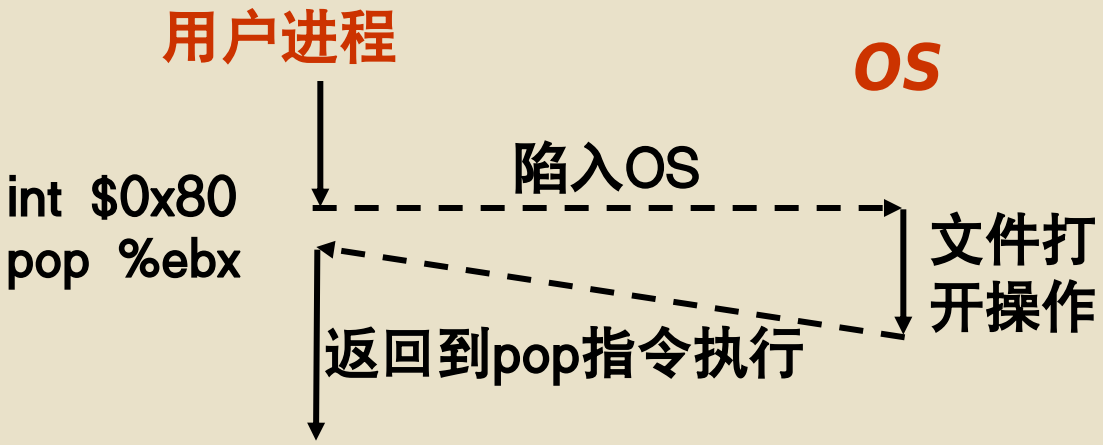
有条件“爆炸”

Trap举例: 打开文件

- 用户程序中调用函数 open(filename，options)
- open函数执行陷阱指令（如x86中系统调用指令“int”）

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                    int     $0x80
 804d084:       5b                       pop     %ebx
 . . .
```

这种“地雷”一定“爆炸”



通过执行“int $0x80”指令，调出OS完成一个具体的“服务”（称为系统调用）

陷阱（Trap）异常

问题：你用过单步跟踪、断点设置等调试功能吗？你知道这些功能是如何实现的吗？　通过“埋地雷”的方式实现

- 利用陷阱机制可实现程序调试功能，包括设置断点和单步跟踪
  - IA-32中，当CPU处于单步跟踪状态（TF=1且IF=1）时，每条指令都被设置成了陷阱指令，执行每条指令后，都会发生中断类型号为1的“调试”异常，从而转去执行“单步跟踪处理程序”。

    注意：当陷阱指令是转移指令时，不能返回到转移指令的下条指令执行，而是返回到转移目标指令执行。

    （在一定的条件下，每条指令都变成“地雷”）
  - LoongArch中，用于程序调试的“断点设置”陷阱指令为break，若调试程序在被调试程序某处设置了断点，则调试程序在该处设置一条break指令。当CPU执行到该指令时，就会暂停当前被调试程序的运行，并抛出断点异常（BRK），最终调出调试程序执行，调试工作结束后再回到被设定断点的被调试程序执行

    （break是一定爆炸的“地雷”）

SKIP

IA-32的标志寄存器

31-22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	O	D	I	T	S	Z	0	A	0	P	1	C

8086

80286/386

- 6个条件标志
  - OF、SF、ZF、CF各是什么标志（条件码）？
  - AF：辅助进位标志（BCD码运算时才有意义）
  - PF：奇偶标志

BACK

- 3个控制标志
  - DF（Direction Flag）：方向标志（自动变址方向是增还是减）
  - IF（Interrupt Flag）：中断允许标志 （仅对外部可屏蔽中断有用）
  - TF（Trap Flag）：陷阱标志（是否是单步跟踪状态）
- ……

异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。 “断点”为发生故障指令的地址

自陷(Trap)：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用（执行访管指令）等。是一种自愿中断。

“断点”为自陷指令下条指令地址

终止(Abort)：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点”是什么？ 随便！

思考1：自陷处理完成后回到哪条指令执行？ 回到下条指令

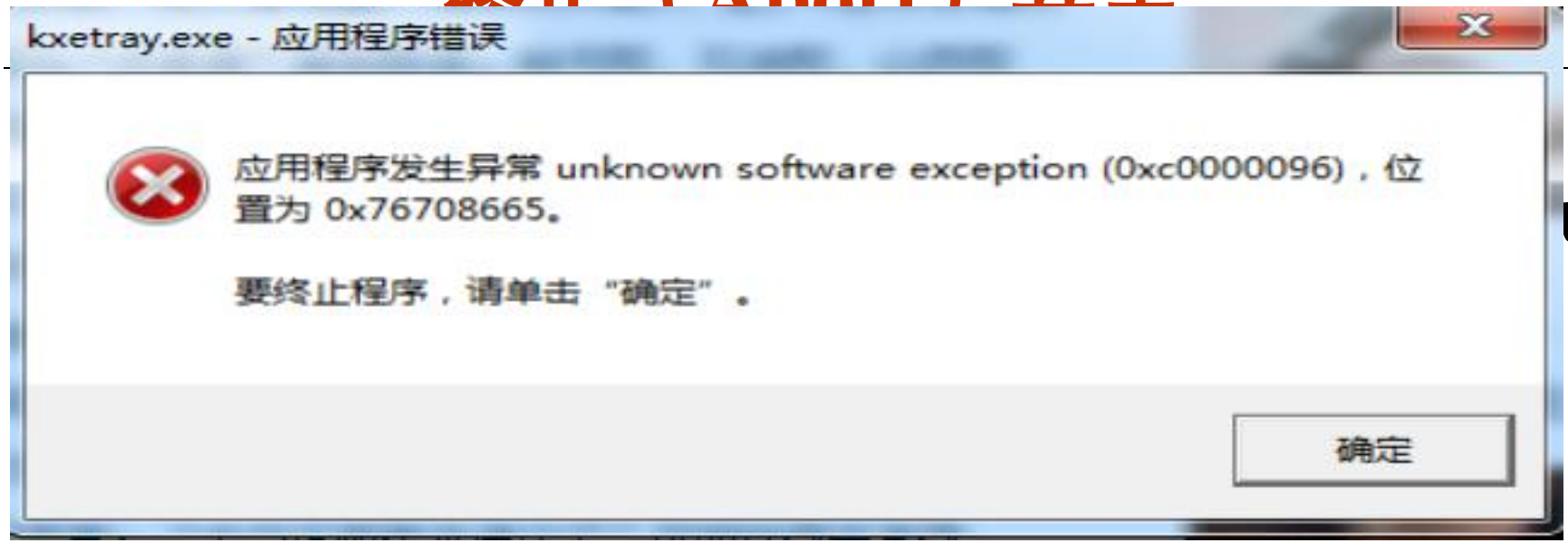
思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法操作、内存保护错等：终止当前进程。

- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！

终止（Abort）异常

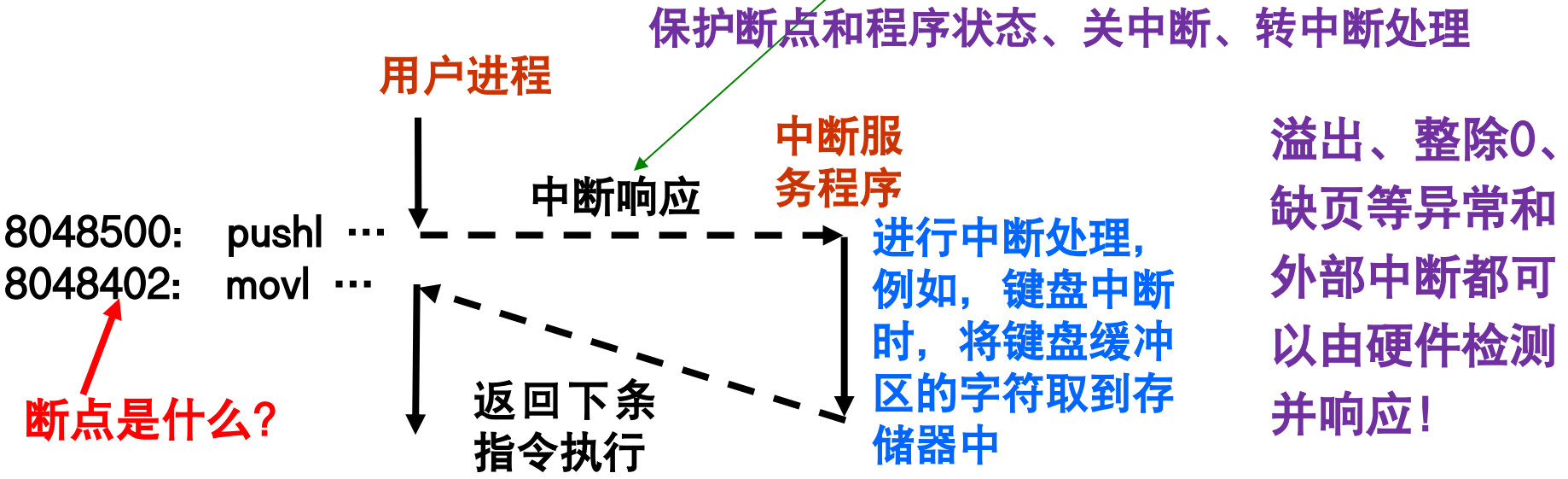


硬件检测到致命错误 → OS的终止处理程序 → 进行相应处理，并转入 abort 例程执行，以终止发生故障的进程或重启系统 → abort

0xc00000096为出错ID

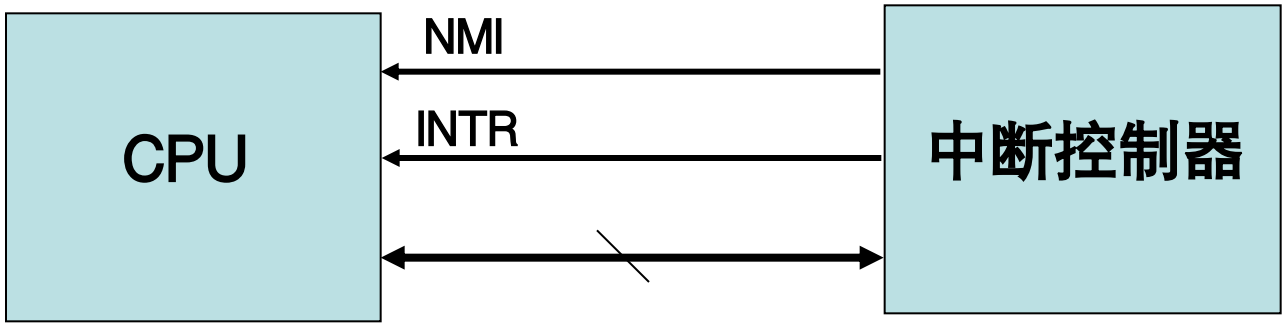
中断的概念

- 外设通过中断请求信号线向CPU提出“中断”请求，不由指令引起，故中断也称为异步异常。
- 事件：Ctrl-C、DMA传送结束、网络数据到达、打印缺纸、……
- 每执行完一条指令，CPU就查看中断请求引脚，若引脚的信号有效，则进行中断响应：将当前PC（断点）和当前机器状态保存到栈中，并“关中断”，然后，从数据总线读取中断类型号，根据中断类型号跳转到对应的中断服务程序执行。中断检测及响应过程由硬件完成。
- 中断服务程序执行具体的中断处理工作，中断处理完成后，再回到被打断程序的“断点”处继续执行。



中断的分类

- Intel将中断分成可屏蔽中断（maskable interrupt）和不可屏蔽中断（nonmaskable interrupt，NMI）。
  - 可屏蔽中断：通过 INTR 向CPU请求，可通过设置屏蔽字来屏蔽请求，若中断请求被屏蔽，则不会被送到CPU。
  - 不可屏蔽中断：非常紧急的硬件故障，如：电源掉电，硬件线路故障等。通过 NMI 向CPU请求。一旦产生，就被立即送CPU，以便快速处理。这种情况下，中断服务程序会尽快保存系统重要信息，然后在屏幕上显示相应的消息或直接重启系统。



异常/中断响应过程

检测到异常或中断时，CPU须进行以下基本处理:

① 关中断（“中断允许位” 清0）：使CPU处于“禁止中断”状态，以防止新中断破坏断点（PC）、程序状态（PSW）和现场（通用寄存器）。

② 保护断点和程序状态：将断点和程序状态保存到栈或特殊寄存器中

PC→栈 或 EPC（专门存放断点的寄存器）

PSWR →栈 或 EPSWR （专门保存程序状态的寄存器）

PSW（Program Status Word）：程序状态字

PSWR（PSW寄存器）：如IA-32中的的EFLAGS寄存器

③ 识别中断事件

有软件识别和硬件识别（向量中断）两种不同的方式。

IA-32中，响应异常时不关中断，只在响应中断时关中断

IA-32的标志寄存器

31-22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	O	D	I	T	S	Z	0	A	0	P	1	C

8086

80286/386

- 6个条件标志
  - OF、SF、ZF、CF各是什么标志（条件码）？
  - AF：辅助进位标志（BCD码运算时才有意义）
  - PF：奇偶标志
- 3个控制标志
  - DF（Direction Flag）：方向标志（自动变址方向是增还是减）
  - IF（Interrupt Flag）：中断允许标志 （仅对外部可屏蔽中断有用）
  - TF（Trap Flag）：陷阱标志（是否是单步跟踪状态）
- ……

异常/中断响应过程

有两种不同的识别方式：软件识别和硬件识别（向量中断）。

（1）软件识别（MIPS采用）

设置一个异常状态寄存器（MIPS中为Cause寄存器），用于记录异常原因。操作系统使用一个统一的异常处理程序，该程序按优先级顺序查询异常状态寄存器，识别出异常事件（异常/中断查询程序）。

（例如：MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序，用于查询异常的具体原因，然后转到内核中相应的异常处理程序段中进行具体的处理）

（2）硬件识别（向量中断）（IA-32采用）

用专门的硬件查询电路按优先级顺序识别中断，得到“中断类型号”，根据此号，到中断向量表中读取对应的中断服务程序的入口地址。

所有事件都被分配一个“中断类型号”，每个中断都有相应的“中断服务程序”，可根据中断类型号找到中断服务程序的入口地址。

中断类型号相当于中断向量表的索引，表中存放中断服务程序首地址

有些架构（如RISC-V、ARM等）采用软件识别和向量中断相结合的方式，先按大类分，用向量中断方式，每一大类再有一个统一的查询程序

LoongArch中，可采用软件识别方式，也可采用硬件识别方式

进程与异常控制流

- 分以下四个部分介绍
  - 第一讲：进程与进程的上下文切换
    - 程序和进程的概念
    - 进程的逻辑控制流
    - 进程与进程的上下文切换
  - 第二讲：异常和中断
    - 异常和中断的基本概念、异常和中断的响应和处理
  - 第三讲：LoongArch+Linux下的异常/中断机制
  - 第四讲：Linux中的进程控制与信号
    - 进程的创建、休眠和终止
    - 进程ID的获取和子进程的回收
    - Linux中的信号处理机制
    - 非本地跳转处理

LoongArch+Linux的异常和中断机制

异常也称为例外（LoongArch架构手册中称为例外），都对应exception

一、 LA中支持的异常/中断类型

1、LA定义的异常

可能是与TLB相关异常（其中包括TLB重填异常），可能是使用ECC等硬件校验方式的存储器发生校验错时触发的异常（这类属于机器错误异常），还可能是系统调用等引起的陷阱类异常和浮点指令触发的一些浮点指令异常等

异常分为TLB重填异常、机器错误异常和普通异常三类

普通异常包括Load操作页无效（PIL）、Store操作页无效（PIS）、地址非对齐（ALE）、边界检查错（BCE）、系统调用（SYS）、断点（BRK）、指令不存在（INE）、指令特权级错（IPE）等20多个异常类型

每个普通异常都有一级编号（Ecode），具有相同一级编号的不同异常还有二级编号（EsubCode）

2、LA定义的中断

有线中断和消息中断两种，线中断必须实现，消息中断可选择实现，它在线中断基础上扩展。线中断可来自处理器核内部，也可来自处理器核外部的其他模块的请求信号

13个线中断，分别为2个软中断（SWI0~SWI1）、8个硬中断（HWI0~HWI7）、1个性能监测计数溢出中断（PMI）、1个定时器中断（TI）和1个核间中断（IPI）

LoongArch+Linux的异常和中断机制

二、异常/中断相关的控制状态寄存器

异常/中断事件触发后，在硬件响应及软件处理的过程中，需将各种与异常/中断相关的信息记录在控制状态寄存器（CSR）中。

因为机器错误异常和TLB重填异常都可能在其他异常的处理过程中被触发，为了在触发机器错误异常和TLB重填异常时不破坏其他异常处理时的机器状态和现场信息，LA为这两种非普通异常各自定义了一组独立的控制状态寄存器。

TLB重填异常相关CSR包括记录断点处处理器核模式信息的TLBRPRMD、记录异常VA的TLBRBADV、记录异常处理程序入口地址的TLBRENTRY、记录返回地址的TLBRERA、异常处理程序保存寄存器TLBRSAVE、记录异常时VPPN和页大小(PS)的TLBREHI等。

机器错误异常相关CSR包括保存断点处模式信息的MERRCTL、配置对应处理程序入口地址的MERRENTRY、记录返回地址（断点信息）的MERRERA等。

为普通异常和线中断设置的CSR包括记录断点处模式信息的PRMD、记录所触发的异常类型编码和中断请求状态的ESTAT、记录配置信息的ECFG、记录入口地址的EENTRY和记录返回地址的ERA。

为消息中断所设置的CSR包括记录消息中断请求状态的MSGIS0～MSGIS3、记录被响应消息中断号的MSGIR、设置消息中断使能优先级门限的MSGIE

LoongArch+Linux的异常和中断机制

三、异常/中断的响应优先级

处理器核根据检测到的不同事件，将对应状态记录在不同的CSR状态位或字段中来标识发生了什么异常和中断。例如，检测到TLB缺失时将CSR.TLBRERA.IsTLBR置1表示发生了TLB重填异常；当检测到某消息中断请求被路由到指定处理器核时，该处理器核会根据消息中断号将核内的MSGIS0～MSGIS3中对应位置1标识消息中断类型。

同时发生多个异常和中断事件时，需根据优先级选择一个进行响应。

异常/中断响应优先级基本原则：1）中断的响应优先级高于异常；2）同时有消息中断和线中断的请求时，消息中断请求的优先级更高；3）对于消息中断和线中断，中断号越大，中断响应优先级越高。4）对于异常，取指令阶段检测到的异常优先级最高，译码阶段检测到的异常优先级次之，执行阶段检测到的异常优先级再次之。5）取指阶段异常优先级依次为监视点、地址错、与TLB相关异常、机器错误异常。6）译码阶段异常彼此互斥，无须考虑优先级。7）执行阶段只有Load/Store指令会同时触发多种异常，优先级依次为地址错ADE、地址对齐错ALE、边界约束错BCE、TLB相关异常等。

所有普通异常共享同一个编码字段CSR.ESTAT.Ecode和CSR.ESTAT.EsubCode标识异常，故该编码字段中记录的是被触发的优先级最高的普通异常编号

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

处理器负责对异常和中断的检测与响应，而操作系统则负责编制好异常/中断处理程序。处理器检测到异常或中断事件后，进行响应，进入内核态并调出相应程序进行处理

异常/中断响应过程：保存返回地址（断点）和上下文模式信息（程序状态）、记录具体的错误信息（如发生TLB缺失的虚拟地址、数据校验错信息等）并跳转到异常/中断处理程序的入口地址处。

开机后系统首先在直接地址翻译模式下工作，在进行一系列的硬件部件检测、引导程序加载并系统初始化后，进入映射地址翻译模式，因此，所描述的异常/中断相关内容都是指在映射地址翻译模式下执行指令时发生的情况

1. TLB重填异常的响应和处理

响应过程：1）保存上下文模式信息，并转入内核态；2）保存断点，并置TLB重填异常状态；3）记录触发异常的具体错误信息；4）转异常处理程序入口处执行。

处理过程：对主存中的页表进行遍历，若主存页表对应页表项中P=1（表示对应虚拟页已装入主存页框中）且未发生访问越权，则进行TLB重填操作；在异常处理程序的最后通过调用ERTN指令返回到所设置的断点处继续执行。

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

2. 机器错误异常的响应和处理

响应过程：（1）将CSR.CRMD的PLV、IE、DA、PG、DATF、DATM分别存到CSR.MERRCTL的PPLV、PIE、PDA、PPG、PDATF、PDATM中（保存模式信息），再将CSR.CRMD中的PLV清0（转内核态）、IE清0（关中断）、设置DA=1且PG=0并将DATF和DATM都设为00（将取指令操作和Load/Store操作的访存类型都设为强序非缓存直接地址翻译模式），若处理器的实现支持监视点功能，则将CSR.CRMD的WE存到CSR.MERRCTL的PWE中，再将CSR.CRMD的WE清0（全局使能位清0）；（2）将触发异常的指令地址存入CSR.MERRERA中（保存断点），同时将CSR.MERRCTL的IsMERR置1（发生机器错误异常时硬件须将该位置1）；（3）将触发异常的数据校验具体错误信息记录在CSR.MERRINFO1和CSR.MERRINFO2中（异常处理时需要用到的信息）；（4）跳转到CSR.MERRENTRY所配置的机器错误异常处理程序入口处执行。

处理过程：根据记录在CSR.MERRINFO1和CSR.MERRINFO2中的具体错误信息进行相应的处理，处理结束后通过调用ERTN指令返回到所设置的断点处继续执行。

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

2. 机器错误异常的响应和处理

响应过程：（1）将CSR.CRMD的PLV、IE、DA、PG、DATF、DATM分别存到CSR.MERRCTL的PPLV、PIE、PDA、PPG、PDATF、PDATM中（保存模式信息），再将CSR.CRMD中的PLV清0（转内核态）、IE清0（关中断）、设置DA=1且PG=0并将DATF和DATM都设为00（将取指令操作和Load/Store操作的访存类型都设为强序非缓存直接地址翻译模式），若处理器的实现支持监视点功能，则将CSR.CRMD的WE存到CSR.MERRCTL的PWE中，再将CSR.CRMD的WE清0（全局使能位清0）；（2）将触发异常的指令地址存入CSR.MERRERA中（保存断点），同时将CSR.MERRCTL的IsMERR置1（发生机器错误异常时硬件须将该位置1）；（3）将触发异常的数据校验具体错误信息记录在CSR.MERRINFO1和CSR.MERRINFO2中（异常处理时需要用到的信息）；（4）跳转到CSR.MERRENTRY所配置的机器错误异常处理程序入口处执行。

处理过程：根据记录在CSR.MERRINFO1和CSR.MERRINFO2中的具体错误信息进行相应的处理，处理结束后通过调用ERTN指令返回到所设置的断点处继续执行。

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

3. 普通异常的响应和处理

触发某普通异常时，由于不同异常所对应事件不同，在响应过程中可能有一些细微差异。

响应过程：所有普通异常的响应过程中共有的操作包括（1）将CSR.CRMD的PLV、IE分别存到CSR.PRMD的PPLV、PIE中（保存模式信息），再将CSR.CRMD中的PLV清0 （转内核态） 、IE清0 （关中断） ，若处理器的实现支持监视点功能，则将CSR.CRMD的WE存到CSR.PRMD的PWE中，再将CSR.CRMD的WE清0 （全局使能位清0）；（2）将触发异常的指令地址（PC）存入CSR.ERA中（异常处理时需要用到的信息）；（3）跳转到CSR.EENTRY所配置的异常处理程序入口地址处执行。

处理过程：在某些普通异常处理过程中，若通过执行特权指令将CSR.CRMD.IE置1来开启中断使能，则需保存CSR. PRMD中的PPLV、PIE等信息，并在异常处理返回前将它们恢复到CSR.PRMD中。处理结束后通过调用ERTN指令返回到所设置的断点处继续执行。

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

4. 线中断的检测、响应和处理

检测过程：所有线中断都是可屏蔽中断，13个线中断的局部使能位（即中断屏蔽位）配置在CSR.ECFG.LIE[12:0]中。处理器根据对各线中断源的中断请求信号的采样结果，将中断请求状态信息记录在CSR.ESTAT.IS位中，然后将CSR.ESTAT. IS[12:0]与CSR.ECFG. LIE[12:0]进行按位与操作（进行中断屏蔽运算），当13位结果不全为0且CSR.CRMD.IE=1（开中断）时，说明在全局中断允许的情况下有未被屏蔽的中断请求，从而进入线中断响应阶段。

响应过程：线中断响应过程与普通异常响应过程相同。

处理过程：在线中断处理过程中，若通过执行特权指令将CSR.CRMD.IE置1来开启中断使能，则需保存CSR. PRMD中的PPLV、PIE等信息，并在中断处理返回前将它们恢复到CSR.PRMD中。处理结束后通过调用ERTN指令返回到所设置的断点处继续执行。

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

5. 消息中断的检测、响应和处理

检测过程：当CSR.MSGIS0～CSR.MSGIS3中的256个消息中断请求状态位为1的消息中断号中，存在高于或等于CSR.MSGIE所设置的消息中断使能优先级门限值的消息中断请求，且CSR.CRMD.IE=1（开中断）时，处理器选择优先级最高的消息中断请求进行响应，将该消息中断号记录在CSR.MSGIR.IntNum中，同时将CSR.MSGIR.Null清0，并将消息中断请求状态位CSR.ESTAT.MsgInt置1。

响应过程：选中响应的消息中断请求后，需要清除其请求状态信息。硬件自动根据CSR.MSGIR.IntNum中的消息中断号，将CSR.MSGIS0～CSR.MSGIS3寄存器中对应状态位清0。该状态位清0后，若CSR.MSGIS0～CSR.MSGIS3中所有位为0，或者虽存在状态为1的位，但其优先级低于消息中断使能优先级门限值（由CSR.MSGIE.PT设置），则说明再没可响应的消息中断请求，下一个时钟周期就将CSR.ESTAT.MsgInt清0，同时将CSR.MSGIR.Null置1。随后响应过程与普通异常和线中断的响应过程相同。

处理过程：在线中断处理过程中，若通过执行特权指令将CSR.CRMD.IE置1来开启中断使能，则需保存CSR. PRMD中的PPLV、PIE等信息，并在中断处理返回前将它们恢复到CSR.PRMD中。处理结束后通过调用ERTN指令返回到所设置的断点处继续执行。

LoongArch+Linux的异常和中断机制

四、异常/中断的响应过程和处理

6. 异常/中断处理结束后的返回过程

异常/中断处理结束后，通过执行ERTN指令返回的过程如下：

1）恢复程序状态（模式信息）。对于TLB重填异常，将CSR.TLBRPRMD中的PPLV、PIE恢复到CSR.CRMD的PLV、IE字段中；对于机器错误异常，将CSR.MERRCTL中的PPLV、PIE、PDA、PPG、PDATF、PDATM恢复到CSR.CRMD的PLV、IE、DA、PG、DATF、DATM字段中；对于普通异常/中断，将CSR.PRMD中的PPLV、PIE恢复到CSR.CRMD的PLV、IE字段中。若支持监测点功能，则分别将CSR.TLBRPRMD、CSR.MERRCTL和CSR.PRMD中的PWE恢复到CSR.CRMD.WE字段中。

2）恢复映射地址翻译模式。对于TLB重填异常，将CSR.CRMD的DA清0、PG置1。

3）清异常/中断位。对于TLB重填异常和机器错误异常，分别将CSR.TLBRERA中的IsTLBR位和CSR.MERRCTL中的IsMERR位清0。

4）返回到断点处执行。对于TLB重填异常、机器错误异常和普通异常/中断，分别将记录在CSR.TLBRERA、CSR.MERRERA和CSR.ERA中的返回地址（断点）送PC。

此时，在下个时钟周期，处理器将回到发生异常/中断的进程断点处继续执行。

LoongArch+Linux的异常和中断机制

五、异常/中断处理程序的入口地址

TLB重填和机器错误这两种非普通异常对应的异常处理程序入口地址分别被配置在CSR.TLBRENTRY和CSR.MERRENTRY中，因而，只要在异常响应的最后，将配置在相应CSR中的入口地址送入PC即可。

普通异常和中断对应的异常/中断处理程序入口地址由入口地址虚页号与入口地址页内偏移通过计算得到。其中，入口地址虚页号都相同，配置在CSR.EENTRY的VPN中，而入口地址页内偏移则可配置为相同，也可配置为各不相同。

1）当CSR.ECFG.VS=0时，普通异常/线中断处理程序入口地址页内偏移相同，因此，系统中有一个统一的普通异常/线中断查询程序，属于软件识别异常/中断方式

2）当CSR.ECFG.VS≠0时，普通异常/线中断处理程序入口地址页内偏移计算公式：

$2^{(CSR.ECFG.VS+2)} \times Ecode$，其中，Ecode值取决于异常类型编号和中断号。普通异常Ecode为CSR.ESTAT.Ecode中的6位1级编号（最大为63），线中断Ecode为中断号加64，即SWI0的Ecode为64，SWI1的Ecode为65，以此类推，线中断中最大的Ecode为12+64=76。

3）消息中断的Ecode为78，其入口地址页内偏移为：$2^{(CSR.ECFG.VS+2)} \times 78$。

4）对于普通异常和中断，对应异常/中断处理程序入口地址的计算公式：

$\{CSR.EENTRY.VPN, 12'b00\ 0000\ 0000\} \mid 2^{(CSR.ECFG.VS+2)} \times Ecode$

其中的运算符“|”表示按位或。

Linux中对异常的处理

Linux中所有异常处理程序的结构是一致的，都可划分成以下三个部分：

（1）**准备阶段：** 在内核栈保存通用寄存器内容（称为现场信息），这部分大多用汇编语言程序实现。

（2）**处理阶段：** 采用C函数进行具体处理。函数名由do_前缀和处理程序名组成，如 do_overflow 为溢出处理函数。

大部分函数的处理方式：保存硬件出错码（如果有的话）和异常类型号，然 后，向当前进程发送一个信号。

（3）**恢复阶段：** 恢复保存在内核栈中的各个寄存器的内容，切换到用户态并返回到当前进程的断点处（可恢复）或信号处理程序（不可恢复）执行。

当前进程接受到信号后，若有对应信号处理程序，则转信号处理程序执行；若

Linux中对异常的处理

- 异常处理程序发送相应的信号给发生异常的当前进程（不能恢复时），或者进行故障恢复（能恢复时），然后返回到断点处执行。

  例如，遇到非法操作码，在对应的异常处理程序中，向当前进程发送一个SIGILL信号，以通知当前进程中止运行。

- 采用向发生异常的进程发送信号的机制实现异常处理，可尽快完成在内核态的异常处理过程，因为异常处理过程越长，嵌套执行异常的可能性越大，而异常嵌套执行会付出较大的代价。

- 并不是所有异常处理都只是发送一个信号到发生异常的进程。

  例如，对于访存异常，需要判断是否为访问越级、越权或越界等，若发生了这些无法恢复的故障（x86中为14号中断，#PF），则页故障处理程序发送SIGSEGV信号给发生页故障异常的进程；若只是缺页，则页故障处理程序负责把所缺失页面从磁盘装入主存，然后返回到发生缺页故障的指令继续执行。

注意：异常处理在内核态，信号处理在用户态　　SKIP

Linu

为何除法错显示却是“浮点异常”的原因

类型号	助记符	含义描述	处理程序名	信号名
0	#DE	除法出错	divide_error( )	SIGFPE
1	#DB	单步跟踪	debug( )	SIGTRAP
2		NMI 中断	nmi( )	无
3	#BP	断点	int3( )	SIGTRAP
4	#OF	溢出	overflow( )	SIGSEGV
5	#BR	边界检测（BOUND）	bounds( )	SIGSEGV
6	#UD	无效操作码	invalid( )	SIGILL
7	#NM	协处理器不存在	device_not_available( )	无
8	#DF	双重故障	doublefault( )	无
9	#MF	协处理器段越界	coprocessor_segment_overrun( )	SIGFPE
10	#TS	无效 TSS	invalid_tss( )	SIGSEGV
11	#NP	段不存在	segment_not_present( )	SIGBUS
12	#SS	栈段错	stack_segment( )	SIGBUS
13	#GP	一般性保护错（GPF）	general_protecton( )	SIGSEGV
14	#PF	页故障	page_fault( )	SIGSEGV
15		保留	无	无
16	#MF	浮点错误	coprocessor_error( )	SIGFPE
17	#AC	对齐检测	alignment_check( )	SIGSEGV
18	#MC	机器检测异常	machine_check( )	无
19	#XM	SIMD 浮点异常	simd_coprocessor_error( )	SIGFPE

x86+Linux中异常对应的信号名和处理程序名

回顾：用“系统思维”分析问题

代码段一：

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

运行结果为-2147483648

objdump反汇编代码，得知除以 -1 被优化成取负指令neg，故未发生除法溢出

代码段二：

```
int a = 0x80000000;
int b = -1;
int c = a / b;
printf("%d\n", c);
```

a/b用除法指令IDIV实现，但它不生成OF标志，那么如何判断溢出异常的呢？

实际上是“除法错”异常#DE（类型0）

Linux中，对#DE类型发SIGFPE信号

对应的信号处理程序显示信息：“Floating point exception”

运行结果为“Floating point exception”，显然CPU检测到了溢出异常

为什么两者结果不同！

异常举例—除法溢出和访存违例

假设在IA-32+Linux系统中一个C语言源程序如下：

```
#include "stdio.h"
unsigned short b[2500];
unsigned short k;
void main() {
        printf("k=");
        scanf("%hd",&k);
    b[1000]=1023;
    b[2500]=2049%k;
    b[10000]=20000;
    printf("ok");
}
```

执行结果如下（无相关编译选项）：

k=0

Floating point exception (core dumped)

k=2

Segmentation fault (core dumped)

(单步跟踪情况下，k=2时不会出现异常，输出ok)

1）为何k=0时，出现浮点错？

2）除0异常由硬件检测还是由软件检测？

3）有的指令集架构由硬件检测除法错，有的由软件检测，编译器分别该如何处理？

4）为何此处无须相应的编译选项？

5）为何这里出现“段故障”？

6）如何（何时）进入内核处理的？

LoongArch+Linux的异常和中断机制

LoongArch中，整数除指令发生**溢出**（在最小负数除以-1时发生）或**除数为0**时，不会像在Intel x86架构中那样由**硬件自动触发异常**。因此，要对除数为0进行处理，需要在**系统软件层面**提供一套处理机制。

在LoongArch+Linux平台中，当执行到指令“break code”时将无条件触发断点（BRK）异常（Ecode=0xC），对应的异常处理函数名为do_bp()。指令中的code为传递给异常处理函数的参数。例如:

0x5用于单步调试，执行到“break 0x5”指令时，当前进程会收到一个SIGTRAP信号，对应信号处理程序在屏幕上输出提示信息“Trace/breakpointtrap”，同时会停在当前指令位置，从而进入调试状态。在GDB调试工具中，软件断点功能通过“break 0x5”指令实现。

0x7对应整除0异常（在break.h中定义为BRK_DIVZERO）。执行“break 0x7”指令触发断点异常后，将陷入内核态并最终调用断点异常处理函数do_bp()。

异常举例—除法溢出和除0

在LA32+Linux系统中所运行程序P中主要包含以下C语言代码段:

```
int a = 0x80000000;
int b,c;
scanf("%d",&b);
c = a/b;
printf("%d\n", c);
```

没有相关编译选项时，执行结果如下:

```
b=-1
-2147483648
b=0
Floating point exception(core dumped)
```

编译器将上述赋值语句“c=a/b;”转换为机器级代码:

```
    ld.w    $r13, $r22, -28(0xffe4)  #R[r13]←b;
    ld.w    $r14, $r22, -20(0xfec)   #R[r14]←a;
    div.w   $r12, $r14, $r13  #R[r12]←a/b;
    bne     $r13, $r0, .L0    #若R[r13]!=0则转.L0
    break   0x7               #触发断点异常
.L0
    st.w    $r12, $r22, -24(0xfe8)   #c←a/b;
```

1）未检测除法溢出

2） LA32中规定：div.w指令在除数为0时正常执行，结果可为任意值

3）软件检测整除0，并在检测到除数为0时，触发断点(break)异常

4） break指令陷入内核后，

异常举例—除法溢出和除0

do_bp()函数中的switch-case语句

```
switch (bcode) {
case BRK_BUG:
        bug_handler(regs);
        break;
case BRK_DIVZERO:
        die_if_kernel("Break instruction in kernel code" , regs);
        force_sig_fault(SIGFPE, FPE_INTDIV, (void __user *) regs->csr_era);
        break;
case BRK_OVERFLOW:
        die_if_kernel("Break instruction in kernel code" , regs);
        force_sig_fault(SIGFPE, FPE_INTOVF, (void __user *) regs->csr_era);
        break;
default:
        die_if_kernel("Break instruction in kernel code", regs);
        force_sig_fault(SIGTRAP, TRAP_BRKPT, (void __user *) regs->csr_era);
        break;
```

没有相关编译选项时，执行结果如下：

b=-1（出现BRK_OVERFLOW情况，但没检测）

-2147483648

b=0（出现BRK_DIVZERO情况）

Floating point exception(core dumped)

异常举例—除法溢出和除0

-fsanitize=undefined和-fsanitize-undefined-trap-on-error

`-fsanitize=undefined` 是一个编译器标志，用于在编译时启用未定义行为检测（UBD）。当你的代码中存在不符合C或C++语言规范的行为时，编译器可以用这个标志来生成额外的代码，以在运行时检测和报告这些问题。

`-fsanitize-undefined-trap-on-error` 是一个编译器标志，用于配合 `-fsanitize=undefined` 标志，指示在检测到未定义行为时，程序应当产生一个硬件异常（比如在x86上产生SIGILL）而不是使用未定义的行为来继续执行。

这两个标志通常一起使用，以便在检测到潜在的错误时中断程序的执行，便于开发者进行调试。

例如，如果你正在编译一个C语言程序，你可以在gcc或clang编译器中这样使用这两个标志：

Bash Copy code

```
1  gcc -fsanitize=undefined -fsanitize-undefined-trap-on-error your_program.c
```

或者使用clang：

Bash Copy code

```
1  clang -fsanitize=undefined -fsanitize-undefined-trap-on-error your_prograr
```

当你的程序中存在未定义行为时，编译器会在生成的代码中插入额外的检测代码，运行时如果发现问题，程序会产生异常并中止执行，方便你查看和调试。

异常举例—除法溢出和除0

在LA32+Linux系统中所运行程序P中主要包含以下C语言代码段:

```
int a = 0x80000000;
int b,c;
scanf("%d",&b);
c = a/b;
printf("%d\n", c);
```

有-fsanitize=undefined和-fsanitize-undefined-trap-on-error编译选项时，执行结果如下:

```
b=-1
Trace/breakpoint trap(core dumped)
b=0
Trace/breakpoint trap(core dumped)
```

编译器将上述赋值语句“c=a/b;”转换为机器级代码段:

```
      ......
   srli.w   $r13,$r13,0x18
   beq      $r13,$r0, .L1
   break    0x0      #发生除法溢出
 .L1
   div.w    $r13,$r16,$r12
   bne      $r12,$r0, .L0
   break    0x7      #发生除数为0
.L0
   st.w     $r13,$r22,-24(0xfe8)
      ......
```

为何同样是break 7，对应的信号（SIGFPE、SIGTRAP）却不同？ 或者应该为 break 0x5 ??

1）软件检测除法溢出，并在检测到除法溢出时，触发断点(break)异常

2）软件检测整除0，并在检测到除数为0时，触发断点(break)异常

3） 将除法溢出看成未定义行为，而整除0

在没有上述编译选项时，也会检测

异常举例—除法溢出和除0

`-fsanitize=undefined` 和 `-fsanitize-undefined-trap-on-error` 是用于检测未定义行为的编译器 instrumentation 选项。当你的代码中存在未定义行为时，这些选项可以帮助你捕捉到这些行为。

但是，这些选项并不直接专门用于检测“整数除法溢出”。整数溢出通常是由 `-fsanitize=signed-integer-overflow` 或 `-fsanitize=unsigned-integer-overflow` 选项检测的。

如果你想检测整数除法时产生的溢出，你需要同时使用整数溢出的检测选项和除法溢出的检测方法。

例如，你可以使用 `-fsanitize=undefined -fsanitize=signed-integer-overflow` 来检测未定义行为和整数溢出。

下面是一个简单的示例代码，它演示了一个整数除法可能引起的问题，以及如何使用编译器选项来检测这种问题：

异常举例—除法溢出和除0

在LA32+Linux系统中所运行程序P中主要包含以下C语言代码段:

```
int a = 0x80000000;
int b,c;
scanf("%d",&b);
c = a/b;
printf("%d\n", c);
```

有-fsanitize=undefined和-fsanitize=signed-integer-overflow编译选项时，编译不通过，显示:

loongarch32r-linux-gnusf-gcc: error: libsanitizer.spec: No such file or directory

LA32架构中，整数除法溢出作为未定义行为，由-fsanitize=undefined和-fsanitize-undefined-trap-on-error编译选项进行处理，因此，无须支持-fsanitize=undefined和-fsanitize=signed-integer-overflow编译选项。整除0是编译器必须进行检测的异常，无须指定专门的编译选项

异常举例—除法溢出和访存违例

假设在RISC-V+Linux系统中一个C语言源程序如下：

```
#include "stdio.h"
unsigned short b[2500];
unsigned short k;
void main() {
        printf("k=");
        scanf("%hd",&k);
    b[1000]=1023;
    b[2500]=2049%k;
    printf("%d",b[2500]);
    b[10000]=20000;
}
```

不加-fsanitize=undefined和-fsanitize-undefined-trap-on-error编译选项时，执行结果如下：

```
k=0
2049
Segmentation fault (core dumped)

k=2
1
Segmentation fault (core dumped)
```

1）不加相关编译选项时，除法不报错

2）执行“b[10000]=20000;”对应的指令时，硬件检测到“段故障”异常

异常举例—除法溢出和访存违例

假设在RISC-V+Linux系统中程序如下：

```
#include "stdio.h"
unsigned short b[2500];
unsigned short k;
void main() {
        printf("k=");
        scanf("%hd",&k);
    b[1000]=1023;
    b[2500]=2049%k;
    printf("%d",b[2500]);
    b[10000]=20000;
}
```

加-fsanitize=undefined和-fsanitize-undefined-trap-on-error编译选项时：

若k=0，则执行ebreak指令，进入OS按 SIGTRAP 信号处理，OS使进程终止，并显示 “Trace/breakpoint trap”

```
632:  a2a78793   add  a5,a5,-1494 # 2058 <b> 636:
3ff00713    li    a4,1023
63a:  7ce79823   sh   a4,2000(a5)
63e:  00002797   auipc  a5,0x2
642:  9ca78793   add    a5,a5,-1590 # 2008 <k> 646:
0007d783   lhu    a5,0(a5)
64a:  2781         sext.w  a5,a5
64c:  873e         mv     a4,a5
64e:  00173713   seqz   a4,a4
652:  0ff77713    zext.b  a4,a4
656:  c311         beqz   a4,65a <main+0x32> 658:
9002          ebreak
65a:  6705         lui    a4,0x1
65c:  8017071b    addw   a4,a4,-2047 # 801
<__FRAME_END__+0x141>
660:  02f747bb    divw   a5,a4,a5
```

LoongArch+Linux的系统调用

- 系统调用（陷阱）是特殊异常事件，是OS为用户程序提供服务的手段。
- Linux提供了几百种系统调用，主要分为以下几类:
  - 进程控制、文件操作、文件系统操作、系统控制、内存管理、网络管理、用户管理、进程通信等
- 系统调用号是系统调用跳转表索引值，跳转表给出系统调用服务例程首址

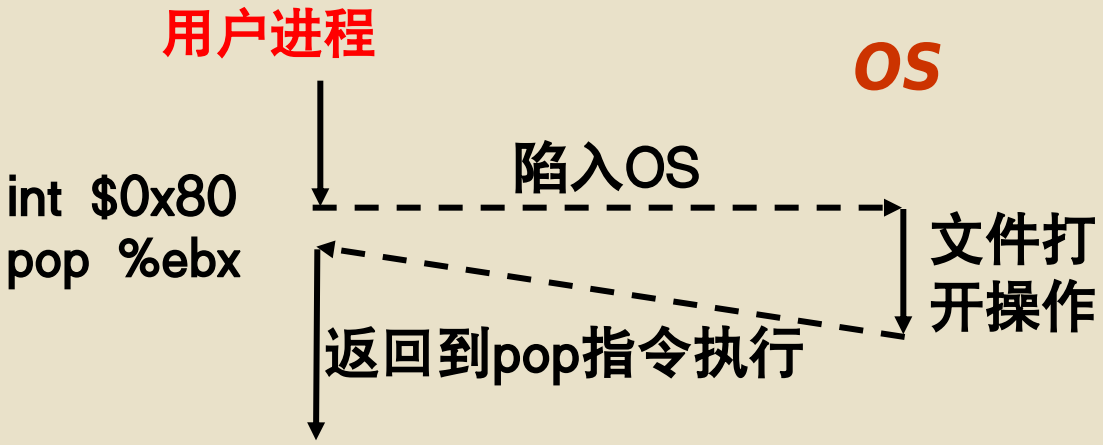
调用号	名称	类别	含义	调用号	名称	类别	含义
93	exit	进程控制	终止进程	49	chdir	文件系统	改变当前工作目录
220	clone	进程控制	克隆一个子进程	169	gettimeofday	系统控制	取得系统时间
63	read	文件操作	读文件	62	lseek	文件系统	移动文件指针
64	write	文件操作	写文件	172	getpid	进程控制	获取进程号
56	openat	文件操作	打开文件	129	kill	进程通信	向进程或进程组发信号
57	close	文件操作	关闭文件	214	brk	内存管理	修改虚拟空间中的堆指针 brk
260	wait4	进程控制	等待子进程终止	222	mmap	内存管理	建立虚拟页面到文件片段的映射
34	mkdir	文件操作	创建目录	80	fstat	文件系统	获取文件状态信息
221	execve	进程控制	运行可执行文件	179	sysinfo	系统控制	获取系统信息

Trap举例: Opening File

- 用户程序中调用函数 open(filename，options)
- open函数执行陷阱指令（即系统调用指令“int”）

这种“地雷”一定“爆炸”

```
0804d070 <__libc_open>:
  . . .
 804d082:      cd 80          int    $0x80
 804d084:      5b             pop    %ebx
  . . .
```



通过执行“int $0x80”指令，调出OS完成一个具体的“服务”（称为系统调用）

LoongArch+Linux的系统调用

- 通常，系统调用被封装成用户程序能直接调用的函数，如exit()、read()和open()，这些是标准C库中系统调用对应的封装函数，称为系统级函数
- Linux中系统调用所用参数通过寄存器传递，在LoongArch+Linux平台中，系统调用号存放在寄存器a7中，传递的参数从左到右依次存放在寄存器a0～a6中，最多可通过寄存器传递7个参数。若参数个数超出7，则将参数块所在存储区首地址放在寄存器中传递。系统调用的返回值存放在寄存器a0和a1中
- 封装函数对应的机器级代码有一个统一的结构:
  - 总是若干条传送指令后跟一条陷阱指令。传送指令用来传递系统调用的参数，陷阱指令syscall用来陷入内核进行处理。
- 例如，若用户程序调用系统调用write(1, “hello，world!\n”,14)，将字符串“hello，world!\n”中14个字符显示在标准输出设备文件stdout上，则其封装函数对应机器级代码（用汇编指令表示）如下:

```
li.w    $a7，64          #write的系统调用号为64，送寄存器a7
    li.w          $a0，1          #标准输出设备stdout的文件描述符为1，送寄存器a0
    la.local  $a1，.L0    #字符串"hello，world!\n"的首地址为.L0，送寄存器a1
    li.w          $a2，$14        #字符串的长度为14，送寄存器a2
    syscall    0x0                #系统调用，从用户态陷入内核态
```

LoongArch+Linux系统调用的返回值

- 系统调用的返回值在a0和a1中，若是正数或0表示成功；遇到错误时，返回值为负数（通常是-1），并设置全局整数变量errno表示出错码
- 将errno作为入口参数调用strerror()函数，可得到对应的错误描述文本串
- 为避免每次系统调用都出现错误检查及处理代码，可使用相应的封装函数，如目标函数fork()的封装函数可以是以下的Fork()函数，这样，在需要调用目标函数时，用调用其封装函数来代替，从而简化程序代码
- 与“库打桩机制”中说明的一样，目标函数和封装函数的原型应该完全一致

目标函数fork()对应的错误检查及处理封装函数Fork()如下：

```
pid_t Fork(void) {
    pid_t pid;
    if ((pid=fork()<0) {
        fprintf(stderr, "fork error: %s\n", strerror(errno));
        exit(0);
    }
    return pid;
}
```

补充说明：fork()对应头文件为unistd.h，stderr为标准错误输出文件，对应头文件和fprintf()对应头文件都是stdio.h，strerror()对应头文件是string.h，全局整数变量errno对应头文件为errno.h，exit()对应在头文件stdlib.h，Fork()的程序中必须包含这些头文件。

进程与异常控制流

- 分以下四个部分介绍
  - 第一讲：进程与进程的上下文切换
    - 程序和进程的概念
    - 进程的逻辑控制流
    - 进程与进程的上下文切换
  - 第二讲：异常和中断
    - 异常和中断的基本概念、异常和中断的响应和处理
  - 第三讲：LoongArch+Linux下的异常/中断机制
  - 第四讲：Linux中的进程控制与信号
    - 进程的创建、休眠和终止
    - 进程ID的获取和子进程的回收
    - Linux中的信号处理机制
    - 非本地跳转处理

进程的创建、休眠和终止

Linux提供了进程控制系统调用，如fork、waitpid、execve、getpid等

在C程序中可调用这类系统级函数进行进程创建和回收等

从程序员角度看，一个进程总是处于以下三种情况之一

运行状态。进程正在CPU上运行或等待被OS调度以换上CPU运行

挂起状态。进程的执行被暂停且不可能被调度执行。当进程收到SIGSTOP、SIGTSTP、SIGTTIN或SIGTTOU信号时，进入挂起（suspended）状态，直到收到一个SIGCONT信号，此时进程再次进入运行状态。这里的信号是Linux系统中提供的在进程之间或进程和操作系统内核之间进行消息传送的一种机制

终止状态。通常以下三种情况导致进程终止：收到一个其默认行为为终止进程的信号、从主程序返回、调用exit()函数

① 进程终止函数：exit()

② 子进程创建函数：fork()

③ 进程休眠函数：sleep()和pause()

进程的创建、休眠和终止

① 进程终止函数exit()：头文件stdlib.h中定义，无返回值，函数原型为

void exit(int status);

② 子进程创建函数fork()：在父进程中创建一个子进程，函数原型为

pid_t fork(void);

在头文件unistd.h中定义，返回值类型pid_t在sys/types.h中定义为int型

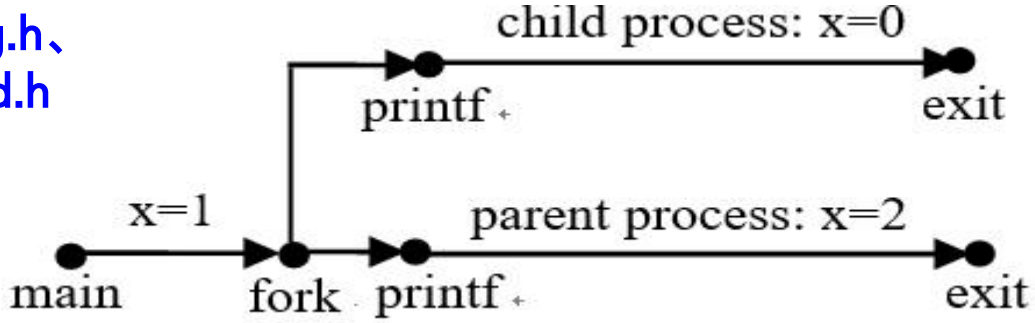
系统中通常用唯一的正整数标识一个进程，称为进程ID，简写为PID

有与父进程完全相同但独立的虚拟地址空间，并继承父进程的打开文件描述符表

调用一次，返回两次，一次在父进程中返回子进程的PID，一次在子进程中返回0

以下程序应包含头文件stdio.h、string.h、errno.h、sys/types.h、stdlib.h、unistd.h

```
int x=1;
int main() {
   pid_t pid;
   if ((pid=Fork())==0) {
     printf("child process: x=%d\n",--x);
     exit(0);
   }
   printf("parent process: x=%d\n",++x);
   exit(0);
}
```



fork()执行后，OS并发执行父进程和子进程，两者先被调度执行的可能性相同

执行结果为

child process: x=0
parent process: x=2

或

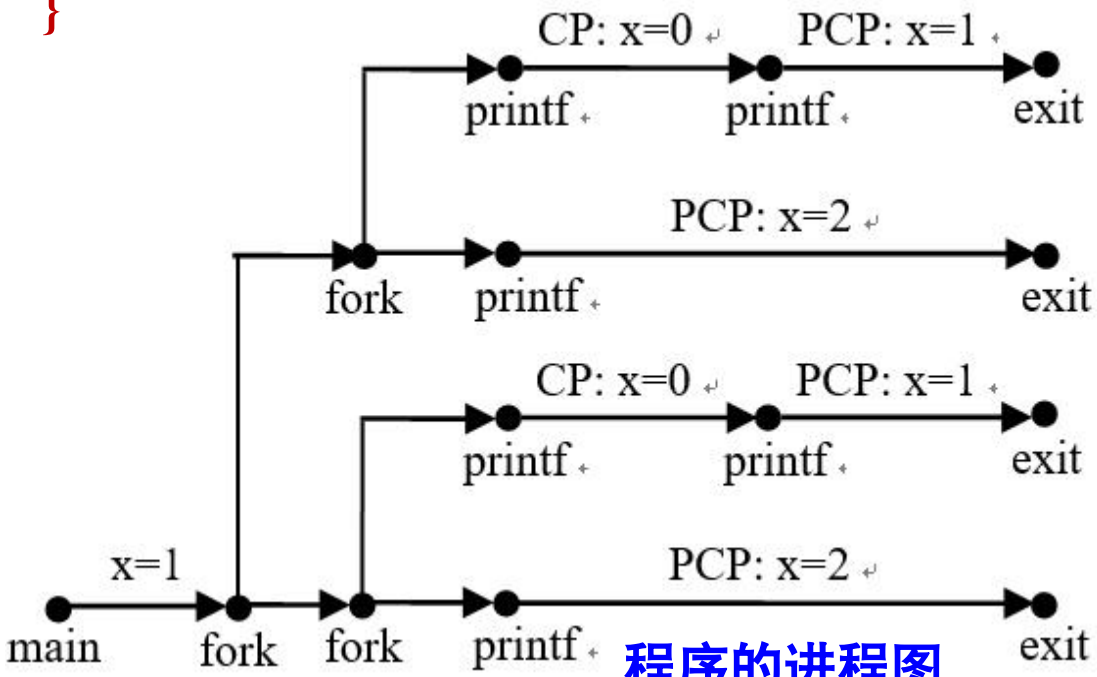
parent process: x=2
child process: x=0

进程的创建、休眠和终止

画出以下程序的进程图，并给出其中4种可能的执行结果。

```
int main(){
    int x=1;
    Fork();
    if (Fork()==0)
        printf("CP: x=%d\n",--x);
    printf("PCP: x=%d\n",++x);
    exit(0);
}
```

执行完两次fork()共有4个进程并发运行，OS调度进程执行的顺序组合很多，导致可能得到许多不同结果，只要保证输出序列中存在两个“CP:x=0→PCP:x=1”的有序输出对即可。例如，以下4种就是其可能的结果。



程序的进程图

① PCP：x=2
CP：x=0
PCP：x=1
PCP：x=2
CP：x=0
PCP：x=1

② CP：x=0
PCP：x=2
PCP：x=1
PCP：x=2
CP：x=0
PCP：x=1

③ CP：x=0
PCP：x=1
PCP：x=2
PCP：x=2
CP：x=0
PCP：x=1

④ PCP：x=2
PCP：x=2
CP：x=0
CP：x=0
PCP：x=1
PCP：x=1

进程的创建、休眠和终止

③ 进程休眠函数：sleep()和pause()

可通过调用sleep()函数让进程休眠指定的一段时间，进程在休眠期间被挂起。

函数原型为

unsigned int sleep(unsigned int s);

参数s是指定的休眠秒数，若给定休眠时间已到，则返回0；若sleep()函数被一个信号中断而提前返回，则返回剩下的秒数。

可使用pause()函数让进程休眠，直到进程收到一个信号为止。

函数原型为

int pause(void);

这两个函数对应头文件为unistd.h

```
int main(){
    int remtime;
    sleep(30)
    remtime=alarm(10); //10s后发送SIGALRM信号
    printf("remaining time: %d\n",remtime);
    pause();      //休眠直到收到SIGALRM信号
}
```

先休眠30秒，再输出“remaining time: 0”，等待10秒后，接收到SIGALRM信号，然后进程终止（未设置SIGALARM对应的信号处理程序时，默认行为是终止进程）

进程ID的获取和子进程的回收

- 僵尸进程

处于终止状态但还未被父进程回收的进程称为僵尸进程

其残留资源还存于内核中，直到被父进程回收时，内核才把其资源收回并从系统中清除，同时把它的退出状态传递给父进程，此时它在系统中完全消亡

- 孤儿进程

若父进程先于子进程消亡，则子进程成为孤儿进程

init进程是所有孤儿进程的父进程，它在系统启动时由内核创建，不会终止，其PID为1，是所有进程的祖先，负责对孤儿进程的回收。

进程ID的获取和子进程的回收

- getpid()和getppid()分别用于获取调用进程、调用进程父进程的PID。

函数原型为

```
pid_t  getpid(void);
pid_t  getppid(void);
```

这两个函数对应头文件为unistd.h，定义pid_t类型的头文件为sys/types.h

- 子进程回收函数：waitpid()

父进程可通过该函数等待子进程终止将其回收，并记录其终止状态或错误码。

waitpid()函数的原型如下：

```
pid_t  waitpid(pit_t  pid，int  *wstatusp，int  options);
```

pid指定等待集，wstatusp中存放回收进程的退出状态，options用于设定按默认行为处理还是对处理行为进行某种修改。options=0为默认情况，其处理行为是：调用waitpid()的调用进程被挂起，直到等待集中的一个进程终止时，waitpid()返回，若等待集中的一个进程在刚调用waitpid()时已经终止，则waitpid()立即返回，这两种情况下，waitpid()的返回值为已终止子进程的PID。若函数发生错误，则返回-1，并将错误码设置在全局变量errno中

进程ID的获取和子进程的回收

- 子进程回收函数：waitpid()

pid_t waitpid(pit_t pid, int *wstatusp, int options);

pid指定等待集，wstatusp中存放回收进程的退出状态，options用于设定按默认行为处理还是对处理行为进行某种修改。

1）指定等待集

若pid>0，则等待集中仅有一个进程ID为pid的子进程

若pid=−1，则等待集中包含调用进程的所有子进程

若pid = 0，则等待集中包含与调用进程组ID相等的进程组中的所有子进程

若pid<−1，则等待集中包含进程组ID为pid绝对值的进程组中的所有子进程

2）修改默认行为

可将options设定为特定常量WNOHANG、WUNTRACED 等的各种组合，以修改默认行为。如WNOHANG | WUNTRACED表示不等待而立即返回

3）判定回收进程的退出状态

若wstatusp为非空，则会在其指向的wstatus中存放对应子进程的退出状态

4）设置错误码

若函数发生错误，则错误码设置在全局变量errno中

可用wait(&wsatus)代替waitpid(−1, $wsatus, 0)

进程ID的获取和子进程的回收

3）判定回收进程的退出状态

若wstatusp为非空，则会在其指向的wstatus中存放对应子进程的退出状态。

头文件sys/wait.h中有若干用于状态值含义解释的宏定义。

常用宏定义如下：

① WIFEXITED(wstatus)：若子进程通过调用exit()或执行return语句正常终止，则返回结果为真。

② WEXITSTATUS(wstatus)：仅当WIFEXITED(wstatus)为真时才有定义，返回结果为正常终止的子进程的退出状态，例如，若子进程以调用函数exit(1)的方式终止，则返回结果为1。

③ WIFSTOPED(wstatus)：若waitpid()是因为子进程被挂起而返回，则返回结果为真。

④ WSTOPSIG(wstatus)：仅当WIFSTOPED(wstatus)为真时才有定义，返回结果为引起子进程被挂起的信号编号，例如，若子进程因接受到SIGSTOP信号而被挂起，则返回结果为对应编号19。

⑤ WIFCONTINUED(wstatus)：若子进程收到SIGCONT信号而被重新启动，则返回结果为真。

进程的创建、休眠和终止

画出以下程序的进程图，并给出执行结果。

```
int main(){
   pid_t pid;
   int cnt=0;
   int wstatus=20;
   if ((pid=Fork())>0)
      if (wait(&wstatus)>0)
        if (WIFEXITED(wstatus)!=0) {
           printf("cnt=%d, wstatus=%d\n",cnt,WEXITSTATUS(wstatus));
           printf("PP PID=%d\n",getpid());
           exit(0);
        }
   else {
      while(1){
         printf("CP PID=%d\n",getpid());
         sleep(1);
         cnt++;
         if(cnt==2) exit(2);
      }
   }
   return 0;
}
```

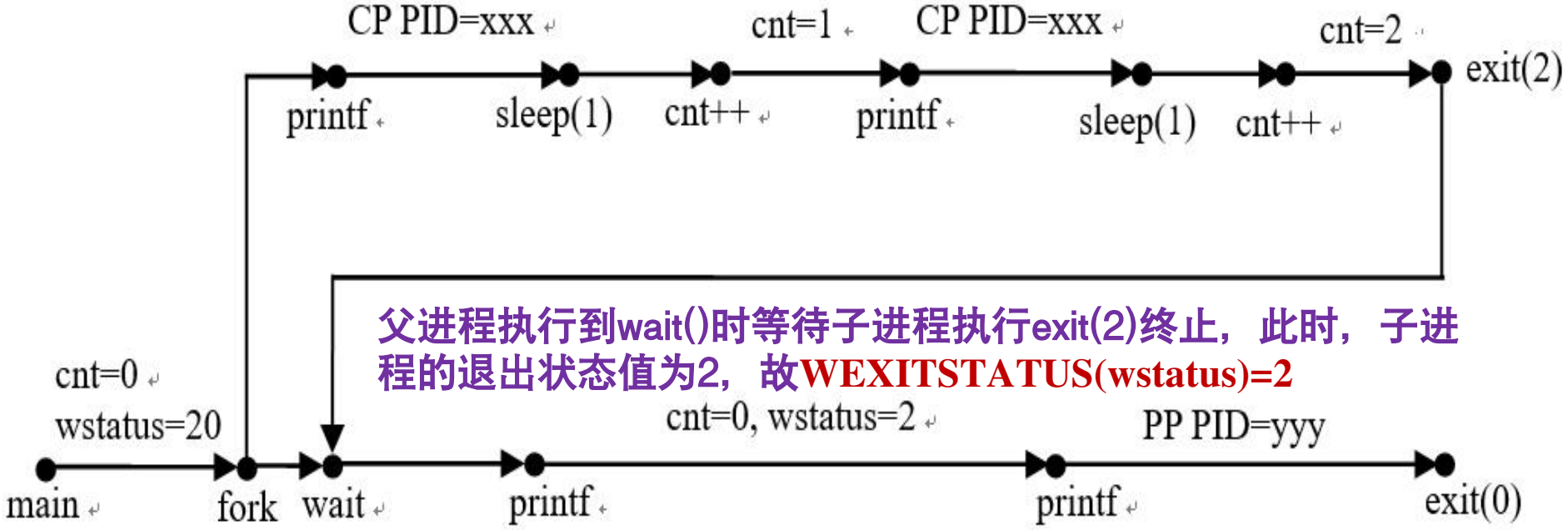
程序中应包含的头文件有sys/types.h、sys/wait.h、stdio.h、stdlib.h、unistd.h、string.h和errno.h。

进程的创建、休眠和终止

```
int main(){
    pid_t pid;
    int cnt=0;
    int wstatus=20;
    if ((pid=Fork())>0)
        if (wait(&wstatus)>0)
          if (WIFEXITED(wstatus)!=0) {
            printf("cnt=%d, wstatus=%d\n",cnt,WEXITSTATUS(wstatus));
            printf("PP PID=%d\n",getpid());
            exit(0);
```

CP PID=xxx
CP PID=xxx
cnt=0, wstatus=2
PP PID=yyy

其中xxx为子进程的PID，yyy为父进程的PID，在子进程中每次输出“CP PID=xxx”后休眠1s。



程序的进程图

程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来启动加载器。
- execve()函数的功能是在当前进程上下文中加载并运行一个新程序。execve()函数的用法如下：

  int execve(char *filename, char *argv[], *envp[]);

  filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回-1，并将控制权交给调用程序；若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。
- 主函数main()的原型形式如下：

  int main(int argc, char **argv, char **envp);　　或者：

  int main(int argc, char *argv[], char *envp[]);

  argc指定参数个数，参数列表中第一个总是命令名（可执行文件名）

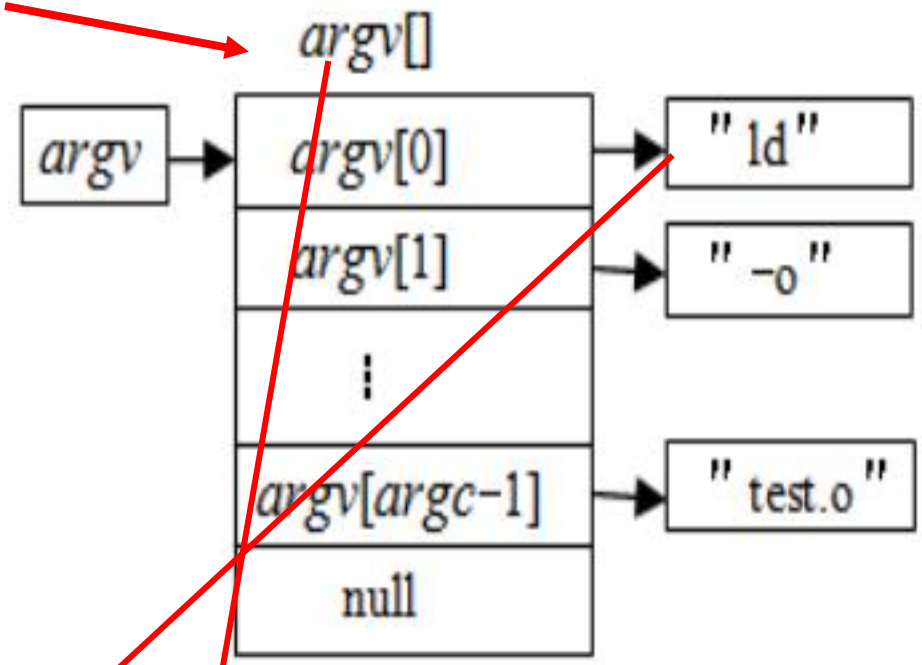
  例如：命令行为“ld -o test main.o test.o” 时，argc=5

程序的加载和运行

若在shell命令行提示符下输入以下命令行

Unix>ld –o test main.o test.o

ld是可执行文件名（即命令名），随后是命令的若干参数，argv是一个以null结尾的指针数组，argc=5



在shell命令行提示符后键入命令并按“enter”键后，便构造argv和envp，然后调用execve()函数来启动加载器，最终转main()函数执行

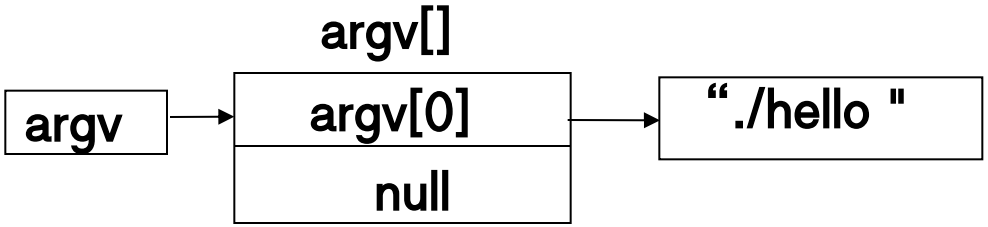
```
int execve(char *filename, char *argv[], *envp[]);
int main(int argc, char *argv[], char *envp[]);
```

程序的加载和运行

问题：hello程序的加载和运行过程是怎样的?

Step1：在shell命令行提示符后输入命令：$./hello[enter]

Step2：shell命令行解释器构造argv和envp



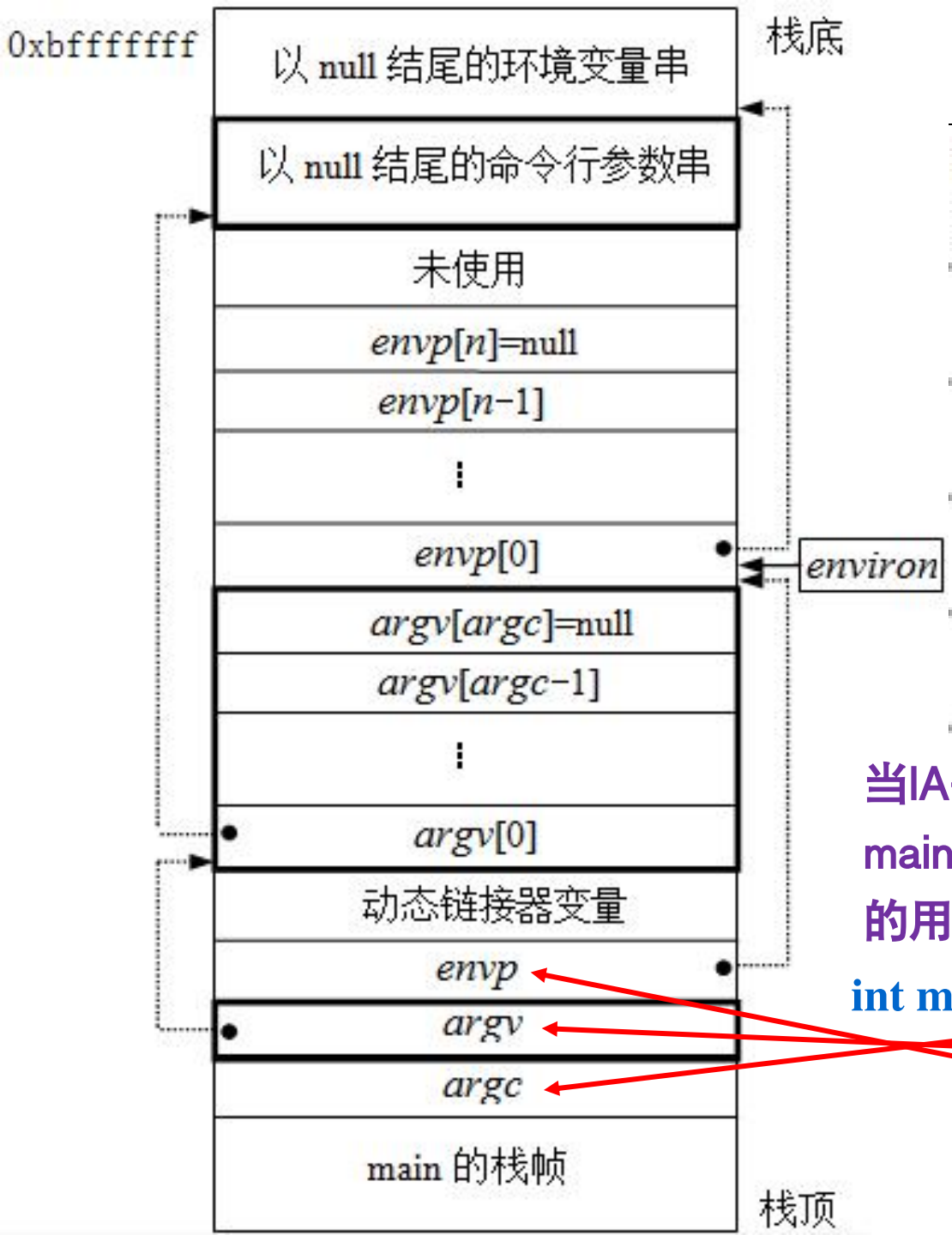
Step3：调用fork()函数，创建一个子进程，与父进程shell完全相同（只读/共享），包括只读代码段、可读写数据段、堆以及用户栈等。

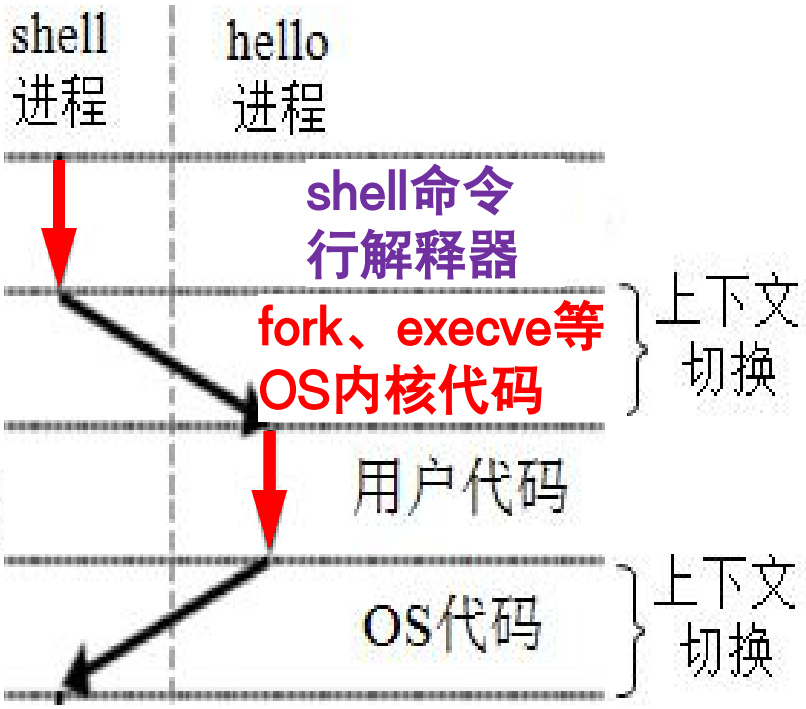
Step4：调用execve()函数,在当前进程（新创建的子进程）的上下文中加载并运行hello程序。将hello中的.text节、.data节、.bss节等内容加载到当前进程的虚拟地址空间（仅修改当前进程上下文中关于存储映像的一些数据结构，不从磁盘拷贝代码和数据等内容）

Step5：调用hello程序的main()函数，hello程序开始在一个进程的上下文中运行。

```
int main(int argc, char *argv[], char *envp[]);
```

程序加载和运行





当IA-32/Linux系统开始执行main()函数时，在虚拟地址空间的用户栈中的结构如左图所示

```
int main(int     argc,
         char *argv[],
         char *envp[]);
```

可执行文件的加载

加载执行可执行文件a.out的大致过程如下：

① shell命令行解释器输出命令行提示符（如：linux>），并开始接收用户输入的命令行。

② 当用户在命令行提示符后输入“./a.out[enter]”后，shell开始解析命令行，获得各命令行参数并构造传递给函数execve()的参数列表argv，将命令行字符串数量送argc。

③ 调用函数fork()。fork()函数创建一个子进程并使新创建的子进程获得与父进程完全相同的存储器映射，即完全复制父进程的mm_struct、vm_area_struct数据结构和页表，并将两者中每一个私有页的访问权限都设置成只读，将两者vm_area_struct中描述的私有区域中的页设置为私有的写时拷贝页。这样，如果其中一个进程写入其中某一页，内核将使用写时拷贝机制在主存中分配一个新页框，并将页面内容拷贝到新页框中。

④ 子进程以第②步命令行解析得到的参数数量argc、参数列表argv以及全局变量environ作为参数，调用函数execve()，在当前进程的上下文中加载并运行a.out程序。

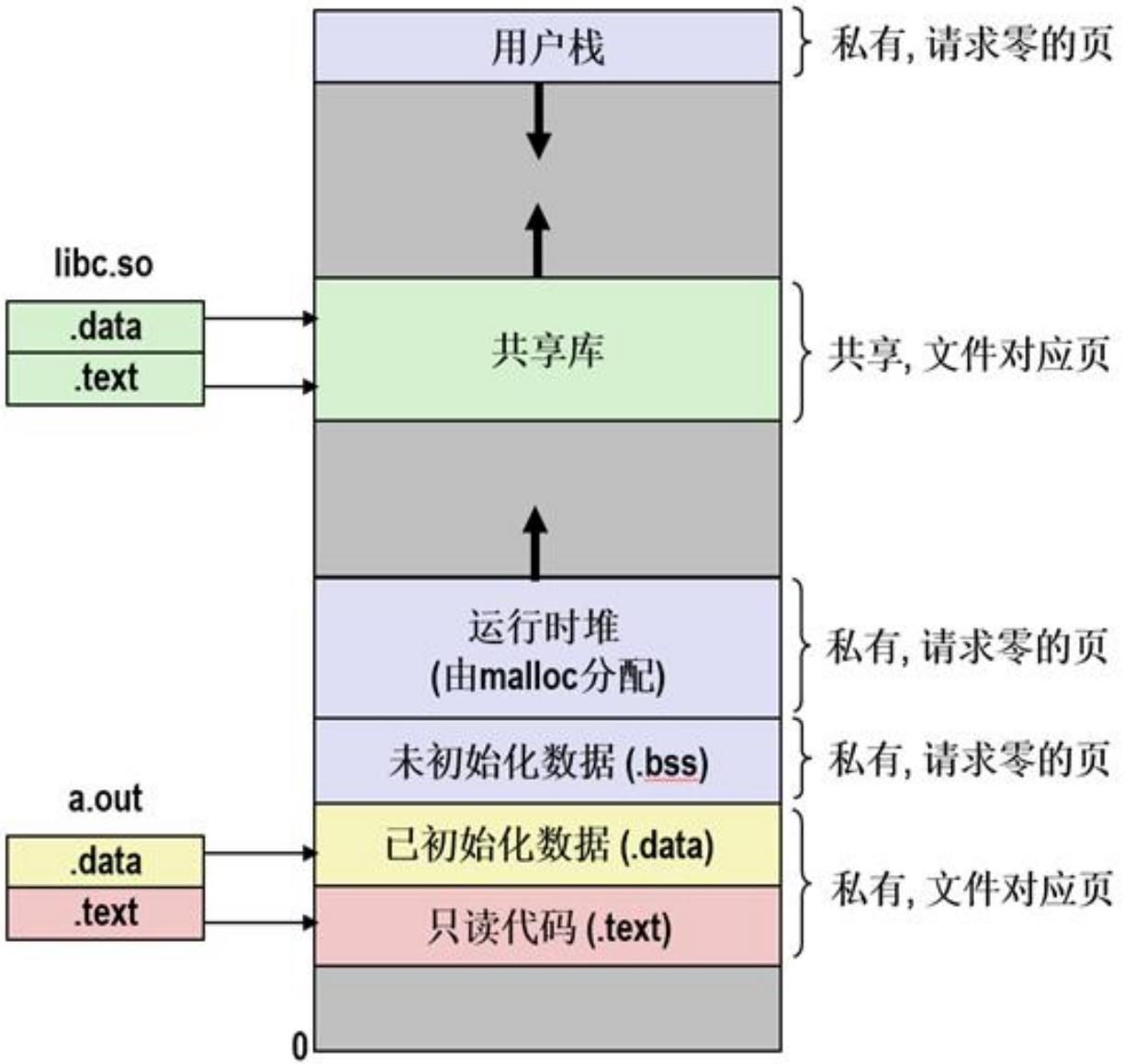
Linux中虚拟地址空间中的区域

函数execve()启动加载器执行加载任务并启动程序运行

具体步骤包括:

1）回收已有的VM用户空间中的结构vm_area_struct及其页表。

2）根据a.out的程序头表创建新进程的VM用户空间中各私有区域（包括只读代码、已初始化数据、未初始化数据、栈和堆），生成相应的vm_area_struct链表，并填写相应的页表项。



Linux中的信号处理机制

- 异常控制流的形成原因:
  - 内部异常（缺页、越权、越级、整除0、溢出等）
  - 外部中断（Ctrl-C、打印缺纸、DMA结束等）
  - 进程的上下文切换（发生在操作系统层）
  - 一个进程直接发送信号给另一个进程（发生在应用软件层）
- 允许用户进程和操作系统内核通过发送信号中断其他进程的执行
- 每种信号代表某类系统事件，通过发送信号给目标进程，以告知系统中发生了一个某种类型的事件
- Linux通过信号机制，让内核在处理异常时发送信号给用户进程，使得用户进程可以通过注册信号处理程序的方式选择如何进行异常处理
- 每种信号都有一个信号名和编号，例如
  - 浮点异常信号名为SIGFPE，编号为8
  - 非法指令对应信号名为SIGILL，编号为4

Linux中的信号处理机制

- 除内部异常外，一些系统事件也可由内核发送信号给用户进程处理。例如
  - 在前台进程运行时若按下Ctrl+C键，则内核会发送SIGINT信号（编号2）给前台进程组中每个进程，该信号的默认处理行为是终止进程
  - 当一个子进程终止或被挂起时，内核会发送SIGCHID信号（编号17）给父进程，该信号的默认行为是什么也不做
- 除内核向用户进程发送信号外，进程之间也可以发送信号。例如
  - 一个进程可通过调用kill()函数请求内核向另一个目标进程发送指定的信号，如SIGKILL信号（编号为9）会强制终止目标进程
  - 一个进程也可以给自己发送信号

前台进程：指控制标准输入输出（终端）的进程。shell进程一开始工作在前台，用户输入命令后，shell进程启动命令执行后被隐藏到后台，而执行命令对应的进程被提到前台，开始接受用户输入。前台进程运行结束后退出，shell进程被自动提到前台，等待用户输入命令。

后台进程：也叫守护进程(Daemon)。耗时长且不使用终端交互的进程可以设置在后台运行，在shell命令行最后加&表示将命令对应的进程设置在后台执行，在后台执行的进程不必等到前一个进程运行完才能运行。

Linux中的信号处理机制

- 当目标进程被内核控制以某种方式对信号的发送进行处理时，称为信号被接收
- 目标进程接收到信号后，会转移到对应的信号处理程序执行，形成异常控制流
- 调用信号处理程序的过程称为信号捕获，执行信号处理程序的过程称为信号处理
- 信号处理结束后，将返回到被中断的程序继续执行
- 有些信号可以忽略，如内核发送给父进程的SIGCHID信号、内核发送给挂起进程的SIGCONT信号等
- 有些信号则既不能被忽略，也不能被捕获，如SIGKILL信号的行为只能是强制终止进程，而不能被忽略（什么都不做），也不能被捕获去调用信号处理程序
- 若一个发出的信号未被接收，则该信号称为待处理信号（pending signal），任何时刻，一个进程中每种标准信号最多只会有一个待处理信号，随后发送过来的同类标准信号直接被丢弃（若是实时信号，则不会被丢弃）
- 一个进程可以有选择地阻塞接收某种信号，当某种信号因被阻塞而未被接收时，则发送过来的信号变为待处理信号，直到取消其阻塞接收。一个待处理信号最多只能被接收一次。内核为每个进程维护一个待处理信号集和一个被阻塞信号集

  阻塞信号集指哪些信号被屏蔽（每个信号对应一个屏蔽位）

信号的发送

- Linux提供了多种信号发送机制
  - Ø 调用专门的系统级函数发送信号
  - Ø 使用/bin/kill程序发送信号
  - Ø 在键盘上按下特定的按键发送信号
- 可指定发送到的目标进程属于哪个进程组
- 每个进程仅属于一个进程组，用正整数标识进程组ID
- 可用getpgrp()返回当前进程所属的进程组ID，函数原型如下

  pid_t getpgrp(void);
- 可用setpgid()设置自己或其他进程的进程组ID，函数原型如下

  int setpgid(pid_t pid, pid_t pgid);

  将pid进程所属的进程组ID改为pgid。

  若pid为0，则表示对调用进程本身进行设置

  若pgid为0，则表示设置的进程组ID为pid

  例如，若调用进程ID为20232，则其中的函数调用语句“setpgid(0, 0);”的功能是，将进程20232所属的进程组ID改为20232。

信号的发送

- 使用kill()函数发送信号

  int kill(pid_t pid, int sig);

  pid为发送信号的目标进程ID，sig为发送的信号

  若pid=0，则目标进程为调用进程所在进程组中每个进程，包括调用进程本身

  若pid<0，则目标进程为ID为pid绝对值（|pid|）的进程组中的每个进程

  sig可用在头文件signal.h中定义的信号名常量来设置，例如，函数调用语句“kill(pid, SIGKILL);”中的发送信号为SIGKILL

```
int main(){
    int x=1
    pit_t pid;
    if ((pid=Fork())==0) {
        printf("CPPID=%d,x=%d\n",getpid(),++x);
        pause();
        printf("CPPID=%d,x=%d\n",getppid(),--x);
        exit(0);
    }
    kill(pid,SIGKILL);
    exit(0);
}
```

子进程在休眠中接受到父进程通过kill()函数向子进程发送的SIGKILL信号，被迫终止

信号的发送

- 使用alarm()函数发送信号

unsigned int alarm(unsigned int seconds);

用于设置闹钟（定时器）并传送SIGALRM信号

在指定的seconds秒后将SIGALRM信号发送给调用进程，参数为0则取消闹钟

若未设置SIGALARM对应的信号处理程序，则默认的处理行为是终止进程

一个进程只能有一个闹钟，若在设定时间内再次调用该函数设置新闹钟，则之前设置的秒数将被新时间取代，所剩秒数作为返回值。若之前未设定过闹钟，则返回0

```
int main(){
int  remtime;
      alarm(50);
      sleep(30);
      remtime=alarm(10);
      printf("remaining  time:  %d\n",remtime);
      pause();
      return  0;
}
```

先休眠30秒，再输出“remaining time: 20”，等待10秒后接收到SIGALRM信号，然后进程终止（未设置SIGALARM对应的信号处理程序时，默认行为是终止进程）

应包含的头文件有stdio.h和unistd.h

信号的发送

- 用/bin/kill命令可向某进程发送任何指定信号，例如

  linux> /bin/kill -9 20232

  将向PID为20232的进程发送编号为9的信号SIGKILL

  进程号为负数表示将信号发送到进程组ID为其绝对值的组中每个进程，例如

  linux> /bin/kill -9 -20232

  将SIGKILL信号发送到进程组20232中的每个进程

- 从键盘发送信号

  从键盘输入Ctrl+C会导致内核向前台进程组中每个进程发送一个SIGINT信号

  默认情况下，将终止前台作业，回到shell进程

  从键盘输入Ctrl+Z则向前台进程组中每个进程发送一个SIGTSTP信号

  默认情况下，将挂起前台作业直到接收到下一个信号SIGCONT

一个shell命令行中所有命令对应的进程构成一个作业，shell为每个作业创建一个独立的前台进程组，所有前台进程组中的进程组成前台作业，如“linux> ls | sort ”（ | 为管道操作符，前一进程的输出是后一进程的输入）命令行对应前台作业中有两个进程：ls和sort

信号的捕获与处理

- OS内核在完成异常/中断处理或进行一次上下文切换而要从内核态切换到用户态的进程p执行时，会检查进程p的未被阻塞的待处理信号集
- 若为空集，则直接返回到进程p的逻辑控制流中原被中断的下一指令处（断点）执行
- 若为非空集，则内核将强制p对集合中编号最小的信号进行接收，从而触发进程p针对接收的信号采取某种处理行为。一旦处理完就返回到p的逻辑控制流中断点处执行
- 系统中每个信号都有一个预定义的默认处理行为，如终止进程、挂起进程以等待被SIGCONT信号重启、忽略信号等。
- 程序员可自行定义信号处理函数，并通过signal函数将其与对应信号绑定，从而修改信号的默认行为。但信号SIGSTOP和SIGKILL的默认行为不能修改
- signal()函数原型如下：

```
typedef  void  (*sighandler_t)(int);
sighandler_t  signal(int  signum,  sighandler_t  handler);
```

若handler为SIG_IGN，则忽略类型为signum的信号

若handler=SIG_DFL，则类型为signum的信号恢复默认行为

否则handler就是用户自定义函数（信号处理程序）的地址

若出错，则返回SIG_ERR。常数SIG_IGN、SIG_ERR等在signal.h中定义

信号的捕获与处理

- signal()函数原型如下：

  typedef void (*sighandler_t)(int);

  sighandler_t signal(int signum, sighandler_t handler);

  handler是用户自定义函数（信号处理程序）的地址

说明以下程序的执行过程。若signal()函数调用改为“signal(SIGINT, SIG_IGN)”或“signal(SIGINT, SIG_DFL)”，则执行结果分别是什么？

```
typedef void (*sighandler_t)(int);
void sigint_handler(int sig) {
    printf("caught SIGINT!\n");
}

int main(){
    if (signal(SIGINT, sigint_handler)==SIG_ERR)
        printf("signal error\n");
    pause();
    return 0;
}
```

通过signal()将sigint_handler()信号处理函数和SIGINT信号进行了绑定，改变了其默认行为。程序执行后一直休眠等待，直到用户输入Ctrl+C时发送SIGINT信号，信号被捕获后进行信号处理，输出“caught SIGINT!”，然后回到主函数执行return，结束程序执行

若改为“signal(SIGINT, SIG_IGN)”，则程序执行过程中按下Ctrl+C键后，没有任何反应。

若改为“signal(SIGINT, SIG_DFL)”，则程序执行过程中按下Ctrl+C键后，程序马上执行结束。

信号的非本地跳转处理

- C语言提供了一种非本地跳转函数，可实现用户级异常控制流
- 非本地跳转函数的执行可将控制直接从一个函数转移到另一个正在执行的函数，而不需经正常的调用-返回（call-return）序列
- 非本地跳转通过setjmp()和longjmp()函数实现，函数原型如下

  int setjmp(jmp_buf env);

  头文件setjmp.h给出了这些函数的原型声明及jmp_buf数据类型的定义

  setjmp()在由env指定的缓冲区中保存**当前调用环境**以供longjmp()使用，并返回0

  setjmp()对一个跳转目标处的程序上下文信息进行初始化并记录在env中

  void longjmp(jmp_buf env, int retval);

  通过调用longjmp()将env中的程序上下文信息恢复作为当前调用环境，从而触发从最近一次的setjmp()返回，此时，setjmp()的返回值为非0的retval，若retval=0，则返回值为1。

  可根据setjmp()的返回值来判定是调用了setjmp()（**此时返回值为0**）还是调用了longjmp()（**此时返回值为非0**）而返回的

  通常在检测到一个程序错误时，调用longjmp()函数

信号的非本地跳转处理

```
jmp_buf env;
int error1=0;
int error2=1;
void err_det1(void) {
      if (error1) longjmp(env,1);
      err_det2()
}
void err_det2(void) {
      if (error2) longjmp(env,2);
}
int main(){
      switch (setjmp(env)) {
      case 0:
            err_det1();
            break;
      case 1:
            printf("error1 detected\n");
            break;
      case 2:
            printf("error2 detected\n");
            break;
      default:
            printf("other error detected\n");
      }
      return 0;
}
```

首先调用setjmp()函数保存当前调用环境，并返回0，因此调用函数err_det1()，在该函数中未检测到错误，因此继续调用函数err_det2()，在该函数中检测到错误后，调用longjmp(env,2)，使得执行setjmp(env)后返回2，从而转到case 2分支执行，输出“error2 detected”后，程序执行结束。

包含的头文件有stdio.h、setjmp.h。

信号的非本地跳转处理

- 非本地跳转的另一种应用场景是在信号处理程序中使用
- 通过sigsetjmp()和siglongjmp()实现接收信号的进程和相应信号处理程序之间的跳转
- 这两个函数的原型如下：

```
int  sigsetjmp(sigjmp_buf  env,int  savesigs);
void  siglongjmp(sigjmp_buf  env,  int  retval);
```

与上述setjmp()和longjmp()类似，sigsetjmp()只被调用一次、返回多次

siglongjmp()被调用一次但不返回

调用sigsetjmp()后返回值为0

以后在调用siglongjmp()时触发从sigsetjmp()返回，返回值为非0。

信号的非本地跳转处理

```
sigjmp_buf buf;
void FLPhandler(int sig) {
      printf( "error type is SIGFPE!\n);
      siglongjmp(buf,1);
}
int main(){
      int a, t;
      signal(SIGFPE, FLPhandler);
      if (!sigsetjmp(buf,1)) {
            printf( "starting\n" );
            a=100;
            t=0;
            a=a/t;
      }
      printf( "I am still alive ……\n" );
      exit(0);
}
```

上述程序的执行结果如下：

```
starting
error type is SIGFPE!
I am still alive ……
```

main中通过signal()将FLPhandler()注册为SIGFPE信号对应的信号处理函数，调用sigsetjmp()的返回值为0，执行到赋值语句a=a/t；时发生整数除0异常，内核中的异常处理程序将发送SIGFPE信号给该进程，从而异步跳转到FLPhandler()执行。

在FLPhandler中，当调用siglongjmp()后，就触发sigsetjmp()函数返回1，从而跳过if分支的执行。

若将main中的语句“signal(SIGFPE, FLPhandler);”注释掉，则SIGFPE信号的处理程序是系统默认的，执行结果如下：

```
starting
Floating point exception
```