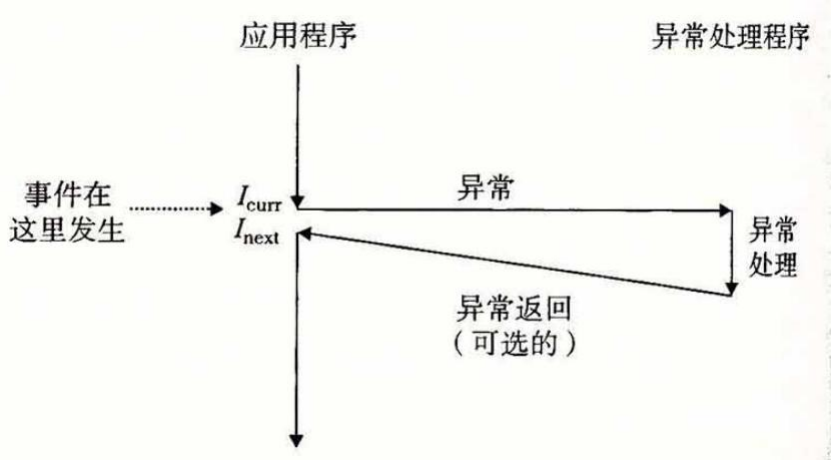
异常&进程

2024.11.13

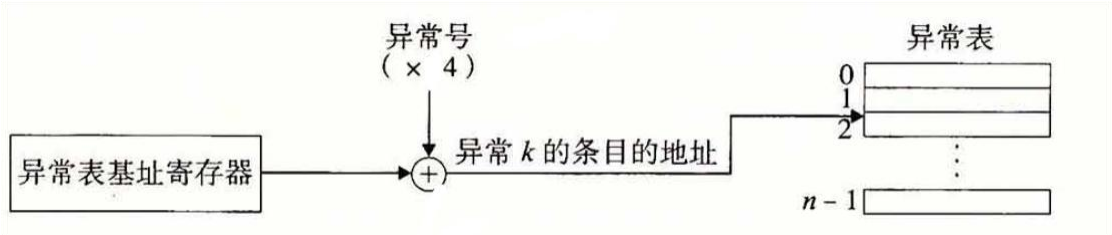
异常

- 异常是控制流中的突变，用来响应处理器中的某些状态变化，一部分由硬件实现，一部分由操作系统实现。
- 状态变化称为事件。
- 处理器检测到有事件发生时会通过一张叫做异常表的跳转表进行间接过程调用，调用异常处理程序。
- 调用完毕后：
  - 返回当前指令
  - 返回下一条指令
  - 终止被中断的程序



异常处理

- 每种种类的异常都有唯一非负整数的异常号。
- 系统启动时，操作系统分配和初始化异常表。
- 异常表的起始地址放在异常表基址寄存器中。
- ◆ 异常与过程调用的区别:
  - 异常调用结束后有三种可能。
  - 处理器也会把一些额外的处理器状态压到栈里。处理程序返回时，重新开始执行被中断的程序需要这些状态。
  - 异常处理程序运行在内核模式下，对所有的系统资源都有完全的访问权限。



异常的类别

- 四类：中断、陷阱、故障、终止
- 中断：
  - 异步发生的，是来自处理器外部I/O设备的信号的结果而不是由任何一条专门的指令造成的。
  - 处理程序返回时，总是将控制返回给下一条指令。
- 陷阱和系统调用：
  - 陷阱是有意的异常，是执行一条指令的结果。
  - 最重要的用途是系统调用。
  - 向内核请求服务：syscall n，执行这条指令时会导致一个到异常处理程序的陷阱。这个处理程序解析参数，并调用适当的内核程序。
  - 注意系统调用与普通函数调用权限不同：
    - 系统调用在内核模式。
    - 普通函数调用在用户模式。

异常的类别

- 故障：
  - 是同步的，由潜在可恢复的错误引起。
  - 若能被故障处理程序修正，则返回当前指令。
  - 否则返回到内核中的abort例程，终止引起故障的应用程序。
- 终止：
  - 是同步的，由不可恢复的致命错误造成，通常是一些硬件错误。
  - 终止处理程序从不将控制返回给应用程序，而是将控制返回给abort例程，终止应用程序。

Linux/x86-64系统调用

- 在x86-64系统上，系统调用是通过syscall陷阱指令提供。
- C程序用syscall函数可以直接调用任何系统调用。
- 所有到Linux系统调用的参数是通过寄存器而不是栈传递的：
  - 寄存器%rax包含系统调用号，参数可依次存在%rdi,%rsi,%rdx,%r10,%r8,%r9中，最多6个。

code/ecf/hello-asm64.sa

```
1   .section .data
2   string:
3     .ascii "hello, world\n"
4   string_end:
5     .equ len, string_end - string
6   .section .text
7   .globl main
8   main:
      First, call write(1, "hello, world\n", 13)
9     movq $1, %rax          write is system call 1
10    movq $1, %rdi          Arg1: stdout has descriptor
11    movq $string, %rsi     Arg2: hello world string
12    movq $len, %rdx        Arg3: string length
13    syscall                Make the system call

      Next, call _exit(0)
14    movq $60, %rax         _exit is system call 60
15    movq $0, %rdi          Arg1: exit status is 0
16    syscall                Make the system call
```

code/ecf/hello-asm64.sa

设置啥系统调用

参数，顺序参考上文

图 8-11　直接用 Linux 系统调用来实现 hello 程序

进程

- 一个执行中程序的实例。
- 系统中每个程序都运行在某个进程的上下文中。上下文是由程序正确运行所需的状态组成的。

- 逻辑控制流：“好像独占地使用处理器”
  - 一个进程中的PC值序列。
  - 进程是轮流使用处理器的，每个进程执行它流的一部分，然后被抢占（暂时挂起），然后轮到其他进程。
- 并发流：
  - 一个逻辑流的执行在时间上与另一个流重叠，称为并发流。与流运行的处理器核数或计算机数无关。
    - 并行流：两个流并发地运行在不同的处理器核或计算机上。

进程

- 私有地址空间：“好像独占地使用内存系统”
  - 进程为每个程序提供它自己的私有地址空间。
  - 地址空间底部保留给用户程序
  - 顶部保留给内核
  - 代码段总是从地址0x400000开始。

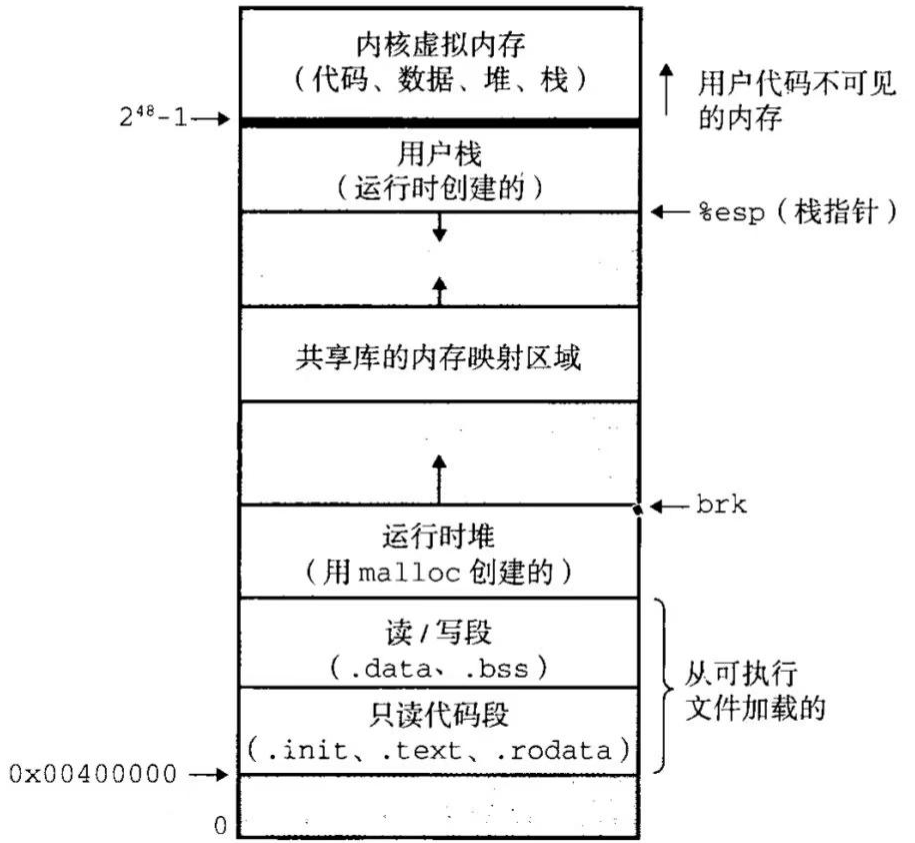


图 8-13　进程地址空间

进程

- 用户模式和内核模式:
  - 通过某个寄存器中的一个模式位实现：设置了模式位时，进程运行在内核模式，否则在用户模式。
  - 进程初始时在用户模式。
  - 异常发生时，控制传递到异常处理程序，处理器将模式从用户模式变为内核模式。
  - 处理程序运行在内核模式中。
  - 返回时，处理器把模式从内核模式改回到用户模式。

- 上下文切换：
  - 较高层形式的异常控制流。
    - 保存当前进程的上下文
    - 恢复某个先前被抢占的进程被保存的上下文
    - 将控制传递给这个新恢复的进程
  - 系统调用因为等待某个事件发生而阻塞，内核可以让当前进程休眠，切换到另一个进程，即上下文切换。
  - sleep系统调用显示地请求让调用休眠
  - 中断

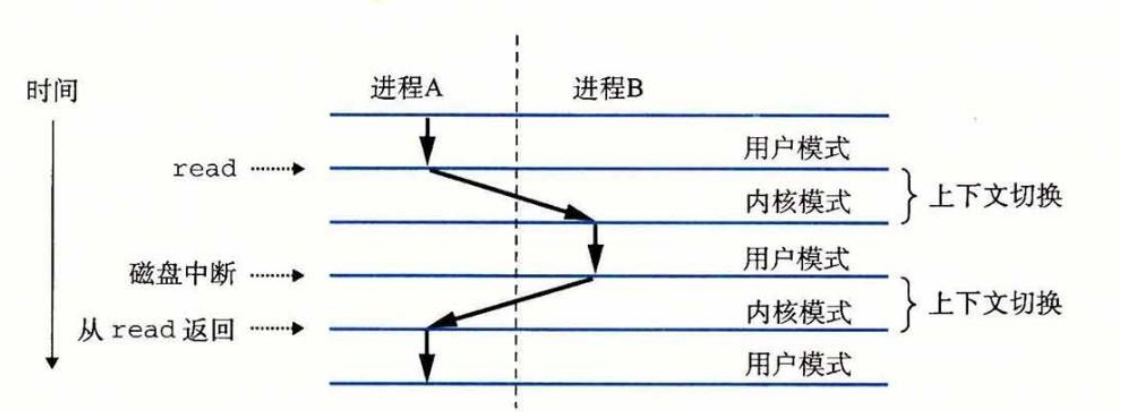


图 8-14　进程上下文切换的剖析

进程控制

- 终止进程：exit函数
  - 从不返回

```
1   int main()
2   {
3       write(1, "hello, world\n", 13);
4       _exit(0);
5   }
```

- 创建进程：fork
  - 父进程调用fork时,子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程最大的区别在于它们有不同的PID。
  - fork()特点：
    - 调用一次，返回两次：一次返回到父进程，一次返回到新创建的子进程。
    - 并发执行：父进程和子进程是并发运行的独立进程。
    - 相同但独立的地址空间。
    - 共享文件：子进程继承了父进程所有的打开文件。

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

返回：子进程返回0，父进程返回子进程的PID，如果出错，则为－1。

回收子进程

- 父进程回收已终止的子进程时，内核将子进程的退出状态传递给父进程，然后抛弃已终止的进程。
- 若一个父进程终止，内核会安排init进程接管其子进程。init进程PID为1，不会终止，是所有进程的祖先。
- 终止了但未被回收的进程称为僵死进程。
- 一个进程可以通过调用waitpid函数等待它的子进程终止或停止。
  - 挂起调用进程的执行，直至它的等待集合中的一个子进程终止。

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statusp, int options);
```

返回：如果成功，则为子进程的 PID，如果 WNOHANG，则为 0，如果其他错误，则为−1。

让进程休眠

- sleep函数：将一个进程挂起一段指定的时间。

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

返回：还要休眠的秒数。

- pause函数：让调用函数休眠，直到该进程收到一个信号。

```
#include <unistd.h>

int pause(void);
```

总是返回－1。

加载并运行程序

- execve：在当前上下文中加载并运行一个新程序。
  - 加载并运行可执行文件filename，并且带参数列表argv和环境变量列表envp。
  - 参数和环境变量格式：key-value
  - 加载filename后调用启动代码。
    int main(int argc,char **argv,char **envp)

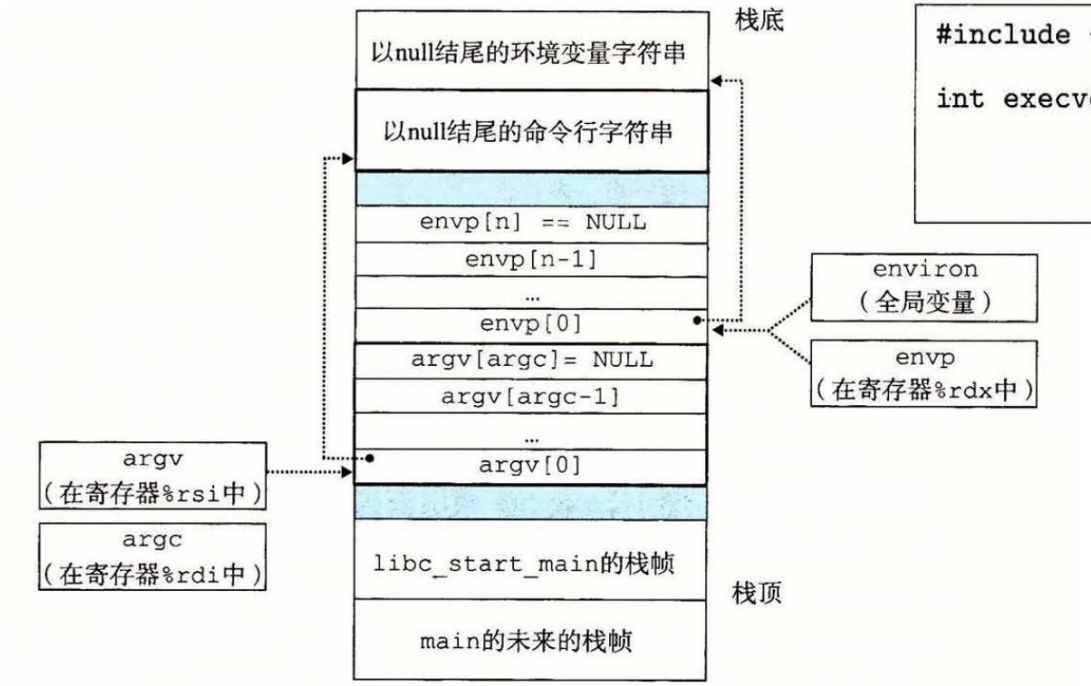


图 8-22　一个新程序开始时，用户栈的典型组织结构

```
#include <unistd.h>

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

                                    如果成功，则不返回，如果错误，则返回－1。
```

加载并运行程序

- fork和execve:
  - exec系列函数没有创建新进程，而是把当前进程对应的应用换成新的应用。
    - 如PID=1000的进程A，执行exec B，那么PID=1000的进程会变成B，A的资源会被系统回收。
    - 具体实现时，将堆栈中存储的返回指令修改为新的程序的头指令和新的堆栈地址并在调用返回时弹出，系统开始执行新程序。
  - fork作用是复制一个进程，两个进程（父进程、子进程）是并发运行的独立进程。

THANKS FOR WATCHING

2024.11.13