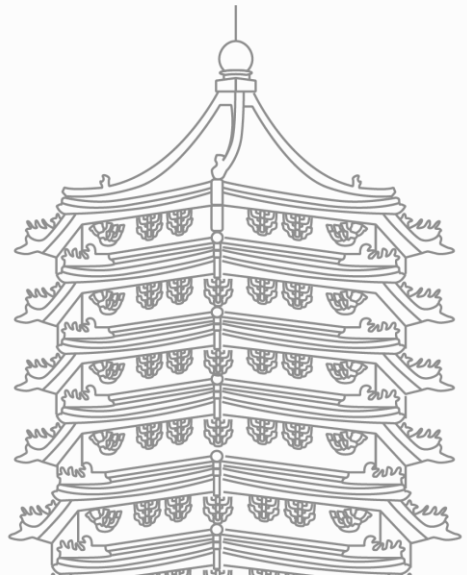
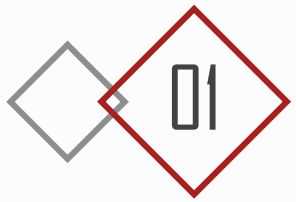
ECF

ICS 2022.11.16

TA：黄柘铳

CONTENTS





异常的概念

异常

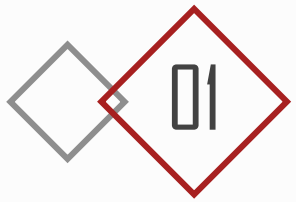
异常是异常控制流的一种形式，由硬件与软件协同实现。

异常的作用是响应处理器状态之中的某种变化，当处理器检测到有某种事件发生时，就会查询异常表(exception table)，并据此跳转到专门处理此类异常的子程序之中。

当异常处理程序运行完成之后，根据引起异常的事件类型：

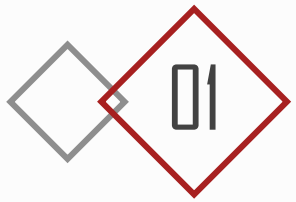
1. 返回当前指令并继续执行
2. 返回下一条指令并继续执行
3. 中断当前程序

不同的异常都有着唯一的非负整数作为异常号(exception number)，部分异常号由处理器设计师分配，部分由操作系统分配。每次触发异常时，通过异常表基址寄存器(exception base register)以及异常号索引异常处理程序的位置。



异常与调用的区别

调用	异常
返回地址一定是下一条指令	返回地址可能是当前指令，可能是下一条，也可能不返回
只把参数和返回地址压入用户栈中	会把恢复中断状态所需的额外的处理器状态（如EFLAGS）压入内核栈中
运行在用户模式	运行在内核模式（超级用户模式）

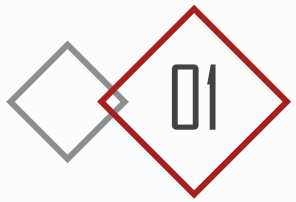


异常的分类

异常

类别	原因	是否同步	返回行为	例子
中断	外部IO设备的信号	异步	返回下一条指令	网络适配器或者磁盘控制器成功获取数据
陷阱	有意的异常	同步	返回下一条指令	通过系统调用向内核请求服务
故障	潜在可恢复的错误	同步	返回当前指令或终止	缺页异常或者除法错误
终止	不可恢复的错误	同步	不返回	DRAM或者SRAM被损坏

注意同步与异步的区别，在这里，同步可以理解为程序执行与异常的发生有着顺序的关系，可以确定什么地方会发生异常。而异步则意味着我们无法预知会在哪里，或者何时发生异常。

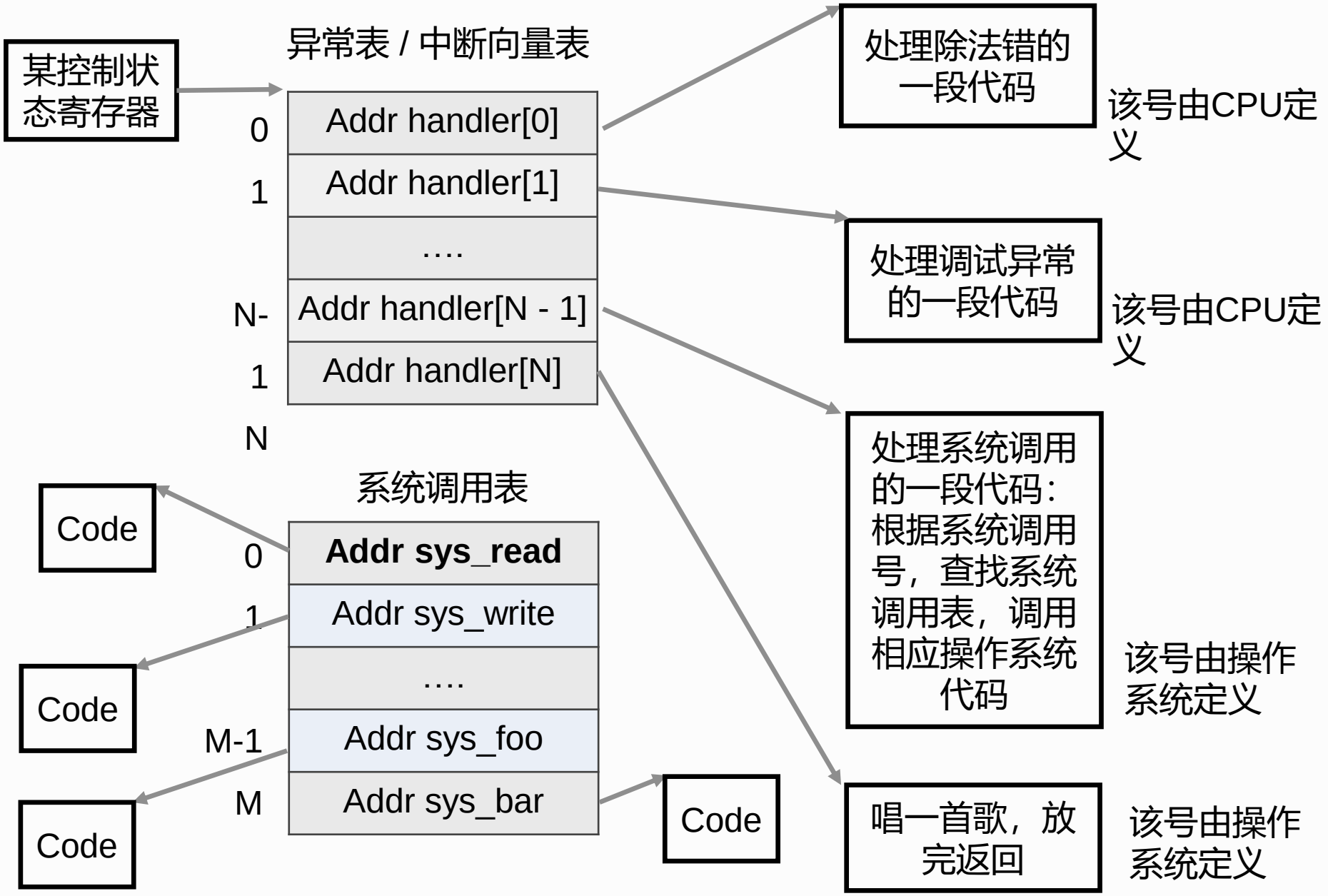


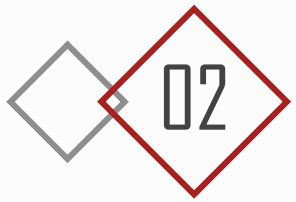
异常的分类

- 下列行为分别触发什么类型的异常?

1. 执行指令mov $57, %eax; syscall
2. 程序执行过程中，发现它所使用的物理内存损坏了
3. 程序执行过程中，试图往main函数的内存中写入数据
4. 按下键盘
5. 磁盘读出了一块数据
6. 用read函数发起磁盘读
7. 用户程序执行了指令lgdt，但是这个指令只能在内核模式下执行

1. (陷阱)
2. (中止)
3. (故障)
4. (中断)
5. (中断)
6. (陷阱)
7. (故障)


某控制状态寄存器
异常表 / 中断向量表
0
1
N-
1
N
Addr handler[0]
Addr handler[1]
….
Addr handler[N - 1]
Addr handler[N]
处理除法错的一段代码
该号由CPU定义
处理调试异常的一段代码
该号由CPU定义
处理系统调用的一段代码：根据系统调用号，查找系统调用表，调用相应操作系统代码
该号由操作系统定义
唱一首歌，放完返回
该号由操作系统定义
系统调用表
0
1
M-1
M
Addr sys_read
Addr sys_write
….
Addr sys_foo
Addr sys_bar
Code
Code
Code
Code

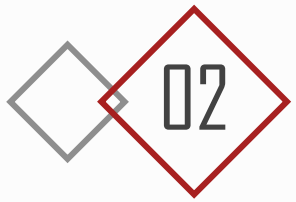


进程的概念

进程

- 进程是执行中程序的实例。
- Process is an execution environment with Restricted Rights.
- 注意区分进程与程序的概念，后者只是保存在磁盘之中的一段代码。

- 上下文(context)是该实例的状态集合，包括代码、数据、栈、通用寄存器、程序计数器、环境变量、打开文件描述符表等。



进程的特性

- 进程的特性包括：

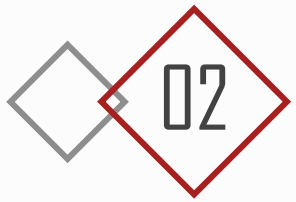
1. 逻辑控制流独立性、对每个进程而言，只有它在独自使用CPU
2. 地址空间私有，对每个进程而言，只有它在独自使用系统内存

* 操作系统为进程提供的服务：

– Protected from each other!

– OS Protected from them

– Processes provides memory protection



内核模式与用户模式 – 如何protect

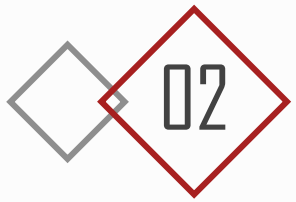
进程

- 处理器维护控制寄存器之中的模式位，从而标识当前处于用户模式还是内核模式。

- 在用户模式之下，程序无法执行特权指令，例如更改模式位或者发起IO等，也不可以访问内核地址空间。（linux可以通过/proc文件系统访问内核数据结构内容）
- 而在内核模式之下，可以执行任何指令，也可以访问任何内存位置。

- 区分用户与内核模式，通常是为了限制用户行为，从而保护系统和硬件资源。

- 用户代码总是要运行在用户态下。处理异常的操作系统代码一般需要运行在内核态下。如果需要使用到用户态不能提供的服务，那么必须在内核态下进行（陷阱存在的意义？）



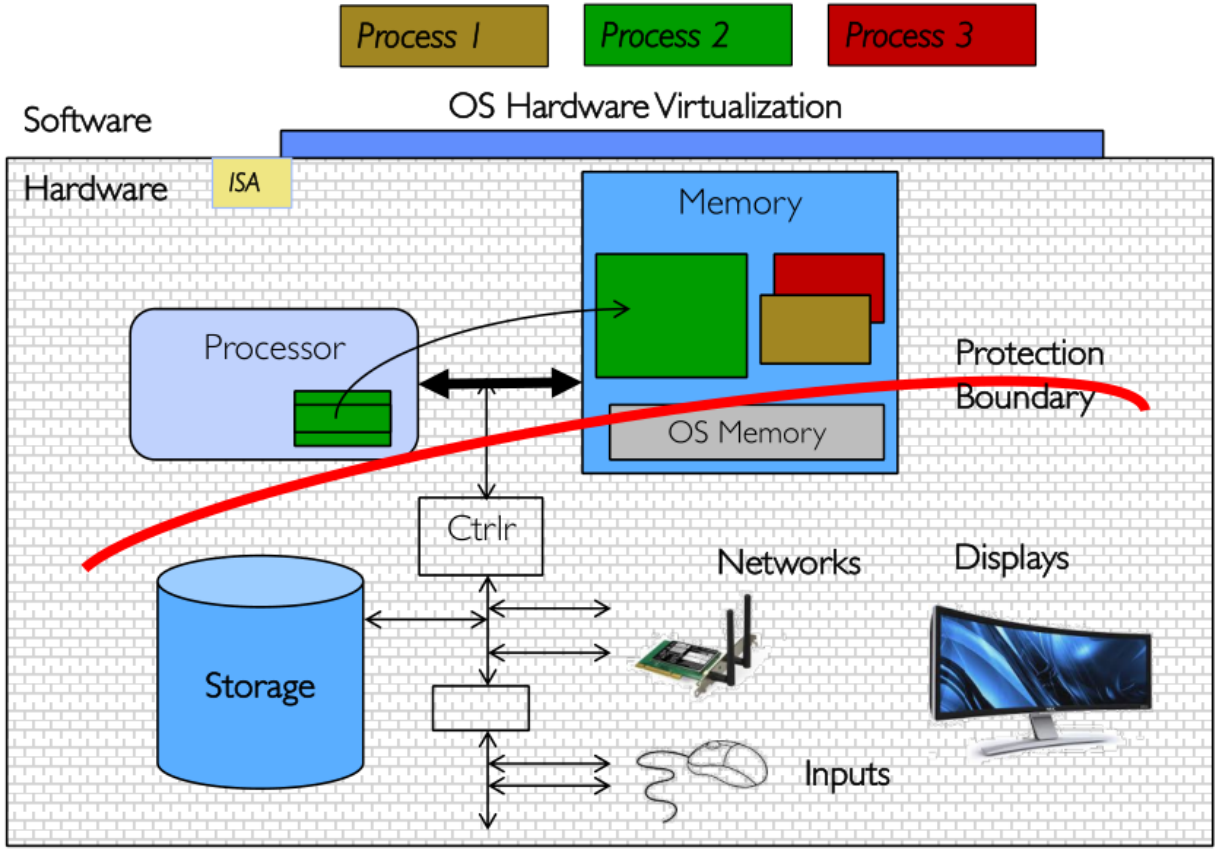
内核模式与用户模式

进程

以下哪些操作需要特权?

访问磁盘	(需要)
访问IO设备	(需要)
清除中断标志	(需要)
修改条件码	(不需要)
修改模式位	(需要)

Protection



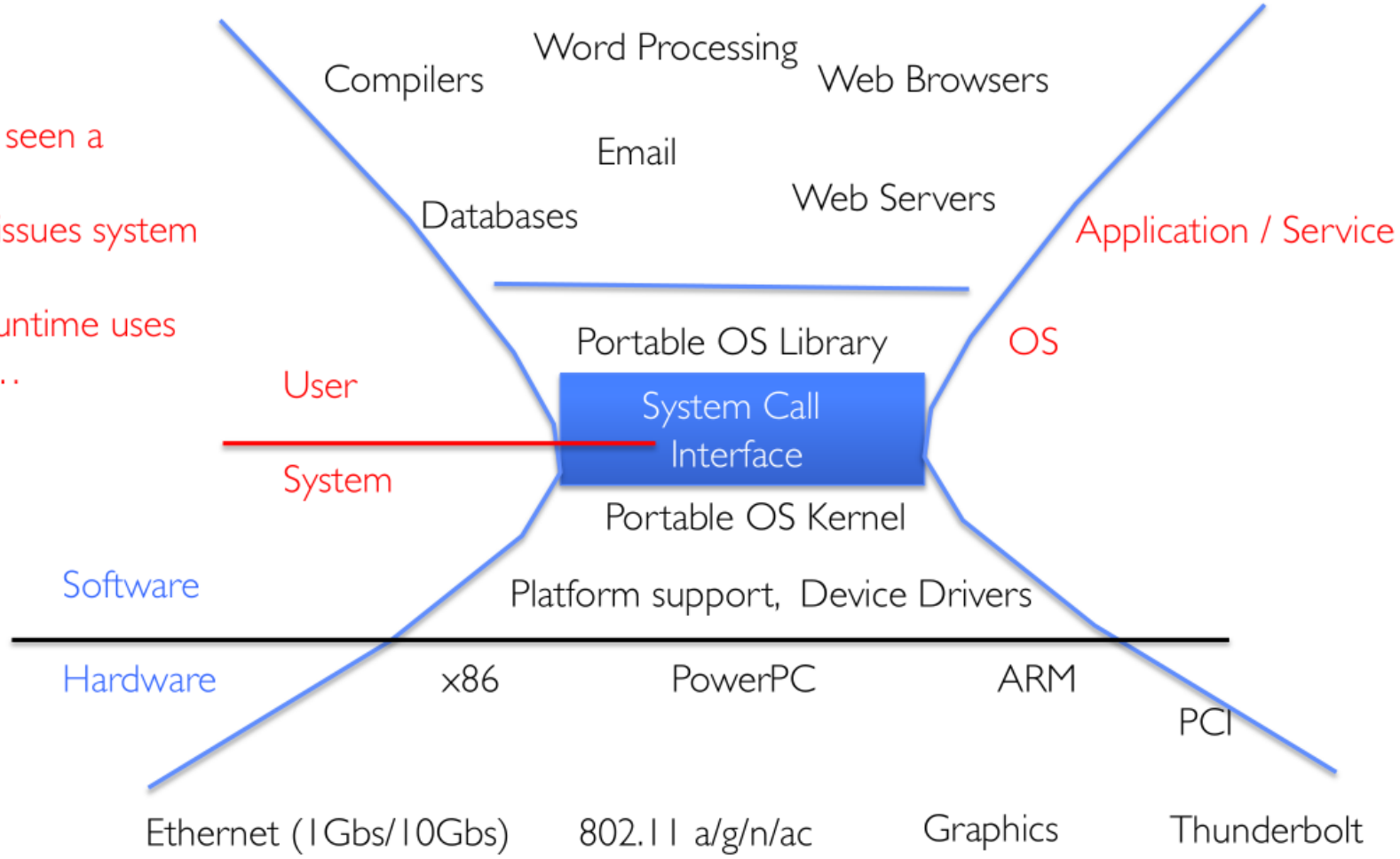
- OS *isolates* processes from each other
- OS *isolates* itself from other processes
- … even though they are actually running on the same hardware!

进程如何申请对于关键资源的使用 - syscall

System Calls ("Syscalls")

"But, I've never seen a syscall!"

- OS library issues system call
- Language runtime uses OS library…



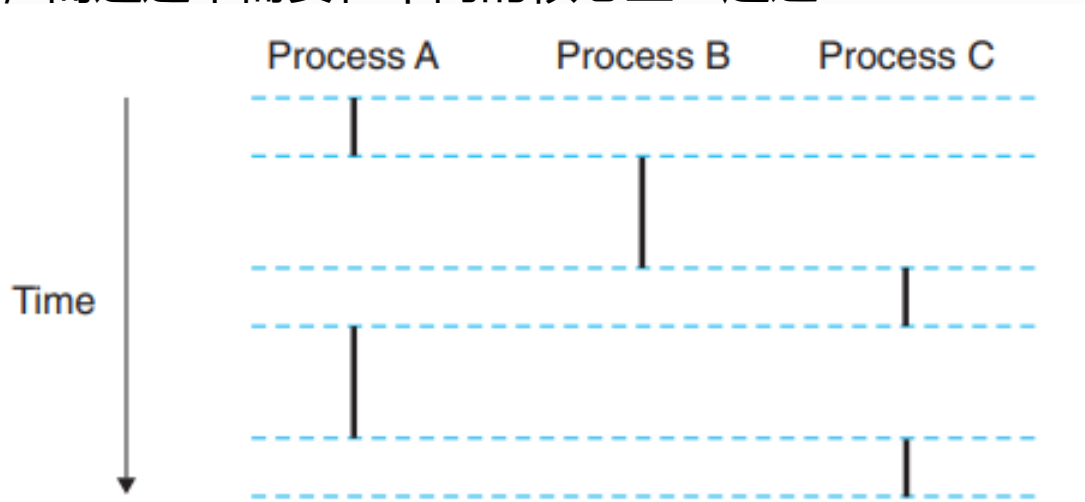
逻辑流与并发流

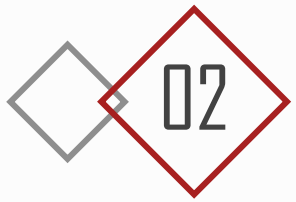
进程

- 在整个系统能不断运行的过程之中，程序计数器(PC)的值会不断变化，这种变化反映着系统在不同的进程的执行之间来回切换，因而被称为逻辑控制流（逻辑流）

- 一个逻辑留在执行时间上与另一个发生重叠，则被称为并发流。

- 注意并发与并行的区别：前者是后者的超集，只要逻辑流有交错就可以被称为并发，但是只有真正在同一时间运行才被称为并行，而这通常需要在不同的核心上一起运行。





逻辑流与并发流

进程

- 考虑如下进程：

进程	开始时间	结束时间	运行处理器
A	5	7	P0
B	2	4	P1
C	3	6	P0
D	1	8	P1

- 判断以下进程对是否并行、并发：AB、AC、AD、BD
- AB都不是、AC并发不并行、AD并发且并行、BD并发不并行

- 实际系统中，有非常多的进程在并发执行

02 上下文切换与调度

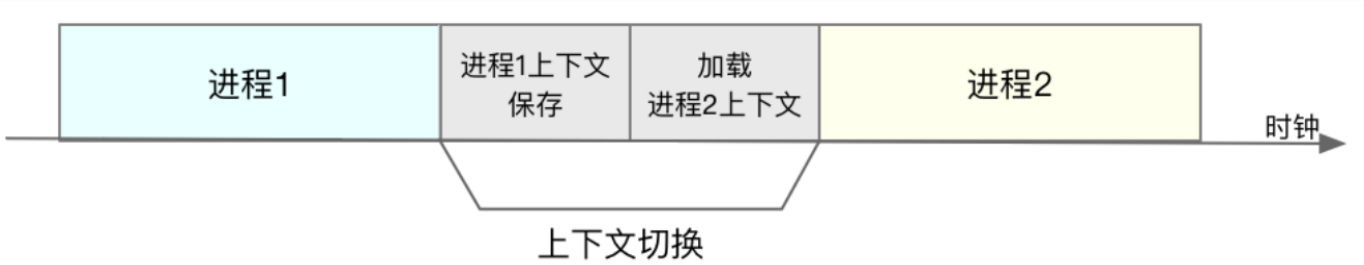
进程

- 内核通过上下文切换来实现控制流在不同进程之间的转移。其流程是：

1. 保存当前进程的上下文(到内核栈): 这样之后才能恢复

2. 恢复某个先前进程(要去的进程)的上下文

3. 将控制转移到要去的进程：更改PC等



内核决定切换进程，并选择下一个执行的进程的决策被称为调度（scheduling）。
调度可以分为抢占式调度（内核强制暂停某个进程，例如发现其它某个进程的IO完成）以及非抢占式调度（进程自动让出控制，例如进程通过pause()等API自动挂起）

上下文切换由内核完成，因而处于内核模式。

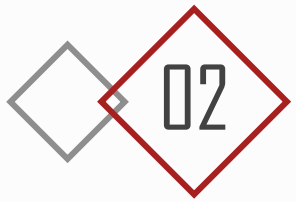
进程的状态

- 从程序员的角度看，我们可以认为每个进程都处于下面三种状态之一

1. 运行（正在CPU上执行或已经做好准备，等待被内核调度）
2. 停止 （进程被挂起并且暂时不会被内核调度）
3. 终止（进程永远停止运行）

内核为每一个进程维护了一个PCB（process control block）用以存放这个进程的相关信息，这个结构处于内核地址空间之中，对于用户而言不可见。

PCB之中一条重要的信息就是进程的PID（正整数），用以标识这个进程的身份。特别的，PID为1的进程是init进程，这是所有进程的共同祖先。



进程的控制及相关API

- 进程相关的API以及代码可以查看官网 http://csapp.cs.cmu.edu/3e/code.html

```
pid_t getpid(void);//获取调用者的进程PID
pid_t getppid(void);//获取调用者的父进程PID
```

```
void exit(int status);//以整数status为状态退出进程
unsigned int sleep(unsigned int secs);//让进程挂起一段时间,返回剩余的休眠秒数
int pause(void);//挂起进程，直到其收到一个信号。返回值永远是-1
```

```
pid_t fork(void);//子进程返回0，父进程返回子进程的pid
int execve(cons char* filename, const char* argv[],
      const char *envp[]);//加载并允许程序，成功则不返回，反之返回-1
pid_t waitpid(pid_t pid,int *stausp,int options);//成功时返回子进程的pid或者0，出错返回-1
```

进程的控制及相关API

- 关于进程，以下说法正确的是：

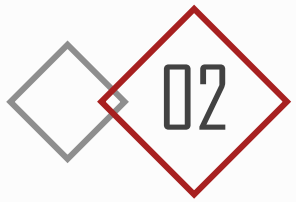
A. 没有设置模式位时，进程运行在用户模式中，允许执行特权指令，例如发起I/O操作。

B. 调用waitpid(-1, NULL, WNOHANG & WUNTRACED)会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回0；如果有停止或终止的子进程，则返回其中一个的PID。

C. execve函数的第三个参数envp指向一个以null结尾的指针数组，其中每一个指针指向一个形如name=value的环境变量字符串。

D. 进程可以通过使用signal函数修改和信号相关联的默认行为，唯一的例外是SIGKILL，它的默认行为是不能修改的。

C



进程的创建 – fork syscall

- 调用fork函数会复制当前进程的上下文并据此创建一个新的进程，称为当前进程的子进程。

- 因此在父进程与子进程之中fork都会返回一个pid_t，区别在于父进程返回的是子进程的pid而子进程返回的是0，程序可以据此区分自己是父进程还是子进程。这也是“调用一次，返回两次”说法的缘由。

- 通过绘制进程图来确定进程之间，指令执行的先后顺序

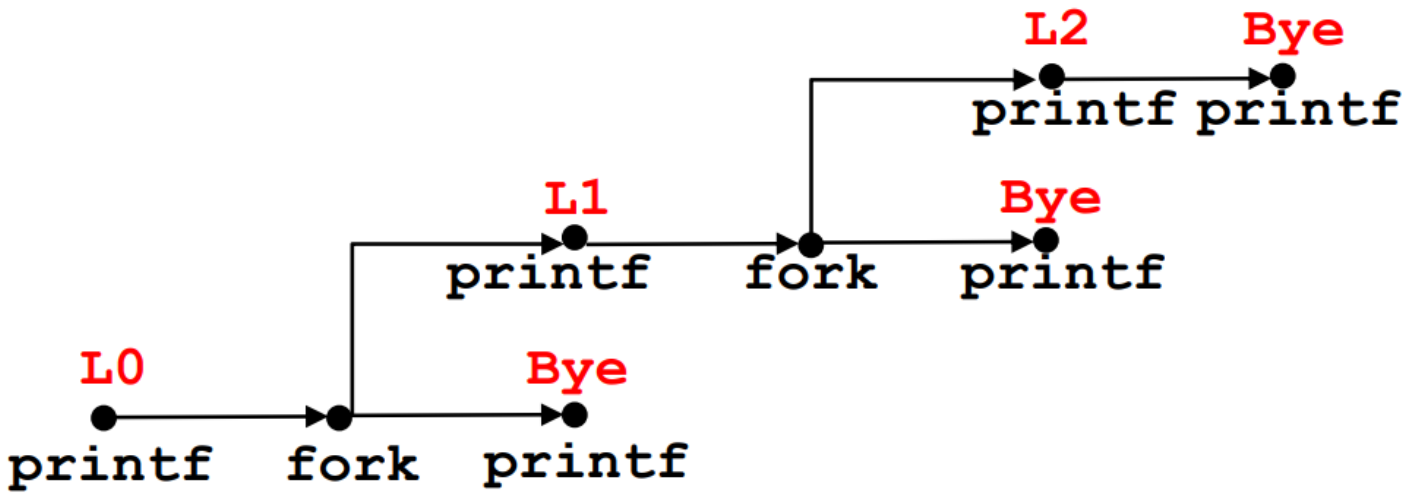
- 我们通常会使用fork+execve来创建一个子进程，并让它运行一个新的程序，execve会加载一个新的程序来运行，并重新组织进程的上下文。

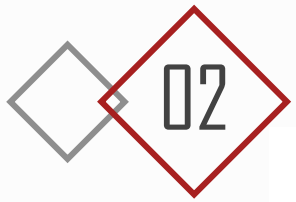
fork

进程

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

forks.c





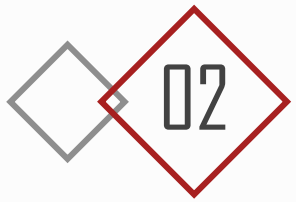
fork

11.在系统调用成功的情况下，下列代码会输出几个 hello？（　　）

```
void doit()
{
    if ( fork() == 0 ) {
        printf("hello\n");
        fork();
    }
    return ;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0) ;
}
```

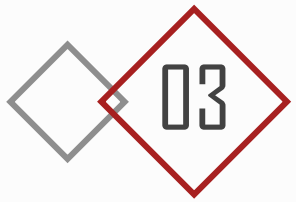
B

A. 3　　B. 4　　C. 5　　D. 6



进程如何执行、退出任务

- **exit** terminates own process
  - Called once, never returns
  - Puts it into “zombie” status
- **wait** and **waitpid** wait for and reap terminated children
- **execve** runs new program in existing process
  - Called once, (normally) never returns



信号的概念

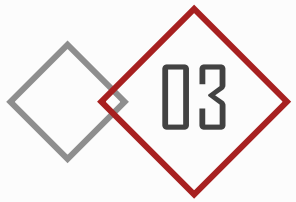
信号

- 之前提到的异常属于底层异常机制，对用户进程不可见。

- 但很多时候，最基本的行为无法满足要求，需要显式地处理异常. 所以引入了Linux信号。这是一种更高层次的，软件层次的异常。

- 信号允许进程或者内核发送信息或者中断其他的进程，也可以用于通知进程发生了某种类型的事件需要处理。如下是一些常见的信号：

名称	默认行为	相应事件
SIGINT	终止	Ctrl+C
SIGQUIT	终止	Ctrl+\
SIGKILL	终止	杀死进程
SIGALRM	终止	来自alarm的定时器信号
SIGSEGV	终止并转储内存	无效内存引用
SIGFPE	终止并转储内存	浮点异常
SIGCHLD	忽略	子进程停止/终止
SIGTSTP	停止直到下一个SIGCONT	Ctrl+Z
SIGSTOP	停止直到下一个SIGCONT	非终端停止信号



信号的概念

信号

- 信号一般存在三种可能的状态：

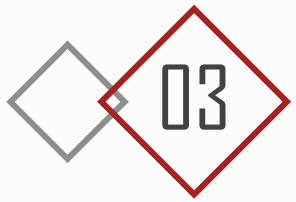
1. 正在被处理
2. 待处理(已经发出但是还没有被接收)
3. 丢弃

每个进程有一个pending位向量存储所有待处理的信号，以及一个blocked位向量存储所有阻塞的信号。

对于一个已经发出的信号，当进程正在处理某个信号或者这个信号被进程所堵塞，则将设置对应的pending中的相应位置, 表示这个信号待处理。如果这个信号已经是待处理的状态，则将简单的丢弃这个信号。

当进程接受了某个信号时，就会将对应的pending中的位置清除，表示接受了这个信号。

基于待处理信号的处理逻辑，不难看出信号具有不排队的属性，即相同信号只能有一个待处理。



发送信号

信号

信号通常可通过3种方式来发送：

1. 在shell之中发送。

输入 /bin/kill/ -signal pid 可以向进程pid发送信号signal，如果pid为负数则向着|pid|这个进程组发送信号signal。如常见的/bin/kill/ -9 pid 指令用于终止进程。

2. 通过键盘发送

输入 ctrl+C可以发送SIGINT（终止）到前台进程组之中的每个进程，输入ctrl+Z可以发送SIGTSTP（挂起）到前台进程组之中的每个进程。

3. 调用相关API进行发送

- int kill(pid_t pid,int sig);//根据pid是正，负，0发送信号sig给对应的进程
- unsigned int alarm(unsigned int secs);//在secs秒之后向自己发送SIGALRM信号

接收信号

信号

- 接收信号的时机：从内核模式切换回用户模式之中时，例如系统调用或者上下文切换完成。

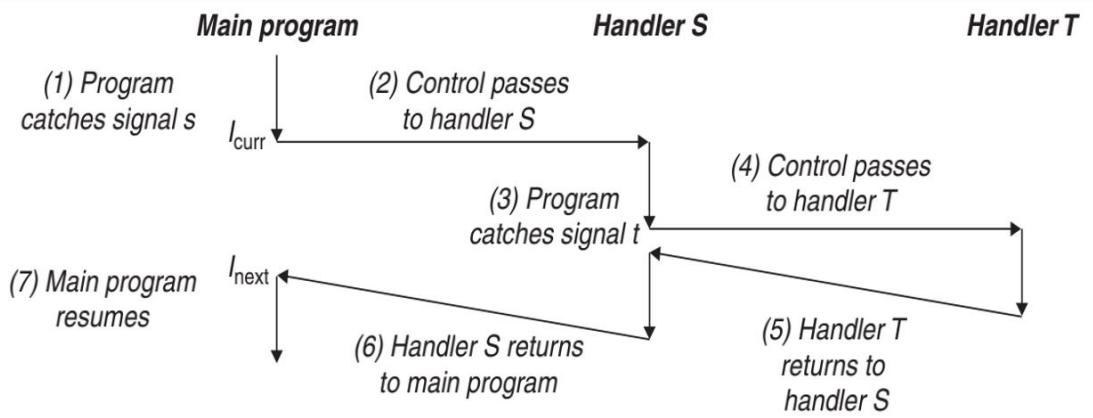
- 此时进程会检查pending位向量，并选择一个待处理的信号来处理。在执行完这个信号处理程序之后，（如果可以正常执行）再返回其其逻辑流之中的下一条语句正常执行。

- 每种信号都有着预定义的处理程序，例如SIGKILL默认行为是终止进程，SIGCHLD默认行为是忽略信号，但是也可以编写自己的信号处理程序并调用

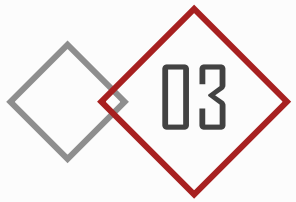
```
sighandler_t signal(int signum,sighandler_t handler);
```

这个API将对于信号signum的处理程序替换为handler

注意唯二的例外是SIGSTOP与SIGKILL，其行为无法被更改。



同时，信号处理程序在其执行过程之中也可能被中断，因此在编写之中需要注意并发的安全问题。



编写信号处理程序的原则

1. 安全性

- G0: handler尽可能简单
  G1: 只调用异步信号安全的函数(可重入/不能被信号处理程序中断)
  safe: _exit, kill, sleep, wait, waitpid, write
  unsafe: exit, printf, sprintf, malloc
  G2: 保存和回复errno
  G3: 访问全局数据结构时, handler和主程序都应该暂时阻塞信号
  G4: 用violatile声明全局变量
  G5: 用sig_atomic_t来声明全局标志 (单个读写是不可中断的)

2. 正确性

避免由于信号不排队的特性导致的信号丢弃，或者父子进程之间race等而产生的并发错误。

3. 可移植性

确保所调用的API在不同平台上运行时行为一致。 （Code once, run everywhere?）

信号的阻塞与等待

信号

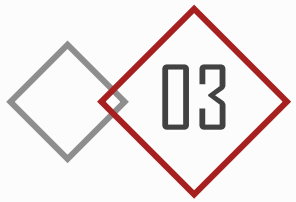
- 信号可以隐式或者显式的进行堵塞
- 隐式阻塞: 内核默认行为. 阻塞当前在处理的信号
- 显式阻塞: sigpromask函数和它的辅助函数

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
                                    Returns: 0 if OK, −1 on error

int sigismember(const sigset_t *set, int signum);
                                    Returns: 1 if member, 0 if not, −1 on error
```

- 除了从内核转到用户模式的情况，程序也可能会显式地等待某个信号进行处理：
- 使用前文提到的pause()或者sleep()等API会出现竞争或者低效等问题，正确的方式是使用sigsuspend函数：

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
                                    Returns: −1
```

- 这个函数会用mask替换当前的堵塞信号集合，并将当前进程挂起，直到收到某个信号之后再返回。并恢复原有的堵塞信号集合。



非本地跳转

信号

- 非本地跳转是c语言提供的一种用户级ECF，支持将控制直接从一个函数转移到另一个当前正在执行的函数函数，不需要经过正常的调用-返回序列。

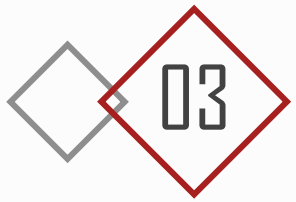
```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
                                    Returns: 0 from setjmp, nonzero from longjmps
```

- Setjmp（调用一次，返回多次）：
- 1.保存当前调用环境(在env缓冲区中)、
- 2.返回0
- 调用环境: 程序计数器, 栈指针, 通用寄存器

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
                                    Never returns
```

- Longjmp（调用一次，从不返回）：
- 1.恢复调用环境(从env缓冲区)
- 2.将控制转移到最近一次初始化env的setjmp的地址(返回retval)

- 主要用于从深层嵌套函数之中立刻返回（类比c++与java之中的try-catch原语），但此时需要预防内存泄露等问题。
- 或者让信号处理程序分支到某个特殊的代码位置



非本地跳转

信号

- 下面关于非局部跳转的描述，正确的是：

A. setjmp可以和siglongjmp使用同一个jmp_buf变量
B. setjmp必须放在main()函数中调用
C. 虽然 longjmp 通常不会出错，但仍然需要对其返回值进行出错判断
D. 在同一个函数中既可以出现setjmp，也可以出现longjmp

D

ECF

END

PKU ICS derives from CMU ICS, but is bound to go beyond CMU ICS.