ECF: Signals & Nonlocal Jumps

李之怡

2023.11.29

Page Fault

- 触发硬件异常： 当进程尝试访问无效的内存地址时，可能会触发硬件异常。这通常是由于内存管理单元（MMU）或其他硬件机制检测到了非法的内存访问。
- 产生缺页中断（Page Fault）： 如果访问的地址对应的页面不在内存中，操作系统会产生一个缺页中断，请求将相应的页面加载到内存中。
- 内核中的异常处理程序： 操作系统内核捕获到缺页中断后，会执行一个异常处理程序。这个程序负责处理页面错误，通常的操作是将相应的页面从磁盘加载到内存中，或者标记页面为无效。
- 判断野指针并发送信号： 内核在处理缺页中断的同时，可能会检测到进程访问了野指针。这时，操作系统可能会向进程发送一个信号，通常是SIGSEGV（Segmentation Fault）信号，表示进程试图访问无效的内存。
- 信号处理： 进程接收到SIGSEGV信号后，信号处理函数/用户终止。

Review: Process & Shell

- Shell
  - Shell 允许用户执行系统中已编译的可执行程序，并创建一个新的进程
- 进程
  - 一个执行中程序的实例
  - 进程轮流使用处理器；内核可以切换进程（上下文切换）
  - 每个进程都有一个唯一的 PID（正数）
    - Waitpid 等待子进程结束
    - Getpid 获取当前进程
  - 父进程&子进程
    - 一个进程可以创建新进程（子进程）（fork），而子进程会继承其父进程的大部分属性和环境

信号

- 在进程系统中，可以通过信号来控制进程
- 信号：

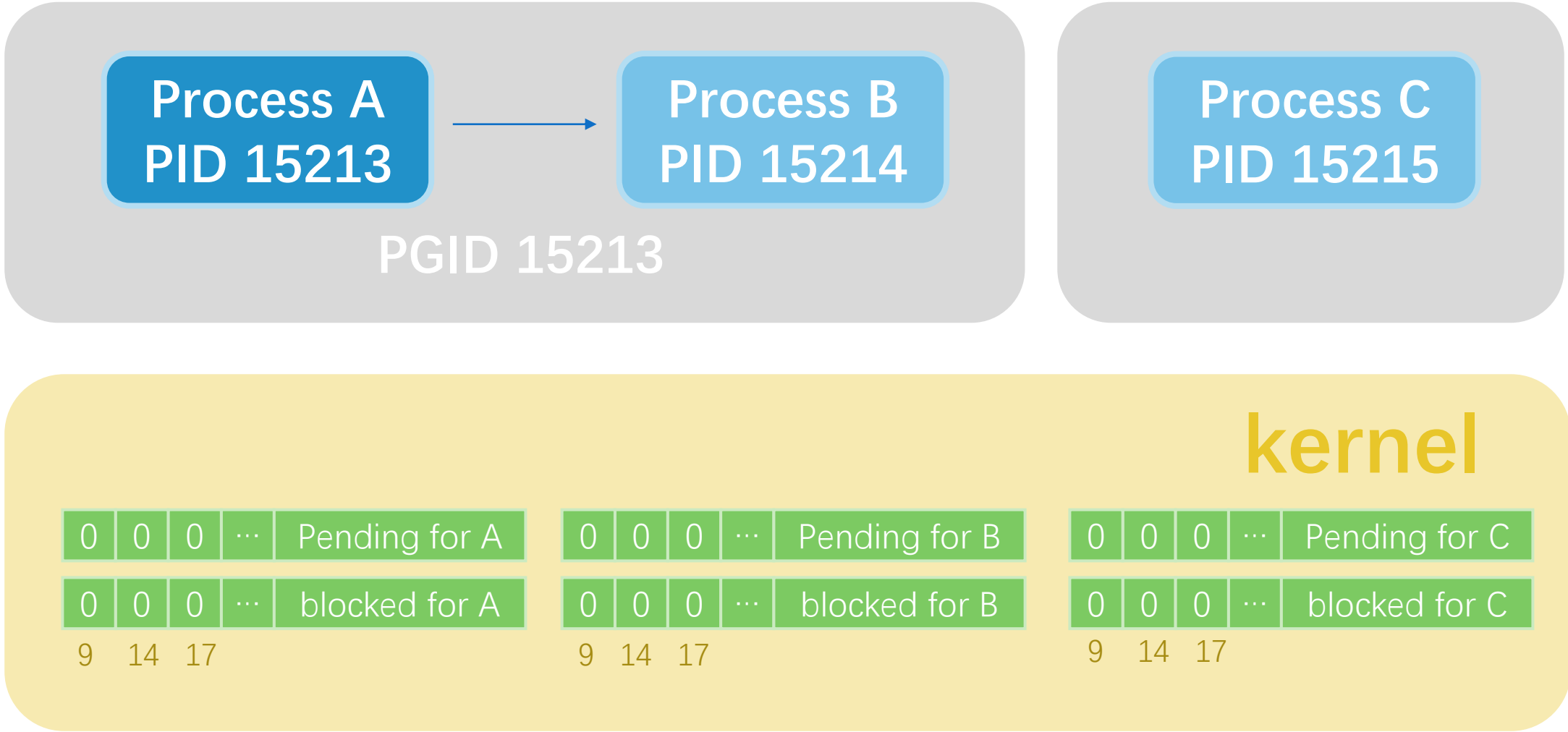
序号	名称	默认行为	相应事件
2	SIGINT	终止	来自键盘的中断
9	SIGKILL	终止	杀死程序
17	SIGCHLD	忽略	子进程停止或终止

  - Linux信号一共有30个
- 发送方：内核，其他进程，终端
- 发送原因：内核检测到系统事件；进程调用kill函数

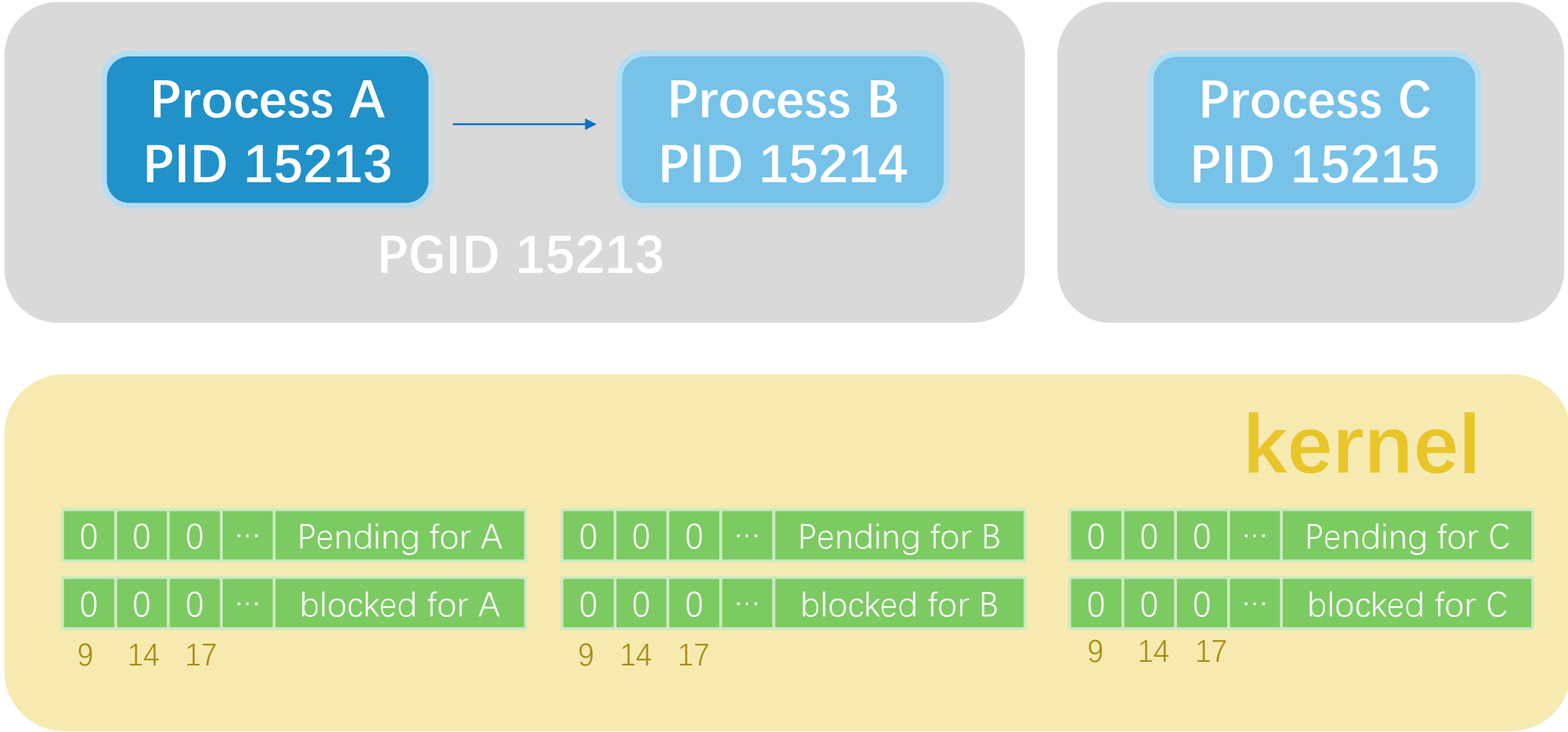
发送信号

- 用键盘发送信号
- 用/bin/kill程序发送信号
  - `Linux>  /bin/kill -9 -15213`
- 用kill函数发送信号
  - `int kill(pid_t pid, int sig);`
- 用alarm函数向自己发送信号
  - `unsigned int alarm(unsigned int secs);`
  - 返回值为前一次闹钟剩余秒数
  - 函数安排内核在secs秒后发SIGALRM信号给调用进程

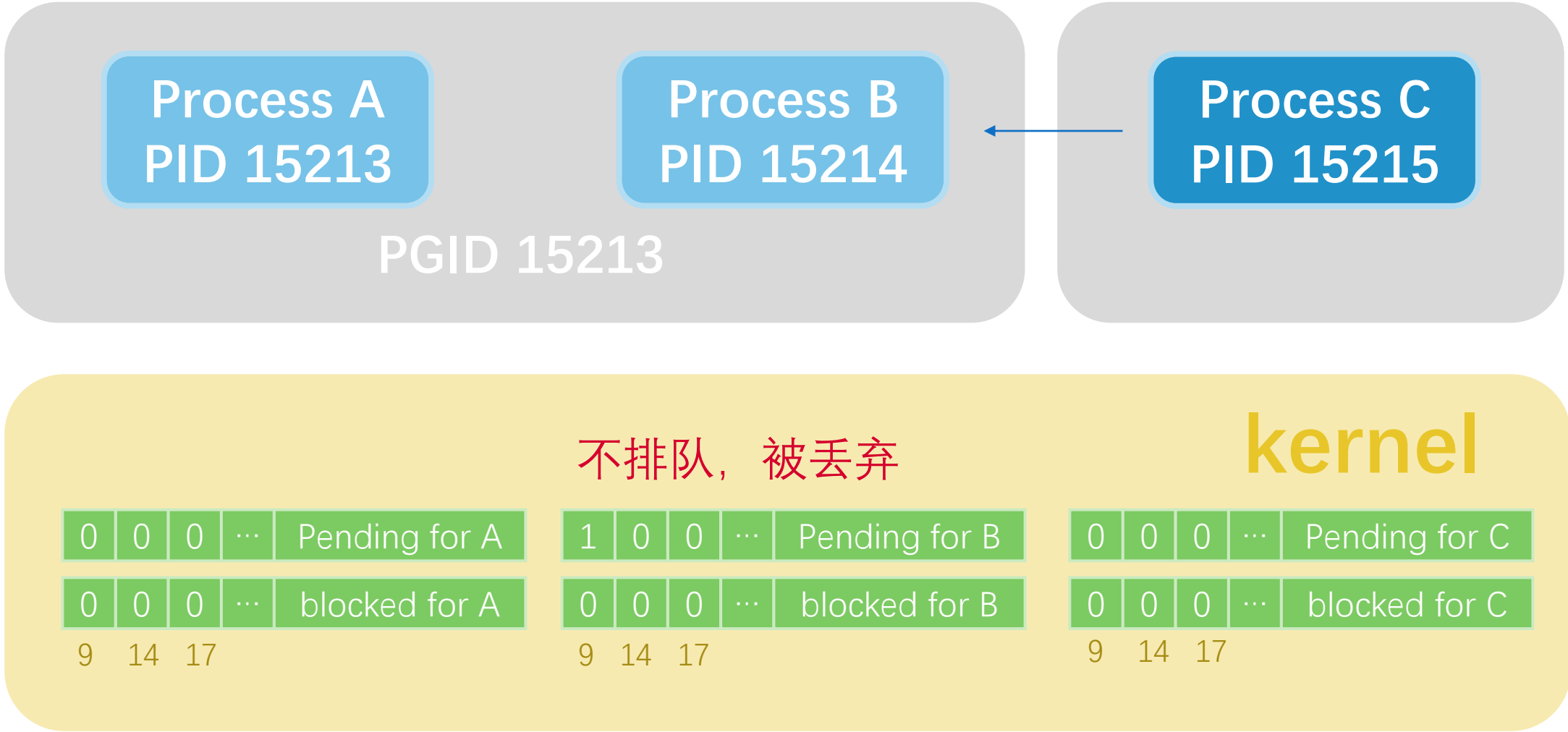
进程组&pending&blocked



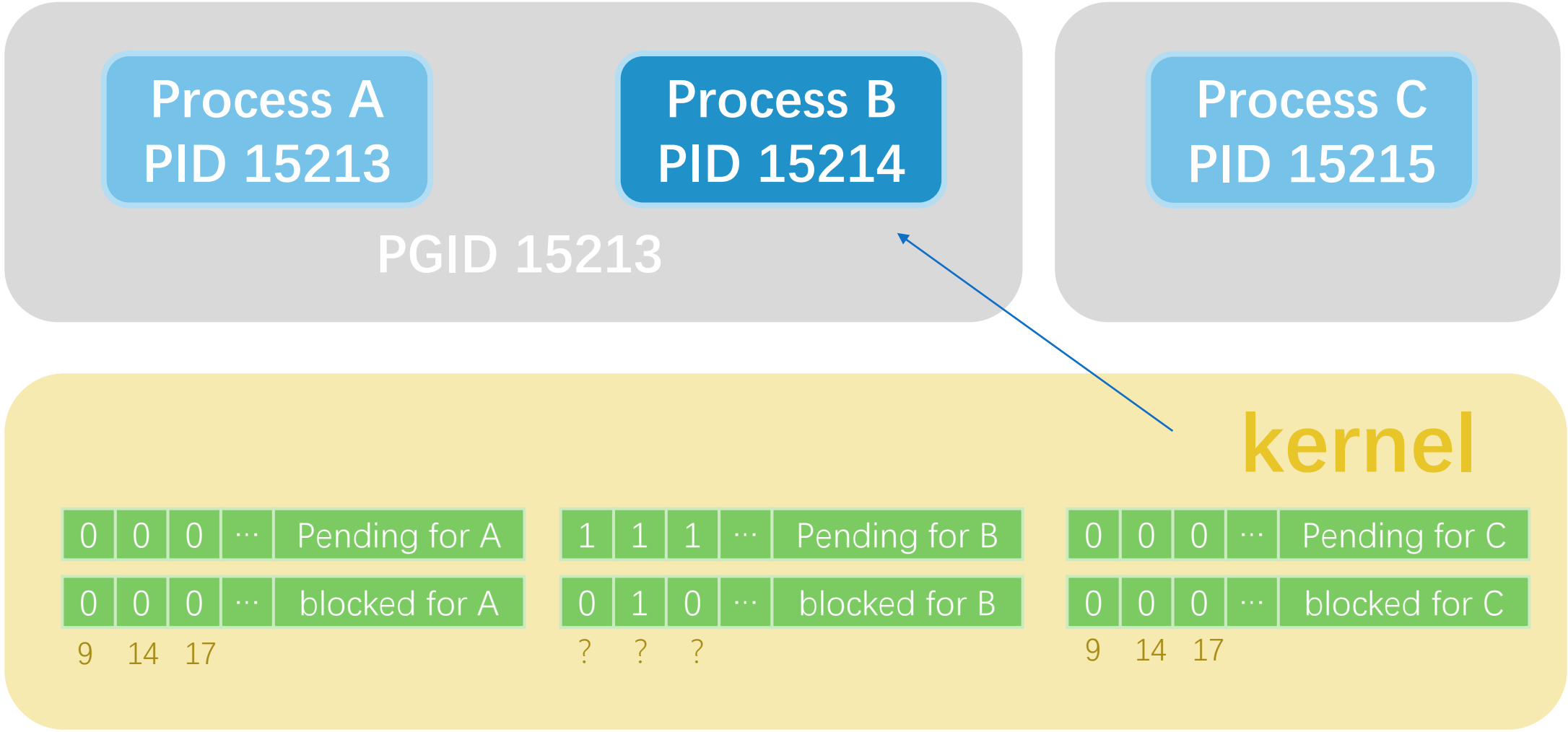
Process A: kill(15214, SIGKILL)



Process C: kill(15214, SIGKILL)

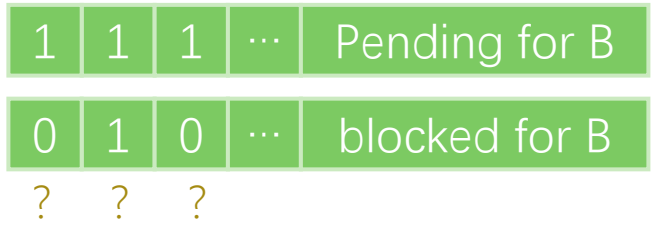


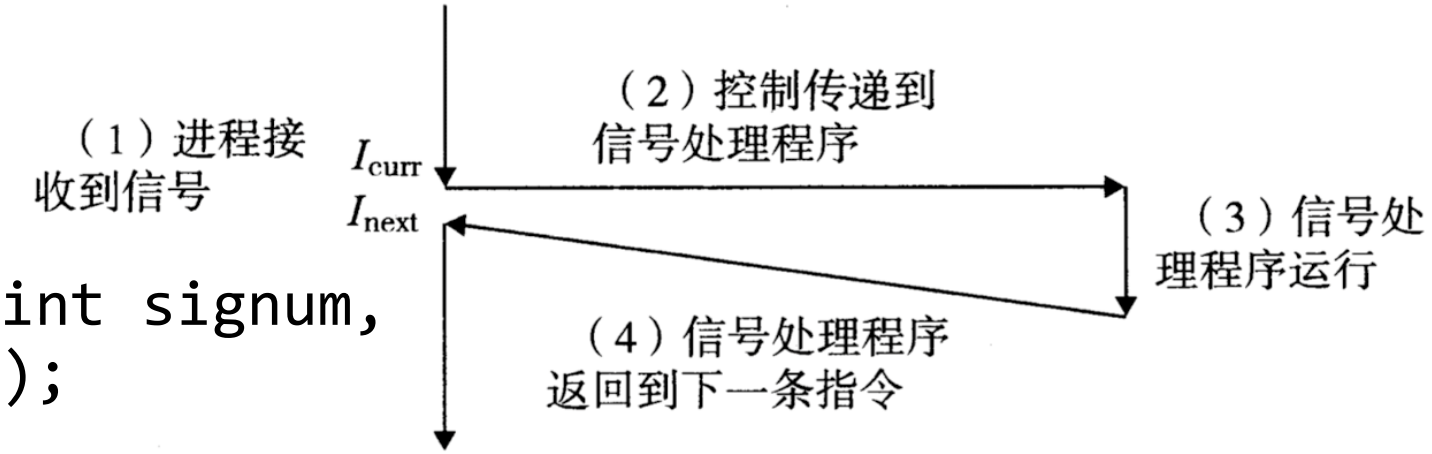
接收信号：内核模式切换至用户模式



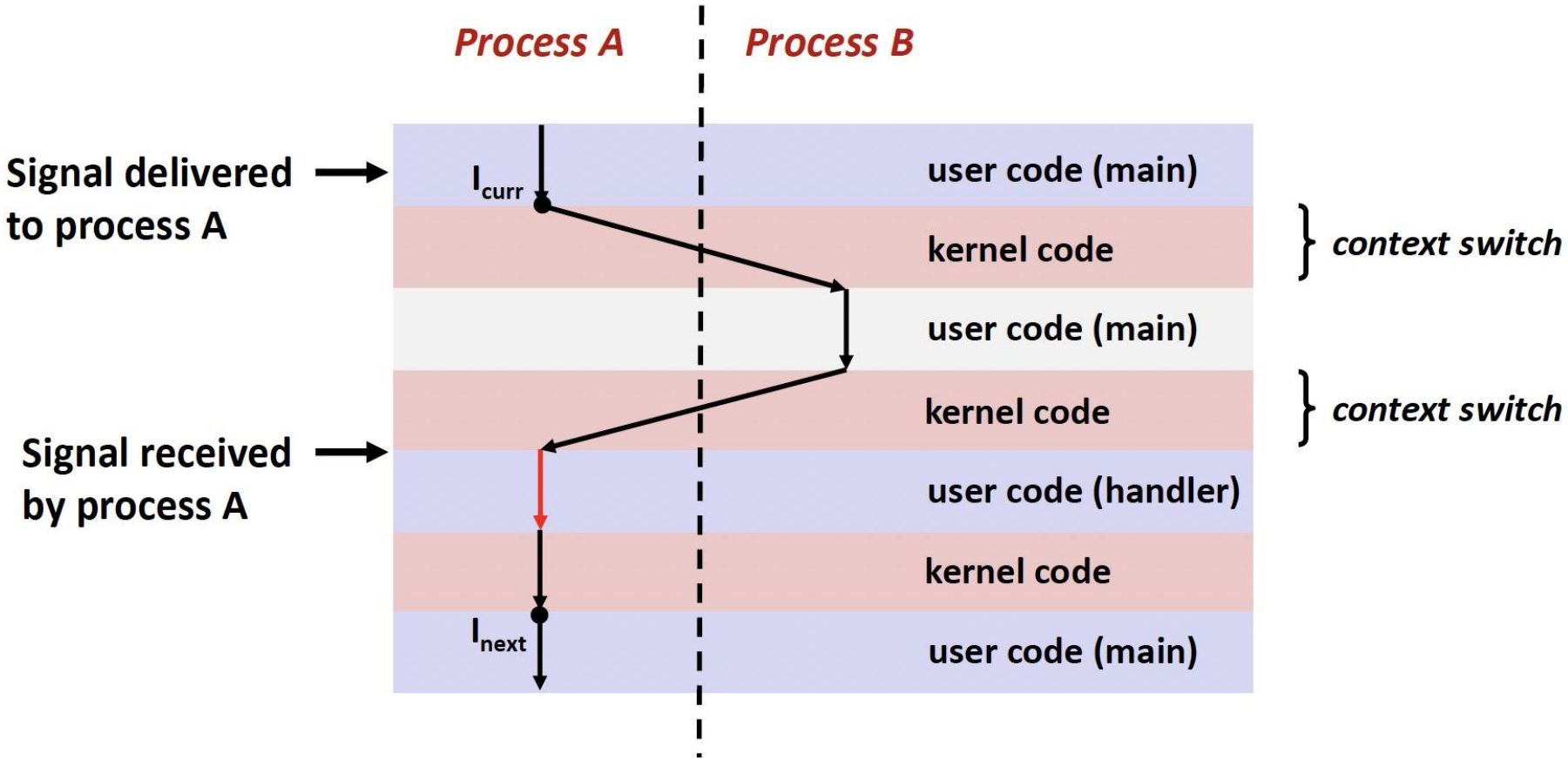
接收信号

- 默认行为
  - 忽略
  - 终止
  - ……
- 修改默认行为
  - Signal函数
  - `sighandler_t signal(int signum, sighandler_t handler);`
  - 用户定义
  - SIGSTOP&SIGKILL





Another View of Signal Handlers as Concurrent Flows



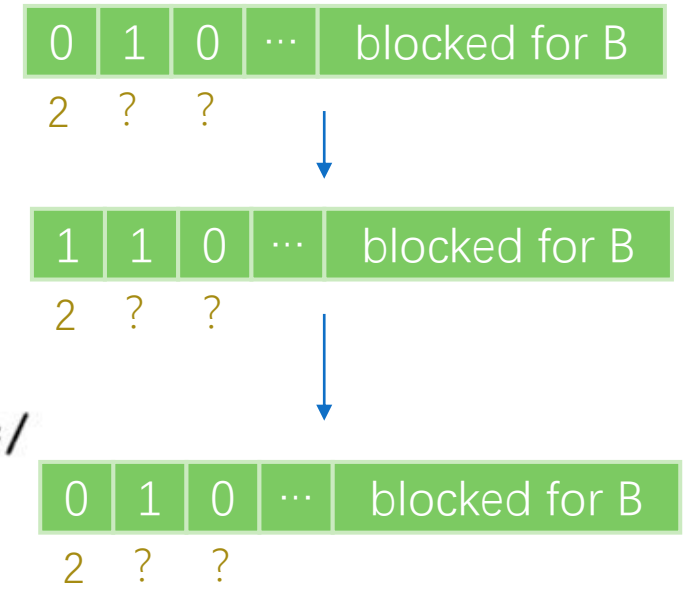
阻塞和解除阻塞信号

- 隐式阻塞机制：当前正在处理的信号类型的待处理的信号
  - 假设程序捕获了信号 s，当前正在运行处理程序 S。如果发送给该进程另一个信号 5，那么直到处理程序 S 返回，s 会变成待处理而没有被接收。
- 显式阻塞机制：sigprocmask函数
  - `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`

sigprocmask函数

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`

```
1     sigset_t mask, prev_mask;
2
3     Sigemptyset(&mask);
4     Sigaddset(&mask, SIGINT);
5
6     /* Block SIGINT and save previous blocked set */
7     Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
8     . . .  // Code region that will not be interrupted by SIGINT
9     /* Restore previous blocked set, unblocking SIGINT */
10    Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```



安全信号处理的原则

- 处理程序要尽可能简单
  - 处理程序可能只是简单地设置全局标志并立即返回；所有与接收信号相关的处理都由主程序执行，它周期性地检查(并重置)这个标志。
- 保存和恢复errno
  - 许多 Linux 异步信号安全的函数都会在出错返回时设置errno。在处理程序中调用这样的函数可能会干扰主程序中其他依赖于 errno 的部分。解决方法是在进人处理程序时把 errno 保存在一个局部变量中，在处理程序返回前恢复它。注意，只有在处理程序要返回时才有此必要。如果处理程序调用exit终止该进程，那么就不需要这样做了。

安全信号处理的原则

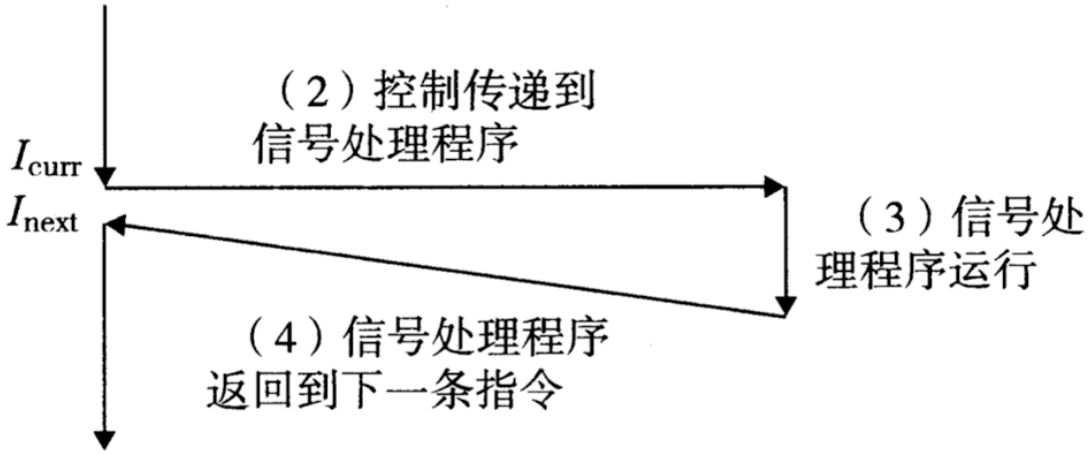
- 在处理程序中只调用异步信号安全的函数
  - malloc, printf, exit
  - write, sio_putl, sio_puts, sio_error
  - 可重入/不能被信号处理中断
    - 多线程；数据共享；被中断
- 阻塞所有的信号，保护对共享全局数据结构的访问
  - 从主程序访问一个数据结构 d 通常需要一系列的指令，如果指令序列被访问 d 的处理程序中断，那么处理程序可能会发现 d 的状态不一致，得到不可预知的结果。在访问 d时暂时阻塞信号保证了处理程序不会中断该指令序列。

安全信号处理的原则

- 用volatile声明全局变量
  - 用 volatile 类型限定符来定义一个变量，告诉编译器不要缓存这个变量。例如 volatile int g;
  - volatile 限定符强迫编译器每次在代码中引用 g时，都要从内存中读取 g 的值
- 用sig_atomic_t声明标志
  - 在常见的处理程序设计中，处理程序会写全局标志来记录收到了信号。主程序周期性地读这个标志，响应信号，再清除该标志。整型数据类型 sig_atomic_t: 读和写保证会是原子的(不可中断的)。

信号不排队问题

- 当处理程序还在处理第一个信号时，第二个信号就传送并添加到了待处理信号集合里。此后不久，第三个信号到达了。因为已经有了一个待处理的 SIGCHLD，第三个 SIGCHLD 信号会被丢弃。一段时间之后，处理程序返回，内核注意到有一个待处理的 SIGCHLD 信号，就迫使父进程接收这个信号。父进程捕获这个信号，并第二次执行处理程序。在处理程序完成对第二个信号的处理之后，已经没有待处理的 SIGCHLD 信号了，而且也绝不会再有，因为第三个 SIGCHLD的所有信息都已经丢失了。
- 不可以用信号对其他进程中发生的事件计数



显式等待信号

- 错误方法：

```
while (!pid)
    pause();
```

- 正确但不好的方法：

```
while (!pid)
    ;
```

```
while (!pid)
    sleep(1);
```

- 正确方法：sigsuspend

```
Sigemptyset(&mask);
Sigaddset(&mask, SIGCHLD);

while (1) {
    Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
    if (Fork() == 0) /* Child */
        exit(0);

    /* Parent */
    pid = 0;
    Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

    /* Wait for SIGCHLD to be received (wasteful) */
    while (!pid)
        ;
```

sigsuspend 函数等价于下述代码的原子的(不可中断的)版本：

```
1       sigprocmask(SIG_SETMASK, &mask, &prev);
2       pause();
3       sigprocmask(SIG_SETMASK, &prev, NULL);
```

显式等待信号

- sigsuspend 暂时挂起进程的执行，等待一个信号的到来。它允许进程临时阻塞某些信号，并在收到指定信号后恢复执行。
- int sigsuspend(const sigset_t *mask)
- 保存当前信号屏蔽字
- 将进程的信号屏蔽字设置为 mask 所指向的信号集
- 等待一个信号的到来
- 在接收到信号后，恢复之前保存的信号屏蔽字，并返回

非本地跳转（Non-local Jump）

- 不通过传统的函数调用和返回的方式，在程序的执行中从一个函数或代码块直接跳转到另一个函数或代码块。
- setjmp和 longjmp
  - setjmp 函数在 env缓冲区中保存当前调用环境，以供后面的 longjmp 使用，并返回0。调用环境包括程序计数器、栈指针和通用目的寄存器。
  - longjmp 函数从 env缓冲区中恢复调用环境，然后触发一个从最近一次初始化 env的 setjmp 调用的返回。然后 setjmp 返回，并带有非零的返回值 retval
- 可能导致一些问题：破坏正常的函数调用和返回流程。使用非本地跳转时，需要小心确保程序状态的一致性，以及资源的正确释放。

C++中捕获和处理程序异常

```
6   try
7   {
8       // 可能抛出异常的代码
9       int divisor = 0;
10      if (divisor == 0)
11      {
12          // 抛出除以零的异常
13          throw std::runtime_error("Division by zero");
14      }
15      // 这里的代码不会执行，因为上面的异常会导致程序跳转到 catch 块
16      int result = 10 / divisor;
17      std::cout << "Result: " << result << std::endl;
18  }
19  catch (const std::exception &e)
20  {
21      // 捕获异常并处理
22      std::cerr << "Exception caught: " << e.what() << std::endl;
23  }
24  catch (...)
25  {
26      // 捕获其他类型的异常
27      std::cerr << "Unknown exception caught" << std::endl;
28  }
```