哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第三章 并发与同步：信号

陈芳林 副教授

哈尔滨工业大学（深圳）

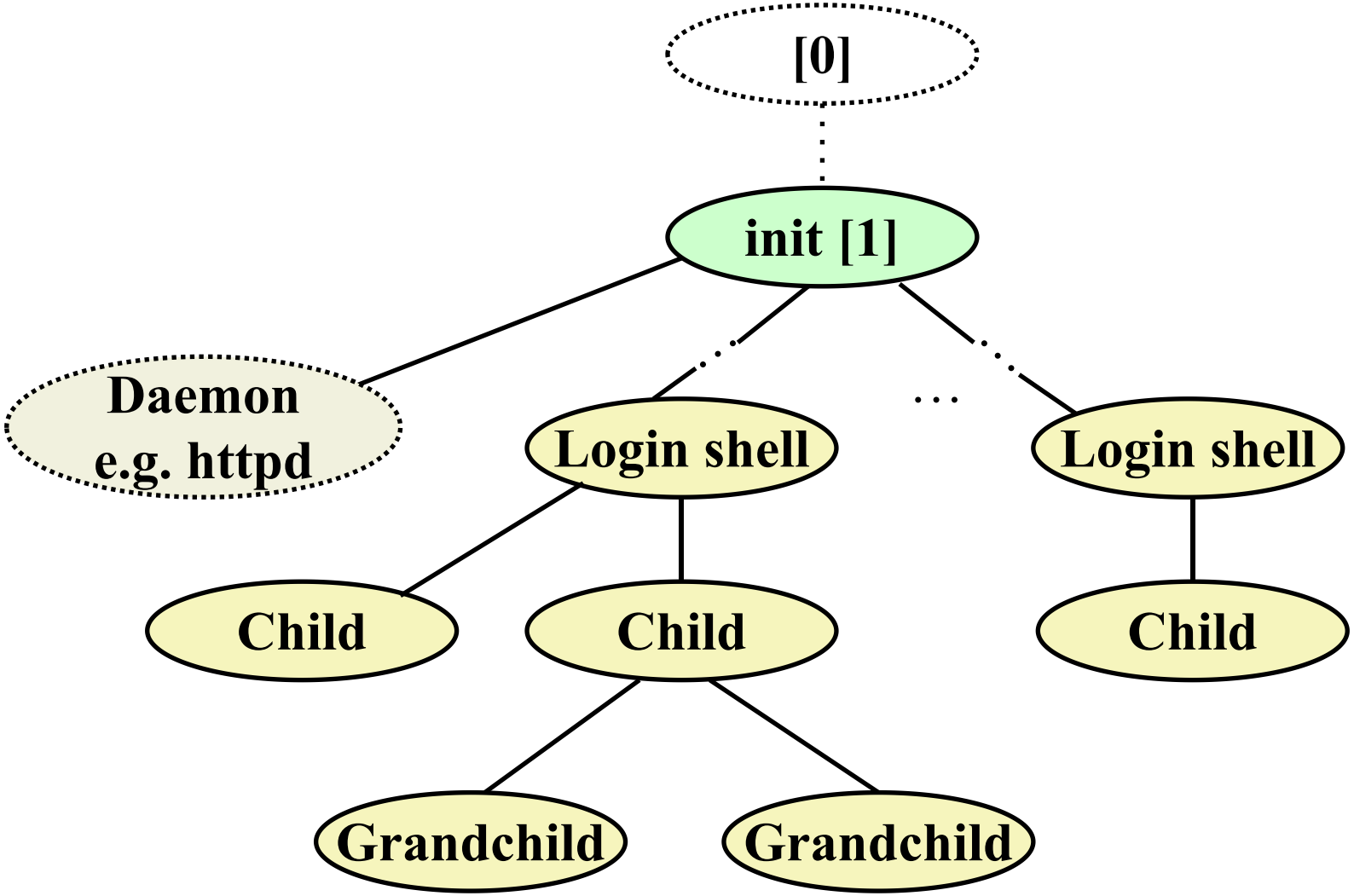
2024年秋

Email：chenfanglin@hit.edu.cn

本章主要内容

- 信号的介绍 （Introduction to Signals）
- 同步（Synchronization）
- 并发编程 （ Concurrent Programming ）

Linux 进程体系


[0]
init [1]
Daemon
e.g. httpd
Login shell
...
Login shell
Child
Child
Child
Grandchild
Grandchild

回顾：Shell 程序

- shell 是一个交互型应用级程序，代表用户运行其他程序

shellex.c

```
int main()
{
    char cmdline[MAXLINE]; /* command line */
    while(1){
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

shell执行一系列的读/求值步骤

读步骤读取用户的命令行，求值步骤解析命令，代表用户运行

回顾：Shell 程序--eval函数


```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        /* Parent waits for foreground job to terminate */父进程等待前台子进程结束
        if (!bg) {                   //fg或bg或&
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

shellex.c

简单shell例子的问题

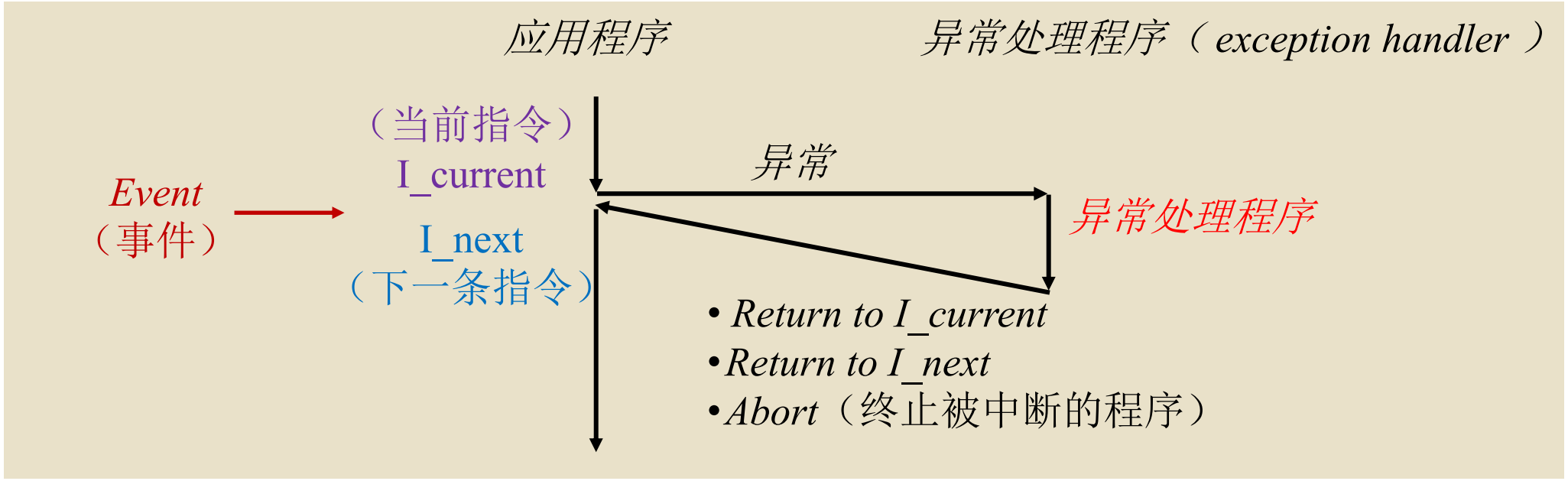
- 在这个例子中shell可以正确等待和回收前台作业
- 但是后台作业呢？
  - 后台进程终止时会成为僵尸进程
  - 永远不会被回收，因为shell（通常）不会终止
  - 将导致内存泄漏

结合进程知识点，怎么办？

- 解决办法：异常控制流
  - 在后台进程完成时内核将中断正常处理程序提醒我们
  - 在Unix里这种提醒机制叫作信号
  - Windows下称为消息

回顾：异常控制流

- 异常（Exception）是响应某些事件（即处理器状态更改）进而将控制权转移到OS内核（kernel）
  - 内核是OS的内存驻留部分
  - 事件举例: 零除指令, 算术溢出, 缺页, I/O请求, 用户输入中断指令, 进程调用kill等



回顾：异常控制流

- 存在于计算机系统的各个层次中
  - 低层次机制
    - ✓对系统时间作出响应并改变程序的控制流（会导致系统状态的改变）
    - ✓通过硬件和操作系统软件的组合来实现
  - 高层次机制
    - ✓进程上下文切换（Process context switch）：通过操作系统软件和硬件时钟定时器来实现
    - ✓信号（Signals）：通过操作系统软件来实现(应用层)
    - ✓全局跳转（Nonlocal jumps）: setjmp() and longjmp()：通过C语言运行时库实现

信号（Signal）

- 信号是内核发出的一个消息，用来通知进程：系统中发生了某种类型的事件。
  - 信号从内核发往进程（有时信号的发送是由另一个进程发起的）
  - 每一种信号都用一个整数ID来表示（例如，Linux支持的ID为1-30）
  - 通常情况下只有信号ID会被发送给进程
  - 使用信号的两个主要目的为：
    - ✓ 1) 通知进程：某种特殊的事件发生了
    - ✓ 2) 迫使进程执行信号处理程序（signal handler）

ID	Name	默认操作	对应的事件
2	SIGINT	终止 Terminate	用户按下 ctrl-c
9	SIGKILL	终止 Terminate	杀死程序（Kill program，不可否决或忽略）
11	SIGSEGV	终止 Terminate	无效的内存引用
14	SIGALRM	终止 Terminate	计时器信号
17	SIGCHLD	忽略 Ignore	子进程终止运行

信号（Signal）

- ctrl-z 发送 SIGTSTP 信号给前台进程组中的所有进程，常用于挂起一个进程。
- 哪一个信号与ctrl-z 相反？
  - 18　　SIGCONT
- ctrl-d 不是发送信号，而是表示一个特殊的二进制值，表示 EOF。
- ctrl-\ 发送 SIGQUIT 信号给前台进程组中的所有进程，终止前台进程并生成 core 文件。
- SIGPIPE表明了管道的什么特性？

序号	名称	默认行为	相应事件
1	SIGHUP	终止	终端线挂断
2	SIGINT	终止	来自键盘的中断
3	SIGQUIT	终止	来自键盘的退出
4	SIGILL	终止	非法指令
5	SIGTRAP	终止并转储内存①	跟踪陷阱
6	SIGABRT	终止并转储内存①	来自 abort 函数的终止信号
7	SIGBUS	终止	总线错误
8	SIGFPE	终止并转储内存①	浮点异常
9	SIGKILL	终止②	杀死程序
10	SIGUSR1	终止	用户定义的信号 1
11	SIGSEGV	终止并转储内存①	无效的内存引用（段故障）
12	SIGUSR2	终止	用户定义的信号 2
13	SIGPIPE	终止	向一个没有读用户的管道做写操作
14	SIGALRM	终止	来自 alarm 函数的定时器信号
15	SIGTERM	终止	软件终止信号
16	SIGSTKFLT	终止	协处理器上的栈故障
17	SIGCHLD	忽略	一个子进程停止或者终止
18	SIGCONT	忽略	继续进程如果该进程停止
19	SIGSTOP	停止直到下一个 SIGCONT②	不是来自终端的停止信号
20	SIGTSTP	停止直到下一个 SIGCONT	来自终端的停止信号
21	SIGTTIN	停止直到下一个 SIGCONT	后台进程从终端读
22	SIGTTOU	停止直到下一个 SIGCONT	后台进程向终端写
23	SIGURG	忽略	套接字上的紧急情况
24	SIGXCPU	终止	CPU 时间限制超出
25	SIGXFSZ	终止	文件大小限制超出
26	SIGVTALRM	终止	虚拟定时器期满
27	SIGPROF	终止	剖析定时器期满
28	SIGWINCH	忽略	窗口大小变化
29	SIGIO	终止	在某个描述符上可执行 I/O 操作
30	SIGPWR	终止	电源故障

信号概念1：发送信号 (Sending a Signal)

- 内核发送信号到目的进程：更改目的进程上下文中的某些状态。
- 内核发送信号的两个原因：
  - 内核检测到了一个系统事件的发生，例如除零（SIGFPE）和子进程终止(SIGCHLD)
  - 进程调用了 kill 来显式地请求内核向目标进程发送信号
    - ✓一个进程可以发送信号给它自己
- 例如：kill -9 <ps>；kill -2 <ps> #ctrl-c

MacOS

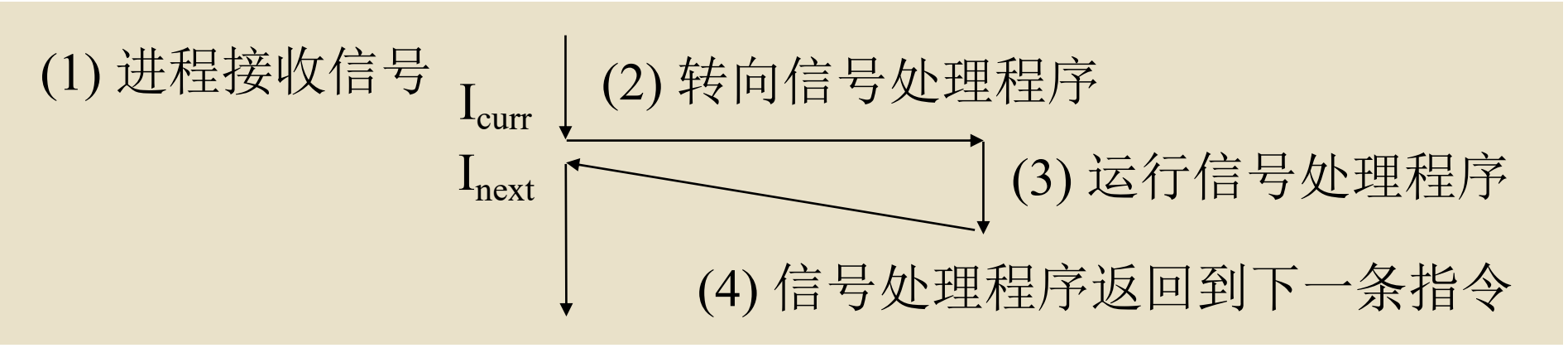
```
→  ~ kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS
 PIPE ALRM TERM URG STOP TSTP CONT CHLD TTIN TTOU IO
 XCPU XFSZ VTALRM PROF WINCH INFO USR1 USR2
```

Ubuntu

```
chenjunjie@icrc-k80:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

信号概念2：接收信号 (Receiving a Signal)

- 当目的进程被内核强制以某种方式对信号的发送做出反应时，它就接收了这个信号
- 一些可能的反应：
  - 忽略（ignore）该信号（什么也不做）
  - 终止（terminate）该进程（核心转储 core dump）
  - 通过执行一个用户级的信号处理程序（signal handler）来接收（catch）该信号
    - ✓类似于在响应异步中断时调用的硬件异常处理程序：



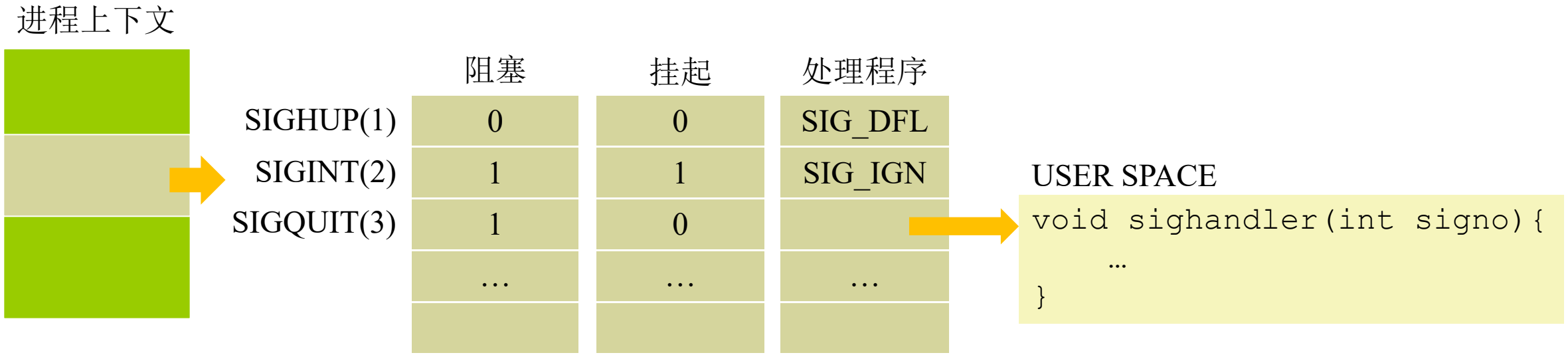
信号概念3：待处理(pending)和阻塞(block)信号

- 一个已经被发送出去但没有被接收的信号就是待处理信号
  - 对一个进程而言，每个类型的信号最多只能有一个待处理信号
  - 注意：信号不会排队
    - ✓假设一个进程有一个ID为*k*的待处理信号，那么后面被发送到该进程的信号*k*会被丢弃，而不是排队等待
  - 一个待处理信号最多只能被接收一次
- 进程可以选择性地阻塞某些信号的接收
  - 被阻塞的信号也可以传输，但在它被解除阻塞之前，该信号不能被接收

挂起/阻塞位 (Pending/Blocked Bits)

- 内核会在每个进程的上下文中维护挂起和阻塞位向量 (pending and blocked bit vectors)
  - 挂起位向量（待处理信号的集合）：
    - ✓当信号*k*被传输时，内核设置挂起向量的第*k*位
    - ✓当信号*k*被接收时，内核将挂起向量第*k*位清除
  - 阻塞位向量（阻塞信号的集合）：
    - ✓可以用sigprocmask函数设置和清除
    - ✓也被称为信号屏蔽掩码 (signal mask)

挂起/阻塞位 (Pending/Blocked Bits)



- SIGHUP信号未阻塞也未产生过，当它接收时执行默认处理程序。
- SIGINT信号产生过，但正在被阻塞，暂时不能被接收。虽然它的处理程序是忽略，但在没有解除阻塞之前不能忽略这个信号，因为进程仍有机会改变处理程序之后再解除阻塞。
- SIGQUIT信号未产生过，一旦产生SIGQUIT型号将被阻塞，它的处理程序是用户自定义函数sighandler。

信号的阻塞(Blocking)与解除阻塞(Unblocking)

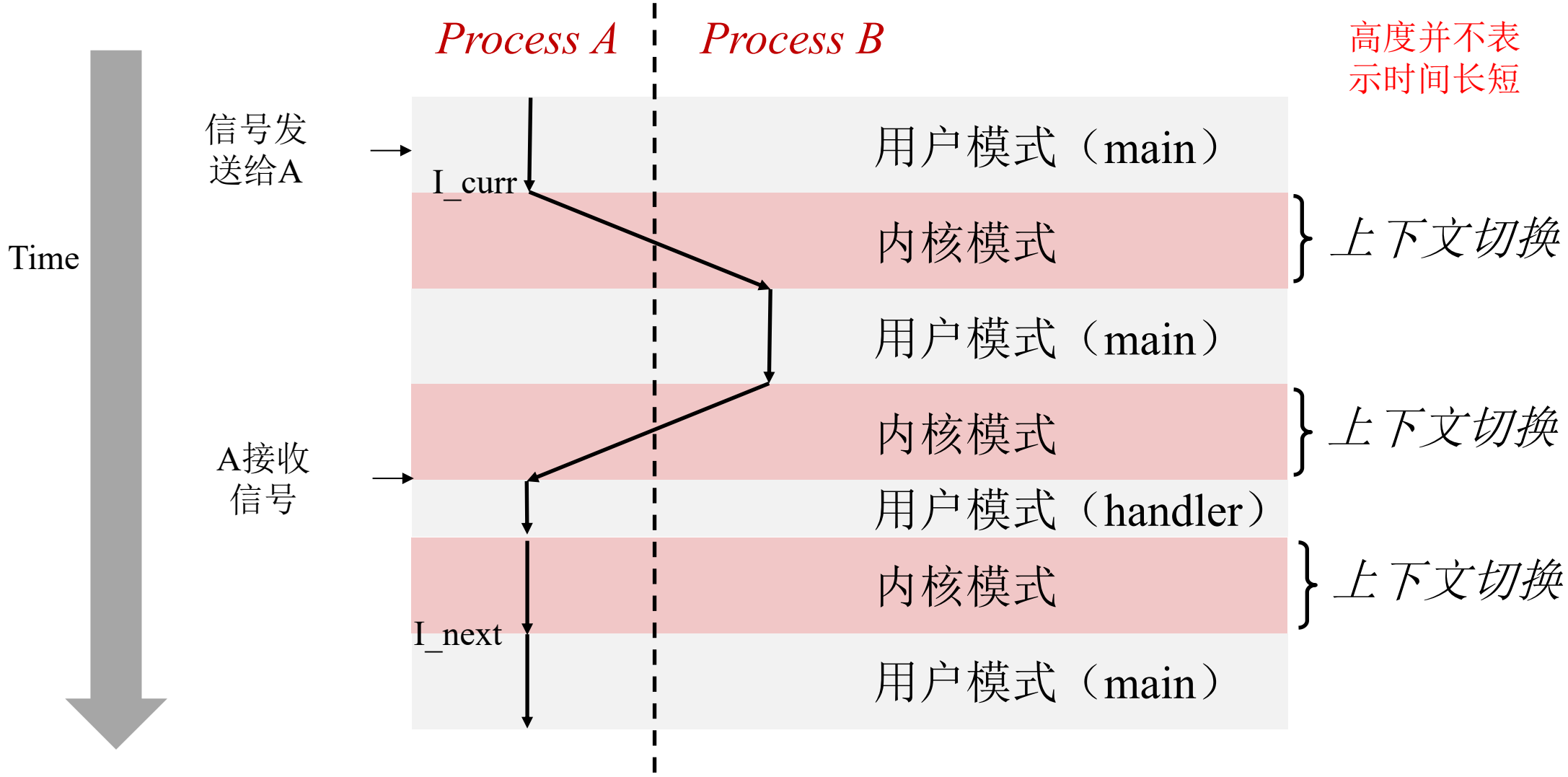
- 隐式阻塞机制
  - 内核默认阻塞与当前正在处理信号类型相同的待处理信号
  - 如：一个SIGINT 信号处理程序不能被另一个 SIGINT信号中断（此时另一个SIGINT信号被阻塞）
- 显式阻塞和解除阻塞机制
  - sigprocmask 函数及其辅助函数可以明确地阻塞/解除阻塞选定的信号：SIG_BLOCK/UNBLOCK/SET_MASK
- 辅助函数
  - sigemptyset – 初始化set为空集合
  - sigfillset – 把每个信号都添加到set中
  - sigaddset – 把指定的信号signum添加到set
  - sigdelset – 从set中删除指定的信号signum

接收信号 (Receiving Signals)

- 假设内核正在从异常处理程序返回，并准备将控制权传递给该进程时
  - 内核检查 pnb = pending & ~blocked
    - ✓进程的未被阻塞的待处理信号的集合
  - If (pnb == 0) 如果集合为空
    - ✓ 将控制传递到进程的逻辑控制流中的下一条指令
  - else 不为空
    - ✓选择集合pnb中最小的非零位k，强制进程处理信号k（清0）
      - 收到信号会触发进程采取某种行为
    - ✓对所有的非零k重复
    - ✓控制传递到进程的逻辑控制流中的下一条指令

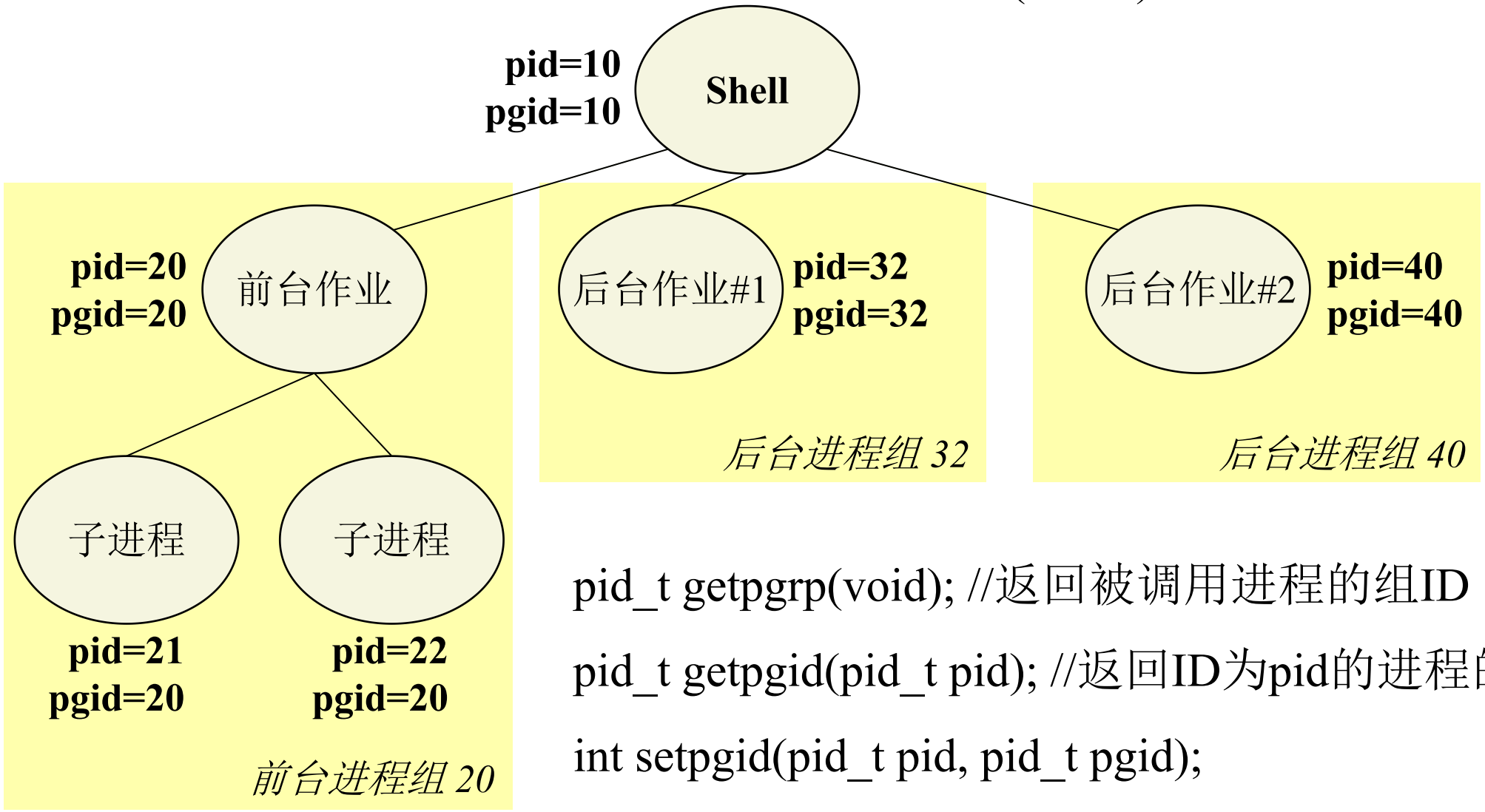
接收信号 (Receiving Signals)

- 假设内核正在从异常处理程序返回，并准备将控制权传递给该进程时



发送信号：进程组 (Process Groups)

- 每个进程只属于一个进程组，用一个正整数组ID表示 (PGID)



pid_t getpgrp(void); //返回被调用进程的组ID

pid_t getpgid(pid_t pid); //返回ID为pid的进程的组ID

int setpgid(pid_t pid, pid_t pgid);

发送信号：进程组 (Process Groups)

Ubuntu

```
chenjunjie@icrc-k80:~$ ps ejH
   PPID     PID    PGID     SID TTY        TPGID STAT   UID   TIME COMMAND
   8755    8764    8764    8764 pts/0     8904 Ss    1001   0:00 -bash LC_TERMI
   8764    8904    8904    8764 pts/0     8904 R+    1001   0:00 ps ejH SHELL=/
```

MacOS

```
→  ~ ps axj
USER               PID  PPID  PGID    SESS JOBC STAT   TT         TIME COMMAND
root                 1     0     1       0    0 Ss     ??    1:23.13 /sbin/launc
root                61     1    61       0    0 Ss     ??    0:03.06 /usr/sbin/s
root                62     1    62       0    0 Ss     ??    0:09.22 /usr/libexe
root                65     1    65       0    0 Ss     ??    0:00.91 /System/Lib
root                66     1    66       0    0 Ss     ??    0:29.99 /System/Lib
root                67     1    67       0    0 Ss     ??    0:19.05 /System/Lib
root                70     1    70       0    0 Ss     ??    1:03.71 /usr/sbin/s
root                72     1    72       0    0 Ss     ??    0:05.93 /usr/libexe
root                73     1    73       0    0 Ss     ??    1:04.97 /Applicatio
root                74     1    74       0    0 Ss     ??    0:00.01 endpointsec
root                75     1    75       0    0 Ss     ??    0:19.97 /System/Lib
```

在Ubuntu上试试 ps axj

发送信号：使用/bin/kill程序

- /bin/kill程序可以向另外的进程或进程组发送任意的信号
- Examples
  - /bin/kill –9 24818

    发送信号9（SIGKILL）给进程 24818
  - /bin/kill –9 –24817

    发送信号SIGKILL给进程组24817中的每个进程
  - 负的PID会导致信号被发送到进程组PID中的每个进程

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

发送信号用 kill 函数

- 进程通过调用kill函数向其他进程（包括它自己）发送信号
- int kill(pid_t pid, int sig)；
  - pid>0：发送信号sig给进程pid
  - pid=0：发送信号sig给调用进程的进程组内的所有进程（包括其自身）
  - pid <0：发送信号sig给进程组$|pid|$（pid的绝对值）内的所有进程
    - ✓特殊情况：pid = -1，发送给当前用户拥有权限的所有进程（ init 进程除外）
- /bin/kill 程序 和 kill 命令 的区别

```
chenjunjie@icrc-k80:~$ /usr/bin/kill -9 0
Killed
chenjunjie@icrc-k80:~$ kill -9 0
Connection to 10.249.144.190 closed.
```

发送信号用 kill 函数示例


```
1. int main()
2. {
3.     pid_t pid;
4.
5.     /* Child sleeps until SIGKILL signal received, then dies */
6.     if ((pid = fork()) == 0) {
7.         pause();  /* Wait for a signal to arrive */
8.         printf("control should never reach here!\n");
9.         exit(0);
10.    }
11.
12.    /* Parent sends a SIGKILL signal to a child */
13.    Kill(pid, SIGKILL);
14.    exit(0);
15.}
```

用 kill 函数发送信号


```
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1) ;
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    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));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

- 如果pid大于零，那么kill函数发送信号号码sig给进程pid。
- 如果pid等于零，那么kill发送信号sig给调用进程所在进程组中的每个进程，包括调用进程自己。
- 如果pid小于零，kill发送信号sig给进程组|pid|（pid的绝对值）中的每个进程。

forks.c

用 kill 函数发送信号


```
void fork12()
{
    pid_t pid[N]; //N = 5
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1) ;
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally
    }
}
```

- 如果pid大于零，那么kill函数发送信号号码sig给进程pid。
- 如果pid等于零，那么kill发送信号sig给调用进程所在进程组中的每个进程，包括调用进程自己。
- 如果pid小于零，kill发送信号sig给进程组|pid|（pid的绝对值）中的每个进程。

```
Killing process 1265019
Killing process 1265020
Killing process 1265021
Killing process 1265022
Killing process 1265023
Child 1265019 terminated abnormally
Child 1265020 terminated abnormally
Child 1265021 terminated abnormally
Child 1265022 terminated abnormally
Child 1265023 terminated abnormally
```

信号的默认操作（Default Actions）

- 每一种信号都有一个预设的默认操作，包括：
  - 进程终止（process terminates）
  - 进程终止并转储内存（process terminates and dumps core）
  - 进程停止（挂起）直到被SIGCONT信号唤起
  - 进程忽略该信号

设置信号处理程序（Installing Signal Handlers）

- 进程可以通过signal函数修改一个信号（signum）的默认操作
  - sighandler_t signal(int signum, sighandler_t handler)
  - 其中SIGSTOP和SIGKILL信号除外：这两个信号的默认操作不可被修改
- handler取不同值时:
  - SIG_IGN：忽略signum信号
  - SIG_DFL：将signum信号重设为默认操作
  - 否则，handler就是用户定义的信号处理程序的函数地址
    - ✓当进程接收到signum信号时调用该程序
    - ✓当处理程序执行return时，控制会传递到控制流中被信号接收所中断的指令处
    - ✓将处理程序的地址传递到signal函数从而改变默认行为，这就叫作设置信号处理程序

信号处理实例（Signal Handling Example）


```
1. void sigint_handler(int sig) /* SIGINT handler */
2. {
3.     printf("So you think you can stop the bomb with ctrl-c, do you?\n");
4.     sleep(2);
5.     printf("Well...");
6.     fflush(stdout);
7.     sleep(1);
8.     printf("OK. :-)\n");
9.     exit(0);
10.}
11.int main()
12.{
13.    /* Install the SIGINT handler */
14.    if (signal(SIGINT, sigint_handler) == SIG_ERR)
15.        unix_error("signal error");
16.
17.    /* Wait for the receipt of a signal */
18.    pause();
19.    return 0;
20.}
```

按一次ctrl+c？

```
chenjunjie@icrc-k80:~/OS_course$ ./sigint
^CSo you think you can stop the bomb with ctrl-c, do you?
Well...OK. :-)
```

逛按ctrl+c，会怎么样？

```
chenjunjie@icrc-k80:~/OS_course$ ./sigint
^CSo you think you can stop the bomb with ctrl-c, do you?
^C^C^C^C^C^C^C^C^C^CWell...^C^C^C^C^COK. :-)
```

sigint.c

信号处理实例（Signal Handling Example）


```
1. void sigint_handler(int sig) /* SIGINT handler */
2. {
3.     printf("So you think you can stop the bomb with ctrl-c, do you?\n");
4.     sleep(2);
5.     printf("Well...");
6.     fflush(stdout);
7.     sleep(1);
8.     printf("OK. :-)\n");
9.     exit(0);
10.}
11.int main()
12.{
13.    /* Install the SIGINT handler */
14.    if (signal(SIGINT, sigint_handler) == SIG_ERR)
15.        unix_error("signal error");
16.
17.    /* Wait for the receipt of a signal */
18.    pause();
19.    return 0;
20.}
```

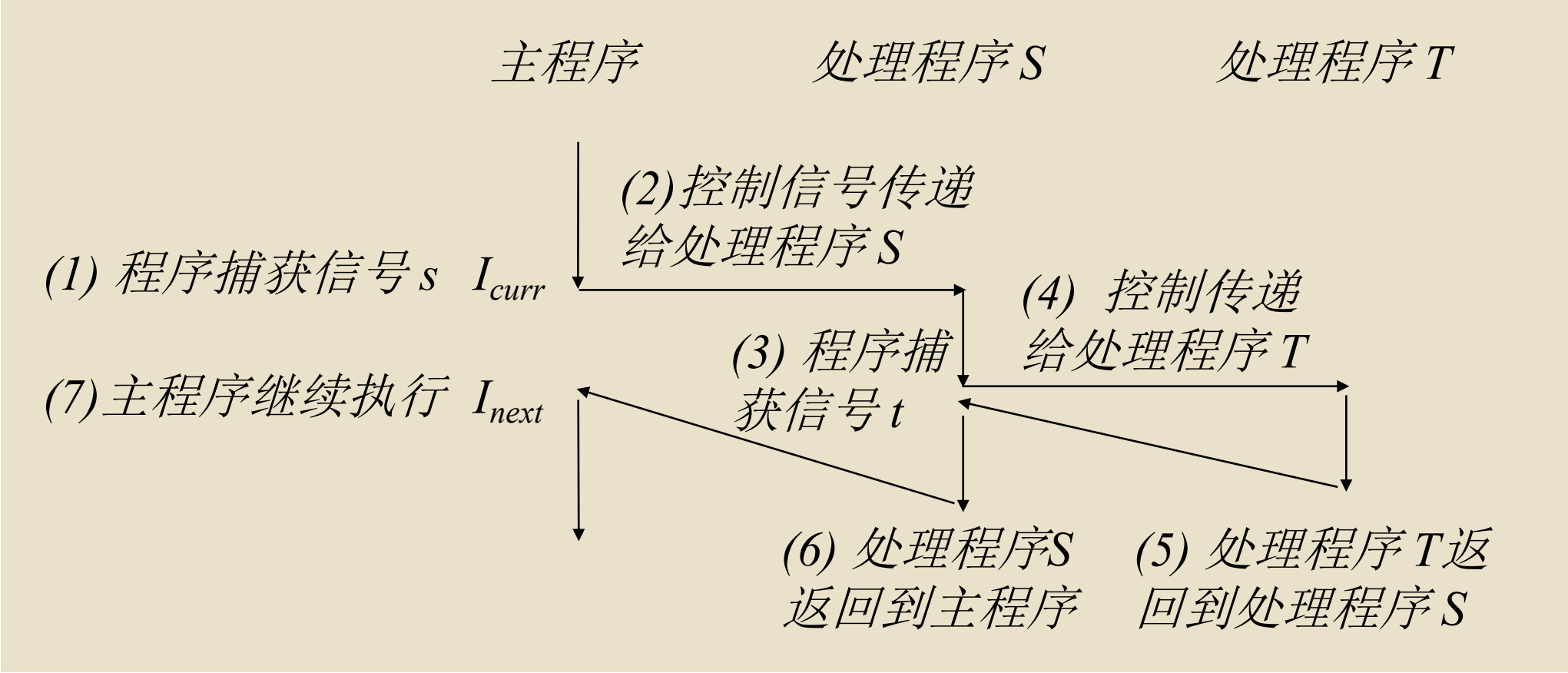
按一次ctrl+c，再按ctrl+z？

```
chenjunjie@icrc-k80:~/OS_course$ ./sigint
^CSo you think you can stop the bomb with ctrl-c, do you?
^Z
[1]+  Stopped                 ./sigint
```

sigint.c

嵌套的信号处理程序

- 信号处理程序可以被其他信号处理程序中断



Shell改进实现

shell2.c

```
1. void handler(int signum)
2. {
3.     int status;
4.     while(waitpid(-1, &status, WNOHANG) > 0);
5. }
6. int main()
7. {
8.     char cmdline[MAXLINE]; /* Command line */
9.     signal(SIGCHLD, handler);
10.    while (1) {
11.       /* Read */
12.       printf("> ");
13.       fgets(cmdline, MAXLINE, stdin);
14.       if (feof(stdin))
15.          exit(0);
16.       eval(cmdline); /* Evaluate */
17.    }
18.    return 0;
19.}
```

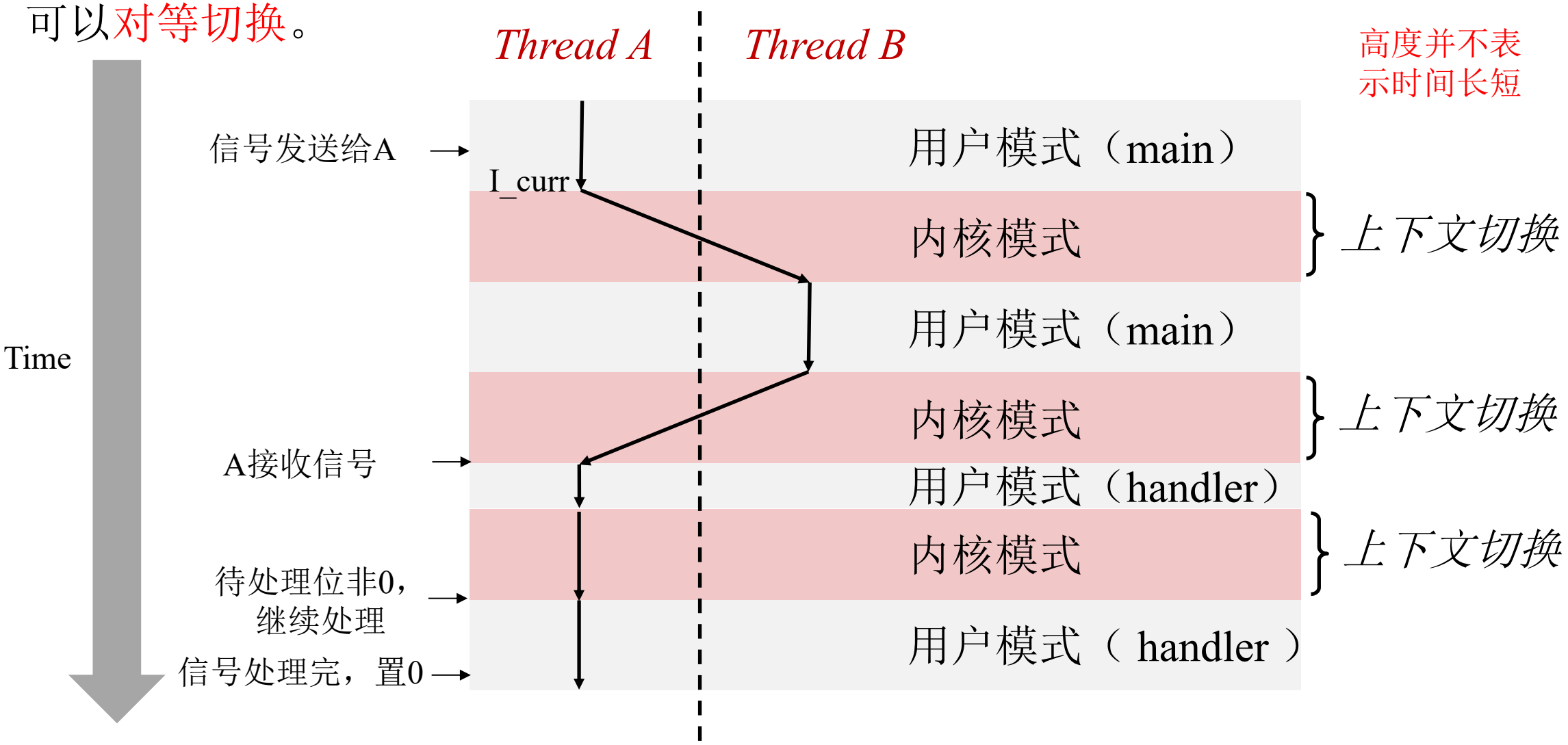
回收任意子进程　　非阻塞回收（轮询）

信号的一些注意事项

- 信号不会排队等待
- 默认情况下，内核会阻塞所有与当前正处理信号相同的待处理信号
- 信号处理是Linux系统层编程最棘手的问题之一：
  - 安全的信号处理
  - 正确的信号处理
  - 可移植的信号处理

信号的一些注意事项

- 信号处理程序与线程执行区别：内核强制以某种方式对待处理信号做出反应，而线程可以对等切换。



安全的信号处理

- 信号处理程序和主程序或其他信号处理程序是并发执行的，因此会带来很多问题。
  - 例如：如果处理程序和主程序并发访问同样的全局数据结构，那么结果就是未知的甚至是致命的

- 一些保守的编写处理程序的原则：
  - G0: 处理程序尽可能地简单
  - 例如： 处理程序只是简单地设置全局标志并且立即返回；所有与接受信号有关的工作都交给主程序来执行，它只周期性地检查（并重置）这个标志。

安全的信号处理

- G1: 在处理程序中只调用异步信号安全的函数
  - 异步信号安全（简称安全）：可重入的（例如只访问局部变量，之后会介绍）或者不能被信号处理程序中断。
  - 常用的安全函数：_exit, fork, accept, kill, execve, signal, waitpid, write,…（详情可见CSAPP，P534，Figure 8-33)
  - 注意：许多常见的函数如printf、sprintf、malloc和exit都不安全
  - 信号处理程序中产生输出唯一安全地方法就是write函数。可以使用下面的安全的I/O（SIO）包，在信号处理程序中打印简单的消息：

```
1.#include “csapp.h”
2.ssize_t sio_put1(long v);
3.ssize_t sio_puts(char s[]);
4.//若成功，则返回输出的字节数，如果出错，返回-1
5.void sio_error(char s[]);    //返回空
```

安全的信号处理

- 下面给出SIGINT的一个安全版本：

```
1.#include “csapp.h”
2.void sigint_handler(int sig)  /* Safe SIGINT handler */
3.{
4.    sio_puts(“Caught SIGINT!\n”);  /* Safe output */
5.    _exit(0);                      /* Safe exit */
6.}
```

安全的信号处理

- G2：保存和恢复errono
  - 许多异步安全的函数都会在出错返回时设置errno，而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。
  - 解决方法就是在进入处理程序时把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，若处理程序调用_exit终止该进程，那么就不需要这样做了）
- G3：阻塞所有的信号，保护对共享全局数据结构的访问
  - 如果处理程序和主程序或其他处理程序共享一个全局数据结构，那么在访问（读或写）该数据结构时，处理程序和主程序应该暂时阻塞所有的信号。

安全的信号处理


```
1.void handler(int sig){
2.     //job为一个全局变量
3.
4.     int olderrno = errno;
5.     sigset_t mask_all, pre_all;
6.     pid_t pif;
7.     sigfillset(&mask_all); //设置mask_all为包含所有信号的集合
8.     while ((pid = waitpid(-1, NULL, 0) > 0) {
9.          //将所有信号都添加到blocked中
10.         sigprocmask(SIG_BLOCK, &mask_all, &pre_all);
11.         deletejob(pid); //删除job列表的pid元素
12.         sigprocmask(SIG_SETMASK, &prev_all, NULL);//恢复blocked原来状态
13.    }
14.    if (errno != ECHILD)
15.         sio_error (“waitpid error”);
16.         errno = olderrno;
17.}
```

安全的信号处理

- G4: 用volatile声明全局变量
  - 考虑一个处理程序和一个main函数，它们共享一个全局变量g。处理程序更新g, main周期性地读g。
  - 对于一个优化缓存器而言，main中g的值看上去从来没有变化过，因此使用缓存在寄存器中g的副本来满足对g的每次使用是很安全的。如果这样，main函数可能永远都无法看到处理程序更新过的值。
  - 可以用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。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。

```
volatile int g;
```

安全的信号处理

- G5: 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。
  - 对于通过这种方式来共享的标志，C提供一种整形数据类型sig_atomic_t，对它的读和写保证会是原子的（不可中断）。

```
volatile sig_atomic_t flag;
```

  - 注意，这里对原子性的保证只适用于单个的读和写，不适用于像flag++或flag=flag+10这样的更新，它们可能需要多条指令。

正确的信号处理

- 未处理的信号是不排队的，因为pending位向量中每种类型的信号只对应有一位，所以每种类型最多只能有一个未处理的信号。
- 如果两个类型为*k*的信号发送给一个目的进程，而目的进程当前正在执行信号*k*的处理程序，所以第一个信号*k*被阻塞了，第二个信号就简单地被丢弃了；它不会排队。
- 关键思想：如果存在一个未处理地信号就表明至少有一个信号到达了。
- 下面是一个简单的应用，说明如何影响正确性：

错误的信号处理示例


```
1. int ccount = 0;
2. void child_handler(int sig) {
3.     int olderrno = errno;
4.     pid_t pid;
5.     if ((pid = wait(NULL)) < 0)
6.         sio_error("wait error");
7.     ccount--;
8.     sio_puts("Handler reaped child
   ");
9.     sio_putl((long)pid);
10.    sio_puts(" \n");
11.    sleep(1);
12.    errno = olderrno;
13.}
```

wrong_handler.c

```
14.void fork12() {
15.    pid_t pid[N];
16.    int i;
17.    ccount = N;
18.    signal(SIGCHLD, child_handler);
19.
20.    for (i = 0; i < N; i++) {
21.        if ((pid[i] = fork()) == 0){
22.            sleep(1);
23.            exit(0);
24.        }
25.    }
26.    /* Parent spins */
27.    while (ccount > 0) ;
```

```
Hello from child 57158
Hello from child 57159
Hello from child 57160
Handler reaped child 57160
Handler reaped child 57159
```

出现僵尸进程！！

正确的信号处理示例

- 必须回收所有终止的子进程：一个信号可能多个终止
  - 将 wait放入一个循环来回收所有终止的子进程

```
1. void child_handler2(int sig)
2. {
3.     int olderrno = errno;
4.     pid_t pid;
5.     while ((pid = wait(NULL)) > 0) {
6.         ccount--;
7.         sio_puts("Handler reaped child ");
8.         sio_putl((long)pid);
9.         sio_puts(" \n");
10.    }
11.    if (errno != ECHILD)
12.        sio_error("wait error");
13.    errno = olderrno;
14.}
```

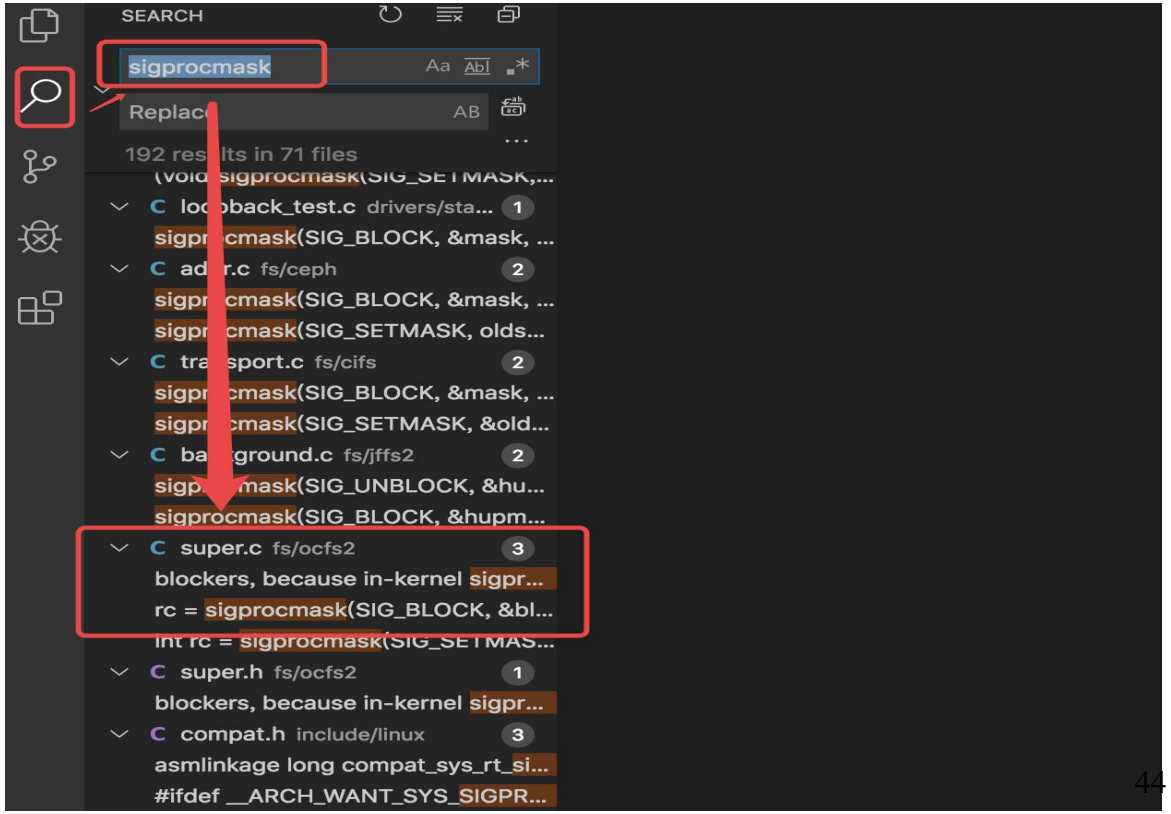
```
Hello from child 57150
Hello from child 57151
Hello from child 57152
Handler reaped child 57150
Handler reaped child 57152
Handler reaped child 57151
```

拓展1： VS Code查阅Linux内核代码

- Linux 内核源码地址： https://github.com/torvalds/linux.git
- 推荐使用工具 ： VS Code https://code.visualstudio.com/ Clion https://www.jetbrains.com/clion/
- Linux内核代码风格： https://www.kernel.org/doc/html/v4.14/translations/zh_CN/coding-style.html#

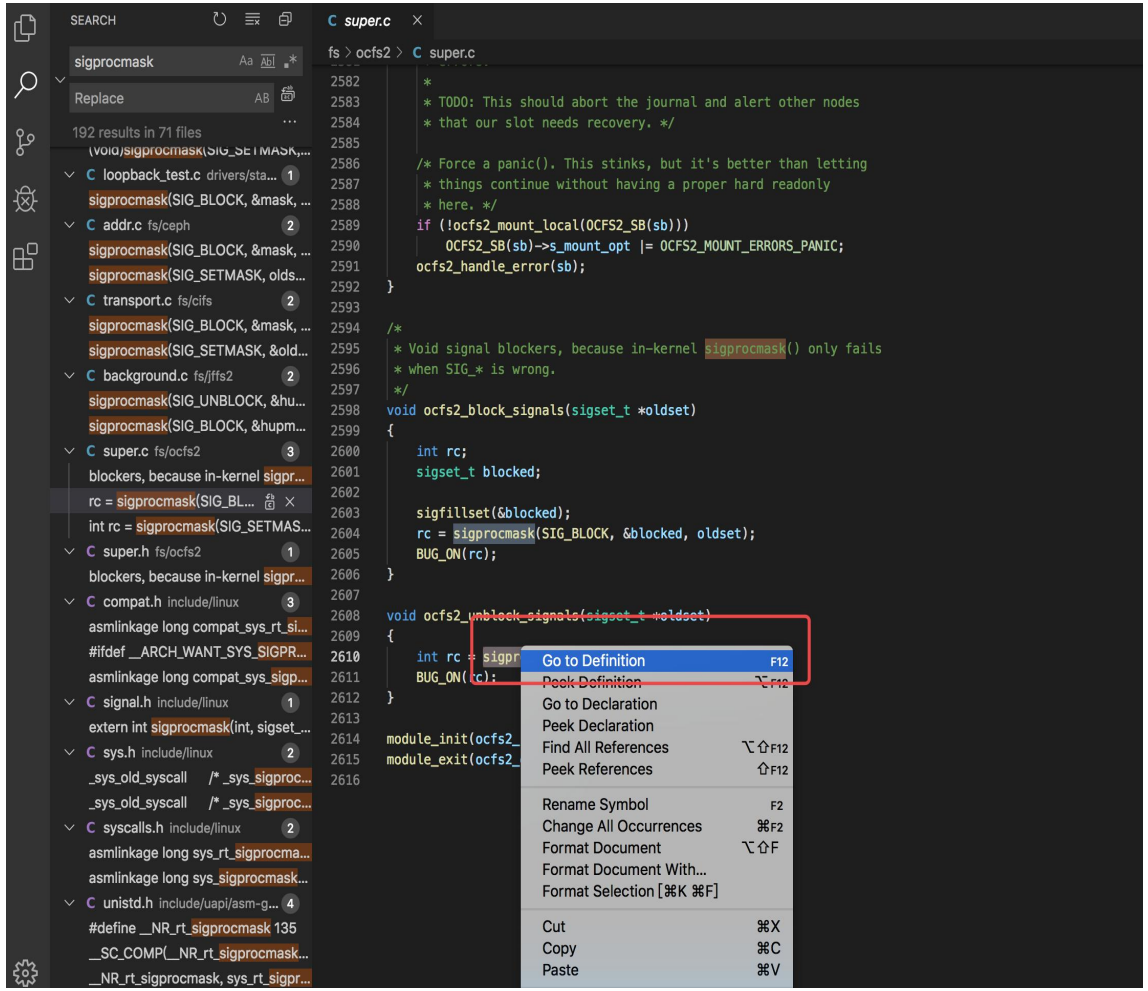
1. 根据sigprocmask函数，在vscode打开的Linux内核源代码中查找信号相关的结构和代码。

2. 在搜索栏中查找关键字，寻找一个sigprocmask 函数（声明、调用、定义）。

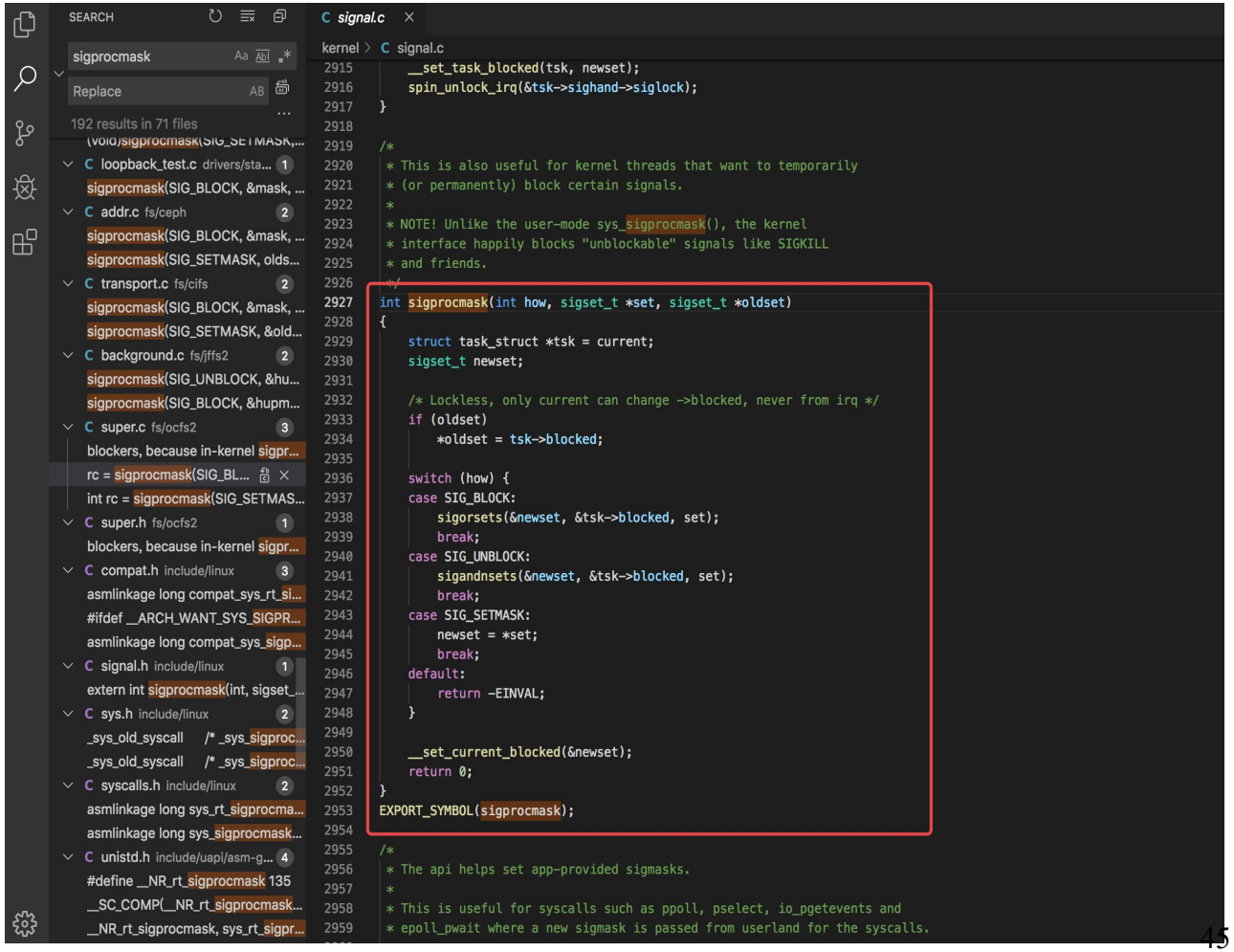


拓展1： VS Code查阅Linux内核代码

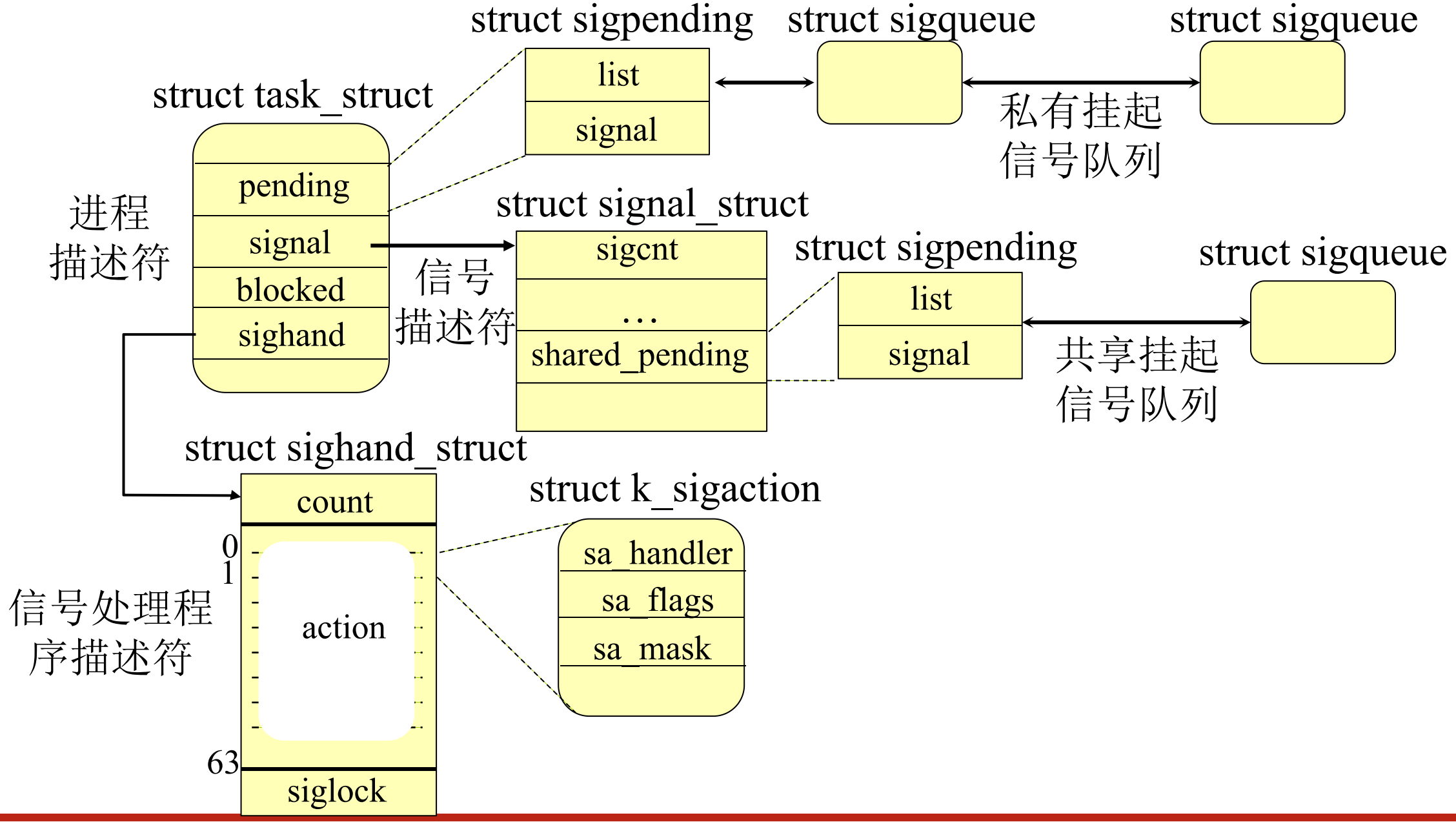
3. 点开之后，选中关键字，右键选择Go to Definition（记得先安装C/C++插件）。



4. 跳转到内核中sigprocmask 源码处，发现它的实现位于kernel/signal.c文件中。

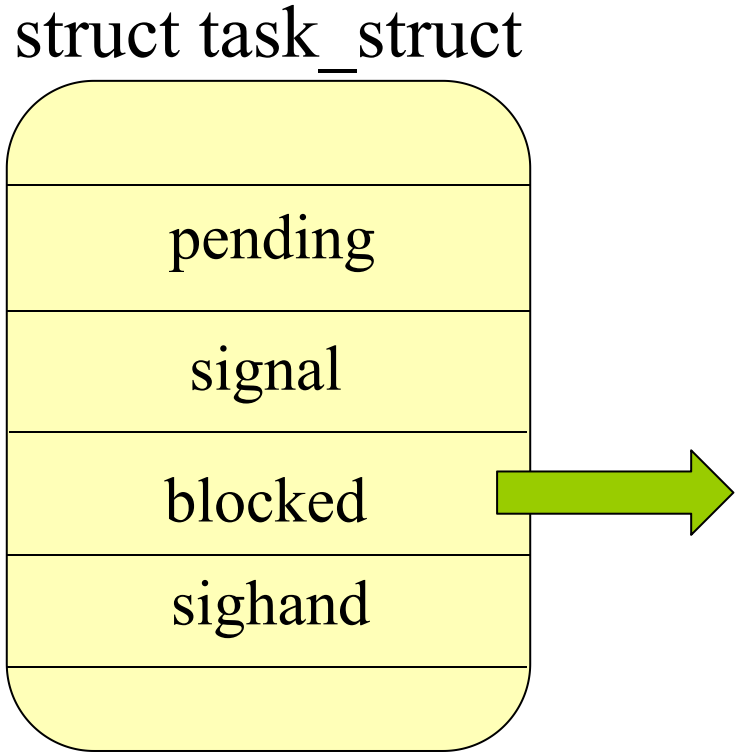


拓展2：Signal内核数据结构



拓展2：Signal内核数据结构--blocked

- blocked类型为sigset_t， 表示被阻塞的信号

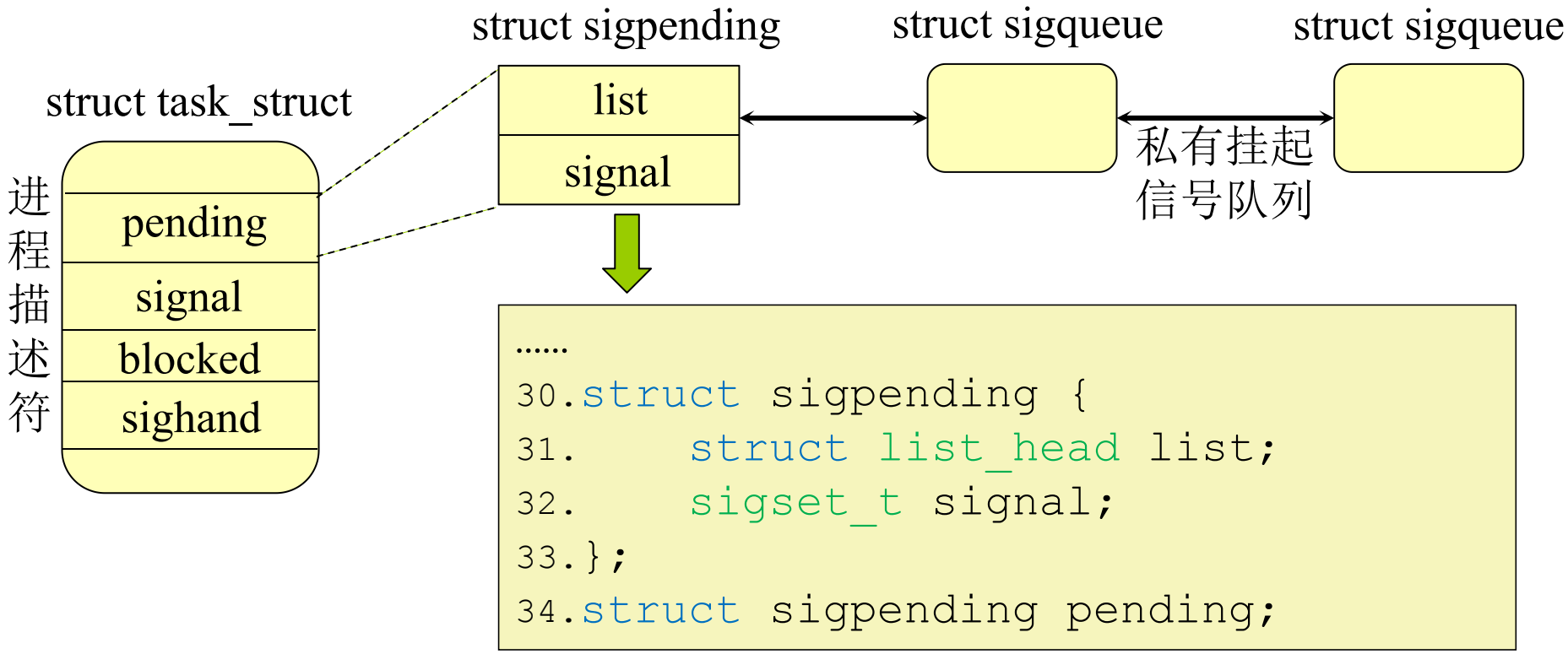



```
…
11.#define _NSIG_WORDS (_NSIG / _NSIG_BPW)
12.
13.typedef struct {
14.     unsigned long sig[_NSIG_WORDS];
15.} sigset_t;
…
sigset_t blocked;
```

arch/x86/include/asm/signal.h

拓展2：Signal内核数据结构--Pending

- pending指向sigpending结构体，表示待处理信息集合
- sigpending由一个双向列表list和一个sigset_t signal组成



include/linux/signal_types.h

习题

- 1.子程序运行结束会向父进程发送_____信号。

答案：SIGCHLD

- 2. 异步信号安全的函数要么是可重入的（如只访问局部变量）要么不能被信号处理程序中断，包括I/O函数（　）

A. printf　　B. sprintf　　C. write　　D. malloc

答案：C

Hope you enjoyed the OS course!