异常控制流

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. **Bryant and** David R. O’Hallaron

异常控制流存在系统每个层次/ECF Exists at All Levels of a System

- **异常/Exceptions**
  - 硬件和系统内核
  - Hardware and operating system kernel software
- **进程切换/Process Context Switch**
  - 硬件时钟和内核软件
  - Hardware timer and kernel software

} **Previous Lecture**

- **信号/Signals**
  - 内核软件和应用
  - Kernel software and application software

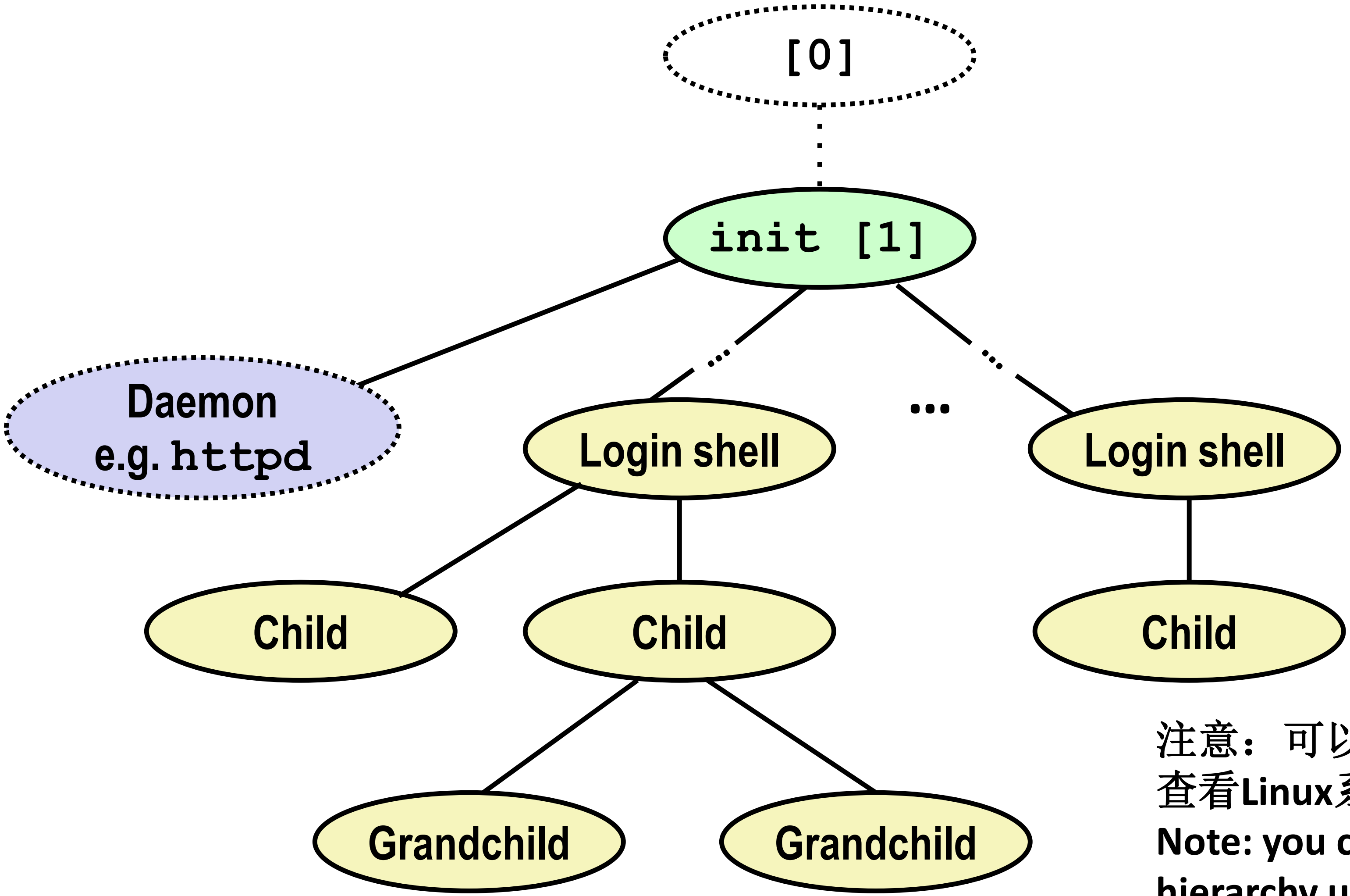
} **This Lecture**

- **非局部跳转/Nonlocal jumps**
  - 应用代码 Application code

} **Textbook and supplemental slides**

目录

Linux进程树/Linux Process Hierarchy



注意：可以用pstree命令查看Linux系统的进程树/
Note: you can view the hierarchy using the Linux `pstree` command

Shell程序/Shell Programs

- Shell是按照用户要求运行程序的应用程序/A *shell* is an application program that runs programs on behalf of the user.
  - `sh` 最早的Unix shell/Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - `csh/tcsh` BSD Unix C shell
  - `bash` "Bourne-Again" Shell (default Linux shell)

```
int main()
{
   char cmdline[MAXLINE]; /* command line */

   while (1) {
      /* read */
      printf("> ");
      Fgets(cmdline, MAXLINE, stdin);
      if (feof(stdin))
         exit(0);

      /* evaluate */
      eval(cmdline);
   }
}
```
shellex.c

执行的过程就是一系列读/求值的步骤/Execution is a sequence of read/evaluate steps

简单的Shell eval函数/Simple Shell `eval` Function

```
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 */
                if (!bg) {
            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程序存在的问题/Problem with Simple Shell Example

- **例子shell只能等待并回收前台任务/Our example shell correctly waits for and reaps foreground jobs**

- 后台任务怎么办？ **/But what about background jobs?**
  - 终止后变成僵尸/Will become zombies when they terminate
  - 由于shell不会终止，所以永远不会被回收/Will never be reaped because shell (typically) will not terminate
  - 会造成系统内存泄露并耗尽内核内存/Will create a memory leak that could run the kernel out of memory

可以利用ECF解决/ECF to the Rescue!

- **解决方案：异常控制流/Solution: Exceptional control flow**
  - 系统在后台进程处理完成后打断正常处理流程并提醒我们/The kernel will interrupt regular processing to alert us when a background process completes
  - Unix系统中这种提醒的机制是信号/In Unix, the alert mechanism is called a ***signal***

Today

- Shells
- 信号/Signals
- 非局部跳转/Nonlocal jumps

信号/Signals

- **信号是用来通知一个进程某种类型的事件在系统中发生了/A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - 类似于异常和中断/Akin to exceptions and interrupts
  - 由内核发送给一个进程（有时是根据另一个进程的请求）Sent from the kernel (sometimes at the request of another process) to a process
  - 信号的类型是用1-30的小整型标识/Signal type is identified by small integer ID’s (1-30)
  - 信号的唯一信息就是这个ID以及信号达到的事实/Only information in a signal is its ID and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	用户输入ctrl-c/User typed ctrl-c
9	SIGKILL	Terminate	杀死程序（不能覆盖或被忽略）Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	段错误/Segmentation violation
14	SIGALRM	Terminate	时钟信号/Timer signal
17	SIGCHLD	Ignore	子进程停止或者终止/Child stopped or terminated

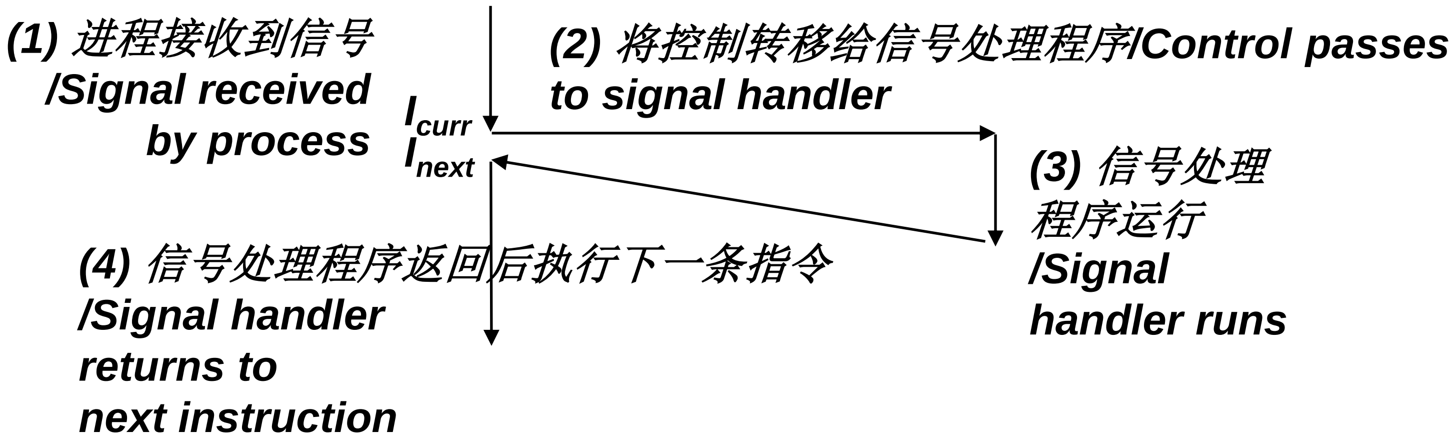
信号概念：发送一个信号/Signal Concepts: Sending a Signal

- **内核通过更新目标进程的某些状态来发送一个信号给目标进程/Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process**

- **内核发送信号是由于以下原因中的一个/Kernel sends a signal for one of the following reasons:**
  - 内核侦测到除零错误或者子进程终止等系统事件/Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - 另外一个进程调用了kill系统调用显式请求内核发送一个信号给目标进程/Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process

信号概念：接收一个信号/Signal Concepts: Receiving a Signal

- 目标进程接收信号是由于系统内核强制其对某个信号的发送做出响应/A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- 可能的响应方式/Some possible ways to react:
  - ***忽略***信号（什么也不做）***/Ignore*** the signal (do nothing)
  - ***终止进程***（可以选择对信息转储）***/Terminate*** the process (with optional core dump)
  - ***调用***用户级信号处理函数对信号进行处理***/Catch*** the signal by executing a user-level function called ***signal handler***
    - 类似于硬件异常处理函数对异步中断的响应/Akin to a hardware exception handler being called in response to an asynchronous interrupt:



(1) 进程接收到信号 /Signal received by process
I_{curr}
I_{next}
(2) 将控制转移给信号处理程序/Control passes to signal handler
(3) 信号处理程序运行 /Signal handler runs
(4) 信号处理程序返回后执行下一条指令 /Signal handler returns to next instruction

信号概念：挂起或者阻塞的信号/Signal Concepts: Pending and Blocked Signals

- **已经发送但是没有被接收的信号处于挂起状态/A signal is *pending* if sent but not yet received**
  - 任何特定类型的信号最多有一个挂起的/There can be at most one pending signal of any particular type
  - 重要：信号不排队/Important: Signals are not queued
    - 如有某个进程有一个类型为k的信号挂起，泽后续发给该进程的其他信号被直接抛弃/If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **一个进程会阻塞某种特定类型信号的接收/A process can *block* the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

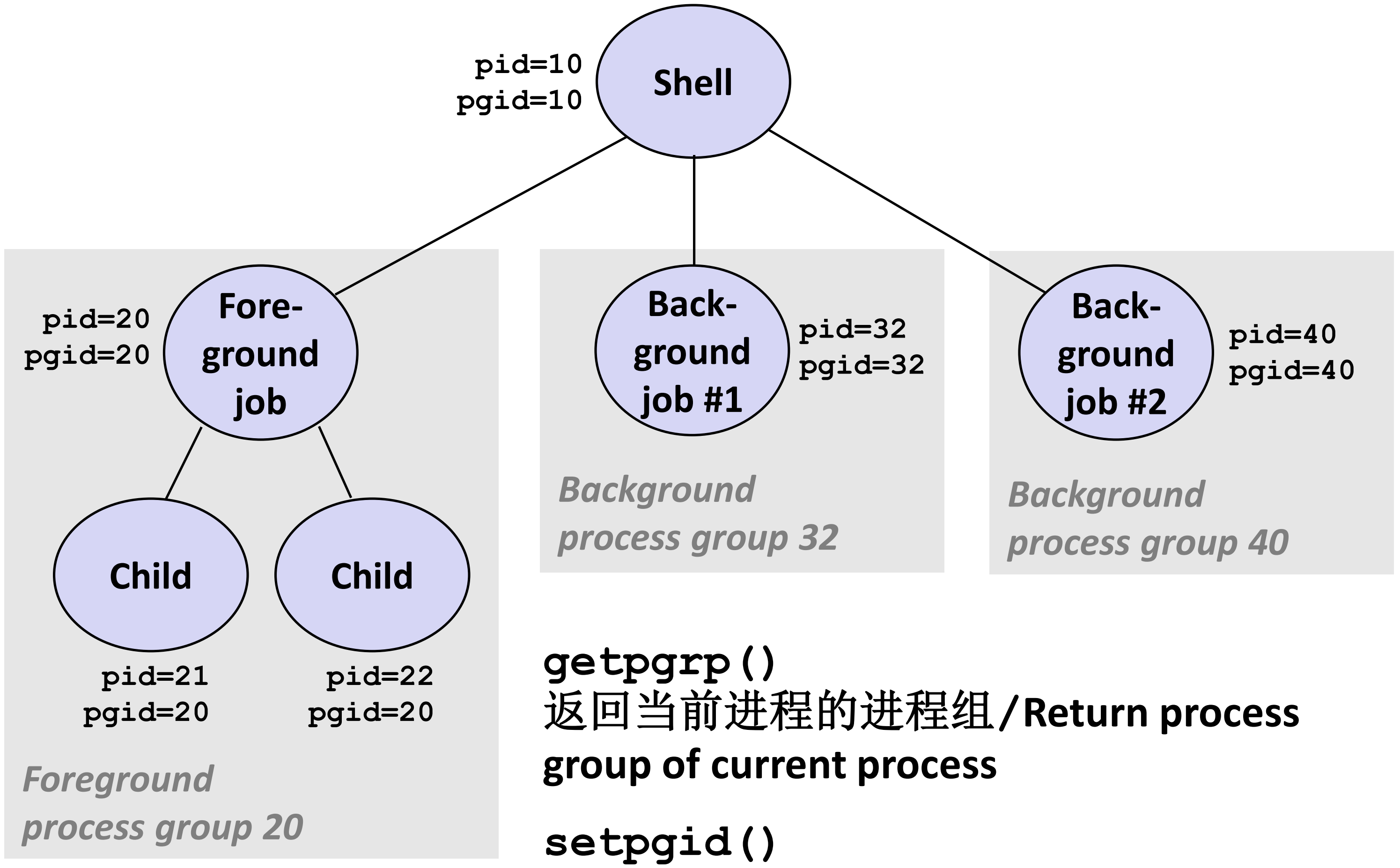
- **挂起的信号最多被接收一次/A pending signal is received at most once**

信号概念：挂起/阻塞位/Signal Concepts: Pending/Blocked Bits

- 内核在每个进程的上下文维护一个挂起和阻塞的比特向量/**Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - **挂起：表示挂起的信号集合/`pending`**: represents the set of pending signals
    - 当发送了一个k类型的信号时系统设置第k个比特位/Kernel sets bit k in `pending` when a signal of type k is delivered
    - 当类型k的信号被接收后系统会将第k个比特位清零/Kernel clears bit k in `pending` when a signal of type k is received
  - **阻塞：表示阻塞的信号集合/`blocked`**: represents the set of blocked signals
    - 可以使用`sigprocmask` 设置或者清除/Can be set and cleared by using the `sigprocmask` function
    - 也称为信号掩码/Also referred to as the *signal mask*.

发送信号：进程组/Sending Signals: Process Groups

- 每个进程只属于一个进程组/**Every process belongs to exactly one process group**



`getpgrp()`
返回当前进程的进程组/**Return process group of current process**

`setpgid()`
修改当前进程的进程组/**Change process group of a process (see text for details)**

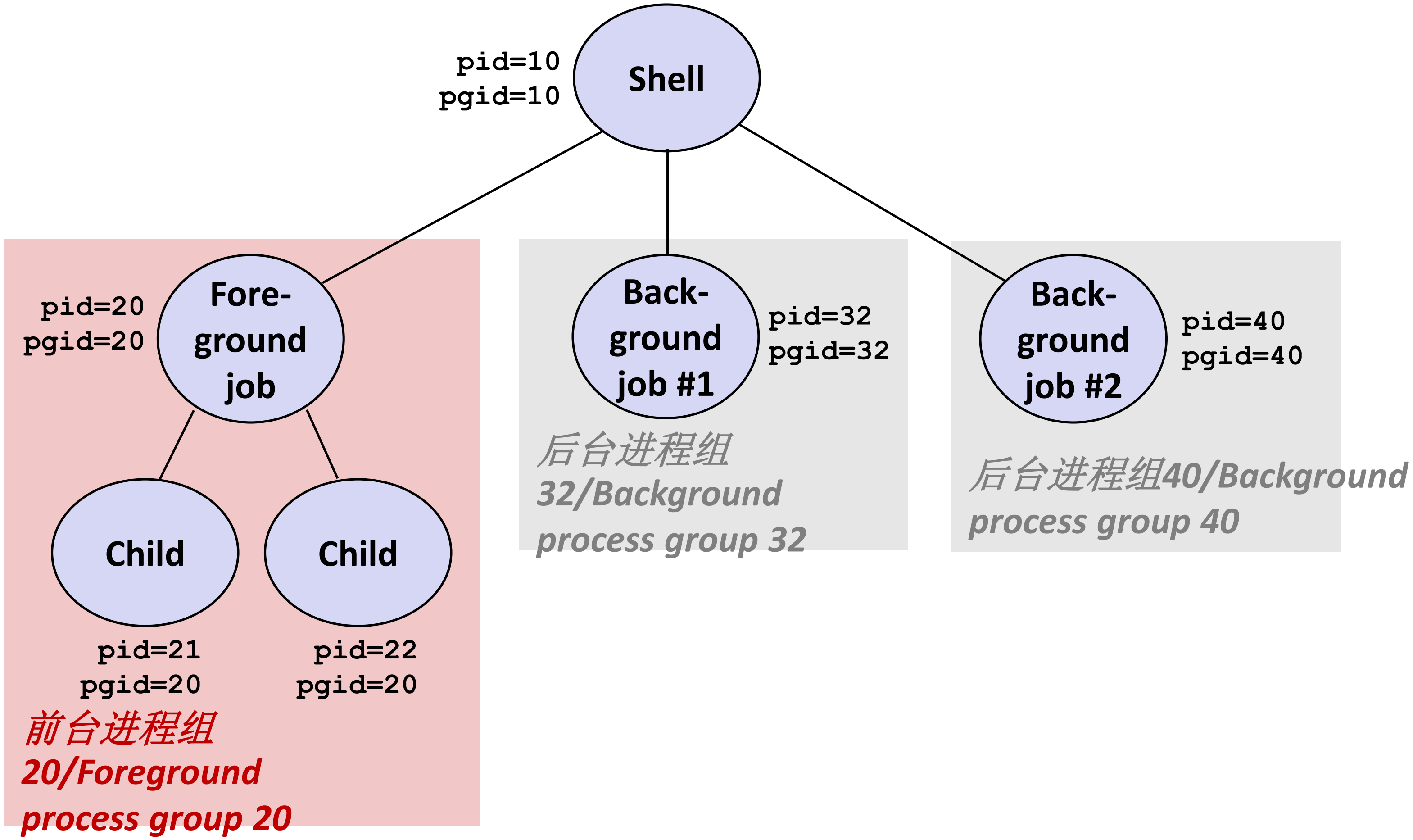
通过/bin/kill程序发送信号/Sending Signals with `/bin/kill` Program

- `/bin/kill`程序可以发送任意信号给一个进程或者进程组/ `/bin/kill` program sends arbitrary signal to a process or process group
- 例如/Examples
  - `/bin/kill –9 24818` **发送SIGKILL给进程24818**/Send SIGKILL to process 24818
  - `/bin/kill –9 –24817` **发送**SIGKILL**给进程组的每个进程**/Send SIGKILL to every process in process group 24817

```
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>
```

通过键盘发送信号/Sending Signals from the Keyboard

- **输入ctrl-c(ctrl-z)会导致系统内核发送一个SIGINT (SIGTSTP) 信号给前台进程组的每个任务/Typing ctrl-c (ctrl-z) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.**
  - SIGINT – default action is to terminate each process/默认终止每个进程
  - SIGTSTP – default action is to stop (suspend) each process/默认停止（挂起）每个进程



ctrl-c和ctrl-z示例/Example of ctrl-c and ctrl-z

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./forks 17
28108 pts/8    T      0:01 ./forks 17
28109 pts/8    R+     0:00 ps w
bluefish> fg
./forks 17
<types ctrl-c>
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

进程状态STAT标记/STAT (process state) Legend:

First letter/第一个字母:

S: sleeping/睡眠
T: stopped/停止
R: running/运行

Second letter/第二个字母:

s: session leader
+: foreground proc group

See "man ps" for more details

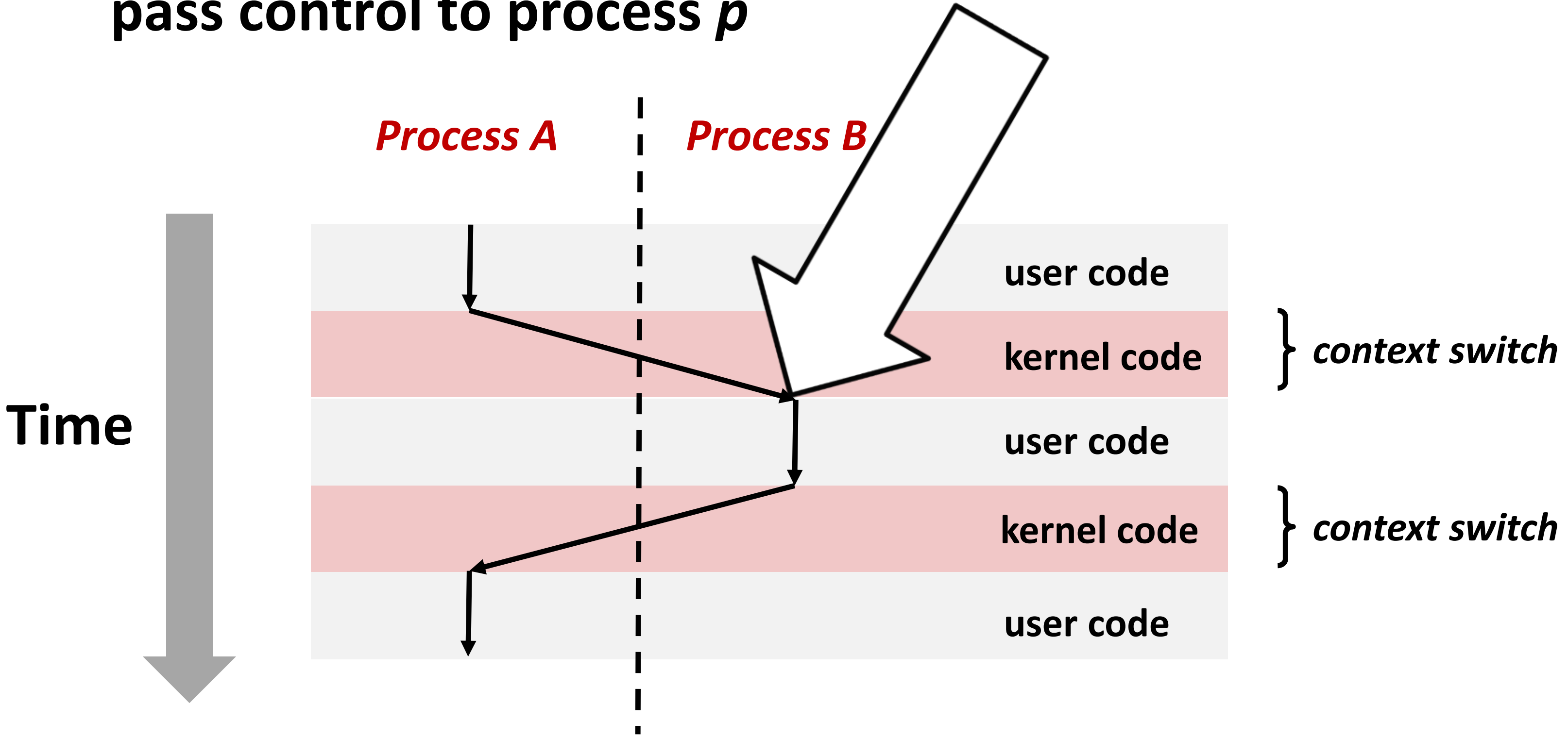
通过kill函数发送信号/Sending Signals with `kill` Function

```
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);
    }
}
```

forks.c

接收信号/Receiving Signals

- 假设从内核信号处理函数返回到进程p/Suppose kernel is returning from an exception handler and is ready to pass control to process *p*



接收信号/Receiving Signals

- **假设从内核信号处理函数返回到进程p/Suppose kernel is returning from an exception handler and is ready to pass control to process *p***
- **内核计算/Kernel computes `pnb = pending & ~blocked`**
  - 进程p挂起但非阻塞信号/The set of pending nonblocked signals for process *p*
- **If (`pnb == 0`)**
  - 将控制权交给p逻辑控制流的下一条指令/Pass control to next instruction in the logical flow for *p*
- **Else**
  - 选择pnb中最低非0位k并强制p接收信号k/Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*
  - 信号的接收出发了p的某些动作/The receipt of the signal triggers some ***action*** by *p*
  - 对pnb中每个非0位重复上述过程/Repeat for all nonzero *k* in **`pnb`**
  - 将控制权交给p逻辑控制流的下一条指令/Pass control to next instruction in logical flow for *p*

默认动作/Default Actions

- 每种类型的信号有一个预定义的默认动作，可能是如下中的一个/**Each signal type has a predefined *default action*, which is one of:**
  - 终止进程/The process terminates
  - 停止进程，直到接收到SIGCONT时重启/The process stops until restarted by a SIGCONT signal
  - 进程忽略掉信号/The process ignores the signal

安装信号处理程序/Installing Signal Handlers

- **函数Signal修改信号signum对应的默认行为/The `signal` function modifies the default action associated with the receipt of signal `signum`:**
  - **`handler_t *signal(int signum, handler_t *handler)`**

- **可能的选项/Different values for `handler`:**
  - SIG_IGN: ignore signals of type **`signum`**/忽略signum类型的信号
  - SIG_DFL: revert to the default action on receipt of signals of type **`signum`**/接收到signum类型的信号时按照默认动作处理
  - 否则handler是用于级信号处理程序的地址/Otherwise, **`handler`** is the address of a user-level ***signal handler***
    - 当进程接收到类型为signum的信号时调用/Called when process receives signal of type **`signum`**
    - 称为安装信号处理程序/Referred to as ***"installing"*** the handler
    - 执行信号处理程序称为/Executing handler is called ***"catching"*** or ***"handling"*** the signal
    - 当信号处理程序返回时，控制权交给进程接收到信号时被打断的指令/When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

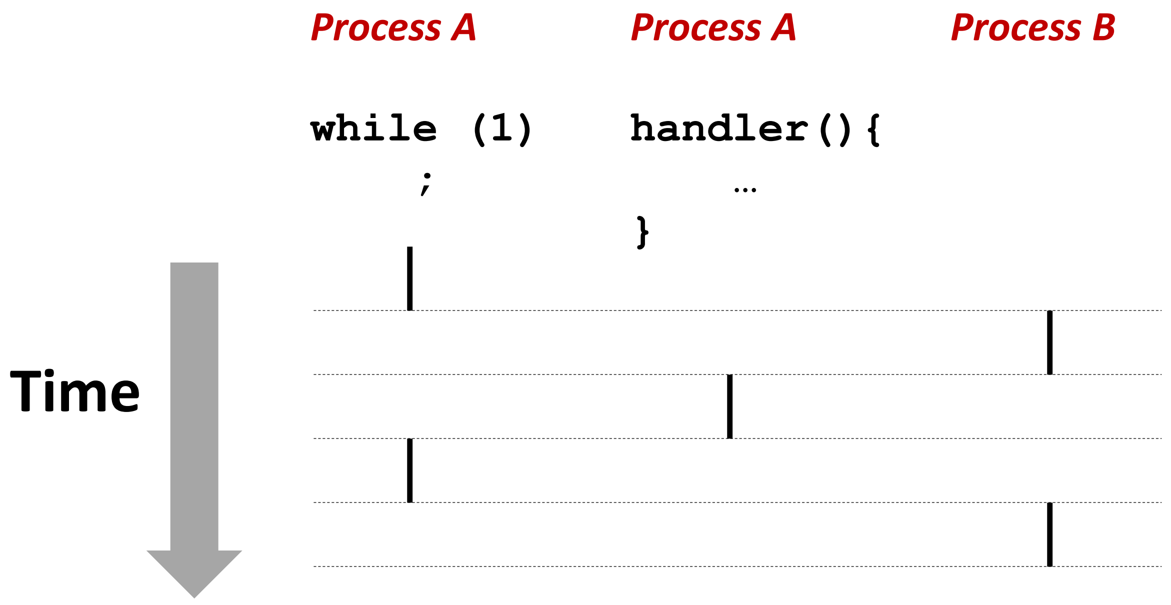
信号处理例子/Signal Handling Example

```
void sigint_handler(int sig) /* SIGINT handler */
{
   printf("So you think you can stop the bomb with ctrl-c, do you?\n");
   sleep(2);
   printf("Well...");
   fflush(stdout);
   sleep(1);
   printf("OK. :-)\n");
   exit(0);
}

int main()
{
   /* Install the SIGINT handler */
   if (signal(SIGINT, sigint_handler) == SIG_ERR)
      unix_error("signal error");

   /* Wait for the receipt of a signal */
   pause();

   return 0;
}
```

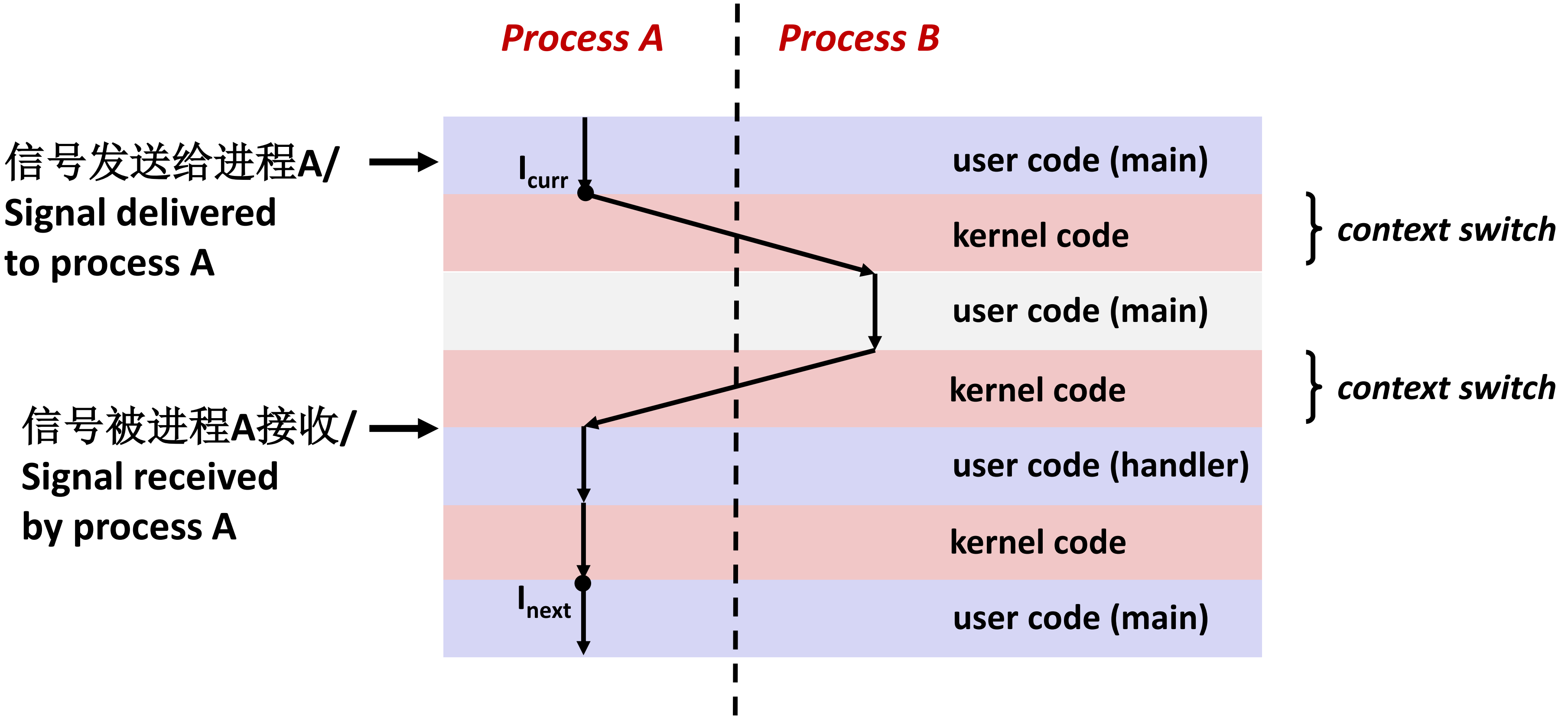
sigint.c

Signals Handlers as Concurrent Flows

- 每个信号处理程序都是一个独立的逻辑控制流（非进程），与主程序并发执行/A signal handler is a separate logical flow (not process) that runs concurrently with the main program

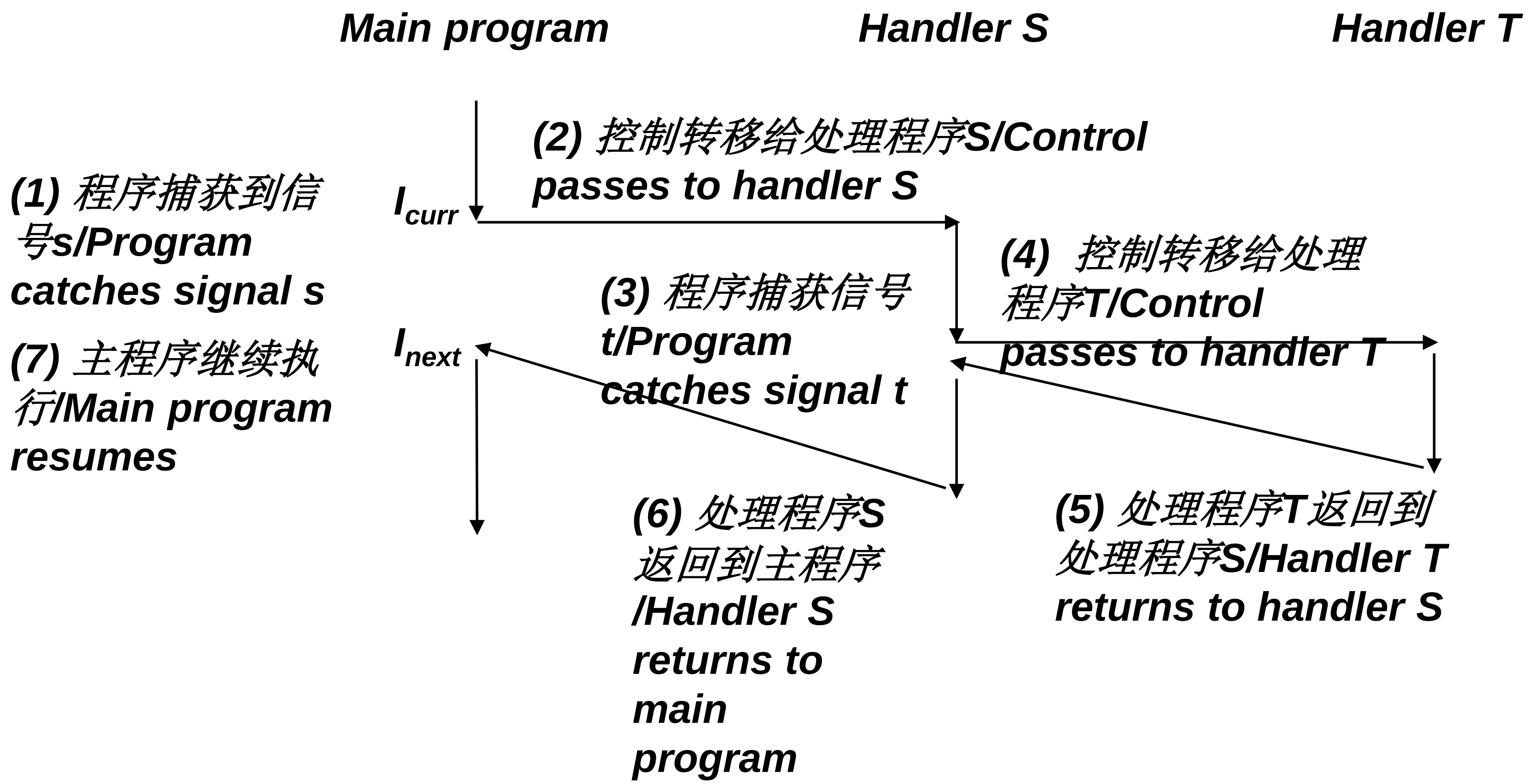


信号处理程序作为并发控制流的另一个视图/Another View of Signal Handlers as Concurrent Flows



嵌套信号处理/Nested Signal Handlers

- 信号处理程序可能被另一个信号处理程序打断 /Handlers can be interrupted by other handlers



Main program
Handler S
Handler T
(2) 控制转移给处理程序S/Control passes to handler S
(1) 程序捕获到信号s/Program catches signal s
I_{curr}
(4) 控制转移给处理程序T/Control passes to handler T
(3) 程序捕获信号t/Program catches signal t
I_{next}
(7) 主程序继续执行/Main program resumes
(6) 处理程序S返回到主程序 /Handler S returns to main program
(5) 处理程序T返回到处理程序S/Handler T returns to handler S

阻塞和解除信号阻塞/Blocking and Unblocking Signals

- **隐式阻塞机制/Implicit blocking mechanism**
  - Kernel会阻塞正在被处理的任何信号/Kernel blocks any pending signals of type currently being handled.
  - 例如SIGINT处理程序不能被另一个SIGINT打断/E.g., A SIGINT handler can't be interrupted by another SIGINT

- **显式阻塞和解除阻塞机制/Explicit blocking and unblocking mechanism**
  - `sigprocmask` function

- **支持函数/Supporting functions**
  - `sigemptyset` – Create empty set/创建一个空的集合
  - `sigfillset` – Add every signal number to set/对集合设置每个信号
  - `sigaddset` – Add signal number to set/对集合设置某个信号
  - `sigdelset` – Delete signal number from set/将信号从集合删除

临时阻塞信号/Temporarily Blocking Signals

```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
  :
  /* Code region that will not be interrupted by SIGINT */
  :
/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

安全的信号处理/Safe Signal Handling

- **信号处理程序比较复杂，是因为他们是和主程序并发运行的，并且共享同样的全局数据结构/Handlers are tricky because they are concurrent with main program and share the same global data structures.**
  - 共享数据结构更容易被破坏/Shared data structures can become corrupted.

- **我们在后面讨论并发的问题/We'll explore concurrency issues later in the term.**

- **现在只给一些避免并发的提示/For now here are some guidelines to help you avoid trouble.**

编写安全处理程序的提示/Guidelines for Writing Safe Handlers

- **G0: 中断处理程序越简单越好/Keep your handlers as simple as possible**
  - 例如，设置全局标记后返回/e.g., Set a global flag and return
- **G1: 只调用异步安全的信号处理函数/Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe! /这些都不安全
- **G2: 进入和退出时保存errno/Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
- **G3:临时阻塞所有的信号后再访问全局数据结构/ Protect accesses to shared data structures by temporarily blocking all signals.**
  - 避免可能的破坏/To prevent possible corruption
- **G4: 将全局变量声明位volatile/Declare global variables as `volatile`**
  - 避免编译器将其存储在寄存器中/To prevent compiler from storing them in a register
- **G5: 将全局标记声明为volatile `sig_atomic_t` / Declare global flags as `volatile sig_atomic_t`**
  - *Flag*只读或只写的变量/:*flag*: variable that is only read or written (e.g. flag = 1, not flag++)
  - 按照这种方式声明的变量不需要像其他全局变量那样保护/Flag declared this way does not need to be protected like other globals

Async-Signal-Safety

- **如果一个函数是可重入的或者不可以被信号打断的则将其称为*async-signal-safe* / Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.**
- **Posix中有117个函数是async-signal-safe /Posix guarantees 117 functions to be async-signal-safe**
  - Source: “`man 7 signal`”
  - Popular functions on the list:
    - `_exit, write, wait, waitpid, sleep, kill`
  - 常见的函数并不在其中/Popular functions that are **not** on the list:
    - `printf`, `sprintf`, `malloc, exit`
    - 不幸的事实：write是唯一async-signal-safe 输出函数 / Unfortunate fact: `write` is the only async-signal-safe output function

安全生成格式化输出/Safely Generating Formatted Output

- 在中断处理程序中使用可重入的SIO/Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers.
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
   Sio_puts("So you think you can stop the bomb with ctrl-c, do you?\n");
   sleep(2);
   Sio_puts("Well...");
   sleep(1);
   Sio_puts("OK. :-)\n");
   _exit(0);
}
```
sigintsafe.c

正确的信号处理 /Correct Signal Handling

```
int ccount = 0;
void child_handler(int sig) {
   int olderrno = errno;
   pid_t pid;
   if ((pid = wait(NULL)) < 0)
      Sio_error("wait error");
   ccount--;
   Sio_puts("Handler reaped child ");
   Sio_putl((long)pid);
   Sio_puts(" \n");
   sleep(1);
   errno = olderrno;
}

void fork14() {
   pid_t pid[N];
   int i;
   ccount = N;
   Signal(SIGCHLD, child_handler);

   for (i = 0; i < N; i++) {
      if ((pid[i] = Fork()) == 0) {
         Sleep(1);
         exit(0);  /* Child exits */
      }
   }
   while (ccount > 0) /* Parent spins */
      ;
}
```

forks.c

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```

- **挂起的信号是不排队的/Pending signals are not queued**
  - 对每个信号类型，只用一个比特位来标识是否有信号被挂起/For each signal type, one bit indicates whether or not signal is pending…
  - 因此每种最多有一个挂起的信号/…thus at most one pending signal of any particular type.
- **不可以使用信号对事件计数，例如子进程终止等/You can't use signals to count events, such as children terminating.**

正确信号处理/Correct Signal Handling

- 必须等待所有终止的子进程/Must wait for all terminated child processes
  - 将wait放入到循环中以回收所有终止的子进程/Put `wait` in a loop to reap all terminated children

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

```
whaleshark> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
whaleshark>
```

可移植的信号处理/Portable Signal Handling

- **不同的Unix版本有不同的信号处理语义/Ugh! Different versions of Unix can have different signal handling semantics**
  - 一些早期的系统在捕获到信号后会恢复默认动作/Some older systems restore action to default after catching signal
  - 有些被中断的系统调用会返回errno == EINTR /Some interrupted system calls can return with errno == EINTR
  - 有的系统并不阻塞正在被处理的信号/Some systems don’t block signals of the type being handled
- **Solution: `sigaction`**

```
handler_t *Signal(int signum, handler_t *handler)
{
   struct sigaction action, old_action;

   action.sa_handler = handler;
   sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
   action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

   if (sigaction(signum, &action, &old_action) < 0)
      unix_error("Signal error");
   return (old_action.sa_handler);
}
```
csapp.c

同步控制流避免数据竞争/Synchronizing Flows to Avoid Races

- 简单的shell程序有个不易发现的同步问题，因为其假设父进程先于子进程 /Simple shell with a subtle synchronization error because it assumes parent runs before child.

```c
int main(int argc, char **argv)
{
   int pid;
   sigset_t mask_all, prev_all;

   Sigfillset(&mask_all);
   Signal(SIGCHLD, handler);
   initjobs(); /* Initialize the job list */

   while (1) {
      if ((pid = Fork()) == 0) { /* Child */
         Execve("/bin/date", argv, NULL);
      }
      Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
      addjob(pid);  /* Add the child to the job list */
      Sigprocmask(SIG_SETMASK, &prev_all, NULL);
   }
   exit(0);
}
```

procmask1.c

同步控制流以避免竞争/Synchronizing Flows to Avoid Races

- 简单shell的SIGCHLD处理程序/SIGCHLD handler for a simple shell

```
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```
procmask1.c

没有数据竞争问题的shell程序/Corrected Shell Program without Race

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
                addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

procmask2.c

显式等待信号/Explicitly Waiting for Signals

- 显式等待SIGCHLD信号的到来/Handlers for program explicitly waiting for SIGCHLD to arrive.

```
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
   int olderrno = errno;
   pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
   errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

显式等待信号/Explicitly Waiting for Signals

```
int main(int argc, char **argv) {
   sigset_t mask, prev;
   Signal(SIGCHLD, sigchld_handler);
   Signal(SIGINT, sigint_handler);
   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)
        ;
            /* Do some work after receiving SIGCHLD */
     printf(".");
   }
   exit(0);
}
```

类似于shell等待一个前台的任务终止/ Similar to a shell waiting for a foreground job to terminate.

waitforsignal.c

显式等待信号/Explicitly Waiting for Signals

- 程序是对的，但是太浪费资源/Program is correct, but very wasteful
- 其他选项/Other options:

```
while (!pid)   /* Race! */
    pause();
```

```
while (!pid) /* Too slow! */
   sleep(1);
```

- 解决方案/Solution: `sigsuspend`

使用sigsuspend等待信号/Waiting for Signals with sigsuspend

- `int sigsuspend(const sigset_t *mask)`

- 等价于原子版本的：**Equivalent to atomic (uninterruptable) version of:**

```
sigprocmask(SIG_BLOCK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

使用sigsuspend等待信号/Waiting for Signals with sigsuspend

```c
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

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

       /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);

       /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```

sigsuspend.c

Today

- Shells
- 信号/Signals
- **非局部跳转/Nonlocal jumps**
  - 参见教材/Consult your textbook and additional slides

总结/Summary

- 信号对应进程级异常处理/**Signals provide process-level exception handling**
  - 可以从用户程序产生/Can generate from user programs
  - 可以声明信号处理程序定义处理效果/Can define effect by declaring signal handler
  - 编写信号处理函数的时候要特别小心/Be very careful when writing signal handlers

- 非局部跳转给出了进程内部的异常控制流/**Nonlocal jumps provide exceptional control flow within process**
  - 遵守栈相关的原则/Within constraints of stack discipline

Additional slides

非局部跳转/Nonlocal Jumps: `setjmp/longjmp`

- **将控制转移到任意位置的强大（但比较危险）用户级机制/Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
  - 受控的打破call/return规则的方式/Controlled to way to break the procedure call / return discipline
  - 通常用户错误处理和信号处理/Useful for error recovery and signal handling

- **`int setjmp(jmp_buf j)`**
  - 必须在longjmp之前调用/Must be called before longjmp
  - 给出后续longjmp对应的返回位置/Identifies a return site for a subsequent longjmp
  - 一次调用，返回一次或者多次/Called **once**, returns **one or more** times

- **实现/Implementation:**
  - 通过将寄存器上下文、堆栈指针和PC值等存储在jmp_buf中记住当前位置 /Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in **`jmp_buf`**
  - 返回0/Return 0

setjmp/longjmp (cont)

- **`void longjmp(jmp_buf j, int i)`**
  - Meaning:
    - return from the **`setjmp`** remembered by jump buffer **`j`** again ...
    - ... this time returning **`i`** instead of 0
  - Setjmp之后调用/Called after **`setjmp`**
  - 一次调用但是不返回/Called **once**, but **never** returns

- **`Longjmp`实现/`longjmp` Implementation:**
  - 从跳转缓冲区j中恢复寄存器上下文/Restore register context (stack pointer, base pointer, PC value) from jump buffer **`j`**
  - 将返回值寄存器%eax设置为i/Set **`%eax`** (the return value) to **`i`**
  - 跳转到跳转缓冲j中指定的位置/Jump to the location indicated by the PC stored in jump buf **`j`**

setjmp/longjmp Example

- 目标：从深度嵌套的函数直接返回最开始的调用者
- Goal: return directly to original caller from a deeply-nested function

```
/* Deeply nested function foo */
void foo(void)
{
   if (error1)
                longjmp(buf, 1);
   bar();
}

void bar(void)
{
   if (error2)
      longjmp(buf, 2);
}
```

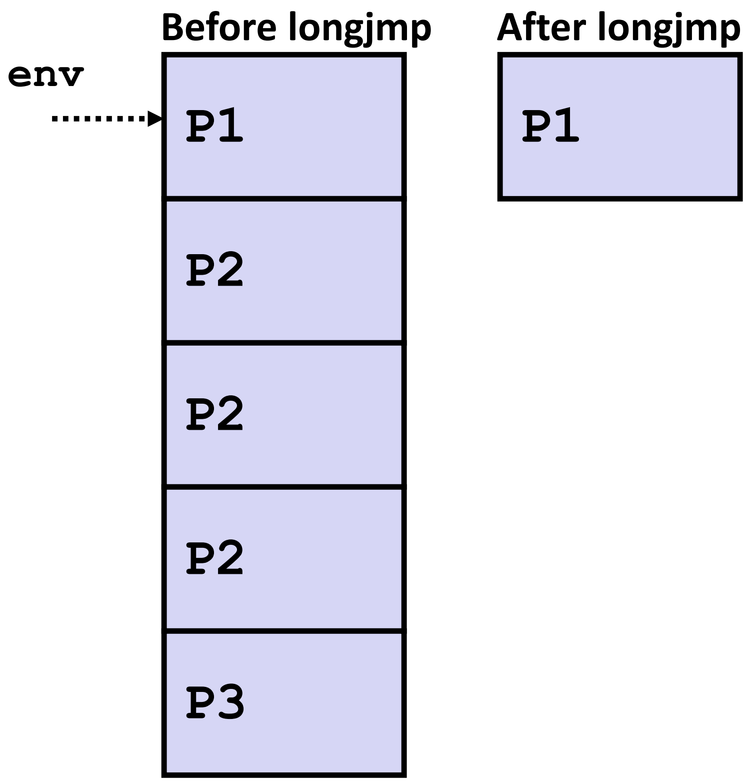
setjmp/longjmp Example (cont)

```
jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
  switch(setjmp(buf)) {
  case 0:
    foo();
    break;
  case 1:
    printf("Detected an error1 condition in foo\n");
    break;
  case 2:
    printf("Detected an error2 condition in foo\n");
    break;
  default:
    printf("Unknown error condition in foo\n");
  }
  exit(0);
}
```

非局部跳转的限制/Limitations of Nonlocal Jumps

- 基于栈原理工作/Works within stack discipline
  - 只能跳转到已经调用但是还没有完成的函数/Can only long jump to environment of function that has been called but not yet completed

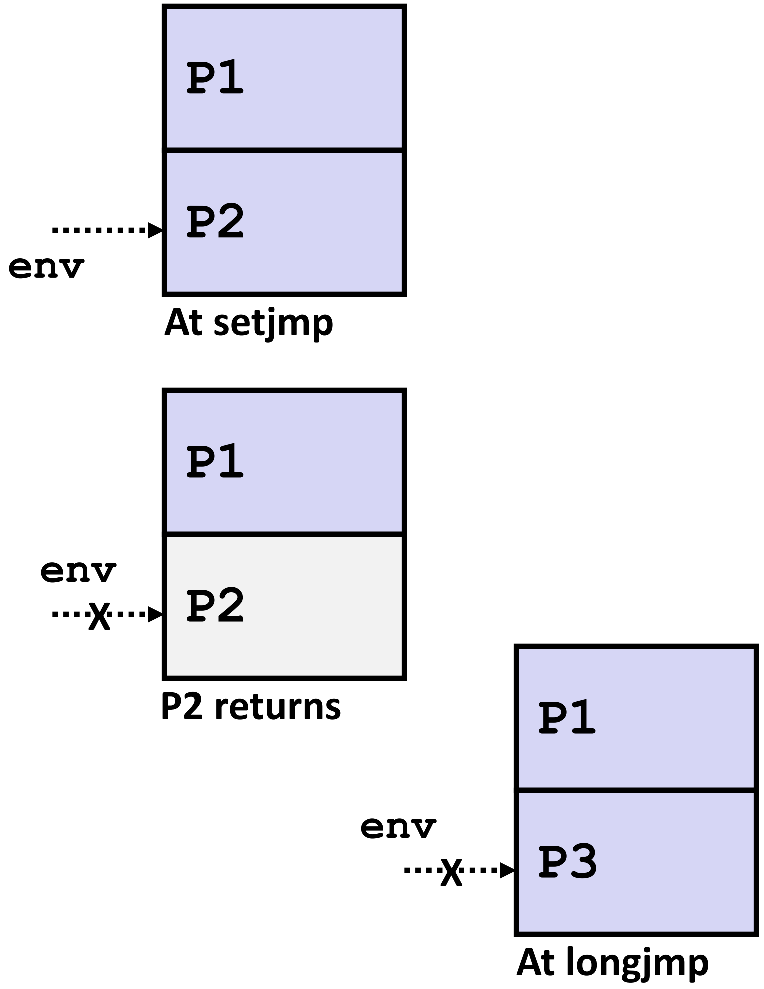
```
jmp_buf env;

P1()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  } else {
    P2();
  }
}

P2()
{  . . . P2(); . . . P3(); }

P3()
{
  longjmp(env, 1);
}
```



非局部跳转的限制/ Limitations of Long Jumps (cont.)

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1()
{
  P2(); P3();
}

P2()
{
   if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3()
{
  longjmp(env, 1);
}
```



Putting It All Together: A Program That Restarts Itself When `ctrl-c`'d

```
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig)
{
   siglongjmp(buf, 1);
}

int main()
{
   if (!sigsetjmp(buf, 1)) {
      Signal(SIGINT, handler);
            Sio_puts("starting\n");
   }
   else
      Sio_puts("restarting\n");

   while(1) {
            Sleep(1);
            Sio_puts("processing...\n");
   }
   exit(0); /* Control never reaches here */
}
```

restart.c

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting
processing...
processing...
restarting
processing.
processing...
processing...
```

Ctrl-c

Ctrl-c