计算机系统

第8章（下）

信号

湖南大学

《计算机系统》课程教学组

授课教师：肖雄仁

异常控制流存在于系统各个层次

- **异常**
  - 硬件与操作系统内核软件
- **进程上下文切换**
  - 硬件计时器和内核软件

上半部分

- **信号**
  - 内核软件和应用软件
- **非本地跳转**
  - 应用程序代码

下半部分

本讲内容

信号
signals

多任务与外壳
Multitasking & shells

非本地跳转
Nonlocal jumps

多任务系统

- **系统并发执行多个进程**
- **进程：正在执行的程序实例**
  - 其状态包括：存储器映像，寄存器值，程序计数器
- **定期从一个进程切换到另一个进程**
  - 当进程需要输入/输出资源或发生计时器事件，挂起进程
  - 当输入/输出可用或设定了调度优先级，则恢复进程
- **对于用户而言，所有进程看起来都是同时执行**
  - 大部分系统，一次只能执行一个进程
  - 除非比单独运行时性能更低

程序员的多任务模型

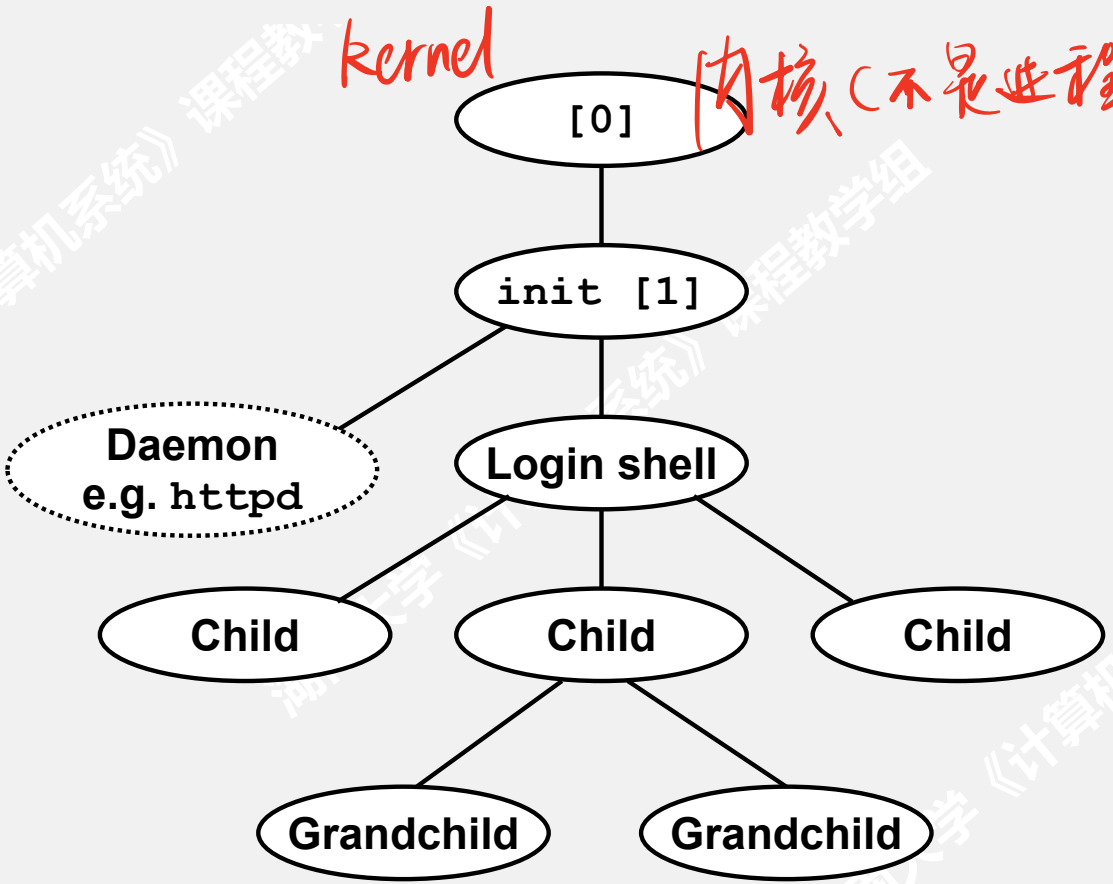
基本函数

- `fork` 产生新进程
  - 调用一次，返回两次
- `exit` 终止自身进程
  - 调用一次，不返回
  - 可能进入僵死状态
- `wait` 与 `waitpid` 等待并回收终止的子进程
- `execve` 在现有进程上运行新程序
  - 调用一次，通常不返回

编程挑战

- 理解函数的非标准语义
- 避免不恰当使用系统资源
  - 例如， “Fork bombs” 可能会令系统崩溃

Unix&Linux 进程层次结构



Shell程序

- 外壳 *shell* 是一个应用程序，运行与用户交互相关的程序
  - **bsh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **csh** BSD Unix C shell (**tcsh**: enhanced csh at CMU and elsewhere)
  - **bash** "Bourne-Again" Shell

```
int main() {
    char cmdline[MAXLINE];

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

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

执行的是一个读/评估的序列

简单shell的eval函数

```
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    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);
        }
    }

    if (!bg) {   /* parent waits for fg job to terminate */
        int status;
        if (waitpid(pid, &status, 0) < 0)
            unix_error("waitfg: waitpid error");
    }
    else         /* otherwise, don't wait for bg job */
        printf("%d %s", pid, cmdline);
    }
}
```

```
int parseline(char *buf, char **argv)
{    …
   /* Build the argv list*/
     …
   /*Should the job run in the background?*/
    if ((bg = (*argv [argc-1] =='&')) != 0)
        argv[--argc] = NULL;
    …
    return bg;
}
```

前台进程

后台进程

什么是“后台作业（Background Job）”？

- 一般而言，用户一次执行一个命令
  - 输入命令，读输出结果，再输入下一条指令
- 部分程序会运行很长时间
  - 例如：“在两小时内删除一个文件”

```
unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
```

- 后台作业是一个无需等待的进程

```
unix> (sleep 7200 ; rm /tmp/junk) &
[1] 907
unix> # ready for next command
```

简单外壳(Shell)示例的问题

- shell示例正确地等待并回收前台作业
- 可是对于后台作业呢?
  - 当它们终止后会变成僵尸进程
  - 永远不会被回收，因为shell（通常）不会终止
  - 产生内存泄漏，导致内核运行内存不足
  - 现代 Unix: 一旦运行进程数超过配额，在shell中不能再运行任何新命令，即fork()函数此时会返回-1

```
unix> limit maxproc         # csh syntax
maxproc       7251

unix> ulimit -u             # bash syntax
7251
```

异常控制流提供解决方案

- 问题
  - shell并不知道后台作业什么时候完成
  - 从本质上讲，它可能随时发生
  - shell的常规控制流程无法及时回收结束的后台进程
  - 常规控制流是“等到正在运行的作业完成，然后回收它”
- 解决方案: 异常控制流
  - 内核将中断常规处理，以预警的方式提示后台作业完成
  - 在Unix中, 这种预警机制称为 **信号 (signal)**

本讲内容

信号
signals

多任务与外壳
Multitasking & shells

非本地跳转
Nonlocal jumps

信号(Signals)

- *signal* 是一条小消息，通知进程，发生了某种类型的系统事件
  - 与异常和中断类似
  - 来自 **内核**（有时是应另一个进程的请求），发送到一个进程
  - 信号类型定义为1-30 的整型 ID 号（Linux系统）
  - 信号内消息就是其ID号，表示信号到达

ID	名称	缺省操作	相应事件
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
4	SIGILL	Core	Illegal Instruction
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

发送信号

- 传送一个信号到目的进程有两个步骤
  - **发送信号**和**接收信号**
- 内核通过更新目标进程的上下文的某些状态，实现从内核发送一个信号到目标进程
- 内核发送信号的可能原因：
  - 内核检测到系统事件，如被零除（SIGFPE）或子进程终止（SIGCHLD）
  - 另一个进程调用KILL系统调用，来显式请求内核发送信号给目标进程

接收信号

- 目标进程接收一个信号，以某种方式响应这个内核发送的信号
- 三种可能的响应方式：
  - *Ignore 忽略* 这个信号 (**什么也不做**)
  - *Terminate 结束* 这个进程 (可选core dump)
  - *Catch捕获* 这个信号，通过执行用户级的函数信号处理程序（signal handler）
    - 类似于为响应异步中断而调用的硬件异常处理程序

待处理和阻塞信号

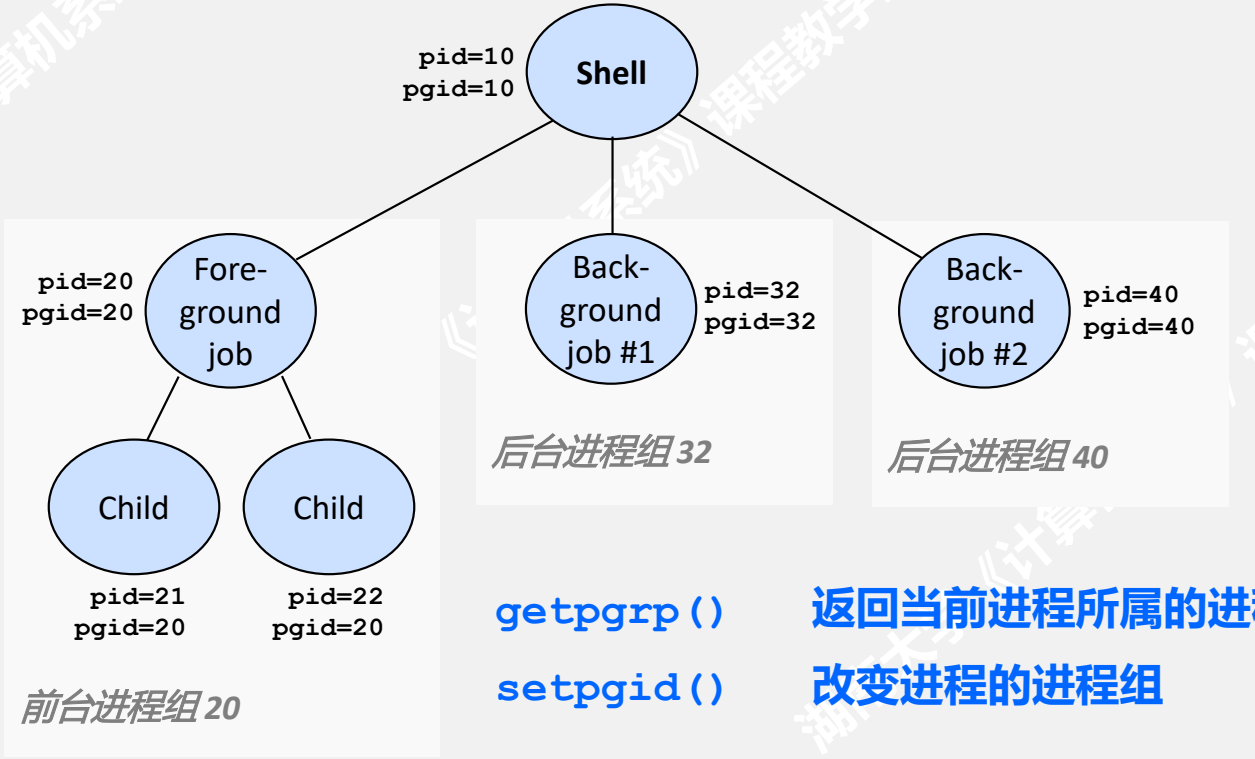
- 如果一个信号被发出但没被接收，则该信号叫待处理信号(pending signal)
  - 对于任何类型的信号，最多只有一个待处理信号 (位向量)
  - **注意：待处理信号不能被排队**
    - 如果某进程有一个类型为K的待处理信号，则后续发送给该进程的类型K的信号均被丢弃
- 一个进程可以有选择性地阻塞 接收特定类型的信号
  - 被阻塞信号可以被传送，但不会被接收，直至信号被停止阻塞

信号的基本概念

- 一个待处理信号最多被接收一次
- 内核在每个进程的上下文中维护待处理pending和阻塞blocked的位向量
  - **pending位向量**：表示待处理信号的集合
    - 当类型k的信号被传送时，内核将pending位向量的第k位置位
    - 当类型k的信号被接收时，内核会清除pending中的第k位
  - **blocked位向量**：表示被阻塞的集合
    - 使用**sigprocmask** 函数可以设置和清除被阻塞置位

进程组

- 每个进程都只属于一个进程组



`getpgrp()` **返回当前进程所属的进程组号**

`setpgid()` **改变进程的进程组**

信号发送方法一：使用/bin/kill程序

- /bin/kill程序给进程或进程组发送任意信号

- 示例
  - **/bin/kill -9 24818**

    向进程 24818发送信号SIGKILL

  - **/bin/kill -9 -24817**

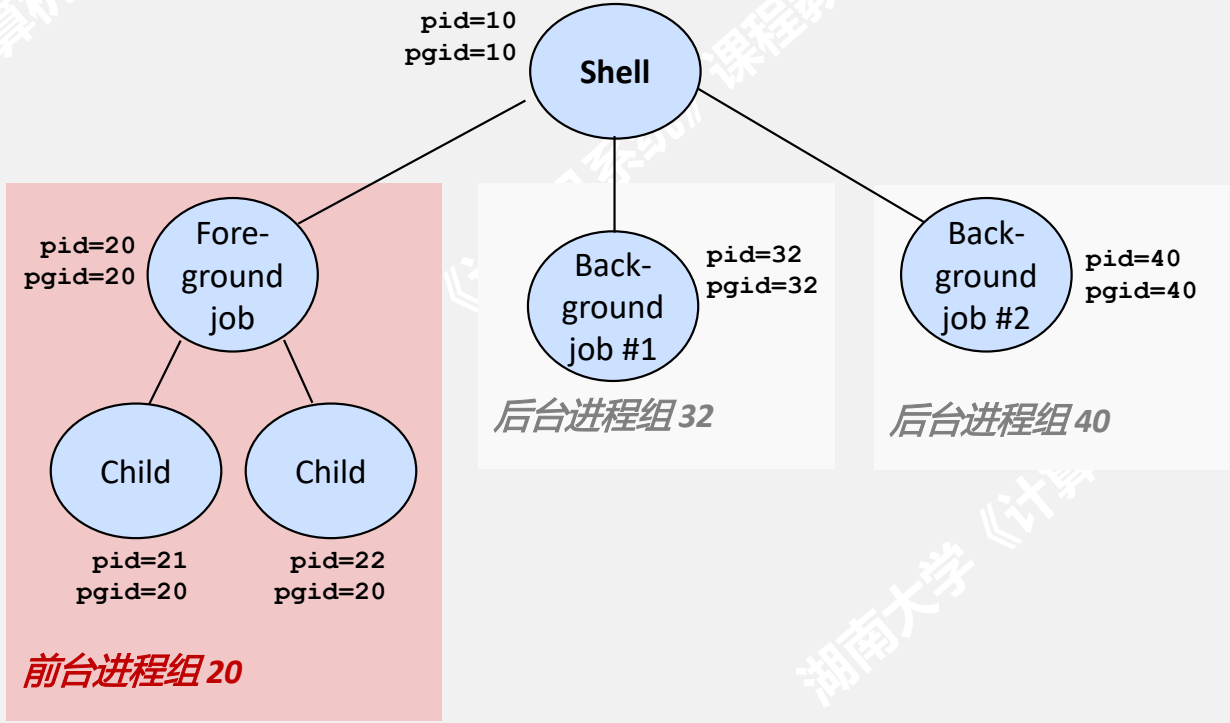
    发送信号SIGKILL给进程组24817中的每一个进程

```
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>
```

信号发送方法二：键盘组合键

- 键盘输入 ctrl-c (ctrl-z) 会向前台进程组的每一个作业（job）发送信号 SIGINT (SIGTSTP)
  - SIGINT – 终止一个进程的缺省操作
  - SIGTSTP – 停止（挂起）一个进程的缺省操作



ctrl-c 与 ctrl-z 示例

```
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
```

STAT（进程状态）说明：

First letter:

S: sleeping
T: stopped
R: running

Second letter:

s: session leader
+: foreground proc group

使用“man ps”了解更多细节

信号发送方法三：用kill函数发送信号

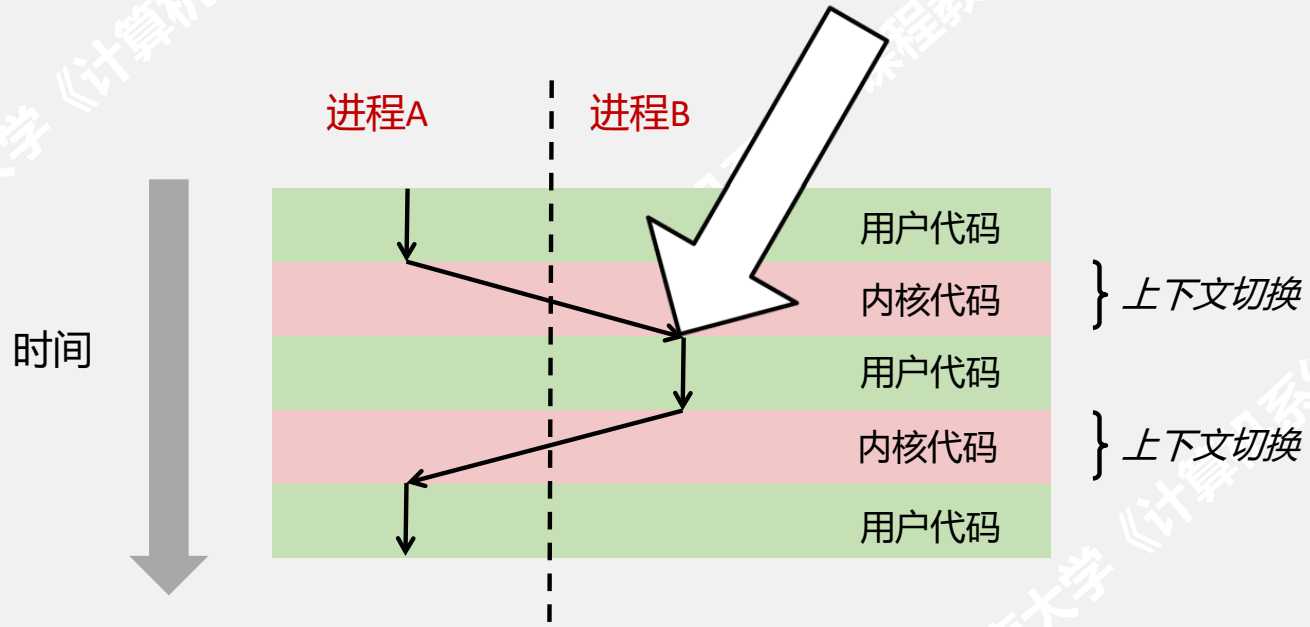
```
void fork12(){
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
       { while(1); }/* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    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);
    }
}
```

接收信号

- 场景：假设内核从异常处理程序返回并准备将控制权交给进程p



注意：所有的上下文切换都是通过调用一些异常处理程序启动的。

接收信号

- 当内核从一个异常处理程序返回，准备将控制传递给进程 *p* 时
- 内核计算 `pnb = pending & ~blocked`
  - 进程 *p* 的未阻塞的待处理信号的集合
- If (`pnb == 0`)
  - 控制传递回进程 *p* 的逻辑控制流的下一条指令
- Else
  - 选择 **pnb**中某个信号*k*(通常是最小的非零位*k*), 并强制进程 *p* 接收信号 *k*
  - 收到的信号触发进程 *p* 的某种行为
  - 对 **pnb** 中的所有非零 *k* 重复以上过程
  - 将控制传递回逻辑控制流中的下一个指令

默认行为

- 每个信号类型有一个预定义的默认行为, 是下面中的一种:
  - 进程终止 （**terminates)**
  - 进程终止并转储存储器 （**dumps core**）
  - 进程停止（**stops**）直到被信号 SIGCONT 重启
  - 进程忽略（**ignores** ）该信号

信号处理程序(Signal Handlers)

- signal函数改变与收到信号signum相关联的默认行为：
  - **handler_t *signal(int signum, handler_t *handler)**
- 对于不同的handler参数值:
  - SIG_IGN: 忽略类型为**signum的信号**
  - SIG_DFL: 类型为**signum**的信号行为恢复为默认行为
  - 否则, **handler** 就是用户定义的信号处理程序（*signal handler*）的函数地址
    - 当接收到一个类型为**signum**的信号，就会调用这个函数
    - 传递handler地址到signal函数被称为安装该处理程序（handler）
    - 执行该处理程序被称为捕获或处理该信号
    - 当处理程序执行它的return语句，控制通常传递回控制流中进程被信号接收中断位置处的指令
    - 返回：若成功则为指向前次处理程序的指针（最近一次调用该函数时第二个参数的值），若出错则为SIG_ERR

信号处理程序 示例

N=5

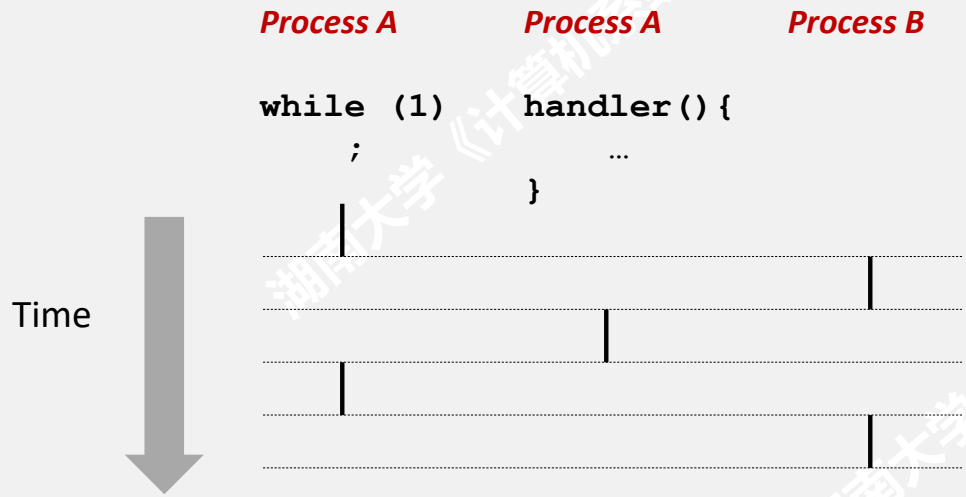
```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            while(1); /* child infinite loop */
        }
    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);
    }
}
```

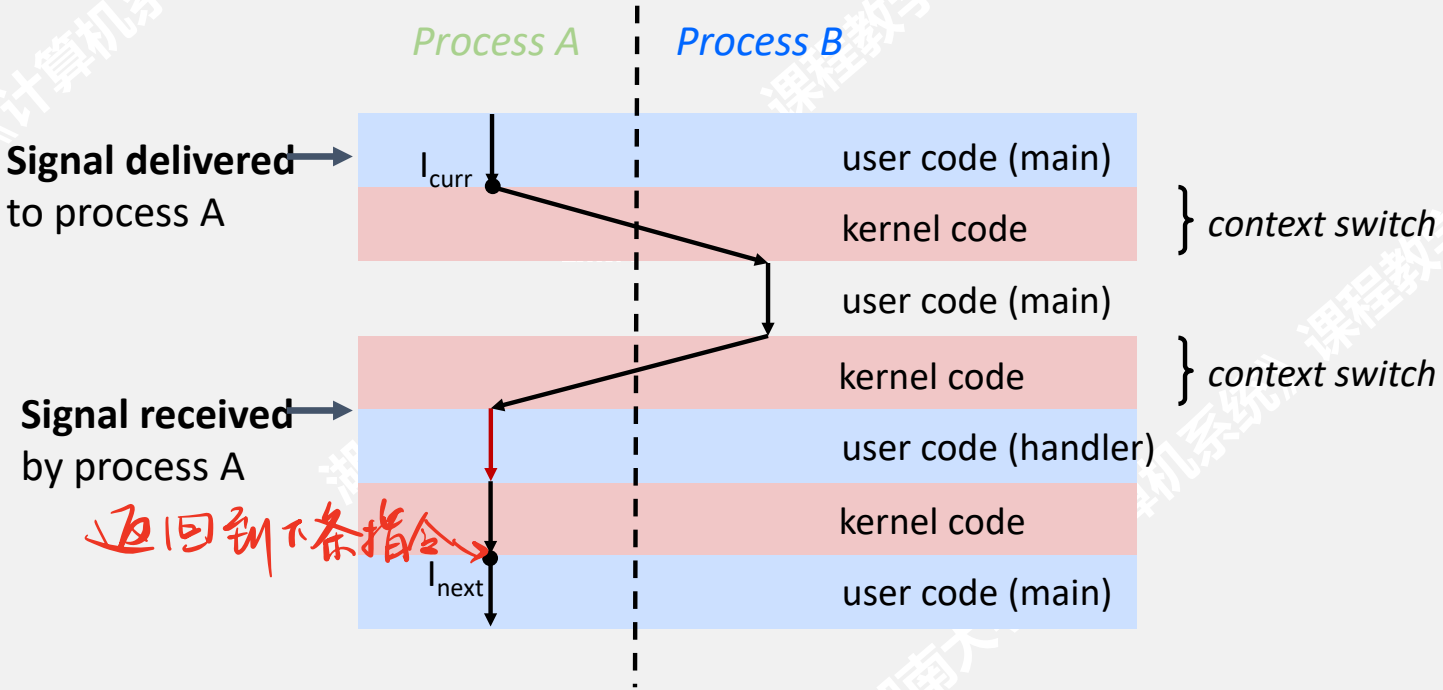
```
linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

信号处理程序作为并发流(Concurrent Flows)

- 信号处理程序是一个单独的逻辑流（非进程），与主程序并发执行
  - “并发”含“非顺序”的含义
  - “不连续”意义上的“同时”
  - 是“并发”，当不是“并行”



信号处理程序作为并发流的另一种视图



信号处理程序的问题

```
int ccount = 0;
void child_handler(int sig){
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    safe_printf(
           "Received signal %d from process %d\n",
           sig, pid);
}

void fork14(){
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0) {
        sleep(1); /* deschedule child */
        exit(0);  /* Child: Exit */
    }
    while (ccount > 0)
    pause(); /* Suspend until signal occurs */
}
```

N=5

- 正在处理的相同类型的待处理信号被阻塞
- 待处理信号不会排队等待
  - 对每一种信号类型，只有一个信号位标示是否待处理
  - 即使有多个进程已经发送了该类型信号
- forks14程序会正常结束吗？

```
linux> ./forks 14
Received SIGCHLD signal 17 for process 21344
Received SIGCHLD signal 17 for process 21345
```

重要教训：不可以用信号来对其他进程中发生的事件计数。

信号不排队的处理办法

- 必须检查所有已终止作业
  - 典型操作是循环使用waitpid（WNOHANG选项）

N=5

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        safe_printf("Received signal %d from process %d\n",
                    sig, pid);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
    while (ccount > 0) {pause();}
}
```

```
linux> forks 15
Received signal 17 from process 27476
Received signal 17 from process 27477
Received signal 17 from process 27478
Received signal 17 from process 27479
Received signal 17 from process 27480
linux>
```

更多信号处理程序的问题

- 在诸如 *read* 这类慢速系统调用中，信号到达
- 信号处理程序中断*read* 调用
  - Linux: 从信号处理程序返回， *read* 调用**自动**重新开始
  - 其它不同版本的Unix系统会使用一个 **EINTER**错误编码(**errno**) 引发 *read* 调用失败，此时应用程序会**自动或手动**重新启动慢速系统调用
- 这些微妙的差异使得使用信号的可移植代码的编写复杂化
  - 阅读教材可以获得更多细节8.5.4

响应外部生成事件的程序

- 外部事件 ——键盘组合 ctrl+c

external.c

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
  safe_printf("You think hitting ctrl-c will stop the bomb?\n");
  sleep(2);
  safe_printf("Well...");
  sleep(1);
  printf("OK\n");
  exit(0);
}

main() {
  signal(SIGINT, handler); /* installs ctrl-c handler */
  while(1) {
  }
}
```

```
linux> ./external
<ctrl-c>
You think hitting ctrl-c will stop the bomb?
Well...OK
linux>
```

响应内部生成事件的程序

- 内部事件——编程代码内部设定

internal.c

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  safe_printf("BEEP\n");

  if (++beeps < 5)
    alarm(1);
  else {
    safe_printf("BOOM!\n");
    exit(0);
  }
}
```

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in 1 second */

  while (1) {
    /* handler returns here */
  }
}
```

```
linux> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```

异步信号安全Async-Signal-Safety

- 如果函数是可重入（所有变量保存在栈帧中）或者信号不可中断，则函数是异步信号安全的
- Posix 确保 117个函数是异步信号安全的
  - **write** 函数包括在内，但不包括 **printf 函数**
- 解决方案:用于printf的异步信号安全包装函数

safe_printf.c

```
void safe_printf(const char *format, ...) {
    char buf[MAXS];
    va_list args;

    va_start(args, format);                          /* reentrant */
    vsnprintf(buf, sizeof(buf), format, args);       /* reentrant */
    va_end(args);                                    /* reentrant */
    write(1, buf, strlen(buf));                      /* async-signal-safe */
}
```

本讲内容

信号
signals

多任务与外壳
Multitasking & shells

非本地跳转
Nonlocal jumps

非本地跳转: setjmp/longjmp

- 强有力(但是危险)的用户级异常控制流机制，可将控制转移到任意位置
  - 无需经过 调用/返回 序列
  - 在错误恢复和信号处理时应用
- `int setjmp(jmp_buf j)` 设置环境
  - 必须在 longjmp 之前被调用
  - 为后续的longjmp定义了一个返回位置
  - **调用一次，返回一次或多次**
- 实现:
  - 在jmp_buf保存当前调用环境（寄存器上下文，栈指针和PC值）来记住的当前的位置
  - 供后面的longjump使用，第一次调用返回 0

setjmp/longjmp (续)

- `void longjmp(jmp_buf j, int i)`
  - 含义:
    - 从 缓冲区 *j* 中恢复调用环境
    - 触发一个从最近一次初始化 *j* 的**setjmp** 调用的返回
  - 在**setjmp**后被调用
  - **调用一次，但不返回**
- `longjmp` 实现：
  - 从缓冲区j中恢复寄存器上下文 (含栈指针、 基址指针、PC 值)
  - 设置寄存器 **%eax** (返回值) 为 **i**
  - 跳转到缓冲区 *j* 中保存的PC所指向的位置

setjmp/longjmp 示例

```
#include <setjmp.h>
jmp_buf buf;

main() {
   if (setjmp(buf) != 0) {
      printf("back in main due to an error\n");
   else
      printf("first time through\n");
   p1(); /* p1 calls p2, which calls p3 */
}
...
p3() {
   <error checking code>
   if (error)
      longjmp(buf, 1)
}
```

longjump函数的限制

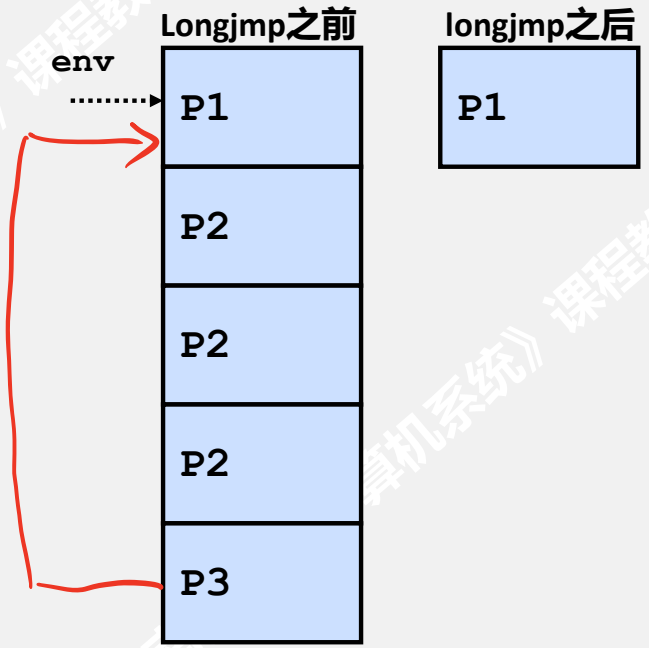
- 以堆栈规则序列工作
  - 只能long jump到已经被调用但尚未结束的函数环境

```
jmp_buf env;

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

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

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



longjump函数的限制 (续)

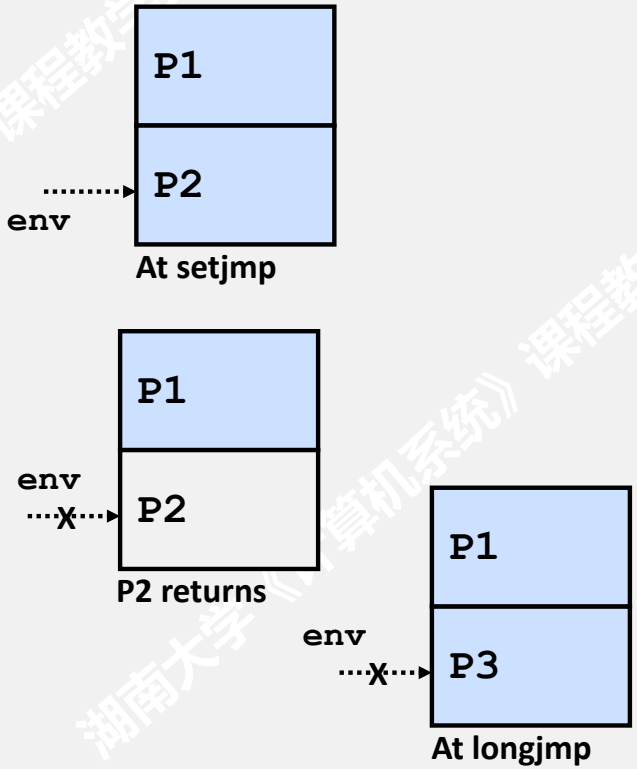
- 以堆栈规则序列工作
  - 只能long jump 到已经被调用但尚未结束的函数的环境

```
jmp_buf env;

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

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

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



以ctrl-c自重启一个程序（综合示例）

restart.c

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

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

main() {
  signal(SIGINT, handler);

  if (!sigsetjmp(buf, 1))
    printf("starting\n");
  else
    printf("restarting\n");

  while(1) {
     sleep(1);
     printf("processing...\n");
  }
}
```

```
linux> ./restart
starting
processing...
processing...
processing...
^Crestarting   <-- Ctrl-c
processing...
processing...
^Crestarting   <-- Ctrl-c
processing...
processing...
processing...
```

本讲小结

1. 信号提供了进程级的异常处理方式
   - 可以从用户程序生成
   - 可通过声明信号处理程序进行有效定义
2. 需特别注意
   - 开销很大
     - >10,000 时钟周期
     - 仅用于异常情况
   - 不能排队
     - 每种类型的信号只有**1 bit** 挂起置位
3. 非本地跳转提供了进程内的异常控制流
   - 受限于栈规则限制

计算机系统

下一讲将进入

虚存

湖南大学

《计算机系统》课程教学组