Technology Tarena 达内科技

Unix系统高级编程

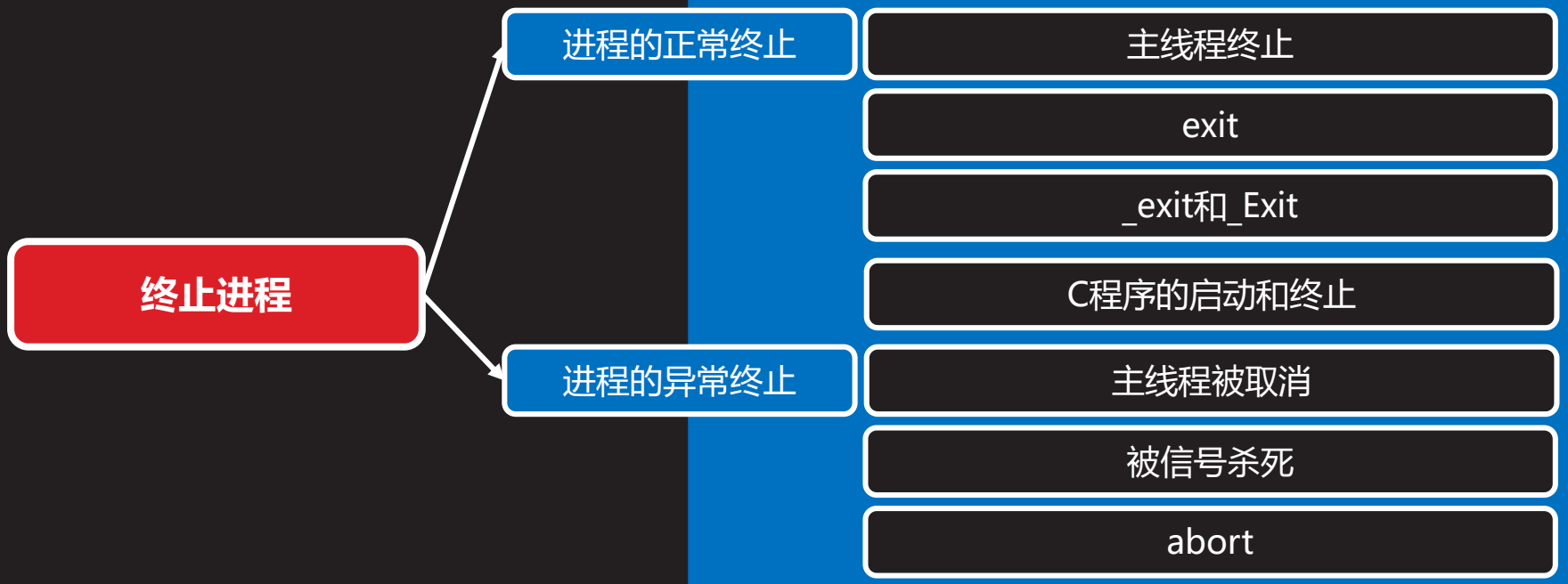
PART 2

DAY04

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	终止进程
	10:30 ~ 11:20	
	11:30 ~ 12:20	回收子进程
下午	14:00 ~ 14:50	
	15:00 ~ 15:50	创建新进程
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑

终止进程


终止进程
进程的正常终止
主线程终止
exit
_exit和_Exit
C程序的启动和终止
进程的异常终止
主线程被取消
被信号杀死
abort

Technology Tarena 达内科技

进程的正常终止

主线程终止

- 从main函数返回可令进程终止
  - 进程是内存中的代码和数据，而线程则是执行代码的过程
  - 每个进程可以包含一到多个线程，但至少要有一个主线程
  - 每个线程都可以被看做是在一个独立的执行过程中调用了一个特殊的函数，谓之线程过程函数。线程开始，线程过程函数即被调用，线程过程函数一旦返回，线程即告终止
  - 缺省情况下，进程主线程的线程过程函数就是main函数
  - main函数一旦返回，主线程即终止；主线程一旦终止，进程即终止；进程一旦终止，进程中的所有线程统统终止。这就是main函数的返回与其它函数的返回在本质上的区别
  - main函数的返回值即进程的退出码，父进程可以在回收子进程的同时获得该退出码，以了解导致其终止的具体原因

主线程终止（续1）

- 除了从main返回以外，在main函数或被main函数直接或间接调用的其它函数中，调用pthread_exit函数，亦可令主线程终止，进而导致进程终止
- 进程一旦终止，被终止进程在用户空间所持有的资源会被自动释放，如代码区、数据区、堆栈区等，但内核空间中与该进程相关的资源，如进程表项、文件描述符等，未必会立即得到释放

知识讲解

exit

- 正常终止调用进程

```
#include <stdlib.h>

void exit (int status);
```

不返回

  - ***status***：进程退出码，相当于main函数的返回值。被终止进程的父进程可以通过wait或waitpid函数，获取***status***参数的低8位，以了解导致该子进程终止的具体原因

- 例如
  - if (! (fp = fopen ("none", "r"))) exit (1);
  - if (! (buf = malloc (1024000))) exit (2);

知识讲解

exit（续1）

- 与通过return语句终止进程不同，只有main函数里的return语句才会终止进程，但exit函数在任何地方调用，包括main函数，被main函数直接或间接调用的其它函数，一般线程过程函数，甚至信号处理函数等等，都会立即令调用进程终止
- exit函数在终止调用进程之前还会做三件收尾工作
  1. 调用事先通过atexit或on_exit函数注册的退出处理函数
  2. 冲刷并关闭所有仍处于打开状态的标准I/O流
  3. 删除所有通过tmpfile函数创建的临时文件
- 事实上main函数里的return语句也会被编译器处理为类似对exit函数的调用，可以认为return x等价于exit (x)

知识讲解

exit（续2）

- 注册退出处理函数

```
#include <stdlib.h>

int atexit (void (*function) (void));
```

成功返回0，失败返回非0

  - ***function***：退出处理函数指针，所指向的函数既无返回值亦无参数，在进程终止前被调用，为进程料理临终事宜

- 注意atexit函数本身并不调用退出处理函数，而只是将***function***参数所表示的退出处理函数地址，保存(注册)在系统内核的某个地方(进程表项)。待到exit函数被调用或在main函数里执行return语句时，再由系统内核根据这些退出处理函数的地址来调用它们。此过程亦称回调

知识讲解

exit (续3)

- 例如
  - void* buf = NULL;
    void doexit (void) {
        if (buf) {
            free (buf);
            buf = NULL; }
    }
  - int main (void) {
        buf = malloc (1024);
        atexit (doexit);
        ...
        return 0;
    }

知识讲解

exit（续4）

- 注册退出处理函数

```
#include <stdlib.h>

int on_exit (void (*function) (int , void*), void* arg);
```

成功返回0，失败返回非0

知识讲解

- ***function***：退出处理函数指针，所指向的函数无返回值但有两个参数。其中第一个参数来自传递给exit函数的***status***参数或在main函数里执行return语句的返回值，而第二个参数则来自传递给on_exit函数的***arg***参数。该函数在进程终止前被调用，为进程料理临终事宜
- ***arg***：泛型指针，将作为第二个参数传递给***function***所指向的退出处理函数

exit (续5)

- 例如
  - void doexit (int status, void* arg) {
    printf ("%d\n", status); // 0
    if (arg)
    free (arg);
    }
  - int main (void) {
    void* buf = malloc (1024);
    on_exit (doexit, buf);
    ...
    return 0;
    }

知识讲解

exit（续6）

- 习惯上，还经常使用EXIT_SUCCESS和EXIT_FAILURE两个宏作为调用exit函数的参数，分别表示成功和失败。它们的值在多数系统中被定义成0和1，但一般建议使用宏，这样做兼容性更好
- 在exit函数的内部调用了更底层的系统调用函数_exit，后者也可以被用户空间的代码直接调用，比如在用vfork函数创建的子进程里

知识讲解

_exit和_Exit

- 正常终止调用进程

```
#include <unistd.h>

void _exit (int status);
```

不返回

  - ***status***：进程退出码，相当于main函数的返回值。被终止进程的父进程可以通过wait或waitpid函数，获取***status***参数的低8位，以了解导致该子进程终止的具体原因

- 例如
  - if (! (fp = fopen ("none", "r"))) _exit (1);
  - if (! (buf = malloc (1024000))) _exit (2);

知识讲解

_exit和_Exit（续1）

- _exit在终止调用进程之前也会做三件收尾工作，但与exit函数所做的不同。事实上，exit函数在做完它那三件收尾工作之后紧接着就会调用_exit函数
  1. 关闭所有仍处于打开状态的文件描述符
  2. 将调用进程的所有子进程托付给init进程收养
  3. 向调用进程的父进程发送SIGCHLD(17)信号
- 标准库中有一个与_exit完全等价的函数，可跨平台使用

```
#include <stdlib.h>

void _Exit (int status);
```

不返回

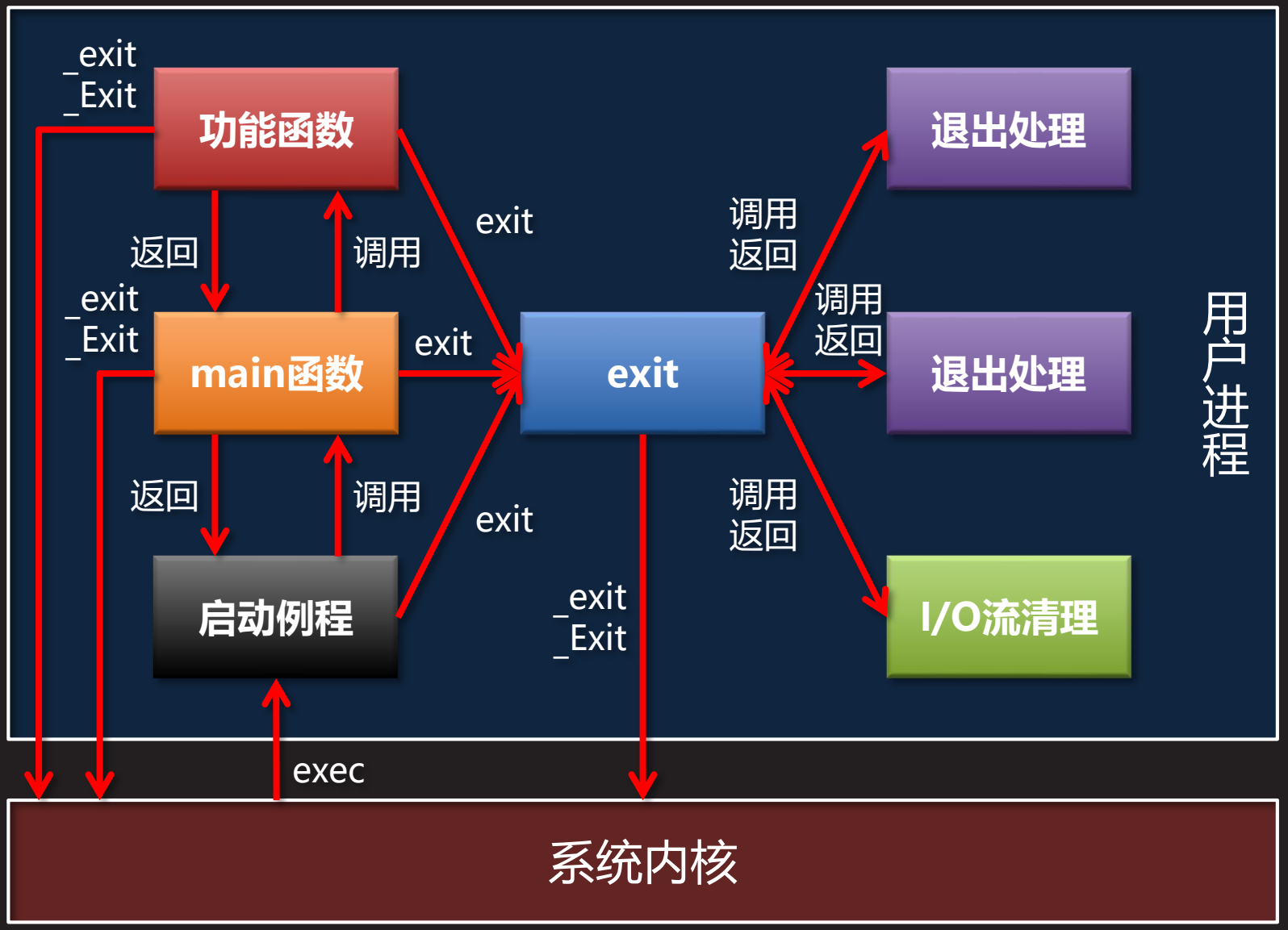
知识讲解

_exit和_Exit（续2）

- 从_exit和_Exit函数所做的三件收尾工作可以看出，它们所影响完全都是调用进程自己的资源，与其父进程没有任何关系。这对用vfork函数创建的子进程而言显得尤其重要。因为用vfork函数创建的子进程存在太多与其父进程共享的资源。如果在子进程里调用了exit函数或者在子进程的main函数里执行了return语句(这二者在本质上是等价的)，不仅销毁了子进程的资源，同时也销毁了父进程的资源，而父进程恰恰要在子进程终止以后从挂起中恢复运行，其后果可想而知

知识讲解

C程序的启动和终止



知识讲解

进程的正常终止

【参见：TTS COOKBOOK 】

课堂练习

- 进程的正常终止

Technology Tarena 达内科技

进程的异常终止

主线程被取消

- 在主线程外部，通过pthread_cancel函数可将主线程取消，进程随即终止
- 通过取消主线程令进程异常终止，往往不是立竿见影的
  - 不是所有的线程都能被取消。主线程可以忽略取消请求
  - 不是所有的线程都能立即响应取消请求。主线程可以选择采用延迟取消，即被取消线程在接收到取消请求之后并不立即做出响应，而是一直等到执行了某个特定的函数(取消点)之后才会响应该请求

知识讲解

被信号杀死

- 当进程执行了某些在系统看来具有危险性的操作，或系统本身发生了某种故障或意外，内核会向相关进程发送特定的信号。如果进程无意针对所收到的信号采取补救或预防措施，那么内核将按缺省方式将进程杀死，并视情形生成核心转储文件(core)以备事后分析，俗称吐核
  - SIGILL (4)：进程试图执行非法指令
  - SIGBUS (7)：硬件或对齐错误
  - SIGFPE (8)：算术异常
  - SIGSEGV (11)：无效内存访问
  - SIGPIPE (13)：向无读取进程的管道写入
  - SIGSTKFLT (16)：协处理器栈错误
  - SIGXFSZ (25)：文件资源超限
  - SIGPWR (30)：断电
  - SIGSYS (31)：进程试图执行无效系统调用

知识讲解

被信号杀死

- 除了系统内核产生的信号以外，也可以人为触发某些特殊的信号，将进程杀死
  - SIGINT (2)：用户产生中断符(Ctrl+C)
  - SIGQUIT (3)：用户产生退出符(Ctrl+\)
  - SIGKILL (9)：不能被捕获或忽略的进程终止信号
  - SIGTERM (15)：可以被捕获或忽略的进程终止信号

知识讲解

abort

- 异常终止调用进程

```
#include <stdlib.h>

void abort (void);
```

不返回

知识讲解

- abort函数首先解除调用进程对SIGABRT(6)信号的阻塞，然后向调用进程自身发送该信号。其结果就是令调用进程异常终止
- 即使调用进程忽略或者捕获SIGABRT(6)信号，abort函数依然可以令调用进程异常终止，因为它会先恢复对SIGABRT(6)信号按默认方式处理，然后再发一次该信号

回收子进程

wait

wait

- 等待并回收任意子进程

```
#include <sys/wait.h>

pid_t wait (int* status);

                成功返回所回收子进程的PID，失败返回-1
```

  - ***status***：输出子进程的终止状态，可置NULL
- 父进程在创建若干子进程以后调用wait函数
  - 若所有子进程都在运行，则阻塞，直至有子进程终止
  - 若有一个子进程已终止，则返回该子进程的PID并通过***status***参数(若非NULL)输出其终止状态
  - 若没有需要等待的子进程，则返回-1，置errno为ECHILD

wait（续1）

- 在任何一个子进程终止前，wait函数只能阻塞调用进程，而waitpid函数可以处理得更灵活一些
- 如果有一个子进程在wait函数被调用之前，已经终止并处于僵尸状态，wait函数会立即返回，并取得该子进程的终止状态，同时子进程僵尸消失。由此可见wait函数主要完成三个任务
  1. 阻塞父进程的运行，直到子进程终止再继续，停等同步
  2. 获取子进程的PID和终止状态，令父进程得知谁因何而死
  3. 为子进程收尸，防止大量僵尸进程耗费系统资源
- 以上三个任务中，即使前两个与具体需求无关，仅仅第三个也足以凸显wait函数的重要性，尤其是对那些多进程服务器型的应用而言

知识讲解

wait（续2）

- 子进程的终止状态通过wait函数的***status***参数输出给该函数调用者。<sys/wait.h>头文件提供了几个辅助分析进程终止状态的工具宏
  - WIFEXITED(***status***)
    判断子进程是否正常终止，是则通过WEXITSTATUS(***status***)宏获取子进程调用exit、_exit或者_Exit函数时所传入的参数或者main函数中return语句返回值的低8位，因此传递给这三个函数的参数和main函数的返回值最好都不要超过[-128, 127]
  - WIFSIGNALED(***status***)
    判断子进程是否异常终止，是则通过WTERMSIG(***status***)宏获取导致子进程异常终止的信号

知识讲解

wait（续3）

- 例如
  - 等待并回收一个子进程，打印其终止状态

```
int status;
pid_t pid = wait (&status);
if (pid == -1) {
    perror ("wait");
    exit (EXIT_FAILURE); }
if (WIFEXITED (status))
    printf ("%d子进程正常终止，退出码%d\n",
        pid, WEXITSTATUS (status));
else
    printf ("%d子进程异常终止，终止信号%d\n",
        pid, WTERMSIG (status));
```

知识讲解

wait（续4）

- 例如
  - 等待并回收所有子进程，忽略其终止状态

```
for (;;) {
    pid_t pid = wait (NULL);
    if (pid == -1) {
        if (errno != ECHILD) {
            perror ("wait");
            exit (EXIT_FAILURE); }
        printf ("子进程都死光了\n");
        break;
    }
    printf ("%d子进程终止\n", pid);
}
```

知识讲解

回收任意或所有子进程

【参见：TTS COOKBOOK 】

课堂练习

- 回收任意或所有子进程

waitpid

waitpid

- 等待并回收任意或特定子进程

```
#include <sys/wait.h>

pid_t waitpid (pid_t pid, int* status, int options);
```

成功返回所回收子进程的PID或0，失败返回-1

- *pid*：可取以下值
  - <-1 - 等待并回收特定进程组(由-*pid*标识)的任意子进程
  - -1 - 等待并回收任意子进程，相当于wait函数
  - 0 - 等待并回收与调用进程同进程组的任意子进程
  - > 0 - 等待并回收特定子进程(由*pid*标识)
- *status*：输出子进程的终止状态，可置NULL

知识讲解

waitpid（续1）

- 等待并回收任意或特定子进程
  - ***options***：可取以下值

    0 - 阻塞模式，若所等子进程仍在运行，则阻塞，直至其终止

    WNOHANG - 非阻塞模式，若所等子进程仍在运行，则返回0
- 事实上，无论一个进程是正常终止还是异常终止，都会通过系统内核向其父进程发送SIGCHLD(17)信号。父进程可以忽略该信号，也可以提供一个针对该信号的信号处理函数，在信号处理函数中以异步的方式回收子进程。这样做不仅流程简单，而且僵尸的存活时间短，回收效率高

知识讲解

waitpid（续2）

- 例如
  - 等待并回收所有子进程，等待的同时做空闲处理

```
for (;;) {
    pid_t pid = waitpid (-1, NULL, WNOHANG);
    if (pid == -1) {
        if (errno != ECHILD) {
            perror ("waitpid");
            exit (EXIT_FAILURE); }
        printf ("子进程都死光了\n");
        break; }
    if (pid) printf ("%d子进程终止\n", pid);
    else { 空闲处理; }
}
```

知识讲解

回收特定子进程

【参见：TTS COOKBOOK 】

课堂练习

- 回收特定子进程

非阻塞模式回收所有子进程

【参见：TTS COOKBOOK 】

课堂练习

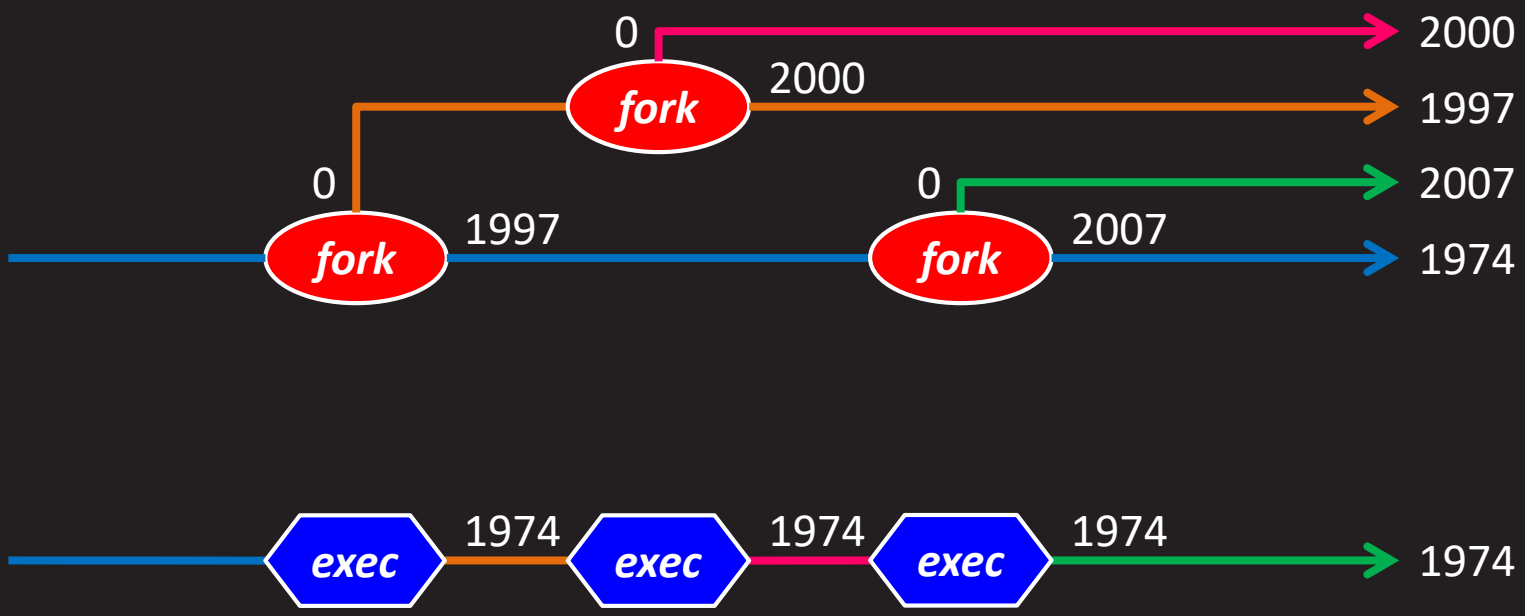
- 非阻塞模式回收所有子进程

创建新进程

exec

exec

- 与fork或vfork函数不同，exec函数不是创建调用进程的子进程，而是创建一个新的进程取代调用进程自身。新进程会用自己的全部地址空间，覆盖调用进程的地址空间，但进程的PID保持不变



exec（续1）

- exec不是一个函数而是一堆函数(共6个)，一般称为exec函数族。它们的功能是一样的，用法也很相近，只是参数的形式和数量略有不同

知识讲解

```
#include <unistd.h>

int execl (const char* path, const char* arg, ...);
int execlp (const char* file, const char* arg, ...);
int execle (const char* path, const char* arg, ...,
    char* const envp[]);
int execv (const char* path, char* const argv[]);
int execvp (const char* file, char* const argv[]);
int execve (const char* path, char* const argv[],
    char* const envp[]);
```

成功不返回，失败返回-1

exec（续2）

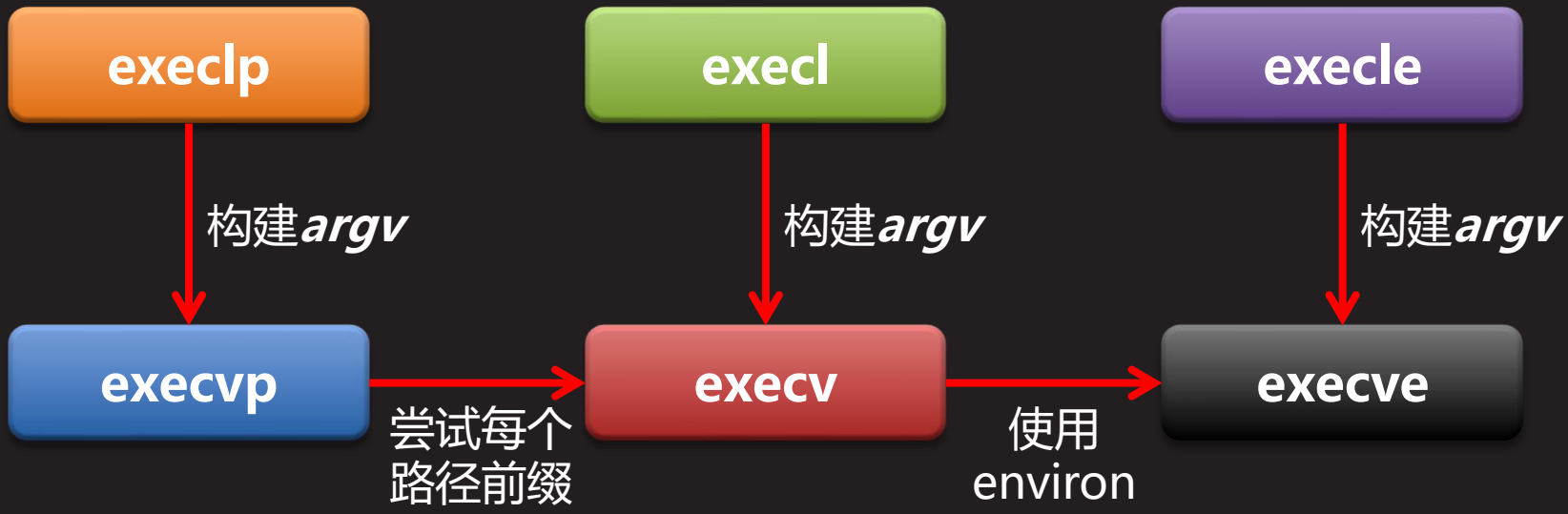
- exec函数族一共包括6个函数，它们的函数名都是在exec后面加上一到两个字符后缀，不同的字符后缀代表不同的含义
  - `l`：即list，新进程的命令行参数以字符指针列表(const char* arg, ...)的形式传入，列表以空指针结束
  - `p`：即path，若第一个参数中不包含“/”，则将其视为文件名，并根据PATH环境变量搜索该文件
  - `e`：即environment，新进程的环境变量以字符指针数组(char* const envp[])的形式传入，数组以空指针结束，不指定环境变量则从调用进程复制
  - `v`：即vector，新进程的命令行参数以字符指针数组(char* const argv[])的形式传入，数组以空指针结束

知识讲解

exec（续3）

- 其实6个exec函数中只有execve是真正的系统调用，其它5个函数不过是对execve函数的简单包装

知识讲解



exec（续4）

- 例如
  - 启动/bin/vi替换当前进程

    ```
    execl ("/bin/vi", "vi", NULL);
    ```

  - 用PATH环境变量作为搜索路径

    ```
    execlp ("vi", "vi", NULL);
    ```

  - 设置特殊的环境变量

    ```
    char* envp[] = {"HOME=/home/tarena", NULL};
    execle ("/bin/vi", "vi", NULL, envp);
    ```

  - 它们的v版本

    ```
    char* argv[] = {"vi", NULL};
    execv ("/bin/vi", argv);
    execvp ("vi", argv);
    execve ("/bin/vi", argv, envp);
    ```

知识讲解

exec（续5）

- 当在Shell下启动进程时，Shell会把路径的最后一个成分(通常是可执行程序的硬链接名)作为传给main函数的第一个命令行参数，即argv[0]。很多系统工具有不同的名称，其实它们不过是指向同一个程序的不同的硬链接。这些工具需要根据Shell传给main函数的第一个参数来判断用户使用的是哪个硬链接，以提供相应的功能。因此，调用exec函数时最好遵循Unix的习俗，用可执行程序的硬链接名作为新进程的第一个命令行参数，如上例中
  - execl ("/bin/vi", "vi", NULL);
  - execlp ("vi", "vi", NULL);
  - execle ("/bin/vi", "vi", NULL, envp);
  - char* argv[] = {"vi", NULL};

知识讲解

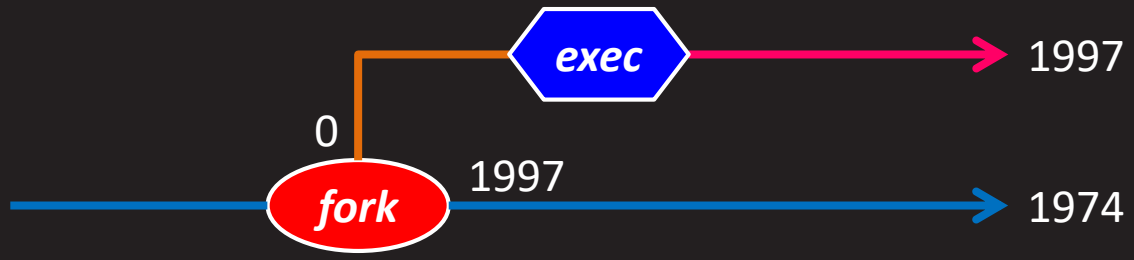
exec（续6）

- 调用exec函数不仅改变调用进程的地址空间和进程映像，调用进程的一些属性也发生了变化
  - 任何处于阻塞状态的信号都会丢失
  - 被设置为捕获的信号会还原为默认操作
  - 有关线程属性的设置会还原为缺省值
  - 有关进程的统计信息会复位
  - 与进程内存相关的任何数据都会丢失，包括内存映射文件
  - 标准库在用户空间维护的一切数据结构(如通过atexit或on_exit函数注册的退出处理函数)都会丢失
- 但也有些属性会被新进程继承下来，比如PID、PPID、实际用户ID和实际组ID、优先级，以及文件描述符等

知识讲解

exec（续7）

- 注意如果进程创建成功，exec函数是不会返回的，因为成功的exec调用会以跳转到新进程的入口地址作为结束，而刚刚运行的代码是不会存在于新进程的地址空间中的。但如果进程创建失败，exec函数会返回-1
- 调用exec函数固然可以创建出新的进程，但是新进程会取代原来的进程。如果既想创建新的进程，同时又希望原来的进程继续存在，则可以考虑fork+exec模式，即在fork产生的子进程里调用exec函数，新进程取代了子进程，但父进程依然存在



知识讲解

exec（续8）

- 例如
  - pid_t pid = fork ();
    if (pid == -1) {
    perror ("fork"); exit (EXIT_FAILURE); }
    if (pid == 0)
    if (execl ("ls", "ls", "-l", NULL) == -1) {
    perror ("execl"); exit (EXIT_FAILURE); }
- 事实上，在这种场合使用vfork更合适一些
  - pid_t pid = vfork ();
    if (pid == -1) {
    perror ("vfork"); exit (EXIT_FAILURE); }
    if (pid == 0)
    if (execl ("ls", "ls", "-l", NULL) == -1) {
    perror ("execl"); _exit (EXIT_FAILURE); }

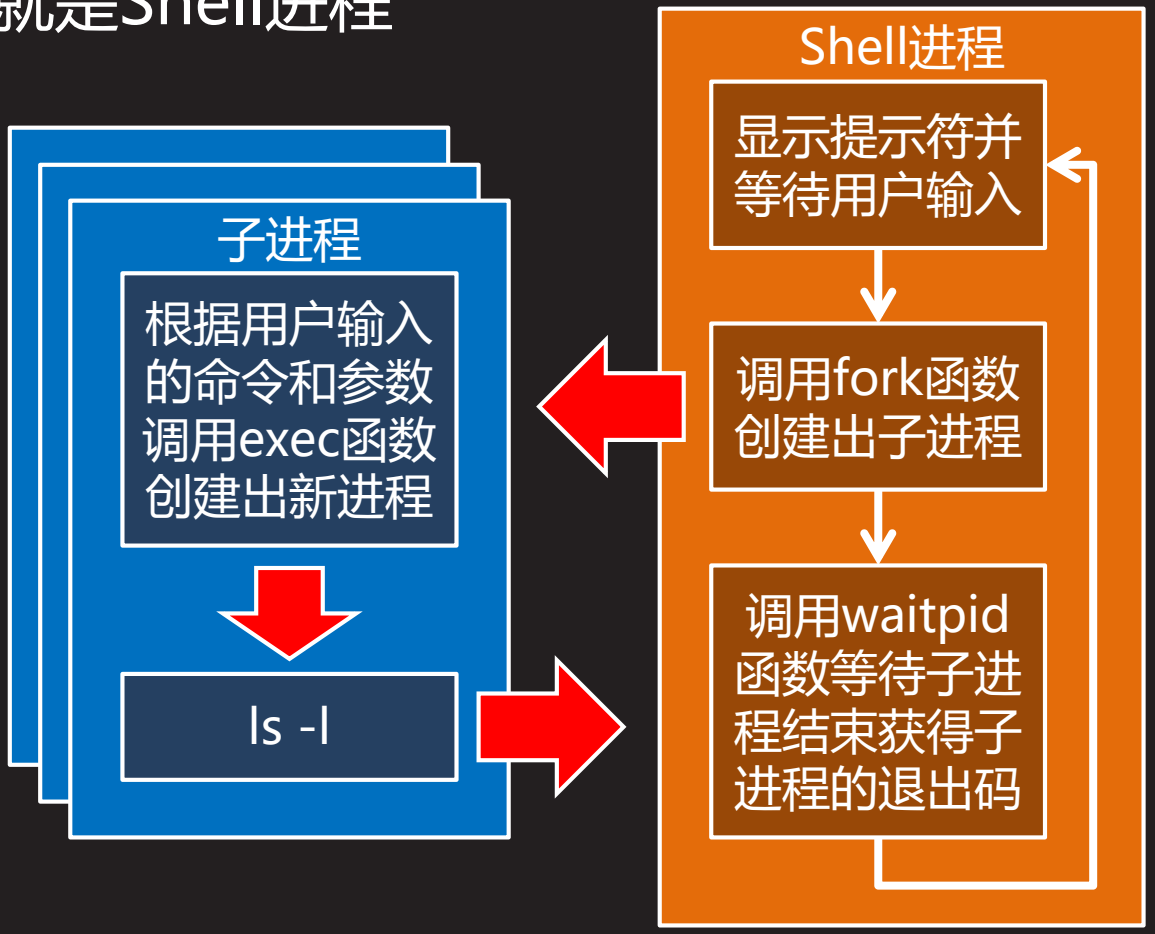
exec（续9）

- 如果原来的进程希望等待新进程结束以后再继续
  - pid_t pid = vfork ();
    if (pid == -1) {
    perror ("vfork"); exit (EXIT_FAILURE); }
    if (pid == 0)
    if (execl ("ls", "ls", "-l", NULL) == -1) {
    perror ("execl"); _exit (EXIT_FAILURE); }
    int status;
    if (waitpid (pid, &status, 0) == -1) {
    perror ("waitpid"); exit (EXIT_FAILURE); }
    if (WIFEXITED (status))
    printf ("%d进程正常终止，退出码%d\n",
    pid, WEXITSTATUS (status));
    else
    printf ("%d进程异常终止，终止信号%d\n",
    pid, WTERMSIG (status));

知识讲解

exec（续10）

- 如果一个进程可以根据用户的输入创建出不同的进程，并在所建进程结束以后继续重复这个过程，那么这个进程其实就是Shell进程



知识讲解

创建新进程

【参见：TTS COOKBOOK 】

课堂练习

- 创建新进程

Technology Tarena 达内科技

system

system

- 执行Shell命令

```
#include <stdlib.h>

int system (const char* command);

                成功返回command进程的终止状态，失败返回-1
```

  - *command*：Shell命令行字符串
- system函数执行*command*参数所表示的命令行，并返回命令进程的终止状态
- 若*command*参数取NULL，返回非0表示Shell可用，返回0表示Shell不可用

知识讲解

system（续1）

- 在system函数内部调用了vfork、exec和waitpid等函数
  - 如果调用vfork或waitpid函数出错，则返回-1
  - 如果调用exec函数出错，则在子进程中执行exit(127)
  - 如果都成功，则返回***command***进程的终止状态(由waitpid的***status***参数获得)
- 使用system函数而不用vfork+exec的好处是，system函数针对各种错误和信号都做了必要的处理，而且system是标准库函数，可跨平台使用

知识讲解

system（续2）

- 例如
  - int status;
    if ((status = system (NULL)) == -1) {
        perror ("system"); exit (EXIT_FAILURE); }
    if (! status)
        printf ("Shell不可用\n");
    else {
        if ((status = system ("ls -l")) == -1) {
            perror ("system"); exit (EXIT_FAILURE); }
        printf ("%d\n", WEXITSTATUS (status));
    }

知识讲解

执行命令

【参见：TTS COOKBOOK 】

课堂练习

- 执行命令

简化实现

【参见：TTS COOKBOOK 】

课堂练习

- 简化实现

Technology Tarena 达内科技

总结和答疑