创建、撤销、阻塞、唤醒……

进程控制

下面我们介绍一下进程控制

0:00 / 14:18

进程控制

进程控制操作完成进程各状态之间的转换，由具有特定功能的原语完成

- 进程创建原语
- 进程撤消原语
- 阻塞原语
- 唤醒原语
- 挂起原语
- 激活原语
- 改变进程优先级
- .....

原语（primitive）

完成某种特定功能的一段程序，具有不

分割性或不可中断性

即原语的执行必须是连续的，在执行过

中不允许被中断

所谓原语，有的时候又称之为原子操作 那么它是完成某种特定功能的一段程序

进程控制

进程控制操作完成进程各状态之间的转换，由具有特定功能的原语完成

- 进程创建原语
- 进程撤消原语
- 阻塞原语
- 唤醒原语
- 挂起原语
- 激活原语
- 改变进程优先级
- ……

原语（primitive）

完成某种特定功能的一段程序，具有不可分割性或不可中断性

即原语的执行必须是连续的，在执行过程中不允许被中断

原子操作（atomi

1.进程的创建

- 给新进程分配一个唯一标识以及进程控制块
- 为进程分配地址空间
- 初始化进程控制块
  - 设置默认值 (如: 状态为 New，...)
- 设置相应的队列指针

  如: 把新进程加到就绪队列链表中

UNIX：fork/exec

WINDOWS：CreateProcess

UNIX 里头呢，进程创建的主要操作是 fork
和 exec 的一个配合使用，在 WINDOWS

2.进程的撤消

结束进程

- 收回进程所占有的资源
  - 关闭打开的文件、断开网络连接、回收分配的内存、...
- 撤消该进程的PCB

UNIX： exit
WINDOWS：
TerminateProcess

里调用了 TerminateProcess，那么这个进程呢就撤消了
进程的阻塞，这个操作

3.进程阻塞

处于运行状态的进程，在其运行过程中期待某一事件发生，如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息，当被等待的事件未发生时，由进程自己执行阻塞原语，使自己由运行态变为阻塞态

UNIX：wait

WINDOWS：WaitForSingleObject

WaitForSingleObject 等，啊，操作函数 完成这项工作。

4.UNIX的几个进程控制操作

- **fork()** 通过复制调用进程来建立新的进程，是最基本的进程建立过程
- **exec()** 包括一系列系统调用，它们都是通过用一段新的程序代码覆盖原来的地址空间，实现进程执行代码的转换
- **wait()** 提供初级进程同步操作，能使一个进程等待另外一个进程的结束
- **exit()** 用来终止一个进程的运行

系统调用

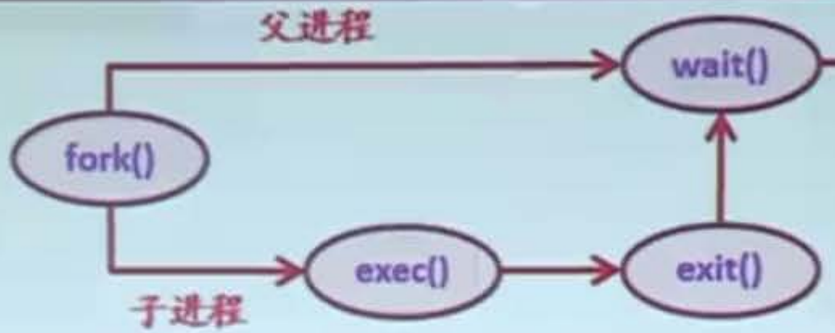
那么这是 UNIX 里头，最重要的几个进程控制操作 它们都是以系统调用的形式

UNIX的FORK()实现

- 为子进程分配一个空闲的进程描述符
  proc 结构
- 分配给子进程唯一标识 pid
- *以一次一页的方式复制父进程地址空间*
- 从父进程处继承共享资源，如打开的文件和当前工作目录等
- 将子进程的状态设为就绪，插入到就绪队列
- 对子进程返回标识符 0
- 向父进程返回子进程的 pid

Linux采用了写时复制技术COW加快创建进程
Copy-On-Write

使用FORK()的示例代码




```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
void main(int argc, char *argv[])
{
pid_t  pid;

①→  pid = fork();     /* 创建一个子进程 */
     if (pid < 0) {    /* 出错 */
         fprintf(stderr, "fork failed");
         exit(-1); }
     else if (pid == 0) {      /* 子进程 */
②→       execlp("/bin/ls", "ls", NULL); }
     else {                    /* 父进程 */
③→       wait(NULL);           /* 父进程等待子进程结束 */
         printf("child complete");
         exit(0);
}
}
```

那么子进程结束之后呢，父进程得到信号，然后父进程再做其它的工作

那么这就是这段代码的基本含义。

pid = xxx

```
pid = fork();
if (pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

PC

pid = 0

```
pid = fork();
if (pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

PC

父进程

pid = xxx

```
pid = fork();
if (pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

PC

子进程

pid = 0

```
pid = fork();
if (pid == 0) {
  printf("child");
} else {
  printf("parent");
}
```

PC

父进程执行父进程所需要的代码 子进程执行子进程所需要的代码

Acknowledgement: Prof. Yuanyuan Zhou at UCSD

0:00 下面我们介绍一下进程控制 进程控制操作呢，主要是完成了进程 之间，进程的各状态之间的什么呀，转换 那么进程控制操作实际上就是具有特定功能的程序 那么这个程序执行的时候呢，由于不允许被中断，所以呢，我们把它称之为原语 那么进程相关的控制原语，有这样一些 那么什么是原语呢？ 所谓原语，有的时候又称之为原子操作 那么它是完成某种特定功能的一段程序 比如说，完成创建，或者是完成阻塞，它是一段程序，完成了某种 特定功能，但是这个程序在执行过程中呢，是具有不可分割性，或者是不可中断的 它必须持续地执行，不允许被打断，这就是原语 当然实现原语，需要操作系统通过屏蔽中断 一些措施来达到这样一个结果 那么下面我们来看一些典型的进程控制操作 最重要的一个呢，是进程的创建 进程创建呢，主要完成以下几个工作 首先，给每一个新的进程分配一个标识，ID 再给它找一个空的、没有用过的进程控制块 然后要给这个进程分配它所需要的地址空间 当然这个地址空间，如果在虚拟存储机制之下 那么这个空间呢，就假设给了它，但是不真正，啊，给它内存，只是给了一个虚拟地址空间 下一步，是初始化 这个进程控制块，填写相应的内容 通常呢，都是设定一些默认值，比如说 状态，进程的状态设定为 New 等等 那么创建好了这个 进程控制块之后呢，要把它插入到相应的队列当中 所以呢，要设置相应的队列指针，比如说，它进入的是就绪队列 就把它，这个指针链，链好 在不同的操作系统当中，啊 提供了不同的进程创建操作，比如说 UNIX 里头呢，进程创建的主要操作是 fork 和 exec 的一个配合使用，在 WINDOWS 当中呢，是 CreateProcess 进程的撤销，进程的撤销实际上就是结束进程 结束进程其实主要做两件事情，第一件事情呢，是把这个进程所占有的资源回收 关闭它打开的文件 如果有网络连接就断开 如果分配了一些内存，就把它回收了 在做完这些资源回收之后 最重要的，是要把分配给它的 PCB 收回，这就是进程的撤销，通常呢，我们调用了 exit 或者在 WINDOWS 里调用了 TerminateProcess，那么这个进程呢就撤消了 进程的阻塞，这个操作 处于运行状态的进程，在其运行过程中，会期待、等待某个事件的发生 比如说，键盘，等待键盘的输入啊 或者是等待磁盘的数据，输，传输 或者是等待其他进程给它发来一些消息 在这些事件没有发生的情况下，那么进程需要自己执行一个阻塞原语 使自己的状态由运行态变为阻塞态 那么我们可以通过 UNIX 的 wait 或者是 WINDOWS 里头的 WaitForSingleObject 等，啊，操作函数 完成这项工作。 那么在 UNIX 里头，有几个非常重要的 进程控制操作，一个就是 fork fork 呢，实际上是创建新的进程，它的创建过程呢是通过复制调用进程本身 来创

建的，那么调用进程我们称之为父进程，也就是通过复制父进程，来创建子进程 这是一个非常基本的进程建立过程 那么 exec，这是一个系列的系统调用 它的主要目的 是通过用一段新的程序 来覆盖原来的地址空间，也就是 父进程，原来是把自己的所有内容 复制给子进程，那么子进程呢，用一些新的代码 程序代码，把父进程拷贝过来的内容，给它覆盖掉 通过这样一个覆盖，实现了进程的执行代码的一个转换 wait，实际上是一个初级的一个同步操作 它能使得一个进程等待另外一个进程的结束 那么 exit 刚才已经说过，它是用来终止一个进程的运行 那么这是 UNIX 里头，最重要的几个进程控制操作 它们都是以系统调用的形式 作为一个接口，呈现给用户程序，由用户程序来调用 下面我们来看看，UNIX 里头，fork 这个系统调用的实现，fork 在实现 过程中，首先会为子进程分配一个空闲的进程描述符 也就是 PCB，然后这个 PCB 呢，在 UNIX 里头我们一般叫 proc 结构 那么它给子进程也找，分配了一个唯一的标识 pid 下面就是给子进程分配地址空间了 那么，在 UNIX 里头，fork 怎么做的呢 它是以一次一页的方式，把父进程的 地址空间内容完全地拷贝给子进程 之后，再从父进程那里 继承各种共享资源，像打开的文件啊，还有工作，当前工作目录等等 那么都是从父进程那里继承下来，子进程的状态 设置为就绪态，并且把它插入到了就绪队列 做完这项工作之后，fork 就为子进程返回一个值 0 而为父进程返回一个值，是子进程的 pid 那么也就是说，fork 执行完后，原来一个进程，父进程就 一分为二，变成了两个进程，一个父进程，一个子进程，在父进程 在父进程的里头，得到的返回值是 pid，在子进程里头，得到的返回值是 0 那这里头呢，我们来看一下这个操作 这个步骤，以一次一页的方式来复制父进程的地址空间 那么这么做，有什么弊端呢？ 我们知道，父进程把它所有的内容都拷贝给子进程 但是子进程是否需要呢？父进程创建子进程 让它做什么事儿呢？通常情况下，父进程 创建子进程是让子进程做与父进程所不同的工作 如果是这样的话，那么你把你的所有内容拷贝给子进程，实际上，子进程也不需要 因此，子进程会接着执行 exec 这样一个函数 来把，用新的，啊，一段代码，来把父进程拷贝过来的这些地址空间给覆盖掉 那么因此，之前的这种复制工作，实际上就是无用功了 而且还花了很多的时间和空间，因此 那么，在 UNIX 里头实现 fork 花费的时间比较长 到了 Linux，Linux 要想改善这样一个调用的这个 实现，那么 Linux 怎么改善呢？ Linux 是这么做的 Linux 使用了一个技术，叫做写时复制技术 Copy-On-Write，那

么这个技术，是在存储管理 这个模块当中提供的一个支持 那么 Linux 就用了这样一个技术 用在了 Linux 的 fork 的实现过程中 那么，采用了这个技术之后 在 Linux 里的 fork 就加快了速度，那么为什么呢？我们来看一下 原来是要复制父进程的地址空间，而现在呢 只需要父进程把地址空间的指针传递给子进程 再把地址空间设置为只读 那么当子进程要往地址空间里写 东西的时候。写时复制，当要写东西的时候啊，譬如写代码啊，写一个值在里头的时候。 这个时候 被操作系统接受，然后呢 操作系统会为子进程单独再开辟一块空间，把相应的内容放进去 那么这样的话呢，节省了 之前复制父进程地址空间的时间，加快了 fork 的实现速度。 那么当 Linux 引入这个技术之后，实际上是 fork 的速度是非常快了 那么这就是 UNIX 的FORK 的实现，以及 Linux 对这段实现的一个改进 下面我们来看一下怎么样来使用 fork 那么我们看一下，这一段代码当中呢 在这里，这个进程呢，创建了一个子进程 那么把这个 fork 创建的子进程把这个 fork 的返回值呢，返回给一个整数 pid 里头 下面就来判断这个 pid 的值 如果 pid 小于 0，说明这个 fork 失败，因此报错 退出就好了。 如果 不小于 0，那么就要判断 pid 的值是 0 还是非 0 如果是 0，表示的是子进程的 代码空间，因此呢执行的是子进程要做的事情 如果不为 0，那么返回的就是子进程的 pid，说明是在 父进程的地址空间。 因此呢，就执行父进程所需要的代码 也就是说 fork 执行完之后，原来的一个进程分支，变成了两个进程 子进程要做的事情，就是用一段代码来覆盖 父进程的地址空间内容。 这个因为是 UNIX，所以它是覆盖过去 然后呢，去执行相应的代码，最后，执行完了以后退出。 而父进程呢 创建完子进程之后呢，就调用了一个 wait 这样一个函数，等待子进程的结束 那么子进程结束之后呢，父进程得到信号，然后父进程再做其它的工作 那么这就是这段代码的基本含义。 我们来看一下 从地址空间的角度，在创建完 进程之后，也就是执行完了这个 fork 父进程的地址空间和子进程的地址空间，这是两个空间 那么在父进程地址空间当中的这个 pid 变量拿到的是 子进程的 pid 号 xxx，啊 那么子进程的空间里头，这个 pid 这个单元得到的是一个 0，表示的这是子进程的空间 那么下一个指令从哪开始执行呢？ 两个空间里头，下一条指令都是从判断 pid 是否等于 0 开始执行，所以那边也是，这边也是 那么对于父进程 pid 因为不等于 0，所以呢父进程就进入到了 它所需要执行的代码段 而子进程由于判断出 pid 等于 0，它就执行这个代码段 我们来看一下，就是父进程就开始执行 print

"parent"，打印一个 "parent"，而子进程呢执行的是打印 "child" 的工作 那么这就是我们说的 fork 执行完之后 变成了一分为二，那么有两个地址空间 每个地址空间当中，这个 pid 的值是不一样的，因此 父进程执行父进程所需要的代码 子进程执行子进程所需要的代码