## 第3-1课 讲稿（I/O多路复用）

1. 实验1常见问题
2. 提交文档不符合要求（命名、文档类型、提交方式等——不仔细看说明）
3. 截图太大
4. 多路复用
5. 函数说明。

前面的fcntl()函数解决了文件的共享问题，接下来该处理I/O复用的情况了。

总的来说，I/O处理的模型有5种。

* 阻塞I/O模型：在这种模型下，若所调用的I/O函数没有完成相关的功能，则会使进程挂起，直到相关数据到达才会返回。对管道设备、终端设备和网络设备进行读写时经常会出现这种情况。
* 非阻塞模型：在这种模型下，当请求的I/O操作不能完成时，不让进程睡眠，而是立即返回。非阻塞I/O使用户可以调用不会阻塞的I/O操作，如open()、write()和read()。如果该操作不能完成，则会立即返回出错（例如：打不开文件）或者返回0（例如：在缓冲区中没有数据可以读取或者没有空间可以写入数据）。
* I/O多路转接模型：在这种模型下，如果请求的I/O操作阻塞，则它不是真正阻塞I/O，而是让其中的一个函数等待，在这期间，I/O还能进行其他操作。本次课要介绍的select()和poll函数()就是属于这种模型。
* 信号驱动I/O模型：在这种模型下，通过安装一个信号处理程序，系统可以自动捕获特定信号的到来，从而启动I/O。这种模型是由内核通知用户何时可以启动一个I/O操作。
* 异步I/O模型：在这种模型下，当一个描述符已准备好，可以启动I/O时，进程会通知内核。

对于I/O复用，可以使用select()和poll()的I/O多路转接模型处理，它可以具体设置程序中每一个所关心的文件描述符的条件、希望等待的时间等，从select()和poll()函数返回时，内核会通知用户已准备好的文件描述符的数量、已准备好的条件等。通过使用select()和poll()函数的返回结果，就可以调用相应的I/O处理函数。

select()函数的作用是在规定的时间内阻塞进程，轮询某个文件描述符集中的文件是否准备好（可读、可写或出错），如果文件没有准备好，则select继续阻塞进程并持续轮询所有文件；如果有文件做好了准备，则返回准备好的文件数量，并解除阻塞。

select()也可以实现精确计时，计时精度可达到微秒（百万分之一秒）级。

select()函数适用于网络编程。

1. 函数格式。

select()函数的语法格式如表1所示。

|  |  |  |
| --- | --- | --- |
| 表1 select()函数的语法格式 | | |
| 所需头文件 | #include <sys/types.h>  #include <sys/time.h>  #include <unistd.h> | |
| 函数原型 | int select(int numfds, fd\_set \*readfds, fd\_set \*writefds，  fd\_set \*exceptfds, struct timeval \*timeout) | |
| 函数传入值 | numfds：需要监视的文件的个数，即需要监视的文件描述符的最大值加1。  因为select()函数遍历文件列表中[0，numfds)的内容，所以只有numfds的值比文件描述符的最大值多1，才能遍历所有的文件。 | |
| readfds：由select()监视的读文件描述符集合 | |
| writefds：由select()监视的写文件描述符集合 | |
| exeptfds：由select()监视的异常处理文件描述符集合 | |
| timeout | NULL：永远等待，直到捕捉到信号或文件描述符已准备好为止 |
| 具体值：struct timeval类型的指针，若等待了timeout时间还没有检测到任何文件描符准备好，就立即返回 |
| 0：从不等待，测试所有指定的描述符并立即返回 |
| 函数返回值 | 大于0：成功，表示有文件已经准备好（可读、可写或出错），返回准备好的文件描述符的数目  0：超时；  -1：出错 | |

|  |
| --- |
| 【思考】  如何确定被监视的文件描述符的最大值？ |

可以看到，select()函数根据希望进行的文件操作对文件描述符进行了分类处理，这里，对文件描述符的处理主要涉及4个宏函数，如表2所示。

|  |  |
| --- | --- |
| 表2 对文件描述符处理的4个宏函数 | |
| FD\_ZERO(fd\_set \*set) | 清除一个文件描述符集 |
| FD\_SET(int fd, fd\_set \*set) | 将一个文件描述符加入文件描述符集中 |
| FD\_CLR(int fd, fd\_set \*set) | 将一个文件描述符从文件描述符集中清除 |
| FD\_ISSET(int fd, fd\_set \*set) | 判断文件描述符fd是否在文件描述符集fd\_set里。如果文件描述符fd为fd\_set集中的一个元素，则返回非零值，否则返回0。  可以用于调用select()之后测试文件描述符集中的文件描述符是否有变化，因为select()轮询时会把没有准备好的fd剔除。 |

一般来说，在使用select()函数之前，首先使用FD\_ZERO()和FD\_SET()来初始化文件描述符集，在使用了select()函数时，可循环使用FD\_ISSET()来测试描述符集，在执行完对相关文件描述符的操作之后，使用FD\_CLR()来清除描述符集。

另外，select()函数中的timeout是一个struct timeval类型的指针，该结构体如下所示：

|  |
| --- |
| struct timeval  {  long tv\_sec; /\* 秒 \*/  long tv\_usec; /\* 微秒 \*/  } |

可以看到，这个时间结构体的精确度可以设置到微秒级，这对于大多数的应用而言已经足够了。

1. 实例：

由于多路复用通常用于I/O操作可能会被阻塞的情况，而对于可能会有阻塞I/O的管道、网络编程，到现在为止还没有涉及，所以这里通过手动创建两个管道文件（用mknod或者mkfifo命令），重点说明如何使用多路复用函数。

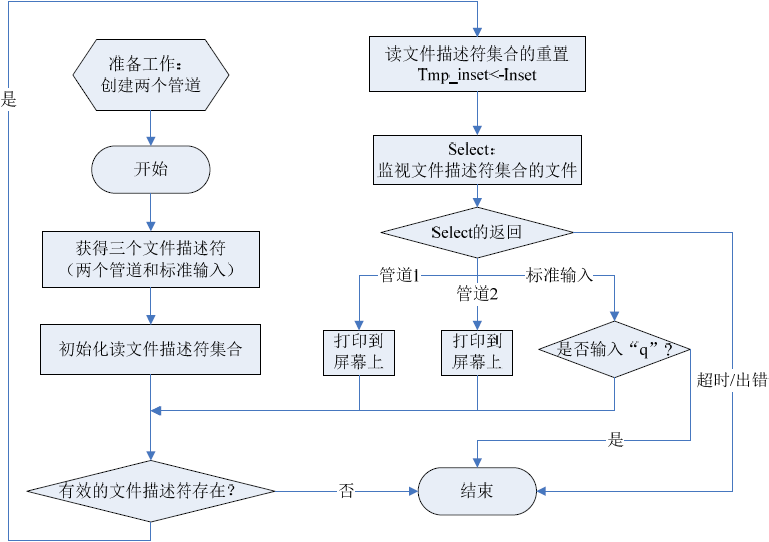
|  |  |
| --- | --- |
| 【补充】管道文件：  管道是Linux中很重要的一种通信方式，是把一个程序的输出直接连接到另一个程序的输入。无名管道和有名管道。  在 Linux中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的file结构体和VFS的索引节点inode。通过将两个 file结构体指向同一个临时的 VFS索引节点，而这个 VFS索引节点又指向一个物理页面而实现的。   |  | | --- | | 【注意】inode和文件描述符   * inode或i节点是指对文件的索引。如一个系统，所有文件是放在磁盘或flash上，就要编个目录来说明每个文件在什么地方，有什么属性，及大小等，就像书本的目录一样，便于查找和管理。 * 文件描述符：在linux中，内核通过inode来找到每个文件，但一个文件可以被许多用户同时打开或一个用户同时打开多次。这就有一个问题，如何管理文件的当前位移量，因为可能每个用户打开文件后进行的操作都不一样，这样文件位移量也不同，当然还有其他的一些问题。所以linux又搞了一个文件描述符（File Descriptor）这个东西，来分别为每一个用户服务。每个用户每次打开一个文件，就产生一个文件描述符，多次打开就产生多个文件描述符，一一对应，不管是同一个用户，还是多个用户。该文件描述符就记录了当前打开的文件的偏移量等数据。所以一个i节点可以有0个或多个文件描述符。多个文件描述符可以对应一个i节点。 |  1. 管道的读写   管道写函数通过将字节复制到 VFS 索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。当然，内核必须利用一定的机制同步对管道的访问，为此，内核使用了锁、等待队列和信号。  当写进程向管道中写入时，它利用标准的库函数write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构。file 结构中指定了用来进行写操作的函数（即写入函数）地址，于是，内核调用该函数完成写操作。写入函数在向内存中写入数据之前，必须首先检查 VFS 索引节点中的信息，同时满足如下条件时，才能进行实际的内存复制工作：   * 内存中有足够的空间可容纳所有要写入的数据； * 内存没有被读程序锁定。   如果同时满足上述条件，写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存。否则，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。写入进程实际处于可中断的等待状态，当内存中有足够的空间可以容纳写入数据，或内存被解锁时，读取进程会唤醒写入进程，这时，写入进程将接收到信号。当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒。  管道的读取过程和写入过程类似。但是，进程可以在没有数据或内存被锁定时立即返回错误信息，而不是阻塞该进程，这依赖于文件或管道的打开模式。反之，进程可以休眠在索引节点的等待队列中等待写入进程写入数据。当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放。  Linux 管道对阻塞之前一次写操作的大小有限制。专门为每个管道所使用的内核级缓冲区确切为 4096 字节。 除非阅读器清空管道，否则一次超过 4K 的写操作将被阻塞。实际上这算不上什么限制，因为读和写操作是在不同的线程中实现的。 |

在本实例中，用select()函数实现如下功能：

【功能描述】通过调用select()函数来监听3个终端的输入（分别重定向到两个管道文件的虚拟终端以及主程序所运行的虚拟终端），并分别进行相应的处理。在这里我们建立了一个select()函数监视的读文件描述符集，其中包含3个文件描述符，分别为一个标准输入文件描述符和两个管道文件描述符。通过监视主程序的虚拟终端标准输入来实现程序的控制，以两个管道作为数据输入，主程序将从两个管道读取的输入字符串写入到标准输出文件（屏幕）。

为了充分表现select()调用的功能，在运行主程序的时候，需要打开3个虚拟终端：首先用mknod命令创建两个管道in1 和in2。接下来，在两个虚拟终端上分别运行cat>in1和cat>in2，同时在第三个虚拟终端上运行主程序。在程序运行之后，如果在两个管道终端上输入字符串，则可以观察到同样的内容将在主程序的虚拟终端上逐行显示。如果想结束主程序，只要在主程序的虚拟终端下输入以‘q’或‘Q’字符开头的字符串即可。如果三个文件一直在无输入状态中，则主程序一直处于阻塞状态。为了防止无限期的阻塞，在select程序中设置超时值（本实例中设置为60s），当无输入状态持续到超时值时，主程序主动结束运行并退出。

该程序的流程图如下图所示。



使用select()函数实现的代码如下所示：

|  |
| --- |
| /\* multiplex\_select \*/  #include <fcntl.h>  #include <stdio.h>  #include <unistd.h>  #include <stdlib.h>  #include <string.h>  #include <time.h>  #include <errno.h>  #define MAX\_BUFFER\_SIZE 1024 /\* 缓冲区大小\*/  #define IN\_FILES 3 /\* 多路复用输入文件数目 \*/  #define TIME\_DELAY 60 /\* 超时值秒数 \*/  #define MAX(a, b) ((a > b)?(a):(b))  int main()  {  int fds[IN\_FILES];  char buf[MAX\_BUFFER\_SIZE];  int i, res, real\_read, maxfd;  struct timeval tv;  fd\_set inset,tmp\_inset;  /\*首先以只读非阻塞方式打开两个管道文件\*/  fds[0] = 0;  if((fds[1] = open ("in1", O\_RDONLY|O\_NONBLOCK)) < 0)  {  printf("Open in1 error\n");  return 1;  }  if((fds[2] = open ("in2", O\_RDONLY|O\_NONBLOCK) ) < 0)  {  printf("Open in2 error\n");  return 1;  }  /\*取出两个文件描述符中的较大者\*/  maxfd = MAX(MAX(fds[0], fds[1]), fds[2]) ;  /\*初始化读集合 inset，并在读集合中加入相应的描述集\*/  FD\_ZERO(&inset);  for (i = 0; i < IN\_FILES; i++)  {  FD\_SET(fds[i], &inset);  }  tv.tv\_sec = TIME\_DELAY;  tv.tv\_usec = 0;  /\*循环测试该文件描述符是否准备就绪，并调用 select 函数对相关文件描述符做对应操作\*/  while(FD\_ISSET(fds[0],&inset) && FD\_ISSET(fds[1],&inset) && FD\_ISSET(fds[2], &inset) )  {  /\* 文件描述符集合的备份， 这样可以避免每次进行初始化 \*/  tmp\_inset = inset;  res = select(maxfd + 1, &tmp\_inset, NULL, NULL, &tv);  switch(res)  {  case -1:  {  printf("Select error\n");  return 1;  }  break;  case 0: /\* Timeout \*/  {  printf("Time out\n");  return 1;  }  break;  default:  {  for (i = 0; i < IN\_FILES; i++)  {  if (FD\_ISSET(fds[i], &tmp\_inset) )  {  memset(buf, 0, MAX\_BUFFER\_SIZE);  real\_read = read(fds[i], buf, MAX\_BUFFER\_SIZE) ;  if (real\_read < 0)  {  if (errno != EAGAIN)  {  return 1;  }  }  else if (!real\_read)  {  close(fds[i]);  FD\_CLR(fds[i], &inset);  }  else  {  if (i == 0)  {/\* 主程序终端控制 \*/  if ((buf[0] == 'q') || (buf[0] == 'Q'))  {  return 1;  }  }  else  {/\* 显示管道输入字符串 \*/  buf[real\_read] = '\0';  printf("From file in%d(fd = %d): %s", i, fds[i], buf);  }  }  } /\* end of if \*/  } /\* end of for \*/  }  break;  } /\* end of switch \*/  } /\*end of while \*/  return 0;  } |

测试程序：

1. 创建管道文件

# mknod in1 p

# mknod in2 p

1. 在终端1中打开管道文件in1

# cat > in1

1. 在终端2中打开管道文件in2

# cat > in2

1. 在终端3中打开主程序

# ./multiplex\_select

1. 分别在in1和in2中输入数据，发现在in1和in2中输入的数据马上会在运行主程序的终端3中显示输出。

q /\* 在终端上输入‘q’或‘Q’则立刻结束程序运行 \*/

程序的超时结束结果如下：

# ./multiplex\_select

……

Time out

可以看到，使用select()可以很好地实现I/O 多路复用。

但是当使用select()函数时，存在一系列的问题，例如：内核必须检查多余的文件描述符，每次调用select()之后必须重置被监听的文件描述符集，而且可监听的文件个数受限制（使用FD\_SETSIZE 宏来表示fd\_set 结构能够容纳的文件描述符的最大数目）等。

除select()函数外，还有poll()函数，也可以实现I/O多路复用，此处仅给出poll的语法格式，不再举例说明。

poll()函数的语法格式如表3所示。

|  |  |
| --- | --- |
| 表3 poll()函数的语法格式 | |
| 所需头文件 | #include <sys/types.h>  #include <poll.h> |
| 函数原型 | int poll(struct pollfd \*fds, int numfds, int timeout) |
| 函数传入值 | fds：struct pollfd 结构的指针，用于描述需要对哪些文件的哪种类型的操作进行监控。  struct pollfd  {  int fd; /\* 需要监听的文件描述符 \*/  short events; /\* 需要监听的事件 \*/  short revents; /\* 已发生的事件 \*/  }  events成员描述需要监听哪些类型的事件，可以用以下几种标志来描述。  POLLIN：文件中有数据可读，用1表示，实验中使用到了这个标志  POLLPRI:：文件中有紧急数据可读，用2表示  POLLOUT：可以向文件写入数据，用4表示  POLLERR：文件中出现错误，只限于输出，用8表示  POLLHUP：与文件的连接被断开了，只限于输出，用16表示  POLLNVAL：文件描述符不合法，即它并没有指向一个成功打开的文件，用32表示 |
| numfds：需要监听的文件个数，即第一个参数所指向的数组中的元素数目 |
| timeout：表示poll阻塞的超时时间（毫秒）。如果该值小于等于0，则表示无限等待 |
| 函数返回值 | 成功：返回大于0 的值，表示事件发生的pollfd结构的个数  0：超时；  -1：出错 |