lab 0 实验报告

lab 0 总结

exercise sleep

这个实验要求实现一个用户态的sleep程序，并通过命令行键入sleep [time]调用，time是timer chip的中断周期数。由于这是一个用户态的程序，我们只需要直接调用操作系统提供给我们的系统调用函数sleep并将参数传入即可，传入参数需要将字符串转换为int，注意在参数不足时报错并终止程序。在调用sleep后也需要终止程序。

```
int main(int argc, char* argv[]){
	if(argc < 2){
		printf("sleep: lack of arguments.\n");
		exit(1);
	}

	sleep(atoi(argv[1]));
	exit(0);
}
```

exercise pipe

这个实验要求实现一个用户态的pingpong程序，通过管道实现父子进程的通信。这里需要创建两个管道，一个父进程写子进程读，一个子进程写父进程读，如果只用一个管道无法确定父子进程从管道中读取的byte是对方发送的同时也无法确定读取发送的先后次序。管道创建完成后调用fork()创建子进程，fork()返回0为子进程，从管道1中读取byte并向管道2发送，fork()返回值为子进程id为父进程，程序类似。注意最后父子进程都要关闭管道，不然管道可能会一直处于等待读取的状态。

```
int main(int argc, char* argv[]){
	int p1[2], p2[2];
	if(pipe(p1) < 0){
		printf("pipe: error.");
		exit(1);
	}
	if(pipe(p2) < 0){
		printf("pipe: error.");
		exit(1);
	}

	char byte = 'x';

	if(fork() == 0){
		// child proc
		if(read(p1[0], &byte, 1) > 0){
			printf("%d: received ping\n", getpid());
			write(p2[1], &byte, 1);
		}
	}
	else{
		write(p1[1], &byte, 1);
		if(read(p2[0], &byte, 1) > 0){
```

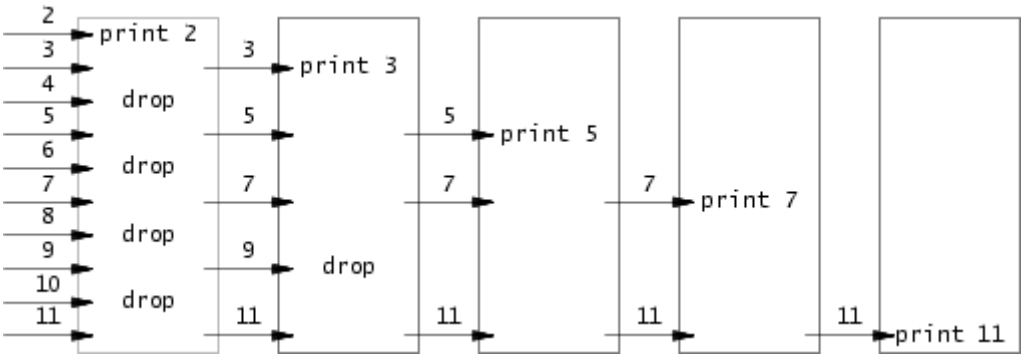
```
                printf("%d: received pong\n", getpid());
        }
    }

    close(p1[0]);
    close(p1[1]);
    close(p2[0]);
    close(p2[1]);

    exit(0);
}
```

踩坑，最开始我以为pipe是可以两端同时写和读的，这样写出来的程序只有子进程接受到了父进程传输的byte（当然），仔细阅读了xv6 book后发现自己对管道的功能理解有误，故改正。

exercise primes

这个实验要求利用进程和pipe实现一个素数筛选算法。根据补充页面的信息，算法大致如下

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
```



将上面的方框看作是线性排列的进程（进程间是父子进程关系），每两个进程通过一个pipe进行通信，一个进程从左边进程的pipe读取数，第一个读取的数是素数，将接下来读取的不被当前第一个数整除的数写入和右边进程的pipe中。通过数学归纳法很容易得到每次读入的第一个数是素数（不被比它小的素数整除）。

具体实现时需要在读入第一个素数后就创建子进程并建立和子进程间的pipe，将从上一级进程的pipe中读取的数进行筛查后写入和子进程的pipe中。为了方便子进程的创建，我额外写了一个primes_pipe函数，其参数为父进程创建的pipe的read fd。如果pipe中没有写入数，就关闭fd并直接返回，该子进程结束。如果有数可以读取，第一个读出的就是素数，接下来创建一个新的pipe并创建一个新的子进程，子进程继续调用primes_pipe，父进程则对从上一个进程读入的数进行筛查再写入和子进程的pipe。

```
void primes_pipe(int rd){
    int n, x;

    if(read(rd, &n, 4) == 0){
        close(rd);
        return;
    }
```

```
	printf("prime %d\n", n);
	int p[2];
	if(pipe(p) < 0){
		printf("pipe: error");
		exit(1);
	}

	if(fork() == 0){
		close(rd);
		close(p[1]);
		primes_pipe(p[0]);
		exit(0);

	}
	else{
		close(p[0]);
		while(read(rd, &x, 4) > 0){
			if(x%n){
				write(p[1], &x, 4);
			}
		}
		close(rd);
		close(p[1]);
		wait(0);
	}
	exit(0);
}



int main(int argc, char* argv[]){
	int p[2];
	if(pipe(p) < 0){
		printf("pipe: error");
		exit(1);
	}

	if(fork() == 0){
		close(p[1]);
		primes_pipe(p[0]);
	}
	else{
		printf("prime 2\n");
		close(p[0]);
		for(int i = 3; i <= 35; ++i){
			if(i%2){
				// 4 bytes
				write(p[1], &i, 4);
			}
		}
		close(p[1]);
		// it will end when child proc returns
		wait(0);
	}
	exit(0);
}
```

踩坑

当一个pipe的一个端口在该进程中没用时要即时关闭，因为父子进程都有自己的一份fd，所以共享一个pipe的父子进程都要及时关闭pipe，否则会因为fd不够用程序无法继续运行，素数的打印也会不完整。

父进程一定要等待子进程终止运行才能结束，否则无法保证输出的顺序，下一次命令行的'$'可能会在中途就输出，因为父进程提前结束了。因此要在父进程的末尾添加wait(0)等待子进程的正常终止运行。

exercise find

这个实验要求在指定目录及子目录下寻找指定名称的文件并输出文件路径。

观察ls.c发现可以使用的两个结构体

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

struct dirent是一个关于目录的结构体，每一个目录下有一系列的dirent结构指示目录下的目录或者文件名（inum我暂时不知道是做什么的）。struct stat是文件状态信息的一个结构体，在find.c中使用到的是文件类型。

find.c的实现类似于ls.c，对于指定路径，打开并获取fd，通过fd得到当前路径的stat，如果当前路径是一个目录，就从fd中读取一连串的dirent，通过dirent的name和path拼接得到当前文件/目录的具体路径，再通过路径获取stat，如果stat的type是文件就比对文件名和查找文件名，相同就输出，如果是目录就递归调用查找函数，在这个目录下继续寻找。

```
void find_target(char* path){
      // printf("path: %s\n", path);
      char buf[512], *p;
      int fd;
      struct dirent de;
      struct stat st;

      if((fd = open(path, 0)) < 0){
            fprintf(2, "find: cannot open %s\n", path);
            return;
      }

      if(fstat(fd, &st) < 0){
            fprintf(2, "find: cannot stat %s\n", path);
            close(fd);
            return;
      }

      // printf("here.\n");
      if(st.type == T_DIR){
            if(strlen(path) + 1 + DIRSIZ + 1 > 512){
```

```
                printf("find: path too long\n");
                goto end;
            }
            strcpy(buf, path);
            p = buf + strlen(path);
            *p++ = '/';
            while(read(fd, &de, sizeof(de)) > 0){
                if(de.inum == 0){
                    continue;
                }
                // ignore . and ..
                if(de.name[0] == '.'){
                    if(de.name[1] == 0 || (de.name[1] == '.' && de.name[2]
== 0)){
                        continue;
                    }
                }
                // printf("de.name: %s\n", de.name);
                memmove(p, de.name, DIRSIZ);
                p[DIRSIZ] = 0;
                if(stat(buf, &st) < 0){ // it is buf not path
                    fprintf(2, "find: cannot stat");
                }
                if(st.type == T_FILE && strcmp(de.name, target) == 0){
                    printf("%s\n", buf); // buf plz
                }
                else if(st.type == T_DIR){
                    find_target(buf);  // buf plz
                }
            }
        }

        end:

        close(fd);
        return;
}
```

和ls.c里的一样，路径过长需要给出警告并终止运行，同时在遇到'.'（当前目录）和'..'（上级目录）时不用继续调用查找函数，否则会无限递归。

踩坑

把需要用buf的地方写成了path，造成了无限递归，但由于fd是有限的其实在一个地方就没有继续运行了。

exercise xargs

实现一个类似于linux xargs的程序，从stdin中读取每一行字符串并作为xargs后的命令的参数运行。我们需要创建一个循环，不断从stdin中读取字符，直到不能再继续读取，每次遇到'\n'就代表读取到了一个完整的参数，将其添加在xargs执行的命令的参数后再运行此命令。执行时创建子进程并且父进程要等待子进程运行结束。整个实现还是比较简单的，利用了在kernel.param.h中定义的MAXARG，exec执行的最多参数个数，并且我添加了MAXARGLEN作为单个参数的最长长度。

```
int main(int argc, char* argv[]){
        if(argc <= 1){
            printf("xargs: arguments too few.");
```

```
            exit(1);
      }

      char** argv_ = malloc(MAXARG); // defined in kernel/param.h
      char* buf = malloc(MAXARGLEN);
      int i = 1;

      for(; i < argc; ++i){
            argv_[i-1] = argv[i];
      }
      argv_[i] = 0;
      --i;

      int n, j;
      while(1){
            j = 0;

            while((n = read(0, &buf[j], 1)) > 0){
                  if(buf[j] == '\n'){
                        // one argument's end
                        break;
                  }
                  ++j;
            }

            if(!n){
                  break;
            }

            buf[j] = 0;
            argv_[i] = buf;

            if(fork() == 0){
                  exec(argv[1], argv_);
            }
            else{
                  wait(0);
            }
      }
      exit(0);
}
```

lab 0 收获

lab util整体上比较容易，主要就是带领我们利用qemu模拟器启动xv6，并且实现几个用户态可以直接使用命令行调用的函数。sleep是一个简单的系统调用，pingpong是利用pipe进行进程间的通信，注意pipe只能一端写一端读。primes的实现比较tricky，利用pipe进行素数的筛选，在primes的实现中即时关闭不使用的pipe端口十分重要，因为端口是有限的。注意到父进程在创建子进程时二者都会有一个端口，需要在父子进程中都关闭才行，如果不能即时关闭端口，会因为端口有限而导致程序无法继续运行。并且由于pipe在等待读的过程中会一直等待读取直到写端口全部被关闭，如果不能即时关闭端口会导致程序一直卡在这里。find是参照ls的一个寻找目标文件的程序，实现find可以了解xv6在用户态获取文件信息的一些基本方法，比如通过打开目录路径读取fd获取dirent信息，通过fstat和stat获取文件状态信息并进行一系列操作。xargs是从stdin中读取参数并执行命令的程序，关键在于从stdin读取以'\n'结尾的参数并通过exec传入命令和参数信息执行。

由于这几个程序都比较独立和简单，所以在lab中并没有遇到特别难调试的地方，我主要通过printf输出关键信息进行调试。并且一些地方是由于我不熟悉一些用法（比如pipe）导致的bug，在查阅了xv6 book后能得以解决。