第一章 pds 的介绍

1.（概念）**POSIX (Portable Open System Interface eXchange)** 的介绍：**16** 页

2.（概念）**Bibliothèque** 和 **appel système**：**19** 页至 **22** 页

3.（程序）**mecho**：**26** 和 **27** 页

注意：1.option -n 2.非法 option 的处理 3.使 argument 后的空格正常化，如果使用 26 页中的"%s "的话，在最后一个 argument 后会多出一个空格。

4.（程序）获取环境变量：

方法一：在 argc，argv 后添加 char **arge，该数组中将包括所有环境变量，可用指针进行调用。
方法二：使用全局变量 environ（char **）进行调用：28 页下半部分，printenv 程序。
方法三：使用 POSIX 函数，getenv( )，核心代码：

```
username = getenv("USER"); 获取对应 user 环境变量的字符串
assert(username != NULL);
printf("Hello %s\n", username);
```

5.（程序）注册在 **main** 函数结束后调用的函数，核心代码：（使用 **atexit** 注册）

```
void bye() {}

int main(int argc, char **argv){

    atexit(bye);   注册函数
    printf("Hello ");
    exit(EXIT_SUCCESS);
    return 0;
```

} bye 函数将在 main 函数结束后被调用，注意，被注册的函数必须没有参数！

第二章 文件系统和输入输出

1.（概念）**fichier** 的基本概念：**7** 页

2.（概念）目录的概念：**8** 页：是特殊的文件，用于记忆文件系统的结构。由操作系统管理各项操作。
注意：目录实际上只保存了文件名，不包含文件！（Le répertoire contient une liste de noms d'entrées）

3.（概念）文件系统的多元性 **(pluralité)**：**9** 页，文件系统=abstraction d'un disque, 有统一的界面（对于不同的系统），例子为 u 盘的文件可以在任意系统下被显示。

4.（概念）文件系统的结构：**11** 页，**12** 页，树状结构。

#racine：记作 / ，是它自己的 parent 。
#arcs（entrée）：被命名，（文件的）注意：可使用所有符号除了'/0'和'/'，且应该避免使用空格，无法打印出来的字符和非 ascii 的字符。
#noeud non terminal：是目录，总有两个 fils，一个是 。一个是 。。，一个点代表它自己，两个点代表它的 parent。
#noeud terminal：是普通的文件. 包含数据。

注意：被命名的是 arc 而不是 noeud，所以文件名和目录名实际上为 arc 的名字！
注意：目录实际上只包含文件名，并不真的包含文件。

5.（概念）**noeud** 的编号：**13** 页。（因为访问时是通过两个编号进行访问的：盘（其他设备）的编号和 noeud 的编号）（如图所示，所有的结点被编号）。

对于每一个 répertoire 存在一个 noeud 编号的表：

```
。     33 （自身）
。。   23 （父目录）
debut  99 （普通文件）
……  ……
```

6.（概念）**lien physique**：15 页。如下，有两个不同的 arc（文件名）指向同一个 noeud，此时存在一个 lien physique 用于连接这两个不同的文件名。

/abc/ts ---> noeud 99
/qdsa/ws --->noeud 99 (在这两个不同的文件名之间存在一个 lien physique)

Disque1：32 99 33 45
Disque2：23 99 34 46

注意 1：在上面的例子中有所体现，由于在不同的盘中允许出现相同的编号，比如在 disque1 中有 99，在 disque2 中也有 99，它们是独立的，所以不能在不同的盘之间作 lien physique。（不同盘之间的 99 都是独立的，是不同的，不存在 lien physique）

注意 2：不可以对目录做 lien physique！

7.（概念）**lien symbolique**（英语名为： **link symbolic**）：
17，18，19 页。 指某个结点的内容即为另一个结点的 chemin（lien，路径），这个 chemin 指向的可以为一个普通文件的也可以为一个目录的。

#rm symlink 会删除这个 lien symbolique 自身，不会删除被指向的文件。
#cat symlink 会贴出它所指向的文件的内容。
#通过 ln -s 来进行创建。

%cat foo.txt
fffffffffffffffffffffffooooo
%ln -s foo.txt bar.txt
%ls
bar.txt foo.txt
%cat bar.txt
fffffffffffffffffffffffooooo

注意：lien symbolique 不是总是有效的，比如（删除指向的文件时，lien 变为无效）：
%rm foo.txt
%cat bar.txt
cat: bar.txt : No such file or directory

#指向其 pere 的 lien symbolique 会构成循环：
%cd /tmp
%ln -s /tmp foo
%cd foo/foo/foo (foo 指向 tmp 目录，切换到第一个 foo 会指向 tmp，切换到第二个还是指向 tmp)
%pwd (打印当前目录)
/tmp

#指向其自身的 lien symbolique 会造成过多 lien 的错误：

%ln -s bar bar
%cd bar
bar: Too many levels of symbolic links.

8.（命令）**mount** 和 **df**：
mount 装载命令
将一个文件系统的顶层目录挂到另一个文件系统的子目录上，使它们成为一个整体，称为装载。

#mount -t vfat /dev/hda1 /mnt/C 把 C 盘分区挂载到/dev/hda1 分区上。

df(disk free)查看磁盘分区实际使用情况,因为要使用 所以免不了要挂载到某个文件系统下面,所以通过它也可以实时了解实际挂载情况

zhaoshichen1@ubuntu:~/Bureau$ df -h
Sys. de fichiers Taille Uti. Disp. Uti% Monté sur
/dev/sda6 92G 39G 53G 43% /host
/dev/sda7 92G 58G 34G 64% /media/娱乐
/dev/sda5 92G 4,1G 87G 5% /media/软件
/dev/sda1 26G 15G 11G 59% /media/Windows7

9（程序）文件的相关信息**: 25** 至 **30** 页

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *path, struct stat *sb);
int lstat(const char *path, struct stat *sb);

**********************************************************************************************
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>

int main(){
        struct stat sb;
        int status;

        status=stat("caonima",&sb);
        /*对于一个 lien symbolique，lstat 才会得到它自己的结构体，stat 会得到它指向的文件的结构体*/
        if(status==0){
                /*基本信息*/
                printf("硬盘分区代号为: %d\n",(int)sb.st_dev);
                printf("noeud 号（重要）为：%d\n",(int)sb.st_ino);
                printf("该文件的 lien physique 数为：%d\n",(int)sb.st_nlink);
                /*文件的类型*/
                printf("该文件为普通文件：%d 该文件为目录：%d 该文件为 lien symbolique：
%d\n",S_ISREG(sb.st_mode),S_ISDIR(sb.st_mode),S_ISLNK(sb.st_mode));
                printf("该文件用户权限含运行：%d\n",sb.st_mode&S_IXUSR);
                /*写权限为 S_IWUSR 读为 S_IRUSR， 还有组权限和其他用户权限 */
                /*文件的大小*/
                printf("Longeur en octet (taille):%d\nPlace occupée sur le disque: %d\n",(int)sb.st_size,(int)sb.st_blocks);
                /*文件的 date 和 modification*/
                /*sb.st_atime 是最后一次进入该文件的时间*/
                /*sb.st_mtime 是最后一次修改该文件内容的时间*/
                /*sb.st_ctime 是最后一次修改该文件内容或属性的时间*/
                printf("最后一次进入该文件的时间 %s\n",ctime(&sb.st_atime));
                /*30 页 使用 ctime 函数将 time 转化成字符串形式用于打印 */
        }
        return 1;
}
```

运行结果为:
硬盘分区代号为: 1792
noeud 号（重要）为：375933
该文件的 lien physique 数为：1
该文件为普通文件：1 该文件为目录：0 该文件为 lien symbolique：0
该文件用户权限含运行：0
Longeur en octet (taille):55
Place occupée sur le disque: 2
最后一次进入该文件的时间 Wed Feb 8 16:35:06 2012

10.（程序）文件的属性（使用权，**droits d'accès**）：详见 **TP1**。**31** 页。

```
（额外加一个）#include <unistd.h>
int main(){
        /*查看文件属性（查看使用权情况）*/
        /*注意，拥有权利时 access 返回 0 */
        if(access("caonima",R_OK)==0) printf("可读文件\n");
        if(access("caonima",W_OK)==0) printf("可写文件\n");
        if(access("caonima",X_OK)==0) printf("chmod 使用前：可执行文件\n");
        else printf("chmod 使用前：不可执行文件\n");

        /*使用 chmod 修改文件属性*/
        /*********************************/
        chmod("caonima",S_IXUSR);
        /*********************************/
        /*S_IWUSR, S_IRUSR 分别表示用户的写，读权*/

        if(access("caonima",X_OK)==0) printf("chmod 使用后：可执行文件\n");
        else printf("chmod 使用后：不可执行文件\n");
        exit(EXIT_SUCCESS);
}
```

运行结果：可读文件
可写文件
chmod 使用前：不可执行文件
chmod 使用后：可执行文件

11.（程序）遍历目录，获取目录中的文件名：函数见 **33** 页。程序详情见 **34** 页。

12.（程序）读取 **lien symbolique** 指向的文件，**readlink** 的使用：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <limits.h>

int main(){
        char buf[PATH_MAX+1];
        ssize_t len;
        if((len=readlink("links/woshilink",buf,PATH_MAX))!=-1)
                buf[len]='\0';
        /*readlink 的返回值 len 为指向文件地址的字符数，地址在 buf 中，为相对于该 link 的地址（相对地址，要注意！）*/
        else perror("error reading symlink");
        printf("%s\n",buf);
        exit(EXIT_SUCCESS);
}
```

13.（程序）文件，目录，**link** 的创建和删除：**36-39** 页。
14.（程序）开文件（**ouverture**）：**42-48** 页。

15.（程序）关文件（**fermeture**）：**49** 页。 使用 **close(int fd)** 即可。

16.（程序）**fstat** 和 **fchmod** 的使用：**50** 页。使用 **descripteur** 代替 **path**。

17.（程序）**read** 和 **write** 函数的使用：**51-54** 页。

注意 1：返回值为实际操作了的字符数，为 0 时代表已经读完，所以常用 while 让 read 自动结束。
注意 2：读写文件时，（比如 copier 操作）需要验证并非同一文件，如 54 页的 ino 验证。

18.（程序）**readv** 和 **writev** 的使用（**Lecture / Ecriture indirecte**）：

****
额外加一个 #include <sys/uio.h>

```
int main(int argc, char** argv, char** arge){
        struct iovec *iov;
        int argl,i;

        for(argl=0;arge[argl];argl++);
        /*统计 arge 的数量*/

        iov=malloc(argl*sizeof(struct iovec));
        assert(iov);
        for(i=0;i<argl;i++){
                /*进行操作之前必要的设定！*/
                iov[i].iov_base=arge[i];
                /*存储要读取的区域的指针*/
                iov[i].iov_len=strlen(arge[i]);
                /*要读取的长度*/
        }
        writev(STDOUT_FILENO, iov, argl);
        /*iov 为一个 iovec 结构体指针，argl 为这个指针中存储的结构体数量,一次性所有的结构体都发动，顺序进行写操作*/
        exit(EXIT_SUCCESS);
}
```

19.（程序）**Postionnement**，**lseek** 的使用：**58-63** 页。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

int main(){
	int fd=open("caonima",O_WRONLY,0666),status=(int)lseek(fd,100,SEEK_SET);
	if(status==-1) perror("erruer in seek");
	printf("Cette fonction retourne la position absolue:%d\n",status);
	/*正常情况下返回绝对位置，返回-1时代表出错，注意对错误的处理*/
	/*第二个参数可以为负，但是要注意使用，有可能出错*/
	/*第三个参数的位置还可以填 SEEK_CUR(从当前位置开始挪动), SEEK_END（从文件尾开始挪动）*/
	write(fd,"caocaocao!!!",strlen("caocaocao!!!"));
	/*比如，该文件原本只有 3 个字符，现在却挪动到绝对位置 100 处开始写，从第 4 个位置到 100 的位置会全部是 0！！！*/
	exit(EXIT_SUCCESS);
}
```

20.（程序）**index** 读写：**64-66** 页。

21.（其他）**verrouillage**（锁）：**67-71** 页。有需要时再补充，貌似不是重点。

22.（其他）**Option d'ouverture**（**(non)bloquant, (non)synchronisé** 模式）：**72-74** 页。有需要时再补充，貌似不是重点。

23.（其他）**Projection mémoire**：（**map** 系函数的使用）**75-77** 页。有需要时再补充，貌似不是重点。

24.（程序）**2011 DS1** 经典文件题：

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>


int empty_dir(const char *dirname){
	/*判断一个文件夹是否为空的函数*/
	DIR *dirp;
	struct dirent *dp;
	int compteur=0;

	if((dirp=opendir(dirname))==NULL){
		perror("couldn't open dirname\n");
		return -1;
	}
	while((dp=readdir(dirp))){
		compteur++;
	}
	if(compteur==2) return 1;
	else return 0;
}
/*比较简单；判断一个文件夹是否为空，方法为数其中包含的文件数，注意所有目录都有.和..两个文件，所以空目录的文件数为2！！！！！！！！！！！*/

void justTellMe(const char* pathnameDelink,char* pathnameDeRep){
	/*处理 link 的，比如说 link 的目录为 zheli/woshilink,指向的为同文件夹的 aim，返回的字符串会为 aim 不是 zheli/aim!!!!*/
	/*这个函数就是将 zheli/woshilink 变成 zheli 这样一个目录，方便加 aim！！*/
	int i,length=strlen(pathnameDelink),compteur1=0,compteur2=0;
	for(i=0;i<length;i++){
		if(*(pathnameDelink+i)=='/'){
			compteur1++;
		}
	}
	for(i=0;i<length;i++){
		if(*(pathnameDelink+i)=='/'){
			compteur2++;
			if(compteur2==compteur1){
				/*注意是读到最后一个斜杠时删除之后的内容！！不是倒数第二个*/
				strcpy(pathnameDeRep,pathnameDelink);
				*(pathnameDeRep+i)='\0';
			}
		}
	}
```

```
}

int rm_empty(const char *pathname)
{

	struct stat sb;
	int status;
	char* buf=(char*)malloc(200);
	int success=1;
	char* path=(char*)malloc(4000);
	ssize_t len;


	status=lstat(pathname,&sb);

	if(status)
	/*注意 lstat 有问题时是 status 等于 1*/
	{
		printf("problème: %s\n",pathname);
		perror("appel de stat");

		return -1;
	}

	if(S_ISLNK(sb.st_mode))
	{
	/*LINK 的情况最复杂，使用 readlink 读出指向文件的路径，但是要处理，因为是相对于 link 所在位置的路径*/
	/*比如 link 的地址为 abc/link aim 为 abc/aim, 这里 justTellMe 之后， path 等于 abc,加上斜杠，加上返回的 aim，成为 abc/aim*/
		if((len=readlink(pathname,buf,4000))!=-1) buf[len]='\0';
		justTellMe(pathname,path);
		strcat(path,"/");
		strcat(path,buf);

		if(access(path,F_OK)!=0){
			printf("Ce lien %s n'est plus valid, donc ",pathname);
			status=unlink(pathname);
			if(status==0){
				printf(", et il est supprimé\n");
				success*=1;
			}
			else if(status==-1){
				perror("suppression");
				success*=0;
			}
		}
		/*使用 access 和 fok 判断文件是否存在！！！经典用法！！ */
		/*在对指向的文件操作前先确定 lien 是不是 valid*/

		else
		{
		/*是 valid 之后，用 rm empty 处理，然后再判断是否 valid！！！！ */
			success*=rm_empty(path);

			if(access(path,F_OK)!=0){
				printf("Ce lien %s n'est plus valid, donc ",pathname);
				status=unlink(pathname);
				if(status==0){
					printf(", et il est supprimé\n");
					success*=1;
				}
				else if(status==-1){
					perror("suppression");
					success*=0;
				}
			}
		}
		return success;
	}

	if(S_ISREG(sb.st_mode))
	{
	/*如果是普通文件只判断是否大小为 0 即可*/
		if(sb.st_size==0){
			printf("%s est vide ",pathname);
			status=unlink(pathname);
			if(status==0){
				printf(", et il est supprimé\n");
				success*=1;
			}
```

```
                else if(status==-1){
                        perror("suppression");
                        success*=0;
                }
        }
        return success;
}

if(S_ISDIR(sb.st_mode))
{
/*目录稍微复杂一点，要对目录中的每一个文件进行循环处理（包括子目录）*/
        DIR *dirp;
        struct dirent* dp;
        if(((dirp=opendir(pathname))==NULL)
        {
                perror("couldn't open pathname\n");
                success*=0;
        }
        while((dp=readdir(dirp)))
        {
                strcpy(buf,pathname);
                strcat(buf,"/");
                strcat(buf,dp->d_name);
                if((strcmp(dp->d_name,".")!=0)&&(strcmp(dp->d_name,"..")!=0)) success*=rm_empty(buf);
                /*注意处理的时候不处理.和..！！！！！！！！！！！！！！！！！！！！！！！！！*/
                buf[0]='\0';
        }
        /*处理完了以后如果目录变空则删除！*/
        if(empty_dir(pathname))
        {
                printf("%s est vide ",pathname);
                status=rmdir(pathname);
                if(status==0){
                        printf(", et il est supprimé\n");
                        success*=1;
                }
                else if(status==-1){
                        perror("suppression");
                        success*=0;
                }
        }
        return success;
}
return success;
}



int main(int argc, char* argv[]){
        if(argc!=2){
                printf("Argument invalid!\n");
                exit(EXIT_FAILURE);
        }
        if(rm_empty(argv[1])==0){
                printf("Il exist des suppression inpossible!\n");
        }
        else printf("Vous avez abouti à supprimer tous les fichiers(répertoires) vides!\n");
        exit(EXIT_SUCCESS);
}
```

第三章 **Processus** 进程

1.（概念） **Processus**，**programme**，**processeur** 的基本概念：**6-8** 页。

2.（概念）**Processus** 和 **Ressource** 的关系：**9** 页。

3.（概念）**Processus** 的 **etat**（状态）：**10** 页。逻辑 **etat** 分为 **actif** 和 **bloqué**（因为暂缺 **ressource**），有效 **etat** 分为 **prêt** 和 **élu**（被处理器选中去运行）和 **bloqué**。

4.（概念）**Processus** 的结构关系：**11** 页。最顶层的 **racine** 为 **init**，之下的 **processus** 以树状排列。子进程由父进程创造。

5.（程序）**Processus** 的各项基本属性：**12-16** 页。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>

static float tics_to_seconds(clock_t tics){
	/*注意 tms 这个结构体内的时间都是用 tics 为单位来表示的，需要进程特殊的转换才能变为浮点数*/
	return tics/(float)sysconf(_SC_CLK_TCK);
}

int main(){
	char* buf=(char*)malloc(300);
	struct tms* stat=(struct tms*)malloc(300);

	printf("当前 processus 的 PID 为：%d\n",getpid());
	printf("当前 processus 的父进程的 PID 为：%d\n",getppid());
	/*另有查看用户 id，getuid(),和该用户所在的组 id，getgid(),有效用户 id 和组 id，geteuid(),getegid()*/
	/*设置用户 id 和组 id，设置失败时返回-1，int setuid(uid_t newUid), int setgid(gid_t newGid) */
	/*详见 12 页*/

	printf("\n 当前 processus 的工作目录为：%s\n",getcwd(buf,300));/*buf 和返回值都为当前目录*/
	if(chdir("tds")==0){
		/*chdir 可以修改 processus 工作的目录（注意不是真正的当前目录，是 processus 的工作目录）*/
		/*修改成功时返回 0，不成功时返回 1*/
		printf("修改后的 processus 的工作目录为：%s\n",getcwd(buf,300));
		/*buf 和返回值都为当前目录*/
	}
	/*详见 13 页*/

	times(stat);
	printf("\n 用户 CPU 时间=%f\n 系统 CPU 时间=%f\n 结束了的子进程的 CPU 时间=%f\n 结束了的子进程的系统 CPU 时间=
%f\n",tics_to_seconds(stat->tms_utime),tics_to_seconds(stat->tms_stime),tics_to_seconds(stat->tms_cutime),tics_to_seconds(stat-
>tms_cstime));
	/*详见 14，15 页*/
	exit(EXIT_SUCCESS);
}
```

6.（概念）启动一个程序：**17** 页。分为 **2** 个步骤：从 **père** 克隆一个子进程，然后 **mutation**（在子进程中运行该程序）。

7.（程序）**Processus** 的克隆：**18-32** 页。
I. 克隆的基本操作：**18-20** 页。使用 **fork** 函数（正常时返回 **fils** 的 **pid**，如果在 **fils** 中时返回 **0**，错误时返回**-1**）。

II.父子进程间的继承关系：**21-24** 页。
子进程的记忆=父进程记忆的 **copie**
子进程与父进程分享 **stdin** 和 **stdout** 的读写缓存，分享同样的 **descripteur**。

III. 没有 **copy** 的属性：**25** 页。进程 **pid** 和父进程 **pid**，运行时间，父进程加的锁**(verrous sur les fichiers détenus par le père)**，信号。

IV.记忆 **copie** 的代价：**25** 页。
Copy on write 的基本 **idée**：**L'idée fondamentale : si de multiples appelants demandent des ressources initialement impossibles à distinguer, vous pouvez leur donner des pointeurs vers la même ressource. Cette fiction peut être maintenue jusqu'à ce qu'un appelant modifie sa "copie" de la ressource. A ce moment-là, une copie privée est créée. Cela évite que le changement soit visible ailleurs. Ceci se produit de manière transparente pour les appelants. L'avantage principal est que si un appelant ne fait jamais de modifications, la copie privée n'est jamais créée.**

V.wait 操作（父进程等待子进程的结束）： **26-29** 页。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <assert.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char** argv){

	pid_t status,FilsID;
	int j=20,i,fd=open(argv[1],O_RDWR);
	assert(fd!=-1);

	/*21，22 页，缓冲区的分享*/
	/*子进程会与父进程分享 stdin 和 stdout 这两个 tampons，即若没有 fflush 的操作，子进程也将输出这样一个句子*/
	printf("我是 stdout 中的缓存哦！ \n");
	fflush(stdout);
	/*fflush 函数清空一个文件流，这样子进程中将无法输出这样一个句子*/

	status=fork();
	switch(status){
		case -1:
			perror("Creation Processus");
			exit(EXIT_FAILURE);

		case 0:
			/*在子进程中，返回 0*/
			for(i=0;i<20000;i++);

			printf("[%d] Je suis fils\n",getpid());
			printf("[%d] Mon père est %d\n",getpid(),getppid());

			write(fd,"caonima",strlen("caonima"));

			break;
			/*这个 break 十分重要，不能忘记！！！！*/

		default:
			/*status 为返回的创造了的 processus 的 PID*/

			/*wait 的使用： 26-28 页。*/
			/*一般来说是 père 拿到 main（先于 fils 运行），用了 wait 后可使其等待 fils 结束后再进行自己下面的内容*/
			FilsID=wait(&status);
			/*参数为一个 int 值的地址，运行后 status 值会被修改，但是会返回 fils 的 PID*/

			/*或者指定一个 fils 来等待： 29 页。
				i=status;
				FilsID=waitpid(i,&status,WUNTRACED);
				i 即指定的要等待的 fils，i 为-1 时表示任意 fils，WUNTRACED 表示会等，WNOHANG 表示不等。
			*/

			if(WIFEXITED(status)){
				/*可以获得子进程的返回值： 27 页*/
				printf("[%d] Mon fils a retourné: %d\n",getpid(),WEXITSTATUS(status));
				printf("[%d] Mon fils est %d\n",getpid(),FilsID);
				write(fd,"rinima",strlen("rinima"));
				write(fd,"位置移动了吧",strlen("位置移动了吧"));
			}


			/*父进程子进程共享 descripteur： 23-24 页。*/
			/*write 方法会改变当前位置，第一个 write 是从文件最开始写，第二个 write 是接在第一个的后面开始写！！*/
			/*由于父进程与子进程之间的继承关系，子进程进行 write 操作时将会跟在父进程的 write 后面写*/
			/*被写过的文件： rinima 位置移动了吧 caonima */
			/*read 文件同理！！！！*/
	}

	printf("[%d] j=%d\n",getpid(),j);
	/*子进程 copy 父进程的记忆： 21，22 页*/
	/*fils 进程的运行是从 fork 语句之后一直到底的，但是由于子进程会 copy 所有父进程的记忆，所以子进程也可以输出 j 的值
*/

	printf("[%d]Je termine\n",getpid());
	exit(EXIT_SUCCESS);
}
```

VI.孤儿进程和僵尸进程：**30-32** 页。

孤儿进程：父进程在子进程之前结束。此时子进程成为孤儿，由 **init**（**racine**）来收养，**ppid** 变为 **1**。
僵尸进程：避免方法：**1.**使用 **double fork() 2.Ignorer le signal SIGCHLD 3.**使用 **wait** 回收子进程。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
	int i;
	pid_t status;

	status=fork();
	switch(status){
		case -1:
			perror("Creation Processus");
			exit(EXIT_FAILURE);
		case 0:
			printf("[%d] Je suis fils de %d\n",getpid(),getppid());
			break;
		default:
			sleep(2);
			/*子进程在父进程 wait 操作出来之前就结束了*/
			/*父进程就会一直等下去，此时子进程实际上为结束了，但是被父进程认为没结束，此时为 zombi*/
			i=wait(&status);
			while(WIFEXITED(status));
			printf("[%d] Mon fils est %d\n",getpid(),i);
	}

	printf("[%d] Je termine\n",getpid());
	exit(EXIT_SUCCESS);
}
```

通过命令：**./test & ps -el** 可看到：

```
0 S  1000  2896  2286  0  80   0 -  1006 hrtime pts/0    00:00:00 test
0 R  1000  2897  2286  0  80   0 -  2807 -      pts/0    00:00:00 ps
1 Z  1000  2898  2896  0  80   0 -     0 exit   pts/0    00:00:00 test <defunct>
```

test 子进程 **2898**（父进程为 **2896**）状态为 **Z**，即 **zombie**。

VII. 2011 DS1 中的同步运行 **rm_empty** 的关键代码：

```
		while((dp=readdir(dirp)))
		{
			strcpy(buf,pathname);
			strcat(buf,"/");
			strcat(buf,dp->d_name);
			if((strcmp(dp->d_name,".")!=0)&&(strcmp(dp->d_name,"..")!=0)){
				status=fork();
				switch(status){
					case -1:
						perror("Creation processus");
						exit(EXIT_FAILURE);
					case 0:/*fils processus*/
						printf("[%d]Je suis fils de %d, pour %s\n",getpid(),getppid(),buf);
						if(rm_empty(buf)) exit(EXIT_SUCCESS);
						else exit(EXIT_FAILURE);
					default:/*père processus*/
						filsID[compteur++]=status;
				}
			}
			/*注意处理的时候不处理.和..！！！！！！！！！！！！！！！！！！！！！！！！！*/
			buf[0]='\0';
		}
		/*父进程等待所有子进程，保证平行运行*/
		for(i=0;i<compteur;i++){
			waitpid(filsID[i],&statusT[i],WUNTRACED);
		}
		/*统一处理返回值的问题*/
		for(i=0;i<compteur;i++){
			if(WIFEXITED(statusT[i])){
				if(WEXITSTATUS(statusT[i])==0){
					success*=1;
					printf("[%d]Mon fils %d a bien fait son travail\n",getpid(),filsID[i]);
				}
				else success*=0;
			}
		}
```

8.（程序）**Processus** 的突变（**mutation**）**: 33-45** 页。
I.基本概念：**33** 页。**mutation=remplacement**，使用 **exec** 家族函数，代替原来的要运行的代码（该语句之后的，之前的不变）。
比如在一个进程中 **A exec B** 三个语句，**A** 会运行，**exec** 替代掉 **B** 及其后的内容，**B** 不会被运行。

代替以后 **PID**，**PPID**，**descripteur** 继承，运行时间，信号继承等均不变。（还是同一个 **processus**，只是改了代码）。

II.exec 函数家族：**36-39** 页。重要！！！运行命令或可执行文件。

9.（概念）**shell** 运行一个命令（程序）的方式：**40-43** 页。

10.（概念）**mutation** 过程中对 **descripteur** 的继承：**44**，**45** 页。

11.（程序）**Redirection des entrées/sorties**：**46-50** 页。**dup** 和 **dup2** 的使用。

12.（程序）进程管理器：**51-58** 页。

第四章 **gestion de signal**

1.（概念）**signal** 的基本概念：**4** 页。
定义：**signal=**软件的中断（以进程为目标，由进程或系统发出，可以不立刻运行（异步）。
处理过程：接收 作出反应 处理（**handler**） 安装 **handler** 调用接收信号的函数 重新进入运行
用途：传递信息，告知事件，使事件的接收者和发出者 **accord**

2.（概念）不同的 **signal** 及作用：**5**，**6**，**7** 页。

Core dump：我们在开发（或使用）一个程序时，最怕的就是程序莫明其妙地崩溃（挂掉）。虽然系统没事，但我们下次仍可能遇到相同的问题。于是，这时操作系统就会把程序挂掉时的内存内容写入一个叫做 **core**（指内存）的文件里（这个写入的动作就叫 **dump**，**dump** 的英语意思是垃圾、倾倒。从这里来 看，这些内存的内容是程序错误运行的结果，所以算是垃圾，把他弄出来就好比从大的内存池里**"**倾倒**"**），以便于我们调试。

名字	触发该信号的事件（信号告知的事件）	作用
SIGINT	指令性中断，Ctrl+c 触发	使进程结束
SIGQUIT	退出性中断，Ctrl+\ 触发	结束进程并 core dump
SIGHUP	连接中断，或 leader de session 结束	结束进程
SIGKILL	暂时的结束进程	持久的中断
SIGTERM	结束进程	结束进程
以上为与进程结束相关的		
SIGFPE	发生算术错误时	结束进程并 core dump
SIGILL	非法指令	结束进程并 core dump
SIGSEGV	破坏记忆（内存）保护	结束进程并 core dump
以上为与错误相关的		
SIGALARM	运行时间结束（échéance horloge）	忽略
SIGUSR1 和 SIGUSR2	由一个进程的用户发出的	结束进程
以上为其他类		
SIGTSTP	susp 暂停，Ctrl+z 触发	进程暂停
SIGSTOP	暂停的信号	进程长时间的暂停
SIGCHILD	一个子进程结束或者停止了	忽略
SIGCONT	一个停止了的进程继续了	如果停止了则继续
以上为停止和继续的		

3.（程序）**signal** 的基本信息（**identification**）：**7**，**8** 页。信号名，信号 **ID**，信号内容。

4.（程序）发出信号和等待信号：重要！！**10**，**11**，**12** 页。**Kill** 和 **pause** 的使用。

5.（概念）信号的状态：**13** 页。寄出，接收，**pendant**，释放。

6.（概念）信号的释放：**14** 页。

7.（概念）在接收信号时的对信号作用的调整：**15-17** 页。

一、保持默认的作用，**SIG_DFL**。
二、忽略该信号，被认为是 **SIG_IGN** 的 **traitant** 名字，在释放信号时啥也不做。
三、修改该信号的 **handler**：即使用 **SIG_DFL, SIG_IGN** 以外的 **handler**，重新定义信号的作用：**21-24** 页。

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


static void handler(int sig){
	/*handler 固定是这样，sig 作为参数，全局并不返回内容*/
	static int nusr=0;
	if(sig==SIGUSR1){
		nusr+=1;
	}
	else if(sig==SIGINT){
		printf("Signaux reçus: %d\n",nusr);
		printf("草尼玛，有损失啊\n");
	}
}


int main(int argc,char** argv){
	struct signaction sa;
	int numberSig=atoi(argv[1]),i;
	/*定义结构体的内容，在设置新的 handler 时有用*/
	sa.sa_hanlder=handler;/*定义 handler*/
	sigemptyset(&sa.sa_mask);/*清空*/
	sa.sa_flags=0;
	/*定义完毕*/
	/*设置新的 handler*/
	sigaction(SIGINT,&sa,(struct sigaction *)0);
	sigaction(SIGUSR1,&sa,(struct sigaction *)0);

	switch(fork()){
		case 0:
			for(i=0;i<numberSig;i++){
				kill(getppid(),SIGUSR1);
			}
			/*虽然寄出了 numberSig 个信号，但是信号可能有丢失*/
			printf("结束\n");
			exit(EXIT_SUCCESS);
		default:
			/*让父进程沉睡直到接收完所有的程序*/
			for(;;) sleep(1);
	}
	exit(EXIT_SUCCESS);
}
```

四、掩藏（**masquage de signaux**）：**18-20** 页。即 **handler** 运行时需要 **masquer** 的信号集，为 **sigset_t** 类型。
注意：对信号集合的处理：重要！**18** 页。

设置新的 **mask**：**sigprocmask(SIG_SETMASK, sigset_t *new, 0);**
显示被 **masquer** 的 **pendant** 的 **signaux**：**sigpending(sigset_t *pending);**

8.（程序）延时发出信号：**SIGALRM** 和 **alarm** 函数的使用：详见 **26**，**27** 页。
alarm(10)；就是在 **10** 秒之后才发出 **SIGALRM** 信号。如果在 **10** 秒钟之内又有一次调用 **alarm**，则旧的延迟时间被取消，采用新的延迟时间！

9.（程序）子进程的结束或停止：（子进程 **SIGTSTP** 或 **SIGSTOP** 或退出）**31-36** 页。父进程会自动收到 **SIGCHLD** 信号。

10.（程序）重启时间的控制：**41-44** 页。在结构体的 **flag** 处加一个 **SA_RESTART**，使得系统调用函数不会因被打断而出错。

TP1-TP5 精选代码：

TP2 du 的实现：算出给定文件夹内所有文件的大小总和。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

	static int opt_apparent_size=0;
	static int opt_follow_links=0;

	static int LesNoeuds[200];
	static int nbChiffre=0;
	/*LesNoeuds est une variable qui peut représenter les fichiers on a déjà parcouru.*/
	/*nbChiffre est le nombre de chiffres dans le 'pathname'*/

int valid_name(const char*name){
	if(!strcmp(name,"."))
		return 0;
	else if (!strcmp(name,".."))
		return 0;
	else return 1;
}

int findChiffre(int *des,int s,int n){
	int i=0;
	int found=0;
	while(((*(des+(i++)))==s)&&(i<n+1)) found=1;
	return found;
}
/*pour éviter de calculer un même fichier( au niveau du noeud ) plusieur fois*/
/*s'il y a un lien symbolique qui indiquer fichier X et X est calculé avant*/
/*on va passer ce fichier et ne calcule rien (0) */

int du_file(const char*pathname){

	struct stat sb;
	struct dirent *dp;
	DIR *dirp;
	int size=0;
	int status;
	status = lstat(pathname,&sb);

	char* copie;
	copie=(char*)malloc(sizeof(char)*200);

	if(status){
		perror("apple de stat");
		return 0;
	}


	if (S_ISREG(sb.st_mode)){
		if(!opt_apparent_size)
			size=sb.st_blocks;
		else size=sb.st_size;
	}

	else if(S_ISLNK(sb.st_mode)){
		if(opt_follow_links){
			stat(pathname,&sb);
			/*on suit le lien*/
			if(status){
				perror("apple de stat");
				return 0;
			}
			if(!opt_apparent_size)
				size=sb.st_blocks;
			else size=sb.st_size;
		}
		else{
			/*si on n'est pas demandé de suivre ce lien on retourne la taille de ce lien lui-même*/
			if(!opt_apparent_size)
				size=sb.st_blocks;
```

```
                else size=sb.st_size;
            }
        }

        else if(S_ISDIR(sb.st_mode)){
            size=sb.st_size;

            if((dirp=opendir(pathname))==NULL){
                perror("couldn't open it");
                return 0;
            }

            while(dp=readdir(dirp)){

                strcpy(copie,pathname);

                if(valid_name(dp->d_name)){
                    strcat(copie,"/");
                    strcat(copie,dp->d_name);
                    size+=du_file(copie);

                }
            }
            closedir(dirp);

        }

        if(!findChiffre(LesNoeuds,sb.st_ino,nbChiffre)){
            *(LesNoeuds+nbChiffre)=sb.st_ino;
            nbChiffre++;
        }
        else return 0;

        printf("path:%s ",pathname);
        printf("taille:%d\n",size);
        /*afficher les détailles des fichiers qu'on calcul pour faire le processus de calcul plus claire et précis*/

        return size;
}

int
main(int argc, char**argv){

        int i=1;
        int taille=0;
        if(argc>1){
            while(i<argc){
                if(strcmp(*(argv+i),"-L")==0){
                    opt_follow_links=1;
                    printf("Vous avez demandé de suivre des links!\n");
                }
                if(strcmp(*(argv+i),"-b")==0){
                    opt_apparent_size=1;
                    printf("Vous avez demandé d'afficher les tailles apparentes!\n");
                }
                i++;
            }
        }
        taille=du_file(*(argv+argc-1));
        printf("\nla taille totale est:%d\n\n",taille);
        return 0;
}
```

TP4 double fork

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

typedef void(*func_t) (void *);

void forkfork(func_t f, void *arg){
	pid_t status,status_pils;
	printf("[%d] Je vais engerdre \n", getpid());
	int k;
	status = fork();
	switch (status){
		case -1:
			perror("Creation processus");
			exit(EXIT_FAILURE);
		case 0:
			printf("[%d] Je vais engerdre \n", getpid());
			printf("[%d] Mon pere est %d \n", getpid(),getppid());

			status_pils = fork();
			if(status_pils == -1){
				perror("Creation processus");
				exit(EXIT_FAILURE);
			}
			else if(status_pils == 0){
				printf("[%d] je suis petit fils\n", getpid());
				printf("[%d] Je viens de naitre \n", getpid());
				printf("[%d] Mon pere est %d \n", getpid(),getppid());
				f(arg);
			}
			else{
				printf("[%d] Je termine\n", getpid());
				exit(EXIT_SUCCESS);
			}
			break;
		default :
			printf("[%d] j'ai engerdre \n", getpid());
			printf("[%d] Mon fils est %d \n", getpid(),status);
			waitpid(status,&k,WNOHANG);
	}
	printf("[%d] Je termine \n", getpid());
	exit(EXIT_SUCCESS);

}


int test(int *m){
	printf("test for %d \n", *m);
}


int main(){
	int i = 2;
	forkfork(test,&i);
	return 0;

}
```

TP5 do 运行多个命令被返回相应的结果

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include <assert.h>
#include "makeargv.h"

int
main(int argc, char** argv){

	int status,status2,i,ok,j=0,k,resultatAnd=1,resultatOr=0,resultatCC=0;
	int ifAnd=0,ifOr=0,nbOption=1,ifCC=0,ifKK=0,ifContinue=1;
	int returnValeur[20];
	char **cmdargv;

	/*traitement de l'option*/

	/*traitement de -and */
	if(strcmp("-and",argv[1])==0){
		ifAnd=1;
	}
	/*traitement de -and fini*/

	/*traitement de -or */
	else if(strcmp("-or",argv[1])==0){
		ifOr=1;
	}
	/*traitement de -or fini*/

	/*traitement des erreurs d'option */
	else{
		printf("Option erreur!\n");
		exit(EXIT_FAILURE);
	}
	/*traitement des erreurs d'option fini*/

	/*traitement de -cc*/
	if(strcmp("-cc",argv[2])==0){
		nbOption++;
		ifCC=1;
	}
	/*traitement de -cc fini*/

	/*traitement de -k*/
	if(strcmp("-k",argv[3])==0){
		nbOption++;
		ifKK=1;
	}
	/*traitement de -k fini*/
	/*traitement d'option fini*/


	/*traitement d'argument*/
	for (i=nbOption+1; i<argc; i++) {

		/* création du argv de l'argument i */
		status2 = makeargv(argv[i], " \t", &cmdargv);
		assert(status2>0);

		/*exécution*/
		if(ifContinue){

		/*ifContinue est la condition pour exécuter, si on a déjà obtenu le résultat (avec l'option -cc)*/
		/*On peut seulement afficher que la commande n'est pas encore terminée */
		/*N'exécute pas le program*/

		/*affichage du contenu d'exécution*/
			printf("\n*************************************************************\n");
			printf("*Le contenu d'exécution sont affichés ci-dessous:     *\n");
			printf("*************************************************************\n");
			fprintf(stderr, "[%s]", cmdargv[0]);
		/*affichage du contenu d'exécution fini*/

		/*création de processus fils pour exécuter*/
			status=fork();
			switch(status){
```

```
                case -1: perror("Creation processus");
                         exit(EXIT_FAILURE);

                case 0: /*fils*/
                        /*l'exécution*/
                        ok=execvp(*cmdargv,cmdargv);
                        if(ok==-1){
                                perror("Execution erreur");
                                exit(EXIT_FAILURE);
                        }
                        exit(EXIT_SUCCESS);
                        /*l'exécution fini*/

                default:/*pere*/
                        wait(&status);
                        if(WIFEXITED(status)){

                                /*si on a pas d'option cc ni k, on stock les valeurs de retour dans un tableau*/
                                /*en affichant les valeurs de retour pour rassurer les utilisateurs*/
                                returnValeur[j++]=WEXITSTATUS(status);
                                printf("[%s]Valeur de retour est %d\n",*cmdargv,returnValeur[j-1]);
                                /*l'affichage est fini*/

                                /*pour l'option -cc on affiche l'information de l'arrête*/
                                if(ifAnd&&returnValeur[j-1]==0&&ifCC){
                                        resultatCC=0;
                                        printf("\n*************************************************************\n");
                                        printf("*Il y a un program qui retourne 0 pour l'option -and  *\n");
                                        printf("*Donc on quitte.                                    *\n");
                                        printf("*************************************************************\n");
                                        ifContinue=0;

                                        /*si l'utilisateur n'a pas demandé -k */
                                        if(!ifKK) exit(EXIT_SUCCESS);
                                        /*si on a pas besoin de indiquer les commandes non terminées on quitte tout de suite!*/

                                }
                                if(ifOr&&returnValeur[j-1]==1&&ifCC){
                                        resultatCC=1;
                                        printf("\n*************************************************************\n");
                                        printf("*Il y a un program qui retourne 1 pour l'option -or   *\n");
                                        printf("*Donc on quitte.                                    *\n");
                                        printf("*************************************************************\n");
                                        exit(EXIT_SUCCESS);
                                        ifContinue=0;
                                        if(!ifKK) exit(EXIT_SUCCESS);
                                }
                                /*pour l'option -cc l'affichage de l'information de l'arrête est fini*/
                        }
                        else{
                                /*traitement des erreurs en fonction de valeur de retour*/
                                printf("Valeur de retour erreur %s\n",argv[i]);
                                exit(EXIT_FAILURE);
                                /*traitement des erreurs fini*/
                        }
                }
                /*exécution fini*/


                /*libération mémoire */
                 freeargv(cmdargv);
        }
        else if(ifKK){

        /*si l'utilisateur a demandé -k on affiche les commandes non terminées*/
                printf("\n#############################################################\n");
                printf("#Commande %s pas encore terminées :(            #\n",*cmdargv);
                printf("#############################################################\n");
        }
        /*l'affichage est fini*/
}

/*traitement de valeur de retour de version 2 (si l'utlisateur n'a pas demandé -cc ni -k)*/
for(k=0;k<j;k++){
        if(ifAnd)   resultatAnd*=returnValeur[k];
        else if(ifOr){
                if(resultatOr<returnValeur[k]) resultatOr=returnValeur[k];
        }
}
```

```
        /*traitement fini*/

        /*affichage du résultat final*/
        if(ifAnd) resultatCC=resultatAnd;
        else if(ifOr) resultatCC=resultatOr;
        printf("\n*********************************************************\n");
        printf("*Le résultat est %d                              *\n",resultatCC);
        printf("*********************************************************\n");
        /*l'affichage est fini*/
        exit(EXIT_SUCCESS);
        /*fin de program*/
}
```