VE482 Homework 7

Wang Lan 519370910084

Ex. 1

1. **Explain the content of the new table entries if a clock interrupt occurs at tick 10.**

   The interrupt will clear R bit:

Page	Time stamp	Present	Referenced	Modified
0	6	1	0	1
1	9	1	0	0
2	9	1	0	1
3	7	1	0	0
4	4	0	0	0

2. **Due to a read request to page 4 a page fault occurs at tick 10. Describe the new table entry.**

   Since page 4 is not presented, page 3 is checked. Since page 3 is not referenced, and it's older than 2 ticks, it will then be replaced.

Page	Time stamp	Present	Referenced	Modified
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	10	1	0	0
4	4	0	0	0

Ex. 2

1. **In which files are:**
   1. **the constants with number and name for the system calls?**

      `/include/minix/callnr.h`

   2. **the names of the system call routines?**

      `/servers/pm/table.c`

   3. **the prototypes of the system call routines?**

      `/servers/pm/proto.h`

   4. **the system calls of type “signal” coded?**

      `/servers/pm/signal.c`

2. **What problems arise when trying to implement a system call `int getchpids(int n, pid_t *childpid)` which "writes" the pids of up to n children of the current process into `*childpid`?**

   PM is in kernel, so a call from user space cannot directly deal with the information of processes.

3. **Write a "sub-system call" `int getnchpid(int n, pid_t *childpid)` which retrieves the n-th child process.**

```c
#define OK 0

int getnchpid(int n, pid_t *childpid) {
    struct mproc *mp;
    int count = 0;
    if (n > NR_PROCS) {
        return -1;  // err
    } else {
        for (mp = &mproc[0]; mp < &mproc[NR_PROCS]; ++mp) {
            if (mp->mp_parent == who_p) {
                ++count;
                if (count == n) {
                    *childpid = mp->mp_pid;
                    return OK; // return 0
                }
            }
        }
    }
    return -1; // err
}
```

4. **Using the previous sub-system call, implement the original `getchpids` system call. The returned `int` value corresponds to the number of pids in `*childpid`, or -1 on an error.**

   The function needs to be defined in `/servers/pm/proto.h`:

```c
int getchpids(int n, pid_t *childpid);
```

```c
int getchpids(int n, pid_t *childpid) {
    int count = 0, ret = 0;
    for (count = 0; count < n; ++count) {
        ret = getnchpid(count, childpid + count);
        if (ret == -1) {
            // err
            childpid = NULL;
            return -1;
        }
    }
    return count;
}
```

5. **Write a short program that demonstrate the previous system calls.**

```c
int main() {
    int n = 10;
```

```
    pid_t chPid[10];

    for (int i = 0; i < n; ++i) {
        pid_t pid = fork();
        if (pid == 0) {
            sleep(1000);
            return 0;
        } else {
            chPid[i] = pid;
        }
    }
    while (waitpid(-1, NULL, 0));

    pid_t childpid[10];
    int ret = getchpids(n, childpid);

    if (ret != -1) {
        for (int i = 0; i < n; i++) {
            printf("%d, %d\n", chPid[i], childpid[i]);
        }
    } else {
        printf("Error\n");
    }

    return 0;
}
```

6. **The above strategy solves the initial problem through the introduction of a sub-system call.**
    1. **What are the drawbacks and benefits of this solution?**

       Drawbacks: time consuming

       Benefits: Make sure that the system call can work correctly

    2. **Can you think of any alternative approach? If yes, provide basic details, without any implementation.**

       The whole process can be done in such a "sub system call", which means that it will only traverse the array once.

Ex. 3 [1]

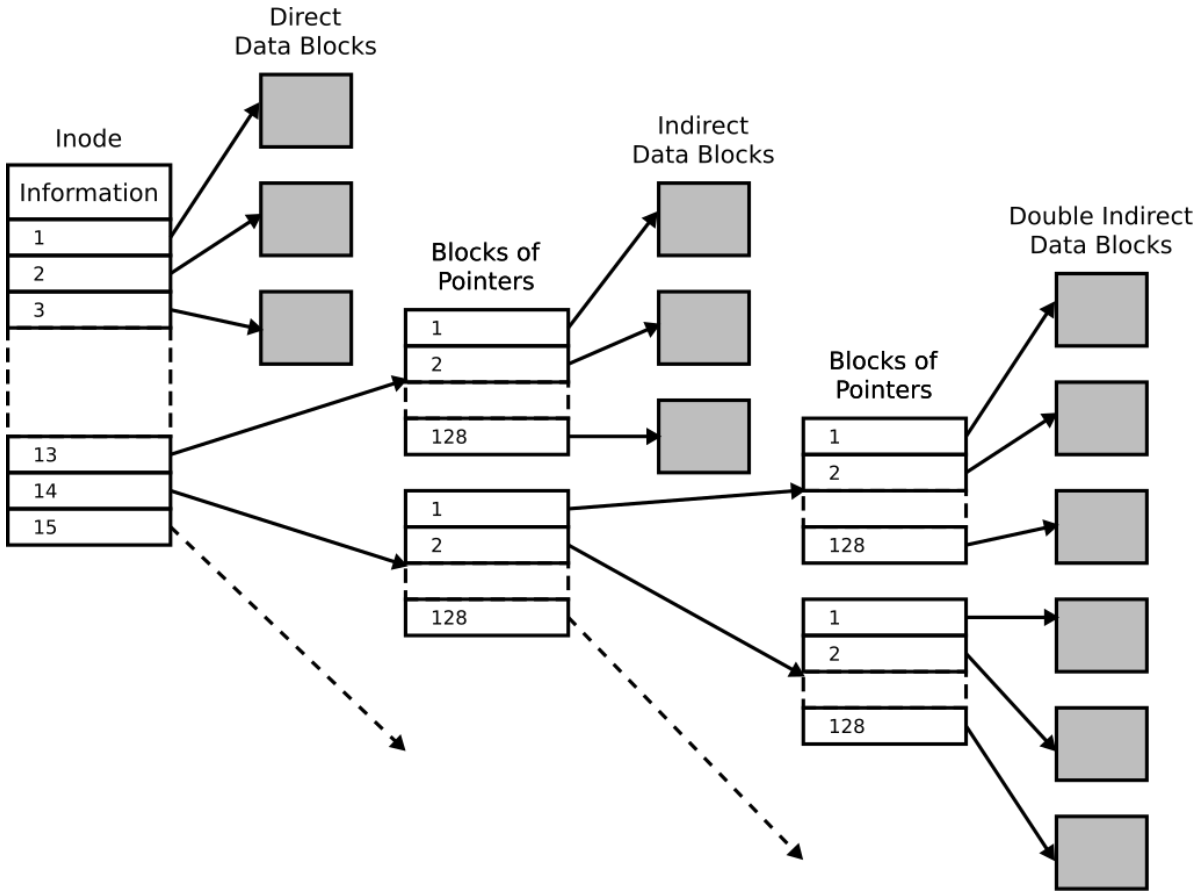
Write about a page on the topic of the `ext2` filesystem. Do not forget to reference your sources.

Block

The space in ext2 is split up into **blocks**, which are grouped into block groups. Data for any given file is typically contained within a single block group where possible. This is done to minimize the number of disk seeks when reading large amounts of contiguous data. Each block group contains a copy of the superblock and block group descriptor table, and all block groups contain a block bitmap, an inode bitmap, an inode table, and finally the actual data blocks.

Inode

Every file or directory is represented by an **inode**. The inode includes data about the size, permission, ownership, and location on disk of the file or directory. There is a structure in ext2 that has 15 pointers. Pointers 1 to 12 point to direct blocks, pointer 13 points to an indirect block, pointer 14 points to a doubly indirect block, and pointer 15 points to a triply indirect block. An example of inode structure is shown below:



Directory

Each **directory** is a list of directory entries. Each directory entry associates one file name with one inode number, and consists of the inode number, the length of the file name, and the actual text of the file name. To find a file, the directory is searched front-to-back for the associated filename. This is inefficient for very large directories.

The root directory is always stored in inode number two, so that the file system code can find it at mount time.

Subdirectories are implemented by storing the name of the subdirectory in the name field, and the inode number of the subdirectory in the inode field.

The special directories "." (current directory) and ".." (parent directory) are implemented by storing the names "." and ".." in the directory, and the inode number of the current and parent directories in the inode field. These two entries are automatically created when any new directory is made, and they cannot be deleted.

Allocating data

When a new file or directory is created, ext2 must decide where to store the data. ext2 attempts to allocate each new directory in the group containing its parent directory, on the theory that accesses to parent and children directories are likely to be closely related. ext2 also attempts to place files in the same group as their directory entries, because directory accesses often lead to

file accesses. However, if the group is full, then the new file or new directory is placed in some other non-full group.

The data blocks needed to store directories and files can be found by looking in the data allocation bitmap. Any needed space in the inode table can be found by looking in the inode allocation bitmap.