Problem 1. Page replacement algorithm

1. Given $\tau = 2$, we first calculate the ages for all pages

Page	Age
0	4
1	1
2	1
3	3
4	6

   We first let out pointer start from page 0, since it has $R = 0$ and $4 > 2$, but the page is not clean, so we cannot claim it immediately. For page 1, since $R = 1$, we set $R = 0$ and move on. For page 2, since $R = 1$, we set $R = 0$ and move on. For page 3, since $R = 0$ and $3 > 2$, and the page is clean, we can claim this page. For page 4, since $P = 0$, we cannot access this page.

2. As the same procedure as stated above, page 3 is claimed and the new page will be put there. The new table entry looks as follows.

Page	Time stamp	Present	Referenced	Modified
0	6	1	0	1
1	9	1	0	0
2	9	1	0	1
3		New page here		
4	4	0	0	0

Problem 2. Minix3

1. (a) `include/minix/callnr.h`
   (b) `servers/pm/table.c`
   (c) `servers/pm/proto.h`
   (d) `servers/pm/signal.c`

2. In order to get pid for multiple, up to n child process, one has to record all the return values of $fork$ system call, however, given the prototype `ssize_t write`, it requires

   (a) The file code (file descriptor or fd).
   (b) The pointer to a buffer where the data is stored.
   (c) The number of bytes to write from the buffer.

   Therefore, we might need to open a buffer in the user space to transform the information to the kernel space, and then the kernel could get process id and other related data, put those thing in the kernel buffer for later give back to the user space.

3. We can use a for loop to traverse to the nth child process and use `getchpid`, return 1 if success, -1 if failed.

```c
int getnchpid(int n, pid_t *childpid) {
    register struct mproc *l;
    int c = 0;
    for (l = &mproc[0]; l != &mproc[NR_PROCS]; l=l->next) {
        if (l->mp_parent == who_p) {
            if (c++ == n) {
                if(*c = l->getpid()){
                    return 1;
                }
                else return -1;
            }
        }
    }
}
```

4. According to the definition of `MINIX_GETPID` in `/include/minix/callnr.h`,

```c
case MINIX_GETPID:
        r = mproc[who_p].mp_pid;
        rmp->mp_reply.reply_res2 = mproc[rmp->mp_parent].mp_pid;
        break;
```

   and the original function of `getpid` in `/lib/libc/sys-minix/getppid.c`,

```c
pid_t getpid()
{
  message m;
  return(_syscall(PM_PROC_NR, MINIX_GETPID, &m));
}
```

   We can implement `getchpids` as follows

```c
int getchpids(int n, pid_t *childpid) {
    message m;
    m.m_lc_pm_getchpid.n = n;
    m_in.m_lc_pm_getchpid.childpid = childpid;
    return _syscall(PM_PROC_NR, getnchpid, &m);
}
```

5. We can test the previous function by comparing with the result of `fork`, n denotes the number of childprocess. We can use `dmesg` to check the real process id running in the background.

```c
void print_chpid(int n) {
    int i;
    pid_t childpid[n];

    for(i = 0; i < n; ++i) {
        pid_t pid = fork();
        if (!pid) {
            break;
        }
    }

    if (getchpids(n, childpid)) {
        for(int i = 0;i < n; ++i) {
            printf("%d\n", childpid[i]);
        }
    }
}
```

6. (a) Drawbacks: It's speed is limited to the communication between kernel space and user space, which is quite slow.
   Benefits: Since it is a sub-system call, it would not block other processes while running, like waiting I/O, it won't freeze other threads.

Problem 3. Research

`ext2` is the most legacy disk file system for Linux. As we can see, each operating system can use different file systems. For example, FAT (or FAT16) is the main file system used by Microsoft operating systems before Windows 98, the so-called NTFS file system for Windows 2000 and later, and Ext2 (Linux second extended) is the standard file system for Linux. file system, ext2fs) this one. Also, by default, Windows operating systems do not recognize Ext2 for Linux, and in traditional disk and file system applications, a partition can only be formatted as a file system, so we can say that a file system is a partition[1].

In traditional disk and file system applications, a partition can only be formatted as a file system, so we can say that a filesystem is a partition. Slot formatting for multiple file systems (e.g. LVM), as well as the ability to combine multiple slots into one file system (LVM, RAID). So, instead of formatting as a partition, we can now call a mountable piece of data a file system instead of a partition slot.

So how does the file system work? This is related to the operating system's file data. The file data of newer operating systems usually contains many attributes besides the actual contents of the file, such as file permissions (rwx) and file attributes (owner, group, time parameters, etc.) for Linux. The file system usually stores these two parts of the data in separate blocks, with the permissions and attributes in the inode and the actual data in the data block. In addition, there is a superblock that keeps track of the entire file system, including the total number of inodes and blocks, their usage, and their remaining capacity[2].

The smallest unit stored in a file system is a block, and the size of a block is determined at the time of formatting. A boot block is 1KB in size and is defined by the PC standard to

[1]http://www.ext2fsd.com
[2]https://www.kernel.org/doc/html/latest/filesystems/ext2.html

store disk partition information and boot information, and cannot be modified by any file system. After the boot block, the ext2 file system begins. ext2 file system divides the entire partition into several block groups of the same size.

In the overall plan, the file system has a boot sector at the front of the system where the boot manager can be installed. This is an important design because it allows us to install different boot managers on the front of individual filesy stems without having to overwrite the unique MBR of the entire hard disk, which allows us to create multiple boot blocks. A guided environment! For each block group, the six main elements are described below.

1. Super Block describes the file system information for the entire partition, such as inode/block size, total amount, usage, remaining amount, and file system format and related information. Super Blocks have a copy at the beginning of each block group (the first block group must have one, the subsequent block groups may not). In order to ensure that the file system works properly in case of physical problems in some sectors of the disk, it is necessary to ensure that the super block information of the file system is also accessible in this case. Therefore, a file system's super block is backed up in multiple block groups, and the data in these super block regions remains consistent. The information recorded by the super block is.
   (a) The total amount of blocks and inodes (the total amount of blocks and inodes of all Block Groups in the partition).
   (b) The number of unused and used inodes/blocks.
   (c) The size of block and inode (1, 2, 4K for block and 128 bytes for inode).
   (d) File system-related information such as the mount time of the filesystem, the time of the last data write, and the time of the last check of the disk (fsck).
   (e) A valid bit value, which is 0 if the file system has been mounted and 1 if it has not been mounted.
2. GDT (Group Descriptor Table) is composed of many block group descriptors, which corresponds to the number of block group descriptors the whole partition is divided into. Each group descriptor stores a block group's descriptive information, such as where in the group the inode table begins, where the data blocks begin, how many free inodes and data blocks there are, etc. The group descriptor stores a block group's descriptive information in each block group. The block group descriptor has a copy at the beginning of each block group.
3. The Block Bitmap is used to describe which blocks are used and which blocks are free in the entire block group. The block bitmap itself occupies one block, in which each bit represents a block of the block group. A bit of 1 means that the block is used, and a bit of 0 means that it is free and available. Assuming that the block size is 1KB when formatted, a block bitmap of this size can represent the occupancy of 1024*8 blocks, so a block group can have a maximum of 10248 blocks.
4. The inode bitmap is similar to the block bitmap, occupying a block by itself, in which each bit indicates whether an inode is free and available. The function of inode bitmap

is to record the usage of inode area in a block group, there can be 16384 inodes in a block group in Ext file system, which means a block group in this Ext file system can describe at most 16384 files.

5. The inode table consists of all inodes in a block group. In addition to data storage, some descriptive information about a file, such as file type, permissions, file size, creation, modification, access time, etc., should be stored in the inode instead of the data block. In the Ext2/Ext3 file system, the location of each file on disk is indexed by an inode pointer in the block group of the file system, and the inode will point the specific location to some blocks that actually record the file data. We call these blocks that actually record file data on the file system Data blocks.

Since each inode and block has a number, and each file occupies an inode, inside the inode is the block number where the data is placed. Therefore, what we can know is that if we can find the inode of a file, we will naturally know the block number of the file where the data is placed, and of course, we will be able to read the actual data of the file. This is a more efficient way of doing things, because it allows our disk to read and retrieve all the data in a short period of time, so the read/write performance is better[3].

[3]http://www.science.unitn.it/ fiorella/guidelinux/tlk/node95.html