VE482 Homework 7

Liu Yihao 515370910207

Ex. 1 — Page replacement algorithm

1. For page 0, $M = 1$, so it is not changed.
   For page 1 and 2, $R = 1$, so R is set to 0.
   For page 3, $9 - 7 = 2$, $\tau = 2$, so it is not changed.
   For page 4, $P = 0$, so it is not changed.

   The content of the new table entries are

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.

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

Ex. 2 — Minix 3

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

2. The order of children can't be ensured

3. ```
   int getnchpid(int n, pid_t *childpid) {
       register struct mproc *rp;
       int children;

       children = 0;
       for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
           if (rp->mp_parent == who_p) {
   ```
   ```

```
            if (children++ == n) {
                *childpid = rp->mp_pid;
                return 1;
            }
        }
    }
    return 0;
}
```

4.
```
// Defined in servers/pm/proto.h
int do_getchpid(void);

// Implemented in servers/pm/forkexit.c
int do_getchpid(void) {
    int n, children, result;
    pid_t *childpid;

    n = m_in.m_lc_pm_getchpid.n;
    childpid = m_in.m_lc_pm_getchpid.childpid;

    for (children = 0; children < n; ++children) {
        result = getnchpid(children, childpid + children);
        if (!result) break;
    }

    return children;
}

// There are also some changes in include/minix/callnr.h and
↪  include/minix/ipc.h

#define PM_GETCHPID         (PM_BASE + 48)
#define NR_PM_CALLS^^I^^I49

typedef struct {
  ^^Iint n;
  ^^Ipid_t *childpid;

  ^^Iuint8_t padding[48];
} mess_lc_pm_getchpid;
_ASSERT_MSG_SIZE(mess_lc_pm_getchpid);

union {
    // ...
    mess_lc_pm_getchpid m_lc_pm_getchpid;
    // ...
};

// Implemented in user library <getchpids.h>
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, PM_GETCHPID, &m);
}
```

5.
```
#include <unistd.h>
#include <getchpids.h>
#include <sys/types.h>
#include <stdio.h>

int main() {
    int parent = 1;

    for(int i = 0;i < 10;++i) {
        pid_t pid = fork();
        if (pid != 0) {
            parent = 0;
            break;
        }
    }

    if (parent == 0) {
        sleep(1000);
        return 0;
    }

    pid_t childpid[20];
    int result = getchpids(20, childpid);

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

6. a) The efficiency is not good.

   b)

Ex. 3 — Research

The ext2 or second extended filesystem is a file system for the Linux kernel. It was initially designed by Rémy Card as a replacement for the extended file system (ext). Having been designed according to the same principles as the Berkeley Fast File System from BSD, it was the first commercial-grade filesystem for Linux.

The canonical implementation of ext2 is the “ext2fs” filesystem driver in the Linux kernel. Other implementations (of varying quality and completeness) exist in GNU Hurd, MINIX 3, some BSD

kernels, in MiNT, and as third-party Microsoft Windows and macOS drivers.

ext2 was the default filesystem in several Linux distributions, including Debian and Red Hat Linux, until supplanted more recently by ext3, which is almost completely compatible with ext2 and is a journaling file system. ext2 is still the filesystem of choice for flash-based storage media (such as SD cards and USB flash drives) because its lack of a journal increases performance and minimizes the number of writes, and flash devices have a limited number of write cycles. However, recent Linux kernels support a journal-less mode of ext4 which provides benefits not found with ext2.

The space in ext2 is split up into blocks. These blocks are grouped into block groups, analogous to cylinder groups in the Unix File System. There are typically thousands of blocks on a large file system. 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.

The superblock contains important information that is crucial to the booting of the operating system. Thus backup copies are made in multiple block groups in the file system. However, typically only the first copy of it, which is found at the first block of the file system, is used in the booting.

The group descriptor stores the location of the block bitmap, inode bitmap, and the start of the inode table for every block group. These, in turn, are stored in a group descriptor table.

Every file or directory is represented by an inode. The term "inode" comes from "index node" (over time, it became i-node and then inode).[8] The inode includes data about the size, permission, ownership, and location on disk of the file or 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. For reasonable directory sizes, this is fine. But for very large directories this is inefficient, and ext3 offers a second way of storing directories (HTree) that is more efficient than just a list of filenames.

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. Hard links are implemented by storing the same inode number with more than one file name. Accessing the file by either name results in the same inode number, and therefore the same data.

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. The only special treatment these two entries receive is that they are automatically created when any new directory is made, and they cannot be deleted.

When a new file or directory is created, ext2 must decide where to store the data. If the disk is mostly empty, then data can be stored almost anywhere. However, clustering the data with related data will minimize seek times and maximize performance.

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.

Reference: https://en.wikipedia.org/wiki/Ext2

Ex. 4 — Simple questions

1. Yes, it is possible. We can use a spin lock to ensure the page is only accessed by one of the two processes.

2. $$\left\lceil \frac{32768}{4096} \right\rceil + \left\lceil \frac{16386}{4096} \right\rceil + \left\lceil \frac{15870}{4096} \right\rceil = 8 + 5 + 4 = 17 > \frac{65536}{4096} = 16$$

   So this program won't fit in the address space.

   If the page size were 512 bytes,

   $$\left\lceil \frac{32768}{512} \right\rceil + \left\lceil \frac{16386}{512} \right\rceil + \left\lceil \frac{15870}{512} \right\rceil = 64 + 33 + 31 = \frac{65536}{512} = 128$$

   Then it will fit.

3. No. The page descriptor can be binded with the segment descriptor so that one level lookup can find both of them.