VE482 - Homework 7

Introduction to Operating Systems

Kexuan Huang 518370910126

November 28, 2021

Ex.1 - Page replacement algorithm

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

During a clock interrupt occurs at tick 10, the reference bit on every clock tick will be set to 0.

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.

A page fault occurs on page 4, then we use WSClock to search for an existing page to replace.

- For page 0, it's modified, then schedule write.
- For page 1, it's referenced, set referenced to 0.
- For page 2, it's referenced and modified, set referenced to 0 and schedule write.
- For page 3, it's not referenced and modified, we replace page 3 and set time stamp to 10.

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

Ex.2 - Minix3

1. In which files are:

a) the constants with number and name for the system calls?

include/minix/callnr.h

```
/* In case it isn't obvious enough: this list is sorted
   numerically. */
#define EXIT            1
#define FORK            2
#define READ            3
#define WRITE           4
#define OPEN            5
#define CLOSE           6
```

b) the names of the system call routines?

servers/pm/table.c

```
int (*call_vec[])(void) = {
    no_sys,     /*  0 = unused  */
    do_exit,    /*  1 = exit    */
    do_fork,    /*  2 = fork    */
    no_sys,     /*  3 = read    */
    no_sys,     /*  4 = write   */
    no_sys,     /*  5 = open    */
    no_sys,     /*  6 = close   */
  ...
}
```

c) the prototypes of the system call routines?

servers/pm/proto.h

```
/* alarm.c */
int do_alarm(void);
int do_itimer(void);
void set_alarm(struct mproc *rmp, clock_t ticks);
void check_vtimer(int proc_nr, int sig);
```

d) the system calls of type “signal” coded?

servers/pm/signal.c

```
/* The entry points into this file are:
 *   do_sigaction:  perform the SIGACTION system call
 *   do_sigpending: perform the SIGPENDING system call
 *   do_sigprocmask:    perform the SIGPROCMASK system call
 *   do_sigreturn:  perform the SIGRETURN system call
 *   do_sigsuspend: perform the SIGSUSPEND system call
 *   do_kill:       perform the KILL system call
 *   do_pause:      perform the PAUSE system call
 *   process_ksig:  process a signal an behalf of the kernel
 *   sig_proc:      interrupt or terminate a signaled process
 *   check_sig:     check which processes to signal with sig_proc
      ()
 *   check_pending: check if a pending signal can now be delivered
 *   restart_sigs:  restart signal work after finishing a VFS call
 */
```

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`?

We don’t know in which order the children are sorted, unless we define a specified order in `pid_t *childpid`, e.g. fork time.

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

```c
#include "pm.h"
#include "mproc.h"

int getnchpid(int n, pid_t *childpid) {
  register struct mproc *rmc; /* pointer to the n-th child */
  if (n > NR_PROCS) return -1;
  rmc = &mproc[n];
  if (rmc->mp_parent != who_p) return -1;
  *childpid = rmc->mp_pid;
  return 0;
}
```

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.

1. servers/pm/proto.h

```
1  int do_getchpids(int n, pid_t *childpid);
```

2. servers/pm/forkexit.c

```
1  #include "pm.h"
2  #include "mproc.h"
3
4  int do_getchpids(int n, pid_t *childpid) {
5    int i;
6    for (i = 0; i < n; i++)
7      if (getnchpid(i, childpid+i) != 0)
8        return -1;
9    return i;
10 }
```

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

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define CHILD_NUM 10
5
6  int main(){
7    pid_t child_pid[10];
8    pid_t ret_pid[10];
9    for (size_t i = 0; i < CHILD_NUM; i++) {
10     pid_t pid = fork();
11     if (pid == 0)
12       exit(0);
13     else
14       child_pid[i] = pid;
15   }
16   printf("Compare children pids\n");
17   if (getchpids(CHILD_NUM, ret_pid) != 0){
18     for (size_t i = 0; i < CHILD_NUM; i++)
19       printf("getchpids: %d, fork: %d\n", ret_pid[i], child_pid[i
           ]);
20   }
21 }
```

6. The above strategy solves the initial problem through the introduction of a sub-system call.

a) What are the drawbacks and benefits of this solution?

Drawbacks: the multi layer sub-system call slows the system call down.
Benefits: the sub-system call ensures a clear system call structure and is easy to be reused.

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

Instead of retrive n-th child pid once per sub-system call, we can pass the whole array and return the chilren pids all at once.

Ex.3 - Research[1]

`ext2` (second extended file system) is a file system for the Linux kernel. was originally released in January 1993. Written by R'emy Card, Theodore Ts'o and Stephen Tweedie, it was a major rewrite of the Extended Filesystem. It is currently still (April 2001) the predominant filesystem in use by Linux. There are also implementations available for NetBSD, FreeBSD, the GNU HURD, Windows 95/98/NT, OS/2 and RISC OS.

`ext2` shares many properties with traditional Unix filesystems. It has the concepts of blocks, inodes and directories. It has space in the specification for Access Control Lists (ACLs), fragments, undeletion and compression though these are not yet implemented (some are available as separate patches). There is also a versioning mechanism to allow new features (such as journalling) to be added in a maximally compatible manner.

`ext2` is built on the premise that the data held in files is kept in data blocks. The Figure below shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks which can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks. The EXT2 file system divides the logical partition that it occupies into Block Groups. Each group duplicates information critical to the integrity of the file system as well as holding real files and directories as blocks of information and data. This duplication is neccessary should a disaster occur and the file system need recovering. The subsections describe in more detail the contents of each Block Group.

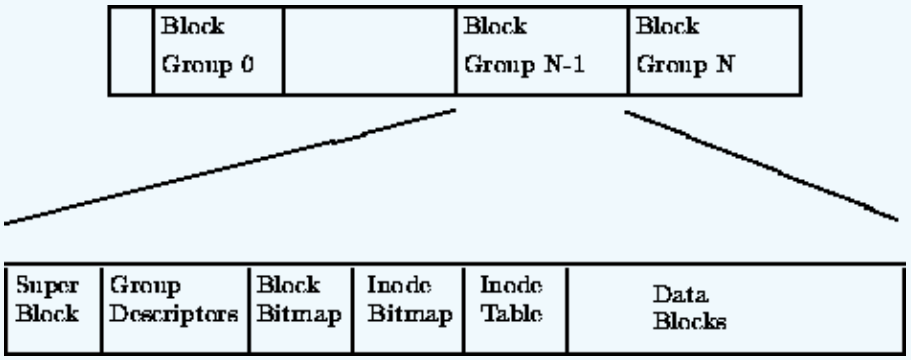

Figure 1: Physical Layout of the EXT2 File system

[1] The Linux Kernel Documentation

Ex.4 - Simple Questions

1. If a page is shared between two processes, is it possible that the page is read-only for one process and read-write for the other? Why or why not?

No. Typically with modern operating systems, when another process is forked from the first, they share the same memory space with a copy-on-write set on all pages. Any updates made to any of the read-write memory pages causes a copy to be made for the page so there will be two copies and the memory page will no longer be shared between the parent and child process. This means that only read-only pages or pages that have not been written to will be shared.

2. A computer provides each process with 65,536 bytes of address space divided into pages of 4096 bytes. A particular program has a text size of 32,768 bytes, a data size of 16,386 bytes, and a stack size of 15,870 bytes. Will this program fit in the address space? If the page size were 512 bytes, would it fit?

- For page of 4096 bytes: No
  - 65536 / 4096 = 16 pages in total
  - text = 32768 / 4096 = 8 pages
  - data = 16386 / 4096 = 5 pages
  - stack = 15870 / 4096 = 4 pages
  - 8 + 5 + 4 = 17 > 16
- For page of 512 bytes: Yes
  - 65536 / 512 = 128 pages in total
  - text = 32768 / 512 = 64 pages
  - data = 16386 / 512 = 33 pages
  - stack = 15870 / 512 = 31 pages
  - 64 + 33 + 31 = 128

3. When both paging and segmentation are being used, first the segment descriptor is found and then the page descriptor. Does the TLB also need a two-levels lookup?

No. The exact page can be found in a single match with segment and page number.