Ex1 Page Replacement Algorithm

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

   During the clock interrupt, the reference bit on every clock tick will be cleared. Therefore, the result is shown as follows:

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.**

   WSClock locates Page 3 to replace Page 4. Therefore, the result is shown as follows:

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

Ex2 Minix 3 System Call

1. **In which files are:**

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

   `/include/minix/callnr.h`

```
#define NCALLS       114    /* number of system calls allowed */

/* 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
/* omit */
```

   b) the names of the system call routines?

   `/servers/*/table.c`

```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 *
 * Created (MFS based):
 *   February 2010 (Evgeniy Ivanov)
 */

#define _TABLE

#include "fs.h"
#include "inode.h"
#include "buf.h"
#include "super.h"

int (*fs_call_vec[])(void) = {
    no_sys,             /* 0   not used */
    no_sys,             /* 1   */       /* Was: fs_getnode */
    fs_putnode,         /* 2   */
    fs_slink,           /* 3   */
    fs_ftrunc,          /* 4   */
/* omit */
```

c) the prototypes of the system call routines?

`/servers/*/proto.h`

```
/* Function prototypes. */

struct vmproc;
struct stat;
struct memory;
struct vir_region;
struct phys_region;
/* omit */
```

d) the system calls of type "signal" coded?

`/servers/pm/signal.c`

```
/* This file handles signals, which are asynchronous events and are
generally
 * a messy and unpleasant business.  Signals can be generated by the KILL
 * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
 * In all cases control eventually passes to check_sig() to see which
processes
 * can be signaled.  The actual signaling is done by sig_proc().
 *
 * 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
 */

/* omit */
static void unpause(struct mproc *rmp);
static int sig_send(struct mproc *rmp, int signo);
static void sig_proc_exit(struct mproc *rmp, int signo);
```

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`?**

   Issues related to memory may occur since the `getchpids` is called from userspace.

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

```
#include <unistd.h>
#include "mproc.h"
/* Global error number set for failed system calls. */
#define OK 0

int getnchpid(int n, pid_t *childpid){
  // register struct mproc *rmp; /* pointer to parent */
  register struct mproc *rmc; /* pointer to child */
  if (childpid == NULL) return -1;
  /* Number of slots in the process table for non-kernel processes. The
number
     * of system processes defines how many processes with special
privileges
     * there can be. User processes share the same properties and count for
one.
     *
     * These can be changed in sys_config.h.
  */
  if (n > NR_PROCS) return -1;
  rmc = &mproc[n];
  if (rmc->mp_parent != who_p) return -1;
  *childpid = rmc->mp_pid;

  return OK;
}
```

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.**

```
#include <unistd.h>
   #include "mproc.h"

   #define OK 0

   int getchpids(int n, pid_t *childpid){
       int i;
```

```
        while(i < n){
          if(getnchpid(i, childpid + i) != OK){
              return -1;
          }
          i++;
        }
        return i;
    }
```

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

```
#include <unistd.h>
#include <stdio.h>

int main(){
  pid_t childpid[10];
  int childNum = 10;
  pid_t childpidRef[10];

  for(int i = 0; i < childNum; i++){
    pid_t pid = fork();
    if (pid == 0) childpidRef[i] = pid;
    else exit (0);
  }

  getchpid(childNum, childpid);

  for(int i=0;i<childNum;i++){
    if(childpidRef[i] != childpid[i]){
        printf("%dth pid is wrong!",i);
        return -1;
    }
  }

  return 0;
}
```

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?

   - Benefits: Very easy to implement.
   - Drawbacks: Maybe slow because of calls for sub-system call.

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

   We may pass the `childpid` array to sub-system call to deal with the job without calling the sub-system call as above.

Ex. 3 - Research[1]

The **ext2** or **second extended file system** is a file system for the Linux kernel. It was initially designed by French software developer 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.

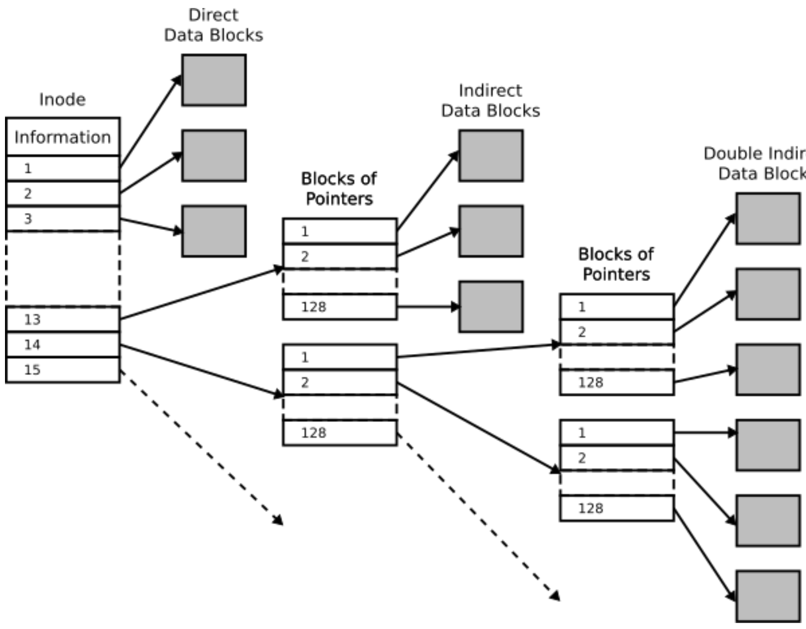
The space in ext2 is split up into blocks. These blocks are grouped into block groups, analogous to cylinder groups in the Unix File System. There are typically thousands of blocks on a large file system. Data for any given file is typically contained within a single block group where possible. This 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.

Inode
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).[10] The inode includes data about the size, permission, ownership, and location on disk of the file or directory.



Directories
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.

Allocating data
The data allocation is also controlled by ext2 strictly. New directory usually is allocated in the group that contains the parent directory. If the group is full, a new directory is seeked to place the data.

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. Any read or write action will lead to a copy of the page. As a result, the page will no longer be shared among those two processes.

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?**

   No for a page of 4096 bytes.

   Yes for a page of 512 bytes.

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. No necessary to have two-levels lookup.

Reference

1. ext2 - Wikipedia ↩