Minix Memory Manager

ECS150 Operating Systems
Winter 2004

Introduction

This document provides useful notes on the Minix Memory Manager (`MM`). It was prepared (rather hastily) by Sophie Engle (sjengle@ucdavis.edu) for ECS150 Operating Systems, Winter 2004. All information was taken from the source code and from the course book[1]. (In fact, it is basically a rehash of section 4.7 and the source code comments.) The layout of this document is as follows:

Section	Description
File List	Provides a list of files in `/usr/src/mm/` along with descriptions
Message Passing	Discusses how messages are passed to and from `MM`
Memory Allocation	Describes how free space is managed
Fork In-Depth	Looks at what `MM` does when a `fork` system call is made
Exec In-Depth	Looks at what `MM` does when an `exec` system call is made

Section 1: File List

All files for the memory manager are located in `/usr/src/mm/` in Minix. Below are descriptions of each file (19 total), organized by type.

Header Files

File Name	Description
`mm.h`	This is the master header file for `MM`. It basically just includes all the files that most `MM` programs will need to function.
`const.h`	This file contains all the constants used `MM`. For example, `PAGE_SIZE` (the number of bytes per page) is defined in this file.
`glo.h`	This file contains all the global variables used by `MM`. For example, a message sent to `MM` is actually stored in the global variable `mm_in`.
`type.h`	This file contains any type definitions local to `MM`. (The file is actually empty, and is only included for consistency with the kernel and the file system.)
`proto.h`	This file contains all the function prototypes local to `MM`.
`mproc.h`	The memory manager keeps its own process table (for storage of information required by `MM`). The `MM` process table is defined in this file, along with any related constants.
`param.h`	This file defines the messages fields used by `MM`. For example, `pid` is found at `mm_in.m1_i1`.

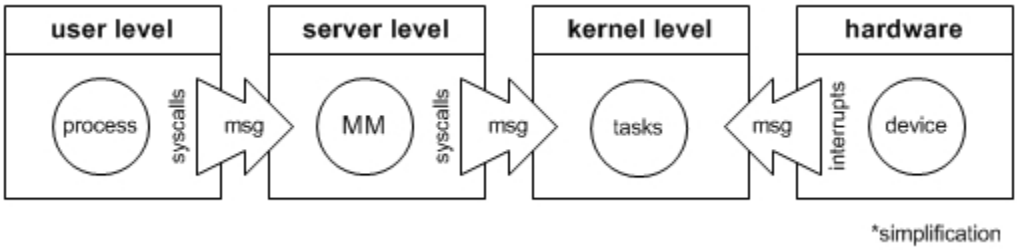
[1] Tanenbaum, Andrew S., and Albert S. Woodhull. Operating Systems: Design and Implementation. 2nd ed. Upper Saddle River: Prentice Hall, 1997.

Program Files

File Name	Description
`main.c`	This is the main `MM` program.
`alloc.c`	This file handles allocating and freeing memory.
`table.c`	This file indicates what functions to call for various system calls. (It also indicates which system calls the `MM` handles.) For example, `do_fork` is called when `MM` handles the `fork` system call.
`break.c`	This file includes the code necessary for `MM` to handle the `break` system call.
`exec.c`	This file includes the code necessary for `MM` to handle the `exec` system call.
`forkexit.c`	This file includes the code necessary for `MM` to handle the `fork` or `exit` system call.
`getset.c`	This file includes the code necessary for `MM` to get and set pids, uids and gids.
`misc.c`	This file includes the code necessary for `MM` to handle the `reboot` and `srvctl` system calls.
`signal.c`	This file includes the code necessary for `MM` to handle system calls regarding signals (like `sigaction` or `pause`.
`trace.c`	This file includes the code necessary for `MM` to handle the `ptrace` system call.
`utility.c`	This file contains some utility functions used by `MM`.
`putk.c`	This file allows `MM` to occasionally print messages.

Section 2: Message Passing

MINIX uses message passing to communicate between layers. The Memory Manager (`MM`) is located in the **server** level (see below).



For a process to make a system call, it must send a message to the server level. Specifically, a user process must send a message to either `MM` or `FS`. The servers may then send any necessary messages to the kernel level.

Not surprisingly, `MM` runs in an infinite loop receiving and processing messages. This can be seen in `main.c`:

```
16636    while( TRUE ) {
16638        get_work();       /* wait for an MM system call */
             ...
16655        reply(...);
16656    }
```

The `get_work()` function in `main.c` simply does a receive from any source, to the message `mm_in`:

```
16663    PRIVATE void get_work()
16664    {
16664        if( receive( ANY, &mm_in ) != OK ) ...
             ...
16670    }
```

Remember that `receive` will block until a message is actually received. Also, since the message `mm_in` is a global variable (declared in `glo.h`), any program in `MM` can access this message. The next step is to figure out what system call was made and what work must be done.

Each system call is assigned a number (`in /usr/include/minix/callnr.h`), and this number is placed in the `m_type` field of the message. Therefore, `MM` just has to look at `mm_in.m_type` to determine what system call was made.

To determine what has to be done for each system call, `MM` refers to `call_vec` in `table.c`. The call vector `call_vec` looks like:

```
16515    _PROTOTYPE (int (call_vec[NCALLS]), (void) ) = {
16516        no_sys,            /*  0 = unused   */
16517        do_mm_exit,        /*  1 = exit     */
16518        do_fork,           /*  2 = fork     */
16519        no_sys,            /*  3 = read     */
             ...
16594    };
```

Therefore, `fork` is assigned to number 2, and `MM` performs `do_fork` when a `fork` system call is made. However, when a `read` system call is made, `MM` does nothing (`no_sys`).

So `MM` performs whatever is necessary based on `call_vec[mm_in.m_type]`. When done, `reply(...)` is called. This is also a function defined in `main.c`, which basically places the results in the message `mm_out` and sends it back to the caller:

```
16676    PUBLIC void reply(...)
16681    {
             ...
16697        if( send( proc_nr, &mm_out ) != OK ) ...
16598    }
```

And that is how `MM` handles message passing!

Section 3: Memory Allocation

One of the most important tasks handled by `MM` is management of free space. Since MINIX does not support paging nor swapping, memory management is rather simple (compared to other operating systems). See section 4.7 in the book for an in-depth discussion, but basically `MM` maintains a list of "holes" or free space sorted by memory address. The actual data structure is located in `alloc.c`:

```
18820    #define NR_HOLES     128  /* max # entries in hole table */
         ...
18823    PRIVATE struct hole {
18824        phys_clicks h_base;   /* where does the hole begin? */
18825        phys_clicks h_len;    /* how big is the hole? */
18826        struct hole *h_next;  /* pointer to next entry on the list */
18827    } hole[NR_HOLES];
         ...
18830    PRIVATE struct hole *hole_head;  /* pointer to first hole */
```

So basically line `18827` defines an `hole` array of size `NR_HOLES`. Each `hole` keeps the beginning of the hole, how large the hole is, and a pointer to the next `hole` or free space. The first `hole` is given by `hole_head`. You can see how this list of holes is iterated through in the function `alloc_mem`:

```
18853    hp = hole_head;
18854    while( hp != NIL_HOLE ) {
             ...
18869        prev_ptr = hp;
18870        hp = hp->h_next;
18871    }
```

This loop is used by `alloc_mem` to find the first open hole large enough for the request. The rest of the functions in `alloc.c` manipulate this linked list depending on whether memory is being allocated or deallocated. If you looked at how MINIX manipulates the ready queue for scheduling, you'll see many of the same pointer manipulations.

Section 4: Fork In-Depth

What happens when a `fork` system call is made? First, `MM` receives a message with `m_type` = 2. Since `call_vec[2] = do_fork`, the flow goes to `do_fork` in `forkexit.c`.

First, `do_fork` checks to make sure there is enough space in the process table. If there is enough space, it calls `alloc_mem` to allocate memory for the child process:

```
16832    PUBLIC int do_fork()
16833    {
             ...
16846        if (procs_in_use == NR_PROCS ) return(EAGAIN);
             ...
16855        if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM ) ...
```

It is important to note that code sharing or **shared text** will be used (p359). This means that the

child process will share the same memory for the instructions, since it shares the same instructions as the parent. Therefore space is only allocated for the child's data and stack (which will be the same size as the parent's data and stack). The next step is to give the child a copy of the parent's data values (see line 16858).

After copying the parent's data, a slot is found in mproc:

```
16864    for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++ )
16865        if ( (rmc->mp_flags & IN_USE) == 0) break;
```

Once the loop breaks, rmc will point to an open slot in the MM process table. A little lost? Confused about mproc? Well, the memory manager must maintain its own process table, mproc, for every process. You can see the exact information stored in the file mproc.c. The next chunk of code in do_fork just involves filling in the proper information for each field in mproc.

The next step is to find an open pid for the new process:

```
16825    PRIVATE pid_t next_pid = INIT_PID+1;  /* next pid to be assigned */
         ...
16832    PUBLIC int do_fork()
16833    {
             ...
16885        do {
16886            t = 0; /* t = 0 means pid still free */
16887            next_pid = (next_pid < 3000 ? next_pid + 1 : INIT_PID + 1);
16888            for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++ )
16889                if (rmp->mp_pid == next_pid ||
                        rmp->mp_procgrp == next_pid) {
16890                    t = 1;
16891                    break;
16892                }
16893            rmc->mp_pid = next_pid; /* assign pid to child */
16894        } while (t);
```

Basically, the loop first increments next_pid. Then, it checks the entire mproc process table to see if that pid is in use. If it is in use, t gets set to 1 and the loop starts over again. If the pid is not in use, t will remain 0, and the loop will exit after assigning next_pid to the child process.

Finally, do_fork has completed the bulk of its work. It then sends messages to the kernel and to FS so that they can perform whatever is necessary to complete the fork system call. (For example, the kernel adds the child process to the process table, and FS lets the child inherit the parent's file descriptors.)

The last thing that happens is do_fork wakes up the child and returns the child pid to the parent process:

```
16832    PUBLIC int do_fork()
16833    {
             ...
```

```
16904         reply(child_nr, 0, 0, NIL_PTR);
16905         return(next_pid);
16906     }
```

Section 5: Exec In-Depth

Sorry everyone! I ran out of time to type this one out. ☹

Basically, you have to understand that memory must be allocated when an `exec` system call is made. It differs from `fork` since **shared text** can not be used and permissions have to be checked. The book gives a pretty good description on pages 368-371, and the source code is well-commented.