Chapter 4 Kernel Initialization (init)

4.1 Introduction

There is only one source code file, main.c, in init/ directory. The system transfer control to this code after boot/head.s finishes executing. While the code is not very long, it performs all the initializations for the kernel and creates of several initial tasks (processes), for example, task 0 and task 1 (the init task). Therefore, we will refer to numerous other initializing routines in the kernel source tree. If we can mainly understand the meaning of these initialization routines, then we have a primary understanding of the working principle of the kernel.

Starting from this chapter, there is a great deal of code written in C language. The C syntax is not used very often in normal application programming. Thus we need some knowledges about C language. One of the best book about C is still the one written by W. kernighan and Dennis M. Ritchie: "The C Programming language". Comprehending chapter 5 in the book is the key to master the C language.

In order to differentiate the orignal commentaries in the source code, we use '//' as the starting of our comments. For the *.h header files at the beginning of the source code, we just give its description in a line. We will describe each header file in chapter 11.

4.2 main.c

4.2.1 Function

The main.c program first sets the device number of root file system and a few variables related to physical ranges by using the parameters obtained by the setup code priviously. These memory variables indicate the total physical memory in the system, the end location of cache area for block devices, the beginning address of main memory area and the space used by ram disk if we selected to use ram disk in the main Makefile. For a machine with 16Mb total physical memory, Figure 4-1 demonstrates the memroy map divided up by the kernel.

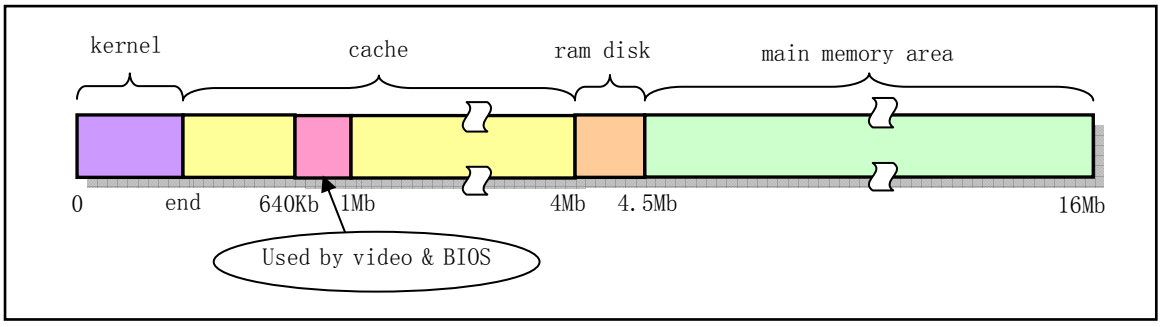

Figure 4-1 The regions of physical memory

From the privious chapter, we know that kernel code (system module) is located at the very begin of physical memory. After that, the main.c program allocates several megabytes of memory to use as cache buffer for the block device. This cache buffer area may be span the memory that is used by video adapter and ROM BIOS. So we must take out this particular area ranging from the memory location 640Kb to 1Mb from it. The cache buffer is managed by fs/buffer.c program with 1024 bytes as a operating unit (the buffer block) . The remaining kernel code can access these buffer blocks freely at anytime. After this comes the ram disk area if we defined its size in linux/Makefile file. The last part of the physical memory is the main memory area which is used by kernel code to allocate indicated size memory used by kernel task data structures and for requests by user programs. The kernel manages this area as continous physical page frames by the programs located within mm/ directory in the source code tree. The page frame is 4Kb in size.

After configured the memory, the kernel does all initializations of the hardware in machine. This involves setting up real interrupt gates, as opposed to dummy gates installed in head.s program, for hardwares in the machine and for the kernel system calls, "manually" loading the LDTR of the first task (task 0, or idle process).

When all of these have been done, it enables the interrupt requests and moves to the task 0 to continue execunting in user mode in supervisor level. At this moment the processor changes its privilege level from kernel level (0) to user level (3) and invokes the fork() function for the first time. This invocation produces the system second task -- task 1 or named as init process, which is used to run init() child process. The whole procedure of kernel initialization is depicted in Figure 4-2.

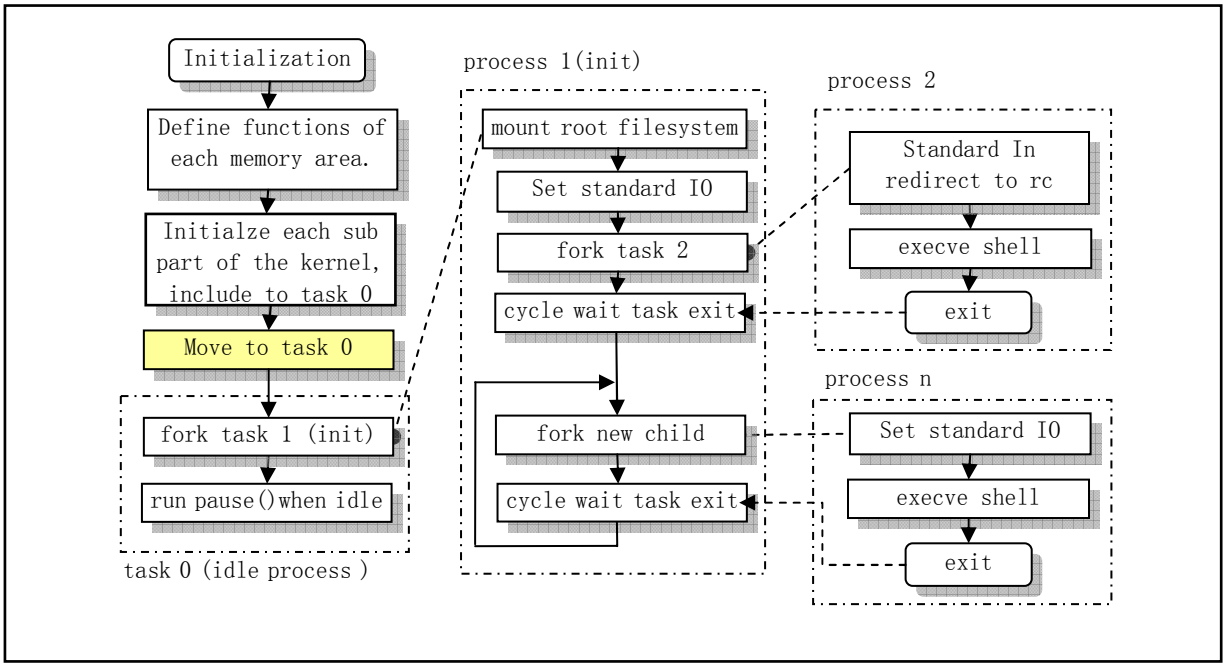

Figure 4-2 The sketch map of the procedure of kernel initialization

In the init process it continues initializing the user application environment and then executes the login shell program while the task 0 is running at kernel idle time.

If the init process succeeds setting up the terminal environment, it will create another new task (process 2) which is used to run shell program /bin/sh. If this new task exits at a later time, its parent process, init process, will fork a new one immediately and execute the /bin/sh again in this new task, while init process will be in a wait state.

Because forking a new process is done by copying its parent's code and data segments (including its user stack), in order to insure that no unwanted infos in the stack is copied we require the first task's stack to be clean before the first forking operation. That is to say the task 0 had better not use its stack by calling any function before forking. Thus, after main.c code runs in task 0, the fork's code can not be invoked as a function. The method implemented in main.c is to execute the fork system call through a macro with inline declaration. For example L23 in main.c, which is defined as a macro in include/unistd.h header file.

> By declaring a function inline, you can direct gcc to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

In addition, the pause() function called in task 0 also needs to be an inline function. If the schedule function runs the new forked init process first, then there is no problems at all. But the order the schedule program executing the parent (task 0) and child (init) process is not predictable, there is fifty percent chance the schedule program may run the task 0 first. So the pause() function needs to be an inline function too.

In this main.c program, tasks forked by init need some user level function supports, for example, open(), dup() and execve(). The kernel source tree uses a special separate directory lib/ to provide these functions for use in this program. The method used is similar to those used to creating a normal static C library.

4.2.2 Code Comments

Program 4-1 linux/init/main.c

```
1  /*
2   *  linux/init/main.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #define __LIBRARY__   // This label is defined to include the macros in unistd.h.
8  #include <unistd.h>   // Standard constants and types file. Defines constants and types,
                         // declaring lots function prototypes. If defined __LIBRARY__, then the
                         // system call labels and macros are also included, e.g. _syscall0().
9  #include <time.h>     // Defines two macros, and declares four types and several function
                         // prototypes for manipulating time.
10
11 /*
12  * we need this inline - forking from kernel space will result
```

```
13  * in NO COPY ON WRITE (!!!), until an execve is executed. This
14  * is no problem, but for the stack. This is handled by not letting
15  * main() use the stack at all after fork(). Thus, no function
16  * calls - which means inline code for fork too, as otherwise we
17  * would use the stack upon exit from 'fork()'.
18  *
19  * Actually only pause and fork are needed inline, so that there
20  * won't be any messing with the stack from main(), but we define
21  * some others too.
22  */
    // A system call with no argument defined as a macro begin from L133 in include/unistd.h.
    // Linux system call uses interrupt number 0x80. '_syscall0(int,fork) ' is the same as
    //'int fork()'. The suffix '0' in name '_syscall0' indicates this system call need no argument.
    // Therefore, if a system call needs one argument, we would use _syscall1(), etc. This line
    // declares a inline function prototype with its scope only in this program.
23  static inline _syscall0(int,fork)
    // int pause() system call: stop current process until receiving a signal.
24  static inline _syscall0(int,pause)
    // int setup(void * BIOS) system call, only used in this program.(at L71 of blk_dev/hd.c)
25  static inline _syscall1(int,setup,void *,BIOS)
    // int sync() system call: update filesystem.
26  static inline _syscall0(int,sync)
27
    // tty.h header file is used to define label constants, parameters about terminals.
28  #include <linux/tty.h>
    // schedule's header file. The task_struct{} and the data for first task (task 0) is
    // defined in the file. There are a few macros defined using assembler instructions, used
    // for handling base & limit in segment descriptors.
29  #include <linux/sched.h>
    // head.h is used to define a simple structure of segment descriptor and a few constans.
30  #include <linux/head.h>
    // Define several macros using asm instructions used for manipulating descriptors/trap
    // gates. The famous macro 'move_to_user_mode()' is defined in this file too.
31  #include <asm/system.h>
    // Several IO port operating macro functions are defined in io.h file with asm instructions.
32  #include <asm/io.h>
33
    // A file resembling a standard C library header. NULL, offsetof(TYPE, MEMBER) were defined.
34  #include <stddef.h>
    // Like a standard arguments header file, declares a type and defines three macros used for
    // functions with variable number of arguments, for example, vsprintf().
35  #include <stdarg.h>
    // [ Why include unistd.h here again? ignored ]
36  #include <unistd.h>
    // Involving constatns, structures and prototypes for file manipulating.
37  #include <fcntl.h>
    // Several types like SIZE_T, TIME_T, pid_t were defined in types.h file.
38  #include <sys/types.h>
39
    // A header file particularly used for filesystem handling. Several important filesystem
    // structures were defined, for example file{}, super_block{}, m_inode{}.
40  #include <linux/fs.h>
41
```

```
42 static char printbuf[1024];        // An buffer array used for kernel printing (printk()).
43
44 extern int vsprintf();              // Send formated output to a string (L92, vsprintf.c).
45 extern void init(void);             // The main stuff running in init process.
46 extern void blk_dev_init(void);     // (L157, blk_drv/ll_rw_blk.c)
47 extern void chr_dev_init(void);     // (L347, chr_drv/tty_io.c)
48 extern void hd_init(void);          // (L343, blk_drv/hd.c)
49 extern void floppy_init(void);      // (L457, blk_drv/floppy.c)
50 extern void mem_init(long start, long end);        // (L399, mm/memory.c)
51 extern long rd_init(long mem_start, int length);  // (L52, blk_drv/ramdisk.c,52)
52 extern long kernel_mktime(struct tm * tm);         // A routine for calc startup time (sec).
53 extern long startup_time;                          // kernel startup time (sec).
54
55 /*
56  * This is set up by the setup-routine at boot-time
57  */
58 #define EXT_MEM_K (*(unsigned short *)0x90002)      // Extended mem size (Kb) after 1Mb.
59 #define DRIVE_INFO (*(struct drive_info *)0x90080)  // Address of hard disk paras table.
60 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)  // Orignal root device number.
61
62 /*
63  * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
64  * and this seems to work. I anybody has more info on the real-time
65  * clock I'd be interested. Most of this was trial and error, and some
66  * bios-listing reading. Urghh.
67  */
68
   // A macro used to read real clock info in CMOS chip.
69 #define CMOS_READ(addr) ({ \
70 outb_p(0x80|addr,0x70); \           // 0x70 is write port. '0x80|addr' is address in CMOS.
71 inb_p(0x71); \                      // 0x71 is read port.
72 })
73
   // A macro for converting BCD code into binary values.
74 #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
75
   //// Read CMOS clock and set kernel startup time in seconds. Refers to Table 4-1.
76 static void time_init(void)
77 {
78         struct tm time;             // Time structure defined in include/time.h
79
   // Access data in CMOS is very slow. In order to minimize time difference, if the value
   // of seconds is changed after read all values, we should re-read all values again. Thus,
   // we can minimize the time difference in less than one second. In addition, all values
   // read are in BCD format.
80         do {
81                 time.tm_sec = CMOS_READ(0);         // Current second.
82                 time.tm_min = CMOS_READ(2);         // Current minute.
83                 time.tm_hour = CMOS_READ(4);        // Current hour.
84                 time.tm_mday = CMOS_READ(7);        // Current day in current month.
85                 time.tm_mon = CMOS_READ(8);         // Current month (1-12).
86                 time.tm_year = CMOS_READ(9);        // Current year.
87         } while (time.tm_sec != CMOS_READ(0));
```

```
 88         BCD_TO_BIN(time.tm_sec);                  // Convert to binary value.
 89         BCD_TO_BIN(time.tm_min);
 90         BCD_TO_BIN(time.tm_hour);
 91         BCD_TO_BIN(time.tm_mday);
 92         BCD_TO_BIN(time.tm_mon);
 93         BCD_TO_BIN(time.tm_year);
 94         time.tm_mon--;                            // Month range is 0 - 11 in tm_mon.
    // Calling the function in kernel/mktime.c, Calculate the seconds passed begin from
    // 1970.1.1.0 to now, used as system power on time.
 95         startup_time = kernel_mktime(&time);
 96 }
 97
 98 static long memory_end = 0;          // Total physical memory (bytes) in the machine.
 99 static long buffer_memory_end = 0;
100 static long main_memory_start = 0;
101
102 struct drive_info { char dummy[32]; } drive_info;  // Used for store hard disk info table.
103
104 void main(void)         /* This really IS void, no error here. */
105 {                       /* The startup routine assumes (well, ...) this */
106 /*
107  * Interrupts are still disabled. Do necessary setups, then
108  * enable them
109  */
    // The following lines (L110-122) are used to determine a few values, which are:
    // Root device number -> ROOT_DEV;      End address of buffer memory ->buffer_memory_end;
    // Total physical memory -> memory_end ; Start address of main memory area.
110         ROOT_DEV = ORIG_ROOT_DEV;               // ROOT_DEV defined at L29 in fs/super.c
111         drive_info = DRIVE_INFO;                // Copy hd para table at 0x90080
112         memory_end = (1<<20) + (EXT_MEM_K<<10); // Size (bytes)= 1Mb + extend mem(Kb)*1024
113         memory_end &= 0xfffff000;               // Ignore end mem less than 4Kb (1 page).
114         if (memory_end > 16*1024*1024)          // Kernel supports no more than 16Mb.
115                 memory_end = 16*1024*1024;
116         if (memory_end > 12*1024*1024)           // If total mem >12Mb,
117                 buffer_memory_end = 4*1024*1024; // then set buffer end=4Mb.
118         else if (memory_end > 6*1024*1024)
119                 buffer_memory_end = 2*1024*1024;
120         else
121                 buffer_memory_end = 1*1024*1024;
122         main_memory_start = buffer_memory_end;
    // If we have defined ram disk at L5 in linux/Makefile, it should be initilized and
    // the space of main memory area is reduced accordingly. see blk_drv/ramdisk.c.
123 #ifdef RAMDISK
124         main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125 #endif
    // Here begins the initialization of all aspects of kernel. It's better to follow into the
    // calling routines to see what is happen there. If we encounter any kinds of difficuties,
    // just go back here again and read the next routine. Doing this gives a way to preview
    // all sources in the kernel tree to have a primary picture of them :)
126         mem_init(main_memory_start,memory_end);
127         trap_init();            // (L181, kernel/traps.c).
128         blk_dev_init();         // (L157, kernel/blk_drv/ll_rw_blk.c).
129         chr_dev_init();         // (L347, kernel/chr_drv/tty_io.c).
```

```
130         tty_init();             // (L105, kernel/chr_drv/tty_io.c).
131         time_init();            // Setup startup time.
132         sched_init();           // (L385, kernel/sched.c). TR, LDTR are loaded for task 0.
133         buffer_init(buffer_memory_end); (L348, fs/buffer.c)
134         hd_init();              // (L343, kernel/blk_drv/hd.c)
135         floppy_init();          // (L457, kernel/blk_drv/floppy.c)
136         sti();                  // All initialization are done, enabling interrupts.
    // The macro bellow uses IRET instruction to start running in task 0 by first setup a few
    // parameters in stack resembling the environment for executing IRET.
137         move_to_user_mode();    // (L1, include/asm/system.h)
138         if (!fork()) {          /* we count on this going ok */
139                 init();         // runs in task 1 (init process).
140         }
    // The following codes are running as in task 0.
141 /*
142  *   NOTE!!   For any other task 'pause()' would mean we have to get a
143  * signal to awaken, but task0 is the sole exception (see 'schedule()')
144  * as task 0 gets activated at every idle moment (when no other tasks
145  * can run). For task0 'pause()' just means we go check if some other
146  * task can run, and if not we return here.
147  */
    // pause() syscall (L144, kernel/sched.c) will change the state of task 0 to TASK_INTERRUPTIBLE,
    // but the schedule() will switch to task 0 whenever there is no other task to run and
    // ignoring the state of task 0.
148         for(;;) pause();
149 }
150
    // Generating formatted output to standard output device (stdout, 1). Argument '*fmt' indicates
    // the format to be used. This routine is just a good example for how to use vsprintf().
    // This routine put the formatted string into the buffer (printbuf) by using vsprintf()
    // and then writes the buffer contents to the standard output. Refer to kernel/vsprintf.c.
151 static int printf(const char *fmt, ...)
152 {
153         va_list args;
154         int i;
155
156         va_start(args, fmt);
157         write(1,printbuf,i=vsprintf(printbuf, fmt, args));
158         va_end(args);
159         return i;
160 }
161
162 static char * argv_rc[] = { "/bin/sh", NULL };     // Argument array used for running shell.
163 static char * envp_rc[] = { "HOME=/", NULL };      // Environment string array.
164
    // The '-' in argv[0] bellow is a flag which is passed to shell program. By identifying this
    // flag, the shell program will run as a login shell. There are some differences when running
    // the shell program under command line prompt.
165 static char * argv[] = { "-/bin/sh",NULL };
166 static char * envp[] = { "HOME=/usr/root", NULL };
167
    // init() is run in the task 0's child process: init process. At first it initializes the
    environment of the program (shell) to be run, then executes the program.
```

```
168 void init(void)
169 {
170         int pid,i;
171
    // This is a syscall defined at L71 in kernel/blk_drv/hd.c. It is used to read the hard disk
    // parameter table including partition table and load the ramdisk if it exists, then mount
    // the root filesystem.
172         setup((void *) &drive_info);
    // The following lines open the device special file '/dev/tty0' with read/write access mode.
    // This file corresponds to terminal console. As this is the first time the file is opened,
    // the file descriptor is sure to be 0 (stdin), which is the default stardard input descriptor
    // of UN*X type of operating system. Openning with read/write mode is used to duplicate it
    // to generate file descriptors of standard output (stdout) and standard error (stderr).
173         (void) open("/dev/tty0",O_RDWR,0);
174         (void) dup(0);                          // Create descriptor 1 for stdout.
175         (void) dup(0);                          // Create descriptor 2 for stderr.
    // Display buffer block number and buffer size in bytes. Show the free size in main memory.
176         printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
177                 NR_BUFFERS*BLOCK_SIZE);
178         printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);
    // The forking below creates a child process (task 2). In the child process, fork() returns
    // a zero, while the child process's pid is returned to its parent process. So the lines
    // L180-184 are to run in the child process.
    // In the child process, the input stream of stdin is directed to '/etc/rc' by close the
    // stdin first and reopen it associated with '/etc/rc'. rc file is used for starting services
    // for the system just like a 'autoexec.bat' file in the DOS operating system. For the Linux
    // 0.11 system, it may contains some commands like update, mount, crond, etc.
    // Then the child process invokes the execve() function to execute the shell program with
    // the contents of 'rc' file as its input streams. The execve() will replace the task 2's code
    // with shell code of '/bin/sh' and never return. If execve() encounter an error and returned
    // unexpectedly, then this child process will exit and terminated with error code 2 (means file
    // or directory not found). execve() is implemented at L182 in fs/exec.c file.
179         if (!(pid=fork())) {
180                 close(0);
181                 if (open("/etc/rc",O_RDONLY,0))
182                         _exit(1);
183                 execve("/bin/sh",argv_rc,envp_rc);
184                 _exit(2);
185         }
    // The following lines are running in the parent process after it forks a child process.
    // wait() is used to wait a child process to exit or terminate and its return value should
    // be the child process pid.
    // The effects of these three lines is that the parent process waiting for the ending of the
    // child above. '&i' is the loaction for storing return status. If the return value of wait()
    // doest not equal the child pid waiting, this parent process will continue waiting.
186         if (pid>0)
187                 while (pid != wait(&i))
188                         /* nothing */;
    // If we reach this location, this means that the child process created above is dead and
    // terminated. In the 'while' loop, we fork another new child again. If forking returns
    // a value less than zero, it means an error has encountered by the forking and a message
    // 'Fork failed in init' will be shown, and the kernel will continue to fork a new process.
    // In the newly created child, it will close all the privious standard file descriptors
```

```
    // and then creates a new session (see the description after the program) and opens standard
    // file descriptors again for the child and finally runs the shell program again. This time
    // the system does not need use '/tec/rc' file as shell's input because the commands in rc
    // file is only need to run once and the shell is in superuser mode. Afterwards, the parent
    // process (init) waits until the child is exited or terminated again. At this moment,
    // init process will print a message stating the child is died and then flushes the buffer
    // using sync() and restarts the whole operatin from L190.
189         while (1) {
190                 if ((pid=fork())<0) {
191                         printf("Fork failed in init\r\n");
192                         continue;
193                 }
194                 if (!pid) {                                     // Child process.
195                         close(0);close(1);close(2);
196                         setsid();                               // create a new session.
197                         (void) open("/dev/tty0",O_RDWR,0);
198                         (void) dup(0);
199                         (void) dup(0);
200                         _exit(execve("/bin/sh",argv,envp));
201                 }
202                 while (1)
203                         if (pid == wait(&i))
204                                 break;
205                 printf("\n\rchild %d died with code %04x\n\r",pid,i);
206                 sync();                                         // syncing the buffer.
207         }
208         _exit(0);       /* NOTE! _exit, not exit() */
    // _exit() and exit() are used to normally terminate a program. The difference between them
    // is that the former is a direct syscall of sys_exit while the later is a function implemented
    // in a normal library. exit() will do some cleaning opeartion first, for example closing all
    // the standard IO, calling the termination process routines, and then invoke the sys_exit.
209 }
210
```

4.2.3 Information

4.2.3.1 CMOS

PC's CMOS (Complementary Metal Oxide Semiconductor) is at least a 64 bytes RAM memory block powered by battery and is part of the system clock chip. This 64-byte CMOS is orignally used by IBM PC-XT to store the real clock time & date using BCD format. As it uses only 14 bytes, the remaining space is used for storing other system information.

The address space of CMOS memory doesn't exist in the physical memory space, so we have to use IO port (0x70, 0x71) to access it. In order to read a indicated byte, we need first output its offset to port 0x70 using instruction OUT and, then, to read the byte from port 0x71 using IN instruction. Table 4-1 is a simple table of the contents of CMOS.

Table 4-1 CMOS RAM memory map

Offset	Description	Offset	Description
0x00	Current seconds in BCD	0x11	System configuration settings
0x01	Alarm second (BCD)	0x12	Hard disk types.

0x02	Current minute(BCD)	0x13	Reserved.
0x03	Alarm minute (BCD)	0x14	Installed equipments.
0x04	Current hour (BCD)	0x15	Base memory low-order byte.
0x05	Alarm hour (BCD)	0x16	Base memory high-order byte.
0x06	Current day of week (BCD)	0x17	Extended memory low-order byte.
0x07	Current date (BCD)	0x18	Extended memory high-order byte.
0x08	Current month (BCD)	0x19-0x2d	Reserved.
0x09	Current year (BCD)	0x2e	CMOS checksum high order byte
0x0a	Status register A	0x2f	CMOS checksum low order byte.
0x0b	Status register B	0x30	Actual extended memory size over 1Mb low-order byte
0x0c	Status register C	0x31	Actual extended memory size over 1Mb high-order byte.
0x0d	Status register D	0x32	Date century in BCD
0x0e	POST diagnostic status	0x33	POST information flag.
0x0f	Shutdown status code	0x34-0x3f	Reserved.
0x10	Floppy disk drive types		

4.2.3.2 Forking new process

fork() is a syscall. It copies the current process (parent process) and generating a new process which runs the same code as its parent. But the new process has its own data space and environment parameters.

In parent process, the return value of fork() is the pid of new process, while in the child process, fork() returns a zero. Thus, although they runs same code, they are running their own code by indentifing the returning value of fork(). If forking failed, it returns a negative value.The procedure of forking is demonstrated in Figure 4-3.

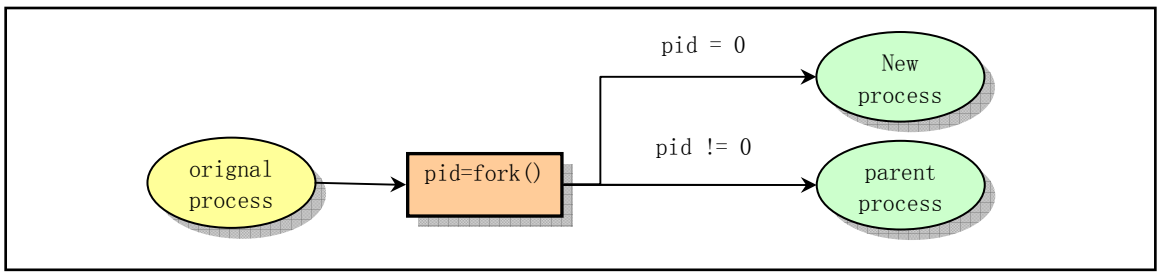

Figure 4-3 Creating new process by invoking fork()

4.2.3.3 Session concepts

As stated in chapter 2, a program is a runnable file, a process is a running instance of the program. In the kernel, every process is identified by a non-negative integer value, called process pid (process ID). While a process can fork one or more new child process by invoking fork(), these processes consist a process group. For example, if we enter the following pipe command line at the shell prompt,

```
[plinux root]# cat main.c | grep for | more
```

then each command in the line belongs to the same process group.

Process group is a set of one or more processes. As with the process, each process group has a process group indentifier gid (Group ID) which also is a non-negative integer value. In a process group, there is a process called group leader. Its pid equals to the process group gid. By calling function setpgid(), a process may participate an existing process group or create a new group. The concepts of process group has many usefull applications. The most popular one is we sending terminating signal (usually press Ctrl-C keys) on a terminal to the front end program, this will terminate all the processes in the group. For example, if we send interrupt signal to the command line above while it's running, three commands will be terminated at the same time.

Session is also a set which contains one or more process groups. In the normal case, all the processes running after a user logged in belong to one session and the login shell process is the session leader created when the user logs in. When the user logs out, all the processes belong to the session will be terminated[1]. System call setsid() is used to create a new session and is normally invoked by environment initializing program at login. See the description below. Thus, the relationship between process, process group and session is tightly associated as illustrated in Figure 4-4.

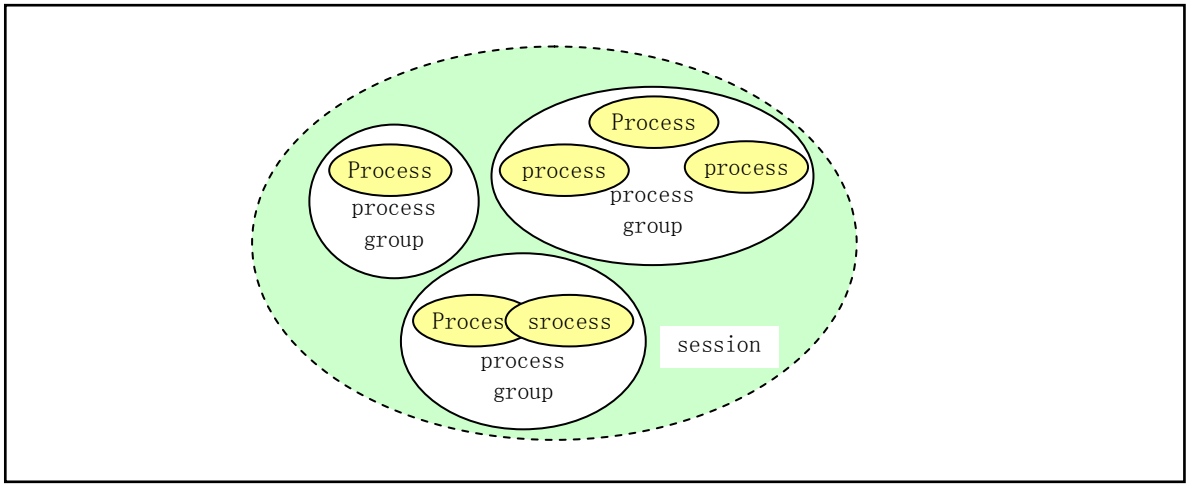

Figure 4-4 The relationship between process, process group and session

4.3 Environment initialization

After the kernel finished its initialization, the operating system needs to initialize the environment based on its actual configuration to achieve to a normal workable environment. In main.c program at L183 and L200, function init() start running shell program (/bin/sh) immediately which is not the normal case as in a system with login like the current Linux system. In order to provide login and multiuser function, A normal system is to execute a init.c program

[1] A process will stay running in background after user logs out if it ignores the SIGHUP signal.

used to continue to initialize the environment for the system. This program will fork a new child process for each terminal supported by the system based on the contents of the configuaration files in /etc/ directory and runs the terminal initializing program agetty (getty) in every child process. The getty will, then, show the message "login: " on the terminal screen. After the user entered his/her name, getty will execute the login program and replaced by it. The login program is mainly used to verify the password entered by the user and invoke the shell program and display the shell command line prompt finally. The relationship of these four program is illustrated in Figure 4-5.

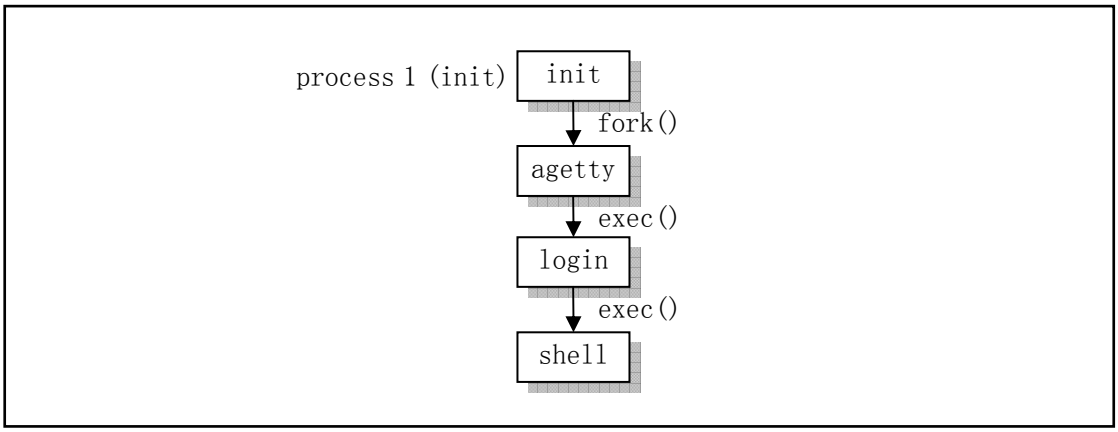

Figure 4-5 Programs relating environment initialization

Although these four programs are not belong to the scope of kernel, it is helpful to understand the kernel if we have some knowledge about them.

Program init executes the commands in the /etc/rc file and then forks child process for each terminal described in /etc/inittab file, each line for one terminal. Afterwards, it invokes wait(), waiting for the termination of any child process. While in each child process, it runs getty program. Whenever a child process ended, it will know the process and corresponding termainal by the return pid of wait() and forks a new child for the terminal again. Thus, each terminal device that permits login always has a corresponding process waiting to operate on it.

In the normal operating case, program init will assure that the getty is running to allow user log in and collecting orphan processes. Orphan process is a process which parent process was terminated or exited. In Linux system, all processes must belong to a single process tree, so the orphan processes must be adopted. When the system begins shutdown operation, init is responsible to kill all other processes, umount all filesystem and stop the CPU and etc.

The primary job of getty program is to configure terminal type, properties, speed and line discipline. It opens and initializes a tty port, showing a prompt message on the screen and waits user to enter a usename. It can only be run by supervisor. Usually, if file /etc/issue is available, getty will first display the contents of it and then prompts the user login message like "plinux login:" before reading the login name and finally executs the login program.

The login program is mainly used to request password of a user logging in. Based on the username, it obtains the line in the /etc/passwd file and invokes getpass() to prompt message "password: " and read in user's password (if a password is required for the username), encripts

the password and compares it with the encripted password in pw_passwd field in /etc/passwd file. If the passwd entered is invalid for several times, login program will exit running with error code 1 indicating the login procedure has failed. At this point, the wait() of the parent process (init process) returns with the pid of exited process. Thus the init process will fork a new child with information recorded and execute the getty program in the child again. The process above is then repeated.

If the password entered is correct, login program will configure the operating environment for the user. For example, setting the current work directory and etc. Afterwards, login process changes its user ID to the logged in user's ID and finally executes the shell program indicated in the /etc/passwd file.

4.4 Summary

From the analysis of the kernel 0.11 code above, we can construct a very simple root file system to run the Linux 0.11 system. The minimum needs are a MINIX filesystem which contains /etc/rc, /bin/sh, /dev/* and several directories like /home/, /home/root/.

From now on, we can use main.c as a main focus for reading subsequence chapters. The chapters need not read in order. If you are not familiar with memory paging, it is recommended that you read the chapter 10 first. In order to easily to understand the contents of subsequent chapters, we strongly request the reader to review the 32-bit protected mode mechinism again.