Chapter 11
File System

This chapter describes the implementation of a simple EXT2 file system [Card et al. 1995; Cao et al. 2007; EXT2 2001; EXT3 2015]. We choose the EXT2 file system for MTX mainly for two reasons. First, EXT2 is a simple file system, which is easy to understand but powerful enough for practical use. Second and more importantly, it is Linux compatible. Since we use Linux as the development platform of MTX, compatibility with Linux avoids unnecessary file conversions, which greatly simplifies the development of the MTX system. In this chapter, we first present an overview of file operations in an operating system. Next, we show how to format a disk image as an EXT2 file system. Then we describe the implementation of a simple EXT2 file system in detail.

11.1 File Operation Levels

File operations consist of five levels, from low to high, as shown in the following hierarchy.

(1) **Hardware Level:** File operations at hardware level include

fdisk: divide a hard disk or usb drive into partitions for file systems.
mkfs: format a disk partition to make it ready for file system.
fsck: check and repair file system.
defragmentation: compact files in a file system.

Most of these are system-oriented utility programs. An average user may never need them, but they are indispensable tools for creating and maintaining file systems.

(2) **File System Functions in OS Kernel:** Every operating system kernel provides support for basic file operations. The following lists some of these functions in a Unix-like system kernel, where the prefix k denotes kernel functions.

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_11

```
kmount(), kumount()           (mount/umount file systems)
kmkdir(), krmdir()            (make/remove directory)
kchdir(), kgetcwd()           (change directory, get CWD pathname)
klink(),  kunlink()           (hard link/unlink files)
kchmod(), kchown(), ktouch()  (change r|w|x permissions,owner,time)
kcreat(), kopen()             (create/open file for R,W,RW,APPEND)
kread(),  kwrite()            (read/write opened files)
klseek(); kclose()            (lseek/close file descriptors)
ksymlink(), kreadlink()       (create/read symbolic link files)
kstat(),  kfstat(), klstat()  (get file status/information)
kopendir(), kreaddir()        (open/read directories)
```

(3) **System Calls:** User mode programs use system calls to access kernel functions. As an example, the following program reads the second 1024 bytes of a file.

```
#include <fcntl.h>
main(int argc, char *argv[ ])   // run as a.out filename
{
   int fd, n;
   char buf[1024];
   if ((fd = open(argv[1], O_RDONLY)) < 0) // if open() fails
        exit(1);
   lseek(fd, 1024, SEEK_SET);   // lseek to byte 1024
   n = read(fd, buf, 1024);     // try to read 1024 bytes
   close(fd);
}
```

The functions open(), read(), lseek() and close() are C library functions. Each library function issues a system call, which causes the process to enter kernel mode to execute a corresponding kernel function, e.g. open goes to kopen(), read goes to kread(), etc. When the process finishes executing the kernel function, it returns to user mode with the desired results. Switch between user mode and kernel mode requires a lot of actions (and time). Data transfer between kernel and user spaces is therefore quite expensive. Although it is permissible to issue a read(fd, buf, 1) system call to read only one byte of data, it is not very wise to do so since that one byte would come with a terrific cost. Every time we have to enter kernel, we should do as much as we can to make the journey worthwhile. In the case of read/write files, the best way is to match what the kernel does. The kernel reads/writes files by block size, which ranges from 1KB to 8KB. For instance, in Linux, the default block size is 4KB for hard disks and 1KB for floppy disks. So each read/write system call should also try to transfer one block of data at a time.

(4) **Library I/O Functions:** System calls allow the user to read/write chunks of data, which are just a sequence of bytes. They do not know, nor care, about the meaning of the data. A user often needs to read/write individual chars, lines or data structure records, etc. With only system calls, a user mode program must do these operations from/to a buffer area by itself. Most users would consider this too inconvenient. The C library provides a set of standard I/O functions for convenience, as well as for run-time efficiency. Library I/O functions include:

```
FILE mode I/O: fopen(),fread();  fwrite(),fseek(),fclose(),fflush()
char mode I/O: getc(), getchar() ugetc(); putc(),putchar()
line mode I/O: gets(), fgets();  puts(), fputs()
formatted I/O: scanf(),fscanf(),sscanf(); printf(),fprintf(),sprintf()
```

With the exceptions of sscanf()/sprintf(), which read/write memory locations, all other library I/O functions are built on top of system calls, i.e. they ultimately issue system calls for actual data transfer through the system kernel.

(5) **User Commands:** Instead of writing programs, users may use Unix/Linux commands to do file operations. Examples of user commands are

mkdir, rmdir, cd, pwd, ls, link, unlink, rm, cat, cp, mv, chmod, etc.

Each user command is in fact an executable program (except cd), which typically calls library I/O functions, which in turn issue system calls to invoke the corresponding kernel functions. The processing sequence of a user command is either

```
   Command => Library I/O function => System call => Kernel Function
OR Command ======================== > System call => Kernel Function
```

(6) **Sh Scripts**: Although much more convenient than system calls, commands must be entered manually, which is tedious and time-consuming. Sh scripts are programs written in the sh programming language, which can be executed by the command interpreter sh. The sh language include all valid Unix commands. It also supports variables and control statements, such as if, do, for, while, case, etc. In practice, sh scripts are used extensively in Unix systems programming. In addition to sh, many other script languages, such as Perl and Tcl, are also in wide use.

11.2 File I/O Operations

Figure 11.1 shows the diagram of file I/O operations.

In Fig. 11.1, the upper part above the double line represents kernel space and the lower part represents user space of a process. The diagram shows the sequence of actions when a process read/write a file stream. Control flows are identified by the labels (1–10), which are explained below.

User Mode Operations

---------- User Mode Operations ----------

(1). A process in User mode executes

```
FILE *fp = fopen("file", "r"); or FILE *fp = fopen("file", "w");
```

which opens a file stream for READ or WRITE.

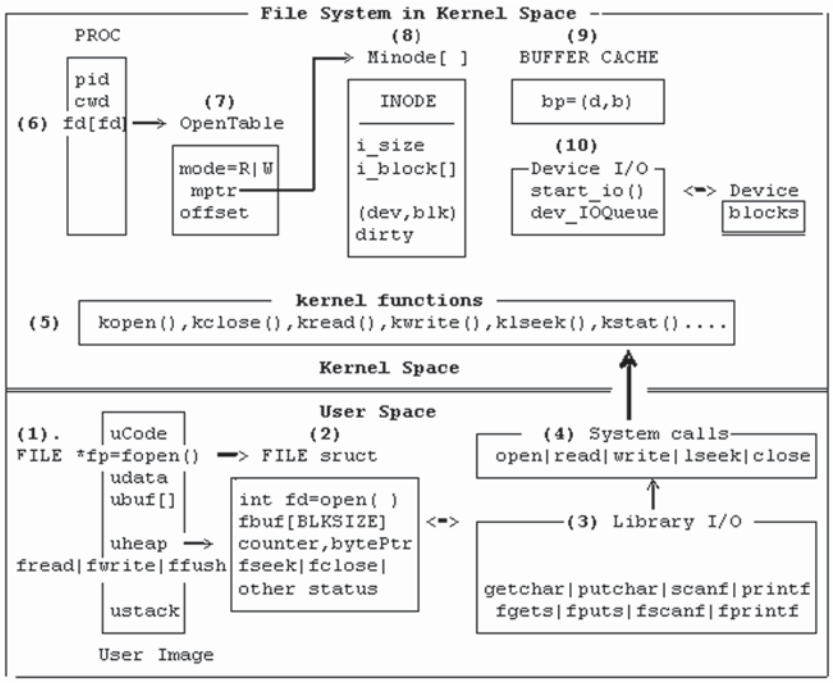


Fig. 11.1 File operation diagram

(2). fopen() creates a FILE structure in user (heap) space containing a file descriptor, fd, and a fbuf[BLKSIZE]. It issues a fd = open("file", flags = READ or WRITE) syscall to kopen() in kernel, which constructs an OpenTable to represent an instance of the opened file. The OpenTable's mptr points to the file's INODE in memory. For non-special files, the INODE's i_block array points to data blocks on the storage device. On success, fp points to the FILE structure, in which fd is the file descriptor returned by the open() syscall.

(3). fread(ubuf, size, nitem, fp): READ nitem of size each to ubuf by

. copy data from FILE structure's fbuf to ubuf, if enough, return;
. if fbuf has no more data, then execute (4a).

(4a). issue read(fd, fbuf, BLKSIZE) syscall to read a file block from kernel to fbuf, then copy data to ubuf until enough or file has no more data.

(4b). fwrite(ubuf, size, nitem, fp): copy data from ubuf to fbuf;

. if (fbuf has room): copy data to fbuf, return;
. if (fbuf is full) : issue write(fd, fbuf, BLKSIZE) syscall to write a block to kernel, then write to fbuf again.

Thus, fread()/fwrite() issue read()/write() syscalls to kernel, but they do so only if necessary and they transfer data in chunks of block size for better efficiency. Similarly, other Library I/O Functions, such as fgetc/fputc, fgets/fputs, fscanf/fprintf, etc. also operate on fbuf in the FILE structure, which is in user space.

========== **Kernel Mode Operations** ==========

(5).File system functions in kernel:

Assume read(fd, fbuf[ ], BLKSIZE) syscall of non-special file.

(6). In a read() syscall, fd is an opened file descriptor, which is an index in the running PROC's fd array, which points to an OpenTable representing the opened file.

(7). The OpenTable contains the files's open mode, a pointer to the file's INODE in memory and the current byte offset into the file for read/write. From the OpenTable's offset,

. Compute logical block number, lbk;
. Convert logical block number to physical block number, blk, via INODE.i_block array.

(8). Minode contains the in-memory INODE of the file. The INODE.i_block array contains pointers to physical disk blocks. A file system may use the physical block numbers to read/write data from/to the disk blocks directly, but these would incur too much physical disk I/O.

(9). In order to improve disk I/O efficiency, the OS kernel usually uses a set of I/O buffers as a cache memory to reduce the number of physical I/O. Details of I/O buffer management will be covered in Chap. 12.

(9a). For a read(fd, buf, BLKSIZE) syscall, determine the needed (dev, blk) number, then consult the I/O buffer cache to

```
.get a buffer = (dev, blk);
.if (buffer's data are invalid){
     start_io on buffer;
     wait for I/O completion;
 }
.copy data from buffer to fbuf;
.release the buffer to buffer cache;
```

(9b). For a write(fd, fbuf, BLKSIZE) syscall, determine the needed (dev, blk) number, then consult the I/O buffer cache to

```
.get a buffer = (dev, blk);
.write data to the I/O buffer;
.mark buffer as dataValid and DIRTY (for delay-write to disk);
.release the buffer to buffer cache;
```

(10): Device I/O: Physical I/O on the I/O buffers ultimately go through the device driver, which consists of start_io() in the upper-half and disk interrupt handler in the lower-half of the driver.

```
---------------  Upper-half of disk driver ----------------------
start_io(bp): //bp=a locked buffer in dev_list, opcode=R|W(ASYNC)
{
  1. enter bp into dev's I/O_queue;
  2. if (bp is FIRST in I/O_queue)
        issue I/O command to device;
}
----------------    Lower-half of disk driver -------------------
Device_Interrupt_Handler:
{
  bp = dequeue(first buffer from dev.I/O_queue);
  if (bp was READ){
     mark bp data VALID;
     wakeup/unblock waiting process on bp;
  }
  else      // bp was for delay write
     release bp into buffer cache;
  if (dev.I/O_queue NOT empty)
     issue I/O command for first buffer in dev.I/O_queue;
  }
```

11.3 EXT2 File System Specification

Chapter 2 contains a brief description of the EXT2 file system, which is sufficient for the following discussions. For more information, the reader may consult EXT2 file system documentations [Card et al. EXT2] for details.

11.4 A Simple mkfs Program

On the MTX Install CD, MTX.programs/mkfs.c is a C program, which formats a disk image as an EXT2 file system. For the sake of simplicity, we assume that the file system has only one disk block group. The maximal number of disk blocks is 8192. The program is intended to be compiled and run under Linux, but it can be adapted to any system environment which supports the write disk block operation. Under Linux, the program runs as follows.

mkfs device nblocks [ninodes]

where nblocks is the number of (1KB) blocks and niondes is the number of inodes. If ninodes is not specified, the program computes a default number of ninodes based on nblocks. The algorithm of mkfs is outlined below.

```
/*********** Algorithm of mkfs devive, nblocks, [ninode]*********/
1. open device for RW mode, write to last disk block;
2. if ninodes is not specified, compute ninodes=8*((nblocks/4)/8);
3. based on ninodes, compute number of inode blocks and used blocks.
4. create super block, write to block#1
5. create group descriptor 0, write to block#2
6. create blocks bitmap; write to blcok#3
7. create inodes bitmap; write to block#4
8. clear inodes blocks on disk; allocate a data block, create a root
   inode in memory with i_blokc[0]=allocated data block, write the
   root inode to #2 inode on disk.
9. initialize root inode's data block to contain . and .. entries;
   write data block to disk;
```

The basic technique used is to create the needed data structure in a 1KB buffer area in memory and then write it to a corresponding block of the disk image. The resulting disk image file can be used as a virtual disk.

11.5 Implementation of EXT2 File System

11.5.1 File System Organization

Figure 11.2 shows the internal organization of an EXT2 file system. The organization diagram is explained by the labels (1–5).

(1). is the PROC structure of the running process. Each PROC has a cwd, which points to the in-memory INODE of the PROC's Current Working Directory. It also has an array of file descriptors, fd[], which point to opened file instances.

(2). is the root pointer of the file system. It points to the in-memory root INODE. When the system starts, one of the devices is chosen as the root device, which must be a valid EXT2 file system. The root INODE (inode #2) of the root device is loaded into memory as the root (/) of the file system. This operation is known as "mount root file system".

(3). is an openTable entry. When a process opens a file, an entry of the PROC's fd array points to an openTable, which points to the in-memory INODE of the opened file.

(4). is an in-memory INODE. Whenever a file is needed, its INODE is loaded into a minode slot for reference. Since INODEs are unique, only one copy of each INODE can be in memory at any time. In the minode, (dev, ino) identify where the INODE came from, for writing the INODE back to disk if modified. The refCount field records the number of processes that are using the minode. The dirty field indicates whether the INODE has been modified. The mounted flag indicates whether the INODE has been mounted on and, if so, the mntabPtr points to the mount table entry of the mounted file system. The lock field is to ensure that an in-memory INODE can only be accessed by one process at a time, e.g. when modifying the INODE or during a read/write operation.

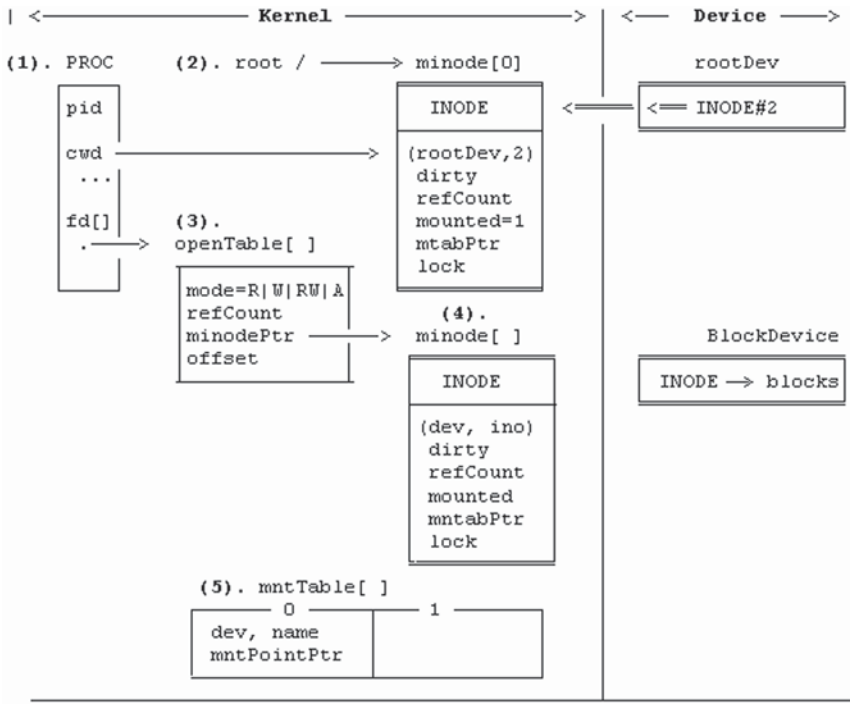


Fig. 11.2 EXT2 File system data structures

(5). is a table of mounted file systems. For each mounted file system, an entry in the mount table is used to record the mounted file system information. In the in-memory INODE of the mount point, the mounted flag is turned on and the mntabPtr points to the mount table entry. In the mount table entry, mntPointPtr points back to the in-memory INODE of the mount point. As will be shown later, these doubly-linked pointers allow us to cross mount points when traversing the file system tree. In addition, a mount table entry may also contain other information of the mounted file system, such as the device name, superblock, group descriptor and bitmaps, etc. for quick reference.

11.5.2 Files in the MTX/FS directory

In the MTX kernel source tree, the FS directory contains files which implement the EXT2 file system. The files are organized as follows.

```
------------------ Common files of FS ------------------------------
type.h  : EXT2 data structure types
global.c: global variables of FS
util.c    : common utility functions: getino(), iget(), iput(), search(), etc.
allocate_deallocate.c : inodes/blocks management functions
```

Implementation of the file system is divided into three levels. Each level deals with a distinct part of the file system. This makes the implementation process modular and easier to understand. Level-1 implements the basic file system tree. It contains the following files, which implement the indicated functions.

```
-------------------- Level-1 of FS ----------------------------
mkdir_creat.c        : make directory, create regular and special file
cd_pwd.c             :  change directory, get CWD path
rmdir.c              :  remove directory
link_unlink.c        : hard link and unlink files
symlink_readlink.c : symbolic link files
stat.c               : return file information
misc1.c              : access, chmod, chown, touch, etc.
-----------------------------------------------------------------
```

User level programs which use the level-1 FS functions include

mkdir, creat, mknod, rmdir, link, unlink, symlink, rm, ls, cd and pwd, etc.

Level-2 implements functions for reading/writing file contents.

```
---------------------- Level-2 of FS ------------------------------
open_close_lseek.c  : open file for READ|WRITE|APPEND, close file and lseek
read.c              : read from an opened file descriptor
write.c             : write to an opened file descriptor
opendir_readdir.c   : open and read directory
dev_switch_table    : read/write special files
block device I/O buffer management : I/O buffer management
```

Level-3 implements mount, umount and file protection.

```
---------------------- Level-3 of FS ------------------------------
mount_umount.c      : mount/umount file systems
file protection     : access permission checking
file-locking        : lock/unlock files
-----------------------------------------------------------------
```

11.5.3 Implementation of Level-1 FS

(1) type.h file: This file contains the data structure types of the EXT2 file system, such as superblock, group descriptor, inode and directory entry structures. In addition, it also contains the open file table, mount table, pipes and PROC structures and constants of the MTX kernel.

(2) global.c file: This file contains global variables of the MTX kernel. Examples of global variables are

```
MINODE minode[NMINODES];     // in memory INODEs
MOUNT  mounttab[NMOUNT];     // mount table
OFT    oft[NOFT];            // Opened file instance
```

(3) util.c file: This file contains utility functions of the file system. The most important utility functions are getino(), iget() and iput(), which are explained in more detail.

(3).1. u32 getino(int *dev, char *pathname): getino() returns the inode number of a pathname. While traversing a pathname the device number may change if the pathname crosses mounting point(s). The parameter dev is used to record the final device number. Thus, getino() essentially returns the (dev, ino) of a pathname. The function uses token() to break up pathname into component strings. Then it calls search() to search for the component strings in successive directory minodes.

(3).2. MINODE *iget(in dev, u32 ino): This function returns a pointer to the in-memory INODE of (dev, ino). The returned minode is unique, i.e. only one copy of the INODE exists in memory. In addition, the minode is locked for exclusive use until it is either released or unlocked.

(3).3. iput(MINODE *mip): This function releases and unlocks a minode pointed by mip. If the process is the last one to use the minode (refCount = 0), the INODE is written back to disk if it is dirty (modified).

(3).4. Use of getino()/iget()/iput(): In a file system, almost every operation begins with a pathname, e.g. mkdir pathname, cat pathname, etc. Whenever a pathname is needed, its inode must be loaded into memory for reference. The general pattern of using an inode is

```
. ino = getino(&dev, pathname);
. mip = iget(dev, ino);
.    use the mip->INODE, which may modify the INODE;
. iput(mip);
```

There are only a few exceptions to this usage pattern. For instance,

chdir: iget the new DIR minode but iput the old DIR minode.

open: iget the minode of a file, which is released when the file is closed.

mount: iget the minode of mountPoint, which is released later by umount.

In general, iget and iput should appear in pairs, like a pair of matching parentheses. We may rely on this usage pattern in the implementation code to ensure that every INODE is loaded and then released properly.

(3).5. Minodes Locking: Every minode has a lock field, which ensures that a minode can only be accessed by one process at a time, especially when modifying the INODE. Unix uses a busy flag and sleep/wakeup to synchronize processes accessing the same minode. In MTX, each minode has a lock semaphore with an initial

value 1. A process is allowed to access a minode only if it holds the semaphore lock. The reason for minodes locking is as follows.

Assume that a process Pi needs the inode of (dev, ino), which is not in memory. Pi must load the inode into a minode entry. The minode must be marked as (dev, ino) to prevent other processes from loading the same inode again. While loading the inode from disk Pi may wait for I/O completion, which switches to another process Pj. If Pj needs exactly the same inode, it would find the needed minode already exists. Without the lock, Pj would proceed to use the minode before it is even loaded in yet. With the lock, Pj must wait until the minode is loaded, used and then released by Pi. In addition, when a process read/write an opened file, it must lock the file's minode to ensure that each read/write operation is atomic.

(4). allocate_deallocate.c file: This file contains utility functions for allocating and deallocating minodes, inodes, disk blocks and open file table entries. It is noted that both inode and disk block numbers count from 1. Therefore, in the bitmaps bit i represents inode/block number i + 1.

(5). mount_root.c file

This file contains the mount_root() function, which is called during system initialization to mount the root file system. It reads the superblock of the root device to verify the device is a valid EXT2 file system. It loads the root INODE (ino = 2) into a minode and sets the root pointer to the root minode. Then it unlocks the root minode to allow all processes to access the root minode. A mount table entry is allocated to record the mounted root file system. Some key parameters on the root device, such as the starting blocks of the bitmaps and inodes table, are also recorded in the mount table for quick reference.

(6). mkdir_creat.c file

This file contains mkdir and creat functions for making directories and creating files, respectively. mkdir and creat are very similar, so they share some common code. Before discussing the algorithms of mkdir and creat, we first show how to insert/delete a DIR entry into/from a parent directory. Each data block of a directory contains DIR entries of the form

```
|ino rlen nlen name|ino rlen nlen name| ...
```

where name is a sequence of nlen chars without a terminating NULL byte. Since each DIR entry begins with a u32 inode number, the rec_len of each DIR entry is always a multiple of 4 (for memory alignment). The last entry in a data block spans the remaining block, i.e. its rec_len is from where the entry begins to the end of block. In mkdir and creat, we assume the following.

(a). A DIR file has at most 12 direct blocks. This assumption is reasonable since, with 1KB block size and an average file name of 16 chars, a DIR can contain more than 500 file names. We may assume that no user would put that many entries in a directory.

(b). Once allocated, a DIR's data block is kept for reuse even if it becomes empty.

With these assumptions, the insertion and deletion algorithms are as follows.

```
/************* Algorithm of Insert_dir_entry *****************/
(1). need_len = 4*((8+name_len+3)/4); // new entry need length
(2). for each existing data block do {
        if (block has only one entry with inode number==0)
           enter new entry as first entry in block;
        else{
(3).       go to last entry in block;
           ideal_len = 4*((8+last_entry's name_len+3)/4);
           remain = last entry's rec_len - ideal_len;
           if (remain >= need_len){
              trim last entry's rec_len to ideal_len;
              enter new entry as last entry with rec_len = remain;
           }
(4).       else{
              allocate a new data block;
              enter new entry as first entry in the data block;
              increase DIR's size by BLKSSIZE;
           }
        }
        write block to disk;
     }
(5). mark DIR's minode modified for write back;

/************* Algorithm of Delete_dir_entry (name) *************/
(1). search DIR's data block(s) for entry by name;
(2). if (entry is the only entry in block)
        clear entry's inode number to 0;
     else{
(3).    if (entry is last entry in block)
            add entry's rec_len to predecessor entry's rec_len;
(4).    else{ // entry in middle of block
            add entry's rec_len to last entry's rec_len;
            move all trailing entries left to overlay deleted entry;
        }
     }
(5). write block back to disk;
```

Note that in the Delete_dir_entry algorithm, an empty block is not deallocated but kept for reuse. This implies that a DIR's size will never decrease. Alternative schemes are listed in the Problem section as programming exercises.

11.5.3.1 Mkdir-creat-mknod

mkdir creates an empty directory with a data block containing the default. and.. entries. The algorithm of mkdir is

```
/********* Algorithm of mkdir *********/
int mkdir(char *pathname)
{
 1. if (pathname is absolute) dev = root    ->dev;
       else                        dev = PROC's cwd->dev
 2. divide pathname into dirname and basename;
 3. // dirname must exist and is a DIR:
    pino = getino(&dev, dirname);
    pmip = iget(dev, pino);
    check pmip    ->INODE is a DIR
 4. // basename must not exist in parent DIR:
       search(pmip, basename) must return 0;
 5. call kmkdir(pmip, basename) to create a DIR;
       kmkdir() consists of 4 major steps:
       5-1. allocate an INODE and a disk block:
              ino = ialloc(dev); blk = balloc(dev);
              mip = iget(dev,ino);  // load INODE into an minode
       5-2. initialize mip->INODE as a DIR INODE;
              mip->INODE.i_block[0] = blk; other i_block[ ] are 0;
              mark minode modified (dirty);
              iput(mip);  // write INODE back to disk
       5-3. make data block 0 of INODE to contain . and .. entries;
            write to disk block blk.
       5-4. enter_child(pmip, ino, basename); which enters
            (ino, basename) as a DIR entry to the parent INODE;
 6. increment parent INODE's links_count by 1 and mark pmip dirty;
    iput(pmip);
}
```

Creat creates an empty regular file. The algorithm of creat is

```
/***************** Algorithm of creat ()*************************/
creat(char * pathname)
{
   This is similar to mkdir() except
   (1). the INODE.i_mode field is set to REG file type, permission
            bits set to 0644 for rw-r--r--, and
   (2). no data block is allocated for it, so the file size is 0.
   (3). Do not increment parent INODE's links_count
}
```

It is noted that the above creat algorithm differs from that in Unix/Linux. The new file's permissions are set to 0644 by default and it does not open the file for WRITE mode and return a file descriptor. In practice, creat is rarely used as a stand-alone syscall. It is used internally by the open() function, which may create a file, open it for WRITE and return a file descriptor. The open operation will be described later.

mknod creates a special file which represents either a char or block device with a device number=(major, minor). The algorithm of mknod is

```
/***************** Algorithm of mknod () **********************/
 mknod(char *name, int type, int device_number)
 {
   This is similar to creat() except
   (1). the default parent directory is /dev;
   (2). INODE.i_mode is set to CHAR or BLK file type;
   (3). INODE.I_block[0] contains device_number=(major, minor);
 }
```

11.5.3.2 Chdir-getcwd-stat

Each process has a Current Working Directory (CWD), which points to the CWD minode of the process in memory. chdir(pathname) changes the CWD of a process to pathname. getcwd() returns the absolute pathname of CWD. stat() returns the status information of a file in a STAT structure. The algorithm of chdir() is

```
/********** Algorithm of chdir ************/
 int chdir(char *pathname)
 {
   (1). get INODE of pathname into a minode;
   (2). verify it is a DIR;
   (3). change running process CWD to minode of pathname;
   (4). iput(old CWD); return 0 for OK;
 }
```

getcwd() is implemented by recursion. Starting from CWD, get the parent INODE into memory. Search the parent INODE's data block for the name of the current directory and save the name string. Repeat the operation for the parent INODE until the root directory is reached. Construct an absolute pathname of CWD on return. Then copy the absolute pathname to user space. stat(pathname, STAT *st) returns the information of a file in a STAT structure. The algorithm of stat is

```
/********* Algorithm of stat *********/
 int stat(char *pathname, STAT *st)
 {
   (1). get INODE of pathname into a minode;
   (2). copy (dev, ino) of minode to (st_dev, st_ino) of the STAT
        structure in user space;
   (3). copy other fields of INODE to STAT structure in user space;
   (4). iput(minode); retrun 0 for OK;
 }
```

11.5.3.3 Rmdir

As in Unix/Linux, in order to rm a DIR, the directory must be empty, for the following reasons. First, removing a non-empty directory implies removing all the files

and subdirectories in the directory. Although it is possible to implement a rrmdir() operation, which recursively removes an entire directory tree, the basic operation is still to remove one directory at a time. Second, a non-empty directory may contain files that are actively in use, e.g. opened for read/write, etc. Removing such a directory is clearly unacceptable. Although it is possible to check whether there are any active files in a directory, it would incur too much overhead in the kernel. The simplest way out is to require that a directory must be empty in order to be removed. The algorithm of rmdir() is

```
/********** Algorithm of rmdir *************/
 rmdir(char *pathname)
 {
   1. get in-memory INODE of pathname:
      ino = getino(&de, pathanme);
      mip = iget(dev,ino);
   2. verify INODE is a DIR (by INODE.i_mode field);
      minode is not BUSY (refCount = 1);
      DIR is empty (traverse data blocks for number of entries = 2);
   3. /* get parent's ino and inode */
      pino = findino(); //get pino from .. entry in INODE.i_block[0]
      pmip = iget(mip->dev, pino);
   4. /* remove name from parent directory */
      findname(pmip, ino, name); //find name from parent DIR
      rm_child(pmip, name);
   5. /* deallocate its data blocks and inode */
      truncat(mip);  // deallocate INODE's data blocks
   6. deallocate INODE
      idalloc(mip->dev, mip->ino); iput(mip);
   7. dec parent links_count by 1;
      mark parent dirty; iput(pmip);
   8. return 0 for SUCCESS.
 }
```

11.5.3.4 Link-unlink

The link_unlikc.c file implements link and unlink. link(old_file, new_file) creates a hard link from new_file to old_file. Hard links can only be to regular files, not DIRs, because linking to DIRs may create loops in the file system name space. Hard link files share the same inode. Therefore, they must be on the same device. The algorithm of link is

```
/********* Algorithm of link ********/
link(old_file, new_file)
{
1. // verify old_file exists and is not DIR;
   oino = getino(&odev, old_file);
   omip = iget(odev, oino);
   check file type (cannot be DIR).
2. // new_file must not exist yet:
   nion = get(&ndev, new_file) must return 0;
   ndev of dirname(newfile) must be same as odev
3. // creat entry in new_parent DIR with same ino
   pmip -> minode of dirname(new_file);
   enter_name(pmip, omip->ino, basename(new_file));
4. omip->INODE.i_links_count++;
   omip->dirty = 1;
   iput(omip);
   iput(pmip);
}
```

unlink decrements the file's links_count by 1 and deletes the file name from its parent DIR. When a file's links_count reaches 0, the file is truly removed by deallocating its data blocks and inode. The agorithm of unlink() is

```
/***********  Algorithm of unlink *************/
unlink(char *filename)
{
  1. get filenmae's minode:
     ino = getino(&dev, filename);
     mip = iget(dev, ino);
     check it's a REG or SLINK file
  2. // remove basename from parent DIR
     rm_child(pmip, mip->ino, basename);
     pmip->dirty = 1;
     iput(pmip);
  3. // decrement INODE's link_count
     mip->INODE.i_links_count--;
     if (mip->INODE.i_links_count > 0){
     mip->dirty = 1; iput(mip);
          }
  4. if (!SLINK file)        // assume:SLINK file has no data block
     truncate(mip); // deallocate all data blocks
     deallocate INODE;
     iput(mip);
}
```

11.5.3.5 Symlink-readlink

symlink(old_file, new_file) creates a symbolic link from new_file to old_file. Unlike hard links, symlink can link to anything, including DIRs or files not on the same device. The algorithm of symlink is

```
Algorithm of symlink(old_file, new_file)
{
  1. check: old_file must exist and new_file not yet exist;
  2. create new_file; change new_file to SLINK type;
  3. // assume length of old_file name <= 60 chars
     store old_file name in newfile's INODE.i_block[ ] area.
     mark new_file's minode dirty;
     iput(new_file's minode);
  4. mark new_file parent minode dirty;
     put(new_file's parent minode);
}
```

readlink(file, buffer) reads the target file name of a SLINK file and returns the length of the target file name. The algorithm of readlink() is

```
Algorithm of readlink (file, buffer)
{
  1. get file's INODE into memory; verify it's a SLINK file
  2. copy target filename in INODE.i_block into a buffer;
  3. return strlen((char *)mip->INODE.i_block);
}
```

11.5.3.6 Other Level-1 Functions

Other level-1 functions include access, chmod, chown, touch, etc. The operations of all such functions are of the same pattern:

```
(1). get the in-memory INODE of a file by
         ino = getinod(&dev, pathname);
         mip = iget(dev,ino);
(2). get information from the INODE or modify the INODE;
(3). if INODE is modified, mark it DIRTY for write back;
(4). iput(mip);
```

11.5.4 Implementation of Level-2 FS

Level-2 of FS implements read/write operations of file contents. It consists of the following functions: open, close, lseek, read, write, opendir and readdir.

11.5.4.1 Open-close-lseek

The file open_close_lseek.c implements open(), close() and lseek(). The system call

int open(char *filename, int flags);

opens a file for read or write, where flags = 0|1|2|3|4 for R|W|RW|APPEND, respectively. Alternatively, flags can also be specified as one of the symbolic con-

stants O_RDONLY, O_WRONLY, O_RDWR, which may be bitwise or-ed with file creation flags O_CREAT, O_APPEND, O_TRUNC. These symbolic constants are defined in type.h. On success, open() returns a file descriptor for subsequent read()/write() system calls. The algorithm of open() is

```
/**************  Algorithm of open() ***********/
int open(file, flags)
{
 1. get file's minode:
    ino = getino(&dev, file);
    if (ino==0 && O_CREAT){
    creat(file); ino = getino(&dev, file);
    }
    mip = iget(dev, ino);
 2. check file INODE's access permission;
    for non-special file, check for incompatible open modes;
 3. allocate an openTable entry;
    initialize openTable entries;
    set byteOffset = 0 for R|W|RW; set to file size for APPEND mode;
 4. Search for a FREE fd[ ] entry with the lowest index fd in PROC;
    let fd[fd]point to the openTable entry;
 5. unlock minode;
    return fd as the file descriptor;
}
```

Figure 11.3 show the data structure created by open(). In the figure, (1) is the PROC structure of the process that calls open(). The returned file descriptor, fd, is the index of the fd[ ] array in the PROC structure. The contents of fd[fd] points to a OFT, which points to the minode of the file. The OFT's refCount represents the number of processes which share the same instance of an opened file. When a process opens a file, the refCount in the OFT is set to 1. When a process forks, the child process

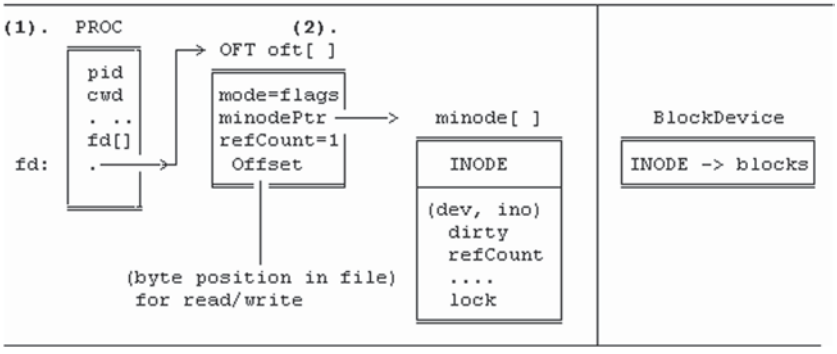


Fig. 11.3 Data Structures of open()

inherits all the opened file descriptors of the parent, which increments the refCount of every shared OFT by 1. When a process closes a file descriptor, it decrements the OFT's refCount by 1, etc. The OFT's offset is a conceptual pointer to the current byte position in the file for read/write. It is initially set to 0 for R|R|RW or to file size for APPEND.

In MTX, lseek(fd, position) sets the offset in the OFT of an opened file descriptor to the byte position relative to the beginning of the file. Once set, the next read/write begins from the current offset position. The algorithm of lseek () is trivial. For files opened for READ, it only checks the position value to ensure it's within the bounds of [0, file_size]. If fd is a regular file opened for WRITE, it allows the byte offset to go beyond the current file size but does not allocate any disk block. Disk blocks will be allocated when data are actually written to the file. The algorithm of closing a file descriptor is

```
/*************  Algorithm of close()  *****************/
int close(int fd)
{
  (1). check fd is a valid opened file descriptor;
  (2). if (PROC's fd[fd] != 0){
  (3).    if (openTable's mode == READ/WRITE PIPE)
             return close_pipe(fd); // close pipe descriptor;
  (4).    if (--refCount == 0){ // if last process using this OFT
             lock(minodeptr);
             iput(minode);      // release minode
             }
       }
  (5). clear fd[fd] = 0;        // clear fd[fd] to 0
  (6). return SUCCESS;
 }
```

11.5.4.2 Read Regular Files

The system call int read(int fd, char buf[ ], int nbytes); reads nbytes from an opened file descriptor into a buffer area in user space. read() invokes kread() in kernel, which implements the read system call. The algorithm of kread() is

```
/**************** Algorithm of kread() in kernel ****************/
int kread(int fd, char buf[ ], int nbytes, int space) //space=K|U
{
 (1). validate fd; ensure oft is opened for READ or RW;
 (2). if (oft.mode = READ_PIPE)
         return read_pipe(fd, buf, nbytes);
 (3). if (minode.INODE is a special file)
         return read_special(device,buf,nbytes);
 (4). (regular file):
         return read_file(fd, buf, nbytes, space);
}

/*************** Algorithm of read regular files ***************/
int read_file(int fd, char *buf, int nbytes, int space)
{
(1). lock minode;
(2). count = 0;          // number of bytes read
     compute bytes available in file: avil = fileSize - offset;
(3). while (nbytes){
       compute logical block: lbk   = offset / BLKSIZE;
       start byte in block:   start = offset % BLKSIZE;
(4).   convert logical block number, lbk, to physical block number,
       blk, through INODE.i_block[ ] array;
(5).   read_block(dev, blk, kbuf); // read blk into kbuf[BLKSIZE];
       char *cp = kbuf + start;
       remain = BLKSIZE - start;
(6)    while (remain){// copy bytes from kbuf[ ] to buf[ ]
          (space)? put_ubyte(*cp++, *buf++) : *buf++ = *cp++;
          offset++; count++;           // inc offset, count;
          remain--; avil--; nbytes--;  // dec remain, avil, nbytes;
          if (nbytes==0 || avail==0)
              break;
       }
     }
(7). unlock minode;
(8). return count;
}
```

The algorithm of read_file() can be best explained in terms of Fig. 11.4. Assume that fd is opened for READ. The offset in the OFT points to the current byte position in the file from where we wish to read nbytes. To the kernel, a file is just a sequence of contiguous bytes, numbered from 0 to fileSize-1. As Fig. 11.4 shows, the current byte position, offset, falls in a logical block, lbk = offset/ BLKSIZE, the byte to start read is start = offset % BLKSIZE and the number of bytes remaining in the logical block is remain = BLKSIZE—start. At this moment, the file has avail = fileSize—offset bytes still available for read. These numbers are used in the read_file algorithm. In MTX, block size is 1 KB and files have at most double indirect blocks.

The algorithm of converting logical block number to physical block number for read is

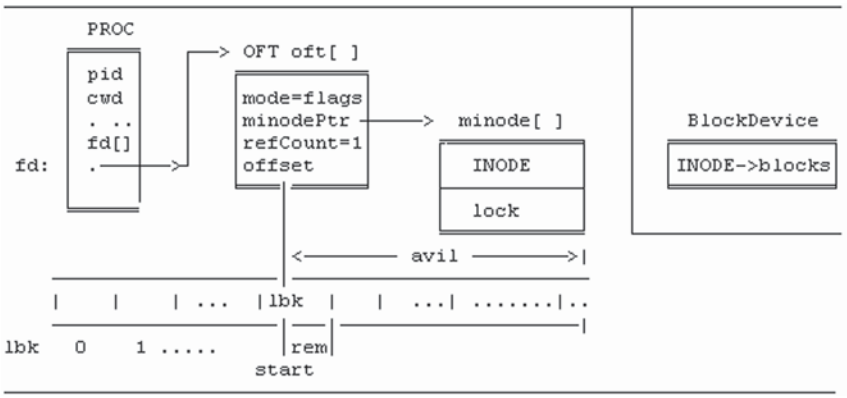


Fig. 11.4 Data Structures for read_file()

```
/* Algorithm of Converting Logical Block to Physical Block */
u32 map(INODE, lbk){                  // convert lbk to blk
    if (lbk < 12)                     // direct blocks
        blk = INODE.i_block[lbk];
    else if (12 <= lbk < 12+256){ // indirect blocks
        read INODE.i_block[12] into u32 ibuf[256];
        blk = ibuf[lbk-12];
    }
    else{                             // doube indirect blocks
        read INODE.i_block[13] into u32 dbuf[256];
        lbk-=(12+256);
        dblk=dbuf[lbk / 256];
        read dblk into dbuf[ ];
        blk =dbuf[lbk % 256];
    }
    return blk;
}
```

11.5.4.3 Write Regular Files

The system call int write(int fd, char ubuf[ ], int nbytes) writes nbytes from ubuf in user space to an opened file descriptor and returns the actual number of bytes written. write() invokes kwrite() in kernel, which implements the write system call. The algorithm of kwrite() is

```
/************** Algorithm of kwrite () in kernel ***********/
int kwrite(int fd, char *ubuf, int nbytes)
{
 (1). validate fd; ensure OFT is opened for write;
 (2). if (oft.mode = WRITE_PIPE)
         return write_pipe(fd, buf, nbytes);
 (3). if (minode.INODE is a special file)
         return write_special(device,buf.nbytes);
 (4). return write_file(fd, ubuf, nbytes);
}
```

The algorithm of write_file() can be best explained in terms of Fig. 11.5.

In Fig. 11.5, the offset in the OFT is the current byte position in the file for write. As in read_file(), it first computes the logical block number, lbk, the start byte position and the number of bytes remaining in the logical block. It converts the logical block to physical block through the file's INODE.i_block array. Then it reads the physical block into a buffer, writes data to it and writes the buffer back to disk. The following shows the write_file() algorithm.

```
/************** Algorithm of write regular file ***************/
int write_file(int fd, char *ubuf, int nbytes)
{
(1). lock minode;
(2). count = 0;          // number of bytes written
(3). while (nbytes){
        compute logical block: lbk = oftp->offset / BLOCK_SIZE;
        compute start byte:  start = oftp->offset % BLOCK_SIZE;
(4).    convert lbk to physical block number, blk;
(5).    read_block(dev, blk, kbuf); //read blk into kbuf[BLKSIZE];
        char *cp = kbuf + start; remain = BLKSIZE - start;
(6)     while (remain){  // copy bytes from kbuf[ ] to ubuf[ ]
            put_ubyte(*cp++, *ubuf++);
            offset++;  count++;     // inc offset, count;
            remain --; nbytes--;    // dec remain, nbytes;
            if (offset > fileSize) fileSize++; // inc file size
            if (nbytes <= 0) break;
        }
(7).    wrtie_block(dev, blk, kbuf);
     }
(8). set minode dirty = 1; // mark minode dirty for iput()
     unlock(minode);
     return count;
}
```

The algorithm of converting logical block to physical block for write is similar to that of read, except for the following difference. During write, the intended data block may not exist. If a direct block does not exist, it must be allocated and recorded in the INODE. If the indirect block does not exist, it must be allocated and initialized to 0. If an indirect data block does not exist, it must be allocated

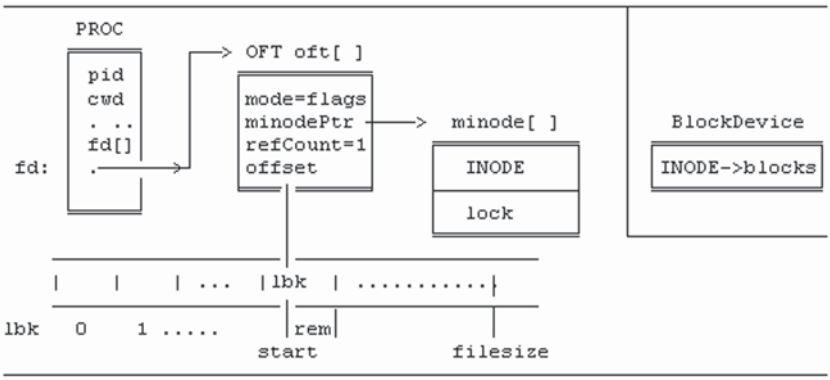


Fig. 11.5 Data structures for write_file()

and recorded in the indirect block, etc. The reader may consult the write.c file for details.

11.5.4.4 Read-Write Special Files

In kread() and kwrite(), read/write pipes and special files are treated differently. Read/write pipes are implemented in the pipe mechanism in the MTX kernel. Here we only consider read/write special files. Each special file has a file name in the /dev directory. The file type in a special file's inode is marked as special, e.g. 0060000 = block device, 0020000 = char device, etc. Since a special file does not have any disk block, i_block[0] of its INODE stores the device's (major, minor) number, where major = device type and minor = unit of that device type. For example,/dev/fd0 = (2,0),/dev/tty0 = (4,0) and /dev/ttyS1 = (5,1), etc. The major device number is an index in a device switch table, dev_sw[ ], which contains pointers to device driver functions, as in

```
struct dev_sw {
    int (*dev_read)();
    int (*dev_write)();
} dev_sw[];
```

Assume that int nocall(){ } is an empty function, and are device driver functions. The device switch table is set up to contain the driver function pointers.

```
struct dev_sw dev_sw[ ] =
{ //  read            write
  //--------         --------
     nocall,         nocall,        // 0 /dev/null
     nocall,         nocall,        // 1 kernel memory
     fd_read,        fd_write,      // 2 FD
     hd_read,        hd_write,      // 3 HD
     console_read,   console_write, // 4 console
     serial_read,    serial_write,  // 5 serial ports
     nocall,         printer_write  // 6 printer
};
```

Then read/write a special file becomes

(1). get special file's (major, minor) number from INODE.i_block[0];
(2). return (*dev_sw[major].dev_read) (minor, parameters); // READ
OR return (*dev_sw[major].dev_write)(minor, parameters); // WRITE

(2). invokes the corresponding device driver function, passing as parameters the minor device number and other parameters as needed. The device switch table is a standard technique used in all Unix-like systems. It not only makes the I/O subsystem structure clear but also greatly reduces the read/write code size. Hopefully, this also helps answer the question from many curious students, namely why the device number of floppy drives is 2 and that of hard drives is 3, etc. It's because the original Unix designers set up their device switch table this way, and everyone else has been a copycat ever since.

11.5.4.5 Opendir-readdir

Unix considers everything as a file. Therefore, we should be able to open a DIR for read just like a regular file. From a technical point of view, there is no need for a separate set of opendir() and readdir() functions. However, different Unix systems may have different file systems. It may be difficult for users to interpret the contents of a DIR file. For this reason, POSIX specifies opendir and readdir operations, which are independent of file systems. Support for opendir is trivial; it's the same open system call, but readdir() has the form

struct dirent *ep = readdir(DIR *dp);

which returns a pointer to a dirent structure on each call. This can be implemented in user space as a library I/O function. Since MTX does not have user-level library stream I/O, we shall implement opendir() and readir() as system calls.

```
int opendir(pathaname)
{   return open(pathname, O_RDONLY|O_DIR); }
```

where O_DIR is a bit pattern for opening the file as a DIR. In the open file table, the mode field contains the O_DIR bit.

```
int readdir(int fd, struct udir *dp) // struct udir{DIR; name[256]};
{
   // same as read() in kernel except:
   use the current byte offset in OFT to read the next DIR record;
   copy the DIR record into *udir in Umode;
   advance offset by DIR entry's rec_len;
}
```

User mode programs must use the readdir(fd, struct udir *dir) system call instead of the readdir(DP) call.

11.5.5 Implementation of Level-3 FS

Level-3 of FS implements mount and umount of file systems and file protection.

11.5.5.1 Mount-umount

The mount command, mount filesys mount_point, mounts a file system to a mount_point directory. It allows the file system to include other file systems as parts of an existing file system. The data structures used in mount are the MOUNT table and the in-memory minode of the mount_point directory. The algorithm of mount is

```
/********* Algorithm of mount **********/
mount()  // Usage: mount [filesys mount_point]
{
1. If no parameter, display current mounted file systems;
2. Check whether filesys is already mounted:
   The MOUNT table entries contain mounted file system (device) names
   and their mounting points. Reject if the device is already mounted.
   If not, allocate a free MOUNT table entry.
3. filesys is a special file with a device number dev=(major,minor).
   Read filesys' superblock to verify it is an EXT2 FS.
4. find the ino, and then the minode of mount_point:
        call ino = get_ino(&dev, pathname);  to get ino:
        call mip = iget(dev, ino); to load its inode into memory;
5. Check mount_point is a DIR and not busy, e.g. not someone's CWD.
6. Record dev and filesys name in the MOUNT table entry;
   also, store its ninodes, nblocks, etc. for quick reference.
7. Mark mount_point's minode as mounted on (mounted flag = 1) and let
   it point at the MOUNT table entry, which points back to the
   mount_point minode.
}
```

The operation Umount filesys detaches a mounted file system from its mounting point, where filesys may be either a special file name or a mounting point directory name. The algorithm of umount is

```
/****************** Algorithm of umount ************************/
umount(char *filesys)
{
 1. Search the MOUNT table to check filesys is indeed mounted.
 2. Check (by checking all active minode[].dev) whether any file is
    active in the mounted filesys; If so, reject;
 3. Find the mount_point's in-memory inode, which should be in memory
    while it's mounted on. Reset the minode's mounted flag to 0; then
    iput() the minode.
 4. return SUCCESS;
}
```

11.5.5.2 Implications of mount

While it is easy to implement mount and umount, there are implications. With mount, we must modify the get_ino(&dev, pathname) function to support crossing mount points. Assume that a file system, newfs, has been mounted on the directory /a/b/c. When traversing a pathname, mount point crossing may occur in both directions.

(1) Downward traversal: When traversing the pathname /a/b/c/x, once we reach the minode of /a/b/c, we should see that the minode has been mounted on (mounted flag = 1). Instead of searching for x in the INODE of /a/b/c, we must

- . Follow the minode's mountTable pointer to locate the mount table entry.
- . From the newfs's dev number, get its root (ino = 2) INODE into memory.
- . Then continue search for x under the root INODE of newfs.

(2) Upward traversal: Assume that we are at the directory /a/b/c/x and traversing upward, e.g. cd ../../, which will cross the mount point /a/b/c. When we reach the root INODE of the mounted file system, we should see that it is a root directory (ino = 2) but its dev number differs from that of the real root, so it is not the real root yet. Using its dev number, we can locate its mount table entry, which points to the mounted minode of /a/b/c/. Then, we switch to the minode of /a/b/c/ and continue the upward traversal. Thus, crossing mount point is like a monkey or squirrel hoping from one tree to another and then back.

11.5.5.3 File Protection

In Unix, file protection is by permission checking. Each file's INODE has an i_mode field, in which the low 9 bits are for file permissions. The 9 permission bits are

```
owner  group  other
-----  -----  -----
r w x  r w x  r w x
------ ------ -----
```

where the first 3 bits apply to the owner of the file, the second 3 bits apply to users in the same group as the owner and the last 3 bits apply to all others. For directories, the x bit indicates whether a process is allowed to go into the directory. Each process has a uid and a gid. When a process tries to access a file, the file system checks the process uid and gid against the file's permission bits to determine whether it is allowed to access the file with the intended mode of operation. If the process does not have the right permission, it is not allowed to access the file. For the sake of simplicity, MTX ignores gid. It uses only the process uid to check for file access permission.

11.5.5.4 Real and Effective uid

In Unix, a process has a real uid and an effective uid. The file system checks the access rights of a process by its effective uid. Under normal conditions, the effective uid and real uid of a process are identical. When a process executes a setuid program, which has the setuid bit (bit 11) in the file's i_mode field turned on, the process' effective uid becomes the uid of the program. While executing a setuid program, the process effectively becomes the owner of the program. For example, when a process executes the mail program, which is a setuid program owned by the superuser, it can write to a mail file of another user. When a process finishes executing a setuid program, it reverts back to the real uid. For simplicity reasons, MTX does not yet support effective uid. Permission checking is based on real uid.

11.5.5.5 File Locking

File locking is a mechanism which allows a process to set locks on a file, or parts of a file to prevent race conditions when updating files. File locks can be either shared, which allows concurrent reads, or exclusive, which enforces exclusive write. File locks can also be mandatory or advisory. For example, Linux supports both shared and exclusive files locks but file locking is only advisory. In Linux, file locks can be set by the fcntl() system call and manipulated by the flock() system call. In MTX, file locking is enforced only in the open() syscall of non-special files. When a process tries to open a non-special file, the intended mode of operation is checked for compatibility. The only compatible modes are READs. If a file is already opened for updating mode, i.e. W|RW|APPEND, it cannot be opened again. This does not apply to special files, e.g. terminals. A process may open its terminal multiple times even if the modes are incompatible. This is because access to special files is ultimately controlled by device drivers.

11.6 Extensions of MTX File System

The simple EXT2 FS in MTX uses 1KB block size and has only 1 disk block group. It can be extended easily as follows.

(1) Multiple groups: The size of group descriptor is 32 bytes. With 1KB block size, a block may contain 1024/32 = 32 group descriptors. With 32 groups, the FS size can be extended to 32*8 = 256 MB.

(2) 4KB block size: With 4KB block size and only one group, the FS size would be 4*8 = 32 MB. With one group descriptor block, the FS may have 128 groups, which extend the FS size to 128*32 = 4 GB. With 2 group descriptor blocks, the FS size would be 8 GB, etc. Most of the extensions are straightforward, which are suitable topics for programming projects.

(3) Pipe files: It's possible to implement pipes as regular files, which obey the read/write protocol of pipes. The advantages of this scheme are: it unifies pipes and file inodes and it allows named pipes, which can be used by unrelated processes. In order to support fast read/write operations, pipe contents should be in memory, such as a RAMdisk. If desired, the reader may implement named pipes as FIFO files.

Problems

1. In the read_file algorithm of Sect. 11.5.2, data can be read from either user space or kernel space.

   (1). Justify why it is necessary to read data to kernel space.
   (2). In the inner loop of the read_file algorithm, data are transferred one byte at a time for clarity. Optimize the inner loop by transferring chunks of data at a time (HINT: minimum of data remaining in the block and available data in the file).

2. Modify the write-file algorithm in Sect. 11.5.3 to allow

   (1). Write data from kernel space, and
   (2). Optimize data transfer by copying chunks of data.

3. Consider the cp f1 f2 operation, which copies file f1 to file f2.

   (1). Design an algorithm for the cp program.
   (2). What if f1 and f2 are the same file?
   (3). With hard links, two filenames may refer to the same file. How to determine whether two filenames are the same file?

4. Assume: dir1 and dir2 are directories. cpd2d dir1 dir2 recursively copies dir1 into dir2.

   (1). Write C code for the cpd2d program.
   (2). What if dir1 contains dir2, e.g. cpd2d /a/b /a/b/c/d?
   (3). How to determine whether dir1 contains dir2?

5. Modify the Delete_dir_entry algorithm as follows. If the deleted entry is the only entry in a data block, deallocate the data block and compact the DIR INODE's data block array. Modify the Insert_dir_entry algorithm accordingly and implement the new algorithms in the MTX file system.
5. In EXT2, the algorithms of inserting/deleting DIR entries amount to applying memory compaction to disk blocks, so that each data block of a directory has only one hole of maximal size at the end of the block. Consider the following Delete_dir_entry algorithm.

```
Delete_dir_entry // delete an entry from a DIR's data block
{
 (1). search DIR's data block(s) for the entry;
 (2). clear entry's inode number to 0;
 (3). if (not first entry in block)
         add entry's rec_len to predecessor entry's rec_len;
}
```

In the algorithm, if the entry to be deleted is the first entry in a data block, clearing its inode number to 0 makes the entry invalid. If it is not the first entry in the block, absorbing its rec_len into its predecessor effectively hides the deleted entry, making it invisible. In both cases, the deleted space is not wasted since it may be reused later when inserting a new entry.

(1). Corresponding to this Deletion algorithm, design an Insertion algorithm which enters a new entry name into a DIR.

(2). Discuss the advantages and disadvantages of the new Insertion/Deletion algorithms.

(3). Implement the new Insertion/Deletion algorithms and compare their performance with that of EXT2.

6. MTX does not yet support file streams. Implement library I/O functions to support file streams in user space.
7. Implement real and effective user IDs in MTX file system.
8. In MTX, mount can only mount block special files, e.g./dev/fd0 and hard disk partitions. Modify mount.c in FS to mount EXT2 disk image files as loop devices.

References

Cao, M., Bhattacharya, S, Tso, T., "Ext4: The Next Generation of Ext2/3 File system", IBM Linux Technology Center, 2007.

Card, R., Theodore Ts'o, T., Stephen Tweedie, S., "Design and Implementation of the Second Extended Filesystem", web.mit.edu/tytso/www/linux/ext2intro.html, 1995

EXT2: http://www.nongnu.org/ext2-doc/ext2.html, 2001

EXT3: http://jamesthornton.com/hotlist/linux-filesystems/ext3-journal, 2015

Chapter 12
Block Device I/O and Buffer Management

12.1 Block Device I/O Buffers

In Chap. 11, we showed the algorithms of read/write regular files. The algorithms rely on two key operations, read_block and write_block, which read/write a disk block to/from a buffer in kernel memory. Since disk I/O are slow in comparison with read/write memory, it is undesirable to read/write disk blocks on every read/write file operation. For this reason, most OS kernels use I/O buffering to reduce the number of physical I/O. I/O buffering is especially important for block devices containing file systems. A well designed I/O buffering scheme can significantly improve I/O efficiency and increase system throughput.

The basic principle of I/O buffering is very simple. The kernel uses a set of I/O buffers as a cache memory for block devices. When a process tries to read a disk block identified by (dev, blk), it first searches the buffer cache for a buffer already assigned to the disk block. If such a buffer exists and contains valid data, it simply reads from the buffer without reading in the disk block again. If such a buffer does not exist, it allocates a buffer for the disk block, reads data from disk into the buffer, then reads data from the buffer. Once a block is read in, the buffer will remain in the buffer cache for the next read/write requests of the same block by any process. Similarly, when a process writes to a disk block, it first gets a buffer assigned to the block. Then it writes data to the buffer, marks the buffer as dirty and releases it to the buffer cache. Since a dirty buffer contains valid data, it can be used to satisfy subsequent read/write requests for the same block without incurring real disk I/O. Dirty buffers will be written to disk only when they are to be reassigned to different blocks.

Before discussing buffer management algorithms, we first introduce the following terms. In read_file/write_file, we have assumed that they read/write from/to a dedicated buffer in kernel memory. With I/O buffering, the buffer will be allocated dynamically from a buffer cache. Assume that BUFFER is the structure type of buffers (defined below) and getblk(dev, blk) allocates a buffer assigned to (dev, blk)

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_12

from the buffer cache. Define a bread(dev, blk) function, which returns a buffer (pointer) containing valid data.

```
BUFFER *bread(dev,blk) // return a buffer containing valid data
{
   BUFFER *bp = getblk(dev,blk); // get a buffer for (dev,blk)
   if (bp data valid)
   return bp;
   bp->opcode = READ;            // issue READ operation
   start_io(bp);                 // start I/O on device
   wait for I/O completion;
   return bp;
}
```

After reading data from a buffer, the process releases the buffer back to the buffer cache by brelse(bp). Similarly, define a write_block(dev, blk, data) function as

```
write_block(dev, blk, data)      // write data from U space
{
    BUFFER *bp = bread(dev,blk); // read in the disk block first
    write data to bp;
    (synchronous write)? bwrite(bp) : dwrite(bp);
}
```

where bwrite(bp) is for synchronous write and dwrite(bp) is for delay-write, as shown below.

```
-------------------------------------------------------------------
bwrite(BUFFER *bp){           |  dwrite(BUFFER *bp){
   bp->opcode = WRITE;        |    mark bp dirty for delay_write;
   start_io(bp);              |    brelse(bp); // release bp
   wait for I/O completion;   |  }
   brelse(bp); // release bp  |
 }                            |
-------------------------------------------------------------------
```

Synchronous write waits for the write operation to complete. It is used for sequential or removable block devices, e.g. tape drives or floppy disks. For random access devices, e.g. hard disks, all writes can be delay writes. In delay write, dwrite(bp) marks the buffer as dirty and releases it to the buffer cache. Since dirty buffers contain valid data, they can be used to satisfy subsequent read/write requests of the same block. This not only reduces the number of physical disk I/O but also improves the buffer cache effect. A dirty buffer will be written to disk only when it is to be reassigned to a different disk block, at which time the buffer is written out by

```
awrite(BUFFER *bp)
{
   bp->opcode = ASYNC;     // for ASYNC write;
   start_io(bp);
}
```

which calls start_io() on the buffer but does not wait for I/O completion.

Physical Block Device I/O Each device has an I/O queue which contains buffers of pending I/O. The start_io() operation on a buffer is

```
start_io(BUFFER *bp)
{
   enter bp into device I/O queue;
   if (bp is first buffer in I/O queue)
       issue I/O command for bp to device;
}
```

When an I/O operation completes, the device interrupt handler finishes the I/O operation on the current buffer and starts I/O for the next buffer in the I/O queue if it is non-empty. The algorithm of the disk I/O interrupt handler is

```
InterruptHandler()
{
   bp = dequeue(device I/O queue); // bp = remove head of I/O queue
   (bp->opcode == ASYNC)? brelse(bp) : unblock process on bp;
   if (!empty(device I/O queue))
      issue I/O command for first bp in I/O queue;
}
```

12.2 Unix I/O Buffer Management Algorithm

Unix I/O buffer management algorithm first appeared in (Ritchie and Thompson 1978; Lion 1996). It is discussed in detail in Chap. 3 of Bach (Bach 1990). The Unix buffer management subsystem consists of the following components.

(1). I/O buffers: A set of NBUF buffers in kernel is used as a buffer cache. Each buffer is represented by a structure.

```
typdef struct buf{
  struct buf *next_free;      // freelist pointer
  struct buf *next_dev;       // dev_list pointer
  u16 dev,blk;                // assigned disk block;
  u16 opcode;                 // READ|WRITE
  u16 dirty;                  // buffer data modified
  u16 async;                  // ASYNC write flag
  u16 valid;                  // buffer data valid
  u16 busy;                   // buffer is in use
  u16 wanted;                 // some process needs this buffer
  struct semaphore lock=1;    // buffer locking semaphore; value=1
  struct semaphore iodone=0; // for process to wait for I/O completion;
  char buf[BLKSIZE];          // block data area
} BUFFER;
BUFFER buf[NBUF], *freelist; // NBUF buffers and free buffer list
```

The buffer structure consists of two parts; a header part for buffer management and a data part for a block of data. To conserve kernel memory, the status fields may be defined as a bit vector, in which each bit represents a unique status condition. They are defined as u16 here for clarity and ease of discussion.

(2). Device Table: Each block device is represented by a device table structure.

```
struct devtab{
  u16  dev;              // major device number
  u32  start_sector;     // for hard disk partitions
  u32  size;             // size of device in blocks
  BUFFER *dev_list;      // device buffer list
  BUFFER *io_queue;      // device I/O queue
} devtab[NDEV];
```

Each devtab has a dev_list, which contains I/O buffers currently assigned to the device, and an io_queue, which contains buffers of pending I/O operations on the device. The I/O queue may be organized for optimal I/O operations. For instance, it may implement the various disk scheduling algorithms, such as the elevator algorithm or the linear-sweep algorithm, etc. For the sake of simplicity, we shall assume FIFO I/O queues.

(3). Buffer Initialization: When the system starts, all I/O buffers are in the freelist and all device lists and I/O queues are empty.

(4). Buffer Lists: When a buffer is assigned to a (dev, blk), it is inserted into the devtab's dev_list. If the buffer is currently in use, it is marked as BUSY and removed from the freelist. A BUSY buffer may also be in the I/O queue of a devtab. Since a buffer cannot be free and busy at the same time, the device I/O queue is maintained by using the same next_free pointer. When a buffer is no longer BUSY, it is released back to the freelist but remains in the dev_list for possible reuse. A buffer may change from one dev_list to another only when it is reassigned. As shown before, read/write disk blocks can be expressed in terms of bread, bwrite and dwrite, all of which depend on getblk and brelse. Therefore, getblk and brelse form the core of the Unix buffer management scheme. The algorithm of getblk and brelse is as follows.

(5). Unix getblk/brelse algorithm: (Lion 1996, Chap. 3 of (Bach 1990)).

```
/* getblk: return a buffer=(dev,blk) for exclusive use */
BUFFER *getblk(dev,blk){
while(1){
  (1). search dev_list for a bp=(dev,blk);
  (2). if (bp in dev_lst){
           if (bp BUSY){
               set bp WANTED flag;
               sleep(bp);
               continue;
           }
           /* bp not BUSY */
           take bp out of freelist;
           mark bp BUSY;
           return bp;
       }
  (3). /* bp not in cache; try to get a free buf from freelist */
       if (freelist empty){
           set freelist WANTED flag;
           sleep(freelist);
           continue;
       }
  (4). /* freelist not empty */
       bp = first bp taken out of freelist;
       mark bp BUSY;
       if (bp DIRTY){       // bp is for delayed write
           awrite(bp);      // write bp out ASYNC;
           continue;
       }
  (5). reassign bp to (dev,blk); // set bp data invalid, etc.
  return bp;
}

/** brelse: releases a buffer as FREE to freelist **/
brelse(BUFFER *bp){
  if (bp WANTED)
     wakeup(bp);       // wakeup ALL proc's sleeping on bp;
  if (freelist WANTED)
     wakeup(freelist); // wakeup ALL proc's sleeping on freelist;
  clear bp and freelist WANTED flags;
  insert bp to (tail of) freelist;
}
```

It is noted that in (Bach 1990), buffers are maintained in hash queues. When the number of buffers is large, hashing may reduce the search time. If the number of buffers is small, hashing may actually increases the execution time due to additional overhead. Furthermore, studies (Wang 2002) have shown that hashing has almost no effect on the buffer cache performance. In fact, we may consider the device lists as hash queues by the simple hashing function hash(dev, blk)=dev. So there is no loss of generality by using the device lists. The Unix algorithm is very simple and easy to understand. Perhaps because of its extreme simplicity, most people are not very impressed by it at first sight. Some may even consider it naive because of the repeated retry loops. However, the more you look at it, the more it makes sense. This amazingly simple but effective algorithm attests to the ingenuity of the original Unix designers. Some specific comments about the Unix algorithm follow.

(1). Data Consistency: In order to ensure data consistency, getblk must never assign more than one buffer to the same (dev, blk). This is achieved by having the process re-execute the "retry loops" after waking up from sleep. The reader may verify that every assigned buffer is unique. Second, dirty buffers are written out before they are reassigned, which guarantees data consistency.

(2). Cache effect: Cache effect is achieved by the following means. A released buffer remains in the device list for possible reuse. Buffers marked for delay-write do not incur immediate I/O and are available for reuse. Buffers are released to the tail of freelist but allocated from the front of freelist. This is based on the LRU ((Least-Recent-Used) principle, which helps prolong the lifetime of assigned buffers, thereby increasing their cache effect.

(3). Critical Regions: Device interrupt handlers may manipulate the buffer lists, e.g. remove a bp from a devtab's I/O queue, change its status and call brelse(bp). So in getblk and brelse, device interrupts are masked out in these critical regions. These are implied but not shown in the algorithm.

(4). Shortcomings of the algorithm:

(4)-1. Inefficiency: the algorithm relies on retry loops, which would cause excessive process switches. For example, releasing a buffer may wake up two sets of processes; those who want the released buffer, as well as those who just need a free buffer. Since only one process can get the buffer, all other awakened processes would have to sleep again.

(4)-2. Unpredictable cache effect: In the Unix algorithm, every released buffer is up for grabs. If the buffer is obtained by a process which needs a free buffer, the buffer would be reassigned, even though there may be processes which still need the buffer.

(4)-3. Possible starvation: The algorithm is based on the principle of "free economy", in which every process is given chances to try but with no guarantee of success. Therefore, process starvation may occur.

(4)-4. The algorithm uses sleep/wakeup, which is only suitable for uniprocessor kernels.

12.3 New I/O Buffer Management Algorithms

In this section, we shall develop new algorithms for I/O buffer management. Instead of sleep/wakeup, we shall use P/V on semaphores for process synchronization. The main advantages of semaphores over sleep/wakeup are

Counting semaphores can be used to represent the number of available resources, e.g. the number of free buffers. When processes wait for a resource, the V operation unblocks only one waiting process, which is guaranteed to have the resource.

These semaphore properties can be used to design more efficient algorithms for buffer management. Formally, we specify the problem as follows.

Assume a uniprocessor kernel (one process runs at a time). Use P/V on counting semaphores to design new buffer management algorithms which meet the following requirements:

1. Guarantee data consistency.
2. Good cache effect.
3. High efficiency: No retry loops and no unnecessary process "wakeups".
4. Free of deadlocks and starvation.

It is noted that merely replacing sleep/wakeup in the Unix algorithm by P/V on semaphores is not an acceptable solution because doing so would retain all the retry loops. We must redesign the algorithm to meet all the above requirements and justify that the new algorithm is indeed better than the Unix algorithm. First, we define the following semaphores.

```
BUFFER buf[NBUF];       // NBUF I/O buffers
SEMAPHORE free = NBUF;  // counting semaphore for FREE buffers
SEMAPHORE buf[i].sem = 1; // each buffer has a lock sem=1;
```

To simplify the notations, we shall refer to the semaphore of each buffer by the buffer itself. As in the Unix algorithm, initially all buffers are in the freelist and all device lists and I/O queues are empty. Before presenting solutions to the problem, we first illustrate the subtlety of the problem by an example. Most students tend to mimic the getblk/brelse algorithm of Unix and propose the following algorithm.

```
BUFFER *getblk(dev, blk) // return a locked buffer pointer
{
    while(1){
 (1). if (bp in dev_list){
         P(bp);
         remove bp from freelist;
         return bp;
      }
      // bp not in cache, try to create a bp=(dev,blk)
 (2). P(free);   // wait for a free buffer
      bp = remove a buffer from freelist;
      P(bp);     // lock buffer
      if (buffer is delay-write){
          awrite(bp);
          goto (2);
      }
      assign buffer to (dev,blk);
      return bp;
    }
}
brelse(BUFFER *bp)
{
 (3). enter bp into freelist;
 (4). V(bp); V(free);
}
```

Unfortunately, such an algorithm is incorrect. To see this, assume that several processes need the same buffer, which does not exist and there are no more free buffers. Then all of such processes would be blocked at (2) in getblk(). When buffers are released as free at (4), these processes would be unblocked one at a time to create the same buffer multiple times. To prevent multiple buffers, such processes would have to execute from (1) again, which is the same as retry. However, by rearranging the execution path of the algorithm slightly, we come up with a simple workable algorithm, which is shown below.

12.4 Simple PV-Algorithm

```
BUFFER *getblk(dev, blk)
{   while(1){
 (1).   P(free);           // get a free buffer first
 (2).   if (bp in dev_list){
 (3).       if (bp not BUSY){
                remove bp from freelist;
                P(bp);     // lock bp but does not wait
                return bp;
            }
            // bp in cache but BUSY
            V(free);       // give up the free buffer
 (4).       P(bp);         // wait in bp queue
            return bp;
        }
        // bp not in cache, try to create a bp=(dev,blk)
 (5).   bp = frist buffer taken out of freelist;
        P(bp);             // lock bp, no wait
 (6).   if (bp dirty){
            awrite(bp);    // write bp out ASYNC, no wait
            continue;      // continue from (1)
        }
 (7).   reassign bp to (dev,blk); // mark bp data invalid, not dirty
        return bp;
    }                      // end of while(1)
}

brelse(BUFFER *bp)
{
 (8). if (bp queue has waiter){ V(bp); return; }
 (9). if (bp dirty && free queue has waiter){ awrite(bp); return; }
 (10). enter bp into (tail of) freelist; V(bp); V(free);
}
```

Next, we show that the simple PV-algorithm is correct and meets the requirements.

(1). Buffer uniqueness: In getblk, if there are free buffers, the process does not wait at (1). Then it searches the dev_list. If the needed buffer already exits, the process does not create the same buffer again. If the needed buffer does not exist, the process creates the needed buffer by using a free buffer, which is guaranteed to have. If there are no free buffers, it is possible that several processes, all of which

need the same buffer, are blocked at (1). When a free buffer is released at (10), it unblocks only one process to create a needed buffer. Once a buffer is created, it will be in the dev_list, which prevents other processes from creating the same buffer again. Therefore, every assigned buffer is unique.

(2). No retry loops: The only place a process re-executes the while(1) loop is at (6), but that is not a retry because the process is continually executing.

(3). No unnecessary wakeups: In getblk, a process may wait for a free buffer at (1) or a needed buffer at (4). In either case, the process is not woken up to run again until it has a buffer. Furthermore, at (9), when a dirty buffer is to be released as free and there are waiters for free buffers at (1), the buffer is not released but written out directly. This avoids an unnecessary process wakeup. The reader is encouraged to figure out why?

(4). Cache effect: In the Unix algorithm, every released buffer is up for grabs. In the new algorithm, a buffer with waiters is always kept for reuse. A buffer is released as free only if it has no waiters. This should improve the buffer's cache effect.

(5). No deadlocks and starvation: In getblk(), the semaphore locking order is always unidirectional, i.e. P(free), then P(bp), but never the other way around, so deadlock cannot occur. If there are no free buffers, all requesting processes will be blocked at (1). This implies that while there are processes waiting for free buffer, all buffers in use cannot admit any new users. This guarantees that a BUSY buffer will eventually be released as free. Therefore, starvation for free buffers cannot occur.

The simple PV-algorithm is very simple and easy to implement, but it does have the following two weaknesses. First, its cache effect may not be optimal. This is because as soon as there is no free buffer, all requesting processes will be blocked at (1), even if their needed buffer may already exist in the buffer cache. Second, when a process wakes up from the freelist queue, it may find the needed buffer already exists but BUSY, in which case it will be blocked again at (4). Strictly speaking, the process has been woken up unnecessarily since it gets blocked twice. The following algorithm, which is optimal in terms of the number of process switches, does not have such weaknesses.

12.5 Optimal PV-Algorithm

In the Optimal PV-algorithm, we assume that every PROC structure has the added fields (dev, blk) and a buffer pointer, BUFFER *bp. In addition, we also assume that the V operation on a semaphore with waiters returns a pointer to the unblocked process PROC, which is a trivial extension of the standard V operation. In order to simplify the discussion, we ignore starvation for free buffers first. Starvation prevention will be considered later. The following shows the optimal PV-algorithm.

```
BUFFER *getblk(dev,blk)
{
(1). set running PROC.(dev,blk)=(dev,blk) and PROC.bp=0;
(2). search dev_list for bp=(dev,blk);
(3). if (found bp){
        if (bp is FREE){
           P(free);        // dec free.value by 1; no wait
           take bp out out freelist;
           P(bp);          // lock bp; no wait
           return bp;
        }
        // bp BUSY
        P(bp);             // wait in bp.queue
(4).    return PROC.bp;    // return buffer pointer in PROC
     }
     // buffer not in cache, try to create the needed buffer
     while(1){
(5).    P(free);           // try to get a free buffer
(6).    if (PROC.bp)       // if waited in free.queue
           return PROC.bp; // return buffer pointer in PROC
(7).    bp = first buffer removed from from freelist;
(8).    P(bp);             // lock bp
        if (bp dirty){
            awrite(bp);    // asynchronous write, no wait
            continue;      // continue from (5)
        }
(9).    reassign bp to (dev,blk);
        return bp;
     }
}
brelse(BUFFER *bp)
{
 (10). if (bp queue has waiter){
          PROC *p=V(bp); p->bp=bp; // give bp to first waiter
          return;
       }
       /* bp queue has no waiter */
 (11). if (free queue has waiter){

    (12).     if (bp is dirty){ // never release dirty buffer as free
                 awrite(bp); return;
              }
              /* free queue has waiter && bp not dirty */
    (13).     swapQueue(free, bp);  // bp.queue=0 but swap anyway
              goto (10)
          }
          // no waiter in both bp and free queues
    (14). enter bp into tail of freelist; V(bp); V(free);
   }
```

In general, the swapQueue() operation at (13) consists of the following steps.

```
(1). Let P1 = first waiter in free queue;
       m = number of PROCs in free.queue;
       n = number of PROCs in free.queue waiting for the same bp as P1;
       k = number of PROCs in bp.queue
(2). Reassign bp to P1.(dev,blk);
(3). Swap bp.queue with PROCs waiting for the same bp in free.queue;
(4). Adjust semaphore values: free.value = -(m+k-n); bp.value = -n;
```

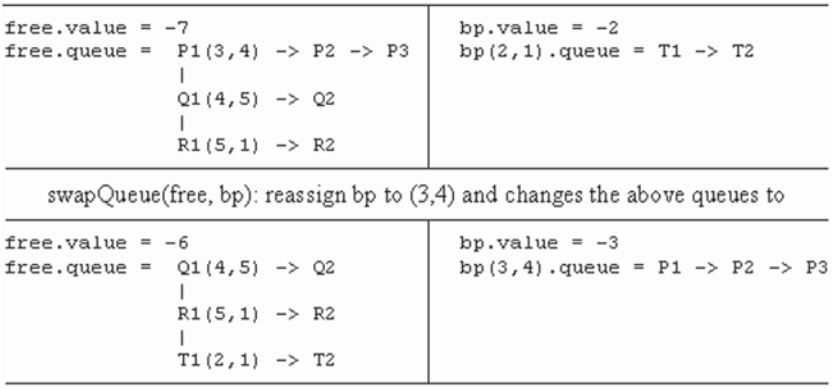


Fig. 12.1 General swapQueue operation

Conceptually, the free.queue may be regarded as a 2-dimensional queue, in which the horizontal queue contains PROCs all of which need the same bp = (dev, blk), and the vertical queue contains PROCs in their requesting order. swapQueue(free, bp) swaps the first horizontal queue of free.queue with bp.queue, adjusts the semaphore values and reassigns bp to the (dev, blk) of the first horizontal queue. Figure 12.1 shows an example of the general swapQueue() operation.

After swapQueue(), bp is reassigned to a new block and its queue contains all the PROCs which were waiting for the newly assigned buffer. The situation becomes identical to that of releasing a buffer with waiters. In comparison with the Unix algorithm, brelse here is slightly more complex, which also makes the algorithm more balanced. The following is a quick walk-through of the Optimal PV-algorithm.

At (1), the process records its (dev, blk) in PROC.(dev, blk) and sets PROC. bp = 0. The recorded PROC.(dev, blk) allows brelse to see what a blocked process is waiting for, and PROC.bp is for a blocked process to return the needed buffer upon wakeup.

(2) and (3) are obvious. At (4), the process returns PROC.bp, which may not be the bp it found earlier. In the optimal PV-algorithm, if a process ever becomes blocked, it simply returns PROC.bp when it runs again.

At (5), the process tries to get a free buffer. If there are free buffers, the process does not wait and its PROC.bp is still 0, in which case it proceeds to (7). Otherwise, it must have waited in free.queue, in which case it simply returns the buffer in its PROC.bp.

(7) to (9) are self-explanatory. (9) is the only place where a process creates a new buffer for itself.

At (10), if the buffer has waiters, the buffer should be kept for reuse. So it is given to the next waiter, which corresponds to Step (4) in getblk. Thus, a buffer with waiters is never released as free, which should improve the cache effect.

At (11), there are no waiters for this buffer but there are waiters for free buffers. At (12), if the buffer is dirty, it is written out directly to avoid an unnecessary process wakeup.

At (13), the buffer has no waiters, is not dirty and there are waiters in free.queue. Instead of releasing the buffer as free, which may lead to multiple buffers, (13) uses a "planned economy" strategy by creating the needed buffer for the waiting process (or processes which need the same buffer). In this case, bp.queue is empty, so swapQueue() only fills bp.queue with those processes from free.queue which are waiting for the same buffer. This ensures that only one buffer is created for such processes. After swapQueue(), the buffer is handed over to a waiter as usual.

At (14), the buffer is truly released as free.

Next, we show that the Optimal PV-algorithm meets all the design requirements.

(1). Buffer Uniqueness: The only possible source of multiple buffers is at (5) of getblk. If several PROCs wanting for the same bp are blocked at P(free), the needed buffer is created at either (10) or (12) by swapQueue() in brelse, which creates only one buffer for all such processes. Therefore, every assigned buffer is unique.

(2). No retry loops: This is obvious. As before, the while(1) is not a retry loop because the process is continually executing.

(3). No race conditions: Race conditions may only occur in the following way. When a buffer bp is released as free, it unblocks a process from free.queue. Before the unblocked process runs, another process may find the bp via its dev_list as exactly what it needs and removes bp from freelist. When the unblocked process runs, the intended free buffer may no longer exist. This is not possible in the Optimal PV-algorithm because such a time gap does not exist. If there are processes waiting for free buffer, their needed buffer is handcrafted and handed over to them directly. Therefore, race conditions cannot occur.

(4). No unnecessary process wakeups: The behavior of every process in getblk is

(a). if it finds the needed buffer in cache and the buffer is free, it returns the buffer.

(b). if it can create the needed buffer by using a free buffer, it creates the buffer and returns the buffer. In these two cases, the process does not wait.

(c). if a process has waited in a semaphore queue, it just returns PROC.bp. When and how did it get such a buffer is of no concern to the process. So there is absolutely no unnecessary process wakeups. The algorithm is therefore optimal in terms of the number of process switches.

(5). Good cache effect: In the new algorithm, buffers that are still in demand are never released as free. Such buffers are always kept for reuse. In the Unix algorithm every released buffer is open to competition, even if there are processes still need that buffer. Therefore, the cache effect of the PV algorithm should be better than that of the Unix algorithm. This is confirmed by simulation study results.

(6). No deadlocks: This is because the semaphore locking order is unidirectional.

(7). Starvation prevention: The only possible cause of starvation is when buffers are always in use, so that some process may wait for free buffer forever. Whereas there is no starvation in the simple PV-algorithm, it is possible in the Optimal PV-algorithm. We can modify the algorithm to prevent starvation as follows.

(7)-1. Assume that every buffer has a use_count and MAX_COUNT is a constant, which can be set to an arbitrarily large value.

(7)-2. In brelse, when a buffer is kept for reuse at (10) and if there are waiters for free buffers, we increment the buffer's use_count by 1. If free.queue is empty, we reset the buffer's use_count to 0. Thus, a buffer's use_count increases only if there are processes waiting for free buffers.

(7)-3. When a buffer's use_count reaches MAX_COUNT, the buffer will be reassigned to prevent starvation. Then the problem is: what to do with those processes still in the buffer's waiting queue? The swapQueue() operation simply swaps the buffer's waiting queue with a horizontal queue of free.queue, reassigns the buffer and adjust the semaphore values accordingly. This would move the processes in the buffer's original waiting queue to the free.queue as if they did not find the buffer before. Thus, while waiting for a needed buffer a process may be moved to different waiting queues, possibly many times. But this should be of no concern to the process. When a blocked process runs again, it simply returns the buffer (pointer) in its PROC.

After swapQueue(), the buffer is reassigned to a new block and its queue contains all the PROCs which were waiting for the newly assigned buffer. The situation becomes identical to that of releasing a buffer with waiters. Implementation of the Optimal PV-algorithm with starvation prevention is left as an exercise.

12.6 Comparisons of I/O Buffer Management Algorithms.

In the above sections, we presented two new I/O buffer management algorithms using semaphores. Simulation studies (Wang 2002) have shown that, under certain conditions, the PV-algorithms perform better than the Unix algorithm. The results are summarized briefly below.

1. When NBUF/NPROC >4, the Unix algorithm is about 10% faster than the PV-algorithms and the buffer cache hit ratio is virtually the same. If we replace the function calls to P/V with in-line code, their running times are also nearly identical. Thus, when there is less process competition for buffers, the retry strategy of the Unix algorithm is very effective.
2. When NBUF/NPROC <=4, the PV-algorithm starts to run faster and the hit ratio is also higher. As the NBUF/NPROC ratio decreases, the PV-algorithms perform much better than the Unix algorithm in both execution time and hit ratio.
3. The test results suggest a hybrid algorithm, which combines the strengths of both algorithms. In the hybrid algorithm, use the Unix algorithm when the ratio of NBUF to NPROC is large. Switch to the PV-algorithm when the ration becomes <=4.

The real mode MTX implements the simple PV-algorithm. With only 4 buffers, the hit ratio is normally around 50% but it can reach over 90% if several processes try to access the same file.

Problems

1. The MTX kernel uses the simple PV algorithm for buffer management. Implement the Optimun-PV algorithm with starvation prevention for buffer management.
2. In the real mode MTX, the number of block device I/O buffers are fixed as NBUF = 4 due to kernel memory size limit. Implement the following schemes:
   a. Define NBUF = 64 buffer heads in kernel with each buffer's 1KB data area in the segment 0×9000–0×A000. This allows the MTX kernel to have 64 I/O buffers at the expense of reducing available free memory by 64KB.
   b. Dynamic buffers: whenever a new buffer is needed, use kmalloc (sizeof(BUFFER)) to allocate a piece of memory for the buffer. When loading a user mode image, if there is not enough memory, evict some of the free buffers to make room for the new image. The advantage of this scheme is that the number of I/O buffers can be very large since all available memory can be used as I/O buffers. The disadvantage is that loading a large user image may trigger a cascade of delay writes of dirty buffers. This is another example of trading space with time.
3. In the optimum-PV algorithm, write operations are assumed to be asynchronous, i.e. process does not wait for the write operation to complete. This depends on the block device driver design. Assume: write operations are asynchronous for IDE hard disks but synchronous for floppy disk. Modify the optimum-PV algorithm for it to work for both kinds of disks.

References

Bach, M.J., "The Design of the Unix operating system", Prentice Hall, 1990

Lion, J., "Commentary on UNIX 6th Edition, with Source Code", Peer-To-Peer Communications, ISBN 1-57398-013-7, 1996.

Ritchie, D.M., Thompson, K., "The UNIX Time-Sharing System", Bell System Technical Journal, Vol. 57, No. 6, Part 2, July, 1978,

Wang, X., "Improved I/O Buffer Management Algorithms for Unix Operating System", M.S. thesis, EECS, WSU, 2002

Chapter 13
User Interface

User interface is a mechanism which allows users to interact with an operating system. As such, it is an indispensable part of every operating system. There are two kinds of user interfaces; Command Line Interface (CLI) and Graphic User Interface (GUI). Since MTX does not yet support graphic display, we shall only discuss command-line user interface. User interface consists of a set of user mode programs, which rely on system calls to the operating system kernel. In this chapter, we first list all the system calls to the RMTX kernel and explain their functions. Then we show how to develop user mode programs that are essential to system operations. Specifically, we shall develop an init program for system startup, a login program for users to login and a simple sh for users to execute commands. In addition, we also list other user mode programs which help demonstrate the capabilities of the RMTX system. Then we present a comprehensive description of the design and implementation of the RMTX kernel. Lastly, we explain the startup sequence of RMTX and how to recompile the RMTX system.

13.1 User Interface in MTX

As an educational system, MTX has several different versions. In addition to RMTX, which is MTX in 16-bit real mode, it also has several versions in 32-bit protected mode, which will be developed in Chaps. 14 and 15. The following discussion applies to all versions of MTX since their user interfaces are identical. Table 13.1 lists the system calls in RMTX.

The system calls are divided into four groups. Group 1 (number 0–19) is for process management. Group 2 (number 20–49) is for file system operations. Group 3 (number 50–53) is for signals and signal processing and Group 4 is for miscellaneous system calls. All the system call functions in Groups 1–3 are compatible with those of Unix/Linux. The only exception is exec. In MTX, when a user enters a command line

a.out arg1 arg2… argn

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_13

Table 13.1 System calls in RMTX Kernel

```
================ RMTX System Call Functions =====================
Number  Name          Usage                      Function
 ------ -------  ----------------------  ----------------------
   0   getpid       getpid()                  get process pid
   1   getppid      getppid()                 get parent pid
   2   getpri       getpri()                  get priority
   3   setpri       setpri(pri)               set priority
   4   getuid       getuid()                  get uid
   5   chuid        chuid(uid,gid)            set uid,gid
   6   yield        yield()                   switch process
   9   exit         exit(value)               terminate process
  10   fork         fork()                    fork child process
 *11   exec         exec(cmd_line)            change image
  12   wait         wait(&status)             wait child to die
  13   vfork        vfork()                   fork child process
  14   thread       thread(fd,stack,flag,prt) create thread
  15   mutex_creat   mutex_creat()            mutex functions
  16   mutex_lock    mutex_lock(&mutex)
  17   mutex_unlock  mutex_unlock(&mutex)
  18   mutex_destroy mutex_destroy(&mutex)
 ------------------------------------------------------------
  20   mkdir        mkdir(pathname)           make directory
  21   rmdir        rmdir(pathname)           rm   directory
  22   creat        creat(pathname)           creat file
  23   link         link(oldname, newname)    hard link to file
  24   unlink       unlink(pathname)          unlink
  25   symlink      symlink(oldname,newname)  create sym link
  26   readlink     readlink(name, buf[ ])    read symlink
  27   chdir        chdir(pathname)           change dir
  28   getcwd       getcwd(buf[ ])            get cwd pathname
  29   stat         stat(filename, &stat_buf) stat file
  30   fstat        fstat(fd, &stat_buf)      stat fd
  31   open         open(filename, flag)      open for R|W|APP
  32   close        close(fd)                 close fd
 *33   lseek        lseek(fd, position)       lseek
  34   read         read(fd, buf[ ], nbytes)  read file
  35   write        write(fd,buf[ ], nbytes)  write to file
  36   pipe         pipe(pd[2])               carete pipe
  37   chmod        chmod(filename, mode)     change permission
  38   chown        chown(filname, uid)       change file owner
  39   touch        touch(filename)           change file time
  40   settty       settty(tty_name)          set proc.tty name
  41   gettty       gettty(buf[ ])            get proc.tty name
  42   dup          dup(fd)                   dup file descriptor
  43   dup2         dup2(fd1, fd2)            dup fd1 to fd2
  44   ps           ps()                      ps in kernel
  45   mount        mount(FS, mountPoint)     mount file system
  46   umount       umount(mountPoint)        umount file system
  47   getSector    getSector(sector, buf[ ]) read CDROM sector
  48   cd_cmd       cd_cmd(cmd)               cmd to CD driver
---------------------------------------  ----------------------
  50   kill         kill(pid, sig#)           send signal to pid
  51   signal       signal(sig#, catcher)     siganl handler
  52   pause        pause(t)                  pause for t seconds
  53   itimer       itimer(sec, action)       set timer request
  54   send         send(msg, pid)            send msg to pid
  55   recv         sender=recv(msg)          receive msg
  56   tjoin        tjoin(n)                  threads join
  57   texit        texit(value)              thread exit
  58   hits         hits()                    I/O buffer hit ratio
  59   color        color(v)                  display color
  60   sync         sync()                    sync file system
=================================================================
```

the entire command line is used in the exec system call but it passes the entire command line to the new image when execution starts. Parsing the command line into argc and argv is done by the startup code, crt0, in the new image.

13.2 User Mode Programs

In RMTX, every user mode program consists of the following parts.

(1). Entry point in u.s file:

```
          .globl _main0, _exit
           call  _main0        ! call main0()
           push  #0            ! exit(0)
           call  _exit
    ! other assembly code, e.g. syscall which issues INT 80
```

The function main0() is in crt0.c, which is precompiled and installed in a mtxlib for linking.

(2). /********** crt0.c file ******************/

```
 main0(char *cmdLine)
 {
   // parse cmdLine as argv[0], argv[1],...., argv[n]
   //       int argc = |<--------- argc ------------|
   main(argc, argv);  // call main(), passing argc and argv
 }
```

(3). The main.c file: As usual, every user mode program can be written as

```
#include "ucode.c"  // system call interface functions
main(int argc, char *argv[])
{
  // main function of C program
}
```

Then use BCC to generate a binary executable (with header), as in

```
as86 -o u.o  u.s
bcc -c -ansi $1.c
ld86 u.o $1.o mtxlib /usr/lib/bcc/libc.a
```

Alternatively, the reader may also use a Makefile to do the compile and linking.

13.3 User Mode Programs for MTX Startup

From the MTX system point of view, the most important user mode programs are init, login and sh, which are necessary to start up the MTX system. In the following, we shall explain the roles and algorithms of these programs.

13.3.1 The INIT Program

When MTX starts, init is loaded as the Umode image of the INIT process P1. It is the first user mode program executed by MTX. A simple init program, which forks only one login process on the PC's console, is shown below. The reader may modify it to fork several login processes, each on a different terminal.

```
/*********************** init.c file ****************************/
#include "ucode.c"
int console;
main()
{  int in, out;      // file descriptors for terminal I/O
   in  = open("/dev/tty0", O_RDONLY);
   out = open("/dev/tty0", O_WRONLY); // for display to console
   printf("INIT : fork a login proc on console\n");
   console = fork();
   if (console)
      parent();
   else              // child exec to login on tty0
      exec("login /dev/tty0");
}
int parent()
{ int pid, status;
  while(1){
      printf("INIT : wait for ZOMBIE child\n");
      pid = wait(&status);
      if (pid==console){
          printf("INIT: forks a new console login\n");
          console = fork();
          if (console)
             continue;
          else
             exec("login /dev/tty0");
      }
      printf("INIT: I just buried an orphan child proc %d\n", pid);
  }
}
```

13.3.2 The login Program

All login processes executes the same login program, each on a different terminal, for users to login. The algorithm of the login program is

```
/****************** Algorithm of login ******************/
// login.c : Upon entry, argv[0]=login, argv[1]=/dev/ttyX
#include "ucode.c"
int in, out, err;
main(int argc, char *argv[])
{
  (1). close file descriptors 0,1 inherited from INIT.
  (2). open argv[1] 3 times as in(0), out(1), err(2).
  (3). settty(argv[1]); // set tty name string in PROC.tty
  (4). open /etc/passwd file for READ;

       while(1){
  (5).    printf("login:");    get user name;
          printf("password:"); get user password;
          for each line in passwd file do{
              tokenize account line;
  (6).        if (user has a valid account){
  (7).           change uid, gid to user's uid, gid; // chuid()
                 change cwd to user's home DIR       // chdir()
                 close opened /etc/passwd file       // close()
  (8).           exec to program in user account     // exec()
              }
          }
          printf("login failed, try again \n");
       }
}
```

13.3.3 The sh Program

After login, the user process typically executes the command interpreter sh, which gets commands from the user and executes the commands. Before showing the algorithm of sh, we need to review I/O redirection and pipe operation first.

(1). I/O Redirection: I/O redirection is based on manipulations of file descriptors. Consider the command a.out < infile, which redirects inputs from a file. Without input redirection, a.out gets all inputs from stdin, whose file descriptor is 0. To redirect inputs from a file, the process can close the file descriptor 0 and then open infile for READ. The file descriptor of the newly opened file would be 0. This is because the kernel's open() function always uses the lowest numbered file descriptor for a newly opened file. After these, a.out would receive inputs from infile rather than from the keyboard. Similarly, for the command a.out > outfile, the process can

close its file descriptor 1 and then open outfile for WRITE, with CREAT if necessary. In addition to the close-open paradigm, file descriptors can also be manipulated directly by dup and dup2 system calls. The syscall newfd=dup(fd) creates a copy of fd and returns the lowest numbered file descriptor as newfd. The sysccall dup2(oldfd, newfd) copies oldfd to newfd, closing newfd first if necessary. After either dup or dup2, the two file descriptors can be used interchangeably since they point to the same open file instance (OFT).

(2). Pipe Operation: Consider the command cmd1 | cmd2, which connects 2 commands by a pipe. The algorithm of a process executing the pipe command is as follows.

```
int pid, pd[2];
pipe(pd);          // creates a pipe
pid = fork();      // fork a child to share the pipe
if (pid){          // parent as pipe READER
   close(pd[1]); dup2(pd[0],0); close(pd[0]);
   exec(cmd1);
}
else{              // child as pipe WRITER
   close(pd[0]); dup2(pd[1],1); close(pd[1]);
   exec(cmd2);
}
```

Multiple pipes in a command line can be handled recursively. Based on these, we formulate an algorithm for the sh program as follows.

```
/******************** Algorithm of sh *************************/
while (1){
   get a command line; e.g. cmdLine = cmd | cam2 | .... | cmdn &
   get cmd token from command line;
   if (cmd == cd || logout || su){  // built-in commands
      do the cmd directly;
      continue;
   }
   // for binary executable command
   pid = fork();                 // fork a child sh process
   if (pid){                     // parent sh
      if (no & symbol)           // assume at most one & for main sh
         pid = wait(&status);
      continue;                  // main sh does not wait if &
   }
   else                          // child sh
      do_pipe(cmd_line, 0);
}
int do_pipe(char *cmdLine, int *pd)
{
   if (pd){ // if has a pipe passed in, as WRITER on pipe pd:
      close(pd[0]); dup2(pd[1],1); close(pd[1]);
   }
   // divide cmdLine into head, tail by rightmost pipe symbol
   hasPipe = scan(cmdLine, head, tail);
   if (hasPipe){
       create a pipe lpd;
       pid = fork();
       if (pid){ // parent
           as READER on lpd:
              close(lpd[1]); dup2(lpd[0],0); close(lpd[0]);
           do_command(tail);
       }
       else
           do_pipe(head, lpd);
   }
   else
       do_command(cmdLine);
}
int do_command(char *cmdLine)
{
    scan cmdLine for I/O redirection symbols;
    do I/O redirections;
    head = cmdLine BEFORE redirections
    exec(head);
}
int sacn(char *cmdLine, char *head, char *tail)
{
  // divide cmdLine into head and tail by rightmost | symbol
  // cmdLine = cmd1 | cmd2 |...|cmdn-1 |  cmdn
  //           |<------- head --------->| tail |; return 1;
  // cmdLine = cmd1 ==> head=cmd1, tail=null;     return 0;
}
```

We illustrate the sh algorithm by an example. Consider the command line

```
cmdLine = "cat < infile | lower2upper | grep print > outfile"
```

where lower2upper converts lower-case letters to upper-case. Assume that the main sh process is P2, the processing sequence of the command line is as follows.

1. The main sh forks a child P3, and waits for the child to terminate.
2. The child sh, P3, calls do_pipe(cmdLine, 0). Since the pipe parameter is 0, P3 skips the "as writer on pipe" part and calls sacn() to look for any pipe symbol. scan() divides the cmdLine, by the rightmost | symbol, into head = "cat < infile | lower2upper"; tail = "grep print > outfile".
3. P3 creates a pipe, lpd[2]. Then it forks a child, P4, which receives the same file descriptors lpd[2]. Then P3 sets itself up as a reader on this pipe and calls do_command(tail). P3 first redirects its outputs to outfile, then it exec("grep print").
4. P4 calls do_pipe("cat < infile | lower2upper", lpd). Since lpd is not null, P4 sets itself up as a writer on the parameter pipe, lpd. Thus, P4 is a writer on lpd and P3 is a reader on lpd. Then P4 calls scan(), which divides the cmdLine parameter into head = "cat < infile", tail = "lower2upper". Then P4 creates a pipe, which is a new instance of lpd[2], and fork a child, P5, which shares the current pipe with P4. P4 sets itself up as the reader on this pipe and exec("lower2upper"). P4 and P3 are thus connected by the pipe created by P3.
5. P5 calls do_pipe("cat < infile", lpd). P5 sets it self up as a writer to the parameter pipe, which connects P5 to P4 through the pipe created by P4. Since there is no more pipe symbol in "cat < infile", P5 calls do_command(), in which it first sets up the input redirection and then exec("cat").

The following pipe-line diagram depicts the processes that execute the above pipe command, in which ==> stands for pipe and –> stands for I/O redirection.

infile --> P5 (cat) ==> P4 (lower2upper) ==> P3 (grep --> outfile):

Note that in the recursive calls to do_pipe(), the command line is processed from right to left, so that the main sh's child is the reader process on the rightmost pipe. In a sequence of commands connected by pipes, the rightmost pipe reader is normally the last one to terminate. In this scheme, the main sh will not prompt again until all the pipe operations have ended. As far as the pipes are concerned, the processing

order does not matter. It is also possible to reverse the direction of command line processing, i.e. from left to right. The disadvantage of this scheme is that the main sh will prompt again as soon as the writer on the leftmost pipe finishes, which may confuse the user since the sh's prompt may become lost among the pipe outputs. In either case, all other processes on the pipes will become orphans, which are released by the INIT process P1.

Sometimes a running program may need to know whether its stdin/stdout has been redirected or not. For example, when the cat program is run without a filename, it should get inputs from stdin (file descriprot 0) but it should also echo the inputs to let the user see the input chars. However, if it is used behind a pipe, as in cat f | cat, it should not echo the inputs. Similarly, when displaying the contents of a file, it must print a\r for each\n in order to produce the right visual effects. This is because in Unix each line of a file ends with a\n without a\r char. If the stdout has been redirected, as in "cat filename > outfile", it must not add a\r to each line. There are many ways to write the cat program to meet these requirements. For example, we may display the\r chars to fd = 2. Alternatively, we may determine whether stdin/stdout has been redirected or not by the following means.

(1). stat(gettty(), &st_tty); fstat(0, &st0); fstat(1, &st1);
(2). 0 has been redirected if st_tty.(dev,ino) != st0.(dev,ino)
(3). 1 has been redirected if st_tty.(dev,ino) != st1.(dev,ino)

13.4 Other User Mode Programs

All the commands in the/bin directory are user-mode programs, which are developed in the same way as described above. If needed, the reader may develop other user-mode programs. Table 13.2 lists all the user-mode programs in RMTX.

Table 13.2 User Programs in RMTX

```
Commands     Usage Example              Function
---------  -----------------  ------------------------------------
cat     :  cat file          :  display file contents to screen
chmod   :  chmod file 0644   :  change permission bits of file
chown   :  chown uid file    :  change file's uid
cp      :  cp src dest       :  copy files or directories
creat   :  creat filename    :  creat a new file
grep    :  grep pattern file :  grep lines matching the pattern
init    :                    :  program of INIT process
kill    :  kill sig# pid     :  send sign# to procees pid
link    :  link oldf newf    :  hard link newfile to oldfile
login   :                    :  program of login processes
lpr     :  lpr file          :  print file
man     :  man               :  a short on-line man page of MTX
mkdir   :  mkdir name        :  make a new directory
more    :  more file         :  display file one page/line at a time
mount   :  mount FS mountDir :  mount (EXT2) FS on mountDir
mv      :  mv file1 file2    :  rename files
ps      :  ps                :  display PROC status
reboot  :  reboot            :  flush kernel buffers and reboot
rm      :  rm file           :  remove file (unlink)
rmdir   :  rmdir DIR         :  remove directory
sh      :  sh                :  command interpreter
symlink :  symlink old new   :  create a symlink from new to old
sync    :  sync              :  flush kernel I/O buffers
touch   :  touch file        :  update time fields of file
umount  :  umount mountDir   :  umount a mounted FS
unlink  :  unlink file       :  decrement link count; rm if count=0
whoami  :  whoami            :  dispaly login uid
----------------- Demonstration Programs --------------------------
Signal catchers:
divide  :  divide            :  install handler for divide-by-0 trap
signal  :  signal            :  demonstrate signal and kill syscalls
trap    :  trap 2-7          :  simulate trap errors with trap handler
Control-C  key               :  send SIGINT(2) to proc on terminal
timer Service:
pause   :  pause t           :  sleep for t seconds
itimer  :  itimer t          :  set timer t seconds; end by SIGALRM(14)
catcher :  catcher           :  re-install signal catcher for SIGALRM
Process Management:
vfork:  :  vfork             :  demonstrate vfork()/exec() in MTX
loop    :  loop              :  simulate compute-bound process
Threads and synchronization:
matrix, race, norace         :  matrix operations by threads
qsort                        :  demonstrate quicksort by threads
Message passing:
sendrecv                     :  send/recv messages
cdserver                     :  CDROM file server: receive message
cdclient                     :  CDROM file system client: send request
Misc.
sbrk/rbrk:                   :  inc/dec user program heap size
hits                         :  Block device I/O buffer hit ratio
mp                           :  display MP table of SMP-compliant PC
=================================================================
```

13.5 Summary of RMTX

This section presents a comprehensive description of the RMTX operating system.

13.5.1 RMTX Source File Tree

The source file tree of RMTX contains the following files and directories.

```
RMTX:
|-- SETUP  : boot.s, setup.s, apentry.s
|-- type.h, Makefile, mk script
|-- kernel   : kernel source directory
|-- fs       : file system directory
|-- driver   : device driver directory
|-- USER     : use commands directory
```

A detailed listing of the RMTX kernel source files follow.

SETUP : RMTX only uses boot.s in which bytes 510-511 contain the boot signature 'RR' for real mode kernel.
type.h : type.h defines RMTX kernel data structure types, such as PROC, PIPE, EXT2 and other data structure types.
Makefile: top-level Makefile for recompiling RMTX by the make utility.
mk : a sh script for recompiling and installing RMTX.
/********** Kernel: Process Management Part **********/
ts.s : startup code, tswitch, interrupt handler entry points and port I/O functions
io.c : kernel I/O functions; printf(), get_byte()/put_byte(), etc.
queue.c : enqueue, dequue, printQueue functions
wait.c : ksleep, kwakeup, kwait, kexit functions
loader.c : executable image loader
mem.c : memory manager
fork.c : kfork, fork, vfork functions
exec.c : kexec function
threads.c : threads and mutex functions
pipe.c : pipe and pipe read/write functions
mes.c : message passing: send/recv functions
syscall.c : implementation of other syscall functions
int.c : syscall routing table
t.c : main entry, initialization, parts of process scheduler
/********** Device Drivers **********/
vid.c : console display driver
timer.c : timer and timer service functions
pv.c : semaphore operations
kbd.c : console keyboard driver
pr.c : parallel printer driver
serial.c : serial ports driver
fd.c : floppy disk driver
hd.c : IDE hard disk driver
atapi.c : ATAPI CDROM driver
/********** File system **********/
fs : implementation of a simple EXT2 file system

The RMTX kernel contains approximately 11000 lines of C code, with 300 lines of assembly code. The USER directory contains approximately 5000 lines of C code.

13.5.2 Process Management in RMTX

(1). PROC structure

In RMTX, each process or thread is represented by a PROC structure, which consists of three parts:

fields for process management,
a resource pointer to a per-process resource structure,
process kernel mode stack.

Since the PROC structure encompasses most of the design decisions of an OS, which in turn affects the implementation of the system, we shall explain it in more detail. In RMTX, the PROC structure is

```
typedef struct proc{
        struct  proc *next;
        int    *ksp;        // saved kstack pointer
        int     uss,usp;    // saved Umode uSS and uSP
        int     inkmode;    // in Kmode counter
        int     pid;        // proc ID
        int     ppid;       // parent pid
        int     status;     // process status
        int     priority;   // scheduling priority
        int     event;      // event to sleep on
        int     exitValue;  // exit status
        int     time;       // time quantum
        int     cpu;        // CPU time ticks used in ONE second
        int     pause;      // pause time
        int     type;       // PROCESS|THREAD type
   struct proc *parent;     // pointer to parent PROC
   struct proc *proc;       // process pointer for threads
   struct semaphore *sem;   // semaphore ptr if proc is BLOCKed
   struct pres *res;        // resource structure pointer
   int kstack[SSIZE];       // per process kernel mode stack
}PROC;
```

In the PROC structure, the next field is used to link the PROCs in various lists or queues. For example, free PROCs are in a freeList, PROCs that are ready to run are in a readyQueue, sleeping PROCs are in a sleepList and a PROC that is blocked on a semaphore is in the semaphore's waiting queue, etc. In RMTX, all link lists and queues are singly-linked. Since the number of PROCs is small, hashing is not used. For the same reason, there are also no child and sibling process pointers in the PROC structure. The ksp field is the saved kernel mode stack pointer of the process. When a process gives up CPU, it saves CPU registers in kstack and saves the stack pointer in ksp. When a process regains CPU, it resumes running

from the stack frame pointed by ksp. The fields uss and usp are for saving the process Umode stack segment and stack pointer. When a process enters Kmode from Umode, by either a syscall or an interrupt, it saves CPU registers in its Umode stack and saves the stack segment and stack pointer in PROC.[uss, usp]. At the end of a syscall or interrupt processing, it restores the Umode stack segment and stack pointer from PROC.[uss, usp] and returns to Umode by the saved interrupt stack frame. The inkmode field is a counter used to keep track of the number of times a process enters Kmode. Since every process begins in Kmode, inkmode = 1 when a process is created. It is decremented by 1 when a process exits Kmode and incremented by 1 when it enters Kmode. For example, if an interrupt occurred in Umode, upon entry to the interrupt handler, inkmode must be 1. On the other hand, if the interrupt occurred in Kmode, the process re-enters Kmode and inkmode must be greater than 1. This allows the interrupt handler to decide whether to return via saved context in Kmode stack or Umode stack. The fields pid, ppid, priority and status are obvious. In most large OS, each process is assigned a unique pid from a range of pid numbers. In MTX, we simply use the PROC index as the process pid, which simplifies the kernel code and also makes it easier for discussion. When a process terminates or exits, it must wakeup the parent, which may be waiting at its PROC address. For convenience, the parent pointer points to the parent PROC. This allows a dying process to find its parent quickly. The event field is the event value when a process goes to sleep. The exitValue field is the exit status of a process. If a process terminates normally by an exit(value) syscall, the exit value is recorded in the low byte of exitValue. If it terminates by a signal, the high byte of exitValue is the signal number. This allows the parent process to extract the exit status of a ZOMBIE child to determine whether it terminated normally or abnormally. The time field is the maximum run time quantum of a process and cpu is its CPU usage time. They are used to compute the dynamic process scheduling priority. The pause field is for a process to sleep for a number of seconds. In MTX, process and thread PROCs are identical. The type field identifies whether a PROC is a PROCESS or THREAD. RMTX uses sleep/wakeup for process management and pipes but it uses semaphores in device drivers and file system for process synchronization. When a process becomes blocked in a semaphore queue, the PROC's sem points to the semaphore. This allows the kernel to unblock a process from a semaphore queue, if necessary. For example, when a process waits for keyboard inputs, it is blocked in the keyboard driver's input semaphore queue. A kill signal delivered to the process should let the process continue. Unlike wakeup(), which simply changes a SLEEP process status to READY and enters it into readyQueue, unblocking a process from a semaphore queue must modify the semaphore's value and waiting queue. Each PROC has a res pointer pointing to a resource structure, PRES, which is

```
typedef struct pres{
   int    uid;                // user id
   int    gid;                // group id
   u16    segment,size,tsize,dsize,SEP; // Umode image memory and size
   MINODE *cwd;               // Current Working Directory pointer
   char   tty[16];            // terminal special file name
   char   name[32];           // program name string
   int    tcount;             // number of threads in process
   u16    signal;             // 15 signals = bits 1 to 14
   u16    sig[NSIG];          // 15 signal handlers
   struct semaphore mlock;    // messageQ lock
   struct semaphore message;  // # of messages
   struct mbuf     *mqueue;   // message queue
   OFT    *fd[NFD];           // open file descriptors
} PRES;
```

The PRES structure contains per-process information. It includes the process uid, gid, Umode image segment and size, current working directory, terminal special file name, program name, signal and signal handlers, message queue and file descriptors, etc. In RMTX the first NPROC PROCs are for processes and the remaining NTHREAD PROCs are for threads. To the RMTX kernel, processes and threads are independent execution units. Each process executes in a unique address space but threads in a process execute in the same address space of a process. During system initialization, each PROCESS PROC is assigned a unique PRES structure pointed by the res pointer. A process is also the main thread of the process. When creating a new thread in a process, its res pointer points to the same PRES structure of the process and its proc pointer points to the process PROC. In MTX all threads in a process share the same resources of the process. For example, they share the same file descriptors, so that opened files are available to all threads in a process. Some OS kernels allow individual threads to open files, which are private to the threads. In that case, each PROC must have its own file descriptor array. Similarly for signals and messages. In the PROC structure, kstack is the process kernel mode stack. It is defined as the last field of PROC for a simple reason. When a process enters kernel from user mode, we must change the execution environment from Umode to Kmode. In particular, we must set the stack pointer to the process kstack in kernel space. All these can only be done in assembly code at the interrupt handler entry point. By placing the kstack at the end of the PROC structure makes these operations easier. The following global variables are defined in the t.c file, which also contains the main() function and the initialization function init().

```
   PROC[NPROC+NTHREAD]: first NPROC PROCs for processes, remaining
                        NTHREAD PROCs for threads.
   freeList  = a list of free PROCs
   tfreeList = a list of free THREAD PROCs
   readyQueue= a priority queue of ready processes and threads.
   PROC *running = pointer to the current running process.
   sleepList = a FIFO list of sleeping processes;
```

(2). io.c: This file implements printf() and a few other I/O functions. In kernel mode, printf() is based on putc() in the display driver. In user mode, it is based on uputc(), which writes to file descriptor 1 of the process.

(3). queue.c:

This file implements get_proc()/put_proc() and enqueue()/dequeue() functions. get_proc() allocates a free PROC for a new process or thread. put_proc() returns a ZOMBIE PROC to a free PROC list. Processes and threads that are ready to run are maintained in a readyQueue by priority. enqueue() enters a PROC into the readyQueue by priority, and dequeue() returns a pointer to the highest priority PROC removed from the readyQueue. In addition, there are also other list/queue manipulation functions, e.g. enter/remove a sleeping process to/from the sleepList, etc. They are implemented in the relevant files for clarity.

(4). wait.c:

This file implements ksleep(), kwakeup(), kwait() and kexit() for process management.

ksleep()/kwakeup():

A process calls ksleep(event) to go to sleep on an event. An event is just a value which represents the sleep reason. When the awaited event occurs, another process or an interrupt handler calls kwakeup(event), which wakes up all the processes sleeping on the event. To ensure that processes are woken up in order, sleeping processes are maintained in a FIFO sleepList.

kwait():

kwait() allows a process to wait for a ZOMBIE child. If a process has children but no ZOMBIE child yet, it sleeps on its own PROC address. When a process terminates, it calls kexit() to becomes a ZOMBIE and wakes up the parent by kwakeup(running->parent). As in Unix, orphan processes become children of the INIT process P1, which repeatedly waits for any ZOMBIE children.

kexit(): every process calls kexit(exitValue) to terminate. The actions of kexit() are:

.give away children, if any, to P1;
.release resources, e.g. free Umode image memory, dispose of cwd, close opened file descriptors and release message buffers in message queue.
.record exitValue in PROC, become a ZOMBIE, wakeup parent and also P1 if it has sent any children to P1.
.call tswitch() to give up CPU for the last time.

ZOMBIE PROCs are freed by their parents through kwait(). Orphaned ZOMBIEs are freed by the INIT process P1.

(5). fork.c: This file contains kfork(), fork() and vfork(), which are implemented as follows.

kfork():

When RMTX starts, it first initializes the kernel data structures. Then it creates a process P0 as the initial running process. P0 runs only in Kmode and has the lowest priority 0. P0 continues to initialize the rest of the RMTX system. When initialization completes, P0 calls kfork("/bin/init") to create a child process P1. kfork()

calls fork1(), which is the common code of creating new processes. fork1() creates a child process ready to run in kernel but without a Umode image. When fork1() returns, kfork() allocates a Umode memory area for P1, loads the /bin/init file as the Umode image and sets up the Umode stack of P1 for it to resume. Then P0 switches process to run P1. P1 returns to Umode to execute /bin/init. P1 is the only process created by kfork(). After P1 runs, all other processes are created by fork() or vfork() as usual. The startup sequence of RMTX is unique. In most other Unix-like systems, the initial image of P1 is a piece of pre-compiled binary executable code containing an exec("/etc/init") system call. After system initialization, P1 is sent to execute the exec() syscall code in Umode, which changes P1's image to /etc/init. In RMTX, when P1 starts to run, it returns to Umode to execute the INIT image directly.

fork():

As in Unix/Linux, fork() creates a child process with an Umode image identical to that of the parent. If fork() succeeds, the parent returns the child's pid and the child returns 0. In RMTX, fork() is implemented as follows.

First, fork() calls fork1() to create a child process ready to run in Kmode. The child PROC inherits all the open file descriptors of the parent but without an Umode image. The child PROC is assigned the base priority of 127 and its resume point is set to _goUmode, so that when the child is scheduled to run, it returns to Umode immediately. Next, fork() allocates a Umode image area for the child of the same size as the parent and copies the parent's Umode image to the child. It then changes the copied DS, ES and CS in the child's Umode stack to child's segments, and sets the child's saved uss to its own data+stack segment. The child's saved usp is identical to that of the parent since it's the same offset relative to the saved uss segment. Then it changes the saved AX register in the child's Umode stack frame to 0. Finally, it returns the child's pid. When the child runs, it returns to its own Umode image with a 0. Because of the image copying, the child's Umode image is identical to that of the parent. The implementation of fork() in MTX is also unique. It only guarantees that the two Umode images are identical. The return paths of the parent and the child processes are very different.

vfork():

After forking a child, the parent usually waits for the child to terminate and the child immediately does exec() to change its Umode image. In such cases, copying image in fork() would be a waste. For this reason, many OS kernels support vfork(), which forks a child process without copying the parent image. RMTX also supports vfork(), which is implemented as follows.

a. Create a child process which shares the same image of the parent. vfork() first calls fork1() to create a child process ready to run in Kmode. When fork1() returns, vfork() does not allocate a Umode image for the child but lets it share the same Umode image with the parent, i.e. let the child PROC.uss be the same as that of the parent.
b. In the ustack area of the parent, create an interrupt stack frame for the child to return to pid=vfork() in Umode with a 0. This is done by copying the parent's interrupt stack frame to a low address area, e.g. at parent's saved usp—1KB and set the child's saved usp point to that area, as if the child had done a vfork() syscall by itself.

c. When either process returns to Umode, it would run in the same image of the parent. In an OS with memory protection hardware, the shared image can be marked as read-only. If either process tries to write to the shared image, the image is split into individual images. Since RMTX in real mode does not have any mechanism for memory protection, we must assume that, after vfork(), both the parent and child execute in read-only mode; either the parent executes wait() or the child executes exec() immediately.
d. In kexec(), a vforked child process creates a new Umode image but does not destroy the shared parent image.
e. In order to ensure that the child runs first, vfork() sets the child's priority to the maximum value 256. When the parent exits Kmode, it will switch process, allowing the child to return to Umode first.

Under normal conditions, vfork() can be used interchangeably with fork(). All the fork() syscalls in RMTX can be replaced with vfork(). However, as noted above, the vfork() in RMTX cannot support processes that modify the (shared) image. In such cases, fork() must be used instead.

(6). loader.c: This file implements the Umode image loader, which loads an executable file to a segment. The load() function uses the internal open() of the file system to open the file for read. It first calls the header() function, which reads the file header to determine the file's tsize, dsize and bss size. Then it loads the file's code+data sections to the specified segment and clears the bss section to 0. The header() function is also used in kexec() to calculate the total memory size needed by a new image.

(7). exec.c: This file implements the kexec() syscall. Unlike exec() in other OS, the parameter to kexec() is the entire command line of the form "cmd argv1 argv2… argvn". kexec() uses the first token, cmd, to create the new image, but it passes the entire command line to the new image. Parsing the command line to argc and argv is done in the new image in user mode. As of now, MTX does not support the PATH environment variable. All executable commands are assumed to be in the /bin directory. If the cmd file name does not begin with a /, it is assumed in the /bin directory by default. kexec() supports binary executable images with either a single segment or separate I&D spaces. It reads the file header to determine the total memory size needed to load the image. For single-segment images, the total memory size is the image's tsize+dsize+bssize plus a defualt ustack size of 20KB. If an image has separate I&D spaces, the image's combined data+stack size is set to the maximum value of 64KB. If the total needed size is not greater than the current image size, it simply loads the new image into the old image area. Otherwise, it calls kmalloc() of the memory manager to allocate the needed space, loads in the new image and releases the old image. Then it re-initializes the ustack to let the process execute from the beginning of the new image when it returns to Umode. When a vforked process calls kexec(), it only creates a new image without releasing the (shared) old image. In addition, kexec() also checks the process thread count. A process may exec() only if its tcount=1, i.e. when there are no other active threads in the process. kexec() clears the process signals and resets the signal handlers to default. MTX does not support close on exec() file descriptors. All opened files remain open after exec(). If needed, the reader may add this and other features to the kexec() function.

(8). mem.c file: This file implements the RMTX kernel memory manager. The RMTX kernel image is a binary executable with separate I&D spaces. The maximum size of the kernel image is 128KB bytes. When RMTX boots up, the kernel image is loaded at the segment 0×1000. Ideally, the entire memory area from the end of the RMTX kernel to the segment 0xA000 should be the initial free memory area. In order to keep the system initialization code simple, the free memory area begins from the segment 0×3000.The memory manager consists of the following components.

freeMem = a list of current free memory areas; initial free memory is from the segment 0x3000 to 0xA000.
kmalloc(size): allocate space for process Umode image size.
kmfree(segment, size): deallocate memory at (segment, size)
sbrk()/rbrk(): increase/decrease heap size of process Umode image.

Free memory areas are managed as variable-sized partitions by the first-fit algorithm. When a process forks, it calls kmalloc() to allocate a Umode image of the same size for the child. When a process terminates, it calls kmfree() to release the Umode image space back to the freeMem list. Similarly, when a process does exec(), it may call kmalloc() and kmfree() if the new image size is larger than the current image size. sbrk() and rbrk() adjust the heap size of the Umode image. They may call kmalloc() and kmfree() as needed. The function kmalloc() is non-blocking. Instead of waiting for free memory to become available, it prints an "out of memory" message and returns 0 if there is not enough free memory for the requested size. Currently, RMTX does not implement memory compaction and user mode image swapping. Both are suitable programming projects for possible improvements to the RMTX system.

(9). threads.c: MTX supports threads. In the RMTX kernel, there are NTHREAD PROCs dedicated to threads. A process PROC is also the main thread of the process. Each process has a tcount, which is the number of threads currently in the process. When a thread is created inside a process, it is allocated a THREAD PROC, which is linked to the process PROC. Threads in a process share the same resources of the process. MTX uses mutex for threads synchronization. Threads implementation and synchronization are discussed in Chap. 5. Several user mode programs, such as matrix, race, norace and qsort, are used to demonstrate the thread capability of RMTX. Due to the small number of thread PROCs (16), the reader should observe this limit when experimenting with threads programming in RMTX.

(10). pipe.c:

This file implements pipes in RMTX. Each pipe is a PIPE structure.

```
typedef struct pipe{
        char  buf[PSIZE];        // PSIZE = 64 chars
        int   head, tail, data, room;
        int   nreader, nwriter;  // number of readers/writers
        int   busy;              // FREE or in use
}PIPE;  PIPE pipe[NPIPE];        // NPIPE = 10
```

A PIPE contains a circular char buffer, buf[PSIZE], with head and tail pointers. The control variables are data, room, nreader and nwriter, where data = number of chars in the buffer, room = number of spaces in the buffer, nreader = number of reader processes on the pipe and nwriter = number of writer processes on the pipe. These variables are used for process synchronization during pipe read/write. When RMTX starts, pipe_init() is called to initialize all the PIPE structures as free. When a process creates a pipe by the pipe(int pd[2]) syscall, it executes kpipe() in kernel. kpipe() allocates a PIPE structure, initializes the pipe variables and allocates two file descriptors, pd[0] and pd[1], which point to the READ_PIPE and WRITE_PIPE OFT instances, respectively. After creating a pipe, a process normally forks a child process to share the pipe. During fork(), the child inherits all the open file descriptors of the parent. If the file descriptor is a pipe, fork() increments the reference count of the OFT by 1 and updates the number of readers or writers in the PIPE structure as needed. Then, each process calls close_pipe() to close its unwanted pipe descriptor. close_pipe() deallocates the PIPE structure if there are no more reader and writer. Otherwise, it does the normal closing of the file descriptor and wakes up any waiting reader/writer processes on the pipe. read_pipe() is for reading from a pipe, and write_pipe() is for writing to a pipe. Processes reading/writing pipes are synchronized by ksleep() and kwakeup(). A reader process returns 0 if the pipe has no data and no writer. Otherwise, it reads as much as it needs, up to the pipe size, and returns the number of bytes read. It waits for data only if the pipe has no data but still has writers. A writer process detects a BORKEN_PIPE error and aborts if there are no readers on the pipe. Otherwise, it writes as much as it can to the pipe. It may wait if there is no room and the pipe still has readers. After each read, the reader wakes up any waiting writers. After each write, the writer wakes up any waiting readers.

The pipe implementation has the following implications. In RMTX, when a process is woken up in kernel, it is assigned a high priority. If user mode processes read/writes pipes one byte at time, it would incur a process switch after each read/write syscall, which may be undesirable. There are two possible ways to remedy this. The first one is to let processes read/write pipes in chucks of data, e.g. by user level I/O buffering as in the standard I/O library functions. The second one is to modify the RMTX process scheduling algorithm. When waking up a process from a pipe, do not assign it a high priority. The reader may try either of these approaches to reduce the number of process switches.

(11). syscall.c:

This file implements the miscellaneous syscalls, such as getpid(), kps(), etc. Any additional syscall functions can also be added here.

(12). signal.c:

This file implements signals and signal handling in RMTX. It contains the following functions.

kkill(sig, pid) implements the kill(sig, pid) syscall. It delivers a signal, sig, to the target process. To keep things simple, MTX does not enforce kill permissions, so a process may kill any other process. If desired, the reader may modify kkill() to

enforce permission checking. RMTX supports only 15 signals, each corresponds to a bit (1–15) in PROC.res.signal. Delivering a signal i sets the i-th bit in the target PROC. res.signal to 1. In Unix/Linux, if a process is in the "uninterruptible sleep" state, it cannot be woken up by signals. In RMTX, processes only sleep for ZOMBIE children and reading/writing pipes, so they are interruptible. Therefore, a signal always wakes up a target process if it is sleeping. If the target process is waiting for block device I/O, it is blocked on a semaphore in the device driver. Unblocking such a process may confuse the device driver. However, if the process is waiting for terminal inputs, it will be unblocked by signals. Since the unblocked process resumes running inside a device driver, the driver's I/O buffer must also be adjusted to ensure consistency.

ksignal() implements the signal(sig, catcher) syscall. It installs a catcher function as the signal handler for the signal sig, except for signal number 9, which cannot be changed. When a process enters Kmode via a syscall and when it is about to return to Umode, it checks and handles outstanding signals by calling kpsig().

kpsig() calls cksig(), which resets the signal bit of an outstanding signal and returns the signal number. For each outstanding signal number n, if the process signal handler function, sig[n], is 0, the process calls kexit(n<<8) to die with an exit status=n<<8. If sig[n] is 1, the process ignores the signal. Otherwise, kpsig() sets up the process interrupt ustack frame in such a way that, when the process returns to Umode, it first returns to the signal catcher function. When the catcher function finishes, it returns to the place where it lastly entered kernel.

kdivide()/ktrap(): kdivide is the divide-by-zero trap handler. When the CPU in real mode encounters a divide-by-zero error, it traps to vector 0, which points to the divide-by-zero trap handler entry point, _divide, in ts.s. The trap handler calls kdivide(), in which the process delivers a number 1 signal to itself followed by kpsig(). If the process has not changed its default sig[1] handler, it would die by the signal. If the process has installed a handler to catch the signal, it will execute the handler function in Umode. A user mode program, divide.c, demonstrates the handling of divide-by-zero trap. When running the divide program, if the user chooses not to install a catcher, the program will die when it tries to divide by 0. If the user has installed a catcher, the program will execute the catcher when it tries to divide by 0. The catcher function uses a long jump to bypass the code containing the divide-by-zero instruction, allowing the program to continue. Handling other kinds of trap errors is similar. Since the PC in real mode does not recognize other kind of traps, trap errors are simulated by INT n instructions, where $1 < n < 8$. The trap.c program demonstrates trap handling by installed trap handlers. Another program that demonstrates signal handling is the itimer.c program, which catches signals from an interval timer.

```
/****** itimer.c: handle SIGALRM (14) signals *********/
int t = 1;                // default timer interval
void catcher(int sig)
{
  signal(14, catcher);    // install signal# 14 catcher again
  printf("catcher() in Umode, sig=%d\n", sig);
  itimer(t);              // set a t sec. interval timer
}
main(int argc, char *argv[])
{
  signal(14, catcher);    // install catcher() for SIGALRM (14)
  if (argc > 1)
     t = atoi(argv[1]);
  printf("request interval timer = %d\n", t);
  itimer(t);              // call itimer() in kernel
  while(1);               // looping
}
```

The program first installs a signal 14 catcher and sets an interval timer of t seconds. Then it loops forever. After the timer interval expires, the kernel's timer interrupt handler sends it a SIGALRM(14) signal. Without the catcher function, the process would die by default. If the catcher is not installed again, the process will die when it gets the next SIGALRM signal. Since the catcher is installed again, the process will be directed to execute the catcher function once every t seconds.

(13). int.c

This file implements the syscall routing table. First, a function pointer table is set up to contain all the syscall function entry addresses.

int (*f[ ])()={getpid, getppid, getpri, ksetpri, getuid,…….};

When a process issues a syscall, e.g. syscall(a, b,c, d,e), it enters kernel by INT 80, which sends it to _int80h: in ts.s, which calls kcinth().

```
int kcinth()
{
  int a,b,c,d,e,r;
  unlock();                // handle syscall with interrupts on
  if (running->res->signal) // check and handle outstanding signals
      kpsig();
  // get syscall parameters a,b,c,d,e from ustack, a=syscall number
  r = (*f[a])(b,c,d,e);       // invokde the syscall function
  if (running->res->signal)   // check and handle signal again
      kpsig();
  put_uword(r, running->usp + 2*8); // return r into saved AX in ustack
  running->pri = 128-running->cpu;  // drop back to Umode priority
}
```

In kcinth(), the process checks and handles signals first. This is because, if the process already has a pending signal, which may cause it to terminate, processing the syscall would be a waste of time. If the process survives the signal processing, it fetches the syscall parameters from the Umode stack, where a is the syscall number. Then it calls the corresponding syscall function by r=(*f[a])(b, c,d, e) to process the

syscall. When the syscall function returns, it checks and handles signals again. Each syscall function returns a value r, except kexit(), which never returns, and exec(), which does not return if the operation succeeds. Before returning to _goUmode in assembly, it writes r to the saved AX register in the interrupt stack frame as the return value to Umode.

(14). t.c

This is the main file of RMTX. When RMTX starts, it executes the startup code in ts.s, which calls main(). main() first calls init() to initialize the RMTX kernel and creates P0 as the initial running process. P0 kforks P1 with a Umode image,/bin/init, and switches to P1. When P1 runs, it forks one or more login processes, each on a separate terminal, and waits for any of the login processes to terminate. When the login processes run, the RMTX system is ready for use. In the t.c file, the functions nextrun(), schedule() and reschedule() are parts of the process scheduler. Process scheduling in RMTX is by both time slice and dynamic process priority, which is discussed in detail in Chap. 8. To observe process switches due to dynamic priorities, the reader may uncomment the print statement in reschedule() and recompile the RMTX kernel. Similarly, the reader may also enable the print statements in the timer interrupt handler (in timer.c) to observe timer based process switching.

(15). ts.s

This is the only assembly code file of RMTX. It consists of four parts:

a. RMTX startup code: The lines from start to call _main are the startup code of the RMTX kernel. During booting, the booter calls the time-of-day function of BIOS and saves it at 0×90000. When the MTX kernel starts, it retrieves the saved BIOS TOD as the system time base. In all versions of MTX, the time is maintained as follows. In the EXT2 file system, the INODE size is 128 bytes but the last 28 bytes are not used. So we define the last two INODE fields as
u32 i_date; u32 i_time which contain the date and time in BCD format. The date and time of each INODE are based on the time-of-day (TOD), which is the initial TOD of BIOS plus the current time in seconds. While RMTX runs, it displays the TOD at the lower right corner of the screen.
b. _tswitch: This is the context switching function. When a process relinquishes CPU, it calls tswitch(), in which it saves CPU registers into its kstack and saves the stack pointer into PROC.ksp. Then, it calls nextrun() to pick a ready PROC with the highest priority as the next running process. Since P0 has the lowest priority, it runs only when all other PROCs are non-runnable. During system initialization, P0 itself may be blocked when it reads the hard disk partition table to initializes the HD driver. In that case, P0 puts the CPU in idle state with interrupts enabled. When the HD interrupt occurs, the HD interrupt handler unblocks P0 and enters it into readyQueue. In nextrun(), it picks a ready PROC with highest priority as the next running PROC, sets the PROC's run time to QUANTUM (6–10 ticks), resets the PROC's CPU usage time to 0 and clears the switch process flag, sw_flag, to 0. Then execution returns to the RESUME part of tswitch(), which resumes the new running process from the saved stack frame in kstack. Since tswitch() is always called in Kmode, there is no need to save/restore CPU segment registers because they are always the same.

c. Interrupt handler entry and exit code: These are calls to the INTH macro. Each macro call is of the form _xinth: INTH chandler, which defines the entry point of an interrupt or trap handler. Interrupt and trap handlers are implemented in C in various parts of the RMTX kernel. When an interrupt handler finishes, execution returns to the interrupted point by _ireturn. If the interrupt occurred in Kmode, it returns via the process kstack. Otherwise, it returns via the process ustack. Before a process returns to Umode, it calls kpsig() to handle any outstanding signal first. If it survives the signal processing, it then calls reschedule(), which may switch to a process with higher priority.
d. The remaining assembly code contains port I/O and CPU interrupts masking functions for implementing critical regions.

13.5.3 Device Drivers

(1). Timer and Timer Service (timer.c):

The PC's channel 0 timer is programmed to generate interrupts every 1/60th of a second. During booting, the booter reads the BIOS time-of-day (TOD) and saves the information at 0×90000. When a MTX kernel starts, it retrieves the saved TOD and uses it as the real time base. At each second, it displays a wall clock. RMTX uses the timer interrupts for process scheduling, as described in Chap. 8 on process scheduling. Timer service is implemented by a TimerQueue, which is a list of interval timer requests (TQEs). Each TQE has a time field and an action function. When a TQE's time expires, the timer interrupt handler invokes the action function, which typically unblocks the process or sends an SIGALRM signal to the process.

(2). Other Device Drivers: MTX supports the following devices:

console video driver (vid.c)
keyboard driver (kbd.c)
printer driver (pr.c)
serial port driver (serial.c)
floppy disk driver (fd.c)
IDE hard disk driver (hd.c)
ATAPI CDROM driver (atapi.c)

All device drivers, except the console video driver, are interrupt-driven. Their interrupt handler entry points are installed in the assembly code file, ts.s, as INTH macro calls of the form _xinth: INTH xhandler, where xhandler is the interrupt handler in C. All device drivers use semaphores for synchronization. Details of device driver design and implementation are discussed in Chap. 10. Therefore, we shall not repeat them here.

13.5.4 File System

MTX implements a simple EXT2 file system, which is described in Chap. 11. The file system is completely Linux compatible.

13.6 Install RMTX

RMTX can be installed to either a real or a virtual PC. The install procedure is described in the Appendix.

13.7 RMTX Startup Sequence

(1). The RMTX kernel image is a binary executable with separate I&D spaces. The first 5 (2-byte) words of the image contains

```
          jmp  start
kds:      .word   0       ! RMTX kernel DS=0x1000+tsize/16
dsize:    .word   0       ! RMTX kernel dsize in bytes
bsize:    .word   0       ! RMTX kernel bss size in bytes
rootdev:  .word   0x0303  ! default root device (major,minor) number
start:                    ! RMTX startup assembly code begins here
```

When compiling the RMTX kernel, the kernel image is first generated as an a.out file with separate I&D spaces. A utility program, h.c, is used to read the file header and patch in the values of kds, dsize and bsize at the word locations 1, 2, 3. Then, it deletes the file header, making the mtx kernel image suitable for booting. In order to be compatible with MTX in protected mode, a bootable RMTX image consists of the following pieces.

```
Sector:|  S0    S1    S3    S4   |  S5 .....................|
       |------|-----|-----|-----|--------------------------|
       | BOOT |SETUP|  APentry  |<------ mtx kernel ----->|
```

where BOOT is for booting the image from a FD, SETUP and APentry are for booting MTX in protected mode. The last 2 bytes in the BOOT sector contain the boot signature, which is PP (0×5050) for protected mode MTX kernel, or RR (0×5252) for real-mode MTX kernel.

(2). Booting RMTX:

Typically, RMTX is installed on a hard disk partition. During booting, the hdbooter loads BOOT+SETUP to 0×90000, APentry to 0×91000 and the RMTX kernel to 0×10000. Then the booter checks the boot signature in the BOOT sector. If the word in 'RR', it jumps to 0×10000 to start the RMTX kernel in real mode. The

booter also writes the boot partition number into word 4 of the RMTX kernel image as the rootdev. When the RMTX kernel starts, it gets the kds segment value at word 1 and uses dsize and bsize to clear the bss section to 0. It sets the segment registers to CS = 0×1000, DS = SS = ES = kds and the stack pointer to the high end of proc[0]. kstack. Then it calls main(), passing rootdev, dsize, bsize as parameters.

(3). RMTX main() function: The actions of main() are as follows:

(3)-1. Call vid_init() to initialize the console display driver to make printf() work. Extract

rootdev minor number as the root device HD.

(3)-2. Call init(HD) to initialize the RMTX kernel. The actions of init() are

Initialize kernel data structures;
Create P0 as the running process.
Initialize interrupt vectors and device drivers. During system initialization, the kernel must read the HD's MBR to find out the start_sector and size of the RMTX partition. Reading HD requires a process to wait on a semaphore for the read operation to complete. Hence P0 must be running first.
Initialize other kernel data structures, e.g. pipes and message buffers.
Initialize memory manager; free memory is from 0×30000 to 0xA0000.
Initialize file system and mount the root file system from rootdev.

(4). After init(), P0 calls kfork("/bin/init") to create P1 and load/bin/init as its Umode image. Then P0 switches to run P1. If all procs are blocked, P0 resumes to a while(1) loop in which it puts the CPU in HLT, waiting for any interrupt. It switches process when the readyQueue is not empty.

(5). Process P1: P1 executes/bin/init in Umode. It forks a login process on the console (/dev/tty0). If desired, it can also fork login processes on serial ports (/dev/ttyS0 and /dev/ttyS1). Then, it waits for any child process to terminate. Another option of P1 is to fork a cdserver process, which runs in Umode and communicates with CDROM client processes by messages. A cdclient process can request the cdserver to display or copy iso9600 files on the CD/DVD-ROM.

(6). Each login process uses exec("login /dev/ttyX") to run the login program on a terminal. It opens the terminal special file 3 times to get the file descriptors 0,1,2 as in, out, err, respectively. Then it displays a login: prompt, waiting for a user to login. Figure 13.1 shows the startup screen of the RMTX operating system.

(7). RMTX is now up and running. When a user tries to login, the login process reads the user name and password. Then it checks the password file,/etc/passwd, to verify the user account. The passwd file has the same format as that of Linux, except that it does not use encryption. If the user has a valid account, the login process becomes the user process by acquiring the user's uid. Then it changes directory to the user's home directory and exec() to run the program listed in the user account, which is usually the command interpreter sh.

(8). The user process in now running sh. The user may enter commands for sh to execute. As of now, the RMTX sh can only execute binary executable files, not sh scripts. However, it supports I/O redirections, background process and multiple commands connected by pipes.

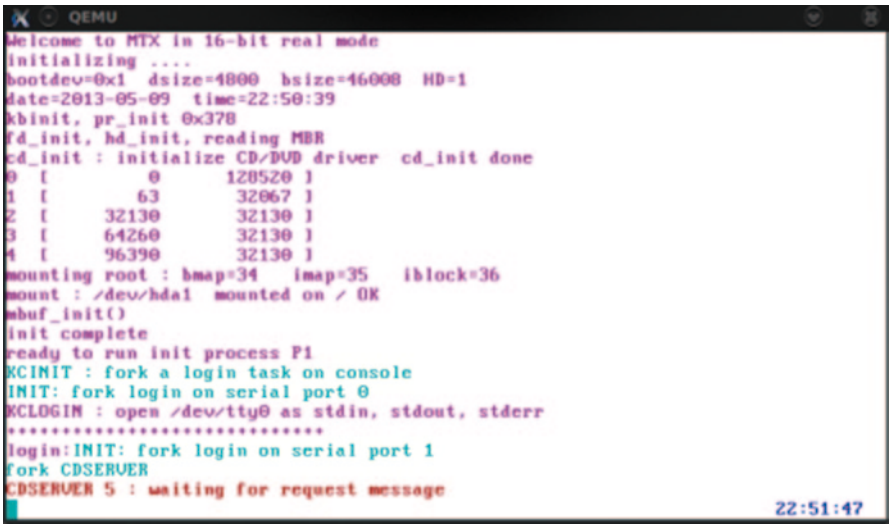


Fig. 13.1 Startup screen of RMTX

(9). When the user logout by entering logout or the Control_D key, the login sh terminates, which wakes up the INIT process P1, which forks another login process on the terminal, etc. In addition, P1 also frees any orphaned ZOMBIE processes.

(10). Shutdown and Reboot RMTX:

RMTX uses I/O buffers to keep recently used disk blocks in a buffer cache. The proper way to shutdown RMTX is by the reboot command or the usual 3-finger salute (Control_Alt_Del). These allow the RMTX kernel to flush the buffer cache and umount any mounted file system before shutdown. At any time, the user may also run the sync command to flush dirty I/O buffers to disk.

13.8 Recompile the RMTX Kernel

To recompile and install the RMTX kernel, from the RMTX source directory, run make followed by make install. Alternatively, mk partition# qemu|vmware, e.g. mk 2 qemu or mk 3 vmware, may also be used to recompile and install the RMTX system. The reader may consult the install procedures in the Appendix for more details.

13.9 Future Development

Although RMTX is a fully functional operating system, it does have some limitations. The most obvious limitation is that it only runs on PCs or virtual machines in 16-bit real mode, so it may not be suitable for practical use. As stated before, the

object of this book is to provide a suitable platform for teaching and learning the theory and practice of operating systems. In order to do these in a meaningful way, it uses the development of RMTX to convey the real intent of this book, which is to show the design principles and implementation techniques of real operating systems in general. On the other hand, small systems like RMTX also have advantages and a wide range of potential applications. The main advantage of RMTX lies in its small size and its ability to run on hardware systems with limited capabilities and resources. The RMTX kernel requires less than 128 KB memory to run, but it supports multitasking, interrupts processing, I/O devices and a complete file system. With the current trend in computing technology, more and more portable devices, such as smart phones, hand-held devices, head-wearing gadgets and embedded systems, etc. will be developed. As such devices and systems become more common, it is only natural for users to expect and demand more functionalities from them, which point to the need for small operating systems designed for simple processors and limited memory space. It is hoped that the design and implementation of RMTX may help shed some light on the development of such systems in the future. In the remaining chapters of this book, we shall extend RMTX to 32-bit protected mode operations.

Problems

1. Create a /sbin directory containing system programs, e.g. login. Set up a /etc/inittab file containing

```
 ID    ACTION       PROCESS (terminal and protocol)
----: --------- :  ---------------------------------
 c1 :  respawn   :  /sbin/login -L tty0  console
 s1 :  respawn   :  /sbin/login -L ttyS0 9600 vt100
 s2 :  respawn   :  /sbin/login -L ttyS1 9600 vt100
```

Rewrite init.c as follows. When init starts up, it forks a login process for each line in the inittab file. The respawn field tells init to fork a new login process on the terminal when the original login process terminates.

2. In RMTX, all commands are in the /bin directory by default. Extend sh to support environment variables, such as PATH=/bin:/sbin:/local/bin, which defines directories containing commands.

3. Extend the sh program to execute simple script files containing command lines.

4. Implement a command history file containing previous input commands. Use the arrow keys to recall previous commands for sh to execute.

5. Currently, RMTX only supports mount/umount of floppy drives containing an EXT2 FS. Modify the RMTX kernel to support mount/umount of hard disk partitions.

6. RMTX uses BCC under Linux for developing both the RMTX kernel and user mode programs. This author has successfully ported BCC's assembler (as86) and linker (ld86) to RMTX, but porting the bcc compiler remains unfinished. Interested readers are invited to port the bcc compiler over to RMTX, which would make RMTX a standalone system.

Chapter 14
MTX in 32-bit Protected Mode

14.1 Introduction to 32-bit Protected Mode

After power on or reset, the x86 CPU (Antonakos 1999) starts execution in 16-bit real mode. The CPU can be switched to 32-bit protected mode (Intel 1990) by setting bit 0 of control register CR0 to 1. Once in protected mode, the CPU's operating environment changes completely. First, the memory management scheme is totally different from that of real mode. Each segment register becomes a segment selector, which is used to select a segment descriptor in a global or local descriptor table. Each segment descriptor specifies the base address, type and size limit of a segment. A logical address comprises a segment selector and a 32-bit offset. The linear address is the segment base address plus the offset. With only segmentation, a linear address is also the physical address. If paging is enabled, the CPU translates the linear address to physical address through two levels of page tables. Second, exception and interrupt vectors are no longer in the low 1 KB of real mode memory. Instead, they are represented by trap and interrupt gates in an interrupt descriptor table. The CPU uses the interrupt descriptor table for exceptions and interrupts processing. The descriptor tables must be set up properly before switching the CPU to protected mode. If using paging, the page tables must also be set up before enabling the paging hardware. The following sections describe protected mode operations in more detail.

14.2 Memory Management in Protected Mode

Memory management consists of address translation and memory protection. In protected mode, the memory management hardware of the x86 CPU supports both segmentation and paging.

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_14

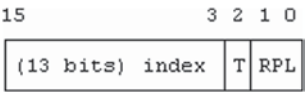


Fig. 14.1 Segment register

14.2.1 Segmentation

A x86 CPU in protected mode has 6 segment registers, denoted by cs, ds, ss, es, fs and gs. Each segment register is 16 bits but its content is no longer a base address as in real mode. Instead, it specifies the index of a segment descriptor in a descriptor table. Each segment descriptor contains the base address and size limit of the intended segment. The linear address, which is also the physical address, is the segment base address plus the offset. The format of a segment register is shown in Fig. 14.1.

where index = 13-bit offset into a descriptor table, T = 0 means the Global Descriptor Table (GDT), T = 1 means the Local Descriptor Table (LDT), RPL is the segment's privilege level for protection. The 2-bit privilege level varies from 00, which is the highest, to 11, which is the lowest. The four privilege levels form a set of protection rings, which can be used to implement secure operating systems with multiple layers of protections. All Unix-like systems use only two privilege levels; kernel level and user level. For MTX in protected mode, we shall also use two privilege levels; RPL = 0 for kernel mode and 3 for user mode. When a process executes at privilege level 0, it can execute any code segment and access any data segment. When a process executes at privilege level 3, it cannot access any segment of RPL 0 directly. This prevents user mode programs from executing kernel code or accessing kernel data. As usual, a user mode process can enter kernel mode only through interrupts, exceptions or by explicit system calls.

In protected mode, a logical address comprises two parts: a 16-bit segment selector and a 32-bit offset, which specifies the relative address within the segment. Given a logical address LA = [segment: offset], the CPU uses the segment selector's T bit to access either the GDT or LDT. If T = 0, it uses the GDT, which is pointed by the CPU's GDTR register. If T = 1, it uses the LDT, which is pointed by the CPU's LDTR register. A system has only one GDT, which specifies the kernel code and data segments that are common to all processes. Each process may have its own LDT, which specifies the user mode address space of that process.

14.2.1.1 Segment Descriptors

Figure 14.2 shows the format of a Code or Data segment descriptor, each 8 bytes long.

Segment Descriptors are in either the GDT or LDT. The CPU's GDTR register points to a GDT descriptor containing the address and size of the GDT. Similarly, the CPU's LDTR register points to a LDT selector in the GDT, which points to the LDT.

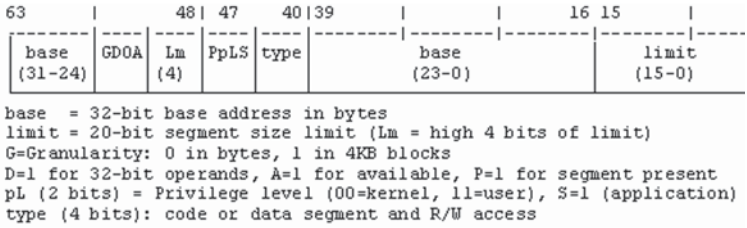


Fig. 14.2 Format of code/data descriptor

When setting up the GDT and LDT it is convenient to define a few segment descriptor prototypes. For example, the following define 4 GB (limit=0xFFFFF) GDT code and data segments.

```
u64 gdt_code = 0x00CF9A000000FFFF; // pL=00,type=0xA=non-conform code
u64 gdt_data = 0x00CF92000000FFFF; // pL=00,type=0x2=read/write data
```

Similarly, the following define 2 MB (limit=0x1FF for 512*4KB) LDT code and data segments.

```
u64 ldt_code = 0x00C0FA00000001FF; // pL=11 type=0xA=non-conform code
u64 ldt_data = 0x00C0F200000001FF; // pL=11 type=0x2=read/write data
```

For different base address, simply change the address field (byte 7 and bytes 4-2) in the prototype. Similarly, change the limit fields for different segment size. In addition to code and data segment descriptors, the GDT may also contain Task State Segment Descriptors (TSSD). A TSSD refers to a Task State Segment (TSS), which is a data structure used by the CPU to save CPU registers during hardware task switching. It also specifies the logical address of the stacks for privilege levels 0, 1 and 2.

14.2.1.2 Segmentation Models

Memory management by segmentation may use several different memory models:

Flat model: In the flat model, all segments have base=0, G=1 (4KB) and limit=0xFFFFF (1M). In this case, virtual address to physical address mapping is one-to-one. A 32-bit virtual address is an offset from 0, so it's also a physical address. A special usage of the flat model is for paging. Before enabling the paging hardware, all segments are set to the maximum size so that the linear address range is 4 GB.

Protected flat model: In the protected flat model, segments are mapped to existing memory by setting the size limit to available physical memory. Any attempt to access memory outside the segment limit will generate a protection error.

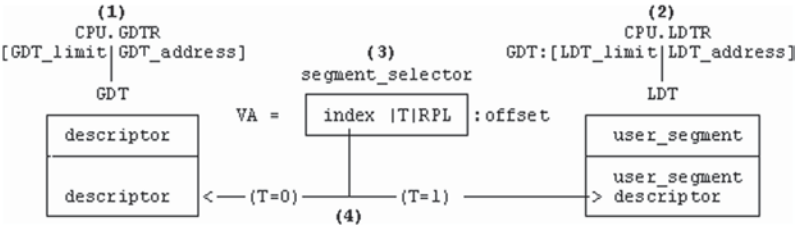


Fig. 14.3 Address translation in segmentation

Multi-segment model: In the multi-segment model, a program may use all the segment registers to access up to 6 protected segments.

Figure 14.3 shows the CPU's address translation and memory protection scheme in segmentation, which is explained by the labels (1) to (4).

(1). The CPU's GDTR register contains a GDT descriptor=[GDT_limit |GDT_address]. The GDT contains 8-byte global segment descriptors. Each segment descriptor has a 32-bit base address, a size limit and a 2-bit privilege level.

(2). Like the GDT, the LDT also contains 8-byte local segment descriptors. Unlike the GDT, which can be located anywhere in memory, the LDT descriptor must be placed in the GDT and load LDTR with its selector in the GDT.

(3). A virtual address VA=[16-bit segment_selector: 32-bit offset].

(4). The 13-bit index is used to access a segment descriptor in either the GDT (T=0) or LDT (T=1). A program executes at the privilege level (CPL) of its code segment selector. When a program uses a selector (RPL) to access a data segment descriptor (DPL), the CPU checks privilege by max(CPL, RPL)<=DPL. If the intended access passes the privilege checking, the physical address PA=segment_base+offset, which must be within the segment limit. A stack segment is a R|W data segment in which the segment type's E bit is set for expand-down and a PA in a stack segment must be greater than the segment limit. As in 16-bit real mode, we shall not use stack segments due to constraints of pointer variables in C.

In general, a program executes in code segments of the same privilege level. It may call procedures in a conforming code segment of higher privilege level but executes at the privilege level of the original program. This feature allows user mode programs to call shared procedures in kernel's conforming code segments. A program can only transfer control to a non-conforming code segment of higher privilege level through gates in the Interrupt Descriptor Table (IDT). For MTX in protected mode, we shall assume two privilege levels and use only non-conforming code segments, which prevent user mode programs from executing kernel code directly.

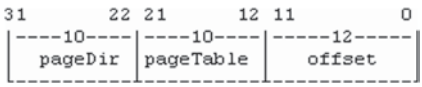


Fig. 14.4 Linear address under paging

14.2.2 Paging

In protected mode, memory can also be managed by paging. In the x86 CPU, paging is implemented on top of segmentation. A logical address is first mapped by segmentation to a linear address. If paging is not enabled, the linear address is also the physical address. If paging is enabled, the linear address is further mapped by the CPU's paging hardware to a physical address. In most paging systems, the term paging refers to pure paging, which does not have any notion of segments. Most Unix-like systems use pure paging. In the x86 CPU, there is no way to disable segmentation but there is a way to get around it. When using paging, we first set up a flat segment model in which all the segments are 4 GB in size. This makes the linear address range from 0 to 4 GB, which effectively hides the segmentation layer, making it transparent. Then we set up page tables and turn on paging. With paging enabled, a 32-bit linear address is treated by the CPU's memory Management Unit (MMU) as a triple = [pageDir, pageTable, offset], as shown in Fig. 14.4.

14.2.2.1 Page Directory and Page Tables

In a linear address, pageDir refers to an entry in a level-1 page table, pageTable refers to an entry in a level-2 page table and offset is the relative address in the page. Normal page size is 4KB. With Page Size Extension (PSE), some x86 CPUs also support super page size of 4 MB. When using paging the MMU first uses the control register CR3 to locate the page directory, which is the level-1 page table. Each page table entry has the format shown in Fig. 14.5.

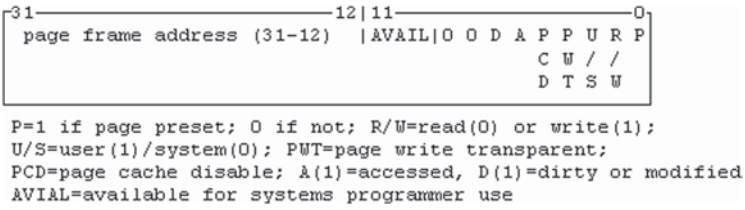


Fig. 14.5 Page table entry format

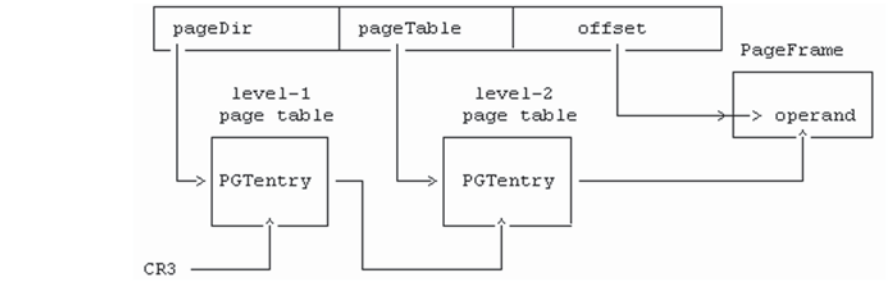


Fig. 14.6 Address translation in paging

14.2.2.2 Address Translation in Paging

Figure 14.6 depicts the address translation procedure of paging. The CPU's CR3 points to the level-1 page table. Given a linear address, the MMU first uses the 10-bit pageDir to locate an entry in the level-1 page table. Assume that the directory page table entry is present and the access checking is OK. It uses the 20-bit page frame address of the page table entry to locate the level-2 page table. Then it uses the 10-bit pageTable to locate the entry in the level-2 page table. Assume that the page entry is present and access checking is OK also. The level-2 page table entry contains the page frame address in memory. The final physical address is PA = (page frame address $\ll 12$) + offset. Since paging usually relies on the flat segment model, protection by checking segment limits no longer makes sense. With paging, protection is enforced by the individual page table entries. A page is either present or not present. Attempt to access a non-present page generates a page fault. In addition, a page table entry can be marked as either read-only or writeable. Attempt to write to a read-only page also generates a page fault. The access (A) and dirty (D) bits can be used to implement page replacement in demand-paging.

14.2.2.3 Translation Lookaside Buffer (TLB)

In order to speed up the paging translation process, the CPU stores the most recently used page table entries in an internal cache, called the TLB. Most paging is performed by using the contents of the TLB. Bus cycles are performed only when a new page is used. Whenever the page tables are changed, the OS kernel must flush the TLB to prevent it from using old page entries in the TLB. Flushing the TLB can be done by reloading the CR3 control register. Individual entries in the TLB can also be flushed by the INVPLG instruction.

14.2.2.4 Full Paging

The simplest paging scheme is full paging. In this scheme, all the pages of a process image are allocated physical page frames at once. After loading a process image

into page frames, the pages are always present. The full paging scheme can be either static or dynamic, depending on how page frames are allocated. In static paging, each image is allocated a single piece of contiguous physical memory aligned to a page boundary. The memory area is divided into a sequence of page frames, which are used as page table entries. The main advantage of static paging is that it is extremely easy to implement. First, process images can be managed as variable-sized partitions by the simple first-fit algorithm. Second, it is very easy to construct the page tables since all the page frames are contiguous. Third, there is no need to maintain a separate data structure to manage free page frames. In dynamic paging, the pages of an image are allocated dynamically. The main advantage of this scheme is that an image can be loaded to any available page frames, which do not have to be contiguous.

14.2.2.5 Demand-Paging

In demand-paging, the page tables of a process image are built by the image size, but not all the pages are allocated page frames. The pages which do not have page frames are marked as not present. The frame address in an absent page table entry may point to its location in a physical device, e.g. a block number in a swap disk containing the page image. During execution, when a process attempts to reference a page that is not present, it generates a page fault, which traps to the OS kernel. The OS kernel's page fault handler can allocate a page frame for the page, load the missing page into the page frame and change the page table entry to present. Then it lets the process continue with the valid page table entry. Demand-paging is the basis of virtual memory, in which the virtual address space of a process can be much larger than the physical memory allocated to it.

14.3 Interrupt and Exception Processing

14.3.1 Exceptions in Protected Mode

Interrupt and exception processing in protect mode differ from that of real mode in two areas. First, the first 32 interrupt vectors are reserved for exceptions, which are listed in Fig. 14.7. The exception vectors overlap with the traditional interrupt vectors of IRQ0 to IRQ7 (0x08 to 0x0F). The IRQ vectors must be remapped to different locations. Second, the exception vectors are no longer in the low 1KB memory area as in real mode. Instead, they are defined as interrupt descriptors in an Interrupt Descriptor Table (IDT).

Exception	Description
0x00	Divide error:
0x01	Single-step/debug exception:
0x02	Non-maskable interrupt:
0x03	Breakpoint by INT 3 instruction
0x04	Overflow
0x05	Bounds check
0x06	Invalid opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor segment overrun
0x0A	Invalid TSS
0x0B	Segment not present
0x0C	Stack exception
0x0D	General protection violation
0x0E	Page fault
0x0F	(Reserved)
0x10	Coprocessor error
0x11-0x1F	(Reserved)

Fig. 14.7 Exception Vectors in Protected Mode

14.3.2 Interrupt Descriptor Table (IDT)

The IDT is a data structure containing interrupt and trap gates. It is pointed by the CPU's IDTR register. The IDT contents are essentially descriptors but Intel chooses to call them interrupt or trap gates. Figure 14.8 shows the format of an interrupt or trap gate.

where P is the present bit, pL is the privilege level, TYPE = 1110 for interrupt gates and 1111 for trap gates. The difference between them is that invoking an interrupt gate automatically disables interrupts but invoking a trap gate does not. Since interrupts and exceptions are processed in kernel mode, the privilege level should be set to 00. It can be set to 11 to allow user mode programs to handle software generated interrupts. For convenience, we define the following prototypes of interrupt and trap gates.

```
u64 int_gate = 0x00008E0000080000; // int  gate(E), seg(08)=Kcode
u64 exe_gate = 0x00008F0000080000; // trap gate(F), seg(08)=Kcode
```

The address fields of the IDT gates can be set to point to the entry points of different exception and interrupt handler functions in the kernel code segment. In addition to interrupt and trap gates, the IDT may also contain call gates and task gates. Calling

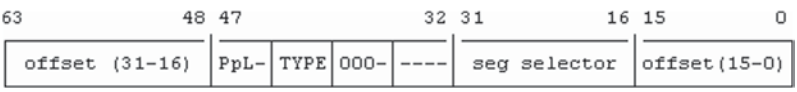


Fig. 14.8 Format of interrupt/trap gate

```
/************* Contents of TSS ***************/
u32 *TSS                    // pointer to next TSS
u32 esp0,ss0;               // privilege 0 esp, ss
u32 esp1,ss1,esp2,ss2;      // privilege 1,2 esp, ss
u32 CR3
u32 eip,eglags,eax,ecx,edx,ebx,esp,ebp,esi,edi
u32 es,cs,ss,ds,fs,gs     // all in low 2 bytes
u32 ldt, iomap
```

Fig. 14.9 Task state segment

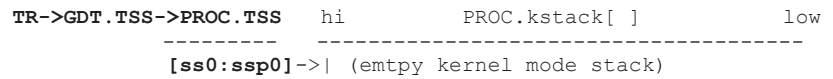
a task or interrupt gate may trigger a task switch by hardware. For MTX in protected mode, we shall not use hardware task switch for the following reasons. First, task switch involves much more than just switching the hardware context of tasks. Second, software task switch is more flexible since it is under the direct control of the OS designer. Last but not least, hardware task switch is only supported in 32-bit CPUs. It is no longer supported in 64-bit x86 CPUs. For MTX in protected mode, we only use TSS to define the process kernel mode stack. To do this, we define a TSS in the PROC structure, place the TSS descriptor in the GDT and let CPU's Task Register (TR) point to the TSS descriptor in the GDT. Figure 14.9 shows the TSS data structure.

14.3.3 Task State Segment (TSS)

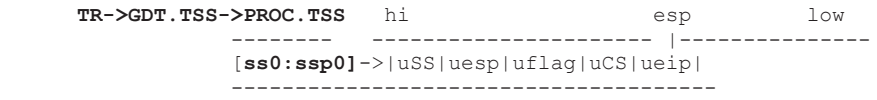
In the TSS structure, the fields from eip to gs are saved CPU registers during hardware task switch. Since we assume only two privilege levels without hardware task switching, the most important fields are esp0 and ss0, which define the CPU's interrupt stack. When an interrupt or exception occurs, the CPU automatically uses [ss0:esp0] as the interrupt stack. Therefore, [ss0:esp0] must point to the kernel mode stack of the current running process. The following discussions help clarify this important point.

14.3.4 TSS and Process Interrupt Stack

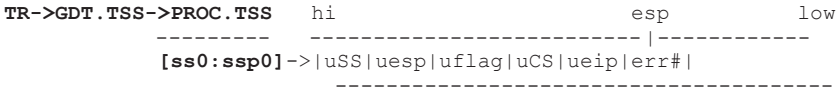
Assume that the CPU is executing a process in user mode. At this moment, the process kernel mode stack is empty. The CPU's TR register points to a TSS selector in the GDT, which points to the process TSS, in which [ss0:esp0] points to the (high end of) the process kernel stack, as shown in the following diagram.



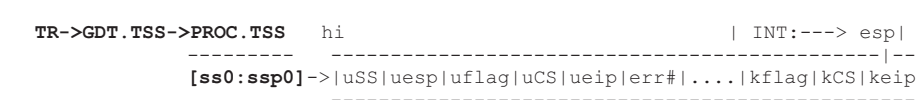
When an interrupt occurs, the CPU saves uSS, uSP, uflags, uCS, ueip into the interrupt stack, which becomes



where the prefix u denotes user mode registers and the saved uSS, uesp are user mode stack at the point of interruption. If an exception occurs in user mode, the situation is exactly the same, except that for some exceptions the CPU also pushes an error number, err#, onto the interrupt stack, which becomes



While in kernel mode, if another interrupt or exception occurs, the CPU continues to use the same interrupt stack to push one more layer of interrupted context. If the CPU is already in kernel mode, re-enter kernel mode does not involve privilege change. In this case, the saved context only has the kernel mode |kflags|kCS|keip|, as shown in the next diagram.



When return from an interrupt/exception handler, the iret operation supports several different types of actions in general. We only consider the relevant case of two privilege levels without hardware task switch. In this case, iret checks the CPLs of the current and the next code segments. If the CPLs are the same, i.e. from kernel back to kernel, it only pops the saved kernel mode registers [keip, kCS,kflags]. If the CPL of the next code segment is greater, i.e. from kernel back to user mode, it pops the saved user mode registers, which includes the saved uesp and uSS.

14.3.5 Process Kernel Stack in Protected Mode

(1). Context switching: As in real mode, context switching in protected mode MTX is also by tswitch(), which is shown below in 32-bit pseudo assembly code.

```
tswitch:
          cli
ENTRY:    pushl  %eax,%ebx,%ecx,%edx,%ebp,%esi,%edi # may use pushal
          pushfl
          movl   running, %ebx    # save Kmode esp in PROC.ksp
          movl   %esp, 4(%ebx)
FIND:     call   scheduler        # find next running PROC
RESMUE:   movl   running, %ebx
          movl   4(%ebx),%esp     # restore running's Kmode esp
          popfl
          popl   %edi,%esi,%ebp,%edx,%ecx,%ebx,%eax # may use popal
          sti
          ret
```

As in real mode, there is no need to save kernel mode segment registers in tswitch() since they are always the same. Corresponding to the ENTRY and RESUME code of tswitch(), the kernel stack frame for a process to resume running is of the form

```
                                          |<-saved PROC.ksp
      |kPC|kax|kbx|kcx|kdx|kbp|ksi|kdi|kflag|
      |<------ for RESUME in tswitch()------|
```

(2). Entry and exit code of interrupt handlers: As in real mode, the entry points of interrupt handlers, including system call, are also defined by an INTH macro, which is shown below in pseudo code.

```
.MACRO INTH handler
   pushl %eax,%ebx,%ecx,%edx,%ebp,%esi,%edi,%ds,%es,%fs,%gs
   set   %ds,%es,%fs,%gs to KERNEL DS_SELECTOR
   movl  running, %ebx   # ebx -> running PROC
   incl  8(%ebx)         # PROC.inkmode++;
   call  handler         # call handler() in C
   jmp   ireturn
.ENDM
```

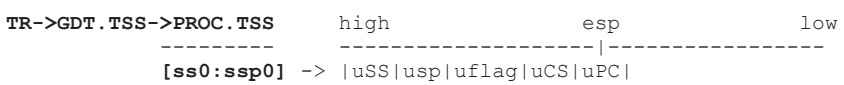
Entry points of interrupt handlers are installed as calls to the INTH macro, e.g

```
int80h: INTH  kcinth      # system call handler
tinth:  INTH  thandler    # timer interrupt handler, etc.
```

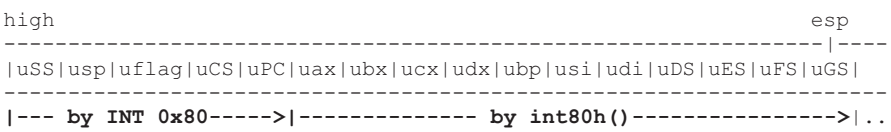
Each interrupt handler calls a handler function in C, which actually handles the interrupt. When the C handler function finishes, it issues EOI and returns to execute the ireturn/goUmode code. If the interrupt occurred in kernel mode, the process restores saved CPU registers and IRET back to the point of interruption directly. Otherwise, it handles signal first and may switch process before returning to user mode. The following shows the pseudo code of ireturn/goUmode.

```
ireturn/goUmode:
       cli
       movl   running, %ebx
       decl   8(%ebx)               # PROC.inkmode--;
       jg     xkmode
       call   kpsig                 # handle signal before return to Umode
       call   reschedule            # call reschedule(); may switch process
xkmode:
      popl   %gs,%fs,%es,%ds,%edi,%esi,%ebp,%edx,%ecx,%ebx,%eax
      iret
```

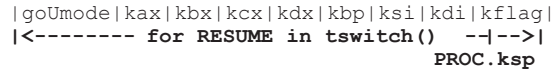
As in real mode, we use a PROC.inkmode counter to keep track of the number of times a process enters kernel mode. In real mode, when a process issues a system call, the CPU saves [uflag, uCS, uPC] in the process user mode stack. The entry code of the interrupt handler, int80h(), continues to save other CPU registers, also in user mode stack. When the system call handler finishes, the process returns to user mode by the interrupt stack frame in the user mode stack. The situation changes only slightly in protected mode. Instead of the user mode stack, all the stack frames are in the process kernel mode stack. In protected mode, we shall implement syscall by INT 0x80, same as in Linux and xv6. When a process issues a syscall, the CPU saves [uSS, usp, uflag, uCS, uPC] into the process kernel stack, which becomes



The syscall handler entry code, int80h(), continues to save other CPU registers in the process kernel stack, which becomes



While in kernel mode, if the process gives up CPU by calling tswitch(), it adds a resume stack frame to the top of the process kernel stack. Therefore, to let a process return to user mode immediately when it resumes running, we can set up its kernel stack to contain two frames; an interrupt stack frame as shown above, followed by a resume stack frame, as shown below.



This technique will be used in all versions of MTX in protected mode.

14.4 GCC in 32-bit Mode

For MTX in 32-bit protected mode, we shall use GCC's assembler, compiler and linker to generate 32-bit code. In 32-bit Linux, GCC generates 32-bit code by default. In 64-bit Linux, use gcc –m32 to generate 32-bit code.

14.4.1 Inline Assembly

The C compiler of GCC supports inline assembly, which is often used in OS kernel code for convenience. The basic format of inline assembly is

__asm__("assembly code"); or simply asm("assembly code");

If the assembly code has more than one line, the statements can be separated by \n\t ; as in

asm("movl %eax, %ebx\n\t; addl $10,%eax\n\t");

Inline assembly code can also specify operands. The template of such inline assembly code is

```
asm ( assembler template
    : output operands
    : input operands
    : list of clobbered registers
    );
```

The assembly statements may specify output and input operands, which are referenced as %0, %1. For example, in the following code segment,

```
int a, b=10;
asm("movl %1,%%eax; movl %%eax,%0;"  // use %%REG for registers
     :"=r"(a)                        // output MUST have =
     :"r"(b)                         // input
     :"%eax"                          // clobbered registers
   );
```

%0 refers to a, %1 refers to b, %%eax refers to the eax register. The constraint operator "r" means to use a register for the operand. The above code is equivalent to

"movl b, %eax\n\t; movl %eax, a";

It also tells the GCC compiler that the eax register will be clobbered by the inline code. Although we may insert fairly complex inline assembly code in a C program, overdoing it may compromise the readability of the program. In practice, inline assembly should be used only if the code is very short, e.g. a single assembly

instruction or the intended operation involves a CPU control register. In such cases, inline assembly code is not only clear but also more efficient than calling an assembly function.

14.4.2 Constant, Volatile and Packed Attributes

In a C program, variables or function parameters that are intended for read-only can be declared with a const qualifier to prevent them from being modified accidentally. If the program code tries to modify a const item, it will be detected as an error at compile-time. For static constants, such as the strings in char *p="abcd"; printf("%d %s\n", ….); etc. the C compiler usually puts them in a read-only data section. During execution, trying to write to read-only memory will generate a segmentation fault. The GCC is an optimizing compiler. When writing device drivers, interrupt handlers or multi-thread C code, it is often necessary to declare variables as volatile, which prevents the GCC compiler from optimizing the code involving such variables. Likewise, the packed attribute of a C structure tells the GCC compiler not to pad the structure with extra bytes for memory alignment.

14.4.3 GCC Linker and Executable File Format

GCC's linker can generate executable files in several different formats. In Linux, the default executable file format is ELF (Tool 1995; Youngdale 1995), which is better suited to dynamic linking. Since MTX does not yet support dynamic linking, we shall use ELF format for statically linked files only.

(1). Linker Script: A linker script is a text file which specifies linking options to the linker. Its usage is

ld -T ld.script [other linker flags]

As an example, the following shows a simple linker script, which may be used to generate either flat binary or ELF executable files.

```
/*************** ld.script ***************/
OUTPUT_FORMAT("binary")
/* OUTP UT_FORMAT("elf32   -linux") */
OUTPUT_ARCH(i386)
ENTRY(u_entry)
SECTIONS
{ . = 0x0;  /* change this if begin VA is not 0 */
   .text : { *(.text) }
   .data : { *(.data) }
   .bss  : { *(.bss)  }
}
```

A flat binary executable file is a single piece of binary code, which does not have separate code and data sections. By changing the output file format to "elf32-linux",

the linker will generate an ELF executable file. For this example, the resulting ELF file has only one (loadable) program section. If desired, the linker script can be modified to generate an ELF executable with separate code, data and bss sections, as in

```
SECTIONS
{ . = 0x0;
   .text : { *(.text) }
   . = 0x8000;
   .data : { *(.  data) }
   . = 0xC000;
   .bss  : { *(.bss)  }
}
```

The resulting ELF file will have separate code, data and bss sections, each with a different starting virtual address.

(2). ELF executable file format: An ELF executable file begins with an elf-header, followed by one or more program section headers, which are defined as

```
struct elfhdr {   // ELF File header
  u32 magic;      // ELF_MAGIC 0x464C457F
  u8  elf[12];
  u16 type;
  u16 machine;
  u32 version;
  u32 entry  ;
  u32 phoffset;   // byte offset of program header
  u32 shoffset;   // byte offset of sections
  u32 flags;
  u16 ehsize;     // elf header size
  u16 phentsize;  // program header size
  u16 phnum;      // number of program section headers
  u16  shentsize;
  u16 shnum;
  u16 shstrndx;
};
// ELF Program section header
struct proghdr {
  u32 type;       // 1 = loadable image
  u32 offset;     // byte offset of program section
  u32 vaddr;      // virtual address
  u32 paddr;      // physical ad     dress
  u32 filesize;   // number of bytes of program section
  u32 memsize;    // load memory size
  u32 flags;      // R|W|Ex flags
  u32 align;      // alignment
};
```

(3). ELF executable file loader: When loading an ELF file the loader must load the various sections of an ELF file to their specified virtual addresses. In addition, each loaded section should be marked with appropriate R|W|Ex attributes for protection.

For example, the code section pages should be marked as REx, data section pages should be marked as RW, etc.

(4). Program Loader in Protected Mode: The program loader of MTX in protected mode can load either flat binary or ELF image files. The loader's algorithm is as follows.

```
/*****************  Loader Algor ithm *********************/
  Open the image file for read (Use MTX's internal open)
  Read the elf -header to check whether it's an ELF file;
  if (!ELF){ // assume flat BINARY file
       determine file size; (by MTX's internal kfdsize())
       load file contents to process image area;
       return;
  }
  /********** ELF file *************/
  locate program header(s);
  for each program header do{
       get section's offset, loading address and memory size;
       load section to virtual address until memory size;
       set section's R|W|Ex attributes in loaded pages;
  }
```

In summary, for protected mode operations we must set up the GDT, remap IRQ vectors, install interrupt and exception handlers in the IDT and define CPU's interrupt stack by a TSS in the GDT. As in real mode, knowing the kernel mode stack contents is essential to understanding how to set up process for execution in protected mode. In the following sections, we shall describe the design and implementation of MTX in 32-bit protected mode. In order to illustrate the different capabilities of the PC's memory management hardware, we shall present three different versions of MTX in 32-bit protect mode, each uses a distinct memory management scheme. The three versions of MTX in 32-bit protected mode are

MTX32.1, which uses protected segments.

MTX32.2, which uses static paging.

PMTX, which uses dynamic paging.

Among these, MTX32.1 and MTX32.2 are intended for demonstration only. PMTX is the final version of MTX in 32-bit protected mode for uniprocessor systems.

14.5 MTX Kernel in 32-bit Protected Mode

The kernel source tree of MTX in 32-bit protected mode is organized as

```
                    |-- SETUP  : boot.s, setup.s, apentry.s
MTX_kernel--|--kernel  : kernel files
                    |--fs      : file system
                    |--driver  : device drivers
```

Each directory has a Makefile for GCC's compiler and linker. The top level Makefile invokes the Makefile of kernel, which in turn invokes those of fs and driver to generate the various objects code files. The object code files are linked by GCC's ld linker to generate a flat binary executable, which is the 32-bit MTX kernel. Then a sh script is used to assemble the binary files of BOOT, SETUP, APentry and MTX kernel into a bootable MTX image consisting of the following pieces.

```
Sector | 0  |  1  |  2    3   | 4 ......................|
       ------------------------------------------------
       |BOOT|SETUP|  APentry  | 32-bit MTX kernel ......|
       ------------------------------------------------
```

In the bootable image, BOOT is for booting the MTX kernel image from a floppy disk. In protected mode, it is used as a communication area between the booter and kernel. SETUP is for transition from 16-bit real mode to protect mode and APentry is the startup code of non-boot processors in SMP mode (SMP MTX is covered in Chap. 15). When installed to a hard disk partition, the entire bootable image is a file,/boot/mtx, in an EXT2/3 file system. A suitable hard disk booter, e.g. the hd-booter developed in Chap. 3 of this book, is needed to boot up the MTX kernel. The booting sequence is similar to booting Linux as described in Chap. 3.

14.5.1 Protected Mode MTX Kernel Startup Sequence

(1). Hd-booter: During booting, the hd-booter loads BOOT+SETUP to 0x90000, APentry to 0x91000 and the MTX kernel to 0x10000. It also writes some boot parameters, such as Time-of-Day (TOD) of BIOS, the boot device number and the start sector of the boot partition, etc. to the BOOT sector for the kernel to use. Then it checks the boot signature at the end of the BOOT sector. If the word is 'RR', it jumps to 0x10000 to run MTX in 16-bit real mode. If the word is 'PP', it jumps to 0x90200 to run SETUP, which brings up MTX in 32-bit protected mode.

(2). SETUP: SETUP is a piece of 16-bit and 32-bit mode code in GCC assembly. It sets up an initial GDT containing 4 GB code and data segments and enters protected mode. Then it moves the MTX kernel to 1 MB (0x100000) and jumps to the entry address of the MTX kernel. The actions of SETUP are identical for all 32-bit MTX. Their only difference is in the number of segments in the initial GDT. During booting, SETUP serves as a transition stage from 16-bit real mode to 32-bit protected mode. Since the transition is crucial, we show the entire SETUP code of MTX32.1 and explain the steps in detail.

```
#---------------- setup.s file for MTX32.1 ---------------------------
          .text
          .set KCODE,   0x08     # kernel code segment selector
          .set KDATA,   0x10     # kernel data segment selector
          .set K_ORG,   0x10000  # MTX kernel loaded here by booter
          .set K_HIM,   0x100000 # MTX kernel running address (1MB)
          .set GDT_ADDR,0x9F000  # hard coded GDT address
          .set GDT_SZIE 40       # GDT size in bytes
          .org 0
.code16  # 16-bit code
# set segments registers cs,ds,ss=0x9020, stack size=8KB
          ljmp  $0x9020, $go
go:       movw  %cs, %ax
          movw  %ax, %ds         # ds = 0x9020
          movw  %ax, %ss         # stack segment = 0x9020
          movw  $8192,%sp        # 8KB stack size
# mov setup_gdt to GDT_ADDR=0x9F000
          movw  $GDT_ADDR>>4,%ax
          movw  %ax, %es
          movw  $gdt,%si
          xorw  %di, %di
          movw  $GDT_SIZE>>2,%cx
          rep
          movsl
# load GDTR with gdt_desc = [GDT_limit|GDT_ADDR]
          lgdt  gdt_desc
# enter protected mode by writing 1 to CR0
          cli
          movl  %cr0,%eax
          orl   $0x1,%eax
          movl  %eax,%cr0
# do a "ljmp" to flush instruction pipeline and set CS to KCODE
          .byte 0x66, 0xea       # prefix + ljmp-opcode
          .long 0x90200+next     # SETUP is at 0x90200
          .word KCODE            # CS selector
# the handcrafted ljmp is equivalent to ljmpl of GCC's 32-bit assembly
#         ljmpl $KCODE,$(0x90200+next)

.code32  # 32-bit code
next: # load other selectors with 0x10 for kernel data entry in GDT
          movl  $KDATA,%eax
          movw  %ax,   %ds
          movw  %ax,   %es
          movw  %ax,   %fs
          movw  %ax,   %gs
          movw  %ax,   %ss
# move MTX kernel from 0x10000 to 0x100000, then jump to 0x100000
          cld
          movl  $K_ORG,%esi
          movl  $K_HIM,%edi
          movl  $1024*64,%ecx    # assume MTX kernel < 256KB
          rep
          movsl
          ljmp  $KCODE,$K_HIM       # ljmp to 0x08:0x100000
gdt:      .quad 0x0000000000000000  # null descriptor
          .quad 0x00CF9A000000FFFF  # kernel cs
          .quad 0x00CF92000000FFFF  # kernel ds
          .quad 0x0000000000000000  # task tss
          .quad 0x0000000000000000  # task ldt
gdt_desc: .word .-gdt-1
          .long GDT_ADDR # hard coded at 0x9F000; can be changed
          .org 512
#------------- end of setup.s file ---------------------------------
```

Upon entry, SETUP begins execution in 16-bit real mode. First, it initializes the CPU's execution environment by setting the segment registers CS, DS, SS to 0x9020 and SP to 8 KB. For MTX32.1, which uses protected segments, SETUP defines an initial GDT containing 5 entries, in which only the kernel code and data segments are initialized as 4 GB segments. It moves the initial GDT to a known physical address, GDT_ADDR = 0x9F000, and loads the GTDR with the 6-byte GDT descriptor at gdt_desc. Then it enters protected mode by setting bit 0 of control register CR0 to 1. Then it executes a handcrafted ljmp to flush the instruction pipeline, preventing the CPU from executing any stale 16-bit code. The ljmp also loads the CS register with KCODE = 0x08, which refers to the kernel code segment descriptor in the GDT. The handcrafted ljmp instruction is equivalent to

```
ljmpl $KCODE, $(0x90200+next)
```

which is supported by GCC's assembler. Then it loads other segment registers, ds to gs, with KDATA = 0x10, which refer to the data segment descriptor in the GDT. The CPU is now executing 32-bit code in protected mode, and it can access the entire physical memory via segmentation. It moves the MTX kernel from 0x10000 to 0x100000 (1 MB). Finally it ljmp to [0x08: 0x100000], which sends the CPU to pmode_entry in the MTX kernel. In both MTX32.1 and MTX32.2, the kernel's starting address is 0x100000 (1 MB), which is directly accessible via segmentation. Therefore, their entry.s file is very simple.

(2). entry.s file:

```
               .global pmode_entry
               .extern p0stack    # high end address of proc[0].kstack
               .org 0             # virtual address=0x100000 (1MB)
pmode_entry:   .code32
                cli
                movl p0stack,%esp # set stack pointer to P0's kstack
                call init
```

pmode_entry is the entry point of the MTX kernel and p0stack is the high end address of proc[0].kstack. It sets the stack pointer to p0stack and calls init() in init.c.

(3). init.c: The following shows the init() function of mtx32.1.

```
/******************** init.c file ***********************/
#include "include.h"                // extern symbols in MTX kernel
int init()
{   vid_init();                     // initialize display driver
    printf("Welcome to MTX in 32-bit Protected Mode\n");
    kernel_init(); // initialize kernel data structs, create and run P0
    // ----------  initialize IRQ and IDT -------------------
    printf("remap IRQs; install IDT vectors\n");
    remap_IRQ();                    // remap IRQ0-15 to 0x20-0x2F
    trap_install();                 // install trap handlers
    printf("install interrupt vectors and handlers\n");
    int_install(0x20, (int)tinth);  // timer IRQ was   0
    int_install(0x21, (int)kbinth); // KBD   IRQ was   1
    int_install(0x23, (int)s1inth); // serial port 1   3
    int_install(0x24, (int)s0inth); // serial port 0   4
    int_install(0x26, (int)fdinth); // FD IRQ was      6
    int_install(0x27, (int)printh); // printer IRQ was 7
    int_install(0x2E, (int)hdinth); // IDE0 was IRQ   14
    int_install(0x2F, (int)cdinth); // IDE1 was IRQ   15
    int_install(0x80, (int)int80h); // syscall  =   0x80

   //-------------------------------------------------------
    printf("initialize I/O buffers and device drivers\n");
    binit();               // I/O buffers
    fd_init(); hd_init(); kb_init(); serial_init(); etc.
    timer_init();          // timer; 8-byte BIOS TOD at 0x90000
    fs_init();             // initialize FS and mount root FS
    running->res->cwd = root; // set P0's CWD to root
    main();                // call main() in t.c
}
```

Init() first initializes the display driver to make printf work. It calls kernel_init() to initialize the kernel data structures, create and run the initial process P0. Then it remaps IRQs, sets up the IDT and installs exception and interrupt vectors and handlers. Besides memory management, this is the second major difference between protected mode and real mode. Since these steps are needed for all protected mode operations, regardless of the memory management scheme used, we shall explain the steps once in detail.

14.5.2 Interrupt and Exception Processing

14.5.2.1 Remap IRQ Vectors

The interrupt vectors of IRQ 0-15 are remapped to 0x20-0x2F by programming the 8259 PICs.

```
#define PIC1 0x20         // master PIC controller register
#define PIC2 0xA0         // slave  PIC controller register
#define ICW1 0x11         // command word 1
#define ICW4 0x01         // command word 4
void remap_IRQ(int irq0, int irq8)
{
  out_byte(PIC1, ICW1);   // write ICW1 to both PICs command regs
  out_byte(PIC2, ICW1);
  out_byte(PIC1+1, irq0); // write ICW2 to both PICs data register
  out_byte(PIC2+1, irq8);
  out_byte(PIC1+1, 4) ;   // write ICW3 for cascaded PICs
  out_byte(PIC2+1, 2);
  out_byte(PIC1+1, ICW4); // write ICW4 for 8086 architecture
  out_byte(PIC2+1, ICW4);
}
remap_IRQ(0x20, 0x28);    // remap IRQ0-15 to 0x20-0x2F
```

14.5.2.2 IDT and Exception Handlers

The Interrupt Descriptor Table (IDT) has 256 entries, which requires 256*8 = 2KB space. The IDT can be located anywhere in memory. In MTX32.1 and MTX32.2, it is placed directly after the GDT but aligned to a 16-byte boundary for better CPU data cache efficiency. The CPU's IDTR is loaded with the 6-byte IDT descriptor

```
                ------------------------------------------------
CPU.IDTR = |u16 size=256*8 -1| u32 address=IDT _ADDR (0x9F040) |
                ------------------------------------------------
```

The IDT must be set up properly before the CPU can accept interrupts or handle exceptions. The steps to set up the IDT entries are as follows.

(1). IDT and Exception Handlers: The first 32 IDT entries are filled with trap gates, each points to the entry address of an exception or trap handler function.

(a). Exception Handler Entry Points: Among the 32 exceptions, some exceptions generate an error number on the interrupt stack while others do not. In order to generate appropriate entry code for different exception handlers, we define the macros, ECODE and NOECODE, which take a vector number n = 0x00 to 0x1F as parameter.

```
        .MACRO  ECODE n       # for exceptions with error number
          .global trap\n
  trap\n:  pushl  $\n         # push vector number
          jmp    trap_all
        .ENDM

        .MACRO NOECODE n      # for exceptions without error number
          .global trap\n
  trap\n:  pushl  $0          # push a 0 as error number
          pushl  $\n          # push vector number
          jmp    trap_all
        .ENDM
```

In both macros, trap_all is a piece of assembly code common to all exception handlers. For each exception vector number, 0x00 to 0x1F, call the macro ECODE or NOECODE, depending on whether the exception generates an error number or not. For example, for the vector 0x00, which corresponds to divide error, call NOECODE 0x00 because it does not generate an error number. For the vector 0x0E, which corresponds to page fault, call ECODE 0x0E because it generates an error number, etc. These macro calls generate the entry points labeled trap0x00 to trap0x1F, which are the contents of the trapEntry[ ] table.

```
int (*trapEntry[ ])() =
{ /******************* 32 trap/fault entryPoints *********************/
trap0x00,trap0x01,trap0x02,trap0x03,trap0x04,trap0x05,trap0x06,trap0x07,
trap0x08,trap0x09,trap0x0A,trap0x0B,trap0x0C,trap0x0D,trap0x0E,trap0x0F,
trap0x10,trap0x11,trap0x12,trap0x13,trap0x14,trap0x15,trap0x16,trap0x17,
trap0x18,trap0x19,trap0x1A,trap0x1B,trap0x1C,trap0x1D,trap0x1E,trap0x1F
};
```

In the IDT, the first 32 trap gates point to the entry points of trap handlers. The remaining 224 entries are filled with a default trap gate. This is done by trap_install(), which calls trap_entry() to install trap handlers in the IDT. It also loads the IDTR pointing to the IDT.

```
/************** set interrupt/trap gates in IDT ******************/
struct idt_descr{       // to be loaded into CPU's IDTR
       u16 length;
       u32 address;
} __attribute__((packed)) idt_descr = {256*8-1, IDT_ADDR};

void trap_entry(int index, u32 entryPoint)
{
   u64 idt_gate = 0x00008F0000080000ULL; // 8F:DPL=0,TRAP gate, 8=KCODE
   u64 addr = (u64)entryPoint;
   idt_gate |= (addr<<32) & 0xffff000000000000ULL;
   idt_gate |= addr & 0xffff;
   idt[index] = idt_gate;
}
void trap_install()
{  int i;
   for (i=0; i<32; i++)                    // 32 exception vectors
       trap_entry(i, (int)(trapEntry[i]));
   for (i=32; i<256; i++)                  // fill rest with default
       trap_entry(i, (int)default_trap);
   asm("lidt %0\n\t"::"m"(idt_descr));     // load CPU's IDTR
}
```

(b). All exception handlers jump to the same assembly code, trap_all, which is

```
trap_all:
            pushal      # save all CPU registers in stack
            pushl  %ds  # save seg registers, may be of Umode
            pushl  %es
            pushl  %fs
            pushl  %gs
            pushl  %ss
            # change to kernel data segments
            movw   $KDATA,%ax
            movw   %ax, %ds
            movw   %ax, %es
            movw   %ax, %fs
            movw   %ax, %gs

HIGH --------- exception stack contents at this moment ------------esp
|oss|osp|oflg|cs|eip|0/err#|nr|ax|cx|dx|bx|esp|bp|si|di|ds|es|fs|gs|ss|
|<----- by exception ----->|  |<----- by pushal ------>| push ds-ss ->|
|                          13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
-------------------------------------------------------------------------
            movl   running, %ebx  # ruuning->inkmode++
            incl   8(%ebx)        # inkmode at byte offset 8 in PROC
            movl   52(%esp),%ecx  # get nr at byte offset 4*13=52
# Use nr to call the corresponding handler (*trapHandler[nr])()
            call   *trapHandler(, %ecx, 4) # use 0 + %ecx*4 as index
  ereturn:
            movl   running, %ebx  # running->inkmode--
            decl   8(%ebx) # test whether was in Umode or Kmode
            jg     kreturn # to kreturn if was in Kmode
            call   kpsig   # return to Umode: handle signal first
  kreturn: # return to point of exception; kmode traps never return
            addl $4, %esp       # for %ss
            popl %gs
            popl %fs
            popl %es
            popl %ds
            popal
            addl $8, %esp       # pop vector_nr and err_code
            iret
```

trap_all saves all CPU registers on the interrupt stack and changes data segments to that of kernel. Then, depending on the vector number nr, which was pushed on the interrupt stack by entryPoint (at byte offset 4*13 = 52), it calls the corresponding handler function in the trapHandler[ ] table.

(c). trapHandler[ ] is a table of function pointers to the entry addresses of 32 exception handler functions, which are defined as divide_error(), general_protection(), page_fault(), reserved(), etc.

```
int (*trapHandler[ ])()= /* exception handler function pointers */
{
  divide_error, debug_exception,  nmi,            breakpoint,
  overflow,     bounds_check,     invalid_opcode, cop_not_avail,
  double_fault, overrun,          invalid_tss,    seg_not_present,
  stack_exception, general_protection, page_fault,reserved,
  floating_point,  reserved,      reserved,       reserved,
  reserved,        reserved,      reserved,       reserved,
  reserved,        reserved,      reserved,       reserved,
  reserved,        reserved,      reserved,       reserved
};
```

(d). Signal number: In all Unix-like systems, an exception is converted to a signal number delivered to the running process. Many exception handler functions share the same entry code because they are assigned the same signal number. In MTX, exceptions (traps in Unix term) are converted to signal numbers as follows.

```
# all these traps in Umode get the SIGFPE (8) signal
  divide_error: floating_point: cop_not_avail:
                pushl $0x08  #SIGFPE 8
                call  ehandler
                addl  $4,%esp
                ret
```

all these traps in Umode get the SIGTRAP (5) signal

debug_exception: nmi:breakpoint: overflow: bounds_check: invalid_opcode: overrun:

```
invalid_tss:reserved:
                pushl $0x05        #SIGTRAP 5
                call  ehandler
                addl  $4,%esp
                ret
```

Treat double_fault as ABORT(6)

```
double_fault:  pushl $0x06        #SIGABRT 6
                  call  ehandler
                  addl  $4,%esp
                  ret
```

consider these as SIGMENTATION FAULTS (11):
general_protection:seg_not_present:stack_exception: page_fault:

```
pushl $0x0B        #SIGSEGV 11
 call  ehandler
 addl  $4,%esp
 ret
```

Each handler function pushes a signal number corresponding to the exception on the interrupt stack and calls ehandler(), which actually handles the exception.

14.5.2.3 The Ultimate Exception Handler in C

The C function, ehandler(), is the ultimate handler of all exceptions. Upon entry, the process interrupt stack contains

```
|SS|SP|FLAG|CS|PC|err/0|nr|ax|cx|dx|bx|sp|bp|si|di|ds|es|fs|gs|ss| retPC  |sig|
|-- by exception -->|entry|---------- by trap_all   ---------->|trap0xYY|
```

Thus, ehnadler() can be written with all the interrupt stack contents as parameters.

```
void ehandler(sig,retPC,ss,gs,fs,es,ds,edi,esi,ebp,esp,ebx,edx,ecx,eax,
      nr,err,eip,cs,eflags,old_esp,old_ss)    //all parameters are u32
{
  if (exception occurred in Umode){
      send signal sig to current running process;
      printf("proc%d trap%d in Umode:sig=%d\n",running->pid,nr,sig);
      setsig(sig);   // send signal sig to running process
  }
  else{              // exception occurred in Kmode
      printf("proc%d kernel PANIC trap=%x\n", nr);
      display CPU registers (parameters) for kernel debugging
      halt();
  }
}
```

In ehandler(), if the exception occurred in user mode, it sends the signal number, sig, to the running process. When the process returns to ereturn: (in trap_all), it checks for signals and handles any pending signal, as described in Chap. 9 on signal processing. The process may return to Umode to continue if it survives the signal processing. If the exception occurred in kernel mode, which must be due to either

```
exception_nr:->IDT[nr]->trapnr:->trap_all:->trapHandler[nr]:sig# ->
   ehandler(sig#, CPU registers, nr, error#, interrupt_point){
            if (trap in Umode)
               send sig# to running process;
            else
               PANIC and halt;
   }
```

Fig. 14.10 Exception processing sequence

a hardware error or most likely a kernel bug, there is nothing the kernel can do. So it prints a PANIC message, displays the CPU registers for debugging and stops. The exception processing sequence is rather long and complex, which may be quite confusing for most beginners. It may help by looking at Fig. 14.10, which depicts the control flow of the exception processing sequence.

In Fig. 14.10, when an exception of vector nr=0xXY occurs, the CPU uses the vector number nr to access the trap gate in the IDT, which routes the CPU to the entry point trap0xXY. In trap0xXY, it pushes a 0 and nr on the stack, if no error number, or just nr, if the exception generates an error number. Then it jumps to the common code trap_all, which saves all CPU registers in the interrupt stack. It uses the vector number nr to call the corresponding handler function in trapHandler[]. trapHandler[nr] pushes a signal number on the stack and calls the ultimate ehandler() in C. By then the interrupt stack contains all the information about the exception, which can be accessed as function parameters. In ehandler(), if the exception occurred in Umode, the process gets a signal number and handles the signal before return to Umode. If the exception occurred in Kmode, the kernel displays a PANIC message and stops. The logic is quite simple. The seemingly complexity is because we are trying to set up the exception handlers in a systematic way, rather than trying to do it one at a time and repeat it 32 times.

14.5.2.4 Interrupt Vectors and Handlers

Interrupt vectors and handlers of I/O devices are installed by int_install(), which is

```
int int_install(int vector, int entryPoint)
{
 u64 int_gate = 0x00008E0000080000ULL; // 8E:DPL=0,INT gates;8=KCODE
 if (vector==0x80)                     // syscall vector
    int_gate = 0x0000EE0000080000ULL;  // EE:DPL=3,INT gate; 8=KCODE
 int_gate |= ((u64)entryPoint <<32) & 0xffff000000000000ULL;
 int_gate |= ((u64)entryPoint) & 0xffff;
 idt[vector] = int_gate;
}
```

For the interrupt vectors 0x20 to 0x2F, the IDT gates are interrupt gates with RPL=0. For the system call vector 0x80, it is an interrupt gate with RPL=3. It may also be a trap gate with RPL=3. In that case, interrupts will be enabled during syscall. The kernel must explicitly disable interrupts in critical regions. As in real mode, interrupt handler entry points, tinth, kbinth, etc. are installed by an INTH macro.

```
   .MACRO INTH handler
    pushl %eax   # save CPU registers in    KSTACK; may use pushal
    pushl %ebx
    pushl %ecx
    pushl %edx
    pushl %ebp
    pushl %esi
    pushl %edi
    pushl %ds            # save CPU selectors
    pushl %es
    pushl %fs
    pushl %gs
    movw  $DATA_SEL,%ax  # change to kernel data segments
    movw  %ax,%ds
    movw  %ax,%es
    movw  %ax,%fs
    movw  %ax,%gs
    movl  running,%ebx   # %ebx points to running PROC
    incl  INK(%ebx)      # INK=8: running   ->inkmode++
    call  \handler       #  \handler is a parameter
    jmp   ireturn
   .ENDM



# interrupt handler entry points
tinth:  INTH  thandler       # PIC timer
kbinth: INTH  kbhandler      # KBD
# Similarly for other interrupt handlers

ireturn/goUmode:             # return to point of interruption
        cli
        movl  running, %ebx  # %ebx points at running PROC
        decl  INK(%ebx)      # running->inkmode--
        jg    xkmode         # if interrupt occurred in Kmode
        call  kpsig          # before return to Umode, handle signal
        call  reschedule     # switch task if sw_flag is set
xkmode:
        popl  %gs
        popl  %fs
        popl  %es
        popl  %ds
        popl  %edi           # may use popal
        popl  %esi
        popl  %ebp
        popl  %edx
        popl  %ecx
        popl  %ebx
        popl  %eax           # NOTE: eax = syscall return value
        iret
```

Summarizing, we note that remap IRQ, set up the IDT and install exception/interrupt vectors are necessary for all protected mode operations, regardless of the memory management scheme used. The good news is that we only need to set up the IDT once for all protected mode operations.

14.6 MTX32.1: 32-bit Protected Mode MTX Using Segmentation

The first version of 32-bit MTX is MTX32.1, which uses protected segments. In MTX32.1, the PROC structure is modified to contain a TSS and 2 LDT descriptors.

```
typedef struct proc{
    // same as bef ore but ADD these for segmentation:
    TSS tss;           // TSS type in type.h
    u64 ldt[2];        // Umode code and data segments
    u32 kstack[SSIZE]; // PROC kstack must be the last field
} PROC;
```

The TSS defines the process interrupt stack and the LDT descriptors define the user mode code and data segments of a process. In kernel mode, the code and data segments are in the GDT, which are full 4 GB segments. If the system has less than 4 GB physical memory, the kernel segment size can be set to the actual amount of physical memory. In user mode, each process has a fixed size Umode image. The MTX kernel runs at 1 MB. Currently, the run-time size of the MTX.32.1 kernel is less than 1 MB. Assume that the maximum run-time kernel size is 1 MB. Then the memory area above 2 MB is free. We shall allocate process user mode images above 2 MB. User mode images can be generated as either flat binary or ELF executable files. Flat binary executables do not have separate code, data and bss size information. Without the size information, we can only assume a fixed image size, e.g. 2 MB for every Umode image. ELF executable files have size information if they are generated with separate code, data and bss sections. To simplify memory management, we shall assume a fixed image size of 2 MB for ELF files also. The actual image size is much smaller, but allocating a larger memory area causes no harm. Specifically, we shall assume that the Umode image of a process is allocated at pid*2 MB, e.g. P1 at 2 MB, P2 at 4 MB, etc. Since the purpose here is to demonstrate segmentation, these assumptions are not important. If desired, the reader may experiment with different image size and memory layout. The startup sequence of MTX32.1 is as follows.

14.6.1 MTX32.1 Kernel Startup Sequence

(1). SETUP: It defines an initial GDT containing 5 entries. In the initial GDT, the kernel code is a non-conforming code segment, so that user mode processes can only execute kernel code through interrupt or trap gates in the IDT.

```
gdt: .quad  0x0000000000000000   # null descriptor
     .quad  0x00cF9A000000ffff   # kernel cs,0xA=nonconform segment
     .quad  0x00cF92000000ffff   # kernel ds, 0x2=R|W data segment
     .quad  0x0000000000000000   # task tss
     .quad  0x0000000000000000   # task ldt
gdt_desc: .word -gdt-1          #  gdt size-1
          .long  GDT_ADDR        # hard coded as 0x9F000
```

After loading SETUP to 0x90200, the booter jumps to 0x90200 to run SETUP, which begins execution in 16-bit real mode. As described in Sect. 14.5.1, SETUP uses the initial GDT to enter protected mode, move the MTX kernel to 1 MB and ljmp to [0x08:0x100000], which sends the CPU to the entry point, pmode_entry, of the MTX kernel.

(2). entry.s:

pmode_entry sets the stack pointer to P0's kstack high end and calls init() in init.c.

(3).

```
int init()
 {
    vid_init();    // initialize display driver
    printf("MTX in 32-bit Protected Mode using Segmentation\n");
    kernel_init(); // initialize kernel, create and run P0
    remap IRQs; install IDT vectors
    initialize I/O buffers and device drivers
    fs_init();  // initialize file system and mount root device
    set P0's CWD to root directory;
    main();     // call main() in t.c
 }
```

init() first initializes the display driver to make printf work. It calls kernel_init() to initialize kernel data structures, create and run the initial process P0. Then it remaps the IRQs and sets up the IDT by installing exception and interrupt vectors and handlers, as described in Sect. 14.3.

14.6.2 GDT for Segmentation

In protected mode MTX, the GDT is placed at the location GDT_ADDR = 0x9F000, which is 4KB below the ROM area at 0xA0000. The CPU's GDTR is loaded with

```
GDTR = |u16 limit=(5*8-1)|u32 GDT_ADDR=0x9F000|
```

In the GDT, only the kernel code and data segment descriptors are initialized as 4 GB segments. The TSS and LDT descriptors are initially 0. In kernel_init(), they are set to the TSS and LDT of the initial process P0. During task switch, we switch both TSS and LDT in the GDT to that of the next running process, and load the CPU's TR and LDTR registers with the modified TSS and LDT descriptors. Thus, the TSS and LDT in the GDT always refer to that of the current running process. Figure 14.11 shows the GDT and LDT contents of MTX32.1 during system operation.

The IDT is placed at 0x9F040, which is aligned to a 16-byte boundary for better CPU data cache efficiency. The choice of the GDT and IDT locations are quite arbitrary. If desired, the reader may experiment with other locations for the GDT and IDT. After initializing the system, init() calls main() in t.c. The logic of main() is

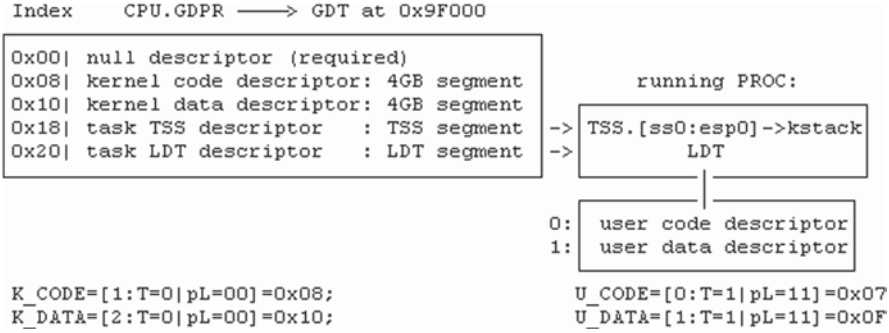


Fig. 14.11 GDT and LDT contents of MTX32.1

```
/****************** t.c file logic **********************/
 PROC proc[NPROC+NTHREAD];
 PROC *freeList,*tfreeList,*running,*readyQueue,*sleepList;
 int kernel_init()
 {
    initialize procs,freeList,tfreeList,readyQueue,sleepList;
    create P0 as the initial running process;
    set P0's TSS.[ss0:esp0] = [kernel ds: p0stack high end]
             LDT.[user code, user data] to defaults
    switch_tss(running): set TSS and LDT descriptors in GDT to
                         P0's TSS and LDT; load TR and LDTR
 }
 int main()
 {
     kfork("/bin/init"); // kfork P1 as the INIT process
     unlock();           // allow interrupts
     while(1){           // P0 loops
        if (readyQueue)
            tswitch();   // switch to a ready proc in readyQueue
        else halt();     // hlt; waiting for interrupts
     }
 }
```

14.6.3 Process TSS and LDT

In kernel_init(), it initializes the TSS structure and LDT of P0. Since P0 only runs in Kmode, it does not need a Umode image, so its LDT entries are set to defaults. Then it calls switch_tss(), which sets the TSS descriptor in the GDT to point to P0's TSS. This makes P0's kstack as the initial interrupt stack of the system. The TSS and LDT of other processes are set up in fork1() when they are created, as shown below.

```
PROC *fork1()
{
  (1). create a new proc pointed by p as in real mode;
  (2). initialize new proc's TSS structure:
       p->tss.esp0 = (u32)&p->kstack[SSIZE];    //interrupt stack
       p->tss.ss0  = KDATA;                     // 0x10
       p->tss.(es, ds, ss, fs, gs) = USER_DATA; // 0x07
       p->tss.cs   = USER_CODE;                 // 0x0F
       p->tss.ldt  = 0x20;                      // 0x20
       p->tss.iobitmap = 0;
  (3). /* allocate Umode image memory at pid*2MB */
       p->res->paddress = (p->pid)*0x200000;
  (4). set_ldt_entries(p); //set new proc's ldt[ ] entries
       return p;
}
```

The LDT entries of a new process are set up by set_ldt_entries(), as shown below.

```
int set_ldt_entries(PROC *p)
{  /* G = 1, size = 512*4K = 2MB */
  u64 ldt_code = 0x00c0fa00000001FFULL; // LDT code 2MB prototype
  u64 ldt_data = 0x00c0f200000001FFULL; // LDT data 2MB prototype
  u64 addr = p->res->paddress;          // fill in p's PA address
  ldt_code |= ((addr)<<16) & 0x000000ffffff0000ULL;
  ldt_code |= ((addr)<<32) & 0xff00000000000000ULL;
  p->ldt[0] = ldt_code;
  ldt_data |= ((addr)<<16) & 0x000000ffffff0000ULL;
  ldt_data |= ((addr)<<32) & 0xff00000000000000ULL;
  p->ldt[1] = ldt_data;
}
```

14.6.4 Switch TSS and LDT during Task Switch

During task switch, switch_tss() switches both TSS and LDT descriptors in the GDT to that of the next running process. It also loads the CPU's TR and LDTR registers with the new TSS and LDT. The code of switch_tss() is

```
int switch_tss(PROC *p)
{
    set_tss((int)&p->tss);              // GDT.tss points to p->tss
    set_ldt((int)&p->ldt);              // GDT.ldt points to p->ldt
    asm("ltrw %%ax\n\t"::"a"(TSS_SEL)); // load Task Register TR
    asm("lldt %%ax\n\t"::"a"(LDT_SEL)); // load LDT Register LDTR
}
int set_tss(u32 tss)
{
    u64 tss_entry = 0x0080890000000067ULL;  // TSS prototype
    u64 addr = (u64)tss;
    tss_entry |= (addr<<16) & 0xffffff0000; // fill in address
    tss_entry |= (addr<<32) & 0xff00000000000000ULL;
    gdt[GDT_TSS] = tss_entry;               // GDT_TSS=3
}
int set_ldt(u32 ldt)
{
    u64 ldt_entry = 0x008082000000000fULL;  // LDT prototype
    u64 addr = (u64)ldt;
    ldt_entry |= (addr<<16) & 0xffffff0000; // fill in address
    ldt_entry |= (addr<<32) & 0xff00000000000000ULL;
    gdt[GDT_LDT] = ldt_entry;               // GDT_LDT=4
}
```

14.6.5 Changes in MTX32.1 Kernel

Other changes in the MTX32.1 kernel are listed and explained below.

(1). kfork(): kfork() is only used by P0 to create the INIT process P1, which is the first process with a user mode image. kfork() first calls fork1() to create a new proc with a Umode memory area and initialize its TSS and LDT. Then it calls load() to load the image file,/bin/init, to P1's memory area and initializes P1's kstack for it to return to Umode. To do this, we pretend again that P1 had issued a system call from virtual address 0 and is about to return to Umode. In real mode, the saved syscall context is in the process user mode stack. In protected mode, it is in the process kernel mode stack. Besides memory management and exception processing, this is the only difference between real and protected modes. When a process does a syscall, it first saves the Umode CPU registers uSS, uSP,uflags, uCS,uPC in the proc's kstack by INT 0x80. Then it enters kernel to execute the syscall entry code int80h(), which saves other Umode CPU registers in kstack. In order for a new process to return to Umode, its kstack must contain such an interrupt stack frame, as shown in Fig. 14.12.

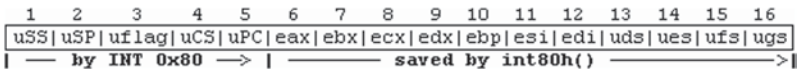


Fig. 14.12 Process kernel stack to goUmode

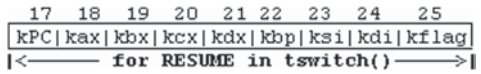


Fig. 14.13 Process kernel stack to resume

In order for the new process to resume in Kmode when it is scheduled to run, its kstack must have a resume stack frame corresponding to the RESUME part of tswitch(), as shown in Fig. 14.13.

Accordingly, we initialize the various fields in the new proc's kstack as follows.

- [uSS|uSP]=[UDS|high end of user stack area=2 MB-4]
- saved Umode registers=[0x0200|UCS|uPC=0|eax to edi=0| uds to ugs=UDS]
- resume stack frame=[goUmode| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0]
- saved ksp points to kstack[SSIZE-25]: p->ksp=&p->kstack[SSIZE-25];

When the new proc is scheduled to run, it first resumes to goUmode (in ts.s), which pops the saved Umode registers, followed by iret, causing it to return to VA=0 in Umode.

(2). fork(): fork() first calls fork1() to create a new proc with a Umode memory area. It copies the parent's Umode image to child. In addition, it also copies the parent's kstack to child's kstack so that their "saved" Umode registers are identical. Then it fixes up the child's kstack for it to return to its own Umode image. Referring to the kstack diagram in Fig. 14.12, we see that in the child's kstack only entries 1 to 16 are relevant. Therefore, we only need to copy 16 entries from the bottom of the parent's kernel stack. The copied Umode segment selectors uCS, uSS, uds, ues, ufs and ugs do not need any change since they are the same for all Umode images. The actual memory area referred to by the selectors is determined by the process LDT. In order for the child process to resume when it is scheduled to run, its kstack must have a resume stack frame. Figure 14.14 shows the kstack contents of the child process.

Therefore, we fix up the child proc's kstack as follows.

- Append a resume stack frame=[goUmode| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0] to the copied kstack.
- Let child PROC.ksp point to kstack[SSIZE-25];
- Change saved eax (at index 6) to 0 for it to return pid=0 to Umode.

For the parent process, simply change the saved eax to child pid as the return value.

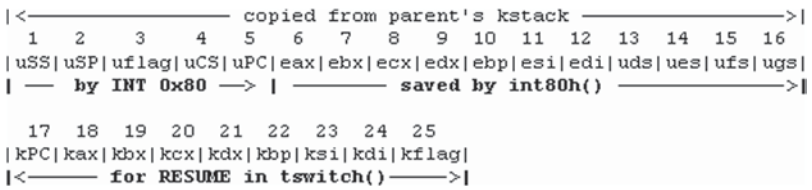


Fig. 14.14 Child process kstack in fork

(3). vfork(): vfork() is identical to fork(), except that it does not copy the parent's image, only the parent's LDT. This allows the child to share the same Umode image with the parent. In real mode, we had to create a Umode stack frame in the parent's ustack for the child to return to Umode. In protected mode, this is no longer necessary since the child process returns from its own kernel stack, which is the same as in fork(). However, in order not to interfere with the parent's ustack contents, a vforked child still needs a separate ustack area for it to return to Umode. For protection, we can mark the data segments shared by both parent and child as read-only (type = 0) until the child exec to its own segments. While sharing the same data segment, if either process tries to write to the segment, it will generate a protection error. We can modify the exception handler to recognize such a trap, split the image and restore the write access to the separated data segments. However, this is not yet implemented in MTX32.1.

(4). Threads: Threads in protected mode are the same as they are in real mode, except that all threads in the same process share the same LDT of the hosting process. The resume stack frame of each thread is created in the thread's kstack, not in the caller's ustack as in real mode.

(5). Access Umode Space from Kmode: In 16-bit real mode, the Kmode and Umode spaces of a process are in different memory segments. It must use the inter-segment copying functions to access the Umode space. In protected mode, a process in Kmode can access its Umode image directly. For example, to get syscall parameters at an offset from the top of ustack, the operation becomes a simple memory reference.

u32 parameter = *(u32 *)(running- > res- > paddress + uSP + offset);

Similarly, writing to Umode space becomes a simple memory reference also.

(6). Memory Management: When using segmentation, process images are allocated in contiguous memory. Memory management can use the same algorithm as in real mode, e.g. as variable-sized partitions.

(7). kexec(): In protected mode, kexec() is exactly the same as in real mode. The only changes are: the return stack frame is in the proc's kstack and the ustack TOP is at the virtual address 2 MB-4, which is an offset in the proc's Umode data segment.

(8). Exception Handling in Protected Mode: This is already explained in Sect. 14.5.

(9). I/O buffers: In real mode, MTX has only a few (4) block device I/O buffers due to limited space in the MTX kernel. In protected mode, this limitation no longer exists. The kernel may provide a large number of I/O buffers to improve I/O performance. In MTX32.1, we use some of the real mode memory as I/O buffers. With a large number of I/O buffers, the buffer hit ratio is constantly around or above 60 %.

(10). Device Drivers: In protected mode, all the device drivers do not need any change, except for the FD driver. This is because the FD controller, which uses channel 2 DMA of the ISA bus, can only accept 18-bit real-mode address. If all the block device I/O buffers reside in the low 1 MB real-mode memory, the FD driver does not need any change. Otherwise, we can use a 1KB area in real mode memory to transfer data between the FD and I/O buffers in high memory.

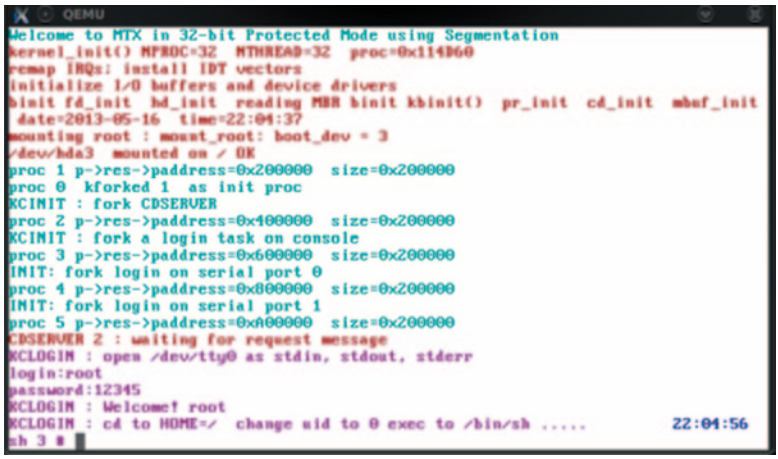


Fig. 14.15 Startup screen of MTX32.1

14.6.6 Summary on Segmentation

Segmentation is a unique feature of the Intel x86 processor architecture. Only a few real OS used segmentation. In the early days of PCs, IBM's OS2 used segmentation. Minix started in 16-bit real mode. Minix2 extended Minix to the Intel i386 architecture and used segmentation. Currently, Minix3 uses paging. We use MTX32.1 primarily as an introduction to protected mode operations. It also allows for a smooth transition from 16-bit real mode to 32-bit protected mode.

14.6.7 Demonstration System of MTX32.1

In the MTX install CD, MTX.images/mtx32.1 is a runnable image of MTX32.1. Figure 14.15 shows the startup screen of MTX32.1 running under QEMU. In the figure, each process displays its starting virtual address and image size. User interface and command executions are identical to that of RMTX in 16-bit real mode.

14.7 MTX32.2: 32-bit Protected Mode MTX using Static Paging

The second version of MTX in 32-bit protected mode is MTX32.2, which uses static paging. In MTX32.2, the user mode image of each process is allocated a piece of contiguous memory, which is aligned to a page boundary and consists of an integral number of pages. Then we set up the paging hardware in such a way that

the image is accessed as contiguous pages. This simplifies memory management since it is the same as in segmentation. It also allows for a smooth transition from segmentation to paging. The drawback is that it does not fully utilize the capability of the paging hardware since all the page frames are contiguous. Dynamic paging will be implemented later in PMTX.

14.7.1 MTX32.2 Virtual Address Spaces

In order to use paging, each process needs a page directory. So we add a u32 *pgdir field to the PROC structure. When using paging, all segments in the GDT are flat 4 GB segments, so that the linear address range is from 0 to 4 GB. In MTX32.2, we divide the 4 GB virtual address space evenly into two halves, each of size 2 GB. In kernel mode, the virtual address range is from 0 to the amount of physical memory, which is identity mapped to PA=[0, size of physical memory]. In user mode, the virtual address range is from 2 GB to 2 GB+Umode image size. Thus, the kernel space is mapped low and the user space is mapped high. The virtual address spaces are set up as follows.

(1). SETUP: During booting, SETUP's GDT defines 6 segments, as shown below.

```
setup_gdt:
 .quad  0x0000000000000000  # null descriptor

 .quad  0x00cF9A000000FFFF  # kcs PpLS=9=1001, type=A,non-conform
 .quad  0x00cF92000000FFFF  # kds PpLS=9=1001, R|W data segment
 .quad  0x0000000000000000  # tss
 .quad  0x00cFFA000000FFFF  # ucs PpLS=F=1111, type=A,non-conform
 .quad  0x00cFF2000000FFFF  # uds PpLS=F=1111, R|W data segment
```

The initial GDT no longer has an LDT. Instead, it defines 2 user mode segments. All the code and data segments are flat 4 GB segments as required by paging. The actions of SETUP are exactly the same as that in MTX32.1. It moves the GDT to 0x9F000, enters protected mode, moves the MTX kernel to 1 MB and jumps to the MTX kernel entry point, pmode_entry, which sets the stack pointer to P0's kstack and calls init() in init.c.

(2). init.c: init() first initializes the display driver to make printf() work. Then it builds page tables to map kernel's virtual address space to physical memory. At this moment, the kernel can access all the available physical memory via segmentation. Although the kernel page directory and page tables can be located anywhere in memory, we shall place them directly above the kernel image. Assume 512 MB physical memory and the MTX kernel occupies the lowest 4 MB. We shall build the kernel page directory at 4 MB and the page tables at 4 MB+4 KB, 4 MB+8 KB, etc. as shown by the following code segments.

```
#define PASIZE   0x20000000        (ASSUME: PA size = 512MB)
#define KPGDIR   0x400000          (kpgdir at 4MB)
#define PAGSIZE  4096
#define KPGTABLE (PGDIR + PGSIZE)  (pgtables are next to kpgdir)
#define ENTRIES  (PASIZE>>22)      (number of page table needed)
int i,j;
u32 *kpgdir = (u32 *)KPGDIR;   // kernel pgdir at 4MB
u32  kpt = KPGDIR+0x3; // page table addr;3=011=Kpage,W,Present
for (i=0; i<ENTRIES; i++){     // fill kpgdir with 128 entries
    kpgdir[i] = kpt;           // each pointing at a page table
    kpt += PGSIZE;
}
// create 512/4=128 page tables starting at 4MB+4KB
kpg = (u32 *)KPGTABLE;
kpte = 0x3; // starting physical address=0,3=011=Kpage,W,Present
for (i=0; i<ENTRIES; i++){
   for (int j=0; j<1024; j++){
       kpg[i*1024+j] = kpte;   // page table entry
       kpte  += PGSIZE;
   }
}
```

14.7.2 MTX32.2 Kernel Page Directory and Page Tables

The kernel page directory and 128 page tables create an identity mapping of VA=[0, 512 MB] to PA=[0, 512 MB]. Once built, kpgdir plays two roles. First, it will be the page directory of the initial process P0, which runs in Kmode whenever no other process is runnable. Second, it will be the prototype of the page directories of all other processes. Each process has its own page directory and associated page tables. Since the kernel mode address space of all the processes are the same, the first 512 entries of their page directories are identical. When creating a new process we simply copy the first 512 entries of kpgdir into the process pgdir. The high (512 to 1023) entries of a process pgdir define the user mode page tables of that process. These entries will be set up when the process is created in fork1(). After setting up the kernel kpgdr and page tables, init() loads the control register CR3 with kpgdir and turns on paging. After these, all kernel addresses are mapped to physical addresses by the paging hardware. In this case, both addresses are the same due to the identity mapping of VA to PA.

14.7.3 MTX32.2 User Mode Page Directory and Page Tables

P0 runs only in Kmode, so it does not need a Umode image. Every other process has a Umode image. To simplify memory management, we assume that the Umode image of each process has a fixed size of 4 MB, which is allocated at (pid+1)*4 MB in physical memory. The choice of 4 MB image size is because it only needs one

page table. For each process, we build a pgdir and its associated page table in the area between 6 and 7 MB, which has enough space for 128 processes. During task switch, we simply load CR3 with the pgdir of the next running process.

14.7.4 MTX32.2 Kernel Startup Sequence

After setting up kernel page tables and enabling paging, init() continues to initialize the MTX kernel by the following steps.

1. Call kernel_init() (in t.c file) to initialize kernel data structures and create the initial process P0. It sets P0's pgdir to kpgdir, GDT.tss to P0's TSS and loads CPU's TR register with GDT.tss, which makes P0's kstack as the initial interrupt stack. After these, the system is running P0.
2. Install IDT vectors and exception handlers (in traps.s and trapc.c) as described in Sect. 14.3. Install I/O interrupt and syscall vectors. These are exactly the same as in MTX32.1.
3. Initialize device drivers and file system. Mount the root device and set P0's CWD to the root directory.
4. Call main() in t.c. In main(), P0 calls kfork("/bin/init") to create the INIT process P1. kfork() calls fork1(), which is the beginning part of both kfork() and fork().

14.7.5 Process Page Directory and Page Table

To support paging, fork1() is modified to create a pgdir and page tables for a new process.

```
PROC *fork1()
{
      int i;  u32 *pgtable, pgt_entry;
(1). create a new proc p; initialize its TSS as in MTX32.1
(2). // simple memory allocation by pid
      p->res->paddress = (p->pid+1)*0x400000; // at 8MB, 12MB, etc.
(3). // allocate a pair of [pgidr, pgtable] in the area of 6MB
      p->res->pgdir = (u32 *)6MB + (p->pid-1)*2048; //6MB,6MB+8KB,etc.
      memset(p->res->pgdir, 0, 4096); // zero out pgidr
(4). // copy first 512 entries of kpgdir and create pgtable pointer
     for (i=0; i<512; i++) p->res->pgdir[i] = kpgdir[i];
     p->res->pgdir[512] = (u32)p->res->pgdir+4096+0x7; //7=111=UWP

(5). // create pgtable to map proc's VA=[0-4M] to PA at paddress
     pgtable = p->res->pgdir + 1024;
     pgt_entry = (u32)(p->res->paddress + 0x7); // PA|7=111=UWP
     for (i=0; i<1024; i++){
          pgtable[i] = pgt_entry;
          pgt_entry += 4096;
     }
     return p;
}
```

fork1() creates a new process and initialize its TSS. It constructs a pair of pgidr and pgtable for the new process in the area of 6 MB+(pid-1)*8KB. Since each process only needs 8KB for its pgdir and pgtable, the 1 MB area at 6 MB has enough space for 128 processes. The first 512 entries of every process pgdir are copied from kpgdir. The remaining pgdir entries are cleared to 0 except pgdir[512], which points to the process pgtable. Each pgtable contains 1024 entries, which map the virtual address space VA=[0x80000000, 0x80000000+4 MB] to the 4 MB PA of the process image.

14.7.6 Process Interrupt and Resume Stack Frames

After creating P1, kfork() loads the Umode image file,/etc/init, to the memory area of P1, sets up P1's kernel stack and enters it into the readyQueue. The initial kernel stack of a process consists of two sections; a RESUME stack frame for it to resume running in Kmode, preceded by an interrupt stack frame for it to return to Umode, as shown in Fig. 14.16.

The kstack frames of P1 are set up as follows.

```
memset(p->kstack, 0, 4*SSIZE);            // zero out kstack
p->kstack[SSIZE-1]  = UDS;                // uss = UDS
p->kstack[SSIZE-2]  = VA(4*1024*1024-4);  // usp = 4MB-4
p->kstack[SSIZE-3]  = UFLAG;              // uflag=0x0200
p->kstack[SSIZE-4]  = UCS;                // ucs = UCS
p->kstack[SSIZE-5]  = VA(0);              // upc = VA(0)
p->kstack[SSIZE-13] to p->kstack[SSIZE-16] = UDS; // other segs
p->kstack[SSIZE-17] = (int)goUmode;       // RESUME to goUmode()
p->ksp = (int *)&(p->kstack[SSIZE-25]);   // PROC saved ksp
```

The only thing new here is the VA macro

```
#define VA(x) ((x) + 0x80000000)
```

which converts an offset address in Umode to a virtual address. When P1 is scheduled to run, it first uses the resume stack frame to resume to goUmode, which restores saved CPU registers and iret back to VA(0) in Umode. After creating P1,

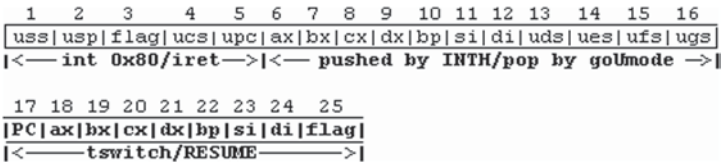


Fig. 14.16 Process interrupt and resume stack frames

P0 switches process to run P1. P1 forks one or more login processes and waits for ZOMBIE children. When the login processes start up, the system is ready for use.

14.7.7 Switch TSS and pgdir during Task Switch

During task switch, we switch TSS in the GDT and reload the CPU's TR and CR3 registers with the TSS and pgdir of the next running process. Reloading CR3 causes the CPU to flush its TLB and switch to the pgdir of the next running process.

14.7.8 Changes in MTX32.2 Kernel

(1). fork(): The only changes are in the interrupt and resume stack frames of the child process, both are in the child proc's kstack.

(2). Memory Management: When using static paging memory management is again trivial. Each process (except P0) has a Umode image of size 4 MB, which is allocated at (pid+1)*4 MB. If desired, the reader may modify this simple scheme by using different image sizes and managing process images as variable-sized partitions.

(3). loader: Both MTX32.1 and MTX32.2 use the same simple memory management scheme as in real mode, i.e. as variable sized partitions. The image loader is the same as in real mode, except that it is modified to load either flat binary or ELF executables.

(4). kexec(): When using static paging, kexec() is exactly the same as in real mode. The only modification is that the TOP of ustack is at the virtual address 4 MB. After loading the new Umode image, the command line is copied to the high end of the Umode stack, and the long word at TOP-512 points to the command line in ustack. Then the PROC's Kmode stack is re-initialized as follows.

```
kstack[SSIZE-1] = UDS;                  // uSS
kstack[SSIZE-2] = VA(TOP - 512);        // usp=0x8000000+TOP-512
kstack[SSIZE-3] = UFLAG;                // uflag
kstack[SSIZE-4] = UCS;                  // uCS
kstack[SSIZE-5] = VA(0);                // uPC=0x80000000 in Umode
kstack[SSIZE-13] to kstack[SSIZE-16] = UDS // other segments = UDS
kstack[SSIZE-17] = (int)goUmode;
running->ksp = (int *)&(running->kstack[SSIZE-25]);
```

When the process returns to Umode, it executes from VA(0) with the command line as parameter in ustack. Parsing the command line into argc and argv[ ] is done in Umode.

(5). Access Umode Space from Kmode: When a process executes in Kmode at privilege level 0, it can access all the pages in both Kmode and Umode. So a process in Kmode can access its Umode space directly.

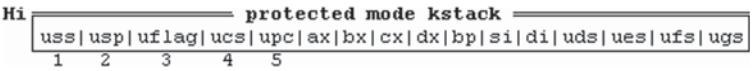


Fig. 14.17 Process kernel stack due to interrupt

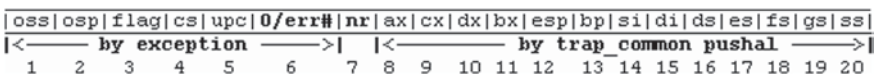


Fig. 14.18 Process kernel stack due to exception

(6). Dispatching Signal Handlers: When a process is about to return to Umode, it checks for signals and handles any pending signal. For signals other than 9, a process may have installed a user mode signal catcher. If so, the process must return to execute the catcher in Umode. In protected mode, dispatching a process to execute a signal catcher is similar to that in real mode. If the process entered kernel via an interrupt or syscall, the process kstack must contain a regular interrupt/syscall stack frame, as shown in Fig. 14.17.

If the process entered kernel due to an exception, its kstack contains an exception stack frame, as shown in Fig. 14.18.

In both cases, dispatching a process to execute a user mode signal catcher is the same:

- save upc (at index 5) in kstack as oldPC
- replace upc in kstack by newPC, which points to catcher()in Umode
- create 2 slots in ustack to contain oldPC, sig#
- decrement usp (osp at index 2) by 2 slots for catcher(int sig#)
- when catcher() finishes, it returns by oldPC to where it lastly entered kernel.

(7). Device Drivers: In 32-bit protected mode, all device drivers remain the same, except for the FD driver, which is modified as follows. In MTX32.2, the data areas of I/O buffers are allocated in the memory area of 7 MB. Since the FD's DMA controller can only accept 18-bit real-mode address, we use a fixed 1KB area in real-mode memory at 0x90000 for data transfer between the FD driver and I/O buffers.

(8). User Mode Programs: In MTX32.2, user mode images begin from the virtual address 0x80000000 (2 GB). To comply with this, the initial virtual address of user mode images is set to 0x80000000 in the Makefile.

14.7.9 Demonstration System of MTX32.2.

In the MTX install CD, MTX.images/mtx32.2 is a runnable image of MTX32.2. Figure 14.19 shows the startup screen of MTX32.2 running under QEMU. In the figure, each process displays its starting virtual address. In addition to the console, the system also supports two login processes on serial ports, which are not shown.

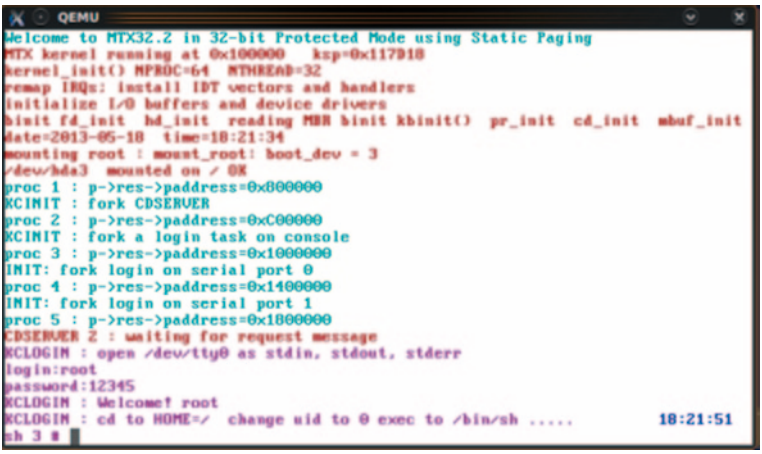


Fig. 14.19 Startup screen of MTX32.2

14.8 PMTX: 32-bit Protected Mode MTX Using Dynamic Paging

The third version of MTX in 32-bit protected mode is PMTX, which uses dynamic paging. It is the final version of MTX in 32-bit protected mode for uniprocessor systems.

14.8.1 PMTX Virtual Address Spaces

In PMTX, the 4 GB virtual address space is also divided into two equal halves, except that the kernel mode virtual address space is from 2 to 4 GB and user mode virtual address space is from 0 to Umode image size. Thus, the user space is mapped low and the kernel space is mapped high. This organization conforms to most other Unix-like systems, such as Linux and xv6. In Linux (Bovet and Cesati 2005), user space is from 0 to 3 GB and kernel space is from 3 to 4 GB. In xv6 (Cox et al. 2011), user space is from 0 to 2 GB and kernel space is from 2 to 4 GB, which is the same as in PMTX. The PMTX kernel is compiled with the starting virtual address 0x80100000 but it runs at the physical address 0x100000 (1 MB). This can no longer be achieved by segmentation. We must use paging when the kernel execution begins. Instead of setting up all the page tables in one step, the virtual address mapping is accomplished in two steps, which resemble that of multi-stage booting. The following describes the steps in detail.

14.8.2 PMTX Kernel Startup Sequence

SETUP: During booting, the booter loads SETUP to 0x90200 and the PMTX kernel to 0x10000. Then it jumps to 0x90200 to run SETUP. The initial GDT in SETUP is

```
 setup_gdt:
    .quad  0x0000000000000000  # null descriptor
    .quad  0x00cF9A000000FFFF  # kcs  PpLS=1001, type=non-conforming
    .quad  0x00cF92000000FFFF  # kds  PpLS=1001, R|W data segment
```

It defines only two 4 GB kernel code and data segments. SETUP moves the initial GDT to 0x9F000 and loads the GDTR register to point at the initial GDT. Then it enters protected mode, moves the MTX kernel to 1 MB and ljmp to the entry address of the MTX kernel at 1 MB. Unlike the previous versions of MTX in protected mode, the initial GDT is only temporary. It provides the initial 4 GB flat segments for the PMTX kernel to get started.

entry.s: pm_entry is the entry point of the PMTX kernel. In order to let the kernel use virtual addresses from 0x80000000 by paging, entry.s defines an initial page directory, ipgdir, two initial page tables, pg0 and pg1, a new GDT, an IDT and a page directory, kpgdir, at offsets from 0x1000 to 0x8000, as shown below.

```
#--------------------- entry.s file --------------------------------
pm_entry: # entry.s beginning code in first 4KB of kernel
      .org 0x1000              # at offset 4KB
ipgdir:
      .long 0x00102007         # point to pg0 at 0x102000
      .long 0x00103007         # point to pg1 at 0x103000
      .fill 510,4,0            # 510 0 entries
      .long 0x00102007         # point at pg0 at 0x102000
      .long 0x00103007         # point at pg1 at 0x103000
      .fill 510,4,0            # 510 0 entries
.org 0x2000                    # at offset 8KB
pg0:       # 2 initial page tables pg0, pg1: to be set in entry.s
.org 0x3000
pg1:
.org 0x4000
kgdt:             # GDT
     .quad 0x0000000000000000 # null descriptor
     .quad 0x00cF9A000000FFFF # kcs  PpLS=9=1001,type=non-conform code
     .quad 0x00cF92000000FFFF # kds  PpLS=9=1001,type=R|W data segment
     .quad 0x0000000000000000 # tss
     .quad 0x00cFFA000000FFFF # ucs  PpLS=F=1111,type=non-conform code
     .quad 0x00cFF2000000FFFF # uds  PpLS=F=1111,type=R|W data segment
kgdt_desc: .word.-kgdt-1
           .long  kgdt
.org 0x5000
idt:  .fill 1024,4,0           # 2KB IDT table at offset 0x5000
.org 0x6000
kpgdir: .fill 1024,4,0         # final kernel mode kpgdir at 0x6000
.org 0x8000                    # Other PMTX kernel code begins here
```

14.8.3 PMTX Kernel Virtual Address Mapping

In the initial page directory, ipgdir, entries 0 and 1 point to the two initial page tables pg0 and pg1. The two initial page tables are filled with page frames from 0 to 8 MB to create an identity mapping of the lowest 8 MB of physical memory. Entries 512 and 513 of ipgdir also point to pg0 and pg1, which map the virtual address range VA=[2 GB to 2 GB+8 MB] to [0 to 8 MB]. The new GDT, kgdt, defines 6 segments, in which tss, ucs and uds are the TSS, user mode code and data segments of the current running process. The actions of entry.s are as follows.

(1). Set up initial page tables pg0, pg1; map VA=[0,8 MB]-[2G,2G+8 MB] to [0,8 MB]:

```
        movl  $pg0   -KVA, %ebx     # KVA = 0x80000000
        movl  $0x07,    %eax     # page R|W and Present
        movl  $2*1024,  %ecx     # loop 2048 times
loop0:  movl  %eax,   0(%ebx)    # start with pageframe=0
        addl  $4,       %ebx
        addl  $4096,    %eax     # next pageframe
        loop  loop0              # loop 2048 times
```

(2). Load CR3 with physical address of ipgdir and turn on paging.

```
movl  $ipgdir  -KVA,%eax   # physical address of ipgdir
movl  %eax, %cr3         # load CR3 with PA(ipgdir)
movl  %cr0, %eax         # enable paging
orl   $0x80000000,%eax
movl  %eax, %cr0
```

(3). Do a jmp to flush the instruction pipeline.

```
    jmp 1f
1:
```

(4). Do another jmp to force the CPU to use virtual address.

```
    movl  $2f,%eax
    jmp  *%eax
2:
```

The second jmp at step (4) is tricky but essential. Before the second jmp, the CPU was executing with real address in the range 0–8 MB. The second jmp uses PC relative addressing, which forces the CPU to switch to virtual addresses in the range of 0x80000000 to 0x80000000+8 MB.

(5). Load GDTR with the new kgdt descriptor.

```
lgdt  kgdt_desc         #load GDT at 0x8010400
```

The new GDT defines 2 kernel mode segments, a TSS and 2 user mode segments. All code and data segments are flat 4 GB segments as required by paging.

(6). initproc is a statically defined PROC structure for the initial process P0. entry.s sets the stack pointer to the high end of initproc's kstack. Then it calls init() to initialize the PMTX kernel.

(7). init.c file: init() first initializes the display driver to make printf() work. At this moment, the kernel's virtual address space is limited to 8 MB. The next step is to expand the virtual address range to the entire available physical memory. Assume 512 MB physical memory and the PMTX kernel occupies the lowest 4 MB. We shall build the new page directory at 0x8016000 and the page tables at 4 MB. This is done by the kpgtable() function shown below.

14.8.3.1 PMTX Kernel Page Directory and Page Tables

```
#define KPG_DIR   0x80106000  // at offset 0x6000 in kernel
#define KPG_TABLE 0x80400000  // begin from 4MB
#define PGSIZE    4096
void kpgtable(void)
{  u32 i, j, NPGTABLES = 128; // 512MB PA needs 512/4=128 pgtabels
   u32 *pgdir  = (u32 *)KPG_DIR;
   u32 *ptable = (u32 *)KPG_TABLE;
   u32 pte = (u32)(PA(KPG_TABLE)|0x3; // begin PA=0,0x3=|Kpage|W|P
   memset(pgdir, 0, 4096);            // zero out kpgdir
   for (i=512; i<512+NPGTABLE; i++){  // from kpgdir[512]
       pgdir[i] = pte;                // pointing at pgtables.
       pte += PGSIZE;
   }
   pte = (u32)0 | 0x03;               // start with PA=0
   ptable  = (u32 *)KPG_TABLE;        // KPG_TABLE at 4MB
   for (i=0; i<NPGTABLES; i++){       // fill 128 pgtables
      for (j=0; j<1024; j++){         // pgtabe, 4KB each
          ptable[i*1024 + j] = pte;
          pte += PGSIZE;
      }
   }
}
```

(7).1. kpgtable() creates a kpgdir at 0x80160000, in which entries 512 to 639 point to 128 page tables, which map VA=[0x80000000, 0x80000000+512 MB] to [0, 512 MB]. As in MTX32.2, the kernel page directory, kpgdir, also plays two roles. First, it will be the pgdir of the initial process P0, which runs in Kmode whenever no other process is runnable. Second, it will be the prototype of the page directories of all other processes. Each process has its own page directory and page tables. Since

the kernel mode address spaces of all processes are the same, the high 512 entries of all page directories are identical. When creating a new process we simply copy the high 512 entries of kpgdir to the process pgdir. The low (0 to 511) entries of a process pgdir define the user mode page tables of the process. These entries will be set up when the process is created.

(7).2. Switch CR3 to the new kernel page directory, kpgdir. This allows the kernel to access all the physical memory from 0 to 512 MB. Figure 14.20 shows the memory map of PMTX.

(7).3. kernel_init(): After setting up the kernel mode kpgdir and page tables, init() calls kernel_init() (in t.c file) to initialize the kernel data structures, such as free PROC lists, ready queue and sleepList, etc. In PMTX, only the initproc and its resource structure are statically defined. The other NPROC (1024) and NTHREAD (512) PROCs are constructed in the memory area of 5 MB. In kernel_init(), it uses initproc to create the initial process P0, which uses kpgdir as the page directory. It sets the privilege level-0 stack in P0's TSS to P0's kstack. Then it calls switch_tss(), which changes the TSS in GDT to P0's TSS and loads the CPU's task state register, TR, with the new TSS. These make P0's kstack as the interrupt stack. The system is now running the initial process P0.

(7).4. Remap IRQ vectors. Set up IDT and install exception/interrupt handlers, as described in Sect. 14.5. The IDT only needs 2KB space. It is constructed at 0x105000. init() proceeds to initialize the IDT and install exception and I/O interrupt vectors. Exception handler entry points are in traps.s. Exception handlers are in trapc.c. I/O interrupt entry points are defined in ts.s by calls to the INTH macro. I/O interrupt handler functions are in the various device drivers. Among the interrupts, vector 0x80 is for system calls.

(7).5. Initialize I/O buffers, device drivers and timer: After setting up the IDT and interrupt vectors, init() initializes I/O buffers. The PMTX kernel has 1024 I/O buffers, which are allocated at 7 MB. When initializing the HD driver P0 also reads the partition table to determine the start sector and size of other partitions, which are used to construct the block device table. Then it initializes the file system and mounts the MTX partition as the root file system.

Fig. 14.20 Memory map of PMTX

0 1MB	1MB real mode memory PMTX kernel in 1MB to 4MB 0x104000:GDT; 0x105000:IDT 0x106000:kpgdir
4MB	128 kernel mode ptables of P0
5MB	1024 process + 512 thread PROCs
6MB	unused, for expansion
7MB	data area of 1024 I/O buffers
8-512MB	free memory for process images

14.8.3.2 Manage Page Frames

PMTX uses a free page list, pfreeList, for allocation/deallocation of page frames. The pfreeList is constructed by the code segment shown below.

```
u32 *free_page_list(u32 startva, u32 endva)
{  u32 *p = (u32 *)startva;
   while(p < (endva-4096)){
     *p = (u32)(p + 1024);
      p += 1024;
   }
   *p = 0;
   return (u32 *)startva;
}
u32 *pfreelist = free_page_list(8MB, 512MB); // build free page list
```

The pfreeList threads all the free page frames from 8 to 512 MB in a link list. Each entry of pfreeList contains the address of the next page frame. As usual, the list ends with a 0 pointer. In order for the kernel to access the entries of pfreeList, the link pointers must use virtual addresses of the page frames. When allocating a page frame the virtual address of the page frame must be converted to physical address. Conversion between virtual address and physical address are by the PA/VA macros.

```
#define PA(x)     ((u32)(x) - 0x80000000)
#define VA(x)     (      (x) + 0x80000000)
```

With the pfreelist data structure, palloc() allocates a page frame and pdealloc(VA(page frame)) inserts a deallocated page frame to pfreeList. The simplest way to deallocate a page frame is to insert it to the front of pfreeList. Alternatively, we may also insert it to the end of pfreeList to ensure that all page frames are used effectively.

(7).6. Finally, init() calls main() in t.c. In main(), P0 calls kfork("/bin/init") to create the INIT process P1 and load/bin/init as its Umode image. Then P0 switches process to run P1. P1 forks one or more login processes and waits for ZOMBIE children. When the login processes start, the system is ready for use.

14.8.4 Changes in PMTX Kernel

(1). PROC kstack: In PMTX, the kstack of each PROC is a u32 pointer. The kstack is allocated a 4KB page dynamically only when it is used to create a process. The combined size of the PROC and PRES structures is less then 512 bytes. This allows us to define a large number of NPROC (1024) and NTHREAD (512) structures, which are constructed at 5 MB. The kstack of a ZOMBIE proc is deallocated by the parent in kwait().

(2). fork1(): fork1() is the beginning part of all other fork functions. It creates a new proc and allocates a kstack for it. When called from kfork() or fork(), it also allocates a pgdir for the new proc. When called from vfork(), it lets the new proc share the same pgdir of the parent. The page tables of a process are created by a separate makePage() function.

```
PROC *fork1(int HOW) // HOW = FORK or VFORK
{ PROC *p;
  if (!(p = get_proc(&freeList)))
     return 0;
  if (!(p->res->kstack = palloc())){
      free p; return 0
  }
  if (HOW==FORK)
     if (!(p->res->pgdir = palloc())){
         free p's kstack,
         free p; return 0
     }
  else               // HOW = VFORK, share pgdir of parent
     p->res->pgdir = running->res->pgdir;
  p->res->size = running->res->size; // same image size as parent
  initialize p's kstack, tss, etc.
  return p;
}

int makePage(PROC *p, int HOW) // HOW=FORK or EXEC
{
  u32 i, j, pte, *pgtable;
  u32 *pgdir = (HOW==FORK)?p->res->pgdir : p->res->new_pgdir
  u32 npgdir = p->res->size/(1024*BSIZE); // number of pgdir entries
  u32 npages = npgdir/1024;   // number of pages needed
  u32 rpages = npages % 1024; // number of pages in last pdgir
  for (i=0; i<1024; i++)      // copy kpgdir to pgdir
      pgdir[i] = kpgdir[i];
  for (i=0; i<npgdir; i++){   // for each npgdir entry
      pgtable = palloc(); // allocate a pgtable; return 0 if fails
      memset(pgtable, 0, PGSIZE);      // zero out pgtable
      pgdir[i] = PA((u32)pgtable) + 7; // record PA+7 in pgdir[i]
      for (j=0; j<1024; j++){ // allocate page frames for page table
          pte = palloc();     // allocate a pte; return 0 if fails
          pgtable[j] = PA(pte+7);    // 7 for USER R|W pages
      }
  }
  if (rpages){    // need one more pgdir entry for rpages
     pgtable = (u32 *)palloc(); // allocate a pgtable;
     p->res->pgdir[i] = PA((u32)pgtable) + 7);
     memset(pgtable, 0, PGSIZE) // zero out pgtable entries
     for (j=0; j<rpages; j++){  // fill pgtable with rpages frames
         pte = palloc();       // allocate a pte; return 0 if fails
         pgtable[j] = PA(pte+7);
     }
  }
  return 1; // for SUCCESS
}
```

(2). kfork(): kfork() is only used by P0 to create the INIT process P1. Corresponding to the modified fork1() and makePage(), the algorithm of kfork() is

```
int kfork(char *filename)
{
    PROC *p = fork1(FORK);   // create a new proc p with a pgdir
    p->res->size = INITSIZE; // INITSIZE=4MB in type.h
    makePage(p, FORK);       // create page tables for P1
    load(filename, p);       // load /etc/init as P1's Umode image
    set up p's kstack for it to return to VA=0 in Umode
    enter p into readyQueue;
}
```

The image size of P1 is defined as INITSIZE = 4 MB mainly for convenience since it only needs one page table. After creating a new proc by fork1(), P0 allocates page frames for P1 and loads the image file. Then it creates an interrupt stack frame, followed by a RESUME stack frame, for P1 to return to VA = 0 in Umode. The stack frames of a new process are set up as follows.

```
/*********  kstack of new proc contains: *********************
|uss|usp|flag|ucs|upc|ax|bx|cx|dx|bp|si|di|uds|ues|ufs|ugs|//INT frame
  1   2   3    4   5  6  7  8  9  10 11 12 13  14  15  16
|PC|ax|bx|cx|dx|bp|si|di|flag|                          // RESUME frame
 17 18 19 20 21 22 23 24  25
************************************************************/
  for (int i=1; i<26; i++)             // clear "saved registers" to 0
       p->kstack[SSIZE-i] = 0;         // p points at new PROC
  // fill in the needed entries in p's kstack
  p->kstack[SSIZE-1] = UDS;            // uSS
  p->kstack[SSIZE-2] = p->res->size-4; // uSP = imageSize-4
  p->kstack[SSIZE-3] = UFLAG;          // uflag
  p->kstack[SSIZE-4] = UCS;            // UCS
  p->kstack[SSIZE-5] = 0;              // eip = VA 0
  p->kstack[SSIZE-13] to p->kstack[SSIZE-16] = UDS; // other segments
  p->kstack[SSIZE-17] = (int)goUmode;  // resume point in Kmode
  p->ksp = (int *)&(p->kstack[SSIZE-25]); // saved ksp in PROC
```

The interrupt stack frame of a new process is logically the same as that in real mode. The only difference is that it is now in the proc's kstack, rather than in the user mode stack.

(3). Loader: The loader is used in both kfork() and kexec() to load an image file. Since the page frames may not be contiguous, the loader is modified accordingly. Instead of loading the image file to a linear address, it loads 4KB blocks of the file to the page frames of a process. Since the modifications are simple, they are not shown here.

(4). fork(): with dynamic paging, the algorithm of fork() is modified as follows.

```
int fork()
{
  PROC *child;
  (1). if (child=fork1(FORK) fails) return -1;
  (2). if (makePage(child, FORK)==0)
          {free child's pages, pgdir, kstack and PROC; return -1}
  (3). copyImage(running, child);  // copy Umode image to child
  (4). copy parent kstack to child kstack; // only the bottom part
  (5). fix up child's kstack for return 0 to the same VA as parent;
  (6). enter child into readyQueue;
  (7). return child pid;
}

void copyImage(PROC *parent, PROC *child)
{ int i, j;
  int npgdirs = (parent->res->size + 4MB-1)/4MB; // # pgdir entries
  u32 *ppgtable, *cpgtable, *ppa, *cpa;
  for (i=0; i<npgdirs; i++){
    pgtable = VA(parent->res->pgdir[i]&0xFFFFF000); //VA(frame addr)
    cpgtable= VA( child->res->pgdir[i]&0xFFFFF000); //VA(frame addr)
    for (j=0; j<1024; j++){
       ppa = VA(ppgtable[j]&0xFFFFF000); // VA(20-bit frame addr)
       cpa = VA(cpgtable[j]&0xFFFFF000); // VA(20-bit frame addr)
       memcpy(cpa, ppa, PGSIZE);
    }
  }
}
```

After copying the Umode image, fork() copies the parent's kstack to that of the child. Then it fixes up the child's kstack frame for it to return to Umode. Similar to fork() in real mode, we do not care about the execution history of the parent in Kmode. All we need is to ensure that when the child runs, it returns a 0 to the same VA in Umode as the parent. Therefore, we only need to manipulate the bottom part of the child's kstack. The idea is exactly the same as in kfork(), i.e. to create a stack frame for the child to resume running in Kmode, preceded by an interrupt stack frame for it to return to Umode. The needed stack frames and operations are shown below.

```
|uss|usp|flag|ucs|upc|ax|bx|cx|dx|bp|si|di|uds|ues|ufs|ugs| //INT frame
  1   2   3    4   5  6  7  8  9  10 11 12 13  14  15  16
|PC|ax|bx|cx|dx|bp|si|di|flag|                          // RESUME frame
 17 18 19 20 21 22 23 24  25
```

```
  child->kstack[SSIZE-6] = 0;     // return value=saved AX = 0
  for (i=18; i<26; i++)
      child->kstack[SSIZE-i] = 0; // Kmode saved regs do not matter
  child->kstack[SSIZE-17] = (int)goUmode; // goUmode directly
  child->ksp = (int *)&child->kstack[SSIZE-25]; // saved ksp
  enqueue(&readyQueue, child);
```

When the child process starts to run, it first resumes to goUmode(), which restores the saved registers (ugs to ax), followed by iret, causing it to return to the same (ucs, upc) as the parent but in its own Umode image.

(5). vfork(): In PMTX, vfork() is implemented as follows.

```
int vfork()
{
(1). PROC *child = fork1(VFORK); // create child share same pgdir
(2). child->vforked = 1;         // mark child as VFROKED
(3). copy parent's kstack to child kstack
(4). fix up child's interrupt and resume stack frames as in fork();
(5). create a separate ustack area for child in parent's ustack
(6). child->kstack[SSIZE-2] = child usp in (5); // child saved usp
(7). change child saved AX to 0;                // child return 0
(8). enter child into readyQueue;
(9). return child pid;
}
```

When a vforked child returns to Umode, it should use its own separate ustack area. However, it seems that GCC generated code always restores user mode esp from a fixed save area, causing both parent and child return to the same ustack frame. If either process makes a syscall, it would change the ustack contents, causing the other one to crash when it returns to Umode. To prevent this, we have to implement the user mode vfork() syscall in assembly directly. The x86 CPU's paging hardware supports Copy-On-Write (COW) pages, which can be used to implement vfork. Using COW pages, vfork can create a child to share the same Umode page frames with the parent. If either process tries to write to a shared COW page, the paging hardware will generate a page fault. The page fault handler can split the pages into separate images. As of now, PMTX does not yet implement this kind of vfork. It is left as an exercise.

(6). kexec(): In general, when a process changes image, it should determine the new image size by the file's loading size. Currently, all executable programs in PMTX are very small. If we use file size as image size, all process images would be very small also. For this reason, we choose to support different image sizes as follows. When PMTX starts, the INIT process image size is set to 4 MB. Under normal conditions, all processes would run with the same image size of 4 MB. To run programs with different image sizes, enter the command with an optional SIZE parameter.

command_line [–m SIZEm]

where SIZE is the new image size in MB. Accordingly, we modify kexec() with an additional size parameter, which is 0 if no SIZE is specified.

```
int kexec(char *command_line, int size) // size = 0 or SIZE
{
 (1). if (caller is not a PROCESS OR has other active THREADs)
         return -1;
 (2). get command_line from original image;
 (3). new_size = (size)? original image size : SIZE
 (4). save pgdir and size of original image;
      if (!(new_pgdir=palloc()))// allocate a new_pgdir
          return -1;
      if (!(makeImage(running, EXEC))){ // if can't alloc new image
         deallocate new image;
         return -1;
      }
      load CR3 with new_pgdir;        // switch pgdir to new image
 (5). if (load(cmd, running) < 0){    // if load image file fails
         load CR3 with origian pgdir; // restore original image
         deallocate new image;
         return -1;                 // return -1 to original image
      }
 (6). if (caller is NOT vforked)
         deallocate original image;
 (7). if (caller is vforked)
         turn off PROC's vforked flag;
 (8). set up kstack for return to VA=0 in Umode.
}
```

In kexec(), it sets mew_size to the original image size or SIZE, if specified. It saves the original image and allocates a pgdir and page tables for the new image. If the allocation fails due to out of memory, it releases the new image, switches back to the original image and returns −1. If the allocation succeeds, it tries to load the image file to the new image. If the loading fails, it restores the original image, releases the new image and returns −1. If the loading succeeds, it releases the original image if the caller is not vforked. Then it fills the ustack with command line parameters and turns off the process vforked flag. Finally, it re-initializes the kernel stack for it to return to VA=0 in Umode, as shown below.

```
/*********  kexec(): kstack of proc contains: *************
|uss|usp|flag|ucs|upc|ax|bx|cx|dx|bp|si|di|uds|ues|ufs|ugs| //INT fra
  1   2   3    4   5  6  7  8  9  10 11 12 13  14  15  16
|goU|PC|ax|bx|cx|dx|bp|si|di|flag|                 // RESUME frame
 17 18 19 20 21 22 23 24 25  26
 ********************************************************/

  kstack[SSIZE-1] = UDS;
  kstack[SSIZE-2] = TOP-512;     // TOP=image size
  kstack[SSIZE-3] = UFLAG;
  kstack[SSIZE-4] = UCS;
  kstack[SSIZE-5] = 0x0;         // virtual address 0 in UCS
  kstack[SSIZE-13] to kstack[SSIZE-16] = UDS;
  kstack[SSIZE-17] = (int)goUmode;
  running->ksp = &(running->kstack[SSIZE-25]);
```

In addition, kexec() also clears the process signal and resets all the signal handlers to default, but it does not close opened file descriptors. Thus, opened files remain open in the new image.

(6). kexit(): When a process terminates, it releases the Umode image by freeImage() unless it's vforked. freeImage(pdgir) traverses the process pgdir to find the Umode page tables, from which to find the page frames. Then it deallocates the Umode page frames and page tables, but keeps both the pgdir and kstack since the process is still executing. The pgdir and kstack of a ZOMBIE process will be deallocated by the parent in kwait(). Note that in kexec() a vforked process always creates a new image, thereby detaching itself from the parent image. Then it tries to load the image file. If the loading succeeds, its vforked flag is turned off, in which case it will release its own image in kexit(). If the loading fails, it releases the new image and returns −1 to the original image, in which case it does not deallocate the shared image since its vforked flag is still on. Alternative schemes will be discussed in the Problem section.

(7). Expand User Mode Heap Size: In PMTX, the heap area of a process is above its Umode image. The sbrk() system call expands the heap size by adding a new page to the process image and returns the virtual address of the newly allocated page frame. In the sample PMTX, the sbrk command demonstrates the sbrk() system call. Assume that a process image has n pgdir entries. On the first call to sbrk(), it allocates a page directory entry, pgdir[n], and allocates a page frame for the page table. Each subsequent call to sbrk() allocates an additional page in the page table. After calling sbrk(), the process may access the expanded heap area for read/write. The reader may modify the sbrk syscall to reduce the heap size by deallocating heap pages.

(8). Device Drivers: When using paging in protected mode, only the FD driver needs a slight modification, as described in Sect. 14.7.8. All other device drivers remain the same as they are in real mode.

14.8.5 Page Faults and Demand-Paging

The current implementation of PMTX does not support demand-paging, mainly because we want to keep the system simple. In the current PMTX, all Umode programs need only a few pages to run, which also makes demand-paging unnecessary. Despite these simplifications, we can demonstrate the principle of demand-paging by simulating page faults and handling the page faults. In PMTX, a process in user mode may issue the system call page_out(n) to mark the nth ($n < 1024$) entry of its page table as not present. When the process attempts to access the missing page, it generates a page fault even though the page frame is already loaded with the image's code or data. When the process traps to Kmode, the page fault handler can determine whether the page fault is due to a missing page or an invalid page reference. If it is due to a missing page, the page fault handler simply marks the missing page as present and let the process continue. If it is due to an invalid page reference, it sends the process a SEGMENTATION FAULT (11) signal as usual. Assuming 4 MB image size, the following shows the algorithm of the page fault handler.

```
   void ehandler(u32 signal, ... u32 err_nr, ..) // err_nr=error number
   {
     u32 cr2, cr3, pgentry, *pgtable;
     if (exception occurred in Umode){  // i.e. running->inkmode==1
        if (err_nr==14){                // page fault
           cr2 = CPU.CR2 register;      // get offending VA from CR2
           pgentry = cr2/4096;   // convert VA to pgtable entry number
           if (pgentry < 1024){  // if < 1024, it's a MISSING page
              pgtable = VA(running->res->pgdir[0]&0xFFFFF000);
              pgtable[pgentry] |= 0x1; // mark ptable entry PRESENT
              load CPU.CR3 with PA(running->res->pgidr); // flush TLB
              return;                  // return to Umode to continue
           }
          // page fault due to invalid address: send sig#11 to process
        }
   }
   else{ // page fault occurred in Kmode: PANIC and stop }
}
```

In the above pseudo code, the page fault handler gets the virtual address (in CR2) that caused the page fault and converts it to a pgtable entry index. If the index is within the Umode VA, it means a missing page. The page fault handler marks the missing page entry as present and reloads the CR3 register with the process pdgir to flush the CPU's TLB cache. When the process returns to Umode, it re-executes the instruction that caused the page fault earlier. Since the page table entry is now present, it will not cause any page fault again. If the page table entry index is outside of the Umode VA range, it must be an invalid address, in which case the process gets a regular page fault signal. In general, a missing page may be within the Umode VA range but does not yet exist. In that case, the page fault handler must allocate a page frame, load the needed page contents into the page frame and mark the page entry present before letting the process continue. This is the basis of virtual memory based on demand-paging. It is left as a programming project.

14.8.6 Page Replacement

In the page fault handler, if a new page must be allocated but there are no free page frames, some existing pages must be evicted to make room for the needed page frame. This is known as page replacement problem in virtual memory, which has been discussed extensively in many OS books. The reader may consult such books for more information. Since PMTX does not support demand-paging, the problem of page replacement also does not exist. Nevertheless, it is still worth discussing the principles of page replacement.

(1). Local Vs. Global: When trying to evict some existing pages, the first question is where to look for such pages? The search is local if we only try to evict pages of the current process. It is global if we may evict pages of other processes in the system. In the latter case, the candidates are usually non-runnable, i.e. blocked or sleeping, processes.

(2). Least-Recently-Used (LRU) Page Replacement: The scheme is to replace a page that has not been referenced for the longest time. To support LRU, we may maintain all the pages in a link list with most recently used pages in front and least recently used pages at the end. During page replacement, choose pages from the end of the link list. Since the link list must be reordered on every memory reference, implementation of LRU by the link list method is feasible but very expensive. An alternative scheme is to assume a large global counter which is updated on every memory reference. Whenever a page is accessed, copy the current counter value as its access time. During page replacement, choose a page with the least access time to replace. However, the scheme is impractical since no such hardware support exists.

(2). Not-Recently-Used (NRU) Page Replacement: Each page frame has an A (access) bit and D (dirty) bit, which can be used to implement a simple NRU page replacement algorithm. In this scheme, the kernel periodically clears these bits (by timer interrupts). During page replacement, use the (A, D) bits to rank the pages and replace a page with the highest rank order.

Rank	(A,D)	Choice
3	(0,0):	Best candidate since page is not accessed and not modified
2	(0,1):	This may happen if (A,D) are not cleared at the same time
1	(1,0):	Third best, the page will probably be accessed again soon
0	(1,1):	Worst candidate, replace this page must write back first

14.8.7 Paging and I/O Buffer Management.

In many OS which support demand-paging, I/O buffers for files blocks may be treated as pages. When a process needs a file block, it maps the file block to a page in its virtual address space. The page is initially marked as not present. When the process attempts to access such a file block, it generates a page fault. The paging subsystem can allocate a page frame for the missing page, page-in the file block data, if necessary, and let the process continue. Once a file block is in the paged buffer cache, it can be reused to avoid physical I/O. This scheme unifies file I/O with demand-paging. It is used in Linux, and also in many other Unix-like systems. The main advantage of using a paged buffer cache is that it maximizes the use of physical memory. Almost all spare physical memory can be used for file system caching. The disadvantage is that, under intensive file I/O demands, it may cause thrashing (constantly swap pages in/out memory), which seriously degrades the system performance. For this reason, most OS sets a size limit on the paged buffer cache in order to prevent thrashing. In addition, they also use a separate internal buffer

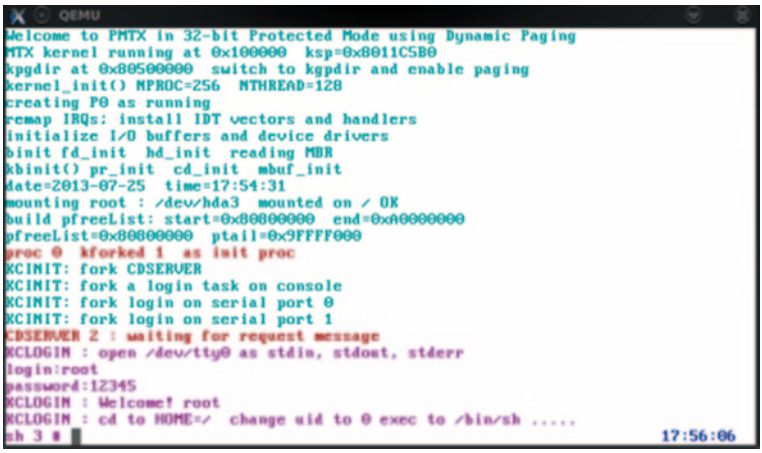


Fig. 14.21 Startup screen of PMTX

cache to maintain important system information, such as the superblock, bitmaps and directories, etc. Since PMTX does not yet support demand-paging, it only uses dedicated I/O buffers for block devices.

14.8.8 PMTX Demonstration System

In the MTX install CD, MTX.images/pmtx is a runnable image of PMTX. Figure 14.21 shows the startup screen of PMTX running under QEMU.

All executable user commands are in the /bin directory. In addition to regular commands, the user may run the following commands to test exception handling in PMTX:

Control_C key: kill process running on console or serial terminal
divide: handle divide error with or without a catcher
itimer: set interval timer and handle SIGALRM(14) signal
kill: kill process by pid, e.g. from console, kill a process running on serial terminal
pagefault: simulate page faults and page fault handling
segfault: handle segmentation fault with or without a catcher
signal: install signal catcher to IGNore or handle signal in user mode

14.9 Extensions of 32-bit Protected Mode MTX

The main advantage of 32-bit protected mode is that it opens the door for many possible extensions to the PMTX system. The following is a list of such areas.

1. EXT4 file system: Extend the file system to EXT3/EXT4 with 4KB block size.
2. Program development environment: port text editors and GCC to PMTX.
3. Dynamic linking: Support dynamic linking by shared and dynamic libraries.
4. Unix compatible sh: port or develop a Unix compatible sh.
5. Device drivers for PCI bus: Support SATA drives and USB devices.
6. Networking: Develop network drivers and port TCP/IP to support networking.
7. Other Unix software: port Unix utilities and X-Windows, etc. to PMTX.

These extensions would greatly enhance the capability of PMTX, making it a more useful system. However, before attempting such extensions, we should point out the following. So far, MTX has been a one-man's endeavor. Trying to do all the extensions is simply beyond the ability and time of any single person. More importantly, we should not lose focus on the original goal of MTX, which is intended as an educational system and should remain so. There is no need to create another PC based OS to compete with Linux (Bovet and Cesati 2005) or FreeBSD (McKusick and Neville-Neil 2004; McKusick and Bostic 1996). While some extensions are definitely needed and worthwhile, overdoing it may be counterproductive. I do not wish to see MTX end up in an awkward situation in which it is too complex for learning yet too primitive for practical use. Given the choice, I would prefer to see it remains simple and useful as an educational system.

14.9.1 64-bit Operating Systems

x86-64 refers to both AMD [AMD64 2011] and Intel 64-bit [Intel, Vol 3, 2014] processors. Currently, many OS, e.g. Linux and freeBSD, already have 64-bit versions. Although hard to justify the need, it is also possible to extend MTX to 64-bit mode. Interested readers may want to experiment with this. In that case, the following information on 64-bit x86-64 CPUs may be helpful.

14.9.2 x86-64 CPUs

AMD classifies the 64-bit environment of x86-64 CPUs into compatibility mode and long mode. In compatibility mode, a x86-64 CPU is the same as a x86 in 32-bit protected mode. In long mode, all registers and addresses are 64 bits, although the current implementation uses only 48-bit address. The startup sequence of a x86-64 CPU is as follows.

(1). Start in 16-bit real mode.

(2). Switch to 32-bit compatibility mode by setting up the GDT, LDT and IDT, page directory and page tables, then enter protected mode and/or enable paging. These are the same as x86 in 32-bit protected mode.

(3). Initialize long mode, which requires a 64-bit IDT containing 64-bit interrupt-gate descriptors and 64-bit interrupt handlers, a GDT, any LDT, if needed, and a single 64-bit TSS. The code segment descriptor must specify whether the CPU is

executing in 64-bit or compatibility mode. The segment selectors DS, ES and SS are still used in compatibility mode but they are ignored in 64-bit mode. The segment selectors FS and GS are for 64-bit mode, which can be used by an OS. In 64-bit mode, paging is by 4 levels (or 3 levels of page tables). Page size is 4KB or 2 MB, super page size is 1 GB. Before entering long mode, paging must be disabled.

(4). Enable long mode by setting the long-mode enable control bit (EFER.LME) to 1. The long mode is activated only when paging is enabled. In 64-bit long mode, there is no segmentation, only paging.

(5). In 64-bit mode, interrupt processing is done by 64-bit IDT and 64-bit interrupt handler code. In compatibility mode, an interrupt pushes/pops [oldSS, oldesp] only if it involves a privilege level change. In 64-bit mode, an interrupt always pushes [oldSS, oldrsp], which is popped by iret.

(6). Hardware task switching is not supported in 64-bit mode. A 64-bit TSS in the GDT is still required to define the interrupt stack. (5) and (6) provide hints as how to set up the initial kernel stack of a process in 64-bit mode.

(7). The x86-64 CPU has additional control registers. The reader may consult AMD64 AMD64 2011 for more information.

14.9.3 Function Call Convention in 64-bit Mode

The x86-64 CPU has 8 more general registers. The registers are denoted as rip, rsp, rax, rbx, rcx, rdx, rbp, rsi, rdi and r8 to r15. During function calls, the first 6 parameters are passed in the registers rdi, rsi, rdx, rcx, r8 and r9, in that order. The called function must save and restore rbx, rbp and r13-15 if they are to be altered. On entry to a called function, the compiled code of GCC may allocate a fixed size stack area for all local variables and use rsp as the stack frame pointer. When developing a 64-bit OS kernel we must observe these function call conventions, especially when interfacing C with assembly code.

14.10 Summary

In this chapter, we presented 3 versions of MTX in 32-bit protected mode. MTX32.1 uses protected segments, MTX32.2 uses static paging and PMTX uses dynamic paging. It is shown that for protected mode operations we only need to add virtual address mapping and set up the IDT for exception and interrupt processing. Other than these, it requires only a few minor changes to the real mode RMTX kernel to make it also work in protected mode. This shows that when studying the principle of OS design, the actual memory management hardware is relatively unimportant. It also demonstrates that, despite the architectural differences between real mode and protected mode, the same design principle and implementation technique can be applied to all cases.

Problems

1. In MTX32.1, the LDT of each process defines only a code segment and a data segment. The data segment is used for both data and stack of the Umode image.

(a). Can we define a separate data and stack segments for each Umode image? Justify.

(b). Assume that the Umode image is an executable a.out or ELF file with size information, such as tsize, dsize and bsize. Show how to set up the LDT segments of a process to enforce proper access to the segments.

2. In MTX32.1, VA=PA in Kmode. In Umode, the VA range of every process is from 0 to 2 MB. Assume that we want the Umode VA begins at 1 MB.

(1). Show how to generate such Umode images. (2). Show how to set up the process LDTs.

3. The Intel x86 CPU supports super pages of 4 MB page size, which may be used when MTX32.2 and PMTX start. Describe the needed changes to entry.s and discuss the merit of using super pages.
4. In MTX32.2, page directories and page tables are constructed in the low end of physical memory. Assume 512 MB physical memory. Modify MTX32.2 to construct the page directories and page tables in the high end of physical memory. What is the disadvantage of this scheme?
5. In MTX32.2, we assume that every Umode image size is 4 MB. Modify fork1() for different image sizes, e.g. 8 MB, 64 MB, etc. Design an algorithm which builds the page tables for an image of size=SIZE in multiples of 4KB.
6. In MTX32.2, the process image is allocated as a piece of contiguous memory. Design a page frame management algorithm, which allocate/deallocate page frames dynamically. Manage the free page fames by (1). a bitmap. (2). a link list.

Discuss the advantages and disadvantages of each scheme.

7. In MTX32.2, the kernel virtual address space is mapped low and user virtual address space is mapped high. Try to reverse the the virtual address mappings, i.e. kernel space begins from 2 GB and user space begins at 0.
8. In PMTX, the physical memory area from 1 MB to 4 MB is dedicated to the PMTX kernel. Assume that PMTX kernel size=5 MB. Show

(1). How to boot up such a PMTX kernel? (Hint: consult booting bzImage of Linux in Chap. 3)

(2). Show how to modify enrty.s to set up the initial paging environment for such a PMTX kernel.

9. In PMTX, a process in kexit() keeps both its kstack and pgdir, which are eventually deallocated by its parent in kwait().

(1). Why are these necessary?

(2). What would happen if a process deallocates its pgdir or kstack in kexit()? How to handle such problems?

10. In PMTX, assume 512 MB physical memory and that the kernel's virtual address space is from 3 GB to 4G.

(1). Show how to set up the kernel mode page directory and page tables.

(2). Show how to set up the user mode page directory and page tables.

11. In PMTX, free page frames are managed by a pfreelist. Use a bit map to manage free page frames. Compare the advantages and disadvantages of the two schemes.
12. Null pointers: In PMTX, if a user mode program tries to dereference a null pointer, it should generate a page fault. Devise a scheme to implement this.
13. Modify PMTX to run processes with Umode images with the memory size option

cmd parameter-list -m SIZE k

where SIZE is a multiple of KB, e.g. –m 4096 k for 4096 KB, -m 7200 k for 7200 KB, etc.

14. In PMTX, kexec() always allocates a new image for the process. Modify it as follows.

(1). Allocate a new image only if the process is vforked or needs a image of different size. Otherwise, use the original image area to load the new file.

(2). Discuss the advantages and disadvantages of the new scheme.

15. In PMTX, the user mode program, demandpage, demonstrates demand-paging. The program issues a page_out(n) system call to mark the nth page of the process as not present. When the process attempts to reference the missing page, it generates a page fault. The kernel's page fault handler (ehandler() in trapc.c) uses page_in(n) to mark the missing page as present, allowing the process to continue.

(1). Modify page_out(n) to deallocate the nth page of a process.

(2). Modify page_in(n) to allocate a new page frame.

(3). The page fault handler assumes 4 MB image size. Modify it for different image sizes.

16. In PMTX without the –m SIZEm option, every Umode image size is 4 MB. The Umode page table of every process has 1024 entries, all of which have pre-allocated page frames. Show how to modify makePage() for the following cases.

(1). By actual image size, e.g. code + data + bss, is only 8KB. The Umode stack is at the high end of the VA with an initial stack size of 8KB. Pages that are not needed should have no page frames.

(2). Same assumptions as in (1), show how to expand the image's heap by page size.

(3). Same assumptions as in (1), except that the stack should be directly above the bss section. Show how to expand the image's heap by page size.

(4). In an ELF executable file, each program section has a virtual address, a memory size and a R|W|Ex flag. When loading an ELF file, allocate pages for the sections as needed and set code pages for RE(executable), rodata pages for RO, data and bss pages for RW.

17. In kexec() of PMTX, a vforked process first allocates a new image. Then it tries to load a image file. If the loading fails, e.g. due to an invalid file name, it release the new image and returns -1 to the original image. In this case, the vforked process should terminate since it cannot continue to execute in the shared image. Implement the following alternatives if loading file fails:

(1). load a default file, which issuses exit(1) to terminate.

(2). copy the binary code of exit(2) to the beginning of the new image and let it return to execute the new image. (HINT: write syscall(9, 2) in assembly and use as –ahls to see the binary code).

(3). call kexit(123) to terminate directly.

18. In PMTX, implement demand-paging by NRU.
19. In PMTX, implement block device I/O buffering by demand-paging.
20. Extend PMTX to 64-bit.

References

AMD64 Architecture Programmer's manual Volume 2: System Programming, 2011
Antonakos, J.L., "An introduction to the Intel Family of Microprocessors", Prentice Hall, 1999.
Bovet, D.P., Cesati, M., "Understanding the Linux Kernel, Third Edition", O'Reilly, 2005
Cox, R., Kaashoek, F., Morris, R. "xv6 a simple, Unix-like teaching operating system, xv6-book@pdos.csail.mit.edu, Sept. 2011.
Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, 2014
Intel i486 Processor Programmer's Reference Manual, 1990
McKusick, M.K., Bostic, K., "The Design and Implementation of the 4.4 BSD Operating System", Addison-Wesley, 1996
McKusick, M.K., Neville-Neil, G.,"The Design and Implementation of the FreeBSD Operating System", Addison-Wesley, 2004.
Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 (May 1995)
Youngdale, E., "The ELF Object File Format: Introduction", Linux Journal, April, 1995.

Chapter 15
Symmetric Multiprocessing MTX

15.1 Multiprocessor Systems

A multiprocessor system consists of a multiple number of processors, which share main memory and I/O devices. If the shared main memory is the only memory in the system, it is called a Uniform Memory Access (UMA) system. If, in addition to the shared memory, each processor also has private local memory, it is called a Non-uniform Memory Access (NUMA) system. If the roles of the processors are not the same, e.g. only some of the processors can execute kernel code while others can not, it is called an Asymmetric MP (ASMP) system. If all the processors are functionally identical, it is called a Symmetric MP (SMP) system. With the current multicore processor technology, SMP has become virtually synonymous with MP.

15.2 SMP-Compliant Systems

A SMP system requires much more than just a multiple numbers of processors or processor cores. In order to support SMP, the system architecture must have additional capabilities. Intel's Multiprocessor Specification (Intel Multiprocessor Specification 1997) defines SMP-compliant systems as PC/AT compatible systems with the following capabilities.

1. Support interrupts routing and inter-processor interrupts. In a SMP-compliant system, interrupts from I/O devices can be routed to different processors to balance the interrupt processing load. Processors can interrupt each other by Inter-Processor Interrupts (IPIs) for communication and synchronization. In a SMP-compliant system, these are provided by a set of Advanced Programmable Interrupt Controllers (APICs). A SMP-compliant system usually has a system-wide IOAPIC and a set of local APICs of the individual processors. Together, the APICs implement an inter-processor communication protocol, which supports interrupts routing and IPIs.

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_15

2. An extended BIOS, which detects the system configuration and builds SMP data structures for the operating system to use.
3. When the system starts, one of the processors is designated as the Boot Processor (BSP), which executes the boot code to start up the system. All other processors are called Application Processors (APs), which are held in the idle state initially but can receive IPIs from the BSP to start up. After booting up, all processors are functionally identical.

15.3 SMP System Startup Sequence

15.3.1 SMP Data Structures

When a SMP system starts, BIOS detects the system hardware configuration and creates a set of SMP data structures for the operating system to use. The data structures include a Floating Pointer Structure (FPS), which is required, and a MP Configuration Table. If the FPS is absent, the system is not SMP-compliant. If the configuration table does not exist, the operating system may use a default MP configuration table, which is valid for only two processors. The FPS is in one of the following locations, which the BSP must search in order.

- first 1 KB of the extended BIOS data area.
- last 1 KB of real-mode base memory.
- BIOS read-only memory between 0xA0000 and 0xFFFFF.

The FPS is a 16-byte structure. It begins with a 4-byte signature "_MP_". The next field is a 4-byte pointer to the MP configuration table. The configuration table is divided into three parts: a header, a base section, and an extended section. The header begins with the four-byte signature "PCMP". It contains OEM information and the number of entries in the base section. The base section consists of a set of entries which describe processors, system buses, IOAPIC, I/O interrupt assignments and local APIC interrupt assignments. The first byte of each entry is the entry type. For processor entries, the type is 0 and the entry length is 20 bytes. All other entries are 8 bytes. The OS kernel must parse the MP configuration table to create OS specific data structures, such as the APIC ID, version and type of each processor, as well as the address of the system's IOAPIC.

As an example, MTX has a mp command, which scans the MP configuration table of a SMP-compliant PC. Figure 15.1 shows the sample outputs (with added comments) of running the mp program for a VMware virtual machine on a dual-core host PC. For other multicore PCs, the outputs vary but are similar.

Figure 15.1 shows that the virtual machine has only one IOAPIC at 0xFEC00000. Local APICs are at 0xFEE00000. These are memory-mapped addresses of processor registers. Each processor may use the same APIC address to access its own local APIC.

```
/************ Sample outputs of MTX's mp command ***********/
// Search for MP flopating pointer structure
Serach BIOS ROM area 0xF0000 to 0xFFFFF      (found the FPS)
signature = _MP_                    // MP table signature
mp_table_addr=0x9FD70               // MP configuration table address
signature = PCMP                    // signature
base_len=268 rev=0x4 cksum=0x6B     // length, MP version 1.4
0x49  0x4E  0x54  0x45  I N T E     // OEM information
0x4C  0x20  0x20  0x20  L           // INTEL 440BX
0x34  0x34  0x30  0x42  4 4 0 B
oemTablePtr=0x0 entryCount=25       // number of entries=25
localAPICAddr=0xFEE00000            // local APIC address
******* CPU #0  ********            // processor entries (type 0)
0x0  0x0  0x11  0x3                 // APICid=0; 0x3=BSP, enabled
cpu_signature=0x6F6 flags=0xFEBFBFF
******* CPU #1  ********
0x0  0x1  0x11  0x1                 // APICid=1; 0x1=AP, enabled
cpu_signature=0x6F6 flags=0xFEBFBFF
// bus entries (type 1)
0x02 PCI                            // bus ID, string
0x23 ISA                            // bus ID, string
// IOAPIC entry (type 2)
0x2 0x2 0x11 0x1 addr=0xFEC00000  // I/O APIC address
// IOAPIC Interrupts Assignment entries (type 3)
0x23  0x0  0x2  0x0                 // ISA IRQ0 routing
// Local APIC Interrupt Assignemnt entry (type 4)
0x23  0x0  0xFF  0x0      // ISA IRQ0 to Int0 of all local APICs
```

Fig. 15.1 MP table example

During booting of a SMP-compliant system, the BSP must initialize the system hardware for SMP operations. These include

- Configure IOAPIC to route interrupts to local APICs.
- Configure and enable BSP's local APIC.
- Send INIT and STARTUP IPIs to activate other APs.
- Continue to initialize the OS kernel until it is ready to run tasks.

The following sections describe these actions in detail.

15.3.2 Configure IOAPIC

1. **Set up IOAPIC Interrupt Registers**

A SMP system usually has only one IOAPIC. The IOAPIC has 24 (64-bit) registers, which specify how to route interrupts and map IRQs to interrupt vectors. The registers are accessed indirectly through a pair of IOREGSEL and IOWIN registers, which are located at 0xFEC00000 and 0xFEC00010, respectively. Other registers are byte offsets from the IOWIN register. All IOAPIC registers must be accessed by 32-bit reads/writes. To access an IOAPIC register, first select the register by writing its byte offset to IOREGSEL. Then read/write a 32-bit value from/to the IOWIN register. For each IOAPIC interrupt register, it takes two operations to read/write

the two 32-bit halves of the register. Standard assignments of the IOAPIC interrupt registers are

```
IRQ 0-15 : IOAPIC registers  0-15
PCI A-D  : IOAPIC registers 16-19
```

For IRQ 0-15 interrupts, they must be set to edge-triggered and active high. For PCI A-D interrupts, they must be set to level-triggered and active low. In protected mode, the first 32 interrupt vectors are reserved. Each of the IOAPIC interrupt registers can be programmed with any of the remaining 224 vectors. The top four bits of an interrupt vector is the interrupt priority. Higher interrupt vector numbers have higher priorities. The standard PIC interrupt priorities are IRQ 0-2, 8-15, 3-7, where IRQ0 has the highest priority. If needed, the IRQs can be mapped by the IOAPIC interrupt registers to vectors that preserve their priorities. There are 16 different interrupt priorities, 0 to F. Since we cannot use the vectors 0x00-0x1F, it leaves only 14 available interrupt priorities. Thus, some of the IRQs must be mapped to vectors with the same priority. For example, if we remap IRQ0-15 to 0x21-0x2F and program the IOAPIC registers 0-15 for vectors 0x21-0x2F, then all the interrupts are of the same priority (2). Alternatively, we may choose vectors 0x20-0x9F for the PIC IRQs and assign two IRQs to each priority level, resulting in the following assignment.

```
PIC IRQs  : 0  1  2  8  9 10 11 12 13 14 15  3  4  5  6  7
vectors 0x:90 91 80 81 70 71 60 61 50 51 40 41 30 31 20 21
```

Similarly for the interrupt registers 16–19, which map PCI interrupts A–D. In addition to interrupt vectors, each IOAPIC interrupt register must also be programmed with interrupt delivery mode and destination. The delivery mode can be either physical or logical. The simplest way is to use logical delivery mode and route interrupts to APICs with the lowest priority. The priority of an APIC is in the APIC task priority register. Alternatively, we may also use physical delivery mode and program the IOAPIC registers to route interrupts to specific processors to balance the interrupt processing load.

2. **Switch to Symmetric I/O Mode**

In order to route interrupts to local APICs, the system must switch to symmetric I/O mode. For PCs with an Interrupt Mask Control Register (IMCR), this can be done by writing 0x01 to the IMCR's data register. Most virtual machines do not have an IMCR, so this step is optional.

```
. out_byte(0x22, 0x70);  // access IMCR
. out_byte(0x23, 0x01);  // force PIC IRQs to IOAPIC
```

3. **Disable 8259 PICs**

After setting up the IOAPIC to route interrupts, the 8259 PICs must be disabled to prevent multiple interrupts from the same source.

```
. out_byte(0x21, 0xFF);  // mask out master PIC
. out_byte(0xA1, 0xFF);  // mask out slave  PIC
```

15.3.3 Configure Local APICs

Each processor has a local APIC at the same base address 0xFEE00000. Local APIC registers are offsets from the APIC base address. The APIC registers can be accessed directly with 32-bit read/write. Some of the APIC registers and their usage are listed in Fig. 15.2. To enable the local APIC, write 0x010F to the spurious interrupt register (0x0F0). This also sets 0x0F as the default vector for spurious interrupts.

15.3.4 Send IPIs to Activate APs

After enabling the local APIC, the BSP must activate other APs by sending them INIT and STARTUP IPIs. This can be done by writing to the Interrupt Command Registers (0x310 = dest, 0x300 = IPI) to send IPIs to each individual AP or broadcast them to all the APs. In the latter case, the IPIs are issued as follows, where the delay is about 200 msec.

Register	Contents
0x020 :	ID register (unique ID; to be used as CPU ID)
0x080 :	task priority (as interrupt priority mask)
0x0B0 :	EOI (write 0 to signal end-of-interrupt)
0x0F0 :	spurious interrupt vector (also for enable APIC)
0x300 :	interrupt command register (generate IPIs)
0x310 :	destination ID of interrupt command
0x320 :	LVT timer (APCI timer: mask bit, mode and vector)
0x350 :	LVT LINT0 (INTR input)
0x360 :	LVT LINT1 (NMI input)
0x370 :	LVT error vector table
0x380 :	initial timer count
0x390 :	current timer count
0x3E0 :	timer divider

Fig. 15.2 APIC registers

```
.write 0x000C4500 to 0x300; delay; // broadcast INIT IPI to all APs
.write 0x000C4691 to 0x300; delay; // broadcast STARTUP IPI to all APs
```

15.3.5 Trampoline Code of APs

Each AP wakes up to execute a piece of trampoline code in 16-bit real mode. The trampoline code must begin at a 4 KB page boundary in real-mode memory. The location is determined by the vector number in the STARTUP IPI. In the above example, the vector value is 0x91, which tells the APs to begin execution from 0x91000 in real-mode memory. Each AP must switch to protected mode, set up page tables and configure local APIC before entering the OS kernel. In SMP, a processor may use IPIs to synchronize with other processors, such as to flush their TLBs and invalidate page table entries, etc.

15.4 From UP to SMP

Once a SMP system boots up, all the processors are functionally identical. In a SMP system, processes may run in parallel on different processors. A SMP kernel must be able to support parallel executions of multiple processes. Therefore, all kernel data structures must be protected to prevent race conditions. Traditionally, most OS kernels are designed for uniprocessor (UP) systems. Adapting a UP kernel for SMP is usually done in three stages (Cox 1995).

15.4.1 Giant Lock Stage

Early Linux (Cox 1995) and FreeBSD (McKusick and Neville-Neil 2004) were adapted to SMP by using a Giant Kernel Lock (GKL). In this scheme, the entire kernel is treated as a critical region and protected by a single global lock, which is usually a spinlock. In order to execute in kernel, a process must acquire the GKL first. Only the process which holds the GKL can execute in kernel. The GKL is not released until the process has completed the kernel mode operation. The main advantage of this approach lies in its simplicity. It requires very little change to a UP kernel to make it work in SMP. The disadvantage is that, while a process executes in kernel, all other processes must either busily wait for the GKL or can only execute in user mode. Such a system supports MP but does not realize the capabilities of a SMP system.

15.4.2 Coarse Grain Locking Stage

To improve concurrency, a SMP kernel may use separate locks to protect different subsystems or groups of closely related data structures. This allows for some degree of concurrency but with only limited improvements in performance. For this reason, most SMP kernels use coarse grain locking as a transitory stage.

15.4.3 Fine Grain Locking Stage

System V Unix, AIX and current versions of Linux (version 2.6 and later) all use fine grain locking for SMP. In this scheme, small locks are used to protect individual kernel data structures, allowing parallel operations on different data structures. Although every SMP system strives for fine grain locking, there is no general consensus as to what constitutes a fine grain lock. Currently, the approach used by almost all SMP kernels is as follows.

- Identify the data structures in a UP kernel that need protection. Use locks to implement operations on the data structures as critical regions. Examples of such data structures include process table, process scheduling queue, page tables, in-memory inodes and I/O buffers, etc. This generally leads to concurrency at the individual data structure level in the original UP kernel.
- To further improve concurrency, try to decompose the data structures into separate parts. If the decomposed parts are logically related, devise ways to synchronize their operations. However, most existing SMP systems do not yet use this strategy.

As the lock grain size decreases, the overhead due to additional locking and synchronization will increase. Eventually, the process of reducing the lock grain size must stop at some point when it no longer improves concurrency and overall speed. The problem of designing SMP kernels is essentially the same as that of designing parallel algorithms (Grama et al. 2003). Although the principle is simple, the difficulty is how to decompose the data structures to allow for maximal concurrency yet with minimal interactions among the decomposed parts. So far, there are no definitive answers.

Rather than discussing SMP in general terms, we shall consider the specific problem of adapting MTX for SMP. In line with the evolutional style of this book, we shall develop the SMP system in two steps. In the first step, we extend the UP PMTX to SMP_PMTX, which uses fine grain locks and modified UP sequential algorithms for SMP. In the second step, we focus on the design and use of parallel algorithms for SMP, and extend SMP_PMTX to the final version of SMP_MTX, which uses parallel algorithms on decomposed kernel data structures to improve both concurrency and efficiency. In addition, we shall also show an experimental SMP MTX in 16-bit real mode.

15.5 SMP_PMTX in 32-bit Protected Mode

In this section, we shall extend the UP PMTX kernel to SMP_PMTX, which is the initial version of SMP_MTX. Although some of the discussions are SMP_PMTX specific, they may also be regarded as a general methodology for adapting UP kernels to SMP systematically.

15.5.1 Running PROCs and CPU Structures

In the PMTX kernel, running is a global pointer to the current running process. In SMP, a single running pointer is no longer adequate because each CPU may be executing a different process. We must be able to identify the process that's executing on each CPU. There are two possible ways to do this. The first one is to define NCPU PROC pointers, PROC *run[NCPU], and let run[i] point to the PROC that's currently executing on CPUi. The drawback of this scheme is that it is rather inconvenient to access run[cpuid] in assembly code unless we store the index in a CPU register. The second method is to use virtual memory. In SMP_PMTX, we represent each CPU by a cpu structure (defined in type.h) and map the cpu structure by a spare segment register, e.g. the gs register, to a virtual address. To every CPU its cpu structure is always at gs:0 and its current executing PROC pointer is always at gs:4, etc. Instead of gs, we may also use es or fs to map the cpu structure.

15.5.2 Spinlocks and Semaphores

In SMP, processes may run in parallel on different CPUs. All data structures in a SMP kernel must be protected to prevent corruptions from race conditions. Typical tools used to achieve this are spinlocks and semaphores. Spinlocks are suitable for CPUs to wait for critical regions of short durations in which task switch is either unnecessary or not permissible, e.g. in an interrupt handler. To access a critical region, a process must acquire the spinlock associated with the critical region first, as in

```
SPINLOCK s = 0;     // initial value = 0
slock(s);           // acquire spinlock
  // access critical region CR
sunlock(s);         // release spinlock
```

When a process exits the critical region CR, it releases the spinlock to allow other CPUs to acquire the spinlock. slock(s)/sunlock(s) are implemented in assembly because they rely on test_and_set like atomic instructions. For the x86 CPUs, the equivalent instruction is XCHG, which atomically exchanges the contents of a CPU register with a memory location.

For critical regions of longer durations, it is preferable to let the process wait by giving up the CPU. In such cases, semaphores are more suitable. In SMP, each semaphore must have a spinlock field to ensure that all operations on a semaphore can only be performed in the critical region of the semaphore's spinlock. Similar modifications are also needed for mutex and mutex operations, which are used for threads synchronization. Since PMTX already uses semaphores in file system and device drivers, in SMP_PMTX we shall use both spinlocks and semaphores to protect kernel data structures.

15.5.3 Use Sleep/Wakeup in SMP

As a synchronization mechanism, sleep/wakeup works well in UP but is unsuited to SMP. This is because an event is just a value, which does not have an associated memory location to record the occurrence of the event. When an event occurs, wakeup tries to wake up all processes sleeping on the event. If no process is sleeping on the event, wakeup has no effect. This requires a process to sleep first before another process tries to wake it up later. This sleep-first and wakeup-later order can always be achieved in UP but not in SMP. In a SMP system, processes may run in parallel on different CPUs. It is impossible to guarantee the process execution order. Therefore, in their original form, sleep/wakeup cannot be used in SMP. However, we can modify sleep/wakeup to make them workable in SMP.

Consider the case in which a parent process waits for a ZOMBIE child. In a UP kernel, if the parent calls wait() before the child has died, the parent would call sleep() to go to sleep since it cannot find a ZOMBIE child. In this case, the parent completes the sleep operation before the child can run. Later, when the child calls exit() to die, it tries to wake up the parent. Since the parent is already in the sleeping state, the child's wakeup call will find the parent. On the other hand, if the child terminates first, the child's wakeup call would miss the parent but it leaves behind a ZOMBIE PROC for the parent to find. When the parent calls wait() later, it will find the ZOMBIE child PROC and does not sleep. In either case, sleep and wakeup work well because in a UP kernel the actions of the parent in wait() and child in exit() are serialized, i.e. one at a time without interleaving. In a SMP kernel, the situation is different. When a parent calls wait(), the child may be running on a different CPU at the same time. Assume that the parent in wait() has just checked that there is no ZOMBIE child and is about to call sleep(). Right in this time gap, if the child running on a different CPU calls exit(), it would become a ZOMBIE and try to wake up the parent. Since the parent has not slept yet, the child's wakeup call would have no effect. Then the parent goes to sleep, but it already missed the child's wakeup call. Based on this analysis, it is immediately clear that, in order for the parent and child not to miss each other, the parent's sleep action and the child's exit action must be serialized. This can be achieved by requiring them to go through a common critical region, which serializes their actions even if they are executing in parallel. Therefore, we can modify sleep/wakeup as follows. For each event, define a spinlock

sw=0, and impose the following restrictions on the usage of sleep/wakeup: when processes call sleep and wakeup on an event, they must execute in the same critical region of the sw spinlock. While holding the spinlock, if a process has to sleep, it must complete the sleep and release the spinlock in a single indivisible or atomic operation, as shown below.

1. For each event, define a SPINLOCK sw=0;
2. A process must acquire the sw spinlock before calling the modified sleep() function.

```
sleep(int event, SPINLOCK sw)
{  // caller holds the spinlock sw
   sleep on event;            // go to sleep as in UP kernel
   sunlock(sw);               // release spinlock
   tswitch();                 // switch process
}
```

3. When a process calls wakeup() for the same event, it must execute in the CR of the spinlock.

```
   slock(sw);                  // acquire spinlock
   change condition to OK;     // set condition inside CR
   wakeup(event);              // issue wakeup inside CR
   sunlock(sw);                // release spinlock
```

where the wakeup() function does not need any change but it must be issued inside the same critical region of a spinlock. With these modifications, we may use sleep/wakeup to implement wait and exit in a SMP kernel. The modified sleep/wakeup can also be used for general process synchronization. Assume that two processes Pi and Pj share some common data, e.g. a counter, which is modified by Pi and used by Pj to make decisions. To synchronize Pi and Pj, define a spinlock sw=0, and require Pi and Pj to obey the following rules.

```
      Process Pi:          |     Process Pj:
-------------------------- |-----------------------------
{                          |  { while(1){
  acquire sw lock;         |      acquire sw lock;
  change counter value;    |      if (counter value OK) break;
  wakeup(event);           |      sleep(event, sw);
  release sw lock;         |    }
                           |    release sw lock;
}                          |  }
-------------------------------------------------------
```

In the above pseudo-code, only one process can be inside the critical region at a time. If Pi and Pj obey these rules, there will be no race condition and hence no missing wakeup calls. This is the process synchronization mechanism used in xv6 (Cox et al. 2011). The modification amounts to putting the original sleep/wakeup on steroids since the shared variable inside a common critical region essentially makes up for the lack of memory location of an event. It is almost equivalent to the condi-

tion variable of Pthreads [Pthreads 2015], which relies on a mutex lock. The only difference is that in the modified sleep/wakeup, when a process resumes after sleep, it must acquire the spinlock again. In Pthreads, when a blocked process resumes inside a condition variable, the condition variable's mutex is automatically locked. Although it is possible to use only the modified sleep/wakeup for process synchronization in SMP, the drawback is that a process must retry after each sleep. In SMP_PMTX, we use semaphore to implement wait and exit as follows. In each PROC structure, define a semaphore wchild = 0. In kwait(), a process uses P(wchild) to wait for ZOMBIE children. In kexit(), a process uses V(parent→wchild) to unblock the parent. Using semaphores, there is no race condition between the parent and child.

15.5.4 Protect Kernel Data Structures

In a UP kernel, only one process executes at a time. So data structures in a UP kernel do not need any protection against concurrent process executions. When adapt a UP kernel for SMP, all kernel data structures must be protected to ensure that processes can only access them one at a time. The required modifications can be classified into two categories.

1. The first category includes kernel data structures which are used for allocation and deallocation of resources. These include
   free PROC list, free page frame list, pipe structures, message buffers, bitmaps for inodes and disk blocks, in-memory inodes, open file table, mount table, etc.

Each of these data structures can be protected by either a spinlock or a lock semaphore. Then modify the allocation/deallocation algorithms as critical regions of the form

```
allocate(resource)
{
   LOCK(resource_lock);
    // allocate resource from the resource data structure;
   UNLOCK(resource_lock);
   retrun allocated resource;
}
deallocate(resource)
{
   LOCK(resource_lock)
    // release resource to the resource data structure;
   UNLOCK(resource_lock);
}
```

where LOCK/UNLOCK denote either slock/sunlock on spinlocks or P/V on lock semaphores. Since P/V require implicit operations on the semaphore's spinlock, it is more efficient to use spinlocks directly. For example, in SMP_PMTX we define a spinlock sfreelist = 0 to protect the free PROC list and modify get_proc()/put_proc() as follows.

```
PROC *get_proc()
{
   slock(sfreelist);
    // *p = get a PROC pointer from free PROC list;
   sunlock(sfreelist);
   return p;
}
void put_proc(PROC *p)
{
    slock(sfreelist);
     // enter p into free PROC list;
    sunlock(sfreelist);
}
```

where the operations on the free PROC list are exactly the same as that in a UP kernel. Similarly, we can use spinlocks to protect other resources. In short, this category includes all the kernel data structures for which the behavior of a process is to access the data structure without pausing.

2. The second category includes the cases in which a process must acquire a lock first in order to search for a needed item in a data structure. If the needed item already exists, the process must not create the same item again, but it may have to wait for the item. If so, it must release the lock to allow concurrency. However, this may lead to the following race condition. After releasing the lock but before the process has completed the wait operation, the item may be changed by other processes running on different CPUs, causing it to wait for the wrong item. This kind of race condition cannot occur in UP but is very likely in SMP. There are two possible ways to prevent such race conditions.

   2.1. Set a reservation flag on the item before releasing the lock. Ensure that the reservation flag can only be manipulated inside the critical region of the lock. For example, in the iget() function of the MTX file system, which returns a locked minode, each minode has a refCount field, which represents the number of processes still using the minode. When a process executes iget(), it first acquires a lock and then searches for a needed minode. If the minode already exists, it increases the minode's refCount by 1 to reserve the minode. Then it releases the lock and tries to lock the minode. When a process releases a minode by iput(), it must execute in the same critical region as iget(). After decrementing the minode's refCount by 1, if the refCount is non-zero, meaning that the minode still has users, it does not free the minode. Since both iget() and iput() execute in the same critical region of a lock, race condition cannot occur.

   2.2. Ensure that the process completes the wait operation on the needed item before releasing the lock, which eliminates the time gap. When using spinlocks this is the same technique as putting sleep/wakeup on steroids. When using semaphore locks we may use the PV(s1,s2) operation defined in Chap. 6, which atomically blocks a process on semaphore s1 before releasing semaphore s2. As an example, consider the iget()/iput() functions again. Assume that mlock is a lock semaphore for all the minodes in memory and each minode has a lock semaphore minode.sem = 1. We only need to modify iget()/iput() slightly as follows.

```
MINODE *iget(int dev, int ino) // return a locked minode=(dev,ino)
{
   P(mlock);                        // acquire minodes lock
   if (needed minode already exists){
      if (!CP(minode.sem)           // if minode is already locked
         PV(minode.sem, mlock);     // atomically (P(s1),V(s2))
      return minode;                // return locked minode
   }
   // needed minode not in memory, still holds mlock
   allocate a free minode;
   P(minode.sem);                   // lock minode
   V(mlock);                        // release minodes lock
   load inode from disk into minode;
   return minode;
}
void iput(MINDOE minode)
{
   // caller already holds minode.sem lock
   P(mlock);
   // do release minode operations as in PMTX;
   V(minode.sem);
   V(mlock);
}
```

Note that, in iget() the locking order is to lock mlock first, then minode, which is opposite to that of iput(). Despite the opposite locking orders, deadlock cannot occur. This is because in iget() a process tries to lock an existing minode by CP(). If the minode is already locked, it waits for the minode and releases the mlock in a single atomic operation.

15.5.5 Adapt UP Algorithms for SMP

In addition to using locks to protect kernel data structures, many algorithms used in UP kernels must be modified to suit SMP. We illustrate this by examples.

15.5.5.1 Adapt UP Process Scheduling Algorithm for SMP

A UP kernel usually has only a single process scheduling queue. With a single process scheduling queue, we may adapt the UP process scheduling algorithm for SMP as follows. Define a spinlock, srQ, to protect the scheduling queue. During task switch, a process must acquire the srQ spinlock first, which is released by the next running process when it resumes.

15.5.5.2 Adapt UP Pipe Algorithm for SMP

In PMTX, pipes are implemented by the UP algorithm of Sect. 6.14.3, which uses the conventional sleep/wakeup for synchronization. We may adapt the UP pipe algorithm for SMP by adding a spinlock to each pipe and requiring pipe readers and

writers to execute in the same critical region of the spinlock. While holding the spinlock, if a process has to sleep for data or room in the pipe, it must complete the sleep before releasing the spinlock, which can be done by replacing the conventional sleep(event) with the modified sleep(event, spinlock) operation.

15.5.5.3 Adapt UP I/O Buffer Management Algorithm for SMP

In Chap. 12, we showed several I/O buffer management algorithms for block devices. Again, the algorithms work only in UP because they all assume only one process runs at a time. We may adapt the algorithms for SMP by adding a spinlock and ensuring that both getblk() and brelse() are executed in the same critical region. While holding the spinlock, if a process has to wait, it must complete the wait and release the spinlock in a single atomic operation. The technique is exactly the same as that of adapting the UP pipe algorithm for SMP. Similarly, we may adapt other UP algorithms for SMP.

15.5.6 Device Driver and Interrupt Handlers for SMP

In an interrupt-driven device driver, process and interrupt handler usually share data buffer and control variables, which form a critical region between the process and interrupt handler. In a UP kernel, when a process executes a device driver, it can mask out interrupts to prevent interference from the interrupt handler. In SMP, masking out interrupts is no longer sufficient. This is because while a process executes a device driver, another CPU may execute the device interrupt handler at the same time. In order to serialize the executions of processes and interrupt handlers, device drivers in SMP must be modified also. Since interrupt handlers cannot sleep or block, they must use spinlock or equivalent mechanisms. In the following, we illustrate the principles of SMP driver design by specific examples in the SMP_PMTX kernel.

1. The PC's console display is a memory-mapped device, which does not use interrupts. To ensure processes executes putc() one at a time, it suffices to protect the driver by a spinlock.
2. In the FD driver, only one process can execute the driver at a time. After issuing an I/O command, the process waits on a semaphore for interrupt. The interrupt handler only unblocks the process but does not access the driver's data structure. So the FD driver does not need any change.
3. In the HD and ATAPI driver, process and interrupt handler shared data structures. For such drivers, it suffices to use a spinlock to serializes the executions of process and interrupt handler.
4. In the timer driver, process and interrupt handler share timer service and process scheduling queues but the process never waits for timer interrupts. In this case, it suffices to protect each timer dependent data structure by a spinlock.

5. Char device drivers also use I/O buffers for better efficiency. In PMTX, all char device drivers use semaphores for synchronization. To adapt the derivers to SMP, each driver uses a spinlock to serialize the executions of process and interrupt handler. While holding the spinlock, if a process has to wait for data or room in the I/O buffer, it uses PV() or PU() to atomically wait and release the spinlock.
6. Process and interrupt handler in a SMP device driver must obey the following timing order.

```
       Process                    Interrupt Handler
-------------------------------------------------------------
(a). disable interrupts     |
(b). acquire spinlock       |   (a). acquire spinlock
(c). start I/O operation    |   (b). process the interrupt
                            |   (c). start next I/O, if needed
(d). release spinlock       |   (d). release spinlock
(e). enable interrupts      |   (e). issue EOI
-------------------------------------------------------------
```

In both cases, the order of (d) and (e) must be strictly observed in order to prevent a process from locking itself out on the same spinlock.

15.5.7 SMP_PMTX Kernel Organization

Following the above principles, we have adapted the UP kernel of PMTX for SMP. The SMP_PMTX kernel consists of the following file tree, which is the same as in PMTX but with added features to support SMP.

```
SMP_PMTX--|-- Makefile and mk script
          |-- SETUP:   boot.s, setup.s, apentry.s
          |-- header:  type.h, include.h     // added SMP types
          |-- kernel:  kernel source files   // added files for SMP
          |-- fs:      file system files     // modified for SMP
          |-- driver:  device driver files   // modified for SMP
```

The main features in support of SMP are in the following files.

type.h define NCPU = 16 CPUs and CPU structure type
apentry.s startup code of APs in 16-bit real mode
mp.h FP and MP structure types
mp.c MP table scanning function
smp.h SMP types and structures, e.g. APIC and IOAPIC structure types
smp.c SMP configuration and startup code

15.5.8 Bootable SMP_PMTX Image

A bootable SMP_PMTX image consists of the following pieces.

```
Sector | 0  |  1  |  2    3   | 4 ......................|
       ---------------------------------------------------
       |BOOT|SETUP|  APentry  | 32-bit SMP_PMTX kernel ..|
       ---------------------------------------------------
```

During booting, the booter loads BOOT+SETUP in block 0 to 0x90000, APentry in block 1 to 0x91000 and the MTX kernel to 0x10000. For protected mode MTX kernels, the last word in the BOOT sector contains the boot signature 'PP'. When the booter detects this, it jumps to 0x90200 to run SETUP, which brings up the system in 32-bit protected mode. When the SMP_PMTX kernel starts, the boot processor (BSP) first calls findmp() to search for the FPS and MP table. It scans the MP table and returns the number of CPUs in the system. Currently, the SMP_PMTX kernel supports up to 16 CPUs. If desired, it can be extended easily for more CPUs. If the system has only one CPU, it falls back to UP mode. Otherwise, it calls smp() to configure the system for SMP operations and bring the system up in SMP mode. After initializing the kernel, the BSP broadcasts INIT and STARTUP IPIs to the APs with the vector 0x91, which corresponds to the loading address of APentry. Each AP begins by executing APentry in 16-bit real mode.

15.5.9 SMP_PMTX Kernel Startup Sequence

This section describes the startup sequence of the SMP_PMTX kernel. In addition to the subsection titles, we also use sequence numbers to show the logical order of the steps.

15.5.9.1 Enter 32-bit Protected Mode

(1). SETUP: SETUP defines an initial GDT to enter protected mode. The initial GDT defines both kernel CS and DS as 4 GB flat segments.

```
igdt:
  .quad 0x0000000000000000  # null descriptor
  .quad 0x00cF9A000000FFFF  # kcs: 9=PpLS=1001, A=non-conforming
  .quad 0x00cF92000000FFFF  # kds: 9=PpLS=1001, 2=R|W data segment
```

After moving the GDT to a known location (0x9F000), SETUP loads GDTR with the initial GDT and enters protected mode. Then it moves the MTX kernel from 0x10000 to 0x100000 (1 MB) and ljmp to pm_entry in entry.s to execute the kernel startup code.

15.5.9.2 Set up Initial Paging and GDTs

(2). entry.s sets up an initial paging environment, which allows the kernel to access the lowest 8 MB physical memory by either physical or virtual addresses. The initial ipgdir is defined at the offset address 0x1000 in the SMP_PMTX kernel.

```
        .org 0
pm_entry:
# entry.s code in 32-bit GCC assembly
# set up initial ipgdir and page tables; then call init()
        .org 0x1000           # at offset 4KB in kernel
ipgdir:
        .long 0x00102007      # point to pg0 at 0x102000
        .long 0x00103007      # point to pg1 at 0x103000
        .fill 510,4,0         # fill with 510 entries of 0
        .long 0x00102007      # point to pg0 at 0x102000 also
        .long 0x00103007      # point to pg1 at 0x103000 also
        .fill 510,4,0         # fill with 510 entries of 0
        .org 0x2000
pg0:                          # by entry.s code: for 0-4MB
        .org 0x3000
pg1:                           # by netry.s code: for 4-8MB
```

The two initial page tables, pg0 and pg1, are filled in entry.s to map the lowest 8 MB physical memory as real address and also as virtual address in the range of [2 GB, 2 GB + 8 MB]. Then it uses the initial ipgdir to turn on paging and forces the BSP to use virtual addresses starting from 0x80000000. So far, the startup actions are identical to that of PMTX. The SMP part begins here. At 0x4000 are NCPU = 16 GDTs, one per CPU, as shown below.

```
        .org 0x4000
kgdt: .rept  NCPU   # NCPU=16    #                      Index
        .quad 0x0000000000000000   # null descriptor      0x00
        .quad 0x00cF9A000000FFFF   # kcs 00cF PpLS=1001=9 0x08
        .quad 0x00cF92000000FFFF   # kds                  0x10
        .quad 0x0000000000000000   # tss                  0x18
        .quad 0x00cFFA000000FFFF   # ucs 00cF PpLS=1111=F 0x20
        .quad 0x00cFF2000000FFFF   # uds                  0x28
        .quad 0x80C0920000000027   # CPU struct=40 bytes  0x30
        .endr
gdt_desc:  .word 56-1              # for CPU0's GDT
           .long kgdt              # GDT address of BSP
        .org 0x5000
idt:  .fill 1024,4,0               # 2KB IDT at 0x5000
        .org 0x6000
kpgdir:.fill 1024,4,0              # kernel mode kpgdir at 0x6000
        .org 0x8000                # other kernel code begins here
```

The first six segments of each GDT are the same as they are in PMTX. The last segment is used to map the CPU structure (defined below) by the gs segment, so that each CPU can access its own CPU structure by the same virtual address gs:0. Each GDT requires only 56 bytes. The 4 KB area at 0x4000 has enough space for a large number of GDTs or CPUs. The (2 KB) IDT at 0x5000 is common to all CPUs. The BSP uses the first GDT at 0x4000 and the initial ipgdir at 0x1000 to turn on paging. Then it sets the stack pointer to initproc[0]'s kstack and calls init() in init.c.

15.5.9.3 Find Number of CPUs in SMP

(3). init() calls findmp() (in mp.c) to get the SMP configuration of the system. It first finds the FPS and MP table. Then it scans the MP table to determine the number of CPU entries and returns the number of CPUs in the system. Each CPU is represented by a cpu structure.

```
struct cpu{
   struct cpu *cpu; // pointer to this cpu struct  at gs:0
   PROC *proc;      // current running PROC on CPU at gs:4
   int   id;        // CPU ID number               at gs:8, etc.
   u32  *pgdir;     // pgdir pointer of CPU
   u32  *pgtable;   // pgtable pointer
   u32  *gdt;       // CPU GDT pointer
   PROC *initial;   // initial PROC pointer of CPU
   // reserved fields for SMP_MTX, not used in SMP_PMTX
   int   srQ;       // spinlock during task switch on CPU
   int   sw;        // switch process flag of CPU
   int   rq;        // last ready queue used by CPU
} cpus[NCPU];       // NCPU=16, each of size=40 bytes
```

15.5.9.4 Initialize CPU Structures

(4). In SMP_PMTX, each CPU has an initial PROC, defined as IPROC initproc[NCPU], each with a statically allocated kstack. All the initial PROCs run in kernel mode only. They share the same pgdir and page tables of CPU0. Assume that the system has 512 MB physical memory and the SMP_MTX kernel occupies the lowest 4 MB. Then the memory area above 4 MB is free. As in PMTX, we shall build the kernel mode kpgdir at 0x106000 and the page tables at 4 MB. The cpu structures, their pgtables and gs segments are initialized by the following code segment.

```
// GDT_ADDR=0x80104000, KPG_DIR=0x80106000, KPG_TABLE=0x80400000
struct cpu *cp;
u32 myaddr,*mygdt,*gdt=GDT_ADDR; // NCPU GDTs at 0x80104000
for (int i=0; i<ncpu; i++){      // ncpu is the actual number of CPUs
    cp = &cpus[i];               // cp points to CPU structure
    cp->cpu = cp;                // cpu structure pointer
    cp->proc = 0;                // current running PROC pointer
    cp->id = i;                  // CPU ID
    cp->pgdir   = KPG_DIR;       // kpgdir at 0x80106000
    cp->pgtable = KPG_TABLE;     // pgtables begin at 4MB
    cp->initial = &initproc[i];  // CPU initial/idle PROC
    if (i==0) kpgtable();        // only CPU0 builds pgtables
    // fix up CPU's gs segment in GDT
    cp->gdt = gdt + i*2*NSEG;    // NSEG=7, pointer to this CPU's GDT
    mygdt  = (u32*)cp->gdt;  // fix up gs segment address in CPU GDT
    myaddr = (u32)&cp->cpu << 8;
    mygdt[13] |= (myaddr >> 24); // gs segment is 6th entry in GDT
    mygdt[12] |= (myaddr << 8);  // index 12=low, 13=high 4-byte
}
```

15.5.9.5 CPU Kernel Mode Page Table

(5). The function kpgtable() sets up the kernel mode kpgdir and page tables, which are shared by all the initial processes of the CPUs. In addition to regular pages, it also fills the last eight entries of kpgdir and their page tables to create an identity mapping of the address range [FE000000, 4 GB], which allows the CPUs to access IOAPIC and APIC register addresses above 0xFE000000.

```
// Assume 512MB; NPGTABLE=512/4=128 pgdir entries
void kpgtable()
{ u32 i, j, pte;
  u32 *pgdir   = cpus[0].pgdir;     // CPU0's pgdir
  u32 *pgtable = cpus[0].pgtable;   // CPU0's pgtable
  memset(pgdir, 0, 4096);           // clear pgdir
  // 128 regular pages for 512MB physical memory
  pte = (u32)0x3;                   // begin PA=0, 0x3=|Kpage|W|P
  for (i=512; i<512+NPGTABLE; i++){ // 128 entries from entry 512
      pgdir[i] = PA(pgtable) + 0x3; // pointing to pgtables
      for (j=0; j<1024; j++){       // pgtabe, 4KB each
          pgtable[j] = pte;
          pte += 4096;              // pte is a u32
      }
      pgtable += 1024;              // pgtable is a u32 pointer
  }
  // LAST 8 pgtables : identity map [0xFE000000, 4GB]
  pte = 0xFE000000 + 0x3;           // begin PA=0xFE000000
  for (i=1016; i<1024; i++){        // pgdir entries 1016 to 1023
      pgdir[i] = PA(pgtable)+0x3;   // pointing to pgtables
      for (j=0; j<1024; j++){       // fill in pgtable 1024 entries
          pgtable[j] = pte;
          pte += 4096;
      }
      pgtable += 1024;
  }
}
```

Figure 15.3 shows the virtual address mapping of the CPUs in SMP_PMTX.

Figure 15.4 shows the memory map of SMP_PMTX, which is similar to that of PMTX. Instead of a single GDT, it has NCPU (16) GDTs, one per CPU beginning at 0x104000. The kernel mode kpgdir and page tables are shared by all the initial PROCs of the CPUs.

Entries	kpgdir: at VA=0x80106000
0-511	empty since all initial/idle PROCs run in Kmode
512-639	128+8 kernel mode page tables begin at PA=4MB 4KB pageTables: map VA=[2GB,2GB+512MB] to PA=[0,512MB]
640-1015	Empty space between 512MB and 0xFE000000
1016-1023	4KB pageTables: identity map [0xFE000000,4GB]

Fig. 15.3 Virtual address mapping of CPUs in SMP

0 to 1MB 1 MB 0x104000 0x105000 0x106000	1 MB real mode memory SMP_PMTX kernel in 1MB to 4MB NCPU(16) GDTs, one per CPU IDT for all CPUs kpgdir for all initial PROCs
4 MB	128 page tables for all initial PROCs
5 MB	1024 PROCESS + 512 THREAD PROC structures
6 MB	unused, for expansion
7 MB	data area of 1024 1KB I/O buffers
8–512 MB	free memory for process Umode images

Fig. 15.4 Memory map of SMP_PMTX

15.5.9.6 Load GDT and Enable Paging

(6). BSP loads GDT with cpus[0].gdt and sets gs = 0x30, which refers to the last segment in the GDT. When an AP starts, it loads its own GDT at cpus[cpuid].gdt and sets gs = 0x30 also. Thus, every CPU can access its CPU structure through the gs segment. In particular, the CPU structure is at gs:0 and the current PROC running on the CPU is at gs:4. To comply with these, we declare (in type.h)

```
extern struct cpu *cpu asm("%gs:0");   // &cpus[cpuid]
extern PROC *running   asm("%gs:4");   // cpus[cpuid].proc
```

The GCC compiler will use gs:0 for cpu, and gs:4 for PROC *running in the kernel's C code. In assembly code, we simply replace them with gs:0 and gs:4, respectively. Then the BSP switches pgdir to cpus[0].pgdir. It can now access all 512 MB physical memory, as well as the IOAPIC and APIC registers above 0xFE000000. When an AP starts, it first uses the ipgdir at 0x101000 to turn on paging. Then it switches pgdir to the same kpgdir and page tables of CPU0, allowing it to access the entire virtual address range directly.

15.5.9.7 Initialize SMP Kernel

(7). BSP: Continue to initialize the kernel. The actions are the same as in PMTX, i.e. initialize kernel data structures, create and run the initial process P0, remap IRQs, install IDT, initialize device drivers and mount the root file system, etc. In SMP_PMTX, the initial PROCs of the CPUs are statically allocated as IPROC initproc[NCPU]. The other NPROC = 1024 process and NTHREAD = 512 thread PROCs are constructed at 5 MB in init(). After initializing the kernel, P0 builds the free page frame list and calls main(ncpu) (in t.c) to create the INIT process P1. If the number of CPUs is 1, P0 brings the system up in UP mode. Otherwise, it calls smp() (in smp.c) to configure the system for SMP operations, which are described next.

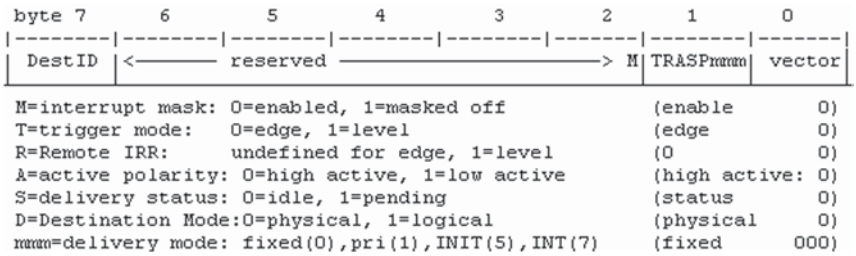


Fig. 15.5 IOAPIC register format

15.5.9.8 Configure IOAPIC to Route Interrupts

(8). Initialize IOAPIC to route interrupts: IOAPIC interrupt registers are set up in ioapic_init(). Figure 15.5 shows the IOAPIC register format.

For better control, we shall use physical delivery mode by processor ID. Therefore, we set each active IOAPIC interrupt register with

DestID=CPUid, M=0, TRASPmmm=0000000, byte0=vector

and mask out all inactive interrupt registers. In SMP, the PIC timer will be disabled. Each CPU uses its own local APIC timer and handles the timer interrupts.

15.5.9.9 Configure APIC and APIC timer

(9). Each CPU calls lapic_init(int cpuid) to configure the local APIC as follows.

- Disable logical interrupt lines: mask out LINT0, LINT1 registers;
- Disable performance register: mask out PCINT (0x340) register;
- Map error interrupts to an exception vector: lapicw(0x370, ERROR_VECTOR);
- Clear error status register: lapicw(0x280, 0);
- Acknowledge any outstanding interrupts: lapicw(0x0B0, 0);
- Configure APIC timer to generate periodic timer interrupts.

Figure 15.6 shows the APIC timer register format, which is configured by lapic_timer().

```
lapic_timer(int vector, u32 int_entry)
 {
   int_install(vector, int_entry);
   set timer register (0x320) to periodic, enabled, with vector
   set initial counter register (0x380) = 0x0010C000
   set current counter register (0x390) = 0x0010C000
   set timer divider register   (0x3E0) = 0x0000000B (divide by 1)
   enable APIC: write 0x010F to spurious interrupt register(0x0F0)
 }
```

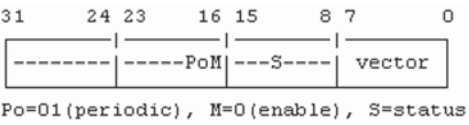


Fig. 15.6 APIC timer register format

APIC timer interrupt vectors of the CPUs are 0x40+cpuid and their interrupt handler entry points are defined as c0inth to c15inth (in ts.s). It is observed that for VMware virtual machines, the APIC timer count 0x0010C00 yields very close to 60 interrupts per second. For other SMP platforms, the APIC timer count may need to be adjusted to better match real time. For instance, the emulated bus frequency of QEMU is much higher than that of VMware. When running SMP MTX under QEMU, the APIC timer count should be increased by a factor of 16 or more.

15.5.9.10 Start up APs

(10). The last action of the BSP is to issue INIT and STARTUP IPIs to start up the APs. It uses the real-mode memory at 0x90000 as a communication area between the BSP and the APs. First, it writes the entry address of APstart() to 0x90000, followed by the initial stack pointers of initproc[i], (i=1 to ncpu-1). Then it executes

```
int go_smp = 1                      // number of active CPUs so far
lapicw(0x0300, 0x00c4500); delay(); // broadcast INIT IPI to APs
lapicw(0x0300, 0x00c4691); delay(); // broadcast STARTUP IPI to APs
while(go_smp < ncpu);               // wait for all APs to be ready
run_task(); // enter scheduling loop to run tasks from readyQueue
```

In the STARTUP IPI, the vector number is 0x91, which corresponds to the loading address of APentry at 0x91000. When an AP starts, it executes the trampoline code APentry at 0x91000 in 16-bit real mode. First, it enters protected mode with 4 GB flat segments and reads the APIC ID register at 0xFEE00020 to get its cpuid number. It uses the ipgdir at 0x101000 to turn on paging. It uses cpuid to get the initial stack pointer at 0x90000+4*cpuid. Then it sets the stack pointer and calls APstart(cpuid) in smp.c.

15.5.9.11 APs Execute Startup Code APstart(cpuid)

(11). The actions of each AP in APstart(cpuid) are as follows.

- Load GDT in cpus[cpuid].gdt, switch pgdir to cpus[cpuid].pgdir;
- Configure local APIC by lapic_init(cpuid);
- Load IDTR with the same IDT at 0x105000 for exception and interrupt processing;
- Set gs=0x30, cpu=&cpus[cpuid] and running=&initproc[cpuid];

- Initialize initproc[cpuid] with pid=2000+cpuid as the initial process running on AP;
- Set GDT.tss to initproc[cpuid].tss as the interrupt stack;
- Configure APIC timer, using the interrupt vector 0x40+cpuid;
- Atomically increment go_smp by 1 to inform BSP that this AP is ready;
- run_task(): enter scheduling loop to run tasks.

(12). When all the APs are ready, the BSP enters scheduling loop to run task also.

After all the above steps (1) to (12), MTX is running in SMP mode. The SMP_PMTX kernel uses sequential UP algorithms adapted for SMP, as described in Sect. **15.5.5.** Since most of the modifications are fairly simple and straightforward, we shall not show them here. The reader may consult the source code of SMP_PMTX for details. As in PMTX, it has only one readyQueue for process scheduling. All CPUs try to run tasks from the same readyQueue. The initial process of each CPU is also the idle process of that CPU. If a CPU finds the readyQueue empty, it runs the idle process, which puts the CPU in halt, waiting for interrupts. The idle processes are special in that they do not enter readyQueue for scheduling and they have no time limits. For this reason, they are assigned the special pid 0 and 2000+cpuid for identification.

15.5.9.12 SMP_PMTX Demonstration System

On the MTX install CD, MTX.images/smp.pmtx is a runnable image of SMP_PMTX. Figure 15.7 show the startup screen of SMP_PMTX under QEMU with eight CPUs. After login, each process displays its pid and the CPU ID it is running on.

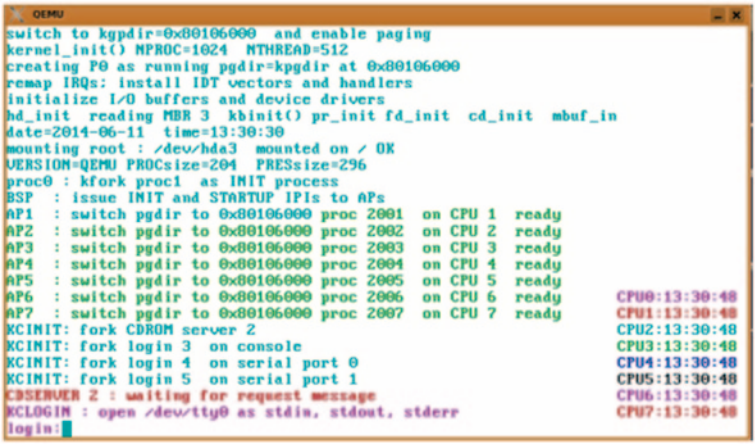


Fig. 15.7 Startup screen of SMP_PMTX

15.6 SMP_MTX using Parallel Algorithms

As a direct extension of PMTX to SMP, the SMP_PMTX kernel is very simple and easy to understand. However, it is not a good SMP system for the following reasons. Since all the kernel data structures have a single copy, processes can only access the same data structure one at a time, which severely limits the concurrency of the system. In particular, because of the single ready queue, process scheduling and switching can only be done by one process at a time, regardless of the number of CPUs, which does not fully utilize the capability of a SMP system. Therefore, simply using fine grained locks to protect UP kernel data structures and adapting UP algorithms for SMP is not the right approach to SMP. In order to truly support SMP, the algorithms must be redesigned to improve concurrency and efficiency. In the following, we shall apply the data partitioning principle of parallel computing (Grama et al., 2003) to design parallel algorithms for SMP. In this scheme, we decompose data structures in a SMP kernel into separate parts to allow parallel operations. If the decomposed parts are logically related, we devise ways to synchronize process executions on the various parts to ensure data consistency. Then we extend SMP_PMTX to the final version of SMP_MTX by using parallel algorithms.

15.6.1 Parallel Algorithms for Process Scheduling

In a UP kernel, it suffices to maintain a single process scheduling queue. In SMP, a single scheduling queue can be a serious bottleneck because it does not allow different CPUs to switch process at the same time, despite the fact that task switching on a CPU involves only the CPU itself. In order to improve concurrency, we shall use parallel algorithms operating on multiple scheduling queues. In SMP_MTX, the parallel process scheduling algorithm is as follows.

1. Multiple process scheduling queues: Instead of a single readyQueue, we define

```
PROC *readyQueue[NCPU];      // one scheduling queue per CPU
```

2. Each CPU is represented by a cpu structure containing a spinlock, srQ, and a switch process flag, sw, which is set when the CPU needs to switch process.
3. Each CPU tries to run tasks from its own readyQueue[cpuid]. If a CPU's readyQueue is empty, it runs a default idle process, which puts the CPU in halt, waiting for interrupts. After an interrupt, it tries to run tasks from the same readyQueue[cpuid] again.
4. When a process calls tswitch() to switch process, it must acquire the CPU's srQ spinlock, which is released by the next running process when it resumes.
5. In order to balance the processing load, ready processes are distributed evenly among the ready queues. Each CPU has a next queue number, rq=cpuid ini-

tially. When a process executing on a CPU makes a process ready, e.g. by fork(), wakeup() or V() operations, it updates the CPU's rq number by rq=(rq+1)% ncpu and enters the ready process into readyQueue[rq].
6. When a new process is created by fork(), it is assigned a base priority of 128. When a process is unblocked from a semaphore queue, it is assigned the highest priority 256, which may set the CPU's switch task flag.
7. Each CPU uses its local APIC timer and handles its own timer interrupts. At each timer tick the timer interrupt handler updates the CPU usage time and time slice of the running process. Periodically, it adjusts the process priorities in the CPU's scheduling queue and sets the CPU's switch process flag if necessary.
8. In SMP, which has more CPUs to run tasks, dynamic process priority is less important than that in UP systems. If desired, process scheduling can be done by a short time slice only.
9. As in UP kernels, process switch takes place only when the running process exits kernel and is about to return to user mode. However, the SMP_MTX kernel supports preemptive process scheduling. Currently, the code (in APIC timer handlers) is commented out in order to avoid excessive process switch overhead in kernel mode. If desired, the reader may activate the code, recompile the system to observe preemptive process scheduling in the SMP_MTX kernel.

15.6.2 Parallel Algorithms for Resource Management

An OS kernel contains many data structures which are crucial to system operations. Examples of such data structures include free PROC list, in-memory minodes and free page frames, etc. When adapting a UP kernel for SMP it suffices to protect each of these data structures by a spinlock. Again, this simple approach has a major drawback in that it does not allow for any concurrency on the same data structure. In order to improve concurrency, we shall decompose such data structures into separate parts and manage them by parallel algorithms.

15.6.2.1 Manage PROC Structures by Parallel Algorithm

In an OS kernel, PROC structures are the focal points of process creation and termination. In a UP kernel, it suffices to maintain all free PROC structures in a single free list. In SMP, we divide free PROCs into separate free lists, each associated with a CPU. This allows processes executing on different CPUs to proceed in parallel during fork and wait for child termination. If a CPU's free PROC list becomes empty, we let it obtain PROCs from other free PROC lists dynamically. In the SMP_MTX kernel, free PROCs are managed by parallel algorithm as follows.

```
Define: PROC *free_proc_list[NCPU];  // free PROC lists, one per CPU
        int   procspin[NCPU]={0};    // spinlocks for PROC lists

PROC *get_proc()  // allocate a free PROC during fork/vfork;
{ int cpuid = cpu->id;               // CPU id
  while(1){
  (1). slock(procspin[cpuid]);       // acquire CPU's spinlock
  (2). If (free_proc_list[cpuid]==0){// if CPU's free list is empty
  (3).    if (refill(cpuid)==0){     // refill CPU's free_proc_list
              sunlock(procspin[cpuid]);
              continue;              // retry
          }
       }
  (4). allocate a PROC *p from free_proc_list[cpuid];
  (5). sunlock(procspin[cpuid]);     // release CPU's spinlock;
  }
  return p;
}
int refill(int cpuid)                // refill a CPU's free PROC list
{  int i, n = 0;
   for (i=0; i<ncpu && i!=cpuid; i++){ // try other CPU's free list
       if (!cslock(procspin[i]))      // if conditional lock fails
           continue;                  // try next free PROC list
       if (freeproclist[i]==0){       // if other free list empty
           sunlock(procspin[i])       // release spinlock
           continue;                  // try next CPU's free list
       }
       remove a PROC from free_proc_list[i];// get a free PROC
       insert into free_proc_list[cpuid];   // add to CPU's list
       n++;
       sunlock(procspin[i]);                // release spinlock
   }
   return n;
}
void put_proc(PROC *p) // release p into CPU's free PROC list
{ int cpuid = cpu->id;
  (1). slock(procspin[cpuid];         // acquire CPU's spinlock
  (2). enter p into free_proc_list[cpuid];
  (3). sunlock(procspin[cpuid]);      // release CPU's spinlock
}
```

The refill() operation tries to get a free PROC from every other free PROC list into the current CPU's free PROC list. Since the process already holds a CPU's spinlock, trying to acquire the spinlock of another CPU may lead to deadlock. So it uses the conditional cslock() instead. If the conditional locking fails, the process backs off to prevent any chance of deadlock. In get_proc(), if after refill the CPU's free PROC list is still empty, it releases the spinlock and retries the algorithm. This prevents self-deadlock on the same CPU.

Note that the PROC of a process may be allocated from the free list of one CPU but released to that of a different CPU. So the algorithm implicitly distributes PROCs among the CPUs. Under normal conditions, free PROCs should be evenly distributed among the CPUs. If a CPU's free list becomes empty, each refill operation may add ncpu-1 free PROCs to a CPU's free list. Since the number of processes can not exceed the total number of PROCs, the refill operation is guaranteed to succeed. Alternatively, we may add a cpuid field to each PROC structure and release a PROC to the same free list form which it is allocated. In that case, the need for refill operations would be greatly reduced. Other ways to further improve concurrency on the PROC structures include

the following. Implement a process family tree to speed up exit and wait operations in kernel. For example, when searching for ZOMBIE child in wait(), a process only needs to lock its own children list, rather than the entire PROC table. Similarly, when a process terminates, it only needs to lock its own children list and that of the INIT process in order to dispose of orphaned children processes. Likewise, implement a process list on the same terminal to speed up signal delivery when an interrupt key is entered, etc.

15.6.2.2 Manage In-memory Inodes by Parallel Algorithm

In an OS kernel, in-memory inodes (minodes) are the focal points of file system operations. To improve concurrency, in SMP_PMTX we divide minodes into separate parts and manage them by parallel algorithms. When the system starts, we assign a fixed number of free minodes to each device and maintain them in a local free minodes list of the device, each protected by a spinlock. This allows processes to execute iget() and iput() on different devices in parallel. If a device's free minodes list becomes empty, we again let it obtain free minodes from other devices dynamically.

15.6.2.3 Manage Page Frames by Parallel Algorithm

In an OS kernel using dynamic paging, free page frames are the focal points of process image management during fork, exec and exit operations. In SMP_MTX, we divide free page frames into separate pfreelists, each associated with a CPU. Then we modify page frame management functions to support parallel operations on different pfreelists. Similar to PROC structures, a process image may be created from the pfreelist of one CPU but released to that of a different CPU, which necessitates the refill operation. For better efficiency, when refilling a pfreelist we try to transfer 1024 page frames at a time from each of the other pfreelists.

15.6.2.4 Manage Other Resources by Parallel Algorithms

We may apply the same data partitioning principle and technique to manage other resources in a SMP kernel. These include bitmaps of file systems, open file table entries, mount table entries, message buffers, etc. In order to keep the system simple, we have not used parallel algorithms for these data structures, which are suitable for programming projects.

15.6.3 Parallel Pipe Algorithm for SMP

In this section, we shall show the design of a parallel pipe algorithm for SMP. For each pipe, we decompose the pipe's data buffer into two separate parts; a read buffer for pipe readers and a write buffer for pipe writers. Then we redesign the pipe algorithm to allow a reader and a writer to execute in parallel on different buffers. Since

the separate buffers are logically the same buffer, we must ensure that read/write of the separate buffers are synchronized properly. The SMP_MTX kernel implements such a parallel pipe algorithm, which is shown below.

```
/********************** Parallel Pipe Algorithm ************************/
Assume: each pipe has a rbuf for pipe readers and a wbuf for pipe writers.
(1). If rbuf has data and wbuf has room, a reader and a writer may proceed in parallel.
(2). If rbuf is empty and wbuf has data, swap the buffers.
(3). If wbuf is full, transfer data to rbuf until rbuf is full or wbuf is empty.

typedef truct pipe{
      char *rbuf, *wbuf;   // palloc() 4KB buffers during creation
      int  rhead = 0;            // read buffer index
      int  rdata = wdata = 0;    // counters
      int  rlock = wlock = 0;    // reader writer spinlocks
      int  nreader, nwriter;     // number of readers and writers
}PIPE;

PIPE pipe[NPIPE];                // NPIPE pipe structures

int rswap(PIPE *p)   // called by pipe reader whenever rbuf is empty
{
  while(1){
    slock(p->wlock);             // acquire wlock
    if (p->wdata > 0){           // if wbuf has data:
       swap(p->rbuf, p->wbuf);   // swap the buffers
       p->rhead = 0;             // reset rbuf index
       p->rdata = p->wdata;      // adjust counters
       p->wdata = 0;
       wakeup(p->wdata);         // wakeup writer
       sunlock(p->wlock);        // release wlock
       return p->rdata;
    }
    if (p->nwriter){   // wbuf empty: if pipe still has writer
       wakeup(p->wdata);         // wakeup writer
      sleep(p->rdata, p->wlock); // sleep on rdata and release wlock
       continue;                 // continue while loop
    }
    return 0;                    // no more writer
  }
}

int wswap(PIPE *p)   // called by pipe writer when wbuf is full
{
  if (!cslock(p->rlock))         // if can't get rlock, back off
     return 0;
  if (p->rdata == 0){            // rbuf empty: swap bufs
     swap(p->rbuf, p->wbuf);
     p->rhead = 0;               // reset read index
     p->rdata = p->wdata;        // adjust counters
     p->wdata = 0;
     wakeup(p->rdata);           // wakeup reader
     sunlock(p->rlock);          // release rlock
     return p->rdata;
  }
  // rbuf not empty: transfer data from wbuf to rbuf
  int count = PSIZE - p->rdata;                  // room in rbuf
  memcpy(p->rbuf, &p->rbuf[p->rhead], p->rdata); // compact rbuf
  memcpy(&p->rbuf[p->rdata], p->wbuf, count);    // transfer data
  memcpy(p->wbuf, &p->wbuf[count], PSIZE-count); // compact wbuf
  p->wdata -= count;             // adjust counters
```

```
  p->rdata = PSIZE;
  p->rhead = 0;
  wakeup(p->rdata);              // wakeup reader
  sunlock(p->rlock);             // release rlock
  return count;
}

int read_pipe(PIPE *p, char *buf, int n) // read_pipe() function
{ int r = 0;
  if (n<=0) return 0;
  while(n){
      slock(p->rlock);           // acquire rlock
      if (p->rdata == 0){        // if rbuf empty
         if (rswap(p) == 0){     // no data and no writer
            sunlock(p->rlock);
            return 0;
         }
      }
      while(p->rdata && n){      // read loop
        *buf++ = p->rbuf[p->rhead++]; // read a byte
         r++; p->rdata--; n--; // adjust counters
      }
      if (n==0 || r){          // has read some data
         sunlock(p->rlock);
         return r;
      }
      sunlock(p->rlock);         // unlock rlock, repeat read loop
  }
}

int write_pipe(PIPE *p, char *buf, int n) // write_pipe function
{ int r = 0;
  if (n<=0) return 0;
  while(n){
     slock(p->wlock);            // acquire wlock
     if (p->nreader == 0){       // no more readers
        sunlock(p->wlock);
        exit(BROKEN_PIPE);       // die by BROKEN PIPE
     }
     while(p->wdata<PSIZE && n){ // write loop
        p->wbuf[p->wdata++] = *buf++; // write a byte
        r++; n--;
     }
     wakeup(p->rdata);           // try to wakeup reader
     if (n<=0){                  // done with writing
        sunlock(p->wlock);
        return r;
     }
     // need space to write but wbuf is full
     if (p->wdata == PSIZE){
        if (wswap(p)==0){        // if did not transfer any data
           wakeup(p->rdata);     // try to wakeup reader
           sleep(p->wdata, p->wlock); //sleep on wdata,release wlock
           continue;
        }
     }
     sunlock(p->wlock);
  }
}
```

In the parallel pipe algorithm, the pipe reader plays the role of an initiator. Whenever the pipe's rbuf is empty, it calls rswap() to refill rbuf with data. In rswap(), it first acquires the wlock to ensure that it can swap the buffers without interference from any writer. It waits only if wbuf is also empty, in which case it will be woken up by the writer when wbuf has data. In contrast, the pipe writer's role is more passive. For better efficiency, the writer calls wswap() only if wbuf is full. While holding the wlock, if the writer tries to acquire the rlock also, it may lead to deadlock with the reader. So it uses conditional locking. If the locking fails, it backs off to prevent any possible deadlock. If the locking succeeds, it has acquired both wlock and rlock. Then it either swaps the buffers or transfers data without interference from the reader. Besides deadlock prevention, it is also worth noting the following. When the reader calls rswap(), it waits for rdata if wbuf is empty. Since the writer calls wswap() only when the wbuf is full, this may cause unnecessary delay of the reader. So it tries to wake up the reader whenever it writes data to wbuf. Since the writer only holds the wlock but not the rlock, its wakeup call may miss the reader if the latter is not yet sleeping. But this causes no harm since the reader will call rswap() when it finds the rbuf empty later.

In terms of parallel programming, pipes represent an extreme case in that the problem can not be parallelized easily. This is because pipe data must be FIFO. If we decompose a pipe's data buffer into more pieces to increase concurrency, additional synchronizations are needed, which may offset the gain in concurrency. However, there are many other problems which can be parallelized more easily. One example is the I/O buffer management problem for block devices.

15.6.4 Parallel I/O Buffer Management Algorithms

In the I/O buffer management problem, buffers are maintained in separate data structures, such as the free list and device lists, etc. These separate data structures lend themselves naturally to parallel operations. The basic principle of parallel buffer management is very simple. Since buffers are maintained in separate device lists, we may protect each device list by a lock and allow processes to access different device lists in parallel. The maximal degree of concurrency of such an algorithm would be the same as the number of devices. If we divide the buffers into more hash queues and protect each hash queue by a lock, the maximal degree of concurrency would be raised to the number of hash queues. Chapter 12 of Bach (Bach 1990) contains such a buffer management algorithm for MP Unix, which is as follows.

15.6.4.1 Unix MP Buffer Management Algorithm

1. Free buffers are maintained in a freelist. Assigned buffers are maintained in hash queues (HQs). An assigned buffer is in a unique HQ and also in the freelist if it is FREE.

2. The freelist has a lock semaphore, each HQ has a lock semaphore and each buffer has a lock semaphore.
3. As in UP Unix, ugetblk/ubrelse are the core of the MP buffer management algorithm.

```
BUFFER *ugetblk(dev,blk) // return a locked buffer for exclusive use
{
 while(buffer not found){
    P(HQ);                        // lock HQ of bp=(dev,blk)
    if (bp in HQ){
       if (!CP(bp)){              // if can't lock bp
          V(HQ);                  // release HQ lock
          P(bp);                  // wait for bp
          if (!CP(HQ)){           // if lock HQ fails
             V(bp); continue;     // retry the algorithm
          }
          else if (bp changed){
             V(bp);
             V(HQ); continue;     // retry
          }
       }
       // locked bp, which must be in freelist; HQ still locked
       while(!CP(freelist));      // spin lock freelist
         remove bp from freelist;
       V(freelist);               // unlock freelist
       V(HQ);                     // release HQ lock
       return bp;
    }
    /******** Part 2: buffer not in HQ ***********/
}
```

Although (Bach 1990) does not show the case when the needed buffer is not in the uffer cache, it is fairly easy to deduce its logic. With the HQ still locked, the process must

- Try to get a free buffer from freelist.
- If freelist is empty, unlock HQ, wait for free buffer; then retry the algorithm;
- If freelist is not empty but can not lock any free buffer, unlock HQ and retry the
- algorithm;
- If locked free buffer is dirty, ASYNC write it out and try freelist again;
- If locked free buffer is in a different hq, try to lock hq in order to get the buffer into HQ but must avoid any possible deadlock, retry the algorithm if necessary;
- With a locked free buffer in HQ, assign it to (dev, blk);
- Release the HQ lock and return the assigned buffer.

Some specific comments about the MP Unix algorithm follow.

1. Based on the single freelist data structure, it is essentially the same UP algorithm in Chap. 3 of (Bach 1990) adapted for MP. As in the UP algorithm, every released buffer is up for grabs, which may be reassigned, thereby reducing the buffer's cache effect.
2. In ugetblk(), when a process finds a needed buffer but the buffer is already locked, it releases the HQ lock to allow concurrency and waits for the buffer.

Once it releases the HQ lock, race condition may occur. For example, the buffer may be released by a process (on another CPU) as FREE, grabbed by yet another process and assigned to a different disk block, etc. When the process eventually acquires the buffer's semaphore lock, the buffer may be changed. If so, the process must give up the buffer and retry the algorithm again. This not only reduces the buffer's cache effect but also causes extra retry loops.

3. Because of the individual buffer lock, each buffer can only be used by one process at a time. In a real system, processes often access files in read-only mode, e.g. when search directories for pathnames, load executable files for execution and read file contents, etc. In a SMP kernel, such processes should be able to read from the same buffer concurrently.
4. The apparent degree of concurrency is equal to the number of HQs but the minimal degree of concurrency is only one due to the single freelist bottleneck.

15.6.4.2 New Parallel I/O Buffer Management Algorithm for SMP

In this section, we shall show a new parallel I/O buffer management algorithm, which does not have the above shortcomings. First, we assume that each allocated buffer is for exclusive use. Then we extend the algorithm to allow concurrent readers on the same buffer. To do these, we modify the buffer data structures as follows.

1. Instead of a single freelist, we distribute free buffers into the local freelists of different hash queues (HQs). Assume n = NBUF/NHQ > 1. Each HQ's local freelist begins with n free buffers.
2. Each HQ has a lock semaphore = 1, each buffer has a lock semaphore rw = 1 and a pcount, which is the number of processes on the buffer. Since each local freelist is associated with a HQ, there is no need for separate local freelist locks. They are the same as the HQ locks.
3. When read/write a disk block, the behavior of every process is

```
BUFFER *bp = mgetblk(dev,blk);     // return a reserved buffer
             P(bp.rw);             // lock buffer
               use the buffer bp; // use the buffer means R|W
             V(bp.rw);             // unlock buffer
             mbrelse(bp);          // release the buffer
```

The following shows the parallel algorithm of mgetblk() and mbrelse().

```
BUFFER *mgetblk(dev,blk)       // return a reserved bp
{
 while(1){
   P(HQ);                      // lock HQ of bp=(dev,blk)
   if (bp in HQ){
      bp.pcount++;             // inc pcount to reserve the buffer
      if (bp.pcount==1)        // bp is free in HQ's local freelist
         remove bp from local freelist;  // bp in HQ, no need to lock
      V(HQ);                   // unlock HQ
      return bp;
   }
   Part2: // bp NOT in cache; HQ still locked
   if (empty(HQ.freelist)||(HQ.buffers all assigned && HQ.nbuf<NBUF))
      brefill(HQ);
   if (empty(HQ.freelist)){ // if after refill freelist still empty
      V(HQ); continue;      // unlock HQ, retry mgetblk()
   }
   // HQ's local freelist not empty
   allocate a bp from HQ's local freelist;
   if (bp dirty){
      P(bp.rw); awrite(bp); // write bp out ASYNC
      goto Part2;           // try next buffer in local freelist
   }
   assign bp to (dev,blk);
   bp.pcount = 1;      // also, set bp data invalid, not dirty, etc.
   V(HQ);
   return bp;
 }
}
int brefill(HQ) //refill HQ's freelist with free bufs from other hq's
{  int n = 0;
   for each hash queue hq != HQ do{  // try every other hq
       if (!CP(hq))                  // if lock hq fails
          continue;                  // try next hq
       if (empty(hq freelist){       // if hq's freelist empty
          V(hq);                     // unlock hq
          continue;                  // try next hq
       }
       while (!empty(hq's freelist)){// locked hq has free buffers
          remove a bp from hq's freelist;
          if (bp dirty){
             P(bp.rw); awrite(bp);   // lock bp and write out ASYNC
             continue;
          }
          insert bp into HQ's local freelist;
          n++; break;
       }
       V(hq);                         // unlock hq
   }
   return n ;
}
void mbrelse(bp)                    // release a buffer
{
   while(!CP(HQ);                   // spin lock HQ
   if (--bp.pcount==0)              // if last user on bp
      release bp into (tail of) HQ's local freelist
   V(HQ)                            // unlock HQ
}
```

The brefill() operation of free buffers is similar to that of free PROCs and minodes, except that it must write out dirty buffers. Note that mgetblk() may call brefill() even if HQ's local freelist is not empty. This allows a HQ to keep its assigned buffers for as long as possible to improve the cache effect. Under normal conditions, free buffers should be evenly distributed among the local freelists. Each refill operation may add HQ-1 free buffers to a local freelist. Next, we show that the new algorithm is correct.

1. Buffer uniqueness: This is because every buffer is created inside a HQ critical region.
2. No race conditions: All operations are performed inside the critical regions of HQs. In particular, the pcount of every buffer is manipulated only inside the HQ critical region. This allows a process to return a reserved buffer even before locking it, which separates buffer allocation from buffer usage. Since each process does P(bp.rw) on an allocated buffer, each buffer is used exclusively.
3. Free of deadlocks: The only possible deadlock is in the refill operation, in which a process holding a HQ lock tries to lock another hq. Since the process backs off if it can not lock the other hq, deadlock cannot occur. The scheme is in fact overly conservative. In order to prevent potential deadlocks due to cross locking, it suffices to let only one side back off. The reader may try to improve the scheme by ranking the HQs.
4. Improved Cache effect: The pcount of each buffer acts as a reservation flag. Once a buffer is reserved, it is guaranteed to exist and do not change. A buffer is released as free only if its pcount is 0, i.e. when all processes have finished using it. This should enhance the buffer's cache effect.
5. If buffers are not always busy, the mgetblk() algorithm is guaranteed to terminate. A process retries only if after refill the local freelist is still empty. If buffers are not always busy, any process executing mgetblk() must eventually succeed.

The parallel buffer management algorithm eliminates the single freelist bottleneck. It raises the minimal degree of concurrency to greater than one. If we assume that each device has an I/O queue, the minimal degree of concurrency would be the same as the number of devices. Since I/O operations are needed only if a requested buffer is not in the buffer cache, the actual degree of concurrency should be higher. Assuming a buffer hit ratio of 50%, the minimal degree of concurrency is at least 50% of the number of HQs. In contrast, in the Unix MP algorithm, every buffer transaction must access the single freelist. The minimal degree of concurrency is only 1 regardless of the number of HQs.

Next, we extend the algorithm to support concurrent readers. For each buffer, we add the lock semaphores r=1, w=1 and a readers counter, which is the number of concurrent readers on the buffer. Define the code segments as shown in Fig. 15.8, which divide the classical reader-writer algorithm into separate parts. Then we rewrite reader and writer processes as shown in Fig. 15.9.

In readerEntry(), if the buffer's data are invalid, only the first reader issues start_io() and waits for I/O completion. Meanwhile, all other readers, which follow the first reader but before any writer, are blocked at P(bp.rw). When the buffer's data

```
readerEntry(bp)
{
   P(bp.rw);
   P(bp.r);
   if (++bp.readers==1){
       P(bp.w);
       if (bp data invalid){
          start_io(bp);
          P(bp.iodone);
       }
   }
   V(bp.r);
   V(bp.rw);
}
```

```
writerEntry(bp)
{
   P(bp.rw);
   P(bp.w);
}
```

```
readerExit(bp)
{
   P(bp.r);
   If (--bp.readers==0)
      V(bp.w);
   V(bp.r);
}
```

```
writerExit(bp)
{
   V(bp.w);
   V(bp.rw);
}
```

Fig. 15.8 Decomposed reader/writer code

```
    Reader Process:
bp = mgetblk(dev,blk);
readerEntry(bp);
 // concurrent read from bp
readerExit(bp);
mbrelse(bp);
```

```
    Writer Process:
bp = mgetblk(dev,blk);
writerEntry(bp);
 // exclusive write to bp;
writerExit(bp);
mbrelse(bp);
```

Fig. 15.9 Reader/writer processes

are ready, the interrupt handler unblocks the first reader. Each reader unblocks a trailing reader, if any. This allows all readers in a batch to read from the same buffer concurrently. The rw semaphore also prevents reader/writer starvation since all processes access the buffer in FIFO order.

In a real system, processes often write only partial blocks. To accommodate this, we can modify the writer process to let it get a buffer as writer but use the buffer to read in data first. As in the Unix MP algorithm, if buffers are always busy, starvation for free buffer is still possible. Although this probably would never occur in a real system, an ideal algorithm should be perfect no matter how small the chances are. The reader may try to extend the algorithm to prevent starvation for free buffers.

The parallel buffer management algorithm assumes that all write operations are delay writes. When a delay write operation completes, the interrupt handler must release the buffer. As pointed out before, an interrupt handler cannot sleep or become blocked. On the other hand, it must lock the hash queue in order to release delay-write buffers. There are two possible ways to resolve this dilemma. The first one is to use a spinlock, which is nonblocking. The second one is to create a pseudo

process to release delay-write buffers. A pseudo-process is logically an extension of the interrupt handler but it runs as a process ahead of all regular processes. Define a semaphore work = 0 and let the pseudo process wait for work by P(work). When releasing a delay-write buffer the interrupt handler simply V(work) to unblock the pseudo-process. The parallel buffer management algorithm has been tested in a simulated SMP system (Wang 2002). The test results show that it indeed performs better than the Unix algorithm in both efficiency and cache performance.

15.6.4.3 I/O Buffer Management in SMP_MTX

The SMP_MTX kernel uses the above parallel algorithm for I/O buffer management but with some refinements to improve efficiency. It is implemented as follows.

1. Define NHQ = PRIME, NBUF = 1024, where PRIME is a prime number used to implement the simple hash function: int hash(dev, blk){ return (dev*blk) % PRIME; }
2. Each HQ is assigned nbuf = NBUF/NHQ free buffers initially.
3. refill(HQ): insert buffers with dev = INVALID(-1) to the front of HQ's local freelist. Used buffers are released to the end of HQ's local freelist to honor the LRU principle.
4. BUFFER *getblk(dev, blk) returns a locked buffer by the buffer's rw semaphore, which eliminates the P(rw) line in both readerEntry() and writerEntry().
5. Modify bread(), bwrite(), awrite() and brelse() with an additional parameter

mode = EXCLUSIVE(0) or READONLY(1) or READWRITE(2),

which classifies the buffer usage as

5.1. EXCLUSIVE: If a buffer is used by only a single process, it suffices to lock the buffer's rw semaphore. For example, when writing to a new disk block the buffer is only used by the current process and there is no need to read in data. It only needs to get a locked buffer for the disk block, write to the buffer and release it for delay-write. In this case, the operations are

bp=getblk(dev,blk); write to buffer; bwrite(bp,EXCLUSIVE);

Similarly, during system initialization, shut-down and sync operations, etc. data blocks may be read in first for possible modifications but the buffers are also used exclusively by a single process. In these cases, the operations are

bp=bread(dev, blk, EXCLUSIVE); use buffer; brelse(bp, EXCLUSIVE);

5.2. READONLY: When traverse directories to resolve pathnames, load INODE into memory and read file contents, etc. disk blocks are read in but never modified. Such buffers should allow concurrent readers. In these cases, the operations are

bp=bread(dev,blk, READONLY); concurrent reads; brelse(bp, READONLY);

5.3. READWRITE: When a process writes, it typically reads the disk block into a buffer, modifies the buffer and releases it for delay-write. If the buffer is not modified, it should be released without incurring a delay-write. In these cases, the operations are

```
bp=bread(dev,blk, READWRITE);
     use buffer;  // may only use the buffer contents but not modify it
if (buffer modified)  bwrite(bp, READWRITE);  //  for delay-write
else                  brelse(bp, READWRITE);  //  no delay-write
```

6. DIRTY buffers are eventually written out by awrite(bp, READWRITE);

When the SMP_MTX kernel starts, it can only access 8 MB of physical memory initially. The NBUF = 1024 I/O buffers are defined in the kernel but their data areas are allocated in the memory area of 7 MB. Each HQ is allocated nbuf = NBUF/NHQ free buffers initially. While a HQ still has unassigned buffers, getblk() continues to use buffers in the HQ's local freelist. Otherwise, it tries to get more free buffers form other HQs dynamically. Since the refill operation would increase the execution time, it is desirable to reduce the number of such operations. We have tested many different combinations of NHQ and nbuf values. The test results indicate that, for a fixed number of NBUF buffers, the algorithm performs better if we keep NHQ small so that nbuf of each HQ is large. The test results also indicate that the hit ratio depends only on the total number of buffers, not on the number of HQs. Since the number of simultaneously executing processes cannot exceed the number of CPUs, there is no need to choose NHQ much larger than NCPU. A large number of HQs may reduce the search time but it also tends to incur more refill operations. In order to maintain a proper balance between concurrency and overhead, for 16 CPUs we may choose NHQ = 17, 19, 23, etc. In SMP_MTX, NHQ is chosen as the Mersenne prime number 2**5 − 1 = 31. It is well known that, for such numbers, an optimizing compiler may generate code for faster modulo operation in the hash function. With this scheme, the buffer hit ratio is around 40 % when the system starts. It increases quickly to over 80 % after executing a few commands and there are virtually no refill operations.

In addition, we also tested the concurrency of the algorithm. In a conread command program, the main process forks many children processes and waits for all the children to finish. Each child process opens the same file for read and reads from the file in 512-byte chunks until it has read the entire file. In the SMP_MTX kernel, we record the number of conflicts on the hash queues. Whenever a process gets blocked on a hash queue, we count it as a conflict. In all the tested cases, the conflict ratio is constantly at 0 %, indicating that the processes are indeed executing in parallel on different HQs. The same program is also used to test concurrent readers. In an OS kernel, when a process reads/writes an opened file (descriptor), it usually locks the file's in-memory minode to ensure that each read or write operation is atomic, but this also prevents any chances of concurrent reads. So we

modify the read function in the SMP_MTX kernel as follows. Instead of locking the mionde, it locks the opened file instance (OFT). This allows concurrent reads on different opened instances of the same file, but not for inherited file descriptors, e.g. by fork(), which share the same OFT. In the conread program, each process opens the same file separately, so they may read concurrently. It is observed that, under normal conditions, the rate of concurrent reads is very low. In order to increase the chances of concurrent readers during testing, we add a small delay loop for the first reader in readerEntry() to simulate reading from a real disk. Under these conditions, the system shows a large number of concurrent readers, which increases as the delay time increases.

15.6.5 Parallel fork-exec Algorithms

In all Unix-like systems, fork and exec are probably the most frequently used system calls. It is therefore worthwhile to speed up these operations in kernel. The sequential fork algorithm for UP kernels consists of four major steps.

```
/************* Sequential fork() Algorithm ******************/
  (1). Create a child process PROC;
  (2). Allocate child user mode image space and copy parent image;
  (3). Copy parent kstack and fix up child's kstack contents;
  (4). Enter child PROC into ready queue; return child pid;
```

Step (2) and step (3) are independent, which can be executed in parallel. So we formulate a parallel fork algorithm as follows.

```
/**************** Parallel pfork() Algorithm ********************/
(1). Call fork1() to create a child PROC;
(2). Create a helper process to run in parallel, which
       copy parent kstack to child kstack;
       fix up child kstack for it to return to user mode;
       notify parent process of completion;
(3). Parent: allocate user mode space for the child and copy image.
(4). Wait for the helper process to finish;
(5). Enter child PROC into ready queue; return child pid;
```

Similarly, we can formulate a parallel exec algorithm.

```
/**************** Parallel pexec() Algorithm ******************/
(1). Fetch command line, verify command is an executable file;
(2). Create a helper process to run in parallel, which
       deallocate old image and fix up main process kstack as in exec;
       notify the main process of completion;
(3). Main process: allocate new image and load image file;
(4). Wait for the helper process to finish;
(5). Return to new image;
```

Since creating and starting up a helper process incur extra overhead, in each of the parallel algorithms we let the helper process perform the less time-consuming

part of the parallelized work. If the helper process finishes before the main process waits, we have achieved a net gain in execution speed. Otherwise, the main process would have to wait for the helper process to finish, which may render the parallel algorithm ineffective. To further improve concurrency, we may create more helpers to share the processing load. For example, in the parallel exec algorithm, we may create another helper process to load one half of the image file, etc.

15.6.6 Parallel Algorithms for File System Functions

File system operations depend heavily on low-level supporting functions, such as management of in-memory inodes and I/O buffers, etc. Although the SMP_MTX kernel already uses parallel algorithms for these functions, it may be possible to further speed up file system operation by using parallel algorithms for the top-level functions. In this section, we investigate this possibility and discuss some of the preliminary test results. For the sake of brevity, we only consider parallel algorithms of mkdir and rmdir.

15.6.6.1 Parallel mkdir-rmdir Algorithms

The sequential mkdir algorithm for UP kernels consists of the following steps.

```
/********** Sequential Algorithm of mkdir **********/
(1). Preliminary checking, e.g. parent DIR exists and new directory
     name does not exist, etc.
(2). Allocate an inode number and a data block number for the new
     directory; Create inode and data block for the new directory.
(3). Enter new name into parent directory, update parent inode.
```

Steps (2) and (3) can be executed in parallel. So we formulate a parallel mkdir algorithm.

```
/************ Parallel pmkdir Algorithm ***************/
(1). Preliminary checking; Allocate ino and bno for new directory;
(2). Create a helper process to run in parallel, which
        create inode and data block for the new directory;
        notify the main process of completion;
(3). Main process: enter new name into parent directory;
(4). Wait for the helper process to finish;
```

Likewise, we can formulate a parallel rmdir algorithm.

```
/************ Parallel prmdir Algorithm ***************/
(1). Preliminary checking, e.g. directory empty and not BUSY;
(2). Create a helper process to run in parallel, which
        deallocate directory data block and inode;
        notify the main process of completion;
(3). Main process: remove dir entry from parent directory;
(4). Wait for the helper process to finish;
```

15.6.7 Performance of Parallel Algorithms

A parallel algorithm may be good in principle but useless if it does not improve the execution speed in a real system. The performance of a parallel algorithm depends on two key factors. First, the time needed to start up the helper process must be short. Second, the synchronization overhead between the main process and the helper process must also be a minimum. In order to test the practicality of the parallel fork-exec and mkdir-rmdir algorithms, we have implemented them in the SMP_MTX kernel and conducted some tests as follows.

1. To minimize the helper process startup time, we set up a few (8) helper PROC structures ready to run in kernel. When creating a helper process we allocate a free helper PROC, initialize it to execute a function with parameters passed in on the kstack, and enter it into the readyQueue of the highest numbered CPU, which is dedicated to run only the helper process.
2. When a blocked process resumes, it may be rescheduled to run on different CPUs, whose current timer counts may differ. So we use the timer of the highest numbered CPU as the time base and set the timer count to 0x1000 (4096 bus cycles) for better resolution.
3. To reduce process synchronization overhead, we let the main process busily wait for a volatile flag variable until it is set by the helper process.
4. Implement the parallel algorithms as separate system calls, pfork-pexec and pmkdir-prmdir.
5. In a user mode program, we use pfork to fork many processes, each executes a different command by pexec. In a similar program, we use sequential fork-exec to do the same. Then we compare their execution time over a series of test runs. The following lists some samples of their comparative running time under QEMU.

```
pfork-pexec :  196608  188416 155648 147456 163840
fork-exec   :  200704  192512 167936 155648 204800
```

The test results show that on average the parallel pfork-pexec algorithms performed slightly better, but the difference is very small, with less than 5% reduction in execution time. The rather miniscule improvement in performance is probably due to the small size of executable files and process images, which limit the effectiveness of the parallel algorithms. We expect the situation to improve for large executable files and process images.

In a similar way, we also tested the parallel pmkdir-prmdir algorithms but got opposite results. The parallel algorithms always ran slower than their sequential counterparts. This is probably due to the following reasons. First, both pmkdir and prmdir have rather lengthy preliminary checking phases, which are inherently sequential. By the time an algorithm has completed the checking phase, most information needed by the parallelizable parts, such as in-memory inodes and data blocks, are already in the I/O buffer cache. This would make the parallelized work

essentially in memory operations. If the parallelized work can be executed in less time than creating and starting up a helper process, using parallel algorithms would not improve the overall execution speed. Second, QEMU supports SMP even on single CPU hosts. Its SMP environment appears to be simulated, in which the virtual CPUs may not be truly executing in parallel. Therefore, starting up a helper process always involves some extra time delay. Further analyses of the parallel pmkdir-prmdir algorithms reveal that, if they complete the checking phases successfully, steps (2) and (3) are also guaranteed to succeed. It is therefore unnecessary for the main process to wait for the helper process to finish. By taking full advantage of parallel executions, the entire operation may be considered as complete as soon as the main process finishes. Alternatively, we may redesign the parallel pmkdir algorithm as follows.

```
/******** Redesigned Parallel mkdir algorithm **********/
(1). Get dev and inode number (pino) of parent directory;
(2). Allocate an ino and bno for the new directory;
(3). Create a helper process, which uses dev, pino, ino, bno to
     construct inode and data block for the new directory;
(4). Main process: if (OK to create new directory)
     {enter new dir entry into parent directory; return 0;}
(5). deallocate ino and bno; return -1 for failed;
```

In the redesigned pmkdir algorithm, after steps (1) and (2), step (3) is guaranteed to succeed but its effect may be cancelled by the main process. In step (4) the main process performs the lengthy checking whether it's OK to create the new directory. If the checking succeeds, it completes the operation. Otherwise, it deallocates ino and bno, which nullifies the effect of the helper process. In either case, it is also unnecessary to wait for the helper to finish. With these modifications, the parallel algorithms indeed performed better than sequential algorithms. The following lists some samples of their running time under QEMU.

```
pmkdir-prmdir:  81920 77824 86016 57344 49152
mkdir-rmdir   :  86016 94208 90112 77824 94208
```

The test results show that we can make the parallel algorithms run faster only under very special conditions. Since it is unreasonable to use a dedicated CPU to run only the helper processes, we have to schedule them as ordinary processes in the SMP_MTX kernel. Under these conditions, pfork-pexec performs as well as fork-exec, but the sequential mkdir-rmdir algorithms are still faster. These seem to suggest that, in order to realize the full potential of parallel algorithms, we may need special hardware support, e.g. equip each main processor with dedicated co-processors which can be set up to execute the helper code directly, bypassing the usual process setup and scheduling in the SMP kernel. Unfortunately, this is beyond the capability of current SMP-compliant systems.

In summary, we believe that a SMP system should use parallel algorithms to improve both concurrency and efficiency. In SMP_MTX, we use parallel algorithms for process scheduling, resource management, pipes and I/O buffer management.

In order to keep the system simple, we only pointed out the principle and demonstrated some of the parallelization techniques but did not try to apply them to all data structures in the SMP_MTX kernel. Although our initial attempt of using parallel algorithms for fork-exec and mkdir-rmdir did not achieve the intended goal, we believe that the effort is worthwhile as it paves the way for further research. Unlike sequential algorithms, which can not be improved, parallel algorithms still have room for improvements. The algorithms could be redesigned and the implementation technique refined to improve their performance.

15.6.8 SMP_MTX Demonstration System

SMP_MTX is the final version of the MTX operating system. It uses parallel algorithms for SMP. The internal organization and startup sequence of SMP_MTX are identical to that of SMP_PMTX. So we shall not repeat them here. On the MTX install CD, MTX.images/smp is a runnable image of SMP_MTX. Figure 15.10 shows its startup screen with 16 CPUs. The reader may test the system as follows. Login in as root, password = 12345.

- Run commands as usual. Each process shows its pid and the CPU ID it is running on.
- Run pipe commands, e.g. cat f | grep line. The commands will run on different CPUs.
- Run thread commands, e.g. matrix, qsort, etc. The threads will run on different CPUs.
- Run segfault, divide and itimer commands to test exception and signal processing.

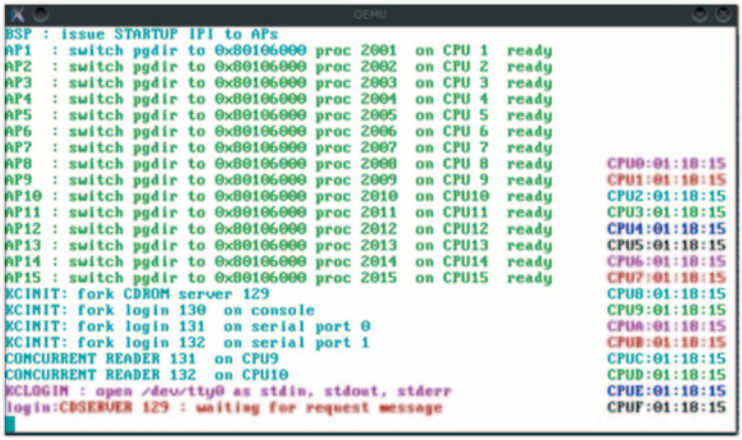


Fig. 15.10 SMP_MTX startup screen

SMP_MTX supports two different versions of file systems: fs.rw and fs.norw. The former supports concurrent readers but the latter does not, so it is much simpler. When compiled with fs.rw, the system usually shows a few concurrent readers during startup. This is because several login processes are trying to load and execute the same login file. Due to their experimental nature, parallel fork-exec and mkdir-rmdir are implemented in the SMP_MTX kernel for testing only. They are not used in regular command processing.

15.7 SMP and Real-time Operating Systems

A real-time operating system (Dietrich and Walker 2015) is an OS intended for real-time applications. Real-time applications usually have two very stringent timing requirements: fast response time and guaranteed completion time. Specifically, a real-time OS must be able to

- respond to interrupt requests of real-time events within a very short time limit.
- complete a requested service within a certain time limit, known as the task deadline.

If a system can always meet these critical timing requirements, it is called a hard real-time system. If it can only meet the requirements most of the time but not always, it is called a soft real-time system. In order to meet these critical timing requirements, a real-time OS is usually designed with the following capabilities.

- Minimum interrupt response latency and task switch time: A real-time OS kernel must not mask out interrupts for long periods of time. Code used to implement critical regions and task switching must be a minimum. All critical regions must be as short as possible.
- An advanced task scheduler: The scheduler must support preemptive task scheduling with a suitable deadline-based scheduling algorithm, such as the Earliest Deadline First (EDF) (Liu and Layland 1973) algorithm. Preemptive scheduling allows higher priority tasks to preempty lower priority tasks at any time. It is a necessary, though not sufficient, condition for real-time operations. Without preemptive scheduling, it would be impossible to meet task deadlines.
- To ensure fast response, tasks in a real-time OS usually do not have a separate user mode. All tasks run in the same address space of the OS kernel.

Due to their unique requirements and objectives, the design and implementation of real-time OS differ from that of general purpose OS. However, they also have many things in common, especially in the area of process synchronization, such as critical regions, protection of data structures to support concurrent executions, prevention of deadlocks and race conditions, etc. Therefore, some of the same design principles and implementation techniques of SMP kernels are also applicable to real-time OS kernels.

15.7.1 SMP_MTX for Real-time Processing

The SMP_MTX kernel supports preemptive task scheduling. We may adapt the SMP_MTX kernel for real-time operations as follows. Divide the processes into two classes; real-time and ordinary. Assign a fixed high priority, e.g. 256, to real-time processes, and lower priorities to ordinary processes. Modify the task scheduler to run real-time processes first with a short time slice by round-robin until there are no runnable real-time processes. Then run ordinary processes, which are scheduled by dynamic process priorities. Whenever a real-time process becomes ready, it preempties any ordinary process. Since the system only guarantees a quick start and a fair share of CPU time for real-time processes but not their completion deadlines, it is therefore only a soft real-time OS. This is the same approach used in (Dietrich and Walke) for real-time processing in Linux.

15.8 SMP in 16-bit Real Mode

Intel's MP Specification defines SMP-compliant systems as PCs with x486 and above processors, which seems to imply that SMP is intended for protected mode only. However, a PC in real mode also has the following capabilities.

- The x86 CPU in 16-bit real mode supports atomic instructions, such as xchg and locked inc/dec, etc. which are essential to SMP.
- If needed, IRQ0 to IRQ15 can also be remapped in real mode.
- In SMP, all the CPUs start in real mode. It should be easy to integrate the APs into a real mode kernel.
- Using BIOS INT15-87, a PC in real mode can read/write IOAPIC and APIC registers in the address range above 0xFEC00000.

Based on these, we have tried to extend the real mode RMTX to SMP. The result is a qualified success. In this section, we describe the adaptation of RMTX to SMP_RMTX and discuss some of the problems and possible remedies.

15.8.1 Real-mode SMP_RMTX startup sequence

1. During booting, CPU0 is the boot processor (BSP), which executes the booter code. As in RMTX, the real mode SMP_RMTX begins execution from the assembly code ts.s, which calls main() in t.c. While in main(), it first calls init() to initialize the kernel, create and run the initial process P0, which calls kfork("/bin/init") to create the INIT process P1 and enter it into the readyQueue of CPU0.
2. P0 calls findmp() to get the number of CPUs in the system. It scans the MP table and returns the number of CPUs in the system. If the number of CPUs is 1, it

falls back to UP mode. Otherwise, it calls smp() to configure the PC for SMP operations.

3. Read/write IOAPIC and APIC Registers: In real mode, we use BIOS INT15-87 to read/write high memory. The MTX kernel is loaded at 0x10000. We use the long word at 0x0F000 as an intermediate data area to read/write APIC registers by the functions

```
u32 apic_read(u8 apic_reg)            // read an APIC register
int  apic_write(u32 w, u8 apic_reg)   // write to an APIC register.
```

The function apic_read() sets up a GDT with source address = 0xFEE00000 + apic_reg and destination address = 0x0000F000. It issues an INT15-87 to read the 32-bit APIC register to 0x0F000 as the return value. Similarly for apic_write(), which writes a long word to 0x0F000 and then issues an INT15-87 to write it to an APIC register. By changing the high memory address to 0xFEC00000 and 0xFEC00010, these functions can also read/write IOAPIC registers.

4. Enable BSP's local APIC and start up APs: During booting, the boot processor is set up by BIOS to receive all interrupts from the 8259 PICs. For SMP in real mode, we do not remap the IRQs and configure the IOAPIC to route interrupts. Instead, we let the BSP handle all the PIC interrupts as in real mode. This helps reduce the needed changes in the RMTX kernel. When the BSP boots up, we enable its local APIC and broadcast INIT and STARTUP IPIs to other APs. When an AP starts, it begins to execute the trampoline code, which must be at a 4 KB page boundary in real mode memory corresponding to the vector number (0 × 11) in the STARTUP IPI. To comply with this, we append the following code segment to the assembly file ts.s. Since the SMP_RMTX kernel is loaded at 0x10000, the added code segment is at the 4 KB aligned page 0x11000. Instead of the actual assembly code, we only show the algorithm of APentry.

```
     .org 4096      ! aligned to 4KB boundary at 0x11000
     .globl _initproc,_APstart,_APspin,_APvmgdt,_APstack
(1). Set DS to MTX kernel DS
(2). Acquire APspin spinlock to ensure one AP at a time
(3). Set SS=DS, sp=APstack as a temporary stack
(4). Use APvmgdt to read APIC ID register at 0xFEE00020
(5). Get CPUID and store it in ES register
(6). Release the APspin spinlock
(7). Use CPUID to set sp to high end of initproc[CPUID]
(8). jmpi _APstart, 0x1000  ! jmp to APstart() in kernel
```

Prior to activating the APs, the BSP initializes the global APvmgdt for reading APIC ID register at 0xFEE00020 by BIOS INT15-87. When an AP starts, it first sets DS to the data segment of the SMP_RMTX kernel. Then it reads the APIC ID register to get its CPUID and stores it in the ES register. It uses initproc[CPUID]'s ksatck to enter APstart() in the RMTX kernel. The algorithm of APstart() is

```
int APstart()
{
      int cpuid = getes(); // ES of CPU contains the CPU ID number
 (a). printf("============= CPU #%d starts ============\n", id);
      let run[cpuid] point to AP's initial PROC initproc[cpuid];
      initialize initproc[cpuid].pid with a unique PID, 123+cpuid;
 (b). configure APIC timer with vector=0x40 + cpuid;
 (c). enable local APIC;
 (d). inform BSP that this AP is ready;
 (e). enter scheduling loop to run tasks from AP's ready queue
}
```

5. Local APIC timer of APs: In the real mode SMP system, we need the AP's timer for the following reason. Since CPU0 receives and handles all the PIC interrupts, the APs do not have any interrupts. In order to run tasks, each AP must be able to examine its ready queue to do task scheduling. There are several possible ways to do this, such as
   - By IPI: CPU0 may issue IPIs to inform other APs to start an action.
   - By shared memory: the APs can monitor some memory contents that are changed by CPU0, but this requires polling by each AP.
   - By a local timer, which interrupts periodically, allowing each AP to look for work by itself.

Among these, a local timer is the simplest to implement. If a CPU finds no work to do, it runs an idle process with interrupts enabled. A timer interrupt will cause it to get up to handle the interrupt and then try to run task again.

6. After setting up the local timer, each AP increments a global variable go_smp by 1 to inform the BSP that the AP is ready. Then it enters the scheduler to run tasks.
7. BSP waits until all the APs are ready. Then it enters the scheduler to run tasks also.

After all the above steps, RMTX is running SMP in 16-bit real mode. As in SMP_MTX, each CPU tries to run tasks from its own ready queue. Ready processes are distributed among the ready queues to balance the processing load of the CPUs.

15.8.2 SMP_RMTX Kernel

In order to support SMP, the RMTX kernel must be modified. The following describes the changes that are specific to the real mode RMTX kernel.

1. CPU ID and Running PROCs: In 16-bit real mode, we let the ES register carry the CPU ID number. In order to identify the PROCs running on different CPUs, we define PROC *run[NCPU] and let run[i] point to the PROC that's currently executing on CPUi. Then we define running as

```
#define running run[cpuid()]
```

This allows the running symbol to be used in the kernel's C code without any change. In assembly code, we simply use ES as an index to access the PROC running on the CPU.

2. Changes to the RMTX kernel for SMP
   2.1. tswitch() in ts.s file: In SMP, there may be several processes executing in parallel. When a PROC running on a CPU calls tswitch() to switch process, we must know the calling PROC in order to save its context. So we modify tswitch() to tswitch(running), passing as parameter the running PROC pointer. After saving its context, the process calls scheduler(), which returns a pointer to the next running PROC. Similar to SMP in protected mode, during task switch the current running process must acquire the CPU's spinlock, which is released by the next running PROC when it resumes.
   2.2. Interrupt entry and exit routines: CPU0 handles all the PIC interrupts. The interrupt entry and exit routines do not need any change for CPU0. Since PROCs running on APs also do syscalls and handle local timer interrupts, both the INTH macro and ireturn code are modified slightly by using the ES register to identify the PROC running on a CPU.
   2.3. Other Changes to the MTX Kernel: The above changes are specific to the real-mode RMTX kernel. Others changes to the RMTX kernel are exactly the same as those in protected mode. Therefore, we shall not repeat them here.
3. Due to the 64 KB code segment limit, the real mode SMP MTX kernel does not have enough room to use parallel algorithms. So it uses modified UP algorithms protected by spinlocks. Also, process scheduling is by time slice only.

15.8.3 SMP_RMTX Demonstration System

On the MTX install CD, SMP_RMTX is SMP MTX in 16-bit real mode. It runs on VMware virtual machines with multiple CPUs. Figure 15.11 shows the startup screen of SMP_RMTX on VMware with four CPUs.

15.8.4 Limitations and Future Work

The real mode SMP MTX seems to work but there are also some minor problems. We invite interested readers to explore these issues further.

1. In the system, CPU0 handles all the PIC interrupts. Each AP only handles its local APIC timer interrupts. Strictly speaking, it is not a SMP system in the true

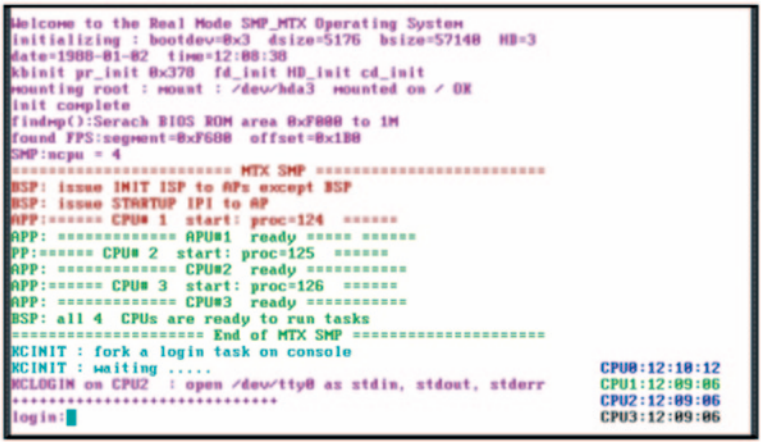


Fig. 15.11 Startup screen of SMP_RMTX on VMware

sense. In order to let the APs handle other interrupts, we should configure the IOAPIC to route interrupts to different CPUs. It is unclear whether this is possible in real mode.

2. Initially, the system had a puzzling problem of losing EOIs. After running for a few minutes, all the APIC timers would stop, causing the system either to hang or fall back to UP mode. The problem was eventually traced to the BIOS INT15-87 routine. The SMP_RMTX kernel relies on the ES register for CPU identification but INT15-87 also uses ES. Although we save ES before calling INT15-87 and restore ES afterward, it seems that the INT15-87 routine enables interrupts as soon as it finishes. If another APIC timer interrupt occurs before ES is restores, the interrupt handler would fail, resulting in lost EOIs. So we try to use CLI to disable interrupts immediately after calling INT15-87. In spite of this, interrupts still occur, although very infrequently, in the time gap between the end of INT15-87 routine and the CLI instruction. Since we can not control the behavior of the BIOS INT15-87 routine, we have to use large APIC timer counter values and also check ES in the interrupt handler entry code. If the ES content is not a valid CPU ID, it must be due to an APIC timer interrupt. In that case, we simply issue an EOI but print an EOI_alert message. The system works normally only after these patches. It can now run for days without losing any EOI. It may be possible to avoid this problem by not using the BIOS routine. For instance, we may switch the CPU to protected mode, write to APIC register and switch back to real mode. But then the question is: if we switch the CPU to protected mode, we could run SMP in protected mode, so why bother to switch it back?
3. The system runs quite well on VMware, most recently on VMware-player 6.0.2 under Slackware Linux 14.1. It should also run on multicore real PCs, but this is

unconfirmed since I can not find any multicore PC that still supports IDE drives. However, it does not run on QEMU. It seems that the APs in QEMU's SMP environment do not respond to any IPI if the BSP is in real mode, so the system just hangs.

Despite these minor problems, the SMP_RMTX system demonstrates that it is indeed possible to do SMP in 16-bit real mode. To the best of my knowledge, it is probably the only 16-bit real mode SMP system in existence. Whether it is useful or not remains to be seen.

15.9 Summary of PMTX and SMP_MTX

This section presents a comprehensive description of the PMTX and SMP_MTX systems.

15.9.1 PMTX and SMP_MTX Source File Tree

The source directories of all versions of MTX contain the following file tree.

```
MTX_VERSION
      |-- SETUP : boot.s, setup.s, apentry.s
      |-- type.h, include.h, Makefile and mk script
      |-- kernel   : kernel source files
      |-- fs       : file system files
      |-- driver   : device driver files
      |-- USER     : commands and user mode programs
```

SETUP	boot.s contains the signatures 'RR' for real mode and 'PP' for protected mode.
setup.s	is for transition from 16-bit real mode to 32-bit protected mode.
apentry.s	is the startup trampoline code of the APs in SMP.
type.h	MTX kernel data structure types.
include.h	constants and function prototypes.
Makefile	top-level Makefile for recompiling MTX by the make utility.
mk	sh script for recompile MTX and install bootable image to HD partition.

15.9.2 PMTX and SMP_MTX Kernel Files

```
---------- Kernel: Process Management Part ----------
type.h     : kernel data structure types, such as PROC, resources, etc.
ts.s       : tswitch, interrupt mask, spinlocks, interrupt handler entry/exit code, etc.
io.c       :  kernel I/O functions; printf(), inter-segment copy in real mode, etc..
mtxlib     :  kernel library functions; memset, memcpy and string operations.
---------- for MTX in protected mode ----------
entry.s    : 32-bit protect mode entry code
init.c     : paging and CPU initialization
traps.s    : low-level exception handler tables in assembly
trapc.c    : high-level exception and interrupt handlers in C
---------- for SMP in protected mode ----------
mp.h     : SMP FP and MP structure types
mp.c       : FP and MP scanning functions
smp.h      : SMP types and structures
smp.c      : SMP configuration and startup code
---------- Common to all MTX ----------
queue.c    : enqueue, dequue, printQueue functions
wait.c     : ksleep, kwakeup, kwait, kexit functions
loader.c   : flat binary and ELF executable image loader
fork.c     :  kfork, fork, vfork functions
exec.c     : kexec function
threads.c : threads and mutex functions
pipe.c     :  pipe creation and read/write functions
mes.c      : message passing: send/recv functions
signal.c   : signals and signal processing
syscall.c :  syscall routing table and simple syscall functions
t.c        : main entry, initialization, parts of process scheduler
---------- Device Drivers ----------
vid.c      : console display driver
timer.c    : timer and timer service functions (for UP MTX)
pv.c       : semaphore operations
kbd.c      : console keyboard driver
pr.c       : parallel printer driver
serial.c   : serial ports driver
fd.c       : floppy disk driver
hd.c       : IDE hard disk driver
atapi.c    : ATAPI CDROM driver
---------- File system ----------
fs         : implementation of a simple EXT2 file system
-----------------------------------------------------------
```

PMTX and SMP_MTX are implemented mostly in C, with less than 3 % of assembly code. The total number of line count in the MTX kernel is approximately 14000 for PMTX and 18000 for SMP_MTX.

15.9.3 Process Management in PMTX and SMP_MTX

1. PROC Structure

Each process or thread is represented by a PROC structure consisting of three parts.

- fields for process management,
- a pointer to a process resource structure,
- kernel mode stack pointer to a dynamically allocated page for kstack.

In protected mode, the PROC structure is

```
typedef struct proc{
        struct  proc *next;// next PROC pointer
        int    *ksp;       // saved kstack pointer
        int     inkmode;   // in Kmode counter
        int     pid;       // process ID
        int     ppid;      // parent pid
        int     status;    // process status: FREE|READY|SLEEP, etc.
        int     priority;  // scheduling priority
        int     event;     // event to sleep on
        int     exitValue; // exit status
        int     vforked;   // proc is vforked
        int     time;      // time quantum
        int     cpu;       // CPU usage time in a reschedule interval
        int     pause;     // pause time
        int     type;      // PROCESS|THREAD type
   struct proc *parent;    // pointer to parent PROC
   struct proc *proc;      // process pointer for threads
   struct pres *res;       // process resource structure pointer
   struct semaphore *sem;  // pointer to semaphore if proc is BLOCKed
   struct semaphore wchild;// semaphore for wait/exit using P/V
        TSS tss;           // TSS structure: PROTECTED MODE ONLY
   int* kstack;            // process kernel mode stack pointer
}PROC;
```

In the PROC structure, the next field is used to link the PROCs in various link lists or queues. The ksp field is the saved kernel mode stack pointer of the process. When a process gives up CPU, it saves CPU registers in kstack and saves the stack pointer in ksp. When a process regains CPU, it resumes running from the stack frame pointed by ksp. In protected mode, when an interrupt or exception occurs, the CPU saves CPU registers in the process kernel mode stack. If a process enters kernel from user mode, the saved CPU registers include uss and usp, which are restored by iret when it returns to user mode. The inkmode field is a counter used to keep track of the number of times a process enters Kmode. When a process is created in kernel, its inkmode is set to one. It is decremented by one when a process exits Kmode and incremented by one when it enters Kmode. This allows the kernel to know whether an exception occurred in Umode or Kmode. In addition, it is used to determine whether a process is about to return to user mode, in which case it must handle any pending signals first, and it may also switch process. The fields pid, ppid, priority and status

are obvious. In most large OS, each process is assigned a unique pid from a range of pid numbers. In MTX, we simply use the PROC index as the process pid, which simplifies the kernel code and also makes it easier for discussion. When a process terminates or exits, it must wakeup/unblock the parent. The parent pointer points to the parent PROC, which allows a dying process to find its parent quickly. The event field is the event value when a process goes to sleep. The exitValue field is the exit status of a process. If a process terminates normally by exit(value) syscall, the low byte is the exit value. If it terminates by a signal, the high byte is the signal number. This allows the parent process to extract the exit status of a ZOMBIE child to determine whether it terminated normally or abnormally. The time field is the maximum run time quantum of a process and cpu is its CPU usage time. They are used to compute the dynamic process scheduling priority. The pause field is for a process to sleep for a number of seconds. In MTX, process and thread PROCs are identical. The type field identifies whether a PROC is a PROCESS or THREAD. PMTX is a uniprocessor (UP) system, which use sleep/wakeup for process management. SMP_MTX uses modified sleep/wakeup only in pipes. It uses semaphores for process management. Device drivers and file system use semaphores for process synchronization. When a process becomes blocked on a semaphore, the sem pointer points to the semaphore. This allows the kernel to unblock a process from a semaphore queue, if necessary. For example, when a process waits for keyboard inputs, it is blocked in the keyboard driver's input semaphore queue. A kill signal or an interrupt key should let the process continue. The sem pointer simplifies the unblocking operation. Each PROC has a res pointer pointing to a resource structure, which is

```
typedef struct pres{
   int    uid, gid;           // user id and group id
   u32    *pgdir,size;        // pgdir and mage size
   u32    *newpgdir, newsize  // used in exec for new image size
   MINODE *cwd;               // Currnt Working Directory pointer
   char   tty[32];            // terminal special file name
   char   name[32];           // program name string
   int    vforked;            // process is vforked flag
   int    tcount;             // number of threads in process
   int    signal;             // 31 signals = bits 1 to 31
   int    sig[NSIG];          // signal handlers
   struct semaphore mlock;    // messageQ lock
   struct semaphore message;  // # of messages
   struct mbuf     *mqueue;   // message queue
   OFT    *fd[NFD];           // open file descriptors
}PRES;
```

The PRES structure contains process specific information. It includes the process uid, gid, pgdir and image size, current working directory, terminal special file name, program name, signal and signal handlers, message queue and file descriptors, etc. In both PMTX and SMP_MTX, PROC and res structures are constructed in the memory between 5 and 6 MB. The first NPROC = 1024 PROCs are for processes and the remaining NTHREAD = 512 PROCs are for threads. Processes and threads are independent execution units. Each process executes in a unique address space but threads in a process execute in the same address space of the process. During

system initialization, each PROCESS PROC is assigned a unique PRES structure pointed by the res pointer. A process is also the main thread of the process. When create a new thread, its proc pointer points to the process PROC and its res pointer points to the same PRES structure of the process. Thus, all threads in a process share the same resources, such as opened file descriptors, signals and messages, etc. Some OS kernels allow individual threads to open files, which are private to the threads. In that case, each PROC must have its own file descriptor array. Similarly for signals and messages, etc. In the PROC structure, kstack is a process kernel mode stack pointer. The kstack of a process is dynamically allocated a 4 KB page only when needed. It is eventually released by the parent in wait(). In protected mode, PROCs are managed as follows.

PMTX: in PMTX, PROCs and resources are defined as

```
IPROC initproc; PROC *proc; PRES *pres;
```

where IPROC and PROC structures are identical except that IPROC contains a statically defined 4 KB kstack. initproc is the initial and idle process of the CPU. When PMTX starts, it can only access the first 8 MB of physical memory via paging. We assume that the kernel occupies the lowest 4 MB. Free PROCs and resources are constructed dynamically in the memory area of 5–6 MB. As in RMTX, free process and thread PROCs are maintained in separate free lists for allocation/deallocation. In PMTX, which is a UP system, there is only one readyQueue for process scheduling. In both PMTX and SMP, the kernel mode stack of each PROC in dynamically allocated a page frame only when needed. When a process terminates, it becomes a ZOMBIE but retains its pgdir and the kstack, which are eventually deallocated by the parent in kwait().

SMP: in SMP, PROC and resource structures are defined as

```
IPROC initproc[NCPU]; PROC *proc; PRES *pres;
```

where initproc[NCPU] are the initial and idle processes of the CPUs. As in PMTX, free PROCs and resources are constructed in the memory area of 5–6 MB but maintained in separate free lists, each associated with a CPU, for allocation/deallocation by parallel algorithms. Each CPU has a separate readyQueue [cpuid] for process scheduling in parallel.

2. ts.s: In protected mode, ts.s is in 32-bit GCC assembly.

PMTX: ts.s contains code for context switch, interrupt handler entry and exit routines, interrupt masking and port I/O, etc. As in real mode, interrupt handlers are installed by the INTH macro. In protected mode, the saved CPU registers include segment selectors.

SMP: to support SMP, ts.s contains the following new functions.

a. slock/sunlock for spinlock operations by the atomic instruction XCHG. It also implements conditional spinlock, which returns 0 if a process can not acquire a spinlock. The SMP kernel uses spinlocks to protect critical regions in which

task switching is either unnecessary or not allowed, e.g. in interrupt handlers. It uses conditional locking and back-off to prevent deadlocks in concurrent algorithms.

b. In SMP, each CPU is represented by a CPU structure, which is mapped to a virtual address by the CPU's gs segment. In the CPU structure, the first three entries contain a pointer to the CPU structure itself, a pointer to the current PROC executing on the CPU and a spinlock for protecting the CPU's scheduling queue. In the SMP kernel, the symbol cpu (gs:0) points to the CPU structure and running (gs:4) points to the process executing on a CPU. When a process calls tswitch() to switch process, it first acquires the CPU's spinlock, which is released by the next running process on the same CPU when it resumes.
c. In SMP, the interrupt/exception entry and exit codes use gs:4 to access the PROC information of the interrupted process.

3. io.c: This file contains kernel I/O functions, such as printf(), which is based on kgetc()/kputc() in the terminal device driver.
4. mtxlib: This file contains precompiled utility functions, such as memset, memcpy and string operations, etc. which are the same for all versions of MTX.
5. entry.s: This file contains the entry code of MTX kernel in protected mode.

PMTX: In PMTX, the kernel is compiled with the starting virtual address 0x80000000 (2 GB) but it runs from the physical address 0x100000 (1 MB). entry.s sets up an initial paging environment to allow the kernel to access the lowest 8 MB physical memory when it starts. To do this, entry.s defines an initial page directory, ipgdir, at 0x101000, two initial page tables at 0x102000, a GDT at 0x104000 and an IDT at 0x105000. Upon entry, entry.s initializes the two page tables to the lowest 8 MB physical memory. It uses the ipgdir to turn on paging, allowing the kernel to access the lowest 8 MB as both real and virtual addresses. Then it forces the CPU to use virtual addresses starting from 0x80000000. It uses initproc's kstack to call init() in init.c to initialize the kernel.

SMP: In SMP, the initial paging environment is identical to that of PMTX, except that entry.s defines NCPU GDTs at 0x104000. Each GDT contains seven segment descriptors; null, kcs, kds, tss, ucs, uds and gs, where the gs segment is used to map the 40-byte CPU structure of each CPU. The 2 KB IDT at 0x105000 is common to all CPUs. The BSP uses the first GDT at 0x104000. It sets the stack pointer to initproc[0].kstack and calls init().

6. init.c: This file contains the initialization code of PMTX and SMP in protected mode.

PMTX: Upon entry to init(), the kernel can only access the lowest 8 MB physical memory through paging. It first initialize the display driver to make printf() work. Then it sets up a new page directory and the associated page tables to expand the virtual address range. Assume 512 MB physical memory and the kernel occupies the lowest 4 MB. The new page directory, kpgdir, is constructed at VA = 0x80106000, in which entries 0–511 are 0's and entries 512–639 point to 128

page tables at 4 MB. Then it switches to the new kpgdir, allowing the kernel to access the entire 512 MB physical memory. The kpgdir will be the page directory of the initial process P0. The pgdir and page tables of other process are dynamically allocated. The last 512 entries of all page directories are identical since the kernel mode address spaces of all processes are the same. Then it calls kernel_init() to initialize kernel data structures. The NPROC (1024) + NTHREAD (512) PROC and resource structures are constructed in the memory area of 5 MB. Free process and thread PROCs are maintained in separate freeList and tfreeList for allocation/deallocation. Since PMTX is a UP system, all kernel resources are maintained in single data structures, which are managed by sequential algorithms. Then it creates and runs the initial process P0, which uses a statically defined PROC structure initproc. P0 continues to initialize the kernel. It remaps IRQs, installs exception and interrupt vectors in the IDT, initializes I/O buffers, device drivers and mounts the root file system. Then it initializes the free page frame list, pfreeList, from 8 to 512 MB for dynamic allocation/deallocation of page frames. Lastly, it calls main() to create and run the INIT process P1.

SMP: The init() function is SMP is slightly more complex. Its actions are as follow.

1. Initialize display driver to make printf() work;
2. Scan the SMP configuration data structures to determine the number of CPUs;
3. Initialize cpu structures: each CPU is represented by a cpus[NCPU] structure at the virtual address gs:0. It contains a pointer to the PROC running on the CPU at gs:4.
4. When start up, each CPU runs an initial PROC, which is also the idle process of the CPU. The initial PROCs are defined statically as initproc[NCPU]. All the initial PROCs run in kernel mode only. They share the same kpgdir and page tables of CPU0, which is the boot processor (BSP). As in PMTX, the BSP constructs the kpgdir at 0x80106000 and the page tables at 4 MB, for the virtual address range [2 GB, 2 GB + 512 MB]. In addition, it also fills the kpgdir's last eight page tables to create an identity mapping of the address range [0xFE000000, 4 GB], allowing the CPUs to access the memory mapped locations of IOAPIC and APIC above 0xFE0000000.
5. After building the kpgdir and page tables, the BSP switches to kpgdir, thereby expanding the virtual address range to [2 GB, 2 GB + 512 MB] and [0xFE000000, 4 GB]. Then it calls kernel_init() to initialize the kernel data structures.
6. kernel_init(): in both PMTX and SMP, PROCs and their resource structures are constructed in the 1 MB memory area from 5–6 MB. In SMP, free PROCs are divided into separate free lists, each associated with a CPU, which are managed by parallel algorithms.
7. After initializing the PROC lists, the BSP uses initproc[0] to create P0 as the initial running process. When the APs start, each AP uses an initproc[cpuid] as the initial process.

8. When kernel_init() returns, P0 remaps the IRQs, installs exception and interrupt vectors in the IDT, initializes I/O buffers, device drivers and mounts the root file system. The data area of (1024) I/O buffers for block devices are at 7 MB. In SMP, the I/O buffers are maintained in hash queues and managed by parallel algorithms.
9. In SMP, free page frames are divided into separate pfreeList[ncpu], each associated with a CPU and protected by a spinlock, for allocation/deallocation in parallel.
10. Then it calls main(ncpu) in t.c to configure the system for SMP operations.

7. traps.s and trapc.c: traps.s contains low-level exception handler tables in assembly. trapc.c implements high-level exception handlers in C. They are the same for all versions of MTX in protected mode.
8. mp.h, mp.c, smp.h, smp.c: These files are for SMP only. The header files contain SMP configuration, IOAPIC and APIC types. mp.c implements the MP table scanning function, which returns the number of CPUs in a SMP system. smp.c contains IOAPIC and APIC functions. It contains code to configure the system for SMP operations and start up the APs. It also contains the C code of APIC timer interrupt handlers and the startup code of the APs.
9. queue.c: This file implements get_proc()/put_proc() and enqueue()/dequeue() functions. get_proc() allocates a free PROC for a new process or thread. put_proc() returns a ZOMBIE PROC to a free PROC list. In PMTX, ready processes and threads are maintained in a single readyQueue by priority. enqueue() enters a PROC into the readyQueue by priority, and dequeue() returns a pointer to the highest priority PROC removed from the readyQueue. In addition, there are also other list/queue manipulation functions, e.g. enter/remove a sleeping process to/from the sleepList, etc. They are implemented in the relevant files for clarity.

SMP: In SMP, free PROC structures are maintained in separate free lists, each associated with a CPU and protected by a spinlock. They are managed by get_proc/put_proc using parallel algorithms.

10. wait.c: This file implements ksleep(), kwakeup(), kwait() and kexit() for process management.

PMTX: In PMTX, the functions are as follows.

1. ksleep()/kwakeup(): A process calls ksleep(event) to go to sleep on an event. An event is just a value which represents the sleep reason. When the awaited event occurs, another process or an interrupt handler calls kwakeup(event), which wakes up all the processes sleeping on the event. To ensure that processes are woken up in order, sleeping processes are maintained in a FIFO sleepList.
2. kwait(): kwait() allows a process to wait for a ZOMBIE child. If a process has children but no ZOMBIE child yet, it sleeps on its own PROC address. When a process terminates, it calls kexit() to becomes a ZOMBIE and wakes up the parent by kwakeup(). As in Unix, orphan processes become children of the INIT process P1, which repeatedly waits for and releases any ZOMBIE children.

3. kexit(): every process calls kexit(exitValue) to terminate. The actions of kexit() are:

   - give away children, if any, to P1;
   - release resources, e.g. free Umode image memory, dispose of cwd, close opened file descriptors and release message buffers in message queue.
   - record exitValue in PROC, become a ZOMBIE, wakeup parent and also P1 if it has sent any children to P1.
   - call tswitch() to give up CPU for the last time.

ZOMBIE PROCs are freed by their parents through kwait(). Orphaned ZOMBIEs are freed by the INIT process P1.

SMP: The conventional sleep and wakeup are unsuited to SMP due to race conditions. In SMP, they are modified to execute in the same critical region of a spinlock. When a process calls the modified psleep(), it completes the sleep operation before releasing the spinlock. The SMP kernel uses the modified psleep/wakeup only in pipes. It uses a PORC.wchild semaphore to synchronize parent and children processes in kwait() and kexit().

11. fork.c: This file contains fork1(), kfork(), fork() and vfork() functions for creating new processes. They are implemented as follows.

    1. kfork():

When the MTX kernel starts, it first initializes the kernel data structures and creates a process P0 as the initial running process, which runs only in kernel mode and has the lowest priority 0. When initialization completes, P0 calls kfork("/bin/init") to create a child process P1. kfork() calls fork1(), which is the common code of creating new processes. fork1() creates a child process ready to run in kernel but without a Umode image. When fork1() returns, kfork() allocates a Umode memory area for P1, loads the/bin/init file as the Umode image and sets up the kstack of P1 for it to resume. Then P0 switches to run P1, which returns to execute/bin/init in Umode. P1 is the only process created by kfork(). After P1 runs, all other processes are created by fork() or vfork() as usual. The startup sequence of MTX is unique. In most other Unix-like systems, the initial image of P1 is a piece of precompiled binary executable code containing an exec("/etc/init") system call. After system initialization, P1 is sent to execute the exec() syscall code in Umode, which changes P1's image to/etc/init. In MTX, when P1 starts to run, it returns to Umode to execute the INIT image directly.

In PMTX, P0 uses the statically allocated PROC structure initproc. In SMP, it uses initproc[0]. In both cases, P0 sets up the kernel mode kpgdir and page tables, which are shared by all the initial processes in SMP. In kfork(), the pgdir and page tables of P1 are allocated dynamically.

2. fork(): As in Unix/Linux, fork() creates a child process with an Umode image identical to that of the parent. If fork() succeeds, the parent returns the child's pid and the child returns 0. It is implemented as follows.

First, fork() calls fork1() to create a child process ready to run in Kmode. The child PROC inherits all the open file descriptors of the parent but without an Umode image. The child PROC is assigned the base priority of 127 and its resume point is set to goUmode, so that when the child is scheduled to run, it returns to Umode immediately. After fork1() returns, fork() allocates a Umode image area for the child of the same size as the parent and copies the parent's Umode image to the child. Then it copies the parent's kstack and fixes up the child's kstack for it to return to Umode. Finally, it returns the child's pid. When the child runs, it returns to its own Umode image with a 0. Because of the image copying, the child's Umode image is identical to that of the parent. The implementation of fork() in MTX is also unique. It only guarantees that the two Umode images are identical. The return paths of the parent and the child processes are very different.

PMTX: In protected mode**,** normal process image size is 4 MB but both PMTX and SMP support different image sizes. In fork(), the pgdir and page tables of the child process are allocated dynamically. The image copying function is modified to copy the page frames of the parent image. The stack frames for the child process to resume are all in the child's kernel mode stack. It consists of stack frame for the child to resume running in kernel, followed by an interrupt stack frame for it to return to user mode.

SMP: In SMP, free PROCs and page frames are maintained in separate free lists, each associated with a CPU, for allocation/deallocation in parallel. In addition, the SMP_MTX kernel also supports parallel fork() and exec() operations, which are implemented in the pforkexec.c file.

3. vfork(): After forking a child, the parent usually waits for the child to terminate and the child immediately does exec() to change its Umode image. In such cases, copying image in fork() would be a waste. For this reason, many OS kernels support vfork(), which forks a child process without copying the parent image. MTX also supports vfork(), which is implemented as follows.

PMTX and SMP: In protected mode**,** vfork() simply lets the child process share the same pgdir, hence the same page tables, with the parent. The resume stack frames of the child process are constructed in the kernel mode stack. A vforked child uses a separate user mode stack frame to return to user mode. The paging hardware supports COW (Copy-On-Write) pages, which can be used to protect shared images, but they are not yet implemented in either PMTX or SMP_MTX.

12. loader.c: This file implements the Umode image loader.

PMTX and SMP: In protected mode, the loader is modified to load either flat binary or ELF executables into the page frames of a process image. The image type is determined by a linker script, ld.script. Most user mode images are generated as statically linked ELF executables. The loader can load ELF executables with separate code, data and bss sections. MTX does not yet support dynamic linking.

13. exec.c: This file implements the kexec() system call. In MTX, the parameter to kexec() is the entire command line of the form "cmd argv1 argv2…argvn". It

uses the first token, cmd, to create the new image, and it passes the entire command line to the new image. Parsing the command line to argc and argv is done in the new image in user mode. As of now, MTX does not support the PATH environment variable. All executable commands are assumed to be in the/bin directory. If the cmd file name does not begin with/, it is assumed in the/bin directory by default.

PMTX and SMP: In protected mode, kexec() supports different image sizes through an optional –m SIZE command-line parameter, in which case it creates a new image of the specified SIZE. A vforked process always creates a new image, thereby detaching it from the shared parent image. In addition, SMP_MTX also supports parallel exec, which is implemented in the pforkexec.c file.

14. Memory Management: In protected mode, memory management is by dynamic paging.

PMTX: Each process has a pgdir and two sets of page tables. In the pgdir, entries 512 and above point to kernel mode page tables, which map the kernel virtual address space beginning from 0x80000000. The low 512 entries (0–511) point to user mode page tables, which map the VA space of the Umode image. A process may call sbrk()/rbrk() to expand/reduce its heap size. Each sbrk() call adds 4 KB to the process image as the new heap space.

SMP: Memory management is identical to that of PMTX, except that free page frames are maintained in separate free lists, each associated with a CPU and protected by a splinlock, for allocation/deallocation by parallel algorithms.

15. threads.c: This file implements threads [Pthreads 2015] support. In the MTX kernel, there are NTHREAD PROCs dedicated to threads. A process PROC is also the main thread of the process. Each process has a thread count, tcount, which is the number of active threads in the process. When a thread is created inside a process, it is allocated a THREAD PROC, which points to the process PROC. Threads in a process share the same resources of the process. They use tjoin and mutex for synchronization.

PMTX and SMP: In protected mode, the resume and interrupt stack frames of a thread are all in its kernel mode stack. When the user mode thread function finishes, it returns to the virtual address 5, which issues an exit(0) to terminate.

16. pipe.c: This file implement pipes. In MTX, pipes are in-memory IPC mechanisms for related processes, i.e. descendants of the same process which created the pipe. They are modeled after the classical producer-consumer problem. Instead of semaphores, they use sleep and wakeup for process synchronization, which are better suited to the semantics of pipes.

PMTX: In PMTX, the pipe structure is

```
typedef struct pipe{
        char  *buf;               // pointer to a 4KB data buffer
        int   head, tail, data, room;
        int   nreader, nwriter;  // number of readers/writers
        int   busy;              // FREE or in use
}PIPE;  PIPE pipe[NPIPE];        // NPIPE = 10
```

A PIPE contains a circular char buffer, buf[PSIZE=4 KB], with head and tail pointers. The pipe control variables are data, room, nreader and nwriter, where data=number of chars in the buffer, room=number of spaces in the buffer, nreader=number of reader processes on the pipe and nwriter=number of writer processes on the pipe. These variables are used for process synchronization during pipe read/write. When PMTX starts, all the pipe structures are initialized as free. When a process creates a pipe by the pipe(int pd[2]) system call, it executes kpipe() in kernel. kpipe() allocates a PIPE structure, initializes the pipe variables and allocates a 4 KB page frame for the pipe's data buffer. Then it allocates two file descriptors, pd[0] and pd[1], which point to the READ_PIPE and WRITE_PIPE OFT instances, which point to the pipe structure. After creating a pipe, a process forks a child process to share the pipe. During fork, the child inherits all the open file descriptors of the parent. If the file descriptor is a pipe, fork increments the reference count of the OFT and also the number of readers or writers in the PIPE structure by one. A process must be either a reader or writer on the same pipe, but not both. Each process must close its unwanted pipe descriptor. close_pipe() deallocates the PIPE structure if there are no more reader and writer. Otherwise, it does the normal closing of the file descriptor and wakes up any waiting reader/writer processes on the pipe. read_pipe() is for reading from a pipe, and write_pipe() is for writing to a pipe. Pipe read/write are synchronized by sleep/wakeup. A reader returns 0 if the pipe has no data and no writer. Otherwise, it reads as much as it needs, up to the pipe size, and returns the number of bytes read. It waits for data only if the pipe has no data but still has writers. After each read operation, the reader wakes up any sleeping writers. A writer process detects a BORKEN_PIPE error and aborts if there are no readers on the pipe. Otherwise, it writes as much as it can to the pipe. It may wait if there is no room and the pipe still has readers. After each write operation, the writer wakes up any sleeping readers.

SMP: The SMP kernel implements a parallel pipe algorithm to improve concurrency**.** The data buffer of each pipe is split into two separate buffers; a read buffer for readers and a write buffer for writers. If the reader buffer has data and the write buffer has room, a reader and a writer can proceed in parallel. Otherwise, they either swap buffers or transfer data from the write buffer to the reader buffer. Reader and writer processes are synchronized by modified sleep/wakeup operations. They use conditional locking and back-off to prevent deadlocks.

17. mes.c: This file implements message passing as a general mechanism for IPC, which allows processes to send/receive messages through the kernel. It is used to implement a CDROM file server, which communicates with client processes by messages to provide iso9660 file system services.

18. signal.c: This file implements signals and signal processing. It contains the following functions, which are the same for all versions of MTX.

    1. kkill() implements the kill(sig, pid) syscall. It delivers a signal, sig, to the target process. To keep things simple, MTX does not enforce kill permissions. If desired, the reader may modify kkill() to enforce permission checking. As in Unix/Linux, MTX in protected mode supports 31 signals, each corresponds to a bit in PROC.res.signal. Delivering a signal i sets the i-th bit in the target PROC's res.signal to 1. In Unix/Linux, if a process is in the "uninterruptible sleep" state, it cannot be woken up by signals. In PMTX, processes only sleep for ZOMBIE children and when reading/writing pipes, so they are interruptible. Therefore, a signal always wakes up the target process if it is in the SLEEP state. If the target process is waiting for I/O, it is blocked on a semaphore in the device driver. Unblocking such a process may confuse the device driver. However, if the process is waiting for terminal inputs, it will be unblocked by signals. Since the unblocked process resumes running inside a device driver, the driver's I/O buffer must also be adjusted to ensure consistency.
    2. ksignal() implements the signal(sig, catcher) syscall. It installs a catcher function as the signal handler for the signal, except for signal number 9, which cannot be changed. When a process enters Kmode via a syscall and when it is about to return to Umode, it checks and handles outstanding signals by calling kpsig().
    3. kpsig() calls cksig(), which resets the signal bit of an outstanding signal and returns the signal number. For each outstanding signal number n, if the process signal handler function, sig[n], is 0, the process calls kexit($n << 8$) to die with an exit status $= n << 8$. If sig[n] is 1, it ignores the signal. Otherwise, kpsig() sets up the process interrupt ustack frame in such a way that, when the process returns to Umode, it first returns to the signal catcher function. When the catcher function finishes, it returns to the place where it lastly entered kernel. To keep the system simple, MTX dose not support signal masking.
    4. In addition to the Control-C key, which generates a SIGINT(2) signal to all processes on a terminal, several user mode programs, e.g. kill, divide, segcatcher, itimer, etc. are used to demonstrate the signal processing capability of PMTX and SMP.

19. syscall.c: This file implements the system call routing table. First, a function pointer table is set up to contain all the syscall function entry addresses.

```
int (*f[ ])() = {getpid, getppid, getpri, ksetpri, getuid, .....};
```

When a process issues a syscall, e.g. syscall(a, b,c, d,e), it enters kernel to execute int80h, which calls kcinth(), which is

```
int kcinth(u32 parameters via kstack)
{
  int a,b,c,d,e,r;
  unlock();                 // handle syscall with interrupts on
  if (running->res->signal) // check and handle outstanding signals
      kpsig();
  // get syscall parameters a,b,c,d,e from ustack, a=syscall number
  r = (*f[a])(b,c,d,e);     // invokde the syscall function
  if (running->res->signal) // check and handle signal again
      kpsig();
  // change saved AX register to r for return value to Umode
  running->priority = 128-running->cpu;  // drop back to Umode priority
}
```

In kcinth(), the process checks and handles signals first. This is because, if the process already has a pending signal, which may cause it to terminate, processing the syscall would be a waste of time. Then it fetches the syscall parameters from the Umode stack, where a is the syscall number. Then it invokes the corresponding kernel function by

r = (*f[a])(b,c,d,e);

When the syscall function returns, it checks and handles signals again. Since syscall is only a special kind of interrupts, the second checking and handling signals are performed in the exit code of interrupt handlers. Each syscall function returns a value r, except kexit(), which never returns. Before returning to goUmode in assembly, it writes r to the saved AX register in the interrupt stack frame as the return value to Umode. In addition, syscall.c also contains simple kernel functions, such as getpid(), kps(), etc. Any additional syscall functions may also be added here.

20. t.c: This file contains the kernel initialization code, the main() function and parts of the process scheduler.

PMTX: In protected mode, pm_entry in entry.s is the entry point. It sets up the initial paging environment and calls init() in init.c. In init(), it sets up a new pgdir and page tables to expand kernel's virtual address space to the entire physical memory. Then it calls kernel_init() in t.c to initialize kernel data structures, creates and runs the initial process P0. P0 sets up the IDT for interrupt and exception processing, initializes I/O buffers, device drivers and mounts the root file system. Then it builds the free page frame list and calls main() to create and run the INIT process P1.

SMP: In SMP, the boot processor (BSP) executes init(), which creates and runs the initial process P0. P0 continues to initialize the kernel and eventually calls main(). In main(), P0 first creates the INIT process P1. Then it calls smp() in smp.c to configure the system for SMP operations and start up the APs. After these, P0 waits until all the APs are ready. Then it enters the scheduler to run tasks. Each AP wakes up to execute the same trampoline code, APstart(cpuid), in which it sets up the AP's execution environment to the SMP kernel, creates and run an initial process. Then it informs the BSP of being ready and enters the scheduler to run tasks

also. In SMP, each CPU has a separate scheduling queue, readyQueue [cpuid], for process scheduling by parallel algorithms.

15.9.4 Process Scheduling in PMTX and SMP

Process scheduling in MTX is based on dynamic priority with time slice. The scheduling policy is aimed to achieve the following goals in order:

- quick response to interactive processes
- fast execution in kernel mode
- fairness to non-interactive processes

Process scheduling in MTX is implemented as follows.

PMTX: The system maintains a priority readyQueue and a global switch process flag, sw_flag, for process scheduling. Each process has a priority, a time slice and a CPU usage time field. Process priorities vary form 0, which is lowest and used only by the initial/idle process P0, to 256, which is the highest. When an ordinary process is created, it is assigned the base priority of 128. Process scheduling is implemented in the following functions.

1. scheduler(), schedule() and reschedule() in t.c file.
   - scheduler(): This function is called in tswitch() to pick the next running process during task switch. If readyQueue is empty, it runs the idle process P0, which does not have any time limit. Otherwise, it selects the next running process from the head of readyQueue, sets its time slice to SLICE = 10 ticks and clears sw_flag to 0.
   - schedule(PROC *p): This function is called from kwakeup() and V() in device drivers and file system for an awakened or unblocked process. It enters p into readyQueue and sets sw_flag to 1 if p has higher priority than the current running process.
   - reschedule(): This function is called when the running process is about to return to user mode. It calls tswitch() to switch process if sw_flag is on.
2. V() in pv.c file: V() is called in device drivers and file system to unblock a process from a semaphore queue. It assigns a high priority of 256 to the unblocked process p and calls schedule(p).
3. Timer interrupt handler in timer.c file: At each timer tick, if the running process is in user mode, it decrements the process time slice by 1 and increments its CPU usage time by 1, with a maximum of 127. When the process time slice reaches 0, it sets a global switch process flag, sw_flag, to 1.
4. At each time interval of N*SLICE ticks ($N >= 2$), the timer interrupt handler recomputes the priority of each process in readyQueue by

$$\text{priority} = \text{priority-CPU time}$$

and reset its CPU time to 0. If a PROC's CPU time was 0, its priority is increased by a value between 1 and SLICE. This allows a process priority to rise if it has not run for a while.

5. Interrupt handler exit code: When a process exits kernel to return to user mode, it drops back to the user mode priority of 128-CPU time and calls resechedule(), which calls tswitch() to switch process if sw_flag is on

SMP: The process scheduling policy is the same as that of PMTX, except that it uses parallel algorithms for process scheduling.

1. Instead of single ready queue, the SMP kernel maintains ncpu ready queues, readyQueue[ncpu], each associated with a CPU and protected by a spinlock in the CPU structure.
2. Each CPU tries to run tasks from its own readyQueue[cpuid]. If the ready queue is empty, it runs the default idle process, which puts the CPU in halt, waiting for interrupts. Then it tries to run tasks from the same ready queue again.
3. Each CPU has a switch process flag, cpu.sw, in the CPU structure.
4. When a process calling tswitch(), it first acquires the CPU's pinlock, cpu→srQ, which is released by the next running process when it resumes on the same CPU.
5. In order to balance the processing load of the CPUs, ready processes are distributed evenly to different ready queues.
6. Each CPU uses the local APIC timer and handles its own timer interrupts.
7. In SMP, which has more CPUs to run tasks in parallel, the role of dynamic process priority is less important than that in UP systems. If desired, process scheduling in SMP can be done by a short time slice only, which is both simple and effective.
8. The SMP kernel is capable of supporting preemptive process scheduling, including process switch in kernel mode. The APIC timer interrupt handlers include code for preemptive process scheduling. It is commented out to avoid excessive process switch overhead in kernel mode.

15.9.5 Device Drivers

In MTX, all device drivers are interrupt-driven, except for the console display driver which does not use interrupts. Device interrupt handler entry points are installed in the assembly code file, ts.s, by INTH macro calls. Each INTH macro call is of the form

```
_xinth: INTH xhandler
```

All device drivers use semaphores for synchronization. Details of device driver design and implementation are discussed in Chap. 10. Device drivers in all versions of MTX are essentially the same, except for the timer, which is different in SMP. In

the following, we only describe the timer and modifications to other device drivers for SMP operations.

1. Timer and Timer Service:

PMTX: Both RMTX and PMTX use the PC's channel 0 timer, which is programmed to generate 60 interrupts per second. During booting, the booter reads the BIOS time-of-day (TOD) and saves it at 0x90000. When a MTX kernel starts, it uses the saved TOD as the real time base. At each second, it displays a wall clock. Timer service is implemented by a timer queue, which is a list of interval timer requests (TQEs). Each TQE has a time field and an action function. When a TQE's time expires, the timer interrupt handler invokes the action function, which typically unblocks the process or sends an SIGALRM(14) signal to the process. Both RMTX and PMTX use the PIC timer interrupts for process scheduling by dynamic priority and time slice.

SMP: In SMP, the PIC timer is disabled. Each CPU uses the local APIC timer and handles its own timer interrupts using the interrupt vector 0x40+CPUID. The APIC timer can support very fine time resolutions. For the sake of simplicity, it is set to generate approximately 60 interrupts per second. The APIC timer interrupt handler entry points, denoted by c0inth to c15inth, are installed by the INTH macro in ts.s as usual. Each timer interrupt entry code calls a handler function in C, all of which call apictimerHandler(CPUID) in smp.c, where the parameter CPUID identifies the CPU. In apictimerHandler(), each CPU maintains its own tick count and a TOD clock. At each second, each CPU calls pr_clock() to display its wall clock on the console screen. Timer service functions, such as FD drive motor on/off control and interval timer requests, are provided by CPU0 only. In addition, each CPU supports process scheduling by dynamic priority and time slice.

2. Modifications to Other Device Drivers in SMP: In a SMP device driver, process and interrupt handler may execute on different CPUs in parallel. If the process and interrupt handler share data structures and control variables, their executions must be serialized to prevent race conditions. In SMP, such device drivers use a spinlock to serialize the executions of process and interrupt handler. While holding the spinlock, if a process has to wait for data or room in the driver's I/O buffer, it must complete the wait operation before releasing the spinlock. To prevent process self-deadlock, a process must release the spinlock before enabling device interrupts. Similarly, an interrupt handler must release the spinlock before issuing EOI.

15.9.6 File System

MTX implements a simple Linux compatible EXT2 file system, which is described in Chap. 12. The file system uses semaphores for process synchronization. In PMTX, all the file system resources, such as in memory inodes, free I/O buffers,

etc. are maintained in single data structures and managed by sequential algorithms. In particular, the I/O buffer management algorithm for block devices is also sequential. In the following, we shall focus on modifications to the file system for SMP operations.

1. Resource Management by Parallel Algorithms

The SMP_MTX kernel uses parallel algorithms for resource management. In a file system, the most heavily contended resource is the in-memory inodes (minodes). To improve concurrency, the minodes are divided into separate lists associated with the devices and managed by parallel algorithms. When the SMP system starts, each device is allocated a fixed number of free minodes. To allocate a free minode for (dev, ino), it tries to allocate a minode from the device's local free list. If the device's local free list is empty, it refills the local free list by transferring minodes from the free lists of other devices. When a minode is no longer needed, it is released to the local free list of the current device for reuse. The same principle and technique can also be used to manage other resources, such as the blocks and inodes bitmaps, etc. but they are not yet implemented in the current SMP kernel.

2. Spinlocks

In SMP, most low-level functions in allocate_deallocate.c and util.c of the file system use spinlocks to protect critical regions. While holding a spinlock, if a process has to wait for a needed resource, it completes the wait operation before releasing the spinlock in order to prevent race conditions.

3. Block Device I/O Buffer Management

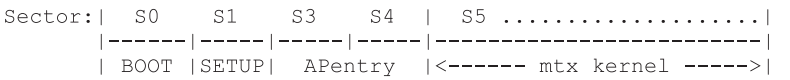
This is the most prominent modification to the file system. In SMP, block device I/O buffers are maintained in hash queues and managed by a parallel buffer management algorithm, which improves both concurrency and the performance of the buffer cache.

4. Parallel fork-exec and mkdir_rmdir Algorithms

SMP_MTX supports fork-exec and mkdir-rmdir by parallel algorithms. The ultimate goal is to replace all sequential algorithms by parallel algorithms in the SMP kernel. As of now, this work is still in an experimental stage. For this reason, parallel fork-exec and mkdir-rmdir functions are included for testing only. Further development is still needed and in progress.

15.9.7 PMTX and SMP Startup and login

In all versions of MTX, a bootable MTX image is composed of 4 contiguous pieces.



where BOOT is for booting the (real mode only) image from a FD, SETUP is for transition from 16-bit real mode to 32-bit protected mode and APentry is the start-up code of the APs in SMP. The last two bytes of the BOOT sector contain the boot signature, which is PP for protected mode kernel or RR for real-mode kernel. When installed to a hard disk, a suitable HD booter is needed to boot up the MTX kernel. The HD booter used in MTX is MBR.ext4 developed in Chap. 3. It can boot MTX, Linux and Windows. During booting, the HD booter loads BOOT+SETUP to 0x90000, APentry to 0x91000 and the MTX kernel to 0x10000. It also writes the BIOS Time-of-Day (TOD) to 0x90000 and the boot partition number to 0x90508 in the BOOT sector. Then it checks the boot signature in the BOOT sector. If the word in RR, it jumps to 0x10000 to start the RMTX kernel in real mode. If the word is PP, it jumps to 0x90200 to run SETUP, which brings up the MTX kernel in protected mode.

15.9.7.1 PMTX Startup Sequence

In PMTX, which uses dynamic paging, the kernel virtual address space is from 2 GB to 2 GB+512 MB, and user mode virtual address space is from 0 to 4 MB. The PMTX kernel is compiled with the starting virtual address 0x80100000 but it runs from the physical address 1 MB. The PMTX kernel must set up the paging environment when it starts. This is done in two steps, which resembles that of multi-stage booting.

1. SETUP: During booting, the booter loads SETUP to 0x90200 and the PMTX kernel to 0x10000. Then it jumps to 0x90200 to run SETUP. SETUP contains an initial GDT, which is

```
setup_gdt: .quad  0x0000000000000000  # null descriptor
           .quad  0x00cF9a000000FFFF  # kcs 00cF 9=PpLS=1001
           .quad  0x00cF92000000FFFF  # kds
```

   The initial GDT defines only two 4 GB kernel code and data segments. SETUP moves the initial GDT to 0x9F000 and loads the GDTR register to point at the initial GDT. Then it enters protected mode, moves the MTX kernel to 1 MB and ljmp to the entry address of the PMTX kernel at 1 MB. The initial GDT is only temporary. It provides the initial 4 GB flat segments for the PMTX kernel to get started.

2. Entry.s: pm_entry in entry.s is the entry point of the PMTX kernel. In order to let the kernel use virtual addresses from 0x80000000 by paging, entry.s defines an initial page directory, ipgdir, two initial page tables, pg0 and pg1, a new GDT, an IDT and a page directory, kpgdir, at offsets from 0x1000 to 0x8000. It fills the two initial page tables, pg0 and pg1, with page frames from 0 to 8 MB to create an identity mapping of VA=[0–8 MB] to PA=[0–8 MB]. Entries 512 and 513 of ipgdir, which point to pg0 and pg1 also, map the virtual address space VA=[2 GB to 2 GB+8 MB] to [0–8 MB]. The new GDT, kgdt,

defines six segments, in which tss is for the TSS in protected mode, ucs and uds are for user mode code and data segments of the current running process. After setting up the initial page tables, entry.s uses ipgdir to turn on paging. It then does a ljmp, using relative PC addressing, to force the CPU to switch to virtual address, allowing the kernel to access the virtual space [0x80000000, 0x80000000 + 8 MB].

3. Load GDTR with the new kgdt descriptor by lgdt kgdt_desc. The new GDT at 0x80104000 defines two kernel mode segments, a TSS and two user mode segments. All code and data segments are flat 4 GB segments, as required by paging.
4. initproc is a statically defined PROC structure for the initial process P0. entry.s sets the stack pointer to the high end of initproc's kstack. Then it calls init() to initialize the PMTX kernel.
5. init.c file: init() first initializes the display driver to make printf() work. At this moment, the kernel's virtual address space is limited to 8 MB. The next step is to expand kernel's virtual address range to the entire available physical memory. Assume 512 MB physical memory and the PMTX kernel occupies the lowest 4 MB. init() builds the new page directory at 0x80106000 and the128 new page tables at 4 MB, 4 MB + 4 KB, etc. which map the kernel's virtual address range to the entire physical memory.
6. kpgtable() creates a kpgdir at 0x80500000, in which entries 512–639 point to 128 page tables, which map VA = [0x80000000, 0x80000000 + 512 MB] to [0, 512 MB]. In PMTX, kpgdir plays two roles. First, it is the pgdir of the initial process P0, which runs in Kmode whenever no other process is runnable. Second, it is the prototype of the page directories of all other processes. Each process has its own page directory and associated page tables. Since the kernel mode address spaces of all processes are the same, the high 512 entries of all page directories are identical. When creating a new process we simply copy the high 512 entries of kpgdir into the process pgdir. The low (0–511) entries of a process pgdir define the user mode page tables of that process. These entries will be set up when the process is created.
7. Switch CR3 to the new kernel page directory, kpgdir. This allows the kernel to access all the physical memory from 0 to 512 MB.
8. kernel_init(): Then init() calls kernel_init() (in t.c) to initialize the MTX kernel data structures, such as free PROC lists, readyQueue and sleepList, etc. In PMTX, only initproc and its resource structure are statically defined. The other NPROC (256) and NTHREAD (128) PROCs are constructed in the memory area of 5 MB. It uses initproc to create the initial process P0, and sets the level-0 interrupt stack in P0's TSS to P0's kstack. Then it calls switch_tss(), which changes the TSS in GDT to P0's TSS and loads the CPU's task state register, TSR, with the new TSS. These make P0's kstack the interrupt/exception stack. The system is now running the initial process P0.
9. The IDT contains 256 8-byte entries, which requires only 2 KB space. It is constructed at 0x105000. init() continues to initialize the IDT and install exception and interrupt vectors in the IDT. Exception handler entry points are in traps.s.

Exception handlers are in trapc.c. I/O interrupt handler entry points are defined in ts.s by INTH macro calls. I/O interrupt handler functions are in the various device drivers. Among the interrupts, vector 0x80 is for system calls.

10. Initialize I/O buffers, device drivers and timer: After setting up the IDT and interrupt vectors, init() initializes I/O buffers. The PMTX kernel has 1024 I/O buffers. Their data areas are allocated at 7 MB. When booting from a hard disk, the hd-booter deposits the boot device number in 0x90508. Before initializing the HD driver, init() extracts the boot device number to set the global variable HD, which is used in both mount_root() and the HD driver. In addition, when initializing the HD driver, P0 also reads the HD's partition table to get the MTX partition's start sector and size. Then it initializes the file system and mounts the boot partition as the root file system.
11. PMTX uses a free page list, pfreeList, for allocation/deallocation of page frames. pfreeList threads all the free page frames from 8 to 512 MB in a link list. Each element of pfreeList contains the address of the next page frame. As usual, the list ends with a 0 pointer. In order for the kernel to access the entries of pfreeList, the link pointers must use virtual addresses of the page frames. When allocating a page frame, the virtual address of the page must be converted to physical address. Conversion between virtual address and physical address are done by the PA/VA macros. With the free page link list, palloc() allocates a free page from pfreeList, and pdealloc(VA(page frame)) inserts a deallocated page frame to pfreeList. In order to use all the available page frames, we add a pfreeTail, which points to the last element of pfreeList, and insert deallocated pages to the tail of pfreeList.
12. Call main() in t.c: In main(), P0 calls kfork("/bin/init") to create the INIT process P1 and switches process to run P1. P1 forks one or more login processes and waits for ZOMBIE children. When the login processes start, PMTX is ready for use. The startup screen of PMTX is shown in Fig. 14.21 in Chap. 14 of the book.

15.9.7.2 SMP_MTX Startup Sequence

SMP is based on PMTX. Its startup sequence is similar to that of PMTX. Therefore, we shall only focus on the SMP part. In SMP, initproc[NCPU] and their resource structures are statically defined. Each CPU uses an initproc[CPUID] as the initial and idle process, which has the special pid = 0 for BSP and 2000 + CPUID for APs. Each initial/idle process runs on a specific CPU. To prevent them from being grabbed by a wrong CPU, they do not enter ready queues for scheduling.

1. SETUP: When the SMP kernel starts, only the BSP (CPU0) is executing. SETUP defines an initial GDT for the BSP to enter protected mode. Then it moves the MTX kernel to 0x100000 (1 MB) and ljmp to pm_entry in entry.s to execute the SMP kernel startup code.

2. entry.s sets up an initial page environment, which allows the kernel to access the lowest 8 MB memory as either physical or virtual addresses. After setting up the initial page tables, entry.s uses the initial ipgdir to turn on paging and forces the BSP to use virtual addresses starting from 0x80000000. So far, the startup actions are identical to that of PMTX. The SMP part begins here. At 0x104000 are NCPU (16) GDTs, one for each CPU, as shown below.

```
kgdt: .rept  NCPU    # NCP=16    #                      Index
      .quad 0x0000000000000000   # null descriptor       0x00
      .quad 0x00cF9a000000FFFF   # kcs 00cF PpLS=1001=9  0x08
      .quad 0x00cF92000000FFFF   # kds                   0x10
      .quad 0x0000000000000000   # tss                   0x18
      .quad 0x00cFFa000000FFFF   # ucs 00cF PpLS=1111=F  0x20
      .quad 0x00cFF2000000FFFF   # uds                   0x28
      .quad 0x80C0920000000027   # CPU struct=40 bytes   0x30
      .endr
gdt_desc:  .word 56-1            # for CPU0's GDT
           .long kgdt            # hard coded GDT address of BSP
      .org 0x5000
idt:  .fill 1024,4,0             # 2KB IDT at 0x5000
      .org 0x8000                # Other kernel code start here
```

The first six segments of each GDT are the same as they are in PMTX. The last segment descriptor is used to map the 40-byte CPU structure by gs, so that each CPU can access its CPU structure by the same virtual addresses in the gs segment. The (2 KB) IDT is common to all CPUs. It will be constructed at 0x105000.The BSP uses the first GDT at 0x104000 and the initial ipgdir at 0x101000 to turn on paging. Then it sets the stack pointer to the high end of initproc[0]'s kstack and calls init().

3. The actions of init() are as follows.

3.1. Find Number of CPUs**:** init() first calls findmp() to get the SMP system configuration information. findsmp() tries to find the FPS and then the MP table. It scans the MP table for the number of processor entries and returns the number of CPUs in the system. Each CPU is represented by a cpu structure (in type.h).

```
struct cpu{
    struct cpu *cpu;    // pointer to this cpu struct
    PROC *proc;         // current running PROC on this CPU
    int   srQ;          // CPU spinlock
    int   sw;           // switch process flag
    int   rq;           // next ready queue to use
    int   id;           // cpu ID number
    u32 *pgdir;         // pgdir pointer of CPU
    u32 *pgtable;       // pgtable pointer
    u64 *gdt;           // per CPU GDT pointer
    PROC *initial;      // initial PROC pointer of CPU
} cpus[NCPU];           // NCPU=16, size = 40 bytes
```

3.2. Initialize CPU structures and gs segments: In SMP, each CPU has an initial PROC, initproc[cpuid]. Since all the initial PROCs run in Kmode only, they share the same pgdir and page tables of CPU0. Assume that the physical memory is 512 MB and the SMP kernel occupies the lowest 4 MB. Then the memory area above 4 MB is free. As in PMTX, the kgdir is at 0x80106000 and the 128 page tables at 4 MB. The cpu structures and their pgdir's and pgtables are initialized by the following code segment.

```
//GDT_ADDR=0x80104000, KPG_DIR=0x80106000, KPG_TABLE=0x80400000
struct cpu *cp;
u32 *gdt = GDT_ADDR;           // NCPU GDTs at 0x80104000 in entry.s
u32 *mygdt, myaddr;
for (i=0; i<ncpu; i++){        // ncpu is the actual number of CPUs
    cp = &cpus[i];
    cp->cpu = &cpus[i];        // cpu structure pointer
    cp->proc = 0;              // current running PROC on this CPU
    cp->id = cp->rq = i;       // CPU ID and initial rq number
    cp->sw = 0;                // switch process flag
    cp->pgdir   = KPG_DIR;     // kpgdir at 0x80106000
    cp->pgtable = KPG_TABLE;   // pgtables begin from 4MB
    cp->initial = &initproc[i];// CPU's initial PROC
    if (i==0)
       kpgtable();             // only CPU0 builds kpgdir, pgtables
    cp->gdt = gdt + i*2*NSEG;  // NSEG=7, pointer to CPU's GDT
    mygdt  = (u32*)cp->gdt;    // fix up gs segment address in GDT
    myaddr = (u32)&cp->cpu << 8;
    mygdt[13] |= (myaddr >> 24); // gs segment is 6th entry in GDT
    mygdt[12] |= (myaddr << 8);  // index 12=low, 13=high 4-byte
}
```

4. Kernel Mode Page Table: The function kpgtable() sets up the kernel mode kpgdir and page tables, which are shared by all the initial processes of the CPUs. In addition to regular pages, it also fills the last eight entries of kpgdir and their page tables to create an identity mapping of the address range [FE000000, 4 GB], allowing the CPUs to access IOAPIC and APIC addresses above 0xFE000000. The virtual address mapping of the CPUs is shown in Fig. 15.3. The memory map of SMP_MTX is shown in Fig. 15.4.
5. Load GDT and Enable Paging: The BSP loads GDT with cpus[0].gdt and sets gs = 0x30, which corresponds to the last segment in the GDT. Similarly, when an AP starts, it loads its own GDT at cpus[cpuid()].gdt and sets gs = 0x30 also. Thus, every CPU can access its own cpu structure and the current PROC running on the CPU using the same virtual address gs:0 and gs:4, respectively. To comply with these, the symbols cpu and running are defined in type.h as

```
extern struct cpu *cpu  asm("%gs:0");  // &cpus[cpuid()]
extern PROC *running    asm("%gs:4");  // cpus[cpuid()].proc
```

The GCC compiler will use gs:0 for cpu, and gs:4 for PROC *running. Therefore, in the SMP kernel's C code, the symbols cpu points to the CPU structure and running points to the PROC that's currently executing on the CPU. In assem-

bly code, they are accessed as gs:0 and gs:4, respectively. Then the BSP switches pgdir to cpus[0].pgdir. The BSP can now access all 512 MB physical memory, as well as the IOAPIC and local APIC registers above 0xFE000000. When an AP starts, it first uses the initial igdt at 0x101000 to turn on paging. Then it switches to the same kernel mode kpgdir and page tables of BSP, allowing each AP to access the entire virtual address range directly.

6. BSP: Continue to initialize the SMP kernel. The actions are the same as in PMTX, i.e. initialize kernel data structures, create and run the initial process P0, remap IRQs, install IDT, initialize device drivers and mount the root file system, etc. As in PMTX, the NPROC=256 process and NTHREAD=128 thread PROCs are constructed at 5 MB. However, instead of a single free list, in SMP free PROCs are divided into separate free lists, each associated with a CPU and protected by a spinlock, for allocation/deallocation by parallel algorithms. After initializing the kernel, P0 sets up the free page frame list and calls main(ncpu) (in t.c) to create the INIT process P1. If the number of CPUs, ncpu, is 1, P0 brings the system up in UP mode. Otherwise, it calls smp() to configure the system for SMP operations by the following steps. The reader may consult Chap. 15 for more details
7. Initialize IOAPIC to route interrupts: Configure APIC and APIC timer:
8. Start up APs: The BSP issues INIT and STARTUP IPIs to start up the APs. The startup code of the APs is loaded at 0x91000. The BSP uses the memory area 0x90000 as a communication area for the APs to start up. First, the BSP writes the entry address of APstart() to 0x90000, followed by the initial stack pointers of initproc[i], (i=1 to ncpu-1). Then it executes

```
int go_smp = 1                          // number of active CPUs
lapicw(0x0300, 0x00c4500); smp_delay(); // broadcat INIT ISP to APs
lapicw(0x0300, 0x00c4691); smp_delay(); // broadcat STARTUP to APs
while(go_smp < ncpu);                   // wait for all APs ready
run_task();   // enter scheduling loop,run tasks from readyQueue[0]
```

When an AP starts, it executes the trampoline code, APentry, at 0x91000 in 16-bit real mode. After entering protected mode with 4 GB flat segments, each AP reads the APIC ID register at 0xFEE00020 to get its CPU ID number. It uses the CPU ID number to fetch the corresponding initial stack pointer deposited by BSP at 0x90000+4*ID. Then, it calls APstart(ID) in smp.c.

9. APs Execute Startup Code APstart(id): Each AP uses its CPU id number to access the cpus[id] structure, which was set up by the BSP in init(). In APstart(), the actions of each AP are as follows.
   - Load its own GDT in cpus[id].gdt, turn on paging and switch pgdir to cpus[id].pgdir;
   - Configure its local APIC by lapic_init(id);
   - Load the same IDT (at 0x105000 in entry.s) for exception and interrupt processing;

- Set gs=0x30, cpu=&cpus[id] and running=&initproc[id];
- Initialize initproc[id] with pid=1000+id as the initial PROC running on AP;
- Set AP's GDT.tss to the initial PROC's tss as the interrupt stack;
- Configure APIC timer, each AP handles its own APIC timer interrupts, using the
- interrupt vector 0x40+CPUID;
- Atomically increment go_smp by 1 to inform BSP that this AP is ready;
- call run_task() to enter scheduling loop to run tasks from readyQueue [cpuid].

10. When all the APs are ready, the BSP enters the scheduling loop to run task also.

After all the above steps, MTX is running in SMP mode. The initial process of each CPU is also the idle process of that CPU. Whenever a CPU finds its ready queue empty, it runs the idle process, which puts the CPU in halt, waiting for interrupts. The idle processes are special in that do not enter ready queues for scheduling and they do not have any time limit. For this reason, they are given the special pid 0 and 2000+cpuid for identification. The startup screen of SMP_MTX for 16 CPUs is shown in Fig. 15.10.

15.9.8 User Interface and System Call Functions

User interface in PMTX and SMP_MTX is identical to that of RMTX in 16-bit real mode, which is covered in Chap. 13. In addition to the system call and user interface functions listed in Chap. 13, PMTX and SMP_MTX have a few additioanl system calls and user mode programs that are apecific to protected mode operations. The reader may consult the ucode.c file in the USER directory and also the syscall.c file in kernel for details.

15.9.9 Recompile PMTX and SMP_MTX

To recompile and install PMTX and SMP_MTX, cd to the source directory and run the mk script as

mk PARTITION qemu|vmware.

Examples:

```
mk 3  qemu     # recompile and install to P3 of a QEMU HD named vdisk
mk 4  vmware   # recompile and install to P4 of a VMware HD named Other.vmdk
```

The reader may examine and modify the dump script file to install to other locations.

Problems

1. When SMP_MTX starts, it can only access 8 MB physical memory via paging initially.
   1. Modify entry.s to let it access 20 MB physical memory via paging.
   2. For each CPU, build its kpgdir in the real mode memory 0x80000-0x90000, which has space for NCPU = 16 kpgdir's.
   3. For each CPU, build its page table in the 1 MB area from 4 to 20 MB.
   4. Let the free page frames begin from 20 MB.
2. Extend SMP_MTX to more than 16 CPUs.
3. In SMP_MTX (also in xv6), the current running PROC pointer of a CPU is at the virtual address gs:4. Instead of using virtual memory, try to maintain the running PROC pointer in a CPU register. Compare the advantages and disadvantages of the two schemes.
4. Modify the UP Unix I/O buffer management algorithm of Chap. 12 for SMP. Implement and test it in SMP_PMTX.
5. Complete Part 2 of the MP Unix buffer management algorithm of Bach. Pay attention to the following cases.
   1. The free buffer list may be empty.
   2. Cross locking: If a process finds a needed buffer, it lock(buffer); lock(freelist); in order to take the buffer out of freelist. Meantime, another process may lock(freelist); lock(buffer) to get a free buffer from freelist;
   3. A locked free buffer is in a different hash queue.
6. Modify the parallel buffer management algorithm to raise its degree of concurrency to the number of hash queues.
7. SMP producer-consumer problem: Try to use signals to inform the producers and consumers of abnormal conditions. For example, when the last producer terminates, send a special SIG_PEND signal to all consumers, which unblocks the consumers if they are waiting for items. Similarly, when the last consumer terminates, send a special SIG_REND signal to all producers, which unblocks the producers if they are waiting for rooms.
   1. How to deliver such special signals?
   2. How does a process handle a special signal?
8. The parallel pipe algorithm allows only one reader and one writer to execute in parallel on the same pipe. In general, a pipe may have multiple readers and writers. Extend the parallel pipe algorithm to support multiple readers and writers on the same pipe.
9. In order to reduce system overhead, in SMP_MTX process switching in kernel mode is disabled. Verify that the SMP_MTX kernel supports preemptive scheduling in kernel mode.
10. It is well known that the problem of optimal task scheduling in MP systems is NP-hard.

1. Consult the literature to find out what NP-hard means and what does it imply?
2. In SMP_MTX, we only try to distribute ready processes to different ready queues evenly. Design other algorithms to balance the processing load of the CPUs.

11. In the SMP_MTX file system, in-memory minodes are maintained in separate devices. The in-memory minodes are not used as a cache memory. After using a minode, it is immediately released by iput(), which removes the minode from the device's minode_list if its refCount reaches zero. Modify iget() and iput() to use the in-memory minodes as a cache memory similar to block device I/O buffers.
12. Modify the SMP_MTX kernel to manage other resources, such as bitmaps and open file tables in the file system, by parallel algorithms.
13. Inter-Processor-Interrupts (IPIs): In SMP, CPUs may interact with one another by IPIs.

    Part 1: Conduct IPI experiments in SMP. Modify the SMP_MTX kernel as follows.

    1. int xhandler(){ printf("CPU %d in IPI handler\n", cpu->id); send _EOI() }
    2. In ts.s: Install xinth and xhandler: xinth: INITH xhandler
    3. In init(): Install a vector, e.g. 0x50 for xinth: int_install(0x50, xinth);
    4. Add a syscall, sendIPI(int cpuid), which sends an IPI with vector 0x50 to the target CPU:

```
. lapicw(0x310, cpuid<<24);      // destination = cpuid
. lapicw(0x300, 0x00004050);     // physical deliver mode, interrupt IPI with vector 0x50
```

    5. Add a user command, sendIPI cpuid, to test the sendIPI syscall. Verify that executing the command causes the target CPU to execute the xhandler function.

    Part 2: Use IPI to implement parallel pfork-pexec and pmkdir-prmdir in the SMP_MTX kernel and compare their performances with sequential fork-exec and mkdir-rmdir algorithms.

14. Analyze other top-level file system algorithms, e.g. creat(), unlink(), etc. in the SMP_MTX kernel for parallelisms and design parallel algorithms for these functions.

15. Modify the real mode SMP_RMTX to do the following:

    1. Remap IRQ0-15 to 0x20-0x2F. Verify that the system still works.
    2. Use local APIC timers for all CPUs.
    3. Try to configure the IOAPIC to route interrupts to different CPUs.
    4. Run SMP_RMTX on other virtual machines.

References

Bach, M.J, “The Design of the Unix operating system”, Prentice Hall, 1990
Cox, A., “An Implementation of Multiprocessor Linux”, 1995
Cox, R., Kaashoek, F., Morris, R. “xv6 a simple, Unix-like teaching operating system, xv6-book@pdos.csail.mit.edu”, Sept. 2011.
Dietrich, S., Walker, D., “The evolution of Real-Time Linux”, http://www.cse.nd.edu/courses/cse60463/www/amatta2.pdf, 2015
Grama, A., Gupta, A., George Karypis, G., Kumar, V., “Introduction to Parallel Computing”, 2nd Edition”, Addison-Wesley, 2003
Intel MultiProcessor Specification, v1.4, 1997
Liu, C.L., Layland, J.W., “Scheduling Algorithm for Multi-programming in a Hard Real-Time Environment,” J. ACM, Vol. 20, pp. 40–61, 1973.
McKusick, M.K., Neville-Neil, G., “The Design and Implementation of the FreeBSD Operating System”, Addison-Wesley, 2004.
Pthreads: https://computing.llnl.gov/tutorials/pthreads/, 2015
Wang, X., “Improved I/O Buffer Management Algorithms for Unix Operating System”, M.S. thesis, EECS, WSU, 2002

Chapter 16
Hybrid Operating Systems

16.1 Monolithic Kernel

In the beginning, computer systems were relatively simple. Most computer systems have only a single CPU, a small amount of memory and a few I/O devices. Therefore, operating systems for early computer systems were also fairly simple and small. Because of their small sizes, most early operating systems, e.g. VAX-11/VMS and Unix, etc. are based on monolithic kernels. A monolithic kernel consists of process management, memory management, device drivers and file systems, all in one integrated unit. A monolithic kernel is a complete kernel since it has all the functionalities of an operating system. As computer systems grew larger and more complex, additional device drivers, new file systems and other functionalities were added to the OS kernel. As a result, OS kernels also grew bigger and much more complex, making them more prone to errors and difficult to maintain. The concept of microkernel (Accetta et al. 1986) was conceived as an alternative approach to kernel design. It is aimed at addressing some of the issues of big kernels.

16.2 Microkernel

A microkernel is the minimum amount of software that can provide the mechanisms needed to implement an operating system. The mechanisms include low-level address space management, process scheduling, fetch and forward interrupts and Inter-Process Communication (IPC). In theory, all other functions of an OS, such as device drivers, memory manager, process manager, file systems and networking, etc. can all be implemented in user space outside of the microkernel. Each of the functions can be implemented as a server process. In a microkernel based OS, user processes do not execute kernel functions directly. Whenever a user process needs to do something, it sends a request message to a server process and waits for reply. The microkernel's IPC routes the request message to a server, which handles the

K. C. Wang, *Design and Implementation of the MTX Operating System,*
DOI 10.1007/978-3-319-17575-1_16

request and sends back a reply along with the results of the requested operation. The microkernel approach has many claimed advantages over monolithic kernels, such as

Because of their small size, microkernels can be made more reliable and portable to different computer hardware platforms.
Microkernel based OS are more modular because all OS functions, e.g. memory manager and file systems, etc. can be implemented as server processes in user space. Microkernel based systems are more reliable and available because errors in one server do not affect other parts of the system. The system can even be configured dynamically by replacing one server with another without recompiling and rebooting the system.
Microkernel based systems can support multiple operating systems.

Microkernel started in the late 80's. Active research and development continued through the 90's. Currently, both research and development seem to have slowed to a standstill. Microkernel based systems can be classified into three generations. The most famous first generation microkernel based system is Mach (Accetta et al. 1986) developed at Carnegie Mellon University. It implemented Unix on top of the Mach microkernel. However, the performances of first generation microkernel based systems were not good. The second generation microkernel based systems include MINIX (Tanenbaum and Woodhull 2006), L4 [L4 2015] and QNX [QNX 2014], etc. The most notable changes occurred in L4. It replaced the inefficient Remote Procedure Calls (RPC) in Mach with new IPCs. By using some clever shortcuts, such as passing message in processor registers, allowing sender-receiver to share address space to avoid message copying, automatic context switching in send-recv and lazy scheduling, etc. L4 was able to reduce the IPC overhead significantly. Despite these efforts, when running Linux on top of a L4 microkernel (Hartig et al. 1997), there is still a 5–10% overhead as compared with native Linux. Third generation microkernels focus mostly on the formal correctness proofs of microkernel, from specification to implementation.

16.3 Comparison of Monolithic and Microkernel Based OS

(1). Portability:

A well designed monolithic kernel can be ported to different computers just as easily as a microkernel. A good example is Linux, which has been ported to many computer platforms, including both CICS and RISC machines.

(2). Device drivers:

In Linux, most device drivers can be compiled as modules, which can be dynamically loaded into the kernel on demand. Although the loaded drivers still run in kernel space, their impact on the kernel size and security is also limited and isolated.

(3). Instead of running multiple OS on top of a microkernel, the current trend is to run multiple OS on virtual machines inside a hosting OS. Whether the hosting OS is microkernel based or monolithic kernel based makes little difference.

(4). Performance

A good example of microkernel based OS is MINIX, which has been a popular educational OS for many years. The microkernel of MINIX3 supports process scheduling, exchange of messages and low-level handling of interrupts. Device drivers are implemented as driver tasks, which have higher priority than regular processes. Process manager, memory manager and file system are implemented as server processes in user space. In MINIX3, when a user process tries to read a file block, it sends a message to the system task in the kernel and waits for a reply. The system task sends the request to the file server, which sends a request to the disk driver task, which starts I/O on the disk and waits for a reply message. When the read operation completes, the disk interrupt handler sends a reply message to the disk driver task, which sends a reply to the file server, which sends a reply to the system task, which finally sends a reply to the user process. As stated in (Tanenbaum and Woodhull 2006), to read a file block the best case requires 4 messages, the worst case requires 11 messages, each requires at least one context switch. For efficiency reasons, data are not transferred through messages. They are copied directly between address spaces. This is the typical way of how a microkernel based OS operates. In contrast, in a monolithic kernel based OS, a similar operation in the best case requires only one system call with no context switch, e.g. when the needed file block is in the buffer cache. Some additional comments on microkernel based OS follow.

(5). In a microkernel based OS, the kernel must schedule tasks/processes for execution. The process manager must manage process creation, process image, signal handling and process termination. The file server must serve process read/write requests by opened file descriptors. Since all the servers are in different address spaces, the process information must be duplicated or split into separate pieces residing in different servers. This adds complexity to the system design and requires more coordination during system operation.

(6). A major claim in support of microkernel is that file systems can be implemented as servers in user space. However, there are also reasons to counter such claims. Let us examine the problem of how to implement a file server in more detail. There are only a few options.

(6).1. As a stateless file server, like the Network File System (NFS) (Sandberg et al. 1985). In this case, the server does not maintain any state information about process file activities. It does not have any notion of opened file descriptors of processes, nor the current read/write position in an opened file. Each read/write request must specify the complete file name as well as the read/write position within the file. Such a file server would be very inefficient and difficult to use in an OS.

(6).2. As a stateless file server, which relies on a separate name server to provide additional information about each read/write request. For example, a stateless file server may use a Domain Name Server (DNS)-like protocol (Comer 1995; Comer

and Stevens 1998) to map a process request in the form of a file descriptor to the file's state information. In a distributed system, relying on a separate name server is a must. In a system with shared memory, using a separate name server is not only hard to justify but also a source of inefficiency.

(6).3. As a stateful file server, which maintains state information of process requests. Since the file server and process manager are in different address spaces, the state information of every process must be split into separate pieces. Each server maintains only a part of the total state information, which means additional synchronization overhead and inefficiency.

(6).4. In a monolithic kernel, operations in the file system are inherently concurrent. For example, if two processes read/write different files, they can proceed independently. In MINIX3, the file server is single-threaded. It serves process requests by multiplexing its executions among different requests. While monolithic kernels strive for improved concurrency, using a single thread to simulate the concurrent operations in a file system is a giant step backward. Some file servers in microkernel based OS do support multi-threads. However, the problem of implementing a multi-threaded file server is exactly the same as that of implementing a file system in a monolithic kernel. In order to support concurrency, both require synchronization on shared data objects in the file system. For example, in a single-threaded file server, buffer management is trivial since there is no competition. In a multi-threaded file server, when a thread finds a buffer in the buffer cache, the buffer may already be locked by another thread. Thus, implementing a multi-threaded file server merely shifts the burden of the problem from the same address space of a monolithic kernel to the address space of a file server.

Based on these comparisons, it is fair to say that microkernel is more suited to distributed environment. Monolithic kernels are more suited to shared memory systems. To impose a microkernel on an OS of a shared memory SMP system would be like "cut off the toes to fit the shoes", a gross misfit.

16.4 Hybrid Operating Systems

MTX is a monolithic kernel based OS because it is designed for systems with shared memory. It is fairly easy to modify the MTX kernel to incorporate some of the microkernel features, making it a hybrid OS.

(1). Device drivers

In a strict microkernel based OS, all device drivers should be implemented in user space. But it is clearly unwise to do so. First, an OS cannot run without a root file system or a timer. To separate the root device and timer drivers from the kernel only slows down the system operation. Second, most PC based OS supports only one user, the sole owner of the PC or laptop, who expects fast console response. To separate the console driver from the kernel again makes little sense. On the other

hand, in a PC based OS a user may never need to login remotely, and may only use the printer and CD/DVD drives occasionally. Such sparsely used device drivers can be implemented outside of the kernel without seriously impairing the system performance. In MTX, such drivers can be implemented as driver tasks, which run in user space with higher priority than ordinary processes. Communication between processes and driver tasks can be implemented by client-server message passing in the MTX kernel.

(2). File System Servers

MTX started in 16-bit real-mode. Due to the 128KB size limit of the RMTX kernel, it only implements an ATAPI driver but not the iso9660 file system in kernel. Support for CDROM file system is implemented in user mode. It has a cdserver command and a cdclient command. When MTX starts up, we may let the INIT process fork a cdserver daemon process. Alternatively, the server process may also be started up later. Then the user may run the cdclient command as a user process, which sends CDROM operations, such as ls, cd, cat, cp, etc. as request messages to the cdserver. The cdserver implements the iso9660 file system in user space. It handles user requests, allowing the user to access the CD/DVD contents. This makes MTX a hybrid OS since it includes features of both microkernel and monolithic kernel. Likewise, we may implement support for other file systems, e.g. DOS, MINIX and NTFS, etc. by server processes in user space. Examples of hybrid kernels are abundant, e.g. Microsoft NT (Solomon 1998), Mac OS X (Singh 2007), etc. Interested reader may consult the references for additional information.

In summary, microkernel and monolithic kernel represent two distinct approaches to OS design. Each has its strength and weakness. As usual, both sides have their strong advocates as well as critics. Occasionally, there has been heated debates as to which approach is better. To this end, it may be appropriate to quote the great pragmatic Chinese leader Deng Xiaoping: “It doesn’t matter if a cat is black or white, so long as it catches mice”. In OS design, a good cat may well be a zebra cat, part black and part white.

Problems

1. Implement other file systems, such a DOS, MINIX and NTFS, by server processes in MTX.

References

Accetta, M. et al., “Mach: A New Kernel Foundation for UNIX Development”, Techical Conference—USENIX, 1986.

Comer, D.E., “Internetworking with TCP/IP: Principles, Protocols, and Architecture, 3/E”, Prentice-Hall, 1995.

Comer, D.E., Stevens, D.L., “Internetworking With TCP/IP: Design, Implementation, and Internals, 3/E”, Prentice-Hall, 1998.

Hartig, H., Hohmuth, M., Liedtke, J., Chhonberg, S., Wolter, J., “The Performance of u-Kernel-Based Systems”, wiki.cs.unm.edu/ssl/lib/exe/fetch.php/papers:l4.pdf, 1997

L4: Microkernel Family: http://en.wikipedia.org/wiki/L4_microkernel_family, 2015
QNX: The QNX Neutrino Microkernel, http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html, 2014
Sandberg, R., Goldberg, D., Kleiman, S., Wash, D., Lyon, B., "Design and Implementation or the Sun Network Filesystem", Sunmicrosystems, Inc. 1985
Singh, A., "Mac OS X Internals", Addison Wesley, 2007
Solomon, D., "Inside Windows NT, Second Edition", Microsoft Press, 1998.
Tanenbaum, A.S., Woodhul, A.S., "Operating Systems, Design and Implementation, third Edition", Prentice Hall, 2006

Appendix

How to Install MTX

MTX Install CD

The MTX install CD, MTXinstallCD.iso, contains the following file tree.

```
MTXinstallCD
    |-- gen          : generate bootable iso image
    |-- isobooter    : CD/DVD booter of the iso-booter in Chapter 3
    |-- vmlinuz      : Linux (2.6.27.7) kernel
    |-- initrd.gz    : initial RAMdisk image of Linux
    |-- initrd-tree  : initrd image files
    |-- hdbooter     : MTX/Linux HD booter of Chapter 3
    |-- MTX.images   : rmtx,pmtx,smp,mtx32.1,mtx32.2, etc.
    |-- MTX.src      : MTX source files
```

The sh script gen is used to create a bootable CDROM image, which uses isobooter as the no-emulation booter. The CDROM image can be burned to a CD/DVD disc or used as a virtual CD directly. During booting, the isobooter first boots up Linux on the RAMdisk image, initrd.gz, which is used to install MTX images form the MTX.images directory.

PC Platforms

MTX can be installed to a real or virtual PC with an IDE hard disk. There are several popular virtual PC platforms, such as QEMU, VMware, VirtualBox and Bochs, etc. MTX has been tested on the following virtual machines.

K. C. Wang, *Design and Implementation of the MTX Operating System*,
DOI 10.1007/ 978-3-319-17575-1

Slackware Linux 14.1 : QEMU and VMware-Player 6.02
Ubuntu Linux 14.4 : QEMU-system-i386

This installation guide covers only QEMU and VMware.

Install and Run MTX Under QEMU

QEMU recognizes many virtual disk formats, including VMware's vmdk. It supports SMP on single CPU hosts, and it provides direct support for serial and parallel ports. Therefore, it is most convenient to run MTX under QEMU. The following examples show how to install and run MTX under QEMU.

Example 1. Use an existing regular virtual disk:
On the MTXinstallCD, vdisk is a flat virtual IDE disk image with 4 partitions. Each partition already contains a runnable version of MTX, which are
Partition 1: rmtx: MTX in 16-bit real mode.
Partition 2: pmtx: MTX in 32-bit protected mode using dynamic paging.
Partition 3: smp: SMP_MTX in 32-bit protected mode.
Partition 4: mtx32.1: MTX in 32-bit protected mode using segmentation.
The simplest way is to run QEMU on the vdisk image directly, as in
qemu –hda vdisk –m 512m –smp 8 –serial mon:stdio
Then boot up and run MTX from the various partitions. To run QEMU with more options, consult QEMU's user manual or the qq script on the MTXinstallCD.

Example 2: Install MTX to an existing virtual disk:
(1). Run QEMU on vdisk but boot up Linux from MTXinstallCD.iso, as in
qemu -hda vdisk –cdrom MTXinstallCD.iso –boot d # assume virtual CD
(2). When Linux comes up, it displays a help menu
MTX Setup: follow the following instructions

Example 3. Create a regular flat virtual disk and install MTX:
Alternatively, the reader may create a flat virtual disk and install MTX as follows.
(1) dd if=/dev/zero of=mydisk bs=1024 count=65536 # creat a file of 64 MB
(2) fdisk -C 8 mydisk # run fdisk on mydisk with 8 cylinders.

partition mydisk; the simplest way is to create only 1 partition.
change the partitions type to 90 for MTX.

(3). Run QEMU on mydisk but boot from MTXinstallCD.iso:
qemu -hda mydisk -cdrom MTXinstallCD.iso -boot d # virtual CD drive
Then install MTX images to the partitions of mydisk as in Example 2.
(4). After installing MTX, run QEMU on mydisk:
qemu -hda mydisk –m −512m –smp 8 -serial mon:stdio
Then boot up and run MTX from the partitions of mydisk.

Install and Run MTX Under VMware

Example 4. Run VMware-Player on an existing virtual machine:

(1). On the MTXinstallCD, the vmware directory contains a VMware-Player virtual machine. Mount the iso CD image and copy the vmware directory to/root, as in

```
mount –o loop MTXinstallCD.iso  /mnt
cp  –av  /mnt/vmware  /root/vmware
umount  /mnt
```

(2). Start VMware. Select "Open a Virtual Machine". From the prompt window, click on the directories and navigate to/root/vmware/Other/Other.vmdk.

(3). Start the virtual machine. Boot up and run MTX from the partitions of the virtual disk.

Example 5. Install MTX to a new VMware virtual machine:

1. Start VMware. Choose "Create a New Virtual Machine". In the following windows, choose "I will install operating system later" and Other for Guest Operating System. Then create a 0.1G IDE hard disk.
2. Configure the virtual machine's CDROM to boot from MTXinstallCD.iso
3. Start the virtual machine to boot up MTX install environment from the virtual CD.
4. Run fdisk to partition the hard disk and change the partition type to 90.
5. Enter install, choose a partition number and a MTX image to install.
6. Quit Linux. Then start the VM and boot up MTX from the installed partition.

When using VMware the reader should beware of the following limitations. VMware uses the vmdk virtual disk format, which can only be mounted by the vmware-mount utility program. VMware's serial ports are files or sockets. A serial port file can receive outputs but can not input. A socket serial port requires a suitable socket interface, such as socat, for I/O. VMware supports SMP but the number of CPUs cannot exceed that of the host machine.

Example 6: Access and convert regular virtual disks:

The main advantage of using a flat virtual disk is that it can be accessed directly under Linux, e.g. run fdisk to partition it. If a virtual disk has been partitioned and formatted as file systems, the partitions can be mounted as loop devices. Even if the partitions are not formatted, they can be mounted as follows.

```
mount -o loop,offset=32256 mydisk /mnt          # 32256=63*512
```

QEMU's qemu-img program can convert virtual disks to different formats, including flat (raw) and vmdk. The reader may consult qemu-img for more detail.

MTX Source Code Files

In the MTXinstallCD.iso CDROM image, the MTX.src directory contains all the source code of this book. It is organized as follows.

MTX.src

|- BOOTERS: booter programs and demonstration systems of Chapter 3.

|- MTX.programs: demonstration sample systems of Chapters 4-10.

|- Source code of mtx32.1, mtx32.2, rmtx, pmtx, smp.mtx, smp.pmx, smp.rmtx.

It is highly recommended that the reader refers to the relevant source code when study the contents of this book.

Index

K. C. Wang, *Design and Implementation of the MTX Operating System*,
DOI 10.1007/ 978-3-319-17575-1