8 General Purpose Embedded Operating Systems

8.1 General Purpose Operating Systems

A General Purpose Operating System (**GPOS**) is a complete OS that supports process management, memory management, I/O devices, file systems and user interface. In a GPOS, processes are created dynamically to perform user commands. For security, each process runs in a private address space that is isolated from other processes and protected by the memory management hardware. When a process completes a specific task, it terminates and releases all its resources to the system for reuse. A GPOS should support a variety of I/O devices, such as keyboard and display for user interface, and mass storage devices. A GPOS must support a file system for saving and retrieving both executable programs and application data. It should also provide a user interface for users to access and use the system conveniently.

8.2 Embedded General Purpose Operating Systems

In the early days, embedded systems were relative simple. An embedded system usually consists of a microcontroller, which is used to monitor a few sensors and generate signals to control a few actuators, such as to turn on LEDs or activates relays to control external devices. For this reason, the control programs of early embedded systems were also very simple. They were written in the form of either a super-loop or event-driven program structure. However, as computing power and demand for multi-functional systems increase, embedded systems have undergone a tremendous leap in both applications and complexity. In order to cope with the ever increasing demands for extra functionalities and the resulting system complexity, traditional approaches to embedded OS design are no longer adequate. Modern embedded systems need more powerful software. Currently, many mobile devices are in fact high-powered computing machines capable of running full-fledged operating systems. A good example is smart phones, which use the ARM core with gig bytes internal memory and multi-gig bytes micro SD card for storage, and run adapted versions of Linux, such as (Android 2016). The current trend in embedded OS design is clearly moving in the direction of developing multi-functional operating systems suitable for the mobile environment. In this chapter, we shall discuss the design and implementation of general purpose operating systems for embedded systems.

8.3 Porting Existing GPOS to Embedded Systems

Instead of designing and implementing a GPOS for embedded systems from scratch, a popular approach to embedded GPOS is to port existing OS to embedded systems. Examples of this approach include porting Linux, FreeBSD, NetBSD and Windows to embedded systems. Among these, porting Linux to embedded systems is especially a common practice. For example, Android (2016) is an OS based on the Linux kernel. It is designed primarily for touch screen mobile devices, such as smart phones and tablets. The ARM based Raspberry PI single board computer runs an adapted version of Debian Linux, called Raspbian (Raspberry PI-2 2016). Similarly, there are also widely publicized works which port FreeBSD (2016) and NetBSD (Sevy 2016) to ARM based systems.

When porting a GPOS to embedded systems, there are two kinds of porting. The first kind can be classified as a procedural oriented porting. In this case, the GPOS kernel is already adapted to the intended platform, such as ARM based

© Springer International Publishing AG 2017
K.C. Wang, *Embedded and Real-Time Operating Systems*,
DOI 10.1007/978-3-319-51517-5_8

systems. The porting work is concerned primarily with how to configure the header files (.h files) and directories in the source code tree of the original GPOS, so that it will compile-link to a new kernel for the target machine architecture. In fact, most reported work of porting Linux to ARM based systems fall into this category. The second kind of porting is to adapt a GPOS designed for a specific architecture, e.g. the Intel x86, to a different architecture, such as the ARM. In this case, the porting work usually requires redesign and, in many cases, completely different implementations of the key components in the original OS kernel to suit the new architecture. Obviously, the second kind of porting is much harder and challenging than the procedural oriented porting since it requires a detailed knowledge of the architectural differences, as well as a complete understanding of operating system internals. In this book, we shall not consider the procedural oriented porting. Instead, we shall show how to develop an embedded GPOS for the ARM architecture from ground zero.

8.4 Develop an Embedded GPOS for ARM

PMTX (Wang 2015) is a small Unix-like GPOS originally designed for Intel x86 based PCs. It runs on uniprocessor PCs in 32-bit protected mode using dynamic paging. It supports process management, memory management, device drivers, a Linux-compatible EXT2 file system and a command-line based user interface. Most ARM processors have only a single core. In this chapter, we shall focus on how to adapt PMTX to single CPU ARM based systems. Multicore CPUs and multiprocessor systems will be covered later in Chap. 9. For ease of reference, we shall denote the resulting system as EOS, for Embedded Operating System.

8.5 Organization of EOS

8.5.1 Hardware Platform

EOS should be able to run on any ARM based system that supports suitable I/O devices. Since most readers may not have a real ARM based hardware system, we shall use the emulated ARM Versatilepb VM (ARM Versatilepb 2016) under QEMU as the platform for implementation and testing. The emulated Versatilepb VM supports the following I/O devices.

(1). SDC: EOS uses a SDC as the primary mass storage device. The SDC is a virtual disk, which is created as follows.

```
dd if=/dev/zero of=$1 bs=4096 count=33280 # 512+32768 4KB blocks
fdisk disk       # create partition 1 = [2048 to 266239] sectors
losetup -o $(expr 2048 \* 512) --sizelimit $(expr 266239 \* 512) \
           /dev/loop1 $1
mke2fs -b 4096 /dev/loop1 32768 # mke2fs with 32K 4KB blocks
mount /dev/loop1 /mnt           # mount as loop device
 (cd /mnt; mkdir bin boot dev etc user) # populate with DIRs
umount /mnt
```

For simplicity, the virtual SDC has only one partition, which begins from the (fdisk default) sector 2048. After creating the virtual SDC, we set up a loop device for the SDC partition and format it as an EXT2 file system with 4 KB block size and one blocks-group. The single blocks-group on the SDC image simplifies both the file system traversal and the inodes and disk blocks management algorithms. Then we mount the loop device and populate it with DIRs and files, making it ready for use. The resulting file system size is 128 MB, which should be big enough for most applications. For larger file systems, the SDC can be created with multiple blocks-groups, or multiple partitions. The following diagram shows the SDC contents.

```
---------|-----------------Partition 1 ------------------------|
|M|booter|super|gd  |. . . |bmap|imap|inodes    |data blocks   |
|---------------------------------------------------------------
         |< ----------------- EXT2 FS ----------------------- >|
         |-- bin : binary executable command files
```

```
|- boot : bootable kernel images
|-- dev : special files (I/O devices)
|-- etc : passwd file
|-- user: user home directories
```

On the SDC, the MBR sector (0) contains the partition table and the beginning part of a booter. The remaining part of the booter is installed in sectors 2 to booter_size, assuming that the booter size is no more than 2046 sectors or 1023 KB (The actual booter size is less than 10 KB). The booter is designed to boot up a kernel image from an EXT2 file system in a SDC partition. When the EOS kernel boots up, it mounts the SDC partition as the root file system and runs on the SDC partition.

(2). LCD: the LCD is the primary display device. The LCD and the keyboard play the role of the system console.
(3). Keyboard: this is the keyboard device of the Versatilepb VM. It is the input device for both the console and UART serial terminals.
(4). UARTs: these are the (4) UARTs of the Versatilepb VM. They are used as serial terminals for users to login. Although it is highly unlikely that an embedded system will have multiple users, our purpose is to show that the EOS system is capable of supporting multiple users at the same time.
(5). Timer: the VersatilepbVM has four timers. EOS uses timer0 to provide a time base for process scheduling, timer service functions, as well as general timing events, such as to maintain Time-of-Day (TOD) in the form of a wall clock.

8.5.2 EOS Source File Tree

The source files of EOS are organized as a file tree.

```
EOS
|- BOOTER    : stage-1 and stage-2 booters
|- type.h, include.h, mk scripts
|- kernel    : kernel source files
|- fs        : file system files
|- driver    : device driver files
|- USER      :  commands and user mode programs
```

BOOTER	:	this directory contains the source code of stage-1 and stage-2 booters
type.h	:	EOS kernel data structure types, system parameters and constants
include.h	:	constants and function prototypes
mk	:	sh scripts to recompile EOS and install bootable image to SDC partition

8.5.3 EOS Kernel Files

```
———————————— Kernel: Process Management Part ————————————
type.h      : kernel data structure types, such as PROC, resources, etc.
ts.s        : reset handler, tswitch, interrupt mask, interrupt handler entry/exit code, etc.
eoslib      : kernel library functions; memory and string operations.
————————————— Kernel files ——————————————————————
queue.c     : enqueue, dequue, printQueue and list operation functions
wait.c      : ksleep, kwakeup, kwait, kexit functions
loader.c    : ELF executable image file loader
mem.c       : page tables and page frame management functions
fork.c      : kfork, fork, vfork functions
exec.c      : kexec function
```

```
threads.c    : threads and mutex functions
signal.c     : signals and signal processing
except.c     : data_abort, prefetch_abort and undef exception handlers
pipe.c       : pipe creation and pipe read/write functions
mes.c        : send/recv message functions
syscall.c    : simple system call functions
svc.c        : syscall routing table
kernel.c     : kernel initialization
t.c          : main entry, initialization, parts of process scheduler
────────────── Device Drivers ───────────────────────────────
lcd.c        : console display driver
pv.c         : semaphore operations
timer.c      : timer and timer service functions
kbd.c        : console keyboard driver
uart.c       : UART serial ports driver
sd.c         : SDC driver
────────────── File system ────────────────────────────────
fs           : implementation of an EXT2 file system
buffer.c     : block device (SDC) I/O buffering
─────────────────────────────────────────────────────────────────
```

EOS is implemented mostly in C, with less than 2% of assembly code. The total number of line count in the EOS kernel is approximately 14,000.

8.5.4 Capabilities of EOS

The EOS kernel consists of process management, memory management, device drivers and a complete file system. It supports dynamic process creation and termination. It allows process to change execution images to execute different programs. Each process runs in a private virtual address space in User mode. Memory management is by two-level dynamic paging. Process scheduling is by both time-slice and dynamic process priority. It supports a complete EXT2 file system that is totally Linux compatible. It uses block device I/O buffering between the file system and the SDC driver to improve efficiency and performance. It supports multiple user logins from the console and serial terminals. The user interface sh supports executions of simple commands with I/O redirections, as well as multiple commands connected by pipes. It unifies exception handling with signal processing, and it allows users to install signal catchers to handle exceptions in User mode.

8.5.5 Startup Sequence of EOS

The startup sequence of EOS is as follows. First, we list the logical order of the startup sequence. Then we explain each step in detail.

(1). Booting the EOS kernel
(2). Execute reset_handler to initialize the system
(3). Configure vectored interrupts and device drivers
(4). kernel_init: initialize kernel data structures, create and run the initial process P0
(5). Construct pgdir and pgtables for processes to use two-level dynamic paging
(6). Initialize file system and mount the root file system
(7). Create the INIT process P1; switch process to run P1
(8). P1 forks login processes on console and serial terminals to allow user logins.
(9). When a user login, the login process executes the command interpreter sh.
(10). User enters commands for sh to execute.
(11). When a user logout, the INIT process forks another login process on the terminal.

(1). SDC Booting: An ARM based hardware system usually has an onboard boot-loader implemented in firmware. When an ARM based system starts, the onboard boot-loader loads and executes a stage-1 booter form either a flash device or, in many cases, a FAT partition on a SDC. The stage-1 booter loads a kernel image and transfers control to the kernel image. For EOS on the ARM Versatilpb VM, the booting sequence is similar. First, we develop a stage-1 booter as a standalone program. Then we design a stage-2 booter to boot up EOS kernel image from an EXT2 partition. On the SDC, partition 1 begins from the sector 2048. The first 2046 sectors are free, which are not used by the file system. The stage-2 booter size is less than 10 KB. It is installed in sectors 2 to 20 of the SDC. When the ARM Versatilepb VM starts, QEMU loads the stage-1 booter to 0x10000 (64 KB) and executes it first. The stage-1 booter loads the stage-2 booter from the SDC to 2 MB and transfers control to it. The stage-2 booter loads the EOS kernel image file (/boot/kernel) to 1 MB and jumps to 1 MB to execute the kernel's startup code. During booting, both stage-1 and stage-2 booters use a UART port for user interface and a simple SDC driver to load SDC blocks. In order to keep the booters simple, both the UART and SDC drivers use polling for I/O. The reader may consult the source code in the booter1 and booter2 directories for details. It also shows how to install the stage-2 booter to the SDC.

8.5.6 Process Management in EOS

In the EOS kernel, each process or thread is represented by a PROC structure which consists of three parts.

```
. fields for process management
. pointer to a per-process resource structure
. kernel mode stack pointer to a dynamically allocated 4KB page as kstack
```

8.5.6.1 PROC and Resource Structures

In EOS, the PROC structure is defined as

```
typedef struct proc{
  struct proc *next;     // next proc pointer
  int    *ksp;           // at 4
  int    *usp;           // at 8 : Umode usp at syscall
  int    *upc;           // at 12: upc at syscall
  int    *ucpsr;         // at 16: Umode cpsr at syscall
  int    status;         // process status
  int    priority;       // scheduling priority
  int    pid;            // process ID
  int    ppid;           // parent process ID
  int    event;          // sleep event
  int    exitCode;       // exit code
  int    vforked;        // whether the proc is VFROKED
  int    time;           // time-slice
  int    cpu;            // CPU time ticks used in ONE second
  int    type;           // PROCESS or THREAD
  int    pause;          // seconds to pause
  struct proc *parent;   // parent PROC pointer
  struct proc *proc;     // process ptr of threads in PROC
  struct pres *res;      // per-process resource pointer
  struct semaphore *sem; // ptr to semaphore proc BLOCKed on
  int    *kstack;        // pointer to Kmode stack
}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

The fields usp, upc and ucpsr are for saving the Umode sp, pc and cpsr during syscall and IRQ interrupt processing. This is because the ARM processor does not stack the Umode sp and cpsr automatically during SWI (system calls) and IRQ (interrupts) exceptions. Since both system calls and interrupts may trigger process switch, we must save the process Umode context manually. In addition to CPU registers, which are saved in the SVC or IRQ stack, we also save Umode sp and cpsr in the PROC structure. 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 EOS, 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, it must wakeup the parent process if the latter is waiting for the child to terminate. In the PROC structure, the parent pointer points to the parent PROC. This allows the 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 low byte of exitValue is the exit value. If it terminates abnormally 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 maximal timeslice of a process, and cpu is its CPU usage time. The timeslice determines how long can a process run, and the CPU usage time is used to compute the process scheduling priority. The pause field is for a process to sleep for a number of seconds. In EOS, process and thread PROCs are identical. The type field identifies whether a PROC is a PROCESS or THREAD. EOS is a uniprocessor (UP) system, in which only one process may run in kernel mode at a time. For process synchronization, it uses sleep/wakeup in process management and implementation of pipes, but it uses semaphores in device drivers and file system. When a process becomes blocked on a semaphore, the sem field 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 inputs from a serial port, it is blocked in the serial port 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;
  int     gid;
  u32     paddress, psize;  // image size in KB
  u32     *pgdir;           // per proc level-1 page table pointer
  u32     *new_pgdir;       // new_pgdir during exec with new size
  MINODE *cwd;              // CWD
  char    name[32];         // executing program name
  char    tty[32];          // opened terminal /dev/ttyXX
  int     tcount;           // threads count in process
  u32     signal;           // 31 signals=bits 1 to 31
  int     sig[NSIG];        // 31 signal handlers
  OFT     *fd[NFD];         // open file descriptors
  struct semaphore mlock;   // message passing
  struct semaphore message;
  struct mbuf     *mqueue;
} PRES;
```

The PRES structure contains process specific information. It includes the process uid, gid, level-1 page table (pgdir) and image size, current working directory, terminal special file name, executing program name, signal and signal handlers, message queue and opened file descriptors, etc. In EOS, both PROC and PRES structures are statically allocated. If desired, they may be constructed dynamically. Processes and threads are independent execution units. Each process executes in a unique address space. All 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 creating 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 structure must have its own file descriptor array. Similarly for signals and messages, etc. In the PROC structure, kstack is a pointer to the process/thread kernel mode stack. In EOS, PROCs are managed as follows.

Free process and thread PROCs are maintained in separate free lists for allocation and deallocation. In EOS, which is a UP system, there is only one readyQueue for process scheduling. The kernel mode stack of the initial process P0 is statically allocated at 8 KB (0x2000). The kernel mode stack of every other PROC is dynamically allocated a (4 KB) page frame only when needed. When a process terminates, it becomes a ZOMBIE but retains its PROC structure, pgdir and the kstack, which are eventually deallocated by the parent process in kwait().

8.5.7 Assembly Code of EOS

The ts.s File: ts.s is the only kernel file in ARM assembly code. It consists of several logically separate parts. For easy of discussion and reference, we shall identified them as ts.s.1 to ts.s.5. In the following, we shall list the ts.s code and explain the functions of the various parts.

```
//--------------------- ts.s file -------------------------
   .text
.code 32
.global reset_handler
.global vectors_start, vectors_end
.global proc, procsize
.global tswitch, scheduler, running, goUmode
.global switchPgdir, mkPtable, get_cpsr, get_spsr
.global irq_tswitch, setulr
.global copy_vector, copyistack, irq_handler, vectorInt_init
.global int_on, int_off, lock, unlock
.global get_fault_status, get_fault_addr, get_spsr
```

8.5.7.1 Reset Handler

```
// -------------------- ts.s.1. ----------------------------
reset_handler:
// set SVC stack to HIGH END of proc[0].kstack[]
  ldr r0, =proc      // r0 points to proc's
  ldr r1, =procsize  // r1 -> procsize
  ldr r2,[r1, #0]    // r2 = procsize
  add r0, r0, r2     // r0 -> high end of proc[0]
  sub r0, #4         // r0 ->proc[0].kstack
  mov r1, #0x2000    // r1 = 8KB
  str r1, [r0]       // proc[0].kstack at 8KB
  mov sp, r1
  mov r4, r0         // r4 is a copy of r0, points PROC0's kstack top

// go in IRQ mode to set IRQ stack
  msr cpsr, #0xD2    // IRQ mode with IRQ and FIQ interrupts off
  ldr sp, =irq_stack // 4KB area defined in linker script t.ld
// go in FIQ mode to set FIQ stack
  msr cpsr, #0xD1
  ldr sp, =fiq_stack  // set FIQ mode sp
// go in ABT mode to set ABT stack
  msr cpsr, #0xD7
  ldr sp, =abt_stack  // set ABT mode stack
// go in UND mode to set UND stack
```

```
  msr cpsr, #0xDB
  ldr sp, =und_stack  // set UND mode stack
// go back in SVC mode
  msr cpsr, #0xD3
// set SVC mode spsr to USER mode with IRQ on
  msr spsr, #0x10     // write to previous mode spsr
```

ts.s.1 is the reset_handler, which begins execution in SVC mode with interrupts off and MMU disabled. First, it initializes proc[0]'s kstack pointer to 8 KB (0x2000) and sets the SVC mode stack pointer to the high end of proc[0].kstack. This makes proc[0]'s kstack the initial execution stack. Then it initializes the stack pointers of other privileged modes for exceptions processing. In order to run processes later in User mode, it sets the SPSR to User mode. Then it continues to execute the second part of the assembly code. In a real ARM based system, FIQ interrupt is usually reserved for urgent event, such as power failure, which can be used to trigger the OS kernel to save system information into non-volatile storage device for recovery later. Since most emulated ARM VMs do not have such a provision, EOS uses only IRQ interrupts but not the FIQ interrupt.

8.5.7.2 Initial Page Table

```
//---------------- ts.s.2 ---------------------------
// copy vector table to address 0
  bl copy_vector
// create initial pgdir and pgtable at 16KB
  bl mkPtable         // create pgdir and pgtable in C
  ldr r0, mtable
  mcr p15, 0, r0, c2, c0, 0   // set TTBR
  mcr p15, 0, r0, c8, c7, 0   // flush TLB
// set DOMAIN 0,1 : 01=CLIENT mode(check permission)
  mov r0,  #0x5               // b0101 for CLIENT
  mcr p15, 0, r0, c3, c0, 0
// enable MMU
  mrc p15, 0, r0, c1, c0, 0
  orr r0, r0, #0x00000001     // set bit0
  mcr p15, 0, r0, c1, c0, 0   // write to c1
  nop
  nop
  nop
  mrc p15, 0, r2, c2, c0
  mov r2, r2
// enable IRQ interrupts, then call main() in C
  mrs r0, cpsr
  bic r0, r0, #0xC0
  mrs cpsr, r0
  BL main                     // call main() directly
  B . // main() never return; in case it does, just loop here
mtable:  .word 0x4000         // initial pgdir at 16KB
```

ts.s.2: the second part of the assembly code performs three functions. First, it copies the vector table to address 0. Then it constructs an initial one-level page table to create an identity mapping of the low 258 MB VA to PA, which includes 256 MB RAM plus 2 MB I/O space beginning at 256 MB. The EOS kernel uses the KML memory mapping scheme, in which the kernel space is mapped to low VA addresses. The initial page table is built at 0x4000 (16 KB) by the mkPtable() function (in t.c file). It will be the page table of the initial process P0, which runs only in Kernel mode. After setting up the initial page table, it configures and enables the MMU for VA to PA address translation. Then it calls main() to continue kernel initialization in C.

8.5.7.3 System Call Entry and Exit

```
/******************** ts.s.3 *****************************/
// SVC (SWI) handler entry point
svc_entry: // syscall parameters are in r0-r3: do not touch
   stmfd sp!, {r0-r12, lr}
// access running PROC
   ldr r5, =running    // r5 = &running
   ldr r6, [r5, #0]    // r6 -> PROC of running
   mrs r7, spsr        // get spsr, which is Umode cpsr
   str r7, [r6, #16]   // save spsr into running->ucpsr
// to SYS mode to access Umode usp, upc
   mrs r7, cpsr        // r7 = SVC mode cpsr
   mov r8, r7          // save a copy in r8
   orr r7, r7, #0x1F   // r7 = SYS mode
   msr cpsr, r7        // change cpsr to SYS mode
// now in SYS mode, sp and lr same as User mode
   str sp, [r6, #8]    // save usp into running->usp
   str lr, [r6, #12]   // save upc into running->upc
// change back to SVC mode
   msr cpsr, r8
// save kmode sp into running->ksp at offest 4;
// used in fork() to copy parent's kstack to child's kstack
   str sp, [r6, #4]    // running->ksp = sp
// enable IRQ interrupts
   mrs r7, cpsr
   bic r7, r7, #0xC0   // I and F bits=0 enable IRQ,FIQ
   msr cpsr, r7
   bl svc_handler      // call SVC handler in C
// replace saved r0 on stack with the return value from svc_handler()
   add sp, sp, #4      // effectively pop saved r0 off stack
   stmfd sp!,{r0}      // push r as the saved r0 to Umode

goUmode:
// disable IRQ interrupts
   mrs r7, cpsr
   orr r7, r7, #0xC0   // I and F bits=1 mask out IRQ,FIQ
   msr cpsr, r7        // write to cpsr
   bl  kpsig           // handle outstanding signals
   bl  reschedule      // reschedule process
// access running PROC
   ldr r5, =running    // r5 = &running
   ldr r6, [r5, #0]    // r6 -> PROC of running
 // goto SYS mode to access user mode usp
   mrs r2, cpsr        // r2 = SVC mode cpsr
   mov r3, r2          // save a copy in r3
   orr r2, r2, #0x1F   // r2 = SYS mode
   msr cpsr, r2        // change to SYS mode
   ldr sp, [r6, #8]    // restore usp from running->usp
   msr cpsr, r3        // back to SVC mode
// replace pc in kstack with p->upc
   mov r3, sp
   add r3, r3, #52     // offset = 13*4 bytes from sp
   ldr r4, [r6, #12]
   str r4, [r3]
```

```
// return to running proc in Umode
   ldmfd sp!, {r0-r12, pc}^
```

8.5.7.4 IRQ Handler

```
// IRQ handler entry point
irq_handler:           // IRQ entry point
   sub lr, lr, #4
   stmfd sp!, {r0-r12, lr}  // save all Umode regs in IRQ stack

// may switch task at end of IRQ processing; save Umode info
   mrs r0, spsr
   and r0, #0x1F
   cmp r0, #0x10      // check whether was in Umode
   bne noUmode        // no need to save Umode context if NOT in Umode
// access running PROC
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running
   mrs r7, spsr
   str r7, [r6, #16]
// to SYS mode to access Umode usp=r13 and cpsr
   mrs r7, cpsr       // r7 = SVC mode cpsr
   mov r8, r7         // save a copy of cpsr in r8
   orr r7, r7, #0x1F  // r7 = SYS mode
   msr cpsr, r7       // change cpsr to SYS mode
// now in SYS mode, r13 same as User mode sp r14=user mode lr
   str sp, [r6, #8]   // save usp into proc.usp at offset 8
   str lr, [r6, #12]  // save upc into proc.upc at offset 12
// change back to IRQ mode
   msr cpsr, r8
noUmode:
   bl irq_chandler    // call irq_handler() in C in SVC mode
// check mode
   mrs r0, spsr
   and r0, #0x1F
   cmp r0, #0x10      // check if in Umode
   bne kiret
// proc was in Umode when IRQ interrupt: handle signal, may tswitch
   bl kpsig
   bl reschedule      // re-schedule: may tswitch

// CURRENT running PROC return to Umode
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running
// restore Umode.[sp,pc,cpsr] from saved PROC.[usp,upc,ucpsr]
   ldr r7, [r6, #16]  // r7 = saved Umode cpsr
// restore spsr to saved Umode cpsr
   msr spsr, r7
// go to SYS mode to access user mode sp
   mrs r7, cpsr       // r7 = SVC mode cpsr
   mov r8, r7         // save a copy of cpsr in r8
   orr r7, r7, #0x1F  // r7 = SYS mode
   msr cpsr, r7       // change cpsr to SYS mode
```

```
// now in SYS mode; restore Umode usp
   ldr sp, [r6, #8]    //  set usp in Umode = running->usp
// back to IRQ mode
   msr cpsr, r8       // go back to IRQ mode
kiret:
   ldmfd sp!, {r0-r12, pc}^ // return to Umode
```

ts.s.3: The third part of the assembly code contains the entry points of SWI (SVC) and IRQ exception handlers. Both the SVC and IRQ handlers are quite unique due to the different operating modes of the ARM processor architecture. So we shall explain them in more detail.

System Call Entry: svc_entry is the entry point of SWI exception handler, which is used for system calls to the EOS kernel. Upon entry, it first saves the process (Umode) context in the process Kmode (SVC mode) stack. System call parameters (a,b,c,d) are passed in registers r0–r3, which should not be altered. So the code only uses registers r4–r10. First, it lets r6 point to the process PROC structure. Then it saves the current spsr, which is the Umode cpsr, into PROC.ucpsr. Then it changes to SYS mode to access Umode registers. It saves the Umode sp and pc into PROC.usp and PROC.upc, respectively. Thus, during a system call, the process Umode context is saved as follows.

$$\text{Umode registers } [\text{r0} - \text{r12},\ \text{r14}] \text{ saved in PROC.kstack}$$
$$\text{Umode } [\text{sp},\ \text{pc},\ \text{cpsr}] \text{ saved in PROC.}[\text{usp},\ \text{upc},\ \text{ucpsr}]$$

In addition, it also saves Kmode sp in PROC.ksp, which is used to copy the parent kstack to child during fork(). Then it enables IRQ interrupts and calls svc_chandler() to process the system call. Each syscall (except kexit) returns a value, which replaces the saved r0 in kstack as the return value back to Umode.

System Call Exit: goUmode is the syscall exit code. It lets the current running process, which may or may not be the original process that did the syscall, return to Umode. First, it disables IRQ interrupts to ensure that the entire goUmode code is executed in a critical section. Then it lets the current running process check and handle any outstanding signals. Signal handling in the ARM architecture is also quite unique, which will be explained later. If the process survives the signal, it calls reschedule() to re-schedule processes, which may switch process if the sw_flag is set, meaning that there are processes in the readyQueue with higher priority. Then the current running process restores [usp, upc, cpsr] from its PROC structure and returns to Umode by

$$\text{ldmfd sp!}, \{\text{r0}-\text{r12}, \text{pc}\}^{\wedge}$$

Upon return to Umode, r0 contains the return value of the syscall.

IRQ Entry: irq_handler is the entry point of IRQ interrupts. Unlike syscalls, which can only originate from Umode, IRQ interrupts may occur in either Umode or Kmode. EOS is a uniprocessor OS. The EOS kernel is non-preemptive, which means it does not switch process while in kernel mode. However, it may switch process if the interrupted process was running in Umode. This is necessary in order to support process scheduling by time-slice and dynamic priority. Task switching in ARM IRQ mode poses a unique problem, which we shall elaborate shortly. Upon entry to irq_handler, it first saves the context of the interrupted process in the IRQ mode stack. Then it checks whether the interrupt occurred in Umode. If so, it also saves the Umode [usp, upc, cpsr] into the PROC structure. Then it calls irq_chandler() to process the interrupt. The timer interrupt handler may set the switch process flag, sw_flag, if the time-slice of the current running process has expired. Similarly, a device interrupt handler may also set the sw_flag if it wakes up or unblocks processes with higher priority. At the end of interrupt processing, if the interrupt occurred in Kmode or the sw_flag is off, there should be no task switch, so the process returns normally to the originally point of interruption. However, if the interrupt occurred in Umode and the sw_flag is set, the kernel switches process to run the process with the highest priority.

8.5.7.5 IRQ and Process Preemption

Unlike syscalls, which always use the process kstack in SVC mode, task switch in IRQ mode is complicated by the fact that, while interrupt processing uses the IRQ stack, task switch must be done in SVC mode, which uses the process kstack. In this case, we must perform the following operations manually.

(1). Transfer the INTERRUPT stack frame from IRQ stack to process (SVC) kstack;
(2). Flatten out the IRQ stack to prevent it from overflowing;
(3). Set SVC mode stack pointer to the INTERRUPT stack frame in the process kstack.
(4). Switch to SVC mode and call tswitch() to give up CPU, which pushes a RESUME stack frame onto the process kstack pointed by the saved PROC.ksp.
(5). When the process regains CPU, it resumes in SVC mode by the RESUME stack frame and returns to where it called tswitch() earlier.
(6). Restore Umode [usp, cpsr] from saved [usp, ucpsr] in PROC structure
(7). Return to Umode by the INTERRUPT stack frame in kstack.

Task switch in IRQ mode is implemented in the code segment irq_tswitch(), which can be best explained by the following diagrams.

(1) Copy INTERRUPT frame from IRQ stack to SVC stack, set SVC_sp,

```
       IRQ_stack                    SVC_stack
 --------------------   copy   ---------------------
   |ulr|ur12 - ur0|      ===>   |ulr|ur12 – ur0|
 --------------------          -------------|-------
   |-- INT frame -|                         SVC_sp
```

(2). Call tswitch() to give up CPU, which pushes a RESUME frame onto the SVC stack and sets the saved PROC.ksp pointing to the resume stack frame.

```
           |-- INT frame -|-RESUME frame-|
           --------------------------------------
SVC_stack =|ulr|ur12 – ur0|klr|kr12 – kr0|
           ----------------------------|---------
                                   SVC_sp = PROC.ksp
```

(3). When the process is scheduled to run, it resumes in SVC mode and returns to

```
       here: // return from tswitch()
      restore Umode.[usp,cpsr] from PROC.[usp,ucpsr]
      ldmfd sp, {r0-r12, pc}^  // return to Umode
// ---------------- ts.s.4 --------------------------
tswitch:              // tswitch() in Kmode
   mrs r0, cpsr        // disable interrupts
   orr r0, r0, #0xC0   // I and F bits=1: mask out IRQ, FIQ
   mrs cpsr, r0        // I and F interrupts are disabled
   stmfd sp!, {r0-r12, lr} // save context in kstack
   ldr r0, =running    // r0=&running; access running->PROC
   ldr r1, [r0, #0]    // r1->running PROC
   str sp, [r1, #4]    // running->ksp = sp
   bl  scheduler       // call scheduler() to pick next running
   ldr r0, =running    // resume CURRENT running
   ldr r1, [r0, #0]    // r1->runningPROC
   ldr sp, [r1, #4]    // sp = running->ksp
   mrs r0, cpsr        // disable interrupts
   bic r0, r0, #0xC0   // enable I and F interrupts
   mrs cpsr, r0
   ldmfd sp!, {r0-r12, pc}
irq_tswitch:        // irq_tswitch: task switch in IRQ mode
   mov r0, sp       // r0 = IRQ mode current sp
   bl copyistack    // transfer INT frame from IRQ stack to SVC stack
   mrs r7, spsr     // r7 = IRQ mode spsr, which must be Umode cpsr
// flatten out irq stack
   ldr sp, =irq_stack_top
// change to SVC mode
   mrs r0, cpsr
   bic r1, r0, #0x1F  // r1 = r0 = cspr's lowest 5 bits cleared to 0
   orr r1, r1, #0x13  // OR in 0x13=10011 = SVC mode
   msr cpsr, r1       // write to cspr, so in SVC mode now
   ldr r5, =running   // r5 = &running
   ldr r6, [r5, #0]   // r6 -> PROC of running
// svc stack already has an irq frame, set SVC sp to kstack[-14]
   ldr sp, [r6, #4]   // SVC mode sp= &running->kstack[SSIZE-14]
   bl tswitch         // switch task in SVC mode
   ldr r5, =running   // r5=&running
   ldr r6, [r5, #0]   // r6 -> PROC of running
   ldr r7, [r6, #16]  // r7 = saved Umode cpsr
// restore spsr to saved Umode cpsr
   msr spsr, r7
// go to SYS mode to access user mode sp
   mrs r7, cpsr       // r7 = SVC mode cpsr
   mov r8, r7         // save a copy of cpsr in r8
   orr r7, r7, #0x1F  // r7 = SYS mode
   msr cpsr, r7       // change cpsr to SYS mode
// now in SYS mode; restore Umode usp
   ldr sp, [r6, #8]   // restore usp
   ldr lr, [r6, #12]  // restore upc; REALLY need this?
// back to SVC mode
   msr cpsr, r8       // go back to IRQ mode
   ldmfd sp!, {r0-r12, pc}^ // return via INT frame in SVC stack
switchPgdir: // switch pgdir to new PROC's pgdir; passed in r0
// r0 contains new PROC's pgdir address
  mcr p15, 0, r0, c2, c0, 0   // set TTBase
  mov r1, #0
  mcr p15, 0, r1, c8, c7, 0   // flush TLB
  mcr p15, 0, r1, c7, c10, 0  // flush cache
  mrc p15, 0, r2, c2, c0, 0
// set domain: all 01=client(check permission)
  mov r0, #0x5                //01|01 for CLIENT|client
  mcr p15, 0, r0, c3, c0, 0
  mov pc, lr                  // return
```

ts.s.4: The fourth part of the assembly code implements task switching. It consists of three functions. tswitch() it for task switch in Kmode, irq_tswitch() is for task switch in IRQ mode and switchPgdir() is for switching process pgdir during task switch. Since all these functions are already explained before, we shall not repeat them here.

```
//-------------ts.s.5 --------------------------
// IRQ interrupt mask/unmask functions
int_on:               // int int_on(int cpsr)
  msr cpsr, r0
  mov pc,lr
```

```
int_off:              // int cpsr = int_off();
  mrs r4, cpsr
  mov r0, r4
  orr r4, r4, #0x80   // set bit means MASK off IRQ interrupt
  msr cpsr, r4
  mov pc,lr
unlock:               // enable IRQ directly
   mrs r4, cpsr
   bic r4, r4, #0x80   // clear bit means UNMASK IRQ interrupt
   msr cpsr, r4
   mov pc,lr
lock:                 // disable IRQ directly
   mrs r4, cpsr
   orr r4, r4, #0x80   // set bit means MASK off IRQ interrupt
   msr cpsr, r4
   mov pc,lr
get_cpsr:
   mrs r0, cpsr
   mov pc, lr
get_spsr:
   mrs r0, spsr
 mov pc, lr
setulr: // setulr(oldPC): set Umode lr=oldPC for signal catcher()
   mrs r7, cpsr   // to SYS mode
   mov r8, r7     // save cpsr in r8
   orr r7, #0x1F  //
   msr cpsr, r7
// in SYS mode now
   mov lr, r0     // set Umode lr to oldPC
   msr cpsr, r8   // back to original mode
   mov pc, lr     // return
vectors_start:
   LDR PC, reset_handler_addr
   LDR PC, undef_handler_addr
   LDR PC, svc_handler_addr
   LDR PC, prefetch_abort_handler_addr
   LDR PC, data_abort_handler_addr
   B .
   LDR PC, irq_handler_addr
   LDR PC, fiq_handler_addr
reset_handler_addr:          .word reset_handler
undef_handler_addr:          .word undef_abort_handler
svc_handler_addr:            .word svc_entry
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:            .word irq_handler
fiq_handler_addr:            .word fiq_handler
vectors_end:
// end of ts.s file
```

The last part of the assembly code implements various utility functions, such as lock/unlock, int_off/int_on, and getting CPU status register, etc. Note the difference between lock/unlock and int_off/int_on. Whereas lock/unlock disable/enable IRQ interrupts unconditionally, int_off disables IRQ interrupts but returns the original CPSR, which is restored in int_on.

These are necessary in device interrupt handlers, which run with interrupts disabled but may issue V operation on semaphores to unblock processes.

8.5.8 Kernel Files of EOS

Part 2. t.c file: The t.c file contains the main() function, which is called from reset_handler when the system starts.

8.5.8.1 The main() Function

The main() function consists of the following steps

(1). Initialize the LCD display driver to make printf() work.
(2). Initialize block device I/O buffers for file I/O on SDC.
(3). Configure VIC, SIC interrupt controllers and devices for vectored interrupts.
(4). Initialize device drivers and start timer.
(5). Call kernel_init() to initialize kernel data structures. Create and run the initial process P0. Construct pgdirs and pgtables for processes. Construct free page frame list for dynamic paging. Switch pgdir to use two-level dynamic paging.
(6). Call fs_init() to initialize file system and mount the root file system.
(7). Create the INIT process P1 and load /bin/init as its Umode image.
(8). P0 switches task to run the INIT process P1.

P1 forks login processes on console and serial terminals for users to login. Then it waits for any ZOMBIE children, which include the login processes as well as any orphans, e.g. in multi-stage pipes. When the login processes start up, the system is ready for use.

```
/*********************** t.c file ***************************/
#include "../type.h"
int main()
{
   fbuf_init();        //initialize LCD frame buffer: driver/vid.c
   printf("Welcome to WANIX in Arm\n");
   binit();            // I/O buffers: fs/buffer.c
   vectorInt_init();   // Vectored Interrupts: driver/int.c
   irq_init();         // Configure VIC,SIC,deviceIRQs: driver/int.c
   kbd_init();         // Initialize KBB driver: driver/kbd.c
   uart_init();        // initialize UARTs:      driver/uart.c
   timer_init();       // initialize timer:      driver/timer.c
   timer_start(0);     // start timer0           driver/timer.c
   sdc_init();         // initialize SDC driver: driver/sdc.c
   kernel_init();      // initialize kernel structs:kernel/kernel.c
   fs_init();          // initialize FS and mount root file system
   kfork("/bin/init"); // create INIT proc P1: kernel/fork.c
   printf("P0 switch to P1\n");
   while(1){           // P0 code
     while(!readyQueue); // loop if no runnable procs
     tswitch();        // P0 switch task if readyQueue non-empty
   }
}
```

8.5.8.2 Kernel Initialization

kernel_init() function: The kernel_init() function consists of the following steps.

(1). Initialize kernel data structures. These include free PROC lists, a readyQueue for process scheduling and a FIFO sleepList containing SLEEP processes.
(2). Create and run the initial process P0, which runs in Kmode with the lowest priority 0. P0 is also the idle process, which runs if there are no other runnable processes, i.e. when all other processes are sleeping or blocked. When P0 resumes, it executes a busy waiting loop until the readyQueue is non-empty. Then it switches process to run a ready process with the highest priority. Instead of a busy waiting loop, P0 may put the CPU in a power-saving WFI state with interrupts enabled. After processing an interrupt, it tries to run a ready process again, etc.
(3). Construct a Kmode pgdir at 32 KB and 258 level-2 page table at 5 MB. Construct level-1 pgdirs for (64) processes in the area of 6 MB and their associated level-2 page tables in 7 MB. Details of the pgdirs and page tables will be explained in the next section on memory management.
(4). Switch pgdir to the new level-1 pgdir at 32 KB to use 2-level paging.
(5). Construct a pfreeList containing free page frames from 8 MB to 256 MB, and implement palloc()/pdealloc() functions to support dynamic paging.
(6). Initialize pipes and message buffers in kernel.
(7). Return to main(), which calls fs_init() to initialize the file system and mount the root file system. Then it creates and run the INIT process P1.

```
/********************* kernel.c file *********************/
#include "../type.h"
PROC proc[NPROC+NTHREAD];
PRES pres[NPROC];
PROC *freelist, *tfreeList, *readyQueue, *sleepList, *running;;
int sw_flag;
int procsize = sizeof(PROC);
OFT  oft[NOFT];
PIPE pipe[NPIPE];

int kernel_init()
{
  int i, j;
  PROC *p; char *cp;
  printf("kernel_init()\n");
  for (i=0; i<NPROC; i++){ // initialize PROCs in freeList
    p = &proc[i];
    p->pid = i;
    p->status = FREE;
    p->priority = 0;
    p->ppid = 0;
    p->res = &pres[i];   // res point to pres[i]
    p->next = p + 1;
    // proc[i]'s umode pgdir and pagetable are at 6MB + pid*16KB
    p->res->pgdir = (int *)(0x600000 + (p->pid-1)*0x4000);
  }
  proc[NPROC-1].next = 0;
  freeList = &proc[0];
  // similar code to initialize tfreeList for NTHREAD procs
  readyQueue = 0;
  sleepList = 0;
  // create P0 as the initial running process;
  p = running = get_proc(&freeList);
  p->status = READY;
  p->res->uid = p->res->gid = 0;
  p->res->signal = 0;
  p->res->name[0] = 0;
  p->time = 10000;       // arbitrary since P0 has no time limit
  p->res->pgdir = (int *)0x8000;  // P0's pgdir at 32KB
  for (i=0; i<NFD; i++)  // clear file descriptor array
     p->res->fd[i] = 0;
  for (i=0; i<NSIG; i++) // clear signals
     p->res->sig[i] = 0;
  build_ptable();        // in mem.c file
  printf("switch pgdir to use 2-level paging : ");
  switchPgdir(0x8000);

  // build pfreelist: free page frames begin from 8MB end = 256MB
  pfreeList = free_page_list((int *)0x00800000, (int *)0x10000000);
  pipe_init();      // initialize pipes in kernel
  mbuf_init();      // initialize message buffers in kernel
}
```

8.5.8.3 Process Scheduling Functions

```
int scheduler()
{
  PROC *old = running;
  if (running->pid == 0 && running->status == BLOCK){// P0 only
     unlock();
     while(!readyQueue);
     return;
  }
  if (running->status==READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  if (running != old){
     switchPgdir((int)running->res->pgdir);
  }
  running->time = 10; // time slice = 10 ticks;
  sw_flag = 0;        // turn off switch task flag
}
int schedule(PROC *p)
{
  if (p->status ==READY)
     enqueue(&readyQueue, p);
  if (p->priority > running->priority)
     sw_flag = 1;
}
int reschedule()
{
  if (sw_flag)
    tswitch();
}
```

The remaining functions in t.c include scheduler(), schedule() and reschedule(), which are parts of the process scheduler in the EOS kernel. In the scheduler() function, the first few lines of code apply only to the initial process P0. When the system starts up, P0 executes mount_root() to mount the root file system. It uses an I/O buffer to read the SDC, which causes P0 to block on the I/O buffer until the read operation completes. Since there is no other process yet, P0 can not switch process when it becomes blocked. So it busily waits until the SDC interrupt handler executes V to unblock it. Alternatively, we may modify the SDC driver to use polling during system startup, and switch to interrupt-driven mode after P0 has created P1. The disadvantage is that it would make the SDC driver less efficient since it has to check a flag on every read operation.

8.5.9 Process Management Functions

8.5.9.1 fork-exec

EOS supports dynamic process creation by fork, which creates a child process with an identical Umode image as the parent. It allows process to change images by exec. In addition, it also supports threads within the same process. These are implemented in the following files.

fork.c file: this file contains fork1(), kfork(), fork() and vfork(). **fork1()** is the common code of all other fork functions. It creates a new proc with a pgdir and pgtables. **kfork()** is used only by P0 to create the INIT proc P1. It loads the Umode image file (/bin/init) of P1 and initializes P1's kstack to make it ready to run in Umode. **fork()** creates a child process with an identical Umode image as the parent. **vfork()** is the same as fork() but without copying images.

exec.c file: this file contains **kexec(),** which allows a process to change Umode image to a different executable file and pass command-line parameters to the new image.

threads.c file: this file implements threads in a process and threads synchronization by mutexes.

8.5.9.2 exit-wait

The EOS kernel uses sleep/wakeup for process synchronization in process management and also in pipes. Process management is implemented in the wait.c file, which contains the following functions.

ksleep(): process goes to sleep on an event. Sleeping PROCs are maintained in a FIFO sleepList for waking up in order
kwakeup(): wakeup all PROCs that are sleeping on an event
kexit(): process termination in kernel
kwait(): wait for a ZOMBIE child process, return its pid and exit status

8.5.10 Pipes

The EOS kernel supports pipes between related processes. A pipe is a structure consisting of the following fields.

```
typedef struct pipe{
  char  *buf; // data buffer: dynamically allocated page frame;
  int   head, tail;        // buffer index
  int   data, room;        // counters for synchronization
  int   nreader, nwriter;  // number of READER,WRITER on pipe
  int   busy;              // status of pipe
}PIPE;
```

The syscall int r = pipe(int pd[]);

creates a pipe in kernel and returns two file descriptors in pd[2], where pd[0] is for reading from the pipe and pd[1] is for writing to the pipe. The pipe's data buffer is a dynamically allocated 4 KB page, which will be released when the pipe is deallocated. After creating a pipe, the process typically forks a child process to share the pipe, i.e. both the parent and the child have the same pipe descriptors pd[0] and pd[1]. However, on the same pipe each process must be either a READER or a WRITER, but not both. So, one of the processes is chosen as the pipe WRITER and the other one as the pipe READER. The pipe WRITER must closes its pd[0], redirects its stdout (fd = 1) to pd[1], so that its stdout is connected to the write end of the pipe. The pipe READER must closes its pd[1] and redirects its stdin (fd = 0) to pd[0], so that its stdin is connected to the read end of the pipe. After these, the two processes are connected by the pipe. READER and WRITER processes on the same pipe are synchronized by sleep/wakeup. Pipe read/write functions are implemented in the pipe.c file. Closing pipe descriptor functions are implemented in the open_close.c file of the file system. A pipe is deallocated when all the file descriptors on the pipe are closed. For more information on the implementation of pipes, the reader may consult (Chap. 6.14, Wang 2015) or the pipe.c file for details.

8.5.11 Message Passing

In addition to pipes, the EOS kernel supports inter-process communication by message passing. The message passing mechanism consists of the following components.

(1). A set of NPROC message buffers (MBUFs) in kernel space.
(2). Each process has a message queue in PROC.res.mqueue, which contains messages sent to but not yet received the process. Messages in the message queue are ordered by priority.
(3). send(char *msg, int pid): send a message to a target process by pid.
(4). recv(char *msg): receive a message form proc's message queue.

In EOS, message passing is synchronous. A sending process waits if there are no free message buffers. A receiving process waits if there are no messages in its message queue. Process synchronization in send/recv is by semaphores. The following lists the mes.c file.

```
/************** mes.c file: Message Passing ************/
#include "../type.h"
/******** message buffer type in type.h ********
typedef struct mbuf{
  struct mbuf *next;  // next mbuf pointer
  int sender;         // sender pid
  int priority;       // message priority
  char text[128];     // message contents
} MBUF;
***********************************************/
MBUF mbuf[NMBUF], *freeMbuflist;  // free mbufs; NMBUF=NPROC
SEMAPHORE mlock; // semaphore for exclusive access to mbuf[ ]
int mbuf_init()
{
  int i; MBUF *mp;
  printf("mbuf_init\n");
  for (i=0; i<NMBUF; i++){ // initialize mbufs
      mp = &mbuf[i];
      mp->next = mp+1;
      mp->priority = 1;    // for enqueue()/dequeue()
  }
  freeMbuflist = &mbuf[0];
  mbuf[NMBUF-1].next = 0;
  mlock.value = 1; mlock.queue = 0;
}
MBUF *get_mbuf()           // allocate a mbuf
{
  MBUF *mp;
  P(&mlock);
  mp = freeMbuflist;
  if (mp)
     freeMbuflist = mp->next;
  V(&mlock);
  return mp;
}
```

```
int put_mbuf(MBUF *mp)     // release a mbuf
{
  mp->text[0] = 0;
  P(&mlock);
    mp->next = freeMbuflist;
    freeMbuflist = mp;
  V(&mlock);
}
int ksend(char *msg, int pid)   // send message to pid
{
  MBUF *mp; PROC *p;
  // validate receiver pid
  if ( pid <= 0 || pid >= NPROC){
     printf("sendMsg : invalid target pid %d\n", pid);
     return -1;
  }
  p = &proc[pid];
  if (p->status == FREE || p->status == ZOMBIE){
     printf("invalid target proc %d\n", pid);
     return -1;
  }
  mp = get_mbuf();
  if (mp==0){
     printf("no more mbuf\n");
     return -1;
  }
  mp->sender = running->pid;
  strcpy(mp->text, msg);  // copy text from Umode to mbuf
  // deliver mp to receiver's message queue
  P(&p->res->mlock);
    enqueue(&p->res->mqueue, mp);
  V(&p->res->mlock);
  V(&p->res->message);    // notify receiver
  return 1;
}
int krecv(char *msg)      // receive message from OWN mqueue
{
  MBUF *mp;
  P(&running->res->message);  // wait for message
  P(&running->res->mlock);
    mp = (MBUF *)dequeue(&running->res->mqueue);
  V(&running->res->mlock);
  if (mp){                    // only if it has message
     strcpy(msg, mp->text);   // copy message contents to Umode
     put_mbuf(mp);            // release mbuf
     return 1;
  }
  return -1; // if proc was killed by signal => no message
}
```

8.5.12 Demonstration of Message Passing

In the USER directory, the programs, send.c and recv.c, are used to demonstrate the message passing capability of EOS. The reader may test send/recv messages as follows.

(1). login to the console. Enter the command line recv &. The sh process forks a child to run the recv command but does not wait for the recv process to terminate, so that the user may continue to enter commands. Since there are no messages yet, the recv process will be blocked on its message queue in kernel.
(2). Run the send command. Enter the receiving proc’s pid and a text string, which will be sent to the recv process, allowing it to continue. Alternatively, the reader may also login from a different terminal to run the send command.

8.6 Memory Management in EOS

8.6.1 Memory Map of EOS

The following shows the memory map of the EOS system.

```
------------------- Memory Map of EOS ---------------------
    0-2MB :  EOS Kernel
  2MB-4MB :  LCD display frame buffer
  4MB-5MB :  Data area of 256 I/O buffers
  5MB-6MB :  Kmode level-2 page tables; 258 (1KB) pgtables
  6MB-7MB :  pgdirs for (64) processes, each pgdir=16KB
  7MB-8MB :  unused; for expansion, e.g. to 128 PROC pgdirs
  8MB-256MB: free page frames for dynamic paging
  256-258MB: 2MB I/O space
-----------------------------------------------------------
```

The EOS kernel code and data structures occupy the lowest 2 MB of physical memory. The memory area from 2 to 8 MB are used by the EOS kernel as LCD display buffer, I/O buffers, level-1 and level-2 page tables of processes, etc., The memory area from 8 to 256 MB is free. Free page frames from 8 to 256 MB are maintained in a pfreeList for allocation/dealloction of page frames dynamically.

8.6.2 Virtual Address Spaces

EOS uses the KML virtual address space mapping scheme, in which the kernel space is mapped to low Virtual Address (VA), and User mode space is mapped to high VA. When the system starts, the Memory Management Unit (MMU) is off, so that every address is a real or physical address. Since the EOS kernel is compile-linked with real addresses, it can execute the kernel's C code directly. First, it sets up an initial one-level page table at 16 KB to create an identity mapping of VA to PA and enables the MMU for VA to PA translation.

8.6.3 Kernel Mode Pgdir and Page Tables

In reset_handler, after initializing the stack pointers of the various privileged modes for exception processing, it constructs a new pgdir at 32 KB and the associated level-2 page tables in 5 MB. The low 258 entries of the new pgdir point to their level-2 page tables at 5 MB+i*1 KB (0<=i<258). Each page table contains 256 entries, each pointing to a 4 KB page frame in memory. All other entries of the pgdir are 0's. In the new pgdir, entries 2048–4095 are for User mode VA space. Since the high 2048 entries are all 0's, the pgdir is good only for the 258 MB kernel VA space. It will be the pgdir of the initial process P0, which runs only in Kmode. It is also the prototype of all other pgdirs since their Kmode entries are all identical. Then it switches to the new pgdir to use 2-level paging in Kmode.

8.6.4 Process User Mode Page Tables

Each process has a pgdir at 6 MB+pid*16 KB. The low 258 entries of all pgdirs are identical since their Kmode VA spaces are the same. The number of pgdir entries for Umode VA depends on the Umode image size, which in turn depends on the executable image file size. For simplicity, we set the Umode image size, USZIE, to 4 MB, which is big enough for all the Umode programs used for testing and demonstration. The Umode pgdir and page tables of a process are set up only when the process is created. When creating a new process in fork1(), we compute the number of Umode page tables needed as npgdir = USIZE/1 MB. The Umode pgdir entries point to npgdir dynamically allocated page frames. Each page table uses only the low 1 KB space of the (4 KB) page frame. The attributes of Umode pgdir entries are set to 0x31 for domain 1. In the Domain Access Control register, the access bits of both domains 0 and 1 are set to b01 for client mode, which checks the Access Permission (AP) bits of the page table entries. Each Umode page table contains pointers to 256 dynamically allocated page frames. The attributes of the page table entries are set to 0xFFE for AP = 11 for all the (1 KB) subpages within each page to allow R|W access in User mode.

8.6.5 Switch Pgdir During Process Switch

During process switch, we switch pgdir from the current process to that of the next process and flush the TLB and I and D buffer caches. This is implemented by the switchPgdir() function in ts.s.

8.6.6 Dynamic Paging

In the mem.c file, the functions free_page_list(), palloc() and pdealloc() implement dynamic paging. When the system starts, we build a pfreeList, which threads all the free page frames from 8 to 256 MB in a link list. During system operation, palloc() allocates a free page frame from pfreeList, and pdealloc() releases a page frame back to pfreeList for reuse. The following shows the mem.c file.

```
/***************** mem.c file: Dynamic Paging ***************/
int *pfreeList, *last;
int mkPtable()  // called from ts.s, create initial ptable at 16KB
{
  int i;
  int *ut = (int *)0x4000; // at 16KB
  u32 entry = 0 | 0x41E;   // AP=01(Kmode R|W; Umode NO) domaian=0
  for (i=0; i<4096; i++)   // clear 4096 entries to 0
    ut[i] = 0;
  for (i=0; i<258; i++){   // fill in low 258 entries ID map to PA
    ut[i] = entry;
    entry += 0x100000;
  }
}
int *palloc()              // allocate a page frame
{
  int *p = pfreeList;
  if (p)
     pfreeList = (int *)*p;
  return p;
}
```

```
void pdealloc(int *p)      // deallocate a page frame
{
  *last = (int)(*p);
  *p = 0;
  last = p;
}
// build pfreeList of free page frames
int *free_page_list(int *startva, int *endva)
{
  int *p;
  printf("build pfreeList: start=%x end=%x : ", startva, endva);
  pfreeList = startva;
  p = startva;
  while(p < (int *)(endva-1024)){
    *p = (int)(p + 1024);
     p += 1024;
  }
  last = p;
  *p = 0;
  return startva;
}
int build_ptable()
 {
   int *mtable = (int *)0x8000; // new pgdir at 32KB
   int i, j, *pgdir, paddr;
   printf("build Kmode pgdir at 32KB\n");
   for (i=0; i<4096; i++){      // zero out mtable[ ]
       mtable[i] = 0;
 }
   printf("build Kmode pgtables in 5MB\n");
   for (i=0; i<258; i++){       // point to 258 pgtables in 5MB
      pgtable = (int *)(0x500000 + i*1024);
      mtable[i] = (int)pgtable | 0x11; // 1KB entry in 5MB
      paddr = i*0x100000 | 0x55E;      // AP=01010101 CB=11 type=10
      for (j=0; j<256; j++){
          pgtable[j] = paddr + j*4096; // inc by 4KB
      }
  }
  printf("build 64 proc pgdirs at 6MB\n");
  for (i=0; i<64; i++){
      pgdir = (int *)(0x600000 + i*0x4000); // 16KB each
      for (j=0; j<4096; j++){ // zero out pgdir[ ]
          pgdir[j] = 0;
      }
      for (j=0; j<258; j++){  // copy low 258 entries from mtable[]
          pgdir[j] = mtable[j];
      }
  }
}
```

8.7 Exception and Signal Processing

During system operation, the ARM processor recognizes six types of exceptions, which are FIQ, IRQ, SWI, data_abort, prefecth_abort and undefined exceptions. Among these, FIQ and IRQ are for interrupts and SWI is for system calls. So the only true exceptions are data_abort, prefeth_abort and undefined exceptions, which occur under the following circumstances.

A **data_abort** event occurs when the memory controller or MMU indicates that an invalid memory address has been accessed. Example: attempt to access invalid VA.

A **prefetch_abort** event occurs when an attempt to load an instruction results in a memory fault. Example: if 0x1000 is outside of the VA range, then BL 0x1000 would cause a prefetch abort at the next instruction address 0x1004.

An **undefined** (instruction) event occurs when a fetched and decoded instruction is not in the ARM instruction set and none of the coprocessors claims the instruction.

In all Unix-like systems, exceptions are converted to signals, which are handled as follows.

8.7.1 Signal Processing in Unix/Linux

(1). Signals in Process PROC: Each PROC has a 32-bit vector, which records signals sent to a process. In the bit vector, each bit (except bit 0) represents a signal number. A signal n is present if bit n of the bit vector is 1. In addition, it also has a MASK bit-vector for masking out the corresponding signals. A set of syscalls, such as sigmask, sigsetmask, siggetmask, sigblock, etc. can be used to set, clear and examine the MASK bit-vector. A pending signal becomes effective only if it is not masked out. This allows a process to defer processing masked out signals, similar to CPU masking out certain interrupts.
(2). Signal Handlers: Each process PROC has a signal handler array, int sig[32]. Each entry of the sig[32] array specifies how to handle a corresponding signal, where 0 means DEFault, 1 means IGNore, other nonzero value means by a preinstalled signal catcher (handler) function in Umode.
(3). Trap Errors and signals: When a process encounters an exception, it traps to the exception handler in the OS kernel. The trap handler converts the exception cause to a signal number and delivers the signal to the current running process. If the exception occurs in kernel mode, which must be due to hardware error or, most likely, bugs in the kernel code, there is nothing the process can do. So it simply prints a PANIC error message and stops. Hopefully the problem can be traced and fixed in the next kernel release. If the exception occurs in User mode, the process handles the signal by the signal handler function in its sig[] array. For most signals, the default action of a process is to terminate, with an optional memory dump for debugging. A process may replace the default action with IGNore(1) or a signal catcher, allowing it to either ignore the signal or handle it in User mode.
(4). Change Signal Handlers: A process may use the syscall

$$\textbf{int r} = \textbf{signal}(\textbf{int signal\_number}, \textbf{void}\ {}^*\textbf{handler});$$

to change the handler function of a selected signal number except SIGKILL(9) and SIGSTOP(19). Signal 9 is reserved as the last resort to kill a run-away process, and signal 19 allows a process to stop a child process during debugging. The installed handler, if not 0 or 1, must be the entry address of a function in User space of the form

$$\textbf{void catcher}(\textbf{int signal\_number})\{\ldots\ldots\ldots\ldots\}$$

(5). Signal Processing: A process checks and handles signals whenever it is in Kmode. For each outstanding signal number n, the process first clears the signal. It takes the default action if sig[n] = 0, which normally causes the process to terminate. It ignores the signal if sig[n] = 1. If the process has a pre-installed catcher function for the signal, it fetches the catcher's address and resets the installed catcher to DEFault (0). Then it manipulates the return path in such a way that it returns to execute the catcher function in Umode, passing as parameter the signal number. When the catcher function finishes, it returns to the original point of interruption, i.e. to the place from where it lastly entered Kmode. Thus, the process takes a detour to execute the catcher function first. Then it resumes normal execution.
(6). Reset user installed signal catchers: User installed catcher functions are intended to deal with trap errors in user program code. Since the catcher function is also executed in Umode, it may commit the same kind of trap error again. If so, the process would end up in an infinite loop, jumping between Umode and Kmode forever. To prevent this, the process typically resets the handler to DEFault (0) before executing the catcher function. This implies that a user installed catcher function is valid for only one occurrence of the signal. To catch another occurrence of the same signal, the Umode program

must install the catcher again. However, the treatment of user installed signal catchers is not uniform as it varies across different versions of Unix. For instance, in BSD the signal handler is not reset but the same signal is blocked while executing the signal catcher. Interested readers may consult the man pages of signal and sigaction of Linux for more details.
(7). Inter-Process Signals: In addition to handling exceptions, signals may also be used for inter-process communication. A process may use the syscall

int r = kill(pid, signal_number);

to send a signal to another process identified by pid, causing the latter to execute a pre-installed catcher function in Umode. A common usage of the kill operation is to request the targeted process to terminate, thus the (somewhat misleading) term kill. In general, only related processes, e.g. those with the same uid, may send signals to each other. However, a superuser process (uid=0) may send signals to any process. The kill syscall may use an invalid pid, to mean different ways of delivering the signal. For example, pid = 0 sends the signal to all processes in the same process group, pid = -1 for all processes with pid > 1, etc. The reader may consult Linux man pages on signal/kill for more details.
(8). Signal and Wakeup/Unblock: kill only sends a signal to a target process. The signal does not take effect until the target process runs. When sending signals to a target process, it may be necessary to wakeup/unblock the target process if the latter is in a SLEEP or BLOCKed state. For example, when a process waits for terminal inputs, which may not come for a long time, it is considered as interruptible, meaning that it can be woken up or unblocked by arriving signals. On the other hand, if a process is blocked for SDC I/O, which will come very soon, it is non-interruptible, which should not be unblocked by signals.

8.8 Signal Processing in EOS

8.8.1 Signals in PROC Resource

In EOS, each PROC has a pointer to a resource structure, which contains the following fields for signals and signal handling.

int signal; //31 signals; bit 0 is not used.
int sig[32]; //signal handlers : 0 = default, 1 = ignore, else a catcher in Umode.

For the sake of simplicity, EOS dose not support signal masking. If desired, the reader may add signal masking to the EOS kernel.

8.8.2 Signal Origins in EOS

(1). Hardware: EOS supports the Control-C key from terminals, which is converted to the interrupt signal SIGINT(2) delivered to all processes on the terminal, and the interval timer, which is converted to the alarm signal SIGALRM(14) delivered to the process.
(2). Traps: EOS supports data_abort, prefetch and undefined instruction exceptions.
(3). From Other Process: EOS supports the kill(pid, signal) syscall, but it does not enforce permission checking. Therefore, a process may kill any process. If the target process is in the SLEEP state, kill() wakes it up. It the target process is BLOCKed for inputs in either the KBD of a UART driver, it is unblocked also.

8.8.3 Deliver Signal to Process

The kill syscall delivers a signal to a target process. The algorithm of the kill syscall is

```
/************* Algorithm of kill syscall  **************/
int kkill(int pid, int sig_number)
```

```
{
  (1). validate signal number and pid;
  (2). check permission to kill; // not enforced, may kill any pid
  (3). set proc.signal.[bit_sig_number] to 1;
  (4). if proc is SLEEP, wakeup pid;
  (5). if proc is BLOCKed for terminal inputs, unblock proc;
}
```

8.8.4 Change Signal Handler in Kernel

The signal() syscall changes the handler function of a specified signal. The algorithm of the signal syscall is

```
/*********** Algorithm of signal syscall ***************/
int ksignal(int sig_number, int *catcher)
{
  (1). validate sig number, e.g. cannot change signal number 9;
  (2). int oldsig = running->sig[sig_number];
  (3). running->sig[sig_number] = catcher;
  (4). return oldsig;
}
```

8.8.5 Signal Handling in EOS Kernel

A CPU usually checks for pending interrupts at the end of executing an instruction. Likewise, it suffices to let a process check for pending signals at the end of Kmode execution, i.e. when it is about to return to Umode. However, if a process enters Kmode via a syscall, it should check and handle signals first. This is because if a process already has a pending signal, which may cause it to die, executing the syscall would be a waste of time. On the other hand, if a process enters Kmode due to an interrupt, it must handle the interrupt first. The algorithm of checking for pending signals is

```
/************ Algorithm of Check Signals ***********/
int check_sig()
{
  int i;
  for (i=1; i<NSIG; i++){
      if (running->signal & (1 << i)){
         running->signal &= ~(1 << i);
         return i;
      }
  }
  return 0;
}
```

A process handles outstanding signals by the code segment

```
if (running->signal)
   psig();
```

The algorithm of psig() is

```
high                                                            low
   1   2   3   4   5   6   7   8   9   10  11  12  13  14
 -----------------------------------------------------------------
 |uLR|r12|r11|r10| r9| r8| r7| r6| r5| r4| r3| r2| r1| r0|
 -----------------------------------------------------------------
 |---------------- trap stack frame -------------------->|
```

Fig. 8.1 Process trap stack frame

```
/************** Algorithm psig() ******************/
int psig(int sig)
{
   int n;
   while(n=check_sig()){ // for each pending signal do
(1).  clear running PROC.signal[bit_n]; // clear the signal bit
(2).  if (running->sig[n] == 1)        // IGNore the signal
         continue;
(3).  if (running->sig[n] == 0)        // DEFault : die with sign#
         kexit(n<<8);    // high byte of exitCode=signal number
(4).  // execute signal handler in Umode
      fix up running PROC's "interrupt stack frame" for it to return
      to execute catcher(n) in Umode;
   }
}
```

8.8.6 Dispatch Signal Catcher for Execution in User Mode

In the algorithm of psig(), only step (4) is interesting and challenging. Therefore, we shall explain it in more detail. The goal of step (4) is to let the process return to Umode to execute a catcher(int sig) function. When the catcher() function finishes, it should return to the point where the process lastly entered Kmode. The following diagrams show how to accomplish these. When a process traps to kernel from Umode, its privileged mode stack top contains a "**trap stack frame**" consisting of 14 entries, as shown in Fig. 8.1.

In order for the process return to execute catcher(int sig) with the signal number as a parameter, we modify the trap stack frame as follows.

(1). Replace the **uLR** (at index 1) with the entry address of catcher();
(2). Replace **r0** (at index 14) with sig number, so that upon entry to catcher(), **r0 = sig**;
(3). Set **User mode lr (r14)** to **uLR**, so that when catcher() finishes it would return by **uLR** to the original point of interruption.

8.9 Device Drivers

The EOS kernel supports the following I/O devices.

LCD display: The ARM Versatilepb VM uses the ARM PL110 Color LCD controller [ARM primeCell Color LCD Controller PL110] as the primary display device. The LCD driver used in EOS is the same driver developed in Sect. 2.8.4. It can display both text and images.

Keyboard: The ARM Versatilepb VM includes an ARM PL050 Mouse-Keyboard Interface (MKI) which provides support for a mouse and a PS/2 compatible keyboard [ARM PL050 MKI]. EOS uses a command-line user interface. The mouse device is not used. The keyboard driver of EOS is an improved version of the simple keyboard driver developed in

Sect. 3.6. It supports both lowercase and uppercase keys. It uses some of the function keys as hot keys to display kernel information, such as free PROC lists, process status, contents of the readyQueue, semaphore queues and I/O buffer usage, etc. for debugging. It also supports some escape key sequences, such the arrow keys, for future expansions of EOS.

UARTs: The ARM Versatilepb VM supports four PL011 UART devices for serial I/O. UART drivers are covered in Sect. 3.7. During booting, the EOS booters use UART0 for user interface. After booting up, it uses the UART ports as serial terminals for users to login.

Timer: The ARM Versatilepb VM contains two SB804 dual timer modules (ARM 926EJ-S 2016). Each timer module contains two timers, which are driven by the same clock. Timer drivers are covered in Sect. 3.5. Among the four timers, EOS uses only timer0 to provide timer service functions.

SDC: The SDC driver supports read/write of multi-sectors of file block size.

With the exception of the LCD display, all device drivers are interrupt-driven and use semaphores for synchronization between interrupt handlers and processes.

8.10 Process Scheduling in EOS

Process scheduling in EOS is by time-slice and dynamic process priority. When a process is scheduled to run, it is given a time-slice of 5–10 timer ticks. While a process runs in Umode, the timer interrupt handler decrements its time-slice by 1 at each timer tick. When the process time-slice expires, it sets a switch process flag and calls resechdule() to switch process when it exits Kmode. In order to keep the system simple, EOS uses a simplified priority scheme. Instead of re-computing process priorities dynamically, it uses only two distinct priority values. When a process is in Umode, it runs with the user-level priority of 128. When it enters Kmode, it continues to run with the same priority. If a process becomes blocked (due to I/O or file system operation), it is given a fixed Kmode priority of 256 when it is unblocked to run again. When a process exits Kmode, it drops back to the user level priority of 128.

Exercise 2: Modify the EOS kernel to implement dynamic process priority by CPU usage time.

8.11 Timer Service in EOS

The EOS kernel provides the following timer service to processes.

(1). pause(t): the process goes to sleep for t seconds.
(2). itimer(t): sets an interval timer of t seconds. Send a SIGALRM (14) signal to the process when the interval timer expires.

To simplify the discussion, we shall assume that each process has only one outstanding timer request and the time unit is in seconds in real time, i.e. the virtual timer of a process continues to run whether the process is executing or not.

(1). Timer Request Queue: Timer service provides each process with a virtual or logical timer by a single physical timer. This is achieved by maintaining a timer queue to keep track of process timer requests. A timer queue element (TQE) is a structure

```
typedef struct tq{
        struct tq *next;    // next element pointer
        int       time;     // requested time
        struct PROC *proc;  // pointer to PROC
        int    (*action)(); // 0|1|handler function pointer
}TQE;
TQE *tq, tqe[NPROC];        // tq = timer queue pointer
```

In the TQE, action is a function pointer, where 0 means WAKEUP, 1 means NOTIFY, other value = entry address of a handler function to execute. Initially, the timer queue is empty. As processes invoke timer service, their requests are added to the timer queue. Figure 8.2 shows an example of the timer queue.

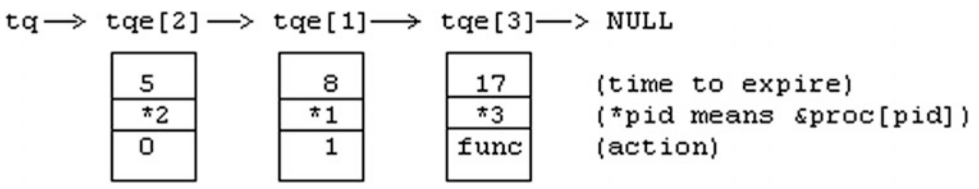


Fig. 8.2 Timer request queue

At each second, the interrupt handler decrements the time field of each TQE by 1. When a TQE's time reaches 0, the interrupt handler deletes the TQE from the timer queue and invokes the action function of the TQE. For example, after 5 s, it deletes tqe[2] from the timer queue and wakes up P2. In the above timer queue, the time field of each TQE contains the exact remaining time. The disadvantage of this scheme is that the interrupt handler must decrement the time field of each and every TQE. In general, an interrupt handler should complete the interrupt processing as quickly as possible. This is especially important for the timer interrupt handler. Otherwise, it may loss ticks or even never finish. We can speed up the timer interrupt handler by modifying the timer queue as shown in Fig. 8.3.

In the modified timer queue, the time field of each TQE is relative to the cumulative time of all the preceding TQEs. At each second, the timer interrupt handler only needs to decrement the time of the first TQE and process any TQE whose time has expired. With this setup, insertion and deletion of a TQE must be done carefully. For example, if a process P4 makes an itimer(10) request, its TQE should be inserted after TQ[1] with a time = 2, which changes the time of TQ[3] to 7. Similarly, when P1 calls itimer(0) to cancel its timer request, its TQE[1] will be deleted from the timer queue, which changes the time of TQE[3] to 12, etc. The reader is encouraged to figure out the general algorithms for insertion and deletion of TQEs.

(2). Timer Queue as a Critical Region: The timer queue data structure is shared by processes and the timer interrupt handler. Accesses to the timer queue must be synchronized to ensure its integrity. EOS is a uniprocessor OS, which allows only one process to execute in kernel at a time. In the EOS kernel a process can not be interfered by another process, so there is no need for process locks. However, while a process executes, interrupts may occur. If a timer interrupt occurs while a process is in the middle of modifying the timer queue, the process would be diverted to execute the interrupt handler, which also tries to modify the timer queue, resulting in a race condition. To prevent interference from interrupt handler, the process must mask out interrupts when accessing the timer queue. The following shows the algorithm of itimer().

```
/*************** Algorithm of itimer() ********************/
int itimer(t)
{
  (1). Fill in TQE[pid] information, e.g. proc pointer, action.
  (2). lock();      // mask out interrupts
  (3).  traverse timer queue to compute the position to insert TQE;
  (4).  insert the TQE and update the time of next TQE;
  (5). unlock();    // unmask interrupts
}
```

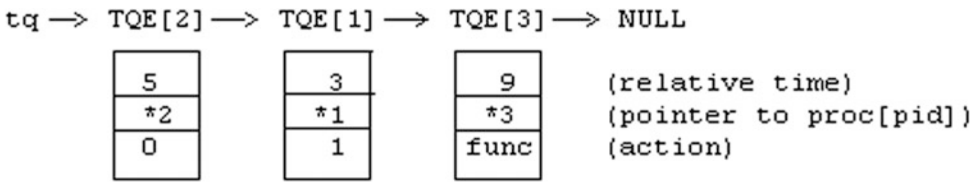


Fig. 8.3 Improved timer request queue

8.12 File System

A general purpose operating system (GPOS) must support file system to allow users to save and retrieve information as files, as well as to provide a platform for running and developing application programs. In fact, most OS kernels require a root file system in order to run. Therefore, file system is an integral part of a GPOS. In this section, we shall discuss the principles of file operations, demonstrate file operations by example programs and show the implementation of a complete EXT2 file system in EOS.

8.12.1 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 mass storage device, e.g. a SDC, into partitions.
mkfs : format a 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 general purpose OS 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, which rely on device drivers for I/O on real devices.

```
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 opened 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.

```
/******* Example program: read first 1KB of a file *******/
#include <stdio.h>
#include <stdlib.h>
```

```
#include <fcntl.h>
int 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, (long)1024, SEEK_SET);   // lseek to byte 1024
   n = read(fd, buf, 1024);           // read 1024 bytes of file
   close(fd);
}
```

In the above example program, 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 of data would come with a terrific cost. Every time we have to enter kernel mode, 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 1 to 8 KB. For instance, in Linux, the default block size is 4 KB for hard disks and 1 KB for floppy disks. Therefore, 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 user mode programs 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 mode program 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
```

Most contemporary OS supports Graphic User Interface (GUI), which allows the user to do file operations through the GUI. For instance, clicking a mouse button on a program name icon may invoke the program execution. Similarly, clicking a pointing device on a filename icon followed by choosing Copy of a pull-down menu copies the file contents to a global buffer, which can be transferred to a target file by Paste, etc. Many users are so accustomed to using the GUI that they often ignore and do not understand what is really going on underneath the GUI interface. This is fine for most naïve computer users, but not for computer science and computer engineering students.

(6). Sh Scripts:

Although much more convenient than system calls, commands must be entered manually, or by repeatedly dragging and clicking a pointing device as in the case of using GUI, 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/Linux commands. It also supports variables and control statements, such as if, do, for, while, case, etc. In practice, sh scripts are used extensively in systems programming on all Unix-like systems. In addition to sh, many other script languages, such as Perl and Tcl, are also in wide use.

8.12.2 File I/O Operations

Figure 8.4 shows the diagram of file I/O operations. In Fig. 8.4, 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) to (10), which are explained below.

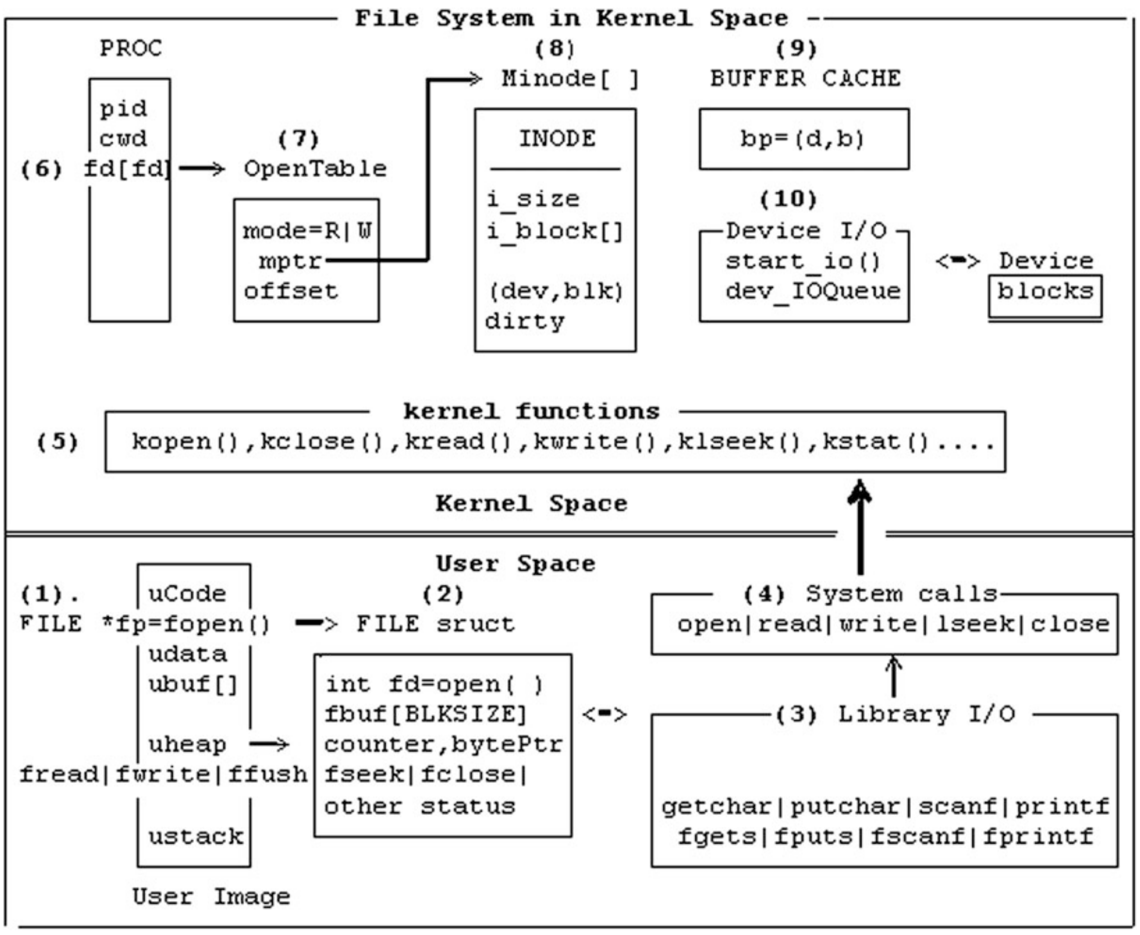


Fig. 8.4 File operation diagram

———————————————— 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.

(2). fopen() creates a FILE structure in user (heap) space containing a file descriptor, fd, and a fbuf[BLKSIZE]. It issues 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 when necessary and they transfer chunks of data from/to kernel in BLKSIZE 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, the kernel kread() function
. Compute logical block number, lbk;
. Convert logical block to physical block, 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.

Processing of the write(fd, fbuf[], BLKSIZE) syscall is similar, except that it may allocate new disk blocks and grow the file size as new data are written to the file.

(9). In order to improve disk I/O efficiency, the OS kernel usually uses a set of I/O buffers as a cache between kernel memory and I/O devices to reduce the number of physical I/O. We shall discuss I/O buffering in later sections.

We illustrate the relationship between the various levels of file operations by the following examples.

Example 1. The example program C8.1 shows how to make a directory by the syscall

```
            int mkdir(char *dirname, int mode)
/**** Program C8.1: mkdir.c: run as a.out dirname ****/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(int argc, char *argv[])
{   int r;
    if (argc < 2){
       printf("Usage: a.out dirname); exit(1);
    }
    if ((r = mkdir(argv[1], 0755)) < 0)
       perror("mkdir"); // print error message
    }
}
```

In order to make a directory, the program must issue a mkdir() system call since only the kernel knows how to make directories. The system call is routed to kmkdir() in kernel, which tries to create a new directory with the specified name and mode. The system call returns r = 0 if the operation succeeds or -1 if it fails. If the system call fails, the error number is in a (extern) global variable errno. The program may call the library function perror("mkdir"), which prints the program name followed by a string describing the cause of the error, e.g. mkdir: File exists, etc.

Exercise 1: Modify the program C8.1 to make many directories by a single command, e.g. mkdir dir1 dir2 …. dirn //with command-line parameters

Exercise 2: The Linux system call **int r = rmdir(char *pathname)** removes a directory, which must be empty. Write a C program rmdir.c, which removes a directory.

Example 2: This example develops a ls program which mimics the ls –l command of Unix/Linux: In Unix/Linxu, the stat system calls

```
int  stat(const char *file_name, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

return information of the specified file. The difference between stat() and lstat() is that the former follows symbolic links but the latter does not. The returned information is in a stat structure (defined in stat.h), which is

```
struct stat {
   dev_t     st_dev;      /* device */
   ino_t     st_ino;      /* inode number */
   mode_t    st_mode;     /* file type and permissions */
   nlink_t   st_nlink;    /* number of hard links */
   uid_t     st_uid;      /* user ID of file owner */
   gid_t     st_gid;      /* group ID of owner */
   dev_t     st_rdev;     /* device type (if inode device) */
   off_t     st_size;     /* total size, in bytes */
   blksize_t st_blksize;  /* blocksize for filesystem I/O */
   blkcnt_t  st_blocks;   /* number of 512-byte sectors */
   time_t    st_atime;    /* time of last access */
   time_t    st_mtime;    /* time of last modification */
   time_t    st_ctime;    /* time of last status change */
};
```

In the stat structure, st_dev identifies the device (number) on which the file resides and st_ino is its inode number on that device. The field st_mode is a 2-byte integer, which specifies the file type, special usage and permission bits for protection. Specifically, the bits of st_mode are

```
  4    3   3   3   3
|----|---|---|---|---|
|tttt|fff|rwx|rwx|rwx|
```

The highest 4 bits of st_mode determine the file type. For examples, b1000 = REGular file, b0100 = DIRectory, b1100 = symbolic link file, etc. The file type can be tested by the pre-defined macros S_ISREG, S_ISDIR, S_ISLNK, etc. For example,

```
if (S_ISREG(st_mode))  // test for REGular file
if (S_ISDIR(st_mode))  // test for DIR file
if (S_ISLNK(st_mode))  // test for symbolic link file
```

The low 9 bits of st_mode define the permission bits as r (readable) w (writeable) x (executable) for owner of the file, same group as the owner and others. For directory files, the x bit means whether cd into the directory is allowed or not allowed. The field st_nlink is the number of hard links to the file, st_size is the file size in bytes, st_atime, st_mtime and st_ctime are time fields. In Unix-like systems, time is the elapsed time in seconds since 00:00:00 of January 1, 1970. The time fields can be converted to strings in calendar form by the the library function char *ctime(time_t *time).

Based on the information returned by the stat system call, we can write a ls.c program, which behaves the same as the ls –l command of Unix/Linux. The program is denoted by C8.2, which is shown below.

```
/** Program C8.2: ls.c: run as a.out [filename] **/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>

char *t1 = "xwrxwrxwr-------";
char *t2 = "----------------";
struct stat mystat, *sp;
int ls_file(char *fname)    // list a single file
{
  struct stat fstat, *sp = &fstat;
  int r, i;
  char sbuf[4096];
  r = lstat(fname, sp);     // lstat the file
  if (S_ISDIR(sp->st_mode))
     printf("%c",'d');      // print file type as d
  if (S_ISREG(sp->st_mode))
     printf("%c",'-');      // print file type as -
  if (S_ISLNK(sp->st_mode))
     printf("%c",'l');      // print file type as l
  for (i=8; i>=0; i--){
    if (sp->st_mode & (1<<i))
       printf("%c", t1[i]); // print permission bit as r w x
    else
 printf("%c", t2[i]); // print permission bit as -
  }
  printf("%4d ", sp->st_nlink);  // link count
  printf("%4d ", sp->st_uid      // uid
  printf("%8d ", sp->st_size);   // file size
  strcpy(ftime, ctime(&sp->st_ctime));
  ftime[strlen(ftime)-1] = 0;       // kill \n at end
  printf("%s ",ftime);              // time in calendar form
  printf("%s", basename(fname));    // file basename
  if (S_ISLNK(sp->st_mode)){        // if symbolic link
     r = readlink(fname, sbuf, 4096);
     printf(" -> %s", sbuf);        // -> linked pathname
  }
  printf("\n");
}
int ls_dir(char *dname)       // list a DIR
{
  char name[256];             // EXT2 filename: 1-255 chars
  DIR *dp;
  struct dirent *ep;
  // open DIR to read names
  dp = opendir(dname);        // opendir() syscall
  while (ep = readdir(dp)){   // readdir() syscall
     strcpy(name, ep->d_name);
     if (!strcmp(name, ".") || !strcmp(name, ".."))
        continue;             // skip over . and .. entries
     strcpy(name, dname);
     strcat(name, "/");
     strcat(name, ep->d_name);
     ls_file(name);           // call list_file()
  }
}
int main(int argc, char *argv[])
{
  struct stat mystat, *sp;
  int r;
  char *s;
  char filename[1024], cwd[1024];
  s = argv[1];               // ls [filename]
  if (argc == 1)             // no parameter: ls CWD
     s = "./";
  sp = &mystat;
  if ((r = stat(s, sp)) < 0){ // stat() syscall
     perror("ls"); exit(1);
  }
  strcpy(filename, s);
  if (s[0] != '/'){          // filename is relative to CWD
     getcwd(cwd, 1024);      // get CWD path
     strcpy(filename, cwd);
     strcat(filename, "/");
     strcat(filename,s);     // construct $CWD/filename
  }
  if (S_ISDIR(sp->st_mode))
     ls_dir(filename);      // list DIR
  else
     ls_file(filename);     // list single file
}
```

The reader may compile and run the program C8.2 under Linux. It should list either a single file or a DIR in the same format as does the Linux ls –l command.

Example 4: File copy programs: This example shows the implementations of two file copy programs; one uses system calls and the other one uses library I/O functions. Both programs are run as **a.out src dest,** which copies src to dest. We list the programs side by side in order to to show their similarities and differences.

```
------------ cp.sysall.c ------------|------ cp.libio.c ------------
#include <stdio.h>                   |  #include <stdio.h>
#include <stdlib.h>                  |  #include <stdlib.h>
#include <fcntl.h>                   |
main(int argc, char *argv[ ])        | main(int argc, char *argv[ ])
{                                    | {
 int fd, gd;                         |  FILE *fp, *gp;
 int n;                              |  int n;
 char buf[4096];                     |  char buf[4096];
 if (argc < 3) exit(1);              |  if (argc < 3) exit(1);
 fd = open(argv[1], O_RDONLY);       |  fp = fopen(argv[1], "r");
 gd = open(argv[2],O_WRONLY|O_CREAT);|  gp = fopen(argc[1], "w+");
 if (fd < 0 || gd < 0) exit(2);      |  if (fp==0 || gp==0) exit(2);
 while(n=read(fd, buf, 4096)){       |  while(n=fread(buf,1,4096,fp)){
    write(gd, buf, n);               |     fwrite(buf, 1, n, gp);
 }                                   |  }
 close(fd); close(gd);               |  fclose(fp); fclose(gp);
}                                    | }
--------------------------------------------------------------------
```

The program cp.syscall.c shown on the left-hand side uses system calls. First, it issues open() system calls to open the src file for READ and the dest file foe WRITE, which creates the dest file if it does not exist. The open() syscalls return two (integer) file descriptors, fd and gd. Then it uses a loop to copy data from fd to gd. In the loop, it issues read() syscall on fd to read up to 4 KB data from the kernel to a local buffer. Then it issues write() syscall on gd to write the data from the local buffer to kernel. The loop ends when read() returns 0, indicating that the source file has no more data to read.

The program cp.libio.c shown on the right-hand side uses library I/O functions. First, it calls fopen() to create two file streams, fp and gp, which are pointers to FILE structures. fopen() creates a FILE structure (defined in stdio.h) in the program's heap area. Each FILE structure contains a local buffer, char fbuf[BLKSIZE], of file block size and a file descriptor field. Then it issues an open() system call to get a file descriptor and record the file descriptor in the FILE structure. Then it returns a file stream (pointer) pointing at the FILE structure. When the program calls fread(), it tries to read data from fbuf in the FILE structure. If fbuf is empty, fread() issues a read() system call to read a BLKSIZE of data from kernel to fbuf. Then it transfers data from fbuf to the program's local buffer. When the program calls fwrite(), it writes data from the program's local buffer to fbuf in the FILE structure. If fbuf is full, fwrite() issues a write() system call to write a BLKSIZE of data from fbuf to kernel. Thus, library I/O functions are built on top of system calls, but they issue system calls only when needed and they transfer data from/to kernel in file block size for better efficiency. Based on these discussions, the reader should be able to deduce which program is more efficient if the objective is only to transfer data. However, if a user mode program intends to access single chars in files or read/write lines, etc. using library I/O functions would be the better choice.

Exercise 3: Assume that we rewrite the data transfer loops in the programs of Example 3 as follows, both transfer one byte at a time.

```
      cp.syscall.c            |      cp.syscall.c
--------------------------------------------------------------
while((n=read(fd, buf, 1)){    |   while((n=fgetc(fp))!= EOF){
   write(gd, buf, n);          |      fputc(n, gp);
}                              |   }
--------------------------------------------------------------
```

Which program would be more efficient? Explain your reasons.

Exercise 4: When copying files, the source file must be a regular file and we should never copy a file to itself. Modify the programs in Example 3 to handle these cases.

8.12.3 EXT2 File System in EOS

For many years, Linux used EXT2 (Card et al. 1995; EXT2 2001) as the default file system. EXT3 (ETX3 2015) is an extension of EXT2. The main addition to EXT3 is a journal file, which records changes made to the file system in a journal log. The log allows for quicker recovery from errors in case of a file system crash. An EXT3 file system with no error is identical to an EXT2 file system. The newest extension of EXT3 is EXT4 (Cao et al. 2007). The major change in EXT4 is in the allocation of disk blocks. In EXT4, block numbers are 48 bits. Instead of discrete disk blocks, EXT4 allocates contiguous ranges of disk blocks, called extents. EOS is a small system intended mainly for teaching and learning the internals of embedded operating systems. Large file storage capacity is not the design goal. Principles of file system design and implementation, with an emphasis on simplicity and compatibility with Linux, are the major focal points. For these reasons, we choose ETX2 as the file system. Support for other file systems, e.g. FAT and NTFS, are not implemented in the EOS kernel. If needed, they can be implemented as user-level utility programs. This section describes the implementation of an EXT2 file system in the EOS kernel. Implementation of EXT2 file system is discussed in detail in (Wang 2015). For the sake of completeness and reader convenience, we include similar information here.

8.12.3.1 File System Organization

Figure 8.5 shows the internal organization of an EXT2 file system. The organization diagram is explained by the labels (1) to (5).

(1) is the PROC structure of the running process. Each PROC has a cwd field, which points to the in-memory INODE of the PROC's Current Working Directory (CWD). It also contains an array of file descriptors, fd[], which point to opened file instances.
(2) is the root directory 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 directory (/) 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.
(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. Naturally, if any such information has been changed while in memory, they must be written back to the storage device when the mounted file system is umounted.

8.12.3.2 Source Files in the EOS/FS Directory

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

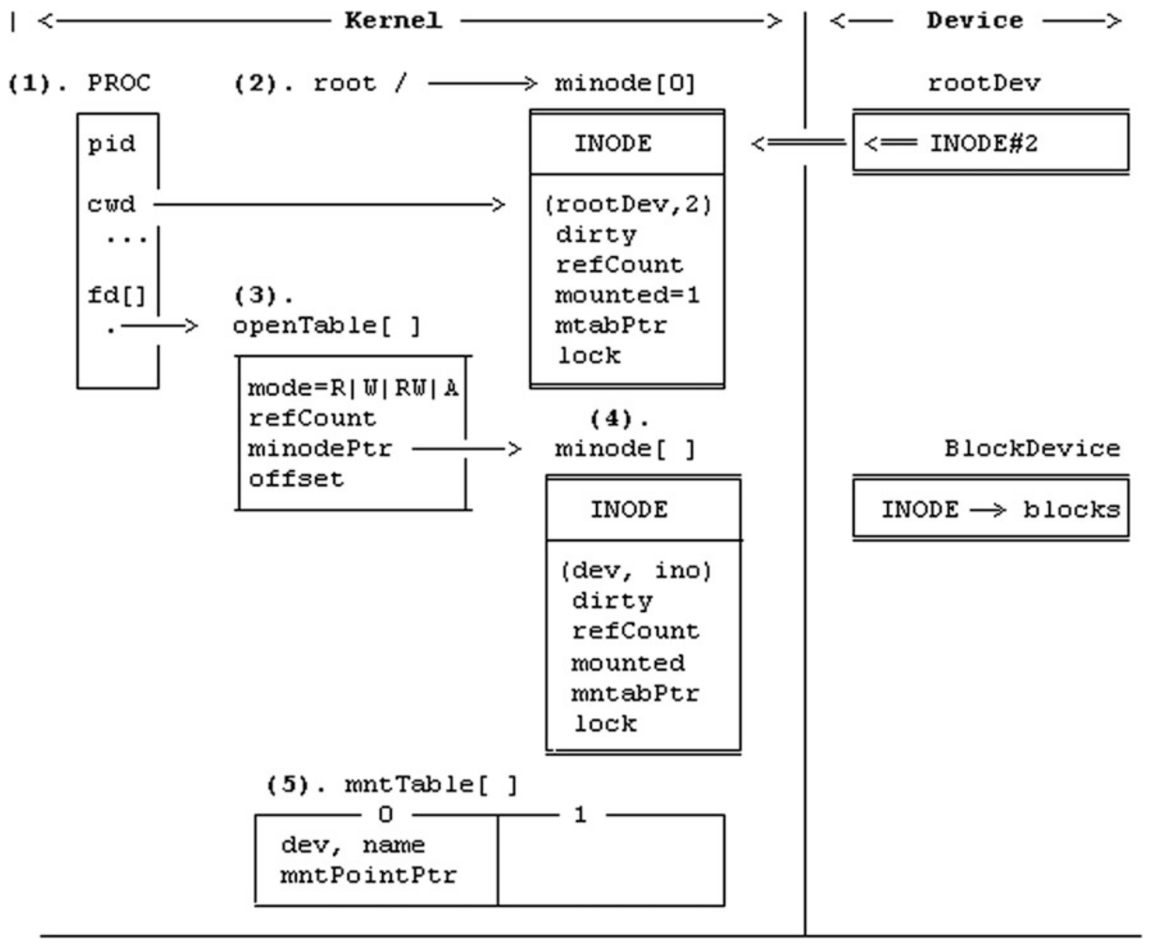


Fig. 8.5 EXT2 file system data structures

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
buffer.c                     : block device 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
—————————————————————————————————————————————————————————————
```

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

```
MINODE minode[NMINODES];    // in memroy 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 tokenize() to break up pathname into component strings. Then it calls search() to search for the component strings in successive directory minodes. Search() returns the inode number of the component string if it exists, or 0 if not.
(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 kernel memory. In addition, the minode is locked (by the minode's locking semaphore) 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. 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 EOS, 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 minode 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 because, with 4 KB block size and an average file name of 16 chars, a DIR can contain more than 3000 entries. We may assume that no user would put that many entries in a single 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 BLKSIZE;
           }
        }
        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.

8.12.4.1 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 similar to mkdir. The algorithm of creat is as follows.

```
/****************** 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, 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 kopen() 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);
}
```

8.12.4.2 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) // st points to STAT struct
{
  (1). get INODE of pathname into a minode;
  (2). copy (dev, ino) to (st_dev, st_ino) of STAT struct in Umode
```

```
    (3). copy other fields of INODE to STAT structure in Umode;
    (4). iput(minode); retrun 0 for OK;
  }
```

8.12.4.3 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 to be removed must be empty. 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.
}
```

8.12.4.4 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 algorithm 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);
}
```

8.12.4.5 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;
     iput(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);
}
```

8.12.4.6 Other Level-1 Functions

Other level-1 functions include stat, 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);
```

8.12.5 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.

8.12.5.1 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 constants 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 8.6 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 first opens a file, it sets the refCount in the OFT to 1. When a process forks, it copies all opened file descriptors to the child process, so that the child process share all the opened file descriptors with 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 and clears its fd[] entry to 0. When an OFT's refCount reaches 0, it calls iput() to dispose of the the minode and deallocates the OFT. The OFT's offset is a conceptual pointer to the current byte position in the file for read/write. It is initialized to 0 for R|W|RW mode or to file size for APPEND mode.

In EOS, 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, lseek allows the byte offset to go beyond the current file size but it does not allocate any disk block for the file. 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;
}
```

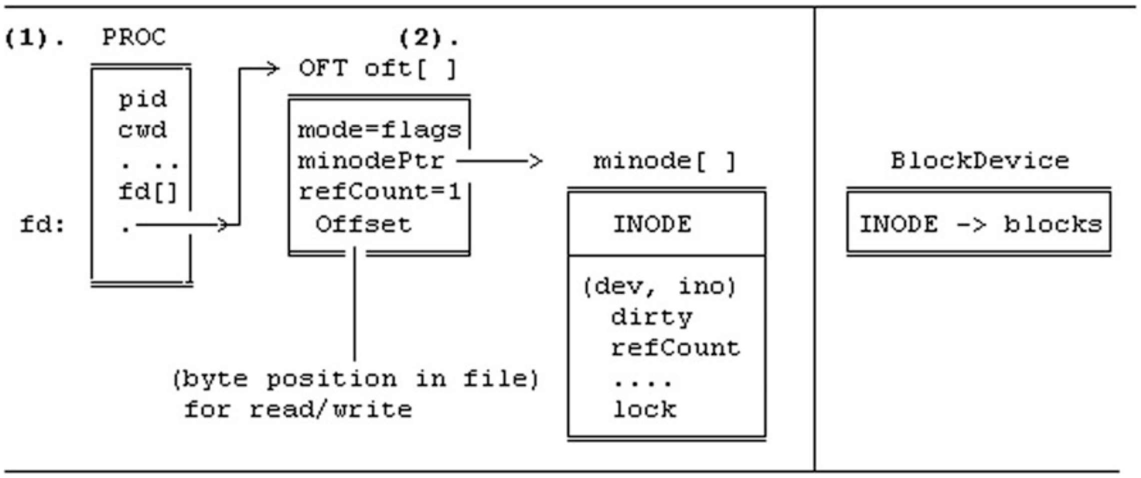


Fig. 8.6 Data structures of open()

8.12.5.2 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; 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. 8.7. 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 (logically) contiguous bytes, numbered from 0 to fileSize-1. As Fig. 8.7 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 EOS, block size is 4 KB and files have at most double indirect blocks.

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

```
/* 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
```

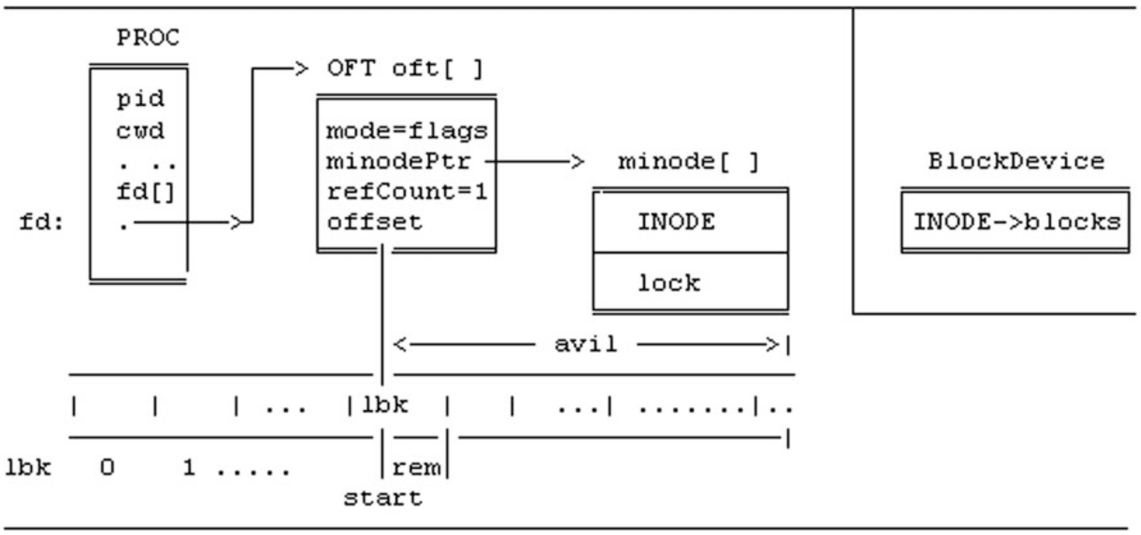


Fig. 8.7 Data structures for Read_file()

```
          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;
 }
```

8.12.5.3 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. 8.8.

In Fig. 8.8, 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

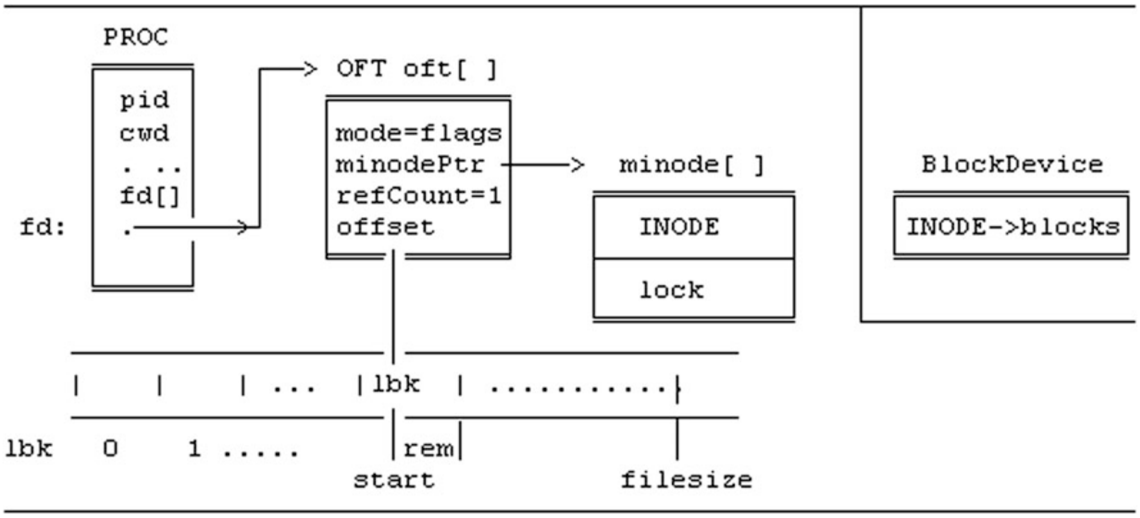


Fig. 8.8 Data structures for write_file()

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

8.12.5.4 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 EOS 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 number of that device type. For example, /dev/sdc0 = (3,0) represents the entire SDC, /dev/sdc1 = (3,1) represents the first partition of a SDC, /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

```
sdc_read(), sdc_write(),         // SDC read/write
console_read(), console_write(), // console read/write
serial_read(), serial_write(),   // serail port read/write
```

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
     nocall,         nocall,        // 2=FD (no FD in EOS)
     sdc_read,       sdc_write,     // 3=SDC
     console_read,   console_write, // 4=console
     serial_read,    serial_write   // 5=serial ports
};
```

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.

8.12.5.5 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-like 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

$$\text{struct dirent} *\text{ep} = \text{readdir}(\text{DIR} *\text{dp});$$

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 EOS does not yet support user-level I/O streams by library functions, 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, which is used to routes readdir syscalls to the kreaddir() function.

```
int kreaddir(int fd, struct udir *dp) // struct udir{DIR; name[256]};
{
   // same as kread() in kernel except:
   use the current byte offset in OFT to read the next DIR record;
   copy the DIR record into *udir in User space;
   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(DIR *dp) call.

8.12.6 Implementation of Level-3 FS

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

8.12.6.1 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.
}
```

8.12.6.2 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 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.

8.12.6.3 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, EOS ignores gid. It uses only the process uid to check for file access permission.

8.12.6.4 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 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, EOS does not yet support effective uid. Permission checking is based on real uid.

8.12.6.5 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 EOS, 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.

File operations from User mode are all based on system calls. As of now, EOS does not yet support library file I/O functions on file streams.

8.13 Block Device I/O Buffering

The EOS file system uses I/O buffering for block device (SDC) to improve the efficiency of file I/O operations. I/O buffering is implemented in the buffer.c file. When EOS starts, it calls binit() to initialize 256 I/O buffers. Each I/O buffer consists of a header for buffer management and a 4 KB data area for a block of SDC data. The 1 MB data area of the I/O buffers is allocated in 4 MB–5 MB of the EOS system memory map. Each buffer (header) has a lock semaphore for exclusive access to the buffer. The buffer management algorithm is as follows.

8.14 I/O Buffer Management Algorithm

(1). bfreelist = a list of free buffers. Initially all buffers are in the bfreelist.
(2). dev_tab = a device table for the SDC partition. It contains the device id number, the start sector number and size in number of sectors. It also contains two buffer linked lists. The dev_list contains all I/O buffers that are assigned to the device, each identified by the buffer's (dev, blk) numbers. The device I/O queue contains buffers for pending I/O.
(3). When a process needs to read a SDC block data, it calls

```
struct buffer *bread(dev, blk)
{
  struct buffer *bp = getblk(dev, blk); // get a bp =(dev,blk)
  if (bp data invalid){
     mark bp for READ
     start_io(bp);        // start I/O on buffer
     P(&bp->iodone);      // wait for I/O completion
  }
  return bp;
}
```

(4). After reading data from a buffer, the process releases the buffer by brelse(bp). A released buffer remains in the device list for possible reuse. It is also in the bfreelist if it is not in use.
(5). When a process writes data to a SDC block, it calls

```
int bwrite(dev, blk)
{
   struct buffer *bp;
   if (write new block or a complete block)
      bp = getblk(dev, blk);  // get a buffer for (dev,blk)
   else                       // write to existing block
      bp = bread(dev,blk);    // get a buffer with valid data
```

```
       write data to bp;
       mark bp data valid and dirty (for delayed write-back)
       brelse(bp);                // release bp
    }
```

(6). Dirty buffer contain valid data, which can be read/write by any process. A dirty buffer is written back to SDC only when it is to be reassigned to a different block, at which time, it is written out by

```
    awrite(struct buffer *bp)  // for ASYNC write
    {
       mark bp ASYNC write;
       start_io(bp);  // do not wait for completion
    }
```

When an ASYNC write operation completes, the SDC interrupt handler turns off the buffer's ASYNC and dirty flags and releases the buffer.

```
int start_io(struct buf *bp) // start I/O on bp
{
  int ps = int_off();
  enter bp into dev_tab.IOqueue;
  if (bp is first in dev_tab.IOqueue){
     if(bp is for READ)
       get_block(bp->blk, bp->buf);
     else  // WRITE
       put_block(bp->blk, bp->buf);
  }
  int_on(ps);
}
```

```
(7). SDC Interrupt Handler:
    {
       bp = dequeue(dev_tab.IOqueue);
       if (bp==READ){
           mark bp data valid;
           V(&bp->iodone);  // unblock process waiting on bp
       else{
           turn off bp ASYNC flag
           brelse(bp);
       }
       bp = dev_tab.IOqueue;
       if (bp){  // I/O queue non-empty
          if (bp==READ)
             get_block(bp->blk, bp->buf);
          else
             put_block(bp->blk, bp->buf);
           }
    }
```

(8). getblk() and brelse() form the core of buffer management. The following lists the algorithms of getblk() and brelse(), which use semaphores for synchronization.

```
SEMAPHORE freebuf = NBUF; // counting semaphore
Each buffer has SEMAPHOREs lock = 1; io_done = 0;

 struct buf *getblk(int dev, int blk)
{
  struct buf *bp;
  while(1){
    P(&freebuf);           // get a free buf
    bp = search_dev(dev,blk);
    if (bp){               // buf in cache
       hits++;             // buffer hits number
       if (bp->busy){      // if buf busy
         V(&freebuf);      // bp not in freelist, give up the free buf
         P(&bp->lock);     // wait for bp
         return bp;
       }
       // bp in cache and not busy
       bp->busy = 1;       // mark bp busy
       out_freelist(bp);
       P(&bp->lock);       // lock bp
       return bp;
    }
    // buf not in cache; already has a free buf in hand
    lock();
      bp = freelist;
      freelist = freelist->next_free;
    unlock();
    P(&bp->lock);         // lock the buffer
    if (bp->dirty){       // delayed write buf, can't use it
       awrite(bp);
       continue;          // continue while(1) loop
    }
    // bp is a new buffer; reassign it to (dev,blk)
    if (bp->dev != dev){
      if (bp->dev >= 0)
         out_devlist(bp);
      bp->dev = dev;
      enter_devlist(bp);
    }
    bp->dev = dev; bp->blk = blk;
    bp->valid = 0; bp->async = 0; bp->dirty = 0;
    return bp;
  }
}
int brelse(struct buf *bp)
{
  if (bp->lock.value < 0){ // bp has waiter
      V(&bp->lock);
      return;
  }
  if (freebuf.value < 0 && bp->dirty){
      awrite(bp);
      return;
  }
  enter_freelist(bp);       // enter b pint bfreeList
  bp->busy = 0;             // bp non longer busy
  V(&bp->lock);             // unlock bp
  V(&freebuf);              // V(freebuf)
}
```

Since both processes and the SDC interrupt handler access and manipulate the free buffer list and device I/O queue, interrupts are disabled when processes operate on these data structures to prevent any race conditions.

8.14.1 Performance of the I/O Buffer Cache

With I/O buffering, the hit ratio of the I/O buffer cache is about 40% when the system starts. During system operation, the hit ratio is constantly above 60%. This attests to the effectiveness of the I/O buffering scheme.

8.15 User Interface

All user commands are ELF executable files in the /bin directory (on the root device). From the EOS system point of view, the most important user mode programs are init, login and sh, which are necessary to start up the EOS system. In the following, we shall explain the roles and algorithms of these programs.

8.15.1 The INIT Program

When EOS starts, the initial process P0 is handcrafted. P0 creates a child P1 by loading the /bin/init file as its Umode image. When P1 runs, it executes the init program in user mode. Henceforth, P1 plays the same role as the INIT process in Unix/Linux. A simple init program, which forks only one login process on the system 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;
int parent()     // P1's code
{
  int pid, status;
  while(1){
    printf("INIT : wait for ZOMBIE child\n");
    pid = wait(&status);
    if (pid==console){   // if console login process died
       printf("INIT: forks a new console login\n");
       console = fork(); // fork another one
       if (console)
           continue;
       else
          exec("login /dev/tty0"); // new console login process
    }
    printf("INIT: I just buried an orphan child proc %d\n", pid);
  }
}
main()
{
   int in, out;   // file descriptors for terminal I/O
   in  = open("/dev/tty0", O_RDONLY); // file descriptor 0
   out = open("/dev/tty0", O_WRONLY); // for display to console
   printf("INIT : fork a login proc on console\n");
   console = fork();
   if (console)  // parent
      parent();
   else          // child: exec to login on tty0
      exec("login /dev/tty0");
}
```

8.15.2 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;   char name[128],password[128]
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:");    gets(name);
          printf("password:"); gets(password);
          for each line in /etc/passwd file do{
              tokenize user 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");
       }
}
```

8.15.3 The sh Program

After login, the user process typically executes the command interpreter sh, which gets command lines from the user and executes the commands. For each command line, if the command is non-trivial, i.e. not cd or exit, sh forks a child process to execute the command line and waits for the child to terminate. For simple commands, the first token of a command line is an executable file in the /bin directory. A command line may contain I/O redirection symbols. If so, the child sh handles I/O redirections first. Then it uses exec to change image to execute the command file. When the child process terminates, it wakes up the parent sh, which prompts for another command line. If a command line contains a pipe symbol, such as cmd1 | cmd2, the child sh handles the pipe by the following do_pipe algorithm.

```
/***************** do_pipe Algorithm **************/
int pid, pd[2];
pipe(pd);  // create a pipe: pd[0]=READ, pd[1]=WRITE
pid = fork();       // fork a child to share the pipe
if (pid){           // parent: as pipe READER
   close(pd[1]);    // close pipe WRITE end
   dup2(pd[0], 0);  // redirect stdin to pipe READ end
   exec(cmd2);
}
else{               // child : as pipe WRITER
   close(pd[0]);    // close pipe READ end
   dup2(pd[1], 1);  // redirect stdout to pipe WRITE end
   exec(cmd1);
}
```

Multiple pipes are handled recursively, from right to left.

8.16 Demonstration of EOS

8.16.1 EOS Startup

Figure 8.9 shows the startup screen of the EOS system. After booting up, it first initializes the LCD display, configures vectored interrupts and initializes device drivers. Then it initializes the EOS kernel to run the initial process P0. P0 builds page directories, page tables and switches page directory to use dynamic 2-level paging. P0 creates the INIT process P1 with /bin/init as Umode image. Then it switches process to run P1 in User mode. P1 forks a login process P2 on the console and another login process P3 on a serial terminal. When creating a new process, it shows the dynamically allocated page frames of the process image. When a process terminates, it releases the allocated page frames for reuse. Then P1 executes in a loop, waiting for any ZOMBIE child.

Each login process opens its own terminal special file as stdin (0), stdout (1), stderr (2) for terminal I/O. Then each login process displays a login: prompt on its terminal and waits for a user to login. When a user tries to login, the login process validates the user by checking the user account in the /etc/passwd file. After a user login, the login process becomes the user process by acquiring its uid and changing directory to the user's home directory. Then the user process changes image to execute the command interpreter sh, which prompts for user commands and execute the commands.

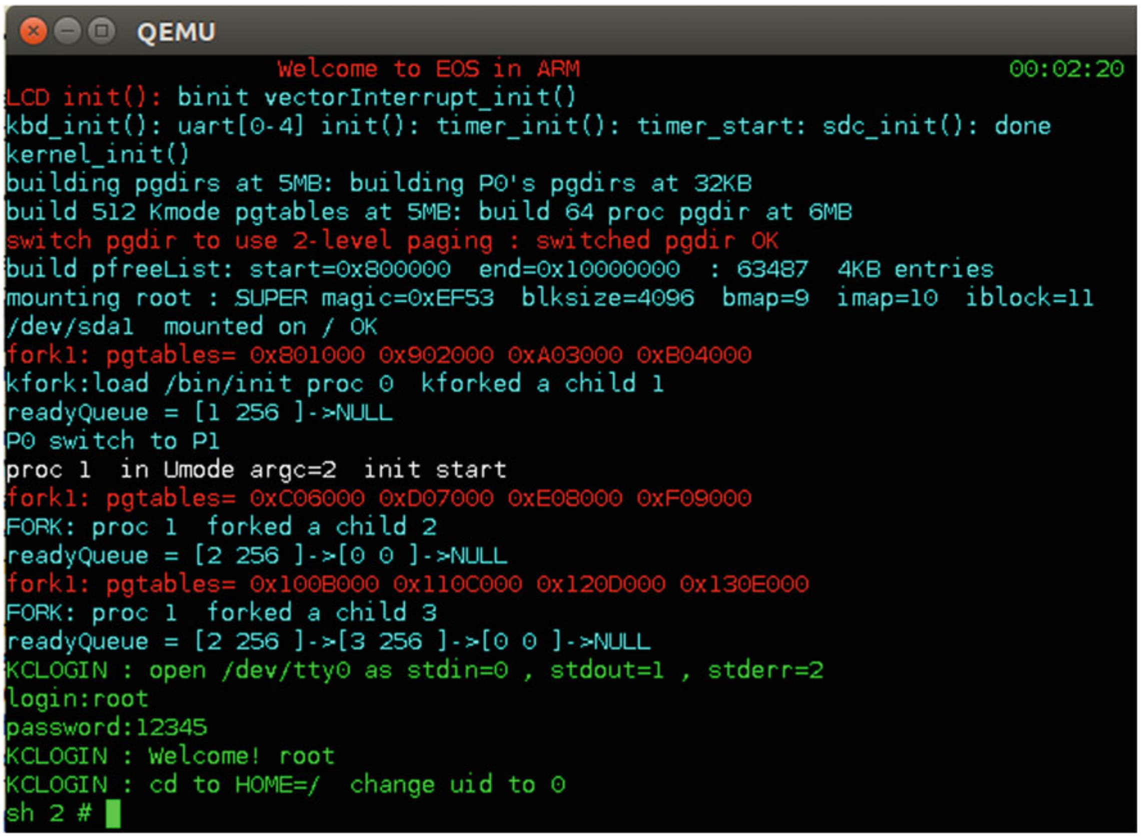
8.16.2 Command Processing in EOS

Figure 8.10 shows the processing sequence of the command line "cat f1 | grep line" by the EOS sh process (P2).

For any non-trivial command line, sh forks a child process to execute the command and waits for the child to terminate. Since the command line has a pipe symbol, the child sh (P8) creates a pipe and forks a child (P9) to share the pipe. Then the child sh (P8) reads from the pipe and executes the command grep. The child sh (P9) writes to the pipe and executes command cat. P8 and P9 are connected by a pipe and run concurrently. When the pipe reader (P8) terminates, it sends the child P9 as an orphan to the INIT process P1, which wakes up to free the orphan process P9.

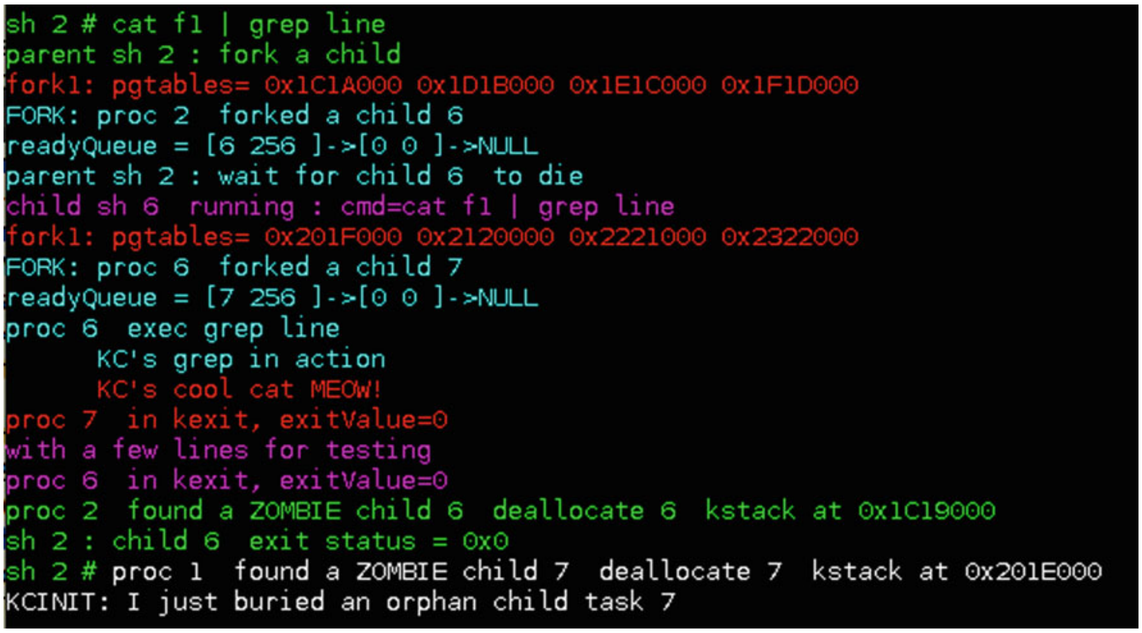
8.16.3 Signal and Exception Handling in EOS

In EOS, exceptions are handled by the unified framework of signal processing. We demonstrate exception and signal processing in EOS by the following examples.



```
QEMU
                    Welcome to EOS in ARM                                  00:02:20
LCD init(): binit vectorInterrupt_init()
kbd_init(): uart[0-4] init(): timer_init(): timer_start: sdc_init(): done
kernel_init()
building pgdirs at 5MB: building P0's pgdirs at 32KB
build 512 Kmode pgtables at 5MB: build 64 proc pgdir at 6MB
switch pgdir to use 2-level paging : switched pgdir OK
build pfreeList: start=0x800000  end=0x10000000  : 63487  4KB entries
mounting root : SUPER magic=0xEF53  blksize=4096  bmap=9  imap=10  iblock=11
/dev/sda1  mounted on / OK
fork1: pgtables= 0x801000 0x902000 0xA03000 0xB04000
kfork:load /bin/init proc 0  kforked a child 1
readyQueue = [1 256 ]->NULL
P0 switch to P1
proc 1  in Umode argc=2  init start
fork1: pgtables= 0xC06000 0xD07000 0xE08000 0xF09000
FORK: proc 1  forked a child 2
readyQueue = [2 256 ]->[0 0 ]->NULL
fork1: pgtables= 0x100B000 0x110C000 0x120D000 0x130E000
FORK: proc 1  forked a child 3
readyQueue = [2 256 ]->[3 256 ]->[0 0 ]->NULL
KCLOGIN : open /dev/tty0 as stdin=0 , stdout=1 , stderr=2
login:root
password:12345
KCLOGIN : Welcome! root
KCLOGIN : cd to HOME=/  change uid to 0
sh 2 #
```

Fig. 8.9 Startup screen of EOS



```
sh 2 # cat f1 | grep line
parent sh 2 : fork a child
fork1: pgtables= 0x1C1A000 0x1D1B000 0x1E1C000 0x1F1D000
FORK: proc 2  forked a child 6
readyQueue = [6 256 ]->[0 0 ]->NULL
parent sh 2 : wait for child 6  to die
child sh 6  running : cmd=cat f1 | grep line
fork1: pgtables= 0x201F000 0x2120000 0x2221000 0x2322000
FORK: proc 6  forked a child 7
readyQueue = [7 256 ]->[0 0 ]->NULL
proc 6  exec grep line
      KC's grep in action
      KC's cool cat MEOW!
proc 7  in kexit, exitValue=0
with a few lines for testing
proc 6  in kexit, exitValue=0
proc 2  found a ZOMBIE child 6  deallocate 6  kstack at 0x1C19000
sh 2 : child 6  exit status = 0x0
sh 2 # proc 1  found a ZOMBIE child 7  deallocate 7  kstack at 0x201E000
KCINIT: I just buried an orphan child task 7
```

Fig. 8.10 Command processing by the EOS sh

8.16.3.1 Interval Timer and Alarm Signal Catcher

In the USER directory, the itimer.c program demonstrates interval timer, alarm signal and alarm signal catcher.

```
/******************* itimer.c file ***********************/
void catcher(int sig)
{ printf("proc %d in catcher: sig=%d\n", getpid(), sig); }

main(int argc, char *argv[])
{
  int t = 1;
  if (argc>1) t = atoi(argv[1]);   // timer interval
  printf("install catcher? [y|n]");
  if (getc()=='y')
     signal(14, catcher);  // install catcher() for SIGALRM(14)
  itimer(t);               // set interval timer in kernel
  printf("proc %d looping until SIGALRM\n", getpid());
  while(1);                // looping until killed by a signal
}
```

In the itimer.c program, it first lets the user to either install or not to install a catcher for the SIGALRM(14) signal. Then, it sets an interval timer of t seconds and executes a while(1) loop. When the interval timer expires, the timer interrupt handler sends a SIGALRM(14) signal to the process. If the user did not install a signal 14 catcher, the process will die by the signal. Otherwise, it will execute the catcher once and continue to loop. In the latter case, the process can be killed by other means, e.g. by the Control_C key or by a kill pid command from another process. The reader may modify the catcher() function to install the catcher again. Recompile and run the system to observe the effect.

8.16.3.2 Exceptions Handling in EOS

In addition to timer signals, we also demonstrate unified exception and signal processing by the following user mode programs. Each program can be run as a user command.

Data.c: This program demonstrates data_abort exception handling. In the data_abort handler, we first read and display the MMU's fault status and address registers to show the MMU status and invalid VA that caused the exception. If the exception occurred in Kmode, it must be due to bugs in the kernel code. In this case, there is nothing the kernel can do. So it prints a PANIC message and stops. If the exception occurred in Umode, the kernel converts it to a signal, SIGSEG(11) for segmentation fault, and sends the signal to the process. If the user did not install a catcher for signal 11, the process will die by the signal. If the user has installed a signal 11 catcher, the process will execute the catcher function in Umode when it gets a number 11 signal. The catcher function uses a long jump to bypass the faulty code, allowing the process to terminate normally.

Prefetch.c: This program demonstrates prefetch_abort exception handling. The C code attempts to execute the in-line assembly code asm("bl 0x1000"); which would cause a prefetch_abort at the next PC address 0x1004 because it is outside of the Umode VA space. In this case, the process gets a SIGSEG(11) signal also, which is handled the same way as a data_abort exception.

Undef.c: This program demonstrates undefined exception handling. The C code attempts to execute

asm(‘‘mcr p14,0,r1,c8,c7,0’’)

which would cause an undef_abort because the coprocessor p14 does not exist. In this case, the process gets an illegal instruction signal SIGILL(4). If the user has not installed a signal 4 catcher, the process will die by the signal.

Divide by zero: Most ARM processors do not have divide instructions. Integer divisions in ARM are implemented by idiv and udiv functions in the aeabi library, which check for divide by zero errors. When a divide by zero is detected, it branches to the __aeabi_idiv0 function. The user may use the link register to identify the offending instruction and take remedial actions. In the Versatilepb VM, divide by zero simply returns the largest integer value. Although it is not possible to generate divide by zero exceptions on the ARM Versatilepb VM, the exception handling scheme of EOS should be applicable to other ARM processors.

8.17 Summary

This chapter presents a fully functional general purpose embedded OS, denoted by EOS. The following is a brief summary of the organization and capabilities of the EOS system.

1. System Images: Bootable kernel image and User mode executables are generated from a source tree by ARM toolchain (of Ubuntu 15.0 Linux) and reside in an EXT2 file system on a SDC partition. The SDC contains stage-1 and stage-2 booters for booting up the kernel image from the SDC partition. After booting up, the kernel mounts the SDC partition as the root file system.
2. Processes: The system supports NPROC=64 processes and NTHRED=128 threads per process, both can be increased if needed. Each process (except the idle process P0) runs in either Kernel mode or User mode. Memory management of process images is by 2-level dynamic paging. Process scheduling is by dynamic priority and time-slice. It supports inter-process communication by pipes and messages passing. The EOS kernel supports fork, exec, vfork, threads, exit and wait for process management.
3. It contains device drivers for the most commonly used I/O devices, e.g. LCD display, timer, keyboard, UART and SDC. It implements a fully Linux compatible EXT2 file system with I/O buffering of SDC read/write to improve efficiency and performance.
4. It supports multi-user logins to the console and UART terminals. The User interface sh supports executions of simple commands with I/O re-directions, as well as multiple commands connected by pipes.
5. It provides timer service functions, and it unifies exceptions handling with signal processing, allowing users to install signal catchers to handle exceptions in User mode.
6. The system runs on a variety of ARM virtual machines under QEMU, mainly for convenience. It should also run on real ARM based system boards that support suitable I/O devices. Porting EOS to some popular ARM based systems, e.g. Raspberry PI-2 is currently underway. The plan is to make it available for readers to download as soon as it is ready.

PROBLEMS

1. In EOS, the initial process P0's kpgdir is at 32KB. Each process has its own pgdir in PROC.res. With a 4MB Umode image size, entries 2048-2051 of each PROC's pgdir define the Umode page tables of the process. When switch task, we use

switchPgdir((int)running–>res–>pgdir);

to switch to the pgdir of the next running process. Modify the scheduler() function (in kernel.c file) as follows.,

```
int *kpgdir = (int *)0x8000; // Kmode pgdir at 32KB
if (running != old){         // truly switch process
   for (i=0; i<npgdir; i++)  // copy Umode pgtables to kpgdir
       kpgdir[2048+i] = running->res->pgdir[2048+i];
   switchPgdir((int)kpgdir); // use the same kpgdir at 32KB
}
```

(1). Verify that the modified scheduler() still works, and explain WHY?
(2). Extend this scheme to use only one kpgdir for ALL processes.
(3). Discuss the advantages and disadvantages of using one pgdir per process vs. one pgdir for all processes.
2. The send/recv operations in EOS use the synchronous protocol, which is blocking and may lead to deadlock due to, e.g. no free message buffers. Redesign the send/recv operations to prevent deadlocks.
3. 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 EOS file system.
4. In the read_file algorithm of Sect. 8.12.5.2, data can be read from a file to 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).
5. Modify the write-file algorithm in Sect. 8.12.5.3 to allow
(1). Write data to kernel space, and
(2). Optimize data transfer by copying chunks of data.
6. 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?
7. Currently, EOS does not yet support file streams. Implement library I/O functions to support file streams in user space.

References

Android: https://en.wikipedia.org/wiki/Android_operating_system, 2016.
ARM Versatilepb: ARM 926EJ-S, 2016: Versatile Application Baseboard for ARM926EJ-S User Guide, Arm information Center, 2016.
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: www.nongnu.org/ext2-doc/ext2.html, 2001.
EXT3: jamesthornton.com/hotlist/linux-filesystems/ext3-journal, 2015.
FreeBSD: FreeBSD/ARM Project, https://www.freebsd.org/platforms/arm.html, 2016.
Raspberry_Pi: https://www.raspberrypi.org/products/raspberry-pi-2-model-b, 2016.
Sevy, J., "Porting NetBSD to a new ARM SoC", http://www.netbsd.org/docs/kernel/porting_netbsd_arm_soc.html, 2016.
Wang, K.C., "Design and Implementation of the MTX Operating System", Springer International Publishing AG, 2015.