Chapter 2. Foundations and Background

Abstract: Chapter 2 covers the foundations and background that are needed to study operating systems. These include computer hardware system, CPU operations, memory models, virtual address and physical address, major functions of operating systems, program development steps, dynamic and static linking, program execution and termination, function calls and stack usage, linking C programs with assembly code, linear to block address conversion algorithm and EXT2 file system. It also explains the login process and command execution in a typical operating system.

2.1. Computer system

2.1.1. Computer Hardware System

Figure 2.1 shows a typical computer hardware system. It consists of one or more Processor Units (PUs), which share a common memory and a set of I/O devices. I/O devices include hard disk, CDROM, USB device, floppy drive, console, terminals and printers, etc.

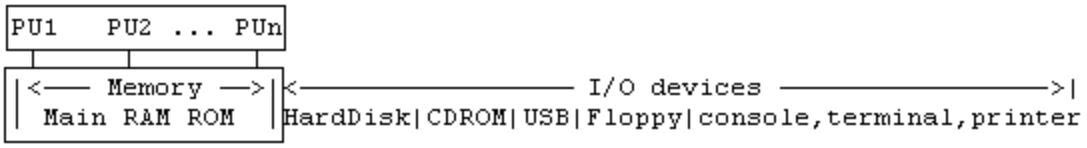

Figure 2.1. Computer Hardware System.

The PUs may be physically separate units. With current multicore processor technology, they may reside in the same processor package. A system with only one PU is called a Uniprocessor (UP) system. Traditionally, the PU in a UP system is called the Central Processing Unit (CPU). A computer system with a multiple number of PUs is called a Multiprocessor (MP) system. In a MP system, the term CPU is also used to refer to the individual PUs. For ease of discussion, we shall assume Uniprocessor systems first.

2.1.2. CPU Operations

Every CPU has a Program Counter (PC), also known as the Instruction Pointer (IP), a flag or status register (SR), a Stack Pointer (SP) and several general registers, where PC points to the next instruction to be executed in memory, SR contains current status of the CPU, e.g. operating mode, interrupt mask and condition code, and SP points to the top of the current stack. The stack is a memory area used by the CPU for special operations, such as push, pop call and return, etc. The operations of a CPU can be modeled by an infinite loop.

```
while(power-on){
    (1). fetch instruction: load *PC as instruction, increment PC to point to the next
         instruction;
```

(2). decode instruction: interpret the instruction's operation code and generate operands;
(3). execute instruction: perform operation on operands, write results to memory if needed; execution may use the stack, implicitly change PC, etc.
(4). check for pending interrupts; may handle interrupts;
}

In each of the above steps, an error condition, called an exception or trap, may occur due to invalid address, illegal instruction, privilege violation, etc. When the CPU encounters an exception, it follows a pre-installed pointer in memory to execute an exception handler in software.

A CPU just keeps executing instructions. It does not know, nor care, about what it is doing. The difference, if any, is purely from the perspective of an outside observer. From a user's point of view, the CPU executes in a certain environment, which includes the code and data area in memory, the execution history, such as function calls and local variables of called functions, etc. which are usually maintained in the stack, and the current contents of CPU registers. These static (in memory) and dynamic (in CPU registers) information is called the context of an execution. If the CPU always executes in the same context of a single application, it is executing in single task mode. If the CPU can execute in the contexts of many different applications, it is executing in multitasking mode. In multitasking mode, the CPU executes different tasks by multiplexing its execution time among the tasks, i.e. it executes one task for a while, then it switches to execute another task, etc. If the switch is fast enough, it gives the illusion that all the tasks are executing simultaneously. This logical parallelism is called concurrency. In a MP system, tasks can execute on different CPUs in parallel. For the sake of simplicity, we shall consider multitasking in UP systems first. Multiprocessor systems will be covered later in Chapter 15.

2.1.3. System Mode and User mode

CPUs designed for multitasking usually have two different execution modes, which are represented by a mode bit in the CPU's status register. If SR.mode=0, the CPU is in system mode. If SR.mode=1, it is in user mode. While in system mode, the CPU can execute any instruction and access all the memory. While in user mode, it can not execute privileged instructions, such as I/O operations or instructions that change the SR.mode, and it can only access the memory area assigned to the application. A multitasking operating system relies on the system and user modes to separate and protect the execution environments of different tasks. Instead of two modes, some CPUs may use two bits in the SR register to provide four different modes, which form a set of protection rings, in which the innermost ring 0 is the most privileged layer, and the outermost ring 3 is the least privileged layer. CPUs with protection rings are intended to implement highly secure systems. Most Unix-like systems, e.g. Linux [Bovet, et al., 2005] and MTX, use only two modes. The operating system kernel runs in system mode and user applications run in user mode. It is very easy to switch the CPU from system mode to user mode, by simply changing SR.mode from 0 to 1. However, once in user mode, a program cannot

change the CPU's mode arbitrarily, for obvious reasons. The only way the CPU can change from user mode to system mode is by one of the following means.

(1). Exceptions or traps: when CPU encounters an exception, e.g. invalid address, illegal instruction, privilege violation, divide by 0, etc. it automatically switches to system mode to execute a trap handler routine. Trap handlers are executed in system mode because the CPU must be able to execute privileged instructions to deal with the error.
(2). Interrupts: interrupts are external signals to the CPU, requesting for CPU service. When an interrupt occurs, if the CPU is in the state of accepting interrupts, i.e. interrupts are not masked out, it switches automatically to system mode to execute an interrupt handler, which again must be in system mode because I/O operations require privileged instructions (or access protected I/O memory areas). Interrupts and interrupts processing will be cover later in Chapter 8.
(3). System calls: every CPU has special instructions which cause the CPU to switch from user mode to system mode. On the Intel x86 CPU it is the INT n instruction, where n is a byte value. A user mode program may issue INT n to explicitly request to enter system mode. Such requests are known as system calls, which are essential to operating systems. System calls will be covered in Chapter 5.

2.1.4. Memory and Memory Models

Memory may be composed of physically different memory chips. To reduce the access time, the memory organization may use one or more levels of cache memories, which are smaller but faster memory devices between the CPU and main memory. A CPU usually has a level-1 (L1) cache memory inside the CPU for caching both instructions and data. In addition, a system may also have L2 and L3 caches, etc. which are outside of the CPU. In a MP system the external caches may be shared by different CPUs. The memory subsystem uses a cache-coherence protocol to ensure that the contents of the various levels of caches are consistent. To most users, such hardware-level details are transparent. But they are not totally invisible to an operating system. For instance, in an OS kernel when process context switching occurs, the CPU's internal cache must be flushed to prevent it from executing stale instructions belonging to an old context.

2.1.5. Virtual Address and Physical Addresses

Ideally, memory should appear to the CPU as a sequence of linearly addressable locations which the CPU can read/write by specifying an address. In practice, this is often not the case. Usually, the CPU executes in the logical or virtual address (VA) space of a program, which may differ from the actual or physical address (PA) in memory. During execution, the CPU’s memory management hardware automatically translates or maps virtual addresses to physical addresses. As an example, consider the Intel x86 CPU [Antonakos, 1999]. When the Intel x86 CPU starts, it is in the so called 16-bit real mode. While in real mode, the CPU can only execute 16-bit code and access the lowest 1MB of physical memory. However, it cannot access the entire 1MB memory all at once. Instead, the CPU regards the memory as composed of four segments. A memory segment is a block of 64 KB memory that begins from a 16-byte address boundary. Since the low 4

bits of a memory segment address are always 0, it suffices to represent a segment by the high 16 bits of the segment's 20-bit real address. The CPU has four 16-bit segment registers, denoted by CS, DS, SS, ES, which point to the segments in memory the CPU is allowed to access. Thus, the CPU can access at most 4 segments or 256 KB of real memory at any time instant. A program may consist of four distinct pieces, denoted by Code, Data, Stack and Extra sections, each up to 64 KB in size. During execution each of the sections is loaded to a memory segment, which is pointed by a corresponding CPU segment register. Within a program, every address is a 16-bit VA, which is an offset in a program's section, hence an offset in the section's segment in memory. For each VA, the CPU uses the corresponding segment register to map it to a PA by

(20-bit)PA = (SegmentRegister << 4) + (16-bit)VA

where the SegmentRegister is determined either by default or by an explicit segment prefix to the instruction. When fetching instructions the CPU uses the default CS register. When accessing stack it uses the default SS register. When reading/writing data it uses the default DS register. It uses ES if the instruction is prefixed with an ES byte.

Although the x86 CPU in real mode can support programs with four segments, a binary executable program generated by a compiler-linker may further restrict the CPU's ability to access memory. Conceptually, the run-time image of a program consists of three sections; Code, Data and Stack. Ideally, each of the sections can be loaded to a separate segment in memory and pointed by a segment register, as shown in Figure 2.2.

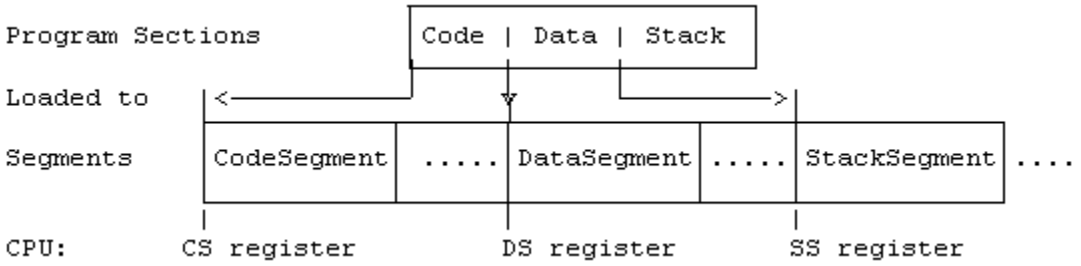

Figure 2.2. Program Sections and Memory Segments

In practice, some of the segments may overlap. For example, by default the binary executable generated by the compiler-linker of BCC [BCC] uses the one-segment memory model, in which the Code, Data and Stack sections are all the same. During execution, a one-segment program is loaded to a single memory segment and the CPU's CS, DS and SS registers must all point to that segment. Thus, the maximum size of a one-segment program is limited to 64 KB, but it can be loaded to, and executed from, any available segment in memory. In addition, BCC can also generate binary executables with separate I (instruction) and D (data) spaces. In the separate I&D memory model, the Code is a section, but the combined Data and Stack is a separate section. During execution, the Code section is loaded to a CS segment and the combined Data and Stack section is loaded to a combined DS and SS segment. BCC does not generate binary executables with separate data and stack segments due to potential ambiguity in dereferencing pointers in C. If the data and stack segments were separate, a pointer at run time may

point to either the data area or the stack. Then *p must use the right segment to access the correct memory area, but the C compiler does not know which segment to use at compile time. For this reason, BCC's compiler and linker only generate binary executables with either one-segment or separate I&D space memory model, which limits a program's virtual address space to either 64 KB or 128 KB. Translation of VA to PA in general will be covered in Chapter 7 on Memory Management.

2.1.6. I/O Devices and I/O Operations

I/O devices can be classified as block and char devices. Block devices, such as disks and CDROM, transfer data in chunks or blocks. Char devices, such as console and terminals, transfer data in bytes. There are devices, e.g. USB drives, which transfer data physically in serial form but appear as a block device. I/O operations can be classified into several modes, as shown in Figure 2.3.

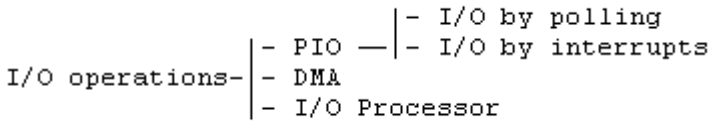

Figure 2.3. I/O Operation Types

In Programmed I/O (PIO), the CPU actively controls each I/O operation. In I/O by polling, the CPU first checks the device status. When the device is ready, it issues an I/O command to start the device. Then it repeatedly checks the device status for I/O completion. I/O by polling is suitable only for single task environment because while the CPU is doing I/O it is constantly busy and can't do anything else. In I/O by interrupts, the CPU starts an I/O operation on a device with the device's interrupt enabled. The CPU does not have to wait for the I/O operation to complete. It can proceed to do something else, e.g. to execute another task. When the I/O operation completes, the device interrupts the CPU, allowing it to decide what to do next. I/O by interrupts is well suited to multitasking. After starting an I/O operation, if the task has to wait for the I/O operation to complete, the CPU can be switched to run another task. In I/O by DMA (Direct Memory Access), the CPU writes the I/O operation information, such as the memory address, data transfer direction, number of bytes to transfer and the intended device, to a DMA controller and starts the DMA controller. Then the CPU can continue to execute. The DMA controller transfers data between memory and I/O device concurrently with CPU execution by sharing bus cycles with the CPU. When data transfer completes, the DMA controller interrupts the CPU. Responding to the interrupt, the CPU can check the I/O completion status and decide what to do next. I/O processors are either processors designed specially for I/O operations or general processors dedicated to I/O tasks. Unlike DMA controllers, which must be programmed by the CPU, I/O processors can execute complex I/O programs to do I/O operations without CPU supervision. In a computer system with multicore processors, some of the processors may be dedicated to I/O tasks rather than to general computing. This is because in such a system computational capacity is no longer the only factor that affects the system performance. Fast I/O and inter-processor communication to exchange data are equally important.

2.2. Operating Systems

An operating system (OS) is a set of programs and supporting files, which runs on a computer system to make the system easier and convenient to use. Without an operating system, a computer system is essentially useless. An OS may mean different things to different people. An average user may be only interested in how to use an OS. A software developer may be more interested in the tools and Application Program Interface (API) of an OS. A system designer may be more interested in the internal organization of an OS. Since this book is about the design and implementation of operating systems, we shall describe them from a functional point of view.

2.2.1. Major Functions of Operating Systems

The objective of an OS is to provide the following functions.

. Process management
. Memory management
. File system support
. Device drivers
. Networking

An OS supports the executions of processes. A process is a sequence of executions regarded by the OS kernel as a single entity for using system resources. System resources include memory space, I/O devices, and most importantly, CPU time. An operating system consists of a set of concurrent processes. Every activity in an OS can be attributed to a process running at that time. Process management includes process creation, process scheduling, changing process execution image, process synchronization and process termination. Thus, process management is the first major function of an OS.

Each process executes an image in memory. The execution image of a process is a memory area containing its code, data and stack. In an operating system, processes are dynamic; they come into existence when they are created and they disappear when their executions terminate. As processes come and go, their memory areas must be allocated and deallocated. During execution, a process may change its image to a different program. In systems with memory protection hardware, the OS kernel must ensure that each process can only access its own image to prevent processes from interfering with one another. All these require memory management, which is the second major function of an OS kernel.

In general, an OS kernel needs supporting files in order to run. During operation, it should allow users to save and retrieve information. In addition, it may also provide an environment for developing application programs. All these require file systems. Thus, the third major function of an OS is to provide file system support. File operations eventually require device I/O. An OS kernel must provide device drivers to support device I/O operations.

With the advent of networking technology, almost every computer is now connected to a network, such as the Internet. Although not essential, an OS should provide support for network access.

As an example, Linux is a general purpose OS, which has all the above functions. In comparison, MTX is a simple OS. It supports process management, memory management, a simple EXT2 file system and some device drivers, but it does not yet support networking at this moment.

2.3. Program Development

2.3.1 Program Development Steps

The steps of developing an executable program are as follows.

(1). Create source files: Use a text editor, such as vi or emacs, to create one or more source files of a program. In systems programming, the most important programming languages are C and assembly. We begin with C programs first. In addition to the standard comment lines in C, we shall also use // to denote comments in C code for convenience. Assume that t1.c and t2.c are the source files of a C program.

```
/********************* t1.c file ****************************/
int g = 100;                  // initialized global variable
int h;                        // uninitialized global variable
static int s;                 // static global variable
main(int argc, char *argv[ ]) // main function
{
  int a = 1; int b;           // automatic local variables
  static int c = 3;           // static local variable
  b = 2;
  c = mysum(a,b);             // call mysum(), passing a, b
  printf("sum=%d\n", c);      // call printf()
}
/********************* t2.c file ****************************/
extern int g;                 // extern global variable
int mysum(int x, int y)       // function heading
{
   return x + y + g;
}
```

Variables in C programs can be classified as global, local, static and automatic, etc. as shown in Figure 2.4.

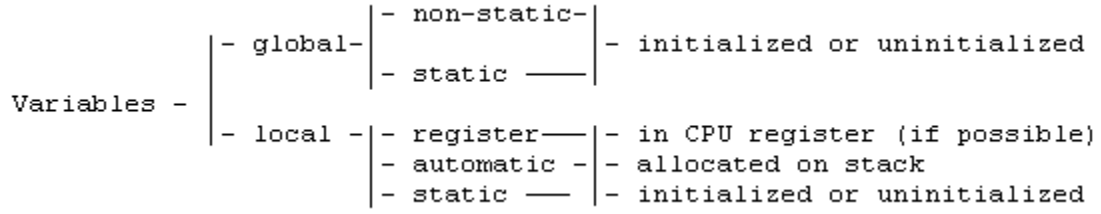

Figure 2.4. Variables in C

Global variables are defined outside of any function. Local variables are defined inside functions. Global variables are unique and have only one copy. Static globals are visible only to the file in which they are defined. Non-static globals are visible to all the files of the same program. Global variables can be initialized or uninitialized. Initialized globals are assigned values at compile time. Uninitialized globals are cleared to 0 when the program execution starts. Local variables are visible only to the function in which they are defined. By default, local variables are automatic; they come into existence when the function is entered and they logically disappear when the function exits. For register variables, the compiler tries to allocate them in CPU registers. Since automatic local variables do not have any allocated memory space until the function is entered, they cannot be initialized at compile time. Static local variables are permanent and unique, which can be initialized. In addition, C also supports volatile variables, which are used as memory-mapped I/O locations or global variables that are accessed by interrupt handlers or multiple execution threads. The volatile keyword prevents the C compiler from optimizing the code that operates on such variables.

In the above t1.c file, g is an initialized global, h is an uninitialized global and s is a static global. Both g and h are visible to the entire program but s is visible only in the t1.c file. So t2.c can reference g by declaring it as extern, but it cannot reference s because s is visible only in t1.c. In the main() function, the local variables a, b are automatic and c is static. Although the local variable a is defined as int a = 1, this is not an initialization because a does not yet exist at compile time. The generated code will assign 1 to the current copy of a when main() is actually entered.

(2). Use cc to convert the source files into a binary executable, as in

cc t1.c t2.c

which generates a binary executable file named a.out. In Linux, cc is linked to gcc, so they are the same.

(3). What's cc? cc is a program, which consists of three major steps, as shown in Figure 2.5.

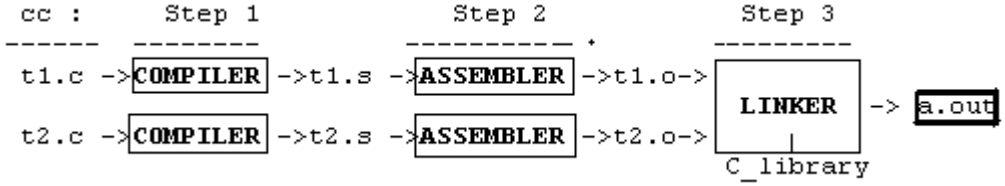

Figure 2.5. Program Development Steps

Step 1. Convert C source files to assembly code files: The first step of cc is to invoke the C COMPILER, which translates the .c files into .s files containing assembly code of the target machine. The C compiler itself has several phases, such as preprocessing, lexical analysis, parsing and code generations, etc, but the reader may ignore such details here.

Step 2. Convert assembly Code to OBJECT code: Every computer has its own set of machine instructions. Users may write programs in an assembly language for a specific machine. An ASSEMBLER is a program, which translates assembly code into machine

code in binary form. The resulting .o files are called OBJECT code. The second step of cc is to invoke the ASSEMBLER to translate .s files to .o files. Each .o file consists of

. a header containing sizes of CODE, DATA and BSS sections
. a CODE section containing machine instructions
. a DATA section containing initialized global and static local variables
. a BSS section containing uninitialized global and static local variables
. relocation information for pointers in CODE and offsets in DATA and BSS
. a Symbol Table containing non-static globals, function names and their attributes.

Step 3: LINKING: A program may consist of several .o files, which are dependent on one another. In addition, the .o files may call C library functions, e.g. printf, which are not present in the source files. The last step of cc is to invoke the LINKER, which combines all the .o files and the needed library functions into a single binary executable file. More specifically, the LINKER does the following:

. Combine all the CODE sections of the .o files into a single Code section. For C programs, the combined Code section begins with the default C startup code crt0.o, which calls main(). This is why every C program must have a unique main() function.
. Combine all the DATA sections into a single Data section. The combined Data section contains only initialized globals and static locals.
. Combine all the BSS sections into a single bss section.
. Use the relocation information in the .o files to adjust pointers in the combined Code section and offsets in the combined Data and bss sections.
. Use the Symbol Tables to resolve cross references among the individual .o files. For instance, when the compiler sees c = mysum(a,b) in t1.c, it does not know where mysum is. So it leaves a blank (0) in t1.o as the entry address of mysum but records in the symbol table that the blank must be replaced with the entry address of mysum. When the linker puts t1.o and t2.o together, it knows where mysum is in the combined Code section. It simply replaces the blank in t1.o with the entry address of mysum. Similarly for other cross referenced symbols. Since static globals are not in the symbol table, they are unavailable to the linker. Any attempt to reference static globals from different files will generate a cross reference error. Similarly, if the .o files refer to any undefined symbols or function names, the linker will also generate cross reference errors. If all the cross references can be resolved successfully, the linker writes the resulting combined file as a.out, which is the binary executable file.

2.3.2. Static vs. Dynamic Linking

There are two ways to create a binary executable, known as static linking and dynamic linking. In static linking, which uses a static library, the linker includes all the needed library function code and data into a.out. This makes a.out complete and self-contained but usually very large. In dynamic linking, which uses a shared library, the library functions are not included in a.out but calls to such functions are recorded in a.out as directives. When execute a dynamically linked a.out file, the operating system loads both

a.out and the shared library into memory and makes the loaded library code accessible to a.out during execution. The main advantages of dynamic linking are:

. The size of every a.out is reduced.
. Many executing programs can share the same library functions in memory.
. Modifying library functions does not need to re-compile the source files again.

Libraries used for dynamic linking are known as Dynamic Linking Libraries (DLLs). They are called Shared Libraries (.so files) in Linux. Dynamically loaded (DL) libraries are shared libraries which are loaded only when they are needed. DL libraries are useful as plug-ins and dynamically loaded modules.

2.3.3. Executable File Format

Although the default binary executable is named a.out, the actual file format may vary. Most C compilers and linkers can generate executable files in several different formats, which include

(1). Flat binary executable: A flat binary executable file consists only of executable code and initialized data. It is intended to be loaded into memory in its entirety for execution directly. For example, bootable operating system images are usually flat binary executables, which simplifies the boot-loader.
(2). a.out executable file: A traditional a.out file consists of a header, followed by code, data and bss sections. Details of the a.out file format will be shown in the next section.
(3). ELF executable file: An Executable and Linking Format (ELF) [ELF, 1995] file consists of one or more program sections. Each program section can be loaded to a specific memory address. In Linux, the default binary executables are ELF files, which are better suited to dynamic linking.

2.3.4. Contents of a.out File

For the sake of simplicity, we consider the traditional a.out files first. ELF executables will be covered later in Chapters 14 and 15 when we discuss MTX in 32-bit protected mode. An a.out file consists of the following sections:

(1). header: the header contains loading information and sizes of the a.out file, where
tsize = size of Code section;
dsize = size of Data section containing initialized globals and static locals;
bsize = size of bss section containing uninitialized globals and static locals;
total_size = total size of a.out to load.
(2). Code Section: also called the text section, which contains executable code of the program. It begins with the standard C startup code crt0.o, which calls main().
(3). Data Section: The Data section contains initialized global and static data.
(4). symbol table: optional, needed only for run-time debugging.

Note that the bss section, which contains uninitialized global and static local variables, is not in the a.out file. Only its size is recorded in the a.out file header. Also, automatic local variables are not in a.out. Figure 2.6 shows the layout of an a.out file.

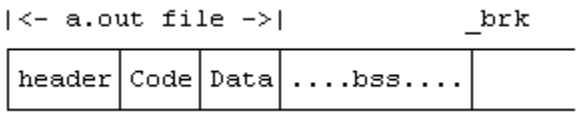

Figure 2.6. Contents of a.out file

where _brk is a symbolic mark indicating the end of the bss section. The total loading size of a.out is usually equal to _brk, i.e. equal to tsize+dsize+bsize. If desired, _brk can be set to a higher value for a larger loading size. The extra memory space above the bss section is the HEAP area for dynamic memory allocation during execution.

2.3.5. Program Execution

Under a Unix-like operating system, the sh command

a.out one two three

executes a.out with the token strings as command-line parameters. To execute the command, sh forks a child process and waits for the child to terminate. When the child process runs, it uses a.out to create a new execution image by the following steps.

(1). Read the header of a.out to determine the total memory size needed:

TotalSize = _brk + stackSize

where stackSize is usually a default value chosen by the OS kernel for the program to start. There is no way of knowing how much stack space a program will ever need. For example, the trivial C program

main(){ main(); }

will generate a segmentation fault due to stack overflow on any computer. So the usual approach of an OS kernel is to use a default initial stack size for the program to start and tries to deal with possible stack overflow later during run-time.

(2). It allocates a memory area of TotalSize for the execution image. Conceptually, we may assume that the allocated memory area is a single piece of contiguous memory. It loads the Code and Data sections of a.out into the memory area, with the stack area at the high address end. It clears the bss section to 0, so that all uninitialized globals and static locals begin with the initial value 0. During execution, the stack grows downward toward low address.

(3). Then it abandons the old image and begins to execute the new image, which is shown in Figure 2.7.

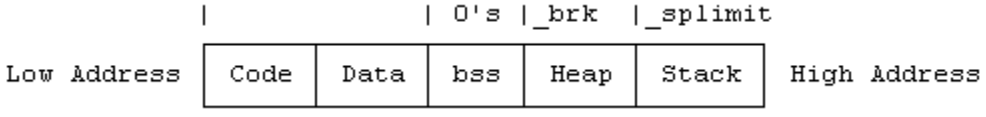

Figure 2.7. Execution Image

In Figure 2.7, _brk at the end of the bss section is the program's initial "break" mark and _splimit is the stack size limit. The Heap area between bss and Stack is used by the C library functions malloc()/free() for dynamic memory allocation in the execution image. When a.out is first loaded, _brk and _splimit may coincide, so that the initial Heap size is zero. During execution, the process may use the brk(address) or sbrk(size) system call to change _brk to a higher address, thereby increasing the Heap size. Alternatively, malloc() may call brk() or sbrk() implicitly to expand the Heap size. During execution, a stack overflow occurs if the program tries to extend the stack pointer below _splimit. On machines with memory protection, this will be detected by the memory management hardware as an error, which traps the process to the OS kernel. Subject to a maximal size limit, the OS kernel may grow the stack by allocating additional memory in the process address space, allowing the execution to continue. A stack overflow is fatal if the stack cannot be grown further. On machines without suitable hardware support, detecting and handling stack overflow must be implement in software.
(4). Execution begins from crt0.o, which calls main(), passing as parameters argc and argv to main(int argc, char *argv[ ]), where argc = number of command line parameters and each argv entry points to a corresponding command line parameter string.

2.3.6. Program Termination

A process executing a.out may terminate in two possible ways.

(1). Normal Termination: If the program executes successfully, main() eventually returns to crt0.o, which calls the library function exit(0) to terminate the process. The exit(value) function does some clean-up work first, such as flush stdout, close I/O streams, etc. Then it issues an _exit(value) system call, which causes the process to enter the OS kernel to terminate. A 0 exit value usually means normal termination. If desired, a process may call exit(value) directly without going back to crt0.o. Even more drastically, a process may issue an _exit(value) system call to terminate immediately without doing the clean-up work first. When a process terminates in kernel, it records the value in the _exit(value) system call as the exit status in the process structure, notifies its parent and becomes a ZOMBIE. The parent process can find the ZOMBIE child, get its pid and exit status by the pid = wait(int *status);
system call, which also releases the ZMOBIE child process structure as FREE, allowing it to be reused for another process.

(2). Abnormal Termination: While executing a.out the process may encounter an error condition, which is recognized by the CPU as an exception. When a process encounters an exception, it is forced into the OS kernel by a trap. The kernel’s trap handler converts the trap error type to a magic number, called a SIGNAL, and delivers the signal to the process, causing it to terminate. In this case, the exit status of the ZOMBIE process is the signal number, and we may say that the process has terminated abnormally. In addition to trap errors, signals may also originate from hardware or from other processes. For example, pressing the Control_C key generates a hardware interrupt, which sends the signal number SIGINT(2) to all processes on that terminal, causing them to terminate. Alternatively, a user may use the command

```
kill  -s signal_number pid        # signal_number=1 to 31
```

to send a signal to a target process identified by pid. For most signal numbers, the default action of a process is to terminate. Signals and signal handling will be covered later in Chapter 9.

2.4. Function Call in C

Next, we consider the run-time behavior of a.out during execution. The run-time behavior of a program stems mainly from function calls, which use the stack. The following discussions apply to running C programs on both 16-bit and 32-bit Intel x86 processors. On these machines, the C compiler generated code passes parameters on the stack in function calls. During execution, it uses a special CPU register (bp or ebp) to point at the stack frame of the current executing function. In 64-bit processors, which have more registers, the function call convention differs slightly. Some parameters are passed in specific registers. A called function may use the stack pointer itself as the stack frame pointer. Despite these minor differences, the same principle still applies.

2.4.1. Run-time Stack Usage

Consider the following C program, which consists of a main() function shown on the left-hand side, which calls a sub() function shown on the right-hand side.

```
-----------------------------------------------------------
  main()                      |     int sub(int x, int y)
  {                           |     {
    int a, b, c;              |         int u, v;
    a = 1; b = 2; c = 3;      |         u = 4; v = 5;
    c = sub(a, b);            |         return x+y+u+v;
    printf("c=%d\n", c);      |     }
  }                           |
-----------------------------------------------------------
```

(1). When executing a.out, a process image is created in memory, which looks (logically) like the diagram shown in Figure 2.8, where Data includes both initialized data and bss.

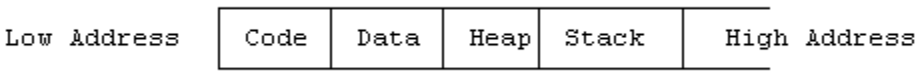

Figure 2.8. Process Execution Image

(2). Every CPU has the following registers or equivalent, where the entries in parentheses denote registers of the x86 CPU:

PC (IP): point to next instruction to be executed by the CPU.
SP (SP): point to top of stack.
FP (BP): point to the stack frame of current active function.
Return Value Register (AX): register for function return value.

(3). In every C program, main() is called by the C startup code crt0.o. When crt0.o calls main(), it pushes the return address (the current PC register) onto stack and replaces PC with the entry address of main(), causing the CPU to enter main(). For convenience, we shall show the stack contents from left to right. When control enters main(), the stack contains the saved return PC on top, as shown in Figure 2.9.

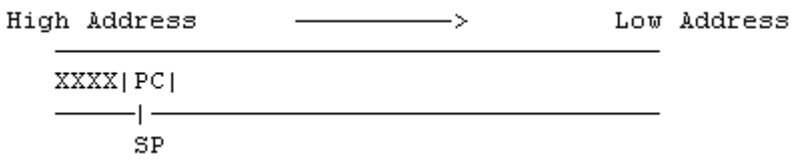

Figure 2.9. Stack Contents in Function Call

where XXXX denotes the stack contents before crt0.o calls main(), and SP points to the saved return PC from where crt0.o calls main().

(4). Upon entry, the compiled code of every C function does the following:

. push FP onto stack # this saves the CPU's FP register on stack.
. let FP point at the saved FP # establish stack frame
. shift SP downward to allocate space for automatic local variables on stack
. the compiled code may shift SP farther down to allocate some scratch space.

For this example, there are 3 automatic local variables, int a, b, c, each of sizeof(int) bytes. After entering main(), the stack contents becomes as shown in Figure 2.10, in which the spaces of a, b, c are allocated but their contents are yet undefined.

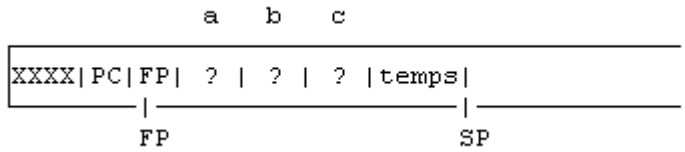

Figure 2.10. Stack Contents: Allocate Local Variables

(5). Then the CPU starts to execute the code a=1; b=2; c=3; which put the values 1, 2, 3 into the memory locations of a, b, c, respectively. Assume that sizeof(int) is 4 bytes. The locations of a, b, c are at -4, -8, -12 bytes from where FP points at. These are expressed as -4(FP), -8(FP), -12(FP) in assembly code, where FP is the stack frame pointer. For example, in 32-bit Linux the assembly code for b=2 in C is movl $2, -8(%ebp), where $2 means the value of 2 and %ebp is the ebp register.

(6). main() calls sub() by c = sub(a, b); The compiled code of the function call consists of

. Push parameters in reverse order, i.e. push values of b=2 and a=1 into stack.
. Call sub, which pushes the current PC onto stack and replaces PC with the entry address of sub, causing the CPU to enter sub().

When control first enters sub(), the stack contains a return address at the top, preceded by the parameters, a, b, of the caller, as shown in Figure 2.11.

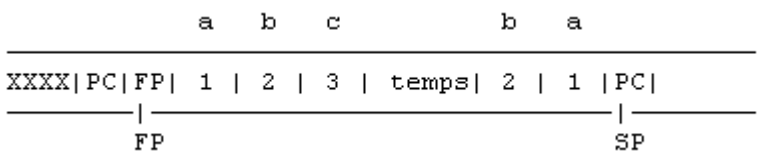

Figure 2.11. Stack Contents: Passing Parameters

(7). Since sub() is written in C, it actions are exactly the same as that of main(), i.e. it

. Push FP and let FP point at the saved FP;
. Shift SP downward to allocate space for local variables u, v.
. The compiled code may shift SP farther down for some temp space on stack.

The stack contents becomes as shown in Figure 2.12.

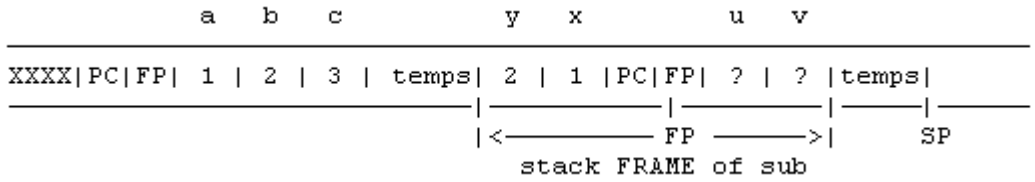

Figure 2.12. Stack Contents of Called Function

2.4.2. Stack Frames

While execution is inside a function, such as sub(), it can only access global variables, parameters passed in by the caller and local variables, but nothing else. Global and static local variables are in the combined Data section, which can be referenced by a fixed base register. Parameters and automatic locals have different copies on each invocation of the function. So the problem is: how to reference parameters and automatic locals? For this example, the parameters a, b, which correspond to the arguments x, y, are at 8(FP) and 12(FP). Similarly, the automatic local variables u, v are at -4(FP) and -8(FP). The stack area visible to a function, i.e. parameters and automatic locals, is called the stack FRAME of a function, like a frame of movie to a person. Thus, FP is called the Stack Frame Pointer. To a function, the stack frame looks like the following.

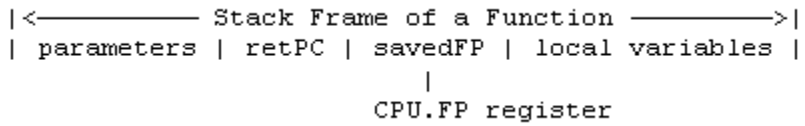

Figure 2.13. Function Stack Frame

From the above discussions, the reader should be able to deduce what would happen if we have a sequence of function calls, e.g.

crt0.o --> main() --> A(par_a) --> B(par_b) --> C(par_c)

For each function call, the stack would grow (toward low address) one more frame for the called function. The frame at the stack top is the stack frame of the current executing function, which is pointed by the CPU's frame pointer. The saved FP points (backward) to the frame of its caller, whose saved FP points back at the caller's caller, etc. Thus, the function call sequence is maintained in the stack as a link list, as shown in Figure 2.14.

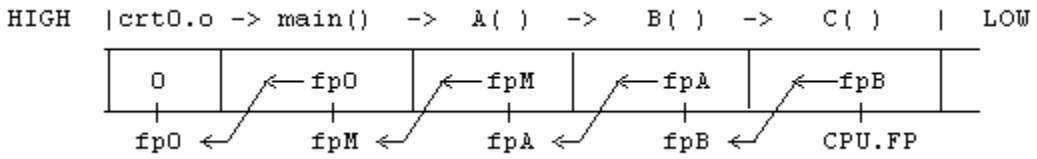

Figure 2.14. Function Call Sequence

By convention, the CPU's FP = 0 when crt0.o is entered from the OS kernel. So the stack frame link list ends with a 0. When a function returns, its stack frame is deallocated and the stack shrinks back.

2.4.3. Return From Function Call

When sub() executes return x+y+u+v, it evaluates the expression and puts the resulting value in the return value register (AX). Then it deallocates the local variables by

```
.copy FP into SP;    # SP now points to the saved FP in stack.
.pop stack into FP;  # this restores FP, which now points to the caller's stack frame,
                     # leaving the return PC on the stack top.
 (On the x86 CPU, the above operations are equivalent to the leave instruction).
.Then, it executes the RET instruction, which pops the stack top into PC register,
 causing the CPU to execute from the saved return address of the caller.
```

(8). Upon return, the caller function catches the return value in the return register (AX). Then it cleans the parameters a, b, from the stack (by adding 8 to SP). This restores the stack to the original situation before the function call. Then it continues to execute the next instruction.

It is noted that some compilers, e.g. GCC Version 4, allocate automatic local variables in increasing address order. For instance, int a, b; implies (address of a) < (address of b). With this kind of allocation scheme, the stack contents may look like the following.

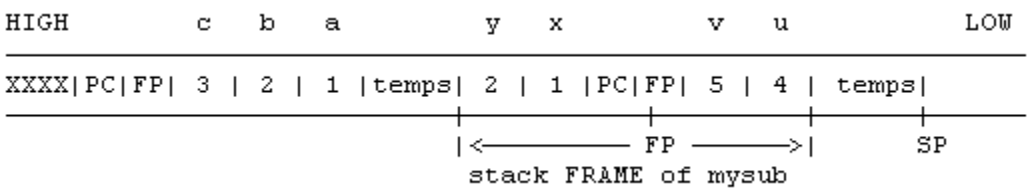

Figure 2.15. Stack Contents with Reversed Allocation Scheme

In this case, automatic local variables are also allocated in "reverse order", which makes them consistent with the parameter order, but the concept and usage of stack frames remain the same.

2.4.4. Long Jump

In a sequence of function calls, such as

main() --> A() --> B()-->C();

when a called function finishes, it normally returns to the calling function, e.g. C() returns to B(), which returns to A(), etc. It is also possible to return directly to an earlier function in the calling sequence by a long jump. The following program demonstrates long jump in Unix/Linux.

```
/***** longjump.c demonstrating long jump in Linux *****/
#include <stdio.h>
#include <setjmp.h>
jmp_buf env;       // for saving longjmp environment
main()
{
  int r, a=100;
  printf("call setjmp to save environment\n");
  if ((r=setjmp(env)) == 0){
     A();
     printf("normal return\n");
  }
  else
     printf("back to main() via long jump, r=%d a=%d\n", r, a);
}
int A()
{   printf("enter A()\n");
    B();
    printf("exit A()\n");
}
int B()
{
  printf("enter B()\n");
  printf("long jump? (y|n) ");
  if (getchar()=='y')
     longjmp(env, 1234);
  printf("exit B()\n");
}
```

In the above program, setjmp() saves the current execution environment in a jmp_buf structure and returns 0. The program proceeds to call A(), which calls B(). While in the function B(), if the user chooses not to return by long jump, the functions will show the normal return sequence. If the user chooses to return by longjmp(env, 1234), execution will return to the last saved environment with a nonzero value. In this case, it causes B() to return to main() directly, bypassing A(). The principle of long jump is very simple. When a function finishes, it returns by the (callerPC, callerFP) in the current stack frame, as shown in figure 2.16.

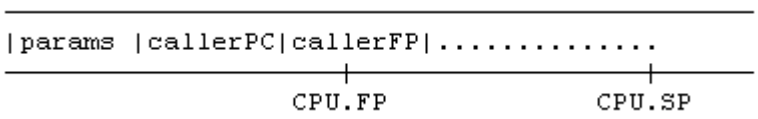

Figure 2.16. Function Return Frame

If we replace (callerPC, callerFP) with (savedPC, savedFP) of an earlier function in the calling sequence, execution would return to that function directly. In addition to the (savedPC, savedFP), setjmp() may also save CPU's general registers and the original SP, so that longjmp() can restore the complete environment of the returned function. Long jump can be used to abort a function in a calling sequence, causing execution to resume from a known environment saved earlier. Although rarely used in user mode programs, it is a common technique in systems programming. For example, it may be used in a signal catcher to bypass a user mode function that caused an exception or trap error. We shall demonstrate this technique later in Chapter 9 on signals and signal processing.

2.5. Link C Program with Assembly Code

In systems programming, it is often necessary to access the hardware, such as CPU registers and I/O port locations, etc. In these situations, assembly code becomes necessary. It is therefore important to know how to link C programs with assembly code. We illustrate the linking process by an example. The following program consists of a tc.c file in C and a ts.s file in assembly.

```
/******************* tc.c file ************************/
extern int g;        // g is extern defined in .s file
int h;               // global h, used in .s file
main( )
{
  int a,b,c,*bp;      // locals of main()
  g = 100;            // use g in .s file
  bp = getbp();       // call getbp() in .s file
  a = 1; b = 2; h=200;
  c = mysum(a,b);     // call mysum() in .s file
  printf("a=%d b=%d c=%d\n", a,b,c);
}
!------------------- ts.s file ----------------------
       .global _h               ! IMPORT _h from C
       .global _getbp,_mysum,_g ! EXPORT global symbols to C
_getbp:                         ! int getbp() function
        mov  ax,bp
        ret
_mysum:                         ! int mysum(int x, int y)
        push bp
        mov  bp,sp              ! establish stack frame
        mov  ax,4[bp]           ! AX = x
        add  ax,6[bp]           ! add y to AX
        add  ax,_h              ! add _h to AX
        mov  sp,bp              ! return to caller
        pop  bp
        ret
_g:     .word 1234              ! global _g defined here
```

The assembly code syntax is that of BCC's as86 assembler. When generating object code BCC's compiler prefixes every identifier with an underscore, e.g. main() becomes _main and getbp() becomes _getbp, etc. BCC's assembler uses the same naming convention. An assembly program may import/export global symbols from/to C code by .global statements. Every global symbol in the assembly code must have an underscore prefix. Likewise, a C program may reference global symbols in assembly by declaring them as extern. In the C code, g is declared as an extern integer. It is the same global symbol _g in the assembly code. Similarly, the global variable h is the same _h in assembly code. In C, function names are global and the function types are int by default. If a function returns a different type, it must be either defined first or declared by a function prototype before it is used. In the C code, the statement g=100 actually changes the contents of _g in the assembly code. When it calls getbp() in assembly, the returned value is the CPU's bp register. When calling mysum(a, b) it pushes the values of b and a onto stack as parameters. Upon entry to _mysum: the assembly code first establishes the stack frame. It uses the stack frame pointer (bp) to access the parameters a, b, which are at 4[bp] and 6[bp], respectively (in BCC integers are 2 bytes). In every function call, the return value is in the AX register. Under Linux, use BCC to compile and link the C and assembly files.

```
bcc -c -ansi tc.c ts.s     # generate tc.o and ts.o
ld86 crt0.o tc.o ts.o mtxlib /usr/lib/bcc/libc.a
```

where crt0.o is the C startup code and mtxlib contains I/O and system call interface to the MTX kernel. The resulting a.out can be executed under MTX.

2.6. Link Library

A link library contains precompiled object code. During linking, the linker searches the link libraries for any function code and data needed by a program. Link libraries used by the BCC linker are standard Unix archive files, which can be manipulated by the ar utility program. The following shows how to create and maintain a link library for use in MTX.

```
(1). Assume: getbp.s is an assebmly code file.
     !--------- getbp.s file -----------
                .globl _getbp  ! OR .global _getbp
        _getbp:  mov ax,bp
                 ret
(2). as86 –o getbp.o getbp.s   # generate getbp.o file
(3). ar r mylib getbp.o # create mylib and add the member getbp.o

(4). Assume: mysum.c is a C code file.
      /*********** mysum.c file **************/
      int mysum(int x, int y){  return x+y;  }
(5). bcc -c –ansi mysum.c    # generate mysum.o file
(6). ar r mylib mysum.o      # add member mysum.o to mylib
(7)  ar t mylib              # list members of mylib
```

Similarly, we may add/delete members to/from an existing library by the ar command. Link libraries are very useful in system development. As we expand a system, we may

include some of the existing parts of the system into a link library and focus our attention on developing new features of the system.

2.7. Mailman's Algorithm

In computer systems, a problem which arises very often is as follows. A city has M blocks, numbered 0 to M-1. Each block has N houses, numbered 0 to N-1. Each house has a unique block address, denoted by (block, house), where 0 <= block < M, 0 <= house < N. An alien from outer space may be unfamiliar with the block addressing scheme on Earth and prefers to address the houses linearly as 0,1,..N-1,N, N+1 ....., etc. Given a block address BA = (block, house), how to convert it to a linear address LA, and vice versa? If everything counts from 0, the conversion is very simple.

```
Linear_address LA = N*block + house;
Block_address  BA = (LA / N, LA % N);
```

Note that the conversion is valid only if everything counts from 0. If some of the items do not count from 0, they can not be used in the conversion formula directly. The reader may try to figure out how to handle such cases in general. For ease of reference, we shall refer to the conversion method as the Mailman's algorithm.

2.7.1. Applications of Mailman's Algorithm

(1). Test, Set and Clear bits in C: In standard C programs, the smallest addressable unit is a char or byte. It is often necessary to manipulate bits in a bitmap, which is a sequence of bits. Consider char buf[1024], which has 1024 bytes, denoted by buf[i], i=0, 1,…., 1023. It also has 8192 bits numbered 0,1,2,....8191. Given a bit number BIT, e.g. 1234, which byte i contains the bit, and which bit j is it in that byte? Solution:

```
i = BIT / 8;   j = BIT % 8;   // 8 = number of bits in a byte.
```

This allows us to combine the Mailman’s algorithm with bit masking to do the following bit operations in C.

```
.TST a bit for 1 or 0 :  if (buf[i] &   (1 << j))
.SET a bit to 1       :      buf[i] |=  (1 << j);
.CLR a bit to 0       :      buf[i] &= ~(1 << j);
```

It is noted that some C compilers allow specifying bits in a structure, as in

```
struct bits{
  unsigned int bit0      :  1;  // bit0 field is a single bit
  unsigned int bit123    :  3;  // bit123 field is a range of 3 bits
  unsigned int otherbits : 27;  // other bits field has 27 bits
  unsigned int bit31     :  1;  // bit31 is the highest bit
}var;
```

The structure defines var as an unsigned 32-bit integer with individual bits or ranges of bits. Then, var.bit0 = 0; assigns 1 to bit 0, and var.bit123 = 5; assigns 101 to bits 1 to 3, etc. However, the generated code still relies on the Mailman's algorithm and bit masking

to access the individual bits. The Mailman's algorithm allows us to manipulate bits in a bitmap directly without defining complex C structures.

(2). Convert INODE number to inode position on disk: In an EXT2 file system, each file has a unique INODE structure. On the file system disk, inodes begin in the inode_table block. Each disk block contains

```
INODES_PER_BLOCK = BLOCK_SIZE/sizeof(INODE)
```

inodes. Each inode has a unique inode number, ino = 1, 2, ..., counted linearly from 1. Given an ino, e.g. 1234, determine which disk block contains the inode and which inode is it in that block? We need to know the disk block number because read/write a real disk is by blocks. Solution:

```
block = (ino - 1) / INODES_PER_BLOCK + inode_table;
inode = (ino - 1) % INODES_PER_BLOCK;
```

Similarly, converting double and triple indirect logical block numbers to physical block numbers in an EXT2 file system also depends on the Mailman's algorithm.

(3). Convert linear disk block number to CHS = (cylinder, head, sector) format: Floppy disk and old hard disk use CHS addressing but file systems always use linear block addressing. The algorithm can be used to convert a disk block number to CHS when calling BIOS INT13.

2.8. EXT2 File System

For many years, Linux used EXT2 [Card et al.] as the default file system. EXT3 [ETX3] is an extension of EXT2. The main addition in 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. EXT4 file system will be covered in Chapter 3 when we develop booting programs for hard disks containing EXT4 file systems. MTX is a small operating system intended mainly for teaching. 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, MTX uses ETX2 as the file system. Support for other file systems, e.g. DOS and iso9660 [ECMA-119, 1987], are not implemented in the MTX kernel. If needed, they can be implemented as user-level utility programs.

2.8.1 EXT2 File System Data Structures

Under MTX, the command, mkfs device nblocks [ninodes], creates an EXT2 file system on a specified device with nblocks blocks of 1KB block size. The device can be either a real device or a virtual disk file. If the number of ninodes is not specified, the default number of inodes is nblocks/4. The resulting file system is totally compatible with

the EXT2 file system of Linux. As a specific example, mkfs /dev/fd0 1440 makes an EXT2 file system on a 1.44MB floppy disk with 1440 blocks and 360 inodes. Instead of a real device, the target can also be a disk image file. The layout of such an EXT2 file system is shown in Figure 2.17.

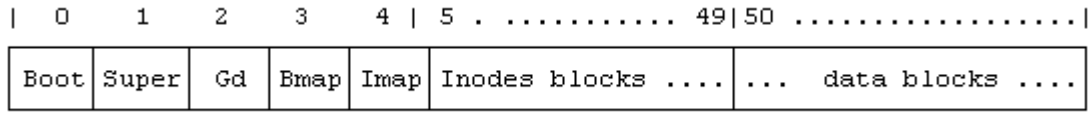

Figure 2.17. Simple EXT2 File System Layout

For ease of discussion, we shall assume this basic file system layout first. Whenever appropriate, we point out the variations, including those in large EXT2/3 FS on hard disks. The following briefly explains the contents of the disk blocks.

Block#0: Boot Block: B0 is the boot block, which is not used by the file system. It contains a booter program for booting up an OS from the disk.
Block#1: Superblock: (at byte offset 1024 in hard disk partitions): B1 is the superblock, which contains information about the entire file system. Some of the important fields of the superblock structure are shown below.

```
struct ext2_super_block {
  u32  s_inodes_count;        // total number of inodes
  u32  s_blocks_count;        // total number of blocks
  u32  s_r_blocks_count;
  u32  s_free_blocks_count;   // current number of free blocks
  u32  s_free_inodes_count;   // current number of free inodes
  u32  s_first_data_block;    // first data block: 1 for FD, 0 for HD
  u32  s_log_block_size;      // 0 for 1KB block, 2 for 4KB block
  u32  s_log_frag_size;       // not used
  u32  s_blocks_per_group;    // number of blocks per group
  u32  s_frags_per_group;     // not used
  u32  s_inodes_per_group;
  u32  s_mtime, s_wtime;
  u16  s_mnt_count;           // number of times mounted
  u16  s_max_mnt_count;       // mount limit
  u16  s_magic;               // 0xEF53
  // MORE non-essential fileds,
  u16 s_inode_size=256 bytes for EXT4
};
```

Block#2: Group Descriptor Block (in s_first_data_block+1 on hard disk): EXT2 divides disk blocks into groups. Each group contains 8192 (32K on HD) blocks. Each group is described by a group descriptor structure.

```
struct ext2_group_desc
{
  u32  bg_block_bitmap;        // Bmap block number
  u32  bg_inode_bitmap;        // Imap block number
  u32  bg_inode_table;         // Inodes begin block number
  u16  bg_free_blocks_count;   // THESE are OBVIOUS
  u16  bg_free_inodes_count;
```

```
  u16  bg_used_dirs_count;
  u16  bg_pad;                 // ignore these
  u32  bg_reserved[3];
};
```

Since a FD has only 1440 blocks, B2 contains only 1 group descriptor. The rest are 0's. On a hard disk with a large number of groups, group descriptors may span many blocks. The most important fields in a group descriptor are bg_block_bitmap, bg_inode_bitmap and bg_inode_table, which point to the group's blocks bitmap, inodes bitmap and inodes start block, respectively. Here, we assume the bitmaps are in blocks 3 and 4, and inodes start from block 5.

Block#3: Block Bitmap (Bmap): (bg_block_bitmap): A bitmap is a sequence of bits used to represent some kind of items, e.g. disk blocks or inodes. A bitmap is used to allocate and deallocate items. In a bitmap, a 0 bit means the corresponding item is FREE, and a 1 bit means the corresponding item is IN_USE. A FD has 1440 blocks but block#0 is not used by the file system. So the Bmap has only 1439 valid bits. Invalid bits are treated as IN_USE and set to 1's.

Block#4: Inode Bitmap (Imap) (bg_inode_bitmap): An inode is a data structure used to represent a file. An EXT2 file system is created with a finite number of inodes. The status of each inode is represented by a bit in the Imap in B4. In an EXT2 FS, the first 10 inodes are reserved. So the Imap of an empty EXT2 FS starts with ten 1's, followed by 0's. Invalid bits are again set to 1's.

Block#5: Inodes (begin) Block (bg_inode_table): Every file is represented by a unique inode structure of 128 (256 in EXT4) bytes. The essential inode fields are listed below.

```
struct ext2_inode {
  u16  i_mode;         // 16 bits = |tttt|ugs|rwx|rwx|rwx|
  u16  i_uid;          // owner uid
  u32  i_size;         // file size in bytes
  u32  i_atime;        // time fields in seconds
  u32  i_ctime;        // since 00:00:00,1-1-1970
  u32  i_mtime;
  u32  i_dtime;
  u16  i_gid;          // group ID
  u16  i_links_count;  // hard-link count
  u32  i_blocks;       // number of 512-byte sectors
  u32  i_flags;        // IGNORE
  u32  i_reserved1;    // IGNORE
  u32  i_block[15];    // See details below
  u32  i_pad[7];
}
```

In the inode structure, i_mode specifies the file's type, usage and permissions. For example, the leading 4 bits =1000 for REG file, 0100 for DIR, etc. The last 9 bits are the rwx permission bits for file protection. The i_block[15] array contains disk blocks of a file, which are

Direct blocks: i_block[0] to i_block[11], which point to direct disk blocks.

Indirect blocks: i_block[12] points to a disk block, which contains 256 block numbers, each of which points to a disk block.
Double Indirect blocks: i_block[13] points to a block, which points to 256 blocks, each of which points to 256 disk blocks.
Triple Indirect blocks: i_block[14] is the triple-indirect block. We may ignore this for "small" EXT2 FS.

The inode size (128 or 256) divides block size (1KB or 4KB) evenly, so that every inode block contains an integral number of inodes. In the simple EXT2 file system, the number of inode blocks is equal to ninodes/8. For example, if the number of inodes is 360, which needs 45 blocks, the inode blocks include B5 to B49. Each inode has a unique inode number, which is the inode's position in the inode blocks plus 1. Note that inode positions count from 0, but inode numbers count from 1. A 0 inode number means no inode. The root directory's inode number is 2.
Data Blocks: Immediately after the inode blocks are data blocks. Assuming 360 inodes, the first real data block is B50, which is i_block[0] of the root directory /.

EXT2 Directory Entries: A directory contains dir_entry structures, in which the name field contains 1 to 255 chars. So the dir_entry's rec_len also varies.

```
struct ext2_dir_entry_2 {
   u32 inode;               // inode number; count from 1, NOT 0
   u16 rec_len;             // this entry's length in bytes
   u8  name_len;            // name length in bytes
   u8  file_type;           // not used
   char name[EXT2_NAME_LEN]; // name: 1-255 chars, no NULL byte
};
```

2.8.2. Traverse EXT2 File System Tree

Given an EXT2 file system and the pathname of a file, e.g. /a/b/c, the problem is how to find the file. To find a file amounts to finding its inode. The algorithm is as follows.

(1). Read in the superblock, which is at the byte offset 1024. Check the magic number s_magic (0xEF53) to verify it's indeed an EXT2 FS.
(2). Read in the group descriptor block (1 + s_first_data_block) to access the group 0 descriptor. From the group descriptor's bg_inode_table entry, find the inodes begin block number, call it the InodesBeginBlock.
(3). Read in InodeBeginBlock to get the inode of /, which is INODE #2.
(4). Tokenize the pathname into component strings and let the number of components be n. For example, if pathname=/a/b/c, the component strings are "a", "b", "c", with n=3. Denote the components by name[0], name[1], .. ,name[n-1].
(5). Start from the root INODE in (3), search for name[0] in its data block(s). For simplicity, we may assume that the number of entries in a DIR is small, so that a DIR inode only has 12 direct data blocks. With this assumption, it suffices to search the 12 direct blocks for name[0]. Each data block of a DIR INODE contains dir_entry structures of the form

[ino rlen nlen NAME] [ino rlen nlen NAME] .....

where NAME is a sequence of nlen chars (without a terminating NULL char). For each data block, read the block into memory and use a dir_entry *dp to point at the loaded data block. Then use nlen to extract NAME as a string and compare it with name[0]. If they do not match, step to the next dir_entry by

dp = (dir_entry *)((char *)dp + dp->rlen);

and continue the search. If name[0] exists, we can find its dir_entry and hence its inode number.

(6). Use the inode number, ino, to locate the corresponding INODE. Recall that ino counts from 1. Use the Mailman's algorithm to compute the disk block containing the INODE and its offset in that block.

blk = (ino - 1) / INODES_PER_BLOCK + InodesBeginBlock;
offset = (ino - 1) % INODES_PER_BLOCK;

Then read in the INODE of /a, from which we can determine whether it's a DIR. If /a is not a DIR, there can't be /a/b, so the search fails. If it's a DIR and there are more components to search, continue for the next component name[1]. The problem now becomes: search for name[1] in the INODE of /a, which is exactly the same as that of Step (5).

(7). Since Steps 5-6 will be repeated n times, it's better to write a search function

```
u32 search(INODE *inodePtr, char *name)
{
   // search for name in the data blocks of this INODE
   // if found, return its ino; else return 0
}
```

Then all we have to do is to call search() n times, as sketched below.

```
Assume: n,  name[0], ...., name[n-1] are globals
INODE *ip points at INODE of /
for (i=0; i<n; i++){
    ino = search(ip, name[i])
    if (!ino){ // can't find name[i], exit;}
    use ino to read in INODE and let ip point to INODE
}
```

If the search loop ends successfully, ip must point at the INODE of pathname. Traversing large EXT2 FS with many groups is similar, which will be shown in Chapter 3 when we discuss booting from hard disk partitions.

2.9. The BCC Cross-Compiling Package

The initial development platform of MTX is BCC under Linux. BCC consists of an assembler, a C compiler and a linker. It is a cross compiling package, which generates 16-bit code for execution on real-mode PC or PC emulators. The latest version of BCC is bcc-0.16.17. Currently, BCC is included in most Linux distributions. If not, it can be downloaded and installed easily. As usual, the bcc command accepts both .s and .c source files and invokes the assembler, compiler and linker to generate a binary executable a.out. The resulting a.out is intended for execution on ELKS (Embedded Linux Kernel System)

[ELKS] or an ELKS simulator. To adapt BCC to the development of MTX, we must run BCC's assembler, compiler and linker in separate steps. Currently, MTX does not have a program development facility of its own. Since most readers have access to Linux and are familiar with its working environment, it is better to use Linux as the development platform of MTX. Usage of the BCC package will be explained in Chapter 3 when we discuss booting. In addition, we also use BCC to develop MTX in 16-bit real mode in Chapters 4 to 13. We shall switch to GCC's 32-bit assembler, compiler and linker when we extend MTX to 32-bit protected mode in Chapters 14 and 15.

2-10. Running MTX

The system image of MTX is an EXT2 file system containing the following contents.

```
      |-- bin   : binary executable programs
      |-- dev   : device special files
 /----|-- etc   : passwd file
      |-- user  : user home directories
      |-- boot  : mtx kernel images
```

There are two types of MTX images.

(1). FDimage: These are small MTX systems, which run on (virtual) floppy disks. These include all the MTX systems developed in Chapters 4 and 5. Each FDimage is a bootable floppy disk image, which can be used as the FD of a virtual machine. For example, to run MTX on a FDimage under QEMU, enter

```
qemu -fda FDimage -no-fd-bootchk
```

To run it on other virtual machines, such as DOSEMU, VMware and VirtualBox, etc., configure the virtual machines to boot from FDimage as the virtual FD disk.

(2). HDimage: These are full-sized MTX images intended for hard disks. In order to run an HDimage, it must be installed to a hard disk partition, along with a MTX hard disk booter. The install procedure is described in the Appendix. After installation, boot up and run MTX as usual.

2.11. From login to Command Execution

This section describes the login process of MTX. Although some of the descriptions are MTX specific, they are also applicable to all Unix-like systems. The first step of running MTX is to boot up the MTX kernel (Booting is covered in Chapter 3). When the MTX kernel starts, it initializes the system and mounts a root file system from the boot device. Then it creates a process P0, which runs only in kernel mode. P0 forks a child process P1, which executes an INIT program in user mode. P1 plays the same role as the INIT process of Unix/Linux. When P1 runs, it forks a child process on each of the login terminals. Then P1 waits for any of the login process to terminate. Each login process executes the same login program, in which it opens its own terminal (special file) to get the file descriptors in=0 for read, out=1 and err=2 for write. Then it displays login: to its

own terminal and waits for users to login. When a user tries to login, the login process checks the user's login name and password in the /etc/passwd file. Each line of the passwd file is of the form

loginname:password:gid:uid:user-full-name:home-directory:program

After verifying that the user has a valid account, the login process becomes the user's process by acquiring the user's gid and uid. It changes directory to the user's home-directory and executes the listed program, which is usually the command interpreter sh. Then the user can enter commands for the sh process to execute. When the user logout, the sh process terminates, which wakes up the INIT process P1, which forks another login process on the terminal, etc.

A command is usually a binary executable file. By default, all executable programs are in the /bin directory. If an executable file is not in the /bin directory, it must be entered as a full pathname. Given a command line, such as “cat filename”, sh forks a child process to execute the cat command and waits for the child process to terminate. The child process executes the command by changing its execution image to the cat program, passing as parameter filename to the program. When the child process terminates, it wakes up the parent sh process, which prompts for another command, etc. In addition to simple commands, the MTX sh also supports I/O redirections and multiple commands connected by pipes. In summary, the process of booting up MTX, login and use the MTX system is similar to, but much simpler than, that of other operating systems, such as Unix or Linux.

Problems

1. Consider the Intel x86 based PC in real-mode. Assume that the CPU's DS register contains 0x1000. What are the (20-bit) real addresses generated in the following instructions?

```
mov ax, 0x1234
add ax, 0x0002
```

A global variable char buf[1024], 20-bit real address of buf = ?

2. A binary executable a.out file consists of a header followed by TEXT and DATA sections.

```
|header| TEXT | DATA |<== BSS ===>|
```

The Unix command size a.out shows the size of TEXT, DATA, BSS of a.out. Use the following C program, t1.c, to generate t2.c to t6.c as specified below.

```
//********** t1.c file ************
  int g;
  main()
  {
     int a, b, c;
     a = 1; b = 2;
     c = a + b;
     printf("c=%d\n", c);
  }
```

t2.c: Change the global variable g to int g=3;
t3.c Change the global variable g to int g[10000];

t4.c Change the global variable g to int g[10000] = {4};
t5.c Change the local variables of main() to int a,b,c, d[10000];
t6.c. Change the local variables of main() to static int a,b,c, d[10000];

(A). In each case, use cc –m32 to generate a.out. Then use ls -l a.out to get a.out size, and use size a.out to get its section sizes. Record the observed sizes in a table:

Case	a.out	TEXT	DATA	BSS
(1)				
etc.				

Then answer the following questions:
(1). For the variables g, a, b, c, d, which variables are in DATA? Which variables are in BSS ?
(2). For the TEXT, DATA and BSS sections, which are in a.out, which is NOT in a.out? WHY?
(B). In each case, use cc -static t.c to generate a.out. Record the sizes again and compare them with the sizes in (A). What are the differences and why?

3. Under Linux, use gcc –m32 to compile and run the following program.

```
int *FP; // a global pointer
main(int argc, char *argv[], char *env[])
{   int a,b,c;
    printf("enter main\n");
    a=1; b=2; c=3;
    A(a,b);
    printf("exit main\n");
}
int A(int x, int y)
{   int d,e,f;
    printf("enter A\n");
    d=4; e=5; f=6;
    B(d,e);
    printf("exit A\n");
}
int B(int x, int y)
{   int u,v,w;
    printf("enter B\n");
    u=7; v=8; w=9;
    asm(“movl %ebp, FP”); // set FP=CPU’s %ebp register
    // Write C code to DO (1)-(3) AS SPECIFIED BELOW
    printf("exit B\n");
}
```

(1). In B(), FP points the stack frame link list in stack. Print the stack frame link list.
(2). Print in HEX the address and contents of the stack from FP to the stack frame of main().

(3). On a hardcopy of the output, identify and explain the stack contents in terms of function stack frames, i.e. local variables, parameters, return address, etc.
(4). Run the program as a.out one two three > file. Then identify the parameters to main(), i.e. where are argc, argv, env located?

4. Under Linux, use gcc –m32 to compile and run the long jump program in Section 2.4.4.
(1). Instead of jmp_buf env of Linux, define int *env[2]. Implement setjmp() and longjmp() as follows. In setjmp(env), save caller's [PC|FP] into env[0|1].
In longjmp(env, r), replace current [PC|FP] on stack with env[0|1], then return r.
Compile and run the program again to verify that the long jump scheme works.
(2). The gcc compiler generated code may reference local variables in main() by %esp register. Show how to modify (1) to make long jump work correctly.

5. Abnormal program termination: Under Linux, try to run the following C programs. In each case, observe what happens and explain WHY?

```
5-1. int *p; main(){   *p = 1;  }
5-2. int a,b,c; main(){ c = a/b; }
5-3. main(){ main(); }
5-4. main(){ printf("pid=%d\n", getpid()); while(1); }
```

(a). From the same X-terminal, enter Control_C key. What would happen and WHY?
(b). From another X-terminal, enter: kill -s 11 pid. What would happen and WHY?

6. Given an EXT2 file system, write a C program to display the contents of superblock, group descriptor 0, bitmaps as char maps, entries of the root directory.

7. Write a C program, showblock, which displays the disk block (direct, indirect and double indirect) numbers of a file in an EXT2 file system.

8. A RAM disk is a memory area used as a disk. Like a real disk, read/write a RAM disk is by 512-byte sectors. For example, in a real mode PC we may designate the 64 KB memory area from the segment 0x8000 to 0x9000 as a RAM disk. Write C functions

```
        read_sector(int sector, char buffer[512]), and
       write_sector(int sector, char buffer[512])
```

which read/write a RAM disk sector to/from a 512-byte buffer.

References

1. Antonakos, J.L., "An introduction to the Intel Family of Microprocessors", Prentice Hall, 1999.
2. BCC: Linux 8086 development environment, version 0.16.17, http://www.debath.co.uk
3. Bovet, D.P., Cesati, M., "Understanding the Linux Kernel, Third Edition", O'Reilly, 2005
4. 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

5. Cao, M., Bhattacharya, S, Tso, T., "Ext4: The Next Generation of Ext2/3 File system", IBM Linux Technology Center, 2007.
6. ELKS: sourceforge.net/projects/elks
7. EXT2: www.nongnu.org/ext2-doc/ext2.html
8. EXT3: jamesthornton.com/hotlist/linux-filesystems/ext3-journal
9. ECMA-119: Standard ECMA-119, Volume and File Structure of CDROM for Information Interchange, 2nd edition, December 1987.
10. ELF: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, 1995