Architectural Issues

Linkers and loaders, along with compilers and assemblers, are exquisitely sensitive to the architectural details of their target computers. In this chapter we cover enough computer architecture to understand the jobs that linkers have to do. The descriptions of all of the computer architectures in this chapter are deliberately incomplete and leave out the parts that don't affect the linker such as floating point and I/O.

Two aspects of architecture affect linkers: program addressing and instruction formats. One of the things that a linker does is to modify addresses and offsets both in data memory and in instructions. In both cases, the linker must ensure that its modifications are correct within the addressing scheme that the computer uses; when modifying instructions it must further ensure that the modifications don't result in an invalid instruction.

At the end of the chapter, we also look at address space architecture, that is, what set of addresses a program has to work with.

Memory Addresses

Every computer includes a main memory. The main memory invariably appears as an array of storage locations, with each location having a numeric address. The addresses start at zero and run up to some large number determined by the number of bits in an address.

Byte Order and Alignment

Each storage location consists of a fixed number of bits. Over the past 50 years computers have been designed with storage locations consisting of as many as 64 bits and as few as 1 bit, but now nearly every computer in production addresses 8 bit bytes. Since much of the data that computers handle, notably program addresses, are bigger than 8 bits, the computers can also handle 16, 32, and often 64 or 128 bit data as well, with multiple adjacent bytes grouped together. Some computers (notably those from IBM and Motorola) address multi-byte data by the address of the most significant byte, while others (notably DEC and Intel) by the address of the least significant byte, Figure 1. In a nod to *Gulliver's Travels* the IBM/Motorola byte order scheme is known as *big-endian* while the

DEC/Intel scheme is *little-endian*.

Figure 1: Byte addressable memory

the usual picture of memory addresses

The relative merits of the two schemes have provoked vehement arguments over the years. In practice the major issue determining the choice of byte order is compatibility with older systems, since it is considerably easier to port programs and data between two machines with the same byte order than between machines with different byte orders. Many recent chip designs can support either byte order, with the choice made either by the way the chip is wired up or by programming at system boot time. (On these switch-hitting chips, the byte order of data handled by load and store instructions changes, but the byte order of constants encoded in instructions doesn't. This is the sort of detail that keeps the life of the linker writer interesting.)

Multi-byte data must usually be *aligned* on a natural boundary. That is, four byte data should be aligned on a four-byte boundary, two-byte on two-byte, and so forth. Another way to think of it is that the address of any N byte datum should have at least log2(N) low zero bits. On some systems (Intel x86, DEC VAX, IBM 370/390), misaligned data references work at the cost of reduced performance, while on others (most RISC chips), misaligned data causes a program fault. Even on systems where misaligned data don't cause a fault, the performance loss is usually great enough that it's worth the effort to maintain alignment where possible.

Many processors also have alignment requirements for program instructions. Most RISC chips require that instructions be aligned on four-byte boundaries.

Address formation

As a computer program executes, it loads and stores data to and from memory, as determined by instructions in the program. The instructions are themselves stored in memory, usually a different part of memory from the program's data. Instructions are logically executed in the sequence they are stored, except that jump instructions specify a new place in the program to start executing instructions. (Jump instructions are also called branches; often an unconditional change of control is called a jump while a conditional change in control is called a branch. We call them all jumps here.) Each instruction that references data memory and each jump specifies the address or addresses or the data to load or store, or the instruction to jump to. All computers have a variety of instruction formats and address formation rules that linkers have to be able to handle as they relocate addresses in instructions.

Although computer designers have come up with innumerable different and complex addressing schemes over the years, most computers currently in production have a relatively simple addressing scheme. (Designers found that it's hard to build a fast version of a complicated architecture.) We'll use three architectures as examples:

- The IBM 360/370/390 (which we'll refer to as the 370). Although this is one of the oldest architectures still in use, its relatively clean design has worn well.
- SPARC V8 and V9. A popular RISC architecture, with fairly simple addressing. V8 uses 32 bit data and addresses, V9 adds 64 bit data and addresses.
- The Intel 386/486/Pentium (henceforth x86). One of the most arcane and irregular architectures still in use, but undeniably the most popular.

Instruction formats

Each architecture has several different instruction formats. We'll only address the format details relative to program and data addressing, since those are the main details that affect the linker. The 370 uses the same format for data references and jumps, while the SPARC has different formats and the x86 has some common formats and some different.

Each instruction consists of an opcode, which determines what the instruction does, and operands. An operand may be encoded in the instruction itself (an *immediate* operand), or located in memory. The address of each operand in memory has to be calculated somehow. Sometimes the address is contained in the instruction (direct addressing.) More often the address is found in one of the registers (register indirect), or calculated by adding a constant in the instruction to the contents of a register. If the value in the register is the address of a storage area, and the constant in the instruction is the offset of the desired datum in the storage area, this scheme is known as *based* addressing. If the roles are swapped and the register contains the offset, the scheme is known as *indexed* addressing. The distinction between based and indexed addressing isn't well-defined, and many architectures combine them, e.g., the 370 has an addressing mode that adds together two registers and a constant in the instruction, arbitrarily calling one of the registers the base register and the other the index register, although the two are treated the same.

Other more complicated address calculation schemes are still in use, but for the most part the linker doesn't have to worry about them since they don't contain any fields the linker has to adjust.

Some architectures use fixed length instructions, and some use variable length instructions. All SPARC instructions are four bytes long, aligned on four byte boundaries. IBM 370 instructions can be 2, 4, or 6 bytes long, with the first two bits of the first byte determining the length and format of the instruction. Intel x86 instructions can be anywhere from one byte to 14 long. The encoding is quite complex, partly because the x86 was originally designed for limited memory environments with a dense instruction encoding, and partly because the new instructions added in the 286, 386, and later chips had to be shoe-horned into unused bit patterns in the existing instruction set. Fortunately, from the point of view of a linker writer, the address and offset fields that a linker has to adjust all occur on byte boundaries, so the linker generally need not be concerned with the instruction encoding.

Addressability

In the earliest computers, memories were small, and each instruction had an address field large enough to contain the address of any memory location in the computer, a scheme now called direct addressing. By the early 1960s, addressable memory was getting large enough that an instruction set with a full address in each instruction would have large instructions that took up too much of still-precious memory. (The 1963 vintage IBM 360 series had 24 bit addresses, but many models had only 8K or 16K of main memory.) To solve this problem, computer architects abandoned direct addressing in some or all of the memory reference instructions, using index and base registers to provide most or all of the bits used in addressing. This allowed instructions to be shorter, at the cost of more complicated programming.

On architectures without direct addressing, including the IBM 370 and SPARC, programs have a ‘‘bootstrapping’’ problem for data addressing. A single routine may have to refer to stack allocated data (arguments and local variables), private static data, and global static data.

For stack allocated data, one of the registers serves as a stack pointer which can be used as an base register for arguments and local variables. In a common variant of this scheme, used with SPARC, a separate frame pointer register is loaded from the stack pointer at the time a procedure starts, which makes it possible to push variable sized objects on the stack but still address arguments and locals at fixed offsets from the frame pointer.

For local and global static data, a compiler can generate a table of pointers to all of the static objects that a routine references. If one of the registers contains a pointer to this table, the routine can address any desired static object by loading the pointer to the object from the table using the table pointer register into another register using the table pointer register as a base register, then using that second register as the base register to address the object. The trick, then, is to get the address of the table into the first register. On SPARC, the routine can load the table address into the register using a sequence of instructions with immediate operands, and on the SPARC or 370 the routine can use a variant of a subroutine call instruction to load the program counter (the register that keeps the address of the current instruction) into a base register, though for reasons we discuss later, those techniques cause problems in library code. A better solution is to foist off the job of loading the table pointer on the routine's caller, since the caller will have its own table pointer already loaded and can get address of the called routine's table from its own table.

Figure 2 shows a typical routine calling sequence. Rf is the frame pointer, Rt is the table pointer, and Rx is a temporary scratch register.

Figure 2: Idealized calling sequence

```
... push arguments on the stack ...
store Rt,xxx(Rf) ; save caller's table pointer in caller's stack frame
load Rx,MMM(Rt)   ; load address of called routine into temp register
load Rt,NNN(Rt)  ; load called routine's table pointer
call (Rx) ; call routine at address in Rs
load Rt,xxx(Rf)  ; restore caller's table pointer
```

Several optimizations are often possible. In many cases, all of the routines in a module share a single pointer table, in which case intra-module calls needn't change the table pointer. The SPARC convention is that an entire library shares a single table, created by the linker, so the table pointer register can remain unchanged in intra-module calls.

One remaining question is how this chain gets started. If each routine gets its table pointer loaded by the preceding routine, where does the initial routine get its pointer? The answer varies, but always involves special-case code. The main routine's table may be stored at a fixed address, or the initial pointer value may be tagged in the executable file so the operating system can load it before the program starts. No matter what the technique is, it invariably needs some help from the linker.

Data and instruction references

We now look more concretely at the way that programs in our three architectures address data values.

IBM 370

The 1960s vintage System/360 started with a very straightforward data addressing scheme, which has become someone more compled over the years as the 360 evolved into the 370 and 390. Every instruction that references data memory calculates the address by adding a 12-bit unsigned offset in the instruction to a base register and maybe an index register. There are 16 general registers, each 32 bits, numbered from 0 to 15, all but one of which can be used as index registers. If register 0 is specified in an address calculation, the value 0 is used rather than the register contents. (Register 0 exists and is usable for arithmetic, but not for addressing.) In instructions which take the target address of a jump from a register, register 0 means don't jump.

Figure 3 shows the major instruction formats. An RX instruction contains a register operand and a single memory operand, whose address is calculated by adding the offset in the instruction to a base register and index register. More often than not the index register is zero so the address is just base plus offset. In the RS, SI and SS formats, the 12 bit offset is added to a base register. An RS instruction has one memory operand, with one or two other operands being in registers. An SI instruction has one memory operand, the other operand being an immediate 8 bit value in the instruction An SS instruciton has two memory operands, storage to storage operations. The RR format has two register operands and no memory operands at all, although some RR instructions interpret one or both of the registers as pointers to memory. The 370 and 390 added some minor variations on these formats, but none with different data addressing formats.

Figure 3: IBM 370 instruction formats

Picture of IBM instruction formats RX, RS, SI, SS

Instructions can directly address the lowest 4096 locations in memory by specifying base register zero. This ability is essential in low-level system programming but is never used in application programs, all of which use base register addressing.

Note that in all three instruction formats, the 12 bit address offset is always stored as the low 12 bits of a 16-bit aligned halfword. This makes it possible to specify fixups to address offsets in object files without any reference to instruction formats, since the offset format is always the same.

The original 360 had 24 bit addressing, with an address in memory or a register being stored in the low 24 bits of a 32 bit word, and the high eight bits disregarded. The 370 extended addressing to 31 bits. Unfortunately, many programs including OS/360, the most popular operating system, stored flags or other data in the high byte of 32 bit address words in memory, so it wasn't possible to extend the addressing to 32 bits in the obvious way and still support existing object code. Instead, the system has 24 bit and 31 bit modes, and at any moment a CPU interprets 24 bit addresses or 31 bit addresses. A convention enforced by a combination of hardware and software states that an address word with the high bit set contains a 31 bit address in the rest of the word, while one with the high bit clear contains a 24 bit address. As a result, a linker has to be able to handle both 24 bit and 31 bit addresses since programs can and do switch modes depending on how long ago a particular routine was written. For historical reasons, 370 linkers also handle 16 bit addresses, since early small models in the 360 line often had 64K or less of main memory and programs used load and store halfword instructions to manipulate address values.

Later models of the 370 and 390 added segmented address spaces somewhat like those of the x86 series. These feature let the operating system define multiple 31 bit address spaces that a program can address, with extremely complex rules defining access controls and address space switching. As far as I can tell, there is no compiler or linker support for these features, which are primarily used by high-performace database systems, so we won't address them further.

Instruction addressing on the 370 is also relatively straightforward. In the original 360, the jumps (always referred to as branch instructions) were all

RR or RX format. In RR jumps, the second register operand contained the jump target, register 0 meaning don't jump. In RX jumps, the memory operand is the jump target. The procedure call is Branch and Link (supplanted by the later Branch and Store for 31 bit addressing), which stores the return address in a specified register and then jumps to the address in the second register in the RR form or to the second operand address in the RX form.

For jumping around within a routine, the routine has to establish ''addressability'', that is, a base register that points to (or at least close to) the beginning of the routine that RX instructions can use. By convention, register 15 contains the address of the entry point to a routine and can be used as a base register. Alternatively an RR Branch and Link or Branch and Store with a second register of zero stores the address of the subsequent instruction in the first operand register but doesn't jump, and can be use to set up a base register if the prior register contents are unknown. Since RX instructions have a 12 bit offset field, a single base register ''covers'' a 4K chunk of code. If a routine is bigger than that, it has to use multiple base registers to cover all of the routine's code.

The 390 added relative forms of all of the jumps. In these new forms, the instruction contains a signed 16 bit offset which is logically shifted left one bit (since instructions are aligned on even bytes) and added to the address of the instruction to get the address of the jump target. These new formats use no register to compute the address, and permit jumps within +/- 64K bytes, enough for any plausible single routine.

SPARC

The SPARC comes close to living up to its name as a reduced instruction set processor, although as the architecture has evolved through nine versions, the original simple design has grown somewhat more complex. SPARC versions through V8 are 32 bit architectures. SPARC V9 expands the architecture to 64 bits.

SPARC V8

SPARC has four major instruction formats and 31 minor instruction formats, Figure 4, four jump formats, and two data addressing modes.

In SPARC V8, there are 31 general purpose registers, each 32 bits, numbered from 1 to 31. Register 0 is a pseudo-register always contains the value zero.

An unusual *register window* scheme attempts to minimize the amount of register saving and restoring at procedure calls and returns. The windows have little effect on linkers, so we won't discuss them further. (Register windows originated in the Berkeley RISC design from which SPARC is descended.)

Data references use one of two addressing modes. One mode computes the address by adding the values in two registers together. (One of the registers can be r0 if the other register already contains the desired address.) The other mode adds a 13 bit signed offset in the instruction to a base register.

SPARC assemblers and linkers support a pseudo-direct addressing scheme using a two-instruction sequence. The two instructions are SETHI, which loads its 22 bit immediate value into the high 22 bits of a register and zeros the lower 10 bits, followed by OR Immediate, which ORs its 13 bit immediate value into the low part of the register. The assembler and linker arrange to put the high and low parts of the desired 32 bit address into the the two instructions.

Figure 4: SPARC

30 bit call 22 bit branch and SETHI 19 bit branch 16 bit branch (V9 only) op R+R op R+I13

The procedure call instruction and most conditional jump instructions (called branches in SPARC literature) use relative addressing with various size branch offsets ranging from 16 to 30 bits. Whatever the offset size, the branch shifts the offset two bits left, since all instructions have to be at four-byte word addresses, sign extends the result to 32 or 64 bits, and adds that value to the address of the current instruction to get the target address. The call instruction uses a 30 bit offset, which means it can reach any address in a 32 bit V8 address space. Call stores the return address in register 15. Various kinds of branches use a 16 or 19 bit offset, which is large enough to branch anywhere in any plausibly sized routine. The 16 bit format breaks the offset into a two-bit high part and a fourteen-bit low part, but that doesn't cause any great trouble for the linker.

SPARC also has a "jump and link" which computes the target address the same way that data reference instructions do, by adding together either two source registers or a source register and a constant offset, and also can store the the return address in a target register.

All SPARC jump and call instructions are *delayed*, which means that the instruction stored immediately after the jump is executed, before the first instruction at the target address. Jumps can include an *annul* bit which prevents the delayed instruction from being executed if the branch isn't taken.

Procedure calls use Call or Jump and Link, which store the return address in register 15, and jumps to the target address. Procedure return uses JMP 8[r15], to return to the instruction after the delayed instruction after the call.

SPARC V9

SPARC V9 expands all of the registers to 64 bits, using the low 32 bits of each register for old 32 bit programs. All existing instructions continue to work as before, except that register operands are now 64 rather than 32 bits. New load and store instructions handle 64 bit data, and new branch instructions can test either the 32 or 64 bit result of a previous instructions. SPARC V9 adds no new instructions for synthesizing full 64 bit addresses, nor is there a new call instruction. Full addresses can be synthesized via long sequences that create the two 32 bit halves of the address in separate registers using SETHI and OR, shift the high half 32 bits to the left, and OR the two parts together. In practice 64 bit addresses are loaded from a pointer table, and inter-module calls load the address of the target routine into a register and then use jump and link to make the call.

Intel x86

The Intel x86 architecture is by far the most complex of the three that we discuss. It features an asymmetrical instruction set and segmented addresses. There are six 32 bit general purpose registers named EAX, EBX, ECX, EDX, ESI, and EDI, as well as two registers used primarily for addressing, EBP and ESP, and six specialized 16 bit segment registers CS, DS, ES, FS, GS, and SS. The low half of each of the 32 bit registers can be used as 16 bit registers called AX, BX, CX, DX, SI, DI, BP, and SP. and the low and high bytes of each of the AX through DX registers are referred to as AL, AH, BL, BH, CL, CH, DL, and DH. On the 8086, 186, and 286, most registers had special functions, but on the 386 and later chips, most but not all of the functions that required specific registers have been generalized to use any register. The ESP is used as a hardware stack pointer, and always contains the address of the current stack. The EBP pointer is usually used to point to the base of the current stack frame. (The instruction set encourages but doesn't require this.)

At any moment an x86 is running in one of three modes: real mode which emulates the original 16 bit 8086, 16 bit protected mode which was added on the 286, or 32 bit protected mode which was added on the 386. Here we primarily discuss 32 bit protected mode. Protected mode involves the x86's notorious segmentation, but we'll disregard that for the moment.

Most instructions that address addresses of data in memory use a common instruction format, Figure 5. (The ones that don't use specific architecture defined registers, e.g., the PUSH and POP instructions always use ESP to address the stack.) Addresses are calculated by adding together any or all of a signed 1, 2, or 4 byte displacement value in the instruction, a base register which can be any of the 32 bit registers, and an optional index register which can be any of the 32 bit registers except ESP. The index can be logically shifted left 0, 1, 2, or 3 bits to make it easier to index arrays of multi-byte values.

Figure 5: Generalized x86 instruction format

one or two opcode bytes, optional mod R/M byte, optional s-i-b byte, optional 1, 2, or 4 byte displacement

Although it's possible for a single instruction to include all displacement, base, and index, most just use a 32 bit displacement, which provides direct addressing, or a base with a one or two byte displacement, which provides stack addressing and pointer dereferencing. From a linker's point of view, direct addressing permits a relocatable address to be embedded anywhere in the program on any byte boundary.

Conditional and unconditional jumps and subroutine calls all use relative addressing. Any jump or call instruction can have a 1, 2, or 4 byte offset which is added to the address of the instruction following the instruction to get the target address. This permits jumps and calls anywhere in the current segment. Unconditional jumps and calls also can compute the target address using the full data address calculation described above, most often used to jump or call to an address stored in a register. Call instructions push the return address on the stack pointed to by ESP.

Unconditional jumps and calls can also have a full six byte segment/offset address in the instruction, or calculate the address at which the segment/offset target address is stored. These call instructions push both the return address and the caller's segment number, to permit intersegment calls and returns.

Paging and Virtual Memory

On most modern computers, each program can potentially address a vast amount of memory, four gigabytes on a typical 32 bit machine. Few computers actually have that much memory, and even the ones that do need to share it among multiple programs. Paging hardware divides a program's address space into fixed size *pages*, typically 2K or 4K bytes in size, and divides the physical memory of the computer into *page frames* of the same size. The hardware includes *page tables* with an entry for each page in the address space, as shown in Figure 6.

Figure 6: Page mapping

Picture of pages mapped through a big page table to real page frames

A page table entry can contain the real memory page frame for the page, or flag bits to mark the page ‘‘not present.’’ When an application program attempts to use a page that is not present, hardware generates a *page fault* which is handled by the operating system. The operating system can load a copy of the contents page from disk into a free page frame, then let the application continue. By moving pages back and forth between main memory and disk as needed, the operating system can provide *virtual memory* which appears to the application to be far larger than the real memory in use.

Virtual memory comes at a cost, though. Individual instructions execute in a fraction of a microsecond, but a page fault and consequent page in or page out (transfer from disk to main memory or vice versa) takes several milliseconds since it requires a disk transfer. The more page faults a program generates, the slower it runs, with the worst case being *thrashing*, all page faults with no useful work getting done. The fewer pages a program needs, the fewer page faults it will generate. If the linker can pack related routines into a single page or a small group of pages, paging performance improves.

If pages can be marked as read-only performace also improves. Such pages don’t need to be paged out since they can be reloaded from wherever they came from originally., If identical pages logically appear in multiple address spaces, which often happens when multiple copies of the same program are running, a single physical page suffices for all of the address spaces.

An x86 with 32 bit addressing and 4K pages would need a page table with 2^20 entries to map an entire address space. Since each page table entry is usually four bytes, this would make the page tables an impractical 4 megabytes long. As a result, paged architectures page the page tables, with upper level page tables that point to the lower level page tables that point to the actual page frames corresponding to virtual addresses. On the 370, each entry in the upper level page table (called the segment table) maps 1MB of address space, so the segment table in 31 bit address mode may contain up to 2048 entries. Each entry in the segment table may be empty, in which case the entire segment is not present, or may point to a lower level page table that maps the pages in that segment. Each lower level page table has up to 256 entries, one for each 4K chunk of address space in the segment. The x86 divides up its page tables similarly, although the boundaries are different. Each upper level page table (called a page directory) maps 4MB of address space, so the upper level page table contains 1024 entries. Each lower level page table also contains 1024 entries to map the 1024 4K pages in the 4MB of address space corresponding to that page table. The SPARC architecture defines the page size as 4K, and has three levels of page tables rather than two.

The two- or three-level nature of page tables are invisible to applications with one important exception: the operating system can change the mapping for a large chunk of the address space (1MB on the 370, 4MB on the x86, 256K or 16MB on SPARC) by changing a single entry in an upper level page table, so for efficiency reasons the address space is usually managed in chunks of that size.

The program address space

Every application program runs in an address space defined by a combination of the computer's hardware and operating system. The linker or loader needs to create a runnable program that matches that address space.

The simplest kind address space is that provided by PDP-11 versions of Unix. The address space is 64K bytes starting at location zero. The read-only code of the program is loaded at location zero, with the read-write data following the code. The PDP-11 had 8K pages, so the data starts on the 8K boundary after the code. The stack grows downward, starting at 64K-1, and as the stack and data grow, the respective areas were enlarged; if they met the program ran out of space. Unix on the VAX, the follow-on to the PDP-11, used a similar scheme. The first two bytes of every VAX Unix program were zero (a register save mask saying not to save anything.) As a result, a null all-zero pointer was always valid, and if a C program used a null value as a string pointer, the zero byte at location zero was treated as a null string. As a result, a generation of Unix programs in the 1980s contained hard-to-find bugs involving null pointers, and for many years, Unix ports to other architectures provided a zero byte at location zero because it was easier than finding and fixing all the null pointer bugs.

Modern Unix systems typically use an only slightly more complex scheme. Programs are still loaded at location zero, or more often one page after location zero, in order to make zero a deliberately invalid address and catch any lurking null pointer bugs. Data starts on the page after the end of the code, and the stack grows downward from a fixed high address.

Unix systems put each application program in a separate address space, and the operating system in an address space logically separate from the applications. (Sometimes the operating system and the running application share a single address space, where addresses with the high bit set are for the system and those with the high bit clear are for the application, but the application has no access to the system space.) Other systems put multiple programs in the same address space, making the linker and particularly the loader's job more complex because a programs actual load address isn't known until the program's about to be run.

MS-DOS systems use no hardware protection, so the system and running applications share the same address space. When the system runs a program, it finds the largest chunk of free memory, which can be anywhere in the address space, loads the program into it, and starts it. IBM mainframe operating systems do roughly the same thing. In both cases, either the program loader or in some cases the program itself has to adjust to the location where the program is loaded.

MS Windows has an unusual loading scheme. Each program is linked to load at a standard starting address, but the executable program file contains relocation information. When Windows loads the program, it places the program at that starting address if possible, but may load it somewhere else if the preferred address isn't available.

Mapped files

Virtual memory systems move data back and forth between real memory and disk, paging data to disk when it doesn't fit in real memory. Originally, paging all went to ''anonymous'' disk space separate from the named files in the file system. Soon after the invention of paging, though, designers noticed that it was possible to unify the paging system and the file system by using the paging system to read and write named disk files. When a program maps a file to a part of the program's address space, the operating system marks all of the pages in that part of the address space not present, and uses the file as the paging disk for that part of the address space, as in Figure 7. The program can read the file merely by referencing that part of the address space, at which point the paging system loads the necessary pages from disk.

Figure 7: Mapping a file

Program points to set of page frames that map to disk file or local RAM

There are three different approaches to handling writes to mapped files. The simplest is to map a file read-only (RO), so that any attempts to store into the mapped region fail, usually causing the program to abort. The second is to map the file read-write (RW), so that changes to the memory copy of the file are paged back to the disk by the time the file is unmapped. The third is to map the file copy-on-write (COW, not the most felicitous acronym). This maps the page read-only until the program attempts to store into the page. At that time, the operating system makes a copy of the page which is then treated as a private page not mapped from a file. From the program's point of view, mapping a file COW is very similar to allocating a fresh area of anonymous memory and reading the file's contents into that area, since changes the program makes are visible to that program but not to any other program that might have mapped the same file.

Shared libraries and programs

In nearly every system that handles multiple programs simultaneously, each program has a separate set of page tables, giving each program a logically separate address space. This makes a system considerably more robust, since buggy or malicious programs can't damage or spy on each other, but it potentially could cause performance problems. If a single program or single program library is in use in more than one address space, the system can save a great deal of memory if all of the address spaces share a single physical copy of the program or library. This is relatively straightforward for the operating system to implement – just map the executable file into each program's address space. Unrelocated code and read only data are mapped RO, writable data are mapped COW. The operating system can use the same physical page frames for RO and unwritten COW data in all the processes that map the file. (If the code has to be relocated at load time, the relocation process changes the code pages and they have to be treated as COW, not RO.)

Considerable linker support is needed to make this sharing work. In the executable program, the linker needs to group all of the executable code into one part of the file that can be mapped RO, and the data into another part that can be mapped COW. Each section has to start on a page boundary, both logically in the address space and physically in the file. When several different programs use a shared library, the linker needs to mark the each program so that when each starts, the library is mapped into the program's address space.

Position-independent code

When a program is in use in several different address spaces, the operating system can usually load the program at the same place in each of the address spaces in which it appears. This makes the linker's job much easier, since it can bind all of the addresses in the program to fixed locations, and no relocation need be done at the time the program is loaded.

Shared libraries complicate this situation considerably. In some simple shared library designs, each library is assigned a globally unique memory address either at system boot time or at the time the libraries are created.

This puts the each library at a fixed address, but at the cost of creating a serious bottleneck to shared library administration, since the global list of library memory addresses has to be maintained by the system manager, not by individual users. Furthermore, if a new version of a library appears that is larger than the previous version and doesn't fit into the address space assigned, the entire set of shared libraries and, potentially, all of the programs that reference them, may need to be relinked.

The alternative is to permit different programs to map a library to different places in the address space. This eases library administration, but the compiler, and linker, and program loader need to cooperate so that the library will work regardless of where in the address space the library appears.

One simple approach is to include standard relocation information with the library, and when the library is mapped into each address space, the loader can fix up any relocatable addresses in the program to reflect the loaded addresses. Unfortunately, the process of fixing up involves writing into the library's code and data, which means that the pages will no longer be shared, if they're mapped COW, or the program will crash if the pages are mapped RO.

To avoid this problem, shared libraries use Position Independent Code (PIC), code which will work regardless of where in memory it is loaded. All the code in shared libraries is usually PIC, so the code can be mapped read-only. Data pages still usually contain pointers which need relocation, but since data pages are mapped COW anyway, there's little sharing lost.

For the most part, PIC is pretty easy to create. All three of the architectures we discussed in this chapter use relative branching, so that branch instructions within the routines need no relocation. References to local data on the stack use based addressing relative to a base register, which doesn't need any relocation, either. The only challenges are calls to routines not in the shared library, and references to global data. Direct data addressing and the MIPS high/low register loading trick won't work, because they both require run-time relocation. Fortunately, there are a variety of tricks one can use to let PIC code handle inter-library calls and global data. We discuss them when we cover shared libraries in detail in Chapter 9.

Intel 386 Segmentation

The final topic in this chapter is the notorious Intel architecture segmentation system. The x86 series is the only segmented architecture still in common use, other than some legacy ex-Burroughs Unisys mainframes, but since it's so popular, we have to deal with it.

The original 8086 was intended as a follow-on to Intel's quite popular 8-bit 8080 and 8085 microprocessors. The 8080 has a 16 bit address space, and the 8086 designers were torn between keeping the 16 bit address space, which made translation of 8085 easier and permitted more compact code, and providing a larger address space to give ''headroom'' for future applications in larger programs. They compromised, by providing multiple 16 bit address spaces. Each 16 bit address space was known as a segment.

A running x86 program has four active segments defined by the four segment registers. The CS register defines the code segment, from which instructions are fetched. The DS register defines the data segment, from which most data are loaded and stored. The SS register defines the stack segment, used for the operands of push and pop instructions, the program address values pushed and popped by call and return instructions, and any data reference made using the EBP or ESP as a base register. The ES register defines the extra segment, used by a few string manipulation instructions. The 386 and later chips define two more segment registers FS and GS. Any data reference can be directed into a specific segment by using a segment override. For example, the instruction MOV EAX,CS:TEMP fetches a data value from the location TEMP in code segment rather than the data segment. The FS and GS segments are only used via segment overrides.

The segment values need not all be different. Most programs set the DS and SS values the same, so that pointers to stack variables and global variables can be used interchangably. Some small programs set all four segment registers the same, providing a single address space known as tiny model.

On the 8086 and 186, the architecture defined a fixed mapping from segment numbers to memory addresses by shifting the segment number four bits to the left. Segment number 0x123 would start at memory location 0x1230 for example. This simple addressing is known as real mode. Programmers often refer informally to *paragraphs*, 16-byte units of memory that a segment number can address.

The 286 added a protected mode, in which the operating system can map segments to arbitrary places in real memory and can mark segments as not present, providing segment based virtual memory. Each segment can be marked executable, readable, or read/write, providing segment-level protection. The 386 extended protected mode to 32 bit addressing, so that each segment can be up to 4GB in size rather than only 64K.

With 16 bit addressing, all but the smallest programs have to handle segmented addresses. Changing the contents of a segment register is quite slow, 9 clock cycles on a 486 compared to 1 cycle to change the contents of a general purpose register. As a result, programs and programmers to go great lengths to pack code and data into as few segments as possible to avoid having to change the contents of the segment registers. Linkers aid this process by providing ‘‘groups’’ that can collect related code or data into a single segment. Code and data pointers can be either near, with an offset value but no segment number, or far, with both segment and offset. Compilers can generate code for various memory models which determine whether code and data addresses are near or far by default. Small model code makes all pointers near and has one code and one data segment. Medium model code has multiple code segments (one per program source file) using far calls, but a single default data segment. Large model code has multiple code and data segments and all pointers are far by default. Writing efficient segmented code is very tricky, and has been well documented elsewhere.

Segmented addressing places significant demands on the linker. Every address in a program has both a segment and an offset. Object files consist of multiple chunks of code which the linker packs into segments. Executable programs to be run in real mode have to mark all of the segment numbers that occur in the program so they can be relocated to the actual segments where the program is loaded. Executable programs to be run in protected mode further have to mark what data is to be loaded into what segment and the protection (code, read-only data, read-write data) for each segment.

Although the 386 supports all of the 16 bit segmentation features of the 286, as well as 32 bit versions of all of the segmentation features, most 32 bit programs don't use segmentation at all. Paging, also added in the 386, provides most of the practical benefits of segmentation without the performance cost and the extra complications of writing segment manipulation code. Most 386 operating systems run applications in the tiny model, creating a single code segment and a single data segment each 4GB long and mapping them both to the full 32 bit paged address space. Tiny model is a misnomer, since even though the program's only using a single segment, that segment can be the full size of the address space.

The 386 makes it possible to use both 16 bit and 32 bit segments in the same program and a few operating systems, notably Windows 95, take advantage of that ability. Windows 95 runs a lot of legacy Windows 3.1 code in 16 bit segments in a shared address space, while each new 32 bit program runs in its own tiny model address space, with the 16-bit programs' address space mapped in to permit calls back and forth.

Exercises

1. Does a linker or loader need to "understand" every instruction in the target architecture's instruction set? If a new model of the target adds new instructions, will the linker need to be changed to support them? What if it adds new addressing modes to existing instructions, like the 386 did relative to the 286?

2. Back in the Golden Age of computing, when programmers worked in the middle of the night because they had to rather than because that's when they woke up, many computers used word rather than byte addresses. The PDP-6 and 10, for example had 36 bit words and 18 bit addressing, with each instruction being a word with the operand address in the low half of the word. (Programs could also store addresses in the high half of the word, although there was no direct instruction set support for that.) How different is a linker for a word-addressed architecture from one for a byte addressed architecture?

3. How hard would it be to build a retargetable linker, that is, one that could be built to handle different target architectures by changing a few specific parts of the source code for the linker? How about a multi-target linker, that could handle code for a variety of different architectures (although not in the same linker job)?