ECE 391, Computer Systems Engineering
Notes for Students

Steven S. Lumetta

Spring 2022

As part of ECE Illinois' determination to make our curricula more inclusive, the faculty decided to consciously identify and eliminate any non-inclusive and potentially offensive terminology from our course materials. In the rare cases in which I feel that students may encounter such terms in their careers, I may mention their previous use. I also encourage you to report any remaining instances to me directly.

ECE391: Computer Systems Engineering — Lecture Notes Set 0

Review Material

This set of notes reviews material that you have probably already seen in ECE220 (or CS225). If you took neither ECE220 nor CS225, some of it may be new to you; the TAs might also talk about some of this material in the first couple of discussion sections, and both MP1 and MP2 will require you to make use of it. Regardless of your previous experience, you may want to scan the list of terms at the end of these notes and review the definitions for any terms that seem unclear to you. In order to make the notes more useful as a reference, definitions are highlighted with boldface, and italicization emphasizes pitfalls.

The notes begin with a review of how information is represented in modern digital computers, highlighting the use of memory addresses as pointers to simplify the representation of complex information types. The next topic is the systematic decomposition of tasks into subtasks that can be executed by a computer, and an example based on a pointer-based data structure. Moving on to high-level languages, and particularly the C language, the notes continue with a review of type and variable declarations in C as well as the use of structures, arrays, and new types. The next topic is C operators, with some discussion of the more subtle points involved in their use. After revisiting the decomposition example as written in C, the notes discuss implicit and explicit conversions between types. Finally, the notes conclude with a review of C's preprocessor. We deliberately omit the basics of C syntax as well as descriptions of C conditional (`if/else` and `switch`) and iteration (`while`, `do`, and `for` loops) statements.

Representation as Bits

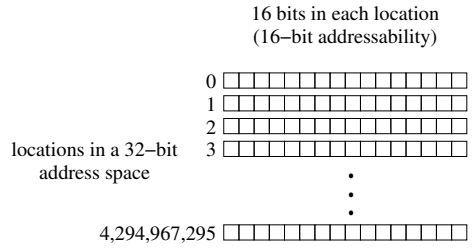
Modern digital computers represent all information as sets of binary digits: 0s and 1s, or **bits**. Whether you are representing something as simple as an integer or as complex as an undergraduate thesis, the data are simply a bunch of 0s and 1s inside a computer. For any given type of information, a human selects a data type for the information. A **data type** (often called just a **type**) consists of both a size in bits and a representation, such as the 2's complement representation for signed integers, or the ASCII representation for English text. A **representation** is a way of encoding the things being represented as a set of bits, with each bit pattern corresponding to a unique object or thing.

In general, computers do not interpret the bits that they store. A typical instruction set architecture (ISA) supports a handful of data types in hardware in the sense that it provides hardware support for operations on those data types. Most modern arithmetic logic units (ALUs), for example, support addition and subtraction of both unsigned and 2's complement representations, with the specific data type (such as 16- or 64-bit 2's complement) depending on the ISA. Data types and operations not supported by the ISA must be handled in software using a small set of primitive operations, which form the **instructions** available in the ISA. Instructions usually include data movement instructions such as loads and stores and control instructions such as branches and subroutine calls in addition to operations.

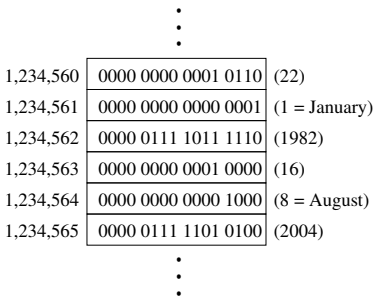
Pointers and Data Structures

One particularly important type of information is memory addresses. Imagine that we have stored some bits representing the number 42 at location 0100100100010100 in a computer's memory. In order to retrieve our stored number, or rather the bits representing it, we must provide the memory with the bits corresponding to the memory location, in this case 0100100100010100. The representation of a memory address as bits is thus straightforward: we simply use the bits that must be provided to the memory in order to retrieve the desired data as the representation for that memory address. Unlike most other types of information, no translation process is necessary; we represent a bunch of bits with a bunch of bits. We might thus think of the bits 0100100100010100 as a **pointer** to the number 42. They point to our stored number in the sense that the memory, when provided with the pointer, produces the stored number.

It is important to recognize, however, that *a pointer is just a bunch of bits, just like any other representation used by a computer.* In particular, we can choose to interpret the bits making up a pointer as an unsigned integer, which provides us with a way of ordering the locations in a memory's address space, as shown to the right for a memory consisting of 2^{32} locations holding 16 bits each, in other words, with an **addressability** of 16 bits.



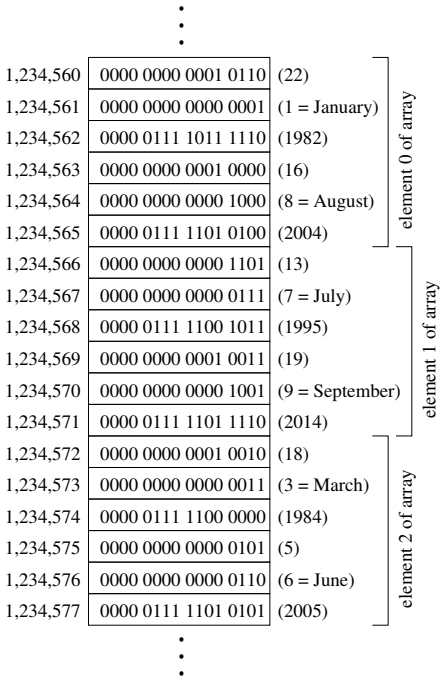
The induced ordering is useful when the representation for some type of information requires more bits than the addressability of the memory. Imagine that you are charged with selecting a representation for the birth and hiring dates of all employees of your company in a memory with 16-bit addressability. Given a reasonable set of assumptions about the range of possible values, representing two dates with only 16 bits is quite challenging. Instead, you might decide to use six memory locations for each person, representing each person with 2's complement integers for the day, month, and year of the two recorded dates. An example appears to the right. Such a multi-location representation is often called a **data structure**, because the information represented is made up of several components, or **fields**, each of which has a data type. In this case, all of the fields are 16-bit 2's complement integers, but the fields of a data structure need not in general have the same type.



In the example above, the value 1,234,560 (represented by the bit pattern 00000000000100101101011010000000) is a pointer to the employee structure, and all fields of the data structure reside at fixed offsets from this pointer. To find the hiring month, for example, one adds four to the pointer and retrieves the value from the resulting memory address (treating the sum as a pointer).

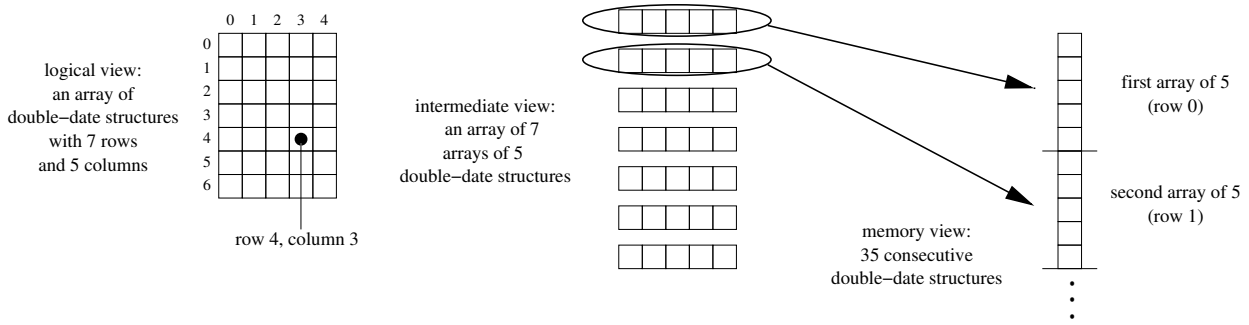
Arrays

Multiple instances of a certain type are often represented as arrays. Again using the ordering induced by treating memory addresses as unsigned integers, the bits for each element of the array are simply laid out sequentially in memory, with each element occupying whatever number of memory locations is necessary for the data type. We refer to this number as the **size** of the type, and (usually) measure it in terms of the addressability of the memory system. Given the address of the first element of an array, we can easily calculate the address of any element by simply adding a multiple of the size of the type. For convenience, we number array elements starting with 0, allowing us to use the product of the array index and the type size as the offset for any given element. *All array elements must be of the same size in order for this calculation to work properly.* In the example shown to the right, we find element 2 by adding 2×6 to 1,234,560. The result, 1,234,572, is a pointer to element 2 of the array. We can also think of the pointer 1,234,572 as a pointer to an array with two fewer elements than the original array.



The **duality** between pointers and arrays, by which any pointer can be treated as an array, and any array can be treated as a pointer, is both useful and a potential source of confusion. If, for example, you create an array of ten of our double-date structures, and write code that takes a pointer to the array of ten and accesses the eleventh (element 10), your code will proceed to read garbage bits from the memory. Adding 10×6 to the pointer to element 0 does produce a valid memory address, but neither the contents of that address nor those of the subsequent five addresses were intended to be a double-date structure. With any pointer (in other words, memory address), your program must keep track of the type of the data to which the pointer points. With a pointer to an array, your code must also keep track of the size of the array. To a computer, bits are just bits.

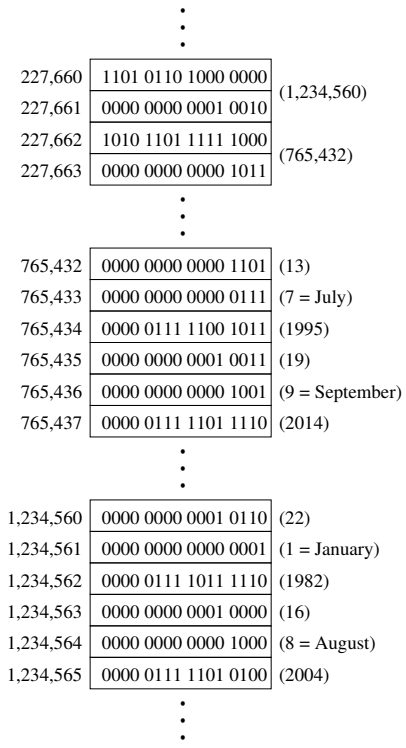
Multidimensional arrays are usually handled by thinking of each of the lower dimensions in the array as a new type of data. For example, let's say that we want to create an array of our double-date structures with seven rows and five columns, as shown in the following figure. We begin by thinking about an array of five double-date structures. Each row in our two-dimensional array is simply an array of five double-date structures. Our entire two dimensional array is then an array of seven of the arrays of five. When we lay it out in memory, the arrays of five are again laid out consecutively.



How do we find an element in our two-dimensional array? Let's say that we want to access row 4 and column 3. The size of the array of five is $5 \times 6 = 30$, and we are looking for row 4, so we multiply 4×30 and add it to the pointer to the start of the array to find a pointer to row 4. We then multiply 3×6 and add it to the pointer to row 4 to obtain a pointer to the desired array element. As you may have already observed, the duality between arrays and pointers also holds for multi-dimensional arrays: we can choose to view our two-dimensional array as an one-dimensional array of 35 structures if it is useful or convenient to do so.

Arrays of Pointers

As mentioned earlier, a pointer is simply a type of information. In practice, it is often useful to associate the type of information to which a pointer points with the pointer itself, but this association is purely an abstraction and has no effect on the bits used to represent the pointer. However, mentally distinguishing a pointer to an integer from a pointer to one of the double-date data structures from our running examples can help when discussing our next step: a pointer may point to a pointer, which may in turn point to another pointer, and so forth.



As a simple example, rather than building an array of our double-date structures in memory, we might instead build an array of pointers to double-date structures, as shown to the right. The upper block in the figure shows the first two pointers in our array of pointers; each pointer occupies 32 bits, or two memory locations. The lower block in the figure shows the first structure referenced by the array; the address of this structure is stored as element 0 in our array of pointers. The middle block in the figure shows the second structure referenced by the array, in other words, the structure to which element 1 of the array points. Given a pointer—227,660—to the array, we obtain a pointer to element 1 by adding 2, the size of a pointer, to the array pointer. The data at that memory address are then the address of the desired structure, in this case the pointer 765,432.

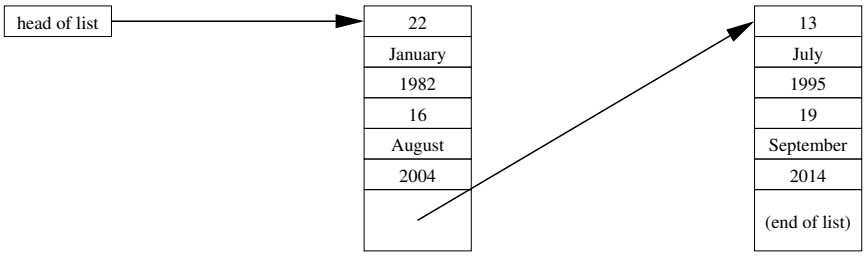
What is the difference between using an array of pointers to structures and using an array of structures? With an array of structures, finding the address of any desired element is easy, requiring only a multiplication and an addition. Using an array of pointers to structures introduces another memory access to retrieve the structure pointer from the array. However, if we want to move elements around within the array, or want to be able to have the same structure appear more than once in the array, using an array of pointers makes our task much easier, as moving and copying pointers is easier than moving and copying whole data structures, particularly if the data structures are large. Similarly, using an array of pointers allows us to use a special value (usually 0) to indicate that some elements do not exist, which may reduce the amount of memory needed to store an array for which the size changes while the program executes.

Pointers within Data Structures

Just as we can form arrays of pointers, we can also include pointers as fields within data structures. For example, we can extend our double-date structure by appending a field that points to another double-date structure, as shown to the right. This new structure can then be linked together, from one to the next, to form a list of structures representing the birth and hiring dates for all employees of a company. We call such a structure a **linked list**, with the pointer field forming the link between each element in the list and the subsequent element. A single pointer then suffices to indicate the location of the start of the list, and a special value (again, 0) is used to indicate the end of the list.

birth day
birth month
birth year
hiring day
hiring month
hiring year
pointer to next structure

The figure below illustrates a linked list of two employees. The upper part of the figure is the logical list structure, and the lower part is a possible organization in memory. For this figure, we have deliberately changed both the addresses and bits for each memory location into hexadecimal form. Hexadecimal notation is easier for humans to parse and remember than is binary notation. However, *use of hexadecimal is purely a notational convenience and does not change the bits actually stored in a computer.*



0x0003794C	0xD620	(0x0012D620)
0x0003794D	0x0012	

0x0012D620	0x0016	(22)
0x0012D621	0x0001	(1 = January)
0x0012D622	0x07BE	(1982)
0x0012D623	0x0010	(16)
0x0012D624	0x0008	(8 = August)
0x0012D625	0x07D4	(2004)
0x0012D626	0xADF8	(0x000BADF8)
0x0012D627	0x000B	

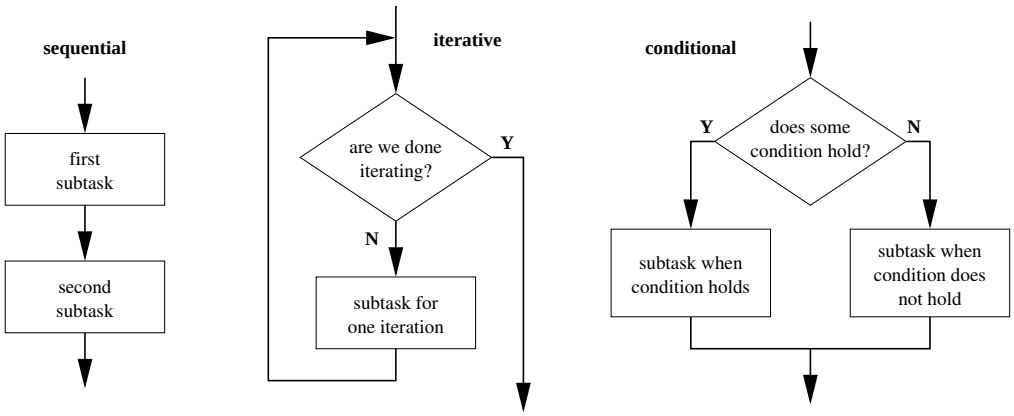
0x000BADF8	0x000D	(13)
0x000BADF9	0x0007	(7 = July)
0x000BADFA	0x07CB	(1995)
0x000BADFB	0x0013	(19)
0x000BADFC	0x0009	(9 = September)
0x000BADFD	0x07DE	(2014)
0x000BADFE	0x0000	(0x00000000)
0x000BADFF	0x0000	

What is the difference between using a linked list and using an array of pointers? A linked list requires only one pointer per element in the list, plus one extra for the list head, and does not require a programmer to guess the maximum size of the list in advance. In contrast, an array of pointers contains a fixed number of pointers; this number must be chosen in advance, and must be large enough for all executions of the program, even if most executions require far fewer list elements than this theoretical maximum. The array itself can be resized dynamically, but doing so requires extra memory accesses and copying. The speed of access in an array is its primary advantage. For comparison, getting to the middle of a list means reading all of the pointers from the head of the list to the desired element. More sophisticated data structures using multiple pointers to provide more attractive tradeoffs between space and operation speed are possible; you will see some of these in ECE391, but are not assumed to have already done so.

Systematic Decomposition

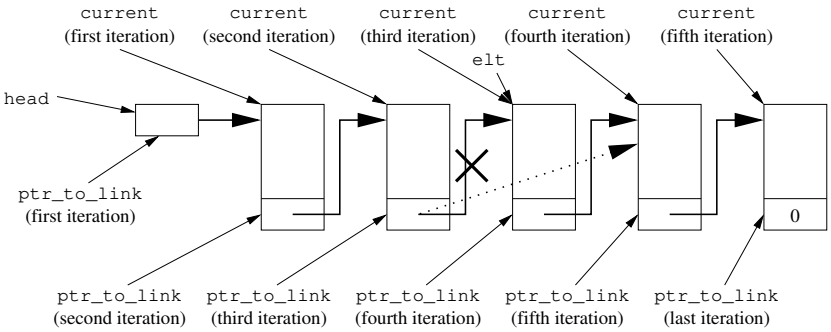
We are now ready to shift gears and talk about the process of turning a specification for a program into a working program. Essentially, given a well-defined task expressed at some suitably high level of abstraction, we want to be able to systematically transform the task into subtasks that can be executed by a computer. We do so by repeatedly breaking each task into subtasks until we reach the desired level, at which each of the remaining tasks requires only a machine instruction or two. The name for this process is **systematic decomposition**, and the ECE220 textbook by Patt and Patel[1] provides a detailed introduction to the ideas; the basics are replicated here for review purposes.

[1] Yale N. Patt, Sanjay J. Patel, *Introduction to Computing Systems: from Bits & Gates to C & Beyond*, 3^{rd} edition, McGraw Hill, New York, New York, 2020, ISBN13 9781260150537.



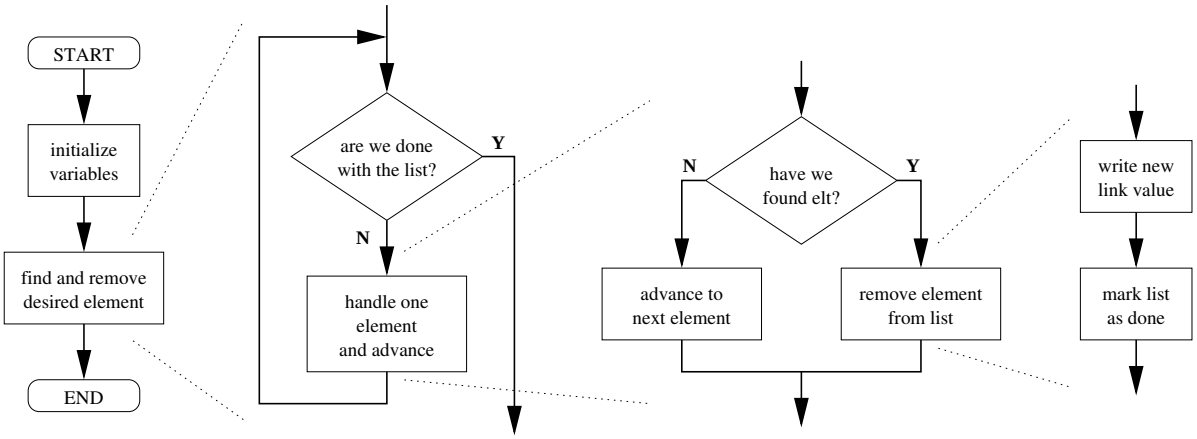
When writing a program, it is often helpful to get out a sheet of paper and draw a few pictures of the data structures before trying to write code. Similarly, it is often useful to explicitly draw the organization of the code, which is where systematic decomposition comes into play. Begin by drawing a single box that implements the entire task. Next, repeatedly select a box that cannot be implemented with a single machine instruction and break it down into smaller tasks using one of the three constructions shown above: sequential, iterative, or conditional. Some tasks require a sequence of subtasks; for these, we use the sequential construction, breaking the original task into two or more subtasks. Some tasks require repetition of a particular subtask. These make use of the iterative construction; determining the condition for deciding when the iterative process should end is critical to using such a construction. Finally, some tasks divide into two (or possibly more) subtasks based on some condition. To make use of the conditional construction, this test must be stated explicitly, and the steps for each outcome clearly stated.

Example: Decomposing Linked List Removal



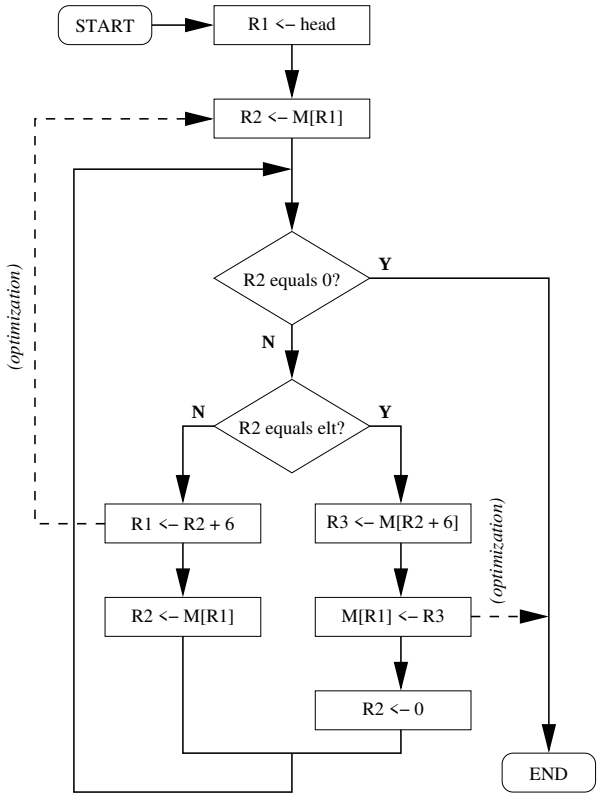
As an example, let's decompose the task of removing one of our double-date data structures from a linked list. For this task, we are given two input variables. The variable `head` holds a pointer to the list head, and the variable `elt` holds a pointer to the element to be removed from the list. The variable names represent the addresses at which these values are stored. Thus, at the **register transfer language (RTL)** level, we obtain a pointer to the first list element by reading from `M[head]`, the memory location to which `head` points. The figure to the right shows a picture of the list with one possible value of `elt`, for which the dotted line and cross indicate the change to be made to the data structure in order to remove `elt` from the list.

The basic operation for removal is thus to iterate over the list elements until we find the element that we want to remove, then change the element's predecessor's link to point to the element's successor, as illustrated in the diagram. For this purpose, we make use of two additional variables that we keep in registers. The first, `ptr_to_link`, points to the link that might be changed, in other words either the link field of a list element or to the memory location in which the pointer to the first element is held. The second, `current`, is simply the value to which `ptr_to_link` points. When `current` has the value 0, we have reached the end of the list. The `current` variable is thus a pointer to a double-date structure, while the `ptr_to_link` variable is a pointer to a pointer to a double-date structure.



We are now ready to systematically decompose the problem. The **flow chart** on the left above shows the process after one step of refinement in which we have dedicated a subtask to initializing the `ptr_to_link` and `current` variables. For the next step, we decompose the task of finding and removing the desired element from the list. As discussed, this step requires that we iterate over the elements of the list, so we use an iterative construction, first testing whether we have reached the end of the list, then handling one element as the iterated subtask. We next decompose the process for handling a single list element into two cases using a conditional construction. If the current list element is not equal to `elt`, we move on to the next element in the list. If the current list element is equal to `elt`, we remove it from the list by overwriting the link pointer, at which point we are done with the task and can leave the loop. To do so, we update our variables to indicate that the loop has completed. The last refinement shown, at the right side of the figure above, shows a sequential decomposition of the task of removing an element into overwriting the link pointer and marking the list as done (by setting the `current` variable to 0, *NOT by changing the list*).

At this point, we are ready to transform the tasks into RTL-level instructions. We use three registers in our code. Register R1 holds `ptr_to_link`, which is the pointer to the pointer to the element being considered for removal as we traverse the list. Register R2 holds `current`, a pointer to the element being considered. We find `current` by **dereferencing** `ptr_to_link`, in other words by reading from the memory location to which `ptr_to_link` points. Finally, register R3 is a temporary used to hold a pointer to the successor of `elt` when removing it.



We initialize our variables by setting R1 equal to `head` and dereferencing R1 to find R2. We compare R2 with 0 to decide whether or not we have finished the list. If not, we check whether we have found `elt` by comparing it with R2. If not (the left side of the conditional), we must advance R1 to point to the link field of the next element in the list. Since R2 already points to the data structure, we need merely add the offset of the link field, which is 6 assuming our example memory, and store the result in R1. We then dereference the new value of R1 to find the new value of R2 and continue with the next iteration. Once we find `elt` (the right side of the conditional), we read the value of the link field in the current element (again at offset 6 from R2) and write it into the predecessor's link field (pointed to by R1), completing the removal of `elt`. We then set R2 to 0 to end the loop. Once we have fully decomposed the problem, two minor optimizations become apparent (dashed lines in figure).

As an exercise, try to systematically decompose the problem of inserting a new element into a linked list, given the address of the head of the list and the address of the new list element. The simple version inserts the element at the front of the list, but try writing a second version that inserts the element at the end of the list.

Designing Iterations

Systematic decomposition provides a sound basis for one's first steps in programming, but addresses only the procedural aspects of a program. While low-level programming tasks are almost always procedural in nature, many high-level decisions involve reasoning about the choice of interfaces between large code modules and the data structures used to implement those code modules. In practice, success with these latter design problems requires substantial programming experience and exposure to previous approaches, and in that sense is more the topic of our course than review material.

However, systematic decomposition can also sometimes lead to code that favors structure over clarity and simplicity, and you should already be aware of alternative systematic approaches (or, perhaps, extensions) that can help you to build your programming skills. Towards this end, we outline how a more experienced programmer designs an iterative task by asking a sequence of questions about the iteration:

1. What is the task to be repeated?
2. What **invariants** hold at the start of each iteration; equivalently, what statements do we know to be true at these times?
3. Under what **stopping conditions** should we end the iteration?
4. What should be done when we stop? The action might be different for each stopping condition.
5. How do we prepare for the first iteration?
6. How do we update variables between iterations?

You might notice that the early description and figure used to describe the linked list removal problem hinted at the answers to these questions, which reflects the degree to which this process becomes internalized and automatic. The concept of an invariant alone is quite powerful and useful in many other contexts, such as the design of data structures and algorithms, while stopping conditions are fully analogous to the base cases in recursive constructs.

We end by explicitly reviewing the answers for the case of linked list removal and showing that this approach leads directly to the optimized form of the process. The task to be repeated is consideration of a single list element for removal. We define two variables to help with the removal, defining their contents as our invariants. In particular `ptr_to_link` points to the link that might be changed, and `current` is the value to which `ptr_to_link` points. Note that these definitions *do not necessarily hold in the middle of the iteration*. We end the loop when we either (1) reach the end of the list, in which case we did not find the element to be removed (and might choose to return an error condition), or (2) find the element to be removed, in which case we do so. To prepare for the first iteration, we need merely initialize our two variables to match their defining invariants. To update, we change the variables to point to the next element. As `current` is defined to be the value to which `ptr_to_link` points, we can always use one subtask to set it up once `ptr_to_link` is ready, whether we are preparing for the first iteration or any other.

Data Types in C

We're now ready to move up a level of abstraction and to review the same set of concepts as they appear in the language C. High-level languages typically associate types with data in order to reduce the chance that the bits making up an individual datum are misused or misinterpreted accidentally. If, for example, you store your name in a computer's memory as a sequence of ASCII characters and write an assembly program to add consecutive groups of four characters as 2's complement integers and to print the result to the screen, the computer will not complain about the fact that your code produces meaningless garbage. In contrast, most high-level languages will give you at least a warning, since such implicit re-interpretations of the bits are rarely intentional and thus rarely correct. The compiler can also generate the proper conversion code automatically when the transformations are intentional, as is often the case with arithmetic.

Some high-level languages, such as Java, go a step further and actually prevent programmers from changing the type of a given datum (in most cases). If you define a type that represents one of your favorite twenty colors, for example, you are not allowed to turn a color into an integer, despite the fact that the color is represented as a handful of bits. Such languages are said to be **strongly typed**.

The C language is not strongly typed, and programmers are free to interpret any bits in any manner they see fit. Taking advantage of this ability in any but a few exceptional cases, however, usually results in arcane and non-portable code, and is thus considered to be bad programming practice. We discuss conversion between types in more detail later in these notes.

Each high-level language defines a number of **primitive** data types that are always available in the language, as well as ways of defining new types in terms of these primitives. The primitive data types in C are signed and unsigned integers of various sizes, single- and double-precision floating-point numbers, and pointers. New types can be defined as arrays of an existing type, structures with named fields, and enumerations (for example, colors: red, yellow, and blue).

The primitive integer types in C include both unsigned and 2's complement representations. These types were originally defined so as to give reasonable performance when code was ported. In particular, the `int` type is intended to be the native integer type for the target ISA, which was often faster than trying to use larger or smaller integer types on a given machine. When C was standardized, these types were defined so as to include a range of existing C compilers rather than requiring all compilers to produce uniform results. At the time, most workstations and mainframes were 32-bit machines, while most personal computers were 16-bit, thus flexibility was somewhat desirable. With the GCC compiler on Linux, the C types are defined in the table above. Although the `int` and `long` types are usually the same, there is a semantic difference in common usage. In particular, on most architectures and most compilers, a `long` has enough bits to hold a pointer, while an `int` may not. When in doubt, the **size in bytes** of any type or variable can be found using the built-in C function `sizeof`.

	2's complement	unsigned
8 bits	`char`	`unsigned char`
16 bits	`short` `short int`	`unsigned short` `unsigned short int`
32 bits	`int`	`unsigned` `unsigned int`
32 or 64 bits	`long` `long int`	`unsigned long` `unsigned long int`
64 bits	`long long` `long long int`	`unsigned long long` `unsigned long long int`

Over time, the flexibility of size in C types has become less important (except for the embedded markets, where one often wants even more accurate bit-width control), and the fact that the size of an `int` can vary from machine to machine and compiler to compiler has become more a source of headaches than a helpful feature. In the late 1990s, a new set of fixed-size types were recommended for inclusion in the C library, reflecting the fact that many companies had already developed and were using such definitions to make their programs platform-independent. Although these types—shown in the table above—are not yet commonplace, we encourage you to make use of them. In Linux, they can be used by including the `stdint.h` header file in user code or the `linux/types.h` header file in kernel code.

	2's complement	unsigned
8 bits	`int8_t`	`uint8_t`
16 bits	`int16_t`	`uint16_t`
32 bits	`int32_t`	`uint32_t`
64 bits	`int64_t`	`uint64_t`

Floating-point types in C include `float` and `double`, which correspond respectively to single- and double-precision IEEE floating-point values. Although the 32-bit `float` type can save memory compared with use of 64-bit `double` values, C's math library works with double-precision values, and single-precision data are uncommon in scientific and engineering codes. Single-precision floating-point currently dominates the graphics industry, but even there support for double-precision will be prevalent soon.

Pointer types in C are formed by appending asterisks to the name of a type. For example, an `int*` is a pointer to an `int`, and a `double*` is a pointer to a `double`. The `void*` type is a generic pointer, and serves as a generic memory address. Pointers can, of course, also point to pointers. For example, a `short***` is a pointer to a pointer to a pointer to a `short` (read it from right to left). Using more than two levels of indirection in this manner is rare, however.

C Variables

Variables can be defined in one of two places in C: outside of any function, or at the start of a compound statement (a sequence of statements enclosed by braces). Historically, and in contrast with C++, variables in C could not be defined in the middle of compound statements, nor in expressions such as `for` loops. Although permitted by recent standards, *declaring variables haphazardly tends to obscure type information*.

The **scope** of a variable specifies what parts of a program can access the variable, while its **storage class** specifies when and where in memory the variable is stored. For C variables, both scope and storage class depend on where the variable is defined and upon whether or not the variable definition is preceded by the qualifier `static`.

Variables defined in compound statements, including function bodies, are accessible only within the compound statement in which they are defined. Defining a variable with the same name as another variable with a broader (but enclosing) scope **shadows** the second variable, making it inaccessible within the first variable's scope. Shadowing variables makes your code harder to read and more likely to contain bugs.

A variable defined outside of a function is accessible by any code in the file after its definition. If the variable is not preceded by the `static` qualifier, it is a **global variable**, and is also accessible from other files provided that the variable has been declared before being used. Here we distinguish between a **variable definition**, which results in the creation of a new variable, and a **variable declaration**, which makes a variable defined elsewhere accessible. In C, variable declarations look the same as definitions, but are preceded by the keyword `extern`. Header files usually include declarations, for example, but not definitions, since *a variable definition in a header file produces a separate copy of the variable for every file that includes the header file*. Global variables should be used sparingly, if at all.

Variables in C are stored either on the stack or in a static data region allocated when the program is loaded into memory. Variables declared outside of functions are placed in the static data region, as are variables declared in compound statements *and preceded by the* `static` *qualifier*. For any variable declared in a compound statement and without the `static` qualifier, space is allocated on the stack as part of the function's **stack frame** whenever the enclosing function begins execution, and is discarded when the function returns.[2] Such variables have **automatic** storage class. Multiple copies may thus exist if a function can call itself recursively, but only one copy is accessible by name from any given execution of the function. Note that if a `static` variable is declared inside a compound statement, *only one copy exists.* That copy is then accessible from all executions of the enclosing function, even if the function calls itself recursively, and the copy persists even when the function is not currently executing, although no code outside of the compound statement can access the variable by name.

The short sample code below illustrates variable declarations and definitions.

```
extern int    a_declaration;        /* must be defined elsewhere */
       int*   a_global_variable;
static double a_file_scope_variable;

int a_function (short int arg)
{
    static long int function_var;         /* unique copy in static data area */
           short    function_var_on_stack; /* one copy per call to a_function */
    {
        unsigned char another_stack_var;
        if (arg == 0)
            return 0;
    }
    /* another_stack_var is not accessible here.  */
    function_var_on_stack = arg - 1;
    return a_function (function_var_on_stack);
}
```

[2] Stack frames, also known as **activation records**, are covered in detail for the x86 ISA in the next set of notes.

Structures and Arrays

Structures can be defined in C using the `struct` keyword. Structure names technically occupy a separate **name space** from variables. In other words, one can use a variable and a structure with identical names without syntactic errors, but overloading the meaning of a symbol in this way makes your code hard to read and should be avoided. If not explicitly defined as a new type, as discussed in the next section, structures used to define variables must include the `struct` keyword in the type name. For example, one can define a structure to represent two dates and then define double-date variables as shown to the right.

```
struct two_date_t {
    short birth_day;
    short birth_month;
    short birth_year;
    short hiring_day;
    short hiring_month;
    short hiring_year;
};

struct two_date_t my_data, your_data;
```

Defining a structure does not cause the compiler to allocate storage for such a structure. Rather, the definition merely tells the compiler the types and names of the structure's **fields**. Variable declarations for variables with these new types look and are interpreted no differently than variable declarations for primitive data types such as `int`.

A structure definition enables the compiler to generate instructions that operate on the fields of the structure. For example, given a pointer to a double-date structure, the compiler must know how much to add to the pointer to find the address of the `hiring_month` field. The compiler must ensure that the instructions do not violate rules imposed by the architecture, such as address alignment. For example, most ISAs use 8-bit-addressable memory but require that 32-bit loads and stores occur as if the memory were 32-bit-addressable, that is, only using addresses that are multiples of four. A processor for such an ISA is in fact not capable of performing an unaligned load, and the compiler must avoid generating code that requires any. It does so by inserting padding bytes into the structure to guarantee proper field alignment for the target ISA. *Making assumptions about fields offsets and size for a structure is thus risky business and should be avoided.* Use the `sizeof` function to find a structure's size and the field names to access its fields.

As with pointers, arrays of any type can be defined in C by simply appending one or more array dimensions to a variable definition:

```
int array[10];                  /* an array of 10 integers                          */
int multi[5][9];                /* an array of 5 arrays of 9 integers               */
struct two_date_t example[7][5]; /* an array of 7 arrays of 5 double-date structures */
```

Arrays are laid out sequentially in memory, and the name of the array is equivalent to a pointer to the first subarray (with one fewer dimension). With the examples above, `array` is a pointer to the first of ten `int`s, `multi` is a pointer to the first of five arrays of nine `int`s, and `example` is a pointer to the first of seven arrays of five `two_date_t` structures.

Defining New Types in C

As we've just seen, defining and using simple structures and arrays does not require the creation of new named types, but building complex data structures without intermediate names for the sub-structures or arrays can be confusing and error-prone. New types in C can be defined using the **type definition** command `typedef`, which looks exactly like a variable declaration, but instead defines a new data type.

Examples appear to the right. The first line creates two new types to represent the width and height of a screen, both of which are simply `int`s. The second line defines our double-date structure as a structure in its own right, allowing us to define variables, pass parameters, and so forth, without writing `struct` over and over again. The last line defines a chess board as an 8-by-8 array of integers. The use of "_t" in the type names is a convention often used to indicate to programmers that the name is a type rather than a variable.

```
typedef int screen_width_t, screen_height_t;
typedef struct two_date_t two_date_t;
typedef struct int chessboard_t[8][8];
```

Enumerations are another convenient method for defining symbolic names for various sets or one-bit flags. By default, constants in an `enum` start at 0 and count upwards by 1, but any can be overridden by assignment, and the symbols need not have unique values.

A few examples follow. The left example defines three constants and a fourth symbol that can be used to size arrays to hold one value (such as a human-readable name) for each of the constants. Constants can also be added or removed, and the last value changes automatically, enabling the compiler to check that arrays have also been updated when such changes are made. The symbolic names are also passed into the debugger, allowing you to use them with variables of type `constant_t`. The middle example defines specific numeric values, some of which are identical. The right example defines a set of values for use as one-bit flags.

```
typedef enum {
    CONST_A,     /* 0 */
    CONST_B,     /* 1 */
    CONST_C,     /* 2 */
    NUM_CONST    /* 3 */
} constant_t;
```

```
typedef enum {
    VALUE_A = 7, /* 7 */
    VALUE_B,     /* 8 */
    VALUE_C = 6, /* 6 */
    VALUE_D      /* 7 */
} value_t;
```

```
typedef enum {
    FLAG_A = 1,  /* 1 */
    FLAG_B = 2,  /* 2 */
    FLAG_C = 4,  /* 4 */
    FLAG_D = 8   /* 8 */
} flag_t;
```

C Operators

Basic **arithmetic operators** in C include addition (+), subtraction (−), negation (a minus sign not preceded by another expression), multiplication (*), division (/), and modulus (%). No exponentiation operator exists; instead, library routines are defined for this purpose as well as for a range of more complex mathematical functions.

C also supports **bitwise operations** on integer types, including AND (&), OR (|), XOR (^), NOT (∼), and left (<<) and right (>>) bit shifts. Right shifting a signed integer results in an **arithmetic right shift** (the sign bit is copied), while right shifting an unsigned integer results in a **logical right shift** (0 bits are inserted).

A range of **relational** or **comparison operators** are available, including equality (==), inequality (!=), and relative order (<, <=, >=, and >). All such operations evaluate to 1 to indicate a true relation and 0 to indicate a false relation. Any non-zero value is considered to be true for the purposes of tests (such as those in `if` statements and `while` loops) in C.

Assignment of a new value to a variable or memory location uses a single equal sign (=) in C. *The use of two equal signs for an equality check and a single equal sign for assignment is a common source of errors,* although modern compilers generally detect and warn about such mistakes. For an assignment, the compiler produces instructions that evaluate the right-hand expression and copy the result into the memory location corresponding to the left-hand expression *The value to the left must thus have an associated memory address.* Binary operators extended with an equal sign can be used in C to imply the use of the current value of the left side of the operator as the first operand. For example, "`A += 5`" adds 5 to the current value of the variable `A` and stores the result back into the variable.

Increment (++) and decrement (--) operators change the value of a variable (or memory location) and produce a result. If the operator is placed before the expression (a **pre-increment**), the expression produces the new, incremented value. If the operator is placed after the expression (a **post-increment**), the expression producs the original, unincremented value. The decrement operator works identically, and both of course require that the expression be stored in memory (for example, using a variable, such as A++).

The address of any expression with an address can be obtained with the **address operator** (&), and the contents of an address can be read using the **dereference operator** (*). Combining pointers and addresses with assignments confuses many. Shown to the right are a few examples translating C assignments on variables `A` and `B` into RTL. Note that the variable names in the RTL correspond to the memory addresses at which the variables are stored.

```
  A = B;   /* M[A] <- M[B]                          */
  A = *B;  /* M[A] <- M[M[B]]                       */
  A = **B; /* M[A] <- M[M[M[B]]]                    */
  A = &B;  /* M[A] <- B                             */
  A = &&B; /* not legal: &B has no address          */
  A = *&B; /* M[A] <- M[B]                          */
  A = &*B; /* M[A] <- M[B] (B must be a pointer)    */
 *A = B;   /* M[M[A]] <- M[B]                       */
**A = B;   /* M[M[M[A]]] <- M[B]                    */
 &A = B;   /* not legal: &A has no address          */
```

The C language supports **pointer arithmetic**, meaning that addition and subtraction are defined for pointer types. The compiler multiplies the value added (or subtracted) by the size of the type to which the pointer points, then adds (or subtracts) the result from the address to produce the new pointer. Thus adding one to a pointer into an array produces

a pointer to the next element of the array.

Two operators are used for **field access** with structures. With a structure, appending a period (`.`) followed by a field name accesses the corresponding field. With a structure pointer, one can dereference the structure and then access the field, but the notation is cumbersome, so a second operator (`->`) is available for this purpose. The code to the right gives a few examples using our double-date structure.

```
two_date_t  my_data;
two_date_t* my_data_ptr = &my_data;

my_data.hiring_day = 21;
(*my_data).hiring_month = 7;
my_data->hiring_year = 1998;
```

Several **logical operators** are available in C to combine the results of multiple test conditions: logical AND (`&&`), OR (`||`), and NOT (`!`). As with relational operators, logical operations evaluate to either true (1) or false (0). The AND and OR operators use **shortcut evaluation**, meaning that the code produced by the compiler stops evaluating operands as soon as the final result is known. As you know, ANDing any result with 0 (false in C) produces a false result. Thus, if the first operand of a logical AND is false, the second operand is not evaluated. Similarly, if the first operand of a logical OR is true, the second operand is not evaluated. Shortcutting simplifies code structure when some expressions may only be evaluated safely assuming that other conditions are true. For example, if you want to read from a stream `f` into an uninitialized buffer `buf` and then check for the presence of the proper string, you can write:

```
if (NULL != fgets (buf, 200, f) && 0 == strcmp (buf, "absquatulate")) {
    /* found the word!  */
}
```

Calling the string comparison function `strcmp` on an uninitialized buffer is not safe nor meaningful, thus if the file is empty, as indicated by `fgets` returning `NULL`, the second call should not be made. Shortcutting guarantees that it is not.

Example: Linked List Removal Function in C

Let's now revisit our instruction-level linked list example and develop it as a function in C. First, we'll need to extend our double-date structure to include a pointer.

As shown to the right, the type definition can precede the structure definition—in fact, by putting the type definition in a header file and keeping the structure definition in the source file containing all structure-related functions, you can prevent other code from making assumptions about how you have implemented the structure.

```
typedef struct two_date_t two_date_t;

struct two_date_t {
    short birth_day;
    short birth_month;
    short birth_year;
    short hiring_day;
    short hiring_month;
    short hiring_year;

    /* pointer to next structure in linked list */
    two_date_t* link;
};
```

The next step is to develop the **function signature** for the linked list removal function. A function signature or **prototype** specifies the return type, name, and argument types for a function, but does not actually define the function. This information is enough for the compiler to generate calls to the function as well as to perform basic checks for correct use and to generate necessary type conversions, as discussed later.

The information needed as input to our function consists of a pointer to the element to be removed and a pointer to the head of the linked list. The element being removed might be the first in the list, and our function will be more useful if it can also update the variable holding this pointer rather than requiring every caller to do so.

As you may recall, C arguments are **passed by value**, meaning that the arguments are evaluated and that the resulting values are copied (usually onto the stack) when a function is called. No C function can change the contents of a variable based only on the value of the variable, thus we must instead pass a pointer to the pointer to the head of the linked list, allowing the function to use this address to change the head as necessary:

```
/* function returns 0 on success, -1 on failure (e.g., element not in list) */
int remove_two_date (two_date_t** head_ptr, two_date_t* elt);
```

The code for the function is not much different structurally from that developed earlier for an instruction-level implementation, but allows us to express the approach a little more clearly, and with explicit information about data types:

```
/* function returns 0 on success, -1 on failure (e.g., element not in list) */
int remove_two_date (two_date_t** head_ptr, two_date_t* elt)
{
    two_date_t** ptr_to_link;
    two_date_t*  current;
    int          ret_val = -1;

    ptr_to_link = head;
    current = *ptr_to_link;
    while (NULL != current) {
        if (elt == current) {
            *ptr_to_link = elt->next;
            current      = NULL;
            ret_val      = 0;
        } else {
            ptr_to_link = &current->next;
            current     = *ptr_to_link;
        }
    }
    return ret_val;
}
```

A more succinct version is also possible using similar optimizations as before, but the syntax of multiple levels of indirection (pointers to pointers), does tend to be a little confusing at first:

```
/* function returns 0 on success, -1 on failure (e.g., element not in list) */
int remove_two_date (two_date_t** head_ptr, two_date_t* elt)
{
    two_date_t** ptr_to_link;

    for (ptr_to_link = head; NULL != *ptr_to_link; ptr_to_link = &(*ptr_to_link)->next) {
        if (elt == *ptr_to_link) {
            ptr_to_link = elt->next;
            return 0;
        }
    }
    return -1;
}
```

Changing Types in C

Changing the type of a datum is necessary from time to time, but sometimes a compiler can do the work for you. The most common form of **implicit type conversion** occurs with binary arithmetic operations. Integer arithmetic in C always uses types of at least the size of `int`, and all floating-point arithmetic uses `double`. If either or both operands have smaller integer types, or differ from one another, the compiler implicitly converts them before performing the operation, and the type of the result may be different from those of both operands. In general, the compiler selects the final type according to some preferred ordering in which floating-point is preferred over integers, unsigned values are preferred over signed values, and more bits are preferred over fewer bits. The type of the result must be at least as large as either argument, but is also at least as large as an `int` for integer operations and a `double` for floating-point operations.

Modern C compilers always extend an integer type's bit width before converting from signed to unsigned. The original C specification interleaved bit width extensions to `int` with sign changes, thus *older compilers may not be consistent, and implicitly require both types of conversion in a single operation may lead to portability bugs.*

The implicit extension to `int` can also be confusing in the sense that arithmetic that seems to work on smaller integers fails with larger ones. For example, multiplying two 16-bit integers set to 1000 and printing the result works with most compilers because the 32-bit `int` result is wide enough to hold the right answer. In contrast, multiplying two 32-bit integers set to 100,000 produces the wrong result because the high bits of the result are discarded before it can be converted to a larger type. For this operation to produce the correct result, one of the integers must be converted explicitly (as discussed later) before the multiplication.

Implicit type conversions also occur due to assignments. Unlike arithmetic conversions, the final type must match the left-hand side of the assignment (for example, a variable to which a result is assigned), and the compiler simply performs any necessary conversion. *Since the desired type may be smaller than the type of the value assigned, information can be lost.* Floating-point values are truncated when assigned to integers, and high bits of wider integer types are discarded when assigned to narrower integer types. *Note that a positive number may become a negative number when bits are discarded in this manner.*

Passing arguments to functions can be viewed as a special case of assignment. Given a function prototype, the compiler knows the type of each argument and can perform conversions as part of the code generated to pass the arguments to the function. Without such a prototype, or for functions with variable numbers of arguments, the compiler lacks type information and thus cannot perform necessary conversions, leading to unpredictable behavior. By default, however, the compiler extends any integers to the width of an `int`.

Pointer types can also be automatically converted, but such conversions are much more restrictive than those for numeric types. Automatic conversion of pointer types occurs only for assignment (and argument passing), and is only allowed when one of the pointers has type `void*`.

Occasionally it is convenient to use an **explicit type cast** to force conversion from one type to another. *Such casts must be used with caution, as they silence many of the warnings that a compiler might otherwise generate when it detects potential problems.* One common use is to promote integers to floating-point before an arithmetic operation, as shown to the right. The type to which a value is to be converted is placed in parentheses in front of the value. In most cases, additional parentheses should be used to avoid confusion about the precedence of type conversion over other operations.

```
{
    int numerator = 10;
    int denominator = 20;

    printf ("%f\n", numerator / (double)denominator);
}
```

Explicit type casts can also be used to force the compiler to treat pointers as integers, and to treat integers as addresses (cast them back to pointers). Obviously, since the number of bits required for a pointer as well as structure sizes can vary based on ISA, operating system, and compiler, such casts must be used with caution.

The C Preprocessor

The C language uses a preprocessor to support inclusion of common information (stored in header files) within multiple source files as well as a macro facility to simplify unification of source code with multiple intended purposes, such as portability across instruction set architectures.

The most common use of the preprocessor is to enable the unique definition of new types, structures, and function prototypes within header files that can then be included by reference within source files that make use of them. This capability is based on the **include directive**, `#include`, as shown here:

```
#include <stdio.h>        /* search in standard directories                    */
#include "my_header.h"    /* search in current followed by standard directories */
```

The preprocessor also supports integration of compile-time constants into the original source files before compilation. For example, many software systems allow the definition of a symbol such as `NDEBUG` (no debug) to compile without additional debugging code included in the sources.

Two directives are necessary for this purpose: the **define directive**, `#define`, which provides a text-replacement facility, and **conditional inclusion** (or exclusion) of parts of a file within `#if`/`#else`/`#endif` directives.

These directives are also useful in allowing a single header file to be included multiple times without causing problems, as C does not generally allow redefinition of types, variables, or other abstractions, even if the definitions match. Most header files are thus wrapped as shown to the right.

```
#if !defined(MY_HEADER_H)
#define MY_HEADER_H
/* actual header file material goes here */
#endif /* MY_HEADER_H */
```

The preprocessor performs a simple linear pass on the source and does not parse or interpret any C syntax. Definitions for text replacement are valid as soon as they are defined and are performed until they are undefined or until the end of the original source file. The preprocessor does recognize spacing and will not replace part of a word, thus "`#define i 5`" will not wreak havoc on your `if` statements.

Using the text replacement capabilities of the preprocessor does have drawbacks, most importantly in that almost none of the information is passed on for debugging purposes. Lists of defined constants have thus been deprecated; use an `enum` instead, which in combination with `typedef` enables your debugger to support symbolic names rather than forcing you to repeatedly translate.

In contrast, many systems still make use of parametrized text replacement, also known as **preprocessor macros**. A macro definition includes a comma-separated list of arguments after the macro name. The arguments can then be used in the replacement text. Consider, for example, the following definition of a macro to calculate the cube of a number:

```
#define CUBE(a)     ((a) * (a) * (a))     /* correct definition                        */
#define CUBE (a)    ((a) * (a) * (a))     /* no space allowed before arguments        */
#define CUBE(a)     (a * a * a)           /* CUBE (4 + 3) -> (4 + 3 * 4 + 3 * 4 + 3) */
#define CUBE(a)     (a) * (a) * (a)       /* similar potential precedence problems    */
```

Macros provide two main advantages in C. First, as C provides little support for variable types, macros provide a way to emulate such support. For example, to define the same arithmetic algorithm on both integers (faster) and floating-point numbers (better dynamic range), one has to either copy the code or make use of macros to produce the two versions. Similar usages are attractive when developing types of objects with some shared properties, such as instructions in a simulator or widgets in a drawing tool. However, as with other uses of define directives, most systems do not pass information about macros to debuggers. In languages like C++ and Java, templates or simply class hierarchies can be used instead, but sometimes at the cost of performance.

Some of the potential pitfalls with macros are shown in the examples above. More subtle errors are also possible, which is why people generally use a different convention for macro names (for example, all capital letters) than for function names. Consider, for example, "`CUBE(i++)`"—after text replacement, the variable `i` is incremented not once but three times, with unpredictable results.

Defining macros for more than expressions can be problematic due to interference between text replacement and C syntax. For example: unlike functions, macros can change the values of their arguments by simply replacing the macro call with an assignment, which can be confusing for programmers not familiar with macros.

Consider the range-limiting variable update shown to the right. The backslashes at the end of each line are necessary to tell the preprocessor that the macro definition continues on the next line. This macro avoids the argument issue by requiring a pointer to the variable to be changed. However, writing a multi-line macro such as that shown above poses several additional problems.

```
#define BACKOFF(_w)          \
        *(_w) *= 0.5;        \
        if (*(_w) < 1E-6) {  \
            *(_w) = 1E-6;    \
        }
```

For example, the macro as defined breaks if used as a simple statement:

```
if (should_backoff)
    BACKOFF (&var);            /* semicolon is NOT part of the macro */
else
    SOMETHING_ELSE (&var);
```

Wrapping the macro up as a compound statement (using braces: { ... }) does not solve the problem because of the spurious semicolon in the user's code.

Instead, we wrap the entire macro as a `do {} while (0)` loop, as shown to the right. This approach works well in the sense that the macro is fully equivalent to a function call returning `void`: it is a single statement from the compiler's point of view, but must be followed by a semicolon when used. Also, most modern compilers will not generate additional instructions for the wrapper code, even when compiling without optimization.

```
#define BACKOFF(_w)                 \
        do {                        \
            (_w) *= 0.5;            \
            if ((_w) < 1E-6) {      \
                (_w) = 1E-6;        \
            }                       \
        } while (0)
```

One final use of macros is worth discussing here: their use in supporting assertions for debugging. The GNU preprocessor used with `gcc` supports additional symbols that identify the current file and line of the macro instance, thus an assertion defined as follows can print the file name and line number of the failed assertion:

```
#define ASSERT(x)                                                               \
    do {                                                                        \
        if (!(x)) {                                                             \
            fprintf (stderr, "ASSERTION FAILED: %s:%d\n", __FILE__, __LINE__);  \
            abort ();                                                           \
        }                                                                       \
    } while (0)
```

Terminology

You should be familiar with the following terms after reading this set of notes.

- information storage in computers
  - bits
  - representation
  - data type
  - data structure
  - fields
  - size of a type
  - addressability of memory
- using memory addresses
  - pointer
  - dereference
  - pointer-array duality
  - linked list
- transforming tasks into programs
  - systematic decomposition (sequential, iterative, and conditional tasks)
  - flow chart
  - register transfer language (RTL)
  - invariant
  - stopping condition
- high-level language data types
  - strongly typed languages
  - primitive data types
  - sizeof built-in function
- structures, arrays, and new types in C
  - data structure in C
  - `struct` keyword
  - fields
  - enumerations
  - type definition
  - name space
- variables in C
  - declaration
  - definition
  - scope
  - storage class
  - shadowing
  - `static` qualifier
  - global variable
  - automatic (stack) variable
  - stack frame/activation record
- C operators
  - arithmetic
  - bitwise
  - comparison/relational
  - assignment
  - pre- and post-increment and decrement
  - address and dereference
  - pointer arithmetic
  - structure field access
  - logical
  - shortcutting
- functions in C
  - function signature/prototype
  - pass by value
- type conversion
  - implicit type conversion
  - explicit type cast
- the C preprocessor
  - include directive
  - define directive
  - preprocessor macro

ECE391: Computer Systems Engineering **Lecture Notes Set 1**

The x86 Instruction Set Architecture

This set of notes provides an overview of the x86 instruction set architecture and its use in modern software. The goal is to familiarize you with the ISA to the point that you can code simple programs (such as MP1) and can read disassembled binary code comfortably. Substantial portions of the ISA are ignored completely for the sake of simplicity. Later in the course, we may use additional portions of the ISA.

Two aspects of the notes may be important to the reader. First, they assume that you are already familiar with the LC-3 ISA developed in Patt and Patel[1] If you are not, you may want to peruse Appendix A of the book, which is available freely online. Second, the notes use the assembly notation used by the GNU tools, including the assembler `as` (used by the compiler `gcc`) and the debugger `gdb`. Other tools may define other notations, but such things are merely cosmetic so long as you pay attention to what you are using at the time.

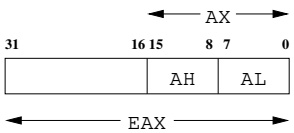
The Basics: Registers, Data Types, and Memory

You may have heard or seen the term "Reduced Instruction Set Computing," or RISC, and its counterpart, "Complex Instruction Set Computing," or CISC. While these terms were never entirely clear and have been further muddied by years of marketing, the x86 ISA is certainly vastly more complex than that of the LC-3 from Patt and Patel (ECE120/220). On the other hand, much of the complexity has to do with backwards compatibility, which is mostly irrelevant to someone writing code today. Furthermore, we need use only a limited subset of the ISA in this class, so instructions for floating-point operations and multimedia ISA extensions (MMX, for example) need only be learned if they are of interest to you.

Modern flavors of x86—also called IA32, or Intel Architecture 32—have eight 32-bit integer registers. The registers are not entirely general-purpose, meaning that some instructions limit your choice of register operands to fewer than eight. A couple of other special-purpose 32-bit registers are also available—namely the instruction pointer (program counter) and the flags (condition codes), and we shall ignore the floating-point and multimedia registers. Unlike most RISC machines, the registers have names stemming from their historical special purposes, as described below.

`%eax`	accumulator (for arithmetic and other operations)
`%ebx`	base (address of array in memory)
`%ecx`	count (of loop iterations)
`%edx`	data (example: second operand for binary operations)
`%esi`	source index (for string copy or array access)
`%edi`	destination index (for string copy or array access)
`%ebp`	base pointer (base of current stack frame)
`%esp`	stack pointer (top of stack)
`%eip`	instruction pointer (program counter)
`%eflags`	flags (condition codes and other things)

32–bit	16–bit	8–bit high	8–bit low
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		



The character "%" is used to denote a register in assembly code and is not considered a part of the register name itself; note also that register names are not case sensitive. The letter "E" in each name indicates that the "extended" version of the register is desired (extended from 16 bits). Registers can also be used to store 16- and 8-bit values, which is useful when writing smaller values to memory or I/O ports. As shown to the right above, the low 16 bits of a register are accessed by dropping the "E" from the register name. For example, `%si` is the low 16 bits of `%esi`. Finally, the two 8-bit halves of the low 16 bits of the first four registers can be used as 8-bit registers by replacing "X" with "H" (high) or "L" (low).

[1] Yale N. Patt, Sanjay J. Patel, *Introduction to Computing Systems: from Bits & Gates to C & Beyond*, 3^{rd} edition, McGraw Hill, New York, New York, 2020, ISBN13 9781260150537.

In contrast with the LC-3, for which all data types other than 16-bit 2's complement must be managed by software, the x86 ISA supports both 2's complement and unsigned integers in widths of 32, 16, and 8 bits, single and double-precision IEEE floating-point, 80-bit Intel floating-point, ASCII strings, and binary-coded decimal (BCD). Most instructions are independent of data type, but some require that you select the proper instruction for the data types of the operands. Try multiplying 32-bit representations of -1 and 1 to produce a 64-bit result, for example.

Use of memory is more flexible in x86 than in LC-3: in addition to load and store operations, many x86 operations accept memory locations as operands. For example, a single instruction serves to read the value in a memory location, add a constant, and store the sum back to the memory location. With x86, memory is 8-bit (byte) addressable and uses 32-bit addresses, although few machines today fully populate this 4 GB address space.

One aspect of x86's treatment of memory may confuse you: *it is little endian*. Little endian means that if you store a 32-bit register into memory and then look at the four bytes of memory one by one, you will find the little end of the 32 bits first, followed by the next eight bits, then the next, and finally the high eight bits of the stored value. Thus 0x12345678 becomes 0x78, 0x56, 0x34, 0x12 in consecutive memory locations. Obviously, values read from memory also use this mapping, so that reading the bytes back in as a 32-bit value produces 0x12345678 again.

A note on I/O before we get down to details: with x86, some I/O is memory mapped, and some uses a separate port space with IN and OUT instructions. The details are device-specific, and you will see plenty of such details later.

LC-3 to x86

We begin our discussion of x86 instructions by considering the x86 equivalents of the LC-3 instructions. After reading this section, you should in theory be able write a simple translator that reads an LC-3 assembly file and produces an x86 assembly file with the same effect.

Operate instructions: The LC-3 has three operate instructions: ADD, AND, and NOT. The set is sparse for simplicity, and all other operations can be built from these three, as you no doubt recall from having done so yourself. Each of these instructions specifies both its operands and a destination register.

The x86 operate instructions differ in three key ways from those of the LC-3. First, they are much more plentiful. Arithmetic operators are ADD, SUB, NEG (negate), INC (increment), and DEC (decrement); logical operators are AND, OR, XOR, and NOT; shift operators are SHL (left), SAR (arithmetic right), and SHR (logical right); finally, one can rotate bits, in other words, shift with wraparound, to the left (ROL) or to the right (ROR). The second key difference is that x86 instructions typically specify one register as both the destination and one of the sources. Thus, one can execute

```
addl %eax,%ebx   # EBX ← EBX + EAX
```

but cannot use a single ADD instruction to put the sum of EAX and EBX into ECX. The part to the right in the example above is an x86 assembly language comment showing you the interpretation of the instruction in RTL, or register transfer language, with which you should already be familiar. Finally, as you may have noticed from the example, the instruction name is extended with a label for the type of data, an "L" in the case above to indicate long, or 32-bit, operands. The other possibilities are "W" for 16-bit (word) and "B" for 8-bit (byte) operands. These markers are not required unless the operand types are ambiguous, but always using them can help to bring bugs to your attention.

Another difference that you might have noticed between LC-3 assembly code and x86 assembly code—as defined by GNU's `as` in the latter case—is that the x86 destination register appears as the last operand rather than the first. Such orderings can be assembler-specific, but keep this ordering in mind when writing x86 assembly in this class.

Operate instructions in LC-3 also allow the use of immediate values, but are usually restricted to values that fit in a handful of bits. The x86 ISA uses variable-length instructions, so immediate values of up to 32-bits are usually allowed, and values that fit into fewer bits are encoded as shorter instructions. Immediate values are preceded with a dollar sign in the assembler, thus

```
addl $20,%esp   # ESP ← ESP + 20
```

adds 20 to the current value of ESP. Numbers starting with digits 1 through 9 are treated as decimal values; numbers starting with the prefix "0x" are treated as hexadecimal values; and numbers starting with the digit 0 (but no "x") are treated as octal values.

One aspect of x86's operand flexibility may end up causing problems for you. In particular, an operand may be either a memory reference or an immediate value. For example, removing the dollar sign from the stack addition above produces:

```
addl 20,%esp   # ESP ← ESP + M[20]
```

in which the contents of memory location 20 are added to ESP rather than the value 20. *Be careful to include a "$" when you want a number interpreted as an immediate value!* Note that labels are also affected in the same way with these instructions. A label without a preceding dollar sign generates a memory reference to the memory location marked by the label. A label preceded by a dollar sign results in an immediate operand with the value of the label (typically a 32-bit value).

One last comment about operate instructions. The x86 does support a fairly general addressing mode that can be used in combination with the LEA instruction (load effective address, just like the LC-3) to support more general addition operations. In particular, the format is

```
displacement(SR1,SR2,scale)
```

which multiplies `SR2` by `scale`, then adds both `SR1` and `displacement`. For example, one can use a single LEA instruction to put the sum of EAX and EBX into ECX as follows:

```
leal (%eax,%ebx),%ecx   # ECX ← EAX + EBX
```

In this case, both `displacement` and `scale` have been left off, in which case they default to zero and one, respectively. The original purpose and limitations of this addressing mode are discussed in the next section.

Finally, the list below shows a few common operations for which one used the three operate instructions in LC-3. Two use the MOV (move) operation, which also plays the role of all LC-3 data movement instructions other than LEA.

```
movl  $10,%esi    # ESI ← 10
movl  %eax,%ecx   # ECX ← EAX
xorl  %edx,%edx   # EDX ← 0
```

Data movement instructions: The LC-3 is a load-store architecture, meaning that the only instructions that access memory are those that move data to and from registers. The LC-3's data movement instructions allow three addressing modes for loads and stores. That is, there are three ways to calculate the memory address to be used to perform a load or a store: PC-relative (LD/ST), indirect (LDI/STI), and base+offset (LDR/STR). The LEA instruction is also classified as a data movement instruction in the LC-3; its name and semantics are identical in x86, other than the fact that it supports more general addressing modes.

The x86 ISA supports both more opcodes and more addressing modes than the LC-3. Loads and stores in x86 are unified into the MOV instruction. However, as many x86 operations can use memory operands directly, not all data movement requires the use of MOV.

Most x86 addressing modes can be viewed as specific cases of the general mode described earlier (`displacement(SR1,SR2,scale)`). The purpose of this addressing mode was originally to support array accesses generated by high-level programs. For example, to access the N^{th} element of an array of 32-bit integers, one could put a pointer to the base of the array into EBX and the index N into ESI, then execute

```
movw (%ebx,%esi,4),%eax   # EAX ← M[EBX + ESI * 4]
```

If the array started at the 28^{th} byte of a structure, and EBX instead held a pointer to the structure, one could still use this form by adding a displacement:

```
movw 28(%ebx,%esi,4),%eax   # EAX ← M[EBX + ESI * 4 + 28]
```

The `scale` can take the values one, two, four, and eight, and defaults to one. Examples of how one can use this mode and its simpler forms include the following:

```
movb  (%ebp),%al                # AL ← M[EBP]
movb  -4(%esp),%al              # AL ← M[ESP - 4]
movb  (%ebx,%edx),%al           # AL ← M[EBX + EDX]
movb  13(%ecx,%ebp),%al         # AL ← M[ECX + EBP + 13]
movb  (,%ecx,4),%al             # AL ← M[ECX * 4]
movb  -6(,%edx,2),%al           # AL ← M[EDX * 2 - 6]
movb  (%esi,%eax,2),%al         # AL ← M[ESI + EAX * 2]
movb  24(%eax,%esi,8),%al       # AL ← M[EAX + ESI * 8 + 24]
```


Consider the LC-3 addressing modes again for a moment. Clearly the register indirect (LDR/STR) mode is covered by the modes just considered. The indirect mode (LDI/STI) is not covered, but that mode is easily replaced by a pair of data movement instructions. The PC-relative mode (LD/ST) is also technically unavailable, but its uses are covered by the availability of a direct addressing mode in x86, in which the address to be used is specified as an immediate value in the instruction. The examples below show a number of ways in which data can be moved to and from specific address (often represented by labels in assembly code).

```
movb  100,%al                    # AL ← M[100]
movb  label,%al                  # AL ← M[label]
movb  label+10,%al               # AL ← M[label+10]
movb  10(label),%al              # NOT LEGAL!

movb  label(%eax),%al            # AL ← M[EAX + label]
movb  13+8*8-35+label(%edx),%al  # AL ← M[EDX + label + 42]

movl  $label,%eax                # EAX ← label
movl  $label+10,%eax             # EAX ← label+10
movl  $label(%eax),%eax          # NOT LEGAL!
```

The middle two examples above return to the more general addressing mode and demonstrate some of the GNU assembler's capabilities in terms of evaluating expressions. Replacing MOVB with LEAL (and AL with EAX) in any of the first two groups of examples results in EAX being filled with the address used for the load. Putting an immediate marker ("$") in front of the label (as in the last group) has the same effect for some forms, but is not always legal.

Condition codes: The LC-3 supplies three condition codes based on ALU results for the purposes of conditional branches. These codes—negative, zero, and positive—are somewhat unusual in that exactly one is set at any time. The x86 flags do not have this property.

For our purposes, the x86 has five relevant condition codes. The sign flag (SF) is the same as the negative (N) flag in the LC-3: it records whether the last result represented a negative 2's complement integer (had its most significant bit set). The zero flag (ZF) is the same as the zero (Z) flag in the LC-3: it records whether the last result was exactly zero. The carry flag (CF) records whether the last result generated a carry or required a borrow, but is also used to hold bits shifted out with shifts, to record whether the high word of a multiplication is non-zero or not, and other such things (in other words, there are far too many instruction-specific effects to list here). The overflow flag (OF) records whether the last operation overflowed when interpreted as a 2's complement operation, and serves additional purposes in the same fashion as CF. Finally, the parity flag (PF) records whether the last result had an even number of 1's or not; it is set for even parity, and clear for odd parity.

One aspect of the x86 ISA's use of flags is important to keep in mind when writing code: *not all result-producing instructions affect the flags, and not all flags are affected by instructions that affect some flags*. That said, most instructions mentioned so far affect all flags, The exceptions include MOV, LEA, and NOT, which affect no flags; ROL and ROR, which affect only OF and CF; and INC and DEC, which affect all but CF.

In order to set the flags based on the result of a MOV or LEA (or any other instruction that doesn't affect the flags), use either a CMP (compare) or TEST instruction to set the flags first. The CMP instruction performs a subtraction, subtracting its first argument from its second, and sets the flags based on the result; nothing else is done with the result. The TEST instruction performs an AND operation between its two operations, sets the flags accordingly (OF and CF are cleared; SF, ZF, and PF are set according to the result), and discards the result of the AND.

Conditional branches: The LC-3 allows conditional branches based on any disjunction (OR) of the three condition codes. Eight basic branch conditions and their inverses are available with the x86 ISA, along with the unconditional branch JMP. These branches are described below, along with the conditions under which the branch is taken.

`jo`	overflow	OF is set	`jb`	below	CF is set
`jp`	parity	PF is set (even parity)	`jbe`	below or equal	CF or ZF is set
`js`	sign	SF is set (negative)	`jl`	less	SF $\neq$ OF
`je`	equal	ZF is set	`jle`	less or equal	(SF $\neq$ OF) or ZF is set

The sense of each branch other than JMP can be inverted by inserting an "N" after the initial "J." For exmaple, JNB jumps if the carry flag is clear. Furthermore, many of the branches have several equivalent names. For example, JZ (jump if zero) can be written in place of JE (jump if equal). Unsigned comparisons should use the "above" and "below" branches, while signed comparisons should use the "less" or "greater" branches, as shown in the table below. The preferred forms are those that the debugger uses when disassembling code.

	jnz	jnae	jna	jz	jnb	jnbe	unsigned comparisons
preferred form	jne	jb	jbe	je	jae	ja	
	$\neq$	$<$	$\leq$	$=$	$\geq$	$>$	
preferred form	jne	jl	jle	je	jge	jg	signed comparisons
	jnz	jnge	jng	jz	jnl	jnle	

The table should be used as follows. After a comparison such as

```
cmp %ebx,%esi   # set flags based on (ESI - EBX)
```

choose the operator to place between ESI and EBX, based on the data type. For example, if ESI and EBX hold unsigned values, and the branch should be taken if ESI $\leq$ EBX, use either JBE or JNA. If ESI and EBX hold signed values, and the branch should be taken if ESI $>$ EBX, use either JG or JNLE. For branches other than JE/JNE based on instructions other than CMP, you should check the stated branch conditions rather than trying to use the table.

Other control instructions: The LC-3 also provides six other control instructions: JMP, JSR, JSRR, RET, RTI, and TRAP. These instructions support indirect jumps, subroutine calls and returns, returns from interrupts, and system calls (sometimes called traps).

Two types of instructions are left for later sets of notes. The x86 INT instruction provides support for system calls through a mechanism almost identical to that defined for the LC-3 TRAP instruction. However, understanding its use in a more modern operating system requires some discussion. Similarly, while you will have opportunities to define and use interrupts in this class, and thus to make use of x86's return from interrupt (IRET) instruction, the issues involved are beyond the scope of these first notes.

Subroutine control is more straightforward. The CALL instruction in x86 plays the role of the LC-3 subroutine call instructions, JSR and JSRR. The single operand of a CALL can be either direct or indirect; indirect operands are preceded by an asterisk:

```
call  printf            # (push EIP), EIP ← printf
call  *%eax             # (push EIP), EIP ← EAX
call  *(%eax)           # (push EIP), EIP ← M[EAX]
call  *fptr             # (push EIP), EIP ← M[fptr]
call  *10(%eax,%edx,2)  # (push EIP), EIP ← M[EAX + EDX*2 + 10]
```

The CALL instruction pushes the return address onto the stack before changing the instruction pointer. Its counterpart, the RET (return) instruction, then pops the return address off the stack and into EIP in order to return from the called routine. The calling convention is described in greater detail later in these notes.

The unconditional branch instruction JMP mentioned earlier also takes the role of the indirect jump instruction in x86, using the same syntax as shown above for the CALL instruction.

Labels, comments, directives, and pseudo-ops: Several differences between syntax in GNU's `as` and the LC-3 assembler have already been mentioned, but are reviewed here for completeness.

Labels can begin with any letter, a period, or an underscore. Characters after the first can also include numbers. Later characters can also include dollar signs, but introducing the context-specific meaning can be confusing. Is it part of a label, or an immediate value marker? *Each label definition must be followed by a colon*; uses of a label do not include this colon, and it is not considered to be part of the label. *Labels are case sensitive.* Labels are the only case sensitive aspect of the `as` assembler mentioned in these notes. Finally, if you look at assembly code generated by the `gcc` compiler, you will notice that it starts its label names with a period; if you want to mix your code with code generated from C, you may want to avoid starting your labels with periods.

Comments can take two forms. The examples in these notes so far have used a form similar to that found in the LC-3 assembler. In particular, the assembler ignores everything on a given line after the first pound sign ('#'). *The*

semicolon (";") used for comments in the LC-3 assembler does not start a comment in `as`. Instead, the semicolon separates x86 instructions grouped onto a single line, as shown here:

```
movw my_data_ptr,%eax ; movw (%eax),%eax   # EAX ← M[M[my_data_ptr]]
```

Multi-line comments are also allowed with `as`, using C-style demarcation:

```
/* A comment of this form
   can span multiple lines.  */
```

The set of assembler directives and pseudo-operations that you need for this class is comparable to that used with the LC-3. We first cover those that are absent or unnecessary. The equivalent of the .ORIG directive exists, but is not necessary; code placement is better left to the linker. The .END directive is also unnecessary; a file ends when it ends. The trap pseudo-ops are not available; their equivalents do exist in DOS and BIOS, but are not as readily accessible from within Linux, although the application-level interface is quite similar.

The remaining directives are all fairly useful. The x86 assembler supports both .GLOBAL and .EXTERN to declare symbols to be visible externally and to be defined externally, respectively. The .EXTERN directive is technically unnecessary: the assembler assumes that any undefined symbol is defined elsewhere. *This assumption implies that the assembler cannot identify undefined symbols until link time!* Take the time to figure out what a missing symbol looks like as a linker error in advance so that you don't have to guess when you see the error later.

The LC-3 .STRINGZ directive becomes .STRING. Empty space can be created using the .SPACE directive, which requires a first argument specifying the number of bytes and accepts an optional second argument specifying the byte value to use as filler. The .FILL directive takes many new forms—actually, the .FILL directive exists as well, but does not have the same meaning, and you probably never want to use it. Examples of these new directives are shown below:

```
.byte    12,-15      # 8-bit values
.word    200,4000    # 16-bit values
.long    -987654321  # 32-bit values
.quad    9999999999  # 64-bit values
.single  1.0,2.0     # single-precision IEEE floating-point
.double  2.0,3.1415  # double-precision IEEE floating-point

# alternate forms
.hword   22000,-17   # 16-bit values
.int     1,4,9,0x16  # 32-bit values
.float   2.7         # single-precision IEEE floating-point
```

Another useful directive is worth mentioning here: .INCLUDE. This directive tells the assembler to read in the contents of another file and to insert it in place of the directive. It is equivalent to the C preprocessor's `#include` directive.

Input and output: The LC-3 uses memory-mapped I/O for all devices, as do most modern architectures. In contrast, older architectures like the x86 were originally designed with separate I/O name spaces and special instructions for accessing them. Originally, the 8086 communicated only through devices such as the serial port, to which one could attach a terminal for displaying the output and a keyboard for driving the input to the processor. Someone soon realized that one could drop a display card with memory into the system bus and take over part of the memory's address space without changing the processor, however, and ever since, x86-based desktop computers have used both memory-mapped and instruction-based I/O.

The I/O instructions have not changed substantially since the original 8086 ISA, and require the use of specific registers. In particular, while one can now write a 32-bit word to a sequence of I/O ports, the data must still be in the EAX register (AX for 16-bit, or AL for 8-bit). Data from ports are also read into EAX. Similarly, the port number to be used can be specified as either an 8-bit immediate or loaded into DX (the port space is 16-bit). The examples below use the notation P[x] to denote port x.

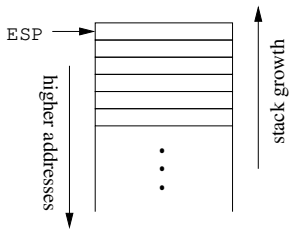
```
inb   $0x40,%al  # AL ← P[0x40]
inw   (%dx),%ax  # AL ← P[DX], AH ← P[DX + 1]
outb  %al,(%dx)  # P[DX] ← AL
outw  %ax,68     # P[68] ← AL, P[69] ← AH
```

As illustrated explicitly in the examples above, the port address space is treated much like memory with x86. In particular, it is both byte-addressable and little endian. Thus writing 16 bits to a port writes the low 8 bits to the named port, and the high 8 bits to the next port. Finally, like MOV, neither IN nor OUT affects the flags.

Other Useful Instructions

The last section focused primarily on x86 instructions similar to those available in the LC-3. This section introduces a few types of instructions that require more substantial software support with LC-3.

Stack operations: The x86 ISA supports a stack abstraction directly rather than as a software convention. The PUSH and POP instructions provide the necessary functionality. As shown to the right, the stack convention used is that ESP contains the address of the element on top of the stack. The stack grows downward in addresses, like that of the LC-3, so




```
pushl %eax   # M[ESP - 4] ← EAX, ESP ← ESP - 4
```

is equivalent to

```
movl  %eax,-4(%esp)  # M[ESP - 4] ← EAX
subl  $4,%esp        # ESP ← ESP - 4
```

Other than a POP into the EFLAGS register, PUSH and POP do not affect the flags.

Multiplication and division: Signed and unsigned forms of integer multiplication and division are available in the x86 ISA. The unsigned multiply (MUL) requires that EAX be one of the operands (or AX, or AL), and places the high bits of the result in EDX, and the low bits in EAX (or DX:AX, or AX). Both signed (IDIV) and unsigned (DIV) division have similar restrictions. The dividend must be placed in EDX:EAX (or DX:AX, or AX). After an IDIV instruction, EAX (or AX, or AL) holds the quotient, and EDX (or DX, or AH) holds the remainder. If the quotient overflows the destination register, an exception is generated.

Signed multiplication is more flexible. Although the instruction formats supported with unsigned multiplication are also available, signed multiplication also allows two- and three-operand forms, as shown below. In these forms, the high bits of the product are discarded.

```
imull  %ebx,%eax         # EAX ← EAX * EBX
imull  $1000,%ebx,%eax   # EAX ← 1000 * EBX
```

The flags are undefined after division, and only the CF and OF flags have a meaning with multiplication (note that the other flags are undefined, not unaffected; do not expect them to retain their previous values). With multiplication, both CF and OF are set whenever the high bits of the result are non-zero.

Data type conversions: The MOV instruction can also be used to convert small integers into larger ones through sign or zero extension. Converting large integers into smaller ones is usually done by simply using other register names (such as AX or AL for a value in EAX). After MOV, add either "S" for sign extension or "Z" for zero extension, then a letter for the original size, and a letter for the final size. For example, MOVZBL zero extends a byte from memory or another register into a long (32 bits). Special forms are available for EAX: CBTW converts signed byte AL to word AX, CWTL converts signed word AX to long word EAX, and CLTD converts signed long word EAX to double word EDX:EAX. The last is useful in preparing for IDIV. Be careful about CWTD: it exists, but changes AX into DX:AX.

The Calling Convention

Writing x86 assembly that interfaces with high-level languages such as C requires that one understand the calling convention for the ISA. This section begins with a description of the rules for passing and returning values and register ownership in the x86 calling convention, then provides an example in which a C function and a use of that function are translated into x86 assembly code.

Parameters, return values, and registers: Parameters passed to a function are pushed onto the stack with x86. In most high-level languages, parameters are pushed from right to left to allow for a variable number of parameters without requiring additional space for parameter counts or sentinels.

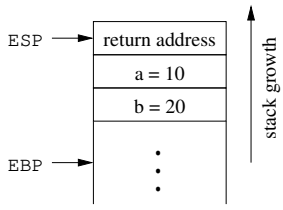
The location of the value returned from a function depends on the type of the value being returned. For pointers and integers requiring no more than 32 bits, the return value is placed in EAX. Integers and other non-floating-point types of more than 32 but no more than 64 bits are split between EDX (high bits) and EAX (low bits). Floating-point values are returned on the top of the floating-point stack (not discussed in these notes).

Most of the registers are considered to be owned by the caller. Both the stack and the frame pointer, ESP and EBP, must be returned unchanged. Similarly, EBX, ESI, and EDI are callee-saved. The return values EAX and EDX must obviously be saved by the caller, if they are to be preserved. The ECX register is also caller-saved (as is EFLAGS).

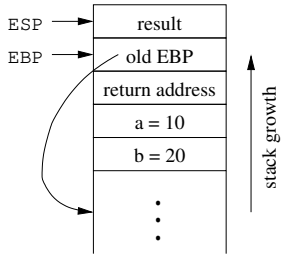
```
/* call the function */    # the call site...
value = a_func (10, 20);        pushl  $20             # push second argument
                                pushl  $10             # push first argument
                                call   a_func          # call the function
                                addl   $8,%esp         # pop the arguments
                                movl   %eax,-4(%ebp)   # store result in 'value'

int                           a_func:
a_func (int a, int b)           pushl  %ebp            # save old frame pointer
{                               movl   %esp,%ebp       # point to new frame
   int result;                  subl   $4,%esp         # make room for 'result'
                                movl   8(%ebp),%eax    # put 'a' into EAX
   result = a * b + 1;          imull  12(%ebp),%eax   # multiply EAX by 'b'
                                incl   %eax            # add one
   return result;               movl   %eax,-4(%ebp)   # store into 'result'
}                               movl   -4(%ebp),%eax   # return 'result' in EAX
                                leave                  # restore frame pointer
                                ret                    # return
```

Caller side: Consider the code shown above in which the C function `a_func` is called by another piece of code. The C versions of the call site and the function appear on the left, with their translations to the right. For clarity, the assembly code is not optimized.

To prepare for a call, the caller first pushes the function parameters onto the stack. In this case, the immediate values 20 and 10 are pushed. The CALL instruction is then executed, which pushes the next EIP (pointing to the ADD after the CALL) onto the stack and changes the EIP to the start of the function. The diagram to the right illustrates the stack pointer values at the start of the function. The top of the stack (ESP) holds the return address, under which are the function parameters (often called **formals** within the function). The EBP register still points to the stack frame of the caller. The function `a_func` then executes as described in the next section, returning the two stack registers to the values shown at the right before executing the RET instruction. The RET instruction pops the return address from the stack. The caller then removes the parameters from the stack using an ADD and stores the return value from EAX into the appropriate local variable (relative to its own frame pointer).



Callee side: The instructions in the function can be broken into three groups: one to set up the function's stack frame, a second to implement the function, and a third to tear down the stack frame. The state of the stack during function execution is shown in the diagram to the right, so the stack frame set up code shifts from the diagram above to the one to the right, and the tear down code returns the stack state to that shown above.



Set up begins by pushing the old frame pointer (EBP) onto the stack, then copying ESP into EBP. A copy of the old frame pointer thus sits at the base of the new stack frame, and the new frame pointer points to it, effectively forming a linked list of stack frames. If any callee-saved registers (EBX, ESI, and EDI) are used by the function implementation, they are then pushed onto the stack (none are used by `a_func`). Finally, the stack pointer is updated to make room for local variables.

Tearing down the stack frame requires only a single instruction when no callee-saved registers have been preserved. The LEAVE instruction restores the old values of EBP and ESP (in RTL, `ESP ← EBP + 4, EBP ← M[EBP]`). When callee-saved registers must be restored, an LEA instruction points the ESP to the uppermost saved register, then a sequence of POPs (the last into EBP) restores the original stack state.

Miscellany

Data type alignment: Most architectures support byte-addressable memory. However, when moving data to and from memory, they often restrict the address used to even multiples of the data type. For example, 16-bit values can only be written or read from even addresses, and 32-bit values require addresses that are multiples of four (32 bits is four bytes). The x86 ISA technically does not require such alignment, but only because the 8086 did not impose any restrictions. From a performance standpoint, however, unaligned loads and stores are extremely slow. The rationale is that anyone running software old enough to contain unaligned accesses could not possibly care about the performance of that software, given that the clock speed of the processor is already three orders of magnitude faster.

The implication for your programs is that arrays of data (as well as code, but for slightly different reasons) should be properly aligned in memory. To ensure proper alignment, use the .ALIGN directive, which takes a single argument and inserts enough blank space to reach the next address that is a multiple of the argument. For example, if you want to declare an array of 32-bit numbers, you should write something similar to the following:

```
            .align  4  # the label my_array (an address) is a multiple of 4
my_array:   .long   100000,4000000000,24,0
```

Support for larger data types: Larger integer data types can be constructed from the existing types by operating on the larger values word by word and using the carry flag to simulate larger adders, multipliers, and so forth. For example, to add two 64-bit numbers, first add the low 32 bits with ADDL, then use ADCL (add with carry, long) to add the high 32 bits along with the carry flag. If you are interested in such things, take a look at add with carry (ADC), subtract with borrow (SBB), and rotate with carry (RCL/RCR).

Getting More Information

Many sources of information are available if you want to learn more about the x86 ISA or implementations thereof. You may want to start with Patt and Patel's Appendix B (available from our class web page), which will also give you some insights as to encoding, which is beyond the scope of this course.[2] You can also find ISA manuals (from Intel, or AMD for AMD-64) online; official manuals for the x86 family are available at `http://www.x86.org/intel.doc/` and many informal guides are available elsewhere. The mnemonics and syntax may differ, but the instruction sets are the same, so you should be able to figure out how to get an instruction to work once you know that it exists.

For information about the GNU assembler, the best source is the info page: type "info as" at the prompt in one of your virtual machines; the info interface is much like that of `emacs`, so hopefully you're comfortable with that interface. For many instructions, calling conventions, function prototypes, and so forth, you can simply write the code you want in C and have gcc compile it to assembly for you with the -S flag. Don't use -o with -S; by default, gcc will create a new file ending in .s, but you can (and should not) override it with -o. Also beware of clobbering a modified .s file! Much of the information in this set of notes was gathered with -S.

A Note on the Notes

We use boldface in these notes to highlight definitions, and italics to emphasize pitfalls. With the possible exceptions of formals and binary-coded decimal, neither of which is critical to the course, none of the terms in this set of notes should be new to someone who has taken both ECE120 and ECE220, but the possible pitfalls in switching assemblers and syntax are numerous.

[2] An anonymous faculty member suggested that I teach you how to write self-modifying code so as to allow you to replicate some of the graphical game work done before graphics cards supported rendering in hardware. Feel free to learn the encoding, if you would like to try such things.

ECE391: Computer Systems Engineering **Lecture Notes Set 2**

Interrupts and Synchronization

These notes cover interrupts and topics relevant to their generation, control, and handling within the Linux kernel on an x86-based computer. The notes begin with a discussion of interrupts and their relationship to system calls and exceptions. The next few sections describe how interrupts are supported within a processor, including necessary aspects of the processor's state machine, external inputs, and architecturally visible (*i.e.*, ISA-level) data structures and registers. We recall the LC-3 implementation as a familiar example before introducing the analogous x86 features. We then consider the problem of sharing data between interrupt handlers and programs, touching briefly on how these issues are exacerbated in the presence of multiple processors. Several synchronization constructs are presented. The notes then continue with discussion of the logic necessary to connect many devices to a processor's single interrupt input, focusing on the Intel 8259A chip used by or emulated in most x86-based systems. After describing how both the interrupt controller and interrupts are abstracted within the Linux kernel, the notes conclude with an example and a few references and pointers for further study.

A Note on the Notes

As mentioned in the last set of notes, a bold font highlights definitions, and italicization emphasizes pitfalls. Linux sources are given with path names relative to the Linux source directory when first referenced. Header files are given with path names relative to the `include` subdirectory. References to Patt and Patel are to the second edition (the differences are fairly minor).

System Calls, Interrupts, and Exceptions

As you may recall from ECE220, system calls (also called traps), interrupts, and exceptions are all quite similar to procedure calls. **System calls** are almost identical to procedure calls. As with procedure calls, a calling convention is used: before invoking a system call, arguments are marshaled into the appropriate registers or locations in the stack; after a system call returns, any result appears in a pre-specified register. The calling convention used for system calls need not be the same as that used for procedure calls, and is typically defined by the operating system rather than the ISA. The details of this convention are generally handled by library code, however, which maps the ISA's calling convention into that of the OS' system calls. Rather than a CALL or JSR instruction, system calls are usually initiated with an INT or TRAP instruction. With many architectures, including the x86, a system call places the processor in privileged or kernel mode, and the instructions that implement the call are considered to be part of the operating system. The term system call arises from this fact.

Unexpected processor interruptions can occur due to interactions between a processor and external devices or to erroneous or unexpected behavior in the program being executed. The term **interrupt** is reserved for asynchronous interruptions generated by other devices, including disk drives, printers, network cards, video cards, keyboards, mice, and any number of other possibilities. **Exceptions** occur when a processor encounters an unexpected opcode or operand. An undefined instruction, for example, gives rise to an exception, as does an attempt to divide by zero. Exceptions usually cause the current program to terminate, although many operating systems allow the program to catch the exception and to handle it more intelligently. The table below summarizes the characteristics of the two types and compares them to system calls.

type	generated by	example	asynchronous	unexpected
interrupt	external device	packet arrived at network card	yes	yes
exception	invalid opcode or operand	divide by zero	no	yes
system call/trap	deliberate, via INT instruction	print character to console	no	no

Interrupts occur asynchronously with respect to the program. Most designs only recognize interrupts between instructions, *i.e.*, the presence of interrupts is checked only after completing an instruction rather than in every cycle. In pipelined designs, however, several instructions execute simultaneously, and the decision as to which instructions occur "before" an interrupt and which occur "after" must be made by the processor. Exceptions are not asynchronous

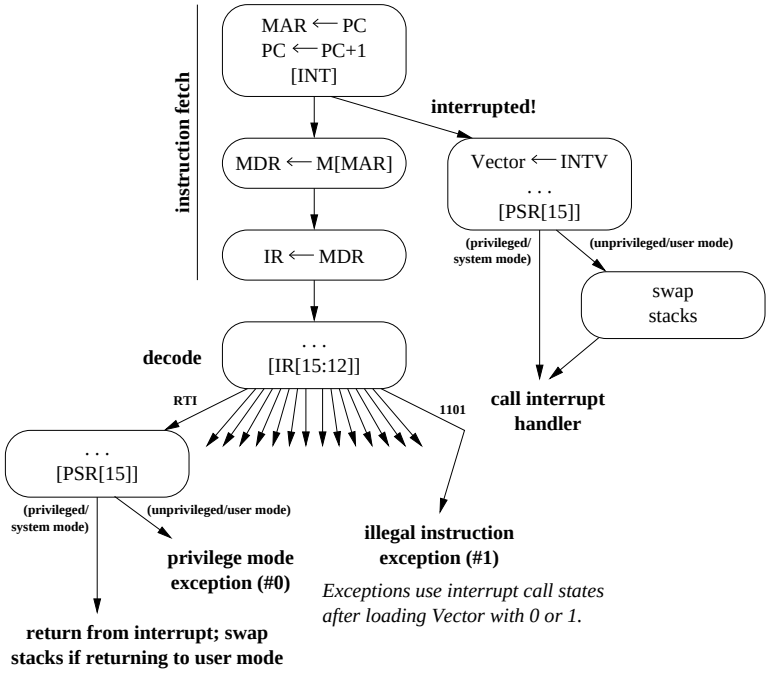
in the sense that they occur for a particular instruction, thus no decision need be made as to instruction ordering. After determining which instructions were before an interrupt, a pipelined processor discards the state of any partially executed instructions that occur "after" the interrupt and completes all instructions that occur "before." The terminated instructions are simply restarted after the interrupt completes. Handling the decision, the termination, and the completion, however, increases the design complexity of the system.

The code associated with an interrupt, an exception, or a system call is a form of procedure called a **handler**, and is found by looking up the interrupt number, exception number, or trap number in a table of function pointers called a **vector table** (or jump table). Separate vector tables can exist for each type (interrupts, exceptions, and system calls), or can be shared. The x86 ISA, unlike the LC-3, uses a single common table.

Processor Support for Interrupts

We now discuss the interrupt support provided by the processor in more detail, first by recalling the extensions made to the basic LC-3 design, then by discussing their analogues in the x86 ISA.

Interrupts in the LC-3: As you may recall, the LC-3's interrupt support is covered only briefly in ECE220. The portion of the LC-3 state machine shown to the right is based on Figure C.7 of Patt and Patel, but has been simplified and annotated to highlight the elements most relevant to our course.



The LC-3 uses a priority encoder to map up to eight device interrupt requests into a three-bit priority level, then raises an interrupt generation signal (INT) internally when an interrupt should occur. The prioritization allows the LC-3 to prevent interrupts with equal or lower priority from interrupting the execution of more important interrupts.

The first state in the instruction fetch sequence checks INT, and, if it is high (is a 1, meaning that it has been asserted by some device), the LC-3 initiates an interrupt handler call. If not, it continues with instruction fetch. Once instruction fetch has progressed to the second cycle, interrupts are ignored until the instruction has been fully processed. The LC-3, like almost all processors, does not allow interrupts to split individual instructions.

Calling an interrupt handler involves reading an eight-bit interrupt vector (INTV in the diagram) supplied as input to the processor when an interrupt is requested, and using this value as an index in the interrupt vector table, an array of interrupt handler addresses. The interrupt vector table is identical in form to the trap vector table (with which you are more familiar), and resides at memory locations 0x100 through 0x1FF. A few other operations are also necessary, such as saving the return address, processor status, and flags on the stack; possibly swapping stacks (discussed later, in the resource-sharing section); and adjusting the PC.

The interrupt handler ends with the RTI instruction, which appears in the lower left of the diagram. The processing for this instruction restores the processor status and flags at the time of the interrupt, swaps the stack if necessary, and resets the PC to point to the instruction that was about to execute when the interrupt occurred.

The LC-3 handles exceptions in the same way that it handles interrupts. Exceptions are generated when an instruction with an illegal opcode is executed, or when the RTI instruction is used outside of an interrupt handler (detected using the privilege bit in the status register). In both cases, after selecting the appropriate vector number, the state machine transitions into the interrupt handler sequence to the right.

Interrupts in the x86: By design, the LC-3 uses a simplified version of the x86 interrupt support. The basic elements are the same. The x86's INTR input indicates that some device has requested an interrupt. In most x86 implementations, these interrupts are prioritized and masked according to priority by an external interrupt controller, which allows for a more flexible prioritization scheme. In some recent implementations, interrupt controller functionality is integrated into the processor.

The x86 allows software to block all interrupts requested by the INTR pin using a processor status flag. If the **interrupt enable flag** (**IF**) is set in the flags register, interrupts are allowed to occur. If it is clear, interrupts are masked regardless of their priority. The STI and CLI instructions change the value of IF. The external interrupt controller (discussed later) also masks equal and lower priority interrupts until the processor indicates that the interrupt handler has completed, so these interrupts are masked whether or not the IF flag is clear.

Unlike the LC-3, the x86 also has a second interrupt input, NMI, that cannot be masked by setting a status flag. These **non-maskable interrupts** are used to indicate serious conditions such as parity failure in memory, critically low energy levels in batteries, *etc.*, and will not be addressed in great detail in our course, as most of the hardware and software mechanisms involved are nearly identical to those used for normal interrupts. For more information, read the file `arch/i386/kernel/nmi.c`.

The x86 uses a single vector table called the **Interrupt Descriptor Table**, or **IDT**, for interrupts, exceptions, and system calls. Although this table is a 256-entry array indexed by vector number, the elements are not simply code addresses, but also include some information about privilege level and other topics to be covered later in this course. For now, however, you can view the IDT as a simple table of code addresses pointing to interrupt, exception, and system call handlers. Each of these ends with an IRET instruction.

As with the LC-3, the exception vectors are specified as part of the ISA; Intel reserves the first 32 values for this purpose. The rest of the table is left to the operating system, although the interrupt controller hardware (discussed later) does restrict interrupt handlers to two contiguous blocks of eight values. Under Linux, the IDT is structured as shown to the right. The abbreviation IRQ stands for interrupt request.

0x00–0x1F defined by Intel	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
	0x04	overflow	
	⋮		
	0x0B	segment not present	
	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
0x20–0x27 primary 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to secondary)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F secondary 8259 PIC	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
	0x2C	IRQ12 — PS/2 mouse	
	0x2D	IRQ13	
	0x2E	IRQ14 — ide0 (hard drive)	
	0x2F	IRQ15	
0x30–0x7F	⋮	APIC vectors available to device drivers	
0x80	0x80	system call vector (INT 0x80)	
0x81–0xEE	⋮	more APIC vectors available to device drivers	
0xEF	0xEF	local APIC timer	
0xF0–0xFF	⋮	symmetric multiprocessor (SMP) communication vectors	

Input and Output

An interrupt usually occurs when a device needs attention. For example, interrupts occur when a key is pressed, when the mouse is moved, and when a block of data is available from the disk. The handler code that executes on behalf of a device interacts with the device using through the processor's input/output, or I/O interface. The interrupts themselves are in fact a special form of I/O in which only the signal requesting attention is conveyed to the processor, but let's leave the details of that signaling for now and recall how a processor sends and receives data from devices.

Communication of data occurs through instructions similar to loads and stores. A processor is designed with an **I/O port space** similar to a memory address space. Devices are connected to the processor through a bus (or several buses), and each device is associated with some port or set of ports. Reads and writes to device registers are then transmitted by the processor on the bus using the port numbers as addresses. When a device recognizes a bus transaction targeting one of its ports, it must respond appropriately. For example, a device may deliver data onto the bus in response to a read and record the data written in a register in response to a write.

The question remains as to exactly how I/O ports are accessed in software. One option is to create special instructions, such as the IN and OUT instructions of the x86 architecture. Port addresses can then be specified in the same way that memory addresses are specified, but use a distinct address space. Just as two sets of special-purpose registers can be separated by the instructions of an ISA, such an **independent I/O** system separates I/O ports from memory addresses by using distinct instructions for each class of operation.

Alternatively, device registers can be accessed using the same load and store instructions as are used to access memory. This approach, known as **memory-mapped I/O**, requires no new instructions for I/O, but demands that a region of the memory address space be set aside for I/O. The memory words with those addresses, if they exist, can not be accessed during normal processor operations.

As mentioned in the last set of notes, most x86-based computers use both models, with some devices fully mapped into the port space, others fully mapped into memory, and still others with registers in both spaces.

Shared Data and Resources

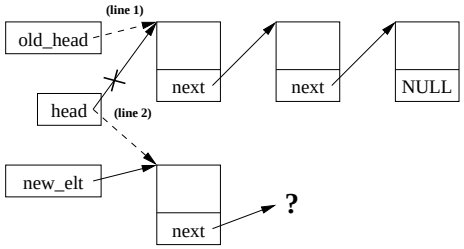
The asynchronous and unexpected nature of interrupts with respect to normal program execution can lead to problems if data and resources shared by the two pieces of code are not handled carefully. This section discusses a wide range of possible problems, ranging from the obvious to the subtle. Solutions to most issues are also given, while subsequent sections describe techniques for correctly managing some of the more complex issues.

Like any other type of procedure, interrupt handlers must preserve the contents of registers and must avoid overwriting memory locations used by the interrupted program. Unlike procedures or system calls, however, interrupts must preserve all registers used. The program does not expect a register to be overwritten between instructions just because the procedure calling convention specifies it as being caller-saved; no call was made! Registers are typically preserved by writing them onto a stack and restoring their values before returning from the interrupt. If an interrupt handler needs additional memory for storage, memory locations can be dedicated to the handler's private use.

Some less obvious shared resources include flag and status registers as well as any data structures (in memory) used to convey information between the interrupt handler and the program. Flag and status registers can be saved to the stack along with all other registers, either by the handler or by the processor itself. As the LC-3 has no instruction capable of loading or storing the Process Status Register (PSR), the processor itself writes the PSR to the stack on an interrupt and restores it when executing an RTI instruction. In the x86 ISA, the EFLAGS register can be pushed and popped, but is also usually saved and restored automatically.

Sharing data structures between a program and an interrupt handler can be complicated, even if the interrupt handler is not changing the data structure. Consider a handler that walks down a linked list. The handler interrupts the code at the top of the next page, which places `new_elt` at the head of the list, between lines 2 and 3. The figure to the right of the code illustrates the state of the list before the code executes (solid lines) and after the first two lines have executed (dashed lines are new). The question mark represents the uninitialized `next` field of `new_elt`. The interrupt handler starts from `head` and follows the dashed line to oblivion.

```
/* line 1 */ old_head = head;
/* line 2 */ head = new_elt;
              /* INTERRUPT OCCURS HERE! */
/* line 3 */ new_elt->next = old_head;
```



The example above illustrates a problem at the level of C source statements, but interrupts occur between instructions, which may or may not be part of the same statement in the source language. You may have noticed that this example can be rewritten to make the change to the list with a single instruction (a store to `head`). In general, all data structure operations can be written in such a manner, but not always without making the operations slow. One must also work around the compiler, which is allowed to reorder and even to intermingle instructions corresponding to statements in high-level languages in the interest of improving performance. Clearly, we need some mechanism to keep interrupts from happening in certain parts of the program.

The most subtle interactions between programs and interrupts involve multi-cycle operations and security. For the first case, consider an LC-3 program that prints a string to the display, and a keyboard interrupt handler that echoes keystrokes back to the display. The program has read the display status and found that it is ready to accept another character, but before writing that character to the display's data register, it is interrupted by a keystroke. The interrupt handler reads in the key and echoes it back to the display, then returns control to the program. The display takes some time to deal with the echoed keystroke, and thus ignores the character sent by the program. From the program's point of view, the display status register magically changed from ready to not-ready between instructions.

A similar type of error can be inserted into your program through compiler optimizations. In order to make the common case fast, compilers ignore the possibility of interrupts affecting memory under the control of the program. Typically, such assumptions do not lead to problems. However, if you deliberately share data in this manner, you may need to tell the compiler. As a simple example, you could create a variable and initialize it to zero, then wait for an interrupt to change the variable to one. The naive implementation is as follows:

```
int sync = 0;
while (!sync);
```

The problem lies in optimization. The compiler analyzes the loop and decides that nothing can change the value of `sync`, and thus that the memory read can be moved out of the loop. Once the load is out of the loop, the value is obviously constant, so the test is always the same, and only need be done once. The result? A load, a single test, and an infinite loop in the form of a single branch instruction. Very fast! Unfortunately, even after the interrupt handler updates `sync`, the optimized program continues to loop.

To prevent such problems, add the qualifier `volatile` to the variable declaration (*i.e.*, `volatile int sync=0;`). This qualifier tells the compiler that the value stored in memory may change at any time, and that it may not assume that two loads of the variable return the same value.

The last issue to be considered is security. One role of the operating system is to provide isolation and protection between programs run by various users. Keeping users from crashing the machine is one implication of this role, and preventing information leaks from the operating system to programs is another. However, using a program's stack to store information during execution of interrupt handlers can be hazardous. If the stack has insufficient space, or if the stack pointer has been set to point to important data rather than a stack, blindly dumping registers onto the stack may crash the operating system. Similarly, when an interrupt handler completes, the data from the handler as well as any calls made by the handler are still on the stack, and are now visible to the program. To avoid these issues, many ISA's use a separate stack register for the operating system (*i.e.*., when operating in privileged mode, protected mode, system mode, kernel mode, *etc.*). This stack-swapping is illustrated by the LC-3 state diagram earlier in these notes; when the processor switches from unprivileged to privileged mode or vice-versa, the stacks are swapped.

Critical Sections

The last section raised the question of how to prevent interrupts from interfering with data structure operations, and in particular how to ensure that interrupts can be prevented from accessing data structures in invalid states (recall the broken linked list).

Conceptually, we want to be able to mark critical sections of the program. A **critical section** is a block of code that executes a set of operations that should be executed without stopping, *i.e.*, without interruption. As a real-world example, if I am changing the battery in my pacemaker, you should not tap me on the shoulder after I pull out the old battery and ask me to make you a pot of coffee. Or, rather, if you do so, I should ignore you until the new battery is safely in place. Pulling out the old battery and putting in the new one is a critical section. Similarly, the code in the linked list example forms a critical section, as shown below. Technically, only lines 2 and 3 need be included for the interrupt handler described in the example, but marking the entire operation as a critical section makes it compatible with interrupt handlers that change the list as well as those that only read it.

```
                /* start of critical section */
/* line 1 */    old_head = head;
/* line 2 */    head = new_elt;
/* line 3 */    new_elt->next = old_head;
                /* end of critical section */
```

Comments are not sufficient, of course; they must be replaced with operations that prevent conflicting operations from interfering and tell the compiler to avoid moving instructions out of the critical section when optimizing the code.

Once the code has been marked appropriately, the critical section occurs **atomically** with respect to the interrupt. The term atomic here implies indivisibility[1]: the entire critical section has either been executed when an interrupt occurs, or none of it has been executed. The interrupt never finds it half done.

What happens if the interrupt must find `new_elt` in the list? For example, consider another interrupt handler that removes a specified element from the list and frees the element's structure. If the element is not found, the kernel leaks dynamically allocated memory until it runs out of space (and probably crashes). If the critical section above is not guaranteed to execute before an interrupt handler tries to remove `new_elt`, the code suffers from a **race condition**. If the critical section wins the race, nothing goes wrong. If the interrupt handler wins the race, garbage piles up in the kernel. Such race conditions are bugs! They can also be quite subtle and difficult to locate, since they may only occur once in a while, and may disappear when you add a debugging print statement.

When such a race condition exists, the only solution is to rewrite the code or extend the critical section to guarantee that the interrupt handler cannot try to remove `new_elt` before the critical section inserts it into the list. In general, removing race conditions may require substantial design changes.

Let's return to the problem of creating critical section boundaries. On computers with only a single processor, the approach is simple enough: use interrupt masking at the boundaries of critical sections.[2] Also, when writing in a high-level language, tell the compiler that all of memory is volatile at these boundaries. Unlike use of the `volatile` keyword, the compiler can then optimize variable accesses inside and outside of critical sections, but cannot optimize accesses across the boundaries.

Masking interrupts does have some drawbacks, of course. One advantage of using interrupts is to reduce the average time that a device must wait for service. Delays can reduce throughput for data transfers to disks and the network, and in extreme cases can cause loss of information. For example, a serial line only buffers one character, allowing the processor about 100 microseconds to read the data before the next character overwrites it. If a program executes for long periods with interrupts masked, problems may arise. Critical sections should thus be as short as possible, and should not include operations that can be performed outside of the critical section without significant drawbacks.

A second drawback to masking arises from the fact that non-maskable interrupts are not masked. The NMI line is typically only used for unusual events such as memory or bus failure, but any data touched by the NMI handler must be protected through means other than mere masking.

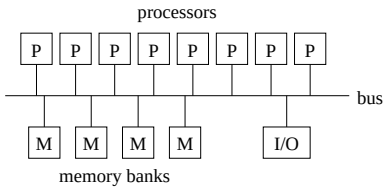
[1] Please don't ask any quarky questions about the atomicity property.

[2] We assume that programs do not share data.

Multiprocessors and Locks

On a uniprocessor, simultaneous execution of several pieces of code can occur only logically, with all but one piece of code suspended while the remaining piece executes. In contrast, multiple processors allow multiple pieces of code to execute at the same time. For example, a program and an interrupt handler can execute concurrently on a multiprocessor. As a result, interrupt masking does not suffice to protect critical sections on a multiprocessor. The interrupt mask flag operates on a per-processor basis, thus only the processor executing the code is prevented from taking an interrupt. Interprocessor communication—for example, to request that all processors mask interrupts—is far too slow and cumbersome to use at every critical section boundary.

Most commercial multiprocessors are of the SMP, or symmetric multiprocessor, variety. The symmetry lies in the relationship between processors and memory banks; while memory data can still be cached close to a processor for performance reasons, the access time from any processor to any uncached memory location is identical. An idealized diagram of an SMP is shown to the right.



Multiprocessor synchronization is a vast topic; in these notes, we focus only on the practical constructs used in Linux: spin locks, semaphores, and reader/writer locks. The discussion here is limited to lock operation semantics; a more advanced discussion is provided at the end of these notes, but in general is beyond the scope of the course.

The term **spin lock** refers to the fact that a program waiting for a lock "spins" idly in a small loop while waiting rather than going off to do other useful work or allowing other programs to use the processor. A spin lock is initially said to be available or unlocked. A program attempts to acquire a lock by atomically changing the lock from the available to the held or locked state. If successful, the program is said to hold or to own the lock. Only one program can own a lock at any time. The basic spin lock call keeps trying until it succeeds, but other lock functions allow a program to try once and to perform other work before trying again. When finished with a critical section, a program releases any locks that it owns by changing the locks' state back to available. *Only the owner of a lock should unlock it.*

The `spinlock_t` structure represents a spin lock in Linux, and a fairly rich set of functions is provided for manipulating them, as shown in the table below. The implementations of these functions depends on the kernel configuration. When configured for a uniprocessor, the macros in `linux/spinlock.h` expand to empty operations and automatic successes. When the kernel is configured for use with an SMP, the actual lock implementations defined in `asm/spinlock.h` are used (by inclusion from `linux/spinlock.h`).

Spin locks must be initialized to the available state before they are used. A statically allocated spin lock can be initialized statically, as shown in the example on the next page. Dynamically allocated spin locks require a call to `spin_lock_init` after allocation; *be sure that no race conditions allow a dynamically allocated spin lock to be used before it is initialized!*

initialization

`void spin_lock_init (spinlock_t* lock);`	Initialize a dynamically-allocated spin lock.

basic lock and unlock functions

`void spin_lock (spinlock_t* lock);`	Obtain a spin lock; call returns only when lock is obtained.
`void spin_unlock (spinlock_t* lock);`	Release a spin lock; must only be called on locks owned by caller.

miscellaneous testing functions

`int spin_is_locked (spinlock_t* lock);`	Check if a spin lock is held. Returns 1 if held, 0 if available. Note that the lock may be claimed again before the caller can do anything!
`int spin_trylock (spinlock_t* lock);`	Make one attempt to obtain a lock. Returns 1 on success, 0 on failure.
`void spin_unlock_wait (spinlock_t* lock);`	Wait until a spin lock is available. Note that the lock may be claimed again before the caller can do anything!

lock and unlock with interrupt masking

`void spin_lock_irqsave (spinlock_t* lock, unsigned long& flags);`	Save processor status in `flags`, mask interrupts, and obtain a spin lock; call returns only when lock is obtained.
`void spin_unlock_irqrestore (spinlock_t* lock, unsigned long flags);`	Release a spin lock, then set processor status to `flags`; must only be called on locks owned by caller.
`void spin_lock_irq (spinlock_t* lock);`	Mask interrupts and obtain a spin lock; call returns only when lock is obtained. Note that this version *does not preserve* the current value of the interrupt masking flag.
`void spin_unlock_irq (spinlock_t* lock);`	Release a spin lock, then enable interrupts; must only be called on locks owned by caller. Note that this version *does not restore* the previous value of the interrupt masking flag.

Normal locking and unlocking, as well as the miscellaneous lock testing functions, are mostly useful on SMPs. You should use the spin lock calls with interrupt masking—most often in the irqsave and irqrestore forms—to protect your critical sections. The `spin_lock_irqsave` prototype in the table uses C++-style reference notation for the `flags` argument to indicate that the flags are passed using only the variable name, but are changed by the function. As you know, such behavior is only possible in C if the "function" is implemented as a preprocessor macro, as is the case here. Let's rewrite our linked list example using the spin lock **application programming interface** (**API**):

```
                static spinlock_t the_lock = SPIN_LOCK_UNLOCKED;
                unsigned long flags;

                /* start of critical section */
/* line 0 */    spin_lock_irqsave (&the_lock, flags);
/* line 1 */    old_head = head;
/* line 2 */    head = new_elt;
/* line 3 */    new_elt->next = old_head;
/* line 4 */    spin_unlock_irqrestore (&the_lock, flags);
                /* end of critical section */
```

The two lock functions called to demarcate the critical section are actually C preprocessor macros defined in `linux/spinlock.h`. The interrupt masking functions called by these macros, `local_irq_save` and `local_irq_restore`, are themselves macros defined in `asm/system.h`. By unrolling those macros, we can rewrite the code as shown on the next page.

The program enters the critical section with line 0, which consists of saving the flags and acquiring a lock. The program saves the current EFLAGS register into a local variable by pushing the flags and then popping it back into some other register chosen by the compiler, after which the compiler writes the register that it chose into the variable `flags`. The "memory" argument tells `gcc` that all of memory should be considered volatile across this piece of assembly code, and that no loads or stores should be moved across it, as is necessary at a critical section boundary. After saving the old value of the flags for later restoration, the program clears IF, blocking all interrupts. Only at that point does it attempt to acquire the lock. For a detailed explanation of the syntax and properties of the `asm` directive in GCC, read the "Extended Asm" node of the info page on GCC.

The program leaves the critical section with line 4, which consists of releasing the lock and restoring the original value of EFLAGS. Interrupts may or may not have been masked before entering the critical section—for example, by this function's caller; with the functions used here, the programmer does not need to know. If the `spin_lock_irq` and `spin_unlock_irq` functions are used instead, interrupts are enabled at the end of the critical section, and no caller to the function containing the critical section can expect to keep interrupts masked across the call. Restoring EFLAGS again involves a PUSH and a POP, this time moving the value of `flags` into EFLAGS. The "cc" argument tells GCC that the condition codes register—the EFLAGS register on the x86—is modified by the assembly code.

The ordering of operations is important. If a program acquires a lock and is then interrupted by a handler that tries to acquire the same lock, the processor **deadlocks**: the handler must wait until the program releases the lock, but the program must wait until the handler finishes, so neither can make progress, and the machine freezes up.

```
                static spinlock_t the_lock = SPIN_LOCK_UNLOCKED;
                unsigned long flags;

                /* start of critical section */
    /* line 0 */ asm volatile ("      # local_irq_save macro implementation
                    pushfl           # save EFLAGS to stack
                    popl %0          # pop EFLAGS into output 0
                    cli              # mask interrupts
                " :  "=g" (flags)    /* output 0 is a general-purpose register */
                                     /*   which should then be stored in flags */
                  :                  /* no inputs                              */
                  :  "memory"        /* see text                               */
                );
                spin_lock (&the_lock);
    /* line 1 */ old_head = head;
    /* line 2 */ head = new_elt;
    /* line 3 */ new_elt->next = old_head;
    /* line 4 */ spin_unlock (&the_lock);
                asm volatile ("      # local_irq_restore macro implementation
                    pushl %0         # save input 0 to stack
                    popfl            # pop input 0 into EFLAGS
                " :                  /* no outputs                             */
                  :  "g" (flags)     /* input 0 is a general-purpose register  */
                                     /*   which should hold the value in flags */
                  :  "memory", "cc"  /* see text                               */
                );
                /* end of critical section */
```

With the ordering shown in the code, interrupts are masked before the lock is acquired and only enabled after the lock is released. The processor on which the program executes can never be interrupted while it holds the lock, thus the deadlock scenario just discussed cannot occur. However, another processor in an SMP can execute the interrupt handler, which may block waiting for the lock to be released by the program. In this case, the low-priority program running on one processor blocks execution of the high-priority interrupt handler running on another processor. This type of behavior makes it even more important that programmers make critical sections as short as possible.

As a final note, the calls to `spin_lock` and `spin_unlock` implement real locks on an SMP, and are empty macros (NOPs) on a uniprocessor. Thus, if you use the style above, your code will work on any platform.[3]

Semaphores

The term semaphore refers to visual systems of signaling, typically with flags. Railroads use semaphores to ensure that two trains do not enter a single stretch of track while heading in opposite directions. Traffic lights are sometimes called semaphores (always in some languages).

In computer systems, a **semaphore** generalizes the concept of a lock to allow some fixed number of programs to enter some set of critical sections simultaneously. For example, if we have three keyboards attached to a computer, we can use a semaphore to arbitrate access to the keyboards. The semaphore's value is initially set to three, meaning that all keyboards are free. When a program wants a keyboard, it executes a **down** operation to claim one, atomically reducing the semaphore's value to two. Down is also called wait, test, or P for the Dutch word proberen (to test). Once all three keyboards have been claimed, the semaphore's value is zero, and programs attempting to down the semaphore block until a keyboard becomes available. When a program is ready to relinquish a keyboard that it owns, it executes an **up** operation on the semaphore, which increments the semaphore's value by one. Up is also called signal, post, or V for the Dutch word verhogen (to increment).[4]

[3] SMP code tested on a uniprocessor "works" by definition, but at least you have some hope of it working on an SMP, too.

[4] Dutch is relevant because E. Dijkstra, a well-known pioneer in algorithms and computer systems, was Dutch. See `http://www.cs.utexas.edu/users/UTCS/notices/dijkstra/ewdobit.html`.

In Linux, semaphores differ from spin locks in that a program waiting on a semaphore allows other programs to execute while it waits (recall that a spin lock spins on the processor, repeatedly trying the lock). You should use semaphores rather than spin locks with any code that manages data shared between user programs executing in the kernel (through a system call). On the other hand, *semaphores should not be used in code that shares data with interrupt handlers, nor should code ever wait on a semaphore while a spin lock is held.*

Allowing other programs to run when waiting on a semaphore means that semaphores can be used to protect longer critical sections. Consider, for example, a device that accepts commands and responds only once a command has been fully executed. Each command/response pair forms a critical section, as intervening commands may be ignored or may cancel the first command. However, command execution may be slow, and maintaining control of the processor while a slow device operates is an unattractive solution. Instead, a programmer can use a semaphore to ensure that the device is not given a new command while it is busy processing, and can yield the processor to other programs even while it holds the semaphore. Allowing this logical concurrency makes synchronization of shared resources with semaphores important even in uniprocessors, and the calls do not turn into empty macros, as do the spin lock calls.

Linux supports a semaphore abstraction optimized for uncontended access. A semaphore is represented by a `struct semaphore`. The full versions of the API functions are in `arch/i386/kernel/semaphore.c`. Fast versions that handle success on the first try and fall back on the full versions when not successful are written as preprocessor macros in the header file, `asm/semaphore.h`. Linux' semaphore API appears below.

initialization

`void sema_init (struct semaphore* sem, int val);`	Initialize a dynamically allocated semaphore to a value.
`void init_MUTEX (struct semaphore* sem);`	Initialize a dynamically allocated semaphore to the value one.
`void init_MUTEX_LOCKED (struct semaphore* sem);`	Initialize a dynamically allocated semaphore to the value zero.

down and up

`void down (struct semaphore* sem);`	Wait on a semaphore; call returns only after success.
`void up (struct semaphore* sem);`	Signal a semaphore; must only be called by programs that have previously waited on the semaphore.

miscellaneous functions

`int down_interruptible (struct semaphore* sem);`	Wait on a semaphore, but allow other programs to execute while waiting; call returns 0 on success or -EINTR on interruption.
`int down_trylock (struct semaphore* sem);`	Make one attempt to wait on a semaphore. Returns 0 on success, 1 on failure (*the opposite of the spin_trylock function!*).

When only one program can enter a critical section at a time, the presence of programs in the critical section is mutually exclusive, and the term **mutex**, an abbreviation of this phrase, is used to describe synchronization of this type. The dynamic initialization routines allow a semaphore to be initialized with a specific initial value (`sema_init`) or as a mutex (a semaphore allowing one program at a time). Semaphores can also be statically allocated and initialized using the macros below, which expand into variable declarations *without a* `static` *qualifier in front of them*, and thus must appear either outside of any function or at the start of a function or compound statement (usually with `static` in front). Each dynamic initialization function has a corresponding macro for static/runtime declaration and initialization:

```
/* Allocate statically and initialize to val.  */
static __DECLARE_SEMAPHORE_GENERIC (name, val);
/* Allocate on stack and initialize to one.  */
DECLARE_MUTEX (name);
/* Allocate on stack and initialize to zero.  */
DECLARE_MUTEX_LOCKED (name);
```

The behavior of the `down` and `up` functions have already been discussed. The `down_interruptible` form is useful when control should be returned to a user-level process on receiving a signal (the user-level form of an interrupt, to be discussed later in the course; as an example, pressing CTRL-C generates a signal).

Reader/Writer Spin Locks

Certain types of synchronization benefit from separating pieces of code that share a particular datum into those pieces that write the datum and those that only read it. Readers in general do not interfere with one another, and thus any number of readers can be allowed to enter their critical sections simultaneously. However, writers may interfere with one another, and writers may also interfere with readers, thus a writer should only be allowed to access the datum when no other programs—readers or writers—are accessing it. These properties are provided by a **reader/writer lock**. Linux supports both reader/writer spin locks and reader/writer semaphores. The former are discussed in this section, and the latter in the next section.

The Linux API for reader/writer spin locks is shown below. As with normal spin locks, these locks should be used whenever data are shared with an interrupt handler, and should most often be used with interrupt masking. A reader/writer spin lock can be statically allocated and initialized as shown here:

```
rwlock_t an_rw_lock = RW_LOCK_UNLOCKED;
```

Dynamically allocated locks must be initialized with `rwlock_init`. There is no non-blocking version of the `read_lock` function, *i.e.*, `read_trylock` is not defined.

initialization

`void rwlock_init (rwlock_t* rw);`	Initialize a dynamically-allocated reader/writer lock.

basic lock and unlock functions

`void read_lock (rwlock_t* rw);`	Lock for reading; blocks until lock is obtained.
`void write_lock (rwlock_t* rw);`	Lock for writing; blocks until lock is obtained.
`void read_unlock (rwlock_t* rw);`	Release a lock previously locked for reading.
`void write_unlock (rwlock_t* rw);`	Release a lock previously locked for writing.

miscellaneous testing functions

`int write_trylock (rwlock_t* rw);`	Make one attempt to lock for writing. Returns 1 on success, 0 on failure.

lock and unlock with interrupt masking

`void read_lock_irqsave (rwlock_t* rw, unsigned long& flags);`	Save processor status in `flags`, mask interrupts, and lock for reading; blocks until lock is obtained.
`void write_lock_irqsave (rwlock_t* rw, unsigned long& flags);`	Save processor status in `flags`, mask interrupts, and lock for writing; blocks until lock is obtained.
`void read_unlock_irqrestore (rwlock_t* rw, unsigned long flags);`	Release a lock previously locked for reading, then set processor status to `flags`.
`void write_unlock_irqrestore (rwlock_t* rw, unsigned long flags);`	Release a lock previously locked for writing, then set processor status to `flags`.
`void read_lock_irq (rwlock_t* rw);`	Mask interrupts and lock for reading; blocks until lock is obtained. Note that this version *does not preserve* the current value of the interrupt masking flag.
`void write_lock_irq (rwlock_t* rw);`	Mask interrupts and lock for writing; blocks until lock is obtained. Note that this version *does not preserve* the current value of the interrupt masking flag.
`void read_unlock_irq (rwlock_t* rw);`	Release a lock previously locked for reading, then enable interrupts. Note that this version *does not restore* the previous value of the interrupt masking flag.
`void write_unlock_irq (rwlock_t* rw);`	Release a lock previously locked for writing, then enable interrupts. Note that this version *does not restore* the previous value of the interrupt masking flag.

The Linux implementation of reader/writer spin locks is fast, but does not prevent writer **starvation**. In particular, readers can enter a critical section even if a writer is waiting. As a result, readers may continuously enter and leave a critical section without the number of active readers ever reaching zero. As writers can only enter their critical sections when the number of readers (and writers) is zero, they may wait forever, a behavior known as starvation.

Reader/Writer Semaphores

The reader/writer semaphore abstraction in Linux has the exclusion properties of reader/writer locks and the scheduling properties of semaphores. Specifically, any number of readers can enter critical sections protected by a reader/writer semaphore simultaneously, but only one writer can enter a critical section at any time, and only when no other readers or writers are in critical sections protected by the same reader/writer semaphore. As with semaphores, a program attempting to acquire a reader/writer semaphore may yield the processor to another program.

Unlike semaphores, reader/writer semaphores are not parametrized by the number of programs allowed to enter critical sections. And unlike Linux reader/writer spin locks, reader/writer semaphores do not admit starvation. A waiting writer blocks any new readers from entering, for example, thus ensuring that the writer gets a turn once the current readers have finished their critical sections.

A `struct rw_semaphore` represents a reader/writer semaphore in Linux. The API for reader/writer semaphores appears below. They can be declared and initialized as shown here:

```
DECLARE_RWSEM (name);
```

or can be allocated dynamically and initialized with `init_rwsem`.

initialization

`void init_rwsem (struct rw_semaphore* sem);`	Initialize a dynamically allocated semaphore.

down and up

`void down_read (struct rw_semaphore* sem);`	Wait for reading; blocks until successful.
`void down_write (struct rw_semaphore* sem);`	Wait for writing; blocks until successful.
`void up_read (struct rw_semaphore* sem);`	Signal a semaphore previously waited on for reading.
`void up_write (struct rw_semaphore* sem);`	Signal a semaphore previously waited on for writing.

Selecting a Synchronization Mechanism

As mentioned earlier in these notes, synchronization is a vast topic, and choosing between the various forms of synchronization available in the Linux kernel based on their properties may seem quite difficult at first. This problem is exacerbated by the fact that the synchronization constructs in Linux have properties not traditionally implied by the names chosen for them. In this section, we present two approaches for selecting synchronization mechanisms.

Both approaches simplify the set of possibilities by using semaphores only as mutexes (allow only one program at a time in critical sections). The first approach further simplifies the decision process by always using the interrupt save and restore form of spin locks. With these restrictions, one can view the four synchronization constructs as a combination of two choices between two sets of properties, as shown in the table below.

The choice for the vertical axis depends on whether any interrupt handler must enter a critical section protected by the mechanism to be chosen. Kernel code is otherwise only executed by programs making system calls, or sometimes by programs running in the kernel, which can be handled similarly. If an operation on a data structure must be atomic with respect to other operations on the data structure, but the data are never accessed by an interrupt handler, semaphores should be used, as shown in the bottom row of the table. The choice for the horizontal axis is between mutual exclusion and reader/writer properties.

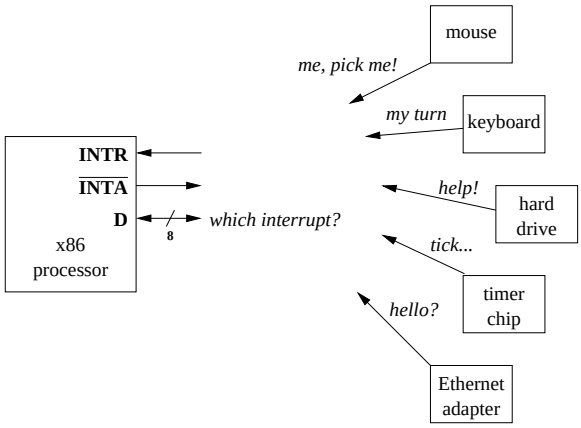
	mutual exclusion	reader/writer	
data shared by interrupt handlers	`spin_lock_irqsave` `spin_unlock_irqrestore`	`read_lock_irqsave` `read_unlock_irqrestore`	`write_lock_irqsave` `write_unlock_irqrestore`
data shared only by system calls	`down` `up`	`down_read` `up_read`	`down_write` `up_write`

Alternatively, one can use the more complex decision tree shown below to obtain slightly faster synchronization. Only the mutual exclusion option is shown here. For reader/writer properties, use reader/writer semaphores in place of semaphores, and use reader/writer spin locks in place of spin locks with the same interrupt masking properties. Finally, when more than one lock must be acquired, only the first lock needs to mask interrupts.

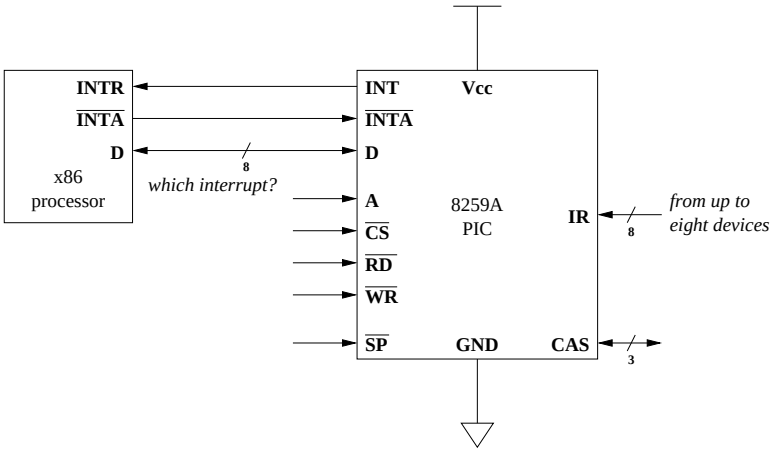
type of code entering critical section	critical section shares data with	for mutual exclusion, use
system calls only	other system calls	`up` `down`
	interrupt handlers	`spin_lock_irq` `spin_unlock_irq`
both system calls and interrupt handlers	both system calls and interrupt handlers	`spin_lock_irqsave` `spin_unlock_irqrestore`
interrupt handlers only	system calls	`spin_lock` `spin_unlock`
	higher priority interrupt handlers	`spin_lock_irqsave` `spin_unlock_irqrestore`

Interrupt Control

At this point, you should feel fairly comfortable with the interrupt support provided by a typical processor. However, connecting a large number of devices to a processor is not completely straightforward. A diagram of an x86 processor with five devices appears to the right. Each device generates a single output signal requesting attention, while the processor has a single interrupt input pin, INTR, an interrupt acknowledgement output, $\overline{\text{INTA}}$ (the bar means that the signal is active low rather than high), and a data bus over which the interrupt vector must be delivered. Device register inputs and outputs are not shown.



One might imagine simply joining the device interrupt lines with an OR gate and feeding it into the processor's INTR input. However, something must be done to choose amongst the devices when more than one needs attention. Clearly, to avoid shorts, only one can write an interrupt vector to the bus. The interrupt acknowledge signal ($\overline{\text{INTA}}$) can be used to pick, but in that case can only be delivered to one device at a time, and thus doesn't resolve the problem. Furthermore, some priority scheme for the devices can be useful, allowing a high priority interrupt to preempt (interrupt) the execution of a low priority interrupt handler.



The solution to this problem requires an **interrupt controller**, an additional piece of hardware to manage the interrupt signals and priorities. The x86 has traditionally used Intel's 8259A **Programmable Interrupt Controller**, or **PIC**, chip, for this purpose. The diagram to the right shows all 28 pins, which we describe piece by piece in the following text. First, however, we review the implementation of the LC-3's interrupt controller.

Although Patt and Patel describes a part of the interrupt controller, and the microarchitecture requires a three-bit priority signal rather than a single interrupt line, most of the controller implementation is external to the LC-3. Using an external implementation allows the processor to work with different prioritization schemes, different numbers of devices, *etc.*, and only the most recent x86 implementations have integrated the interrupt controller onto the processor. The simplest external implementation is to use a priority encoder to select from up to eight devices with fixed prioritization. The output of the priority encoder is fed into the LC-3's interrupt priority inputs, and can also be used to select the interrupt vector number or to gate the devices' writing a vector.

The 8259A is both more complicated and more flexible than the basic design just described. It operates asynchronously, with data bus transactions driven by control signals from the processor. The most common operation is reporting an interrupt to the processor, and is handled as follows. The PIC uses internal state to track which of the eight interrupts are currently being serviced by the processor. When one of the IR inputs goes high, the PIC decides whether or not it should report the new interrupt immediately based on the prioritization scheme and the set of interrupts currently in service. Lower-numbered IR lines have higher priority, and an interrupt of priority equal or lower to any interrupt already in service is masked. To report an interrupt, the PIC raises the INT output and waits for the processor. The processor (or some glue logic) strobes the $\overline{\text{INTA}}$ input to request that the PIC write the interrupt vector to the data bus D. Once the vector has been written, the PIC marks the reported interrupt as being in service and returns to waiting. At some point, the interrupt handler tells the PIC that the interrupt has been serviced by writing certain bits to the address A and data D inputs with an OUT instruction. On receiving this **end-of-interrupt** (**EOI**) signal, the PIC removes the interrupt from its set of in-service interrupts. *If an interrupt handler fails to send an EOI, the PIC continues to mask all interrupts of equal and lower priority indefinitely!* If the interrupt handler has not interacted with the device that raised the interrupt before sending the EOI, the interrupt is likely to be generated again immediately, thus it is important to perform these actions in the correct order. As we discuss later, the proper ordering is embedded into the software infrastructure in the Linux kernel, making it substantially harder for a programmer to screw up.

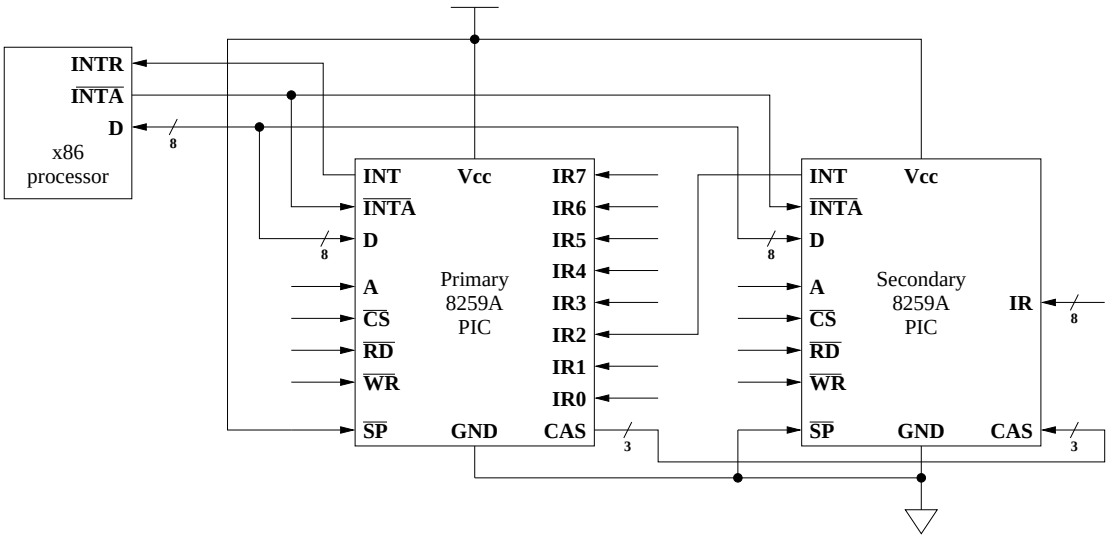
The simple overview just provided leaves a number of inputs and outputs in the diagram unused. The next block to be considered are the address A, chip select $\overline{\text{CS}}$, read $\overline{\text{RD}}$, and write $\overline{\text{WR}}$ signals. The read and write signals are named from the processor's point of view and tell the PIC that the CPU has written data to the data bus D or expects to read data from D. However, the PIC is not the only device on the data bus! An 8259A is mapped into two adjacent locations in the processor's port space, and must only respond when the processor communicates with one of those two ports. The chip select input must be active (low) only when the processor specifies one of these two ports on its address bus. If you assume that the processor uses a 16-bit address bus and wants to map the 8259A at ports 0x20 and 0x21, you should be able to generate the PIC's $\overline{\text{CS}}$ and A inputs using a handful of gates.

The 8259A uses two ports because the full functionality extends well beyond that described so far, and, while we shall not make use of much of it in the class, you should recognize that it is a programmable chip. The options supported by the 8259A include support for more than one ISA (the other ISA uses a slightly different handshaking protocol on the bus), options for various priority schemes such as rotating priorities for fairness, masking certain interrupts out indefinitely (used by Linux and most operating systems to prevent spurious interrupts), and setting the five high bits of the interrupt vector number; the three low bits are used to identify which IR input is being reported. You will probably be asked to implement a simpler, synchronous version as part of a homework. If you are interested in the asynchronous implementation, you should take ECE462.

One problem with the 8259A as presented so far is that eight interrupt lines may not be enough. So what should be done? One can certainly hook two, three, or four 8259A's to a processor using different port addresses, but doing so only reintroduces the original problem of how to merge the INT signals from the PICs into a single INTR input for the processor. One could design a new PIC with 16 inputs, or 64 inputs, or 256 inputs, but that may be a bit wasteful if many applications use only a few interrupt lines. The solution? Allow multiple 8259As to operate as a hierarchy (a cascade) and let the system designer pick.[5]

Most x86-based computers use two 8259A's connected as shown in the diagram on the next page, supporting seven IR lines on the primary PIC and eight more on the secondary PIC. Interrupts generated by devices hooked to the secondary PIC are delivered to IR_2 on the primary PIC, and only delivered to the processor when no other higher-priority interrupts (IR_0 and IR_1 on the primary PIC) are in service.

[5] Sorry.



If you think carefully about the protocol for interactions between the processor and the PIC, you should realize that the functionality discussed so far is not sufficient to support the cascaded design shown above. In particular, which PIC should write an interrupt vector to the data bus when the processor strobes $\overline{\text{INTA}}$? The secondary PIC may have raised an interrupt, but it doesn't know whether the primary is reporting its interrupt or one with higher priority, so it cannot decide which chip should provide the vector. Similarly, while the primary knows which chip should provide the vector, it knows neither the high bits of vector numbers used by the secondary PIC nor which of the secondary PIC's interrupt lines is being reported, and thus cannot report on behalf of the secondary PIC. Instead, it must tell the secondary PIC to write a number to the data bus. The CAS bus supports this communication: effectively, the primary transmits the identification number for one of up to eight possible secondary PICs on the bus (8259A's can be cascaded to allow up to 64 interrupt inputs). The secondary PIC program input $\overline{\text{SP}}$ is used to differentiate the primary from the secondary PICs and to configure the CAS bus as either input or output.

The PIC in Hardware and Software

Now that you are familiar with the basic purpose of the 8259A and some of its operations, we are ready to discuss the configuration used with most x86-based processors and the software and protocols used to drive the PICs in more detail. This section also discusses the abstractions and functions used to represent interrupt controllers within the Linux kernel.

Two 8259A PICs are used in cascade, as shown in the figure in the previous section. The primary PIC is mapped at ports 0x20 and 0x21, and the secondary PIC at ports 0xA0 and 0xA1. The INT output on the secondary PIC feeds into IR_2 on the primary PIC. The file `arch/i386/kernel/i8259.c` holds the Linux source code for configuring and interacting with the 8259A. The first function of interest is the initialization routine:

```
void init_8259A (int auto_eoi);
```

The single argument allows for use of the 8259A's optional automatic generation of EOI signals (as opposed to requiring the interrupt handler to send them to the PIC), but is not used in Linux.

Only one processor should initialize the PICs, and it should not be interrupted while changing the PICs' internal state. The entire initialization function is thus one critical section. After masking interrupts on the processor and acquiring a lock, the code masks all interrupts (on both PICs), executes the initialization sequence, restores the mask settings, releases the lock, and restores the IF flag. All interrupts are initially masked during boot, but in theory one could reinitialize the PICs by calling this function, in which case any active interrupts must be allowed.

Masking and unmasking of interrupts on an 8259A outside of the interrupt sequence requires only a single write to the second port (0x21 or 0xA1), and the byte written to this port specifies which interrupts should be masked.

The initialization sequence requires that four initialization control words (ICWs) be sent to the 8259A. You may want

to look through the 8259A specification to understand the detailed meaning of the control words used in Linux.[6] The first word, ICW1, is delivered to the first PIC port—either 0x20 or 0xA0—and tells the PIC that it is being initialized, that it should use edge-triggered input signals, that it is operating in cascade mode (*i.e.*, using more than one 8259A), and that four control words will be sent in all. The remaining ICWs are written to the second port. The high bits of the interrupt vector numbers are provided in ICW2: the primary 8259A is mapped to interrupt vectors 0x20 through 0x27, and the secondary 8259A is mapped to interrupt vectors 0x28 through 0x2F. The specific IR pin used in the primary/secondary relationship is specified by ICW3. Finally, ICW4 specifies the 8086 protocol, normal EOI signalling, and a couple of other (unused) options that we have ignored here.

Linux abstracts interrupt controllers using a **jump table** structure, thus allowing other code to perform generic operations on the relevant interrupt controller without knowing the details of the controller itself. Although the machines that you use in this class have only the two 8259As described here, many modern processors have embedded "advanced" PICs, or APICs, and most SMPs require I/O APICs for synchronization of interrupt delivery to a single processor and to support interprocessor communication. The protocol used to control an 8259A has essentially nothing in common with the protocols used for these APICs other than at the level of abstract operations listed in the jump table. The jump table, called a `hw_irq_controller`, is defined in `linux/irq.h` as shown below:

```
struct hw_interrupt_type {
    const char* typename;
    unsigned int (*startup) (unsigned int irq);
    void (*shutdown) (unsigned int irq);
    void (*enable) (unsigned int irq);
    void (*disable) (unsigned int irq);
    void (*ack) (unsigned int irq);
    void (*end) (unsigned int irq);
    void (*set_affinity) (unsigned int irq, unsigned long mask);
};
typedef struct hw_interrupt_type hw_irq_controller;
```

The jump table for the 8259, `i8259A_irq_type`, is defined in the `i8259.c` file mentioned earlier, and is filled with the names of the appropriate functions, which are also defined in the file. The `typename` field is a human-readable name used when reading the contents of the `/proc/interrupts` file, and is set to "XT-PIC" for the 8259A.

The other fields in the jump table are function pointers to controller-specific code. The `startup` function is called when the first request is made to attach an interrupt handler to a particular interrupt (more than one can be associated with a given interrupt, as discussed later in these notes). Both PICs are initially configured to mask all interrupts; the `startup` function for the 8259A tells the appropriate PIC to allow the interrupt line to generate interrupts, *i.e.*, it unmasks the interrupt on the PIC.[7] The `shutdown` function is called when the last handler is removed from an interrupt, and, in the case of the 8259A's function, tells the appropriate PIC to mask the interrupt.

The `disable` and `enable` functions allow nested disabling and re-enabling of active interrupts. The generic code only calls these controller-specific functions on the first call to disable and the last call to enable. The 8259A functions are identical to those used for `startup` and `shutdown`: they unmask and mask the specified interrupt on the appropriate PIC, respectively.

The `ack` and `end` functions are used to wrap the interrupt handler associated with a given interrupt. When an interrupt occurs, the controller-specific `ack` function is called to acknowledge receipt of the interrupt. The interrupt handler is then executed. Finally, the controller-specific `end` function is called to end the interrupt. The `ack` function for the 8259A masks the interrupt on the PIC, then sends the EOI signal. Although this ordering differs from the one described earlier, the mask bit on the PIC prevents further interrupts from occurring even though the device has yet to be serviced. *However, if your handler disables and re-enables the interrupt before servicing the device, a new interrupt will be generated, and will make your interrupts slower.* The generic interrupt handling code in Linux keeps such interactions from creating infinite loops by squashing the second interrupt and only leaving a "pending" marker behind, which causes the first interrupt to re-execute the handler after it finishes the first time. The 8259A's `end` function unmasks the interrupt on the appropriate PIC.

[6] Please be aware, should you choose to do so, that you will encounter terminology that you may find offensive.

[7] The return value from `startup` is always 0 for the 8259A and is only used by the auto-probe code, which is outside of the scope of this course; see the section on advanced topics at the end of these notes for pointers.

Finally, the `set_affinity` function is used to specify which CPUs within an SMP are allowed to execute a given interrupt. The most common use of this function is to restrict an interrupt to execute only on one processor, which simplifies data sharing at the expense of performance.

Common Interrupt Abstractions

We are almost ready to explore the infrastructure provided by Linux to abstract and manage interrupts within the kernel. Before doing so, however, we discuss two common abstractions used in many systems to extend the utility of interrupts as supported by most processors: interrupt chaining, and soft(ware) interrupts.

Interrupt chaining: We have thus far assumed that each interrupt vector is associated with a single handler, and such is certainly the case as far as the hardware is concerned: an x86 transfers control to a handler specified by a single entry in the IDT, and control returns when a IRET instruction is executed. However, several software systems may wish to act when an interrupt occurs, and providing the handler is the only method through which they can take action. Early DOS systems supported terminate-and-stay-resident (TSR) programs, which often installed interrupt handlers to respond to keystrokes or mouse motion (the handlers stayed resident), then returned control to DOS (the program terminated). These programs **chained** their interrupt handlers to those already present in the interrupt vector table, usually by rewriting a JMP instruction at the end of their handler to jump to the old handler. As a result, the resulting chain (linked list) of handlers was somewhat brittle: none of the programs had any way to walk over the chain, so removing a single program's handler cleanly was effectively impossible.

Similarly, although only a single interrupt is generated when an interrupt controller's input line goes high, it is possible to connect more than one device to that input, in which case any of the devices so connected may have been the source of the interrupt. Hooking multiple devices to an interrupt line typically also requires that the software allow chaining of interrupt handlers, and furthermore that the the devices associated with the chain can be queried for their interrupt status. Clearly, only the device that generated the interrupt should receive service; others should be ignored. When an interrupt occurs, control is passed to the handler for the first device, which accesses device registers to determine whether or not that device generated an interrupt. If it did, the appropriate service is provided. If not, or after the service is complete, control is passed to the next handler in the chain, which handles interrupts from the second device, and so forth until the last handler in the chain completes. At this point, registers and processor state are restored and control is returned to the point at which the interrupt occurred. Such device interrogation is often slow, thus chaining of this type occurs fairly rarely in practice.

Soft interrupts: The purpose and importance of interrupts generated by hardware is probably clear to you. They allow relatively slow devices to get attention from a processor in a timely manner without requiring the processor to poll the devices periodically. An interrupt handler takes control of the processor as soon as it needs to run, thus interrupts can be thought of as having higher priority than most programs (except for critical sections).

These two features are also attractive for systems based purely on software. For example, I may have one program that controls a database, and a second that manages a network connection with a remote user. A single database query may require several packets of data to be sent over the network, thus not every hardware interrupt generated by the network card corresponds to a new command from the remote user. However, once a complete command has been received from the user, as recognized by the second program, allowing this program to interrupt the database control program is a useful abstraction. Unfortunately, the network program is not a device, and has no access to the INTR input on the processor.

Similarly, hardware interrupts generally require some small amount of work to service a device, but may require much more work to handle any data delivered by the device or modified as a result of the interrupt. Network packets arrive at a machine, for example, and must be examined to identify the program to which they are being sent. Making this decision and giving the data to the program takes time, much more time than should be spent in a hardware interrupt handler, especially since the network card neither needs any more service nor can help with the remaining work.

Most operating systems support the notion of software-generated, or **soft interrupts**, to address these needs. Soft interrupts generally operate at a priority level between hard interrupts and programs. They can interrupt a program, but can in turn be interrupted by an interrupt generated by a device. Also, many hard interrupts have associated soft interrupts to handle any work that does not require interaction with the device. The hard interrupt handler in this case generates the soft interrupt, which typically takes control of the processor after the hard interrupt has finished its work, but defers to other hard interrupts handlers.

Basics of Linux Interrupts

This section discusses the basics of setting up, executing, and removing interrupt handlers within Linux. Many of Linux' interrupt management routines are defined in `arch/i386/kernel/irq.c` and declared in `linux/interrupt.h`. For now, we limit the discussion to hard interrupts; in the next section, we discuss the tasklet abstraction provided to support soft interrupts in Linux.

Installing an interrupt handler: In order to install an interrupt handler in Linux, a call is made—usually by a device driver—to the function `request_irq` shown here:

```
int request_irq (unsigned int irq,
                 void (*handler) (int, void*, struct pt_regs*),
                 unsigned long irqflags,
                 const char* devname,
                 void* dev_id)
```

The function takes five arguments. The `irq` value specifies the interrupt vector. A pointer to the interrupt handler is passed in `handler`. The `irqflags` argument provides a bit vector specifying options (discussed below). Human-readable interfaces, such as the file `/proc/interrupts`, make use of the `devname` field. Finally, the `dev_id` is an arbitrary pointer that is returned to the interrupt handler when the interrupt occurs. Usually, it is a pointer to a structure holding information about the device's status within the kernel; the contents of this block are defined by the device driver, and the kernel views the block as opaque, as is the case here. The function returns 0 on success, and a negative error code on failure. The prototype for `request_irq` resides in `linux/sched.h`, and the implementation in `irq.c`.

Two important flags are defined for the `request_irq` call: SA_SHIRQ and SA_INTERRUPT. The shared IRQ (SA_SHIRQ) flag allows the interrupt vector to be shared by other handlers, but only if *all* of the handlers agree to share. If disagreement occurs, the later request to add a handler is denied through the function's return value. The SA_INTERRUPT flag keeps all interrupts masked on the processor throughout the handler's execution, and should only be used for short handlers. The PIC already masks all interrupts of equal and lower priority, so using SA_INTERRUPT can invert the normal priority.[8]

Interrupt handlers are typically written in C and use standard C linkage, *i.e.*, the usual calling convention. The three arguments to the handler are the interrupt vector (in case one handler is used for multiple vectors), the `dev_id` pointer provided in the call to `request_irq`, and a pointer to a structure holding the values of the registers at the time that the interrupt occurred. The structure actually resides on the stack, and is simply the memory locations to which the registers were pushed at the start of the interrupt.

The Linux kernel keeps track of information pertaining to interrupt vectors in an array, `irq_desc`, of descriptors (of type `irq_desc_t`, defined in `linux/irq.h`). Each descriptor holds a bit vector tracking the status of the interrupt, a pointer to a jump table for the associated interrupt controller, a linked list of interrupt handlers for the vector, a count of calls to disable the vector (these are nested), and a spin lock to manage access to the descriptor.

When a device driver calls the `request_irq` function to install a handler, a new structure to represent the handler is allocated and filled in. The kernel uses the `irqaction` structure, defined in `linux/interrupt.h`, to represent the handler. This structure holds the information provided to `request_irq`, including the handler pointer, the flags, the device name, and the `dev_id` pointer.

When no handlers are currently associated with the requested interrupt vector, the new `irqaction` structure becomes the action list (of one element) for that vector in the descriptor array, and the interrupt controller startup function is called. When other handlers are already present, the new handler and flags are first checked for compatibility with existing handlers. For example, do all agree to share the vector? If the new handler is incompatible, the call fails. Otherwise, the new `irqaction` is linked at the end of the handler list in the requested interrupt vector's descriptor. In effect, the interrupt handler is chained to the end of the existing chain of handlers for that vector.

Linux makes many kernel internals available to the superuser—*i.e.*, the `root` login—through the `/proc` directory. Reading and writing files in this directory translates to executing associated functions within the kernel. As an example, after installing a new handler, `request_irq` creates a new directory in `/proc/irq`; on an SMP, this directory holds

[8] While not mixing these two flags is not explicitly forbidden, only the *first* handler's SA_INTERRUPT flag is checked when deciding whether to unmask interrupts or not before calling the handlers in a list.

a human-readable hexadecimal value representing the bit vector of processors allowed to execute the interrupt (recall the `smp_affinity` function in the controller jump table). The superuser can write a new hex value into this file to change the allowed set of processors for an interrupt. Most operating systems do not expose their internal data in this manner, but it can be convenient at times.

Uninstalling an interrupt handler: Removing an interrupt handler in Linux uses the function `free_irq` shown below, which declared and defined in the same files as `request_irq`.

```
void free_irq (unsigned int irq, void* dev_id);
```

The function takes two arguments: the interrupt vector, and the pointer to the device information structure passed earlier to `request_irq` when installing the handler to be removed.

The function checks the linked list of actions associated with the specified interrupt vector for one with a matching `dev_id` pointer, and, if one is found, removes it from the linked list. When a vector has several handlers chained together, the use of the `irqaction` structures allows Linux to remove the specified handler cleanly, thereby extracting a single link from the chain. Recall that the use of data embedded in the interrupt handlers themselves in DOS made this operation effectively impossible.

If the action being removed is the only one in the vector's list, the interrupt controller's shutdown function is called on the interrupt vector. As before, both the linked list of actions and the associated interrupt controller's jump table are obtained from the interrupt vector's descriptor in the array `irq_desc`.

Interrupt invocation: The main interrupt handling function in Linux is the C function `do_IRQ` in `irq.c`. However, the interface used by the hardware when generating an interrupt is essentially unrelated to the C calling convention. Interrupt invocation for 8259A interrupts thus requires some assembly code to wrap the C function. Such code is called **linkage** because it links the interface used by the hardware to the calling convention used when compiling `do_IRQ`.

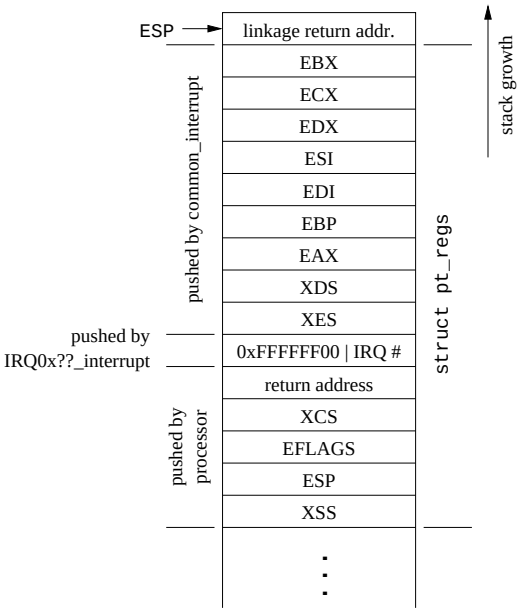
When starting an interrupt, the x86 switches the stack if necessary and records the (unprivileged) stack pointer the flags, and the return address. The first bit of linkage (*e.g.*, `IRQ0x06_interrupt`) pushes the interrupt number onto the stack (*e.g.*, 6) and jumps to a second bit of linkage called `common_interrupt`. This bit of handoff code is generated in `i8259.c` for each 8259A interrupt through use of the `BUILD_IRQ` macro in `asm/hw_irq.h`. The IDT entry is then set up by the `init_IRQ` function by calling `set_intr_gate` with a pointer to the handoff code.

The `common_interrupt` linkage, as defined in `hw_irq.h`, saves all registers not already saved by the processor, calls the `do_IRQ` function, and jumps to `ret_from_intr`, the last bit of linkage, from `arch/i386/kernel/entry.S`. This final part can involve some scheduler activity to give control of the processor to a new program from time to time, but eventually restores all of the registers and executes an IRET instruction (possibly returning control to a different program, but we leave that issue for later in the course).

Interrupt execution: The `do_IRQ` function in `irq.c` serves as the interrupt handler for all 8259A interrupts:

```
unsigned int do_IRQ (struct pt_regs regs);
```

Notice that the argument is a structure, requiring that the whole structure be copied onto the stack before calling the function. In the case of `do_IRQ`, this "structure" is simply the copy of the registers that had to be pushed onto the stack to save them anyway, and no extra work is required to create it. The top of the stack on entry to `do_IRQ` appears in the diagram to the right. The `struct pt_regs` is defined in `asm/ptrace.h`, and the `SAVE_ALL` and `RESTORE_ALL` macros in `entry.S` push and pop the registers from the stack, respectively. The segment registers—XDS, XES, XCS, and XSS—were deliberately ignored in our discussion of x86 assembly code. They are used to partition a machine's physical address space amongst several programs, and will be discussed in more detail later in the course. For now, consider XCS (extended code segment) to be part of the return address, and XSS (extended stack segment) to be part of the stack pointer. The XDS and XES registers are effectively constants.



You might wonder why the linkage is used to record the interrupt vector and to pass control to a general function rather than saving some overhead by instantiating separate copies the code for each interrupt vector. Two factors contribute to this choice: first, the function is not entirely trivial, and making a large number of copies can bloat the kernel; second, while we are considering only 16 interrupts, most servers support up to 224 interrupts through the I/O APIC, making the kernel bloat more substantial.

Put briefly, the `do_IRQ` function uses the interrupt vector passed by the assembly linkage discussed in the last section to find the correct element of the interrupt descriptor array `irq_desc`, interacts with the corresponding interrupt controller using the jump table pointer in the interrupt's descriptor, and calls any handlers installed by `request_irq`.

The detailed behavior of `do_IRQ` depends on the status flags in the interrupt descriptor. As mentioned earlier, each interrupt descriptor contains a bit vector of status flags, which are defined in `linux/irq.h`. The four flags relevant to our discussion are IRQ_PENDING, which indicates that an interrupt has occurred; IRQ_INPROGRESS, which indicates that handlers are currently being executed; IRQ_DISABLED, which indicates that the interrupt vector has been disabled temporarily; and IRQ_REPLAY, which indicates that the source of an interrupt is software rather than hardware, and in particular that the interrupt is being replayed because it was disabled by software when the hardware generated it, as described below.

The first step taken by `do_IRQ` is to acknowledge the interrupt to the interrupt controller. For the 8259A, this call masks the specific interrupt on the PIC and sends an EOI for that interrupt. If this interrupt is disabled and then re-enabled by an interrupt handler, the PIC may raise the interrupt again before `do_IRQ` finishes. After the acknowledgement, the IRQ_REPLAY flag is cleared, as the source of the interrupt is no longer relevant.

Interrupts in Linux may occur while disabled by software or while the same interrupt is already being serviced. The IRQ_DISABLED status flag in the descriptor indicates that the interrupt should not be executed immediately.[9] Instead, the IRQ_PENDING flag is set, and the interrupt is executed (synchronously) when the call is made to re-enable that interrupt. When calling `do_IRQ` from software for this purpose, the IRQ_REPLAY flag is set. An interrupt that occurs while the same interrupt is already being serviced simply leaves the IRQ_PENDING status flag set in the interrupt descriptor; this flag is used later to repeat the handlers for the second (third, *etc.*) interrupt. Both cases skip the next portion of `do_IRQ`, in which the handlers provided by calls to `request_irq` are executed.

Handler execution begins by changing the status from IRQ_PENDING to IRQ_INPROGRESS. A call to `handle_IRQ_event` executes the chain of handlers, and is repeated whenever another interrupt has occurred during execution, as indicated by the presence of the IRQ_PENDING flag. The `handle_IRQ_event` subroutine records the fact that the processor is executing a hardware interrupt by calling `irq_enter`, sets the IF flag to enable interrupts on the processor unless the first handler is marked with the SA_INTERRUPT flag, executes all handlers in the chain for the interrupt, disables interrupts again, and records the fact that the processor is no longer executing a hardware interrupt by calling `irq_exit`. *Note that the function* `do_IRQ` *executes in its entirety with the IF flag clear,* i.e.*, with all interrupts masked. Only when executing a handler without the SA_INTERRUPT flag set are interrupts allowed, and the STI and CLI instructions appear in* `handler_IRQ_event`. Once the handlers have been executed without an intervening interrupt of the same type, the IN_PROGRESS flag is removed, and `do_IRQ` proceeds to the cleanup phase.

The last step for handling the hardware interrupt in `do_IRQ` is to tell the interrupt controller that the interrupt has ended. For the 8259A, this call normally unmasks the interrupt on the PIC. However, in the case of an interrupt taken while disabled in software or while the same interrupt is already in progress, it does nothing.

Finally, `do_IRQ` checks for the need to execute soft interrupt handlers by calling `do_softirq`. The return value from `do_IRQ` is always 1.

Additional Interrupt Abstractions

The main soft interrupt support in Linux takes the form of tasklets. A **tasklet** is a data structure used to wrap a single handler function used as a soft interrupt handler. A soft interrupt corresponding to a tasklet is generated when some piece of code—usually a hard interrupt handler—requests that the handler function associated with the tasklet be scheduled for execution. When scheduled, a tasklet is linked into a list to be executed along with any other tasklets

[9] Taking an 8259A interrupt with IRQ_DISABLED set for that interrupt can only occur on an SMP, but can be common with interrupt controllers that do not support masking of specific interrupts on the controller.

of the same priority level. A separate list is maintained for each processor, thus tasklets in an SMP execute on the processor on which they were scheduled. Tasklets typically execute whenever a hard interrupt handler has completed, but may be deferred (due to software having disabled the tasklet, for example), in which case another program running in the Linux kernel periodically tries to execute them.

The tasklet data structure, `struct tasklet_struct`, is defined in `linux/interrupt.h` along with declarations for the tasklet API. The structure has five fields: a pointer to the associated handler function, a 32-bit value to be given to the handler when called, a count of times that the tasklet has been disabled (as with hard interrupts, calls to disable and to re-enable a tasklet can be nested), a bit vector of status flags, and a pointer to form a linked list.

The tasklet API appears in the table below. Dynamically allocated tasklets can be initialized using the `tasklet_init` call. Statically allocated and initialized tasklets can be declared using either of the forms here:

```
DECLARE_TASKLET (name, func, data);
DECLARE_TASKLET_DISABLED (name, func, data);
```

As with the other DECLARE_ macros discussed in these notes, the qualifier `static` should prefix the declaration of any tasklet declared within a function to ensure that the tasklet is not allocated on the stack. In the second form shown above, the tasklet's disable count is initialized to one rather than zero, and an unpaired call to `tasklet_enable` or `tasklet_hi_enable` should be used to enable the tasklet initially. Otherwise, calls to disable and to re-enable a tasklet, like those for hard interrupts, are intended to be nested and to appear in pairs in the code.

initialization

`void tasklet_init (struct tasklet_struct* t, void (*func) (unsigned long), unsigned long data);`	Initialize a dynamically allocated tasklet.

scheduling

`void tasklet_schedule (struct tasklet_struct* t);`	Schedule a tasklet for execution with low priority.
`void tasklet_hi_schedule (struct tasklet_struct* t);`	Schedule a tasklet for execution with high priority.

disabling and re-enabling

`void tasklet_disable_nosync (struct tasklet_struct* t);`	Disable a tasklet, *i.e.*, mask the soft interrupt associated with the tasklet. Pair with calls to enable.
`void tasklet_disable (struct tasklet_struct* t);`	Disable a tasklet and, if it is currently executing on some processor, wait for it to finish. *Can deadlock if called from within the tasklet's handler on an SMP.* Pair with calls to enable.
`void tasklet_enable (struct tasklet_struct* t);`	Re-enable a tasklet, *i.e.*, unmask the soft interrupt associated with the tasklet. Pair with calls to disable.
`void tasklet_hi_enable (struct tasklet_struct* t);`	Re-enable a high-priority tasklet, *i.e.*, unmask the soft interrupt associated with the tasklet. Pair with calls to disable.

Four soft interrupt types are defined in Linux, two of which serve as tasklet priority levels. The handler pointers for each type are kept in the `softirq_vec` array. The array also has a second field called `data` for each soft interrupt, which is apparently unused. The soft interrupt types are:

- HI_SOFTIRQ — high-priority tasklets (function `tasklet_hi_action`)
- NET_TX_SOFTIRQ — network transmission
- NET_RX_SOFTIRQ — network reception
- TASKLET_SOFTIRQ — low-priority tasklets (function `tasklet_action`)

Tasklet priority has little significance, but does affect the order in which tasklets are executed. As mentioned in the previous section, soft interrupts are executed by the `do_softirq` function in `kernel/softirq.c`. Most of the tasklet support code is also in this file.

The do_softirq function first checks whether or not an interrupt of any type, either hard or soft, is already being executed by the processor. If so, the function terminates.

The next few steps are performed with interrupts masked on the processor. After masking interrupts, do_softirq checks for pending soft interrupt types on the processor, and terminates if none are pending (after restoring the previous IF value). Otherwise, it records any pending soft interrupt types and resets the bit vector of pending types. Finally, do_softirq records the fact that the processor is executing a soft interrupt using the __local_bh_disable macro from asm/softirq.h. The IF flag is then set, re-enabling hard interrupts.

With hard interrupts enabled, do_softirq then executes the handler function for each soft interrupt type that was either pending at the start of the function or was raised during the execution of other handlers. The handler for each type is executed at most once. The two tasklet handlers, which we describe in more detail later in the notes, walk a per-CPU linked list of scheduled tasklets and execute each tasklet in the list.

Hard interrupts are then disabled again for the final few steps. The __local_bh_enable macro is then used to record the fact that the processor is no longer executing a soft interrupt. If a soft interrupt type that was executed was raised (scheduled) again during the function's execution, do_softirq wakes a kernel daemon associated with the processor. These daemons are created by spawn_ksoftirqd, which starts a kernel thread for each processor to serve as the soft interrupt daemon, using ksoftirqd as the main function for each thread. The threads check for pending soft interrupts, execute them if necessary by calling do_softirq, and put themselves to sleep. When a soft interrupt must be postponed on some processor, that processor's daemon is reawakened and, once it is given a turn on the processor, the daemon tries to execute the scheduled soft interrupts. After restoring the original IF flag, the do_softirq function terminates.

Tasklet scheduling and execution makes use of the status bit vector in the tasklet structure. Two flags are defined: TASKLET_STATE_SCHED, and TASKLET_STATE_RUN. The TASKLET_STATE_SCHED flag indicate is used both to indicate that a tasklet has been scheduled but has yet to execute, and as a lock bit to ensure that a tasklet is only scheduled on one CPU at a time. On an SMP, a tasklet may be scheduled again while it executes. In this case, the TASKLET_STATE_RUN bit is used (as a lock bit) to serialize execution of the tasklet on the processors on which it is scheduled.

When a tasklet is changed from unscheduled to scheduled, it is linked into either the tasklet_hi_vec or tasklet_vec list for the scheduling processor, depending on the priority requested. A tasklet executes at least once after being scheduled, and usually executes exactly once.

The two tasklet execution functions, tasklet_hi_action and tasklet_action, are almost exactly the same, and merely operate on different linked lists (the two just mentioned). In each case, the current list is extracted with interrupts masked (a critical section, given that the separate lists are maintained for each processor). For each tasklet in this initial list, the tasklet is marked as running (only on an SMP), the disabled count is checked, the scheduled flag is cleared, and the tasklet is executed. If any of the checks fail, *e.g.*, the tasklet is already running elsewhere, or is currently disabled, execution is postponed by linking the tasklet back into the original list and marking the corresponding soft interrupt type as pending for that processor (which causes do_softirq to wake the daemon for that processor).

Interrupt Control and Status Functions

Specific hard interrupts can be disabled and re-enabled using functions declared in asm/irq.h. These functions are intended to be used in disable/enable pairs, and can be used without concern about previous calls, as the total number of calls is tracked in the interrupt descriptor array. For example, rather than masking all interrupts to keep the keyboard interrupt (typically IRQ1) from executing, one might write:

```
disable_irq (1);
/* protected keyboard access code goes here */
enable_irq (1);
```

Enabling and disabling individual interrupts is more effective than masking interrupts through IF, since a disabled interrupt cannot run on any processor in an SMP. However, the call to disable can require interaction with the 8259A, which is substantially slower than flipping the IF flag. Note that the disable_irq call waits for the interrupt to finish executing on any processor on which it is currently executing. As a result, *calling this function from an interrupt*

handler may lead to deadlock. Instead, call the `disable_irq_nosync` function, which disables the interrupt but does not wait for a currently executing handler to finish.

As mentioned in the previous section, similar nesting of disables and enables is allowed with the tasklet API. In the case of tasklets, disabling a single tasklet with `tasklet_disable` and disabling all tasklets with `local_bh_disable` (discussed below) have roughly the same cost, but analogous scope, *i.e.*, `tasklet_disable` disables one tasklet on all processors, and `local_bh_disable` disables all tasklets on one processor.

Information about the interrupts currently executing on each processor is recorded in the `irq_stat` array. Each array element is an `irq_cpustat_t` structure, as defined in `asm/hardirq.h`, and is accessed using the standard macros defined in `linux/irq_cpustat.h`:

`softirq_pending (cpu)`	bit vector of pending soft interrupt types
`local_irq_count(cpu)`	number of hard interrupts in progress
`local_bh_count(cpu)`	number of soft interrupts in progress
`syscall_count(cpu)`	number of system calls in progress
`ksoftirqd_task(cpu)`	soft interrupt daemon identifier
`nmi_count(cpu)`	number of non-maskable interrupts in progress

These macros are **lvalues**—*i.e.*, you can assign values to them, increment them, *etc.*—and should be used with another specific macro in places of the `cpu` argument. For example,

```
softirq_pending (smp_processor_id ())
```

returns the bit vector of pending soft interrupt types on the current processor. In GDB, a fixed processor number can be used, and 0 should be used on a uniprocessor. Try `print irq_stat[0]`.

A number of status and interrupt control functions and macros make use of the `irq_stat` array. These functions are defined in either `asm/hardirq.h` or `asm/softirq.h`, and are described in the table below.

status functions

`int in_interrupt ();`	Returns 1 if processor is currently executing an interrupt handler (either soft or hard), 0 otherwise.
`int in_irq ();`	Returns 1 if processor is currently executing a hard interrupt handler, 0 otherwise.
`int in_softirq ();`	Returns 1 if processor is currently executing a soft interrupt handler, 0 otherwise.

control functions

`void irq_enter (int cpu, int irq);`	Increment the count of hard interrupts being executed on processor `cpu`. Second argument is ignored.
`void irq_exit (int cpu, int irq);`	Decrement the count of hard interrupts being executed on processor `cpu`. Second argument is ignored.
`void local_bh_disable ();`	Mask/disable soft interrupt execution on this processor; works by incrementing the count of soft interrupts being executed on current processor.
`void local_bh_enable ();`	Unmask/enable soft interrupt execution on this processor; works by decrementing the count of soft interrupts being executed on current processor.

Example: The Real-Time Clock Driver

Many x86-based computers have real-time clocks that are compatible with Motorola's MC146818 chip. In this section, we briefly discuss some of the capabilities of this chip and examine the interrupt-related functionality in the driver. The first programming assignment for our course, MP1, involves using the real-time clock driver to animate text-based graphics on the screen using system calls and a soft interrupt handler. You will only write the x86 assembly code for handling the system calls and soft interrupts; we will provide the linkage between the driver and your code as well as a user-level test harness to enable you to perform most debugging without recompiling the kernel.

All device driver code for the real-time clock is in `drivers/char/rtc.c`. The interrupt handler, `rtc_interrupt`, is installed in by the `rtc_init` function, as shown here:

```
if (request_irq (RTC_IRQ, rtc_interrupt, SA_INTERRUPT, "rtc", NULL)) {
    printk (KERN_ERR "rtc:  IRQ %d is not free.\n", RTC_IRQ);
    return -EIO;
}
```

RTC_IRQ is defined as 8 in `asm/mc146818rtc.h`. The `rtc_init` function is called by the kernel initialization code, or, if the the real-time clock driver is compiled as a kernel module, when the module is loaded into the kernel.[10] The following declarations generate the appropriate calls, including a call to `rtc_exit` if the driver is compiled as a module and removed from the kernel.

```
module_init (rtc_init);
module_exit (rtc_exit);
```

The `rtc_exit` function uninstalls the interrupt handler using this call:

```
free_irq (RTC_IRQ, NULL);
```

The MC146818 can generate interrupts for three types of events; each type can be configured and handled separately. The first is an alarm clock, which generates an interrupt at a specified time. The second is an update notification, which generates an interrupt each second, *i.e.*, whenever the time changes. The third is an interrupt generated periodically, with a period ranging from 2 to 8192 Hz (limited by Linux), programmable in powers of two. The function below handles the hard interrupts generated by the clock; the modifications in boldface were made to create and schedule an associated soft interrupt after each hard interrupt. These soft interrupts are then used to drive the animation in MP1, and the function `mp1_rtc_tasklet` referenced in the tasklet declaration must be written by each student.

```
/* Student's tasklet */
static DECLARE_TASKLET (mp1_rtc_tasklet_struct, mp1_rtc_tasklet, 0);

static void rtc_interrupt (int irq, void* dev_id, struct pt_regs* regs)
{
    /*
     * Can be an alarm interrupt, update complete interrupt,
     * or a periodic interrupt.  We store the status in the
     * low byte and the number of interrupts received since
     * the last read in the remainder of rtc_irq_data.
     */

    spin_lock (&rtc_lock);
    rtc_irq_data += 0x100;
    rtc_irq_data &= ~0xff;
    rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

    if (rtc_status & RTC_TIMER_ON)
        mod_timer (&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

    spin_unlock (&rtc_lock);

    /* Now do the rest of the actions */
    wake_up_interruptible (&rtc_wait);

    /* Schedule the MP1 tasklet to run later */
    tasklet_schedule (&mp1_rtc_tasklet_struct);

    kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
}
```

[10] Linux supports loading sections of code called modules into the kernel on demand, *i.e.*, only when they are needed, and being able to remove them when they are no longer necessary. Modules can keep the kernel small while supporting plug-and-play style functionality, but they make debugging with KGDB more complex. We thus chose to turn module support off in our kernel configuration files, and will not discuss it in any detail in the class.

The real-time clock device provides a file interface through which four-byte updates arrive periodically. The `rtc_irq_data` variable tracks both the number of interrupts that have occurred since the last update through the file interface as well as the type of interrupt last seen. The interrupt type occupies the low byte of the variable, and the remaining (high) 24 bits are used to count the interrupts between updates. Symbolically, as given in `linux/mc146818rtc.h` the flag bits are RTC_IRQF to indicate that some interrupt has occurred, RTC_PF for a periodic interrupt, RTC_AF for an alarm clock interrupt, and RTC_UF for an update interrupt. The contents of the `rtc_irq_data` variable are delivered as the update when a program reads the clock via the file interface.

The file interface may be running on another processor when the interrupt occurs, so the handler first acquires a spin lock to protect the accesses as well as the device interaction (using `asm/mc146818rtc.h`'s CMOS_READ macro). The conditional and timer operation within the critical section handle cases in which the period of the periodic interrupt is set too high and an interrupt is missed. Certain devices have higher priority than the real-time clock, and their interrupt handlers may prevent an overly frequent clock interrupt from being observed before the next one should occur. In this case, the clock may stop generating interrupts, and the device driver must notice, reset it, and estimate the number of interrupts lost. Use of Linux' internal timers are outside of the scope of our interest for now, but feel free to read the rest of the clock driver code as an example of use.

After releasing the spin lock, the original interrupt handler wakes up any program blocked waiting to read an update from the clock's device file (`wake_up_interruptible`) and sends a signal to any program that has asked to receive signals when the clock's device file has new data to be read (`kill_fasync`). For MP1, we simply inserted scheduling the tasklet into this list of things that need to be done by the handler. Along with the declaration and writing the soft interrupt handler (left to you in MP1), this simple addition to the code is enough to support our needs. For proper module support, we must also add a call to `tasklet_kill` after the call to `free_irq` to wait until any scheduled soft interrupts have been executed.

Terminology

You should be familiar with the following terms after reading this set of notes.

- procedural abstractions
  - system call
  - interrupt
  - exception
  - handler (function)
- processor support
  - vector table
  - Interrupt Descriptor Table (IDT)
  - interrupt enable flag (IF)
  - non-maskable interrupt (NMI)
- input/output concepts
  - I/O port space
  - independent vs. memory-mapped I/O
- software and programming abstractions
  - application programming interface (API)
  - linkage
  - lvalues
  - jump table (for function indirection)
- synchronization concepts
  - critical section
  - atomicity
- synchronization mechanisms
  - mutex
  - spin lock: lock/obtain and unlock/release operations
  - semaphore: down/wait and up/signal operations
  - reader/writer lock
- synchronization problems
  - race condition
  - deadlock
  - starvation
- interface between processor and devices
  - interrupt controller
  - Programmable Interrupt Controller (PIC), such as Intel's 8259A
  - end-of-interrupt (EOI) signal
- interrupt abstractions in software
  - interrupt chaining
  - hard vs. soft interrupts
  - Linux' tasklet abstraction

Advanced Topics

This section mentions a few topics for those interested in further study. None of these are included in the required course material.

Auto-probing for interrupts: Linux has support for automatically finding the interrupt vector generated by a device rather than hardcoding it into a device driver. The protocol for using this support is described at end of `linux/interrupt.h`. Essentially, the device driver puts the device into a state in which it will generate an interrupt shortly. The driver then waits for a bit, during which time the vector numbers of all interrupts received but not already associated with other devices are recorded. At the end of the waiting period, the correct vector is identified if exactly one unknown interrupt arrived during the period.

Spin locks: The caching and bus transaction aspects of the algorithms may be beyond you at this point, but are mentioned here for interest.

The x86 ISA uses the notion of instruction prefixes to change the properties of certain instructions. Instruction prefixes are usually one-byte sequences placed in the code before another instruction. One such prefix is called "LOCK," and has the effect of claiming the memory bus on behalf of the processor for the duration of the following instruction. On a uniprocessor, or for an instruction that performs zero or one memory bus transactions, this prefix has no effect. However, for an instruction that reads a value, modifies it, and writes it back to memory—sometimes called a read-modify-write operation—locking the bus on a multiprocessor guarantees that no other processor's bus transactions can be interleaved between the two transactions associated with our processor's instruction. Read-modify-write operations are the instruction-level equivalent of critical sections. If two processors try to add one to the value at a memory location in this way, most programmers would expect a total of two to be added to the value in memory. However, without lock prefixes on the instructions, both processors may read the same value from memory and thus write back the same value, *i.e.*, the original value plus one. In most modern systems, atomic operations are performed in the lowest-level cache, which is much faster and less intrusive than locking the memory bus, but is still fairly slow relative to the processor's clock (tens of cycles per operation).

The simplest spin lock algorithm, known as **test-and-set** (T&S), repeatedly attempts to obtain a lock by atomically reading a bit from memory and replacing that bit with a 0 (using either an instruction with a LOCK prefix or a special atomic instruction). If the bit was previously a 1, the lock has been obtained. If it was previously a 0, some other code is holding the lock, and the algorithm tries again. Most modern ISAs, including newer flavors of x86, have instructions designed specifically for this purpose (see BTR and BTS).

While the T&S algorithm can be best when lock contention is low, it generates far too much bus traffic when more than one processor is waiting on a lock. Each processor continuously demands ownership of the lock, which thrashes back and forth across the bus, slowing traffic from other processors, potentially including the one that owns the lock.

Linux uses a variant of a slightly better algorithm known as **test-and-test-and-set** (T&T&S). This algorithm takes advantage of the fact that the cache line with the lock can be held in many processors' caches so long as none of them tries to write to the lock. In particular, after failing to obtain the lock once, a processor backs off and reads the lock until it sees the lock present. In effect, bus traffic is generated only when a new processor tries to obtain the lock or the processor with the lock gives it up. The specific variant used in Linux decrements a lock byte atomically and decides whether or not the lock was obtained by checking the sign flag.[11]

Memory consistency: The level at which we have discussed synchronization is necessarily somewhat simplistic, and ignores the fact that some ISAs allow a processor to reorder memory transactions in the interest of performance. On an ISA that allows such reordering, telling the compiler not to move memory operations past critical section boundaries is not sufficient to ensure atomicity. Instead, additional instructions must be used to tell the processor itself to avoid such behavior. These issues are covered to some degree in ECE411, and in more depth in ECE511.

Lock-free synchronization: For general and specific methods of constructing synchronization methods without locks, the non-blocking and wait-free work by Herlihy and Wing, as well as the notion of universal synchronization primitives, is the natural starting point. Rather than provide citations here, I refer you to the bibliography of my dissertation, which you can find online through my home page. To my knowledge, almost none of this material is covered by classes here, although you may see some of it in ECE/CS428.

Connecting devices to the processor: If you are interested in the details of the hardware used to connect a processor to devices, busses, and other hardware elements, you should take ECE412, which focuses on this type of problem in the context of an IPAQ PDA (personal data assistant, *i.e.*, handheld computer) running an embedded version of Linux.

[11] This approach relies on the number of contenders being no more than 129, at which point wraparound magically generates 128 new locks.

Synchronization implementations: Some documentation on the algorithms used for the synchronization constructs in Linux is available in the header files.

Header and Source Files

The table below serves as a brief reference on the contents of those header and source files within the Linux-2.4.18-3 kernel that are relevant to the topics covered in these notes.

header files	
include/asm	
atomic.h	a few atomic primitives for x86
bitops.h	bit operations (*e.g.*, test bit, find first bit set, *etc.*)
hardirq.h	irq_cpustat_t structure, which tracks hard and soft IRQ information for each processor; a few interrupt status-checking functions; irq_enter and irq_exit macros
hw_irq.h	SYSCALL_VECTOR (0x80); BUILD_IRQ macro used in `i8259.c` to generate handler code for IDT
irq.h	controller-independent interrupt management interface
locks.h	antiquated — DO NOT USE
mc146818rtc.h	x86-dependent parameters for real-time clock
rwlock.h	helper code for r/w locks and semaphores
rwsem.h	x86 implementation of reader/writer semaphore macros
semaphore.h	defines semaphores in interruptible and non-interruptible forms
softirq.h	soft interrupt management and status-checking functions
spinlock.h	spin lock and rw_lock functionality for x86 SMPs
system.h	x86-specific hardware interrupt primitives (*e.g.*, masking, flag save/restore)
include/linux	
interrupt.h	architecture-independent hard and soft interrupt structures and functions
irq.h	interrupt controller abstraction (PIC jump table), hard interrupt description structure, status flags
irq_cpustat.h	standard access macros for CPU stat structure defined in `hardirq.h`
mc146818rtc.h	architecture-independent parameters for real-time clock
rwsem.h	public interface to reader/writer semaphores
rwsem-spinlock.h	architecture-independent implementation of reader/writer semaphore macros
tqueue.h	task queues; antiquated in Linux-2.6 — DO NOT USE
sched.h	request and free IRQ function prototypes

source files	
arch/i386/kernel	
entry.S	interrupt, exception, and system call linkage
i8259.c	8259A PIC code, including jump table (XT-PIC in `/proc/interrupts`)
irq.c	irq_desc declaration and initialization; most of the hard interrupt handling code
nmi.c	NMI handling
semaphore.c	functions used to support semaphores and reader/writer spin locks when contention seen on lock (fast path for no contention given inline via header files)
drivers/char	
rtc.c	real-time clock device driver (mc146818)
kernel	
softirq.c	soft interrupt code, including kernel daemon for wakeup, *etc.*
lib	
rwsem.c	functions used to support reader/writer semaphore when contention seen
rwsem-spinlock.c	alternate implementation — NOT COMPILED INTO OUR KERNEL

ECE391: Computer Systems Engineering **Lecture Notes Set 3**

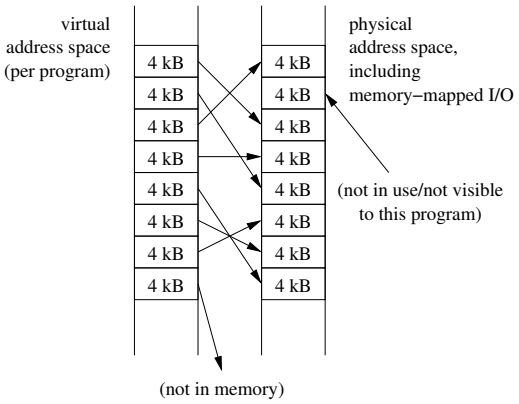
Virtual Memory: Segmentation and Paging

The notes provided here are primarily a transcription of the lectures on the subject of virtual memory. In addition to the class textbooks, material for these lectures was drawn from the Intel ISA manuals and the ECE 411 lecture notes, both of which are available to you online.

Virtual Memory

The central concept of **virtual memory** is the insertion of a level of indirection between the memory address space seen by a program and the address space of the actual memory in the system. For clarity, we refer to the addresses seen by a program as **virtual addresses**; they are also sometimes called **logical addresses**, and simply addresses when the context makes it clear that they pertain to the address space of a particular program. In contrast, we refer to the addresses used by the memory as **physical addresses**. The hardware that translates virtual addresses to physical addresses is sometimes referred to as a **memory management unit** (MMU), although this term more often refers to hardware that sits between the system bus and the memory chips. Virtual to physical address translation occurs on the processor itself in most designs, thus physical addresses are usually the same as the values placed on the system/memory bus, and are always the addresses seen and used by the memory chips. Some bus designs, however, support the use of virtual addresses, with translation to physical addresses occurring in an MMU between the bus and the memory chips. It is also worth mentioning that some ISAs, such as x86, use virtual addresses regardless of the privilege level, while other ISAs use virtual addresses only while executing at user level. When operating at kernel level, these ISAs use physical addresses.

Keeping track of a mapping from virtual addresses to physical addresses requires a certain amount of space. To keep the required space small, the mapping is usually done at a fairly large granularity. Generally speaking, the mapping for each **page**—the term for a chunk of virtual memory—requires a pointer into physical memory and a few extra bits. Typically, the extra bits can be absorbed into the space required for a normal pointer by recognizing the fact that, for simplicity, the page should be aligned to a multiple of the page size in both the virtual and physical address spaces. For example, with pages of 4 kB, the page address is a multiple of 4,096 in both address spaces, thus the low 12 bits of a pointer can be reclaimed for other purposes. For a physical address space that contains 2^{32} addresses, a 4 B pointer suffices for each page.



A program's memory mapping may not map every page into a page in memory. Instead, some pages may not be mapped at all, while others are mapped into a device's memory (*e.g.*, the video memory) or into a block on a disk. This flexibility helps virtual memory to achieve the four objectives for its use, as discussed next.

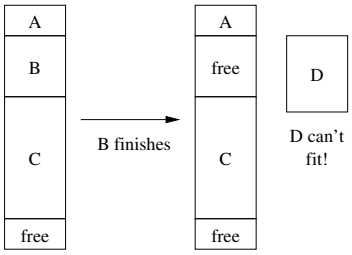
Tradeoffs of Virtual Memory

The indirection used by virtual memory provides four main advantages over a system without such indirection. In decreasing order of importance, these advantages are protection between programs, more effective sharing of memory resources, reduction or elimination of memory fragmentation, and the elimination of the need for code relocation when starting a program.

Beginning with **protection**, virtual memory prevents a program from accidentally or deliberately destroying another program's data. The memory used by the second program is simply not accessible to the first, as no virtual address in the first program translates into the physical address used to hold the second program's data. The kernel similarly benefits from this protection, although simpler mechanisms can also be used for user-kernel protection, such as restricting access to a particular range of physical addresses.

Sharing occurs through use of shared libraries, which is particularly common with dynamically loaded/linked libraries (DLLs), but is not necessarily supported by every system that allows dynamic linking. **Sharing** takes advantage of the fact that none of the programs modifies the code in the library by placing the code in physical memory and mapping it into two or more programs' virtual address spaces, possibly at entirely different virtual addresses. Similarly, programs are able to share memory more effectively by only bringing library code actually being used into physical memory. Although the rest of each library is still logically a part of the program's virtual address space, it is never actually loaded from the disk into memory.

Memory fragmentation occurs when a system without virtual memory attempts to run more than one program at a time. The memory allocated to each program must usually be a contiguous region of system's physical address space, but this address space becomes fragmented as programs enter and leave the system. Imagine that a system with 128 MB of memory executes programs A, B, and then C, with total sizes 16 MB, 32 MB, and 64 MB. If the system fills memory from the low end, program A starts at address 0, B at address 16 MB, and C at address 48 MB. The region from 112 MB to the end of memory is free, allowing a fourth program requiring up to 16 MB to execute. If program B terminates, its memory is freed, leaving a hole of 32 MB and the original free region of 16 MB. At this point, despite the fact that a total of 48 MB are free, the system can only start a new program requiring 32 MB or less, since the largest contiguous region is the one previously occupied by program B.

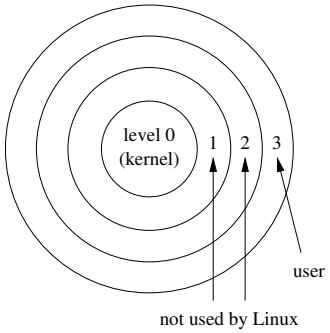


Finally, virtual memory avoids the problem of **relocation** when a program is loaded for execution. In the example just mentioned, the addresses used by each program varied. Thus, any absolute addresses in either the code or data must be adjusted to avoid incorrectly operating on memory not owned by the program (or executing code that is not part of the program!). The need for relocation is one of the main reasons that systems without virtual memory typically require contiguous chunks of memory to start programs; if a program is broken up into multiple non-contiguous regions to fit into memory, correcting any stored addresses becomes much harder. With only a single region, adding the value of the address at the start of the region usually suffices for relocation. For most ISAs, information as to which parts of the code and data must be updated during relocation must be stored along with the code and data for the program.

These benefits do come at a cost, of course. The indirection necessary to support virtual memory introduces additional complexity, storage requirements, and time overhead. In desktop or server systems, these costs are generally acceptable. In contrast, many embedded processors do not support virtual memory, either because the set of programs to be run on the processor never changes, or because the additional power, space, or variability in time required for address translation are too burdensome.

x86 Protection Model

The protection model in the x86 ISA is used to support the protection provided by virtual memory, thus we introduce it before describing the details of the mechanisms used to support virtual memory. The model can be illustrated by a set of concentric rings, and is often referred to as a ring or donut scheme. The center, or level 0, is the most privileged level, and corresponds the core of the operating system. Most operating systems, including Linux, do not use levels 1 or 2, but the x86 ISA provides these additional levels for less trusted code or users. Finally, the outermost ring, level 3, corresponds to the privilege level for user code. All user code operates at this level; superuser privileges are supported through special checks in the kernel rather than through direct execution of the superuser's programs at privilege level 0.



To make this type of privilege system work as intended, code operating at a given privilege level must never call into code operating at a lower (numerically larger) privilege level. Similarly, calls from a given privilege level must pass through a small set of narrowly defined interfaces (*e.g.*, system calls) to request services from higher (numerically smaller) privilege levels, and those services must first validate any data passed in by the calling code.

The **current privilege level**, or CPL, is kept in a register as part of the processor state. Accesses to code or data are also associated with a **request privilege level**, or RPL. The RPL allows code executing on behalf of less privileged code to have the hardware check the necessary privileges for certainly rather than making all checks in software. In particular, each memory location accessed is also associated with a **descriptor privilege level**, or DPL. On any access, the processor checks that the maximum of CPL and RPL is no greater than DPL. If either CPL or RPL is larger than DPL, the processor generates an exception (a general protection fault), preventing the illegal access.

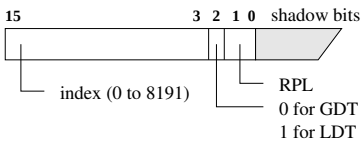
x86 Segmentation

The x86 ISA provides two distinct methods to support protection between programs: segmentation and paging. Segmentation has been present from the beginning of the ISA, whereas support for paging was added in the 80386 ISA. Although segmentation is no longer used by most operating systems, it remains for backwards compatibility reasons (the ISA manuals still claim that it is useful). Paging, on the other hand, is almost always used, and is the abstraction typically associated with virtual memory. An x86-based processor translates each virtual address into a **linear address** using segmentation, then optionally translates each linear address into a physical address using paging. If paging is not used, the linear address is used directly as a physical address.

A **segment** is a contiguous portion of an address space, such as the 32-bit space of physical addresses. A processor implementing the x86 ISA *always* uses segmentation. It cannot be turned off, only simplified to the point of being meaningless.

Segments are described in one of two arrays called descriptor tables. The **Global Descriptor Table**, or GDT, can be used by any program, while each program can also have its own **Local Descriptor Table**, or LDT. A pointer (a physical address) to the GDT is held in the 48-bit GDT register (GDTR) along with a 16-bit limit (size - 1 in bytes). Each segment descriptor takes 8 B, thus the GDT can describe up to 8,192 segments. Segment 0 of the GDT is not used. Each segment descriptor contains a base address, a limit on the offset, a DPL, and some other bits. Given a reference to a particular segment, the processor checks that the given address does not exceed the segment limit, then simply adds the base address for the segment to calculate the linear address. The additional bits in a segment descriptor allow the descriptor to differentiate code (executable and possibly readable) from data (readable and possibly writable) as well as a few other somewhat useful things. See the ISA manual for further details.

Six segment registers are used to select the segment being referenced. These registers include a code segment (CS), data segment (DS), extra segment (ES), stack segment (SS), and two others (FS and GS). The first four are used implicitly by a number of useful instructions. Subroutine calls use the code segment, for example, while loads and stores (MOVs) use the data segment. String copying uses both DS and ES. Stack operations use SS. Any segment can be used explicitly with a given instruction by prefixing the instruction with an additional byte or two.

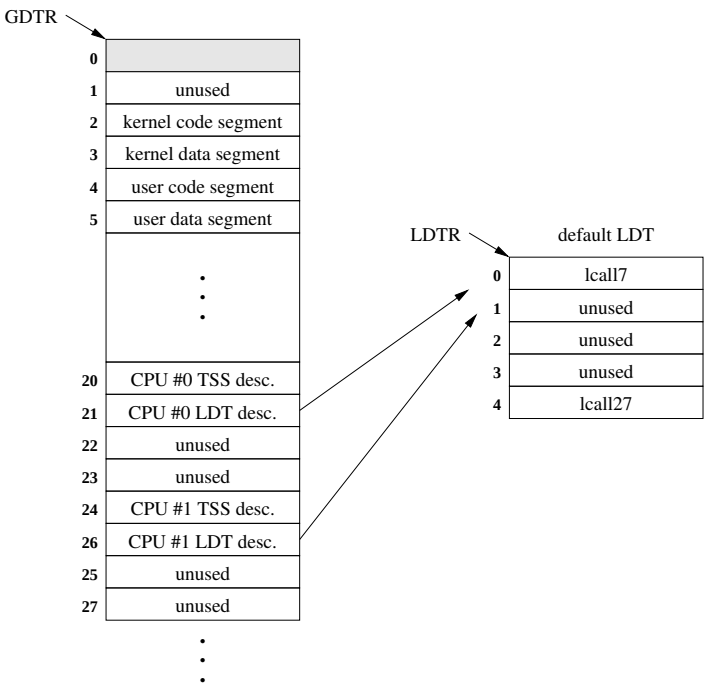


As shown above, only 16 bits of each segment register are visible architecturally. The ISA states that additional **shadow bits** (not directly accessible to the ISA) should be used to cache the values stored in the GDT or LDT. These bits are not kept consistent with the tables themselves, thus segment registers must be reloaded whenever entries in the GDT or LDT are changed, or when a new LDT is selected.

The interpretation of the 16 visible bits of each segment register is as follows. Bit 2 selects either the GDT (0) or the LDT (1). Bits 15 through 3 are an index into the appropriate table; alternatively, these bits can be viewed as an offset, since each descriptor occupies eight bytes. By zeroing the low three bits and adding the table pointer, one obtains a pointer to the desired descriptor. Bits 1 and 0 are the RPL to be used for the access, as described in the previous section.

Entries in the GDT do not always describe segments. They can also describe LDTs as well as **task state segments** (TSSs), which are intended to hold information pertaining to an individual program. As with the GDT, a special LDT register (LDTR) is used to hold a pointer to the LDT. The LDTR is a 64-bit register, holding the pointer, a limit on the LDT (size - 1 in bytes), and the index of the LDT within the GDT.

Linux simplifies segmentation as much as possible. The picture to the right is redrawn from `asm/desc.h`. Essentially, four segment descriptors are used to support code and data accesses by both the kernel and user-level programs. Only the type (code/data) and RPLs (0/3) differ; otherwise, all four segments start at address 0 and extend over the entire 4 GB range of physical addresses.



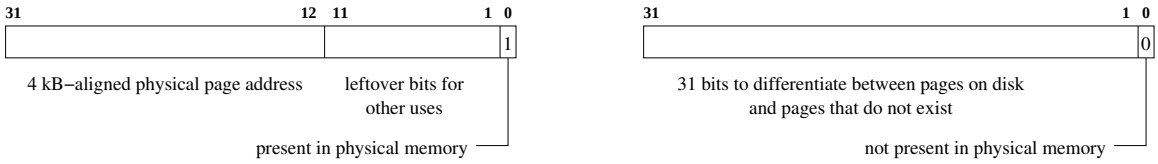
Linux also uses one GDT entry to hold an LDT descriptor for each processor, and a second entry to hold a TSS descriptor for each processor. Hardware support for multiple programs is not used; instead, Linux rewrites the TSS for a CPU before running a new program on that CPU. Similarly, only a single LDT actually exists; all processors' LDT entries point to the same LDT in physical memory. This LDT contains only two descriptors, which are used to support execution of binaries from other operating systems. Thus, Linux does everything to avoid segmentation short of turning it off; the x86 ISA does not allow that.

x86 Paging

As mentioned before, the x86 ISA uses two levels of indirection to map each virtual address into a physical address. Segmentation provides the first level of indirection. The second level is provided by paging, which is optional on x86, but is a more effective abstraction for achieving the benefits of virtual memory. Paging is the only mechanism usually discussed in conceptual descriptions of virtual memory.

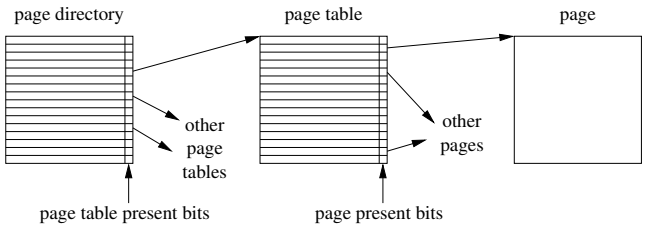
Any given page of a virtual address space can be in one of three states: present, swapped out, or nonexistent. Starting from the last state, not all pages of a virtual address space need to be defined. Just as a program running on a system without virtual memory does not make use of all memory addresses, a program running on a system with virtual memory never touches many of the possible addresses. The mapping from some pages can thus be left completely undefined. The second case is that in which a page exists but is not actually in the physical memory. We say that such a page is **swapped out**, as the contents are typically present on a disk. In the case that the page contains code that has not been previously executed by the program, the data are usually left in the regular filesystem (the executable image on the disk). In the case of data that were in use but had to be moved out of memory to make room for more data or another program, the data for the page are kept on a special **swap disk**, which is inaccessible to normal users. If a program attempts to access (read from or write to) a page that does not exist or that has been swapped out, the processor throws an exception, allowing the operating system to either put the program to sleep and move the data into memory (when swapped out), create a new page (growing the stack, for example), or send a signal to the program (a last resort intended to allow the program to deal with the failure itself, if desired). The "Segmentation fault" error common in pointer-based code is the default result of this signal.

The encoding shown on the next page is called a **page table entry** (PTE), and an array of them is a **page table**. A page table stores the mapping from virtual to physical memory or disk. As shown in the figure, we can encode the state of a page of virtual memory using four bytes. Bit 0 is used to differentiate pages that are present in physical memory from those that are not. If this bit is 0, an x86-based processor ignores the rest of the bits, allowing the operating system to use them for differentiating between pages that do not exist and pages stored on a disk. If the present bit is set, bits 31 to 12 are interpreted as a physical page address (the low bits are replaced with 0s). The leftover bits can be used for other purposes; the uses for most of these bits are out of the scope of our course, but we discuss a few of the others later in these notes.



Consider the use of a single page table for a program. For every 4 kB of space in the virtual address space, we need 4 B to store a PTE. If the virtual address space contains 2^{32} addresses, we need 2^{20} PTEs, or 4 MB of data. That's still a lot of space, particularly when one considers the fact that most of the virtual address space in most programs is simply empty. Do we really need to use so much space to describe unmapped pages? How can we work around this problem?

The solution is to page the page table. Page tables are simply another form of data, so we can treat them as pages and leave them undefined or swap them out onto disk when they are not in active use. Alternatively, one can view this solution as a hierarchy of page tables. The figure to the right illustrates the idea: page tables are organized under a **page directory**. Each page table is represented by a page directory entry (PDE), which, like a PTE, may also indicate that the page table does not exist, or that it is not currently in physical memory.

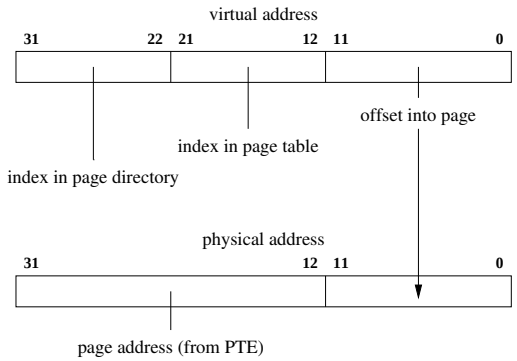


A single page table should occupy the same amount of memory as a page in order to simplify the process of moving page tables in and out of memory. Given a 4 kB page size and a 4 B PTE, a single page table then holds 1,024 PTEs. Similarly, the page directory must hold enough PDEs to describe the entire virtual address space. Each PDE describes 2^{10} pages of 2^{12} bytes. Thus, with a 32-bit address space, we need $2^{10} = 1,024$ PDEs in a page directory. Conveniently, a page directory can thus be treated as yet another page and swapped out when a program is inactive; Linux does not currently do so, however.

The encoding of the x86 PDE is nearly identical to that of the x86 PTE; the main differences lie in the meaning of the leftover bits. In particular, when a PDE has the present bit (bit 0) set, the high 20 bits provide a page-aligned physical address for a page table, just as the high 20 bits of a PTE provide a page-aligned physical address for a page.

Another special register, known informally as the **Page Directory Base Register** (PDBR) and in assembly as Control Register 3 (CR3), holds the physical address of the page directory. This register is changed whenever the operating system decides to allow a new program to execute on the processor; remember that each program has its own virtual address space, and thus its own page directory somewhere in physical memory. The kernel must keep track of these structures and set PDBR correctly whenever a new program is allowed to execute.

Consider the process by which an x86-based processor translates a program's virtual address, as provided by a machine instruction, into a physical address for use by the memory. The processor starts with the value in PDBR, which points to the page directory (a physical address to avoid infinite recursion in the translation process). The high ten bits of the virtual address are used to form an index into the page directory, from 0 to 1,023. If the corresponding PDE has its present bit clear, the page table is absent, and an exception is thrown to allow the operating system to decide what to do. If the PDE's present bit is set, the processor extracts a pointer to the appropriate page table; again, this pointer is a physical address. Next, the processor uses bits 21 through 12 to form an index (0 to 1,023) into the page table. If the PTE at this index indicates that the page is not present, an exception is generated by the processor. If it is present, the processor extracts the physical address of the page from the PTE. The remaining 12 bits of the virtual address are then simply appended to the page address, and operate as an offset within the page. *Note that the process of translation is far too slow and involves too many memory accesses to occur every time a load or store instruction is executed.*



Modern processors use additional hardware to cache the results of this translation. Each of these **translation lookaside buffers (TLBs)** thus maps a 20-bit virtual address for a program into a 20-bit physical address; the offset varies by access. Again for performance reasons, the number of TLBs is usually fairly small, with perhaps 32 or 64 used by a typical processor implementation. Translation is still necessary when no TLB holds the necessary mapping for an address, in which case the access (load or store) is said to take a **TLB miss**. The x86 ISA uses the page tables to load the TLB with hardware, while some other ISAs transfer control directly to the operating system on a TLB miss, allowing the operating system to load the TLB and eliminating any need to understand the page table structures in hardware. Note that since the translations cached by the TLBs pertain to the address space of a single program, the translations are **flushed** (*i.e.*, discarded) when the value of the PDBR is changed.

PTE/PDE Functionality

The last points for discussion center on the uses of the leftover bits in both the PTE and PDE encodings for pages present in physical memory. The x86 ISA leaves a few of these bits to the discretion of the operating system, while others are understood by the hardware but outside the scope of our class. In general, however, the bits are used for two purposes: additional protection and improved performance. We consider examples of both uses.

On the protection side, each PTE (and PDE) contains a bit for the privilege level required for use in address translation. This bit can either be set to User (U) or to Supervisor (S). When set to U, no privilege level check is made; the translation is accessible to anyone. When set to S, a privilege level (maximum of CPL and RPL) of less than 3 is necessary to use the translation. Similarly, PDEs and PTEs can mark the targeted pages as read-only or read/write. Writes made to pages marked as read-only in this manner generate exceptions.

We also discuss two bits that serve to improve performance. The first is the global (G) flag. A global page table or page is one that is present in all programs' virtual address spaces (*in the same place!*), allowing the processor to retain TLB translations even when PDBR is changed. Operating systems such as Linux use the G flag to map kernel code and data into all programs, for example, and thus avoid having to flush TLBs for these translations every time the kernel starts executing a different program.

Finally, the x86 ISA supports both 4 kB and 4 MB pages, with separate TLBs for each page size. Use of larger pages reduces the number of TLBs required to access a given region of physical memory, but requires that larger chunks of contiguous physical memory be handed out (thus risking fragmentation). The translation process uses a Page Size (PS) bit in the PDE (not the PTE) to decide whether the physical address in the PDE points to a page table or directly to a 4 MB page.

Terminology

You should be familiar with the following terms after reading this set of notes.

- virtual memory
  - virtual/logical address
  - linear address
  - physical address
- problems addressed by virtual memory
  - protection between programs
  - code/data sharing
  - memory fragmentation
  - code/data relocation
- x86 segmentation support
  - segment
  - shadow bits (in a register)
  - Global Descriptor Table (GDT)
  - Local Descriptor Table (LDT)
  - task state segment (TSS)
- x86 privilege model
  - current privilege level (CPL)
  - request privilege level (RPL)
  - descriptor privilege level (DPL)
- paging abstractions
  - Page Directory Base Register (PDBR)
  - page directory
  - page table
  - page
  - page table/directory entry (PTE/PDE)
  - swapped out
  - swap disk
- address translation
  - memory management unit
  - translation lookaside buffer (TLB)
  - TLB miss
  - TLB flush

From C to C: System Call Linkage

System calls have much in common with other types of function calls, but use a slightly different calling convention. In this section, we examine the Linux system call interface, first in the abstract, and then with an example illustrating the linkage used to connect a user-level program with the appropriate function in the kernel.

Traditionally, system calls are identified by number rather than by name, and generally use registers to pass operands into the operating system rather than pushing the operands on to the stack. The system call numbers for Linux are listed in `asm/unistd.h`; if new system calls are added, they are usually appended to the end of the list so as to maintain backwards compatibility with existing code. Operating systems can instead use a dynamic string-to-number mapping to keep the table organization cleaner, but such a mapping incurs a little overhead and does not solve the backwards compatibility problem.[1]

The return value from a system call is placed in EAX, and can have an arbitrary type (of the appropriate size). Errors are indicated by reserving a small range of possible return values (from -1 to -4095) and returning an error values from a second enumerated list (see `asm/errno.h`). The error values are listed as positive numbers, but by convention are negated before being returned by a system call, and are typically re-negated by library code before being handed back to a program.

System calls in Linux are typically written as C functions, and the standard C library provides C function interfaces to many system calls, but between the two lies the processor's system call vector table, which requires linkage from C functions at user level to the form understood by the processor and back to the C functions inside the kernel.

We start by looking at the kernel side. In Linux, all system calls use a single entry in the IDT, with vector number 0x80. This entry is created when the kernel is started as part of the `trap_init` function in `arch/i386/kernel/traps.c`. The handler for all system calls is the `system_call` function in `arch/i386/kernel/entry.S`.

The `system_call` function begins by saving all registers to the stack. It next checks whether the program is under the control of a debugger, in which case the debugger is allowed to intercept the system call before the system call executes on behalf of the program. The EAX register is used to select the specific system call, and its value is checked to ensure that a valid system call was requested. If valid, EAX is used as an index into a jump table, `syscall_table`, which also appears in `entry.S`, and an indirect call is made to the code for the specific system call. After this call returns, the function does a little cleanup and returns control to the program.

As you may have guessed, the top of the stack on entry to a system call looks exactly the same as it does on entry to the C function for interrupts, `do_IRQ`. However, rather than make use of all of the saved registers, system calls use only the first few saved registers on the stack, treating them as if they had been passed as C parameters. The `asmlinkage` macro defined in `linux/linkage.h` ensures that the functions do not expect any arguments in registers, although this behavior is the default anyway. Starting from the top of the stack, the registers are EBX, ECX, EDX, ESI, EDI, EBP, and EAX. System calls that take one argument thus require that a program put the argument into EBX and execute an `INT $0x80` instruction; those with two place the first argument in EBX and the second in ECX; and so forth. The system calls' calling convention is thus implicitly defined by the order of register pushes chosen by the OS.

To see how the linkage works on the user-level side, let's take a look at the C library call that serves as a wrapper to `sys_open`. The system call side is defined in `fs/open.c` with the following signature:

```
asmlinkage long sys_open (const char* name, int flags, int mode);
```

and the user-level library function is declared in `/usr/include/sys/fcntl.h` as follows:

```
int open (const char* name, int flags, int mode);
```

[1] Indeed, it's perhaps a little worse, since compatibility now implies supporting human-readable names that may have been poorly chosen rather than numbers that can be renamed for clarity in the code.

The GNU version of C's `open` call is loaded on demand when a program attempts to make use of one of the functions in `libc`. This function is called by `fopen` to open the I/O channel to be used for the stream, and its arguments are simple functions of the `fopen` arguments. The version below was captured with `gdb` and hand-optimized to remove redundancy and unnecessary distractions; the core of the code is the same as the original and illustrates the typical linkage necessary for system calls.

```
open:  pushl  %ebx                # save EBX to stack
       movl   0x10(%esp),%edx     # EDX ← mode
       movl   0x0C(%esp),%ecx     # ECX ← flags
       movl   0x08(%esp),%ebx     # EBX ← name
       movl   $0x05,%eax          # open is system call #5
       int    $0x80               # do the system call
       cmpl   $0xFFFFF001,%eax    # -1 to -4095 are errors
       jb     done                # others are valid descriptors
       xorl   %edx,%edx           # negate error number
       subl   %eax,%edx
       pushl  %edx                # save EDX (caller-saved)
       call   __errno_location    # get pointer to errno
       popl   %ecx                # pop error number into ECX
       movl   %ecx,(%eax)         # save error number in errno
       orl    $0xFFFFFFFF,%eax    # return -1
done:  popl   %ebx                # restore EBX from stack
       ret
```

The `__errno_location` function is an interesting example of a trick used to find static data contained in a relocatable library. As the library may be loaded to different parts of memory from the program's point of view, the address must be adjusted for the current execution. Code addresses are linked in dynamically through stubs much like a jump table. Data addresses, on the other hand, often use the trick shown below.

```
__errno_location:
       call   getIP               # EAX ← raddr
raddr: addl   $errno-raddr,%eax   # adjust return value
       ret
getIP: movl   (%esp),%eax         # read return address into EAX
       ret
```

The code uses the call instruction to force the processor to store the next EIP on the stack, then reads the stored value into EAX, effectively loading EAX with the address of a known position in the library code. As this position always resides at a fixed offset from the address of the desired data, one can obtain a pointer to the data—in this case `errno`—by simply adjusting the result by the right amount.

ECE391: Computer Systems Engineering **Lecture Notes Set 5**

Signals, or User-Level Interrupts

As with the virtual memory notes, this set contains mostly the content to be discussed in lecture rather than any additional material. The source for these notes is Chapter 10 of Understanding the Linux Kernel and the Linux manual pages and source code. Note that not all operating systems implement the POSIX standard, so you may need some amount of OS-dependent code for a portable implementation using signals.

The Interrupt Analogy

A **signal** is the user-level analogue of an interrupt request line, and the similarities between the two by far outnumber the differences. Signals are a software abstraction that affects a single program—the virtualization of a processor—in the same way that interrupts affect a real processor. Like interrupts, signals are raised asynchronously with respect to a program's execution, and can cause execution of an associated handler function. Although most signals can be **ignored**, **blocked** (masked temporarily), or **caught** (caused to execute a program-defined handler function), certain signals operate as NMIs, and can never be ignored, blocked, or caught. As with physical interrupt request lines, raising a signal twice does not necessarily cause two invocations of an associated handler; if the two signals are raised closely enough in time, the handler is executed only once. Traditionally, and by default, no information other than the signal number is provided to the handler, and a signal is masked while the handler for that signal executes.

The traditional difference between signals and interrupts is limited to the lack of formal association between programs sending signals and the numbers assigned to the signals. Certain permissions are required in order to send a signal to a program. As the cooperation of the kernel is necessary to deliver a signal, these permissions are enforced by the system call (`kill`) used to send a signal from one program to another. Any entity with the appropriate permission to send signals to a program, however, can send any signal; there is no implicit association between programs and signal numbers, nor is there an equivalent of the mapping from interrupt number to "device," as there is with hardware interrupts, although such a mapping can be constructed as part of the software using signals.

A new set of real-time (RT) signals was introduced by a POSIX standard. Unlike traditional signals, these signals are **queued** for delivery, so raising a signal three times causes the associated handler to be executed three times, regardless of timing. The standard also extended the signal interface to allow additional, signal-specific information to be passed to signal handlers. This additional information can in some cases serve to differentiate senders without additional infrastructure. Finally, POSIX standards also define interfaces to be used for sending signals to specific threads rather than allowing a signal to be caught by any thread in the target program.

Signal Behavior Options

Unlike hard interrupts, each signal is associated with a particular default action to be taken by the program. These actions are set up by the operating system, although they can also be inherited from a program's parent (the program that started the execution of a given program). Default actions for signals include ignoring the signal, terminating the program, terminating the program and dumping an image of the program's memory—called a **core file**—to disk, stopping the program's execution temporarily, and letting the program continue execution. A list of signals and the default action taken for each appears in the table on the next page. Many of the signals in the list correspond to exceptions in the x86 ISA, and are generated by the OS in response to the actual exceptions. As mentioned earlier, however, nothing prevents an entity capable of sending signals to a program from sending these signals.

The specific numeric values for each signal vary from operating system to operating system, and although they are technically standardized by POSIX, the standardization occurred much later than the varying definitions and use by compiled code, thus many systems retain their original numberings for compatibility with existing code.

The default behavior of most signals can be changed, either by explicitly forcing the program to ignore them, or by associating them with a handler function to be executed when the signal is delivered to the program. The exceptions are SIGKILL and SIGSTOP, which can neither be ignored nor caught, and always execute their default actions.

name	default action	description
SIGHUP	terminate	hangup detected on controlling terminal, or death of controlling process
SIGINT	terminate	interrupt (CTRL-C) from keyboard
SIGQUIT	dump core	quit (CTRL-\) from keyboard
SIGILL	dump core	illegal instruction
SIGTRAP	dump core	trace/breakpoint trap
SIGABRT	dump core	abort signal from `abort()`
SIGIOT		IOT trap; a synonym for SIGABRT
SIGBUS	dump core	bus error (bad memory access)
SIGFPE	dump core	floating point exception
SIGKILL	terminate (always)	kill signal
SIGUSR1	terminate	user-defined signal 1
SIGSEGV	dump core	invalid memory reference
SIGUSR2	terminate	user-defined signal 2
SIGPIPE	terminate	broken pipe: write to pipe with no readers
SIGALRM	terminate	timer signal from `alarm()`
SIGTERM	terminate	termination signal
SIGSTKFLT	terminate	stack fault on coprocessor
SIGCHLD	ignore	child stopped or terminated
SIGCLD		synonym for SIGCHLD
SIGCONT	continue	continue if stopped
SIGSTOP	stop (always)	stop process
SIGTSTP	stop	stop (CTRL-Z) from keyboard
SIGTTIN	stop	keyboard (tty) input for background process
SIGTTOU	stop	console (tty) output for background process
SIGURG	ignore	urgent condition on socket
SIGXCPU	dump core	CPU time limit exceeded
SIGXFSZ	dump core	file size limit exceeded
SIGVTALRM	terminate	virtual alarm clock
SIGPROF	terminate	profiling timer expired
SIGWINCH	ignore	window resize signal
SIGIO	terminate	I/O now possible
SIGPOLL		pollable event; synonym for SIGIO
SIGPWR	terminate	power failure
SIGSYS	dump core	bad system call

The `sigaction` in `/usr/include/bits/sigaction.h` defines signal behavior at user level, and appears *logically* as follows:

```
struct sigaction {
    void (*sa_handler) (int);
    void (*sa_sigaction) (int, siginfo_t*, void*);
    sigset_t sa_mask;
    int sa_flags;
};
```

An operating system maintains copies of this data internally and may include other information as well. In Linux 2.4, the internal structure is little more than a wrapper around the structure shown above (see `asm/signal.h`), but does put the `sa_mask` field at the end to allow it to be extended to include more signals in the future. The two handler functions are actually joined in a union, so only one should be used at a time. If only the signal number is to be delivered to the signal handler, the `sa_handler` field should be set. If additional information is to be passed, the `sa_sigaction` field should be set instead.

The sa_flags bit vector field in the sigaction structure is used to specify several options regarding signal behavior. Each signal has its own flag values, although not all flags affect all signals. The following flag values can be used:

SA_SIGINFO	send more than just the signal number to the handler when this signal is delivered
SA_RESETHAND	reset this signal's handler to the default when the signal is delivered
SA_NOMASK	do not mask this signal while its handler is executing, thereby allowing a second signal of the same type to interrupt the first
SA_RESTART	restart system calls if possible rather than interrupting them when this signal is delivered
SA_NOCLDSTOP	do not send a signal to this program when any of its children programs stop or continue; the parent/child relationship between processes will be discussed later in the course; this flag affects only SIGCHLD
SA_ONSTACK	use an alternate stack to hold the signal handler stack frame (see sigaltstack)

The sa_mask field in the sigaction structure specifies which signals should be masked when the signal handler specified by the structure executes. Sets of signals are represented as bit vectors in the sigset_t structure, which consists of an array of integers used to store the bits. The following macros are used to manipulate these bits:

```
int sigemptyset (sigset_t* set);
int sigfillset (sigset_t* set);
int sigaddset (sigset_t* set, int sig_num);
int sigdelset (sigset_t* set, int sig_num);
int sigismember (sigset_t* set, int sig_num);
```

The sigemptyset function assigns the empty set—the one containing no signals, which is represented as all zeroes—to the specified set. Similarly, the sigfillset function assigns the set of all signals to the specified set. The sigaddset and sigdelset functions add and remove individual signals from a set, respectively. These four functions return 0 on success and -1 if an error occurs. The last function, sigismember, checks whether a signal is in a set, returning 1 if such is the case and 0 if the signal is not in the set.

Control of Signal Behavior

Signal behavior is tracked and used by the operating system when delivering signals, and the default behaviors are set up by the operating system when a program starts. However, the program itself manages the signal behavior (for most signals) by making system calls to change the sigaction structures stored by the kernel. We discuss four calls used to control the behavior and delivery of signals:

```
int sigaction (int signum, const struct sigaction* act, struct sigaction* oldact);
int sigprocmask (int how, const sigset_t* set, sigset_t* oldset);
int sigpending (sigset_t* set);
int sigsuspend (const sigset_t* mask);
```

The first function—sigaction—serves as the main function for changing the behavior of signals. The arguments are a signal number, a pointer to a sigaction structure defining the behavior for that signal, and a pointer to a second structure into which the previous behavior is written. The function returns 0 on success and -1 on failure.

The function is defined so as to allow a caller to change aspects of signal behavior, such as a single flag value, without the need to specify all behavior. For example, if the first pointer is NULL, the behavior is not changed. If the second pointer is NULL, no copy of the old values is made. Two calls can thus be used to read the old value, modify it appropriately, and write it back into the operating system's structures.

Rather than specifying a handler for a signal, one can also specify the use of the default action using SIG_DFL (a symbolic constant) or specify that the signal should be ignored using SIG_IGN. A signal that is ignored by a program, whether by default or through a call to sigaction, is simply discarded: no handler calls are made, no bits are set to indicate that the signal existed.

The `sigprocmask` function allows a program to read and/or change its signal mask. Analogous to masked interrupts, a signal sent to a program but masked by the program is held by the operating system until the program unmasks the signal, at which point the signal is delivered immediately. The second and third arguments point to bit vector structures specifying one or more signals. The first argument specifies how the bit vector of masked signals for the program should be changed: SIG_BLOCK adds signals to the mask, SIG_UNBLOCK removes signals from the mask, and SIG_SETMASK replaces the old mask with the new one.

A signal that is sent to a program but is masked is said to be **pending** until it is unmasked and can be delivered. The pending signals for a program are tracked by the operating system, and this bit vector of signals can be read using the `sigpending` function. However, it is important to note that signals are rarely left pending while a program continues to execute unless those signals are masked, thus signals are not seen as pending unless they are masked.

The last call, `sigsuspend`, provides a mechanism for putting a program to sleep until a certain signal (or one of several signals) occurs. A bit vector of signals is passed to the function, and the program goes to sleep until one of these signals is received, at which point the signal is delivered and the function returns -1 (the usual return value when interrupted by a signal).

Signal Generation

Signals can be **generated** by user programs or inside the kernel itself. User programs generate signals through the `sys_kill` system call, which checks that the calling program has permission to generate signals for the target program (specified by pid). Permission is granted only if the caller is owned by the same user as the target, is in the same login session as the target (checked by comparison of login shell pids), or is owned by the machine's super-user.

Signal generation inside the kernel typically occurs in response to an interrupt or exception, but can also happen when another program performs a system call other than `sys_kill`. For example, one program may send data to another program using `sys_write`; if the receiving program has requested signals when data are ready on the file descriptor from which it will read those data, a signal is generated.

The functions used to generate signals inside the kernel are defined in `linux/sched.h`. Four functions are defined:

```
int send_sig_info (int sig_num, struct siginfo_t* info, task_t* target);
int send_sig (int sig_num, task_t* target, int privileged);
int force_sig_info (int sig_num, struct siginfo_t* info, task_t* target);
int force_sig (int sig_num, task_t* target);
```

The `send_sig_info` function is the main function for generating signals within Linux. The `info` argument points to a structure that contains additional information about the reason for the signal. If no additional information is available, the value 0 can be sent to indicate that the sender is a user process, or the value 1 can be sent to indicate that the kernel itself generated the signal. In the latter case, the permission checks described earlier are skipped—the kernel is always allowed to generate signals for any program. A similar permission-check override is also made when the `info` pointer is valid using the `si_info field` of the structure. When the kernel generates a signal, the `si_info` must be greater than zero; when a user program is the source of the signal (via a system call of one form or another), the field must hold a value less or equal to zero.

The `send_sig` function is a wrapper function used when no additional information is available to provide to the signal handler. This function simply calls `send_sig_info` using either 0 or 1 as the `info` argument, depending on the value of its own `privileged` argument (again 0 or 1).

The last two functions are used to **force** a program to receive a signal even if the program has requested that the signal be masked. In particular, the two functions forcibly change an ignored signal back to its default behavior, remove the signal from the set of masked signals, and then call `send_sig_info`. Note that the handler is changed only for signals that are being ignored; a program that provides a handler will receive the signal via that handler.

Under normal circumstances, the forced delivery functions should not be used. If a program chooses to mask out a particular signal, the programmer is likely to assume that the signal will not be delivered while it is masked, thus forcing delivery may introduce a bug into the program.

So when are the forcing functions useful? They are used primarily to deliver signals generated by exceptions. A program that generates an exception cannot make forward progress, since the processor does not know how to proceed. For example, the processor cannot simply keep executing a program that contains an illegal instruction, as the processor cannot execute the illegal instruction. In this case, the operating system can either terminate the program or make a last-ditch effort to get its attention, which is the purpose of the signal forcing functions.

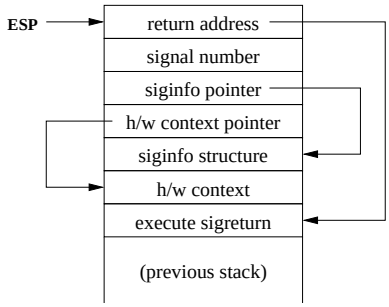
Once a signal of a given type has been generated for a given program, the kernel examines the program's signal behavior for that type. If the signal is being ignored, it is simply discarded. Otherwise, the signal is added to the bit vector of signals pending for that program, and the program is woken up if it is asleep (*e.g.*, waiting on some event).

Signal Delivery

Signals can be **delivered** to the program currently running on the processor whenever the system returns from an interrupt, exception, or system call. As these three paths represent the only methods for signal generation, checking for signals at other times is pointless. However, it is important to note that signals are delivered only to the program currently running; a program with pending signals is not given priority over another program for use of the processor just because of the pending signals. Signals may thus remain pending for some time while other programs execute.

When the kernel decides that a signal should be delivered, it examines the signal behavior for that signal and takes the appropriate action. If the default behavior has not been overridden, it is performed by the kernel at this point.

If a handler has been provided for the signal, the kernel sets up a stack frame for the signal handler before returning control to the program. The stack frame used when a `siginfo` structure is available is shown to the right. For traditional calls, the two extra pointer arguments and the `siginfo` structure are omitted.



The first thing to notice is that the return address points to a short piece of code written onto the stack itself. This code makes a system call, `sys_sigreturn`, that tears down the stack frame. Obviously, the kernel cannot trust a user program to make this call properly, but the normal path for a non-malicious program is to call back into the kernel. Even non-malicious user programs may override this behavior by backing out of failed code in a signal handler (using `longjmp`, for example).

The user program's hardware context, including all registers, is also copied from the kernel stack to the program's stack, and is copied back during the `sigreturn` system call. This copying serves two purposes. First, remember that signals are the analogue of interrupts, and a program running on a private machine might want to manipulate registers during an interrupt. A user-level thread package, for example, may want to switch to a different thread when a signal handler executes. To do so, it needs access to the machine state. The other reason for copying the hardware state to user space is to avoid having the kernel stack overflow due to malicious programs. A program may never actually call `sigreturn`, in which case a copy of the hardware context left on the kernel stack stays on the kernel stack forever. Given that the kernel stack is limited in size and not designed to tolerate overflows, giving a user program a method by which it can force an eventual overflow is not a good idea.

Assuming that the call to `sigreturn` does execute, the call copies the hardware context back onto the kernel stack and tears down the signal handler stack frame. When returning control to the user program, it then checks whether the program was in the middle of a system call or not. If not, the kernel simply returns control to the user program. For programs that were executing system calls when the signal was delivered, the kernel must decide whether or not to restart the system call based on the specified behavior for the signal. If system calls are not to be restarted, the system call is forced to return -EINTR to indicate interruption by a signal. To restart a system call, the kernel restores the value of EAX used for the system call and moves the program's PC back to the `INT 0x80` instruction (most specific instructions in an ISA have fixed size).

Terminology

You should be familiar with the following terms after reading this set of notes.

- signal abstractions
  - interrupt analogy
  - generation
  - forced signal
  - pending signal
  - queued signal
  - delivery
- signal behavior
  - default behavior
  - catch
  - block or mask
  - ignore
  - core file