Advanced Operating Systems

Article *in* Computer · November 1984

DOI: 10.1109/MC.1984.1658969 · Source: IEEE Xplore

3 authors, including:

Walter Tichy
Karlsruhe Institute of Technology
265 PUBLICATIONS **4,748** CITATIONS

SEE PROFILE

Advanced Operating Systems
Design, Implementation, and Real Time

Stanley A. Wileman, Jr.
University of Nebraska at Omaha

This is an incomplete work-in-progress. It is primarily intended for the students in Computer Science 8530 at the University of Nebraska at Omaha.

Table of Contents (Topics) — Not Nearly Final

Appendices

Index

Introduction 1

These notes are provided for those individuals who need to gain a more complete understanding of operating systems than is provided by a typical undergraduate course. The notes are currently a "work in progress" and suggestions to the author are welcome.

1.1 Background and Prerequisites

All undergraduate computer science curriculum recommendations include a course on operating systems. This required course is *not* one in which the user is taught how to enter commands, create and list directories, and install and maintain the operating system. These are certainly appropriate skills, but don't provide the understanding of operating systems expected by the designers of computer science curricula.

Instead, the undergraduate operating systems course that is an appropriate prerequisite for this material covers the purposes of an operating system, the variety of designs possible, and the traditional algorithms used in an operating system implementation. Two topics covered in some depth in that course are concurrency and mutual exclusion. These topics are not exclusively part of the operating system domain, but the undergraduate operating systems course is usually the first one in which students are exposed to these topics.

There is a wealth of information to be covered in the undergraduate course, and it is rare that students have many opportunities to engage in significant implementation tasks (except for small "toy" projects). Indeed, it is usually the case that there is insufficient time in the course to cover all of the background material.

1.2 The Approach

There are two parts to the material in these notes. Although they are presented sequentially, the material in the first part will prove beneficial later.

1.2.1 Design and Implementation of the Tempo Operating System

We will first examine in detail the design and implementation of an operating system called Tempo. This is not a "toy" operating system of the kind that is usually provided for use with undergraduate operating system textbooks (with some exceptions). Instead, it is a complete system that can boot and run on most typical x86-based personal computers.

There are numerous goals we hope to achieve by studying this system:

- **Better understanding of the purpose of system calls:** By examining the detail of how system resources and services are provided to application programs, we can better appreciate the facilities provided by those operating systems we typically use (e.g. Microsoft Windows and UNIX varieties).

- **Better understanding of design alternative and their effects:** There are often multiple approaches that can be taken to manage the resources of a computer system, each with positive and negative consequences. A real operating system will allow us to examine the effects of such design decisions.

- **Better understanding of existing operating systems:** By examining the design and implementation of Tempo, we will gain insights into how popular operating systems deal with similar issues.

- **Better understanding of concurrency and mutual exclusion:** Reading about concurrency and studying classic algorithms for synchronization is valuable, but seeing the implementation and use of these in a real operating system gives real understanding of them. We know that modern processors have almost reached the "legal" speed limit, and understand that only by utilizing multiple processes running on multiple processors will we achieve additional performance gains. Multiple processes concurrently or simultaneously working on the same data must be synchronized.

- **Better understanding of how applications can use system services:** Every application uses the services of the operating system, but those services are usually hidden below a layer of library functions. Application designers often overlook available facilities and resources because they mistakenly limit themselves to the features provided by a particular programming language. Even if only those language-provided facilities are used, understanding the system services their implementations use will improve the application designer's understanding of the cost of those facilities.

1.2.2 Introduction to Real-Time[1] Systems and Applications

The second part of these notes provides introductory material on real-time systems and programming. Real-time systems have two parts: a real-time operating system and one or more real-time applications. It is common for such systems to be dedicated to running those applications (as opposed to attempting to be general-purpose, or all things to all applications).

If you could successfully enumerate all the computer systems and applications in use at any instant, you would find that perhaps as much as ninety percent of those systems and applications could be characterized using the term real-time. That's right; although there are many PCs out there, they represent only a small fraction of the computer systems in use. And most computer scientists (and even a larger fraction of computer users) don't have even a small understanding of real-time systems and programming.

This part will introduce the basics of real-time programming, focusing on the system services needed to ensure timely delivery of results. Several classic algorithms for real-time scheduling and dealing with priority inversion (an undesirable situation that can occur unless carefully avoided) are examined. Operating system design decisions that affect the capability of the system to support real-time applications are also considered.

[1] There is apparently no consensus on a canonical presentation of this term. Various other ways of writing it are "realtime," and "real time" (two words). It appears to be most frequently written as shown here, with a hyphen.

The Tempo Operating System 2

2.1 History

The name "Tempo", the initial definition of system calls, and the first implementation were associated with the textbook **An Introduction to Real-Time Systems: From Design to Networking with C/C++** by Buhr and Bailey (Prentice-Hall, 1999). Although usable, this effort depended on MS-DOS for loading, input/output services, and first-level interrupt handling. Since it used MS-DOS, it follows that it was a 16-bit system and used none of the features provided by modern x86 processors.

The system was also primarily written using C++, but only a limited subset of the features of that language. Modern operating systems (e.g. Microsoft Windows and virtually every "flavor" of UNIX, including Linux) are written using C and assembly language. While object-oriented paradigms are important in operating systems (as we shall see), the C++ language features often result in the use of significant amounts of "hidden" code with unanticipated increases in running time.

2.1.1 Tempo/C

The first version of Tempo (apart from that used with the textbook) was developed in early 2000. The following list summarizes the major changes made in the system.

- **Replacement of all C++ code:** All of the C++ code in the original system was removed and replaced by C and assembler language (using the Intel/Microsoft syntax). This improved the usability of the system for education purposes, since it no longer had reliance on the Borland 16-bit C++ compiler, but instead could use the more commonly available Microsoft products. To enable clear distinction between the old C++ system and the new system written in C, its name was changed to Tempo/C.

- **System call redefinition:** The C++ version of the system took advantage of the ability to provide default values for certain parameters omitted from system calls. In particular, some system calls have the ability to wait either an arbitrary (potentially infinite) amount of time for completion, or for a user-specified maximum amount of time. If the timeout parameter was not supplied, the assumption was made that the caller was willing to wait for completion whenever it occurred. It was also the case that some system calls that could benefit from having timeouts were not provided with them. The definition of these system calls were rewritten to require explicit specification of the timeout parameter in all cases, and to add timeouts to those calls that could use them but did not have them.

- **Keyboard input:** The original system had no provision for input from a keyboard, so a system call was added to provide that capability.

- **Examples:** Many applications were developed to demonstrate the system calls provided by the system.

- **Distribution:** The assembler and C source code for the system and the examples were packaged together for ease of distribution. Those distributions used in the classroom were packaged with the appropriate Microsoft tools (for which licenses were available).

2.1.2 Tempo/32

During the summer and fall of 2000, Tempo/C was completely rewritten and renamed Tempo/32. As you can infer from the name change, the system was modified to utilize the 32-bit features of modern x86 processors, including protected mode. Here is a summary of the major changes:

- **32-bit protected mode:** Except for the bootstrap loader (see below), the system now uses the 32-bit instructions and registers in the x86 processor. Additionally, it makes use of the protected mode features of the processors to provide two execution modes: kernel and user. This is consistent with the approach taken by "real" operating systems, although it does make the system more complicated than "toy" systems.

- **Stand-alone booting:** Instead of having to be started as an application from the MS-DOS operating system, a floppy-disk bootstrap loader was added to Tempo. This was the only place where 16-bit code was used (by necessity), since all x86 processors begin execution after power-up or processor reset in "real" mode (which assumes all operands of instructions are either 8 or 16 bits long, and physical addresses are limited to 20 bits).

- **Flat address space:** All x86 memory addresses (in the processor) have two components: a segment and an offset within the segment. 16-bit real-mode programs are limited to 20 bit addresses that are computed as 16 × segment-register-value + offset, which yields an address that is 20-bits long[2]. Since at most 64KB or memory can be accessed with one segment register value, programs that access large regions of memory are somewhat cumbersome, requiring frequent changes to the segment register values. In protected mode, the x86 processors provide more advanced segment facilities in which segments can have almost arbitrarily large sizes. This feature allows us to make the segmentation almost transparent, by arranging for all segments to begin with physical address 0 and have a size of 4GB. Now code can use 32-bit linear addresses without undue concern about the segment register setting.

- **Limited use of x86 paging hardware:** Tempo/32 also utilizes another feature found only in protected mode — the paging hardware. Page tables (and directories, more later) enable non-contiguous regions of physical memory, called pages, to appear contiguous to an application program. Individual pages can be marked as present or not present, and can be protected (read-only or read-write), so errant program activity can be trapped. Tempo/32 utilized the paging facilities, but not extensively, primarily as a demonstration of the hardware features of the x86 processors.

- **Microsoft development tools replaced:** The Microsoft MS-DOS system, C compiler, assembler, and linker were replaced by the GNU gcc C compiler, assembler, linker, and

[2] There are a few cases where this value can be longer than 20 bits. For example, if the segment register contains `0xfff0` and the offset is `0xfff0`, the resulting address is `0x10fef0`. We will touch on this again when we talk about the *A20 address line*.

make utility (and a few others). All of the development tools and source code for the system were thus freely available to anyone, with no licensed software required. All of the scripts for building the system and running the examples were rewritten to use these tools.

- **Cygwin used in the Windows environment:** Since Microsoft Windows is ubiquitous in many educational environments, Cygwin was employed to provide the necessary development environment on Windows platforms. Cygwin is a popular freely-available system which runs on Microsoft Windows and provides a near-perfect emulation of a UNIX development environment and many of the popular UNIX/Linux development tools.

- **Bochs used for emulated execution:** Bochs is an open source x86 emulator that runs on Microsoft Windows and UNIX/Linux systems[3]. It faithfully emulates the processor and most of the common devices attached to a modern PC system. Although Tempo can be booted and executed directly on a stand-alone system with no additional software support (except the system's BIOS, which we'll cover later), doing so is somewhat cumbersome when doing development. Rebooting a system is usually a time-consuming activity, especially since the physical media (a floppy disk) must be created on the development system, then inserted in the drive on the test system, which is then reset. Bochs can boot directly from a file containing an image of a floppy disk (which the Tempo development tools build), requiring substantially less time than if a real machine was used. Appropriate shell scripts and configuration for the use of Bochs are included with the Tempo/32 distributions.

2.1.3 Tempo/32 in Fall 2005

During the fall of 2005, additional features were added to Tempo/32.

- **Simple hard-disk input/output:** Hard disk drives are common on almost all PC systems, so it was logical to add a device driver and simple system calls to allow blocks on a hard disk to be read and written. That Bochs also emulates the disk controller and drive made this a reasonable extension to Tempo/32.

- **Existing system calls restructured:** As with most large development projects, some earlier work on system calls needed to be revisited. In particular, some of the original Tempo system calls had flaws (which will be discussed later). Documenting the system calls and guaranteeing that their implementation is consistent with their definition is necessary for any system, Tempo included.

- **New POSIX-like I/O, command-line arguments, and system calls for process termination and awaiting process termination added:** POSIX is a (set of) standards for operating system services, including I/O system calls. The Tempo I/O facilities (still one of the least highly structured system components) were modified so they looked more like the POSIX definitions. The ability to pass arguments to the main function of a program in the same manner as that done in UNIX systems was added. Two new system calls, `kill` and

[3] Bochs is also capable of running on Apple Macintosh systems, and Tempo systems can be executed using Bochs in that environment. Unfortunately, the other Apple development tools do not appear to be completely usable for Tempo development, although this issue is being investigated.

`waitproc`, were added to allow forcible termination of a process and to wait for a process to terminate.

- **Code reorganization:** The Tempo/32 source code was reorganized to facilitate reading and modification.

2.1.4 Tempo/32 in Spring 2006

Some substantial additions were made to Tempo/32 in the spring of 2005:

- **File system code enhanced:** The simple file system developed with the hard disk device driver and system calls was enhanced. The features it provides are similar in many ways to those available in the traditional UNIX file system.

- **Assembler and "C-minus" compiler ported to Tempo/32:** A few tools developed for another operating system project included a simple x86 assembler and a compiler for a subset of C, appropriately named C-minus. These tools were ported to Tempo/32.

- **New `run` system call provided:** To accompany the new assembler and compiler, a new system call was added to allow complete programs to be loaded from the filesystem on the hard disk.

- **Dynamic allocation of pages to processes:** Since the "new-style" processes (loaded from disk) are not "compiled-in" (as were all previous "processes"), it became necessary to do dynamic allocation of primary memory for these. Structures to keep track of page usage, and to allocate and free physical pages of memory were added. Additional attention was required to the system code executed when a process was terminated, since the pages allocated for such processes had to be reclaimed.

2.1.5 Tempo/32 in Summer and Fall 2006

Work during this period focused on improving the portability of the system and its usability in multiple development environments.

- **Additional work on loaded processes:** The C-minus development tools, although usable, were far from perfect. Instead, attention was turned to loading executables in different formats — in particular, ELF (executable and loadable format) and PE (portable executable). The GNU tools on a UNIX/Linux platform commonly produce ELF files, and those tools when used with Cygwin produce the PE executables.

- **Dynamic stack expansion for loaded processes:** the size of the stack for a "compiled-in" process is fixed at the time the process is created, and there is no way it can be enlarged. Processes loaded from disk do not explicitly specify their stack size, and are given one page of stack space by default. This is insufficient for some processes, and so some mechanism was needed to dynamically cause the stack size to grow. The trigger for such growth is a page fault, and code was added to the page fault handler to detect those faults caused by stack overflow and then automatically add a page of memory to the stack for the process.

2.1.6 Current (2007) and Future Directions

A number of applications have been written for or ported to Tempo, including a full-screen text editor, a BASIC interpreter, a Forth interpreter, and several simple shells. Current activity includes reorganizing the input/output system, writing NIC (network interface card) drivers (the National Semiconductor DP83815 chip driver is functional, and work is proceeding on a driver for the Intel 8255x family of chips; unfortunately NICs aren't as standardized as disk controllers and the other support chips in PCs), and augmenting the standard C library for application use. Once a few NIC drivers are stable, the next major project will be to port the TCP/IP implementation from the XINU operating system to Tempo and add support for SMB-based remote file systems. Remote and wake-on-LAN booting and porting of the FAT filesystem code from XINU are on the visible horizon. The number of future development activities possible with the Tempo system is almost limitless.

2.2 What is Tempo?

From the discussion of Tempo's history you may have already formed some ideas about what the system looks like. But let's look at some specific characteristics.

- **Tempo is a stand-alone system.** It does not depend on any other software to boot or run, except a standard PC system's BIOS (Basic Input/Output System). This is good, since it requires us to understand everything that an operating system needs in order to run; and from an educational perspective, this is appropriate. It is also somewhat problematic, since if we want a feature or function, we must provide it ourselves.

- **Tempo is a monolithic, non-preemptible kernel.** Monolithic means that all components of the operating system kernel are linked together at the time the system is built. A feature found in many modern systems is the capability to dynamically load some software after the system has been booted. In particular, device drivers can be dynamically loaded and unloaded, which is particularly useful for USB devices. But Tempo doesn't do that — yet. Non-preemptible means that when a process is executing code inside the operating system kernel, it cannot be preempted by the occurrence of an interrupt. For example, this means that if the programmable interval timer "ticks" while we are executing inside the kernel, this "tick" will not be recognized until we leave the kernel and return to a user application (or the idle process). This means that Tempo may be slow to recognize interrupts. It is worth noting that until recently most operating systems (except those specifically designed for real-time purposes) were designed as monolithic kernels. For one thing, they're easier to design and implement.

- **Tempo provides two different kinds of "processes.**" The first, and original, type of "process" provided by Tempo is linked into the system when it is built. The code and data for such processes is always part of the system, even when they aren't needed. In addition, care must be taken to ensure that the external names used by these "processes" do not clash with those external names used by the kernel, since they share the same namespace during the linking process. This type of process is closer to the objects known as "kernel threads." The second type of process is more traditional. Its executable code and data are prepared separately from the kernel of the system, stored on a secondary storage device (like a disk drive), and is allocated storage and loaded from disk only on demand (using the `run`

system call). Tempo supports executable files in several formats, including ELF, PE, and the unique format used by the C-minus compiler.

- **Tempo uses priority scheduling.** The "golden rule" of priority scheduling is that at any instant, the process that is running is the highest-priority ready-to-run process in the system (or is one of a set of such highest-priority processes). Of course the non-preemptible character of the Tempo kernel means that should a process with a higher priority logically become ready to execute while a lower priority process is executing a system call, then we will have to wait until that system call completes (or the process blocks) before the higher priority process is given a chance to execute.

- **Tempo uses Round-Robin or Run-To-Completion scheduling.** Depending on current settings, processes with the same priority will take turns using the CPU, each running for at most a quantum of time before relinquishing the CPU to other ready processes at the same priority. Alternately, the system will allow a process to use as much CPU time as it wants (subject to the priority "golden rule"), relinquishing the CPU only when it terminates execution, faults, blocks, or explicitly yields the processor (using the `yield` system call) to another process at the same priority.

- **Tempo has traditional counting semaphores with timeouts.** Semaphores are the most well-known process synchronization and interprocess communication (IPC) tool. Tempo provides a pool of semaphores and system calls for allocating and initializing a semaphore (`newsema`), freeing a semaphore when its use is no longer required (`freesema`), and performing P and V (`down` and `up`) operations. If a semaphore's count is 0 at the time a `down` operation is performed, the system allows the process to block indefinitely, never block (to do polling), or block for at most a specified maximum period of time.

- **Tempo has signals and messages.** Each process has a fixed number of queues to which signals or messages may be directed. Tempo's signals and messages are synchronous, unlike those provided by UNIX. This means that a process will not be interrupted by the arrival of a signal or message, but must explicitly check for the presence of a signal or message. A signal is similar to those provided by UNIX in that it provides a numbered, anonymous notification of an event. Tempo counts signals that have been delivered but not yet received by a process, unlike UNIX[4]. A message is signed by its sender (that is, the sender's process identification is attached to the message), and includes a 32-bit message that, in some cases, can effectively be used as a pointer to a larger data structure. Messages are queued in FIFO order on each process' signal/message queues. An individual queue may be masked (or unmasked) to prevent (or allow) recognition of signals and messages on that queue. Messages and signals are recognized in order from the lowest numbered queue to the largest, effectively providing them with priorities.

- **Tempo provides separation of user and kernel execution.** The Intel x86 architecture provides four privilege levels, numbered 0 to 3. Most operating systems, Tempo included, use only two of these, 0 and 3. Level 0, the most privileged, is used by programs executing inside the kernel. Programs executing outside the kernel use level 3, the least privileged.

[4] We shall have much more to say about signals, in particular POSIX signals, when we consider real-time systems.

Certain instructions (e.g. input/output) cannot be successfully executed when executing at level 3, and only the kernel has the capability to access all primary memory.

- **Tempo provides POSIX-like I/O facilities.** The system calls for performing input/output are similar to those specified by the POSIX standards:
  - `open(path,mode)` — open an existing file
  - `creat(path)` — create a new file
  - `mkdir(path)` — create a directory
  - `close(fd)` — close an open file
  - `read(fd,buffer,size)` — read from an open file
  - `write(fd,buffer,size)` — write to an open file
  - `seek(fd,offset,origin)` — set the position that will next be read or written
  - `remove(path)` — delete a file
- **Tempo development is done using open-source tools.**
  - gcc — GNU c compiler
  - ld — GNU linker
  - make — build utility
  - objcopy — copy and translate object files between formats
  - dd — raw file copy (for disk images)
  - Cygwin — UNIX environment for Microsoft Windows
  - Bochs — PC hardware system emulation for Microsoft Windows, Linux, BSD, etc.
  - Shell scripts for Tempo system compiling, linking, disk image creation, etc.

Processes 3

Processes are used to represent the executable objects in a system. The operating system itself is not a process, although there may be one or more processes created to support the actions of the operating system (primarily for convenience).

In Tempo, there are two different kinds of executable objects. Historically (at least in Tempo's history) a process was (or is) a separate thread of execution with its code and data linked together with that of the operating system kernel and device drivers[5]. In other systems these might be appropriately called "kernel threads" since they run in the address space of the kernel. Since they are linked together with the kernel, care must be taken to avoid defining external symbols that are the same as those in the kernel, since there is only a single namespace.

The second kind of executable object is appropriately called a process, since it is not statically linked with the kernel, but is loaded (in Tempo, from a disk) into primary memory allocated for it only at the time it is executed. As just mentioned, the code and data for the historic processes in Tempo is always present in the system, occupying primary (and virtual) memory, even though they may not currently be candidates for execution.

3.1 Process Objects

Each Tempo process, regardless of its type, is represented in the system as an entry of type **`struct Proc`** (h/types.h)[6] in the array named **`proctab`** (kernel/sys.c). The number of entries in this array is a constant named **`NPROC`** (h/sysparm.h). That is, the number of processes that can concurrently exist in the system can be no larger than `NPROC`. This is a classic example of static allocation in an operating system. At least as recently as the 2.2 kernel, Linux[7] also employed static allocation for the array holding these *process descriptors*. Only the objects identified in this array are capable of being selected by the operating system for execution.

`struct Proc` entries (which are alternately called process table entries or process descriptors) that do not currently represent a process are identified by the value **`PROCESS_FREE`** (h/sysparm.h) in the `state` field of the structure. When it is necessary to create a new process, one of these unused entries must be allocated. If no free entries are available, the process creation, of necessity, must fail. As we shall see later, when Tempo needs an unused process table entry, it performs a linear search through the array for the first available unused entry. Here is another example of a design decision in an operating system. We could, if desired, keep a list of unused entries, and when one is needed, we could simply "pluck" it from the head of the list (assuming the list wasn't empty). This would certainly make things faster at the cost of complicating the code and making it more difficult to understand. The linear search

[5] The code and data for "normal" processes — not threads — could also be provided in the same file as the kernel. That is, the distinction between a kernel thread and a process is not the source of its code and data.
[6] When we reference the Tempo source code, we will usually identify the file being referenced in this manner.
[7] See, for example, **Understanding The Linux Kernel** by Bovet and Cesati (O'Reilly, 2001).

decision was made to favor simplicity and readability (and the probability of correctness). And how much faster would the process descriptor allocation be with the linked list? The current size of **NPROC** is 20, and with a pedagogic system like Tempo, we are unlikely to be using many of those entries at the same time.

3.1.1 Process States

As we just seen, there is a **state** field in every process table entry. The content of the field, as expected, describes the state of the process. Conceptually, every process is either ready to be executed ("ready" or **PROCESS_READY** in Tempo), currently being executed ("running" or **PROCESS_CURRENT** in Tempo), or is incapable of being executed because it is waiting for some resource (e.g. completion of an input operation, a semaphore, a message, or the expiration of a timer). Processes in this last state are "blocked." The complete list of process states in Tempo can be found in file h/sysparm.h.

Some operating systems may choose to use a single value of a field like `state` to identify blocked processes, with a separate field giving the reason the process is blocked. In Tempo there are multiple blocked states, one for each reason a process might be blocked. The difference between these approaches is illustrated in the diagrams below.

Figure 1. Three-state process transition diagram

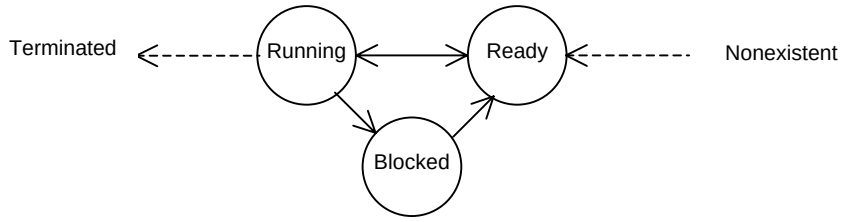
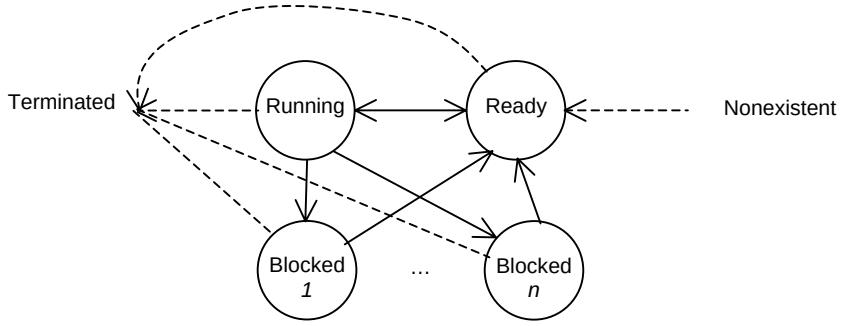


Figure 2. Tempo's process transition diagram



There are several observations we should make about the differences in these figures. Although conceptually they appear to have significant differences, this is not really the case. If we group all the blocked states in Tempo, we have the same set of states shown in the first figure (where a separate field in the process table entry is used to record the reason a process is blocked). There

is, however, another difference, and this is the presence in Tempo of transitions from any state directly to the terminated state. In UNIX systems, for example, a process that is not actually running cannot be terminated! We will discuss this later when UNIX/POSIX signals are considered in detail.

We note that the "terminated" and "nonexistent" conditions are really the same in Tempo, corresponding to a **`state`** field value of **`PROCESS_FREE`**, as noted earlier. In UNIX systems there is a state between terminated and nonexistent; processes in this transitory state are said to be "zombie" processes, a condition we will revisit again.

3.1.2 Process Priorities

Another field of the process table entry is **`priority`** (h/types.h). As you might expect, this field holds the current priority associated with the process, which may be between **`LOWEST_PRIORITY`** and **`HIGHEST_PRIORITY`** (both defined in h/sysparm.h). There is actually another allowed priority, which is one smaller than LOWEST_PRIORITY. This priority is used only for the idle process, which runs only when all other processes are blocked. By guaranteeing that the idle process priority is always smaller than any legal priority for "normal" processes, we guarantee that the idle process will never be executed if any other process is capable of being executed.

3.1.3 Process Queues

Each process may be found on one or more queues. Some queues are formed by linking together process table entries using the **`prev`** and **`next`** fields. Another important data structure that contains lists of processes is the array of queues named **`readyList`** (kernel/sys.c). This array has one entry (the head of a queue) for every possible process priority (including the priority for the idle process). The list of processes on one of these queues is the set of ready processes (that is, processes in the **`PROCESS_READY`** state). When the kernel needs to obtain a ready process to execute, it invokes the getready function (kernel/sys.c) that searches through the **`readyList`** queues, from highest priority to lowest priority, for the first non-empty queue. If such a queue is located, then the process at the head of that queue is removed, and a pointer to the process table entry for that process is returned.

Additional queues containing processes may be present, and we will examine them when the various system call implementations are considered.

3.1.4 Process Identification

Within the kernel itself, processes are most frequently identified by a pointer to their process table entry. This is, of course, just a memory address, and is certainly inappropriate for user processes to use when then need to make reference to a particular process. For this purpose, most operating systems use some sort of generic "process ID". In UNIX systems, for example, a process ID is just a small integer, which may just be the subscript of the process descriptor in the process table. Tempo just assigns a unique sequential integer to every new process. Since this is a 32-bit integer, this means that if the system runs long enough without being rebooted, and enough processes are created, that ID will wrap back around to 0. The function **`pid2a`** (kernel/sys.c), which takes a process ID as an argument, will translate a process ID to the corresponding pointer to the process table entry, if it exists, or **`NULL`** otherwise.

3.1.5 Other Process Components

As we shall see later, a process has two separate stacks associated with it, one for use when it is running in user mode, and one for use when it is executing code within the kernel.[8] Additionally, every process has values stored in various processor registers when it is executing. Since a process executing in user mode can not predict when it will be interrupted by an external event (such as an input/output operation completion, or a timer expiration), it cannot anticipate when the contents of these registers must be saved so another process can use the CPU's registers; that is the operating system's responsibility. A convenient place to save the contents of the registers is on the stack, and that is exactly what happens. After processing the interrupt, the operating system kernel will restore the register contents for the interrupted process and return to exactly the point in the code at which the interrupt occurred.[9] It is, however, necessary to retain the address of the user's stack in the process table entry, since the kernel must have some way to locate the user's stack in order to restore the registers!

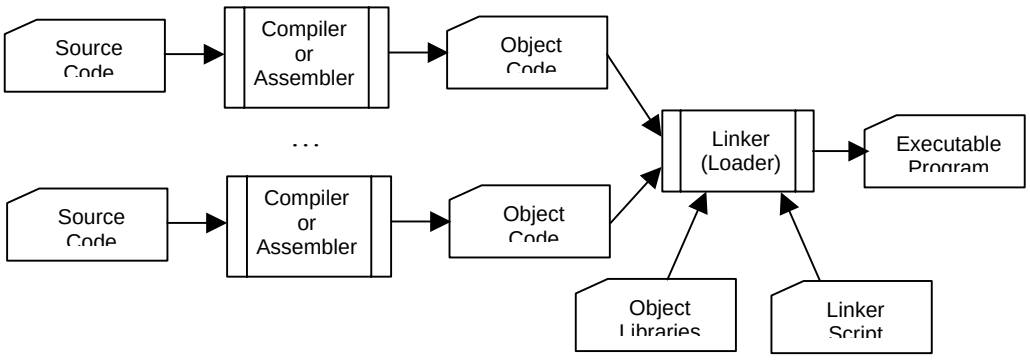
3.2 Program Preparation

We are going to divert our attention momentarily from issues unique to the operating system to discuss (in this section) some of the "mechanical" aspects of the C and assembler code used in operating systems (and other applications), and (in the next section) some details of how C functions use the stack. The observations made here are true for all types of programs, not just operating systems. However it is critical for operating system designers to understand these topics in more detail that the typical application programmer.

3.2.1 Review of Program Preparation

Figure 3 illustrates the basic steps used in preparing any program for execution, regardless of the programming language or operating system being used. This does not, however, apply to interpreted code such as that written in PERL or PHP (although it does apply to the interpreter).

Figure 3. Program Preparation for Execution



[8] A third stack, not unique to the process, is used during process termination activities.
[9] For most processors, an interrupt is always recognized between instructions.

The essential steps in program preparation are as follows:

- **Prepare source program(s).** One or more source programs are prepared (for example, source programs written in C). These are stored in appropriately named files. For some languages, additional related source files may be required, such as header files. Although these are not shown in Figure 3, their existence is assumed.

- **Compile or assemble the source program(s).** The appropriate compiler (and preprocessor) or assembler is used to process the program and produce *object code* that is also stored in a file (an *object code file*, sometimes called an *object module*). Errors (and possibly warnings) detected during this step must be corrected (in the source and/or header files) and the compilation or assembly repeated.

- **Link the object code files.** The linker is invoked to combine the object code files into a single executable file. The linker may also use libraries of previously prepared object code, either implicitly or explicitly specified, in resolving *external references*.

3.2.2 The Linking Process

We will not discuss compilation further in these notes, but we must pay careful attention to the actions taken by the linker. This development tool is sometimes called a loader (indeed, the GNU linker's name is "`ld`"), but it does not perform loading — that is the responsibility of the operating system.

A linker has several tasks to perform:

- **Identify definitions of and references to external identifiers, and resolve the references to the corresponding definitions.** Let's consider an example. When you write a program that uses the `qsort` function, you usually do not actually provide the source code for qsort. Instead, you expect the previously prepared and compiled version of qsort, stored somewhere in a object code library, will be retrieved and used in conjunction with your program. But when your source program is compiled, the compiler does not know where in memory the qsort function's entry point will be located at execution time. Only the linker knows this, and so it is the linker's responsibility to put the appropriate memory address in the instruction from your program that invokes qsort. The linker must do this for all external references, keeping track of the definitions in a symbol table. Some Tempo systems have more than 700 externally-defined symbols.[10]

- **Recognize the different sections in each object module and decide where they will be placed in memory at execution time.** A typical object module will include several sections. The "text" section contains executable code (and sometimes read-only constants; ELF and PE format object files use the "rodata" — read-only data — section for this purpose). The "data" section contains static data (variables declared with the static keyword

[10] Many modern operating systems support dynamic linking and loading of code modules at execution time. That is, the actual resolution of an external reference does not take place until the program is executed. This idea, which also allows multiple programs to share one copy of commonly-used code (like qsort), was actually used in much earlier systems (e.g. Multics).

in C, or many variables declared outside a function). The "bss" and COMMON sections are typically used for variables that are shared between object modules. The linker, guided by the linker script, will usually collect all of the text sections and group them so they will be placed in one region of memory at execution time, then do the same for the remaining sections.

- **Adjust addresses in code and data objects that are marked relocatable.** Compilers and assemblers may occasionally need to generate instructions and data items that contain the addresses of other objects. As noted above, if these objects are defined in other object modules, an external reference can be used to provide the address of the object, since it has an externally-defined name. But if it an object at a location in the same object module, it may not have an explicit name, so the compiler or assembler generates an address relative to the beginning of the section in which the object appears. These addresses must be altered by the linker so that refer to the actual address at which the object will appear when it is being executed. This process is called relocation.[11]

- **Write an executable file containing the resulting combined sections of code and data, with all relocation completed and all external references resolved.** This file will usually contain information organized in the same manner as an object module, except it will need no further relocation and will have no unresolved external references. Of course it is possible that relocation was unsuccessful or some of the externally-referenced symbols could not be resolved. For example, suppose your program mistakenly made a reference to a function named qsrt instead of qsort. The compiler or assembler could not detect this error, but the linker certainly can (unless the system being used actually has such a function defined in an object code library used by the linker).[12]

3.2.3 Execution

Assuming we now have a object code module marked executable by the linker, we must consider how it is actually executed. There are several steps required.

- **Allocate and initialize a process table entry for the new process.** As you know, a process is used to record many of the dynamic aspects of a program in execution. The executable object code file represents the static program – when it is not being executed. When you type the name of a program on a command line in a UNIX system, or click on the icon representing an executable program in a graphical user interface, the command interpreter will request execution of the program. This requires that creation of a new process associated with the program.

[11] Many modern processors provide instructions that can utilize code-relative addresses, eliminating the need for some relocation. For example, a looping structure will usually require a (possibly conditional) branch instruction at the end of the loop back to its beginning instruction. A branch to a negative offset relative to the program counter can often be used for that instruction.

[12] Many linkers also have the capability of producing partially-linked files that require further linking and/or relocation before they can be executed. This is useful, for example, if you have a large number of functions that are reasonably static – they're robust and need infrequent modification – that you use in a number of other programs. The object code modules for these functions can be partially linked, saving time when linking them repeatedly with other object code modules.

- **Allocate appropriate memory resources for the program.** Every process has regions of physical memory for code, data, stack, bss, and its other sections. Some of these regions may be shared with other processes (such as the text region, when multiple processes are executing the same program), but each process must have private regions of memory for its stacks. If sufficient memory is not available, then the program cannot be executed.

- **Load the executable program into primary memory, if necessary.** Unless the code and data is already present in memory (because another process was already executing the program), the various sections of the executable object code file are copied into the memory regions allocated for the process. Some regions of memory (in particular, the bss and COMMON sections) are expected (by the compiler and the programmer) to contain zeroes at the beginning of execution, and the operating system will also perform this task at this time.[13]

- **Prepare the initial contents of the kernel and/or user-mode stacks.** In most systems, programs begin execution assuming that environment information and information from the command line that invoked the program will be found at well-defined locations on the stack. The initial contents of processor registers will also be obtained from the stack. We will have more to say about this step later.

- **Mark the process as ready for execution.** Once the process table entry has been initialized, memory and stack space has been allocated, the code and data has been loaded, and the initial stack contents have been prepared, the process is capable of being selected for execution by the operating system.

3.3 Stacks

3.3.1 Stack Usage

Understanding how the stack is used during program execution is necessary if we are to understand process implementation. A stack is used for several purposes in a typical program:

- **A stack is used to store a function's local variables.** The variables declared inside a C function, unless explicitly declared as **`static`** or **`extern`**, will have the **`auto`** storage class. Programmers almost never write the auto keyword explicitly because, as its name implies, it is the default storage class for local variables. The storage for these variables is allocated on a function's stack when the function (or block) containing their declarations is entered, and that storage is reclaimed when the function or block is exited. This behavior is common to most modern programming languages like C, C++, and Java.

- **A stack is used for temporary variables.** Any processor has a limited number of registers that can be used to hold the results of computations. RISC processors typically have many more registers than CISC processors, but they are still a limited resource. When sufficient

[13] Many operating systems support execution of programs that are not completely loaded into memory. For a program to execute, all it requires is that the current instruction and its operands be present in memory. Such systems must have the capability of dynamically allocating memory and loading additional portions of the executable program and data when they are actually referenced. Tempo does not yet support this facility.

numbers of registers are not available to complete a computation, temporary storage elsewhere must be used, and that storage is usually on the stack.

- **A stack is used to store arguments to functions and the return addresses from functions.** The code generated for a function invocation by compilers for most modern programming languages will push each of the function's arguments on the stack, then execute a call instruction to the function. That call instruction will push the address of the instruction after the call on the stack. During the function execution it accesses function arguments at known locations on the stack (relative to the top of stack address when the function was entered), and on completion, it returns to the address pushed on the stack by the call instruction. Functions typically leave results that don't require much storage in processor registers.

- **A stack is used to save the contents of processor registers during interrupt processing.** Interrupts are the mechanism used to report significant or exceptional events. Typical sources of interrupts are protection violations (e.g. trying to write into memory that is marked read-only), computational errors (division by zero), reports of events considered significant by devices external to the processor (a disk read operation completes, or a timer "ticks"), or an explicit interrupt generated by a program to request system services. Many interrupts are generated asynchronously – that is, without regard to what the processor is doing at the time the interrupt is generated, but others may occur synchronously with the processor's execution of an instruction. For example, the interrupt signaling division by zero will naturally occur only as a result of and at the time of a divide instruction's execution. Regardless of the source of the interrupt, when the processor recognizes it, it will save the contents of all or some of its registers on the stack. The details differ between processor families, but this behavior is typical.

3.3.2 Processor Stack Implementation

Space for a stack (both its current contents and space for its growth) is reserved by the operating system before a process begins execution. In some operating systems, the space reserved for a stack may be increased if it outgrows the allocated region. Other systems may allocate a fixed amount of space for a stack that cannot be increased during the process execution. Tempo exhibits both of these behaviors (for the two different kinds of "processes").

It is typical for a processor to have a register called the *stack pointer* (the **esp** register[14] in the Intel x86 processors). This register contains the address of the memory location holding the data item at the top of the stack. If the stack grows downward in memory (as it does in the Intel x86 processors), a push instruction will decrement the stack pointer so it points to the new top of stack location, and store the item being pushed at that location. A pop instruction will copy the value at the top of the stack to a processor register and increment the stack pointer's value.

The operating system will set the stack pointer appropriately at the beginning of process execution. The stack pointer setting for a process will not be modified when the operating system handles interrupts or performs services on behalf of other processes. But managing the stack, and thus the stack pointer, is the responsibility of the process itself.

[14] The esp register is the name used for the 32-bit stack pointer. When an x86 is in 16-bit mode, the stack pointer is called the **sp** register.

3.3.3 Intel x86 Assembly Language Basics

Most of the code for modern operating systems is written in C. There are exceptions, of course, but UNIX and its variants (e.g. Linux, Solaris, BSD) and Microsoft Windows use C for the majority of the code. Some components of the operating system must, however, be written in assembly (or assembler) language, and this is also true of Tempo. Application programs never use some types of instructions, so compilers for high-level languages usually have no statements that can generate those instructions.

In order to be able to understand the assembly language code in Tempo we must spend just a little time exploring the Intel x86 assembly language.

Unfortunately, there is no canonical definition of assembly language for the x86. There are two major "flavors" of assembly language in popular use – the so-called Intel/Microsoft syntax and the AT&T syntax. To avoid confusion, we will only consider the syntax used by Tempo, which is the AT&T syntax. This is also the variant used by the assembly language source components of Linux and by the GNU assembler that is used to build Linux and Tempo.

As usual in assembly language, most instructions have an optional label, a mnemonic for a machine language instruction (e.g. **`add`**), and register or memory operand specifications (most instructions have two). There are also *pseudo-operations* (sometimes called *pseudos*, *pseudo-ops*, or *directives*) that do not correspond to machine language instructions, but are used instead to instruct the assembler to do things like reserve memory, define initialized memory locations, or mark a symbol as an external definition. Comments may appear by themselves on separate lines or be added at the end of instructions.

Operands for machine language instructions are *usually* written with the source operand first and the destination operand second[15]. The destination operand identifies the register or memory that will usually be modified by execution of the instruction. For example, the statement

```
movl    %eax,total
```

will be translated by the assembler to an instruction that will "move" (copy) the contents of the processor's **`eax`** register to the four bytes of memory at the location identified with the symbol "**`total`**".

Register names are always prefixed with a percent sign (**`%`**) to distinguish them from other symbols, as shown in the example above. Constant operands are always prefixed with a dollar sign (**`$`**). Integer values are written as they would be in C: **`12`**, **`014`**, and **`0xc`** all have the same value. The statement

```
movl    $0x80000000,%eax    # set hi-order bit
```

causes the assembler to generate an instruction that sets the high-order bit of the **`eax`** register to 1 and the remaining bits to 0. This statement also includes a comment, which is prefixed with a sharp sign (**`#`**).

[15] Amazingly, the operands are written in the opposite order –with the destination operand first – in the Intel/Microsoft syntax, which can cause a great deal of confusion for developers who must use both assembler formats.

Mnemonics for many instructions in the Intel x86 processor can be used with operands of various sizes. For example, the **`mov`** mnemonic is used to generate instructions that copy 8, 16, or 32 bits (or 64 bits on a 64-bit processor). In some cases, the assembler can determine which instruction variant is required from the operands, but when it isn't possible to do so, the mnemonic must be written with a single-letter suffix that explicitly indicates how many bits are to be moved. Specifically, '**`b`**' is used for byte (8-bit) operands, '**`w`**' for word (two-byte or 16-bit) operands, and '**`l`**' for longword (four-byte or 32-bit) operands. Mnemonics in Tempo's assembly language code usually always include the operand size suffix, even if it is not required. This makes it easier to read and understand the code.

3.3.4 Memory Operands[16]

Memory operands can have up to four components, depending on the particular instruction: a displacement, a base register, an index register, and a scale factor. The displacement is usually a constant or the label of an instruction or variable, and is 8, 16, or 32 bits long. The base register is a general purpose register (**`eax`**, **`ebx`**, **`ecx`**, **`edx`**, **`ebp`**, **`esi`**, or **`edi`**) or the stack pointer register (**`esp`**), and the index register is another general purpose register (but not the stack pointer). The scale factor is 1 (which is implied if it is omitted), 2, 4, or 8. Each of these (except the scale factor) can be positive, negative, or zero. During execution, the value of

```
offset = displacement + base + index × scale
```

is computed as the *effective address* by the processor, and it is added to the base of the (implied or explicit) segment register to determine the actual memory address of the operand[17].

The general assembly language format for a memory operand is

```
displacement(base,index,scale)
```

It is not necessary to specify all four components of a memory operand in every instruction. If no base, index, or scale is required, then only the displacement need be specified. If only a base register is needed, then the index and scale may be omitted. Most instructions in Tempo use only a displacement, or a displacement and a base register.

Memory operands also require reference to a segment register (as mentioned earlier in section 2.1.2). By default, operands that refer to instructions (such as branch and call instructions) implicitly use the code segment register (**`cs`**). Instructions that reference the stack (like push and pop) implicitly use the stack segment register (**`ss`**), and instructions that reference data objects implicitly use the data segment register (**`ds`**). If the implicit segment register is inappropriate, a segment register override can be specified.

Typical memory operands are illustrated in table 1.

---

[16] For complete details, see section 3.7 in volume 1 of the **IA-32 Intel Architecture Software Developer's Manual**.

[17] If paging is being used, the address resulting from combing the base address of the segment and the effective address is further translated; we will cover this activity later in the notes.

Table 1. Assembler Format for Typical Memory Operands

`movl`	`$4,count`	Store the constant 4 in the memory location at `count`.
`addl`	`$12,(%esp)`	Add 12 to the longword (4-bytes) pointed to by the stack pointer register `esp`.
`cmpb`	`%al,2(%ebx)`	Compare the byte in the al register (low order 8-bits of the `eax` register) with the byte in the location whose address is the sum of the value in the `ebx` register and the constant 2.
`subl`	`%eax,tbl(,%ecx,4)`	Subtract the longword in the `eax` register from the longword in `tbl` with the index given in the `ecx` register. Paraphrasing this as C code, we might write `int tbl[100]; tbl[ecx] -= eax;`

3.3.5 Stack Frames

Each function (or block) has a portion of stack space it uses for local and temporary variables, and for the function's return address (that is, the address of the instruction following the call that invoked the function). This is called a stack frame. During the execution of the function, the bottom of the stack frame (or some other fixed location relative to the bottom of the frame) will have its address stored in a processor register, normally called the *frame pointer register*. In the Intel x86 processors, this is usually the `ebp`[18] register.

To call a function, the compiler will generate code to push the function arguments on the stack, and then generate the call instruction. For C programs, the last argument is pushed first, with the other arguments following in order from right to left. The called function will first push the current content of the frame pointer register on the stack, copy the stack pointer register's contents to the frame pointer register, and then reserve space on the stack for the function's local variables. In the x86 this means the `ebp` register is pushed on the stack, the `esp` register is copied to the `ebp` register, and the `esp` register is decremented to reserve space for the local variables.

An example will illustrate these ideas. Let's consider the simple C program shown in Figure 4. This program has a main function with three local variables (`x`, `y`, and `z`) and a second function `f` with two arguments (`a1` and `a2`) and a local variable (`temp`). Also shown is an edited version of the assembler code generated by the gcc compiler for this program.

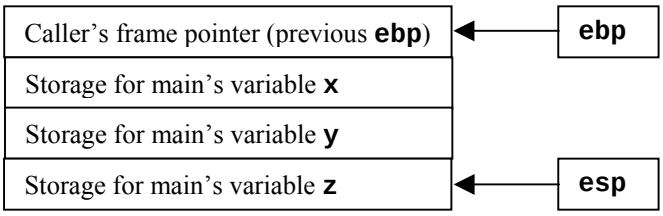
[18] The 16-bit frame pointer register in the Intel x86 is called the `bp` register. This naming pattern applies to most of the other processor registers as well.

Figure 4. A Simple C Program and Equivalent Assembly Language Code

```
Line  Statement
 1    int f(int a1, int *a2)
 2    {
 3       int temp;
 4       temp = a1 * 3;
 5       *a2 = temp;
 6       Return a1 – 5;
 7    }
 8
 9    void main(void)
10    {
11       int x, y, z;
12       x = 12;
13       z = f (x, &y);
14    }
```

```
Line   Statement                                  C Line
 1     f:                                          1
 2             pushl   %ebp
 3             movl    %esp, %ebp
 4             subl    $4, %esp                    3
 5             movl    8(%ebp), %edx               4
 6             movl    %edx, %eax
 7             sall    %eax
 8             addl    %edx, %eax
 9             movl    %eax, -4(%ebp)
10             movl    12(%ebp), %edx              5
11             movl    -4(%ebp), %eax
12             movl    %eax, (%edx)
13             movl    8(%ebp), %eax               6
14             subl    $5, %eax
15             leave
16             ret
17    main:                                        9
18             pushl   %ebp
19             movl    %esp, %ebp
20             subl    $12,%esp                   11
21             movl    $12, -4(%ebp)              12
22             leal    -8(%ebp), %eax             13
23             pushl   %eax
24             pushl   -4(%ebp)
25             call    f
26             addl    $8, %esp
27             movl    %eax, -12(%ebp)
28             leave                              14
29             ret
```

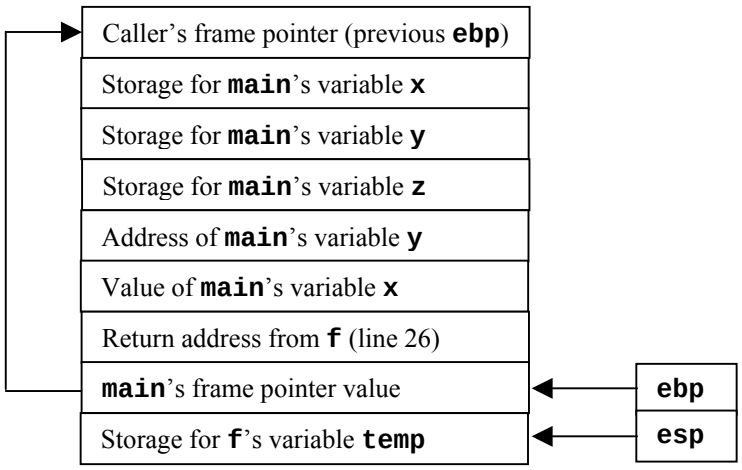
When the main function begins execution, the ebp register is pushed on the stack (line 18) and the esp register is copied to the ebp register. It is important to note at this point that main is just like any other function (although this isn't true for C++), and it is entered with ebp pointing to the stack frame of whatever called it (usually special code provided by the C compiler). On line 20, 12 is subtracted from the esp register, reserving 3 longwords or 12 bytes for the three integer variables x, y, and z (4 bytes or 32 bits each). At this point, the stack looks like this:



On line 21 of the assembler source we see the constant 12 moved into the memory location that has an effective address equal to the sum of **-4** and the contents of the **ebp** register. Recall that the stack grows downward, toward smaller addresses, in the x86, so this is the the effective address of main's variable x, which is consistent with the C statement **x = 12;**

We now turn to the execution of line 13 of the C program: **z = f (x, &y);** This will require calling the function **f** with two arguments, the value of **x** and the address of **y**. Recall that arguments to a function are pushed on the stack in right to left order, so we expect to see the address of **y** pushed first, followed by the value of **x**. Line 22 of the assembler program uses the instruction **lea** (load effective address) to compute the effective address of the memory operand 8 bytes below the location to which the **ebp** register is pointing. Not surprisingly, this is the location containing **y**. The effective address is stored in the **eax** register that is then pushed on the stack (line 23). Line 24 pushes the value of x using the same operand we saw in line 21. Now that both arguments have been pushed, we call the function **f** on line 25.

The call instruction pushes the return address on the stack and transfers control the address specified by its operand, which is clearly the instruction of line 1 of the source code. In this case, the return address is that of the instruction on line 26. The function **f** begins with code very similar to that used when main began execution: the **ebp** register's contents are pushed on line 2 – saving the pointer to main's stack frame), the **esp** register is copied to the **ebp** register on line 3 – establishing the stack frame pointer for **f**, and the stack pointer is reduced by 4 on line 4 to allocate space for the local variable **temp** declared in **f**. The stack now looks like this:



Line 5 of the assembler code uses the operand **8(%ebp**) to reference the location on the stack where the value of argument 1 (**a1**) is located. Once again recalling that the stack grows downward toward smaller addresses, **8(%ebp)** refers to a location higher in memory than the location pointed to by the **ebp** register. This value is copied to the **edx** register. On line 6 that value is also copied to the **eax** register, which is then shifted left one bit (by the **sal** instruction on line 7), effectively multiplying it by 2. To that value is added (on line 8) the contents of the **edx** register, giving us the value **3 × a1**. On line 9, that value is stored in **temp**, which is four bytes below the location to which ebp points.

The next line of the C program, line 5, stores the value of temp into the memory location pointed by the second argument, a2: *a2 = temp; Line 10 of the assembler code copies the value of the second argument, which is 12 bytes above the location pointed to by the ebp register, to the edx register. Line 11 copies the value of temp to the eax register, and finally line 12 copies that value to the location pointed to by the edx register (which is the location to which a2 points). The x86 architecture does not normally allow an instruction to have two memory operands, which is why it is necessary to copy at least one of the operands to a register. The compiler did not optimize the code we are examining (on purpose), which is why it is somewhat longer than required.

The last line of function **f**, line 6, computes the value of **a1 – 5**, and the assembler instructions on lines 13 and 14 should be easy to understand. The value of this expression is left in the **eax** register, which is where the calling program expects the result. The **leave** instruction is a one-byte instruction equivalent to copying the contents of the stack pointer register (**esp**) to the base pointer register (**ebp**), then popping the stack into the base pointer register. This reestablishes the **esp** and **ebp** register contents as they existed immediately after the call instruction on line 25. The **ret** (return) instruction pops the stack into the processor's **eip** register (the program counter, or next instruction address), which returns control to the **main** function.

The purpose of line 26 may not be immediately obvious at this point. Recall that the **main** function pushes two argument values on the stack prior to calling **f**. Although f referenced these values, the stack pointer on return from **f** was exactly the same as it was immediately after the call instruction was executed. Either the **main** function (the caller) or the called function (**f**) must take responsibility for removing those two arguments from the stack (or, saying it more correctly, restoring the stack pointer to its value before the arguments were pushed). We now see that line 26 effectively "pops" two 32-bit quantities from the stack, discarding their values.

Line 27 stores the result of the function call in **main**'s variable **z** (which is 12 bytes below the location pointed to by the **ebp** register. Finally, main executes a **leave** and **ret** instruction, just like **f**, to return to its caller.

Whew! If that appeared to be laborious, you're right – a lot of work was performed to accomplish a simple (nonsensical) computation. But understanding exactly what is happening to the stack in this example is very important to our later understanding of how the operating system deals with interrupts and service requests from application programs.

3.3.6 The Initial User-mode Stack for a Process

When the main function of a process begins execution, it expects to find two arguments (on the stack). And since it is a function, it also expects to find a return address on the stack. Something other than the **`main`** function is responsible for establishing these initial stack contents. In many systems, there is code provided by the compiler libraries that will prepare the stack as required, dealing with whatever mechanism is used by the operating system to provide the needed information. In Tempo, the operating system itself prepares the stack appropriately for the main function.

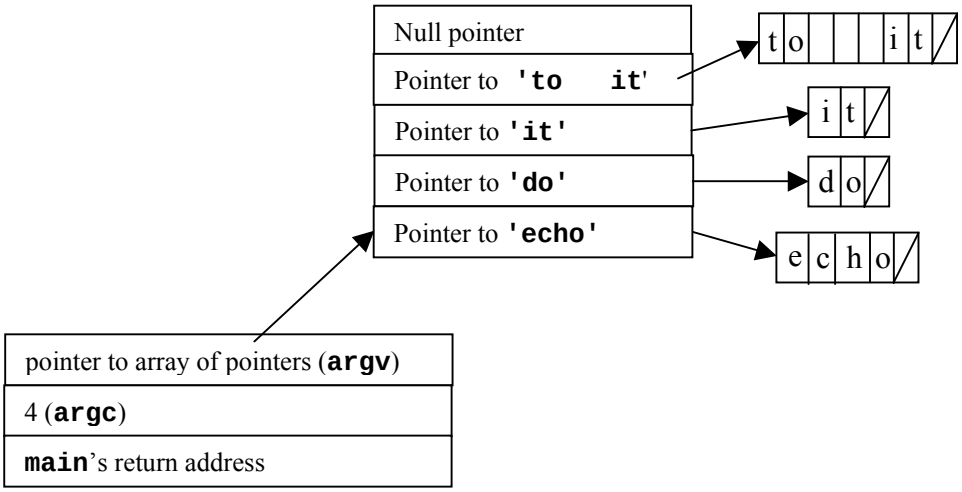
The two arguments **`main`** expects are traditionally called **`argc`** and **`argv`**. **`argc`** is an integer that specifies the number of arguments that were provided on the command line. For example, in UNIX, the command line

```
echo    do it        "to   it"
```

specifies four arguments: **`echo`**, **`do`**, it, and "**`to   it`**" (with three spaces between **`to`** and **`it`**). So the main function for the **`echo`** program should find that its argument **`argc`** has the value 4.

The second argument, argv, is a pointer to an array of pointers to null (zero-byte) terminated strings representing the command line arguments, followed by an additional null pointer. Figure 5 shows what **`echo`** should see when it is invoked with the command line shown above. The required stack contents are shown on the left, and the additional storage that must be initialized is shown toward the right.

Figure 5. Initial Stack Contents, `argv` Array, and Command Line Arguments



Although an application programmer probably doesn't think much about it (if at all), the storage occupied that the **`argv`** array and the character strings to which it points occupy storage that must be part of the address space that belongs to the process. We know that **`argv`**, **`argc`**, and

main's return address are on main's stack, but where is the storage for the **argv** array and the command line argument strings? Is it something that the application programmer can explicitly allocate in the **main** program? No, since the application programmer has no way of knowing how many command line arguments will be supplied or how long each of the strings will be. There may be an *expectation* of a particular number and type of character string arguments, but the reality is not under the control of the application programmer – the person who types the command line that invokes the program controls the number and length of those strings.

One alternative is to use some of the stack space for the **argv** array and the command line arguments. This is the solution used in Tempo. Those items are allocated storage on the stack immediately below (at higher addresses) than the initial stack contents expected by the **main** function. The code to initialize the user stack is somewhat intricate, and is found in the functions **_newproca** and **_run** (both in kernel/sys.c). We will examine it in detail later.

The other item on the stack when the user's main function begins execution is the return address. As we shall see, there are several methods by which a process can terminate, and one (perhaps the most traditional) is for the main function to return. As we have seen, the return from a function (any function, even main) restores the frame pointer and stack pointer registers used by the function's caller, and then executes a **ret** instruction to transfer control back to the caller. The main function for Tempo processes is not called in the traditional manner (with a **call** instruction), so there really isn't a normal return address available. Instead, Tempo supplies the address of a special termination function called processDone (kernel/sys.c) as the return address.

3.3.7 Names

As you read through the source code for the Tempo system you will find some naming conventions that are unusual, at least to an application programmer. Two of these are most obvious, the prefixing of some names with an underscore, and the name **Main** used instead of **main** for the function with which a process begins execution.

Externally-defined symbols in C programs (and usually those written in other languages) are prefixed by the compiler with an underscore when they are emitted into an object code module. Within the C source code, the programmer uses the name without the underscore; it is only the linker that sees the name with the underscore prefix. Indeed, if the C programmer should write an externally-defined symbol name with an explicit underscore prefix, then the linker will see the name as having two prefixed underscores, one explicitly written by the programmer and the other implicitly added by the compiler.

The assembler, on the other hand, does not such prefixing of externally-defined symbols. So if we wish to reference a C function from assembler source code we must explicitly write the symbol with the underscore prefix. This also means that externally-defined symbols in the assembler source code that do not begin with an underscore cannot be referenced from C programs, since the linker will not perform the appropriate resolution of the names.

Assembler symbols are marked as external definitions by specifying them as operands to the **.globl** pseudo-op. For example, the "function" **xmain** (kernel/kernel.s) is actually named **_xmain** in the assembler source code so it can be referenced from C code:

```
.globl  _xmain
```

The other naming oddity, using **`Main`** instead of **`main`** for the function with which a process begins execution, is due to an apparent inadequacy (or requirement) in the Cygwin version of the GNU C compiler. Since there is no **`main`** function in the operating system (which begins execution with an explicit branch from the bootstrap loader), the name **`main`** should not be considered special. In fact, it is possible to use the name **`main`** with processes when the system is built using the Linux GNU C compiler. When the name **`main`** is used with the Cygwin compiler, however, it insists on forcing the linking of additional object code from library modules (in particular, the code that prepares command line arguments and passes them to the **`main`** function). The only way (so far) that has been found to avoid this problem is to not use the name **`main`** for any functions. This is also done when the system is built on UNIX/Linux platforms for consistency. That is, it is desirable for the same source code to be used regardless of the platform used for building the system.

3.4 The Process Table Entries

We are now ready to take our first look at the contents of an entry in the process table. As you recall, one of these entries is associated with every process in the system, regardless of its state. The purpose and use of some fields in the process table entries will not be clear at this point, but we will revisit them at appropriate points in the future.

Let us first just look at the actual C structure used for the process table entries, **`struct Proc`** (toward the end of h/types.h). It appears in a slightly edited form in Table 2.

3.4.1 Self-identifying Structures

Like any large software effort, it is inevitable that errors will be made during the development of an operating system. Since it must manipulate a large number of different data structures, usually identified by a pointer to the structure, a common mistake such as using the wrong pointer value will have significant and far-reaching effects. A common technique that can be used to identify such problems is to make the commonly-used structures *self-identfying*. In Tempo, many structures begin with a field named tag. When the structure is first initialized, this field is set to a value that uniquely identifies the type of structure in which it appears. Later, when the structure is used, the tag field can be queried to determine if it has the value appropriate to the type of structure expected. If it does not, then an error can be reported immediately, avoiding spending a great deal of time later attempting to backtrack to the source of the error.

3.4.2 Types and typedefs

C is not a strongly-typed language. It is possible, however, to give names to types of objects that are used frequently, even if the actual types are simple things like integers and characters. The mechanism used to accomplish this is the **`typedef`** declaration. The syntax for such a declaration is basically the keyword **`typedef`** followed by what appears to be the declaration of a new variable using existing type names. For example, consider the following declaration:

```
typedef struct Proc *Process;
```

Without the keyword **typedef**, this would declare that **Process** is a pointer to a **struct Proc**. As a type definition, however, this makes **Process** a new type name. Now pointers to **struct Proc**s can be declared with the type **Process**, making the code somewhat more readable.

Several such type name definitions appear in the h/types.h file. These frequently are named for a data structure with the suffix "**_t**" to indicate it is a type name. For example, **sem**_t is used to describe the type of the variable that holds a semaphore's identifier, and **pid_t** is used to describe a variable that holds a process identifier.

Table 2. struct Proc

```
struct Proc
{
    tag_t tag;
    pid_t pid;
    Process next;
    Process prev;
    struct Queue *queue;
    int priority;
    int state;
    int stksize;
    unsigned int *stkbase;
    unsigned int *kstkbase;
    int signaledOn;
    int timedOut;
    Process dnext;
    unsigned int delta;
    unsigned int *ksp;
    pid_t waitingon;
    int exitstatus;
    unsigned int eventaddr;
    struct m_header *mfree;
    unsigned int *ptaddr;
    unsigned int staddr;
    int badSema;
    unsigned char wakeupstat;
    char *brkbase;
    char *brkcurr;
    unsigned int tdlo, tdhi;
    unsigned int stklo, stkhi;
    struct EQ eventQueue[NQUEUE];
    struct FD fd[NFD];
    char env[MAXENV];
    char workdir[MAXPATH+1];
}
```

3.4.3 Basic Information

The first few fields in the process table entry provide basic information. **`pid`** contains the unique integer that identifies the process in application programs. Recall that in the kernel we usually use a pointer to the process table entry for this purpose.

The fields **`next`** and **`prev`** are used to construct doubly-linked or singly-linked lists of process table entries. Perhaps the most visible lists constructed using these links are those pointed to by the entries of the readyList array (kernel/sys.c). Each entry in this array is of type struct Queue, which has fields named **`tag`**, **`head`** and **`tail`**. **`head`** and **`tail`** are of type **`Process`**, which we now know are just pointers to process table entries. There is one entry in **`readyList`** for every possible priority that a process (including the idle process) may have. When the system must locate a ready process to replace the current process (when it terminates or becomes blocked), it will search the **`readyList`** array, from the highest priority entry downward, until if finds a non-empty queue – which means there is at least one ready process with that priority, which must be the highest possible priority associated with any ready process in the system.

The **`queue`** field in the process table usually identifies the queue on which the process is located (that is, it is a pointer to the **`struct Queue`** containing the **`head`** and **`tail`** pointers for the queue). This is not always the case, however.

The last three basic information items are **`priority`**, st**a**te, and **`stksize`**. **`priority`**, as its name implies, is the priority associated with the process. **`state`**, again somewhat obviously, identifies the state of the process (all possible state values are given in h/sysparm.h). **`stksize`** informs us as to the size of the user-mode stack associated with the process; the units for **`stksize`** are 4Kbyte pages.

3.4.4 The Other Fields

We are going to defer consideration of the remaining fields in the process table entries until they are encountered in the discussion of other features of the system. We will observe, however, that the majority of these fields are used to record information about memory utilization by the process, information about the process that is changed by the execution of other processes or interrupts (which the affected process is not running), environment strings for the process, and information about files the process has open.

3.4.5 Credentials

Information missing from the Tempo process table entry that is found in those for other systems (like UNIX and Microsoft Windows) is that which would generally be called “credentials.” In particular, we note that Tempo does not have usernames, passwords, user identification numbers, or group identification numbers. These things could be added to the system, if desired, but in a simple pedagogically focused system, they would only tend to obfuscate. And we certainly don’t want that!

3.4.6 Process Ancestry

Another category of information not used by Tempo is related to process relationships. In UNIX systems, the parent of a process and its child processes (created with the fork system call) are

identified by fields in the process table entry. (Technically, a general tree is used in UNIX so a process can have an arbitrarily large number of children without requiring a variable number of fields in the process table entry to identify them.) Since Tempo does not have individual user identification, all processes are have the same ownership, so we do not need to validate the owner of a process that attempts to affect another process (e.g. send it a signal or kill it).

3.5 Transition Between User and Kernel Mode

As you may recall, the primary functions of an operating system are to manage the resources of the system (both the hardware and the information resources), and to provide the user with a virtual machine that is far simpler to use than the basic hardware. To achieve those goals, the operating system retains control of the resources and allows user processes to utilize them by issuing requests for service; these requests are usually named *system calls*.

Actually, the typical application programmer may not use many system calls directly in an application, but may rely on a library of functions or the interpreter for a high-level language to make the needed system calls. For example, programs in C or C++ use library functions to perform input/output, and these in turn may make system calls to carry out the C/C++ program's requests. A Java program is compiled to produce instructions for a virtual machine, but it still uses functions from libraries, and the virtual machine interpreter is dependent on system calls to provide interpretation of physical input/output instructions.

3.5.1 Making a System Call

Making a system call is essentially very similar to calling an ordinary function. Each system call has a set of arguments that specify the details of the service being requested and perhaps provides one or more pointers identifying where results should be placed. System calls almost always provide an indication of success or failure.

Perhaps the major difference between a system call and an ordinary function invocation is that it requires a transition between user mode and kernel mode execution. In user mode a process is prohibited (by the processor) from taking actions that might adversely affect other processes or shared information resources[19]. For example, in user mode a process may not execute input/output instructions, disable interrupt recognition, access memory not explicitly made available to it, halt the CPU, attempt to execute data as instructions, or change the data structures used by the processor to map virtual to physical addresses. These actions are permitted when a user process is executing in kernel mode, but then it is constrained to execute only the code provided by the operating system. Hopefully that code has been carefully prepared to eliminate the possibility that user processes will accidentally or intentionally cause it to fail or produce undesirable results.

In Tempo, the x86 instruction **`int $0x30`** is used to make a system call. Prior to executing this instruction, the process must have placed a number identifying the particular system call being requested in the **`eax`** register (see h/syscall.h for a complete list), and the **`ebx`** register

[19] Actually, such actions may be requested, but the processor recognizes them and generates an exception or fault that is handled by the operating system. In many cases, the offending process will then be terminated.

must either contain the system call's parameter or point to a region of storage that contains the parameters (if more than 32-bits are required).

For example, to change its priority a process will use the **setprio** system call. A single integer parameter is used to specify the requested priority, and the system call number is 11. So the following sequence of instructions could be used to request setting the priority to 5:

```
movl  $11,%eax          # setprio system call number
movl  $5,%ebx           # requested priority
int   $0x30             # make the system call
```

It is considerably more convenient for application programmers to write in a higher-level language, so a function is typically provided for each system call with the same parameters as the system call. The function will place the parameters in the appropriate registers and execute the **int $0x30** instruction. In Tempo, these functions are found in kernel/libs.c. The setprio function looks like this:

```
int setprio(int prio)
{
    return syscall(SYSCALL_SETPRIO,(void *)prio);
}
```

The function **syscall** is written in assembler language (kernel/liba.s). Its arguments are the values to be placed in the **eax** and **ebx** registers, and it returns the value left in the **eax** register by the system call:

```
    .globl  _syscall
_syscall:
    movl    4(%esp),%eax      # put syscall# in %eax
    pushl   %ebx              # ebx must be saved for C calls
    movl    12(%esp),%ebx     # put param ptr in %ebx
    int     $0x30             # execute the system call
    popl    %ebx              # restore the ebx register
    ret
```

3.5.2 The Effects of a System Call

When the x86 processor executes an int $0x30 instruction (when the processor is executing in protected mode) it first uses the interrupt code (0x30) as an index to an array of 8-byte descriptors (the Interrupt Descriptor Table, or IDT). The descriptor at this location provides the following information[20]:

- **The code segment to be accessed.** In the case of the system call descriptor, the segment will be that containing the operating system code (**SYS_CODE_SEL**).

[20] The descriptor also specifies a few other things that are not of interest to us here. See section 4.8.3 in the Intel x86 System Programming Guide for more detail.

- **The entry point for the procedure to be executed.** This will be the primary entry point for all system calls, **`isr30`** (kernel/kernel.s).

- **The privilege level required for a caller trying to access the procedure.** For a system call, this will be 3, the lowest privilege level provided by the processor. Thus code executing at any privilege level can make a system call.

When the system call is made from user mode (privilege level 3), the processor's privilege level is changed to 0 and a stack switch occurs. The particular stack to be used is specified in a special segment called the Task State Segment[21]. This stack switch is performed to guarantee sufficient stack space exists for the kernel's execution (since the amount of space remaining on the user's stack cannot be guaranteed at the time of a system call).

The privilege level and stack switch do not occur if the system call is made from within the kernel, as it is already at privilege level 0 and is using a kernel stack.

In addition to (possibly) switching stacks and changing privilege level, the processor will also save sufficient information to allow the system to return to the program after the system call has been completed. In the case of a simple **`call`** instruction, this information is just the address of the instruction after the call (that is, the value of the **eip** register). For a system call (or any interrupt) the processor will also save the selector for the code segment (**`cs`**), the **`eflags`** register and, in the case of a privilege change, the stack segment selector and stack pointer register (**`ss`** and **`esp`**). The IF flag in the **`eflags`** register will also be cleared to prevent the processor from responding to external interrupts.

3.5.3 Handling the System Call

The handling of any Tempo system call begins at isr30 (kernel/kernel.s). The following steps are taken for any system call:

- **The system call number (in the `eax` register) is validated.** If the number in the **`eax`** register does not correspond to an existing system call, the request is rejected and an appropriate error code is returned to the caller in the **`eax`** register.

- **The interrupt number (0x30), the user's segment registers, and the user's general purpose registers are saved on the kernel stack.** Since the system call code will need to change the values in the processor registers, these are saved so they may be restored before the call returns. The interrupt number is also saved (for reasons we will consider later).

- **Segment registers are set to reference kernel data segments.** The kernel's code and stack segment registers were set appropriately as a result of the **`int`** instruction, but the registers used to access segments containing data must be explicitly changed.

- **The address of the function that will handle the particular system call is obtained.** The **`sftab`** ("system function table") array is indexed by the value in the **`eax`** register (the system call number) to obtain the address of the function that will deal with this particular system call.

[21] See section 4.8.5 of the Intel 86 System Programming Guide for more detail.

- **The content of the `ebx` register is pushed onto the kernel stack.** Recall that the **`ebx`** register contains the system call parameter, or a pointer to a block of storage containing the parameters. By pushing this onto the stack, it makes it available to a C function as an argument.

- **The reschedule flag (`_rs`) is set to 0.** When a system call or external interrupt has been completed, the kernel will use the value of the reschedule flag to decide if it should return to the process that was previously executing (**`_rs`** = 0), select a (potentially different) process for execution (**`_rs`** =1), or terminate the current process (**`_rs`** = 2). The default is to return to the process that was running.

- **A `call` instruction is executed to transfer control to the function that will handle the system call.** This is a traditional call, and will save the return address on the stack.

3.5.4 After the Call Has Been Handled

Once the work of the system call has been completed, the parameter is removed from the stack and the result of the call (in the **`eax`** register) is saved in the stack location where the caller's original **`eax`** register contents were saved. In this way, the system call's results will be available in the **`eax`** register when the system returns to user mode.

It then remains to either return control to the process that made the call, to terminate the process, or – in the event the system call may have changed the execution eligibility of the current process – to make a general selection of the highest priority ready process in the system. The value in the reschedule flag (**`_rs`**) informs as to which of these cases applies.

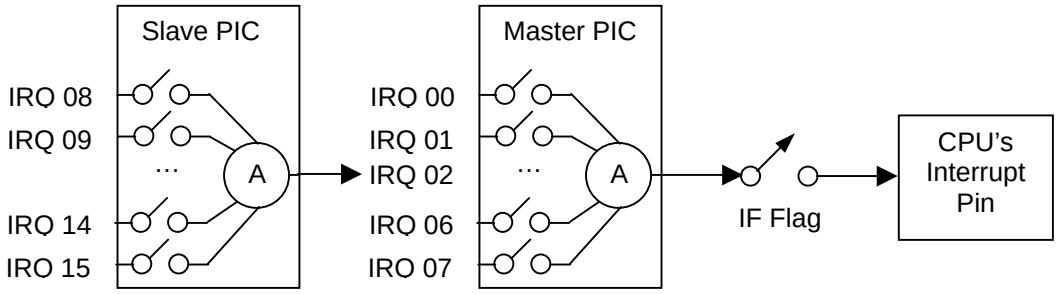
Interrupt Handling X

This chapter describes interrupts and how Tempo deals with them.

X.1 x86 External Interrupt Hardware

Figure 1 is a simple block diagram of the external interrupt hardware on a typical x86 system.

Figure 1. Simple x86 System External Interrupt Hardware



The *Master PIC* and the *Slave PIC* are external interrupt controllers like an Intel 8259A chip. The acronym *PIC* can be interpreted as meaning either *Programmable* Interrupt Controller or *Priority* Interrupt Controller; the device is programmable and it does prioritize simultaneous interrupts.

The lines labeled IRQ00 through IRQ15 are lines (e.g. wires) that carry signals representing external interrupt requests from devices (like a disk drive or the programmable interval timer). When a device detects a condition that is of interest to the system (e.g. the timer determines that a specified time interval has elapsed), it generates a signal on its interrupt request line, informing the PIC that an external interrupt is being requested.

Inside each PIC is an arbitration circuit (shown as a circle labeled "A" in the figure). When more than one of the IRQ lines coming into the PIC are set to request an interrupt, the arbitration circuit decides which of them will be passed on. How it makes this decision is determined by the PIC's programming.

X.1.1 Masking Interrupt Recognition

There is also a logical switch associated with each IRQ line connected to a PIC. Each switch controls whether the signal from the corresponding interrupt line is passed to the arbitration circuit. When the switch is open (as shown in the figure), we say the interrupt request line is "masked," and any signal on that line is blocked from reaching the arbitration circuit. It is important to note, however, that masking the interrupt request line does not result in the removal

of the interrupt request signal itself. If the interrupt request line is later unmasked (effectively closing the switch), the signal will then be considered by the arbitration circuit[22].

In early systems there was only one PIC (that is, one 8259A chip) that could handle 8 interrupt request lines. As systems became more advanced, the need for more than 8 interrupts was recognized, so a second PIC was added. The CPU basically has only a single external interrupt request line, so the output of two PICs cannot be sent directly to the CPU. Instead, the output of the Slave PIC is directed to an interrupt request line on the Master PIC. Specifically, IRQ02 is receives the input from the Slave PIC. Masking IRQ02, then, masks recognition of IRQ08 through IRQ15.

The output of the Master PIC is connected to a pin on the CPU itself. When the Master PIC decides to pass an interrupt request on to the CPU, it signals the CPU on this pin. But the CPU can also mask recognition of all external interrupts by controlling the *IF Flag* bit, which logically controls the state of the switch labeled "IF Flag" in Figure 1. When the IF Flag bit is 0, the switch is open, and interrupt requests from the Master PIC are deferred much as an individual interrupt request to a PIC is deferred when it is masked. As before, clearing the IF Flag bit does not cause the PIC to abandon its efforts to interrupt the CPU; it just prevents the CPU from recognizing the request for the time being.

The IF Flag is part of the EFLAGS register in the processor, and can be controlled in several ways. The CLI instruction will clear the flag (causing external interrupts to be ignored), and the STI instruction will set the flag[23]. Each of these instructions is privileged, and will cause a general protection fault if executed in a user program. The POPF instruction, which pops the stack into the EFLAGS register, may also affect the IF Flag setting if it is executed by suitably privileged code. Attempting to clear the IF Flag with a POPF instruction in user programs is ignored, and does not cause a general protection fault.

Clearing the IF Flag does not prevent the processor from recognizing processor-generated exceptions – internal interrupts (e.g. divide by zero or a page fault).

The interrupt hardware in modern systems is more complicated than what has been presented here, including local APICs (Advanced Programmable Interrupt Controllers), system-based I/O APICs, and NMIs (non-maskable interrupts). The basics still apply, however, and we will not consider these advanced concepts.

X.1.2 CPU Recognition of External Interrupts

Eventually, the CPU will have the IF Flag set and an interrupt request will be signaled from the Master PIC. At that time, the following actions will take place.

---

[22] This behavior is similar to the way electrical wiring and switches in your home operate. If you should turn off a lamp switch and put your finger in the socket, you will not be shocked (at least you shouldn't be – if your home is wired correctly). If you should later turn on the lamp switch – with your finger still in the socket, then sparks will fly, since the electrical current was still waiting, just on the other side of the switch. **Warning: do not try this at home!**

[23] The interrupt recognition is actually not enabled until after the execution of the instruction following STI. This allows a RET instruction, for example, to be completed before interrupt recognition is enabled. See the Intel Instruction Set Reference manual for details.

- **The CPU completes the execution of the current instruction.** Interrupt requests may be signaled by the Master PIC at any time, but the CPU does not respond to them until it has completed the instruction it is currently executing. (Some machines, like the DEC VAX-11, can respond to interrupt requests in the middle of long-running instructions, but the x86 architecture doesn't behave this way).

- **The CPU sends an interrupt acknowledgement signal to the Master PIC.** Since the Master PIC may have to wait until the current instruction is completed and the IF Flag is set, there must be some way to inform it when the CPU has actually recognized its request. That is the purpose of this acknowledgement signal.

- **The Master PIC does priority resolution and reports to the CPU.** At this time the particular interrupt request to be communicated to the processor is determined, and the Master PIC communicates this information to the CPU.

- **The CPU uses the interrupt number (from the PIC) to select an interrupt service route (interrupt handler), and then invokes it.** The service routine (interrupt handler) for the particular external interrupt is called.

- **The interrupt handler responds to the interrupt and sends an End of Interrupt (EOI) to the PIC.** Interrupts at the same or lower priority than the one being serviced are inhibited until the PIC receives an End of Interrupt command from the CPU. A higher-priority interrupt will be recognized and passed on to the CPU and acknowledged (unless the IF Flag is clear).

XXX – More to come

Memory Management X

This chapter describes how Tempo manages the memory in the system.

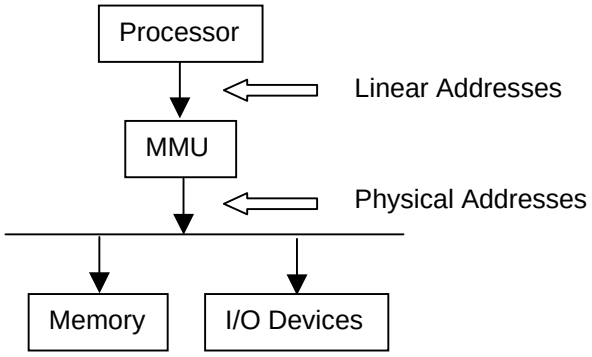
X.1 Memory Addresses

Figure 1 is a simple block diagram of an x86 system that focuses on memory addresses. In protected mode[24], an x86 processor has a *physical address space* of 4GB[25]. This physical address space is mapped to read-write memory (RAM), read-only memory (ROM), and memory on some I/O devices.

During execution, the processor generates *logical addresses* consisting of a *segment selector* and an *offset*. Segment selectors (usually specified implicitly by the type of item being referenced – code, data, or stack) identify a segment descriptor that provides (among other things) the *base address* of the segment. The segment's base address and the offset are added to produce a *linear address*. Each segment descriptor used by the Tempo kernel has a base address of zero, so the linear address is always equal to the offset component of the logical address.

If paging is not being performed by the MMU (memory management unit – actually part of the processor, but shown separately in Figure 1) then the logical address becomes the physical address. If paging is being performed, then the MMU will use a page directory and a page table to transform the linear address into a physical address. We will discuss paging later.

Figure 1. Simple x86 System



[24] In real mode, which is in effect when the processor is reset (or initially receives power), XXX
[25] Starting with the Pentium Pro, it is possible to address 64GB using processor extensions. We will not cover that feature in these notes.

X.1.1 Available Physical Memory

Look again at Figure 1 and recall that there are three different kinds of physical memory: read-write memory, read-only memory, and I/O device memory. Read-write memory, or RAM, is the traditional memory that can read or written by suitably privileged programs. Read-only memory, or ROM holds the instructions and static data the BIOS uses to perform system and I/O device initialization immediately after the system is powered up or reset. The third category, I/O device memory, may be unfamiliar, but is very common. I/O devices may provide access to their registers by mapping them into the physical address space, or on the x86, by mapping them into the I/O address space (where they can be accessed using the processor's IN and OUT instructions).

It should be obvious that the operating system cannot utilize the physical memory regions occupied by ROM or I/O device memory for arbitrary purposes. In order to avoid doing so, the operating system must have complete information about the regions those types of memory occupy.

X.2 Determining Available Memory

XXX – BIOS Information

XXX – Approaches to determining memory usability, and problems that may result.

XXX – More to come

Ethernet Services X

This chapter describes the Ethernet facilities provided by Tempo. It is very definitely a work in progress.

X.1 Brief Introduction to Ethernet

Ethernet is the dominant type of local area networking used today. New PCs have Ethernet devices provided on the system board, and hundreds of different add-in Ethernet devices are available for older PCs. Ethernet network wiring is commonly installed in new buildings, and even in new homes.

Ethernet frames

Flavors of Ethernet (10 base T, 100 base TX, gigabit Ethernet, etc); wireless

Full duplex, half duplex

Introduce the acronym NIC.

X.2 The Tempo Ethernet Services

During system initialization, Tempo locates all Ethernet NICs in the system and attempts to initialize them. Although the electrical (or wireless) connections between Ethernet NICs are standardized, and Ethernet frames from one NIC are usable by all others, there is no such standardization in the software interface to the NIC from a host operating system. Instead, every different type of NIC must have a unique driver prepared for it. There are drivers for a few types of NICs available for Tempo, but if no driver is available, then Tempo will be unable to use the NIC.

There are two primary services provided by Tempo that may be used to access NICs for which drivers are available: **`nicread`** and **`nicwrite`**.

We may later add services that allow specification of multicast addresses, and retrieval/resetting of statistical information (e.g. number of good/bad frames sent/received).

X.2.1 nicread

`nicread` has three arguments:

- **`int nicnum`:** The first argument is an integer that identifies the previously initialized NIC from which a frame is to be read. If this argument is incorrect, nicread will return **`NOSUCHNIC`**.

- **`char *buffer:`** The second argument is a pointer to the buffer to which a frame will be copied. This buffer must be sufficiently large to hold a maximum-sized Ethernet frame (1514 bytes). If this parameter is NULL (or otherwise invalid), **`nicread`** will return **`NULLPARM`**.

- **`int timeout:`** The third argument is a typical timeout specification. If this value is INFINITE (-1), then **`nicread`** will cause the process making the call to be blocked until such time as a good frame is delivered to the process. A value of 0 is used to poll the NIC to see if a frame is immediately available. If not, then **`nicread`** immediately returns **`TIMEOUT`**. A positive non-zero value will cause the process making the call to block for at most that many milliseconds before returning **`TIMEOUT`** or a received frame.

If a frame is successfully copied to buffer, then nicread will return a value between 60 and 1514 indicating the number of bytes in the frame.

If Ethernet support has not been enabled, nicread will return **`NONETSUP`**.

If the NIC identified by **`nicnum`** is inoperative (e.g. no carrier), **`nicread`** will return **`NICINOP`**.

If **`nicread`** returns successfully (with a value between 60 and 1514), the frame copied to the buffer will have the MAC address of the receiving NIC in the first six bytes and the MAC address of the sending NIC in the second six bytes. The frame checksum bytes are not copied to the buffer; **`nicread`** will never return a bad Ethernet frame (e.g. a frame with an invalid size or one with an incorrect checksum).

X.2.2 nicwrite

`nicwrite` has three arguments:

- **`int nicnum`**: The first argument is an integer that identifies the previously initialized NIC which will be used to send the frame. If this argument is incorrect, nicwrite will return **`NOSUCHNIC`**.

- **`char *buffer:`** The second argument is a pointer to the buffer containing the frame to be written. The first six bytes in this buffer must contain the MAC address of the destination. The second group of six bytes in the buffer (bytes 7 to 12) is ignored, but the frame actually written will have these six bytes replaced by the MAC address of the sending NIC. The contents of the buffer pointed to by this argument will not be modified. In particular, the caller of nicwrite is responsible for guaranteeing the Ethertype field (bytes 13 and 14 of the frame) is in network byte order. If this parameter is NULL (or otherwise invalid), **`nicwrite`** will return **`NULLPARM`**.

- **`Size_t len:`** The third argument specifies the length of the frame to be written. This value must be in the range 60 to 1514. If it is not, **`nicwrite`** will return **`BADVAL`**.

If Ethernet support has not been enabled, **`nicwrite`** will return **`NONETSUP`**.

If the NIC identified by **nicnum** is inoperative (e.g. no carrier), **nicwrite** will return **NICINOP**.

The process calling **nicwrite** will be blocked until the frame is accepted for transmission. Successful return from **nicwrite** with a value of **NOERROR** does not necessarily mean the frame was successfully transmitted or received at the destination, but only that it has been accepted for transmission.

X.3 Generic NIC Data Structures

Each NIC that is located and successfully initialized (even if it is not actually operational) will be identified by an entry in the **_nicdev** array (kernel/sys.c). The number of such devices is specified by **_numnic** (kernel/sys.c), which will never be larger than **MAXNIC** (kernel/kernel.h – XXX – should this be moved to h/sysparm.h?) If there are more than **MAXNIC** acceptable NICs in the system, some of them will be unusable. Currently there is no report of this situation during initialization, but this may be added in the future.

Each entry in the **_nicdev** array is of type **struct nic** (h/enet.h) and has the following fields:

- **unsigned char nic_bus:** the PCI bus on which the NIC was found
- **unsigned char nic_dev:** the NIC's device number on **nic_bus**
- **int nic_force**: used to force a particular speed and duplex setting to be used for the device
- **unsigned char nic_speed**: the speed at which the NIC is operating. This may be **NIC_0**, **NIC_10**, **NIC_100**, or **NIC_1000**, corresponding to a NIC that is inoperable (e.g. no carrier) or operating at 10,100, or 1000 Mbps.
- **unsigned char nic_duplex**: either **NIC_HALF** or **NIC_FULL**, indicating whether the NIC is operating in half or full duplex mode. The value in this field is undefinedif nic_speed is **NIC_0**.
- **Eaddr nic_maddr**: six bytes containing the binary MAC address of the NIC
- **Eaddr nic_baddr**: six bytes containing the binary MAC broadcast address for the NIC
- **unsigned long nic_rcvd**: number of good frames received
- **unsigned long nic_sent**: number of frames accepted for transmission
- **unsigned long nic_ierrs**: number of input frame errors (e.g. frames dropped because they were too long or too short, had bad checksums, or were received with no space to store them – overruns).

- **`unsigned long nic_oerrs`**: number of output frame errors (to the extent that such errors can be identified by the driver)
- **`int (*nic_write)(struct nic *, unsigned char *, unsigned short len)`**: pointer to a driver-specific function that will begin the sending of a frame
- **`void *nic_ext:`** pointer to a structure that is unique to the particular NIC; the contents of this structure are opaque to the nicread and nicwrite system calls.
- **`int nic_ninq:`** the number of good received frames waiting delivery to a process
- **`sem_t nic_readsem:`** a semaphore on which processes calling **`nicread`** will block while waiting for a frame; the count in this semaphore will be equal to **`nic_ninq`**.
- **`sem_t nic_writesem:`** a semaphore on which processes calling **`nicwrite`** will block while waiting to send a frame; the count in this semaphore will be equal to 0 when a process is executing **`nicwrite`** and 1 otherwise.
- **`struct inq nic_inq[MAXINQ]:`** an array containing descriptions of frames available to be read by **`nicread`**. The first **`nic_ninq`** entries (with subscripts 0 to **`nic_ninq-1`**) are valid.
- **`unsigned char bfree[MAXINQ]:`** an array of 0/1 values indicating if the corresponding input buffer (pointed to by elements of the **`inqbuff`** array) is in use (contains an unprocessed frame) or not. A value of 1 means the buffer is available for use.
- **`unsigned char *inqbuff[MAXINQ]:`** an array of pointers to buffers with at least **`EP_MAXLEN`** bytes each.
- **`unsigned char rxbuff[MAXINQ*EP_MAXLEN]:`** a static storage area that will be used to hold good received Ethernet frames that are available for use by **`nicread`**.

X.4 Initialization

The function `enetinit` (kernel/enetinit.c) is called during system initialization to identify and initialize up to `MAXNIC` Ethernet devices. It currently can only recognize NICs on a PCI bus.

The function scans the PCI busses for devices that are identified as network controllers. It stops after scanning all busses and devices, or **`MAXNIC`** devices have been identified and initialized. When it finds a candidate device, it calls the initialization function in each of the known Ethernet drivers. As soon as one of these successfully returns, the PCI bus scan continues. If all the driver initialization functions have been called but none has returned successfully, we conclude that this NIC is one for which we have no driver, and ignore the device. We may later wish to report finding devices without drivers.

X.4.1 Individual Device Initialization

The initialization function for each NIC is called with three arguments, a pointer to the **`struct nic`** that should be prepared for the device, the PCI bus number on which the device was found, and the device number of the NIC on the specified PCI bus.

XX – More to come