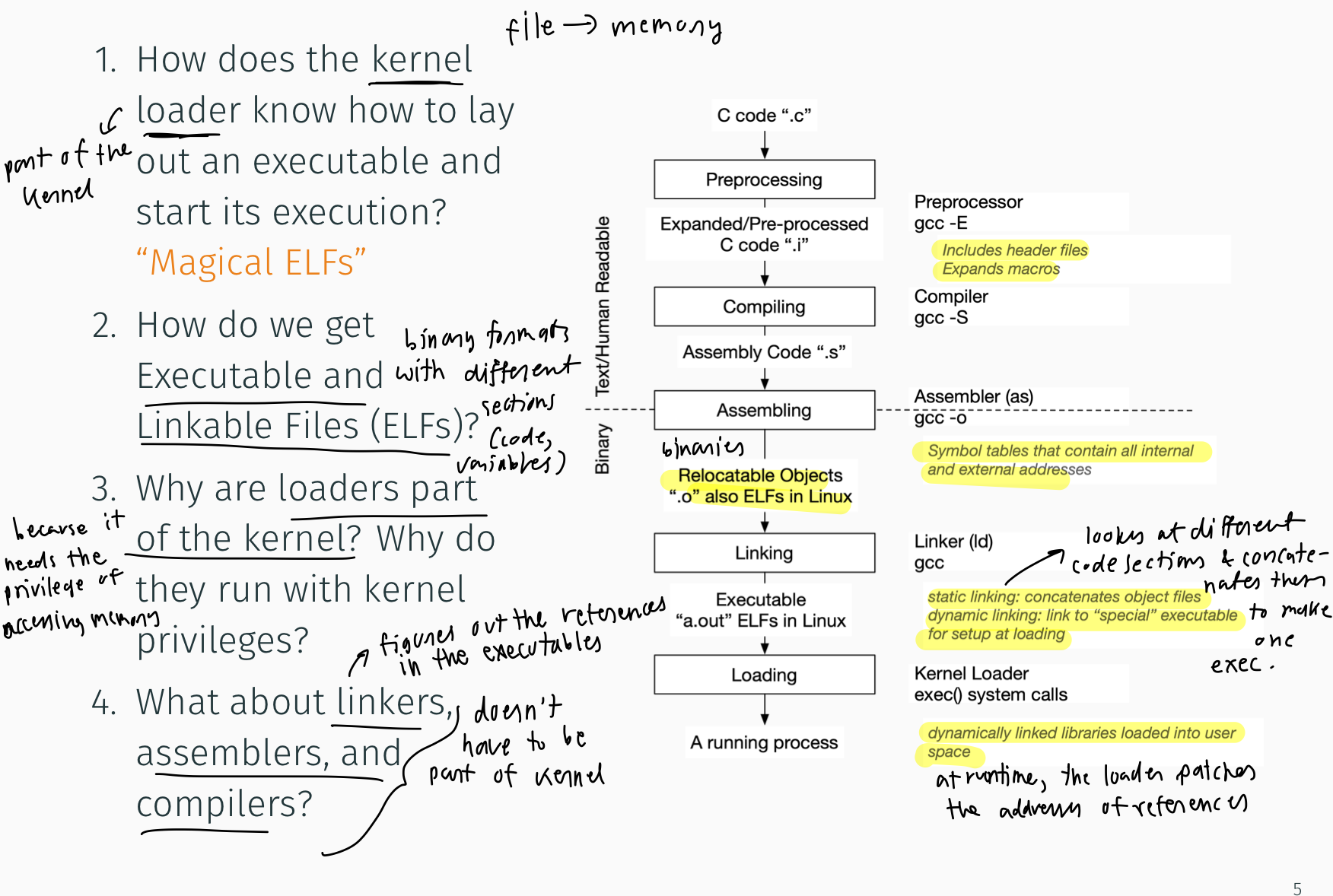
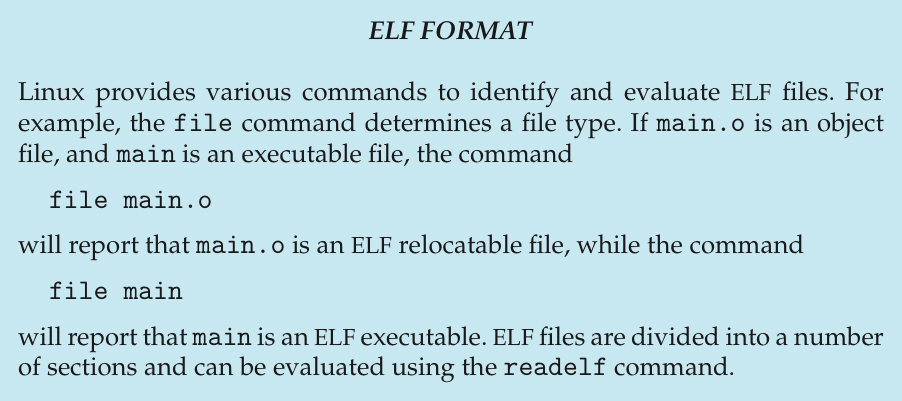
**2.5 Linkers and Loaders**

****

1. **Pre-processor**: The preprocessor handles “#include” “#define” “#ifdef”… and so forth. It might also strip comments and unnecessary white space.
2. **Compiler**: Takes that code and turns it into an “object file” - which typically contains machine code.
3. Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as a **relocatable object file.**
4. **Assembler**: An assembler converts from symbolic machine code to binary machine code, and from symbolic data to binary data.
5. **Linker**: derives the executable file derives from combining one or more program object files and probably some library object files. The linker is a normal user program, without any privileges.
6. A run of a program is initiated by loading the contents of an (binary) executable code file into memory, using a **loader**. The loader is usually an integrated part of the operating system. All initialized parts of the program derive from the executable code file, in which all addresses should be based on segments starting at zero. The loader reads these segments from the executable code file and copies them to suitable memory segments; it then creates a stack segment and jumps to a predetermined location in the code segment, to start the program. So, the executable code file must contain a code segment and a data segment; it may also contain other indications, for example the initial stack size and the execution start address.
7. **Relocation** assigns final addresses to the program parts and adjusts code and data in the program to match addresses so that, for example, the code can call library functions and access its variables as it executes. This happens during linking and loading.
8. Libraries can also be linked dynamically (DLL in Windows). It avoids linking and loading libraries that may end up not being used into an executable file.
9. Code and data segments in object files may contain addresses of locations in other program object files or in library object files. A location *L* in an object file, whose address can be used in other object files, is marked with an **external symbol**, also called an external name; an external symbol looks like an identifier. The location *L* itself is called an **external entry point**. Object files can refer to *L* by using an external reference to the external symbol of *L*. Object files contain information about the external symbols they refer to and the external symbols for which they provide entry points. This information is stored in an **external symbol table**.
10. Object files and executable files typically have standard formats that include compiled machine code and a symbol table containing metadata about functions and variables that are reinforced in the program. For Linux, the standard format is called an ELF (**Executable Linkable Format**).
11. One universal concept among all different ELF file types (and also a.outand many other executable file formats) is the notion of a section. Simply put, a section is a collection of information of a similar type. Each section represents a portion of the file. For example, executable code is always placed in a section known as **.**text; all data variables initialized by the user are placed in a section known as .data; and uninitialized data is placed in a section known as .bss**.**
12. Given that we want all executable portions of an executable in read-only memory and all modifiable locations of memory (such as variables) in writable memory, it turns out to be most efficient to group all of the executable portions of an executable into one section of memory (the **.text** section), and all modifiable data areas together into another area of memory (henceforth known as the **.data** section).
13. If the user has not specified the initial value of a variable, there is no sense wasting space in the executable file to store the value. Thus, initialized variables are grouped into the **.data** section, and uninitialized variables are grouped into the **.bss** section, which is special because it doesn't take up space in the file—it only tells how much space is needed for uninitialized variables.



**2.9 Building and Booting an Operating System**

**2.9.2 System Boot**

1. The process of starting a computer by loading the kernel is known as **booting** the system.
   1. A small piece of code known as the bootstrap program or boot loader locates the kernel.
   2. The kernel is loaded into memory and started.
   3. The kernel initializes hardware.
   4. The root file system is mounted (system is **running**).
2. When the computer is first powered on, a small boot loader located in nonvolatile firmware known as **BIOS** is run.
3. The initial boot only loads a second boot loader, located at a fixed disk location called the **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it must fit in a single disk block) and knows only the address on disk and the length of the remainder of the bootstrap program.
4. Many recent computer systems have replaced the BIOS-based boot process with UEFI (Unified Extensible Firmware Interface). UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. Perhaps the greatest advantage is that UEFI is a single, complete boot manager and

therefore is faster than the multistage BIOS boot process.

**20.6.3 Execution and Loading of User Programs**

1. The Linux kernel’s execution of user programs is triggered by a call to the exec() system call.
2. The exec() call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program.
3. It first verifies that the calling process has permission rights to the file being executed.
4. Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an exec() system call is made.
5. Older Linux kernels understood the a.out format for binary files —a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern **ELF format**, now supported by most current UNIX implementations. **ELF has a number of advantages over a.out, including flexibility and extendibility. New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and a.out binary formats in a single running system**.

**20.6.3.1 Mapping of Programs into Memory**

1. The pages of the binary file are mapped into regions of virtual memory.
2. The kernel’s binary loader sets up the initial memory mapping.
3. An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader reads the header and maps the sections of the file into separate regions of virtual memory.
4. The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses.
5. The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.

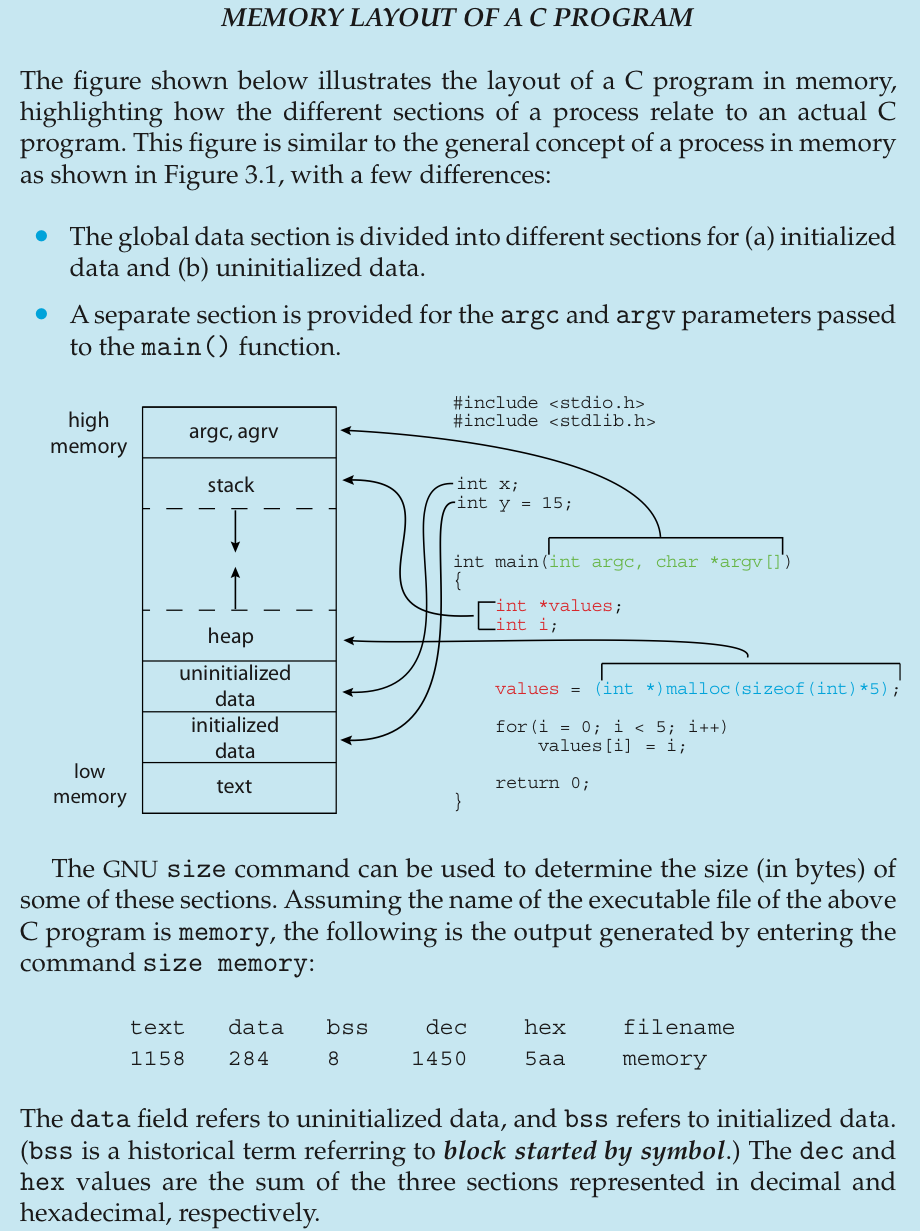
**20.6.3.2 Static and Dynamic Linking**

1. Static linking: library functions are embedded directly in the program’s executable binary file.
2. The main disadvantage of static linking is that every program generated must contain copies of the same common system library functions.
3. Dynamic linking: much more efficient, in terms of both physical memory and disk-space usage, loads the system libraries into memory only once. System libraries that are linked to user programs when the processes are run, with linking postponed until execution time.

*Boot Loading*

* Why does loading often occur in a multi-stage process?
  1. **Process Concept**
     1. **The Process**

1. A process is a CPU activity.
2. A program itself is not a process. A program is a passive entity (file containing a list of instructions stored on disk), while a process is an active entity (with a program counter specifying the next instruction to execute and a set of associate resources). A program becomes a process when an executable file is loaded onto memory.
3. The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor’s registers
4. The memory layout of a process is divided into multiple sections:
   1. Text section: the executable code
   2. Data section: global variables
   3. Heap section: memory that is dynamically allocated during program run time
   4. Stack section: temporary data storage when invoking functions (such as function parameters, return addresses, local variables)



1. Each time a function is called, an **activation record** containing function parameters, local variables, and the return address is pushed onto the stack; when control is returned from the function, the activation record is popped from the stack.
2. The heap will grow as memory is dynamically allocated and will shrink when memory is returned to the system.

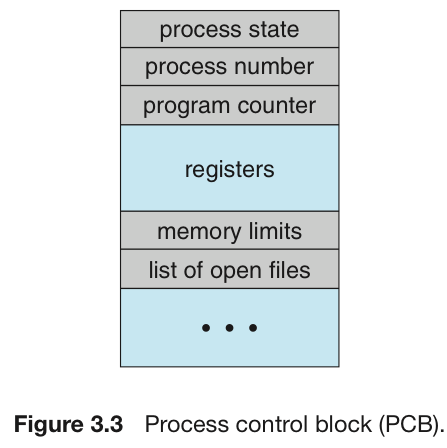
**3.1.2 Process State**

1. As a process executes, it changes state.
   1. New: The process is being created.
   2. Running: Instructions are being executed.
   3. Waiting: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
   4. Ready: The process is waiting to be assigned to a processor.
   5. Terminated: The process has finished execution.

**3.1.3 Process Control Block**

1. Each process is represented in the OS by a process control block (PCB) / task control block.

* Process state: The state may be new, ready, running, waiting, halted, and so on.
* Program counter: The counter indicates the address of the next instruction to be executed for this process.
* CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
* CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
* Memory-management information: This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
* Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
* I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.



**2.3 System Calls**

1. System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, but low-level tasks may have to be written in assembly-language instructions.
2. It serves three purposes:
   1. provides an abstracted hardware interface for user-space.
   2. system calls ensure system security and stability. With the kernel acting as a middle- man between system resources and user-space, the kernel can arbitrate access based on permissions, users, and other criteria.
   3. a single common layer between user-space and the rest of the system allows for the virtualized system provided to processes.
3. System calls are the only means user-space has of interfacing with the kernel; they are the only legal entry point into the kernel other than exceptions and traps.
4. It is not possible for user-space applications to execute kernel code directly. They cannot simply make a function call to a method existing in kernel-space because the kernel exists in a protected memory space. If applications could directly read and write to the kernel’s address space, system security and stability would be nonexistent.

**2.3.2 Application Programming Interface**

1. Application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
2. Another important factor in handling system calls is the run-time environment (RTE) — the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.
3. The RTE provides a system-call interface that serves as the link to system calls made available by the operating system.
4. In Linux, each system call is assigned a **syscall number**.This is a unique number that is used to reference a specific system call. When a user-space process executes a system call, the syscall number identifies which syscall was executed; the process does not refer to the syscall by name.
5. The kernel keeps a list of all registered system calls in the system call table, stored in sys\_call\_table.

**12.2.3 Interrupts**

1. An interrupt is physically produced by electronic signals originating from hardware devices and directed into input pins on an interrupt controller, a simple chip that multiplexes multiple interrupt lines into a single line to the processor. Upon receiving an interrupt, the interrupt controller sends a signal to the processor. The processor detects this signal and interrupts its current execution to handle the interrupt. The processor can then notify the operating system that an interrupt has occurred, and the operating system can handle the interrupt appropriately.
2. Different devices can be associated with different interrupts by means of a unique value associated with each interrupt. These interrupt values are often called interrupt request (IRQ) lines. Each IRQ line is assigned a numeric value.
3. The mechanism to signal the kernel is a software interrupt: Incur an exception, and the system will switch to kernel mode and execute the exception handler. The exception handler, in this case, is actually the system call handler.
4. Exceptions occur synchronously with respect to the processor clock.
5. The defined software interrupt on x86 is interrupt number 128, which is incurred via the int $0x80 instruction. It triggers a switch to kernel mode and the execution of exception vector 128, which is the system call handler.
6. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction.
7. When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the **interrupt-handler routine** at a fixed address in memory.
8. The function the kernel runs in response to a specific interrupt is called an interrupt handler or interrupt service routine (ISR). Each device that generates interrupts has an associated interrupt handler. For example, one function handles interrupts from the system timer, whereas another function handles interrupts generated by the keyboard. The interrupt handler for a device is part of the device’s driver—the kernel code that manages the device.
9. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.
10. Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors.
11. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.
12. The interrupt mechanism also implements a system of **interrupt priority levels**.
13. An interrupt handler needs to execute quickly, and also do a large amount of work, so the processing of interrupts is split into two parts. The interrupt handler is the top half, run immediately upon receipt of the interrupt and performs only the work that is time-critical. Work that can be performed later is deferred until the bottom half, which runs in the future, at a more convenient time, with all interrupts enabled.
    1. If the work is time sensitive, perform it in the interrupt handler.
    2. If the work is related to the hardware, perform it in the interrupt handler.
    3. If the work needs to ensure that another interrupt (particularly the same interrupt) does not interrupt it, perform it in the interrupt handler.
    4. For everything else, consider performing the work in the bottom half.
14. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected.
15. Because interrupt handing in many cases is time and resource constrained and therefore complicated to implement, systems frequently split interrupt management between a **first-level interrupt handler (FLIH)** and a **second-level interrupt handler (SLIH)**. The FLIH performs the context switch, state storage, and queuing of a handling operation, while the separately scheduled SLIH performs the handling of the requested operation.
16. A program uses library calls to issue system calls. The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a **software interrupt** (lower priority than hardware interrupt), or **trap**. This instruction has an operand that identifies the desired kernel service.
17. When executing an interrupt handler, the kernel is in interrupt context. Process context is the mode of operation the kernel is in while it is executing on behalf of a process. In process context, the current macro points to the associated task. Because a process is coupled to the kernel in process context, process context can sleep or otherwise invoke the scheduler.
18. Interrupt context, on the other hand, is not associated with a process. The current macro is not relevant (although it points to the interrupted process). Without a backing process, interrupt context cannot sleep, or reschedule. Certain functions cannot be called from interrupt context. If a function sleeps, it cannot be used from interrupt handler, which limits the functions that can be called from an interrupt handler.

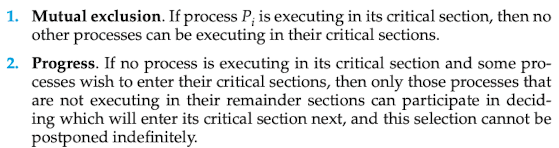
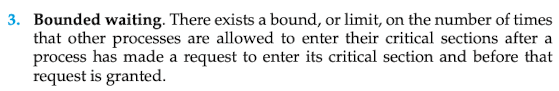
*System Calls vs. Interrupts*

* In x86, what does the INT instruction do?
* What is the difference between a system call, an exception and an interrupt?

**6.1 Background**

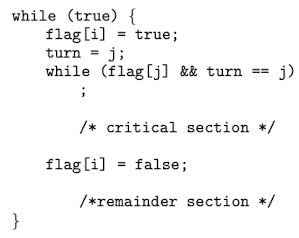
1. **Race condition**: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place. To guard against the race conditions, we need to synchronize the processes.

**6.2 The Critical-Section Problem**

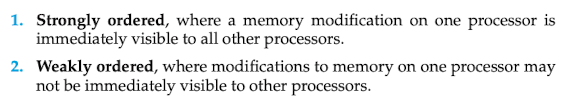
1. Each process has a segment of code, called a **critical section**, in which the process may be accessing/updating data shared with at least one other process.
2. When a process is executing its critical section, no other process is allowed to execute its critical section.
3. The critical section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data.
4. Each process must request permission to enter its critical section.
5. The section of code implementing this request is the **entry section**.
6. The critical section may be followed by an **exit section**.
7. The remaining code is the **remainder section**.
8. A solution to the critical-section problem must satisfy the following requirements:   
     
   
9. Two general approaches are used to handle critical sections in operating systems:
   1. Preemptive kernels: allows a process to be preempted while it is running in kernel mode
   2. Nonpreemptive kernels: a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.
10. There are no race conditions in nonpreemptive kernels
11. A preemptive kernel may be
    1. More responsive, since there is less risk of a kernel-mode process running for an arbitrarily long period before relinquishing the processes to other processes
    2. More suitable for real-time programming, allows a real-time process to preempt a process currently running in the kernel.

**6.3 Peterson’s solution**





**6.4.1 Memory Barriers**

1. How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**.   
   
2. Computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**. When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.

* What are race conditions? What are possible consequences?
* When would you use a spin lock vs a mutex (blocking lock) or semaphore?
* Can the underlying architecture (uni-processor, multiprocessor, support for certain instructions, etc.) affect your synchronization choice? If so how?
* How can mutexes, semaphores, monitors and condition variables be used to solve different critical section problems?
* What are deadlocks and how can they occur?
* What is the produce/consumer problem?