Virtual Memory

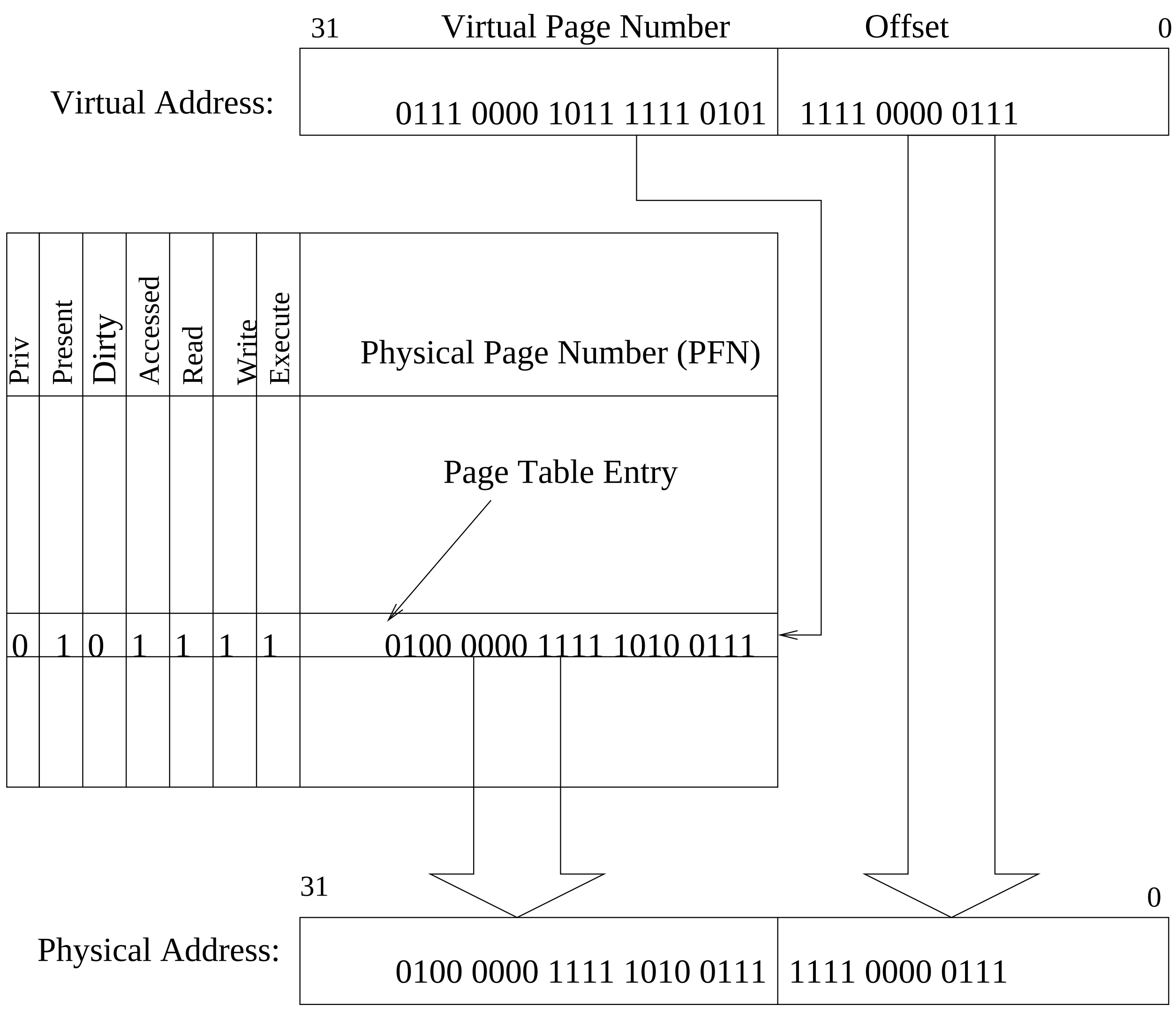
We have stated that a UNIX process is a virtual computer in which a thread of execution (virtual processor) runs within a private virtual address space. In this unit, we will begin to explore the mechanisms by which this is implemented, both from a generic hardware and a generic kernel standpoint. In a later unit, we will return to this topic and see specific hardware architecture and kernel code.

All addresses used in a program are **virtual addresses**. Before being used to address physical memory or memory-mapped I/O, virtual addresses are **translated** to **physical addresses**. This translation is performed **by hardware** within the processor known as the **Memory Management Unit (MMU)**.

Since the operating system kernel has sole access to the MMU's lookup tables, it has the ability to completely control the view of memory a given process can see.

Address translation is performed with the granularity of a **page**. An address is divided into two parts: The **page number**, comprising the most significant bits, and the **offset**, comprising the least significant bits. Address translation operates on the **virtual page number**, replacing it with the **physical page number**. The offset is passed through unchanged. Typical page sizes are 2K, 4K and 8K. On the X86 32-bit and 64-bit architectures a page size of 4K is used.

Another name for a physical page of memory is a **page frame**, because we can think of physical pages as empty frames into which the actual meaningful content is placed from time to time. We'll see that a given page frame will hold many different virtual page images over time.



The virtual page number (conceptually) indexes the following per-page data:

- **Present bit**: If set, this virtual page is resident (maps to a physical page frame) and the physical page number field is valid. If the Present bit is clear, there is no physical page frame allocated to this virtual address. When there is no translation (page table entry has clear Present bit), it results in a **Page Fault**. As we will see, this is not necessarily a bad thing.

- The **Physical Page Number** (also known as the Page Frame number or PFN) which is translated for this virtual page. If the Present bit is clear, this field is invalid.

- **Protection bits**: Determines what type of accesses are allowed to this page: read, write, execute. An attempt to perform a disallowed access results in a **protection fault**. (Not all hardware maintains all three of these protection bits. In particular, the x86 architecture prior to Pentium-4 has only two levels of access control: readonly and readwrite, and does not distinguish between protection faults and page faults.)

- **Dirty bit**: Set by hardware when a page is modified. Cleared by the kernel when the page has been synced to backing storage.

- **Accessed bit**: Set by hardware when a page is accessed (read, write or execute). Cleared by the kernel while aging (scanning) page frames for re-use.

- **Privileged Bit**: When set, this page can only be accessed while the processor is in privileged (i.e. "kernel") mode. This allows kernel memory to be mapped into the virtual address space of a process and be visible only when the process enters the kernel. The Linux X86 architecture uses the Privileged Bit approach (we'll see how in Unit 12), but other architectures switch to a different set of page tables upon entry to the kernel.

Multi-Level Page Tables

Conceptually, the virtual page number is an index into an array (located at a specific physical address), each entry of which contains the physical page number and flags described above. There are several problems which prevent this from being a realistic implementation.

Let us consider a 32-bit architecture with 4K pages, therefore the page numbers are 20 bits long and there are 2ˆ20 possible page numbers. If each PTE is 32 bits long, then each page table would consume 4MB. Although this might seem like a small amount of memory, remember that each process on the system has its own page table. A system with 100 processes would therefore be wasting a large amount of memory on page tables.

The problem is even more severe on 64-bit machines which have a larger virtual address space. Even if only 48 bits of the address space are recognized, with a 4K page, there would be 2ˆ36 PTEs (each PTE must be 64 bits to handle the longer addresses) consuming 2ˆ39, or 512GB, of physical memory per process!

The problem of large page tables is addressed with **multi-level** page tables. The physical address of the top-level page table is given in the CPU/MMU "BASE" register (register %cr3 on X86). The most significant bits of the virtual address index into this top-level table. The result is not the page table entry, but is the physical address of the corresponding table at the next lower level of the hierarchy. That physical address, plus the bits from the next most significant part of the virtual address, locates the entry in the next page table, etc. At the last step, the actual PTE (page table entry) is fetched, which contains the PFN and the flags.

On modern computer architectures, 2-, 3- or 4-level paging structures are the norm. The illustration below shows 3 levels, and is meant to be schematic, not any particular processor type. 2-level or 3-level structures are typical for 32 bit machines, while 64 bit virtual addresses mandate 4-level structures to achieve any kind of memory efficiency.

On the X86 32 bit system, the page table is 2-level with the page number split 10/10 bits. Since PTEs fit in a 32-bit word, the top-level table and all of the second-level tables are 4096 bytes long, i.e. one page. On X86-64, although addresses are 64 bits, only 48 bits are significant. With a 4K page size, this leaves a 36 bit page number, which is translated via a 4-level page table split 9/9/9/9. 64 bits are required for each PTE. Once again, at each level the table is one page long.

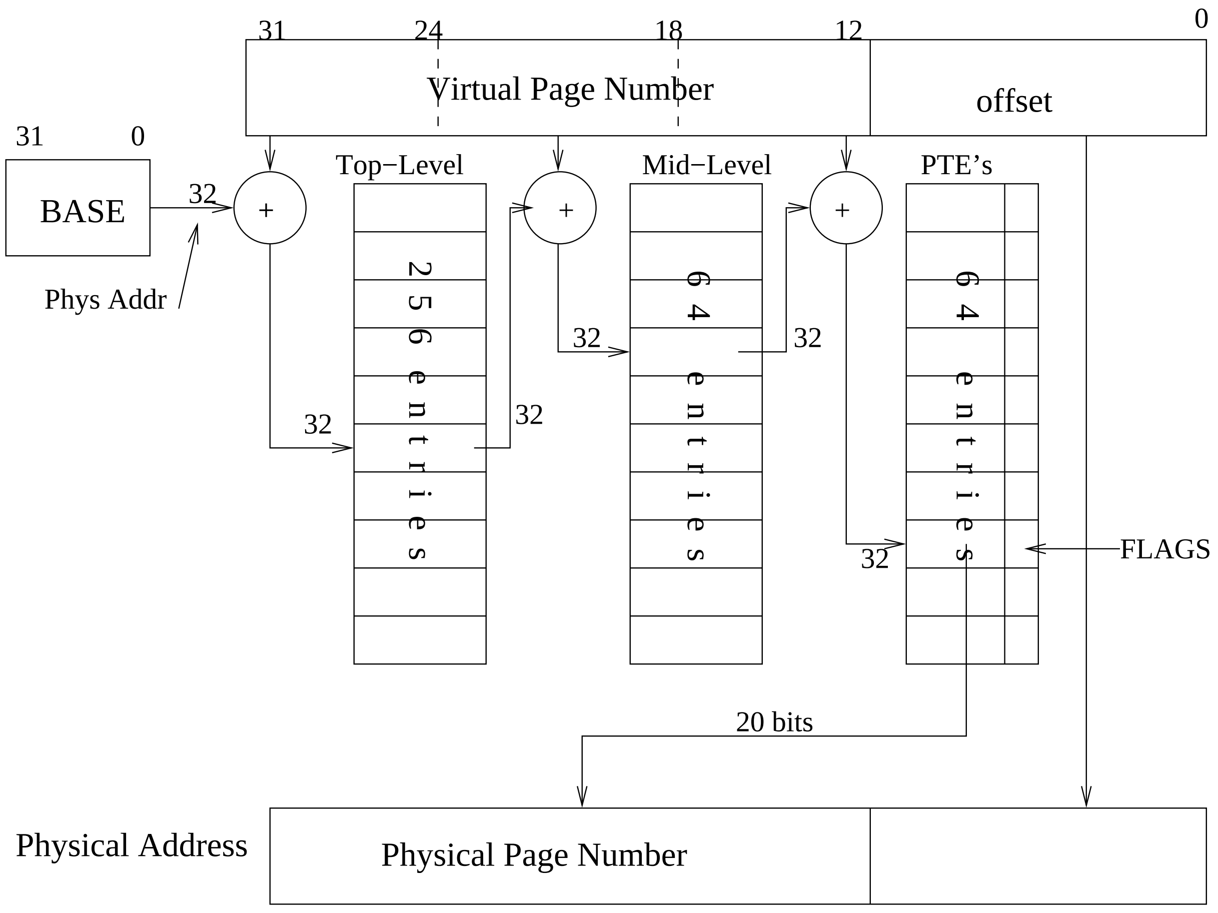
It is possible for entries in these tables (other than the lowest-level page table itself) to contain "NULL pointers". This means that the entire contiguous range of virtual addresses corresponding to that entry is not mapped. Whenever a NULL pointer is encountered during the traversal of the multi-level page table structure, it causes a Page Fault, the same as if the Present bit of the PTE were 0.

As a specific example, let us say the address size is 32 bits and the page table structure is 10/10. Therefore, each entry in the top-level page table covers 10+12=22 bits, or 4MB of address space. Suppose that the first entry in the top-level table is NULL. Then virtual addresses 0x00000000-0x003FFFFF will be unmapped. This scheme avoids having to allocate page table entries for large, unmapped portions of the virtual address space. This is analogous to sparse file allocation (recall from unit 2 that when one lseeks beyond the current end of file and writes, this creates a "hole" in the file for which actual disk storage may not be allocated)

The entire page table structure of every process on the system need not be resident at any given time. The kernel can construct the necessary structures "on-the-fly" in response to a page fault. Only the top-level page table needs to be allocated for each process.

Page tables are in effect a data structure, kept in physical memory. The layout of this data structure is mandated by the processor architecture. The kernel has been compiled with architecture-specific code that understands the layout of the page tables and can manipulate and manage them.

Since the base address of the page table for the currently running process is stored in a CPU register (register CR3 for the X86), at **context switch** time, the kernel changes this register to point to the top-level page table of the new process, thus switching to its virtual address space. The figure below is a *hypothetical* 3-level architecture with a 32-bit address, 4K page size and 8/6/6 split.



Translations Cache

Page table entries reside in main memory, and thus consume two important resources: The memory needed to store the page tables and the time required to perform translations. We have seen how multi-level page tables address the former concern. The latter is extremely significant too, since a translation must be performed on every memory access. If the MMU worked literally as described thus far, for a 2-level page table, 2 additional memory accesses would be required for every access attempted, i.e. memory performance would be reduced to one third. Therefore, a cache, often called the **Address Translation Cache (ATC)** or **Translation Lookaside Buffer (TLB)**, is used to speed translations.

Like any cache, the ATC is a content-addressable memory. A **key** is presented, and either the stored **value** is accessed (a "**hit**"), or the key is not found in the cache, which is known as a **cache miss**. Recall that the address translation process takes the page number portion of a virtual address and returns a page table entry. Therefore, the key in this case is the virtual address being translated (or more correctly, the page number portion of the VA), and the value accessed is the PTE.

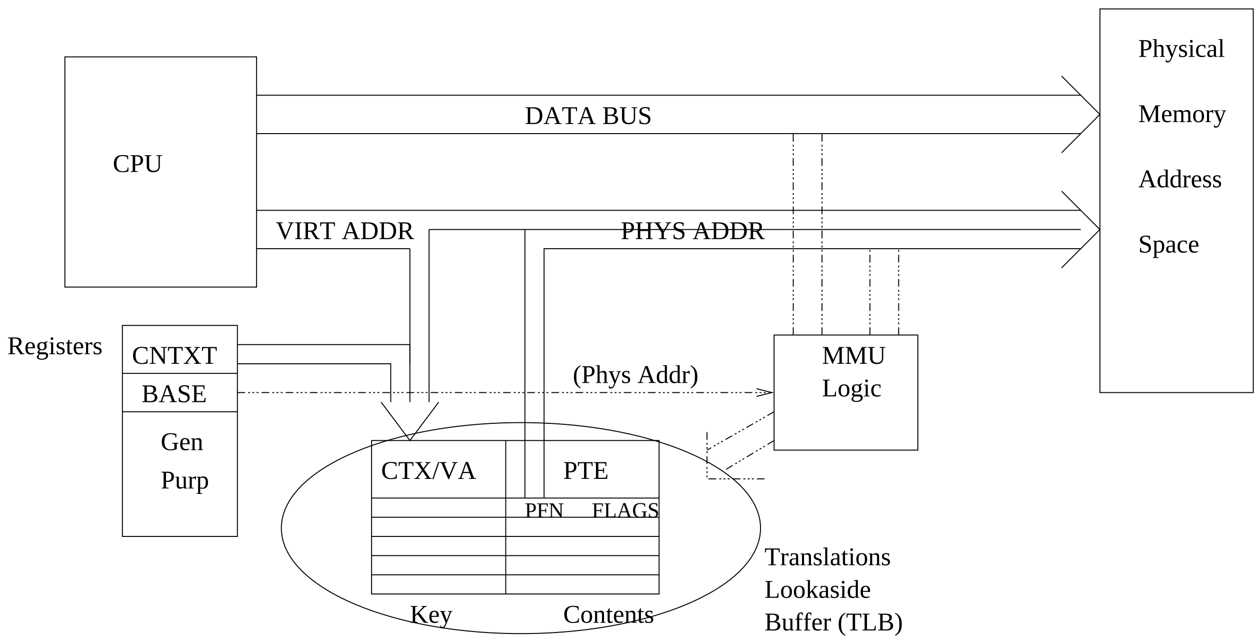
For every virtual memory access, the ATC is searched to see if a valid, cached PTE exists for the referenced virtual page number. If so, the translation is performed using the cached page table entry. Otherwise, hardware traverses the page table structure and loads the correct translation, keeping it cached for possible future re-use. The cache is of finite (and typically rather small) size, so the insertion of a new cached translation into the ATC generally eradicates an older cache entry to make room.

On some architectures (e.g. SPARC), a **context number** or **Address Space ID** is used in conjunction with the ATC/TLB to disambiguate the virtual addresses of multiple processes' translations which might be resident in the cache at once. Each process is assigned a unique context number. A CPU special register contains the current context number and it is that number, combined with the virtual page number, which keys the associative array. To perform a context switch, both the page table base address register and the context number register are changed at the same time.

We'll see that on Linux/X86-32, lacking a context tag in the TLB, it is necessary to issue special, privileged TLB flush instructions during a context switch to invalidate those mappings which will not be valid in the new context.

Because the TLB is small, performance is greatly improved when memory accesses are clustered so that, other than the first access, they "hit" the TLB cache. The kernel tries to optimize its internal data structures layout so as to keep frequently accessed data together within the same page. Thus the choice of memory address allocation within the kernel can have a profound impact on performance.

The figure below depicts the flow of address on a system with an MMU and a TLB, having the context number feature.



The UNIX user-level memory model

We have seen that the virtual address space of a process consists of a number of regions. These include:
• text region: holds the executable code
• data region: initialized global variables
• bss region: uninitialized (0-filled) global variables including dynamically-allocated variables. The ending address of this region is known as the **break address**, and can be queried or set with the `brk` system call. The break address may turn out to not fall on a page boundary, in which case the bss region really extends to the next page boundary beyond the break address. The standard C library function `malloc` ultimately uses `brk` to request additional bss memory from the kernel. Another name for the dynamically allocated part of the bss region is the `heap`. Typically, the static part of the bss region (corresponding to the declared variables) is the first part (lower memory addresses), and the heap is contiguous with it.
• stack region: function call stack and local variables. With multi-threaded processes, each thread has its own stack. Stack regions have the unusual property that they automatically grow (in the direction in which stacks usually grow, towards low memory addresses on most architectures) without an explicit system call.
• dynamically-linked libraries code
• initialized global variables for dynamically-linked libraries
• uninitialized global variables for dynamically-linked libraries
• shared memory being used for inter-process communication
• memory-mapped files

We will now see that from the kernel's standpoint, these memory region names have no particular significance. The kernel views a virtual address space as a list of regions, each of which has associated with it:
• The starting virtual address of the region (always aligned to a page boundary)
• The length of the region (a multiple of the page size), or equivalently the ending virtual address of the region.
• Bitwise flags which describe properties of the region such as the protections, or the fact that the region automatically grows towards low memory
• Backing store: If the memory region is backed by a node in the filesystem, then this mapping is described by device#/inode# and offset. Otherwise, the region is **anonymous**, and has no corresponding file.

Demand-Paging

Through virtual addressing manipulation, the operating system can maintain the illusion of having more memory on the system than there is real, physical RAM available.

Physical RAM is only mapped into a process's virtual address space when it is accessed, a form of "just in time" allocation.

Consider what happens when a process requests (e.g. through the sbrk system call) an additional 8 MB memory allocation. The operating system "grows" the bss region by recording its new size. However, no physical page numbers are assigned at this time and the page table entries are left in the NOT PRESENT state.

When the process actually performs the first read or write access to the newly allocated memory, the MMU looks up the virtual address translation, finds that the page table entry is not marked as PRESENT, and raises a **page fault interrupt**. The operating system's page fault interrupt handler examines the faulting operation and faulted virtual address. It knows which process was running at the time of the fault. Consulting its data structures for the process's address space, the operating system determines that this is indeed a valid virtual memory access. It allocates a free physical page and adjusts the page table entry accordingly to translate to this physical address (also allocating and populating any intermediate level page tables if necessary). Since the fault was in the bss region, the kernel clears the newly allocated page to 0 bytes. The interrupt handler then returns, allowing the faulting instruction to be re-tried. This time, there is a valid translation and the execution continues successfully.

We call this just-in-time allocation of page frames **Demand-Paging**.

Paging-out and Backing Store

Eventually, the sum of virtual memory allocated to all of the processes may exceed the amount of real RAM available. This is where the ACCESSED and DIRTY bits of the page table entry come into play. Periodically, the operating system kernel clears the ACCESSED bit of each virtual page in the system that has a mapping to physical memory. As long as the page is being accessed (either read or write) with some frequency, the ACCESSED bit will continue to be set periodically by hardware. Pages that have fallen into disuse will wind up with their ACCESSED bits clear.

The kernel scans for such "cold" pages and unmaps them, making their physical RAM available to other virtual pages. Unmapping the page simply means clearing the PRESENT bit in the PTE, and placing the page frame into a free list. However, when a page is freed, if the DIRTY bit is set, the contents of that page must first be saved somewhere before the page frame is placed into the free pool for reuse. This operation is known as **Paging-Out**.

The place where virtual pages reside while not mapped in is called the **Backing Store**. This is one of the properties associated with each region in the process's virtual address space. There are two choices:
• The region corresponds to a contiguous portion of a file (possibly at some page-aligned offset from the start of the file) which can be accessed through the filesystem.

• The region is **anonymous**. It does not correspond to a file in the filesystem, and the contents of the region are not **persistent.** I.e. when the process exits, nobody cares about what was in the anonymous regions. These regions use backing store called **swap space**. We can think of swap space as a pool of disk blocks which aren't normally accessible through the filesystem. Pages in an anonymous region are always zero-filled upon initial allocation.
• Memory regions that have the `MAP_PRIVATE` property (discussed later) will page-in from their backing store (file), but will page-out to swap. Thus they are a hybrid and as the program executes, some virtual pages will still correspond to the file that backs them, and others that have been dirtied will be equivalent to anonymous pages.

Paging-in, minor and major faults

At some later time, a virtual page that had been freed might be accessed by a process, resulting in a page fault, as the PRESENT bit of the page table entry would be clear. If the kernel is lucky, the physical page frame which was holding that virtual page's contents may still be valid, i.e. it was placed on the free list, but not yet re-used by another process. If so, it is a simple matter to restore the previous mapping and allow the process to go on its way. This is known as a **minor page fault**.

Otherwise, the kernel must allocate a new physical page (by re-using the oldest physical page on the free list) and arrange to fill it with the data that are associated with that virtual page. This requires reading in the data from backing store (disk), which is a potentially slow operation. The process blocks until the I/O request completes, then the process is allowed to continue, the correct mapping being in place. This is known as a **major page fault**.

Thus the pool of physical RAM can be thought of as a cache of recently-accessed virtual pages. When virtual memory demand is high, such that it can not be satisfied from physical memory alone, system performance will suffer.

Page / Protection Faults and signals

A page fault is generated when an access is attempted on a virtual page number which does not have a physical page present (or for which part of the page table structure is not current allocated), while a protection fault results from an attempted operation on a page which is disallowed by the page permissions bits, for example, writing to a read-only page. Protection faults generally result in the delivery of SIGSEGV to the faulting process (but watch out for a major exception to be discussed shortly)

When, in handling a page fault, the faulted-on virtual address falls outside of the bounds of all existing regions of the process, then the process is given a SIGSEGV. An exception is if the fault is caused by the growth of a stack. UNIX automagically grows stack regions. I.e., if the faulted address is adjacent to the top of (the lowest address) a stack region, the stack region is expanded by one page (towards low memory) and then covers the faulted address. Without this behavior, function call depth would be artificially and statically limited. The stack is located at a virtual address that is a large distance away from adjacent regions. For a collision to occur, the stack would have to grow very large. Of course, the kernel does check. Additionally, a per-process limit can be specified on stack size.

The Linux kernel uses `SIGBUS` when a page fault is the result of a valid memory access, but the kernel is unable to page-in the requested page. For example, an I/O error might occur when trying to read in the data from backing store. SIGBUS may also be used when the file which is the backing store has been truncated after having been mapped, such that the area in the file coresponding to the faulted memory address no longer exists.

Mmap

The mmap(2) system call causes a specified file to be mapped into the process's address space, creating a new region. It can also be used to create a new anonymous memory region. The first address of the mapped region will correspond to the specified offset in the file, which must be a multiple of the page size, for the specified length. The virtual address at which to map the segment can be specified explicitly, but usually it is left to the kernel to choose.

```
void *mmap(void *addr, size_t len, int prot, int flags,
                    int file_descriptor, off_t offset)
```

`prot` is a bitwise flag combination which can include the bits PROT_READ, PROT_WRITE or PROT_EXEC.

`flags` is a bitwise flag which can contain, among others:
• MAP_SHARED: Write operations to mapped region change the contents of the file.
• MAP_PRIVATE: Mutually exclusive with MAP_SHARED: Write operations to the mapped region do not change the contents of the file.
• MAP_DENYWRITE: Setting it will prevent write access to the mapped filed using the traditional write system call. See discussion at end of this unit. However, the Linux kernel currently prevents programs from setting this flag with the mmap system call, although the flag is used internally by the kernel to protect executable files.
• MAP_GROWSDOWN: The region should have the property that it automagically grows towards low memory, i.e. it is intended to be a stack region.
• MAP_ANONYMOUS: A new, anonymous region is being requested. The file_descriptor and offset parameters are ignored.

Once the new region has been created and mapped to the file, the process may close the file desciptor, and the mapping remains in effect. The mapping establishes the backing store for this region, and establishes an equivalency between the region and the corresponding area in the file.

When MAP_SHARED is specified, writing to the memory region is immediately and transparently visible to this or any other process through the traditional system calls such as `read`. Likewise, any changes to the file, e.g. through `write`, are immediately visible through the mapped region.

When the mapping is MAP_PRIVATE, something similar to Copy on Write (see below under Fork) happens. Upon the first memory write access to a page in a MAP_PRIVATE region, the association between that virtual page and the file is broken. Thus writes to a MAP_PRIVATE region do NOT cause the associated file to be modified. It is unspecified if writes to the file (e.g. via the write system call) are visible in the memory region, before the association has been broken by a write to the memory region. On the Linux 2.6 kernel, they are (but see discussion of MAP_DENYWRITE).

Pages in a MAP_PRIVATE area, once they have been written to, become anonymous pages, because the association of that particular virtual page with that particular area of the mapped file has been broken. Therefore, if they need to be paged-out, they are sent to swap; they can not be paged-out to the original file.

The mapped region may extend beyond the current end of the file. This could happen either because the `len` parameter specified is larger than the file, or because after the mapping has been established, the file size is reduced (e.g. through the `truncate` system call). If the process attempts to access memory which corresponds to just beyond the current end of file, but not beyond a page size boundary, it will see bytes with a value of 0. However, when the process tries to go further out, beyond the page boundary, even though the memory address is within the bounds of the mapped region, the kernel will be unable to satisfy the page fault, because the backing store does not exist in the filesystem. Under this condition, the kernel must deliver a **SIGBUS** to the process.

Read and write access via a file-mapped region will set the inode atime, mtime and ctime fields appropriately, but not necessarily instantaneously, for reasons which should become clear in subsequent units.

mmap'd segments are inherited by child processes in fork, using the copy-on-write mechanism for any writable regions, as described later. Multiple processes can share a mapped segment. In fact, a MAP_ANONYMOUS|MAP_SHARED region can be a useful inter-process communications tool. Note that when a region of memory has the MAP_SHARED property, copy-on-write does not apply to that region, and both processes after a fork continue to share the same physical memory for that region.

The `munmap` system call destroys a memory region and any associated mapping to a file. The `mremap` system call is used to make an existing memory region larger or smaller, or to move it to another virtual address. The reader is encouraged to read the man pages for mmap, munmap and mremap carefully.

Exec and virtual memory

During the exec system call, a new address space is initialized and the existing address space of a process is freed. The page tables themselves are freed and any physical pages that the process had mapped are placed on the free list. It is as if the process had called `munmap` for each of its regions.

The basis of the initialization of the new address space is the filename passed to the exec system call. This refers to an executable file in one of the binary executable formats which that operating system and processor type can execute. The first few bytes of the file determines the executable format, and are known as the **magic number**. We have seen that the a.out file contains the text region image, the data region initial image (variable initializers), and the initial requested size of the bss region.

In Unit 3, a highly conceptual model was presented of how a new program is loaded into memory during exec. Now that we understand mappings between files and virtual memory, we can take a more refined look and see that executable "loading" is really the establishment of a number of mmap regions.

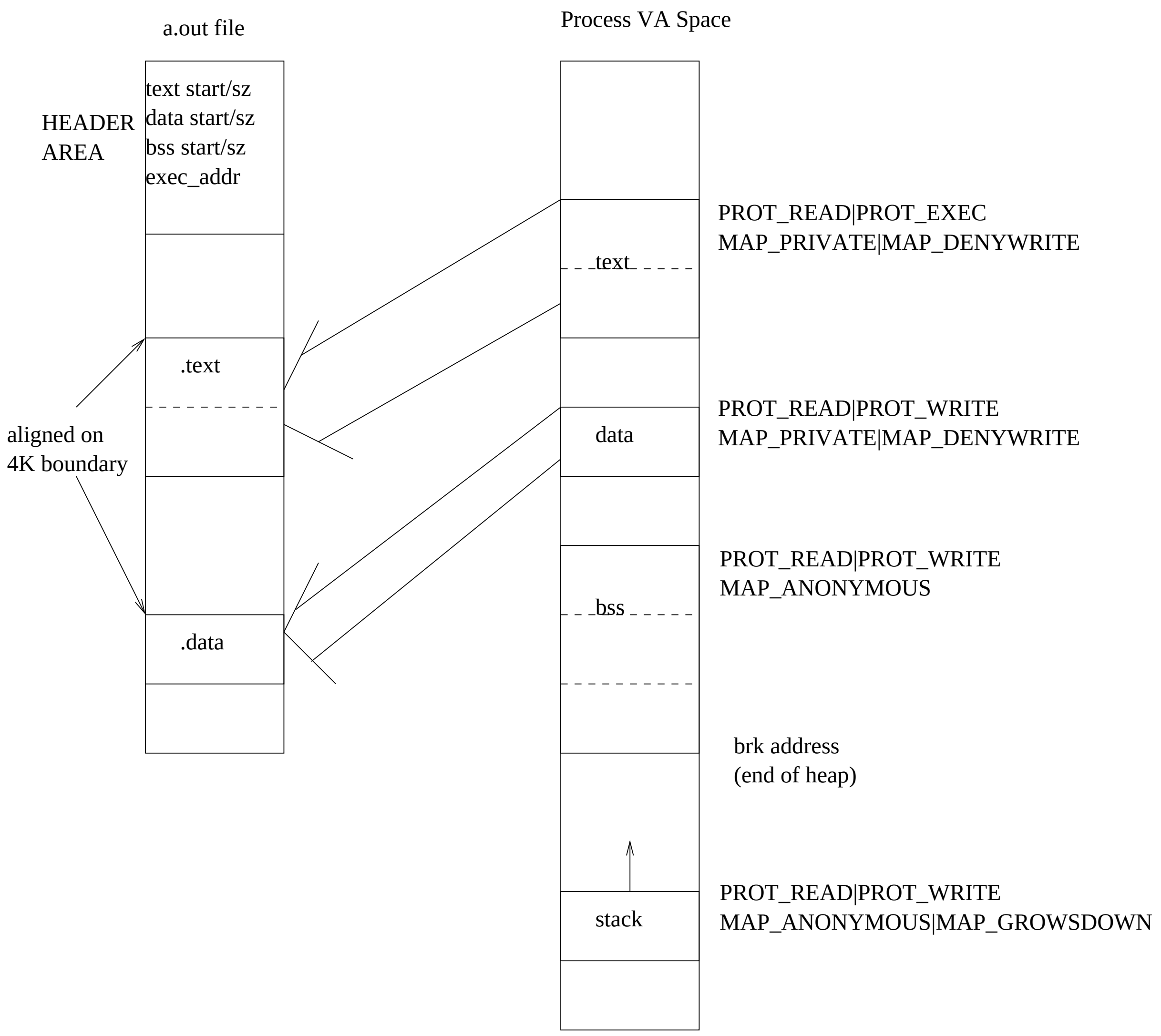
• The text region is created as if the process had used mmap to create a mapping to the area in the a.out file holding the text image. The protections of the text region are PROT_READ|PROT_EXEC. The flags are MAP_PRIVATE|MAP_DENYWRITE. Note that since page frames are never allocated to virtual address spaces until an actual access occurs, the very first thing which the new program does, upon attempting to fetch its initial opcode from memory and execute it, is to cause a page fault! The program is demand-paged in from the a.out file as needed.

• The data region is created as if by mmap with MAP_PRIVATE|MAP_DENYWRITE. The mapping is to the area of the a.out file which contains the .data section (image of initializer values). The protections are PROT_READ|PROT_WRITE. Because the mapping is MAP_PRIVATE, the program "sees" the correct initializers when they are first accessed, but writes to those variables don't cause the unpleasant consequence of modifying the a.out file.

• The bss region is created, with the size requested in the a.out header, as a MAP_ANONYMOUS|MAP_PRIVATE mapping, with PROT_READ|PROT_WRITE protections. As it is accessed, the kernel will allocate page frames, zero-filling them first. As a MAP_ANONYMOUS region, the bss will be backed by swap space.

• The stack region is created with a small initial size and with the MAP_ANONYMOUS|MAP_GROWSDOWN properties. The protections are PROT_READ|PROT_WRITE. The stack region will be grown as it is accessed, as described previously, with new page frames being zero-filled. The stack is also backed by swap space.

In the example below, a new program has been set up by exec with a text region of 2 pages, data region 1 page long, and an initial bss region of 3 pages.



Why the MAP_DENYWRITE flag?

Both text and data regions are mapped with the MAP_DENYWRITE flag set. The reason for this is the following situation: Let’s say a program is running from a particular a.out file, and then someone comes along and re-writes that executable (e.g. it is being recompiled or reinstalled). However, the running program is still mapped to the file, so what gets paged-in would be a mixture of the original contents of the file and the newer contents. Clearly this won’t work...it amounts to a corruption of the program. The DENYWRITE flag will prevent any kind of write access to the executable file, including truncation. It will fail with the error ETXTBUSY. However, one can unlink or rename the executing a.out file, and install a new version with the same name. Since the mapping is based on inode number, not pathname, the currently running process (or processes) will continue to use the old executable, while any new exec's would reference the new executable (since they evaluate a pathname, as if by **open**).

Those responsible for the development of the Linux kernel removed MAP_DENYWRITE from the list of flags that can be specified by the mmap system call (the flag is accepted but silently ignored). The reasoning went like this: User A which has read-only access to a file F could establish a read-only mmap mapping with DENYWRITE, even though they do not have write permission on F. This can create a denial-of-service attack, in that another user B, who does have write permission, would now be prevented from writing to the file.

Fork and copy-on-write

We have said that during a fork, the entire virtual address space of the parent is duplicated and the child begins life with that cloned address space. This was an oversimplification, of course. It would be very costly to copy all that memory!

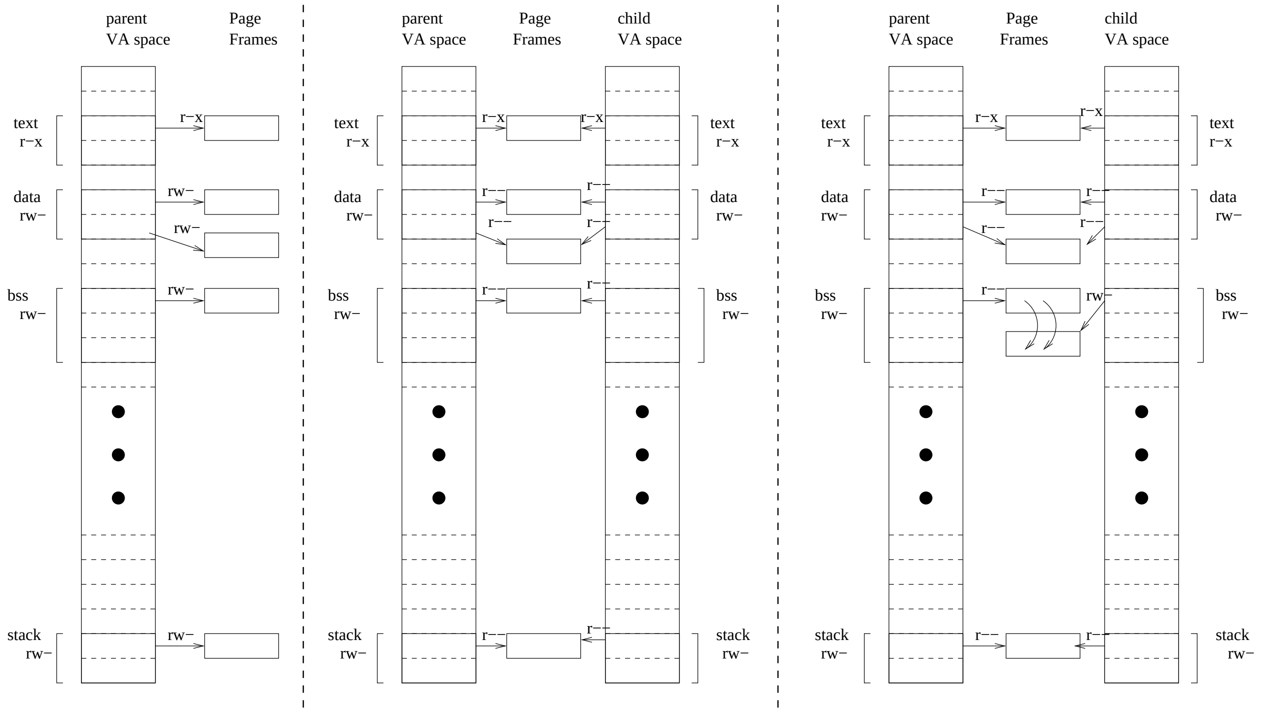
During fork, the page tables of the parent process are copied to produce the child's address space. The pages in read-only segments of the address space (e.g. the .text segment) are forever shared between parent and child process. Both processes' page tables map the same virtual addresses to the same physical pages.

Initially, the pages in the writable segments of both processes (e.g. the .bss segment) are shared. They are **marked down** to READONLY and point to the same physical memory mapping in both processes. As either of the processes attempts to write to a shared page, a protection fault occurs. The kernel, seeing the protection fault, checks the protections associated with the region in which the fault happened. Since the region has PROT_WRITE protections, the kernel knows that the access is legitimate, and does not kill the process.

Instead, the protection fault is resolved by allocating a new physical page, copying the existing data to the new page, and pointing the faulting process at the new page, leaving the other process pointing to the old page, and upgrading the faulting Page Table Entry to READWRITE. This mechanism is called **Copy On Write**, or COW, and avoids unnecessary copying of physical memory, especially considering that after most forks, the child process quickly execs.

If you are wondering why both PTEs are not upgraded to RW, it is because the page may be shared by more than two virtual address spaces. E.g. it might be an mmap-ed memory region, or there could be multiple forks involved. In this case, the remaining processes continue to share the page read-only until a write access severs this. The kernel also keeps a counter of how many PTEs point at a given page frame. If a marked-down

PTE is faulted on write, but there are no other PTEs pointing to the page frame (it is the last of what had been shared mapping) then the kernel doesn't need to do a memcpy and simply upgrades the PTE.



Appendix: How an executable file is created

In the original UNIX kernel, each program was entirely **statically-linked**, i.e. all of the code required to execute the program, and in particular all of the libraries which the program requires, are put together at compile-time into one large monolithic executable file. This practice began to change during the 1990s, and today almost all programs in almost all UNIX-derived systems are *dynamically linked*. It is still possible to compile a program which is statically linked, but this is exceptional. We will first discuss how statically linked programs are linked together, and then delve into dynamic linking.

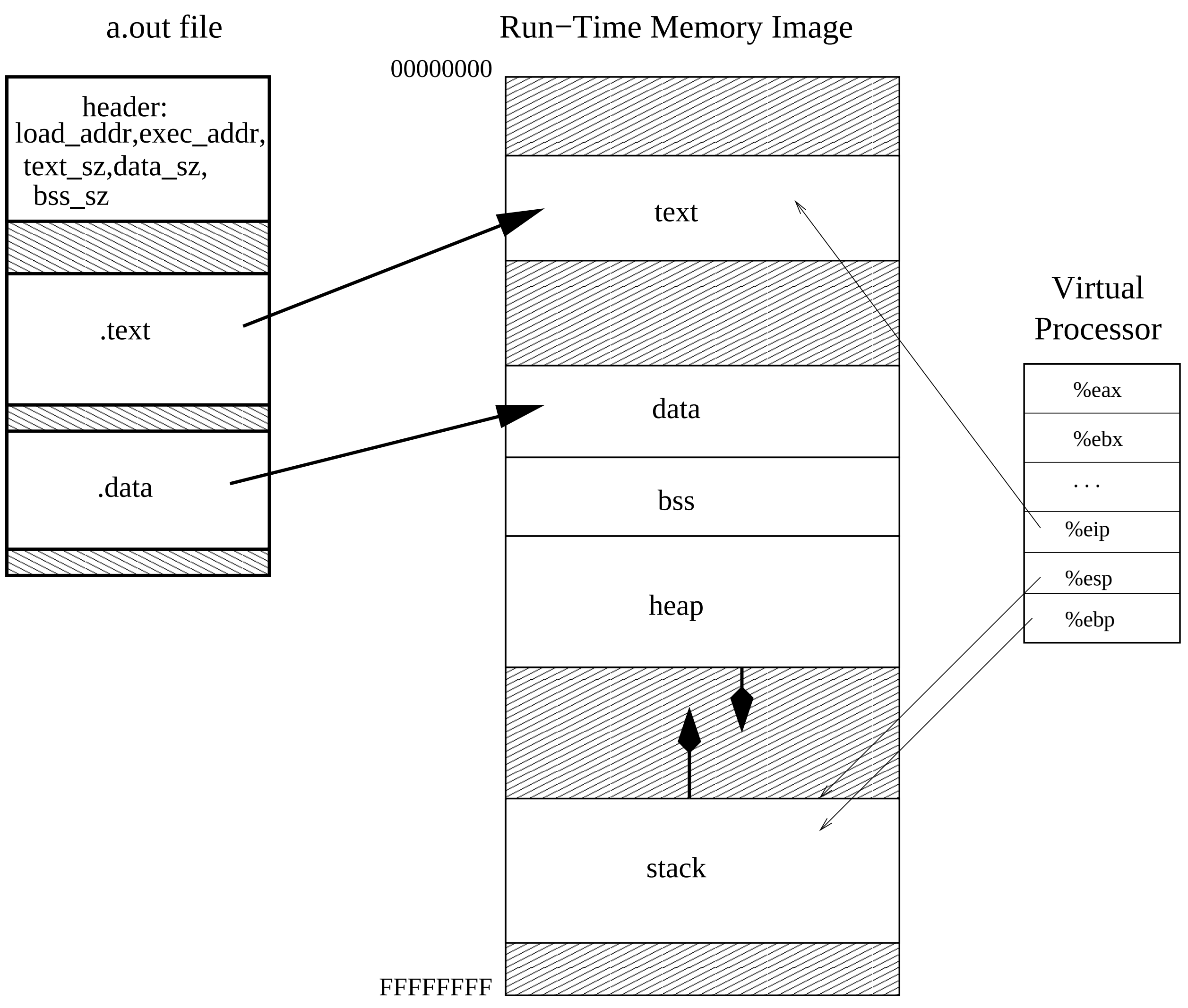
Compiling a C program into an executable file is a multi-stage process involving several tools. When one simply runs:

```
cc test.c
```

The `cc` command transparently executes the tools, detailed below, resulting in an executable file called `a.out` (if there are no fatal errors in compilation). The "a" in "a.out" stands for "Absolute", meaning that all symbolic references have been resolved. An a.out file contains pure machine code that can be directly executed by the processor. The executable file ("a.out") contains everything that the operating system needs to create a program’s initial memory configuration and begin execution. The header of the a.out identifies it to the operating system as an executable file and specifies the processor architecture on which the machine code instructions will run. A program compiled for an x86 architecture can not be executed, for example, on a SPARC processor. The header also gives the size of the text, data and bss memory requirements, as explained below.

The `a.out` file includes the literal bytes of executable code which will be loaded into the process’s `text` region. These are in a contiguous part of the `a.out` file. The header gives the offset of the beginning of the `.text` section of the `a.out`, and also the **load address**, i.e. the virtual address at which the compilation system expects this text to be loaded.

Likewise, there is a `.data` section in the `a.out` which contains an image of what the `data` region of the process will look like at program startup.


a.out file
header:
load_addr,exec_addr,
text_sz,data_sz,
bss_sz
.text
.data
Run−Time Memory Image
00000000
text
data
bss
heap
stack
FFFFFFFF
Virtual
Processor
%eax
%ebx
. . .
%eip
%esp
%ebp

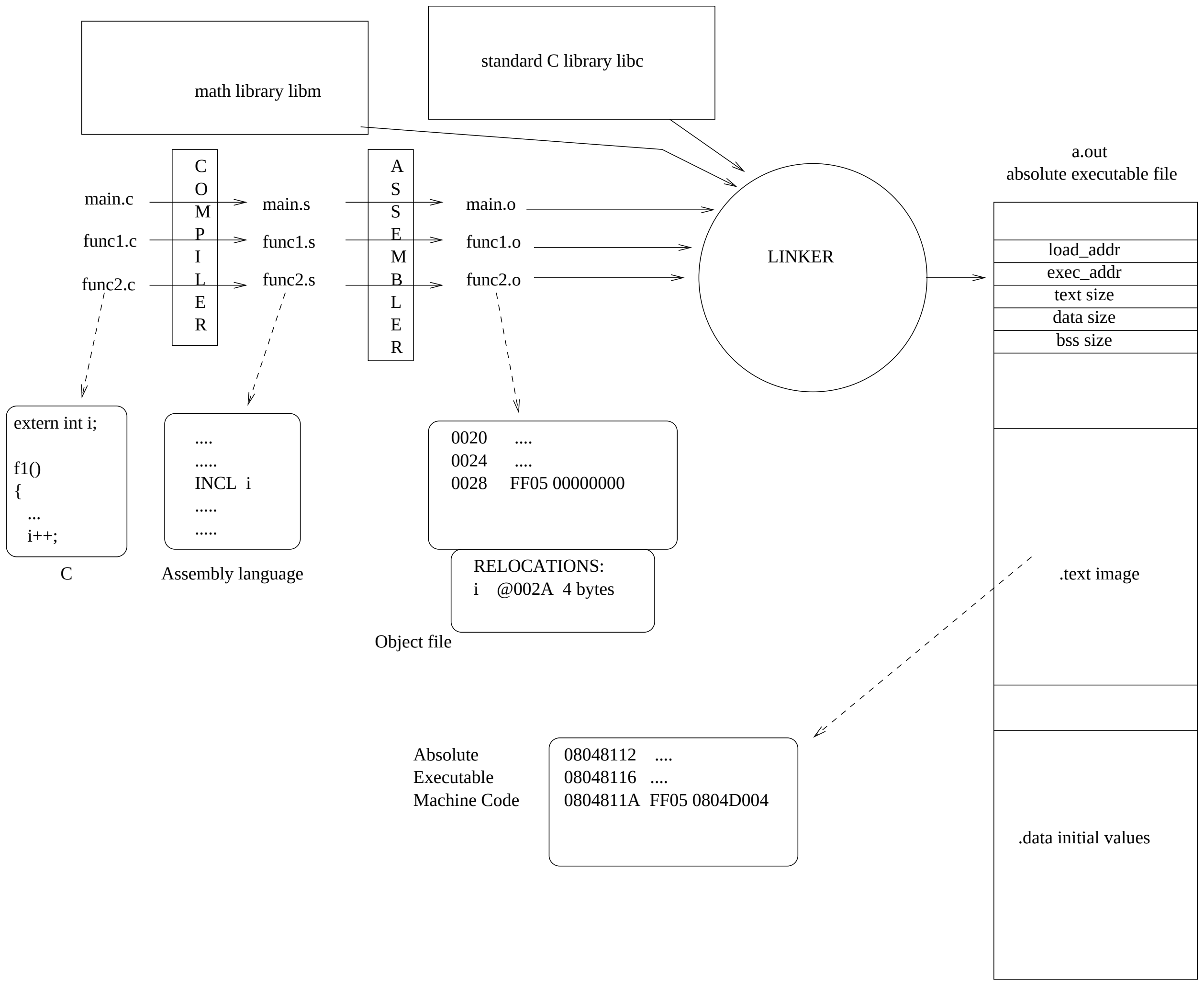
The **bss** section contains all un-initialized global variables. The C language specification states that all such variables, lacking an explicit initializer, must be initialized by the operating system or C run-time environment to 0.

```
int j=2;
int k;
main()
{
 int l;
         /*...*/
}
```

In the example above, `j` has an explicit initializer, and will be in the data segment. The value of the initializer will be found in the .data section of the a.out. `k` is uninitialized. Therefore it will reside in the bss segment of memory and will have an initial value of 0. The ISO C standard says that "If an object that has static storage duration is not initialized explicitly, it is initialized impliclitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a NULL pointer constant". This is satisfied by filling every byte of the bss region with 0, and is performed by the operating system. The variable `l` has automatic storage scope, and therefore will be found on the stack (or, if the compiler is set for heavy optimization and a pointer to `l` is never taken, it may live in a register). Automatic variables are not 0-initialized.

Behind cc-losed doors

The `cc` command is often and erroneously called "the compiler." In fact, it is a wrapper program which invokes a series of tools. Each tool transforms a specific kind of input file to a specific kind of output file. It is possible to instruct `cc` to interrupt the process at any stage, and to access and manipulate these intermediate files as needed.



The compiler proper takes C code, passing it through the macro pre-processor, and compiles it into symbolic assembly language. By using the `-S` option to `cc`, we can halt the process before the assembler and linker are invoked, and view the intermediate assembly language file, which will have a `.s` extension.

The assembler takes symbolic assembly language files (`.s`) and converts them into relocatable object files (which have a `.o` extension). An object file is similar to an executable ("a.out") file in that it contains machine code, however symbols which can not be resolved (as detailed below) are left dangling. Therefore, it is not ready to run until these dangling links are settled, possibly by linking with additional object files. By giving `cc` the `-c` option, the linker will not be run and the `.o` files will be left for subsequent examination or linking. It is also possible to invoke the assembler directly with the `as` command.

Although some compilers do not separate the assembler, it is useful to divorce it from the compiler and make it a separate utility. In this way, a common assembler program can be written and employed by compilers for different languages. For example, the GNU compilers for C, C++ and FORTRAN are all different programs, yet they all call on the `as` program for assembly. It is also useful to be able to introduce `.s` files that have been written by hand for certain performance optimizations or platform-specific features.

The linker takes multiple object (`.o`) files (or libraries of object files, with `.a` extensions, as discussed below), resolves symbol references, and creates an absolute executable file. If unresolved symbols remain at this point, the linker will not be able to create an absolute executable and an error will result.

Like the assembler, the linker is a separate tool which can be called directly as `ld`. The reason for this rather un-mnemonic two-letter name is that historically, the linker has also been called the "loader", because it was responsible for creating the "load deck" of punch cards that could then be directly run by the computer. Besides, `ln` was already taken as a command name.

When compiling C programs, it is best to allow the `cc` wrapper to invoke `ld` rather than trying to do it manually as there are a number of complex platform-specific options needed to actually create a run-able program.

The `.o` files need not have come from C source code. It is possible to distribute binary object files or libraries which can be linked against user-supplied code. It is also possible to create cross-language executables, where part of the code is written directly in low-level assembly language, or in another high-level language (with all due caution about mixed run-time environment expectations).

Separate Compilation

Very simple C programs can be contained in a single `.c` file. As the size of the project increases, it becomes extremely inefficient to do this. The tasks of pre-processing, compiling and assembling are CPU-intensive. With a single file, any change, no matter how small, requires that the entire file be recompiled.

The program can be split up into a number of smaller `.c` files. There are varying opinions on how finely to divide the program and on what basis to select which parts go into which file. Some guidelines:

1) Groups of functions which form a coherent subsystem are good candidates for including in their own file. In C++ terms, this would be a class. Some subsystems are so complex that they need to be further subdivided.

2) Whenever a single `.c` file is on the order of 1000 lines of code, it is probably getting

too big. On the other hand, having lots of files each containing one or two 10-line functions is also wasteful.

3) Code which is being incorporated from another source, or which is being worked on by a different programmer, should have its own file, or perhaps be structured as a library, to make it easier to deal with changes.

As an example, let's say our program has been divided into files `f1.c`, `f2.c` and `f3.c`. To compile this, we could run: `cc f1.c f2.c f3.c`, which would perform the entire pre-processing, compiling and assembling process on each of the three files, then invoke the linker to create the `a.out`. However, suppose we make a change to `f1.c` only. Why do all that work again for the other two files, which have not changed?

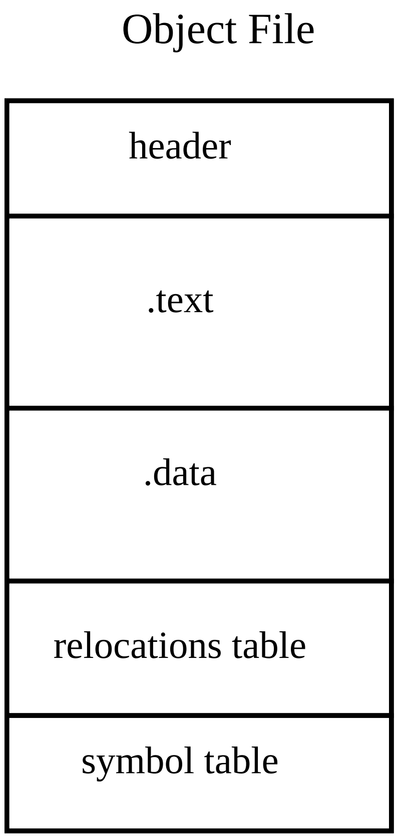
Instead, we can first run: `cc -c f1.c f2.c f3.c`, which will pre-process, compile and assemble each of the files, creating `f1.o`, `f2.o` and `f3.o`. Then we run: `cc f1.o f2.o f3.o`. The `cc` wrapper sees that we are giving it `.o` files and passes them directly to the linker. Now, if we change `f1.c`, we run `cc -c f1.c` to re-compile it alone, and then run `cc f1.o f2.o f3.o` again to link the program. This process is automated by the tool `make`, which uses dependency rules and file modification times to determine which compilation phases to invoke and on which files.

The linker vs the compiler

The job of the linker is to take one or more relocatable object files, resolve symbol references, and create an absolute executable file. The linker does not know about C's type algebra, nor should it, since the linker is independent of any particular high-level language.

Unfortunately for the student, the C language also deals in symbols, e.g. variable and function names, and this issue is sometimes difficult to separate from the symbol processing which happens at link time. Here are some things that never appear in an object file and do not concern the linker:
• C language type specifiers (such as "p is a pointer to a pointer to a function returning int and taking two int arguments")
• C language structure, union and enum definitions and typedef names.
• `goto` labels
• Internal labels that may be generated by the compiler for loops, switch statements, if statements, etc.
• automatic (local) variables. An important exception is a local variable that is declared with the `static` storage class. This variable has local scope, i.e. its name is not visible outside of the curly-braced block in which it is declared, but it lives in the same

neighborhood as global variables.



What's in a .o?

Let's see what is actually in an object file and how it gets there:

```
/* f1.c */
extern int i;

f()
{
         i=2;
}

/* f2.c */
int i;
main()
{
         i=1;
         f();
}
```

In `f1.c`, the `extern` storage class for variable *i* tells the compiler that this variable is external to that `.c` file. Therefore, the compiler does not complain when it does not see a declaration that variable. Instead, it knows that `i` is a global variable, and should be accessed by using an absolute addressing mode. Neither the compiler nor the assembler knows what that actual address will be. That is not decided until the linker puts all the object files together and assigns addresses to symbols. Therefore, the assembler must leave a "place-holder" in the object file. Likewise, in f2.o, there will be a reference to the symbol `f`.

Let us examine the assembly language files produced (using an older gcc under Linux on an x86 system):

```
**** f1.s
         .text
         .globl f
        .type    f,@function
f:
        pushl %ebp          *These instructions set up
        movl %esp,%ebp               *the stack frame pointer
        movl $2,i
        leave                        *Restore frame pointer
        ret
.Lfe1:
        .size    f,.Lfe1-f

**** f2.s
         .text
         .globl main
        .type    main,@function
main:
        pushl %ebp          *These instructions set up
        movl %esp,%ebp               *the stack frame pointer
         movl $1,i
        call f
        leave                        *Restore frame pointer
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .comm   i,4,4
```

The assembler directive `.text` tells the assembler that it is assembling opcodes to go in the .text section of the object file. The `.globl` directive will cause the assembler to export the associated symbol as a defining instance. `.type` is used to pass along information into the object file as to the type of the symbol. Please note that it has nothing to do with the C language notion of type. Symbol types may either be functions or variables. The linker is able to catch gross errors such as if `f` were defined as a variable in `f1.c` instead of a function. The `.size` directive calculates the size of the function by subtracting the value of the symbol representing the first instruction of the function (e.g. `main`) from a placeholder assembler symbol (e.g. `.Lfe1`) representing the end of the function. Note that the CALL to function `f` is done symbolically, as is the assignment into global variable `i`.

The resulting object file is similar to an a.out file, however all addresses are relative. In addition, the object file will have a section known as the **symbol table** containing an entry for every symbol that is either defined or referenced in the file, and another section called the **relocations table**, described below.

Now, let us view the .text section of `f1.o` (the listing below was re-formatted from the output of objdump -d):

```
Offset Opcodes                          Disassembly
0000   55                           pushl %ebp
0001   89E5                         movl %esp,%ebp
0003   C7050000000002000000         movl $2,0
000D   C9                           leave
000E   C3                           ret
```

First, note that the offset of the first instruction is 0. Obviously, this can not be a valid memory address. All offsets in object files are relative to the object file. It isn't until the linker kicks in that symbols gain absolute, usable addresses.

Next note that in the instruction beginning at offset 0003, the constant 2 is moved to memory address 0. We know from examining the corresponding assembly language input file that this is the instruction that moves 2 into variable `i`. The assembler has left a place-holder of 00000000 in the object file where the linker will have to fill in the actual 32-bit address of symbol `i` once that is known. C705 is the x86 machine language opcode for the MOVL instruction where the source addressing mode is Immediate and the destination mode is Absolute. The next 4 bytes are the destination address and the final 4 bytes of the instruction are the source operand (which is in Intel-style, or "little-endian" byte order).

Here is the dump of f2.o:

```
Offset Opcodes                           Disassembly
0000   55                          pushl %ebp
0001   89E5                        movl %esp,%ebp
0003   C7050000000001000000        movl $1,0
000D   E8FCFFFFFF                  call 000E
0012   C9                          leave
0013   C3                          ret
```

Note that the placeholder in the CALL instruction is FFFFFFFC, but the disassembler decodes that as 000E. This is because the CALL instruction uses a Program Counter Relative addressing mode. The address to which execution jumps is the operand in the instruction added to the value of the Program Counter register *corresponding to the byte beyond the last byte of the CALL instruction*. In two's complement, FFFFFFFC is -4, therefore the CALL appears to be to the instruction at offset 000E. Of course, all of this is meaningless since it is just a placeholder that will be overwritten by the linker. Nonetheless, it reminds us that there are different **Relocation Types** depending on the addressing mode being used.

The symbol and relocation tables for the two object files will look something like this:

```
f1.o:
                   SYMBOL TABLE
NAME    TYPE         VALUE
f       func         0 in .text
i       variable     [reference]
                   RELOCATIONS TABLE
SYMBOL=i   OFFSET=0005  LENGTH=4  RELTYPE=ABSOLUTE



f2.o:
                   SYMBOL TABLE
NAME    TYPE         VALUE
f       function     [reference]
i       variable     8 in .bss
main    function     0 in .text
                   RELOCATIONS TABLE
SYMBOL=i   OFFSET=0005  RELTYPE=ABSOLUTE
SYMBOL=f   OFFSET=000E  RELTYPE=PCRELATIVE
```

What ld does

The first task of `ld` is to take inventory of all of the `.o` files being presented to it. It loads in the symbol tables from all of the object files to create a unified symbol table for the entire program. There is only one global namespace for all linker symbols. This might be considered a deficiency. Let's say we have a module `foo.c` that defines a function called `calculate`. If we attempt to incorporate that module into a program written by someone else, they may have also made a function called `calculate`. It is, after all, a common name.

If there is more than one defining instance of a symbol, i.e. if a symbol is *multiply-defined,* this is generally a fatal error. Consider what would happen if a programmer accidentally included two versions of function `f` above in two different `.c` files. When `f` is called somewhere in the program, which version should be called?

To ameliorate this problem of flat global linker namespace, a convention exists that one should prepend to one's global variable and function names a reasonably unique prefix. Therefore, we might call our function `foo_calculate`. This is less likely to conflict with another name. It isn't a perfect solution, but it works fairly well in reality.

In languages like C++, which takes name overloading to an extreme, the identified names used in the source code are **name-mangled** to be non-conflicting when placed into the global linker namespace.

Global variable and function names that are intended to remain private to the `.c` file in which they are declared should be protected with the `static` storage class (see below). `static` symbols still require the assistance of the linker to be relocated. However, the use of `static` causes the compiler and the assembler to flag that symbol as a LOCAL symbol. The linker will then enter the symbol into a private namespace just for the corresponding object file, and the symbol will never conflict with symbols from other object files.

There are rare cases where it is useful to deliberately redefine a symbol. For instance, we may need to change how a piece of code, available only in library or object file form, calls another function. This all falls under the heading of "wild and crazy ld tricks" and will not be discussed further. Just remember that any duplicate symbol definitions are generally wrong.

There is one antiquated exception which `ld` still honors: the so-called COMMON block, which is a hangover from FORTRAN. If there are multiple defining instances of a variable in the bss section (i.e. a global variable without an initializer), `ld` will look the other way. This is because there really isn't a fatal conflict. As long as the definitions all give the same size and all agree that it is a bss symbol, there's no worry about which definition is the correct one. All are equivalent. So, if one accidentally declares a global uninitialized variable twice, the program will compile fine. It is sloppy, however.

Frequently symbols have a defining instance, but they are never actually referenced. For example, the programmer may write a function, but never call it, or declare a global variable, yet never use it in an expression. The linker doesn't care about this.

However, it cares deeply about the opposite case: a symbol that is referenced (by appearing in a relocations table) but which has no defining instance. Such a situation makes it impossible for the linker to complete its task of creating an absolute executable file with no dangling references. Therefore, it will stop with a fatal error, reporting the undefined symbol and the object file or files in which it is referenced.

Once all of the symbol definitions and references are resolved, `ld` will assemble the `a.out` file by concatenating all of the text sections of all of the object files, forming the single `.text` section of the `a.out`. In doing so, the text sections are relocated. An instruction which was at offset 10 in a particular object file's text section may now be at offset 1034 in the `a.out`.

Furthermore, `ld` has a concept of the `load address` of the program, i.e. the memory address at which the first byte of the `.text` section of the `a.out` will be loaded. This knowledge is part of configuring `ld` for a particular operating system and processor type, and is not normally something that the programmer needs to worry about. The load address is also placed in the `a.out` file so the operating system is sure to load the program at the address for which it was linked. Knowing the size of each object file's `.text` section, `ld` can calculate the absolute address that it will occupy in the final image, and can thus complete the symbol table, assigning absolute values to each text symbol.

A similar process is undertaken for the `data` and `bss` sections. `ld` assigns an absolute address to each `data` or `bss` symbol, based on the configured starting address of the `data` and `bss` memory regions. In the case of `data` symbols, the initializers contained in the individual object files are concatenated, forming the `.data` section of the `a.out`, which will thus contain the byte-for-byte image of what that section will look like when the program starts. For `bss` symbols, there are obviously no initializers. `ld` keeps track of the total number of `bss` bytes needed, and places that information into the `a.out` header so the operating system can allocate the memory when the program is loaded. During this process, each `data` or `bss` symbol is assigned an absolute address in the symbol table.

Now `ld` concatenates the various object files, processing the relocation records, replacing each "placeholder" with the actual, absolute address that the associated symbol is now known to have. In the case of Program-Counter-Relative relocation types, the proper offset is calculated with respect to the absolute address of the placeholder in question.

Libraries

When you write a C program, you expect certain functions, such as `printf`, to be available. These are supplied in the form of a **library**. A library is basically a collection of `.o` files, organized together under a common wrapper format called a `.a` file. It is similar to a "tar" or "zip" archive (although no compression is provided).

Convention is that a library *ZZZ* is contained in the library file: `libZZZ.a`. Using the `-l` option to `cc` tells the `cc` program to ask the linker to link with the specified library. For example: `cc myprog.c -lm` asks for the system library file `libm.a` (the math library) to be additionally linked. By default, `cc` always links the standard C library `libc.a`. These libraries are located in a system directory, typically `/usr/lib`.

Although a library embodies a collection of object files, the behavior when linking to a library is slightly different than if one just linked in all of the object files individually. In the latter case, the `.text` and `.data` sections of each object file would wind up in the `a.out`, regardless of whether or not anything in a particular object file was actually used. With a library, `ld` builds a symbol table of all the object files in the library, and then only selects those object files that are actually needed for inclusion into the executable.

Dynamic Linking

When dynamic (shared) libraries are used, there are two parts to the linkage process. At compile time, `ld` links against the dynamic library (which has a `.so` extension in UNIX) for the purpose of learning which symbols are defined by it. However, none of the code or data initializers from the dynamic library are actually included in the a.out file. Instead, `ld` records which dynamic libraries were linked against.

At execution time, the second phase takes place before the `main` gets invoked. A small helper program called `ld.so` (typically, actual names vary depending on operating system build) is loaded into the process address space by the kernel and executed. It is provided with the list of dynamic libraries. The dynamic loader finds each of the required libraries, provides new memory regions to hold the text and data of each library, and conceptually loads them into those regions (in fact, it performs `mmap` system calls to create the regions and map them to the shared library file).

Some additional magic is required to link the static and the dynamic portions of the executable. This aspect is extremely specific to the operating system and processor architecture. In general, for functions which are defined in a shared library, dummy stubs are provided in what is known as a **Procedure Linkage Table**. These stubs contain jumps to addresses which are filled in by the dynamic linker. This allows the static code to make function calls without being dynamic-aware. Likewise, a table known as the **Global Offset Table** provides transparent access to any global variables which a shared library exports. The dynamic libraries themselves need to be compiled in such as way that they are capable of being loaded at any virtual address. This is done with the `-fpic` (position independent code) flag to the compiler.

There is a slight performance penalty when using dynamic libraries. In additional to the delay in executing the program imposed by needing to load and link the libraries, calls to functions in the dynamic library require an additional branch, and accesses to global symbols from within the shared library must use a position-independent addressing method, which generally requires an indirect addressing mode.

One down-side of dynamic linking is that a missing dynamic library (".so" files in UNIX terms, or "DLL"s in Windows-speak) will prevent a program from running, even if it was complete and correct at compile time. However, this disadvantage is more than outweighed by the reduction in executable size and the convenience of being able to correct errors in system libraries on the fly, without having to locate and recompile each and every executable based on the faulty library.

The program `ldd` can be used to debug dynamic library resolution problems. This example shows the dynamic libraries that are needed for `ls` on a 32-bit Linux system:

```
$ ldd /bin/ls
        linux-gate.so.1 =>  (0xb77c4000)
        librt.so.1 => /lib/librt.so.1 (0xb77b6000)
        libacl.so.1 => /lib/libacl.so.1 (0xb77ad000)
        libc.so.6 => /lib/libc.so.6 (0xb765b000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7642000)
        /lib/ld-linux.so.2 (0xb77c5000)
        libattr.so.1 => /lib/libattr.so.1 (0xb763b000)
```

The numbers in parentheses are the addresses at which each of the dynamic libraries would load if the program were actually being executed.

Under certain circumstances, such as security-sensitive applications, it may be appropriate to use static linkage. This ensures that the program will always be able to run, and will always run with exactly the expected code.