What is some existing documentation on Linux memory management?

Ulrich Drepper (the ex-glibc maintainer) wrote an article series called "What every programmer should know about memory":

Part 1: http://lwn.net/Articles/250967/
Part 2: http://lwn.net/Articles/252125/
Part 3: http://lwn.net/Articles/253361/
Part 4: http://lwn.net/Articles/254445/
Part 5: http://lwn.net/Articles/255364/
Part 6: http://lwn.net/Articles/256433/
Part 7: http://lwn.net/Articles/257209/
Part 8: http://lwn.net/Articles/258154/
Part 9: http://lwn.net/Articles/258188/

Mel Gorman's book "Understanding the Linux Virtual Memory Manager" is available online:

http://kernel.org/doc/gorman/

What is virtual memory?

Virtual memory provides a software-controlled set of memory addresses, allowing each process to have its own unique view of a computer's memory.

Virtual addresses only make sense within a given context, such as a specific process. The same virtual address can simultaneously mean different things in different contexts.

Virtual addresses are the size of a CPU register. On 32 bit systems each process has 4 gigabytes of virtual address space all to itself, which is often more memory than the system actually has.

Virtual addresses are interpreted by a processor's Memory Management Unit (mmu), using data structures called page tables which map virtual address ranges to associated content.

Virtual memory is used to implement lazy allocation, swapping, file mapping, copy on write shared memory, defragmentation, and more.

For details, see Ulrich Drepper's "What every programmer should know about memory, Part 3: Virtual Memory":

http://lwn.net/Articles/253361/

What is physical memory?

Physical memory is storage hardware that records data with low latency and small granularity. Physical memory addresses are numbers sent across a memory bus to identify the specific memory cell within a piece of storage hardware associated with a given read or write operation.

Examples of storage hardware providing physical memory are DIMMs (DRAM), SD memory cards (flash), video cards (frame buffers and texture memory), network cards (I/O buffers), and so on.

Only the kernel uses physical memory addresses directly. Userspace programs exclusively use virtual addresses.

For details, see the ARS Technica DRAM Guide:

http://arstechnica.com/paedia/r/ram_guide/ram_guide.part1-1.html
http://arstechnica.com/paedia/r/ram_guide/ram_guide.part2-1.html
http://arstechnica.com/paedia/r/ram_guide/ram_guide.part3-1.html

And Ulrich Drepper's "What every programmer should know about memory, Part 1":

    http://lwn.net/Articles/250967/

What is a Memory Management Unit (MMU)?

The memory management unit is the part of the CPU that interprets virtual addresses. Attempts to read, write, or execute memory at virtual addresses are either translated to corresponding physical addresses, or else generate an interrupt (page fault) to allow software to respond to the attempted access.

This gives each process its own virtual memory address range, which is limited only by address space (4 gigabytes on most 32-bit system), while physical memory is limited by the amount of available storage hardware.

Physical memory addresses are unique in the system, virtual memory addresses are unique per-process.

What are page tables?

Page tables are data structures which containing a process's list of memory mappings and track associated resources. Each process has its own set of page tables, and the kernel also has a few page table entries for things like disk cache.

32-bit Linux systems use three-level tree structures to record page tables. The levels are the Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entry (PTE). (64-bit Linux can use 4-level page tables.)

For details, see:

    http://en.wikipedia.org/wiki/Page_table

What are memory mappings?

A memory mapping is a set of page table entries describing the properties of a consecutive virtual address range. Each memory mapping has a start address and length, permissions (such as whether the program can read, write, or execute from that memory), and associated resources (such as physical pages, swap pages, file contents, and so on).

Creating new memory mappings allocates virtual memory, but not physical memory (except for a small amount of physical memory needed to store the page table itself). Physical pages are attached to memory mappings later, in response to page faults. Physical memory is allocated on-demand by the page fault handler as necessary to resolve page faults.

A page table can be thought of as a description of a set of memory mappings. Each memory mapping can be anonymous, file backed, device backed, shared, or copy on write.

The mmap() system call adds a new memory mapping to the current process's page tables. The munmap() system call discards an existing mapping.

Memory mappings cannot overlap. The mmap() call returns an error if asked to create overlapping memory mappings.

Virtual address ranges for which there is no current memory mapping are said to be "unmapped", and attempts to access them generate a page fault which cannot be handled. The page fault handler sends a Segmentation Violation signal (SIGSEGV) to the program on any access to unmapped addresses. This is generally the result of following a "wild pointer".

Note that by default, Linux intentionally leaves the first few kilobytes (or even megabytes) of each process's virtual address space unmapped, so that attempts to dereference null pointers generate an unhandled page fault resulting in an immediate SIGSEGV, killing the process.

For details, see the Single Unix Specification version 4 entry for the mmap() system call:

http://www.opengroup.org/onlinepubs/9699919799/functions/mmap.html

And the glibc mmap() documentation:

http://www.gnu.org/s/libc/manual/html_node/Memory_002dmapped-I_002fO.html

What are shared pages?

Multiple Page Table Entries can map the same physical page. Access through these virtual addresses (often in different processes) all show the same contents, and changes to it are immediately visible to all users. (Shared writable mappings are a common high-performance inter-process communication mechanism.)

The number of PTEs mapping each physical page is tracked, and when the reference count falls to zero the page is moved to the free memory list.

What is an anonymous mapping?

Anonymous memory is a memory mapping with no file or device backing it. This is how programs allocate memory from the operating system for use by things like the stack and heap.

Initially, an anonymous mapping only allocates virtual memory. The new mapping starts with a redundant copy on write mapping of the zero page. (The zero page is a single page of physical memory filled with zeroes, maintained by the operating system.) Every virtual page of the anonymous mapping is attached to this existing prezeroed page, so attempts to read from anywhere in the mapping return zeroed memory even though no new physical memory has been allocated to it yet.

Attempts to write to the page trigger the normal copy-on-write mechanism in the page fault handler, allocating fresh memory only when needed to allow the write to proceed. (Note, prezeroing optimizations change the implementation details here, but the theory's the same.) Thus "dirtying" anonymous pages allocates physical memory, the actual allocation call only allocates virtual memory.

Dirty anonymous pages can be written to swap space, but in the absence of swap they remain "pinned" in physical memory.

Anonymous mappings may be created by passing the MAP_ANONYMOUS flag to mmap().

What is a file backed mapping?

File backed mappings mirror the contents of an existing file. The mapping has some administrative data noting which file to map from, and at which offset, as well as permission bits indicating whether the pages may be read, written, or executed.

When page faults attach new physical pages to such a mapping, the contents of those pages is initialized by reading the contents of the file being mapped, at the appropriate offset for that page.

These physical pages are usually shared with the page cache, the kernel's

disk cache of file contents. The kernel caches the contents of files when the page is read, so sharing those cache pages with the process reduces the total number of physical pages required by the system.

Writes to file mappings created with the MAP_SHARED flag update the page cache pages, making the updated file contents immediately visible to other processes using the file, and eventually the cache pages will be flushed to disk updating the on-disk copy of the file.

Writes to file mappings created with the MAP_PRIVATE flag perform a copy on write, allocating a new local copy of the page to store the changes. These changes are not made visible to other processes, and do not update the on-disk copy of the file.

Note that this means writes to MAP_SHARED pages do not allocate additonal physical pages (the page was already faulted into the page cache by the read, and the data can be flushed back to the file if the physical page is needed elsewhere), but writes to MAP_PRIVATE pages do (the copy in the page cache and the local copy the program needs diverge, so two pages are needed to store them, and flushing the page cache copy back to disk won't free up the local copy of the changed contents).

What is the page cache?

The page cache is the kernel' cache of file contents. It's the main user of virtual memory that doesn't belong to a specific process.

See "What is a file backed mapping" and "What is free memory" for more info.

What is CPU cache?

The CPU cache is a very small amount of very fast memory built into a processor, containing temporary copies of data to reduce processing latency.

The L1 cache is a tiny amount of memory (generally between 1k and 64k) wired directly into the processor that can be accessed in a single clock cycle. The L2 cache is a larger amount of memory (up to several megabytes) adjacent to the processor, which can be accessed in a small number of clock cycles. Access to uncached memory (across the memory bus) can take dozens, hundreds, or even thousands of clock cycles.

(Note that latency is the issue CPU cache addresses, not throughput. The memory bus can provide a constant stream of memory, but takes a while to start doing so.)

For details, see Ulrich Drepper's "What every programmer should know about memory, Part 2":

    http://lwn.net/Articles/252125/

What is a Translation Lookaside Buffer (TLB)?

The TLB is a cache for the MMU. All memory in the CPU's L1 cache must have an associated TLB entry, and invalidating a TLB entry flushes the associated cache line(s).

The TLB is a small fixed-size array of recently used pages, which the CPU checks on each memory access. It lists a few of the virtual address ranges to which physical pages are currently assigned.

Accesses to virtual addresses listed in the TLB go directly through to the associated physical memory (or cache pages) without generating page faults (assuming the page permissions allow that category of access). Accesses to virtual addresses not listed in the TLB (a "TLB miss") trigger a page table lookup, which is performed either by hardware, or by the page fault

handler, depending on processor type.

For details, see:

    http://en.wikipedia.org/wiki/Translation_lookaside_buffer

1995 interview with Linus Torvalds describing the i386, PPC, and Alpha TLBs:

    http://www.linuxjournal.com/article/36

What is a page fault handler?

A page fault handler is an interrupt routine, called by the Memory Management Unit in response an attempt to access virtual memory which did not immediately succeed.

When a program attempts to read, write, or execute memory in a page that hasn't got the appropriate permission bits set in its page table entry to allow that type of access, the instruction generates an interrupt. This calls the page fault handler to examines the registers and page tables of the interrupted process and determine what action to take to handle the fault.

The page fault handler may respond to a page fault in three ways:

1) The page fault handler can resolve the fault by immediately attaching a page of physical memory to the appropriate page table entry, adjusting the entry, and resuming the interrupted instruction. This is called a "soft fault".

2) When the fault handler can't immediately resolve the fault, it may suspend the interrupted process and switch to another while the system works to resolve the issue. This is called a "hard fault", and results when an I/O operation most be performed to prepare the physical page needed to resolve the fault.

3) If the page fault handler can't resolve the fault, it sends a signal (SIGSEGV) to the process, informing it of the failure. Although a process can install a SIGSEGV handler (debuggers and emulators tend to do this), the default behavior of an unhandled SIGSEGV is to kill the process with the message "bus error".

A page fault that occurs in an interrupt handler is called a "double fault", and usually panics the kernel. The double fault handler calls the kernel's panic() function to print error messages to help diagnose the problem. The process to be killed for accessing memory it shouldn't is the kernel itself, and thus the system is too confused to continue.

  http://en.wikipedia.org/wiki/Double_fault

A triple fault cannot be handled in software. If a page fault occurrs in the double fault handler, the machine immediately reboots.

  http://en.wikipedia.org/wiki/Triple_fault

How does the page fault handler allocate physical memory?

The Linux kernel uses lazy (on-demand) allocation of physical pages, deferring the allocation until necessary and avoiding allocating physical pages which will never actually be used.

Memory mappings generally start out with no physical pages attached. They define virtual address ranges without any associated physical memory. So malloc() and similar allocate space, but the actual memory is allocated later by the page fault handler.

Virtual pages with no associated physical page will have the read, write, and execute bits disabled in their page table entries. This causes any access to that address to generate a page fault, interrupting the program and calling the page fault handler.

When the page fault handler needs to allocate physical memory to handle a page fault, it zeroes a free physical page (or grabs a page from a pool of prezeroed pages), attaches that page of memory to the Page Table Entry associated with the fault, updates that PTE to allow the appropriate access, and resumes the faulting instruction.

Note that implementing this requires two sets of page table flags for read, write, execute, and share. The VM_READ, VM_WRITE, VM_EXEC, and VM_SHARED flags (in linux/mm.h) are used by the MMU to generate faults. The VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, and VM_MAYSHARE flags are used by the page fault handler to determine whether the attempted access was legal and thus the fault handler should adjust the PTE to resolve the fault and allow the process to continue.

How does fork work?

The fork() system call creates a new process by copying an existing process. A new process is created with a copy of the page tables that calls fork(). These page tables are all copy on write mappings sharing the existing physical pages between parent and child.

How does exec work?

The exec() system call executes a new file in the current process context. It blanks the process's current page table, discarding all existing mappings, and replaces them with a fresh page table containing a small number of new mappings, including an executable mmap() of the new file passed to the exec() call, a small amount of administrative space containing the environment variables and command line arguments passed into the new program, a new process stack, and so on.

The normal way to launch a new process in Unix-like systems is to call fork(), followed immediately by a call to exec() in the new process. Thus fork() copies the parent process's existing process's memory mappings into the new process, then exec() immediately discards them again. Because these were shared mappings, the fork() allocates a lot of virtual space but consumes very few new physical pages.

How do shared libraries work?

Only statically linked executables are executed directly. Shared libraries are executed by the dynamic linker (either ld-linux.so.2 or ld-uClibc.so.0), which despite the name is a statically linked executable that works a bit like #!/bin/sh or #!usr/bin/perl in shell scripts. It's the binary that's launched to run this program, and the path to this program is fed to it as its first argument.

The dynamic linker mmap()s the executable files, and any shared libraries it needs, using the MAP_PRIVATE flag. This allows it to write to those pages to perform the dynamic linking fixups allowing the executable's calls out to the shared library code to connect. (It calls mprotect() to set the pages read only before handing control over to the linked executable.) The dynamic linker traces through various lists of calls in the program's ELF tables, looks up the appropriate function pointer for each one, and writes that pointer to the call site in the memory mapping.

Pages the dynamic linker writes to are essentially converted to anonymous pages by breaking the copy-on-write. These new "dirtied" pages allocate physical memory visible only to this process. Thus keeping to a minimum

the number of dirtied pages allows the rest to remain shared, and thus saves memory.

Shared libraries are normally compiled with the -fpic flag (Position Independent Code), which creates an object table containing all the references to external calls to data and functions. Instead of reaching out and touching the shared objects directly, the code bounces off this table. This makes the code slightly larger by inserting an extra jump or extra load, but the advantage is that all the external references modified by the linker are grouped together into a small number of pages.

So the binary is slightly larger, but more of the pages are shared since the number of physical pages dirtied by the dynamic linker is smaller each time the shared object is used.

Normally only shared libraries are compiled this way, but some programs (such as busybox) which the system expects to run many instances of may also benefit from the increased sharing more than they suffer from the increased size.

Note that statically linked programs have no fixups applied to them, and thus no private executable pages. Every page of their executable mapping remains shared. They also spawn faster, since there's no dynamic linker performing fixups. Thus in some circumstances, static linking is actually more efficient than dynamic linking. (Strange but true.)

How does copy on write work?

If two page table entries point to the same physical page, the contents of that page show up in both locations when read. A reference counter associated with the page tracks how many page table entries point to that page. Each of those page table entries has read permission (but not write permissions) for the page.

Attemps to write to the page generate a page fault. The page fault handler allocates a new physical page, copies the contents of the shared page into the new page, attaches the new page to the faulting page table entry, sets the updated PTE writeable, decrements the count on the old shared page (possibly removing its shared status if the reference count falls to 1), and resumes the faulting process allowing the write to go through to the new nonshared copy of the page.

This is a form of lazy allocation, deferring memory allocation until the new memory is actually used.

What are clean pages?

Clean pages have copies of their data stored elsewhere, such as in swap space or in a file. Thus the physical memory storing that information may be reclaimed an reused elsewhere by detaching the physical page from the associated Page Table Entry. When the page's contents are needed again, a new physical page may be allocated and its contents read from the stored copy.

What are active pages?

Active pages are page table entries that have associated physical pages which have been used recently.

The system can track active pages by removing the read, write, and execute bits from page table entries but leaving the associated physical page still attached, then taking a soft fault the next time that page is accessed. The fault handler can cheaply switch the appropriate access bit back on and resume the faulting instruction, thereby recording that the page is currently in use.

Pages which are active are poor candidates for page stealing, even if they are clean, because the process using them will quickly fault a new physical page back in again if the current one is reclaimed.

Note that reading a lot of filesystem data marks cache pages as active, since they're recently used. This not only causes the page cache to allocate lots of physical pages, but prevents those pages from being reclaimed since they were used more recently than other pages in the system.

What are free pages?

When a page's reference count goes to zero, the page is added to the free list inside the kernel. These free pages are not currently used for any purpose, and are essentially wasted until some use is found for them.

A freshly booted system starts with lots of free pages. Free pages also occur when processes exit(), or when munmap() discards a mapping and the private pages associated with it.

The kernel tries to minimize the number of free pages in the system. Instead, it tries to find uses for these pages which improve performance of existing processes, but which leaves them easily reclaimable if the memory is needed for another purpose.

For example, the page cache keeps around copies of file contents long after they were last accessed, potentially avoiding an expensive disk access if that file is read again in future. If those pages are needed for other purposes, the cached contents can easily be discarded and the pages reallocated.

What is page stealing?

Page stealing is a response to a shortage of free pages, by "stealing" existing allocated physical pages from their current users. It's a statistical method of effectively obtaining extra physical pages by identifying existing allocations unlikely to be used again in the near future and recycling them.

Page stealing removes existing physical pages from their mappings, disposes of their current contents (often by writing them to disk), and reuses the memory elsewhere. If the original user needs their page back, a new physical page is allocated and the old contents loaded into the new page.

Page stealing looks for inactive pages, since active ones would probably just be faulted back in again immediately. Clean inactive pages are almost as good as free pages, because their current contents are already copied somewhere else and can be discarded without even performing any I/O. Dirty pages are cleaned by scheduling I/O to write them to backing store (swap for anonymous pages, to the mapped file for shared file backed mappings).

Page stealing attempts to determine which existing physical pages are least likely to be needed again soon, meaning its trying to predict the future actions of the processes using those pages. It does so through various heuristics, which can never be perfect.

What is a "working set" of pages?

A working set is the set of memory chunks required to complete an operation. For example, the CPU attempts to keep the set of cache lines required for tight inner loops in L1 cache until the loop completes. It attempts to keep the set of frequently used functions from various parts of a program (including shared libraries) in the L2 cache. It does

so both by prefetching cache lines it predicts it may need soon, and by making decisions about which cache lines to discard and which to keep when making space to load new cache lines.

The page fault handler attempts to keep each currently running process's working set of pages in physical memory until the process blocks awaiting input or exits. Unused portions of program code may never even be loaded on a given program run (such as an "options" menu for a program that isn't currently being configured, or portions of generic shared libraries which this program doesn't actually use).

The working set is determined dynamically at runtime, and can change over time as a program does different things.

The objective of page stealing is to keep the "working set" of pages in fast physical memory, allowing processes to "race to quiescence" where the system completes its current tasks quickly and settles down into an idle state waiting for the next thing to do. From this point of view, physical memory can be seen as a cache both for swap pages and for executables in the filesystem. The task of keeping the working set in physical memory (and avoiding page faults that trigger I/O) is analogous to the CPU's task of keeping the appropriate contents in L1 and L2 caches.

What is thrashing?

In low memory situations, each new allocation involves stealing an in-use page from elsewhere, saving its current contents, and loading new contents. When that page is again referenced, another page must be stolen to replace it, saving the new contents and reloading the old contents.

It essentially means that the working set required to service the main loops of the programs the system is running are larger than available physical memory, either because physical memory is tied up doing something else or because the working set is just that big.

This can lead to a state where the CPU generates a constant stream of page faults, and spends most of its time sitting idle, waiting for I/O to service those page faults.

This is often called "swap thrashing", and in some ways is the result of a failure of the system's swap file.

If the swap file is too small (or entirely absent), the system can only steal pages from file backed mappings. Since every executable program and shared library is a file backed mapping, this means the system yanks executable pages, which is generally faults back in fairly rapidly since they tend to get used a lot. This can quickly lead to thrasing.

The other way to encourage swap thrashing is by having too large of a swap file, so that programs that query available memory see huge amounts of swap space and try to use it. The system's available physical memory and I/O bandwidth don't change with the size of the swap file, so attempts to use any significant portion of that swap space result memory accesses occuring at disk I/O speed (four orders of magnitude slower than main memory, stretching each 1/10th of a second out to about two minutes).

What is the Out Of Memory (OOM) killer?

If the system ever truly ran out of physical memory, it could reach a state where every process is waiting for some other process to release a page before it could continue. This deadlock situation would freeze the system.

Before this happened, the system would start thrashing, where it would slow itself to a crawl by spending all its time constantly stealing pages only to steal them back again immediately. This situation is almost as

bad as true deadlock, slowing response time to useless levels (five or ten minute latency on normally instantaneous responses is not unusual during swap thrashing; this is assuming your operation does not time out instead).

A system that enters swap thrashing may take hours to recover (assuming a backlog of demands does not emerge as it fails to service them, preventing it from ever recovering). Or it can take just as long to proceed to a true deadlock (where the flood of swap I/O stops because the CPU is pegged at 100% searching for the next page to steal, never finding one, and thus stops scheduling new I/O).

To avoid either situation, Linux introduced the OOM killer. When it detects the system has entered swap thrashing, it heuristically determines a process to kill to free up pages. It can also be configured to reboot the entire system instead of selecting a specific process to kill.

The OOM killer's process-killing capability is a reasonable way to deal with runaway processes and "fork bombs", but in the absence of a clearly malfunctioning process that is truly "at fault", killing any process is often unacceptable.

Note that the OOM killer doesn't wait for a true memory exhaustion to deadlock the system, both because the system is effectively down while thrashing, and because a paralyzed system might not be able to run even the OOM killer.

The OOM killer's process killing heuristics are a reasonable way to deal with runaway processes and "fork bombs", but in the absence of a clearly malfunctioning process that is truly "at fault", killing any process is often unacceptable. Developers often argue about the choice of processes to kill, and exactly when the thrashing is bad enough to trigger the OOM killer and when to allow the system to attempt to work its way through to recovery. Both of these heuristics are by their nature imperfect, because they attempt to predict the future.

In general, developers try to avoid triggering the OOM killer, and treat its occurrence as the userspace equivalent of a kernel panic(). The system got into an untenable state, it might be good to find out why and prevent its recurrence.

Why is "strict overcommit" a dumb idea?

People who don't understand how virtual memory works often insist on tracking the relationship between virtual and physical memory, and attempting to enforce some correspondence between them (when there isn't one), instead of controlling their programs' behavior.

Many common unix programming idioms create large virtual memory ranges with the potential to consume a lot of physical memory, but never realize that potential. Linux allows the system to "overcommit" memory, creating memory mappings that promise more physical memory than the system could actually deliver.

For example, the fork/exec combo creates transient virtual memory usage spikes, which go away again almost immediately without ever breaking the copy on write status of most of the pages in the forked page tables. Thus if a large process forks off a smaller process, enormous physical memory demands threaten to happen (as far as overcommit is concerned), but never materialize.

Dynamic linking raises similar issues: the dynamic linker maps executable files and shared libraries MAP_PRIVATE, which allows it to write to those pages to perform the dynamic linking fixups allowing the shared library calls to connect to the shared library. In theory, there could be a call to functions or data in a shared library within every page of the

executable (and thus the entire mapping could be converted to anonymous
memory by the copy on write actions of the dynamic linker). And since
shared libraries can call other shared libraries, those could require
private physical memory for their entire mapping too.

In reality, that doesn't happen. It would be incredibly inefficient and
defeat the purpose of using shared libraries in the first place. Most
shared libraries are compiled as Position Independent Code (PIC), and
some executables are Position Independent Executalbles (PIE), which
groups

People who don't understand how virtual memory works often insist on
tracking the relationship between virtual and physical memory, and attempting
to enforce some correspondence between them (when there isn't one). Instead
of designing their programs with predictable and controllable behavior, they
try to make the operating system predict the future. This doesn't work.

Strict overcommit encourages the addition of gigabytes (even terabytes)
of swap space, which leads to swap thrasing if it is ever used. Systems
with large amounts of swap space can literally thrash for days, during which
they perform less computation than they could perform in a single minute
during non-thrashing operation (such as if the OOM killer triggered a
reboot).

What is the appeal of huge pages?

The CPU has a limited number of TLB entries. A huge page allows a single TLB
entry to translate addresses for a large amount of contiguous physical
memory. (For example, the entire Linux kernel fits within a single 2
megabyte "huge page" on x86 systems. Keeping the entire kernel in a
single TLB entry means that calling into the kernel doesn't flush the
userspace mappings out of the TLB.)

Using huge pages means the MMU spends less time walking page tables to refill
the TLB, and can lead to about a 10% performance increase if used
correctly.

What are memory zones and "high memory"?

http://kerneltrap.org/node/2450
http://linux-mm.org/HighMemory
http://www.cs.columbia.edu/~smb/classes/s06-4118/l19.pdf
http://book.opensourceproject.org.cn/kernel/kernelpri/opensource/0131181637/ch04lev1sec2.html