Operating Systems
CS-C3140, Lecture 6

Alexandru Paler

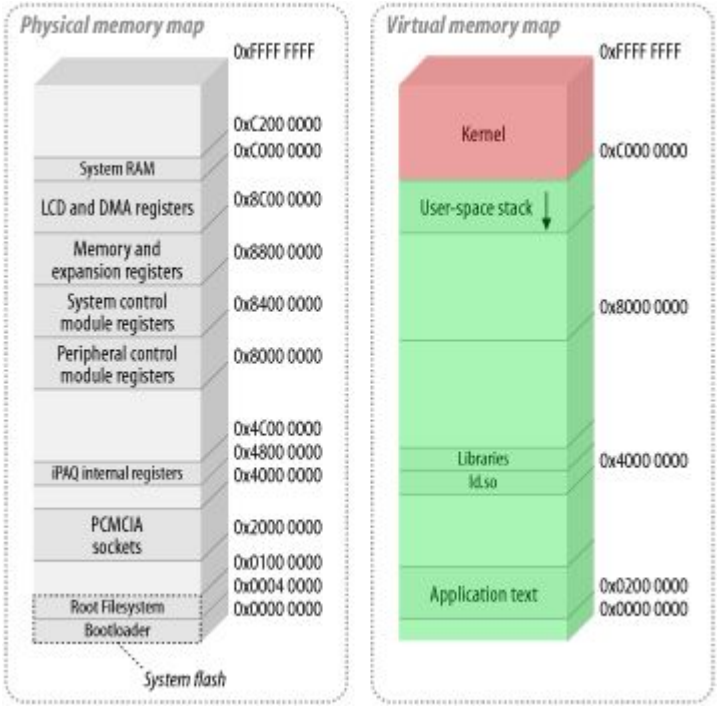
Grading

- The grade is based on points
  - The exercises: n (8?) rounds and 28 points/round -> n*28 points from exercises
  - The exam points are max. 400 - n*28
- To pass the course you have to pass both the exam and the exercises
  - A minimum number of points for the exam (to be announced separately)
  - Final grade = round ((exercises + exam + 100) / 100)
    - 449 -> 400 -> 4
    - 451 -> 500 -> 5
  - Per round limits for the exercises (you have to pass (n-2)/n rounds)
    - 50% of max points for passing the assignments
    - 75% of max points if submitted one week later
    - 2 trials per exercise
      - Do not solve the exercises in the A+ interface
      - Use A+ only for submission
      - Solve in different software (e.g. Notepad)

Linux Memory Management

Linux Memory Management

- Linux memory
  - quite complex
  - shares many characteristics with UNIX
  - process virtual memory
  - kernel memory allocation
- It is traditional and good to have the **kernel mapped in every user process**
  - kernel at virtual addresses 0xC0000000 – 0xFFFFFFFF of every address space
  - the range 0x00000000 – 0xBFFFFFFF for user code, data, stacks, libraries, etc.
- Kernels that have such design are said to be "in the higher half" by opposition to kernels that use lowest virtual addresses for themselves, and leave higher addresses for the applications.

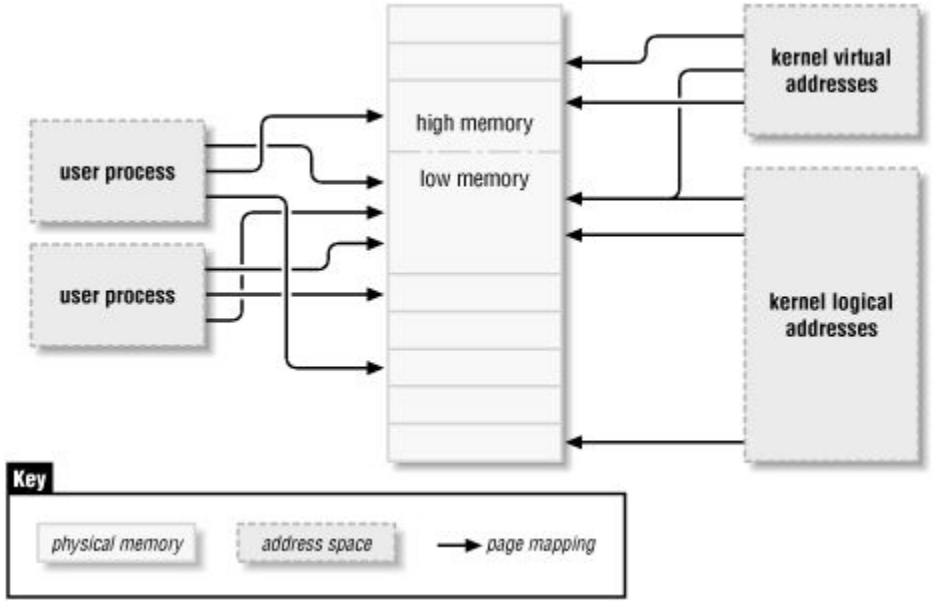


https://wiki.osdev.org/Higher_Half_Kernel
https://stackoverflow.com/questions/56213569/linux-memory-segmentation

Address Types (Kernel)

- **User virtual addresses**: These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.
- **Physical addresses**: The addresses used between the processor and the system's memory. Physical addresses are 32- or 64-bit quantities; even 32-bit systems can use 64-bit physical addresses in some situations.
- Bus addresses: The addresses used between peripheral buses and memory. Often they are the same as the physical addresses used by the processor, but that is not necessarily the case.
- **Kernel logical addresses**: These make up the normal address space of the kernel. **These addresses map most or all of main memory, and are often treated as if they were physical addresses.** On most architectures, logical addresses and their associated physical addresses differ only by a constant offset. Logical addresses use the **hardware's native pointer size**, and thus may be unable to address all of physical memory on heavily equipped 32-bit systems.
- **Kernel virtual (linear) addresses**: These differ from logical addresses in that they **do not necessarily have a direct mapping to physical addresses**. All logical addresses are kernel virtual addresses; memory allocated by vmalloc also has a virtual address (but no direct physical mapping).

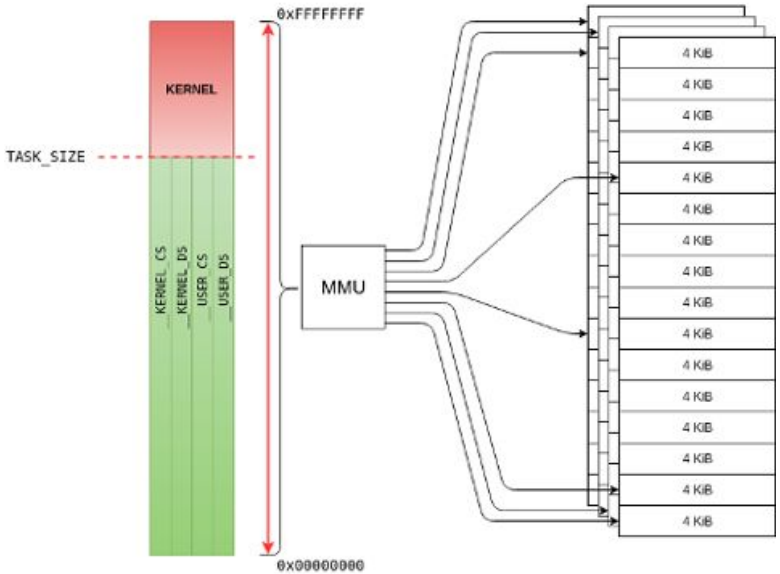
Figure 2-1. Logical address translation



https://www.xml.com/ldd/chapter/book/ch13.html

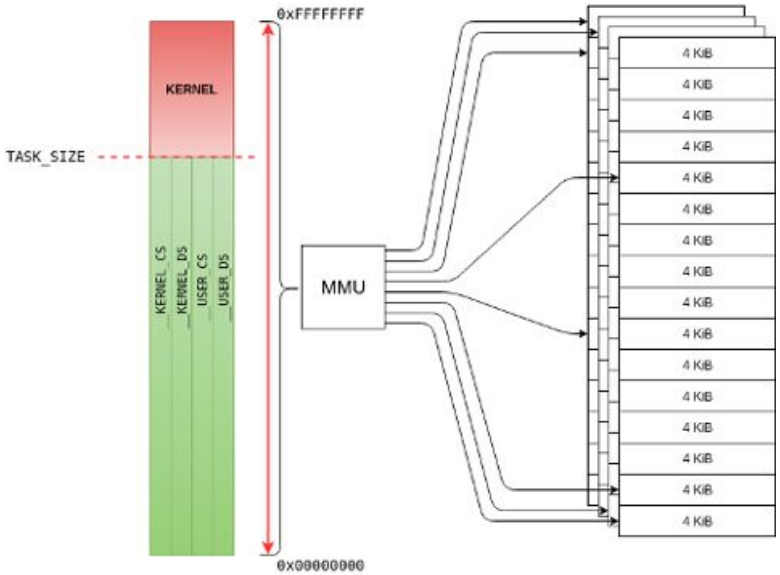
Difference between logical and kernel virtual addresses

- With 32 bits, it is possible to address 4 GB of memory.
  a. Linux was unable to handle more memory than it could set up logical addresses for
  b. Linux needed directly mapped kernel addresses for all memory (kernel mapped in every user process – speed)
- Addresses in the range 0xC0000000 - 0xFFFFFFFF are kernel virtual addresses (red area).
  a. The 896MiB range 0xC0000000 - 0xF7FFFFFF **directly maps kernel logical addresses 1:1 with kernel physical addresses** into the contiguous **lowmem** pages
  b. The remaining 128MiB range 0xF8000000 - 0xFFFFFFFF is used to map virtual addresses for large buffer allocations (a window where parts of the extra RAM is mapped as needed), MMIO ports (Memory-Mapped I/O) and/or PAE memory into the not-contiguous **highmem** pages
  c. lowmem is needed for per-process things including kernel stacks for every task (aka thread), and for page tables themselves
- Addresses in the range 0x00000000 - 0xBFFFFFFF
  a. decision taken by Linus that user process virtual addresses start at 3GB (1GB virtual address for kernel was plenty many decades ago)
  b. user virtual addresses (green area)
  c. userland code, data and libraries reside
  d. mapping can be in not-contiguous low-memory and high-memory pages.



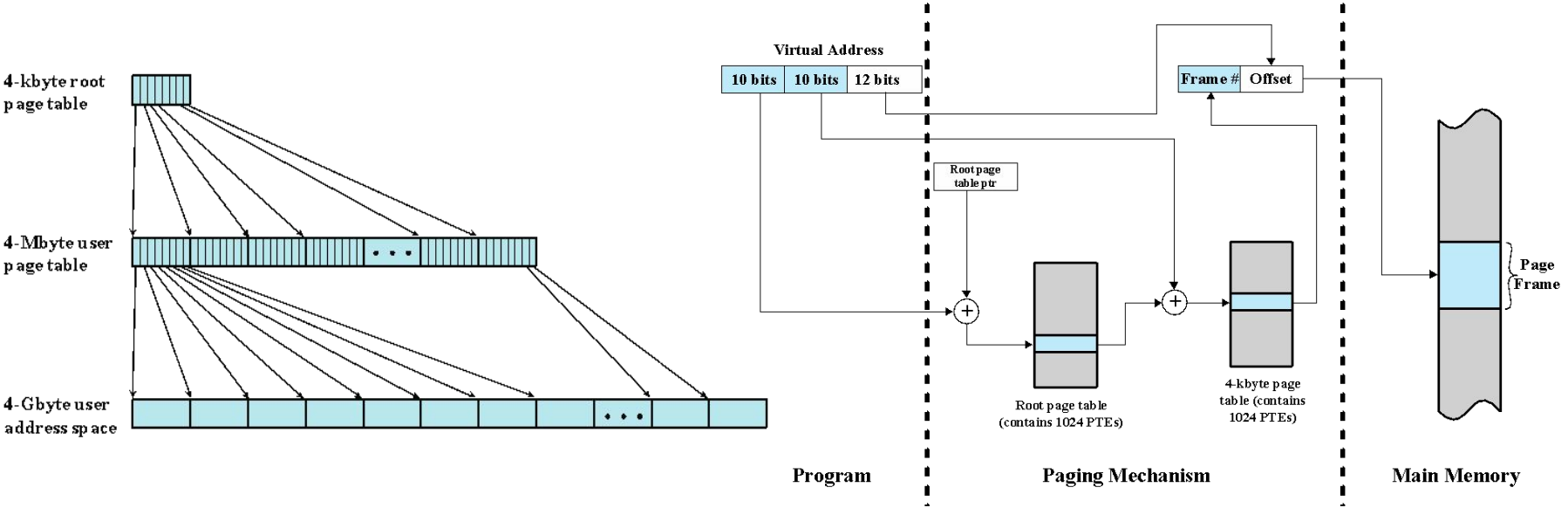
Highmem and 64 bits

```
config NOHIGHMEM
        bool "off"
        help
          Linux can use up to 64 Gigabytes of physical memory on x86 systems.
          However, the address space of 32-bit x86 processors is only 4
          Gigabytes large. That means that, if you have a large amount of
          physical memory, not all of it can be "permanently mapped" by the
          kernel. The physical memory that's not permanently mapped is called
          "high memory".

          If you are compiling a kernel which will never run on a machine with
          more than 1 Gigabyte total physical RAM, answer "off" here (default
          choice and suitable for most users). This will result in a "3GB/1GB"
          split: 3GB are mapped so that each process sees a 3GB virtual memory
          space and the remaining part of the 4GB virtual memory space is used
          by the kernel to permanently map as much physical memory as
          possible.

          If the machine has between 1 and 4 Gigabytes physical RAM, then
          answer "4GB" here.

          If more than 4 Gigabytes is used then answer "64GB" here. This
          selection turns Intel PAE (Physical Address Extension) mode on.
          PAE implements 3-level paging on IA32 processors. PAE is fully
          supported by Linux, PAE mode is implemented on all recent Intel
          processors (Pentium Pro and better). NOTE: If you say "64GB" here,
          then the kernel will not boot on CPUs that don't support PAE!

          The actual amount of total physical memory will either be
          auto detected or can be forced by using a kernel command line option
          such as "mem=256M". (Try "man bootparam" or see the documentation of
          your boot loader (lilo or loadlin) about how to pass options to the
          kernel at boot time.)

          If unsure, say "off".
```



https://elixir.bootlin.com/linux/v5.8/source/arch/x86/Kconfig#L1363

(Recap) Two-Level Hierarchical Page Table



Page Tables

Page Directory (PGD)

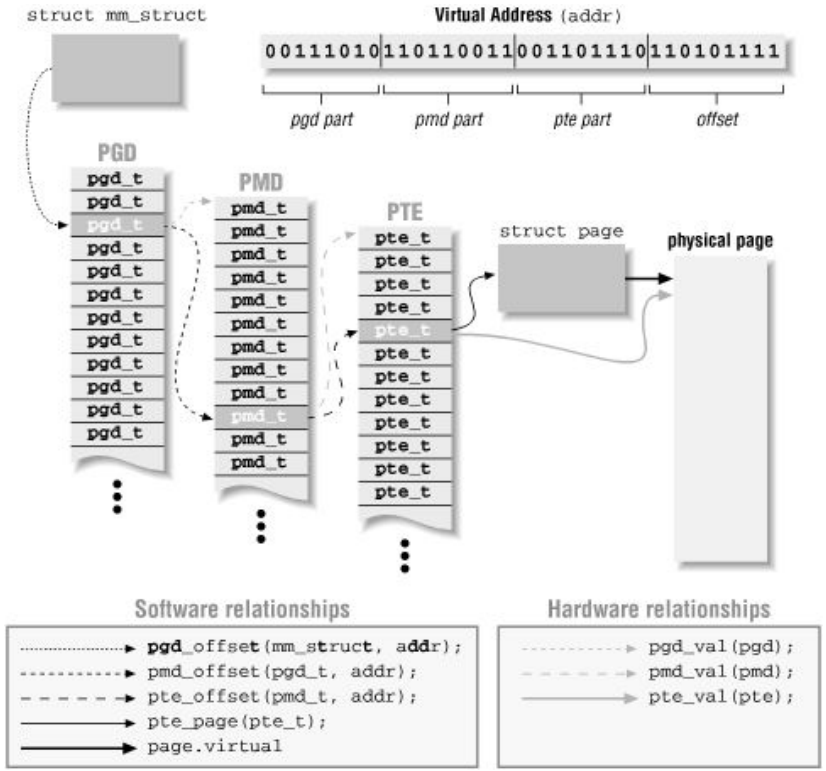
The top-level page table. The PGD is an array of pgd_t items, each of which points to a second-level page table. **Each process has its own page directory**, and there is one for kernel space as well. You can think of the page directory as a page-aligned array of pgd_ts.

Page mid-level Directory (PMD)

The second-level table. The PMD is a page-aligned array of pmd_t items. A pmd_t is a pointer to the third-level page table. Two-level processors have no physical PMD; they declare their PMD as an array with a single element, whose value is the PMD itself.

Page Table

A page-aligned array of items, each of which is called a Page Table Entry. The kernel uses the pte_t type for the items. A pte_t contains the physical address of the data page.



https://www.xml.com/ldd/chapter/book/ch13.html

Kernel Memory Allocation

- Kernel memory capability manages physical main memory page frames
  - Primary function is to allocate and deallocate frames for particular uses

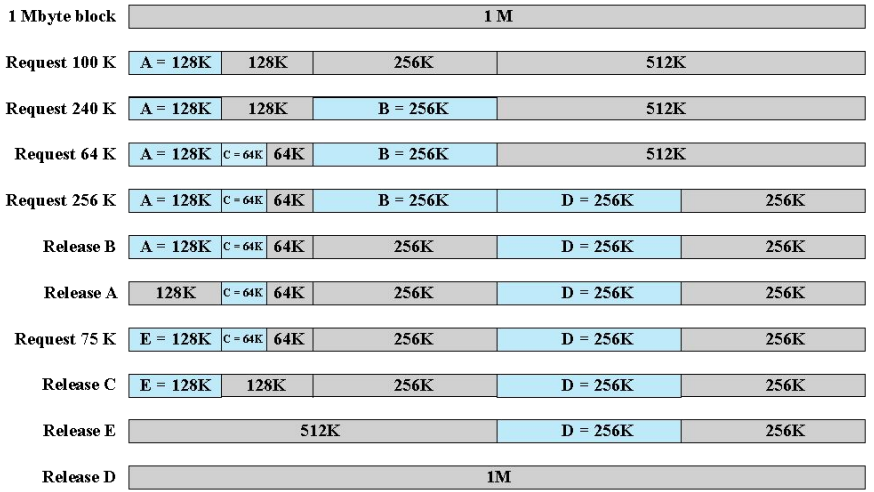
Possible owners of a frame include:

- User-space processes
- Dynamically allocated kernel data
- Static kernel code
- Page cache

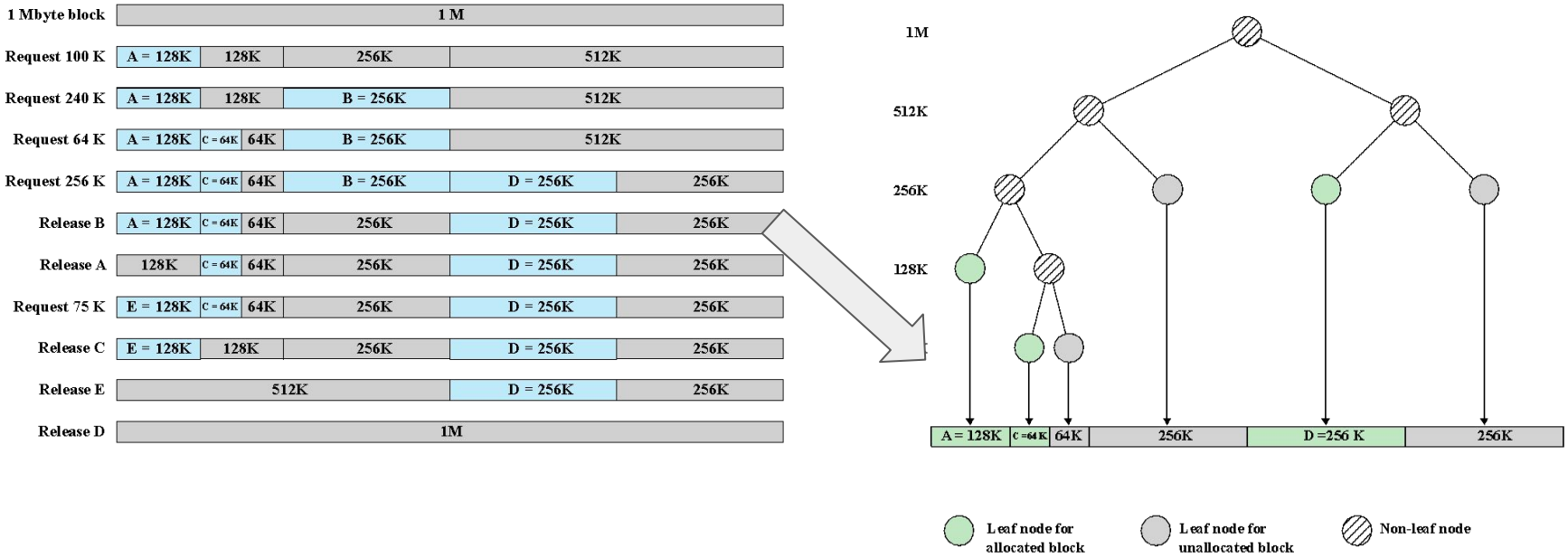
  - A buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages
  - Page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes
    - Slab allocation
    - On a x86 machine, the page size is 4 Kbytes, and chunks within a page may be allocated of sizes 32, 64, 128, 252, 508, 2,040, and 4,080 bytes.
    - Used by Linux to accommodate small chunks

Example: Buddy System

- Space available for allocation is treated as a single block
- Memory blocks are available of size 2^K words, $L \leq K \leq U$, where
  - 2^L = smallest size block that is allocated
  - 2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation
- Fast algorithm for allocation and deallocation



Example of using a Buddy System



Physical Page Allocation: Managing Free Blocks

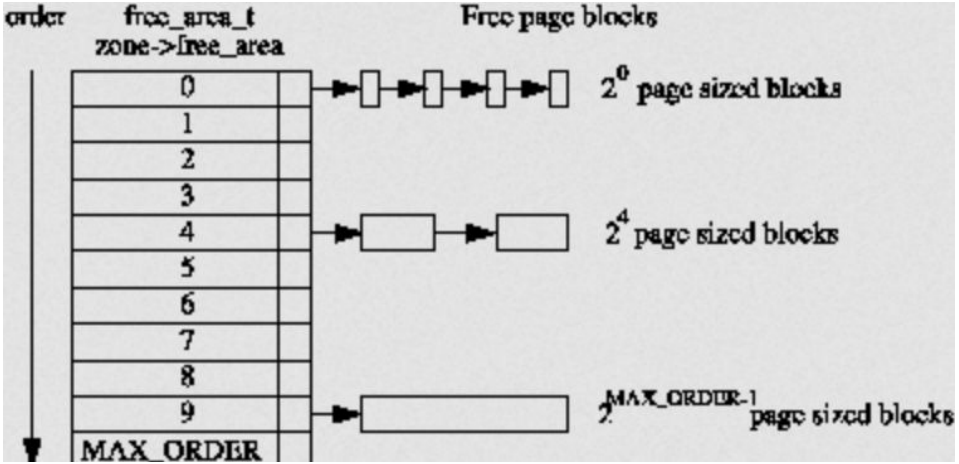
The allocator maintains blocks of free pages where each block is a power of two number of pages.

Eliminates the chance that a larger block will be split to satisfy a request where a smaller block would have sufficed.

The exponent for the power of two sized block is referred to as the *order*.

An array of free_area_t structs are maintained for each order that points to a linked list of **blocks of pages** that are free.

- the 0th element of the array will point to a list of free page blocks of size 2^0 or 1 page,
- the 1st element will be a list of 2^1
- up to $2^{MAX\_ORDER-1}$ number of pages, where the MAX_ORDER is currently defined as 10



https://www.kernel.org/doc/gorman/html/understand/understand009.html

Physical Page Allocation

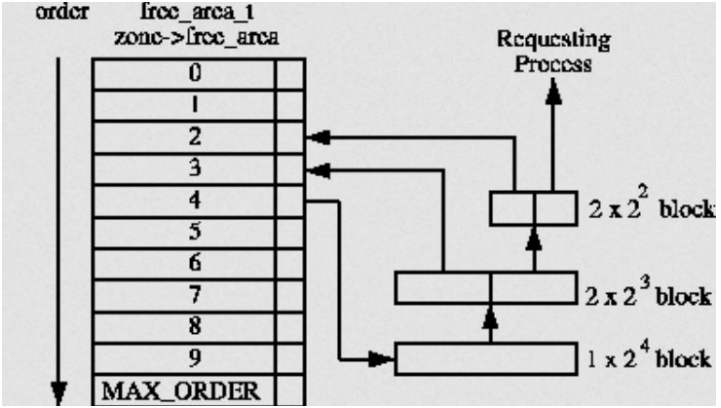
Allocations are always for a specified order

- 0 in the case where a single page is required
- If a free block cannot be found of the requested order, a higher order block is split into two buddies
- One is allocated and the other is placed on the free list for the lower order.

When the block is later freed

- the buddy will be checked
- If both are free, they are merged to form a higher order block and placed on the higher free list where its buddy is checked and so on.
- If the buddy is not free, the freed block is added to the free list at the current order

During these list manipulations, **interrupts have to be disabled** to prevent an interrupt handler manipulating the lists while a process has them in an inconsistent state.



A 2^4 block is split and the buddies are added to the free lists until a block for the process is available

https://www.kernel.org/doc/gorman/html/understand/understand009.html

Linux Page Replacement

- Prior to 2.6 based on the clock algorithm
  - The use bit is replaced with an 8-bit age variable
  - Incremented each time the page is accessed
- Periodically decrements the age bits
  - A page with an age of 0 is an “old” page that has not been referenced in some time and is the best candidate for replacement
- A form of least recently used (LRU) policy
  - **active_list** contains the working set of all processes
  - **inactive_list** contains reclaim candidates
  - replacement policy is global: reclaimable pages are contained in just two lists and pages belonging to any process may be reclaimed, rather than just those belonging to a faulting process

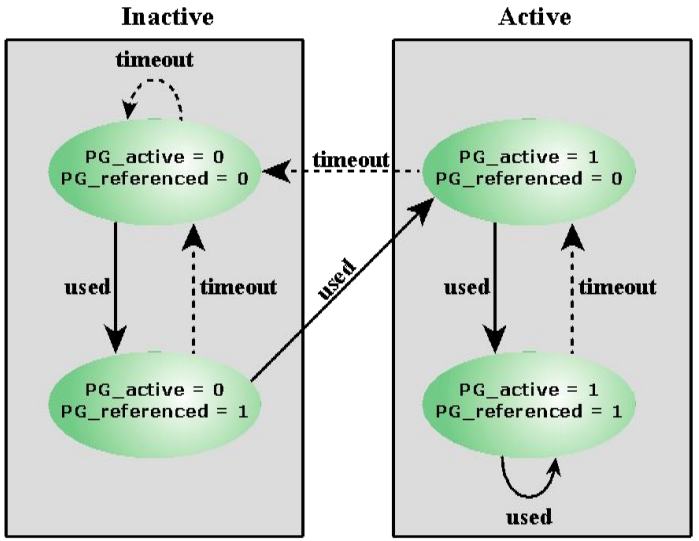
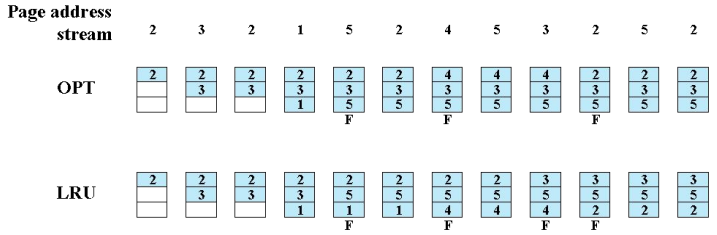


Figure 8.24 Linux Page Reclaiming



/proc/[pid]/maps

```
address           perms offset  dev   inode       pathname
00400000-00452000 r-xp 00000000 08:02 173521      /usr/bin/dbus-daemon
00651000-00652000 r--p 00051000 08:02 173521      /usr/bin/dbus-daemon
00652000-00655000 rw-p 00052000 08:02 173521      /usr/bin/dbus-daemon
00e03000-00e24000 rw-p 00000000 00:00 0           [heap]
00e24000-011f7000 rw-p 00000000 00:00 0           [heap]
...
35b1800000-35b1820000 r-xp 00000000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a1f000-35b1a20000 r--p 0001f000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a20000-35b1a21000 rw-p 00020000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a21000-35b1a22000 rw-p 00000000 00:00 0
35b1c00000-35b1dac000 r-xp 00000000 08:02 135870  /usr/lib64/libc-2.15.so
35b1dac000-35b1fac000 ---p 001ac000 08:02 135870  /usr/lib64/libc-2.15.so
35b1fac000-35b1fb0000 r--p 001ac000 08:02 135870  /usr/lib64/libc-2.15.so
35b1fb0000-35b1fb2000 rw-p 001b0000 08:02 135870  /usr/lib64/libc-2.15.so
...
f2c6ff8c000-7f2c7078c000 rw-p 00000000 00:00 0    [stack:986]
...
7fffb2c0d000-7fffb2c2e000 rw-p 00000000 00:00 0   [stack]
7fffb2d48000-7fffb2d49000 r-xp 00000000 00:00 0   [vdso]
```

- The address field is the address space in the process that the mapping occupies.
- The perms field is a set of permissions:
  - r = read w = write x = execute
    s = shared p = private (copy on write)
- The offset field is the offset into the file/whatever;
- dev is the device (major:minor);
- inode is the inode on that device. 0 indicates that no inode is associated with the memory region, as would be the case with BSS (uninitialized data).
- The pathname field will usually be the file that is backing the mapping.

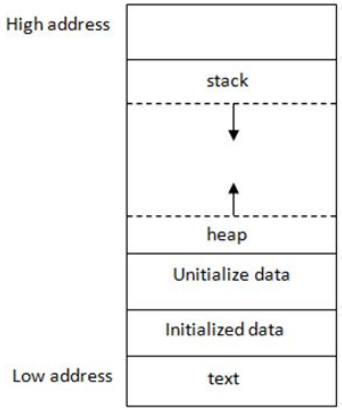
Additional helpful pseudo-paths

- [stack] The initial process's (also known as the main thread's) stack.
- [heap] The process's heap.
- [stack:<tid>] (from Linux 3.4 to 4.4) A thread's stack (where the <tid> is a thread ID). Was removed in Linux 4.5, since providing this information for a process with large numbers of threads is expensive.
- [vdso] The virtual dynamically linked shared object.

https://man7.org/linux/man-pages/man5/proc.5.html

C Tutorial

Stack and Heap



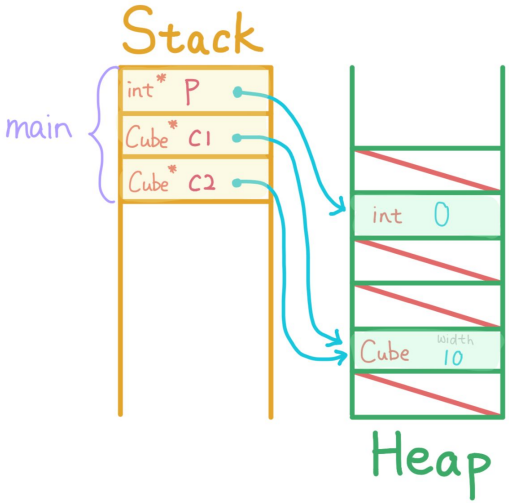
A running program takes up memory

- users are often not aware of allocated memory
- the program is allocating memory for each variable
- each byte of memory has an address

Each running program has its own memory layout, separated from other programs. The layout consists of a lot of segments, including

- stack: stores local variables
- heap: dynamic memory for programmer to allocate
- data: stores global variables, separated into initialized and uninitialized
- text: stores the code being executed

Heap segments have low address numbers, while the stack memory has higher addresses.




```
int main() {
  int *p = new int;
  Cube *c1 = new Cube();
  Cube *c2 = c1;
  c2->setLength( 10 );
  return 0;
}
```

https://courses.grainger.illinois.edu/cs225/fa2022/resources/stack-heap/

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/4/Lecture4.pdf

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/6/Lecture6.pdf

https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/7/Lecture7.pdf