Linux Memory Management

W4118 Operating Systems I

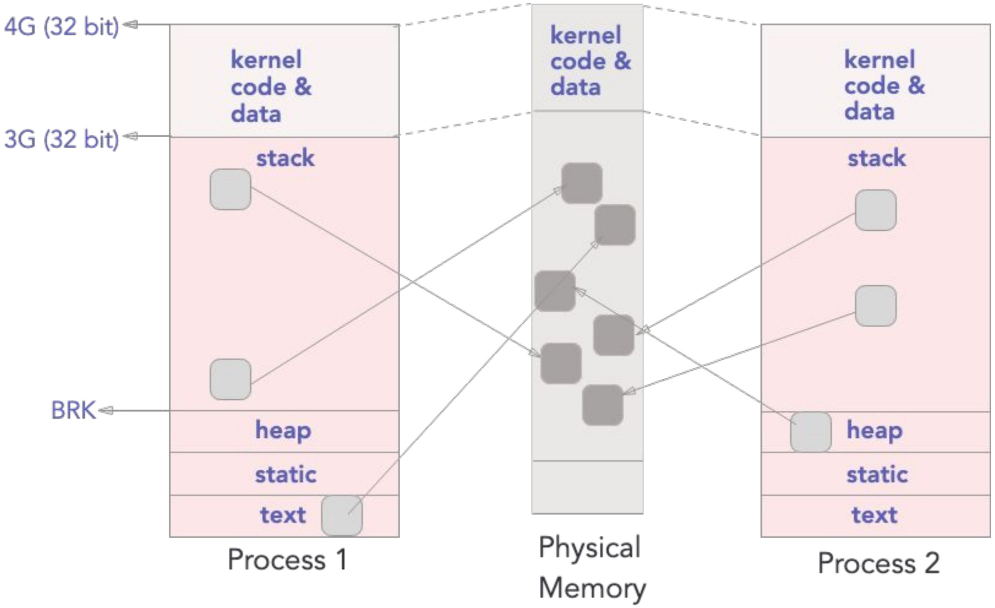
https://cs4118.github.io/www/2024-1/

Credits to Jae and Hans

Address space one more time!

We covered how the following fit into the virtual address space:

- program break
- file-backed mappings (e.g. shared C std library)
- anonymous mappings
- kernel code & data



Checking the mappings

Run `prog` and `cat /proc/<pid>/maps`

Some regions of interest:

- Program code/text/static regions (notice file path)
- Program heap
- Shared C std libraries
- File-backed mapping (foo.txt)
- Anonymous mappings (entries without path)
- Stack
- vvar/vdso/vsyscall: Implementations of virtual system calls (e.g. kernel maps simple syscalls like gettimeofday() code and data into userspace to avoid cost of context-switch)

`struct mm_struct` (source code)

Each task has a memory descriptor – a structure that manages its virtual address space.

- `task_struct` field: `struct mm_struct *mm`

Notable fields

Linking together virtual memory areas of a virtual address space:

- `struct vm_area_struct *mmap;  /* list of VMAs */`
- `struct rb_root mm_rb;`

Note two forms of traversal:

- linear traversal via linked list (useful for proc maps)
- logarithmic traversal via red-black tree (useful for quickly finding vma of a given vaddr)

`struct mm_struct` [(source code)]

Pointer to task's page global directory (PGD), the top of the page table hierarchy:

- `pgd_t * pgd;`

Start page table traversal here for HW7 part1!

Virtual address boundaries for fundamental areas (code, data, heap, stack, argv, env):

- `unsigned long start_code, end_code, start_data, end_data;`
- `unsigned long start_brk, brk, start_stack;`
- `unsigned long arg_start, arg_end, env_start, env_end;`

`struct vm_area_struct` (source code)

Each virtual memory area (VMA) also has a descriptor, containing metadata for 1 region within the task's virtual address space

Notable fields

Fields for traversing VMAs:

```
/* The first cache line has the info for VMA tree walking. */
unsigned long vm_start;     /* Our start address within vm_mm. */
unsigned long vm_end;       /* The first byte after our end address
                               within vm_mm. */
/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next, *vm_prev;
struct rb_node vm_rb;
```

`struct vm_area_struct` (source code)

Page permissions and VMA permissions:

```
/*
* Access permissions of this VMA.
* See vmf_insert_mixed_prot() for discussion.
*/
pgprot_t vm_page_prot;
unsigned long vm_flags;    /* Flags, see mm.h. */
```

`vm_flags` can specify a subset of permissions that `vm_page_prot` advertises (e.g. you can map a RW file as only R)

`struct vm_area_struct` (source code)

Fields for managing file-backed mappings:

```
struct file * vm_file;     /* File we map to (can be NULL). */
unsigned long vm_pgoff;    /* Offset (within vm_file) in
                              PAGE_SIZE units */
```

Fields for managing anonymous mappings:

```
struct list_head anon_vma_chain; /* Serialized by mmap_lock &
                                  * page_table_lock */
struct anon_vma *anon_vma;      /* Serialized by page_table_lock */
```

See how reverse mapping works here

`struct vm_area_struct` [(source code)]

VMAs have methods that operate on them (similar to `struct sched_class`)

```
/* Function pointers to deal with this struct. */

const struct vm_operations_struct *vm_ops;
```

For example, handler for page fault after hardware raises exception:

```
vm_fault_t (*fault)(struct vm_fault *vmf);
```

Tracing a page fault

```
do_page_fault()  // Exception handler for page fault, recall "exception"
                 // category of interrupts from syscall lecture
\_ find_vma()    // rbtree traversal to find VMA holding faulting address
\_ handle_mm_fault() ->__handle_mm_fault() -> handle_pte_fault()
   \_ do_fault()
      \_ do_cow_fault()  // e.g. You can tell if a page is marked for COW
                         // when the PTE is write protected but the VMA isn't.
      \_ do_read_fault()
      \_ do_shared_fault()
      -> __do_fault() called by the above three handlers
        \_ vma->vm_ops->fault()
```

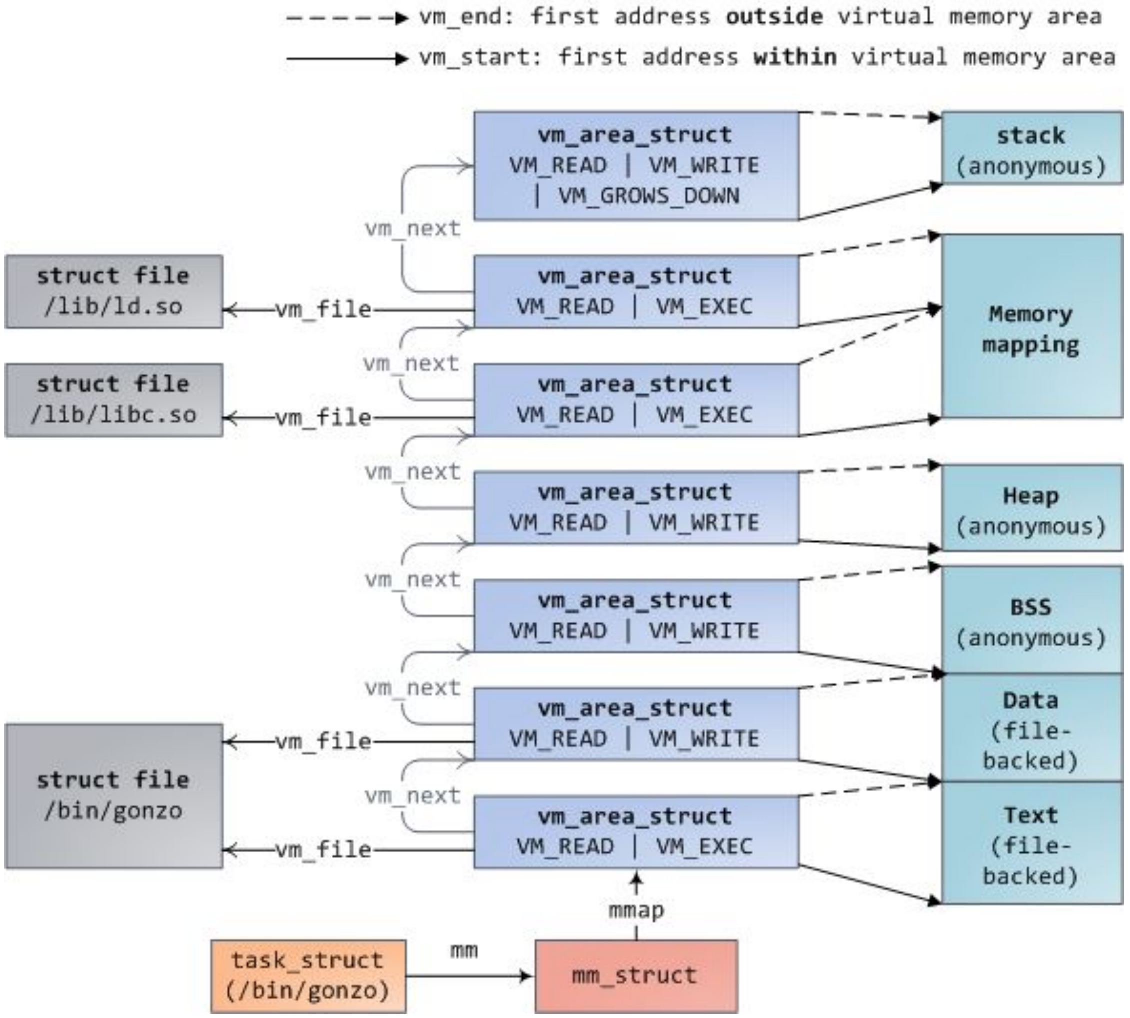
Tracing `mmap()`

```
/mm/mmap.c


SYSCALL_DEFINE6(mmap_pgoff)
\_ ksys_mmap_pgoff()
   \_ vm_mmap_pgoff()
      \_ mmap_region()
```

As VMAs are created, kernel will try to coalesce adjacent VMAs, assuming they have the same backing and permissions.

Virtual Address Space, again!



`struct page` (source code)

Each physical frame has a corresponding `struct page`, which houses metadata regarding the physical frame.

Global array of `struct page *` that is aligned with the physical frames:

```
struct page *mem_map;
```

Given a PFN, you can easily derive the physical frame and the corresponding struct page… at least, in the case of the FLATMEM memory model. See more complicated physical memory models here.

`struct page` (source code)

Notable fields

- Page reference count, access via `page_count()`

```
atomic_t _refcount;
```

You can expect shared mappings to reference pages that have `page_count() > 1`

- Entry in Linux data structure enforcing LRU eviction policy, we'll revisit this shortly:

```
/**

* @lru: Pageout list, eg. active list protected by pgdat->lru_lock. Sometimes used as a generic list by the page owner.

*/

struct list_head lru;
```

`struct page` [(source code)]

- Associates this struct page with a mapping (file-backed/anonymous):

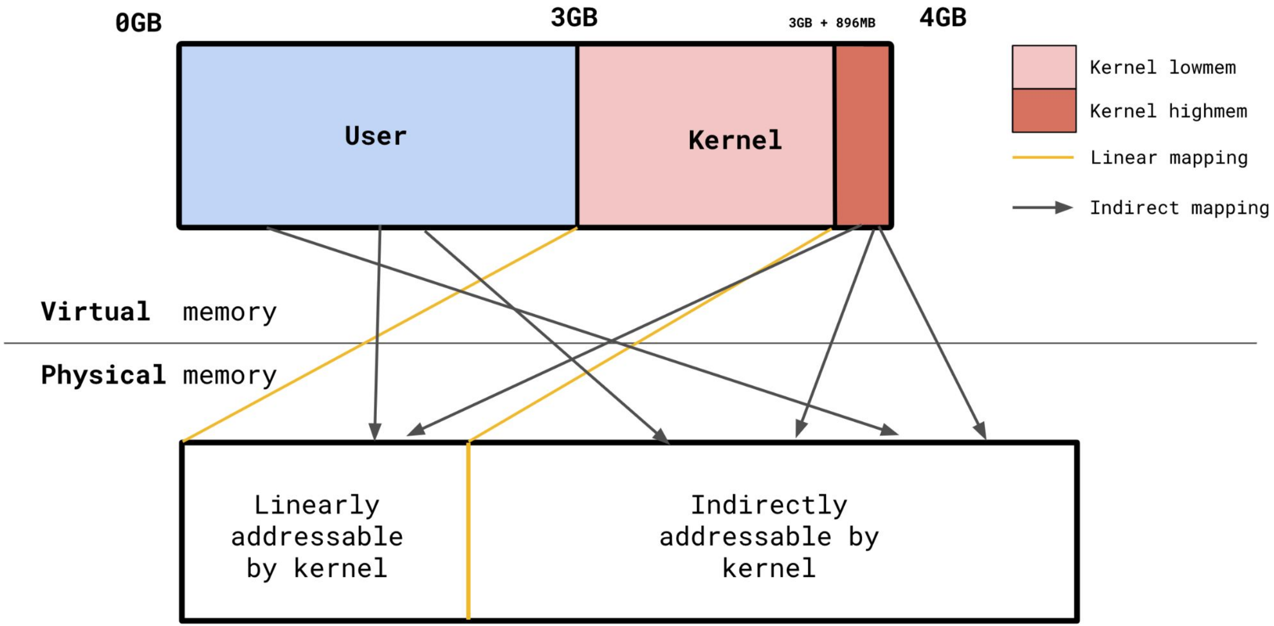
```
struct address_space *mapping;
pgoff_t index;       /* Our offset within mapping. */
```

- Kernel virtual address corresponding to the physical frame:

```
/*
* On machines where all RAM is mapped into kernel address space,
* we can simply calculate the virtual address. On machines with
* highmem some memory is mapped into kernel virtual memory
* dynamically, so we need a place to store that address.
*/
void *virtual;            /* Kernel virtual address (NULL if not kmapped, i.e. highmem) */
```

`highmem`

The kernel reserves a small part of its 1GB virtual address space for “arbitrary” mappings



Linux Page Cache

Recall that `struct vma` has a reference to the open file backing the memory mapping

```
struct file * vm_file;   /* File we map to (can be NULL). */
```

`struct file` references `struct inode`, which has the following field:

```
struct address_space   *i_mapping;
```

This is named rather misleadingly… think about it as a reference to all frames backed by the file associated with the inode (recall field from `struct page`). The data structure is meant be generic, so it also holds cached pages!

Serving read()s and write()s

Reads and writes don’t just go to disk every time.

Try to serve **reads** from the page cache if it’s there. Otherwise, read from disk, allocate page for it, associate it into `address_space` to cache it.

Write to the page cache and mark the page dirty.

Eventually, get written back either by the kernel asynchronously or by the user synchronously (e.g. via sync()).

This is known as a write-back cache (as opposed to a write-through cache, which writes to disk immediately).

Serving mmap()

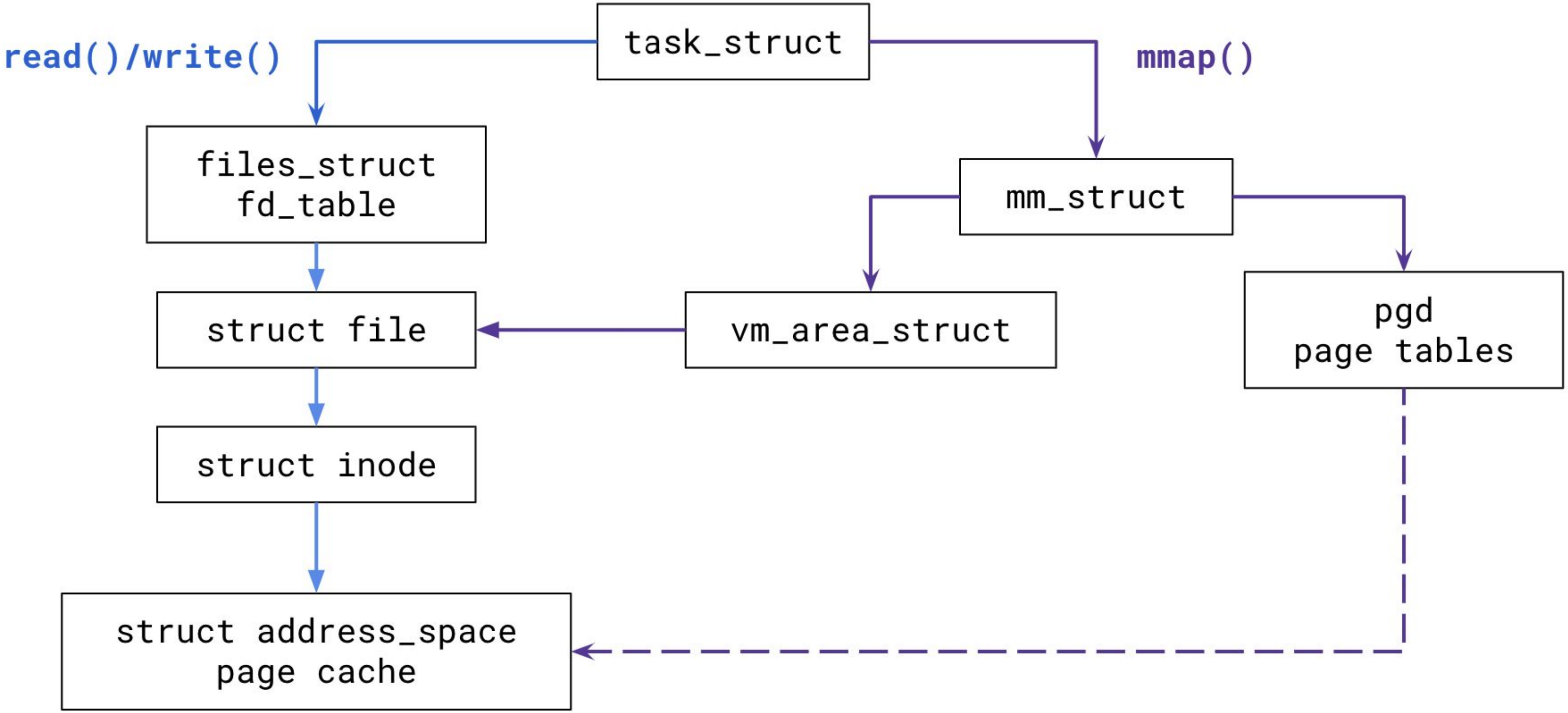
Each `mmap()` call creates or extends a single VMA which is linked into the task's `mm_struct`. That VMA can be an anonymous mapping or a file-backed mapping.

File-backed mappings also hook into the page cache.

- After new VMAs are created, the file contents are eventually faulted into memory.
- The pages from the page cache are installed into the task's page tables.

Assuming `MAP_SHARED, read()/write()` and the shared filed-back mapping will target the same pages in the page cache!

Page Cache, visualized



Cache Replacement Policy

The Linux page cache implements variant of LRU approximation we have discussed

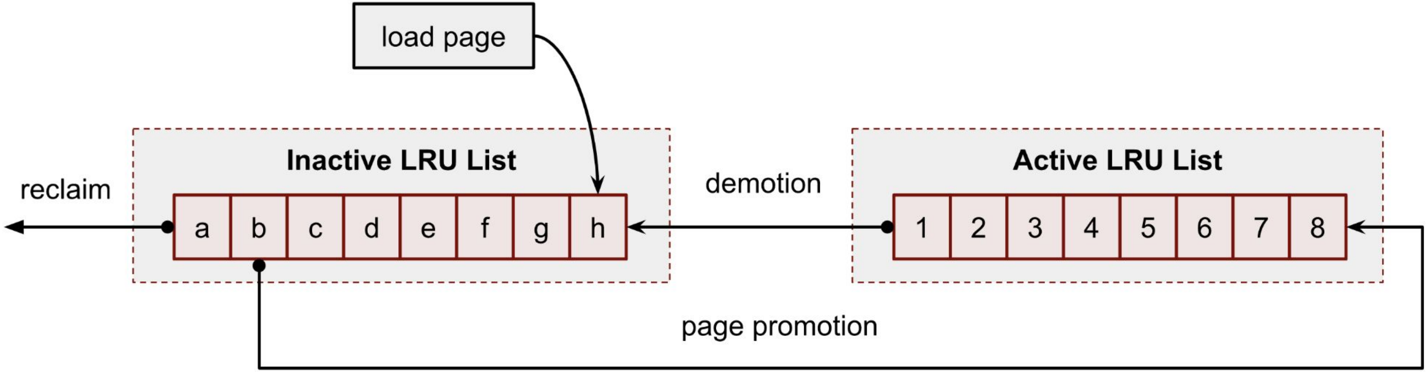
But consider a process that maps in a huge file, reads it once, then never touches the file again → Invalidates the whole cache!

Cache Replacement Policy

The Linux page cache implements variant of LRU approximation we have discussed

But consider a process that maps in a huge file, reads it once, then never touches the file again → Invalidates the whole cache!

Solution: maintain two LRU lists: active and inactive list



Linux Out of Memory (OOM) Killer

Show `memory-hog` example

OOM killer logic is housed in `mm/oom_kill.c`

1. A process needs to be selected for killing:

```
/*
 * Simple selection loop. We choose the process with the highest number of
 * 'points'. In case scan was aborted, oc->chosen is set to -1.
 */
static void select_bad_process(struct oom_control *oc);
```

Linux Out of Memory (OOM) Killer

2. The decision is based on heuristics:

```
/**
 * oom_badness - heuristic function to determine which candidate task to kill
 * @p: task struct of which task we should calculate
 * @totalpages: total present RAM allowed for page allocation
 *
 * The heuristic for determining which task to kill is made to be as simple and
 * predictable as possible.  The goal is to return the highest value for the
 * task consuming the most memory to avoid subsequent oom failures.
 */
long oom_badness(struct task_struct *p, unsigned long totalpages);
```

Linux Out of Memory (OOM) Killer

3. Once we find a victim process, kill it by sending it a **`SIGKILL`**

The oom_score can be adjusted from userspace, see **`/proc/<pid>/oom_score_adj`**.

See current score in **`/proc/<pid>/oom_score`**

BUT the victim process is not even given a warning

OOM Alternatives

The Linux OOM killer is considered to be “draconian”!

There are a few other alternatives that could kick in before the OOM killer:

- daemons monitoring memory usage
  - systemd-oomd: a userspace out-of-memory (OOM) killer… take corrective action before an OOM occurs in the kernel space.
- userspace memory limits