Virtual Memory

Michael E. Locasto
CPSC 457
Agenda

Review memory address translation
  concepts
  mechanism
Virtual Memory
  key problem
  how it works
Topic Wish List

> I'm curious about the conversion/compatibility of software across different operating systems... it would be interesting to learn more about the reasons why a lot of software doesn't get made in such a way as to work on Linux as well as Windows or OSX.

> I'd like to learn more about the implementation of network primitives in the kernel, and understand some of the magic that constructs useful packets from a stream of bytes that a NIC receives.

> I'm interested in how the OS handles / allocates resources for concurrent processes and ensures that they don't conflict.

> I want to understand how the OS manages memory, the OS's control of hardware and software I/O, and the connection between kernel mode and user mode.

> I would like to know how the OS contributes to performance as well as how it processes graphics to display them on-screen (more specifically how the software-hardware interaction for this works).

> I am interested in seeing how the OS translates programs into concurrently running processes

> I am interested in learning how to talk directly to the OS without having the software applications to do that for me.
Topic Wish List

> I'm curious about the conversion/compatibility of software across different operating systems... it would be interesting to learn more about the reasons why a lot of software doesn't get made in such a way as to work on Linux as well as Windows or OSX.

> I'd like to learn more about the implementation of network primitives in the kernel, and understand some of the magic that constructs useful packets from a stream of bytes that a NIC receives.

> I'm interested in how the OS handles / allocates resources for concurrent processes and ensures that they don't conflict.

> I want to understand how the OS manages memory, the OS's control of hardware and software I/O, and the connection between kernel mode and user mode.

> I would like to know how the OS contributes to performance as well as how it processes graphics to display them on-screen (more specifically how the software-hardware interaction for this works).

> I am interested in seeing how the OS translates programs into concurrently running processes

> I am interested in learning how to talk directly to the OS without having the software applications to do that for me.
Where Does the Problem Originate?

Segmentation & paging provide isolation

Binding between variables and physical location is broken (good!)

Processes cannot access same physical (or even virtual) memory location w/o asking kernel for help setting up IPC (threads, shared memory, sockets, or file I/O)

But none of this accounts for the number of processes!
Figure 3-1. Segmentation and Paging

“hidden” segment register state bits  TLB caching
[michael@proton ~]$ ps aux | wc
 131 1565 12383
[michael@proton ~]$  

VMware Tools is not installed. Choose the Virtual Machine > Install VMware Tools menu.
10/11/14

locasto CPSC 457

```bash
ps aux | wc
```

```
137 2136 45994
```

```
10/11/14
```

Processes: 140 total, 6 running, 134 sleeping, 817 threads
Load Avg: 1.33, 1.11, 1.18  CPU usage: 19.86% user, 9.15% sys, 70.99%
SharedLibs: 7904K resident, 3828K data, 0B linkedit.
MemRegions: 38691 total, 3333M resident, 74M private, 768M shared.
PhysMem: 1057M wired, 4607M active, 1008M inactive, 6672M used, 1517M free
VM: 247G vsize, 1043M framework vsize, 18045932(0) pageins, 949246(0) pageouts
Networks: packets: 46046083/28G in, 200230404/247G out.
Disks: 4673909/254G read, 3057857/111G written.

<table>
<thead>
<tr>
<th>PID</th>
<th>COMMAND</th>
<th>%CPU</th>
<th>TIME</th>
<th>#TH</th>
<th>#WQ</th>
<th>#POR</th>
<th>#MREG</th>
<th>RPRVT</th>
<th>RVRVT</th>
</tr>
</thead>
<tbody>
<tr>
<td>37899</td>
<td>mdworker</td>
<td>0.0</td>
<td>00:00.06</td>
<td>3</td>
<td>1</td>
<td>49</td>
<td>65</td>
<td>1892K</td>
<td></td>
</tr>
<tr>
<td>37898</td>
<td>top</td>
<td>10.6</td>
<td>00:06.13</td>
<td>1/1</td>
<td>0</td>
<td>25</td>
<td>33</td>
<td>1396K</td>
<td></td>
</tr>
<tr>
<td>37888</td>
<td>ocspd</td>
<td>0.0</td>
<td>00:00.01</td>
<td>1</td>
<td>0</td>
<td>23</td>
<td>29</td>
<td>552K</td>
<td></td>
</tr>
<tr>
<td>37886</td>
<td>cupsd</td>
<td>0.0</td>
<td>00:00.05</td>
<td>3</td>
<td>1</td>
<td>37</td>
<td>61</td>
<td>2964K</td>
<td></td>
</tr>
<tr>
<td>37723</td>
<td>Microsoft D</td>
<td>0.0</td>
<td>00:00.02</td>
<td>3</td>
<td>1</td>
<td>81</td>
<td>217</td>
<td>2440K</td>
<td></td>
</tr>
</tbody>
</table>
Focus Question

How does an OS kernel keep all those processes in memory at once?
Focus Question

How does an OS kernel keep all those processes in memory at once?

It doesn’t.
Focus Question (cont.)

1. There is only a limited amount of RAM
2. On average, NNN processes
3. Processes don’t know how much memory they will actually need
4. Each believes they can access at least 4GB

So how do we keep this fiction alive?
The Gap

Total memory demands outstrip physical supply
# of processes X 4GB (worst case on 32-bit)
140*4GB (picture above) =~ 560GB

But: only 8GB of physical RAM on this laptop
Other Problems...

Internal fragmentation (segments)
External fragmentation (pages)

Heaps (for example) need to grow dynamically. How do we keep the new physical addresses associated with the contiguous virtual addresses?
Memory hierarchy and locality

SEEDS OF A SOLUTION
The Memory Hierarchy

- **Registers**
- **CPU cache**
- **Primary Memory**
- **SSD**
- **Rotational disk**
- **Optical Disk**

- cost → speed → size / capacity

Locality rides to the rescue; program data may actually reside anywhere here, but is still logically in the registers.
## History / Previous Approaches

<table>
<thead>
<tr>
<th>Swapping</th>
<th>Overlays</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swap entire jobs to disk in between quantums (i.e., on a context switch)</td>
<td>Key insight: split program up into relatively small units called “overlays”</td>
</tr>
<tr>
<td>Slow. Penalty coming and going</td>
<td>One overlay points to the next; in essence, the program occupies the same memory for its lifetime</td>
</tr>
<tr>
<td>Problematic for dynamic memory allocation</td>
<td>Programmer needs to do the work of splitting</td>
</tr>
</tbody>
</table>

10/11/14
Virtual Memory

A more elegant version of overlays.

Swapping in a new page means servicing a page fault and reading the data in from disk – two relatively costly procedures. But the alternatives are much more expensive: full swapping or inflexible overlays.

Call such units "pages" of a standard size, and we can instead maintain a collection of such pages (a subset of the entire program) in memory as the "working set" of a process.

Should we need to use the physical page frame underlying any particular page, we can swap that page out to disk and bring in the new, needed page.

If the working set is relatively stable, and most programs don't use their full virtual address space, virtual memory can multiplex primary memory enough to maintain the fiction implied by the Process Address Space abstraction.
Virtual Memory

We know from page address translation is that the mapping between a logical page and a physical frame need not be consistent.

A logical page can "move" around between physical frames, and the way the OS keeps track is by updating the corresponding page table entries.

Adding this layer of indirection is the fundamental mechanism that permits virtual memory. Logical pages that are absent from physical memory can actually spend time on some other storage medium (i.e., disk).
Bits: Contents of Page Table Entries

20 bits: page frame index
Present bit: is this logical page in a frame?
Protection bits: R/W/X (R sometimes implies X)
Dirty bit: was this page written?
Accessed bit: hardware sets this; OS must unset.
    used to determine candidates for swapping
User/Supervisor: ring 3 or everything else
Virtual Memory Mechanism

Path 1
Process issues M; M in physical

Linear address belongs to a page that is actually in physical memory.

Page translation takes care of identifying the appropriate physical frame. The OS is minimally involved.

Path 2
Process issues M; M absent

Linear address is not in physical memory. Page translation circuitry discovers this during translation (Present bit=0); the hardware raises a Page Fault exception (14) that the OS must now service.

The linear address is placed in cr2.

The OS attempts to find the missing page on the hard disk. If the page is not valid, then the OS raises a SIGSEGV. Process will likely crash.
## Picking a Destination Frame

<table>
<thead>
<tr>
<th>Case A: Free Frame</th>
<th>Case B: No free frame.</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a free page frame.</td>
<td>The OS will:</td>
</tr>
<tr>
<td>The OS selects it, fetches the data from disk, and writes the page into the frame, updating the indexing as needed.</td>
<td>1. select some frame (via a page replacement algorithm)</td>
</tr>
<tr>
<td></td>
<td>2. evicts its contents (if it is dirty) to disk (and mark its indexing information in the page tables as 'absent')</td>
</tr>
<tr>
<td></td>
<td>3. write the new page contents to frame.</td>
</tr>
</tbody>
</table>
Finishing up page transfer

The OS will update the indexing as needed.

Once the page content is in place, the OS restarts the execution of the faulting process at the instruction that faulted.

During this time, since the faulting process is performing I/O, the OS may choose to schedule another process while it waits for the hard disk to supply the missing data (unless executing with interrupts disabled or in a critical section).
Effective Access Time

How much do page faults degrade performance?

EAT is the sum of two components:

1. normal performance: $(1-p) \times \text{access time}$
2. fault performance: $p \times \text{pf overhead}$

where $p$ is the rate of page faults $[0..1]$
EAT Example

Let $p = 0.001$
Let $accessTime = 200 \times 10^{-9}$
Let $overhead = 8 \times 10^{-3}$

$EAT = (1-p) \times accessTime + p \times pf \times overhead$

$= 0.999 \times 200 \text{ ns} + 0.001 \times 0.008$
$= 0.0000001998 + 0.000008$
$= 0.0000081998 = 8.1998 \times 10^{-6}$

$8199.8 \text{ ns} :: 200 \text{ ns} \text{ is } 40.9 \text{ times more costly}$
MISC. TOPICS
Where are the Page Frames?

What data structures does the kernel use to reference pages?

Where are the page tables?

We've already seen the data structures used to index memory regions. We have seen the pgd_t pointer pgd in mm_struct. This field holds the base address of the Page Global Directory (or Page Directory). This field caches the value of cr3 in between quantums.
How Does Linux Track Pages?

```c
/* use the per-pgdat data instead for discontigmem - mbligh */

unsigned long max_mapnr;

struct page *mem_map;

EXPORT_SYMBOL(max_mapnr);
EXPORT_SYMBOL(mem_map);

#define

/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page, though if it is a pagecache page, rmap structures can tell us
 * who is mapping it.
 */

struct page {
    unsigned long flags; /* Atomic flags, some possibly
    atomic_t u_count;    * updated asynchronously */
    union {
        atomic_t mapcount; /* Usage count, see below. */
                        /* Count of ptes mapped in mms,
                        * to show when page is mapped
                        * & limit reverse map searches.
                        */
            struct {
                u64 inuse;
                u64 objects;
            };
        }
    union {
        struct {
            unsigned long private; /* Mapping-private opaque
```
Example of Linux Tracking All Physical Pages

4KB pages
2GB physical memory

4KB = 2^{12}
2GB = 2^{31}

2^{31} / 2^{12} = 524,288

each phys page descriptor ~20 bytes

20*524,288 = 10,485,760 (about 10MB)
typedef union {
    struct {
        unsigned long pte_low, pte_high;
    };
    pteval_t pte;
} pte_t;

extern unsigned long pg0[];

#define pte_present(x) ((x).pte_low & (PAGE_PRESENT | PAGE_PROTNONE))

/* To avoid harmful races, and pte_present() should check only the lower bits */
The Linux OOM Killer

All available memory may be exhausted.
No process can make progress (and thereby free up frames)
__alloc_pages() -> out_of_memory()
   -> select_bad_process() //select
   -> oom_kill_process() //kill

http://lxr.linux.no/#linux+v2.6.27.41/mm/oom_kill.c
How OOM Picks a Victim

Should NOT be:

- a process owned by root; 0, 1, or kernel thread directly accessing hardware
- not be a long-running job

Victim should:

- have a lot of pages to reclaim (+children)
- possibly low static priority
unsigned char fpu_counter;
s8 oomkilladj; /* OOM kill score */

#ifdef CONFIG_BLK_DEV_IO_TRACE
unsigned int btrace_seq;
#endif

unsigned int policy;
cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
int rcu_read_lock_nesting;
int rcu_fliptr_idx;
#endif /* #ifdef CONFIG_PREEMPT_RCU */

#if defined(CONFIG_SCHEDSTATS) || defined(
struct sched_info sched_info;
#endif

struct list head tasks;

struct mm_struct *mm, *active_mm;

/* task state */
struct linux_binfmt *binfmt;
int exit_state;
int exit_code, exit_signal;
int pdeath_signal; /* The signal */

unsigned int personality;
unsigned did_exec:1;
pid_t pid;
pid_t tgid;
Running in Kernel Mode

The kernel “borrows” processes to execute.
‘mm’ is mostly irrelevant:
  nothing below 0xc0000000
  kernel mem does not use memory regions
BUT:
  need reference to Page Tables
  want to avoid TLB and cache flushes
  so borrow those of the user process
  mm=NULL; active_mm=prev->mm->active_mm;
The memory descriptor: struct mm_struct

```c
struct mm_struct {
    struct vm_area_struct * mmap; /* list of VMAs */
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    unsigned long (* get_unmapped_area) (struct file * filp,
        unsigned long addr, unsigned long len,
        unsigned long pgoff, unsigned long flags
    void (* unmap_area) (struct mm_struct * mm, unsigned long addr);
    unsigned long mmap_base; /* base of mmap area */
    unsigned long task_size; /* size of task vm space */
    unsigned long cached_hole_size; /* if non-zero, the larg
    unsigned long free_area_cache; /* first hole of size ca
    pgd_t * pgd;
    atomic_t mm_users; /* How many users with u
    atomic_t mm_count; /* How many references t
    int map_count;
    struct rw_semaphore mmap_sem;
    /*
```

struct vm_area_struct

/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task. A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */

struct vm_area_struct {
    struct mm_struct * vm_mm;    /* The address space we belong to. */
    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;

    pgprot_t vm_page_prot;       /* Access permissions of this VMA. */
    unsigned long vm_flags;      /* Flags, see mm.h. */

    struct rb_node vm_rb;
}