Linux Kernel Support for Enterprise Systems

Simon WINWOOD
University of New South Wales, Australia

INTRODUCTION

THIS TALK:

Goals:

- ➜ Introduce some current work for Enterprise Systems
- ➜ Give an idea of the future direction of the Linux kernel

Contents:

- ➜ Introduction
- ➜ Kernel level scalability (RCU)
- ➜ Security (LSM and SELinux)
- ➜ Application performance (MPSS)

IBM: ENTERPRISE LINUX GROUP:

- ➜ Goal is to improve linux kernel for enterprise systems
- ➜ Multi-Queue scheduler (SMP)
- ➜ Block I/O performance
- ➜ Fast locking
- ➜ Multiple page size support (this talk)

UNSW: OPERATING SYSTEMS, EMBEDDED AND DISTRIBUTED SYSTEMS RESEARCH GROUP:

- ➜ General Operating System Research.
- ➜ SASOS features in IA64 Linux
- ➜ GELATO: Large disk and file support
- ➜ GELATO: Large page support (IA64)

ENTERPRISE SYSTEM CHARACTERISTICS:

Mission Critical

- ➜ May be in a high-risk environment (i.e. Internet)
- ➜ Sensitive data
- ➜ Need control over exactly what applications can do

Multi Processor

- ➜ Large (> 2) numbers of processors
- ➜ Threads share memory — lock contention
- ➜ Memory coherence is potentially expensive

Large Memory Sizes

- ➜ Applications have larger working sets
- ➜ Applications use more memory

SECURITY

LSM AND SELINUX:

`http://lsm.immunix.org/`

`http://www.nsa.gov/selinux/`

BACKGROUND:

SELinux was introduced at the March 2001 2.5 Kernel Summit.

Added *Non-Discretionary Access Control* to Linux.

Implemented as a patch against vanilla Linux — added hooks to various functions.

Linus suggested a more generic approach: add hooks which call functions in a module

➜ LSM

WHAT'S WRONG WITH *chmod*?:

Gives complete control to the `root` user.

Applications which require some privilege are granted all rights.

- ➜ If the application gets hacked, an intruder can take control over the whole system.
- ➜ Why should `sendmail` (for example) be able to add users?

There is no *administrator* enforced security policy (i.e., no *Mandatory Access Control* (MAC)).

- ➜ A company may wish to restrict the sharing of certain sensitive information between employees.

LINUX SECURITY MODULES:

Provide generic hooks for implementing an arbitrary policy.

Security policies are loaded using kernel modules.

Provides very fine-grained security decisions.

Security modules may allow or disallow an access.XS

Available modules:

- ➜ SELinux
- ➜ DTE Linux
- ➜ Openwall kernel patch
- ➜ POSIX.1e capabilities
- ➜ Linux Intrusion Detection System (LIDS)
- ➜ Default (super-user)

SECURITY ENHANCED LINUX (SELINUX):

SELinux is a proof-of-concept for MAC under Linux.

It contains policy-independent security enforcement and a replaceable security server.

The example security server implements:

- ➜ Identity-Based Access Control (IBAC)
- ➜ Role-Based Access Control (RBAC)
- ➜ Type Enforcement (TE)

(don’t worry, I will explain what these mean)

WHAT DOES IT ALL MEAN?:

Mandatory Access Control (MAC)

- ➜ Allows for a global security policy, enforced over the *whole* system.

Identity-Based Access Control (IBAC)

- ➜ Similar to a Linux UID in that they represent a user.
- ➜ Orthogonal to a UID in that they aren't changed by `su` etc.

Role-Based Access Control (RBAC)

- ➜ An *identity* is restricted to a set of roles.
- ➜ An *identity* may assume a role only through certain programs.

Type Enforcement (TE)

- ➜ Every object (file, socket, process, etc.) in the system is assigned a type
- ➜ Each *role* is associated with a set of allowable types
- ➜ *Roles* give much coarser control than *types*

SELINUX AND LSM:

① On a security check, LSM calls the appropriate security hook.

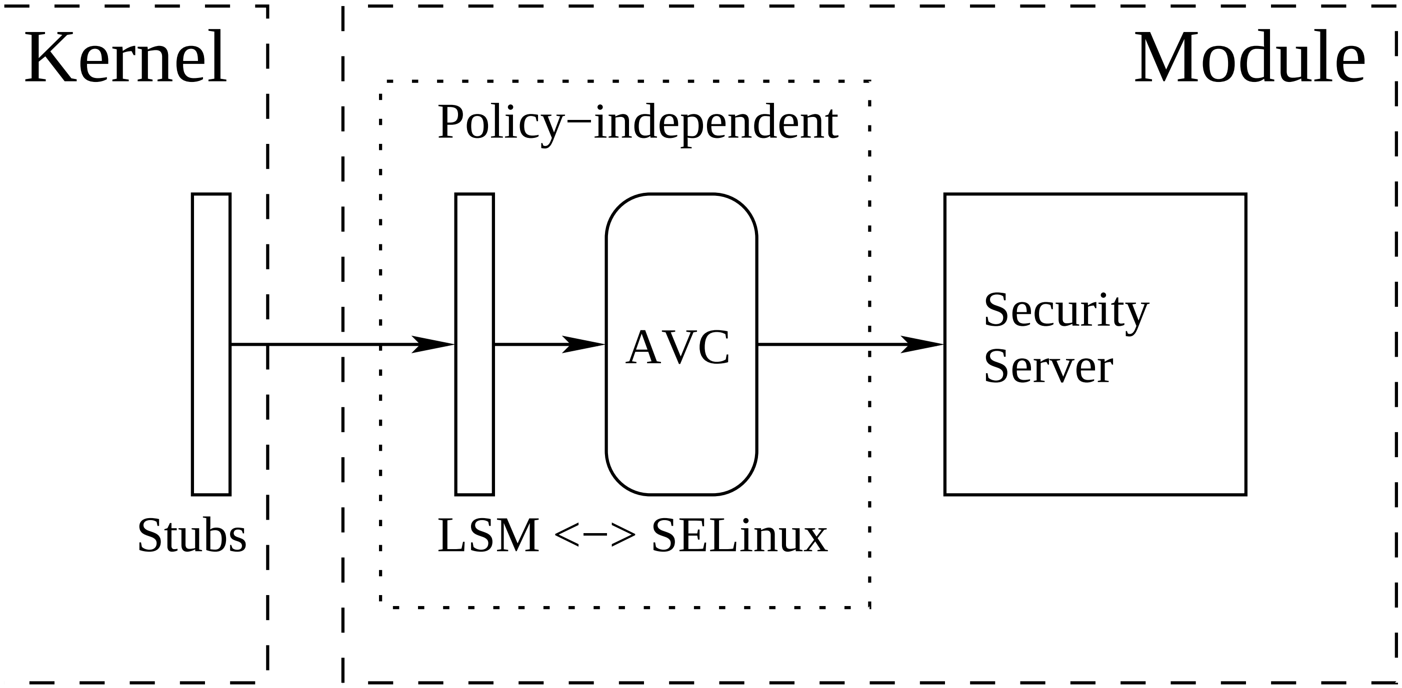
② SELinux infers the source and target Security ID (SID) from the LSM security handle.

③ SELinux looks up the <source, target, access> vector in the policy-independent access vector cache (AVC) — assume no match.

④ SELinux consults the security server to see if the <source, target, access> vector is allowed.

⑤ Operation is allowed or denied as appropriate.

SELINUX AND LSM (CONT.):



SOME EXAMPLES:

Protecting physical disks:

```
allow fsadm_t fsadm_exec_t:process
      { entrypoint execute };
allow fsadm_t fixed_disk_device_t:blk_file
      { read write };
allow initrc_t fsadm_t:process transition;
allow sysadm_t fsadm_t:process transition;
```

Restricting module insertion:

```
allow insmod_t insmod_exec_t:process
      { entrypoint execute };
allow insmod_t self:capability sys_module;
allow sysadm_t insmod_t:process transition;
```

SOME EXAMPLES: RESTRICTING `sendmail`:

```
allow sendmail_t etc_aliases_t:file
      { read write };
allow sendmail_t etc_mail_t:dir
      { read search add_name remove_name };
allow sendmail_t etc_mail_t:file
      { create read write unlink };
allow sendmail_t smtp_port_t:tcp_socket name_bind;
allow sendmail_t mail_spool_t:dir
      { read search add_name remove_name };
allow sendmail_t mail_spool_t:file
      { create read write unlink };
allow sendmail_t mqueue_spool_t:dir
      { read search add_name remove_name };
allow sendmail_t mqueue_spool_t:file
      { create read write unlink };
```

WHAT DOES IT COST?:

LSM:

- ➜ Micro-benchmark (`lmbench`): Worst case 7.2%, best case 0–2%
- ➜ Macro-benchmark (kernel compile): negligible.

SELinux:

- ➜ Micro-benchmark (`lmbench`): Worst case 33%, best case 1–2%
- ➜ Macro-benchmark (kernel compile): 4%.
- ➜ Macro-benchmark (`WebStone 2.5`): negligible.

READ-COPY-UPDATE (RCU)

`http://lse.sourceforge.net/locking/rcupdate.html`

LOCKING IN THE KERNEL:

In general, if 2 threads may access the same data at the same time, locking is required.

For short lived locks, basic primitive is the *spin lock*:

```
spinlock(lock) {
  success = false;
  while(success == false) {
    begin_atomic {
      if(lock == 0) {
         lock = 1;
         success = true;
      }
    }
  }
}
```

WHAT IS THE PROBLEM?:

If two or more processors try to access the same lock, one processor will fail.

If neither processor is modifying the list, then a *read-write* lock can be used

- ➜ This lock allows multiple readers to hold the lock at any time, or 1 writer
- ➜ The lock variable still needs to be modified

The lock needs to be locked, even if no other processor will attempt to use it.

- ➜ that is, even in the common case, the lock variable still needs to be modified
- ➜ accessing a dirty cache line on another processor can be expensive!

SOLUTION: READ-COPY-UPDATE (RCU):

To modify a list, update all global references to the object, and delete when no references remain.

- ➜ Big problem: how to determine if an object is still referenced.

Note that a thread cannot hold a lock across a context switch

- ➜ After each processor has had a context switch, we can delete the object.

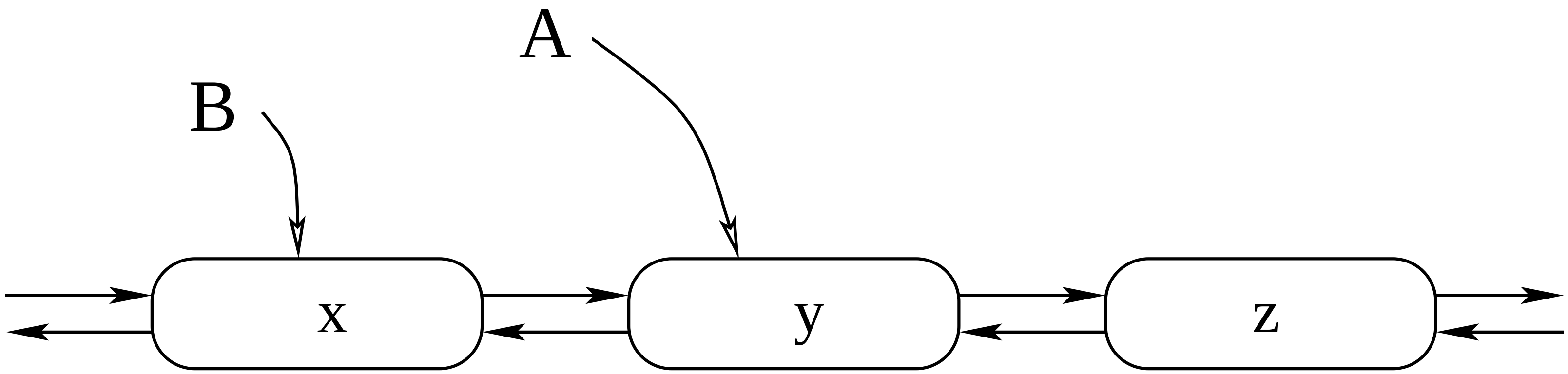
QUIESCENT POINTS:

A context switch is an example of a *quiescent point*

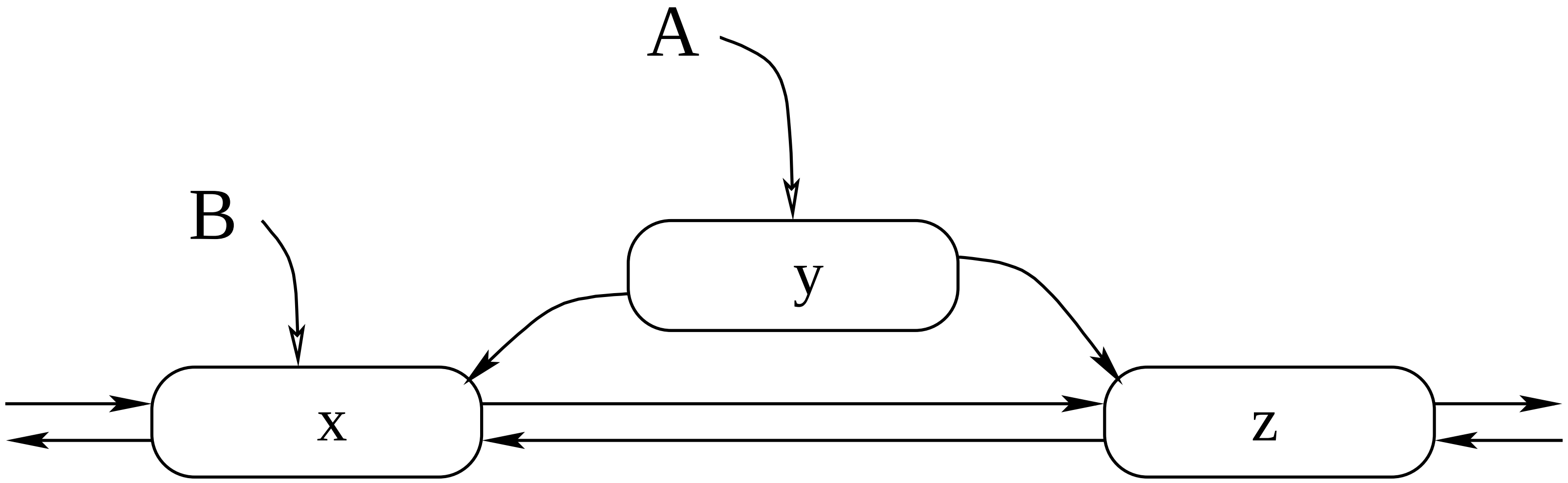
A *quiescent point* is any point at which the current processor does not hold any references to shared objects. Other *quiescent points* include:

- ➜ Idle loop execution
- ➜ User-mode execution
- ➜ Daemon execution

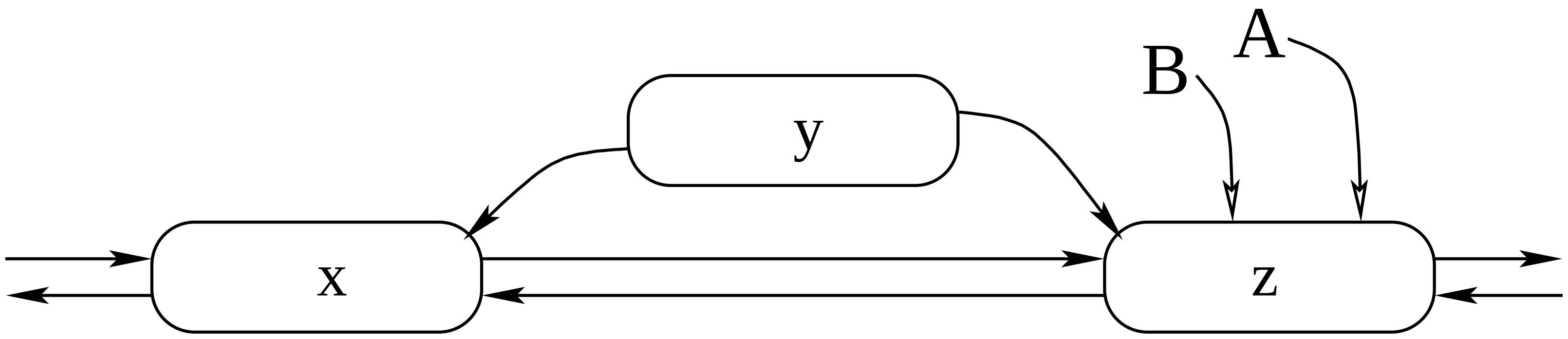
HOW THEY WORK:



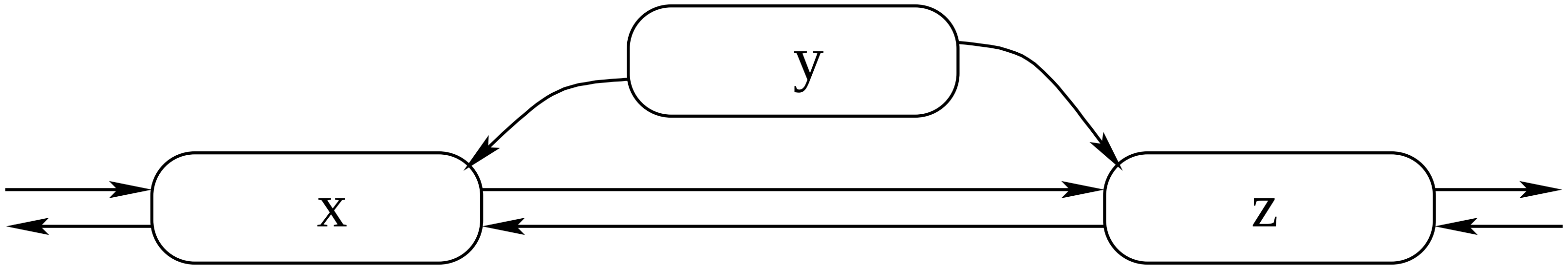
HOW THEY WORK:



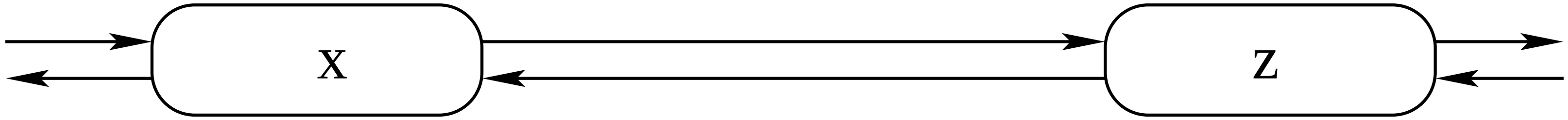
HOW THEY WORK:



HOW THEY WORK:



HOW THEY WORK:



IMPLEMENTATION:

Instead of freeing an object, it is added to an RCU queue

When the kernel has determined that each processor has gone through a quiescent point, all objects on the list are freed

Various different implementations exist, differing in how they determine quiescent points

- ➜ there is also an implementation for the pre-emptible kernel
- ➜ implementations differ in the overhead and latency of object deletion

AN EXAMPLE (WITHOUT RCU):

```
int lookup_key(list_t *list, key_t key)
{
  int found = 0;
  list_t *this;

  spin_lock(&list->lock);
  for(this = list->next; this != list; this = this->next)
     if(this->key == key) {
       found = 1;
       goto out;
     }
out:
  spin_unlock(&list->lock);
  return found;
}
```

AN EXAMPLE (WITHOUT RCU) (CONT.):

```
void delete_element(list_t *list, list_t *el)
{
  spin_lock(&list->lock);
  el->prev->next = el->next;
  el->next->prev = el->prev;
  spin_unlock(&list->lock);
  kfree(el);
}
```

AN EXAMPLE (WITH RCU):

```
int lookup_key(list_t *list, key_t key)
{
  list_t *this;

  for(this = list->next; this != list; this = this->next)
     if(this->key == key)
       return 1

  return 0;
}
```

AN EXAMPLE (WITH RCU) (CONT.):

```
void delete_element(list_t *list, list_t *el)
{
  spin_lock(&list->lock);
  el->prev->next = el->next;
  el->next->prev = el->prev;
  spin_unlock(&list->lock);
  call_rcu(&el->rcu_head, my_kfree, el);
}

void my_kfree(list_t *el)
{
  kfree(el);
}
```

PROJECTS USING RCU:

The following projects are using RCU:

- ➜ Directory Entry scalability
- ➜ Hot-Plug processor support
- ➜ Module unloading and cleanup
- ➜ Scalable file descriptor management
- ➜ IPV4 route cache lookup

PERFORMANCE:

RCU performs best when the majority of accesses to a list are reads

The following results have been reported:

- ➜ FD management: 30%
- ➜ DCache management: 25%

RCU will become more important as linux scales to larger numbers of CPUs

MULTIPLE PAGE SIZE SUPPORT

INTRODUCTION

MOTIVATION:

Enterprise Linux Group’s focus: performance

- ➜ Linux for Enterprise Computing (scalability, funct..)
- ➜ Linux for Scientific Computing

Applications’ working sets are outstripping TLB coverage

Evidence that large pages might improve performance: previous work USENIX-98

- ➜ Ganapathy et al.: SGI IRIX-6.4
- ➜ Subramaniam et al.: HP HP-UX

GOALS:

Evaluate large pages: are they really worth it?

Architecture independent design

Support for multiple (concurrent) page sizes

Only incur large page overhead when needed

Minimise modifications

- ➜ Easier to test/understand
- ➜ Can extend implementation if required
- ➜ Higher likelihood for adoption

BACKGROUND: GENERAL VIRTUAL MEMORY (VM)

TLBS:

A TLB caches virtual to physical mappings

Accessed for every memory instruction

- ➜ Critical to overall performance => small and fast
- ➜ Physically indexed caches require translation before lookup

TLB misses are expensive!!!!!!

Modern CPUs support larger page sizes for greater TLB coverage

- ➜ E.g. Pentium 4: 64 entry TLB coverage of 256K @ 4K pages, 256M @ 4M pages

LARGE PAGES:

Pros:

- ➜ Primary: Reduce TLB misses by increasing coverage
- ➜ Secondary: Increase I/O bandwidth utilisation and lower total I/O time, if I/O supports it
- ➜ Secondary: Reduce memory requirements for page tables — E.g. A 128M mapping only requires 32 * 4M-PTEs vs. 32K * 4K-PTEs => 32 pages of unswappable memory and Linux is not a swappable kernel !

LARGE PAGES:

Cons:

- ➜ Assumes some page locality
- ➜ Applications with small working sets or very sparse working sets will not benefit
- ➜ Increased kernel complexity
- ➜ Increased page fault latency
- ➜ Higher granularity of resource accounting
- ➜ Using large pages may waste space needlessly

Other:

- ➜ Architectures usually support a range of page sizes
  - ➜ IA32: 4K, 4M/2M
  - ➜ IA64: 4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M
  - ➜ UltraSparc II: 8K, 64K, 512K, 4M
  - ➜ Similar for Alpha, PA-RISC, MIPS, some PPCs, etc.
- ➜ Aligned in both physical and virtual space — E.g. 4M pages aligned to a 4M phys. address

BACKGROUND: LINUX VM

REPRESENTING MAPPINGS (VIRT. -> PHYS):

2/3-level hierarchical page tables

Regions are described with VMA data structures

- ➜ Start/end of region
- ➜ access rights
- ➜ backing file (if any)
- ➜ behavior hints
- ➜ `nopage` method for establishing mappings

Page frames are represented by the `page` data structure

- ➜ Contains flags such as `dirty`. `referenced`, `locked`
- ➜ Used by the swap subsystem to choose victim pages

THE PAGE CACHE:

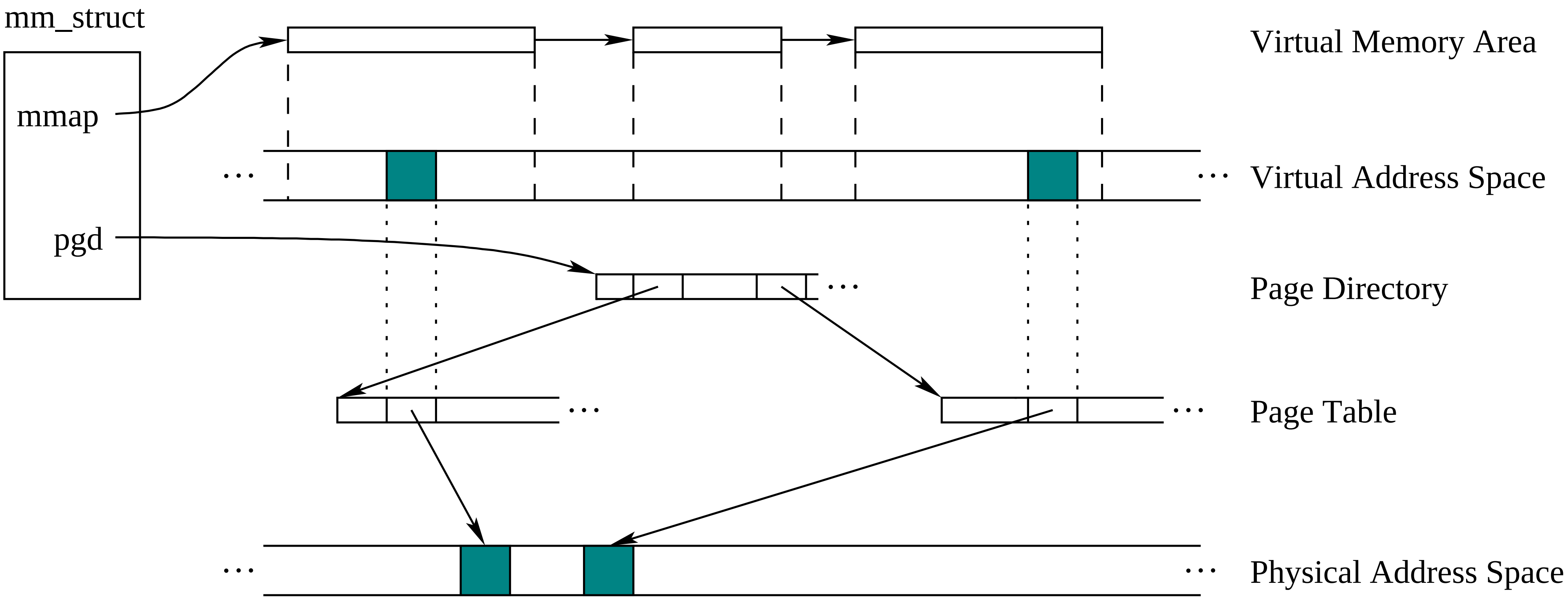
Caches file data: allows multiple tasks to map a file using the same physical page

Used by setting `nopage` to `filemap_nopage` in VMA

Also implements `read` and `write`

Used by most filesystems

LINUX VM:


mm_struct
mmap
pgd
Virtual Memory Area
· · ·
· · ·
Virtual Address Space
· · ·
Page Directory
· · ·
· · ·
Page Table
· · ·
· · ·
Physical Address Space

IMPLEMENTATION:

Chose application hints over kernel heuristics

- Application may know more about its behavior than kernel
- Much simpler
- Implies modifications to applications or libraries

Page size is per-VMA

- Can split a VMA if the application requests a sub-region

Applications use the `madvise` system call

- Added a new `setpageorder(o)` operation

REPRESENTING SUPERPAGES:

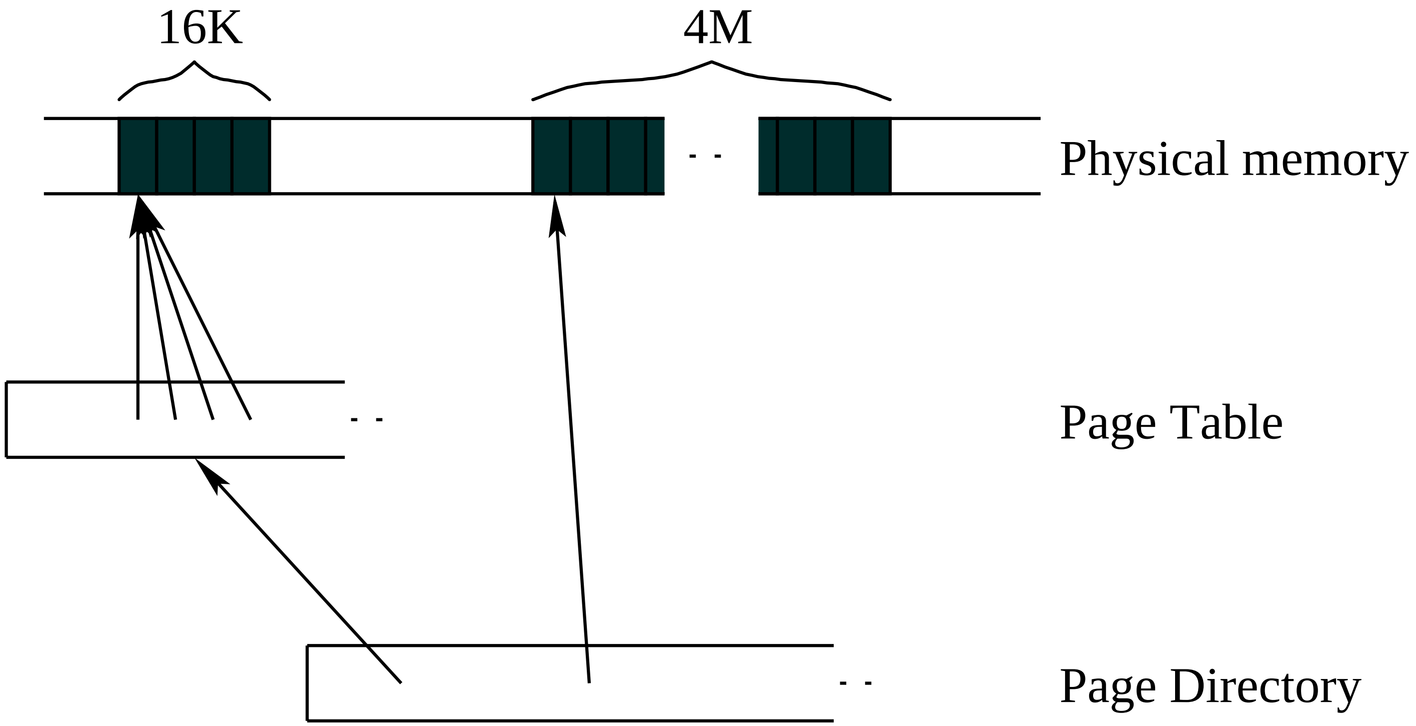
Don’t want to implement a totally new page table (PT)

Constrained by i386 PT structure

Use existing PT with some modifications:

- ➜ Store the page size in every page in the superpage
- ➜ If a page size is greater than the number of PTEs, store the PTE in the next level up
- ➜ Need to modify all kernel functions which modify the PT

REPRESENTING SUPERPAGES (CONT.):



REPRESENTING SUPERPAGES (CONT.):

Store the largest page size a frame belongs to in `page` data Structure

- ➜ A superpage is a sequence of contiguous `page` data structures

Operations which need to use the superpage as a whole use the first page

- ➜ referencing the page
- ➜ dirtying the page
- ➜ backing file information

Operations which are per-page are unchanged

- ➜ wait queue
- ➜ `locked, error, uptodate`

ALLOCATION OF SUPERPAGES:

Basic idea is a special *large page zone* (pool)

Avoids problem of OS “polluting” pages with kernel data (unswappable)

Obviously a short-term solution

- May be OK for some dedicated applications
- The *rmap* patch may be useful

WHAT HAPPENS ON A PAGE FAULT:

① Application accesses VA -> TLB miss

② Hardware or kernel looks up page table -> Page fault

③ Kernel looks up VMA corresponding to fault addr.

④ Kernel checks whether it is a new mapping (not swapped out, etc.)

⑤ Kernel scans page table for an empty region <= `vma->vm_order`

⑥ Kernel calls `nopage` method (`filemap_nopage`)

⑦ Pagecache checks if corresponding file data is cached.

⑧ Pagecache allocates memory and reads each page in.

⑨ Kernel inserts new page into the pagetable and restarts faulting instruction

MICROBENCHMARK:

Show efficacy of large Page size in a controlled setup

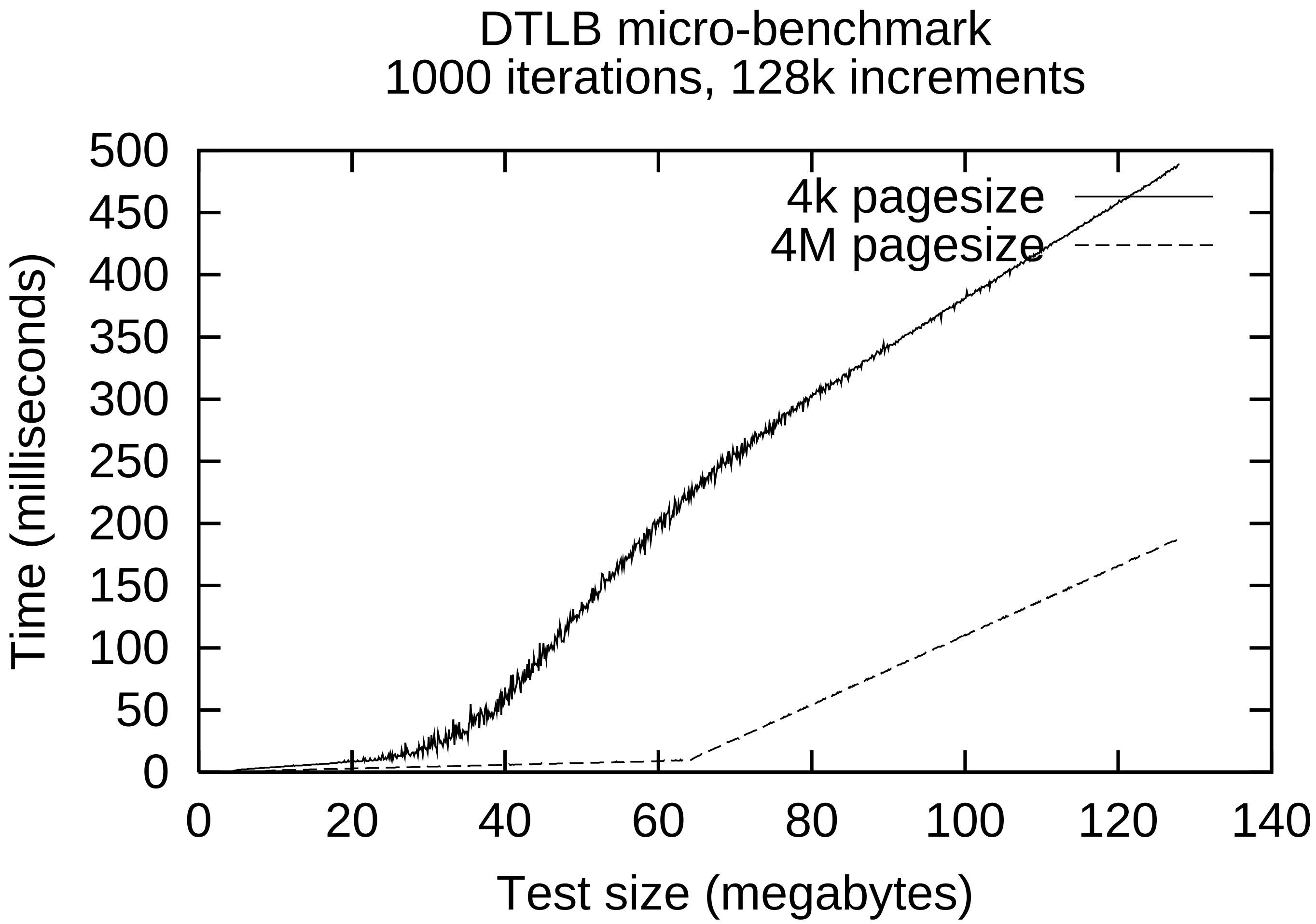
System Pentium-4:

- ➜ 64-byte cachelines
- ➜ L2-cache: 256K, 64 B CL, 8-way Set-Associate
- ➜ D-L1 cache: 8K, 64 B CL, 4-way Set-Associate
- ➜ D-TLB-4K: 64 entries, Fully-Associate
- ➜ D-TLB-4M: Shared with 4K D-TLB.

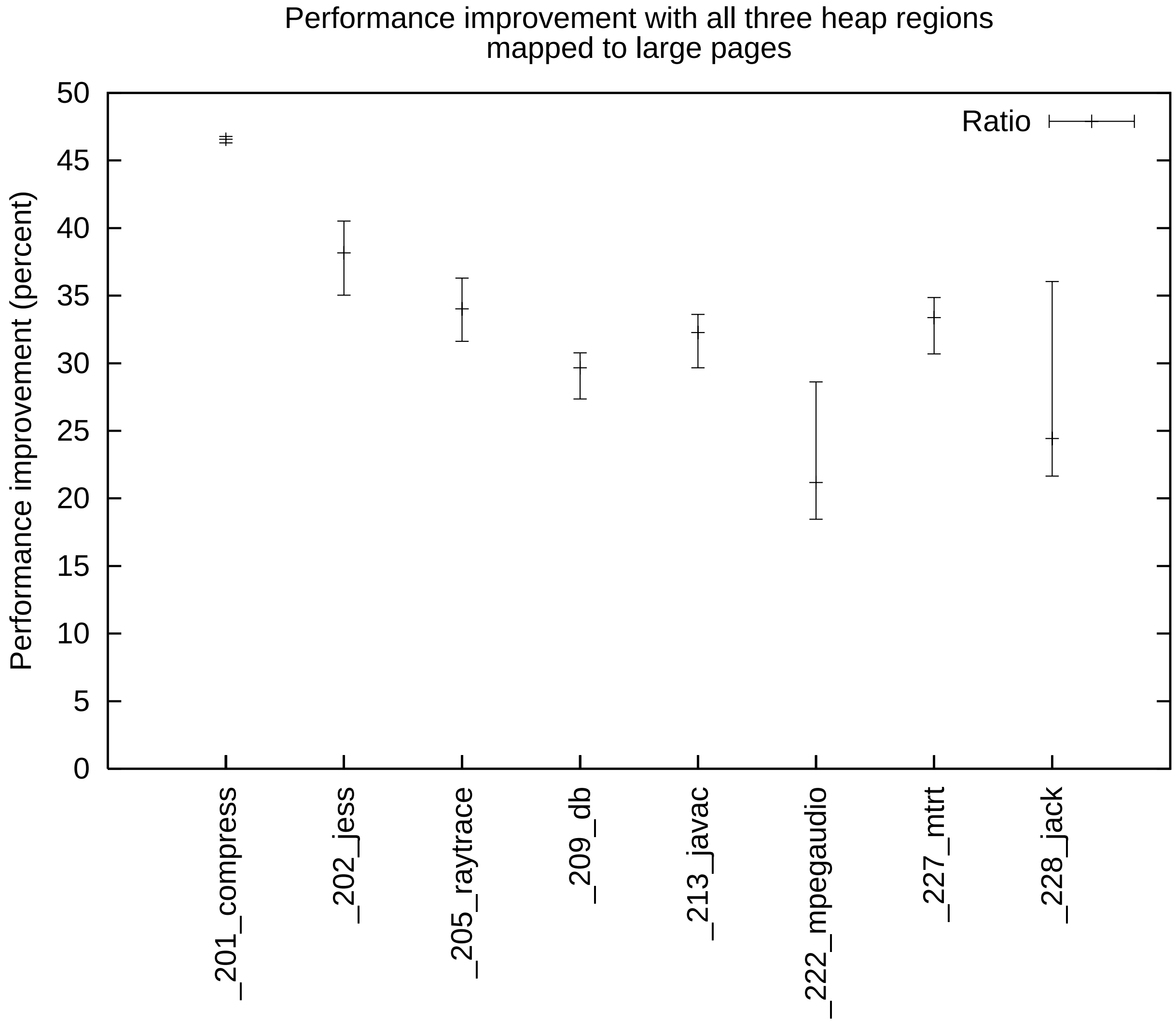
Stride through memory such that each load results in new data cacheline and new cacheline for PTE

- ➜ avoid reuse of PTEs
- ➜ Access every 16th page (4-bytes per PTE) + 64 bytes

MICROBENCHMARK (CONT.):



SPECJVM98:


Performance improvement with all three heap regions mapped to large pages
Performance improvement (percent)
0
5
10
15
20
25
30
35
40
45
50
Ratio
_201_compress
_202_jess
_205_raytrace
_209_db
_213_javac
_222_mpegaudio
_227_mtrt
_228_jack

SPECINT2000:

Benchmark	Improvement (%)
gzip	12.3
vpr	16.7
gcc	9.3
mcf	9.4
crafty	15.2
parser	16.3
eon	12.1
gap	5.9
vortex	22.2
bzip2	14.4
twolf	12.5

WHAT'S NEXT?:

Moving upgrade decisions into the kernel

Splitting pages for reference and dirty accounting

Better memory management

- ➜ Page amalgamation daemon
- ➜ Intelligent swapping
- ➜ Intelligent allocation