Linux Memory Management

Virtual memory layout: a closer look

- ➔ Applications require memory with different properties
  - ◆ access permissions
  - ◆ sharing
  - ◆ file backed vs. anonymous
  - ◆ dynamically sized
- ➔ /proc/{pid}/maps and mmap() system call

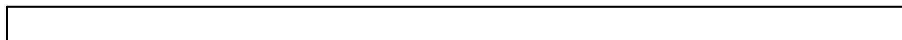
- ➔ Why OS should worry how user-space virtual addresses are managed?
  - ◆ let a user-space library handle it
  - ◆ only virtual to physical translation is managed by OS
  - ◆ possible?

Managing virtual memory (user space address)

```
{
mmap(size, type, permissions)+
…………..
munmap( )+
…………….
}
```

Virtual address space management (kernel)

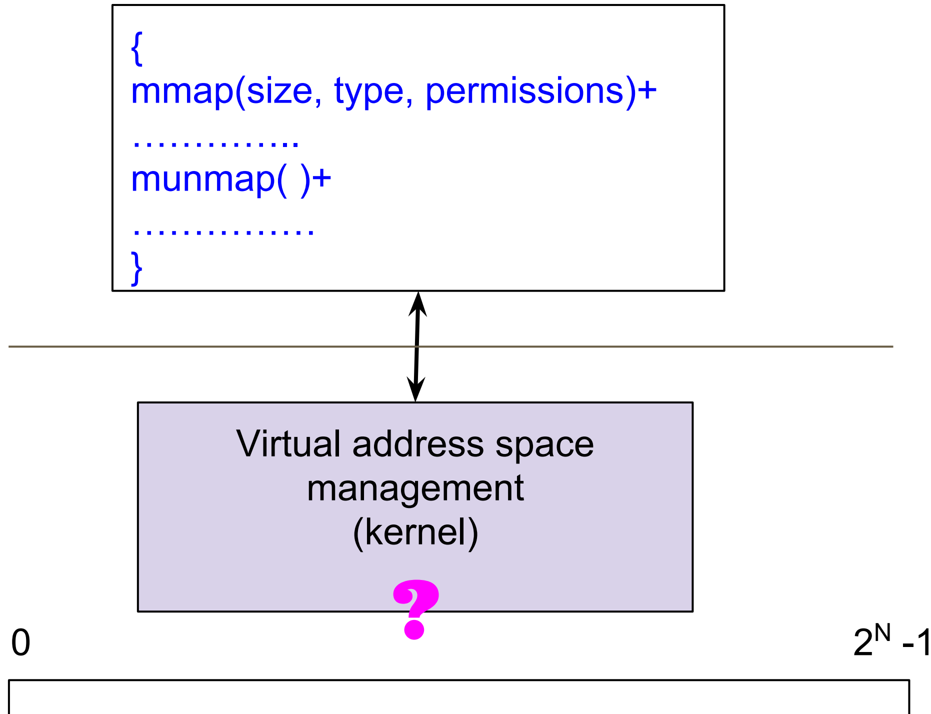
0 2^N -1



Some design consideration

- virtual address space is quite large (for 64-bit)
- can not assume virtual address usage size
- efficiency concerns: CPU and Memory
- address space requirements → hardware structures (MMU)

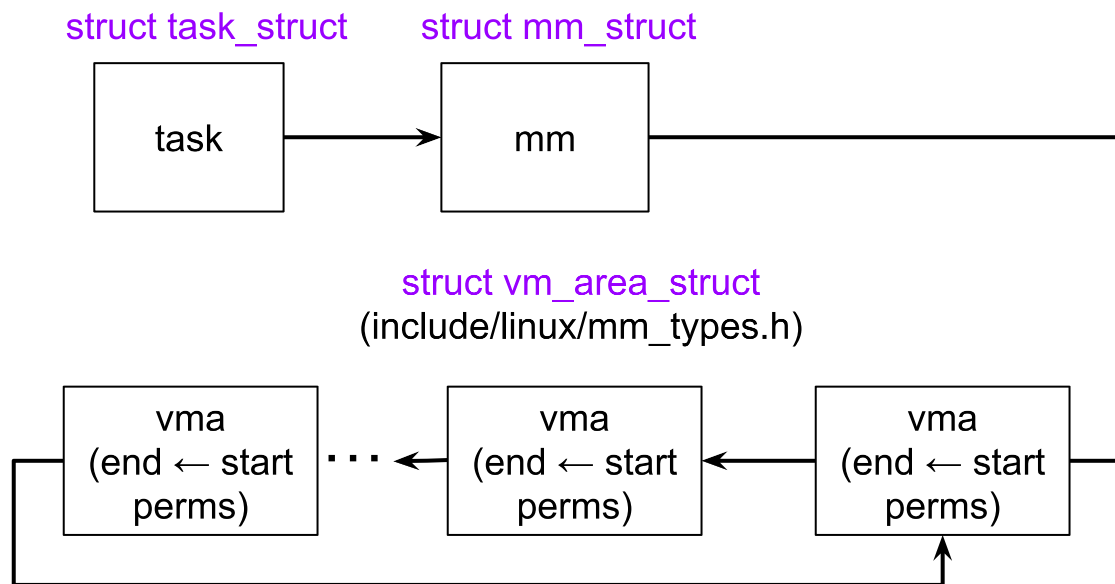
Managing virtual memory (user space address)



Virtual address space management alternatives

- ➔ contiguous allocation based on memory region type
  - inflexible
  - scalability issues
- ➔ sparse allocation
  - sorted list of used ranges
  - scalability issues
    - Can be solved using balanced search trees

How linux does it?



- ➔ start and end never overlaps between two vm areas
- ➔ can merge/extend vmas if permissions match
- ➔ linux maintains both rb_tree and a sorted list (see mm/filemap.c)

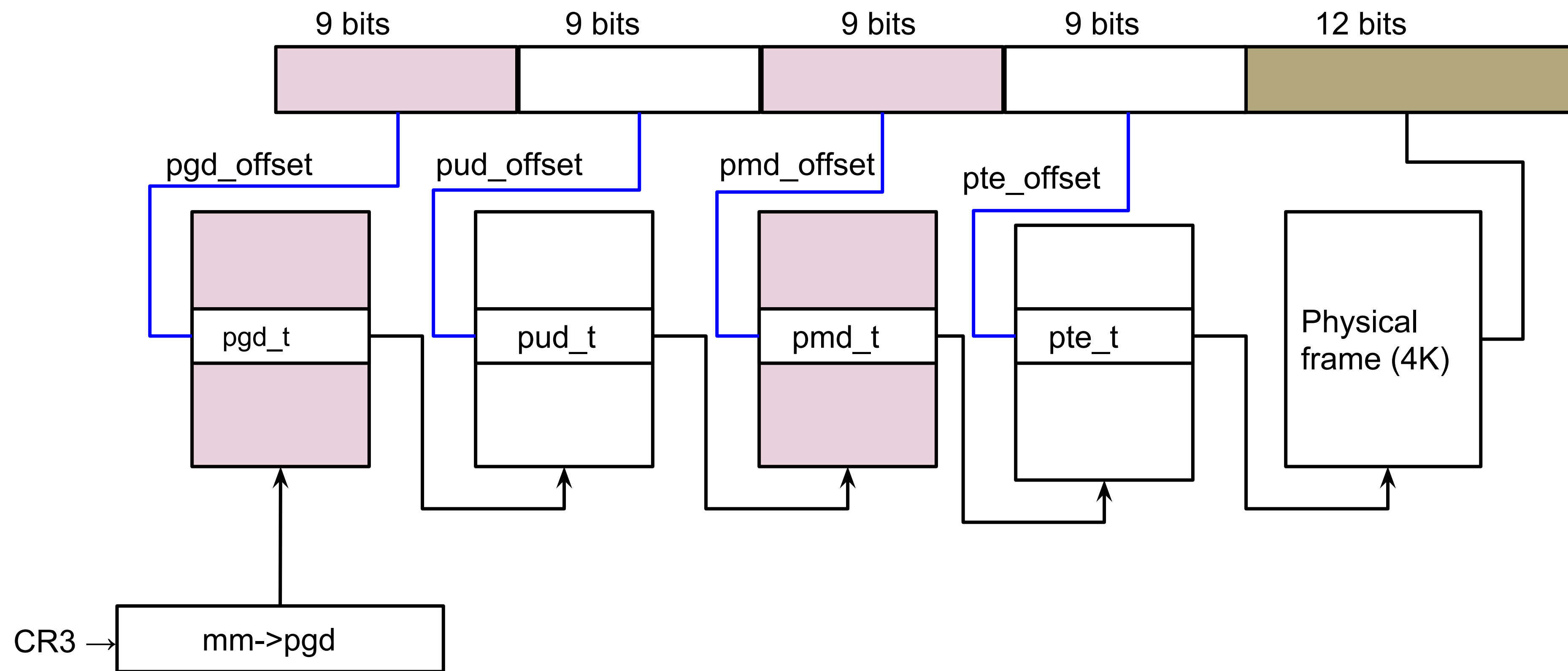
Example usage

- mmap( ), munmap( ), mremap( ) system calls
  - some useful calls: find_vma( ) , get_unmapped_area( ), vma_merge ()
  - can be found in mm/mm.c
- Page fault handler
  - require vm_area access permissions to fix the page fault
  - Ex: fault handling for a read-only vm_area vs. read-write vm_area
- Feature: vma-area specific page fault handler
  - struct vm_operations_struct *vm_ops
  - mechanism to register call backs on page fault (and some other events)

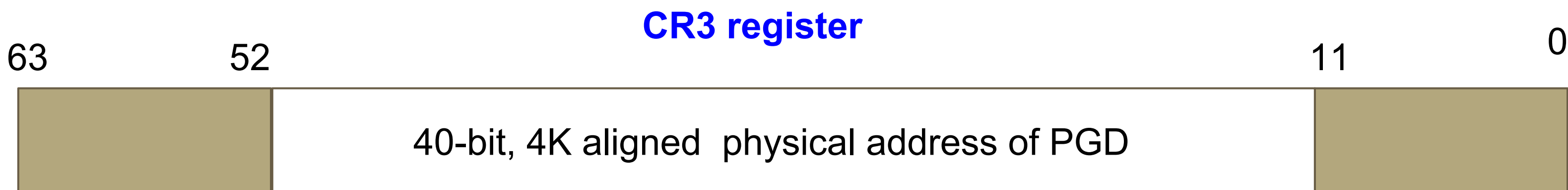
Demand paging: background

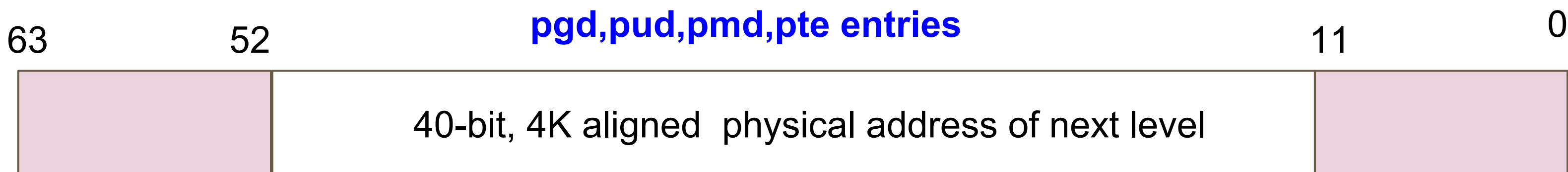
- Why not use physical addressing?
- Considering application expectation of virtual address space flexibility
  - Why not use segmentation-only design?
  - Why use paging?
- Challenges in paging
  - Size of translation meta-data
  - Additional memory accesses during translation

4-level page tables (48-bit virtual address)



X86_64 page table entries (48-bit)





Some important flags

0 (present/absent) 1 (read/write) 2 (user/supervisor), 5(accessed) 7(huge page) 63(execute permissions)

*Source: Intel manual Vol: 3A 4.5

Walk page table in s/w for fun and profit

```
#define PAGE_SIZE 4096
#define PAGE_SHIFT 12

get_pa(task_struct *tsk, unsigned long va)
{
    unsigned long address = (va >> PAGE_SHIFT) << PAGE_SHIFT;
    mm = task->mm;
    pgd = pgd_offset(mm, address);
    pud = pud_offset(pgd, address);
    pmd = pmd_offset(pud, address);
    pte = pte_offset(pmd, address); /*pte_flags, pte_none ... */
    pfn = pte_pfn(pte);
    return (pfn << PAGE_SHIFT) + va - address)
}
```

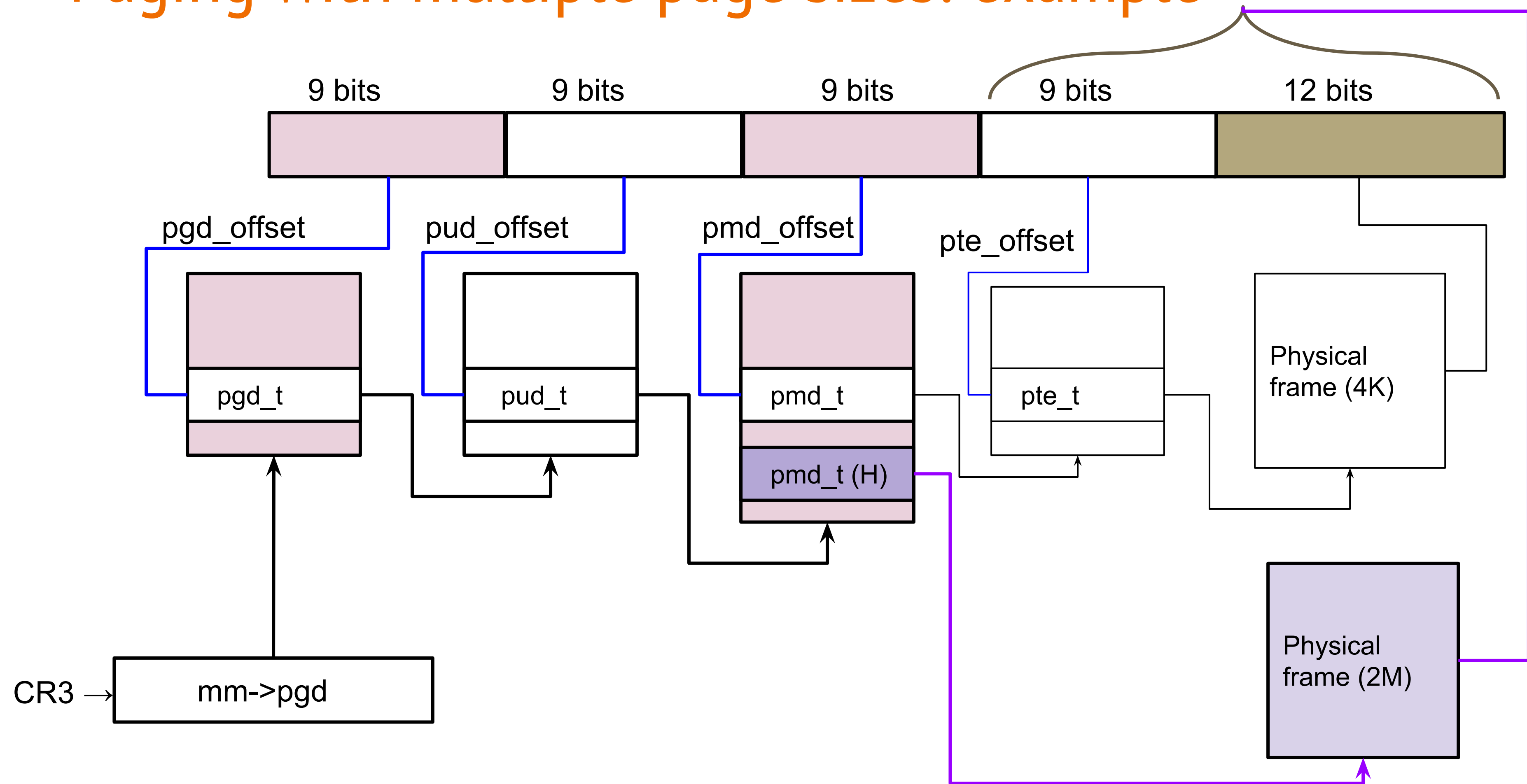
Page table translations: debate

- ➔ Page table physical frames themselves may not be swapped? why or why not?
- ➔ How much memory required to maintain page tables in a system? What are the determinant parameters?
- ➔ How can paging performance be optimized? In the proposed optimizations
  - ◆ What are the tradeoffs?
  - ◆ What are the assumptions?

Multiple page size support (4K, 2M and 1G)

- ➔ Same process can have multiple page sizes
  - ◆ All depends on how page table is organized
- ➔ Strategy: collapse page tables starting from lowest level (pte)
  - ◆ pte level is removed and pmd addresses 21-bits = 2MB
  - ◆ pte and pmd are removed, pud addresses 30-bits = 1GB
- ➔ Recall the huge page bit (PS bit) in page table entries
- ➔ What are the challenges?

Paging with multiple page sizes: example



Page faults

- ➔ Synchronous or asynchronous?
- ➔ When can it happen? What are the triggers?
- ➔ Do we require any information regarding the faulting task? What and why?
- ➔ Why do we require page walk in software?

Page fault handling

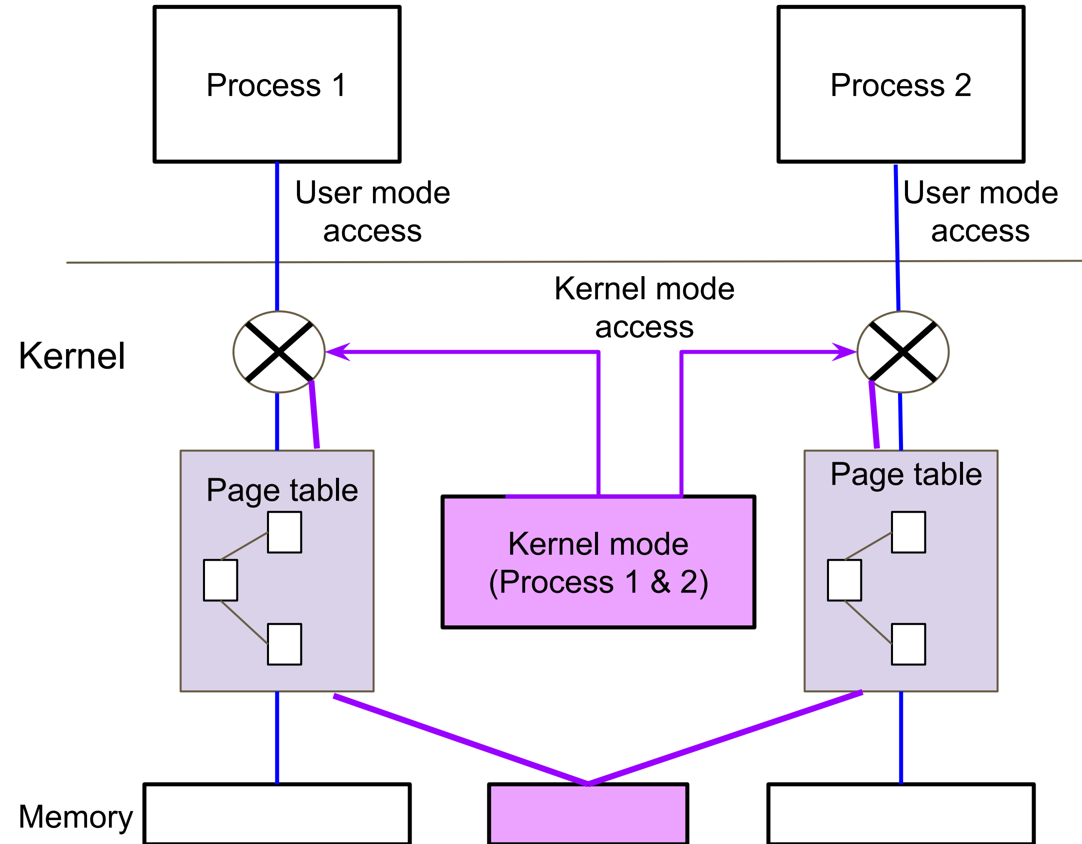
- Page fault is an exception (#14 in x86)
  - Translation missing in any level of page table hierarchy
  - Translation present, but access rights do not permit access
  - Error code provided by CPU to distinguish the above scenario
- Page fault handling
  - CR2 register in x86
  - Perform software walk to check missing entries
  - Allocate missing entries in the page table hierarchy
  - May require new physical allocation
- Code reference
  - arch/x86/mm/fault.c   do_page_fault( )

What about kernel virtual addresses?

Addressing in kernel

- Kernel executes on behalf of ….
- Kernel state is accessible from all processes in kernel mode
  - chardev example: P1 can read( ) a buffer allocated by a write( ) call from P2
  - How buffer (a virtual address) is accessible from P1's kernel context?
- Alternate 1: During process entry into kernel, change the page tables
  - Manage one or more kernel page table
  - Switch the page table when entering kernel mode
  - Why not design a system like this?
- Alternate 2: kernel memory mapped to every process page table
  - Pros and cons?

Monolithic kernel: every process has the same heart!

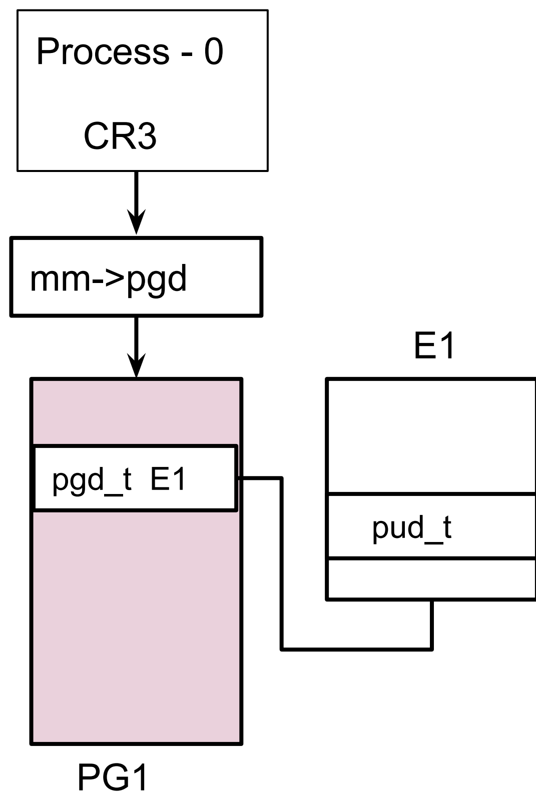


- ➔ Kernel virtual address mapping should be present in both process page tables.
- ➔ How to design?
  - Kernel can perform dynamic allocation→ should reflect in every process page table
  - Processes are dynamically created and destroyed

Linux strives on family values!

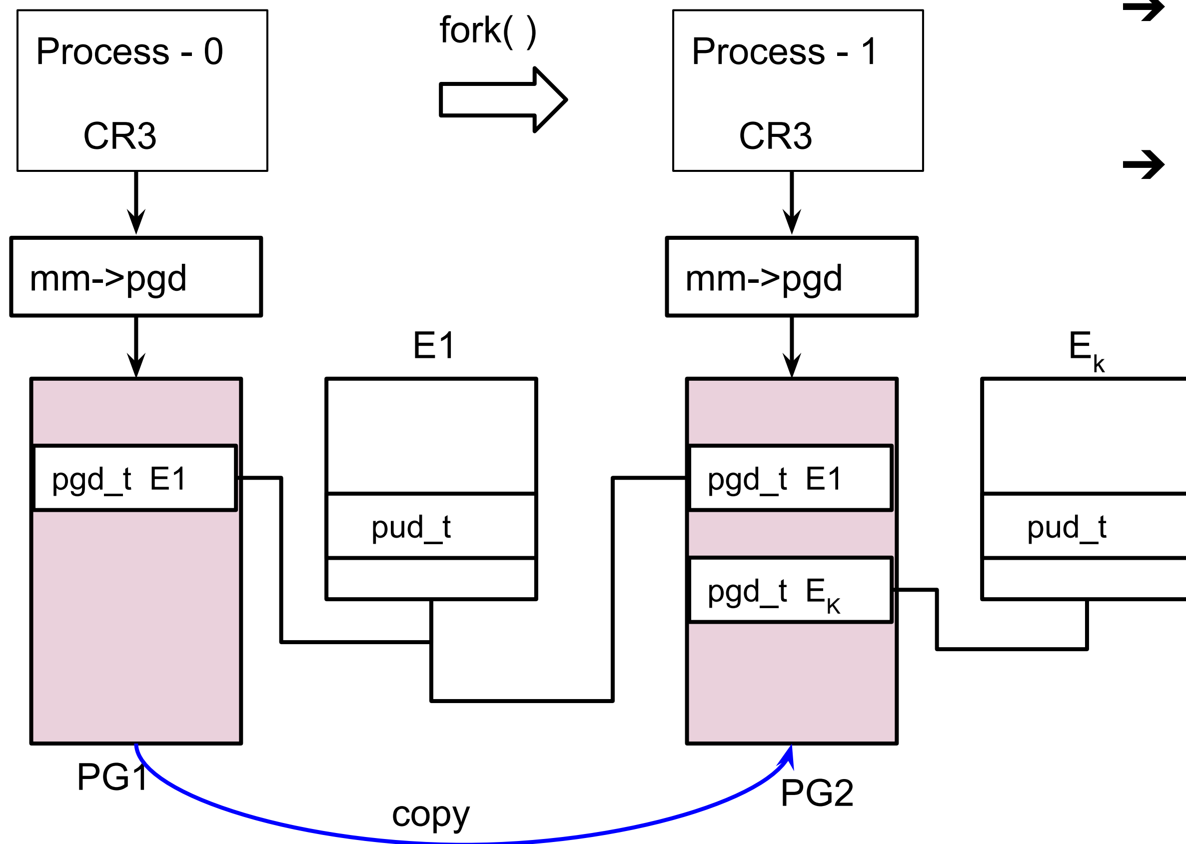
- A child process page table inherits kernel mappings of the parent
- By implication, the inheritance tree is rooted at the first process
- What about changes in mapping?
  - Can be done by any process in kernel context
  - Update mapping in every process?
- What about reservation (and propagation) of some pgd entries (pointing to pud-level translation page frames) for kernel virtual addresses?
  - Will it work?
  - What happens when kernel virtual address mappings change?

A possible solution



- ➔ One (or more) entries in PGD-level (level-4)
- ➔ VA-range covered by one entry = ?
- ➔ How many entries?

A possible solution



- ➔ PG2 is a copy of PG1 (initially)
- ➔ Any restriction on kernel usable VA range?

All the best !!