<Slides download>
http://www.pf.is.s.u-tokyo.ac.jp/class.html

Advanced Operating Systems

#4

Shinpei Kato
Associate Professor

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo

Course Plan

- Multi-core Resource Management
- Many-core Resource Management
- GPU Resource Management
- Virtual Machines
- Distributed File Systems
- High-performance Networking
- Memory Management
- Network on a Chip
- Embedded Real-time OS
- Device Drivers
- Linux Kernel

Schedule

1.	2018.9.28	Introduction + Linux Kernel (Kato)
2.	2018.10.5	Linux Kernel (Chishiro)
3.	2018.10.12	Linux Kernel (Kato)
4.	2018.10.19	Linux Kernel (Kato)
5.	2018.10.26	Linux Kernel (Kato)
6.	2018.11.2	Advanced Research (Chishiro)
7.	2018.11.9	Advanced Research (Chishiro)
8.	2018.11.16	(No Class)
9.	2018.11.23	(Holiday)
10.	2018.11.30	Advanced Research (Kato)
11.	2018.12.7	Advanced Research (Kato)
12.	2018.12.14	Advanced Research (Chishiro)
13.	2018.12.21	Linux Kernel
14.	2019.1.11	Linux Kernel
15.	2019.1.18	(No Class)
16.	2019.1.25	(No Class)

Process Management

Abstracting Computing Resources

/* The case for Linux */

Acknowledgement:

Prof. Pierre Olivier, ECE 4984, Linux Kernel Programming, Virginia Tech

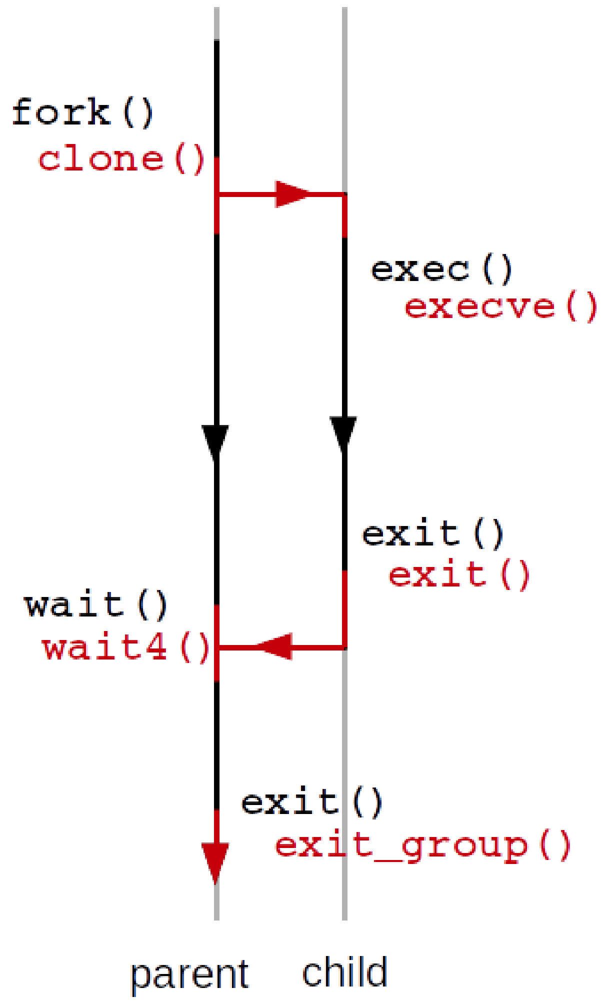
Outline1

Outline1

Process

Definition

- Refers to a **program currently executing in the system**
  - CPU registers
  - Location and state of memory segments (text, data, stack, etc.)
  - Kernel resources (open files, pending signals, etc.)
  - Threads
- Managed on a per-program way:
  - *Virtualization* of the processor and the memory
- Let's check an example with `strace` (`-f`)



Process

Sample program

```c
/* process.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
  pid_t pid = -42;
  int wstatus = -42;
  int ret = -1;

  pid = fork();
  switch(pid)
  {
    case -1:
      perror("fork");
      return EXIT_FAILURE;

    case 0:
      sleep(1);
      printf("Noooooooo!¥n");
      exit(0);
    default:
      printf("Iamyourfather!¥n");
      break;
  }

  ret = waitpid(pid, &wstatus, 0);
  if(ret == -1)
  {
    perror("waitpid");
    return EXIT_FAILURE;
  }
  printf("Childexitstatus:%d¥n",
      WEXITSTATUS(wstatus));

  return EXIT_SUCCESS;
}
```

```
gcc -Wall -Werror process.c -o process
./process
strace -f ./process > /dev/null
```

Process

`fork()` & `exec()` usage

- Tutorial on `fork()` usage:
  - http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
- Combining `fork()` and `exec()`:
  - https://ece.uwaterloo.ca/dwharder/icsrts/Tutorials/fork_exec/

Outline1

The process descriptor: `task_struct`

Presentation

- List of processes implemented as a linked list of `task_struct`

```c
struct tastk_struct {
  volatilelong state;
  void *stack;
  /* ... */
  int prio;
  /* ... */
  cpumask_t cpus_allowed;
  /* ... */
  struct list_head tasks;
  /* ... */
  struct mm_struct *mm;
  /* ... */
  pid_t pid;
  /* ... */
  struct task_struct *parent;
  struct list_head children;
  struct list_head sibling;
  /* ... */
}
```

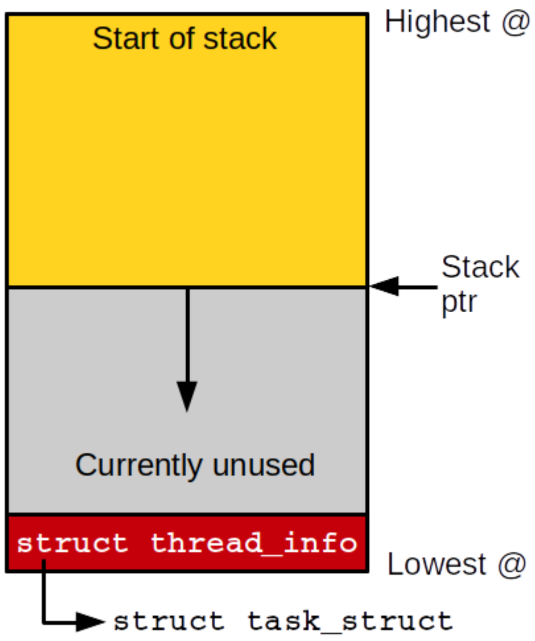
- Total size (Linux 4.8): 6976 bytes
- Full structure definition in `linux/sched.h`

The process descriptor: task_struct

Allocation & storage

- Prior to 2.6: task_struct allocated at the end of the kernel stack of each process
  - Allows to retrieve it without storing its location in a register
- Now dynamically allocated (heap) through the *slab allocator*
  - A struct thread_info living at the bottom of the stack

```
struct thread_info {
        struct task_struct        *task;
        u32                       flags;
        u32                       status;
        u32                       cpu;
};
```



- Moved off the stack in 4.9 [2] because of potential exploit [1] when overflowing the kernel stack

The process descriptor: `task_struct`

Allocation & storage (2)

- **Process Identifier (PID):** `pid_t` (`int`)
  - Max: 32768, can be increased to 4 millions
  - Wraps around when maximum reached
- Quick access to `task_struct` of the task currently running on a core: `current`
  - `arch/x86/include/asm/current.h`:

```c

DECLARE_PER_CPU(struct task_struct *, current_task);

static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}

#define current get_current()
```

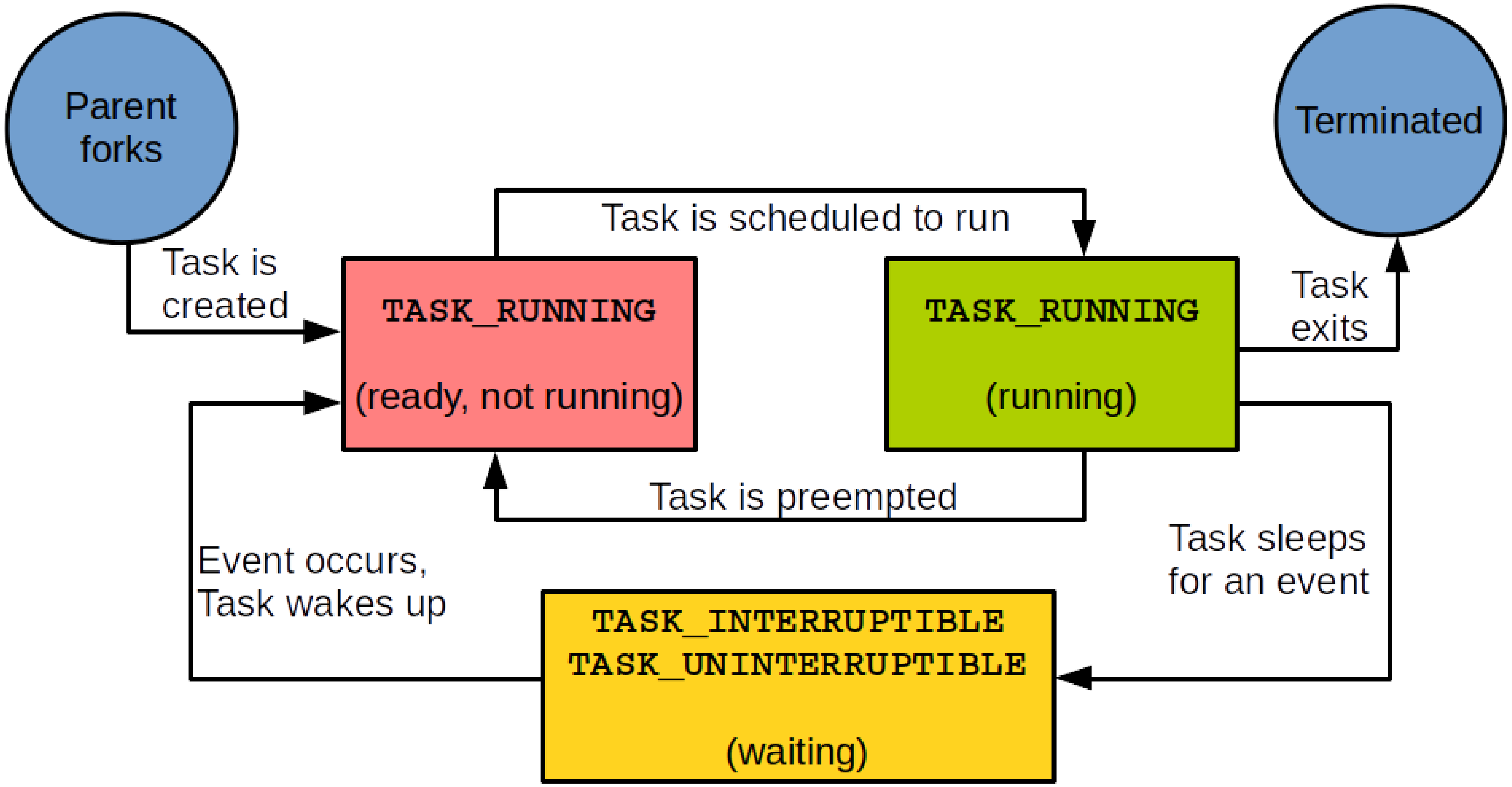
The process descriptor: `task_struct`

Process states

- **`state`** field of the `task_struct`
  - **`TASK_RUNNING`:**
    - Process is runnable (running or in a CPU run queue)
    - In user or kernel space
  - **`TASK_INTERRUPTIBLE`:**
    - Process is sleeping waiting for some condition
    - Switched to `TASK_RUNNING` on condition true or signal received
  - **`TASK_UNINTERRUPTIBLE`:**
    - Same as `TASK_INTERRUPTIBLE` but does not wake up on signal
  - `__TASK_TRACED`: Traced by another process (ex: debugger)
  - `__TASK_STOPPED`: Not running nor waiting, result of the reception of some signals to pause the process

The process descriptor: `task_struct`

Process states: flowchart



The process descriptor: `task_struct`

Process context and family tree

- The kernel can executes in **process** vs **interrupt** context
  - `current` is meaningful only when the kernel executes in *process context*
    - I.e. following a system call or an exception
- **Process hierarchy**
  - Root: `init`, PID 1
    - Launched by the kernel as the last step of the boot process
  - `fork`-based process creation:
    - Each process has a parent: `parent` pointer in the `task_struct`
    - Processes may have children: `children` field (`list_head`)
    - Processes may have siblings: `siblings` field
    - List of all tasks: `tasks` field

      - Easy manipulation through `next_task(t)` and `for_eachprocess(t)`
  - Let's check it out with the `pstree` command

Outline1

Process creation

Presentation, Copy-On-Write

- Linux does not implements creating a tasks from nothing (*spawn*)
- **`fork()` & `exec()`**
  - `fork()` creates a child, copy of the parent process
    - Only PID, PPID and some resources/stats differ
  - `exec()` loads into a process address space a new executable
- On `fork()`, Linux duplicates the parent page tables and creates a new process descriptor
  - It's *fast*, as **the address space is not copied**
    - Page table access bits: read-only
    - **Copy-On-Write** (COW): memory pages are copied only when they are referenced for write operations

Process creation

Forking: `fork()` and `vfork()`

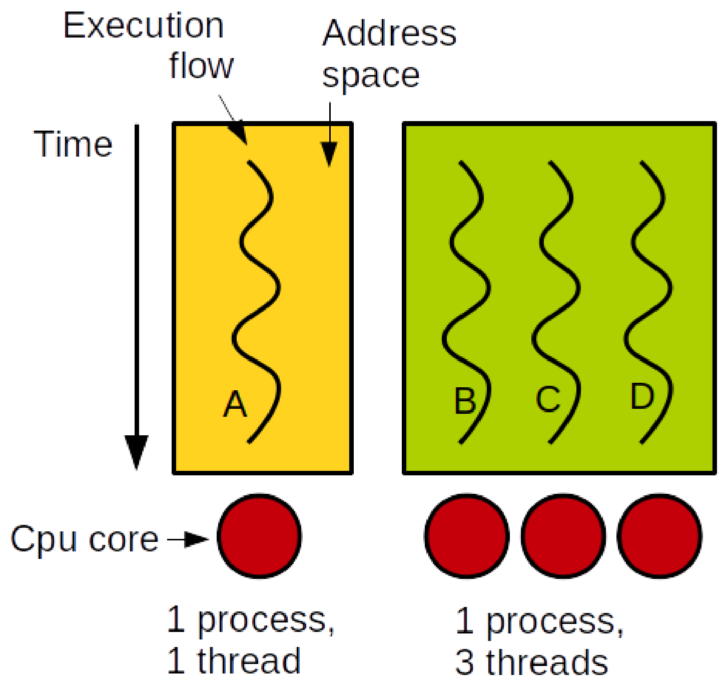
- `fork()` is implemented by the **`clone()`** system call
  1. `sys_clone()` calls `do_fork()`, which calls **`copy_process()`** and starts the new task
  2. **`copy_process()`**:
     1. Calls `dup_task_struct()`
        - Duplicates kernel stack, `task_struct` and `thread_info`
     2. Checks that we do not overflow the processes number limit
     3. Small amount of values are modified in the `task_struct`
     4. Calls `sched_fork()` to set the child `state` set to `TASK_NEW`
     5. Copies parent info: files, signal handlers, etc.
     6. Gets a new PID through `alloc_pid()`
     7. Returns a pointer to the created child `task_struct`
  3. Finally, `do_fork()` calls `wake_up_new_task()`
     - State becomes `TASK_RUNNING`
- `vfork()`: alternative without copy of the address space

Outline1

Threads

Presentation

- Theory:



- **Threads** are concurrent flows of execution belonging to the same program **sharing the same address space**
- In Linux there is no concept of a thread
  - No scheduling particularity
  - A thread is just another process sharing some information with other processes
  - Each thread has its own `task_struct`
  - Created through `clone()` with specific flags indicating sharing

Threads

Kernel threads

- To perform background operations in the kernel: **kernel threads**
- Very similar to user space threads
    - They are *schedulable entities* (like regular processes)
- However **they do not have their own address space**
    - `mm` in `task_struct` is `NULL`
- Used for several tasks:
    - Work queues (`kworker`)
    - Load balancing between CPU scheduling runqueues (`migration`)
    - etc.
    - List of all them with `ps --ppid 2`

Threads

Kernel threads: creation

- Kernel threads are all forked from the `kthread` kernel thread (PID 2), using `clone()`
  - To create a kernel thread, use `kthread_create()`
  - `include/linux/kthread.h`:

```
#define kthread_create(threadfn, data, namefmt, arg...) ¥
  kthread_create_on_node(threadfn, data, NUMA_NO_NODE, namefmt, ##arg)
```

```
struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
                void *data,
                int node,
                constchar namefmt[], ...);
```

- When created through `kthread_create()`, the thread is not in a runnable state
  - Need to call `wake_upprocess()`:

```
int wake_up_process(struct task_struct *p);
```

  - Or use `kthread_run()`

Threads

Kernel threads: creation (2)

- kthread_run():

```
#define kthread_run(threadfn, data, namefmt, ...)                     ¥
({                                                                    ¥
   struct task_struct * k                                             ¥
     = kthread_create(threadfn, data, namefmt, ##_VA_ARGS_);          ¥
   if (!IS_ERR( k))                                                   ¥
     wake_up_process( k);                                             ¥
   k;                                                                 ¥
})
```

- Thread termination:
  - Thread runs until it calls do_exit():

```
void do_exit(long error_code) noreturn;
```

  - Or until another part of the kernel calls kthread_stop():

```
int kthread_stop(struct task_struct *k);
```

Outline1

Process termination

Termination steps: `do_exit()`

- Termination on invoking the `exit()` system call
  - Can be implicitly inserted by the compiler on `return` from `main`
  - `sys_exit()` calls `do_exit()`
- `do_exit()` (`kernel/exit.c`):
  1. Calls `exit_signals()` which set the `PF_EXITING` flag in the `task_struct`
  2. Set the exit code in the `exit_code` field of the `task_struct`
     - To be retrieved by the parent
  3. Calls `exit_mm()` to release the `mm_struct` for the task
     - If it is not shared with any other process, it is destroyed
  4. Calls `exit_sem()`: process dequeued from potential semaphores queues
  5. Calls `exit_fs()` and `exit_files()` to update accounting information
     - Potential data structures that are not used anymore are freed

Process termination

Termination steps: `do_exit()` (2)

- `do_exit()` (continued):
  6. Calls `exit_notify()`
     - Sends signals to parent
     - Reparent potential children
     - Set the `exit_state` of the `task_struct` to `EXIT_ZOMBIE`
  7. Calls `do_task_dead()`
     - Sets the `state` to `TASK_DEAD`
     - Calls `schedule()` and never returns
- At that point, what is left is the `task_struct`, `thread_info` and kernel stack
  - To provide information to the parent
  - Parent notifies the kernel when everything can be freed

Process termination

`task_struct` cleanup

- Separated from the process of exiting because of the need to pass exit information to the parent
  - `task_struct` must survive a little bit before being deallocated
    - Until the parent grab the exit information through `wait4()`
- Cleanup implemented in `release_task()` called from the `wait4()` implementation
  - Remove the task from the task list
  - Release and free remaining resources

Process termination

Parentless tasks

- **A parent exits before its child**
  - Child must be *reparented*
    - To another process in the current thread group ...
    - ... or `init` if that fails
- `exit_notify()` calls `forget_original_parent()`, that calls `find_new_reaper()`
  - Returns the `task_struct` of another task in the thread group if it exists, otherwise the one from `init`
  - Then, all the children of the currently dying task are reparented to the reaper

BibliographyI

1 Exploiting stack overflow in the linux kernel.
https://jon.oberheide.org/blog/2010/11/29/exploiting-stack-overflows-in-the-linux-kernel/.
Accessed: 2017-01-23.

2 Security things in linux v4.9.
https://outflux.net/blog/archives/2016/12/12/security-things-in-linux-v4-9/.
Accessed: 2017-01-23.

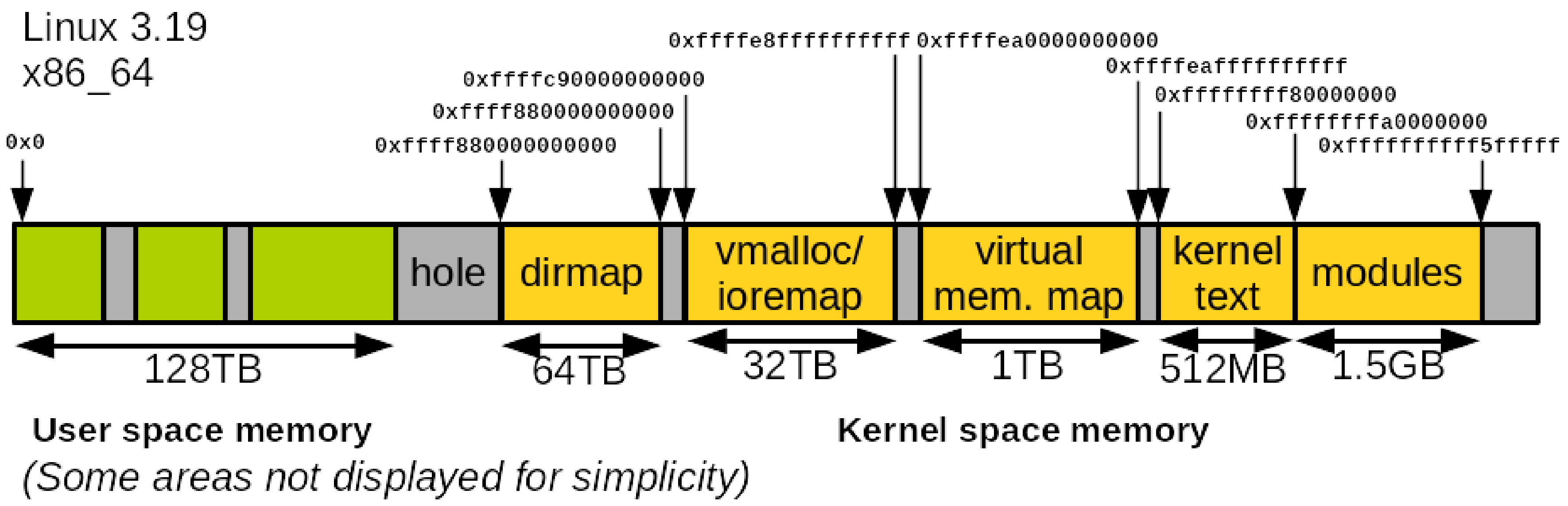
Outline2

Outline2

Address space and memory descriptor

Address space

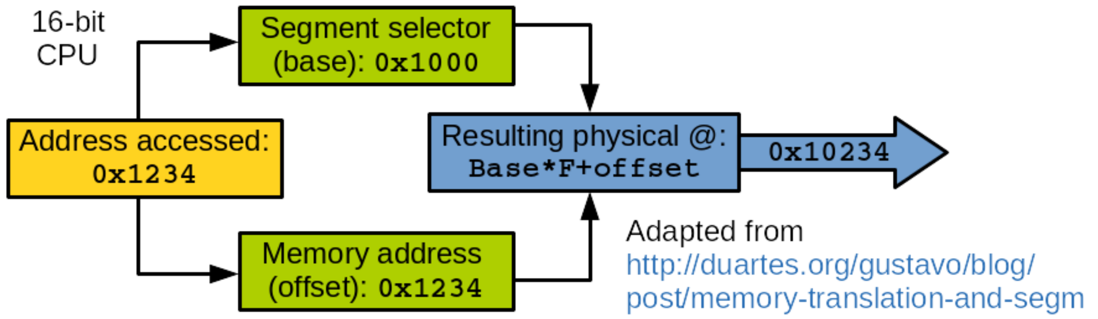
- **The memory that a process can access is called its address space**
  - Illusion that the process can access 100% of the system memory
  - With virtual memory, can be much larger than the actual amount of physical memory
- Defined by the process page table set up by the kernel



Address space and memory descriptor

Address space (2)

- Each process is given a *flat* 32/64-bits address space
  - *Flat* as opposed to segmented



16-bit CPU
Segment selector (base): 0x1000
Address accessed: 0x1234
Resulting physical @: Base*F+offset
0x10234
Memory address (offset): 0x1234

Adapted from http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation/

- A **memory address** is an index within the address spaces:
  - Identify a specific byte
  - Example: `0x8fffa12dd24123fd`

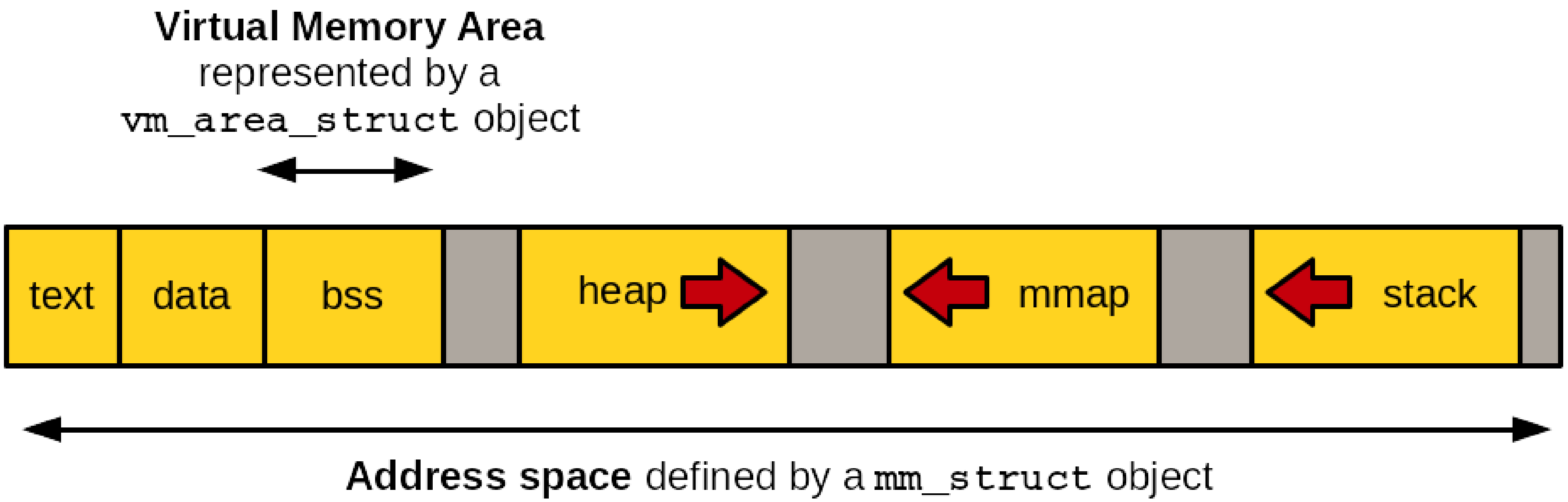
Address space and memory descriptor

Address space (3)

- Interval of addresses that the process has the right to access: **virtual memory areas (VMAs**)
  - VMAs can be dynamically added or removed to the process address space
  - VMAs have associated permissions: read, write, execute
    - When a process try to access an address outside of valid VMAs, or access a VMA with wrong permissions: **segmentation fault**
- **VMAs can contain**:
  - Mapping of the executable file code (*text section*)
  - Mapping of the executable file initialized variables (*data section*)
  - Mapping of the zero page for uninitialized variables (*bss section*)
  - Mapping of the zero page for the user-space stack
  - Text, data, bss for each shared library used
  - Memory-mapped files, shared memory segment, anonymous mappings (used by malloc)

Address space and memory descriptor

Address space (4)


Virtual Memory Area represented by a `vm_area_struct` object
text
data
bss
heap
mmap
stack
Address space defined by a `mm_struct` object

Address space and memory descriptor

Memory descriptor

- The kernel represent a process address space through a **`struct mm_struct`** object, the **memory descriptor**
  - Defined in `include/linux/mm_types.h`
  - Interesting fields:

```
struct mm_struct {
  struct vm_area_struct *mmap; /* listofVMAs       */

  struct rb_root         mm_rb;            /* rbtreeofVMAs     */
  pgd_t                  *pgd;             /* pageglobaldirectory */
  atomic_t               mm_users;         /* addressspaceusers */
  atomic_t               *mm_count;        /* primaryusagecounters */
  int                    map_count;        /* numberofVMAs     */
  struct rw_semaphore    mmap_sem;         /* VMAsemaphore  */
  spinlock_t             page_table_lock;  /* pagetablelock    */
  struct list_head       mmlist;           /* listofallmm_struct */
  unsignedlong           start_code;       /* startaddressofcode */
  unsignedlong           end_code;         /* endaddressofcode */
  unsignedlong           start_data;       /* startaddressofdata */
  unsignedlong           end_data;         /* endaddressofdata */
/* ... */
```

Address space and memory descriptor

Memory descriptor (2)

```
    /* ... */
    unsignedlong           start_brk;  /* startaddressofheap    */
    unsignedlong           end_brk;    /* endaddressofheap    */
    unsignedlong           start_stack; /* startaddressofstack    */
    unsignedlong           arg_start;   /* startofarguments    */
    unsignedlong           arg_end;     /* endofarguments    */
    unsignedlong           env_start;   /* startofenvironment    */
    unsignedlong           total_vm;    /* totalpagesmapped    */
    unsignedlong           locked_vm;   /* numberoflockedpages    */
    unsignedlong           flags;       /* architecturespecificdata    */
    spinlock_t             ioctx_lock;  /* AsynchronousI/Olistlock    */
    /* ... */
};
```

- `mm_users`: number of processes (threads) using the address space
- `mm_count`: reference count:
    - +1 if `mm_users` > 0
    - +1 if the kernel is using the address space
    - When `mm_count` reaches 0, the `mm_struct` can be freed

Address space and memory descriptor

Memory descriptor (3)

- `mmap` and `mm_rb` are respectively a linked list and a tree containing all the VMAs in this address space
  - List used to iterate over all the VMAs
    - Links all VMAs sorted by ascending virtual addresses
  - Tree used to find a specific VMA
- All `mm_struct` are linked together in a doubly linked list
  - Through the `mmlist` field if the `mm_struct`

Address space and memory descriptor

Memory descriptor allocation

- **A task memory descriptor is located in the `mm` field of the corresponding `task_struct`**
  - Current task memory descriptor: `current->mm`
  - During `fork()`, `copy_mm()` is making a copy of the parent memory descriptor for the child
    - `copy_mm()` calls `dup_mm()` which calls `allocate_mm()` → allocates a `mm_struct` object from a slab cache
  - Two threads sharing the same address space have the `mm` field of their `task_struct` pointing to the same `mm_struct` object
    - Threads are created using the `CLONE_VM` flag passed to `clone()` → `allocate_mm()` is not called
    - in `copy_mm()`:

```c
/* ... */
struct mm_struct *oldmm;
oldmm = current->mm;
/* ... */
if (clone_flags & CLONE_VM) {
  atomic_inc(&oldmm->mm_users);
  mm = oldmm;
```

```c
    goto good_mm;
}
/* ... */
good_mm:
    /* ... */
return 0;
```

Address space and memory descriptor

Memory descriptor destruction

- When a process exits, `do _exit()` is called
  - It calls `exit_mm()`
    - Performs some housekeeping/statistics updates
    - Calls `mmput()`

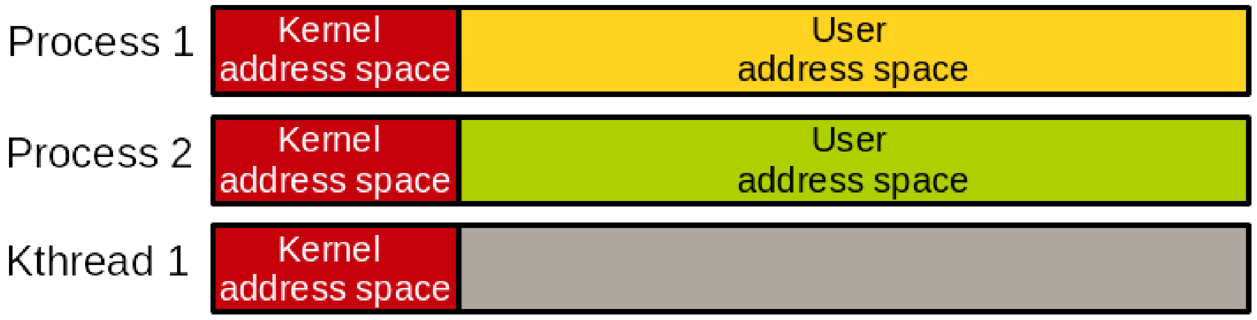
```
void mmput(struct mm_struct *mm)
{
  might_sleep();

  if (atomic_dec_and_test(&mm->mm_users))
    __mmput(mm);
}
```

- `mmput()` decrements the `users` field and calls_`mmput()` if it reaches 0
- _`mmput()` calls `mmdrop()`, that decrements the `count` field, and calls_`mm drop()` if it reaches 0
- _`mmdrop()` calls `free mm()` which return the memory for the `mm_ struct()` to the slab cache (i.e. free)

Address space and memory descriptor

Memory descriptor and kernel threads



- Kernel threads do not have a user-space address space
  - `mm` field of a kernel thread `task_struct` is `NULL`
- However they still need to access the kernel address space
  - When a kernel thread is scheduled, the kernel notice its `mm` is `NULL` so it keeps the previous address space loaded (page tables)
  - Kernel makes the `active_mm` field of the kernel thread to point on the *borrowed* `mm_struct`
  - OK because **kernel part is the same in all address spaces**

Outline2

Virtual Memory Area

vm_area_srtuct

- Each VMA is represented by an object of type **`vm area struct`**
  - Defined in `include/linux/mm_types.h`
  - Interesting fields:

```
struct vm_area_struct {
   struct i                    mm_struct *vm_mm; /* associatedaddressspace(mm_struct)    */
   unsignedlong                vm_start;         /* VMAstart,inclusive   */
   unsignedlong                vm_end;           /* VMAend,exclusive   */
   struct vm_area_struct       *vm_next;         /* listofVMAs   */
   struct vm_area_struct       *vm_prev;         /* listofVMAs   */
   pgprot_t                    vm_page_prot;     /* accesspermissions  */
   unsignedlong                vm_flags;         /* flags */
   struct rb_node              vm_rb;            /* VMAnodeinthetree     */
   struct list_head            anon_vma_chain;   /* listofanonymousmappings    */
   struct anon_vma             *anon_vma;        /* anonmousvmaobject   */
   struct vm_operation_struct *vm_ops;           /* operations */
   unsignedlong                vm_pgoff;         /* offsetwithinfile   */
   struct file                 *vm_file;         /* mappedfile(canbeNULL)     */
   void                        *vm_private_data; /* privatedata */
   /* ... */
}
```

Virtual Memory Area

`vm_area_srtuct` (2)

- The VMA exists over `[vm_start, vm_end[` in the corresponding address space
  - Address space is pointed by the `vm_mm` field (of type `mm_struct`)
- Size in bytes: `vm_end - vm_start`
- Each VMA is unique to the associated `mm_struct`
  - Two processes mapping the same file will have two different `mm_struct` objects, and two different `vm_area_struct` objects
  - Two threads sharing a `mm_struct` object also share the `vm_area_struct` objects

Virtual Memory Area

Flags

- **Flags** specify properties and information for all the pages contained in the VMA

- `VM_READ`: pages can be read from
- `VM_WRITE`: pages can be written to
- `VM_EXEC`: code inside pages can be executed
- `VM_SHARED`: pages are shared between multiple processes (if unset the mapping is *private*)
- `VM_MAYREAD`: the `VM_READ` flag can be set
- `VM_MAYWRITE`: the `VM_WRITE` flag can be set
- `VM_MAYEXEC`: the `VM_EXEC` flag can be set
- `VM_MAYSHARE`: the `VM_SHARED` flag can be set
- `VM_GROWSDOWN`: area can grow downwards
- `VM_GROWSUP`: area can grow upwards
- `VM_SHM`: area can be used for shared memory
- `VM_DENYWRITE`: area maps an unwritable file
- `VM_EXECUTABLE`: area maps an executable file

Virtual Memory Area

Flags (2)

- `VM_LOCKED`: the area pages are locked (will not be swapped-out)
- `VM_IO`: the area maps a device IO space
- `VM_SEQ_READ`: pages in the area seem to be accessed sequentially
- `VM_RAND_READ`: pages seem to be accessed randomly
- `VM_DONTCOPY`: area will not be copied upon `fork()`
- `VM_DONTEXPAND`: area cannot grow through `mremap()`
- `VM_ACCOUNT`: area is an accounted VM object
- `VM_HUGETLB`: area uses `hugetlb` pages

Virtual Memory Area

Flags (3)

- Combining `VM_READ`, `VM_WRITE` and `VM_EXEC` gives the permissions for the entire area, for example:
  - Object code is `VM_READ` and `VM_EXEC`
  - Stack is `VM_READ` and `VM_WRITE`
- `VM_SEQ_READ` and `VM_RAND_READ` are set through the `madvise()` system call
  - Instructs the file pre-fetching algorithm *read-ahead* to increase or decrease its agressivity
- `VM_HUGETLB` indicates that the area uses pages larger than the regular size
  - 2M and 1G on x86
  - Smaller page table → good for the TLB
  - Less levels of page tables → faster address translation

Virtual Memory Area

VMA operations

- `vm_ops` in `vm_area_struct` points to a **`vm_operations_struct`** object
  - Contains function pointers to operate on a specific VMAs
  - Defined in `include/linux/mm.h`

```
struct vm_operations_struct {
  void (*open)(struct vm_area_struct * area);
  void (*close)(struct vm_area_struct * area);
  int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
  int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
  /* ... */
}
```

Virtual Memory Area

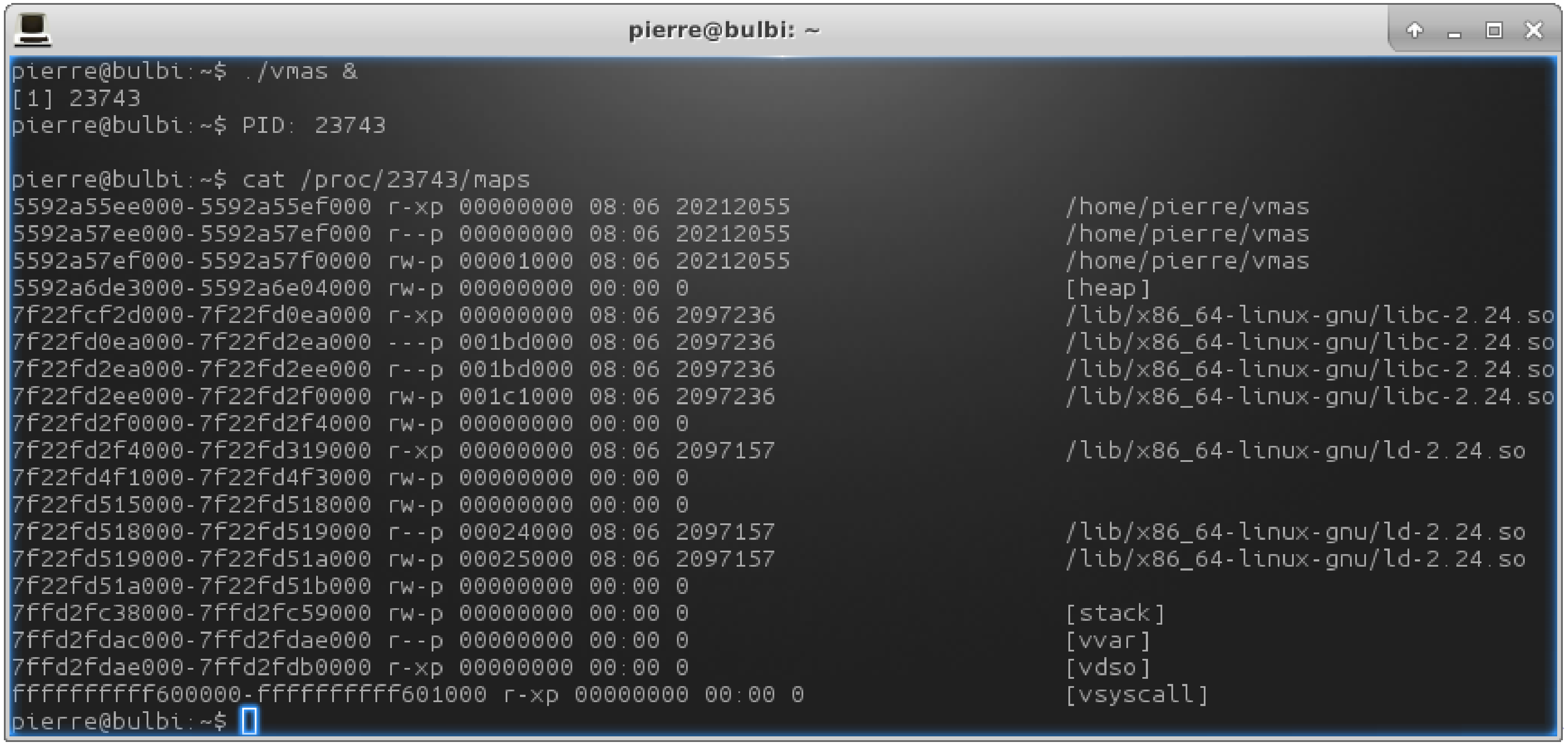
VMA operations (2)

- Function pointers in `vm_operations_struct`:
  - `open()`: called when the area is added to an address space
  - `close()`: called when the area is removed from an address space
  - `fault()`: invoked by the page fault handler when a page that is not present in physical memory is accessed
  - `page_mkwrite()`: invoked by the page fault handler when a previously read-only page is made writable
  - Description of all operations in `include/linux/mm.h`

Virtual Memory Area

VMAs in real life

- From userspace, one can observe the VMAs map for a given process:
  - `cat /proc/<pid>/maps`



pierre@bulbi: ~

```
pierre@bulbi:~$ ./vmas &
[1] 23743
pierre@bulbi:~$ PID: 23743

pierre@bulbi:~$ cat /proc/23743/maps
5592a55ee000-5592a55ef000 r-xp 00000000 08:06 20212055                   /home/pierre/vmas
5592a57ee000-5592a57ef000 r--p 00000000 08:06 20212055                   /home/pierre/vmas
5592a57ef000-5592a57f0000 rw-p 00001000 08:06 20212055                   /home/pierre/vmas
5592a6de3000-5592a6e04000 rw-p 00000000 00:00 0                          [heap]
7f22fcf2d000-7f22fd0ea000 r-xp 00000000 08:06 2097236                    /lib/x86_64-linux-gnu/libc-2.24.so
7f22fd0ea000-7f22fd2ea000 ---p 001bd000 08:06 2097236                    /lib/x86_64-linux-gnu/libc-2.24.so
7f22fd2ea000-7f22fd2ee000 r--p 001bd000 08:06 2097236                    /lib/x86_64-linux-gnu/libc-2.24.so
7f22fd2ee000-7f22fd2f0000 rw-p 001c1000 08:06 2097236                    /lib/x86_64-linux-gnu/libc-2.24.so
7f22fd2f0000-7f22fd2f4000 rw-p 00000000 00:00 0
7f22fd2f4000-7f22fd319000 r-xp 00000000 08:06 2097157                    /lib/x86_64-linux-gnu/ld-2.24.so
7f22fd4f1000-7f22fd4f3000 rw-p 00000000 00:00 0
7f22fd515000-7f22fd518000 rw-p 00000000 00:00 0
7f22fd518000-7f22fd519000 r--p 00024000 08:06 2097157                    /lib/x86_64-linux-gnu/ld-2.24.so
7f22fd519000-7f22fd51a000 rw-p 00025000 08:06 2097157                    /lib/x86_64-linux-gnu/ld-2.24.so
7f22fd51a000-7f22fd51b000 rw-p 00000000 00:00 0
7ffd2fc38000-7ffd2fc59000 rw-p 00000000 00:00 0                          [stack]
7ffd2fdac000-7ffd2fdae000 r--p 00000000 00:00 0                          [vvar]
7ffd2fdae000-7ffd2fdb0000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
pierre@bulbi:~$ 
```

Virtual Memory Area

VMAs in real life (2)

- `/proc/<pid>/maps` columns description:
    1. Address range
    2. Permissions
    3. Start offset of file mapping
    4. Device containing the mapped file
    5. Mapped file inode number
    6. Mapped file pathname
- Can also use the command `pmap <pid>`

Outline2

VMA manipulation

Finding a VMA

- **`find_vma()`**: used to find a VMA in which a specific memory address resides
  - Prototype in `include/linux/mm.h`:

```
struct vm_area_struct *find_vma(struct mm_struct *mm, unsignedlong addr);
```

  - Defined in `mm/mmap.c`:

```
struct vm_area_struct *find_vma(struct
      mm_struct *mm, unsignedlong addr)
{
   struct rb_node *rb_node;
   struct vm_area_struct *vma;

   /* Checkthecachefirst.    */
   vma = vmacache_find(mm, addr);
   if (likely(vma))
     return vma;

   rb_node = mm->mm_rb.rb_node;

   while (rb_node) {
     struct vm_area_struct *tmp;
     tmp = rb_entry(rb_node, struct
       vm_area_struct, vm_rb);

     if (tmp->vm_end > addr) {
       vma = tmp;
       if (tmp->vm_start <= addr)
         break;
       rb_node = rb_node->rb_left;
     } else
       rb_node = rb_node->rb_right;
   }

   if (vma)
     vmacache_update(addr, vma);
   return vma;
}
```

VMA manipulation

Finding a VMA (2)

- `find_vma_prev()`: returns in addition the last VMA before a given address
  - `include/linux/mm.h`:

```c
struct vm_area_struct *find_vma_prev(struct mm_struct *mm, unsignedlong addr,
        struct vm_area_struct **pprev);
```

- `find_vma_intersection()`: returns the first VMA overlapping a given address range
  - `include/linux/mm.h`:

```c
static inline struct vm_area_struct * find_vma_intersection(struct mm_struct *
      mm, unsignedlong start_addr, unsignedlong end_addr)
{
  struct vm_area_struct * vma = find_vma(mm,start_addr);

  if (vma && end_addr <= vma->vm_start)
    vma = NULL;
  return vma;
}
```

VMA manipulation

Creating an address interval

- **`do_mmap()`** is used to create a new *linear address interval*:
  - Can result in the creation of a new VMAs
  - Or a merge of the create area with an adjacent one when they have the same permissions
  - `include/linux/mm.h`:

```
externunsignedlong    do_mmap(struct file *file, unsignedlong addr,
   unsignedlong len, unsignedlong prot, unsignedlong flags,
   vm_flags_t vm_flags, unsignedlong pgoff, unsignedlong  *populate);
```

- Caller must hold `mm->mmap_sem` (RW semaphore)
- Maps the file `file` in the address space at address `addr` for length `len`. Mapping starts at offset `pgoff` in the file
- `prot` specifies access permissions for the memory pages:
  - `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`

VMA manipulation

Creating an address interval (2)

- `flags` specifies the rest of the VMA options:

- `MAP _SHARED`: mapping can be shared
- `MAP _PRIVATE`: mapping cannot be shared
- `MAP _FIXED`: created interval *must* start at `addr`
- `MAP _ANONYMOUS`: mapping is not file-backed
- `MAP _GROWSDOWN`: corresponds to `VM_GROWSDOWN`
- `MAP _DENYWRITE`: corresponds to `VM DENYWRITE`
- `MAP _EXECUTABLE`: corresponds to `VM_EXECUTABLE`
- `MAP _LOCKED`: corresponds to `VM_LOCKED`
- `MAP _NORESERVE`: no space reserved for the mapping
- `MAP _POPULATE`: populate (default) page tables
- `MAP _NONBLOCK`: do not block on IO

VMA manipulation

Creating an address interval (3)

- On error `do_mmap()` returns a negative value
- On success:
  - The kernel tries to merge the new interval with an adjacent one having same permissions
  - Otherwise, create a new VMA
  - Returns a pointer to the start of the mapped memory area
- `do_mmap()` is exported to user-space through `mmap2()`

```c
void *mmap2(void *addr, size_t length, int prot, int flags, int fd, off_t pgoffset);
```

VMA manipulation

Removing an address interval

- Removing an address interval is done through `do_munmap()`
  - `include/linux/mm.h`:

```c
int do_munmap(struct mm_struct *, unsignedlong , size_t);
```

  - 0 returned on success
- Exported to user-space through `munmap()`:

```c
int munmap(void *addr, size_t len);
```

Outline2

Page tables

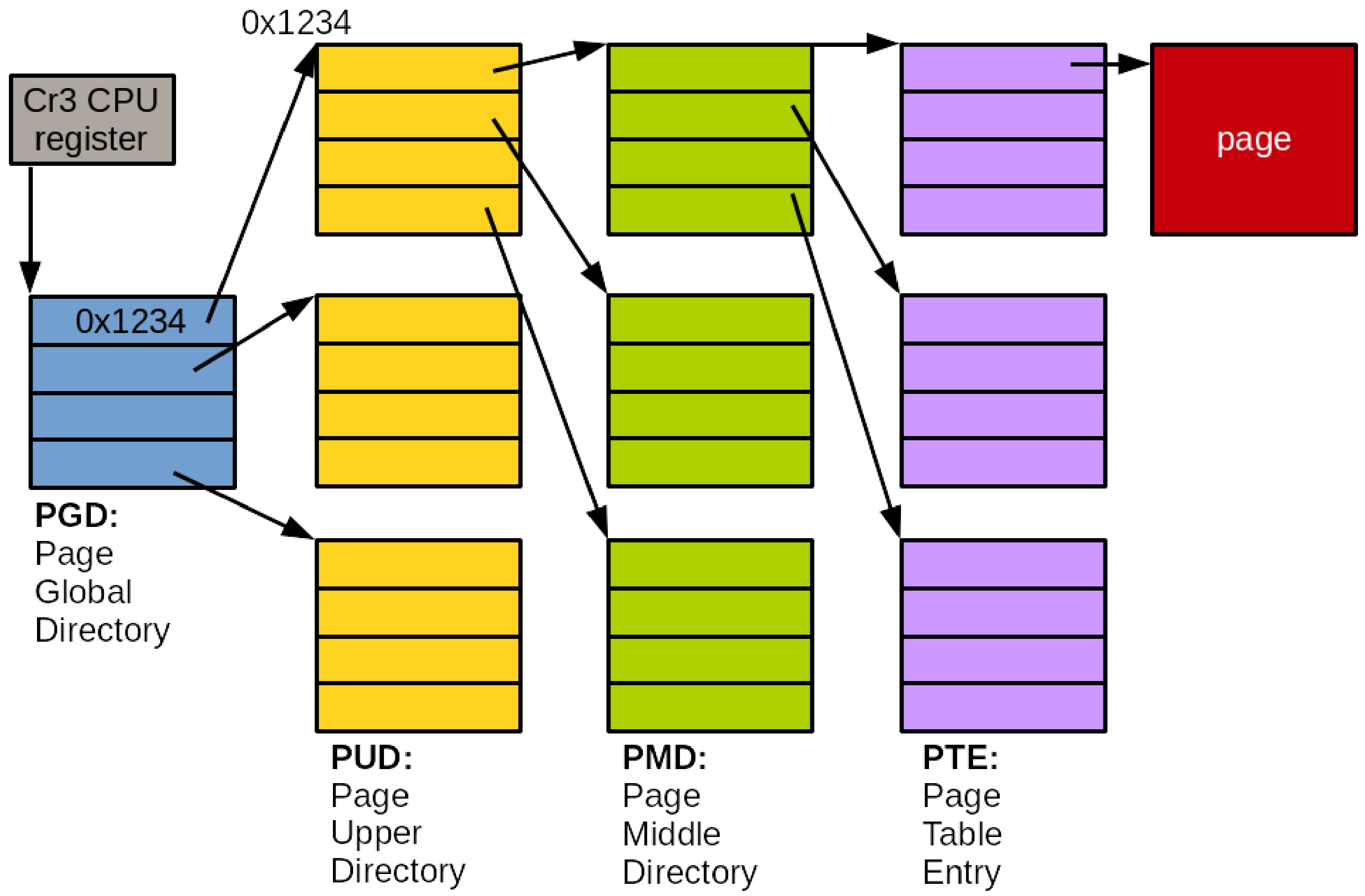
Presentation

- Linux enables **paging** early in the boot process
  - **All memory accesses made by the CPU are virtual and translated to physical addresses through the *page tables***
  - Linux set the page tables and the translation is made automatically by the hardware (MMU) according to the page tables content
- The address space is defined by VMAs and is sparsely populated
  - One address space per process → one page table per process
  - Lots of ”empty” areas
  - **Defining the page table as a single static array is a huge waste of space**
    - A hierarchical tree structure is used

Page tables

Page table setup (2)

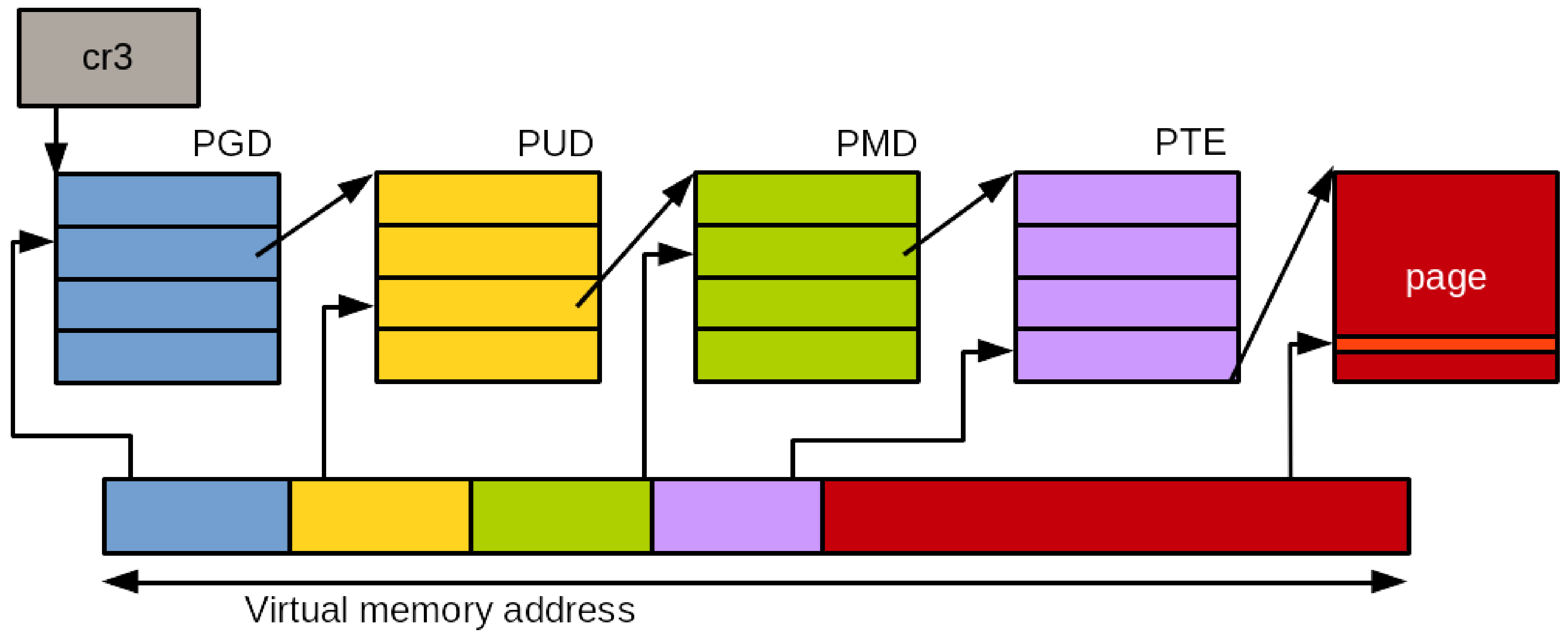
- **Setting up the page table is performed by the kernel**


0x1234
Cr3 CPU register
0x1234
page
PGD: Page Global Directory
PUD: Page Upper Directory
PMD: Page Middle Directory
PTE: Page Table Entry

Page tables

Address translation

- **Address translation is performed by the hardware**



- More info on page tables and memory management: [2, 1]

BibliographyII

[1]Four-level page tables.
https://lwn.net/Articles/106177/.
Accessed: 2017-03-25.

[2]Gorman, M.
Understanding the Linux virtual memory manager.
Prentice Hall Upper Saddle River, 2004.
Accessed: 2017-03-25.