Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[10] Process Management

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v2

Outline

1. **Process Control Block**
    1. Accessing the PCB
2. **The fork()/exec() model**
    1. Kernel Threads
3. **Out Of Memory (OOM) Killer**
4. **Process Starting**
    1. The ELF Format
    2. Dynamic Linking
    3. Initial Steps of Programs' Life

10.1

10. Process Management

Process Control Block

DIAG

Processes

The concept of a process is fundamental to any multiprogramming OS. The term process is often used with several different meanings, for us it means an **instance of a program in execution** (or even a set of data structures which describes how far the execution has progressed).

When a process is created is almost **identical** to its parent:

- it receives a logical copy of the parent's address space
- it executes the same code of the parent, at next instruction after the fork()

But the child process has separate copies of the data (stack and heap) so that changes in the child are invisible to the parent and vice versa.

Processes

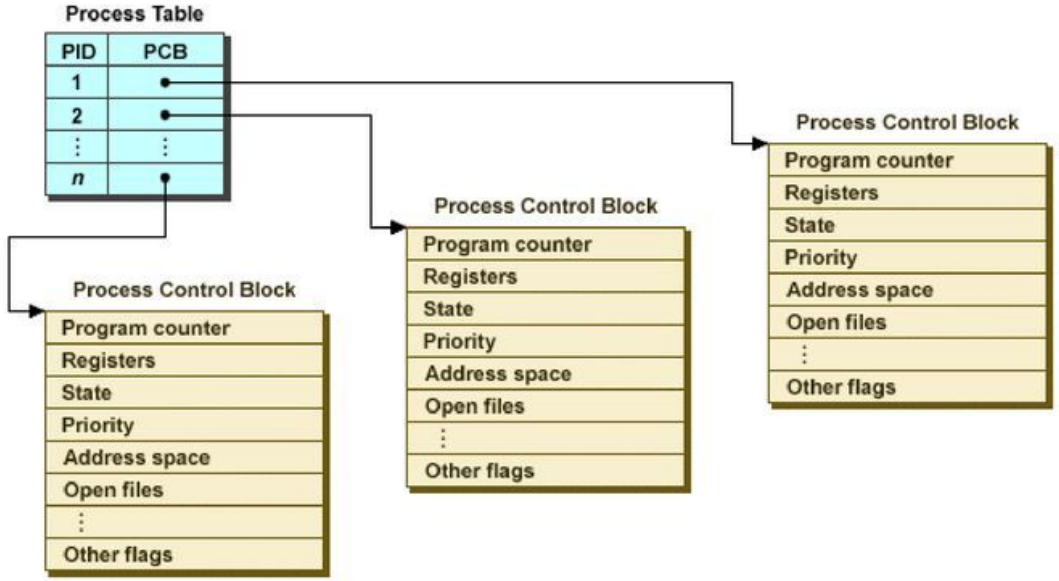
While earlier versions of Unix supported this model, modern ones do not. They instead supported multithreaded applications, in which a process is composed of several ***user threads*** (or simply threads), each of which represents a an execution flow of the process (pthread library).

Older versions of Linux did not support multithreaded applications, so from the kernel point of view a multithreaded application was just a normal process. So threads were created, handled and scheduled in User Mode. Therefore if a thread was blocked for a system call, every other thread would be blocked.

Nowadays Linux uses **lightweight processes** that are independent from each other but at the same time they can share resources (e.g. memory). They are mapped, in the end, to threads. A process in modern versions of Linux is just a group of lightweight processes, also called a **thread group** (created with pthread library).

The Process Control Block

To manage processes, the kernel must have a clear picture of what each process is doing, for instance, the priority, the state, the address space and so on. This is the role of the **process descriptor** (also called Process Control Block - PCB).



task_struct

The **struct** task_struct object represents the Process Control Block within the Linux Kernel. This is declared in `include/linux/sched.h` and it is one of the largest structures in the kernel (almost 600 LOCs). Relevant members are:

- **volatile long** state
- **struct** mm_struct *mm
- **struct** mm_struct *active_mm
- pid_t pid
- pid_t tgid
- **struct** fs_struct *fs
- **struct** files_struct *files
- **struct** signal_struct *sig
- **struct** thread_struct thread /* CPU-specific state: TSS, FPU, CR2, perf events, ... */
- **int** prio; /* to implement nice() */
- **unsigned long** policy /* for scheduling */
- **int** nr_cpus_allowed
- cpumask_t cpus_allowed

task_struct

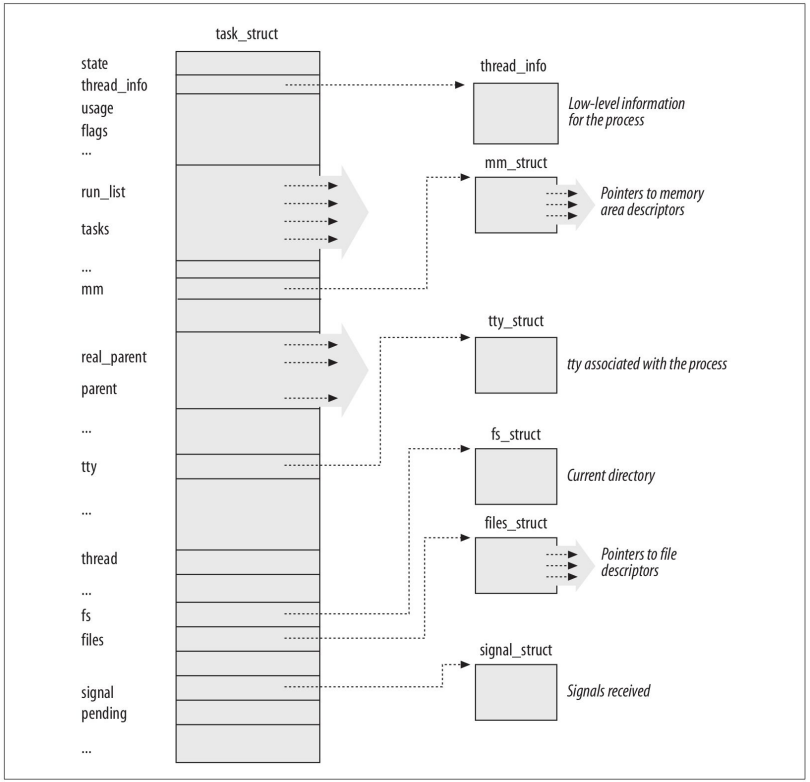


Figure 3-1. The Linux process descriptor

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Process State

The state field of a process descriptor describe what is currently happening to the process. Possible states are always exclusive:

- `TASK_RUNNING` the process is either executing on CPU or waiting to be executed
- `TASK_INTERRUPTIBLE` the process is sleeping until some condition becomes true
- `TASK_UNINTERRUPTIBLE` like TASK_INTERRUPTIBLE but except that raising a signal to the process will leave the state unchanged
- `TASK_STOPPED` process has stopped (after signal SIGSTOP, SIGTSTP)
- `TASK_TRACED` process execution has stopped by a debugger

Then to the `exit_state` field we can have

- `EXIT_ZOMBIE` process terminated but the parent did not issued wait to retrieve the data, so the kernel cannot discard it
- `EXIT_DEAD` the parent issued wait

The mm member

The field `mm` points to a `mm_struct` defined in `include/linux/mm_types.h`. The `mm_struct` is used to manage the memory map of the process:

- virtual address of the page table (**pgd** member)
- a pointer to a list of `vm_area_struct` records (mmap field)

Each record tracks a user-level virtual memory area which is valid for the process. `active_mm` is used to "steal" a `mm` when running in an anonymous process, and `mm` is set to NULL. Non-anonymous processes have `active_mm == mm`.

```
386  struct mm_struct {
387          struct {
388                  struct vm_area_struct *mmap;            /* list of VMAs */
389                  struct rb_root mm_rb;
390                  u64 vmacache_seqnum;                   /* per-thread vmacache */
391  #ifdef CONFIG_MMU
392                  unsigned long (*get_unmapped_area) (struct file *filp,
393                                  unsigned long addr, unsigned long len,
394                                  unsigned long pgoff, unsigned long flags);
395  #endif
396                  unsigned long mmap_base;        /* base of mmap area */
397                  unsigned long mmap_legacy_base; /* base of mmap area in bottom-up allocations */
398  #ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES
399                  /* Base adresses for compatible mmap() */
400                  unsigned long mmap_compat_base;
401                  unsigned long mmap_compat_legacy_base;
402  #endif
403                  unsigned long task_size;        /* size of task vm space */
404                  unsigned long highest_vm_end;   /* highest vma end address */
405                  pgd_t * pgd;
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/mm_types.h#L386 - https://www.kernel.org/doc/html/latest/vm/active_mm.html

vm_area_struct

The struct vm_area_struct describes a Virtual Memory Area (VMA). It contains:

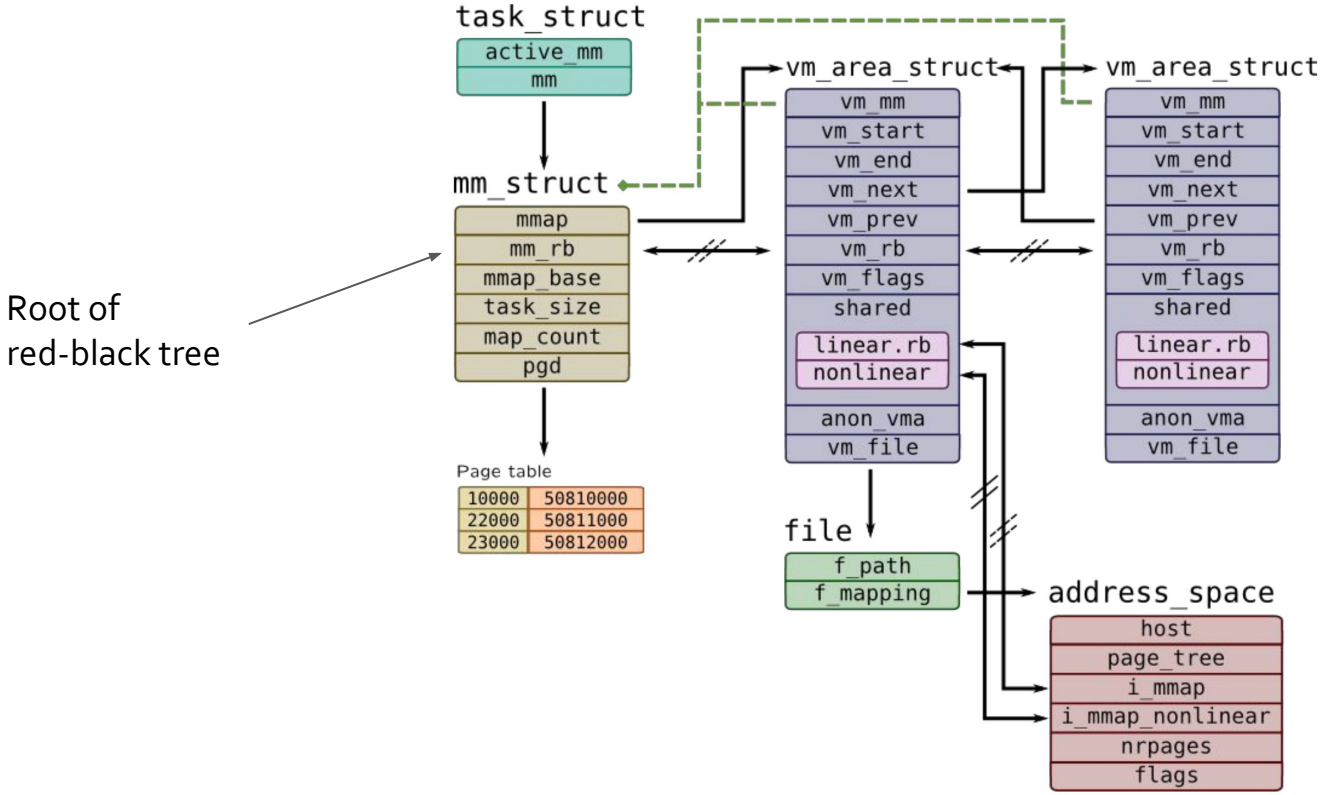
- `struct mm_struct *vm_mm`: the address space the structure belongs to
- `unsigned long vm_start`: the start address in `vm_mm`
- `unsigned long vm_end`: the end address
- `pgprot_t vm_page_prot`: access permissions of this VMA
- `const struct vm_operations_struct *vm_ops`: operations to deal with this structure
- `struct mempolicy *vm_policy`: the NUMA policy for this range of addresses
- `struct file *vm_file`: pointer to a memory-mapped file
- `struct vm_area_struct *vm_next, *vm_prev`: linked list of VM areas per task, sorted by address

vm_operations_struct

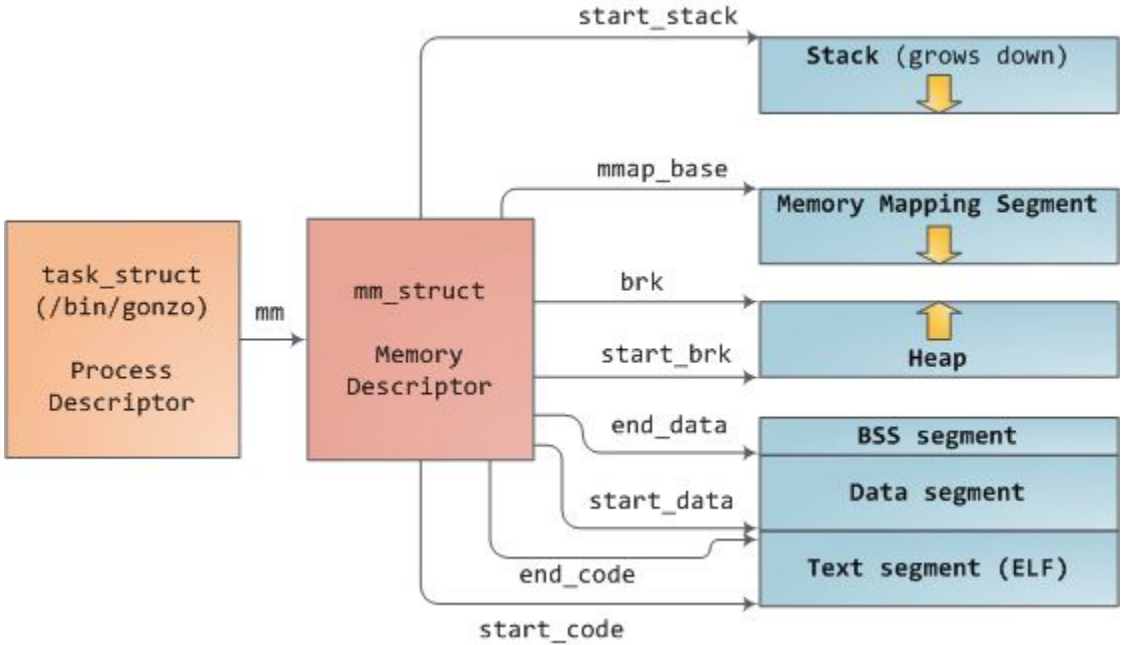
```
560  /*
561   * These are the virtual MM functions - opening of an area, closing and
562   * unmapping it (needed to keep files on disk up-to-date etc), pointer
563   * to the functions called when a no-page or a wp-page exception occurs.
564   */
565  struct vm_operations_struct {
566          void (*open)(struct vm_area_struct * area);
567          void (*close)(struct vm_area_struct * area);
568          /* Called any time before splitting to check if it's allowed */
569          int (*may_split)(struct vm_area_struct *area, unsigned long addr);
570          int (*mremap)(struct vm_area_struct *area, unsigned long flags);
571          /*
572           * Called by mprotect() to make driver-specific permission
573           * checks before mprotect() is finalised.   The VMA must not
574           * be modified.  Returns 0 if eprotect() can proceed.
575           */
576          int (*mprotect)(struct vm_area_struct *vma, unsigned long start,
577                          unsigned long end, unsigned long newflags);
578          vm_fault_t (*fault)(struct vm_fault *vmf);
579          vm_fault_t (*huge_fault)(struct vm_fault *vmf,
580                          enum page_entry_size pe_size);
581          void (*map_pages)(struct vm_fault *vmf,
582                          pgoff_t start_pgoff, pgoff_t end_pgoff);
583          unsigned long (*pagesize)(struct vm_area_struct * area);
584
585          /* notification that a previously read-only page is about to become
586           * writable, if an error is returned it will cause a SIGBUS */
587          vm_fault_t (*page_mkwrite)(struct vm_fault *vmf);
588
```

```
595          int (*access)(struct vm_area_struct *vma, unsigned long addr,
596                        void *buf, int len, int write);
597
598          /* Called by the /proc/PID/maps code to ask the vma whether it
599           * has a special name.  Returning non-NULL will also cause this
600           * vma to be dumped unconditionally. */
601          const char *(*name)(struct vm_area_struct *vma);
602
603  #ifdef CONFIG_NUMA
604          /*
605           * set_policy() op must add a reference to any non-NULL @new mempolicy
606           * to hold the policy upon return.  Caller should pass NULL @new to
607           * remove a policy and fall back to surrounding context--i.e. do not
608           * install a MPOL_DEFAULT policy, nor the task or system default
609           * mempolicy.
610           */
611          int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);
612
613          /*
614           * get_policy() op must add reference [mpol_get()] to any policy at
615           * (vma,addr) marked as MPOL_SHARED.  The shared policy infrastructure
616           * in mm/mempolicy.c will do this automatically.
617           * get_policy() must NOT add a ref if the policy at (vma,addr) is not
618           * marked as MPOL_SHARED. vma policies are protected by the mmap_lock.
619           * If no [shared/vma] mempolicy exists at the addr, get_policy() op
620           * must return NULL--i.e., do not "fallback" to task or system default
621           * policy.
622           */
623          struct mempolicy *(*get_policy)(struct vm_area_struct *vma,
624                                          unsigned long addr);
```

Userspace Memory Management

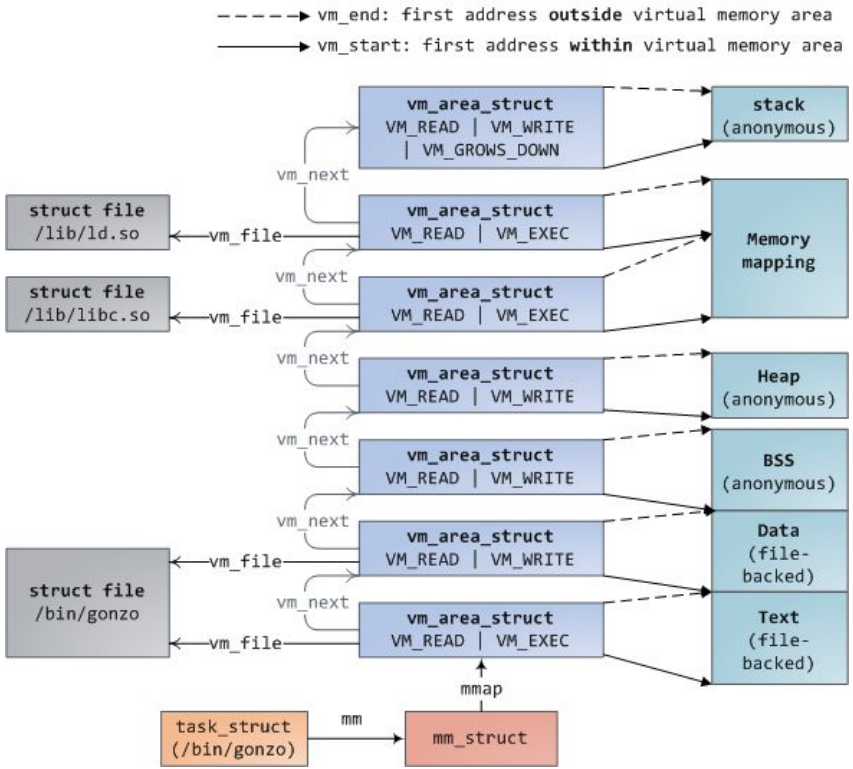


Userspace Memory Management



https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

Userspace Memory Management



https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

PCB Allocation

From Kernel 2.6

Processes are very dynamic entities whose lifetime ranges from a few milliseconds to months. For this reason the kernel must be able to handle many processes at the same time and process descriptors are allocated in **dynamic memory**, rather than in the memory permanently assigned to the kernel. Therefore PCBs can be dynamically allocated upon request.

For each process, Linux packs **two different data structures** in a single per-process memory area: `thread_info` and the Kernel Mode process stack. The length of this area is usually **2 pages** (8,192Kbytes).

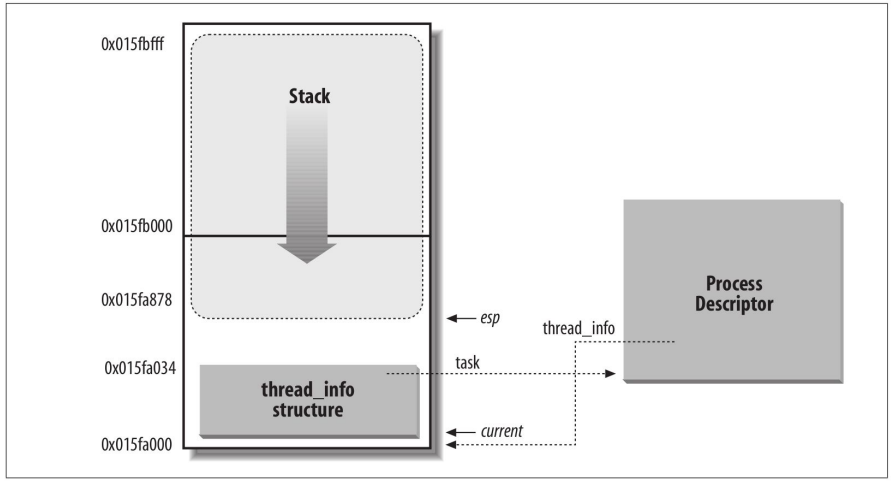


Figure 3-2. Storing the thread_info structure and the process kernel stack in two page frames

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

PCB Allocation

V5.11

`union thread_union`

This union is used to easily allocate `thread_info` at the base of the stack, independently of its size. It works as long as its size is smaller than the stack's, of course, this is mandatory.

```
1732  union thread_union {
1733  #ifndef CONFIG_ARCH_TASK_STRUCT_ON_STACK
1734          struct task_struct task;
1735  #endif
1736  #ifndef CONFIG_THREAD_INFO_IN_TASK
1737          struct thread_info thread_info;
1738  #endif
1739          unsigned long stack[THREAD_SIZE/sizeof(long)];
1740  };
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/sched.h#L1732

PCB Allocation

v4.3

`struct thread_info`

This is the organization of thread_info up to version 4.3. Later on, `thread_info` has been progressively deprived of most members on x86. Security implications of this struct on the stack have been severe.

```
55  struct thread_info {
56          struct task_struct      *task;          /* main task structure */
57          __u32                   flags;          /* low level flags */
58          __u32                   status;         /* thread synchronous flags */
59          __u32                   cpu;            /* current CPU */
60          int                     saved_preempt_count;
61          mm_segment_t            addr_limit;
62          void __user             *sysenter_return;
63          unsigned int            sig_on_uaccess_error:1;
64          unsigned int            uaccess_err:1;  /* uaccess failed */
65  };
```

https://elixir.bootlin.com/linux/v4.3/source/arch/x86/include/asm/thread_info.h

PCB Allocation

`struct thread_info`

V5.11

```
56  struct thread_info {
57          unsigned long           flags;          /* low level flags */
58          unsigned long           syscall_work;   /* SYSCALL_WORK_ flags */
59          u32                     status;         /* thread synchronous flags */
60  };
```

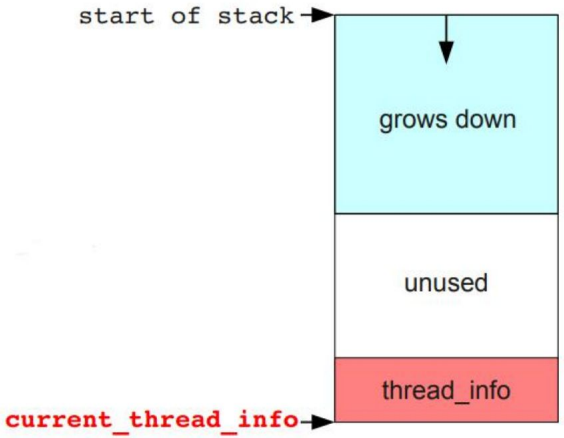
https://elixir.bootlin.com/linux/v5.11/source/arch/x86/include/asm/thread_info.h

Where's the `task_struct` pointer?

Virtually Mapped Kernel Stack

Kernel-level stacks have always been the weak point in the system design. This is quite small: you must be careful to avoid overflows. **Stack overflows** (and also recursion overwrite) have been successfully used as attack vectors.

When an overflow occurs, the Kernel **is not easily able to detect it**. Even less able to counteract on it! Stacks are in the `ZONE_NORMAL` memory and are contiguous but access is done through the MMU via virtual addresses



Virtually Mapped Kernel Stack

There is no need to have a physically contiguous stack, so Andy Lutomirski within its patch proposed to allocate stack relying on `vmalloc()`. This had different benefits:

- resolved the problem of fragmentation (since you do not need anymore contiguous memory for the stack)
- it added graceful handling of overflows, killing the responsible process

But this had a big drawback since it introduced a 1.5µs delay in process creation which was unacceptable. Instead of improving vmalloc subsystem, Linus suggested to add a per-CPU cache of kernel-level stacks getting memory from vmalloc() has been introduced.

In the end it was also decided to move `thread_info` completely off the stack and its content was moved to the `task_struct`.

https://lwn.net/Articles/692208/ - https://lwn.net/Articles/692953/

10.1.1

10. Process Management
1. Process Control Block

Accessing the PCB

DIAG

current

`current` always refers to the currently-scheduled process, it is therefore architecture-specific. It returns the memory address of its PCB (evaluates to a pointer to the corresponding task_struct).

On early versions, it was a macro current defined in `include/asm-i386/current.h` it performed computations based on the value of the stack pointer, by exploiting that the stack is aligned to the couple of pages/frames in memory, therefore changing the stack's size requires re-aligning this macro.

When `thread_info` was introduced, masking the stack gave the address to `task_struct`. To return the `task_struct`, the content of the task member of `task_struct` was returned.

Later, current has been mapped to the static `__always_inline struct task_struct *get_current(void)` function. It returns the per-CPU variable current_task declared in `arch/x86/kernel/cpu/common.c`. The scheduler updates the `current_task` variable when executing a context switch. This is compliant with the fact that `thread_info` has left the stack

current

```
 9  struct task_struct;
10
11  DECLARE_PER_CPU(struct task_struct *, current_task);
12
13  static __always_inline struct task_struct *get_current(void)
14  {
15          return this_cpu_read_stable(current_task);
16  }
17
18  #define current get_current()
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/include/asm/current.h#L18

Accessing the PCB

Up to 2.6

This function in `include/linux/sched.h` allows to retrieve the memory address of the PCB by passing the process/thread pid as input.

```
static inline struct task_struct *find_task_by_pid(int pid) {
     struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];
     for(p = *htable; p && p->pid != pid; p = p->pidhash_next);
     return p;
}
```

Accessing the PCB

Since 2.6

find_task_by_pid has been replaced by:

```
struct task_struct *find_task_by_vpid(pid_t vpid)
```

This is based on the notion of **virtual pid**. It has to do with userspace namespaces, to allow processes in different namespaces to share the same pid numbers.

A **namespace** is a feature of the Linux kernel which partitions the available resources in such a way all the process in the same namespace see the same amount of resources. At boot every process belongs to the same namespace. Namespaces are used for implementing containers. Namespaces are used in conjunction with `cgroups`, another kernel feature that limits the usage of CPU/RAM/IO for a specific set of processes.

Accessing the PCB

Up to 4.14

```
/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;
```

There is a hash defined as below in `include/linux/sched.h`

```
-   #define PIDHASH_SZ (4096 >> 2)
-   extern struct task_struct *pid_hash[PIDHASH_SZ];
-   #define pid_hashfn(x) ((((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

Accessing the PCB

Today

The hash data structure has been replaced by a **radix tree**.

- PIDs are replaced with Integer IDs (`idr`)
- `idr` is the kernel-level library for the management of small integer ID numbers

An `idr` is a sparse array mapping integer IDs onto arbitrary pointers. Look back at the data structures lab.

10.2

10. Process Management

The fork()/exec() model

DIAG

Creating a new process

To create a new process, a couple of `fork()` and `exec*()` calls should be issued. In general new process share everything with the parent so it would be inefficient to truly copy all the data. To overcome this the Linux kernel:

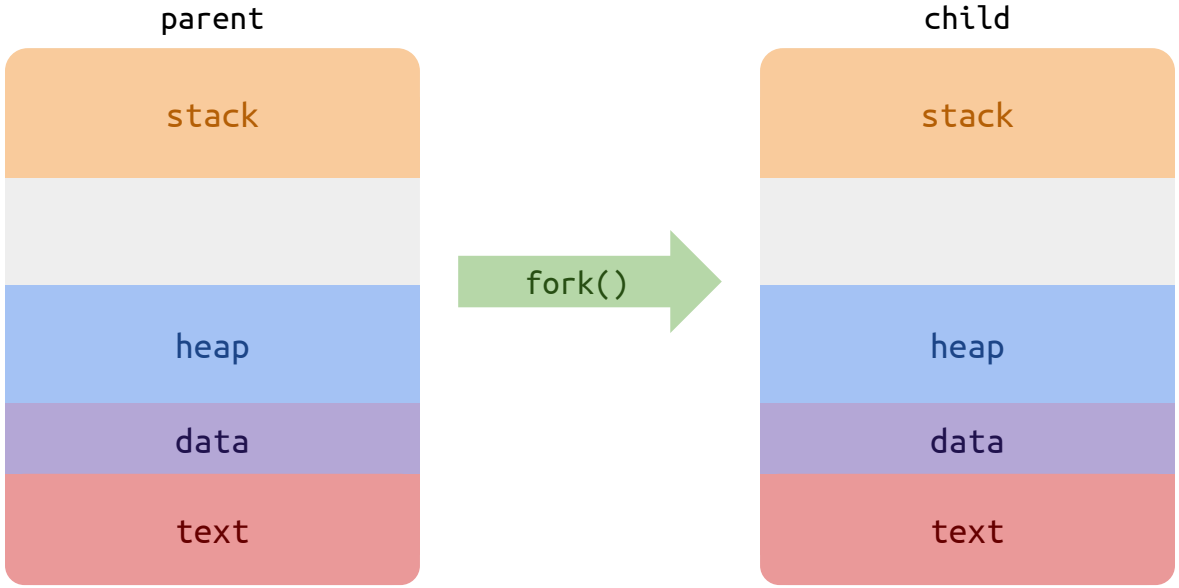
- implements the Copy-on-Write that allows both parent and child to read the same physical pages, whenever one tries to write on a physical page the kernel copies its content into a new physical page;
- lightweight processes allow both parent and child to share many kernel data structures, such as the paging tables, open files struct and signals

Not every child need to share everything from the parent, for this reason right after a `fork()` we can issue an `exec*()`.

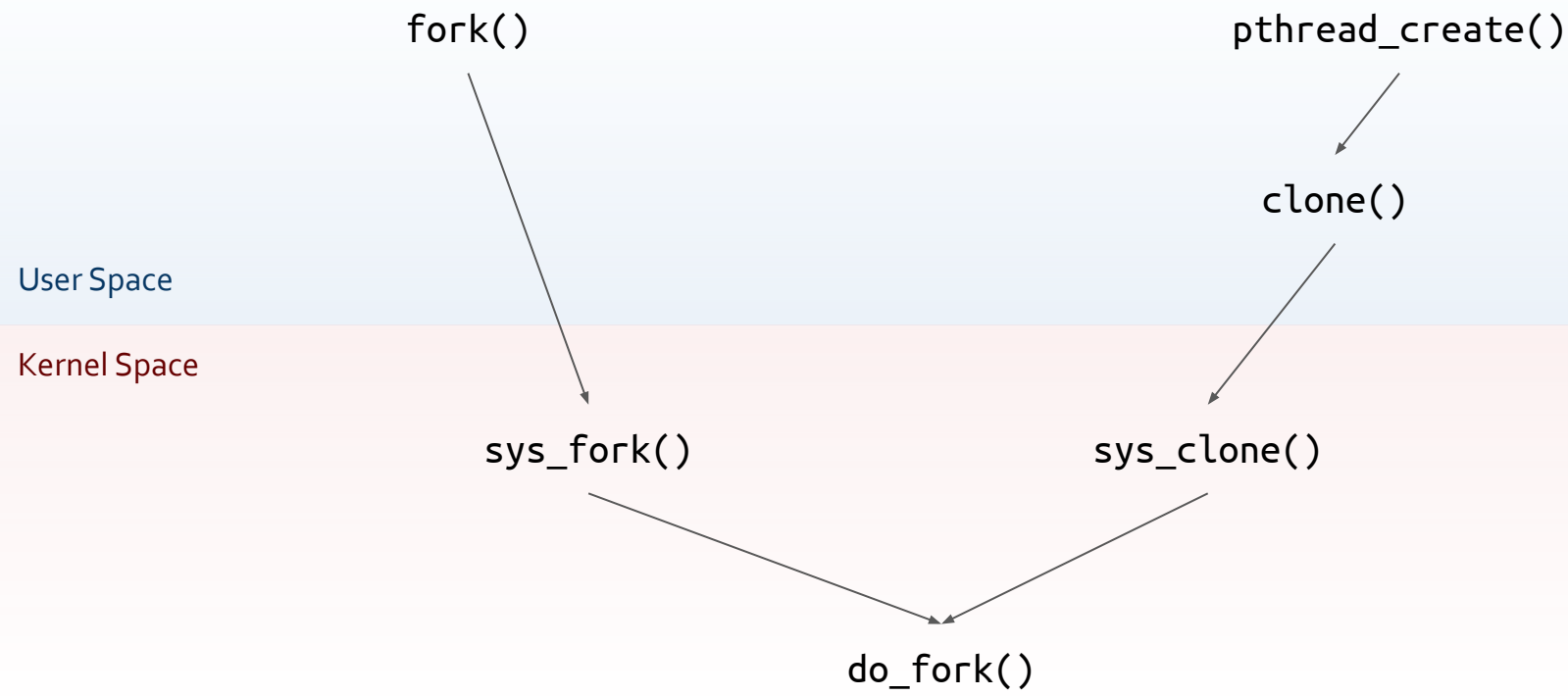
fork()

This function **creates a new process**. The return value is zero in the child and the process-id number of the child in the parent, or -1 upon error.

Both processes start executing from the next instruction to the `fork()` call.



Processes and threads creation



Calling sys_clone() from Userspace

Lightweight processes are created by using a function named `clone()`.

```
long clone(unsigned long flags, void *child_stack, int *ptid,
                    int *ctid, unsigned long newtls);
```

When using `sys_clone()`, we **must allocate** a new stack first. By convention, userspace memory is always allocated from userspace. Indeed, a thread of the same process share the same address space. Also, the TLS (Thread Local Storage) must be allocated in user space, this is architecture-dependent, thus the **`unsigned long`** `type`. `glibc` offers a uniform function but the implementation of the syscall entry points is slightly different on every architecture.

sys_fork() and sys_clone()

```
SYSCALL_DEFINE0(fork)
{
    return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
}

SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp, int
__user *, parent_tidptr, int __user *, child_tidptr, unsigned long, tls)
{
    return _do_fork(clone_flags, newsp, 0, parent_tidptr,
                    child_tidptr, tls);
}
```

do_fork()

V5.11

The do_fork() function makes use of an auxiliary function called copy_process() to set up the process descriptor and any other kernel data structure for child's execution. Here's the main steps:

1. allocates a new PID, a new PCB and a new kernel stack
2. copies PCB information/data structures by using copy_process(). The information copied depends on the passed flags. That for example are:

```
10  #define CSIGNAL         0x000000ff      /* signal mask to be sent at exit */
11  #define CLONE_VM        0x00000100      /* set if VM shared between processes */
12  #define CLONE_FS        0x00000200      /* set if fs info shared between processes */
13  #define CLONE_FILES     0x00000400      /* set if open files shared between processes */
14  #define CLONE_SIGHAND   0x00000800      /* set if signal handlers and blocked signals shared */
15  #define CLONE_PIDFD     0x00001000      /* set if a pidfd should be placed in parent */
16  #define CLONE_PTRACE    0x00002000      /* set if we want to let tracing continue on the child too */
17  #define CLONE_VFORK     0x00004000      /* set if the parent wants the child to wake it up on mm_release */
18  #define CLONE_PARENT    0x00008000      /* set if we want to have the same parent as the cloner */
19  #define CLONE_THREAD    0x00010000      /* Same thread group? */
20  #define CLONE_NEWNS     0x00020000      /* New mount namespace group */
21  #define CLONE_SYSVSEM   0x00040000      /* share system V SEM_UNDO semantics */
22  #define CLONE_SETTLS    0x00080000      /* create a new TLS for the child */
23  #define CLONE_PARENT_SETTID     0x00100000      /* set the TID in the parent */
24  #define CLONE_CHILD_CLEARTID    0x00200000      /* clear the TID in the child */
25  #define CLONE_DETACHED          0x00400000      /* Unused, ignored */
26  #define CLONE_UNTRACED          0x00800000      /* set if the tracing process can't force CLONE_PTRACE on this
27  #define CLONE_CHILD_SETTID      0x01000000      /* set the TID in the child */
28  #define CLONE_NEWCGROUP         0x02000000      /* New cgroup namespace */
29  #define CLONE_NEWUTS            0x04000000      /* New utsname namespace */
30  #define CLONE_NEWIPC            0x08000000      /* New ipc namespace */
31  #define CLONE_NEWUSER           0x10000000      /* New user namespace */
32  #define CLONE_NEWPID            0x20000000      /* New pid namespace */
33  #define CLONE_NEWNET            0x40000000      /* New network namespace */
34  #define CLONE_IO                0x80000000      /* Clone io context */
```

https://elixir.bootlin.com/linux/v5.11/source/include/uapi/linux/sched.h#L11

do_fork()

V5.11

```
2518  SYSCALL_DEFINE0(fork)
2519  {
2520  #ifdef CONFIG_MMU
2521          struct kernel_clone_args args = {
2522                  .exit_signal = SIGCHLD,
2523          };
2524
2525          return kernel_clone(&args);
2526  #else
2527          /* can not support in nommu mode */
2528          return -EINVAL;
2529  #endif
2530  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/fork.c#L2518

```
2413  /*
2414   *  Ok, this is the main fork-routine.
2415   *
2416   * It copies the process, and if successful kick-starts
2417   * it and waits for it to finish using the VM if required.
2418   *
2419   * args->exit_signal is expected to be checked for sanity by the caller.
2420   */
2421  pid_t kernel_clone(struct kernel_clone_args *args)
2422  {
2423          u64 clone_flags = args->flags;
2424          struct completion vfork;
2425          struct pid *pid;
2426          struct task_struct *p;
2427          int trace = 0;
2428          pid_t nr;
2429
2462          p = copy_process(NULL, trace, NUMA_NO_NODE, args);
2463          add_latent_entropy();
2464
2465          if (IS_ERR(p))
2466                  return PTR_ERR(p);
2474          pid = get_task_pid(p, PIDTYPE_PID);
2475          nr = pid_vnr(pid);
2476
2496
2497          put_pid(pid);
2498          return nr;
2499  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/fork.c#L2518

copy_process()

The function implements several checks on namespaces. Pending signals are processed immediately in the parent process.

- `p = dup_task_struct(current, node);`
- `setup_thread_stack(tsk, orig);`
- `copy_creds(p, clone_flags);`
- `copy_files(clone_flags, p);`
- `copy_fs(clone_flags, p);`
- `copy_mm(clone_flags, p);` -> `dup_mm()`

dup_mm()

V5.11

```
1335  /**
1336   * dup_mm() - duplicates an existing mm structure
1337   * @tsk: the task_struct with which the new mm will be associated.
1338   * @oldmm: the mm to duplicate.
1339   *
1340   * Allocates a new mm structure and duplicates the provided @oldmm structure
1341   * content into it.
1342   *
1343   * Return: the duplicated mm or NULL on failure.
1344   */
1345  static struct mm_struct *dup_mm(struct task_struct *tsk,
1346                                  struct mm_struct *oldmm)
1347  {
1348          struct mm_struct *mm;
1349          int err;
1350
1351          mm = allocate_mm();
1352          if (!mm)
1353                  goto fail_nomem;
1354
1355          memcpy(mm, oldmm, sizeof(*mm));
1356
1357          if (!mm_init(mm, tsk, mm->user_ns))
1358                  goto fail_nomem;
1359
1360          err = dup_mmap(mm, oldmm);
1361          if (err)
1362                  goto free_pt;
1363
1364          mm->hiwater_rss = get_mm_rss(mm);
1365          mm->hiwater_vm = mm->total_vm;
```

Allocates a new PGD

https://elixir.bootlin.com/linux/v5.11/source/kernel/fork.c#L1345

10.2.1

10. Process Management
2. The `fork()`/`exec()` model

Kernel Threads

DIAG

Kernel Thread Creation API

V5.11

This is seen as any other task by the scheduler

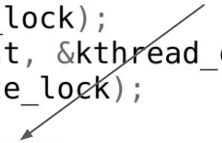
```
11  struct task_struct *kthread_create_on_node(int (*threadfn)(void *data),
12                                             void *data,
13                                             int node,
14                                             const char namefmt[], ...);
15
16  /**
17   * kthread_create - create a kthread on the current node
18   * @threadfn: the function to run in the thread
19   * @data: data pointer for @threadfn()
20   * @namefmt: printf-style format string for the thread name
21   * @arg...: arguments for @namefmt.
22   *
23   * This macro will create a kthread on the current node, leaving it in
24   * the stopped state.  This is just a helper for kthread_create_on_node();
25   * see the documentation there for more details.
26   */
27  #define kthread_create(threadfn, data, namefmt, arg...) \
28          kthread_create_on_node(threadfn, data, NUMA_NO_NODE, namefmt, ##arg)
29
30
31  struct task_struct *kthread_create_on_cpu(int (*threadfn)(void *data),
32                                            void *data,
33                                            unsigned int cpu,
34                                            const char *namefmt);
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/kthread.h#L27

Kthreads are always stopped upon creation, they must be activated by calling `wake_up_process()`.

Kernel Thread Daemon

```
330  struct task_struct *__kthread_create_on_node(int (*threadfn)(void *data),
331                                               void *data, int node,
332                                               const char namefmt[],
333                                               va_list args)
334  {
335          DECLARE_COMPLETION_ONSTACK(done);
336          struct task_struct *task;
337          struct kthread_create_info *create = kmalloc(sizeof(*create),
338                                                       GFP_KERNEL);
339
340          if (!create)
341                  return ERR_PTR(-ENOMEM);
342          create->threadfn = threadfn;
343          create->data = data;
344          create->node = node;
345          create->done = &done;
346
347          spin_lock(&kthread_create_lock);
348          list_add_tail(&create->list, &kthread_create_list);
349          spin_unlock(&kthread_create_lock);
350
351          wake_up_process(kthreadd_task);
```



https://elixir.bootlin.com/linux/v5.11/source/kernel/kthread.c#L330

Kernel Thread Daemon

```
630  int kthreadd(void *unused)
631  {
632          struct task_struct *tsk = current;
633
634          /* Setup a clean context for our children to inherit. */
635          set_task_comm(tsk, "kthreadd");
636          ignore_signals(tsk);
637          set_cpus_allowed_ptr(tsk, housekeeping_cpumask(HK_FLAG_KTHREAD));
638          set_mems_allowed(node_states[N_MEMORY]);
639
640          current->flags |= PF_NOFREEZE;
641          cgroup_init_kthreadd();
642
643          for (;;) {
644                  set_current_state(TASK_INTERRUPTIBLE);
645                  if (list_empty(&kthread_create_list))
646                          schedule();
647                  __set_current_state(TASK_RUNNING);
648
649                  spin_lock(&kthread_create_lock);
650                  while (!list_empty(&kthread_create_list)) {
651                          struct kthread_create_info *create;
652
653                          create = list_entry(kthread_create_list.next,
654                                              struct kthread_create_info, list);
655                          list_del_init(&create->list);
656                          spin_unlock(&kthread_create_lock);
657
658                          create_kthread(create);
659
660                          spin_lock(&kthread_create_lock);
661                  }
662                  spin_unlock(&kthread_create_lock);
663          }
664
665          return 0;
666  }
```

https://elixir.bootlin.com/linux/v5.11/source/kernel/kthread.c#L630

10.3

10. Process Management

Out Of Memory (OOM) Killer

DIAG

The OOM Killer

It is implemented in `mm/oom_kill.c`. This module is activated (if enabled) when the system runs out of memory.

There are three possible actions:

- kill a random task (bad)
- let the system crash (worse)
- try to be smart at picking the process to kill

The OOM Killer picks a "good" process and kills it in order to reclaim available memory.

The OOM Killer

Entry point of the system is `out_of_memory()`. It tries to select the "best" process checking for different conditions:

- if a process has a pending `SIGKILL` or is exiting, this is automatically picked (check done by `task_will_free_mem()`)
- Otherwise, it issues a call to `select_bad_process()` which will return a process to be killed:
  - the picked process is then killed
  - if no process is found, a `panic()` is raised

select_bad_process()

This iterates over all available processes calling `oom_evaluate_task()` on them, until a killable process is found. Unkillable tasks (i.e., kernel threads) are skipped, `oom_badness()` implements the heuristic to pick the process to be killed by computing the "score" associated with each process, the higher the higher the score the higher the probability of getting killed.

oom_badness()

A score of zero is given if:

- the task is unkillable
- the mm field is `NULL`
- if the process is in the middle of a fork

The score is then computed proportionally to the RAM, swap, and pagetable usage:

```
points = get_mm_rss(p->mm) + get_mm_counter(p->mm, MM_SWAPENTS) +
                mm_pgtables_bytes(p->mm) / PAGE_SIZE;
```

10.4

10. Process Management

Process Starting

DIAG

How a Program is Started?

We all know how to compile a program:

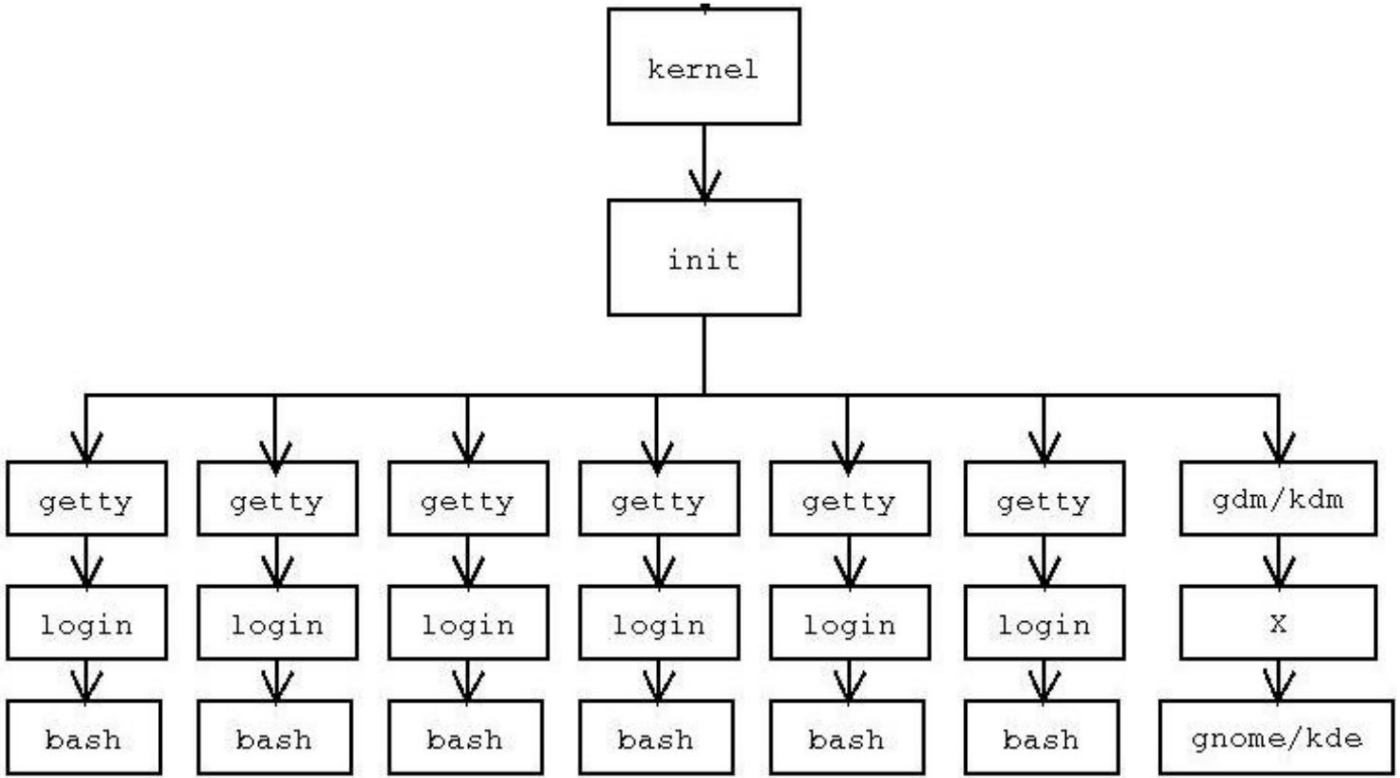
```
gcc program.c -o program
```

We all know how to launch the compiled program:

```
./program
```

The question is: why does all this work? What is the *convention* used between kernel and user space?

In the beginning there was `init`



Starting a program from bash

```
static int execute_disk_command (char *command, int pipe_in,
                                 int pipe_out, int async,
                                 struct fd_bitmap *fds_to_close) {
    pid_t pid;
    pid = make_child (command, async);

    if (pid == 0) {
        shell_execve (command, args, export_env);
    }
}
```

From the bash shell source - https://github.com/bminor/bash/blob/master/execute_cmd.c

Starting a program from bash

```
pid_t make_child (char *command, int async_p) {
    pid_t pid;
    int forksleep;
    start_pipeline();
    forksleep = 1;
    while ((pid = fork ()) < 0 && errno == EAGAIN && forksleep < FORKSLEEP_MAX) {
        sys_error("fork: retry");

        reap_zombie_children();
        if (forksleep > 1 && sleep(forksleep) != 0)
            break;
        forksleep <<= 1;
    }

    /* ... */

    return (pid);
}
```

https://github.com/bminor/bash/blob/f3a35a2d601a55f337f8ca02a541f8c033682247/jobs.c#L2132

Starting a program from bash

```
int shell_execve (char *command, char **args, char **env) {
    execve (command, args, env);

    READ_SAMPLE_BUF (command, sample, sample_len);

    if (sample_len == 0)
        return (EXECUTION_SUCCESS);

    if (sample_len > 0) {
        if (sample_len > 2 && sample[0] == '#' && sample[1] == '!')
            return (execute_shell_script(sample, sample_len, command, args, env));
        else if (check_binary_file (sample, sample_len)) {
            internal_error (_("%s: cannot execute binary file"), command);
            return (EX_BINARY_FILE);
        }
    }

    longjmp(subshell_top_level, 1);
}
```

```
/* These are extern so execute_simple_command can set them, and then
   longjmp back to main to execute a shell script, instead of calling
   main () again and resulting in indefinite, possibly fatal, stack
   growth. */
procenv_t subshell_top_level;
```

https://github.com/bminor/bash/blob/master/shell.c

https://github.com/bminor/bash/blob/f3a35a2d601a55f337f8ca02a541f8c033682247/execute_cmd.c#L5794

exec*

exec*( ) changes the program file that an existing process is running:

- it first **wipes** out the memory state of the calling process
- it then goes to the filesystem to **find** the program file requested
- it **copies** this file into the program's memory and initializes register state, including the PC
- It **doesn't alter** most of the other fields in the PCB. The process calling exec*( ) (the child copy of the shell, in this case) can, e.g., change the opened files

Let's see how exec*() is implemented.

struct linux_binprm

The **struct** linux_binprm is in charge of keeping information about a binary file.

```
struct linux_binprm {
        char buf[BINPRM_BUF_SIZE];
        struct page *page[MAX_ARG_PAGES];
        unsigned long p; /* current top of mem */
        int sh_bang;
        struct file* file;
        int e_uid, e_gid;
        kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
        int argc, envc;
        char *filename; /* Name of binary */
        unsigned long loader, exec;
};
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/binfmts.h#L17

do_execve()

In kernel 5.11 do_execve_atcommon()

https://elixir.bootlin.com/linux/v5.11/source/fs/exec.c#L1855

```
int do_execve(char *filename, char **argv, char **envp, struct pt_regs *regs) {
        struct linux_binprm bprm;
        struct file *file;
        int retval;
        int i;

        file = open_exec(filename);

        retval = PTR_ERR(file);
        if (IS_ERR(file))
                return retval;

        bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
        memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));
        bprm.file = file;
        bprm.filename = filename;
        bprm.sh_bang = 0;
        bprm.loader = 0;
        bprm.exec = 0;
        if ((bprm.argc = count(argv, bprm.p / sizeof(void *))) < 0) {
                allow_write_access(file);
                fput(file);
                return bprm.argc;
        }
```

count(argv, max) counts the number of strings

do_execve()

```
if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {
        allow_write_access(file);
        fput(file);
        return bprm.envc;
}

retval = prepare_binprm(&bprm);
if (retval < 0)
        goto out;

retval = copy_strings_kernel(1, &bprm.filename, &bprm);
if (retval < 0)
        goto out;

bprm.exec = bprm.p;
retval = copy_strings(bprm.envc, envp, &bprm);
if (retval < 0)
        goto out;

retval = copy_strings(bprm.argc, argv, &bprm);
if (retval < 0)
        goto out;

retval = search_binary_handler(&bprm,regs);
if (retval >= 0)
        /* execve success */
        return retval;
```

do_execve()

```
out:
/* Something went wrong, return the inode and free the argument pages*/
allow_write_access(bprm.file);
if (bprm.file)
     fput(bprm.file);

for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
     struct page * page = bprm.page[i];
     if (page)
          __free_page(page);
}

return retval;
```

search_binary_handler()

The function scans a list of binary file handlers registered in the kernel. If no handler is able to recognize the image format, syscall returns the ENOEXEC error (“Exec Format Error”).

For ELF files we have in `fs/binfmt_elf.c`:

- `load_elf_binary()`, the function:
  - loads image file to memory using mmap;
  - reads the program header and sets permissions accordingly
  - `elf_ex = *((struct elfhdr *)bprm->buf);`

10.4.1

10. Process Management
4. Process Starting

The ELF Format

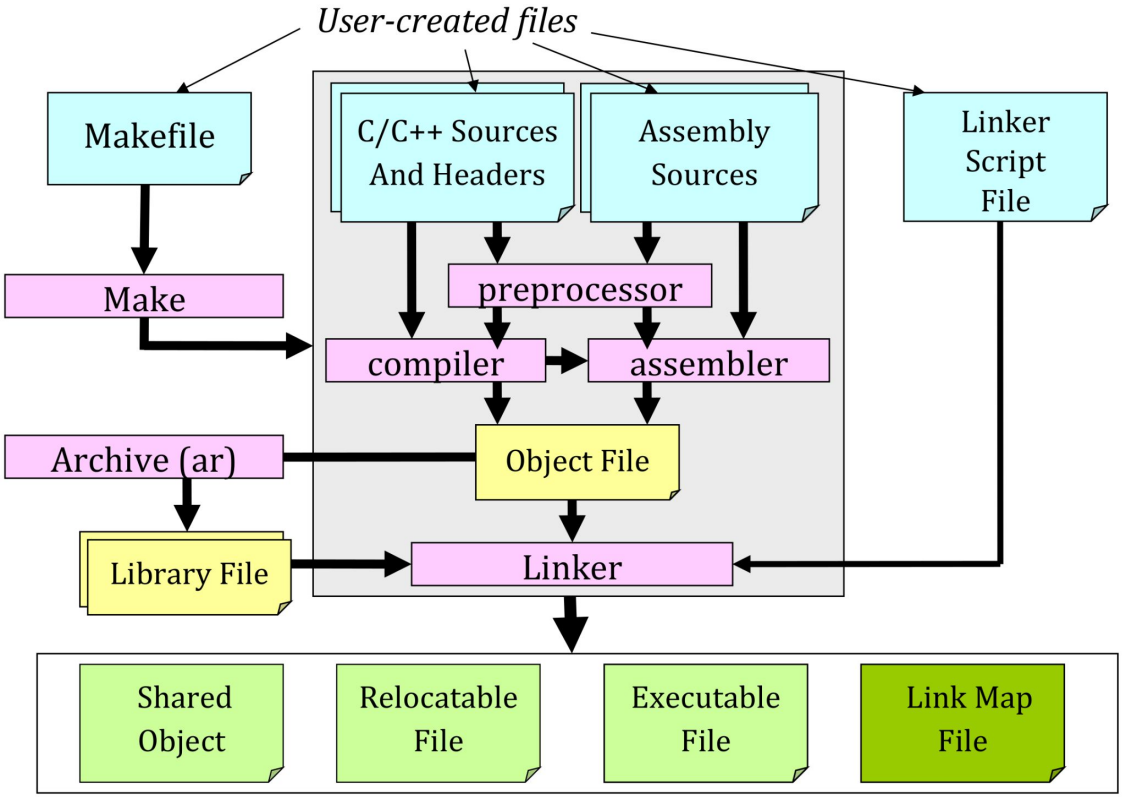
DIAG

ELF: Executable and Linking Format

The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). The Tool Interface Standards committee (TIS) has selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems.

The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments. This should reduce the number of different interface implementations, thereby reducing the need for recoding and recompiling code.

The Compiling Process



ELF Format

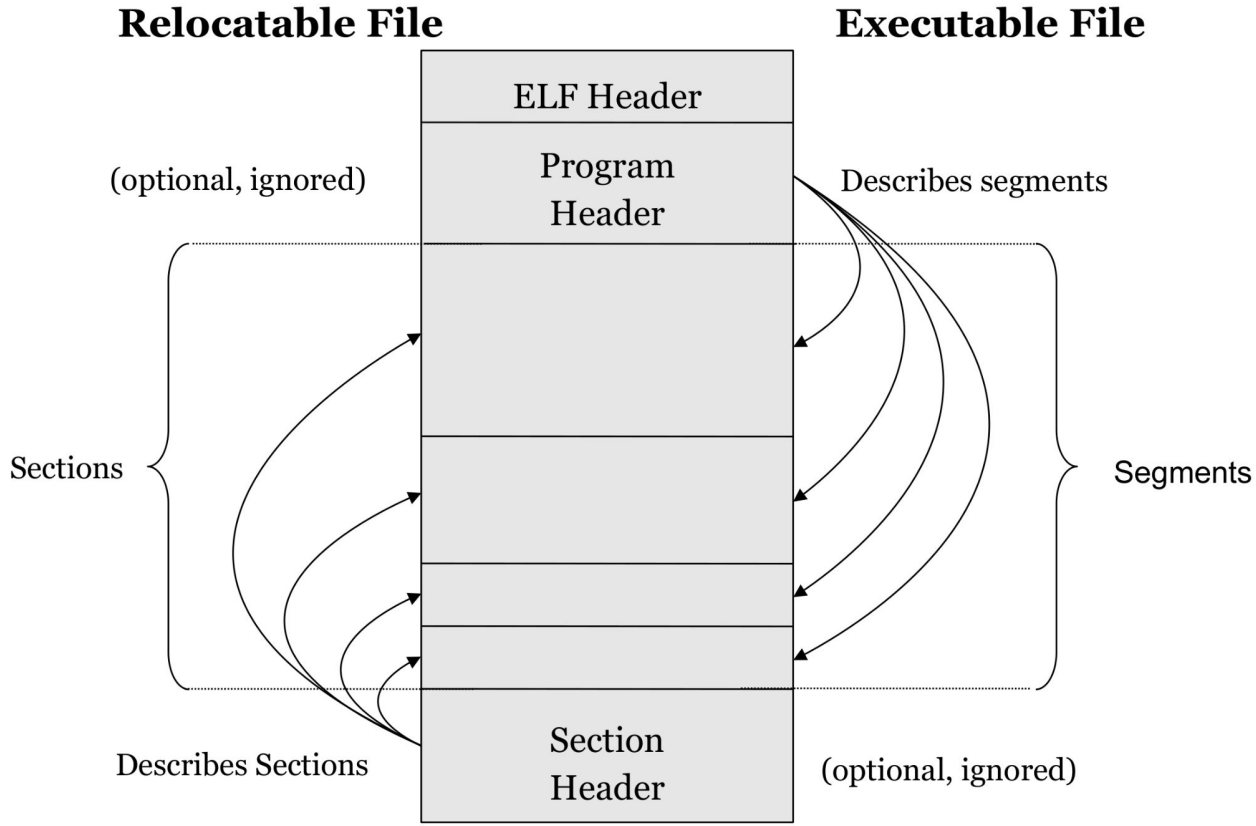
ELF defines the format of binary executables. There are four different categories:

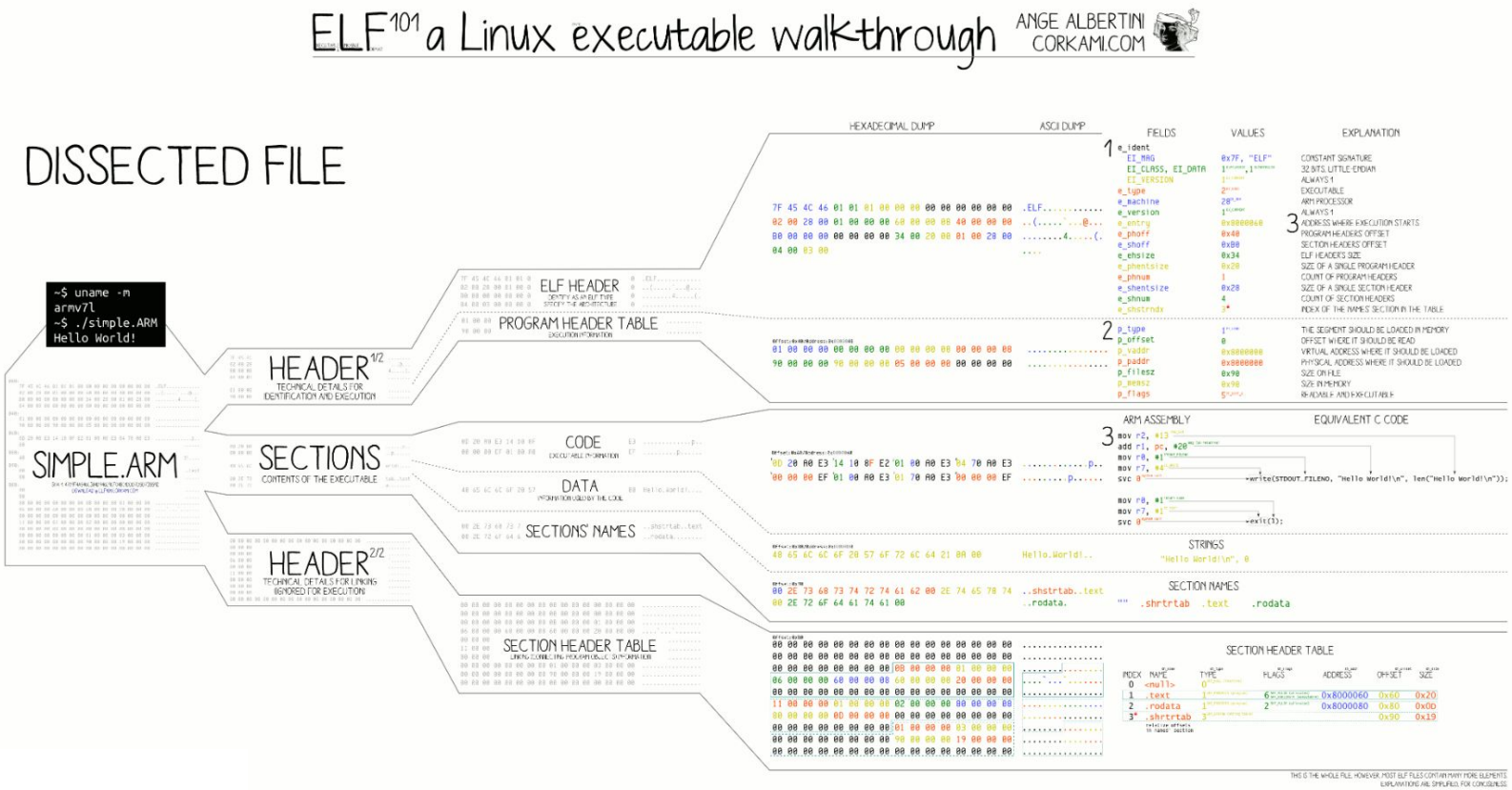
- **Relocatable**, created by compilers and assemblers. Must be processed by the linker before being run.
- **Executable**, all symbols are resolved, except for shared libraries' symbols, which are resolved at runtime.
- **Shared object**, a library which is shared by different programs, contains all the symbols' information used by the linker, and the code to be executed at runtime.
- **Core file**, a core dump.

ELF files have a twofold nature

- compilers, assemblers and linkers handle them as a set of **logical** sections;
- the system loader handles them as a set of **segments**.

ELF Format


Relocatable File
Executable File
ELF Header
Program Header
(optional, ignored)
Describes segments
Sections
Segments
Describes Sections
Section Header
(optional, ignored)



https://upload.wikimedia.org/wikipedia/commons/e/e4/ELF_Executable_and_Linkable_Format_diagram_by_Ange_Albertini.png

Relocatable File

A relocatable file or a shared object is a collection of sections. Each section contains a single kind of information, such as executable code, read-only data, read/write data, relocation entries, or symbols.

Each symbol's address is defined in relation to the section which contains it. For example, a function's entry point is defined in relation to the section of the program which contains it.

Section Header

```
typedef struct {
    Elf32_Word sh_name; /* Section name (string tbl index) */
    Elf32_Word sh_type; /* Section type */
    Elf32_Word sh_flags; /* Section flags */
    Elf32_Addr sh_addr; /* Section virtual addr at execution */
    Elf32_Off sh_offset; /* Section file offset */
    Elf32_Word sh_size; /* Section size in bytes */
    Elf32_Word sh_link; /* Link to another section */
    Elf32_Word sh_info; /* Additional section information */
    Elf32_Word sh_addralign; /* Section alignment */
    Elf32_Word sh_entsize; /* Entry size if section holds table */
} Elf32_Shdr;
```

Section Header

Types and Flags

Types:

- `PROGBITS`: The section contains the program content (code, data, debug information).
- `NOBITS`: Same as `PROGBITS`, yet with a null size.
- `SYMTAB` and `DYNSYM`: The section contains a symbol table.
- `STRTAB`: The section contains a string table.
- `REL` and `RELA`: The section contains relocation information.
- `DYNAMIC` and `HASH`: The section contains dynamic linking information.

Flags:

- `WRITE`: The section contains runtime-writeable data.
- `ALLOC`: The section occupies memory at runtime.
- `EXECINSTR`: The section contains executable machine instructions.

Sections

Examples

- `.text`: contains program's instructions
  - Type: `PROGBITS`
  - Flags: `ALLOC` + `EXECINSTR`
- `.data`: contains pre-initialized read/write data
  - Type: `PROGBITS`
  - Flags: `ALLOC` + `WRITE`
- `.rodata`: contains pre-initialized read-only data
  - Type: `PROGBITS`
  - Flags: `ALLOC`
- `.bss`: contains uninitialized data. Will be set to zero at startup.
  - Type: `NOBITS`
  - Flags: `ALLOC` + `WRITE`

Executable Files

Usually, an executable file has only few segments:

- A read-only segment for code.
- A read-only segment for read-only data.
- A read/write segment for other data.

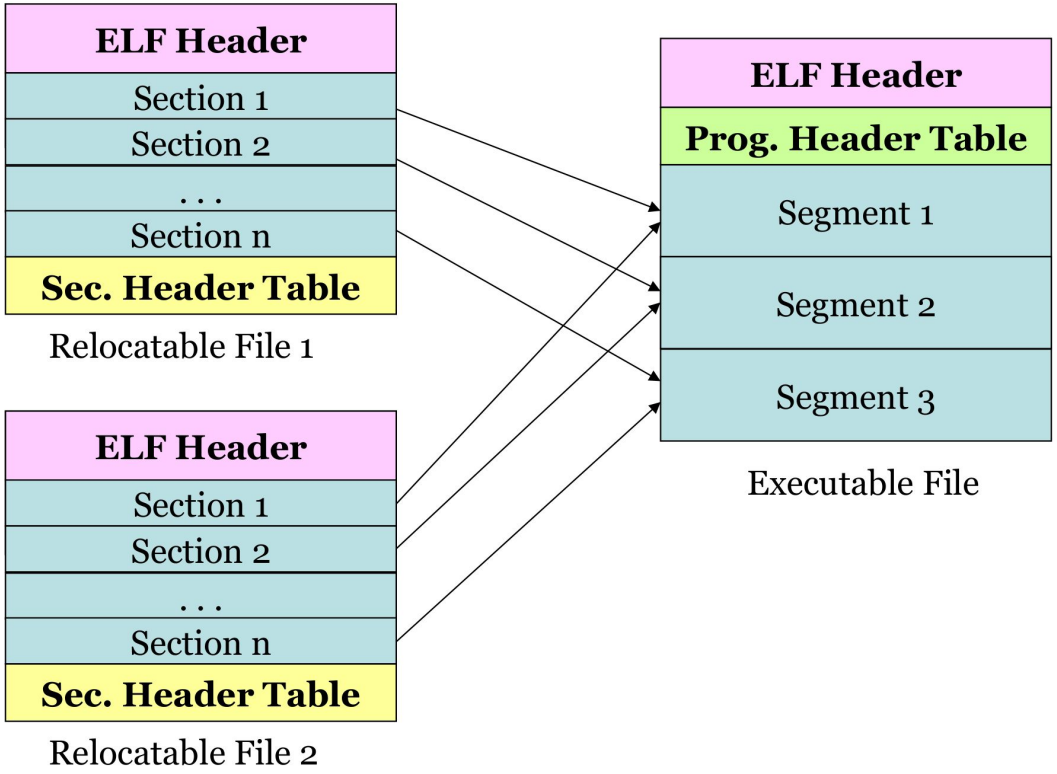
Any section marked with flag `ALLOC` is packed in the proper segment, so that the operating system is able to map the file to memory with few operations.

If `.data` and `.bss` sections are present, they are placed within the same read/write segment.

Program Header

```
typedef struct {
    Elf32_Word p_type; /* Segment type */
    Elf32_Off p_offset; /* Segment file offset */
    Elf32_Addr p_vaddr; /* Segment virtual address */
    Elf32_Addr p_paddr; /* Segment physical address */
    Elf32_Word p_filesz; /* Segment size in file */
    Elf32_Word p_memsz; /* Segment size in memory */
    Elf32_Word p_flags; /* Segment flags */
    Elf32_Word p_align; /* Segment alignment */
} Elf32_Phdr;
```

Linker's Role



Static Relocation

text section

```
                (1)                  ...
1bc1: e8 00 00 00 00        (call ???)
1bc6: 83 c4 10              add $0x10, %rsp
1bc9: a1 00 00 00 00        (movb 0x0, %eax)
     (1)    (2)                  ...
2bd7: 55                    push %rbp
2bd8: 48 89 e5              mov %rsp, %rbp
```

data section

```
732e 6d79 6174 0062 732e 7274 6174 0062
732e 7368 7274 6174 0062 742e 7865 0074
642e 7461 0061 622e 7373 6174 0062 7865
```

string table

```
NUL f o o NUL m y _ v a r NUL
    (1)   (2)
```

info tells you the index in the symbol table and the type of the symbol

	offset	info	addend
(1)	1bc2	(1) 0 /off	4
(2)	1bca	(2) 1 / addr	0

.text.rela table

value is the address in the section

	name	value	section
(1)	1	2bd7	text
(2)	5	812f	data

symbol table

name is the position of the name in the string table

Symbols Visibility

A symbol can be:

- **strong**, a strong symbol replaces a weak one and if two strong symbols have the same name the linker resolves in favour of the first; by default every symbol is strong
- **weak**, more modules can have a symbol with the same name of a weak one, the declared entity cannot be overloaded by other modules; It is useful for libraries which want to avoid conflicts with user programs.

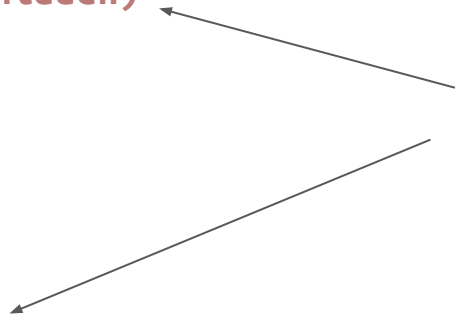
gcc version 4.0 gives the command line option `-fvisibility`:

- default: normal behaviour, the symbol is seen by other modules;
- hidden: two declarations of an object refer the same object only if they are in the same shared object;
- internal: an entity declared in a module cannot be referenced even by pointer;
- protected: the symbol is weak;

Symbols Visibility

```
int variable __attribute__ ((visibility (“hidden”)));

#pragma GCC visibility push(hidden)
int variable;
int increment(void) {
     return ++variable;
}
#pragma GCC visibility pop
```



Set the default declaration as hidden in the scope

10.4.2

10. Process Management
4. Process Starting

Dynamic Linking

DIAG

Program Entry Point

The main() function is not the actual entry point for the program. glibc inserts auxiliary functions. The actual entry point is called _start.

The Static Relocation works at linking time but you obviously do not want to include all the libraries that you use in your program in your executable file, this because eats up memory and almost all the programs use the same set of libraries (e.g. the stdlib). Symbols that are not included in the final executable file are resolved with the Dynamic Linking that is performed by the kernel when the program starts.

The Kernel starts the dynamic linker which is stored in the `.interp` section of the program (usually `/lib/ld-linux.so.2`). If no dynamic linker is specified, control is given at address specified in `e_entry`.

Dynamic Linking

Initialization steps:

- Self initialization
- Loading Shared Libraries
- Resolving remaining relocations
- Transfer control to the application

The most important data structures which are filled are:

- **Procedure Linkage Table** (PLT), used to call functions whose address isn't known at link time
- **Global Offsets Table** (GOT), similarly used to resolve addresses of data/functions

Dynamic Linking

Data Structures

- `.dynsym`: a minimal symbol table used by the dynamic linker when performing relocations
- `.hash`: a hash table that is used to quickly locate a given symbol in the .dynsym, usually in one or two tries.
- `.dynstr`: string table related to the symbols stored in .dynsym

These tables are used to fill the GOT table, that is populated upon need (lazy binding).

Dynamic Linking

Steps

The first PLT entry is special. Other entries are identical, one for each function needing resolution.

1. A jump to a location which is specified in a corresponding GOT entry
2. Preparation of arguments for a resolver routine
3. Call to the resolver routine, which resides in the first entry of the PLT

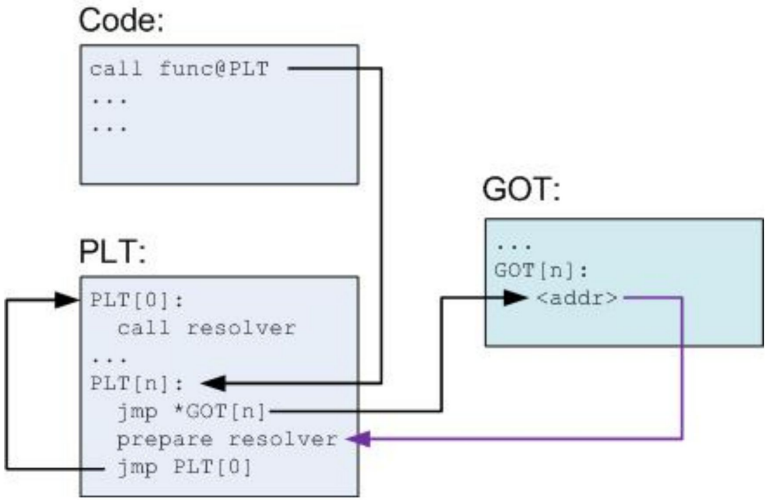
The first PLT entry is a call to the ***resolver*** located in the dynamic loader itself.

Dynamic Linking

Steps

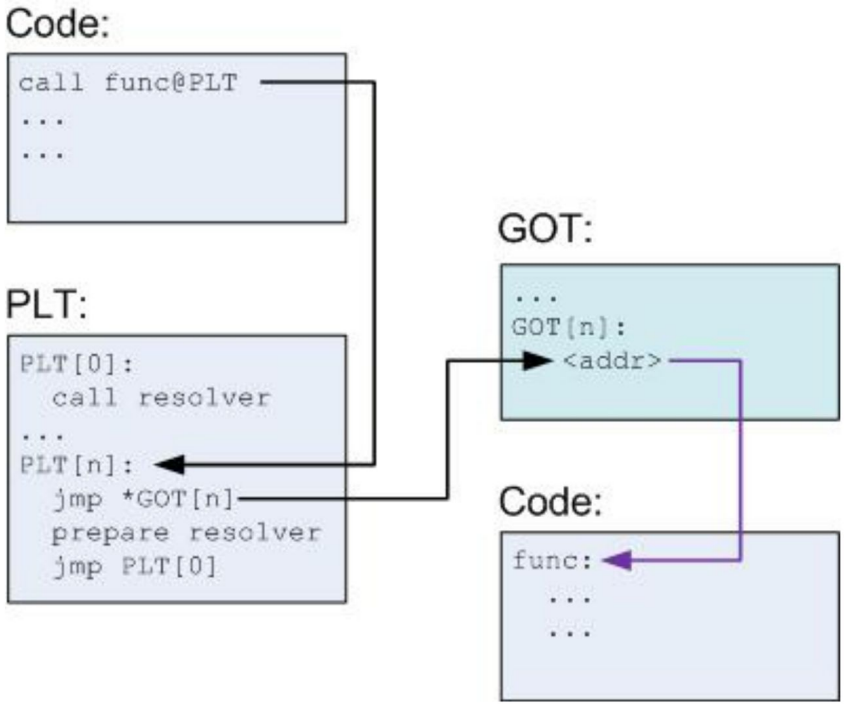
When **func** is called for the first time:

1. PLT[n] is called, and jumps to the address pointed to it in GOT[n]
2. This address points into PLT[n] itself, to the preparation of arguments for the resolver.
3. The resolver is then called, by jumping to PLT[0]
4. The resolver performs resolution of the actual address of func, places its actual address into GOT[n] and calls func.



Dynamic Linking

Steps after the first resolution



10.4.3

10. Process Management
4. Process Starting

Initial Steps of Programs' Life

DIAG

Initial Steps

So far the dynamic linker has loaded the shared libraries in memory. GOT is populated when the program requires certain functions. Then, the dynamic linker calls `_start`

```
<_start>:
      xor %ebp, %ebp
      pop %esi
      mov %esp, %ecx
      and $0xfffffff0, %esp
      push %eax
      push %esp
      push %edx
      push $0x8048600
      push $0x8048670
      push %ecx
      push %esi
      push $0x804841c
      call 8048338 <__libc_start_main>
      hlt
      nop
      nop
```

Suggested by ABI to mark outermost frame

the pop makes argc go into %esi

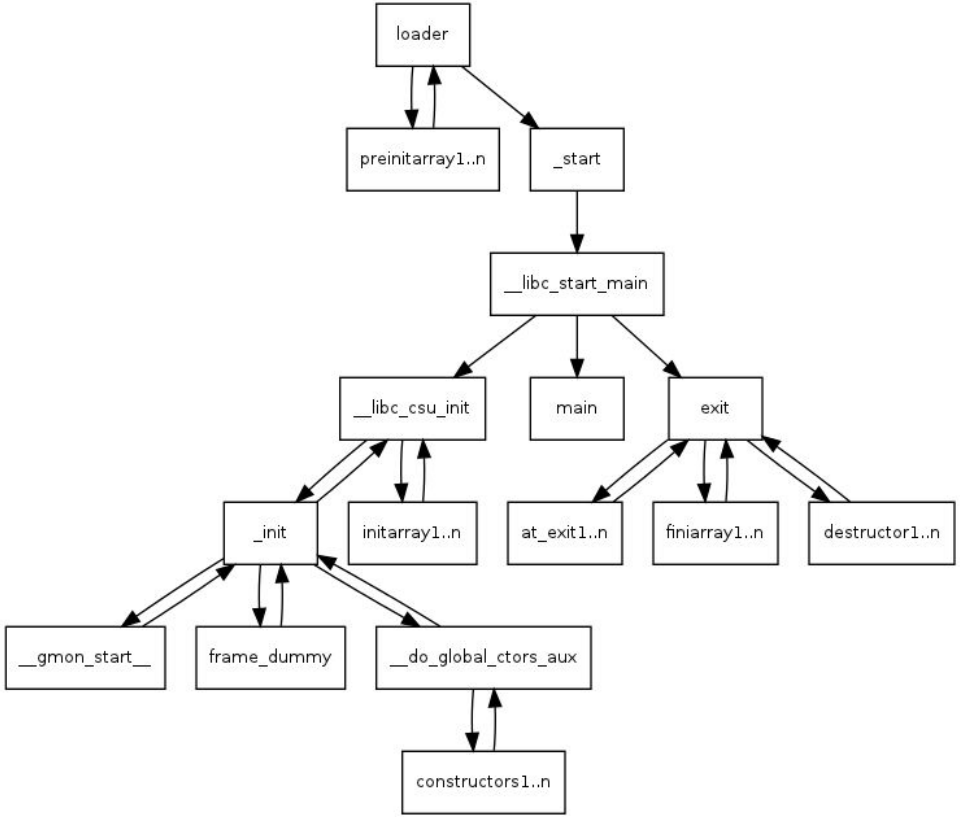
%esp is now pointing at argv. The mov puts argv into %ecx without moving the stack pointer

Align the stack pointer to a multiple of 16 bytes

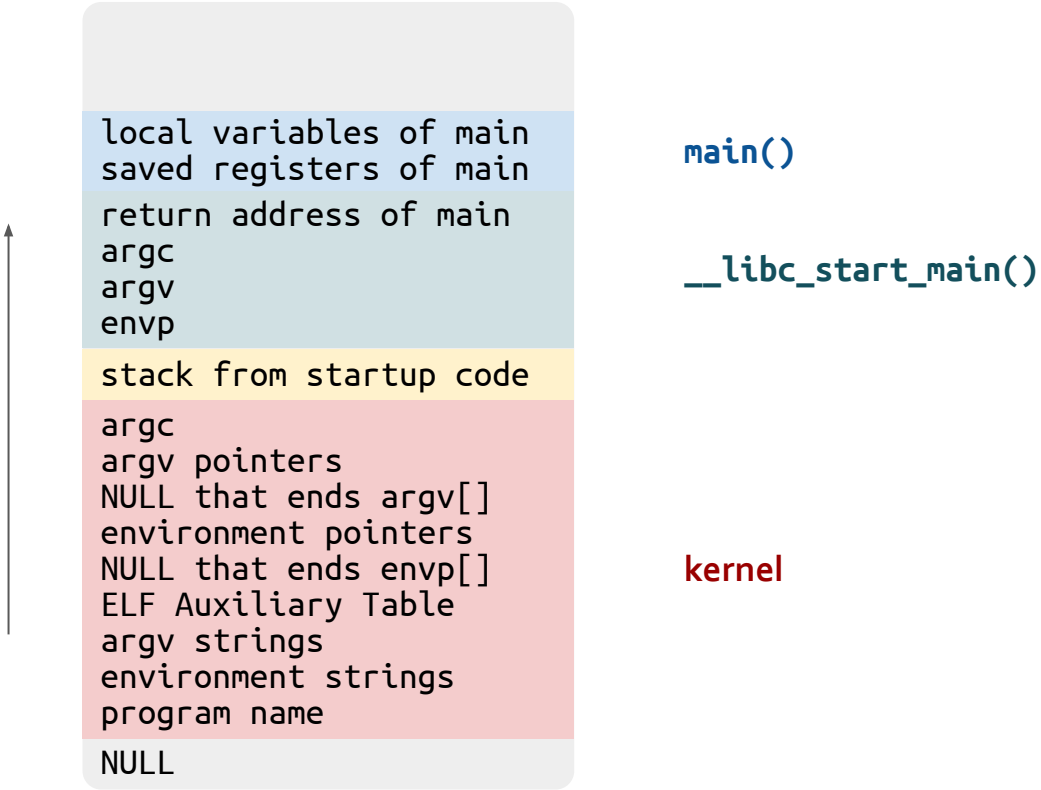
Prepare parameters to __libc_start_main %eax is garbage, to keep the alignment

This instruction should never be executed

Userspace Life of a Program



Stack Layout at Program Startup



Advanced Operating Systems and Virtualization

[10] Process Management

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG