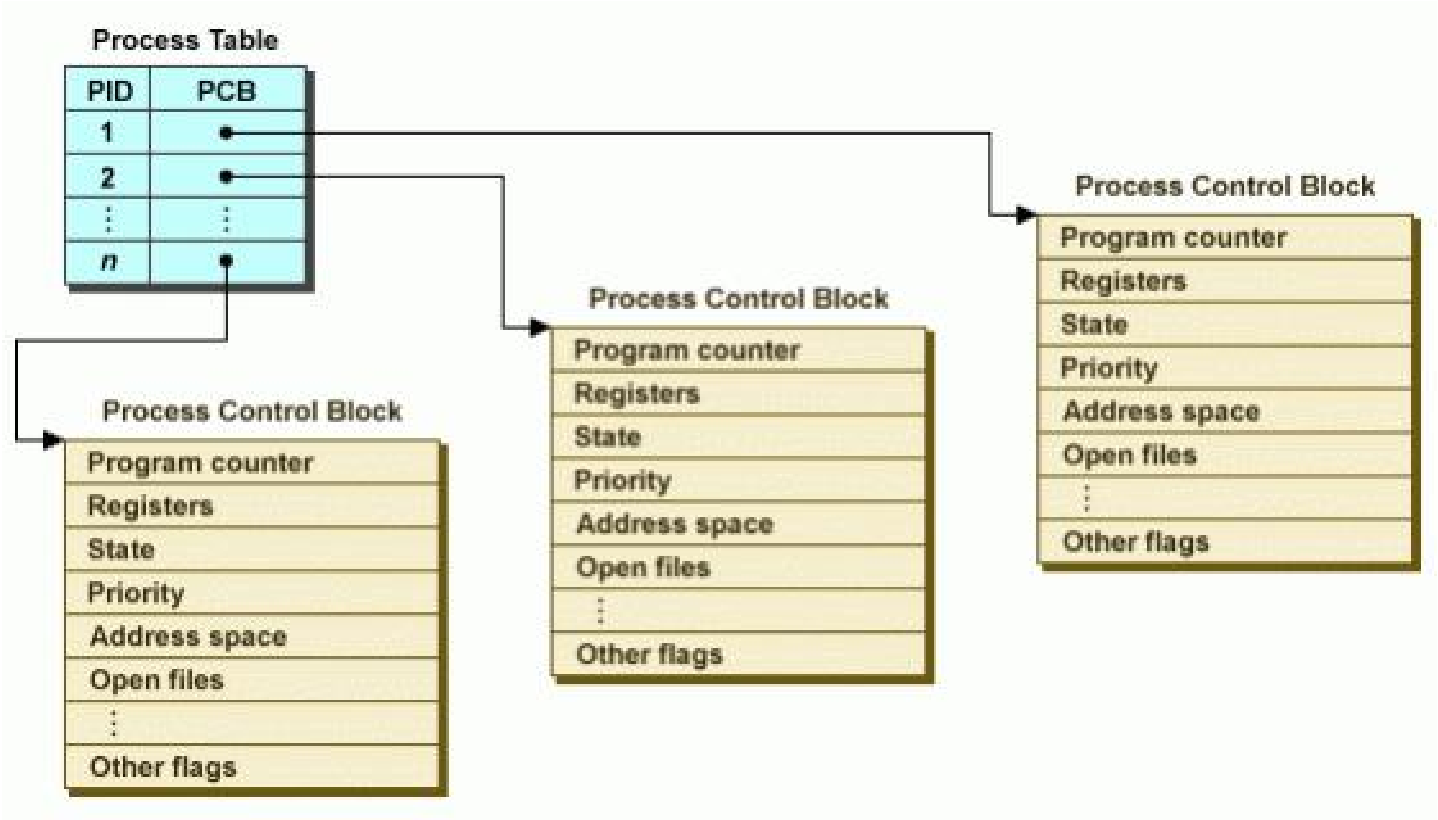
Process Management and Startup

Advanced Operating Systems and Virtualization
Alessandro Pellegrini
A.Y. 2019/2020

Process Control Block



Process Control Block

- This is `struct task_struct` in `include/linux/sched.h`
- 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;`

The mm member

- mm points to a mm_struct defined in include/linux/mm_types.h
- 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

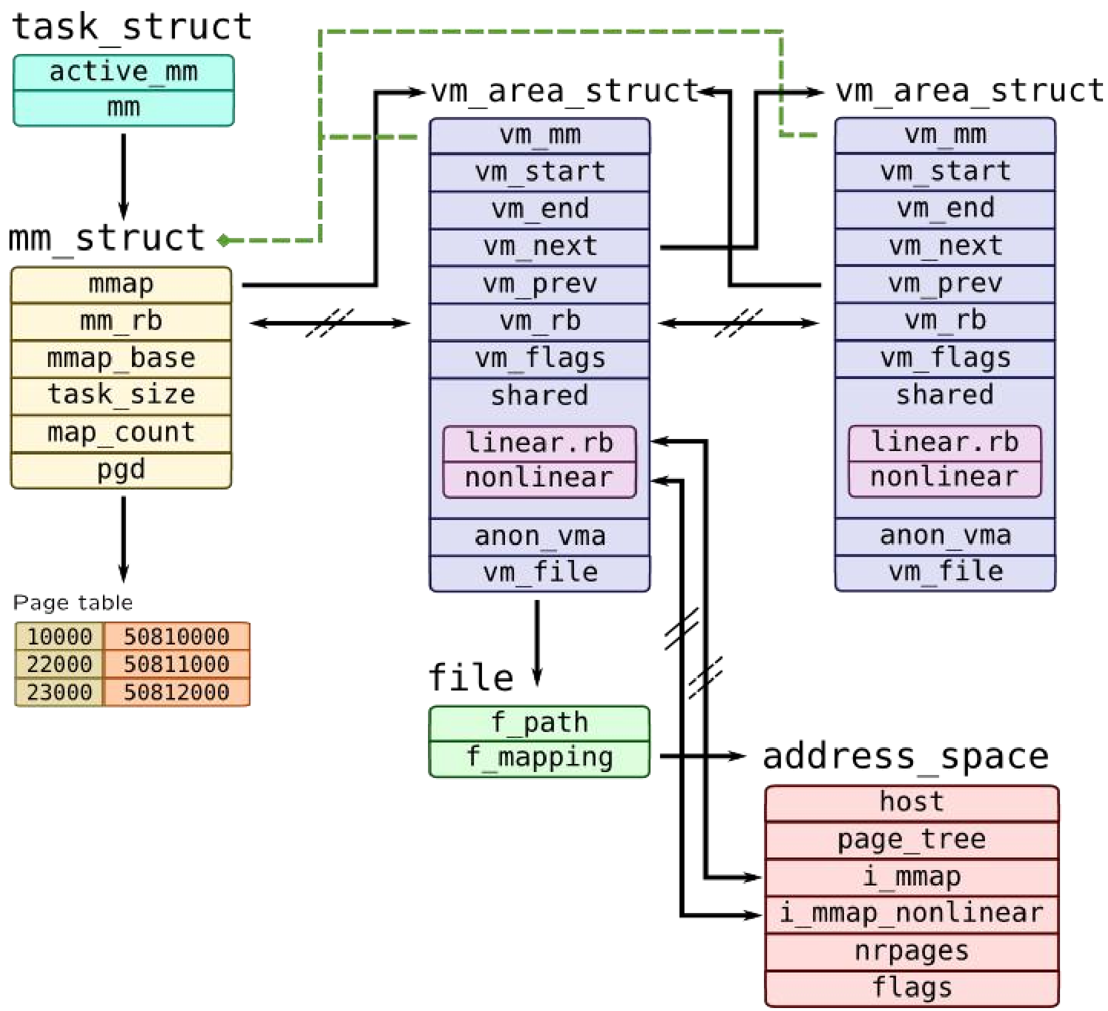
vm_area_struct

- Describes a Virtual Memory Area (VMA):
  - `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

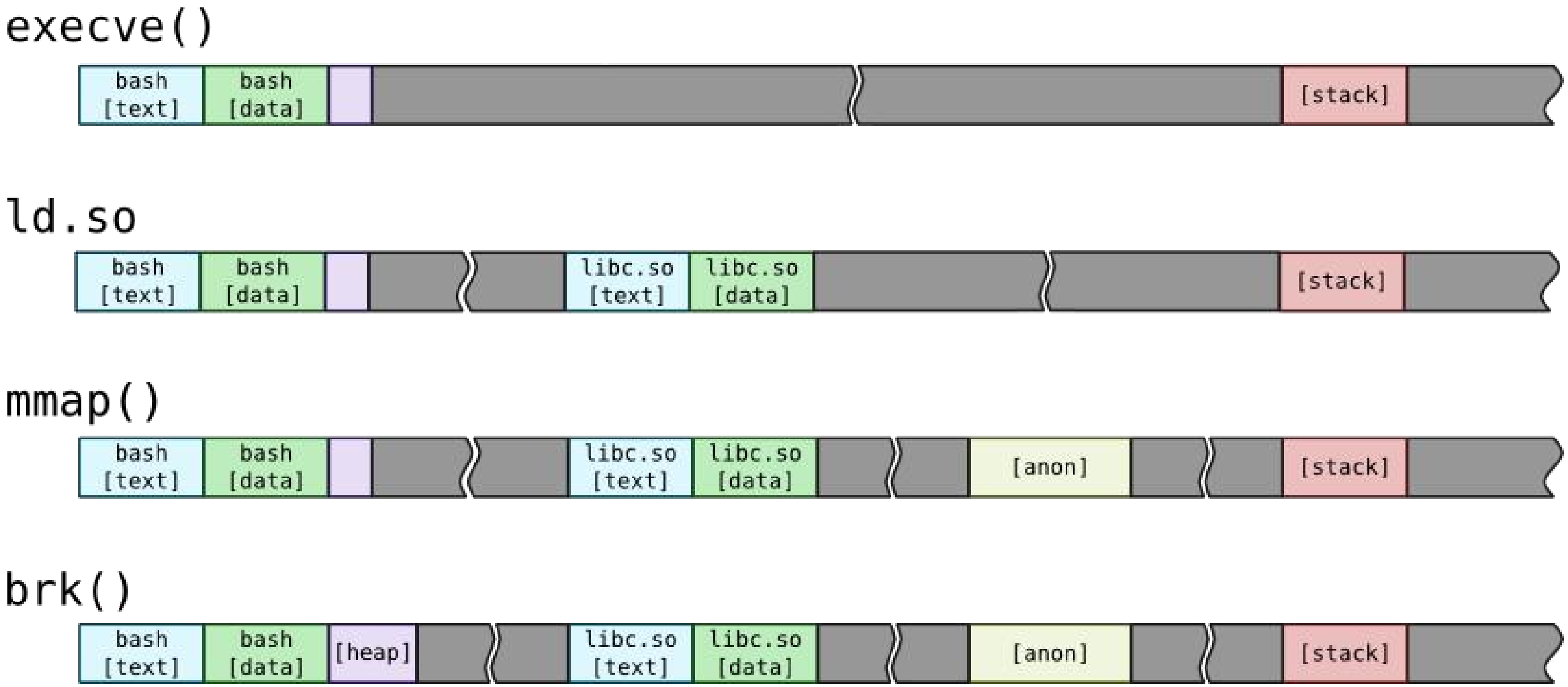
vm_operations_struct

```
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);
        void (*map_pages)(struct vm_area_struct *vma, struct
                           vm_fault *vmf);

        /* notification that a previously read-only page is about
         * to become writable, if an error is returned it will
         * cause a SIGBUS */
        int (*page_mkwrite)(struct vm_area_struct *vma, struct
                            vm_fault *vmf);
        ...
        int (*set_policy)(struct vm_area_struct *vma, struct
                          mempolicy *new);
        struct mempolicy *(*get_policy)(struct vm_area_struct
                          *vma, unsigned long addr);
};
```

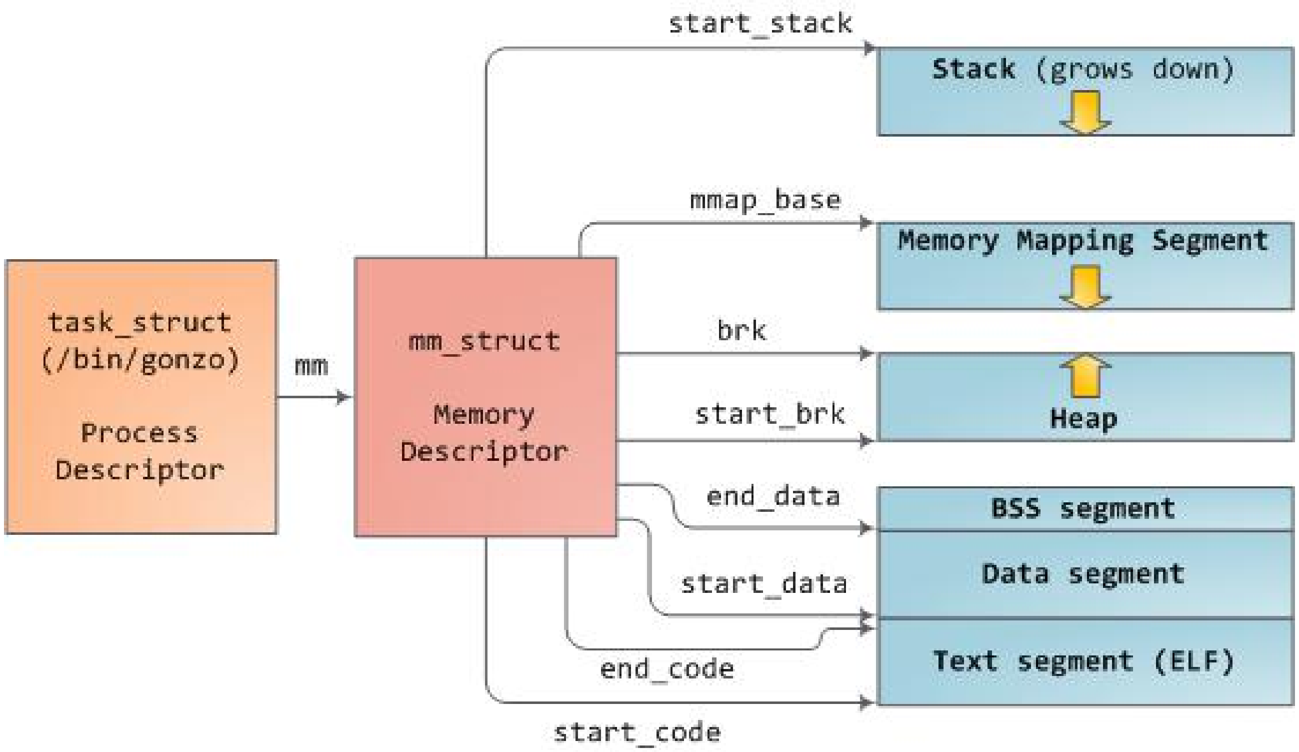
Userspace Memory Management



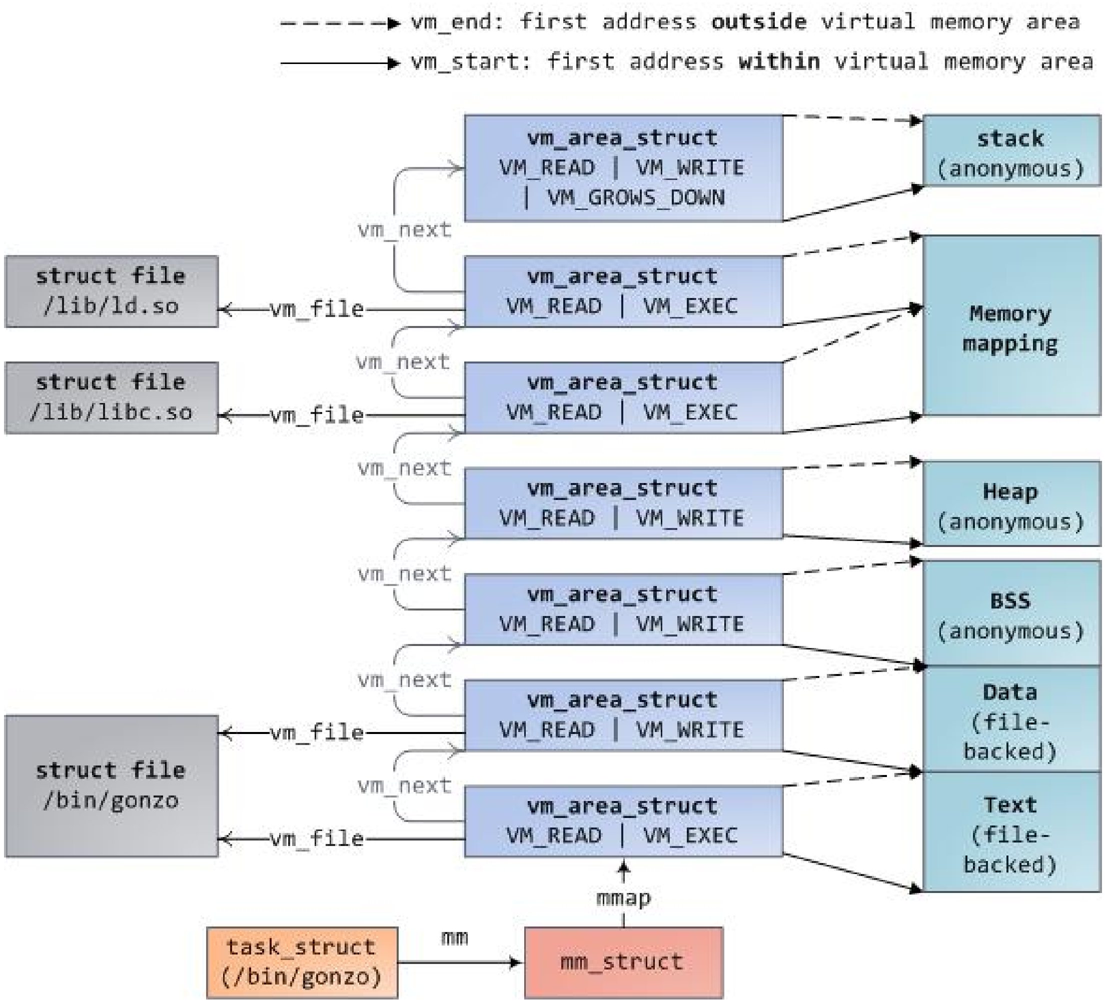
Userspace Memory Management



Userspace Memory Management

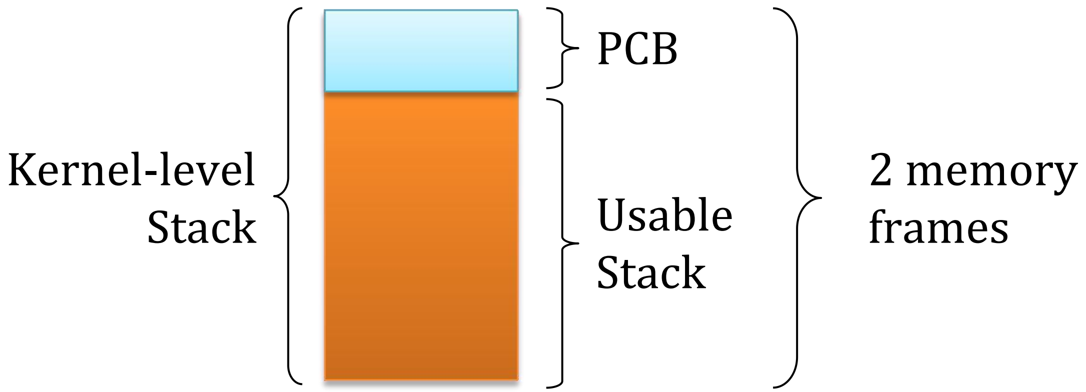


Userspace Memory Management



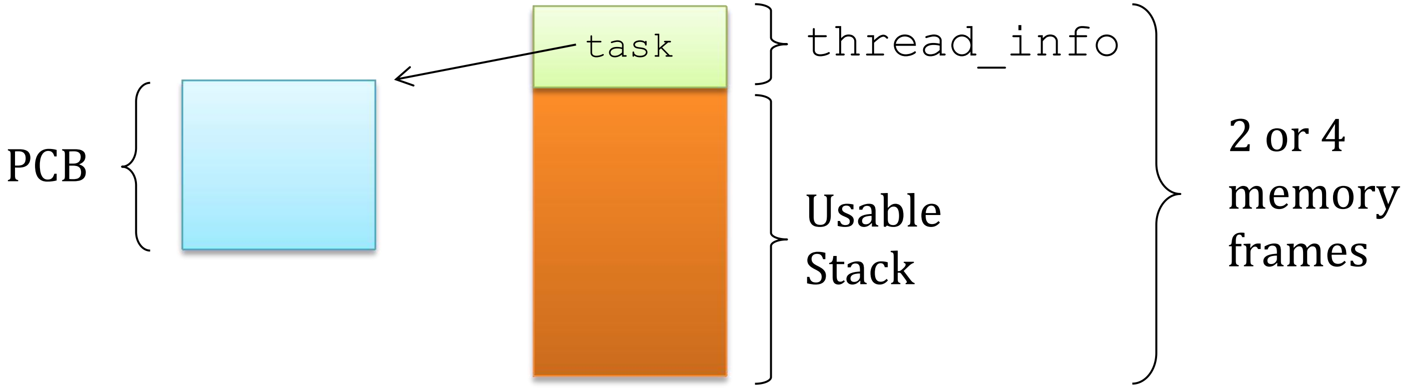
PCB Allocation up to 2.6

- PCBs can be dynamically allocated upon request
- The PCB is directly stored at the bottom of the kernel-level stack of the process which the PCB refers to



PCB Allocation since 2.6

- The PCB is moved outside of the kernel-level stack
- At the top, there is the `thread_info` data structure



`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

```
union thread_union {
      struct thread_info thread_info;
      unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

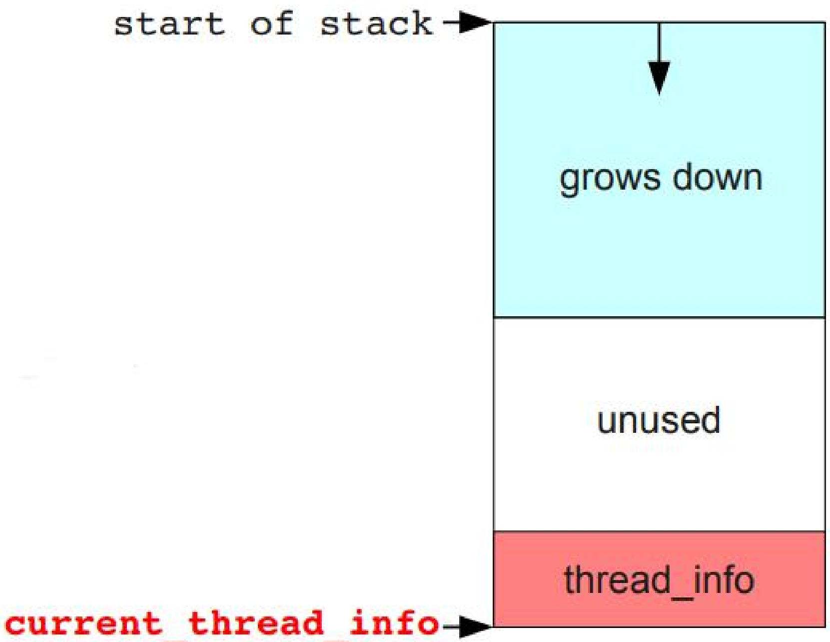
struct thread_info

- 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

```
struct thread_info {
          struct task_struct  *task;             /* main task structure */
          struct exec_domain  *exec_domain;      /* execution domain */
          __u32      flags;                      /* low level flags */
          __u32      status;                     /* thread synchronous flags */
          __u32      cpu;                        /* current CPU */
          int                                    saved_preempt_count;
          mm_segment_t                           addr_limit;
          void __user                            *sysenter_return;
          unsigned int                           sig_on_uaccess_error:1;
          unsigned int                           uaccess_err:1;     /* uaccess failed */
};
```

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



`struct thread_info` in 3.19.8

```
struct thread_info {
  struct task_struct *task;
  struct exec_domain *exec_domain;
  __u32 flags;
  __u32 status;
  __u32 cpu;
  int preempt_count;
  mm_segment_t addr_limit;
  struct restart_block restart_block;
  ...
};
```

U/K Boundary!
(affects, e.g., `access_ok()`)
(can write into kmem)

Has a function pointer!
(triggered by syscall `restart()`)
(can be overridden with userspace pointers)

Virtually Mapped Kernel Stack

- 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 stack was created relying on `vmalloc()`
- This introduced a 1.5μs delay in process creation which was unacceptable
- A cache of kernel-level stacks getting memory from `vmalloc()` has been introduced
- This allows to introduce surrounding unmapped pages
- `thread_info` is moved off the stack
  - it's content is moved to the `task_struct`

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
- Changing the stack's size requires re-aligning this macro

current

- When `thread_info` was introduced, masking the stack gived 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

Accessing PCBs (up to 2.6.26)

- 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 PCBs (after 2.6.26)

- `find_task_by_pid` has been replaced :
  - `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

Accessing PCBs (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 PCBs (currently)

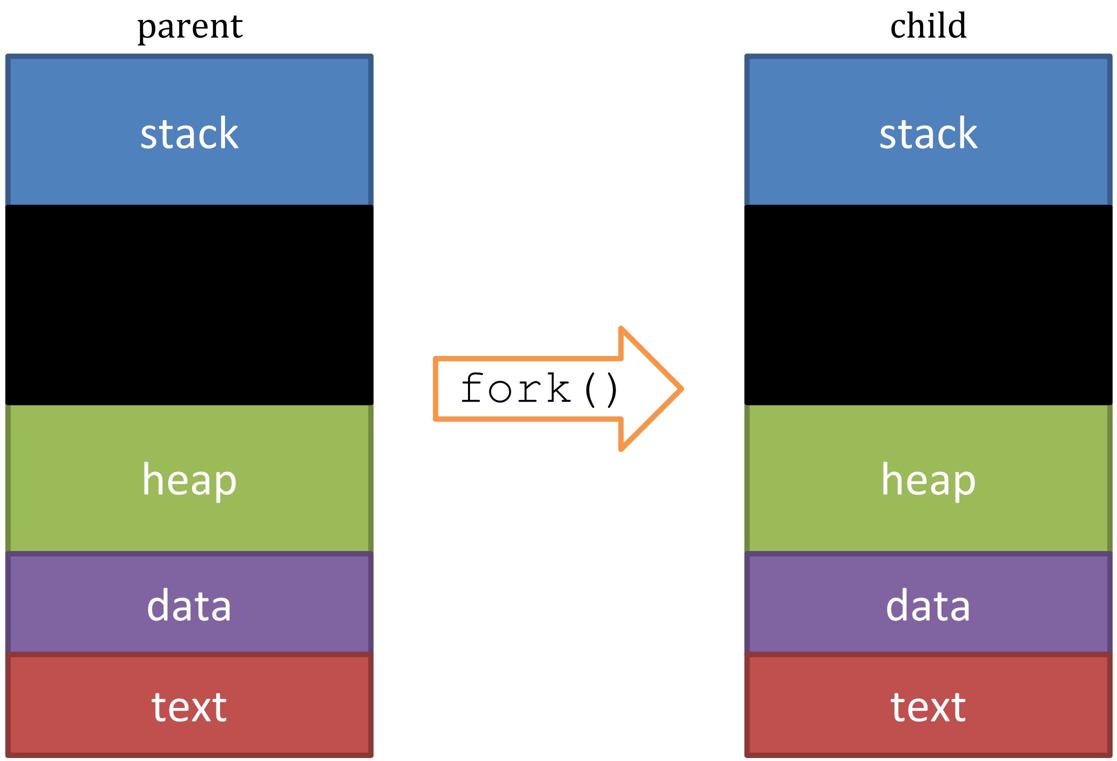
- The hash data structure has been replaced by a *radix tree*
- PIDs are replaced with Integer IDs (idr)
- idr is the kernel-level library for the management of small integer ID numbers
- An idr is a sparse array mapping integer IDs onto arbitrary pointers
  - Look back at the data structures lecture

`fork()`/`exec()` Model

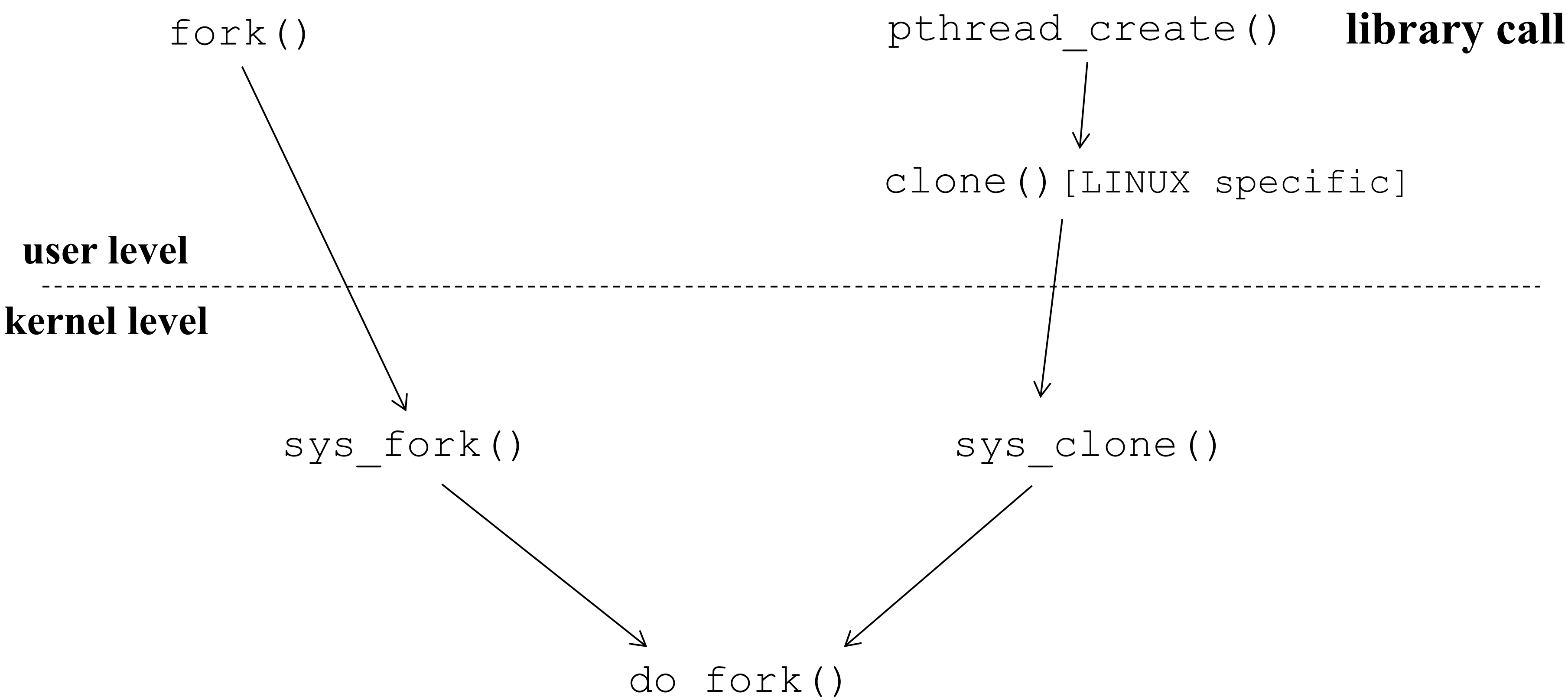
- To create a new process, a couple of `fork()` and `exec*()` calls should be issued
  - Unix worked mainly with multiprocessing (shared memory)
  - `fork()` relies on COW
  - `fork()` followed by `exec*()` allows for fast creation of new processes, both for sharing memory view or not

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.



Process and thread creation



Calling `sys_clone()` from Userspace

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

- When usign `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, TLS must be allocated in user space
  - This is architecture-dependent, thus the `unsigned long` type
- glibc offers a uniform function
  - 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()

- Fresh PCB/kernel-stack allocation
- Copy/setup of PCB information/data structures
- What information is copied or inherited (namely shared into the original buffers) depends on the value of the flags passed as input to `do_fork()`
- Legit values for the flags are defined in `include/linux/sched.h`
  - `CLONE_VM`: set if VM is shared between processes
  - `CLONE_FS`: set if fs info shared between processes
  - `CLONE_FILES`: set if open files shared between processes
  - `CLONE_PID`: set if pid shared
  - `CLONE_PARENT`: set if we want to have the same parent as the cloner

do_fork() (5.0)

```
long do_fork(unsigned long clone_flags, unsigned long stack_start,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr,
             unsigned long tls)
{
    struct pid *pid;
    struct task_struct *p;
        ...
    p = copy_process(clone_flags, stack_start, stack_size, child_tidptr,
                     NULL, trace, tls, NUMA_NO_NODE);
        ...
    pid = get_task_pid(p, PIDTYPE_PID);
        ...
    wake_up_new_task(p);
}
```

`copy_process()`

- Copy process 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()

```
static struct mm_struct *dup_mm(struct task_struct *tsk)
{
        struct mm_struct *mm, *oldmm = current->mm;
        mm = allocate_mm();
...
        memcpy(mm, oldmm, sizeof(*mm));
        if (!mm_init(mm, tsk, mm->user_ns))
                goto fail_nomem;
        err = dup_mmap(mm, oldmm);
        if (err)
                goto free_pt;
...
        return mm;
...
}
```

allocates new PGD

Kernel Thread Creation API

This is seen as a task by the scheduler

Entry point parameters

```
struct task_struct *kthread_create(
      int (*function)(void *data), void *data,
       const char namefmt[], ...)
```

The name of the thread

The thread entry point

- Kthreads are stopped upon creation
- It must be activated with a call to `wake_up_process()`

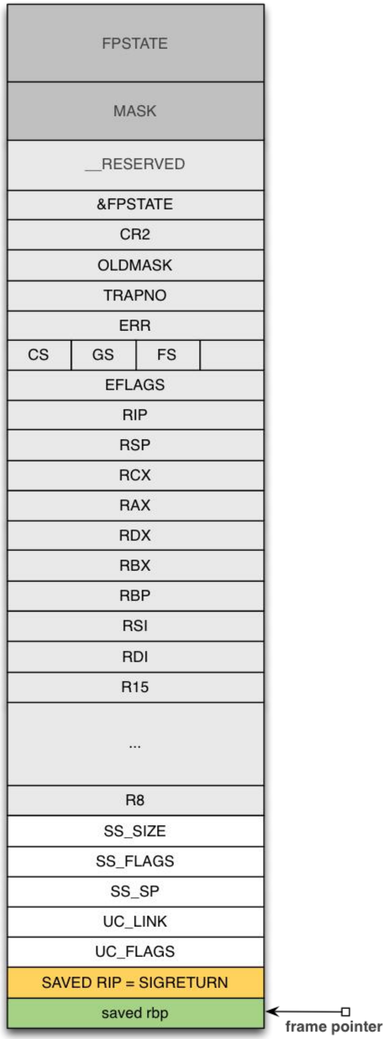
__kthread_create_on_node()

```
struct task_struct *__kthread_create_on_node(int (*threadfn)(void *data),
                          void *data, int node,
                          const char namefmt[],
                          va_list args)
{
    struct task_struct *task;
    struct kthread_create_info *create = kmalloc(sizeof(*create), GFP_KERNEL);

    if (!create)
        return ERR_PTR(-ENOMEM);
        create->threadfn = threadfn;
        create->data = data;
        create->node = node;
        create->done = &done;

        spin_lock(&kthread_create_lock);
        list_add_tail(&create->list, &kthread_create_list);
        spin_unlock(&kthread_create_lock);

        wake_up_process(kthreadd_task);
        ...
}
```

Kernel Thread Daemon

Signal Handlers Management

- Once a non-masked pending signal is found for a certain process, before returning control to it a proper stack is assembled
- Control is then returned to the signal handler



Out of Memory (OOM) Killer

- 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

Out of Memory (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
  - it computes the "score" associated with each process, 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:

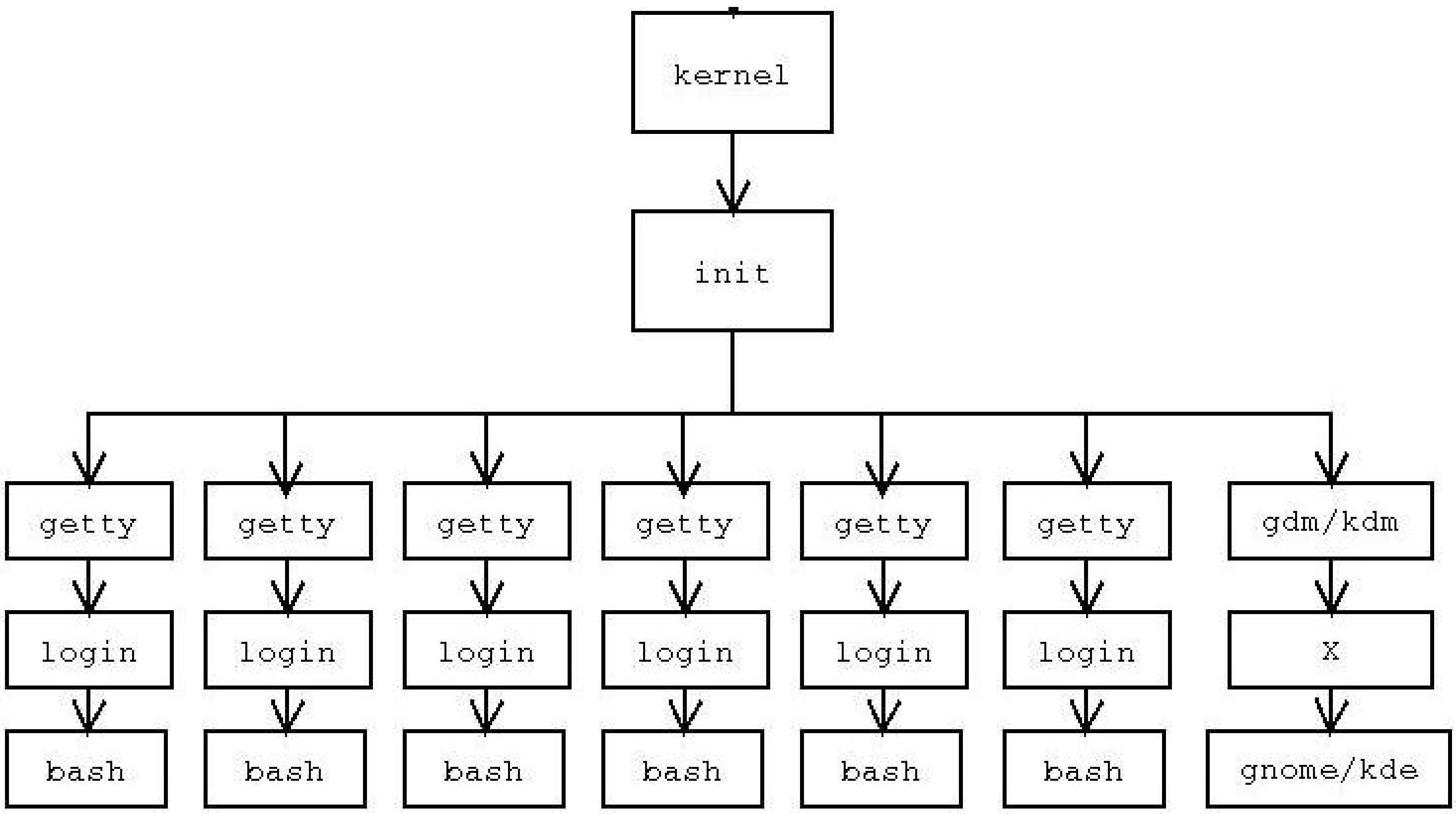
```
points = get_mm_rss(p->mm) + get_mm_counter(p->mm, MM_SWAPENTS) +
                mm_pgtables_bytes(p->mm) / PAGE_SIZE;
```

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);
  }
}
```

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);
}
```

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);
}
```

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 open files

struct linux_binprm

```
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;
};
```

do_execve()

```
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;
    }
```

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;
```

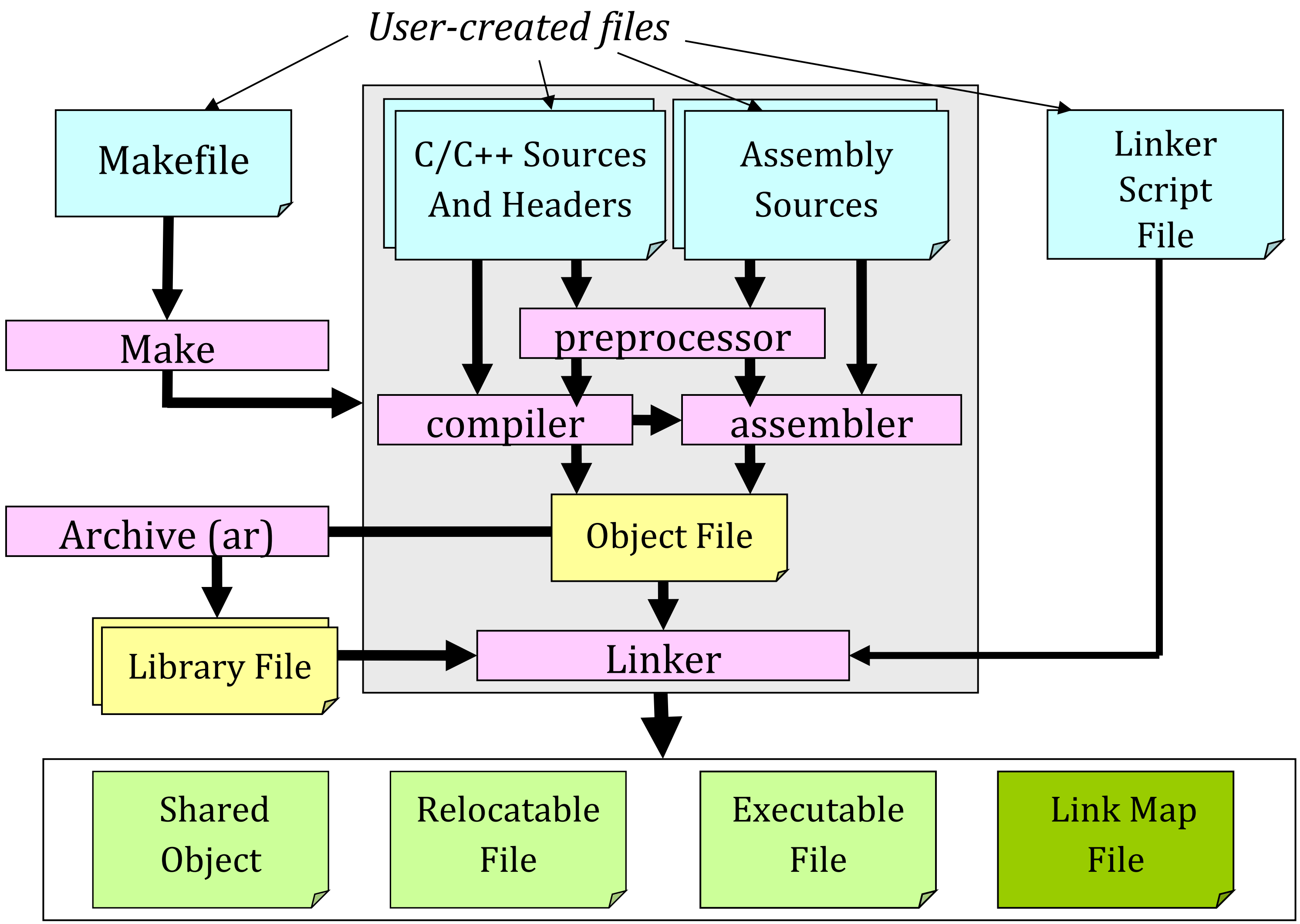
do_execve()

```
  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()

- search_binary_handler():
  - Scans a list of binary file handlers registered in the kernel;
  - If no handler is able to recognize the image format, syscall returs the ENOEXEC error (“Exec Format Error”);
- In fs/binfmt_elf.c:
  - load_elf_binary():
    - Load image file to memory using mmap;
    - Reads the program header and sets permissions accordingly
    - **elf_ex = *((struct elfhdr *)bprm->buf);**

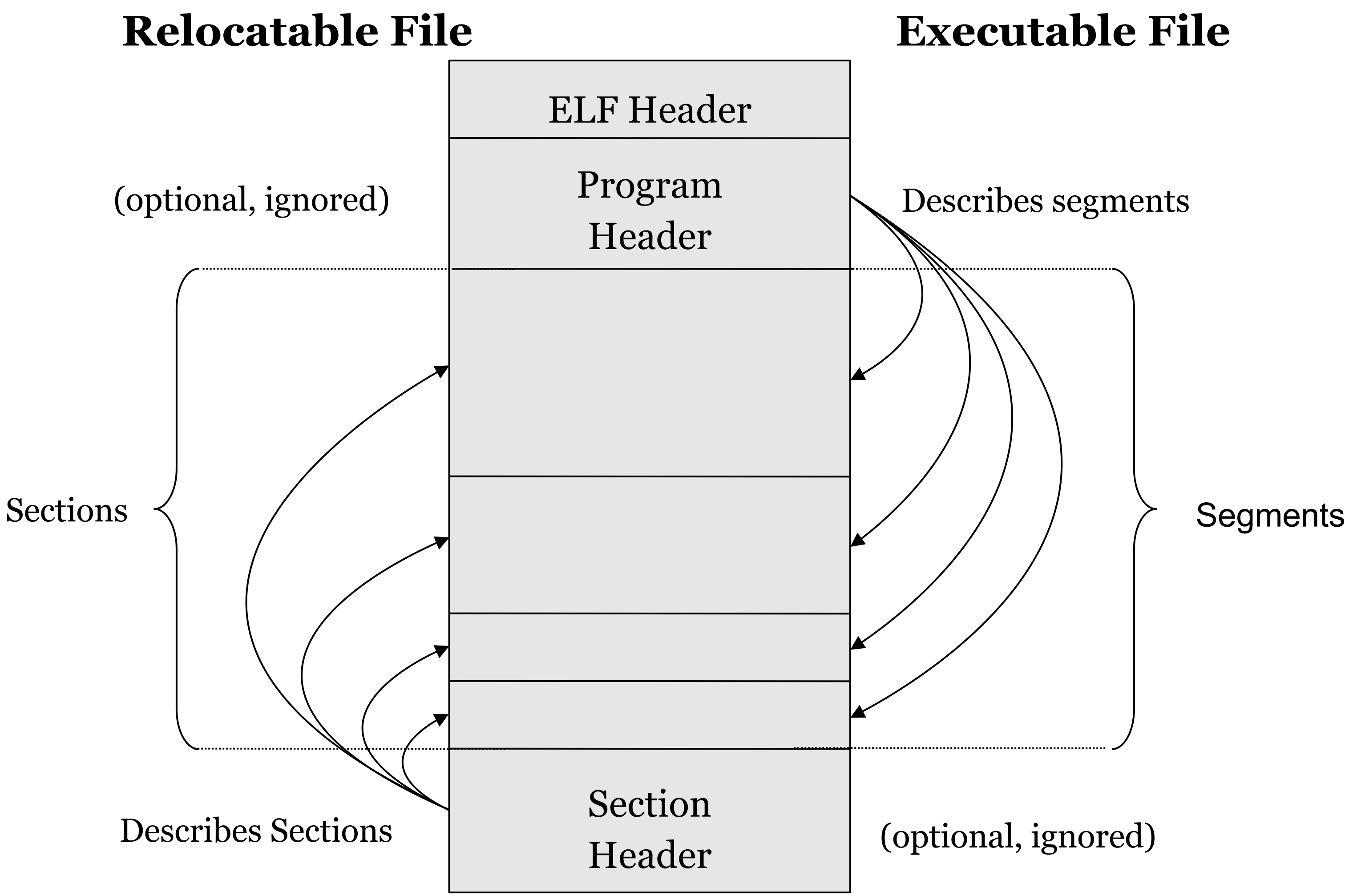
Compiling Process



ELF Types of Files

- 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 File's Structure



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;
```

Types and Flags in Section Header

`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.

`WRITE`: The section contains runtime-writeable data.

`ALLOC`: The section occupies memory at runtime.

`EXECINSTR`: The section contains executable machine instructions.

Some Sections

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

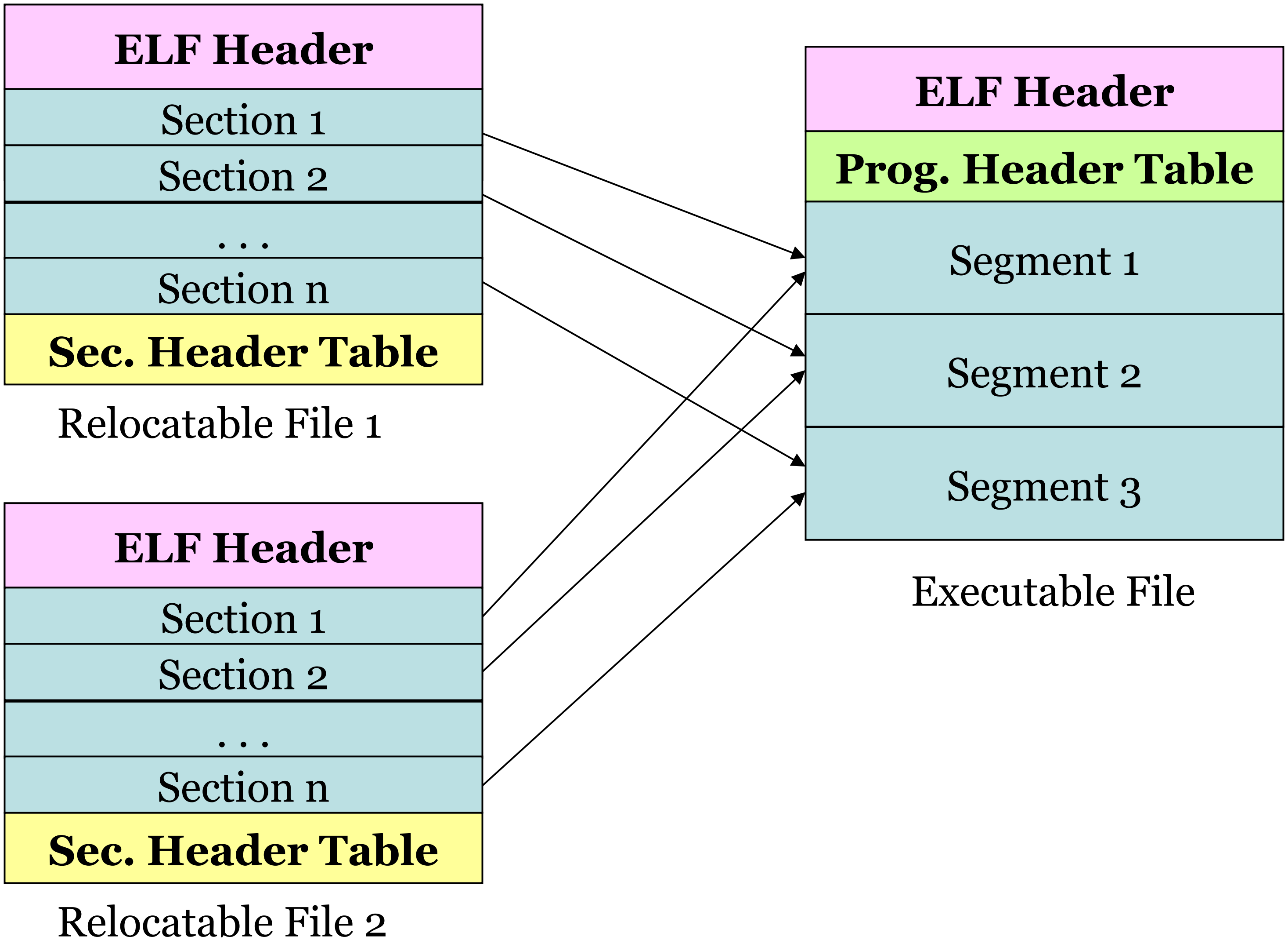
Executable Files

- 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 `ALLOCATE` 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 Data Structures

text section

```
                         ...
1bc1: e8 00 00 00 00   (call ???)
1bc6: 83 c4 10         add $0x10, %rsp
1bc9: a1 00 00 00 00   (movb 0x0, %eax)
                         ...
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
```

symbol table

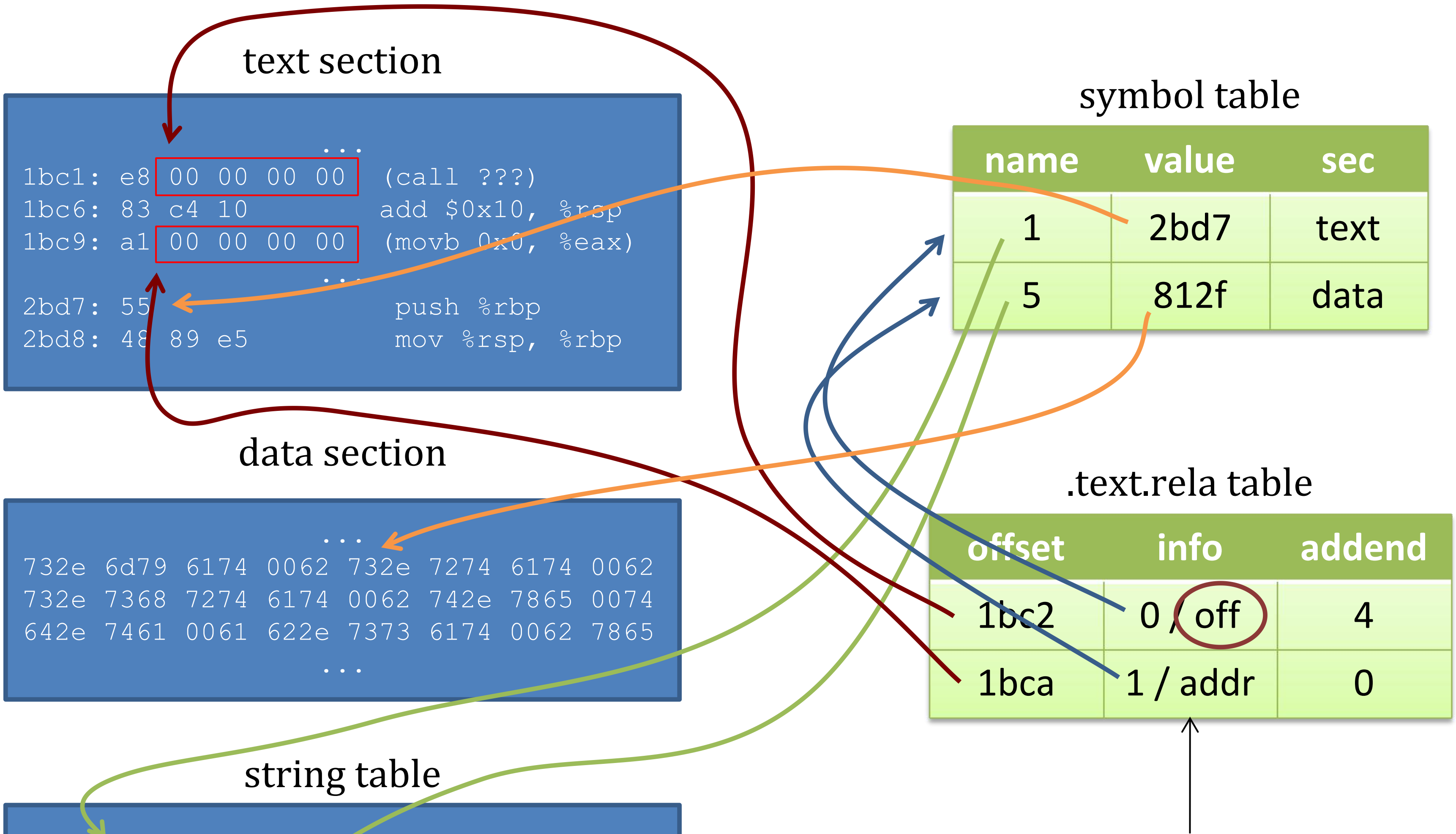
name	value	sec
1	2bd7	text
5	812f	data

.text.rela table

offset	info	addend
1bc2	0 / off	4
1bca	1 / addr	0

This member also tells what kind of relocation should be performed

Static Relocation Data Structures



This member also tells what kind of relocation should be performed

Symbols Visibility

- *weak* symbols:
  - 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
```

Entry Point for the Program

- `main()` is not the actual entry point for the program
- glibc inserts auxiliary functions
  - The actual entry point is called `_start`
- 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 Linker

- 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 Relocation Data Structures

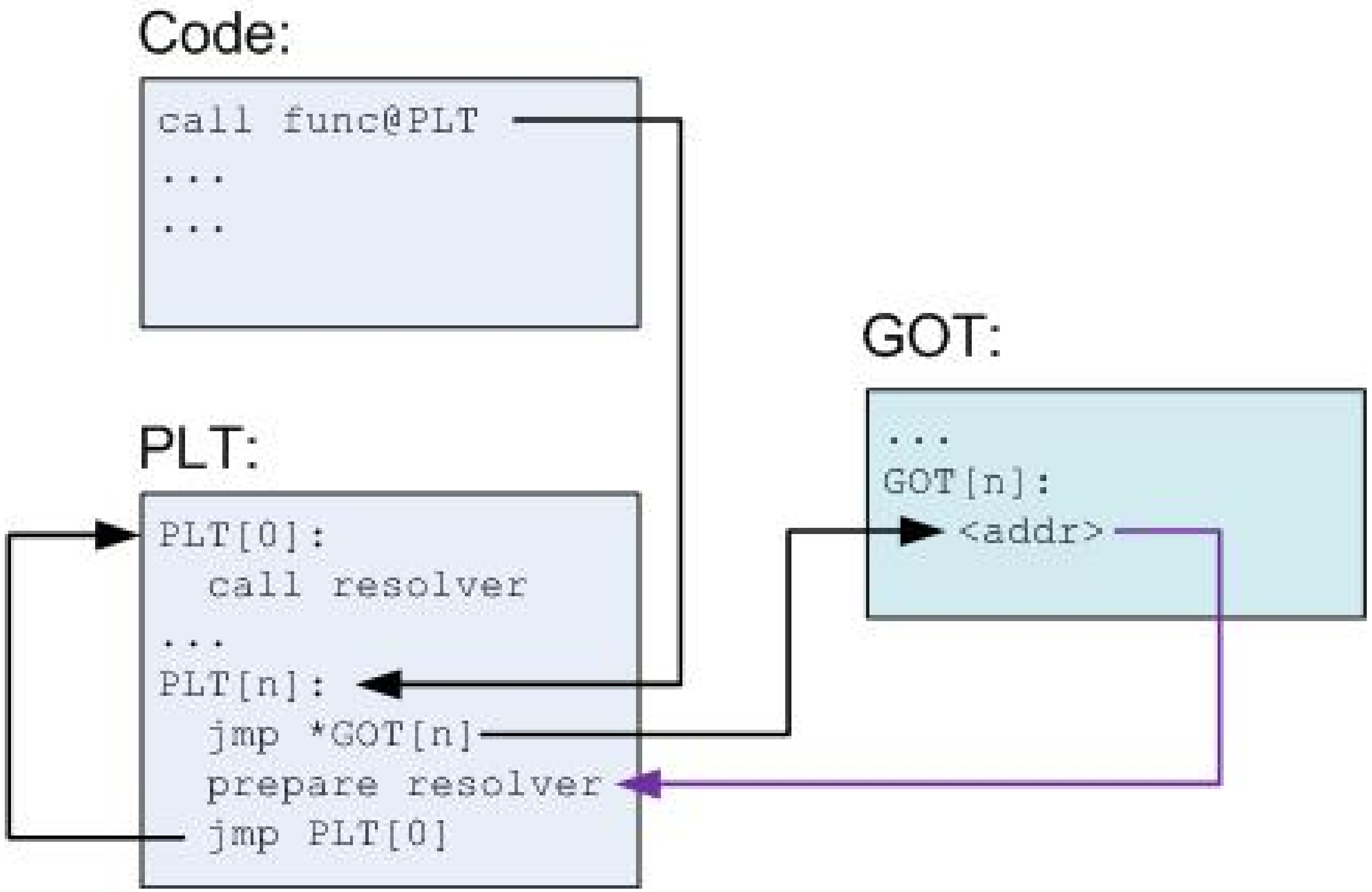
- `.dynsym`: a minimal symbol table used by the dynamic linker when performing relocations
- `.hash`: a hash table that is used to quickly locate a given symbol in the `.dynsym`, usually in one or two tries.
- `.dynstr`: string table related to the symbols stored in `.dynsym`

- These tables are used to populate the GOT table
- This table is populate upon need (*lazy binding*)

Steps to populate the tables

- The first PLT entry is special
- Other entries are identical, one for each function needing resolution.
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a *resolver* routine
  - 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

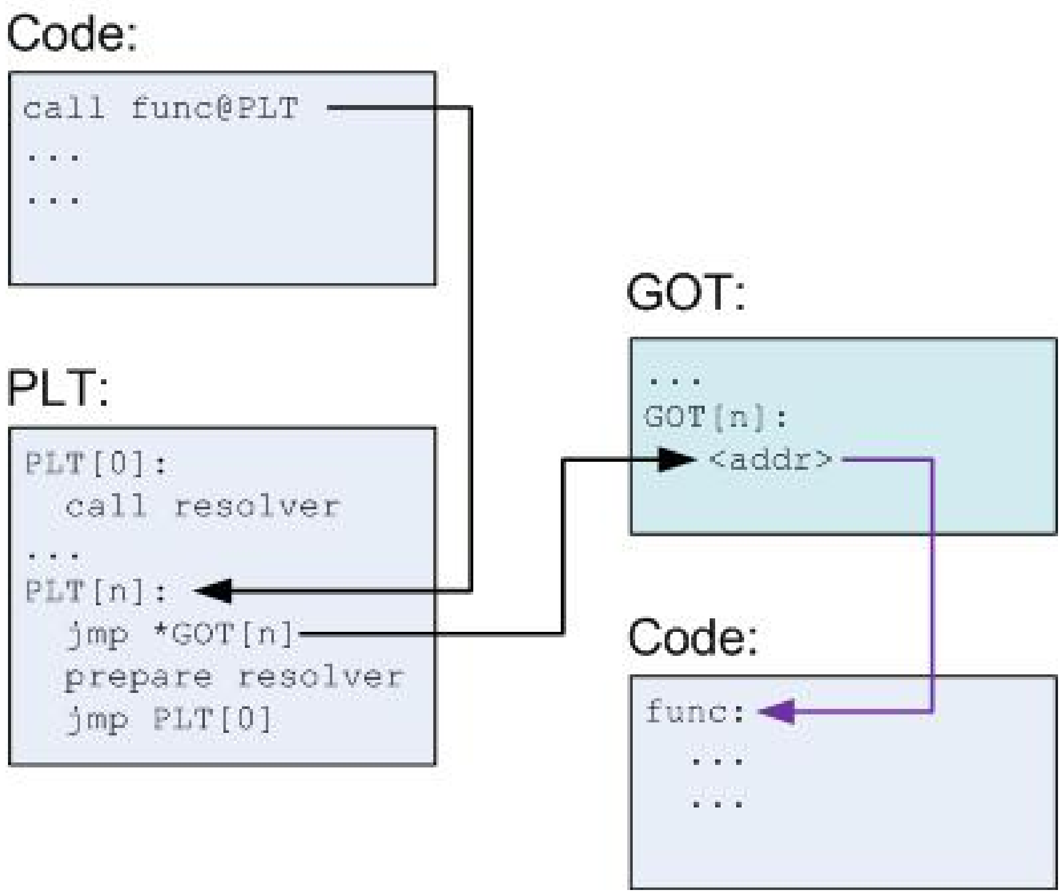
GOT and PLT after library loading



Steps to populate the tables

- When `func` is called for the first time:
  - `PLT[n]` is called, and jumps to the address pointed to it in `GOT[n]`
  - This address points into `PLT[n]` itself, to the preparation of arguments for the resolver.
  - The resolver is then called, by jumping to `PLT[0]`
  - The resolver performs resolution of the actual address of `func`, places its actual address into `GOT[n]` and calls `func`.

GOT and PLT after first call to `func`



Initial steps of the Program's Life

- 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`

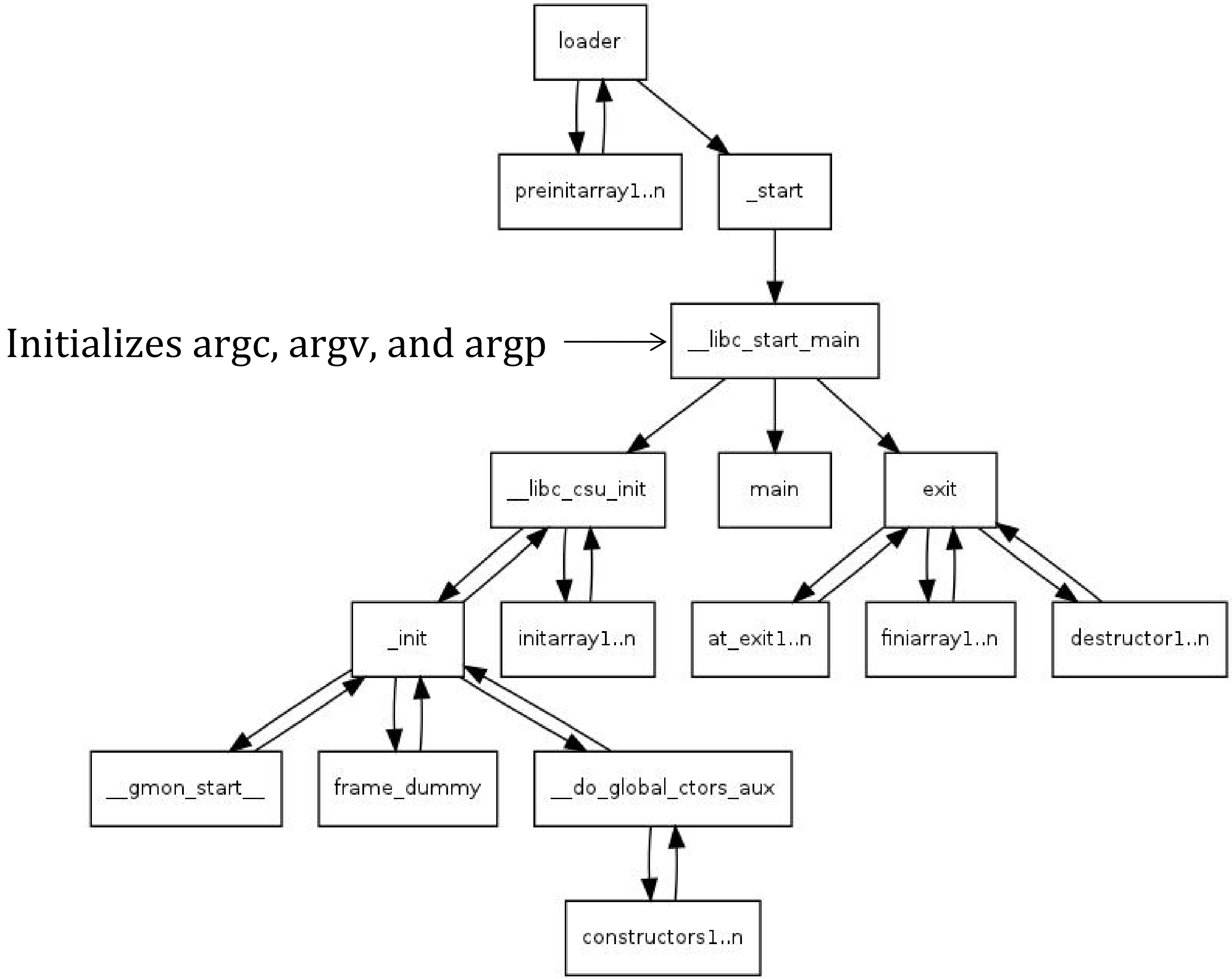
%esp is now pointing at argv. The mov puts argv into %ecx without moving the stack pointer

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 be never executed!

Userspace Life of a Program



Stack Layout at Program Startup

local variables of main saved registers of main	actual main()
return address of main argc argv envp	__libc_start_main()
stack from startup code	
argc argv pointers NULL that ends argv[] environment pointers NULL that ends envp[] ELF Auxiliary Table argv strings environment strings program name NULL	kernel

vDSO is here (→ ELF Auxiliary Table)