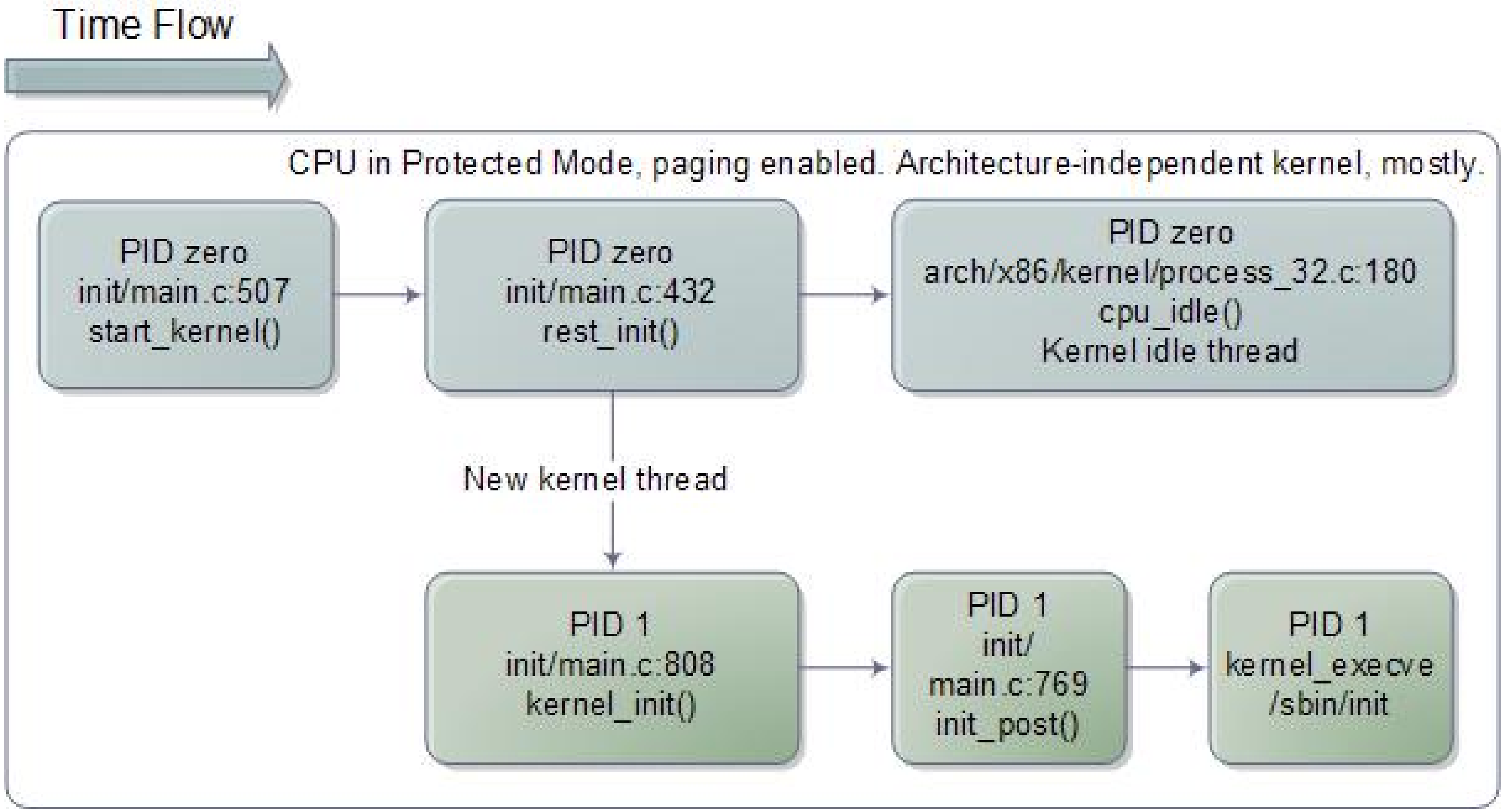
Starting and Managing Userspace Processes

Advanced Operating Systems and Virtualization
Alessandro Pellegrini
A.Y. 2017/2018

Boot Sequence

BIOS/UEFI	The actual Hardware Startup
Bootloader Stage 1	Executes the Stage 2 bootloader (skipped in case of UEFI)
Bootloader Stage 2	Loads and starts the Kernel
Kernel Startup	The Kernel takes control of and initializes the machine (machine-dependent operations)
Init	**First process: basic environment initialization (e.g., SystemV Init, systemd)**
Runlevels/Targets	Initializes the user environment (e.g., single-user mode, multiuser, graphical, ...)

Back to Kernel Initialization



Time Flow

CPU in Protected Mode, paging enabled. Architecture-independent kernel, mostly.

PID zero
init/main.c:507
start_kernel()

PID zero
init/main.c:432
rest_init()

PID zero
arch/x86/kernel/process_32.c:180
cpu_idle()
Kernel idle thread

New kernel thread

PID 1
init/main.c:808
kernel_init()

PID 1
init/
main.c:769
init_post()

PID 1
kernel_execve
/sbin/init

`rest_init()`

- We need to start other processes than idle!
- A new **kernel thread** is created, referencing `kernel_init()` as its entry point
- A call to `schedule()` is issued, to start scheduling the newly-created process
- This is done right before PID 0 calls into `cpu_idle()`

Starting `/sbin/init`

- `/sbin/init` is the first userspace process ever started
- This process is commonly stored into the ramdisk, to speedup the booting process
- init will have to load configuration files from the hard drive
- We have to find out how to allow userspace processes to access, e.g., a disk: VFS

File System: Representations

- In RAM:
  - Partial/full representation of the current structure and content of the File System
- On device:
  - (possibly outdated) representation of the structure and of the content of the File System
- Data access and manipulation:
  - FS-independent part: interface towards other subsystems within the kernel
  - FS-dependent part: data access/manipulation modules targeted at a specific file system type
- In UNIX: "*everything is a file*"

Connecting the two parts

- Any FS object (dir/file/dev) is represented in RAM via specific data structures
- They keep a reference to the code which correctly "speaks" to the actual device, if any
- The reference is accessed using File System independent APIs by other kernel subsystems
- Function pointers are used to reference actual drivers' functions

File System Initialization

- FS initialization takes place in `start_kernel()` according to this sequence:
  - `vfs_caches_init()` (in `fs/dcache.c`)
  - `mnt_init()` (in `fs/namespace.c`)
  - `init_rootfs()` (in `fs/ramfs/inode.c`)
  - `init_mount_tree()` (in `fs/namespace.c`)
- In this way subsystems able to handle FS are setup
- Typically, at least two different FS types are supported:
  - Rootfs (file system in RAM)
  - EXT
- In principle, Linux could be configured to support no FS

File system types

- The `file_system_type` structure describes a file system (it is defined in `include/linux/fs.h`)
- It keeps information related to:
  - The file system name
  - A pointer to a function to be executed upon mounting the file system (superblock-read)

```
struct file_system_type {
   const char *name;
   int fs_flags;
   struct super_block *(*read_super)(struct super_block *,
   void *, int);
   struct module *owner;
   struct file_system_type * next;
   struct list_head fs_supers;
};
```

Declaring and Registering FS Types

- Any kind of File Systems can be linked to another via mountpoints
- Instances must be recognized by the Kernel
- New `file_system_type`s must be declared and registered in the FS table (which is implemented as a list)

```
DECLARE_FSTYPE(var, type, read, flags)
```

(in `include/linux/fs.h`)

```
int register_filesystem(struct file_system_type *)
```

(in `fs/super.c`)

Declaring and Registering Rootfs

- Rootfs is declared statically in `init/do_mounts.c`
  - the variable is `rootfs_fs_type`
- The registration is done by `init_rootfs()`

```
static struct file_system_type rootfs_fs_type = {
        .name       = "rootfs",
        .mount      = rootfs_mount,
        .kill_sb    = kill_litter_super,
    };

    int __init init_rootfs(void)
    {
       return register_filesystem(&rootfs_fs_type);
    }
```

Mounting the Rootfs instance

- This is done in `init_mount_tree()`
- Four different data structures are involved:
  - ➢`struct vfsmount` (in `include/linux/mount.h`)
  - ➢`struct super_block` (in `include/linux/fs.h`)
  - ➢`struct inode` (in `include/linux/fs.h`)
  - ➢`struct dentry` (in `include/linux/dcache.h`)

- `vfsmount` and `struct super_block` keep information on the file system (e.g. in terms of relation with other file systems)
- `struct inode` and `struct dentry` are instantiated for each file/directory in the file system

vfsmount

```
struct vfsmount
{
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent;     /*fs we are mounted on */
    struct dentry *mnt_mountpoint;   /*dentry of mountpoint */
    struct dentry *mnt_root;         /*root of the mounted tree*/
    struct super_block *mnt_sb;      /*pointer to superblock */
    struct list_head mnt_mounts;     /*list of children, anchored
                                                here */
    struct list_head mnt_child;     /*and going through their
                                                mnt_child */
    atomic_t mnt_count;
    int mnt_flags;
    char *mnt_devname;               /* Name of device e.g.
                                           /dev/dsk/hda1 */
    struct list_head mnt_list;
};
```

struct super_block

```
struct super_block {
        struct list_head      s_list;   /* Keep this first */
        ……
        unsigned long         s_blocksize;
        ……
        unsigned long long    s_maxbytes;    /* Max file size */
        struct file_system_type       *s_type;
        struct super_operations       *s_op;
        ……
        struct dentry         *s_root;
        ……
        struct list_head      s_dirty;       /* dirty inodes */
        ……
        union {
                struct minix_sb_info  minix_sb;
                struct ext2_sb_info   ext2_sb;
                struct ext3_sb_info   ext3_sb;
                struct ntfs_sb_info   ntfs_sb;
                struct msdos_sb_info  msdos_sb;
                ……
                void                  *generic_sbp;
        } u;
        ……
};
```

struct dentry

```
struct dentry {
      unsigned int dflags;
      ……
      struct inode  * d_inode;   /* Where the name belongs to */
      struct dentry * d_parent;  /* parent directory */
      struct list_head d_hash;   /* lookup hash list */
      ……
      struct list_head d_child;  /* child of parent list */
      struct list_head d_subdirs;      /* our children */
      ……
      struct qstr d_name;
      ……
      struct lockref d_lockref;  /*per-dentry lock and refcount*/
      struct dentry_operations  *d_op;
      struct super_block * d_sb; /* The root of the dentry tree*/
      ……
      unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

`struct qstr`

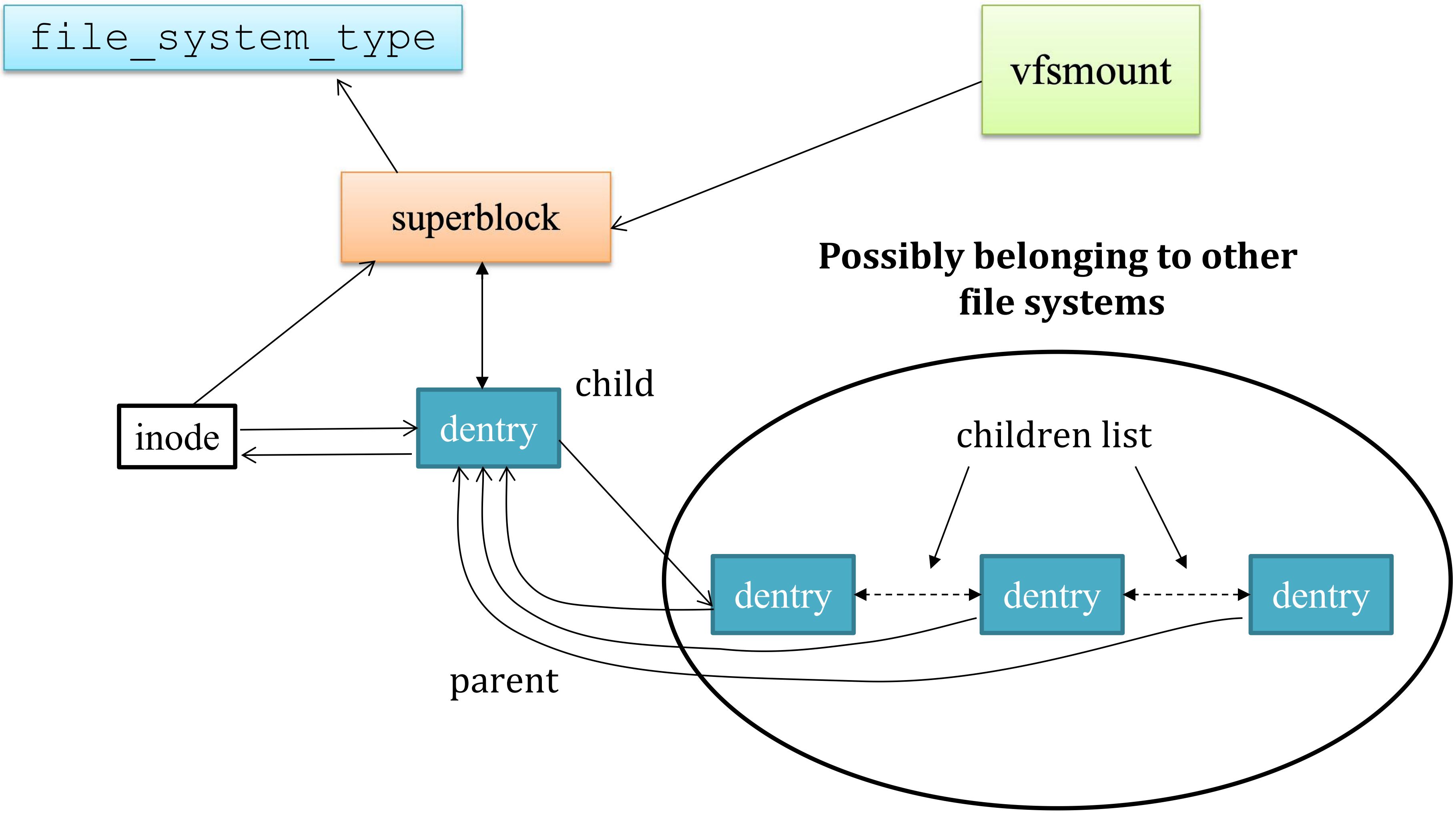
- Eases parameter passing
- Saves "metadata" about the string

```
#define HASH_LEN_DECLARE u32 hash; u32 len
struct qstr {
     union {
          struct {
               HASH_LEN_DECLARE;
          };
          u64 hash_len;
     };
     const unsigned char *name;
};
```

struct inode

```
struct inode {
        ......
        struct list_head      i_dentry;
        ......
        uid_t                 i_uid;
        gid_t                 i_gid;
        ......
        unsigned long         i_blksize;
        unsigned long         i_blocks;
        ......
        struct inode_operations        *i_op;
        struct file_operations         *i_fop;
        struct super_block             *i_sb;
        wait_queue_head_t              i_wait;
        ......
        union {
                ......
                struct ext2_inode_info        ext2_i;
                struct ext3_inode_info        ext3_i;
                ......
                struct socket                 socket_i;
                ......
                void                          *generic_ip;
        } u;
};
```

Global Organization



Initializing the Rootfs instance

- The main tasks, carried out by `init_mount_tree()`, are:
  1. Allocation of the 4 data structures for Rootfs
  2. Linking of the data structures
  3. Setting the name “/” to the root of the file system
  4. Linking the idle process to Rootfs
- The first three are carried out by `vfs_kern_mount()` which executes the super-block read-function for Rootfs
- The last is done via `set_fs_pwd()` and `set_fs_root()`

Initializing the Rootfs instance

```
static void __init init_mount_tree(void)
{
        struct vfsmount *mnt;
        struct mnt_namespace *ns;
        struct path root;
        struct file_system_type *type;

        type = get_fs_type("rootfs");
        if (!type)
                panic("Can't find rootfs type");
        mnt = vfs_kern_mount(type, 0, "rootfs", NULL);
        put_filesystem(type);
        if (IS_ERR(mnt))
                panic("Can't create rootfs");
        .........

        root.mnt = mnt;
        root.dentry = mnt->mnt_root;
        mnt->mnt_flags |= MNT_LOCKED;

        set_fs_pwd(current->fs, &root);
        set_fs_root(current->fs, &root);
}
```

VFS and PCBs

- In the PCB, `struct fs_struct *fs` points to information related to the current directory and the root directory for the associated process
- `fs_struct` is defined in `include/fs_struct.h`

```
struct fs_struct {
    atomic_t count;
    rwlock_t lock;
    int umask;
    struct dentry * root, * pwd, * altroot;
    struct vfsmount * rootmnt, * pwdmnt,
                                  * altrootmnt;
};
```

- The idle has both `root` and `pwd` pointing the only existing dentry (at this point of the boot process)

`fs_struct` in 3.0

```
struct fs_struct {
         int users;
         spinlock_t lock;
         seqcount_t seq;
         int umask;
         int in_exec;
         struct path root, pwd;
 };
```

Superblock operations

- Superblock operations must:
  - Manage statistic of the file system
  - Create and manage i-nodes
  - Flush to the device updated information on the state of the file system
- Some File Systems might not use some operations (think of File Systems in RAM)
- Functions to access statistics takes are invoked by system calls `statfs` and `fstatfs`

struct super_operations

- It is defined in `include/linux/fs.h`

```
struct super_operations {
   struct inode *(*alloc_inode)(struct super_block *sb);
   void (*destroy_inode)(struct inode *);
   void (*read_inode) (struct inode *);
   void (*read_inode2) (struct inode *, void *) ;
   void (*dirty_inode) (struct inode *);
   void (*write_inode) (struct inode *, int);
   void (*put_inode) (struct inode *);
   void (*delete_inode) (struct inode *);
   void (*put_super) (struct super_block *);
   void (*write_super) (struct super_block *);
   int (*sync_fs) (struct super_block *);
   void (*write_super_lockfs) (struct super_block *);
   void (*unlockfs) (struct super_block *);
   int (*statfs) (struct super_block *, struct statfs *);
   ...
};
```

Ramfs Example

- Defined in `fs/ramfs/inode.c` and `fs/libfs.c`

```
int simple_statfs(struct dentry *dentry,
                  struct kstatfs *buf)
{
   buf->f_type = dentry->d_sb->s_magic;
   buf->f_bsize = PAGE_SIZE;
   buf->f_namelen = NAME_MAX;
   return 0;
}


static const struct super_operations ramfs_ops = {
   .statfs            = simple_statfs,
   .drop_inode        = generic_delete_inode,
   .show_options      = ramfs_show_options,
};
```

dentry operations

- They specify non-default operations for manipulating d-entries
- The table maintaining the associated function pointers is defined in `include/linux/dcache.h`
- For the file system in RAM this structure is not used

```
struct dentry_operations {
   int (*d_revalidate)(struct dentry *, int);
   int (*d_hash) (struct dentry *, struct qstr *);
   int (*d_compare) (struct dentry *,
                     struct qstr *, struct qstr *);
   void (*d_delete)(struct dentry *);
   void (*d_release)(struct dentry *);
   void (*d_iput)(struct dentry *, struct inode *);
   ...
};
```

Removes the pointed i-node (when releasing the dentry)

Removes the dentry, when the reference counter is set to zero

i-node operations

- They specify i-node related operations
- The table maintaining the corresponding function pointers is defined in `include/linux/fs.h`

```
struct inode_operations {
  ...
  int (*create) (struct inode *,struct dentry *,int);
  struct dentry * (*lookup) (struct inode *,struct dentry *);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,int);
  int (*rmdir) (struct inode *,struct dentry *);
  int (*mknod) (struct inode *,struct dentry *,int,int);
  ...
};
```

An example for the file system in RAM

- i-node operations for the File System in RAM are defined in `fs/ramfs/inode.c`

```
static struct inode_operations
ramfs_dir_inode_operations = {
    .create =             ramfs_create,
    .lookup =             simple_lookup,
    .link =               simple_link,
    .unlink =             simple_unlink,
    .symlink =            ramfs_symlink,
    .mkdir =              ramfs_mkdir,
    .rmdir =              simple_rmdir,
    .mknod =              ramfs_mknod,
    .rename =             simple_rename,
};
```

struct nameidata

- `struct nameidata` is used in several VFS operations (e.g., to manipulate strings)
- It is defined in `include/linux/fs.h`:

```
struct nameidata {
    struct dentry *dentry;
    struct vfsmount *mnt;
    struct qstr last;
    unsigned int flags;
    int last_type;
};
```

VFS Intermediate Functions

- A set of API which ensure consistency when managing a VFS (e.g., pathname lookup)
- A lot of different locking strategies on all data structures are used

`static int path_lookupat(struct nameidata *nd, unsigned flags, struct path *path)` (in `fs/namei.c`)

  - it is used when an existing object is wanted such as by `stat()` or `chmod()`. It calls `walk_component()` on the final component through a call to `lookup_last()`. `path_lookupat()` returns just the final dentry
  - `LOOKUP_FOLLOW` allows to follow symlinks

VFS Intermediate Functions

`int vfs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)` (in `fs/namei.c`)

- Creates an i-node and associates it with `dentry`. `dir` points to a parent i-node from which basic information for the setup of the child is retrieved. `mode` specifies the access rights for the created object

`static __inline__ struct dentry * dget(struct dentry *dentry)` (in `include/linux/dcache.h`)

- Acquires a dentry (by incrementing the reference counter)

VFS Intermediate Functions

`void dput(struct dentry *dentry)` (in `include/linux/dcache.c`)

- Release a dentry. This will drop the usage count and if appropriate call the dentry unlink function as well as removing it from the queues and releasing its resources. If the parent dentries were scheduled for release they too may now get deleted.

`long do_mount(const char *dev_name, const char __user *dir_name, const char *type_page, unsigned long flags, void *data_page)` (in `fs/namespace.c`)

- Mounts a device onto a target directory

`static struct inode *alloc_inode(struct super_block *sb)` (in `fs/inode.c`)

- allocates an i-node and initializes it according to the specific file system rules

VFS Intermediate Functions

`struct dentry *d_hash_and_lookup(struct dentry *dir, struct qstr *name)` (in `fs/dcache.c`)

- hash the qstr then search for a dentry. `dir` is the directory to search in, `name` is the qstr of name
- Relies on `d_lookup()`

`struct dentry *d_lookup(const struct dentry *parent, const struct qstr *name)` (in `fs/dcache.c`)

- search the children of the `parent` dentry for the `name` in question. If the dentry is found its reference count is incremented and the dentry is returned. The caller must use `dput()` to free the entry when it has finished using it. NULL is returned if the dentry does not exist.

`int vfs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool want_excl)` (in `fs/namei.c`)

- Create an i-node linked to `dentry`, which is child of the i-node pointed by `dir`. The parameter `mode` corresponds to the value of the permission mask passed in input to the open system call. It relies on the i-node-operation `create`

Relations to `kernel_init()`

- Initialization of VFS and execution of init stems from kernel_init() at various points
- `do_basic_setup()`: initializes drivers, also ramfs drivers (after page table init)
- `prepare_namespace()` (in `init/do_mounts.c`):
  - Wait for devices to complete their probing (delays in boot are often caused here)
  - Mounts the `/dev` pseudofolder (devtmpfs)
  - Loads initramfs
  - Mounts initramfs as `"/"`
- `run_init_process()`: invoked multiple times over multiple binaries
  - relies on `do_execve()` in (`fs/exec.c`)

```
if (!try_to_run_init_process("/sbin/init") ||
     !try_to_run_init_process("/etc/init") ||
     !try_to_run_init_process("/bin/init") ||
     !try_to_run_init_process("/bin/sh"))
    return 0;

panic("No working init found.  Try passing init= option to kernel. "
      "See Linux Documentation/admin-guide/init.rst for guidance.");
```

Device Numbers

- Each device is associated with a couple of numbers: MAJOR and MINOR
- MAJOR is the key to access the device driver as registered within a *driver database*
- MINOR identifies the actual instance of the device driven by that driver (this can be specified by the driver programmer)
- There are different tables to register devices, depending on whether the device is a *char device* or a *block device*:
  - `fs/char_dev.c` for char devices
  - `fs/block_dev.c` for block devices
- In the above source files we can also find device-independent functions for accessing the actual driver

Identifying Char and Block Devices

```
$ ls -l /dev/sda /dev/ttyS0
brw-rw---- 1 root disk 8,  0  9 apr 09.31 /dev/sda
crw-rw---- 1 root uucp 4, 64  9 apr 09.31 /dev/ttyS0
```

type

major

minor

Major and Minor Numbers

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8, 0  9 apr 09.31 /dev/sda
brw-rw---- 1 root disk 8, 1  9 apr 09.31 /dev/sda1
brw-rw---- 1 root disk 8, 2  9 apr 09.31 /dev/sda2
```

Same driver, different disks or partitions

- The same major can be given to both a character and a block device!
- Numbers are "assigned" by the Linux Assigned Names and Numbers Authority (http://lanana.org/) and kept in `Documentation/devices.txt`.
- Defines are in `include/uapi/linux/major.h`

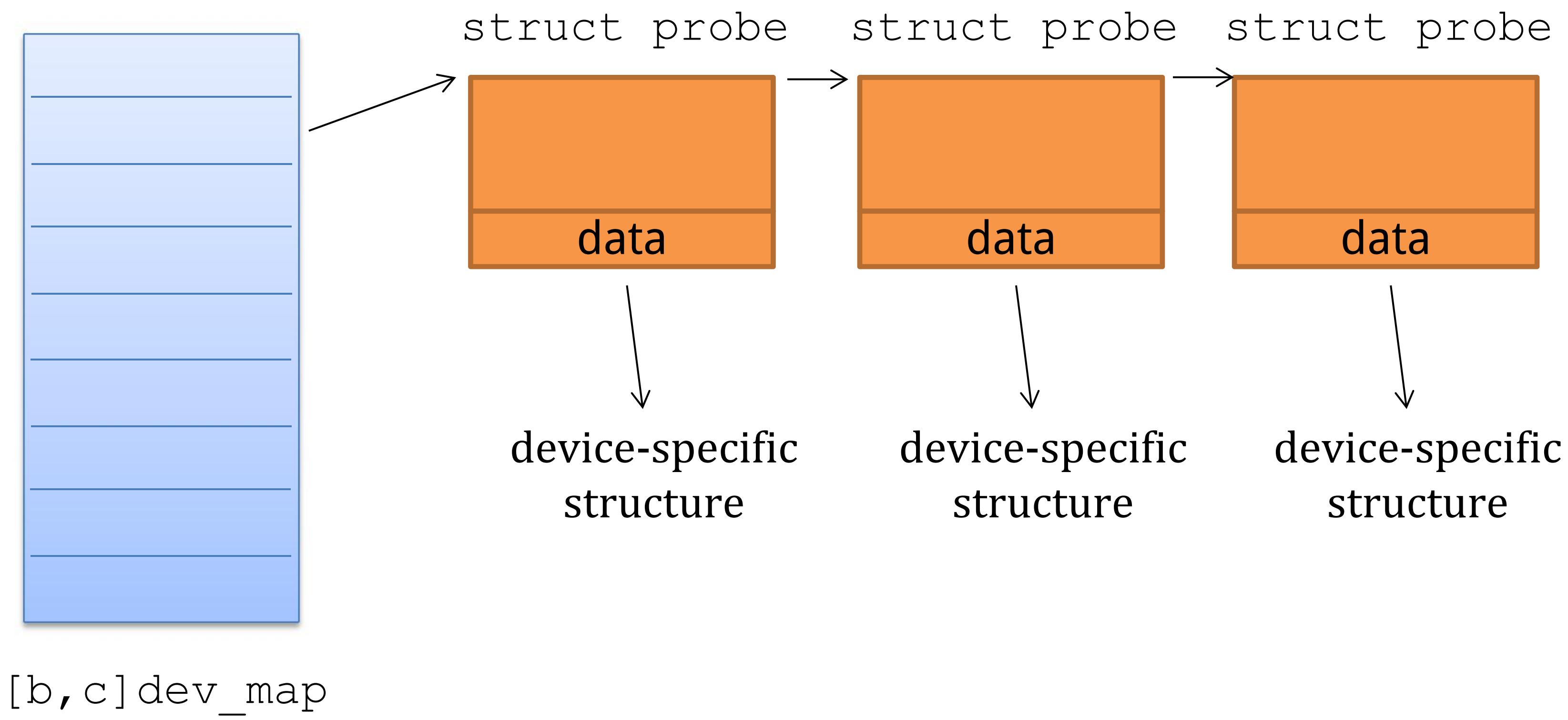
The Device Database

- Char and Block devices behave differently, but they are organized in identical databases which are handled as hashmaps
- They are referenced as `cdev_map` and `bdev_map`

```
struct kobj_map {
     struct probe {
          struct probe *next;
          dev_t dev;
          unsigned long range;
          struct module *owner;
          kobj_probe_t *get;
          int (*lock)(dev_t, void *);
          void *data;
     } *probes[255];
     struct mutex *lock;
};
```

hasing is done as:
major % 255

The Device Database



Device Numbers Representation

- The `dev_t` type keeps both the major and the minor (in `include/linux/types.h`)

```
typedef __u32 __kernel_dev_t;
typedef __kernel_dev_t          dev_t;
```

- In `linux/kdev_t.h` we find facilities to manipulate it:

```
#define MINORBITS 20
#define MINORMASK       ((1U << MINORBITS) - 1)
#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

struct cdev

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
} __randomize_layout;
```

Char Devices Range Database

- Defined in `fs/char_dev.c`
- Used to manage device number allocation to drivers

```
#define CHRDEV_MAJOR_HASH_SIZE 255
static struct char_device_struct {
      struct char_device_struct *next;
      unsigned int major;
      unsigned int baseminor;
      int minorct;
      char name[64];
      struct cdev *cdev;
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

Registering Char Devices

- `linux/fs.h` provides the following wappers to register/deregister a driver:
  - `int register_chrdev(unsigned int major, const char *name, `**`struct file_operations`**` *fops)`: registration takes place onto the entry at displacement MAJOR (0 means the choice is up o the kernel). The actual MAJOR number is returned
  - `int unregister_chrdev(unsigned int major, const char *name)`: releases the entry at displacement MAJOR
- They map to actual operations in `fs/char_dev.c`:
  - `int __register_chrdev(unsigned int major, unsigned int baseminor, unsigned int count, const char *name, const `**`struct file_operations`**` *fops)`
  - `void __unregister_chrdev(unsigned int major, unsigned int baseminor, unsigned int count, const char *name)`

struct file_operations

- It is defined in `include/linux/fs.h`

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode*, struct file *, unsigned int,
                       unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        ...
};
```

Registering Device Numbers

- A driver might require to *register* or *allocate* a range of device numbers
- API are in `fs/char_dev.c` and exposed in `include/linux/fs.h`
- `int register_chrdev_region(dev_t from, unsigned count, const char *name)`
  - Major is specified in `from`
- `int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)`
  - Major and first minor are returned in `dev`

Block Devices

- The structure corresponding to cdev for a block device is `struct gendisk` in `include/linux/genhd.h`

```
struct gendisk {
      int major;          /* major number of driver */
      int first_minor;
      int minors;         /* maximum number of minors, =1 for
                           * disks that can't be partitioned. */
      char disk_name[DISK_NAME_LEN];/* name of majordriver */
      ...
      const struct block_device_operations *fops;
      struct request_queue *queue;
};
```

- In `block/genhd.c` we find the following functions to register/deregister the driver:

```
int register_blkdev(unsigned int major,           const
char * name, struct    block_device_operations *bdops)

int unregister_blkdev(unsigned int major, const char * name)
```

`struct block_device_operations`

- It is defined in `include/linux/fs.h`

```
struct block_device_operations {
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   int (*ioctl) (struct inode *, struct file *,
                 unsigned, unsigned long);
   int (*check_media_change) (kdev_t);
   int (*revalidate) (kdev_t);
   struct module *owner;
};
```

- There is nothing here to read and write from the device!

Read/Write on Block Devices

- For char devices the management of read/write operations is in charge of the device driver
- This is not the same for block devices
- read/write operations on block devices are handled via a single API related to buffer cache operations
- The actual implementation of the buffer cache policy will determine the real execution activities for block device read/write operations

Request Queues

- Request queues (strategies in UNIX) are the way to operate on block devices
- Requests encapsulate optimizations to manage each specific device (e.g. via the *elevator algorithm*)
- The Request Interface is associated with a queue of pending requests towards the block device

Linking Devices and the VFS

- The member `umode_t i_mode` in `struct inode` tells the type of the i-node:
  - directory
  - file
  - char device
  - block device
  - (named) pipe
- The kernel function `sys_mknod()` creates a generic i-node
- If the i-inode represents a device, the operations to manage the device are retrieved via the device driver database
- In particular, the i-node has the `dev_t i_rdev` member

The `mknod()` System Call

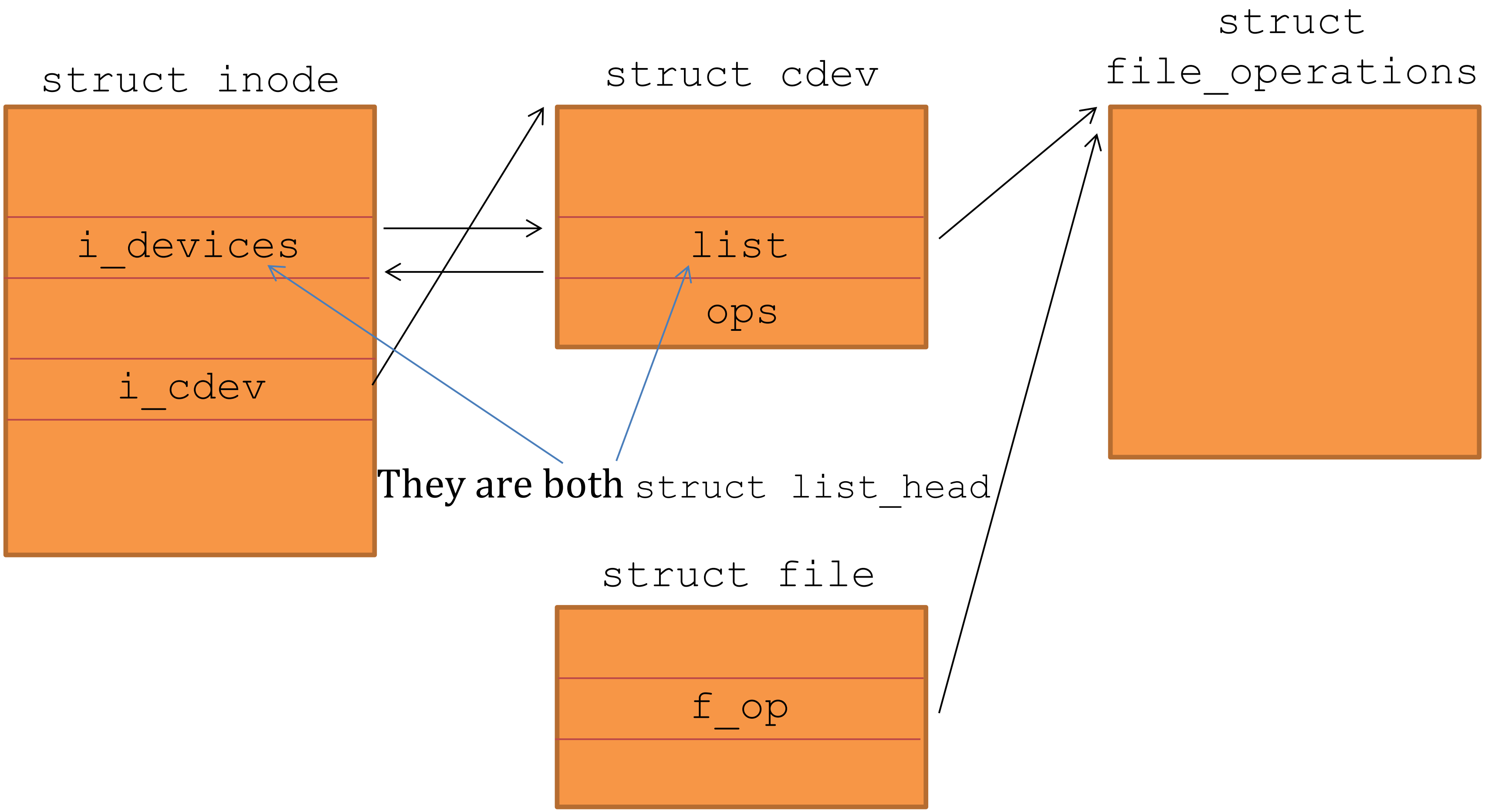
```
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

- `mode` specifies permissions and type of node to be created
- Permissions are filtered via the `umask` of the calling process (`mode & umask`)
- Different macros can be used to define the node type: `S_IFREG`, `S_IFCHR`, `S_IFBLK`, `S_IFIFO`
- When using `S_IFCHR` or `S_IFBLK`, the parameter `dev` specifies Major and Minor numbers of the device file to create, otherwise it is a don't care

Opening Device Files

- In `fs/devices.c` there is the generic `chrdev_open()` function
- This function needs to find the dev-specific file operations
- Given the device, number, `kobject_lookup()` is called to find a corresponding `kobject`
- From the `kobject` we can navigate to the corresponding `cdev`
- The device-dependent file operations are then in `cdev->ops`
- This information is then cached in the i-node

i-node to File Operations Mapping



They are both `struct list_head`

The `mount()` system call

```
int  mount(const  char  *source, const char *target,
const char *filesystemtype, unsigned long mountflags,
const void *data);
```

- `MS_NOEXEC`: Do not allow programs to be executed from this file system.
- `MS_NOSUID`: Do not honour set-UID and set-GID bits when executing programs from this file system.
- `MS_RDONLY`: Mount file system read-only.
- `MS_REMOUNT`: Remount an existing mount. This allows you to change the mountflags and data of an existing mount without having to unmount and remount the file system. `source` and `target` should be the same values specified in the initial `mount()` call; filesystem type is ignored.
- `MS_SYNCHRONOUS`: Make writes on this file system synchronous (as though the `O_SYNC` flag to `open(2)` was specified for all file opens to this file system).

Mount Points

- Directories selected as the target for the mount operation become a “mount point”
- This is reflected in `struct dentry` by setting in `d_flags` the flag `DCACHE_MOUNTED`
- Any path lookup function ignores the content of mount points (namely the name of the dentry) while performing pattern matching

File descriptor table

- The PCB has a member `struct files_struct *files` which points to the descriptor table defined in `include/linux/fdtable.h`:

```
struct files_struct {
        atomic_t count;
        bool resize_in_progress;
        wait_queue_head_t resize_wait;

        struct fdtable __rcu *fdt;
        struct fdtable fdtab;

        spinlock_t file_lock ____cacheline_aligned_in_smp;
        unsigned int next_fd;
        unsigned long close_on_exec_init[1];
        unsigned long open_fds_init[1];
        unsigned long full_fds_bits_init[1];
        struct file __rcu *fd_array[NR_OPEN_DEFAULT];
};
```

struct fdtable

```
struct fdtable {
    unsigned int max_fds;
    struct file __rcu **fd
    unsigned long *close_on_exec;
    unsigned long *open_fds;
    unsigned long *full_fds_bits;
    struct rcu_head rcu;
};
```

bitmaps

struct file

```
struct file {
     struct path              f_path;
     struct inode             *f_inode;
     const struct file_operations  *f_op;
     spinlock_t               f_lock;
     atomic_long_t            f_count;
     unsigned int             f_flags;
     fmode_t                  f_mode;
     struct mutex             f_pos_lock;
     loff_t                   f_pos;
     struct fown_struct       f_owner;
     const struct cred        *f_cred;
     ...
     struct address_space     *f_mapping;
     errseq_t            f_wb_err;
}
```

Opening a file

- `do_sys_open()` in `fs/open.c` is logically divided in two parts:
  - First, a file descriptor is allocated (and a suitable `struct file` is allocated)
  - The second relies on an invocation of the intermediate function `struct file *do_filp_open(int dfd, struct filename *pathname, const struct open_flags *op)` which returns the address of the `struct file` associated with the opened file

`filp_open()`

- Two main tasks are executed here:
  1. Setup of the current namei data (and later restore)
  2. Navigation of the FS tree to create a `struct file` for the working session on the file
- The first task is carried out via the function `set_nameidata` defined in `fs/namei.c` which:
  - Sets up a `struct nameidata` data structure and links that to the PCB of the running process
  - This structure keeps information about the current path and relations to other elements in the VFS
- The second task exploits the `path_openat()` function in `fs/namei.c`

`path_openat()`

- Get a zeroed free file descriptor via `get_empty_filp()` (returns a pointer to a struct file taken from the slab)
- Initializes internal data structures related to the file path via `init_path()`
- Performs name resolution mapping the path to the actual dentry representing the file
  - `static int link_path_walk(const char *name, struct nameidata *nd):`
  - `nd` will reference the dentry

do_sys_open()

```
long do_sys_open(int dfd, const char __user *filename,
int flags, umode_t mode) {
        struct filename *tmp;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}
```

get_unused_fd_flags()

```
int __alloc_fd(struct files_struct *files, unsigned start,
unsigned end, unsigned flags) {
        unsigned int fd;
        int error;
        struct fdtable *fdt;

        spin_lock(&files->file_lock);
repeat:
        fdt = files_fdtable(files);
        fd = start;
        if (fd < files->next_fd)
                fd = files->next_fd;
        if (fd < fdt->max_fds)
                fd = find_next_fd(fdt, fd);

        error = -EMFILE;
        if (fd >= end)
                goto out;

        error = expand_files(files, fd);
        if (error < 0)
                goto out;

        if (error) /* fdt expansion is blocking */
                goto repeat;
        ...
}
```

Kernel Pointers and Errors

- From include/linux/err.h

```
#define IS_ERR_VALUE(x) unlikely((unsigned long)(void
*)(x) >= (unsigned long)-MAX_ERRNO)

static inline void * __must_check ERR_PTR(long error) {
        return (void *) error;
}

static inline long __must_check PTR_ERR(__force const
void *ptr) {
        return (long) ptr;
}

static inline bool __must_check IS_ERR(__force const void
*ptr) {
        return IS_ERR_VALUE((unsigned long)ptr);
}
```

Closing a file

- The close() system call is defined in `fs/open.c` as:
  - `SYSCALL_DEFINE1(close, unsigned int, fd)`
- This function basically calls (in `fs/file.c`):

```
int __close_fd(struct files_struct *files,
                              unsigned fd)
```

- `__close_fd()`:
  - Closes the file descriptor by calling into `__put_unused_fd();`
  - Calls `filp_close(struct file *filp, fl_owner_t id)`, defined in `fs/open.c`, which flushing the data structures associated with the file (struct file, dentry and i-node)

`filp_close()`

- This relies on the following internal functions:

`void dnotify_flush(struct file *filp, fl_owner_t id)`
(in `fs/dnotify.c`)

Notifies that file flushing has been finalized so that dentry and i-node operations can be carried out

`void locks_remove_posix(struct file *filp,`
`fl_owner_t owner)` (in `fs/locks.c`)

Removes the lock on `struct file`

`void fput(struct file * file)` (in `fs/file_table.c`)

deallocates `struct file`

__close_fd()

```
int __close_fd(struct files_struct *files, unsigned fd)
{
        struct file *file;
        struct fdtable *fdt;

        spin_lock(&files->file_lock);
        fdt = files_fdtable(files);
        if (fd >= fdt->max_fds)
                goto out_unlock;
        file = fdt->fd[fd];
        if (!file)
                goto out_unlock;
        rcu_assign_pointer(fdt->fd[fd], NULL);
        __put_unused_fd(files, fd);
        spin_unlock(&files->file_lock);
        return filp_close(file, files);

out_unlock:
        spin_unlock(&files->file_lock);
        return -EBADF;
}
```

__put_unused_fd()

```
static void __put_unused_fd(struct files_struct *files,
unsigned int fd) {
      struct fdtable *fdt = files_fdtable(files);
      __clear_open_fd(fd, fdt);
      if (fd < files->next_fd)
            files->next_fd = fd;
}


static inline void __clear_open_fd(unsigned int fd,
struct fdtable *fdt) {
      __clear_bit(fd, fdt->open_fds);
      __clear_bit(fd / BITS_PER_LONG, fdt->full_fds_bits);
}
```

The `write()` system call

- Defined in `fs/read_write.c`

```
SYSCALL_DEFINE3(write, unsigned int fd, const char __user
*, buf, size_t, count) {
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;
        if (f.file) {
                loff_t pos = file_pos_read(f.file);
                ret = vfs_write(f.file, buf, count, &pos);
                if (ret >= 0)
                        file_pos_write(f.file, pos);
                fdput_pos(f);
        }
        return ret;
}
```

vfs_write()

- Performs some security checks and then calls:

```
ssize_t __vfs_write(struct file *file, const char __user *p,
size_t count, loff_t *pos) {
      if (file->f_op->write)
            return file->f_op->write(file, p, count, pos);
      else if (file->f_op->write_iter)
            return new_sync_write(file, p, count, pos);
      else
            return -EINVAL;
}
```

The `read()` system call

- Defined in `fs/read_write.c`

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *,
buf, size_t, count) {
      struct fd f = fdget_pos(fd);
      ssize_t ret = -EBADF;

      if (f.file) {
            loff_t pos = file_pos_read(f.file);
            ret = vfs_read(f.file, buf, count, &pos);
            if (ret >= 0)
                  file_pos_write(f.file, pos);
            fdput_pos(f);
      }
      return ret;
}
```

vfs_read()

- Performs some security checks and then calls:

```
ssize_t __vfs_read(struct file *file, char __user *buf,
size_t count, loff_t *pos) {
        if (file->f_op->read)
                return file->f_op->read(file, buf, count, pos);
        else if (file->f_op->read_iter)
                return new_sync_read(file, buf, count, pos);
        else
                return -EINVAL;
}
```

proc File System

- An in-memory file system which provides information on:
  - Active programs (processes)
  - The whole memory content
  - Kernel-level settings (e.g. the currently mounted modules)
- Common files on proc are:
  - `cpuinfo` contains the information established by the kernel about the processor at boot time, e.g., the type of processor, including variant and features.
  - `kcore` contains the entire RAM contents as seen by the kernel.
  - `meminfo` contains information about the memory usage, how much of the available RAM and swap space are in use and how the kernel is using them.
  - `version` contains the kernel version information that lists the version number, when it was compiled and who compiled it.

proc File System

- `net/` is a directory containing network information.
- `net/dev` contains a list of the network devices that are compiled into the kernel. For each device there are statistics on the number of packets that have been transmitted and received.
- `net/route` contains the routing table that is used for routing packets on the network.
- `net/snmp` contains statistics on the higher levels of the network protocol.
- `self/` contains information about the current process. The contents are the same as those in the per-process information described later.

proc File System

- `pid/` contains information about process number *pid*. The kernel maintains a directory containing process information for each process.
- `pid/cmdline` contains the command that was used to start the process (using null characters to separate arguments).
- `pid/cwd` contains a link to the current working directory of the process.
- `pid/environ` contains a list of the environment variables that the process has available.
- `pid/exe` contains a link to the program that is running in the process.
- `pid/fd/` is a directory containing a link to each of the files that the process has open.
- `pid/mem` contains the memory contents of the process.
- `pid/stat` contains process status information.
- `pid/statm` contains process memory usage information.

proc Features

- The `file_system_type` is defined in fs/proc/root.c

```
static struct file_system_type proc_fs_type
= {
        .name              = "proc",
        .mount             = proc_mount,
        .kill_sb           = proc_kill_sb,
        .fs_flags          = FS_USERNS_MOUNT,
};
```

- This is a single-instance memory-mapped File System

Creation of the proc instance

- Done in `proc_root_init()`
- The File System is registered using `register_filesystem()`
- Subfolders are created (such as `net`, `sys`, `sys/fs`)
  - This is done using `proc_mkdir()`

Core data structures for proc

- proc is represented using the data structure defined in `fs/proc/internal.h`

```
   struct proc_dir_entry {
      unsigned short low_ino;
      unsigned short namelen;
      const char *name;
      mode_t mode;
      nlink_t nlink;    uid_t uid;    gid_t gid;
      unsigned long size;
      struct inode_operations * proc_iops;
      struct file_operations * proc_fops;
      ...
      read_proc_t *read_proc;
      write_proc_t *write_proc;
      ...
};
```

Mounting proc

- proc is mounted only if asked at compile-time in makeconfig (see the macro `CONFIG_PROC_FS`)
- This File System is mounted by init
- proc can be remounted in userspace *namespaces*

Handling proc (`include/linux/proc_fs.h`)

```
struct proc_dir_entry *proc_mkdir(const char *name, struct
proc_dir_entry *parent)
```

- Creates a directory called `name` within the directory pointed by `parent`. Returns the pointer to the new `struct proc_dir_entry`

```
static inline struct proc_dir_entry
*create_proc_read_entry(const char *name, mode_t mode, struct
proc_dir_entry *base, read_proc_t *read_proc, void * data)
```

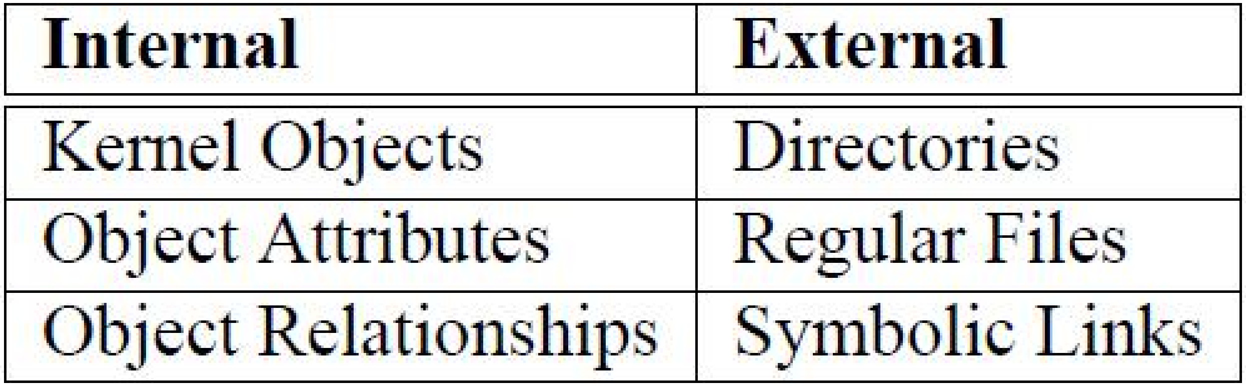
- Creates a node called `name`, with type and permissions `mode`, linked to `base`, and where the reading function is set to `read_proc` end the data field to `data`. It returns the pointer to the new `struct proc_dir_entry`

```
struct proc_dir_entry *create_proc_entry(const char *name,
mode_t mode,  struct proc_dir_entry *parent)
```

- Creates a node called `name`, with type and permissions `mode`, linked to `parent.` It returns the pointer to the new `struct proc_dir_entry`

The Sysfs File System (since 2.6)

- Similar in spirit to proc, mounted to `/sys`
- It is an alternative way to make the kernel export information (or set it) via common I/O operations
- Very simple API
- More clear structuring



Internal	**External**
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

Sysfs Core API

```
int sysfs_create_file(struct kobject *, const struct attribute *);

void sysfs_remove_file(struct kobject *, const struct attribute *);

int sysfs_update_file(struct kobject *, const struct attribute *);
```

```
struct attribute {
    char              *name;
    struct module     *owner;
    mode_t            mode;
};
```

The owner field may be set by the caller to point to the module in which the code to manipulate the attribute exists

Kernel Objects (*knobs*)

- Kobjects don't live on their own: they are embedded into objects (think of `struct cdev`)
- They keep a reference counter (`kref`)

```
void kobject_init(struct kobject *kobj);
int kobject_set_name(struct kobject *kobj,
const char *format, ...);
struct kobject *kobject_get(struct kobject
*kobj);
void kobject_put(struct kobject *kobj);
```

struct kobject

```
struct kobject {
     const char          *name;
     struct list_head    entry;
     struct kobject      *parent;
     struct kset         *kset;
     struct kobj_type    *ktype;
     struct kernfs_node *sd; /* sysfs
                         directory entry */
     struct kref         kref;
};
```

`struct kobj_type`

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

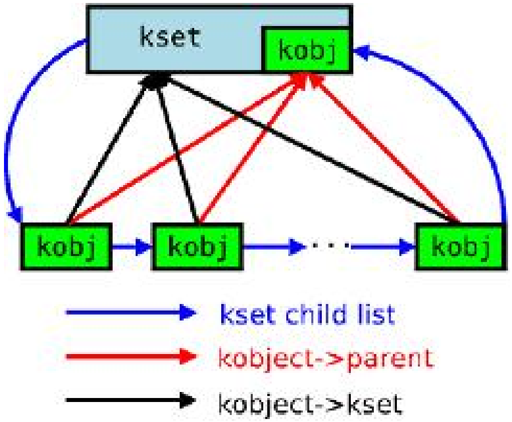
- A specific object type is defined in terms of the `sysfs_ops` to be executed on it, the defaul attributes (if any), and the `release` function

Sysfs Read/Write Operations

- These operations are define in the kobject thanks to the `struct kobj_type *ktype` member:
  - `struct kobject->ktype->sysfs_ops`

```
struct sysfs_ops {
        /* method invoked on read of a sysfs file */
        ssize_t (*show) (struct kobject *kobj,
                         struct attribute *attr,
                         char *buffer);

        /* method invoked on write of a sysfs file */
        ssize_t (*store) (struct kobject *kobj,
                          struct attribute *attr,
                          const char *buffer,
                          size_t size);
};
```

ksets



```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);
kobject_set_name(my_set->kobj, "The name");
```

Hooking into Sysfs

- When a kobject is created it does not immediately appear in Sysfs
- It has to be explicitly added (although the operation can fail):
  - `int kobject_add(struct kobject *kobj);`
- To remove a kobject from Sysfs:
  - `void kobject_del(struct kobject *kobj);`

Device Classes

- Devices are organized into "classes"
- A device can belong to multiple classes
- Class membership is shown in `/sys/class/`
  - Block devices are automatically placed under the "block" class
  - This is done automatically whe the gendisk structure is registered in the kernel
- Most devices don't require the creation of new classes

Managing New Classes

- Manage classes, we instantiate and register the struct class declared in `linux/device.h`

```
static struct class sbd_class = {
      .name = "class_name",
      .class_release = release_fn
};


int class_register(struct class *cls);
void class_destroy(struct class *cls);


struct class *class_create(struct module *owner, const
char *name, struct lock_class_key *key)
```

Managing Devices in Classes

- `struct device *device_create(struct class *class, struct device *parent, dev_t devt, void *drvdata, const char *fmt, ...)`

printf-like way to specify the device node in `/dev`

- `void device_destroy(struct class *class, dev_t devt)`

Device Class Attributes

- Specify attributes for the classes, and functions to "read" and "write" the specific class attributes
- `CLASS_DEVICE_ATTR(name, mode, show, store);`
- This is expanded to a structure called `dev_attr_name`

- `ssize_t (*show)(struct class_device *cd, char *buf);`
- `ssize_t (*store)(struct class_device *, const char *buf, size_t count);`

Creating Device Attribute Files

- Again placed in `/sys`

- `int device_create_file(struct device *dev,const struct device_attribute *attr)`

- `void device_remove_file(struct device *dev, const struct device_attribute *attr)`

udev

- udev is the userspace Linux device manager
- It manages device nodes in `/dev`
- It also handles userspace events raised when devices are added/removed to/from the system

- The introduction of udev has been due to the degree of complexity associated with device management
- It is highly configurable and rule-based

udev rules

- Udev in userspace looks at /sys to detect changes and see whether new (virtual) devices are plugged
- Special rule files (in /etc/udev/rules.d) match changes and create files in /dev accordingly
- Syntax tokens in syntax files:
  - KERNEL: match against the kernel name for the device
  - SUBSYSTEM: match against the subsystem of the device
  - DRIVER: match against the name of the driver backing the device
  - NAME: the name that shall be used for the device node
  - SYMLINK: a list of symbolic links which act as alternative names for the device node

- `KERNEL=="hdb", DRIVER=="ide-disk", NAME="my_spare_disk", SYMLINK+="sparedisk"`

Boot Sequence

BIOS/UEFI	The actual Hardware Startup
Bootloader Stage 1	Executes the Stage 2 bootloader (skipped in case of UEFI)
Bootloader Stage 2	Loads and starts the Kernel
Kernel Startup	The Kernel takes control of and initializes the machine (machine-dependent operations)
Init	First process: basic environment initialization (e.g., SystemV Init, systemd)
Runlevels/Targets	Initializes the user environment (e.g., single-user mode, multiuser, graphical, ...)

Startup Services

- Hostname
- Timezone
- Check the hard drives
- Mount the hard drives
- Remove files from /tmp
- Configure network interfaces
- Start daemons and network services

Startup Run Levels

Level	**Mode**
1 (S)	Single user
2	Multiuser (no networking)
3	Full Multiuser
4	Unused
5	X11
6	Reboot
0	Halt

Run Level Scripts

- Actual scripts placed in: `/etc/rc.d/init.d/`
- `/etc/rc.d/rc#.d/`:
  - Symbolic links to `/etc/init.d` scripts
  - S## - Start scripts
  - K## - Stop scripts
  - `/etc/sysconfig/`: script configuration files

- `chkconfig <script> on|off`
- `service <script> start|stop|restart`

`/etc/inittab`

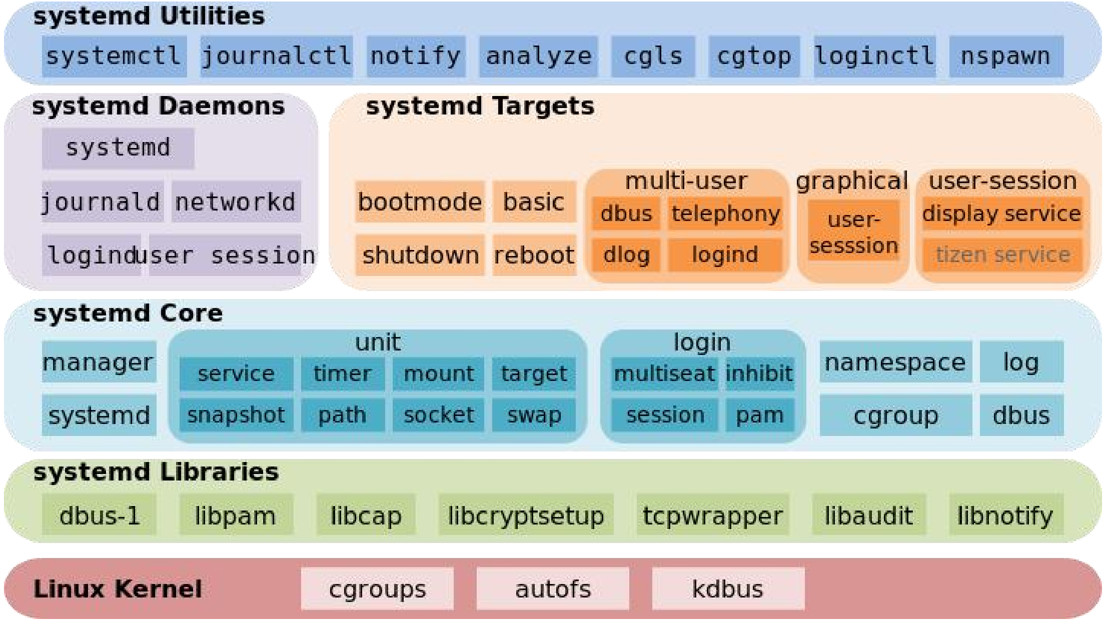
- Initializes system for use
- Format: `id:rl:action:process`
  - `id`: uniquely identifies entry
  - `rl`: what runlevels the entry applies to
  - `action`: the type of action to execute
  - `process`: process command line

- An example:

```
2:23:respawn:/sbin/getty 38400 tty2
```

Systemd

- Becoming more prevalent in Linux Distros
- Mostly compatible with the init system
  - init scripts could be read as alternative format
- Based on the notion of "units" and "dependencies"



systemd Utilities
systemctl journalctl notify analyze cgls cgtop loginctl nspawn

systemd Daemons
systemd
journald networkd
logind user session

systemd Targets
bootmode basic
shutdown reboot
multi-user: dbus telephony dlog logind
graphical: user-sesssion
user-session: display service tizen service

systemd Core
manager
systemd
unit: service timer mount target snapshot path socket swap
login: multiseat inhibit session pam
namespace log
cgroup dbus

systemd Libraries
dbus-1 libpam libcap libcryptsetup tcpwrapper libaudit libnotify

Linux Kernel
cgroups autofs kdbus

Systemd Targets

- The concept of "runlevel" is mapped to "targets" in systemd jargon
- Runlevel is defined through a symbolic to one of the runlevel targets
- Runlevel Target
  - Runlevel 3: /lib/systemd/system/multi-user.target
  - Runlevel 5: /lib/systemd/system/graphical.target
- Change Runlevel:
  - Remove current link `/etc/systemd/system/default.target`
  - Add a new link to the desired runlevel

Systemd Unit Types

- Different unit types control different aspects of the operating system
    - service: handles daemons
    - socket: handles network sockets
    - target: logical grouping of units (example: runlevel)
    - device: expose kernel devices
    - mount: controls mount points of the files system
    - automount: mounts the file system
    - snapshot: references other units (similar to targets)

Systemd Unit Section

- [Unit]
  - Description: A meaningful description of the unit
  - Requires: Configures dependencies on other units
  - Wants: Configures weaker dependencies
  - Conflicts: Negative dependencies
  - Before: This unit must be started before these others
  - After: This unit must be started after these others (unlike Requires, it doest not start the unit if not already active)

Systemd Service Section

- [Service]
  - Type= simple|oneshot|forking|dbus|notify|idle
  - ExecStart
  - ExecReload
  - ExecStop
  - Restart=no|on-success|on-failure|on-abort|always

Systemd Install Section

- [Install]
  - Wantedby=

- Used to determine when to start
  (e.g. Runlevel)

An Example

```
[Unit]
Description=Postfix Mail Transport Agent
After=syslog.target network.target
Conflicts=sendmail.service exim.service

[Service]
Type=forking
PIDFile=/var/spool/postfix/pid/master.pid
EnvironmentFile=-/etc/sysconfig/network
ExecStartPre=-/usr/libexec/postfix/aliasesdb
ExecStartPre=-/usr/libexec/postfix/chroot-update
ExecStart=/usr/sbin/postfix start
ExecReload=/usr/sbin/postfix reload
ExecStop=/usr/sbin/postfix stop

[Install]
WantedBy=multi-user.target
```

Boot Sequence

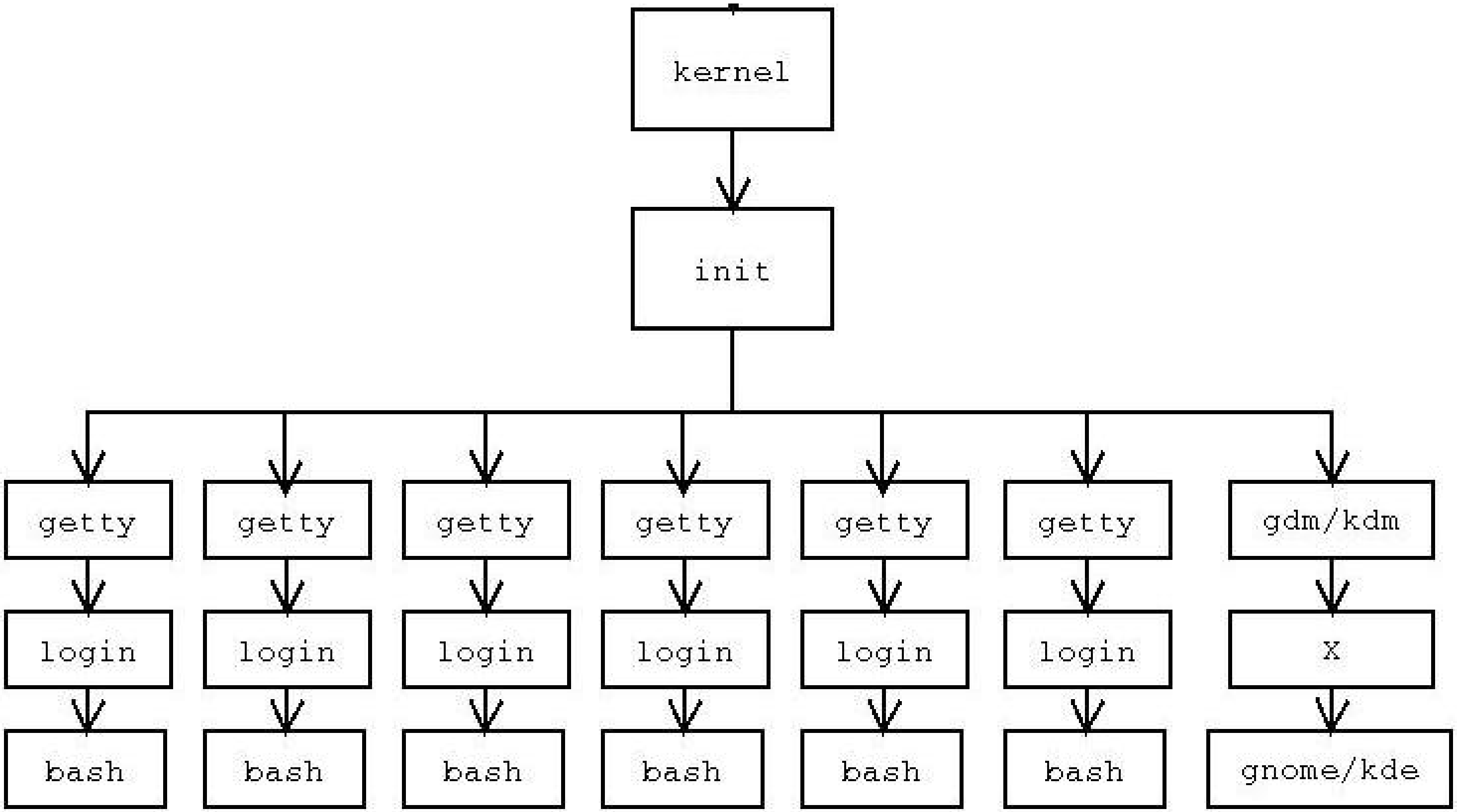
BIOS/UEFI	The actual Hardware Startup
Bootloader Stage 1	Executes the Stage 2 bootloader (skipped in case of UEFI)
Bootloader Stage 2	Loads and starts the Kernel
Kernel Startup	The Kernel takes control of and initializes the machine (machine-dependent operations)
Init	First process: basic environment initialization (e.g., SystemV Init, systemd)
Runlevels/Targets	Initializes the user environment (e.g., single-user mode, multiuser, graphical, ...)

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;
  }

  if (pid < 0) {
      sys_error ("fork");
      throw_to_top_level ();
  }

  if (pid == 0) {
      sigprocmask (SIG_SETMASK, &top_level_mask, (sigset_t *)NULL);
  } else {
      last_made_pid = pid;
      add_pid (pid, async_p);
  }
  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);
}
```

`fork()` and `exec*()`

- 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

`do_fork()`

- Fresh PCB/kernel-stack allocation
- Copy/setup of PCB information
- Copy/setup of PCB linked 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()`
- Admissible 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

`exec*()`

- `exec*()` does not create a new process
- it just 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;
```

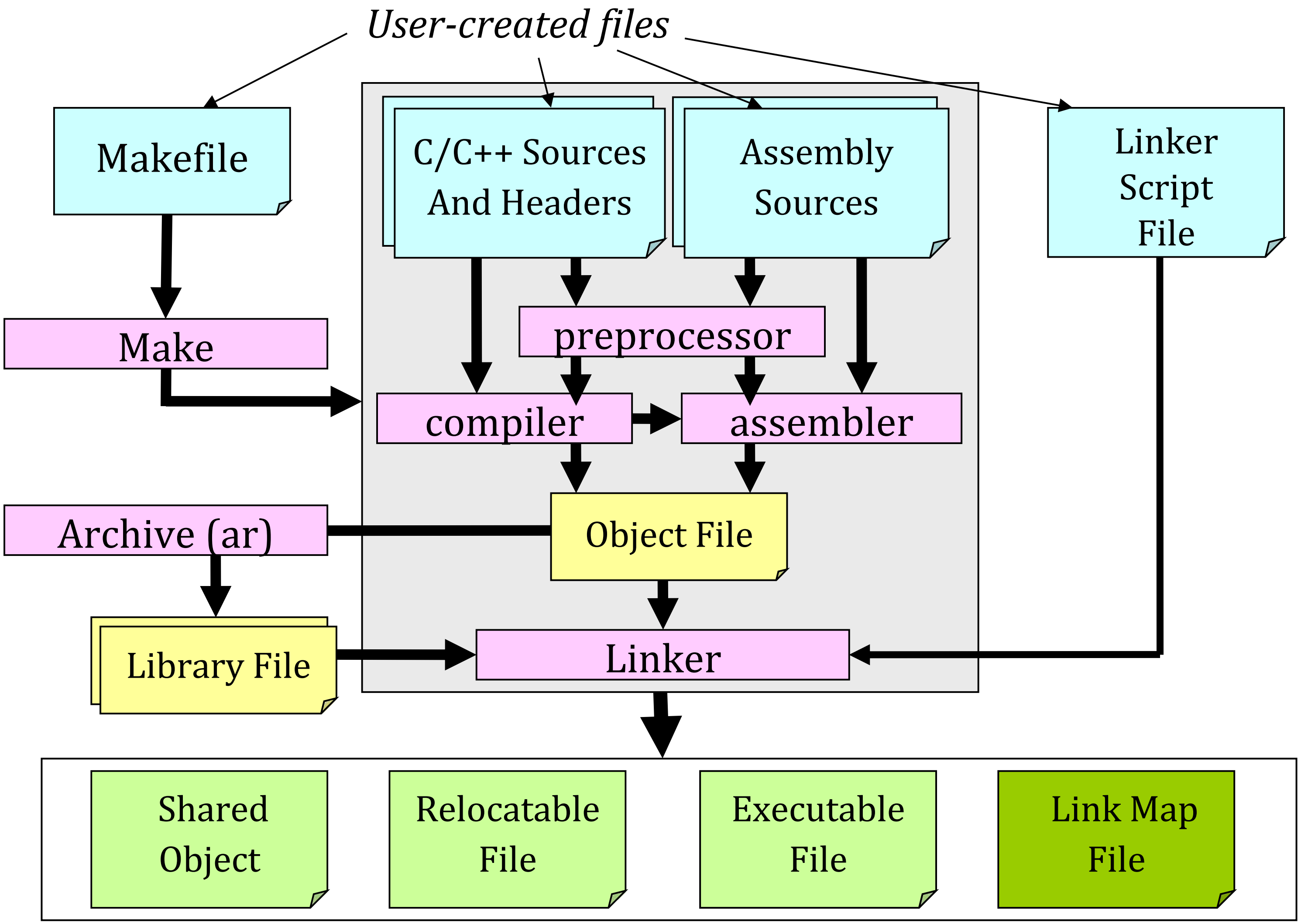
do_execve()

```
  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



Object File Format

- For more than 20 years, *nix executable file format has been `a.out` (since 1975 to 1998).
- This format was made up of at most 7 sections:
  - *exec header*: loading information;
  - *text segment*: machine instructions;
  - *data segment*: initialized data;
  - *text relocations*: information to update pointers;
  - *data relocations*: information to update pointers;
  - *symbol table*: information on variables and functions;
  - *string table*: names associated with symbols.

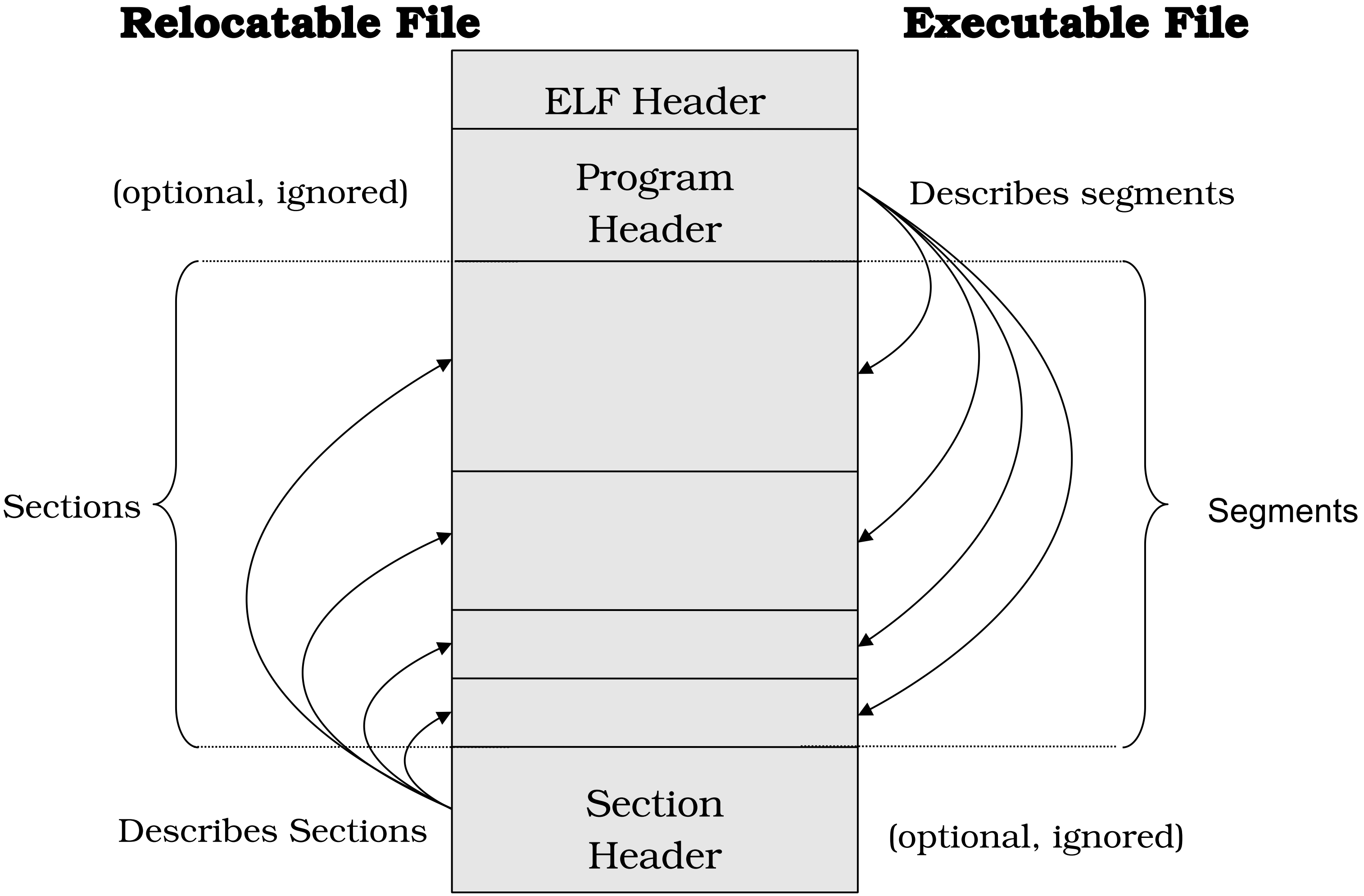
Object File Format

- This format's limits were:
  - cross-compiling;
  - dynamic linking;
  - creation of simple shared libraries;
  - lack for support of initializers/finalizers (e.g. constructors and destructors).
- Linux has definitively replaced `a.out` with ELF (Executable and Linkable Format) in version 1.2 (more or less in 1995).

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

Executable File

ELF Header

(optional, ignored)

Program Header

Describes segments

Sections

Segments

Describes Sections

Section Header

(optional, ignored)

ELF Header

```
#define EI_NIDENT (16)

typedef struct {
  unsigned char e_ident[EI_NIDENT];/* Magic number and other info */
  Elf32_Half    e_type;      /* Object file type */
  Elf32_Half    e_machine;   /* Architecture */
  Elf32_Word    e_version;   /* Object file version */
  Elf32_Addr    e_entry;     /* Entry point virtual address */
  Elf32_Off     e_phoff;     /* Program header table file offset */
  Elf32_Off     e_shoff;     /* Section header table file offset */
  Elf32_Word    e_flags;     /* Processor-specific flags */
  Elf32_Half    e_ehsize;    /* ELF header size in bytes */
  Elf32_Half    e_phentsize; /* Program header table entry size */
  Elf32_Half    e_phnum;     /* Program header table entry count */
  Elf32_Half    e_shentsize; /* Section header table entry size */
  Elf32_Half    e_shnum;     /* Section header table entry count */
  Elf32_Half    e_shstrndx;  /* Section header string table index */
} Elf32_Ehdr;
```

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`

String Table

- Sections keeping string tables contain sequence of null-terminated strings.
- Object files use a string table to represent symbols' and sections' names.
- A string is referenced using an index in the table.
- Symbol table and symbol names are separated because there is no limit in names' length in C/C++

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Index	String
0	*none*
1	name.
7	Variable
11	able
16	able
24	*null string*

Symbol Table

- The Symbol Table keeps in an object file the information necessary to identify and relocate symbolic definitions in a program and its references.

```
typedef struct {
  Elf32_Word     st_name;    /* Symbol name  */
  Elf32_Addr     st_value;   /* Symbol value */
  Elf32_Word     st_size;    /* Symbol size */
  unsigned char st_info;    /* Symbol binding */
  unsigned char st_other;   /* Symbol visibility */
  Elf32_Section st_shndx;   /* Section index */
} Elf32_Sym;
```

Static Relocation Table

- Relocation is the process which connects references to symbols with definition of symbols.
- Relocatable files must keep information on how to modify the contents of sections.

```
typedef struct {
  Elf32_Addr    r_offset; /* Address */
  Elf32_Word   r_info;   /* Relocation type and symbol index */
} Elf32_Rel;


typedef struct {
  Elf32_Addr    r_offset; /* Address */
  Elf32_Word    r_info;   /* Relocation type and symbol index */
  Elf32_Sword   r_addend; /* Addend */
} Elf32_Rela;
```

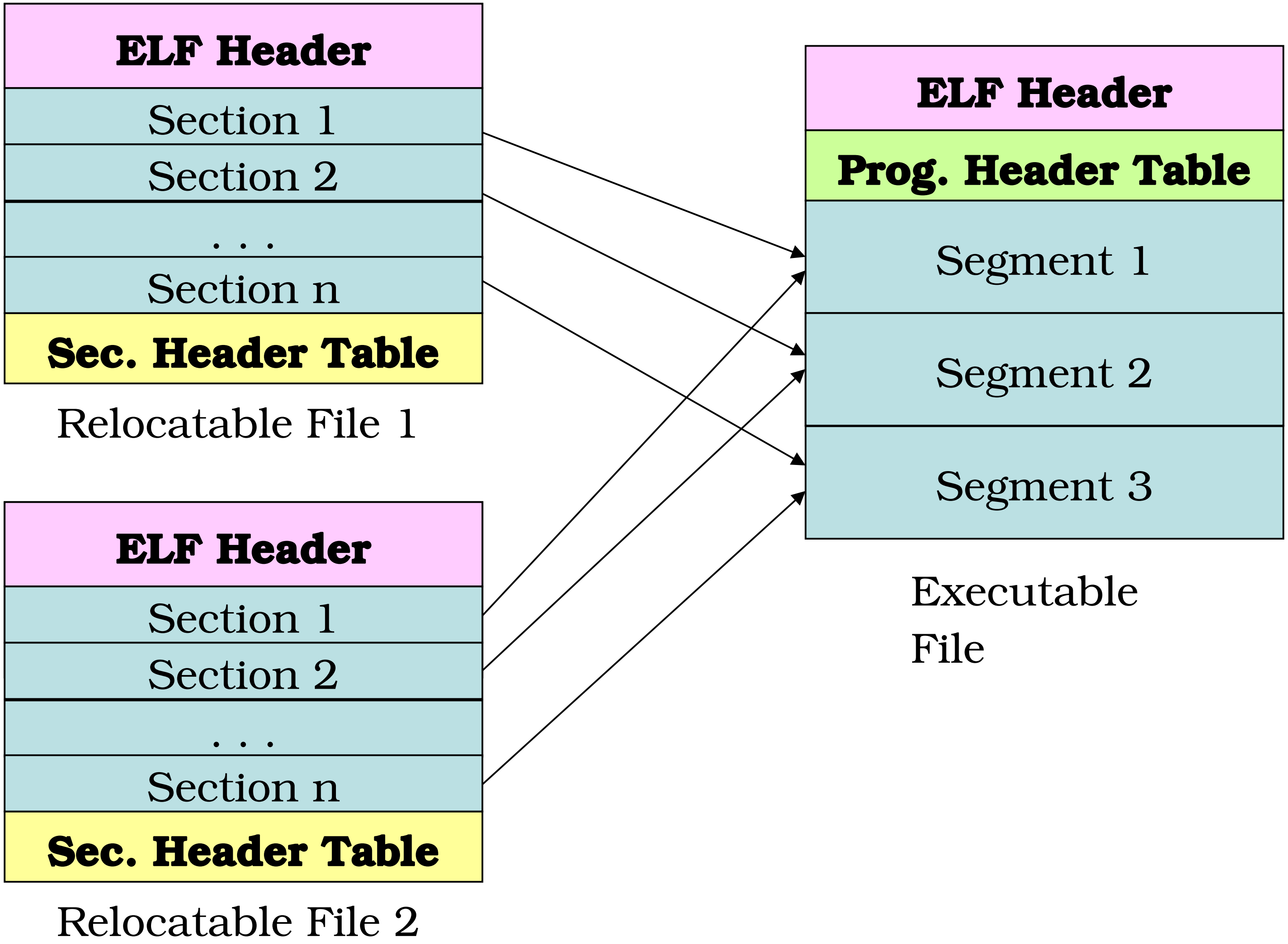
Executable Files

- 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


ELF Header

Section 1

Section 2

. . .

Section n

Sec. Header Table

Relocatable File 1

ELF Header

Section 1

Section 2

. . .

Section n

Sec. Header Table

Relocatable File 2

ELF Header

Prog. Header Table

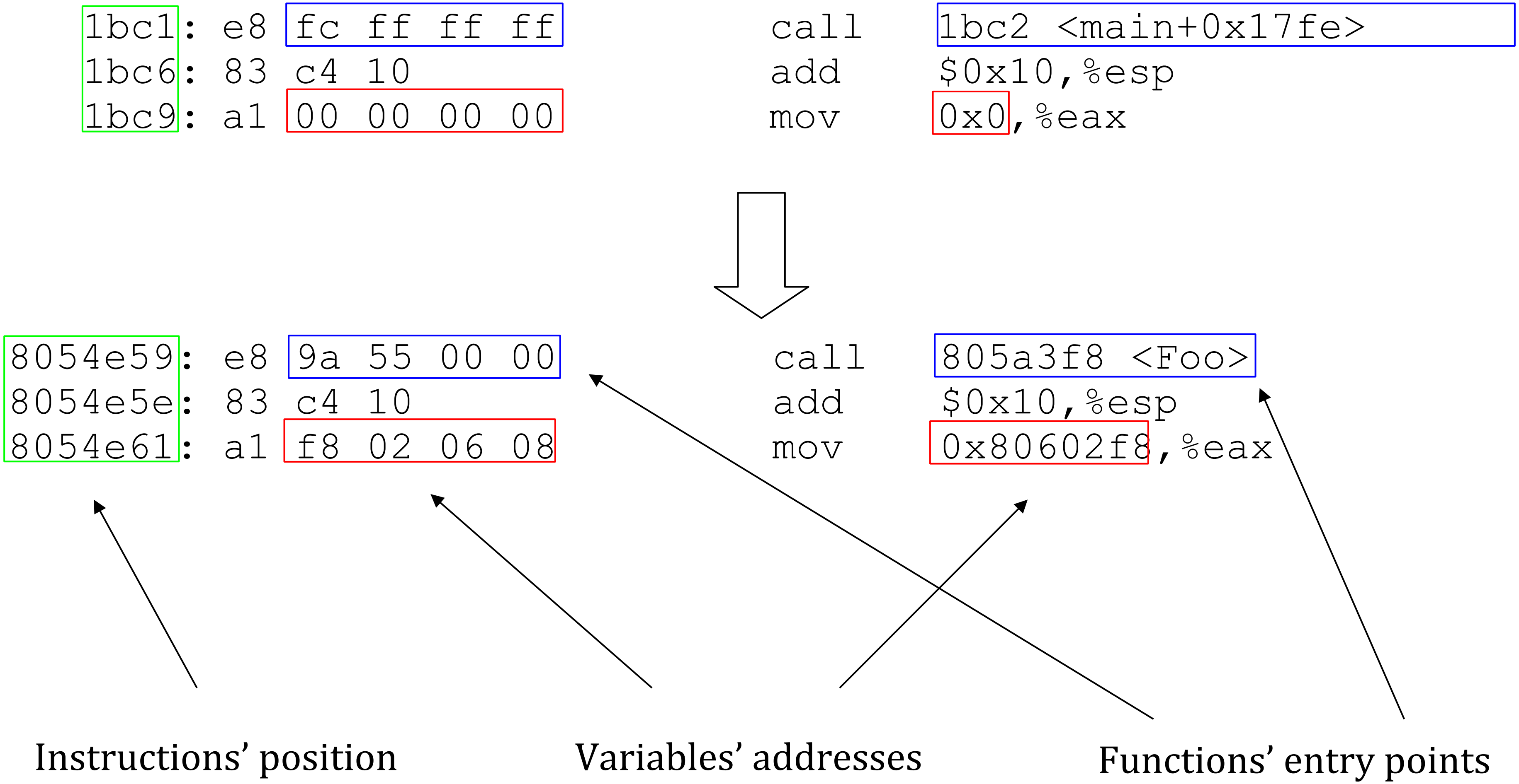
Segment 1

Segment 2

Segment 3

Executable
File

Static Relocation



```
1bc1: e8 fc ff ff ff                 call    1bc2 <main+0x17fe>
1bc6: 83 c4 10                       add     $0x10,%esp
1bc9: a1 00 00 00 00                 mov     0x0,%eax
```

```
8054e59: e8 9a 55 00 00              call    805a3f8 <Foo>
8054e5e: 83 c4 10                    add     $0x10,%esp
8054e61: a1 f8 02 06 08              mov     0x80602f8,%eax
```

Instructions' position

Variables' addresses

Functions' entry points

Directives: Linker Script

- The simplest form of linker script contains only a `SECTIONS` directive;
- The `SECTIONS` directive describes memory layout of the linker-generated file.

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

Sets *location counter*'s value

Places all input files's `.text` sections in the output file's `.text` section at the address specified by the *location counter*.

Example: C code

```
#include <stdio.h>

int xx, yy;

int main(void) {
   xx = 1;
   yy = 2;
   printf ("xx %d yy %d\n", xx, yy);
}
```

Example: ELF Header

```
$ objdump -x example-program

esempio-elf: file format elf32-i386
architecture: i386,
flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048310
```

Example: Program Header

```
   PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
        filesz 0x00000100 memsz 0x00000100 flags r-x
 INTERP off    0x00000134 vaddr 0x08048134 paddr 0x08048134 align 2**0
        filesz 0x00000013 memsz 0x00000013 flags r--
   LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
        filesz 0x000004f4 memsz 0x000004f4 flags r-x
   LOAD off    0x00000f0c vaddr 0x08049f0c paddr 0x08049f0c align 2**12
        filesz 0x00000108 memsz 0x00000118 flags rw-
DYNAMIC off    0x00000f20 vaddr 0x08049f20 paddr 0x08049f20 align 2**2
        filesz 0x000000d0 memsz 0x000000d0 flags rw-
   NOTE off    0x00000148 vaddr 0x08048148 paddr 0x08048148 align 2**2
        filesz 0x00000020 memsz 0x00000020 flags r--
  STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
        filesz 0x00000000 memsz 0x00000000 flags rw-
  RELRO off    0x00000f0c vaddr 0x08049f0c paddr 0x08049f0c align 2**0
        filesz 0x000000f4 memsz 0x000000f4 flags r--
```

Example: Dynamic Section

```
NEEDED          libc.so.6
INIT            0x08048298
FINI            0x080484bc
HASH            0x08048168
STRTAB          0x08048200
SYMTAB          0x080481b0
STRSZ           0x0000004c
SYMENT          0x00000010
DEBUG           0x00000000
PLTGOT          0x08049ff4
PLTRELSZ        0x00000018
PLTREL          0x00000011
JMPREL          0x08048280
```

There is the need to link to this shared library to use printf()

Example: Section Header

```
Idx Name                Size      VMA       LMA        File off  Algn

2   .hash           00000028  08048168  08048168  00000168  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
10  .init           00000030  08048298  08048298  00000298  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
11  .plt            00000040  080482c8  080482c8  000002c8  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
12  .text           000001ac  08048310  08048310  00000310  2**4
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
13  .fini           0000001c  080484bc  080484bc  000004bc  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
14  .rodata         00000015  080484d8  080484d8  000004d8  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, ATA
22  .data           00000008  0804a00c  0804a00c  0000100c  2**2
                    CONTENTS, ALLOC, LOAD, DATA
23  .bss            00000010  0804a014  0804a014  00001014  2**2
                    ALLOC
```

Example: Symbol Table

```
...
00000000 l    df *ABS*     00000000         esempio-elf.c
08049f0c l       .ctors    00000000         .hidden __init_array_end
08049f0c l       .ctors    00000000         .hidden __init_array_start
08049f20 l     O .dynamic  00000000         .hidden _DYNAMIC
0804a00c  w      .data     00000000         data_start
08048420 g     F .text     00000005         __libc_csu_fini
08048310 g     F .text     00000000         _start
00000000  w      *UND*     00000000         __gmon_start__
...
08049f18 g     O .dtors    00000000         .hidden __DTOR_END__
08048430 g     F .text     0000005a         __libc_csu_init
00000000       F *UND*     00000000         printf@@GLIBC_2.0
0804a01c g     O .bss      00000004         yy
0804a014 g       *ABS*     00000000         __bss_start
0804a024 g       *ABS*     00000000         _end
0804a014 g       *ABS*     00000000         _edata
0804848a g     F .text     00000000        .hidden __i686.get_pc_thunk.bx
080483c4 g     F .text     0000004d         main
08048298 g     F .init     00000000         _init
0804a020 g     O .bss      00000004         xx
```

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

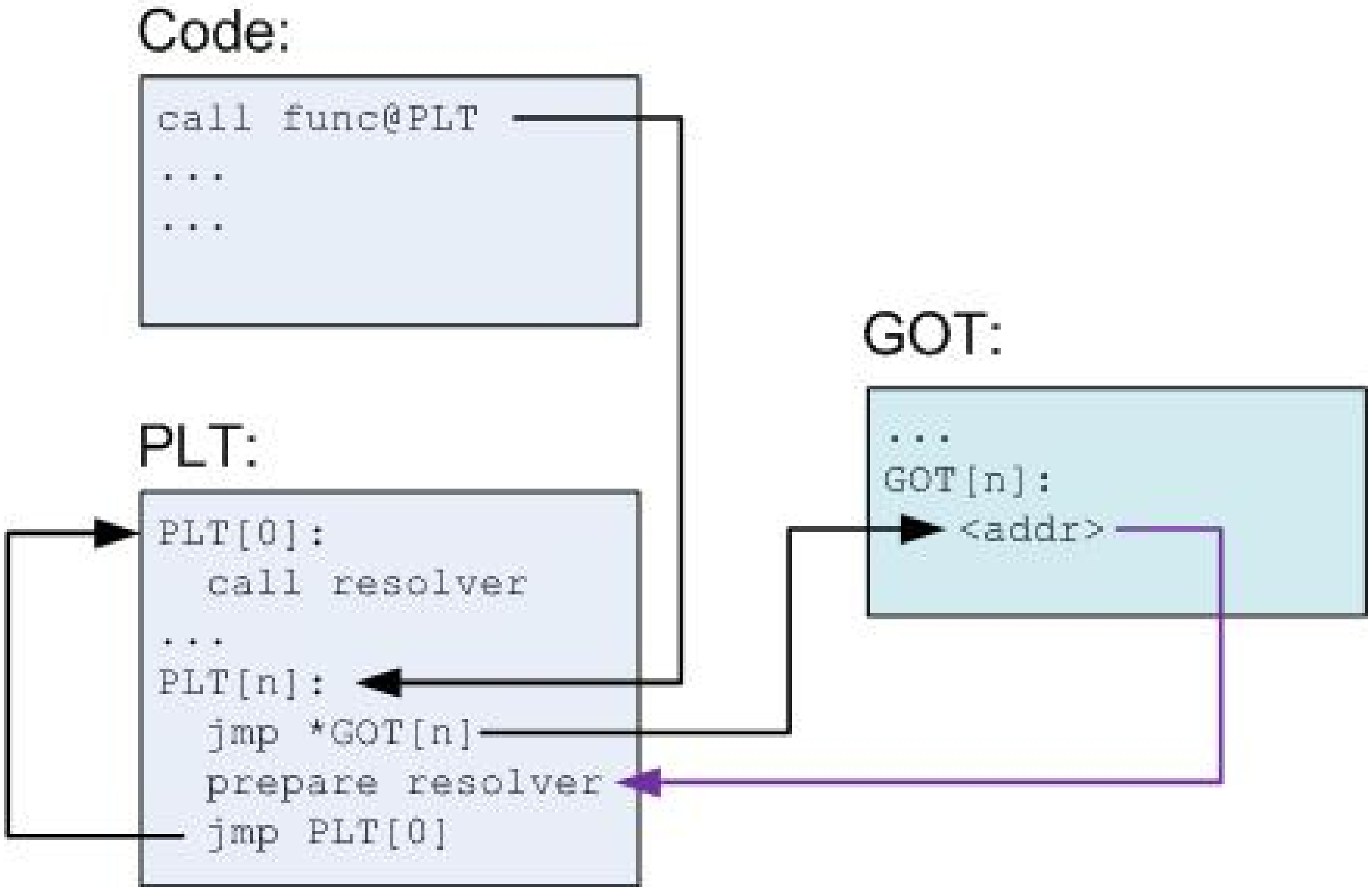
- `.dynsym`: a minimal symbol table used by the dynamic linker when performing relocations
- `.hash`: a hash table that is used to quickly locate a given symbol in the `.dynsym`, usually in one or two tries.
- `.dynstr`: string table related to the symbols stored in `.dynsym`

- 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



Code:

```
call  func@PLT
...
...
```

PLT:

```
PLT[0]:
  call  resolver
...
PLT[n]:
  jmp  *GOT[n]
  prepare  resolver
  jmp  PLT[0]
```

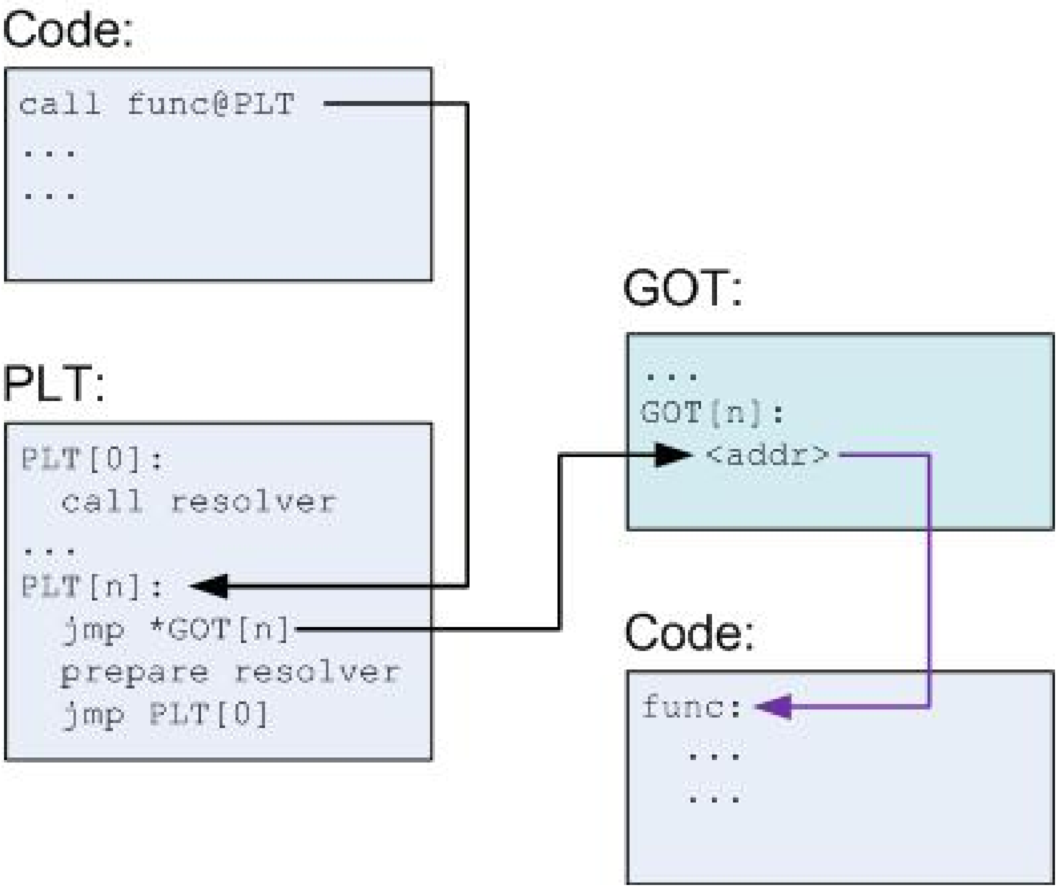
GOT:

```
...
GOT[n]:
   <addr>
```

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!

`__libc_start_main()`

- This function is defined as:

```
int __libc_start_main(
   int (*main)(int, char **, char **),
   int argc, char **ubp_av,
   void (*init)(void),
   void (*fini)(void),
   void (*rtld_fini)(void),
   void *stack_end
);
```

- `__start()` pushes parameters in reverse order on stack

Explanation of Parameters

`int(*main)(int, char**,char**)`	`main` of our program called by `__libc_start_main`. Return value of main is passed to `exit()` which terminates our program.
`arcg`	`argc` off of the stack.
`char **ubp_av`	`argv` off of the stack.
`void (*init)(void)`	`__libc_csu_init` - Constructor of this program. Called by `__libc_start_main` before `main`.
`void (*fini)(void)`	`__libc_csu_fini` - Destructor of this program. Registered by `__libc_start_main` with `__cxat_exit()`.
`void (*rtld_fini)(void)`	Destructor of dynamic linker from loader passed in %edx. Registered by `__libc_start_main` with `__cxat_exit()` to call the FINI for dynamic libraries that got loaded before us.
`void (*stack_end)`	Our aligned stack pointer.

...what about environment variables?

- There are no environment variables passed here!
- `__libc_start_main` calls `__libc_init_first`
  - It finds the first argument after the `NULL` terminating `argv`
  - Sets the global variable `__environ`
- `__libc_start_main` uses the same trick
  - After the `NULL` terminating `envp` there is another vector
  - This is the **ELF Auxiliary table**
  - It holds information used by the loader

ELF Auxiliary Table

- Setting the environment variable `LD_SHOW_AUXV=1` before running the program dumps its content

```
$ LD_SHOW_AUXV=1 ./example-program
AT_SYSINFO: 0xe62414
AT_SYSINFO_EHDR: 0xe62000
AT_HWCAP: fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush acpi mmx fxsr ss
sse2 ss ht tm pbe
AT_PAGESZ: 4096
AT_CLKTCK: 100
AT_PHDR: 0x8048034
AT_PHENT: 32
AT_PHNUM: 8
AT_BASE: 0x686000
AT_FLAGS: 0x0
AT_ENTRY: 0x80482e0
AT_UID: 1002 AT_EUID: 1002 AT_GID: 1000 AT_EGID: 1000 AT_SECURE: 0
AT_RANDOM: 0xbff09acb
AT_EXECFN: ./example-program
AT_PLATFORM: i686
```

`__libc_start_main()`

- Starts up threading
- Registers the `fini` (our program), and `rtld_fini` (run-time loader) arguments to get run by `at_exit` to run the program's and the loader's cleanup routines
- Calls `__libc_csu_init` which calls `_init`
- Calls the main with the `argc` and `argv` arguments passed to it and with the global `__environ` argument as detailed above.
- Calls exit with the return value of main

`_init()`

- This is the *program's constructor*
  - Constructors came far before C++!
- Three main steps:
  - If `gmon_start` in the PLT is not null, the program is being profiled. So `gmon_start` is called to setup profiling
  - Call `frame_dummy`, which sets up parameters to calls `__register_frame_info`: this sets up frame unwinding for exceptions management
  - Last call is done to invoke recursively actual constructors: `_do_global_ctors_aux`

`_do_global_ctors_aux()`

- This is defined in gcc’s source code in `crtstuff.c`

```
__do_global_ctors_aux (void) {
  func_ptr *p;
  for (p = __CTOR_END__ - 1; *p != (func_ptr) -1; p--)
  (*p) ();
}
```

- `__CTOR_END__` is a global variable keeping the number of constructors available for the program

How to implement a Constructor

- It's gcc stuff, so we can use a gcc attribute

```
#include <stdio.h>

void __attribute__ ((constructor)) a_constructor() {
    printf("%s\n", __FUNCTION__);
}
```

- `a_constructor()` will be called right before giving control to `main()`

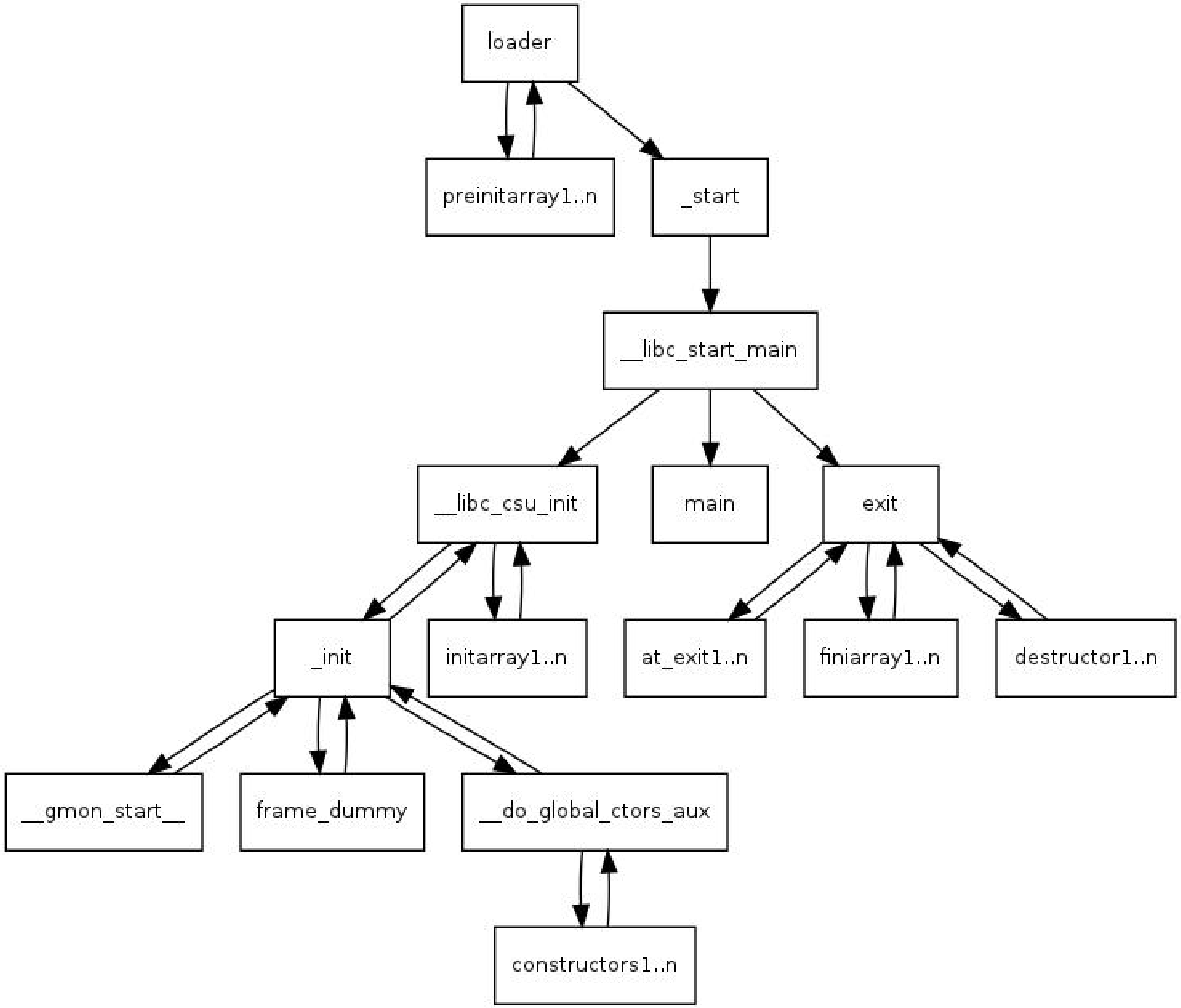
Back to `__libc_csu_init()`

```
void __libc_csu_init(int argc, char **argv, char **envp) {
  _init ();
  const size_t size = __init_array_end-__init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i])(argc, argv, envp);
}
```

- Again, we can directly run code here, getting arguments as well
- We can hook a function pointer in this way:

```
__attribute__((section(".init_array")))
typeof(init_function) *__init = init_function;
```

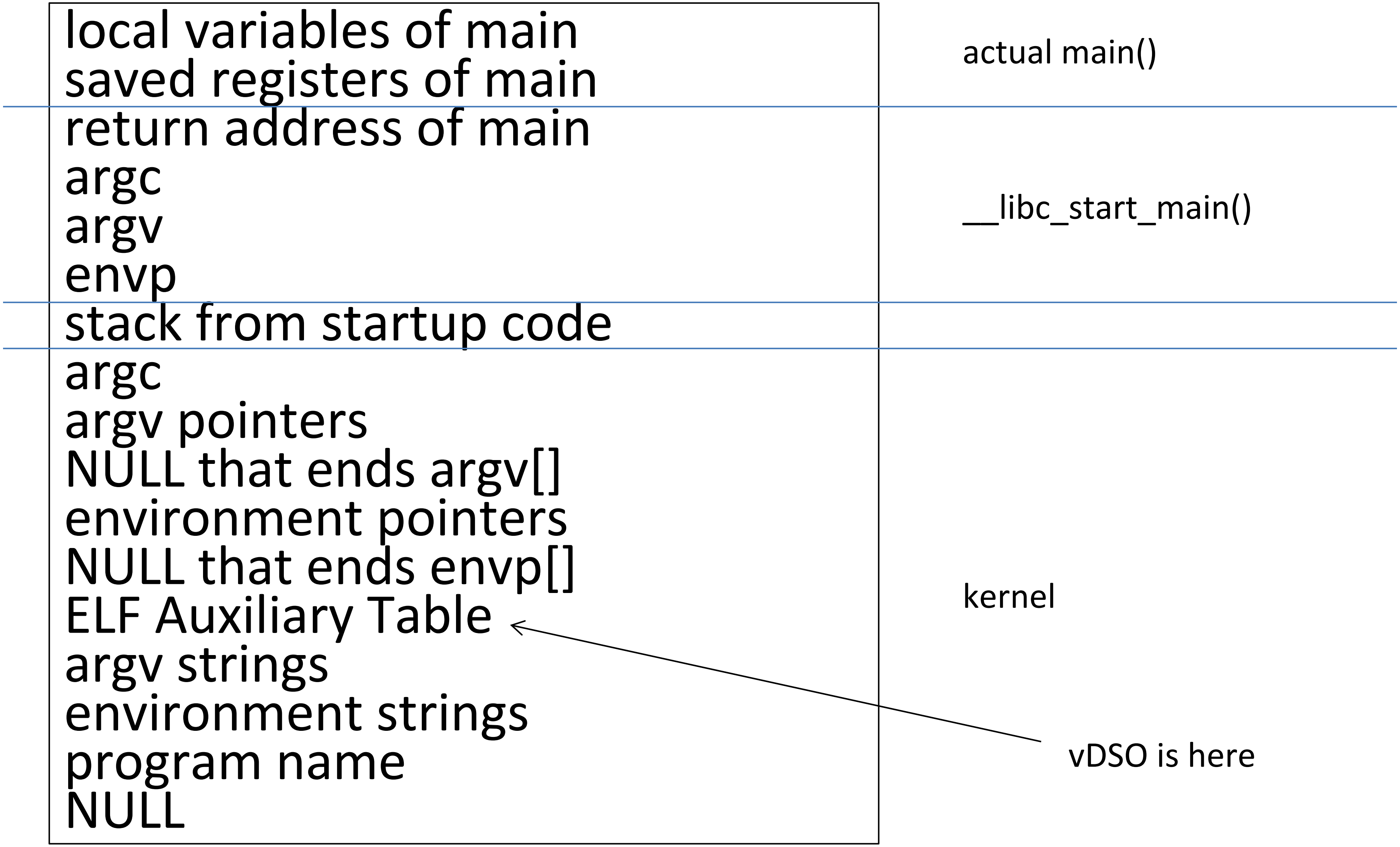
The Final Picture



Using all facilities of program startup/finalization

EXAMPLE SESSION

Stack Layout at Program Startup



local variables of main
saved registers of main

return address of main
argc
argv
envp

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

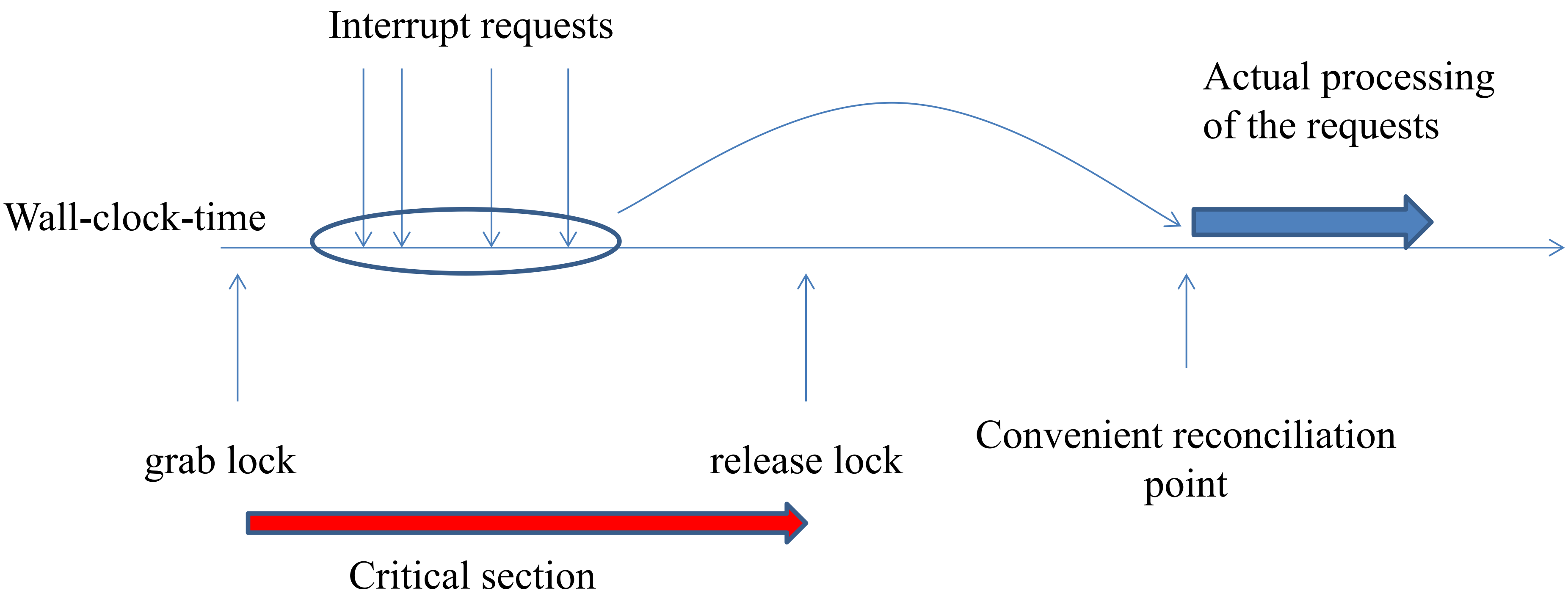
Tasks vs Processes

- Different types of execution traces must be handled:
  - User mode process/thread
  - Kernel mode process/thread
  - Interrupt management
- Non-determinism:
  - Due to nesting of user/kernel mode traces and interrupt management traces
- Performance:
  - Non-determinism may give rise to inefficiency whenever the evolution of the traces is tightly coupled (like on SMP and multi-core machines)
  - *Timing expectations for critical sections can be altered*

Design methodologies

- **Temporal reconciliation**
  - Interrupt management is mapped to process/thread traces and shifted in time
  - Management of events in the system can be aggregated (many-to-one aggregation)
  - Priority-based scheduling mechanisms are required not to induce starvation

A schematization



Interrupt requests

Actual processing of the requests

Wall-clock-time

grab lock

release lock

Convenient reconciliation point

Critical section

Reconciliation points

- **Guarantees:**
  - "Eventually"
- **Conventional support:**
  - When returning from syscall:
    - Application-level related techniques
  - Upon context-switch:
    - Idle-process related techniques
  - Reconciliation in process-context:
    - Kernel-thread realted techniques

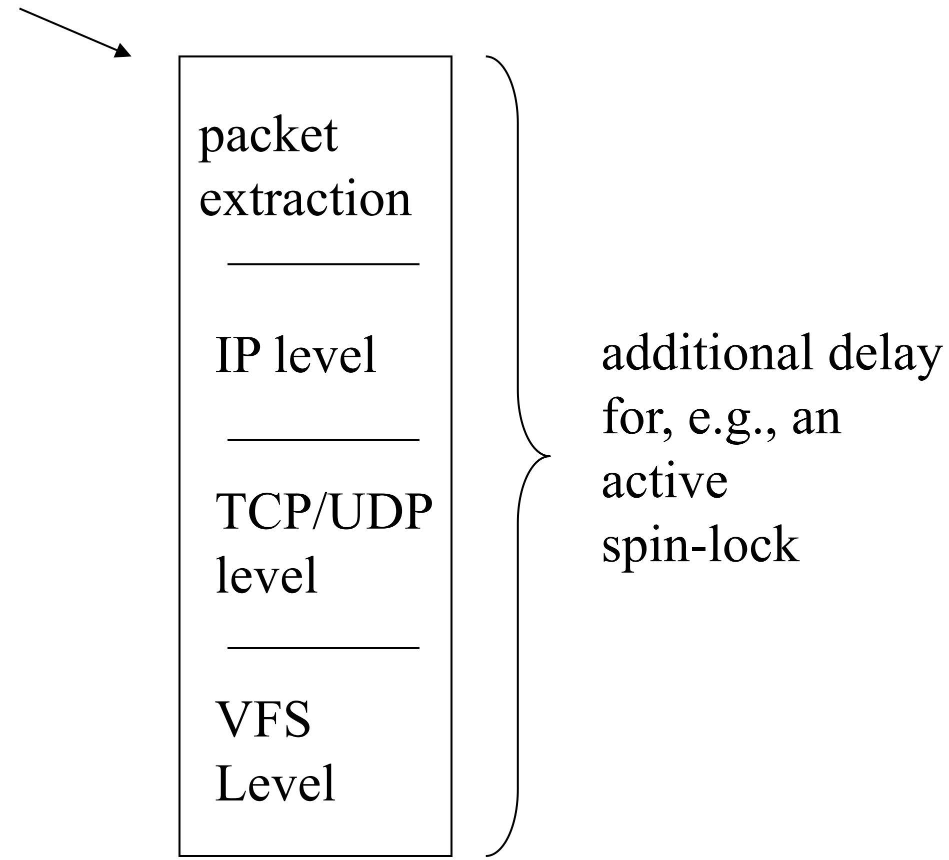
The historical concept: top/bottom halves

- Management of tasks comes at two levels: top half and bottom halves
- The top-half executes a minimal amount of work which is mandatory to later finalize the whole interrupt management
- The top-half code is typically managed according to a non-interruptible scheme
- The finalization of the work takes place via the bottom-half level
- The top-half takes care of *scheduling* the bottom-half task by queuing a record into a proper data structure

An example: sockets

monolithic execution

interrupt from network device



packet extraction

IP level

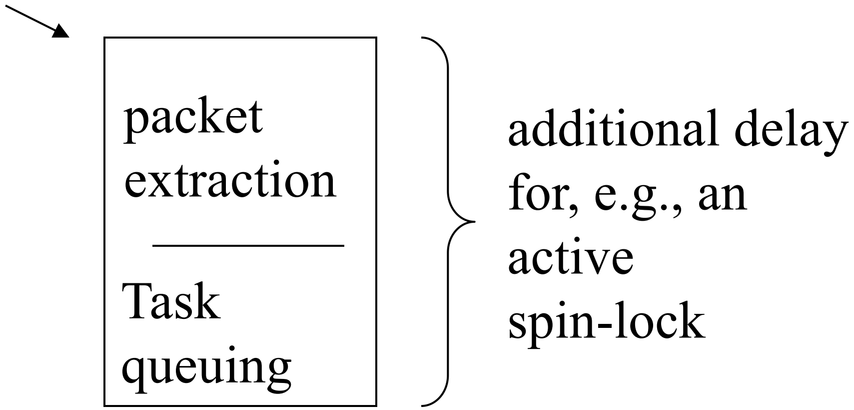
TCP/UDP level

VFS Level

additional delay for, e.g., an active spin-lock

top/bottom half

interrupt from network device

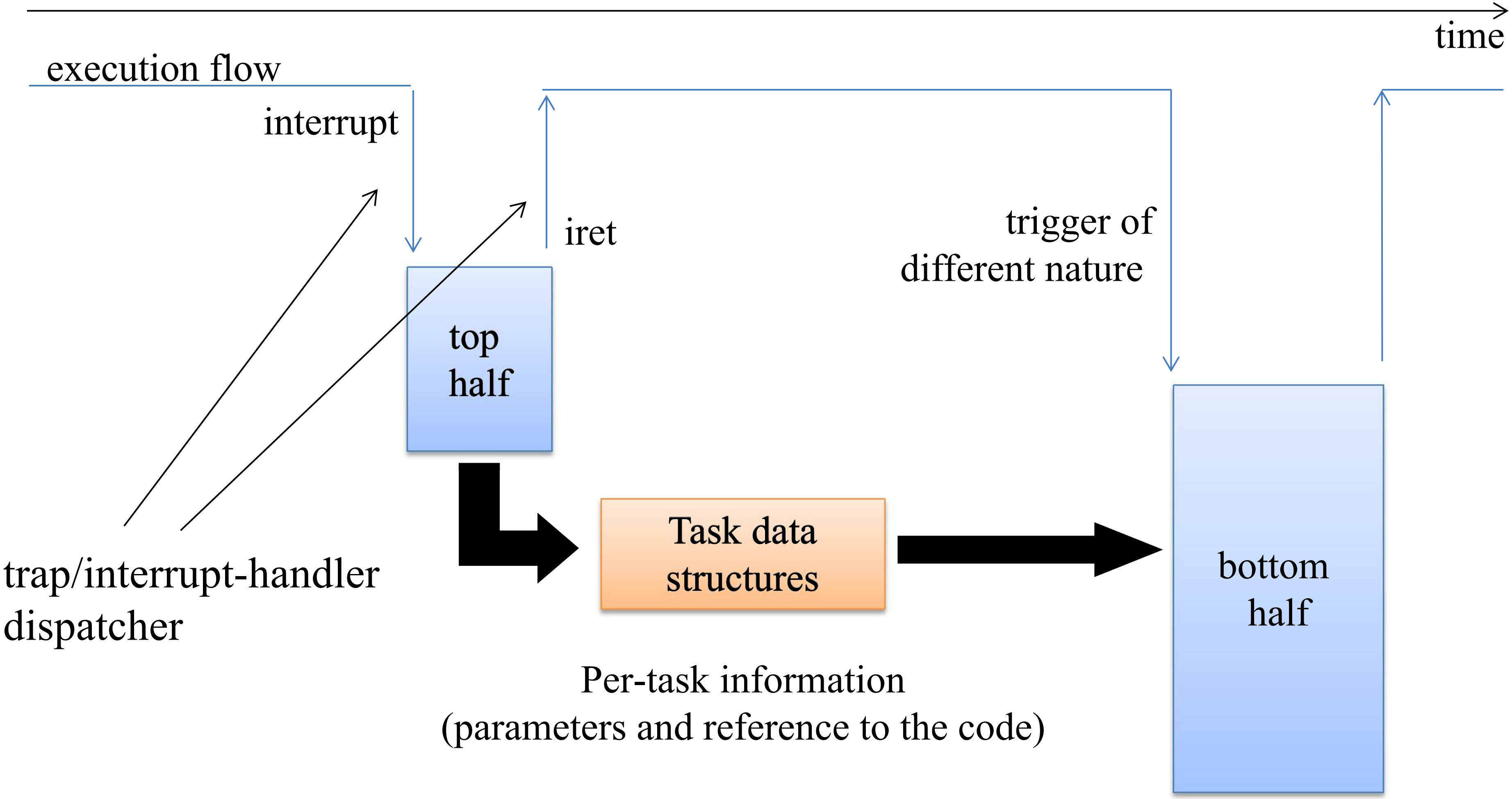


packet extraction

Task queuing

additional delay for, e.g., an active spin-lock

The historical concept: top/bottom halves



Historical Evolution in Linux

- Up to Kernel 2.5, there was one single facility:
  - Task queues
- Later versions improved the organization towards SMP/multicore systems
- More facilities have been added:
  - Tasklet / Soft IRQs
  - Work Queues
  - Kernel Timers

Task Queues

- Queuing structures, which can be associated with variable names
- Linux 2.2 already had some predefined task-queues:
  - `tq_immediate`: task to be executed upon timer-interrupt or syscall return
  - `tq_timer`: task to be executed upon timer-interrupt
  - `tq_scheduler`: task to be executed in *process context*

Task Queues' Data Structures

- Additional task queues could be declared using the macro `DECLARE_TASK_QUEUE(queuename)` defined in `include/linux/tqueue.h`
  - this macro also initializes the task queue as empty
- The structure of a task was defined in `include/linux/tqueue.h`

```
struct tq_struct {
   struct tq_struct *next; /* l-list of active bh's*/
   int sync; /* must be initialized to zero */
   void (*routine)(void *); /* function to call */
   void *data; /* argument to function */
}
```

Task management API

- To register a task to a task queue: `int queue_task(struct tq_struct *task, task_queue *list)`
- To flush tasks: `void run_task_queue(task_queue *list)`
  - All tasks were executed and deallocated
  - Non-predefined queues must be explicitly flushed
- The `tq_schedule` queue had a specific function to run tasks: `int schedule_task(struct tq_struct *task)`
- The return value of queuing functions is non-zero if the task is not already registered in the queue
  - The `sync` member is set to 1 when the task is queued
- A call to `void mark_bh(IMMEDIATE_BH)` is mandatory for the immediate task queue

Bottom-half Activation and Caveats

- Linux called `do_bottom_half()` (defined in `kernel/softirq.c`)
  - In `schedule()`
  - In `ret_from_sys_call()`
- The execution of a bottom half was done in process context
- Yet, blocking services had not to be executed
  - This could have created problems to the consistency of the context

Task Queues Limitations

- Original task queues limitations:
  - Single thread execution of the tasks
  - Locality was not maximized
  - Heavy interrupt load was problematic

- The newer approach:
  - Multithread execution of bottom half tasks
  - Binding of task execution to CPU cores

- Task Queues are no longer present in the Kernel source

Tasklets

- Tasklets are data structures used to track a specific task, related to the execution of a specific function in the kernel
- The function accepts a parameter (an `unsigned long`) and is of type `void`
- Tasklets are declared as (`include/linux/interrupt.h`):
  - `DECLARE_TASKLET(tasklet, function, data)`
  - `DECLARE_TASKLET_DISABLED(tasklet, function, data)`
- If declared as disabled, tasks will not be executed until enabled

Enabling and Running Tasklets

```
tasklet_enable(struct tasklet_struct *tasklet)
tasklet_hi_enable( struct tasklet_struct *);
tasklet_disable(struct tasklet_struct *tasklet)
void tasklet_schedule(struct tasklet_struct *tasklet)
```

- Each tasklet represents a single task, it is not equivalent to a Task Queue
- Subsequent reschedule of a same tasklet may still result in a single execution, depending on whether the tasklet was already flushed or not (no concept of queueing)

Tasklet Execution

- Tasklets are executed on *specific kernel threads* (CPU affinity could be used)
- If the tasklet has already been scheduled, it will not be moved to another CPU if it's still pending
- Tasklets have schedule levels similar to that of `tq_schedule`
- Execution context should be an “interrupt-context”: no-sleep phases within the tasklet

How Tasklets are Run

- Tasklets are run using Soft IRQs

  «First of all, it's a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them».
- Enable functions are mapped to Soft IRQs:
  - `tasklet_enable()` mapped to `TASKLET_SOFTIRQ`
  - `tasklet_hi_enable()` mapped to `HI_SOFTIRQ`
- Soft IRQ management takes place in `kernel/softirq.c` via per-CPU variables

Soft IRQ Firing

- There are two places where Soft IRQs are fired:
  - At the end of the processing for a hardware interrupt
    - Drivers often set Soft IRQs
    - It is cache-wise to check for them immediately
  - Any time that kernel code re-enables softirq processing (via a call to functions like `local_bh_enable()` or `spin_unlock_bh()`)
    - A victim is randomly chosen here!

Work Queues

- More recent deferral mechanisms introduced in 2.5.41
- Made Task Queues deprecated
- They can have a latency higher than Tasklets, but have a richer API and blocking calls can be issued (although discouraged)
- Interrupts and bottom halves are both enabled while the work queues are being run
- They are run in separate workers

Work Queue Main Datastructure

- This is defined in `linux/workqueue.h` as:

```
struct work_struct {
     atomic_long_t data;
     struct list_head entry;
     work_func_t func;
};


typedef void (*work_func_t)(struct work_struct
*work);
```

Work Queues Main API Function

```
INIT_WORK( work, func );
INIT_DELAYED_WORK( work, func );
INIT_DELAYED_WORK_DEFERRABLE( work, func );

struct workqueue_struct *create_workqueue( name );
void destroy_workqueue( struct workqueue_struct * );
int schedule_work( struct work_struct *work );
int schedule_work_on( int cpu, struct work_struct *work );

int scheduled_delayed_work( struct delayed_work *dwork,
unsigned long delay );
int scheduled_delayed_work_on(int cpu, struct
delayed_work *dwork, unsigned long delay );
```

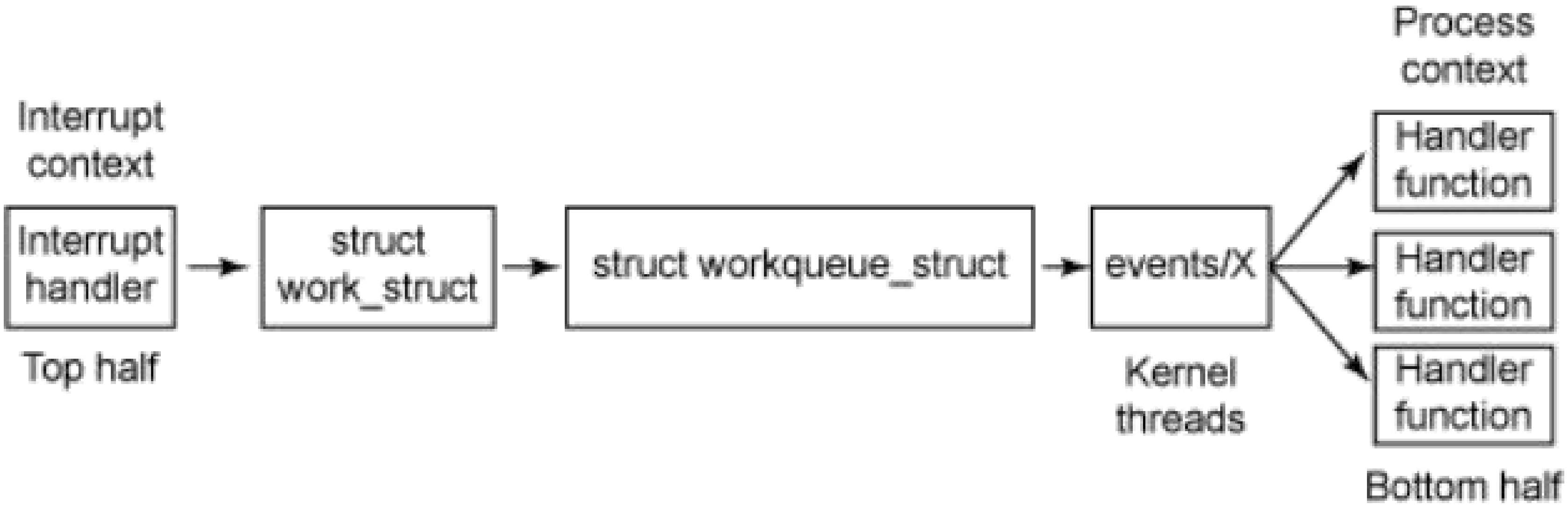
struct delayed_work

```
struct delayed_work {
    struct work_struct work;
    struct timer_list timer;

    /* target workqueue and CPU ->timer uses to queue ->work */
    struct workqueue_struct *wq;
    int cpu;
};
```

struct workqueue_struct

```
struct workqueue_struct {
    struct list_head pwqs;/* WR: all pwqs of this wq */
    struct list_head list;/* PR: list of all workqueues */
    struct mutex mutex;           /* protects this wq */
    int  work_color;         /* WQ: current work color */
    ...
    struct list_head maydays;     /* MD: pwqs requesting
rescue */
    struct worker *rescuer;       /* I: rescue worker */
    char        name[WQ_NAME_LEN];      /* I: workqueue name
*/
    ...
    struct pool_workqueue __percpu *cpu_pwqs;
    ...
};
```

Work Queue Summary



Kernel Timers

- In Linux, time is measured by a global variable named `jiffies`, which identifies the number of ticks that have occurred since the system was booted (in `kernel/time/jiffies.c`)
- The `jiffies` global variable is used broadly in the kernel for a number of purposes
- One purpose is the current absolute time to calculate the time-out value for a timer

Kernel Timer Main Data Structure

- Defined in `include/linux/timer.h`

```
struct timer_list {
        /*
         * All fields that change during normal runtime
         * grouped to the same cacheline
         */
        struct hlist_node  entry;
        unsigned long      expires;
        void               (*function)(struct timer_list *);
        u32                flags;
};
```

Kernel Timer API

```
void init_timer( struct timer_list *timer );
```

```
void setup_timer( struct timer_list *timer,                         void (*function)(unsigned long), unsigned long data );
```

```
int mod_timer( struct timer_list *timer, unsigned long expires );
```

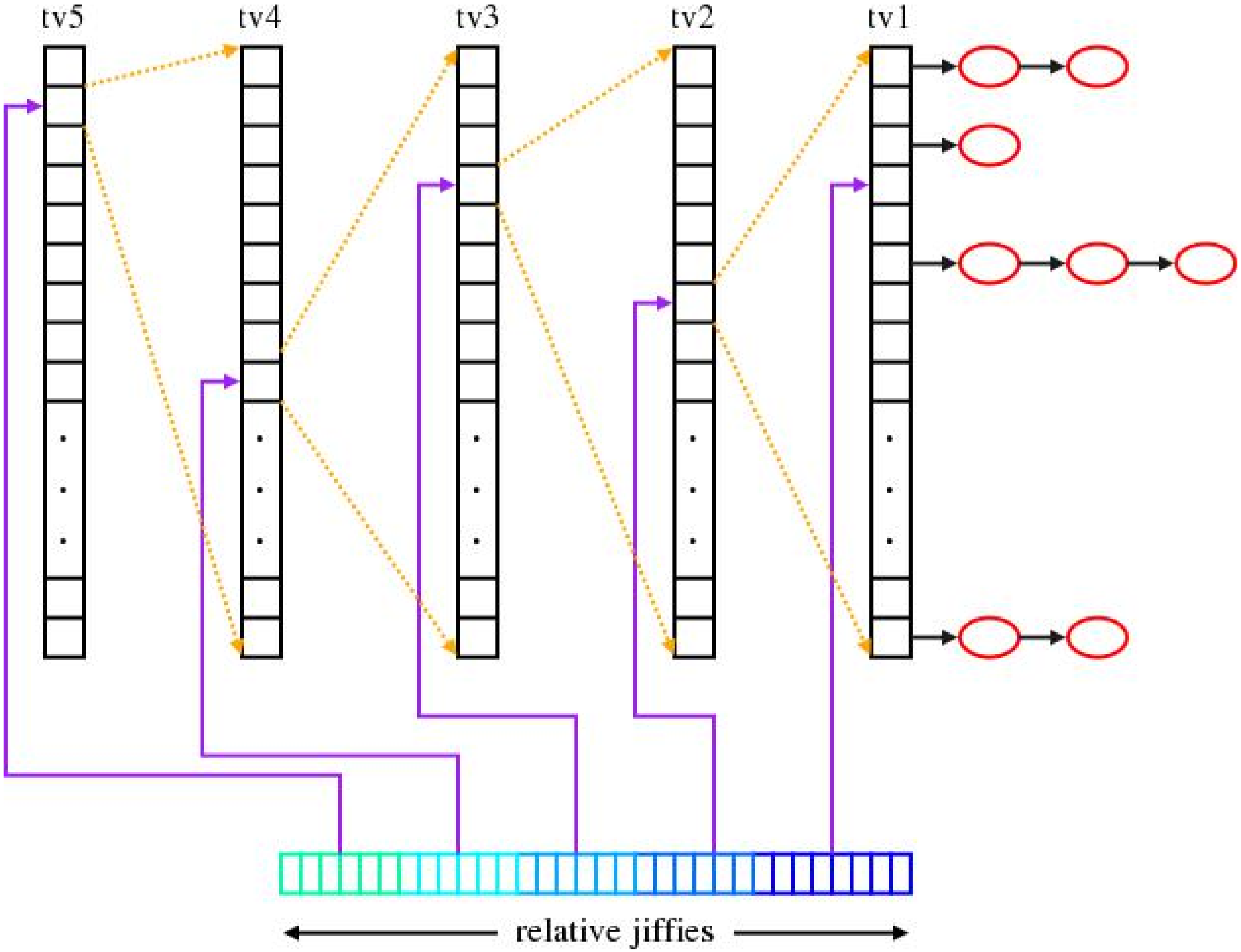
```
void del_timer( struct timer_list *timer );
```

```
int timer_pending( const struct timer_list *timer );
```

Kernel Timer Management

- Early Linux implementations had timers organized in a single list with nodes (slightly) ordered according to expiration time
- This was significantly unreliable and inefficient
- The *Timer Wheel*
  - A nested structure

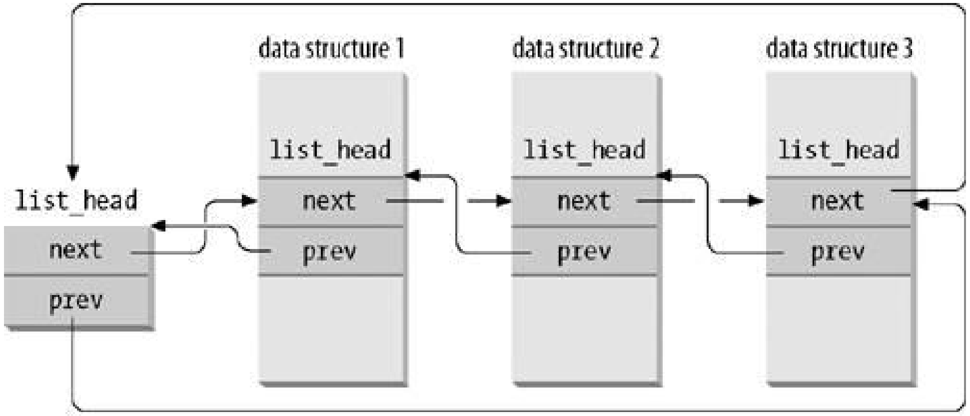
The Timer Wheel (2005)



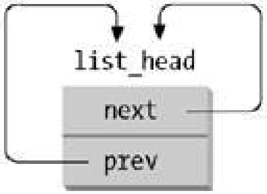
Generic Lists in Linux

```
struct workqueue_struct {
   struct list_head pwqs;/* WR: all pwqs of this wq */
   struct list_head list;/* PR: list of all workqueues */
   struct mutex mutex;          /* protects this wq */
   int  work_color;         /* WQ: current work color */
   ...
   struct list_head maydays;    /* MD: pwqs requesting
rescue */
   struct worker *rescuer;      /* I: rescue worker */
   char       name[WQ_NAME_LEN];      /* I: workqueue name
*/
   ...
   struct pool_workqueue __percpu *cpu_pwqs;
   ...
};
```

Generic Lists in Linux

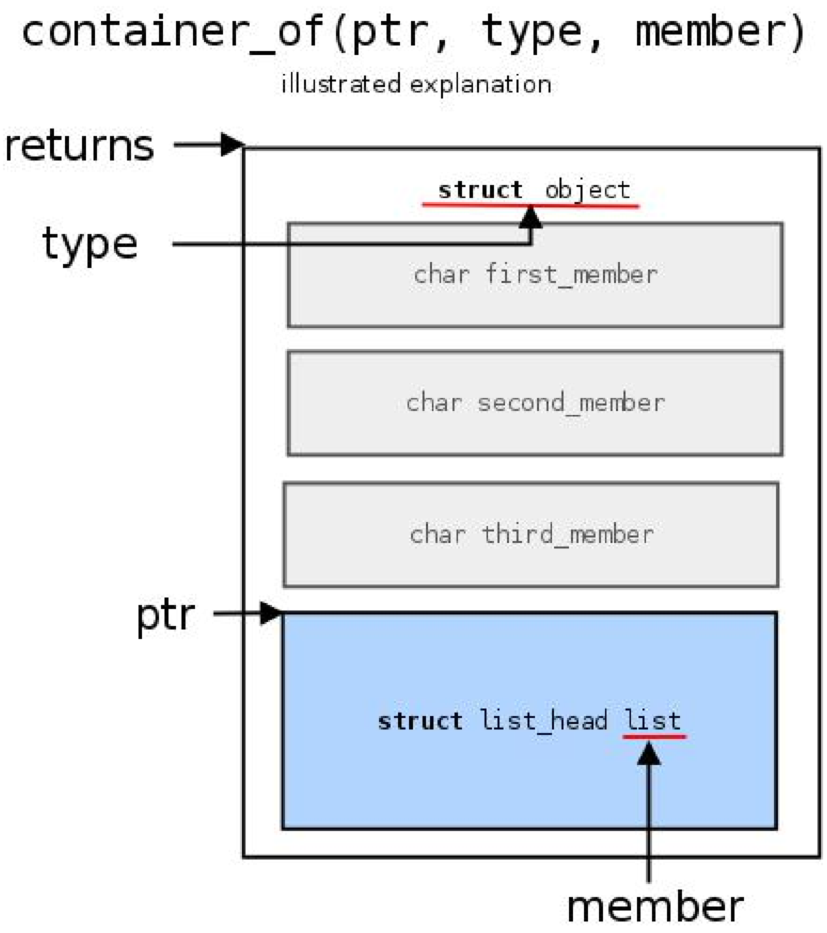


(a) a doubly linked listed with three elements



(b) an empty doubly linked list

Generic Lists in Linux



Look at `include/linux/list.h` for the API to manage and access lists

Timer Interrupts Management on 2.4

- They are handled according to the top/bottom half paradigm
- The top half executes the following actions:
  - Flags the Task Queue `tq_timer` as ready for flushing
  - Increments `jiffies`
  - Checks whether the CPU scheduler needs to be activated, and in the positive case flags `need_resched` (more on this later)
  - The bottom half is scheduled in the `tq_timer` Task Queue

Timer Interrupt Top Half on 2.4

- Defined in `linux/kernel/timer.c`

```
void do_timer(struct pt_regs *regs) {
   (*(unsigned long *)&jiffies)++;
 #ifndef CONFIG_SMP
   /* SMP process accounting uses
         the local APIC timer */

   update_process_times(user_mode(regs));
 #endif
   mark_bh(TIMER_BH);
   if (TQ_ACTIVE(tq_timer))
         mark_bh(TQUEUE_BH);
}
```

Timer Interrupt Bottom Half on 2.4

- Defined in `linux/kernel/timer.c`

```
void timer_bh(void)
{
   update_times();
   run_timer_list();
}
```

Timer Interrupt Activation on 2.4

```
Linux Timer IRQ
IRQ 0 [Timer]
 |
\|/
|IRQ0x00_interrupt        //   wrapper IRQ handler
   |SAVE_ALL              ---
      |do_IRQ                |   wrapper routines
         |handle_IRQ_event  ---
            |handler() -> timer_interrupt  // registered IRQ 0 handler
               |do_timer_interrupt
                  |do_timer
                     |jiffies++;
                     |update_process_times
                     |if (--counter <= 0) { // if time slice ended then
                        |counter = 0;        //   reset counter
                        |need_resched = 1;   //   prepare to reschedule
                     |}
         |do_softirq
         |while (need_resched) { // if necessary
            |schedule             //   reschedule
            |handle_softirq
         |}
   |RESTORE_ALL
```

Timer Interrupt Activation on 2.4

- `IRQ0x00_interrupt, SAVE_ALL [include/asm/hw_irq.h]`
- `do_IRQ, handle_IRQ_event [arch/i386/kernel/irq.c]`
- `timer_interrupt, do_timer_interrupt [arch/i386/kernel/time.c]`
- `do_timer, update_process_times [kernel/timer.c]`
- `do_softirq [kernel/soft_irq.c]`
- `RESTORE_ALL [arch/i386/kernel/entry.S]`

Timer Interrupt Activation on ≥2.6

```
__visible void __irq_entry smp_apic_timer_interrupt(struct
pt_regs *regs) {
    struct pt_regs *old_regs = set_irq_regs(regs);

    /*
     * NOTE! We'd better ACK the irq immediately,
     * because timer handling can be slow.
     *
     * update_process_times() expects us to have
     * done irq_enter().
     * Besides, if we don't timer interrupts ignore the global
     * interrupt lock, which is the WrongThing (tm) to do.
     */
      entering_ack_irq();
      trace_local_timer_entry(LOCAL_TIMER_VECTOR);
      local_apic_timer_interrupt();
      trace_local_timer_exit(LOCAL_TIMER_VECTOR);
      exiting_irq();

      set_irq_regs(old_regs);
}
```

Timer Interrupt Activation on ≥2.6

- In `arch/x86/kernel/apic/apic.c`

```
static DEFINE_PER_CPU(struct clock_event_device, lapic_events);

static void local_apic_timer_interrupt(void)
{
struct clock_event_device *evt = this_cpu_ptr(&lapic_events);

      ...
      inc_irq_stat(apic_timer_irqs);

      evt->event_handler(evt);
}
```

Clock Events

- They are an abstraction introduced in 2.6
- Clock Events are generated by Clock Event Devices
- This interface allows to drive hardware which can be programmed to send interrupts at different grains (e.g. the i8253)
- They are currently being used to implement a "tickless" kernel and a real-time kernel

High-Resolution Timers

- They are based on the `ktime_t` type (nanosecond scalar representation) rather than jiffies

```
struct hrtimer {
        struct timerqueue_node          node;
        ktime_t                         _softexpires;
        enum hrtimer_restart
        (*function)(struct hrtimer *);
        struct hrtimer_clock_base    *base;
        u8                           state;
        u8                           is_rel;
};
```

High-Resolution Timers API

```
void hrtimer_init( struct hrtimer *time,
clockid_t which_clock, enum hrtimer_mode
mode );
```

```
int hrtimer_start(struct hrtimer *timer,
ktime_t time, const enum hrtimer_mode mode);
```

```
int hrtimer_cancel(struct hrtimer *timer);
```

```
int hrtimer_try_to_cancel(struct hrtimer
*timer);
```

```
int hrtimer_callback_running(struct hrtimer
*timer);
```

POSIX Clocks

- `CLOCK_REALTIME`: This clock provides a best effort estimate of UTC in a way that is backwards compatible with existing practice. Very little is guaranteed for this clock. It will never show leap seconds
- `CLOCK_UTC`: This clock is only available when the system knows with high assurance Coordinated Universal Time (UTC) with an estimated accuracy of at least 1 s
- `CLOCK_TAI`: This clock is only available when the system knows International Atomic Time (TAI) with at least an accuracy of 1 s
- `CLOCK_MONOTONIC`: This clock never jumps, it is guaranteed to be available all the time right after system startup, and its frequency never varies by more than 500 ppm
- `CLOCK_THREAD`: This clock started its Epoch when the current thread was created and runs only when the current thread is running on the CPU
- `CLOCK_PROCESS`: This clock starts its Epoch when the current process was created and runs only when a thread of the current process is running on the CPU

Timer Interrupts and the Scheduler

- At some specific points (e.g., when returning from a syscall) the `need_resched` variable is checked
- In case of positive check, the actual scheduler is activated
- It corresponds to the `schedule()` function, defined in `kernel/sched/core.c`
- New versions replace `need_resched` with a call to `test_thread_flag(TIF_NEED_RESCHED)`

Back to the Task State Segment

- Each core has one per-CPU TSS:
  - `DECLARE_PER_CPU_PAGE_ALIGNED(struct tss_struct, cpu_tss_rw)` in `arch/x86/include/asm/processor.h`
- TSS is necessary to correctly support ring-change operations
- Upon reschedule, the current TSS is stored into the PCB of the about-to-be-descheduled process

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 for memory management purposes for the specific process, such as
  - 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`

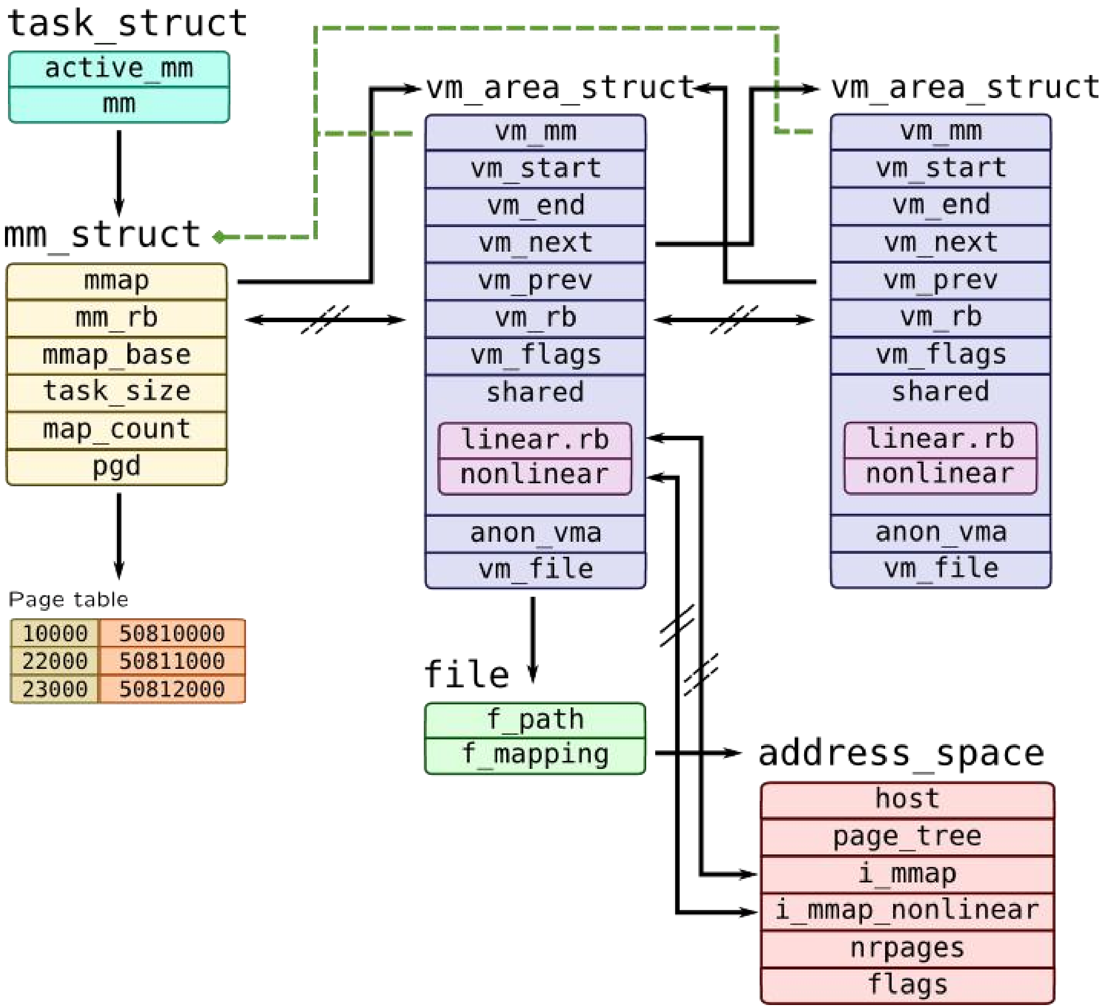
`vm_area_struct`

- 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



task_struct
active_mm
mm

mm_struct
mmap
mm_rb
mmap_base
task_size
map_count
pgd

Page table

10000	50810000
22000	50811000
23000	50812000

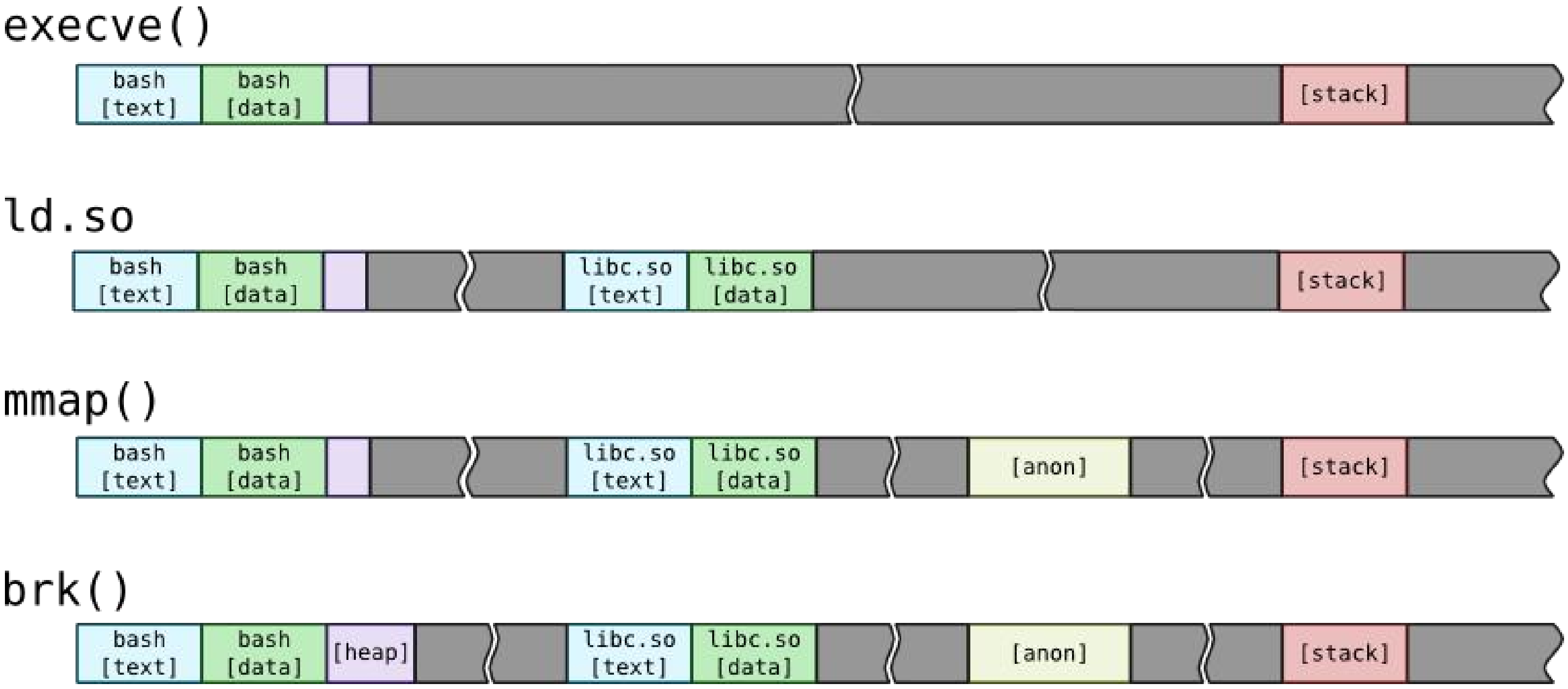
vm_area_struct
vm_mm
vm_start
vm_end
vm_next
vm_prev
vm_rb
vm_flags
shared
linear.rb
nonlinear
anon_vma
vm_file

vm_area_struct
vm_mm
vm_start
vm_end
vm_next
vm_prev
vm_rb
vm_flags
shared
linear.rb
nonlinear
anon_vma
vm_file

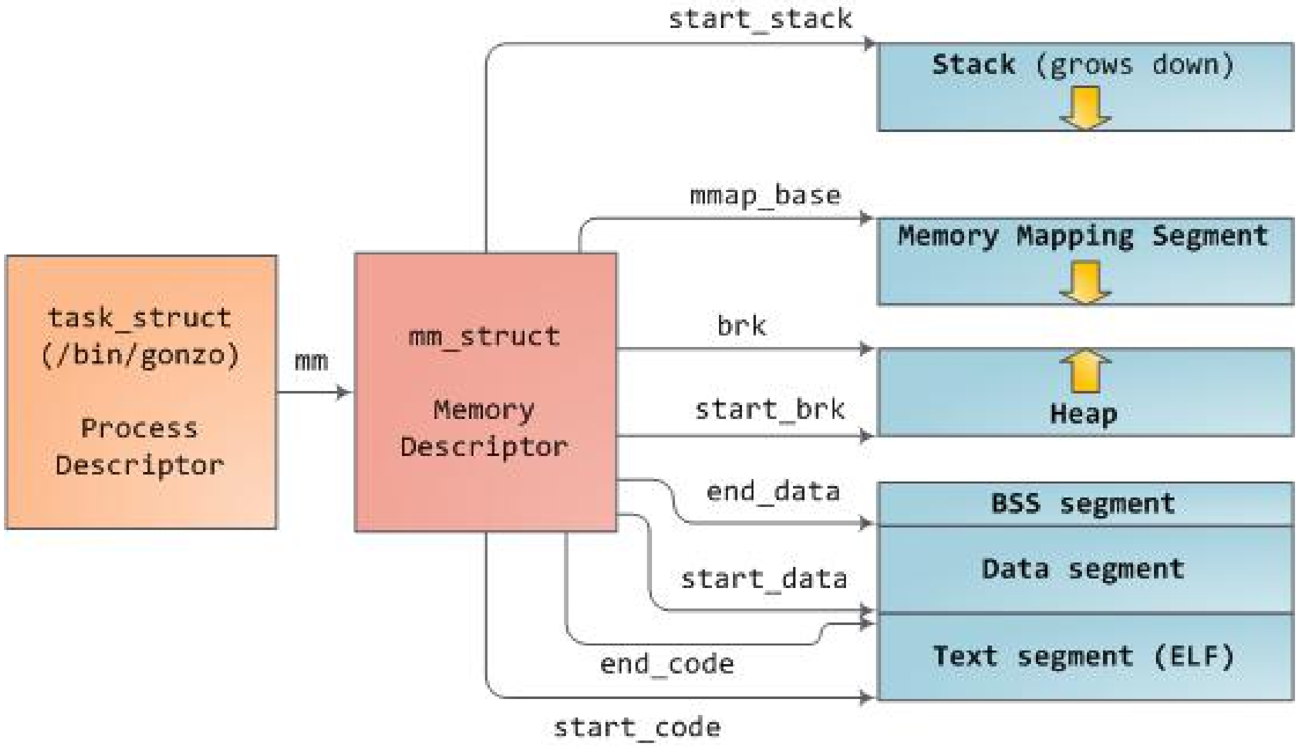
file
f_path
f_mapping

address_space
host
page_tree
i_mmap
i_mmap_nonlinear
nrpages
flags

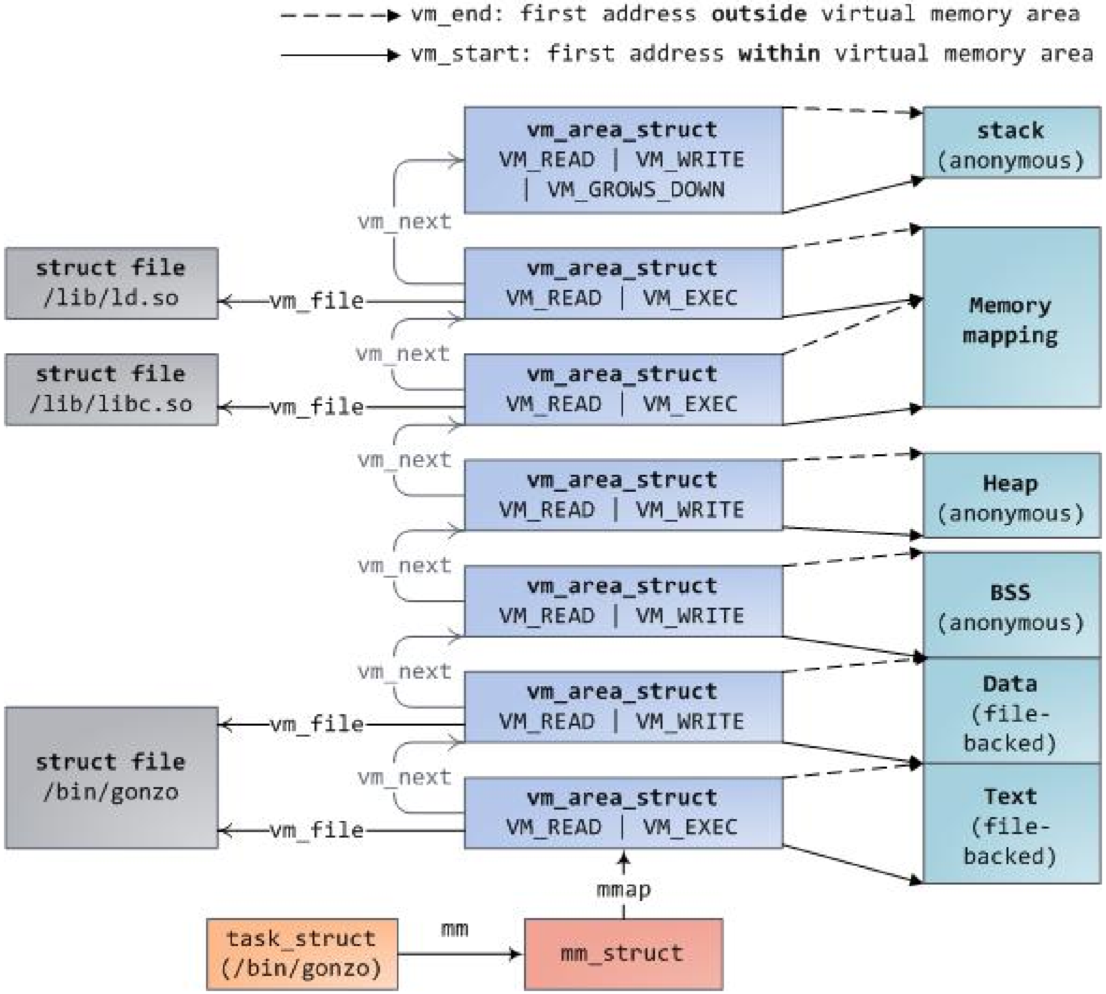
Userspace Memory Management



Userspace Memory Management



Userspace Memory Management



vm_end: first address **outside** virtual memory area

vm_start: first address **within** virtual memory area

vm_area_struct
VM_READ | VM_WRITE
| VM_GROWS_DOWN

stack
(anonymous)

vm_next

struct file
/lib/ld.so

vm_file

vm_area_struct
VM_READ | VM_EXEC

Memory mapping

vm_next

struct file
/lib/libc.so

vm_file

vm_area_struct
VM_READ | VM_EXEC

vm_next

vm_area_struct
VM_READ | VM_WRITE

Heap
(anonymous)

vm_next

vm_area_struct
VM_READ | VM_WRITE

BSS
(anonymous)

vm_next

struct file
/bin/gonzo

vm_file

vm_area_struct
VM_READ | VM_WRITE

Data
(file-backed)

vm_next

vm_file

vm_area_struct
VM_READ | VM_EXEC

Text
(file-backed)

mmap

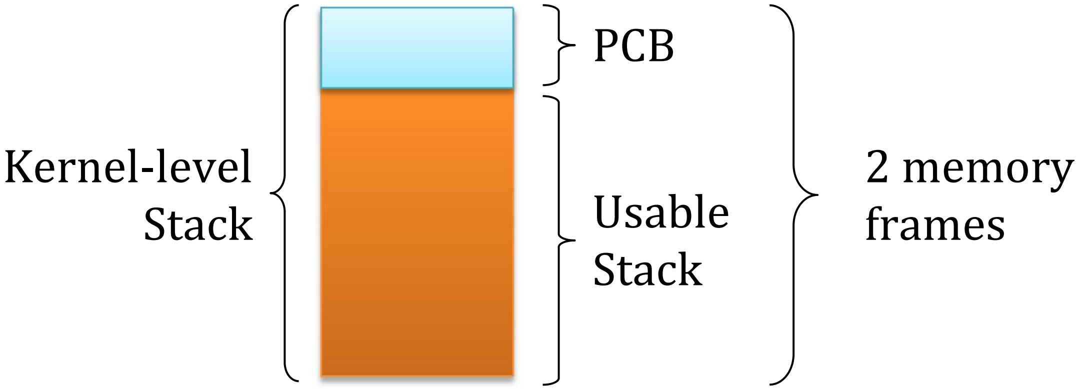
task_struct
(/bin/gonzo)

mm

mm_struct

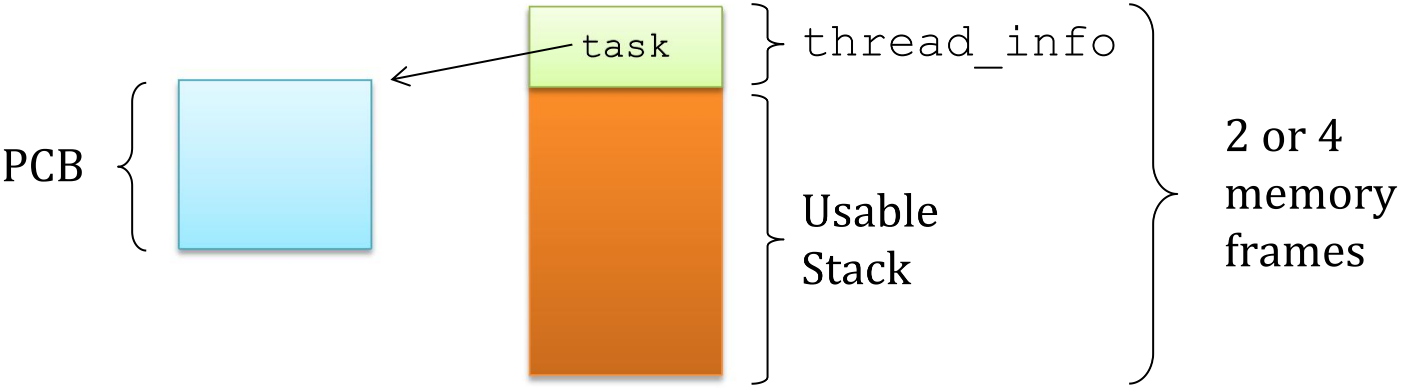
PCB Allocation up to 2.6

- 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

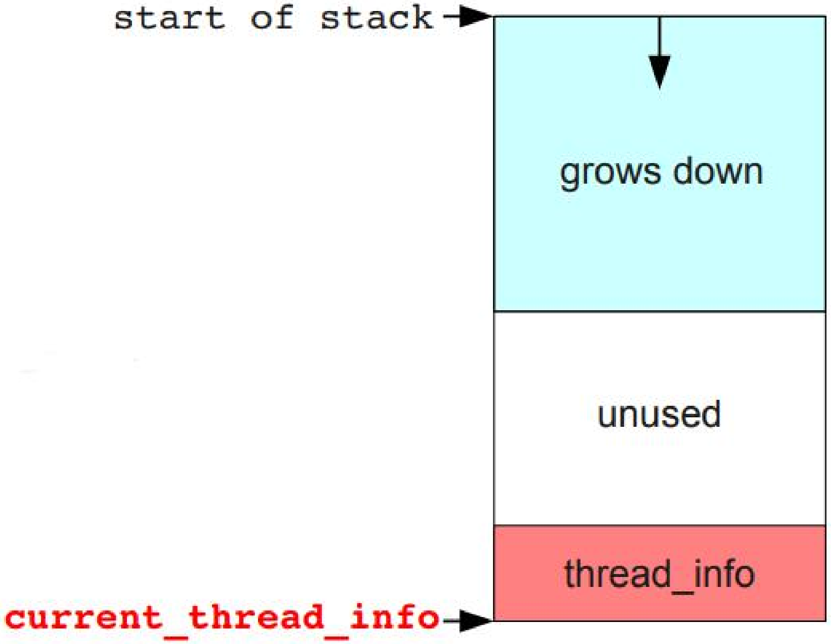
```
union thread_union {
      struct thread_info thread_info;
      unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

struct thread_info

```
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



Old `struct thread_info`

```
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!
(affect, 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

Linux Scheduler

- The scheduler is a fundamental subsystem of the kernel
- Different scheduling strategies exist
  - Take into account priority
  - Take into account responsiveness
  - Take into account fairness
- The history of Linux has seen different algorithms

Process Priority

- Unix demands for priority based scheduling
  - This relates to the *nice* of a process in [-20, 19]
  - The higher the nice, the lower the priority
  - This tells how nice a process is towards others
- There is also the notion of "real time" processes
  - Hard real time: bound to strict time limits in which a task must be completed (not supported in mainstream Linux)
  - Soft real time: there are boundaries, but don't make your life depend on it
  - Examples: burning data to a CD ROM, VoIP

Process Priority

- In Linux, real time priorieties are in [0, 99]
  - Here higher value means higher priority
- Implemented according to the Real-Time Extensions of POSIX

```
ps -eo pid,rtprio,cmd  ('-' = no realtime)
chrt -p pid
chrt -p prio pid
```

Process Priority in the Kernel

- Both nice and rt priorities are mapped to a single value in [0, 139] in the kernel
- 0 to 99 are reserved to rt priorities
- 100 to 139 for nice priorities (mapping exactly to [-20, 19])
- Priorities are defined in `include/linux/sched/prio.h`

Process Priority in the Kernel

```
#define MAX_NICE 19
#define MIN_NICE -20
#define NICE_WIDTH (MAX_NICE - MIN_NICE + 1)

#define MAX_USER_RT_PRIO 100
#define MAX_RT_PRIO MAX_USER_RT_PRIO
#define MAX_PRIO (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO (MAX_RT_PRIO + NICE_WIDTH / 2)
```

Process Priority in the Kernel

```
/*
* Convert user-nice values [ -20 ... 0 ... 19 ]
* to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
* and back.
*/
#define NICE_TO_PRIO(nice) ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio) ((prio) - DEFAULT_PRIO)

/*
* 'User priority' is the nice value converted to
something we
* can work with better when scaling various scheduler
parameters,
* it's a [ 0 ... 39 ] range.
*/
#define USER_PRIO(p) ((p)-MAX_RT_PRIO)
#define TASK_USER_PRIO(p) USER_PRIO((p)->static_prio)
#define MAX_USER_PRIO (USER_PRIO(MAX_PRIO))
```

Process Priority in `task_struct`

- `static_prio`: priority given “statically” by a user (and mapped into kernel’s representation)
- `normal_priority`: based on `static_prio` and scheduling policy of a process: Tasks with the same static priority that belong to different policies will get different normal priorities. Child processes inherit the normal priorities from their parent processes when forked.
- `prio`: “dynamic priority”. It can change in certain situations, e.g. to preempt a process with higher priority
- `rt_priority`: the realtime priority for realtime tasks in [0, 99]

Computing `prio`

- In `kernel/sched/core.c`

```
p->prio = effective_prio(p);

static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}
```

Returns `static_priority` or maps `rt_priority` to kernel representation

Load Weights

- `task_struct->se` is a `struct sched_entity` (in `include/linux/sched.h`):
  - It keeps a `struct load_weight load`:

```
struct load_weight {
    unsigned long weight;
    u32 inv_weight;
};
```

- Load weights are used to scale the time slice assigned to a scheduled process

Load Weights

- From `kernel/sched/core.c`:

Nice levels are multiplicative, with a gentle 10% change for every nice level changed. I.e. when a CPU-bound task goes from nice 0 to nice 1, it will get ~10% less CPU time than another CPU-bound task that remained on nice 0.

The "10% effect" is relative and cumulative: from _any_ nice level, if you go up 1 level, it's -10% CPU usage, if you go down 1 level it's +10% CPU usage. (to achieve that we use a multiplier of 1.25. If a task goes up by ~10% and another task goes down by ~10% then the relative distance between them is ~25%.)

Load Weights

- From `kernel/sched/core.c`:

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

- This array takes a value for each possible nice level in [-20, 19]

Some Examples

- Two tasks running at nice 0 (weight 1024)
  - Both get 50% of time: 1024/(1024+1024) = 0.5
- Task 1 is moved to nice -1 (priority boost):
  - T1: 1277/(1024+1277) ≈ 0.55
  - T2: 1024/(1024+1277) ≈ 0.45 (10% difference)
- Task 2 is then moved to nice 1 (priority drop):
  - T1: 1277/(820+1277) ≈ 0.61
  - T2: 820/(820+1277) ≈ 0.39 (22% difference)

Different Scheduling Classes

- `SCHED_FIFO`: Realtime FIFO scheduler, in which a process has to explicitly yield the CPU
- `SCHED_RR`: Realtime Round Robin Scheduler (might fallback to FIFO)
- `SCHED_OTHER`/`SCHED_NORMAL`: the common round-robin time-sharing scheduling policy
- `SCHED_DEADLINE` (since 3.14): Constant Bandwidth Server (CBS) algorithm on top of Earliest Deadline First queues
- `SCHED_DEADLINE` (since 4.13): CBS replaced with Greedy Reclamation of Unused Bandwidth (GRUB).

Scheduling Classes

```
struct sched_class {
  const struct sched_class *next;
  void (*enqueue_task) (struct rq *rq, struct task_struct *p, int
                      flags);
  void (*dequeue_task) (struct rq *rq, struct task_struct *p, int
                      flags);
  void (*yield_task) (struct rq *rq);

  void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int
                      flags);
  struct task_struct * (*pick_next_task) (struct rq *rq, struct
                      task_struct *prev, struct rq_flags *rf);
  void (*put_prev_task) (struct rq *rq, struct task_struct *p);
  ...
  void (*set_curr_task) (struct rq *rq);
  int  (*select_task_rq)(struct task_struct *p, int task_cpu,
                        int sd_flag, int flags);
  ...
};
```

Scheduler Code Organization

- General code base and specific scheduler classes are found in `kernel/sched/`
- `core.c`: the common codebase
- `fair.c`: implementation of the basic scheduler (CFS: Completely Fair Scheduler)
- `rt.c`: the real-time scheduler
- `idle_task.c`: the idle-task class

Run Queues

```
struct rq {
      unsigned int nr_running;
      #define CPU_LOAD_IDX_MAX 5
      unsigned long cpu_load[CPU_LOAD_IDX_MAX];
      /* capture load from all tasks on this cpu */
      struct load_weight load;
      struct cfs_rq cfs;
      struct rt_rq rt;
      struct task_struct *curr, *idle, ...;
      u64 clock;
      /* cpu of this runqueue */
      int cpu;
}
```

Run Queues

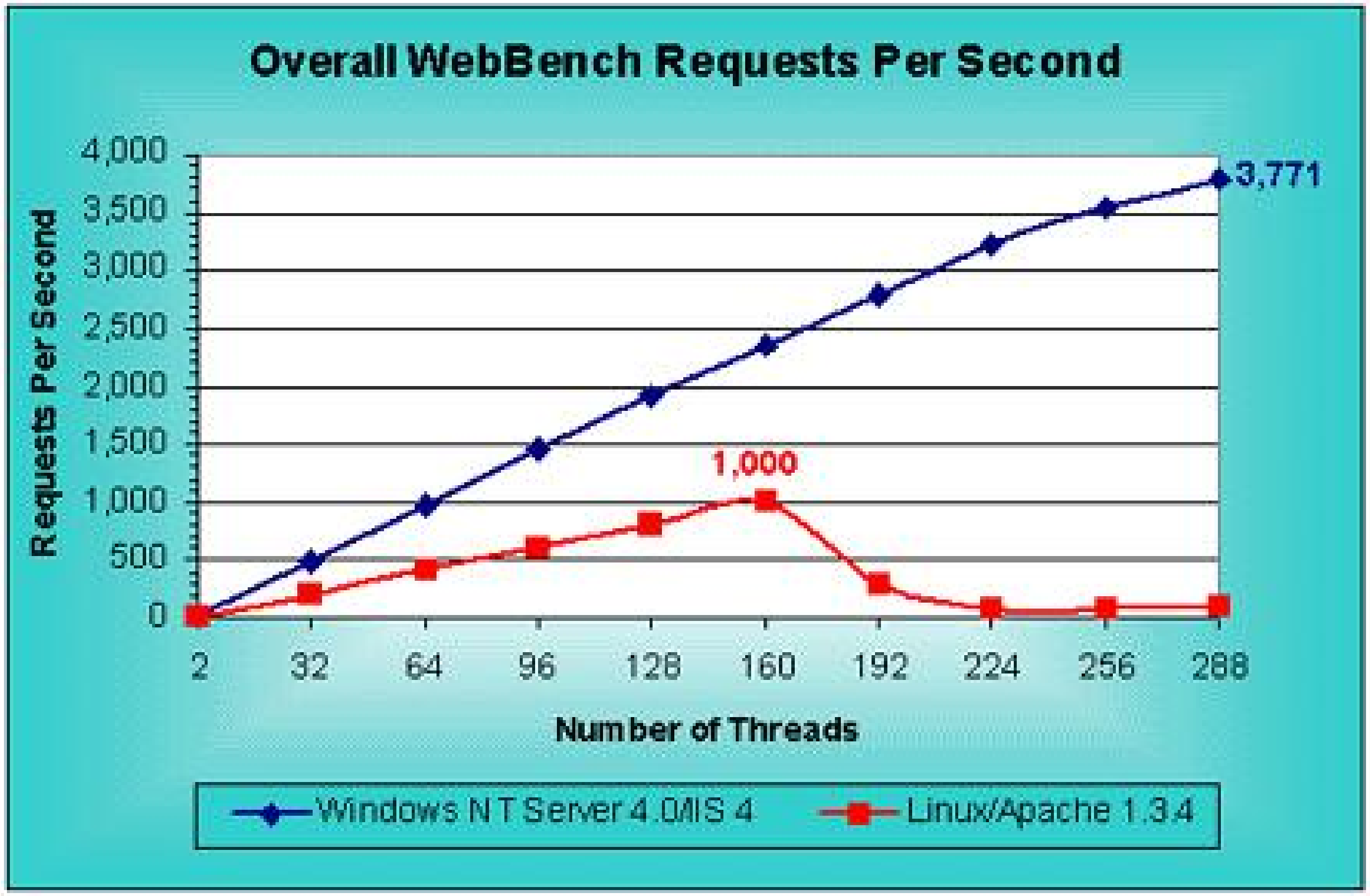
- Added in 2.6
- Defined in `kernel/sched/sched.h`

```
DECLARE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);

#define cpu_rq(cpu)    (&per_cpu(runqueues, (cpu)))
#define this_rq()      this_cpu_ptr(&runqueues)
#define task_rq(p)     cpu_rq(task_cpu(p))
#define cpu_curr(cpu) (cpu_rq(cpu)->curr)
```

Wait Queues

- Defined in `include/linux/wait.h`
- This is a set of data structures to manage threads that are waiting for some condition to become true
- This is a way to put threads to sleep in kernel space
- It is a data structure which changed many times in the history of the kernel
- Suffered from the "Thundering Herd" performance problem

Thundering Herd Effect



Taken from 1999 Mindcraft study on Web and File Server Comparison

Wait Queues

```
#define WQ_FLAG_EXCLUSIVE       0x01

struct wait_queue_entry {
      unsigned int        flags;
      void                *private;
      wait_queue_func_t   func;
      struct list_head    entry;
};

struct wait_queue_head {
      spinlock_t          lock;
      struct list_head    head;
};
typedef struct wait_queue_head wait_queue_head_t;
```

Wait Queue API

- **Implemented as macros in** `include/linux/wait.h`

```
static inline void init_waitqueue_entry(struct
wait_queue_entry *wq_entry, struct task_struct
*p)
wait_event_interruptible(wq_head, condition)
wait_event_interruptible_timeout(wq_head,
condition, timeout)
wait_event_hrtimeout(wq_head, condition,
timeout)
wait_event_interruptible_hrtimeout(wq,
condition, timeout)
```

Wait Queue API

```
void add_wait_queue(struct wait_queue_head *wq_head,
struct wait_queue_entry *wq_entry) {
     unsigned long flags;

     wq_entry->flags &= ~WQ_FLAG_EXCLUSIVE;
     spin_lock_irqsave(&wq_head->lock, flags);
     list_add(&wq_entry->entry, &wq_head->head);
     spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

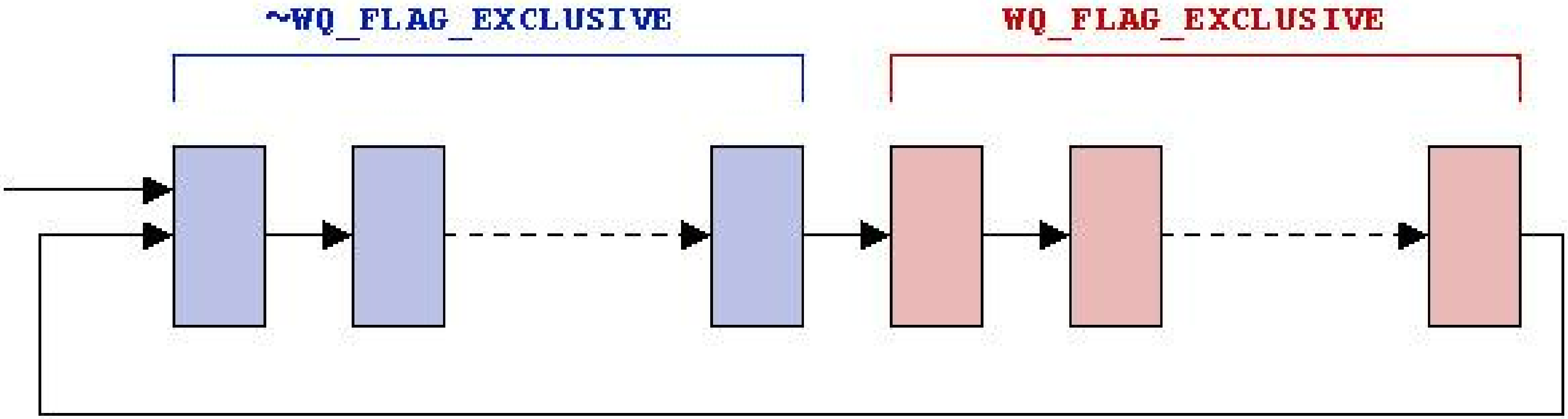
Wait Queue API

```
void add_wait_queue_exclusive(struct wait_queue_head
*wq_head, struct wait_queue_entry *wq_entry) {
     unsigned long flags;

     wq_entry->flags |= WQ_FLAG_EXCLUSIVE;
     spin_lock_irqsave(&wq_head->lock, flags);
     list_add_tail(&wq_entry->entry, &wq_head->head);
     spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Wait Queue API

```
void remove_wait_queue(struct wait_queue_head
*wq_head, struct wait_queue_entry *wq_entry) {
       unsigned long flags;

       spin_lock_irqsave(&wq_head->lock, flags);
       list_del(&wq_entry->entry);
       spin_unlock_irqrestore(&wq_head->lock, flags);
}
```

Wait Queue Exclusive



Wait Queue API

- Implemented as macros in `include/linux/wait.h`
- `wake_up(x)`
- `wake_up_nr(x, nr)`
- `wake_up_all(x)`
- `wake_up_locked(x)`
- `wake_up_all_locked(x)`

- `wake_up_interruptible(x)`
- `wake_up_interruptible_nr(x, nr)`
- `wake_up_interruptible_all(x)`
- `wake_up_interruptible_sync(x)`

Thread States

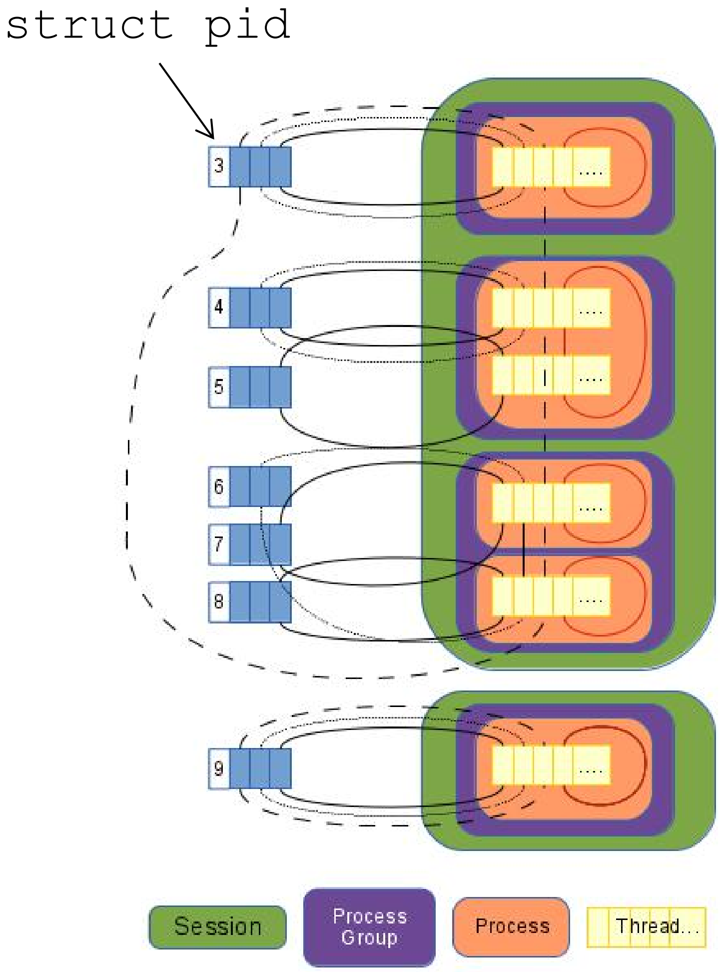
- The `state` field n the PCB tracks the current state of the process/thread
- Values are defined in `inlude/linux/sched.h`
  - `TASK_RUNNING`
  - `TASK_INTERRUPTIBLE`
  - `TASK_UNINTERRUPTIBLE`
  - `TASK_ZOMBIE`
  - `TASK_STOPPED`
  - `TASK_KILLABLE`
- All the PCBs registered in the runqueue are `TASK_RUNNING`

Accessing PCBs

- In some circumstances, the kernel must derive the `task_struct` given the PID of a process
  - Think for example of the `kill()` system call
- Scanning a list of PCBs is inefficient
- There are multiple hash tables available

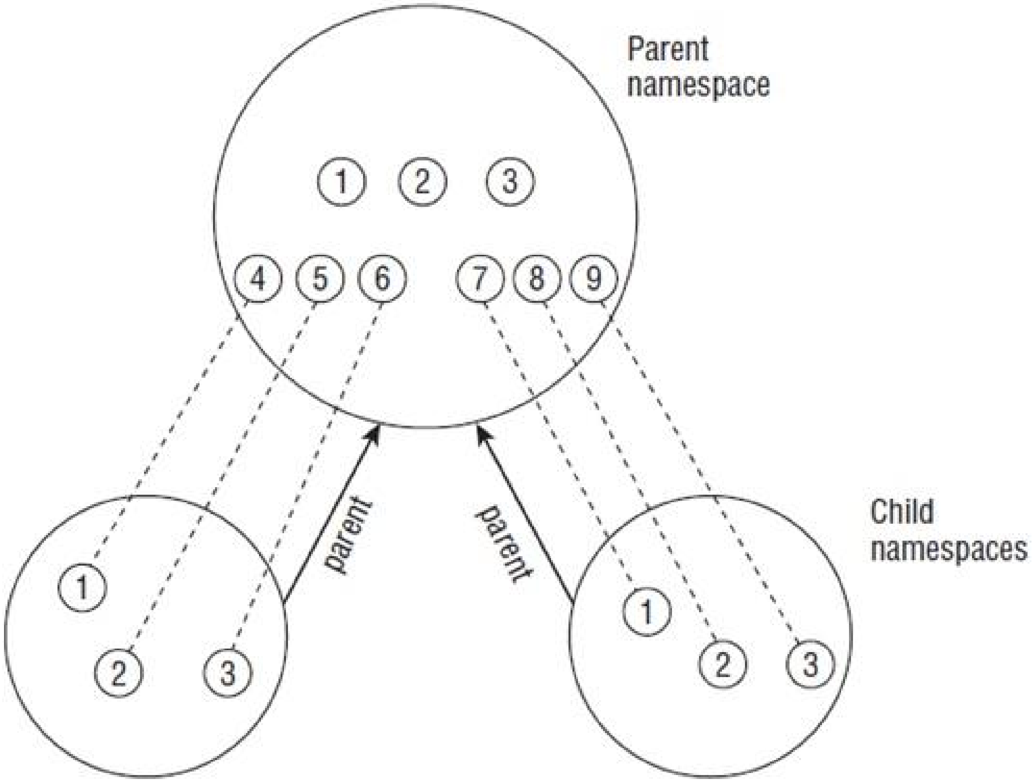
Hash table type	Field name	Description
PIDTYPE_PID	pid	PID of the process
PIDTYPE_TGID	tgid	PID of thread group leader process
PIDTYPE_PGID	pgrp	PID of the group leader process
PIDTYPE_SID	session	PID of the session leader process

PID Relations



- `task_struct` maps to Thread (beware of the overload of the word "task")
- Process groups can be used to avoid scanning the whole PID list
- `struct pid` links together pids in the namespace world

PID Namespaces



A new PID namespace is created by calling `clone()` with the `CLONE_NEWPID` flag

`struct pid`

- `struct pid` is the kernel's internal notion of a process identifier. It refers to individual tasks, process groups, and sessions. While there are processes attached to it the `struct pid` lives in a hash table, so it and then the processes that it refers to can be found quickly from the numeric pid value. The attached processes may be quickly accessed by following pointers from struct pid.
- Storing `pid_t` values in the kernel and refering to them later has a problem. The process originally with that pid may have exited and the pid allocator wrapped, and another process could have come along and been assigned that pid.
- Referring to user space processes by holding a reference to `struct task_struct` has a problem. When the user space process exits the now useless `task_struct` is still kept. A `task_struct` plus a stack consumes around 10K of low kernel memory. More precisely this is `THREAD_SIZE + sizeof(struct task_struct).` By comparison a `struct pid` is about 64 bytes.
- Holding a reference to `struct pid` solves both of these problems. It is small so holding a reference does not consume a lot of resources, and since a new struct pid is allocated when the numeric pid value is reused (when pids wrap around) we don't mistakenly refer to new processes.

`struct pid`

- Defined in `include/linux/pid.h`

```
struct pid {
      atomic_t count;
      unsigned int level;
      /* lists of tasks that use this pid */
      struct hlist_head tasks[PIDTYPE_MAX];
      struct rcu_head rcu;
      struct upid numbers[1];
};

enum pid_type { PIDTYPE_PID, PIDTYPE_PGID, PIDTYPE_SID,
                PIDTYPE_MAX };
```

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)

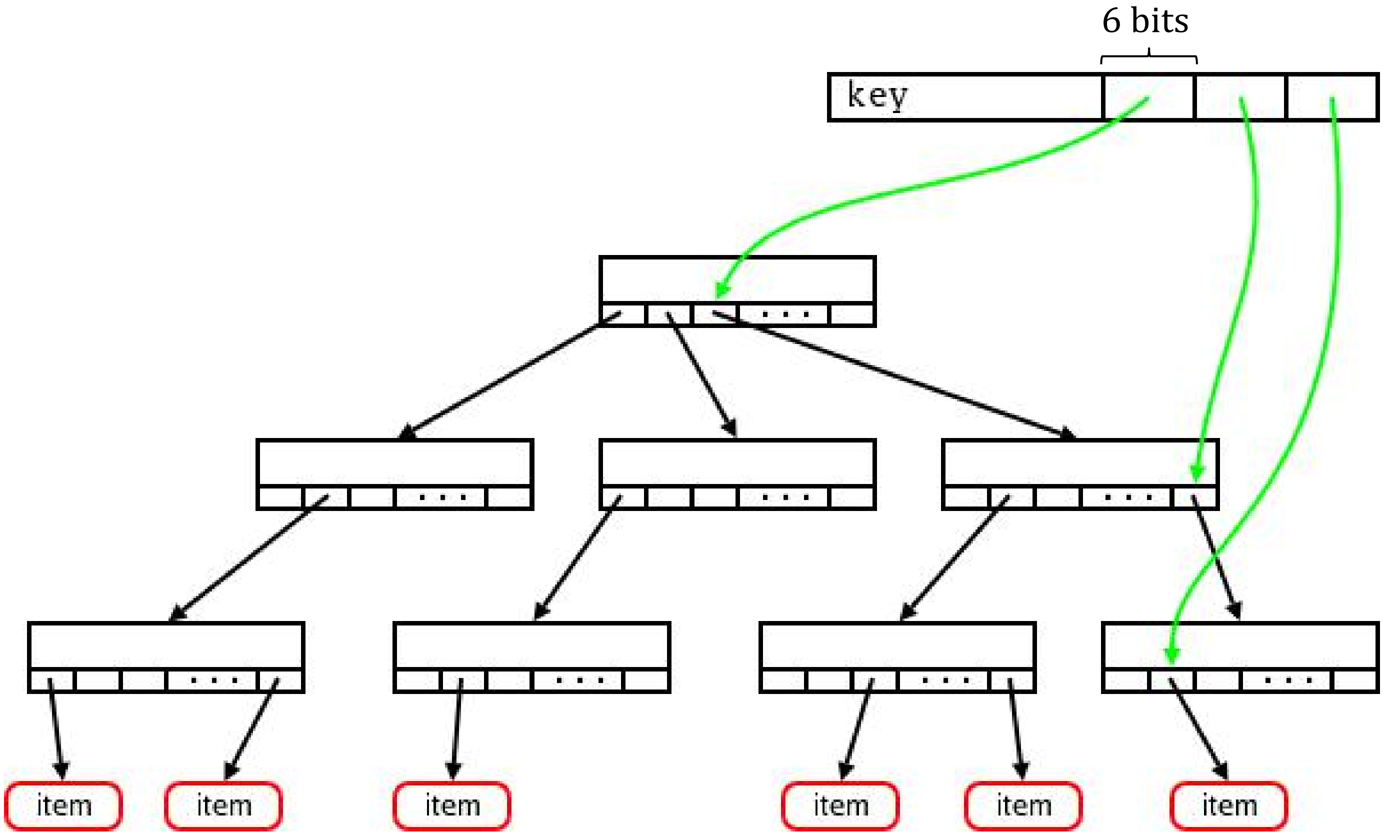
```
/* PID hash table linkage. */
struct task_struct *pidhash_next;
struct task_struct **pidhash_pprev;
```

- 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

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

Radix Trees



Radix Tree API is in `linux/radix-tree.h`

Scheduler Entry Point

- The entry point for the scheduler is schedule(void) in `kernel/sched.c`
- This is called from several places in the kernel
  - *Direct Invocation*: an explicit call to `schedule()` is issued
  - *Lazy Invocation*: some hint is given to the kernel indicating that `schedule()` should be called soon (see `need_resched`)
- In general `schedule()` entails 3 distinct phases, which depend on the scheduler implementation:
  - Some checks on the current process (e.g., with respect to signal processing)
  - Selection of the process to be activated
  - Context switch

Periodic Scheduling

- `schedule_tick()` is called from `update_process_times()`
- This function has two goals:
  - Managing scheduling-specific statistics
  - Calling the scheduling method of the class

schedule_tick()

```
/*
* This function gets called by the timer code, with HZ
frequency.
* We call it with interrupts disabled.
*/
void scheduler_tick(void) {
      int cpu = smp_processor_id();
      struct rq *rq = cpu_rq(cpu);
      struct task_struct *curr = rq->curr;
      ...
      update_rq_clock(rq);
      curr->sched_class->task_tick(rq, curr, 0);
      update_cpu_load_active(rq);
      ...
}
```

Process Going to Sleep

- In case an operation cannot be completed immediately (think of a `read()`) the process goes to sleep in a wait queue
- While doing this, the task enters either the `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` state
- At this point, the kernel thread calls `schedule()` to effectively put to sleep the currently-running one and pick the new one to be activated

More on `TASK_*INTERRUPTIBLE`

- Dealing with `TASK_INTERRUPTIBLE` can be difficult:
  - At kernel level, understand that the task has been resumed due to an interrupt
  - Clean up all the work that has been done so far
  - Return to userspace with `-EINTR`
  - Userspace has to understand that a syscall was interrupted (bugs here!)
- Conversely, a `TASK_UNINTERRUPTIBLE` might never be woken up again (the dreaded D state in ps)
- `TASK_KILLABLE` is handy for this (since 2.6.25)
  - Same as `TASK_UNINTERRUPTIBLE` except for fatal sigs.

Sleeping Task Wakes Up

- The event a task is waiting for calls one of the `wake_up*()` functions on the corresponding wait queue
- A task is set to runnable and put back on a runqueue
- It the woken up task has a higher priority than the other tasks on the runqueue, `TIF_NEED_RESCHED` is flagged

O(n) Scheduler (2.4)

- It has a linear complexity, as it iterates over all tasks
- Time is divided into *epochs*
- At the end of an epoch, every process has run once, using up its whole quantum if possible
- If processes did not use the whole quantum, they have half of the remaining timeslice added to the new timeslice

O(n) Scheduler (2.4)

```
asmlinkage void schedule(void) {
  int this_cpu, c; /* weight */
  ...
 repeat_schedule:
  /* Default process to select.. */
  next = idle_task(this_cpu);
  c = -1000; /* weight */
  list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
      int weight = goodness(p, this_cpu, prev->active_mm);
      if (weight > c)
        c = weight, next = p;
    }
  }
}
```

Computing the Goodness

goodness (p)= 20 – `p->nice` (base time quantum)

+ `p->counter` (ticks left in time quantum)

+1 (if page table is shared with the previous process)

+15 (in SMP, if p was last running on the same CPU)

Computing the Goodness

- Goodness values explained and special cases:
  - -1000: never select this process to run
  - 0: out of timeslice (`p->counter == 0`)
  - >0: the goodness value, the larger the better
  - +1000: a realtime process, select this

Epoch Management

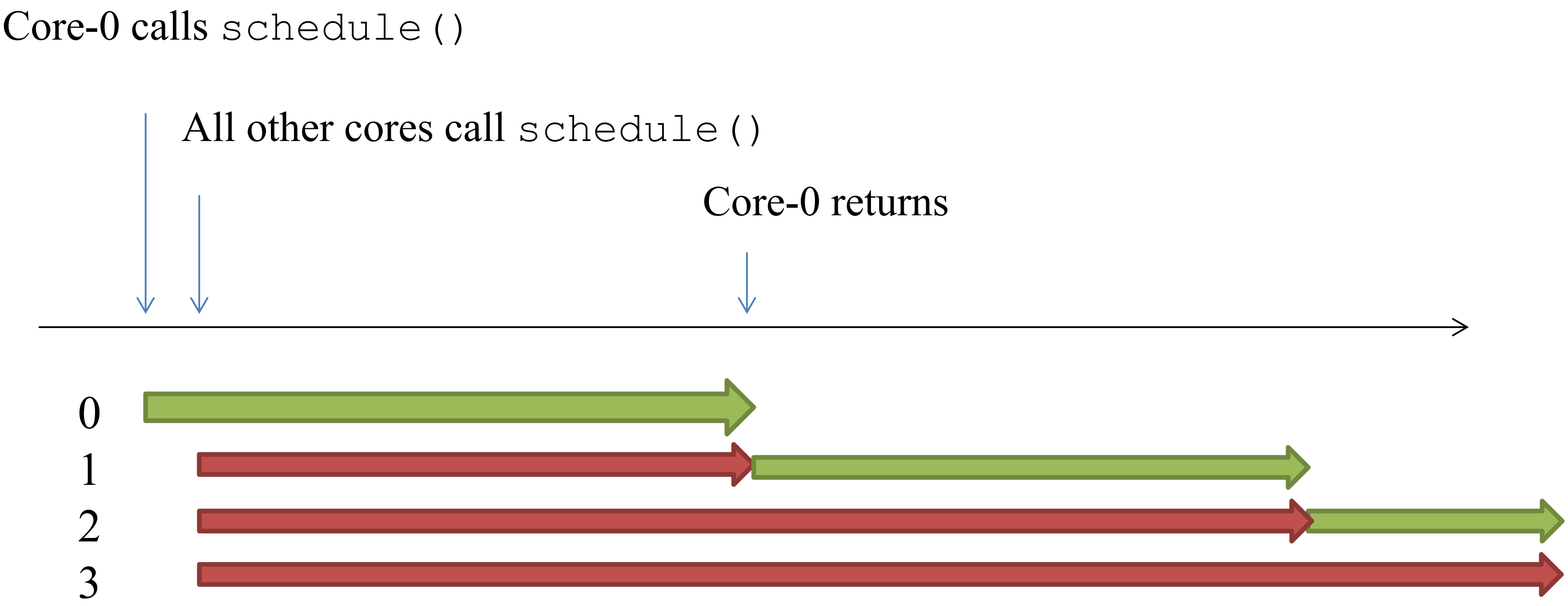
```
................
/* Do we need to re-calculate counters? */
if (unlikely(!c)) {
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
                p->counter = (p->counter >> 1) +
                                NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
................
```

`6 - p->nice/4`

Analysis of the O(n) Scheduler

- Disadvantages:
  - A non-runnable task is also searched to determine its goodness
  - Mixture of runnable/non-runnable tasks into a single runqueue in any epoch
  - Performance problems on SMP, as the length of critical sections depends on system load
- Advantages:
  - Perfect Load Sharing
  - No CPU underutilization for any workload type
  - No (temporary) binding of threads to CPUs

Contention in the O(n) Scheduler on SMP



Core-0 calls `schedule()`

All other cores call `schedule()`

Core-0 returns

O(1) Scheduler (2.6.8)

- By Ingo Molnár
- Schedules tasks in constant time, indepentendly of the number of active processes
- Introduced the global priority scale which we discussed
- Early preëmption: if a task enters the `TASK_RUNNING` state its priority is checked to see whether to call `schedule()`
- Static priority for real-time tasks
- Dynamic priority for other tasks, recalculated at the end of their timeslice (increases interactivity)

Runqueue Revisited

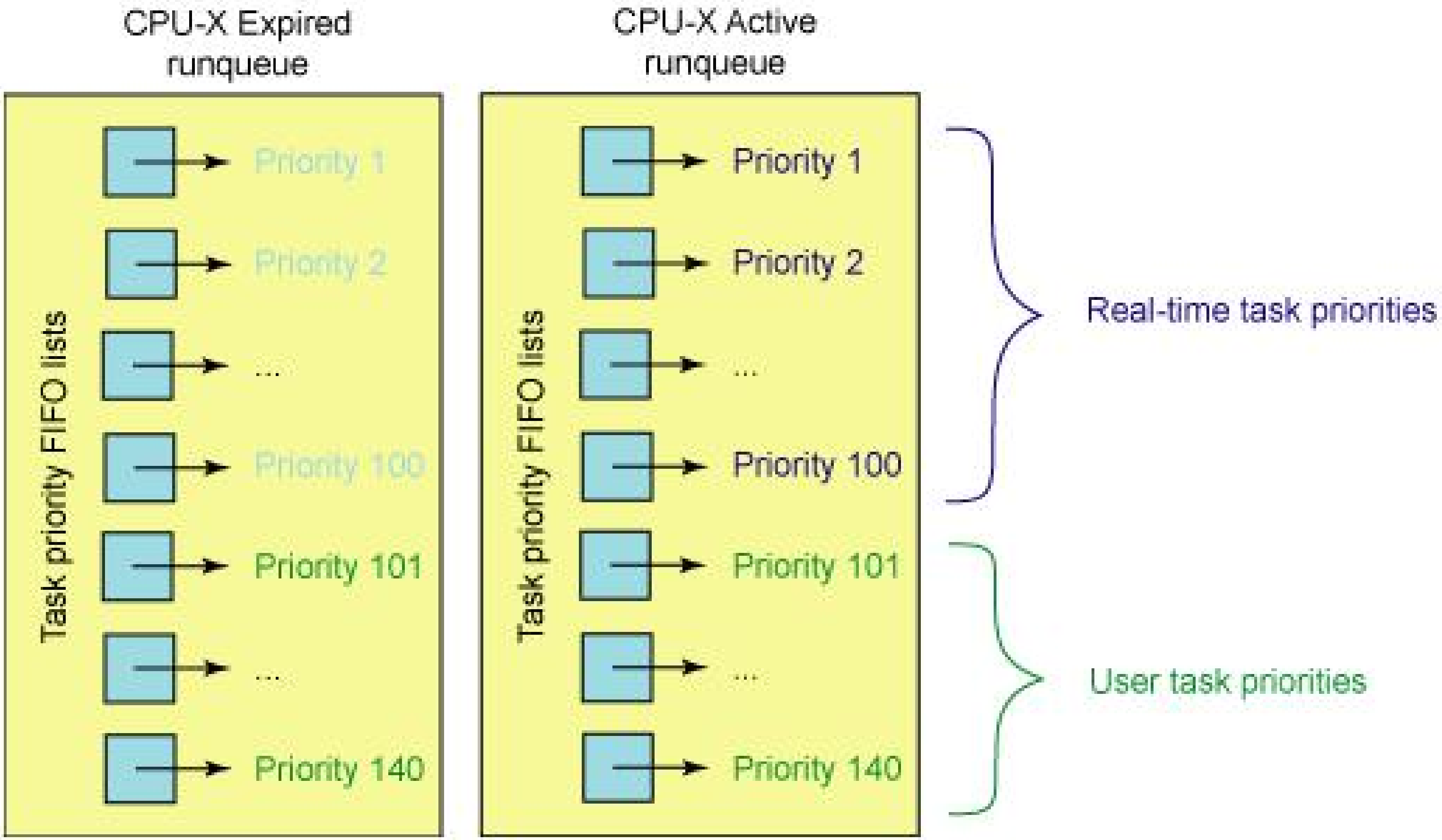
```
struct runqueue {
    /* number of runnable tasks */
    unsigned long nr_running;
    ...
    struct prio_array *active;
    struct prio_array *expired;
    struct prio_array arrays[2];
}
```

Runqueue Revisited

- Each runqueue has two `struct prio_array`:

```
struct prio_array {
    int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```

Runqueue Revisited

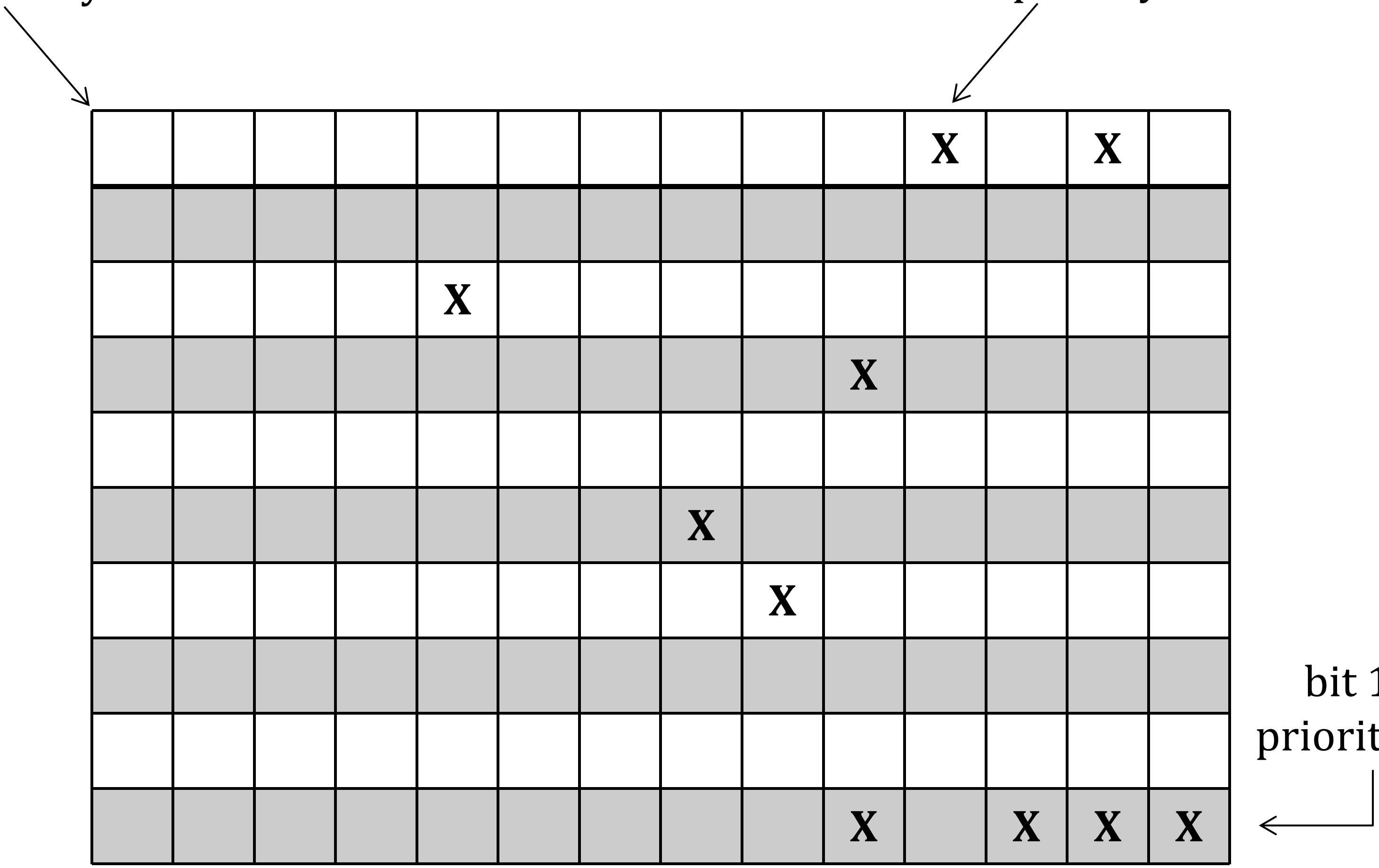


Runqueue Revisited

`schedule()` → `schedule_find_first_set()`

bit 0, priority 0

bit 10, priority 10



bit 139
priority 139

Cross-CPU Scheduling

- Once a task lands on a CPU, it might use up its timeslice and get put back on a prioritized queue for rerunning—but how might it ever end up on another processor?
- If all the tasks on one CPU exit, might not one processor stand idle while another round-robins three, ten or several dozen other tasks?
- The 2.6 scheduler must, on occasion, see if cross-CPU balancing is needed.
- Every 200ms a CPU checks to see if any other CPU is out of balance and needs to be balanced with that processor. If the processor is idle, it checks every 1ms so as to get started on a real task earlier

2.6 O(1) Scheduler API

Function name	**Function description**
`schedule`	The main scheduler function. Schedules the highest priority task for execution.
`load_balance`	Checks the CPU to see whether an imbalance exists, and attempts to move tasks if not balanced.
`effective_prio`	Returns the effective priority of a task (based on the static priority, but includes any rewards or penalties).

2.6 O(1) Scheduler API

`recalc_task_prio`	Determines a task's bonus or penalty based on its idle time.
`source_load`	Conservatively calculates the load of the source CPU (from which a task could be migrated).
`target_load`	Liberally calculates the load of a target CPU (where a task has the potential to be migrated).
`migration_thread`	High-priority system thread that migrates tasks between CPUs.

Stack Variables Refresh

```
asmlinkage void __sched schedule(void)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq *rq;
        int cpu;

need_resched:
        preempt_disable();
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        rcu_qsctr_inc(cpu);
        prev = rq->curr;
        switch_count = &prev->nivcsw;
...
```

Stack Variables Refresh

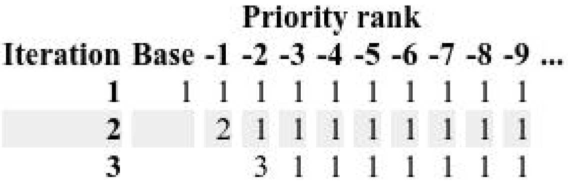
```
...
if (unlikely(!rq->nr_running)) idle_balance(cpu, rq);

prev->sched_class->put_prev_task(rq, prev);
next = pick_next_task(rq, prev);

if (likely(prev != next)) {
        sched_info_switch(prev, next);

        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        context_switch(rq, prev, next); /* unlocks the rq */
        /* the context switch might have flipped the stack from under
           us, hence refresh the local variables. */
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
} else spin_unlock_irq(&rq->lock);
...
```

Staircase Scheduler

- By Con Kolivar, 2004 (none of its schedulers in the official Kernel tree)
- The goal is to increase "responsiveness" and reduce the complexity of the O(1) Scheduler
- It is mostly based on dropping the priority recalculation, replacing it with a simpler rank-based scheme
- It is supposed to work better up to ~10 CPUs (tailored for desktop environments)

Staircase Scheduler

- The expired array is removed and the staircase data structure is used instead



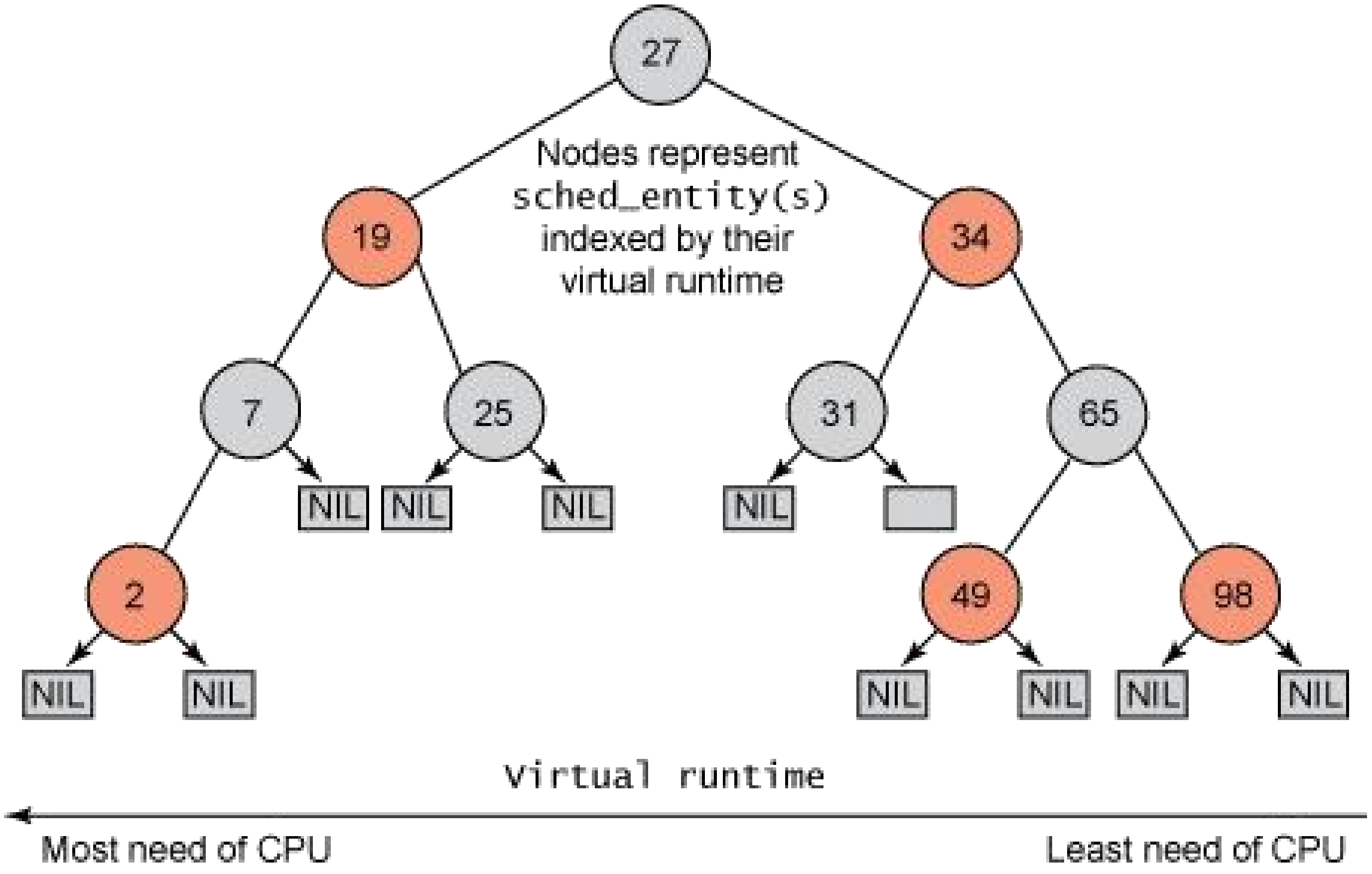
	Priority rank										
Iteration	Base	-1	-2	-3	-4	-5	-6	-7	-8	-9	...
1	1	1	1	1	1	1	1	1	1	1	
2		2	1	1	1	1	1	1	1	1	
3			3	1	1	1	1	1	1	1	

- A process expiring its timeslice is moved to a lower priority
- At the end of the staircase, it gets to a MAX_PRIO-1 level with one more timeslice
- If a process sleeps (i.e., an interactive process) it get backs up in the staircase
- This approach favors interactive processes rather CPU-bound ones

Completely Fair Scheduler (2.6.23)

- Merged in October 2007
- This is since then the default Scheduler
- This models an "ideal, precise multitasking CPU" on real hardware
- It is based on a red-black tree, where nodes are ordered by process execution time in nanoseconds
- A maximum execution time is also calculated for each process

Completely Fair Scheduler (2.6.23)



Nodes represent `sched_entity(s)` indexed by their virtual runtime

`Virtual runtime`

Most need of CPU

Least need of CPU

Context switch (2.4)

- Context switch is implemented in the `switch_to()` macro in `include/asm-i386/system.h`
- It jumps to `void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)` in `arch/i386/kernel/process.c`
- The macro is machine-dependent code
- `__switch_to()` mainly executes the following two tasks
  - TSS update
  - CPU control registers update

switch_to()

```
#define switch_to(prev,next,last) do {                       \
       asm volatile("pushl %%esi\n\t"                        \
               "pushl %%edi\n\t"                      \
               "pushl %%ebp\n\t"                      \
               "movl %%esp,%0\n\t"       /* save ESP */       \
               "movl %3,%%esp\n\t"       /* restore ESP */    \
               "movl $1f,%1\n\t"         /* save EIP */\
               "pushl %4\n\t"            /* restore EIP */    \
               "jmp __switch_to\n"                            \
               "1:\t"                                         \
               "popl %%ebp\n\t"                       \
               "popl %%edi\n\t"                       \
               "popl %%esi\n\t"                       \
               :"=m" (prev->thread.esp),"=m" (prev->thread.eip),\
                "=b" (last)                                   \
               :"m" (next->thread.esp),"m" (next->thread.eip),\
                 "a" (prev), "d" (next),              \
                "b" (prev));                                  \
} while (0)
```

__switch_to()

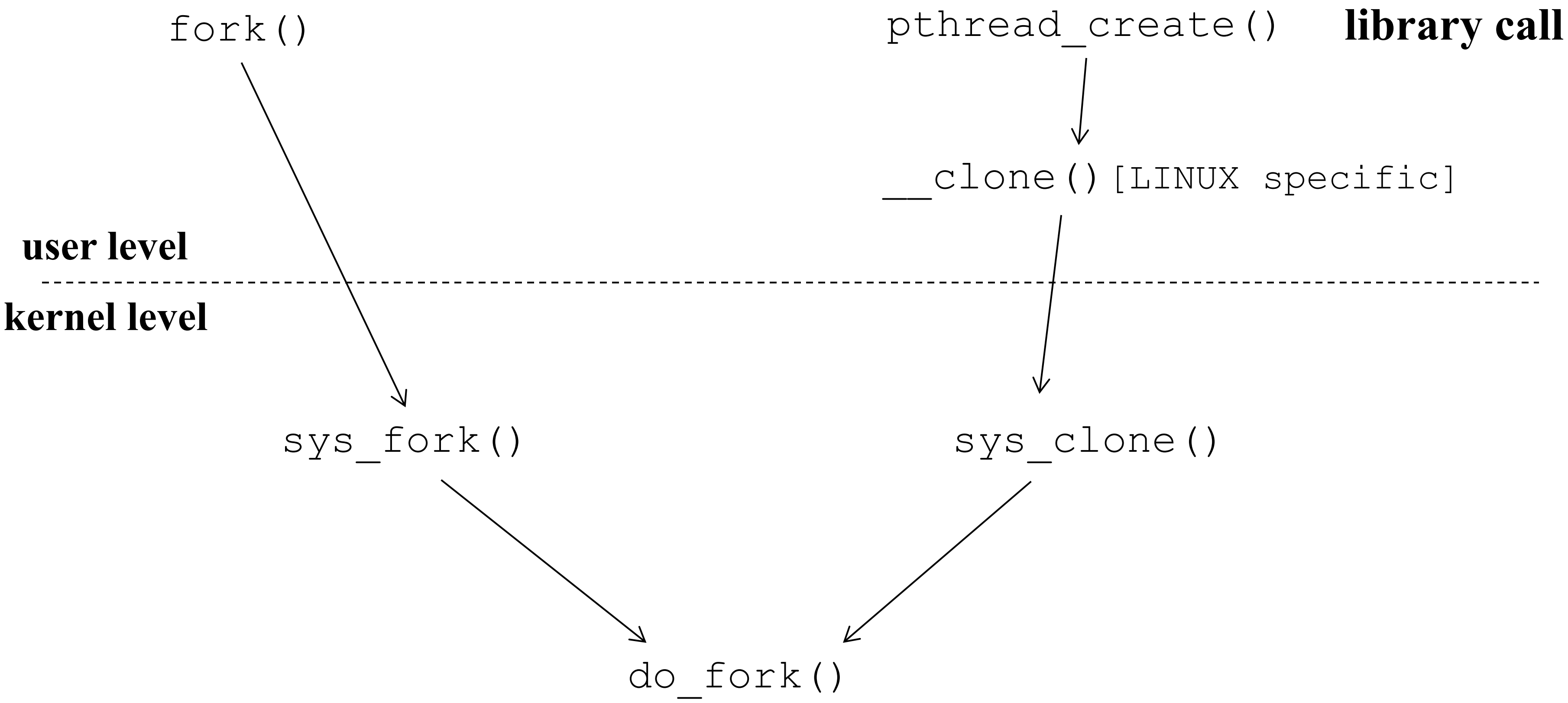
```
void __switch_to(struct task_struct *prev_p,
                           struct task_struct *next_p){

       struct thread_struct *prev = &prev_p->thread,
                            *next = &next_p->thread;
       struct tss_struct *tss = init_tss + smp_processor_id();
       ……

       /* Reload esp0, LDT and the page table pointer: */
       tss->esp0 = next->esp0;

       /*  Save away %fs and %gs. No need to save %es and %ds, as
        * those are always kernel segments while inside the kernel
        */
       asm volatile("movl %%fs,%0":"=m" (*(int *)&prev->fs));
       asm volatile("movl %%gs,%0":"=m" (*(int *)&prev->gs));

       /* Restore %fs and %gs. */
       loadsegment(fs, next->fs);
       loadsegment(gs, next->gs);
       ……
}
```

`fork()` initialization

- Initialization of the fork subsystem occurs via `fork_init()` in `kernel/fork.c`
- This sets some fields of the idle process PCB to values inherited by other processes

```
void __init fork_init(unsigned long mempages){
       /*
        * The default maximum number of threads is set to a safe
        * value: the thread structures can take up at most half
        * of memory.
        */
       max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 8;

       init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
       init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
}
```

Process and thread creation



`fork()`

`pthread_create()`

library call

`__clone()[LINUX specific]`

user level

kernel level

`sys_fork()`

`sys_clone()`

`do_fork()`

`sys_fork()` and `sys_clone()`

```
asmlinkage int sys_fork(struct pt_regs regs)
{
      return do_fork(SIGCHLD, regs.esp, &regs, 0);
}

asmlinkage int sys_clone(struct pt_regs regs)
{
      unsigned long clone_flags;
      unsigned long newsp;

      clone_flags = regs.ebx;
      newsp = regs.ecx;
      if (!newsp)
            newsp = regs.esp;
      return do_fork(clone_flags, newsp, &regs, 0);
}
```

Calling `sys_clone()` from Userspace

- When usign `sys_clone()`, we must allocate a new stack first
- Indeed, a thread of the same process share the same address space
- The VA base of the new stack must be passed into ecx right before giving control to `sys_clone()`
- Thread activation flags must be passed in ebx
- The documented `__clone()` is thus a wrapper of the actual system call

`do_fork()` (again)

- Fresh PCB/kernel-stack allocation
- Copy/setup of PCB information
- Copy/setup of PCB linked data structures
- What information is copied or inherited (namely shared into the original buffers) depends on the value of the flags passed in input to do_fork()
- Admissible 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() (2.4)

```
int do_fork(unsigned long clone_flags, unsigned long stack_start,
           struct pt_regs *regs, unsigned long stack_size)
{
       ……
       p = alloc_task_struct();
       if (!p) goto fork_out;
       *p = *current;


       ……
       p->state = TASK_UNINTERRUPTIBLE;
       ……
       p->pid = get_pid(clone_flags);
       if (p->pid == 0 && current->pid != 0)
              goto bad_fork_cleanup;

       p->run_list.next = NULL;
       p->run_list.prev = NULL;
       ……
       init_waitqueue_head(&p->wait_chldexit);
       ……
```

do_fork() (2.4)

```
p->sigpending = 0;
init_sigpending(&p->pending);
...
p->start_time = jiffies;
...
/* copy all the process information */
if (copy_files(clone_flags, p))   goto bad_fork_cleanup;
if (copy_fs(clone_flags, p))      goto bad_fork_cleanup_files;
if (copy_sighand(clone_flags, p)) goto bad_fork_cleanup_fs;
if (copy_mm(clone_flags, p))      goto bad_fork_cleanup_sighand;
retval = copy_namespace(clone_flags, p);
if (retval)   goto bad_fork_cleanup_mm;
retval = copy_thread(0, clone_flags, stack_start,
                                  stack_size, p, regs);
if (retval) goto bad_fork_cleanup_namespace;
p->semundo = NULL;

...
p->exit_signal = clone_flags & CSIGNAL;
...
```

do_fork() (2.4)

```
/* "share" dynamic priority between parent and child thus
 * the total amount of dynamic priorities in the system
 * doesn't change, more scheduling fairness. This is only
 * important in the first timeslice, on the long run
 * the scheduling behaviour is unchanged. */
p->counter = (current->counter + 1) >> 1;
current->counter >>= 1;
if (!current->counter)
        current->need_resched = 1;

/*
 * Ok, add it to the run-queues and make it
 * visible to the rest of the system.
 *
 * Let it rip!
 */
retval = p->pid;
...
```

do_fork() (2.4)

```
    /* Need tasklist lock for parent etc handling! */
    write_lock_irq(&tasklist_lock);

    /* CLONE_PARENT re-uses the old parent */
    p->p_opptr = current->p_opptr;
    p->p_pptr = current->p_pptr;
    if (!(clone_flags & CLONE_PARENT)) {
            p->p_opptr = current;
            if (!(p->ptrace & PT_PTRACED))
                    p->p_pptr = current;
    }
    ……
    SET_LINKS(p);
    hash_pid(p);
    nr_threads++;
    write_unlock_irq(&tasklist_lock);
    ……
    wake_up_process(p);          /* do this last */
    ++total_forks;
    ……
fork_out:
    return retval;
    ……
}
```

`copy_thread()` (2.4)

- Part of the job of `do_fork()` is carried out by the `copy_thread()` function in `arch/i386/kernel/process.c`
- This function prepares the PCB so that the user-level stack pointer is correctly initialized
- It also sets up the return value (zero) for the `clone()` system call thus indicating whether we are running into the child process/thread

copy_thread() (2.4)

```
int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
        unsigned long unused,
        struct task_struct * p, struct pt_regs * regs)
{
  struct pt_regs * childregs;

  childregs = ((struct pt_regs *) (THREAD_SIZE + (unsigned long) p)) - 1;
  struct_cpy(childregs, regs);
  childregs->eax = 0;
  childregs->esp = esp;

  p->thread.esp = (unsigned long) childregs;
  p->thread.esp0 = (unsigned long) (childregs+1);

  p->thread.eip = (unsigned long) ret_from_fork;

  savesegment(fs,p->thread.fs);
  savesegment(gs,p->thread.gs);

  unlazy_fpu(current);
  struct_cpy(&p->thread.i387, &current->thread.i387);

  return 0;
}
```

copy_mm() (2.4)

```
static int copy_mm(unsigned long clone_flags,
                                 struct task_struct * tsk)
{
        struct mm_struct * mm, *oldmm;
        int retval;
        ……
        tsk->mm = NULL;
        tsk->active_mm = NULL;
        ……
        oldmm = current->mm;
        ……
        if (clone_flags & CLONE_VM) {
                atomic_inc(&oldmm->mm_users);
                mm = oldmm;
                goto good_mm;
        }

        retval = -ENOMEM;
        mm = allocate_mm();
        if (!mm)
                goto fail_nomem;
```

copy_mm() (2.4)

```
        /* Copy the current MM stuff.. */
        memcpy(mm, oldmm, sizeof(*mm));
        if (!mm_init(mm)) goto fail_nomem;
        ……

        down_write(&oldmm->mmap_sem);
        retval = dup_mmap(mm);
        up_write(&oldmm->mmap_sem);

        if (retval) goto free_pt;
    // child gets a private LDT if there was an LDT in the parent
        copy_segments(tsk, mm);
good_mm:
        tsk->mm = mm;
        tsk->active_mm = mm;
        return 0;

free_pt:
        mmput(mm);
fail_nomem:
        return retval;
}
```

Support Functions for `copy_mm()`

- in `kernel/fork.c`
  - `mm_init()`
    - Allocation of a fresh PGD
  - `dup_mmap()`
    - Sets up any information for memory management within the new process context

`mm_init()` (2.4)

```
static struct mm_struct * mm_init(struct mm_struct
                                                * mm)
{
       atomic_set(&mm->mm_users, 1);
       atomic_set(&mm->mm_count, 1);
       init_rwsem(&mm->mmap_sem);
       mm->page_table_lock = SPIN_LOCK_UNLOCKED;
       mm->pgd = pgd_alloc(mm);
       mm->def_flags = 0;
       if (mm->pgd)
              return mm;
       free_mm(mm);
       return NULL;
}
```

Notes on `mm_init()`

- `pgd_alloc()` in `include/asm-i386/pgalloc.h` allocates a frame for the PGD and:
  - Resets the PGD (the first 768 entries) for the portion associated with user space addressing (0-3 GB)
  - Copies into it kernel-level addressing information from the current process PGD (from entry 768)
  - This is implemented in `get_pgd_slow()` in `include/asm-i386/pgalloc.h`

dup_mmap() (2.4)

```
static inline int dup_mmap(struct mm_struct * mm)
{
   struct vm_area_struct * mpnt, *tmp, **pprev;
   int retval;
   ...
   mm->mmap = NULL;
   mm->mmap_cache = NULL;
   mm->map_count = 0;
   ...
   pprev = &mm->mmap;
   ...
    for (mpnt = current->mm->mmap ; mpnt ; mpnt = mpnt->vm_next){
      ...
         tmp = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
         if (!tmp)  goto fail_nomem;
         *tmp = *mpnt;
         tmp->vm_flags &= ~VM_LOCKED;
         tmp->vm_mm = mm;
         tmp->vm_next = NULL;
         ...
         retval = copy_page_range(mm, current->mm, tmp);
         ...
}
```

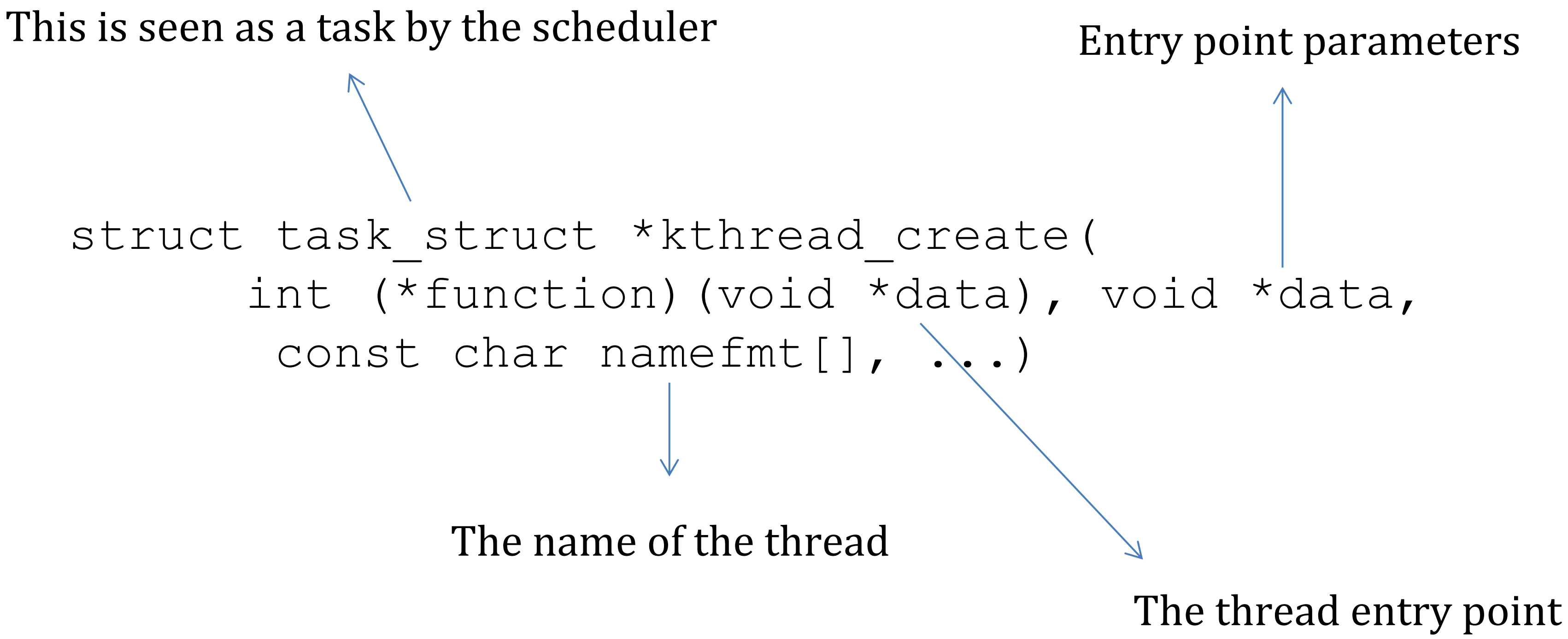
`copy_page_range()` (2.4)

- Defined in `linux/mm/memory.c`
- For any range of addresses associated with the `vm_area_struct` structure, this function sets the PTE page table
- This may lead to cover the user-level addressing range only partially
- In this case, additional PTE tables will be allocated as when userspace allocates new memory

copy_page_range() and COW

```
int copy_page_range(struct mm_struct *dst, struct mm_struct *src,
                        struct vm_area_struct *vma){
        pgd_t * src_pgd, * dst_pgd;
        unsigned long address = vma->vm_start;
        unsigned long end = vma->vm_end;
        unsigned long cow =
                (vma->vm_flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;
        ……
        for (;;) {
        ……
                do {
                        pte_t * src_pte, * dst_pte;
        ………
                        src_pte = pte_offset(src_pmd, address);
                        dst_pte = pte_alloc(dst, dst_pmd, address);
        ……
                        do {
                                pte_t pte = *src_pte;
                                ……
/* If it's a COW mapping, write protect it both in the parent and the child */
                                if (cow && pte_write(pte)) {
                                        ptep_set_wrprotect(src_pte);
                                        pte = *src_pte;
                                }
        ……
}
```

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

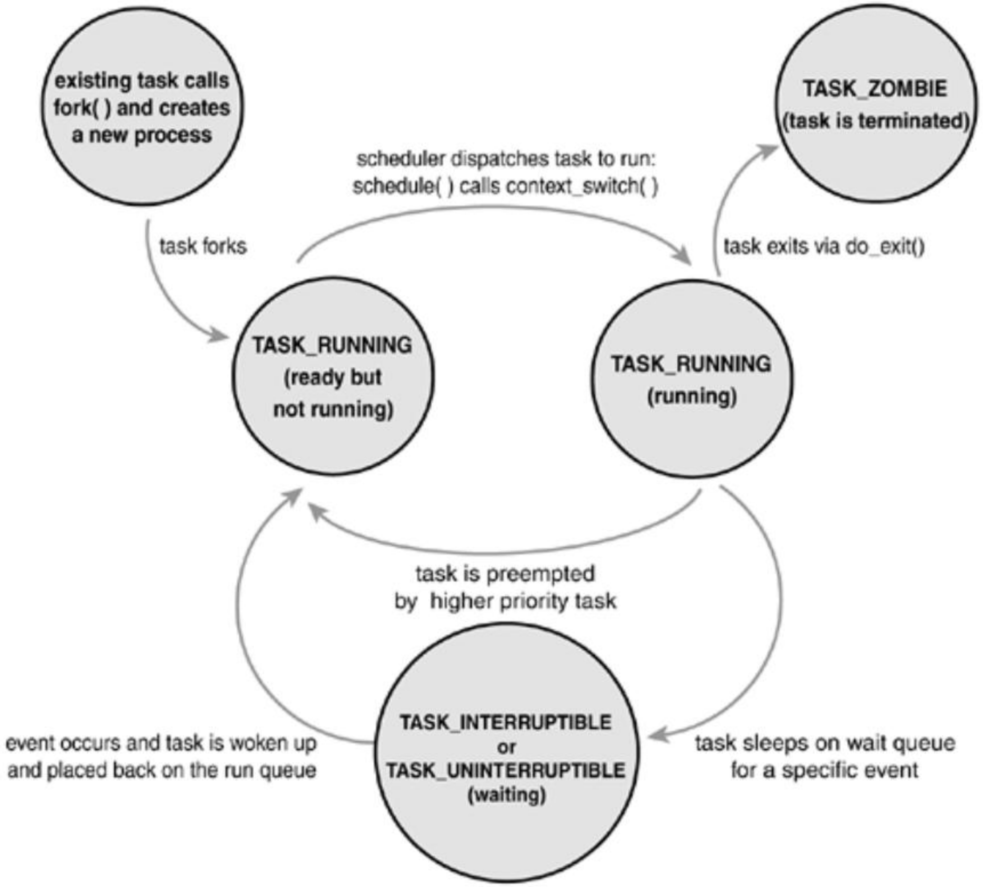
- Kthreads are stopped upon creation
- It must be activated with a call to `wake_up_process()`

__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);
        ...
}
```

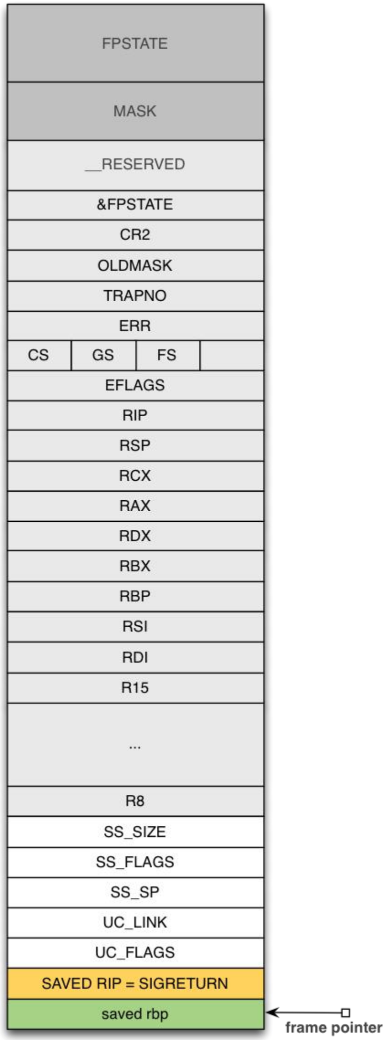
Kernel Thread Daemon

Task State Transition



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;
```

Linux Watchdog

- A watchdog is a component that monitors a system for “normal” behaviour and if it fails, it performs a system reset to hopefully recover normal operation.
- This is a last resort to maintain system availability or to allow sysadmins to remotely log after a restart and check what happened
- In Linux, this is implemented in two parts:
  - A kernel-level module which is able to perform a hard reset
  - A user-space background daemon that refreshes the timer

Linux Whatchdog

- At kernel level, this is implemented using a Non-Maskable Interrupt (NMI)
- The userspace daemon will notify the kernel watchdog module via the `/dev/watchdog` special device file that userspace is still alive

```
while (1) {
        ioctl(fd, WDIOC_KEEPALIVE, 0);
        sleep(10);
}
```