Linux Kernel Details

Have a look

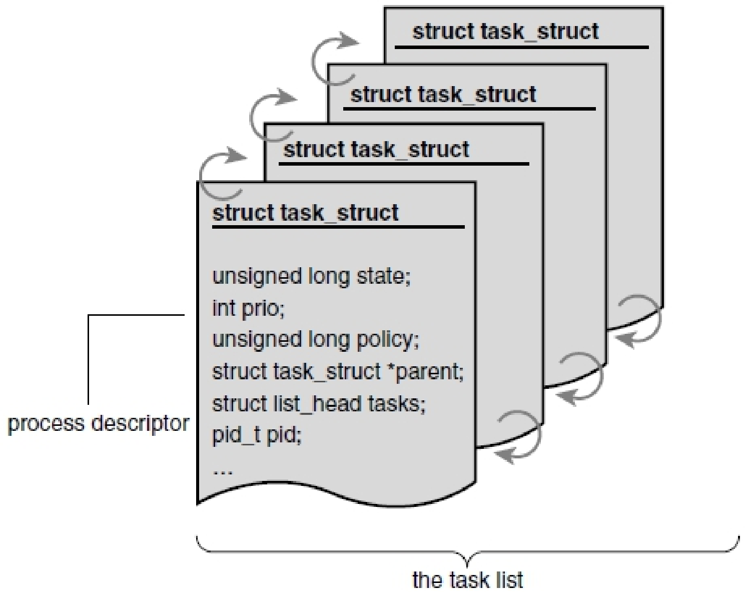
- Before getting into this, please have a look at,
  - *Prof: Ahmed Elarabawy courses*

    Course101: Introduction to Embedded Linux.

    Course102: Understanding Linux.
  - *Linux Kernel Bird's Eye View videos.*

Process Descriptor

- Each process created, has a data structure descriping it, called ***task_struct.***
- The kernel stores the list of processes that created in the system in a circular doubly linked list called the **task list**, each element in the task list is a process descriptor of the type struct ***task_struct***, which is defined in *<linux/sched.h>*.
- The process descriptor contains the data that describes the executing program (open files, the process's address space, pending signals, the process's state, and much more).
- AS we know, A process begins when it's created using the ***fork()*** system call which creates a new process by duplicating an existing one.
- The process that calls ***fork()*** is the ***parent***, whereas the new process is the ***child.***
- The ***task_struct*** structure is allocated via the slab allocator to provide a ***slab cache*** object.



Process Descriptor, *Cont'd*

- ***task_struct*** structure has so many fields, Generally categorized into,
  - ***State*** and execution information such as ***pending signals***, process identification number (***pid***), pointers to ***parents*** and other ***related processes***, ***priorities***, and time information on program execution (e.g., ***CPU time***).
  - Information on allocated ***virtual memory (mm_struct)***.
  - Process credentials such as ***user*** and ***group ID***.
  - ***Files*** used, Not only the binary file with the program code but also filesystem information on all files handled by the process must be saved.
  - ***Thread*** information, which records the CPU-specific runtime data of the process.
  - Info about ***Cgroups*** and ***Namespaces*** related.
  - Information on ***interprocess communication*** required when working with other processes.
  - ***Signal handlers*** used by the process to respond to incoming signals.

```c
struct task_struct {
    .
    .
    .
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long           state;

    /*
     * This begins the randomizable portion of task_struct. Only
     * scheduling-critical items should be added above here.
     */
    randomized_struct_fields_start

    void                    *stack;
    refcount_t              usage;
    /* Per task flags (PF_*), defined further below: */
    unsigned int            flags;
    unsigned int            ptrace;

#ifdef CONFIG_SMP
    int                 on_cpu;
    struct __call_single_node   wake_entry;
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /* Current CPU: */
    unsigned int            cpu;
#endif
    unsigned int            wakee_flips;
    unsigned long           wakee_flip_decay_ts;
    struct task_struct      *last_wakee;
```

Descriptor Fields Highlights

- Process attributes,
  - ***state***: (**TASK_RUNNING**, **TASK_INTERRUPTIBLE**, **TASK_TRACED** by debugger, **TASK_STOPPED**, ...).
  - ***pid***: Unique process identifier.
    tgid: thread group identifier.
  - ***flags***: records various attributes corresponding to a process.
  - ***ptrace***: This field is enabled and set when the process is put into trace mode using the ***ptrace()*** system call.
- Process relations,
  - ***parent***: Pointer to the parent process structure.
  - ***children***: Pointer to a list of child task structures.
  - ***sibling***: Pointer to a list of sibling task structures.
  - ***group_leader***: Pointer to the task structure of the process group leader.

```c
/* -1 unrunnable, 0 runnable, >0 stopped: */
volatile long            state;

pid_t                    pid;
pid_t                    tgid;

unsigned int             flags;

/*
 * Per process flags
 */
#define PF_VCPU          0x00000001  /* I'm a virtual CPU */
#define PF_IDLE          0x00000002  /* I am an IDLE thread */
#define PF_EXITING       0x00000004  /* Getting shut down */
#define PF_IO_WORKER     0x00000010  /* Task is an IO worker */
#define PF_WQ_WORKER     0x00000020  /* I'm a workqueue worker */
#define PF_FORKNOEXEC    0x00000040  /* Forked but didn't exec */
#define PF_MCE_PROCESS   0x00000080  /* Process policy on mce errors */
#define PF_SUPERPRIV     0x00000100  /* Used super-user privileges */
#define PF_DUMPCORE      0x00000200  /* Dumped core */
#define PF_SIGNALED      0x00000400  /* Killed by a signal */
.
.
.

unsigned int             ptrace;

struct task_struct __rcu *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head         children;
struct list_head         sibling;
struct task_struct       *group_leader;
```

Descriptor Fields Highlights, *Cont'd*

- Process Scheduling,
  - ***prio***: Dynamic priority derived from the nice value.
  - ***static_prio***: static priority for real time processes.
  - ***se***, ***rt***, and ***dl***: To keep track of process accounting (notion of the timeslice of a process) for normal, real-time, dead-line processses.
  - ***policy***: whiche class (CFS, Real-time, Dead-line).
  - ***nr_cpus_allowed***: # of CPU(s) the process is eligible to be scheduled in a multi-processor system.
  - ***rlimit*** (handled by ***signal_struct **** in ***task_struct***): Kernel imposes some limits to the resources of a process and increase or decrease it when the user tells that through a system call (***setrlimit()***).

```
int                 prio;
int                 static_prio;

struct sched_entity      se;
struct sched_rt_entity      rt;
struct sched_dl_entity      dl;
```

```
unsigned int              policy;
int                 nr_cpus_allowed;
```

```
struct rlimit {
    unsigned long rlim_cur;
    unsigned long rlim_max;
}
```

```
struct rlimit rlim[RLIM_NLIMITS];
```

```
#define RLIMIT_CPU      0   /* CPU time in sec */
#define RLIMIT_FSIZE        1   /* Maximum filesize */
#define RLIMIT_DATA     2   /* max data size */
#define RLIMIT_STACK        3   /* max stack size */
#define RLIMIT_CORE     4   /* max core file size */
```

Descriptor Fields Highlights, *Cont'd*

- Related files and filesystem to the process,
  - ***fs***: Pointer to filesystem struct that describes interactions between a process and a filesystem (process current working directory, prcocess root directory, ...).
  - ***files***: pointer to all the files that a process opens to perform various operations, files_struct, has a file table strucure.

```
/* Filesystem information: */
struct fs_struct              *fs;
```

```
/* Open file information: */
struct files_struct           *files;
```

Process Signaling,

- ***signal***: Pointer to the ***signal_struct*** that descibes the different signals related to this process.
- ***sighand***: Pointer to signal handlers.
- ***blocked***: identify the signals that are currently masked or blocked by process.

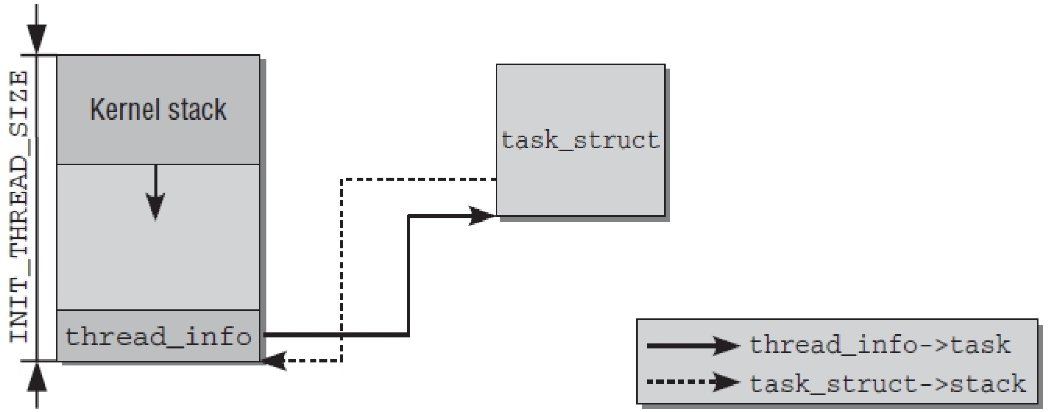
```
struct signal_struct              *signal;
struct sighand_struct __rcu       *sighand;
```

```
sigset_t                          blocked;
```

struct thread_info

- *thread_info* contains CPU-specific's (cpu context registers, sys call nr,...) so that it's HW dependent.
- Once the process created, the kernel will create a corresponding **kernel stack** linked with the process discriptor "which is different from the stack used by the process in User Mode" to be used by the kernel codes.
- ***void *stack*** will allocate a union ***thread_union*** which holds the ***thread_info*** element and ***stack[]*** and usually is contained a double page frames (8K).
- ***thread_info*** lives at the top of the stack and stack pointer "of kernel mode" is started from other side and grows.
- So that The address of ***thread_info*** can be determined from the kernel stack pointer.

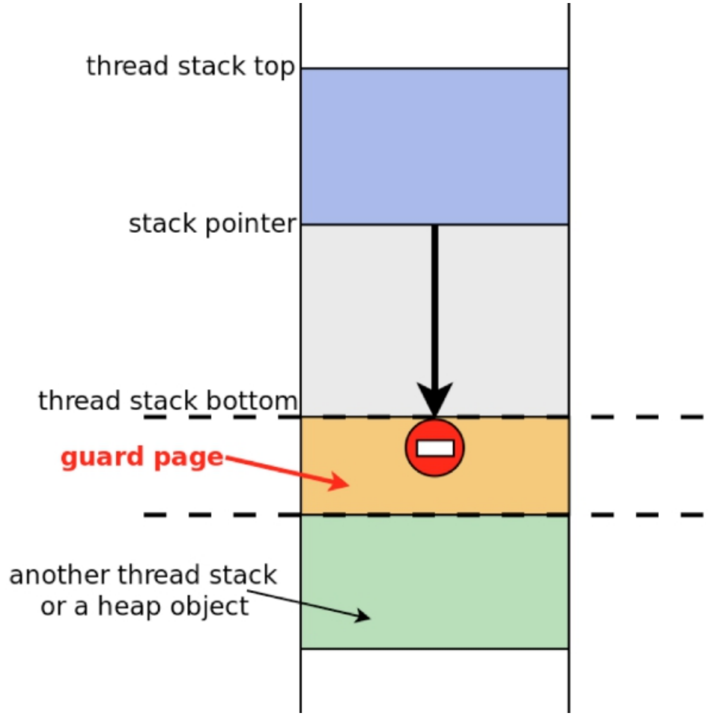
```
union thread_union {
.
.
.
#ifndef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info thread_info;
#endif
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```




```
 * low level task data that entry.S needs immediate access to.
 * __switch_to() assumes cpu_context follows immediately after cpu_domain.
 */
struct thread_info {
	unsigned long		flags;		/* low level flags */
	int			preempt_count;	/* 0 => preemptable, <0 => bug */
	mm_segment_t		addr_limit;	/* address limit */
	struct task_struct	*task;		/* main task structure */
	__u32			cpu;		/* cpu */
	__u32			cpu_domain;	/* cpu domain */
#ifdef CONFIG_STACKPROTECTOR_PER_TASK
	unsigned long		stack_canary;
#endif
	struct cpu_context_save	cpu_context;	/* cpu context */
	__u32			syscall;	/* syscall number */
	__u8			used_cp[16];	/* thread used copro */
	unsigned long		tp_value[2];	/* TLS registers */
#ifdef CONFIG_CRUNCH
	struct crunch_state	crunchstate;
#endif
	union fp_state		fpstate __attribute__((aligned(8)));
	union vfp_state		vfpstate;
#ifdef CONFIG_ARM_THUMBEE
	unsigned long		thumbee_state;	/* ThumbEE Handler Base register */
#endif
};
```

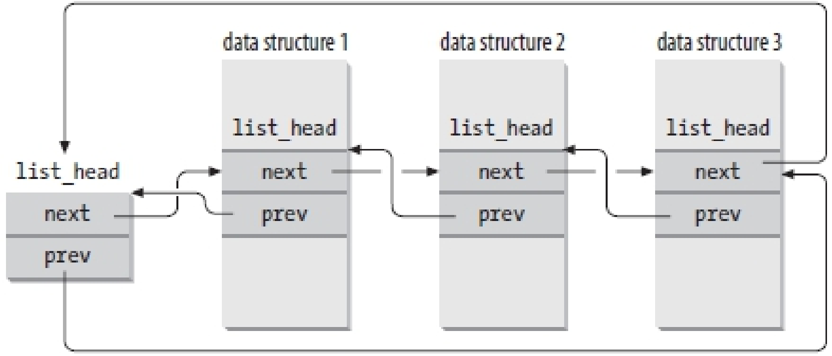
Kernel Stack

- The implementation of feature-rich and deeply layered kernel subsystems may cause **stack overflow** so, Kernel programmers tend to follow coding standards, minimizing the use of local data, avoiding recursion, and avoiding deep nesting among others to cut down the probability of a stack breach.
- A Conventional protection was to use sort of a **guard-page**.
- in order to overcome this problesm specially in case of storage subsystem where filesystems, storage drivers, and networking code can be stacked up in several layers, the kernel stack has been expanded to be **16KB (4 page frames)** (x86-64, since kernel **3.15**).
- Generally the stack was allocated in a direct mapped memory "***Kmalloc***" but after the expansion of the kernel stack, the kernel has come with a new system to set up virtually mapped kernel stacks "***vmalloc***" (kernel **4.9**).
- In x86-64 architecture, the **virtually mapped** kernel stacks with **guard page** both can be supported.



Process Relationships

- If process A forks to generate process B, A is known as the **parent** process and B as the **child** process.
- If process A forks several times therefore generating several child processes B1, B2,Bn, the relationship between the Bi processes is known as a **siblings** relationship.
- ***children*** and ***sibilings*** are of type ***list_head*** structure.
- ***list_head***, such list structure inserted in a wide range of kernel code structures to link them together in a doubly linked list.
- ***real_parent***, the implementation of **POSIX threads** in linux kernel "Talk about later" has a small difference with **processes**, so there are two kinds of parent processes, ***real_parent*** and ***parent***.
- ***Parent*** is the process that receives the **SIGCHLD** signal on child's termination, whereas **real_parent** is the thread that actually created this child process in a multithreaded environment.
- For a normal process, both (***parent***, ***real_parent***)these two values are same, but for a POSIX thread which acts as a process, these two values may be different.



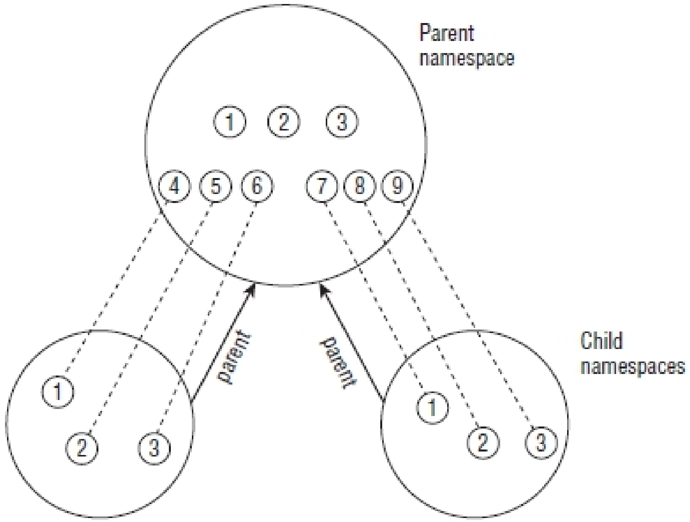

```
struct list_head {
    struct list_head *next, *prev;
};
```

```
struct task_struct {
    .
    .
    .
    /* Real parent process: */
    struct task_struct __rcu     *real_parent;

    /* Recipient of SIGCHLD, wait4() reports: */
    struct task_struct __rcu     *parent;

    /*
     * Children/sibling form the list of natural children:
     */
    struct list_head        children;
    struct list_head        sibling;
```

Process and Namespaces

- Namespaces are mechanisms to abstract, isolate, and limit the visibility that a group of processes has over various system entities such as process trees, network interfaces, user IDs, and filesystem mounts.
- Namespaces can be hierarchically related, such that parent namespace spawned child Nmaespaces, None of the child Namespaces has any notion about other Namespaces in the system but the parent knows about the children, and sees all processes they execute.
- Namespaces can also be non-hierarchical such that there is no connection between parent and child namespaces.
- Each namespace is identified by ***nsproxy*** *new_ns_struct_container;* that says I'm a ns and I have those differnet values of (mnt_namespace, ipc_namespace, net_ns, uts_namespace, ...).

```
struct nsproxy {
	atomic_t count;
	struct uts_namespace *uts_ns;
	struct ipc_namespace *ipc_ns;
	struct mnt_namespace *mnt_ns;
	struct pid_namespace *pid_ns_for_children;
	struct net 	     *net_ns;
	struct time_namespace *time_ns;
	struct time_namespace *time_ns_for_children;
	struct cgroup_namespace *cgroup_ns;
};
```

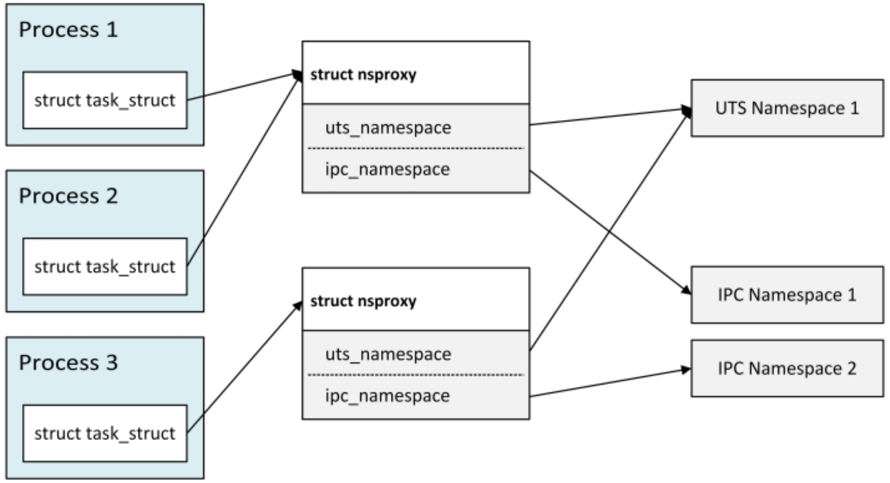


Process and Namespaces, *Cont'd*

- ***nsproxy*** structure contains (***ipc_namespace***: all information related to inter-process communication for this ns, ***mnt_namespace***: all mounted filesystems of this ns,.....).
- Each process "***task_struct***" pointing to which ns it belongs to by ****nsproxy***.
- A new namespace is created Once a fork() is instructed to open when a new task is created with appropriate flags (***CLONE_NEWUTS***, ***CLONE_NEWIPC***,....).
- The initial default global namespace that every process belongs to "***init_nsproxy***".

```
struct task_struct {
    .
    .
    .
    /* Namespaces: */
    struct nsproxy          *nsproxy;
```

```
struct nsproxy init_nsproxy = {
    .count          = ATOMIC_INIT(1),
    .uts_ns         = &init_uts_ns,
#if defined(CONFIG_POSIX_MQUEUE) || defined(CONFIG_SYSVIPC)
    .ipc_ns         = &init_ipc_ns,
#endif
    .mnt_ns         = NULL,
    .pid_ns_for_children    = &init_pid_ns,
#ifdef CONFIG_NET
    .net_ns         = &init_net,
#endif
#ifdef CONFIG_CGROUPS
    .cgroup_ns      = &init_cgroup_ns,
#endif
#ifdef CONFIG_TIME_NS
    .time_ns        = &init_time_ns,
    .time_ns_for_children   = &init_time_ns,
#endif
};
```




```
struct nsproxy {
    atomic_t count;
    struct uts_namespace *uts_ns;
    struct ipc_namespace *ipc_ns;
    struct mnt_namespace *mnt_ns;
    struct pid_namespace *pid_ns_for_children;
    struct net          *net_ns;
    struct time_namespace *time_ns;
    struct time_namespace *time_ns_for_children;
    struct cgroup_namespace *cgroup_ns;
};
```

```
#define CLONE_NEWUTS        0x04000000  /* New utsname namespace */
#define CLONE_NEWIPC        0x08000000  /* New ipc namespace */
#define CLONE_NEWUSER       0x10000000  /* New user namespace */
#define CLONE_NEWPID        0x20000000  /* New pid namespace */
```

Let's Discuss "process related" pid_namespace

- Generally PID namespaces isolate process ID numbers, and allow duplication of PID numbers across different PID namespaces while, the process IDs within a PID namespace are unique, and are assigned sequentially starting with PID 1.
- First, let's get some points in mind,
  - Each process internally has not only one identification with respect to kernel but, (***PIDTYPE_PID***: General id for the process, ***PIDTYPE_PGID***: Id for the process inside the group it belongs to, ***PIDTYPE_SID***: Id for the process inside specific session).
  - So a ***struct pid{}*** is created to be the kernel-internal representation of a ***PID*** that the user sees.
  - This ***struct pid{}*** is considered as an <u>independent unit</u>, each process "task_struct" created, points to one ***struct pid{}*** that represents it to the rest of the kernel.
  - More than one process may point to the same ***struct pid{}*** but should be related to different namespaces as we agreed.

```
struct task_struct {
    ...
    /* PID/PID hash table linkage. */
    struct pid          *thread_pid;
```

Let's Discuss "process related" pid_namespace, *Cont'd*

- First, let's get some points in mind, *cont'd*
  - Each ***struct pid{}*** has an array of ***struct upid*** that represents the information that is visible for each namespace.
  - This array of ***struct upid numbers[1];*** by default of **1** entry as the process is contained only in the global namespace, and can be extnded by **idr lib** management.
  - Another general C trick used in the kernel a lot to append at run time newly created extension,

```
struct my_struct {
    ...
    struct internal_struct list[1];
};
struct my_struct * prt_struct = (my_struct *) kmalloc(
sizeof(my_struct) + sizeof(internal_struct) * N, GFP_KERNEL);
```

- ***pid_namespace*** structure main elements,
  - ***child_reaper***: Pointer to the first process that's considered the **init task** for this **ns** which will be the parent of all upcoming childs and the parent of the orphaned processes. 'Orphaned": Process that its parent terminated and becomes **zombie**.

```
struct pid
{
    refcount_t count;
    unsigned int level;
    spinlock_t lock;
    /* lists of tasks that use this pid */
    struct hlist_head tasks[PIDTYPE_MAX];
    struct hlist_head inodes;
    /* wait queue for pidfd notifications */
    wait_queue_head_t wait_pidfd;
    struct rcu_head rcu;
    struct upid numbers[1];
};
```

```
struct upid {
    int nr;
    struct pid_namespace *ns;
};
```

```
struct pid_namespace {
    struct idr idr;
    struct rcu_head rcu;
    unsigned int pid_allocated;
    struct task_struct *child_reaper;
    struct kmem_cache *pid_cachep;
    unsigned int level;
    struct pid_namespace *parent;
#ifdef CONFIG_BSD_PROCESS_ACCT
    struct fs_pin *bacct;
#endif
    struct user_namespace *user_ns;
    struct ucounts *ucounts;
    int reboot; /* group exit code if this pidns was rebooted */
    struct ns_common ns;
} __randomize_layout;
```

Let's Discuss "process related" pid_namespace, *Cont'd*

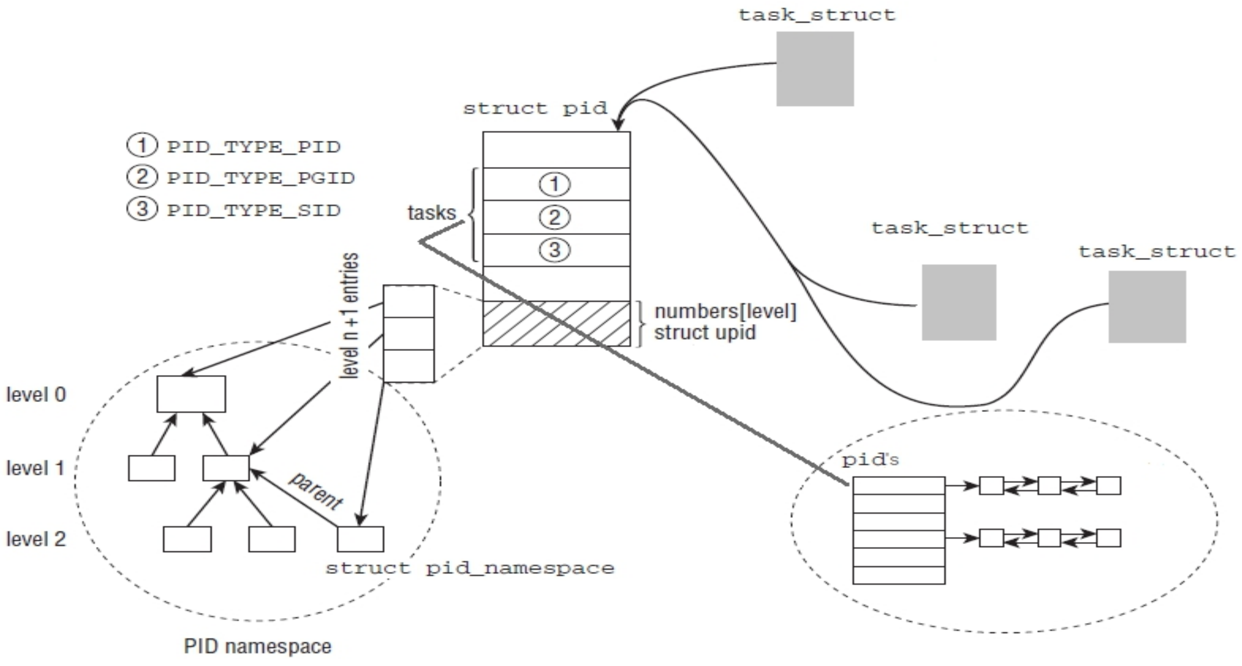
- ❑ ***pid_namespace*** structure main elements, *Cont'd*
  - ➢ ***parent***: Pointer to the parent ns as we know the namespaces could be hierarchical.
- ❑ Pig picture as follows,
  - ➢ ***Each struct pid{}*** has the start head of a list for all the processes linked to this ***struct pid{}*** represented by ***struct hlist_head tasks***.

```
struct pid
{
    refcount_t count;
    unsigned int level;
    spinlock_t lock;
    /* lists of tasks that use this pid */
    struct hlist_head tasks[PIDTYPE_MAX];
    struct hlist_head inodes;
    /* wait queue for pidfd notifications */
    wait_queue_head_t wait_pidfd;
    struct rcu_head rcu;
    struct upid numbers[1];
};
```

```
struct upid {
    int nr;
    struct pid_namespace *ns;
};
```

```
struct task_struct {
    ...
    struct hlist_node        pid_links[PIDTYPE_MAX];
```

```
struct pid_namespace {
    struct idr idr;
    struct rcu_head rcu;
    unsigned int pid_allocated;
    struct task_struct *child_reaper;
    struct kmem_cache *pid_cachep;
    unsigned int level;
    struct pid_namespace *parent;
#ifdef CONFIG_BSD_PROCESS_ACCT
    struct fs_pin *bacct;
#endif
    struct user_namespace *user_ns;
    struct ucounts *ucounts;
    int reboot; /* group exit code if this pidns was rebooted */
    struct ns_common ns;
} __randomize_layout;
```



Let's Discuss "process related" pid_namespace, *Cont'd*

- Pig picture as follows *Cont'd* ,
  - Pid structure allocation:

    pid_idr_init() in /init/main.c, allocates the first ***Pid{}*** then while the system is running allocates dynamiclly by ***alloc_pid()***.

```
void __init pid_idr_init(void)
{
    ...
    init_pid_ns.pid_cachep = KMEM_CACHE(pid,
            SLAB_HWCACHE_ALIGN | SLAB_PANIC | SLAB_ACCOUNT);
}
```

```
struct pid *alloc_pid(struct pid_namespace *ns, pid_t *set_tid,
              size_t set_tid_size)
{
    ...
    pid = kmem_cache_alloc(ns->pid_cachep, GFP_KERNEL);
    if (!pid)
        return ERR_PTR(retval);

}
```

```
struct pid_namespace {
    struct idr idr;
    struct rcu_head rcu;
    unsigned int pid_allocated;
    struct task_struct *child_reaper;
    struct kmem_cache *pid_cachep;
    unsigned int level;
    struct pid_namespace *parent;
#ifdef CONFIG_BSD_PROCESS_ACCT
    struct fs_pin *bacct;
#endif
    struct user_namespace *user_ns;
    struct ucounts *ucounts;
    int reboot; /* group exit code if this pidns was rebooted */
    struct ns_common ns;
} __randomize_layout;
```

Process Context Switching

- Every process switch consists of two steps,
  - Switch the virtual memory mapping from the previous process to that of the new process.
  - Switching the Kernel mode stack and the hardware context, which provides all the information needed by the kernel to execute the new process, including the CPU registers.
- Once ***schedule()*** triggering the switch process, it passes 3 parameters (prev, next, and last),

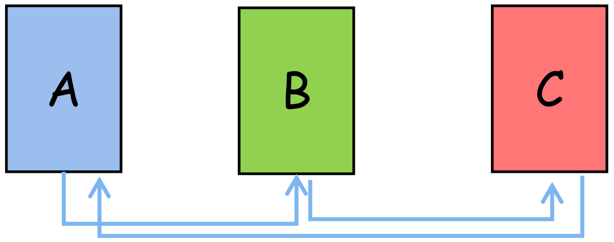
  switch_to*(prev, next, last)*

  prev: The memory locations containing the descriptor address of the process being replaced.

  next: The memory locations containing the descriptor address of the new process.

  last: The memory locations containing the descriptor address of the last process. *Why we need it!!?*

  -> *To adjust the Kernel stack* ***prev*** *element after restoring the process context as the kernel needs the last operated process, Imagine the following (A,B,C) processes.*




```
#define switch_to(prev,next,last)                    \
do {                                                 \
    __complete_pending_tlbi();                       \
    last = __switch_to(prev,task_thread_info(prev), task_thread_info(next));   \
} while (0)
```

```
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next, struct rq_flags *rf)
{
    ...
    /* Here we just switch the register state and the stack. */
    switch_to(prev, next, prev);
    barrier();

    return finish_task_switch(prev);
}
```

Preemption

- User preemption occurs when,

  The kernel is about to return from a system call to user-space as it finished all things that the kernel is done on behalf of the user process, or when returning from an interrupt handler (tick or others,...) and ***need_resched*** flag is raised so, the **scheduler** will be invoked to select a new process to be executed.

- Kernel Preemption,
  - **Before Kernel 2.6,**

    When we are running in the kernel side, the kernel code is scheduled cooperatively, not preemptively, Kernel code runs until it finishes (returns to user-space) or explicitly blocks (Processes put themselves on a wait list and then ***schedule()***).

  - **After Kernel 2.6,**
    - It's possible to preempt a task at any point inside a kernel code running on bealf of it, as long as the kernel in a safe state to ***schedule()***.
    - Safe state means, the task running in the kernel side does not hold any sort of locks, so that a variable ***preempt_count*** added in the ***thread_info*** to denote #of locks acquired, once it's zero, and ***need_resched*** flag is raised, the kernel can be preempted.

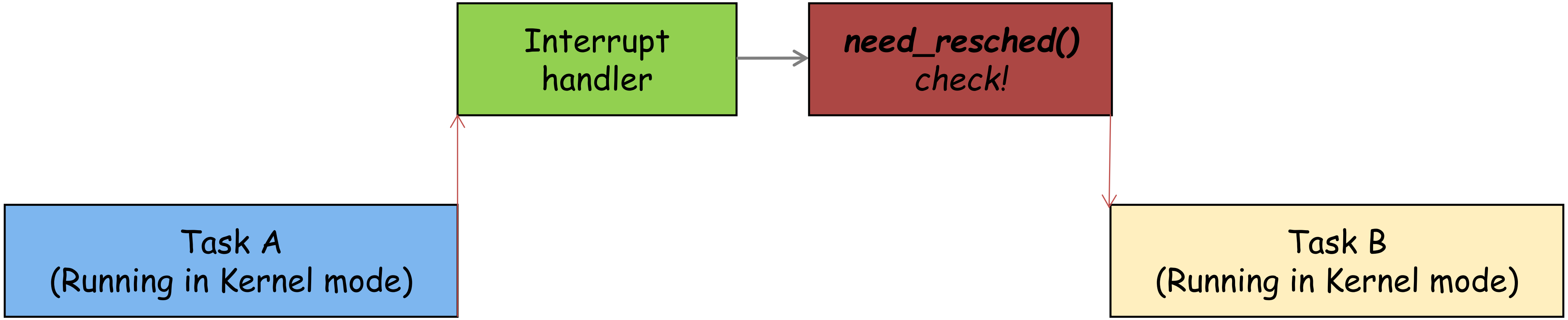
```
struct thread_info {
    ...
    int         preempt_count;  /* 0 => preemptable, <0 => bug */
```

Preemption, *Cont'd*

- Kernel Preemption, *cont'd*
  - **After Kernel 2.6**, *cont'd*

    So, Kernel preemption can occur,

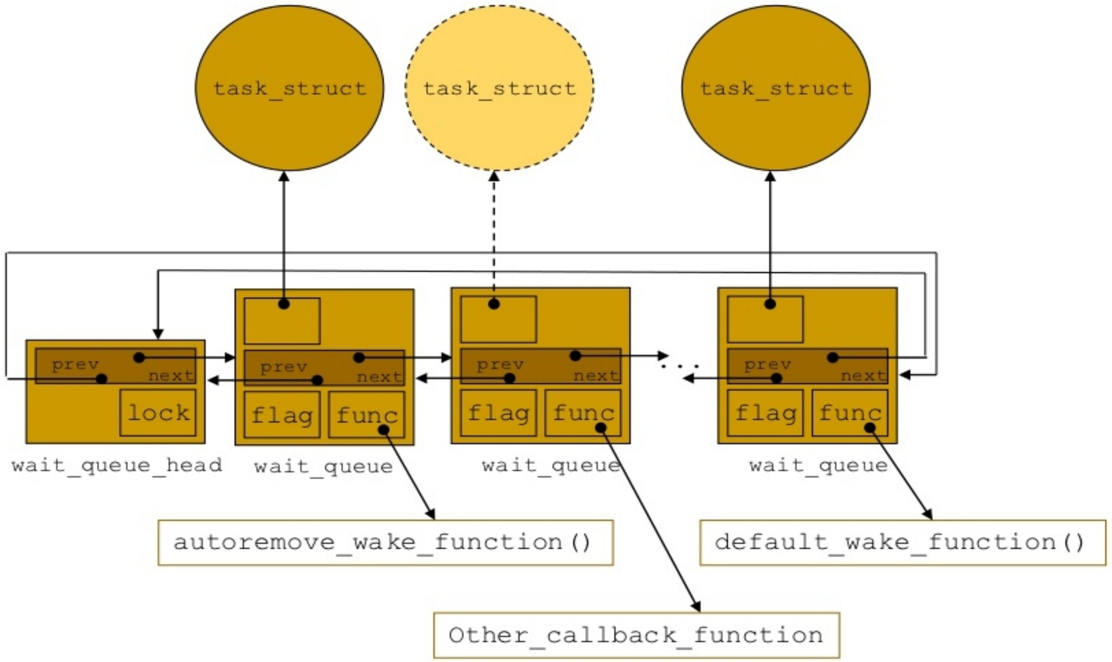
    - When an interrupt happened and the handler exits.
    - If a task in the kernel explicitly calls ***schedule()***.
    - If a task in the kernel blocks because of general locking synchronizarion (which results in a call to ***schedule()***).



Wait Queues

- The kernel code is responsible for pushing whatever process to the ***sleep*** state or ***wake*** it up.
- Sleeping is handled via wait queues.
- Wait queue is a simple list of processes waiting for an event to occur.
- Each wait queue started with a <u>head</u> called ***wait_queue_head*** and at least one ***wait_queue_entry*** is linked.
- Processes put themselves on a wait queue and mark themselves not runnable.
- Wait queues diagram.

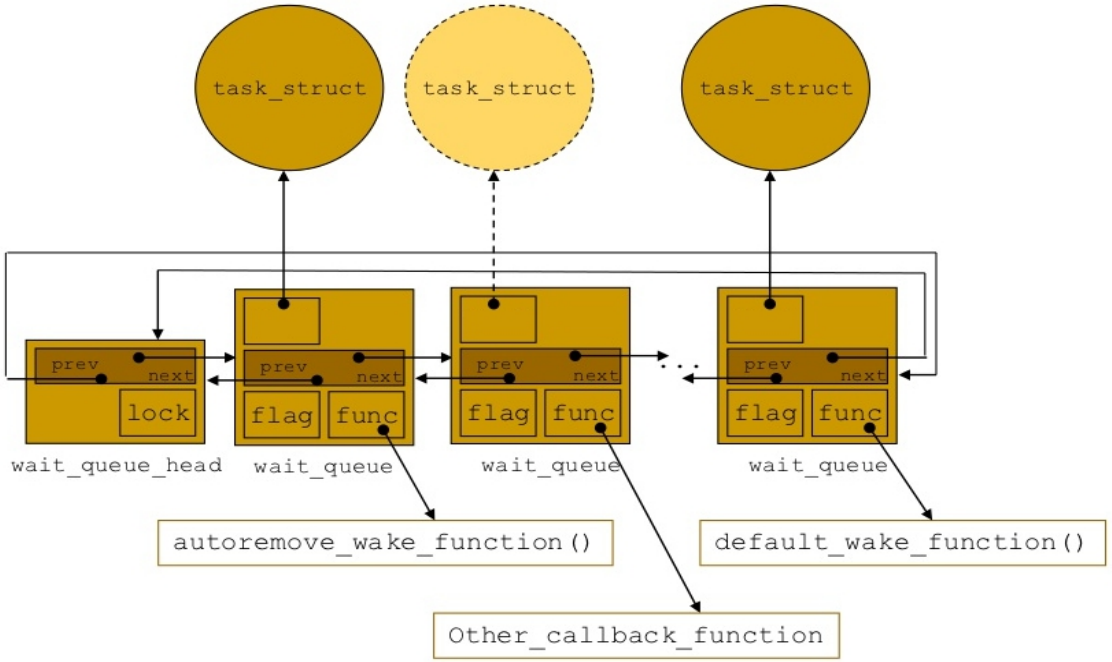
```
struct list_head {
    struct list_head *next, *prev;
};
struct wait_queue_head {
    spinlock_t       lock;
    struct list_head     head;
};

typedef int (*wait_queue_func_t)(struct wait_queue_entry *wq_entry,
    unsigned mode, int flags, void *key);

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



Wait Queues, *Cont'd*

- Ex. 'q' is the ***wait_queue_head*** we wish to sleep on.
- ***wait_queue_func_t func;*** is invoked to wake the task.
- ***DEFINE_WAIT():*** Creates ***wait_queue_entry*** statically.
- ***add_wait_queue*** to link the head with the entry.
- Calls ***prepare_to_wait()*** to change the process state to sleep ***TASK_INTERRUPTIBLE*** .
- Imagine any Signal wakes the process so, it gets to check the ***signal_pending(current)*** for the process.
- Before I choose ***scheduling()*** myself, the process should release all the locks acquired before.

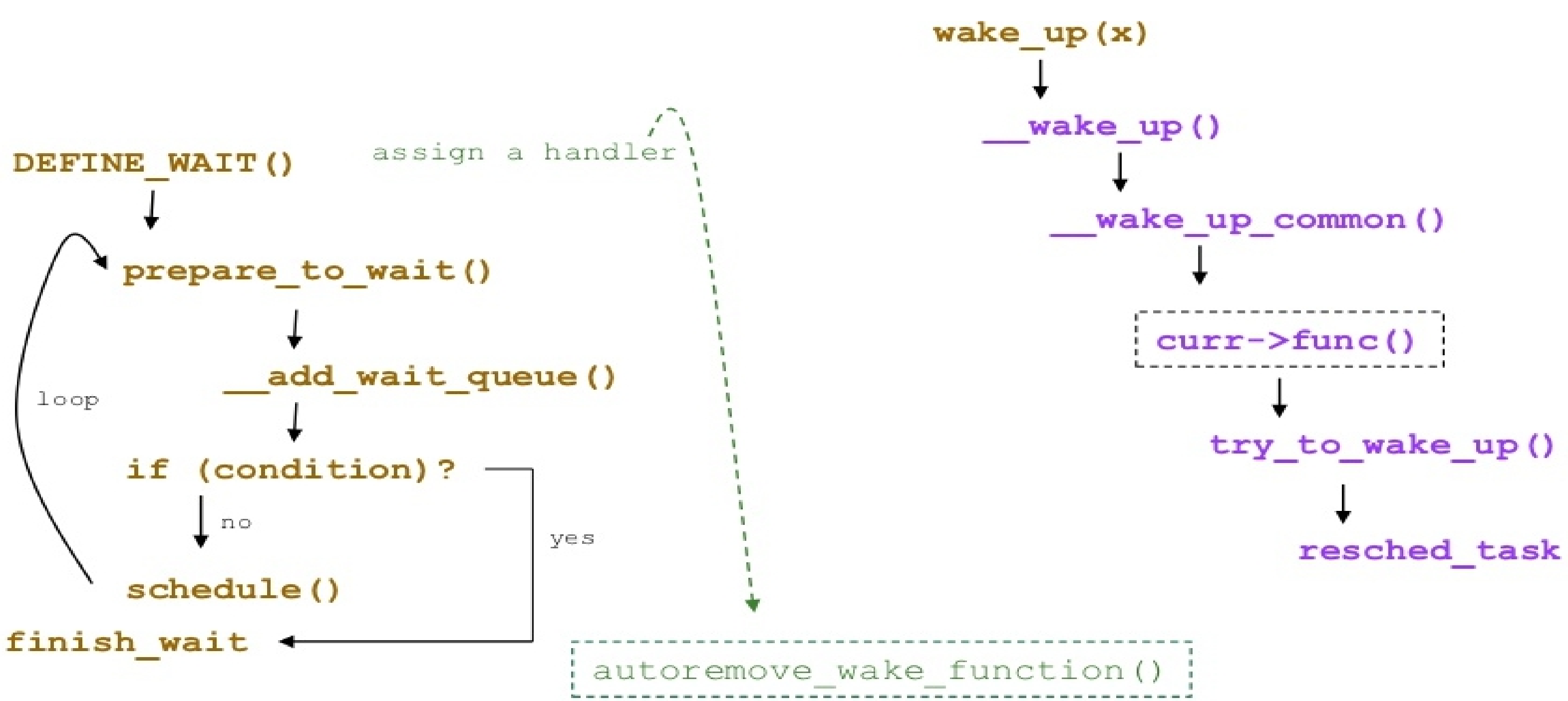
```
wait_queue_head_t q;
DEFINE_WAIT(wait);
add_wait_queue(q, &wait);
while (!condition) { /* condition is the event that we are waiting for */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        ;/* handle signal */
    schedule();
}
finish_wait(&q, &wait);
```



Wait Queues, *Cont'd*

- After the condition acheived, the other code that raise this condition must ***wake_up(q);*** to wake up the entries linked by the passed q head by call the callback ***func*** of each entry.
- Now that the condition is true, the task sets itself to **TASK_RUNNING** and removes itself from the wait queue via ***finish_wait()***.

```
wait_queue_head_t q;
DEFINE_WAIT(wait);
add_wait_queue(q, &wait);
while (!condition) { /* condition is the event that we are waiting for */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        ;/* handle signal */
    schedule();
}
finish_wait(&q, &wait);
```



Process Creation

- ***fork()*** system call uses ***clone()*** system call in implementation!!!, Remember when we called the linux thread is a light weight process as its creation uses **"*clone()*"**, the only difference is what specifically the resources shared between the caller process and the new one is forked!.
- The ***clone()*** system call, calls ***kernel_clone()*** the main fork-routine that calls ***copy_process()*** that's doing the main work,
  - ***dup_task_struct()***, which creates a new *kernel stack*, ***thread_info*** structure, and ***task_struct*** for the new process such that the new values are identical to those of the current task and at this point, the child and parent process descriptors are identical.
  - The child needs to differentiate itself from its parent, various members of the process descriptor are cleared or set to initial values.
  - The child's state is set to ***TASK_UNINTERRUPTIBLE*** to ensure that it does not yet run.
  - Adjust per task ***flags***, i.e *PF_FORKNOEXEC* flag, denotes a process forked but didn't ***exec()***.
  - ***alloc_pid()*** to assign an available PID to the new task.

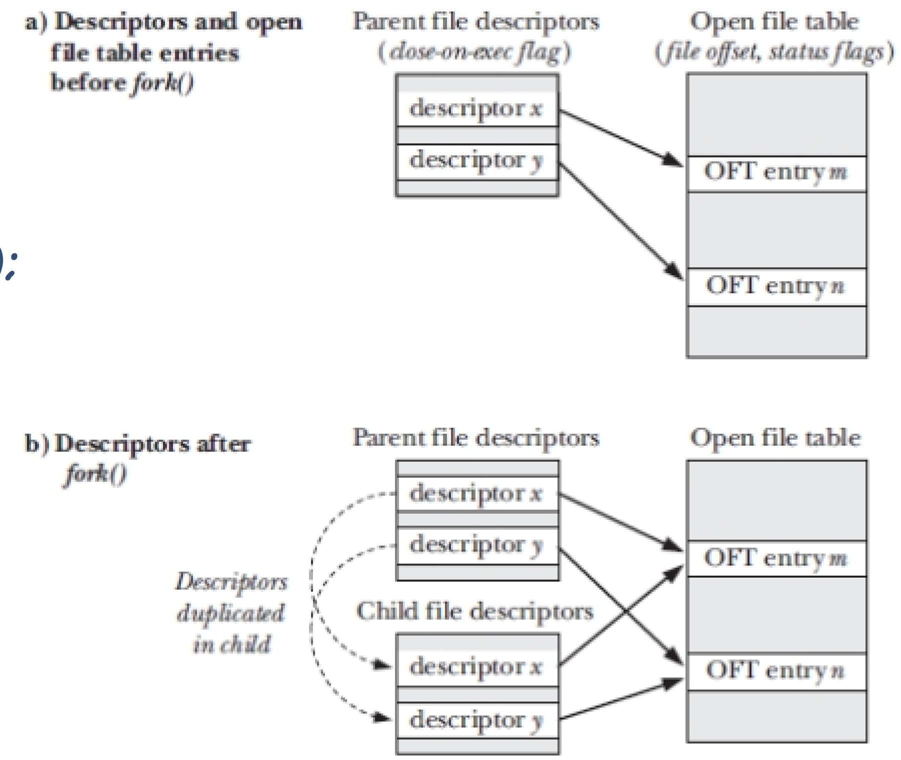
Process Creation, *Cont'd*

- ***copy_process()*** *cont'd,*
  - Depending on the args passed to ***clone()***, ***copy_process()*** either duplicates or shares **open files**, **filesystem** information, **signal handlers**, **process address space**, and **namespace**. These resources are typically shared between ***threads*** *"Linux light weight process meaning"* in a given process; otherwise they are unique and thus copied here,

    so we can imagine process and thread creation,

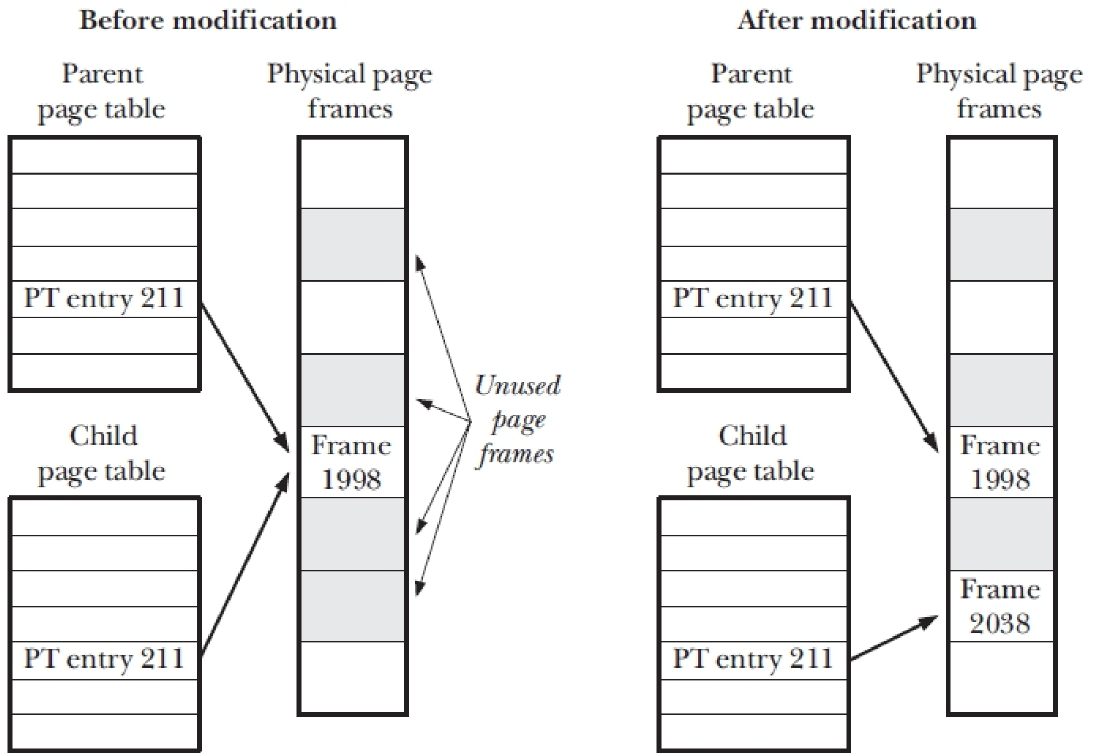
    process: ***clone(SIGCHLD, 0);***

    thread: ***clone(CLONE_VFORK | CLONE_VM | SIGCHLD, 0);***



Copy on Write

- As we knew ***fork()*** will duplicate the resources of the parent in favor of child but the kernel doing that through the use of copy-on-write mechanism.
- In other words, the technique of Copy-on-write (COW) is to delay or altogether preventing copying of the data so that, the address space (~10's of megas) of parent and the child can share a single copy temporary.
- Until then, the address space is marked as read-only, once the data of one of the processes is written, a duplicate is made and each process receives a unique copy.
- We will talk later in detalis about the process address space from user space perspective



execve, How programs get run!

- ***exec()*** family loads the program into the current process space and runs it from the entry point.
- The ***execve()*** system call entry is ***do_execve()***, and the body of implementation in ***do_execveat_common***,
  - The main purpose of ***do_execveat_common*** is to build a new ***struct linux_binprm*** instance that describes the current program invocation operation, parameters of the new process (e.g., euid, egid, argument list, environment, filename, etc.).
  - The ***bprm_mm_init()*** function allocates and sets up the associated ***struct mm_struct*** and ***struct vm_area_struct*** for managing the address space and virtual memory of the current process for the new program and also sets up an initial stack.
  - Raise unsafe flag if program execution might not be safe, as Linux Security Module uses this information to deny the program execution operation.
  - At the end, information about the program invocation is copied into the top of new program's stack, using the local ***copy_strings()*** and ***copy_strings_kernel()*** utility functions.
  - The stack looks like,

```
SYSCALL_DEFINE3(execve,
        const char __user *, filename,
        const char __user *const __user *, argv,
        const char __user *const __user *, envp)
{
    return do_execve(getname(filename), argv, envp);
}
```

```
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

```
---------Memory limit (Top of the Stack)---
NULL pointer
program_filename string
envp[envc-1] string
...
envp[1] string
envp[0] string
argv[argc-1] string
...
argv[1] string
argv[0] string
```

Note: euid, egid are the effective ID's describes the user, group whose file access permissions are used by the process.

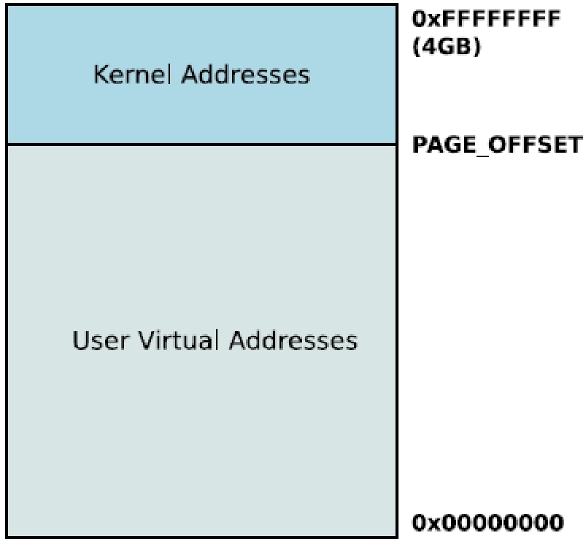
execve, How programs get run!, *Cont'd*

- ***do_execveat_common*** *cont'd,*
  - With a complete ***struct linux_binprm*** in hand, we need to perform the real execution of a binary by ***exec_binprm()*** that uses ***search_binary_handler()*** that iterates over a list of ***struct linux_binfmt*** objects, each of which provides a handler for a particular format of binary programs.
  - For each ***struct linux_binfmt*** handler object, the ***linux_binfmt.load_binary()*** function pointer for loading the binary, if the handler code supports the binary format, it does whatever is needed to prepare the program for execution and returns success, if not, try the next handler.
  - Because of the execution of a particular program may rely on execution of a different program so, ***search_binary_handler()*** code can be called recursively, so that, a limit on the stack usage by binfmt code is setup by ***bprm_stack_limits().***
  - For some binary formats, check the kernel source */fs/*

binfmt_script.c: Support for interpreted scripts, starting with a #! line.
binfmt_misc.c: Support miscellaneous binary formats, according to runtime configuration.
binfmt_elf.c: Support for ELF format binaries.
binfmt_aout.c: Support for traditional a.out format binaries.
binfmt_flat.c: Support for flat format binaries.
binfmt_em86.c: Support for Intel ELF binaries running on Alpha machines.
binfmt_elf_fdpic.c: Support for ELF FDPIC binaries.
binfmt_som.c: Support for SOM format binaries (an HP/UX PA-RISC format).

Kernel Threads

- As we know Kernel threads are uesed to handle some of the background works (flushing disk caches, swapping out unused pages, servicing network connections, and so on).
- The significant difference between kernel threads and normal processes,
  - Kernel threads do not have an address space, (mm pointer, which points at their address space, is NULL).
  - Kernel threads run only in Kernel Mode, while regular processes run in Kernel Mode and in User Mode, so that, the kernel threads use only linear addresses > ***PAGE_OFFSET***, the regular processes, use all 4 GB of linear addresses, in either User Mode or Kernel Mode.
  - The kernel thread is created by ***kthread_create()***, it's by default in an unrunnable state (-1) and can explicitly wake it up via ***wake_up_process()*** to be ***TASK_NORMAL***.
  - Using ***kthread_run()*** to create and wake a thread directly.
  - ***kthread_stop()*** to kill the kernel thread.
  - The first kernel thread ***kthreadd (PID 2)*** is created once the kernel started by ***start_kernel()***.



Destroying Processes

- Process destruction occurs when,
  - Explicitly the process calls the ***exit()*** system call when it is ready to terminate.
  - Implicitly on return from the main subroutine of any program, (That is, the C compiler places a call to ***exit()*** after ***main()*** returns).
  - Process receives a signal or exception it cannot handle or ignore i.e. kill signal.
- Regardless of how a process terminates, the bulk of the work is handled by ***do_exit(),***
  - It raise ***PF_EXITING*** <u>flags</u> inside ***task_struct.***
  - Destruct everything related to the process,

    hrtimer_cancel(), exit_itimers(): cancel POSIX timers related, ***exit_mm():*** to release the specified ***mm_struct*** held by this process, ***exit_sem():*** dequeued the process from any waiting queue, ***exit_shm():*** destroy all already created shared segments by the process, ***exit_files()*** and ***exit_fs():*** to inform the kernel internal data structure that the usage of file descriptors and filesystem data respectively decreased by one, ***exit_task_namespaces():*** remove from any namespace, ***exit_task_work():*** cancel any kernel work subscripted by the process, ***exit_thread():*** free thread data structures created, ***exit_notify***(): to notify to the task's parent.

Destroying Processes

- ***do_exit()**, cont'd*
  - After **do_exit()** completes, the process descriptor for the terminated process still exists, zombie state but why!!!?

    -> Because this enables the system to obtain info about a child process after it has terminated.
- *When will the process descriptor get destructed?*
  - *After the parent has obtained the info about the terminated child and destruct it.*
  - *It can suspend itself using **wait()** until one of its children exits, at that time the **wait()** returns with the PID of the exited child and deallocates the process descriptor internally by **release_task()**.*

Code Example

- Code example.1 - fork
- Use GCC to build.

  $gcc fork_exec.c -o fork_exec.out
- Execute the built program.

  $./fork_exec.out 0 1
- For simplicity, the inputs not checked.
- Output,

```
karim_eshapa@karimeshapa-vm:~/kernel_sessions/processes/fork_exec$ ./fork_exec.out 0 1

The parent will execute ls -l

Child process will execute date
Sat Mar 13 22:22:10 EET 2021
total 12
-rw-rw-r-- 1 karim_eshapa karim_eshapa  684 Mar 13 22:07 fork_exec.c
-rwxrwxr-x 1 karim_eshapa karim_eshapa 7568 Mar 13 22:08 fork_exec.out
```

```c
#define BASH     "/bin/bash"
#define NR_PROGS    2

char * arr_prog[NR_PROGS] = {
    [0] = "ls -l",
    [1] = "date",
};

int main(int args, const char * argv[])
{
    pid_t pid;

    pid = fork();
    if (pid == 0) {
        printf("\nChild process will execute %s\n", arr_prog[atoi(argv[2])]);
        execl(BASH, BASH, "-c", arr_prog[atoi(argv[2])], NULL);
        exit(0);
    }
    else if (pid < 0) {
        printf("The fork failed\n");
        return -1;
    }
    else {
        printf("\nThe parent will execute %s\n", arr_prog[atoi(argv[1])]);
        execl(BASH, BASH, "-c", arr_prog[atoi(argv[1])], NULL);
        exit(0);
    }

    return 0;
}
```

Code Example, *Cont'd*

- Code example.2 - execve
- Use GCC to build.

  $gcc execve_sh.c -o execve_sh.out
- Execute the built program.

  $./execve_sh.out
- Output,

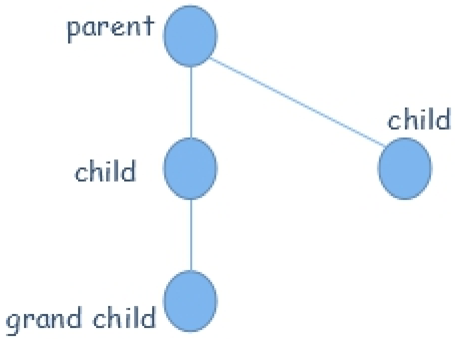
```
karim_eshapa@karimeshapa-vm:~/kernel_sessions/processes/execve$ ./execve_sh.out
$ echo $USER
Kareem
$ ls
execve_sh.c  execve_sh.out
$ 
```

```c
int main(void)
{
    char *argv[] = { "/bin/sh", 0 };
    char *envp[] =
    {
        "HOME=/",
        "PATH=/bin:/usr/bin",
        "USER=Kareem",
        "LOGNAME=Kareem",
        0
    };
    execve(argv[0], &argv[0], envp);
    fprintf(stderr, "Oops!\n");
    return -1;
}
```

```c
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
```

Code Example, Cont'd

- Code example.3 - fork tree




```c
int main()
{
    pid_t pid1 = fork();
    pid_t pid2 = fork();

    if (pid1 > 0 && pid2 > 0) {
        /* Parent */
    } else if (pid1 == 0 && pid2 > 0) {
        /* Child 1 */
    } else if (pid1 > 0 && pid2 == 0) {
        /* Child 2 */
    } else if (pid1 == 0 && pid == 0) {
        /* Grand child */
    }
}
```

Code Example, *Cont'd*

- Code example.4 fork copies of data
- Terminal,

  $gcc fork_copies.c.c -o fork_copies.c.out

  $./fork_copies.c.out

- Output,

```
Parent copy my_var 6
Child copy my_var 8
```

```c
int main(int args, const char * argv[])
{
    int my_var = 5;
    pid_t pid = fork();

    if (pid == 0) { /* Child */
        my_var += 3;
        printf("Child copy my_var %d \n", my_var);

    } else if (pid > 0) { /* Parent */
        my_var ++;
        printf("Parent copy my_var %d \n", my_var);

    } else { /* failed */
        printf("Fork failed\n");
        return 1;
    }

    return 0;
}
```

Code Example, *Cont'd*

- Code example.5 pid namespaces
- Taken from [source].
- Terminal,

  $sudo su "to be allowed creating a new ns"

  $gcc pid_ns.c -o pid_ns.out

  $./pid_ns.out

- Why ***child_stack+1048576 !***

  Because stacks grow downward on most of processors that run Linux,so stack usually points to the topmost address of the memory space set up for the child stack.

- Output,

```
clone() = 9656
PID: 1
```

```c
static char child_stack[1048576];

static int child_fn() {
  printf("PID: %ld\n", (long)getpid());
  return 0;
}

int main() {
  pid_t child_pid = clone(child_fn, child_stack+1048576, CLONE_NEWPID | SIGCHLD, NULL);
  printf("clone() = %ld\n", (long)child_pid);

  waitpid(child_pid, NULL, 0);
  return 0;
}
```