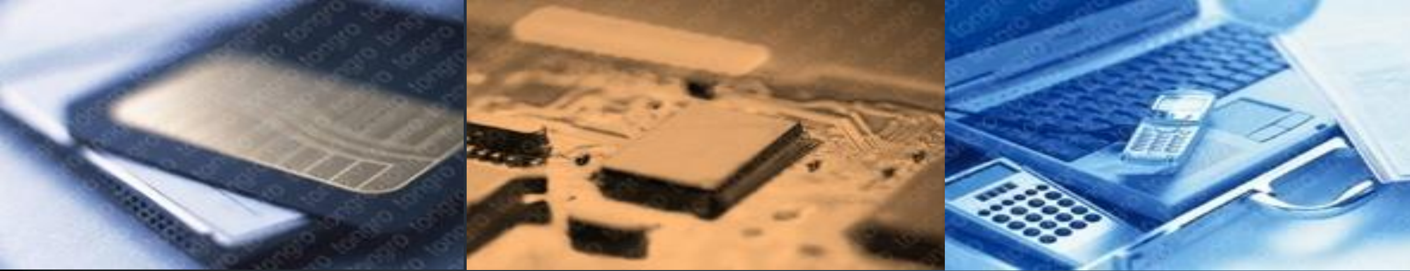


Process Management and Scheduling

오명훈
snt2426@gmail.com

Process Management and Scheduling

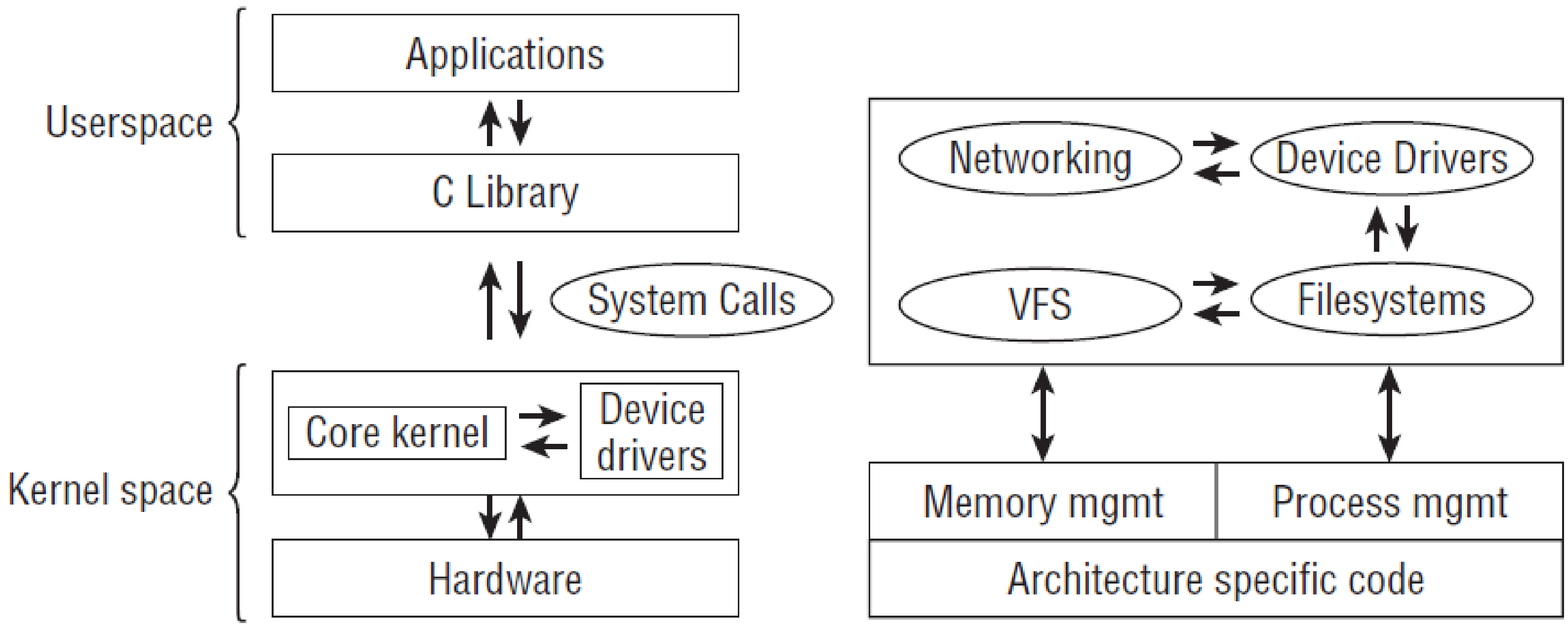
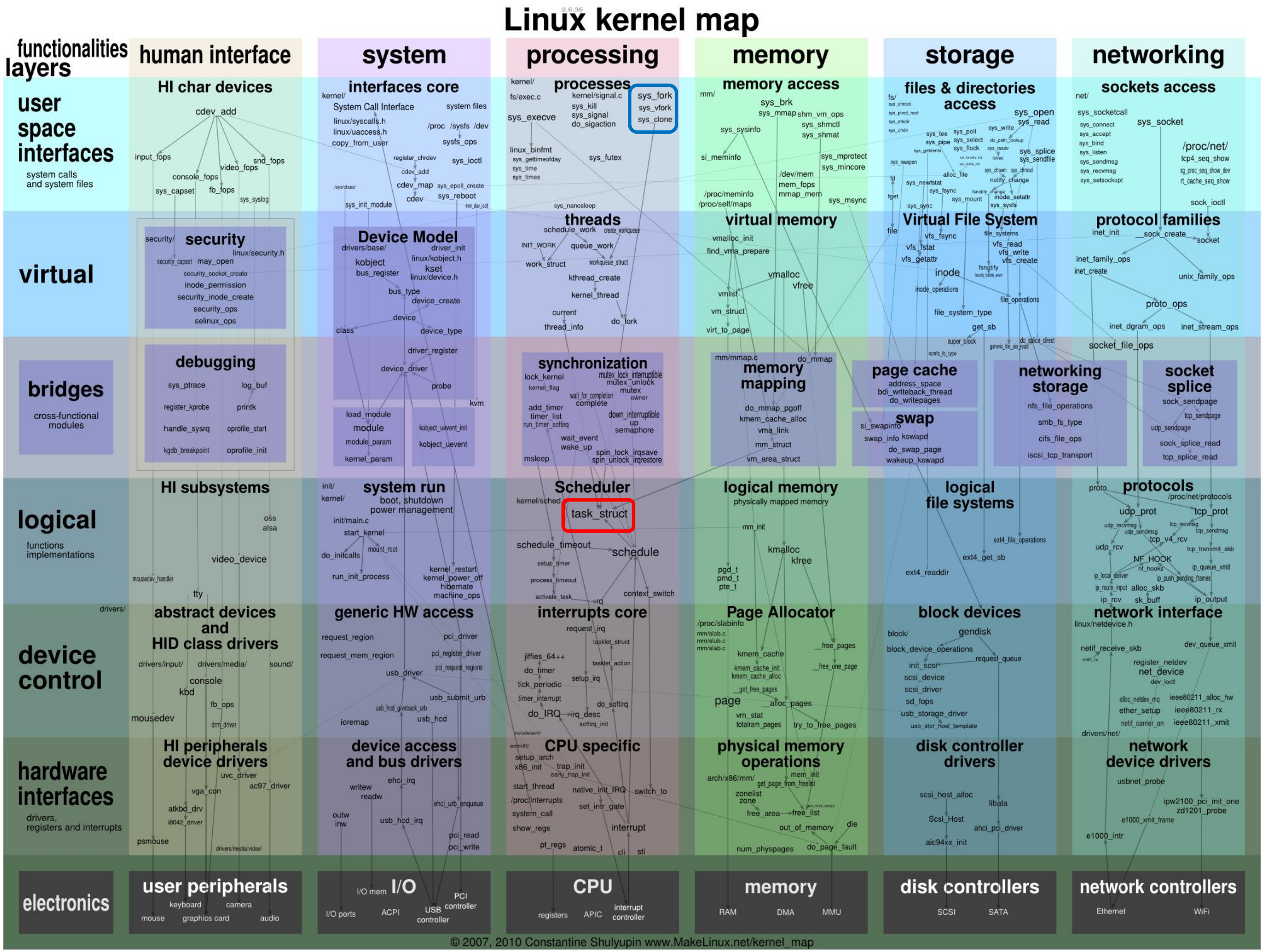
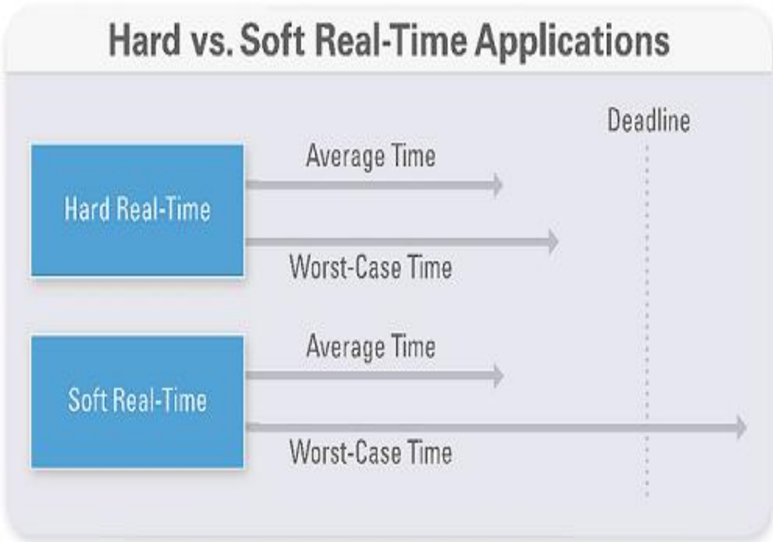


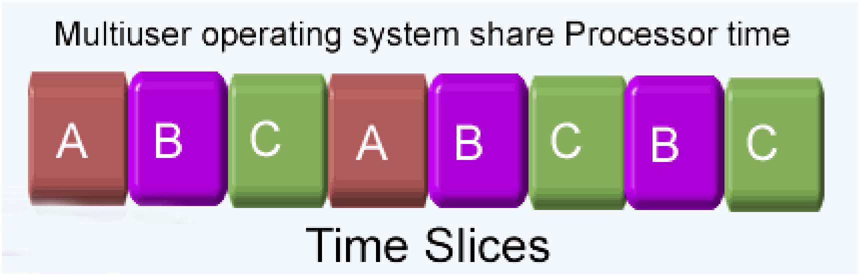
Figure 1-1: High-level overview of the structure of the Linux kernel and the layers in a complete Linux system.

Process Management and Scheduling



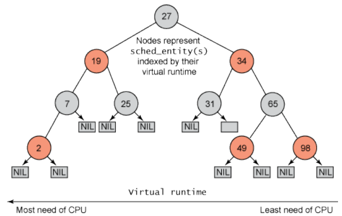
Process Priorities: real-time, non-real time





- Hard real-time processes
  - Air traffic control, Vehicle subsystems control
  - Nuclear power plant control
- Soft real-time processes
  - Multimedia transmission and reception
  - Networking, telecom networks, Web sites and services
  - Computer games

Process Life Cycle



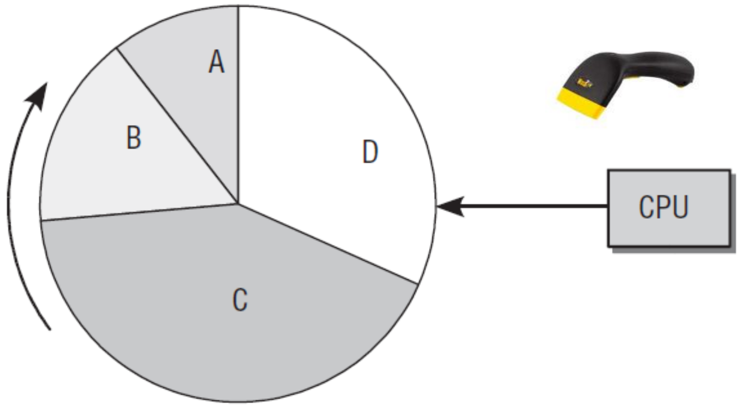
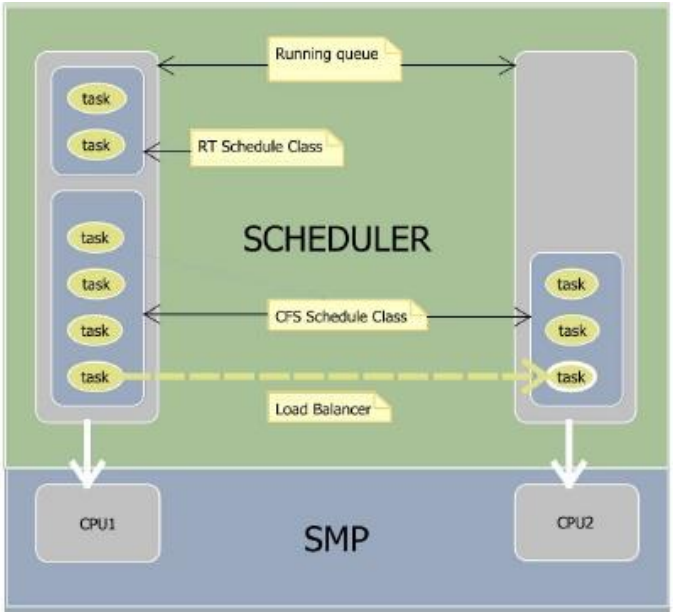
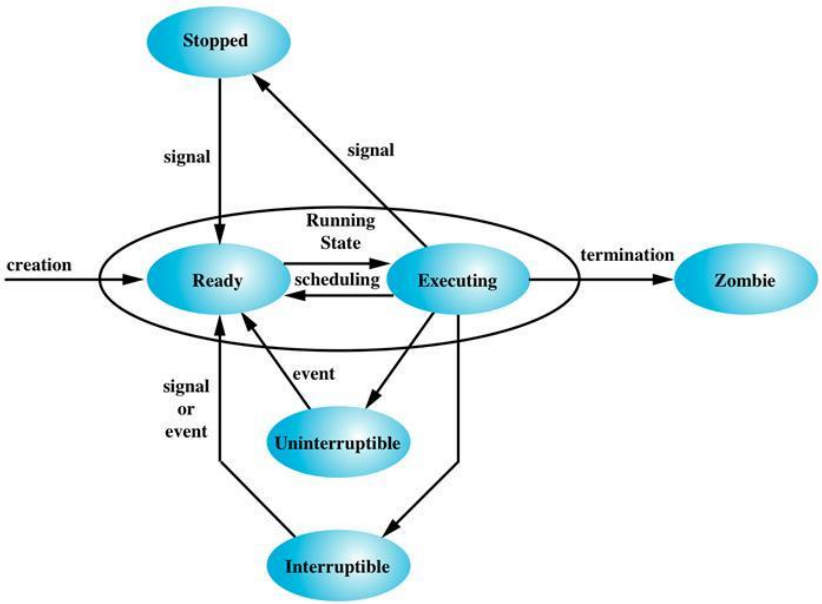


Figure 2-1: Allocation of CPU time by means of time slices.



- Preemptive multitasking
  - The length of the time slice varies depending on the importance of the process
  - Linux supports different scheduling classes
    - completely fair scheduling
    - real-time scheduling

Process Life Cycle




```
/*
 * wake_up_new_task - wake up a newly created task for the first time.
 *
 * This function will do some initial scheduler statistics housekeeping
 * that must be done for every newly created context, then puts the task
 * on the runqueue and wakes it.
 */
void wake_up_new_task(struct task_struct *p)
{
    struct rq_flags rf;
    struct rq *rq;

    raw_spin_lock_irqsave(&p->pi_lock, rf.flags);
    p->state = TASK_RUNNING;
```

```
int do_wait_intr(wait_queue_head_t *wq, wait_queue_entry_t *wait)
{
    if (likely(list_empty(&wait->entry)))
        __add_wait_queue_entry_tail(wq, wait);

    set_current_state(TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        return -ERESTARTSYS;

    spin_unlock(&wq->lock);
    schedule();
    spin_lock(&wq->lock);
    return 0;
}
EXPORT_SYMBOL(do_wait_intr);
```

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

    __set_current_state(TASK_RUNNING);
```

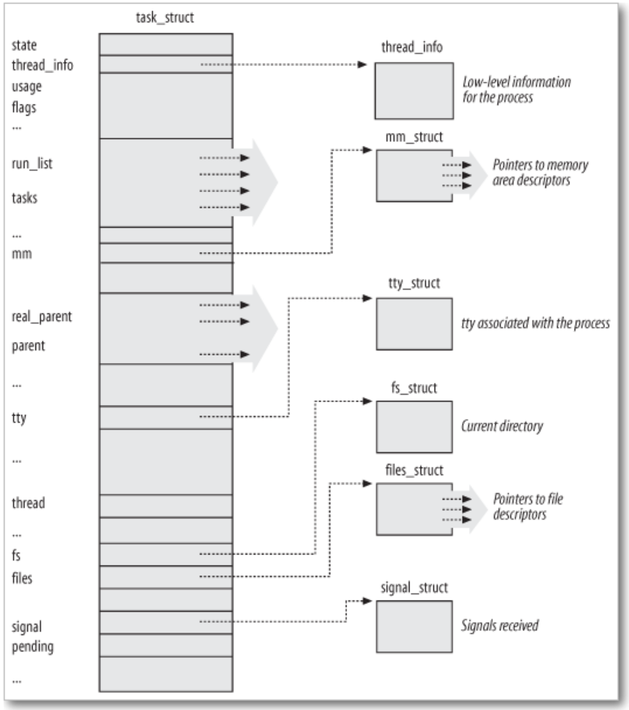
- Process State
  - Ready, Executing
  - Sleep(Interruptible, Uninterruptible)
    - Marked, Lock, Event, Signal
  - Stopped
  - Zombie
    - The wait call may be executed in sequential code, but it is commonly executed in a handler for the SIGCHILD signal, which the parent receives whenever a child has died
    - ptrace

Process Representation

```c
unsigned long wait_task_inactive(struct task_struct *p, long match_state)
{
```

```c
        break;

    /*
     * Was it really running after all now that we
     * checked with the proper locks actually held?
     *
     * Oops. Go back and try again..
     */
    if (unlikely(running)) {
        cpu_relax();
        continue;
    }

    /*
     * It's not enough that it's not actively running,
     * it must be off the runqueue _entirely_, and not
     * preempted!
     *
     * So if it was still runnable (but just not actively
     * running right now), it's preempted, and we should
     * yield - it could be a while.
     */
    if (unlikely(queued)) {
        ktime_t to = NSEC_PER_SEC / HZ;

        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule_hrtimeout(&to, HRTIMER_MODE_REL);
        continue;
    }
```

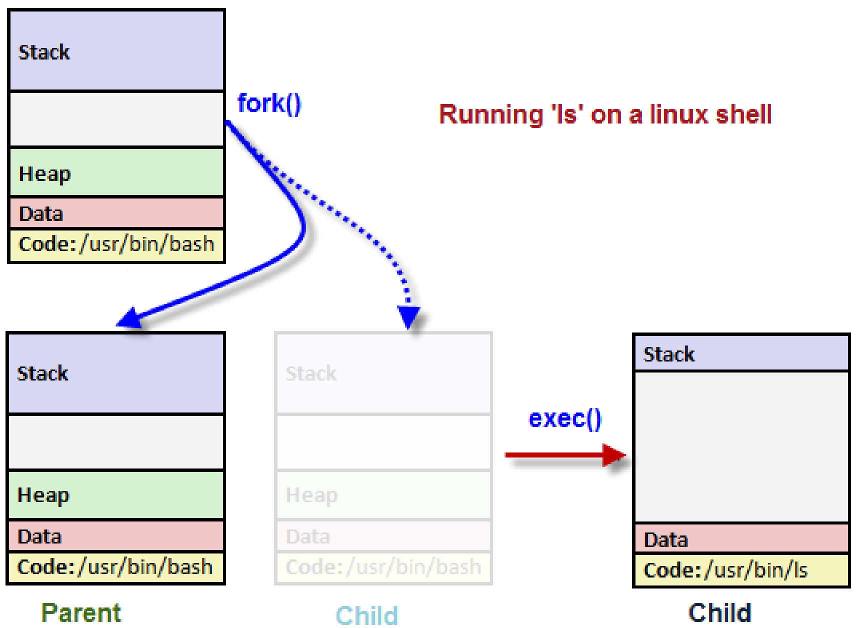
Process Representation




```c
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	/* -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;
	atomic_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;
	unsigned int			ptrace;

#ifdef CONFIG_SMP
	struct llist_node		wake_entry;
	int				on_cpu;
#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;

	int				wake_cpu;
#endif
	int				on_rq;

	int				prio;
	int				static_prio;
	int				normal_prio;
	unsigned int			rt_priority;

	const struct sched_class	*sched_class;
	struct sched_entity		se;
	struct sched_rt_entity		rt;
#ifdef CONFIG_CGROUP_SCHED
	struct task_group		*sched_task_group;
#endif
	struct sched_dl_entity		dl;
```

- task_struct
  - 500 lines
  - thread Low-level information
  - virtual memory
  - open file, descriptor, file system, signals, signal handlers
  - rlimit

Process Type



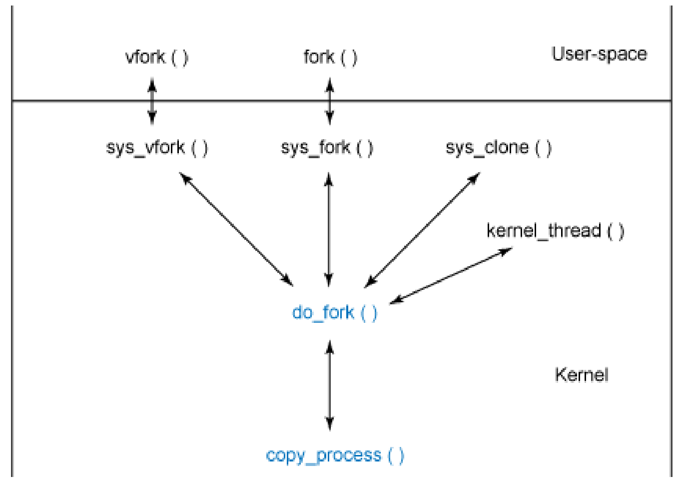

```
/*
 * sys_execve() executes a new program.
 */
static int do_execveat_common(int fd, struct filename *filename,
                              struct user_arg_ptr argv,
                              struct user_arg_ptr envp,
                              int flags)
{
```

```
/*
 * This structure is used to hold the arguments that are used when loading binaries.
 */
struct linux_binprm {
    char buf[BINPRM_BUF_SIZE];
#ifdef CONFIG_MMU
    struct vm_area_struct *vma;
    unsigned long vma_pages;
#else
# define MAX_ARG_PAGES  32
    struct page *page[MAX_ARG_PAGES];
#endif
    struct mm_struct *mm;
    unsigned long p; /* current top of mem */
```

- Fork()
- Exec()
  - Load binary file, fs/exec.c
- Clone()
  - Clone is used to implement threads
  - Both the clone() and fork() library function are wrappers around the clone() syscall

Process Type

```c
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
#else
    /* can not support in nommu mode */
    return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
SYSCALL_DEFINE0(vfork)
{
    return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
            0, NULL, NULL, 0);
}
#endif

#ifdef __ARCH_WANT_SYS_CLONE
#ifdef CONFIG_CLONE_BACKWARDS
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
         int __user *, parent_tidptr,
         unsigned long, tls,
         int __user *, child_tidptr)
#elif defined(CONFIG_CLONE_BACKWARDS2)
SYSCALL_DEFINE5(clone, unsigned long, newsp, unsigned long, clone_flags,
         int __user *, parent_tidptr,
         int __user *, child_tidptr,
         unsigned long, tls)
#elif defined(CONFIG_CLONE_BACKWARDS3)
SYSCALL_DEFINE6(clone, unsigned long, clone_flags, unsigned long, newsp,
        int, stack_size,
        int __user *, parent_tidptr,
        int __user *, child_tidptr,
        unsigned long, tls)
#else
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
         int __user *, parent_tidptr,
         int __user *, child_tidptr,
         unsigned long, tls)
#endif
{
    return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
}
#endif
```




```c
p->pid = pid_nr(pid);
if (clone_flags & CLONE_THREAD) {
    p->exit_signal = -1;
    p->group_leader = current->group_leader;
    p->tgid = current->tgid;
} else {
    if (clone_flags & CLONE_PARENT)
        p->exit_signal = current->group_leader->exit_signal;
    else
        p->exit_signal = (clone_flags & CSIGNAL);
    p->group_leader = p;
    p->tgid = p->pid;
}
```

```c
#define CLONE_PARENT    0x00008000  /* set if we want to have the same parent as the cloner */
#define CLONE_THREAD    0x00010000  /* Same thread group? */
```

Process Representation

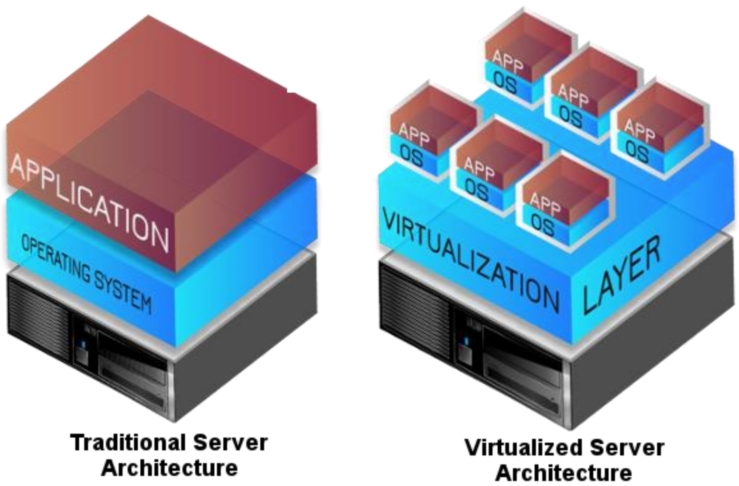
```c
long _do_fork(unsigned long clone_flags,
          unsigned long stack_start,
          unsigned long stack_size,
          int __user *parent_tidptr,
          int __user *child_tidptr,
          unsigned long tls)
{
    struct task_struct *p;
    int trace = 0;
    long nr;
```

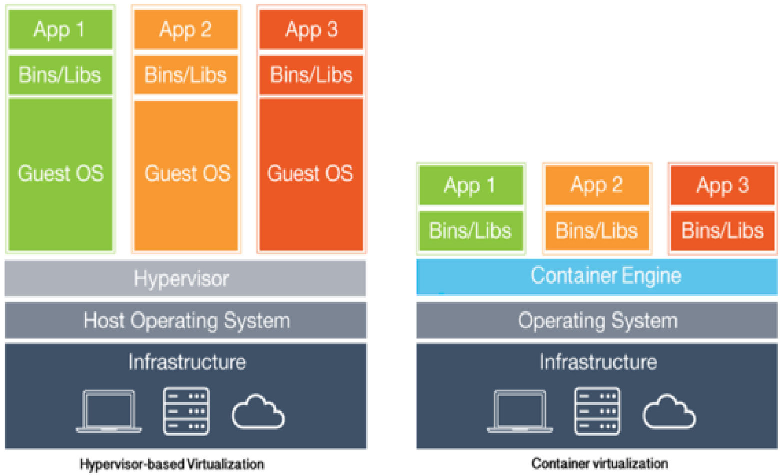
```c
p = copy_process(clone_flags, stack_start, stack_size,
         child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
add_latent_entropy();
```

```c
/*
 * Do this prior waking up the new thread - the thread pointer
 * might get invalid after that point, if the thread exits quickly.
 */
if (!IS_ERR(p)) {
    struct completion vfork;
    struct pid *pid;

    trace_sched_process_fork(current, p);

    pid = get_task_pid(p, PIDTYPE_PID);
    nr = pid_vnr(pid);

    if (clone_flags & CLONE_PARENT_SETTID)
        put_user(nr, parent_tidptr);

    if (clone_flags & CLONE_VFORK) {
        p->vfork_done = &vfork;
        init_completion(&vfork);
        get_task_struct(p);
    }

    wake_up_new_task(p);

    /* forking complete and child started to run, tell ptracer */
    if (unlikely(trace))
        ptrace_event_pid(trace, pid);

    if (clone_flags & CLONE_VFORK) {
        if (!wait_for_vfork_done(p, &vfork))
            ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
    }

    put_pid(pid);
} else {
    nr = PTR_ERR(p);
}
return nr;
```

```c
/*
 *  switch_to(x,y) should switch tasks from x to y.
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single tes
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer not supported too.
 */
__visible __notrace_funcgraph struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread;
    struct thread_struct *next = &next_p->thread;
    struct fpu *prev_fpu = &prev->fpu;
    struct fpu *next_fpu = &next->fpu;
    int cpu = smp_processor_id();
    struct tss_struct *tss = &per_cpu(cpu_tss, cpu);
    unsigned prev_fsindex, prev_gsindex;

    switch_fpu_prepare(prev_fpu, cpu);

    /* We must save %fs and %gs before load_TLS() because
     * %fs and %gs may be cleared by load_TLS().
     *
     * (e.g. xen_load_tls())
     */
    savesegment(fs, prev_fsindex);
    savesegment(gs, prev_gsindex);

    /*
     * Load TLS before restoring any segments so that segment loads
     * reference the correct GDT entries.
     */
    load_TLS(next, cpu);

    /*
     * Leave lazy mode, flushing any hypercalls made here.  This
     * must be done after loading TLS entries in the GDT but before
     * loading segments that might reference them, and and it must
     * be done before fpu__restore(), so the TS bit is up to
     * date.
     */
    arch_end_context_switch(next_p);

    /* Switch DS and ES.
     *
     * Reading them only returns the selectors, but writing them (if
     * nonzero) loads the full descriptor from the GDT or LDT.  The
     * LDT for next is loaded in switch_mm, and the GDT is loaded
     * above.
     *
     * We therefore need to write new values to the segment
     * registers on every context switch unless both the new and old
     * values are zero.
     *
     * Note that we don't need to do anything for CS and SS, as
     * those are saved and restored as part of pt_regs.
     */
    savesegment(es, prev->es);
    if (unlikely(next->es | prev->es))
        loadsegment(es, next->es);

    savesegment(ds, prev->ds);
    if (unlikely(next->ds | prev->ds))
        loadsegment(ds, next->ds);
```

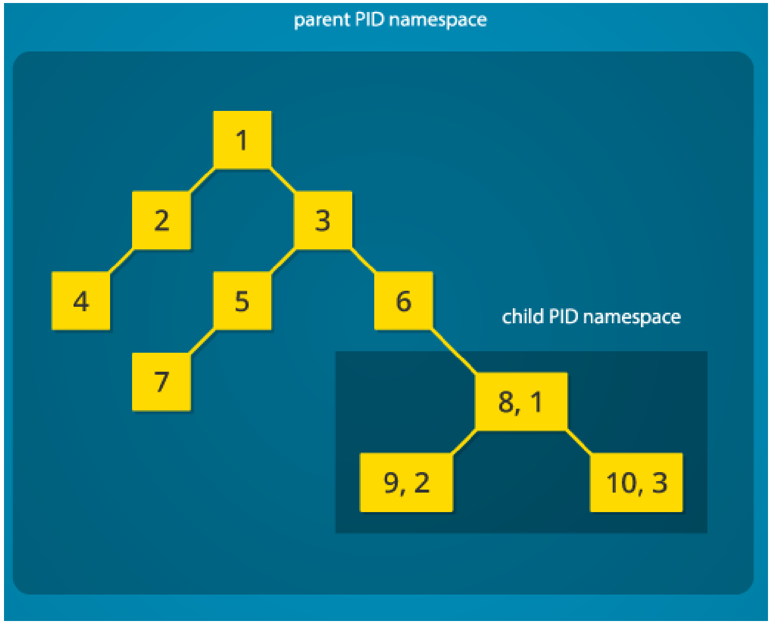
Namespace: Virtualization

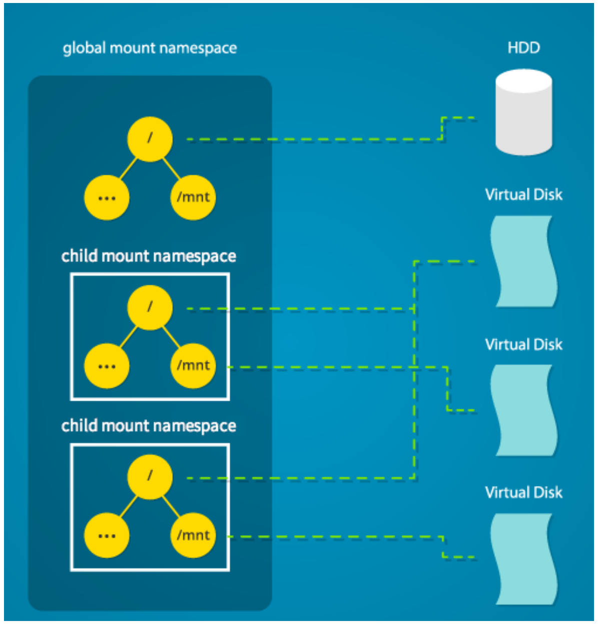




- Virtualization
  - One of the most actively adopted technologies in computer systems
  - Increasing hardware utilization, reducing the total cost of ownership, and enhancing flexibility
  - Used for cloud computing, large scale data centers, server consolidation…

Namespaces





- Namespace
  - Namespaces essentially create different *views* of the system
  - Namespaces can be hierarchically related
  - chroot syscall

Namespace: Example Code

```c
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

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;

}
```

```
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 62022
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 63038
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 63515
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 63769
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 64046
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 64287
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 65461
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 713
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 6355
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 6684
PID: 1
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 6949
PID: 1
root@embedded11:/home/hoon/K-study#
```

```c
pid_t child_pid = clone(child_fn, child_stack+1048576, SIGCHLD, NULL);
```

```c
#define CLONE_NEWCGROUP     0x02000000  /* New cgroup namespace */
#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 */
#define CLONE_NEWNET        0x40000000  /* New network namespace */
```

```
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 17985
PID: 17985
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 17987
PID: 17987
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 17989
PID: 17989
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 17991
PID: 17991
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 17993
PID: 17993
root@embedded11:/home/hoon/K-study# ./user_namespace.out
clone() = 17995
PID: 17995
root@embedded11:/home/hoon/K-study#
```

Namespace: Implementation

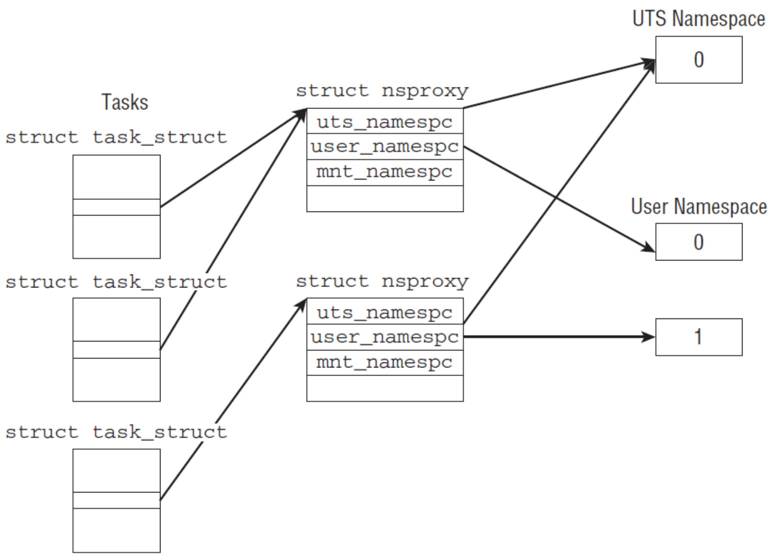


Figure 2-4: Connection between processes and namespaces.

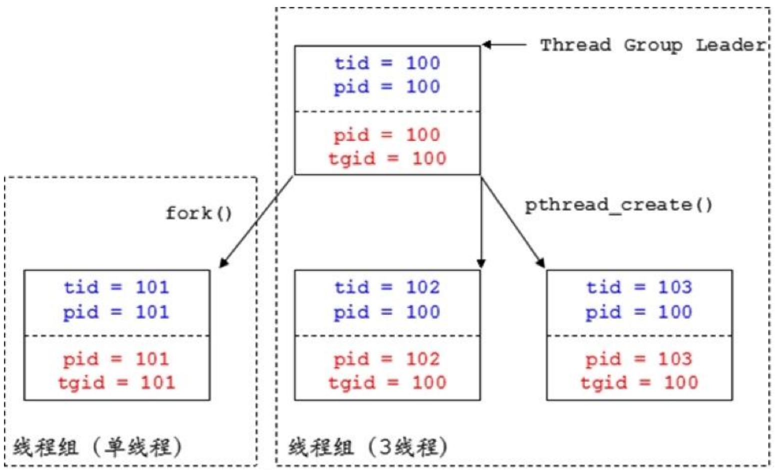
- nsproxy, X_namespace
  - requires two components
    - per-subsystem namespace structures
    - mechanism that associates a given process with the individual namespaces to which it belongs
  - Because a pointer is used, a collection of sub-namespaces can be shared among multiple processes

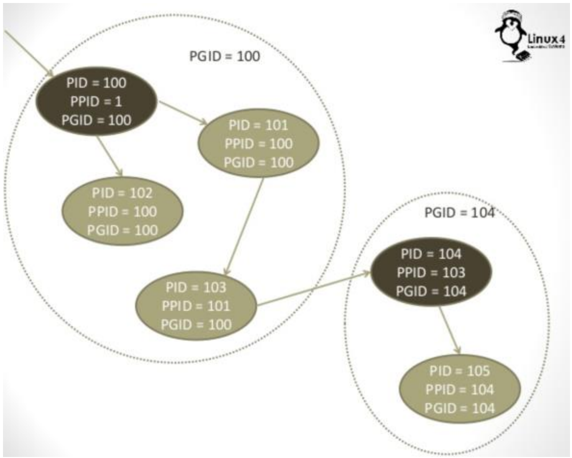
Process Identification Numbers

```
pid_t                    pid;
pid_t                    tgid;
```

< task_struct >

```
enum pid_type
{
    PIDTYPE_PID,
    PIDTYPE_PGID,
    PIDTYPE_SID,
    PIDTYPE_MAX,
    /* only valid to __task_pid_nr_ns() */
    __PIDTYPE_TGID
};
```





- Process Identification, Group
  - TID
  - PID, TGID
  - PGID, SID

PID: Managing Data Structure

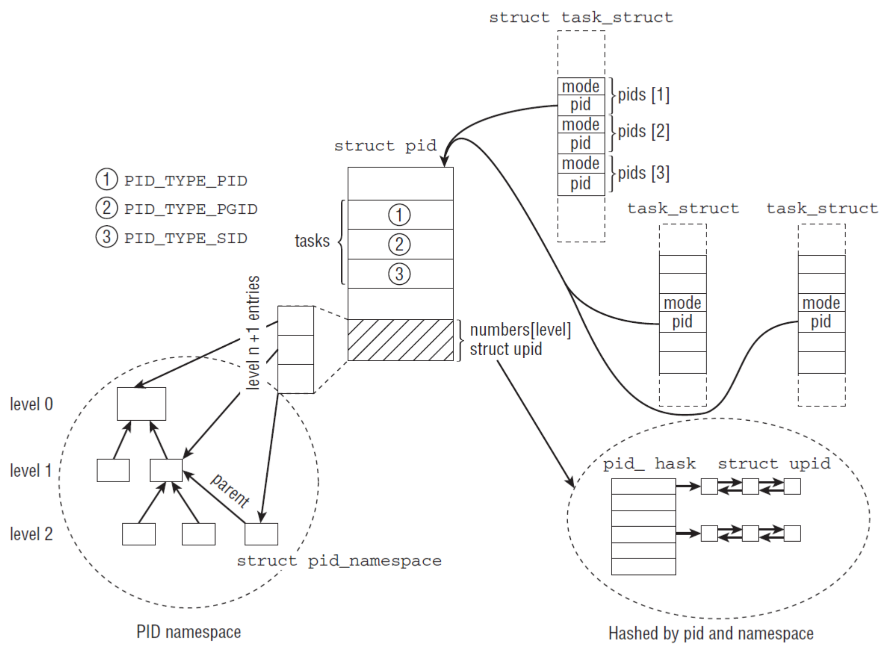


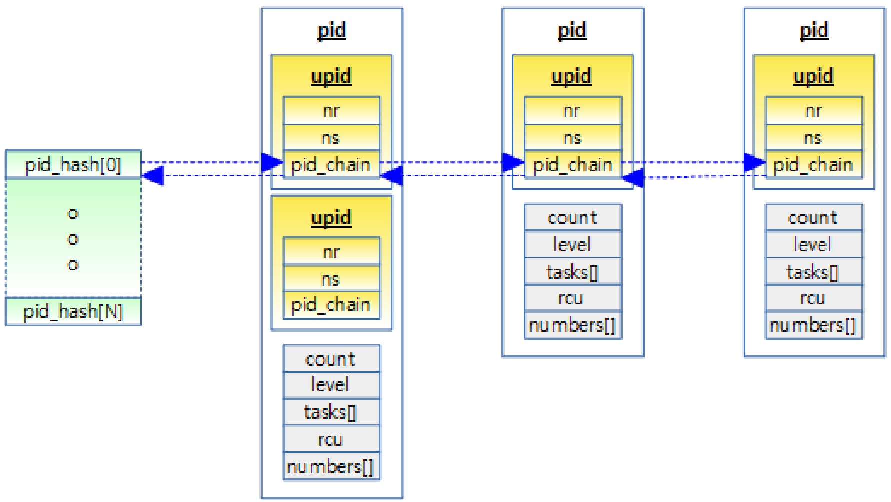
Figure 2-5: Overview of data structures used to implement a namespace-aware representation of IDs.

```
/* PID/PID hash table linkage. */
struct pid_link         pids[PIDTYPE_MAX];
struct list_head        thread_group;
struct list_head        thread_node;
```

```
struct upid {
    /* Try to keep pid_chain in the same cacheline as nr for find_vpid */
    int nr;
    struct pid_namespace *ns;
    struct hlist_node pid_chain;
};

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];
};

extern struct pid init_struct_pid;

struct pid_link
{
    struct hlist_node node;
    struct pid *pid;
};
```

- Data structure and Function
  - Level, Global, Local PID
  - A hash table is used to find the pid instance that belongs to a numeric PID value in a given namespace
  - All upid instances are kept on a hash table to which we will come in a moment, and pid_chain allows for implementing hash overflow lists with standard methods of the kernel

PID: Managing Data Structure




```
struct task_struct *find_task_by_vpid(pid_t nr);
struct task_struct *find_task_by_pid_ns(pid_t nr, struct pid_namespace *ns);
```

```
pid_t pid_nr_ns(struct pid *pid, struct pid_namespace *ns)
{
    struct upid *upid;
    pid_t nr = 0;

    if (pid && ns->level <= pid->level) {
        upid = &pid->numbers[ns->level];
        if (upid->ns == ns)
            nr = upid->nr;
    }
    return nr;
}
EXPORT_SYMBOL_GPL(pid_nr_ns);
```

- Data structure and Function
  - PIDTYPE
  - Level
    - Global, Local PID
  - A hash table is used to find the pid instance that belongs to a numeric PID value in a given namespace

PID: Generating Unique PID

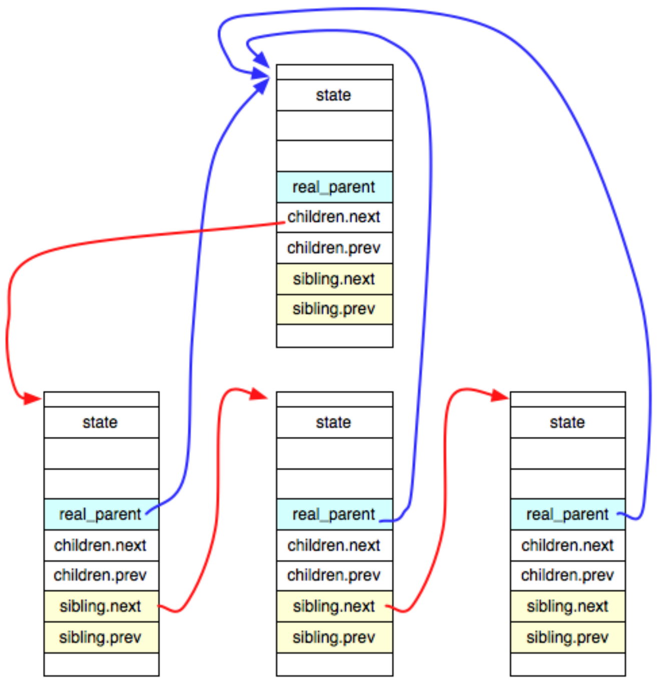
```c
struct pid *alloc_pid(struct pid_namespace *ns)
{
```

```c
tmp = ns;
pid->level = ns->level;
for (i = ns->level; i >= 0; i--) {
    nr = alloc_pidmap(tmp);
    if (nr < 0) {
        retval = nr;
        goto out_free;
    }

    pid->numbers[i].nr = nr;
    pid->numbers[i].ns = tmp;
    tmp = tmp->parent;
}
```

```c
static int alloc_pidmap(struct pid_namespace *pid_ns)
{
```

```c
pid = last + 1;
if (pid >= pid_max)
    pid = RESERVED_PIDS;
offset = pid & BITS_PER_PAGE_MASK;
map = &pid_ns->pidmap[pid/BITS_PER_PAGE];
/*
 * If last_pid points into the middle of the map->page we
 * want to scan this bitmap block twice, the second time
 * we start with offset == 0 (or RESERVED_PIDS).
 */
max_scan = DIV_ROUND_UP(pid_max, BITS_PER_PAGE) - !offset;
for (i = 0; i <= max_scan; ++i) {
    if (unlikely(!map->page)) {
        void *page = kzalloc(PAGE_SIZE, GFP_KERNEL);
        /*
         * Free the page if someone raced with us
         * installing it:
         */
        spin_lock_irq(&pidmap_lock);
        if (!map->page) {
            map->page = page;
            page = NULL;
        }
        spin_unlock_irq(&pidmap_lock);
        kfree(page);
        if (unlikely(!map->page))
            return -ENOMEM;
    }
    if (likely(atomic_read(&map->nr_free))) {
        for ( ; ; ) {
            if (!test_and_set_bit(offset, map->page)) {
                atomic_dec(&map->nr_free);
                set_last_pid(pid_ns, last, pid);
                return pid;
            }
            offset = find_next_offset(map, offset);
            if (offset >= BITS_PER_PAGE)
                break;
            pid = mk_pid(pid_ns, map, offset);
            if (pid >= pid_max)
                break;
        }
    }
    if (map < &pid_ns->pidmap[(pid_max-1)/BITS_PER_PAGE]) {
        ++map;
        offset = 0;
    } else {
        map = &pid_ns->pidmap[0];
        offset = RESERVED_PIDS;
        if (unlikely(last == offset))
            break;
    }
    pid = mk_pid(pid_ns, map, offset);
}
```

Task Relationships




```
/*
 * Pointers to the (original) parent process, youngest child, younger sibling,
 * older sibling, respectively.  (p->father can be replaced with
 * p->real_parent->pid)
 */

/* 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;
struct task_struct      *group_leader;
```

- Family
  - Parent
    - 부모 태스크가 자식 태스크보다 먼저 죽는다면 자식 태스크의 부모는 init 태스크로 변경됨 real_parent와 parent는 부모 태스크를 모두 가리키는데, 자식 태스크보다 부모 태스크가 먼저 죽을 때 parent를 init 태스크로 변경
  - Child
  - Sibling
  - Group Leader