# Process Management

Jakob Frank
Dillon Tice

# Linux

- Processes are managed in memory as pointers to structs known as task_structs

- Each struct contains a large amount of information for each process, such as process id, state, name of executable, and links to other processes such as children, parents, and siblings

- fork() and exec() are primary used for the creation of new processes

# Structure

- task_struct
  - Differs by architecture
- Init
  - Statically allocated task_struct
  - /init/init_task.c

```c
struct task_struct init_task
#ifdef CONFIG_ARCH_TASK_STRUCT_ON_STACK
	__init_task_data
#endif
= {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	.thread_info	= INIT_THREAD_INFO(init_task),
	.stack_refcount = REFCOUNT_INIT(1),
#endif
	.state		= 0,
	.stack		= init_stack,
	.usage		= REFCOUNT_INIT(2),
	.flags		= PF_KTHREAD,
	.prio		= MAX_PRIO - 20,
	.static_prio	= MAX_PRIO - 20,
	.normal_prio	= MAX_PRIO - 20,
	.policy		= SCHED_NORMAL,
	.cpus_ptr	= &init_task.cpus_mask,
	.cpus_mask	= CPU_MASK_ALL,
	.nr_cpus_allowed= NR_CPUS,
	.mm		= NULL,
	.active_mm	= &init_mm,
	.restart_block	= {
		.fn = do_no_restart_syscall,
	},
	.se		= {
		.group_node	= LIST_HEAD_INIT(init_task.se.group_node),
	},
	.rt		= {
		.run_list	= LIST_HEAD_INIT(init_task.rt.run_list),
		.time_slice	= RR_TIMESLICE,
	},
	.tasks		= LIST_HEAD_INIT(init_task.tasks),
```

# task_struct fields

- state
  - A set of bits indicating process state
    - Running, stopped, interruptible, uninterrup
- flags
  - Is a process being created? Exiting? Allocating
- comm
  - Name of the executable without the path
- static_prio
  - Priority
    - Lower priority > higher priority
    - *Actual* priority determined dynamically

```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;
	refcount_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;

```

# task_struct fields

- mm and active_mm
  - Process memory descriptors
  - Active is the previous process descriptors
    - Context switching
- tasks
  - Used for linked list representation
    - Init is the head
  - Prev and next pointer
- thread_struct
  - Context switch storage
    - Registers, program counter, etc.

```c
#ifdef CONFIG_SMP
	struct plist_node		pushable_tasks;
	struct rb_node			pushable_dl_tasks;
#endif

	struct mm_struct		*mm;
	struct mm_struct		*active_mm;

	/* Per-thread vma caching: */
	struct vmacache			vmacache;

#ifdef SPLIT_RSS_COUNTING
	struct task_rss_stat		rss_stat;
#endif
```

```c
	/*
	 * New fields for task_struct should be added above here, so that
	 * they are included in the randomized portion of task_struct.
	 */
	randomized_struct_fields_end

	/* CPU-specific state of this task: */
	struct thread_struct		thread;

	/*
	 * WARNING: on x86, 'thread_struct' contains a variable-sized
	 * structure.  It *MUST* be at the end of 'task_struct'.
	 *
	 * Do not put anything below here!
	 */
```

# Overview of Linux Process Management

- With regards to creating and managing process, linux contains the following commands:
  - Commands for creating processes via fork() and exec()
    - Can also use System() but it is slow and a security risk
  - Commands for killing processes such as kill, exit, pkill, and killall
  - Commands for viewing process information such as pgrep(), ps(), and top()

# Process States

- Possible process states:
  - TASK_RUNNING
    - Running or in run queue
  - TASK_INTERRUPTIBLE
    - Sleeping
  - TASK_UNINTERRUPTIBLE
    - Sleeping but unable to be awoken
  - TASK_STOPPED
    - Task becomes a zombie (stopped but still contains an entry in the process table)

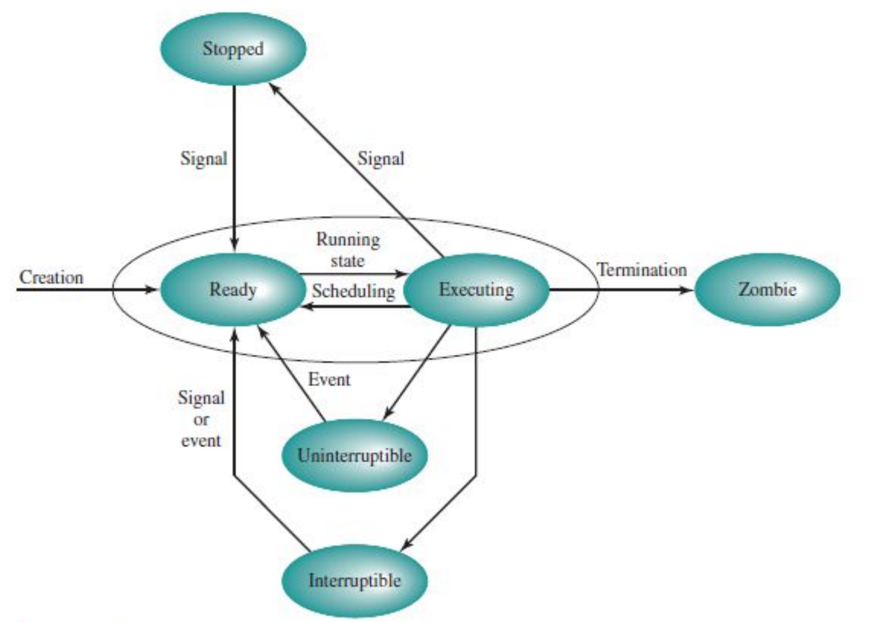


**Figure 4.15** **Linux Process/Thread Model**

# Creation

- fork() -> clone() -> do_fork()
- do_fork() - clone_flags
    - copy_process(..., args)
- copy_process()
    - Security_task_create
        - Further SELinux functions
        - Return value == 0 ?
    - dup_task_struct()
    - copy_creds()
    - CPU time reset
    - CPU time sharing setup

```c
long _do_fork(struct kernel_clone_args *args)
{
	u64 clone_flags = args->flags;
	struct completion vfork;
	struct pid *pid;
	struct task_struct *p;
	int trace = 0;
	long nr;

	/*
	 * Determine whether and which event to report to ptracer.  When
	 * called from kernel_thread or CLONE_UNTRACED is explicitly
	 * requested, no event is reported; otherwise, report if the event
	 * for the type of forking is enabled.
	 */
	if (!(clone_flags & CLONE_UNTRACED)) {
		if (clone_flags & CLONE_VFORK)
			trace = PTRACE_EVENT_VFORK;
		else if (args->exit_signal != SIGCHLD)
			trace = PTRACE_EVENT_CLONE;
		else
			trace = PTRACE_EVENT_FORK;

		if (likely(!ptrace_event_enabled(current, trace)))
			trace = 0;
	}

	p = copy_process(NULL, trace, NUMA_NO_NODE, args);
	add_latent_entropy();

	if (IS_ERR(p))
		return PTR_ERR(p);

```

_do_fork

# Destruction

POSIX defines two ways regarding how a process can be terminated

- A process can terminate itself, either by calling exit(), _exit(), returning from main(), or the last thread of the process terminates
- A process can be killed by a signal, possibly sent by the kernel, another process, or the process itself

# Alternative Ideas/Strategies

- Different built in Linux scheduling policies
  - SCHED_FIFO
  - SCHED_RR
  - SCHED_OTHER
- Different task manager implementations
  - htop
  - conky
  - pstree
  - GNOME System Monitor