Processes

Lecture 3

Copyright © Politehnica Bucharest 2025, licensed under CC BY-SA 4.0.

Processes

- Process and threads
- Context switching
- Blocking and waking up
- Process context

Process

groups together resources

- An address space
- One or more threads
- Opened files
- Sockets
- Semaphores
- Shared memory regions
- Timers
- Signal handlers

Many other resources and status information, all stored in the **Process Control Block** (*PCB*)

Threads

A thread is the basic unit that the kernel process scheduler uses to allow applications to run the CPU. A thread has the following characteristics:

- Each thread has its own stack and together with the register values it determines the thread execution state
- A thread runs in the context of a process and all threads in the same process share the resources
- The kernel schedules threads not processes and user-level threads (e.g. fibers, coroutines, etc.) are not visible at the kernel level

Processes in Tock

Bibliography

for this section

Alexandru Radovici, Ioana Culic, *Getting Started with Secure Embedded Systems*

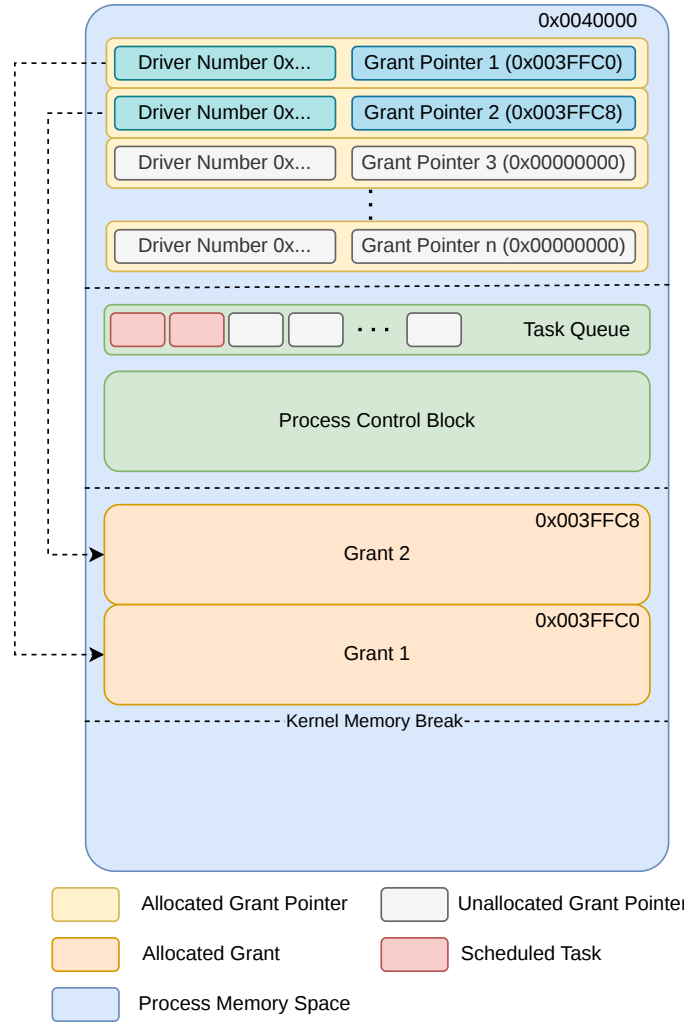
- Chapter 3 - *The Tock system architecture*

Process Control Block

information that the kernel keeps about the processes

- the actual *Process Control Block*
- the Task Queue
- Grants pointers
- Grants Data

The PCB is represented by the `Process` trait.



`Process` trait

identification

```rust
pub trait Process {
    fn processid(&self) -> ProcessId;

    /// Returns the [`ShortId`] generated by the application binary checker at loading.
    fn short_app_id(&self) -> ShortId;

    /// Returns the version number of the binary in this process, as specified
    /// in a TBF Program Header. If the binary has no version assigned this
    /// returns [`None`].
    fn binary_version(&self) -> Option<BinaryVersion>;

    /// Return the credential which the credential checker approved if the
    /// credential checker approved a credential. If the process was allowed to
    /// run without credentials, return `None`.
    fn get_credential(&self) -> Option<AcceptedCredential>;

    /// Returns how many times this process has been restarted.
    fn get_restart_count(&self) -> usize;

    /// Get the name of the process. Used for IPC.
    fn get_process_name(&self) -> &'static str;
}
```

`Process` trait

tasks

```rust
pub trait Process {
    /// Return if there are any Tasks (upcalls/IPC requests) enqueued for the process.
    fn has_tasks(&self) -> bool;

    /// Returns the number of pending tasks.
    fn pending_tasks(&self) -> usize;

    /// Queue a [`Task`] for the process. This will be added to a per-process buffer and executed by the
    // scheduler. [`Task`]s are some function the process should run, for example a upcall or an IPC call.
    fn enqueue_task(&self, task: Task) -> Result<(), ErrorCode>;

    /// Remove the scheduled operation from the front of the queue and return it
    /// to be handled by the scheduler.
    fn dequeue_task(&self) -> Option<Task>;

    /// Search the work queue for the first pending operation with the given
    /// `upcall_id` and if one exists remove it from the queue.process
    fn remove_upcall(&self, upcall_id: UpcallId) -> Option<Task>;

    /// Remove all scheduled upcalls with the given `upcall_id` from the task queue.
    /// Returns the number of removed upcalls.
```

`Process` trait

state

```rust
pub trait Process {
    /// Returns the current state the process is in.
    fn get_state(&self) -> State;

    fn ready(&self) -> bool;
    fn is_running(&self) -> bool;

    fn set_yielded_state(&self);
    fn set_yielded_for_state(&self, upcall_id: UpcallId);

    fn stop(&self);
    fn resume(&self);

    fn set_fault_state(&self);

    fn start(&self, cap: &dyn crate::capabilities::ProcessStartCapability);
    fn try_restart(&self, completion_code: Option<u32>);

    fn terminate(&self, completion_code: Option<u32>);

    fn get_completion_code(&self) -> Option<Option<u32>>;
}
```

`Process` trait

memop

```
pub trait Process {
    fn brk(&self, new_break: *const u8) -> Result<CapabilityPtr, Error>;
    fn sbrk(&self, increment: isize) -> Result<CapabilityPtr, Error>;

    fn number_writeable_flash_regions(&self) -> usize;

    fn get_writeable_flash_region(&self, region_index: usize) -> (usize, usize);

    fn update_stack_start_pointer(&self, stack_pointer: *const u8);
    fn update_heap_start_pointer(&self, heap_pointer: *const u8);

    fn build_readwrite_process_buffer(&self,
        buf_start_addr: *mut u8, size: usize,
    ) -> Result<ReadWriteProcessBuffer, ErrorCode>;
    fn build_readonly_process_buffer(&self,
        buf_start_addr: *const u8, size: usize,
    ) -> Result<ReadOnlyProcessBuffer, ErrorCode>;

    unsafe fn set_byte(&self, addr: *mut u8, value: u8) -> bool;

    fn get_command_permissions(&self, driver_num: usize, offset: usize) -> CommandPermissions;
```

`Process` trait

memory protection unit

```
pub trait Process {
    /// Configure the MPU to use the process's allocated regions.
    fn setup_mpu(&self);

    /// Allocate a new MPU region for the process that is at least
    /// `min_region_size` bytes and lies within the specified stretch of
    /// unallocated memory.
    fn add_mpu_region(
        &self,
        unallocated_memory_start: *const u8,
        unallocated_memory_size: usize,
        min_region_size: usize,
    ) -> Option<mpu::Region>;

    /// Removes an MPU region from the process that has been previously added
    /// with `add_mpu_region`.
    fn remove_mpu_region(&self, region: mpu::Region) -> Result<(), ErrorCode>;
}
```

`Process` trait

grant

```
pub trait Process {
    fn allocate_grant(&self, grant_num: usize, driver_num: usize, size: usize, align: usize) -> Result<(), ()>;
    fn grant_is_allocated(&self, grant_num: usize) -> Option<bool>;

    fn allocate_custom_grant(&self,
        size: usize, align: usize
    ) -> Result<(ProcessCustomGrantIdentifier, NonNull<u8>), ()>;

    fn enter_grant(&self, grant_num: usize) -> Result<NonNull<u8>, Error>;
    fn enter_custom_grant(&self, identifier: ProcessCustomGrantIdentifier) -> Result<*mut u8, Error>;

    unsafe fn leave_grant(&self, grant_num: usize);

    fn grant_allocated_count(&self) -> Option<usize>;

    fn lookup_grant_from_driver_num(&self, driver_num: usize) -> Result<usize, Error>;
}
```

`Process` trait

subscribe

```
pub trait Process {
    /// Verify that an upcall function pointer is within process-accessible
    /// memory.
    fn is_valid_upcall_function_pointer(&self, upcall_fn: *const ()) -> bool;

    /// Set the return value the process should see when it begins executing
    /// again after the syscall.
    fn set_syscall_return_value(&self, return_value: SyscallReturn);
}
```

`Process` trait

context switch

```rust
pub trait Process {
    fn set_process_function(&self, callback: FunctionCall);

    /// Set the function that is to be executed when the process is resumed.
    fn switch_to(&self) -> Option<syscall::ContextSwitchReason>;

    fn get_addresses(&self) -> ProcessAddresses;
    fn get_sizes(&self) -> ProcessSizes;

    fn get_stored_state(&self, out: &mut [u8]) -> Result<usize, ErrorCode>;
}
```

`Process` trait

debug

```
pub trait Process {
    fn print_full_process(&self, writer: &mut dyn Write);

    fn debug_syscall_count(&self) -> usize;
    fn debug_dropped_upcall_count(&self) -> usize;
    fn debug_timeslice_expiration_count(&self) -> usize;

    /// Increment the number of times the process has exceeded its timeslice.
    fn debug_timeslice_expired(&self);

    /// Increment the number of times the process called a syscall and record
    /// the last syscall that was called.
    fn debug_syscall_called(&self, last_syscall: Syscall);

    /// Return the last syscall the process called. Returns `None` if the
    /// process has not called any syscalls or the information is unknown.
    fn debug_syscall_last(&self) -> Option<Syscall>;
}
```

ProcessStandard

the reference implementation of the `Process` trait

```
pub struct ProcessStandard<'a, C: 'static + Chip, D: 'static + ProcessStandardDebug + Default> {
    process_id: Cell<ProcessId>,
    app_id: ShortId,

    memory_start: *const u8,
    memory_len: usize,
    grant_pointers: MapCell<&'static mut [GrantPointerEntry]>,
    kernel_memory_break: Cell<*const u8>,
    app_break: Cell<*const u8>,
    allow_high_water_mark: Cell<*const u8>,

    flash: &'static [u8],

    stored_state:
        MapCell<<<C as Chip>::UserspaceKernelBoundary as UserspaceKernelBoundary>::StoredState>,

    state: Cell<State>,

    tasks: MapCell<RingBuffer<'a, Task>>,

    /* ... */
}
```

`ProcessStandard`

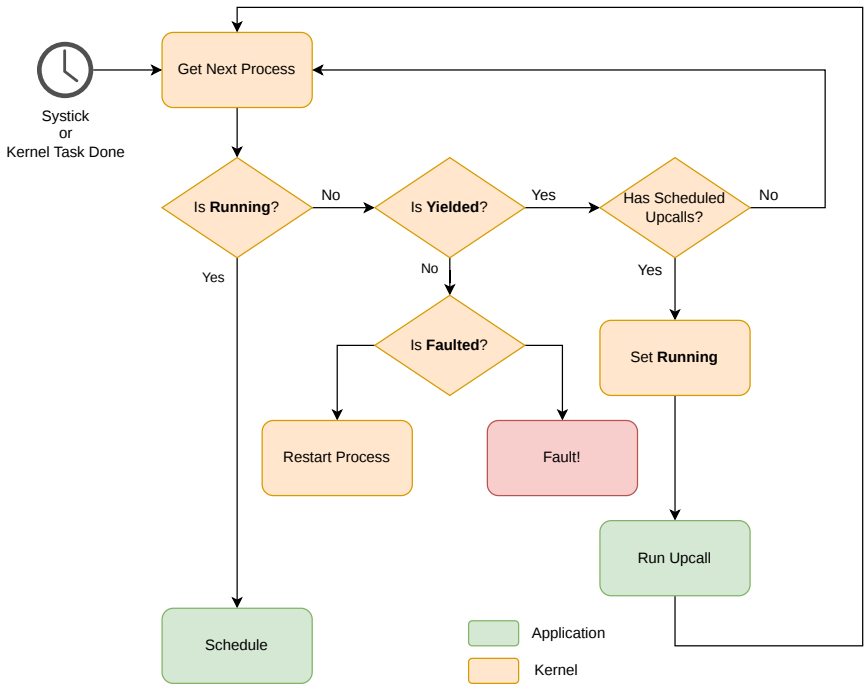
the reference implementation of the `Process` trait

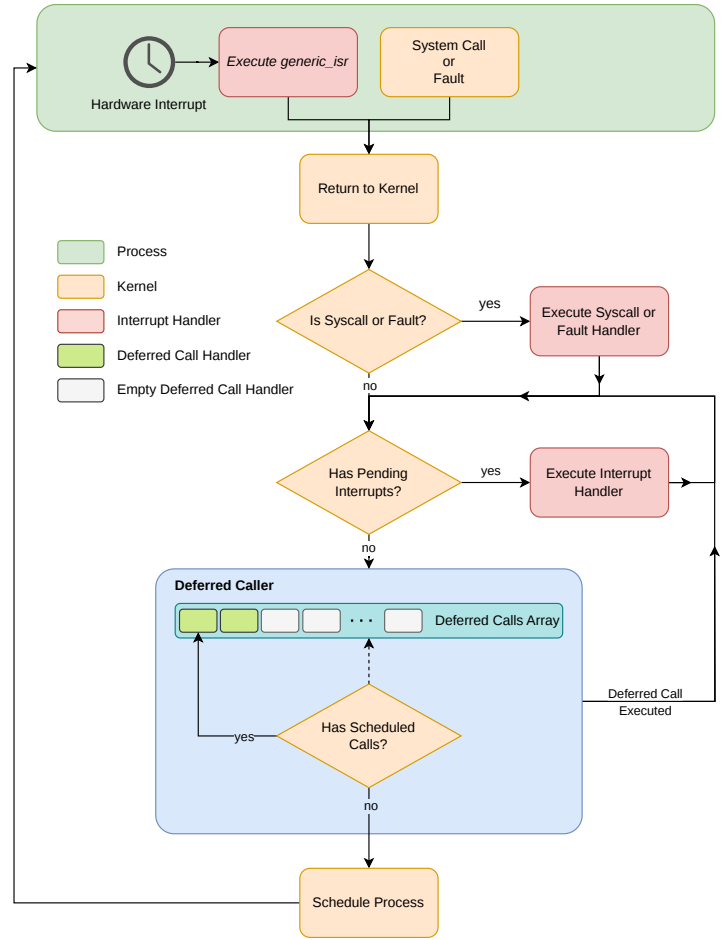
```
pub struct ProcessStandard<'a, C: 'static + Chip, D: 'static + ProcessStandardDebug + Default> {
    /* ... */
    footers: &'static [u8],
    header: tock_tbf::types::TbfHeader,
    credential: Option<AcceptedCredential>,

    kernel: &'static Kernel,
    chip: &'static C,

    fault_policy: &'a dyn ProcessFaultPolicy,
    storage_permissions: StoragePermissions,
    mpu_config: MapCell<<<C as Chip>::MPU as MPU>::MpuConfig>,
    mpu_regions: [Cell<Option<mpu::Region>>; 6],



    restart_count: Cell<usize>,
    completion_code: OptionalCell<Option<u32>>,

    debug: D,
}
```

`Scheduler` trait

```rust
pub trait Scheduler<C: Chip> {
    fn next(&self) -> SchedulingDecision;
    fn result(&self, result: StoppedExecutingReason, execution_time_us: Option<u32>);

    /// Tell the scheduler to execute kernel work such as interrupt bottom
    /// halves and dynamic deferred calls.
    unsafe fn execute_kernel_work(&self, chip: &C) {
        chip.service_pending_interrupts();
        while DeferredCall::has_tasks() && !chip.has_pending_interrupts() {
            DeferredCall::service_next_pending();
        }
    }

    /// Ask the scheduler whether to take a break from executing userspace
    /// processes to handle kernel tasks.
    unsafe fn do_kernel_work_now(&self, chip: &C) -> bool {
        chip.has_pending_interrupts() || DeferredCall::has_tasks()
    }

    /// Once a process is scheduled the kernel will try to execute it until it
    /// has no more work to do or exhausts its timeslice.
    unsafe fn continue_process(&self, _id: ProcessId, chip: &C) -> bool {
        !(chip.has_pending_interrupts() || DeferredCall::has_tasks())
```

Default Scheduling





Processes in Linux

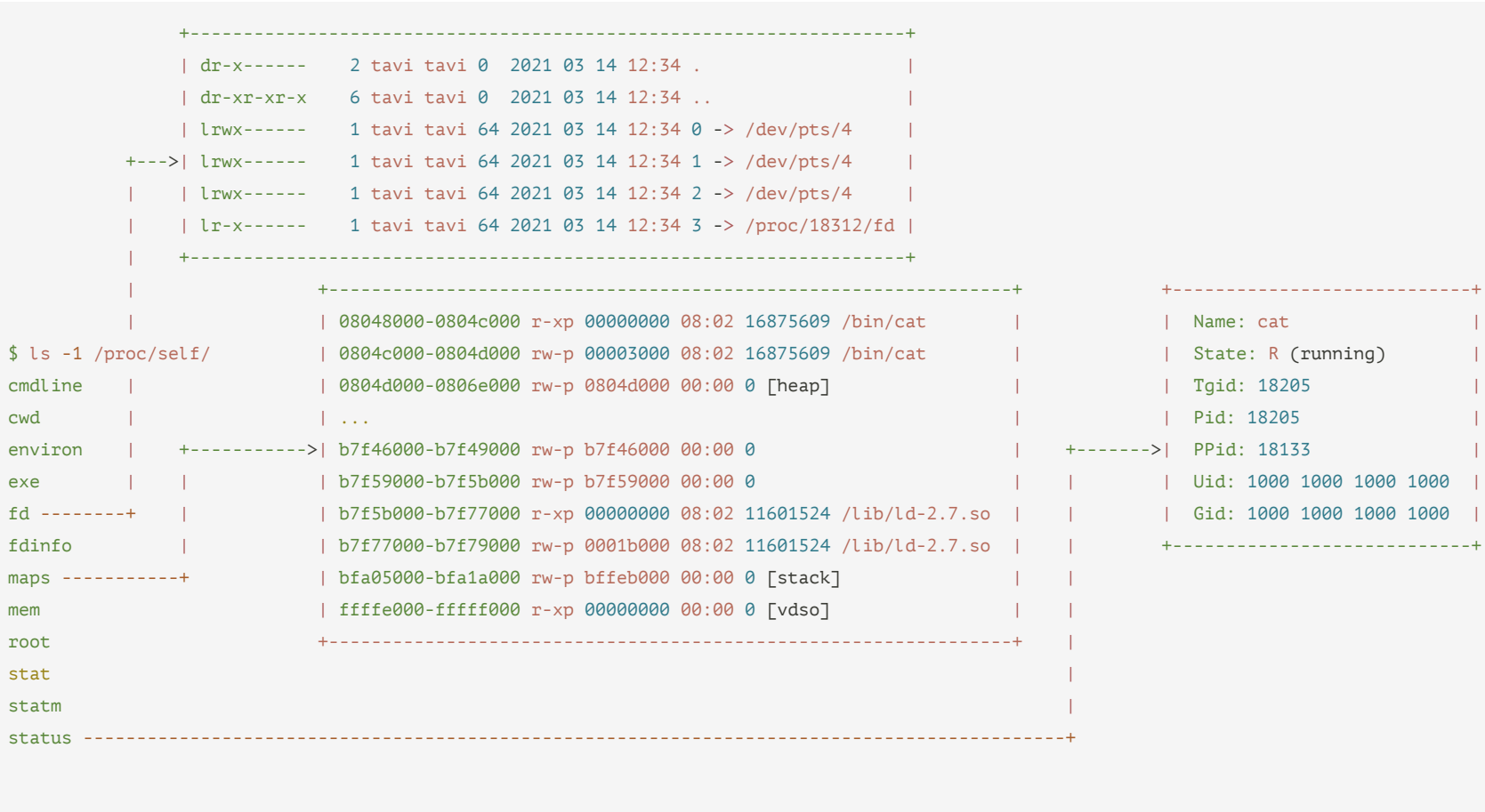
Bibliography

for this section

Daniel P. BOVET & Marco CESATI, *Understanding the LINUX KERNEL*, 3rd Edition, O'Reilly

- Chapter 3, Processes

procfs


dr-x------ 2 tavi tavi 0 2021 03 14 12:34 .
dr-xr-xr-x 6 tavi tavi 0 2021 03 14 12:34 ..
lrwx------ 1 tavi tavi 64 2021 03 14 12:34 0 -> /dev/pts/4
lrwx------ 1 tavi tavi 64 2021 03 14 12:34 1 -> /dev/pts/4
lrwx------ 1 tavi tavi 64 2021 03 14 12:34 2 -> /dev/pts/4
lr-x------ 1 tavi tavi 64 2021 03 14 12:34 3 -> /proc/18312/fd
$ ls -1 /proc/self/
cmdline
cwd
environ
exe
fd --------
fdinfo
maps -----------
mem
root
stat
statm
status
08048000-0804c000 r-xp 00000000 08:02 16875609 /bin/cat
0804c000-0804d000 rw-p 00003000 08:02 16875609 /bin/cat
0804d000-0806e000 rw-p 0804d000 00:00 0 [heap]
...
b7f46000-b7f49000 rw-p b7f46000 00:00 0
b7f59000-b7f5b000 rw-p b7f59000 00:00 0
b7f5b000-b7f77000 r-xp 00000000 08:02 11601524 /lib/ld-2.7.so
b7f77000-b7f79000 rw-p 0001b000 08:02 11601524 /lib/ld-2.7.so
bfa05000-bfa1a000 rw-p bffeb000 00:00 0 [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]
Name: cat
State: R (running)
Tgid: 18205
Pid: 18205
PPid: 18133
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000

struct task_struct

```
struct task_struct {
    struct thread_info thread_info;                 /*     0     8 */
    volatile long int          state;               /*     8     4 */
    void *                     stack;               /*    12     4 */

    ...

    /* --- cacheline 45 boundary (2880 bytes) --- */
    struct thread_struct thread __attribute__((__aligned__(64))); /*  2880  4288 */

    /* size: 7168, cachelines: 112, members: 155 */
    /* sum members: 7148, holes: 2, sum holes: 12 */
    /* sum bitfield members: 7 bits, bit holes: 2, sum bit holes: 57 bits */
    /* paddings: 1, sum paddings: 2 */
    /* forced alignments: 6, forced holes: 2, sum forced holes: 12 */
} __attribute__((__aligned__(64)));
```

Threads (Windows)

how threads should look like

The typical thread implementation is one where the threads are implemented as a separate data structure which is then linked to the process data structure. For example, the *Windows* kernel uses such an implementation:

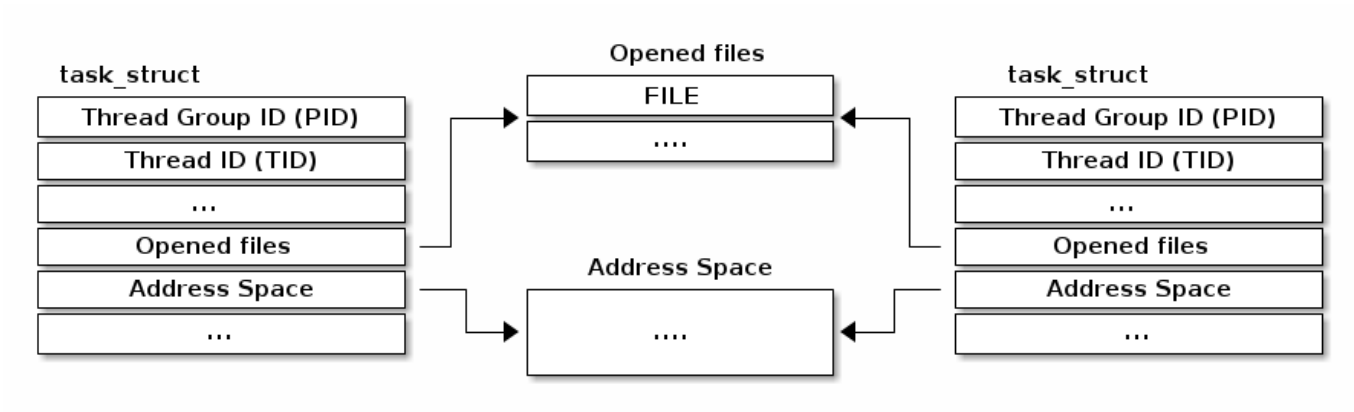
Threads

the Linux way

Linux uses a different implementation for threads. The basic unit is called a *task* (hence the `struct task_struct`) and it is used for both threads and processes.

Thus, if two threads are in the same process will point to the same resource structure instance.

If two threads are in different processes they will point to different resource structure instances.



The `clone` system call

create a process or a thread

In Linux a new thread or process is created with the `clone()` system call. Both the `fork()` system call and the `pthread_create()` function use the `clone()` implementation.

It allows the caller to decide what resources should be shared with the parent and which should be copied or isolated:

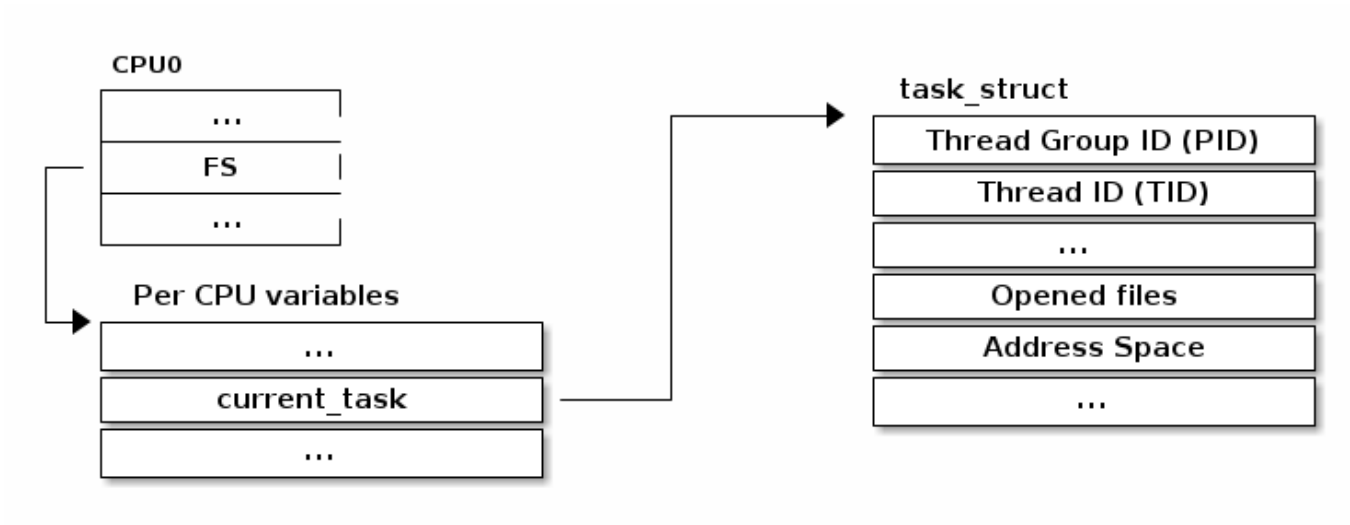
- `CLONE_FILES` - shares the file descriptor table with the parent
- `CLONE_VM` - shares the address space with the parent
- `CLONE_FS` - shares the filesystem information (root directory, current directory) with the parent
- `CLONE_NEWNS` - does not share the mount namespace with the parent
- `CLONE_NEWIPC` - does not share the IPC namespace (System V IPC objects, POSIX message queues) with the parent
- `CLONE_NEWNET` - does not share the networking namespaces (network interfaces, routing table) with the parent

Access the current `struct task_struct`

Over 90% of the system calls need to access the current process structure so it needs to be fast

- opening a file needs access to `struct task_struct`'s `file` field
- mapping a new file needs access to `struct task_struct`'s `mm` field

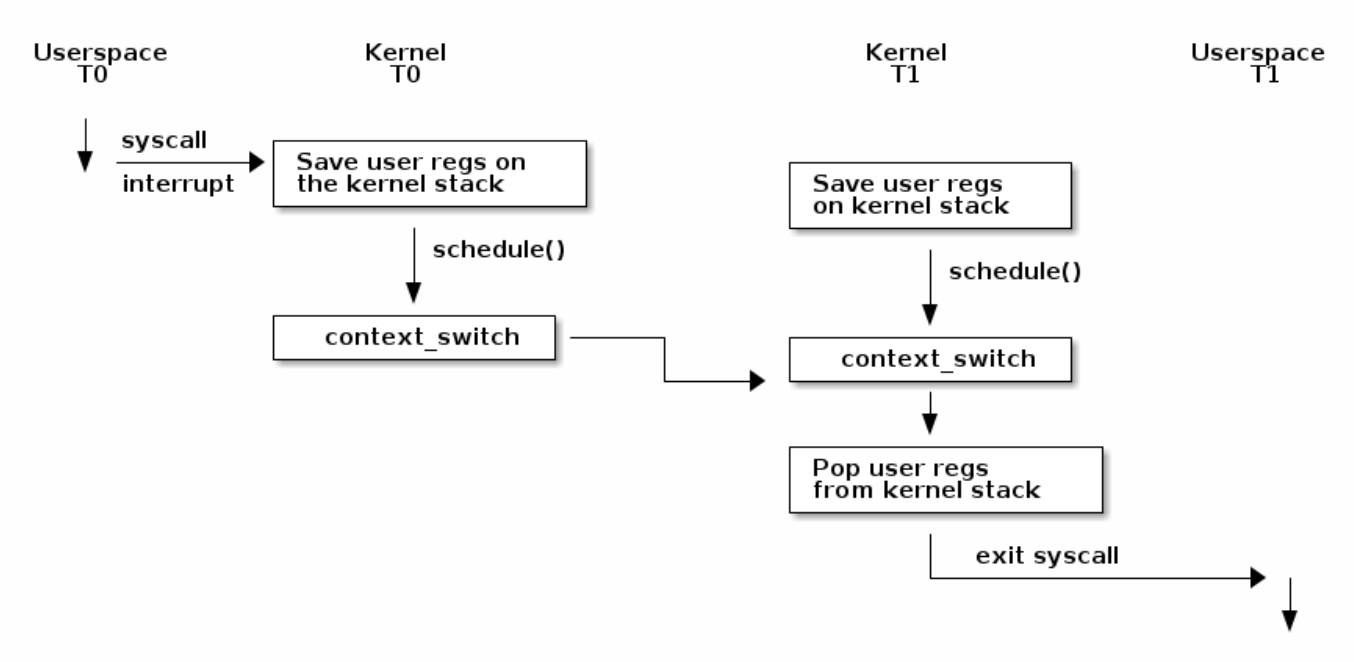
The `current` macro is available to access the current process's `struct task_struct`



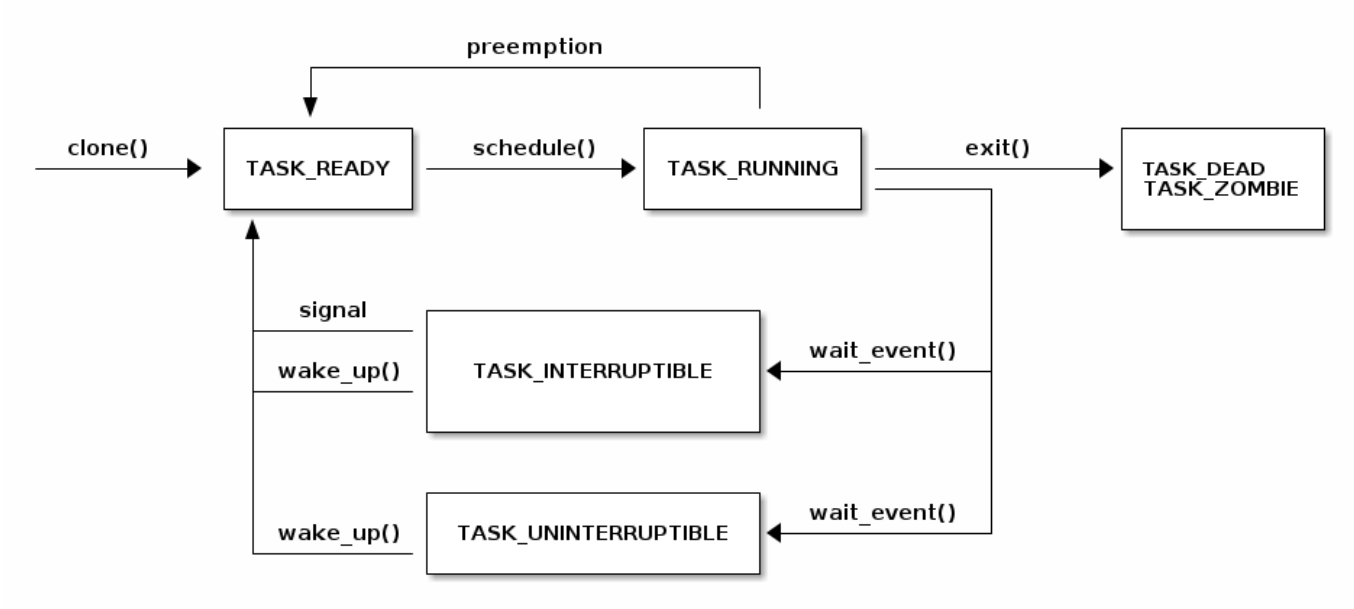
Useful process macro's

```
1   /* how to get the current stack pointer from C */
2   register unsigned long current_stack_pointer asm("esp") __attribute_used__;
3
4   /* how to get the thread information struct from C */
5   static inline struct thread_info *current_thread_info(void)
6   {
7      return (struct thread_info *)(current_stack_pointer & ~(THREAD_SIZE - 1));
8   }
9
10  #define current current_thread_info()->task
```

Context Switch



Task States


preemption
clone()
TASK_READY
schedule()
TASK_RUNNING
exit()
TASK_DEAD
TASK_ZOMBIE
signal
wake_up()
TASK_INTERRUPTIBLE
wait_event()
wake_up()
TASK_UNINTERRUPTIBLE
wait_event()

Conclusion

we talked about

- Process and threads
- Context switching
- Blocking and waking up
- Process context