SO2 Lecture 03 - Processes

View slides

Lecture objectives

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

Processes and threads ¶

A process is an operating system abstraction that groups together multiple 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 this information is grouped in the Process Control Group (PCB). In Linux this is `struct task_struct`.

Overview of process resources

A summary of the resources a process has can be obtain from the */proc/<pid>* directory, where *<pid>* is the process id for the process we want to look at.

```
                 +--------------------------------------------------------------------+
                 | 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 |
            |    +--------------------------------------------------------------------+
            |                 +------------------------------------------------------------------+
            |                 | 08048000-0804c000 r-xp 00000000 08:02 16875609 /bin/cat          |
$ ls -1 /proc/self/           | 0804c000-0804d000 rw-p 00003000 08:02 16875609 /bin/cat          |
cmdline     |                 | 0804d000-0806e000 rw-p 0804d000 00:00 0 [heap]                   |
cwd         |                 | ...                                                              |
environ     |    +----------->| b7f46000-b7f49000 rw-p b7f46000 00:00 0                          |
exe         |    |            | b7f59000-b7f5b000 rw-p b7f59000 00:00 0                          |
fd ---------+    |            | b7f5b000-b7f77000 r-xp 00000000 08:02 11601524 /lib/ld-2.7.so    |
fdinfo           |            | b7f77000-b7f79000 rw-p 0001b000 08:02 11601524 /lib/ld-2.7.so    |
maps -----------+             | bfa05000-bfa1a000 rw-p bffeb000 00:00 0 [stack]                  |
mem                           | ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]                   |
root                          +------------------------------------------------------------------+
stat                  +----------------------------+
statm                 |  Name: cat                 |
status ------+        |  State: R (running)        |
task         |        |  Tgid: 18205               |
wchan        +------>|  Pid: 18205                |
                      |  PPid: 18133               |
                      |  Uid: 1000 1000 1000 1000  |
                      |  Gid: 1000 1000 1000 1000  |
                      +----------------------------+
```

`struct task_struct`

Lets take a close look at `struct task_struct`. For that we could just look at the source code, but here we will use a tool called *pahole* (part of the dwarves install package) in order to get some insights about this structure:

```
$ pahole -C task_struct vmlinux

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)));
```

As you can see it is a pretty large data structure: almost 8KB in size and 155 fields.

Inspecting task_struct

The following screencast is going to demonstrate how we can inspect the process control block (`struct task_struct`) by connecting the debugger to the running virtual machine. We are going to use a helper gdb command *lx-ps* to list the processes and the address of the task_struct for each process.

Quiz: Inspect a task to determine opened files

Use the debugger to inspect the process named syslogd.

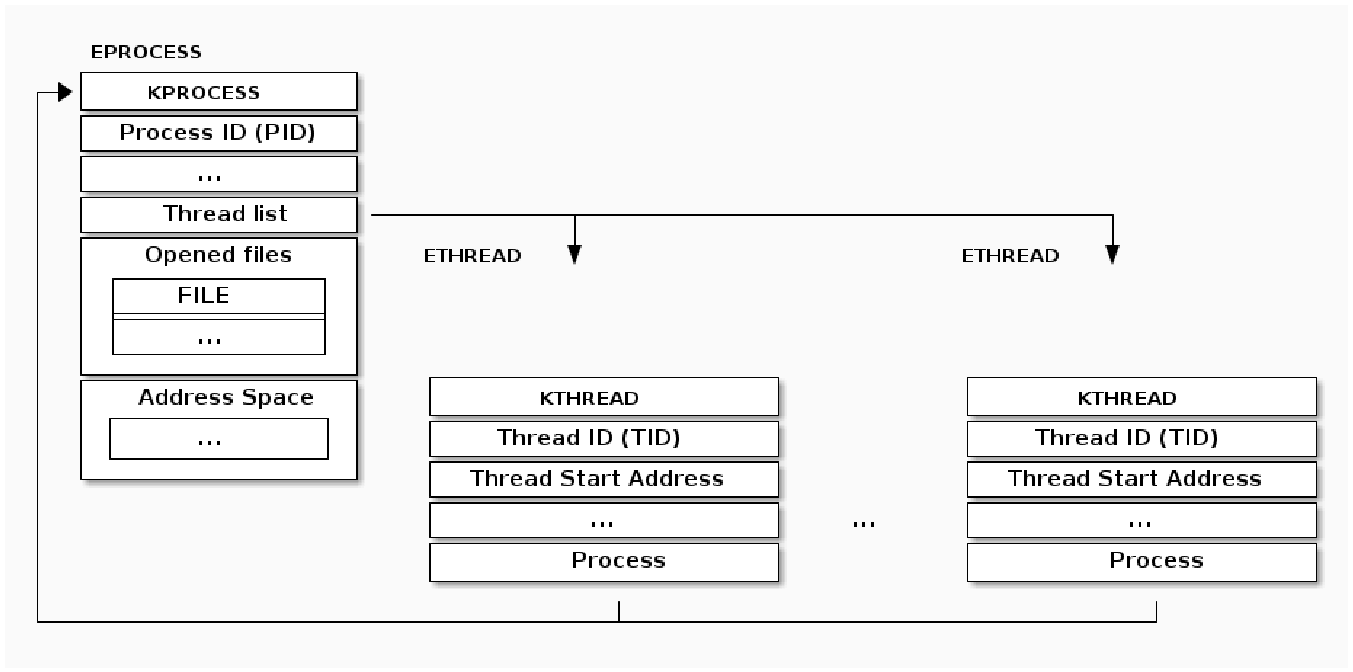
- What command should we use to list the opened file descriptors?
- How many file descriptors are opened?
- What command should we use the determine the file name for opened file descriptor 3?
- What is the filename for file descriptor 3?

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:

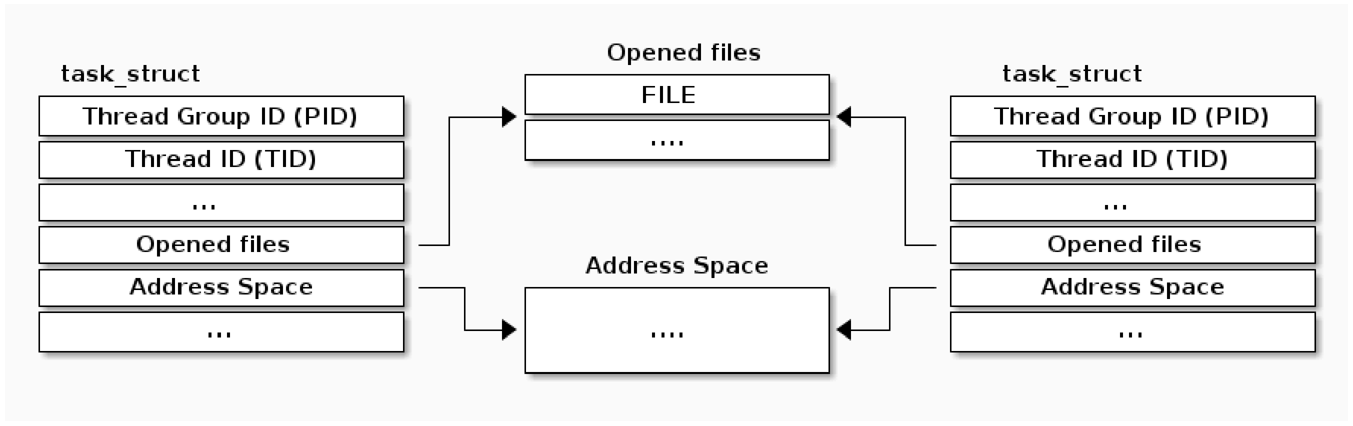
- Each thread has its own stack and together with the register values it determines the thread execution state
- A thread runs in the context of a process and all threads in the same process share the resources
- The kernel schedules threads not processes and user-level threads (e.g. fibers, coroutines,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

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



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. Instead of embedding resources in the task structure it has pointers to these resources.

Thus, if two threads are 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

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

For example, if *CLONE_FILES* | *CLONE_VM* | *CLONE_FS* is used by the caller then effectively a new thread is created. If these flags are not used then a new process is created.

Namespaces and "containers"

"Containers" are a form of lightweight virtual machines that share the same kernel instance, as opposed to normal virtualization where a hypervisor runs multiple VMs, each with its one kernel instance.

Examples of container technologies are LXC - that allows running lightweight "VM" and docker - a specialized container for running a single application.

Containers are built on top of a few kernel features, one of which is namespaces. They allow isolation of different resources that would otherwise be globally visible. For example, without containers, all processes would be visible in /proc. With containers, processes in one container will not be visible (in /proc or be killable) to other containers.

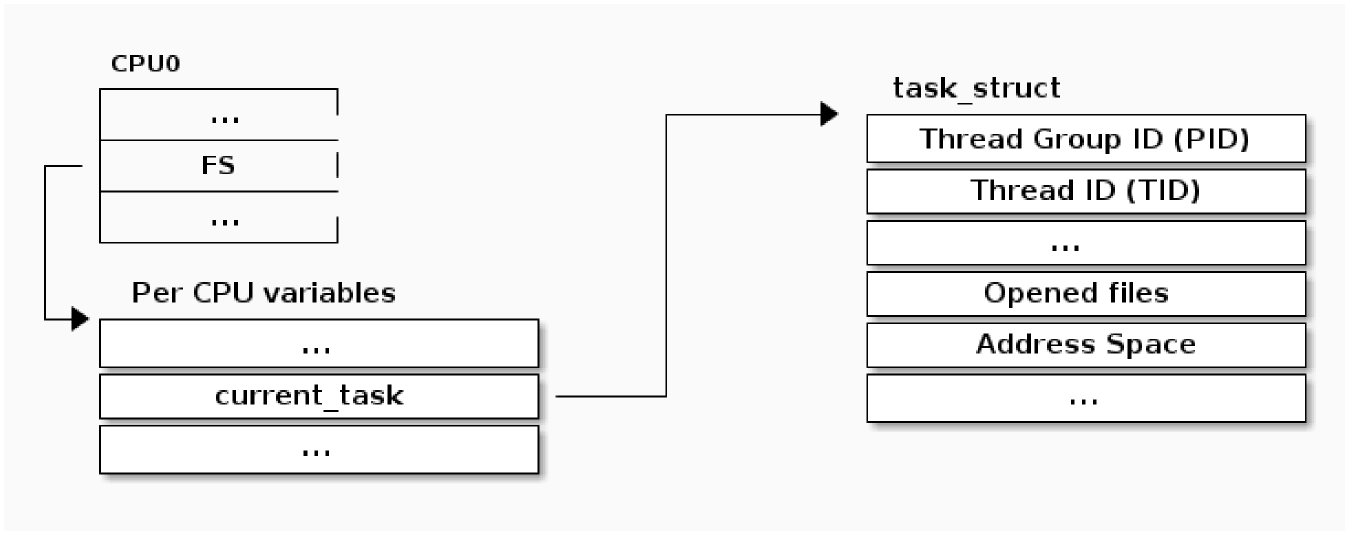
To achieve this partitioning, the `struct nsproxy` structure is used to group types of resources that we want to partition. It currently supports IPC, networking, cgroup, mount, networking, PID, time namespaces. For example, instead of having a global list for networking interfaces, the list is part of a `struct net`. The system initializes with a default namespace (`init_net`) and by default all processes will share this namespace. When a new namespace is created a new net namespace is created and then new processes can point to that new namespace instead of the default one.

Accessing the current process

Accessing the current process is a frequent operation:

- 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
- Over 90% of the system calls needs to access the current process structure so it needs to be fast
- The `current` macro is available to access to current process's `struct task_struct`

In order to support fast access in multi processor configurations a per CPU variable is used to store and retrieve the pointer to the current `struct task_struct`:



Previously the following sequence was used as the implementation for the `current` macro:

```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp") __attribute_used__;

/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
   return (struct thread_info *)(current_stack_pointer & ~(THREAD_SIZE - 1));
}

#define current current_thread_info()->task
```

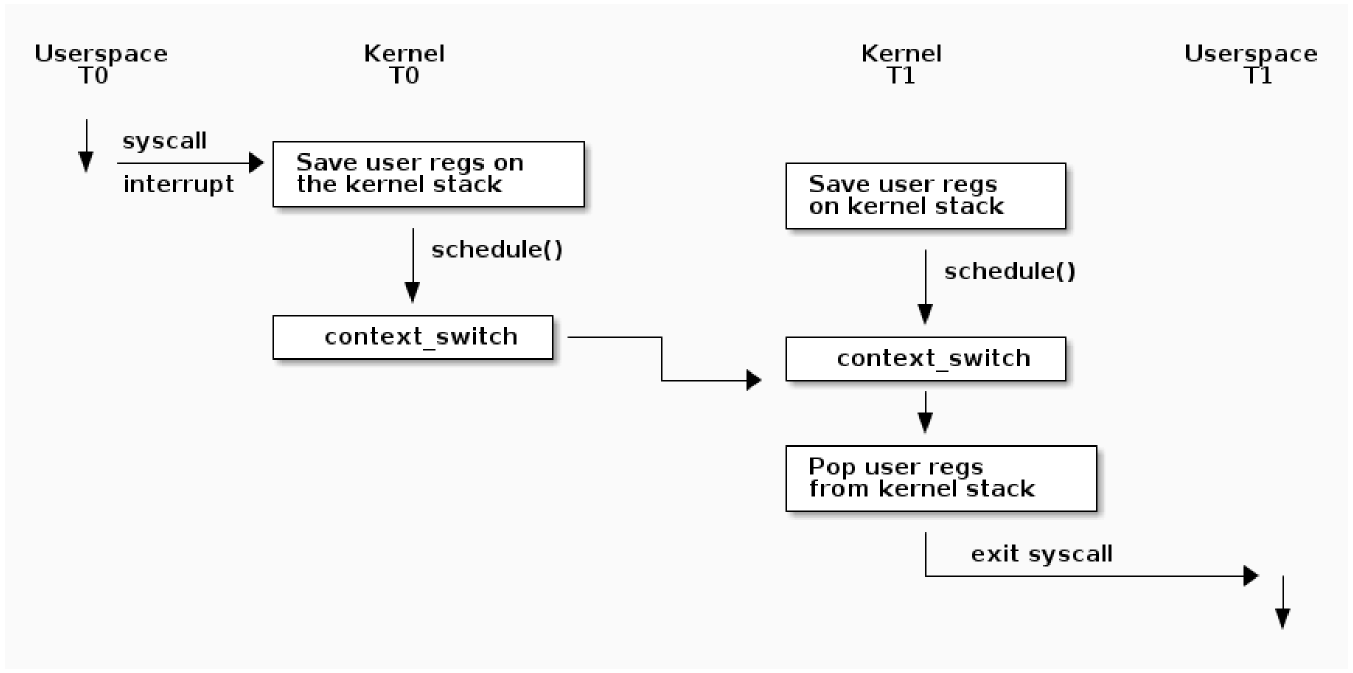
Quiz: previous implementation for current (x86)

What is the size of `struct thread_info`?

Which of the following are potential valid sizes for `struct thread_info`: 4095, 4096, 4097?

Context switching

The following diagram shows an overview of the Linux kernel context switch process:



Note that before a context switch can occur we must do a kernel transition, either with a system call or with an interrupt. At that point the user space registers are saved on the kernel stack. At some point the `schedule()` function will be called which can decide that a context switch must occur from T0 to T1 (e.g. because the current thread is blocking waiting for an I/O operation to complete or because it's allocated time slice has expired).

At that point `context_switch()` will perform architecture specific operations and will switch the address space if needed:

```c
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
         struct task_struct *next, struct rq_flags *rf)
{
    prepare_task_switch(rq, prev, next);

    /*
     * For paravirt, this is coupled with an exit in switch_to to
     * combine the page table reload and the switch backend into
     * one hypercall.
     */
    arch_start_context_switch(prev);

    /*
     * kernel -> kernel   lazy + transfer active
     *   user -> kernel   lazy + mmgrab() active
     *
     * kernel ->   user   switch + mmdrop() active
     *   user ->   user   switch
     */
    if (!next->mm) {                                // to kernel
        enter_lazy_tlb(prev->active_mm, next);

        next->active_mm = prev->active_mm;
        if (prev->mm)                           // from user
            mmgrab(prev->active_mm);
        else
            prev->active_mm = NULL;
    } else {                                        // to user
        membarrier_switch_mm(rq, prev->active_mm, next->mm);
        /*
         * sys_membarrier() requires an smp_mb() between setting
         * rq->curr / membarrier_switch_mm() and returning to userspace.
         *
         * The below provides this either through switch_mm(), or in
         * case 'prev->active_mm == next->mm' through
         * finish_task_switch()'s mmdrop().
         */
        switch_mm_irqs_off(prev->active_mm, next->mm, next);

        if (!prev->mm) {                        // from kernel
            /* will mmdrop() in finish_task_switch(). */
            rq->prev_mm = prev->active_mm;
            prev->active_mm = NULL;
        }
    }

    rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);

    prepare_lock_switch(rq, next, rf);

    /* Here we just switch the register state and the stack. */
    switch_to(prev, next, prev);
    barrier();

    return finish_task_switch(prev);
  }
```

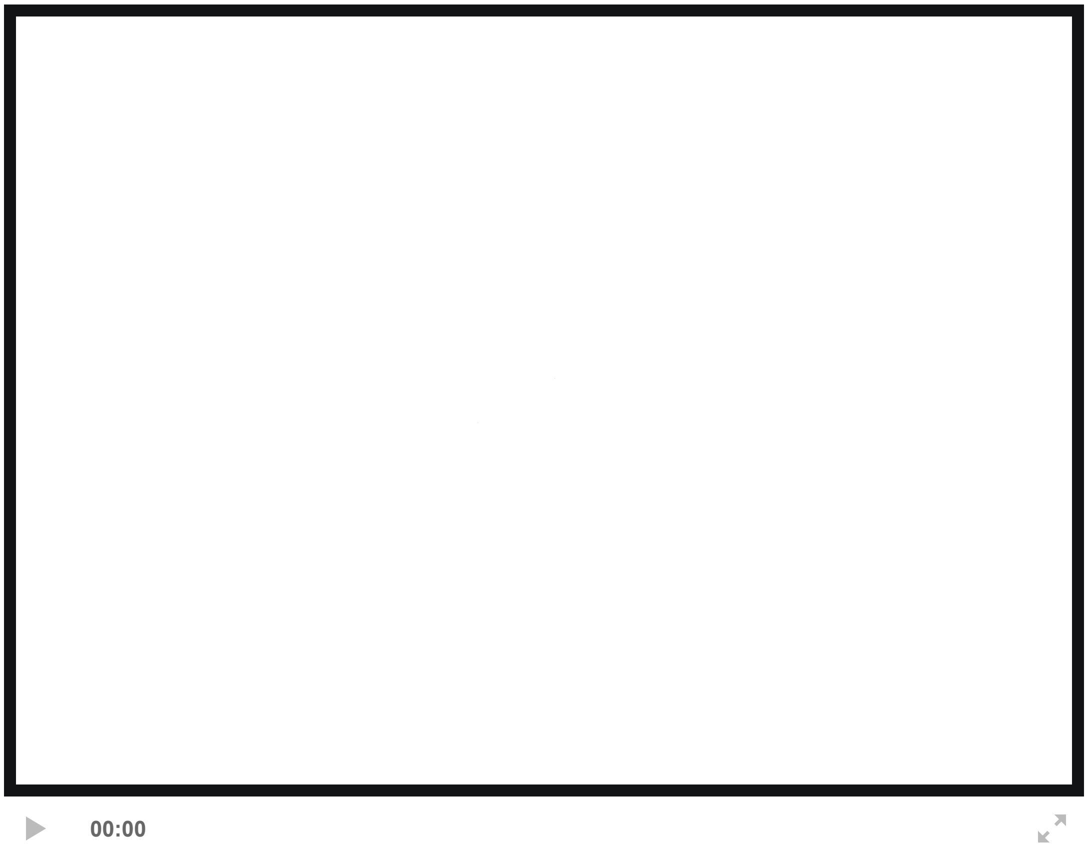
Then it will call the architecture specific `switch_to` implementation to switch the registers state and kernel stack. Note that registers are saved on stack and that the stack pointer is saved in the task structure:

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


/*
 * %eax: prev task
 * %edx: next task
 */
.pushsection .text, "ax"
SYM_CODE_START(__switch_to_asm)
    /*
     * Save callee-saved registers
     * This must match the order in struct inactive_task_frame
     */
    pushl   %ebp
    pushl   %ebx
    pushl   %edi
    pushl   %esi
    /*
     * Flags are saved to prevent AC leakage. This could go
     * away if objtool would have 32bit support to verify
     * the STAC/CLAC correctness.
     */
    pushfl

    /* switch stack */
    movl    %esp, TASK_threadsp(%eax)
    movl    TASK_threadsp(%edx), %esp

  #ifdef CONFIG_STACKPROTECTOR
    movl    TASK_stack_canary(%edx), %ebx
    movl    %ebx, PER_CPU_VAR(stack_canary)+stack_canary_offset
  #endif

  #ifdef CONFIG_RETPOLINE
    /*
     * When switching from a shallower to a deeper call stack
     * the RSB may either underflow or use entries populated
     * with userspace addresses. On CPUs where those concerns
     * exist, overwrite the RSB with entries which capture
     * speculative execution to prevent attack.
     */
    FILL_RETURN_BUFFER %ebx, RSB_CLEAR_LOOPS, X86_FEATURE_RSB_CTXSW
    #endif

    /* Restore flags or the incoming task to restore AC state. */
    popfl
    /* restore callee-saved registers */
    popl    %esi
    popl    %edi
    popl    %ebx
    popl    %ebp

    jmp     __switch_to
  SYM_CODE_END(__switch_to_asm)
  .popsection
```

You can notice that the instruction pointer is not explicitly saved. It is not needed because:

- a task will always resume in this function
- the `schedule()` (`context_switch()` is always inlined) caller's return address is saved on the kernel stack
- a jmp is used to execute `__switch_to()` which is a function and when it returns it will pop the original (next task) return address from the stack

The following screencast uses the debugger to setup a breaking in __switch_to_asm and examine the stack during the context switch:



Quiz: context switch

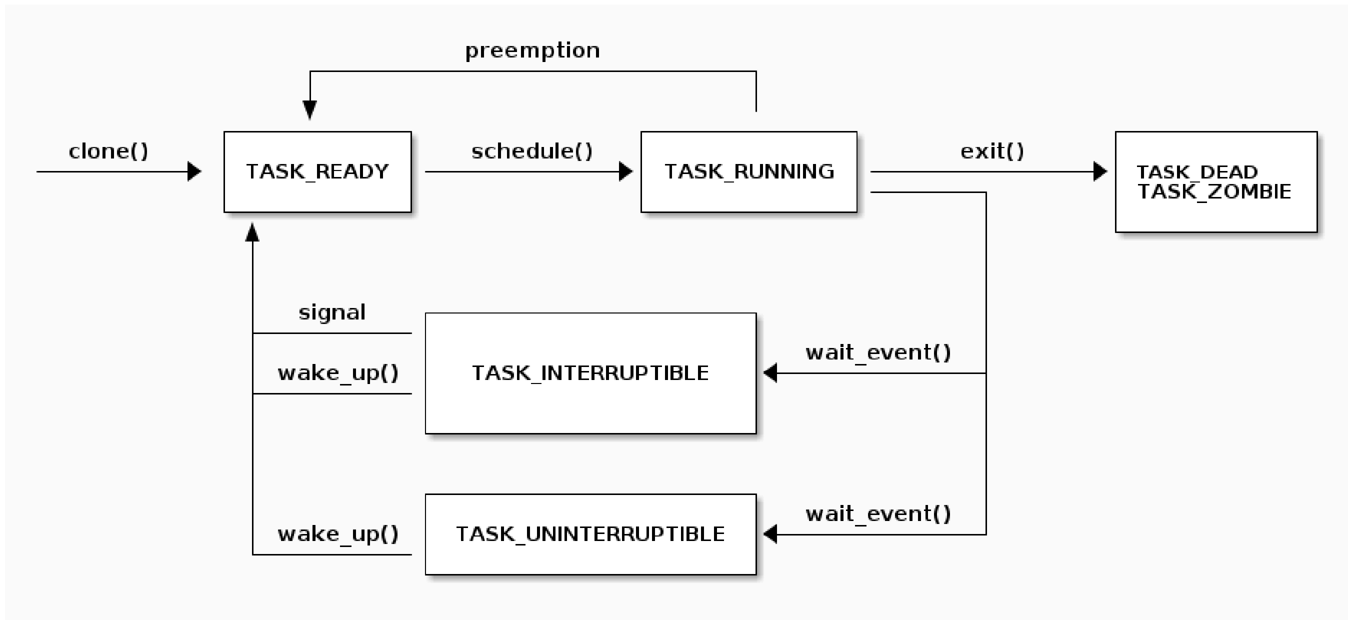
We are executing a context switch. Select all of the statements that are true.

- the ESP register is saved in the task structure
- the EIP register is saved in the task structure
- general registers are saved in the task structure
- the ESP register is saved on the stack
- the EIP register is saved on the stack
- general registers are saved on the stack

Blocking and waking up tasks

Task states

The following diagram shows to the task (threads) states and the possible transitions between them:



Blocking the current thread

Blocking the current thread is an important operation we need to perform to implement efficient task scheduling - we want to run other threads while I/O operations complete.

In order to accomplish this the following operations take place:

- Set the current thread state to TASK_UINTERRUPTIBLE or TASK_INTERRUPTIBLE
- Add the task to a waiting queue
- Call the scheduler which will pick up a new task from the READY queue
- Do the context switch to the new task

Below are some snippets for the `wait_event` implementation. Note that the waiting queue is a list with some extra information like a pointer to the task struct.

Also note that a lot of effort is put into making sure no deadlock can occur between `wait_event` and `wake_up`: the task is added to the list before checking `condition`, signals are checked before calling `schedule()`.

```
/**
 * wait_event - sleep until a condition gets true
 * @wq_head: the waitqueue to wait on
 * @condition: a C expression for the event to wait for
 *
 * The process is put to sleep (TASK_UNINTERRUPTIBLE) until the
 * @condition evaluates to true. The @condition is checked each time
 * the waitqueue @wq_head is woken up.
 *
 * wake_up() has to be called after changing any variable that could
 * change the result of the wait condition.
 */
#define wait_event(wq_head, condition)            \
do {                                              \
  might_sleep();                                  \
  if (condition)                                  \
          break;                                  \
  __wait_event(wq_head, condition);               \
} while (0)

#define __wait_event(wq_head, condition)                                        \
    (void)___wait_event(wq_head, condition, TASK_UNINTERRUPTIBLE, 0, 0,   \
                        schedule())

/*
 * The below macro ___wait_event() has an explicit shadow of the __ret
 * variable when used from the wait_event_*() macros.
 *
 * This is so that both can use the ___wait_cond_timeout() construct
 * to wrap the condition.
 *
 * The type inconsistency of the wait_event_*() __ret variable is also
 * on purpose; we use long where we can return timeout values and int
 * otherwise.
 */
#define ___wait_event(wq_head, condition, state, exclusive, ret, cmd)    \
({                                                                      \
    __label__ __out;                                                    \
    struct wait_queue_entry __wq_entry;                                 \
    long __ret = ret;       /* explicit shadow */                       \
                                                                        \
    init_wait_entry(&__wq_entry, exclusive ? WQ_FLAG_EXCLUSIVE : 0);    \
    for (;;) {                                                          \
        long __int = prepare_to_wait_event(&wq_head, &__wq_entry, state);\
                                                                        \
        if (condition)                                                  \
            break;                                                      \
                                                                        \
        if (___wait_is_interruptible(state) && __int) {                 \
            __ret = __int;                                              \
            goto __out;                                                 \
        }                                                               \
                                                                        \
        cmd;                                                            \
    }                                                                   \
    finish_wait(&wq_head, &__wq_entry);                                 \
   __out:  __ret;                                                       \
 })
```

```
void init_wait_entry(struct wait_queue_entry *wq_entry, int flags)
{
   wq_entry->flags = flags;
   wq_entry->private = current;
   wq_entry->func = autoremove_wake_function;
   INIT_LIST_HEAD(&wq_entry->entry);
}

long prepare_to_wait_event(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry,
int state)
{
    unsigned long flags;
    long ret = 0;

    spin_lock_irqsave(&wq_head->lock, flags);
    if (signal_pending_state(state, current)) {
        /*
         * Exclusive waiter must not fail if it was selected by wakeup,
         * it should "consume" the condition we were waiting for.
         *
         * The caller will recheck the condition and return success if
         * we were already woken up, we can not miss the event because
         * wakeup locks/unlocks the same wq_head->lock.
         *
         * But we need to ensure that set-condition + wakeup after that
         * can't see us, it should wake up another exclusive waiter if
         * we fail.
         */
        list_del_init(&wq_entry->entry);
        ret = -ERESTARTSYS;
    } else {
        if (list_empty(&wq_entry->entry)) {
            if (wq_entry->flags & WQ_FLAG_EXCLUSIVE)
                __add_wait_queue_entry_tail(wq_head, wq_entry);
            else
                __add_wait_queue(wq_head, wq_entry);
        }
        set_current_state(state);
    }
    spin_unlock_irqrestore(&wq_head->lock, flags);

    return ret;
}

static inline void __add_wait_queue(struct wait_queue_head *wq_head, struct wait_queue_entry
*wq_entry)
{
    list_add(&wq_entry->entry, &wq_head->head);
}

static inline void __add_wait_queue_entry_tail(struct wait_queue_head *wq_head, struct
wait_queue_entry *wq_entry)
{
    list_add_tail(&wq_entry->entry, &wq_head->head);
}

/**
 * finish_wait - clean up after waiting in a queue
 * @wq_head: waitqueue waited on
 * @wq_entry: wait descriptor
 *
 * Sets current thread back to running state and removes
 * the wait descriptor from the given waitqueue if still
 * queued.
 */
void finish_wait(struct wait_queue_head *wq_head, struct wait_queue_entry *wq_entry)
{
    unsigned long flags;

    __set_current_state(TASK_RUNNING);
    /*
     * We can check for list emptiness outside the lock
     * IFF:
     *  - we use the "careful" check that verifies both
     *    the next and prev pointers, so that there cannot
     *    be any half-pending updates in progress on other
     *    CPU's that we haven't seen yet (and that might
     *    still change the stack area.
     * and
     *  - all other users take the lock (ie we can only
     *    have _one_ other CPU that looks at or modifies
     *    the list).
     */
    if (!list_empty_careful(&wq_entry->entry)) {
        spin_lock_irqsave(&wq_head->lock, flags);
        list_del_init(&wq_entry->entry);
        spin_unlock_irqrestore(&wq_head->lock, flags);
    }
}
```

Waking up a task

We can wake-up tasks by using the `wake_up` primitive. The following high level operations are performed to wake up a task:

- Select a task from the waiting queue
- Set the task state to TASK_READY
- Insert the task into the scheduler's READY queue
- On SMP system this is a complex operation: each processor has its own queue, queues need to be balanced, CPUs needs to be signaled

```
#define wake_up(x)                        __wake_up(x, TASK_NORMAL, 1, NULL)

/**
 * __wake_up - wake up threads blocked on a waitqueue.
 * @wq_head: the waitqueue
 * @mode: which threads
 * @nr_exclusive: how many wake-one or wake-many threads to wake up
 * @key: is directly passed to the wakeup function
 *
 * If this function wakes up a task, it executes a full memory barrier before
 * accessing the task state.
 */
void __wake_up(struct wait_queue_head *wq_head, unsigned int mode,
               int nr_exclusive, void *key)
{
    __wake_up_common_lock(wq_head, mode, nr_exclusive, 0, key);
}

static void __wake_up_common_lock(struct wait_queue_head *wq_head, unsigned int mode,
                  int nr_exclusive, int wake_flags, void *key)
{
  unsigned long flags;
  wait_queue_entry_t bookmark;

  bookmark.flags = 0;
  bookmark.private = NULL;
  bookmark.func = NULL;
  INIT_LIST_HEAD(&bookmark.entry);

  do {
          spin_lock_irqsave(&wq_head->lock, flags);
          nr_exclusive = __wake_up_common(wq_head, mode, nr_exclusive,
                                          wake_flags, key, &bookmark);
          spin_unlock_irqrestore(&wq_head->lock, flags);
  } while (bookmark.flags & WQ_FLAG_BOOKMARK);
}

/*
 * The core wakeup function. Non-exclusive wakeups (nr_exclusive == 0) just
 * wake everything up. If it's an exclusive wakeup (nr_exclusive == small +ve
 * number) then we wake all the non-exclusive tasks and one exclusive task.
 *
 * There are circumstances in which we can try to wake a task which has already
 * started to run but is not in state TASK_RUNNING. try_to_wake_up() returns
 * zero in this (rare) case, and we handle it by continuing to scan the queue.
 */
static int __wake_up_common(struct wait_queue_head *wq_head, unsigned int mode,
                            int nr_exclusive, int wake_flags, void *key,
                  wait_queue_entry_t *bookmark)
{
    wait_queue_entry_t *curr, *next;
    int cnt = 0;

    lockdep_assert_held(&wq_head->lock);

    if (bookmark && (bookmark->flags & WQ_FLAG_BOOKMARK)) {
          curr = list_next_entry(bookmark, entry);

          list_del(&bookmark->entry);
```

```
            bookmark->flags = 0;
    } else
            curr = list_first_entry(&wq_head->head, wait_queue_entry_t, entry);

    if (&curr->entry == &wq_head->head)
            return nr_exclusive;

    list_for_each_entry_safe_from(curr, next, &wq_head->head, entry) {
            unsigned flags = curr->flags;
            int ret;

            if (flags & WQ_FLAG_BOOKMARK)
                    continue;

            ret = curr->func(curr, mode, wake_flags, key);
            if (ret < 0)
                    break;
            if (ret && (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
                    break;

            if (bookmark && (++cnt > WAITQUEUE_WALK_BREAK_CNT) &&
                            (&next->entry != &wq_head->head)) {
                    bookmark->flags = WQ_FLAG_BOOKMARK;
                    list_add_tail(&bookmark->entry, &next->entry);
                    break;
            }
    }

    return nr_exclusive;
}

int autoremove_wake_function(struct wait_queue_entry *wq_entry, unsigned mode, int sync, void
*key)
{
    int ret = default_wake_function(wq_entry, mode, sync, key);

    if (ret)
        list_del_init_careful(&wq_entry->entry);

    return ret;
}

int default_wake_function(wait_queue_entry_t *curr, unsigned mode, int wake_flags,
                    void *key)
{
    WARN_ON_ONCE(IS_ENABLED(CONFIG_SCHED_DEBUG) && wake_flags & ~WF_SYNC);
    return try_to_wake_up(curr->private, mode, wake_flags);
}

/**
 * try_to_wake_up - wake up a thread
 * @p: the thread to be awakened
 * @state: the mask of task states that can be woken
 * @wake_flags: wake modifier flags (WF_*)
 *
 * Conceptually does:
 *
 *   If (@state & @p->state) @p->state = TASK_RUNNING.
 *
 * If the task was not queued/runnable, also place it back on a runqueue.
 *
 * This function is atomic against schedule() which would dequeue the task.
 *
 * It issues a full memory barrier before accessing @p->state, see the comment
 * with set_current_state().
 *
 * Uses p->pi_lock to serialize against concurrent wake-ups.
 *
 * Relies on p->pi_lock stabilizing:
 *  - p->sched_class
 *  - p->cpus_ptr
 *  - p->sched_task_group
 * in order to do migration, see its use of select_task_rq()/set_task_cpu().
 *
 * Tries really hard to only take one task_rq(p)->lock for performance.
 * Takes rq->lock in:
 *  - ttwu_runnable()    -- old rq, unavoidable, see comment there;
 *  - ttwu_queue()       -- new rq, for enqueue of the task;
 *  - psi_ttwu_dequeue() -- much sadness :-( accounting will kill us.
 *
 * As a consequence we race really badly with just about everything. See the
 * many memory barriers and their comments for details.
 *
 * Return: %true if @p->state changes (an actual wakeup was done),
 *           %false otherwise.
 */
static int
try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
{
    ...
```

Preempting tasks

Up until this point we look at how context switches occurs voluntary between threads. Next we will look at how preemption is handled. We will start wight the simpler case where the kernel is configured as non preemptive and then we will move to the preemptive kernel case.

Non preemptive kernel

- At every tick the kernel checks to see if the current process has its time slice consumed
- If that happens a flag is set in interrupt context
- Before returning to userspace the kernel checks this flag and calls `schedule()` if needed
- In this case tasks are not preempted while running in kernel mode (e.g. system call) so there are no synchronization issues

Preemptive kernel

In this case the current task can be preempted even if we are running in kernel mode and executing a system call. This requires using a special synchronization primitives: `preempt_disable` and `preempt_enable`.

In order to simplify handling for preemptive kernels and since synchronization primitives are needed for the SMP case anyway, preemption is disabled automatically when a spinlock is used.

As before, if we run into a condition that requires the preemption of the current task (its time slices has expired) a flag is set. This flag is checked whenever the preemption is reactivated, e.g. when exiting a critical section through a `spin_unlock()` and if needed the scheduler is called to select a new task.

Process context

Now that we have examined the implementation of processes and threads (tasks), how context switching occurs, how we can block, wake-up and preempt tasks, we can finally define what the process context is what are its properties:

The kernel is executing in process context when it is running a system call.

In process context there is a well defined context and we can access the current process data with `current`

In process context we can sleep (wait on a condition).

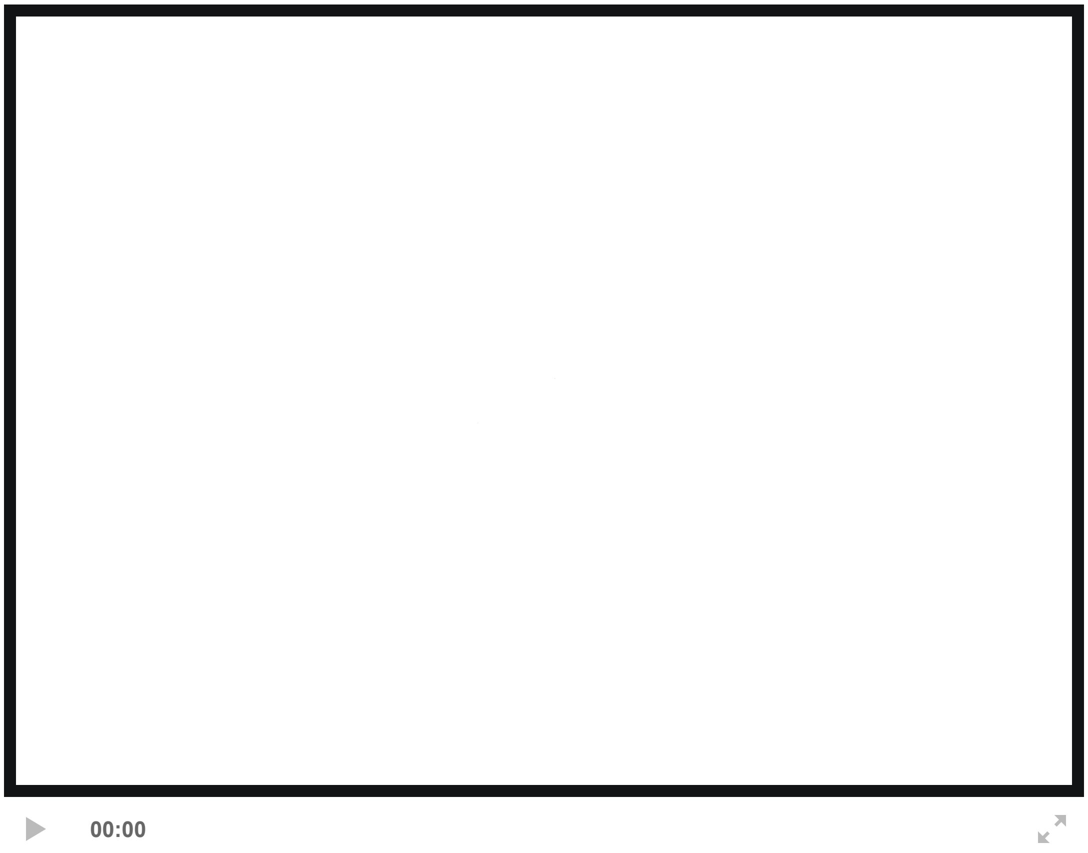
In process context we can access the user-space (unless we are running in a kernel thread context).

Kernel threads

Sometimes the kernel core or device drivers need to perform blocking operations and thus they need to run in process context.

Kernel threads are used exactly for this and are a special class of tasks that don't "userspace" resources (e.g. no address space or opened files).

The following screencast takes a closer look at kernel threads:



Using gdb scripts for kernel inspection

The Linux kernel comes with a predefined set of gdb extra commands we can use to inspect the kernel during debugging. They will automatically be loaded as long gdbinit is properly setup

```
ubuntu@so2:/linux/tools/labs$ cat ~/.gdbinit
add-auto-load-safe-path /linux/scripts/gdb/vmlinux-gdb.py
```

All of the kernel specific commands are prefixed with lx-. You can use TAB in gdb to list all of them:

```
(gdb) lx-
lx-clk-summary        lx-dmesg              lx-mounts
lx-cmdline            lx-fdtdump            lx-ps
lx-configdump         lx-genpd-summary      lx-symbols
lx-cpus               lx-iomem              lx-timerlist
lx-device-list-bus    lx-ioports            lx-version
lx-device-list-class  lx-list-check
lx-device-list-tree   lx-lsmod
```

The implementation of the commands can be found at *script/gdb/linux*. Lets take a closer look at the lx-ps implementation:

```
task_type = utils.CachedType("struct task_struct")


def task_lists():
 task_ptr_type = task_type.get_type().pointer()
 init_task = gdb.parse_and_eval("init_task").address
 t = g = init_task

 while True:
     while True:
         yield t

         t = utils.container_of(t['thread_group']['next'],
                                task_ptr_type, "thread_group")
         if t == g:
             break

     t = g = utils.container_of(g['tasks']['next'],
                                task_ptr_type, "tasks")
     if t == init_task:
         return


 class LxPs(gdb.Command):
 """Dump Linux tasks."""

 def __init__(self):
     super(LxPs, self).__init__("lx-ps", gdb.COMMAND_DATA)

 def invoke(self, arg, from_tty):
     gdb.write("{:>10} {:>12} {:>7}\n".format("TASK", "PID", "COMM"))
     for task in task_lists():
         gdb.write("{} {:^5} {}\n".format(
             task.format_string().split()[0],
             task["pid"].format_string(),
             task["comm"].string()))
```

Quiz: Kernel gdb scripts

What is the following change of the lx-ps script trying to accomplish?

```
diff --git a/scripts/gdb/linux/tasks.py b/scripts/gdb/linux/tasks.py
index 17ec19e9b5bf..7e43c163832f 100644
--- a/scripts/gdb/linux/tasks.py
+++ b/scripts/gdb/linux/tasks.py
@@ -75,10 +75,13 @@ class LxPs(gdb.Command):
     def invoke(self, arg, from_tty):
         gdb.write("{:>10} {:>12} {:>7}\n".format("TASK", "PID", "COMM"))
         for task in task_lists():
-            gdb.write("{} {:^5} {}\n".format(
+            check = task["mm"].format_string() == "0x0"
+            gdb.write("{} {:^5} {}{}{}\n".format(
                 task.format_string().split()[0],
                 task["pid"].format_string(),
-                task["comm"].string()))
+                "[" if check else "",
+                task["comm"].string(),
+                "]" if check else ""))


 LxPs()
```