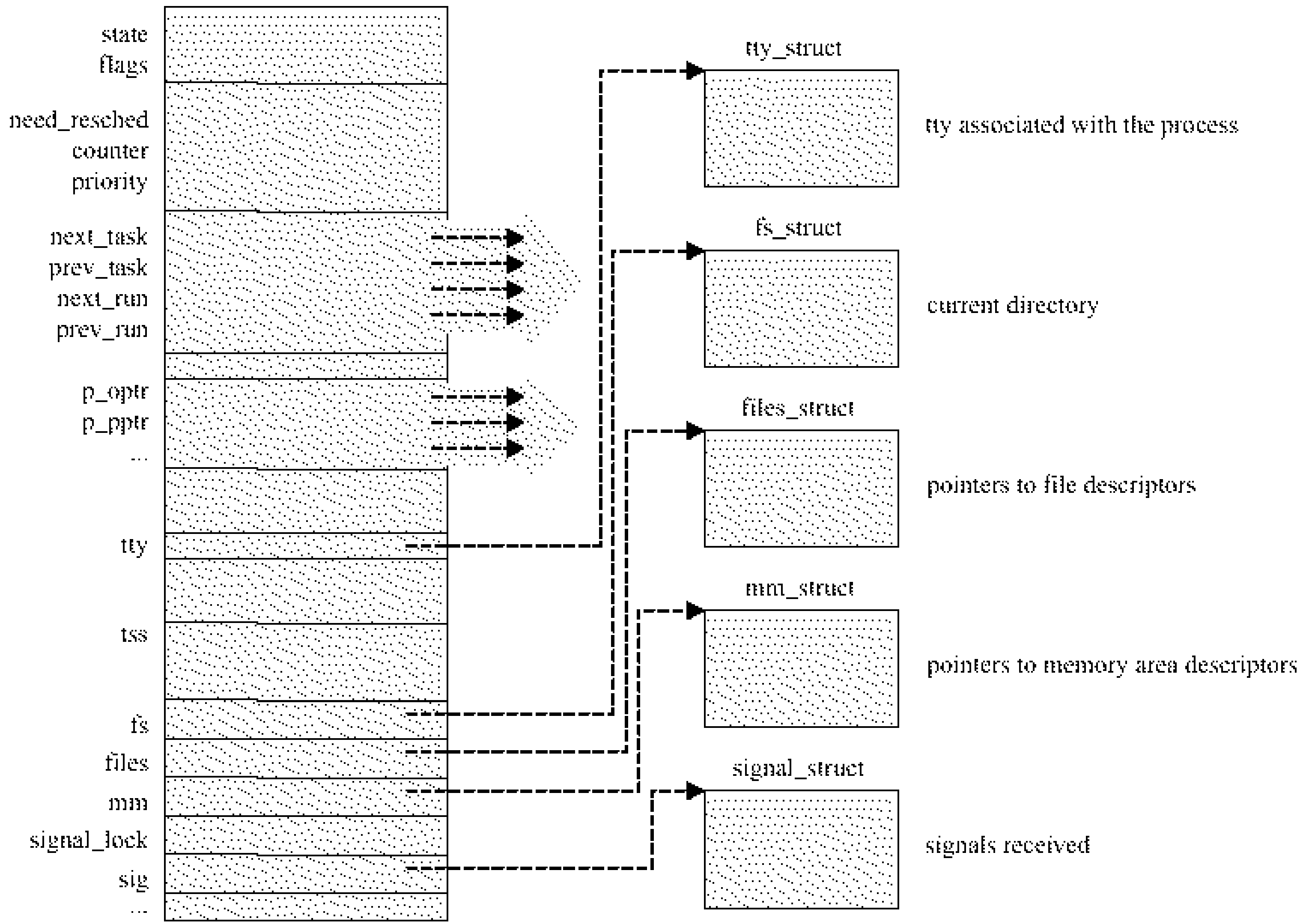
Ausgewählte Betriebssysteme

Processes and Threads

What is a process

- Fundamental concept for multiprogramming
- Instance of program in execution
  - Sequential control flow
- Entity to which system resources are allocated
  - Might be shared among processes (threads)

task_struct



state
flags
need_resched
counter
priority
next_task
prev_task
next_run
prev_run
p_optr
p_pptr
...
tty
tss
fs
files
mm
signal_lock
sig
...
tty_struct
tty associated with the process
fs_struct
current directory
files_struct
pointers to file descriptors
mm_struct
pointers to memory area descriptors
signal_struct
signals received

Process state

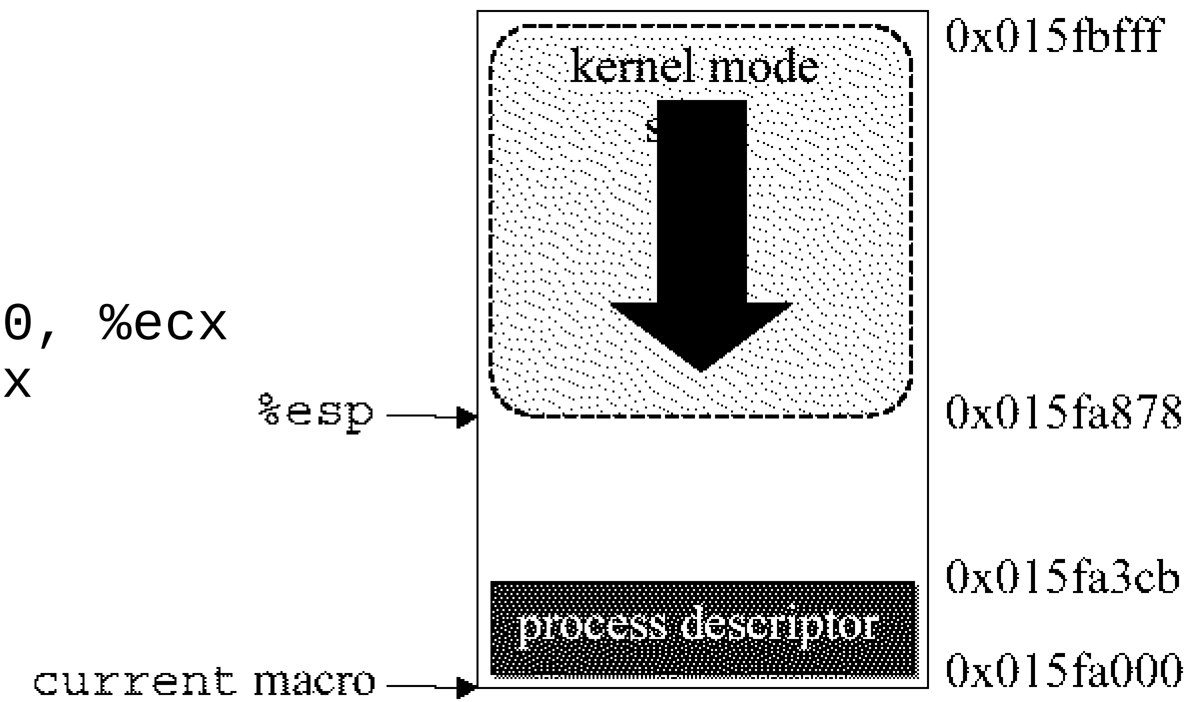
- Field in task_struct
- Currently available
  - TASK_RUNNING - executing or ready for execution
  - TASK_INTERRUPTIBLE - suspended
  - TASK_UNINTERRUPTIBLE - suspend, no signals
  - TASK_STOPPED - execution has been externally stopped
  - TASK_ZOMBIE - terminated

Process descriptor handling

- Processes are dynamic entities
  - Dynamic allocation
  - Half of all physical memory might be used for PCB
    - `max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;`
    - `/proc/sys/kernel/threads-max`
- Two different data structures per process
  - Process descriptor
  - Kernel stack

task_union

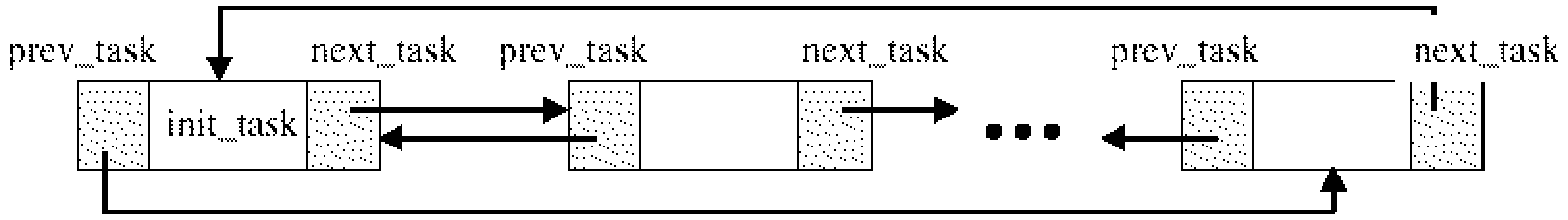
```
movl $0xffffe000, %ecx
andl $%esp, %ecx
movl %ecx, p
```




```
union task_union {
   struct task_struct task;
   unsigned long stack
[2048];
};
```

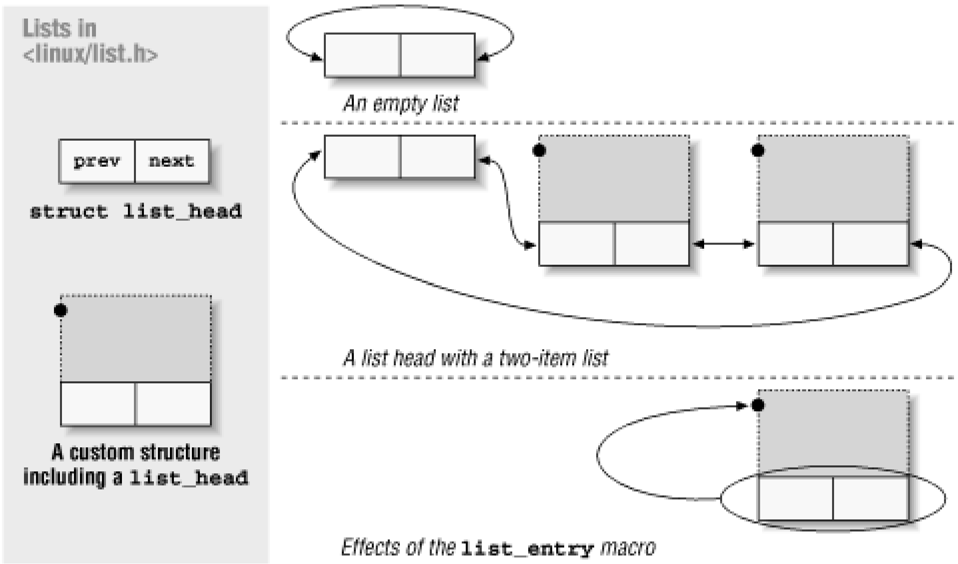
Process List

- Linux keeps list of processes for different purposses
  - Special properties (e.g. runnable)
- Process List
  - All processes in the system
  - Circular double linked list
  - SET_LINKS/REMOVE_LINKS macros ensure consistency
  - next_task, prev_task field in task_struct



Doubly linked lists (implementation)

- Often used
- Reusable implementation
  - Access functions and macros



Run queue

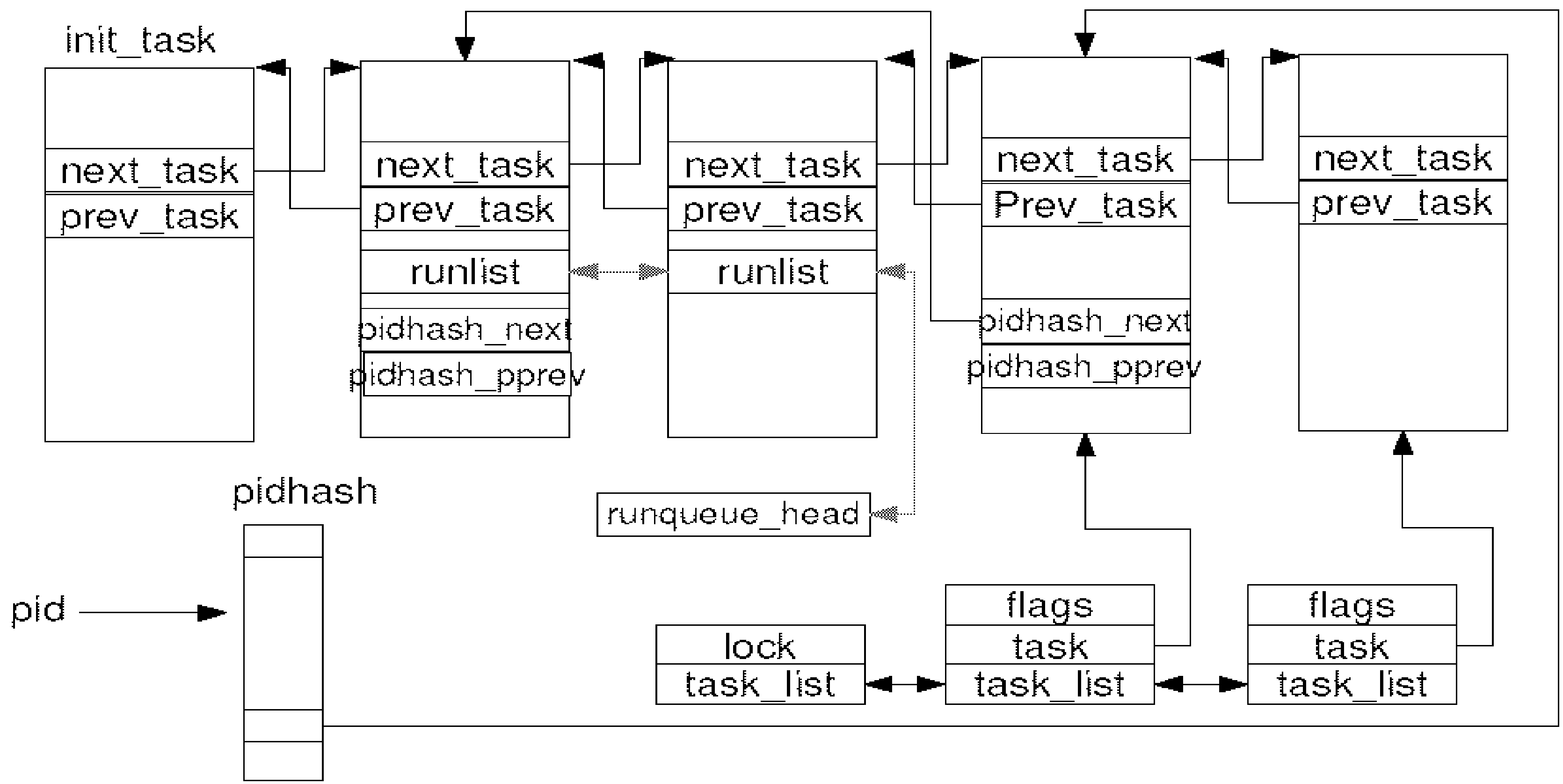
- Scheduling needs only to consider runnable processes
- Linked through `struct list_head run_list`
- Select most viable process to run next

```
|schedule
   |do_softirq // manages post-IRQ work
   |for each task
      |calculate counter
   |prepare_to__switch // does anything
   |switch_mm // change Memory context (change CR3 value)
   |switch_to (assembler)
      |SAVE ESP
      |RESTORE future_ESP
      |SAVE EIP
      |push future_EIP *** push parameter as we did a call
         |jmp __switch_to (it does some TSS work)
         |__switch_to()
          ..
         |ret *** ret from call using future_EIP in place of call address
      new_task
```

Process identification

- Address of PCB is unique in kernel address space
- PID used at user level
- Process list traversal to slow
- Hash table for fast lookup

PCB linking



init_task
next_task
prev_task
next_task
prev_task
runlist
pidhash_next
pidhash_pprev
next_task
prev_task
runlist
next_task
Prev_task
pidhash_next
pidhash_pprev
next_task
prev_task
pidhash
runqueue_head
pid
flags
task
task_list
flags
task
task_list
lock
task_list

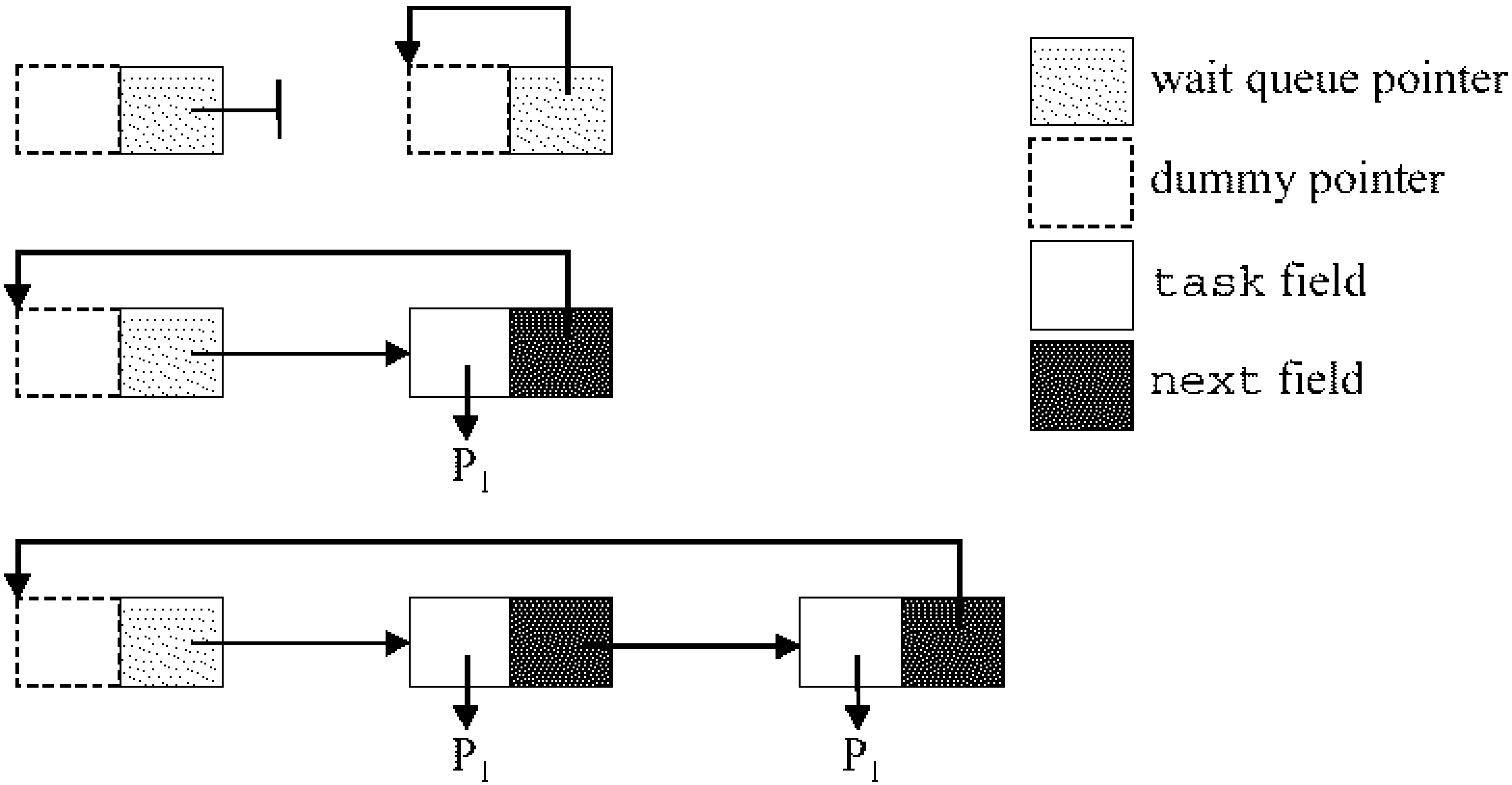
Process management

- Process queue
- TASK_RUNNABLE
  - Run queue
- TASK_STOPPED, TASK_ZOMBIE
  - Not grouped
- TASK_(UN)INTERRUPTIBLE
  - Subdivided into many classes, each of which correspondends to a specific event
  - State alone does not provide enough information
  - Specific lists of processes called *wait queues*

Wait Queues

- Define a new wait queue if needed
  - `DECLARE_WAIT_QUEUE_HEAD(...)`
- Functions
  - `add_wait_queue(..), remove_wait_queue(...)`
  - `sleep_on`
  - `wake_up`

Wait queue structure



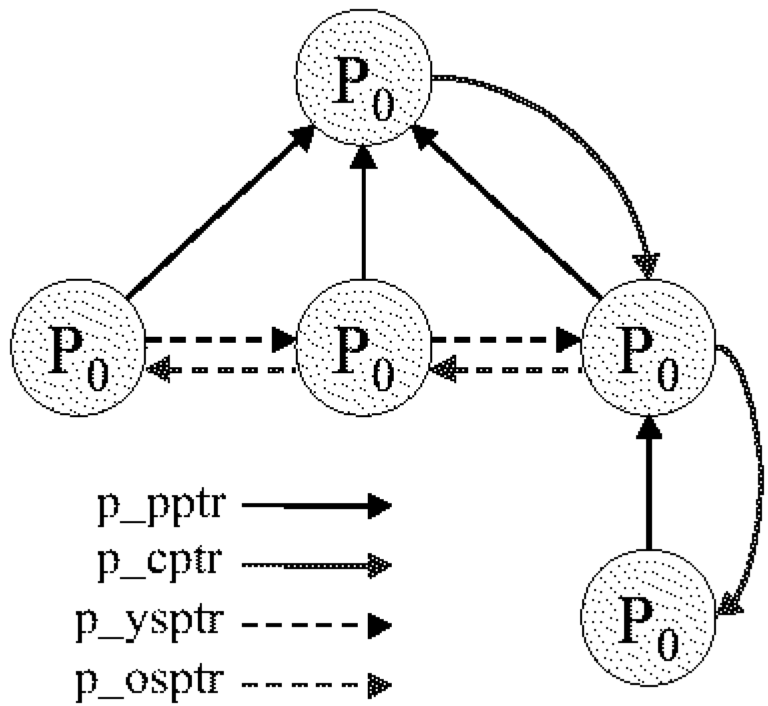
Process creation

- `fork` syscall
  - Copy process
  - Idependent new execution context
- `clone` syscall
  - Share resources with the new context
  - lightweight

Forking

```
|sys_fork
   |do_fork
      |alloc_task_struct
         |__get_free_pages
       |p->state = TASK_UNINTERRUPTIBLE
       |copy_flags
       |p->pid = get_pid
       |copy_files
       |copy_fs
       |copy_sighand
       |copy_mm // should manage CopyOnWrite (I part)
          |allocate_mm
          |mm_init
             |pgd_alloc -> get_pgd_fast
                |get_pgd_slow
          |dup_mmap
             |copy_page_range
                |ptep_set_wrprotect
                   |clear_bit // set page to read-only
          |copy_segments // For LDT
       |copy_thread
          |childregs->eax = 0
          |p->thread.esp = childregs // child fork returns 0
          |p->thread.eip = ret_from_fork // child starts from fork exit
       |retval = p->pid // parent fork returns child pid
       |SET_LINKS // insertion of task into the list pointers
       |nr_threads++ // Global variable
       |wake_up_process(p) // Now we can wake up just created child
       |return retval
```

Process relationship



Kernel threads

- Critical tasks implemented as intermittently running processes
  - Flushing disk caches
  - Swapping out unused page frames
- Regular scheduling
  - No unbound kernel activities
- Special characteristics
  - Mostly only one single kernel function
  - No user mode part

Kernel thread creation

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
        long retval, d0;

        __asm__ __volatile__(
                "movl %%esp,%%esi\n\t"
                "int $0x80\n\t"         /* Linux/i386 system call */
                "cmpl %%esp,%%esi\n\t"  /* child or parent? */
                "je 1f\n\t"             /* parent - jump */
                /* Load the argument into eax, and push it.  That way, it does
                 * not matter whether the called function is compiled with
                 * -mregparm or not.  */
                "movl %4,%%eax\n\t"
                "pushl %%eax\n\t"
                "call *%5\n\t"          /* call fn */
                "movl %3,%0\n\t"        /* exit */
                "int $0x80\n"
                "1:\t"
                :"=&a" (retval), "=&S" (d0)
                :"0" (__NR_clone), "i" (__NR_exit),
                 "r" (arg), "r" (fn),
                 "b" (flags | CLONE_VM)
                : "memory");
        return retval;
}
```

Kernel threads in action

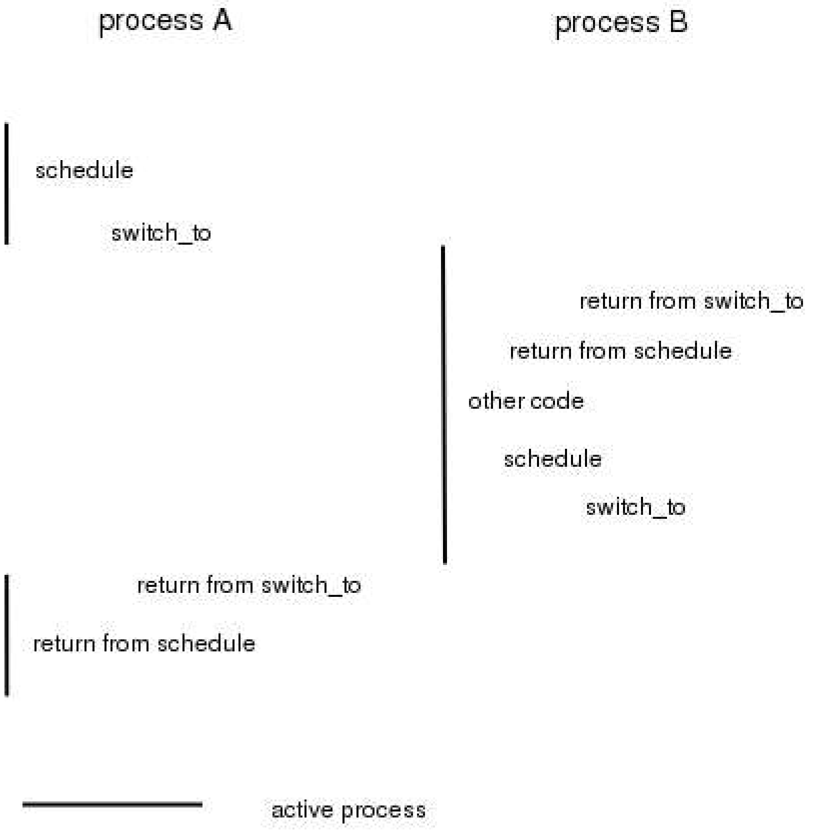
```
init,1
  |-(bdflush,6)
  |-(keventd,2)
  |-(khubd,53)
  |-(kjournald,10)
  |-(kjournald,89)
  |-(kjournald,90)
  |-(kjournald,1969)
  |-(ksoftirqd_CPU0,4)
  |-(kswapd,5)
  |-(kupdated,7)
  |-(lockd,19499)
```

Context switch

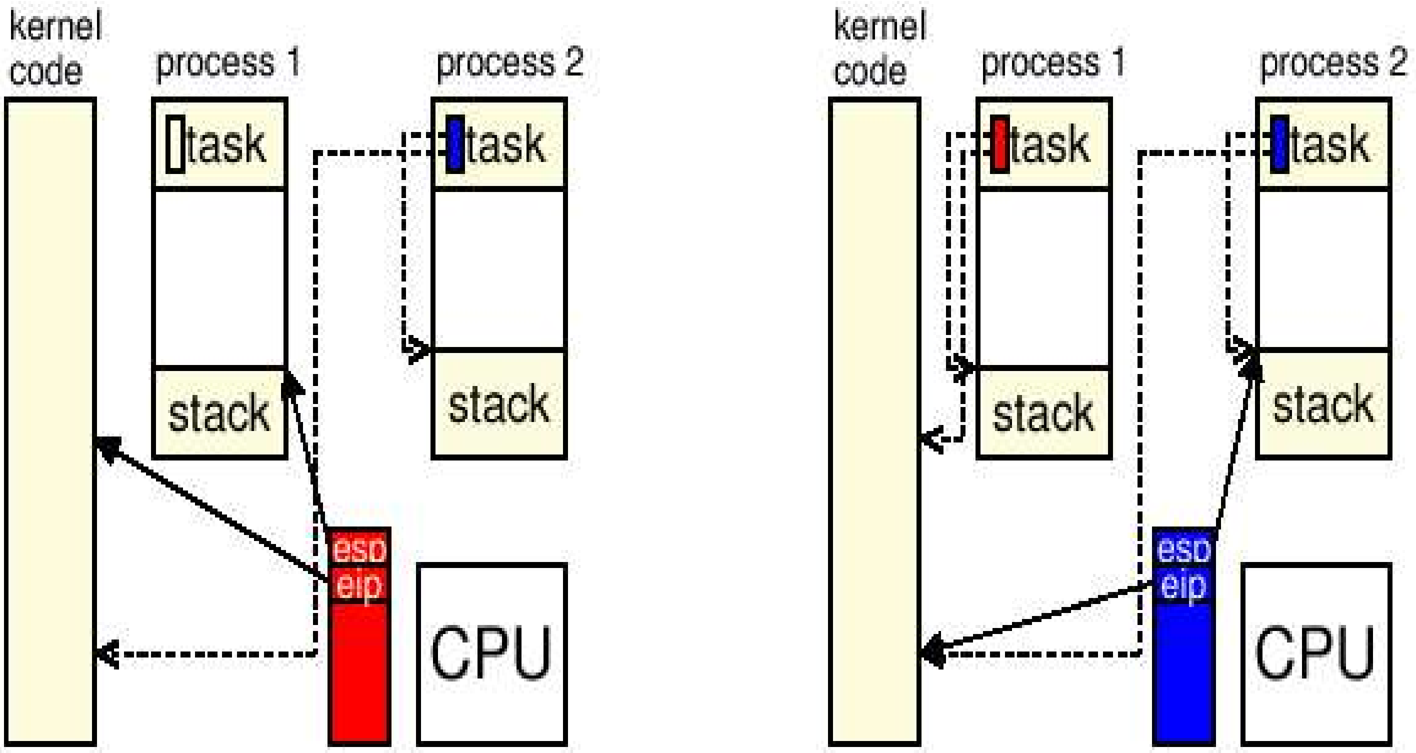
- Transfer control between contexts
  - Save state of current context
  - Load state of next context and resume execution
- Execution context
  - Architectural (user level) cpu state
  - Virtual memory

Context switch (2)

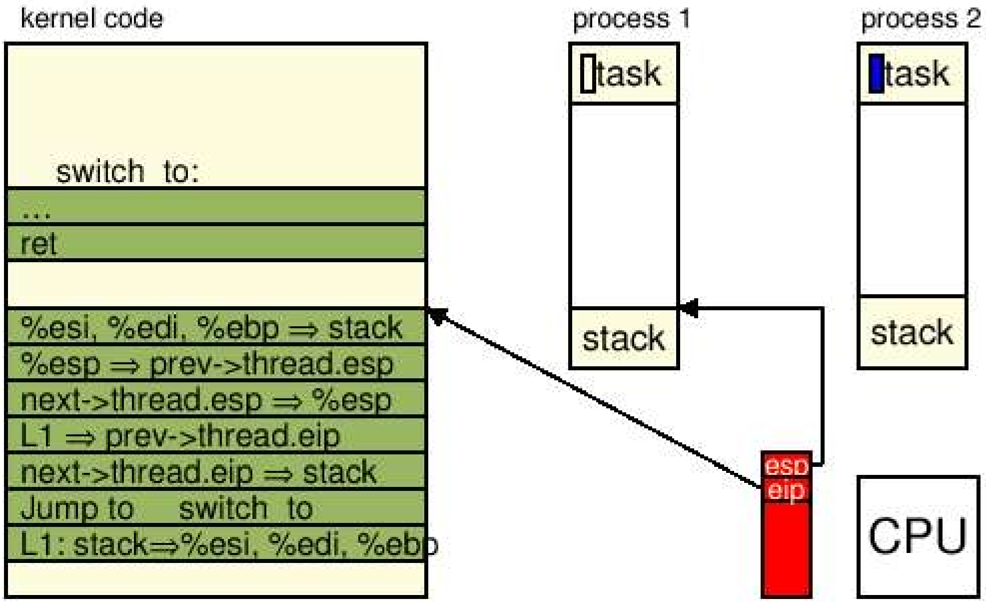
- For the kernel programmer context switching looks like a ordinary function call.
- Interleaved activities of other processes are transparent
-
```
void schedule(void){
   .
   /* calc next process */
   .
    switch_to(..., next, ...)
   .
   .
}
```



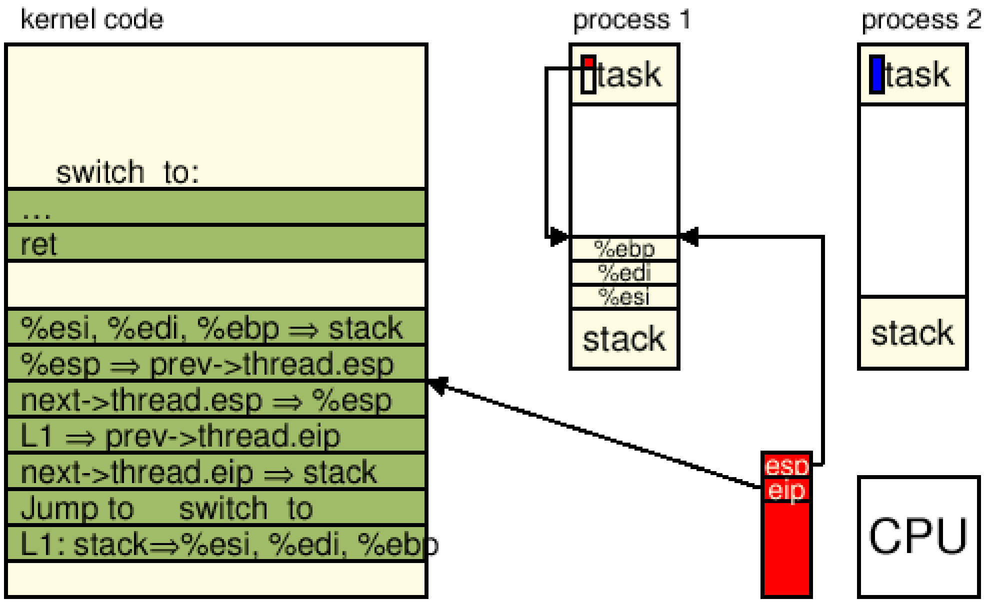
Switch 1


kernel code
process 1
process 2
task
task
stack
stack
esp
eip
CPU
kernel code
process 1
process 2
task
task
stack
stack
esp
eip
CPU

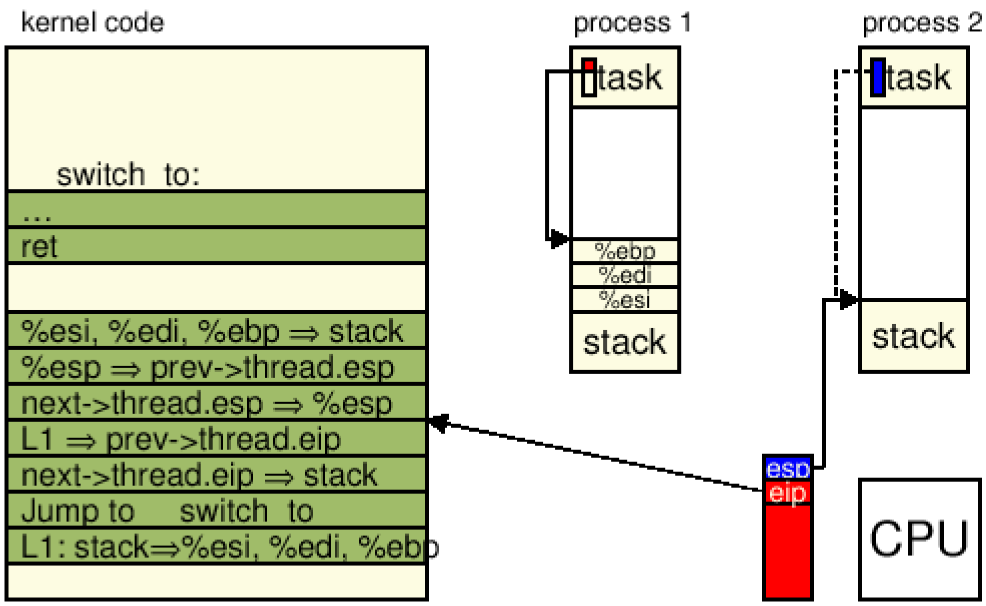
Switch (2)


kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to __switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
stack
process 2
task
stack
esp
eip
CPU

Switch (3)

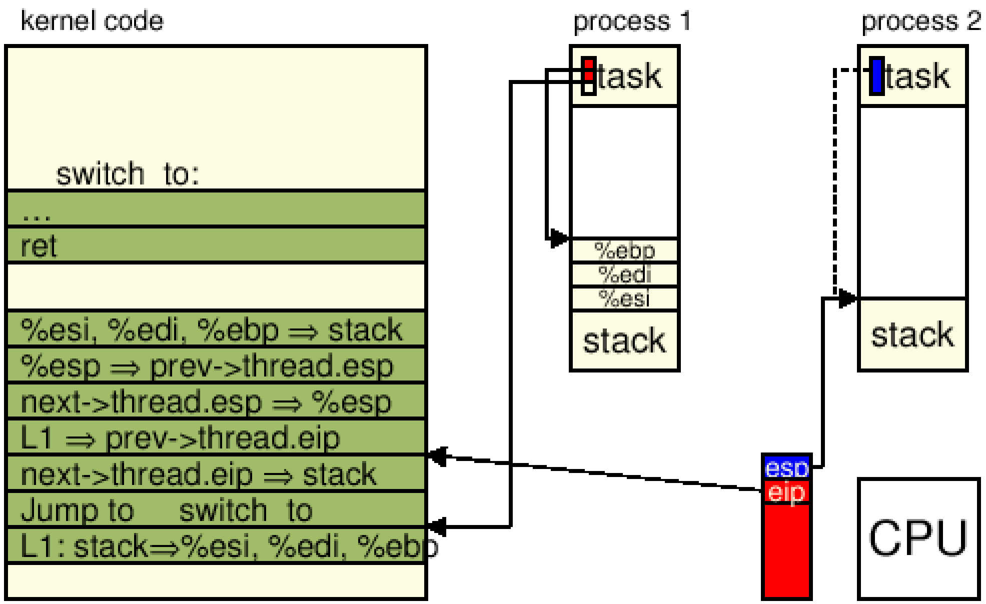

kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to __switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
stack
esp
eip
CPU

Switch (4)



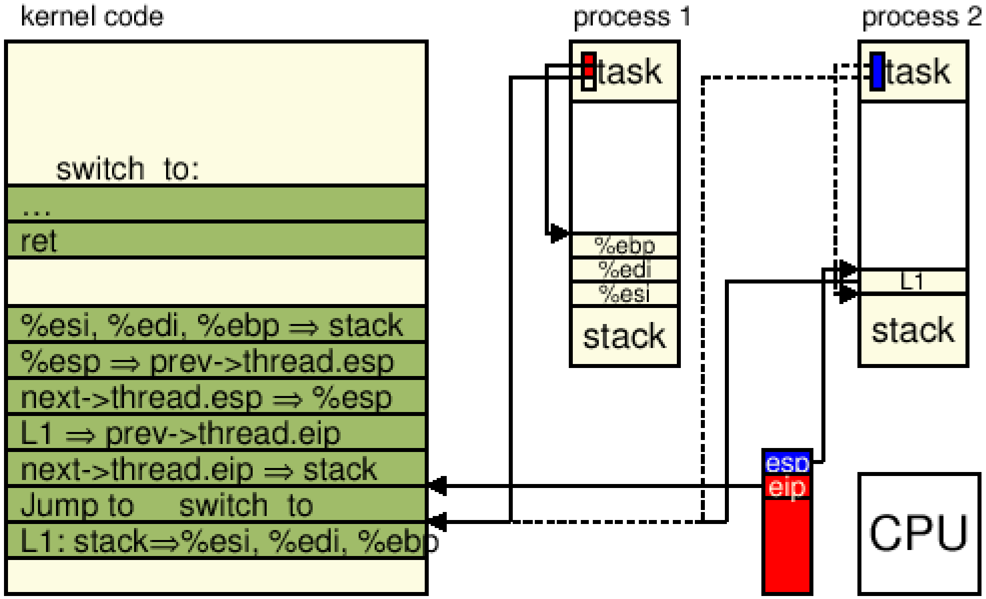
kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
stack
esp
eip
CPU

Switch(5)



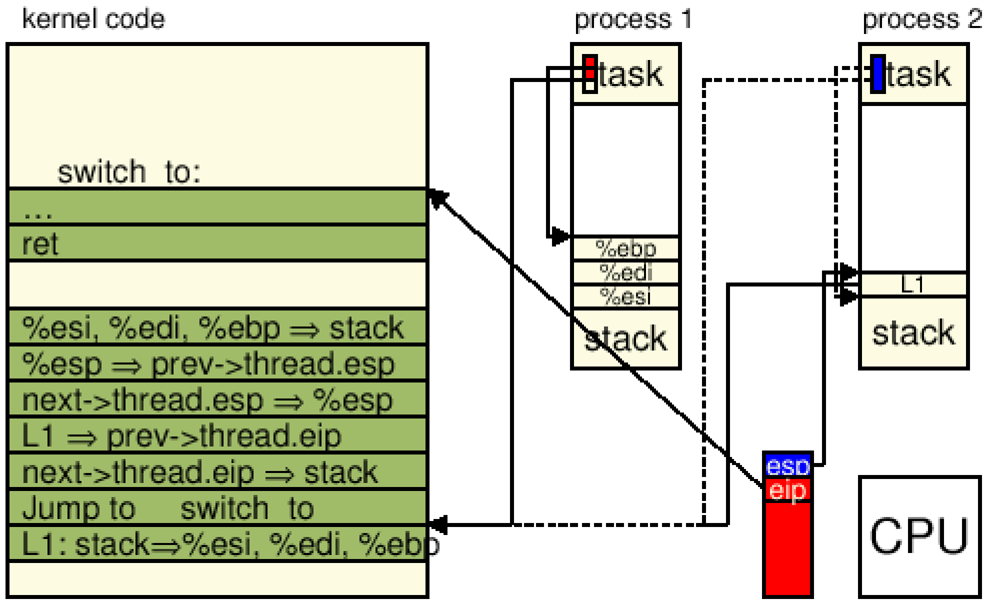
kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
stack
esp
eip
CPU

Switch (6)



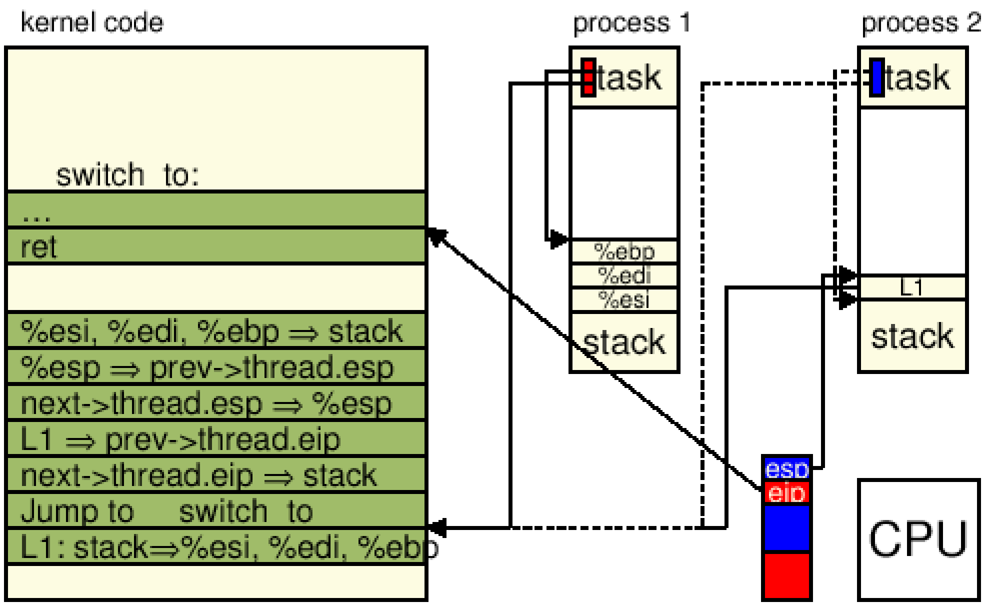
kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to __switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
L1
stack
esp
eip
CPU

Switch (7)

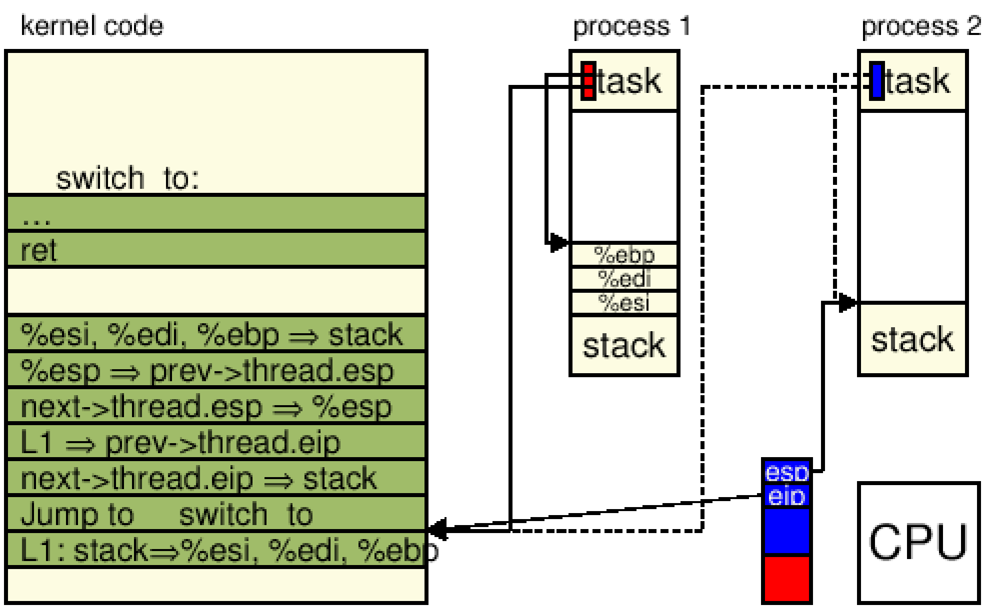


kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to __switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
L1
stack
esp
eip
CPU

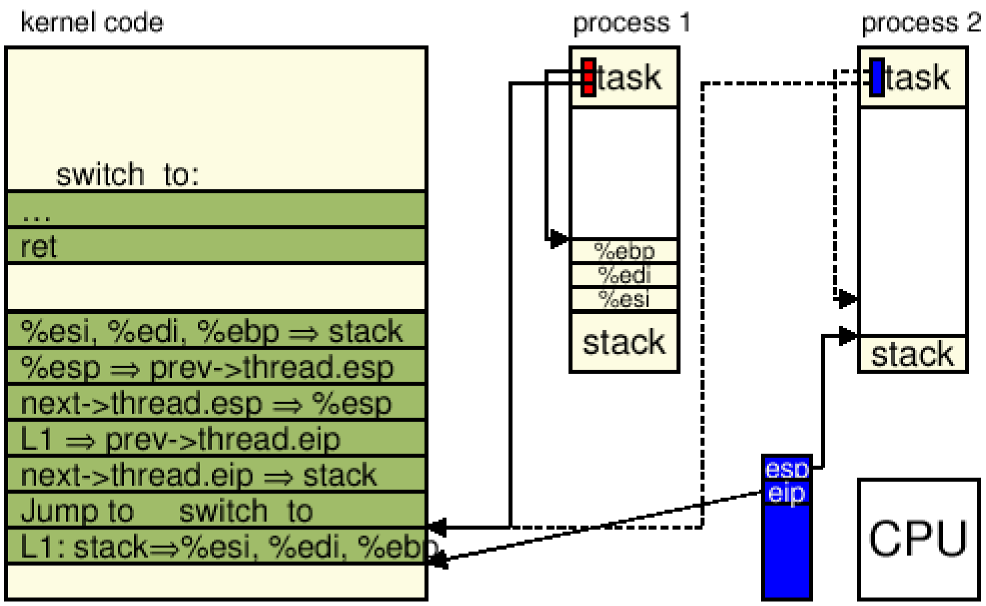
Switch (8)


kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to __switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
L1
stack
esp
eip
CPU

Switch (9)


kernel code
switch to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to switch to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
stack
esp
eip
CPU

Switch (10)



kernel code
switch_to:
...
ret
%esi, %edi, %ebp ⇒ stack
%esp ⇒ prev->thread.esp
next->thread.esp ⇒ %esp
L1 ⇒ prev->thread.eip
next->thread.eip ⇒ stack
Jump to switch_to
L1: stack⇒%esi, %edi, %ebp
process 1
task
%ebp
%edi
%esi
stack
process 2
task
stack
esp
eip
CPU

Process switch - Code

```
#define switch_to(prev,next,last) do {                \
    unsigned long esi,edi;                        \
    asm volatile("pushfl\n\t"                     \
             "pushl %%ebp\n\t"                    \
             "movl %%esp,%0\n\t"   /* save ESP */      \
             "movl %5,%%esp\n\t"   /* restore ESP */  \
             "movl $1f,%1\n\t"     /* save EIP */      \
             "pushl %6\n\t"    /* restore EIP */  \
             "jmp __switch_to\n"                  \
             "1:\t"                       \
             "popl %%ebp\n\t"                     \
             "popfl"                          \
             :"=m" (prev->thread.esp),"=m" (prev->thread.eip),
\
              "=a" (last),"=S" (esi),"=D" (edi)           \
             :"m" (next->thread.esp),"m" (next->thread.eip),  \
              "2" (prev), "d" (next));              \
} while (0)
```

Threads

- LinuxThreads is the standard POSIX thread library for Linux (1996)
- Based on principles of kernels of that time
  - Cheap kernel thread switches
  - Missing thread aware ABI
    - Thread local data with fixed relation to stack
  - Management thread necessary for creation etc.
  - No adequate kernel synchronization support
    - Signals abused
- Kernel is not aware of threads
  - Processes cooperate

LinuxThreads problems

- Signal handling is not POSIX compliant
- Extra management thread
- `ps` shows all threads in a process, procfs littered
- Core dumps do not contain the stack and machine registers for all threads
- `getpid()` returns different results for each thread
- Threads cannot wait for threads created by another thread
- Parent-child relationship instead of being peers
- Threads do not share user and group ids

Kernel support added

- TLS (thread local storage) support in the kernel
- `Clone` syscall extensions
  - Flag indicates that thread is created
- POSIX signal handling in the kernel
  - SIGSTOP forwarded to all threads of a process
- `exit` in two flavors for thread and process
- User level synchronization support
  - futex (**f**ast **u**ser mu**tex**)

Native POSIX thread library (NPTL)

- Better POSIX compliance
- Low startup/teardown costs
- Scalability
  - Enormous (100000) number of threads supported
- NUMA support
  - Node aware memory allocation
- Integration with C++