W4118 Operating Systems

Instructor: Junfeng Yang

Outline

- ❑ Linux process overview
- ❑ Linux process data structures
- ❑ Linux process operations

Finding process information

- ps
- top
- For each process, there is a corresponding directory /proc/<pid> to store this process information in the /proc pseudo file system

Process-related files

- Header files
  - include/linux/sched.h – declarations for most process data structures
  - include/linux/wait.h – declarations for wait queues
  - include/asm-i386/system.h – architecture-dependent declarations
- Source files
  - kernel/sched.c – process scheduling routines
  - kernel/signal.c – signal handling routines
  - kernel/fork.c – process/thread creation routines
  - kernel/exit.c – process exit routines
  - fs/exec.c – executing program
  - arch/i386/kernel/entry.S – kernel entry points
  - arch/i386/kernel/process.c – architecture-dependent process routines

Kernel address space

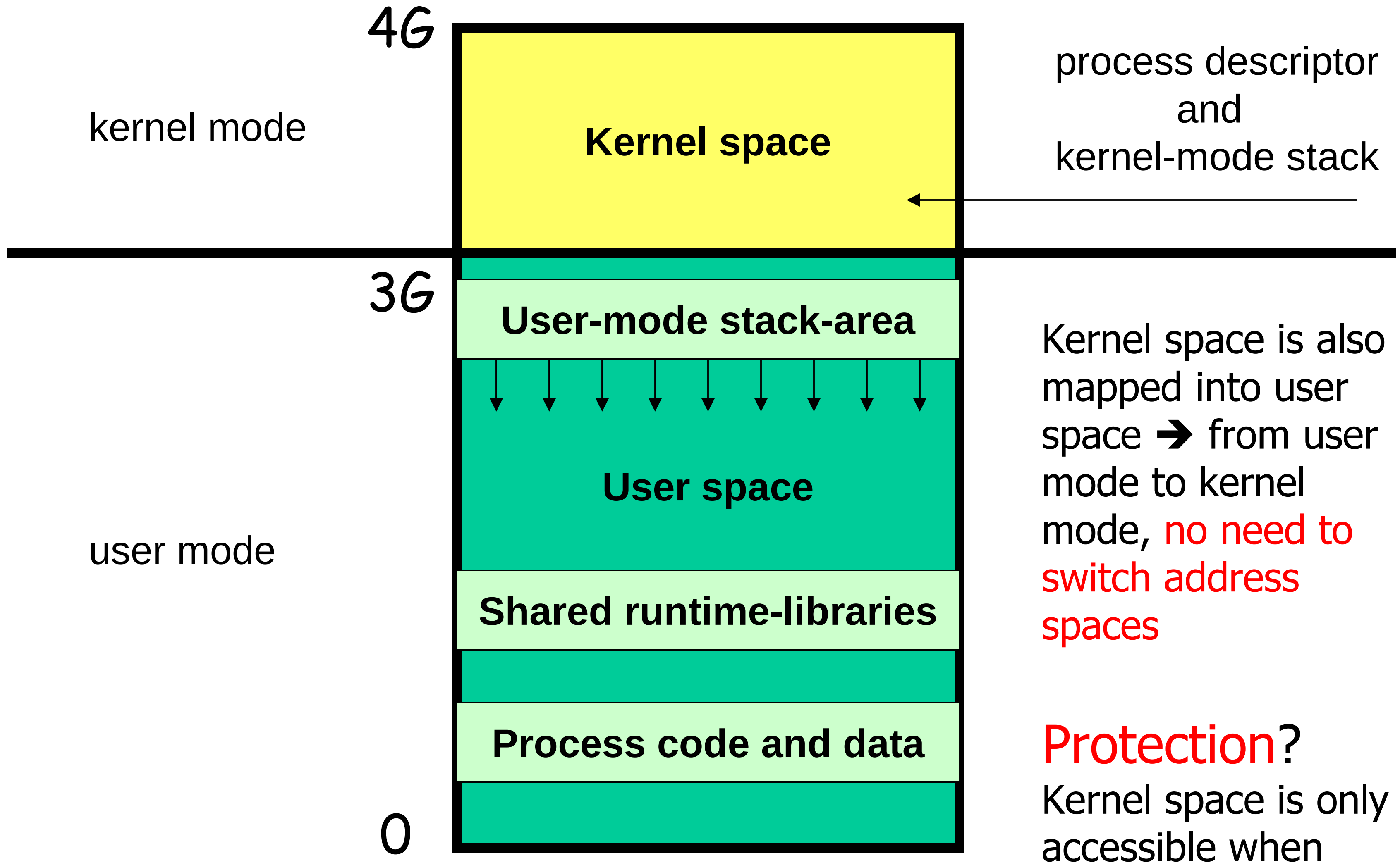
- Kernel needs work space as well
    - Store kernel code, data, heap, and stack
        - E.g., process control blocks
    - Must be protected from user processes

- Can give kernel its own address space

- Problem: switching address space is costly

- Solution: map kernel address space into process address space

Linux process address space



4G

kernel mode

Kernel space

process descriptor
and
kernel-mode stack

3G

User-mode stack-area

User space

user mode

Shared runtime-libraries

Process code and data

0

Kernel space is also mapped into user space ➔ from user mode to kernel mode, no need to switch address spaces

Protection?

Kernel space is only accessible when mode bit = 0

Linux: processes or threads?

- Linux uses a neutral term: tasks
    - Tasks represent both processes and threads
    - When processes trap into the kernel, they share the Linux kernel's address space ➔ kernel threads

Outline

- Linux process overview
- Linux process data structures
- Linux process operations

Linux task data structures

- task_struct: process control block
- thread_info: low level task data, directly accessed from entry.S
- kernel stack: work space for systems calls (the kernel executes on the user process's behalf) or interrupt handlers
- Task queues: queues that chain tasks together

Process Control Block in Linux

- task_struct (process descriptor in ULK)
  - include/linux/sched.h
  - Each task has a unique task_struct

Process states

- state: what state a process is in
  - TASK_RUNNING – the thread is running on the CPU or is waiting to run
  - TASK_INTERRUPTIBLE – the thread is sleeping and can be awoken by a signal (EINTR)
  - TASK_UNINTERRUPTIBLE – the thread is sleeping and cannot be awakened by a signal
  - TASK_STOPPED – the process has been stopped by a signal or by a debugger
  - TASK_TRACED – the process is being traced via the ptrace system call

- exit_state: how a process exited
  - EXIT_ZOMBIE – the process is exiting but has not yet been waited for by its parent
  - EXIT_DEAD – the process has exited and has been waited for

Hardware state

- Thread: *thread_struct* – hardware state, e.g., registers
- x86 hardware state is defined in include/asm-i386/processor.h

Process scheduling

- prio: priority of the process
- Static_prio, run_list, array, sleep_avg, timestamp, last_ran, time_slice, …
  - More on Linux scheduling later

Process IDs

- process ID: pid
- thread group ID: tgid
  - pid of first thread in process
  - getpid() returns this ID, so all threads in a process share the same process ID

- many system calls identify a process by its PID
  - Linux kernel uses pidhash to efficiently find processes by pids
  - see include/linux/pid.h, kernel/pid.c

Process Relationships

- Processes are related: children, sibling
  - Parent/child (fork()), siblings
  - Possible to "re-parent"
    - Parent vs. original parent
  - Parent can "wait" for child to terminate

- Process groups: signal_struct->pgrp
  - Possible to send signals to all members

- Sessions: signal_struct->session
  - Processes related to login

Other PCB data structures

- user: *user_struct* – per-user information (for example, number of current processes)
- mm, active_mm: *mm_struct* – memory areas for the process (address space)
- fs: *fs_struct* – current and root directories associated with the process
- files: *files_struct* – file descriptors for the process
- signal: *signal_struct* – signal structures associated with the process

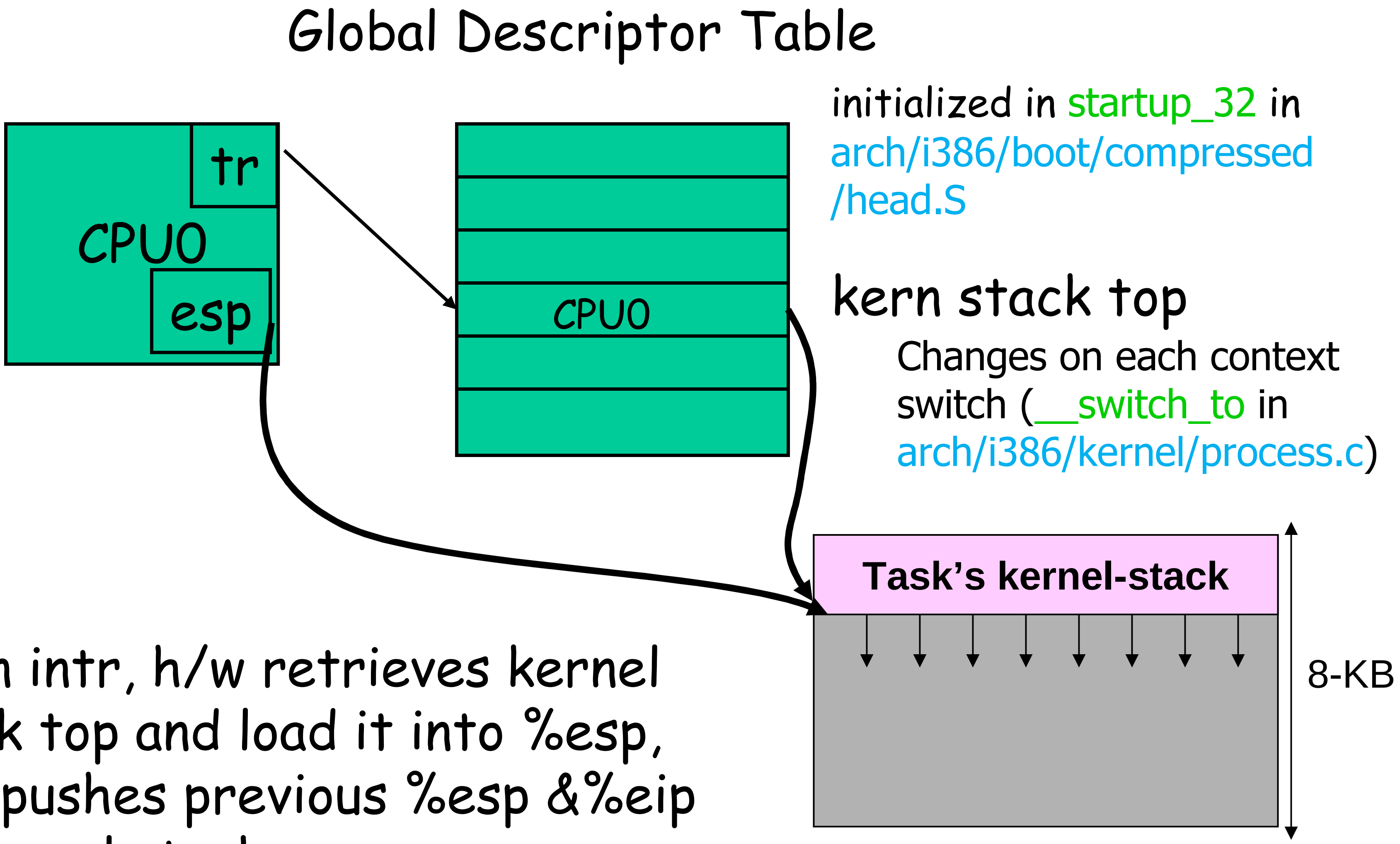
thread_info

- include/asm-i386/thread_info.h
- low level task data, directly accessed from entry.S
- current_thread_info: get current thread_info struct from C

kernel stack

- Each process in Linux has two stacks, a user stack and a kernel stack (8KB by default)
  - Kernel stack can only be accessed in kernel mode
  - Kernel code runs on kernel stack
- Why not reuse user-space stack?
  - homework

Finding kernel stack (on x86)



initialized in startup_32 in arch/i386/boot/compressed /head.S

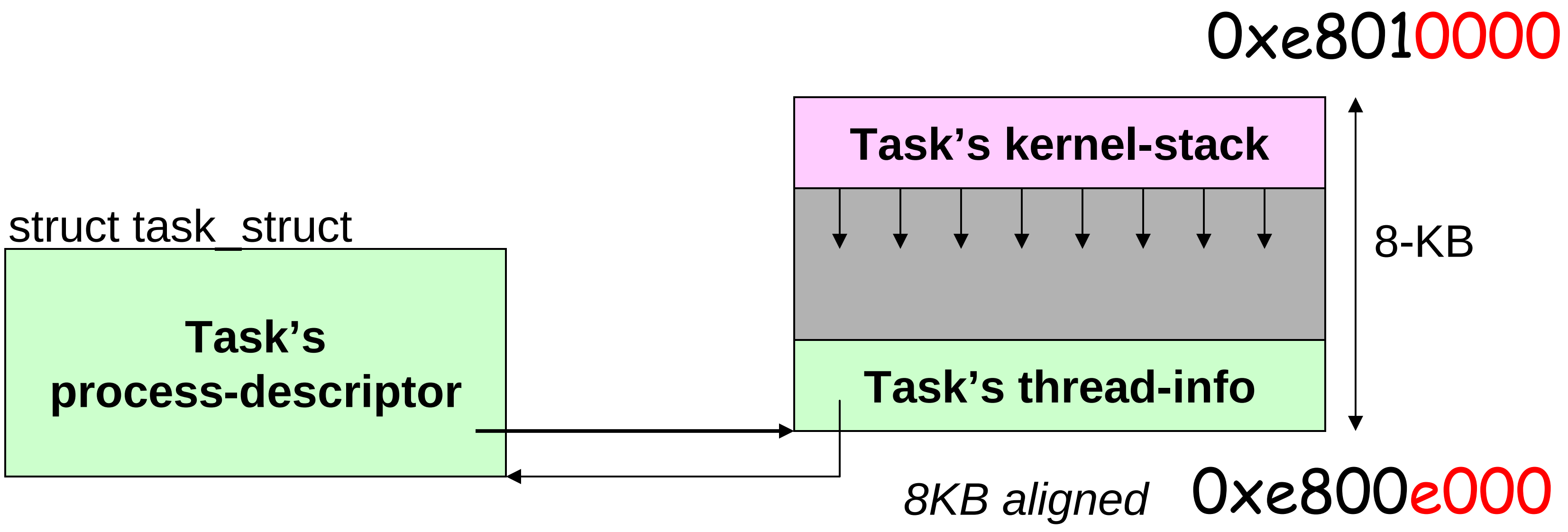
kern stack top

Changes on each context switch (__switch_to in arch/i386/kernel/process.c)

Upon intr, h/w retrieves kernel stack top and load it into %esp, also pushes previous %esp &%eip on kernel stack

Connections between task_struct and kernel stack

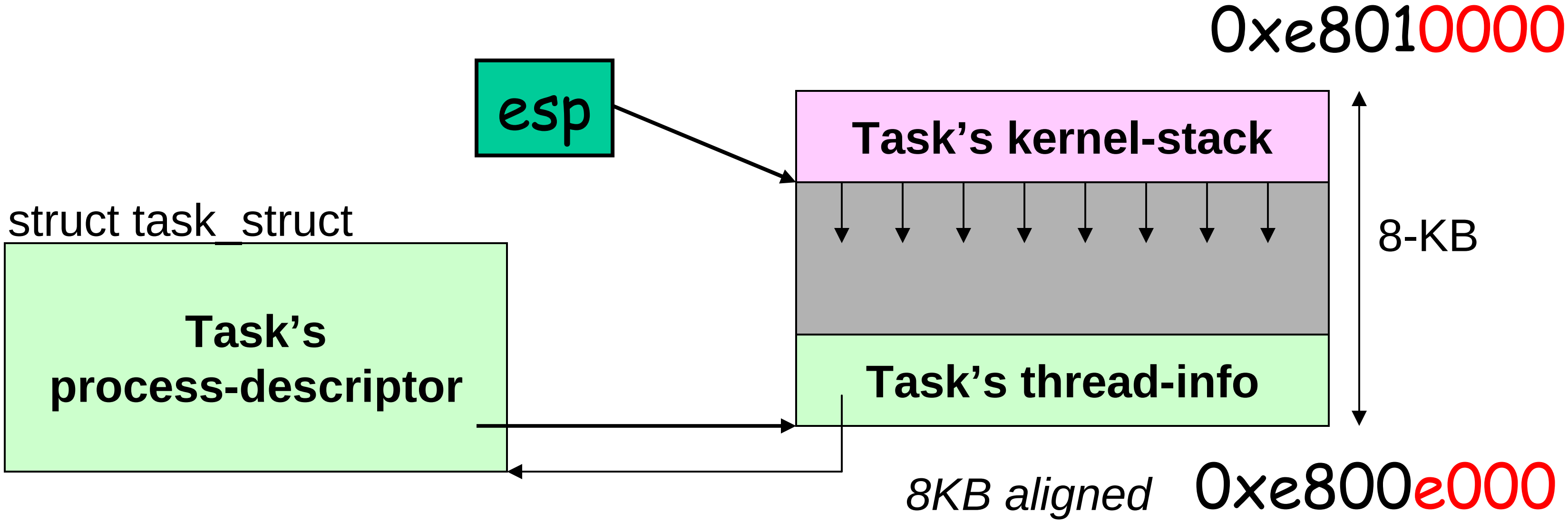
- Linux uses part of a task's kernel-stack to store a structure thread_info
- thread_info and task_struct contain pointers to each other



How to find thread_info from kernel stack?

movl $0xFFFFE000, %eax

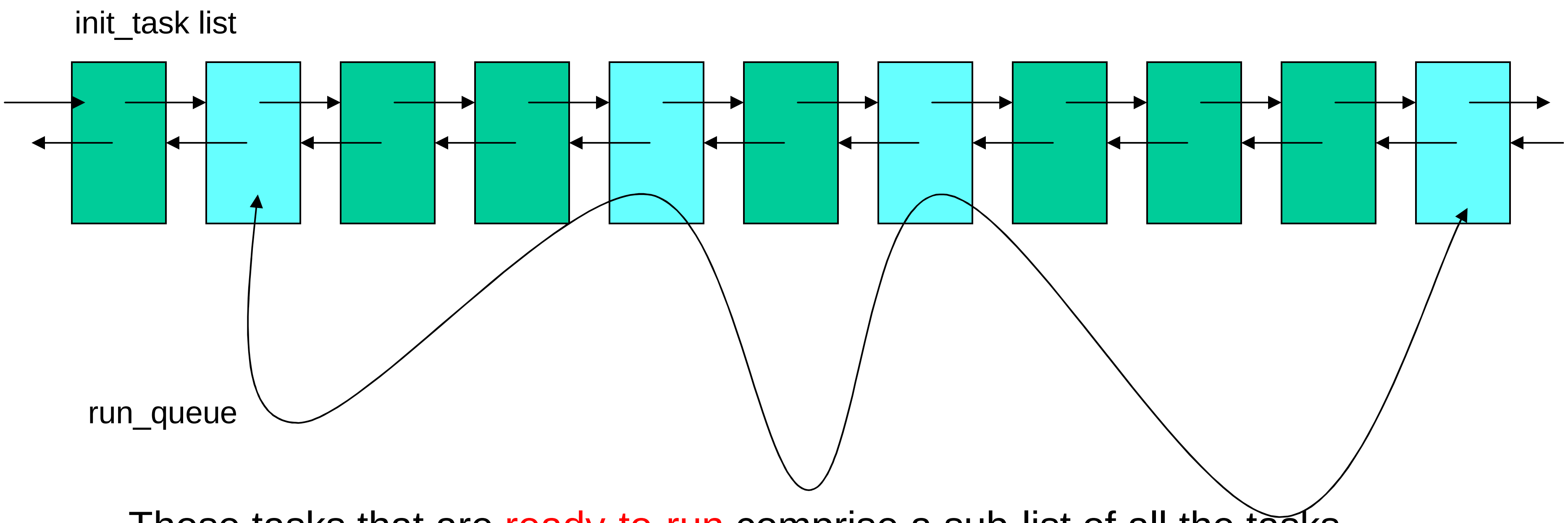
andl %esp, %eax (mask out last 13 bits)



How Linux manages processes

- Linux uses multiple queues to manage processes
  - Queue for all tasks
  - Queue for "running" tasks
  - Queues for tasks that temporarily are "blocked" while waiting for a particular event to occur
- These queues are implemented using doubly-linked list (struct list_head in include/linux/list.h)

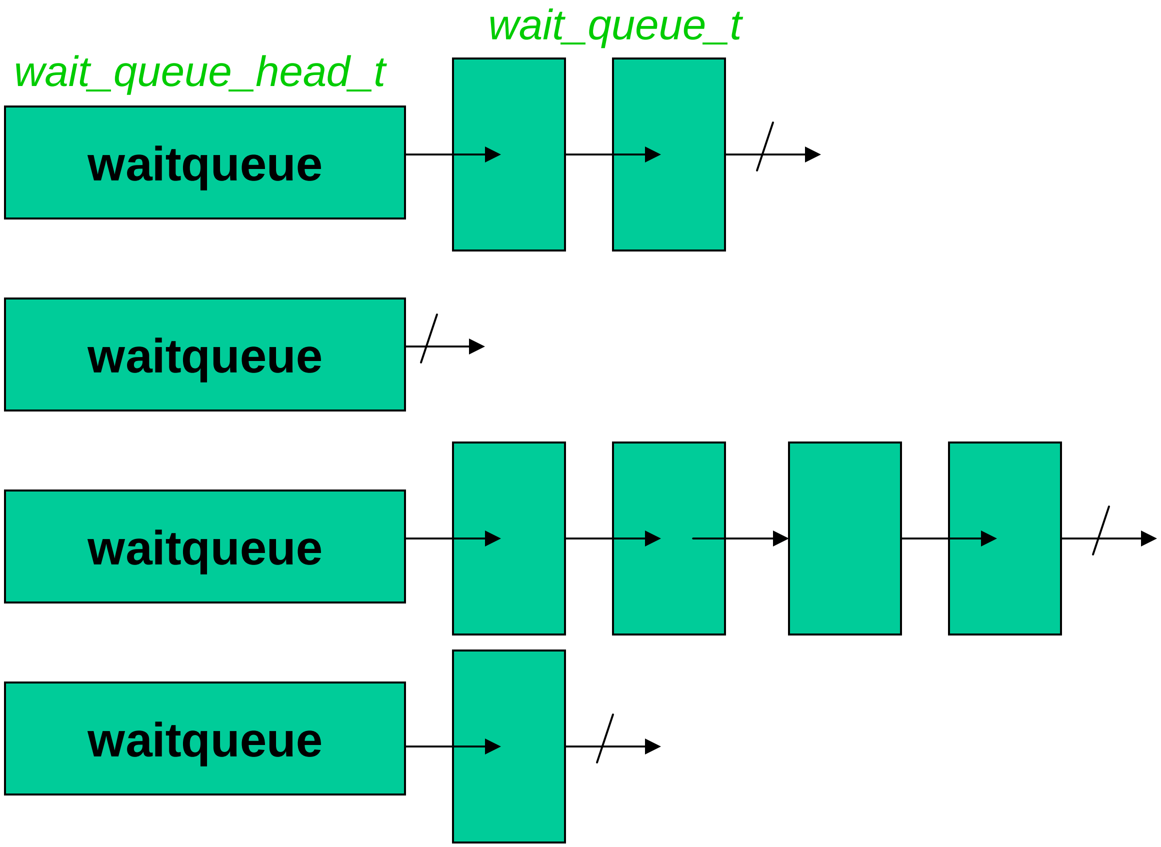
Some tasks are ‘ready-to-run’



Those tasks that are ready-to-run comprise a sub-list of all the tasks, and they are arranged on a queue known as the ‘run-queue’ (struct *runqueue* in kernel/sched.c)

Those tasks that are blocked while awaiting a specific event to occur are put on alternative sub-lists, called ‘wait queues’, associated with the particular event(s) that will allow a blocked task to be unblocked (*wait_queue_t* in include/linux/wait.h and kernel/wait.c)

Kernel Wait Queues



wait_queue_head_t can have 0 or more *wait_queue_t* chained onto them

However, usually just one element

Each *wait_queue_t* contains a *list_head* of tasks

All processes waiting for specific "event“

Used for timing, synch, device i/o, etc.

Outline

- ❑ Linux process overview
- ❑ Linux process data structures
- ❑ Linux process operations

fork() call chain

- libc fork()
  - system_call (arch/i386/kernel/entry.S)
    - sys_clone() (arch/i386/kernel/process.c)
      - do_fork() (kernel/fork.c)
        - copy_process() (kernel/fork.c)
          - p = dup_task_struct(current) // shallow copy
          - copy_* // copy point-to structures
          - copy_thread () // copy stack, regs, and eip
        - wake_up_new_task() // set child runnable

exit() call chain

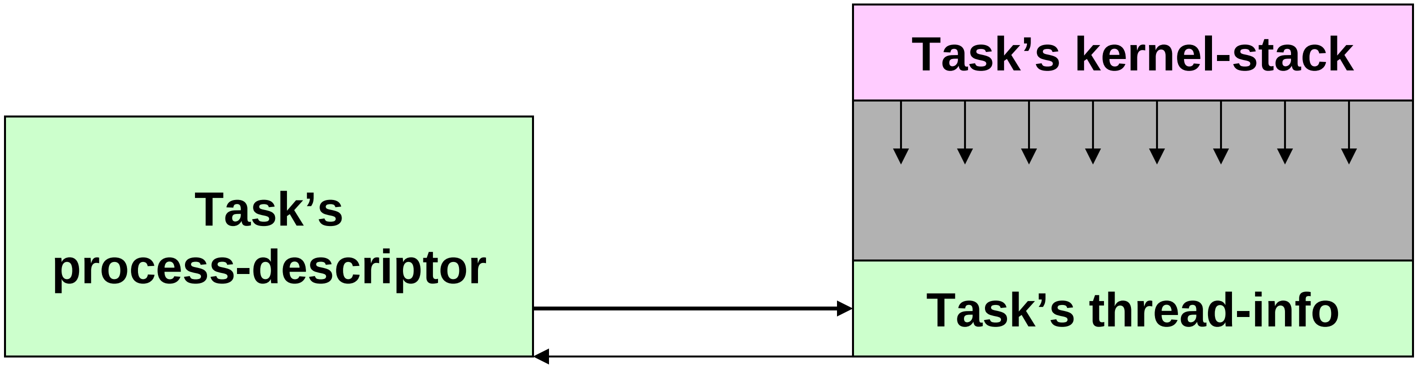
- libc exit(code)
  - system_call (arch/i386/kernel/entry.S)
    - sys_exit() (kernel/exit.c)
      - do_exit() (kernel/exit.c)
        - exit_*() // free data structures
        - exit_notify() // tell other processes we exit
          // reparent children to init
          // EXIT_ZOMBIE
          // EXIT_DEAD

Context switch call chain

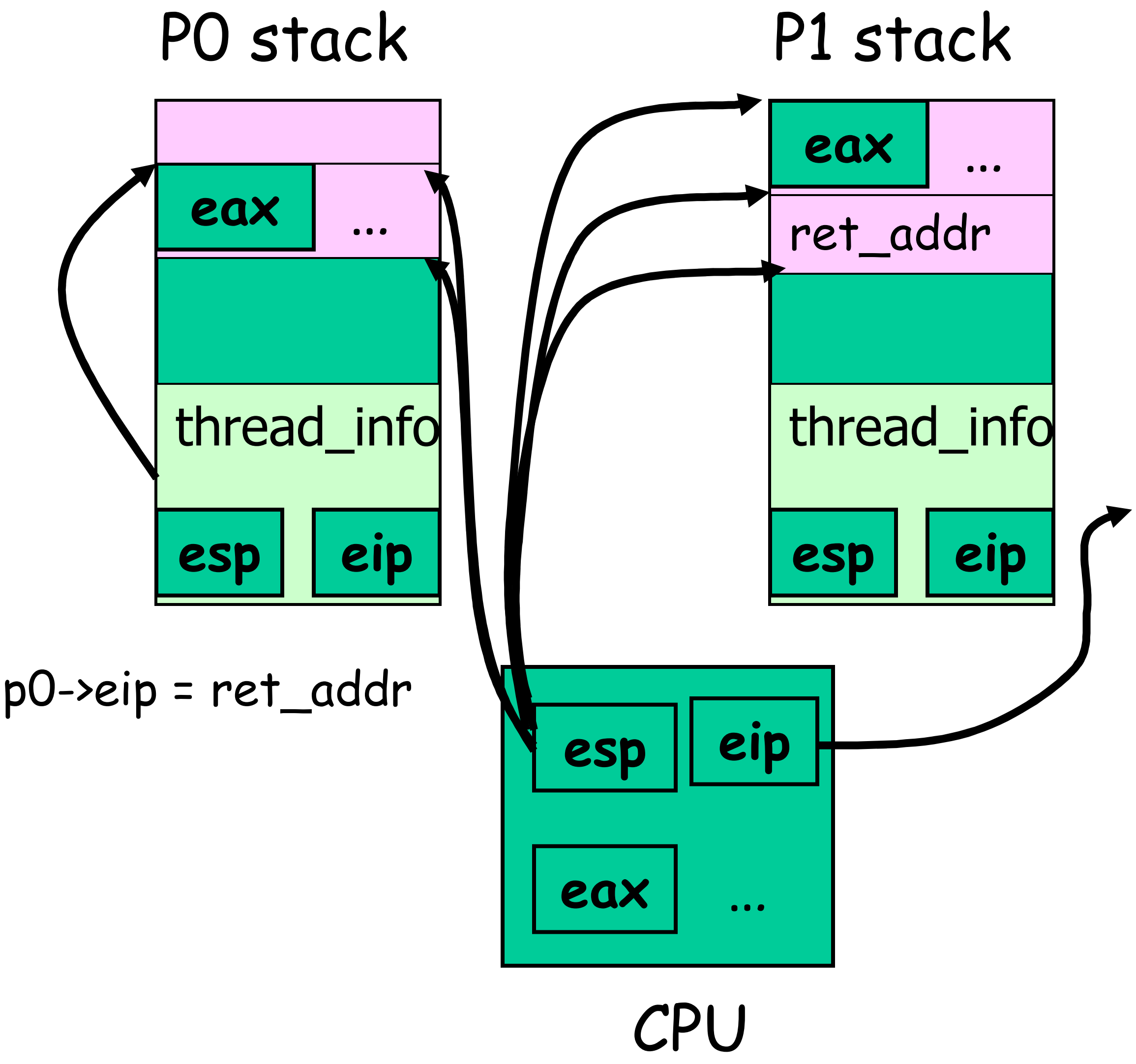
- schedule() (kernel/sched.c) (talk about scheduling later)
  - context_switch()
    - swtich_mm (include/asm-i386/mmu_context.h)
      // switch address space
    - switch_to (include/asm-i386/system.h)
      - __swtich_to (arch/i386/kernel/process.c)
        // switch stack to switch CPU context

__swtich_to: context switch by stack switch (the idea)

- Kernel stack captures process states
  - All registers
  - task_struct through thread_info
- Changing the stack pointer %esp changes the process



Context switch by stack switch (the simplified implementation)



swtich_to(p0,p1)
- save registers on stack
- p0->esp = %esp
- p0->eip = ret_addr;
- %esp = p1->esp;
- push p1->eip;
- ret

ret_addr:
- pop registers from stack