W4118: xv6 and Linux processes

Instructor: Junfeng Yang

References: Modern Operating Systems (3rd edition), Operating Systems Concepts (8th edition), previous W4118, and OS at MIT, Stanford, and UWisc

xv6 processes

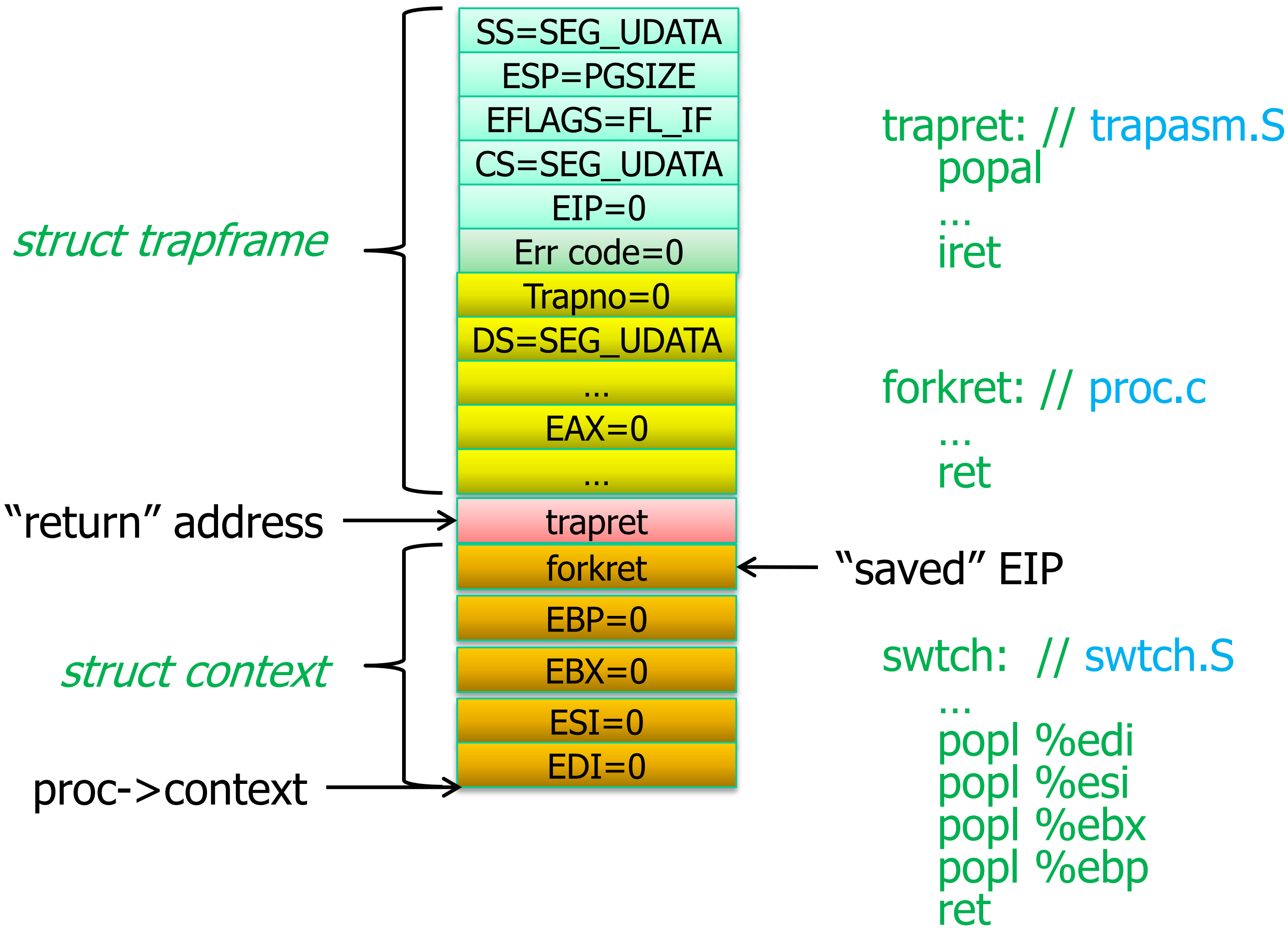
- How to create the first user process
- fork()
- exit()
- wait()
- kill()
- exec()
- sleep()
- wakeup()

Create the first user process

- Idea: create a fake trap frame, then reuse trap return mechanism

- userinit() in proc.c
  - allocproc() in vm.c allocates PCB, sets trap return address to trapret in trapasm.S, and sets "saved" kernel CPU context
  - inituvm() in vm.c sets up user space
    - Allocates a physical page for the process, sets up page table, and copies initcode
  - Set up fake trap frame
  - Set up current working directory

Init process's kernel stack



initcode.S

```
// equivalent C code
char init[] = "/init\0";
char *argv = {init, 0};
exec(init, argv);
for(;;) exit();
```

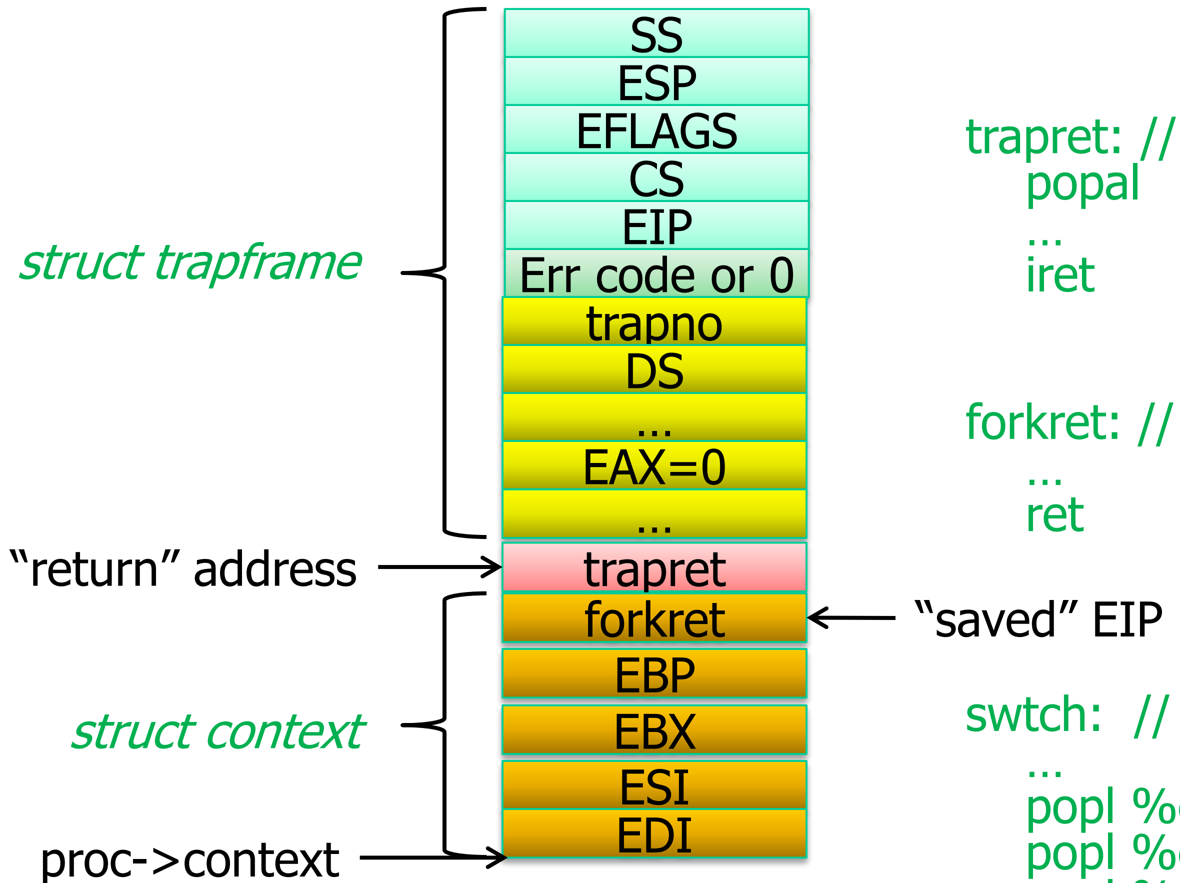
- Assembly code that
  - Sets up system call arguments
  - Moves SYS_exec to EAX
  - Traps into kernel via INT 64
- Execute init generated from init.c
- Compiled and linked into kernel
  - Makefile

fork()

- syspro.c, proc.c

- Allocate new PCB and stack
  - Set up EIP of child to forkret → trapret
- Copy address space
  - Copy both page tables and physical pages
  - Can you do better?
- Set parent pointer
- Copy parent's trap frame
- Change EAX in trap frame so that child returns 0
- Copy open file table

Child process's kernel stack




```
trapret: // trapasm.S
    popal
    ...
    iret

forkret: // proc.c
    ...
    ret

swtch:  // swtch.S
    ...
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

exit()

- sysproc.c, proc.c

- Close open files
- Decrement reference count to current working directory
- Wake up waiting parents
- Re-parent children to init
- Set state to zombie
- Yield to scheduler

wait()

- sysproc.c, proc.c
- Find a zombie child by iterating process table
  - Can you do better?
- If there is one,
  - Free their PCB and other resources
  - Return child PID
- If no child or killed, return -1
- Repeat

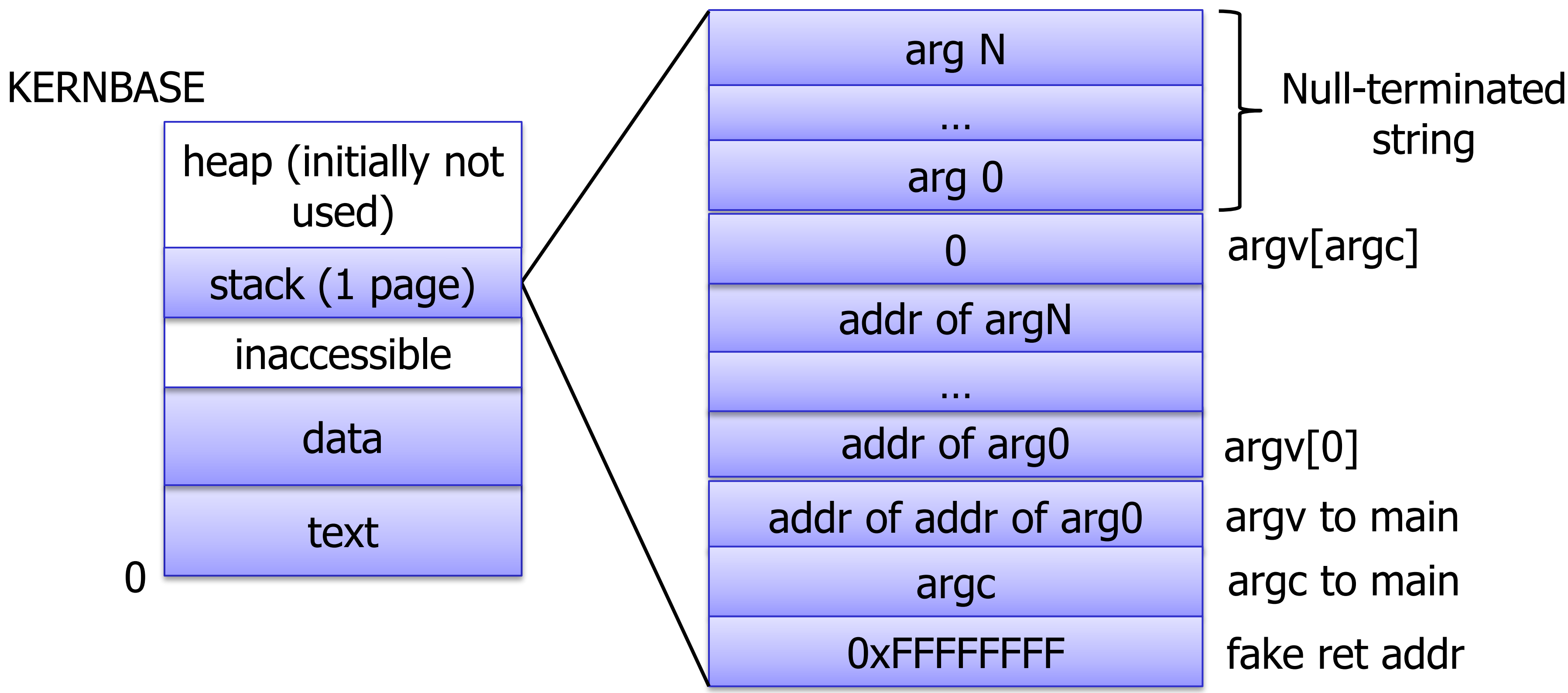
kill()

- syspro.c, proc.c

- Set proc->killed to 1
- At various places in kernel, check this flag, and if process is killed, exit
  - trap() in trap.c
  - sys_sleep() in sysproc.c
  - piperead() & pipewrite() in pipe.c
  - proc.c

exec()

- sysfile.c, exec.c
- Set up user page table
- Load segments of the executable file into memory
- Set up stack and arguments to main(int argc, char* argv[])
- Jump to entry point (main) of the executable



sleep()

- proc.c

- Remember what we wait for (proc->chan)
- Set process state
- Yield to scheduler

wakeup()

- proc.c

- Scan through all processes
- Wake up those waiting on chan

Linux processes

- Relevant source files
- Linux process control block
- Process queues
- Context switching
- Creating and destroying processes

Header Files

- The major header files used for process management are:

  `include/linux/sched.h` - declarations for most task data structures

  `include/linux/threads.h` - some configuration constants (unrelated to threads)

  `include/linux/times.h` - time structures

  `include/linux/time.h` - time declarations

  `include/linux/timex.h` - wall clock time declarations

Source Code

- The source code for process and thread management is in the `kernel` directory:
  - `sched.c` - task scheduling routines
  - `signal.c` - signal handling routines
  - `fork.c` - process/thread creation routines
  - `exit.c` - process exit routines
  - `time.c` - time management routines
  - `timer.c` - timer management routines
- The source code for the program initiation routines is in `fs/exec.c`.

Linux processes

- Relevant source files
- Linux process control block
- Process queues
- Context switching
- Creating and destroying processes

Linux: Processes or Threads?

- Linux uses a neutral term: tasks
  - Tasks represent both processes and threads
- Linux view
  - Threads: processes that share address space
  - Linux "threads" (tasks) are really "kernel threads“
- Lighter-weight than traditional processes
  - File descriptors, VM mappings need not be copied
  - Implication: file table and VM table not part of process descriptor

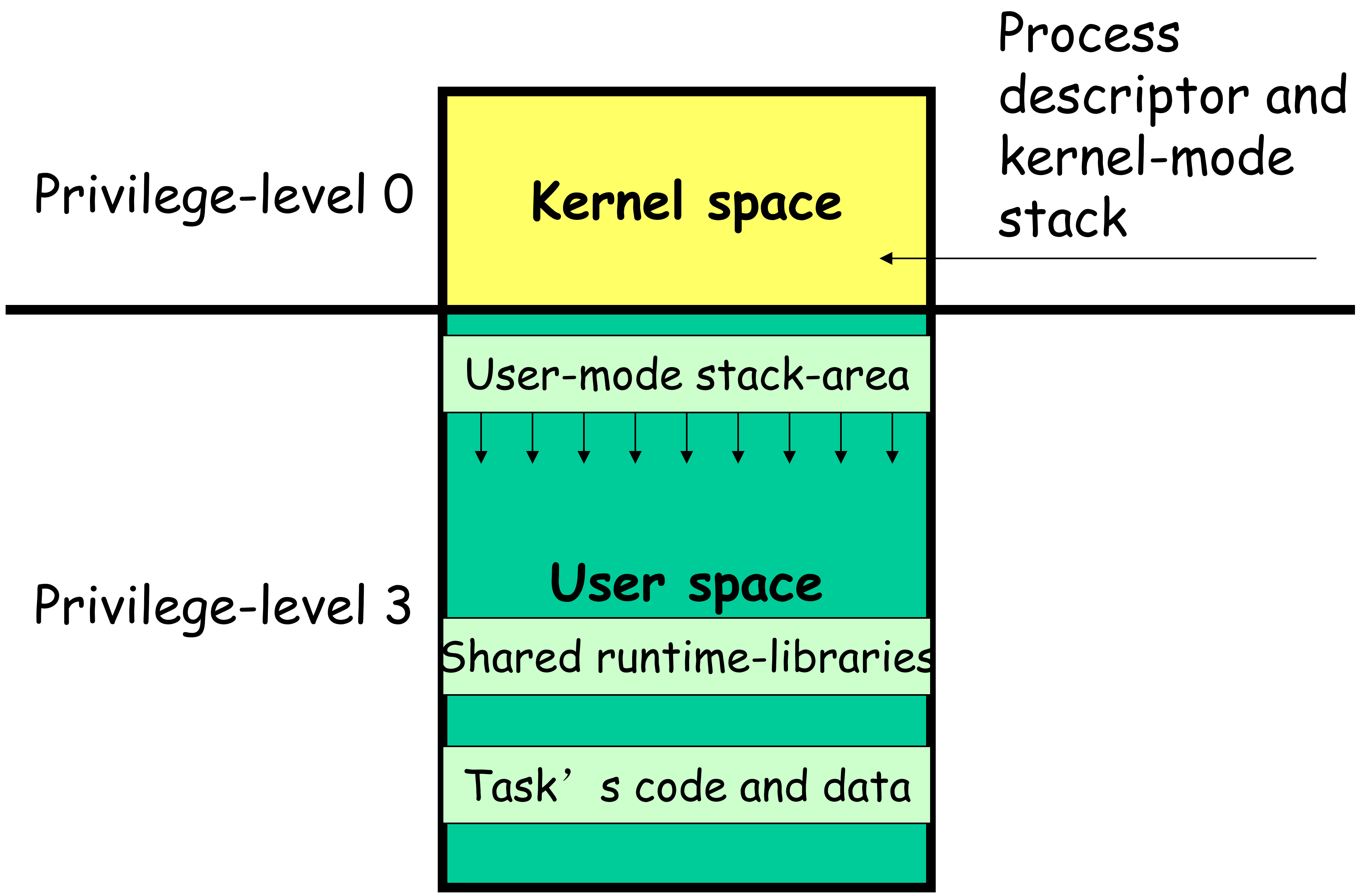
Stacks and task-descriptors

- To manage multitasking, the OS needs to use a data-structure which can keep track of every task's progress and usage of the computer's available resources (physical memory, open files, pending signals, etc.)
- Such a data-structure is called a 'process descriptor' – every active task needs one
- Every task needs its own 'private' stack
- So every task, in addition to having its own code and data, will also have a stack-area that is located in user-space, plus another stack-area that is located in kernel-space
- Each task also has a process-descriptor which is accessible only in kernel-space

Kernel Stacks

- Why need a special kernel stack?
  - Kernel can't trust addresses provided by user
  - Address may point to kernel memory
  - Address may not be mapped
  - Memory region may be swapped out from physical RAM
  - Leftover data from kernel ops could be read by process
- Why a different stack for every process?
  - What to do if a process sleeps while executing kernel code?
  - Wasn't a problem up to Linux 2.4; not pre-emptive
  - Need multiple kernel stacks for pre-emptive kernels

A task's virtual-memory layout



Process Descriptor

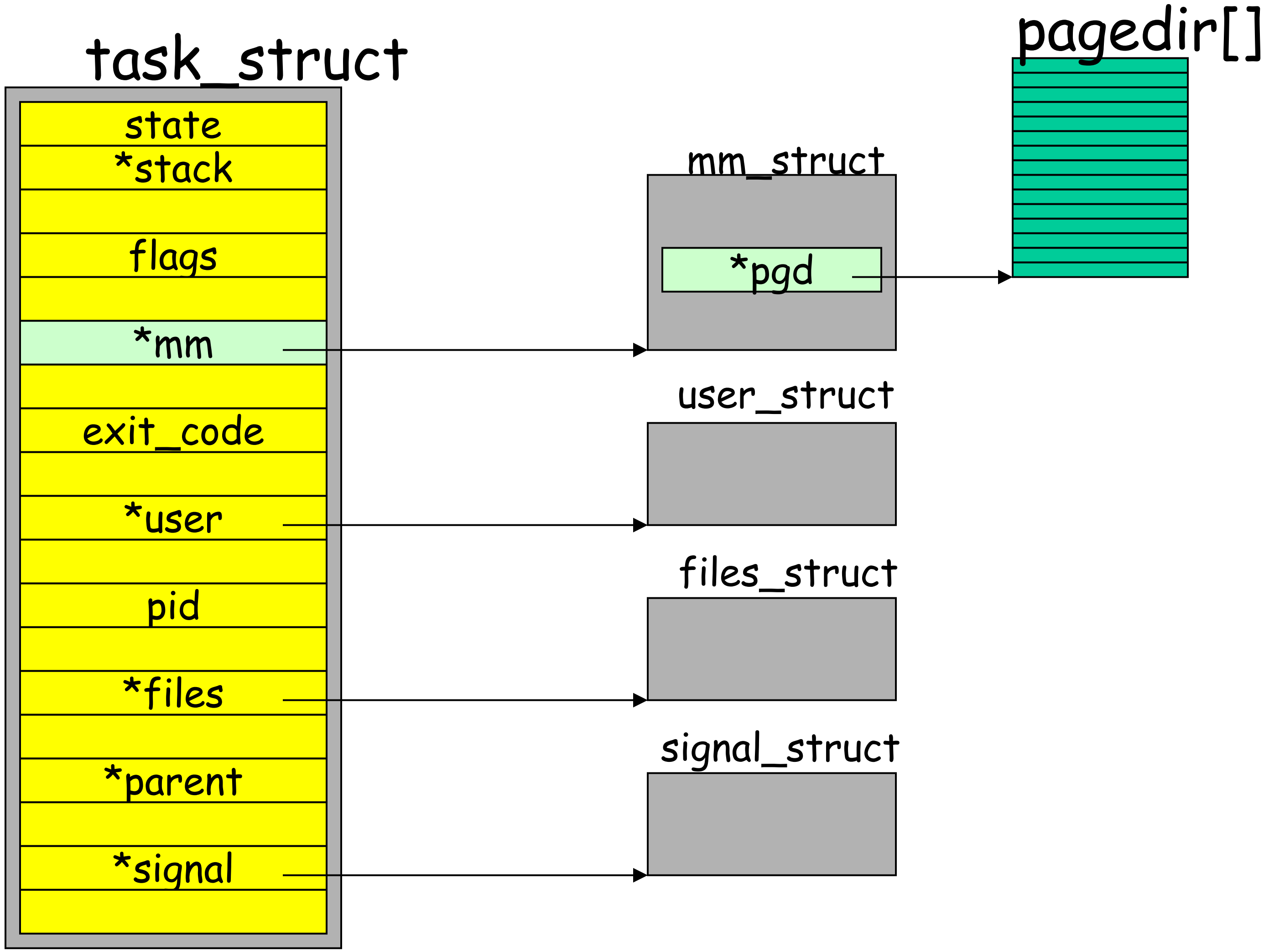
- Process – dynamic, program in motion
  - Kernel data structures to maintain "state"
  - Descriptor, PCB (control block), task_struct
  - Larger than you think! (about 1K)
  - 160+ fields
  - Complex struct with pointers to others
- Type of info in task_struct
  - state, id, priorities, locks, files, signals, memory maps, locks, queues, list pointers, …
- Some details
  - Address of first few fields hardcoded in asm
  - Careful attention to cache line layout

The Linux process descriptor

Each process descriptor contains many fields

and some are pointers to other kernel structures

which may themselves include fields that point to structures

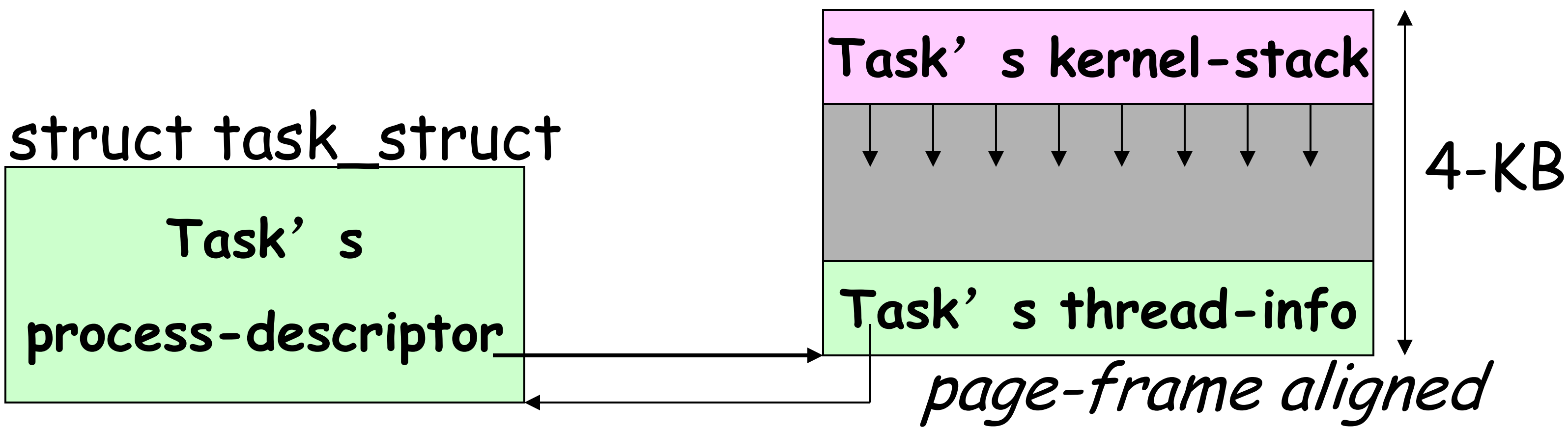


The Task Structure

- The *task_struct* is used to represent a task.
- The *task_struct* has several sub-structures that it references:
  - *tty_struct* – TTY associated with the process
  - *fs_struct* – current and root directories associated with the process
  - *files_struct* – file descriptors for the process
  - *mm_struct* – memory areas for the process
  - *signal_struct* – signal structures associated with the process
  - *user_struct* – per-user information (for example, number of current processes)

Process/Thread Context

- Linux uses part of a task's kernel-stack page-frame to store thread information
- The thread_info includes a pointer to the task's process-descriptor data-structure



Finding a task's 'thread-info'

- During a task's execution in kernel-mode, it's very quick to find that task's *thread_info* object
- Just use two assembly-language instructions:

```
movl    $0xFFFFF000, %eax
andl    %esp, %eax
```

Ok, now %eax = the thread-info's base-address

- Masking off 13 bits of the stack yields *thread_info*
- Macro *current_thread_info* implements this computation
- *thread_info* points to *task_struct*
- *current* macro yields the *task_struct*
- *current* is not a static variable, useful for SMP

Finding task-related kernel-data

- Use a macro '**task_thread_info**(task)' to get a pointer to the 'thread_info' structure:

```
struct thread_info  *info = task_thread_info( task );
```

- Then one more step gets you back to the address of the task's process-descriptor:

```
struct task_struct  *task = info->task;
```

PID Hashes and Task Lookup

- PID: 16-bit process ID
- *task_struct*s are found by searching for *pid* structures, which point to the *task_struct*s. The *pid* structures are kept in several hash tables, hashed by different IDs:
  - process ID
  - thread group ID // pid of first thread in process
  - process group ID // job control
  - session ID // login sessions
  - (see `include/linux/pid.h`)
- Allocated process IDs are recorded in a bitmap representing around four million possible IDs.
- PIDs dynamically allocated, avoid immediate reuse

Process Relationships

- Processes are related
  - Parent/child (fork()), siblings
  - Possible to "re-parent"
    - Parent vs. original parent
  - Parent can "wait" for child to terminate
- Process groups
  - Possible to send signals to all members
- Sessions
  - Processes related to login

Task Relationships

- Several pointers exist between *task_struct*s:
  - *parent* – pointer to parent process
  - *children* – pointer to linked list of child processes
  - *sibling* – pointer to task of "next younger sibling" of current process
- *children* and *sibling* point to the *task_struct* for the first thread created in a process.
- The *task_struct* for every thread in a process has the same pointer values.

Task States

From kernel-header: <linux/sched.h>

- #define TASK_RUNNING 0
- #define TASK_INTERRUPTIBLE 1
- #define TASK_UNINTERRUPTIBLE 2
- #define __TASK_STOPPED 4
- #define __TASK_TRACED 8
- #define EXIT_ZOMBIE 16
- #define EXIT_DEAD 32
- #define TASK_DEAD 64
- #define TASK_WAKEKILL 128
- #define TASK_WAKING 256

Task States

- 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
- TASK_DEAD – the process is being cleaned up and the task is being deleted
- TASK_WAKEKILL – similar to TASK_UNINTERRUPTIBLE with the ability to respond to fatal signals
- TASK_WAKING – someone is already waking the task

Exit States

```
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
```

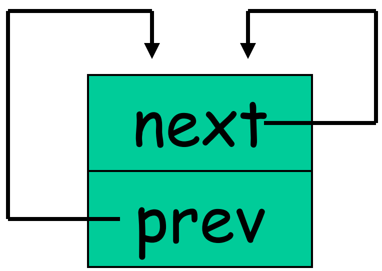
- EXIT_ZOMBIE - the process is exiting but has not yet been waited for by its parent
- EXIT_DEAD - the process has exited and has been waited for

Linux processes

- Relevant source files
- Linux process control block
- Process queues
- Context switching
- Creating and destroying processes

List Operations

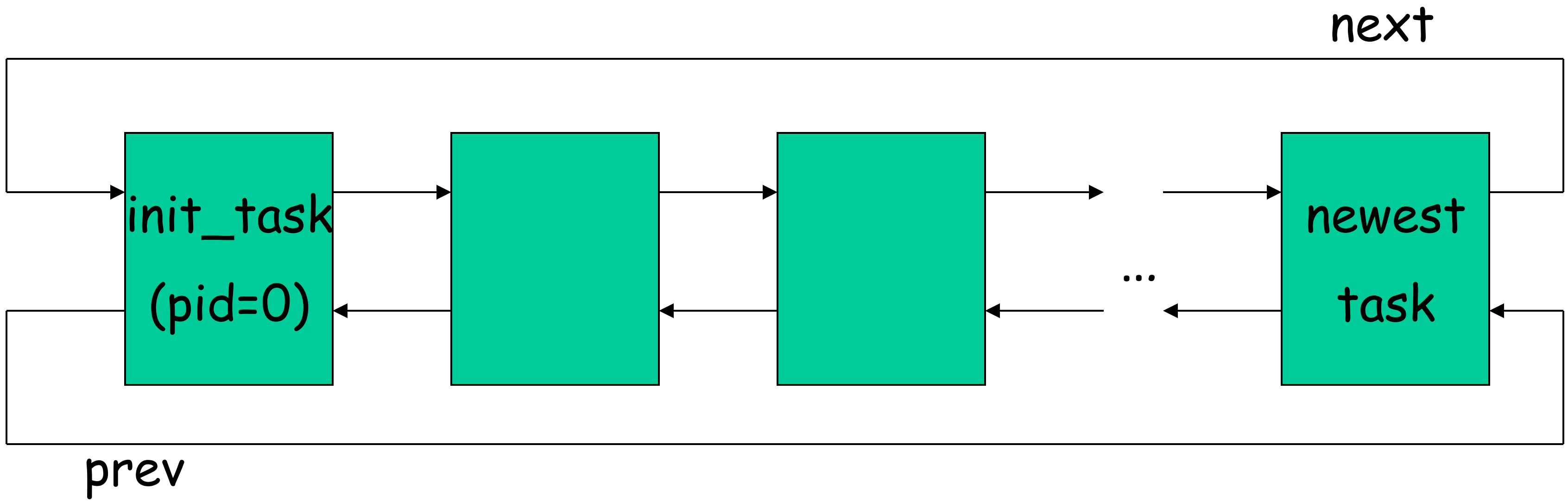
- The *list_head* is a generic list structure with a set of services:
  - *LIST_HEAD* - declare and initialize list head
  - *list_add* - add a *list_head* after item
  - *list_add_tail* - add a *list_head* before item
  - *list_del* - remove *list_head* from list
  - *list_del_init* - remove and initialize *list_head*
  - *list_empty* - is a list empty?
  - *list_for_each, list_for_each_entry, list_entry*



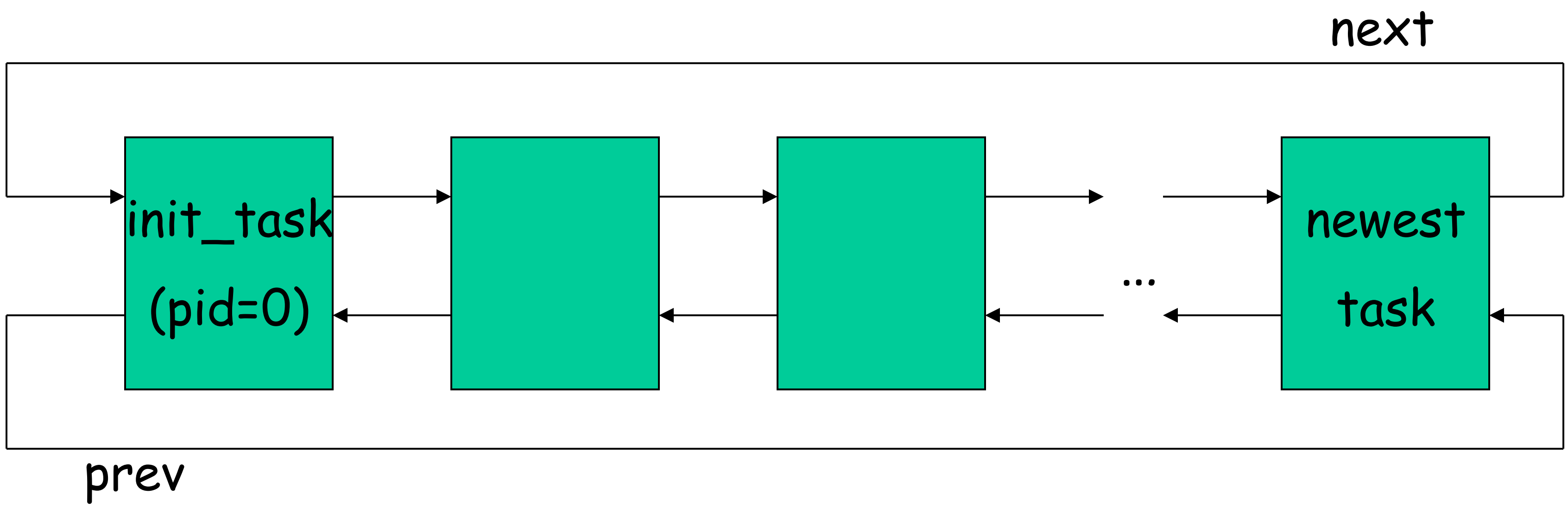
The Kernel's 'task-list'

- Kernel keeps a list of process descriptors
- A 'doubly-linked' circular list is used
- The 'init_task' serves as a fixed header
- Other tasks inserted/deleted dynamically
- Tasks have forward & backward pointers, implemented as fields in the 'tasks' field
- To go forward: task = next_task(task);
- To go backward: task = prev_task(task);

Doubly-linked Circular List



Locking during Access

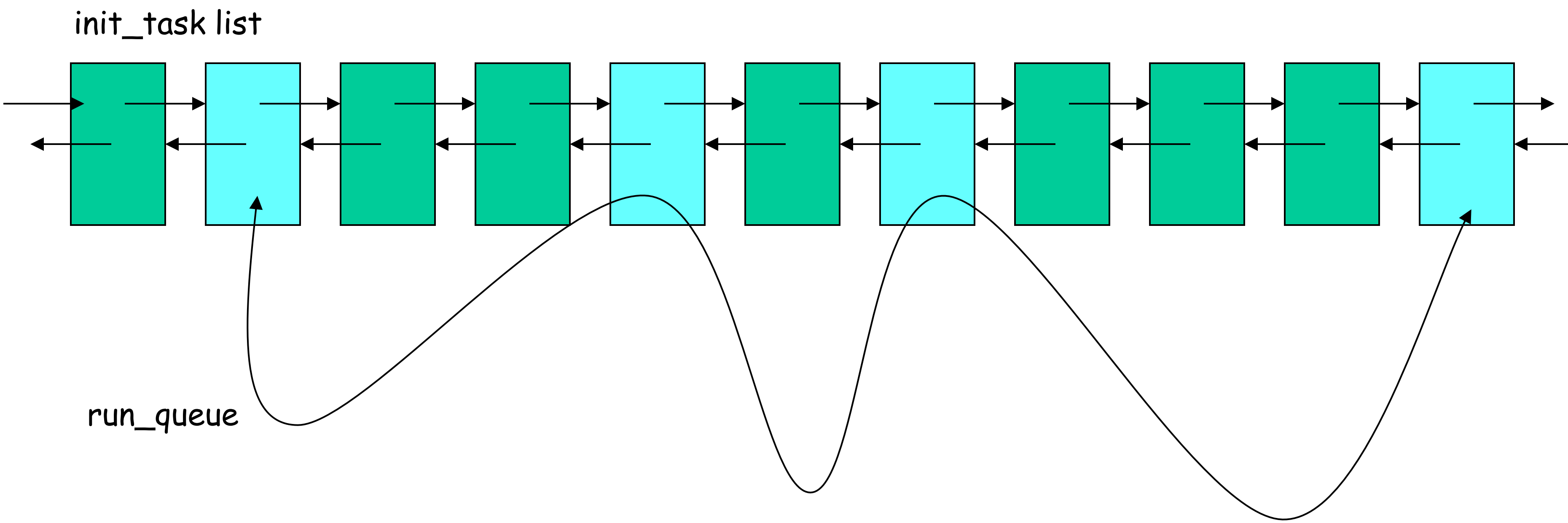


- When traversing the task list, must protect against concurrent accesses
  - read_lock_irq(&tasklist_lock), read_unlock_irq(&tasklist_lock)
- When modifying a task_struct
  - task_lock(task), task_unlock(task)
- Don't sleep when holding a lock on task list or structs!

‘run’ queues and ‘wait’ queues

- In order for Linux to efficiently manage the scheduling of its various ‘tasks’, separate queues are maintained for ‘running’ tasks and for tasks that temporarily are ‘blocked’ while waiting for a particular event to occur (such as the arrival of new data from the keyboard, or the exhaustion of prior data sent to the printer)

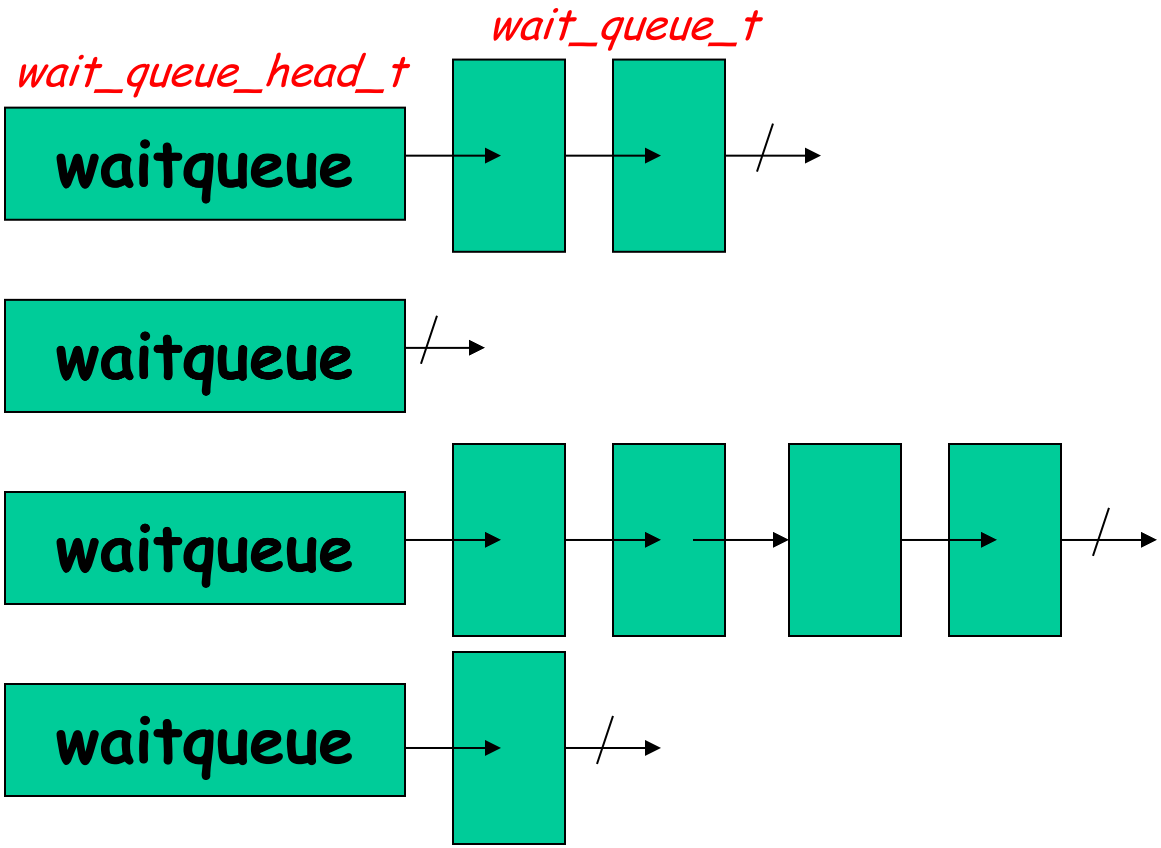
Some tasks are 'ready-to-run'



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'

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

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.

Transition between queues

- Wait on a wait queue
  - add_wait_queue, prepare_to_wait, schedule/schedule_timeout, finish_wait
  - Options: TASK_INTERRUPTIBLE, exclusive, timeout
  - Other functions available
- Wake up
  - wake_up_process, …
- LKD page 58--60

Linux processes

- Relevant source files
- Linux process control block
- Process queues
- Context switching
- Creating and destroying processes

Context Switch

- *schedule* determines the next task to run, calls *context_switch* (kernel/sched.c)
- calls *switch_mm* to change the process address space
- calls *switch_to* (include/asm/system.h and arch/x86/kernel/process_32.c) to context switch to the new task.

Context Switch: switch_mm

- *switch_mm* is architecture specific. It generally loads any hardware state required to make the process' user address space addressible in user mode. If the address space is unchanged (task switching between threads in one process), very little is done.

Context Switch: switch_to

- *switch_to* is architecture specific.
- Generally, it saves the old task's hardware state of the CPU (registers) to one of three places:
  - The task's kernel stack
  - the *thread_struct*
  - *task_struct->thread*
- It then copies the new task's hardware state from the appropriate places
  - Stack is in *next->thread.esp*

The Role of the Stack

- One process must save state where another can find it
- When the new state is loaded, the CPU *is* running another process -- the state *is* the process!
- The stack pointer determines most of the state
- Some of the registers are on the stack
- The stack pointer determines the location of *thread_info,* which also points to *task struct*
- *Changing the stack pointer changes the process!*

Linux processes

- Relevant source files
- Linux process control block
- Process queues
- Context switching
- Creating and destroying processes

Creating New Processes

- The *fork* system call is used to create a new process.
  - Identical to parent except …
  - execution state
  - process ID
  - parent process ID.
  - other data is either copied (like process state) or made copy on write (like process address space).
- Copy on write allows data to be shared as long as it is not modified, but each task gets its own copy when one task tries to modify the data.

Creating New Processes (cont.)

- The *fork* system call uses *do_fork* to create a new task. The flags passed to *do_fork* indicate which task attributes to copy and which to create anew.
- *do_fork* calls *copy_process* to create a new *task_struct* and initialize it appropriately.

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
```

do_fork

- *do_fork* creates a new task and allows the new task to share resources with the calling task.
- The following options specify what should be shared with the calling task:
  - CLONE_VM - share address space
  - CLONE_FS - share root and current working directories
  - CLONE_FILES - share file descriptors
  - CLONE_SIGHAND - share signal handlers
  - CLONE_PARENT – share parent process ID
  - CLONE_THREAD – create thread for process

Creating New Threads

- The *clone* system call also uses *do_fork* to create a new task.
- The *clone* system call takes flags which are passed to *do_fork* to indicate which task attributes to copy and which to create anew.
- This system call gives applications the ability to create new processes, new threads, or new tasks that have the attributes of both processes and threads.
- *clone* is used by threads libraries to create new kernel threads.

vfork System Call

- What usually happens after a fork()?
  - execve() call to start new executable
  - Replace entire process address space
  - Then why bother duplicating?
- Enter vfork()
  - Create child with same page tables as as parent
  - Child only allowed to invoke execve()
  - Pause the parent until child invokes execve()
  - Then resume parent/child
  - Faster than fork+exec
- Implemented through clone() syscall
  - CLONE_VFORK flag needs to be set in the clone call
  - Tells clone to suspend parent until child calls execve or exit

Destroying a Task

- Tasks stop executing when they call the *exit* system call, are killed by the kernel (due to an exception), or are killed by a fatal signal which was sent.
- *exit* calls *do_exit* which decrements usage counts on the sub-structures of the *task_struct*. Any substructure with a zero usage count has its memory freed.
- Lastly, the task is changed to the *EXIT_ZOMBIE* state.
- *task_struct*s are actually destroyed by *release_task*, which is called when the process' parent calls the *wait* system call.
  - extremely difficult for a task to delete its own task structure and kernel stack.
  - also provides an easy mechanism for parents to determine their children's *exit* status.
- *release_task* removes the task from the task list and frees its memory.
- The `init` process cleans up children.

exit() Call Chain

```
libc exit (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
```

Backup slides

Pre-emptive Kernels

- Pre-emptive kernel different from process pre-emption
  - A non-preemptive kernel may not task switch while executing kernel code on behalf of a process
  - Up to Linux 2.4, implemented through BKL (big kernel lock)
  - Each syscall acquires BKL before execution
  - All other syscalls block. So, kernel code must run fast!
  - Inefficient on multicore architectures!
  - Finally removed in 2011
- Pre-emptive kernel: allow task switch while in kernel mode
  - What to do with kernel state?
  - Need per-process kernel stack!
  - What to do with interrupts?
  - Share process kernel stack (previously), or get their own (now)
  - All interrupts share single 4KB or 8KB kernel stack
- Which stack is being used determines kernel "context"

How Do I Block?

- By calling one of the sleep_on functions:
  - *sleep_on, interruptible_sleep_on, sleep_on_timeout,* etc.
- These functions create a *wait_queue* and place the calling task on it
- Modify the value of its '*state*' variable:
  - TASK_UNINTERRUPTIBLE
  - TASK_INTERRUPTIBLE
- Then call *schedule* or *schedule_timeout*
- The next task to run calls *deactivate_task* to move us out of the run queue
- Only tasks with '*state == TASK_RUNNING*' are granted time on the CPU by the scheduler

How Do I Wake Up?

- By someone calling one of the wake functions:
  - *wake_up, wake_up_all, wake_up_interruptible,* etc.
- These functions call the *curr->func* function to wake up the task
  - Defaults to *default_wake_function* which is *try_to_wake_up*
- *try_to_wake_up* calls *activate_task* to move us out of the wait queue
- The '*state*' variable is set to *TASK_RUNNING*
- Sooner or later the scheduler will run us again
- We then return from *schedule* or *schedule_timeout*

What are all these options?

- *INTERUPTIBLE* vs. *NON-INTERUPTIBLE*:
  - Can the task be woken up by a signal?
- *TIMEOUT* vs no timeout:
  - Wake up the task after some timeout interval
- *EXCLUSIVE* vs. *NON-EXCLUSIVE*:
  - Should only one task be woken up?
  - Only one *EXCLUSIVE* task is woken up
    - Kept at end of the list
  - All *NON-EXCLUSIVE* tasks are woken up
    - Kept at head of the list
  - Functions with _nr option wake up number of tasks

Other Wait Queue Notes

- Process can wakeup with event not true
  - If multiple waiters, another may have resource
  - Always check availability after wakeup
  - Maybe wakeup was in response to signal
- ‘Interruptible’ functions are preferred
- *sleep_on* functions are deprecated
  - *sleep_on* functions suffer from race conditions
  - Want to atomically test and sleep
  - *prepare_to_wait* functions preferred

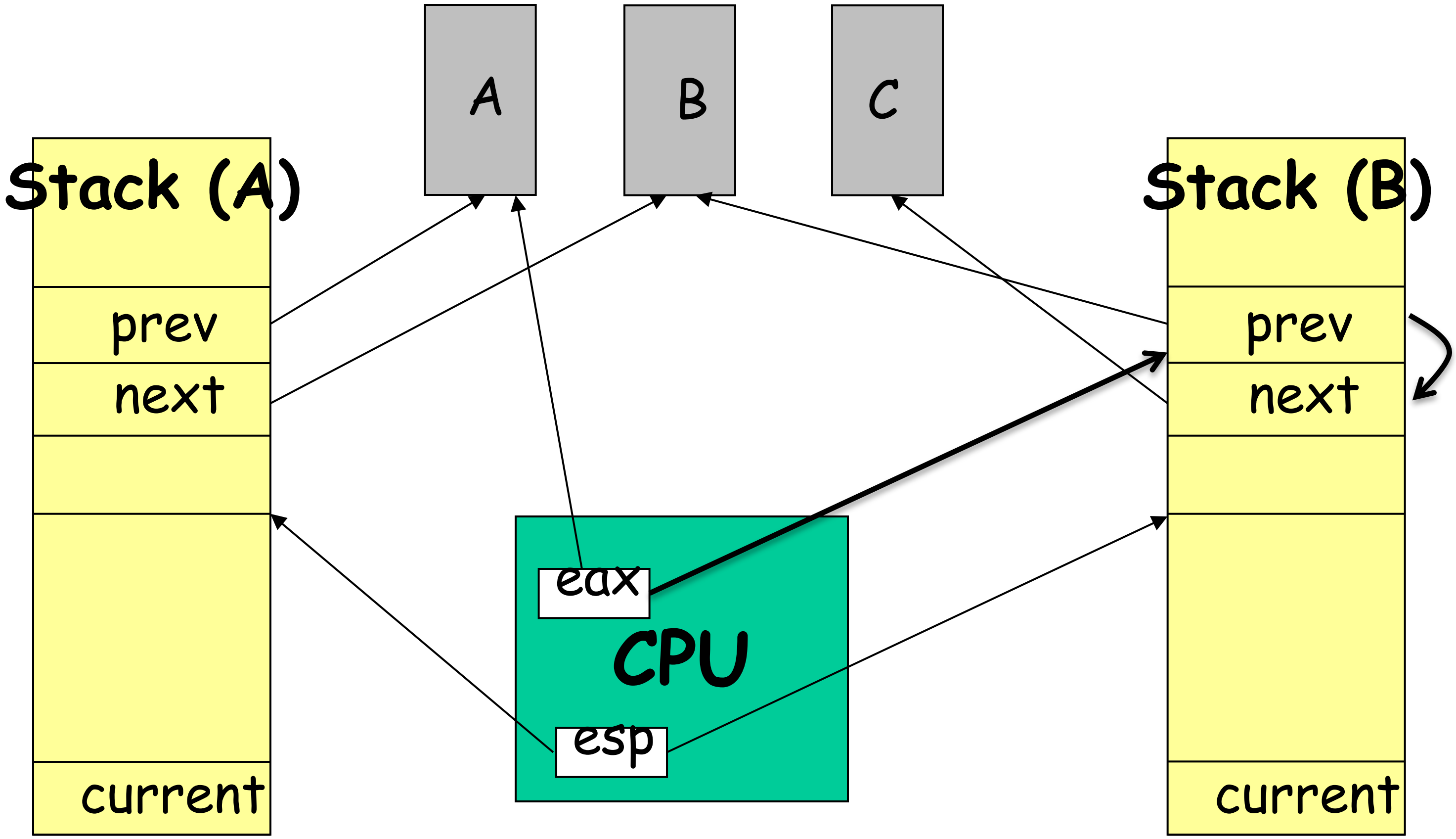
Context Switch: switch_to

- *switch_to* is architecture specific.
- Generally, it saves the old task's hardware state of the CPU (registers) to one of three places:
  - The task's kernel stack
  - the *thread_struct*
  - *task_struct->thread*
- It then copies the new task's hardware state from the appropriate places
  - Stack is in *next->thread.esp*

The Role of the Stack

- One process must save state where another can find it
- When the new state is loaded, the CPU *is* running another process -- the state *is* the process!
- The stack pointer determines most of the state
- Some of the registers are on the stack
- The stack pointer determines the location of *thread_info,* which also points to *task struct*
- *Changing the stack pointer changes the process!*

Stack Switching



- switch_to: A -> B

Context Switch: FP Registers

- Many CPU architectures support lazy saving of floating point state (registers) by allowing floating point capability to be disabled, resulting in an exception when a floating point operation is performed.
- With this capability, state save can detect when a thread first uses floating point and only save floating point state from then on. It can also only load floating point state after a floating point operation following a context switch.

Context Switch: FP Registers

- On context switch:
  - Hardware flag set: *TS* in *cr0*
  - Software flag *TS_USEDFPU* is cleared in *task_struct*
- If task uses floating point instruction and hardware flag is set:
  - Hardware raises "device not available" exception (trap)
  - Kernel restores floating point registers
  - TS is cleared
  - *TS_USEDFPU* is set in the *task_struct* for this process
- Any time it's set, floating point registers are saved for that process at switch time (but not restored for the next)
- Bottom line: only done if needed; if only one process uses floating point, no save/restore needed

- Not needed on modern processors! More efficient FPU.

Threads

- Threads in a process are represented by creating a *task_struct* for each thread in the process and keeping most of the data the same for each *task_struct*.
- ultimately done by using *do_fork*
- simplifies some algorithms because there is only one structure for both processes and threads.
- can improve performance for single threaded processes.
- Process data is generally in task sub-structures which can be shared by all tasks in the process.

Thread Structures

- The thread state is represented by the *thread_info* structure.
- The *thread_info* structure has a reference to the *task_struct* for the thread as well as the execution domain for the program the thread is executing within.
- The *thread_info* structure and the thread's kernel stack are located together within a *thread_union* structure. size varies by architecture
- thread's stack thus also varies by architecture
  - just less than 4K in size on 32-bit architectures
  - just less than 8K in size on 64-bit architectures.

Kernel Threads

- Linux has a small number of kernel threads that run continuously in the kernel (daemons)
  - No user address space
  - Only execute code and access data in kernel address space
- How to create: *kernel_thread*
- Scheduled in the same way as other threads/tasks
- Process 0: idle process
- Process 1: init process
  - Spawns several kernel threads before transitioning to user mode as /sbin/init
  - kflushd (bdflush) – Flush dirty buffers to disk under "memory pressure"
  - kupdate – Periodically flushes old buffers to disk
  - kswapd – Swapping daemon

Task Zero

- The task with process ID zero is called the swapper or the idle task
- Its task structure is in *init_thread_union*, which also includes its kernel stack.
- The kernel builds this task piece by piece to use to boot the system. (All other tasks are copied from an existing task by *do_fork*.)
- All other tasks are maintained in a linked list off of this task.
- This task becomes the idle task that runs when no other task is runnable.
- This task forks the init task (task 1) and is the ancestor of all other tasks.

Task Zero

- On SMP systems, this task uses *clone* to create duplicate tasks which run as the idle task on each of the other processors.
- All of these tasks have process ID zero.
- Each of these tasks is used only by its associated processor.