ECE391
Computer System Engineering
Lecture 18

Dr. Jian Huang

University of Illinois at Urbana- Champaign

Fall 2018

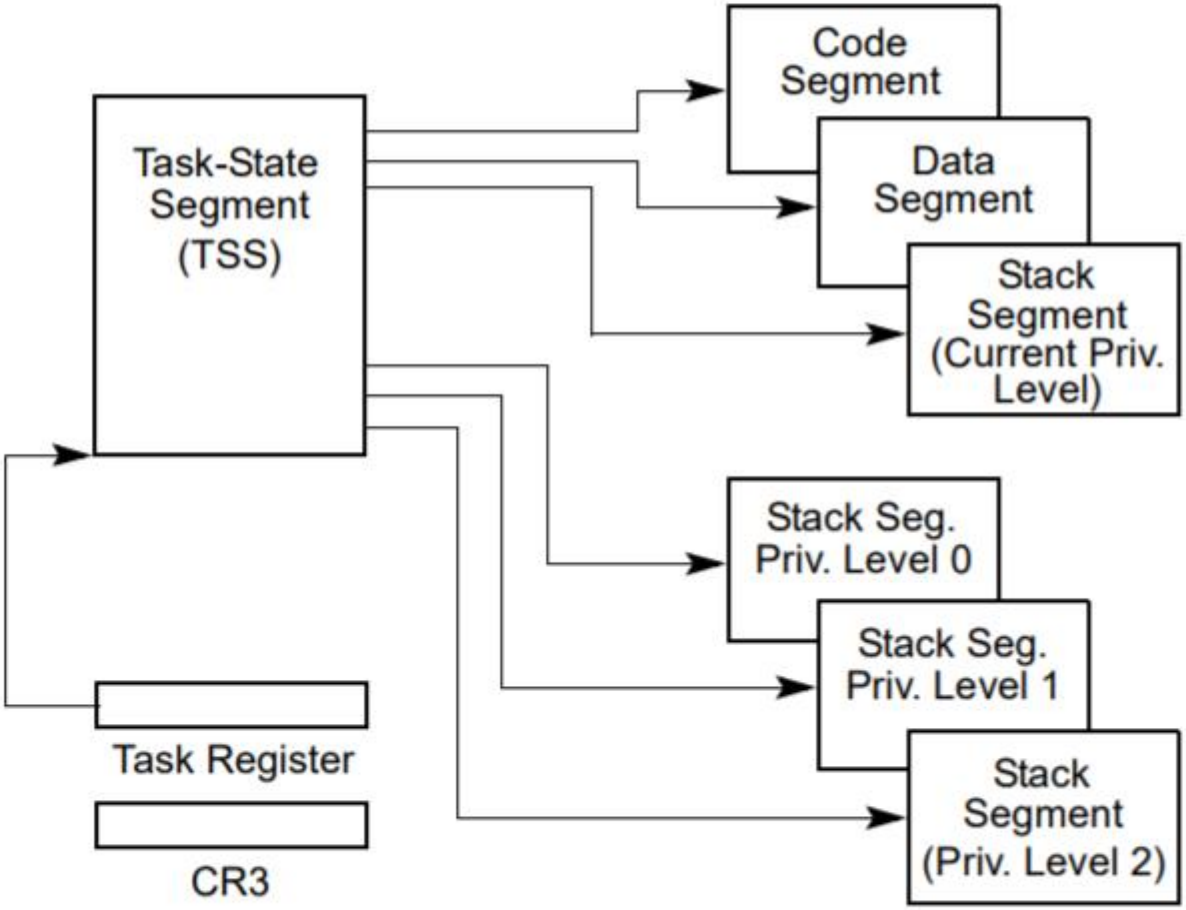
x86 Task State Segment

31	15	0	
I/O Map Base Address		T	100
	LDT Segment Selector		96
	GS		92
	FS		88
	DS		84
	SS		80
	CS		76
	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
	SS2		24
ESP2			20
	SS1		16
ESP1			12
	SS0		8
ESP0			4
	Previous Task Link		0

Reserved bits. Set to 0.

Figure 6-2. 32-Bit Task-State Segment (TSS)

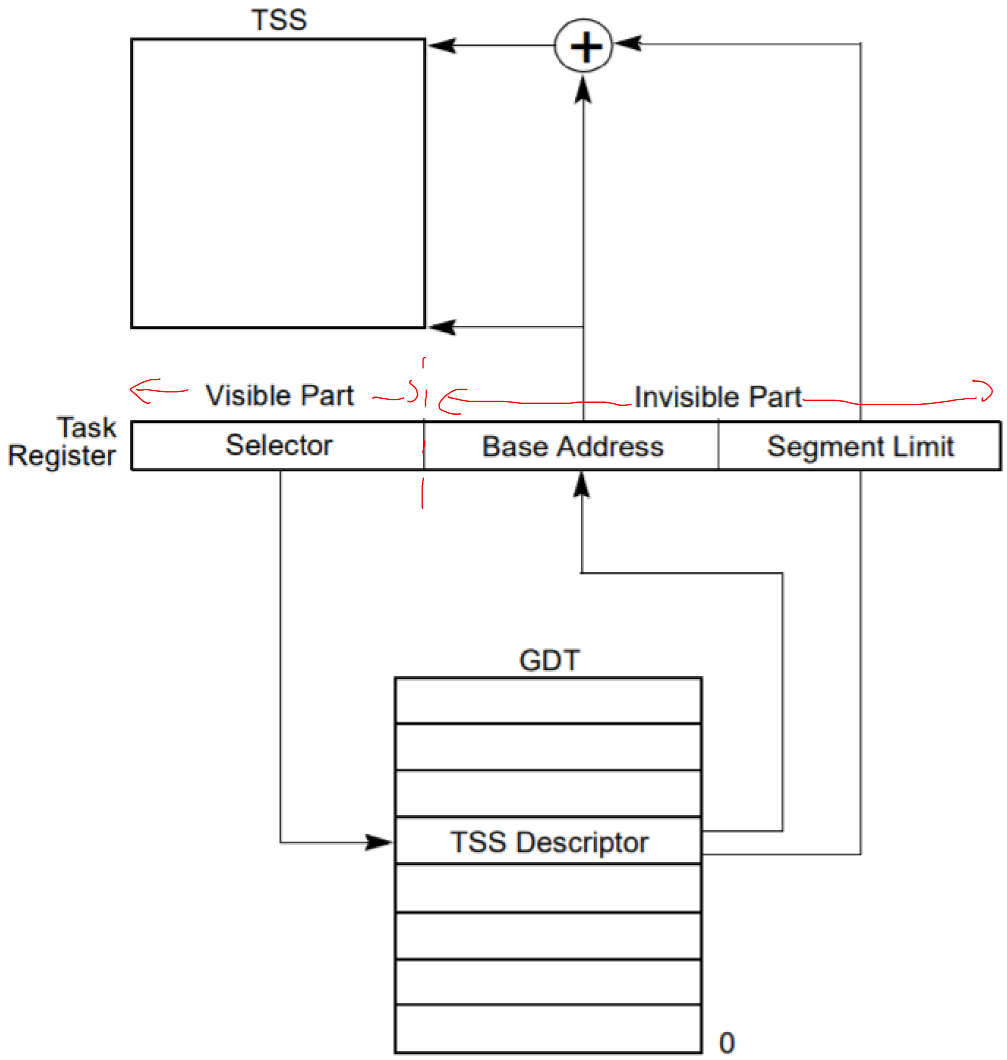
Purposes of Task State Segment



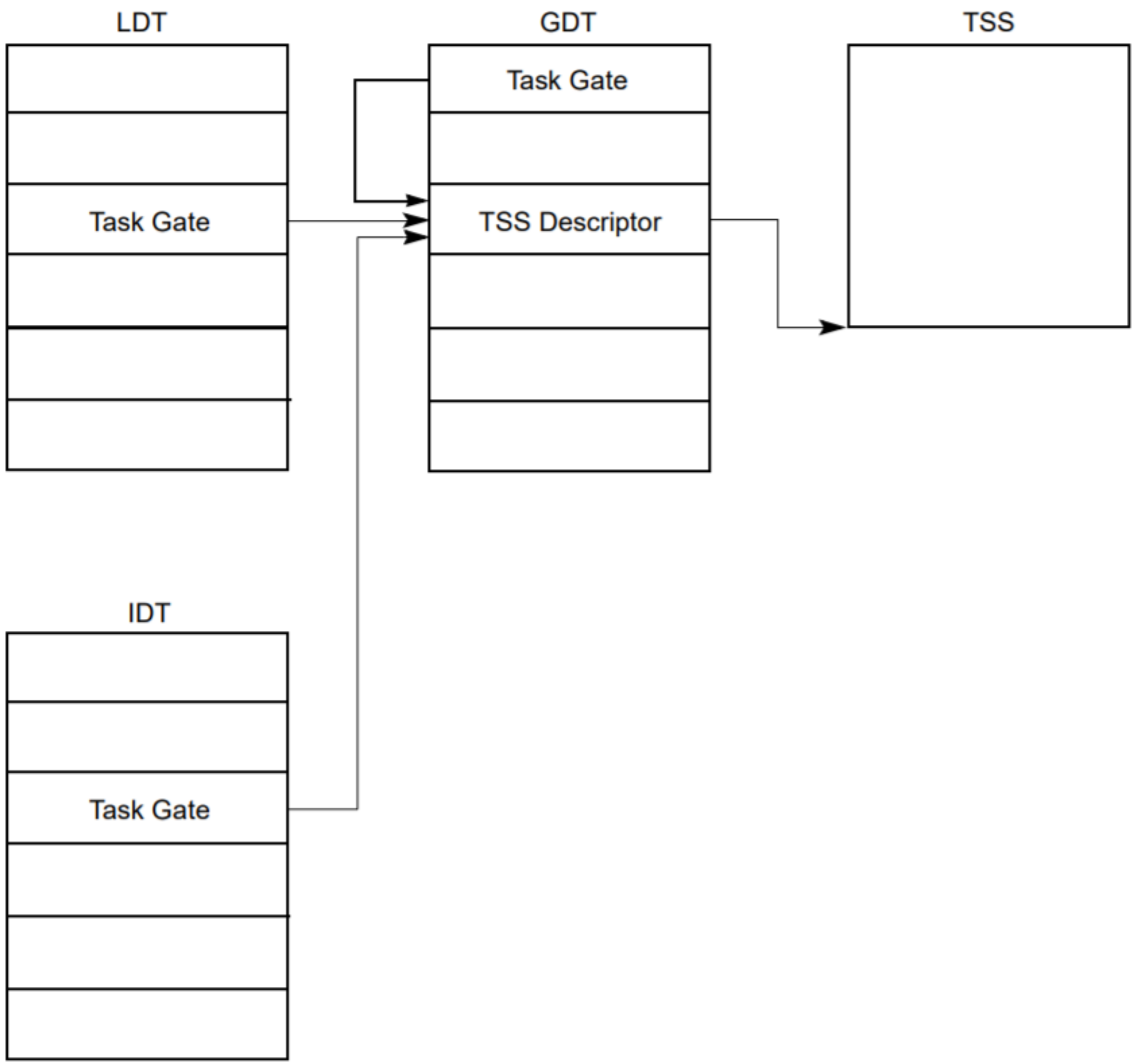
How to Execute a Task?

- A explicit call to a task with the CALL instruction
- A explicit jump to a task with the JMP instruction
- An implicit call to an interrupt-handler task
- An implicit call to an exception-handler task
- A return when the NT (nested task) flag in the EFLAGS register is set

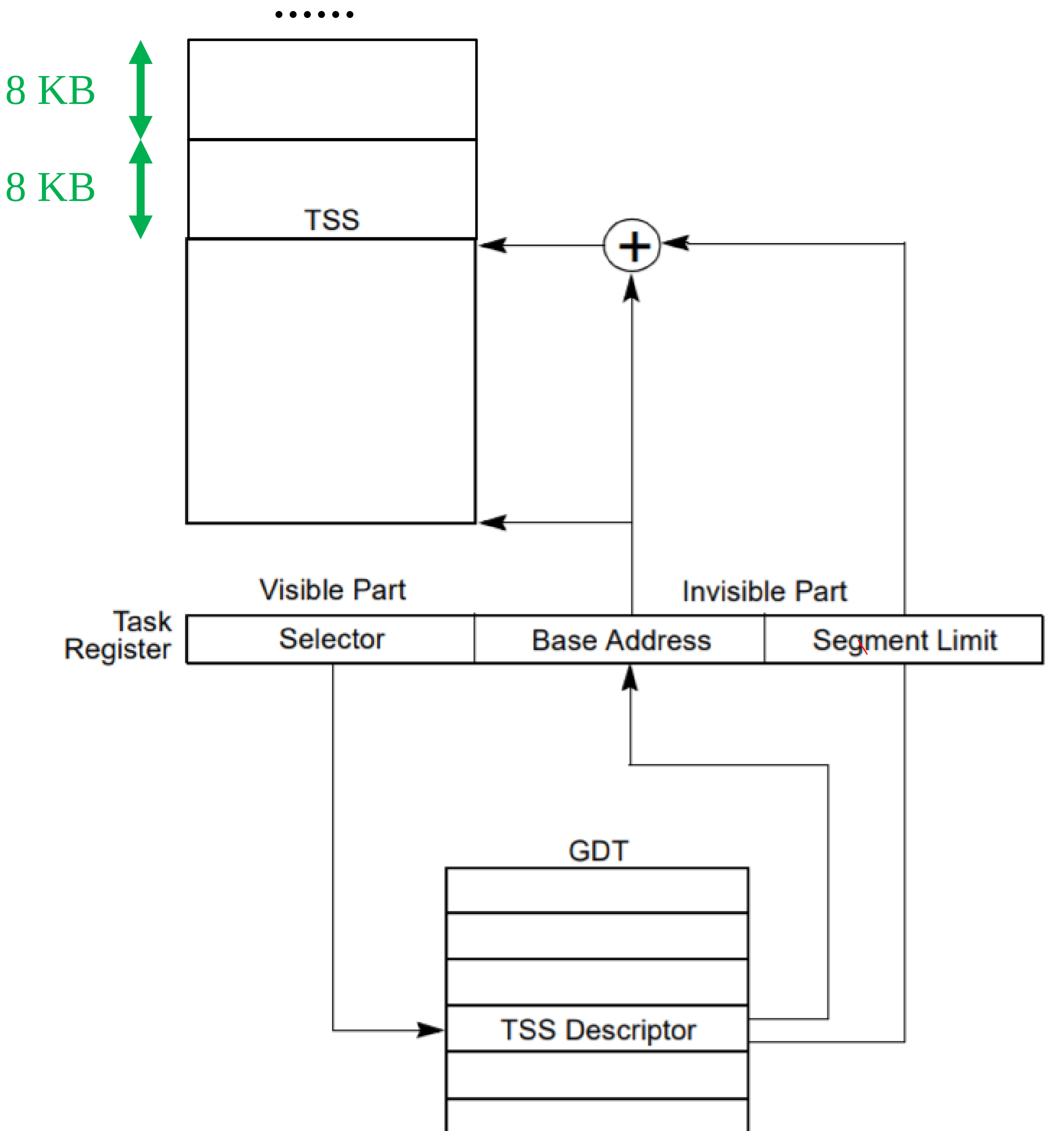
How to Switch a Task?



How to Switch a Task? (cont.)



About Task Management in MP3

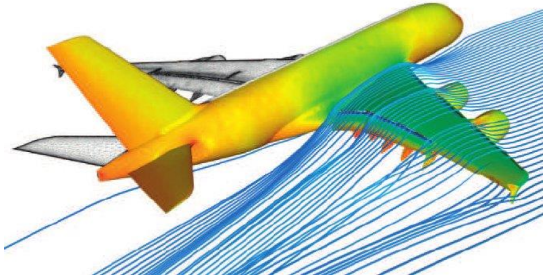


Lecture Topics

- Scheduling
  - Task Types
  - Philosophy
  - Algorithm
  - Data structures

Task Types

- Several types of jobs exist
  - Interactive
    - examples: editors, GUIs
    - driven by human interaction (e.g., keystrokes, mouse clicks)
    - important to respond quickly
  - Batch
    - examples: compilation, simulation
    - usually only time to completion matters
    - want a fair share of CPU computation



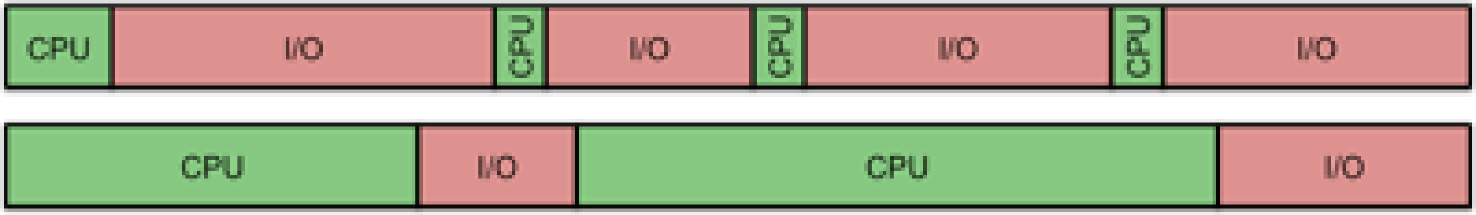
Task Types (cont.)

– Real-time
  - examples: music, video, teleconferencing
  - periodic deadlines (e.g., 30 frames per second of video)
  - work often only useful if finished on time

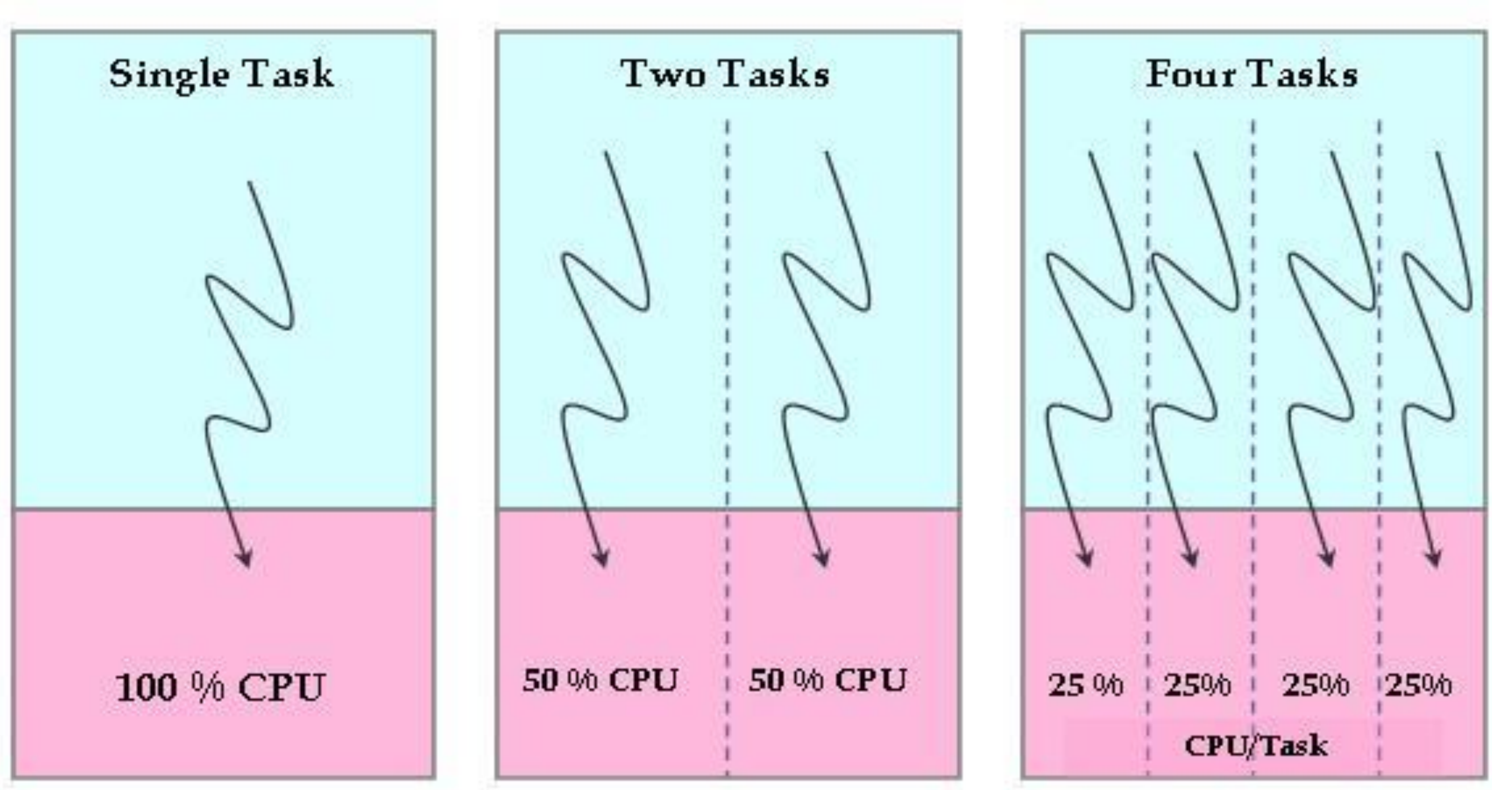
– alternative taxonomy: I/O-bound vs. compute-bound

I/O bounded

Compute bounded



Goals of Task Scheduling



[Source: stackoverflow]

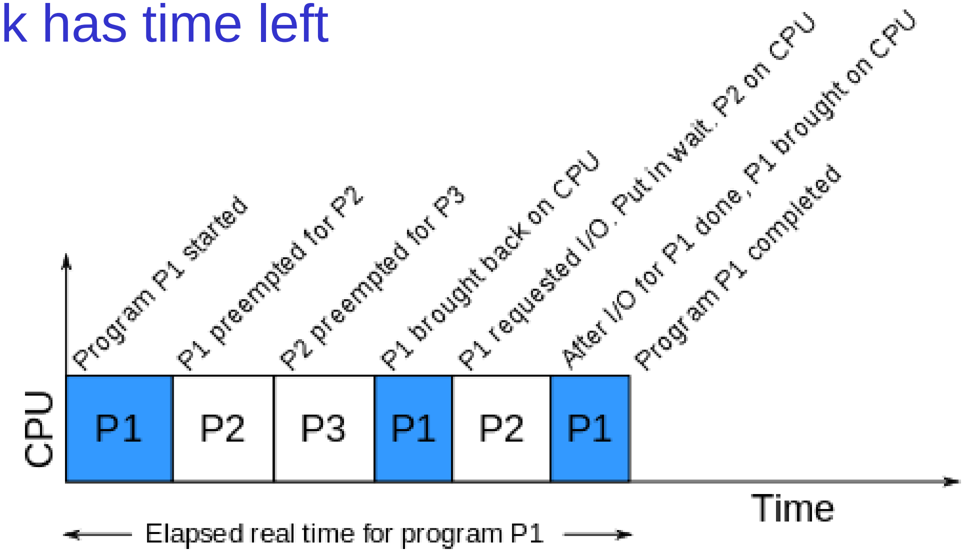
Goals: Efficient, Fairness, & Responsive

Scheduling Philosophy

- General strategy
  - break time into slices
  - allow interactive jobs to preempt the current job based on interrupts
    - typically, a job becomes runnable when an interrupt occurs (e.g., a key is pressed)
    - Linux checks for rescheduling after each interrupt, system call, and exception

General Scheduling Algorithm Used by Linux

- Time broken into epochs
  - each task given a quantum of time (in ticks of 10 milliseconds)
  - run until no runnable task has time left
  - then start a new epoch



- Real-time jobs
  - always given priority over non-real-time jobs
  - prioritized amongst themselves
- Static and dynamic priorities used for non-real-time jobs

General Scheduling Algorithm Used by Linux (cont.)

- Interactive jobs handled with heuristics
  - heuristic estimate job interactiveness
  - an interactive job
    - can continue to run after running out of time
    - takes turns with other interactive jobs
  - philosophy is that they don't usually use up quantum
  - heuristic ensures that job can't use lots of CPU and still be "interactive"

Task State (fields of ***task_struct****)*

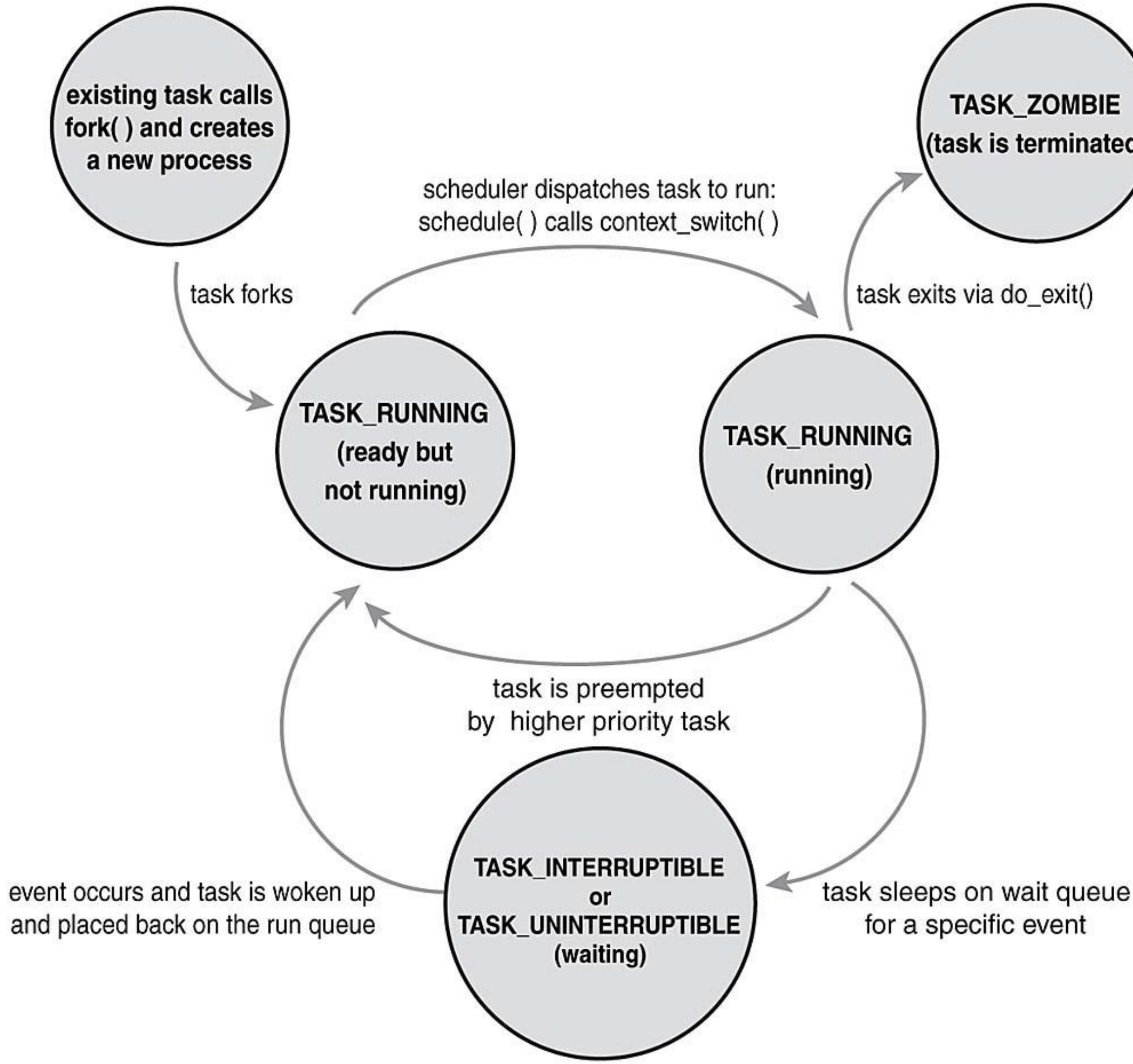
- State field can be one of
  - TASK_RUNNING
    - task is executing currently or waiting to execute
    - task is in a run queue on some processor
  - TASK_INTERRUPTIBLE
    - task is sleeping on a semaphore/condition/signal
    - task is in a wait queue
    - can be made runnable by delivery of signal
  - TASK_UNINTERRUPTIBLE
    - task is busy with something that can't be stopped (e.g., atomic operation)
    - e.g., a device that will stay in unrecoverable state without further task interaction
    - cannot be made runnable by delivery of signal

*Task State (fields of **task_t**) (cont.)*

- TASK_STOPPED
  - task is stopped
  - task is not in a queue; must be woken by signal

- TASK_ZOMBIE
  - task has terminated
  - task state retained until parent collects exit status information
  - task is not in a queue

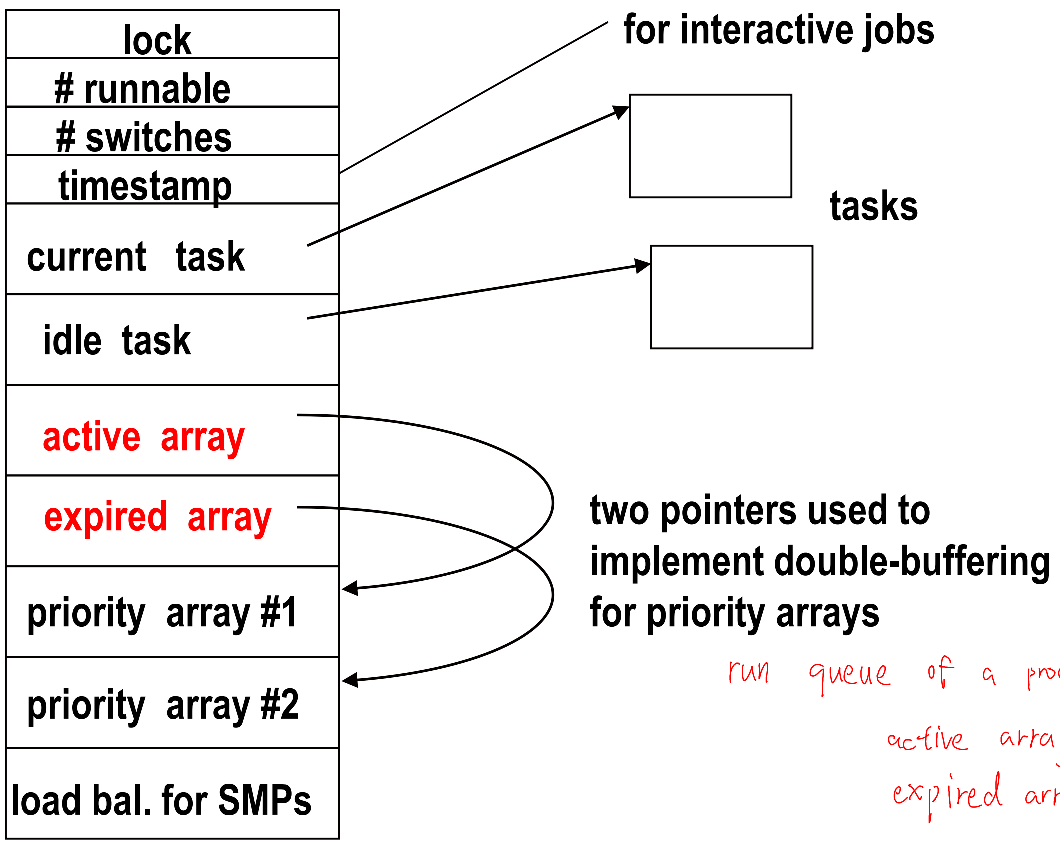
Task State Transition



Task stoped ?

where ?

Scheduling Data Structures



run queue of a processor:

active array

expired array

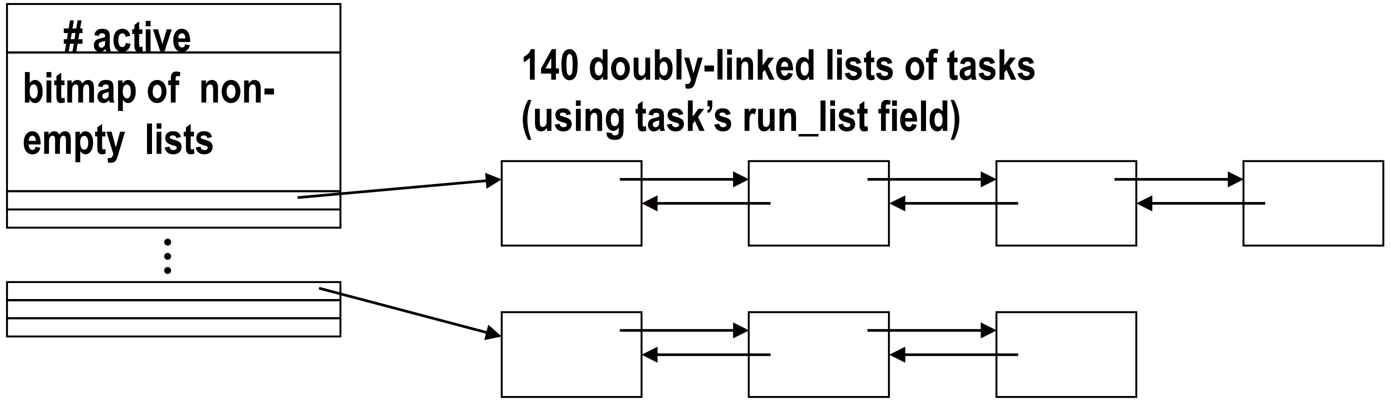
- Each processor has a run queue (struct runqueue in sched.c)
  - each run queue has two priority arrays
  - arrays are lists of tasks of each priority
  - they are double-buffered to implement epochs

Data Structure of Task Queues

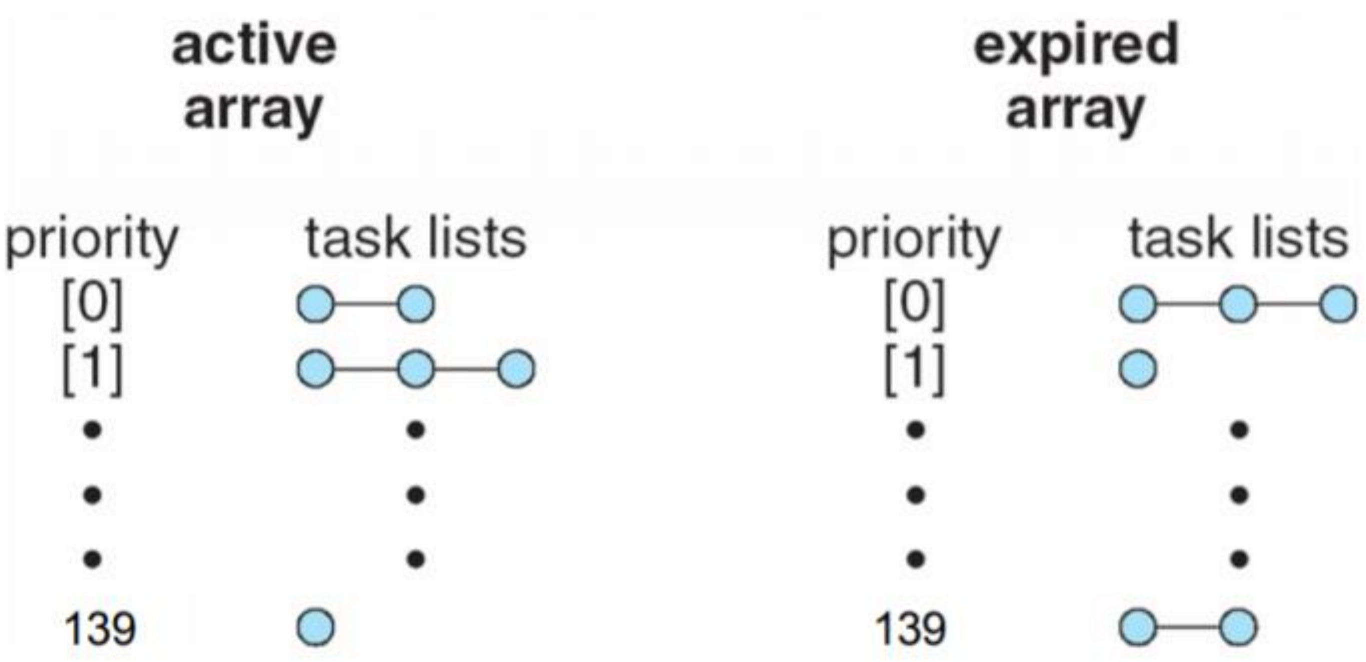
```
struct runqueue {
        struct prioarray   *active;
        struct prioarray   *expired;
        struct prioarray    arrays[2];
};

struct prioarray {
        int     nr_active;  /* Runnable */
        unsigned long      bitmap[5];
        struct list_head   queue[140];
}
```

Priority Array Structure
(struct prio_array in sched.c)

- 100 real-time priorities
- 40 regular priorities
- One list per priority and a bitmap to make finding non-empty list fast



Active and Expired Array



[Source: http://lobello.dieei.unict.it/]

Swapping Arrays

```
struct prioarray   *array = rq->active;

if (array->nr_acive == 0) {

        rq->active = rq->expired;
        rq->expired = array;

}
```


Rescheduling and Yielding

- Task can change (called a context switch) when
  - current task yields by calling schedule (sched_yield at user level)
  - current task runs out of time
- Other places where a task may yield implicitly
  - semaphores
  - wake_up_process
  - copy_to/from_user
- At every timer tick (interrupt, generated every 10 milliseconds)
  - run a function (scheduler_tick) to reduce current task's time
  - executes with IF=0 (interrupts will be ignored)
  - replace it with another task if appropriate
  - interrupt is IRQ0 (system timer)

Linux Scheduler

heduler_tick and task_running_tick (kernel/sched.c)

```
void scheduler_tick (void)
{
    unsigned long long now = sched_clock ();
    struct task_struct* p  = current;
    int cpu                = smp_processor_id ();
    int idle_at_tick       = idle_cpu (cpu);
    struct rq* rq          = cpu_rq (cpu);

    update_cpu_clock (p, rq, now);

    if (!idle_at_tick)
        task_running_tick (rq, p);
}
```

idle_cpu returns 1 if CPU is running the idle task

update_cpu_clock tracks nanoseconds since last tick/context switch with time stamp counter (TSC)

for all but idle task, call task_running_tick

```
static void task_running_tick (struct rq* rq, struct task_struct* p)
{
    if (p->array != rq->active) {
        set_tsk_need_resched (p);
        return;
    }
    spin_lock (&rq->lock);
    if (rt_task (p)) {
        if ((p->policy == SCHED_RR) && !--p->time_slice) {
            p->time_slice = task_timeslice (p);
            p->first_time_slice = 0;
            set_tsk_need_resched (p);
            requeue_task (p, rq->active);
        }
        goto out_unlock;
    }
    if (!--p->time_slice) {
        dequeue_task (p, rq->active);
        set_tsk_need_resched (p);
        p->prio = effective_prio (p);
        p->time_slice = task_timeslice (p);
        p->first_time_slice = 0;

        if (!rq->expired_timestamp)
            rq->expired_timestamp = jiffies;
        if (!TASK_INTERACTIVE (p) || expired_starving (rq)) {
            enqueue_task (p, rq->expired);
            if (p->static_prio < rq->best_expired_prio)
                rq->best_expired_prio = p->static_prio;
        } else
            enqueue_task (p, rq->active);
    } else {
        /* Prevent long timeslices that allow a task to monopolize the
         * CPU by splitting up the timeslice into smaller pieces.  We
         * requeue this task to the end of the list on this priority
         * level, which is a round-robin of tasks with equal priority. */
        if (TASK_INTERACTIVE (p) &&
            !((task_timeslice (p) - p->time_slice) % TIMESLICE_GRANULARITY (p)) &&
            (p->time_slice >= TIMESLICE_GRANULARITY (p)) && (p->array == rq->active)) {

            requeue_task (p, rq->active);
            set_tsk_need_resched (p);
        }
    }
out_unlock:
    spin_unlock (&rq->lock);
}
```

Take care of task which already expired

Critical section begins

Handle real-time tasks

if a task's time slice expires

dequeue and reschedule the task

breaks long time slices into pieces

Critical section ends

Comments on `scheduler_tick` *function*

- `idle_cpu` returns 1 if CPU is running the idle task
- `update_cpu_clock` tracks nanoseconds since last tick/context switch
- For all but idle task, call `task_running_tick`

Comments on `task_running_tick` function

- If the task has already expired
  - make sure that it gets rescheduled on return from interrupt
  - by setting **`TIF_NEED_RESCHED`**

- The remainder is a critical section for the run queue

- Next handle real-time tasks
  - real-time round-robin tasks take turns by placing themselves at the end of the list of their priority
  - otherwise real-time tasks just keep rescheduling (until yield or preemption by higher priority)

Comments on `task_running_tick` *function*

- If a task's time slice expires
  - take it out of the run queue
  - mark it as needing to be removed from processor
  - give it a new time slice
  - if it's an interactive job, and expired tasks are not being starved by interactive ones, put it back into run queue (at end)
  - normal tasks go into expired queue
- Last block
  - breaks long time slices into pieces
  - round-robin between tasks at same priority

Linux Scheduler (Part1)

schedule function (kernel/sched.c)

```
asmlinkage void schedule (void)
{
    task_t* prev    = current;
    task_t* next;
    runqueue_t* rq = this_rq ();
    prio_array_t* array;
    list_t* queue;
    int idx;

    if (in_interrupt ())
        BUG();
    release_kernel_lock (prev, smp_processor_id ());
    prev->sleep_timestamp = jiffies;
    spin_lock_irq (&rq->lock);

    switch (prev->state) {
        case TASK_INTERRUPTIBLE:
            if (signal_pending (prev)) {
                prev->state = TASK_RUNNING;
                break;
            }
        default:
            deactivate_task (prev, rq);
        case TASK_RUNNING:
            ;
    }
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        goto switch_tasks;
    }

    array = rq->active;
    if (!array->nr_active) {
        /*
         * Switch the active and expired arrays.
         */
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired_timestamp = 0;
    }

    idx = sched_find_first_bit (array->bitmap);
    queue = array->queue + idx;
    next = list_entry (queue->next, task_t, run_list);
```

Sleep timestamp records time at which task leaves CPU

Critical section begins

if the task is trying to go to sleep check for receipt of signal

default case removes task from run queue

no tasks are runnable

nothing is left in the active run queue

find the next task to run
- find first no-zero bit in the bitmask of the active set
- take the first task at the list indicataed by that bit

*Comments on **schedule** function*

- *Schedule()* function called when processor needs to change to new task
  - time has expired for current task, or
  - task has voluntarily yielded (by calling this function)
- Sleep timestamp records time at which task leaves CPU
- Remainder is run queue critical section
- If the task is trying to go to sleep
  - check for receipt of signal
  - handles race condition with sleeping in wait queue
- Default case removes task from run queue

Comments on ***schedule*** *function (cont.)*

- If no tasks are runnable
  - run the idle task
  - reset forced expiration of interactive tasks
- If nothing is left in the active run queue
  - the epoch is over
  - swap the priority arrays
  - and reset forced expiration of interactive tasks
- Find the next task to run
  - find first bit selects the priority
  - then take the first task at that priority (linked list)

Linux Scheduler (Part2)

schedule function (kernel/sched.c)

```
switch_tasks:
        prefetch (next);
        prev->need_resched = 0;

        if (prev != next) {
            rq->nr_switches++;
            rq->curr = next;
            context_switch (prev, next);
            /*
             * The runqueue pointer might be from another CPU
             * if the new task was last running on a different
             * CPU - thus re-load it.
             */
            barrier ();
            rq = this_rq ();
        }
        spin_unlock_irq (&rq->lock);

        reacquire_kernel_lock (current);
        return;
}
```

- clear need_resched flag for next time current task is scheduled
- Switch if newly selected task is not the current one
- call architecture-dependent switch function
- Critical section ends

Comments on schedule *function (cont.)*

- This portion of the code implements the actual switch
- First clears need_resched flag for next time current task is scheduled
- Switch only occurs if newly selected task is not the current one
- Switch does two things
  - does some accounting
  - calls architecture-dependent switch function (context_switch)