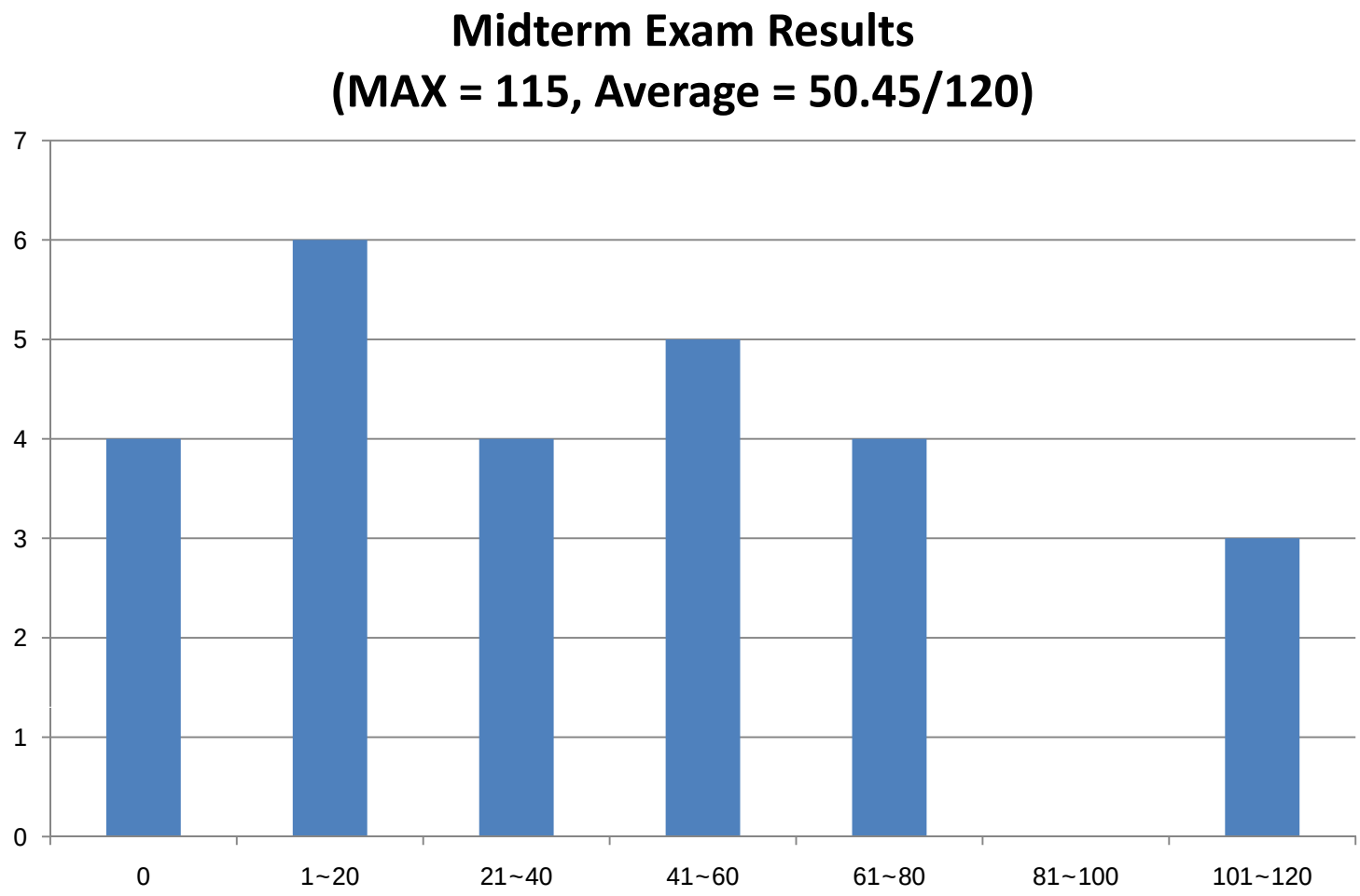
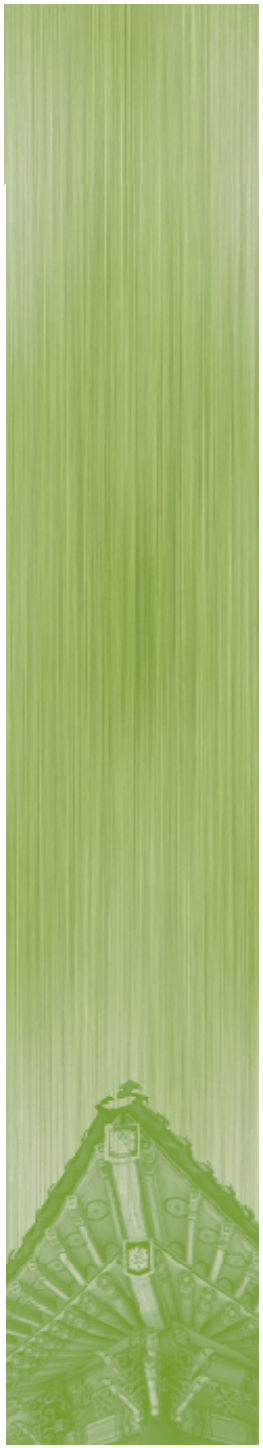
Midterm Exam





CPU Scheduling

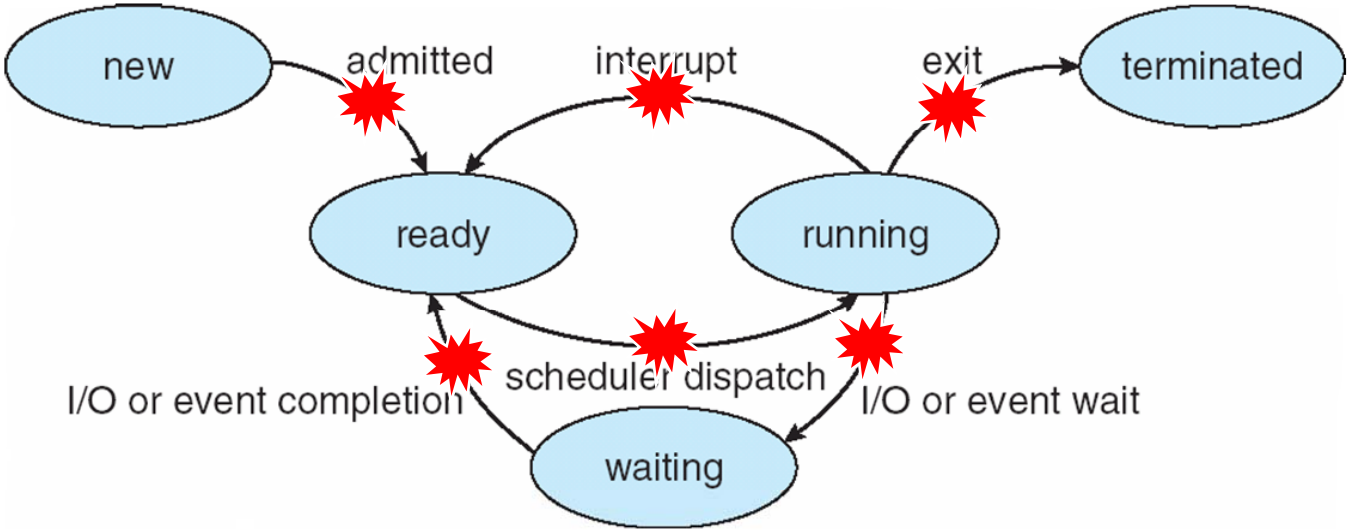
Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Today's Topics

- General scheduling concepts
- Scheduling algorithms
- Case studies
  - Linux

CPU Scheduling (1)

- **CPU scheduling**
  - Deciding which process to run next, given a set of runnable processes.
  - Happens frequently, hence should be fast.
- **Scheduling points**



CPU Scheduling (2)

- **Scheduling algorithm goals**
  - All systems
    - No starvation
    - Fairness: giving each process a fair share of the CPU
    - Balance: keeping all parts of the system busy
  - Batch systems
    - Throughput: maximize jobs per hour
    - Turnaround time: minimize time between submission and termination
    - CPU utilization: keep the CPU busy all the time
  - Interactive systems
    - Response time: respond to requests quickly
    - Proportionality: meet users' expectations
  - Real-time systems
    - Meeting deadlines: avoid losing data
    - Predictability: avoid quality degradation in multimedia system

CPU Scheduling (3)

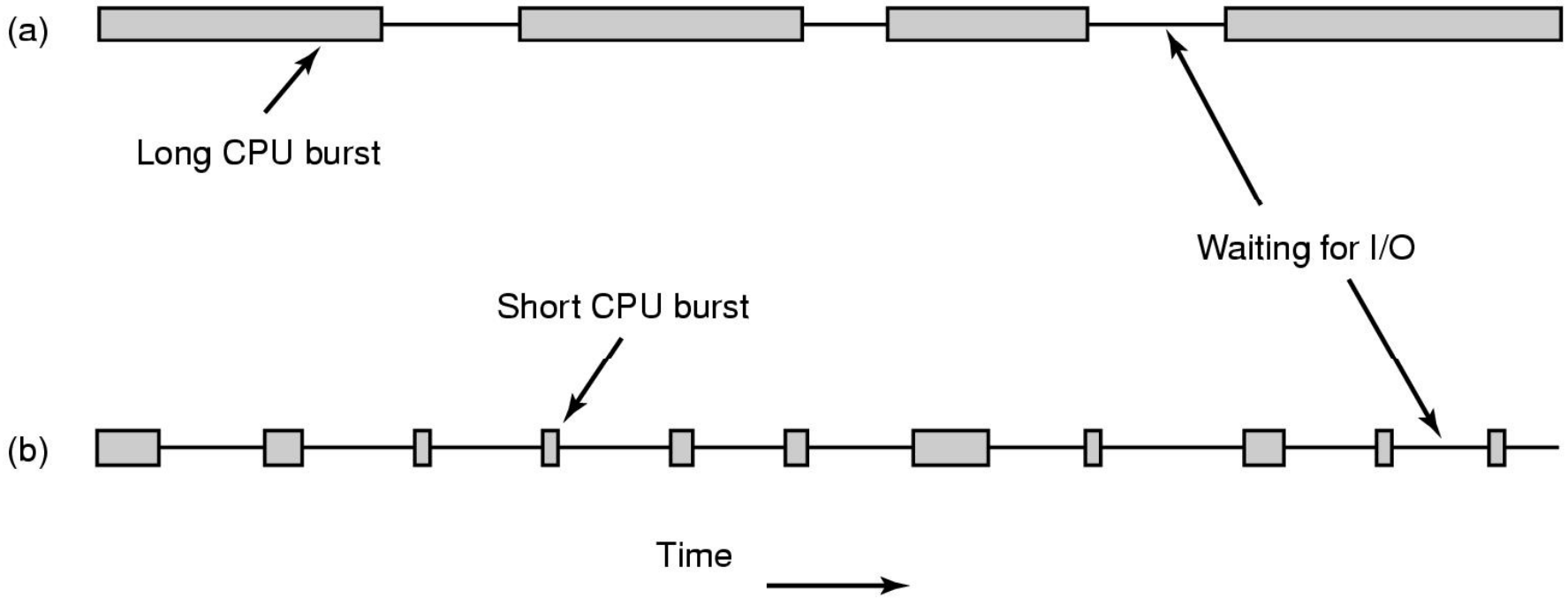
- **Starvation**
  - A situation where a process is prevented from making progress because another process has the resource it requires.
    - Resource could be the CPU or a lock.
  - A poor scheduling policy can cause starvation
    - If a high-priority process always prevents a low-priority process from running on the CPU.
  - Synchronization can also cause starvation
    - One thread always beats another when acquiring a lock.
    - Constant supply of readers always blocks out writers.

CPU Scheduling (4)

- **Non-preemptive scheduling**
  - The scheduler waits for the running job to voluntarily yield the CPU.
  - Jobs should be cooperative.
- **Preemptive scheduling**
  - The scheduler can interrupt a job and force a context switch.
  - What happens
    - If a process is preempted in the midst of updating the shared data?
    - If a process in a system call is preempted?

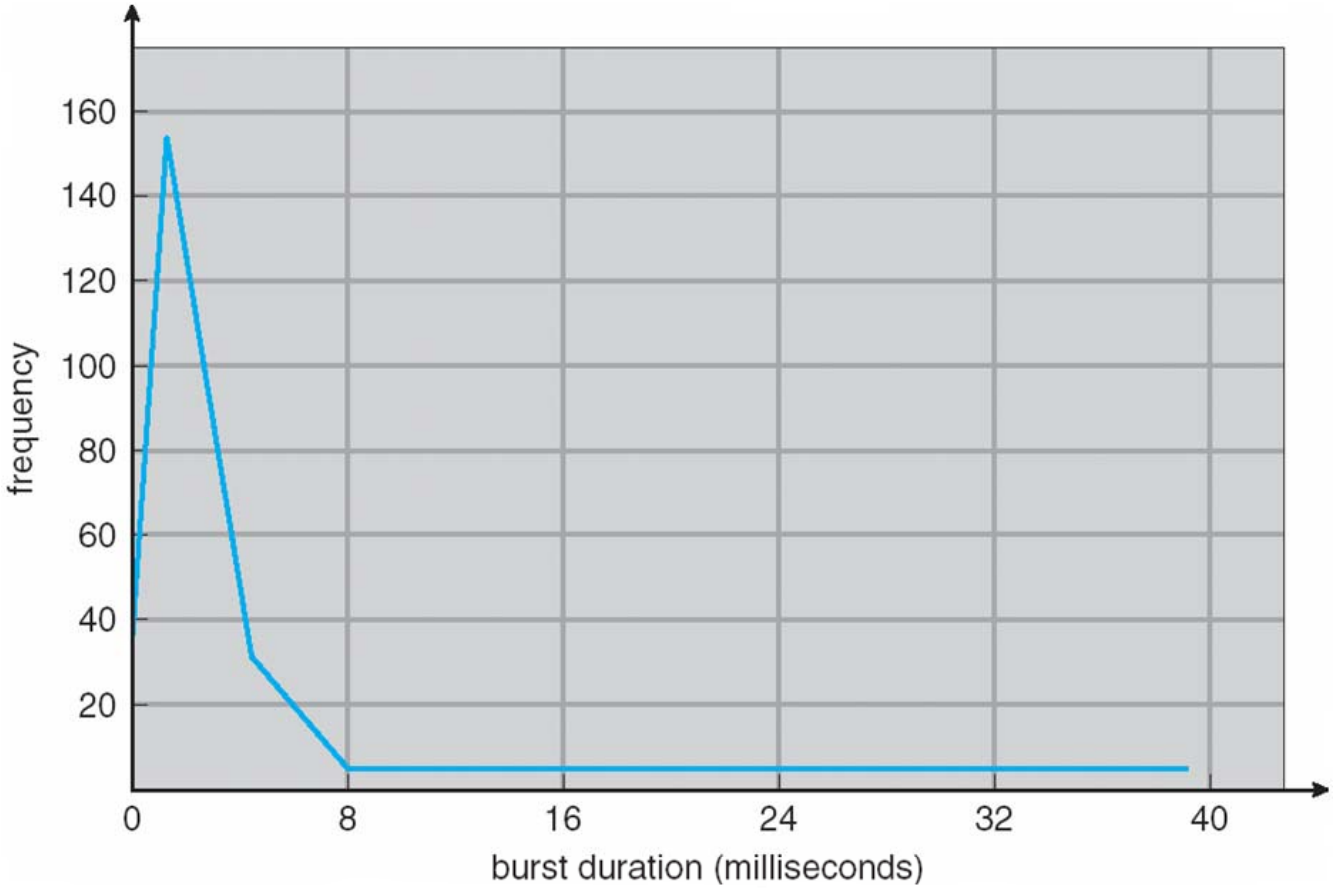
Execution Characteristics (1)

- **CPU burst vs. I/O burst**
  - A CPU-bound process
  - An I/O-bound process



Execution Characteristics (2)

- **Histogram of CPU-burst Times**



FCFS/FIFO

- **First-Come, First-Served**
  - Jobs are scheduled in order that they arrive.
  - "Real-world" scheduling of people in lines
    - e.g., supermarket, bank tellers, McDonalds, etc.
  - Typically, non-preemptive
  - Jobs are treated equally: no starvation.
- **Problems**
  - Average waiting time can be large if small jobs wait behind long ones.
    - Basket vs. cart
  - May lead to poor overlap of I/O and CPU.

SJF

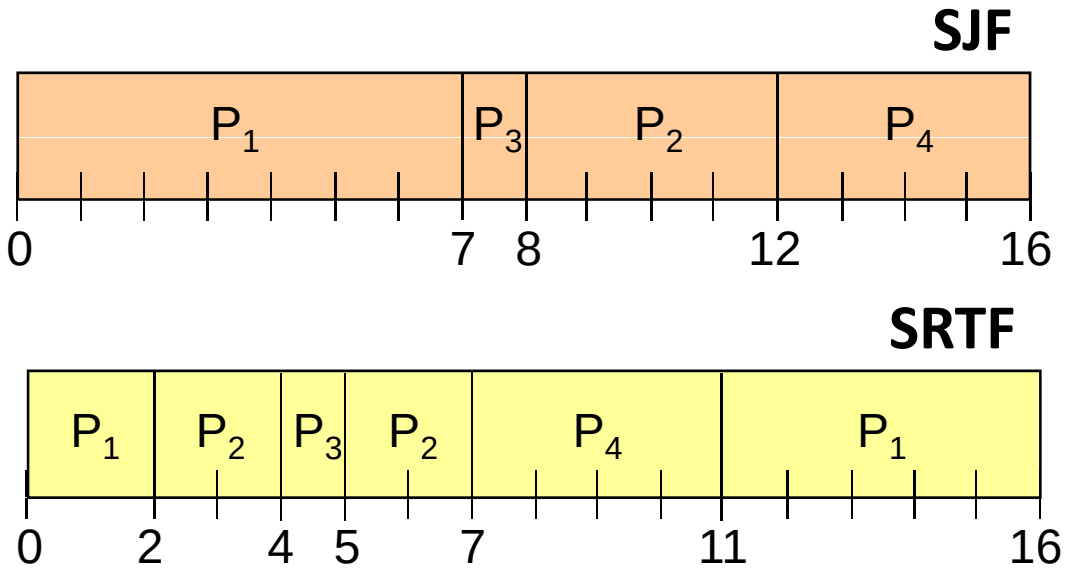
- **Shortest Job First**
  - Choose the job with the smallest expected CPU burst.
  - Can prove that SJF has optimal min. average waiting time.
    - Only when all jobs are available simultaneously.
    - e.g., A(2), B(4), C(1), D(1), E(1) at 0,0,3,3,3
  - Non-preemptive

- **Problems**
  - Impossible to know the size of future CPU burst.
  - Can you make a reasonable guess?
  - Can potentially starve.

SRTF

- **Shortest Remaining Time First**
  - Preemptive version of SJF.
  - If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.

Process	Arrival Time	Burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



RR

- **Round Robin**
  - Ready Q is treated as a circular FIFO Q.
  - Each job is given a time slice (or time quantum).
    - Usually 10-100 ms.
  - Great for timesharing
    - No starvation
    - Typically, higher average turnaround time than SJF, but better response time.
  - Preemptive
  - What do you set the quantum to be?
    - A rule of thumb: 80% of the CPU bursts should be shorter than the time quantum.
  - Treats all jobs equally

Priority Scheduling (1)

- **Priority scheduling**
  - Choose job with highest priority to run next
  - SJF = Priority scheduling, where
    priority = expected length of CPU burst
  - Round-robin or FIFO within the same priority
  - Can be either preemptive or non-preemptive
  - Priority is dynamically adjusted.
  - Modeled as a Multi-level Feedback Queue (MLFQ)

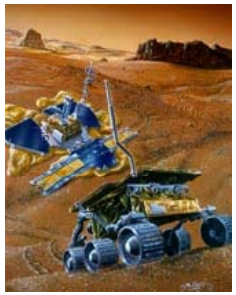
Priority Scheduling (2)

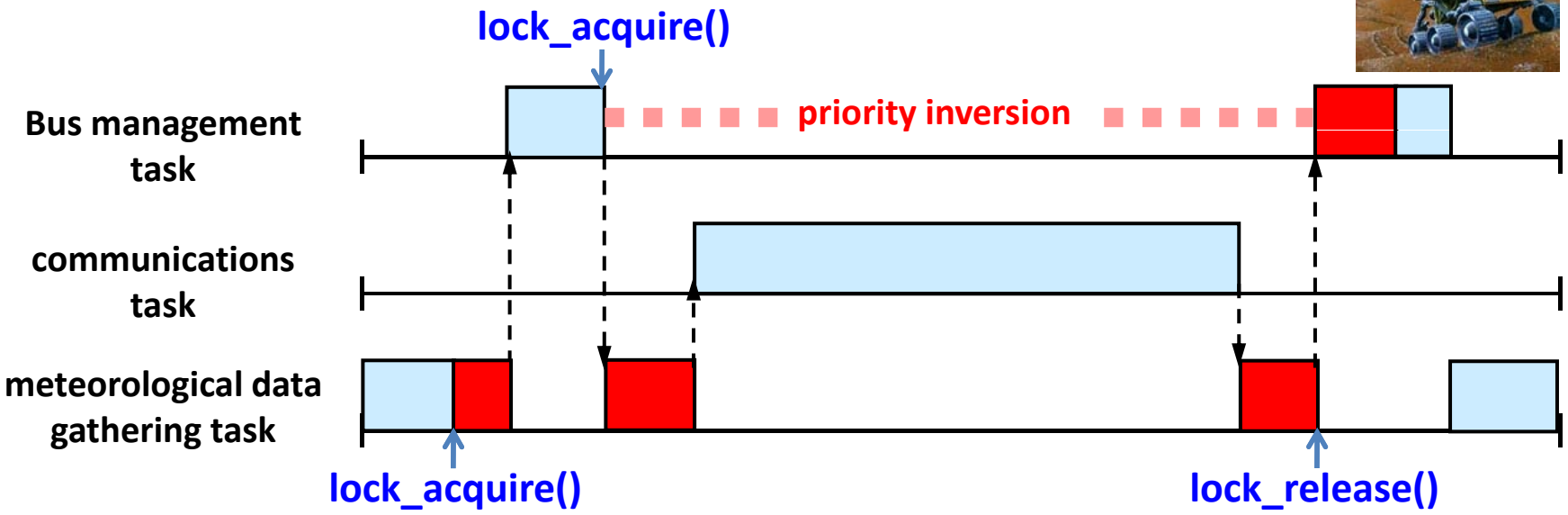
- **Starvation problem**
  - If there is an endless supply of high priority jobs, no low priority job will ever run.

- **Solution: Aging**
  - Increase priority as a function of wait time.
  - Decrease priority as a function of CPU time.
  - Many ugly heuristics have been explored in this area.

Priority Scheduling (3)

- **Priority inversion problem**
  - A situation where a higher-priority job is unable to run because a lower-priority job is holding a resource it needs, such as a lock.
  - *What really happened on Mars?*





Priority Scheduling (4)

- **Priority inheritance protocol (PIP)**
  - The higher-priority job can donate its priority to the lower-priority job holding the resource it requires.

- **Priority ceiling protocol (PCP)**
  - The priority of the low-priority thread is raised immediately when it gets the resource.
  - The priority ceiling value must be predetermined.

Priority Scheduling (5)

- **Multilevel Feedback Queue**
  - Multilevel queue scheduling, which allows a job to move between the various queues.
  - Queues have priorities.
    - Batch, interactive, system, CPU-bound, I/O-bound, …
  - When a process uses too much CPU time, move to a lower-priority queue.
    - Leaves I/O-bound and interactive processes in the higher-priority queues.
  - When a process waits too long in a lower priority queue, move to a higher-priority queue.
    - Prevents starvation.

UNIX Scheduler (1)

- **Characteristics**
  - Preemptive
  - Priority-based
    - The process with the highest priority always runs.
    - 3 – 4 classes spanning ~170 priority levels (Solaris 2)
  - Time-shared
    - Based on timeslice (or quantum)
  - MLFQ (Multi-Level Feedback Queue)
    - Priority scheduling across queues, RR within a queue.
    - Processes dynamically change priority.

UNIX Scheduler (2)

- **General principles**
  - Favor I/O-bound processes over CPU-bound processes
    - I/O-bound processes typically run using short CPU bursts.
    - Provide good interactive response; don't want editor to wait until CPU hog finishes quantum.
    - CPU-bound processes should not be severely affected.
  - No starvation
    - Use aging
  - Priority inversion?

Linux 2.4 Scheduling (1)

- **General characteristics**
  - Linux offers three scheduling algorithms.
    - A traditional UNIX scheduler: SCHED_OTHER
    - Two "real-time" schedulers (mandated by POSIX.1b): SCHED_FIFO and SCHED_RR
  - Linux scheduling algorithms for real-time processes are "soft real-time".
    - They give the CPU to a real-time process if any real-time process wants it.
    - Otherwise they let CPU time trickle down to non real-time processes.
  - Here, we study the scheduling algorithm implemented in the Linux 2.4.18 kernel.

Linux 2.4 Scheduling (2)

- **Priorities**
  - Static priority
    - The maximum size of the time slice a process should be allowed before being forced to allow other processes to complete for the CPU.
  - Dynamic priority
    - The amount of time remaining in this time slice; declines with time as long as the process has the CPU.
    - When its dynamic priority falls to 0, the process is marked for rescheduling.
  - Real-time priority
    - Only real-time processes have the real-time priority.
    - Higher real-time priority values always beat lower values.

Linux 2.4 Scheduling (3)

- **Related fields in the task structure**

`long counter;`	time remaining in the task's current quantum (represents dynamic priority)
`long nice;`	task's nice value, -20 to +19. (represents static priority)
`unsigned long policy;`	SCHED_OTHER, SCHED_FIFO, SCHED_RR
`struct mm_struct *mm;`	points to the memory descriptor
`int processor;`	processor ID on which the task will execute
`unsigned long cpus_runnable;`	~0 if the task is not running on any CPU (1<<cpu) if it's running on a CPU
`unsigned long cpus_allowed;`	CPUs allowed to run
`struct list_head run_list;`	head of the run queue
`unsigned long rt_priority;`	real-time priority

Linux 2.4 Scheduling (4)

- **Scheduling policies**
  - SCHED_OTHER
  - SCHED_FIFO
    - A real-time process runs until it either blocks on I/O, explicitly yields the CPU, or is preempted by another real-time process with a higher rt_priority.
    - Acts as if it has no time slice.
  - SCHED_RR
    - It's the same as SCHED_FIFO, except that time slices do matter.
    - When a SCHED_RR process's time slice expires, it goes to the back of the list of SCHED_FIFO and SCHED_RR processes with the same rt_priority.

Linux 2.4 Scheduling (5)

- **Scheduling quanta**
  - Linux gets a timer interrupt or a *tick* once every 10ms on IA-32. (HZ=100)
    - Alpha port of the Linux kernel issues 1024 timer interrupts per second.
  - Linux wants the time slice to be around 50ms.
    - Decreased from 200ms (in v2.2)

```
/* v2.4 */
#if HZ < 200
#define TICK_SCALE(x)    ((x) >> 2)
#endif
#define NICE_TO_TICKS(nice)    (TICK_SCALE(20-(nice))+1)

/* v2.2 */
#define DEF_PRIORITY     (20*HZ/100)
```

Linux 2.4 Scheduling (6)

- **Epochs**
  - The Linux scheduling algorithm works by dividing the CPU time into epochs.
    - In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins.
    - The epoch ends when all runnable processes have exhausted their quantum.
    - The scheduler recomputes the time-quantum durations of all processes and a new epoch begins.
  - The base time quantum of a process is computed based on the nice value.

Linux 2.4 Scheduling (7)

- **Selecting the next process to run**

```
repeat_schedule:
    next = idle_task(this_cpu);
    c = -1000;
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
            if (can_schedule(p, this_cpu)) {
                int weight = goodness(p, this_cpu,
                                      prev->active_mm);
                if (weight > c)
                    c = weight, next = p;
                }
        }
```

Linux 2.4 Scheduling (8)

- **Recalculating counters**

```
if (unlikely(!c)) {          /* New epoch begins … */
    struct task_struct *p;

    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task(p)
        p->counter = (p->counter >> 1) +
                     NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
```

Linux 2.4 Scheduling (9)

- **Calculating goodness()**

```
static inline int goodness (p, this_cpu, this_mm) {
    int weight = -1;
    if (p->policy == SCHED_OTHER) {
        weight = p->counter;
        if (!weight) goto out;
        if (p->mm == this_mm || !p->mm)
            weight += 1;
        weight += 20 – p->nice;
        goto out;
    }
    weight = 1000 + p->rt_priority;
out:    return weight;
}
```

`weight = 0`
p has exhausted its quantum.

`0 < weight < 1000`
p is a conventional process.

`weight >= 1000`
p is a real-time process.

Linux 2.4 Scheduling (10)

- **Linux scheduler is not so scalable!**
  - A single run queue is protected by a run queue lock.
    - As the number of processors increases, the lock contention increases.
  - It is expensive to recalculate `goodness()` for every task on every invocation of the scheduler.
    - A profile of the kernel taken during the VolanoMark runs shows that 37-55% of total time spent in the kernel is spent in the scheduler.
    - The VolanoMark benchmark establishes a socket connection to a chat server for each simulated chat room user. For a 5 to 25-room simulation, the kernel must potentially deal with 400 to 2000 threads.