CSCI-GA.2250-001

Operating Systems

Process/Thread Scheduling

Hubertus Franke
frankeh@cs.nyu.edu

Scheduling

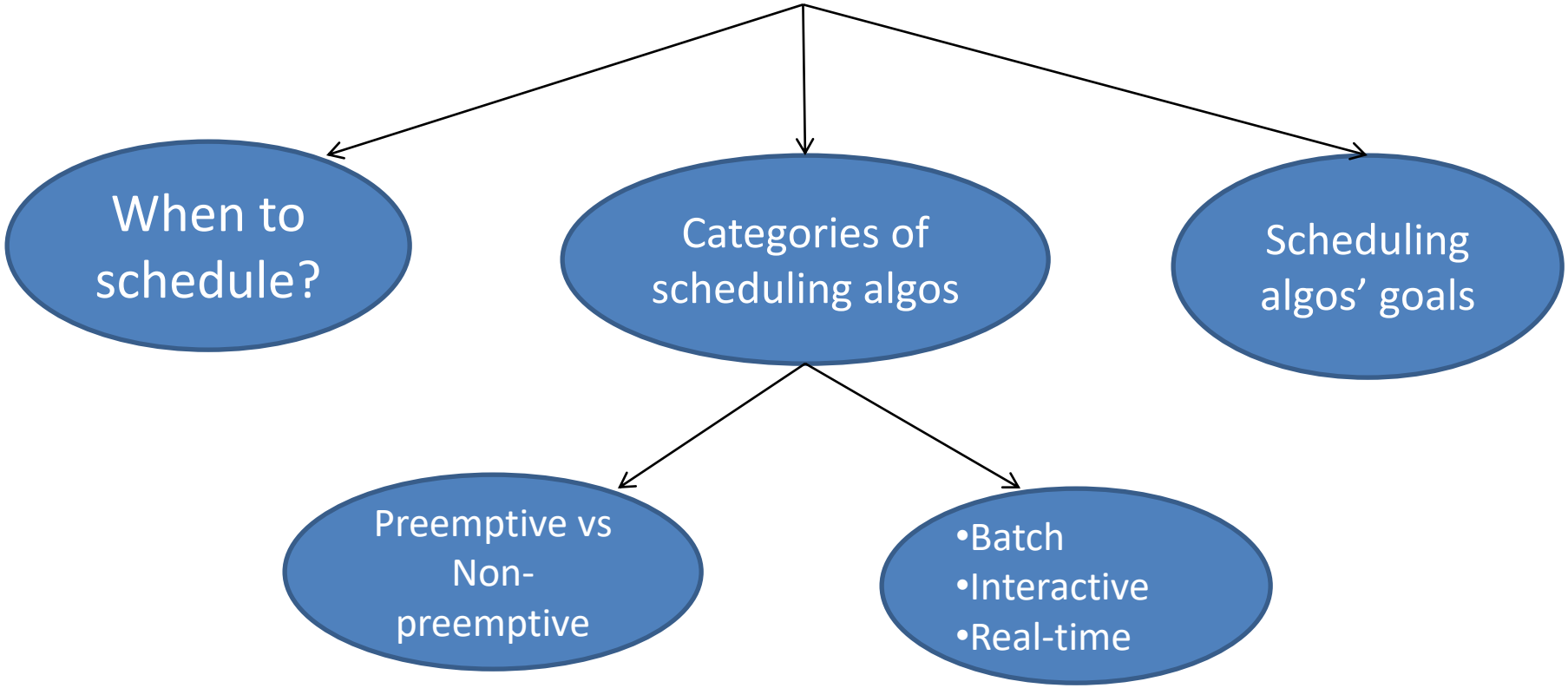
- Whether scheduling is based on processes or threads depends on whether the OS is multi-threading capable.

- In this deck/lab2 we use *process* as the assumption, just be aware that it applies to threads in a multi-threading capable OS.

- Given a group of ready processes or threads, which process/thread to run?

Scheduling

Given a group of ready processes, which process to run?



When to Schedule?

- When a process is created
- When a process exits
- When a process blocks
- When an I/O interrupt occurs

Categories of Scheduling Algorithms

- Interactive
  - preemption is essential

    preemption == a means for the OS to take away the CPU from a currently running process/thread

- Batch
  - No user impatiently waiting
  - mostly non-preemptive, or preemptive with long period for each process

- Real-time
  - deadlines

Scheduling Algorithm: Goals and Measures

- Turn Around Time (Batch)
- Throughput (e.g. jobs per second)
- Response Time (Interactive)
- Average wait times (how long waiting in readyQ)

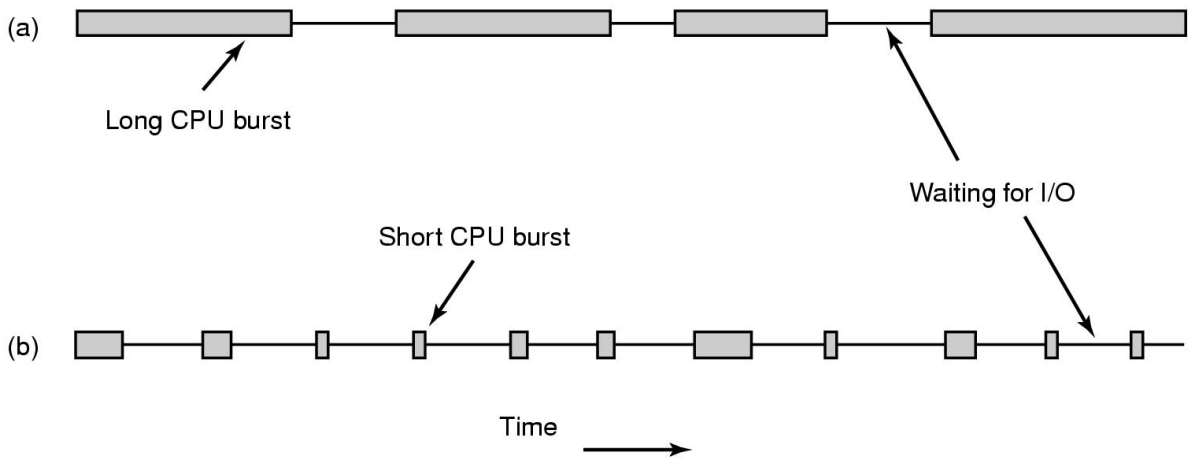
CPU / IO Burst

- CPU Burst:
  a sequence of instructions a process runs without requesting I/O.
  Mostly dependent on the program behavior.

- IO "Burst":
  time required to satisfy an IO request while the Process can not run any code.
  Mostly dependent on system behavior
  (how many other IOs, speed of device, etc.)

- NOTE: these implicitely due to program and system behaviors, NOT something that a user specifies !!

Scheduling – Process Behavior



CPU-Burst and IO-Burst are simple runtime behaviors of applications, they continuously change

Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

Trick Question: what is the cpu burst of the following program?

```
int main() {
        int i=0;
        for (;;) i++;
        return i;
}
```

Scheduling Algorithms Goals

All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- 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

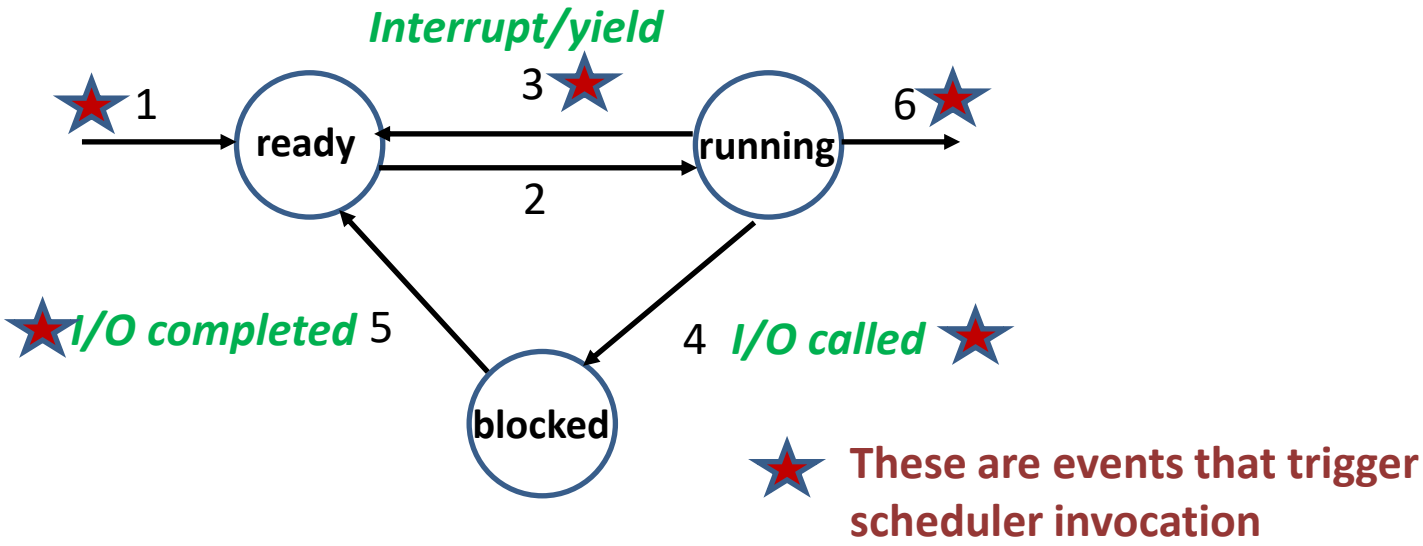
Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Almost ALL scheduling algorithms can be described by the following process state transition diagram or a derivative of it (we covered some more sophisticated one in prior lecture)



The OS Scheduler maintains a data structure called "RunQueue" or "ReadyQueue" where all processes that are <ready to run> are maintained
(doesn't have to be strictly a queue, think "pool")

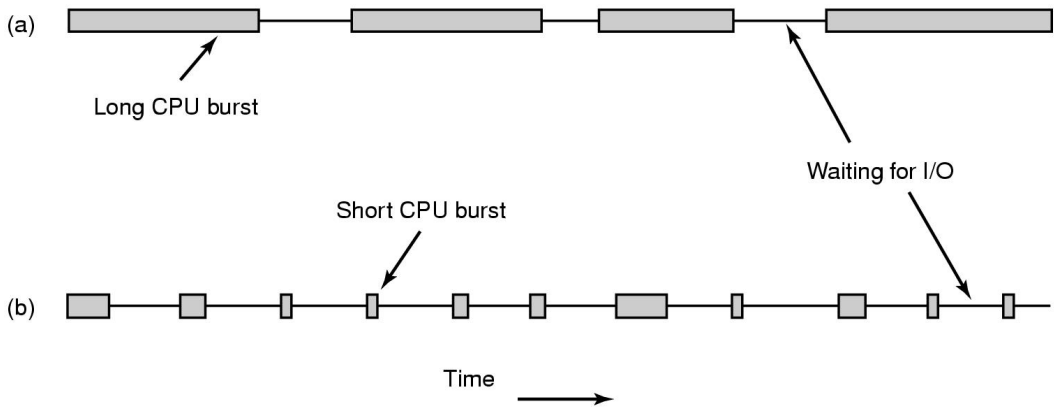
Scheduling :

The Simplest Algorithm of them All

- Random() !!!
- Simply maintain a pool and pick one.
- Things to consider:
  - Starvation
  - Fairness

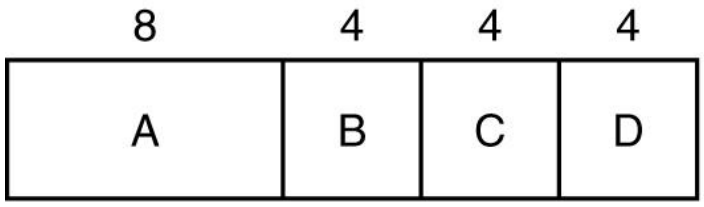
Scheduling in Batch Systems: First-Come First-Served (FIFO / FCFS)

- Non-preemptive (run till I/O or exit)
- Processes ordered as queue
- A new process is added to the end of the queue
- A blocked process that becomes ready added to the end of the queue
- Main disadv: Can hurt I/O bound processes or processes with frequent I/O



Scheduling in Batch Systems: Shortest Job First

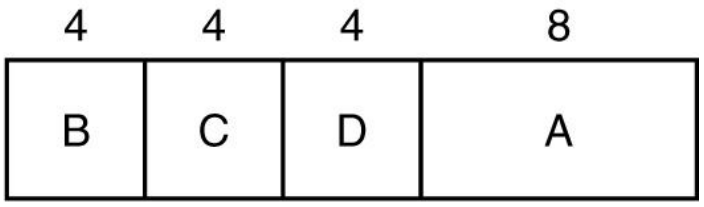
- Non-preemptive
- Assumes runtime is known in advance (☹)
- Is only optimal when all the jobs are available simultaneously



(a)

Run in original order

Avg-waittime:
(0+8+12+16) / 4 = 9



(b)

Run in shortest job first

Avg-waittime:
(0+4+8+12) / 4 = 6

Scheduling in Batch Systems: Shortest Remaining Time Next

- Scheduler always chooses the process whose remaining time is the shortest
- Runtime has to be known in advance (☹)
- Preemptive (see next slide) or non-preemptive (wait till process blocks or is done)
- Note: in lab2 we are using the non-preemptive version of "shortest remaining time next" since we have preemption in other schedulers already.
- This typically reduces average turn-around time

Scheduling in Interactive Systems: Round-Robin

- Each process is assigned a time interval referred to as quantum
- After this quantum, the CPU is given to another process (i.e. CPU is removed from the process/thread aka preemption)
- RR = FIFO + preemption
  or (FIFO = RR with huge quantum) [hint → lab2]
- What is the length of this quantum?
  - too short -> too many process context switches
    --> lower CPU efficiency
  - too long --> poor response to short interactive
  - quantum longer than typical CPU burst is good (why?)

Round-Robin Scheduling (cont)

- Promotes Fairness (due to preemption)

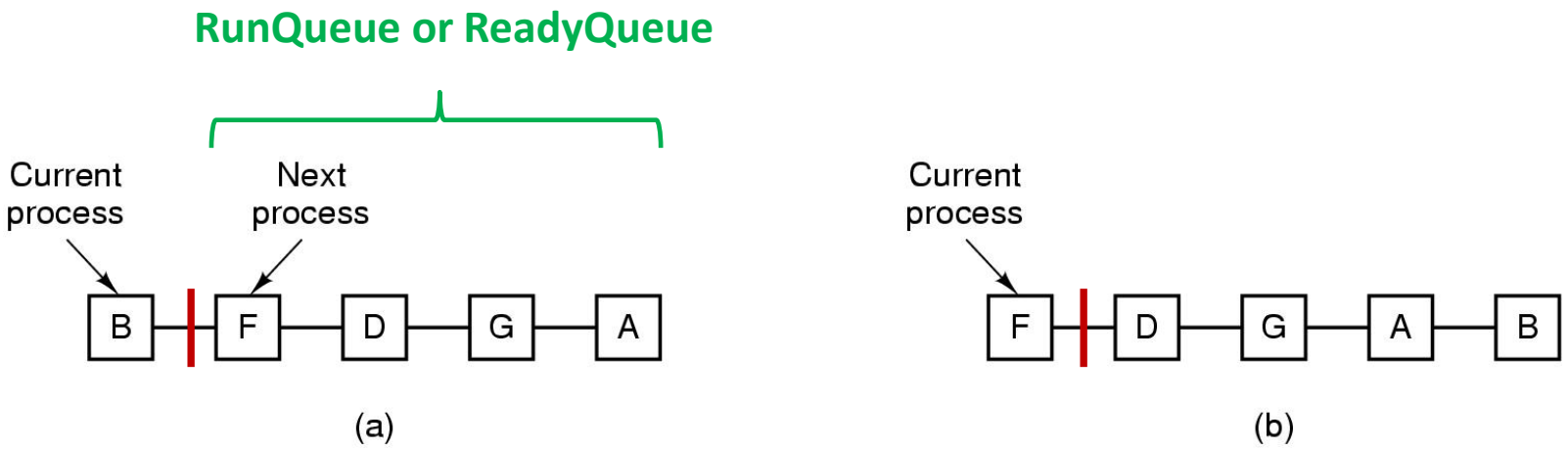
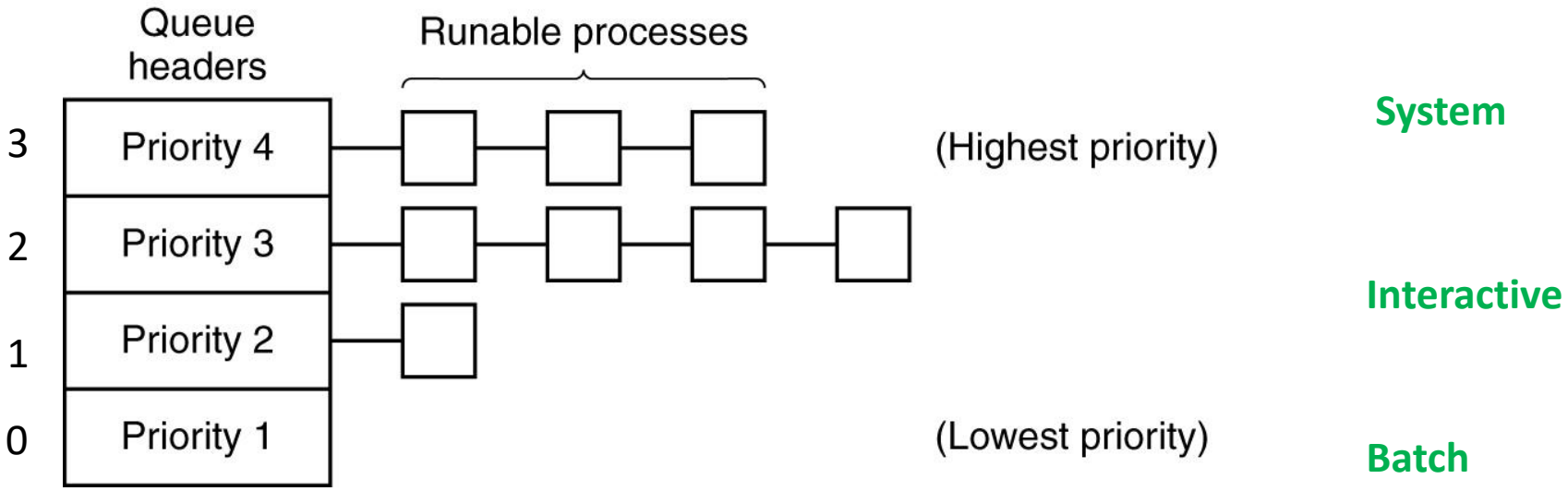


Figure 2-41. Round-robin scheduling.
(a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.

Scheduling in Interactive Systems: Priority Scheduling

- Each process is assigned a priority
- runnable process with the highest priority is allowed to run
- Priorities are assigned statically or dynamically
- Must not allow a process to run forever
  - Can decrease the priority of the currently running process
  - Use time quantum for each process

Scheduling in Interactive Systems: Multiple Level Queuing (MLQ)



- `Process::static_priority`: it is a parameter of the process (non-changing unless requested through system call)
- Multiple levels of priority (MLQ) plus each level is run round-robin
- Issue: starvation if higher priorities have ALWAYS something to run

Multi-Level Feedback Queueing
[MLFQ aka priority decay scheduler]

- Multiple levels of priority MLQ , but
- `Process::dynamic_priority`: [0 .. p->static_priority - 1]
  (this changes constantly based on current state of process)
- If process has to be preempted, moves to (`dynamic_priority--`).
- When it reaches "-1", dynamic priority is reset to (`static_priority-1`)
  - This creates some issues when high prio is reset before other low prio is executing.
- When a process is made ready (from blocked):
  it's dynamic priority is reset to (`static_priority-1`)
- What kind of process should be in bottom queue?
  - Best to assign a higher priority to IO-Bound tasks
  - Best to assign a lower priority to CPU-Bound tasks
- Discussion:

Lottery Scheduling

- Each runnable entity is given a certain number of tickets.
- The more tickets you have, the higher your odds of winning.
- Trade tickets?
- Problems?

Fair Share Scheduler

- Schedule not only based on individual processes, but process's owner.
- N users, each user may have different # of processes.
- Does this make sense on a PC?

Policy versus Mechanism

- Separate what is allowed to be done from how it is done
- Scheduling algorithm parameterized
  - mechanism in the kernel (e.g. quantum)
  - Context switches
- Parameters filled in by user processes
  - policy set by user process (priority, scheduling type)

Scheduling in Real-Time

- Process must respond to an event within a deadline
- Hard real-time vs soft real-time
- Periodic vs aperiodic events
- Processes must be schedulable
- Scheduling algorithms can be static or dynamic

Thread Scheduling

- Already covered in previous lecture:
- Two levels of parallelism: processes and threads within processes
- Kernel-based vs. user-space
  (recall that kernel only schedules threads for kernel based thread model)

SMP Scheduling
(SMP = Symmetric Multi Processors)

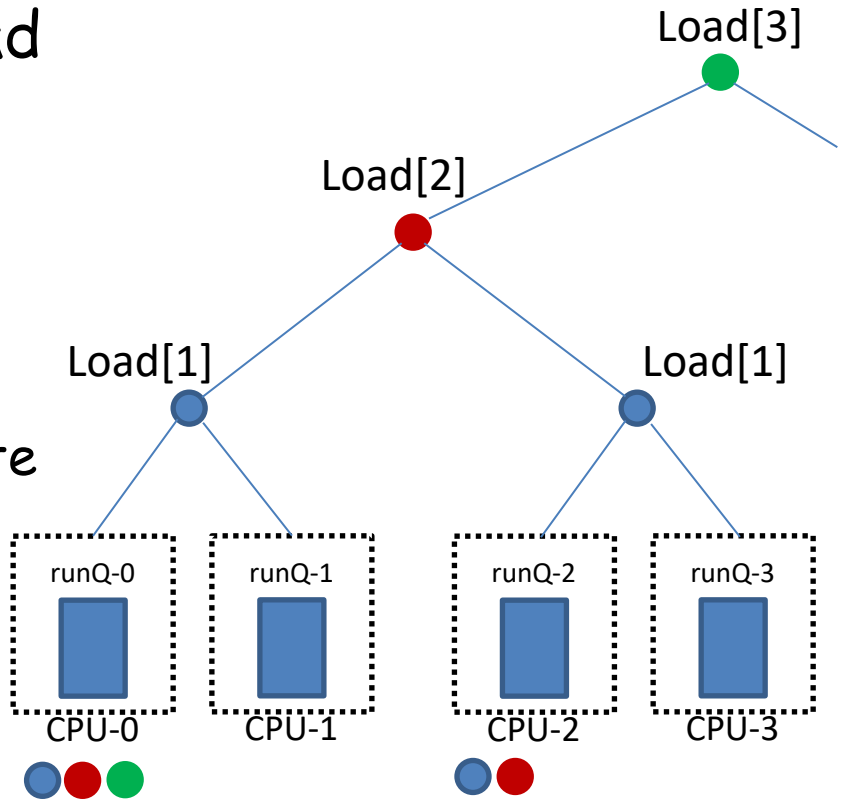
- One Scheduler Data Structure per CPU (hw-thread)
- Note that modern systems can have 100s of "cpus"
- Each CPU schedules for itself and is also triggered by its own timer interrupt (potential preemption)

  ➔ This is known as the local scheduler

- New processes (fork) or threads (clone) are typically assigned to the local CPU
- This "obviously" will lead to imbalances which needs to be dealt with
- Imbalances lead to idle resources and reduce fairness.

Load Balancing (LB)

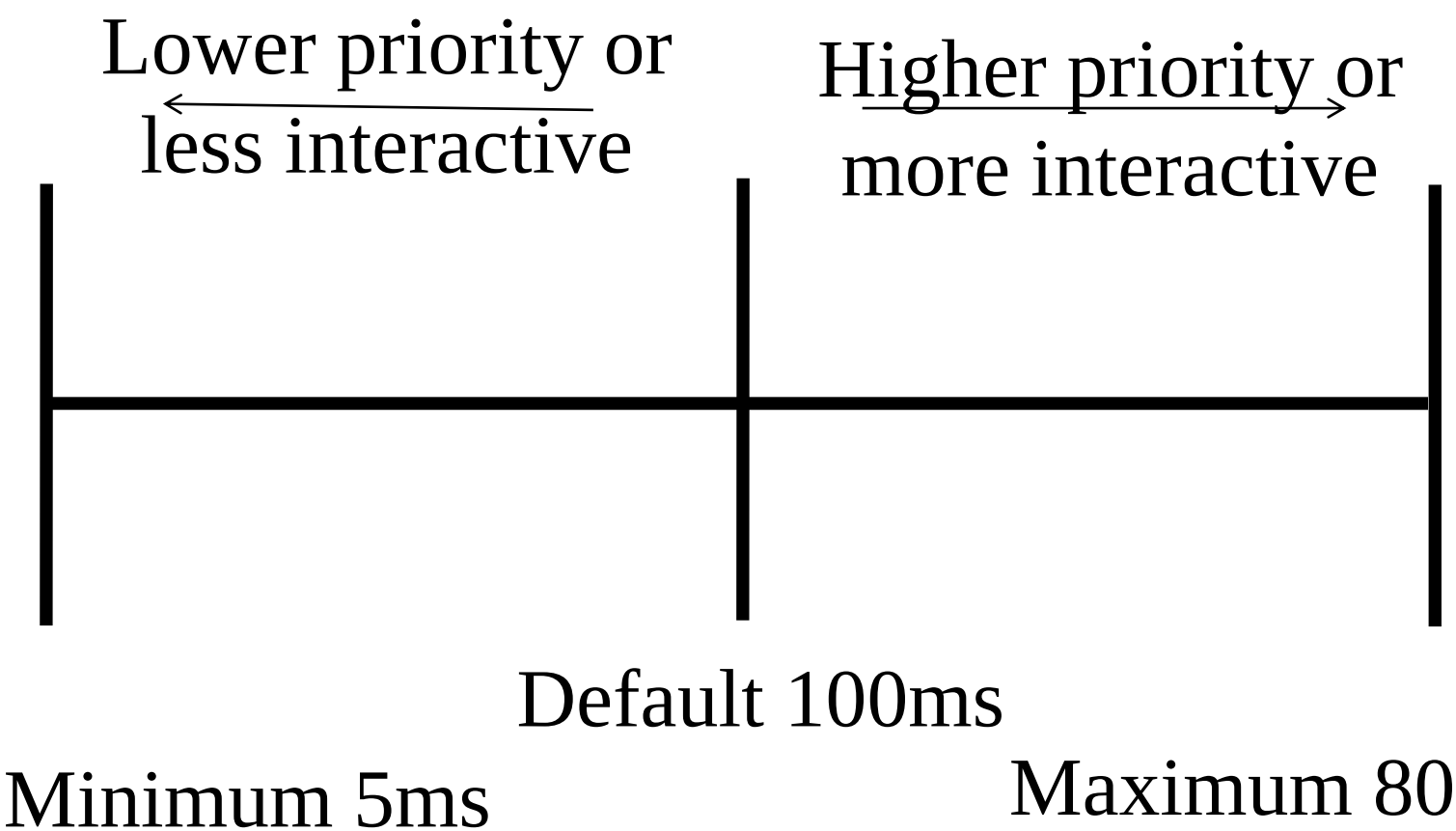
- Occasionally or when no process is runnable, scheduler[i] looks to steal work elsewhere
- Each scheduler maintains a load average and history to determine stability of its load
- LB typically done in a hierarchy
- Frequency of neighbor check:
  - Level in hierarchy ~ cost to migrate
  - Make "small" changes by pulling work from other cpu



Example Linux Scheduling

- Implementation has changed multiple times
- Dynamic Priority-Based Scheduling (classic)
- Two (static) Priority Ranges:
  - Nice value: -20 to +19, default 0.
    Larger values are lower priority.
    Nice value of -20 receives maximum timeslice, +19 minimum.
  - Real-time priority: By default values range 0 to 99.
    Real-time processes have a higher priority than normal processes (nice).

Linux Timeslice



Linux Scheduler Goals

- O(1) scheduling – constant time

  vs O(N) which implies dependent on number of ready processes.

- SMP – each processor has its own locking and individual runqueue

- SMP Affinity. Only migrate process from one CPU to another if imbalanced runqueues.

- Good interactive performance

- Fairness

- Optimized for one or two processes but scales

Priority Arrays

- Each runqueue has two priority arrays:
  - (a) active and (b) expired
    [note these are just pointers to arrays]
- Each priority array contains a bitmap
  - If bit-i is set in bitmap, it indicates there are processes runnable at given priority-i.
- Allows constant retrieval algorithm to find highest set bit (see next slide)

Runqueues
<kernel/sched.c> struct runqueue

activeQ – active priority array
expiredQ – expired priority array

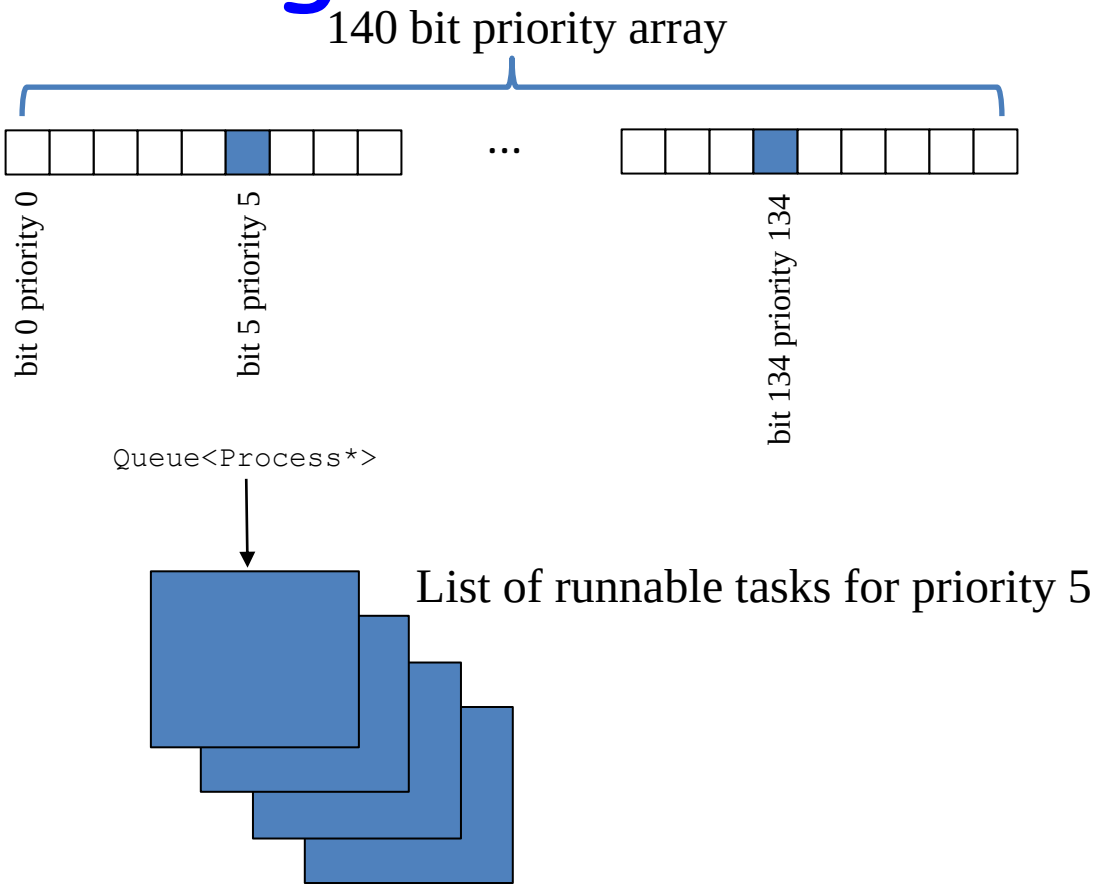
```
Queue<Process*> *activeQ =
     calloc(sizeof(Queue<Process*>), maxprios);
Queue<Process*> *expiredQ =
     calloc(sizeof(Queue<Process*>), maxprios);

   OR

Queue<Process*> *activeQ =
     new Queue<Process*> [ maxprios ];
Queue<Process*> *expiredQ =
     new Queue<Process*> [ maxprios ];
```

---

migration_thread
migration_queue
nr_iowait – number of tasks waiting on I/O

Scheduler Algorithm

- Maintain bitmap of state of the queues (in three 64-bit words)
- Identify highest prio queue that is not empty:
  - Use ffz() or ffnz() to find first bit that is set/notset supported by hardware instructions.
  - Only needs 3 64-bits to be examined to return correct priority (queue)
- Then dequeue_front from that queue.



ffz() : https://www.kernel.org/doc/htmldocs/kernel-api/API-ffz.html
https://en.wikipedia.org/wiki/Find_first_set

Scheduler Operations

- add_to_runqueue(Process *p)

```
if (p -> dynamic prio < 0)
        reset and enter <p> into expireQ
else
        add <p> to activeQ
```

- Process* get_from_runqueue()

```
if activeQ not empty
        return activeQ[highest prio].front()  // you have to pop()
else
    swap(activeQ,expiredQ) and try again
    [ swapping means swapping the pointers to array
      not swapping the entire arrays .. Smart programming ]
```

Calculating Priority and Timeslice

`effective_prio()` returns task's dynamic priority.

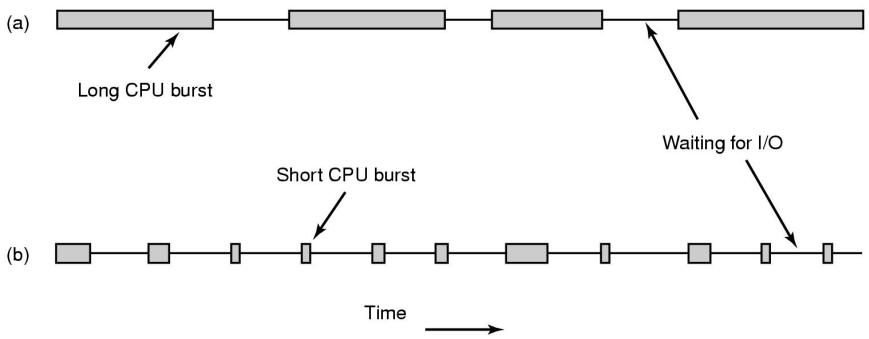
nice value + or – bonus in range -5 to +5

Interactivity measure by how much time a process sleeps. Indicates I/O activity.

sleep_avg incremented to max_sleep_avg (10 millisecs) every time it does I/O. If no I/O, decremented.

How to measure interactivity ?

How to determine interactivity?



- Sum(cpu_burst) / wall time
- Sum(io_burst) / wall time
- Sum(cpu_burst) / Sum(io_burst)

- Note excludes queue-time in the ready queue

- Need to take "recency" into account:
- Mostly interested in recent history
- Use a weighting mechanism

- `value(t) = measure(t) * R + value(t-1) * (1-R)`

R in (0 .. 1].

- If R == 1 then only the last period counts. The closer R gets to 0 the more history counts