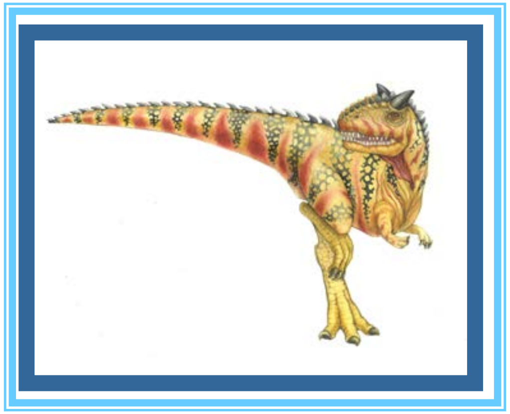
# Chapter 5: Process Scheduling



# Chapter 5: Process Scheduling

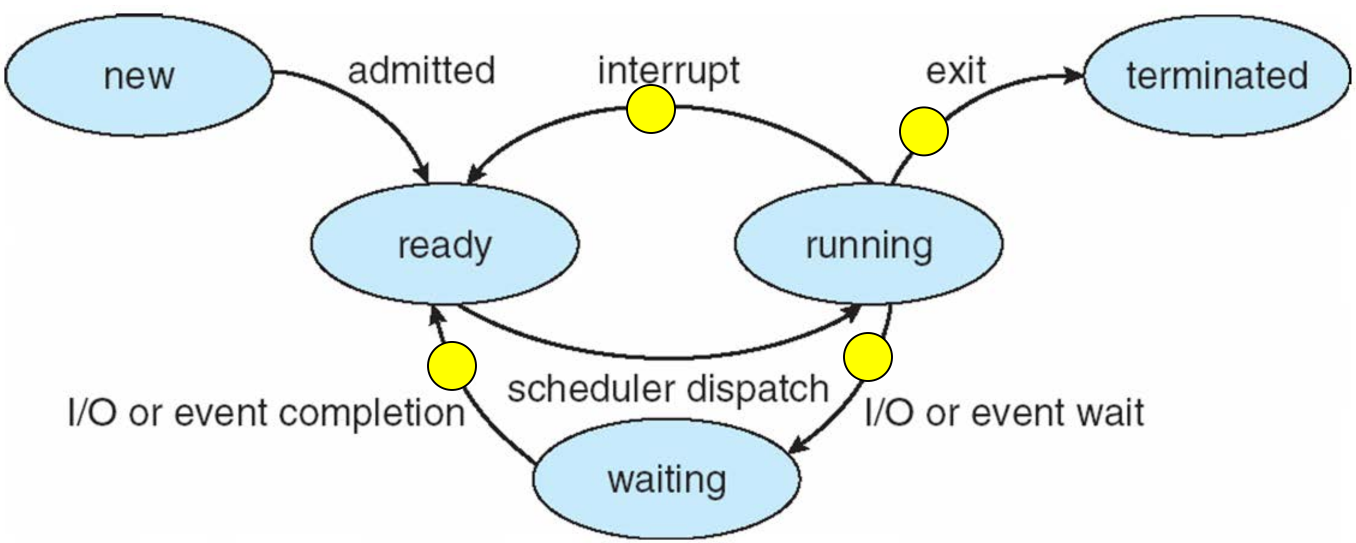
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

# Process Scheduling

- Have M jobs ready to run
- Have $N \geq 1$ CPUs
- Which job to assign to which CPU(s) at what time?

# CPU Scheduler

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- **nonpreemptive scheduler uses 1 and 4**
- **Preemptive scheduler kicks in for all four time points**



# CPU Scheduler

LXR linux/fs/block_d

lxr.linux.no/linux+v3.6.3/fs/block_dev.c#L748

```c
        else
                return true;    /* is a partition of an un-held device */
}

/**
 * bd_prepare_to_claim - prepare to claim a block device
 * @bdev: block device of interest
 * @whole: the whole device containing @bdev, may equal @bdev
 * @holder: holder trying to claim @bdev
 *
 * Prepare to claim @bdev.  This function fails if @bdev is already
 * claimed by another holder and waits if another claiming is in
 * progress.  This function doesn't actually claim.  On successful
 * return, the caller has ownership of bd_claiming and bd_holder[s].
 *
 * CONTEXT:
 * spin_lock(&bdev_lock).  Might release bdev_lock, sleep and regrab
 * it multiple times.
 *
 * RETURNS:
 * 0 if @bdev can be claimed, -EBUSY otherwise.
 */
static int bd_prepare_to_claim(struct block_device *bdev,
                               struct block_device *whole, void *holder)
{
retry:
        /* if someone else claimed, fail */
        if (!bd_may_claim(bdev, whole, holder))
                return -EBUSY;

        /* if claiming is already in progress, wait for it to finish */
        if (whole->bd_claiming) {
                wait_queue_head_t *wq = bit_waitqueue(&whole->bd_claiming, 0);
                DEFINE_WAIT(wait);

                prepare_to_wait(wq, &wait, TASK_UNINTERRUPTIBLE);
                spin_unlock(&bdev_lock);
                schedule();
                finish_wait(wq, &wait);
                spin_lock(&bdev_lock);
                goto retry;
        }

        /* yay, all mine */
        return 0;
}
```

# Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

# Context switch

lxr.linux.no/linux+v3.6.3/kernel/sched/core.c#L2046

```c
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
{
        struct mm_struct *mm, *oldmm;

        prepare_task_switch(rq, prev, next);

        mm = next->mm;
        oldmm = prev->active_mm;
        /*
         * For paravirt, this is coupled with an exit in switch_to to
         * combine the page table reload and the switch backend into
         * one hypercall.
         */
        arch_start_context_switch(prev);

        if (!mm) {
                next->active_mm = oldmm;
                atomic_inc(&oldmm->mm_count);
                enter_lazy_tlb(oldmm, next);
        } else
                switch_mm(oldmm, mm, next);

        if (!prev->mm) {
                prev->active_mm = NULL;
                rq->prev_mm = oldmm;
        }
        /*
         * Since the runqueue lock will be released by the next
         * task (which is an invalid locking op but in the case
         * of the scheduler it's an obvious special-case), so we
         * do an early lockdep release here:
         */
#ifndef __ARCH_WANT_UNLOCKED_CTXSW
        spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
#endif

        /* Here we just switch the register state and the stack. */
        switch_to(prev, next, prev);

        barrier();
        /*
         * this_rq must be evaluated again because prev may have moved
         * CPUs since it called schedule(), thus the 'rq' on its stack
         * frame will be invalid.
         */
        finish_task_switch(this_rq(), prev);
}
```

switching address space

switching register state and stack

# Context switch

lxr.linux.no/linux+v3.6.3/arch/x86/include/asm/switch_to.h

```c
/* frame pointer must be last for get_wchan */
#define SAVE_CONTEXT    "pushf ; pushq %%rbp ; movq %%rsi,%%rbp\n\t"
#define RESTORE_CONTEXT "movq %%rbp,%%rsi ; popq %%rbp ; popf\t"

#define __EXTRA_CLOBBER  \
        , "rcx", "rbx", "rdx", "r8", "r9", "r10", "r11", \
          "r12", "r13", "r14", "r15"

#ifdef CONFIG_CC_STACKPROTECTOR
#define __switch_canary                                                   \
        "movq %P[task_canary](%%rsi),%%r8\n\t"                            \
        "movq %%r8,"__percpu_arg([gs_canary])"\n\t"
#define __switch_canary_oparam                                            \
        , [gs_canary] "=m" (irq_stack_union.stack_canary)
#define __switch_canary_iparam                                            \
        , [task_canary] "i" (offsetof(struct task_struct, stack_canary))
#else   /* CC_STACKPROTECTOR */
#define __switch_canary
#define __switch_canary_oparam
#define __switch_canary_iparam
#endif  /* CC_STACKPROTECTOR */

/* Save restore flags to clear handle leaking NT */
#define switch_to(prev, next, last) \
        asm volatile(SAVE_CONTEXT                                         \
             "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */       \
             "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */    \
             "call __switch_to\n\t"                                       \
             "movq "__percpu_arg([current_task])",%%rsi\n\t"              \
             __switch_canary                                              \
             "movq %P[thread_info](%%rsi),%%r8\n\t"                       \
             "movq %%rax,%%rdi\n\t"                                       \
             "testl  %[_tif_fork],%P[ti_flags](%%r8)\n\t"                 \
             "jnz   ret_from_fork\n\t"                                    \
             RESTORE_CONTEXT                                              \
             : "=a" (last)                                                \
               __switch_canary_oparam                                     \
             : [next] "S" (next), [prev] "D" (prev),                      \
               [threadrsp] "i" (offsetof(struct task_struct, thread.sp)), \
               [ti_flags] "i" (offsetof(struct thread_info, flags)),      \
               [_tif_fork] "i" (_TIF_FORK),                               \
               [thread_info] "i" (offsetof(struct task_struct, stack)),   \
               [current_task] "m" (current_task)                          \
               __switch_canary_iparam                                     \
             : "memory", "cc" __EXTRA_CLOBBER)

#endif /* CONFIG_X86_32 */

#endif /* _ASM_X86_SWITCH_TO_H */
```

switching kernel stack happens in here

# Context switch / switch kernel stack

lxr.linux.no/linux+v3.6.3/arch/x86/kernel/process_64.c#L269

```c
 *
 * This could still be optimized:
 * - fold all the options into a flag word and test it with a single test.
 * - could test fs/gs bitsliced
 *
 * Kprobes not supported here. Set the probe on schedule instead.
 * Function graph tracer not supported too.
 */
__notrace_funcgraph struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
        struct thread_struct *prev = &prev_p->thread;
        struct thread_struct *next = &next_p->thread;
        int cpu = smp_processor_id();
        struct tss_struct *tss = &per_cpu(init_tss, cpu);
        unsigned fsindex, gsindex;
        fpu_switch_t fpu;

        fpu = switch_fpu_prepare(prev_p, next_p, cpu);

        /*
         * Reload esp0, LDT and the page table pointer:
         */
        load_sp0(tss, next);

        /*
         * Switch DS and ES.
         * This won't pick up thread selector changes, but I guess that is ok.
         */
        savesegment(es, prev->es);
        if (unlikely(next->es | prev->es))
                loadsegment(es, next->es);

        savesegment(ds, prev->ds);
        if (unlikely(next->ds | prev->ds))
                loadsegment(ds, next->ds);


        /* We must save %fs and %gs before load_TLS() because
         * %fs and %gs may be cleared by load_TLS().
         *
         * (e.g. xen_load_tls())
         */
        savesegment(fs, fsindex);
        savesegment(gs, gsindex);

        load_TLS(next, cpu);
```

Switch kernel stack

# Scheduling Criteria

- **Throughput** – # of processes that complete their execution per time unit
  - Higher is better
- **Turnaround time** – amount of time to execute a particular process
  - Lower is better
- **Response time** – time from request to first response
  - E.g. mouse clicking on the menu bar to the showing of the menu
  - Lower is better
- Above criteria are affected by secondary criteria
  - **CPU utilization** – keep the CPU as busy as possible
  - **Waiting time** – amount of time a process has been waiting in the ready queue

# First-Come, First-Served (FCFS) Scheduling

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Suppose that the processes arrive in the order: $P_1$ , $P_2$ , $P_3$
  The Gantt Chart for the schedule is:

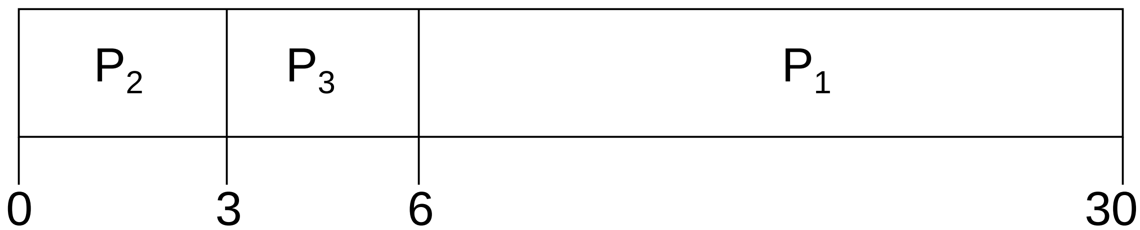

- Waiting time for $P_1$ = 0; $P_2$ = 24; $P_3$ = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Throughput: 3 processes / 30 seconds = 0.1 processes / sec
- Turnaround Time: $P_1$: 24, $P_2$: 27, $P_3$: 30
  - Avg. TT: (24+27+30)/3 = 27

# FCFS Scheduling (Cont)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

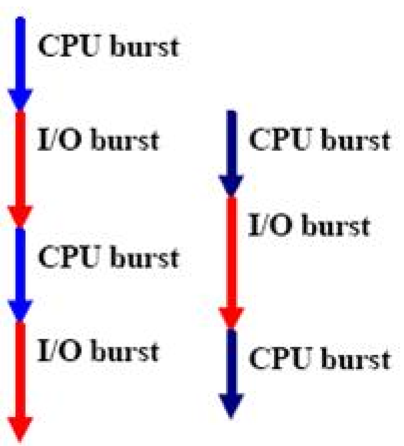
- The Gantt chart for the schedule is:

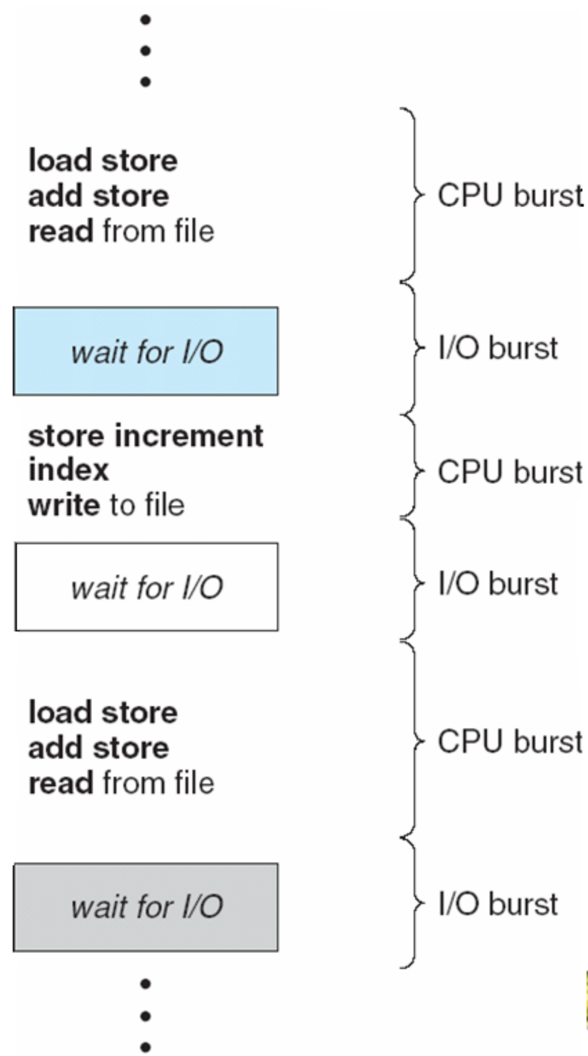



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Throughput: 3 / 30 = 0.1 processes / sec
- Turnaround time: Time: $P_1$: 30, $P_2$: 3, $P_3$: 6
  - Avg. TT: (30+3+6)/3 = 13
- Much better than previous case
- Scheduling algorithm can reduce TT
  - Minimize waiting time to minimize TT
- What about throughput?

# Alternating Sequence of CPU And I/O Bursts

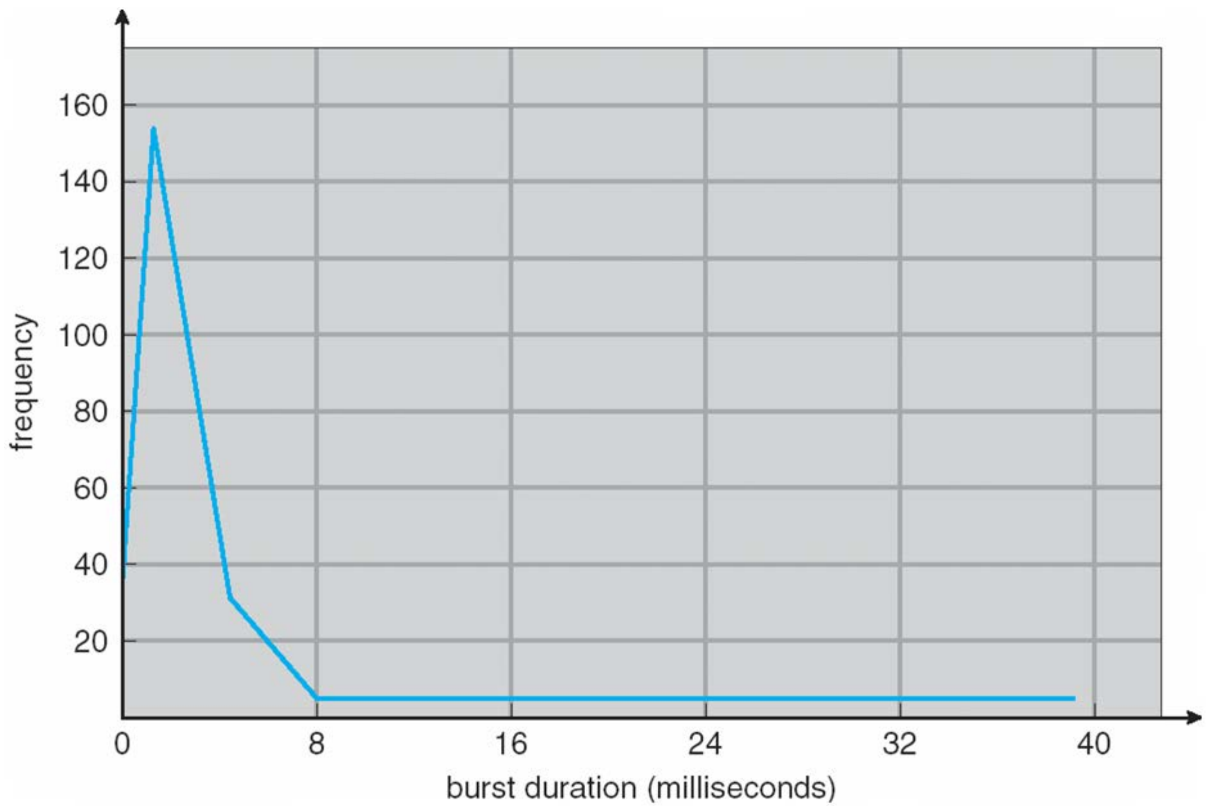
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution, I/O wait, and event wait
- An I/O device can be considered as a special purpose CPU
- Goal: keep all CPUs and all I/O devices busy



# Histogram of CPU-burst Times

Many short CPU bursts an few long bursts




What does this mean for FCFS?

# FCFS Convoy effect

- CPU bound jobs will hold CPU until exit or I/O
  - I/O rare for CPU-bound thread
  - Long periods where no I/O requests issued, and CPU held
  - => poor I/O device utilization
- Example: one CPU-bound job, many I/O bound
  - CPU bound runs (I/O device idle)
  - CPU bound blocks
  - I/O bound job(s) run, quickly block on I/O
  - CPU bound runs again
  - I/O completes
  - CPU bound still runs while I/O device idle (continue?)

# Shortest-Job-First (SJF) Scheduling

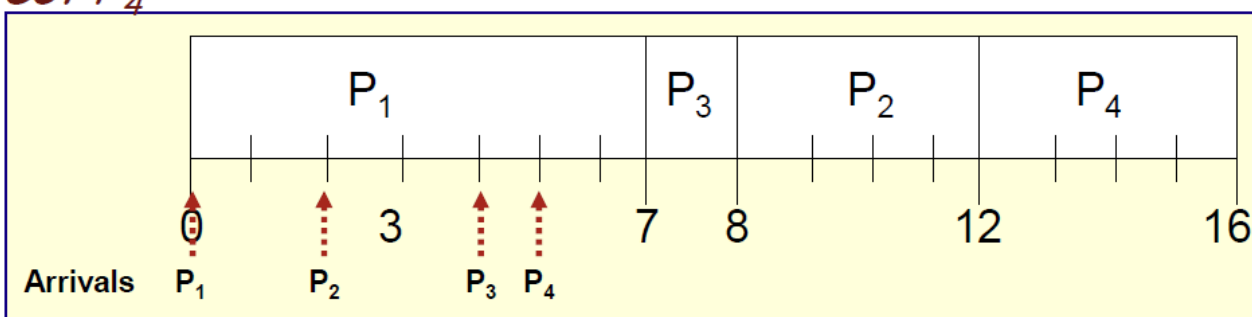
- Associate with each process the length of its next CPU burst.  Use these lengths to schedule the process with the shortest time
- Process with shortest burst goes next
  - If tie then use FCFS to break tie
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
- Two schemes:
  - *Non-preemptive* – once CPU assigned, process not preempted until its CPU burst completes
  - *Preemptive* – if a process with CPU burst less than remaining time of current, preempt
    - Shortest-Remaining-Time-First (SRTF)

# Example of Non-Preemptive SJF

- T = 0: RQ = {$P_1$}
  *Select $P_1$*
- T = 2: RQ = {$P_2$}
  *No-Preemption*
- T = 4: RQ = {$P_3$, $P_2$}
  *No-Preemption*
- T = 5: RQ = {$P_3$, $P_2$ , $P_4$}
  *No-Preemption*
- T = 7: RQ = {$P_3$, $P_2$ , $P_4$}
  *$P_1$ completes*, *Select $P_3$*
- T = 8: RQ = {$P_2$ , $P_4$}
  *$P_3$ completes*, *Select $P_2$*
- T = 12: RQ = {$P_4$}
  *$P_2$ completes*, *Select $P_4$*
- T = 16: RQ = {}
  *$P_4$ completes*

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

- Average Waiting Time:
  [0 + (8 – 2) + (7 – 4) + (12 – 5)]/4 =
  [6 + 3 + 7]/4 = 4

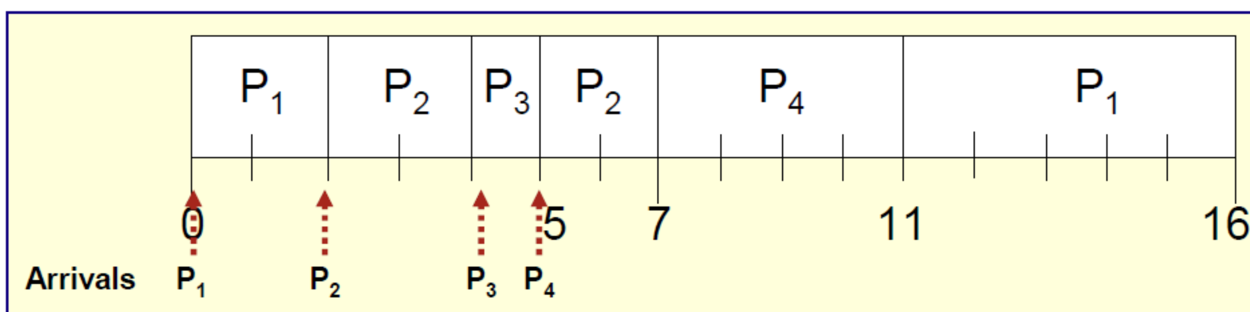


# Example of Preemptive SJF

- T = 0: RQ = {$P_1$}
  *Select $P_1$*
- T = 2: RQ = {$P_2$}
  *preempt $P_1$ , Select $P_2$*
- T = 4: RQ = {$P_3$, $P_1$}
  *preempt $P_2$ , Select $P_3$*
- T = 5: RQ = {$P_2$, $P_4$ , $P_1$}
  *$P_3$ completes, Select $P_2$*
- T = 7: RQ = {$P_4$, $P_1$}
  *$P_2$ completes, Select $P_4$*
- T = 11: RQ = {$P_1$}
  *$P_4$ completes, Select $P_1$*
- T = 16: RQ = {}
  *$P_2$1completes*

| Process | Arrival Time | Burst Time |
|---|---|---|
| *P1* | 0.0 | 7 |
| *P2* | 2.0 | 4 |
| *P3* | 4.0 | 1 |
| *P4* | 5.0 | 4 |

- Average Waiting Time:
  [(11-2) + (5-4) + (0) + (7-5)]/4 =
  [ 9 + 1 + 0 + 2]/4 = 3

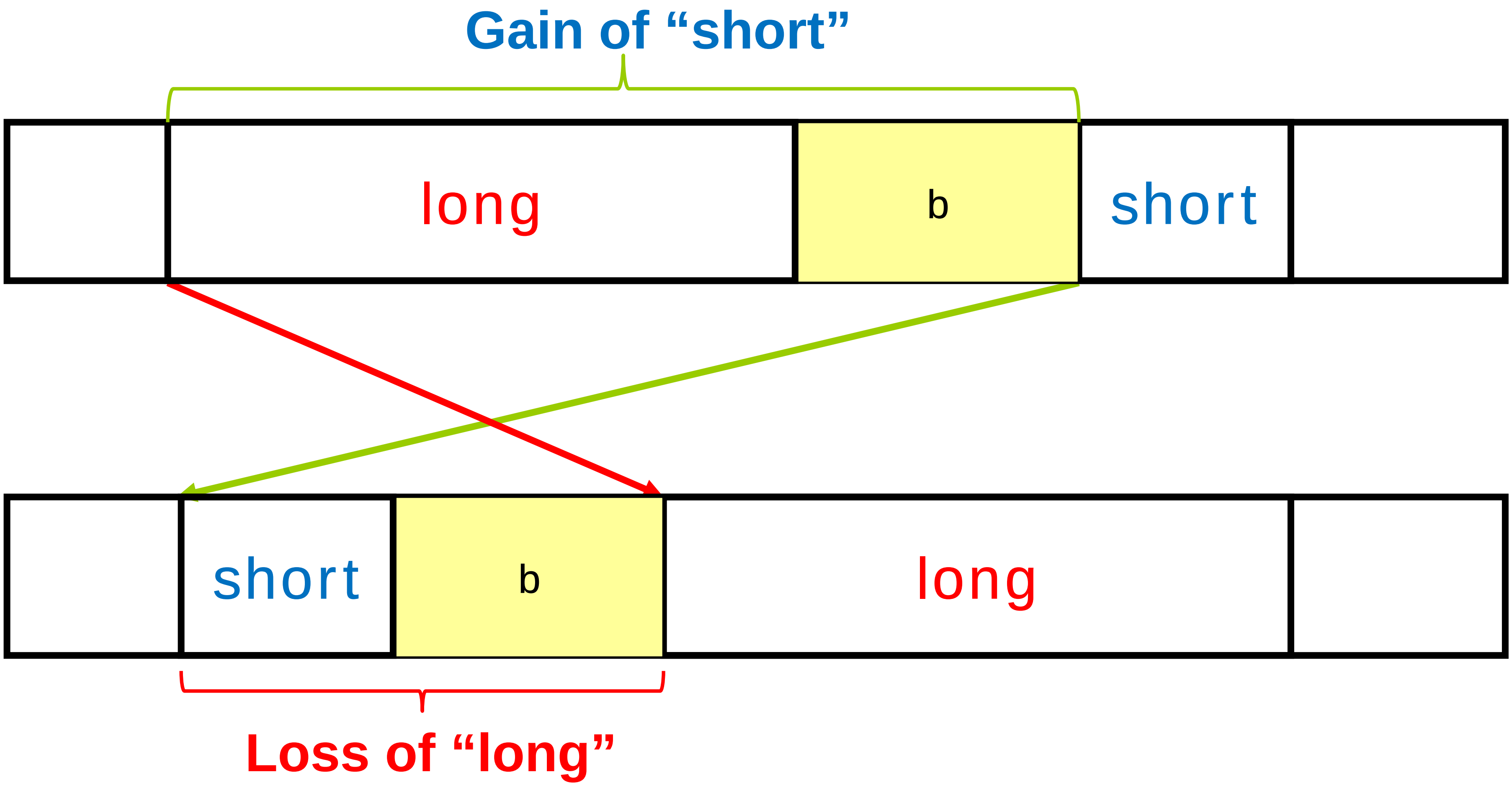


# Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual length of $n^{th}$ CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

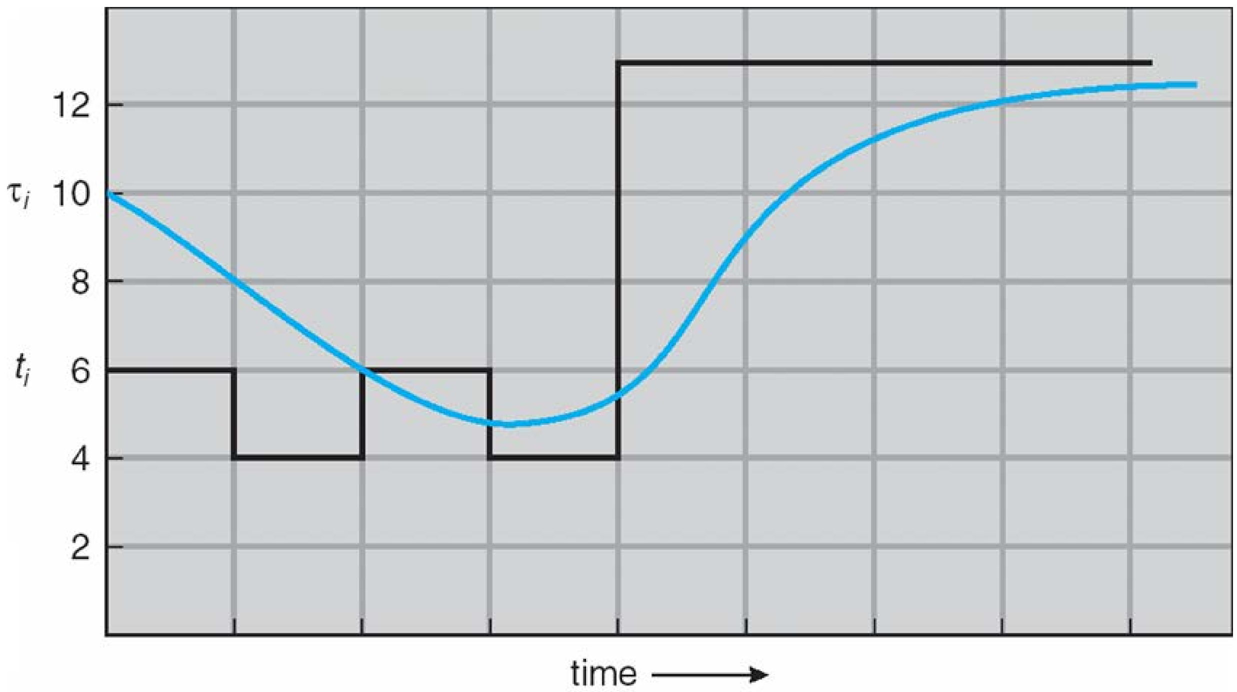
# SJF Optimality




Proof that the SJF algorithm is optimal

**Gain of short > Loss of long**

# Prediction of the Length of the Next CPU Burst




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 | ... |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | ... |

# Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_n -1 + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

# Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ processor sharing (appears as dedicated processor with speed 1/n actual)
  - *q* must be large with respect to context switch, otherwise overhead is too high

# Example of RR with Time Quantum = 4

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

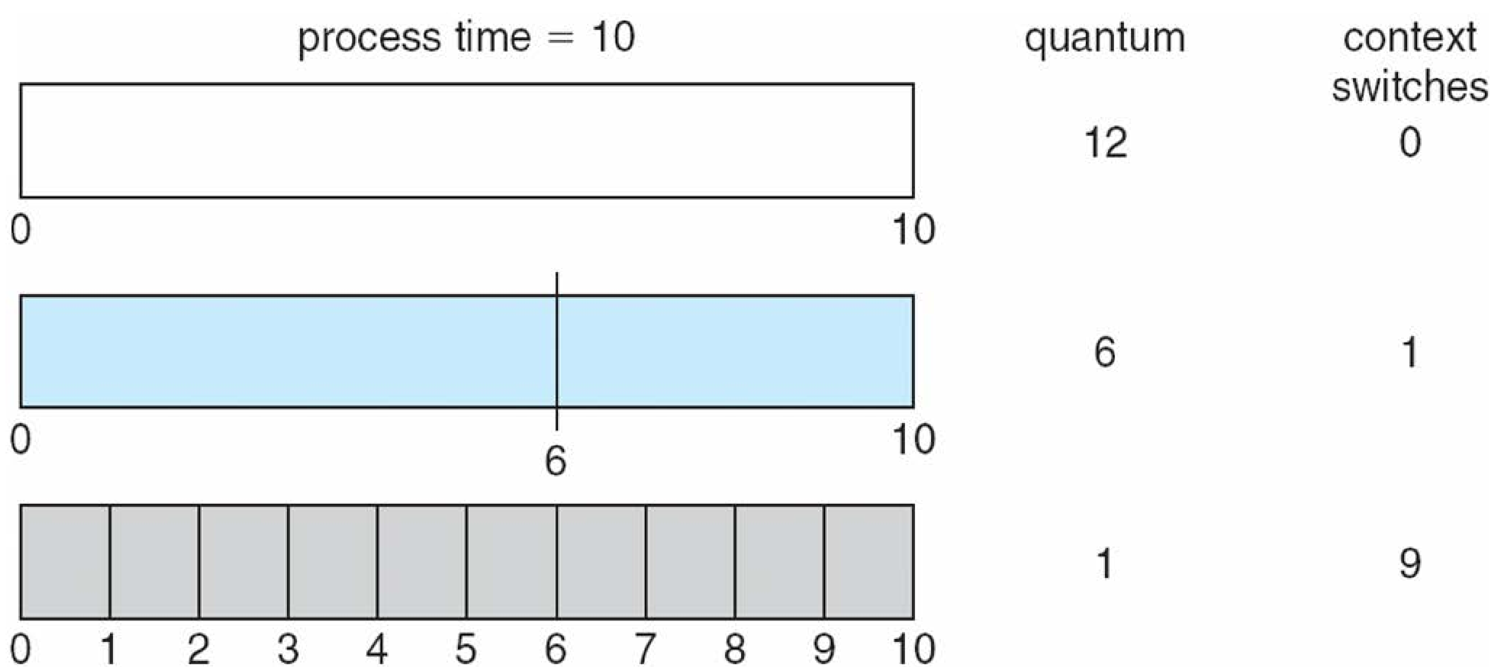
- The Gantt chart is:

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ |
|---|---|---|---|---|---|---|---|

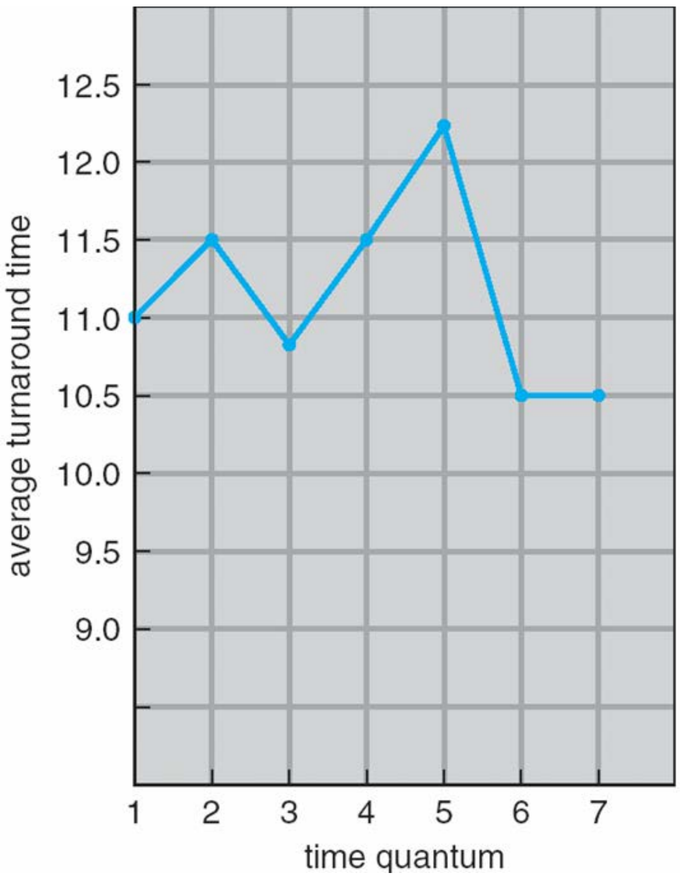
0 4 7 10 14 18 22 26 30

- Typically, higher average turnaround than SJF, but better *response*

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum

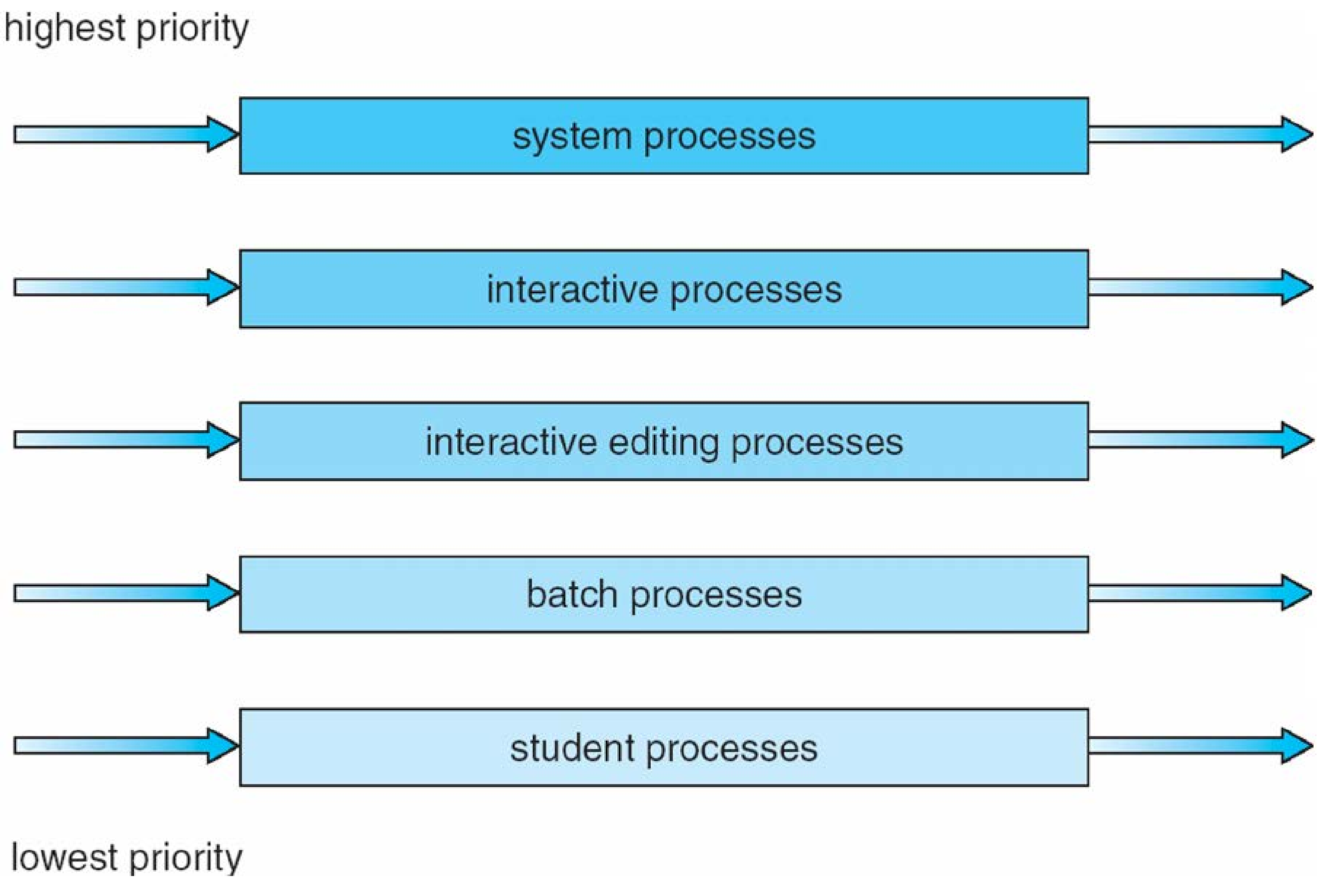



| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

# Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

# Multilevel Queue Scheduling



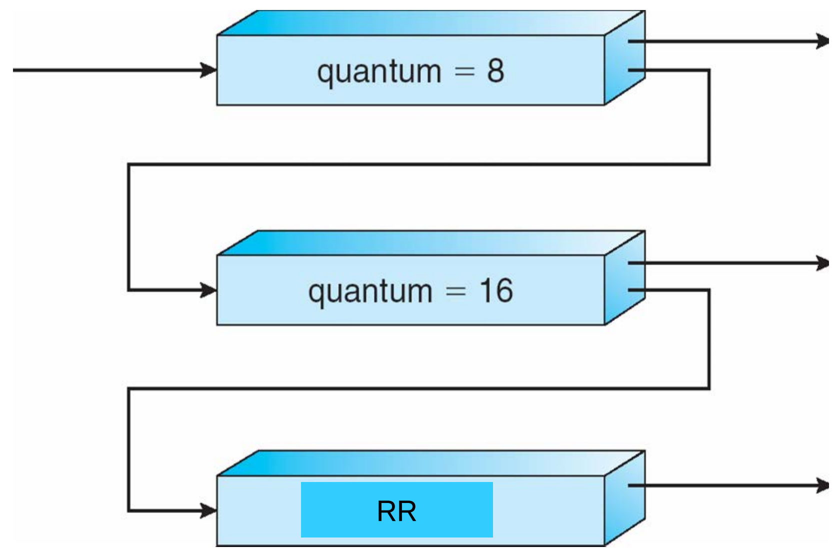
# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

# Multilevel Feedback Queues

1. Each priority level has a ready queue, and a time quantum
2. A new process is positioned at the end of the top-level queue
3. At some stage the process reaches the head of the queue and is assigned the CPU
4. If the process is completed it leaves the system
5. If the process voluntarily relinquishes control it leaves the queuing network, and when the process becomes ready again it enters the system on the same queue level.
6. If the process uses all the quantum time, it is pre-empted and positioned at the end of the next lower level queue.
7. At the base level queue the processes circulate in round robin fashion until they complete and leave the system.
8. If a process blocks for I/O, it is 'promoted' one level, and placed at the end of the next-higher queue. This allows I/O bound processes to be favored by the scheduler and allows processes to 'escape' the base level queue.

Multiple FIFO queues



# Multilevel Feedback Queues

- I/O bound processes tend to congregate in higher-level queues
  - This implies greater device utilization
- CPU-bound processes will sink deeper into the queues
- Quantum in top queue should be large enough to satisfy majority of I/O-bound processes
- Can assign a process a lower priority by starting it at a lower-level queue
- Can raise priority by moving process to a higher queue, thus can use in conjunction with aging

# Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

# Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
  - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

# Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
     int i;
    pthread t tid[NUM THREADS];
    pthread attr t attr;
    /* get the default attributes */
    pthread attr init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread attr setschedpolicy(&attr, SCHED OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
            pthread create(&tid[i],&attr,runner,NULL);
```

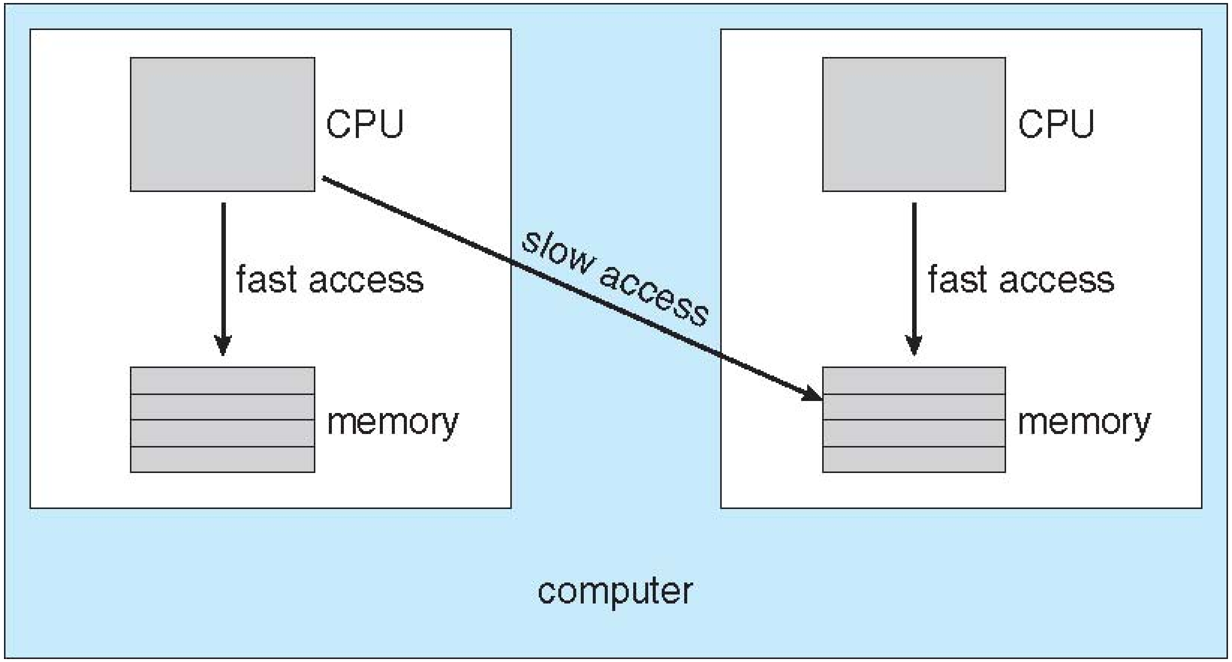
# Pthread Scheduling API

```
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
            pthread join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```

# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**

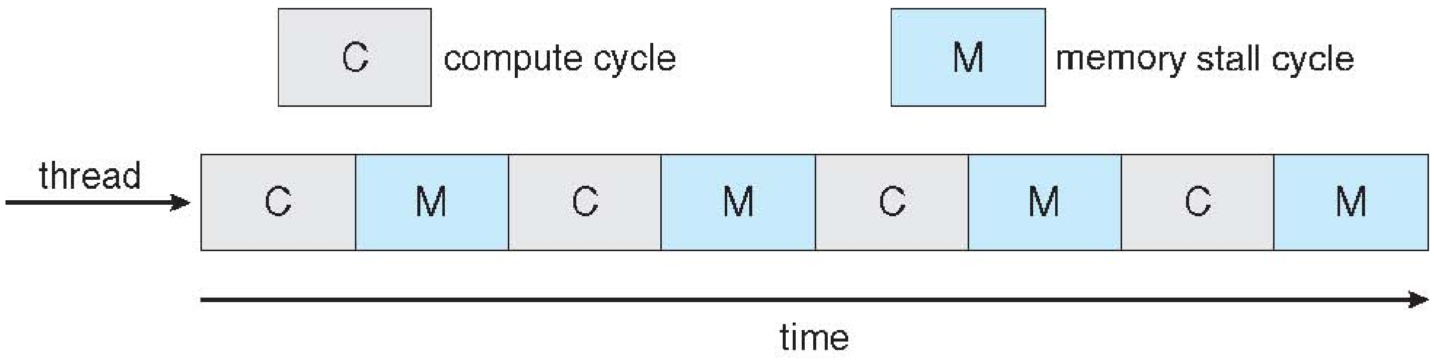
# NUMA and CPU Scheduling



# Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

# Multithreaded Multicore System



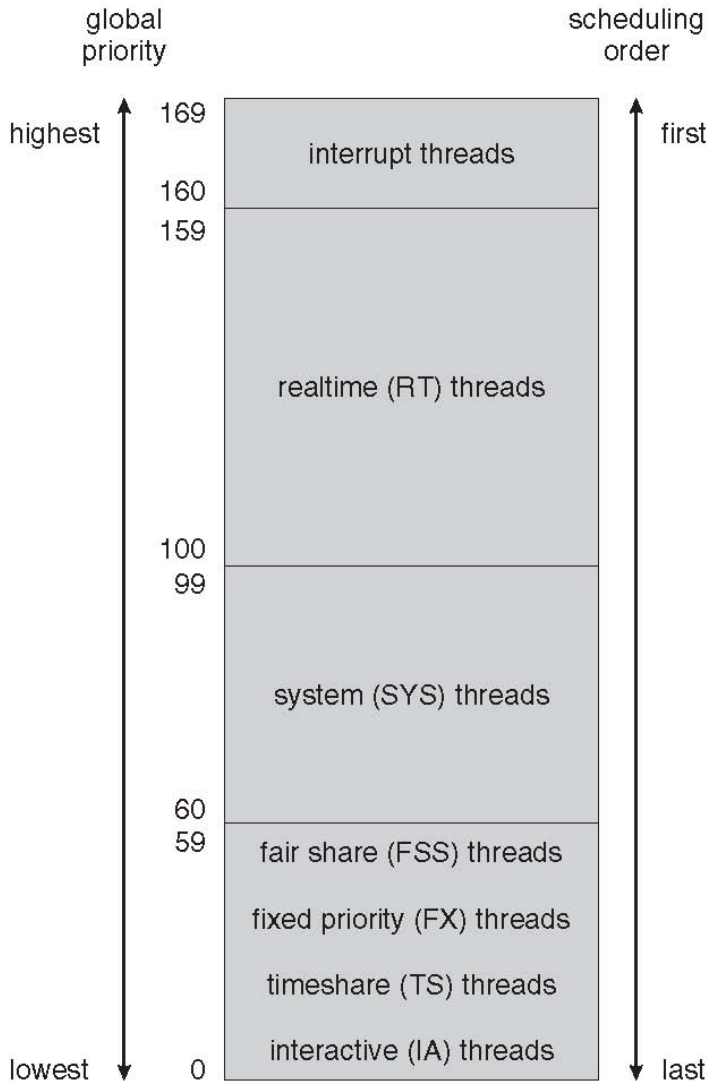
# Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

# Solaris Dispatch Table

| priority | time quantum | time quantum expired | return from sleep |
|---|---|---|---|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

# Solaris Scheduling



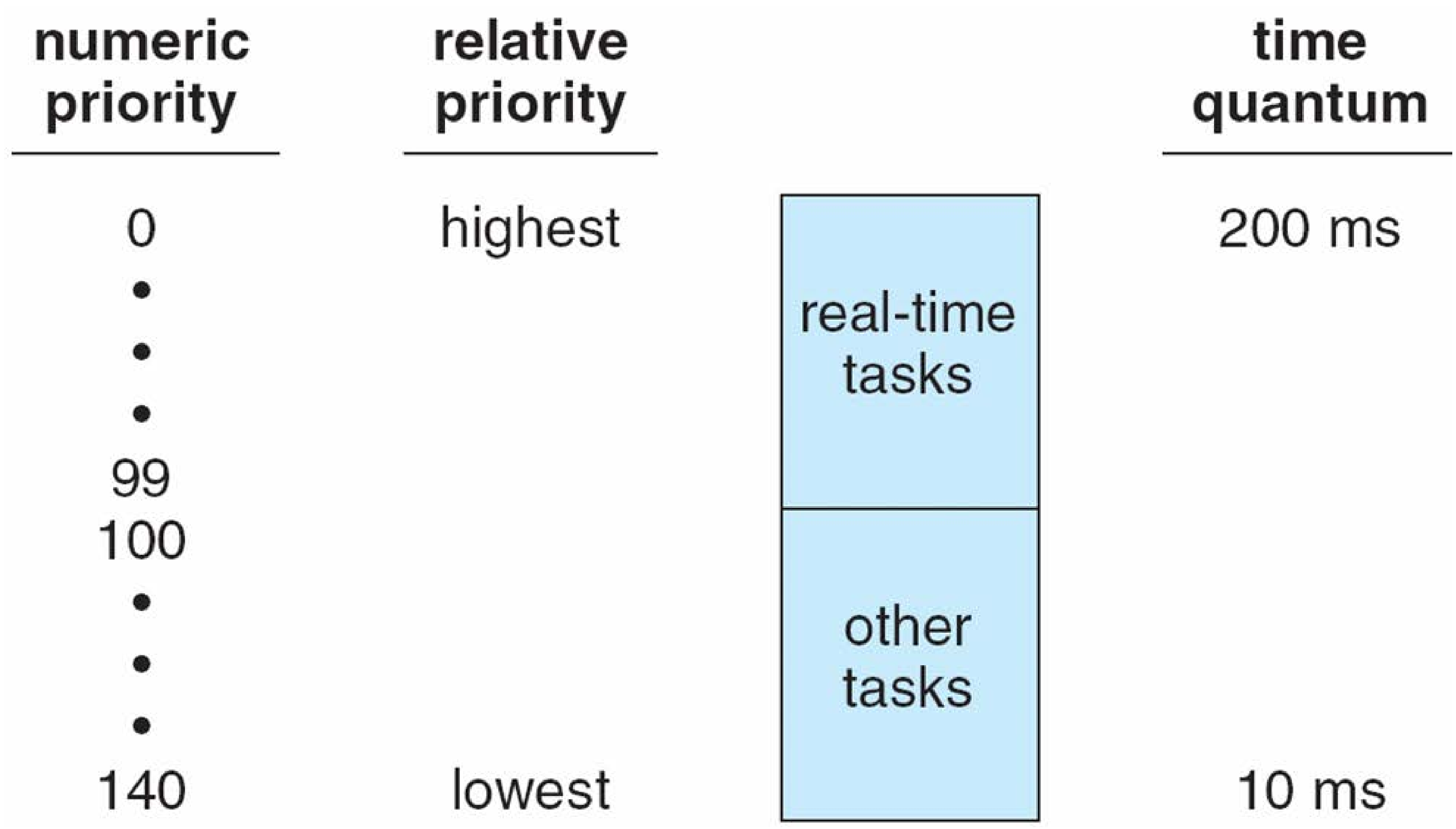
# Windows XP Priorities

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

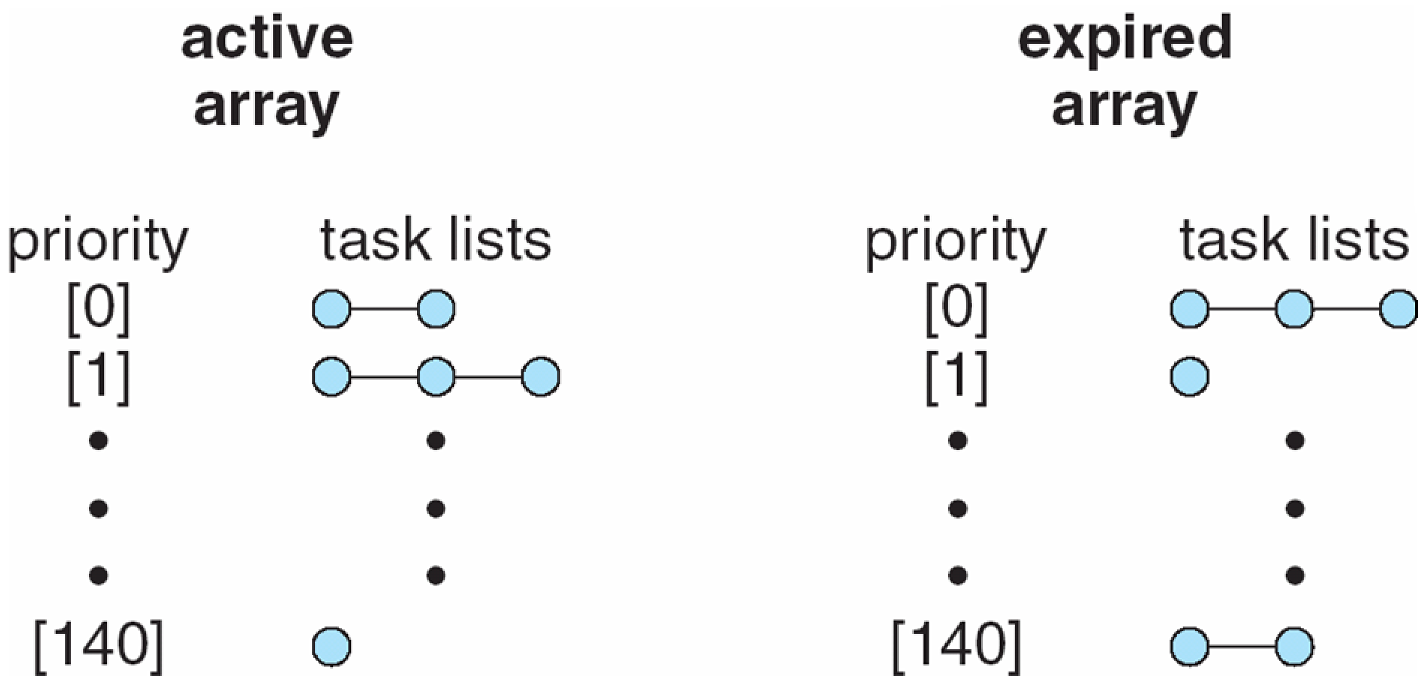
# Linux Scheduling

- Constant order *O*(1) scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- (figure 5.15)

# Priorities and Time-slice length



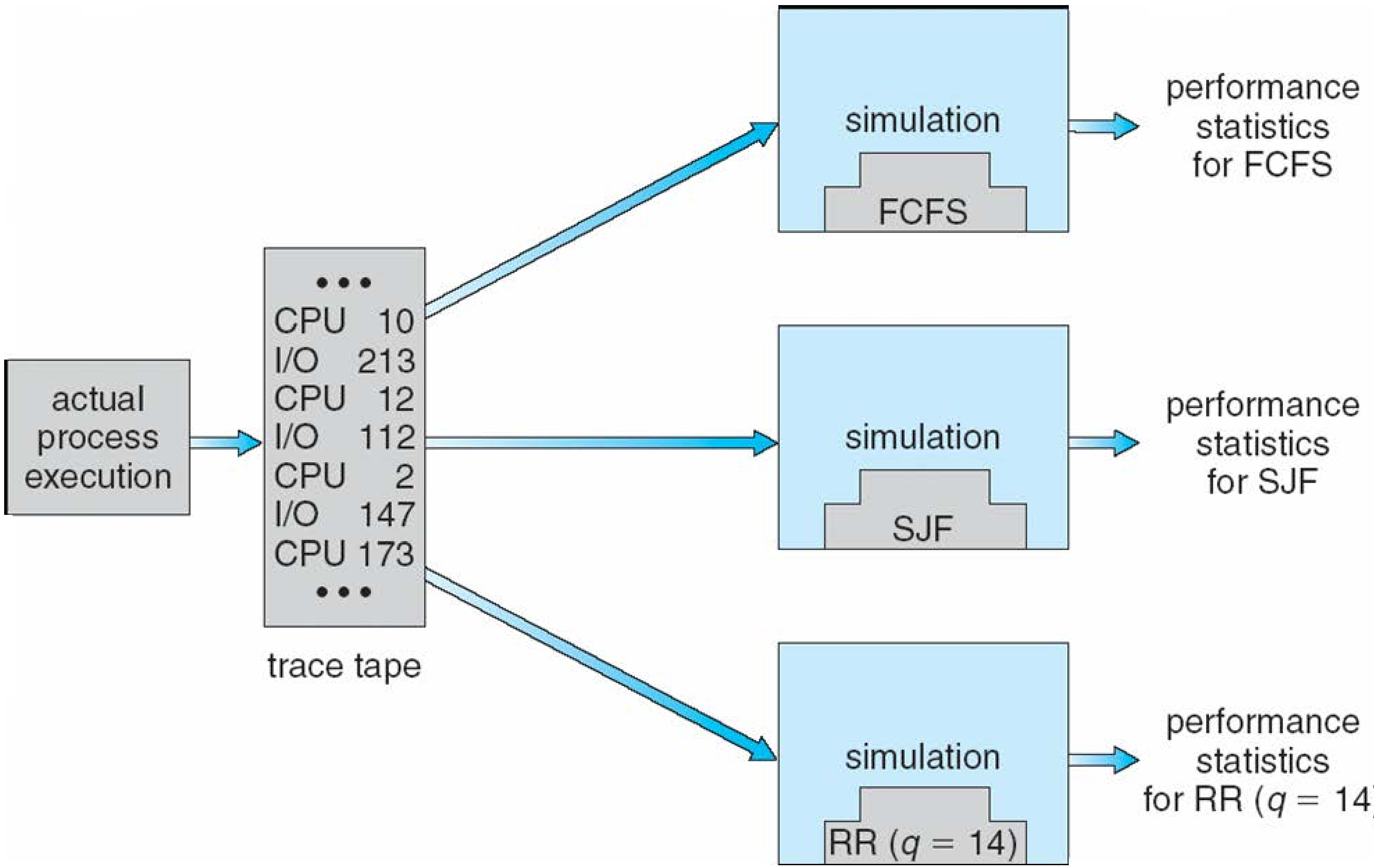
# List of Tasks Indexed According to Priorities



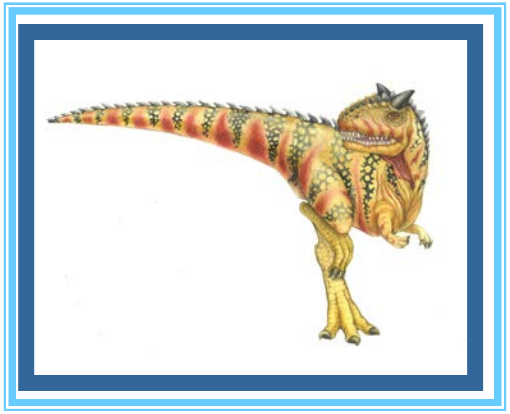
# Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm  for that workload
- Queueing models
- Implementation

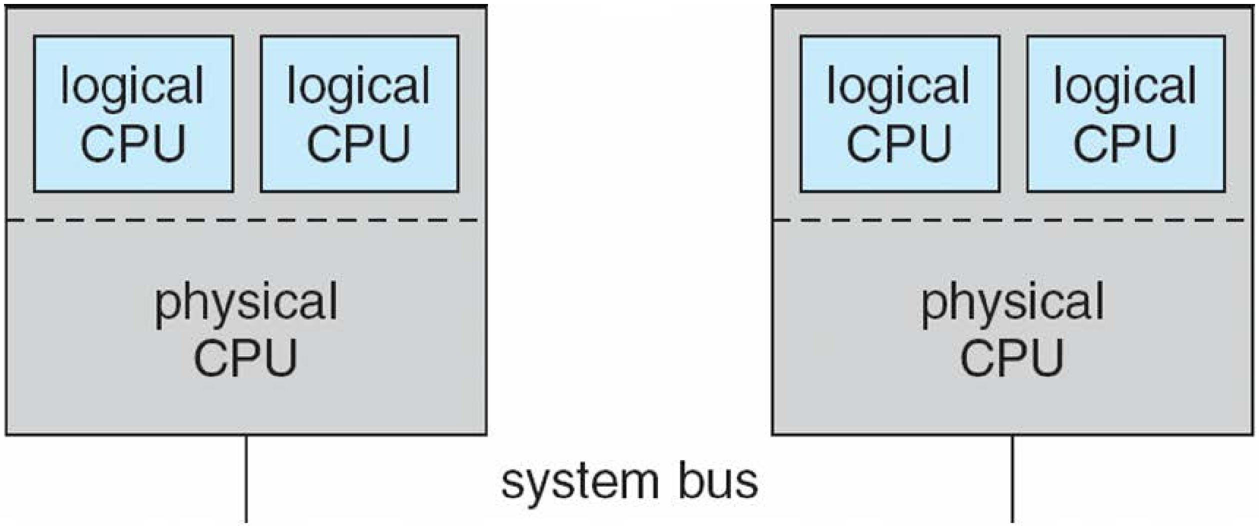
# Evaluation of CPU schedulers by Simulation



# End of Chapter 5

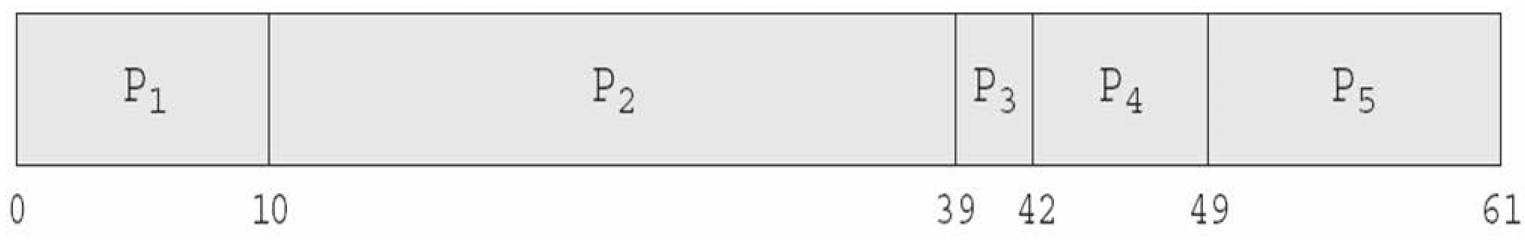


# 5.08

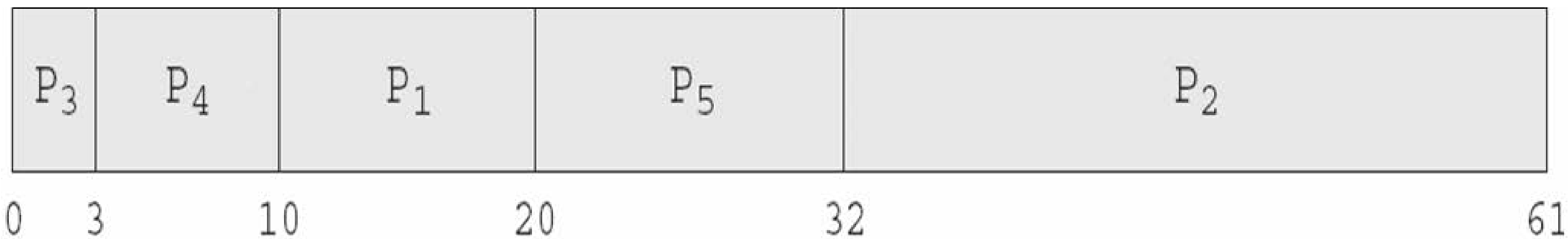


logical CPU
logical CPU
logical CPU
logical CPU
physical CPU
physical CPU
system bus

# In-5.7



# In-5.8

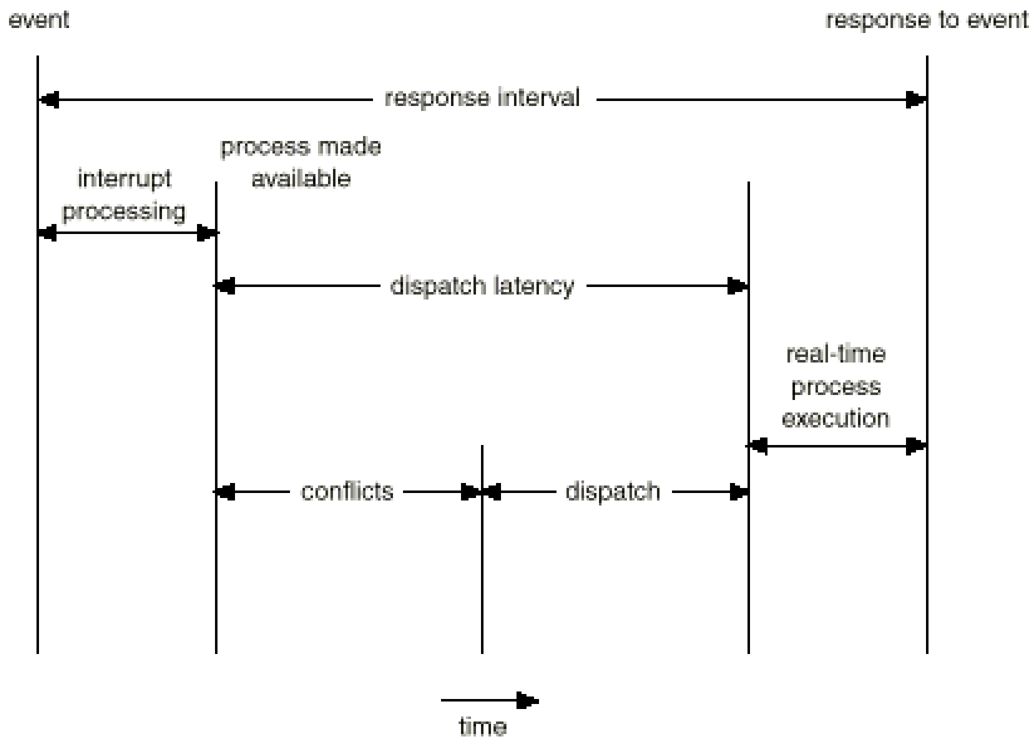


# In-5.9

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_2$ | $P_5$ | $P_2$ |
|---|---|---|---|---|---|---|---|

0 10 20 23 30 40 50 52 61

# Dispatch Latency



# Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

# Java Thread Scheduling (cont)

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

# Time-Slicing

Since the JVM Doesn't Ensure Time-Slicing, the yield() Method May Be Used:

```
while (true) {
        // perform CPU-intensive task
        . . .
        Thread.yield();
}
```

This Yields Control to Another Thread of Equal Priority

# Thread Priorities

| Priority | Comment |
|---|---|
| Thread.MIN_PRIORITY | Minimum Thread Priority |
| Thread.MAX_PRIORITY | Maximum Thread Priority |
| Thread.NORM_PRIORITY | Default Thread Priority |

Priorities May Be Set Using setPriority() method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```

# Solaris 2 Scheduling