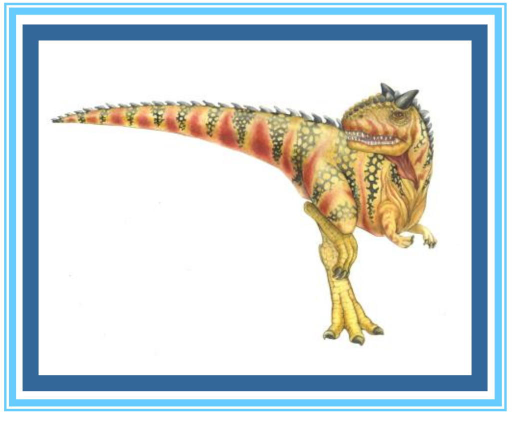
Chapter 6: CPU Scheduling



Chapter 6: CPU Scheduling

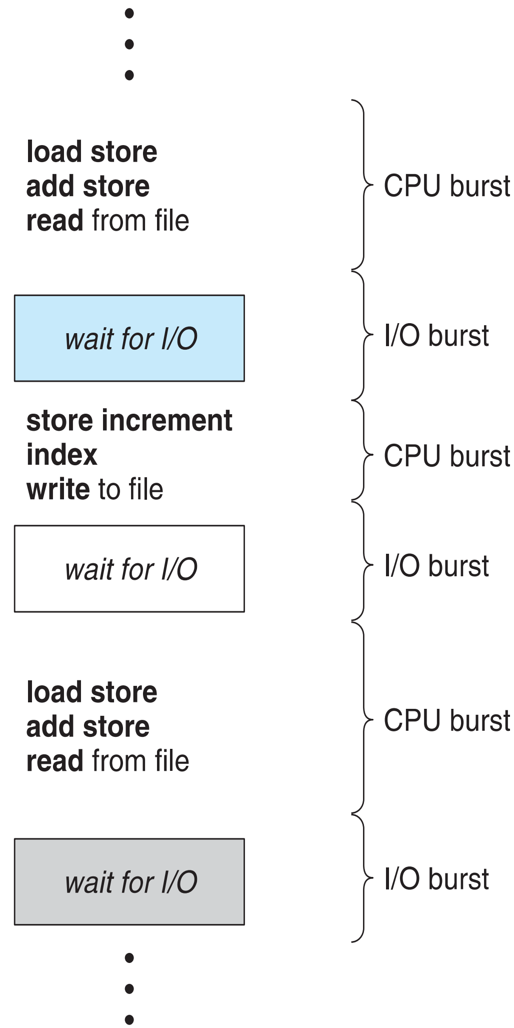
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

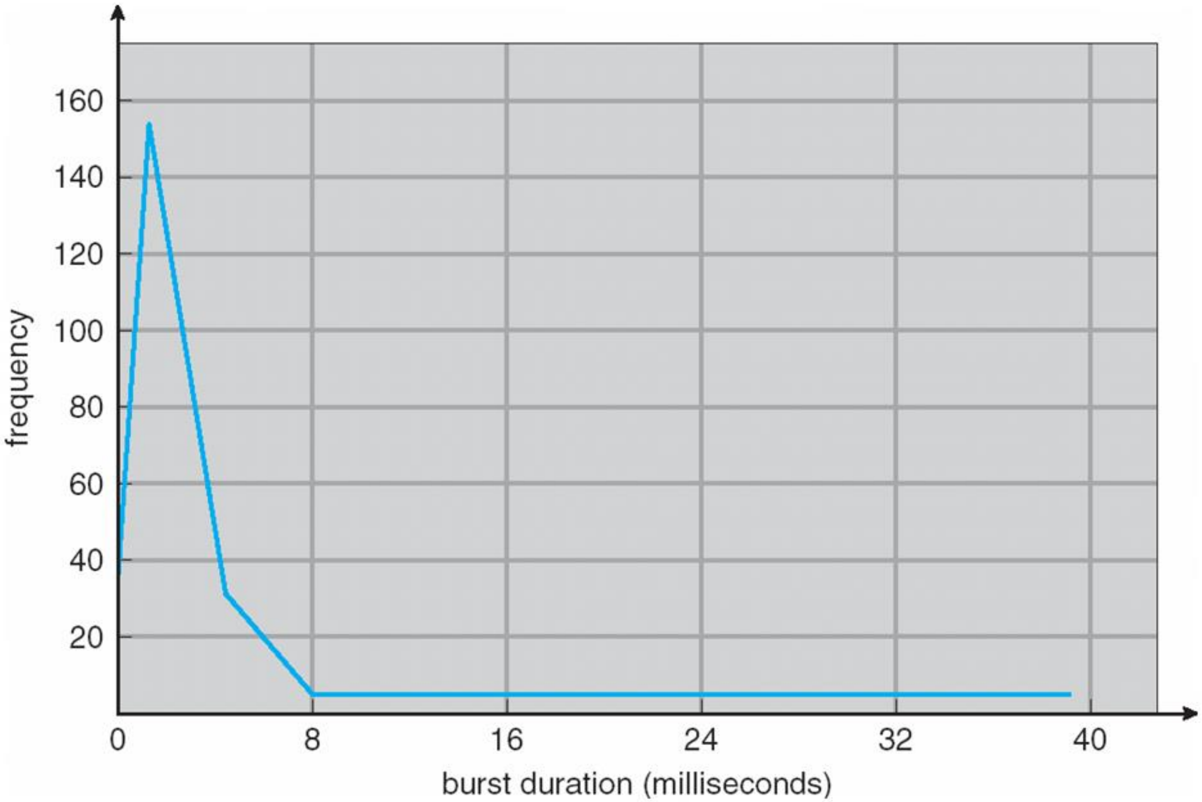
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



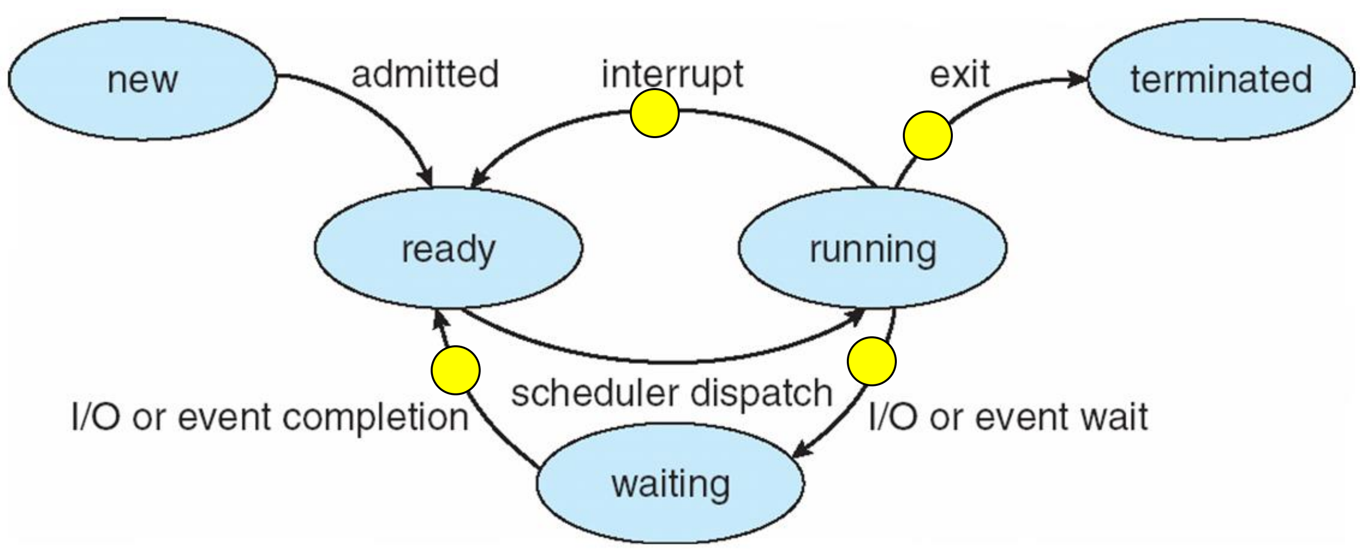
CPU Scheduler

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

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

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

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

Much better than previous case

Convoy effect - short process behind long process

Consider one CPU-bound and many I/O-bound processes

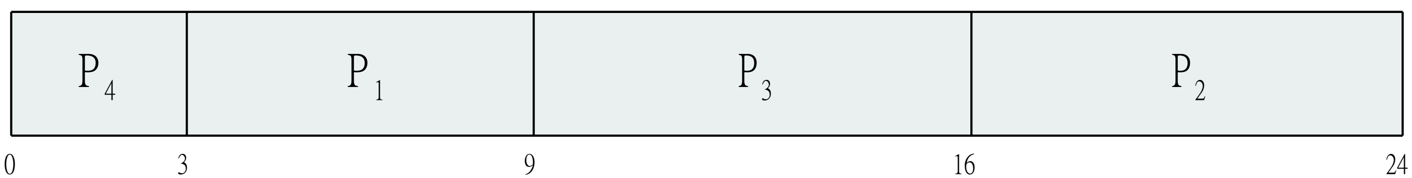
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- 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
  - Could ask the user

Example of SJF

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



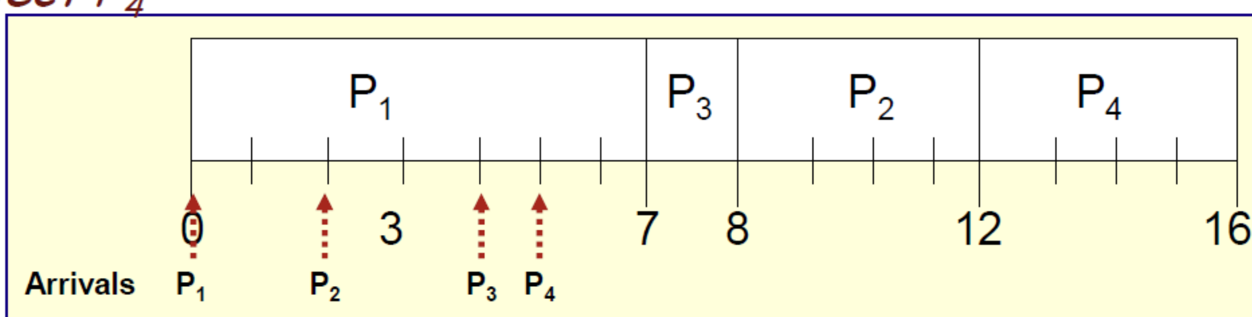
Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

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

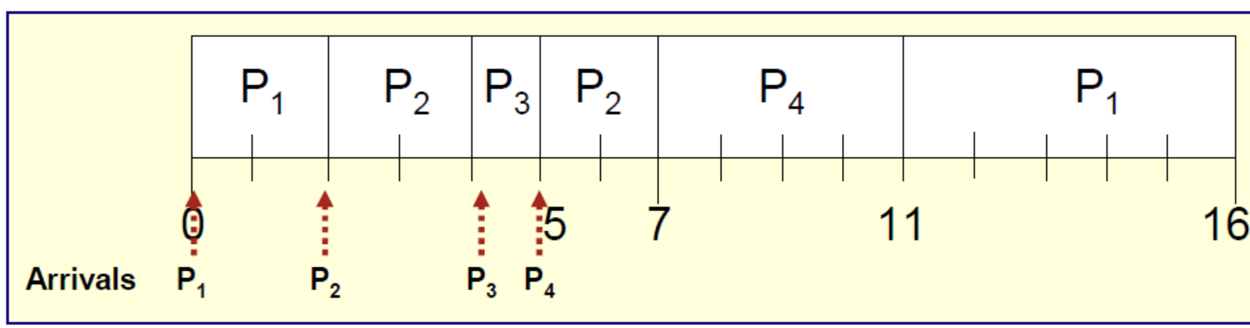


Example of Preemptive SJF

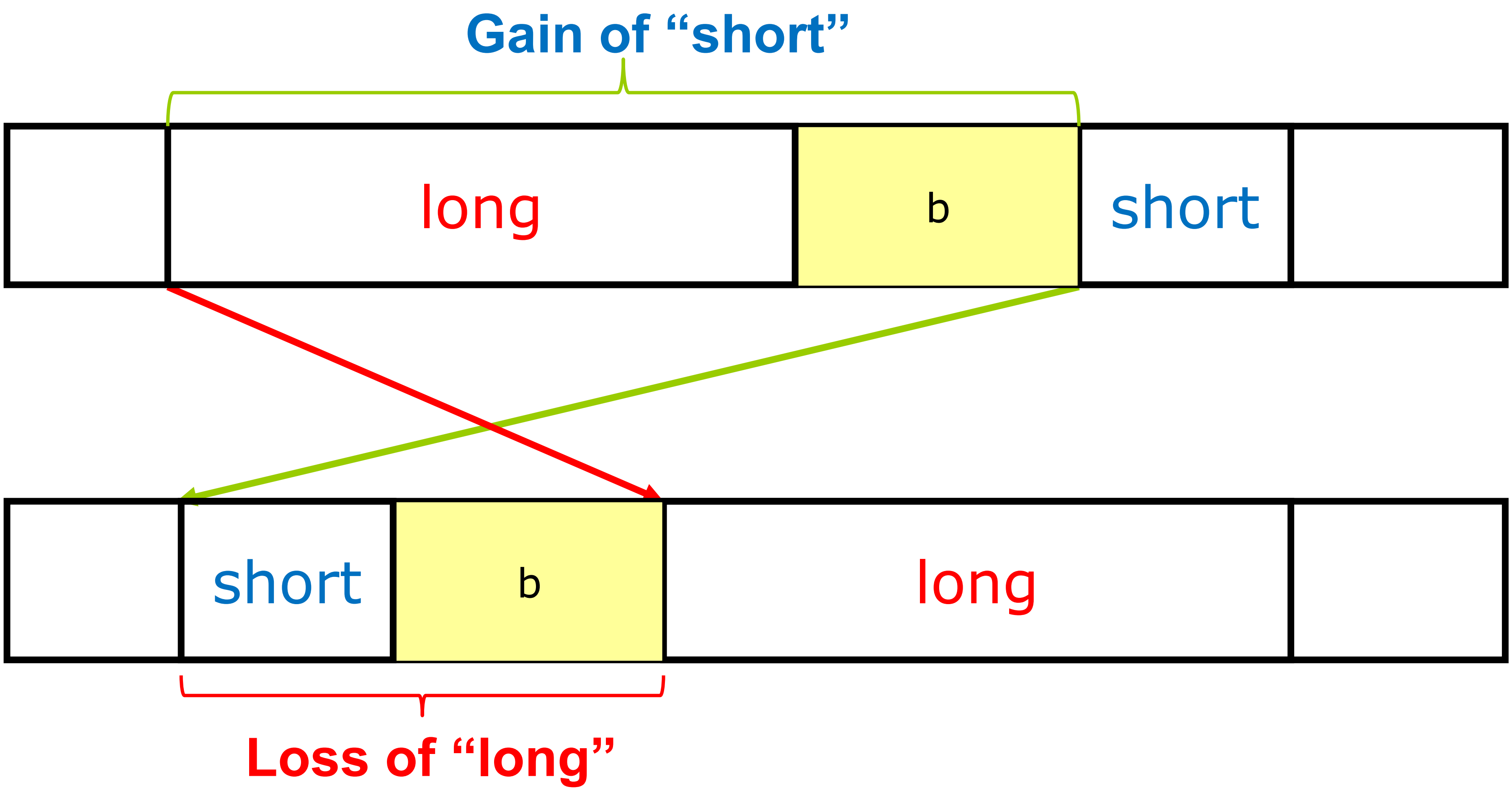
- T = 0: RQ = {P_1}
  Select P_1
- T = 2: RQ = {P_2}
  preempt P_1 , Select P_2
- T = 4: RQ = {P_3, P_1}
  preempt P_2 , Select P_3
- T = 5: RQ = {P_2, P_4 , P_1}
  P_3 completes, Select P_2
- T = 7: RQ = {P_4, P_1}
  P_2 completes, Select P_4
- T = 11: RQ = {P_1}
  P_4 completes, Select P_1
- T = 16: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



SJF Optimality



Proof that the SJF algorithm is optimal

Gain of short > Loss of long

Determining Length of Next CPU Burst

Can only estimate the length – should be similar to the previous one

Then pick process with shortest predicted next CPU burst

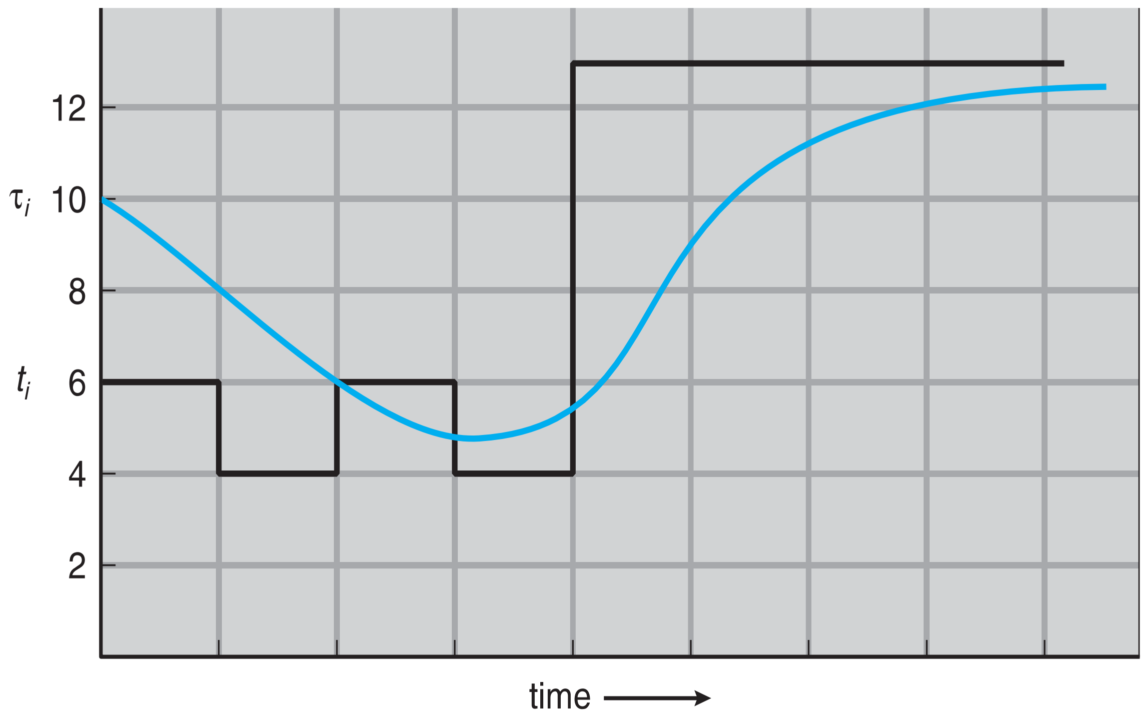
Can be done by using the length of previous CPU bursts, using exponential averaging

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 : $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n .$

Commonly, α set to ½

Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_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:

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

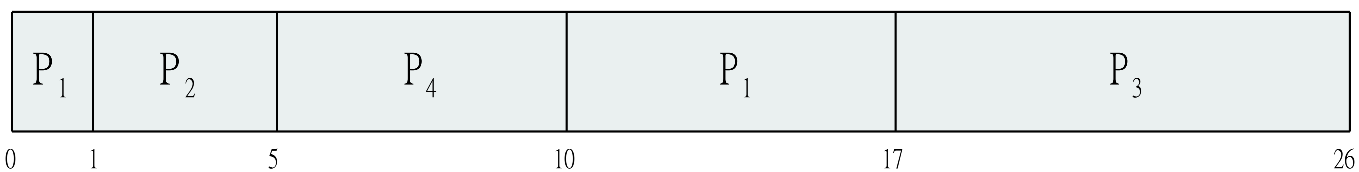
Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Preemptive SJF Gantt Chart



Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

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
- Nonpreemptive

SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

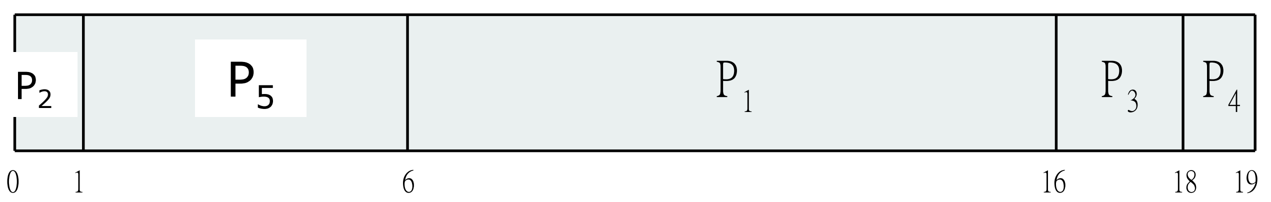
Problem ≡ **Starvation** – low priority processes may never execute

Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priority scheduling Gantt Chart



Average waiting time = 8.2 msec

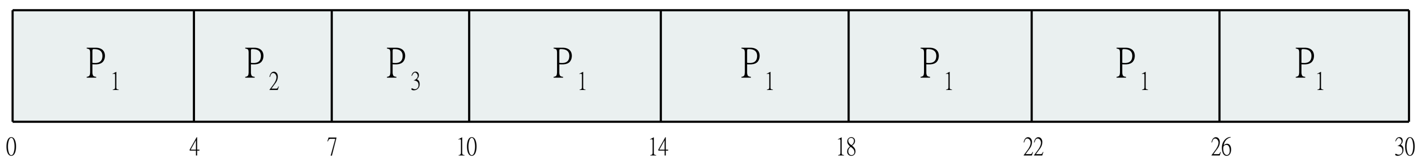
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *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:

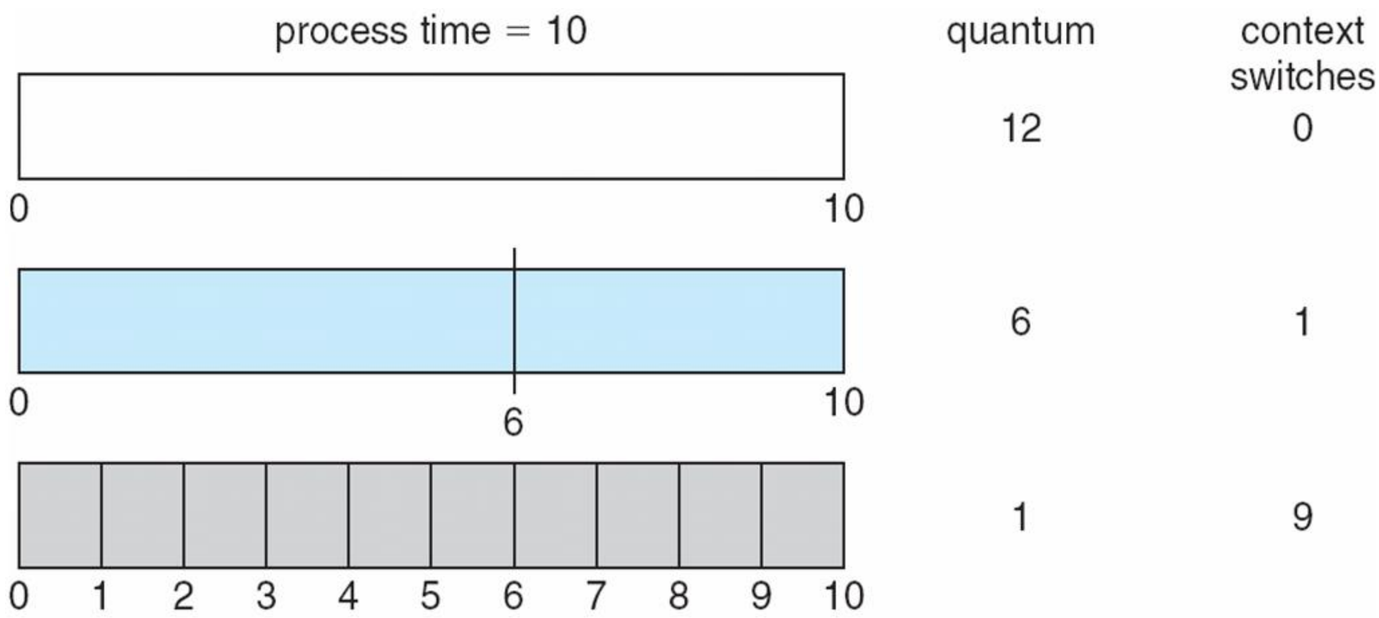


Typically, higher average turnaround than SJF, but better ***response***

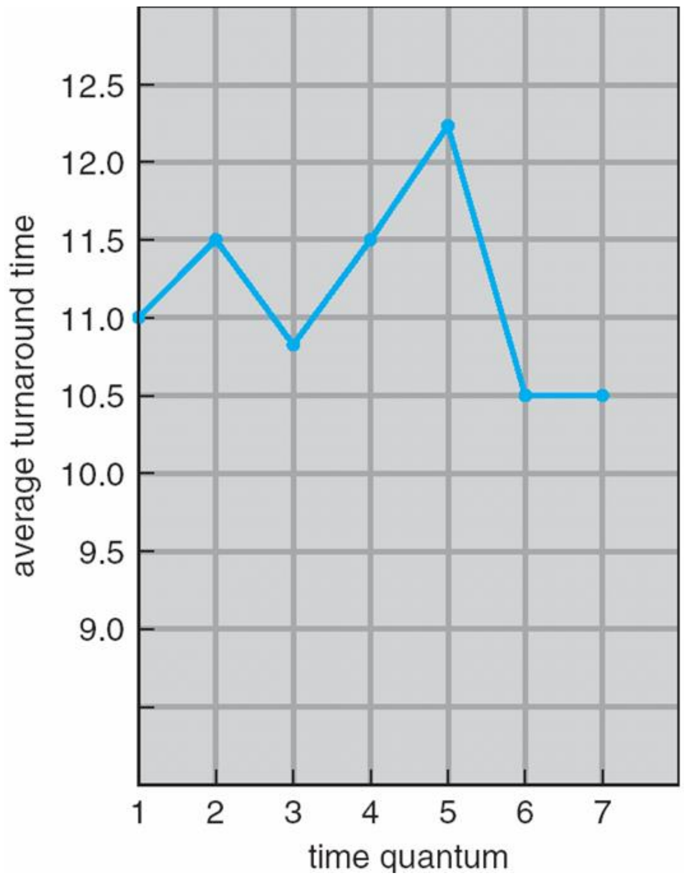
q should be large compared to context switch time

q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



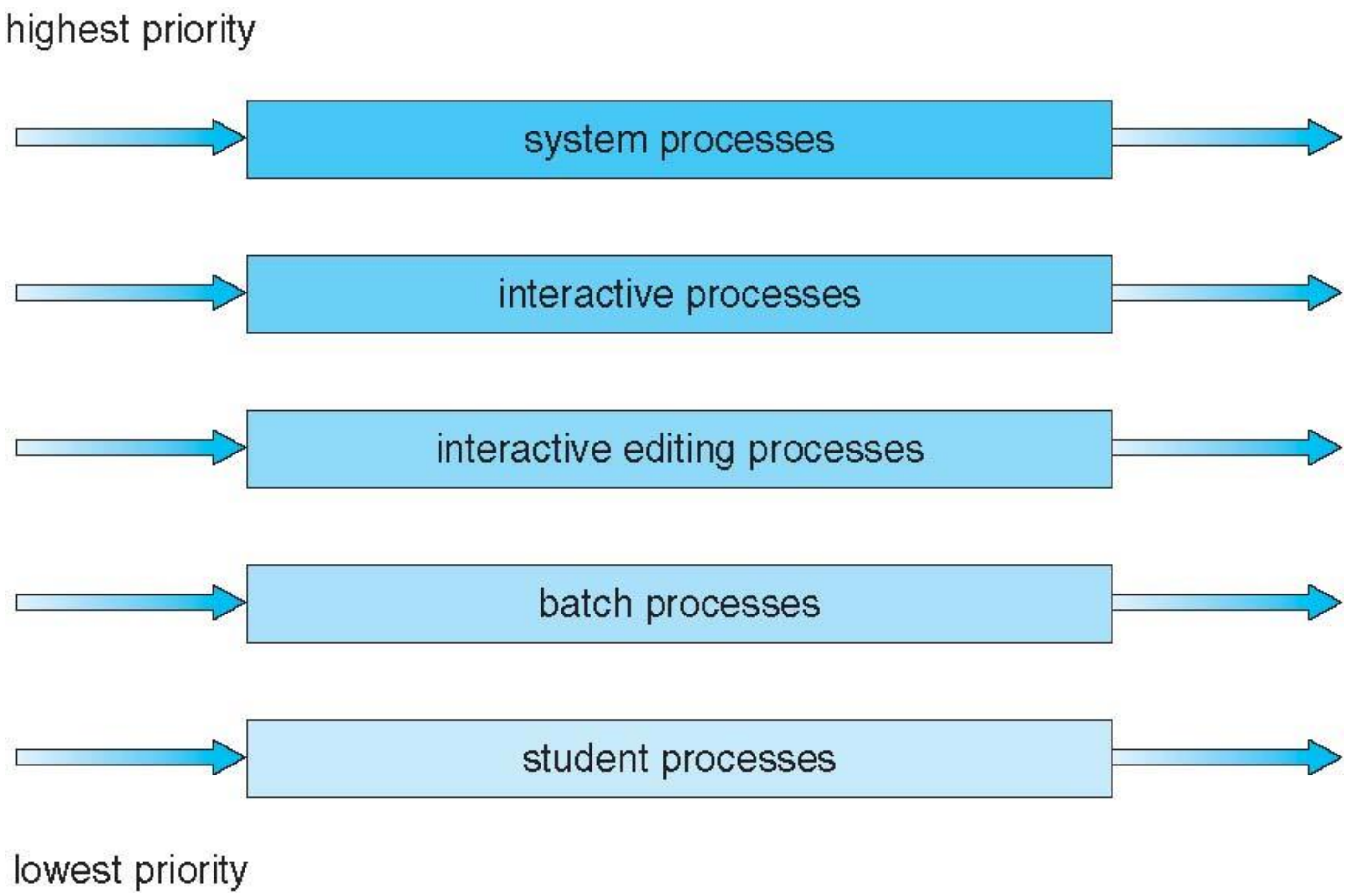
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- 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
  - Give preference to short jobs
  - Give preference to I/O bound processes
  - Separate processes into categories based on their need for the processor
- 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

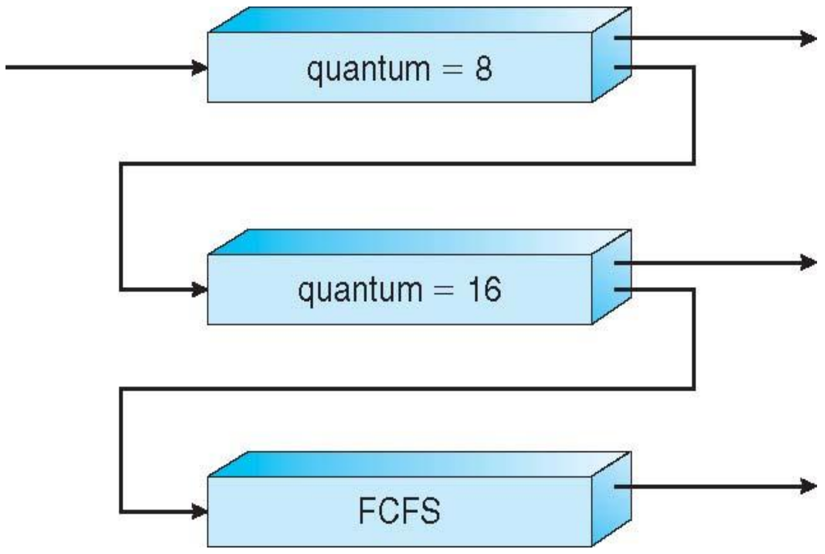
Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

Scheduling

- A new job enters queue Q_0 which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2



Multilevel Feedback Queue

A new process is inserted at the end (tail) of the top-level FIFO queue.

At some stage the process reaches the head of the queue and is assigned the CPU.

If the process is completed within the time quantum of the given queue, it leaves the system.

If the process voluntarily relinquishes control of the CPU, it leaves the queuing network, and when the process becomes ready again it is inserted at the tail of the same queue which it relinquished earlier.

If the process uses all the quantum time, it is pre-empted and inserted at the end of the next lower level queue. This next lower level queue will have a time quantum which is more than that of the previous higher level queue.

This scheme will continue until the process completes or it reaches the base level queue.

At the base level queue the processes circulate in round robin fashion until they complete and leave the system. Processes in the base level queue can also be scheduled on a first come first served basis.

Optionally, 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.

(https://en.wikipedia.org/wiki/Multilevel_feedback_queue)

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- 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
  - Typically done via priority set by programmer
- 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
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

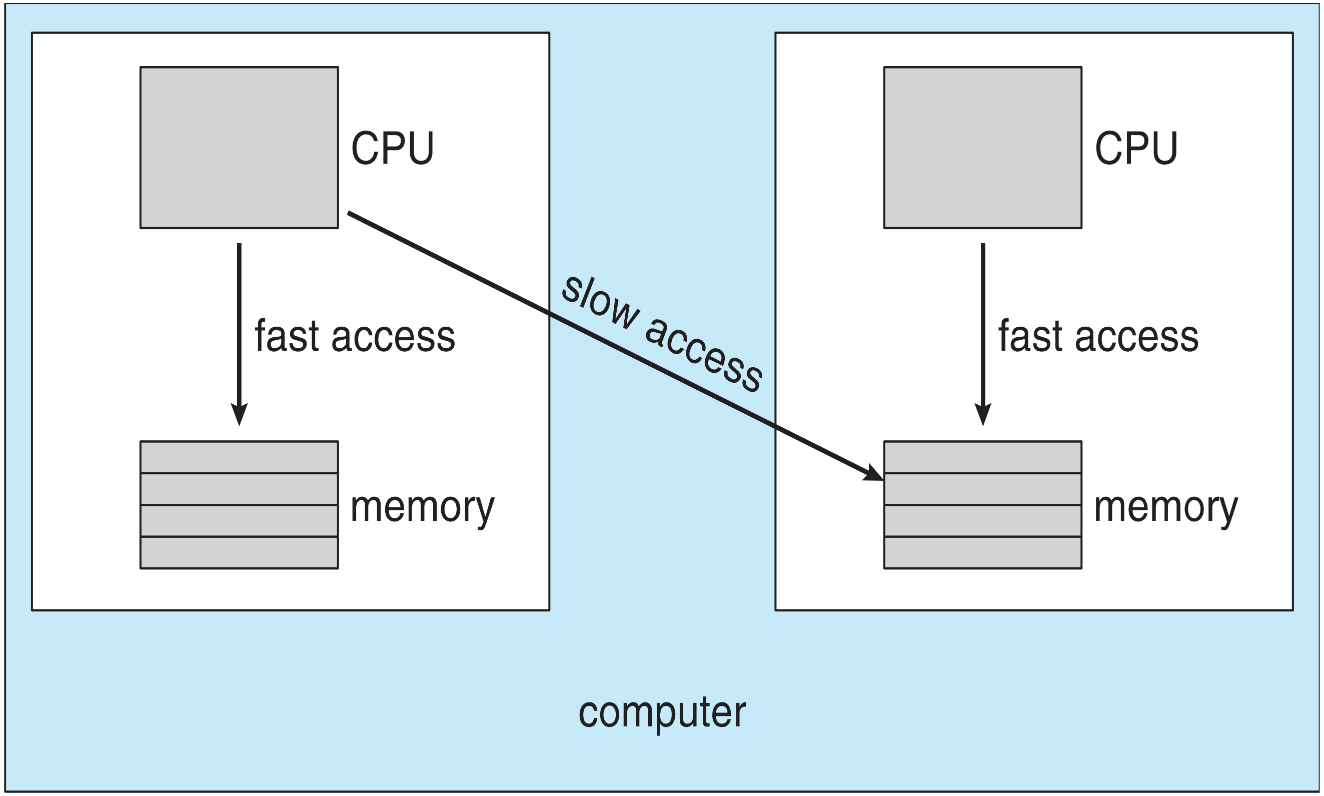
Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* 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)
{
    /* do some work ... */
    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
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

NUMA and CPU Scheduling

https://www.itread01.com/articles/1493819836.html

```
#numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
node 0 size: 16290 MB
node 0 free: 11947 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
node 1 size: 16384 MB
node 1 free: 14282 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

此系統共有2個node，各領取16個CPU和16G內存。

這裏假設我要執行一個Java param命令，此命令需要12G內存；一個python param命令，需要16G內存。
最好的優化方案時python在node0中執行，而java在node1中執行，那命令是：

```
#numactl --cpubind=0 --membind=0 python param
#numactl --cpubind=1 --membind=1 java param
```

當然，也可以自找沒趣進行如下配置：

```
#numactl --cpubind=0 --membind=0,1 java param
```

通過numastat命令可以查看numa狀態
註：numastat - Show per-NUMA-node memory statistics for processes and the operating system

```
# numastat
                           node0           node1
numa_hit             61086587932     25494360922
numa_miss              101325832     28581785059
numa_foreign         28581785059       101325832
interleave_hit             28949           28518
local_node           61086561129     25494416828
other_node             101352635     28581729153
```

other_node過高意味著需要重新規劃numa.

Multiple-Processor Scheduling – Load Balancing

If SMP, need to keep all CPUs loaded for efficiency

Load balancing attempts to keep workload evenly distributed

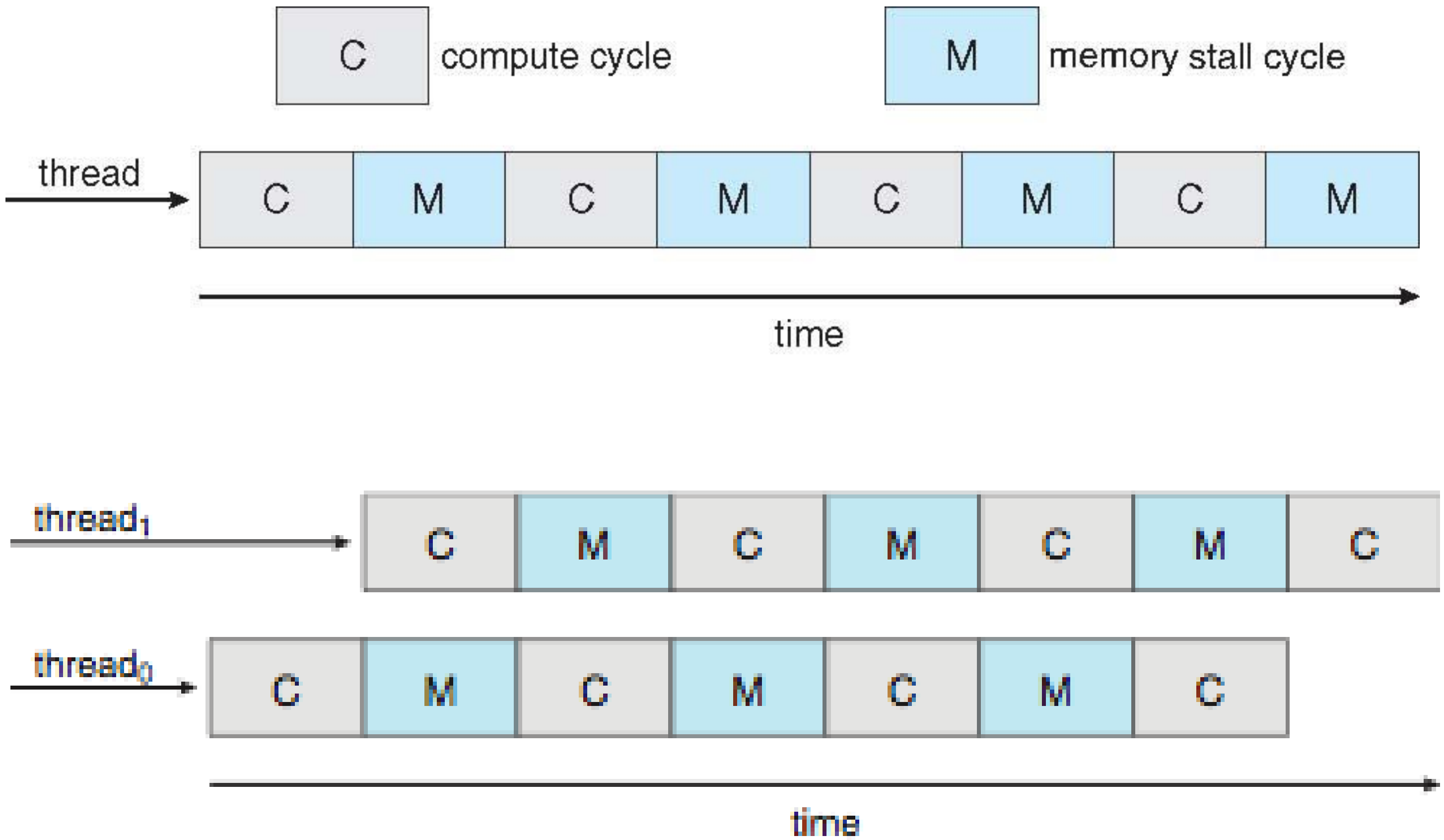
Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs

Pull migration – idle processors pulls waiting task from busy processor

Multicore Processors

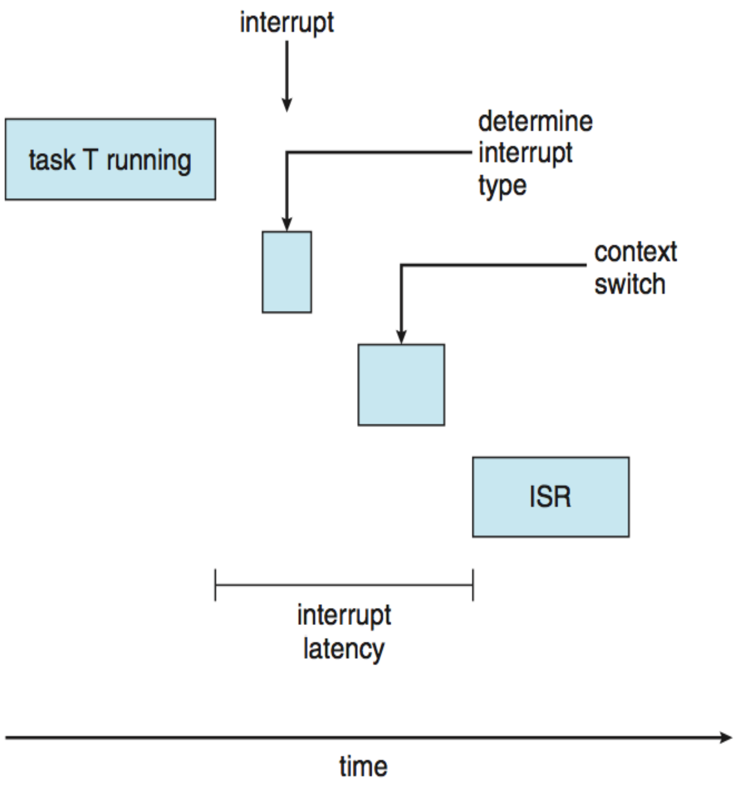
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



Real-Time CPU Scheduling

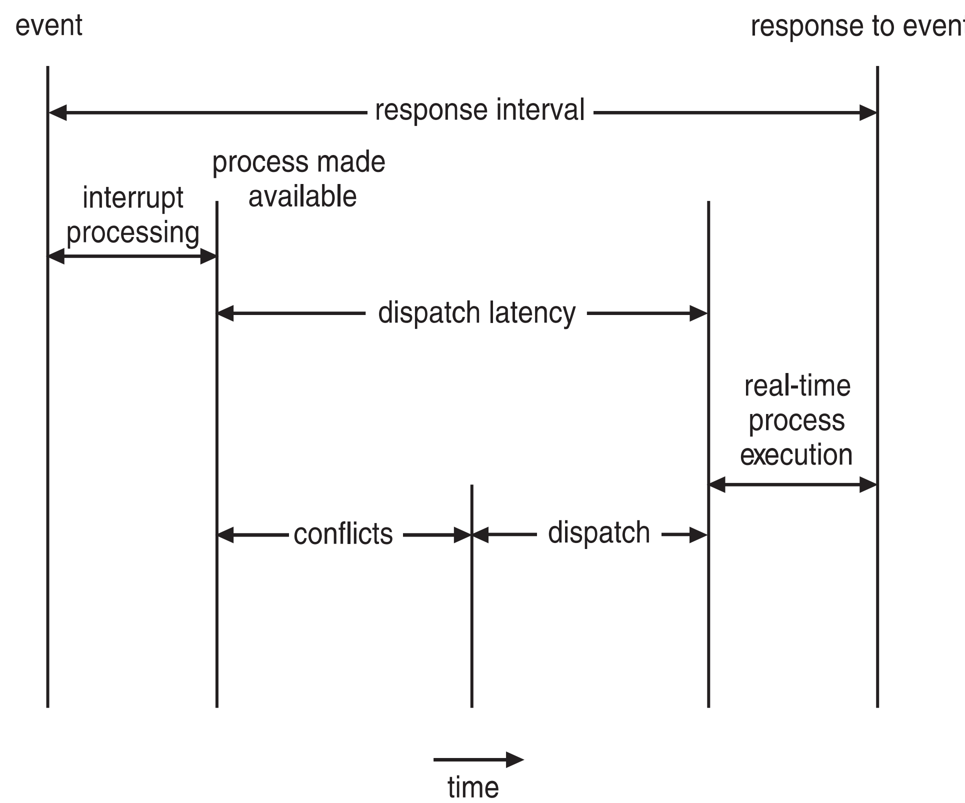
- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Real-Time CPU Scheduling (Cont.)

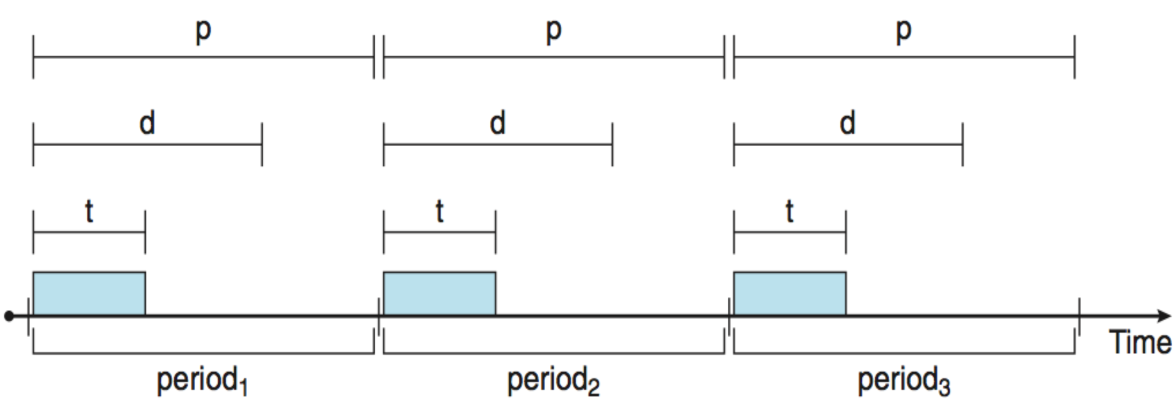
Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time t, deadline d, period p
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



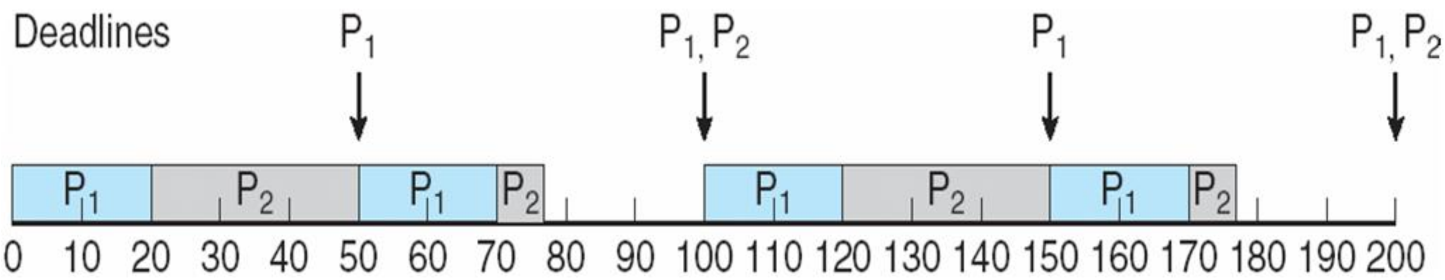
Rate Monotonic Scheduling

A priority is assigned based on the inverse of its period

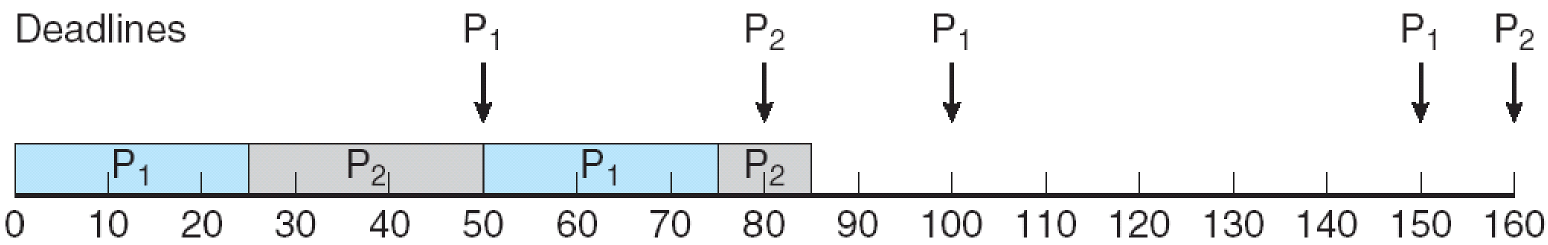
Shorter periods = higher priority;

Longer periods = lower priority

P_1 is assigned a higher priority than P_2.



Missed Deadlines with Rate Monotonic Scheduling



Feasible schedule exists if $U := \sum_{i=1}^{n} \frac{t_i}{p_i} \leq n\left(2^{1/n} - 1\right)$

$0.5 + \frac{(25 + 10)}{80} = 0.9375$ $U = 2 \cdot \left(2^{0.5} - 1\right) = 0.8284$

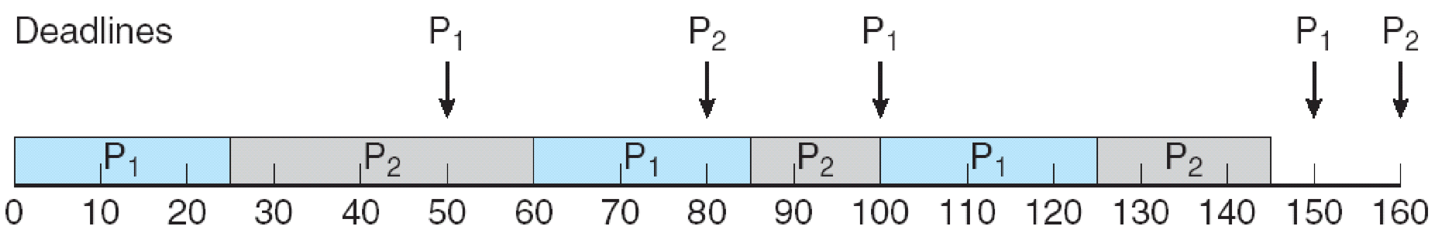
*Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, **20** (1): 46–61,*

Earliest Deadline First Scheduling (EDF)

Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;

the later the deadline, the lower the priority



EDF can schedule task set iff $U \leq 1$

Proportional Share Scheduling

T shares are allocated among all processes in the system

An application receives N shares where $N < T$

This ensures each application will receive $\boldsymbol{N} / T$ of the total processor time

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:

1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority

- Defines two functions for getting and setting scheduling policy:

1. **`pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`**
2. **`pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`**

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t_tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
       fprintf(stderr, "Unable to set policy.\n");
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* 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)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn't own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

Operating System Examples

Linux scheduling

Windows scheduling

Solaris scheduling

Linux Scheduling Through Version 2.5

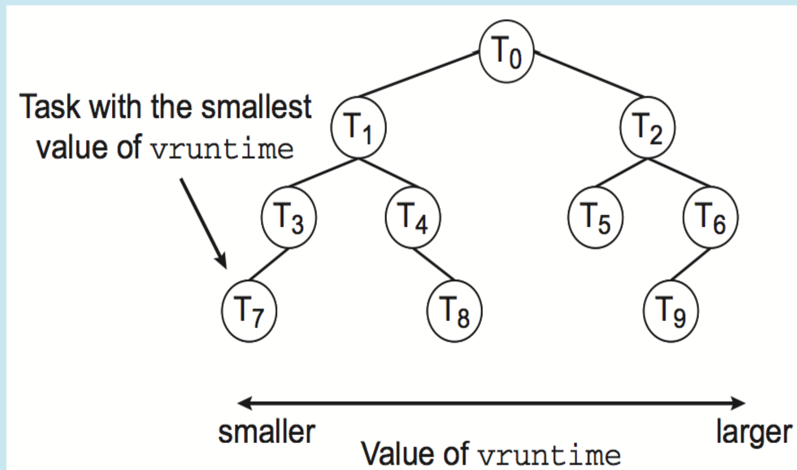
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order $O(1)$ scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux CFS

kernel/sched/core.c (v5.0.9)

```
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{i=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$

Linux Scheduling

```
#include <sched.h>

int sched_setattr(pid_t pid, struct sched_attr *attr,
                  unsigned int flags);

int sched_getattr(pid_t pid, struct sched_attr *attr,
                  unsigned int size, unsigned int flags);
```

SCHED_OTHER the standard round-robin time-sharing policy (CFS)
SCHED_BATCH for "batch" style execution of processes
SCHED_IDLE for running very low priority background jobs

SCHED_FIFO a first-in, first-out policy; and
SCHED_RR a round-robin policy.
SCHED_DEADLINE EDF-based

Real-time

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL        0
#define SCHED_FIFO          1
#define SCHED_RR            2
#define SCHED_BATCH         3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE          5
#define SCHED_DEADLINE      6
```

include/uapi/linux/sched.h (v5.0.9)

Linux Scheduling (Cont.)

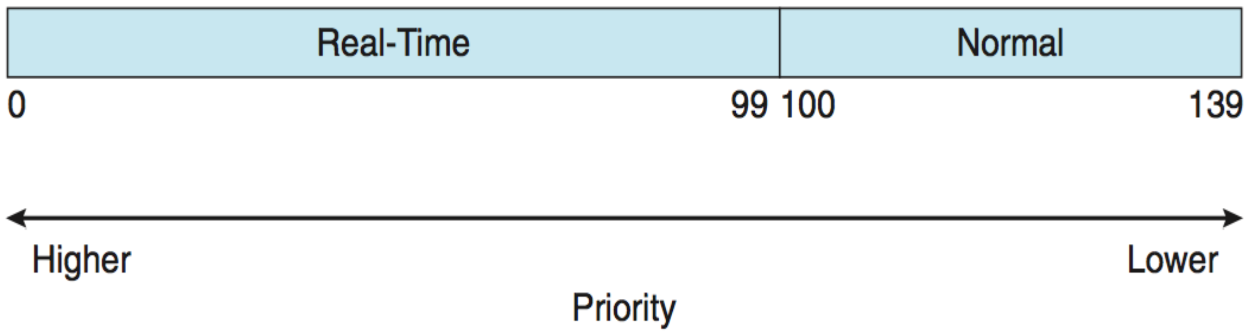
Real-time scheduling according to POSIX.1b

- Real-time tasks have static priorities

Real-time plus normal map into global priority scheme

Nice value of -20 maps to global priority 100

Nice value of +19 maps to priority 139



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows 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

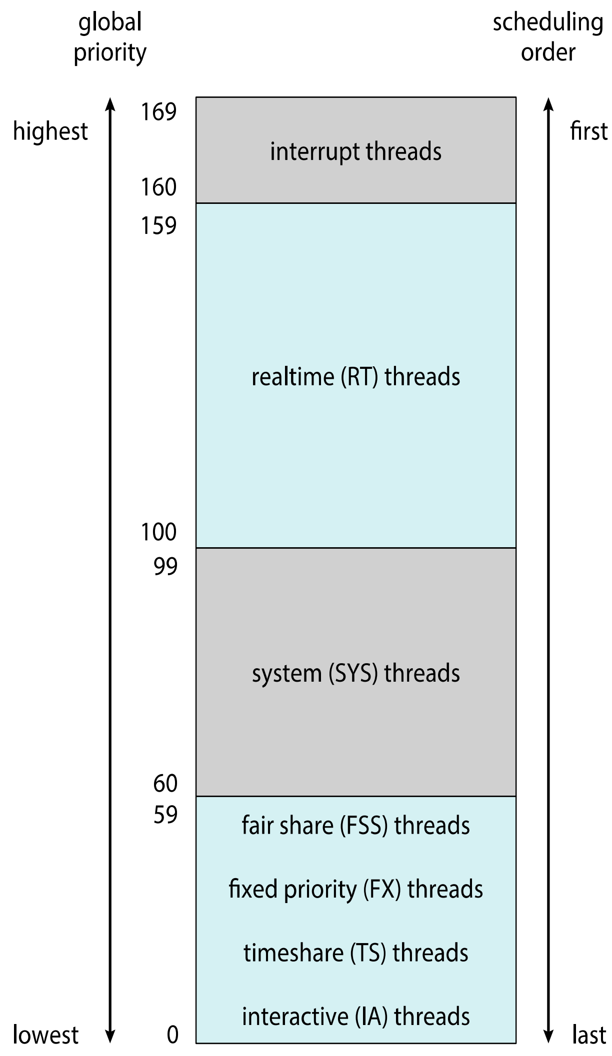
Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

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



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

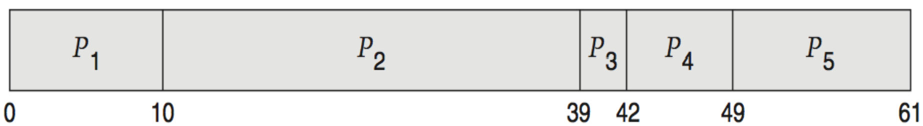
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

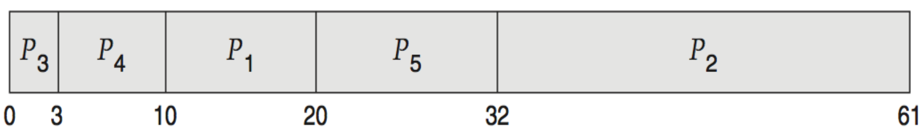
For each algorithm, calculate minimum average waiting time

Simple and fast, but requires exact numbers for input, applies only to those inputs

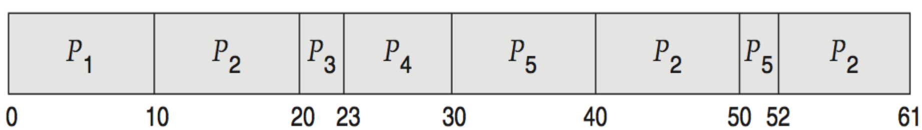
FCS is 28ms:



Non-preemptive SFJ is 13ms:



RR is 23ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula

n = average queue length

W = average waiting time in queue

λ = average arrival rate into queue

Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

$n = \lambda \times W$

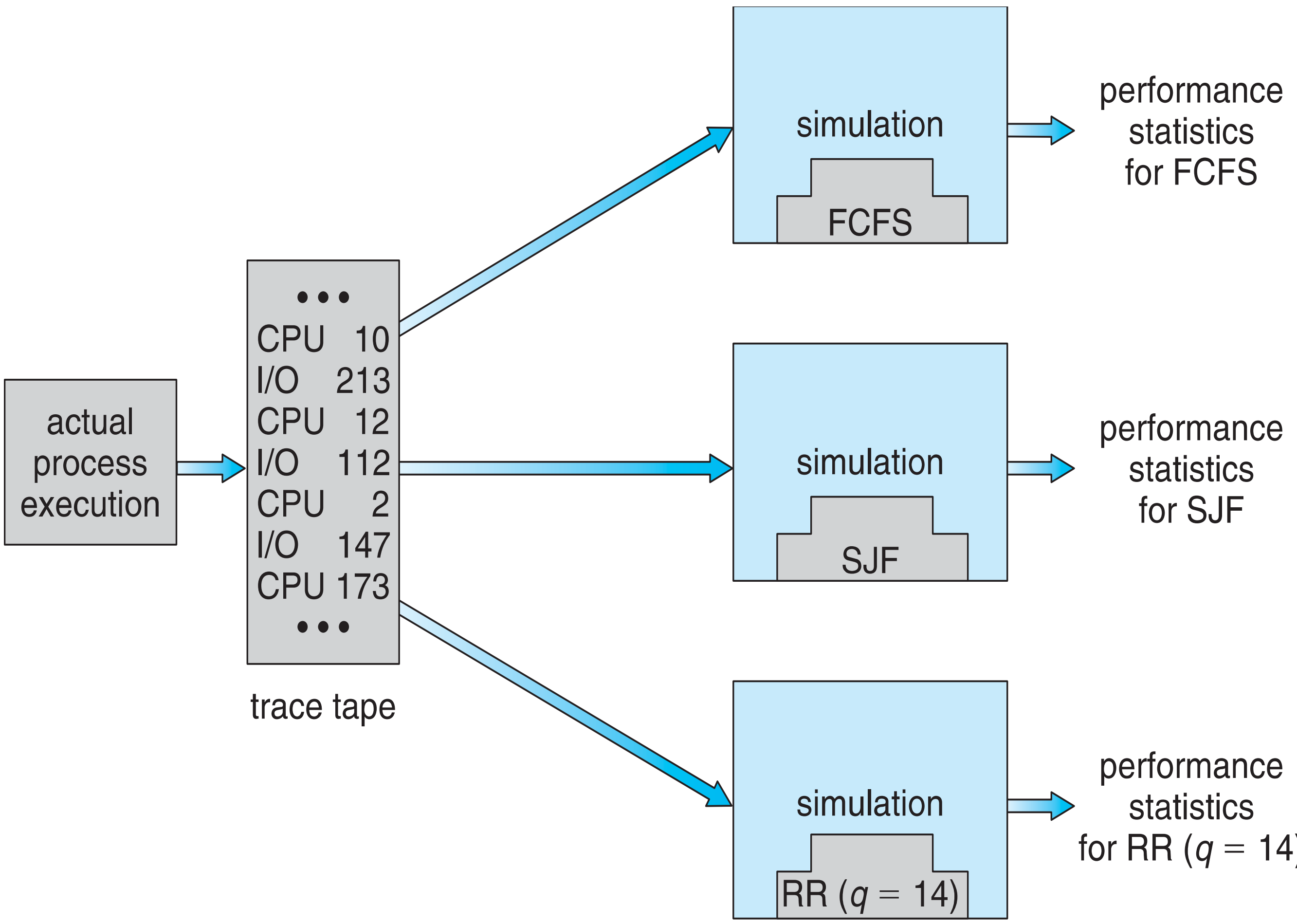
Valid for any scheduling algorithm and arrival distribution

For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Chapter 6