G22.2250-001
Operating Systems

Lecture 5

Language Support for Synchronization
CPU Scheduling

October 2, 2007

Outline

Announcements

- Lab 2 due this Friday: Please see me after class if you have not yet started this lab
- Homework 1: Turn-in is optional
  - Covering material covered in Lectures 1-4, first half of today's lecture
- Office hours on October 9th from 4:00 – 5:00pm

- Language support for synchronization
  - Conditional critical regions
  - Monitors

- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms
  - example: Windows XP scheduler
  - advanced topic: Real-time scheduling

[Silberschatz/Galvin/Gagne: Sections 6.8, 5.1 – 5.3, 5.6 – 5.7]

(Review)

Conditional Critical Regions

- A high-level language declaration
  - informally, it can be used to specify that while a statement *S* is being executed, no more than one process can access a distinguished variable *v*
  - notation

```
var v: shared t;
region v when B do S;
```

    - ***v*** is shared and of type *t*
      - can only be accessed within a region statement
    - ***B*** is a Boolean expression
    - ***S*** is a statement
      - can be a compound statement

- Semantics
  - A process is guaranteed mutually exclusive access to the region `v`
  - Checking of `B` and entry into the region happens atomically

(Review)

Conditional Critical Regions: Benefits

Bounded-buffer producer/consumer

```
var buffer : shared record
   pool: array [0..n-1] of item;
   count, in, out: integer;
end;

Producer:
region buffer when count < n
  do begin
     pool[in] := nextp;
     in := (in + 1) mod n;
     count := count + 1;
  end;

Consumer:
region buffer when count > 0
  do begin
     nextc := pool[out];
     out := (out + 1) mod n;
     count := count - 1;
  end;
```

- Guards against simple errors associated with semaphores
  - e.g., changing the order of P and V operations, or forgetting to put one of them
- Division of responsibility
  - the *developer* does not have to program the semaphore or alternate synchronization explicitly
  - the *compiler* ``automatically'' plugs in the synchronization code using predefined libraries
  - once done carefully, *reduces* likelihood of mistakes in designing the delicate synchronization code

Conditional Critical Regions: Implementation

```
var mutex: semaphore;

P( mutex );
while not B
   do begin
        try-and-enter;
   end;
S;
leave-critical-region;
```

```
var delay: semaphore;
var count: integer;

count++ ;
V( mutex );
P( delay );
// check condition
if ( not B )
  if ( count > 1 )
    // release another
    V( delay );
    P( delay );
  else
    V( mutex );
    P( delay );
else count-- ;

if ( count > 0 )
   V( delay );
else V( mutex );
```

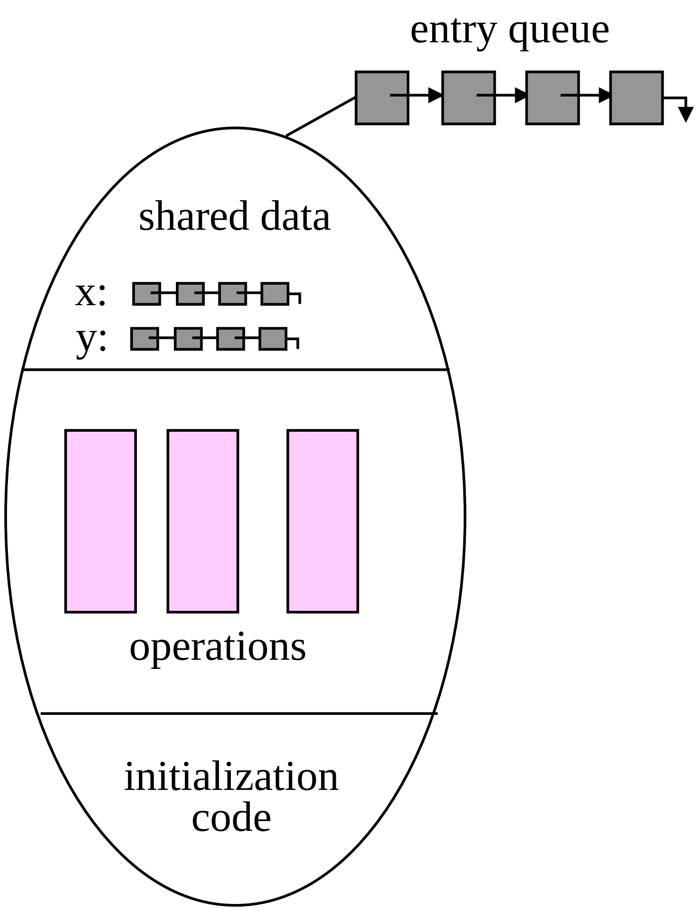
```
var first, second: semaphore;
var fcount, scount: integer;

fcount++ ;
if ( scount > 0 ) V( second );
else V( mutex );
P( first );
fcount-- ;
scount++ ;
if ( fcount > 0 ) V( first );
else V( second );
P( second );
scount-- ;

if ( fcount > 0 ) V( first );
else if ( scount > 0 ) V( second );
else V( mutex );
```

Language Support (2): Monitors

- An abstract data type
  - private data
  - public procedures
    - only one procedure can be in the monitor at one time
    - each procedure may have
      - local variables
      - formal parameters
  - condition variables
    - queues of processes
    - *wait*: block on a condition variable
    - *signal*: unblock a waiting process
      - no-op if no process is waiting
- Processes can only invoke the public procedures
  - raises the granularity of atomicity to a single user-defined procedure



Waiting in the Monitor

- Note that the semantics of executing a *wait* in the monitor is that several processes can be waiting "inside" the monitor at any given time but only one is executing
  - wait queues are internal to the monitor
  - there can be multiple wait queues

- Who executes after a signal operation? (say P signals Q)
  - (Hoare semantics) signallee Q continues
    - logically natural since the condition that enabled Q might no longer be true when Q eventually executes
      - P needs to wait for Q to exit the monitor
  - (Mesa semantics) signaller P continues
    - Q is enabled but gets its turn only after P either leaves or executes a *wait*
  - require that the *signal* be the last statement in the procedure
    - advocated by Brinch Hansen (Concurrent Pascal)
    - easy to implement but less powerful than the other two semantics

Use of Monitors: Bounded-buffer

```
type bounded_buffer = monitor

  var buffer: array [0..N] of char;
  var in, out, count: integer;
  var notfull, notempty: condition;

  procedure entry append ...
  procedure entry remove ...

  begin
   in = 0; out = 0; count = 0;
  end;
```

```
procedure entry append(x: char);
   if (count==N) notfull.wait;
   buffer[in] := x;
   in := (in+1) mod N;
   count := count+1;
   notempty.signal;

procedure entry remove(x: char);
   if (count==0) notempty.wait;
   x := buffer[out];
   out := (out+1) mod N;
   count := count-1;
   notfull.signal;
```

Is this solution correct under all monitor semantics? (P signals Q)

- Hoare: Q continues, P suspends .... YES
- Mesa: P continues, Q is put into ready queue .... NO
- Brinch-Hansen: P exits monitor, Q continues ....YES

Use of Monitors: Bounded-buffer (Mesa Semantics)

```
type bounded_buffer = monitor

  var buffer: array [0..N] of char;
  var in, out, count: integer;
  var notfull, notempty: condition;

  procedure entry append ...
  procedure entry remove ...

  begin
    in = 0; out = 0; count = 0;
  end;
```

```
procedure entry append(x: char);
   while (count==N) notfull.wait;
   buffer[in] := x;
   in := (in+1) mod N;
   count := count+1;
   notempty.signal;

procedure entry remove(x: char);
   while (count==0) notempty.wait;
   x := buffer[out];
   out := (out+1) mod N;
   count := count-1;
   notfull.signal;
```

Use of Monitors: Dining Philosophers

- Goal: Solve DP without deadlocks

- Informally:
  - algorithm for Philosopher I

    ```
    dp.pickup(i);
    eat;
    dp.putdown(i);
    ```

  - use array to describe state

    ```
    var state: array [0..4] of
       (thinking, hungry,
      eating);
    ```

  - use array of condition variables to block on when required resources are unavailable

    ```
    var self: array [0..4] of
      condition;
    ```

- pickup(i)
  - changes state to hungry
  - checks if neighbors are eating
  - if not, grabs chopsticks, and changes state to eating
  - otherwise, waits on self(i)

- putdown(i)
  - checks both neighbors
  - if either is hungry and can proceed, releases him/her

Dining Philosophers using Monitors - 2

```
type dining_philosophers = monitor

  var state: array [0..4] of
    (thinking, hungry, eating);
  var self: array [0..4] of
    condition;

  procedure entry pickup ...
  procedure entry putdown ...
  procedure test ...

  begin
    for i := 0 to 4 do
      state[i] := thinking;
  end;
```

```
procedure entry pickup(i: 0..4);
     state[i] := hungry;
     test(i);
     while ( state[i] != eating )
        self[i].wait;

procedure entry putdown(i: 0..4);
     state[i] := thinking;
     test (ln(i));
     test (rn(i));

procedure test(i: 0..4);
     if (state[ln(i)] != eating and
         state[i] == hungry and
         state(rn(i)) != eating)
       state[i] := eating;
       self[i].signal;
```

Dining Philosophers using Monitors - 3

- What is missing?
  - philosophers cannot deadlock but can starve
    - for example, we can construct timing relationships such that a waiting philosopher will be stuck in the “self” queue forever
  - monitors have to be enhanced with a fair scheduling policy to avoid starvation
    - both at the level of accessing the monitor
    - as well as to regulate “waking-up”" those that are waiting inside
  - how can this be done?
    - use fair enqueue and dequeue policies

Monitors: Other Issues

- Expressibility: Are monitors more/less powerful than semaphores or conditional critical regions?
  - these three constructs are equivalent
    - the same kinds of synchronization problems can be expressed in each
  - the other two can be implemented using any one of the constructs
    - e.g., critical regions and monitors using semaphores
      - we talked about how critical regions can be implemented
      - in Lab 2: you built condition variables using semaphores
        - » this implementation can be extended to build monitors

- Do monitors have any limitations?
  - absence of concurrency within a monitor
    - workarounds introduce all the problems of semaphores
    - monitor procedures will need to be invoked before and after
    - possibility of improper access, deadlock, etc.

Outline

Announcements

- Lab 2 due this Friday: Please see me after class if you have not yet started this lab
- Homework 1: Turn-in is optional
  - Covering material covered in Lectures 1-4, first half of today's lecture
- Office hours on October 9th from 4:00 – 5:00pm

- Language support for synchronization
  - Conditional critical regions
  - Monitors

- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms
  - example: Windows XP scheduler
  - advanced topic: Real-time scheduling

[Silberschatz/Galvin/Gagne: Sections 6.8, 5.1 – 5.3, 5.6 – 5.7]

CPU Scheduling: Overview

- What is scheduling?
  - Deciding which process to execute and for how long

- Why do we need it?
  - Better resource utilization
  - Improve the system performance for desired load pattern
  - Support multitasking for interactive jobs
    - Example: Editing and compiling
  - Enable providing of specific guarantees

Scheduling: Components

- Processes

- Scheduler
  - focus on *short-term scheduling* (of the CPU)
  - decide which process to give the CPU to next
    - rationale: utilize CPU resource better
    - can also be necessary because of other factors: fairness, priorities, etc.

- Dispatcher:
  - suspends previous process and (re)starts new process
    - context switch, including adjusting and updating the various process queues
    - switch to user mode from the scheduler's supervisor mode
    - jump to the appropriate point in user space and resume executing "running" process

Scheduling: Operation Details

- (Review) Queues associated with process states
  - Running, Ready, Waiting

- Scheduler invoked in the following situations (triggers)
  - process switches from **running** to **waiting** state
    - e.g., block for I/O, wait for child
  - process switches from **running** to **ready** state
    - e.g., expiration of timer
  - process switches from **waiting** to **ready** state
    - e.g., completion of I/O
  - process **terminates**

Preliminaries: Model of Process Behavior

- CPU versus I/O bursts
  - a given process' behavior is broken into
    - a run of activity on the CPU referred to as a *CPU burst*
    - a run of non-CPU (usually I/O) activity or an *I/O burst*
  - the overall execution of a process is alternating CPU and I/O bursts
  - CPU burst lengths typically characterized as *exponential* or *hyperexponential*
    - CPU bound processes: few, long CPU bursts
    - I/O bound processes: many, very-short CPU bursts

	CPU	IO	CPU	IO	CPU
Process 1	10	1000	15	4000	5
Process 2	20	2	20	2	20

Preliminaries: Preemption

- *Preemptive* versus *non-preemptive* scheduling
  - the corresponding scheduling *policy* is non-preemptive
    - if a process switches to a waiting state *only* as a function of its own behavior
      - i.e. when it invokes OS services, or when it terminates
  - it is preemptive
    - if its state can be switched otherwise

- Cost of preemption: Maintaining consistent system state while the processes are suspended in the midst of critical activity
  - suspension might need interrupts to be turned off
    - e.g., the process being suspended is updating sensitive kernel data-structures
    - however, interrupts cannot always be ignored
  - poses challenging problems to coordinate the states of processes interrupted in a preemptive way

Preliminaries: Scheduling Metrics

User Oriented

Performance Related

- *response time:* time it takes to produce the first response
- *turnaround time*: time spent from the time of "submission" to time of completion
- *deadlines*: the time within which the program must complete (the policy must maximize percentage of deadlines met)

Other

- *predictability:* expectation that the job runs the same regardless of system load

System Oriented

Performance Related

- *waiting time*: time spent waiting to get the CPU
- *throughput*: the number of processes completed per unit time (directly affected by the waiting time)
- *CPU utilization*: percentage of time the CPU is busy

Other

- *fairness*: no process should suffer starvation
- *enforcing priorities*: higher priority processes should not wait

Scheduling Algorithms (1)
First-come First-served (FCFS)

- Non-preemptive

- Implementation
  - a queue of processes
  - new processes enter the ready queue at the end
  - when a process terminates
    - the CPU is given to the process at the beginning of the queue
  - (in practice) when a process blocks
    - it goes to the end of the queue
    - the CPU is given to the process at the beginning of the queue

- How does FCFS perform?

Performance of FCFS

- 3 processes P1, P2, and P3 with CPU requirements 24, 3, and 3 msecs
  - Arrive at the same time in that order

- Average waiting time = (0+24+27)/3 = 17
- Average turnaround time = (24+27+30)/3 = 27
- Average throughput = (30)/3 = 10
- Can we do better?

- Average waiting time = (0+3+6) / 3 = 3 !!!
- Average turnaround time = (3+6+30)/3 = 13 !!!
- Average throughput = (30)/3 = 10

Evaluation of FCFS

- *Pro*: Very simple code, data-structures and hence low overhead
- *Con*: Can lead to large average waiting times
- General disadvantage due to lack of preemption
  - when a poorly (long-term) scheduled collection has one large task with lots of CPU needs and a collection of others with I/O intensive needs
    - the CPU intensive process can cause very large delays for the processes needing (mostly) I/O

Scheduling Algorithms (2)
Shortest Job First (SJF)

- The next process to be assigned the CPU is one that is ready and with *smallest next CPU burst;* FCFS is used to break ties
  - From the previous example,
    - P1, P2, P3 arrive at the same time in that order, needing CPU times 24, 3, 3
      - FCFS yielded an average waiting time of 17 units
      - SJF yields order P2, P3, P1, with average waiting time of 3 units
  - Another example
    - P1, P2, P3, P4 requiring bursts of 8, 9, 4, and 5 arrive 1 time unit apart

FCFS: Average waiting time = (0 + (8 – 1) + (17 – 2) + (21 – 3))/4 = 10 units

SJF: Average waiting time = (0 + (17 – 1) + (8 – 2) + (12 – 3))/4 = 7.75 units

Evaluation of SJF

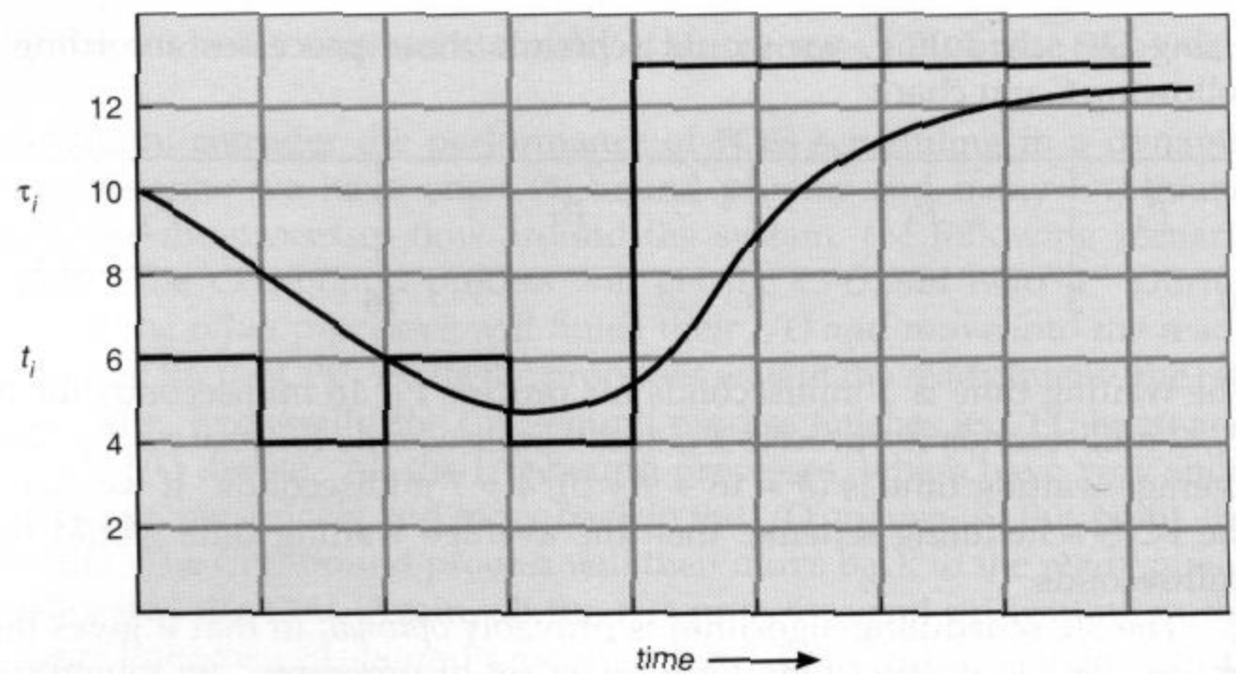
- *Pro:* If times are accurate, SJF gives *minimum* average waiting time

Estimating the burst times

- For long-term scheduling, user can be "encouraged" to give estimate
  - part of the job submission requirements

- For short-term scheduling, scheduler attempts to predict value
  - the approach assumes some locality in process CPU burst times
    - Use exponential averaging
      - $\tau_{n+1} = \alpha * T_n + (1 - \alpha) * \tau_n$
    - where,
      - τ_n is the estimated value for the n'th CPU burst
      - T_n is the actual most recent burst value
    - $\alpha = 0$ implies fixed estimate; $\alpha = 1$?; $\alpha = 0.5$?
  - the estimate lags the (potentially) sharper transitions of the CPU bursts

Estimating the CPU Burst (contd.)



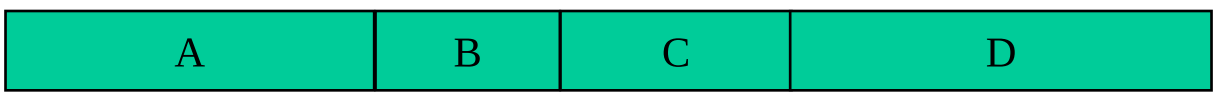
CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Figure 5.3 Prediction of the length of the next CPU burst.

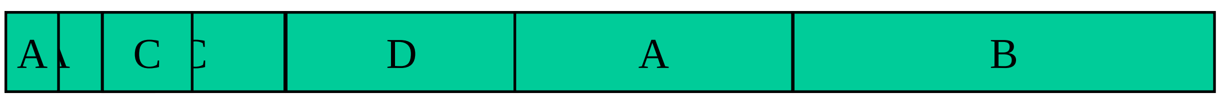
Modifications to SJF

- Preemptive SJF (also called shortest remaining time first)
  - if the shortest estimated CPU burst among all processes in the ready queue is less than the remaining time for the one running,
    - preempt running process; add it to ready queue w/ remaining time
    - give CPU to process with the shortest CPU burst
  - policy prioritizes jobs with short CPU bursts

- Example: A, B, C, D with bursts 8, 9, 4, 5 arrive 1 time unit apart



SJF: Average waiting time = (0 + (17 – 1) + (8 – 2) + (12 – 3))/4 = 7.75 units



Preemptive SJF: Average waiting time =
((0 – 0 + 9) + (17 – 1 + 0) + (2 – 2 + 0) + (6 – 3 + 0))/4 = 7 units

Scheduling Algorithms (3)
Priorities: A More General Scheduling Notion

- Elements of a priority-based scheduler
  - Process priorities (for example 0..100)
    - convention: a smaller number means higher priority
  - Tie-breaker mechanism
    - Example: FCFS
  - Map priority to considerations we have in mind
    - Internal
      - memory and other needs of the job
      - ratio of CPU to I/O burst times
      - number of open files etc.
    - External
      - the amount of money paid by the process owner
      - the importance of the user group running the process

- Priority-based scheduling
  - assign the CPU to the process with highest priority
  - may be used with or without preemption

Priority-based Scheduling: Example

- Consider five processes A, B, C, D, and E
  - With burst times: 10, 1, 2, 1, 5
  - With priorities: 3, 1, 3, 4, 2 (lower is better)
  - Arriving at times: 0, 0, 2, 2, 3

Without preemption:

Average waiting time: ((1 – 0) + (0 – 0) + (16 – 2) + (18 – 2) + (11 – 3))/5 = 7.8

With preemption:

Average waiting time: ((1 – 0 + 7) + (0 – 0) + (16 – 2) + (18 – 2) + (3 – 3))/5 = 7.6

Problems with Priority Schemes

- Process can be overtaken by higher priority processes arriving later
  - can happen continuously: leads to *starvation*
  - leads to better *overall* performance perhaps
    - but not from the point of view of the process in question
- Common solution: A process' priority goes up with its age
  - FCFS is used to break ties between processes with equal priorities
  - For a process in ready queue, its priority will eventually be the highest

- A low-priority process holds resources required by a high-priority process? (priority inversion)
- Common solution: Priority inheritance
  - process with lock inherits priorities of processes waiting for the lock
  - priority reverts to original values when lock is released

Example of Priority Ageing: Unix

- Priority goes up with lack of CPU usage
  - process accumulates CPU usage
  - every time unit (~ 1 second)
    - recalculates priority

      priority = CPUusage + basepriority
    - halves CPUusage carried forward

      CPUusage = (CPUusage) / 2
    - recall that smaller number implies a higher priority
  - basepriority is settable by user
    - within limits
    - using “nice”

- Assuming all processes have the same base priority:
  - Are new processes prioritized over existing ones?
  - How does the priority of a process change over its lifetime?

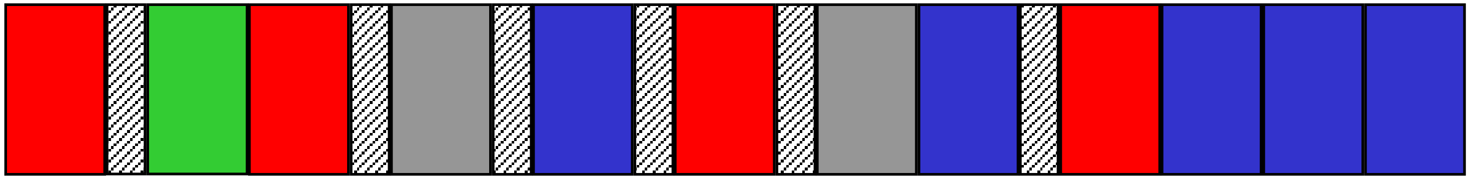
Scheduling Algorithms (4): Round Robin (RR)

- A strictly preemptive policy

- At a general level
  - choose a fixed time unit, called a quantum
  - allocate CPU time in quanta
  - preempt the process when it has used its quantum
    - Unless the process yields the CPU because of blocking
  - typically, FCFS is used as a sequencing policy
    - each new process is added at the end of the ready queue
    - when a process blocks or is preempted, it goes to the end of the ready queue
  - very common choice for scheduling interactive systems

Round-robin Scheduling: Example

- Consider five processes **A**, **B**, C, and **D**
  - With burst times: 4, 1, 2, 5
  - Arriving at times: 0, 0, 2, 3
- Round-robin system with quantum size 1 unit
  - Overhead of context switching a process: 0.2 units
    - Incurred only when a process is preempted or needs to block



Waiting time = ((0 – 0 + 6.2) + (1.2 – 0 + 0) + (3.4 – 2 + 2.6) + (4.6 – 3 + 3.6))/4 = **4.15** units
FCFS = (0 + (4-0) + (5-2) + (7-3))/4 = **3.75** units

Response time = ((0 + (1.2 – 0) + (3.4 – 2) + (4.6 – 3))/4 = **1.05** units
FCFS = (0 + (4-0) + (5-2) + (7-3))/4 = **3.75** units

CPU utilization?

Choice of Quantum Size

- Quantum size *q* is critical
- Affects waiting and turnaround times
  - if *q* is the quantum size and there are *n* processes in the ready queue,
    - the maximum wait is *(n-1)* . *q* units of time
  - as *q* increases, we approach FCFS scheduling
  - as q decreases
    - ⬇ the rate of context switches goes up, and the overhead for doing them
    - ⬆ the average wait time goes down, and the system approaches one with *1/n* the speed of the original system

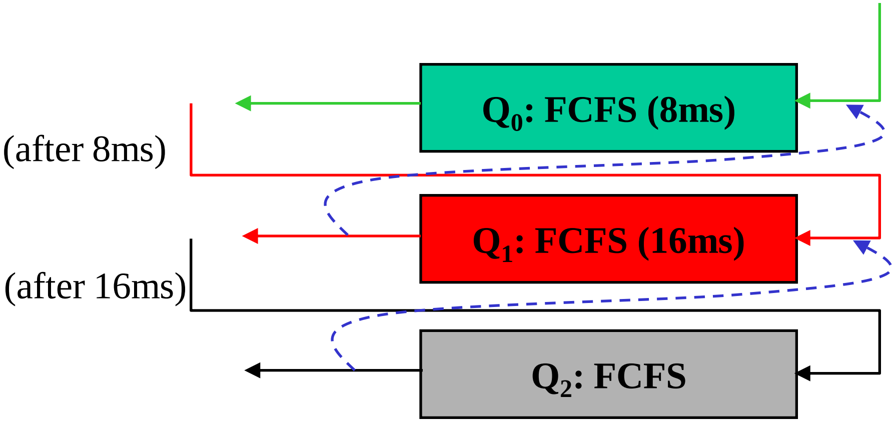
Hybrid Schemes: Multilevel Queue Scheduling

- Processes are partitioned into groups based on static criteria
  - background (batch)
  - foreground (interactive)
- All the processes in a fixed group of the partition share the same scheduling strategy and a distinct family of queues
  - different scheduling algorithm can be used across different groups
    - foreground: Round Robin
    - background: FCFS
- Need to schedule the CPU between the groups as well; for example,
  - fixed-priority: e.g., serve all from foreground, then from background
    - possibility of starvation
  - time slice: each group gets a certain fraction of the CPU
    - e.g., 80% to foreground in RR, 20% to background in FCFS

Generalization: Multilevel Feedback Queues

- Provide a mechanism for jobs to move between queues
  - ageing can be implemented this way
- Complete specification
  - queues: number, scheduling algorithms (within and across queues)
  - promotion and demotion policies
  - which queue should a process enter when it needs service?

- Example: 3 queues: Q_0 (FCFS, 8ms), Q_1 (FCFS, 16ms), Q_2 (FCFS)



Choosing a Scheduling Approach

- Identify metrics for evaluation
  - we have already seen a variety of metrics
    - throughput, wait time, turnaround time, ...
  - the goal is to start with an expectation or specification of what the scheduler should do well
    - for example, we might wish to have a system in which
      - the CPU utilization is maximized, subject to a bound on the response time
- Evaluate how different scheduling algorithms perform
  - deterministic modeling
    - requires accurate knowledge of job and system characteristics
    - practical only for real-time and embedded systems
  - more detailed performance evaluation
    - queueing models, simulation, measurement
- See Section 5.7 for details

Windows XP Scheduler (Section 5.6.2)

- Preemptive, priority based
- 32 priority levels: Higher priority numbers imply higher priority
  - priority level 0: memory management thread
  - 1-15 are variable priority classes
    - processes start off with a base priority (one of these levels)
      - HIGH (13), ABOVE_NORMAL (10), NORMAL (8), BELOW_NORMAL(6), IDLE (1)
    - threads in the process can start at priority = *(base_priority* ± 2*)*
      - Additional support for TIME_CRITICAL (15) and IDLE (1) threads
      - OS raises priorities of I/O-bound threads (max value is 15)
        - » Amount of boost depends on type of I/O
      - OS lowers priorities of CPU-bound threads (min value is *base_priority-2*)
      - distinction between foreground and background processes in NORMAL class
  - 16-31 are real-time priority classes
    - real-time threads have a fixed priority
    - threads within a particular level processed according to RR

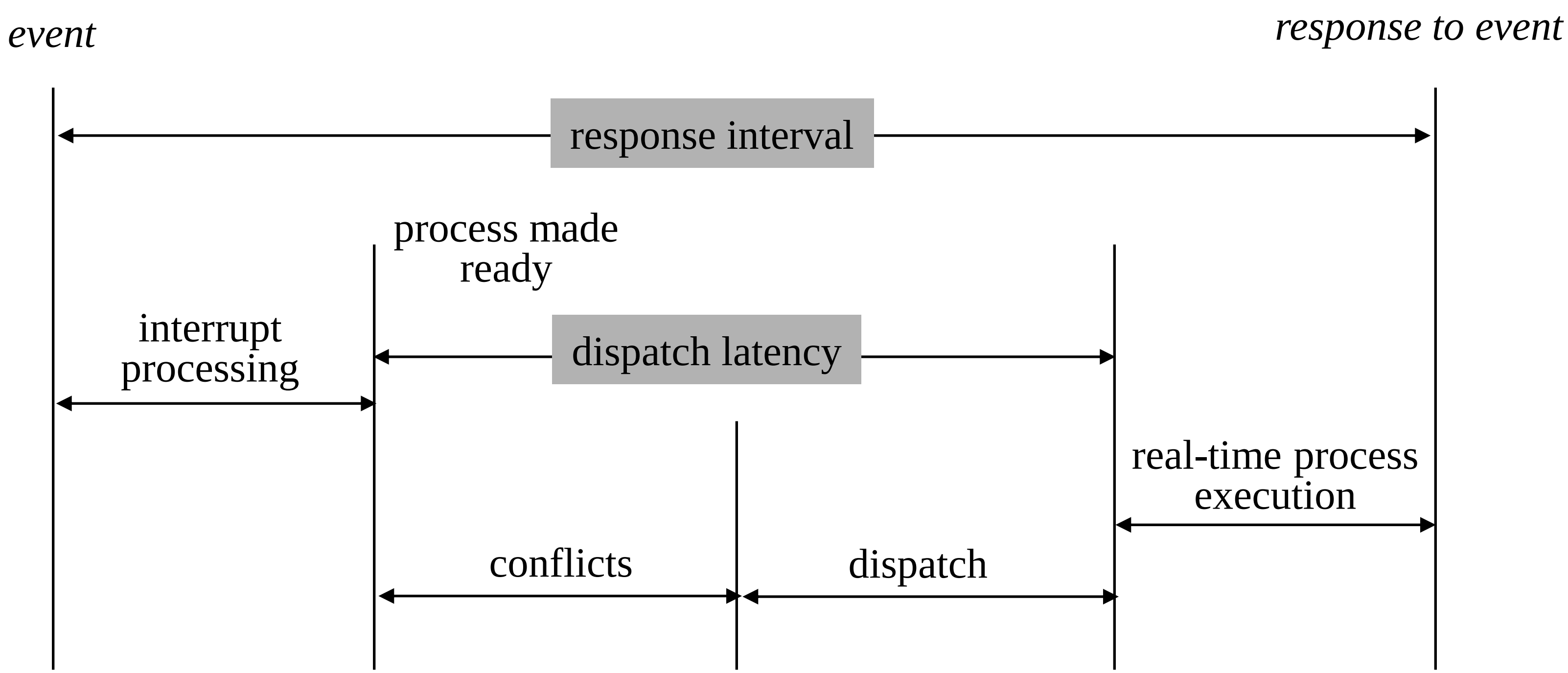
Advanced Topic: Real-Time Scheduling

- Processes have real-time requirements (deadlines)
  - e.g., a video-frame must be processed within certain time
  - growing in importance
    - media-processing on the desktop
    - large-scale use of computers in embedded settings
      - *sensors* produce data that must be processed and sent to *actuators*

- Real-time tasks typically considered along two dimensions
  - aperiodic (only one instance) versus periodic (once per period T)
  - hard real-time (strict deadlines) versus soft real-time
    - hard real-time tasks require *resource reservation*, and (typically) *specialized hardware* and scheduling *algorithms*
      - earliest-deadline first
      - rate-monotonic scheduling
      - details are beyond the scope of this class
    - our focus is on supporting soft real-time tasks in a general environment

Soft Real-Time Scheduling

- Most contemporary, general-purpose OSes deal with soft real-time tasks by being *as responsive as possible*
  - ensure that when a deadline approaches, the task is quickly scheduled
    - minimize latency from arrival of interrupt to start of process execution



Soft Real-Time Scheduling: OS Requirements

- Minimize interrupt processing costs
  - minimization of intervals during which interrupts are disabled

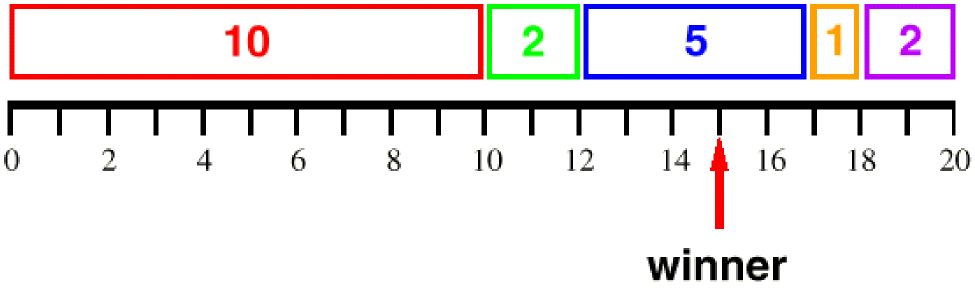
- Minimize dispatch latency
  - preemptive priority scheduling
    - real-time processes have higher priority than non real-time processes
    - priority of real-time processes does not degrade over time
  - current activity must be preemptible
    - Unacceptable options
      - traditional UNIX approach (waiting for system call completion)
      - preemption at *safe points*
    - Acceptable: entire kernel must be preemptible (e.g., Solaris 2)
      - kernel data structures protected by synchronization mechanisms
    - Must cope with the priority inversion problem
      - A lower-priority process holds a resource required by the higher-priority process

Fair-Share Scheduling

- Problems with priority-based systems
  - priorities are absolute: no guarantees when multiple jobs with same priority
  - no encapsulation and modularity
    - behavior of a system module is unpredictable: a function of absolute priorities assigned to tasks in other modules
- *Solution*: Fair-share scheduling
  - each job has a *share:* some measure of its relative importance
    - denotes user's share of system resources as a fraction of the total usage of those resources
    - e.g., if user A's share is twice that of user B
      - then, in the long term, A will receive twice as many resources as B
- Traditional implementations
  - keep track of per-process CPU utilization (a running average)
  - reprioritize processes to ensure that everyone is getting their share
  - are slow!

Example Fair-Share Policy: Lottery Scheduling

- A randomized mechanism for efficient *proportional-share* resource management
  - each process has certain number of lottery tickets (its share)
    - Processes reside in a conventional ready queue structure
  - each allocation is determined by holding a *lottery*
    - Pick a random ticket number
    - Grant resource to process holding the winning ticket



Why Does Lottery Scheduling Work?

- Expected allocation of resources to processes is proportional to the number of tickets that they hold

- Number of lotteries won by a process has a binomial distribution
  - probability p of winning = t/T
  - after n lotteries, E[w] = np and variance = $np(1-p)$

- Number of lotteries to first win has a geometric distribution
  - E[n] = $1/p$, and variance = $(1-p)/p^2$