# CS452/652 Real-Time Programming Course Notes

Daniel M. Berry, Cheriton School of Computer Science
University of Waterloo




## Train Tracking

Objective:

- to know where the trains are at all times
- to send the trains to specific destinations




## Train Tracking, Cont,d

Complications:

- A train's speed is not always what it should be, i.e., even with no speed change command, a train's speed changes.
- Different trains behave differently.
- Speed change is not instantaneous; a train accelerates and decelerates.




## Train Tracking, Cont,d

- Any sensor may not fire when a train is present, or it may fire in the absence of any train.
- Any switch may not respond to commands.
- An act of God may cause a train to removed from the track, a train to be put down on the track, a sensor to be tripped, or a switch to be thrown (the marker acting as God ☺ ).

# Notation for Track Position

See the course Website

- *sensor# + offset*
- *sw#[s/c] + offset*



# Tracking Assignment 1

- one train,
- marks based on accuracy of system's display of train location,
- command init *trainNumber* finds a given train on the track,
- run the train, throw switches, stop the train, check actual location compared to displayed location,
- error recovery: tracker must be able to handle spurious sensor hits, failed switching, acts of God, etc.



# Strong Hint

Design for the final result from the beginning:

- support offsets,
- choose algorithms that scale well to multiple trains,
- test all events, including low-probability events.



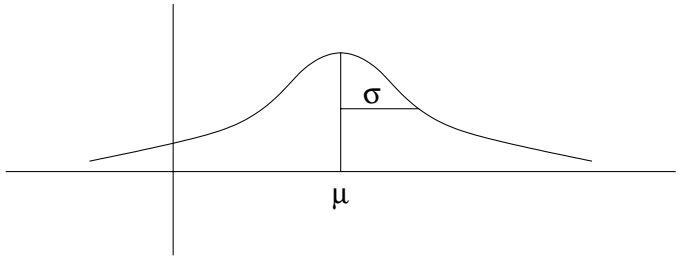
# Train Tracking

The distance $D_i$ that a train $t_i$ can travel in $k$ clock ticks is a random variable.

$$D_i \sim N(\mu_i, \sigma_i)$$

$\sim N$ is read "varies according to the normal distribution" where $\mu$ and $\sigma$ are as on the graph on the next slide

# Normal Distribution






# PDF

Probability distribution function (pdf):

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$



# PDF, Cont'd

Note that this function, given μ, the expected location of a train, σ, the standard deviation about that μ, and an *x*, the location of a sensor trip, computes the probability $p(x)$ that the sensor trip corresponds to the expected location μ.



# PDF, Cont'd

The existence of this function means that you can program this function as a procedure that is invokable at any time.

So all your program has to keep track of is μs and σs.

The rest of this section describes how to determine a new $\mu_i$ and and a new $\sigma_i$ for train *i* from older ones under a variety of circumstances.

# Addition of Random Variables

Random variables add as follows:

$$D_i' \sim N(\mu_i', \sigma_i')$$

$$D_i + D_i' \sim N(\mu_i + \mu_i', \sqrt{\sigma_i^2 + \sigma_i'^2})$$

That is, the means add and the variances, not the standard deviations, add.

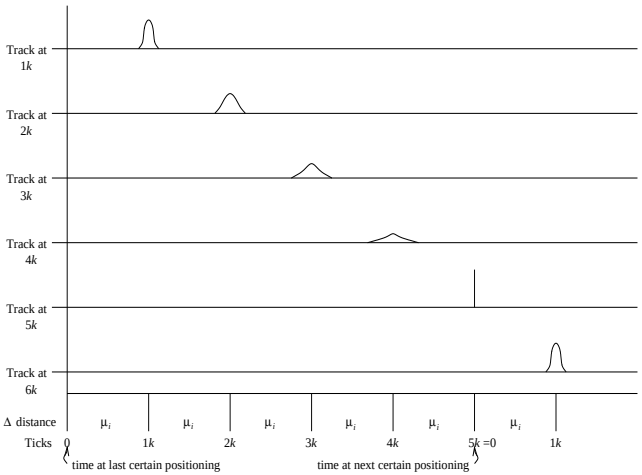
$\therefore$, the distance traveled in $2k$ clock ticks is
$\sim N(2\mu_i, \sqrt{2}\,\sigma_i)$



# Train's Position

On the basis of these distributions, a train's position is a probability distribution, shown on the next slide:








# Train's Position, Cont'd

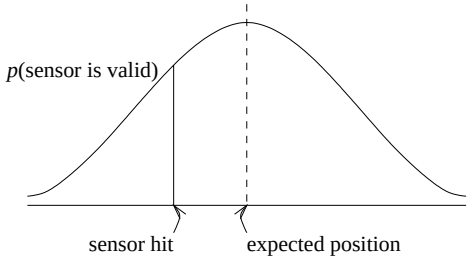
Each track line represents a different time, from top to bottom, at $1k$, $2k$, $3k$, $4k$, $5k$, and $6k$ ticks.

Note that the distribution widens and flattens over time.

The idea is to try to locate the train with enough certainty that the distribution is collapsed to a spike and you start all over with ever widening and flattening distributions.

# Sensor Hit

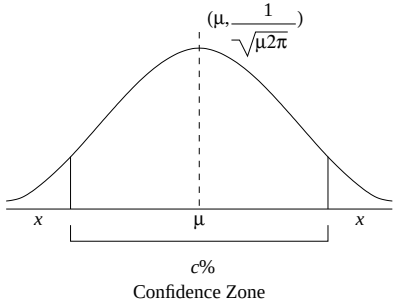
On a sensor hit, the distance from the expected position gives the probability that the sensor was triggered by the train.







# Confidence Zone




with $c$ = 95 or some other high number




# Confidence Zone, Cont'd

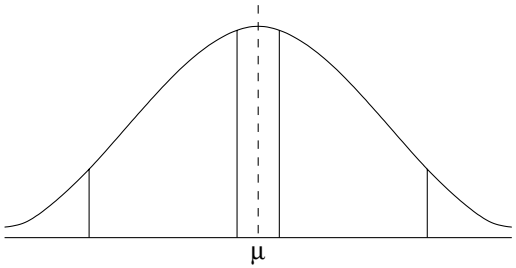
The area symmetric about μ that covers $c$% of the area under the curve is the confidence zone.

We say "With $c$% probability, the train is in this zone."

If the sensor hit is outside, then with $c$% probability, the hit is spurious.




# Highest Probability

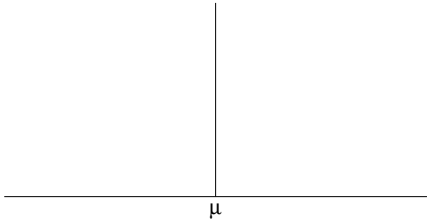



The zone right around μ has the highest probability of containing the train.

# Highest Probability, Cont'd

Find a suitable size for this zone, e.g., 2 cm?

If the sensor hits in this highest probability zone, then believe that the train is there with certainty and collapse the curve, and reset time to 0.






# In Between

In the rest of the confidence zone, assign intermediate confidence to the sensor hit, i.e., calculate a new distribution for the next *k* ticks point that is flatter and wider than the current one.



# New Distribution

If the sensor hit is valid with probability $p$ and invalid with probability $1-p$, then the new distribution is D:

Let $D_i$ be the current distribution of train $i$'s position, and let $s_i$ be the sensor position of train $i$, where $s_i \sim N(\mu, 0)$, where $\mu$ is the definite location of the sensor and $\mu_i$ is the expected location of train $i$..

$$D = (1-p)\,D_i + p s_i$$

$$D \sim N((1-p)\,\mu_i + p\mu,\ \sigma_i\sqrt{1-p}\ )$$



# Going Over a Switch

As a train is about to go over a switch, the expected path is determined by the program's model of the direction in which the switch is thrown.

However, there is a small probability, $p$, that your program's model of the direction of any switch is wrong

We estimate this probability as about 1%.

# Going Over a Switch, Cont'd

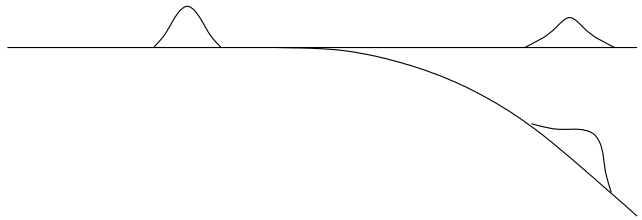
Although acts of God have been known to change this probability. ☺

By the way, these acts of God, in which you play God, are a good way to test how well your program reacts to low probability events, which may never happen in a normal test.



# Going Over a Switch, Cont'd

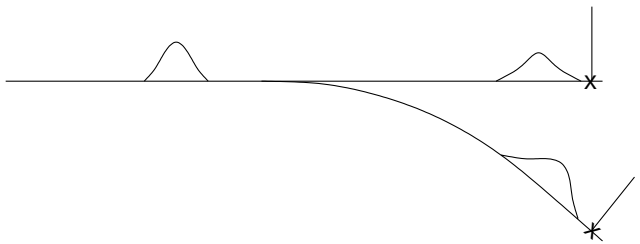
Split the distribution into: $(1-p)D_{\text{correct}}$ and $pD_{\text{incorrect}}$





# Going Over a Switch, Cont'd

If an expected sensor fires in the high confidence zone on one or both paths, collapse the curve as before:





# Going Over a Switch, Cont'd

The inclusion of the word "both" may be counterintuitive to you.

You are right!

If both sensors just after a switch fire, then the probability that the train is on each branch is 50% of the probability that it was at the switch.

# Going Over a Switch, Cont'd

However, under each choice, the location is accurate and is a good basis for estimating at the next *k* tick point where the train should be in the track that flows from the choice.

But because of the uncertainty about *which* choice was taken, you have to keep both distributions active until you become sufficiently certain, as a result of additional distribution collapsing sensor hits, about which choice was taken.



# Going Over a Switch, Cont'd

Further down the line, we expect that the next sensor of the branch not taken by the train to not report any hit. So we consider how to use that non-hit to begin to discount that branch's distribution.



# Going Over a Switch, Cont'd

If an expected sensor does not report any hit, scale its distribution down by some factor *f* and renormalize.

On the assumption that a switch is not likely to fail temporarily more than once in an interval that matters, this *f* is a number in the range of 2 through 6 or so, that has to be determined experimentally.



# Going Over a Switch, Cont'd

Suppose $D_A$ and $D_B$ are the distributions of two maintained paths *A* and *B*.

Start with $p_A D_A + p_B D_B$, where $p_A + p_B = 1$

*B* failing to report an expected sensor hit maps to

$$\frac{p_A D_A + \frac{p_B}{f} D_B}{p_A + \frac{p_B}{f}}$$

## Going Over a Switch, Cont'd

This diminishes $D_B$ and amplifies $D_A$.

Once $D_B$'s amplitude falls below a certain threshold, remove it and renormalize the remaining distributions.

This can be generalized to any number of maintained paths, e.g, for a switch right after another, yielding three maintained paths, *A*, *B*, and *C*.



## Going Over a Switch, Cont'd

Start now with $p_A D_A + p_B D_B + p_C D_C$, where $p_A + p_B + p_C = 1$

*C* failing to report an expected sensor hit maps to

$$\frac{p_A D_A + p_B D_B + \frac{p_C}{f} D_C}{p_A + p_B + \frac{p_C}{f}}$$



## Going Over a Switch, Cont'd

This diminishes $D_C$ and amplifies $D_A$ and $D_B$.

Once $D_C$'s amplitude falls below a certain threshold, remove it and renormalize the remaining distributions.



## Going Over a Switch, Cont'd

Recall that we have been talking what happens if an expected sensor fires in the high confidence zone in *both* paths following a switch.

The most common circumstance is that only *one* expected sensor fires in the two high confidence zones.

In that case, believe it!

# Tracking Assignment 2

- multiple trains
- routing

Routing is sending a train to a specified location.

The track is a graph, in which each switch is a node.

Therefore, routing ≡ graph search



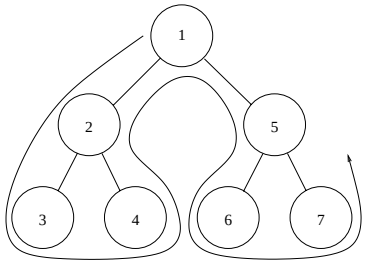
# Graph Search Algorithms

- Depth-first search
- Breadth-first search
- Iterate deepening depth-first search
- Dijkstra's algorithm
- Floyd-Warshall algorithm



# Depth-First Search

Go down the graph, expanding children until none remain, then backtrack and check siblings






# Depth-First Search, Cont'd

```
DFS (start,goal){
    if (start == goal) {
        stop;
    } else {
        foreach child, v, of start {
            DFS(v,goal);
        }
    }
}
```

# Depth-First Search, Cont'd

If $G = (V,E)$, run time is $O(|V| + |E|)$.

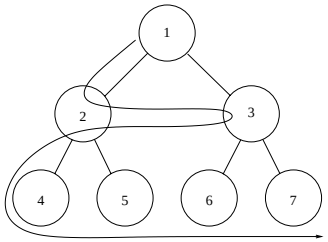
DFS(s,g) may not terminate if the graph has a cycle.

- need to keep track of visited nodes,
- first solution may not be the shortest possible



# Breadth-First Search

Go across the graph, examining all children, before descending to grandchildrem.




Needs data structure, a queue of nodes to visit.



# Breadth-First Search, Cont'd

```
BFS(Q,goal) {
    q = dequeue(Q);
    if (q == goal) {
        stop;
    } else {
        foreach child, v, of q {
            Q = enqueue(Q,v);
        }
    BFS(Q,goal);
    }
}
```



# Breadth-First Search, Cont'd

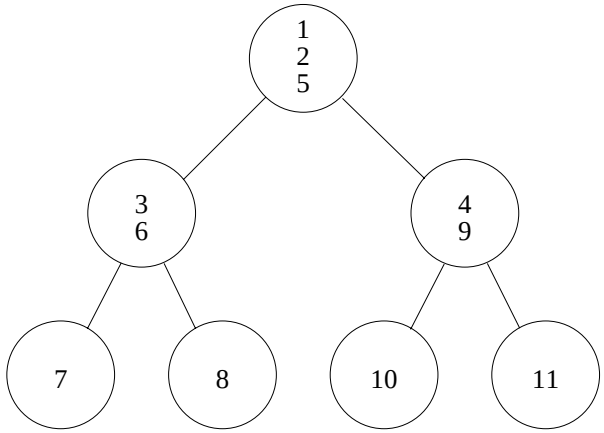
- if $G = (V,E)$, run time is $O(|V| + |E|)$
- guaranteed to terminate if there is a solution
- first solution reported is the shortest

# Iterative Deepening DFS, Cont'd

```
for n = 1 .. ∞ {
    perform DFS to maximum search depth n
    if solution found, stop
}
```



# Iterative Deepening DFS






# Iterative Deepening DFS, Cont'd

- run time is $O(b \times d)$, where
  $d$ = depth at which solution occurs
  $b$ = maximum number of branches at any node
- guaranteed to terminate if there is a solution
- first solution reported is the shortest



# Dijkstra's Algorithm

Finds shortest paths from a given source to all targets in an edge-weighted graph

$\therefore$ accounts for actual track lengths

Array d[v] ≡ distance from source to vertext v;
Initialize d[v] = ∞ for all v;
d[source] = 0;
Q = all unvisited nodes (priority queue);

# Dijkstra's Algorithm, Cont'd

```
while (Q is not empty) {
    u = getMin(Q);
    foreach neighbor v of u {
        d[u] = min(d[v], d[u]+length(u,v));
        if the min is d[u]+length(u,v) {
            include edge (u,v) in the
            min spanning tree;
        }
    }
}
```



If $G = (V,E)$, with efficient data structures, can be implemented in $O(|E| + |V| + \log|V|)$



# Floyd-Warshall Algorithm

Similar to Dijkstra's algorithm, but computes shortest paths between *all* pairs of vertices.

A dynamic programming algorithm:

Build up a 2-dimensional array path

path[i,j] = shortest path between nodes i and j, based on nodes considered so far

Initially path[i,j] = edgecost(i,j) $\forall$ i,j and is undefined if i and j do not share an edge



# Floyd-Warshall Algorithm, Cont'd

```
for k = 1 .. n {
    for each (i,j) {
        path[i,j] = shorter of path[i,j] and
            concat(path[i,k],path(k,j))
    }
}
```

Could be used to precompute all routes, rather than running Dijkstra on the fly, leading to a potential time savings.

# Floyd-Warshall Algorithm, Cont'd

If $G = (V,E)$, run time is $O(|V|^3)$, i.e., $\approx O(|V|)$ for each pair of vertices

But, precomputation fails if part of the track becomes unusable, e.g., a switch is not working or a train is in the way.

$\therefore$, you probably need on-the-fly calculation, at least as a back up.



# Real-Time Scheduling Theory

We have discussed real-time constraints in the context of Assignment 1, i.e., for the the cyclic executive, which does not really use tasks.

Now we use real tasks and add priority.



# RT Scheduling Theory, Sources

Paper:

C.L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM* **20**:1, pp. 46–61, 1973.

Book:

Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages*, Third Edition, Addison-Wesley, Harlow, England, 2001



# Problem

Given a set of *periodic* tasks, $t_1, \ldots, t_m$,

with periods, $T_1, \ldots, T_m$,

execution times, $c_1, \ldots, c_m$,

and priorities, $p_1, \ldots, p_m$,

can every task meet its deadline?

## Assumptions

- Any ready, high priority task always preempts a low priority task.
- Tasks do not block on each other.
- We are free to choose the priority of each task.



## Critical Instant

A *critical instant* for a task *t* occurs when every higher priority task is at the beginning of its period.

This critical instant gives the worst case response time from *t*'s viewpoint.

A *global critical instant* is the critical instant of the lowest priority task.

That is, every task is at the beginning of its period.



## Theorem

If a set of task is schedulable at its global critical instant, then it is schedulable.



## Rate-Monotonic Scheduling (RMS) Algorithm

Here "rate" means "task period"

Assign priorities such that if $T_i < T_j$, then $P_i > P_j$.

That is, tasks with shorter periods get higher priorities.

Every task $t_i$ has a *deadline* $D_i$, the time by which it *must* finish.

For now, we take $D_i = T_i$; later we take $D_i < T_i$

# Fundamentals of RMS

For RMS, every task will meet its deadline if:

$$\sum_{i=1}^{m} \frac{c_i}{T_i} \leq m(2^{\frac{1}{m}} - 1)$$

Note that $\frac{c_i}{T_i}$ is the fraction of the CPU time used by task $t_i$.



# Fundamentals of RMS, Cont'd

As $m \to \infty$, $m(2^{\frac{1}{m}} - 1) \to 69\%$ from above.

Therefore, if $\sum_{i=1}^{m} \frac{c_i}{T_i} \leq .69$, then you are safe without calculating the specific $m(2^{\frac{1}{m}} - 1)$



# Fundamentals of RMS, Cont'd

But, if $\sum_{i=1}^{m} \frac{c_i}{T_i} > .69$, …

you may still make it if $m$ is small enough,

e.g. if $m = 2$, $m(2^{\frac{1}{m}} - 1) = 2 \times (2^{1/2} - 1) \approx .828$



# Fundamentals of RMS, Cont'd

Even if the very conservative formula cannot promise that you will find a schedule that permits every task to meet its deadline, you may still make it …

if the particular configuration of $c_i$s and $T_i$s you have is such that a schedule can be found by brute force.

# Fundamentals of RMS, Cont'd

Consider the case with one task, i.e., $m = 1$.

Then $m(2^{\frac{1}{m}} - 1) = 1 \times (2^1 - 1) = 1$

So one task is schedulable if that one task does not use more than 100% of the CPU, i.e., if $c_1 \leq T_1$!



# Example 1

| Priority | Task | $T_i$ | $c_i$ | $\frac{c_i}{T_i}$ |
|---|---|---|---|---|
| high 1 | $t_0$ | 30 | 10 | 0.33 |
| 2 | $t_1$ | 40 | 10 | 0.25 |
| low 3 | $t_2$ | 50 | 12 | 0.24 |
| Total CPU | Usage | | | 0.82 |

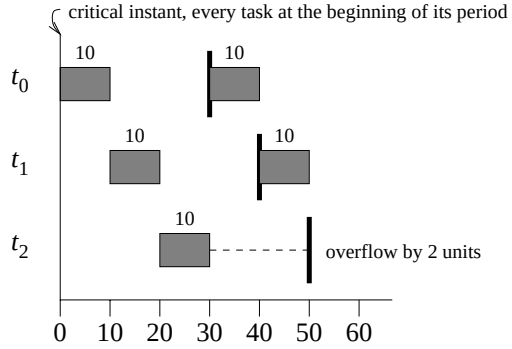
The priorities are derived from the $T_i$s

$3(2^{1/3} - 1) \approx 0.78$

Since 0.82 > 0.78, the test fails.



# Brute Force?




Nope!



# Brute Force, Cont'd

This set of tasks may appear schedulable if you don't start at a critical instant …

but that doesn't tell you much because you did not start at the worst possible instant.

# Example 1'

Now change $c_2$ from 12 to 10.

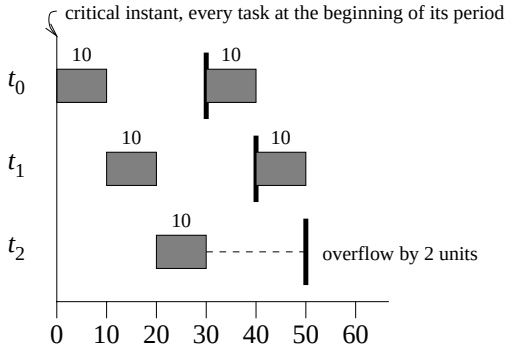
| Priority | Task | $T_i$ | $c_i$ | $\frac{c_i}{T_i}$ |
|---|---|---|---|---|
| high 1 | $t_0$ | 30 | 10 | 0.33 |
| 2 | $t_1$ | 40 | 10 | 0.25 |
| low 3 | $t_2$ | 50 | 10 | 0.20 |
| Total CPU | Usage | | | 0.78 |

$3(2^{1/3}-1) \approx 0.78$

Since $0.78 \leq 0.78$, the test succeeds.



# Brute Force!




Yep!



# Example 2

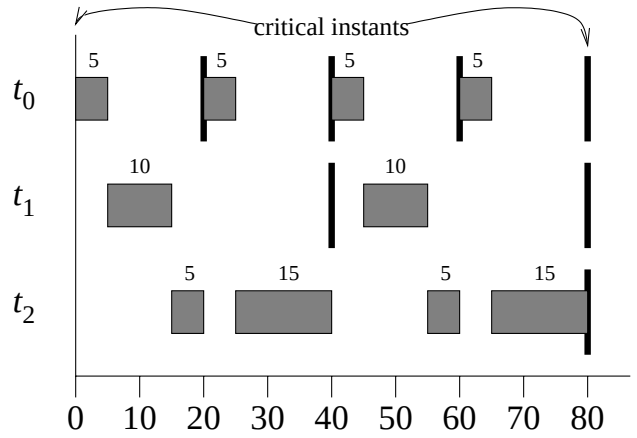
| Priority | Task | $T_i$ | $c_i$ | $\frac{c_i}{T_i}$ |
|---|---|---|---|---|
| high 1 | $t_0$ | 20 | 5 | 0.25 |
| 2 | $t_1$ | 40 | 10 | 0.25 |
| low 3 | $t_2$ | 80 | 40 | 0.50 |
| | | | | 1.00 |

$3(2^{1/3}-1) \approx 0.78$

Since 1.00 > 0.78, the test fails.



# Brute Force?



# Brute Force, Cont'd

Since 5+15+5+15 = 40, this set of tasks is schedulable from critical instant to critical instant.

Therefore, it is schedulable.



# Sporadic Tasks

A task is sporadic if it is not strictly periodic.

Then $T_i$ represents a minimum period for task $t_i$.

Deadline $D_i < T_i$; the completion deadline is before the earliest possible next execution.

For sporadic tasks, we use deadline-monotonic priority assignment (DMPO) ("O" for "ordering").



# DMPO

In DMPO, we assign priorities so that if $D_i < D_j$, then $P_i > P_j$,

i.e., to give highest priorities to tasks with the shortest deadlines.



# Theorem

If a feasible priority assignment exists, then also DPMO is feasible.

Equivalently:

If DMPO is infeasible, then so is every other priority assignment.

# RMS is Special Case of DMPO

If $D_i = T_i$ for all $i$, then DMPO reduces to rate-monotonic scheduling.

Therefore, if $D_i = T_i$ for all $i$ and a feasible priority assignment exists, then also rate-monotonic scheduling is feasible.



# Computation of DMPO Worst Case

To compute the worst-case response time for DMPO:

Define *response time* $R_i$ for task $t_i$ as the time it takes for $t_i$ to complete after $t_i$ has become ready.

A schedule is feasible if and only if $R_i < D_i$ for all $i$.

Define *interference time* $I_i$ for task $t_i$ as the maximum time spent executing higher priority tasks when $t_i$ is ready.

Then $R_i = c_i + I_i$.



# Worst Case, Cont'd

If $t_H$ is the highest priority task, then:

$$R_H = c_H$$

If $t_{H_2}$ is the second highest priority task, then:

$$R_{H_2} = c_{H_2} + x\ c_H$$

where $x$ = the number of times $t_H$ is scheduled before $t_{H_2}$ completes.



# Worst Case, Cont'd

That is,

$$R_{H_2} = c_{H_2} + \left\lceil \frac{R_{H_2}}{T_H} \right\rceil c_H$$

In general, for all $t_j$ with $P_j > P_i$, the maximum interference from $t_j$ is:

$$\left\lceil \frac{R_i}{T_j} \right\rceil c_j$$

# Worst Case, Cont'd

Let $X_i$ be the set of tasks with priority > $P_i$.

Then:

$$R_i = c_i + \sum_{t_j \in X_i} \left\lceil \frac{R_i}{T_j} \right\rceil c_j$$

Solve for $R_i$.



# Worst Case, Cont'd

How do we compute $\left\lceil \frac{R_i}{T_j} \right\rceil$ without knowing $R_i$?

Solution: use a fixed-point computation.

Let $r_i^k$ denote the *k*th guess as to the value of $R_i$.

Initial guess: $r_i^0 = c_i$. Then let

$$r_i^{n+1} = c_i + \sum_{t_j \in X_i} \left\lceil \frac{r_i^n}{T_j} \right\rceil c_j$$



# Worst Case, Cont'd

Fact: $r_i^0 \le r_i^1 \le r_i^2 \cdots$.

If, for any *n*, we obtain $r_i^{n+1} = r_i^n$, then take $R_i = r_i^n$, and we have a solution.

If, for any *n*, we obtain $r_i^n > D_i$, then also every subsequent $r_i^k > D_i$, and we have no solution.



# Example 3

| Priority | Task | $T_i$ | $c_i$ | $D_i$ |
|---|---|---|---|---|
| High | $t_1$ | 7 | 3 | 7 |
| Medium | $t_2$ | 12 | 3 | 12 |
| Low | $t_3$ | 20 | 5 | 20 |

We let $D_i = T_i$ for simplicity.

# Example 3, Cont'd

$R_1 = c_1 = 3$

$r_2^0 = c_2 = 3$

$$r_2^1 = c_2 + \sum_{j<2} \left\lceil \frac{r_2^0}{T_j} \right\rceil c_j = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 3 + 1 \times 3 = 6$$



# Example 3, Cont'd

$$r_2^2 = 3 + \sum_{j<2} \left\lceil \frac{r_2^1}{T_j} \right\rceil c_j = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 3 + 1 \times 3 = 6$$

$r_2^2 = r_2^1$. $\therefore R_2 = 6$.



# Example 3, Cont'd

$r_3^0 = c_3 = 5$

$$r_3^1 = c_3 + \sum_{j<3} \left\lceil \frac{r_3^0}{T_j} \right\rceil c_j = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$r_3^2 = c_3 + \sum_{j<3} \left\lceil \frac{r_3^1}{T_j} \right\rceil c_j = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$



$$r_3^3 = c_3 + \sum_{j<3} \left\lceil \frac{r_3^2}{T_j} \right\rceil c_j = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$r_3^4 = c_3 + \sum_{j<3} \left\lceil \frac{r_3^3}{T_j} \right\rceil c_j = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$r_3^5 = c_3 + \sum_{j<3} \left\lceil \frac{r_3^4}{T_j} \right\rceil c_j = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

$r_3^5 = r_3^4$. $\therefore R_3 = 20$.

# Example 3, Cont'd

$R_1 = 3$, $R_2 = 6$, and $R_3 = 20$

$R_i < D_i \ \forall i, \therefore$ this set of tasks is schedulable.



# Example 1 Revisited

with task subscripts upped by 1:

| Priority | Task | $T_i$ | $c_i$ | $\frac{c_i}{T_i}$ |
|---|---|---|---|---|
| high 1 | $t_1$ | 30 | 10 | 0.33 |
| med 2 | $t_2$ | 40 | 10 | 0.25 |
| low 3 | $t_3$ | 50 | 12 | 0.24 |
| | | | | 0.82 |

Recall that $0.82 > 3(2^{1/3} - 1)(approx 0.78)$, and the test failed.



# Example 1, Cont'd

Response Time Calculation:

Start with:

$R_1 = c_1 = 10$



# Example 1, Cont'd

$r_2^0 = c_2 = 10$

$r_2^1 = 10 + \left\lceil \frac{10}{30} \right\rceil 10 = 20$

$r_2^2 = 10 + \left\lceil \frac{20}{30} \right\rceil 10 = 20$

$\therefore R_2 = 20$

# Example 1, Cont'd

$r_3^0 = c_3 = 12$

$r_3^1 = 12 + \left\lceil \frac{12}{30} \right\rceil 10 + \left\lceil \frac{12}{40} \right\rceil 10 = 32$

$r_3^2 = 12 + \left\lceil \frac{32}{30} \right\rceil 10 + \left\lceil \frac{32}{40} \right\rceil 10 = 42$



$r_3^3 = 12 + \left\lceil \frac{42}{30} \right\rceil 10 + \left\lceil \frac{42}{40} \right\rceil 10 = 52$

$r_3^4 = 12 + \left\lceil \frac{52}{30} \right\rceil 10 + \left\lceil \frac{52}{40} \right\rceil 10 = 52$

$\therefore$, $R_3 = 52 > T_3 = 50$, a missed deadline.



# Example 1, Cont'd

BUT: $R_3$ *is* a solution. Therefore, task $t_3$ will complete at time 52.

Rate-monotonic assignment is not schedulable, i.e., there is no schedulable priority assignment for these tasks.

However, the CPU usage is only 82%. So there are cycles left over.



# Dynamic Scheduling

In dynamic scheduling, priorities are allowed to change over time.

Earliest deadline first (EDF) assigns priorities dynamically so that the task with the earliest deadline has the highest priority.

# Example 1 Re-Revisited

| Priority | Task | $T_i$ | $c_i$ | $\frac{c_i}{T_i}$ |
|---|---|---|---|---|
| high 1 | $t_1$ | 30 | 10 | 0.33 |
| med 2 | $t_2$ | 40 | 10 | 0.25 |
| low 3 | $t_3$ | 50 | 12 | 0.24 |
| | | | | 0.82 |

Recall that $0.82 > 3(2^{1/3} - 1)(approx 0.78)$, and the test failed

and with DMPO, while solvable, $t_3$ missed its deadline



# EDF

EDF is feasible if

$$\sum_{i=1}^{m} \frac{c_i}{T_i} \leq 1$$

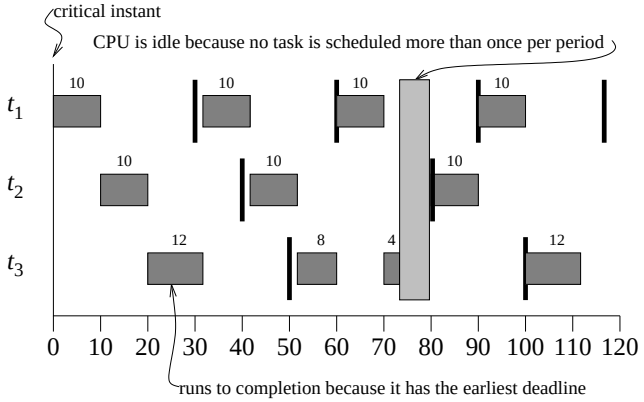
assuming $D_i = T_i$,

as it is in this case.

The original priorities are meaningless in EDF.



# Example 1, Cont'd



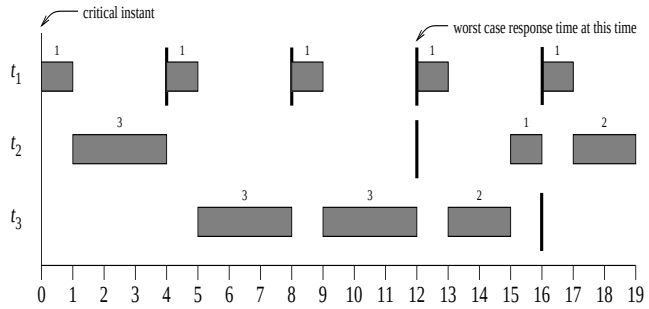



# Example 4

Worst-case response time for EDF does not necessarily occur at the critical instant, as in this example:

| Task | $T_i = D_i$ | $c_i$ | $\frac{c_i}{T_i}$ |
|---|---|---|---|
| $t_1$ | 4 | 1 | .25 |
| $t_2$ | 12 | 3 | .25 |
| $t_3$ | 16 | 8 | .5 |
| | | | 1.00 |

# Example 4, Cont'd






# Example 4, Cont'd

At the critical instant at time 0, the response time of $t_2$, $R_2 = 4$.

But at time 12, $t_3$ is still active, with a shorter deadline, 16, compared to 24 for $t_2$.

But at time 12, $t_1$ preempts $t_2$, and $t_2$'s response time $R_2 = 7$.

The response time for $t_i$ depends on the tasks with deadlines earlier than that of $t_i$.



# Tradeoffs

| Method | When Time's Spent | Space Needed at Run Time | Effectiveness in Meeting Deadlines |
|---|---|---|---|
| RMS | design time | priority table | √ - |
| BMPO | design time | priority table | √ |
| EDF | run time | a whole lot of data | √ ++ |



# Tradeoffs, Cont'd

Which one to use?

Look at the data.

If total CPU utilization is small, e.g., < .5, don't sweat it, use RMS.

If not, then may have to play with system to see.

But, try RMS first, DMPO after, and then EDF only if necessary.

# Application Project Requirements

Your application

- must track *at least* two trains,
- must include routing and collision detection,
- must display track layout on screen, showing the location of all trains,
- must include a display of time that does not slow down or speed up, that is, it does not gain or lose ticks.
- must be deadline-oriented, the more real-time content the better.



# Application Requirements, Cont'd

Your application may have extra features, such as mouse, sound, etc.

However, having these are not as important as meeting the real-time requirements.

In fact, their presence cannot make up for a failure to meet any real-time requirement.



# Proposals

In class on 6 November.

- Make a *short*, 5–10 minute, presentation to the class about your proposed application.
- Hand in a two-page proposal on paper.
- Your presentation must include a structure diagram and may include other visuals.
- You may use the blackboard, the overhead projector, or the data projector, or more than one.
- The instructor and class mates may ask questions during the presentation.
- Each group member must speak.
- Start thinking now.



# Tracking Assignment 2

Your program

- must track two trains, but collision detection not required yet,
- must recover if a train makes a wrong move, and

# Tracking Assignment 2, Cont'd

- must be able to send one train to an arbitrary location on the track, assuming that
  - there is no other train on the track,
  - the location is defined by an arbitrary offset from any switch or sensor,
  - the route chosen must be the shortest or very near to the shortest, and
  - the command for this routing is tr *trainNumber speed destination*.



# Scheduling for Task Interactions

- Most tasks do not run independently of each other. A typical task blocks on another task.
- Under deadline- or period-oriented priority assignments, a high priority task may Send to a low priority task, leading to priority inversion.
- But if priorities are assigned to avoid priority inversion, then tasks may end up not being schedulable.

∴ Response time analysis must account for blocking time.



# Task Interaction, Cont'd

Under priority inheritance, we can find an upper bound for the time spent blocking for task $t_i$

Let $B_i$ denote the maximum blocking time that task $t_i$ can suffer.

Let $K$ denote the number of critical sections in the system, where a critical section is a Receive-to-Receive cycle.



# Task Interaction, Cont'd

Then, $B_i = \sum_{j=1}^{K} usage(j,i)\, c(j)$, where

$$usage(j,i) = \begin{cases} 1 & \text{if section } j \text{ is sent to by at least one task with priority} < p_i \text{ and at least one task with priority} > p_i \\ 0 & \text{otherwise} \end{cases}$$

and $c(j)$ is the worst-case execution time for the $j$th critical section

# Task Interaction, Cont'd

The intuition behind the definition of *usage* is:

Priority inversion happens when a lower priority task $t_l$ is running when $t_i$, with $P_i > P_l$ is ready and wants to run.

So, *if* $t_l$ is running as a result of priority inheritance, *then*, $t_l$ is running at a priority $p_h$, where $P_h \geq P_i$, because $t_l$ shares a resource *rs* with $t_h$.



# Task Interaction, Cont'd

$\therefore$, for an inversion to occur, we need a resource to be shared and two things to happen:

1. The resource must be used by some task with a priority greater than or equal to $P_i$,

*and*

2. the resource must be used also by some task with priority less than $P_i$.



# Task Interaction, Cont'd

If we do not have 1, the resource is shared by two tasks with priorities less than $P_i$, and we cannot have inversion on $t_i$.

If we do not have 2, the resource is shared by only tasks with priorities greater than or equal to $P_i$, and the resulting delay is included already in the interference term, $I_i$, calculated by the $\sum$.



# Task Interaction, Cont'd

Then, the response time for task $t_i$ is:

$$R_i = c_i + B_i + \sum_{t_j \in X_i} \left\lceil \frac{R_i}{T_j} \right\rceil c_j$$

Solve by fixed-point computation as before:

$$r_i^0 = c_i + B_i$$

$$r_i^{n+1} = c_i + B_i + \sum_{t_j \in X_i} \left\lceil \frac{r_i^n}{T_j} \right\rceil c_j$$

# Task Interaction, Cont'd

Note that the worst case $B_i$ may not actually be achieved, e.g., if all tasks have the same period.