

Least Slack Time Scheduling with Aging

Introduction

Real-time systems are integral to various applications, from embedded systems in automotive and aerospace industries to complex data centers and cloud computing environments. These systems must efficiently manage both critical and non-critical workloads to ensure timely execution and optimal performance. Traditional scheduling algorithms, such as Earliest Deadline First (EDF) and Least Slack Time (LST), have been widely used to address these challenges. However, they often fall short in terms of fairness and handling mixed workloads, particularly in dynamic environments where task priorities and system conditions can change rapidly.

This project introduces a novel approach called **Least Slack Time Scheduling with Aging (LSTA)**. LSTA is a hybrid algorithm that combines the principles of slack time management with mechanisms to mitigate starvation. By dynamically adjusting task priorities based on both slack time and starvation metrics, LSTA aims to provide a more adaptable and fair scheduling solution. This approach not only improves the responsiveness of the system but also ensures that all tasks, regardless of their criticality, are given equitable consideration.

Abstract

This project presents **Least Slack Time Scheduling with Aging (LSTA)**, a hybrid scheduling algorithm designed for real-time systems with mixed workloads. LSTA integrates slack time calculations with starvation tracking to dynamically adjust task priorities, addressing key challenges such as fairness, responsiveness, and system utilization. Unlike traditional algorithms like EDF and LST, LSTA adapts priorities in real time, ensuring equitable task scheduling even in dynamic and multiprocessor environments.

The algorithm operates by computing the remaining slack for each process and tracking their waiting times. These metrics are scaled and combined to determine task priorities, which are then used to assign processes to CPUs or resources. Through simulations, LSTA has demonstrated superior performance in terms of reduced total scheduling time, lower average waiting time, and improved average response time compared to traditional methods. This makes LSTA a robust and efficient solution for real-time systems, capable of managing both critical and non-critical workloads effectively.

Modules of the Project

1. Slack Time Calculation

- Compute remaining slack for each process.
- Update slack dynamically as tasks progress.

2. Starvation Tracking

- Track waiting times for processes.
- Scale starvation impact on priority dynamically.

3. Priority Adjustment

- Combine slack and starvation data to determine task priority.
- Utilize exponential scaling for adaptability.

4. Scheduling and Execution

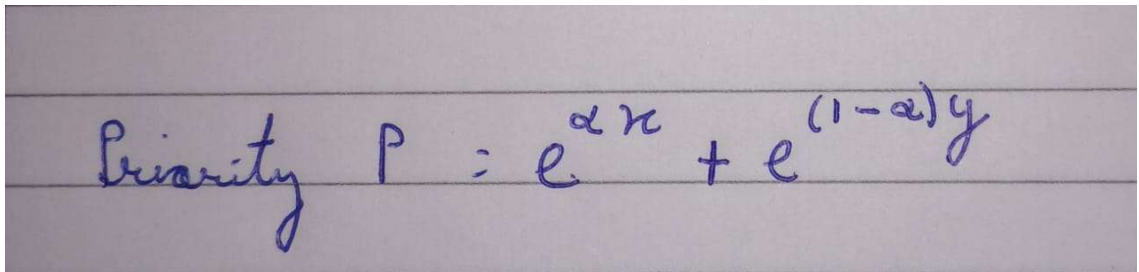
- Assign processes to CPUs or resources based on calculated priorities.
- Simulate task execution and log metrics like utilization and deadlines met.

Algorithm

- 1) All Processes are noted and stored in a structure or in an array along with its Arrival Time, Burst Time and Deadline. The time elapsed (or starvation time) for each process is set to 0.
- 2) A loop is started which goes on until all processes are completed. Time is incremented by 1 second (for this case) in each iteration.
- 3) For each process that has arrived, the slack time is calculated.
- 4) The slack time and starvation time is scaled in a range from 0-1 using MinMax scaling. (Slack Time has to be inversely scaled)
- 5) Using a formula which includes both the scaled starvation time and slack time, we calculate a priority.
- 6) The process with the max priority is processed for one second (for this case). If the max priority is equal between two or more processes, the process is determined on the basis of FCFS.

- 7) If the process has completed its execution, then its completion status is changed.
- 8) Increase the starvation time of each process by one second (for this case).
- 9) End loop

Formula for Calculating the priority:



A photograph of a piece of lined paper with the formula for Priority P written in blue ink. The formula is:
$$\text{Priority } P = e^{\alpha x} + e^{(1-\alpha)y}$$

Where,

x is the scaled Slack Time value

y is the scaled Starvation Time value

Alpha is a parameter, ranging from 0 to 1, which controls the relative weightage of the two evaluation criteria, Slack Time and Starvation Time.

CODE (IN C)

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <math.h>
```

```
struct Process
```

```
{
```

```
    char id[5];
```

```
    int arrival_time;
```

```
    int burst_time;
```

```
    int deadline;
```

```
    int completion_time;
```

```
    int slack_time;
```

```
    int starvation_time;
```

```
    double slack_time_scaled;
```

```
    double starvation_time_scaled;
```

```
    double priority;
```

```
};
```

```
double alpha = 0.5;
```

```
int is_all_completed(struct Process processes[], int n)
```

```
{
```

```
    for (int i = 0; i < n; i++)
```

```
    {
```

```
        if (processes[i].burst_time > 0)
```

```
            return 0;
```

```
    }
```

```
    return 1;
```

```
}
```

```
void find_slack_time(struct Process processes[], int n, int current_time)
```

```
{
```

```

for (int i = 0; i < n; i++)
{
    if (processes[i].arrival_time <= current_time && processes[i].burst_time > 0)
    {
        processes[i].slack_time = processes[i].deadline - current_time - processes[i].burst_time;
    }
}
}

```

```

int max_slack_time, min_slack_time, max_starvation_time, min_starvation_time;

```

```

void find_max_min(struct Process processes[], int n, int current_time)

```

```

{
    int this_slack = 1, this_starvation = 1;
    for (int i = 0; i < n; i++)
    {
        if (processes[i].arrival_time <= current_time && processes[i].burst_time > 0)
        {
            if (this_slack)
            {
                this_slack--;
                max_slack_time = processes[i].slack_time;
                min_slack_time = processes[i].slack_time;
            }
            else
            {
                if (processes[i].slack_time < min_slack_time)
                    min_slack_time = processes[i].slack_time;
                if (processes[i].slack_time > max_slack_time)
                    max_slack_time = processes[i].slack_time;
            }
            if (this_starvation)
            {

```

```

        this_starvation--;

        max_starvation_time = processes[i].starvation_time;
        min_starvation_time = processes[i].starvation_time;
    }
    else
    {
        if (processes[i].starvation_time < min_starvation_time)
            min_starvation_time = processes[i].starvation_time;
        if (processes[i].starvation_time > max_starvation_time)
            max_starvation_time = processes[i].starvation_time;
    }
}
}
}

```

```

void scale_starvation_time(struct Process processes[], int n, int current_time)
{
    if (min_starvation_time == max_starvation_time)
    {
        for (int i = 0; i < n; i++)
            processes[i].starvation_time_scaled = 0.5;
        return;
    }
    for (int i = 0; i < n; i++)
    {
        if (processes[i].arrival_time <= current_time && processes[i].burst_time > 0)
            processes[i].starvation_time_scaled = (double)(processes[i].starvation_time - min_starvation_time) /
(max_starvation_time - min_starvation_time);
    }
}

```

```

void scale_slack_time(struct Process processes[], int n, int current_time)
{

```



```

if (min_slack_time == max_slack_time)
{
    for (int i = 0; i < n; i++)
        processes[i].slack_time_scaled = 0.5;
    return;
}
for (int i = 0; i < n; i++)
{
    if (processes[i].arrival_time <= current_time && processes[i].burst_time > 0)
        processes[i].slack_time_scaled = 1 - (double)(processes[i].slack_time - min_slack_time) /
(max_slack_time - min_slack_time);
}
}

```

```

void find_priority(struct Process processes[], int n, int current_time)
{
    for (int i = 0; i < n; i++)
    {
        if (processes[i].arrival_time <= current_time && processes[i].burst_time > 0)
            processes[i].priority = exp(alpha * processes[i].slack_time_scaled) + exp((1 - alpha) *
processes[i].starvation_time_scaled);
    }
}

```

```

int find_highest_priority_process(struct Process processes[], int n, int current_time)
{
    int highest_priority_index = -1;
    double highest_priority = 0.0;

    for (int i = 0; i < n; i++)
    {
        if (processes[i].arrival_time <= current_time && processes[i].burst_time > 0)
        {
            if (processes[i].priority > highest_priority)

```

```

        {
            highest_priority = processes[i].priority;
            highest_priority_index = i;
        }
    }
}

return highest_priority_index;
}

int main()
{
    int num_processes;
    int current_time = 0, idle_time = 0;

    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);

    struct Process processes[num_processes];

    for (int i = 0; i < num_processes; i++)
    {
        printf("Enter details for Process %d:\n", i + 1);
        printf("Process ID: ");
        scanf("%s", &processes[i].id);
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Deadline: ");
        scanf("%d", &processes[i].deadline);
        processes[i].starvation_time = 0;
        printf("\n");
    }
}

```

```
}
```

```
// printf("Process scheduling using Least Slack Time (LST) with Arrival Time:\n\n");
```

```
while (!is_all_completed(processes, num_processes))
```

```
{
```

```
    find_slack_time(processes, num_processes, current_time);
```

```
    find_max_min(processes, num_processes, current_time);
```

```
    scale_slack_time(processes, num_processes, current_time);
```

```
    scale_starvation_time(processes, num_processes, current_time);
```

```
    find_priority(processes, num_processes, current_time);
```

```
    int process_index = find_highest_priority_process(processes, num_processes, current_time);
```

```
    if (process_index == -1)
```

```
    {
```

```
        idle_time++;
```

```
        current_time++;
```

```
        printf(" %d idle %d,", current_time - 1, current_time);
```

```
        continue;
```

```
    }
```

```
    processes[process_index].burst_time--;
```

```
    current_time++;
```

```
    if (processes[process_index].burst_time == 0)
```

```
        processes[process_index].completion_time = current_time;
```

```
    printf(" %d %s %d,", current_time - 1, processes[process_index].id, current_time);
```

```
    for (int i = 0; i < num_processes; i++)
```

```
    {
```

```
        if (processes[i].burst_time > 0 && processes[i].arrival_time < current_time && i != process_index)
```

```
        {
```

```
            processes[i].starvation_time++;
```

```
        }
```

```
    }
```

```

    }

    printf("\nAll processes finished.\n");

    for (int i = 0; i < num_processes; i++)

        printf("Process %s - Completion Time: %d, Deadline: %d\n", processes[i].id, processes[i].completion_time,
        processes[i].deadline);

    return 0;
}

```

Sample Input:

Process ID	Arrival Time	Burst Time	Deadline
P1	0	5	15
P2	2	3	10
P3	4	7	12

Sample Output (For Alpha = 0.5):

0 P1 1, 1 P1 2, 2 P2 3, 3 P1 4, 4 P2 5, 5 P1 6, 6 P3 7, 7 P2 8, 8 P1 9, 9 P3 10, 10 P3 11, 11 P3 12, 12 P3 13, 13 P3 14, 14 P3 15,

All processes finished.

Process P1 - Completion Time: 9, Deadline: 15

Process P2 - Completion Time: 8, Deadline: 10

Process P3 - Completion Time: 15, Deadline: 12

Case Study

Efficient process scheduling is crucial in operating systems to maximize resource utilization and ensure timely task execution. This case study presents a detailed comparative analysis of two scheduling algorithms: **Least Slack Time (LST)** and **Least Slack Time with Aging (LSTA)**.

The **Least Slack Time (LST)** algorithm prioritizes processes with the minimum slack time, calculated as the difference between a process's deadline, remaining burst time, and the current time. It aims to meet deadlines and minimize idle CPU time by focusing on processes most at risk of missing their deadlines.

On the other hand, the **Least Slack Time with Aging** algorithm incorporates a hybrid approach by considering both slack time and starvation time. STAS uses scaled factors of these metrics, adjusted dynamically, to compute a priority score for each process. This strategy mitigates the issue of starvation, ensuring fairness while still prioritizing critical tasks.

The case study compares the algorithms using a simulation of 10000 processes with randomly generated arrival times, burst times, and deadlines. The algorithms are assessed on the following metrics:

Total Time Taken

CPU Usage

Throughput

Average Waiting Time

Average Response Time

Results

For the LST Algorithm:

Total Time Taken: 55312.00 units

CPU Usage: 100.00%

Throughput: 0.18 processes/unit time

Average Waiting Time: 29098.68 units

Average Response Time: 27023.71 units

For the LSTA Algorithm:

Total Time Taken: 54694.00 units

CPU Usage: 100.00%

Throughput: 0.18 processes/unit time

Average Waiting Time: 28506.86 units

Average Response Time: 26502.61 units

Findings

The LSTA algorithm completed the scheduling approximately 618 units faster than LST, indicating improved efficiency due to its prioritization mechanism.

Both algorithms utilized 100% of the CPU, signifying no idle time during the process execution.

Both algorithms achieved the same throughput, showing that the number of processes completed per unit of time was consistent.

The LSTA algorithm reduced the average waiting time by approximately 592 units, demonstrating better handling of process scheduling by considering starvation alongside slack time.

The LSTA algorithm achieved an average response time approximately 521 units lower than LST, indicating improved responsiveness, particularly for processes that would have experienced starvation in the LST approach.

The **LSTA algorithm** outperforms the LST algorithm in terms of **total time taken**, **average waiting time**, and **average response time**, making it more efficient and equitable for scheduling large numbers of processes.

Despite these advantages, both algorithms have similar **CPU usage** and **throughput**, suggesting that the improvement is primarily in the quality of scheduling and fairness rather than the volume of tasks completed

Comparison

1) Least Slack Time First

Advantages:

- i) Reduced Starvation: as aging is included
- ii) Enhanced Adaptability: due to customizable alpha value

Disadvantages:

- i) Increased Complexity: extra calculation is needed to get priority
- ii) Potential Deadline Misses: Deadlines can be missed more as aging is also considered

2) Earliest Deadline First

Advantages:

- i) Starvation Management: in EDF, one process can monopolize the cpu if it has an early deadline.

- ii) Flexibility with Deadline management: due to customizable alpha value

Disadvantages:

- i) Lower Deadline adherence: Deadlines can be missed more as aging is also considered
- ii) Increased Overhead Cost: due to extra calculations

3) Enhanced EDF-LST Algorithms

A recent study explores an integrated approach that combines Earliest Deadline First (EDF) with Least Slack Time (LST) to improve scheduling for real-time tasks, especially for strict deadline adherence. The two aspects combined handles resource allocation better when the system utilization is near maximum. However, like LST, it is also computationally expensive.

4) Least Slack Time Rate (LSTR) Algorithm

The LSTR algorithm extends LST by considering the rate at which slack time decreases relative to task deadlines. This approach allows for better prioritization in multiprocessor environments by minimizing idle times. However, LSTR's main limitation is that it assumes all tasks are periodic and may not perform as well with a mix of periodic and aperiodic tasks.

Advantage:

- i) Fair Task Treatment
- ii) Adaptable

Disadvantage:

- i) Complexity (but same with LSTR and EDF-LST)

Conclusion

The proposed **Least Slack Time Scheduling with Aging (LSTA)** algorithm demonstrates significant improvements in real-time system scheduling by effectively balancing slack time and starvation metrics. Through dynamic priority adjustments, LSTA ensures fair and responsive task scheduling, outperforming traditional algorithms like EDF and LST in terms of efficiency and fairness.

The case study results highlight that LSTA reduces total scheduling time, average waiting time, and average response time compared to LST, while maintaining 100% CPU utilization and consistent throughput. These findings underscore the algorithm's capability to handle mixed workloads in dynamic environments, making it a robust solution for real-time systems.

In summary, LSTA offers a promising approach to real-time scheduling, addressing the limitations of existing algorithms and providing a more adaptable and equitable framework for managing critical and non-critical workloads.