COMP2432 Operating Systems Group Project Report – Group 3
The *SmarkPark* Management System – SPMS

Yuqi WANG[1], Xikun YANG[1], Siyuan LIU[1], and Yixiao JIN[1]

Abstract—Efficient offline scheduling is a critical challenge in many real-world applications. In this project report, we investigate a parking lot booking scenario by developing the *SmartPark* Management System (SPMS), which implements three distinct scheduling algorithms — FCFS, PRIO, and OPTI — each executed concurrently using `fork()` to minimize latency.

For OPTI, we present GAPS, an optimization-based stochastic greedy scheduling algorithm. GAPS integrates advanced data structures and techniques, including Lazy-Propagated Segment Tree, dynamic array, and the high-performance Xorshift128+, collectively enhancing computational efficiency. Extensive simulated experiments across diverse data distributions demonstrate that GAPS consistently outperforms both FCFS and PRIO.

I. INTRODUCTION

Multi-class offline scheduling scheduling problems can be found in many real-world scenarios. Distributed training of deep neural networks, for examples, requrires coordination of various types of resources such as CPU, GPU, and specialized accelerators. Under-utilization of these expensive hardware can be highly wasteful. This project aims to address one of such multi-class resource allocation challenge of parking lot booking, where the primary objective is to maximize facility utilization.

Conventional methods such as First-Come First-Serve (FCFS) and handcrafted priority-based scheduler (PRIO) inherently lacks flexibility, often leading to suboptimal results and lost of potential revenue. Algorithms based on Dynamic Programming or Network Flow can achieve optimal results in single-class offline scheduling problems, but rapidly deteriorates in multi-class settings due to NP-hard nature of conflict resolution. Simple greedy based algorithms such as Shortest Job First (SJF) or Longest Job First (LJF) performs more consistently in such setting but remain constrained by their rigid decision rules.

To address this limitation, we present **G**reedy **A**nnealed **P**robabilistic **S**cheduler, or **GAPS**, for our OPTI. This novel approach combines an enhanced greedy algorithm foundation with stochastic optimization techniques, iteratively refining arrangements through principles adapted from Thermal Dynamics. Extensive experimental results demonstrate that GAPS delivers substantially superior perfor-

Algorithm Comparison (num_trials=8)
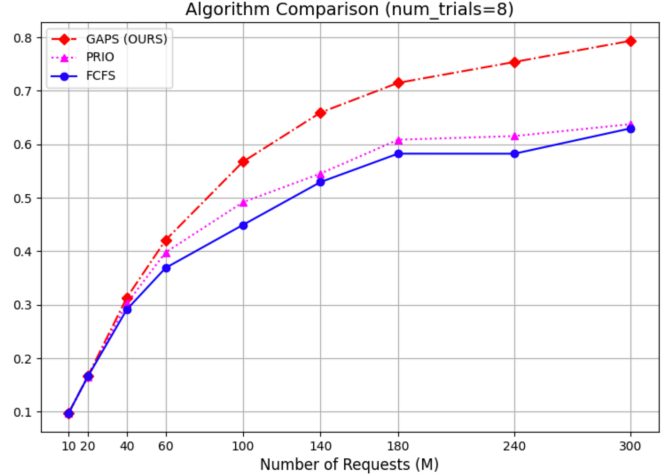
-mance compared to both FCFS and PRIO approaches across diverse scheduling scenarios.

Package into a comprehensive *SmartPark* Management System (SPMS), we provide seamless user experience when interacting with these three algorithms while maintaining exceptional speed. SPMS is highly computationally efficient. Leveraging system calls such as `fork()` and `pipe()` [1], SPMS executes all three schedulers concurrently, reducing response latency. We further optimize query processing through lazy-propagated segment tree, reducing time complexity from linear time to $O(\log N)$ with a minimal constant factor [2], [3], [4]. An extremely efficient Pseudo-Random Number Generator (PRNG) is also implemented to support the stochastic process of **GAPS** [5].

TL;DR
Our innovations can be summarized as:
1) Creation of **GAPS**, a stochastic optimization algorithm that outperforms FCFS, PRIO, and most traditional approaches.
2) Implemented concurrent scheduler execution via system-level parallelism through `fork()`,`pipe()`,`read()`,`write()`
3) Optimization of query complexity from $O(N) \rightarrow O(\log N)$ by implementing lazy-propagated segment tree
4) Integration of high-performance Xorshift128+ PRNG and Dynamic Arrays to handle arbitrary number of requests

II. SCOPE AND RELATED WORK

A. Multiprocessing

The `fork()` system call is used extensively as a foundational component in SPMS. `fork()` spawns a new child process by duplicating

parent process's memory space [1]. For our implementation, we fork SPMS into two processes:

1) The main process handles user input (including batch files)
2) The child processes is dedicated handling `printBooking` by coordinating FCFS, PRIO, and OPTI, and printing booking information and/or the summary report.

Notably, when the user enter `printBooking -ALL`, the child process further forks three sub-processes, allowing for system-level parallelism when executing the three scheduling algorithms, reducing latency. Their synchronization mechanisms are discussed in Section II.B.

To avoid accumulation of zombie processes and thus potential memory leaks, we call `exit(0)` within each child process and `wait(0)` in parent process to guarantee child process have terminated successfully.

B. Interprocess Communication

We extensively utilize `read()` and `write()` to communicate between child processes spawned for each scheduler through pipes created by `pipe()` [1]. The benefit of this approach is that it allows us to synchronize their progress, and maintaining output order and format.

For example, when FCFS, PRIO, and OPTI are spawned in three separate child processes via `fork()`, they can execute and finish in arbitrary order. If they finish and hence output at the same time, their results can become mixed up. Through synchronization mechanisms implemented through `read()` and `write()`, however, we can mitigate this issue by having the parent process coordinate their output order, letting schedulers who finished early to wait for pending ones.

An overview of its usage can be found in Section V.D.

C. Memory management

Unlike languages with automated memory handling mechanisms such as Objective-C, Java, and Swift, or even C++'s RAII-based smart pointer abstractions (`std::unique_ptr` and `std::shared_ptr`), the position of C as a systems programming language requires us to implement explicit memory management [6].

We use `malloc()` to allocate memory, `realloc()` for dynamic resizing of `Vector`, and `free()` to deallocate no longer needed memory. We employ `memset()` and `memmove()` for array initializations and string manipulations, which provide performance advantage through direct memory access.

Sophisticated data structures implemented in our system such as Dynamic queues and Segment Trees (detailed in Section III), require specialized deep-copy and deep-free operations to handle nested pointer structures and prevent memory leak.

III. CONCEPT: DATA STRUCTURES

This section details the pre-requite concepts mentioned throughout this report, as well as providing a deep dive into some of the algorithms and data structures introduced to greatly improve the efficiency and efficacy of SPMS. The section can be organized as follows:

1) Section III.A and B discuss the motivation and usefulness of our data structures (Dynamic Array and Segement Tree)
2) Section III.C brief the implementation of the three schedulers

3) Section III.D formally defines our assumptions *Utilization* used as the objective of OPTI.

A. Dynamic Array

Since user entered requests as well as requests from batch files can be arbitrary in size, we want an array-like data structure that is able to:

1) retrieve or modify elements in $O(1)$ as in regular arrays
2) automatically adapt to growing number of requests.

To address this, we draw inspiration from C++'s `std::vector`, which creates a new array whose size if 2x larger than the original, whenever the current size exceeds capacity, and copies the original content to the new array.

---

function vector-add(**vector instance:** *vec*, **new element to add:** *elem*)

---

1 **if** *vec.size* > *vec.capacity* **then**

2   *new_capacity* ← *vec.capacity* · 2

3   reallocate a size of *new_capacity* for *vec*

4   move the existing elements of *vec* into the newly allocated space

5   *vec.capacity* ← *new_capacity*

6 *vec*[*vec.size*] ← *elem*

7 *vec.size* ← *vec.size* + 1

Our custom implemented `Vector` data structure is dedicated to store a struct type called `Request`, which stores compact representation of request informations.

Intuitively, one might think that such data structure will degrade to linear insertion performance due to having to perform an $O(N)$ resize operation every N insertions. However, it can be shown that the amortized performance remains $O(1)$ [7]. Here we provide a simple proof using potential function analysis:

Let the cost of inserting the i-th element be t_i, where

$$t_i := \begin{cases} 1 & \text{if } s_{i-1} < c_{i-1} \\ c_{i-1} + 1 & \text{if } s_{i-1} = c_{i-1} \end{cases} \quad (1)$$

Define the potential function

$$\Phi(i) = 2s_i - c_i, \Phi(i) \geq 0, \forall i \in [0, N) \quad (2)$$

Thus, the amortized cost $\hat{t}_i$ is

$$\hat{t}_i = t_i + \Phi(i) - \Phi(i-1) \quad (3)$$

Scenario 1: No resize needed ($s_{i-1} < c_{i-1}$)

- Actual cost $t_i = 1$
- Change in potential:

$$\Phi(i) - \Phi(i-1) = (2s_i - c_i) - (2s_{i-1} - c_{i-1})$$
$$= 2(s_{i-1} + 1) - c_{i-1} - (2s_{i-1} - c_{i-1}) \quad (4)$$
$$= 2$$

- Amortized cost: $\hat{t}_i = 1 + 2 = 3$

Scenario 2: Resize needed ($s_{i-1} = c_{i-1}$)

- Actual cost $t_i = c_{i-1} + 1$
- Change in potential:

$$\Phi(i) - \Phi(i-1) = (2s_i - c_i) - (2s_{i-1} - c_{i-1})$$
$$= 2 - c_{i-1} \quad (5)$$

- Amortized cost: $\hat{t}_i = (c_{i-1} + 1) + (2 - c_{i-1}) = 3$

Therefore, whether resize or not the amortized cost remains constant ($\hat{t}_i = 3$). Therefore, the amortized cost per insertion is $O(1)$, instead of the seemingly $O(N)$

B. Lazy Propagation Segment Tree

Typically, checking overlaps of two intervals require linear searching through all intervals in the set, resulting in an $O(n)$ cost. For large number of requests, this can be very inefficient. By using the idea divide-and-conquer, we can maintaining intervals with a tree structure, which reduces the time complexity of interval queries down to $O(\log n)$.

Specifically, we implemented a **k-parallel lazy propagation segment tree**. Or, in simpler terms, k segment trees with lazy tagging packed in parallel to allow for batch querying.

Our segment tree mainly supports two functionalities:
1) `segtree_range_set`: which sets the value of an interval to a specific value in $O(\log N)$
2) `segtree_range_query`: which queries the maximum value in the provided interval in $O(\log N)$

The lazy tagging part allows us to delay the propagation of new values to leaf nodes, thus preventing the efficiency of range update to degrade to $O(N)$ during rapid large interval range set operations. Whereas the the k-parallel aspect makes it easy to represent one type of resource as a single `SegTree` instance.

Formally, we define
- *segtree*: as the segment tree instance.
- *p*: as the node index of a segment tree. We use *segtree*[p] to access the node p of the segment tree. For example, if the required interval is $[l_{\min}, l_{\max}]$, then the interval represented by *segtree*[1] is

$$[l_{\min}, l_{\max}] \tag{6}$$

Then, *segtree*[2] would represent

$$\left[l_{\min}, \quad \frac{l_{\min} + l_{\max}}{2} \right] \tag{7}$$

while *segtree*[3] would represent

$$\left[\frac{l_{\min} + l_{\max}}{2} + 1, \quad l_{\max} \right] \tag{8}$$

More generally, for any node p, the index of its left child is $2p$, and the index of its right child is $2p + 1$. The left child represents the smaller half of the interval represented by this node, and the right child represents the bigger half of the interval.
- *ifLazy*: as the lazy tag array. If the current node stores values that needs to be updated but has not yet been propagated downward, the value of its *ifLazy* flag should be set to true.
- *lazy*: as the lazy value array. *lazy*[p] stores the value that node p need to propagate downward.

Then, the lazy propagated segement tree with range-max and range-set functions can be roughly outlined as following:

---

function range-max(**segment tree instance:** *segtree*, **resource type:** k, **query range:** l, r, **current range:** cl, cr, **current node:** p)

---

1 **if** $l > cr$ **or** $r < cl$ **then**
2 | **return** -1
3 **if** $l \leq cl$ **and** $r \geq cr$ **then**
4 | **return** *segtree*.tree[k][p]
5 maintain-max(*segtree*, k, cl, cr, p)
6 $cm \leftarrow cl + (cr - cl) / 2$
7 *left_max* $\leftarrow$ range-max(*segtree*, k, l, r, cl, cm, $2p$)
8 *right_max* $\leftarrow$ range-max(*segtree*, k, l, r, $cm + 1$, cr, $2p + 1$)
9 **return** max(*left_max*, *right_max*)

---

function range-set(**segment tree instance:** *segtree*, **resource type:** k, **set range:** l, r, **value:** *val*, **current range:** cl, cr, **current node:** p)

---

1 **if** $l > cr$ **or** $r < cl$ **then**
2 | **return**
3 **if** $l \leq cl$ **and** $r \geq cr$ **then**
4 | *segtree*.lazy[k][p] $\leftarrow$ *val*
5 | *segtree*.ifLazy[k][p] $\leftarrow$ **true**
6 | *segtree*.tree[k][p] $\leftarrow$ *val*
7 | **return**
8 maintain-max(*segtree*, k, cl, cr, p)
9 $cm \leftarrow cl + (cr - cl) / 2$
10 range-set(*segtree*, k, l, r, *val*, cl, cm, $2p$)
11 range-set(*segtree*, k, l, r, *val*, $cm + 1$, cr, $2p + 1$)
12 *segtree*.tree[k][p] $\leftarrow$ max(*segtree*.tree[k][2p], *segtree*.tree[k][2p + 1])
13 **return**

The k-parallel segment tree (`SegTree`) is used in the following way:
- We assign one such `SegTree` for each type of resources, and the k parameter set to the number of resources of that type. Therefore, we would have four `SegTree` in total, where $k = 10, 3, 3, 3$ for each type of resource respectively.
- The tree spans the interval of [05-10 0:00, 05-16 23:59], which translates to [0, 10080) in integer minute representation
- When user makes a request, the corresponding time range for the relevant resource gets marked with its index

C. Concept: Schedulers

Three schedulers are implemented:
- First-Come-First-Serve Scheduler (FCFS),
- Priority Scheduler (PRIO),
- Optimal Scheduler (OPTI).

This section focuses primarily on the first two, FCFS and PRIO. Whereas, the OPTI scheduler is discussed in detail in Section IV.

a) *The FCFS Scheduler:*

The FCFS scheduler is the simplest scheduler. As the name suggests, we schedule user requests non-preemptively based on insertion order. The pseudocode for the FCFS scheduler is as follows:

---

function run-fcfs(**vector of user requests:** *reqs*)

---

1 **for** *request* **in** *reqs* **do**
2   query the segment tree using `segtree_range_query`
3   **if not** the requested resources are all occupied **thens**
4   accept request (`vector_add(accepted, request)`)
5   mark range occupied (`segtree-range-set`)
6   update the statistics
7   **else**
8   reject this request

---

b) *The PRIO Scheduler:*

We assign different priority level to each types of request, closely referred to the requirements outlined in the instruction document. Specifically, the their priority levels are (lower is higher priority):

TABLE I.   PRIORITY LEVELS OF DIFFERENT TYPES OF REQUEST

Request Type	Calling Method	Priority Level
Event	*addEvent*	**0** (highest priority)
Reservation	*addReservation*	**1**
Parking	*addParking*	**2**
Essentials	*bookEssentials*	**3** (lowest priority)

Implementing PRIO, is as simple as first sorting all the requests by their priority level. Aftwards, the rest of the process is the same as FCFS. The pseudocode for the PRIO scheduler is as follows:

---

function run-prio(**vector of user requests:** *reqs*)

---

1 Sort requests in *reqs* based on priority level, for requests with the same priority level, those with a start time between 08:00 AM and 08:00 PM can receive an additional higher priority
2 **for** *request* **in** *reqs* **do**
3   query the segment tree using `segtree_range_query`
4   **if not** the requested resources are all occupied **then**
5   accept request (`vector_add(accepted, request)`)
6   mark range occupied (`segtree-range-set`)
7   update the statistics
8   **else**
9   reject this request
10 **return** the scheduling results

D. Resource Utilization Rate

We noticed that the project description PDF mentioned to *better performance to utilize the facilities in PolySPMS*. Since the problem statement did not clearly specify the formula of the metric *utilize*, here we provide our own definitions:

Let their be N types of resources, each with m_i instances ($1 \leq i \leq N$). Let's say we are also given a set of P accepted requests $R = \{r_1, r_2, ..., r_P\}$, each described by a time interval $[l_k, r_k]$ and a counter c_k of the types of resources occupied, where ($1 \leq k \leq P$).

Assuming that all of R can be fitted into the resource without collisions, we can then write the utilization rate for the i-th type of resource can be calculated as follows:

$$u(i) = \frac{\sum_{k=1}^{P} \mathbb{I}(r_k, i) \cdot (r_k - l_k + 1)}{(T_{\max} - T_{\min} + 1) \cdot m_i} \tag{9}$$

Where,

$$\mathbb{I}(r_k, i) = \begin{cases} 1 & \text{if } r_k \text{ requested } i\text{-th type resource} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Since minute is used as the smallest unit of time, and that the SPMS is tested only from 10 to 16 May, 2025, the equation can be simplified to:

$$u(i) = \frac{\sum_{k=1}^{P} \mathbb{I}(r_k, i) \cdot (r_k - l_k + 1)}{10080 \cdot m_i} \tag{11}$$

Finally, the total utilization of facilities can be defined as:

$$U = \frac{\sum_{i=1}^{N} u(i) \cdot m_i}{\sum_{i=1}^{N} m_i} \tag{12}$$

This is the **optimization objective** we use for OPTI.

IV. THE OPTI SCHEDULER

For our OPTI algorithm, we present **G**reedy **A**nnealed **P**robabilistic **S**cheduler, or **GAPS**. It combines an improved Longest-Job-First (LJF) greedy strategy foundation with an optimization technique inspired from Thermal Dynamics known as *Simulated Annealing*.

A. Simulated Annealing

***Simulated Annealing* (SA)** is a probabilistic optimization algorithm inspired by the process of annealing in metallurgy. SA is effective for solving complex optimization problems and escaping local minima by initially allowing worsen solutions [8].

In metallurgy, *annealing* means to heat a metal material to a high temperature and then *gradually* cooling it down. Annealing is designed to make metal atoms settle into a low-energy state and thus minimizing defects [9].

In *Simulated Annealing*, the algorithm starts with an initial solution and a high temperature. New solutions are generated by making perturbation to the current solution.

Suppose that the new solution's total utilization is U', while the previous solution have utilization U. Then, SA accepts new solution with a probability of p_i during the i-th iteration of the optimization process.

$$p_i = \begin{cases} 1 & \text{if } U' > U \\ e^{(U'-U)/T_i} & \text{otherwise} \end{cases} \tag{13}$$

where, T_i is the current temperature during the i-th round.

---

function simulated-annealing(**problem:** *p*, **initial solution:** *s*, **initial temperature:** *T*, **cooling rate:** *decay*)

---

    return *best*
1   *cur* ← *s* // current solution

function simulated-annealing(**problem:** *p*, **initial solution:** *s*, **initial temperature:** *T*, **cooling rate:** *decay*)	

2	*best* ← *s* // best solution
3	*temp* ← *T* // temperature
4	**while** *temp* > minimum threshold **do**
5	*new* ← perturb(*cur*)
6	*delta* ← *U*(*new*) - *U*(*cur*)
7	**if** *delta* > 0 or random[0,1) < exp(*delta* / *temp*) **then**
8	*cur* ← *new*
9	**if** quality(*cur*) > quality(*best*) **then**
10	*best* ← *cur*
11	*temp* ← *temp* · *decay*^{t} // use the fast pow algorithm to calculate this exponentiation. See appendix.

The perturbation process is defined by two parameters p and q. Each round, the function randomly deletes requests from the current solution with probability of q; then, it greedily adds back resources with a probability of p after first sorting the available requests by "volume then resource count". This process is implemented as the functions `opti_greedy` and `opti_delete` within our codebase.

V. SOFTWARE STRUCTURE

Our program is well-organized and carefully designed. We have also integrated the concepts of **event-driven programming** and **object-oriented programming**.

A. Source code structure

All source code is saved in the `src/` directory, as clearly evidenced by our repository structure.

B. Logical structure and modules

Our program is divided into multiple modules, each consisting of one or more `.c` or `.h` files. We present the logical structure of our program along with its corresponding `.c` or `.h` files.

C. Multiprogramming and multi-process design

Our program utilizes multi-process technique.

1) **Main Process I:** The main process is responsible for running the "*console*", which is the Input Module. The main process also includes storing data.
2) **Main Process II:** The main process also serves the function of running the *recursive* process batch feature. This is highly useful because a batch file may contain commands for another process batch within it. If a fatal error occurs, we expect the program to terminate immediately. Therefore, we place this process within the main process.
3) **Scheduling Processes:** We design a separate process for each scheduler. Therefore, this section comprises three distinct processes: the FCFS Scheduler Process, the PRIO Scheduler Process, and the OPTI Scheduler Process. Each process is designed to handle specific scheduling algorithms independently. Here, we adopt **the principle of non-interference**,

meaning that one scheduling algorithm should not modify the data being used by another scheduling algorithm, nor should it be influenced by the operations of other scheduling algorithms.

4) **Output Processes:** These processes are designed for output. Therefore, they are critical components of both Part III and Part IV. We designed these processes to achieve **the principle of component separation**, meaning that all functionalities should not be consolidated into a single process. Specifically, different scheduling algorithms correspond to distinct *printBookings* processes, and the feature for printing summary report is also handled in a separate process.

For process creation, we utilize the `fork()` system call. Additionally, in the main process (or the parent process of a specific process, as the Output Process may sometimes be directly derived from the Scheduling Process to improve efficiency), we use `wait(NULL)` to wait for the completion of child processes.

Below, we present the `main()` function involved with the multi-process component.

TABLE II. COMPONENTS OF THE PROGRAM

Module Name	Description	Filenames
Main Entry	Entry point of the program	`SPMS.c`
Encapsulated Structures	`Statistic` for managing statistical data `Tracker` for interval-based information in segment trees	`state.h,` `state.c`
Data Structure Dynamic Array	Implementation of dynamic array	`vector.h,` `vector.c`
Data Structure Lazy SegTree	Implementation of k-parallel lazy propagation segment tree for efficient range queries and updates	`segtree.h,` `segtree.c`
Utilities	Includes utility functions such as date parsing and essential pair checking	`utils.h,` `utils.c`
RNG	Implementation of the Xorshift128+ random number generator	`rng.h, rng.c`
Part I Input Module	Handles user input	`input.h,` `input.c`
Part II Scheduling Module	Handle scheduling algorithms	`scheduler.h,` `scheduler.c`
Part II The OPTI Scheduler	Helper methods for OPTI scheduling algorithm	`opti.h, opti.c`
Part III Output Module	Responsible for printing scheduling results	`output.h,` `output.c`
Part IV Analyzer Module	Generates and prints summary reports	`output.h,` `output.c`
Part V Priority Module	Assigns priority levels to different request types	`utils.h,` `utils.c`

function main()

1	*reqs* ← an empty *Vector* for storing user requests
2	**while** receiving user inputs **do**
3	**if** user requests for parking, reservations, events, or booking essentials **then**
4	│ add the requests to *reqs*
5	**else if** user asks to print bookings **then**
6	**fork new processes**, run the required schedulers (run-fcfs, run-prio and/or run-opti) in the child process
7	**fork new processes**, print all required booking information and summary report (if applicable) in the child process
8	**else if** user asks to process batch file **then**
9	│ process the batch file in the main process
10	**else if** user asks to quit the program **then**
11	│ **break**

D. Inter-process communication design

Our program includes numerous inter-process communication (IPC) components, and we have implemented these communications using `pipe()`, `write()`, `read()`, and `close()`.

One of our major IPC examples occurs when the user issues the `printBookings -all` command. In this scenario, the booking information from the three scheduling algorithms must be output in a specific order: **(FCFS, PRIO, OPTI)**. Any other sequence is unacceptable. In addition to outputting the booking information of the three scheduling algorithms, the Summary Report should also be generated immediately afterward.

function main-output-process(**all statistics information:** *stats*)

1	// This process is the main process of the Output Processes.
2	
3	**if** need to print the booking info of the FCFS scheduler **then**
4	│ **fork a new process**, run print-fcfs(*stats.fcfs*)
5	**if** need to print the booking info of the PRIO scheduler **then**
6	│ **fork a new process**, run print-fcfs(*stats.prio*)
7	**if** need to print the booking info of the OPTI scheduler **then**
8	│ **fork a new process**, run print-fcfs(*stats.opti*)
9	
10	**if** need to print the booking info of the FCFS scheduler **then**
11	│ tell the FCFS scheduler process to print its booking information
12	│ wait for the FCFS scheduler to finish printing
13	**if** need to print the booking info of the PRIO scheduler **then**
14	│ tell the PRIO scheduler process to print its booking information
15	│ wait for the PRIO scheduler to finish printing
16	**if** need to print the booking info of the OPTI scheduler **then**
17	│ tell the OPTI scheduler process to print its booking information
18	│ wait for the OPTI scheduler to finish printing
19	
20	**if** need to print the summary report **then**
21	│ Ask FCFS process to print its part of the summary report.

function main-output-process(**all statistics information:** *stats*)

22	│ wait for the FCFS scheduler to finish printing
23	│ Ask PRIO process to print its part of the summary report.
24	│ wait for the PRIO scheduler to finish printing
25	│ Ask OPTI process to print its part of the summary report.
26	│ wait for the OPTI scheduler to finish printing
27	**else**
28	│ tell the FCFS scheduler process to exit (if applicable)
29	│ tell the PRIO scheduler process to exit (if applicable)
30	│ tell the OPTI scheduler process to exit (if applicable)
31	**return //** or **exit**

For the functions print-fcfs, print-prio, and print-opti, their logic is identical. Therefore, we will only provide the implementation details for run-fcfs as a representative example.

function print-fcfs(**statistics information of FCFS:** *stats*)

1	wait approval from main output process
2	print the booking info of the FCFS scheduler
3	tell the main output process that the printing task has been completed
4	wait the next message from the main output process
5	**if** the main output process asks to print the summary report
6	│ print the summary report (OPTI part)
7	**else if** the main output process asks to exit
8	│ **return** or **exit**
9	**return** or **exit**

VI. TESTING CASES AND ASSUMPTIONS

A. Test Sample Generation: Overview

We have carefully designed a large number of practical and reliable test samples to demonstrate the performance of our program and scheduling algorithms. Our assumptions are as follows:

- The time range is from 00:00, May 10 to 23:59, May 16, 2025, spanning a total of 7 days, which is equivalent to 168 hours or 10,080 minutes. Thus our samples will be generated within the range $[0, 168)$ (in hours). We assume that all request data falls within this range. Any requests outside this range (including those with an end time exceeding this range) will be automatically disregarded. **Additionally, note that we have inquired from the course instructor that all samples should be in whole hours.**

- We assume that the minimum parking time is 1 hour, and the maximum is overnight, 14 hours.

- We assume that the start time of the requested parking follows a **bimodal normal distribution**, with the highest demand occurring at 10:00 AM and 2:00 PM. Therefore, there will be 14 peak values within the entire range. We will explain how to model this distribution later.

- The parking time follows an exponential distribution. However, the parameter of the exponential distribution — the expected parking time — is related to the start time of the requests. **Requests made at night are more likely to involve long-term parking (overnight), while requests made during the day are more likely to involve short-term parking (1 to 3 hours).**
- The user's selection of which essential pairs and how many sets of essential pairs is completely random. However, we ensure that they meet the requirements and do not contain too many invalid requests.

B. Test Sample Generation: Multimodal Normal Distribution

We begin by presenting the Probability Density Function (PDF) of a bimodal normal distribution, which is expressed as a weighted sum of two normal distributions:

$$f_{\text{BMN}}(x; \mu_1, \mu_2, \sigma_1, \sigma_2) = w_1 \cdot \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{(x-\mu_1)^2}{2\sigma_1^2}\right) + \\ w_2 \cdot \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left(-\frac{(x-\mu_2)^2}{2\sigma_2^2}\right) \tag{14}$$

subject to the constraint:

$$w_1 + w_2 = 1 \tag{15}$$

In our scenario, the means μ_1 and μ_2 are are periodic and defined as follows:

$$\mu_1 = 10, 34, 58, ... \tag{16}$$

$$\mu_2 = 14, 38, 62, ... \tag{17}$$

Formally, they can be expressed as:

$$\mu_1 \in \{t = 10 + 24k : k \in \mathbb{Z}_{\geq 0} \wedge t < 168\} \tag{18}$$

$$\mu_2 \in \{t = 14 + 24k : k \in \mathbb{Z}_{\geq 0} \wedge t < 168\} \tag{19}$$

For the standard deviations σ_1 and σ_2, we assume the same value:

$$\sigma_1 = \sigma_2 = 1 \text{ hour} \tag{20}$$

Our chosen values are meaningful because many PolyU activities usually start between 9:00-11:00 and 13:00-15:00. Therefore, setting $\sigma = 1$ is reasonable.

For the weights w_1 and w_2, we assume they are equal,

$$w_1 = w_2 = 0.5 \tag{21}$$

This indicates that the number of requests from people coming to PolyU in the morning is equally as frequent as those coming in the afternoon.

We define the sets of possible values for μ_1 and μ_2 as M_1 and M_2, respectively.

Next, we present the distribution function for the start time of requests, denoted as $S(t)$.

$$S(t) = \sum_{d=0}^{6} \left[\frac{1}{14} f_{\text{N}}(t; \mu = 10 + 24d, \sigma = 1) + \\ \frac{1}{14} f_{\text{N}}(t; \mu = 14 + 24d, \sigma = 1) \right] \tag{22}$$

which simplifies to:

$$S(t) = \frac{1}{14} \sum_{d=0}^{6} [f_{\text{N}}(t, \mu = 10 + 24d, \sigma = 1) + \\ f_{\text{N}}(t, \mu = 14 + 24d, \sigma = 1)] \tag{23}$$

where the function $f_{\text{N}}(x; \mu, \sigma)$ is a normal distribution and defined as:

$$f_{\text{N}}(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{24}$$

Note the range of t is $[0, 168]$.

We ignore the cases of $S(t)$ at the beginning and the end of the interval. That is, we assume

$$\int_0^{168} S(t)\mathrm{d}t = \int_{-\infty}^{\infty} S(t)\mathrm{d}t = 1 \tag{25}$$

C. Test Sample Generation: Exponential Distribution

The exponential distribution is a continuous probability distribution often used to **model the duration of time between events in a Poisson process**, where events occur continuously and independently at a constant average rate.

The PDF of an exponential distribution is given by:

$$f(x; \lambda) = \lambda e^{-\lambda x} \tag{26}$$

where $\lambda > 0$ is the rate, representing the rate of occurrence of the event (i.e., the number of events expected to occur per unit of time).

The mean and the variance are given by:

$$\mu = \frac{1}{\lambda}, \quad \sigma^2 = \frac{1}{\lambda^2} \tag{27}$$

In our scenario, we consider λ to be a function of time t. We map the range of $t \in [0.168)$ to a single day range $\tau \in [0, 24)$, that is,

$$\tau = t \bmod 24 \tag{28}$$

We can express the duration function of a request, denoted as $D(t)$:

$$D(\tau) = f(\tau; \lambda(\tau)) = \lambda(\tau) e^{-\lambda(\tau)\tau} \tag{29}$$

Next, we define the parameter function $\lambda(\tau)$. We believe that the duration of overnight requests will be quite long, as cars arriving at night are more likely to stay overnight. At the same time, cars arriving during the day tend to have shorter parking durations. Therefore, we make the following assumptions:

- Cars arriving between 8:00 PM and 4:00 AM will stay for 10 hours on average. That is,

$$\lambda(\tau) = 10, \quad \tau \in [0, 4] \cup [20, 24) \tag{30}$$

- Cars arriving at any other time will stay for an average of 2 hours on average. That is,

$$\lambda(\tau) = 2, \quad \tau \in (4, 20) \tag{31}$$

D. Test Sample Generation: Generate

We use Python to quickly generate a large number of test samples. We have set the sample size as follows: $\{10, 20, 40, 60, 100, 140, 180, 240, 300\}$,

Note that for large sizes, there may be a significant number of misses (requests rejected), and this is when the OPTI scheduling algorithm is truly put to the test. For the results of the test samples across all schedulers, please refer to the Performance Analysis section.

VII. PERFORMANCE ANALYSIS

This section focuses mainly on **intuitive and qualitative** analysis of the performances of each algorithm, both in terms of efficacy and efficiency. For quantitative and visual results, please refer to Section IX.

A. FCFS

Since we did not sort the requests by any specific metric beforehand, the insertion order is completely decided by the input order. In the usual case where input order is unpredictable, FCFS can essentially be seen as random insertion. However, one can easily engineer an adversary input that can degrade its performance significantly (e.g., very scattered short jobs).

B. PRIO

Intuitively, PRIO can be seen as a coarse *More Resource Type First* algorithm, since the priority rules provided in the project description PDF is essentially the expected number of types of resources it will occupy.

However, this strategy presents numerous weaknesses:

1) The expected number of types of resources is at best an "educated guess" based on the upper limit of resource types that the command type can choose, since there is no guarantee that `addEvent` will have more resource types than `addParking`.
2) The algorithm only considers resource types, but not the duration of the requests, making it an even worse estimator of utilization rate as discussed in Section III.D.
3) Similar to FCFS, adversaries can easily deteriorate the system by booking many `Event` requests that are short and sparse.

C. OPTI

In contrast, **GAPS** successfully overcame these limitation:

1) By prioritizing requests with large total time and resource types, we are effectively maximizing utilization rate.
2) By incorporating the stochastic perturbation discussed in Section IV, **GAPS** can escape potential local minima and converging to more optimal solutions than naive LJF.
3) The stochastic process also makes scheduling process less predictable, hence more robust to engineered attackes.

Each round of SA requires iterating over all requests and checking for overlaps. Let the number of requests be N and the max iteration count of SA be M. Then, a naive implementation of SA would be $O(MN^2)$. However, with lazy segment tree, the time complexity is reduced to $O(MN \log N)$, allowing us to handle thousands of requests with ease.

D. Effects of Data Distribution

As discussed earlier, we introduced normal distribution and exponential distribution. In this section, we discuss its effect on the performance of the three scheduling algorithms.

We found that all three algorithms perform worse on these distributions than uniform distributions. However, results in Section IX demonstrates that **GAPS** algorithm remains superior to FCFS and PRIO despite the more challenging test cases.

VIII. PROGRAM SET UP AND EXECUTION

A. Compile using `gcc` **Recommended**

We strongly recommend using `gcc` to compile the project. This compilation method has been tested on *Apollo2*.

To get started, please first `cd` into the project directory (i.e., parent folder of the `src/` folder) of our project. Then, execute the following command to compile our project:

```
gcc -I./src src/*.c -o SPMS -lm -w -O2
```

Alternatively, if the wildcard `*` is not supported:

```
 gcc -I./src src/SPMS.c src/opti.c src/
rng.c src/input.c src/output.c src/
scheduler.c src/utils.c src/vector.c src/
segtree.c src/state.c -o SPMS -lm -w -O2
```

The `libm.a` math library provides many functions declared in `math.h`, though these are not part of `libc.a`. Some overlap exists, as certain functions are included in `libc.a`, which might cause confusion. Generally, the C standard library (`libc`) guarantees support for functions mandated by the ANSI standard. Thus, when relying solely on ANSI standard functions, linking with `-lm` is unnecessary. **To ensure compatibility and prevent potential compilation errors, we routinely include `-lm` in our build commands.**

B. Compile using `CMake`

If none of the GCC commands work, then this is the final alternative-For users of CLion, it is possible to use CMake to compile the project [10]. Please use the following script to build this project with CMake:

```
cmake_minimum_required(VERSION 3.30)
project(COMP2432_GroupProject_SPMS C)

set(CMAKE_C_STANDARD 11)

file(GLOB SOURCES "src/*.c")

add_compile_definitions(DEBUG_MODE)

add_executable(COMP2432_GroupProject_SPMS
${SOURCES})

set(CMAKE_BUILD_TYPE Debug)

if(CMAKE_BUILD_TYPE STREQUAL "Debug")
    add_compile_options(
        -g
        -D_GLIBCXX_DEBUG
        -D_GLIBCXX_DEBUG_PEDANTIC
    )
else()
    add_compile_options(
        -O2
    )
endif()
```

C. Program Execution

Please execute the following command to start the program:

```
./SPMS
```

Before this command, please ensure that your working directory is the one containing the SPMS executable file.

D. Functionality Implementation

After starting the program, users can input the following commands. The end of each command must be a semicolon (`;`). **Incorrect commands will lead to unpredictable results.**

Below are the user commands supported by this program, each presented with a format template and examples. These commands are all required by the instruction document, and the format we handle is entirely consistent with what is provided in the instruction document.

***addParking*:** Book a parking place together with optional essentials devices.

```
addParking -member_[A/B/C/D/E] YYYY-MM-DD
hh:mm n.n [essentials];
addParking -member_A 2025-05-16 10:00 3.0
battery;
```

where,
- *member_[A/B/C/D/E]*: member name;
- *YYYY-MM-DD hh:mm*: booking time;
- *n.n*: duration of time usage;
- *[essentials]*: required essential(s).

***addReservation*:** Book a parking place together with mandatorily required essentials devices.

```
addReservation -member_name YYYY-MM-DD
hh:mm n.n [essentials];
addReservation -member_B 2025-05-14 08:00
3.0 locker umbrella;
```

where,
- *member_[A/B/C/D/E]*: member name;
- *YYYY-MM-DD hh:mm*: booking time;
- *n.n*: duration of time usage;
- *[essentials]*: required essentials.
- ***addEvent*:** Book a parking place together with mandatorily required essentials devices. This command has higher priority than ***addReservation***.

  ```
  addEvent -member_name YYYY-MM-DD hh:mm
  n.n [essentials];
  addEvent -member_E 2025-05-16 14:00
  2.0 locker umbrella valetpark;
  ```

  where,
  - *member_[A/B/C/D/E]*: member name;
  - *YYYY-MM-DD hh:mm*: booking time;
  - *n.n*: duration of time usage;
  - *[essentials]*: required essentials.

***bookEssentials*:** Reserve a specific essential only.

```
bookEssentials -member_name YYYY-MM-DD
hh:mm n.n [essentials];
bookEssentials -member_C 2025-05-011 13:00
4.0 battery;
```

where,
- *member_[A/B/C/D/E]*: member name;
- *YYYY-MM-DD hh:mm*: booking time;
- *n.n*: duration of time usage;
- *[essentials]*: required essentials.

***addBatch*:** Process a batch file, which is an external `.dat` file containing a set of commands.

```
addBatch -[filename]
addBatch -batch001.dat
addBatch -./data/batch/batch001.dat
```

where,
- `filename`: the filename of the batch file. Note that a pathname can also be used here. If the file pathname is not used, the program will look for the file in the **working directory**. Therefore, before running this program, we strongly recommend using `cd` to change the working directory to the directory where the program and batch files are located.

***printBookings*:** Print booking information and summary report (if applicable).

```
printBookings -[fcfs/prio/opti/all];
```

- *fcfs*, *prio*, *opti*, *all*: algorithm used (if *all* is used, a Performance Report will be generated).

***endProgram*:** Quit the program.

```
endProgram;
```

E. Error Handling

For all requests, we assume that the input format is correct. Requests with invalid user name, start time, or duration time will be rejected.

If the essential item or pair is invalid, the corresponding request for reservation will be denied. For *addReservation*, the number of essentials will also be checked to guarantee that an essential pair is received.

We have provided the necessary error handling to identify incorrect formats, but the error content of the incorrect formats may deviate to some extent.

Errors in user input will **not** cause the program to exit.

In terms of the program itself, we have added error checks for system calls like `fork()` (e.g., print an error message when the return value is less than `0`).

F. Special Libraries (Built-in Algorithms)

We need to point out that we used some special libraries, that is, built-in algorithms. These algorithms include:

- **Built-in string algorithms.** Thanks to the support of the compiler, using built-in string algorithms can result in "*surprising*" speed improvements.
- **Quicksort algorithm.** The PRIO scheduler requires sorting, and quicksort can sort a sequence of length n in average $O(n \log n)$ time. We use `qsort()` function to perform quicksort, and thanks to function pointers, we can customize the comparison logic [11].

Any other data structures or algorithms are manually implemented.

Reason: we use these special libraries (built-in algorithms) is that the compiler can provide additional optimization and stability guarantees, while also avoiding repetitive work, allowing us to focus on implementing our innovative parts.

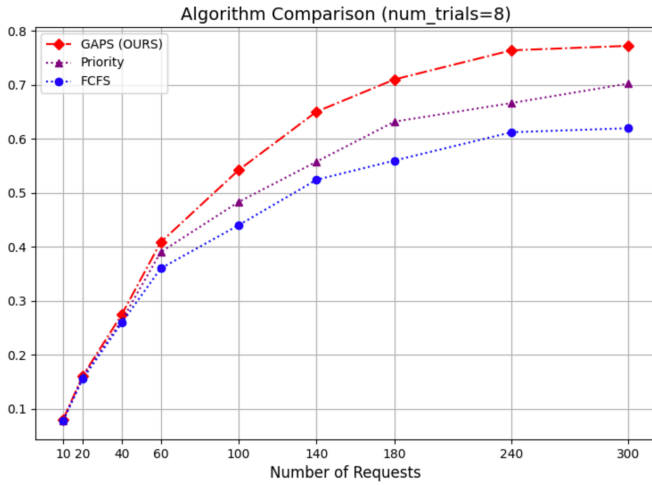## IX. RESULTS / GRAPHS / FIGURES DISCUSSION

Fig. 1. Algorithm Comparison between FCFS, LJF, and GAPS (Uniform Distribution for both Duration and Start Time)

TABLE III. COMPARISON OF THE FCFS, PRIO, LJF, AND GAPS SCHEDULER (UNIFORM DISTRIBUTION)

M	U_{FCFS}	U_{PRIO}	U_{LJF}	U_{GAPS}
10	0.078	0.078	0.080	**0.080**
20	0.155	0.160	0.161	**0.161**
40	0.259	0.263	0.274	**0.274**
60	0.360	0.390	0.405	**0.408**
100	0.440	0.483	0.526	**0.542**
140	0.524	0.558	0.622	**0.650**
180	0.560	0.632	0.688	**0.710**
240	0.612	0.666	0.726	**0.764**
300	0.620	0.702	0.740	**0.772**

This section presents a comprehensive performance analysis of our proposed OPTI scheduling algorithm—GAPS—in comparison with baseline approaches FCFS and PRIO. Utilizing the methodologically rigorous test data described in Section VI.D, we demonstrate the superior performance characteristics of GAPS through quantitative metrics presented in Table III, Table IV, Figure 1, and Figure 2.

Figure 2 and Table IV illustrate algorithm performance when applied to our challenging exponential + multimodal distribution dataset, specifically designed to simulate realistic temporal patterns in parking requests as elaborated in Section VI.D. Complementarily, Figure 1 and Table III present performance metrics on uniformly distributed requests, providing baseline comparison under simplified conditions.

Performance evaluation focuses on *Total Utilization Rate*—a critical metric reflecting the efficiency of resource allocation—across varying request volumes (parameterized by M). To ensure statistical robustness, each configuration undergoes 8 independent trials, with reported values representing ensemble averages.
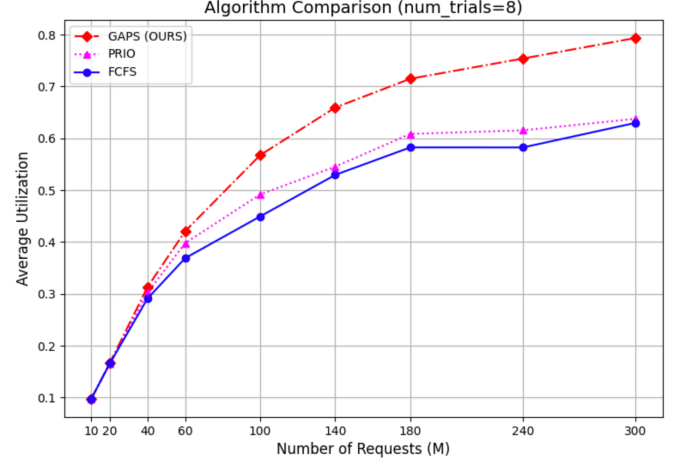

Fig. 2. Algorithm Comparison for Exponential Distribution (for Duration) + Multimodal Normal Distribution (for Start Time)

TABLE IV. COMPARISON OF THE FCFS, PRIO, LJF, AND GAPS SCHEDULER (EXPONENTIAL + MULTIMODAL DISTRIBUTION)

M	U_{FCFS}	U_{PRIO}	U_{LJF}	U_{GAPS}
10	0.097	0.097	0.097	**0.097**
20	0.166	0.165	0.167	**0.167**
40	0.291	0.302	0.308	**0.313**
60	0.369	0.398	0.414	**0.421**
100	0.449	0.492	0.539	**0.568**
140	0.529	0.545	0.622	**0.659**
180	0.583	0.609	0.677	**0.715**
240	0.582	0.615	0.721	**0.754**
300	0.630	0.638	0.748	**0.794**

The results demonstrate the exceptional capability of our OPTI algorithm under diverse testing conditions. Particularly noteworthy is GAPS's performance resilience when confronted with the challenging Exponential + Multimodal distribution (Figure 1), where both FCFS and PRIO exhibit significant performance degradation. In contrast, GAPS maintains consistently high utilization rates across all testing scenarios, empirically validating the algorithm's robust optimization capabilities and adaptive resource allocation mechanism.

X. CONCLUSIONS

We successfully developed and implemented the Smart Parking Management System (SPMS) for PolyU, effectively addressing resource utilization inefficiencies through advanced algorithmic techniques while maintaining high computational performance via specialized data structures and system architecture. The integration of system-

level parallelism through strategic process management enables SPMS to achieve lower latency via concurrent execution of multiple scheduling approaches.

Our GAPS scheduler demonstrated superior performance. By combining a slightly improved Longest-Job-First (LJF) greedy strategy with principles from Simulated Annealing (SA), GAPS consistently achieves significantly higher resource utilization rates compared to FCFS and PRIO.

From a methodological perspective, key innovations include: (1) efficient interval management through lazy-propagated segment trees, dramatically reducing query complexity; (2) dynamic arrays for scalable request handling with optimal memory utilization; and (3) integration of high-performance pseudorandom number generation for stochastic optimization processes. These technical components collectively enable high-throughput request processing while maintaining algorithmic sophistication.

Our experimental evaluation employs statistically rigorous testing under realistic conditions, utilizing multimodal-normal and time-dependent exponential distributions to simulate challenging real-world request patterns. Results conclusively demonstrate GAPS's exceptional scheduling capabilities, consistently achieving approximately 80% average utilization even under challenging distributional assumptions. Future work might include more sophisticated hyper-parameter tuning techniques for *Simulated Annealing* and potential implementation of dynamic decay schedulers for the stochastic parameters p and q, further enhancing GAPS's capability to converge on optimal solutions.

APPENDIX

A. Assumptions

a) *Offline Scheduling:*

An online algorithm operates on data sequentially as it arrives, making decisions on-the-fly; this implies that schedulers are not allowed to change existing schedules. In contrast, an offline algorithm runs with perfect knowledge and have the ability to re-schedule previous arrangements when received new requests.

In our project, we make the critical assumption that SPMS operates under **offline setting**, which means that the algorithms essentially re-arrange the entire schedule from scratch each time `printBooking` is called. No connection between print sessions are maintained, and thus we do not guarantee the consistencies between old schedules and new schedules.

Based on this theoretical understanding, we designed SPMS to execute scheduling algorithms only when necessary—specifically when users request information via commands such as *printBookings*. This approach maximizes efficiency while maintaining responsive system behavior.

b) *Error Handling:*

In our implementation of SPMS, we assume that commands adhere to the basic structural format specified in the project requirements PDF. Under this assumption, we perform comprehensive validation of parameter values, with explicit error handling for invalid inputs.

c) *Optimization Objective—Utilization Rate:*

To quantify scheduling efficiency, we formalize the optimization objective for our OPTI algorithm. Consider a system with N distinct resource types, each having m_i instances ($1 \leq i \leq N$). Given a set of P accepted requests $R = \{r_1, r_2, ..., r_P\}$, where each request r_k is characterized by a time interval $[l_k, r_k]$ and resource type indicator c_k ($1 \leq k \leq P$), we define the utilization rate as follows:

$$U = \frac{\sum_{i=1}^{N} u(i) \cdot m_i}{\sum_{i=1}^{N} m_i}$$

$$u(i) = \frac{\sum_{k=1}^{P} \mathbb{I}(r_k, i) \cdot (r_k - l_k + 1)}{10080 \cdot m_i}$$

$$(32)$$

Where the indicator function $\mathbb{I}(r_k, i)$ is defined as:

$$\mathbb{I}(r_k, i) = \begin{cases} 1 & \text{if } r_k \text{ requested } i\text{-th type resource} \\ 0 & \text{otherwise} \end{cases} \quad (33)$$

This formulation enables precise quantification of resource utilization across heterogeneous resource types, providing a unified optimization objective for the GAPS algorithm. More details in Section III.D.

B. Command

To address potential ambiguities in the project requirements document, we provide the following clarifications regarding command syntax and behavior in SPMS:

- Resource type identifiers include: `battery`, `cable` or `cables`, `locker`, `umbrella`, `InflationService`, and `valetPark`.
- The system accepts `cables` as a valid alternative to `cable`, treating them as functionally equivalent.
- All resource identifiers are case-insensitive. For example, `battery`, `Battery`, and `bAttery` are treated as identical.
- Command termination requires a semicolon (`;`); omission may result in undefined behavior.

These specifications ensure consistent interpretation of user commands while providing reasonable flexibility in input formatting.

C. Sample Output

Below we present a representative example of the *Performance Report* generated by SPMS. This output illustrates the comprehensive performance metrics captured by the system across all three scheduling algorithms. Note that due to the formatting constraints of this document, some alignment characteristics may differ from the actual terminal output:

```
*** Parking Booking Manager - Summary
Report ***

Performance:

 For FCFS:
        Total Number of Booking Received:
1864 (100.00%)
        Total Number of Booking Assigned:
290 (15.56%)
        Total Number of Booking Rejected:
1574 (84.44%)


        Utilization of Time Slot:
```

```
        Parking:          - 26.61%
        Battery:          - 35.71%
        Cable:            - 35.71%
        Locker:           - 37.90%
        Umbrella:         - 37.90%
        Inflation Service: - 38.89%
        Valet Parking:    - 38.89%

        Invalid request(s) made: 137

 For PRIO:
        Total Number of Booking Received:
1864 (100.00%)
        Total Number of Booking Assigned:
299 (16.04%)
        Total Number of Booking Rejected:
1565 (83.96%)

        Utilization of Time Slot:
            Parking:          - 33.33%
            Battery:          - 33.53%
            Cable:            - 33.53%
            Locker:           - 40.08%
            Umbrella:         - 40.08%
            Inflation Service: - 39.68%
            Valet Parking:    - 39.68%

        Invalid request(s) made: 137

 For OPTI:
        Total Number of Booking Received:
1864 (100.00%)
        Total Number of Booking Assigned:
242 (12.98%)
        Total Number of Booking Rejected:
1622 (87.02%)

        Utilization of Time Slot:
            Parking:          - 37.26%
            Battery:          - 40.28%
            Cable:            - 40.28%
            Locker:           - 43.85%
            Umbrella:         - 43.85%
            Inflation Service: - 48.61%
            Valet Parking:    - 48.61%

        Invalid request(s) made: 137
```

D. Xorshift PRNG

To support the stochastic optimization process in GAPS, we implemented a high-efficiency Pseudo-Random Number Generator (PRNG). PRNGs are deterministic algorithms that produce sequences of numbers exhibiting statistical properties of randomness, initialized by a seed value to ensure reproducibility.

Our implementation utilizes *Xorshift128+*, an algorithm renowned for its exceptional combination of performance and statistical quality. This algorithm offers significantly better performance characteristics than standard library random number generators while maintaining excellent statistical properties required for effective stochastic optimization. The core algorithm is implemented as follows:

---

function xorshift128plus-rand()

---

1 // Generate a random number in Xorshift128+

2 $t \leftarrow state0$

3 $state0 \leftarrow state1$

4 $t \leftarrow t \oplus (t \ll 23)$

5 $t \leftarrow t \oplus (t \gg 17)$

6 $t \leftarrow t \oplus state1 \oplus (state1 \gg 26)$

7 $state1 \leftarrow t$

8 **return** $state0 + state1$

E. Fast Power Algorithm

The initialization phase of GAPS requires precise computation of temperature decay rates to transition from initial to final acceptance probabilities over a specified number of iterations. This computation involves raising values to potentially large exponents, which would be computationally expensive using naive approaches.

To optimize this process, we implemented a logarithmic-time exponentiation algorithm that reduces the computational complexity from $O(N)$ to $O(\log N)$. This recursive divide-and-conquer approach is particularly valuable for the parameter tuning process in our simulated annealing implementation.

REFERENCES

[1] G. C. O. Docs, "Linux System Call Table." N/A, Jun. 2021.

[2] N. Roussopoulos, *Efficient data structures for range queries*. in Advances in Database Systems. Orlando, FL, USA: Academic Press, 1985, pp. 373–393.

[3] M. Bell, "Efficient Algorithms and Data Structures for Geometric Range Queries and Interval Management," Berkeley, CA, USA, 2008.

[4] D. Kunal and S. Kumar, "Segment Trees and Lazy Propagation," Delhi, India, 2015.

[5] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[6] E. Conrad, S. Misenar, and J. Feldman, "Chapter 6 - Domain 6: Security Architecture and Design," *Eleventh Hour CISSP (Second Edition)*, pp. 95–116, 2014.

[7] A. S. Tanenbaum, "Amortized Analysis of Dynamic Arrays," Amsterdam, Netherlands, 2006.

[8] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," in *Science*, American Association for the Advancement of Science (AAAS), 1983, pp. 671–680.

[9] P. K. Anderson, "Annealing and self-organizing systems," *Journal of Computational Physics*, vol. 65, pp. 523–538, 1986.

[10] K. Inc., "CMake Tutorial and Reference." Clifton Park, NY, USA, Apr. 2021.

[11] "GNU C Reference Manual – qsort Function." Cambridge, MA, USA, Aug. 2022.