

MINISTRY OF SCIENCE AND HIGHER EDUCATION OF THE RUSSIAN FEDERATION
FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
"NORTH CAUCASUS FEDERAL UNIVERSITY"

SUGGESTIONS FOR TEACHING
to practical classes
in the discipline "Data Structure and Algorithm"
for students of the field of study

11.03.02 Infocommunication Technologies and Communication Systems,
Specialization "Intelligent Data Processing in Infocommunication Systems and
Networks"

Stavropol

2026

CONTENTS

PRACTICAL WORK No 1.....	6
Topic: Determining the complexity function of algorithms	6
PRACTICAL WORK No 2.....	13
Topic: Simulation modeling of the job flow service system	13
PRACTICAL WORK No 3.....	27
Topic: Construction of an environment graph for a pathfinding algorithm.....	27
PRACTICAL WORK No 4.....	37
Subject: Simulation of a mobile robot in a warehouse (using Gazebo).....	37
PRACTICAL WORK No 5.....	47
Topic: Algorithm optimization using hash tables.	47
PRACTICAL WORK No 6.....	54
Implementing a multithreaded algorithm.....	47
PRACTICAL WORK No 7.....	100
Topic: Implementation of the SLAM algorithm	100
PRACTICAL WORK No 8.....	113

Topic: Introduction to YOLO architecture for object detection.	113
PRACTICAL WORK No 9.....	114
Topic: Algorithm optimization using a profiler	114

INTRODUCTION

Methodological instructions for laboratory work are compiled in accordance with the work program of the discipline "Algorithms and Data Structures in Robotics" and are intended for students of the direction of training 09.03.01 Informatics and Computer Engineering.

The purpose of mastering the discipline "Algorithms and Data Structures in Robotics" is to form students' theoretical knowledge and practical skills in the field of algorithms and data structures necessary for effective solving of robotics problems, including motion planning, navigation, sensor data processing and robot control in real time.

1. List of planned learning outcomes in disciplines (module) correlated with the planned results of the development of the educational program

Code, formulation of competence	Indicator code, wording	Planned learning outcomes in the discipline (module), characterizing the stages of the formation of competencies, indicators
PC-2. Capable of conceptual, functional and logical design of systems of medium and large scale and	ID-6PC-2 Applies methods and algorithms for optimizing executable code	Able to design complex algorithmic systems in robotics, from concept to logical implementation, taking into account

complexity	ID-7PC-2 Uses modern software development methodologies and programming technologies	scalability, performance and modularity, including working with ROS, distributed computing and robotic platforms.
	ID-14PK-2 Defines and applies the necessary algorithms and data structures to solve the problems of navigation, planning, sensor data processing, control and decision-making in robotic systems	
	ID-9PC-2 Designs Data Structures	

PRACTICAL WORK No 1.

Topic: "Determination of the complexity function of algorithms"

The purpose of the work is to study the principle of determining the complexity function of an algorithm.

Theoretical information

1. Algorithm complexity function

The effectiveness of a program is a very important characteristic of it. The effectiveness of a program has two components: memory (or space) and time.

Spatial efficiency is measured by the amount of memory required to execute a program. Computers have a limited amount of memory. If two programs implement identical functions, the one that uses less memory is more spatially efficient.

The best way to compare the efficiency of algorithms is to compare their orders of complexity. This method applies to both temporal and spatial complexity. The order of complexity of an algorithm expresses its efficiency, usually in terms of the amount of data being processed. For example, an algorithm may depend significantly on the size of the array being processed. If, say, the processing time doubles with doubling the size of the array, then the order of temporal complexity of the algorithm is defined as the size of the array. The function $f(n)$ has the order $O(g(n))$ if there is a constant K and a counter n_0 such that $f(n) \leq K \cdot g(n)$ for $n > n_0$.

An algorithm has a complexity of $O(f(N))$ if, as the dimensionality of the source data N increases, the execution time of the algorithm increases at the same rate as the function $f(N)$.

The complexity function O expresses the relative speed of an algorithm depending on some variable (or variables). There are three important rules for defining a complexity function:

1) $O(k*f) = O(f)$

2) $O(f*g) = O(f)*O(g)$ or $O(f/g) = O(f)/O(g)$.

3) The $(f+g)$ is equal to the dominant $O(f)$ and $O(g)$.

Here, "k" stands for a constant, and f and g are functions.

The first rule declares that constant factors are irrelevant to determining the order of difficulty

$$O(1,5*N)=O(N).$$

It follows from the second rule that the order of complexity of the product of two functions is equal to the product of their complexities

$$O((17*N)*N)=O(17*N)*O(N)=O(N)*O(N)=O(N*N)=O(N^2).$$

It follows from the third rule that the order of complexity of the sum of functions is determined as the order of the dominant of the first and second terms, i.e., the largest order is chosen

$$O(N^5 + N^2) = O(N^5).$$

2. Types of complexity function of $O(I)$ algorithms

In constant-complexity algorithms, most operations in a program are performed one or more times. Any algorithm that always requires the same amount of time, regardless of the size of the data, has a constant complexity.

1) $O(N)$. The running time of the program is linear, usually when each element of input data needs to be processed only a linear number of times.

2) $O(N^2)$, $O(N^3)$, $O(N^a)$. Polynomial complexity. $O(N^2)$ is quadratic complexity, $O(N^3)$ is cubic complexity

3) $O(\log(N))$. When the runtime of a program is logarithmic, the program starts to run much slower as N increases.

4) $O(N \cdot \log(N))$. This is the time of operation of those algorithms that divide a large problem into small ones, and then, having solved them, combine their solutions.

5) $O(2^n)$. Exponential complexity. Such algorithms most often arise as a result of an approach called the method of brute force.

3. Analysis of the complexity function according to the program

A programmer must be able to analyze algorithms and determine their complexity. The time complexity of an algorithm can be calculated based on the analysis of its control structures. Algorithms without loops and recursive calls have constant complexity. If there is no recursion and loops, all control structures can be reduced to structures of constant complexity. Consequently, the entire algorithm is also characterized by constant complexity. Determining the complexity of an algorithm is mainly reduced to the analysis of loops and recursive calls. For example, let's consider an algorithm for handling array elements (Fig. 11.1a).

The complexity of this algorithm is $O(N)$, since the loop body is executed N times, and the complexity of the loop body is $O(1)$. If one loop is nested inside another and both loops depend on the size of the same variable (Figure 11.1b), then the whole construct is characterized by quadratic complexity. The complexity of such a program is $O(N^2)$.

The most complex parts of a program are usually the execution of loops and procedure calls. If one procedure calls another, the complexity of the latter must be evaluated more carefully. If a certain number of instructions are executed, such as

printing, it has little effect on the order of complexity. On the other hand, if $O(N)$ steps are performed in the called procedure, the function can add much more complexity to the algorithm. If the procedure is called within a loop, the effect can be much greater.

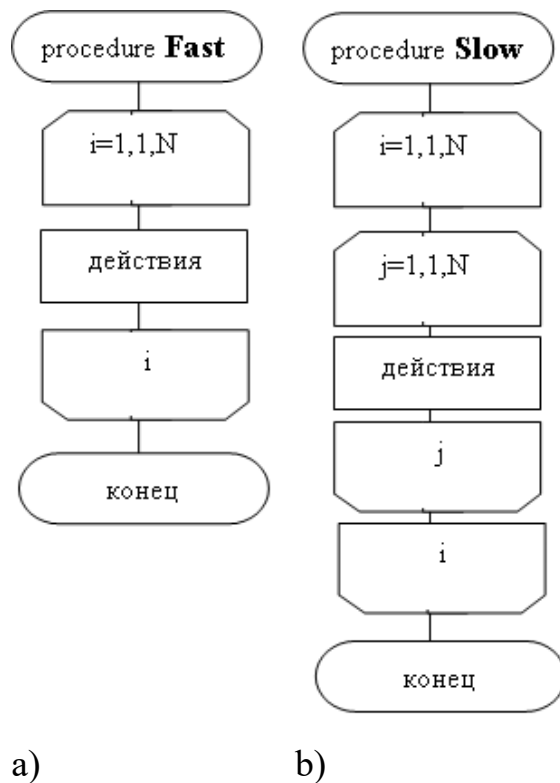


Figure 11.1 – Block diagram of the algorithm for processing a one-dimensional (a) and two-dimensional array (b)

As an example, let's take a program containing a slow Slow procedure with a complexity of order $O(N^2)$ and a fast procedure Fast with a complexity of order $O(N)$ – **Fig . 11.1**. The complexity of the entire program depends on the relationship between these two procedures.

If the Slow procedure is called every time the cycles of the Fast procedure are executed, the complexities of the procedures are multiplied. The total complexity is equal to the product of both complexities. In this case, the complexity of the algorithm is $O(N) * O(N^2)$ or $O(N * N^2) = O(N^3)$.

On the other hand, if the host program calls the procedures separately, their computational complexity is added. In this case, the total complexity is of an order of magnitude equal to $O(N) + O(N^2) = O(N^2)$.

Consider the algorithm of Fig.11.2, which for the matrix $A[N \times N]$ finds the maximum element in each row.

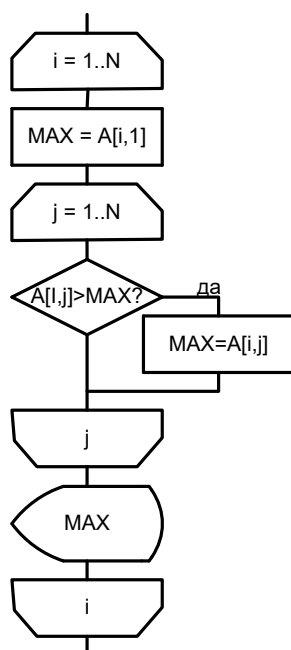


Figure 11.2 – Fragment of the algorithm for finding the maximum element in each row of the array

In this algorithm, the variable i changes from 1 to N . For each change in i , the variable j also changes from 1 to N . During each of the N iterations of the outer loop, the inner loop is also executed N times. The total number of iterations of the inner loop is $N \cdot N$. This determines the complexity of the algorithm $O(N^2)$.

$$O(f) = O(N) * (O(1) + (O(N) * (O(1) + O(1)))) + O(1) = O(N) * O(N) = O(N^2)$$

As a rule, about 90% of the time a program runs requires repetition and only 10% is computational. Analysis of the complexity of programs shows which fragments this 90% falls on — these are the cycles of the greatest depth of nesting. Repetitions can be organized in the form of nested loops or nested recursion.

This information can be used by the programmer to build a more efficient program in the following ways. First of all, you can try to reduce the depth of nesting of repetitions. Then, you should consider reducing the number of statements in the loops with the highest nesting depth. If 90% of the execution time is the execution of internal loops, then a 30% reduction in these -small sections results in a 27% reduction in the execution time of the entire program.

Procedure for performing laboratory work

It is necessary to implement three algorithms for finding a path in the graph. Compare and justify the choice of the most optimal of them for robotic systems.

Control questions

- 1) What determines the complexity function of an algorithm?
- 2) Specify the types of complexity function of the algorithms.
- 3) How is the time function of the algorithm's complexity determined?

- 4) Indicate the methods of analyzing the complexity function according to the algorithm.
- 5) Specify the equation by which the complexity function is determined.

PRACTICAL WORK No 2.

Topic: Simulation modeling of the job flow service system

Objective: practical implementation of the dynamic data structure of the Queue.

A queue is characterized by such an order of value processing, in which new elements are inserted at the end of the queue, and extracted from the beginning. Such data organization is widely found in various applications. As an example of using a queue, the problem of developing a system for simulating a single-processor computer is proposed. The simulation scheme considered within the framework of the laboratory work is one of the simplest models for serving tasks in a computer system and thus provides only an initial acquaintance with the problems of modeling and analyzing the efficiency of the functioning of real computing systems.

Setting a learning task

Basic Concepts and Definitions

Simulation modeling of actually existing objects and phenomena – physical, chemical, biological, social processes, living and non-living systems, engineering structures, constructed objects – is the construction of a mathematical model that describes the phenomenon under study with sufficient accuracy, and the subsequent implementation of the developed model on a computer for conducting computational experiments in order to study the properties of the simulated phenomena. The use of

simulation modeling makes it possible to study the studied objects and phenomena without conducting real (natural) experiments.

A queue is a scheme for storing information, in which each newly received element of information occupies an extreme position (*the end of the queue*). When information is returned from the queue, the element located first in the queue (*the beginning of the queue*) is returned, and the remaining elements move to the beginning; therefore, the element that arrived first is returned first.

Lab Requirements

For a computer system (AC) with a single processor and a single-program sequential mode of execution of incoming tasks, it is necessary to develop a software system to simulate the process of task servicing in the aircraft. When building a model of aircraft operation, the following main points of task servicing should be taken into account:

- generation of a new task;
- Queue the job to wait for the processor to be released.
- Retrieves a job from the queue when the processor is released after serving the next job.

Based on the results of the computational experiments, the simulation system should display information about the conditions of the experiment (intensity of the task flow, size of the task queue, processor performance, number of simulation cycles) and the indicators of the functioning of the computing system obtained as a result of the simulation, including:

- the number of tasks received by the Armed Forces;

- Number of job denials due to queue overflow
- the average number of cycles for completing tasks;
- The number of ticks that the processor is idle due to no jobs to be serviced.

Indicators of the computer system functioning obtained with the help of simulation systems can be used to assess the efficiency of aircraft use; based on the results of the analysis of indicators, recommendations can be made on the expedient modernization of aircraft characteristics (for example, in case of long downtime of the processor and in the absence of refusals to service tasks, it is desirable to increase the intensity of the flow of serviced tasks, etc.).

Conditions and restrictions

Let's make the following basic assumptions:

- When scheduling the order of job service, the ability to set priorities is not taken into account.
- When new jobs appear and when the processor is freed, they are treated as random events.

Solution Method

Data Structures

Let us recall that a *dynamic structure* is a mathematical structure that corresponds to a partially ordered (by inclusion) base set M , the insertion and deletion operations of which are data structures. In this case, inclusion relations are induced by operations of transforming the data structure.

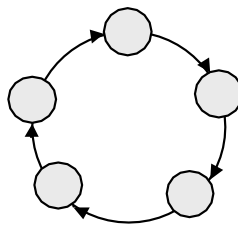
Thus, *the queue* is a dynamic structure, insert and delete operations move the queue from one state to another, while adding new items is done at the end of the queue, and retrieving is done from the beginning of the queue (service discipline

"First come, first served.

An important challenge in implementing a queuing system is to choose a storage structure that solves the problem of efficient memory usage without repacking and without the use of linked lists (which require additional memory for pointers).

As in the case of a stack, *it is proposed to use a one-dimensional (single-index) array located in a dynamic memory area as the queue storage structure. Due to the nature of processing queued values, it is necessary to have two pointers to specify the data stored in the queue - to the beginning and end of the queue. These pointers increase their value: one when inserted, the other when the element is retrieved. Thus, during the operation of the queue, a situation may arise when both pointers reach their maximum value and further replenishment of the queue becomes impossible, despite the availability of free space in the queue. A ring buffer is a storage structure obtained from a vector by extending the sequence relation of the pair $p(an, a\ 1)$.*

Рис. 2. General diagram of the storage structure of the ring buffer type



The storage structure of a queue in the form of a ring buffer can be defined as a one-dimensional (single-index) array located in a dynamic area of memory and the location of the data in which is determined using the following set of parameters:

pMem is a pointer to the memory allocated for the ring buffer,

MemSize – the size of the allocated memory,

MaxMemSize —The default memory size if the required number of memory elements is not explicitly specified when creating the ring buffer,

DataCount – the number of values stored in the queue,

Hi is the index of the array element in which the last queue element is stored,

Li is the index of the array element in which the first queue element is stored.

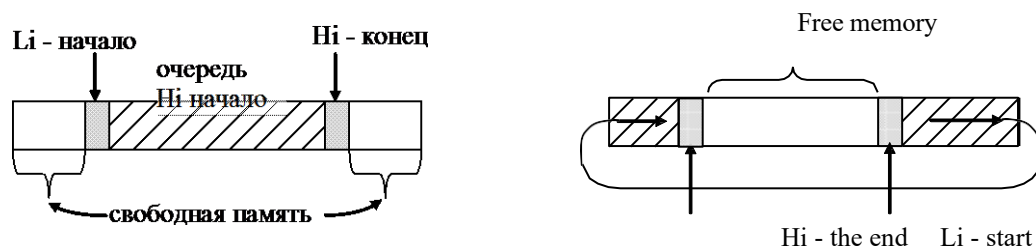


Рис. 3. General Schema: Implementing a Ring Buffer on a Single-Index Array

Because the queue storage structure is largely similar to the stack storage structure, it is proposed that the queue implementation class be built by inheriting from the stack class described in Lab No3. For inheritance, it is sufficient to override the Get and GetNextIndex methods. In the Get method, the index is modified to retrieve the element (values are retrieved from the beginning of the queue), and the GetNextIndex method implements a follow relation on the ring buffer.

Algorithms

Queue

To work with the queue, it is proposed to implement the following operations:

- Put method - add an element;

Adding an item to a queue is similar to adding to a stack, using a method of the stack class.

- Get method - delete the element;

When you remove an item from the queue, you must return a value from the dynamic array on the start index of the queue, move the start pointer, and reduce the number of items.

- The IsEmpty method is to check the queue for emptiness;
- The IsFull method is to check the queue for completeness. These methods are inherited from the stack class.

Task flow modeling

If the value of the sensor is less than a certain threshold value $q1$, $0 \leq q1 \leq 1$, then it is considered that a new task enters the computing system on this simulation cycle (thus, the parameter $q1$ can be interpreted as a value that regulates the intensity of the job flow – a new task is generated on average once per $(1/q1)$ cycles).

Processor Simulation

Simulation of the moment when the processing of the next task is completed can also be performed according to the scheme described above. With the help of a random number sensor, another random value is generated, and if this value is less than the threshold value $q2$, $0 \leq q2 \leq 1$, then it is assumed that on this simulation cycle the processor has completed the service of the next task and is ready to start processing the task from the waiting queue (thus, the parameter $q2$ can be interpreted as a value characterizing the performance of the processor of the computing system - each task is served on average for $(1/q2)$ clock cycles).

Simulation of a computing system

A possible simple scheme for simulating the process of receiving and servicing tasks in a computer system is as follows.

1. Each task in the system is represented by some unambiguous identifier (for example, a task sequence number).
2. To perform calculations, the number of cycles of the system operation is recorded (or indicated in the dialog).
3. On each cycle, the state of the task thread and the processor are polled.
4. Registering a new job in the computing system can be as simple as remembering the job ID in the processor waiting queue. A job queue overflow situation should be understood as a lack of computing system resources to enter a new job (denial of service).
5. To simulate the process of processing tasks, it should be taken into account that the processor may be busy servicing the next task, or it may be in a waiting (idle) state.
6. If the processor is freed, an attempt is made to boot it. To do this, the job is retrieved from the waiting queue.
7. CPU downtime occurs when the queue of pending jobs is empty when the next job is finished.
8. After all the simulation cycles are carried out, the characteristics of the computer system are displayed:
 - the number of tasks received by the computer system during the entire simulation process;
 - Number of job denials due to queue overflow – it is reasonable to consider the failure rate as the ratio of the number of unqueued tasks to the total number of jobs generated, multiplied by 100%;
 - the average number of cycles for completing the task;
 - Number of processor idle ticks due to no job queues for maintenance—It is useful to think of the processor idle clock percentage as the ratio of the number of processor idle ticks to the total number of simulation cycles, multiplied by 100%.

Software development

Structure

S accounting formulated above offers K Implementation The following modular structure of the program seems appropriate:

TStack.h, TStack.cpp – a module with a class that implements operations on the stack;

TQueue.h, TQueue.cpp is a module with a class that implements operations on the queue;

TJobStream.h, TJobStream.cpp – a module with a class that implements a task stream;

TProc.h, TProc.cpp – a module with a class that implements the processor;

QueueTestkit.cpp is a module of the testing program.

Class Specifications

Taking into account the algorithms proposed for implementation, the following class declarations can be made.

TStack *class* (TStack.h file) *TQueue class* (TQueue.h file):

```
class TQueue : public TStack {
protected:
    int Li;                Queue Start Index
    int GetNextIndex (int index); Get the following public index:
    TQueue (int Size = MaxMemSize):TStack(Size) {Li
    = 0;}; TElem Get ( void ) ;    extract value
```

```

#include "Queue.h"

int Queue :: GetNextIndex(int index) { // Get the next index return ++
    index % MemSize ;
}

TElem Queue:: Get ( void ) { // extract value
    Head = GetNextIndex(
    Li ); DataCount--;
    return tmp;
}
}
if ( ~IsEmpty() { // the queue is not empty?
    TElem tmp = pMem[Li];

```

The TElem type describes the value type of the data structure elements (defined in TStack.h). The implementation of the TElem class is described in detail in Lab No3. To define the methods of the TQueue class, a fragment of the TQueue.cpp file (the implementation module) is given below.

The constructor calls the base class constructor, determines the size of the memory area as a constructor parameter (defaults to MaxMemSize) and explicitly takes into account the type of the queue item (TElem), and sets the queue start index to $Li = 0$.

The destructor, IsEmpty, IsFull, and Put methods are inherited from the base class. Before accessing the Put function, you need to check the queue for completeness.

The Get method is overridden for the queue and ensures that the value is retrieved from the queue by the queue start pointer; the operation is performed by removing the return value from the queue. Before accessing this function, you must check the queue for emptiness.

The GetNextIndex method is overridden in the queue class and provides a circular buffer representation of the queue, allowing you to set the values of the start and end of the queue indexes when they reach a boundary value of 0.

TJobStream class (TJobStream.h file)

```
class TJobStream {
private:
    Task flow intensity
public:
    Intens;
    int GetNewJob(void); Generating a new task
};
```

TProc Class (TProc.h file)

```
class TProc {
private:
    CPU performance
    double Id of the job in progress
    ProcRate; int
    int IsProcBusy(void) ; Is the processor busy?
    int RunNewJob (int JobId); start a new mission
};
```

Stages of development

As before, an important part of the lab is the organization of step-by-step program development. The following development sequence is proposed (taking into account the availability of a workable TStack class):

Stage 1. Implementation of queue support programs.

Stage 2. Implementation of the task maintenance simulation system.

Stage 3. Performing additional tasks of laboratory work.

When performing laboratory work, it is necessary to continue the practical development of the technology of iterative development with the most thorough testing of the correctness of the programs. To check the correctness, a sufficient number of tests should be developed, which are guaranteed to check the main possible situations during

the execution of programs. The transition to the next stage of development should be made only after the successful completion of tests; when checking the developed programs of the next stage, tests of all previous stages should be involved.

Given the readiness of the TStack class, the following implementation sequence can be suggested when developing queue programs:

1. Implementation of the TQueue constructor, overloading the GetNextIndex follow method. Testing of developed programs.

2. Implementation of the dequeue method. Testing.

For stage 2, the following development sequence can be used:

1. Development of the TJobStream class specification, implementation of the constructor and method for generating a new job.

2. Development of the TProc class specification, implementation of the constructor.

3. Implement a method for polling the state of the processor and a method for starting a new job for execution. Testing.

4. Development of a control program for the simulation system, which ensures the cyclic execution of simulation cycles: generation of a new task, checking the state of the processor, starting a new task. Testing.

5. Development of a program for displaying the results of the simulation.

6. Perform computational experiments. Prepare a report on the completion of laboratory work.

The input parameters of the computational experiment are: the intensity of the task thread, the performance of the processor and the number of simulation cycles. The maximum queue size is determined by a constant value.

Development considerations

Provide for two modes of the program operation: with a small number of simulation cycles, you should print out the state of the computer system and print statistics, for long periods of simulation, only statistics should be printed.

The development of laboratory programs should be carried out with a mandatory check for possible error situations (for example, retrieving a value from an empty queue). If errors are detected, the program should be terminated with the issuance of a termination alarm code or, more correctly, exception handling can be applied.

Possible topics for additional tasks

For the development of practical programming skills, the following directions for expanding the problem statement are recommended:

1. Implement the ability to set priorities for incoming tasks – in this case, the tasks in the queue should be ordered according to their priorities.
2. To supplement the organization of laboratory work with the possibility of modeling for incoming tasks with different durations of execution.
3. Extend the developed system to simulate multiprocessor computing systems that may have multiple processors. A possible extension of the lab design may be to use multiple processors to perform individual tasks. An important consideration in analyzing the behavior of such systems is to choose a strategy for allocating the available processors for tasks (for example, there may be situations of prolonged postponement of tasks that require a large number of processors).

4. Expand developed system Simulations opportunity Use multiple queues of input jobs.
When expanding the developed programs, you should pay attention to the labor intensity of the necessary modification of the existing program code: the number of methods in which changes are required; the number of new development errors introduced, etc.

Criteria for evaluating the performance of laboratory work

PRACTICAL work is considered to be completed when the minimum set of requirements is met – all classes are implemented, with small values of the number of simulation cycles, the final statistics correspond to the results of the performance of the exercise.

Test Suite to Check:

1. Number of Imitation Bars 10

Queue size 3 Task flow rate 0.5

Processor Performance 0.5

2. Number of Imitation Bars 10

Queue size 3 Task flow rate 0.5

Processor Performance 0.2

3. Number of Imitation Bars 10

Queue size 3 Task flow rate 0.2

Processor Performance 0.5

In the first test, the system is balanced, in the second test, there is a low-intensity flow of tasks with a high-performance processor (the computational experiment should result in a large percentage of processor downtime), in the third test, there is a heavy flow of tasks with a low-performance processor (the computational experiment should show a large number of tasks that are denied service due to an overflowing task queue).

The construction of such estimates of the results of computational experiments and recommendations for changing the computing system can be included in the simulation system being developed.

Questions and tasks for self-control

- What changes might be required to the storage structure when implementing a prioritized queue?
- What additional concepts can be proposed in the development of a simulation system for a more adequate description of the functioning of computing systems?
- How accurately does the used method of job generation describe the actual job streams in computing systems? What additional schemes for generating the input job stream can be proposed?

- Can the developed simulation system be used in other application areas?
- What are the rules for working with a queue implemented in the STL library?
- What are some examples of tasks that might require the use of a queue?

PRACTICAL WORK No 3.

Topic: Construction of an environment graph for a pathfinding algorithm.

The purpose of the work is to implement algorithms for processing graph structures: searching for various paths, checking connectivity, building spanning trees of minimum cost.

Brief theoretical information

A graph is a finite set of vertices and edges connecting them, i.e.:

$$G = \langle V, E \rangle,$$

where V is a finite non-empty set of vertices; E is a set of edges (pairs of vertices).

If pairs E (edges) have a direction, then the graph is called directed (orgraph), otherwise - undirected (neograph). If pairs E include only different vertices, then there are no loops in the graph. If an edge of a graph has a weight, then the graph is called weighted. The degree of a graph vertex is equal to the number of edges entering and exiting it (incidental to it). A neograph is called connected if there is a path from each vertex to any other.

Let us denote the number of vertices as $n = |V|$ and the number of edges as $m = |E|$.

Graphs in memory can be represented in different ways. One type of graph representation is the adjacency matrix $B(n \times n)$; In this matrix, the element $b[i,j]=1$ if the edge connecting the vertices V_i and V_j exists and $b[i,j]=0$ if there is no edge. In undirected graphs, the adjacency matrix is always symmetrical.

In many cases, it is more convenient to represent a graph as a so-called list of adjacency. A list of adjacency contains, for each vertex from a set of vertices, V , a list of those vertices that are directly related to that vertex. Each element ($ZAP[u]$) of the adjacency list is a record containing that vertex and a pointer to the next entry in the list (for the last entry in the list, this pointer is empty). The entries to the adjacency lists for each vertex of the graph are stored in a table (array) ($BEG[u]$)

An example of an undirected graph is shown in Fig. 1, the adjacency matrix for this graph is shown in Fig. 2, and the adjacency list is shown in Fig. 3.

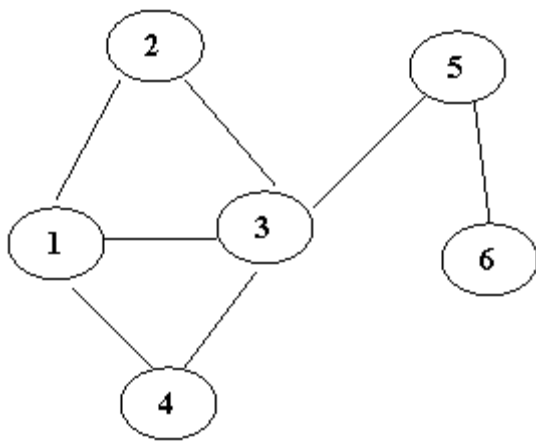


Fig. 1

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	0	0	0
3	1	1	0	1	1	0

4	1	0	1	0	0	0
5	0	0	1	0	0	1
6	0	0	0	0	1	0

Fig. 2

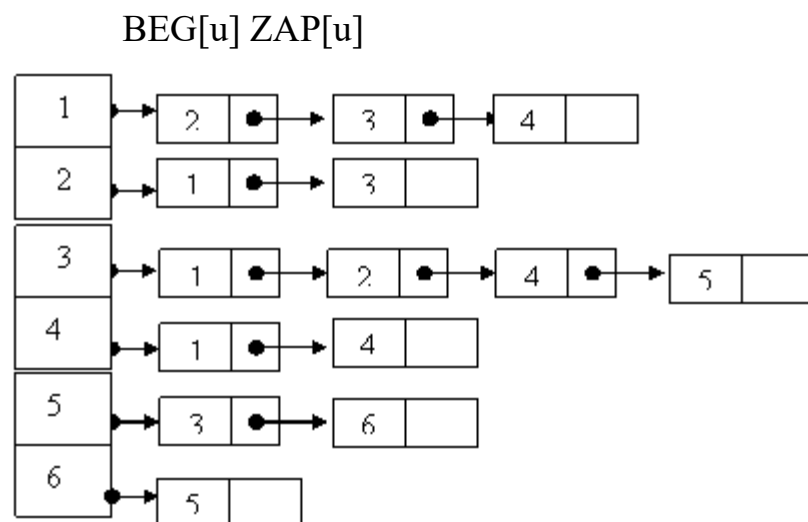


Fig. 3

Let's list the main operations for working with graph structures:

- search for the shortest path from one vertex to another (if there is one);

- search for the shortest path from one peak to all others;
- search for the shortest paths between all vertices;
- search for the Euler path (if there is one);
- search for the Hamiltonian path (if there is one).

Most graph algorithms are based on a systematic search of graph vertices, in which each vertex is viewed exactly once, and the number of graph edge scans is limited by a given constant (preferably no more than once).

One of the main methods of graph algorithm design is depth first search (DFS), in which starting from an arbitrary vertex v_0 , the nearest adjacent vertex v is searched, which in turn is searched in depth (i.e., the nearest adjacent vertex is searched again) until the previously viewed vertex is encountered. or the adjacency list of vertex v does not end (that is, the vertex is fully processed). If there are no new vertices adjacent to v , then vertex v is considered used, returns to the vertex from which **the vertex v** was hit, and the process continues until we get $v = v_0$. In other words, a depth search from vertex v Based on a depth search of all new vertices adjacent to vertex V .

The path obtained by the depth search method is generally not the shortest path from the vertex v to the vertex u . This is a common disadvantage of the depth search [1,2]. Fig. 4 shows the path obtained by traversing the depth graph.

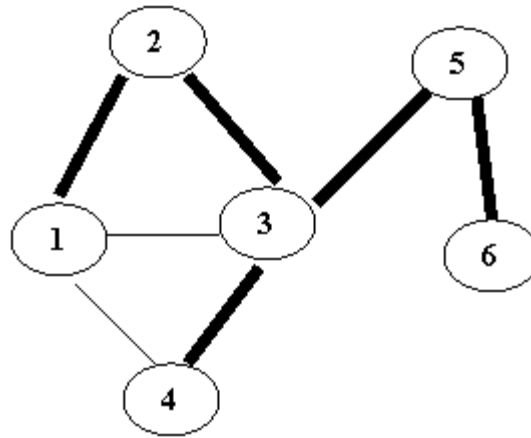


Fig. 4

This drawback is devoid of another method of traversing the graph – breadth first search (BFS). Vertex v is processed by looking at all the new neighbors of this vertex at once.

Fig. 5 shows the path found by the wide-first search method

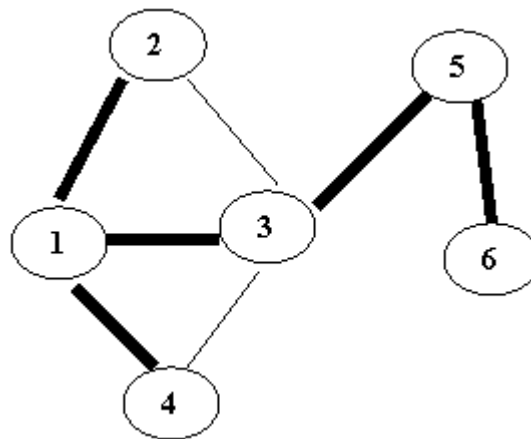


Fig. 5

When using the DFS and BFS algorithms, graphs are traversed in different ways, resulting in some subgraphs that have specific names: frames, skeletons, or pulling trees

[1]. All vertices of the original graph are included in the resulting pulling tree, and all branches of the tree are a set of edges from all sets of edges of the graph (see Figs. 4, 5).

An arbitrary path in a graph that passes through each edge of the graph exactly once is called an Euler path. At the same time, if a path passes over some vertices repeatedly, then it is not simple. If the path is closed, then we have an Euler cycle. For the existence of an Euler path in a connected graph, it is necessary and sufficient that the graph contains no more than two vertices of odd degree [2,3]

A path in a graph that passes exactly once through each vertex of the graph (and not every edge) and the corresponding cycle are called Hamiltonian paths and do not exist for every graph, just as they do not exist for an Euler path. In contrast to Euler paths, there is not a single simple necessary and sufficient condition for the existence of Hamiltonian paths. Even an algorithm of polynomial complexity that verifies the existence of a Hamiltonian path in an arbitrary graph is unknown. The problem of the existence of a Hamiltonian path belongs to the class of so-called NP-complete problems [2, 3].

Finding the shortest paths to all vertices from a single specified vertex for a weighted orgraph (having a value, i.e., weight or cost, edges) with non-negative edges is done using Dijkstra's algorithm. Dijkstra's algorithm is based on choosing to include in the path each time the vertex that has the lowest shortest path estimate (by edge weights), that is, the smallest path to that vertex of all the possible paths discussed earlier [1].

The Bellman-Ford algorithm allows you to solve the problem of finding the shortest paths from one selected vertex to all other vertices at any edge weights, including negative ones. First, the path from the selected vertex to all vertices connected to it is searched (similar to the widest search), and then the shortest path to all other vertices, with an attempt to go to them sequentially, i.e. first through the first vertex, then through

the second, through the third, and so on to the last vertex. In addition, the algorithm returns TRUE if there is no negative weight cycle in the graph achievable from this vertex [2].

To find the shortest paths between all vertices, the Floyd-Warshall algorithm is used. According to the Floyd-Warshall algorithm, the shortest path from one vertex to all vertices accessible from it is first searched, then the same actions are performed, but trying to go from this vertex to all available from it, passing through a new vertex each time (first through the first, then through the second, etc.). All vertices are processed in this way. [2].

As already mentioned, the backbone tree of a graph is a tree containing all the vertices of the graph. The cost of this tree is the sum of the values of all edges [1].

Two algorithms are most often used to construct mainstay trees of minimum value: the Kruskal algorithm and the Prim algorithm.

Kruskal's algorithm, which when viewing a graph in an edge forest consisting of several connected components (trees), adds a minimal edge each time, the ends of which lie in different components of the forest.

Prim's algorithm, when viewing the graph, builds one tree, to which an edge of the minimum weight is added each time [1].

For example, a typical application of least-cost spanning trees is the construction of communication lines between cities, where the cost of edges is the cost of communication networks.

Task

Process the graph structure in accordance with the given option. Justify the choice of the necessary algorithm and the choice of the structure for graph representation. Data input is at the discretion of the programmer. The result is given in graphical form.

Sample versions of the tasks are given in the appendix.

Instructions for the work

When debugging the program, it is necessary to check the correctness of the data entry. The program must adequately respond to incorrect input, including "empty" input. It is necessary to check the search for non-existent paths in the graph, use various graph options to check the correct operation of the program.

When developing the program interface, it is necessary to provide:

- indication of the type, format and range of data to be entered;
- indication of the actions performed by the program;
- the presence of explanations when displaying the result;
- graph output should be carried out graphically (or offer another visualization in the form of a graph);
- Displays the program execution time and the amount of memory required when using different structures to represent the graph.

When testing the program, it is necessary to:

- o check the correctness of data input and output (i.e. their compliance with the required type and format), ensure an adequate response of the program to incorrect data entry;
- o Ensure that messages are displayed in the absence of input data ("empty input");
- o check the correctness of operations;
- o Provide message output when searching for non-existent paths in the graph.

Contents of the report

The report on the laboratory work should substantiate the choice of the form of representation of the graph and the algorithm used to solve the problem. The complexity of the algorithm used should be indicated. It is also necessary to offer a variant of a real problem for which the developed program can be used.

After completing the work, the student should be able to answer the following questions:

1. What is a graph?
2. How are graphs represented in memory?
3. What operations are possible on graphs?
4. What are the ways to traverse graphs?
5. Where are neck structures used?
6. What paths in the column do you know?
7. What are graph wireframes?

The report is submitted in electronic or printed form.

Sample variants of tasks:

1 Find all the vertices of the graph that can be reached from a given vertex by a path no longer than A.

2 The graph is not as a tree. Check if it can be turned into a tree by removing a single vertex along with its edges.

3 A two-way road system has been specified, where for any pair of cities there is a path connecting them. Find a city with the minimum sum of distances to other cities. The path between two cities in two directions can be different.

4 A system of two-way roads is specified. Find many cities whose distance to the selected city (capital) is greater than T . The path between two cities in two directions can be different.

5 Find the minimum (by number of edges) subset of edges whose removal turns a given connected graph into a disconnected graph

PRACTICAL WORK No 4.

Subject: Simulation of a mobile robot in a warehouse (using Gazebo)

Goal: to get acquainted with RTS and the Gazebo simulation system.

MATLAB is required to perform the work, examples are available at the execution of the command:

```
openExample('robotics/SimulateAMobileRobotInAWarehouseUsingGazeboExample')
```

Progress of laboratory work

This example shows how to simulate a warehouse robot in Gazebo. Gazebo allowed you to simulate a mobile robot that uses a value area sensor when performing certain tasks in a simulated environment. This example details how to use the simulation tool to apply Complete Tasks to the Robot's Warehouse Example, where the robot delivers packages in a warehouse scenario. The robot uses the simulated value area sensor in Gazebo to detect and avoid possible collisions with the dynamic environment.

Prerequisites

- Download [the ROS and Gazebo Virtual Machine](#) to set up the simulated robot.
- Consider [Perform Tasks for the Warehouse](#) Robot Example for the Path Planning Workflow and transitioned to the warehouse scenario.
- Consider [Control Robot with Differential Drive in Gazebo with Example Simulink](#) for the basic steps of collecting sensor data and controlling the robot in Gazebo.

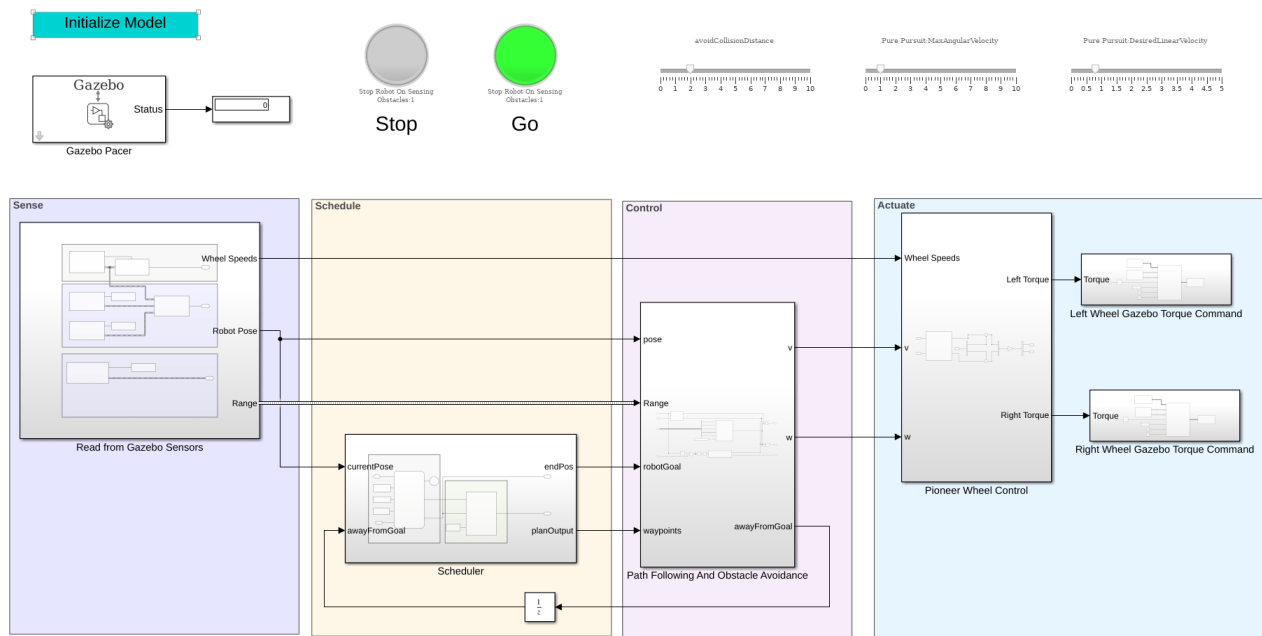
Model Overview

Open the model.

```
open_system('simulateWarehouseRobotInGazebo.slx')
```

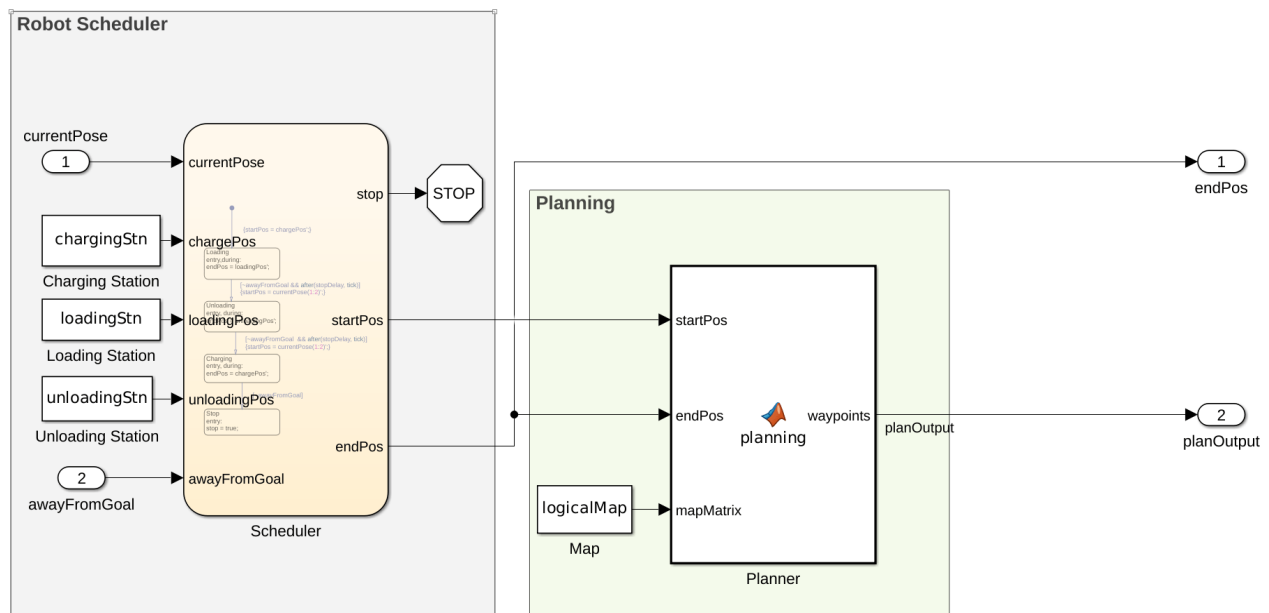
The model can be divided into the following elements:

- Meaning: Read data from sensors in Gazebo.
- Schedule: Assign packages and plan paths for robots to deliver packages.
- Management: Generate commands to follow a predefined path and avoid obstacles.
- Set in motion: Send commands to Gazebo to set the robot in motion in the environment.



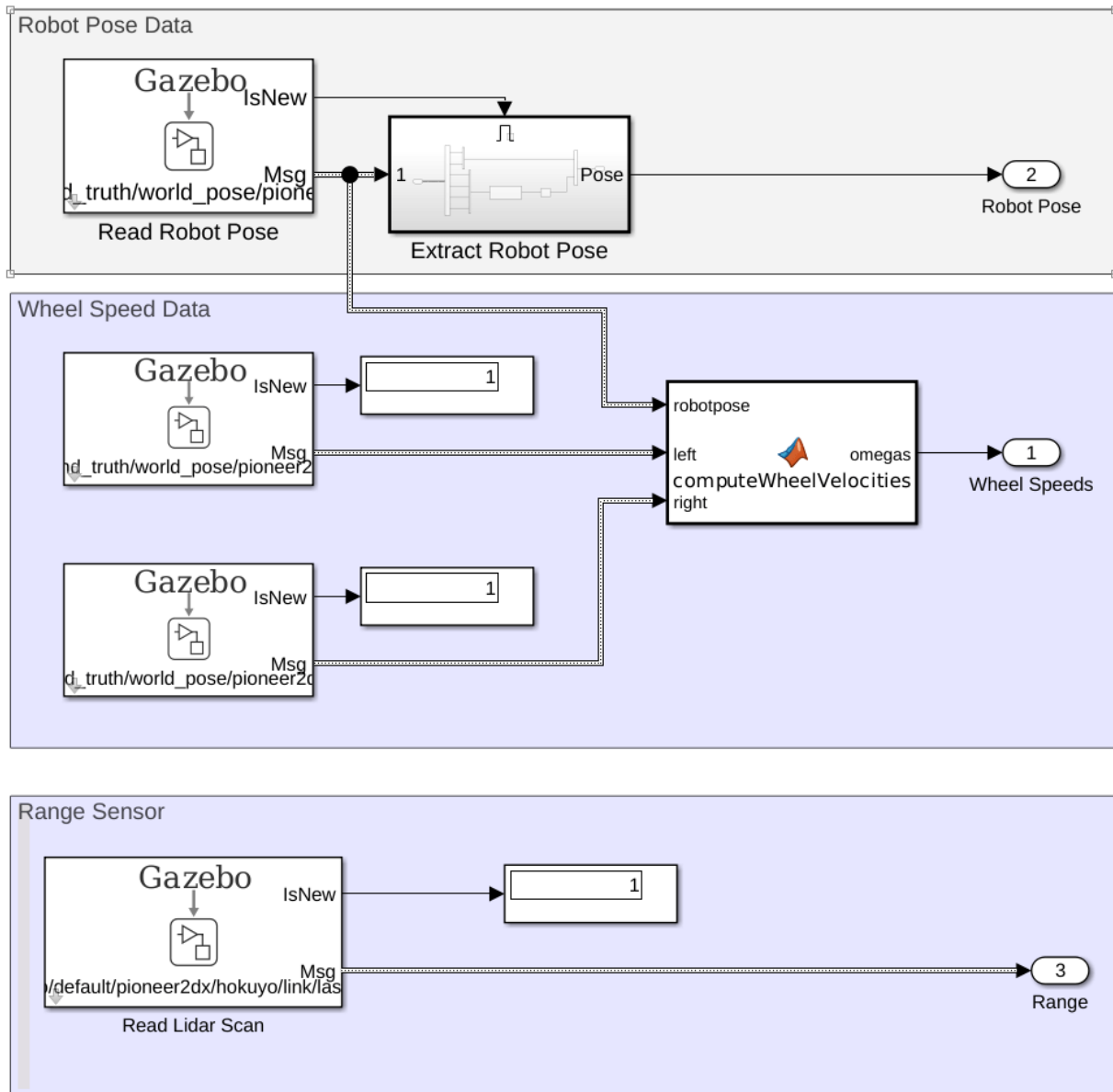
Schedule

The robot performs the task of moving between the charging station, the loading station and the unloading station, as carried out by the Planner.



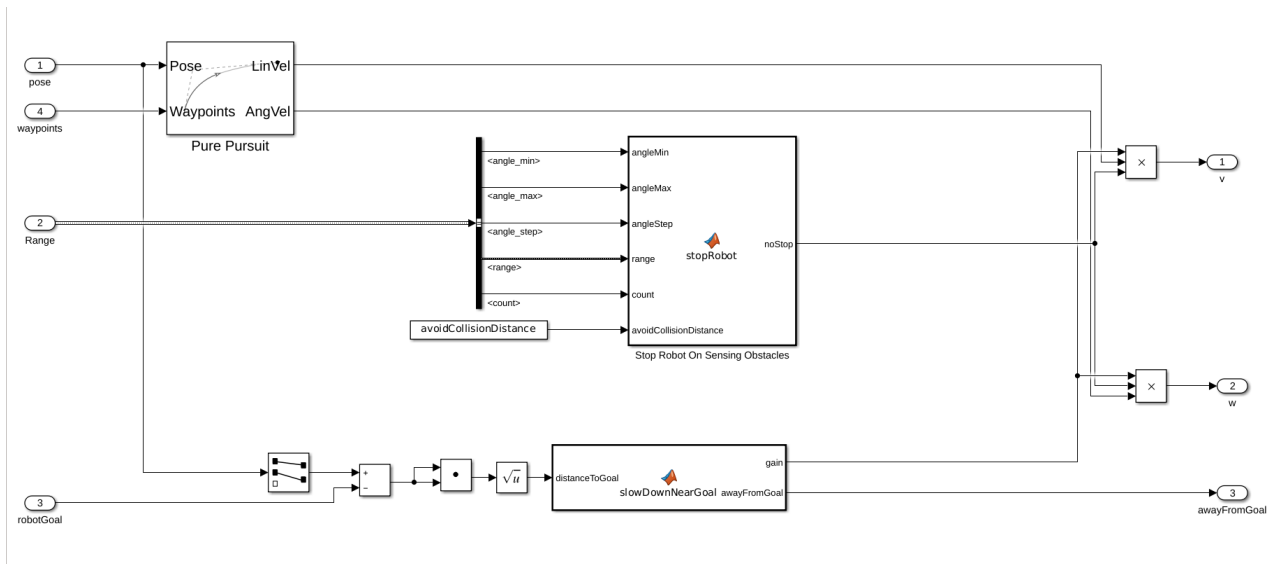
Meaning

The robot's current position, wheel speeds, and sensor readings are read from the simulated environment in Gazebo. The figure below is an extended view of the Read from the Gazebo Sensor Subsystem.



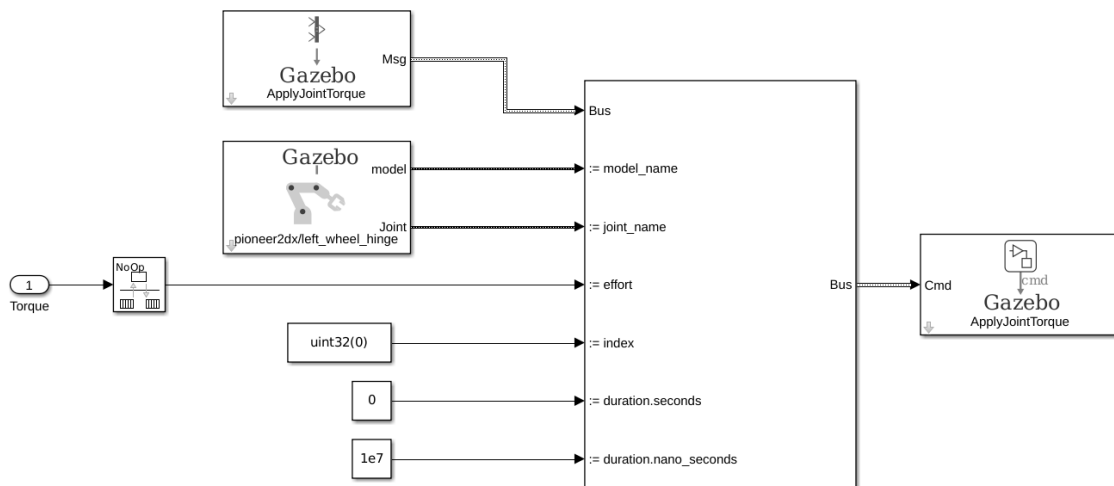
Controls

The controller generates control commands for the next waypoints using the Pure Pursuit block. If the value area sensor on the robot detects an obstacle in the avoidCollisionDistance threshold, the robot stops. In addition, the robot stops when it gets close enough to the target.



Set in motion

Based on the generated control commands, the Pioneer Wheel Control subsystem generates a torque value for each wheel. The torque is applied as an ApplyJointTorque command.



Setup

Warehouse

Load the map file as an example, map, which is a matrix of logical values that indicate occupied space in the warehouse. Invert this matrix to indicate the available space and make the binaryOccupancyMap object. Set the resolution to 100 cells per meter.

The map is based away warehouseExtensions.world a file that was made with the Create Editor on the same scale factor as mentioned below. .png file for the map can be made using collision_map_creator_plugin plugin to generate a map matrix. Details on how to install the plugin can be found in the Collision Map Creator Plugin.

```
mapScalingFactor = 100;  
load gazeboWarehouseMap.mat map  
logicalMap = ~map;  
map = binaryOccupancyMap(logicalMap,mapScalingFactor);  
show(map)
```

Assign *xy locations* to the charging station by sorting the station and the unloading location near the shelves in the warehouse. The selected values are based on the simulated world in Gazebo.

```
chargingStn = [12,5];  
loadingStn = [24,5];  
unloadingStn = [15,24];
```

Show different locations on the map.

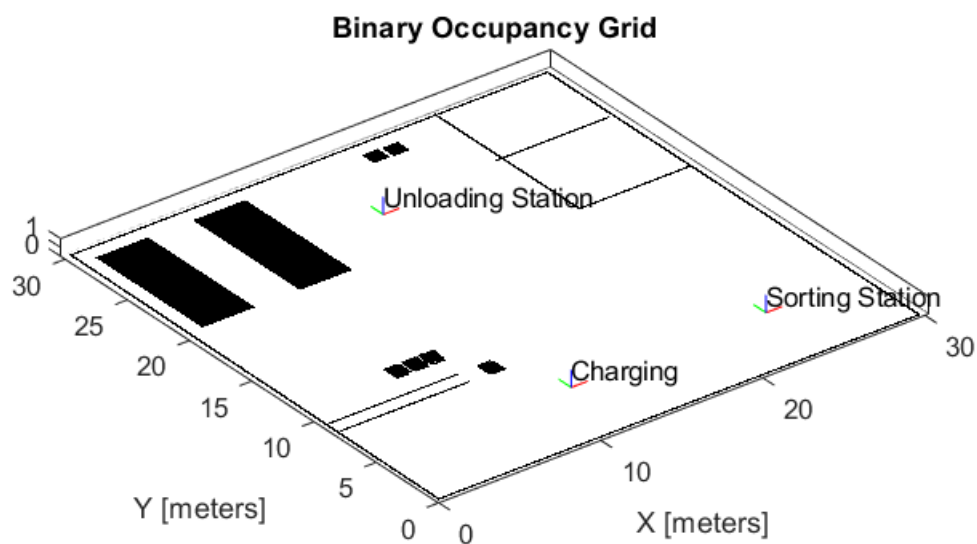
hold on;

```
text(chargingStn(1), chargingStn(2), 1, 'Charging');  
plotTransforms([chargingStn, 0], [1 0 0 0])
```

```
text(loadingStn(1), loadingStn(2), 1, 'Sorting Station');  
plotTransforms([loadingStn, 0], [1 0 0 0])
```

```
text(unloadingStn(1), unloadingStn(2), 1, 'Unloading Station');  
plotTransforms([unloadingStn, 0], [1 0 0 0])
```

```
hold off;
```



Value Area Sensor

The Read Lidar Scan block in the Sensing section is used to read range values from a simulated value area sensor. warehouseExtensions.world file contains the details of the

various models and agents (warehouse workers) in the scene. Because <actor> tags are static links with visual meshes only, the sensor type of the area sensor is `gpu_ray`.

In addition, the Value Area Sensor uses 640 Value Areas, but the default value is 128. This requires modifying the tires used in the Read Lidar Scan block. Download the `exampleHelperWarehouseRobotWithGazeboBuses.mat` file to get the modified tire with `Gazebo_SL_Bus_gazebo_msgs_LaserScan.range` set to 640. The modified tires have been saved to a .mat file using the Tire Editor.

```
load exampleHelperWarehouseRobotWithGazeboBuses.mat
```

Collision avoidance

Agents in the world bypass a predefined trajectory. The robot uses a value area sensor to check for obstacles within a range of 2.0 m (`avoidCollisionDistance`) with range angles from $[-\pi/10, \pi/10]$. After non-zero readings in that value and view area, the robot stops and only resumes after the value area is clear. The "Robot Stop on Recognized Obstacles" function block includes this logic.

When performing a simulation, the Stop lamp turns green when the robot senses that it is good to continue. If it stops, the lamp turns red.

```
avoidCollisionDistance = 2;
```

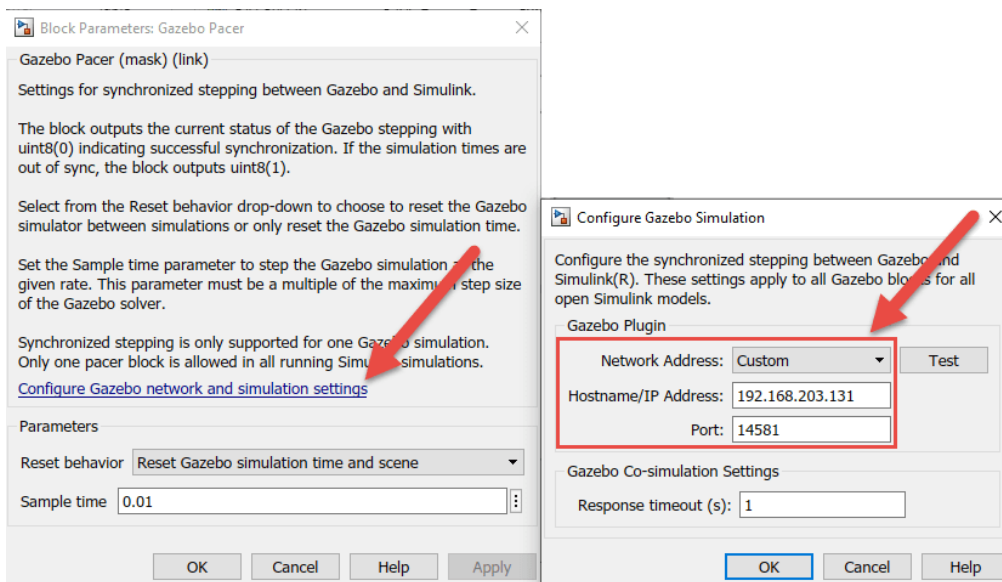
Simulate

To simulate a scenario, set up a connection with Gazebo.

First, launch the Gazebo Simulation Tool. In the virtual machine, click on the Gazebo Robot Warehouse Icon. If the Gazebo Simulation Tool fails to open, you may need to reinstall the plugin. See [Installation, Gazebo Plugin Manually in Performs Co-Simulation Between Simulink and Gazebo](#).

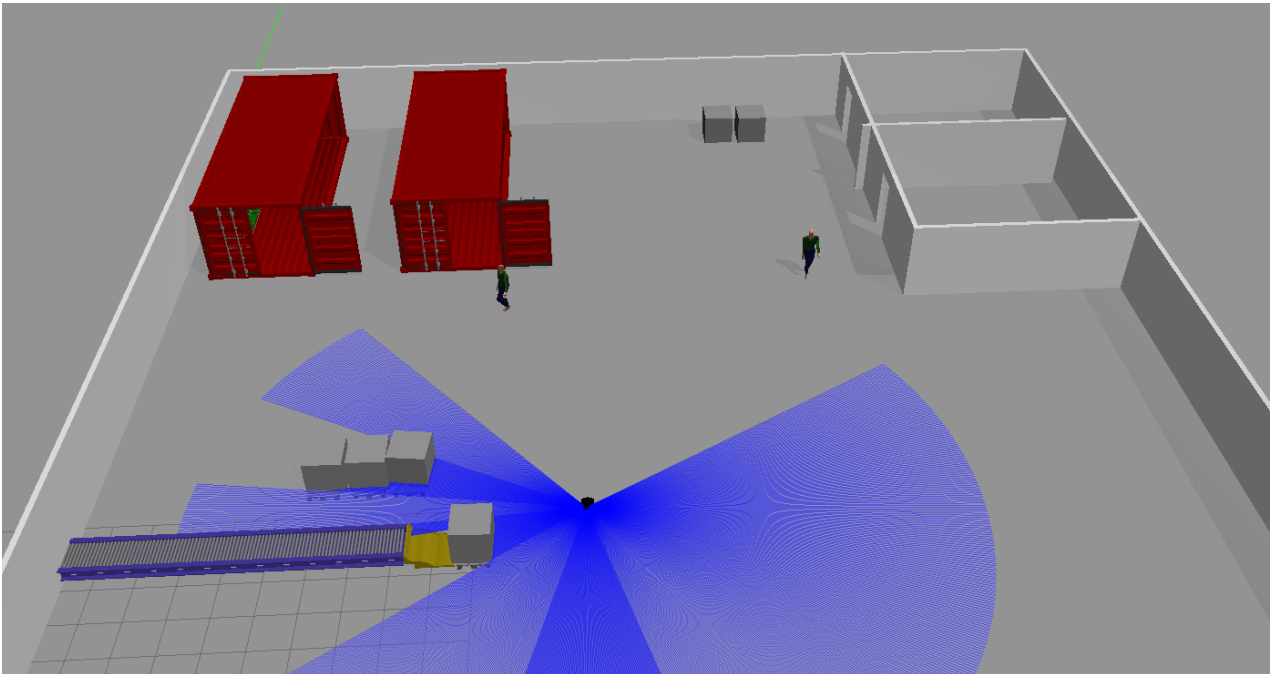
In Simulink, the open Gazebo Pacer block and the Configure Gazebo network click buttons and configure the simulation. Set the Network Address as Custom, the HOSTNAME/IP ADDRESS for your Gazebo simulation, and Port 14581, which is the default port for Gazebo. The VM desktop displays the IP address.

For more information about connecting to Gazebo to enable co-simulation, see, [Perform Co-Simulation Between Simulink and Gazebo](#).



Click the Initialize Model button at the top of the model to initialize all the variables declared above.

Run the simulation. The robot manages around the environment and stops every time the worker gets within the preset threshold.



For more setup details, see <https://docs.exponenta.ru/robotics/ug/simulate-a-mobile-robot-using-gazebo.html>

Contents of the report:

An example of running the presented robot simulation model in Gazebo.

PRACTICAL WORK No 5.

Topic: Algorithm optimization using hash tables.

Objective: to study the principles of construction and operation of hash tables.

Brief theoretical information

Let's consider the algorithm for constructing a binary tree. If we add nodes to the tree and place them evenly on the left and right, we will get a tree with no more than one difference in the number of vertices in the left and right subtrees.

Recursion is used to construct a perfectly balanced tree. For a tree of n nodes, where nl is the number of nodes in the left subtree, nr is the number of nodes in the right subtree, the algorithm for constructing a perfectly balanced tree is described as follows:

1. Choose one vertex as the root;
2. recursively build the left subtree with $nl = n \div 2$ nodes;
3. Recursively build the right subtree with $nr = n - nl - 1$ nodes.

Perfect balancing gives the lowest height of the tree, and since the height of the tree determines the length of the search path in it, it therefore shortens the search. But keeping the tree perfectly balanced when including or excluding an element is a rather difficult procedure.

Adelson-Velsky and Landis formulated a less rigid balance criterion as follows: a binary tree is said to be balanced if the height of the two subtrees differs by no more than one at each node of the tree. Such a tree is called *an AVL tree*.

Using this criterion results in easy balancing. At the same time, the average search length remains almost the same as that of a perfectly balanced tree.

When you include a node in a B-tree, there are 3 possible cases: (we are considering inclusion in the left subtree):

- 1) the left and right subtrees become unequal in height, but the balance criterion is not violated;
- 2) the left and right subtrees acquire equal height and, thus, the balance even improves;
- 3) the balance criterion is violated, and the tree must be rebuilt.

The algorithm for including and balancing depends significantly on the way the tree balance information is stored. One solution is to store a balance score at each vertex. In this case, the balance will be defined as the difference between the height of the right and left subtrees. For example, $Bal = 1$ if the right subtree is higher than the left subtree, $Bal = 0$ if the right and left subtrees are of equal height, and $Bal = -1$ if the left subtree is higher than the right subtree.

The process of enabling a node actually consists of three consecutive subtasks:

- 1) passing through the search path (until we make sure that there is no element with such a key in the tree);
- 2) turning on a new node and determining the balance indicator (Bal);
- 3) "retreat" (return) along the search path, with checking the balance indicators for each vertex, and, if necessary, balancing.

This method requires some redundant validation, but it can be implemented by simply extending the search algorithm with inclusion. At each step, information about the height of the tree is required.

A balancing operation consists only of successive reassignments of references. In fact, the references are cycled, resulting in two or three nodes involved in the balancing process rotating once or twice. In addition to rotation, the balance scores of these nodes must also be adjusted appropriately. [1]

The algorithm for removing a node from the B-tree is based on the algorithm for removing from the tree (replacing the node to be removed with the leftmost child from the right subtree or with the rightmost child from the left subtree), taking into account the balancing operation, that is, the same rotations of nodes. [1]

The use of trees to search for information is quite efficient (labor intensity – $O(\log_2 n)$). Is it possible to create an even more efficient structure or method that allows better information search? To do this, it would be good to immediately determine the index of the element of the array in which the information is stored by the value of the key. That is, it is necessary to create a function by which this index can be calculated. Such a function is called *a hash function* (from the English to hash - to crumble, chop) and it assigns to each key k_i the index of cell j , where the element with this key is located, in the following way:

$$h(k_i) = j, \text{ if } j \in (1, m),$$

where j belongs to the set from 1 to m , and m is the dimension of the array.

An array filled in the order defined by the hash function is called *a hash table*. The minimum effort to search a hash table is $O(1)$!

It is generally accepted that a good function is one that satisfies the following conditions:

- the function must be computationally simple;
- The function should distribute the keys in the hash table as evenly as possible.

So, the first property of a good hash function depends on the characteristics of the computer, and the second property depends on the data values. If all data were random, then hash functions would be very simple (a few bits of a key, for example). In practice, however, random data is extremely rare, and you have to create a function that depends on the entire key.

A situation may arise when different keys correspond to the same hash value, that is, when $h(K_1) = h(K_2)$, while $K_1 \neq K_2$. This situation is called *a collision*. In this case, obviously, it is necessary to find a new place to store keys claiming the same cell of the hash table.

There are two main types of hashing, one based on division and the other based on multiplication. However, these are not the only methods that exist, moreover, they are not always optimal.

If there are fewer keys than array elements, then the calculation of the remainder of the integer key divided by the dimension of the array (m) can be taken as a hash function, that is:

$$H(k) = (\text{turn of } M),$$

with $n < m$, where n is the number of keys.

This function is very simple, although not the best. In general, you can use any dimension (constant m), but it should be such as to minimize the number of collisions. For this, it is better to use a prime number. In most cases, such a choice is quite satisfactory. For a character string, the key may be the remainder of dividing, for example, the sum of the string's character codes by m .

In practice, the method of division is the most common

There are several possible options for resolving conflicts, which have their advantages and disadvantages.

The first method is external (open) hashing (chain method)

In the case when the index table element returned by the hash function is already in use, a linked list is attached to it. Thus, if the same hash value is returned for several different key values, then a pointer to a linked list that contains all the values is located at this address. Searching in this list is a simple search, since with the right choice of hash function, any of the lists turns out to be quite short.

Another way to solve the problem associated with collisions is internal (closed) hashing (open addressing). It consists in completely abandoning references. In this case, if the cell with the calculated index is occupied, then you can simply look at the next records of the table in order (with step 1), until the key K or an empty position in the table is found. At the same time, if the index of the next element to be viewed is determined by adding some constant step (from 1 to n), then this method of resolving collisions is called linear addressing. To calculate the step, you can also use the formula:

$$h = h + a_2,$$

where a is the number of the key lookup attempt. This kind of addressing is called quadratic or random addressing.

With any method of conflict resolution, it is necessary to limit the length of the item search. If more than 3-4 comparisons are needed to find an item, then the effectiveness of using such a hash table is lost and it should be restructured (i.e., find another hash function) to minimize the number of comparisons to find the item

Task

Build a hash table based on the specified data. Compare the search performance in the Bin Balanced Binary Tree, in the binary search tree, and in the hash table. Display the trees and the hash table. Calculate the average number of comparisons to search for data in the specified structures. Restructure the hash table if the average number of comparisons is greater than the specified number. Evaluate the efficiency of using these structures (in terms of time and memory) for the task at hand.

Sample versions of the tasks are given in the appendix.

Instructions for the work

When developing the program interface, it is necessary to provide:

- indication of the type, format and range of data to be entered;
- indication of the actions performed by the program;
- the presence of explanations when displaying the result;
- to display trees graphically (or to offer another visualization in the form of a tree);
- displaying a hash table;
- the ability to change the hash function if the table needs to be restructured;
- Displays the time and number of comparisons when searching for the same data in different data structures.

When testing the program, it is necessary to:

- o check the correctness of data input and output (that is, their compliance with the required type and format), ensure an adequate response of the program to incorrect data entry;
- o Ensure that messages are displayed in the absence of input data ("empty input");
- o check the correctness of operations;
- o provide for the output of a message if it is necessary to restructure the hash table;
- o Check the search for existing data and the search for non-existent data;

Contents of the report

The report on the laboratory work should draw conclusions about which structure (binary search tree, B-tree, or hash table) is appropriate for solving the problem, and what advantages the use of this or that structure provides. The conclusions should be confirmed by the result of numerical comparisons of memory consumption and program execution time, as well as the number of comparisons when searching for data. The report should also indicate when it is necessary to restructure the hash table and what should be paid special attention to when testing the program

The laboratory report should answer the following questions:

1. What is the difference between a perfectly balanced tree and an AVL tree?
2. What is the difference between searching in the AVL tree and searching in the binary search tree?
3. What is a hash table, what is the principle of its construction?
4. What are collisions? What are the methods for resolving them?
5. When does a hash table search become ineffective?
6. Search performance in AVL trees, binary search tree, and hash tables

The report is submitted in electronic or printed form.

Annex

Sample variants of tasks

- 1 Build a binary search tree from the letters of the entered string. Display it as a tree. Highlight all letters that occur more than once. Remove these letters from the tree. Balance the tree after removing duplicate letters. Display it as a tree. Create a hash table containing the letters and the number of occurrences in the entered string. Display the table. Search for the entered letter in the binary search tree, in the balanced tree, and in the hash table. Compare the search time, memory size, and number of comparisons when using different data structures.
- 2 Build a hash table from the words of an arbitrary text file by setting its dimension from the screen. Display the constructed table of words on the screen. Search for the entered word. Run the program for different dimensions of the table and compare the search time and the number of comparisons. Delete all words starting with the specified letter, display the table. Compare the search time, memory size, and number of comparisons when using balanced trees and hash tables.
- 3 Build a hash table for reserved words of the programming language used (at least 20 words) containing HELP for each word. Display a hint for the entered word. Compare the search time, memory size, and number of comparisons when using balanced trees and hash tables. Add a hint for a newly entered word, using table restructuring if necessary. Compare the efficiency of adding a key to a table or restructuring it for different degrees of table fullness.
- 4 A text file contains integers. Build a binary search tree from the file numbers, balance the resulting tree. Display it as a tree. Build a hash table from the file numbers. Search the entered integer in the binary search tree, in the B-tree, and in the hash table. Compare the search time, memory, and number of comparisons when using different data structures.

PRACTICAL WORK No 6.

Topic: Implementation of a multithreaded algorithm

The purpose of the work: to get acquainted with the mechanism of multitasking in modern operating systems, to gain skills in developing a multithreaded application in Windows.

1.1 Theoretical information 1.1.1 Concepts of multiprogramming and multitasking

The most important function of an operating system is to organize the rational use of all its hardware and information resources. The main resources can include processors, memory, external devices, data and programs. The main difficulties arise in multi-program operating systems (OS), in which several applications compete for resources at once. That is why most of all problems relate to multi-program systems.

Multiprogramming, or *multitasking*, is a way of organizing a computing process in which several programs are alternately executed on one processor at once [1]. These programs share not only the processor, but also other computer resources: RAM and external memory, I/O devices, data.

1.1.2 The concepts of "process" and "flow"

In order to support multiprogramming, the operating system must define and design for itself those internal units of work between which the processor and other computer resources will be divided. Nowadays, most operating systems define two types of units of work. A larger unit of work, usually called *a process* or *task*, requires the first execution of several smaller tasks, for which the terms *"thread"* or *"thread"* are used [1].

Thus, a process is an object to which all resources, including the address space, are mapped, i.e. it is a structure in memory. In addition to codes and data, there are also threads in the address space of a process.

The work of a computing system consists in the execution of a certain program. Therefore, a certain program code is associated with the process and with the thread, which for this purpose is drawn up in the form of an executable module. In order for this

program code to be executed, it must be loaded into RAM, possibly allocate some disk space for data storage.

Thus, for efficient processing of tasks in any operating system, along with processes, a special mechanism for paralleling calculations is required, which would take into account the close connections between individual branches of calculations of the same application. For these purposes, modern operating systems use the mechanism of *multithreading* [1]. At the same time, a new unit of work is introduced - the thread of execution, and the concept of "process" significantly changes the meaning. The concept of "thread" corresponds to the sequential transition of the processor from one program instruction to another.

Creating threads requires less overhead from the OS than processes. Unlike processes that belong to different or competing applications, all threads in the same process always belong to the same application, so all threads in the same process share files, timers, devices, the same area of RAM, the same address space. This means that they share the same global variables. Since each thread can access any virtual address of a process, one thread can use the stack of another thread. There is no complete protection between threads of the same process because, firstly, it is not possible, and secondly, it is not necessary. On the other hand, threads of different processes are still well protected from each other.

Thus, multiprogramming is more efficient at the level of threads rather than processes [1].

The use of threads is not only to improve system performance through parallel computing, but also to create more readable, logical programs. The introduction of multiple threads simplifies programming.

The greatest effect of the introduction of multithreaded processing is achieved in multiprocessor systems, in which threads, including those belonging to the same process, can be executed on different processors in real parallel (and not pseudo-parallel).

1.1.3 Features of the implementation of the multiprogramming mechanism to increase the efficiency of the use of the computing system

Multiprogramming is essentially designed to increase the efficiency of using a computing system, but efficiency can be understood in different ways. The most characteristic criteria for the efficiency of computing systems are [1]:

- 1) *bandwidth* - the number of tasks performed by a computer system per unit of time;
- 2) user experience;
- 3) *System reactivity* is the ability of the system to withstand predetermined (possibly very short) time intervals between the start of the program and the receipt of the result.

Depending on the selected criterion of efficiency, operating systems are divided into batch processing systems, time-sharing systems and real-time systems [1]. Each type of operating system has specific internal mechanisms and special areas of application. Some operating systems can support several modes at the same time, for example, some tasks can be performed in batch processing mode, and some - in real time or in time-sharing mode.

Batch Processing Systems

When using multiprogramming to increase the throughput of a computer, the main goal is to minimize downtime for all devices on the computer, especially the CPU. Such downtime can occur due to the suspension of a task for internal reasons, such as waiting for data input to be processed. When this kind of task lock occurs, the processor is switched to another task that has data to be processed. This concept of multiprogramming is the basis of batch systems.

Batch processing systems were designed to solve mainly computational problems that did not require quick results. The main goal and criterion for the efficiency of batch processing systems is maximum throughput, that is, solving the maximum number of tasks per unit of time.

To achieve this goal, batch processing systems use the following scheme of operation: at the beginning of work, a batch of tasks is formed, from this batch of tasks a

multiprogram mixture is formed, i.e. a set of simultaneously executed tasks. For simultaneous execution, tasks are selected so that a balanced load of all devices of the computer is ensured. Thus, a "profitable" task is selected. In computing systems operating under the control of batch operating systems, it is impossible to guarantee the execution of a particular task for a certain period of time.

The timing of I/O and computation operations can be achieved in a variety of ways. One of these is the use of computers that have a specialized I/O processor (channel). Typically, a channel has a set of instructions that are designed to control external devices, such as "check device status", "set magnetic head", "set the beginning of the sheet", "print line". The CPU instruction set provides a special instruction by which parameters and instructions are transmitted to the channel as to what I/O program it should execute. From this point on, the CPU and the channel can work in parallel (Figure 1, a).

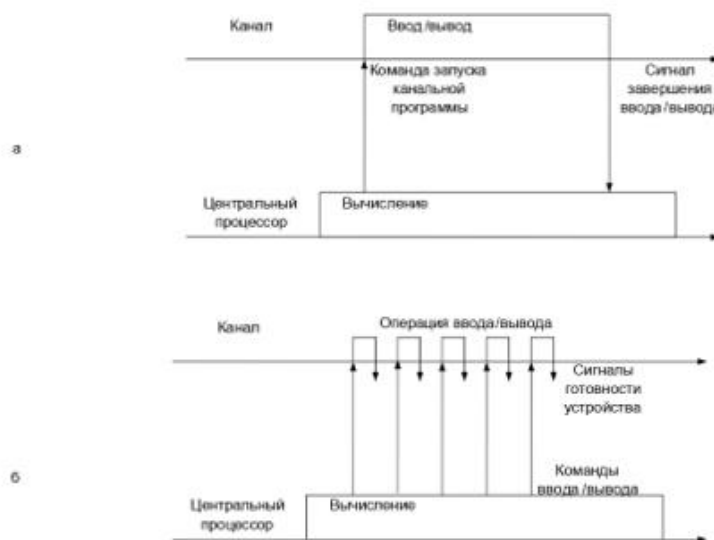
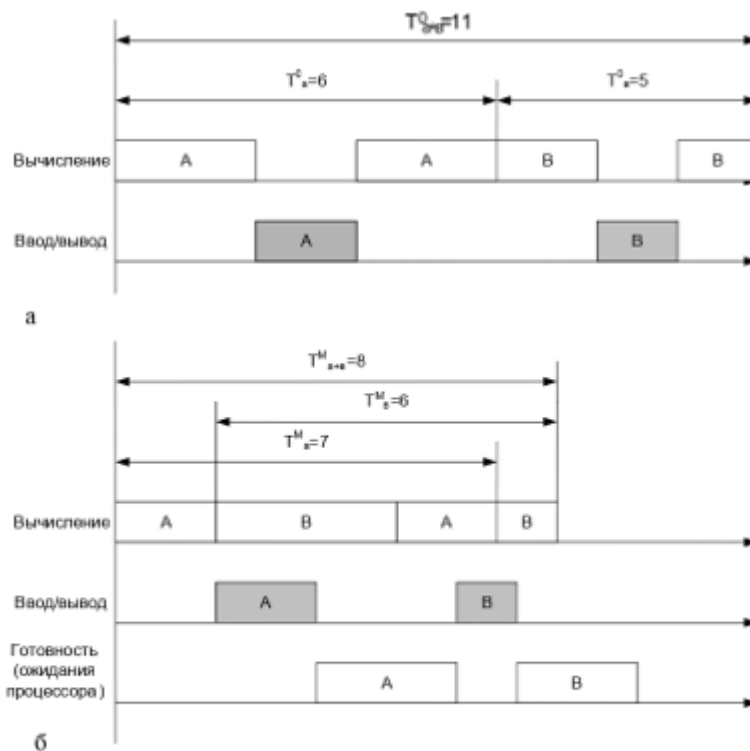


Figure 1 - Parallel execution of calculations and I/O operations [1]

Another way of combining calculations with I/O operations is implemented in computers in which external devices are controlled not by the I/O processor, but by controllers. Each external device (or group of external devices of the same type) has its own controller, which autonomously processes commands coming from the central processor. In this case, the controller and the central processor work asynchronously. The controller executes its device control commands much slower than the central processor executes theirs. This circumstance is used to organize parallel execution of calculations and I/O operations: in the interval between the transmission of commands to the controller, the central processor can perform calculations (Figure 1, b). The controller can inform the central processor that it is ready to receive the next command by an interrupt signal, or the central processor learns about this by periodically polling the state of the controller.



а – в однопрограммной системе; б – в мультипрограммной системе [1]
Рисунок 2 – Время выполнения двух задач

If several tasks are running simultaneously in the system, it becomes possible to combine the calculations of one task with the I/O of another. The total execution time of a mixture of tasks is often less than their total time of sequential execution (Figure 2, a).

However, when you share a processor on a system, there may be situations where a task is ready to run, but the processor is busy performing another task. In such cases, the task that has completed the I/O is forced to wait for the processor to become free, and this extends its execution time.

In batch processing systems, the processor switches from one task to another at the initiative of the most active task. In this case, one task can occupy the processor for a long time and the execution of interactive tasks will become impossible. The interaction of the user with the computer on which the batch processing system is installed is reduced

to the fact that he brings the task, gives it to the dispatcher-operator, and at the end of the day, after completing the entire batch of tasks, receives the result. Obviously, this order increases the efficiency of the equipment, but reduces the efficiency of the user's work.

Features of the implementation of the multiprogramming mechanism in time-sharing systems

Improving the convenience and efficiency of the user's work is the goal of the study of time-sharing systems. In them, users (or one user) are given the opportunity to work interactively with several applications at once. To do this, each application must regularly receive the opportunity to "communicate" with the user. At the same time, the possibilities of the user's dialogue with the application are limited.

In OS time-sharing systems, all applications are alternately allocated a quantum of processor time, during which the program processes messages received by it, after which, regardless of the state of the program, the system takes control from the program and transfers it to another program.

Thus, users who have launched programs for execution get the opportunity to maintain a dialogue with them.

Time-sharing systems are designed to correct the main drawback of batch processing systems - the isolation of the user-programmer from the process of performing his tasks.

Time-sharing systems have less bandwidth than batch processing systems, since each task started by the user is accepted for execution, and not the one that is "beneficial" to the system. This is consistent with the fact that the criterion for the effectiveness of time-sharing systems is not the maximum throughput, but the convenience and efficiency of the user experience. At the same time, the multi-program execution of interactive applications also increases the throughput of the computer (although not to the same extent as batch systems). Hardware loads better because while one application waits for the user to communicate, other applications can be processed by the processor.

Multiprogramming in Real-Time Systems

Another type of multiprogramming is used in *real-time systems* designed to control various technical objects or technological processes from a computer [1]. The criterion

for the efficiency of these systems is the ability to maintain predetermined time intervals between the start of the program and the receipt of the result (i.e., the reaction time of the system). This property of the system is called reactivity.

In real-time systems, a multi-program mixture is a fixed set of pre-developed programs, and the selection of a program for execution is carried out by interruptions (based on the current state of the object) or in accordance with the schedule of planned work.

The ability of computer hardware and OS to respond quickly depends mainly on the speed of switching from one task to another and, in particular, on the speed of processing interrupt signals.

In real-time systems, they do not strive to maximize the load of all devices, on the contrary, they usually have a certain "reserve" of computing power in case of peak load.

If a real-time system is not designed to support peak demand, it can happen that the system will not be able to do the job exactly when it is needed most.

1.1.4 Process and flow planning

One of the main subsystems of a multiprogram OS, which directly affects the functioning of a computer, is the subsystem for managing processes and threads, which is engaged in their creation and destruction, maintains interaction between them, and also distributes processor time between several simultaneously existing processes and threads in the system.

The process and thread management subsystem is responsible for *providing processes with the necessary resources* [1]. The OS maintains special information structures in memory, in which it records what resources are allocated to each process. It can assign resources to the process for sole use or for shared use with other processes. The process management subsystem interacts with other OS subsystems responsible for resource management, such as the memory management subsystem, the input-output subsystem, and the file system.

When multiple independent tasks are running on a system at the same time, additional problems arise. Thread rate negotiation is also very important to avoid "races" (when multiple threads try to modify the same file), deadlocks, or other collisions that

occur when resources are shared. *Thread synchronization* is one of the important functions of the process and thread management engine.

Each time a process is terminated, the OS takes steps to "clean up the traces" of its presence in the system. The process management subsystem closes all files with which the process has been working, frees up areas of RAM allocated for codes, data, and system information structures of the process. All kinds of OS queues and resource lists that contained references to the process being terminated are corrected.

1.1.5 Creating Processes and Threads

To create a process means, first of all, to create *a process handle*, which is one or more information structures containing all the information about the process necessary for the operating system to manage it [1]. Such information may include, for example, the process identifier, data on the location of the executable module in memory, the degree of privilege of the process (priority and access rights), etc.

Creating a process involves loading the code and data of the executable program of the given process from disk into RAM. To do this, the OS must detect the location of such a program on the disk, reallocate RAM, and allocate memory to the executable program of the new process. Next, it must read the program to its allocated memory locations and possibly change the program's settings depending on the location in memory. On systems with virtual memory, only some of the code and data of the process can be loaded at the initial moment in order to "swap" the rest as needed.

In a multithreaded system, when creating a process, the OS creates at least one thread of execution for each process. When creating a thread, just like when creating a process, the operating system generates a special information structure - a thread handle, which contains the thread identifier, data on access rights and priority, on the state of the thread and other information. In its initial state, the thread (or process, if we are talking about a system in which the concept of "thread" is not defined) is in a suspended state. The moment of selecting a thread for execution is carried out in accordance with the rule of providing processor time adopted in this system and taking into account all currently

existing threads and processes. A prerequisite for activating a thread of a process is also the availability of space in RAM for loading its executable module.

1.1.6 Thread states

The OS schedules threads based on their state. In a multiprogram system, a thread can be in one of three basic states:

1) *Execution* is an active state of a thread, during which the thread has all the necessary resources and is directly executed by the processor;

2) *Waiting* is a passive state of a thread in which a thread is blocked for its internal reasons (waiting for some event to occur, for example, the completion of an I/O operation, receiving a message from another thread or releasing some resource it needs);

3) *Readiness* is also a passive state of a thread, but in this case the thread is blocked due to a circumstance external to it (it has all the resources it needs, is ready to execute, but the processor is busy executing another thread).

During its lifetime, each thread transitions from one state to another according to the thread scheduling algorithm adopted by the operating system.

Consider a typical thread state graph (Figure 3). A newly created thread is in a ready state, ready to execute, and queues with the processor. When, as a result of scheduling, the threading engine decides to activate this thread, it enters the execution state and stays in it until it either frees the processor itself, going into a state of waiting for some event, or is forcibly "evicted" from the processor, for example, due to the exhaustion of the quantum of processor time allotted to this thread. In the latter case, the thread returns to the ready state. The thread enters the same state from the waiting state after the expected event occurs.

A uniprocessor system can have no more than one thread in an execution state, and there can be multiple threads in each of the wait and ready states. These threads form queues of waiting threads and ready threads, respectively. Thread queues are organized by combining individual threads into lists of handles. Thus, each thread handle, among other things, contains at least one pointer to another handle adjacent to it in the queue. This queuing arrangement makes it easy to reorder them, include and exclude threads,

and move threads from one state to another. Assuming that Figure 4 shows a queue of ready threads, the scheduled execution order is A, B, E, D, C.



Figure 3 - Graph of thread states in a multitasking environment

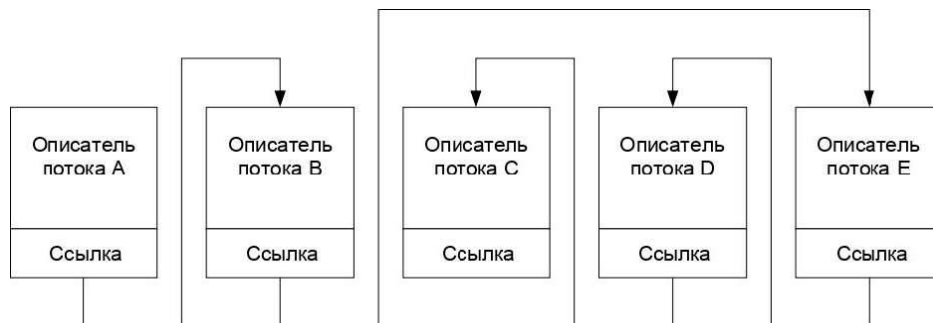


Figure 4 - Queue of threads [1]

1.1.7 Preemptive and non-preemptive planning algorithms

The whole set of planning algorithms can be divided into two classes: preemptive and non-preemptive planning algorithms.

1) *Non-preemptive* algorithms are based on the fact that the active thread is allowed to execute until it voluntarily gives control to the operating system so that it selects the next thread ready to execute from the queue.

2) *Preemptive* algorithms are thread scheduling techniques in which the decision to switch the processor from one thread to another is made by the operating system rather than the active task.

The main difference between preemptive and non-preemptive algorithms is the degree of centralization of the thread scheduling mechanism. In preemptive multiprogramming, the thread scheduling functions are entirely concentrated in the operating system. In this case, the operating system performs the following functions: determines when the active thread is terminated, remembers its context, selects the next thread from the queue of ready threads, starts a new thread for execution by loading its context.

In non-preemptive multiprogramming, the scheduling mechanism is distributed between the operating system and the application programs. The application program, having received control from the operating system, determines when the next cycle of its

execution is completed, and only then transfers control to the OS using some system call. The OS queues threads and selects the next thread for execution according to some rule (for example, taking into account priorities). This mechanism creates problems for both users and application developers.

For users, this means that control of the system is lost for an arbitrary period of time, which is determined by the application (and not by the user). Therefore, developers of applications for an operating environment with non-preemptive multitasking are forced, taking on some of the functions of the scheduler, to create applications so that they perform their tasks in small parts. Such a method of dividing time between tasks significantly complicates the development of programs and imposes increased requirements on the qualifications of the programmer. The programmer must ensure a "friendly" attitude of his program to other programs running simultaneously with it. To do this, the program must provide for frequent transfers of control to the operating system. An extreme manifestation of the "unfriendliness" of an application is its hanging, which leads to a general collapse of the system. In the

As a rule, these situations are excluded because the central scheduling mechanism has the ability to remove the hung task from execution.

However, distributing thread scheduling functions between the system and applications can be an advantage under certain conditions, because it allows the application designer to design the scheduling algorithm that is most appropriate for a given set of tasks. A significant advantage of non-preemptive scheduling is a faster rate-to-thread switchover.

Almost all modern operating systems focused on high-performance application execution implement preemptive thread scheduling algorithms [1, 2].

1.1.8 Thread scheduling and dispatching

Over the lifetime of a process, its threads can be interrupted and continued many times. The transition from one thread to another is done as a result of scheduling and dispatching [1]. The work of determining when to abort the execution of the currently

active thread and which thread should be allowed to execute is called *scheduling* [1]. Thread scheduling is based on information stored in process and thread handles.

Flow planning, in fact, includes the solution of two tasks:

- 1) determination of the moment in time to change the current active flow;
- 2) Selecting a thread from a queue of ready threads to execute.

There are many different flow scheduling algorithms that solve each of the above problems in their own way. Scheduling algorithms can have different goals and provide different quality of multiprogramming.

In most general-purpose operating systems, scheduling is carried out *dynamically* (on-line), i.e., decisions are made while the system is running based on an analysis of the current situation. The OS operates under conditions of uncertainty - threads and processes appear at random moments in time and also end unpredictably. Dynamic schedulers can flexibly adapt to changing situations and do not use any assumptions about a multi-program mixture.

16

Another type of scheduling - static - can be used in specialized systems in which the entire set of simultaneously performed tasks is defined in advance, for example, in real-time systems. A *scheduler* is called *static* (or preliminary) if it makes planning decisions not while the system is running, but in advance (off-line).

The result of a static scheduler is a table called a schedule, which specifies which thread/process should be provided with a processor, when, and for how long. To build a schedule, a scheduler needs as much prior knowledge as possible of the characteristics of a set of tasks.

A ready-made schedule can be used by the operating system to switch threads and processes. At the same time, the overhead of the OS for the execution of the schedule is reduced only to the dispatch of threads/processes.

Dispatching consists in the implementation of a planning solution (dynamic or statistical), that is, switching the processor from one thread to another. Before terminating the execution of a thread, the OS remembers its context in order to later use this information to subsequently resume the execution of this thread. Context reflects, firstly,

the state of the computer's hardware at the time of the thread interruption: the value of the instruction counter, the contents of general-purpose registers, the mode of operation of the processor, flags, interrupt masks and other parameters.

Dispatching is reduced to the following [1]:

- 1) Save the context of the current thread that you want to change.
- 2) Loading the context of a new thread selected as a result of scheduling.
- 3) Start a new thread for execution.

Because the context switching operation has a significant impact on the performance of the computing system, the OS software modules perform thread dispatch in conjunction with the processor hardware.

To implement the scheduling algorithm, the OS must take control whenever an event occurs in the system that requires a reallocation of CPU time. Such events can include the following [1]:

1) a timer interrupt that signals that the time allotted to the active task for execution has run out - the scheduler puts the task in a ready state and reschedules;

2) An active task has made a system call related to a request for I/O or access to a resource that is currently in use—the scheduler puts the task in a waiting state and performs rescheduling.

3) An active task has made a system call related to the release of a resource - the scheduler checks to see if any task is waiting for this resource.

4) An external (hardware) interrupt that signals that the peripheral device has completed an I/O operation, transfers the task to the ready queue, and scheduling is performed.

5) An internal abort signals an error that occurred as a result of the active task - the scheduler cancels the task and reschedules.

For each of these events, the scheduler scans the queues and decides which task will be performed next. In addition to these, there are a number of other events (often related to system calls) that require rescheduling. For example, requests for applications and

users to create a new task or to increase the priority of an existing task create a new situation that requires queue revisions and possibly CPU switching.

In real-time systems, the scheduler is activated by interrupts from the timer to work out a static schedule. These interrupts permeate the entire time axis, occurring at short constant intervals. After each interruption, the scheduler reviews the schedule and checks whether it is time to switch tasks. In addition to interrupts from the timer, in real-time systems, task rescheduling can occur based on interrupts from external devices - various types of sensors and actuators.

1.1.9 Planning algorithms based on quantization

Many preemptive scheduling algorithms are based on the concept of *quantization* [1]. In accordance with this concept, each thread is given a limited continuous period of processor time - a *quantum* - to execute in turn. A change of an active thread occurs if:

- 1) the thread has terminated and left the system;
- 2) an error has occurred;
- 3) the thread has entered a waiting state;
- 4) The quantum of processor time allotted to this thread has been exhausted.

The flow that has exhausted its quantum is transferred to a state of readiness and waits for a new quantum of CPU time to be allocated, and a new thread is selected from the ready queue to execute according to a certain rule. The thread state graph shown in Figure 5 corresponds to a quantization-based scheduling algorithm.

The quanta allocated to threads can be the same for all threads or different. Consider, for example, the case where all threads are provided with quanta of the same length q (Figure 6). If there are n threads in the system, then the time that the thread spends waiting for the next quantum can be roughly estimated as $q(n - 1)$. The more threads in the system, the longer the waiting time and the less the possibility of simultaneous interactive work for several users. But if the value of the quantum is chosen very small, then the value of the product $q(n - 1)$ will still be small enough for the user not to feel discomfort from the presence of other users in the system. The typical value of the quantum in time-sharing systems is tens of milliseconds.

If the quantum is short, the total time a thread spends waiting for the processor is directly proportional to the time it takes to execute (i.e., the time it would take for the thread to execute if the computing system were exclusive). The larger the quantum, the more likely it is that threads will terminate in the first execution cycle, and the less obvious the dependence of thread wait time on their execution time becomes. With a large enough quantum, the quantization algorithm degenerates into a sequential processing algorithm inherent in single-program systems, in which the waiting time for a task in the queue does not depend on its duration at all.

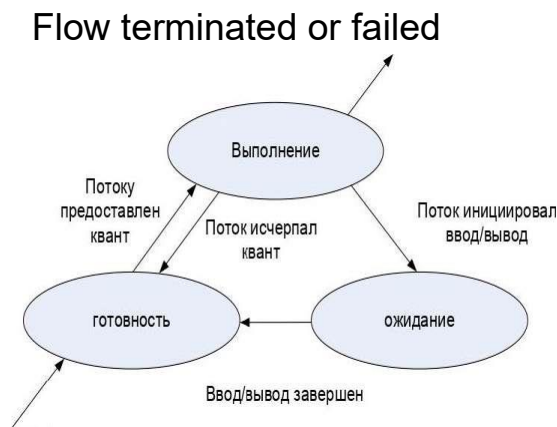


Figure 5 - Graph of flow states in a system with quantization [1]

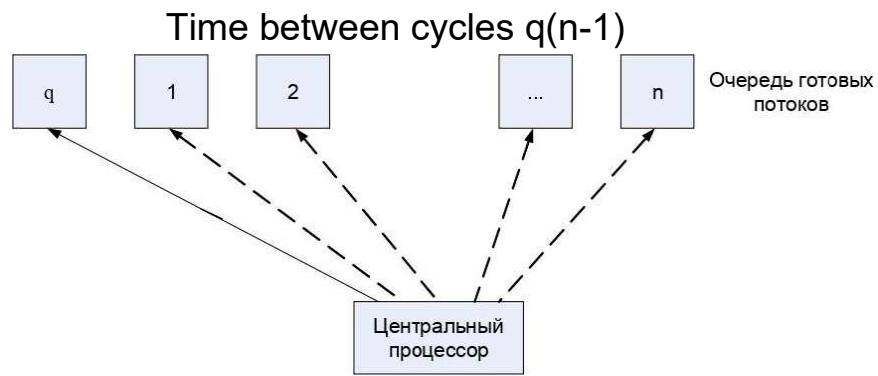


Figure b - Illustration of queue wait time calculation

The quanta allocated to a single thread can be fixed, or they can change at different times in the thread's lifetime. This approach reduces the overhead of switching tasks when multiple tasks perform long-running calculations at once.

Threads receive a time quantum to execute, but some of them do not fully use it, for example, because they need to perform data input or output. The scheduling algorithm can correct this "injustice". As compensation for unused quanta, threads receive privileges for subsequent maintenance. To do this, the scheduler creates two queues of ready threads (Figure 7).



Figure 7 - Quantization with a preference for I/O intensive flows [1]

Queue 1 is formed by threads that have reached a state of readiness as a result of the exhaustion of the time quantum, and queue 2 is formed by threads that have completed the I/O operation. When selecting a thread to execute on the processor, the second queue is first looked, and only if it is empty, the quantum is allocated to the thread from the first queue.

Multitasking operating systems lose some CPU time to perform auxiliary work during task context switching. At the same time, registers, flags, and stack pointers are remembered and restored, and the status of tasks is checked for handover. The cost of these auxiliary actions does not depend on the value of the time quantum, so the larger the quantum, the lower the total overhead associated with thread switching [1].

1.1.10 Priority-based planning algorithms

Another important concept underlying many crowded-out scheduling algorithms is *priority service* [1]. Priority

Maintenance assumes that threads have priority, on the basis of which the order of their execution is determined. *Priority* is a number that characterizes the degree of privilege of a thread when using computer resources, in particular CPU time: the higher the priority, the higher the privileges and the less time the thread will spend in queues [1].

Priority can be expressed as an integer or a fractional, positive or negative value. In some operating systems, it is accepted that the priority of a thread is higher the higher the number denoting priority (in the arithmetic sense). In other systems, on the contrary, the lower the number, the higher the priority.

In most operating systems that support threads, the priority of a thread is directly related to the priority of the process within which the thread runs. The priority of a process is assigned by the operating system when it is created. The priority value is included in the process handle and is used when assigning priority to threads of that process. When assigning priority to a newly created process, the OS takes into account whether the process is system or application, what is the status of the user who started the process, and whether the user explicitly indicated that the process was assigned a certain priority level to the process.

In many operating systems, it is possible to change priorities during the life of a thread. Priority changes can occur at the initiative of the thread itself or at the initiative of the user. In addition, the OS itself can change the priorities of threads depending on the situation in the system. In the latter case, priorities are called *dynamic*, in contrast to immutable, *fixed*, priorities [1].

The efficiency of the entire computing system depends significantly on what priorities are assigned to threads. In most cases, the OS assigns priorities to threads by default.

As an example, let's look at the thread prioritization scheme adopted in the Windows NT operating system (Figure 8). The system defines 32 priority levels and two classes of threads - real-time threads and threads with variable priorities. The range from 1 to 15 inclusive is reserved for threads with variable priorities, and from 16 to 31 for more time-critical real-time streams (priority 0 is reserved for system purposes).

When a process is created, a thread is given a default base priority at the top or bottom of the range, depending on the class. The base priority of the process can be further increased or lowered by the operating system.

In Windows NT, over time, the priority of a thread that belongs to the variable priority class can deviate from the basic priority.

22

Theta of the thread. The OS can increase the priority of a thread (which in this case is called dynamic) in cases where the thread has not fully utilized its allotted quantum, or lower the priority if the quantum has been fully utilized. The OS increases the priority differentially depending on what type of event prevented the thread from fully using the quantum. In particular, the OS prioritizes more q threads that are waiting for keyboard input (interactive applications) and to a lesser extent - threads that perform disk operations. It is on the basis of dynamic priorities that thread scheduling.

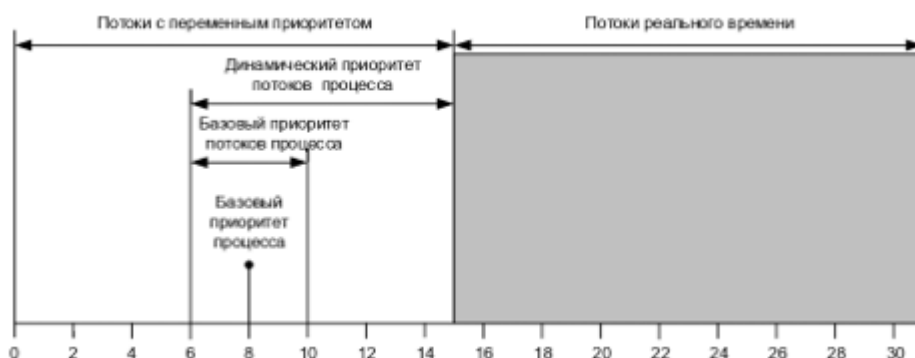


Figure 8 - Prioritization scheme in Windows NT [1]

There are two types of priority planning: service with relative priorities and service with absolute priorities.

In both cases, the selection of a thread to execute from the ready queue is the same: the thread with the highest priority is selected. However, the problem of determining when the active thread changes is solved in different ways. On systems with relative priorities, the active thread executes until it leaves the processor by itself in a waiting state (or an error occurs, or the thread terminates). Figure 9 shows a graph of thread states in a system with relative priorities.

On systems with absolute priorities, the execution of the active thread is interrupted, in addition to the reasons mentioned above, under one other condition: if a thread whose priority is higher than the priority of the active thread appears in the queue of ready threads. In this case, the interrupted thread enters the ready state (Figure 10).

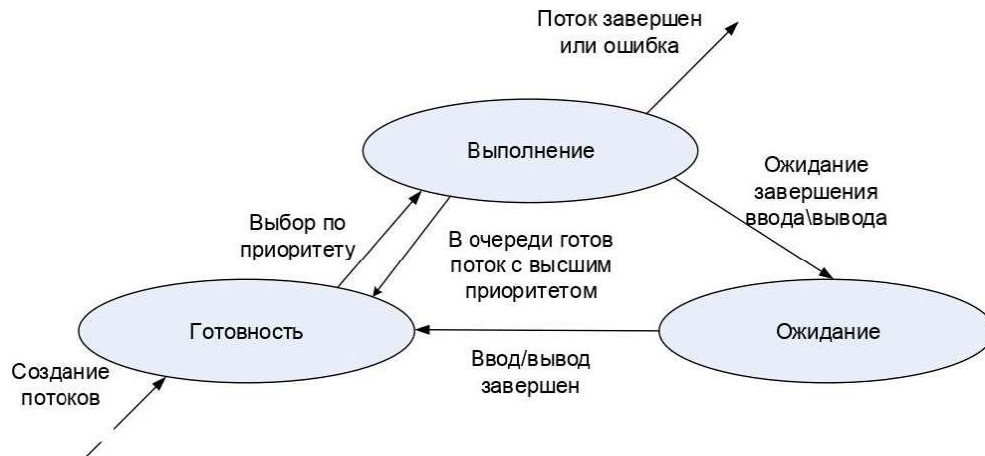


Figure 10 - Graphs of flow states in systems with absolute priorities [1]

In systems where planning is based on relative priorities, the cost of switching the processor from one job to another is minimized. On the other hand, there may be situations where one task takes up a long time on the processor. Clearly, this service discipline is not suitable for time-sharing and real-time systems: an interactive application can wait for its turn for hours until the computational task needs I/O.

On systems with absolute priorities, queued thread wait times can be minimized by assigning it the highest priority. Such a thread will evict all other threads from the processor (except for threads that have the same highest priority). This makes absolute priority scheduling suitable for object management systems where a quick response to an event is important.

1.1.11 Mixed planning algorithms

In many operating systems, scheduling algorithms are built using both the concept of quantization and priorities [1]. For example, scheduling is based on quantization, but the size of the quantum and/or the order in which a thread is selected from the ready queue is determined by the priorities of the threads. This is how scheduling is implemented in the Windows NT system, which combines quantization with dynamic absolute priorities. The ready thread with the highest priority is selected for execution. It is allocated a time quantum. If a thread with a higher priority appears in the ready queue during execution, it preempts the executing thread. The preempted thread returns to the ready queue, and it becomes ahead of all other threads with the same priority.

1.1.12 Thread Synchronization Task

There is a fairly extensive class of operating system tools that can be used to synchronize processes and threads. The need for thread synchronization arises only in a multiprogram operating system and is associated with the joint use of hardware and information resources of the computing system. Synchronization is necessary to eliminate races and dead ends when exchanging data between threads, dividing data, when accessing the processor and I/O devices.

In many operating systems, these tools are called InterProcess Communications (IPC) [1]. Usually, IPC tools include not only interprocess synchronization tools, but also interprocess data exchange tools.

Threads in the general case (when the programmer has not taken special measures to synchronize them) flow independently, asynchronously to each other. This is true both for threads of the same process executing common program code, and for threads of different processes, each of which executes its own program.

Any interaction between processes or threads is associated with their *synchronization*, which consists in coordinating their rates by suspending the flow until the occurrence of a certain event and its subsequent activation when this event occurs. Synchronization is the basis of any interaction between threads.

Hundreds of resource allocation and release events occur in the system every second, and the OS must have reliable and powerful features that allow it to synchronize threads with events occurring in the system.

To synchronize the threads of application programs, the programmer can use both his own synchronization tools and techniques, and the tools of the operating system. However, in many cases, the synchronization tools provided by the operating system in the form of system calls are more effective or even the only possible ones. For example, threads belonging to different processes do not have the ability to interfere in any way with each other's work. Without the mediation of the operating system, they cannot suspend each other or notify about an event that has occurred. Synchronization tools are used by the operating system not only to synchronize application processes, but also for its internal needs.

Typically, operating system developers provide application and system programmers with a wide range of synchronization tools. These tools can form a hierarchy, where simpler tools build on more complex ones, and they can also be functionally specialized. Often, the functionality of different synchronization system calls overlaps, so that a programmer can use several calls to solve one problem, depending on his personal preferences.

Neglecting synchronization in a multithreaded system can lead to a wrong solution or even a crash of the system. Consider, for example, the task of maintaining a customer database of a business (Figure 11). Each customer is assigned a separate record in the database, which has the Order and Payment fields, among other fields. The program that maintains the database is designed as a single process with several threads, including thread A, which records information about orders received from customers in the database, and thread B, which records information about customer payments to the database.

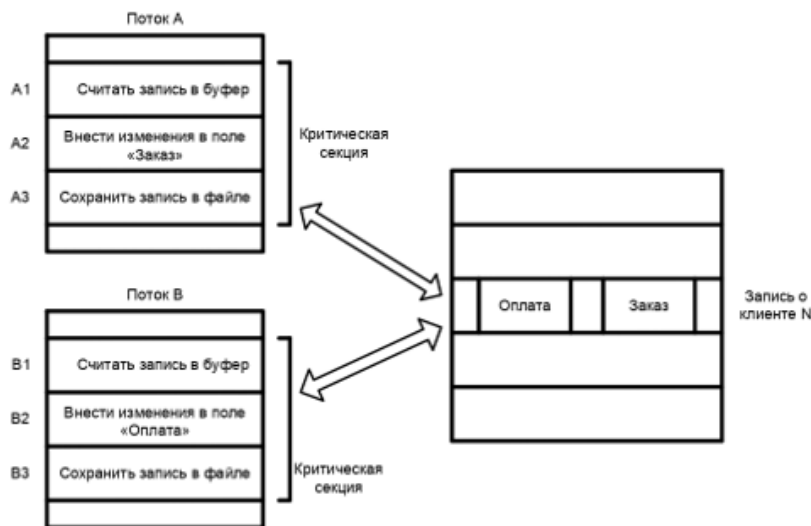


Figure 11 - Occurrence of races when accessing shared data

Both of these threads work together on a common database file, using the same type of algorithms that include three steps [1]:

- 1) read from the database file to the buffer a record about the client with the specified identifier;
- 2) add a new value to the Order (for flow A) or Payment (for flow A) field C);
- 3) Return the modified record to the database file.

Let's denote the corresponding steps for thread A as A1, A2, and A3, and for thread B as B1, B2, and OT. Suppose that at some point thread A updates the Order field of the customer record N. To do this, it reads this record to its buffer (step A1), modifies the value of the Order field (step A2), but does not have time to make the record to the database (step A3), because its execution is interrupted, for example, due to the completion of a time quantum.

Suppose also that thread B also needs to enter payment information for the same customer N. When it is thread B's turn, it has time to buffer the record (step B1) and update the Billing field (step B2), and then aborts. When control is handed over to thread A again, it will continue its work and write the customer record N with the modified Order field to the database (step A3). After the interruption of thread A and the activation of thread B, the latter will write to the database on top of the newly updated record about customer N its version of the record, in which the value of the Payment field has been updated. Thus, the database will record information that customer N has made a payment, but the information about his order will be lost (Figure 12, a).

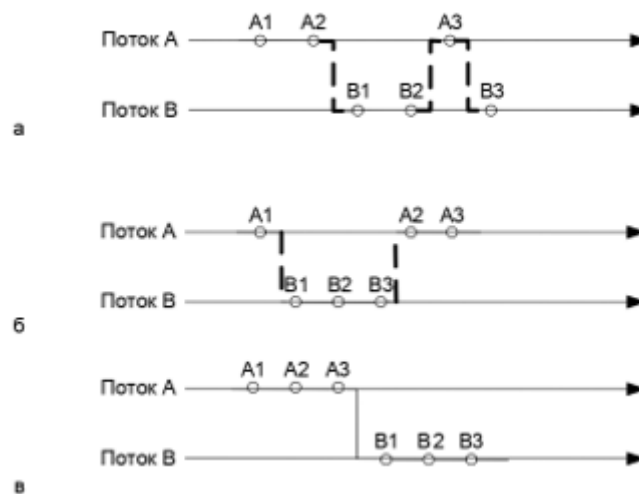


Figure 12 - Influence of relative flow rates on the result of the solution Tasks

The complexity of the synchronization problem lies in the irregularity of the situations that arise. For example, in the previous example, we can imagine another development of events: information about payment (Figure 12, b) may have been lost, rather than about the order (Figure 12, b) or, on the contrary, all corrections were successfully made (Figure 12, c). Everything is determined by the mutual speeds of the streams and the moments of their interruption. Therefore, debugging interacting threads is a difficult task.

1.1.13 The concept of a critical section

An important concept of thread synchronization is the concepts of *"critical section"* and *"blocking variables"*. A *critical section* is a part of a program whose execution result

can change unpredictably if the variables related to this part of the program are changed by other threads while the execution of this part is not yet completed.

To avoid the effect of races on critical data, you need to ensure that only one thread is in the critical section associated with that data at a time. This technique is called *mutual exclusion*. The operating system uses different ways to implement mutual exclusion.

The simplest and most inefficient way to enforce mutual exclusion is for the operating system to allow the thread to deny any interruptions while it is in the critical section. However, this method is almost non-existent because it is dangerous to entrust control of the system to a user thread - it can occupy the processor for a long time, and if a thread in a critical section crashes, the entire system will crash, because interrupts will never be allowed.

Blocking variables are variables to which all threads of the process have direct access. The programmer works with them without referring to the OS system calls.

Each set of critical data is assigned a binary variable to which the thread assigns a value of 0 when it enters the critical section and a value of 1 when it leaves it. Figure 13 shows a fragment of a thread algorithm that uses the blocking variable $F(D)$ to implement mutual exclusion of access to critical data D .

Before entering the critical section, the thread checks to see if any thread is already working with data D . If the variable $F(D)$ is set to 0, the data is busy and the check is looped. If the data is free ($F(D) = 1$), then the value of the variable $F(D)$ is set to 0 and the thread enters the critical section. After the thread has performed all actions with data D , the value of the variable $F(D)$ is set to 1 again.

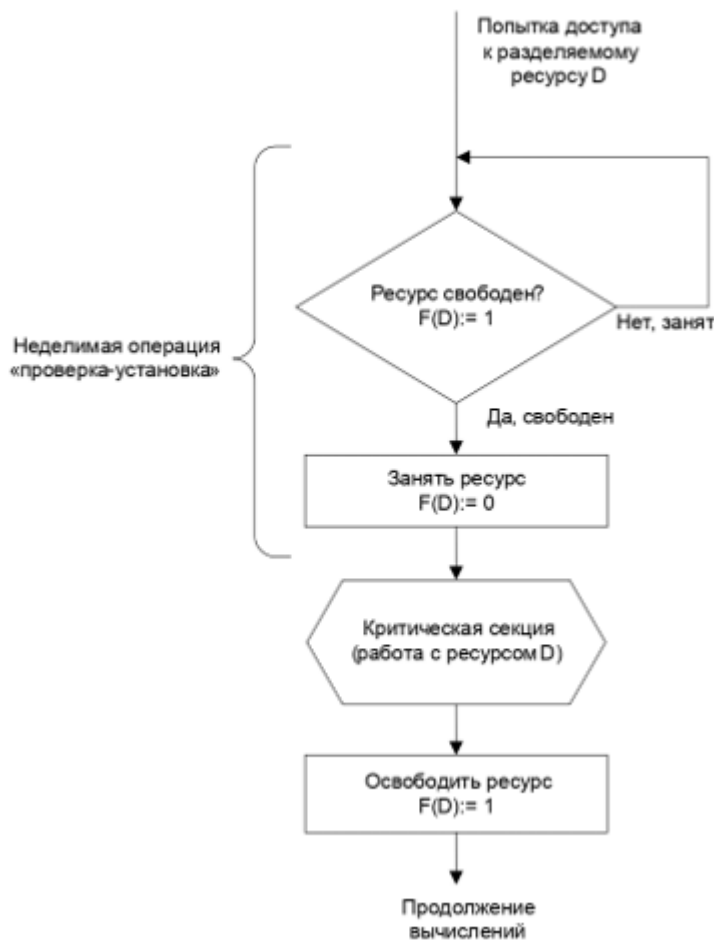


Figure 13 - Implementation of critical sections using blocking variables [1]

Locking variables can be used not only when accessing shared data, but also when accessing shared resources of any kind.

Threads can be interrupted by the operating system at any time and in any place, including in the critical section.

However, a thread should not be interrupted between checking and setting a blocking variable. This method of limiting interrupts has a significant drawback: during the time that one thread is in a critical section, another thread that needs the same resource will continuously poll the blocking variable when it accesses the processor, wasting the CPU time allocated to it that could be used to execute some other thread.

So, Figure 14 shows how the functions for working with the critical section are used to implement mutual exclusion in the Windows NT operating system.

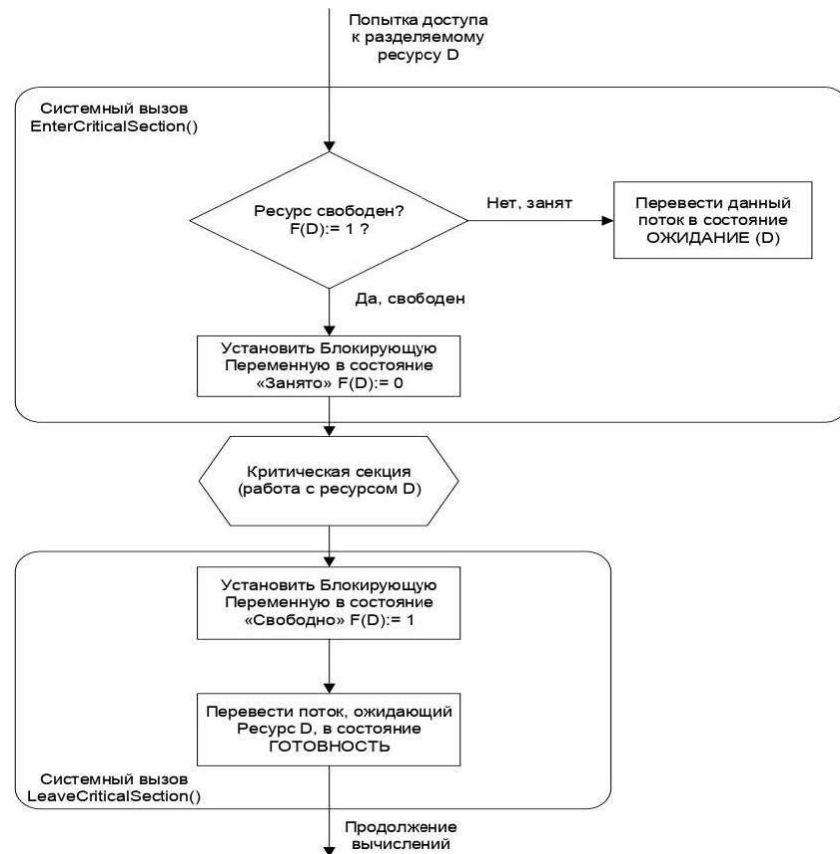


Figure 14 - Implementation of mutual exclusion using the system functions of entering and exiting a critical section [1]

Before starting to modify critical data, a thread executes a system call to `EnterCriticalSection()`. As part of this call, it first checks for a blocking variable that reflects the state of the critical resource. If the system call determines that the resource is busy ($F(D) = 0$), it puts the thread in a waiting state (D) and notes that

that the thread should be activated when the resource is freed. The thread that is using the resource at that time must execute the `LeaveCriticalSection()` system function after exiting the critical section, causing the blocking variable to assume the value corresponding to the free state of the resource ($F(D) = 1$), and the operating system looks at the queue of threads waiting for the resource and puts the first thread out of the queue into the ready state.

Thus, there is no unproductive loss of CPU time on a cyclic check for the release of a busy resource.

1.1.14 OS Synchronization Objects

The synchronization mechanisms discussed above, which are based on the use of global process variables, have a significant drawback—they are not suitable for synchronizing threads of different processes. In such cases, the operating system must provide threads with system synchronization objects that are visible to all threads, even if they belong to different processes and operate in different address spaces.

Examples of such OS synchronization objects are system semaphores, mutexes, events, timers, and so on.

In addition, such "normal" OS objects as files, processes, and threads can be used for synchronization. All these objects can be in two states: signaled and non-signaled - free. For each object, the meaning of the concept of "signaling state" depends on the type of object. For example, a thread enters a signaling state when it terminates. A process enters a signaled state when all its threads are terminated. A file enters a signaled state when the I/O operation for this file is completed. For other objects, the signaled state is established as a result of special system calls. Threads are suspended and activated depending on the state of synchronizing OS objects.

Threads use a special system call to tell the operating system that they want to synchronize their execution with the state of some object. We will refer to this system call as `Wait(X)`, where `X` is a pointer to the synchronization object. The system call that a thread can use to put a synchronization object into a signaled state will be called `Set(X)`.

A thread that has made a system call to `Wait(X)` is placed in a wait state by the operating system until object `X` enters the signaled state. Examples of system calls of type `Wait()` and `Set()` are the calls to `WaitForSmgleObjectQ` and `SetEventQ` in Windows NT.

A thread can wait for several objects to set the signaled state. A thread can also specify as an argument to the `Waitf` system call the maximum time it will wait for the object to enter the signaled state, after which the OS must activate it anyway.

Synchronization is closely related to thread scheduling. First, any thread that accesses the `Wait(X)` system call entails actions in the scheduling subsystem—the thread is taken out of execution and placed in the waiting thread queue, and a new thread is selected and activated from the queue of ready threads. Second, when an object enters a signaled state (as a result of the execution of some thread, either system or application), the thread (or threads) waiting for that object is transferred to the queue of ready-to-execute threads. In both cases, threads are rescheduled.

However, the range of events with which a thread may need to synchronize its execution is by no means limited to the completion of a thread, process, or I/O operation. Therefore, the OS usually has other, more generic synchronization objects, such as *an event, a mutex, a system semaphore*, and so on.

Semaphores are variables that can take integer non-negative values.

To work with semaphores, two primitives are introduced, traditionally referred to as `P` and `V`. Let the variable `S` be the semaphore. Then the actions `V(S)` and `P(S)` are defined as follows:

- 1) `V(S)`: The variable `S` is incremented by 1 single action. Fetching, growing, and remembering cannot be interrupted. The variable `S` cannot be accessed by other threads while this operation is in progress;

2) P(S): Decrease S by 1, if possible. If S=0 cannot be reduced by remaining in the non-negative integer range, then the thread calling operation P waits for the reduction to become possible. Successful validation and decrement are also indivisible operations.

No interruptions during the execution of the V and P primitives are not allowed.

In the special case, when the semaphore S can only accept the values 0 and 1, it becomes a blocking variable called a binary semaphore.

Let's consider the use of semaphores using the example of the interaction of two multi-programmable threads, one of which writes data to the buffer pool and the other reads it from the buffer pool. Suppose the buffer pool consists of N buffers, each of which can contain one record. In general, a writer thread and a reader thread can have different

rate and access the buffer pool at a variable rate. To work together properly, the writer thread must pause when all buffers are occupied, and activate when at least one buffer is freed. In contrast, the reader thread must pause when all buffers are empty and activate when at least one entry appears.

Let's introduce two semaphores: e is the number of empty buffers and f is the number of full buffers, and in the initial state $e = N$, and $f = 0$. Then the work of threads with a shared buffer pool can be described as follows (Figure 15).

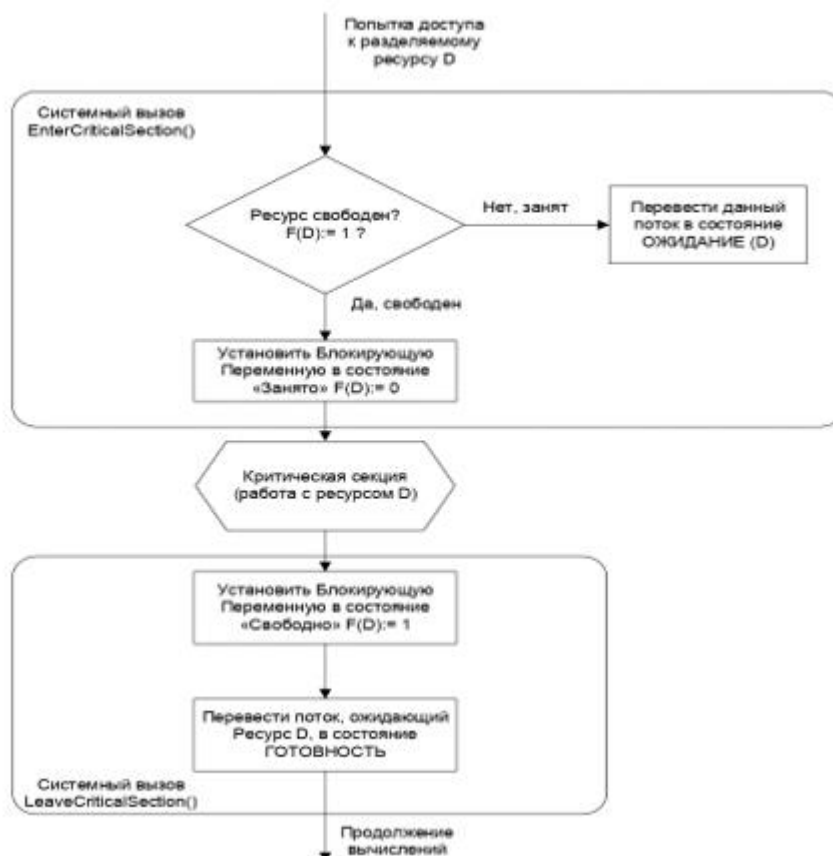


Figure 15 - Using semaphores to synchronize flows [1]

The writer thread first performs the P(e) operation, which checks to see if there are any empty buffers in the buffer pool. According to the semantics of the P operation, if the semaphore e is 0 (that is, there are no free buffers at the moment), the writer thread enters the waiting state. If e is a positive number, it reduces the number of free buffers, writes the data to the next free buffer, and then increases the number of occupied buffers with the V(f) operation. The reader thread acts in a similar way, except that it starts by checking for full buffers, and after reading the data, it increases the number of free buffers.

In this case, it is preferable to use semaphores instead of blocking variables. Indeed, the critical resource here is the buffer pool, which can be represented as a set of identical resources - separate buffers. The semaphore allows a specified number of threads to the shared resource pool. In the buffer pool example, a maximum of N threads can be running, some of which can be "writers" and some of which can be "readers".

In this way, semaphores allow you to effectively solve the problem of synchronizing access to resource pools.

The semaphore can also be used as a blocking variable. In the example discussed above, in order to avoid collisions when working with a shared memory area, we will assume that writing to the buffer and reading from the buffer are critical sections. We will provide mutual exclusion using a binary semaphore (Figure 16).

Both threads must check the availability of the critical section after checking the availability of the buffers.

A mutex, like a semaphore, is typically used to control access to data.

Unlike thread objects, process objects, and file objects, which, when signaled, put all threads waiting for this event into a ready state, a mutex object "releases" only one thread from the waiting queue.

The work of a mutex is well explained in terms of "ownership". Suppose a thread trying to access critical data has made a system call `Wait(X)`, where `X` is a pointer to the mutex. If the mutex is in a signaled state, in this case the thread immediately becomes its owner, setting it to a non-signaled state, and enters the critical section. After the thread has done its work with the critical data, it "gives" the mutex, setting it to the signaled state. At this point, the mutex is free and does not belong to any thread. If any thread is waiting for it to be released, it becomes the next owner of this mutex, at the same time the mutex goes into a non-signaled state.

An event object (in this case, the word "event" is used narrowly to refer to a particular kind of synchronization object) is usually used not to access data, but to alert other threads that some actions have been completed. For example, suppose that in some application the work is arranged in such a way that one thread reads data from a file into a memory buffer and other threads process that data, then the first thread reads the new piece of data, and the other threads process it again, and so on. At the beginning, the first thread sets the event object to a non-signaled state. All other threads have made a call to `Wait(X)`, where `X` is the event pointer, and are in a suspended state, waiting for this event to occur. As soon as the buffer is full, the first thread informs the operating system by making a call to `Set(X)`. The operating system looks at the queue of waiting threads and activates any threads that are waiting for this event.

1.1.15 Threads in Windows. Thread Function. Creating a Thread

Thread Function [2]

In the primary thread, the thread function is *main*, *wmain*, *WinMain*, or *wWinMain*. If you create a secondary thread, it will also have an input function like this:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {  
    DWORD rtwResult = 0;  
  
    return(dwResult); }
```

A thread function can perform any task. Sooner or later, it will finish its work and return control. At that point, the thread will stop and the memory allocated to its stack will be freed, and the user counter of its kernel object "thread" will decrease by 1. When the counter is zeroed, this kernel object will be destroyed. But like the kernel object "process", it can live much longer than the thread associated with it.

Creating a Flow [2]

When the `CreateProcess` **function is called**, the primary thread of the process appears. To create additional threads, you need to call the `CreateThread` function from the primary thread:

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES psa,  
    DWORD cbStack,  
    LPTHREAD_START_ROUTINE pfnStartAddr,  
    LPVOID pvParam,  
    DWORD tdwCreate,  
    LPDWORD pdwThreadId) ;
```

Each time this function is called, the system creates a kernel object called a thread. This is not the thread itself, but a compact data structure that is used by the operating system to control the thread and stores statistical information about the thread.

The system allocates memory to the thread stack from the address space of the process. The new thread runs in the context of the same process as the parent thread. Therefore, it accesses all kernel object handles, all memory, and the stacks of all threads in the process. This allows threads within the same process to communicate with each other easily.

The `psci` parameter is a pointer to the `SECURITY_ATTRIBUTES` structure. If you want the kernel thread object to be assigned default security attributes, `NULL` is passed

in this parameter. And in order for child processes to inherit the object's handle, you need to define the SECURITY_ATTRIBUTES structure and initialize its *hInheritHandle* element to TRUE.

The cbStack parameter. This parameter determines how much of the address space a thread can use for its stack. Each thread is allocated a separate stack. The CreateProcess function calls CreateThread when the application starts, and this function initializes the primary thread of the process.

The pfnStartAddr and pvParam parameters. The pfnStartAddr parameter specifies the address of the thread function with which the thread to be created, and the pvParam parameter is identical to the pvParam parameter of the thread function. CreateThread only passes this parameter along the baton to the function with which the thread being created begins. Thus, this parameter allows you to pass some initializing value to the thread function. It can be either just a numeric value, or a pointer to a data structure with additional information.

The fdwCreate parameter. This parameter defines additional flags that control the creation of a thread. It takes one of two values: 0 (thread execution starts immediately) or CREATE_SUSPENDED. In the latter case, the system creates the thread, initializes it, and suspends it until further notice CREATE_SUSPENDED.

The pdwThreadId parameter. The last parameter of the function is the address of a variable of type DWORD, in which the function returns an identifier assigned by the system to the new thread.

Completion of the stream [2]

A flow can be terminated in four ways:

- 1) The thread function returns control (the recommended way).
- 2) The thread self-destructs by calling the ExitThread function (an unwanted way).
- 3) one of the threads of this or a third-party process calls the TerminateThread function (an unwanted method);

4) The process that contains the thread is terminated (unwanted).

Return control of a thread function. We recommend that you design a function so that the thread terminates only after it returns. This is the only way to ensure that all resources that belong to the thread are cleaned up gracefully.

The *ExitThread function*. A thread can be forcibly terminated by calling:

```
VOID ExitThread(DWORD dwExitCode);
```

This frees up all operating system resources allocated to the thread, but does not clear C/C++ resources (such as objects created from C++ classes). This is why it is better to return from a thread function than to call the *ExitThread* function yourself.

The *dwExitCode parameter contains* a value that the system treats as thread termination code. This function does not have a return value because the thread ceases to exist when it is called.

TerminateThread function. Calling this function also terminates the thread:

```
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode)  
;
```

Unlike *ExitThread*, which destroys only the calling thread, this function terminates the thread specified in the *hThread parameter*. The *dwExitCode parameter contains* a value that the system treats as thread termination code. After the thread is destroyed, the user count decreases by 1.

Pausing and resuming threads [2]

When *CreateProcess* **or** *CreateThread* **is called**, it is initialized with a value of 1, which prohibits the system from allocating CPU time to a new thread.

After the thread is fully initialized, CreateProcess or CreateThread checks to see if the CREATE_SUSPENDED flag has been passed to it, and if so, returns to the thread, leaving the thread suspended. Otherwise, the idle counter is zeroed.

After you create a thread in a suspended state, you can configure some of its properties. When you are done configuring, you need to allow the thread to run. To do this, use the Resume Thread function:

DWORD ResumeThread(**HANDLE** *hThread*) ;

If the call to ResumeThread is successful, this function returns the previous value of the thread's idle counter; otherwise, 0xFFFFFFFF.

A single thread can be suspended multiple times. If a thread is suspended three times, it must also be suspended three times. A thread can be suspended not only when it is created with the CREATE_SUSPENDED flag, but also by calling SuspendThread:

DWORD SuspendThread(**HANDLE** *hThread*) ;

Any thread can call this function and suspend the execution of another thread.

Sleep function [2]

A thread can instruct the system not to allocate CPU time to it for a certain period by calling:

VOID ***Sleep***(**DWORD** *dwMilliseconds*) ;

This function pauses the thread at *dwMilliseconds*.

Features:

1) by calling Sleep, the thread voluntarily gives up the remainder of the time quantum allocated to it;

- 2) The system stops allocating CPU time to the thread for a period approximately equal to the specified one;
- 3) you can call Sleep and pass the value INFINITE to *dwMilliseconds* , disabling the thread from scheduling at all;
- 4) you can call Sleep and pass a zero value to *dwMilliseconds* - that is, give up the rest of your time quantum and force the system to connect another thread to the processor.

When the system is booted, a special thread is created - a zero page thread, which is assigned a zero priority level. No thread, in addition, can have a zero priority level. It zeroes free pages in RAM in the absence of other threads that require attention from the system.

1.1.16 Thread synchronization in WINDOWS

Synchronization with critical sections [2-4]

Windows uses the CRITICAL_SECTION structure to implement critical sections, which is handled exclusively through Windows functions, passing them the address of the corresponding instance of the structure. The structure is initialized using the following function:

```
VOID InitializeCriticalSection(  
                                LPCRITICAL_SECTION pes);
```

This function initializes the elements of the CRITICAL_SECTION structure that the *pes* parameter points to . Because all the work of this function is to initialize multiple member variables, it fails and therefore returns nothing (void). InitializeCriticalSection must be called before one of the threads can access EnterCriticalSection. A piece of code that works with a shared resource begins with a call:

VOID EnterCriticalSection(LPCRITICAL_SECTION pes);

EnterCriticalSection - examines the values of the CRITICAL_SECTION structure elements. If the resource is busy, they contain information about which thread is using the resource. EnterCriticalSection does the following:

If the resource is free, EnterCriticalSection modifies the structure elements to indicate that the calling thread is taking the resource, and then immediately returns and the thread continues its work.

If the resource has already been captured by the calling thread, EnterCriticalSection updates the resource to note the number of times the thread has grabbed the resource in a row and immediately returns. This is not often the case, except when the thread calls EnterCriticalSection twice in a row without an intermediate call to LeaveCriticalSection.

If the values of the structure elements indicate that the resource is occupied by another thread, EnterCriticalSection puts the calling thread on standby. The system remembers that the thread wants to access the resource, and as soon as the thread that occupied the resource calls LeaveCriticalSection, it starts allocating CPU time to the thread again.

At the end of a code chunk that uses a shared resource, there must be a call:

VOID LeaveCriticalSection(LPCRITICAL_SECTION pes);

This function looks at the elements of the CRITICAL_SECTION structure and decrements the count of the resource grabs by the calling thread by 1. If the value is greater than 0, LeaveCriticalSection does nothing and simply returns.

If the counter value reaches 0, LeaveCriticalSection first checks to see if there are other threads on the system waiting for the resource in the EnterCriticalSection call. If there is at least one such thread, the function configures the values of the structure elements to signal that the resource is in use, and gives it to one of the waiting threads

(the thread is chosen "fairly"). If no one needs the resource, LeaveCriticalSection resets the structure elements.

An example of a critical section implementation:

```
HANDLE htl;  
CRITICAL_SECTION cs;  
int x;  
PVOID pvParam =(PVOID) &x;
```

—Initializing a critical section -----

```
InitializeCriticalSection((LPCRITICAL_SECTION)& cs);
```

—ThreadFunc1 thread function-----

```
DWORD WINAPI ThreadFunc1(PVOID pvParam){ while (true){  
    EnterCriticalSection((LPCRITICAL_SECTION)& cs);  —Code  
that works with the share //—Exit critical section  
    LeaveCriticalSection((LPCRITICAL_SECTION)& cs);  
    Sleep(100); }  
}
```

—Delete a critical section-----

```
DeleteCriticalSection(&cs);
```

Thread synchronization using kernel objects. Kernel object wait functions [5,6]

Windows has special functions for waiting for synchronizing kernel objects (wait functions) that allow a thread to pause at any time and wait for some kernel object to be freed. Of the entire family of these functions, WaitForSingleObject is the most commonly used:

```
DWORD WaitForSingleObject( HANDLE hHandle, DWORD  
dwMilliseconds) ;
```

When a thread calls this function, *the hHandle* parameter identifies a kernel object that maintains a free-busy state. The second parameter, *dwMilliseconds*, specifies how long, in milliseconds, the thread is willing to wait for the object to be released.

The next call tells the system that the thread will wait until the process identified by the *hProcess* handle completes.

```
WaitForSingleObject(hProcess, INFINITE);
```

In this case, the constant INFINITE tells the system that the calling thread is ready to wait for this event for at least an eternity. It is this constant

42

is usually passed to the WaitForSingleObject function, but you can specify any value in milliseconds.

The WaitForMultipleObjects function allows you to wait for several objects to be released at once or one from the list of objects:

```
DWORD WaitForMultipleObjects( DWORD dwCount,  
CONST HANDLE* phHandles, BOOL bWaitAll, DWORD dwMilli-  
seconds) ;
```

The *dwCount* parameter specifies the number of kernel objects of interest. Its value must be between 1 and MAXIMUM_WAIT_OBJECTS (defined as 64 in Windows header files). The *phHcuidles* parameter is a pointer to an array of kernel object handles.

WaitForMultipleObjects suspends the thread and causes it to wait for all of the specified kernel objects or one of them to be released. The *bWaitAll* parameter specifies

what is required of the function. If it is TRUE, the function will not allow the thread to resume until all objects are freed.

The *dwMilliseconds* *parameter* is identical to the parameter of the same name in the *WaitForSingleObject* function.

The return value of the *WaitForMultipleObjects* function tells you why the thread that called it resumed execution.

Synchronizing Streams with Mutexes [4]

To use a mutex object, one of the processes must first create it by calling *CreateMutex*:

```
HANDLE CreateMutex( LPSECURITY_ATTRIBUTES psa, BOOL  
blInitialOwner, LPCTSTR pszName) ;
```

Any process can obtain its ("process-dependent") handle to an existing mutex object by calling *OpenMutex*:

```
HANDLE OpenMutex( DWORD fdwAccess, BOOL blnHeritHandle,  
PCTSTR pszName) ;
```

The */InitialOwner* *parameter* specifies the initial state of the mutex. If it passes FALSE (which is usually the case), the mutex object does not belong to any of the threads and is therefore in a free state.

43

tifier of the calling thread, and the recursion counter is set to 1. Because the thread ID is now different from 0, the mutex is initially in a busy state.

When the thread occupying the resource finishes working on it, it must release the mutex by calling the *ReleaseMutex* function:

```
BOOL ReleaseMutex(HANDLE hMutex) ;
```

This function decrements the recursion counter in the mutex object by 1. If the object has been transferred to the thread repeatedly, the thread must call `ReleaseMutex` as many times as necessary to reset the recursion counter. Once the counter is 0, the variable that stores the thread ID will also be zeroed and the mutex object will be freed. The system will then check to see if there are any other threads waiting for the mutex to be released. If so, the system will honestly select one of the waiting threads and transfer ownership of the mutex object to it.

Other kernel objects differ from a mutex object in that they are able to remember which thread they belong to. If some extraneous thread tries to free the mutex by calling the `ReleaseMutex` function, it will not do anything after checking the thread IDs and finding that they do not match, but will simply return `FALSE`.

An example of using a mutex is shown below. `HANDLE hti, hMutex; int x;`
`BOOL TermFlagl=FALSE;`

—Creating a mutex using the `CreateMutex` function —

```
hMutex=CreateMutex(
    NULL, FALSE,
    NULL);
```

—ThreadFunc1 thread function -----

```
DWORD WINAPI ThreadFunc1() {
while (! TermFlagl) { WaitForSingleObject(
hMutex, INFINITE);
```

—Code that works on a share

44

```
ReleaseMutex(hMutex);
```

```

        Sleep (1500); }
        return(0); }
}

```

– Creating a Flow -----

```

        DWORD          dwThread;
TermFlagl=FALSE;
        h      11=CreateThread(NULL,0,    (LPTHREAD_START_ROUTINE)
ThreadFunc1, (PVOID)&x, 0, &dwThread);

```

—Closing the mutex-----

```

CloseHandle(hMutex);

```

Synchronizing Threads with Semaphores [4]

Kernel semaphore objects are used to account for resources. Like all kernel objects, they contain a counter for the number of users, but they also support two signed 32-bit values: one for the maximum number of resources (controlled by the semaphore), and the other for the current number of resources.

The following rules are defined for semaphores:

- 1) When the counter of the current number of resources becomes greater than 0, the semaphore goes into a free state;
- 2) If this counter is 0, the semaphore is busy.
- 3) The system does not allow you to assign negative values to the counter of the current number of resources;
- 4) The current resource count cannot be greater than the maximum number of resources.

The kernel "semaphore" object is created by calling CreateSemaphore:

```

HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTE
    psa, LONG linitialCount, LONG lMAXimumCount,
    LPCTSTR pszName) ;

```

Any process can obtain its ("process-sensitive") handle to an existing semaphore object by calling `OpenSemaphore`.

```

HANDLE OpenSemaphore( DWORD fdwAccess,
    BOOL bInheritHandle, LPCTSTR pszName) ;

```

The `lMAXimumCount` *parameter* tells the system the maximum number of resources that the application can handle. Because this is a 32-bit signed value, the resource limit can be as high as 2,147,483,647.

A thread accesses a resource by calling one of the wait functions and passing it a semaphore handle that guards the resource. The wait function checks the semaphore for the counter of the current number of resources, if its value is greater than 0 (the semaphore is free), decrements the counter by 1, and the calling thread remains scheduled. Only after the resource counter is reduced by 1 can another thread request access to the resource.

If the wait function determines that the counter for the current number of resources is 0 (the semaphore is busy), the system puts the calling thread in a waiting state. When another thread increases the value of this counter, the system remembers the waiting thread and begins allocating CPU time to it again.

The thread increments the value of the current resource count counter by calling the `ReleaseSemaphore` function:

```

BOOL ReleaseSemaphore( HANDLE hSemaphore, LONG lRe-
    leaseCount, LPLONG lpPreviousCount) ;

```

ReleaseSemaphore simply adds the *IReleaseCount* value to the counter value of the current number of resources. Typically, the *IReleaseCount* parameter passes 1, but this is not necessary. The function returns the original value of the resource count to **lpPreviousCount*.

Example of semaphore
implementation: HANDLE htl,hSemaphore;
BOOL TermFlagl=FALSE;

—Create a semaphore using the CreateSemaphore function
hSemaphore=CreateSemaphore (
 NULL, 1/ Ir
 NULL);

—ThreadFunc1 thread function -----
DWORD WINAPI ThreadFunc1 () {

```

        while (!TermFlag1) {
            WaitForSingleObject (
                hSemaphore,
                INFINITE);

```

—Code that works on a share

```

ReleaseSemaphore(hSemaphore,1,NULL);
    Sleep (1500); }
    return ( 0); }

```

//— Creating a Flow

```

        DWORD dwThread; int x;
        TermFlag1=FALSE;
        htl=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE
) ThreadFunc1, (PVOID)&x, 0, &dwThread);

```

//—Close the semaphore

```

CloseHandle(hSemaphore);

```

1.2 Work procedure

- 1 To study the theoretical foundations of the implementation of the multithreading mechanism in operating systems and a set of system functions for working with threads in Windows.
- 2 Get an assignment from the teacher.
- 3 In the Builder C++ or MS Visual Studio environment, create an application and demonstrate its correct operation to the teacher.
- 4 Draw up a report on laboratory work.

1.3 Contents of the Report

- 1 Title of the work and purpose of the work.
- 2 Initial task.
- 3 Listing of the developed program.
- 4 Conclusions.

1.4 Control questions

- 1 How do I pause/resume a thread with Windows system tools?
- 2 Which system resources are mapped to a thread object?

- 3 What is the difference between a mutex object and a semaphore object in relation to the task of thread synchronization?
- 4 Is it possible to use a critical section to synchronize streams for different processes ?
- 5 How do I force a flow to end using Windows system tools?
- 6 What are the ways to synchronize streams in modern operating systems?
- 7 What is the functional difference between the objects "flow" and "process"?
- 8 What are the three basic states in which a thread can be?
- 9 Is it possible to influence the time that a single thread is queued for resources?

PRACTICAL WORK No 7.

Topic: Implementation of the SLAM algorithm

Objective: Familiarization and implementation of the SLAM algorithm

Develop a Visual SLAM Algorithm Using Unreal Engine Simulation

This example shows how to develop a visual Simultaneous Localization and Mapping (SLAM) algorithm using image data obtained from the Unreal Engine® Simulation Environment.

All examples and a detailed description are available at the link:

<https://docs.exponenta.ru/R2021b/driving/ug/develop-visual-slam-algorithm-using-unreal-engine-simulation.html>

Visual SLAM is the process of calculating the position and orientation of the camera relative to its environment, while simultaneously matching the environment. Developing a visual SLAM algorithm and evaluating its performance under different conditions is challenging. One of the most difficult tasks is generating the basic truth of the camera sensor, especially in outdoor environments.

This example demonstrates the use of the Unreal Engine Simulation to develop a visual SLAM algorithm for either a monocular or a stereo camera in a parking scenario. For more information on implementing visual SLAM piping, see [Monocular Visual Concurrent Localization and Mapping Example](#) and [Visual Concurrent Stereo Localization and Mapping Example](#).

Set up the scenario in the simulation environment

Use the Simulation 3D Scene Configuration Block to set up the simulation environment. Select the built-in Large Parking Scene that contains multiple parked vehicles. The SLAM visual algorithm matches the features through sequential images. To increase the number of potential feature matches, you can use the Parked Vehicle Subsystem to add more parked vehicles to the scene. To set the parking positions of vehicles, use the `helperAddParkedVehicle` function. If you choose a more natural scene, the presence of additional vehicles is not necessary. Natural scenes usually have enough structure and show a variety suitable for the matching function.

You can follow [Select example Waypoints for Unreal Engine Simulation](#) to interactively select a location parking sequence. You can use the same approach to select a sequence of waypoints and generate a reference path for a vehicle equipped with a sensor. This example uses a recorded reference path and parked vehicle locations.

```
% Load reference path
data = load('parkingLotReferenceData.mat');

% Set reference trajectory of the ego vehicle
refPosesX = data.refPosesX;
refPosesY = data.refPosesY;
refPosesT = data.refPosesT;

% Set poses of the parked vehicles
parkedPoses = data.parkedPoses;

% Display the reference path and the parked vehicle locations
sceneName = 'LargeParkingLot';
hScene = figure;
helperShowSceneImage(sceneName);
hold on
plot(refPosesX(:,2), refPosesY(:,2), 'LineWidth', 2, 'DisplayName', 'Reference Path');
scatter(parkedPoses(:,1), parkedPoses(:,2), [], 'filled', 'DisplayName', 'Parked Vehicles');
xlim([-60 40])
superior([10 60])
hScene.Position = [100, 100, 1000, 500]; % Resize figure
legend
hold off
```

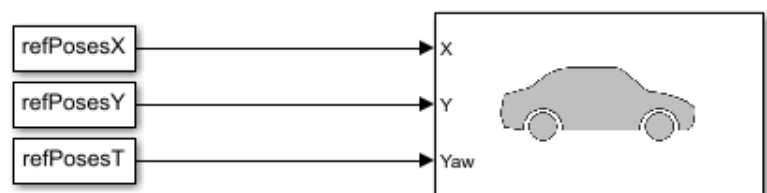
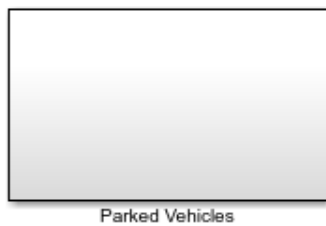
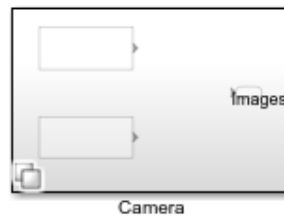
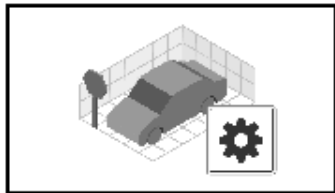


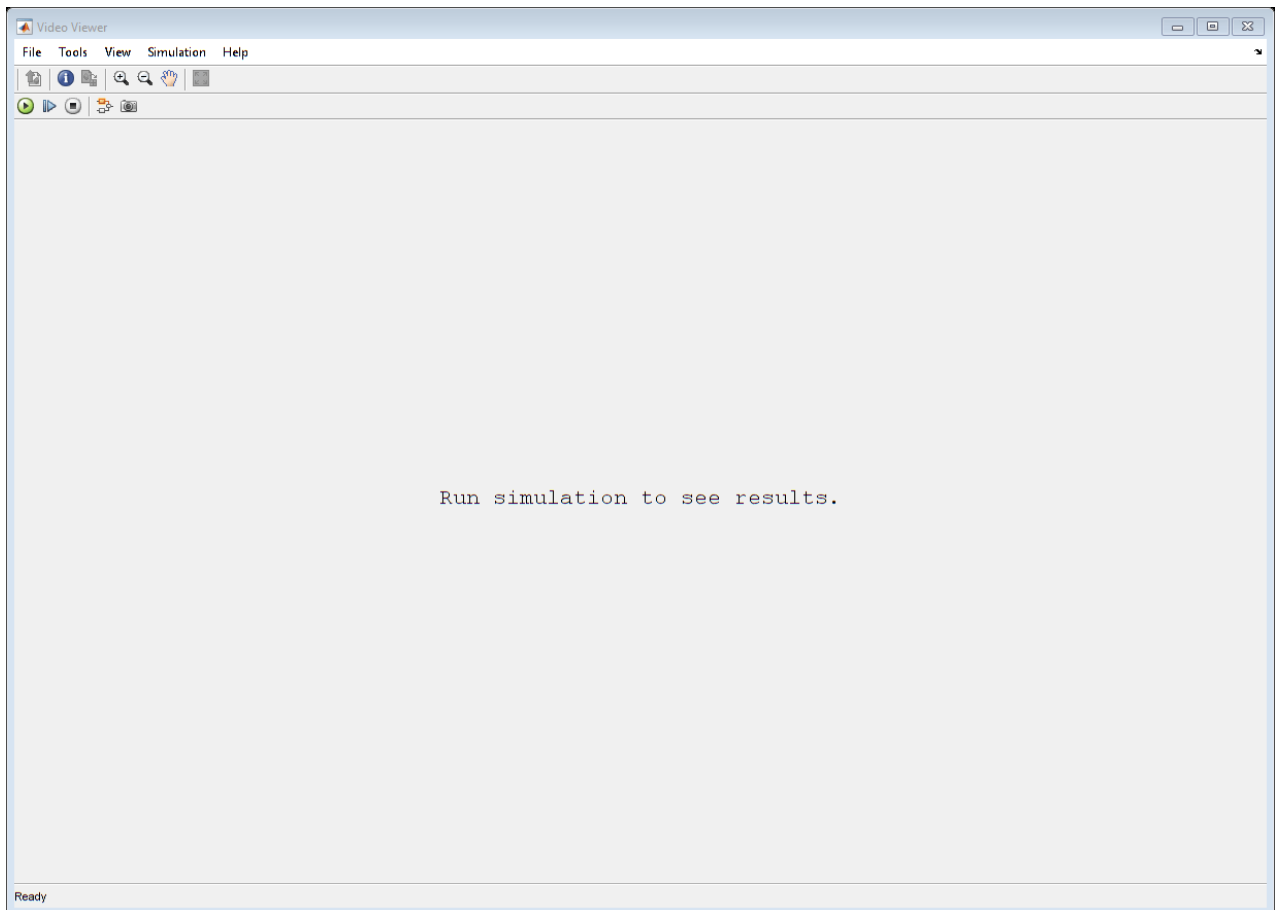
Open the model and add parked vehicles

```
modelName = 'GenerateImageDataOfParkingLot';
open_system(modelName);
snapnow;

helperAddParkedVehicles(modelName, parkedPoses);
```

Generate Image Data of Parking Lot





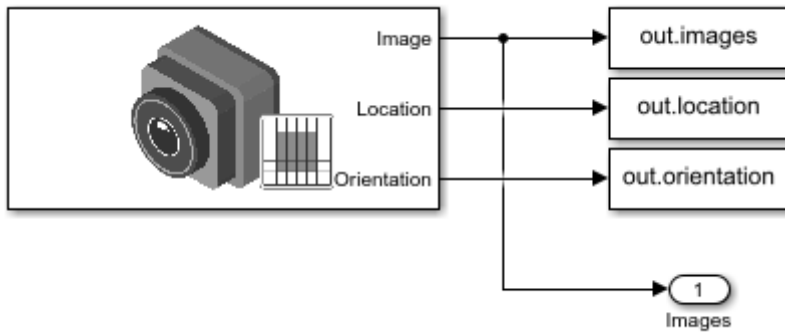
Set up the sensor of the car equipped with the sensor and the camera

Configure a vehicle equipped with a sensor to traverse a specified reference path using the 3D Vehicle Simulation with Ground Following block. The Camera Variant subsystem contains two camera sensor settings: monocular and stereo. In both settings, the camera is mounted on the center of the vehicle's roof. You can use [Camera Calibrator](#) or app [Stereo Camera Calibrator](#) to evaluate the internal parameters of the actual camera you want to simulate. This example shows a monocular camera workflow first followed by a stereo camera workflow.

```
% Select monocular camera
useMonoCamera = 1;

% Inspect the monocular camera
open_system([modelName, '/Camera/Monocular']);
snapnow;

% Camera intrinsics
focalLength      = [700, 700]; % specified in units of pixels
principalPoint   = [600, 180]; % in pixels [x, y]
imageSize        = [370, 1230]; % in pixels [mrows, ncols]
intrinsics        = cameraIntrinsics(focalLength, principalPoint, imageSize);
```



Visualize and record sensor data

Run the simulation to visualize and record sensor data. Use the Video Viewer block to visualize the image of the camera sensor exit. Use the To Workspace block to record the location of the primary truth and the orientation of the camera sensor.

```
close(hScene)
```

```
if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + "Windows" + char(174) + ".");
end
```

```
% Run simulation
```

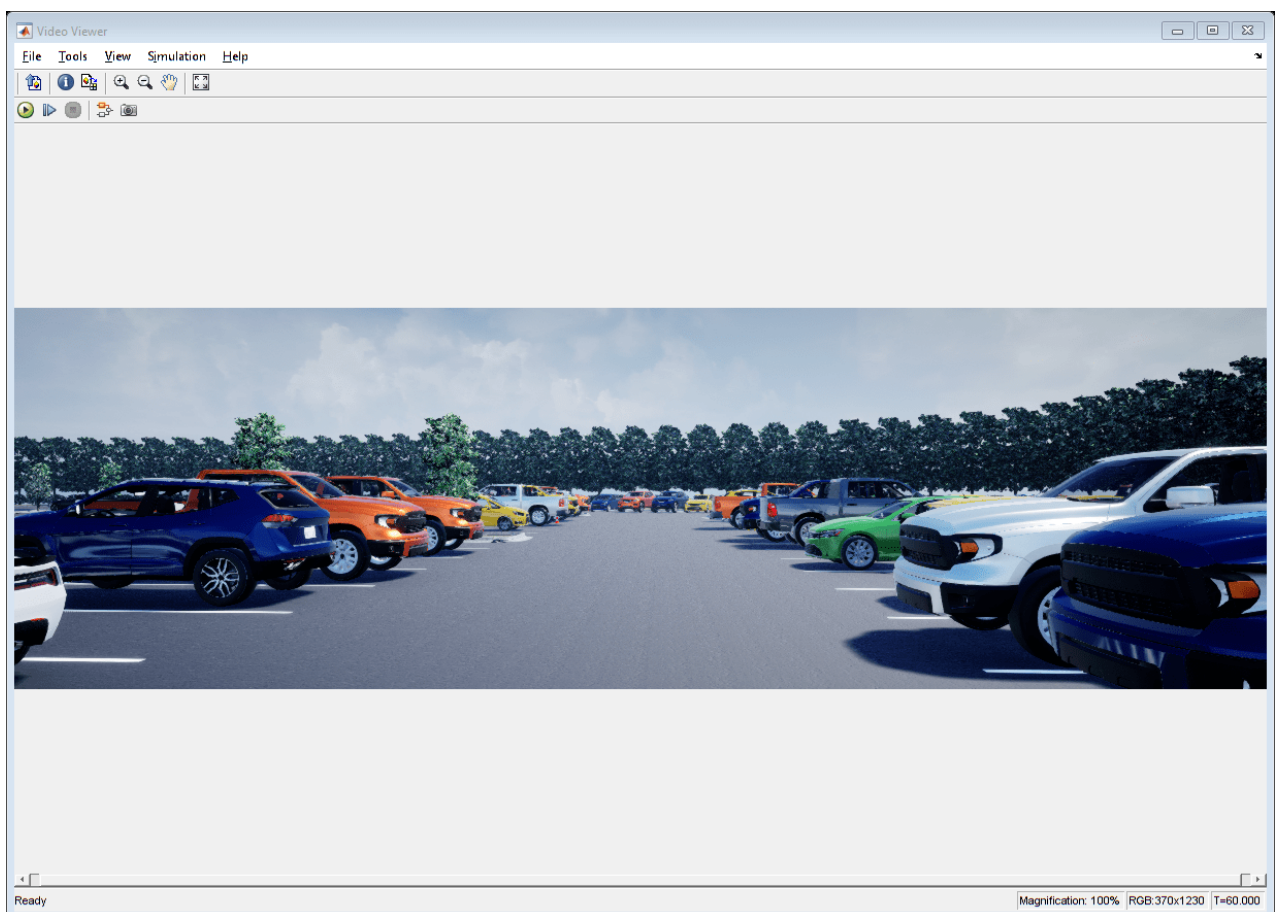
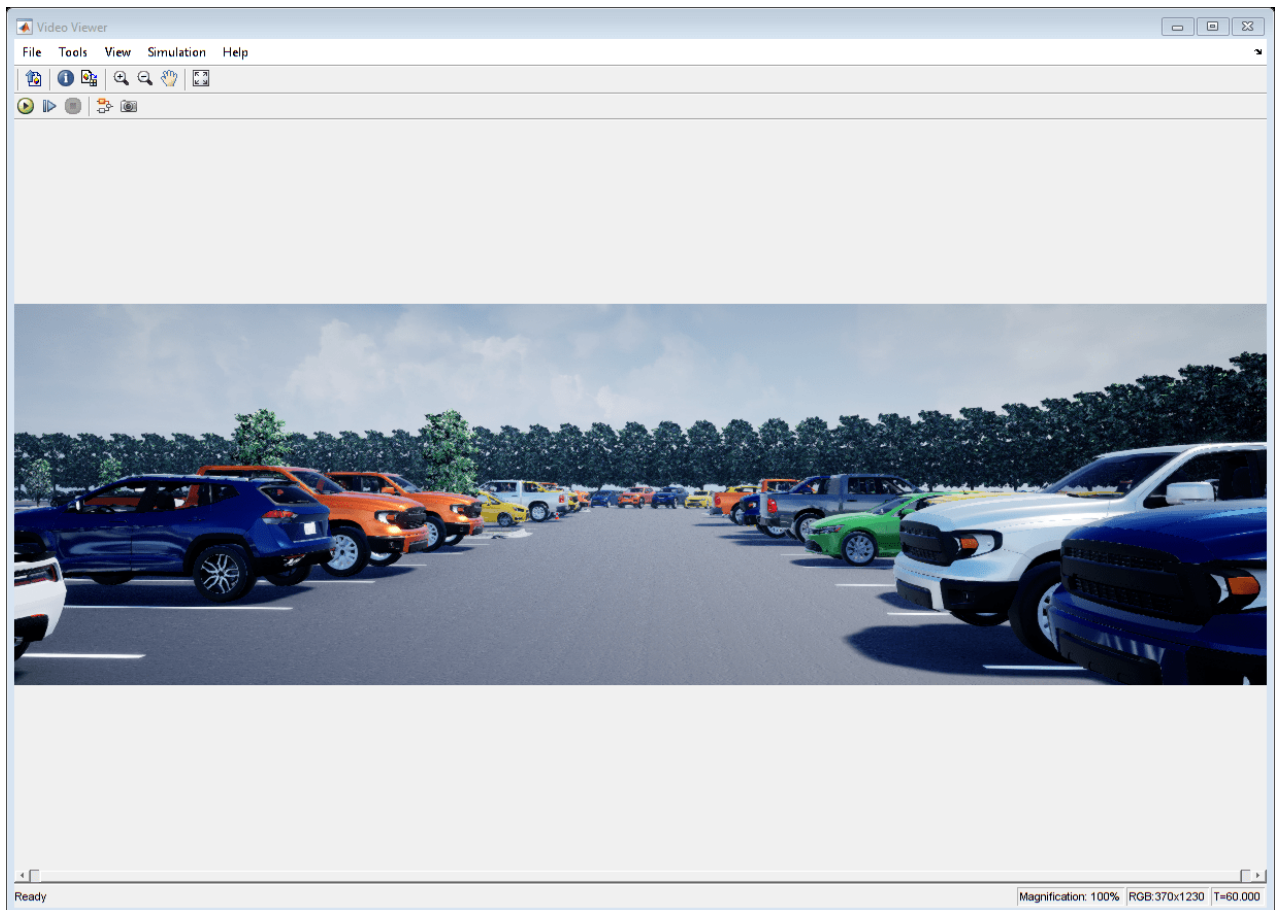
```
simOut = sim(modelName);
snapnow;
```

```
% Extract camera images as an imageDatastore
```

```
imds = helperGetCameraImages(simOut);
```

```
% Extract ground truth as an array of rigid3d objects
```

```
gTruth = helperGetSensorGroundTruth(simOut);
```



Develop a Monocular Visual SLAM Algorithm Using Recorded Data

Use the images to evaluate the monocular SLAM visual algorithm. The functional helper `VisualSLAM` implements the ORB-SLAM monocular piping:

Card initialization: ORB-SLAM is triggered by initializing a 3-D point map from two images. Use `relativeCameraPose` to compute relative position based on 2D ORBs show correspondences and `triangulate` calculate the 3-D points of the map. The two coordinate systems are stored in the `imageviewset` objects as keyframes. The 3-D map points and their correspondence to keyframes are stored in the `worldpointset` object.

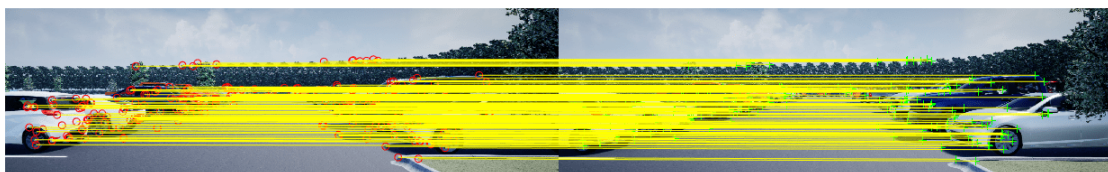
Tracking: If the map is initialized, for each new image, the functional `helperTrackLastKeyFrame` evaluates the position of the camera by matching the functions in the current coordinate system to the functions in the last keyframe. The functional `helperTrackLocalMap` improved the intended position of the camera by tracking the local map.

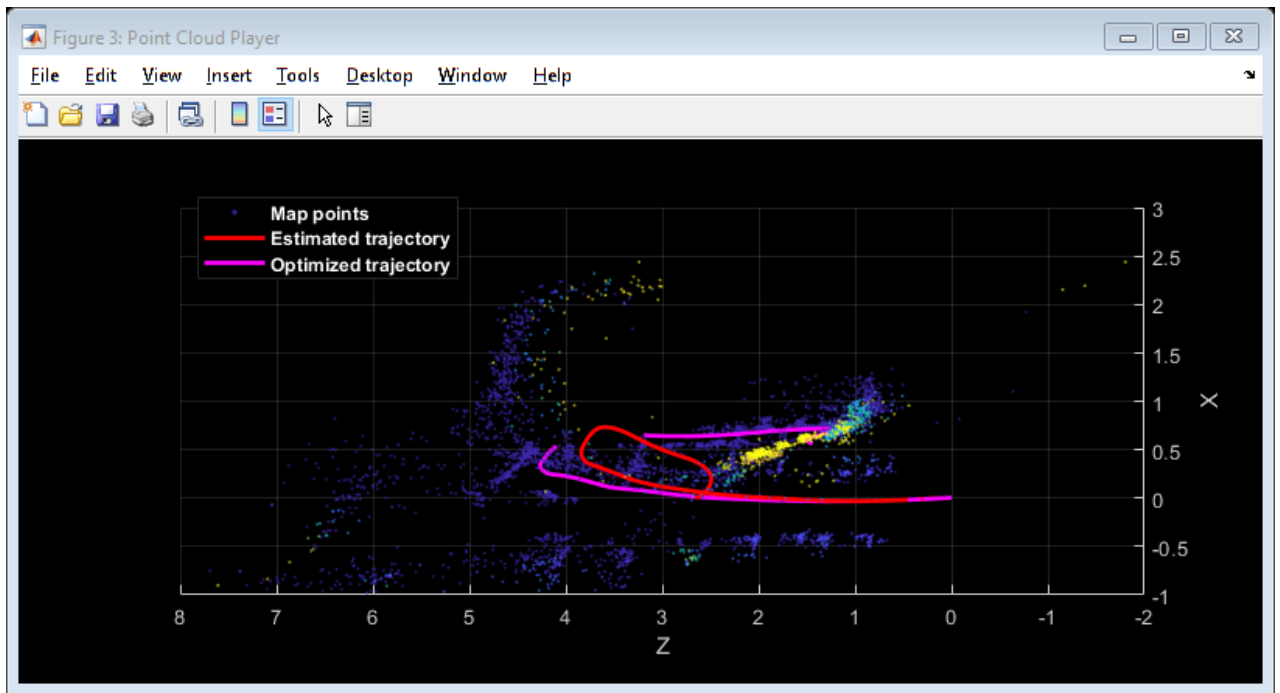
Local Display: The current coordinate system is used to create new 3-D map points if it is identified as a keyframe. At this point, `bundleAdjustment` is used to minimize reprojection errors by adjusting the camera position and 3-D points.

Closing the cycle: Loops are detected for each keyframe by comparing it against all previous keyframes using a feature set approach. If loop closure is detected, the position graph is optimized to improve the camera positions of all keyframes with `optimizePoseGraph` (Navigation Toolbox) function.

For more information about implementing the algorithm, see [Monocular Visual Concurrent Localization and Mapping Example](#).

```
[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAM(imds, intrinsics);  
Map initialized with frame 1 and frame 3  
Loop edge added between keyframe: 7 and 156
```



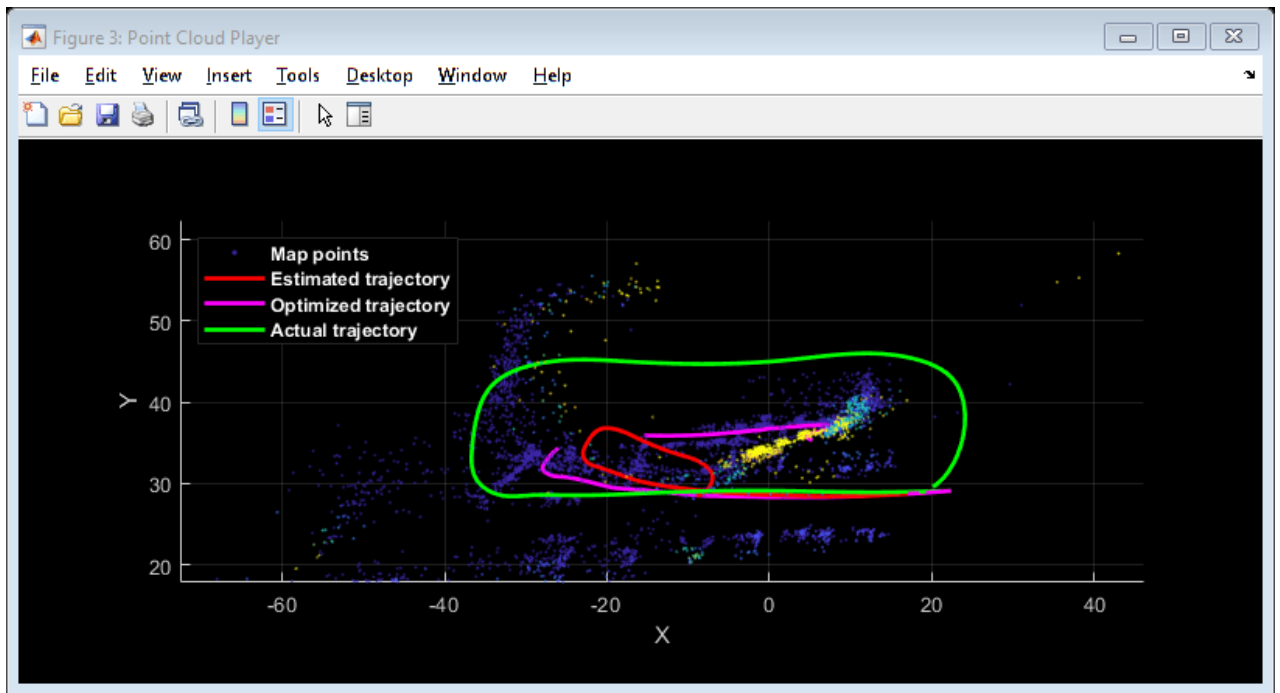


Evaluate against the basic truth

It is possible to evaluate the optimized camera trajectory against the basic truth derived from the simulation. Since the images are generated from a monocular camera, the camera trajectory can only be reconstructed to an unknown scale factor. It is possible to approximate the scale factor from the underlying truth, thus simulating what you would normally get from an external sensor.

```
% Plot the camera ground truth trajectory
scaledTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx),
optimizedPoses);

% Show legend
showLegend(mapPlot);
```



You can also calculate the root mean square error (RMSE) of the path estimates.

```
helperEstimateTrajectoryError(gTruth(addedFramesIdx), scaledTrajectory);
Absolute RMSE for key frame trajectory (m): 11.8389
```

Stereo SLAM Visual Algorithm

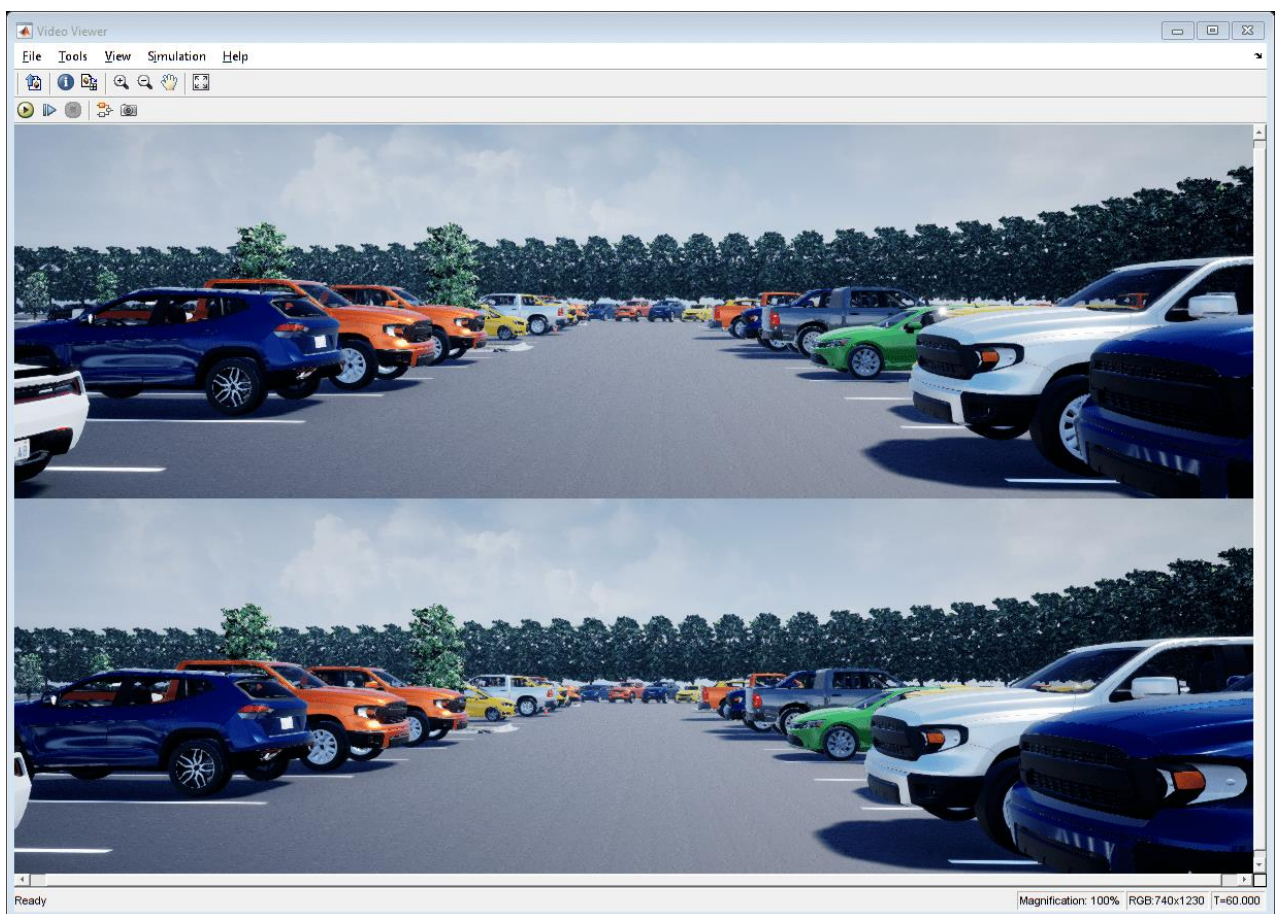
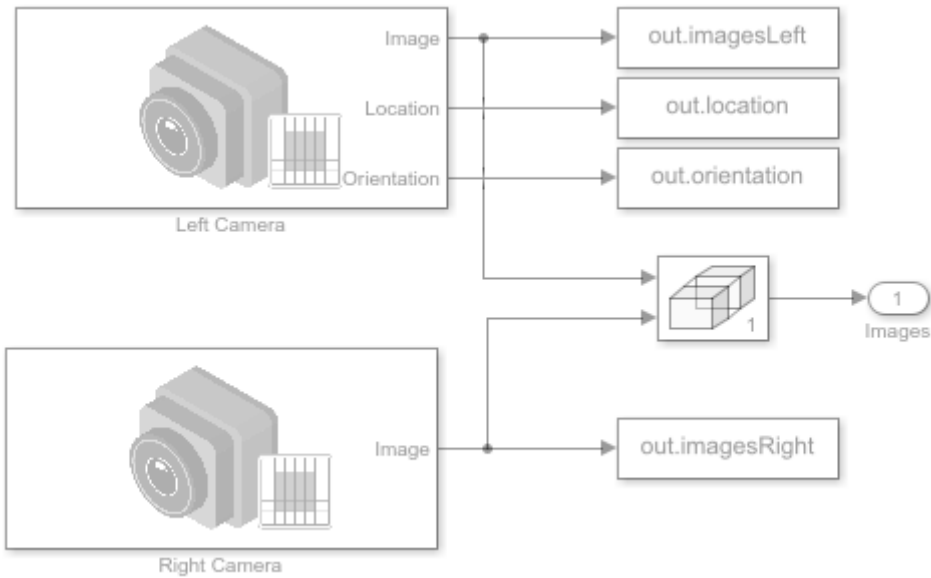
In a monocular visual SLAM algorithm, depth cannot be accurately determined with a single camera. The scale of the map and the intended trajectory is unknown and drifts with time. In addition, because map points often cannot triangulate from the first coordinate system, loading the system requires multiple representations to produce the initial map. Using a stereo camera solves these problems and provides a more reliable visual SLAM solution. Functional `helperVisualSLAMStereo` implements a visual SLAM stereo piping. The main difference from monocular piping during the map initialization phase is that the stereo piping creates 3-D map points from a pair of stereo images of the same coordinate system, instead of creating them from two images of different coordinate systems.

```
% Select stereo camera
useMonoCamera = 0;

% Inspect the stereo camera
open_system([modelName, '/Camera/Stereo']);
snapnow;

% Set stereo camera baseline
baseline = 0.5; % In meters

% Run simulation
simOut = sim(modelName);
snapnow;
```



Extract stereo images

```
[imdsLeft, imdsRight] = helperGetCameraImagesStereo(simOut);

% Extract ground truth as an array of rigid3d objects
gTruth = helperGetSensorGroundTruth(simOut);
Run the Stereo SLAM Visual Algorithm

[mapPlot, optimizedPoses, addedFramesIdx] = helperVisualSLAMStereo(imdsLeft,
imdsRight, intrinsics, baseline);

% Plot the camera ground truth trajectory
```

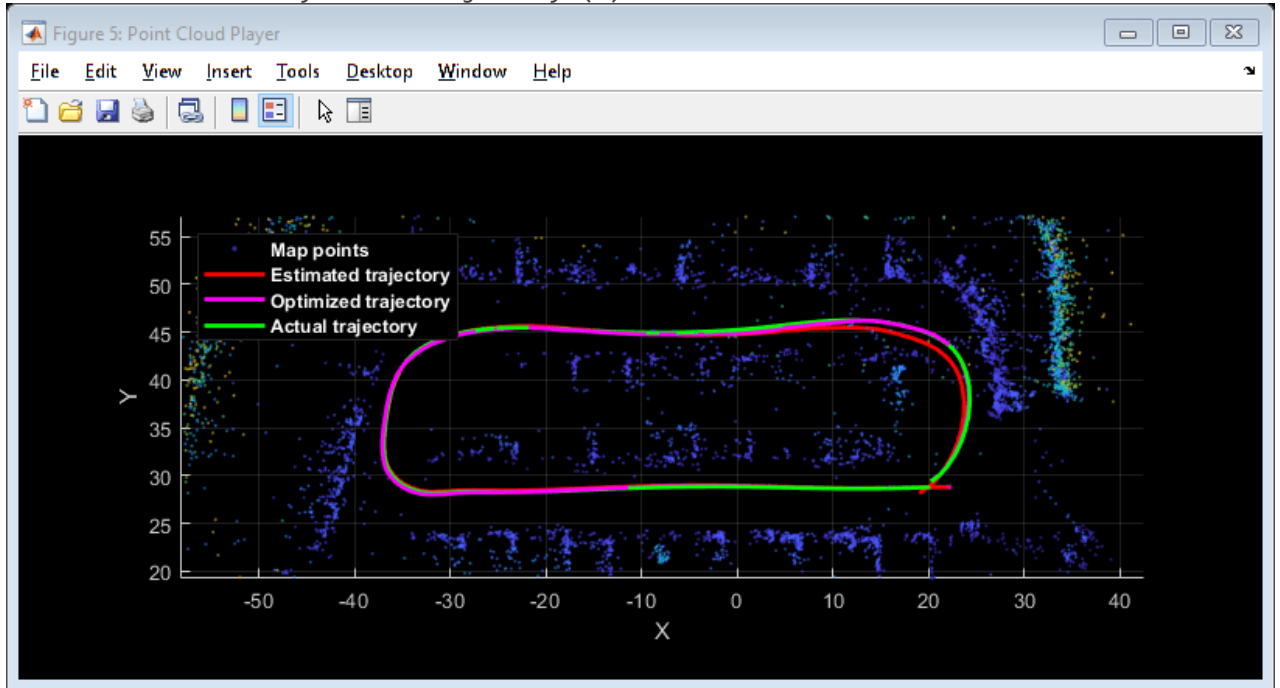
```

optimizedTrajectory = plotActualTrajectory(mapPlot, gTruth(addedFramesIdx));

% Show legend
showLegend(mapPlot);

% Calculate the root mean square error (RMSE) of trajectory estimates
helperEstimateTrajectoryError(gTruth(addedFramesIdx), optimizedTrajectory);
Loop edge added between keyframe: 5 and 145
Absolute RMSE for key frame trajectory (m): 0.25164

```



Compared to the monocular SLAM visual algorithm, the SLAM stereo visual algorithm produces a more accurate estimate of the camera trajectory.

Close the model and shapes.

```

close_system(modelName, 0);
close all

```

Helper Functions

helperGetCameraImages Output the camera

```

function imds = helperGetCameraImages(simOut)
% Save image data to a temporary folder
dataFolder = fullfile(tempdir, 'parkingLotImages', filesep);
folderExists = exist(dataFolder, 'dir');
if ~folderExists
    mkdir(dataFolder);

```

```

end

files = dir(dataFolder);
if numel(files) < 3
    numFrames = numel(simOut.images.Time);
    for i = 3:numFrames % Ignore the first two frames
        img = squeeze(simOut.images.Data(:,:,i));
        imwrite(img, [dataFolder, sprintf('%04d', i-2), '.png'])
    end
end

% Create an imageDatastore object to store all the images
imds = imageDatastore(dataFolder);
end

helperGetCameraImagesStereo Output Camera

function [imdsLeft, imdsRight] = helperGetCameraImagesStereo(simOut)
% Save image data to a temporary folder
dataFolderLeft = fullfile(tempdir, 'parkingLotStereoImages', filesep, 'left',
filesep);
dataFolderRight = fullfile(tempdir, 'parkingLotStereoImages', filesep, 'right',
filesep);
folderExists = exist(dataFolderLeft, 'dir');
if ~folderExists
    mkdir(dataFolderLeft);
    mkdir(dataFolderRight);
end

files = dir(dataFolderLeft);
if numel(files) < 3
    numFrames = numel(simOut.imagesLeft.Time);
    for i = 3:numFrames % Ignore the first two frames
        imgLeft = squeeze(simOut.imagesLeft.Data(:,:,i));
        imwrite(imgLeft, [dataFolderLeft, sprintf('%04d', i-2), '.png'])

        imgRight = squeeze(simOut.imagesRight.Data(:,:,i));
        imwrite(imgRight, [dataFolderRight, sprintf('%04d', i-2), '.png'])
    end
end

% Use imageDatastore objects to store the stereo images
imdsLeft = imageDatastore(dataFolderLeft);
imdsRight = imageDatastore(dataFolderRight);
end

helperGetSensorGroundTruth Stores the basic truth of the sensor

function gTruth = helperGetSensorGroundTruth(simOut)
numFrames = numel(simOut.location.Time);
gTruth = repmat(rigid3d, numFrames-2, 1);
for i = 1:numFrames-2 % Ignore the first two frames
    gTruth(i). Translation = squeeze(simOut.location.Data(:, :, i+2));
    % Ignore the roll and the pitch rotations since the ground is flat
    yaw = double(simOut.orientation.Data(:, 3, i+2));
    gTruth(i). Rotation = [cos(yaw), sin(yaw), 0; ...
        -sin(yaw), cos(yaw), 0; ...
        0, 0, 1];
end
end

helperEstimateTrajectoryError Calculate tracking error

function rmse = helperEstimateTrajectoryError(gTruth, scaledLocations)
gLocations = vertcat(gTruth.Translation);

```

```
rmse = sqrt(mean( sum((scaledLocations - gLocations).^2, 2) ));  
disp(['Absolute RMSE for key frame trajectory (m): ', num2str(rmse)]);  
end
```

PRACTICAL WORK No 8.

Topic: Introduction to YOLO architecture for object detection.

Objective: to familiarize with the methods used for computer vision of robots.

Getting the job done

Get a NNMA_YOLO_Lab file from your instructor. Run it in Jupiter notebook and demonstrate the results of processing the video file.

A healthy file is a lab report.

PRACTICAL WORK No 9.

Topic: Algorithm optimization using a profiler

Objective: To get acquainted with the tools for profiling program code.

When developing applications, developers periodically have a question - why is the implemented code slow, and where valuable time and resources are wasted? This is where profiling tools come to the rescue. In this post, we will consider the possibilities of profiling applications implemented in Python using various profiling tools, as well as consider tools that allow you to build a graphical representation of profiling results and create reports.

Profiling as a concept

To identify places where the application is wasting resources, we need the following information:

- The order in which the code is executed (which functions are called in what order)
- the number of calls to individual code fragments (functions/procedures/methods, etc.);
- Time spent executing a single piece of code (function/method/procedure, etc.)

With the above information, we can identify bottlenecks in our project, and by concentrating on a single piece of code, refactor it to improve the efficiency of the application as a whole.

Profile: A set of statistics that describes how often and for how long different parts of a program were executed.

Python Profiling Tools

cProfile и profile

Python has built-in tools for profiling, which can be found in the `cProfile` and `profile` modules. These modules allow you to obtain statistics that can later be converted into a report using the `pstats` module or other tools. The `cProfile` module is implemented as a C extension with a low overhead and is recommended for use as the main one.

Let's create a small Python application that calls a few functions and outputs messages to the console, and place the code in the *test.py* module

```
def print_message(message):
    print(message)

def range_output(external_step, internal_steps_number):
    for internal_step in range(internal_steps_number):
        print_message(f'[{external_step}: {internal_step}]')

def main():
    for i in range(3):
        range_output(external_step=i, internal_steps_number=2)

if __name__ == '__main__':
    main()
```

Let's profile this code using cProfile:

```
python -m cProfile test.py
[0: 0]
[0: 1]
[1: 0]
[1: 1]
[2: 0]
[2: 1]
      19 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.000   0.000   0.000   0.000 test.py:10(main)
      1   0.000   0.000   0.000   0.000 test.py:3(<module>)
      6   0.000   0.000   0.000   0.000 test.py:3(print_message)
      3   0.000   0.000   0.000   0.000 test.py:6(range_output)
      1   0.000   0.000   0.000   0.000 {built-in method builtins.exec}
      6   0.000   0.000   0.000   0.000 {built-in method builtins.print}
      1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof. Profiler'
objects}
```

As a result, we received the information we were interested in:

- ncalls - number of function calls;
- tottime - the total time spent in this function (excluding the time spent on calling subfunctions);
- percall - time spent on 1 function call (tottime / ncalls);
- cumtime is the total time spent in this function, including the time spent calling all subfunctions (from call to exit). This value is also true for recursive functions;
- percall - time spent on 1 function call (cumtime / ncalls);
- filename:lineno(function) - <filename>:<No strings>(<function name>)

We can also create a `pstats` file for use as data for graphical reporting tools (discussed below), which is implemented through the `-o` flag with a file name with the `.pstats` extension, for example, `cprofile_output.pstats`:

```
python -m cProfile -o cprofile_output.pstats test.py
```

Yappi

Library `Yappi` (opens new window) ("Yet Another Python Profiler") - has more functionality than the built-in Python tools. `Yappi` implemented as a C library is fast, allows you to profile multithreaded applications, applications that use `asyncio` (opens new window) and `Giving` (opens new window), allows you to display the results in the `formatcallgrind` (opens new window) and `pstat` (opens new window), the results of the profiler's work can display `Wall time` (opens new window) or `CPU Time` (opens new window) and can be aggregated from different sessions. In addition, when using `PyCharm`, the `Yappi` profiler is used by default in this IDE.

Installing `yappi` is as follows:

```
pip install yappi
```

Now let's change our code a bit so that `yappi` can analyze it and collect statistics in `pstat` and `callgrind` formats:

```
import yappi

def print_message(message):
    print(message)

def range_output(external_step, internal_steps_number):
    for internal_step in range(internal_steps_number):
        print_message(f'[{external_step}: {internal_step}]')

def main():
    for i in range(3):
        range_output(external_step=i, internal_steps_number=2)

if __name__ == '__main__':
    yappi.start() # This is where the monitoring of the running code begins
    main()
    func_stats = yappi.get_func_stats() # Collect statistics of the functions
    performed
    func_stats.print_all() # Display the collected statistics in the console
    func_stats.save('yappi_output.pstats', 'PSTAT') # Сохраняем в формате pstat
    func_stats.save('yappi_output.callgrind', 'CALLGRIND') # Save in callgrind format
```

```
yappi.stop() # Stop surveillance
yappi.clear_stats() # Clearing statistics
```

Running this code will create *an output.pstats* file that can be handled using the `gprof2dot` previously discussed, and *an output.callgrind* file in callgrind format (see how to use it below), and output the profiling results to the console:

```
python test.py
```

```
Clock type: CPU
```

```
Ordered by: totaltime, desc
```

```
Name ncall tsub ttot tavg
```

```
.. ver/MarketAnalyser/test.py:10 main 1      0.000017 0.000141 0.000141
```

```
.. ketAnalyser/test.py:6 range_output 3      0.000029 0.000125 0.000042
```

```
.. andAnalyze/test.py:3 print_message 6 0.000028 0.000095 0.000016
```

Here you can see:

- Clock type: CPU – Indicates that the profiling time statistics shown are retrieved using the CPU clock speed. This means that the actual CPU time spent on the function is displayed.
- Ordered by: totaltime, desc - shows the parameter by which the sorting was performed and its order, in our case the sorting was performed by total time (totalitme) in descending order
- name - the full unique name of the called function
- ncall - the number of calls to the function
- tsub - the total time spent by the function, without additional calls
- ttot is the total time elapsed by the function, including helper calls
- tavg is the average time elapsed by the function, including helper calls.

The main advantage `yappi` is an implementation of profiling in multithreaded applications and coroutines. For more information and examples of implementations, see [GitHub page of the project](#) (opens new window).

Building reports on the results of profiling

The information obtained with the help of the above profiling tools is very useful, but its output is not very convenient for analysis, especially in the case of a real application, where the number of code fragments and their calls will be huge. For this reason, the help of additional tools that allow you to analyze this information graphically is often required.

gprof2dot

The `gprof2dot` library (opens new window) allows you to convert the output of profiling results in `pstat` format into a convenient graph of function calls, which is for the final application.

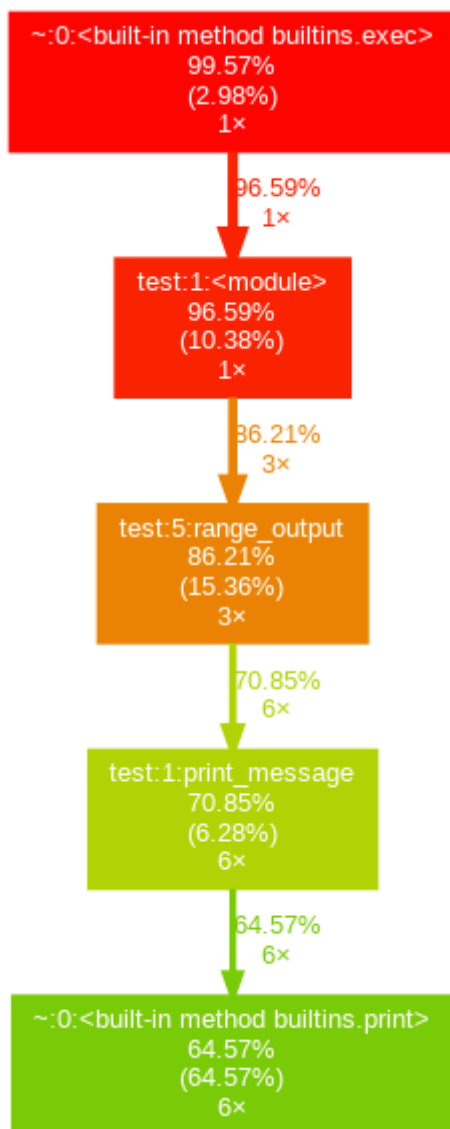
`gprof2dot` uses the `graphviz` library to build graphics. To install on Linux, you need to run the following commands:

```
sudo apt install graphviz
pip install gprof2dot
```

Now let's display the call graphs we created earlier with `gprof2dot`, which will be saved in `png` format:

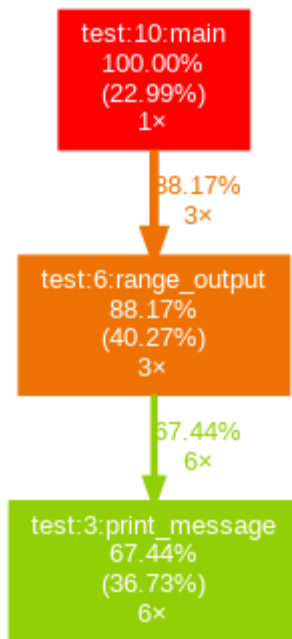
```
gprof2dot -f pstats cprofile_output.pstats | dot -Tpng -o cprofile_output.png
gprof2dot -f pstats yappi_output.pstats | dot -Tpng -o yappi_output.png
```

Let's open the image `cprofile_output.png`:



Here we can clearly see that *86.21%* of the execution time was taken by the execution of the `range_output` function, and this method was called 3 times, and during this time it called the `print_message` method 6 times.

The graph resulting from `yappi` looks a bit shorter because the record is made from the point specified in the code:



Here, similar to the `cProfile` example, we see a graph of function calls from the beginning of the `yappi.start()` record to the completion of `yappi.stop()`.

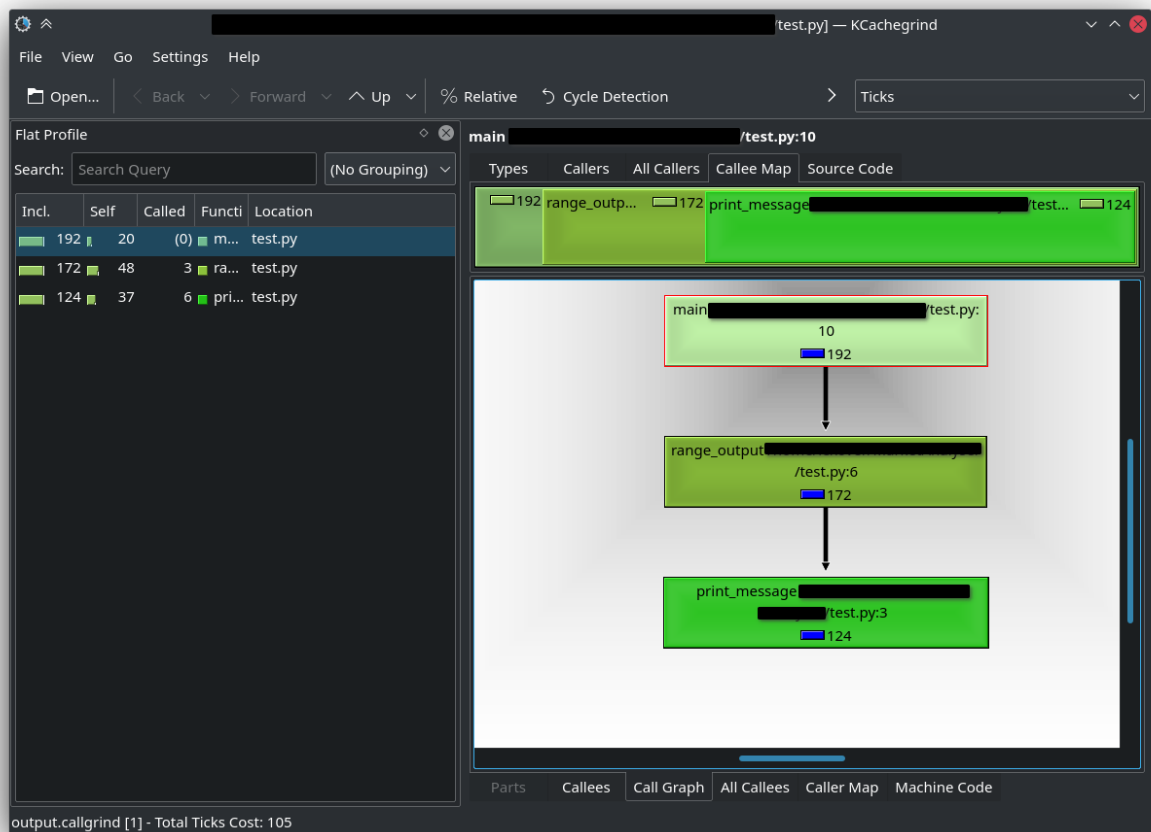
As we see the `toolgprof2dot` allows us to build graphical representations of the sequences of calls to our application's functions, indicating the number of calls and the resources spent. But `yappi` also allows you to output grading results in the format `valgrind` which is called `callgrind` (opens new window). Accordingly, this format can be opened by third-party applications that can work with them, one of these is `KCachegrind`.

KCachegrind

By creating a `callgrind` file, we can use advanced tools to view the results of the `KCachegrind` (opens new window) profiling results. In a Debian environment, it can be installed:

```
sudo apt install kcache
```

After installation, launch the `kcache` application and open the `outpul.callgrind` file we created and get a detailed interactive report:



PyCharm Profile

Also, if you are using the PyCharm Pro IDE, this IDE uses Yappi as a profiler by default and has built-in tools for graphical display of profiling results, and when you run profiling in the IDE, we get a report in the form of a table and a graphical representation:

Statistics		Call Graph		
Name	Call Count	Time (ms)		Own Time (ms) ▾
test.py	1	0	0,0 %	0 0,0 %
main	1	0	0,0 %	0 0,0 %
range_output	3	0	0,0 %	0 0,0 %
print_message	6	0	0,0 %	0 0,0 %

print_message times;6Total: **0ms** 0,0 %Own: **0ms** 0,0 %**range_output times;3**Total: **0ms** 0,0 %Own: **0ms** 0,0 %**main times;1**Total: **0ms** 0,0 %Own: **0ms** 0,0 %**test.py times;1**Total: **0ms** 0,0 %Own: **0ms** 0,0 %