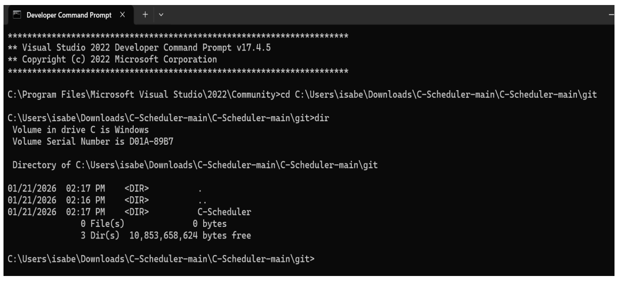
Implementierung des Round Robin CPU-Schedulers

Kline-Pearson, Mika (376064)
Garron Pereira, Isabella (382401)

Kline-Pearson, Mika (376064)
Garron Pereira, Isabella (382401)

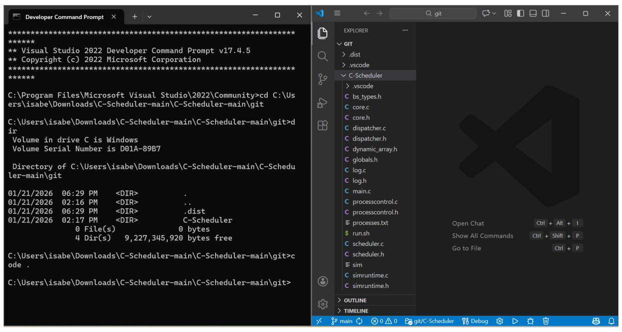
GETTING STARTED

1. Open Visual Studio Developer Command Prompt
2. Using the `cd` command, navigate to the root directory of this project.
   You can use the dir command to verify that you are in the correct directory.



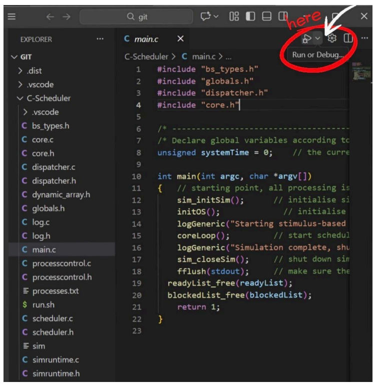
```
**********************************************************************
** Visual Studio 2022 Developer Command Prompt v17.4.5
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************

C:\Program Files\Microsoft Visual Studio\2022\Community>cd C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git

C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git>dir
 Volume in drive C is Windows
 Volume Serial Number is D01A-89B7

 Directory of C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git

01/21/2026  02:17 PM    <DIR>          .
01/21/2026  02:16 PM    <DIR>          ..
01/21/2026  02:17 PM    <DIR>          C-Scheduler
               0 File(s)              0 bytes
               3 Dir(s)  10,853,658,624 bytes free

C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git>
```

3. In that Command Prompt, enter `code .` A VSC window opens.



```
**********************************************************************
******
** Visual Studio 2022 Developer Command Prompt v17.4.5
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************
******

C:\Program Files\Microsoft Visual Studio\2022\Community>cd C:\Us
ers\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git

C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git>d
ir
 Volume in drive C is Windows
 Volume Serial Number is D01A-89B7

 Directory of C:\Users\isabe\Downloads\C-Scheduler-main\C-Schedu
ler-main\git

01/21/2026  06:29 PM    <DIR>          .
01/21/2026  02:16 PM    <DIR>          ..
01/21/2026  06:29 PM    <DIR>          .dist
01/21/2026  02:17 PM    <DIR>          C-Scheduler
               0 File(s)              0 bytes
               4 Dir(s)   9,227,345,920 bytes free

C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git>c
ode .

C:\Users\isabe\Downloads\C-Scheduler-main\C-Scheduler-main\git>
```

4. In Visual Studio Code, navigate to the main.c file. Click "Run and Debug".



```c
#include "bs_types.h"
#include "globals.h"
#include "dispatcher.h"
#include "core.h"

/* -------------------------------------
/* Declare global variables according to
unsigned systemTime = 0;    // the curre

int main(int argc, char *argv[])
{   // starting point, all processing is
    sim_initSim();      // initialise si
    initOS();            // initialise
    logGeneric("Starting stimulus-based
    coreLoop();         // start schedul
    logGeneric("Simulation complete, shu
    sim_closeSim();     // shut down sim
    fflush(stdout);     // make sure the
  readyList_free(readyList);
  blockedList_free(blockedList);
    return 1;
}
```

5. A terminal emulator window opens within Visual Studio Code, compiling the program to a file named `scheduling.exe`.
   You can use the `dir` command to verify that it has been created.
6. Enter `.\scheduling.exe`. The program runs, and the output is visible in the Visual Studio Code terminal emulator's standard output.

Kline-Pearson, Mika (376064)
Garron Pereira, Isabella (382401)

DESIGNREPORT

Round Robin

Data Types

Dynamically allocated Array

The only data type used is a dynamically allocated array (dynamic_array.h). This is the struct for both readyList as well as blockedList, and as such, a single C macro was used to implement it for each.

A video was used as a tutorial for the base concepts, and our implementation expands further on the tutorial's delivered concept (*"Dynamic Arrays in C", Channel: Tsoding, link: https://www.youtube.com/watch?v=95M6V3mZgrI*).
The Array is a struct with a count (tracking the number of elements) and a capacity (the max size that has been allocated). It also has a pointer to a contiguous block of memory (*elems), which holds all the elements of the array.

We chose to use this data type over the choice of a Linked List, because the array's contiguity assists in preventing cache misses, which is crucial for smaller Quantum sizes. It is dynamically allocated due to the unknown number of possible processes to schedule; if we were to fix the size of the array, scheduling more than that number of processes would be impossible. The Array therefore reallocates itself if it ever goes over capacity.

The following standard functions are provided for this data type:

Signature	Usage
`init(void)`	Initializes array with count=0; capacity=256; Allocates space for its elements.
`realloc(array_t)`	Doubles the capacity of the array and reallocates its elements
`isEmpty(Boolean)`	Returns TRUE if count is greater than 0, else FALSE
`free(void)`	Frees the allocated space, and itself. Since the scheduler is constantly running, this is only used at end of simulation.

And two functions to add and remove elements:

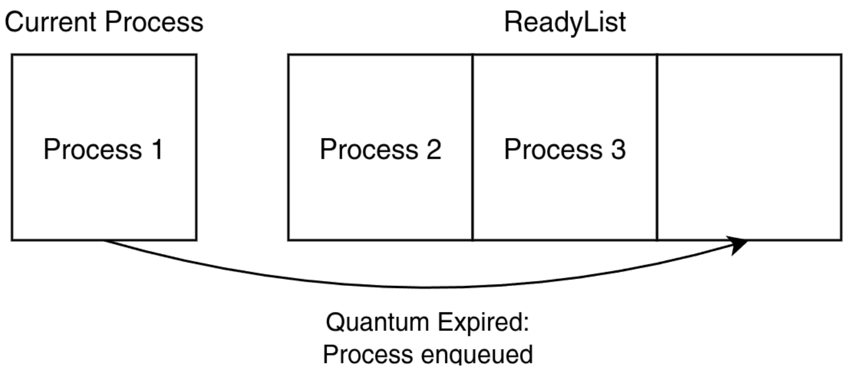
Signature	Usage
`push (array_t, element_t)`	Appends an element. Reallocates if it would go over capacity. Due to the assignment's requirements, the blockedList does not use push(), but instead inserts at the required index (sorted by IOready).
`remove_by_pid (array_t, pid_t)`	Removes an element from the array with a given pid. The following elements are moved back to preserve order.

Whereby remove_by_pid assumes that the elements are structs with a pid field; since both the readyElements and blockedElements are effectively wrappers over a process id (and are the only elements used in the scheduler), we believe that this is a fair design concession.

Implementation of the Queues

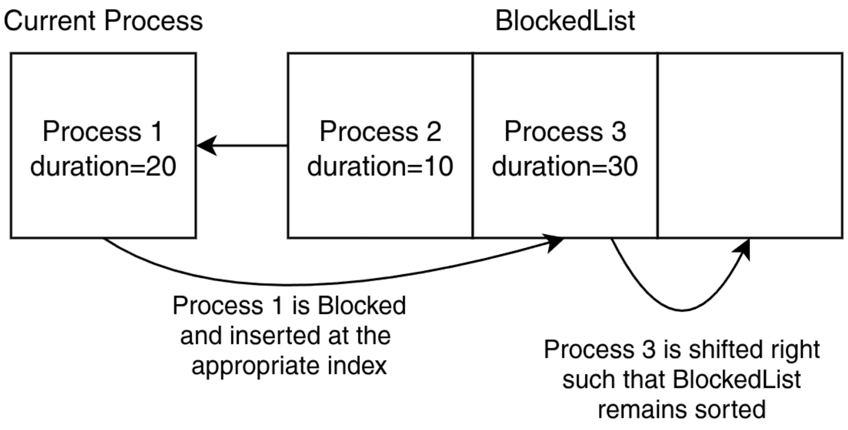
ReadyList

The ReadyList contains the provided ReadyListElements.
Since elements are appended to the end and removed from the head, it functions purely as a FIFO queue. The Round Robin scheduler will then simply need to deliver the head of the queue. This system prevents starvation by never having a possible state in which processes are permanently waiting; if a process enters the ReadyList, it will inevitably exit the list in a definite amount of time (namely, its place in the list multiplied by the quantum constant).



BlockedList

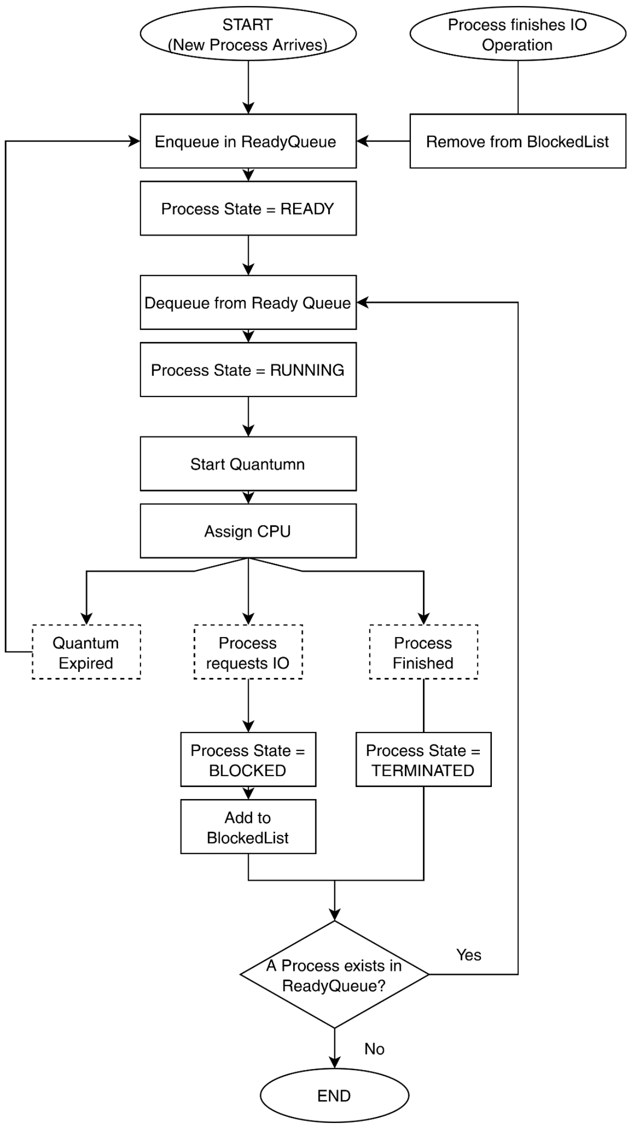
The BlockedList contains the provided BlockedListElements.
Due to the requirements of this implementation, it must be sorted by IOready. This is achieved in addBlocked() by traversing the list, inserting at the first instance of an element with a shorter blockDuration, and shifting all following elements up 1 spot. The BlockedList is therefore always in a sorted state.



Kline-Pearson, Mika (376064)
Garron Pereira, Isabella (382401)

Round Robin Scheduler

The core loop of the Round Robin scheduler follows our architectural diagram:



Handling Release Event While Busy:

A process can enter the ReadyList under one of two circumstances: Either it has just started, or it has just become unblocked. Once it is in the ReadyList, its State becomes READY.

The Scheduler periodically (after each time quantum) dequeues the first element in the ReadyList, and sets its state to RUNNING. The Quantum is started, and the CPU is assigned.

Handling Scheduling Events:

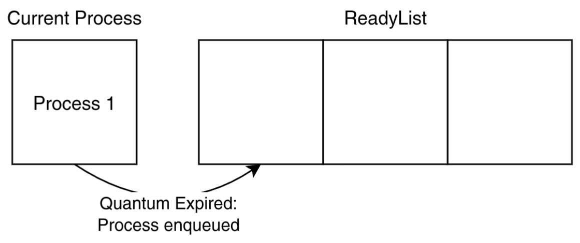
During this time, one of three possible events can happen:

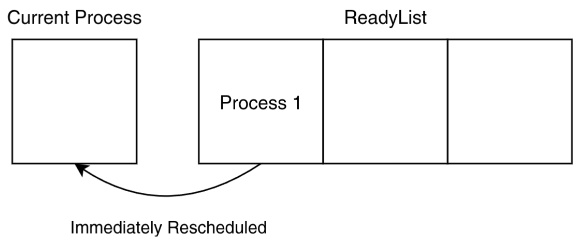
- ➢ **Quantum expires**
  (Process goes to the back of the ReadyList).
- ➢ **Current Process requests IO**
  (Process is inserted into the BlockedList).
- ➢ **Current Process completes**
  (Process is terminated).

After this, the next item in the ReadyList is scheduled.

Implementation Detail:
Re-Enqueueing a ReadyQueue while only one process exists

It can be the case that the ReadyQueue is empty, and the time quantum of the active process has expired without any new process entering the ReadyQueue. In this case, our implementation of the scheduler will simply re-enqueue the active process, then immediately dequeue it and re-assign it as the active process:





This is in contrast to the option of first checking to see if the ReadyQueue is empty, and in the event that it is, resetting the quantum without affecting the queue at all. The reasoning for this is twofold:

1. The probability of this event (there only being exactly one active process) is low, causing the vast majority of cases to undergo completely redundant checks.
2. The process of checking the length of the queue is not significantly shorter than simply adding and then removing the single element from the queue.