University of Idaho
College of Engineering

THREADING AND YOU: A ROS 2 PRIMER

JACOB FRIEDBERG

OUTLINE:

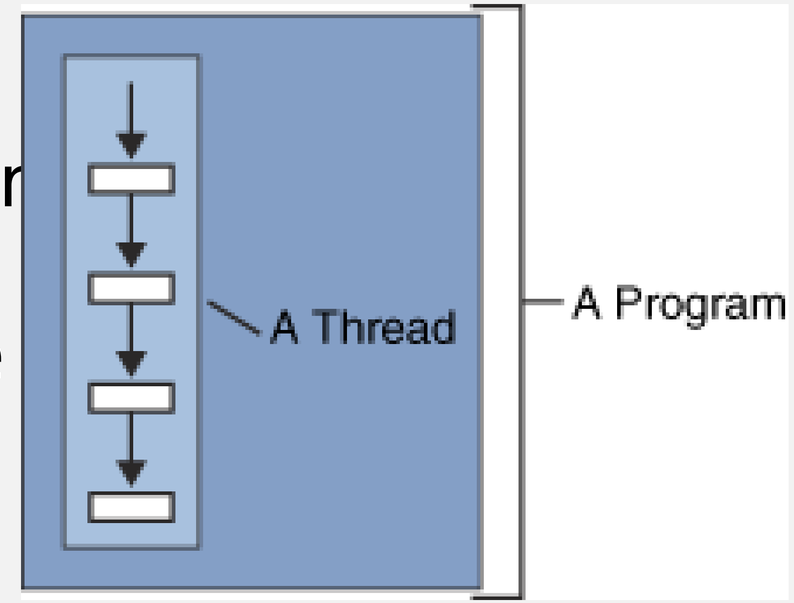
- Basics of Threading
  - When and Why to use them, Creating threads, Hazards.
- Access control
  - Critical Sections, Semaphores, Locks, Mutex
- Executors
  - Thread pools, rclpy.spin, Single Threaded, Multithreaded
- Futures
  - Waiting for results, properties
- Callback groups
  - Mutually exclusive, Reentrant, Parallel
- Scheduling in ros2
- Goal Status
- New Skeleton

THE BASICS OF THREADING

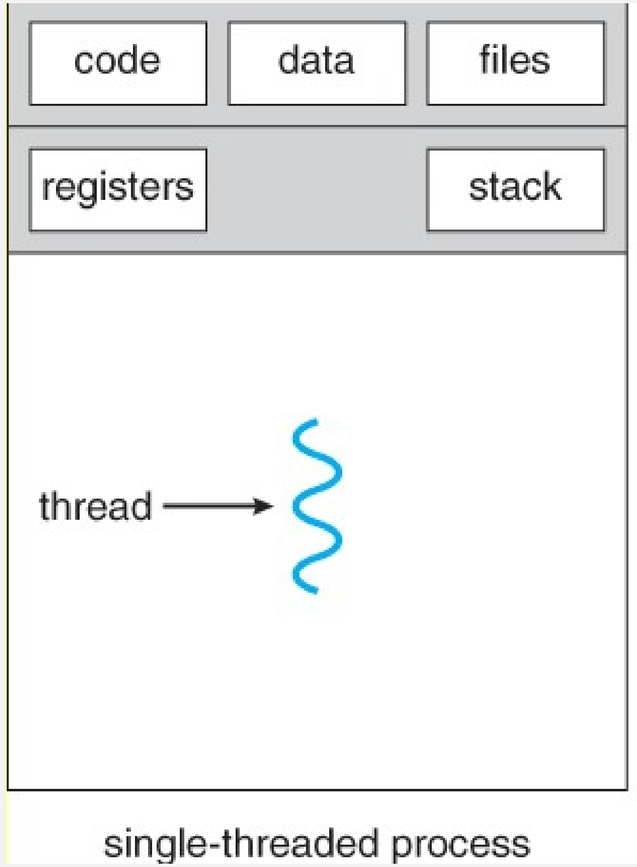
BASICS OF THREADING

- What is a thread(single threaded)?
  - A flow of execution
  - Probably most of your progr

    you have written are single threaded
    - Main()
  - A thread has its own resources allocated
    - Call Stack, variables, Scope
  - Runs on a core of the CPU



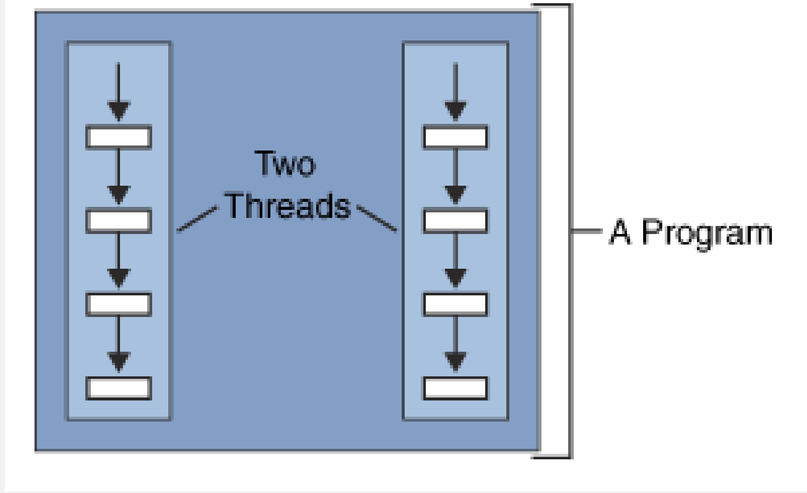
What Is a Thread? (iitk.ac.in)



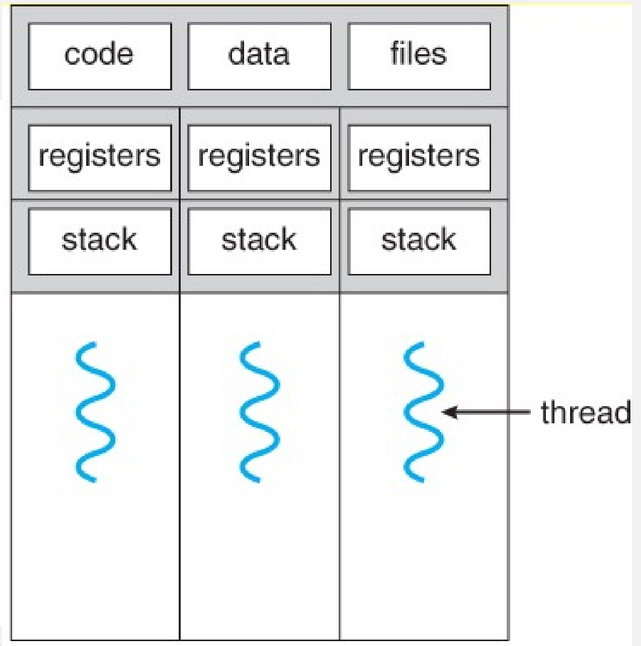
Operating Systems: Threads (uic.ed

BASICS OF THREADING

- What is multi threading?
  - Multiple flows of execution running in your program
  - Each thread has its own resources allocated
    - Call Stack, variables, scope.
  - Shares common variables and constructs
    - Open files, Global variables,



What Is a Thread? (iitk.ac.in)

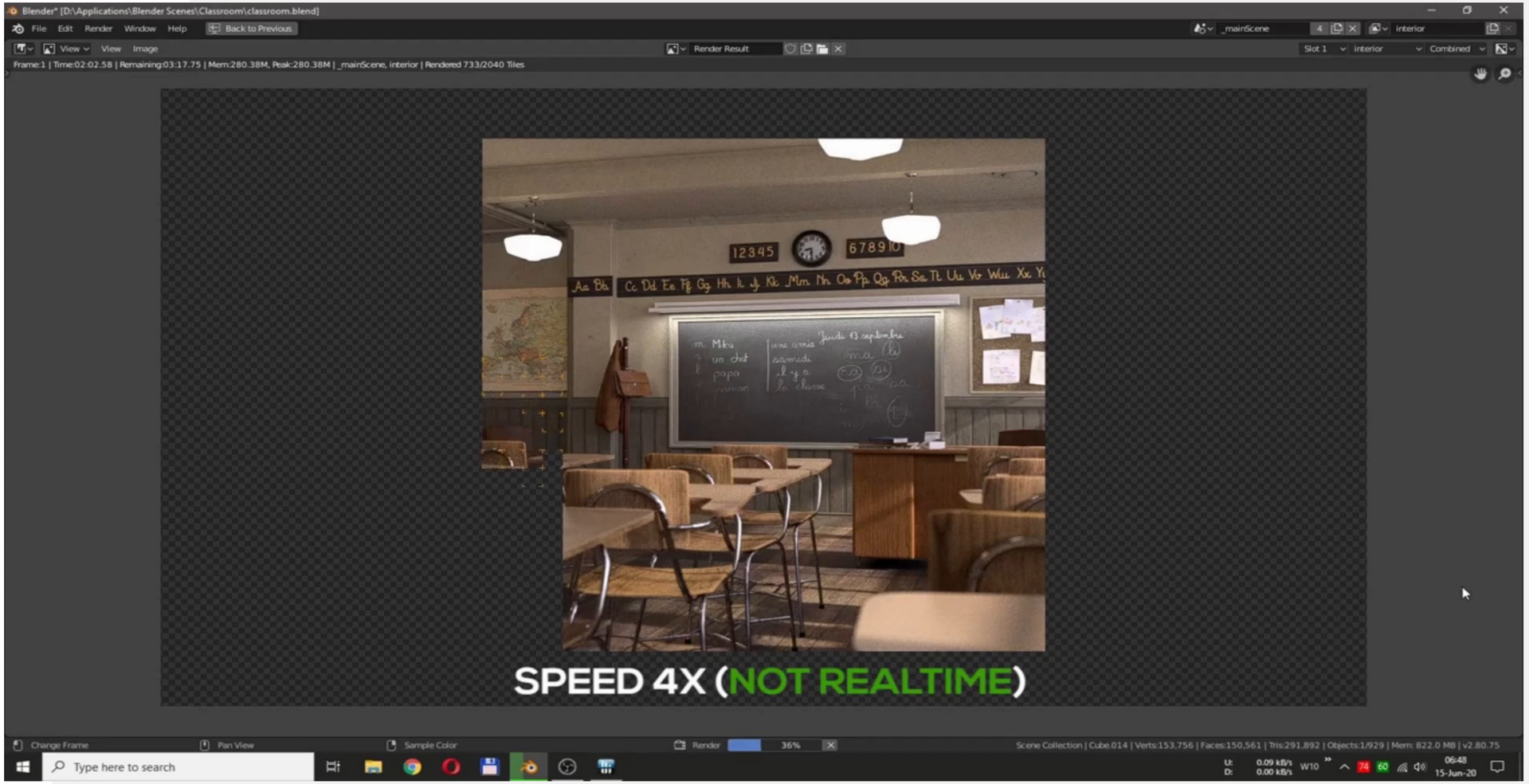


Operating Systems: Threads (uic.ed

BASICS OF THREADING

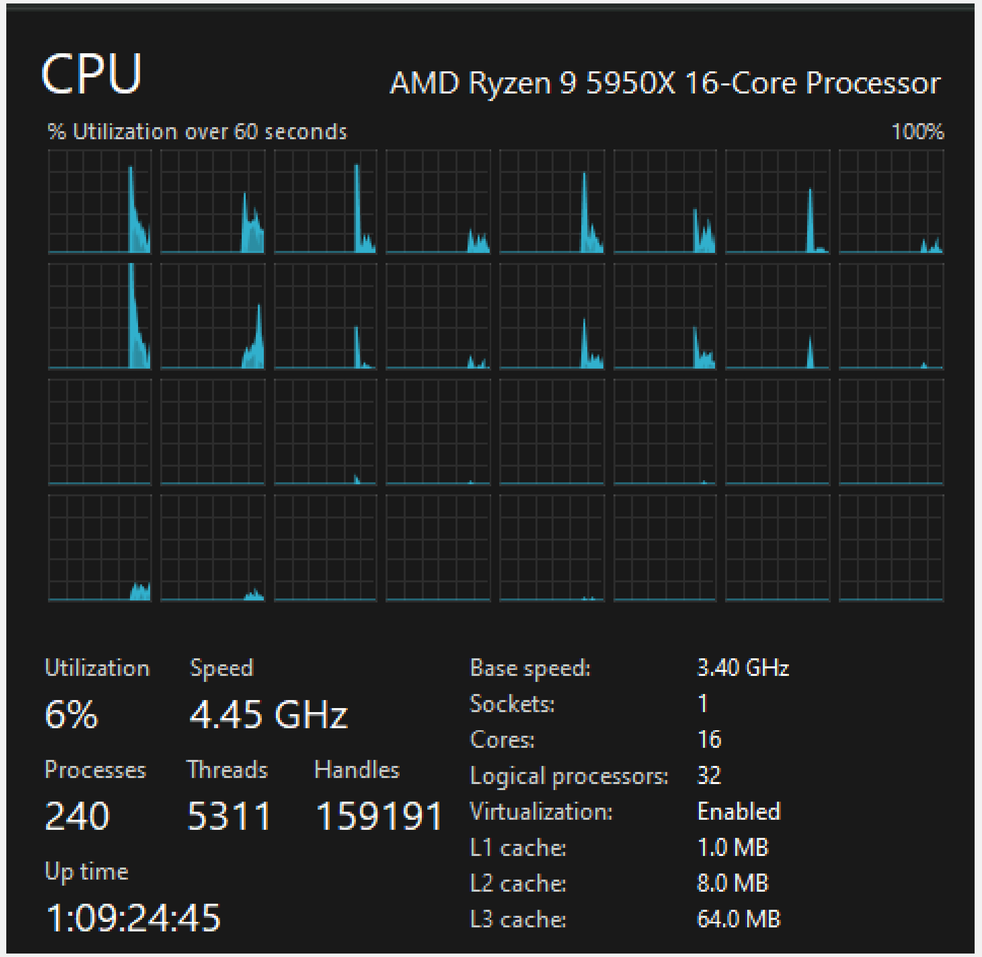
- Why use threads?
  - Allows your programs to do more than one task at once.
  - Useful, for speeding up execution of parallelizable tasks
- What is a parallelizable task?
  - A task that can be worked on independently, by multiple threads(workers)
    - Matrix multiplication, Array of calculations, Split up work into chunks that do not depend on one another.
- When to use them?
  - When you don’t want to lock up your program doing work. Or want to do something in the background.

BASICS OF THREADING



BASICS OF THREADING

- Threads are commonly used.
  - ~5300 threads running on my computer with just a few applications open.
- Most CPU’s have some form of Multi threading for their cores.
  - Can run 2 threads per core, at a slight performance cost.
- How quickly a thread works depends on many factors



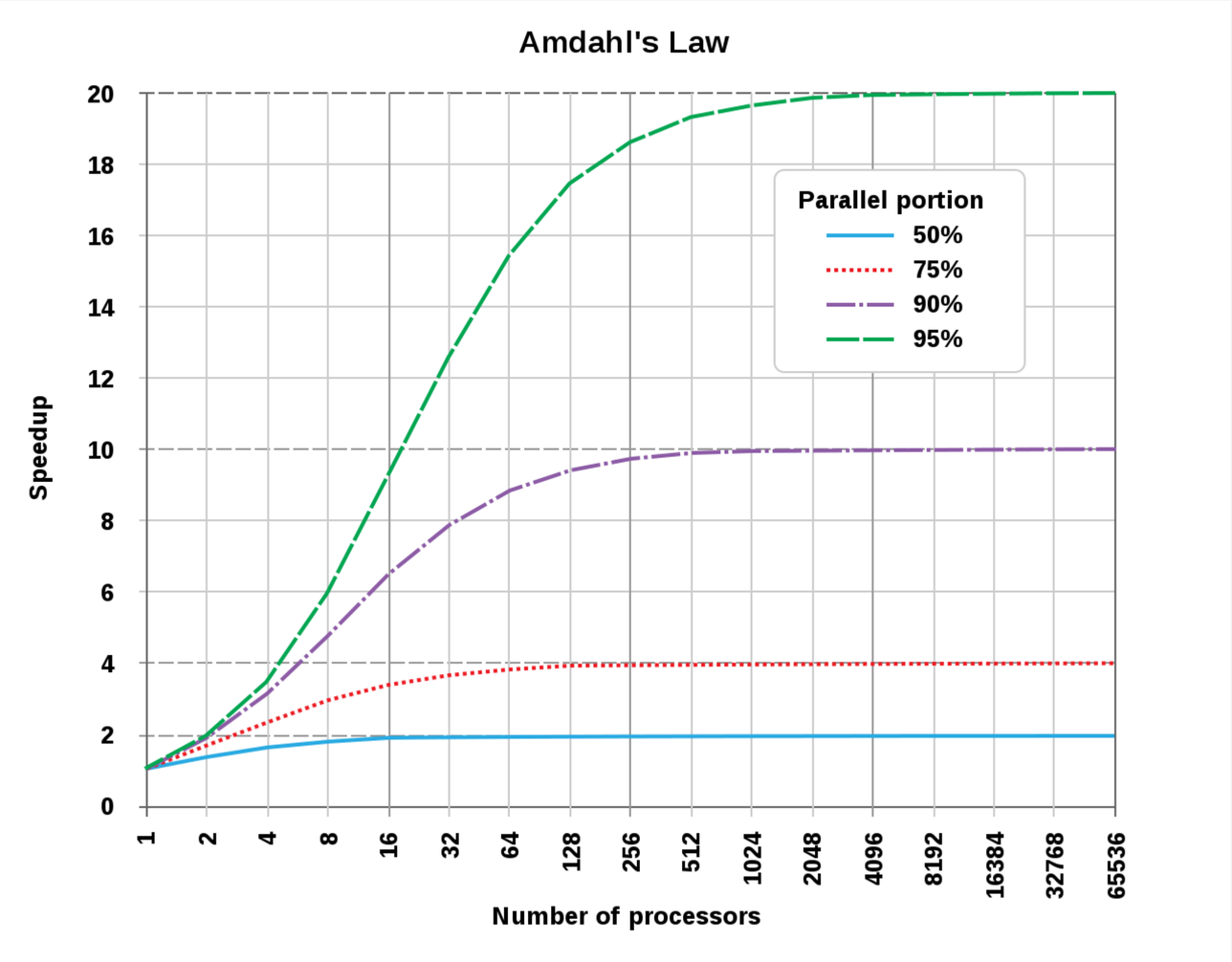
BASICS OF THREADING

- Creating Threads Python.
  - Excellent resource:
    An Intro to Threading in Python – Real Python
  - 2 main types: Background, Foreground
    - Background end when the program ends
    - Foreground run until killed(can be problematic)
  - Threading Module:
    threading — Thread-based parallelism — Python 3.11.2 documentation

CODE EXAMPLE

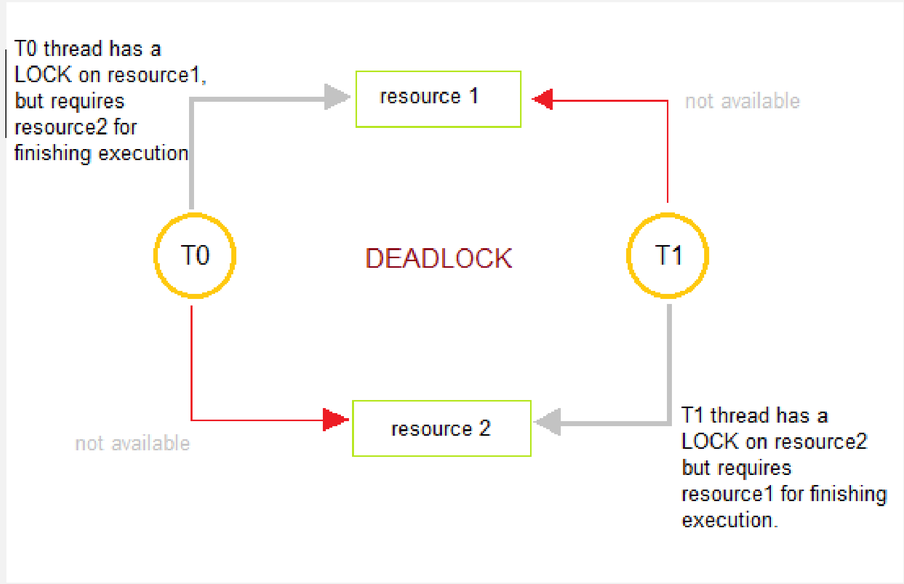
BASICS OF THREADING

- Note about parallelism
  - It is governed by Amdahls law
    - Speed up for threads is limited by parallelizable portion of the code. Serial will be the bottleneck
    - Amdahl's law - Wikipedia

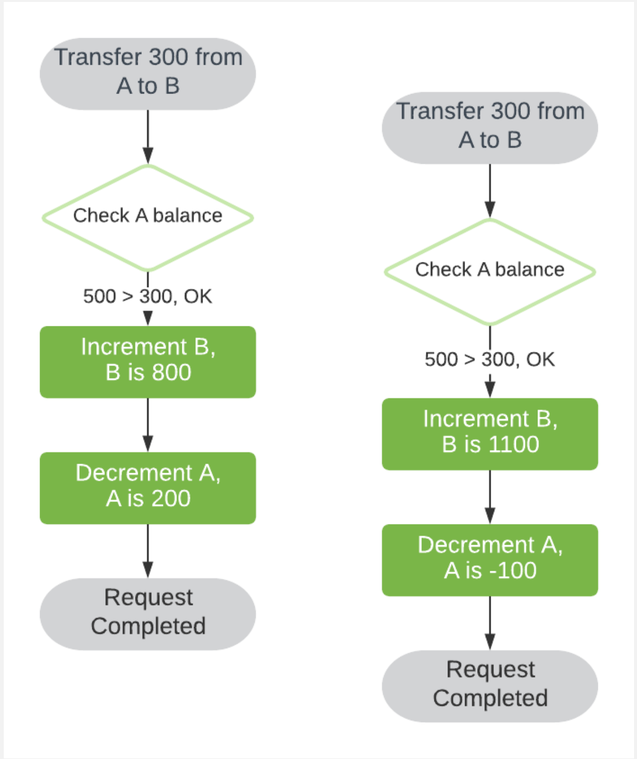


BASICS OF THREAD

- Hazards with threads.
  - Deadlock
    - Threads depend on resources that are not relinquished
  - Race Condition
    - Two threads access the same resource and want to change it. It is a race to see who writes last.
    - One thread writes data, and another thread reads during the write. Old value was read, but value is already



Introduction to Deadlocks in Operating System | Studyto



ACCESS CONTROL

ACCESS CONTROL

- Shared resources must be protected. Due to various hazards.
- Critical sections
  - areas of code that have potential hazards usually relating to thread unsafe resources. This area should only be accessed by one thread at a time.
- Python provides a bunch of methods: Link
  - Locks
  - Mutex’s
  - Barriers
  - Semaphores

ACCESS CONTROL

- Locks:
  - Only one thread can hold the lock. Mutual exclusion.
  - Acquire and Release
  - Blocking/Non-Blocking
- Barriers
  - Synchronizes threads. Block until multiple threads are ready.
  - Thread number to wait, and timeout to block
- Semaphores
  - Define max number of threads who can acquire this resource

ACCESS CONTROL

Code example

EXECUTORS

EXECUTORS

- RCLPY has two types of executors
  - Single threaded
  - Multi threaded
- What are executors?
  - High level objects that contain a callback list(work to do) and a thread pool. Plus, some other stuff that we don’t care about.
- What is a thread pool?
  - A thread pool is a higher-level object that manages creation and lifetime of threads, and schedules work for them to do.

EXECUTORS

RCLPY Single Threaded Executor

Executors — ROS 2 Documentation: Foxy documentation

- A thread pool with a single thread.
- This type of executor should never have its thread blocked.
  - RCLPY has internal function callbacks(action clients…) that need a thread to function correctly, and update values(Goal handles)
- RCLPY.spin() creates this type of executor and calls spin on it.

EXECUTORS

- RCLPY Multi Threaded Executor
  - A thread pool with a user defined number of threads.
  - This type of executor can have its threads blocked if you are careful about it.
    - Depends on how many threads were given to the executor. You **ALWAYS** need at least 1 thread free to perform internal callbacks.
    - If you do block, there is still a chance of deadlock. Your milage may vary.
  - executor.spin(node) initiates work. This will still consume the main_thread, but you can

FUTURES

FUTURES

What is a Future object? Futures — Python 3.11.2 documentation

- Async functions are functions that do not block and may have a long run processes to return a result.(IO, networking, user defined).
- Async functions provide a future value, which is an object that indicates that the result of that function is ready.
  - This is to allow you to do other things while you wait (non blocking)
- Future.done()
  - Returns true when your future result is ready to be assigned.
- Future.Result()
  - Returns the result of the future which is of a type specified by the async function.

FUTURES

Notes about RCLPY futures

- These are rclpy specific and implement most of the functionality of python futures. Some things are the same, others are slightly different.
- Read the source code for rclpy if something is not working as described i

`send_goal_async`(*goal*, *feedback_callback=None*, *goal_uuid=None*)

Send a goal and asynchronously get the result.

The result of the returned Future is set to a ClientGoalHandle when receipt of the goal is acknowledged by an action server.

Parameters:
- **goal** (*action_type.Goal*) – The goal request.
- **feedback_callback** (*function*) – Callback function for feedback associated with the goal.
- **goal_uuid** – Universally unique identifier for the goal. If None, then a random UUID is generated.

Type: unique_identifier_msgs.UUID

Returns: a Future instance to a goal handle that completes when the goal request has been accepted or rejected.

Return type: `rclpy.task.Future` instance

Raises: TypeError if the type of the passed goal isn't an instance of the Goal type of the provided action when the service was constructed.

CALLBACK GROUPS

CALLBACK GROUPS

What is a callback group?

Using Callback Groups — ROS 2 Documentation: Foxy documentation

- A way to control the scheduling of the callback functions for executors in RCLPY
- 2 types
  - Reentrant
    - “Allows the executor to schedule and execute the group’s callbacks in any way the it sees fit, without restrictions. This means that, in addition to different callbacks being run parallel to each other, different instances of the same callback may also be executed concurrently”
  - Mutually exclusive
    - “Mutually Exclusive Callback Group prevents its callbacks from being executed in parallel - essentially making it as if the callbacks in the group were executed by a SingleThreadedExecutor”

CALLBACK GROUPS

What is a callback group?

Using Callback Groups — ROS 2 Documentation: Foxy documentation

- Notes about the groups
  - “Callbacks belonging to different callback groups (of any type) can always be executed parallel to each other”
  - “It is also important to keep in mind that different ROS 2 entities relay their callback group to all callbacks they spawn. For example, if one assigns a callback group to an action client, all callbacks created by the client will be assigned to that callback

```
my_callback_group = MutuallyExclusiveCallbackGroup()
my_subscription = self.create_subscription(Int32, "/topic", self.callback, qos_profile=1,
                                           callback_group=my_callback_group)
```

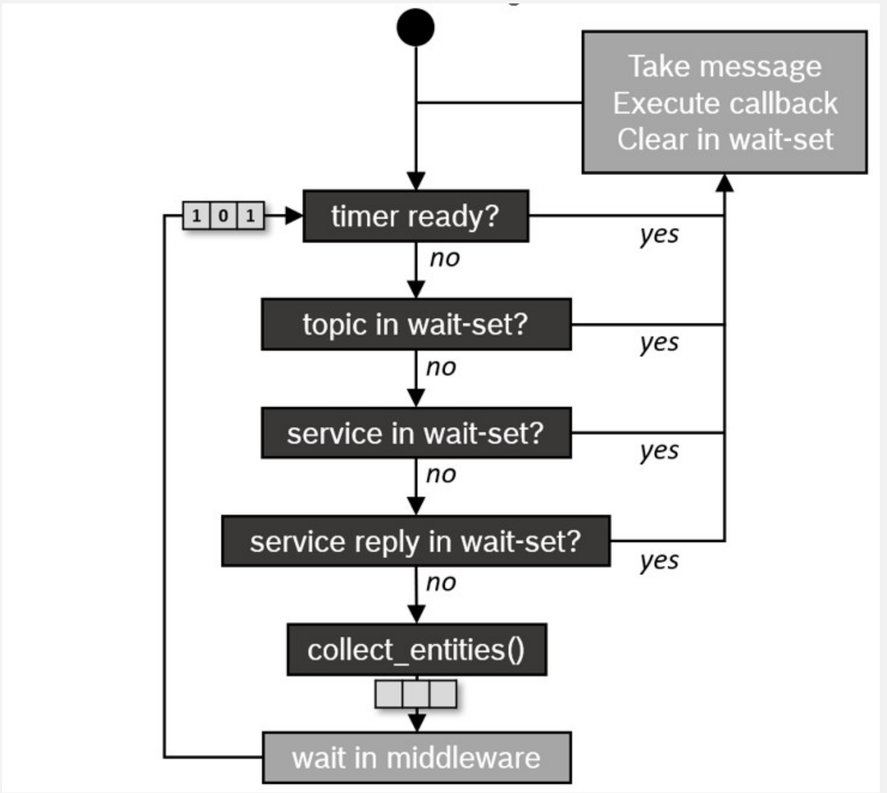
SCHEDULING IN ROS2

SCHEDULING IN ROS2

What is scheduling?

https://docs.ros.org/en/foxy/Concepts/About-Executors.html

- Scheduling is the order in which our callbacks are executed.
- In ROS2, if your callbacks are short and are “shorter than the period in which messages and events occur” in ROS2, then they will likely be executed in FIFO(first in first out) order.
- Long running callbacks will cause a backup in messeges/events and will be executed in a round robin fashion as defined on the right.



SCHEDULING IN ROS2

- Why does it matter?

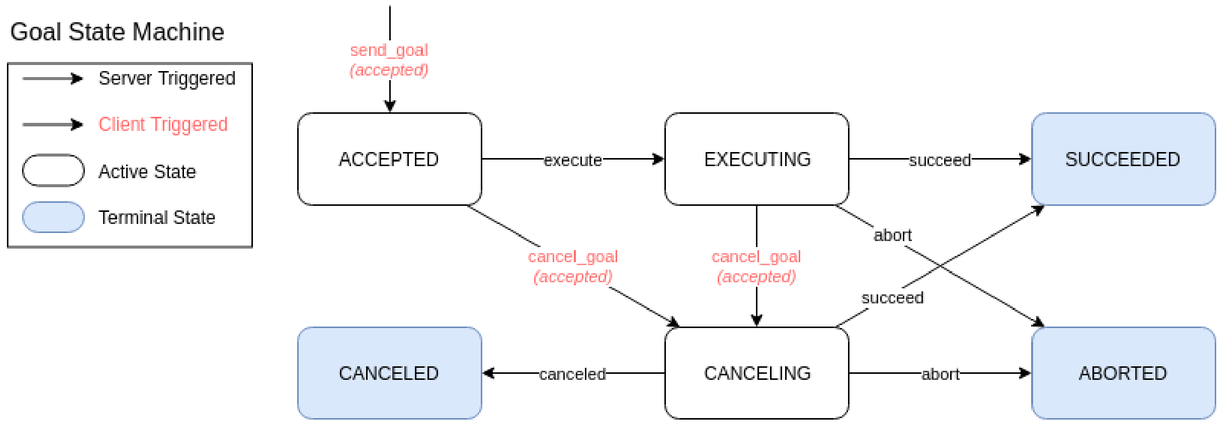
  Executors — ROS 2 Documentation: Foxy documentation

  - Callbacks may suffer priority inversion
    - Higher priority tasks are blocked until low priority callbacks finish
  - No explicit control over callback execution order
  - No built-in control over triggering for specific topics.
- ROS2 is considered a distributed(due to the nodes) RTOS but has challenges meeting hard real time requirements.
  - Hard real time: Events/actions must happen right now! Ex: Pull the fuel rods out of the reactor.

GOAL STATUS

SCHEDULING IN ROS2

- Goal Status defines the current state of a goal.
  Actions (ros2.org)
- This value is stored in the goal handle as GoalHandle.Status
- Use these statuses to determine the state of the



There are three active states:

File: action_msgs/msg/GoalStatus.msg

Raw Message Definition

```
# An action goal can be in one of these states after it is accepted by an action
# server.
#
# For more information, see http://design.ros2.org/articles/actions.html

# Indicates status has not been properly set.
int8 STATUS_UNKNOWN = 0
# The goal has been accepted and is awaiting execution.
int8 STATUS_ACCEPTED = 1
# The goal is currently being executed by the action server.
int8 STATUS_EXECUTING = 2
# The client has requested that the goal be canceled and the action server has
# accepted the cancel request.
int8 STATUS_CANCELING = 3
# The goal was achieved successfully by the action server.
int8 STATUS_SUCCEEDED = 4
# The goal was canceled after an external request from an action client.
int8 STATUS_CANCELED = 5
# The goal was terminated by the action server without an external request.
int8 STATUS_ABORTED = 6
# Goal info (contains ID and timestamp).
GoalInfo goal_info
# Action goal state-machine status.
int8 status
```

NEW SKELETON