Week 11

태그
내용

6장, RQt

- `RQT` 는 ROS + Qt 의 합성어
- `RQT` 는 GUI 형태의 도구와 사용자 인터페이스를 구현할 수 있는 프레임워크
- `RQT` 는 리눅스, macOS, 윈도우에서 지원되며 C++ 과 python 으로 개발

RQt 플러그인

- `RQt` 플러그인 형태로 개발하면 여러가지 장점이 있음
- `RQt` 플러그인 프레임워크를 구성하는 패키지
  1. rqt 패키지: `RQt` 메타패키지로, rqt_qui, rqt_gui_cpp, rqt_gui_py, rqt_py_common 포함
  2. rqt_gui 패키지: 여러 `RQt` 위젯을 단일 창에 도킹하는 Widget 패키지
  3. rqt_gui_cpp 패키지: C++ 클라이언트 用 라이브러리
  4. rqt_gui_py 패키지: python 클라이언트 用 라이브러리
  5. rqt_py_common 패키지: python 으로 작성된 공용 라이브러리 패키지
  6. qt_qui_core 패키지: qt_gui, qt_gui_cpp, qt_gui_py_common 등을 포함한 메타패키지
  7. python_qt_binding 패키지: `QtCore` `QtGui` `QtWidgets` Qt API 와 연결되는 바인딩 패키지
- `PyQt` 는 GPL, `PySide` 는 LGPL 라이선스 → python_qt_binding 통해 자유롭게 사용

RQt 예제 설정 파일 살펴보기

```
<package format="3">
...
  <export>
    <build_type>ament_cmake</build_type>
    <rqt_gui plugin="${prefix}/plugin.xml"/>
  </export>
</package>
```

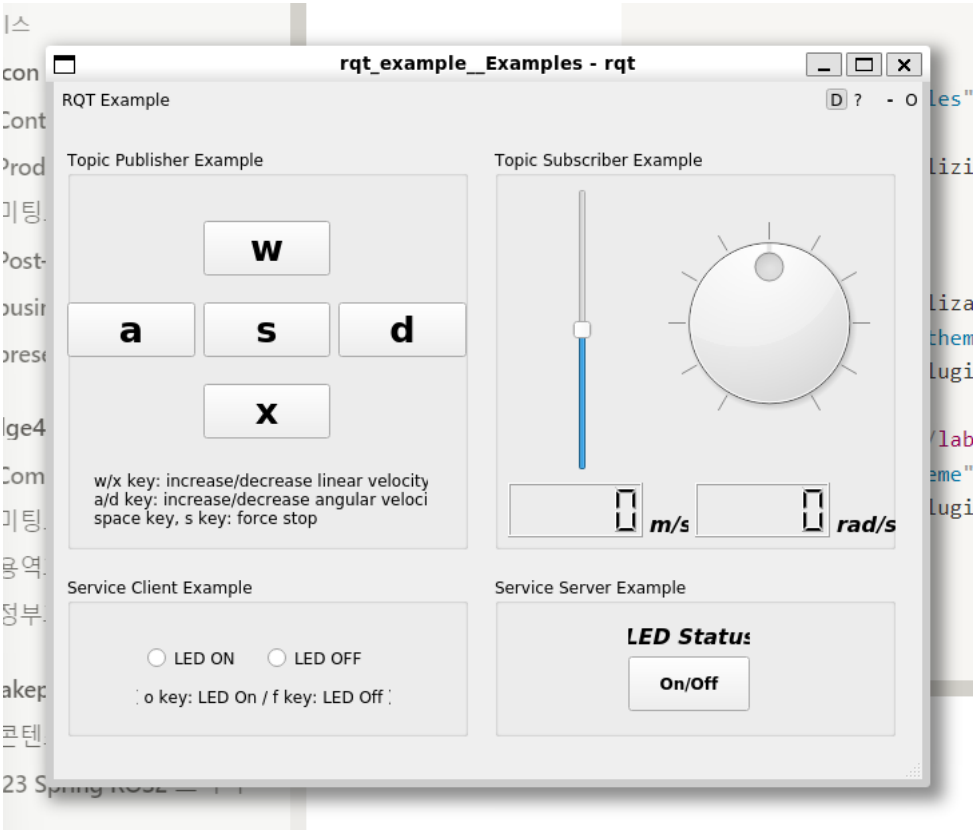
1. package.xml 파일에 rqt_gui 태그로 plugin.xml 파일을 넣는다.

```
<library path="src">
  <class name="Examples" type="rqt_example.examples.Examples" base_class_type="rqt_gui_py::Plugin">
    <description>
      A plugin visualizing messages and services values
    </description>
    <qtgui>
      <group>
        <label>Visualization</label>
        <icon type="theme">folder</icon>
        <statustip>Plugins related to visualization</statustip>
      </group>
      <label>Viewer</label>
      <icon type="theme">utilities-system-monitor</icon>
      <statustip>A plugin visualizing messages and services values</statustip>
    </qtgui>
  </class>
</library>
```

1. plugin.xml 의 예제 파일 입니다.
2. qtgui group label 등의 태그로 플러그인을 명세합니다.

```
$ cbp rqt_example
$ ros2 run rqt_example rqt_example
```

bashrc 에 추가한 cbp 명령어로 빌드 하고, ros2 run 명령어로 실행 하면, 다음과 같은 RQt GUI 화면이 나타난다.



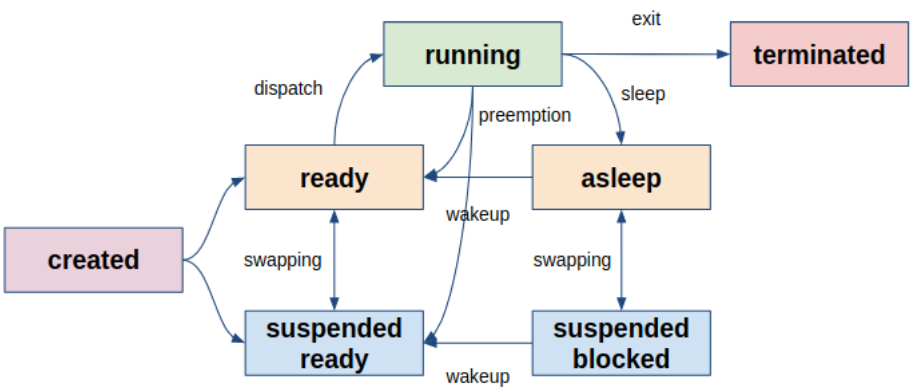
1. x, a, s, d, w 를 통해 turtle 을 이동한다. (토픽과 연결된 RQt GUI)
2. service client example 인, LED on, off

7장 Lifecycle

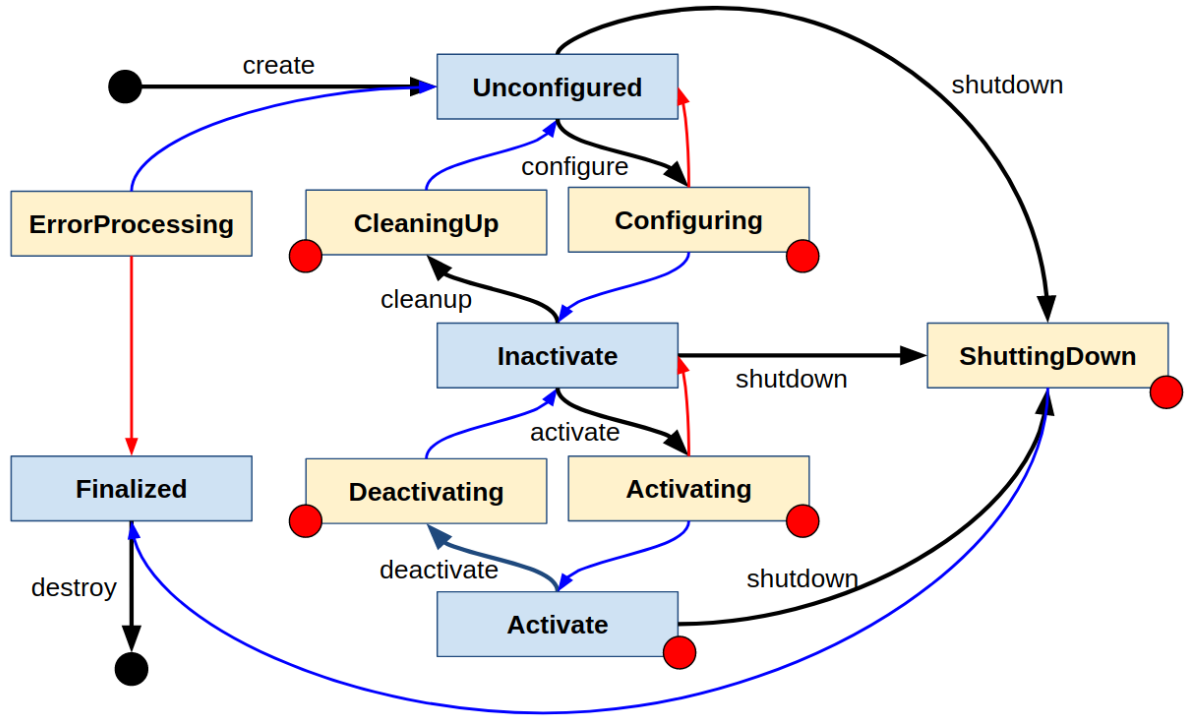
`ROS2` 는 로봇의 구성요소들이 `노드` 단위로 설계되고 구현된다. `노드`는 프로세스와 유사하게 라이프사이클을 갖는다.

노드관리

- ROS2 노드의 상태를 관리하기 위해 `Lifecycle` API 를 제공
- `Lifecycle` API 를 통해 노드의 상태를 확인, 재실행, 교체할수 있다.



Lifecycle



Lifecycle 은 총 4개의 주요 상태로 구성된다.

- `Unconfigured` 노드가 생성(create)된 직후의 상태, 에러 발생 이후 다시 조정될 수 있는 상태
- `Inactivate` 노드가 생성이 되었지만, 아무런 동작을 수행하지 않는 상태
- `Activate` 노드가 활성화 되어 정상적으로 동작하는 상태
- `Finalized` 노드가 종료되어 메모리에서 해제되기 직전의 상태

Lifecycle 은 총 6개의 전환 상태가 있다.

- `Configuring` 노드를 생성되고 활성화를 위해 설정을 수행하는 상태
- `CleaningUp` 노드가 처음 생성되었을 때 상태와 동일하게 만드는 과정
- `Activating` 노드가 동작을 수행하기 전 마지막 준비 과정 수행
- `Deactivating` 노드가 동작을 수행하기 전으로 돌아가는 과정 수행
- `ShuttingDown` 노드가 메모리에서 해제되기 전에 필요한 과정을 수행

```
...
class LifecycleTalker : public rclcpp_lifecycle::LifecycleNode
{
public:
  explicit LifecycleTalker(const std::string & node_name, bool intra_process_comms = false)
  : rclcpp_lifecycle::LifecycleNode(node_name,
      rclcpp::NodeOptions().use_intra_process_comms(intra_process_comms))
  {}

  void
  publish()
  {
    static size_t count = 0;
    auto msg = std::make_unique<std_msgs::msg::String>();
    msg->data = "Lifecycle HelloWorld #" + std::to_string(++count);

    // Print the current state for demo purposes
    if (!pub_->is_activated()) {
      RCLCPP_INFO(
        get_logger(), "Lifecycle publisher is currently inactive. Messages are not published.");
    } else {
      RCLCPP_INFO(
        get_logger(), "Lifecycle publisher is active. Publishing: [%s]", msg->data.c_str());
```

```
  }

  // We independently from the current state call publish on the lifecycle
  // publisher.
  // Only if the publisher is in an active state, the message transfer is
  // enabled and the message actually published.
  pub_->publish(std::move(msg));
}

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_configure(const rclcpp_lifecycle::State &)
{
  pub_ = this->create_publisher<std_msgs::msg::String>("lifecycle_chatter", 10);
  timer_ = this->create_wall_timer(
    1s, std::bind(&LifecycleTalker::publish, this));

  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
}

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_activate(const rclcpp_lifecycle::State & state)
{
  // The parent class method automatically transition on managed entities
  // (currently, LifecyclePublisher).
  // pub_->on_activate() could also be called manually here.
  // Overriding this method is optional, a lot of times the default is enough.
  LifecycleNode::on_activate(state);

  RCUTILS_LOG_INFO_NAMED(get_name(), "on_activate() is called.");

  std::this_thread::sleep_for(2s);

  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
}

/// Transition callback for state deactivating
/**
 * on_deactivate callback is being called when the lifecycle node
 * enters the "deactivating" state.
 * Depending on the return value of this function, the state machine
 * either invokes a transition to the "inactive" state or stays
 * in "active".
 * TRANSITION_CALLBACK_SUCCESS transitions to "inactive"
 * TRANSITION_CALLBACK_FAILURE transitions to "active"
 * TRANSITION_CALLBACK_ERROR or any uncaught exceptions to "errorprocessing"
 */
rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_deactivate(const rclcpp_lifecycle::State & state)
{
  // The parent class method automatically transition on managed entities
  // (currently, LifecyclePublisher).
  // pub_->on_deactivate() could also be called manually here.
  // Overriding this method is optional, a lot of times the default is enough.
  LifecycleNode::on_deactivate(state);

  RCUTILS_LOG_INFO_NAMED(get_name(), "on_deactivate() is called.");

  // We return a success and hence invoke the transition to the next
  // step: "inactive".
  // If we returned TRANSITION_CALLBACK_FAILURE instead, the state machine
  // would stay in the "active" state.
  // In case of TRANSITION_CALLBACK_ERROR or any thrown exception within
  // this callback, the state machine transitions to state "errorprocessing".
  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
}

rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
on_cleanup(const rclcpp_lifecycle::State &)
{
  // In our cleanup phase, we release the shared pointers to the
  // timer and publisher. These entities are no longer available
  // and our node is "clean".
  timer_.reset();
  pub_.reset();

  return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
}

/// Transition callback for state shutting down
/**
 * on_shutdown callback is being called when the lifecycle node
 * enters the "shuttingdown" state.
 * Depending on the return value of this function, the state machine
 * either invokes a transition to the "finalized" state or stays
 * in its current state.
 * TRANSITION_CALLBACK_SUCCESS transitions to "finalized"
 * TRANSITION_CALLBACK_FAILURE transitions to current state
```

```
   * TRANSITION_CALLBACK_ERROR or any uncaught exceptions to "errorprocessing"
   */
  rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
  on_shutdown(const rclcpp_lifecycle::State & state)
  {
    // In our shutdown phase, we release the shared pointers to the
    // timer and publisher. These entities are no longer available
    // and our node is "clean".
    timer_.reset();
    pub_.reset();

    RCUTILS_LOG_INFO_NAMED(
      get_name(),
      "on shutdown is called from state %s.",
      state.label().c_str());

    // We return a success and hence invoke the transition to the next
    // step: "finalized".
    // If we returned TRANSITION_CALLBACK_FAILURE instead, the state machine
    // would stay in the current state.
    // In case of TRANSITION_CALLBACK_ERROR or any thrown exception within
    // this callback, the state machine transitions to state "errorprocessing".
    return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
  }

private:
  // We hold an instance of a lifecycle publisher. This lifecycle publisher
  // can be activated or deactivated regarding on which state the lifecycle node
  // is in.
  // By default, a lifecycle publisher is inactive by creation and has to be
  // activated to publish messages into the ROS world.
  std::shared_ptr<rclcpp_lifecycle::LifecyclePublisher<std_msgs::msg::String>> pub_;

  // We hold an instance of a timer which periodically triggers the publish function.
  // As for the beta version, this is a regular timer. In a future version, a
  // lifecycle timer will be created which obeys the same lifecycle management as the
  // lifecycle publisher.
  std::shared_ptr<rclcpp::TimerBase> timer_;
};

/**
 * A lifecycle node has the same node API
 * as a regular node. This means we can spawn a
 * node, give it a name and add it to the executor.
 */
int main(int argc, char * argv[])
{
  // force flush of the stdout buffer.
  // this ensures a correct sync of all prints
  // even when executed simultaneously within the launch file.
  setvbuf(stdout, NULL, _IONBF, BUFSIZ);

  rclcpp::init(argc, argv);

  rclcpp::executors::SingleThreadedExecutor exe;

  std::shared_ptr<LifecycleTalker> lc_node =
    std::make_shared<LifecycleTalker>("lc_talker");

  exe.add_node(lc_node->get_node_base_interface());

  exe.spin();

  rclcpp::shutdown();

  return 0;
}
```

8장 Security

- `ROS2` 에서 디자인 설계 단계에서 보안 기능 개선을 심각하게 고려하였음
- `ROS2` 에서 기존의 routing server 의존한 메시지 모델이 아닌 `DDS` 를 선정
- `DDS` 를 선택하면서 자연스럽게 `DDS-Security` 라는 `DDS` 보안 사양을 적용
- `ROS` 커뮤니티에서 `SROS2` 를 개발하여 보안 기능을 지원하고 있음

SROS2 (Secure Robot Operating System 2)

- `DDS` 의 확장 기능인 DDS-Security 를 ROS2 에서 사용할 수 있도록 개발

- `SROS2` 는 RCL 에 domain participant 를 위한 보안 기능을 추가

  추가된 기능은 보안파일 지원, 실행옵션, 보안기능 on/off 등
- `DDS-Security` 에서 5가지 기능
  - `Authentication` DDS 도메인 참가자 확인
  - `Access control` 참가자가 수행할 수 있는 DDS 관련 작업에 대한 제한
  - `Cypyography` 암호화, 서명, 해싱 처리
  - `Logging DDS` 보안 관련 이벤트를 감시하는 기능
  - `Data tagging` 데이터 샘플에 태그를 추가하는 기능

인증 (Authentication)

- 인증 플러그인에서 도메인 참가자를 확인하는 작업에 x.509 PKI 를 사용한다.
- x.509 인증서는 CA 에 의해 서명이 되어야 한다.

> 💡 x.509 는 공개키 인증서 형식의 표준, HTTPS에 서용되는 TLS, SSL 등등에서 사용 中

엑세스 제어 (Access control)

- 특정 도메인 참여자의 DDS 관련 기능에 대한 제한을 정의하고 적용한다.
- XML 문서의 형태로 엑세스 제어 방칙을 적용할 수 있다.

암호화 (Crytography)

- 암호화 플러그인에서는 암호화, 복호화, 서명, 해싱 등의 모든 암호화 관련 작업을 담당

> 💡 현재 SROS2 의 개발은 RTI Connnext Secure 5.3x 및 eProsima’ s Fast-RTPS 1.6x 버전 이상에서 테스트 되고 있다.

노드 인증

- 암호키 등의 보안 설정을 보관하는 폴더를 생성해야한다.

```
$ ros2 security create_keystore key_box
creating keystore: key_box
creating new CA key/cert pair
creating governance file: key_box/enclaves/governance.xml
creating signed governance file: key_box/enclaves/governance.p7s
all done! enjoy your keystore in key_box
cheers!

$ ls key_box/
enclaves  private  public

$ tree key_box/
.
├── enclaves
│   ├── governance.p7s
│   └── governance.xml
├── private
│   ├── ca.key.pem
│   ├── identity_ca.key.pem -> ca.key.pem
│   └── permissions_ca.key.pem -> ca.key.pem
└── public
    ├── ca.cert.pem
    ├── identity_ca.cert.pem -> ca.cert.pem
    └── permissions_ca.cert.pem -> ca.cert.pem

3 directories, 8 files
```

- 키 저장을 위한 폴더와 OpenSSL 기반의 개인키와 공개키가 생성 되었다.

엑세스 제어

- 엑세스 제어는 지정한 네임스페이스, 서비스, 액션에 대해 송신, 수신, 실행을 UNIX 퍼미션 같은 개념으로 쓰기, 읽기, 실행 등의 허가

- XML 형식으로 파일을 만들어서 엑세스 제어 방침으로 관리 할 수 있다.

```
$ cat sample.policy.xml
<?xml version="1.0" encoding="UTF-8"?>
<policy version="0.2.0"
  xmlns:xi="http://www.w3.org/2001/XInclude">
  <enclaves>
    <xi:include href="talker_listener.policy.xml"
      xpointer="xpointer(/policy/enclaves/*)"/>
    <xi:include href="add_two_ints.policy.xml"
      xpointer="xpointer(/policy/enclaves/*)"/>
    <xi:include href="minimal_action.policy.xml"
      xpointer="xpointer(/policy/enclaves/*)"/>
    <enclave path="/sample_policy/admin">
      <profiles>
        <profile ns="/" node="admin">
          <xi:include href="common/node.xml"
            xpointer="xpointer(/profile/*)"/>
          <actions call="ALLOW" execute="ALLOW">
            <action>fibonacci</action>
          </actions>
          <services reply="ALLOW" request="ALLOW">
            <service>add_two_ints</service>
          </services>
          <topics publish="ALLOW" subscribe="ALLOW">
            <topic>chatter</topic>
          </topics>
        </profile>
      </profiles>
    </enclave>
  </enclaves>
</policy>
```

보안 vs 성능

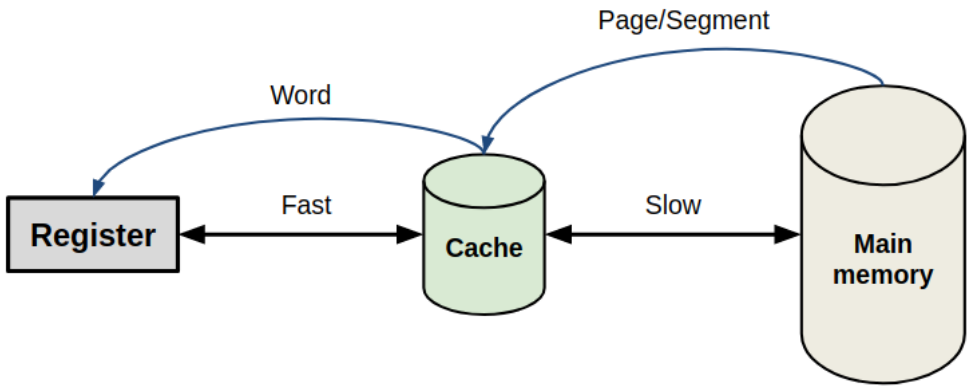
- `SORS2` 의 보안 기능은 overhead 로 인해 성능과 tradeoff 관계 (지연시간 5대, 대역폭 1/5)

https://arxiv.org/pdf/2208.02615.pdf

9장 Real-time

실시간 시스템이란?

- 정해진 시간 (Deadline) 에 입력에 대한 출력을 보장하는 (Determinism) 시스템 이다.
  - Hard real-time system
  - Firm real-time system
  - Soft real-time system

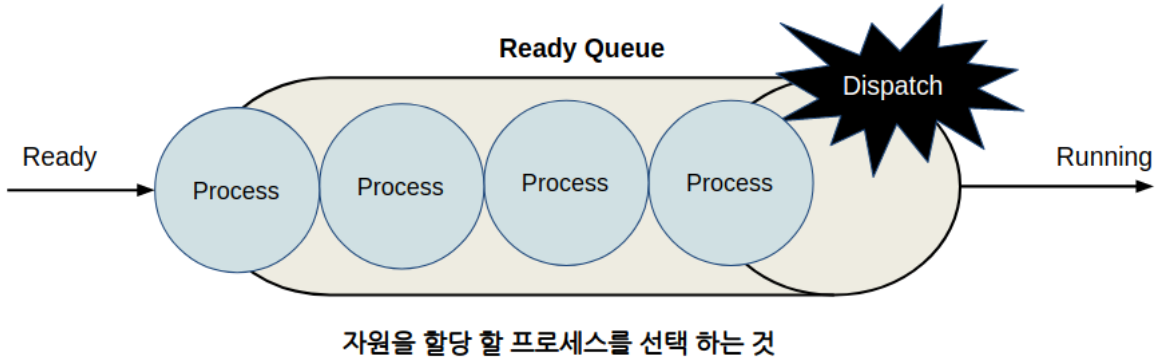


- Page fault 는 real-time 을 저해하는 하나의 원인이다.

- mlockall 으로 가상 메모리 조사를 RAM 에 미리 할당 한다.
- dynamic memory pool 으로 가상 메모리를 고정된 크기로 할당
- 기존 memory allocation 은 실시간성 보장이 어렵기 때문에 별도로 개발
- Global variables and arrays 를 사용
- 가상 메모리를 사용하는 것보다 물리 메모리의 포인터를 사용

프로세스 스케줄링

- 스케쥴링 정책은 크게 선점여부와 우선순위로 나뉜다.
- 선점 스케줄링(Preemptive scheduling)은 현재 실행중인 프로세스를 인터럽트 하여 다른 프로세스에 자원을 할당 시킨다. 비선점 스
- 우선순위는 정적 우선순위(Static priority)와 동적 우선순위(Dynamic priority) 로 구분



Performance benchmarking

실시간성을 판단하기 위해서는 페이지 폴트와 스케쥴링 지터(Scheduling jitter)를 확인하고 개선한다.

1) cyclictest

실시간 환경의 스케쥴링 지터를 측정하기 위한 리눅스 명령어. 스레드의 갯수와 우선순위, 스케쥴러 타입을 설정하여 레이턴시와 지터가 얼

2) rttest

ROS 2 에서 지원하는 라이브러리로 사용자 코드에서 발생하는 스케쥴링 지터를 저장하여 알려줌

3) getrusage

페이지 폴트, 메모리 스왑, I/O

데모 코드

- ROS 2 를 이용한 실시간 프로그래밍 데모 —> pendulum_control 패키지 (최소 8Gb RAM 에서 동작 가능)
- 기대결과 : page fault 0, avr. laterncy 30,000ns 이하

```
$ . ~/ros2_foxy/install/local_setup.bash
$ cd ~/ros2_foxy/install/pendulum_control/bin
$ ./pendulum_demo > output.txt
```

- output.txt 에 저장된 로그를 확인해 보면 rttest 결과 페이지 폴트가 한번 있었음을 확인할 수 있다.

```
$ cat output.txt
Initial major pagefaults: 172
Initial minor pagefaults: 4115
rttest statistics:
  - Minor pagefaults: 1
  - Major pagefaults: 0
  Latency (time after deadline was missed):
    - Min: 56179 ns
    - Max: 170799 ns
    - Mean: 129188.509000 ns
    - Standard deviation: 40351.343055
```

- 페이지 폴트 횟수를 줄이기 위해서는 RAM 에 저장할 수 있는 메모리 크기에 대한 제한을 풀어줘야만 한다.
- 관리자 권한으로 /etc/security/limits.conf 파일 가장 아랫줄에 memlock 옵션(${username} - memlock -1)을 추가해주고, 재부팅 이

```
$ ulimit -l unlimited
```

- output.txt 에 장된 로그를 열어보면 rttest 결과 에서 페이지 폴트가 완전히 없어진 것도 확인할 수 있다.

```
$ cat output.txt
Initial major pagefaults: 0
Initial minor pagefaults: 2124268
rttest statistics:
  - Minor pagefaults: 0
  - Major pagefaults: 0
  Latency (time after deadline was missed):
    - Min: 53620 ns
    - Max: 177322 ns
    - Mean: 146186.291000 ns
    - Standard deviation: 17348.023064
```

다음은 logger 노드를 이용해서 pendulum_demo 의 런타임 로그를 확인해보자. 새로운 터미널 창을 열어 아래 명령어로 logger 노드를 만

Terminal 2
$. ~/ros2_foxy/install/local_setup.bash
$ cd ~/ros2_foxy/install/pendulum_control/bin
$ ./pendulum_demo > output.txt

```
# Terminal 1
$ . ~/ros2_foxy/install/local_setup.bash
$ cd ~/ros2_foxy/install/pendulum_control/bin
$ ./pendulum_logger
# ... 내용 생략 ...
Commanded motor angle: 1.570796
Actual motor angle: 1.541950
Current latency: 99387 ns
Mean latency: 105697.995992 ns
Min latency: 53414 ns
Max latency: 170257 ns
Minor pagefaults during execution: 0
Major pagefaults during execution: 0
```

pendulum_demo 노드를 종료한 다음 output.txt 에 저장된 실시간성에 대한 결과를 확인해보자.

PendulumMotor received 553 messages
PendulumController received 967 messages

```
$ cat output.txt
Initial major pagefaults: 16
Initial minor pagefaults: 2124638
rttest statistics:
  - Minor pagefaults: 0
  - Major pagefaults: 0
  Latency (time after deadline was missed):
    - Min: 53414 ns
    - Max: 170257 ns
    - Mean: 105750.270000 ns
    - Standard deviation: 33639.231572
```

- 페이지 폴트는 나타나지 않지만, 최소 레이턴시와 최대 레이턴시가 크게 차이나는 모습을 확인할 수 있다. 이는 커널 수정 없이 데모 코
- 이를 해결하기 위해 관리자 권한으로 /etc/security/limits.conf 파일 가장 아랫줄에 우선순위 옵션 (${username} - rtprio 98)을 추가하
- 우선순위는 0-99까지 설정 가능하지만 가장 높은 우선순위는 중요한 시스템 프로세스가 가지고 있기 때문에 그보다 하나 낮은 98 로 설
- 변경된 우선순위로 인한 실시간성의 변화를 쉽게 확인하기 위해 그래프를 그려보도록 하자. pendulum_demo 에 -f 매개변수를 추가하

```
$ . ~/ros2_foxy/install/local_setup.bash
$ cd ~/ros2_foxy/install/pendulum_control/bin
$ ./pendulum_demo -f pendulum_demo_results
$ rttest_plot pendulum_demo_results
```