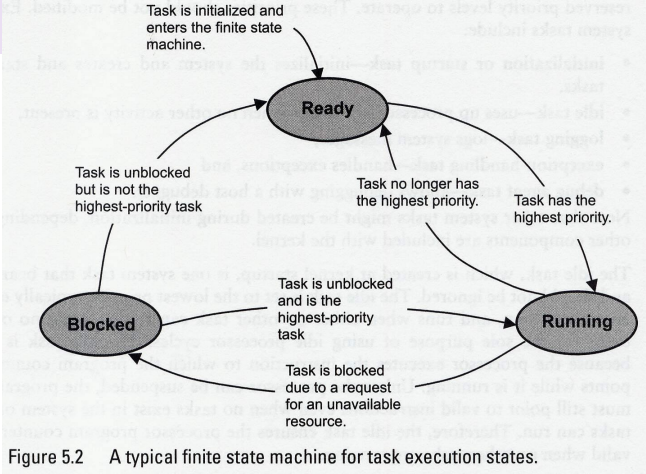
|  |  |
| --- | --- |
| 년도-학기 | 2021년 2학기 |
| 과목명 | 자동화프로그래밍 |

|  |  |
| --- | --- |
| **번호** | **실험 제목** |
| LAB 8 | FreeRTOS |

|  |  |
| --- | --- |
| 실험 일자 | 2021년 11월 9일 |
| 제출자 이름 | 강\*\* |
| 제출자 학번 | 201803\*\*\*\* |
| 팀원 이름 |  |
|  |  |

**Chapter 1. 관련 이론(Theoretical Background)**

Task는 실행의 가장 기본적인 단위를 말한다. ready, running, blocked의 3가지 state로 구분되어 다음과 같은 동작을 보인다.



Task는 초기화되고 준비상태에 들어간다.

Task가 unblocked 되었지만 가장 높은 priority가 아니다.

Task가 resource를 요청하고 기다리는 blocked 상태가 된다.

Task가 unblocked 되고 가장 높은 priority이다.

Task가 가장 높은 priority를 가진다.

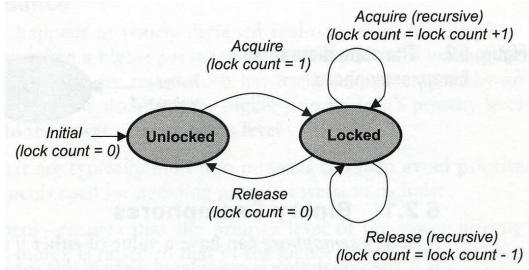
Task의 priority가 가장 높지 않다.

Semaphore는 일종의 열쇠와 같은 역할을 한다. Semaphore를 획득 가능한 상태를 available이라고 하며 semaphore를 더 이상 획득하지 못하는 상태를 unavailable 상태라고 한다. 어떤 task가 semaphore를 release해야 다시 available 상태로 돌아간다. Semaphore가 여러 개인 counting semaphore의 동작 구조는 다음 그림과 같다.



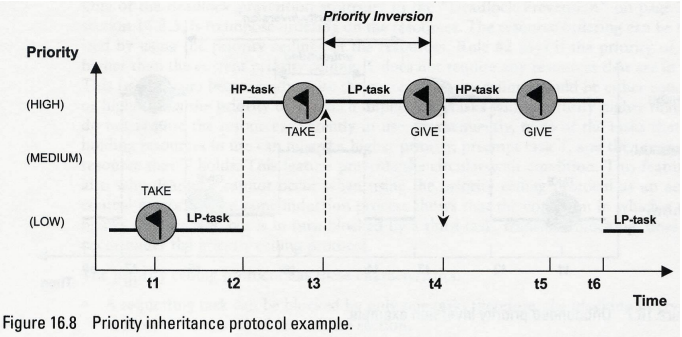
가진 semaphore를 task가 acquire 할 때마다 count를 감소시키고, release 될 때마다 count를 증가시키며 상태를 확인할 수 있다. Binary semaphore는 counting semaphore에서 semaphore가 하나, count하는 개수가 하나 인 경우를 의미한다.

Mutex는 잠금의 개념으로 이해할 수 있다. Semaphore와 같은 원리로 동작하지만ownership을 가져 locked(acquire)한 대상만이 unlocked(release)할 수 있다.



Semaphore와 mutex는 task 사이의 synchronization 역할도 하지만, 공유되는 resource에 접근하지 못하도록 막아주는 역할도 한다.

Priority inheritance는 low priority task가 mutex나 semaphore를 acquire 한 상태에서 high priority task가 mutex나 semaphore를 요청할 때, 일시적으로 low priority task가 high priority task와 같은 priority를 갖게 되는 것을 말한다. low priority task가 release 한 후 바로 high priority task가 acquire한다. 이 과정을 다음 그림과 같이 나타낼 수 있다.



**Chapter 2. 실험 결과(Experimental Results)**

[Exercise 1]

**텍스트, 실내이(가) 표시된 사진

자동 생성된 설명**

코드를 위와 같이 수정하였다.

텍스트, 전자기기, 회로이(가) 표시된 사진

자동 생성된 설명 텍스트, 전자기기, 회로이(가) 표시된 사진

자동 생성된 설명

코드 변경 전과 동일하게 초록색 led가 켜지고 빨간색 led가 켜진다. 0.5초마다 초록색과 빨간색 led가 번갈아 가며 켜지고 꺼진다. Task1과 2의 코드를 살펴보면 초록색 led는 켜짐 -> osDelay -> 꺼짐 -> osDelay, 빨간색 led는 꺼짐 -> osDelay -> 켜짐 -> osDelay 순서로 작성되어 있다. Task1의 priority가 task2보다 높아져도 osDelay에는 blocked 상태와 같으므로 초록색 led가 osDelay가 될 때 빨간색 led에 대한 동작이 실행 될 수 있다. 따라서 변경 전과 차이가 없다.

[Exercise 2]

텍스트이(가) 표시된 사진

자동 생성된 설명

Exercise 1의 상황에서 StartTask1 함수의 코드를 위와 같이 osDelay 대신 HAL\_Delay로 수정하였다.

텍스트, 전자기기, 회로이(가) 표시된 사진

자동 생성된 설명텍스트, 전자기기, 회로이(가) 표시된 사진

자동 생성된 설명

초록색 led만 0.5초마다 켜지고 꺼지며, 빨간색 led는 꺼져 있다. HAL\_Delay는 osDelay와 달리 Blocked 상태가 아니라 계속 실행 중인 상태이다. 또한 task1의 priority가 task2보다 높기 때문에 먼저 실행된다. 따라서 task1은 계속 실행 중이게 되고, 빨간색 led는 계속 ready 상태에 놓여있게 되어 켜지지 못한다.

[Exercise 3]

텍스트, 실내이(가) 표시된 사진

자동 생성된 설명

Exercise2에서 Task1의 priority를 Task2와 같도록 코드를 수정하였다.

텍스트, 전자기기, 회로이(가) 표시된 사진

자동 생성된 설명텍스트, 전자기기, 회로이(가) 표시된 사진

자동 생성된 설명

같은 priority를 가지면 Round robin 방식을 사용하기 때문에 exercies2와 다르게 Task1이 HAL\_Delay를 사용하더라도 Task1과 Task2가 번갈아가며 실행되어 초록색 led와 빨간색 led가 Exercise1과 같이 번갈아 가며 켜지고 꺼지는 것을 볼 수 있었다.

[Exercise 4]

텍스트, 실내이(가) 표시된 사진

자동 생성된 설명위와 같이 다른 task가 시작되기 전에 semaphore를 acquire하도록 코드를 수정하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

좌측 사진은 코드를 수정하기 전 결과이고 우측 사진은 수정 후의 결과이다. 다른 task가 시작되기 전에 semaphore를 acquire하도록 코드를 수정하였기 때문에 task2가 바로 semaphore를 acquire하지 못하고, task1이 semaphore를 release한 후 task2가 semaphore를 acquire하게 된다.

[Exercise 5]

텍스트이(가) 표시된 사진

자동 생성된 설명

Task가 시작되기 전에 semaphore 2개를 먼저 acquire하도록 코드를 수정하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

따라서 좌측의 수정 전 결과와 다르게 우측 수정 후 결과를 보면, 코드를 수정하여 2개의 semaphore가 먼저 acquire 되었으므로 task3는 task1,2가 semaphore를 release한 뒤에 task3이 semaphore를 acquire 하는 것을 확인할 수 있다.

[Exercise 6]

텍스트이(가) 표시된 사진

자동 생성된 설명

Task1의 코드에서 mutex에 대한 부분을 삭제하였다.

텍스트, 컴퓨터이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

코드 수정 전(좌측 사진)에는 task1은 mutex lock -> before값 출력 -> osDelay -> after값 출력 -> unlock -> osDelay, task2는 mutex lock-> shared\_data 증가 -> unlock 이므로 공유되는 resource에 서로 접근할 수 없다. 따라서 before 값과 after 값을 출력하는 동안 resource data의 값에는 변화가 없다.

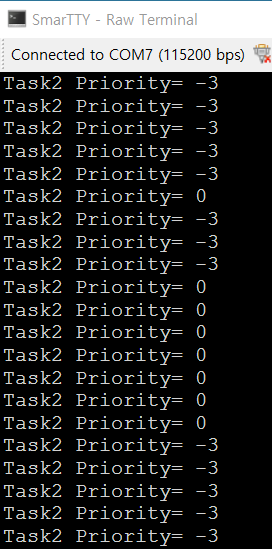
하지만 코드를 수정하면 before 값과 after 값을 출력할 때 mutex를 필요로 하지 않으므로 출력을 하는 상황에서도 shared data는 계속 증가하게 된다. 따라서 우측 사진에서 출력할 때마다 값이 계속 변화하는 모습을 볼 수 있다.

[Exercise 7]

텍스트이(가) 표시된 사진

자동 생성된 설명

osMutexWait 함수의 두번째 입력 값은 timeout 값이다. 1000일 경우에는 1000ms 동안 기다린 후 정상 실행이 안되면 다음으로 진행하고, -1 일 경우는 영원히 기다린다. Exercise7에서는 이 값을 1000에서 -1로 변경하였다.

 텍스트이(가) 표시된 사진

자동 생성된 설명

버튼을 누르면 priority가 0인 task1이 mutex를 요청한다. Priority가 -3인 task2는 priority inheritance에 의해 task1의 priority와 동일한 priority를 갖게 된다. 하지만 task2는 mutex를 release하지 않는다.

여기서 수정 전(좌측 사진)에서는 task1이 1초 동안 mutex가 release되는 것을 기다리고, acquire하지 못하면 다시 처음으로 돌아가 반복하게 된다.

하지만 수정 후(우측 사진)에는 버튼을 누르고 나면 task1이 mutex가 release 되기를 영원히 기다리기 때문에 task2의 priority가 0인 상태가 지속된다.

**Chapter 3. 결론 및 Discussion**

이번 실습은 크게 어려운 점 없이 진행할 수 있었다. 실습을 해보며 semaphore와 mutex에 대한 이해도를 높일 수 있었다. acquire하고 release하는 개념 자체는 쉽게 이해할 수 있었지만, 막상 실습을 해보니 조금 다르게 알고 있었던 것을 제대로 알 수 있었다. Preemptive inheritance에서 low priority task가 semaphore나 mutex를 가지고 있을 때, high priority task가 요청하면 high priority task가 우선시되어 실행되는 것으로 착각하고 있었다. 하지만 과제를 진행해보며 다시 확인해보니 low priority task의 실행이 끝나고 release 한 다음에 high priority task가 실행되는 것이었다. 잘못된 이해를 바로잡을 수 있어 다행이었다.

**Appendix: 없음**