과제 시작 전, fifo_scheduler 이해하기. (전 과제를 참조하는 경우가 있어서 다음 과제 때 다음 번 빠른 이해를 위해 따로 작성하였습니다)

fifo schedule에서는 프로세스가 현재 wait의 상태면 다음 번 프로세스로 넘어가기 위한 선택과정을 위한 코드로 넘어간다. 프로세스의 lifespan이 아직 진행된 tick의 수인 age보다 커서 일할 거리가 남아 있는 상태라면 현재 프로세스를 유지하도록 한다.(non-preemptive하기 때문에) fifo에서 다음에 수행할 프로세스 선정 과정에서는 수행될 프로세스들이 readyqueue에 들어있고 이 readyqueue가 비어 있지 않다면 수행 되어야할 프로세스들이 남아있다는 뜻으로 조건으로 설정 되어있다. next는 다음 프로세스를 선택하기위한 변수로 readyqueue에는 list_add_tail로 들어온 프로세스 순으로 처리하는 queue의 형태이기 때문에 first부분을 가져와 process의 들어온 순번대로 되어있는 readyqueue에서 다음에 들어온 process를 선택하기 위한 list_first_entry로 다음 process를 지정해준다. 후에 선택된 프로세스는 readyqueue에서 빼내기 위한 list_del_init를 사용한다. list_del과 차이는 리스트에서 빼낸 이 항목을 더 이상 사용하지 않는다는 이유를 가지고 list_head를 재초기화 하는 차이를 가진다.

fcfs_acquire함수는 만약 i번 리소스의 현재 owner가 없는 경우라면 current를 owner로 설정해주고 owner가 있는 경우라면 프로세스를 PORCESS_WAIT상태로 설정한다. 그리고 readyqueue에서 나온 것을 waitqueue에 그 프로세스를 넣어 해당 프로세스가 i번 리소스의 사용을 기다리고 있음을 알려준다. fcfs_release함수는 어떤 프로세스가 해당 리소스를 모두 사용하면 owner를 null로 바꿔주고 waitqueue다음번에 줄 서 있는 프로세스를 waiter변수를 통해 설정해주고 waitqueue에서 지워주며 상태를 ready로 바꿔준 뒤 readyqueue로 넘겨준다.

1. Description how each scheduling policy is implemented

- **SJF scheduler**

fifo와 마찬가지로 non-preemptive의 특징을 가지고 있기 때문에 현재 프로세스가 wait의 상태면 바로 다음 프로세스 선택을 위한 부분으로 넘어가주고 age보다 lifespan이 커서 현재프로세스가 주어진 일을 마치지 않았으면 non-preemptive이기에 current를 반환해서 계속 현재상태를 유지하도록 한다. 다음 프로세스를 선택하는 부분에서 next에 readyqueue에 대기중인 다음 프로세스를 선택하고 readyqueue를 iteration으로 순회하는 pos라는 임시 프로세스 선택변수를 생성하여 비교를 해서 next를 변경해주도록 하였다. 이를 위해 list_for_each_entry를 사용하였다. sjf는 일이 짧은 프로세스를 먼저 선택하도록 하므로 각 프로세스의 lifespan을 비교하여 짧은 것으로 next프로세스를 변경하도록 하였다. 후에 fifo와 마찬가지로 선택된 프로세스는 readyqueue에서 delete하였고 next를 반환해주었다.

- **SRTF scheduler**

srtf는 sjf에서 non-preemptive한 성격을 preemptive하게 바꿔준 것이다. 프로세스의 lifespan이 작은 순으로 선택해 붙잡고 있는 것이 아니라 수행할 때마다 남은 일의 양을 비교해 프로세스를 선택하는 preemptive한 성격을 가진다. 따라서 현재 프로세스가 wait의 상황이라면 동일하게 다음 프로세스를 선택하기 위한 과정으로 넘어가고 preemptive하기 때문에 일이 남아있는 상태라면 현재프로세스를 반환하면서 유지하는 것이 아니라 readyqueue의 가장 마지막으로 보내도록 하기위해 list_add_tail을 사용하였다. pick_next부분은 마찬가지로 readyqueue가 비어 있지 않는 경우 수

행하도록 하였으며 next를 먼저 readyqueue에 대기중인 첫번째 것을 선택하고 임의로 pos를 만들어 readyqueue를 순회하면서 남은 일의 양을 비교하여 더 적은 것을 선택하여 next를 바꿔주도록 하였다.

- **RR scheduler**

round robin방식은 일을 정해진 quantum만큼 돌린 뒤 다음 프로세스를 선택하는 식으로 진행된다. 따라서 일은 전체적으로 readyqueue에 들어온 순서대로 진행되며 우리 과제에서 quantum은 1 tick이기 때문에 한번 수행되면 나머지 프로세스들이 전부 수행된 뒤에 수행되어야 하므로 일이 남아있는 경우에 list_add_tail로 readyqueue의 마지막으로 보내도록 하였다. 나머지는 들어온 순서대로 일을 처리하는 방식인 fifo의 스케줄링과 같은 방식이기 때문에 코드를 건드리지 않았다.

- **Priority scheduler**

priority scheduler는 우선순위를 비교하여 next를 결정한다. 우선 일이 들어온 순서대로 readyqueue에 쌓이고 readyqueue중에서 prio를 비교하여 더 큰 값을 가진 prio가 우선순위를 갖기 때문에 비교할 수 있도록 했다. 프로세스가 wait이면 바로 다음 프로세스를 선택하는 부분으로 넘어가는 것은 동일하며 프로세스의 일이 남아있는 상태라면 일을 하고나서 list_add_tail로 readyqueue의 가장 마지막에 다시 붙여주도록 하였다. queue의 마지막에 더해주는 이유는 만약 같은 우선순위일 때 일을 tick단위로 번갈아가며 실행하도록 하라는 조건이 있기 때문이다. 즉, round robin을 base로한 priority 스케줄러이기 때문이다. 후에 next를 선택하는 부분은 next와 readyqueue를 순회하는 pos를 이용하여 우선순위를 비교해주고 만약 어떤 프로세스가 우선순위가 next보다 높다면 next를 그 프로세스로 변환해주고 반환하도록 하였다.
priority scheduler부터는 resource 사용을 요구하기도 한다.(전의 다른 스케줄러도 요구 가능하지만 우리 과제의 테스트케이스를 기준으로) 만약 acquire 1 2 4라면 이 프로세스는 readyqueue에 들어왔을 때부터 2tick 수행 뒤 1번 리소스를 4tick간 사용한다는 의미를 가진다. 여기서 priority scheduler에서는 리소스를 acquire할때는 fcfs_acquire와 달리 특정조건을 갖지 않기 때문에 fcfs를 복제한 prio_acquire와 fcfs_release에서 같은 리소스를 원하는 프로세스들이 watiqueue에서 기다리고 있을 때 우선순위가 더 높은 것에 먼저 그 리소스를 할당해주기 위한 조건으로 waitqueue를 list_for_each_entry로 순회하여 watier와 임의의 pos와 우선순위를 비교하여 더 높은 것을 선택하여 readyqueue로 보내주도록 추가적인 코드를 작성한 prio_release를 사용하였다.

- **Priority scheduler + aging**

aging기법은 우선순위 스케줄에서 우선순위가 낮은 것이 starvation이 되는 것을 막기 위한 것으

로 prio_schedule의 코드에 pos를 통해 순회를 할 때마다 readyqueue에 있는 모든 프로세스들의 우선순위를 1씩 높여주도록 하였다. 먼저 prio에 1씩 더해주며 그 pos와 next의 우선순위를 비교하여 pos의 우선순위가 더 높은 경우 next를 pos로 바꿔주도록 하였다. 후에 선택된 프로세스의 우선순위는 기다리는 동안 더해진 값을 다시 초기화 시켜줘야 하기 때문에 prio를 prio_orig으로 바꿔줄 수 있도록 하였다.

- **Priority scheduler + PCP**

PCP는 기존의 priority scheduler에서 프로세스가 resource를 잡을 때 우선순위를 확 높여줘서 priority inversion을 피하는 방식으로 prio_acquire에서 owner가 없어 설정될 때 resource를 잡은 현재 프로세스의 우선순위를 MAX_PRIO로 높여주도록 설정하였다. resource를 release할때는 MAX_PRIO로 높여준 우선순위를 원래 해당프로세스의 priority로 바꿔 주기 위해서 prio_release에서 현재 프로세스의 우선순위만 prio_orig으로 바꿔주는 코드를 삽입하였다. schedule의 경우 priority scheduler를 기반으로 하기 때문에 따로 정의하지 않았다.

- **Priority scheduler + PIP**

PCP와 마찬가지로 schedule의 경우 priority scheduler를 기반으로 하기 때문에 따로 정의하지 않았다. pcp와는 다르게 현재 내가 원하는 리소스를 다른 프로세스가 가지고 있을 때 그 프로세스의 우선순위가 낮아서 priority inversion이 발생할 수 있는 상황을 다른 프로세스의 우선순위를 나에게 맞춰줘서 inversion을 회피할 수 있도록 한다. 따라서 기존 prio_acquire함수에서 리소스를 소유하는 프로세스의 우선순위가 현재 프로세스의 우선순위보다 작은 경우 현재 프로세스의 우선순위로 소유하고 있는 프로세스의 우선순위를 변경해준 pip_acquire을 사용한다. release할 때는 owner를 해제하기 전에 소유하던 프로세스의 원래 우선순위로 다시 변경을 해주고 해제할 수 있도록 한 pip_release를 사용하도록 했다.

2. Show how the priorities of processes are changed over time for aging and PIP

- **Priority scheduler + aging, testcase prio (~12nd tick)**

프로세스는 총 6개이며 순서대로 10, 20, 15, 5, 30, 0의 우선순위를 갖는다.
Tick 0: 6개 process모두 tick0에서 start하며 우선순위가 가장 높은 process5가 선택되어 수행된다. 이때 선택 받지 못한 나머지 process들은 우선순위가 1씩올라 11, 21, 16, 6, 30, 1의 우선순위를 갖게 된다.
Tick 1: 우선순위를 비교했을 때 여전히 process 5가 가장 높기 때문에 선택된다. 우선순위는 12, 22, 17, 7, 30, 2가된다.

Tick 2: 우선순위를 비교했을 때 여전히 process 5가 가장 높기 때문에 선택된다. 우선순위는 13, 23, 18, 8, 30, 3이된다.
Tick 3: 우선순위를 비교했을 때 여전히 process 5가 가장 높기 때문에 선택된다. 우선순위는 14, 24, 19, 9, 30, 4가된다.
Tick 4: process 5는 lifespan을 모두 채웠기 때문에 사라지며 readyqueue에 있는 나머지 중 우선순위가 가장 높은 process 2가 수행된다. 우선순위는 15, 20, 20, 10, X(process 5), 5가 된다.
Tick 5: process2와 process3의 우선순위가 같지만 rr기반이기 때문에 process2는 Tick4 수행 후 readyqueue의 마지막으로 가고 process3가 선택되어 수행된다. 우선순위는 16, 21, 15, 11, X(process 5), 6이 된다.
Tick 6: 우선순위를 비교했을 때 process 2가 가장 높기 때문에 선택된다. 우선순위는 17, 20, 16, 12, X(process 5), 7이된다.
Tick 7: 우선순위를 비교했을 때 여전히 process 2가 가장 높기 때문에 선택된다. 우선순위는 18, 20, 17, 13, X(process 5), 8이된다.
Tick 8: 우선순위를 비교했을 때 여전히 process 2가 가장 높기 때문에 선택된다. 우선순위는 19, 20, 18, 14, X(process 5), 9가된다.
Tick 9: process 2는 lifespan을 모두 채워 사라지며 남은 것 중 우선순위가 가장 높은 process1이 선택된다. 우선순위는 10, X(process 2), 19, 15, X(process 5), 10이 된다.
Tick 10: 우선순위를 비교했을 때 가장 높은 process 3이 선택된다. 우선순위는 11, X(process 2), 15, 16, X(process 5), 11이 된다.
Tick 11: 우선순위를 비교했을 때 가장 높은 process 4가 선택된다. 우선순위는 12, X(process 2), 16, 5, X(process 5), 12이 된다.
Tick 12: 우선순위를 비교했을 때 가장 높은 process 3이 선택된다. 우선순위는 13, X(process 2), 17, 6, X(process 5), 13이 된다.

- **Priority scheduler + PIP, testcase resource-adv2 (~12nd tick)**

Tick 0: process 1만 존재하며 1, 2, 3, 4번 리소스를 바로 가지게 된다. Process 1 수행한다.
Tick 1: process 2가 start되며 1번 리소스 사용을 원하기 때문에 상태가 wait이되고 1번 리소스 waitqueue로 들어가게된다. 우선순위는 2가 1보다 높아 inversion방지를 위해 1의 우선순위를 잠시 2의 우선순위와 같게 만들어준다.
Tick 2: process 3, 4가 start하여 순서대로 readyqueue로 들어가며 process 4가 바로 1번 리소스 사용을 원하기 때문에 상태가 wait이되고 1번 리소스 waitqueue로 들어가게된다. Tick1과 마찬가지로 inversion방지를 위해 process4의 우선순위가 높아 process1의 우선순위를 4와 같게 해준다.
Tick 3: process1이 수행되며 tick0, 3에서 1번리소스를 총 2번 사용해 1번 리소스를 release한다.
Tick 4: 1번 리소스의 waitqueue에 process2와 4중 우선순위가 4가 높기 때문에 4에게 1번리소스를 주고 1번실행후 1번리소스를 다시 release한다.

Tick 5: process 4는 lifespan을 모두 채워 사라지며 readyqueue에서 우선순위가 높은 process3이 수행된다.

Tick 6: process 3이 우선순위가 가장 높아 한번 더 수행된다.

Tick 7: tick 5,6에서 process3이 2tick 수행되어 2번 리소스를 요구하지만 2번리소스는 process1이 사용하고 있어 2번 리소스의 waitqueue로 들어가게 된다. Process1의 우선순위를 process 3과 동일시 해준다.

Tick 8: 현재 1번 process의 우선순위는 2번 process보다 높기 때문에 먼저 수행된다. Tick 0, 3, 8에서 process1을 3번수행해 2번 리소스를 release한다.

Tick 9: 2번 리소스를 대기중이던 process 3이 2번리소스를 acquire해서 수행된다.

Tick 10: 한번 더 process3을 수행한다. Tick9, 10에서 2번리소스를 2번사용해 2번 리소스를 release한다.

Tick 11: process3은 tick5, 6, 9, 10에서 4번 수행해 lifespan이 모두 채워져 사라지고 1번리소스를 대기중이던 process2에게 acquire해준다. 그리고 수행한다.

Tick 12: 2번 process가 한번 수행되었기 때문에 2번 리소스를 acquire하며 process2를 수행하고 tick 11과 tick12에서 1번 리소스를 2tick실행해 반환해주고 tick12에서 2번 리소스를 1tick실행해 둘다 반환한다.

3. Lesson learned

- Pa1에서도 말했듯 queue를 구현하는데 list_add_tail이 훨씬 편함을 알 수 있었다. Pa1에서 알게 되어 이번과제에 정확히 적용할 수 있었던 것 같다.

- list_for_each_entry vs list_for_each_entry_safe
  list_for_each_entry는 iteration중 무언가 삭제되면 리스트 순회를 멈춰 문제를 야기할 수 있는 반면 list_for_each_entry_safe는 멈추지 않는 다는 것을 알게 되었다. 물론 이번 코드 중 readyqueue가 정제?된 상태에서 next를 고르는 식으로해 탈없이 구현이 가능했던 같지만 좀더 동적으로 readyqueue가 변하는 상태였다면 list_for_each_entry_safe가 훨씬 안전함을 알게 되었다.

- 전체적으로 프로세스를 스케줄링하는 방식이 조금씩 조금씩 변화했다는 것을 알게 되었다. Aging을 구현하던 도중 그 전까지는 각 scheduling을 각각 연관되지 않은 전혀 다른 방식들로 쳐다봐서 좀더 어렵게 다가갔었지만 aging을 고민하면서 priority에서 readyqueue를 순회할 때 aging해주면서 다가가면 코드의 많은 변화없이 스케줄링의 변화를 깨달았고 sjf는 fifo에서 선택하는 방식만 rr은 fifo에서 non-preemptive한 부분만 priority는 rr을 기반으로 우선순위를 비교하는 방식만 등등 이 여러가지 방식들이 조금씩 변해가면서 생겼음을 알게 되었고 pcp와 pip구현에는 많은 시간소비를 하지 않고 구현할 수 있었던 것 같다.