[SJF 구현법]

FIFO에서 순서대로 처리하는 것이 아닌, SJF에서는 lifespan이 짧은 것을 먼저처리하는 것으로 수정하면 되겠다는 생각을 하였다. 따라서 FIFO의 코드 중 현재 진행중인 것이 없는 경우 pick_next로 가고, 현재 진행중인 것이 있는 경우, 현재 것을 진행한다는 개념은 그대로 가져왔다. 다음 것을 고르는 pick_next에서 readyqueue가 비어있지 않은 경우, for문과 유사한 list_for_each를 통하여 readyqueue에 있는 process들 중 Lifespan이 가장 짧은 것을 next로 설정하는 코드를 만들어야 겠다는 생각을 하였다. 그리고 이 코드가 끝나면, next로 실행 될 것은 list를 삭제해주어야 하므로, list_del_init을 통해 next실행될 것의 list를 떼어냈다. 그리고 return next를 하였다. struct scheduler sjf_scheduler에는 .schedule방법으로 sjf_schedule을 이용하겠다는 코드를 추가하였다.

[SRTF 구현법]

이는 SJF의 preemptive한 버전이므로, 전체적인 틀은 SJF를 많이 참고하였다. 다음 것을 고르는 pick_next를, 원래 SJF에서는, current한 것이 실행중이지 않는 경우만 확인하고 goto pick_next하였지만, 실행중인 것이 있더라도 readyqueue가 비어있지 않다면, readyqueue 중 lifespan이 짧은 것이 있는지 확인하고, 짧은 것이 있다면 next를 그 아이로 바꿔주어야 한다. 따라서 실행중인 것이 없든지, 아니면 실행중인 것은 있으나 readyqueue가 비어있지 않은 경우는 pick_next로 가도록 하였다. readyqueue가 비어있지 않은 경우, current한 것이 없거나, 현재 실행중인 작업이 끝났으면, 다음 것을 골라주기 위하여 (lifespan) – (age), 즉 남은시간을 비교하여, 가장 짧은 것을 Next로 실행될 수 있도록 하고, list_del_init(&next->list)를 해주었다. 만약 현재 실행중인 것이 아직 다 실행되지 않았고, lifespan이 남은 경우, 가장 readyqueue에서 (lifespan – age)가 짧은 것을 골라내어, current의 남은시간과 비교해주고, current의 남은시간보다 남은 시간이 짧은 경우 current는 list_add_tail로 list에 추가해주고, 다음에 실행될 next의 list는 떼어주었다. current가 더 짧으면 next=current로 하였다. 그리고 return next하였다. readyqueue가 비어있고, 현재 진행중인 것이 실행이 끝나지 않았다면, current한 것을 실행하도록 하였다.

struct scheduler srtf_scheduler에는 .schedule방법으로 srtf_schedule을 이용하겠다는 코드를 추가하였다. 이는 waiting queue가 비어있지 않고, lifespan이 남은 경우를 따로 else if해주지 않고, else로 waiting queue가 비어있지 않고, 현재 실행중인 프로세스가 있다는 것을 한번에 묶어 버리는 경우 , 제대로 작동하지 않는 것을 볼 수 있었다. 이는 dump_status로 확인한 결과, 이미 실행이 끝난 것이 제대로 처리되지 않아서 발생하는 문제였다. 따라서 경우를 제대로 나누어주어야 하고, preemptive되는 경우, list_del_init을 꼭 해주어야 한다는 것을 알 수 있었다.

[RR 구현법]

RR은 FIFO를 1quantum마다 다른 것을 실행되도록 응용시킨 것이라고 생각하였다. 따라서 FIFO의 기본틀에서, 현재돌아가고 있는 것이 없거나, Readyqueue가 비어있지 않은 경우, 다음에 무엇을 실행할지 pick_next로 가서 골라주어야 하므로, 이를 한 경우로 묶었고, 현재 실행중인 것이 있지만 Readyqueue가 빈 경우, 현재 것을 계속 실행하는 게 하였다. pick_next한다면, readyqueue가 비어있지 않은 경우, next는 FIFO방식을 따르므로, 가장 앞에 것이다. 현재 실행하고 있는 것이 있고, 그 실행중인 것이 아직 lifespan이 남아있다면, 실행중인 것은 readyqueue 맨 뒤에 다시 붙여준다. 이는 circular FIFO queue방식이므로, 이러한 생각을 하게 되었다. 그리고 다

음에 실행될 것은 list에서 떼어내야하므로 list_del_list를하고 next는 return하였다.
struct scheduler rr_scheduler에는 .schedule방법으로 rr_schedule을 이용하겠다는 코드를 추가하였다. 이는 FIFO와 circular queue의 응용이여서 가장 적은 시간이 걸렸던 것 같다.

[priority 구현법]
이는 SRTF가 SJF을 preemptive하게 만든 것이라는 것을 응용하여, SRTF에서 남은 시간을 계산하는 부분을 prio 을 비교하는 방법으로 바꿔서 생각하면 어떨지에 대해서 생각을 해보게 되었다. 내가 구현한 SRTF에서 남은 시간을 계산하던, (실행해야되는 것) – (이미 실행한 것) 부분을 prio으로 바꾸고, 부등호의 방향만 바꾸어주었더니 바로 돌아갔다.
즉, readyqueue가 있는 경우와 readyqueue가 없는 경우로 나누어 주고, readyqueue가 없는 경우, 현재 실행중인 것을 계속 실행시켰다. readyqueue가 있는 경우, 현재 실행되고 있는 것이 없거나 실행되던 것이 종료된 경우, 새로운 Priority높은 애를 돌리도록 하였고, 현재 실행되던 것이 아직 lifespan을 다 돌지 못한 경우, priority가 높은 것을 찾았다. 이 때, priority가 current이 가장 높으면 next는 current이 되는 것이고, readyqueue에 있던 process가 priority가 높은 경우에는 next는 이것이 되고, 돌고 있던 것은 list에 추가하는 형태로 구현하였다. 여기서 priority가 같은 것은 RR 형식으로 작동한다 하였으므로 '='같은 priority가 있는 경우, 1tick 마다 preemptive되도록 구현하였다.
struct scheduler prio_scheduler에는 .schedule방법으로 prio_schedule을 이용하겠다는 코드를 추가하였다. 이는 acquire과정은 fcfs와 다르지 않아도 된다 생각하여 prio_acquire는 fcfs와 동일하게 하였고, prio_release의 경우는 waiter를 waitqueue 첫번째 것으로 지정하는 것이 아닌, 가장 prio가 높은 것으로 지정하였다. 이는 pip구현 후 느낀 것이므로, Pip구현에서 자세히 말하고자 한다.

[priority + PCP 구현법]
struct scheduler pcp_scheduler에는 schedule방법으로 prio_schedule을 이용하겠다는 코드를 추가하였다. 이는 acquire과 release과정에서 수정해줘야 할 것이 있다고 생각하여 bool pcp_acquire과 void pcp_release를 만들어 수정하였다.
pcp_acquire에서는 resource를 잡고 있는 것이 없는 경우, 현재 process가 resource의 owner가 된다는 fcfs에서 r->owner의 priority를 MAX_PRIO로 높여주었다. PCP는 resource를 잡는 순간 release할 때까지, 다른 것들이 뺏어가지 못하게 하므로 최댓값인 MAX_PRIO로 올려주는 방법이기 때문이다. release할 때는 r->owner에 NULL값을 넣기 전에 그 프로세스의 priority를 원래대로 복구시켜주는 r->owner->prio = r->owner->prio_orig의 과정을 거쳤다.

[priority + PIP 구현법]
pcp와 비슷하게 접근하였다. struct scheduler pip_scheduler에는 schedule방법으로 prio_schedule을 이용하겠다는 코드를 추가하였다. 이는 acquire과 release과정에서 살짝 수정해줘야 할 것이 있다고 생각하여 bool pip_acquire과 void pip_release를 만들어 수정하였다.
위에서 pcp_acquire에서 리소스 잡은 것의 Priority를 MAX_PRIO로 올려주었던 것을 삭제하고, pip_acquire에서는 r->owner->prio보다 priority가 더 높은 것이 와서 같은 리소스를 얻으려 하게 되면, r->owner의 priority는 더 높여줘야 한다고 생각하였다. 따라서 r->owner->prio가 current-

>prio보다 작은 경우 r->owner->prio를 current->prio값으로 대체하는 코드를 넣어주었다. pip_release는 pcp_release와 동일하게 구현하였다. 하지만 이렇게 하게 되면, 히든인풋에서 알맞은 값을 얻지 못한다.

나는 이유를 찾기 위해 handout을 다시 읽었는데, 완벽한 점수를 얻기 위해서는 release하는 경우, 리소스 확보상태를 확인하여 프로세스의 현재 우선순위를 계산하여야 한다는 힌트가 적혀있었다. 이 의미를 알기 위해 계속 생각한 결과, release하는 경우, resource의 waitqueue에서 가장 prio가 높은 것을 waiter로 지정하여, 이 리소스를 받기 위해 기다리던 여러가지 process중 우선순위가 높은 것을 readyqueue안에 들어가 실행 준비를 시켜야겠다는 생각을 하였다. 내가 실시간 수업 때 질문했던 resource의 waitqueue에서 나가는 순서가 있냐는 질문에 교수님께서 보통은 fcfs이지만 prio에서는 prio를 고려할 수 있다는 답변이 생각났다. 이 생각을 바탕으로 구현한 결과 히든인풋이 전부다 돌아갔다. 나는 이 pip_release에서 생각난, waitqueue에서 가장 prio가 높은 것을 waiter로 지정하여 readyqueue에 넣는 방식을, 우선순위를 적용한 prio_release, pcp_release에 모두 추가해주었다. 이게 가장 많은 생각을 요구했고, 이 생각을 적용하여 돌아갔을 때 말도 안되게 기뻤던 것 같다.

[느낀점 및 배운점]

과거에 fork의 강의노트를 우분투에서 실행해보고 실행 순서에 대해 질문했는데, 실행결과는 CPU scheduling 순서을 적용하여 나온다고 하였다. 나는 이때 CPU scheduling이 궁금했었다. CPU scheduling을 1부터 4까지 배우면서, 하나하나 방법들이 추가되는 것을 보고, 점진적으로 발달해간다는 생각을 하였었는데, 이번에 코드를 구현해본 결과 정말 이것을 몸소 느낄 수 있었다.

'**SJF**는 FIFO에서 readyqueue 순서대로 실행시켜주는 것이 아닌, lifespan이 짧은 것부터 시작하자.

SRTF는 SJF의 preemptive버전이니까 SJF에서 lifespan이 아닌 남은 시간을 비교해주고, preemptive를 적용가능하도록 하기 위해 readyqueue가 비어있지 않으면 남은 시간이 짧은게 있나 찾아보자.

RR은 quantum은 1이므로,FIFO 반복느낌으로 tick이 흐를 때마다 readyqueue순서대로 실행하자.

priority는 우선순위를 적용하므로, srtf에서 남은시간 비교하는 것이 아닌, 우선순위를 비교하자.

pcp는 acquire에서 resource를 얻으면 prio를 무지하게 큰 양으로 점프시키고, release에서 원래 prio로 복구하자.

pip는 pcp의 acquire에서 prio를 무지하게 큰양으로 점프시키지말고, 현재 그 리소스를 잡은 것이 prio가 높다면 그 아이의 prio를 resource owner에 적용시켜주자, release에서 원래 Prio로 복구하고 wait중인 것들 중 prio가 가장 높은 것을 ready상태로 만들어주자.'

과제가 나오기 전에, scheduling을 복습하면서 서로의 연관성을 느꼈는데, 과제를 진행하다보니 이러한 서로서로의 연관관계가 너무 잘 느껴졌다.

scheduling code를 짜기 위한 지식은, 이번 수업에서 예시를 풍부하게 들어서 이해가 잘되었던 것이 큰 뒷받침이 되었다고 생각한다. 앞으로도 지금처럼 예시가 많은 수업이라면, 이해에 도움이 많이 될 것 같다는 생각을 하였다.

linked list를 struct list_head를 이용하여 처음으로 해보았는데, 사이트와 기존의 코드를 보면서 이해할 수 있게 해주셔서 이해가 잘되었고, 이는 굉장히 간단하게 linked list를 처리하는 것 같아 보여서 익숙해진다면 더 편리할 것 같다는 생각이 들었다.