- Description how each scheduling policy is implemented

1. SJF-schedule

readyqueue 리스트를 순회하며 lifespan이 가장 짧은 process를 찾아 current process로 실행하는 방법으로 SJF를 구현하였다. 또한 non-preemptive 스케줄링이기 때문에, 새로 생성된 프로세스는 다른 처리없이 ready queue에 바로 넣어주어 current process의 실행이 종료되면, ready queue에서 SJF방법으로 다음 실행할 process를 찾아 이를 반복하도록 하였다.

2. SRTF-schedule

첫 실행을 하는 current process는 readyqueue에 있는 process 중 lifespan(=lifespan-0(시작할 때의 age))이 가장 짧은 process로 정해주었고, 새로운 프로세스가 생성될 때마다 current process와 생성된 process의 프로세스 실행 종료까지의 remain time (= (lifespan) - (age))을 비교하여 더 작은 값을 가지는 process가 이어서 실행되게끔 구현하였다. 즉, 프로세스가 생성될 때마다 실행종료까지의 remain time이 가장 적은 프로세스를 찾아 먼저 프로세스를 실행하도록 구현하였다. SRTF 스케줄링의 경우, 새로운 프로세스가 생성될 때를 제외하고는 current 프로세스가 가장 적은 remain time을 가지고 있으므로, 새로운 프로세스가 생성될 때에만 remain time을 비교해주어, 생성된 프로세스가 더 적은 remain time을 가지는 경우에 실행하던 프로세스를 readyqueue에 다시 넣어주고, 생성된 프로세스를 current로 하여 실행시킬 수 있도록 구현하였다.

3. RR-schedule

readyqueue에 대기중인 프로세스들을 순서대로 뽑아 1tick을 실행하고, 다음 프로세스를 또 뽑아 1tick을 실행하며 모든 프로세스가 종료될 때까지 반복하도록 구현하였다. 즉, process의 age나 remain time등에 상관없이 ready queue에 있는 모든 프로세스가 순서대로 선택되어 1tick씩 실행되고, 선택되어 1tick이 실행된 프로세스는 (실행이 종료되지 않았다면) 다시 readyqueue에 넣어져 다른 프로세스들이 모두 1tick씩 수행되면 다시 1tick을 실행하도록 기다리도록 구현하였다.

4. PRIORITY-schedule

priority scheduling은 두 가지 경우에 프로세스간 priority를 비교하는데, 새로운 프로세스가 생성되었을 경우와 그렇지 않을 경우이다. 단순 priority scheduling은 처음 process에게 주어진 priority가 중간에 변하지 않기 때문에, current 프로세스를 실행하는 도중에는 새로운 프로세스가 생성되는 경우에만 새로 생성된 프로세스와 current 프로세스와 priority를 비교하여, 더 낮은 priority를 갖는 프로세스는 ready queue에 넣고, 더 높은 priority를 가진 프로세스를 이어서 실행시키면 된다. 만약 새로운 프로세스가 생성되지 않는다면, current process가 종료되었을 때, readyqueue에서 priority가 최소가 되는 프로세스를 찾아 실행시키면 된다. (다만 본 과제에서는 PIP, PCP를 동일한 prio_schedule()함수를 통해 구현하였으므로, 위의 상황뿐만 아니라, 예외적인 상황도 고려해 구현해주었다. 해당 설명은 PCP, PIP에서 할 것이다.) 또한 priority가 같은 process가 존재할경우에는 두 process가 각 1tick씩 번갈아가며 실행을 하도록 하였다.

priority scheduling에서 process가 resource를 acquire/release하는 방법은 FCFS에서의 acquire/release방법과 거의 동일한데, 주어진 resource를 어떤 프로세스도 사용하고 있지 않는다면, resource의 owner를 resource를 요청한 프로세스로 설정하고, acquire를 요청한 resource가 이미 다른 프로세스가 사용하고 있는 경우에는 해당 resource에 대한 waitiqueue에 resource를 요청한 프로세스를 집어넣는 것은 FCFS에서의 acquire방법과 동일하다. 여기에 waitqueue에 들어있는 process를 구분하기 위해 mark[]배열을 만들어 (process_pid)번째 process가 waitqueue에 들어있는지를 mark에 표시해주었다. (ex. pid가 3인 프로세스가 waitqueue안에 들어있다면,

mark[3]=1이고, 들어있지 않다면 mark[3]=0이다.) resource를 release하는 것도 FCFS의 release 방법과 거의 동일한데, waitqueue안에 process가 대기하고 있다면, waitqueue에서 프로세스를 뽑아 readyqueue에 다시 넣어주어 실행할 수 있도록 한 방법은 동일하다. 하지만, priority scheduling에서는 waitqueue에 들어온 순서가 아닌, priority가 높은 순서대로 프로세스가 실행되어야 하므로, waitqueue에서 process를 순서대로 꺼내는 것이 아니라, waitqueue를 순회하며 가장 높은 priority를 갖는 process를 찾아 꺼내어 readyqueue에 넣어주어 실행할 수 있게 추가 구현하였고, waitqueue에서 process를 꺼냈으므로 해당 process에 대한 mark[process_pid]=0으로 바꾸어주었다.

5. PA-schedule

전체적으로 priority schedule과 비슷하게 구현되었다. 매 프로세스가 새로 생성될 때마다 priority를 current와 비교하여 preempt 할 수 있도록 구현하였고, 새로 생성되는 프로세스가 없다면 readyqueue에서 priority가 최대가 되는 프로세스를 찾아 실행시키도록 하였다. priority schedule과의 차이점은 매 tick마다 current process의 priority는 본래 처음에 주어진 priority로 reset해주었고, 나머지 readyqueue를 순회하며 readyqueue안에 있는 모든 프로세스의 priority를 1씩 증가시켜 주었다. 또한 normal priority scheduling과 다르게 새로 생성된 process가 없더라도, 각 프로세스의 priority가 매 tick마다 변경되기 때문에, 매 tick마다 current process와 readyqueue안의 모든 process의 priority 중 minimum을 갖는 process를 찾아 실행할 수 있도록 구현해주었다.

6. PCP

전체적인 스케줄링 방법은 priority scheduling을 기반하고 있으므로, acquire/release 부분만 PCP를 적용할 수 있도록 만들어주었다. PCP는 process가 resource를 갖는 즉시 해당 프로세스의 priority를 maximum으로 높여 주어야하므로, 앞서 구현한 priority scheduling의 acquire방법에서 resource의 owner를 정해줄 때, resource를 acquire하는 프로세스의 priority를 MAX_PRIO로 설정해주었다. 또한 resource를 반환할 때에는 priority를 높여주었던 프로세스의 priority를 원래 priority로 변경해주어야 하므로, priority release방법에서 release하는 프로세스의 priority를 해당 프로세스의 prio_orig으로 set해주었다. 또한 priority scheduling을 할 때, current의 priority가 원래대로 바뀐 process의 priority도 다시 비교하여 스케줄링 할 수 있도록 change[]배열을 통해 (process_pid)번째 프로세스의 priority가 바뀌었음을 표시해주었다. (ex. 2번째 프로세스의 priority가 변경되었을 때, change[2]=1, priority가 변하지 않았을 때, change[2]=0) 물론 PCP_release함수에서 표시한 change를 고려할 수 있도록 priority schedule함수에도 change[pid]가 1이면 바뀐 priority를 다음에 실행할 process와 비교해주어 해당 프로세스를 계속 실행할 것인지, 더 높은 priority를 가진 프로세스를 실행할 것인지 결정해주도록 구현하였고 이후 다시 change[pid]를 0으로 reset하도록 하였다.

7. PIP

PIP또한 전체적인 스케줄링 방법은 priority scheduling을 기반하고 있으므로, acquire/release 부분만 PIP를 적용할 수 있도록 만들어주었다. PIP는 low-priority process에 의해 이미 잡혀있는 resource를 high-priority 프로세스가 요청했을 때, resource를 잡고 있는 process의 priority를 resource를 새롭게 요청한 high-priority 프로세스의 priority로 설정해주어야 하므로, 앞서 구현한 priority scheduling의 acquire방법에서 이미 잡혀있는 resource를 요청한 프로세스를 waitqueue에 넣을 때에, resource owner process의 priority를 새로 resource를 요청한 process의

priority로 변경해주도록 하였다. 즉, waitqueue에 process가 들어가기 전에 waitqueue에 들어갈 프로세스의 priority를 resource의 owner의 priority로 설정해주었다. PIP에서의 release방법은 PCP와 동일하게 resource가 release될 때, owner였던 프로세스의 바뀌었던 priority를 원래대로 바꾸어주면 되므로, PCP에서의 release 함수를 그대로 사용하였다.

- Show how the priorities of processes are changed over time for aging and PIP. Use prio testcase for PA scheduler, and resources-adv2 for PIP.

< prio testcase for PA >

- 각각의 process 상태는 #tick이 실행된 후입니다.

# tick	current process	process in Ready queue(age/lifespan/priority)	설명
0	P5(1/4/30)	P1 (0/4/11) P2 (0/4/21) P3 (0/4/16) P4 (0/4/6) P6 (0/4/1)	P5가 가장 높은 priority를 가졌으므로 current가 되고, readyqueue에 있는 나머지 Process들의 priority는 모두 1씩 증가하였다.
1	P5 (2/4/30)	P1 (0/4/12) P2 (0/4/22) P3 (0/4/17) P4 (0/4/7) P6 (0/4/2)	tick 1과 동일
2	P5 (3/4/30)	P1 (0/4/13) P2 (0/4/23) P3 (0/4/18) P4 (0/4/8) P6 (0/4/3)	tick 2와 동일
3	P5 (4/4/30)	P1 (0/4/14) P2 (0/4/24) P3 (0/4/19) P4 (0/4/9) P6 (0/4/4)	tick 3과 동일
4	P2 (1/4/20)	P1 (0/4/15) P3 (0/4/20) P4 (0/4/10) P6 (0/4/5)	P5가 종료되고, readyqueue에서 가장 높은 priority를 가지고 있던 P2가 current가 되었고, P2의 priority의 원래 priority로 reset되었다. 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.
5	P3 (1/4/15)	P1 (0/4/16) P4 (0/4/11) P6 (0/4/6) P2 (1/4/21)	P2와 P3의 priority가 같아져 P3가 current가 되었고(by RR), P3의 priority의 원래 priority로 reset되었다. P2는 readyqueue에 들어갔고 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.
6	P2 (2/4/20)	P1 (0/4/17) P4 (0/4/12)	P2가 P3보다 priority가 높아져 current가 되었고, P2의 priority가 원래 priority로 reset되었다. P3은

		P6 (0/4/7) P3 (1/4/16)	readyqueue에 들어갔고 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.
7	P2 (3/4/20)	P1 (0/4/18) P4 (0/4/13) P6 (0/4/8) P3 (1/4/17)	여전히 P2의 priority가 가장 크므로 실행되고, readyqueue의 process들의 priority는 1증가되었다.
8	P2 (4/4/20)	P1 (0/4/19) P4 (0/4/14) P6 (0/4/9) P3 (1/4/18)	tick 8과 동일
9	P1 (1/4/10)	P4 (0/4/15) P6 (0/4/10) P3 (1/4/19)	P2가 종료되고, readyqueue에서 가장 높은 priority를 가지고 있던 P1이 current가 되었고, P1의 priority의 원래 priority로 reset되었다. 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.
10	P3 (2/4/15)	P4 (0/4/16) P6 (0/4/11) P1 (1/4/11)	P3이 P1보다 priority가 높아져 current가 되었고, P3의 priority가 원래 priority로 reset되었다. P1은 readyqueue에 들어갔고 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.
11	P4 (1/4/5)	P6 (0/4/12) P1 (1/4/12) P3 (2/4/16)	P4가 P3보다 priority가 높아져 current가 되었고, P4의 priority가 원래 priority로 reset되었다. P3은 readyqueue에 들어갔고 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.
12	P3 (3/4/15)	P6 (0/4/13) P1 (1/4/13) P4 (1/4/6)	P3가 P4보다 priority가 높아져 current가 되었고, P3의 priority가 원래 priority로 reset되었다. P4는 readyqueue에 들어갔고 마찬가지로 readyqueue의 process들의 priority가 1증가되었다.

< resources-adv2 for PIP >

# tick	current process	process in Ready queue(age/ lifespan/priority)	설명
0	P1 (1/4/0)		P1가 resource1,2,3,4를 요청->resource1,2,3,4 holding
1	P2 (0/3/5)	P1 (1/4/5)	P2가 생성되었고, P2의 priority가 더 높으므로 실행된다. P2가 resource1을 요청했지만, P1이 잡고 있으므로 P1 priority=5(P2의 priority)로 바뀌었다.
2	P4 (0/1/30)	P1 (1/4/30) P2 (0/3/5) P3 (0/4/10)	P3, P4가 생성되었고, P4가 priority가 가장 높아져 실행된다. P4가 resource1을 요청했지만, P1이 잡고 있으므로 P1 priority=30(P4의 priority)로 바뀌었다.
3	P1 (2/4/0)	P2 (0/3/5) P3 (0/4/10) P4 (0/1/30)	P1의 priority가 가장 높아져 실행하고, P1이 resource1을 반환했다. 따라서 P1의 priority가 0(P1의 prio_orig)으로 reset되었다.

4	P4 (1/1/30)	P2 (0/3/5) P3 (0/4/10) P1 (2/4/0)	P4의 priority가 가장 높아져 실행하고, P4가 resource1을 잡아 1tick을 실행하고 반환하였고, P4가 종료되었다.
5	P3 (1/4/10)	P2 (0/3/5) P1 (2/4/0)	P3의 priority가 가장 높아져 실행한다.
6	P3 (2/4/10)	P2 (0/3/5) P1 (2/4/0)	5 tick과 동일
7	P3 (2/4/10)	P2 (0/3/5) P1 (2/4/10)	P3이 resource2요청했지만, P1이 resource2를 잡고 있으므로, P1의 priority=10(P3의 priority)으로 바뀌었다.
8	P1 (3/4/0)	P2 (0/3/5) P3 (2/4/10)	P1의 priority가 가장 높아져 실행하고, P1이 resource2를 반환했다. 따라서 P1의 priority가 0(P1의 prio_orig)으로 reset되었다.
9	P3 (3/4/10)	P2 (0/3/5) P1 (3/4/0)	P3의 priority가 가장 높아져 실행된다. P3가 resource2를 잡아 1tick 실행하였다.
10	P3 (4/4/10)	P2 (0/3/5) P1 (3/4/0)	P3가 resource2를 잡은 상태로 또 1tick 실행하고 resource2를 반환하였고, P3가 종료되었다.
11	P2 (1/3/5)	P1 (3/4/0)	P2가 resource1을 잡았고, 1tick 실행하였다.
12	P2 (2/3/5)	P1 (3/4/0)	P2가 resource2를 잡았고, 1tick 실행한 후 resource1,2를 모두 반환하였다.

- Lesson learned

이번과제를 통해서 수업시간에 배운 scheduling 방법들을 구현해보았다. 이론으로만 배웠을 때는 scheduling 방법론들이 매우 간단하게 느껴졌지만, 실제로 구현해보니 고려해 주어야할 문제나 처리해주어야 할 예외가 꽤 많다는 것을 깨달았다. 예를 들면 프로세스가 생성될 때만 priority를 비교해주면 모든 priority scheduling에서 정상 작동할 것이라고 생각했지만, 실제로 aging, PIP, PCP 등을 고려해주기 위해서는, process의 priority가 갑자기 바뀌었을 때에도 scheduling을 통해 계속해서 current process를 실행할지, current를 바꾸어줘야 할지를 고려해 주어야 했다. 또한 운영체제 과목을 통해 자주 사용하던 연결리스트이지만, 디테일한 원리를 알지 못하는 것들이 많았다는 것을 깨달았다. 실제로 이번 과제를 구현하면서 하나의 연결리스트를 서로 다른 연결리스트에 동시에 넣을 수 없다는 것을 알게 되었는데, 이를 파악하는 데에 꽤나 오래 걸려 과제를 진행하는 데에 큰 어려움이 되기도 했었다. 이번과제를 통해 간단한 기능 구현뿐만이 아닌, 디테일한 예외에 대해 고민하고 고려할 수 있게 되었으며, 과제에서 구현한 스케줄러가 어떻게 작동하는지는 완벽하게 이해할 수 있었다.