OS project2 scheduler simulator report

학번 : 201520908

이름 : 유성민

학과 : 소프트웨어

학년 : 3

첫 번째는 sjf_scheduler입니다. 교수님께서 제시해주신 fifo_scheduler를 많이 참고했습니다. Sjf mechanism은 웨이트 큐가 비지 않았을 때, list_for_each_entry() 함수를 사용하여 레디 큐(링크드 리스트)를 순회하면서 lifespan이 제일 작은 프로세스를 뽑아서 다음 번 스케쥴 대상으로 선택하게 만들었습니다. 그리고 이 뽑은 프로세스는 레디 큐에서 빠져야 하므로 list_del_init() 함수를 사용하여 삭제시켰습니다. 다음 프로세스를 뽑지 않아도 되는 경우는 "current->age < current->lifespan" 이라는 조건을 사용하여 이 조건을 만족할 때 current를 리턴 시켜 다음 프로세스 뽑는 과정을 못하게 막았습니다. Resource에 대한 부분은 acquire은 fcfs_acquire() 함수를 사용하였고, release는 fcfs_release() 함수를 사용했습니다.

두 번째는 srtf_scheduler입니다. 이것 또한 fifo_scheduler와 비슷한 구조로 짜여 있습니다. Next를 뽑는 mechanism은 sjf_schedule의 mechanism을 따르되, preemptive하게 구현해야 하므로 "current->age < current->lifespan" 조건 에서 현재 프로세스를 list_move()함수를 사용하여 레디 큐의 첫번째 entry로 넣어주었습니다. 이유는 당연하게도 이미 lifespan이 최소인 프로세스가 뽑혔고 현재 돌고있는 프로세스의 remain time 보다 lifespan이 작은 프로세스가 나중에 도착하더라도 next에서 뽑힐 것이므로 "current->age < current->lifespan" 조건 에서 current를 리턴 하는 하는게 아닌 레디 큐에 다시 넣고 다시 next를 뽑는 메커니즘으로 넘어가서 제일 일찍 끝나는 프로세스를 뽑으면 해결됩니다. Resource에 대한 부분은 sjf와 마찬가지로 acquire은 fcfs_acquire() 함수를 사용하였고, release는 fcfs_release() 함수를 사용했습니다.

세 번째는 rr_scheduler입니다. 알아서 framework가 잘라주므로 time_slice는 구현할 필요가 없었고 단지 next를 뽑는 기준은 도착한 순서이므로 레디 큐에서 list_first_entry() 함수를 사용하여, 첫번째 엔트리를 뽑고, 이 프로세스를 레디 큐에서 list_del_init() 함수를

이용하여 지우고, 정해진 time_slice가 되면 알아서 다음 프로세스가 스케쥴 되므로 "current->age < current->lifespan" 조건에서 현재 스케쥴이 되고 있는 프로세스를 list_move_tail()함수를 사용하여 레디 큐의 꼬리에 다시 붙여 주기만 하면 알아서 스케쥴이 됩니다. Resource에 대한 부분은 acquire은 fcfs_acquire() 함수를 사용하였고, release는 fcfs_release() 함수를 사용했습니다.

네 번째로는 prio_scheduler입니다. next를 뽑는 mechanism은 prio값이 높을수록 먼저 뽑혀야 하는 것이므로 list_for_each_entry_safe() 함수를 사용하여 제일 prio값이 높은 프로세스를 스케쥴 하도록 했습니다. 물론 이것도 위의 스케쥴 함수들과 마찬가지로 뽑힌 프로세스는 레디 큐에서 빠져야 하므로 list_del_init() 함수를 사용하여, 레디 큐에서 지웠습니다. Priority scheduler를 preemptive하게 구현했고, "current->age < current->lifespan" 조건에서 srtf schedule 방식과 마찬가지로 현재 프로세스를 list_move_tail() 함수를 사용하여 다시 레디 큐에 넣어주고 다시 next를 뽑는 메커니즘으로 돌아가서 제일 높은 우선순위를 가지는 프로세스가 뽑히도록 구현했습니다. 위의 세가지 스케쥴 방식과 다르게 prio scheduler는 acquire는 직접 구현한 prio_acquire()함수를 사용하였고, 마찬가지로 release도 직접 구현한 prio_release()함수를 사용하였습니다.

다섯 번째로는 pip_scheduler입니다. Prio와 똑같고 단지, protocol이 가동될 수 있도록 global variable "working"을 선언하고 pip 스케쥴러가 선택되는 경우 값을 "true"로 바꿔서 pip가 동작되도록 하였습니다. pip구현은 prio_acquire() 함수와 prio_release() 함수에서 구현했습니다. 이외에는 prio_scheduler와 동일하므로 설명은 생략하겠습니다.

여기부터는 구현한 **prio_acquire()함수와 prio_release()함수**에 대해서 설명하겠습니다.

먼저 prio_acquire() 함수입니다. "!r->owner" 조건을 통해 현재 리소스를 잡고있는 프로세스가 없으면 이 리소스를 잡고자 하는 프로세스를 owner로 만듭니다. 그리고 fcfs_acquire() 함수와는 다르게 PIP구현을 위해 현재 리소스를 잡은 프로세스의 prio값을 프로세스 구조체에 있는 prio_orig 값에 넣어줬습니다. 리소스를 잡고있는 프로세스보다 높은 우선순위를 가진 프로세스가 리소스를 잡으려고 할 때, 이 프로세스로부터 prio값을 상속 받고 다시 본래의 prio값으로 돌아오기 위해 현재의 prio값을 유지할 필요가 있기 때문입니다. 그리고 리소스를 잡은 것을 알리기 위해 "true"를 리턴합니다.

Owner가 있는 경우는 현재 프로세스는 해당 리소스를 이용 못하므로 해당 리소스의 웨이트 큐에서 대기 상태로 존재해야 합니다. "current->status = PROCESS_WAIT" 코드를 이용하여 프로세스 상태를 "Wait"상태로 바꾸고 list_add_tail() 함수를 사용하여 리소스의 웨이트 큐의 꼬리에 붙입니다. 그리고 리소스를 잡지 못했으므로 false를 리턴합니다.

중간에 위치한 working 변수를 이용한 if문은 PIP구현부분입니다. 이부분은 현재 프로세스의 prio가 owner의 prio보다 높은 경우 즉, "current->prio > (r->owner)->prio" 이 조건을 만족하면, "(r->owner)->prio = current->prio" 코드를 이용하여 현재 프로세스의 prio 값을 owner의 prio에 상속 시킵니다. 그리고 prio값이 상속 됐으면 리소스를 놓을 때 원상복구를 해줘야 하므로 이 프로세스의 prio값이 상속된 것인지 아닌 지에 대한 판별이 필요한데, 이 부분은 global variable "change" 를 사용하여 구현 했습니다. 이 변수의 값이 "true"면 상속이 됐음을 의미합니다.

다음은 prio_release() 함수입니다. 이 함수는 위의 acquire() 함수에서 이용했던 "change" 변수의 값이 ture이면 prio값을 원상 복구 시키는 "(r->owner)->prio = (r->owner)->prio_orig" 코드를 구현했습니다. 그리고 r->owner = NULL 로 바꿔서 해당 리소스의 owner가 없는 상태로 바꾸고 웨이트 큐가 비지 않았을 때,

"!list_empty(&r->waitqueue)" 조건을 사용하여 list_for_each_entry_safe() 함수를 사용하여서 waitqueue를 순회하면서, 웨이트 큐에서 리소스를 놓은 프로세스를 waitqueue에서 지우고 이 프로세스의 상태를 "READY"상태로 변경하고 list_add() 함수를 사용하여, readyqueue의 첫번째 엔트리로 집어넣습니다. 이 과정에서 현재 prio_schedule 함수에서 다음 스케쥴 대상으로 뽑히고 running 준비가 된 프로세스 current와 충돌이 발생할 수 있기 때문에 list_del_init() 함수를 사용하여 current를 리스트에서 삭제 시킵니다. 그리고 다시 prio_schedule로 돌아가서 다음에 스케쥴 될 프로세스를 뽑습니다.