1. Spinlock

critical section에 진입이 불가능할 때 진입이 가능할 때까지 루프를 돌면서 재시도하는 방식이다. 따라서 lock을 획득할 때까지 해당 thread가 빙빙 돌고 있어야 한다.

그러기 위해서는 더 이상 쪼개질 수 없이 한 번에 쭉 진행되는 atomic operation으로 구현해야 한다. 이 때 사용한 방식은 compare_and_swap이다. 이것은 기대되는 값과 현재 가지고 있는 값이 동등한가를 비교하는 방식으로 동작한다.

while(compare_and_swap(&lock->held, 0, 1)

while(compare_and_swap(&lock->held, 0, 1)은 lock 이 두 번째 매개변수와 같을 때만 lock을 1로 설정한다. 하지만 return 값은 두 번째, 세 번째 매개변수에 상관없이 항상 lock의 상태를 반환한다. 즉, lock이 아무것도 안 걸려있으면 0을 반환하고 lock을 1로 바꿔주고, lock이 걸려있으면 1을 반환하고 lock은 그대로 1로 내버려둔다. 따라서 lock이 1이면 다른 thread가 수행되고 있다는 뜻이므로 계속 while문에서 빠져나오지 못하고 기다린다.

2. Blocking Counting Semaphore

mutex를 초기화할 때 waiter라는 list_head를 만든다. waiter는 lock을 받기를 기다리는 list이다. count와 lock을 각각 1과 0으로 초기화한다.

acquire_mutex

acquire_mutex는 다음 로직을 통해 구현하였다. Thread가 signal interrupt를 받을지 말지를 설정하기 위해 sigemptyset으로 signal set을 비워준 뒤 특정 signal만(wake up할 때 필요한 signal) sigaddset을 통해 signal set에 추가한다. 그리고 pthread_sigmask를 통해 critical section 진입 전에 SIG_BLOCK을 인자로 넣어 다른 thread들로부터 interrupt를 받지 않도록 한다. Critical section을 빠져나오면 SIG_UNBLOCK을 통해 interrupt를 받아도 되게 한다. 이는 race condition를 막기 위해 사용된다. 그 후 lock을 걸고 count가 1로 초기화되어 있으므로 0보다 크면 자신의 count를 감소시킨다. 이는 다른 thread가 들어오지 못하도록 한다. 그 다음 pthread_sigmask에서 SIG_UNBLIOCK을 통해 interrupt를 받아도 되는 상태로 변경한다. 다른 thread들과 concurrent하게 진행되므로 한 thread가 lock을 잡고 있다면 이후에 들어온 thread는 pthread_self()를 통해 thread를 calling하고 waiter라는 list에 추가된다. 그 후 sigwaitinfo를 통해 특정 signal을 기다리는 sleep상태에 빠진다.

release_mutex

release_mutex는 다음 로직을 통해 구현하였다. Thread가 signal interrupt를 받을지 말지 설정하고 lock을 거는 과정은 acquire_mutex와 동일하다. 그 후 만약 waiter가 비었다면 count를 증가시킴으로써 원위치하여 해제하고, 그렇지 않다면 waiter의 첫번째 인자를 빼서 지우고 pthread_kill을

통해 특정 signal을 보냄으로써 thread를 wake up한다.

3. Ring Buffer

empty와 full 두 개의 counting semaphores와 하나의 mutex를 써서 ring buffer를 구현하였다. semaphore 구조체에 count와 lock, 그리고 list_head waiter를 넣었다.

init_ringbuffer

empty의 count(비어있는 공간의 수)는 nr_slots, full의 count(차있는 공간)는 0으로 초기화하였다. empty와 full의 lock은 0으로 초기화하였고 mutex는 기존에 구현했던 초기화 함수를 사용하였다.

Enqueue_into_ringbuffer

버퍼에 남아있는 공간(empty)가 0이면 value를 넣을 수 없기 때문에 대기한다. acquire_mutex과 release_mutex를 재사용하는 이유는 race condition을 막기 위해서이다. 인자로 들어온 value는 0으로 초기화된 in을 인덱스로 하여 slots라는 배열에 들어가게 된다. 이때 원형큐의 방법으로 value가 들어간다. value를 넣었기 때문에 채워져 있는 공간의 수(full count)를 증가시킨다.

dequeue_from_ringbuffer

채워져 있는 버퍼의 공간(full)이 0이면 value를 뺄 수 없기 때문에 대기한다. acquire_mutex과 release_mutex를 재사용하는 이유는 race condition을 막기 위해서이다. value값을 return하기 위해 slots에서 값을 가져와 저장한다. 이 역시 원형큐의 방법으로 value를 빼온다. 그 다음 value를 뺏기 때문에 버퍼에 남아있는 공간의 수(empty count)를 증가시킨다.

wait

인자로 semaphore가 들어간다. mutex 방법대로 interrupt를 못 받게 한다. 그리고 lock을 걸어 count(empty or full의 count)를 감소시킨다. 만약 count가 0보다 작으면 대기를 해야 하므로 waiter에 넣어준다. 이 과정은 lock이 걸려있어 atomic하게 수행된다. 이 작업을 마치면 lock을 해제하고 종료한다.

signal_s

wait와 같은 방식으로 interrupt 수신의 유무를 설정한 후 count(empty or full의 count)를 증가시킨다. 0보다 작거나 같을 때 일어날 수 있는 상황을 예를 들어 설명하자면 다음과 같다. Enqueue에서 버퍼에 남아있는 공간이 0이 되어 대기 중일 때(waiter에 넣어진 상태) dequeue에서 signal_s를 호출하여 버퍼에 남아있는 공간을 증가시켜 대기상태를 풀어줘야 한다. 그렇기 때문에 count 값이 0이하면 waiter에서 대기중인 해당 thread를 불러 wake up해야한다. 이 작업 역시 atomic하게 수행 후 lock을 해제하고 종료한다.