**Operating Systems**

**Spring, 2020**

**School of Software, CAU**

**Project #1**

**- A Thread Systems and Synchronization –**

[Pintos 소스코드 분석 및 구현 보고서]

학번: 20150804

이름: 정 민 준

**1. Pintos 소스코드 분석 보고서**

- 본 장에서는 Pintos의 스레드 구현 방식 및 동기화 장치의 구현 방식을 분석한 결과를 요구사항에 맞추어 기술한다.

**1.1 Thread**

**1.1.1 Pintos에서 정의된 Thread status**

- 먼저 Pintos에 thread 구조체가 정의된 소스코드를 확인해 보겠습니다.

- enum thread\_status를 통해 쓰레드 상태를 확인할 수 있습니다. pintos에서 정의된 쓰레드의 상태는 총 4 가지 입니다.

1. THREAD\_RUNNING은 쓰레드가 작동 중인 상태를 의미합니다.

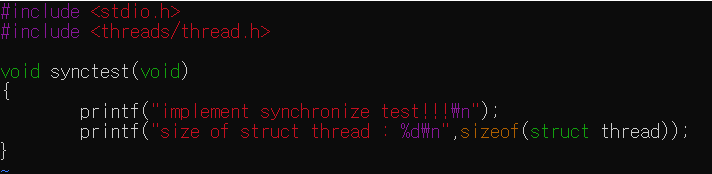
2. THREAD\_READY는 돌기 위해 ready\_list에 위치하는 쓰레드 상태를 말합니다.

3. THREAD\_BLOCKED은 쓰레드 상태를 블락으로 바꿔줍니다. 그리고 schedule 함수를 호출하면서 블락시킵니다.

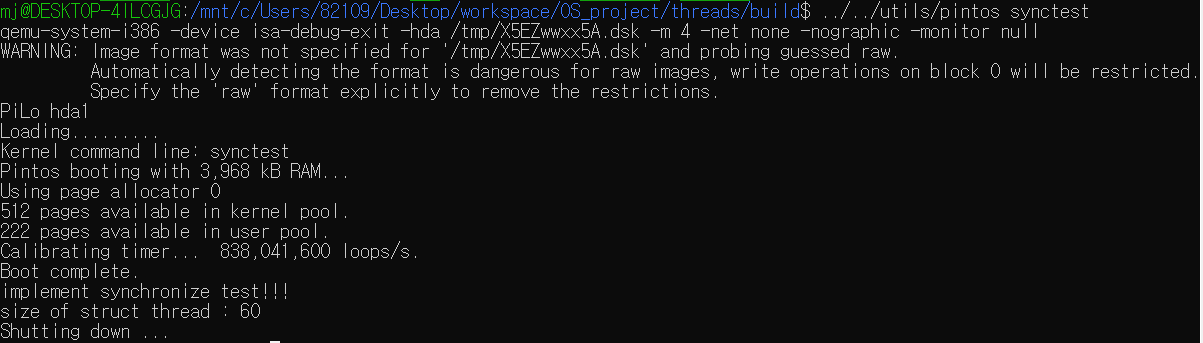
4. THREAD\_DYING는 thread\_schedule\_tail 함수에서 쓰레드를 free 해주는 상태입니다.

**1.1.2 ‘struct thread’의 크기**

- 앞서 쓰레드 구조체의 소스코드를 통해서 어떻게 쓰레드가 정의되었는지 알 수 있었습니다. (tid: 쓰레드 식별자, name: 쓰레드 명, priority: 쓰레드 우선순위 ….) 쓰레드 구조체의 크기를 확인하기 위해서 synctest.c 소스코드에 struct thread의 크기를 출력해주는 코드를 추가했습니다.



<sizeof(struct thread) 출력구문 추가>



- 실행 결과를 통해서 쓰레드 구조체 크기가 60Byte임을 알 수 있었습니다.

**1.1.3 Thread 함수 동작방식**

-먼저 thread\_init 함수를 살펴보겠습니다.

**- thread\_init :** 쓰래드를 사용하기 위한 환경을 조성해주는 함수로 먼저 쓰레드를 관리할 tid\_lock, ready\_list, all\_list, sleep\_list를 초기화 하고 현재 작동 중인 쓰레드를 running\_thread 함수로 불러와 initial\_thread 변수에 저장합니다. 그리고 initial\_thread의 상태를 THREAD\_RUNNING로 바꾸고 tid를 할당합니다.

\* ready\_list: 상태가 THREAD\_READY 인 쓰레드의 연결리스트.

\* all\_list: 존재하는 모든 쓰레드를 지닌 리스트.

\* sleep\_list: 상태가 THREAD\_BLOCKED인 쓰레드의 연결리스트.

**- thread\_create :** 새로운 쓰래드를 생성하여 run queue에 넣어주는 함수로 4 가지 구조체 포인터 변수와 tid를 확인할 수 있습니다. 쓰레드의 공간할당을 위해서 palloc\_get\_page 함수를 사용했습니다. 이 때 kernel\_pool에 page를 할당하게 되는데 kernel\_pool은 critical section이기 때문에 struct lock구조체를 이용한 lock이 활용되었습니다. struct lock은 최대값으로 1값을 갖는 semaphore로 구현되어 있습니다. 즉, 일반semaphore를 활용해도 되지만 struct lock을 사용하였습니다. 그 이유는 kernel pool 같은 경우cs에 접근할 수 있는 최대 개수가 1인 특별한 경우였기 때문에 범용적으로 사용될 수 있는 semaphore가 아닌 조금 더 많은 정보가 담긴 함수를 활용하여 이해하기 쉬운 코드를 작성하기 위해서 였다고 생각합니다.

그리고 init\_thread함수를 호출하여 쓰레드의 이름, 우선순위를 지정합니다. 이 때 쓰래드를 구분하기 위한tid를 할당하게 되는데 tid는 증감연산자를 통해 1씩증하게 됩니다. 이러한 tid는 쓰래드를 구분하기 위한 구분자이기 때문에 kernel\_pool과 마찬가지로 critical section이고 여러쓰래드가 동시에 접근하는 것을 막기 위해 struct lock구조체를 사용합니다. (즉, lock\_aquire()와 lock\_release를 활용하여 mutual exclusion구현)

그 다음 kernel\_thread(), switch\_entry(), switch\_threads() 함수를 위한 스텍 프레임을 할당합니다. 그리고 완성된 thread구조체를 thread\_unblock()함수를 활용하여 ready\_list(run\_queue)에 넣어줍니다.

**- thread\_unblock :** thread를 Run Queue에 추가해 주는 함수로 thread 포인터 변수를 함수 인자로 받습니다. 해당 쓰레드를 ready\_list 제일 뒤에 추가한 다음 상태를 THREAD\_READY로 바꿔줍니다.

**- thread\_block :** 현재 동작중인 쓰래드를 block상태로 만드는 함수로 thread\_unblock 함수와 다르게 함수인자를 받지 않고 현재 쓰레드의 상태를 THREAD\_BLOCKED로 바꿔준 다음 schedule 함수를 호출 합니다. 이어서 schedule 함수를 보겠습니다.

**- schedule :** 현재 쓰래드를 Run\_Queue 에 존재하는 쓰래드로 교체해 주는 함수로schedule 함수에서는 running\_thread 함수를 통해 현재 작동 중인 쓰레드 포인터 변수 cur, next\_thread\_to\_run 함수를 통해 다음에 작동할 쓰레드 포인터 변수 next, 이전 쓰레드 포인터 변수 prev를 선언합니다. 조건문을 통해 cur과 next가 같지 않다면 다음 순서인 쓰레드를 switch\_threads 함수를 통해 동작 시키고 prev에 cur을 저장합니다. 그리고 Thread 상태를 THREAD\_RUNNING로 업데이트 하기 위해 thread\_schedule\_tail 함수를 호출합니다.

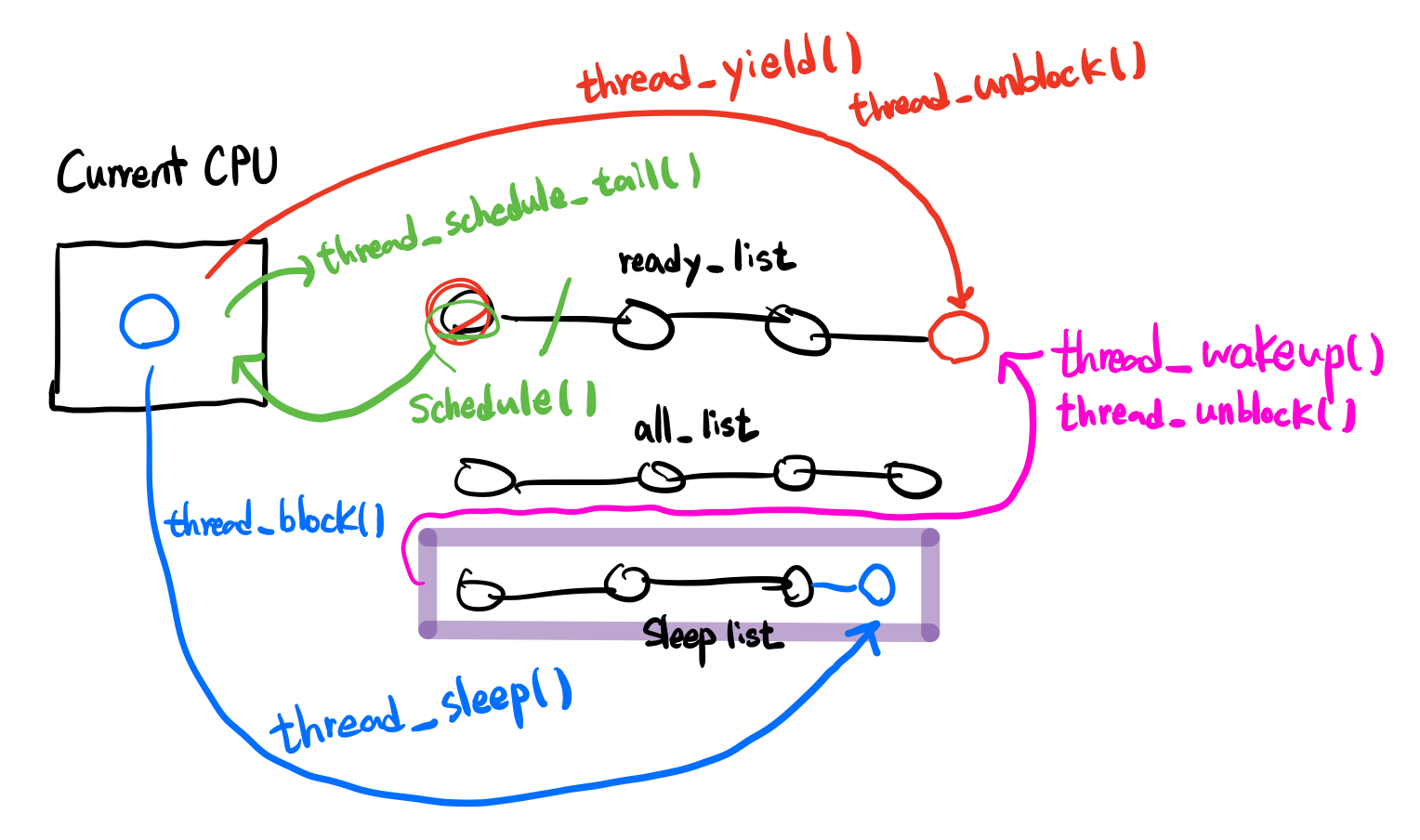
**- thread\_schedule\_tail :** running\_thread 함수를 통해 현재 작동 중인 쓰레드를 쓰레드 포인터 변수 cur에 저장합니다. 그리고 cur의 상태를 THREAD\_RUNNING으로 선언합니다.

**- thread\_sleep :** 현재 동작중인 쓰래드를 재우는 함수로 쓰레드 포인터 변수 cur 에 현재 작동 중인 쓰레드를 thread\_current 함수를 통해 저장합니다. 그리고 sleep\_list의 맨 뒤에 추가하고 thread\_block 함수를 통해 쓰레드 상태를 바꿔줍니다.

**- thread\_wakeup :** 앞서 thread\_sleep 함수에서는 작동 중이던 쓰레드의 상태를 THREAD\_BLOCKED로 바꾸고 sleep\_list에 푸쉬하였습니다. 반대로 thread\_wakeup 함수에서는 sleep\_list에 존재하는 쓰레드들을 thread\_unblock 함수를 사용하여 ready\_list로 이동하는 기능을 담당하고 있습니다. 소스코드 순서대로 설명하겠습니다. list\_lem 구조체 포인터 변수 e를 선언합니다. 그리고 list\_begin 함수 인자에 sleep\_list 포인터 변수를 선언하여 호출함으로써 e가 list의 시작 구조체 포인터에 접근합니다. sleep\_list의 첫 번째 쓰레드 구조체 에 접근하여 삭제와 해당 쓰레드를 unblock 시킴과 동시에 다음 쓰레드에 접근함으로써 결국 sleep\_list의 모든 쓰레드들을 sleep\_list에서 삭제와 ready\_list로 이동시키는 기능을 수행함을 알 수 있습니다.

**- thread\_yield :** 현재 쓰래드를 Run\_Queue로 이동시키고 새로운 쓰래드를 실행시키는 함수로 쓰레드 구조체 포인터 변수 cur에 현재 작동 중인 쓰레드를 선언합니다. cur을 ready\_list로 푸쉬하고 상태를 THREAD\_READY로 바꿔준 다음 schedule 함수를 호출합니다.

- Thread 함수들을 정리해보겠습니다.



- Thread\_create()를 통해 쓰레드가 생성되고 해당 기능을 수행하면 Thread\_exit()을 통해 상태를 THREAD\_DYING으로 설정합니다.

- Schedule()가 호출되면 ready\_list의 쓰레드가 현재 작동 중인 쓰레드 갱신됩니다. thread\_schedule\_tail()는 해당 쓰레드의 상태를 THREAD\_RUNNING으로 설정합니다.

- thread\_sleep()는 현재 작동 중인 쓰레드의 상태를 THREAD\_BLOCKED로 바꿔준 뒤 sleep\_list에 푸쉬합니다. thread\_wakeup()은 sleep\_list의 첫 번째 쓰레드에 시작하여 순차적으로 마지막 쓰레드까지 접근하여 쓰레드의 상태를 THREAD\_READY로 바꿔준 뒤 ready\_list에 푸쉬합니다.

- thread\_yield()는 현재 작동 중인 쓰레드의 상태를 THREAD\_READY로 바꾸고 ready\_list에 푸쉬한 후 schedule()을 호출하여 ready\_list 맨 앞에 위치한 쓰레드를 작동시킵니다.

**1.2 동기화 장치**

**1.2.1 동기화 장치 함수들의 동작 방식**

**1) Semaphore**

- 먼저 sync.h에 세마포어 구조체를 살펴보겠습니다. 세마포어는 unsigned value 변수와 리스트 구조체 waiters 변수를 가지고 있습니다.

**- sema\_init :** 세마포어가 가질 value를 선언하고 list를 초기화 합니다. 여기서 value는 공유 자원에 접근할 수 있는 쓰레드의 수, waiters는 공유 자원 접근을 대기중인 쓰레드 리스트입니다.  
  
**- sema\_down :** 세마포어 구조체 포인터 변수 sema를 함수 인자로 받습니다. while문을 통해 만약 sema의 value가 0이라면 쓰레드를 허용하지 않겠다는 의미로 sema의 waiters 리스트에 현재 쓰레드의 elem을 푸쉬합니다. 이어서 thread\_block을 호출하여 쓰레드의 상태를 THREAD\_BLOCKED로 갱신합니다. 만약 쓰레드를 허용할 수 있는 상황이라면 sema의 value를 1 감소시킵니다. 이는 sema\_up 함수를 통해 기다리고 있던 쓰레드들이 순차적으로 실행될 것입니다.

**- sema\_up :** sema\_down와 같이 세마포어 구조체 포인터 변수 sema를 함수 인자로 받습니다. sema의 waiters 리스트가 비어있지 않다면 리스트 맨 앞 쓰레드를 pop하고 list\_entry호출로 쓰레드 포인터를 반환합니다. 그리고 thread\_block 호출을 통해 반환된 쓰레드의 상태를 ready\_list로 푸쉬하고 THREAD\_READY 상태로 변경합니다. 다음 sema의 value를 1 감소합니다.

**- sema\_try\_down :** bool 변수 success를 선언합니다. 그리고 sema의 value를 조건문을 통해 현재 쓰레드를 작동 가능한 상태이면 (=value > 0) value를 1만큼 감소하고 success를 true로 선언합니다. 반대의 경우는 success에 false를 선언합니다. 그리고 success를 반환합니다. 이를 통해 현재 쓰레드를 실행시킬 수 있는지에 대한 여부를 판단할 수 있게끔 합니다.

**2) Lock**

- sync.h에 lock 구조체 소스코드를 살펴보겠습니다. 쓰레드 구조체 포인터 변수 holder와 세마포어 구조체 변수 semaphore을 가지고 있습니다.

**- lock\_init :** lock의 holder를 NULL로 선언하고 sema\_init 를 호출하는데 value가 1인 세마포어가 생성됩니다.

**- lock\_acquire :** lock 구조체 포인터 변수 lock을 함수 인자로 받는다. 그리고 lock의 semaphore를 sema\_down에 호출합니다. 그리고 lock을 걸고있는 쓰레드는 현재 작동 중인 쓰레드임으로 lock의 holder 변수에 thread\_current를 호출하여 쓰레드 포인터를 저장합니다. 세마포어와 비교해 보자면 제약 조건이 추가한 것으로 볼 수 있습니다. 다른 쓰레드가 아닌 현재 쓰레드만이 락을 가지고, release 가 필요할 때까지 대기합니다. 앞서 말했듯이 이는 sema\_down 함수를 호출하여 기다리게 되고 lock의 holder 변수에 현재 쓰레드를 저장함으로써 가능합니다.

**- lock\_try\_acquire :** lock 구조체 포인터 변수 lock을 함수 인자로 받는다. 그리고 bool 변수 success를 선언한다. lock의 세마포어를 sema\_try\_down에 넘겨줌으로 결과를 success에 변수에 저장합니다. 만약 success가 참이라면 lock의 holder에 작동 중인 쓰레드 포인터를 저장하고 success를 반환하고 아니라면 false값이 저장된 success가 반환될 것입니다.

**- lock\_release :** lock 구조체 포인터 변수 lock을 함수 인자로 받는다. lock의 holder를 NULL로 선언함으로써 현재 lock이 걸리지 않은 상태로 전환합니다. 이어서 lock의 semaphore을 sema\_up 호출로 semaphore의 value를 1 증가시킵니다.

**3) Condition**

- condition 구조체는 기다리고 있는 쓰레드를 가진 리스트 waiters를 가지고 있습니다.

**- cond\_init :** condition 구조체 포인터 변수 cond를 함수 인자로 받아 cond의 waiters 리스트를 초기화 합니다.

**- cond\_wait :** 해당 함수가 호출되기 전에 lock이 반드시 걸린 상태여야 합니다. Shared data를 확인하여 원하는 조건이 아닌 경우 쓰레드가 실행될 수 없습니다. sema\_init에 앞서 선언한 semaphore\_elem 구조체 변수 waiter 포인터와 0을 넣어 호출하여 리스트를 초기화 하고 value를 0으로 지정합니다. 그리고 cond의 waiters 리스트를 waiter.elem에 푸쉬합니다. 그러면 함수인자로 받은 cond 구조체의 waiters 리스트가 sema\_init을 통해 만든 waiter 리스트에 저장됩니다. 현재 lock이 걸려있는 상태이기 때문에 lock\_release 호출로 lock을 해제하고 sema\_down를 호출합니다. value가 0이기 때문에 어떤 쓰레드도 실행되지 못하고 모두 대기하게 됩니다. 그리고 다시 lock\_acquire를 호출하여 lock을 걸게 됩니다.

**- cond\_broadcast :** 특정 조건이 만족하여 기다리고 있던 쓰레드 리스트를 깨워주기 위해 함수 인자로 받은 cond의 waiters 리스트가 비어있을 때까지 cond\_signal를 호출합니다. 정리하자면 condition variable에서 기다리는 모든 쓰레드에게 signal을 호출해 주는 기능을 수행합니다.

**- cond\_signal :** cond\_broadcast를 통해 호출되어 cond의 waiters 리스트 맨 앞 쓰레드부터 순차적으로 제거하고 sema\_up에 list\_entry 함수를 통해 해당 쓰레드 포인터를 전달합니다. 결국 cond\_broadcast가 호출되면 cond\_signal를 통해 기다리고 있던 모든 쓰레드들의 상태가 THREAD\_READY로 저장되고 ready\_list로 푸쉬하게 됩니다.

**4) Extra**

- sema\_up, sema\_try\_down, sema\_up 세 함수는 인터럽트의 방해를 받지 않고 온전히 제 기능을 수행할 수 있게 보장받습니다. 함수 시작과 동시에 old\_level 변수를 선언하는데 intr\_disable 함수를 통해 인터럽트를 받아들이지 않도록 합니다. 마지막 부분에 intr\_set\_level(old\_level) 함수를 호출하여 인터럽트를 받아들이도록 합니다. 인터럽트 관련 함수에 대해서 설명하겠습니다.

- 먼저 intr\_level은 INTR\_OFF, INTR\_ON 두 가지를 가지는데 인터럽트를 받을지에 대한 여부를 결정합니다.

- intr\_disable 함수는 인터럽트를 받지 않게끔 하고 이전 인터럽트 상태를 반환합니다.

- 위에서 반환된 이전 인터럽트 상태를 intr\_set\_level 함수 인자로 넘깁니다. Intr\_set\_level 함수에서는 해당 level == INTR\_ON, 즉 가능한지 확인합니다. 참이라면 intr\_enable를 호출하고 아니라면 intr\_disable을 호출합니다. 이전 상태는 인터럽트가 가능했기 때문에 intr\_enable를 호출하여 인터럽트를 받을 수 있는 상태가 됩니다.

**2. Pintos thread 동기화 기능 구현**

- 본장에서는 Pintos의 쓰레드 동기화 기능 중 하나인 thread\_join() 함수를 구현하고 그 구현 방식과 이 함수가 올바르게 동작함을 보이기 위한 테스트 방법 및 결과를 기술한다.

**2.1 Thread\_join() 함수 구현**

**1. struct thread**

- 먼저 부모 쓰레드가 자식 쓰레드를 식별할 수 있도록 하기위해 쓰레드 구조체에 children\_list, children\_elem 두 list, list\_elem 구조체 변수를 추가했습니다. 또한 join과 자식 쓰레드의 메모리 해제를 위해서 destory\_sema, join\_sema 두 세마포어 구조체 변수를 추가했습니다. 세마포어 변수를 가져오기 위해 sync.h를 include 했습니다.

**2. init\_thread**

- 쓰레드 구조체에 변수가 추가되었음으로 먼저 destroy\_sema 와 join\_sema를 sema\_init를 통해 0으로 초기화 시켰습니다. 그리고 children\_list를 초기화 하고 쓰레드의 children\_list에 children\_elem을 푸쉬 했습니다.

**3. thread\_join**

- thread\_join 함수는 먼저 함수인자로 할당받은 tid가 자신의 자식 쓰래드인지 확인하고, 자식 쓰래드가 아니라면 대기하지 않고 바로 return 됩니다.

- thread\_join 함수에서 대기를 구현하기 위한 semaphore로 자식 쓰래드의 destroy\_sema와 join\_sema를 사용했습니다. 이 때 두개의 semaphore를 사용한 이유는 자식 쓰래드가 실행을 끝냈을 때 부모 쓰래드가 자식의 semaphore를 접근하게 되면 이미 할당 해제된 변수를 접근한 것이기 때문에 에러가 발생하게 됩니다. 그렇기 때문에 모든 자식 쓰래드는 부모 쓰래드가 종료되기 전까지 대기를 해야 하고 이를 구현하기 위해 semaphore를 종료를 대기하기위한 destroy\_sema, thread\_join의 대기를 구현하기위한 join\_sema. 즉 두개의 semaphore를 사용하였습니다.

- thread\_join이 호출되면 자식쓰래드의 join\_sema를 내림으로서 대기를 수행하게 되고 자식쓰래드가 종료되면 대기가 종료되게 됩니다. 그렇기 때문에 대기했던 쓰래드를 children\_list에서 제거해 주고 자식쓰래드의 distory\_sema를 올림으로서 자식쓰래드의 종료 대기를 풀어줍니다.

**4. thread\_exit**

- 현재 가지고 있는 모든 자식 children\_list를 순회하며 쓰래드들은 대기상태 이므로 destroy\_sema를 올림으로서 대기를 풀어줍니다.

- 만약 부모 쓰래드가 thread\_join을 호출하여 대기상태일 수 있으므로 thread\_join를 올림을서 부모쓰래드가 join을 호출해도 대기하지 않도록 합니다.

- 그리고 이 쓰래드는 부모 쓰래드의 자식이기 때문에 부모쓰래드가 죽기 전까지 대기하기 위해 destroy\_sema를 내려줍니다.

**2.2 Thread\_join() 함수 테스트 및 결과**

- synctest.c에 먼저 void test\_helper 함수를 추가했습니다.