Back to back

생각한 구현 방식

1. argv 값을 이용하여 ucontext_t 구조체의 주소값을 넘겨서, 이를 통해서 리턴받도록 한다.
2. 스택에 ucontext_t 구조체의 주소값을 참조시켜서 나중이 이를 통해서 리턴받도록 한다.
3. symbol 테이블을 조사하여, exit 함수의 호출을 조사한후, 여기에 적절한 명령어 집합을 넣어서 구현한다.
4. fork 한후 child process는 load elf를, parent는 wait 하게 하여 구현한다.

> 이렇게 구현한다면 exit 과는 관련이 없음으로 문제의 의도와는 일치하지 않는 것 같다.

5. exit 함수가 호출되는 메모리 영역에 다시 돌아오도록 하는 어셈블리 코드를 써 놓는다.
6. exit 함수에서 시그널을 호출하여, 시그널 handler가 다시 원래대로 코드를 실행시키게 한다. => 쉬운 방법임

> 6번 방법을 응요하면, yield또한 쉽게 구현할 수 있다. 왜냐하면 시그널을 그냥 호출시키면 되기 때문이다. 근데 이러면 시그널을 호출해야 되서 결국 시스템콜을 이용한 커널 모드로 접근하게 된다. 이렇게 된다면 user thread를 사용하는 이유가 사라지지 않을까? 즉 시그널을 이용한 방식은 쉽긴 하겠지만, user thread의 효율성 측면에서는 좋지 않은 방법이 될 수 있을 것 같다.

7. context에 대한 refernce를 설정한후, 그 reference의 값을 현재 loader에 있는 context의 reference 로 load time 에서 설정한다.

구현

1. 우선 Back to Back Loader와 같은 경우는 시그널로 구현한다. 그 다음 시간이 남으면 한번 exit을 직접 수정한는 것을 시도해 본다
2. user level thread는 setjmp보다는 setcontext가 스택 측면에서 간단하게 구현할 수 있다.

> 문서를 읽고서는, 이 user level thread 구현에서 ELF 프로그램들을 순차적으로 실행해야 하는지, 아니면 한 프로그램의 여러 함수들의 스레딩을 구현해야 하는지 애매한 부분이 있다.

user level thread

queue.c

스레드 스케쥴링에 필요한 queue 자료 구조를 정의한다. Github에서 잘 카피하도록 하자.

mthread.c

스레드 스케쥴링에서 필수적으로 필요한 함수인

int mthread_create(mthread_t *thred, void (*strt_function)(void *), void *arg)

1. 첫번째 argument인 thread 는 스레드가 성공적으로 생성되었을때 생성된 스레드를 식별하기 위해서 사용되는 스레드 식별자이다.
2. 2번째 argument인 start_routine는 분기시켜서 실행할 스레드 함수이며,
3. 3번째 argument인 arg는 스레드 함수의 인자이다.
4. 성공적으로 생성될경우 0을 리턴한다.
5. thread_tcb 구조체 생성과 메모리 할당

6. 스레드가 사용할 스택 영역 할당
7. 스레드 context 지정
8. 큐에 enqueue

mthread_t mthread_self(void)

1. 현재 이 함수를 호출한 함수에서 자기자신의 스레드 아이디를 얻고 싶을때, 스레드를 ti를 리턴한다.

void pthread_exit(void *retval);

1. 스레드를 종료시키고 자신을 생성한 함수에 retval을 리턴시킨다.
2. 현재 thread상태를 terminated로 변경후 thread_yield 호출

int pthread_yield()

1. 스케쥴러를 명시적으로 호출한다.

int pthread_join(mthread_t th, void **thread_return);

사실 thread join은 현재 1 thread에서 스케쥴링이 일어나는 현 상황에서는 필요가 없다. 여기서는 리턴값의 회수라는 측면에 초점을 맞추어서 한번 실행해 보도록 하자. 또한 메인 스레드를 스레딩의 영역에 포함시킬 것인지는 구현후 생각해보도록 하자.

1. 목표 스레드가 완수될때까지 현재 스레드를 호출한 스레드의 실행을 정지시킨다.
2. 0을 리턴하게 되며, thread_exit에서 인자로 넘겨진 retval이 명시된 thread_return을 통해서 리턴되어진다.

자료구조

1. typedef enum { RUNNING, READY, WAITING, SLEEP, TERMINATED } mthread_state_t;
2. struct mthread_tcb { ucontext_t *context; mthread_state_t state; mthread_t tid; void* return_value; }
3. typedef uint8_t mthread_t; 스레드의 id 저장

mschedule.c

여기서는 스케쥴링 함수를 정의 한다. 제일 단순한 FIFO함수를 만들도록 하자. 또한 yield 함수가 여러번 호출될 일은 없어서 (1 thread에서 작동함으로) preemption 함수는 작성할 필요가 없다. 만약 이 user thread가 여러 논리 코어를 사용할 수 있게 된다면 구현하도록 한다. 또한 time schedular에 의해서 작동할 경우에도 preemption을 구현하여야 하지만, 또한 구현의 범위를 넘어섬으로 생략하도록 한다.

1. 현재 thread를 READY로 변경 현재 thread가 TERMINATED면 메모리 해제후 큐에서 제거.
2. 현재 스레드가 READY면 큐 맨 뒤로 넣기
3. 큐에서 다음 스레드의 상태를 RUNNING으로 변경
4. 다음 스레드를 현재 스레드로 지정
5. context 스위칭 실행

void fifo_schduler(int signum)

이 함수는 timer에서 발생한 시그널을 받아서 스케쥴링를 하거나 (구현할 필요 없음) 혹은 yield가 실행되어서 다음 스레드를 실행시키게 된다. 주의 할 점은 스택 bp, sp를 복구 해야한다는 점, 그리고 context를 설정해주어야 하는점등이 있을 것이다.

mthread_tcb* mthread_current(void)

1. 큐에서 제일 위에 있는 스레드의 구조체를 이용하여 스레드 구조체의 포인터를 넘겨준다.

mthread_tcb* mthread_next(void)

1. 다음에 작동시킬 스레드의 구조체를 리턴한다.

자료구조

1. Queue readyQueue
2. Queue exitedQueue