Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – hoseong Lee(이호성)

hslee00001@naver.com

2일차

포크와 쓰레드 수행결과의 차이

clone→ 옵션에 따라 스레드도 될 수 있고, 프로세스가 될 수 있다.

clone의 경우는 용도나 특성에 있어서 위의 두 함수와 살짝 차이가 존재한다. 이미 process descriptor 인 task_struct구조체를 살펴보면서, 리눅스 커널에서는 스레드를 위한 구조체가 따로 존재하지 않고 프로세스와 스레드의 구분을 위해 약간 다른 처리를 한다고 확인했다. clone은 프로세스를 생성하긴 하나, 용도 상 프로세스 생성보다는 스레드 생성의 의미에 가깝다. 리눅스에는 실제 스레드라는 것이 없고, clone함수를 이용하여 프로세스를 생성하면, 자식 프로세스가 부모 프로세스의 메모리 영역을 공유 한다. 즉 fork는 별도의 공간을 할당한 뒤 부모 프로세스 영역의 값을 복사하는 형식인 것에 반해 clone은 메모리 공간 자체를 공유한다. 물론 코드 영역도 공유한다. 이것이 가지는 의미는 자식 프로세스에서 어떤 메모리 영역의 값을 변경시키면 부모 프로세스에서도 그 영역을 공유하므로 부모 프로세스 직접적으로 영향을 준다는 사실이며, 이것이 곧 스레드의 특성과 매우 흡사하다. 대개, 리눅스에서는 fork로 프로세스를 생성하고 clone 으로 스레드를 생성한다고 받아드려도 되겠다.

1) 쓰레드 상에서 수행결과의 차이 -54p

쓰레드는 같은 주소공간을 공유한다.

→ IPC의 개념들을 사용했다

→ 프로세스의 경우 메모리를 공유하지않아서 다른 방법을 사용했다 → 예) 쉐어드 메모리

쓰레드의 경우에는 크리티컬 섹션이 문제가된다. (바로 공유되고있는 공간)

스핀락, 세마포어, 뮤텍스 를 사용하여 이 크리티컬 섹션을 보호했다.

56페이지 예제

vfork

포크 자체에서는 디멘딩페이지를 했다. 이걸 하기 싫다면 infork를 사용해라

*햇갈리지마라.

task_struct - 프로세스, 쓰레드

mm_struct- 메모리

리눅스는 이를 관리하기위해 task_struct구조를 가진다.

→ 프로세스 구조를 전부다 알 수가 있게 된다.

58page 첫번째문단 마지막줄

task_struct 구조를 관리하게 된다.

리눅스 커널은 프로세스든 쓰레드든 모두 task_struct로 관리한다.

59페이지 첫번째줄

결론은 태스크

그림봐바

→ 커널에서 구동돼는 모습을 말한다.

→ 코드분석을 해보자 59페이지 그림

user app - fork, vfork, clone (우리가 만든것) → c liberary 가 돌아간다.(커널개발자가 만든) → 인터럽트 128번이 호출돼고 제어권이 커널로 된다. Sys_ 붙은 애들을 모두 처리하게 된다. 어디에 연결되있을까? v_fork에 연결되어 있다. 이들은 모두 커널 쓰레드를 생성한다.

= 결론적으로 fork, clone이든 뭐든 커널쓰레드이고 두포크가 관리를 한다.

Sched_fork

SYSCALL_METADATA(##sname,0);

asmlinkage long sys_##sname(void): 시스템콜은 어셈블리어로 동작하기때문에 어셈블리어는 c 코드를 불러올 수 없기 댸문에 c를 호출하려고 넣어준다. - 180409 1분40초

vi -t SYSCALL_DEFINE0

16 번

grep -rn "fork" | grep "SYSCALL"

숙제 : 1. sys fork 분석해 보기 do fork에서 뭐하는지 분석 !

2. NPTL (개념)

v2.4 커널에서 리눅스의 쓰레드는 우리가 pthread라고 칭하는 user 레벨에서 수행되던 light weight process의 개념이었다. 리눅스가 처음 개발되었을 때 커널에서 진짜 스레드를 지원하지 않았다. 하지만 리눅스는 clone()시스템 호출을 통해 스케쥴 가능한 엔티티로 프로세스를 지원했다. 이 호출은 호출하는 프로세스의 복사본을 생성해 호출자의 주소 영역에서 복사 공유가 가능하게 만들었다. LinuxThreads 프로젝트는 이런 시스템 호출을 사용해 사용자 영역에서 스레드 지원을 완벽하게 흉내냈지만, 이런 접근 방식에는 특히 신호처리, 스케줄링, 프로세스간 동기화 부문에서 몇가지 단점이 있었고, posix요구사항을 준수하지 못했다. 이러한 pthread의 성능 개선(커널 지원과 새로 작성한 스레드 라이브러리)을 위해 나타난 것이 Red Hat 의 NPTL 프로젝트이다. (v2.6 커널의 glibc에 포함) 이는 커널 레벨의 thread 스케줄링 환경 제공을 의미한다. 스레드는 프로그램을 쪼개어 동시에 동작하는 태스크를 하나 이상으로 나눈다. 단일 프로세스의 상태 정보를 공유하며, 다른 자원과 메모리를 직접 공유한다는 점에서 스레드는 전통적인 멀티태스킹에서 말하는 프로세스와 다르다. 동일 프로세스에 존재하는 스레드 사이에서 일어나는 문맥 전환은 일반적으로 프로세스 사이에서 일어나는 문맥 전환보다 훨씬 더 빠르다. 따라서 멀티스레드 프로그램은 멀티프로세스 응용보다 더 빠르다는 장점이 있다. 또한 스레드를 사용하면 동시에 처리하도록 구현할 수 있다. 이런 상대적인 장점이 프로세스 기반 접근 방식을 벗어나 LinuxThreads 구현을 이끌었다.

→ 리눅스 스레드 와 NPTL의 차이점 비교 (리눅스 스레드에서 발전한것이므로)

리눅스 스레드의 가장 핵심적인 특징 중하나는 관리자 스레드이다. 관리자 스레드는 다음 요구를 맞춘다. 시스템은 중요한 시그널에 반응하고 전체 프로세스에 kill로 시그널을 보낼 수 있어야 한다. 스택으로 사용된 메모리 할당 해제는 스레드 종료에 앞서 일어나야 한다. 따라서 스레드는 이런 작업을 스스로 할 수 없다. 반드시 스레드 종료를 기다려 좀비로 변하지 않도록 해야 한다. 스레드 지역 자료 할당 해제는 모든 스레드에 반복적으로 수행할 필요가 있다. 관리자 스레드가 이런 작업을 수행해야 한다. 메인 스레드가 pthread_exit()를 호출할 필요가 있을 때, 프로세스는 종료되지 않는다. 메인 스레드는 잠들기로 빠지며, 모든 다른 스레드를 종료하고 나서 관리자 스레드가 메인 스레드를 깨운다. 스레드 지역 자료와 메모리 유지를 위해, LinuxThreads는 스택 주소의 바로 아래 있는 프로세스 주소 공간의 상위 메모리를 사용한다. 동기화 요소는 시그널을 사용해서 달성한다. 예를 들어, 스레드 차단은 시그널로 깨어난다. clone 시스템의 초기 설계하에서, LinuxThreads는 각 스레드를 독자적인 프로세스 ID를 부여한 다른 프로세스로 구현했다. 치명적인 시그널은 모든 스레드를 죽일 수 있다. 이런 점에서 LinuxThreads 설계에는 일관성이 있어왔다. 일단 프로세스가 치명적인 시그널을 받으면, 스레드 관리자는 다른 모든 스레드(프로세스)에 kill로 똑같은 시그널을 보낸다. LinuxThreads 설계에 따르면, 비동기식 시그널을 보내면 관리자 스

레드는 이 시그널을 스레드 중 하나로 배달할 것이다. 대상 스레드가 현재 시그널을 차단하고 있다면 시그널은 대기 상태를 유지한다. 이렇게 하는 이유는 관리자 스레드가 프로세스에 시그널을 보낼 수 없기 때문이다. 대신 각 스레드는 프로세스처럼 행동한다. 스레드 사이에 일어나는 스케줄링은 커널 스케줄러가 처리한다.

→ 리눅스 스레드는 과도한 응용으로 부하를 주면 성능, 확장성, 사용성에 문제를 일으킨다. 블로그 참고(http://blog.naver.com/PostView.nhn?blogId=daleena&logNo=100104353225)

LinuxThreads 초기 설계는 관련된 프로세스 사이에서 일어나는 문맥 전환이 충분히 빨라서 개별 커널 스레드가 대응하는 사용자 수준 스레드를 처리할 수 있다는 믿음으로 출발했다. 이는 1대1 스레드 모델이라는 혁신을 이끌었다.

NPTL(Native POSIX Thread Library)은 LinuxThreads의 단점을 극복하기 위한 새로운 구현으로 POSIX 요구사항 또한 충족한다. NPTL은 성능과 확장성 측면에서 LinuxThreads보다 강력한 개선 사항을 제공한다. LinuxThreads와 같이 NPTL은 1대1 모델을 구현한다.

NPTL의 장점

NPTL은 관리자 스레드를 사용하지 않는다.프로세스의 일부로 모든 스레드에 치명적인 시그널을 보내는 등 관리자 스레드에서 필요한 몇 가지 요구 사항이 존재하지 않는다. 커널 자체가 이런 작업을 신경쓸 수 있기 때문이다. 커널은 또한 각 스레드 스택이 사용한 메모리를 할당 해제한다. 심지어 어버이 스레드를 정리하기 앞서 기다리고 있는 모든 스레드 종료를 관리하므로 좀비를 막을 수 있다.

관리자 스레드를 사용하지 않기 때문에, NPTL 스레드 모델은 NUMA와 SMP 시스템에서 좀 더 나은 확장성과 동기화 메커니즘을 제공한다.

새로운 커널 구현과 함께 NPTL 스레드 라이브러리는 시그널을 사용한 스레드 동기화 기법을 피한다. 이런 목적으로 NPTL은 퓨텍스(futex)라는 새로운 메커니즘을 도입했다. 퓨텍스는 공유 메모리 영역에서 동작하므로 프로세스 사이에 공유가 가능하므로 프로세스 간 POSIX 동기화를 제공한다. 또한 프로세스 사이에서 퓨텍스를 공유할 수도 있다. 이런 행동 양식은 프로세스 간 동기화를 현실로 만든다. 실제로 NPTL은 PTHREAD_PROCESS_SHARED라는 매크로를 포함해 개발자에게 다른 프로세스의 스레드 사이에 뮤텍스를 사용자 수준 프로세스에서 공유하기 위한 핸들을 제공한다.

NPTL이 POSIX 규약을 따르므로 NPTL은 프로세스 단위로 시그널을 처리한다. getpid()는 모든 스레드에서 똑같은 프로세스 ID를 반환한다. 예를 들어 시그널 SIGSTOP을 보내면 전체 프로세스가 멈춘다. LinuxThreads에서는 이 시그널을 받은 스레드만 멈춘다. 이는 NPTL 기반 응용에서 GDB와 같은 디버그 지원을 강화한다.

NPTL에서 모든 스레드에는 어버이 프로세스 하나만 존재하므로, 어버이에게 보고되는 자원 사용(CPU나 메모리 퍼센트와 같은)은 스레드 하나가 아니라 전체 프로세스에 보고된다.

NPTL 스레드 라이브러리에 도입된 중요한 특징 중 하나는 ABI(Application Binary Interface) 지원이다. 이는 LinuxThreads와 하위 호환이 가능하도록 돕는다. 다음에 다룰 LD_ASSUME_KERNEL의 도움을 받아 ABI 지원을 처리한다.

NPTL의 단점

최근 레드햇 커널에서 간단한 스레드 응용 프로그램이 단일 프로세스 기계에서는 정상으로 동작했지만 SMP에서는 얼어버린다. 하이엔드 응용을 만족시키기 위해 확장성을 높이려면 리눅스에서 해야 할 일이 여전히 많다.

current란 ?

(전류)

들어가기 vi -t get_current

grep -rn "getpid" ./ | grep SYSCALL’

→ vi kernel/sys.c

→ task_tgid_vnr (current)

task → tgid

pid 랑 tgid 같다면

vi kernel/sys.c

grep -rn "thread_union"

vi -t thread_union

```c
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[THREAD_SIZE/sizeof(long)];
};

#ifndef __HAVE_ARCH_KSTACK_END
static inline int kstack_end(void *addr)
{
        /* Reliable end of stack detection:
         * Some APM bios versions misalign the stack
         */
        return !(((unsigned long)addr+sizeof(void*)-1) & (THREAD_SIZE-sizeof(void*)));
}
#endif
```

녹음 190409_3 06분 00초

system context 란?

System context 를 관리해야하는 이유

Memory context 를 관리해야하는 이유

자기가 구동시키고있는 프로세스가 날아가기 때문에.

Network

하드웨어 context

→ 전부 context swiching 할때 필요함

전부 thread_union 에 있다. Task_struct 이기 때문이당

67페이지 두번째문단까지..

pid값은 프로세스의 고유 민증과 같은것

10분 10초

쓰레드 들의 리드격인 것이 pgid pid 가 같다

pid 와 pgid 가 다른것이

uid

euid - sudo와 같은 것 (임의로 루트권한 빌려오는것) 해당권한을 빌려줄수 있는 사람인가 아닌가

running - 현재 태스크가 구동이 되고있다.

- intrupt 를 수신할수 있다.
- 인터럽트를 무시하겠다.

lt seek 좀비

18분 - 이중연결리스트

→ 모두 vi -t task_struct → } → 144 해서 찾으면됨.

list_head 27 번

prio

signal_struct

mm_struct : 가상메모리, pgd () ,red_black트리로 메모리관리, v_mm_struct (세그먼트들 가상메모리 레이아웃관리), files_struct (

inode 5장에서 실제 디스크가 어떻게 연결되서 파일이라는 것을 보는지
super_block , inode

녹음 30분
thread_struct → cpu_context 로 요즘 바뀌었다.
실행 : vi -t thread_info , 4 번

녹음 39- 70page

p72
kill able - task가 task를 죽일 수 있는 상태 데몬
사용자가 9를 날려야 죽는다 → 시그널을 막고죽을것인가 아닌가

p73

첫번쨰 시스템콜 - 구동시키는것이 시스템콜 (유일한 소프트웨어 인터럽트) → 인터럽트가 발생하면 어쩌고저쩌고 유저한테 돌려준다.

두번째 인터럽트의 발생 - 사실 시스템콜도 인터럽트이다. 여기서말하는 인터럽트는 진짜 하드웨어 인터럽트이다. 동작 방식은 다르다.

시스메콜은 시스템콜 핸들러가 처리하고, 하드웨어는 하드웨어 마다 처리하는 별도의 핸들러들이 있다. 이것은 우리가 구현을 해야한다. 디바이스 드라이버까지..

73페이지 세번째

커널도 스택을 필요로한다 → thread_union 이다.

리눅스커널은 intel은 16kb

arm은 8kb 커널스택을 할당한다. 커널스택은 바로 thread_union 이다.

→ thread_info를 포함한다 이는 pd(프로세스 디스크립터) 라부른다.

74페이지

pt_regs 지워라x 표시

요기 들어가야할 것은

cpu_context_save cpu_context→ pt_legs를 대체한다.

레지를 저장하는 이유는? 꼬이게되니까

75페이지

문제 스케쥴링을 언제해야할까? 케이스 4개중 1

flag 비트에 1로 set되어 있는 경우 이당.

셋째 연기된 루틴들이 존재하면 이들을 수행한다.

논블럭킹에대한것들을 처리하겠다. 논블럭킹 방식으로 처리하는 것들이 뭐가있을까?

Bottom half

런큐와 스케쥴링

리눅스는 하이브리드방식을 사용한다. 140단계의 우선순위중 100개는 실시간으로 태스크를 제어한다.(리얼타임방식)

100~139까지는 동적우선순위로 태스크를 제어한다. 동적우선순위는 40개이다 키보드나 마우스가 이에 해당한다. 속도가 빨라야하므로

스케쥴링은 이런식으로 진행된다.

다음은 런큐이다

일반적으로 수행가능한상태 이중연결리스트였죠? 페이지 어디에있어요? 여러분은 봤어요 68페이 다에 있어요

태스크들을 어떻게 물려놨어요?

그리고 리눅스에서는 이 자료구조를 런큐방식으로 저장합니다

on_rq → 런큐에 있는지 없는지 여부

실제 런큐가 task_struct ?? 아님 mm_struct (linux_binfmt) 의위치가 갑자기 여기에 있었다.

mm_context_t → 메모리 context 암은 3번이나 4번을 보면된다. → vdso 가상메모리 관리해주는곳

메모리 컨텍스트가 이런것을 관리하는구나 하고 보면됨.

런큐를 찾아보자

:rq → cfs_rq 이녀석이 바로

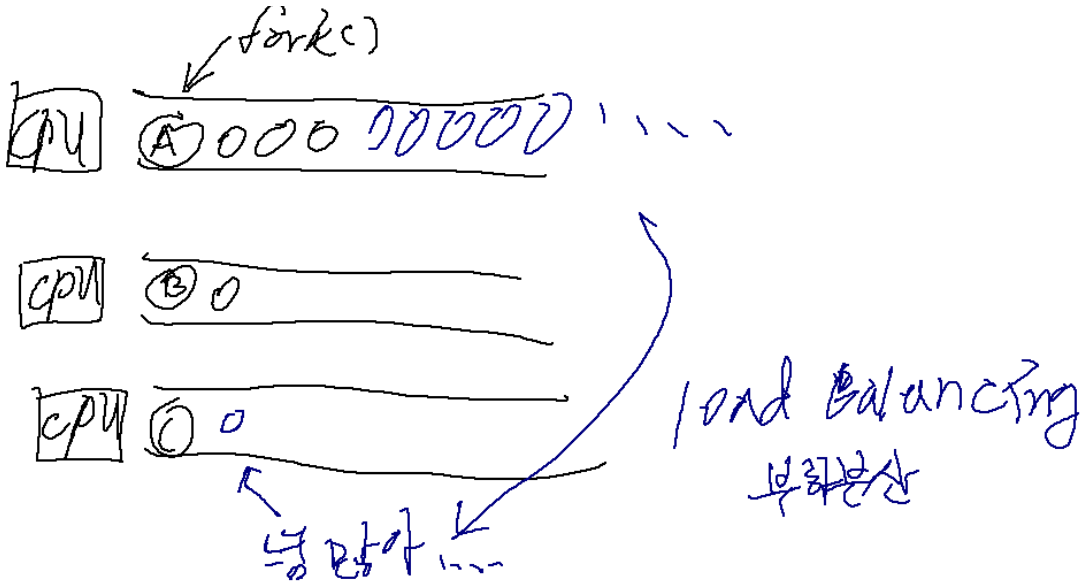
 rt_rq → 리얼타임 런큐이다   (어디에있냐면 sche_entity se; 와 sched_rt entity rt;

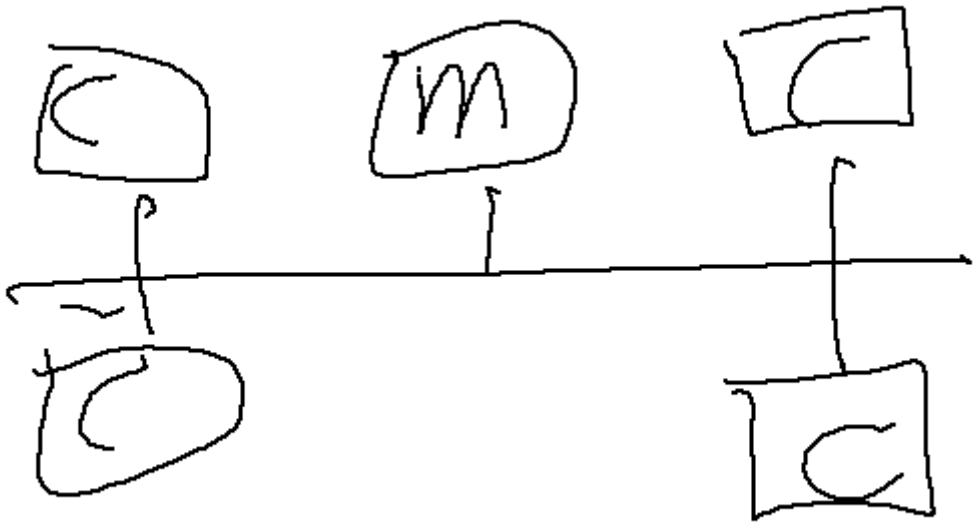
리얼타임 런큐가 무엇인지 적으시오

어디 밑에 밑에 rt_rq를 적으면됨 .

파일내에 리얼타임스케쥴을 하느냐 하이브리드라서 둘다 쓴다.

76페이지에 재밌는게 나온댄다..

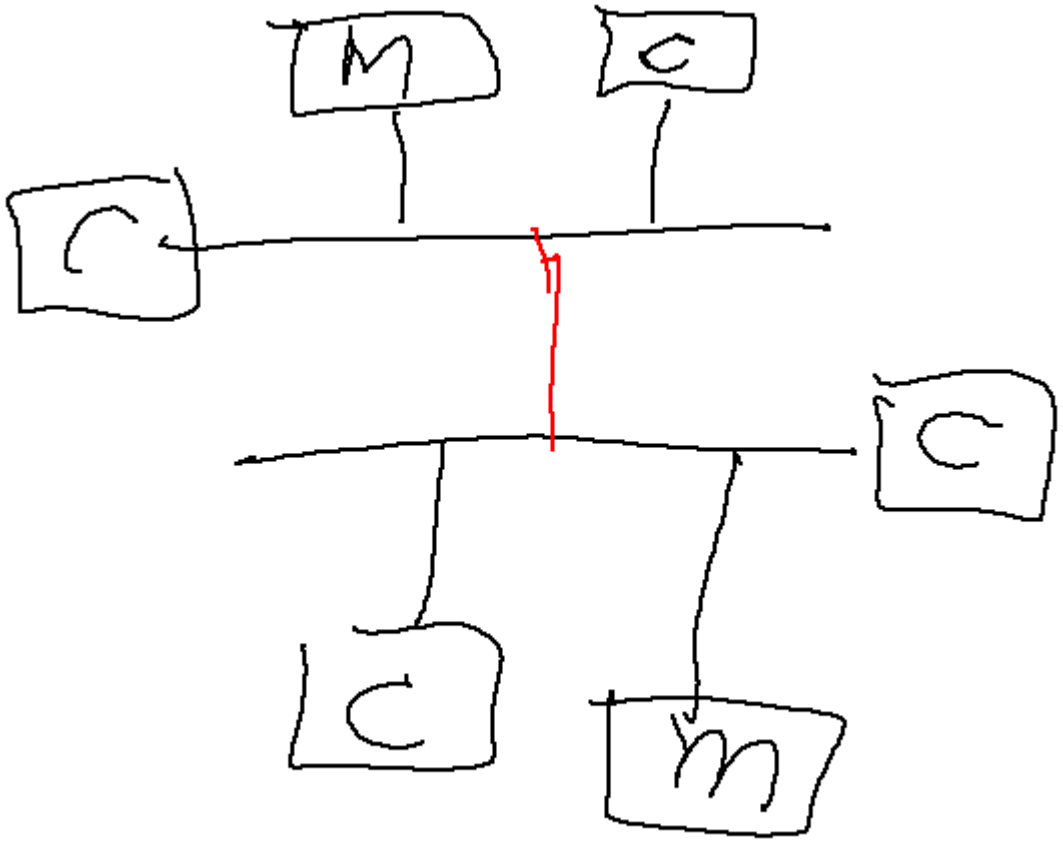




하이퍼쓰레딩 이란?

78페이지 녹음 1시간31분

SMP (UniforM Access) -모든 시피유들이 메모리를 공유한다.



누마(Non UniforM Access) - 속도가 다르다.

→ Heterogeneous Achitecture -이기종 시스템 구조

76~86 요기는 내일

87페이지 문맥교환

context switching

```
struct thread_struct {
                                                /* fault info     */
        unsigned long           address;
        unsigned long           trap_no;
        unsigned long           error_code;
                                                /* debugging      */
        struct debug_info       debug;
};
```

fault 제어를 위해 thread_struct가 존재한다.