TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 최대성
c3d4s19@naver.com

2018.04.09. – 033일차 수업

---

ctags를 이용하여fork(), vfork(), clone(), pthread_create()등의 함수 구현 방식 찾기

1. grep -rn "fork" ./ | grep SYSCALL

2. vi -t SYSCALL_DEFINE0

3. 16번 SYSCALL_DEFINE0(fork) 선택

4. _do_fork() 함수 확인 가능

```
long do_fork(unsigned long clone_flags,
        unsigned long stack_start,
        unsigned long stack_size,
        int __user *parent_tidptr,
        int __user *child_tidptr)
{
        return _do_fork(clone_flags, stack_start,
stack_size, parent_tidptr, child_tidptr, 0);
}
```

NPTL 조사

NPTL (Native POSIX Thread Library)는 리눅스의 스레드 구현이다. 레드햇에서 시작된 프로젝트로서, 비슷한 시기에 시작된 IBM의 NGPT (Next Generation POSIX Threads)는 폐기되었으나, NPTL은 계속 유지되어온 끝에 glibc에 포함됐다.

특징

- 1:1 모델

- 커널이 각 스레드 스택이 사용한 메모리 할당 해제 (메인 스레드 정리 이전에 앞선 스레도 모두 정리)

- 관리자 스레드 미사용으로 SMP, NUMA 등에서 확장성 증대

- 동기화에 futex 사용

- Shared memory 영역에서 동작하므로 다른 프로세스 간에 공유 가능 (PTHREAD_PROCESS_SHARED 매크로 참조)

- 모두 같은 PID

- 프로세스 단위의 시그널 처리

- 메인 스레드에 자원 사용이 보고되며, 이는 전체 프로세스에 반영됨

- ABI (Application Binary Interface)

- LD_ASSUME_KERNEL에 따라 LinuxThreads, NPTL 선택적 사용됨 (호환을 위한 것)

- getconf GNU_LIBPTHREAD_VERSION으로 Posix thread library 버전 확인

```
task_struct  // -> Task마다 생성됨
    state // -> Task 현재 상태
    files_struct  // ->
        files
            inode
            path
                dentry
                    inode
                        super_block  // -> 디스크 관리
                            s_dev
                            s_root
    mm_struct  // -> 메모리 관리
        pgd  // -> 물리메모리 페이지 테이블 관리(페이지 디렉토리 시작주소 들어있음)
        vm_area_struct  // -> 가상메모리(세그먼트) 관리
    list_head  // -> 이중연결리스트
    on_rq  // -> Run Queue에 있는지 없는지 확인
    prio, static_prio, normal_prio  // -> 우선순위
    sched_entity se  // -> 노멀, 동적우선순위 (100~139단계 우선순위)
        cfs_rq  // -> cfs 스케줄러 Run Queue
    sched_rt_entity rt  // -> real tmie 방식 (0~99단계 우선순위)
        rt_rq  // -> real time 스케줄러 Run Queue
    thread_struct thread  // -> 지금은 디버깅 용도로만 사용
        address
        trap_no  // -> fault number
        error_code
        debug
thread_union  // -> 커널 스택
    thread_info  // -> current의 task_struct의 정보를 가르키는 포인터(Process Descriptor)
        cpu_context_save  // -> context switching 용도
            __u32 r4, r5, r6, r7, r8, r9
            __u32 sl, fp, sp, pc, extra[2]
    syscall
```

Task Context (Task의 모든 정보가 들어있음)

-> 3가지 부분으로 구분가능함

1. System Context

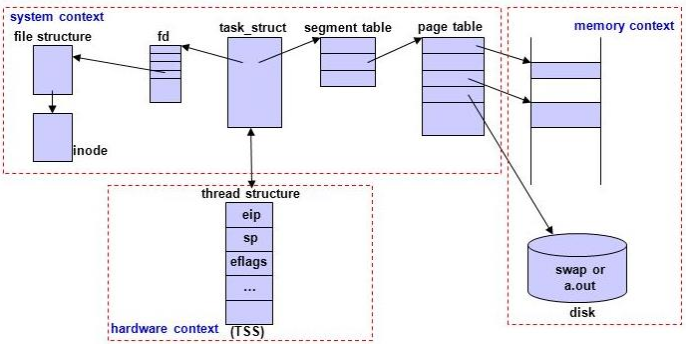
-> task_struct, fd(파일 디스크립터), 파일 테이블, 세그먼트 테이블, 페이지 테이블 등

2. Memory Context

-> 텍스트, 데이터, 스택, 힙 영역, 스왑 공간 등

3. Hardware Context

-> context switching할때 Task가 어디까지 실행했는지 저장해두는 공간



사용자 수준 실행상태에서 커널 수준 실행 상태로 전이 할 수 있는 방법

1. 시스템 호출(System Call)사용

-> 유일한 Software 인터럽트로 커널 system call handler가 직접 처리한다.

2. 인터럽트 발생

-> Hardware 인터럽트는 hardware마다 처리한다.

시스템 호출 또는 인터럽트 처리 후 리눅스 커널 할일

1. 현재 실행 중인 Task가 Signal을 받았는지 확인 후 필요시 Signal처리 핸들러를 호출함

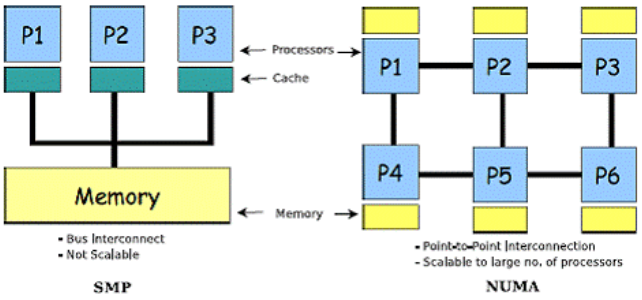
2. 다시 스케줄링 해야 하는 경우 스케줄러 호출

3. 커널 내에서 연기된 루틴(NON-Blocking 함수, Bottom Half -> NON-Blocking 방식의 인터럽트) 수행

하이퍼 쓰레딩(hyper-threading)

-> fork가 회로상에서 이루어진 것이라 보면 된다.

SMP(=UMA) / NUMA 비교



UMA -> 모든 CPU들이 메모리 공유함

NUMA -> CPU 부하와 메모리 접근 시간 차이 등을 고려하여 부하 균등을 시도한다.
