# TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)

gcccompil3r@gmail.com

학생 – 문한나

mhn97@naver.com

**NPTL(Native POSIX Thread Library )이란?**

NPTL 은 소위 1 × 1 스레드 라이브러리로, 사용자가 만든 스레드 (pthread_create () 라이브러리 함수를 통해)가 커널의 스케줄 가능한 엔티티 (Linux 의 경우 작업)와 1-1 로 대응한다.

특징

- 커널이 각 스레드 스택이 사용한 메모리 할당 해제 (메인 스레드 정리 이전에 앞선 스레도 모두 정리)
- 1 : 1 모델
- 관리자 스레드 미사용으로 SMP, NUMA 등에서 확장성 증대
- 동기화에 futex 사용
    - Shared memory 영역에서 동작하므로 다른 프로세스간에 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_KERNEL 에 따라 LinuxThreads, NPTL 선택적 사용됨 (호환을 위한 것)
- getconf GNU_LIBPTHREAD_VERSION 으로 Posix thread library 버전 확인

**실습)**

리눅스는 프로세스와 쓰레드의 자원을 관리하기 위해서 각 프로세스마다 task_struct 라는 자료구조를 생성한다.
(프로세스와 쓰레드 모두 태스크에 해당한다)
태스크가 생성되면 이 태스크를 위한 유일한 번호를 pid 로 할당해준다.
그런 뒤 만약 사용자가 프로세스를 원하는 경우라면 생성된 태스크의 tgid 값을 새로 할당된 pid 값과 동일하게 넣어준다. tgid 는 getpid 로 얻어올 수 있다.

```
821 /**
822  * sys_getpid - return the thread group id of the current process
823  *
824  * Note, despite the name, this returns the tgid not the pid.  The tgid and
825  * the pid are identical unless CLONE_THREAD was specified on clone() in
826  * which case the tgid is the same in all threads of the same group.
827  *
828  * This is SMP safe as current->tgid does not change.
829  */
830 SYSCALL_DEFINE0(getpid)
831 {
832         return task_tgid_vnr(current);
833 }
834
```

SYSCALL_DEFINE0(getpid) → 0 이 의미하는 것은 옵션의 갯수이다.
current 는 커널 내부에 정의되어 있는 매크로로써(arm 은 8kb 스택 할당받음) 현재 태스크의 task_struct 구조체를 가리킬 수 있게 해 준다. (thread info 가 가리키는 task)

```
44
45 /*
46  * low level task data that entry.S needs immediate access to.
47  * __switch_to() assumes cpu_context follows immediately after cpu_domain.
48  */
49 struct thread_info {
50         unsigned long           flags;          /* low level flags */
51         int                     preempt_count;  /* 0 => preemptable, <0 => bug */
52         mm_segment_t            addr_limit;     /* address limit */
53         struct task_struct      *task;          /* main task structure */
54         __u32                   cpu;            /* cpu */
55         __u32                   cpu_domain;     /* cpu domain */
56         struct cpu_context_save cpu_context;    /* cpu context */
57         __u32                   syscall;        /* syscall number */
58         __u8                    used_cp[16];    /* thread used copro */
59         unsigned long           tp_value[2];    /* TLS registers */
60 #ifdef CONFIG_CRUNCH
61         struct crunch_state     crunchstate;
62 #endif
63         union fp_state          fpstate __attribute__((aligned(8)));
64         union vfp_state         vfpstate;
65 #ifdef CONFIG_ARM_THUMBEE
66         unsigned long           thumbee_state;  /* ThumbEE Handler Base register */
67 #endif
68 };
```

```
18
19 #define THREAD_SIZE_ORDER       1
20 #define THREAD_SIZE             (PAGE_SIZE << THREAD_SIZE_ORDER)
21 #define THREAD_START_SP         (THREAD_SIZE - 8)
22
```

```
90
91 static inline struct thread_info *current_thread_info(void)
92 {
93         return (struct thread_info *)
94                 (current_stack_pointer & ~(THREAD_SIZE - 1));
95 }
96
```

PAGE_SIZE 는 4KB THREAD_SIZE_ORDER 의 정의는 1
비트연산 후 8KB 가 된다. 즉 8KB 의 스택을 할당받았다.

```
#define _GNU_SOURCE //clone 사용 시 필요
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
#include <sched.h>

int sub_func(void *arg){

        printf("TGID(%d), PID(%d) : Child \n", getpid(), syscall(__NR_gettid));
        sleep(2);
        return 0;

}

int main(void){

        int pid;
        int child_a_stack[4096], child_b_stack[4096];

        printf("before clone \n\n");
        printf("TGID(%d), PID(%d) : Parnet \n",getpid(), syscall(__NR_gettid));

        clone(sub_func, (void *)(child_a_stack+4095), CLONE_CHILD_CLEARTID | CLONE_CHILD_SETTID,
NULL);
        clone(sub_func, (void *)(child_b_stack+4095), CLONE_VM | CLONE_THREAD | CLONE_SIGHAND,
NULL);

        sleep(1);
```

```
        printf("after clone \n\n");

        return 0;

}
```

clone 은 주는 옵션에 따라 프로세스, 쓰레드 둘 다 만들 수 있다.
clone()의 인자로 CLONE_CHILD_CLEARTID 와 CLONE_CHILD_SETTID 를 설정하면 리눅스 커널은 태스크를 생성할 때 프로세스로 해석될 수 있도록 자원 공유가 되지 않는 형태로 생성한다.
만약 CLONE_THREAD 로 설정하면 태스크 생성 시 쓰레드로 해석될 수 있도록 자원 공유가 되는 형태로 생성한다.

```
mhn@mhn-Z20NH-AS51B5U:~/kernel/linux-4.4$ ./a.out
before clone

TGID(6278), PID(6278) : Parnet
TGID(6278), PID(6280) : Child
TGID(6279), PID(6279) : Child
after clone
```

쓰레드와 프로세스가 생성되었다.

| | reader(=process) | thread |
|---|---|---|
| tgid | 100 | 100 |
| pid | 100 | 300 |

**태스크 문맥**

시스템 문맥(System Context) - tssk_struct, 파일 디스크립터, 파일 테이블, 세그먼트 테이블, 페이지 테이블 등 추상화된 것들을 관리한다. 만약 이것을 관리해주지 않는다면 다른프로세스가 덮어씌어 네트워크가 끊겨버릴 수도 있다.
메모리 문맥(Memory Context) – 텍스트, 데이터, 스택, heap 영역, 스왑 공간 등이 포함된다.
하드웨어 문맥(Hardware Context) – Context Switching 할 때 태스크의 현재 실행 위치에 대한 정보를 유지한다.

**task_struct 구조 살펴보기**

1. task identification - 태스크를 인식하기 위한 변수들
    pid, tgid, uid(사용자 id), euid(유효 사용자 id) 등
2. state - 태스크를 관리하기 위한 변수들
    TASK_RUNNING(0) - 현재 task 가 run queue
    TASK_INTERRUPTIBLE(1) – 현재 task 가 interrupt 수신해도 지장이 없다
    TASK_UNINTERRUPTIBLE(2) - interrupt 수신을 허용하지 않음
    TASK_STOPPED(4) - ctrl + z (과도하게 사용하면 뻗을 수 있으니 조심하자)
    TASK_TRACED(8) - 디버깅 상태
    EXIT_DEAD(16) – 종료되었을 때 ex)return 0,signal 등
    EXIT_ZOMBIE(32) - 좀비 프로세스
3. task relationship – 가족관계도
4. scheduling information - 스케줄링 관련 변수 prio, policy, cpus_allowed, time_slice, rt_priority
5. signal information - 시그널 관련 변수 signal, sighand, blocked, pending
6. memory information - 데이터의 위치와 크기, 접근 제어 정보 등을 관리하는 변수

7.file information – 태스크가 오픈한 파일들은 task_struct
files_strcut
files 이름의 변수로 접근 가능
inode 는 디스크에서 파일 위치를 알려줌
8. cpu_context_save – context_switching 시 태스크가 어디까지 실행되었는지 기억해놓는 공간
9. time information - 태스크의 시간 정보를 위한 변수 ex)start_time, real_start_time 등
10. format – personality 같은 변수를 사용하여 도메인 지원
11. resource limits - 태스크가 사용할 수 있는 자원의 한계