TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-04-09 (33 회차)

강사: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생: 정유경
ucong@naver.com

*. task_struct 는 fork, pthread_create 할 때 생성된다.
*. 프로세스와 쓰레드가 각각 생성될때 마다 task_struct 자료구조를 생성
*. 한프로세스 내의 쓰레드는 동일한 PID 를 공유, 프로세스와 쓰레드를 구별하려면 TGID 사용

[Ch 3 : 내용정리 및 복습]

3. 프로세스와 쓰레드의 생성과 수행
프로세스의 생성 : fork( ), vfork( )
프로세스가 생성되면 주소공간을 포함한 이 프로세스를 위한 모든 자원들이 새롭게 할당 됨
자식프로세스의 연산 결과는 부모 프로세스 주소공간의 변수에는 영향을 주지 않는다.
부모프로세스와 자식프로세스는 서로 다른 주소공간을 갖는다.
자식프로세스의 결함은 부모프로세스에 전파되지 않음
→ 프로세스는 **결함 고립**에 적합한 프로그래밍

쓰레드의 생성 : clone( ), pthread_create( )
쓰레드는 자신을 생성한 태스크와 동일한 pid 를 갖는다.
자신이 생성한 쓰레드가 변수를 수정하면 그 결과를 볼 수 있다.
쓰레드 생성시 기존 부모쓰레드와 자식쓰레드는 서로 같은 주소공간을 공유한다.
→ 쓰레드는 **자원 공유**에 적합한 프로그래밍 모델

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

int g =2;

int sub_func(void *arg)
{
        g++;
        printf("PID(%d): Child g=%d\n", getpid(),g);
        sleep(2);
        return 0;

}

int main(void)
{
        int pid;
        int child_stack[4096];
        int l=3;
        printf("PID(%d): Parent g=%d, l=%d\n", getpid(),g,l);
        clone(sub_func, (void*)(child_stack+4095), CLONE_VM | CLONE_THREAD|CLONE_SIGHAND,NULL);
/*쓰레드는 sub_func()를 실행하고 종료한다.
스택이 필요하므로 4096bytes(물리메모리 최소단위 4KB)할당, 배열시작은 0 번부터니까 4095
VM 쓰겠다|프로세스가 아닌 쓰레드로 만들겠다(태스크를 생성할대 쓰레드로 해석될 수 있고록 자원 공유가 되는 형태로 생성한다 -
65p)|시그널처리하겠다*/
        sleep(1);
        printf("PID(%d): Parent g=%d, l=%d\n", getpid(),g,l);
        return 0;
}
```

*. vfork()를 사용하는 이유

: fork()하면 부모프로스세의 메모리 영역(페이지 테이블 영역?) 복사하지만, vfork()사용하면 가상메모리 레이아웃 제외한 정보들만 복사. 메모리레이아웃은 exec()할때 잡음→ 최적화 방법의 하나

최근 리눅스는 fork()시 COW(Copy-On-Write) 기법을 도입하여 fork()시 발생하는 주소공간 복사비용을 많이 줄임 → 현재 vfork()는 거의 사용되지 않는다고 함

*. COW(Copy-On-Write): fork()는 부모의 모든 자원을 복제하여 자식 프로세스로 넘겨주는데, 많은 자원을 복제한다는 점에서 너무 단순하고 비효율적. Copy-On-Write 란 데이터의 복제를 지연 또는 방지하는 기법. 즉 프로세스 주소공간을 복제하는 대신, 부모와 자식이 하나의 공간을 공유. 만약 데이터를 써넣을 일이 발생하면, 그제서야 주소 공간을 복제하여 자식에게 넘겨줌. 결국 자원의 복제는 쓰기가 발생할 때에만 일어나게 되고 복제 전까지는 부모와 자식이 읽기 전용 주소공간을 공유. 이 기법은 대량의 데이터 복제를 방지하여 성능을 최적화.

4. 리눅스의 태스크 모델

*. 리눅스는 프로세스가 생성되든 쓰레드가 생성되든 **tast_struct** 라는 자료구조(구조체)를 생성하여 관리
즉, 프로세스이던 쓰레드이던 커널 내부에서는 **태스크**라는 객체로 관리
→ fork, vfork, clone, pthread_create 무엇을 하든 커널 내부에서 마지막으로 호출되는 함수는 **do_fork()**로 동일
*. do_fork()함수는 이름표를 만든다 → task_struct 구조체에 pid, ppid, 메모리레이아웃 등 자세한 정보를 기록

1. 61p.c

```
/* 61p.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>

int main(void)
{

        int pid;
        printf("before fork()\n\n");
        if(pid=fork() < 0)
        {
                printf("fork error\n");
                exit(-2);
        }
        else if(pid ==0)
        {
                printf("TGID(%d), PID(%lu): Child \n", getpid(), syscall(__NR_gettid));
        }
```

```
        else
        {
                printf("TGID(%d), PID(%lu): Parent\n", getpid(), syscall(__NR_gettid));
                sleep(2);
        }

        printf("after fork\n\n");
        return 0;
}
```

```
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$ vi 61p.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$ gcc 61p.c
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$ ./a.out
before fork()

TGID(7779), PID(7779): Child
after fork

TGID(7780), PID(7780): Child
after fork

yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$
```

2. 62p.c

```
/* 62p.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>

int main(void)
{
        int pid;
        printf("before vfork\n\n");

        if((pid=vfork()) < 0)
        {
                printf("fork error\n");
                exit(-2);
        }

        else if(pid ==0)
        {
                printf("TGID(%d), PID(%lu): Child \n", getpid(), syscall(__NR_gettid));
                _exit(0);
        }
        else
        {
                printf("TGID(%d), PID(%lu): Parent\n", getpid(), syscall(__NR_gettid));
        }

        printf("after vfork\n\n");

        return 0;
```

```
}
```

```
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$ ./a.out
before vfork

TGID(7838), PID(7838): Child
TGID(7837), PID(7837): Parent
after vfork

yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$
```

3. 63p.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
#include <pthread.h>


void *t_function(void* data)
{

        int id;
        int i=0;
        pthread_t t_id;
        id = *((int*)data);
        printf("TGID(%d), PID(%lu), pthread_self(%ld): Child\n", getpid(), syscall(__NR_gettid), pthread_self());
        sleep(2);
}

int main(void)
{

        int pid, status;
        int a=1;
        int b=2;

        pthread_t p_thread[2];
        printf("before pthread create\n\n");
        if((pid=pthread_create(&p_thread[0], NULL, t_function, (void*)&a))<0)
        {
                perror("thread create error: ");
                exit(1);
        }

        if((pid=pthread_create(&p_thread[1], NULL, t_function, (void*)&b))<0)
        {
                perror("thread create error: ");
                exit(2);
        }
```

```
        pthread_join(p_thread[0], (void**)&status);
        printf("pthread_join(%d)\n", status);
        pthread_join(p_thread[1], (void**)&status);
        printf("pthread_join(%d)\n", status);
        printf("TGID(%d), PID(%lu): Parent\n", getpid(), syscall(__NR_gettid));

        return 0;
}
```

4. 64p.c

```
#define _GNU_SOURCE
#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(%lu): 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(%lu): Parent\n", getpid(), syscall(__NR_gettid));
/*clone() 을 이용한 프로세스와 쓰레드 생성*/
        clone(sub_func, (void*)(child_a_stack+4095), CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID, NULL);
/* 태스크 생성시 프로세스로 해석될 수 있도록 자원공유가 되지 않는 형태로 생성 - 65p*/
        clone(sub_func, (void*)(child_a_stack+4095), CLONE_VM|CLONE_THREAD|CLONE_SIGHAND, NULL);
```

```
        sleep(1);
        printf("after clone\n\n");
        return 0;

}
```

```
yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$ ./a.out
before clone

TGID(7957), PID(7957): Parent
TGID(7957), PID(7959): Child
TGID(7958), PID(7958): Child
after clone

yukyoung@yukyoung-Z20NH-AS51B1U:~/Workspace/0409$
```

*. task_struct 내의 tgid 값을 출력하는 함수는 getpid()
task_struct 내의 pid 값을 출력하는 함수는 gettid()

*. 커널내부에서 **getpid()함수** → task_tgid_vnr(current) : task_struct 구조체의 **tgid** 필드 리턴
- current 매크로는 현재 태스크의 task_struct 구조체를 가리킴 - 65p

5. 태스크 문맥 : 태스크와 관련된 '모든' 정보
1) 시스템 문맥 : 태스크의 정보를 유지하기 위해 커널이 할당한 자료구조들 (task_struct, 파일 디스크립터, 파일 테이블, 세그먼트 테이블, 페이지 테이블 등)

2) 메모리 문맥 : 텍스트, 데이터, 스택, heap 영역, 스왑 공간

3) 하드웨어 문맥 : (지금까지 배워왔던 것)문맥 교환 할 때 태스크의 현재 실행 위치에 대한 정보를 유지 ← '쓰레드 구조' or '하드웨어 레지스터 문맥' 이라 불림

[task_struct 자료구조의 각 변수 이름들 – 관련있는 것끼리]

가. task identification : 태스크 ID 를 나타내는 pid, 태스크가 속해있는 쓰레드 그룹 ID 를 나타내는 tgid, 태스크에 대한 접근권한을 제어하는 데 이용되는 uid(사용자 ID), euid(유효 사용자 ID)

나. state : 태스크는 생성~소멸까지 많은 상태를 거치며, 이를 관리하기 위한 state 변수가 존재

TASK_RUNNING: 현재 태스크가 런큐에 위치, 준비&실행의 두가지 상태로 나뉨 → 보다 높은 우선순위를 가지는 태스크로 인해 준비상태로 전환 이때 리눅스는 여러 태스크들이 공**정**하게 CPU 를 사용할 수 있도록 CFS 기법을 사용

TASK_INTERRUPTIBLE: 인터럽트 수신가능, 크게 지장 없음

↔ TASK_UNINTERRUPTIBLE: 허용안함, 대기하는 동안 자신이 기다리는 그 사건 외에는 일체 방해받아선 안 되는 경우, 이후 기다리던 사건 발생 시 대기 상태에 있던 TASK 가 다시 준비(TASK_INTERRUPTIBLE)상태로 전이하게 되며 다시 다른 태스크들과 함께 스케줄링 되기 위해 경쟁함.

*. TASK_UNINTERRUPTIBLE 상태의 Task 가 시그널에 반응하지 않기때문에 생기는 문제점

'kill -9 pid' 명령으로도 태스크를 종료시킬 수 없다. → TASK_KILLABLE 상태 도입 : SIGKILL 과 같은 fatal signal 에 반응

TASK_STOPPED: 일시정지 (ctrl+z: 많이 하지 말 것 시스템 뻗는다)

ex. SIGSTOP 시그널을 받으면 STOPPED 상태로 전이 추후 SIGCONT 시그널을 받아 다시 RUNNING 상태로 전이 -71p

TASK_TRACED: 디버깅상태 ex. 디버거의 **ptrace()** 호출에 의해 TRACED 상태로 전이-71p

EXIT_DEAD: 정상종료, exit(), 시그널 호출로 종료

*. 부모 task 의 wait()함수 호출 전 먼저 사라지게 되면 좀비상태의 자식 태스크 즉, 고아 태스크가 영원히 존재해 시스템의 오버헤드로 작용: 신이 처리해주어야 한다 (신: 최초의 프로세스, 1 번 프로세스) 고아 task 의 부모를 init task 로 바꾸어줌 (그래서 task_struct 구조체에 real_parent 와 parent 2 개의 필드가 존재)

EXIT_ZOMBIE: Task 에게 할당되어 있던 자원을 대부분 커널에게 반납한 상태 , 부모 Task 가 wait()을 호출하면 자식 태스크의 상태가 EXIT_DEAD 로 바뀜

다. task realationship - 태스크는 이중연결리스트로 연결되어 있음(prev, next)

라. scheduling information - task_struct 에서 스케줄링과 관련된 변수 :prio, policy, cpus_allowed

마. signal information - 시그널은 비동기!! (관련 변수 : signal, sighand, blocked, pending)

바. memory information : task_struct > mm_struct

사. file information

- 태스크가 오픈한 파일들은 task_struct > files_struc 구조체 형태인 **files** 라는 이름의 변수로 접근 가능

- 루트 디렉터리의 inode 와 현재 디렉터리의 inode 는 fs_struct 구조체 형태인 **fs** 라는 변수로 접근 가능

아. CPU_context_save : 문맥 교환을 수행할 때 태스크가 현재 어디까지 실행되었는지 기억해놓는 공간

자. time information

차. format

- 이진 포맷을 지원: bin fmt 구조체

- int personality: 하위 호환을 위해 존재

*. file [실행파일] : ELF (디버깅포맷:DWARF)

6. 상태전이와 실행수준 변화

*. task_struct 구조체 > state : 태스크의 상태를 나타냄

*. 사용자수준 실행상태에서 커널 수준 실행 상태로 전이할 수 있는 방법 2 가지

1) 시스템호출 - 유일한 sw 인터럽트, 인터럽트는 커널이 처리

2) hw 인터럽트의 발생

*. 커널수준의 코드도 스택(thread_union) 필요
리눅스 커널은 태스크별로 커널 스택 8KB(ARM) or 16KB(인텔)의 스택을 할당
8KB 의 스택은 thread_union 이라 불림
*. tread_union = thread_info(프로세스 디스크립터)구조체 + 커널스택 관련.
thread_info (*task: task_struct 가리킴, flags: 스케쥴링 할지말지 여부)
*. pt_regs 대체 → cpu_sntext_save

→ 결론적으로 태스크가 생성되면 task_struct 구조체, 커널스택(thread_union)이 할당됨

*. 커널이 커널수준 실행상태에서 사용자수준 실행상태로 전이 할 때 처리하는 일들 – 75p
1) 시그널 받았는지 확인하고 시그널 핸들러 호출(tast_struct > sig_struct, sighand_struct)
2) 스케줄링해야한다면 스케줄러 호출 (need_resched 플래그 확인)
3) 커널 내 연기된 루틴 존재하면 수행 (논블록 & Bottom Half)

7. 런큐와 스케쥴링

*. 리눅스의 태스크는 실시간 태스크와 일반 태스크로 나뉘며 각각 스케줄링 알고리즘이 구현
0~ 99 단계: RT 방식
100~139: 동적우선순위 - 우선순위를 상황에 따라 높인다.
*. task_struct > RT_entity or RT_rq

[1] 런큐와 태스크
*. 운영체제는 스케줄링 작업 수행을 위해, 수행 가능한 상태의 태스크를 자료구조를 통해 관리
→ 이 자료구조를 런 큐(Runqueue)라 함, ~/kernel/sched/sched.h 파일 내에 struct rq 라는 이름으로 정의, 부팅 이후 각 CPU 별로 하나씩의 런큐를 유지

*각각의 CPU 런큐에서 다음번 수행시킬 태스크를 고를 때 고려사항

1) 새로 생성된 태스크는 부모 태스크가 존재하던 런 큐로 삽입 (더 높은 캐시 친화력 (cache affinity)를 얻을 수 있기 때문)

2) 런 큐간의 부하가 균등하지 않은 경우 → 부하 균등 (load balancing) :특정 CPU 가 많은 작업을 수행하고 있느라 매우 바빠 한가한 다른 CPU 에 태스크를 ‘이주’ 시켜서 성능 향상

*. 하이퍼 스레딩

- fork()가 회로로 구현되어 있다 , 실제 CPU 는 4 개인데 8 개처럼 동작하게 만든것

- 코어에서 스레드를 실행하고 있는 중에, 다른 스레드가 실행되기를 원하는데 그 기능이 코어에서 놀고있는 기능이면 동시에 실행해 준다. 두 스레드가 같은 기능을 요청하면 코어에서는 하나의 스레드만 실행 된다.

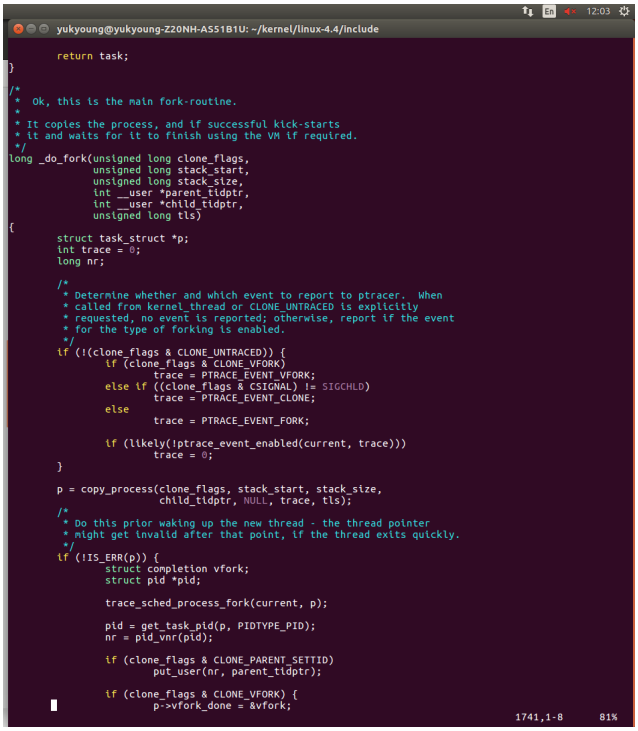
*. 멀티프로세싱 기법 → SMP (UMA): NUMA

UMA: 메모리 접근속도 같다. 메모리에 접근하기위한 버스가 1 개

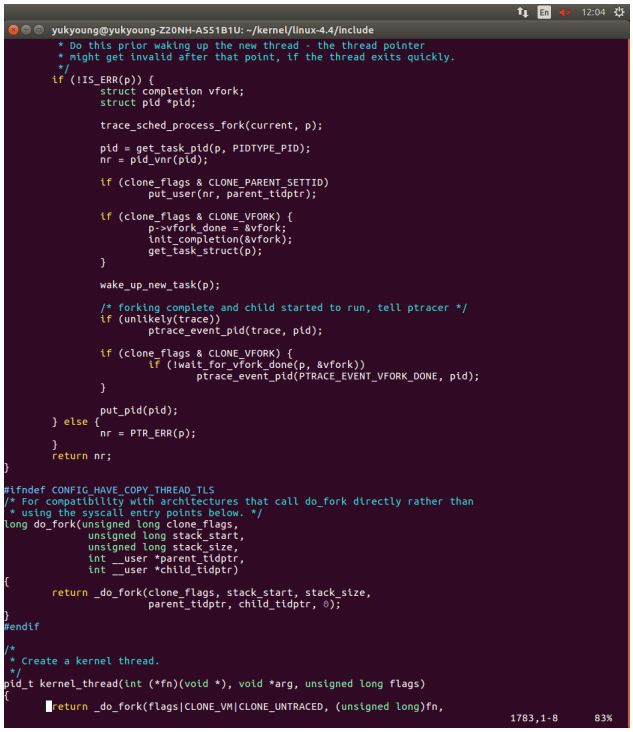
NUMA: 메모리에 접근하기 위한 버스가 2 개→ 네트워크 BUS 로 정보를 주고 받는다.

메모리 접근속도가 다르다, Heterogeneous(이기종) Architecture 라고 한다, 멀티코어에서 더 발전한 형태

[과제] sys_fork 분석 (미완료)



```
yukyoung@yukyoung-Z20NH-AS51B1U: ~/kernel/linux-4.4/include

        return task;
}

/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long _do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr,
              unsigned long tls)
{
        struct task_struct *p;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the event
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if ((clone_flags & CSIGNAL) != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }

        p = copy_process(clone_flags, stack_start, stack_size,
                         child_tidptr, NULL, trace, tls);
        /*
         * Do this prior waking up the new thread - the thread pointer
         * might get invalid after that point, if the thread exits quickly.
         */
        if (!IS_ERR(p)) {
                struct completion vfork;
                struct pid *pid;

                trace_sched_process_fork(current, p);

                pid = get_task_pid(p, PIDTYPE_PID);
                nr = pid_vnr(pid);

                if (clone_flags & CLONE_PARENT_SETTID)
                        put_user(nr, parent_tidptr);

                if (clone_flags & CLONE_VFORK) {
                        p->vfork_done = &vfork;
                                                        1741,1-8      81%
```



```
yukyoung@yukyoung-Z20NH-AS51B1U: ~/kernel/linux-4.4/include

         * Do this prior waking up the new thread - the thread pointer
         * might get invalid after that point, if the thread exits quickly.
         */
        if (!IS_ERR(p)) {
                struct completion vfork;
                struct pid *pid;

                trace_sched_process_fork(current, p);

                pid = get_task_pid(p, PIDTYPE_PID);
                nr = pid_vnr(pid);

                if (clone_flags & CLONE_PARENT_SETTID)
                        put_user(nr, parent_tidptr);

                if (clone_flags & CLONE_VFORK) {
                        p->vfork_done = &vfork;
                        init_completion(&vfork);
                        get_task_struct(p);
                }

                wake_up_new_task(p);

                /* forking complete and child started to run, tell ptracer */
                if (unlikely(trace))
                        ptrace_event_pid(trace, pid);

                if (clone_flags & CLONE_VFORK) {
                        if (!wait_for_vfork_done(p, &vfork))
                                ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
                }

                put_pid(pid);
        } else {
                nr = PTR_ERR(p);
        }
        return nr;
}

#ifndef CONFIG_HAVE_COPY_THREAD_TLS
/* For compatibility with architectures that call do_fork directly rather than
 * using the syscall entry points below. */
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);
}
#endif

/*
 * Create a kernel thread.
 */
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
        return _do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn,
                                                        1783,1-8      83%
```

[과제] NPTL (Native POSIX Thread Library) 60p ← ??

→ 뛰어난 성능을 가짐, 태스크의 속성에 따라 쓰레드는 자원공유의 장점을 충분히 이용하도록 구현되었기 때문

*. 쓰레드 모델은 크게 세가지 user-level threads (1-on-N), kernel supported threads (1-on-1), multi-level (userland/kernel hybrid) threads (M-on-N)

*. 커널쪽에서 보면 하나의 쓰레드고 사용자쪽에서 보면 여러 개의 쓰레드인 모델을 1-on-N(1 대 N) 모델→ 사용자레

벨 쓰레드 , 커널쪽과 사용자쪽의 쓰레드가 1 대 1 로 대응하는 모델 → 1-on-1 , 커널 쓰레드의 개수와 사용자 쓰레드의 개수가 1 대 1 대응관계를 이루지 않는 모델→ M-on-N 모델

*. 사용자 쓰레드 방식이 커널 쓰레드보다 오버헤드가 적은 이유는 쓰레드간을 전환할 때마다 커널 스케줄러를 호출할 필요가 없기 때문. 커널 스케줄러로 진입하려면 프로세서 모드를 사용자 모드에서 커널 모드로 전환해야 하는데, 이때 사용자쪽 하드웨어 레지스터를 전부 저장시키고, 커널레지스터를 복구하는 등의 수많은 작업이 일어남.
따라서 사용자 모드와 커널 모드를 많이 왔다갔다 할 수록 성능은 급격하게 떨어짐
사용자 쓰레드는 쓰레드 스케줄러가 사용자 모드에만 있기 때문에 그런 오버헤드는 발생하지 않음. 그런데 사용자 쓰레드의 결정적인 단점은 프로세스내의 한 쓰레드가 커널로 진입하는 순간 나머지 쓰레드들도 전부 정지된다는 점.
커널이 쓰레드의 존재를 알지 못하므로 불가피한 현상. 쓰레드가 커널로 진입할 때는 write(), read(), ...같은 시스템 호출을 부를 때인데, 시스템 호출 길이가 짧아서 바로 리턴할 때는 문제가 없지만 연산이 길어지면 상당한 문제.
전체 프로세스의 응답성이 상당히 떨어짐.
*. 커널 쓰레드를 쓰면 멀티프로세서를 활용할 수 있다는 큰 장점. 사용자 쓰레드는 CPU 가 아무리 많더라도 커널 모드에서 쓰레드 단위로 스케줄이 안되므로 각 CPU 에 효율적으로 쓰레드를 배당할 수 없음 (프로세스 단위로만 배당이 됨)

*. 사용자 쓰레드는 리눅스에서는 별로 쓰는 사람이 없는 것 같고, 1-on-1 커널 쓰레드 방식인
glibc 내의 LinuxThreads 가 가장 널리 쓰임 → 이걸 더 발전시킨 것이 NPTL(Native POSIX Threading Library)

*. 커널 2.6 에서는 NPTL 이라는 Native 스레드를 구현함으로써 지금까지 리눅스 스레드의 문제점을 해결하고 대폭적인 성능 향상을 이룸 ← 설계에 있어서 커널 스레드와 사용자 레벨 스레드 간의 관계를 어떻게 둘 것인가 주요 쟁점(1:1 관계 or M:N 관계) 후자의 경우 M 개의 커널 스레드와 N 개의 커널 스레드가 상관성을 가짐, 이 두 종류의 스레드를 위한 두 개의 스케줄러가 필요. 사용자 레벨 스케줄러는 커널 스케줄러에게 정보를 전달할 수 있을 것이고 커널 스케줄러는 결정된 내용을 사용자 레벨 스케줄러에게 알려줄 것. 그런데 M:N 은 리눅스에서는 적합하지 않음. 이를 구현하기에는 너무 많은 비용이 듬 (사용자 레벨에서 스케줄을 하기 위해서는 커널 공간의 레지스터 내용을 복사해 와야 하는 등의 문제가 있기 때문) → **결론적으로 NPTL 은 1:1 관계를 채택하고 있음.**

cf. M:N 방식으로는 **NGPT** 라는 Next Generation POSIX Threading 이 있음.