



**Xilinx Zynq FPGA, TI DSP,  
MCU 기반의  
프로그래밍 전문가 과정**

날 짜 : 2018 . 4 . 9

강사 – Innova Lee(이상훈)  
[gcccompil3r@gmail.com](mailto:gcccompil3r@gmail.com)

학생 – 정한별  
[hanbulkr@gmail.com](mailto:hanbulkr@gmail.com)

## < 리눅스 커널 내부 구조\_ Chapter\_ 3>

### vfork ()

fork 의 경우 부모의 메모리를 복사 하는데 exce 하면 덮어쓴다. 그러면 굳이 부모꺼를 복사하고 다시 덮어 쓸 필요가 없다. 이런 낭비를 아끼기 위해서 쓰는 것이 vfork 이다.

vfork 를 쓰면 부모거를 복사하지 않고 자식 프로세스를 만들 수가 있다.

( fork() ->copy on write 기법 , 필요한 만큼 복사해서 씀, process 주소 공간 복사 대신 process 간 같은 공간을 공유한다. 이 방법으로 주소 공간 복사 비용을 많이 줄였다.)

// clone.c //

#define \_GNU\_SOURCE // 맨꼭대기에 있어야함.

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

```
int g=2 ;
```

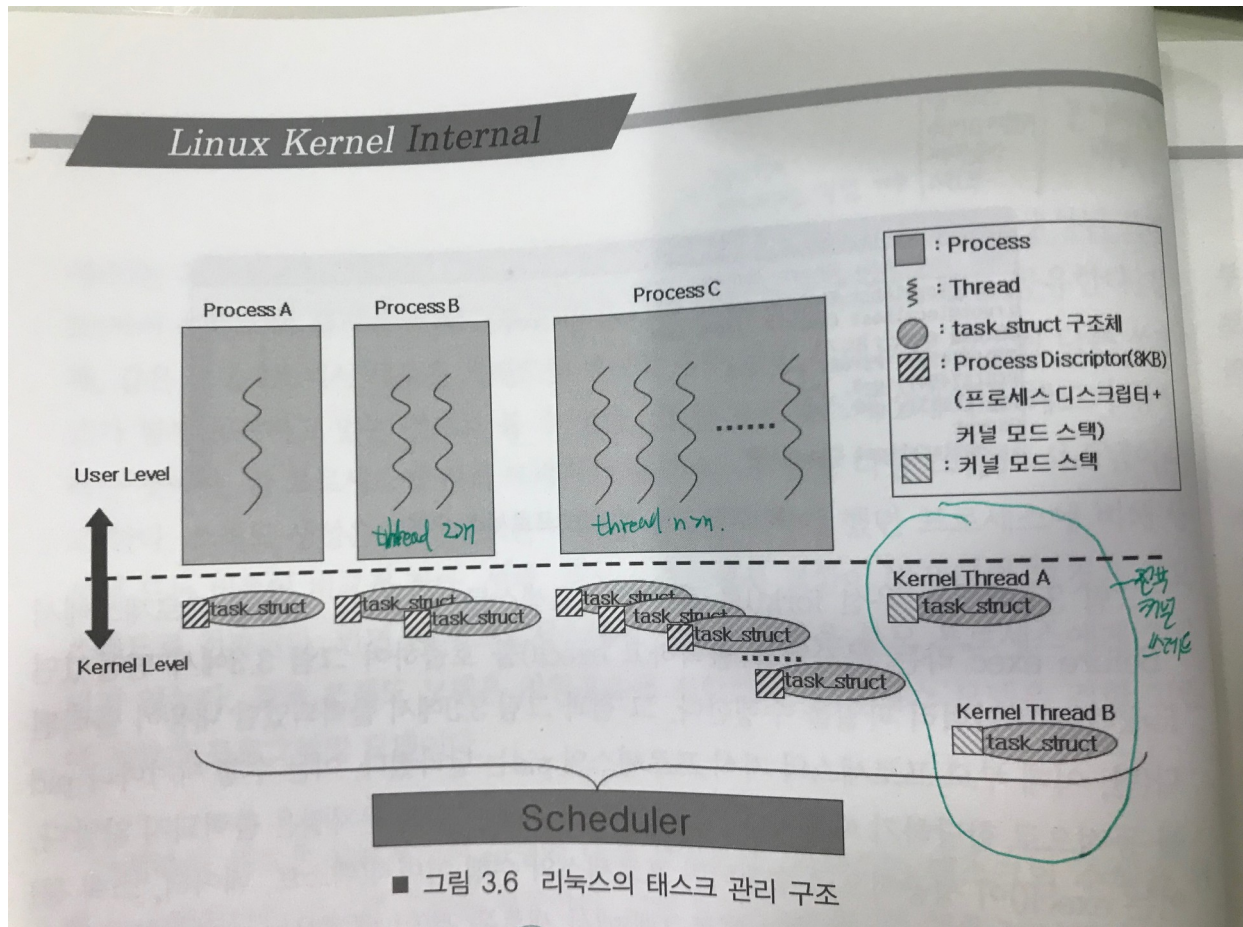
// arg 에는 NULL 이 들어간다.

```
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 i = 3;
    printf("PID(%d) : Parent g = %d, i = %d \n", getpid(), g,i);
    // clone 은 #define _GNU_SOURCE
    //      #define <sched.h>      을 하라고 man 에 써있다.
    // 4096 은 스택 크기를 물리메모리 초소단위인 4k 단위만큼 할당하는 것이다.
    clone (sub_func, (void*)(child_stack +4095), CLONE_VM | CLONE_THREAD |
CLONE_SIGHAND, NULL);
    sleep(1);
    printf("PID(%d) : Parent g= %d, i= %d \n", getpid(), g, i);
    return 0;
}
```

## < 리눅스 태스크 모델 >

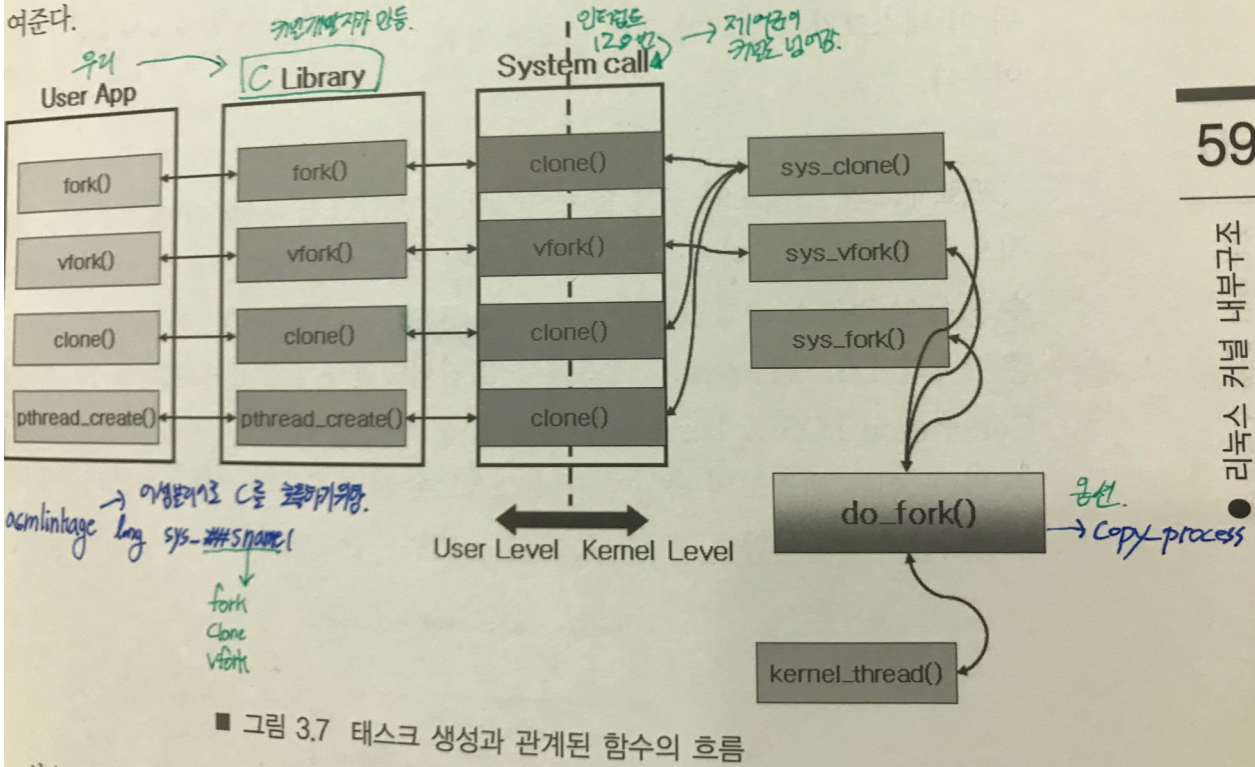


- 프로세스는 자신이 사용하는 자원과 그자원에서 수행되는 수행 흐름으로 구성된다.  
이를 관리하기 위해 각 프로세스마다 **task\_struct** 라는자료 구조를 생성한다.

- 결국 리눅스 커널은 프로세스 또는 쓰레드 중에서 어떤것이 요청될 지라도, 모두 task\_struct 자료 구조로 동일하게 관리 한다.

것이다.

이러한 특성은 실제 함수들이 구현된 방식에서도 나타난다. 그림 3.7은 리눅스에서 `fork()`, `vfork()`, `clone()`, `pthread_create()` 등의 함수들이 구현되어 있는 방법을 보여준다.



`fork()`는 프로세스를 생성하는 함수 이고, `clone()`은 쓰레드를 생성하는 함수인데 **커널 내부에서 마지막으로 호출되는 함수 `do_fork()`로써 동일하다.** 이게 가능한 이유는 둘다 '태스크' 이기 때문이다.

`fork()`는 부모 태스크와 덜 공유하는 태스크 이고,  
`clone()`으로 생성되는 태스크는 비교적 부모 태스크와 많이 공유하는 태스크이다.

`do_fork()`는 어떤 일을 수행 할까? 태스크를 위해 일종의 이름표를 하나 준비한다. 이렇게 새로 생성된 태스크의 이름과 태어난 시간, 부모님 이름, 소지품등 이 있어야 나중에 찾기가 쉽다.  
 그 이름표는 `task_struct` 구조체 이다.

NPTL(Native POSIX Thread Library)는 태스크의 속성에 따라쓰레 드는 자원 공유의 장점을 충분히 이용하도록 구현되었기 때문이다.

```
// fork_pt.c//
```

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

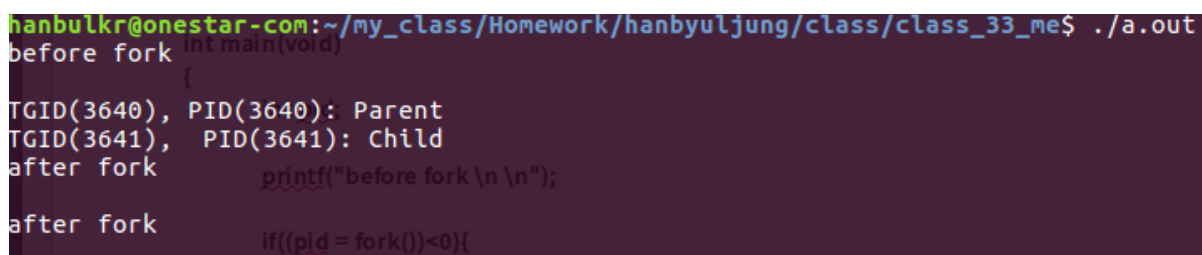
int main(void)
{
    int pid;

    printf("before fork \n \n");

    if((pid = fork())<0){
        printf("fork error \n");
        exit(-2);
    }
    else if(pid == 0)
        printf("TGID(%ld), PID(%ld): Child\n", getpid(), syscall(__NR_gettid));
    else
        printf("TGID(%ld), PID(%ld): Parent \n", getpid(), syscall(__NR_gettid));
    sleep(2);

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

    return 0;
}
```



```
hanbulkr@onestar-com:~/my_class/Homework/hanbyuljung/class/class_33_me$ ./a.out
before fork
TGID(3640), PID(3640): Parent
TGID(3641), PID(3641): Child
after fork
after fork
```

<그림 1>

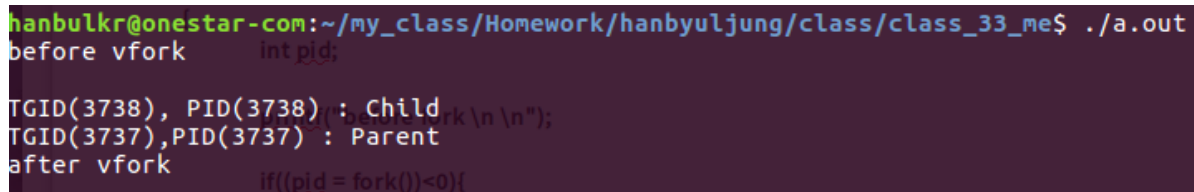
```
// vfork_pt.c//
```

```
#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(%d) : Child\n", getpid(), syscall(__NR_gettid));
        _exit(0);
    }
    else{
        printf("TGID(%d),PID(%d) : Parent \n", getpid(), syscall(__NR_gettid));
    }
    printf("after vfork \n\n");
    return 0;
}
```



```
hanbulkr@onestar-com:~/my_class/Homework/hanbyuljung/class/class_33_me$ ./a.out
before vfork
TGID(3738), PID(3738) : Child
TGID(3737),PID(3737) : Parent
after vfork
```

<그림 2>



```
// pthread_pt.c//
```

```
#include<pthread.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<linux/unistd.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_creadte \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("ptherad_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;
}
```

```
hanbulkr@onestar-com:~/my_class/Homework/hanbyuljung/class/class_33_me$ ./a.out
before pthread_creadte
TGID(3777), PID(3778), pthread_self(140169525425920) :Child
TGID(3777), PID(3779), pthread_self(140169517033216) :Child
ptherad_join(0)
pthread_join(0)
TGID(3777), PID(3777):Parent
```

<그림 3>

```
// clone_pt.c //
```

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

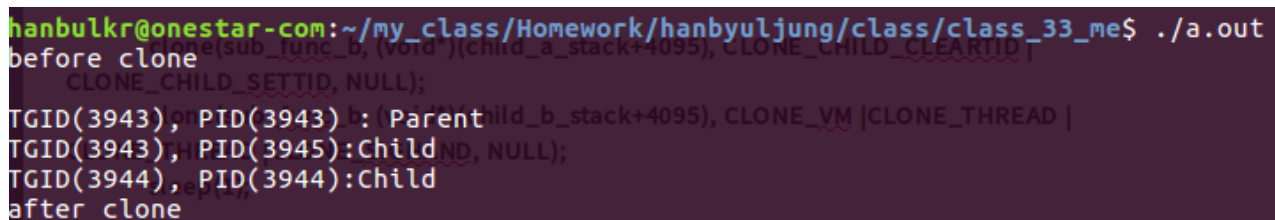
int sub_func_b(void*arg)
{
    printf("TGID(%d), PID(%d):Child \n", getpid(), syscall(__NR_gettid));
    sleep(2);
    return 0;
}

int main(void)
{
    int pid;
    int child_a_stack[4096], child_b_stack[4096];

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

    clone(sub_func_b, (void*)(child_a_stack+4095), CLONE_CHILD_CLEARTID |
    CLONE_CHILD_SETTID, NULL);
    clone(sub_func_b, (void*)(child_b_stack+4095), CLONE_VM |CLONE_THREAD |
    CLONE_THREAD |CLONE_SIGHAND, NULL);
    sleep(1);

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



```
hanbulkr@onestar-com:~/my_class/Homework/hanbyuljung/class/class_33_me$ ./a.out
before clone
TGID(3943), PID(3943) : Parent
TGID(3943), PID(3945):Child
TGID(3944), PID(3944):Child
after clone
```

<그림 4>

- <그림 1>, <그림 2>의 fork() , vfork()에서는 각 태스크의 pid 와 tpid 가 부모 태스크와 자식 태스크 간에 서로 다른 것을 알 수 있다. 즉 사용자 입장에서 다른 프로세스가 만들어 진 것이다.

- 반면 <그림 3> 의 pthread\_create()에서는 각 태스크의 pid 는 서로 다르지만 tgid 는 서로 동일함을 알 수 있다. 즉 같은 프로세스 내부에 2 개의 서로 다른 쓰레드가 생성된 것이다.

- <그림 4> 의 CLONE\_CHILD\_SETTID, CLONE\_CHILD\_CLEARID 이면 자원공유가 안되도록 Task 생성. CLONE\_THREAD 를 설정하면 태스크를 생성할 때 쓰레드로 해석되게 자원공유가 되는 형태로 생성.



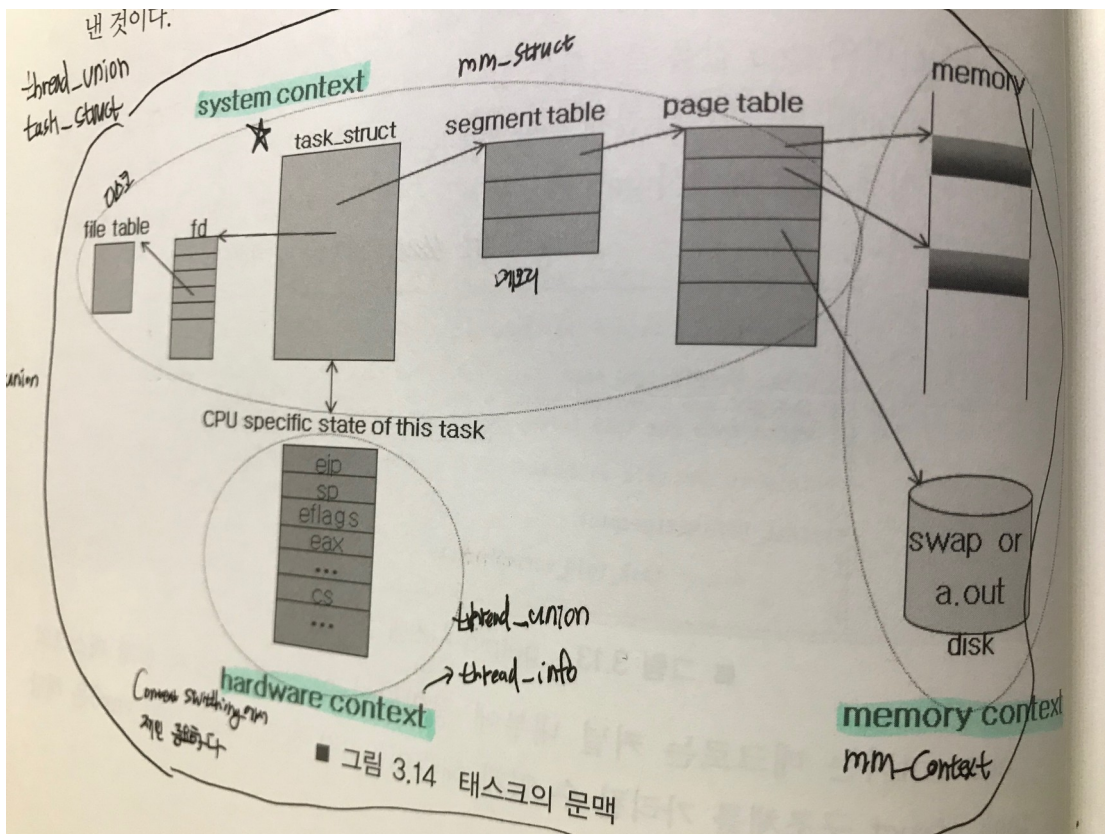
## |태스크 문맥|

프로세스 혹은 쓰레드 마다 생기는 task\_struct 라는 태스크를 관리하기 위한 자료 구조가 피룡함을 배웠다.  
리눅스 커널은 이런 태스크들마다 이러한 정보들을 관리하고 있다.

태스크가 실행 되면 생기는 fd, 스케줄링하며 필요한 우선순위, cpu 사용량 , 태스크 가족관계 , 시그널 , 사용하고 있는 자원 정보도 관리 해 주어야 한다. 이런 모든 것들을 '문맥(context)' 라고 한다.

```
thread_union{
    task_struct{
        system context
        hardware context
        memory context
    }
}
```

< 이런 느낌으로 문맥들이 포함 되어 있다. >



- 시스템 문맥이 할당된 자료구조 : task\_struct, 파일 디스크립터, 파일 테이블, 세그먼트 테이블, 페이지 테이블
- 메모리 문맥: 텍스트, 데이터 , 스택 ,heap 영역, 스왑 공간 등이 여기 포함 된다.
- 하드웨어 문맥: context switch 할때 태스크의 현재 실행 위치 정보 유지, 쓰레드구조 또는 하드웨어레지스터 문맥이라고 불린다.

## |task\_identification|

\*uid(사용자 ID), euid(유효 사용자 ID, sudo)

\*system call 은 유일한 소프트웨어 인터럽트이다.

## |state|

- task 는 생성에서 소멸까지 많은 과정을 거친다. 이를 관리하기 위한 state 변수가 존재한다.
- 이변수에는 TASK\_RUNNING(0),
  - 러닝중
- TASK\_INTERRUPTIBLE(1),
  - 인터럽트 수신가능
- TASK\_UNINTERRUPTIBLE(2),
  - 인터럽트 허용 X
- TASK\_STOPPED(4),
  - 정지, ctrl+z → 너무 많이 쓰면 컴퓨터에 무리가 간다.
- TASK\_TRACED(8),
  - 디버깅
- EXIT\_DEAD(16),
  - return 0, exit(0), 시그널에 맞아 죽는 것
- EXIT\_ZOMBIE(32)
  - 자식이 죽고 부모가 처리 안했을 때

## |task\_relationship|

태스크는 생성되면서 가족관계를 맺는다. 리눅스 커널에 존재하는 모든 태스크들은 이중 연결 리스트로 연결되어 있는데 이 연결 리스트의 시작은 init\_task 로 부터 시작되며, tasks 라는 이리스트 헤드를 통해 연결된다.

## |scheduling information|

se→ 유저에 한해서

rt→ 시스템에 한해서

## |signal information|

시그널은 태스크에게 비동기적 신호를 알린다. signal, sighand, blocked, pending 등이 있다.

## | memory information|

- (mm\_struct , red\_black, pgd, vm\_struct, 가상 메모리.)
- 태스크는 자신의 명령어와 데이터를 텍스트, 데이터, 스택, 힙 등에 저장한다.
- 가상 주소를 물리주소로 변환하기 위해 페이지 디렉토리와 페이지 테이블 등의 변환 정보 task\_struct 에 존재.

## | file information |

- file\_struct 구조체 형태인 files 라는 이름의 변수로 접근 가능.
- inode(연결리스트)(스왑), fs\_struct 구조체 형태인 fs 라는 변수로 접근 가능.
- inode : 디스크에서 파일위치 알려줌.

## | thread structure |

쓰레드 구조는 문맥 교환을 수행할 때 태스크가 현재 어디까지 실행되었는지 기억해놓는 공간이다.

## State transition

- 태스크는 당장 제공해 줄 수 있는 자원을 요청한다면 커널은 이 태스크를 잠시 대기 하도록 만든 뒤 다른 태스크를 먼저 수행시키며, 태스크가 요청했던 자원이 사용 가능해지면 다시 수행 시켜 줌으로 써 보다 높은 시스템 활용률을 제공하려 한다. 따라서 태스크는 상태 전이 라는 특징을 가지게 된다.

- cpu 는 오직 한 순간에 한가지 동작을 한다.

- 부모 태스크가 자식 태스크에게 wait() 등의 함수를 호출하기 전에 먼저 종료되어 없어지면 어떻게 될까?

----고아 프로세스---- 가 된다. 이런 문제로 계속 태스크가 남아있을 수 있다. 하지만 이런 고아태스크의 부모를 init 태스크(신이라 불리는 1 번 태스크이다.)로 바꾸어 주며 init 태스크가 wait() 등의 함수를 호출할 때 고아 태스크는 최종 소멸된다.

- 시그널을 받은 태스크는 TASK\_STOPPED 상태로 전이되며, 디버거의 ptrace() 호출에 의해 디버깅 되고 있는 태스크는 시그널을 다는 경우 TASK\_TRACED 상태로 전이 될 수 있다.

- c 와 어셈블리로 작성된 소프트웨어 수행을 위해 스택을 필요로 한다. Intel (16KB), arm (8KB).

- 커널스택은 thread\_union 이라 불리고 thread\_info 구조체를 포함한다.

- thread\_info 구조체를 (프로세스 디스크립터(fd) → task 마다 존재) 라 부른다.

\*pt\_regs 라는 이름 → cpu\_context\_save 으로 바꿔 보아야 한다.

1. 커널은 현재 실행 중인 태스크가 시그널을 받았는지 확인, 받았으면 필요에 따라 시그널 처리 핸들러 호출

2. 다시 스케줄이 필요하면 (need\_resched → flags 의 플래그가 1 로 set 된 경우) 스케줄러 호출

3. 커널 내에서 연기된 루틴들( 1. nonblocking, 2. 하드웨어 인터럽트 → Bottom Half(nonblock 과 비슷, 우선순위에 맞게 해결한다.)

## 런큐와 스케줄링

- 140 단계의 우선순위 중 실시간 태스크는 0~99 단계 까지 사용  
동적우선 순위 태스크는 100 ~ 139 까지 사용.  
따라서 **실시간 태스크는 항상 일반 태스크 보다 우선**하여 실행됨을 의미 한다.

## 런큐와 태스크

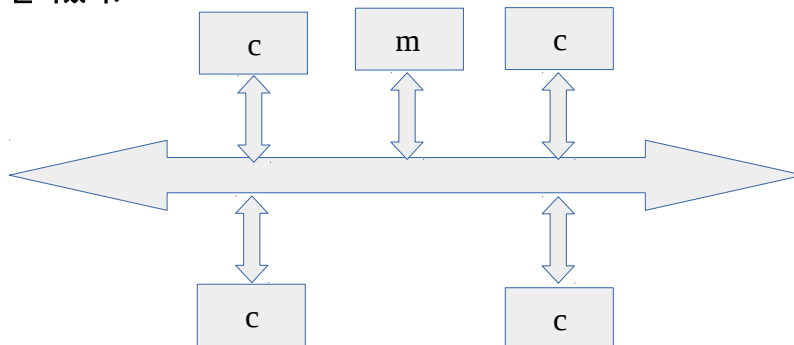
- 운영체제가 스케줄링 작업 수행을 위해의 **수행 가능한 상태 태스크**를 **run queue** 라고 한다.

```
struct sched_entity se;  
struct sched_rt_entity rt; → 리얼타임  
(on_rq 런큐의 여부, rt_rq 리얼타임)
```

하이퍼 쓰레딩(hyper\_threading) : cpu 가 4 개인데 8 개로 보임 이유는 **fork()**를 **회로로 구현**해서 2 배가 됨.

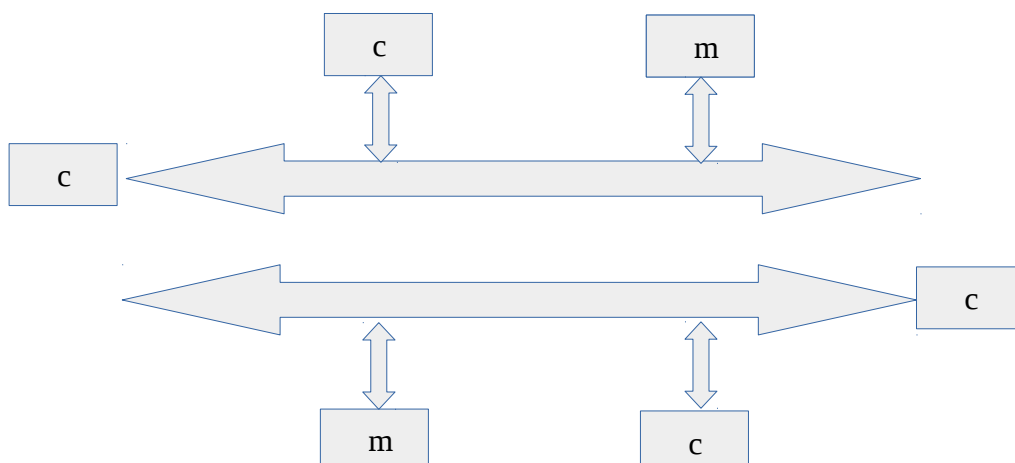
## UMA (Uniform Memory Access)

- 단일 버스에 물려있다.



## NUMA (Non- Uniform Memory Access)

- cpu 부하 뿐만아니라 메모리 접근 시간의 차이 등도 고려 하여 부하 균등을 시도한다.



## NPTL 에 대해

- NPTL(Native POSIX Thread Library)은 LinuxThreads 의 단점을 극복하기 위한 새로운 구현으로 POSIX 요구사항 또한 충족한다. NPTL 은 성능과 확장성 측면에서 LinuxThreads 보다 강력한 개선 사항을 제공한다. LinuxThreads 와 같이 NPTL 은 1 대 1 모델을 구현한다.

- 스레드 구현은 대규모 프로세서를 탑재한 시스템에서도 잘 동작해야 한다.
- 심지어 작은 작업을 위해 새로운 스레드를 생성하더라도 시작 비용이 낮아야 한다.
- 새로운 스레드 라이브러리는 NUMA 지원을 활용할 수 있어야 한다.

### NPTL 은 LinuxThreads 에 비해 여러 가지 장점이 있다.

NPTL 은 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.

- NPTL 스레드 라이브러리는 시그널을 사용한 스레드 동기화 기법을 피한다. 이런 목적으로 NPTL 은 퓨텍스(futex)라는 새로운 메커니즘을 도입했다. 퓨텍스는 공유 메모리 영역에서 동작하므로 프로세스 사이에 공유가 가능하므로 프로세스 간 POSIX 동기화를 제공한다.

- NPTL 은 프로세스 단위로 시그널을 처리한다.

-NPTL 스레드 라이브러리에 도입된 중요한 특징 중 하나는 **ABI(Application Binary Interface) 지원**이다. 이는 LinuxThreads 와 **하위 호환이 가능**하도록 돕는다. 다음에 다룰 **LD\_ASSUME\_KERNEL**의 도움을 받아 ABI 지원을 처리한다.

<http://www.test104.com/kr/tech/3337.html> 참조.