TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
33 일차 (2018. 04. 09)

목차

1. Chapter 3 : 태스크 관리

3) 프로세스와 쓰레드의 생성과 수행

프로세스는 fork()와 vfork()로 생성된다. 사용법은 같은데 이 두 가지의 차이점은 무엇일까? 이는 exec()를 사용하던 단점에서 나오게 되었다. 이 함수를 실행하게 되면 기존 메모리가 없어지게 되어 exec()이후의 명령들을 실행하지 못한다는 점이다. 그래서 fork()와 exec()을 같이 사용하여 기존의 메모리를 유지했다. 이 때의 문제점은 fork()하여 부모의 메모리를 복사하고 자식 프로세스에 exec()하여 메모리를 덮어쓰게 되어 쓸데없이 메모리를 낭비하게 된다는 것이다. 이를 보완하기 위해 vfork()를 만든 것이다. fork()하고 exec()하여 가상 메모리 레이아웃을 제외한 정보들, 프로세스 구성 정보만 복사하고 싶을 때 vfork()를 사용한다.

그리고 최근 리눅스는 C.O.W(Copy On Write)기법을 도입해서 fork()할 때, 야기되는 주소 공간 복사 비용을 많이 줄였다.

fork()가 메모리를 공유하지 않아 자식 프로세스 주소 공간에만 영향을 주고 부모 프로세스 주소 공간의 변수에는 영향을 주지않는다면, 메모리를 공유하는 쓰레드 생성은 어떤 차이가 있을까?

```
#define _GNU_SOURCE   //예전에는 상관없었지만 현재는 clone()를 쓰기 위해선 넣어줘야 한다.
#include<stdio.h>
#include<unistd.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];   // 물리 메모리 최소 단위가 4096 이라 한 번에 할당한다.
                                         조금씩 할당 받아 사용하는 것보다 성능이 좋다.
                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);
      /*     첫번째 인자 : sub_func 라는 함수를 구동시킬 것이다.
```

```
        두번째 인자 : thread 도 sub_func 을 구동시켜서 스택이 필요하기에 잡아준 것.
        세번째 인자 : 옵션
                        CLONE_VM        : 가상 메모리 사용.
                        CLONE_THREAD  : 쓰레드 생성. 이 때, CLONE_CHILD_CLEARID 와
                                        CLONE_CHILD_SETTID 를 설정하면 프로세스도 가능
                        CLONE_SIGHAND : 시그널 처리.
        네번째 인자 : NULL 도 옵션인데 지금은 그냥 옵션인 것만 알고 넘어가자 */
         sleep(1);
         printf("PID(%d) : Parent g = %d, l=%d₩n", getpid(), g, l);
         return 0;                                                    }
```

결과는 부모의 pid 값은 동일하게 나오며, clone()실행 후 처음과 달리 g 가 2 에서 3 으로 변한다. fork()의 경우 2 인데, thread 는 3 이 되는 것은 자식프로세스와 메모리를 공유하여 영향을 받기 때문이다. 기존에 수행되던 쓰레드는 자신이 생성한 쓰레드가 변수를 수정하면 그 수정된 결과를 그대로 볼 수 있는 것이다. 즉, **쓰레드를 생성하면 자식 쓰레드와 부모 쓰레드는 서로 같은 주소공간을 공유한다**는 걸 뜻하는 예제이다. 쓰레드가 아닌 프로세스였다면 프로세스는 pid 값이 자식 프로세스에서는 달랐을 것 이고, 메모리를 공유하지 못하니 값이 갱신되지 못했을 것이다. 이렇듯, 메모리를 공유하다 보면 연산 중에 context switching 으로 다른 함수에 영향을 받을 수 있다. 그렇다면 그 값은 정상적인 도출이 힘들 것이고 결국 원하는 값이 나오지 않을 것이다. 이러한 영역을 critical section 라 하고, 이를 방지하기 위해 mutex, semaphore, spin lock 같은 것을 사용하는 것이다.

4) 리눅스의 태스크 모델

리눅스에서는 프로세스와 쓰레드를 관리하기 위해 각 프로세스와 쓰레드마다 task_struct 라는 동일한 자료구조를 생성하여 관리한다. 즉, task_struct 가 있으면 프로세스와 관련된 정보를 알 수 있다. 프로세스가 수행되려면 자원(resource)과 수행 흐름(flow of control)이 필요한데, 기존 운영체제 연구자들은 자원을 태스크로 제어 흐름을 쓰레드로 정의하였다.

리눅스에서 fork(), vfork(), clone(), pthread_create()등이 구현되는 방법을 보자면, 이 함수들이 커널 영역으로 들어가면, 커널 내부 함수인 do_fork()를 호출한다. do_fork()의 내부를 보면 다음과 같다.

```
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;
                     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;
}
```

do_fork()가 수행하는 일은 새로 생성되는 태스크를 위해 일종의 이름표를 하나 준비한다. 여기에 부모 프로세스 이름(ppid), 메모리 레이아웃 등 자세한 정보를 기록해둔다. 나중에 새로 생성된 태스크를 쉽게 찾아내고 그 태스크의 정보를 알 수 있게 하기 위해서이다. 이를 통해, tgid 와 pid 가 같으면 프로세스 tgid 는 같고 pid 는 다르면 같은 프로세스의 쓰레드(구성 쓰레드)들, tgid 와 pid 모두 다르면 다른 프로세스라는 것을 알 수 있다.

task_struct 의 pid 값을 출력해주는 gettid() 구현을 살펴 보면 current 가 나온다. current 라는 매크로는 커널 내부에 정의되어 있는 매크로로써 현재 태스크의 task_struct 구조체를 가리킬 수 있게 해준다. task_tigd_vnr()은 해당 task_struct 구조체의 tgid 필드를 리턴한다.