Xilinx Zynq FPGA, TI DSP MCU 기반의

프로그래밍 및 회로 설계 전문가

강사 이상훈

(Innova Lee)

Gcccompil3r@gmail.com

학생 김민호

minking12@naver.com

http://www.test104.com/kr/tech/3337.html

리눅스 스레드 모델 비교: LinuxThreads 와 NPTL

LinuxThreads 프로젝트는 원래 리눅스(Linux®)에서 멀티스레딩 기능을 제공했지만, POSIX 스레딩 표준을 준수하지는 않았습니다.
좀 더 최근에 NPTL(Native POSIX Thread Library)이 이런 간극을 좀 더 메워 나가고는 있지만, 다른 문제점이 여전히 남아 있습니다.
이 기사에서는 두 가지 리눅스 모델에 대한 몇 가지 차이점을 기술해 응용 프로그램을 LinuxThreads 에서 NPTL 로 이식하기를 원하거나 단순히 차이점이 무엇인지 살펴보기를 원하는 개발자를 지원합니다.

리눅스가 처음 개발되었을 때 커널에서 진짜 스레드를 지원하지 않았다.
하지만 리눅스는 clone() 시스템 호출을 통해 스케쥴 가능한 엔티티로 프로세스를 지원했다.
이 호출은 호출하는 프로세스의 복사본을 생성해 호출자의 주소 영역에서 복사 공유가 가능하게 만들었다.
LinuxThreads 프로젝트는 이런 시스템 호출을 사용해 사용자 영역에서 스레드 지원을 완벽하게 흉내내었다.
불행하게도, 이런 접근 방식에는 특히 신호 처리, 스케줄링, 프로세스 간 동기화 부문에서 몇 가지 단점이 있었다.
또한 스레드 모델이 POSIX 요구사항을 준수하지 못했다.
LinuxThreads 를 개선하기 위해, 몇 가지 커널 지원과 새로 작성한 스레드 라이브러리가 필요함이 명백해졌다.
이런 요구 사항을 충족하기 위해 두 가지 경쟁 관계에 있는 프로젝트가 출발했다.
IBM 에서 나온 팀은 NGPT(Next-Generation POSIX Threads)를 만들었다.
반면 레드햇에서 나온 팀은 NPTL 을 만들고 있었다. NGPT 는 2003 년 중반에 좌초되었으며 NPTL 만 살아남았다.
LinuxThreads 를 대신한 NPTL 선택은 필연적인 결론처럼 보이지만, 늙어가는 리눅스 배포판을 위한 응용을 유지보수하고 있으며 조만간 판올림을 계획하고 있다면, NPTL 이주는 프로세스 이식 과정에서 아주 중요한 주제가 될 것이다.
이주의 대안으로, 이런 차이점을 파악해 예전 기술과 새 기술을 동시에 수용하도록 응용 프로그램을 설계할 수도 있다.
이 기사는 어떤 배포판에서 어떤 스레드 모델이 구현되었는지 세부 사항을 다룬다.

LinuxThreads 설계 명세

스레드는 프로그램을 쪼개어 동시에 동작하는 태스크를 하나 이상으로 나눈다.
단일 프로세스의 상태 정보를 공유하며, 다른 자원과 메모리를 직접 공유한다는 점에서 스레드는 전통적인 멀티태스킹에서 말하는 프로세스와 다르다.
동일 프로세스에 존재하는 스레드 사이에서 일어나는 문맥 전환은 일반적으로 프로세스 사이에서 일어나는 문맥 전환보다 훨씬 더 빠르다.

따라서 멀티스레드 프로그램은 멀티프로세스 응용보다 더 빠르다는 장점이 있다.
또한 스레드를 사용하면 동시에 처리하도록 구현할 수 있다.
이런 상대적인 장점이 프로세스 기반 접근 방식을 벗어나 LinuxThreads 구현을 이끌었다.
LinuxThreads 초기 설계는 관련된 프로세스 사이에서 일어나는 문맥 전환이 충분히 빨라서 개별 커널 스레드가 대응하는 사용자 수준 스레드를 처리할 수 있다는 믿음으로 출발했다.
이는 1 대 1 스레드 모델이라는 혁신을 이끌었다.
LinuxThreads 설계 명세에 대한 핵심을 검토해 보자.
LinuxThreads 의 가장 핵심적인 특징 중 하나는 관리자 스레드다. 관리자 스레드는 다음 요구 사항을 만족한다.
시스템은 중요한 시그널에 반응하고 전체 프로세스에 kill 로 시그널을 보낼 수 있어야 한다.
스택으로 사용된 메모리 할당 해제는 스레드 종료에 앞서 일어나야 한다. 따라서 스레드는 이런 작업을 스스로 할 수 없다.
반드시 스레드 종료를 기다려 좀비로 변하지 않도록 해야 한다.
스레드 지역 자료 할당 해제는 모든 스레드에 반복적으로 수행할 필요가 있다.
관리자 스레드가 이런 작업을 수행해야 한다.
메인 스레드가 pthread_exit()를 호출할 필요가 있을 때, 프로세스는 종료되지 않는다.
메인 스레드는 잠들기로 빠지며, 모든 다른 스레드를 종료하고 나서 관리자 스레드가 메인 스레드를 깨운다.
스레드 지역 자료와 메모리 유지를 위해, LinuxThreads 는 스택 주소의 바로 아래 있는 프로세스 주소 공간의 상위 메모리를 사용한다.
동기화 요소는 시그널을 사용해서 달성한다. 예를 들어, 스레드 차단은 시그널로 깨어난다.
clone 시스템의 초기 설계하에서, LinuxThreads 는 각 스레드를 독자적인 프로세스 ID 를 부여한 다른 프로세스로 구현했다.
치명적인 시그널은 모든 스레드를 죽일 수 있다.
이런 점에서 LinuxThreads 설계에는 일관성이 있어왔다. 일단 프로세스가 치명적인 시그널을 받으면, 스레드 관리자는 다른 모든 스레드(프로세스)에 kill 로 똑같은 시그널을 보낸다.
LinuxThreads 설계에 따르면,
비동기식 시그널을 보내면 관리자 스레드는 이 시그널을 스레드 중 하나로 배달할 것이다.
대상 스레드가 현재 시그널을 차단하고 있다면 시그널은 대기 상태를 유지한다. 이렇게 하는 이유는 관리자 스레드가 프로세스에 시그널을 보낼 수 없기 때문이다.
대신 각 스레드는 프로세스처럼 행동한다.
스레드 사이에 일어나는 스케줄링은 커널 스케줄러가 처리한다.
LinuxThreads 와 제약점
일반적으로 LinuxThreads 설계는 멋지게 동작한다.
하지만 과도한 응용이 부하를 주면 성능, 확장성, 사용성에 문제를 일으킨다.
LinuxThreads 설계에서 몇 가지 제약을 살펴보자.
LinuxThreads 는 각 프로세스가 소유한 모든 스레드 사이에서 생성과 조율을 위해 관리자 스레드를 사용한다.

이는 스레드 생성과 파괴에 부하를 높인다.
관리자 스레드 방식 설계 때문에 문맥 전환이 상당히 많이 일어나며, 확장성과 성능에 잠재적인 장애를 일으킨다.
관리자 스레드는 단지 CPU 하나에서만 동작하므로, 동기화 수행은 SMP 나 NUMA 시스템에서 확장성 문제를 초래할 수 있다.
스레드 관리 방식과 각 스레드마다 다른 프로세스 ID 할당으로 인해, LinuxThreads 는 다른 POSIX 관련 스레드 라이브러리와 호환되지 않는다.
시그널은 동기화 요소를 구현하는 데 사용되었다.
이는 연산의 반응 속력에 영향을 미쳤다. 또한 메인 프로세스로 시그널 전송 개념이 존재하지 않는다.
따라서 시그널 처리 과정에서 POSIX 방식을 따르지 않는다.
LinuxThreads 내부에서 시그널 처리는 각 스레드가 독자적인 프로세스 ID 를 할당 받았기에 프로세스 기반이 아니라 스레드 기반으로 수행한다.
시그널은 지정 스레드로 보내지기 때문에 시그널이 직렬화된다. 다시 말해 시그널은 이 스레드를 통해 다른 스레드로 전달된다.
이는 시그널 병렬 처리를 위한 POSIX 표준 요구 사항과 대조적이다.
예를 들어, LinuxThreads 하에서, kill()로 보낸 시그널은 전체 프로세스가 아니라 개별 스레드에 전송될 것이다.
이는 스레드가 시그널을 차단하고 있다면 LinuxThreads 는 시그널을 차단하지 않는 다른 스레드가 핸들러를 즉시 수행하는 대신 단순히 이 스레드에 대한 큐에 시그널을 밀어 넣어서 스레드가 시그널 차단을 해제할 때 핸들러를 실행한다.
LinuxThreads 에서 개별 스레드는 프로세스이므로, 사용자와 그룹 ID 정보가 단일 프로세스에서 모든 스레드에 공통으로 작용하지 않을 것이다.
따라서 멀티스레드 setuid()/setgid() 결과는 스레드마다 달라질 수 있다.
생성된 멀티스레드 코어 덤프는 모든 스레드 정보를 포함하고 있지 않다.
다시 말해, 이런 행동 방식은 각 스레드가 프로세스이기 때문에 일어나는 결과다.
비정상 종료가 스레드 중 하나에서 발생하면, 시스템 코어 파일에서 해당 스레드만 보인다. 하지만 이런 행동 양식은 LinuxThreads 구현 중에서 주로 예전 버전에 해당된다.
각 스레드는 독자적인 프로세스이므로, /proc 디렉터리는 이상적으로 스레드가 되어야 할 수 많은 프로세스 항목으로 가득 차버린다.
각 스레드는 프로세스이므로 응용을 위해 생성 가능한 스레드 숫자에 제한이 있다.
예를 들어 IA32 시스템에서 생성 가능한 스레드 숫자이자 생성 가능한 프로세스 숫자는 4090 개다.
스레드 지역 자료 계산 방법이 스택 주소 위치에 기반을 두므로 자료 접근이 아주 느리다.
또 다른 단점으로 사용자는 스택 크기를 명확히 지시하지 못한다. 사용자는 스택 영역을 다른 목적으로 사용되는 영역에 우연히 사상할 수도 있기 때문이다.
요청에 따라 자라나는 개념(부동 스택 개념이라고도 부른다)은 리눅스 커널 2.4.10 이후에 구현되었다. 이 버전 이전에는 LinuxThreads 가 정적 스택을 사용했다.
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.
Ulrich Drepper 와 Ingo Molnar 는 NPTL 설계에 참여한 레드햇 직원이다.

전반적인 설계 목표 중 몇 가지는 다음과 같다.
새로운 스레드 라이브러리는 POSIX 를 준수해야 한다.
스레드 구현은 대규모 프로세서를 탑재한 시스템에서도 잘 동작해야 한다.
심지어 작은 작업을 위해 새로운 스레드를 생성하더라도 시작 비용이 낮아야 한다.
NPTL 스레드 라이브러리는 LinuxThreads 와 이진 호환이 가능해야 한다.
이런 목적으로 LD_ASSUME_KERNEL 을 사용할 수 있다.
이 기사 뒷부분에 다루겠다.
새로운 스레드 라이브러리는 NUMA 지원을 활용할 수 있어야 한다.

NPTL 의 장점
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.
관리자 스레드를 사용하지 않기 때문에, NPTL 스레드 모델은 NUMA 와 SMP 시스템에서 좀 더 나은 확장성과 동기화 메커니즘을 제공한다.
새로운 커널 구현과 함께 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.
또한 프로세스 사이에서 퓨텍스를 공유할 수도 있다.
이런 행동 양식은 프로세스 간 동기화를 현실로 만든다.
실제로 NPTL 은 PTHREAD_PROCESS_SHARED 라는 매크로를 포함해 개발자에게 다른 프로세스의 스레드 사이에 뮤텍스를 사용자 수준 프로세스에서 공유하기 위한 핸들을 제공한다.
NPTL 이 POSIX 규약을 따르므로 NPTL 은 프로세스 단위로 시그널을 처리한다.
getpid()는 모든 스레드에서 똑같은 프로세스 ID 를 반환한다.
예를 들어 시그널 SIGSTOP 을 보내면 전체 프로세스가 멈춘다.
LinuxThreads 에서는 이 시그널을 받은 스레드만 멈춘다. 이는 NPTL 기반 응용에서 GDB 와 같은 디버그 지원을 강화한다.
NPTL 에서 모든 스레드에는 어버이 프로세스 하나만 존재하므로, 어버이에게 보고되는 자원 사용(CPU 나 메모리 퍼센트와 같은)은 스레드 하나가 아니라 전체 프로세스에 보고된다.
NPTL 스레드 라이브러리에 도입된 중요한 특징 중 하나는 ABI(Application Binary Interface) 지원이다.
이는 LinuxThreads 와 하위 호환이 가능하도록 돕는다. 다음에 다룰 LD_ASSUME_KERNEL 의 도움을 받아 ABI 지원을 처리한다.

LD_ASSUME_KERNEL 환경 변수
앞서 설명한 바에 따르면,
ABI 도입은 코드에서 NPTL 과 LinuxThreads 모델을 둘 다 지원할 수 있게 만든다.
기본적으로 어떤 런타임 스레드 라이브러리를 동적으로 링크할지 결정하는 작업은 ld(동적 링커/로드) 몫이다.
예를 들어, WebSphere® 응용 서버가 사용하는 몇 가지 공통 변수 설정은 다음과 같다.
요구사항에 맞춰 적절히 시도해보자.
LD_ASSUME_KERNEL=2.4.19: 이는 NPTL 구현을 덮어쓴다.
이 구현은 일반적으로 부동 스택 기능을 활성화한 표준 LinuxThreads 모델을 의미한다.
LD_ASSUME_KERNEL=2.2.5: 이는 NPTL 구현을 덮어쓴다.
이 구현은 일반적으로 고정 스택 기능을 활성화한 LinuxThreads 모델을 의미한다.
다음과 같은 명령으로 이 환경 변수를 설정해보자.

```
export LD_ASSUME_KERNEL=2.4.19
```

LD_ASSUME_KERNEL 설정을 위한 지원은 스레드 라이브러리를 위해 현재 지원하는 ABI 버전에 의존한다.
예를 들어, 스레드 라이브러리가 ABI 버전 2.2.5 를 지원하지 않으면, 사용자는 LD_ASSUME_KERNEL 을 2.2.5 로 설정할 수 없다. 일반적으로 NPTL 은 2.4.20 을 요구하며, LinuxThreads 는 2.4.1 을 요구한다.
이 모든 설정은 일반적으로 NPTL 활성 리눅스 배포판을 돌리지만 응용이 LinuxThreads 모델을 기초로 설계되어 있을 경우에 사용한다.
GNU_LIBPTHREAD_VERSION 매크로
대다수 현대적인 리눅스 배포판은 LinuxThreads 와 NPTL 양쪽을 모두 포함하며, 양쪽 사이에 전환이 가능한 방법을 제공한다. 현재 시스템에서 사용 중인 스레드 라이브러리 버전을 확인하려면 다음 명령을 내린다.

```
$ getconf GNU_LIBPTHREAD_VERSION
```

출력은 다음과 같다.

```
NPTL 0.34
```

또는 다음과 같다.

```
Linuxthreads-0.10
```

리눅스 배포판에 따른 스레드 모델, glibc 버전, 커널 버전

표 1. 리눅스 배포판과 스레드 구현

스레드 구현	C 라이브러리	배포판	커널
LinuxThreads 0.7, 0.71 (for libc5)	libc 5.x	Red Hat 4.2	
LinuxThreads 0.7, 0.71 (for glibc 2)	glibc 2.0.x	Red Hat 5.x	
LinuxThreads 0.8	glibc 2.1.1	Red Hat 6.0	
LinuxThreads 0.8	glibc 2.1.2	Red Hat 6.1 and 6.2	
LinuxThreads 0.9		Red Hat 7.2	2.4.7
LinuxThreads 0.9	glibc 2.2.4	Red Hat 2.1 AS	2.4.9
LinuxThreads 0.10	glibc 2.2.93	Red Hat 8.0	2.4.18
NPTL 0.6	glibc 2.3	Red Hat 9.0	2.4.20
NPTL 0.61	glibc 2.3.2	Red Hat 3.0 EL	2.4.21
NPTL 2.3.4	glibc 2.3.4	Red Hat 4.0	2.6.9
LinuxThreads 0.9	glibc 2.2	SUSE Linux Enterprise Server 7.1	2.4.18
LinuxThreads 0.9	glibc 2.2.5	SUSE Linux Enterprise Server 8	2.4.21
LinuxThreads 0.9	glibc 2.2.5	United Linux	2.4.21
NPTL 2.3.5	glibc 2.3.3	SUSE Linux Enterprise Server 9	2.6.5

커널 2.6.x 과 glibc 2.3.3 이후에는 NPTL 버전 번호 관례가 바뀐듯이 보인다.
이제 스레드 라이브러리는 사용 중인 glibc 버전과 일치한다.
JVM(Java™ Virtual Machine) 지원은 버전에 따라 달라질 수 있다.
IBM 이 이식한 JVM 은 glibc 2.1 이상을 탑재한 표 1 에 나온 대다수 배포판을 지원한다.

결론

LinuxThreads 의 한계는 나중에 나온 LinuxThreads 버전은 물론이고 NPTL 로 극복해왔다.
예를 들어, 최신 LinuxThreads 구현은 스레드 지역 자료 위치를 지정하기 위해 스레드 레지스터를 사용한다.
예를 들어, 인텔(Intel®) 프로세서에서 LinuxThreads 는 %fs 와 %gs 세그먼트 레지스터를 사용해 스레드 지역 자료에 접근하는 가상 주소를 지정한다.
LinuxThreads 에서 변경 결과 개선이 있었지만, 좀 더 높은 부하와 스트레스 테스트에서 여전히 문제점이 고개를 내민다.
관리자 스레드에 대한 과도한 의존, 시그널 처리와 같은 요인 때문이다.
LinuxThreads 로 라이브러리를 만드는 동안에 -D_REENTRANT 컴파일 플래그를 잊어버리지 말자.
이는 라이브러리를 스레드 안전한 상태로 만든다.
마지막으로 아마도 가장 중요한 힌트로, LinuxThreads 는 프로젝트 주창자가 NPTL 을 대안으로 생각하기에 더 이상 활발하게 갱신하지 않는다는 사실을 기억하자.
LinuxThreads 의 단점으로 NPTL 이 오류가 없다는 식으로 받아들여서는 곤란하다.
SMP 위주 설계로 만들어진 NPTL 역시 단점이 있다.
최근 레드햇 커널에서 간단한 스레드 응용 프로그램이 단일 프로세스 기계에서는 정상으로 동작했지만 SMP 에서는 얼어버리는 상황을 목격했다.
하이엔드 응용을 만족시키기 위해 확장성을 높이려면 리눅스에서 해야 할 일이 여전히 많다는 생각이다.

```c
#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()
{
    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);
    sleep(1);
    printf("PID(%d):parent g=%d,l=%d\n",getpid(),g,l);
    return 0;
}
```

```
minking@minking-Z20NH-AS51B1U:~$ ./a.out
PID(10923):parent g=2, l=3
PID(10923):Child g=3
PID(10923):parent g=3,l=3
minking@minking-Z20NH-AS51B1U:~$
```

```
31
32 #include<stdio.h>
33 #define test(name) sys_##name()
34
35 void sys_fork()
36 {
37     printf("This is sys_fork function by macro test(name)\n");
38 }
39
40 int main()
41 {
42     test(fork);
43     sys_fork();
44     return 0;
45 }
~
~
~
```

```
minking@minking-Z20NH-AS51B1U:~$ vi 0409.c
minking@minking-Z20NH-AS51B1U:~$ gcc 0409.c
minking@minking-Z20NH-AS51B1U:~$ ./a.out
This is sys_fork function by macro test(name)
minking@minking-Z20NH-AS51B1U:~$ vi 0409.c
minking@minking-Z20NH-AS51B1U:~$ vi 0409.c
minking@minking-Z20NH-AS51B1U:~$ gcc 0409.c
minking@minking-Z20NH-AS51B1U:~$ ./a.out
This is sys_fork function by macro test(name)
This is sys_fork function by macro test(name)
minking@minking-Z20NH-AS51B1U:~$ vi 0409.c
minking@minking-Z20NH-AS51B1U:~$
```

fork()와 vfork()의 차이점

fork()
UNIX, LINUX 에서는 새로운 Process 를 생성하기 위해 fock()함수를 호출한다.
fork()함수가 호출되면 부모의 Process Memory, File Descriptor Table 을 복사하여 자신과 동일한 Process 를 생성한다.

vfork()
기본적으로 fork()함수와 하는 일을 동일하다.
fork()는 fork() 함수 호출이후 exec()함수 호출 까지의 parent process 와 child process 의 수행이 non-blocking 형식인데 반해, vfork()는 Blocking 이 된다. (child process 먼저 수행) fork()는 exec()함수 호출에 의해 새로운 바이너리로 교체하여 parent process 의 복사본을 만드는데, 이것은 불필요한 오버헤드가 된다.

COW(Copy On Write)
vfork()처럼 parent process 의 데이터 부분에 대한 참조만 소유하고 있다가 변경이 되는 시점에 복사하여 사용하는 방식을 이야기하며 현재의 Linux 에서는 COW 기법을 도입한 fork()를 제공한다.

출처:http://atsequence.tistory.com/21[A.T.S Mucha]

```
1764 #ifndef CONFIG_HAVE_COPY_THREAD_TLS
1765 /* For compatibility with architectures that call do_fork directly rather than
1766  * using the syscall entry points below. */
1767 long do_fork(unsigned long clone_flags,
1768               unsigned long stack_start,
1769               unsigned long stack_size,
1770               int __user *parent_tidptr,
1771               int __user *child_tidptr)
1772 {
1773         return _do_fork(clone_flags, stack_start, stack_size,
1774                         parent_tidptr, child_tidptr, 0);
1775 }
1776 #endif
1777
1778 /*
1779  * Create a kernel thread.
1780  */
1781 pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
1782 {
1783         return _do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn,
1784                 (unsigned long)arg, NULL, NULL, 0);
1785 }
1786
1787 #ifdef __ARCH_WANT_SYS_FORK
1788 SYSCALL_DEFINE0(fork)
1789 {
1790 #ifdef CONFIG_MMU
1791         return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
1792 #else
1793         /* can not support in nommu mode */
1794         return -EINVAL;
1795 #endif
1796 }
1797 #endif
1798
1799 #ifdef __ARCH_WANT_SYS_VFORK
1800 SYSCALL_DEFINE0(vfork)
1801 {
1802         return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
1803                         0, NULL, NULL, 0);
1804 }
1805 #endif
1806
1807 #ifdef __ARCH_WANT_SYS_CLONE
1808 #ifdef CONFIG_CLONE_BACKWARDS
1809 SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
1810                  int __user *, parent_tidptr,
1811                  unsigned long, tls,
1812                  int __user *, child_tidptr)
1813 #elif defined(CONFIG_CLONE_BACKWARDS2)
1814 SYSCALL_DEFINE5(clone, unsigned long, newsp, unsigned long, clone_flags,
1815                  int __user *, parent_tidptr,
1816                  int __user *, child_tidptr,
1817                  unsigned long, tls)
1818 #elif defined(CONFIG_CLONE_BACKWARDS3)
1819 SYSCALL_DEFINE6(clone, unsigned long, clone_flags, unsigned long, newsp,
1820                 int, stack_size,
1821                 int __user *, parent_tidptr,
1822                 int __user *, child_tidptr,
1823                 unsigned long, tls)
1824 #else
```