TI DSP, MCU 및 Xilinx Zynq FPGA
프로그래밍 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 윤연성
whatmatters@naver.com

fork

fork 가 실행되면 페이지 테이블만 복사하고 물리 주소에 있는 데이터는 같은 데이터를 사용하다가, 자식 프로세스가 데이터를 수정하게 되면 그 부분의 데이터만 복사하여 다른 곳에 저장한다.

vfork

fork 와 마찬가지로 자식 프로세스를 생성하는 함수이지만, vfork 는 COW 가 적용되지 않고 계속 같은 자원을 공유한다.

exec 은 기존의 메모리가 날라감 (그뒤에 것을 실행할수 없다)
그래서 fork 를 하던가 자식을 exec 한다

clone 은 thread 가 될수도있음

vfork 찾기

ys@ys-Z20NH-AS51B5U:~/my_proj/kernel/linux-4.4$ vi -t task_struct

grep -rn “fork”

SYSCALL_DEFINE0 (fork) ./| grep “SYSCALL”

##이 붙으면 뒤에있는걸 붙여버림
sys_##name

sys_fork 분석해보기
NPTL 조사하기

getpid 는 pgid 값을 얻어오는거

커널내부에서는 pid 값을 가져옴!

NPTL

ㄹㅣ눅스커널 2.6 버전 이전에는 스레드 외에 특별한 기능없이
부분적으로 신호 처리, 스케줄링 및 프로세스 간 동기화 프리미티브와 관련하여 부분적으로 사용되며
스레드에 시스템을 양보하지않는 경우가 간혹있었다
NPTL 은 LinuxThreads 와 유사한 접근 방식을 사용합니다. 커널에서 알려진 기본 추상화는 여전히 프로세스이며 clone () 시스템 호출 (NPTL 라이브러리에서 호출 됨)을 사용하여 새로운 스레드를 만듭니다.
그러나 NPTL 은 쓰레드가 다시 깨어 야 할 필요가있는 동기화 프리미티브 (예를 들어)를 구현하기 위해 특별한 커널 지원을 필요로한다.
이것을 위해 사용되는 프리미티브는 퓨 텍스 (futex)로 알려져 있습니다. 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 대응합니다. 이것은 가능한 가장 간단한 스레딩 구현입니다.
NPTL 의 1 × 1 모델의 대안은 m × n 모델입니다

task_struct

thread_info

Thread_union = 커널 스택

가상메모리
pgb
R/b tree
VM_

bottom half : 논블로킹방식

p .76
만약 cpu 가 3 개 라도
1 ooo 1 번이 포크하면 이래도 3 번으로 안가고 1 번으로 포크
2 oo
3 o
이유는 캐시메모리를 쓰던거를 쓰는게 좋음
단 많은 데이터가 들어가면 내부적으로 load balancing(내부분산)
함 캐시메모리도 용량의 한계가 있기떄문

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

int g = 2; //전역변수

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]; //스레드가 사용할 메모리 공간 할당
int l = 3;
printf("PID(%d) : parnet g=%d,i=%d\n", getpid(), g, l);
clone(sub_func, (void *)(child_stack+4095), CLONE_VM|
CLONE_THREAD|CLONE_SIGHAND, NULL);
//메모리 공간 , 스레드가 사용할 메모리할당(배열이라 4095)
sleep(1);
printf("PID(%d) : parent g=%d, l =%d \n", getpid(), g, l);
return 0;
}
```

```
#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(%d) : child \n", getpid(),
syscall(__NR_gettid));
}
else
{
printf("TGID(%d), PID(%d) ; parent \n", getpid(),
syscall(__NR_getpid));
sleep(2);
}
printf("after fork \n\n");
return 0;
}
```

```
#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(%d) : child \n", getpid(),
syscall(__NR_gettid));
}
else
{
printf("TGID(%d), PID(%d) ; parent \n", getpid(),
syscall(__NR_getpid));
sleep(2);
}
printf("after fork \n\n");
return 0;
}
```