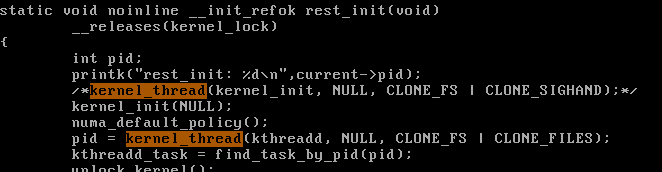
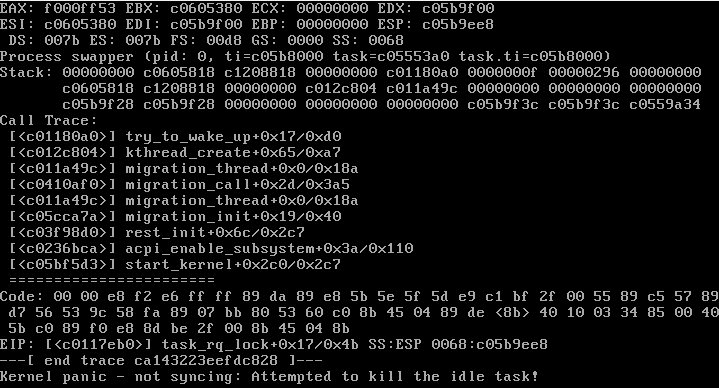
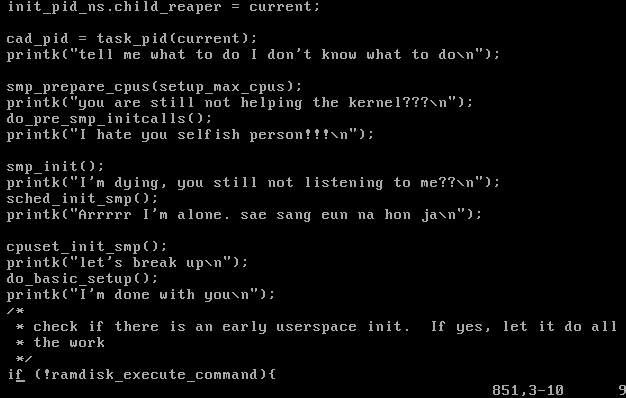
(여기는 핸드폰으로 찍음!!!)

11.3) What happens if the kernel calls "kernel\_init" directly instead of calling kernel\_thread(kernel\_init, ...) in rest\_init()? Trace the kernel code and show where the kernel falls into panic.





원래는 kernel thread로 인해 프로세스가 두개가 돼서 두번째 프로세스가 kernel\_init을 실행하게 되는데, 그 과정이 없이 바로 kernel\_init을 호출하면 프로세스가 init\_task 하나인 상태에서 kernel\_init으로 가게된다. 이렇게 커널을 직접 지우려고 시도하게 되기 때문에 커널 패닉이 일어난다.





핸드폰으로 부팅과정을 찍고 천천히 돌려봤을 때, 이전 과제에서 했던 것에 의해 “kernel\_init:0”이 제대로 나왔기 때문에 최소 kernel\_init 까지는 제대로 진입했단걸 알 수 있었다. 어디에서 문제가 발생하는지 알아보기 위해서 kernel\_init 안에 많은 printk들을 넣고, 재부팅 하는 과정을 핸드폰으로 찍고 천천히 돌려봐서 문제의 부분을 캡쳐했다. smp\_prepare\_cpus 함수의 전에 넣어뒀던 printk 메시지까지는 출력됐지만 그 뒤로는 출력이 안된걸로 보아 smp\_prepare\_cpus에서 문제가 발생했다고 볼 수 있다.

5.4) The last function call in start\_kernel() is rest\_init(). If you insert printk() after rest\_init(), it is not displayed during the system booting. Explain the reason.

void start\_kernel(){

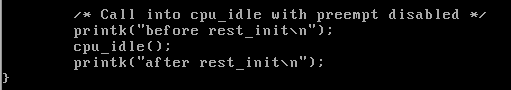
............

printk("before rest\_init\n"); // this will be printed out

rest\_init();

printk("after rest\_init\n"); // but this will not.

}





rest\_init 함수는 kernel\_thread를 통해서 kernel\_init을 만들고 kernel\_init은 다른 프로세스들을 만들고 rest\_init 함수는 그 다음 명령을 가리키게 된다. 그러고 나서 rest\_init 안의 cpu\_idle에 들어오면 거기에서 스케쥴러에 의해 priority가 더 높은 kernel\_init이 스케쥴된다. 그 뒤로 나머지 프로세스들은 block된 상태가 되고, 마지막 /sbin/agetty가 login:을 찍게 된다. 이제 사용자 입력을 대기하는 상태가 되고 다른 스케쥴할 스케쥴러가 없기 때문에 init\_task이 스케쥴 되지만, init\_task가 가리키는 위치에 있는 스케쥴러에 의해서 자기 자신만 계속해서 스케쥴하고 그렇게 무한루프를 돌게 되기 때문에 더 이상 cpu\_idle()의 다음 문장은 출력이 안된다. (과제 잘못했다ㅠㅠ 그래서 rest\_init 이후의 printk는 출력이 안된다.)

5.5) The CPU is either in some application program or in Linux kernel. You always should be able to say where is the CPU currently. Suppose we have a following program (ex1.c).

void main(){

printf("korea\n");

}

When the shell runs this, CPU could be in shell program or in ex1 or in kernel. Explain where is CPU for each step of this program. Start the tracing from the moment when the shell prints a prompt until it prints next prompt.

shell: printf("#"); // CPU is in shell

=> write(1, "#", 1) // CPU is in c library

=> INT 128 // CPU is in c library

kernel:

sys\_write() // CPU is in kernel and display '#' in screen

// after sys\_write() kernel schedules shell again

// and CPU goes back to shell

shell:

scanf(“%s”,buf) //CPU is in shell

(. pressed)

=>INT 33 //CPU is in c library

kernel:

atkbd\_interrupt() //CPU is in kernel and store .

//CPU goes back to cpu\_idle()

(. released)

=>INT 33

atkbd\_interrupt() //CPU is in kernel and ignore release

//CPU goes back to cpu\_idle()

(/ pressed)

=>INT 33

…….

…….

(enter pressed)

=>INT 33

atkbd\_interrupt() //store “./ex1” in shell’s buf and wakes up shell

shell:

=> read //CPU is in c library

=>INT 128 //CPU is in c library

kernel:

sys\_read() //CPU reads buf.

shell:

x=fork(); //CPU is in shell

=> fork //CPU is in c library

=>INT 128 //CPU is in c library

kernel:

sys\_fork() //CPU is in kernel and makes a child

//CPU goes back to shell

shell:

wait(); //CPU is in shell

=> wait //CPU is in c library

=> INT 128 //CPU is in c library

kernel:

sys\_wait() //CPU is in kernel and waits

//after sys\_wait() scheduler picks child

=> INT 128 //CPU is in c library

=> execve //CPU is in c library

=>sys\_execve() //CPU now runs ex1 and shows “korea”

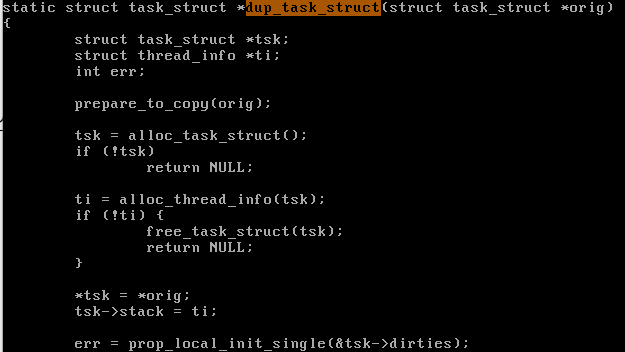
5.6) Trace fork, exec, exit, wait system call to find the corresponding code for the major steps of each system call.

fork:

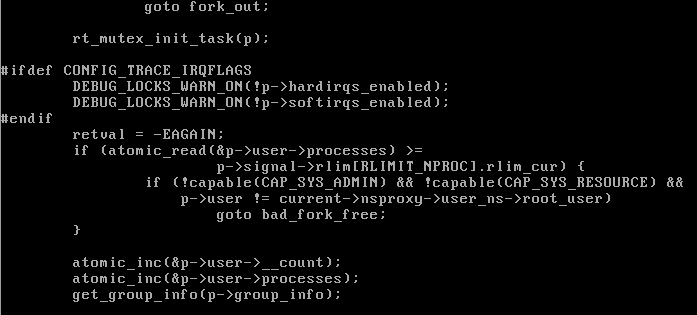
(1) copy the body of the parent process



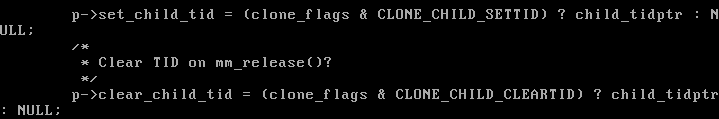
(2) copy thread\_union of the parent process, and adjust some information



(3) insert child into the process queue

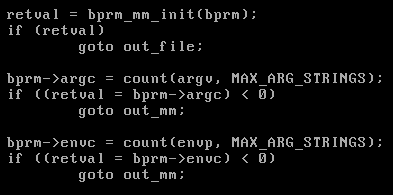


(4) return 0 to the child, and return child’s pid to the parent

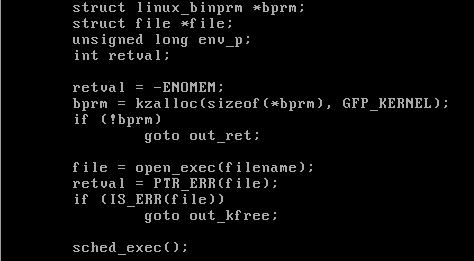


exec:

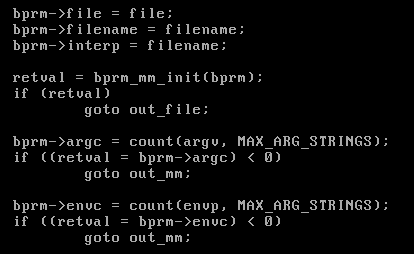
- remove old body



- load new body

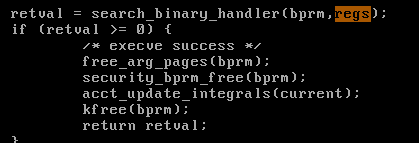


- update the task\_struct



- update the KMS (the stack portion in thread\_union) such that

KMS.eip = starting location of the new body

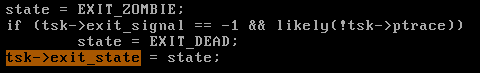


exit:

- remove body



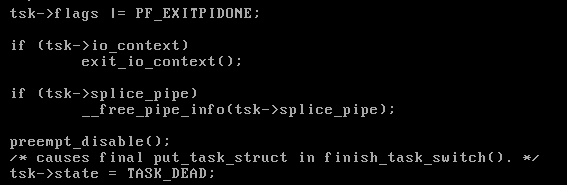
- make it a zombie



- send SIGCHLD to the parent



- adopt children to init process



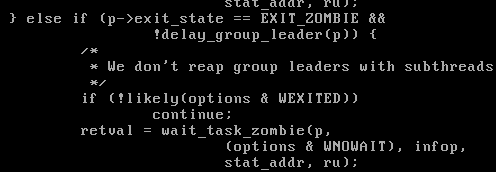
- schedule next process



wait:

if child has exit first (that is, if the child is a zombie)

let it die completely (remove its process descriptor)



else (child is not dead yet)

block parent



remove parent from the run queue



schedule next process



5.1.1) Try below (ex1.c) and explain the result.

#include <stdio.h>

unsigned long long sum=0;

void main(){

int x=fork();

if (x==0){ // child

int i, j;

for(i=0;i<20000;i++)

for(j=0;j<2000;j++)

sum++;

printf(“child sum:%llu\n”,sum);

}else{ // parent

int i, j;

for(i=0;i<20000;i++)

for(j=0;j<2000;j++)

sum++;

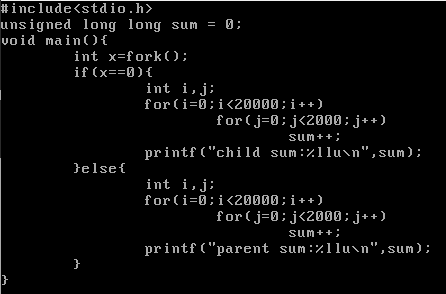
printf(“parent sum:%llu\n”,sum);

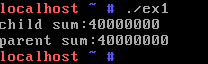
}

}

#gcc -o ex1 ex1.c

#./ex1





fork()를 하면 부모와 자식의 메모리 위치도 다르고 body도 공유하지 않기 때문에 parent에서의 sum과 child에서의 sum은 서로 다른 sum이된다. 그렇기 때문에 child와 parent 다 20000\*2000번인 40000000번 돌게 돼서 출력값이 40000000이 된다.

5.1.2) Try below (th.c) and explain the result.

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t t1, t2; // thread 1, thread 2

unsigned long long sum=0;

void \* foo1(void \*arg){

int i,j;

for(i=0;i<20000;i++){

for(j=0;j<2000;j++)

sum += 1;

}

printf(“thread 1 sum:%llu\n”, sum);

return NULL;

}

void \* foo2(void \*arg){

int i,j;

for(i=0;i<20000;i++){

for(j=0;j<2000;j++)

sum += 1;

}

printf(“thread 2 sum:%llu\n”, sum);

return NULL;

}

int main(void){

pthread\_create(&t1, NULL, &foo1, NULL);

pthread\_create(&t2, NULL, &foo2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}

# gcc -o th -lpthread th.c

#./th

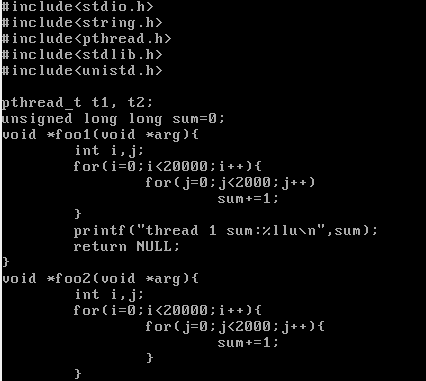
....

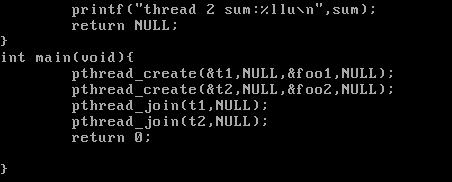
#./th

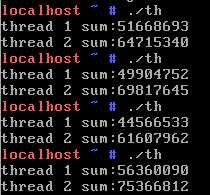
....

#./th

....







여기에선 pthread를 썼기 때문에 쓰레드 두개가 서로 body를 공유해서 같은 sum을 쓰게 된다. 여기서 문제점은 코드중 ++sum은 어셈블리코드로 MOV eax,sum INC eax MOV sum,eax와 같은데, 하나의 쓰레드에서 MOV sum,eax까지 다 끝내지 않고 중간 단계에서 멈췄을 경우, 다시 쓰레드가 동작할 때 처음부터 시작하지 않고 멈췄던 부분에서 시작하기 때문에 새로 업데이트된 sum값을 불러서 고치는게 아니라 과거의 sum값을 불러서 고치게 된다. 이렇게 race condition이 발생하고, 결국 thread2는 80000000에 도달하지 못하게 된다.

5.1.3) Try below(th2.c) and explain the result.

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t t1, t2; // thread 1, thread 2

pthread\_mutex\_t lock; // semaphore

unsigned long long sum=0;

void \* foo1(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock);

for(j=0;j<2000;j++)

sum += 1;

pthread\_mutex\_unlock(&lock);

}

printf(“thread 1 sum:%llu\n”, sum);

return NULL;

}

void \* foo2(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock);

for(j=0;j<2000;j++)

sum += 1;

pthread\_mutex\_unlock(&lock);

}

printf(“thread 2 sum:%llu\n”, sum);

return NULL;

}

int main(void){

pthread\_mutex\_init(&lock, NULL);

pthread\_create(&t1, NULL, &foo1, NULL);

pthread\_create(&t2, NULL, &foo2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

# gcc -o th2 -pthread th2.c

#./th2

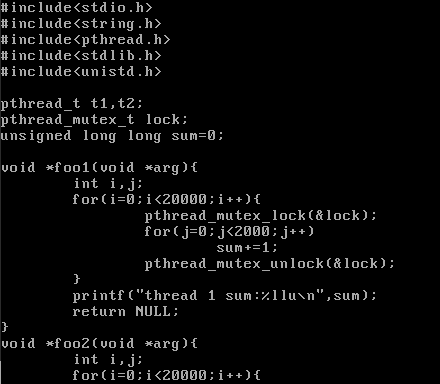
...

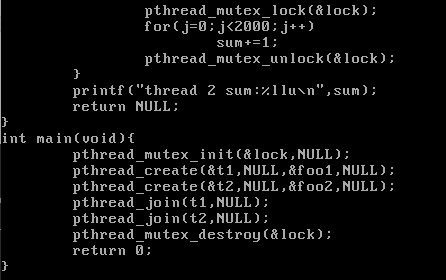
#./th2

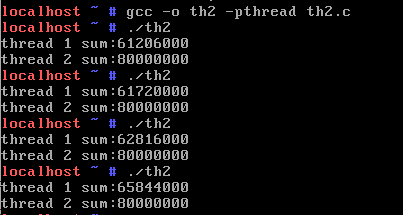
...

#./th2

.....







race condition이 일어나는 이유는 두개의 프로세스가 동시에 critical section에 접근했기 때문이다. 이를 방지하기 위해서 mutual exclusion의 방법중 하나인 semaphore를 쓸 수가 있다. 위의 코드가 바로 그 semaphore를 쓴 상황이다. pthread\_mutex\_lock를 사용해서 lock을 걸게 되면, 그 lock에 걸린 프로세스만 실행되고 그동안 다른 프로세스는 critical section에 접근하지 못하게 된다. 이런 방법을 쓰면 thread 2의 값은 정상적으로 80000000이 된다. (왜냐면 th1과 th2 다 C 단계에서 잘 멈추기 때문에)

(thread는 하나가 돌다가 타이머가 다하면 멈춘다.)

5.1.4) (Deadlock) Try below(th3.c) and explain the result.

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t t1, t2; // thread 1, thread 2

pthread\_mutex\_t lock1; // semaphore 1 for sum 1

pthread\_mutex\_t lock2; // semaphore 2 for sum 2

unsigned long long sum1=0;

unsigned long long sum2=0;

void \* foo1(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock1);

pthread\_mutex\_lock(&lock2);

for(j=0;j<2000;j++)

sum1 += 1;

pthread\_mutex\_unlock(&lock1);

for(j=0;j<2000;j++)

sum2 += 1;

pthread\_mutex\_unlock(&lock2);

}

printf(“thread 1 sum1:%llu\n”, sum1);

printf(“thread 1 sum2:%llu\n”, sum2);

return NULL;

}

void \* foo2(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock2);

pthread\_mutex\_lock(&lock1);

for(j=0;j<2000;j++)

sum1 += 1;

pthread\_mutex\_unlock(&lock1);

for(j=0;j<2000;j++)

sum2 += 1;

pthread\_mutex\_unlock(&lock2);

}

printf(“thread 2 sum1:%llu\n”, sum1);

printf(“thread 2 sum2:%llu\n”, sum2);

return NULL;

}

int main(void){

pthread\_mutex\_init(&lock1, NULL);

pthread\_mutex\_init(&lock2, NULL);

pthread\_create(&t1, NULL, &foo1, NULL);

pthread\_create(&t2, NULL, &foo2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_mutex\_destroy(&lock1);

pthread\_mutex\_destroy(&lock2);

return 0;

}

# gcc -o th3 -pthread th3.c

#./th3

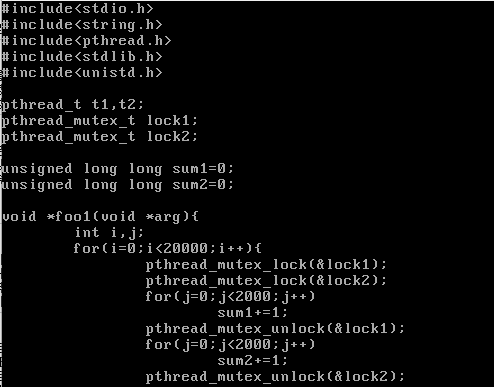
...

#./th3

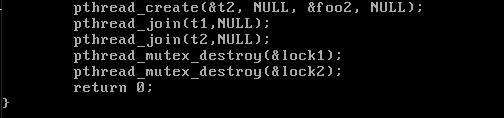
...

#./th3

.....









여기에서는 semaphore를 두개 이상을 쓰게 되는데, 이렇게 되면 두개 이상의 쓰레드가 서로의 작업이 끝나기만을 기다려서 작업이 영원히 안끝나게되는 상황이 올 수 있다. 그러면 위의 실행결과처럼 프로그램이 아예 멈추게 된다. 이런 상황을 데드락이라하고, 데드락의 발생조건은 네가지이고 그 네가지가 동시에 만족돼야 데드락이 발생하기 때문에 데드락을 막기 위해선 그 네가지 조건중 하나라도 발생하지 않도록 미리 예방하거나 조건은 그대로 두고 발생을 막는 알고리즘을 사용하거나 데드락이 발생하고 나서 해결하는 방법을 쓸 수가 있다.

(lock1이 자원을 사용하고 있는 상태에서 lock2가 요청.)