# Семинар 2 ОС (02.03.2022)

…

В linux дескриптор процесса называется struct task\_struct

Содержит всю необходимую системе информацию для управления процессом, большая часть которой явлется указателями на другие структуры. У процесса виртуальное адресное пространство.

Первая строка этой структуры – это:  
struct thread\_info thread\_info;

task\_struct заканчивается (после этого ничего не может быть):  
struct thread\_struct thread;

Всегда в программе будет основной поток.

int prio; // не единственное поле связанное с планированием  
int \_\_state; // (двойное подчёркивание перед state)

Интересная тема для чтения (то, что связано с планированием)

Очень интересное поле:

struct list\_head tasks; //всё через списки

В ядре используются связные списки, потому что использование массивов – трудозатратно.

…

pid\_t pid;   
…  
struct **task\_stuct** \_\_rcu \*parent// структура описывающая дескриптор процесса предка (двойное подчёркивание перед rcu)  
…  
struct pid \*thread\_pid; //указатель на struct pid

В рамках нашего задания, мы должны вывести в syslog имя файла.

char comm[TASK\_COMM\_LEN]; // ?имя файла которым был процесс до того как мы его не запустили на выполнение?

Есть структуры ?описывающие файловую подсистему? (Сейчас рассматривать их не будем, будет ЛР Открытые файлы).

!Еще раз: Дескриптор процесса содержит всю необходимую информацию чтобы можно было выполнять этот процесс и управлять этим процессом (выделять ему ресурсы).

…  
struct thread\_struct thread;  
}

## Загружаемый модуль ядра

В ядре мы не можем использовать стандартные библиотеки C. Мы можем использовать только функции, структуры … ядра. Это сделано для безопасности. Часто при просмотре ядра вы будете наталкиваться на функции, начинающиеся с двойного подчёркивания. По существу, эти два подчёркивания говорят “убедитесь, что вы знаете, что делаете”.  
#include <linux/indule.h>  
#include <linux/kernel.h>  
static int \_\_init md\_init(void)  
{  
 struct task\_struct \*task=&init\_task;  
 do  
 {  
 printk(KERN\_INFO “\_pid = %d, name = %s, …, ppid=%d, p\_name=%sin”,  
 task->pid, task->comm, …, task->parent->pid, task->parent->comm);  
 }while((task = next\_task(task)) != finit\_task);

printl(KERN\_INFO “…\n”, …); //current (вместо task - current)  
 return 0;  
}

Символ current  
 Insmode  
 lsmode  
 rmmode

static void \_\_exit md\_exit(void)  
{  
 printk(KERN\_INFO “Exit\n”);   
}  
module\_init(md\_init);

module\_exit(md\_exit);

Это структура загружаемого модуля ядра.

Кроме того, что мы написали, в модулях принято писать макросы для установки информации для модулей. (На мой взгляд, единственный обязательный MODULE\_LICENSE)

Можно после #include’ов, можно после #define’ов  
… #include…  
MODULE\_LICENSE (“GPL”);

… static…

Давайте посмотрим простейший makefile.

obj -m += module.ko  
all:  
 make -c /lib/modules/$/shell uname -r)/build M=$(PWD) modules  
clean:

make -c /lib/modules/$/shell uname -r)/build M=$(PWD) clean

-c – опция – смена каталога

M – опция которая заставляет makefile вернуться в директорию … ?исходных кодов?

Вторая часть нашей лабораторной работы выполняется по этому материалу, который у Цилюрика называется “Первые шаги”. 2ая часть называется взаимодействие модулей ядра.

Программы могут взаимодействовать через разделяемые переменные/память. (Очень хорошо расписано в книге Пильщикова Assembler)

Assembler – собрание. Собрание команд. Это мнемоническое представление машинных кодов. Соответственно должен быть транслятор, и он тоже называется assembler.

md.h

extern char \*md1\_data;  
extern char \*md1\_proc(void);

md1.c

#include <linux/init.h>  
#include <linux/module.h>  
#include “md.h”  
MODULE\_LICENSE(“GPL”);  
char \*md1\_data = “Hello world”; //”aaa”  
extern char \*md1\_proc(void)  
{  
 return md1\_data;  
}  
static char md1\_local(void)  
{  
 return md1\_data;  
}  
extern char \*md1\_noexport(void)  
{  
 return md1\_data;  
}

EXPORT\_SYMBOL(md1\_data);  
EXPORT\_SYMBOL(md1\_proc);  
static int \_\_init md\_init (void)  
{  
 printk(“+module start\n”);  
 return 0;  
}  
static void \_\_exit md\_exit(void)  
{  
 printk(“+Exit\n”);  
 return 0;  
}  
module\_init(md\_init);  
module\_exit(md\_exit);

В файле md2.c

#include <linux/init.h>  
#include <linux/module.h>  
#include “md.h”  
MODULE\_LICENSE(“GPL”);  
static int \_\_init md\_init(void)  
{  
 printk(“+module2 start\n”);  
 printk(“+string: %s\n”, md1\_data);  
 printk(“+string proc: %s\n”, md1\_proc());  
 return 0;  
}

Если в этом модуле вызвать функцию local, то возникнет ошибка (Когда? Какое сообщние ошибки?). Если вызвать noextern (Те же вопросы).

В этой работе есть файл md3.c такой же как md2, но в init’е стоит return -1; соответственно нужно посмотреть, что произойдёт, будет ли загружен модуль md3 в ядро. Нужно будет прочитать что он там пишет.

Особое внимание обратить на перечисление 1 2 3 4 (?В методе?). Особенно пункт 2. … Связываются по прямой… Абсолютный адрес – физический адрес в наших системах - байта памяти (ячейка памяти). Каждый байт имеет уникальный адрес. … Поэтому имеет значение последовательность загрузки и выгрузки модулей в память.

\*(Тип влияет на размер участка памяти выделяемой под переменную определённого типа)