Assignment 1: System Call

Jan Lucca Thümmel - jathu21@student.sdu.dk

March 2, 2024

Contents

0.1 Introduction

The aim of this project is to implement two system calls to handle interprocess communication. This is achieved by implementing a message box in kernel space. The system calls must be able to retrieve a message from kernel to user space and place a message in the message box from user space. The message box is implemented by a stack, were incoming messages are placed on the top of the stack and messages are retrieved from the top of the stack.

0.2 Design Choices

During the process of implementing the project there where multiple design choices to be made.

For example was it a requirement to implement the message box using a stack, were new messages were put on top of the stack by the system call and the top message would always be retrieved. Using a stack in this way makes it easier to retrieve messages as there is no need to look for specific messages. This is great from an implementation point of view, because it makes it easier to implement. But it also puts restrictions on how we can use the message box.

The two system calls both take a pointer to a buffer. For the system call that retrieves a message, the pointer points to the buffer where the message should be copied to from. For the system call that stores a message on the stack the pointer points to the message that we want to store. They also both have the parameter length which is an integer that defines the size of the buffer.

Another design choice is how big the messages can be, that can be stored on the stack in kernel space. In this implementation one message is limited to 101 characters. 101 was chosen because it allows to have 100 characters and the 0 terminator. Messages can not be shorter than one character as a message with less than that would not make any sense and just cause trouble.

For error handling the error codes for system calls are used. These error codes are defined in the file ”errno.h” in the Linux source code. These error codes are negative integers that are returned. An example use is the use of ”-EINVAL” which is used if an invalid argument is passed. For example when the message is to long or to short. Another one is ”-ENOMEM” which is used when ”kmalloc” was not able to allocate memory for the message. ”-EFAULT” is used when ”access_ok” checks if it is safe to read or write from user space memory.

One important design choice is how to handle simultaneous access to resources of the message box. If this is ignored the system call could be interrupted while in a critical region, causing problems. So it is important that data is accessed and modified atomically to prevent race conditions. To deal with this, interrupts can be temporarily disabled in specific regions of the code. This is done with ”local_irq_save(flags);” and ”local_irq_restore(flags);”. These have to be used with caution and not unnecessarily often as critical systems like the CPU clock run on interrupts. If interrupts are disabled the CPU clock could get out of sync. This could have repercussions for other parts of the system. These functions disable interrupts when specific parts of the code are running. In this case specifically operations involving memory. Disabling interrupts is in this case probably the more efficient approach if compared to locks. Locks require more overhead but are also safer to use in comparison to disabling interrupts. But as the critical sections in this program are rather short interrupts are disabled only for a short amount of time.

One bad design choice that was made is that both system calls are implemented independently instead of in the message box file. This results in that the system call implementation calls the functions from the message box instead of being implemented directly in the message box. This implementation is quite unnecessary. It does result in some modularity but at what cost. The two system calls are also in the appendix 0.6.1.

0.3 Implementation

Because we are working in kernel space the implementation differs slightly from what it would look like in user space. For example "kmalloc" has to be used instead of the normal "malloc". The main differences between "kmalloc" and "malloc" are that "kmalloc" is specifacly for kernel memory. "kmalloc" also requires more parameters than "malloc", it needs a flag that tells it how to allocate memory. In this case "GFP_KERNEL" is used. This flag makes "kmalloc" wait until memory is available if there is none. There are also flags that would return instantly if no memory was available. Below is an example of how "kmalloc" is used. As can be seen it does not differ much from the normal "malloc".

```
msgStack* msg = kmalloc(sizeof(msgStack), GFP_KERNEL);
if(msg == NULL){
    return -ENOMEM;
}
```

As seen in the example above error handling is also close to normal C code, but system call error codes are used.

With great "kmalloc" there comes great "kfree". As this is real programming we need to free the memory we allocate using "kmalloc". For that we use "kfree", it is the kernel equivalent to the normal "free" it is used just like the normal C "free". It just frees memory in kernel space. Here is an example from the project code

```
kfree(msg -> message);
kfree(msg);
```

Because interacting with user space memory from kernel space can give complications the "access_ok" function is used to prevent memory violations. Before reading from or writing to user space, "access_ok" checks if the address is viable and accessible thus protecting the kernel from accessing illegal memory. Below is an example from the "dm510_msgbox_put" system call where we check if the location is safe before "copy_from_user()" is used to copy from user memory.

```
if(!access_ok(buffer, length)){
    return -EFAULT;
}
int copyUser = copy_from_user(msg -> message, buffer, length);
if(copyUser != 0){
    return -EFAULT;
}
```

the functions "copy_from_user" and "copy_to_user" are used to move data between user and kernel space. "copy_from_user" transfers data from user space to kernel space. "copy_to_user" copies data from kernel to user space. An example from the code is shown below. As arguments both functions take a pointer to were the data should be moved and a pointer from were. They also take an integer representing how many bytes should be moved. Both functions return zero on success which can be used for error handling.

```
int copyUser = copy_from_user(msg -> message, buffer, length);
if(copyUser != 0){
    return -EFAULT;
}
```

```
int cpyToUser = copy_to_user(buffer, msg->message, msgLength);
if(cpyToUser != 0){
    return -EFAULT;
}
```

As mentioned before the message box uses a stack to manage messages new messages are put on top of the stack. The top message is also always the first one to be retrieved. When pushing a message onto the stack it becomes the new top. We also check if the stack is empty, if it is the pushed message also becomes the bottom. To keep track of the messages below top, every message also references the previous message. If the message was the first message on the stack the previous message is set to "NULL" otherwise the new messages previous pointer points to the old top message. If the stack is not empty the pushed message becomes the new top. The push operation is shown below.

```
//if the stack is empty
if(bottom == NULL){
    bottom = msg;
    top = msg;
    msg -> previous = NULL;
}
// if stack is not empty
else{
    msg -> previous = top;
    top = msg;
}
```

For the pop operation the message top points to is removed from the stack. Because every message points its previous message, the previous message top points to becomes the new top. If top becomes "NULL", because the last message has been retrieved, we return from the function. The implementation is shown below.

```
if(top != NULL) {
    msgStack* msg = top;
    int msgLength = msg->length;
    top = msg->previous;

    // some other stuff
}
```

The code handles multiple error conditions to make the program as safe and resistant as possible. If the message to be put in the message box is to long or to short we gracefully return the appropriate system call error code as shown below.

```
if(length <= 0){
    return -EINVAL;
}
if(length > 101){
    return -EINVAL;
}
```

Memory allocation failures are also handled with the appropriate error code as shown below.

```
msgStack* msg = kmalloc(sizeof(msgStack), GFP_KERNEL);
if(msg == NULL){
    return -ENOMEM;
}
```

Access validation error handling is also implemented to handle, if memory access operations fail or are invalid, as well as protection against reading or writing outside allowed memory locations. This is also shown below

```
if(!access_ok(buffer, length)){
    return -EFAULT;
}
int copyUser = copy_from_user(msg -> message, buffer, length);
if(copyUser != 0){
    return -EFAULT;
}
```

Another important part of the program is the use of flags to disable interrupts. This is done for parts that access memory or shared data to protect data integrity and prevent race conditions. Below is a section that uses the flags to disable interrupts.

```
unsigned long flags;
local_irq_save(flags); //start of critical region

//if the stack is empty
if(bottom == NULL){
    bottom = msg;
    top = msg;
    msg -> previous = NULL;
}
// if stack is not empty
else{
    msg -> previous = top;
    top = msg;
}
local_irq_restore(flags); // critical region end
```

These code snippets show the important and interesting parts of the program

0.4 Testing

Testing is an important part of the development process to ensure that the program behaves as intended and that possible bugs are fixed and to make sure that the program is safe to use. For this project there are multiple tests that come to mind. These tests are basic tests that show that the program works as intended. This was tested by giving a message to the program through the system call and then retrieving the message through the other system call. Because there could be put multiple messages on the stack before retrieving any this was also tested. The code that was used to test the program can be seen in the appendix 0.6.1.

```
void testBasicFunc(){
    char *in = "This is the basic function test";
    char msg[99];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, strlen(in));

    msglen = syscall(__NR_dm510_msgbox_get, msg, 99);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}
```

This is the test to test if the program works as intended.

After confirming that the program works as intended it is important to test if it also behaves as intended with invalid inputs. As discussed before the choice was made to limit the number of characters to between one and onehundred. This choice was primarily made as the need for longer messages was not deemed useful and because for extremely long messages it would probably be better to implement a shared memory system. So the tests include tests for inputs of length 0, negative length as well as length bigger than 100. Messages of length one were also tested. Other important test are if we can retrieve anything from the stack without putting anything on to the stack before hand. All these tests behaved as expected. The code and output can be seen in the appendix 0.6.1. Two important tests are also to test the program with bad pointers. The program was tested with a null pointer which ended in a segmentation fault, as expected and with an uninitialized pointer. The test with an uninitialized pointer resulted in a segmentation fault or in accessing a random part of memory. This is because an uninitialized pointer points to a random place in memory. The results for the NULL pointer and the uninitialized memory were as expected. These tests are also in the appendix 0.6.1. The behavior of the uninitialized pointer is not optimal as it is never a good idea to have a program access random memory location. In future iterations of the program it would be appropriate to implement a solution for that.

As mentioned before we could run into race conditions with these system calls we test this by spawning multiple threads that operate on the message box. But we can not guarantee that there will be a simultaneous access of to threads to the message box. Thus it is not 100 percent guaranteed that our implementation of disabling interrupts works perfectly, but it is a good indicator that it should work. The code is also included in the appendix.

as all test are run from one file one after another, the tests are executed in the following order in the video: Testing basic functions, Test with zero length, test with length one, concurrency test, test with length 100, test with length 200, test with negative length, test with multiple messages, test with an empty stack, test with a bad address and the test with a NULL address. The time in the where the tests start and the output is shown is 00:34.

0.5 Conclusion

The message box was successfully implemented by using a stack to manage messages, using a LIFO(last in first out) pattern. Memory allocation in kernel space was successfully implemented using "kmalloc" and "kfree". The safe communication between user and kernel space was successfully implemented using "access_ok", "copy_from_user" and 'copy_to_user". The code handles potential errors conditions for invalid message lengths, memory allocation failure and invalid user space addresses. Thereby the code handles common risks like memory leaks, kernel panics or unauthorized memory access. By disabling local interrupts we handle that concurrent access does not corrupt data or lead to race conditions. While the current implementation works as intended there are multiple areas that could be improved in later iterations of the program. For example does the program not handle uninitialized memory pointers well. In future iterations that may include longer critical sections, more sophisticated locks could be implemented, like spin or mutex locks. Better testing of concurrency would also be an improvement for the program. Overall the program does what it was intended to do. This was confirmed by testing the program with multiple different scenarios.

0.6 Apendix

0.6.1 Code

msgbox

```
#include </home/jathu21/dm510/linux-6.6.9/include/linux/slab.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include </home/jathu21/dm510/linux-6.6.9/arch/um/include/asm/uaccess.h>
#include "linux/kernel.h"
#include "linux/unistd.h"
#include "/home/jathu21/dm510/linux-6.6.9/arch/um/include/asm/dm510_msgbox.h"

typedef struct _msgStack msgStack;

struct _msgStack {
    msgStack* previous;
    int length;
    char* message;
};

static msgStack *bottom = NULL;
static msgStack *top = NULL;


int dm510_msgbox_put(char* buffer, int length){
    if(length <= 0){
        return -EINVAL;
    }
    if(length > 101){
        return -EINVAL;
    }
    msgStack* msg = kmalloc(sizeof(msgStack), GFP_KERNEL);
    if(msg == NULL){
        return -ENOMEM;
    }

    msg -> previous = NULL;
    msg -> length = length;
    msg -> message = kmalloc(length, GFP_KERNEL);
    if(msg -> message == NULL){
        kfree(msg);
        return -ENOMEM;
    }

    if(!access_ok(buffer, length)){
        return -EFAULT;
    }

    int copyUser = copy_from_user(msg -> message, buffer, length);
    if(copyUser != 0){
        return -EFAULT;
    }

    unsigned long flags;
    local_irq_save(flags); //start of critical region

    //if the stack is empty
    if(bottom == NULL){
        bottom = msg;
        top = msg;
        msg -> previous = NULL;
    }
    // if stack is not empty
    else{
        msg -> previous = top;
        top = msg;
    }
    local_irq_restore(flags); // critical region end

    return 0;
}


int dm510_msgbox_get(char* buffer, int length){
    unsigned long flags;
    local_irq_save(flags); //start of critical region
    if(top != NULL) {
        msgStack* msg = top;
        int msgLength = msg->length;

        top = msg->previous;

        if(!access_ok(buffer, length)){
            return -EFAULT;
        }
        int cpyToUser = copy_to_user(buffer, msg->message, msgLength);
        if(cpyToUser != 0){
            return -EFAULT;
        }

        kfree(msg -> message);
        kfree(msg);

        return msgLength;
    }
    local_irq_restore(flags); // critical region end

    return -1;

}
```

Test

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include "arch/x86/include/generated/uapi/asm/unistd_64.h"
#define NUM_THREADS 10


void testBasicFunc(){
    char *in = "This is the basic function test";
    char msg[99];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, strlen(in));

    msglen = syscall(__NR_dm510_msgbox_get, msg, 99);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}


void testZeroLeng(){
    char *in = "";
    char msg[99];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, 0);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 99);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    //printf("%ld", result);
```

```
}

void testOneLeng(){
    char *in = "J";
    char msg[99];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, strlen(in)+1);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 99);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    //printf("%ld", result);
}

void test100Leng(){
    char *in = "87WHsekAo4OSElsK9BkLuaLLwnZeU9vdBRSCa1HzvrMqjPEH9SV
    pUyPy9NKnxS6vPHy25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf";
    char msg[100];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, strlen(in)+1);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void test200Leng(){
    char *in = "87WHsekAo4OSElsK9BkLuaLLwnZeU9vdBRSCa1HzvrMqjPEH9SVpUyPy
    9NKnxS6vPHy25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf87WHsekAo4OSElsK9BkLuaL
    LwnZeU9vdBRSCa1HzvrMqjPEH9SVpUyPy9NKnxS6v
    PHy25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf";
    char msg[100];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, strlen(in)+1);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void testNegativeLength(){
    char *in = "ghjghj";
    char msg[100];
    long msglen;

    printf("message: %s\n", in);
    syscall(__NR_dm510_msgbox_put, in, -1);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void testMultipleMsg(){
    char msg[100];
    long msglen;
    char *in1 = "Hello1";
    char *in2 = "Hello2";
    char *in3 = "Hello3";
    char *in4 = "Hello4";
    char *in5 = "Hello5";
    char *in6 = "Hello6";

    printf("message: %s\n", in1);
    syscall(__NR_dm510_msgbox_put, in1, strlen(in1)+1);
    printf("\n");

    printf("message: %s\n", in2);
    syscall(__NR_dm510_msgbox_put, in2, strlen(in2)+1);
    printf("\n");

    printf("message: %s\n", in3);
    syscall(__NR_dm510_msgbox_put, in3, strlen(in3)+1);
    printf("\n");

    printf("message: %s\n", in4);
    syscall(__NR_dm510_msgbox_put, in4, strlen(in4)+1);
    printf("\n");

    printf("message: %s\n", in5);
    syscall(__NR_dm510_msgbox_put, in5, strlen(in5)+1);
    printf("\n");

    printf("message: %s\n", in6);
    syscall(__NR_dm510_msgbox_put, in6, strlen(in6)+1);
    printf("\n");

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void testGetEmpty(){
    char msg[100];
    long msglen;

    msglen = syscall(__NR_dm510_msgbox_get, msg, 100);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void testBadAdr(){
    char *in;
    char msg[99];
    long msglen;


    syscall(__NR_dm510_msgbox_put, in, strlen(in));

    msglen = syscall(__NR_dm510_msgbox_get, msg, 99);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void testNULLAdr(){
    char *in = NULL;
    char msg[99];
    long msglen;


    syscall(__NR_dm510_msgbox_put, in, strlen(in));

    msglen = syscall(__NR_dm510_msgbox_get, msg, 99);
    printf("message length: %ld\n", msglen);
    printf("the message: %s\n", msg);
}

void* test_msgbox_concurrency(void* arg) {
    char *in = "Concurrent message test";
    char msg[102];
    long msglen;

    // Put a message into the box
    syscall(__NR_dm510_msgbox_put, in, strlen(in));

    // Get a message from the box
    msglen = syscall(__NR_dm510_msgbox_get, msg, sizeof(msg)-1);
    msg[msglen] = '\0'; // Ensure null-termination

    printf("Thread %ld: message length: %ld\n", pthread_self(), msglen);
    printf("Thread %ld: the message: %s\n", pthread_self(), msg);

    return NULL;
}

void testConcurrency() {

    pthread_t threads[NUM_THREADS];
    int i;

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, test_msgbox_concurrency, NULL);
    }

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}

int main(int argc, char ** argv) {
    printf("\n");
    printf("Basic function test:\n");
    testBasicFunc();
    printf("\n");

    printf("Zero length test:\n");
    testZeroLeng();
    printf("\n");

    printf("One length test:\n");
    testOneLeng();
    printf("\n");
    printf("Concurrency test:\n");
    testConcurrency();
    printf("\n");

    printf("100 length test:\n");
    test100Leng();
    printf("\n");

    printf("200 length test:\n");
    test200Leng();
    printf("\n");

    printf("Negative length test:\n");
    testNegativeLength();
    printf("\n");

    printf("Multiple messages test:\n");
    testMultipleMsg();
    printf("\n");

    printf("Empty stack test:\n");
    testGetEmpty();
    printf("\n");

    printf("Bad address test:\n");
    testBadAdr();
    printf("\n");

    printf("Null address test:\n");
    testNULLAdr();
    printf("\n");



    return 0;
}
```

Test Output

```
Basic function test:
message: This is the basic function test
message length: 31
the message: This is the basic function test

Zero length test:
message:
message length: 4294967295
the message:   @
```

```
One length test:
message: J
message length: 2
the message: J

Concurrency test:
Thread 1092757184: message length: 23
Thread 1092757184: the message: Concurrent message test
Thread 1109542592: message length: 23
Thread 1109542592: the message: Concurrent message test
Thread 1101149888: message length: 23
Thread 1101149888: the message: Concurrent message test
Thread 1117935296: message length: 23
Thread 1117935296: the message: Concurrent message test
Thread 1084364480: message length: 23
Thread 1084364480: the message: Concurrent message test
Thread 1126328000: message length: 23
Thread 1126328000: the message: Concurrent message test
Thread 1134720704: message length: 23
Thread 1134720704: the message: Concurrent message test
Thread 1143113408: message length: 23
Thread 1143113408: the message: Concurrent message test
Thread 1159898816: message length: 23
Thread 1159898816: the message: Concurrent message test
Thread 1151506112: message length: 23
Thread 1151506112: the message: Concurrent message test

100 length test:
message: 87WHsekAo4OSElsK9BkLuaLLwnZeU9vdBRSCa1HzvrMqjPE
H9SVpUyPy9NKnxS6vPHy25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf
message length: 101
the message: 87WHsekAo4OSElsK9BkLuaLLwnZeU9vdBRSCa1HzvrMq
jPEH9SVpUyPy9NKnxS6vPHy25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf

200 length test:
message: 87WHsekAo4OSElsK9BkLuaLLwnZeU9vdBRSCa1HzvrMqjPEH9SV
pUyPy9NKnxS6vPHy25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf87WHsekAo4
OSElsK9BkLuaLLwnZeU9vdBRSCa1HzvrMqjPEH9SVpUyPy9NKnxS6vPHy
25Br8cT2vJEYtKCwimdfUfvHCHjitZrtf
message length: 4294967295
the message:   @

Negative length test:
message: ghjghj
message length: 4294967295
the message:   @

Multiple messages test:
message: Hello1
```

```
message: Hello2

message: Hello3

message: Hello4

message: Hello5

message: Hello6

message length: 7
the message: Hello6
message length: 7
the message: Hello5
message length: 7
the message: Hello4
message length: 7
the message: Hello3
message length: 7
the message: Hello2
message length: 7
the message: Hello1

Empty stack test:
message length: 4294967295
the message:   @

Bad address test:
message length: 4
the message: ‘!@

Null address test:
testsystemcall[690]: segfault at 0 ip 0000000040192d59 sp 0000007fbff1fb58
error 4 in libc.so.6[40063000+155000]
Segmentation fault
```

other

dm510_msgbox_get

```
#include "linux/kernel.h"
#include "linux/unistd.h"
#include "/home/jathu21/dm510/linux-6.6.9/arch/um/include/asm/dm510_msgbox.h"

asmlinkage
int sys_dm510_msgbox_get(char* buffer, int length) {
    return dm510_msgbox_get(buffer, length);
}
```

dm510_msgbox_put

```
#include "linux/kernel.h"
#include "linux/unistd.h"
#include "/home/jathu21/dm510/linux-6.6.9/arch/um/include/asm/dm510_msgbox.h"

asmlinkage
int sys_dm510_msgbox_put(char* buffer, int length) {
    return dm510_msgbox_put(buffer, length);
}
```

dm510_msgbox.h

```
#ifndef __DM510_MSGBOX_H
#define __DM510_MSGBOX_H

extern int dm510_msgbox_put( char*, int );
extern int dm510_msgbox_get( char*, int );

#endif //JATHU21_DM510_MSGBOX_H
```

Edited Makefile

```
 SPDX-License-Identifier: GPL-2.0
#
# Copyright (C) 2002 - 2007 Jeff Dike (jdike@{addtoit,linux,intel}.com)
#

# Don't instrument UML-specific code; without this, we may crash when
# accessing the instrumentation buffer for the first time from the
# kernel.
KCOV_INSTRUMENT                := n

CPPFLAGS_vmlinux.lds := -DSTART=$(LDS_START) \
                        -DELF_ARCH=$(LDS_ELF_ARCH) \
                        -DELF_FORMAT=$(LDS_ELF_FORMAT) \
$(LDS_EXTRA)
extra-y := vmlinux.lds

obj-y = config.o exec.o exitcode.o irq.o ksyms.o mem.o \
physmem.o process.o ptrace.o reboot.o sigio.o \
signal.o sysrq.o time.o tlb.o trap.o \
um_arch.o umid.o maccess.o kmsg_dump.o \
capflags.o hellokernel.o dm510_msgbox_put.o \
dm510_msgbox_get.o dm510_msgbox.o skas/
obj-y += load_file.o

obj-$(CONFIG_BLK_DEV_INITRD) += initrd.o
obj-$(CONFIG_GPROF) += gprof_syms.o
obj-$(CONFIG_OF) += dtb.o
obj-$(CONFIG_EARLY_PRINTK) += early_printk.o
obj-$(CONFIG_STACKTRACE) += stacktrace.o
obj-$(CONFIG_GENERIC_PCI_IOMAP) += ioport.o
```

```
USER_OBJS := config.o

include $(srctree)/arch/um/scripts/Makefile.rules

targets := config.c config.tmp capflags.c

# Be careful with the below Sed code - sed is pitfall-rich!
# We use sed to lower build requirements, for "embedded" builders for instance.

$(obj)/config.tmp: $(objtree)/.config FORCE
$(call if_changed,quote1)

quiet_cmd_quote1 = QUOTE   $@
      cmd_quote1 = sed -e 's/"/\\"/g' -e 's/^/"/' -e 's/$$/\\n",/' \
   $< > $@

$(obj)/config.c: $(src)/config.c.in $(obj)/config.tmp FORCE
$(call if_changed,quote2)

quiet_cmd_mkcapflags = MKCAP   $@
      cmd_mkcapflags = $(CONFIG_SHELL) $(srctree)/$(src)/../../x86/kernel/
      cpu/mkcapflags.sh $@ $^

cpufeature = $(src)/../../x86/include/asm/cpufeatures.h
vmxfeature = $(src)/../../x86/include/asm/vmxfeatures.h

$(obj)/capflags.c: $(cpufeature) $(vmxfeature) $(src)/../../x86/kernel/
cpu/mkcapflags.sh FORCE
$(call if_changed,mkcapflags)

quiet_cmd_quote2 = QUOTE   $@
      cmd_quote2 = sed -e '/CONFIG/{'          \
  -e 's/"CONFIG"//'            \
  -e 'r $(obj)/config.tmp'     \
  -e 'a \'                     \
  -e '""'                      \
  -e '}'                       \
  $< > $@
```

Last part of unistd_64.h

```
#define __NR_map_shadow_stack 453
#define __NR_hellokernel 454
#define __NR_dm510_msgbox_put 455
#define __NR_dm510_msgbox_get 456


#ifdef __KERNEL__
#define __NR_syscalls 457
#endif
```

```
#endif /* _UAPI_ASM_UNISTD_64_H */
```

Important part from syscall_64.tbl

```
453 64 map_shadow_stack sys_map_shadow_stack
454 common  hellokernel     sys_hellokernel
455 common  dm510_msgbox_put    sys_dm510_msgbox_put
456 common  dm510_msgbox_get    sys_dm510_msgbox_get
```