console.c

console_write()

Seems like it just takes in a string and a len and prints something using the host operation that we provide. This could be useful as an initial test as there is a tangible output that uses a hos operation.

I am not sure why one of the parameters is a console since it is not called anywhere. Also that could give an error based off of which compiler flags are being thrown in.

struct console lkl_boot_console

It looks like it initializes a type of console, is this a console the user sees or a virtual console that is just interfacing with actual linux? I assume because we are using hostops print we are actually seeing the output that is going through this console. Also why is the index -1, does that mean that it is not initialized.

lkl_boot_console_init
This answers some of the questions I was pondering about the console not being initialized yet.

I guess the structure of the code12 to 26:

If we have a flag of LKL_EARLY CONSOLE, we define an empty “lkl_boot_console” and then linux register it so we can use it later.

For the next segment of the code we do a similar thing except we declare a regular lkl_console. We then initialize it using lkl stuff.

From console.c I see some functions that are used, but not declared. Mainly, early_initcall and core_initcall

cpu.c
From the description it looks like there is only one thread (?) that is allowed to use linux (?) at a time.

The lkl_cpu structure is what controls what accesses the CPU that is executing linux code.

__cpu_try_get_lock
It looks like this is the method that allows a thread to access the cpu. It checks to make sure we are not at a thread level above max threads. It then iterates a count variable which is a variable for the amount of times a thread has interacted (?) with the cpu.
__cpu_try_get_unlock

If lock return (I think this means that the thread is currently locked) is greater than -2 we unlock the thread.

lkl_cpu_change_owner
We first lock the thread that is the owner, and then we set the owner to our thread that we want to be a new owner.

lkl_cpu_get
The while loop kind of confuses me. It checks to a thread is not the owner i.e it is accessing the cpu. It unlocks the thread and then goes to the next thread that it waiting in the semaphore i.e the next thread inline for the CPU. It basically gets the next thread that isn't sleeping(?)

void lkl_cpu_put(void)
First we lock the mutex that is controlling the cpu. The we check to make sure that we are currently the thread in the cpu. We then check to make sure that cpu count exists and cpu owner exists. We then check to make sure that the cpu owner is not the same as the current thread.

The next while loop checks to see if there are threads waiting for the cpu. It then says that that this thread is no longer accessing the cpu and the unlocks the thread and runs an interrupt and then locks the thread again.

If there are sleeping threads it then moves to those threads.

Syscalls.c

Run_syscall

Through this we get to the idea of a system call table which is an array of function pointerswhich point to functions that implement system calls. My hypothesis is that the return value is the value of the sys call.

New host task

We first declare a process id and then make sure we the host process (?) we then get a process id from the kernel thread if it is an error i.e - we return the id. We then lock our current thread and then set our new task to the pid that was returned from before. We then unlock and update a counter saying there is another host task

THIS USES JUMP BUF SET should go over
Delete host task

We create a task and get some info on the thread we are trying to run we save this info to a different process and then we delete the thread through a jump buf

lkl_syscall

We make a task equal to what the host is doing(?) we then get the return value of the cpi thread. If there is no task we use this as a task and then we set the task as the task.

Switch host task I think is just switching to kernel mode for that task

@M1cha Unless I am missing something, the way setjmp/longjmp is used should be safe, we always do it in the same thread. The idea is to return early from __switch_to and skip running the schedule tail code. So the stack context should be always the safe and we should never jump to uninitialized stack.

https://github.com/lkl/linux/pull/246