Module 1-2
Linux Kernel Services

Delay Execution

- You can measure time lapses and current time by:
  - Knowing the current time
  - Delaying execution of a process for a specified amount of time

- ***Jiffies*** variable (declared as volatile unsigned long) counts number of ticks since boot. Volatile ensures that jiffies, that is updated by the timer interrupt handler every tick, is reread. Volatile avoids compiler memory reads optimization.

- Number of ticks can be 100 or 1000 per seconds and saved as HZ macro, represents clock frequency. It means if HZ value is 100, a jiffy is a 10 milliseconds duration.

- When delaying execution better approach is to sleep-wait (blocking), instead of busy-wait (spinning)
  - **ndelay(),udelay(), mdelay() - busy-wait, use it when cannot sleep**
  - **msleep(), msleep_interruptable(), ssleep() - sleep-wait**

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;
j = jiffies;                          /* read the current value */
stamp_1 = j + HZ;                     /* 1 second in the future */
stamp_half = j + HZ/2;                /* half a second */
stamp_n = j + n * HZ / 1000;          /* n milliseconds */
```

Kernel Timers

- Linux provides ***timers*** for executing function at future time without blocking the current process
- There is also High-resolution timers (hrtimers) available in Linux that provides a high-precision framework. Timers operate on the granularity of jiffies, whereas hrtimers operate at the granularity of nanoseconds.
- That means timer functions are run asynchronously as compared to Delay functions. Delay functions run in the context of process executing.
- Timer function runs on the CPU where it is submitted
- Timer function runs in the context of "software interrupt" (softirq). That means your function code is subject to *"interrupt context"* restrictions:
  - No user space access
  - No sleeping
  - No kmalloc(.., GFP_KERNEL)
  - No ioctl
  - No semaphores etc..

How to use Kernel Timers

- Timer is initialized using *init_timer*() or *DEFINE_TIMER*() dynamically or statically respectively
- Populate a ***timer_list*** structure with the address of your function and arguments that will be called at time expiration.
- Register the timer using ***add_timer*()**
- Requires reinstalling if you want to run it periodically.
  - That means timers in Linux are NOT ***interval*** timers. Timer functions that re-registers itself always runs on the same cpu to achieve better cache locality.
- Change the expiration time of pending timer by using ***mod_timer*()**
- ***del_timer*()** to cancel timer. *del_timer_sync*() should be run on SMP systems to guarantee that timer function is not running on any cpu – avoids race condition
- Data structures accesses by your timer function should be protected from concurrent access.
- ***timer_pending*()** to check if any timer is pending.

Kernel Timers
Example

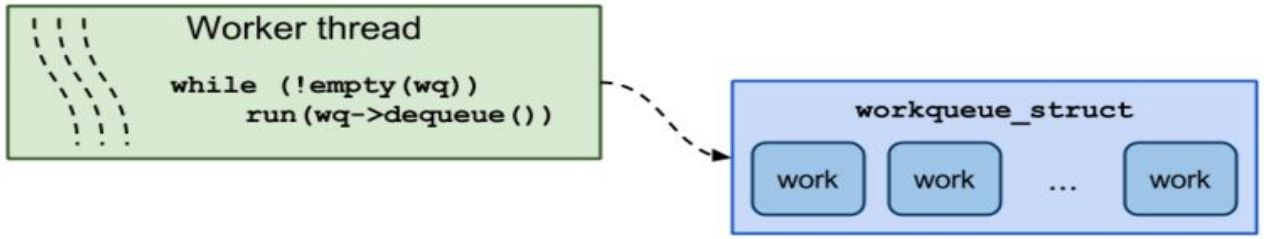
```
#define INITIAL_DELAY    HZ/2
static unsigned long delay = INITIAL_DELAY;
/*my timer function to call at expiration */
static void my_timer_func( unsigned long data) {
   unsigned long new_delay = data;
   my_timer.expires = jiffies + new_delay;
   printk("helloplus: timer %lu%lu\n", jiffies, my_timer.expires);
   add_timer(&my_timer);                    /* re-register timer function to act as a periodic timer */
 }
/* Set up the timer the first time in driver's init routine */
struct timer_list my_timer;
init_timer(&my_timer);
my_timer.function = my_timer_func;
my_timer.data = delay;
my_timer.expires = jiffies + delay;
add_timer(&my_timer);

/* Deleting the pending timer and reinstalling it with new delay*/
del_timer_sync(&my_timer);
my_timer.function = my_timer_func;
delay = HZ;  /* every second */
my_timer.data = delay;
my_timer.expires = jiffies + delay;
add_timer(&my_timer);
```

Kernel Threads

- Kernel threads in Linux kernel perform background tasks
- A kernel thread is a process that exists only in kernel space and have access to kernel data structures but no access to user space
- Linux represents kernel thread and process using task_struct and both are scheduled the same way.
- Driver can attain some parallelism by spawning kernel threads to perform background task
- Device drivers utilizes services of kernel threads. Few examples of kernel threads are listed below:
  - khubd – Monitors USB events and configures the hotplug device
  - ksoftirqd/n – implement softirq, bottom half interrupt processing.
  - events/n – implement shared work queue. Way to deferred work

Work Queues

- Work queues mechanism is a convenient way of creating kernel threads to perform low priority or other tasks on driver behalf. Commonly used as “Bottom half” for interrupt processing
- A work queue contains a link list of functions (represented by work structure) which needs to be run as soon as possible.
- Work queue (*struct workqueue_struct*) is a structure onto which the work is placed. Where work is represented by *work_struct*, which identifies the work to be deferred and the deferred function to use.
- The event/n kernel threads (one per cpu) extract work from the work queue and executes the function provided in the *work_struct*
- Whenever, the driver needs to perform any task using a worker thread:
  - It creates a work function (*work_struct*)
  - It populates it with the address and arguments of work function and link it to the work queue.
  - Driver can do other task while worker thread performs the task on behalf of the driver
- Driver can create its own work queue to be serviced by single or multiple dedicated threads
- Driver can also use a default work queue (one per-CPU worker threads) shared by all kernel modules. ***event/n*** is the name of the kernel thread that implements shared work queue.



Work queue API

```
struct workqueue_struct *create_workqueue( name );
void destroy_workqueue( struct workqueue_struct * );
```

```
INIT_WORK( work, func );
INIT_DELAYED_WORK( work, func );
```

```
INIT_DELAYED_WORK_DEFERRABLE( work, func );
```

```
int queue_work( struct workqueue_struct *wq, struct work_struct *work );
int queue_work_on( int cpu, struct workqueue_struct *wq, struct work_struct *work );

int queue_delayed_work( struct workqueue_struct *wq,
                        struct delayed_work *dwork, unsigned long delay );

int queue_delayed_work_on( int cpu, struct workqueue_struct *wq,
                        struct delayed work *dwork, unsigned long delay );
```

```
int schedule_work( struct work_struct *work );
int schedule_work_on( int cpu, struct work_struct *work );

int scheduled_delayed_work( struct delayed_work *dwork, unsigned long delay );
int scheduled_delayed_work_on(
                int cpu, struct delayed work *dwork, unsigned long delay );
```

```
int flush_work( struct work_struct *work );
int flush_workqueue( struct workqueue_struct *wq );
void flush_scheduled_work( void );
```

```
work_pending( work );
delayed_work_pending( work );
```

```
int cancel_work_sync( struct work_struct *work );
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Work Queue - example

```
#include <linux/workqueue.h>
/* declare a work queue that is serviced by kernel thread */
struct workqueue_struct *wq;

wq = create_singlethread_workqueue("mydrv");

static void work_func (struct work_struct *work){
      printk( " work function is called, %d\n");
}
/* Declare and allocate memory for work element */
  struct work_struct *mywork;
  mywork = kmalloc(sizeof(struct work_struct), GPF_KERNEL);

/* Populate the work element with our function (work_func) and arg */
 INIT_WORK(mywork, work_func, arg);

/* Submit work to dedicated work queue from driver's entry point. To submit work to default shared
queue, use  schedule_work() instead */

queue_work(wq, mywork);
```

Work function is called with work struct as a parameter. Work struct can also be placed inside a implementation specific structure, and then use the container_of() macro or type casting.

Interrupts

• Interrupts provide a way for a device to gain CPU's attention

• Devices require a prompt response from the CPU when various events occur, even when the CPU is busy running application or kernel code

• Forcibly change normal flow of control. Similar to context switch (but lighter weight)

– Hardware saves some context info on the stack; Includes interrupted instruction if restart needed

– Kernel calls interrupt handler routine to handle the interrupt

– Execution resumes with special "**iret"** x86 cpu instruction

Types of Interrupts

- **Asynchronous:** caused by external source, such as I/O device. Not related to instruction being executed
- **Synchronous** (also called exceptions): *Processor detected exception*
  - Faults: correctable; offending instruction is retried
  - Traps: often for debugging; instruction is not retried
  - Abort: major error (hardware failure)

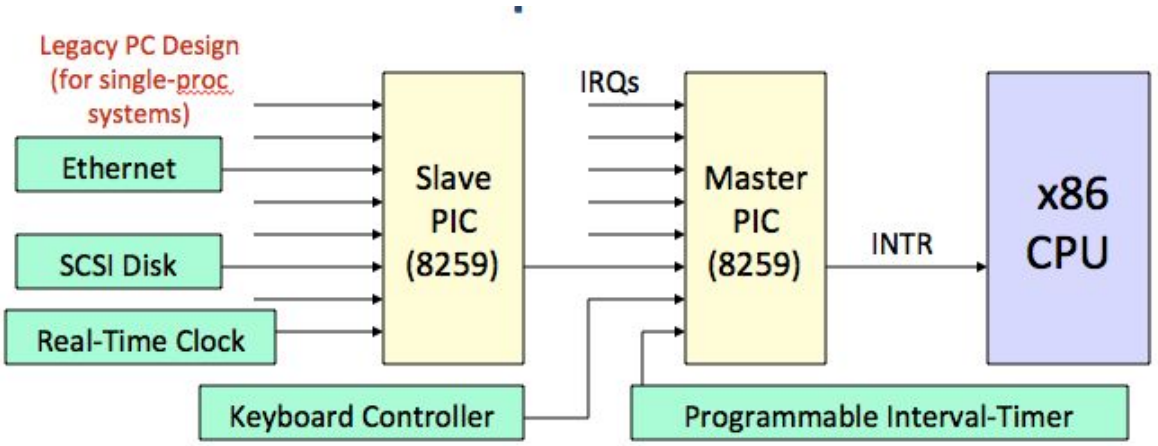
- **Programmed:** Requests for kernel intervention (softirq or software interrupts, syscalls)

IO Interrupt handling

Hardware devices capable of issuing interrupt requests has a single output line designated as the Interrupt ReQuest line (IRQ) connected to APIC (Advanced Programmable Interrupt Controller). APIC assigns each device a unique IRQ number.

- APIC is responsible for notifying CPU when a specific external device needs attention.
- To identify the interrupting device, APIC translates IRQ number assigned to device in to vector number
- APIC raises CPU interrupt until cpu acknowledges it

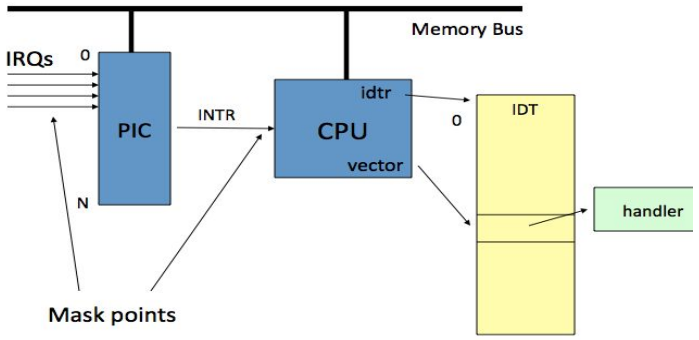
It is possible to "mask" or disable interrupts at PIC or individual CPU level



Dispatching Interrupts

On Intel processors, Interrupt and trap gate descriptors are used for handling device interrupts and exceptions (divide by zero, page faults). Both are identical and are called Interrupt descriptors

- Interrupt descriptors are stored in ***Interrupt Descriptor Table (IDT)***. ***idtr*** register has address of IDT table in memory
- Each interrupt is assigned a number between 0-255, called ***vector***, which CPU uses as an index into the IDT table.
- IDT has gate descriptors for each interrupt vector. Hardware locates the proper gate descriptor for this interrupt vector, and locates the new context
- For new context, a new stack pointer, program counter, CPU and memory state, etc., are loaded. The old program counter, stack pointer, CPU and memory state, etc., are saved.
- Global interrupt mask is set. A specific device handler is invoked. Each interrupt has to be handled by a special device specific or trap-specific routine, called ISR (interrupt service routine)



Vectors and IRQs relationship

Vector: It is an index (0-255) into interrupt descriptor table. Vector are usually IRQ# + 32

- Vector 128 used for syscall
- Vectors 251-255 used for IPI (Inter-Processor-Intr)

Below 32 reserved for non-maskable intr & exceptions. Maskable interrupts can be assigned as needed

Interrupt Masking

Interrupts can be masked at:

- Global - delays all interrupts
- per IRQ - individual IRQs can be masked selectively (preferred).

Selective masking is usually what's needed as it is common to have interference from interrupts of the same type

Finding an Interrupt Handler

Multiple I/O devices can share a single IRQ and hence interrupt vector.

- First differentiator is the interrupt vector
- Multiple interrupt service routines (ISR) can be associated with a vector
- ISR determines whether interrupt is generated by its device by testing the argument passed to the ISR routine. Argument is normally points to some private data structure of the driver.

Interrupt Stack

When an interrupt occurs due to:

- Exceptions: The kernel stack of the current process is used. There's always some process running. "idle" process, if nothing else to run.
- Interrupts:
  - Hard IRQ: hardware interrupts use dedicated per processor (cpu) stack. There is one stack per cpu
  - Soft IRQs: Soft interrupt (softirq) generated by timers, tasklets, softirq. There is one stack per cpu

These stacks are configured in the IDT and TSS at boot time by the kernel

Registering Interrupt Handler

Once the IRQ number is determined, driver can call **request_irq()** and pass its IRQ number and the handler, called Interrupt Service Routine (ISR), to be called when the IRQ is raised. Interrupt Handlers are registered via:

```
request_irq(irq, handler, flags, name, dev_id)
```

– **irq** is the interrupt vector number assigned during pci enumeration, as part of pci_dev structure passed to pci driver's probe routine.

– handler is a pointer to device interrupt handler routine (ISR).

– **flags** is a bit mask of options:

- *IRQF_SHARED – IRQ can be shared. True for PCI devices*
- *IRQF_DISABLED – fast handler (Interrupts are disabled on local cpu)*
- *IRQ_TRIGGER_RISING/IRQ_TRIGGER_HIGH – For edge and level sensitive devices*

– **name** is used to identify this device in /proc/interrupts

– **dev_id** typically, a pointer to some internal data structure of the driver. NULL can be used

Device drivers usually request the IRQ when the device is first opened and free it on last close.

Registering Interrupt Handler

- Kernel thread can be scheduled to perform bottom half instead of explicit call to schedule a tasklets/softirq or kernel threads (work queues) when registering interrupt handler
- This can be done by specifying a third argument when registering interrupt handler using threaded version:

```
int request_threaded_irq ( unsigned int inq, irq_handler_t handler,
irq_handler_t thread_fn, unsigned long flags, const char *name, void *dev);
```

- ISR can return *IRQ_WAKE_THREAD*, if the thread function needs to be invoked (bottom half) as a deferred processing. That will also enable interrupts. If no bottom half work is required, *IRQ_HANDLED* can be returned.

Top Half - Interrupt Service Routine (ISR)

- ISR routine is also called "**TOP Half**" of interrupt processing.
- ISR should be efficient and perform minimal amount of work required considering the same and lower interrupts are masked while ISR is running.
- Consider delegating less critical and heavy processing to "Bottom Half" such as: Kernel threads, softirq, tasklets.
- ISR routine runs in interrupt context and thus no user or kernel thread can run during that time, that may result in higher latencies, if not careful.

Device driver registers ISR routine using the prototype:

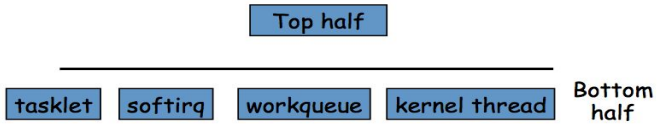
```
irqreturn_t handler(int irq, void *dev_id)
```

- **irq** is the interrupt number. Used only when handler can service more than one intr.
- **dev_id** data is the pointer to some internal data structure given during registration
- **irqreturn_t** is a special return type. Only legitimate values:
  - *IRQ_HANDLED on success*
  - *IRQ_NONE otherwise*
  - *IRQ_WAKE_THREAD - Scheduled a kernel thread for bottom half processing*

Return value is also used by the kernel to detect spurious interrupts. An interrupt handler is removed from a given interrupt line via free_irq(irq, dev_id). It is recommended to free the IRQ when the device is closed, if possible. Do not wait until driver module is unloaded

Bottom Half

- To handle work scheduled by Top Half. Also called deferred processing.
- Runs in interrupt context (except workqueues) but with hardware interrupts enabled to help avoid delay in interrupt processing.
- Typically, perform interrupt-related work deferred by ISR routine.
- Bottom half processing can be performed using:
  - Softirqs:
    - Statically defined. Requires kernel rebuild to add new softirq.
    - Most efficient as can be run simultaneously on all available processors. code must be reentrant and do locking.
    - Kernel checks it if softirq is pending (flag is set) when returning from ISR, syscalls, exceptions. Thus Low dispatch latency
    - Bound to the cpu where it was submitted
  - Tasklets:
    - Dynamically created.
    - Built on top of Sofirqs but with restricted execution environment.
    - Unlike softirq, same tasklet cannot be run on multiple processor concurrently.
    - Simple to use as compared to softirq as it avoids synchronization issues
  - Workqueues: Kernel threads are scheduled to perform deferred processing. More scalable, as kernel can schedule them to any available cpu.



Softirqs (example)

- To define a new softirq, you have to statically add an entry to include/linux/interrupt.h and compile the kernel. Not recommended!
- If you must, then add an entry for new softirq, ***dev_soft_irq,*** *for example,* in the enum list provided in kernel: *include/linux/interrupt.h*. Build kernel
- Initialize the softirq, ***dev_soft_irq***, in driver init routine:

```
void __init dev_init(){
..
open_softirq(dev_soft_irq, checksum_packet,NULL)  // register dev_soft_irq() as softirq
}
```

- Write a checksum_packet function that will be executed in the context of softirq

```
void checksum_packet(){
/* cpu intensive work */
}
```

- In driver ISR routine, invoke raise_softirq(dev_soft_irq) to perform deferred processing

```
static irqreturn_t dev_interrupt(int irq, void * dev_id){
/* perform minimum required processing and then mark softirq pending */
rasie_softirq(dev_soft_irq);
return IRQ_HANDLED
}
```

Tasklet (example)

Tasklets can be initialized dynamically.

```
void __init dev_init(){
  ..
   tasklet_init(&dev_struct->tsklt, checksum_packet, dev); // register checksum_packet() as tasklet
}
```

- Write a checksum_packet function that will be executed in the context of softirq

```
void checksum_packet(){
   /* cpu intensive work */
}
```

- In driver ISR routine, invoke tasket_schedule to perform deferred processing

```
static irqreturn_t dev_interrupt(int irq, void * dev_id){
/* perform minimum required processing and then mark schedule tasklet */
tasklet_schedule(&dev_struct->tsklt);
return IRQ_HANDLED;
}
```

Comparison

	Softirq	Taskets	Work Queues***
Execution Context	Interrupt	Interrupt	Process
Reentrancy	Can run simultaneously on different cpus	Different cpus can run different tasklets but not the same one	Can run simultaneously on different cpus
Sleep Semantics	cannot sleep	cannot sleep	sleep
Ease of use	Not easy to use	easy to use	easy to use
When to use	If deferred work cannot sleep and has high scalability and performance requirements	If deferred work cannot go to sleep	If deferred work can go to sleep

Common Kernel Data Structures and API

Understanding of common kernel data structure and API will help develop efficient and bug free code. Before building your own data structures, see if generic one provided by kernel can be used:

Generic Linked Lists

- Singly, doubly, circular linked lists.
- Linked List API: https://kernelnewbies.org/FAQ/LinkedLists
- Example: http://www.roman10.net/2011/07/28/linux-kernel-programminglinked-list/

RB-trees

- Semi-balanced, binary search tree offers order "log(n)" search, insert, and delete operations.
- RB-Tree API: https://lwn.net/Articles/184495/
- Examples: Linux Memory Management Subsystem uses RB-Trees to track process memory regions (start address serves as a key). CFS scheduler uses it to track process virtual runtime (runtime serve as a key)

Radix tree

- Provide a mapping from a number (unsigned long) to an arbitrary pointer (void *) though radix-tree. Allows up to two "tags" to be stored with each entry.
- Radix Tree API: https://lwn.net/Articles/175432/
- Example: File system cache (page cache) is implemented using Radix Tree.

Generic Hash Tables

- Kernel hash table implementation to hlist buckets to build simple hash tables
- HashTable API: https://kernelnewbies.org/FAQ/Hashtables

Compiling Kernel Module

To compile a kernel module outside of kernel build, you need a Makefile as simple as this one:

```
# To build modules outside of the kernel tree, we run "make"  in the kernel source tree; the
# Makefile these then includes this Makefile once again.
#This conditional selects whether we are being included from the  kernel Makefile or not.
ifeq ($(KERNELRELEASE),)
# Assume the source tree is where the running kernel was built .
# You should set KERNELDIR in the environment if it's elsewhere
        KERNELDIR ?= /lib/modules/$(shell uname -r)/build
        # The current directory is passed to sub-makes as argument
        PWD := $(shell pwd)
modules:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
modules_install:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
clean:
        rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
.PHONY: modules modules_install clean
else
        # called from kernel build system: just declare what our modules are
        obj-m := my_driver.o
endif
```

Few GuideLines

- Driver file operation methods should return error code (errno) for application to handle it properly:
  - Returning **0** means pass
  - Returning **-1** means failed. Failure should accompany with standard error code provided in Linux kernel header files ***errno.h***. There are set of helper routines ***ERR_PTR***, ***IS_ERR*** and ***PTR_ERR*** available to test the error condition and print the errno.
- Driver shares kernel stack that is small. Do not declare large automatic variables, structures on the stack. Allocate dynamically instead
- Use caution when using kernel routines with (__). These routines are low level part of kernel interface. Consider higher level interface - more portable
- No floating pointing arithmetic
- Use ***goto*** statement for dealing with serious failure condition, such as failing to allocate memory, or registering to some kernel subsystem. It is less repeated, readable and keep the error handling code out of the way. Commonly used across all the kernel code.
- *gcc* compiler offers directive ***likely/unlikely*** to pick the fall though path code.
- Driver/module can check if the code is executing in interrupt context by calling the ***in_interrupt( )***, which takes no parameters and returns nonzero if the code is currently running in interrupt context, either hardware or software interrupt.
- Device drivers should never try add a "*syscall*". It is decided by Linux maintainers and involve assembly and modifying software interrupt.
- Do not mix formats or do type conversion in kernel space

Kernel Build Options For Driver Development

Development kernel should have debug options enabled to help find bugs. These debug options are found in "*kernel hacking*".

–**CONFIG_DEBUG_KERNEL**: Makes other debugging options available; it should be turned ON before you enable other debug options.

–**CONFIG_DEBUG_INFO:** Compile kernel with debug info. You can use "*objdump –d –l vmlinux*" to see the source-to-object code mapping.

–**CONFIG_DEBUG_SLAB**: Initialize buffer with hex value "**0xa5**" when buffer is allocated. Set to "**0x6b**" when the buffer is freed. This helps isolating memory overruns or attempt to access uninitialized memory due to dangling pointer. There is also byte of extra padding provided to each buffer allocated by the driver. If the extra byte of padding is ever touched by the driver, kernel may cause oops or panic with kernel stack reporting the offending code.

–**CONFIG_DEBUG_SPINLOCK:** kernel catches operations on uninitialized spinlocks and various other errors (such as unlocking a lock twice).

–**CONFIG_DEBUG_SPINLOCK_SLEEP**: checks for attempts to sleep while holding a spinlock. In fact, it complains if you call a function that could potentially sleep, even if the call in question would not sleep.

–**CONFIG_MAGIC_SYSRQ**: Enables the "***magic SysRq***" key, helps with breaking into a hung system

–**CONFIG_DEBUG_STACKOVERFLOW/CONFIG_DEBUG_STACK_USAGE**: Track down kernel stack overflows. The first option adds explicit overflow checks and second monitors stack usage and keep statistics that can be dumped using magic SysRq key.

Kernel Build Options For Driver Development

–**CONFIG_PRINTK_TIME**: Adds timing instrumentation to printk() output, so you can use printk as checkpoints for measuring execution times and identifying slow code regions.

–**CONFIG_DETECT_SOFTLOCKUP**: is defined as bug that cause the cpu to loop in kernel mode for more than 20 seconds without giving other tasks a chance to run. Kernel will print the current stack trace , but the system will stay locked up. Setting *kernel.softlockup_panic* tunable will result in system to panic. softlockup detection can be disabled by setting: *kernel.softwatchdog=0*

–**CONFIG_NMI_WATCHDOG:** is defined as a bug that causes the CPU to loop in kernel mode for more than 10 seconds without letting other interrupts have a chance to run. Kernel will print the current stack trace, but the system will stay locked. Setting *kernel.'hardlockup_panic* will result in system to panic. Hard lockups" Both soft and hard lockups are detected via watchdogs mechanism. hard lockup detection can be disabled by setting: *kernel.nmi_watchdog=0*

https://www.kernel.org/doc/Documentation/lockup-watchdogs.txt

–**CONFIG_DEBUG_BUGVERBOSE**: Produces extra debug information when any kernel code invokes BUG(), assuming that you have CONFIG_BUG turned on during kernel configuration

– **CONFIG_KPROBES:** Allows dynamic inserting probes into running kernel and kernel modules. Under "Instrumentation Support", look for "Kprobes" and set to "y"

– **CONFIG_KALLSYMS_ALL**: kprobe address resolution code uses it.

– **CONFIG_DEBUG_PAGEALLOC**: Helps isolate memory corruption issues. This option may slow down the system.

– **CONFIG_DEBUG_LIST**: Help debug issues related to linked list insertion/deletions

– **CONFIG_SLUB_DEBUG:** Performs number of checks on allocated and freed objects. Learn more about Slub Allocator: Documentation/vm/slub.tx

Kernel Build Options For Driver Development

– **CONFIG_DEBUG_RODATA:** Makes kernel text and data segment read-only, cause a page fault if try writing to it.

– **CONFIG_DEBUG_MUTEXES:**check mutex semantics violations and mutex related deadlocks

– **CONFIG_DEBUG_LOCK_ALLOC:** Detect incorrect freeing of live locks

– **CONFIG_PROVE_LOCKING:** This feature enables the kernel to report locking related deadlock events before they happen. *Documentation/locking/lockdep-design.txt.*

– **CONFIG_LOCKDEP:** detects locking and deadlock issues. *perf lock* requires it to be enabled.

– **CONFIG_LATENCYTOP:** Report system latencies resulted from reading file, waiting on a socket

– **CONFIG_DETECT_HUNG_TASK:** When a hung task is detected, the kernel will print the current stack trace, but the task will stay in uninterruptible state. If lockdep is enabled then all held locks will also be reported. This feature has negligible overhead. Helpful in figuring out what's causing kernel freeze

– **CONFIG_LOG_BUF_SHIFT=22**: sets the kernel buffer log size 2^22 (4 MB). Useful when doing heavy printk logging.

– **CONFIG_PRINTK_TIME:** add timestamps to printk messages dumped into dmesg

MISC

What is Notifier Chains

Linux uses a notifier chain, that simply a list of functions that are executed when an event of interest occurs. These notifier chains work in a publish-subscribe model.

- subscriber, kernel module, that requires notification of a certain event, registers itself with the publisher, low level kernel module/driver. The publisher informs the subscriber whenever an event of interest occurs by executing the subscriber registered function

Notifier Chains Purpose:

- Kernel modules runs in the same kernel space and are interconnected to one another. An event captured by a certain module might be of interest to another module. Events like:
  - When a device is plugged, a plugin notification is sent
  - A die notification is sent when a kernel function triggers a trap/fault (caused by oops, page fault or a breakpoint, tracepoint)
  - A notification about change in cpu frequency is generated when transition in processor frequency
  - Address notification is sent when change in IP address of network card is detected or network card status is changed up/down.
- Subscriber initialize the data structure ***notifier_block*** (include/linux/notifier.h) that contains a pointer to function to be called when an event occurs.
- Publisher (notification module) maintains a list head to manage and traverse the *notifier_block* list. A subscriber function is added to the head of list by calling *xxx_notifier_chain_register*() and deleted by calling *xxx_notifier_chain_unregister*()
- When an event occurs, then *xxx_notifier_call_chain* API is used to traverse a particular list and service the subscriber.

Types of Notifier Chains

Notifier chains are classified according to the execution context, locking requirements of the Notifier chain:

- **Atomic notifier chains:**
  - Function runs in interrupt or atomic context (non-blockable).
  - Linux modules can use it be notified about watchdog timers or high priority message handling. For example, KeyBoard driver calls *register_keyboard_notifier that* uses atomic_notifier_chain_register to get called back on keyboard events.

- **Blocking notifier chains:**
  - Function runs in the process context (blocking)
  - Linux modules can use it for notifications that are not time critical. For example, USB driver uses *usb_register_notify* to get notification when either USB devices or buses being added or removed.

- **Raw notifier chains:**
  - A raw notifier chain does not manage the locking and protection of the callers. Also, there are no restrictions on callbacks, registration, or de-registration.
  - It provides flexibility to user to have individual lock and protection mechanisms. Linux uses the raw notifier chain in low-level events. For example, *register_cpu_notifier* uses it to pass on CPU going up/down information.

- **SRCU notifier chains:**
  - Sleepable Read Copy Update (SRCU) notifier chains, similar to the blocking notifier chain and run in the process context.
  - It differs in the way it handles locking and protection.
  - Use when calling notifier call often and there's very little requirement for removing from the chain. For example, Linux module uses it in CPU frequency handling.

Notifier Chains (Example)

- Driver interested in receiving USB hotplug events first initialize the *notifier_block* struct to register the function to be called by the publisher. In this case the notifier (publisher) is a USB core, that is of type *blocking notifier chain*

```
static struct notifier_block usbdev_nb = {
    .notifier_call =        usbdev_notify,    // function to call
};
```

- Notifier function should have the prototypes with parameters: pointer to notifier_block, notifier specific events, and data specific to the event. In USB case, event can be of type: USB_DEVICE_ADD, USB_DEVICE_REMOVE.. and data is the USB device (dev)

```
static int usbdev_notify(struct notifier_block *self, unsigned long action, void *dev)
{
    switch (action) {
    case USB_DEVICE_ADD:
      break;
    case USB_DEVICE_REMOVE:
      usbdev_remove(dev);
      break;
    }
    return NOTIFY_OK;
}
```

- In driver init function, call *usb_register_notify(&usbdev_nb) and in cleanup routine call usb_unregister_notify(&usbdev_nb)*

```
int __init usb_devio_init(void)
{...
 usb_register_notify(&usbdev_nb);
}
init usb_devio_cleanup(void)
{..
 usb_unregister_notify(&usbdev_nb);
}
```

Notifier Chains (Example)

- Driver interested in customizing *oops* message can subscribe to *die* notifier, that is of type *atomic notifier chain*.

```
 static struct notifier_block oops_nb = {
     .notifier_call =        oops_notify,    // function to call
};
```

- die notifier parameters are event type

```
 static int oops_notify(struct notifier_block *self, unsigned long event, void *data){
   struct die_args *args = (struct die_args *) data;
   if (event ==1) {            /* '1' corresponds to an OOPS event in die notifier chain*/
     printk("oops_notify is called: OOPS! at EIP=%lx\n", args->regs-eip);  // print EIP register value when trap is taken
   return 0;
 }
```

- In driver init function, call *oops_notify(&oops_nb) and in cleanup routine call oops_notify(&oops_nb) to register/unregister with die notifier chain*

```
int __init oops_init(void)
{...
 register_die_notifier(&oops_nb);
}
init oops_cleanup(void)
{..
 unregister_die_notifier(&oops_nb);
}
```