Tasks and concurrency

KThreads

kthread.h

```
// Create and start a kernel thread
struct task_struct *kthread_run(int (*threadfn)(void *data)
  , void *data, const char namefmt[], ...);

// Stop a kernel thread (to be invoked by the main thread
int kthread_stop(struct task_struct *k);
// Check if a thread should stop
bool kthread_should_stop(void);
```

Spinlocks

spinlock.h

```
DEFINE_SPINLOCK(<name>);
spin_lock(spinlock_t *lock);
spin_unlock(spinlock_t *lock);

/* Spins until lock free, then
   acquires it and disables interrupts
   saving their current state in flags */
spin_lock_irqsave(spinlock_t *lock,
  unsigned long flags);
/* Unlock, and restore interrupt flags */
spin_unlock_irqrestore(spinlock_t *lock,
  unsigned long flags);

/* read write spinlocks */
DEFINE_RWLOCK(<name>);
void read_lock(rwlock_t *lock);
void read_unlock(rwlock_t *lock);
void write_lock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
/* *_irqsave, *_irqrestore variants do exist as well. */
```

Waitqueues

wait.h

```
// declares and inits wait queue head <name>
DECLARE_WAIT_QUEUE_HEAD(<name>);
// or initialise it
init_waitqueue_head(*wq_head);

// Wait for a condition to be true, interruptible
wait_event_interruptible(*wq_head, condition);

// As above, sleep until a condition gets true.
// The condition is checked under the lock. This is expected
// to be called with the lock taken. And will return with lock
// taken
wait_event_interruptible_lock_irq(*wq_head, condition, lock)

// Wakes all non-exclusive waiters from the wait queue
// that are in an interruptible sleep state and
// just one in interruptible exclusive state. This must be called
// whenever variables that supposedly impact the "condition"
// are changed.
void wake_up_interruptible(*wq_head);
```

RCU

rcupdate.h

```
void rcu_read_lock(void);
void rcu_read_unlock(void);
// Wait for all pre-existing RCU read-side
// critical sections
void synchronize_rcu(void);
// Call a function after all pre-existing RCU
// read-side critical sections
void call_rcu(struct rcu_head *head, rcu_callback_t func);
// Assign v to p
void rcu_assign_pointer(void *p, void *v);
// Access data protected by RCU
void *rcu_dereference(void *p);
```

Atomic variables && bitops

atomic.h

```
void atomic_set(atomic_t *v, int i);
int  atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int  atomic_inc_and_test(atomic_t *v);
int  atomic_dec_and_test(atomic_t *v);
int  atomic_cmpxchg(atomic_t *v,  int old, int new);

void set_bit(int nr,  volatile unsigned long *addr);
void clear_bit(int nr,  volatile unsigned long *addr);
```

Per CPU variables

percpu-defs.h

```
// Declare a per-CPU variable
DEFINE_PER_CPU(type, name);

// Access per-CPU variable for the current CPU,
// enter preempt disabled section. Note that
// name is an L-value.
get_cpu_var(name);

// Exit preempt disabled section
void put_cpu_var(name);
```

IO

Character devices

fs.h

```
struct file_operations {
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)
    ssize_t (*write) (struct file *, const char __user *, size_t, lo
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
}
// Register a chrdev device driver, if major=0, returns assigned maj
int register_chrdev(unsigned int major, const char *name,
                  const struct file_operations *fops);
void unregister_chrdev(unsigned int major, const char *name)
// Get minor from the device file pointer
minor = iminor(file→f_path.dentry→d_inode);
```

Port based IO

ioport.h

```
struct resource * request_region( unsigned long first,
       unsigned long n,  const char *name);
void release_region(unsigned long start, unsigned long n);
unsigned in[b,w,l](int port); /* read one,two and four bytes */
void out[b,w,l](unsigned char byte, int port) /* write */
/* Ports can use ioread and iowrite (below) but
   you must map those in memory with ioport_map */
void *ioport_map(unsigned long port, unsigned int count);
void ioport_unmap(void *addr);
```

Memory mapped IO

ioport.h

```
struct resource *request_mem_region(unsigned long start,
  unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long len);
/* For memory mapped IO you must obtain a virtual address in
   kernel space before any access */
void *ioremap(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
unsigned int ioread[8,16,32](void *addr);
void iowrite[8,16,32](u8 value, void *addr);
```

Interrupts

interrupt.h

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
int request_irq(unsigned int irq_no, irq_handler_t handler,
  unsigned long flags, const char *dev_name, void *dev_id);
void free_irq(unsigned int irq_no, void *dev_id);
```

Tasklets

interrupt.h

```
// the callback type
void (*callback)(struct tasklet_struct *t);
// declare a tasklet
DECLARE_TASKLET(name, callback);
// schedule a tasklet; essentially puts the tasklet
// in a softirq queue, if it is not already scheduled.
void tasklet_schedule(struct tasklet_struct *t);
```

Workqueues

workqueue.h

```
// Declare a work data structure variable called name
DECLARE_WORK(name, void (*func)(struct work_struct *work));
// Schedule work on the system workqueue. Return 0 if
// the same work is already scheduled.
bool schedule_work(struct work_struct *work);
```

Timers

hrtimer.h

```
// Initializes a high-resolution timer.
// clock_id can be:
// - CLOCK_MONOTONIC: always moves forward without regression
// - CLOCK_REALTIME: current time of day, subject to regression
//
// mode can be:
// - HRTIMER_MODE_ABS (absolute, e.g., at 12:00AM)
// - HRTIMER_MODE_REL (relative, e.g., 10 seconds from now)
//
void hrtimer_init(struct hrtimer *timer,
   clockid_t clock_id, enum hrtimer_mode mode);
// After initializing the timer, you need to register the
// callback function that will be called when the timer
// expires. You do it by assigning the function field of
// the timer to the callback function.
timer->function = <your_callback_function>;
// Starts a high-resolution timer. You need to specify
// the timer to start (timer), the time at which the timer
// should expire (time), and the mode (mode).
void hrtimer_start(struct hrtimer *timer, ktime_t time,
  const enum hrtimer_mode mode);
// Converts fro/to number of milli/nanoseconds to/fro a ktime_t value.
ktime_t (ns/ms)_to_ktime(u64 ns);
u64 ktime_to_(ns/ms)(u64 ns/ms)
ktime_t ktime_set(s64 secs, unsigned long ns)
// Get current time measured relative to
// the UNIX epoch starting in 1970.
ktime_t ktime_get_real()
// Moves the timer forward by the specified interval
// (in any case is relative).
ktime_t hrtimer_forward_now(struct hrtimer *timer,
ktime_t interval);
// Cancels a timer.
int hrtimer_cancel(struct hrtimer *timer);
```

Misc

Memory management

slab.h

```
// Flags gfp_t:
//  - `%GFP_KERNEL` - May sleep.
//  - `%GFP_NOWAIT` - Will not sleep.
// Allocate contiguous memory for an object of size `size`
void *kmalloc(size_t size, gfp_t flags);
void kfree(void *ptr);
// Allocate non-contiguous memory`
void *vmalloc(unsigned long size);
void vfree(void *addr);
// kmem_cache_create - create a new cache.
// Use NULL for ctor, flags and align if unsure
struct kmem_cache *kmem_cache_create(const char *name,
  size_t size, size_t align, unsigned long flags,
  void (*ctor)(void *));
// or
 KMEM_CACHE(my_object_type, flags);
//kmem_cache_destroy - destroy a cache
void kmem_cache_destroy(struct kmem_cache *cachep);
// kmem_cache_alloc - allocate an object from a cache
void *kmem_cache_alloc(struct kmem_cache *cachep,
    gfp_t flags);
// kmem_cache_free - free an object to a cache
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

File management

```
// In kernel file operations
// flags: O_CREAT, O_APPEND, O_WRONLY, O_RDONLY`
struct file *filp_open(const char *, int flags, umode_t mode);
// just use id=NULL to close
int filp_close(struct file *, fl_owner_t id);
ssize_t kernel_read(struct file *, void *, size_t, loff_t *);
ssize_t kernel_write(struct file *, const void *, size_t, loff_t *);
```

Containers

container_of.h

```
//container_of - derive a pointer to the containing structure
// given a pointer to a member
container_of(member_ptr, container_type, member_field_name)
// Declare a list head called 'head'
static LIST_HEAD(head);
// Define your custom list_element structure
// which must have a field of type `struct list_head`
struct my_list_element_t {
  int data;
  struct list_head __list;
};
// Declare an element of type `my_list_element_t`
struct my_list_element_t t;
// Add `t` to front or end of the list `head`
list_add(&t->__list, &head);
list_add_tail(&t->__list, &head);
// remove `t` from the list in which it is contained
list_del(&t->__list);
// Iterate over the list
list_for_each_entry(e, &head, __list) {
  // inside here, e is a pointer to the current element
  // visited and has type `struct my_list_element_t *e`;
}
```

Userspace access

uaccess.h

```
// copy_to_user - copy data from kernel space to
// user space returns the number of bytes that were
// not copied
unsigned long copy_to_user(void *to, const void *from,
   unsigned long n);
// copy_from_user - copy data from user space to
// kernel space returns the number of bytes that were
// not copied
unsigned long copy_from_user(void *to, const void *from,
   unsigned long n);
```

Other (Random Numbers)

```
// produce random number
void get_random_bytes(void *buf, int nbytes);
u32 get_random_u32();
u64 get_random_u64();
```