Barriers

`asm/barrier.h`

`barrier()`	Compiler barrier
`mb()`	Full system (I/O) memory barrier
`rmb()`	↪ reads only
`wmb()`	↪ writes only
`smp_mb()`	SMP (conditional) memory barrier
`smp_rmb()`	↪ reads only
`smp_wmb()`	↪ writes only
`smp_store_mb(v, val)`	Write `val` to `v`; then full memory barrier
`smp_load_acquire()`	Order preceding accesses against following read
`smp_store_release()`	Order following accesses against preceding write
`smp_mb__before_atomic()`	Order preceding accesses against atomic op.
`smp_mb__after_atomic()`	Order following accesses against atomic op.

Barriers must always be *paired* to be effective, although some operations (e.g. acquiring a lock) contain memory barriers implicitly.

Atomic operations

`asm/rwonce.h`

`READ_ONCE(x)`	Emit single instruction to load `x`
`WRITE_ONCE(x, val)`	Emit single instruction to store `x`

`asm/atomic.h`

`atomic_t`	Atomic 32-bit (signed) integer type
`atomic_read(v)`	Read from `v`
`atomic_set(v, i)`	Write `i` to `v`
`atomic_inc`❶`(v)`	Increment by 1
`atomic_inc_not_zero(v)`	↪ if the original value ≠ 0
`atomic_dec`❶`(v)`	Decrement by 1
`atomic_dec_and_test(v)`	↪ return true if the new value = 0
`atomic_add`❶`(i, v)`	Add `i` to (and write to) `v`
`atomic_add_return*(i, v)`	↪ return the new value
`atomic_fetch_add*(i, v)`	↪ return the old value
`atomic_add_unless(v, i, u)`	↪ unless the existing value is `u`
`atomic_sub`❶`(i, v)`	Subtract `i` from (and write to) `v`
`atomic_sub_and_test()`	↪ return true if the new value is 0
`atomic_and`❶`(i, v)`	`v &= i;`
`atomic_andnot`❶`(i, v)`	`v &= ~i;`
`atomic_or`❶`(i, v)`	`v |= i;`
`atomic_xor`❶`(i, v)`	`v ^= i;`
`atomic_xchg`❶`(v, n)`	Swap `v` and `n`; return original value
`atomic_cmpxchg`❶`(v, o, n)`	↪ if the original value = `o`
`atomic_try_cmpxchg`❶`(v, &o, n)`	↪ return true if swapped

Variants:

❶`_relaxed`	unordered
❶`_acquire`	read is ordered against subsequent reads
❶`_release`	write is ordered against preceding writes

Overflow/underflow is defined as two's complement.

`asm/atomic-long.h`

`atomic_long_t`	Atomic 64-bit (signed) integer type

Operations are the same as for `atomic_t`, i.e. `atomic_inc()` becomes `atomic_long_inc()`.

Reference counters

`linux/refcount.h`

`refcount_t`	Atomic reference count type
`r = REFCOUNT_INIT(n)`	Initialize `r`
`refcount_read(r)`	Read from `r`
`refcount_set(r, i)`	Write `i` to `r`
`refcount_inc`❶`(r)`	Increment `r` by 1
`refcount_add`❶`(i, r)`	Add `i` to `r`
`refcount_dec`❷`(r)`	Decrement `r` by 1
`refcount_dec_and_test(r)`	↪ return true if new value is 0
`refcount_dec_and_lock(r, mut)`	↪ lock mutex if new value is 0
`refcount_dec_and_lock_irqsave(r, spin, flags)`	↪ disable interrupts if enabled
`refcount_dec_and_mutex_lock(r, spin)`	↪ lock mutex if new value is 0
`refcount_sub_and_test(i, r)`	

Variants:

❶`_not_zero`	only if the original value is not 0
❷`_not_one`	only if the original value is not 1
❷`_if_one`	only if the original value is 1
❷`_and_test`	return true if the new value is 0

Spinlocks

`asm/spinlock.h`

`spinlock_t`	Spinlock type
`DEFINE_SPINLOCK()`	Variable definition
`spin_lock_init()`	Initialize spinlock
`spin_is_locked()`	Return `true` if spinlock is held (by any CPU)
`spin_trylock`❶`()`	Try to acquire spinlock without spinning; ⚠ returns `true` if spinlock was acquired
`spin_lock`❶`()`	Acquire spinlock; busy-looping
`spin_unlock`❷`()`	Release spinlock

Variants:

❶❷`_bh`	Disable soft-IRQs while locked
❶❷`_irq`	Disable interrupts while locked
❶`_irqsave`	Conditionally disable interrupts if enabled
❷`_irqrestore`	Conditionally reenable interrupts if originally enabled

In general, the variants must be paired, e.g. `spin_lock_bh()` with `spin_unlock_bh()` or `spin_lock_irqsave()` with `spin_unlock_irqrestore()`.

`linux/rwlock.h`

`rwlock_t`	Reader-writer spinlock type
`DEFINE_RWLOCK()`	Variable definition
`rwlock_init`	Initialize
`read_trylock`❶`()`	→ see `spin_trylock()`
`read_lock`❶`()`	→ see `spin_lock()`
`read_unlock`❷`()`	→ see `spin_unlock()`
`write_trylock`❶`()`	→ see `spin_trylock()`
`write_lock`❶`()`	→ see `spin_lock()`
`write_unlock`❷`()`	→ see `spin_unlock()`

Variants: ↑ see spinlocks.

The lock can be held by either a single writer or multiple readers.

Mutexes (sleeping)

`linux/mutex.h`

`struct mutex`	Mutex type
`DEFINE_MUTEX(name)`	Variable definition
`mutex_init(mut)`	Initialize `mut`
`mutex_is_locked(mut)`	True when `mut` is locked (by any thread)
`mutex_trylock(mut)`	Try to acquire `mut` without sleeping; ⚠ returns `true` if mutex was acquired
`mutex_lock`❶❷❸`(mut)`	Acquire `mut`; sleeping
`mutex_unlock(mut)`	Release `mut` (may schedule)

Variants:

❶`_interruptible`❸	Return `-EINTR` if a signal arrives
❶`_killable`❸	Return `-EINTR` if killed
❷`_io`❸	Account sleeping time as IO wait time
❸`_nested(mut, c)`	Used when acquiring two mutexes of the same class; `c` is a nesting level/class

Mutexes cannot be held, acquired, or released in atomic contexts.

Semaphores (sleeping)

`linux/semaphore.h`

`struct semaphore`	Semaphore type
`DEFINE_SEMAPHORE(name)`	Variable definition
`sema_init(sem, val)`	Initialize
`down_trylock(sem)`	Try to acquire `sem` without sleeping; ⚠ returns 0 if semaphore was acquired
`down`❶❷❹`(sem)`	Acquire `sem`; sleeping
`up(sem)`	Release `sem`

Variants: ↑ see mutexes;
❹`_timeout(sem, timeout)` Return if timeout expires

`linux/rwsem.h`

`struct rw_semaphore`	Reader-writer semaphore type
`DECLARE_RWSEM(name)`	Variable definition
`init_rwsem(sem)`	Initialize
`rwsem_is_locked(sem)`	Return true if `sem` is locked
`down_read_trylock(sem)`	→ see `down_trylock()`
`down_read`❶❸`(sem)`	→ see `down()`
`up_read(sem)`	→ see `up()`
`down_write_trylock(sem)`	→ see `down_trylock()`
`down_write`❶❸`(sem)`	→ see `down()`
`up_write(sem)`	→ see `up()`

Variants: ↑ see mutexes.

The lock can be held by either a single writer or multiple readers.

Linux kernel concurrency cheat sheet

Interrupts & preemption

linux/irqflags.h

local_irq_disable()	Unconditionally disable interrupts
local_irq_enable()	Unconditionally enable interrupts
local_irq_save(flags)	Conditionally disable interrupts
local_irq_restore(flags)	Conditionally reenable interrupts
irqs_disabled()	True when interrupts are disabled

Interrupt handlers run with interrupts disabled, are non-preemptible, and are atomic (cannot sleep).

Disabling interrupts implicitly disables soft-IRQs.
Disabling interrupts implicitly disables preemption.

linux/bottom_half.h

local_bh_disable()	Disable soft-IRQs (on this CPU)
local_bh_enable()	Enable soft-IRQs (on this CPU)
local_bh_blocked()	True when soft-IRQs are disabled (on this CPU)

Soft-IRQs (also known as *bottom halves* or *bh*) run with interrupts enabled.

linux/preempt.h

in_nmi()	True when in NMI context
in_hardirq()	True when in interrupt context
in_serving_softirq()	True when in soft-IRQ context
in_task()	True when in task context
in_atomic()	True when the caller cannot sleep (⚠ with exceptions)
preemptible()	True when in preemptible context
preempt_disable()	Disable preemption (nested)
preempt_enable()	Enable preemption (nested)
in_irq() (deprecated)	Same as in_hardirq()
in_softirq() (deprecated)	True when in soft-IRQ or soft-IRQ disabled
in_interrupt() (deprecated)	True when in NMI, interrupt, soft-IRQ, or soft-IRQ disabled

Preemption refers to being scheduled out. A non-preemptible context cannot be scheduled out, but may be interrupted.

preempt_disable() and preempt_enable() nest in such a way that preemption remains disabled as long as there is at least one unmatched call to preempt_disable() active.

Completions

linux/completion.h

struct completion	Type
DECLARE_COMPLETION(name)	Variable definition
init_completion(work)	Initialize work
reinit_completion(work)	Reinitialize after completion
completion_done(w)	True when completion is done
wait_for_completion❶❷(w)	Wait for a completion (sleeping)
try_wait_for_completion(w)	↪ without blocking; return 1 if done
complete(w)	Wake up a single waiting thread
complete_all(w)	Wake up all waiting threads

Variants:

❶_interruptible	Return -ERESTARTSYS if a signal arrives
❶_killable	Return -ERESTARTSYS if killed
❶_io	Account sleeping time as IO wait time
❷_timeout(w, timeout)	Return if timeout expires

Per-CPU variables

linux/percpu.h

cpu = get_cpu()	Disable preemption; return CPU number
put_cpu()	Reenable preemption
DECLARE_PER_CPU❶(type, name)	Variable declaration
DEFINE_PER_CPU❶(type, name)	Variable definition
EXPORT_PER_CPU_SYMBOL(name)	Export symbol
per_cpu(var, cpu)	Dereference per-CPU variable
get_cpu_var(var)	↪ disabling preemption
put_cpu_var(var)	↪ enabling preemption
per_cpu_ptr(var, cpu)	Get address of per-CPU variable
get_cpu_ptr(var)	↪ disabling preemption
put_cpu_ptr(var)	↪ enabling preemption
this_cpu_ptr(var)	Get address of this CPU's value
this_cpu_read(var)	Read this CPU's value
this_cpu_write(var)	Write this CPU's value
this_cpu_*()	→ see atomic operations

Variants:

❶_ALIGNED	Cacheline-aligned
❶_SHARED_ALIGNED	↪ accessible by other CPUs
❶_PAGE_ALIGNED	Page-aligned
❶_READ_MOSTLY	Rarely written to

RCU (Read-Copy-Update)

linux/rcupdate.h

rcu_read_lock❶()	Enter critical section
rcu_read_unlock❶()	Leave critical section
rcu_dereference❶(p)	Dereference p
rcu_access_pointer(p)	Fetch pointer p without dereferencing
rcu_assign_pointer(p, v)	Assign v to *p
rcu_replace_pointer(p, v)	↪ return original value
struct rcu_head	RCU head type
rcu_head_init(head)	Initialize
call_rcu(head, fn)	Call fn after grace period
kfree_rcu(p, name)	Free p after grace period, using p->name
synchronize_rcu()	Wait for readers to complete

Variants:

❶_bh()	Also disable (reenable) soft-IRQs
❶_sched()	Also disable (reenable) preemption

Writers must always use either a single atomic update *or* exclude other writers using other synchronization mechanisms (like spinlocks).

Sequence locks

linux/seqlock.h

seqcount_t	Type
SEQCOUNT_ZERO	Static initializer
seqcount_init(s)	Initialize

Writer:

```
write_seqcount_begin(&s);
...
write_seqcount_end(&s);
```

Reader:

```
do {
   seq = read_seqcount_begin(&s);
   ...
} while (read_seqcount_retry(&s, seq));
```

Wait queues

linux/wait.h

Queues:

wait_queue_head_t	Wait queue type
DECLARE_WAIT_QUEUE_HEAD(name)	Variable definition
DECLARE_WAIT_QUEUE_HEAD_ONSTACK(name)	↪ for local variables
init_waitqueue_head(wq)	Initialize
wait_event❶❷❸(wq, cond)	Sleep until condition is true
io_wait_event(wq, cond)	↪ using io_schedule()
wake_up(wq)	Wake up waiters

Variants: (⚠ incomplete)

❶_interruptible❷❸	Returns -ERESTARTSYS if interrupted
❶_killable❷	Returns -ERESTARTSYS if killed
❶_freezable❷	Allow freezing while waiting
❷_timeout(wq, cond, t)	Also returns when timeout expires
❸_lock_irq(wq, cond, lock)	Hold spinlock while checking condition

Entries:

wait_queue_entry_t	Wait queue entry type
DEFINE_WAIT(e)	Variable definition
DEFINE_WAIT_FUNC(e, fn)	↪ using custom wake function
init_wait(e)	Initialize
prepare_to_wait(wq, e, state)	Enqueue wait-queue entry
prepare_to_wait_exclusive(...)	↪ only wake the first thread
finish_wait(wq, e)	Dequeue wait-queue entry

Lists

linux/list.h

struct list_head	Type
LIST_HEAD()	Define
INIT_LIST_HEAD(head)	Initialize
list_add(e, head)	Add e to the start of head
list_add_tail(e, head)	Add e to the end of head
list_del(e)	Remove e
list_del_init(e)	↪ reinitialize e
list_replace(old, new)	Replace old by new
list_replace_init(old, new)	↪ reinitialize old
list_swap(e1, e2)	Swap e1 and t2
list_move(e, head)	Remove e; add to the start of head
list_move_tail(e, head)	Remove e; add to the end of head
list_is_head(e, head, member)	True when e is the head of the list
list_is_first(e, head)	True when e is the first element of head
list_is_last(e, head)	True when e is the last element of head
list_empty(head)	True when head is an empty list
list_is_singular(head)	True when head contains one element
list_for_each_entry❶(...)	Iterate over list

Variants:

❶(e, head, member)	Forward iteration
❶_safe(e, tmp, head, member)	↪ allow node deletion
❶_reverse(e, head, member)	Backwards iteration
❶_safe_reverse(e, tmp, head, member)	↪ allow node deletion