Synchronization & Interprocess Communication

Have a look

- Before getting into **Kernel Synch & IPC**, please take a look at
  - LK_Bird's Eye View session,

    S8 Synch,Lock,Interrupt,
    Exception,Softirq,Tasklet,WorkQueue.

Synchronization

- Atomic operations
  - Atomicity guarantees indivisible and uninterruptible execution of the operation initiated.

    Linux kernel code uses atomic operations for various use cases, such as reference counters in shared data structures,...

    1) Atomic integer operations

Interface macro/Inline function	Description
`ATOMIC_INIT(i)`	Macro to initialize an atomic counter
`atomic_read(v)`	Read value of the atomic counter v
`atomic_set(v, i)`	Atomically set counter v to value specified in i
`atomic_add(int i, atomic_t *v)`	Atomically add i to counter v
`atomic_sub(int i, atomic_t *v)`	Atomically subtract i from counter v
`atomic_inc(atomic_t *v)`	Atomically increment counter v
`atomic_dec(atomic_t *v)`	Atomically decrement counter v

/include/linux/types.h

```
typedef struct {
    int counter;
} atomic_t;

#ifdef CONFIG_64BIT
typedef struct {
    s64 counter;
} atomic64_t;
#endif
```

Synchronization

- Atomic operations, *Cont'd*
  - 1) Atomic integer operations, *Cont'd*
    read-modify-write (RMW) operations

Operation	Description
`bool atomic_sub_and_test(int i, atomic_t *v)`	Atomically subtracts `i` from `v` and returns `true` if the result is zero, or `false` otherwise
`bool atomic_dec_and_test(atomic_t *v)`	Atomically decrements `v` by 1 and returns `true` if the result is 0, or `false` for all other cases
`bool atomic_inc_and_test(atomic_t *v)`	Atomically adds `i` to `v` and returns `true` if the result is 0, or `false` for all other cases
`bool atomic_add_negative(int i, atomic_t *v)`	Atomically adds `i` to `v` and returns `true` if the result is negative, or `false` when result is greater than or equal to zero
`int atomic_add_return(int i, atomic_t *v)`	Atomically adds `i` to `v` and returns the result

/include/linux/types.h

```
typedef struct {
    int counter;
} atomic_t;

#ifdef CONFIG_64BIT
typedef struct {
    s64 counter;
} atomic64_t;
#endif
```

Synchronization

- Atomic operations, *Cont'd*
  - 2) Atomic bitwise operations

Operation interface	Description
`set_bit(int nr, volatile unsigned long *addr)`	Atomically set the bit `nr` in location starting from `addr`
`clear_bit(int nr, volatile unsigned long *addr)`	Atomically clear the bit `nr` in location starting from `addr`
`change_bit(int nr, volatile unsigned long *addr)`	Atomically flip the bit `nr` in the location starting from `addr`

`int test_and_set_bit(int nr, volatile unsigned long *addr)`	Atomically set the bit `nr` in the location starting from `addr`, and return old value at the nr^{th} bit
`int test_and_clear_bit(int nr, volatile unsigned long *addr)`	Atomically clear the bit `nr` in the location starting from `addr`, and return old value at the nr^{th} bit
`int test_and_change_bit(int nr, volatile unsigned long *addr)`	Atomically flip the bit `nr` in the location starting from `addr`, and return old value at the nr^{th} bit

/include/linux/types.h

```
typedef struct {
    int counter;
} atomic_t;

#ifdef CONFIG_64BIT
typedef struct {
    s64 counter;
} atomic64_t;
#endif
```

Synchronization

- Spinlock
  - Data structures should be protected from being concurrently accessed by kernel control paths that run on different CPUs.
  - When a caller context attempts to acquire **spinlock** while it is locked (or held by another context), the lock function iteratively polls or spins for the lock until available, causing the caller context to hog the CPU until lock is acquired.
  - The generic interface provides a bunch of ***lock()*** and ***unlock()*** operations, each implemented for a specific use case.

    spin_lock() actual implementation for SMP
    __raw_spin_lock()

    spin_unlock() actual implementation for SMP
    __raw_spin_unlock()

```
typedef struct spinlock {
    union {
        struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
        struct {
            u8 __padding[LOCK_PADSIZE];
            struct lockdep_map dep_map;
        };
#endif
    };
} spinlock_t;
```

/include/linux/spinlock.h

```
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}

static __always_inline void spin_unlock(spinlock_t *lock)
{
    raw_spin_unlock(&lock->rlock);
}
```

/include/linux/spinlock_api_smp.h

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}

static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    spin_release(&lock->dep_map, _RET_IP_);
    do_raw_spin_unlock(lock);
    preempt_enable();
}
```

Synchronization

- Spinlock, *Cont'd*
  - Alternate spinlock APIs
    Standard spinlock are suitable for shared resources that are accessed only from the **process** context kernel path. But not for shared resource can be accessed from both the **process and interrupt** context.

    let's assume the following events occur in order :
    1) Process context routine of the driver acquires lock (using the standard ***spin_lock()*** call).

    2) While the critical section is in execution, an interrupt occurs and is driven to the local CPU, causing the process context routine to preempt and give away the CPU for interrupt handlers.

    3) Interrupt context path of the driver (ISR) starts and tries to acquire lock (using the standard ***spin_lock()*** call), which then starts to spin for lock to be available.

    So, CPU is hard locked with a spinning interrupt handler that never yields.

  - Kernel provides an alternate locking routine ***spin_lock_irqsave()*** for SMP, which disables interrupts on the current processor along with kernel preemption.

    /include/linux/spinlock_api_smp.h

```
static inline unsigned long __raw_spin_lock_irqsave(raw_spinlock_t *lock)
{
    unsigned long flags;

    local_irq_save(flags);
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
    return flags;
}
```

Synchronization

- Spinlock, *Cont'd*
  - List of the kernel spinlock API interface

Function	Description
`spin_lock_init()`	Initialize spinlock
`spin_lock()`	Acquire lock, spins on contention
`spin_trylock()`	Attempt to acquire lock, returns error on contention
`spin_lock_bh()`	Acquire lock by suspending BH routines on the local processor, spins on contention
`spin_lock_irqsave()`	Acquire lock by suspending interrupts on the local processor by saving current interrupt state, spins on contention
`spin_lock_irq()`	Acquire lock by suspending interrupts on the local processor, spins on contention
`spin_unlock()`	Release the lock
`spin_unlock_bh()`	Release lock and enable bottom half for the local processor
`spin_unlock_irqrestore()`	Release lock and restore local interrupts to previous state
`spin_unlock_irq()`	Release lock and restore interrupts for the local processor
`spin_is_locked()`	Return state of the lock, nonzero if lock is held or zero if lock is available

Synchronization

- Reader-writer spinlocks
  - **Reader-writer** locks enforce exclusion between reader and writer paths; this allows concurrent **readers** to **share** lock and a **reader** task will need to wait for the lock while a **writer** owns the lock.

Function	Description
`read_lock()`	Standard read lock interface, spins on contention
`read_trylock()`	Attempts to acquire lock, returns error if lock is unavailable
`read_lock_bh()`	Attempts to acquire lock by suspending BH execution for the local CPU, spins on contention
`read_lock_irqsave()`	Attempts to acquire lock by suspending interrupts for the current CPU by saving current state of local interrupts, spins on contention
`read_unlock()`	Releases read lock
`read_unlock_irqrestore()`	Releases lock held and restores local interrupts to the previous state
`read_unlock_bh()`	Releases read lock and enables BH on the local processor
`write_lock()`	Standard write lock interface, spins on contention
`write_trylock()`	Attempts to acquire lock, returns error on contention
`write_lock_bh()`	Attempts to acquire write lock by suspending bottom halves for the local CPU, spins on contention
`wrtie_lock_irqsave()`	Attempts to acquire write lock by suspending interrupts for the local CPU by saving current state of local interrupts,. spins on contention
`write_unlock()`	Releases write lock
`write_unlock_irqrestore()`	Releases lock and restores local interrupts to the previous state
`write_unlock_bh()`	Releases write lock and enables BH on the local processor

/include/linux/rwlock_types.h

```
typedef struct {
    arch_rwlock_t raw_lock;
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} rwlock_t;
```

Synchronization

- Mutex
  - Kernel mutexes are an implementation of **sleeping locks**: when a caller task attempts to acquire a mutex that is unavailable (already owned by another context), it is put into sleep and moved out into a wait queue, forcing a **context switch** allowing the CPU to run other tasks.
  - Each mutex contains a 64-bit ***atomic_long_t owner*** counter which is used both for holding lock state, and to store a reference to the task structure of the **current** task owning the lock.
  - It contains a wait-queue ***wait_list*** and a spin lock ***wait_lock*** that serializes access to ***wait_list.***

/include/linux/mutex.h

```
struct mutex {
    atomic_long_t       owner;
    raw_spinlock_t      wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
    struct list_head    wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
#endif
};
```

Synchronization

- Mutex, *Cont'd*
  - Some Mutex APIs

    void mutex_lock(struct mutex *lock) : acquires the mutex, caller process uninterruptile.

    int __must_check mutex_lock_interruptible(struct mutex *lock) : acquires the mutex, caller process interruptible by signals.

    int mutex_trylock(struct mutex *lock) : try to acquire the mutex, without waiting.

    int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock) : return holding mutex if we dec to 0.

    void mutex_unlock(struct mutex *lock) : release the mutex.

/include/linux/mutex.h

```
struct mutex {
    atomic_long_t       owner;
    raw_spinlock_t      wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
    struct list_head    wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
#endif
};
```

Synchronization

- Wait/wound mutex
  - Imagine two threads (T1 and T2) that attempt to lock 2 buffers in the opposite order: T1 starts with Buffer A, while T2 starts with Buffer B.
  - A Dead Lock will happen, and both threads will stuck.
  - Wait/wound mutex prevents this by letting the thread that grabbed the lock first to remain in sleep, waiting for nested lock to be available.

    The other thread is wound, causing it to release its holding lock and start over again.

    Suppose T1 got to lock on bufA before T2 could acquire lock on bufB. T1 would be considered as the thread that got there first and is put to sleep for lock on bufB, and T2 would be wound, causing it to release lock on bufB and start all over.

    This avoids deadlock and T2 would start all over when T1 releases locks held.

```
Thread T1          Thread T2
===========        ==========
lock(bufA);        lock(bufB);
lock(bufB);        lock(bufA);
 ....               ....
 ....               ....
unlock(bufB);      unlock(bufA);
unlock(bufA);      unlock(bufB);
```

Synchronization

- Wait/wound mutex, *Cont'd*
  - Some w/w APIs

    void ww_acquire_init(struct ww_acquire_ctx *ctx, struct ww_class *ww_class) : initializes a w/w acquire context.

    int ww_mutex_lock(struct ww_mutex *lock, struct ww_acquire_ctx *ctx) : acquires the mutex.

    int ww_mutex_lock_interruptible(struct ww_mutex *lock, struct ww_acquire_ctx *ctx) : acquires the mutex, caller process interruptible by signals.

    void ww_acquire_done(struct ww_acquire_ctx *ctx) : marks the end of the acquire phase, any further w/w mutex lock calls using this context are forbidden.

    void ww_acquire_fini(struct ww_acquire_ctx *ctx) : releases a w/w acquire context. This must be called _after_ all acquired w/w mutexes have been released with ***ww_mutex_unlock()***.

/include/linux/ww_mutex.h

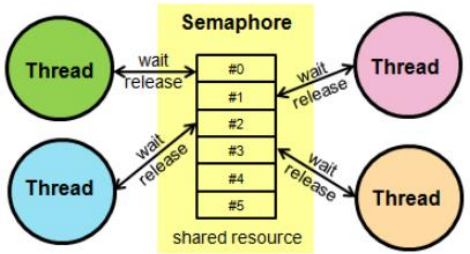
```
struct ww_mutex {
    struct WW_MUTEX_BASE base;
    struct ww_acquire_ctx *ctx;
#ifdef DEBUG_WW_MUTEXES
    struct ww_class *ww_class;
#endif
};
```

Synchronization

- Semaphores
  - When a semaphore is used to protect a shared resource, its counter is initialized to a number greater than zero, which is considered to be unlocked.
  - i.e. for cases where a resource needs to be accessible to a specific number of tasks at any point in time, the semaphore count can be initialized to the number of tasks that require access, say 10, which allows a maximum of 10 tasks access to shared resource at any time.
  - Semaphore count can be initialized to 1, resulting in a maximum of one task to access the resource at any given point in time.

/include/linux/semaphore.h

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```



Synchronization

- Semaphores, *Cont'd*
  - Some Semaphores APIs

    void sema_init(struct semaphore *sem, int val) : Dynamically initialize to any positive number.

    int down_interruptible(struct semaphore *sem) : Attempts to acquire the semaphore. If no more tasks are allowed to acquire the semaphore, calling this function will put the task to sleep.

    int down_trylock(struct semaphore *sem) : Try to acquire the semaphore atomically. Returns 0 if the semaphore has been acquired successfully or 1 if it cannot be acquired.

    int down_timeout(struct semaphore *sem, long timeout) : acquire the semaphore within a specified time.

    void up(struct semaphore *sem) : release the semaphore, Unlike mutexes, ***up()*** may be called from any context and even by tasks which have never called ***down()***.

/include/linux/semaphore.c

Synchronization

- Completion locks
  - Completion locks achieve code synchronization if you need one or multiple threads of execution to wait for completion of some event, such as waiting for another process to reach a point or state.
  - Completion locks may be preferred over a semaphore for a couple of reasons: multiple threads of execution can wait for a completion, and using ***comlete_all()***, they can all be released at once.
  - It uses a FIFO to queue the threads waiting for the completion event ***swait_queue_head.***

/include/linux/completion.h

```
struct completion {
    unsigned int done;
    struct swait_queue_head wait;
};
```

Synchronization

- Completion locks, *Cont'd*
  - Some Completion APIs
    void init_completion(struct completion *x) : initialize a dynamically created completion structure.

    void __sched wait_for_completion(struct completion *x) : waits to be signaled for completion of a specific task. It is NOT interruptible and there is no timeout.

    void complete(struct completion *x) : This will wake up a single thread waiting on this completion. Threads will be awakened in the same order in which they were queued.

    void complete_all(struct completion *x) : This will wake up all threads waiting on this particular completion event.

/include/linux/completion.h

```
struct completion {
    unsigned int done;
    struct swait_queue_head wait;
};
```

Synchronization

- RCU (Read Copy Update)
  - Protect data structures that are mostly accessed for reading by several CPUs.
  - RCU allows many readers and many writers to proceed concurrently.
  - RCU is lock free.

  - Constraints :
    Only data structures that are dynamically allocated and referenced via pointers can be protected by RCU.
    No kernel control path can sleep inside a critical section protected by RCU.

  - Mechanism :
    Kernel keeps track of all users of the pointer to the shared data structure. When the structure is supposed to change, a copy (or a new instance that is filled in appropriately, this does not make any difference) is first created and the change is performed there.

    After all previous readers have finished their reading work on the old copy, the pointer can be replaced by a pointer to the new, modified copy.

Synchronization

- RCU (Read Copy Update), *Cont'd*
  - Some Core API
    i.e. Suppose that a pointer ***per_cpu_ptr(&sft_data, cpu)*** points to a data structure that is protected by RCU.

    rcu_read_lock() and ***rcu_read_unlock()*** : mark Begin/End of an RCU read-side critical section.

    It is forbidden to simply de-reference the pointer, but ***rcu_dereference()*** must be invoked before the result be dereferenced.

    publish a new pointer must be done through ***rcu_assign_pointer().***

    synchronize_rcu() : waits until all existing readers have finished their work. After the function returns, it is safe for the kernel to free the memory associated with the old pointer.

    Note 

/drivers/base/arch_topology.c

```
rcu_read_lock();

for_each_cpu(cpu, cpus) {
    sfd = rcu_dereference(*per_cpu_ptr(&sft_data, cpu));

    if (sfd && sfd->source == source) {
        rcu_assign_pointer(per_cpu(sft_data, cpu), NULL);
        cpumask_clear_cpu(cpu, &scale_freq_counters_mask);
    }
}

rcu_read_unlock();

/*
 * Make sure all references to previous sft_data are dropped to avoid
 * use-after-free races.
 */
synchronize_rcu();
```

If updates can come from many places in the kernel, protection against concurrent write operations must be provided using regular synchronization primitives, for instance, spinlocks. While RCU protects readers from writers, it does not protect writers against writers!

Synchronization

- Ordering and Barriers
  - When talking with hardware, you often need to ensure that a given read **(Load)** occurs before another read or write **(Store)**.
  - All processors that do **reorder** reads or writes provide machine instructions to enforce ordering requirements.
  - It is possible to instruct not to reorder instructions around a given point.These instructions are called ***barriers***.
  - ***rmb()*** : arch-specific read memory barrier. It ensures that no loads are reordered across the ***rmb()*** call.That is, no loads prior to the call will be reordered to after the call, and no loads after the call will be reordered to before the call.
  - ***wmb()*** : arch-specific write barrier. It functions in the same manner as ***rmb()***, but with respect to stores instead of loads, it ensures no stores are reordered across the barrier.

/arch/arm64/include/asm/barrier.h

```
#define dsb(opt)    asm volatile("dsb " #opt : : : "memory")
#define mb()        dsb(sy)
#define rmb()       dsb(ld)
#define wmb()       dsb(st)
```

Synchronization

- Ordering and Barriers, *Cont'd*
  - ***mb()*** : arch-specific both a read barrier and a write barrier. No loads or stores will be reordered across a call to ***mb()***.
  - Consider this example of 2 threads.
    Using the ***mb()*** ensured that ***a*** and ***b*** were written in the intended order, whereas the ***rmb()*** insured ***c*** and ***d*** were read in the intended order.
  - Some other Barrier APIs

Barrier	Description
`smp_rmb()`	Provides an `rmb()` on SMP, and on UP provides a `barrier()`
`smp_wmb()`	Provides a `wmb()` on SMP, and provides a `barrier()` on UP
`smp_mb()`	Provides an `mb()` on SMP, and provides a `barrier()` on UP
`barrier()`	Prevents the compiler from optimizing stores or loads across the barrier

/arch/arm64/include/asm/barrier.h

```
#define dsb(opt)    asm volatile("dsb " #opt : : : "memory")
#define mb()        dsb(sy)
#define rmb()       dsb(ld)
#define wmb()       dsb(st)
```

Thread 1	Thread 2
`a = 3;`	—
`mb();`	—
`b = 4;`	`c = b;`
—	`rmb();`
—	`d = a;`

IPC (Interprocess Communication)

- Pipes and FIFOs
  - Pipes provide a unidirectional interprocess communication channel.
  - Pipes introduce **communication synchronization**, as if the writing process writes much faster than the reading process reads, the pipes capacity will fail to hold excess data and invariably block the writing process until the reader reads and frees up data.
  - Pipes are referred to as ***nonymous/unnamed*** pipes, since they are not enumerated as files under the ***rootfs*** tree (/).
  - ***pipe2*** syscall allocates appropriate data structures and sets up pipe buffers. It maps a pair of file descriptors, one for reading on the pipe buffer and another for writing on the pipe buffer.

/fs/pipe.c

```
static int do_pipe2(int __user *fildes, int flags)
{
    struct file *files[2];
    int fd[2];
    int error;

    error = __do_pipe_flags(fd, files, flags);
    if (!error) {
        if (unlikely(copy_to_user(fildes, fd, sizeof(fd))))
{
            fput(files[0]);
            fput(files[1]);
            put_unused_fd(fd[0]);
            put_unused_fd(fd[1]);
            error = -EFAULT;
        } else {
            fd_install(fd[0], files[0]);
            fd_install(fd[1], files[1]);
        }
    }
    return error;
}

SYSCALL_DEFINE2(pipe2, int __user *, fildes, int, flags)
{
    return do_pipe2(fildes, flags);
}
```

IPC (Interprocess Communication)

- Pipes and FIFOs, *Cont'd*
  - FIFOs considered as ***named*** pipes as the communication between processes requires the pipe file to be enumerated into ***rootfs.***
  - FIFOs can be created either from a process using the ***mkfifo()*** API from user-space.
  - ***mknod()*** is invoked for creating a FIFO, which internally invokes VFS routines to set up the named pipe.
  - Pipes and FIFOs are created and managed by a special filesystem called ***pipefs***,
    it registers with VFS as a special filesystem.
  - VFS by enumerating an ***inode*** instance representing each pipe; this allows applications to engage common file APIs ***read*** and ***write***.

glibc-2.36.9000

```
int
mkfifo (const char *path, mode_t mode)
{
  return __mknod (path, mode | S_IFIFO, 0);
}
```

/fs/ext4/namei.c

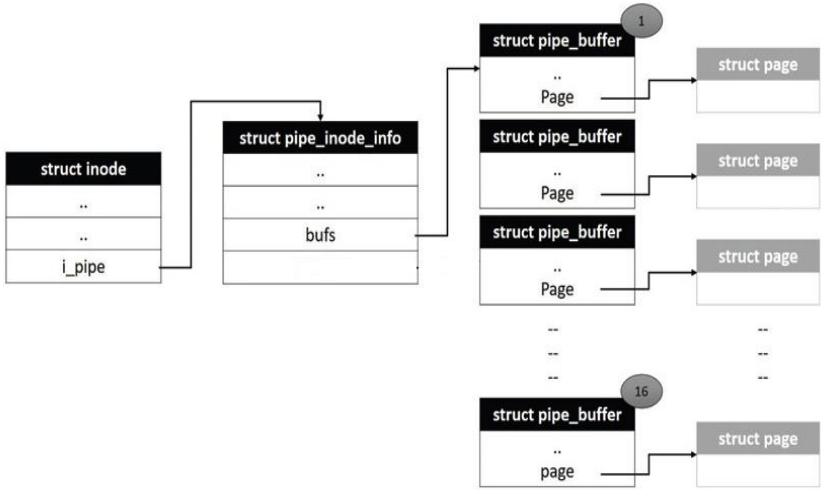
```
static int ext4_mknod(struct user_namespace *mnt_userns, struct
inode *dir, struct dentry *dentry, umode_t mode, dev_t rdev)
{
    ...
}
```

/fs/pipe.c

```
static struct file_system_type pipe_fs_type = {
    .name       = "pipefs",
    .init_fs_context = pipefs_init_fs_context,
    .kill_sb    = kill_anon_super,
};

static int __init init_pipe_fs(void)
{
    int err = register_filesystem(&pipe_fs_type);

    if (!err) {
        pipe_mnt = kern_mount(&pipe_fs_type);
        if (IS_ERR(pipe_mnt)) {
            err = PTR_ERR(pipe_mnt);
            unregister_filesystem(&pipe_fs_type);
        }
    }
#ifdef CONFIG_SYSCTL
    register_sysctl_init("fs", fs_pipe_sysctls);
#endif
    return err;
}

fs_initcall(init_pipe_fs);
```

IPC (Interprocess Communication)

- Pipes and FIFOs, *Cont'd*
  - ***pipe_inode_info*** contains all pipe-related metadata.
  - ***bufs*** refers to the pipe buffer; each pipe is by default assigned a total buffer of 65,535 bytes (64k) arranged as a circular array of **16** pages.
  - User processes can alter the total size of the pipe buffer.




```
struct inode {
    ...
    union {
        struct pipe_inode_info  *i_pipe;
        struct cdev             *i_cdev;
        char                    *i_link;
        unsigned                i_dir_seq;
    };
    ...
};
```

/include/linux/pipe_fs_i.h

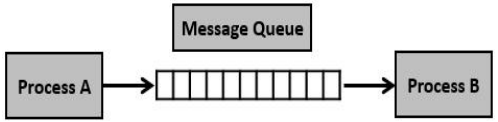
```
struct pipe_inode_info {
    ...
    struct pipe_buffer *bufs;
    ...
};
```

/include/linux/pipe_fs_i.h

```
struct pipe_buffer {
    struct page *page;
    ...
};
```

IPC (Interprocess Communication)

- POSIX message queues (mq)
  - POSIX message queues implement priority-ordered messages. Each message written by a sender process is associated with an integer number which is interpreted as **message priority**; messages with a higher number are considered higher in priority.
  - POSIX message queues are managed by a special filesystem called ***mqueue*** FS. Each message queue is identified by a filename.
  - Metadata for each queue is described by an instance of *mqueue_inode_info.*



/ipc/mqueue.c

```
static struct file_system_type mqueue_fs_type = {
    .name           = "mqueue",
    .init_fs_context    = mqueue_init_fs_context,
    .kill_sb        = kill_litter_super,
    .fs_flags       = FS_USERNS_MOUNT,
};
```

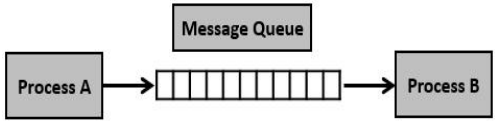
/ipc/mqueue.c

```
struct mqueue_inode_info {
    spinlock_t lock;
    struct inode vfs_inode;
    ...
};
```

IPC (Interprocess Communication)

- POSIX message queues, *Cont'd*
  - mq user-space APIs.

API interface	Description
`mq_open()`	Create or open a POSIX message queue
`mq_send()`	Write a message to the queue
`mq_timedsend()`	Similar to `mq_send`, but with a timeout parameter for bounded operations
`mq_receive()`	Fetch a message from the queue; this operation is possible on unbounded blocking calls
`mq_timedreceive()`	Similar to `mq_receive()` but with a timeout parameter that limits possible blocking for bounded time
`mq_close()`	Close a message queue
`mq_unlink()`	Destroy message queue
`mq_notify()`	Customize and set up message arrival notifications
`mq_getattr()`	Get attributes associated with a message queue
`mq_setattr()`	Set attributes specified on a message queue



IPC (Interprocess Communication)

- POSIX shared memory (shm)
  - kernel supports POSIX shared memory through a special filesystem called ***tmpfs*** which is rooted at ***/dev/shm***.
  - Each shared memory allocation to be represented by a unique filename and ***inode***.
  - This interface is considered more flexible by application programmers since it allows standard POSIX file-mapping routines ***mmap()*** and ***unmap()*** for attaching and detaching memory segments into the caller process address space.
  - shm user-space APIs

API	Description
`shm_open()`	Create and open a shared memory segment identified by a filename
`mmap()`	POSIX standard file mapping interface for attaching shared memory to caller's address space
`sh_unlink()`	Destroy specified shared memory block
`unmap()`	Detach specified shared memory map from caller address space