while (true) {

    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
while (true) {
    while (count == 0) {
        /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
    }
    /* consume the item in nextConsumed */
}
Race Condition

- `count++` could be implemented as
  
  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- `count--` could be implemented as
  
  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

- Consider this execution interleaving with “count = 5” initially:

  
  ```
  S0: producer execute register1 = count  {register1 = 5}
  S1: producer execute register1 = register1 + 1  {register1 = 6}
  S2: consumer execute register2 = count  {register2 = 5}
  S3: consumer execute register2 = register2 - 1  {register2 = 4}
  S4: producer execute count = register1  {count = 6}
  S5: consumer execute count = register2  {count = 4}
  ```
Need for Synchronization

- When data generated by one process are transferred to another
- When data are shared
- When processes are forced to wait for each other
- When resource usage needs to be coordinated
Kernel Synchronization

- Can think of the kernel as a server
  - Concurrent requests are possible
  - Synchronization is (usually) required
- Need to avoid race conditions
  - Correctness violated by timing changes
  - Need to identify, secure critical section (mutex)
- Kernel vs. userland synch primitives
  - example: semaphore system call vs. in-kernel semaphore
- Synchronization is complex and subtle
- Hierarchy of primitives
  - lowest level: hardware primitives
  - higher level: built using lower-level
    - e.g. semaphores use atomic inc, spinlocks, waitqueues
Linux Synch Primitives

- Memory barriers
  - Avoids compiler, cpu instruction re-ordering
- Atomic operations
  - Memory bus lock, read-modify-write ops
- Interrupt/softirq disabling/enabling
  - Local, global
- Spin locks
  - General, read/write, big reader
- Semaphores
  - General, read/write
Choosing Synch Primitives

- Generally, choice is affected by
  - Will contention be high?
  - Are you in process context?
  - How much do you need to do inside of critical section?
  - Do you need to sleep?
  - Do you need to acquire lock frequently?
Choosing Synch Primitives

- Avoid synch if possible! (clever instruction ordering)
  - Example: inserting in linked list (needs barrier still)
- Use atomics or rw spinlocks if possible
- Use semaphores if you need to sleep
  - Can’t sleep in interrupt context
  - Don’t sleep holding a spinlock!
- Complicated matrix of choices for protecting data structures accessed by deferred functions
The implementation of the synchronization primitives is extremely architecture dependent. This is because only the hardware can guarantee atomicity of an operation. Each architecture must provide a mechanism for doing an operation that can examine and modify a storage location atomically. Some architectures do not guarantee atomicity, but inform whether the operation attempted was atomic.
Barriers: Motivation

- The compiler can:
  - Reorder code as long as it correctly maintains data flow dependencies within a function and with called functions
  - Reorder the execution of code to optimize performance

- The processor can:
  - Reorder instruction execution as long as it correctly maintains register flow dependencies
  - Reorder memory modification as long as it correctly maintains data flow dependencies
  - Reorder the execution of instructions (for performance optimization)
Barriers: Definition

- **Barriers** are used to *prevent* a processor and/or the compiler from reordering instruction execution and memory modification.

- **Barriers** are instructions to hardware and/or compiler to *complete all pending accesses* before issuing any more
  - **read memory barrier** – acts on read requests
  - **write memory barrier** – acts on write requests

- **Intel** –
  - **certain instructions** act as barriers: lock, iret, control regs
  - **rmb** – `asm volatile("lock;addl $0,0(%%esp)":::"memory")`
    - add 0 to top of stack with lock prefix
  - **wmb** – Intel never re-orders writes, just for compiler
Barriere Operations

- **barrier** – prevent only compiler reordering
- **mb** – prevents load and store reordering
- **rmb** – prevents load reordering
- **wmb** – prevents store reordering
  - **smp_mb** – prevent load and store reordering only in SMP kernel
  - **smp_rmb** – prevent load reordering only in SMP kernels
  - **smp_wmb** – prevent store reordering only in SMP kernels
  - **set_mb** – performs assignment and prevents load and store reordering
Serializing with Interrupts

- Basic primitive in original UNIX
- Doesn’t protect against other CPUs
- Intel: “interrupts enabled bit”
  - cli to clear (disable), sti to set (enable)
- Enabling is often wrong; need to restore
  - local_irq_save()
  - local_irq_restore()
Services used to serialize with interrupts are:

- `local_irq_disable` - disables interrupts on the current CPU
- `local_irq_enable` - enable interrupts on the current CPU
- `local_save_flags` - return the interrupt state of the processor
- `local_restore_flags` - restore the interrupt state of the processor

Dealing with the full interrupt state of the system is officially discouraged. Locks should be used.
Disabling Deferred Functions

- Disabling interrupts disables deferred functions
- Possible to disable deferred functions but not all interrupts
- Operations (macros):
  - `local_bh_disable()`
  - `local_bh_enable()`
Atomic Operations

- Many instructions not atomic in hardware (smp)
  - Read-modify-write instructions: inc, test-and-set, swap
  - Unaligned memory access

- Compiler may not generate atomic code
  - Even i++ is not necessarily atomic!

- If the data that must be protected is a single word, atomic operations can be used
  - These functions examine and modify the word atomically

- The atomic data type is **atomic_t**
Atomic Operations

- Execute in a single instruction
- Can be used in or out of process context (i.e., softirqs)
- Never sleep
- Don’t suspend interrupts
Atomic Operations

ATOMIC_INIT – initialize an atomic_t variable
atomic_read – examine value atomically
atomic_set – change value atomically
atomic_inc – increment value atomically
atomic_dec – decrement value atomically
atomic_add - add to value atomically
atomic_sub – subtract from value atomically
atomic_inc_and_test – increment value and test for zero
atomic_dec_and_test – decrement value and test for zero
atomic_sub_and_test – subtract from value and test for zero
atomic_set_mask – mask bits atomically
atomic_clear_mask – clear bits atomically
Spin Locks

- A spin lock is a data structure (*spinlock_t*) that is used to synchronize access to critical sections.

- Only one thread can be holding a spin lock at any moment. All other threads trying to get the lock will “spin” (loop while checking the lock status).

- Spin locks should not be held for long periods because waiting tasks on other CPUs are spinning, and thus wasting CPU execution time.
Spin Lock Operations

- Functions used to work with spin locks:
  - `spin_lock_init` – initialize a spin lock before using it for the first time
  - `spin_lock` – acquire a spin lock, spin waiting if it is not available
  - `spin_unlock` – release a spin lock
  - `spin_unlock_wait` – spin waiting for spin lock to become available, but don't acquire it
  - `spin_trylock` – acquire a spin lock if it is currently free, otherwise return error
  - `spin_is_locked` – return spin lock state
Spin Locks & Interrupts

- The spin lock services also provide interfaces that serialize with interrupts (on the current processor):
  - `spin_lock_irq` - acquire spin lock and disable interrupts
  - `spin_unlock_irq` - release spin lock and reenable interrupts
  - `spin_lock_irqsave` - acquire spin lock, save interrupt state, and disable
  - `spin_unlock_irqrestore` - release spin lock and restore interrupt state
A read/write spin lock is a data structure that allows multiple tasks to hold it in "read" state or one task to hold it in "write" state (but not both conditions at the same time).

This is convenient when multiple tasks wish to examine a data structure, but don't want to see it in an inconsistent state.

A lock may not be held in read state when requesting it for write state.

The data type for a read/write spin lock is `rwlock_t`.

Writers can starve waiting behind readers.
Several functions are used to work with read/write spin locks:

- `rwlock_init` – initialize a read/write lock before using it for the first time
- `read_lock` – get a read/write lock for read
- `write_lock` – get a read/write lock for write
- `read_unlock` – release a read/write lock that was held for read
- `write_unlock` – release a read/write lock that was held for write
- `read_trylock`, `write_trylock` – acquire a read/write lock if it is currently free, otherwise return error
The read/write lock services also provide interfaces that serialize with interrupts (on the current processor):

- `read_lock_irq` - acquire lock for read and disable interrupts
- `read_unlock_irq` - release read lock and reenable
- `read_lock_irqsave` - acquire lock for read, save interrupt state, and disable
- `read_unlock_irqrestore` - release read lock and restore interrupt state

Corresponding functions for write exist as well (e.g., `write_lock_irqsave`)
Semaphores

- A **semaphore** is a data structure that is used to synchronize access to critical sections or other resources.
- A **semaphore** allows a fixed number of tasks (generally one for critical sections) to "hold" the semaphore at one time. Any more tasks requesting to hold the **semaphore** are blocked (put to sleep).
- A **semaphore** can be used for serialization only in code that is allowed to block.
Operations for manipulating semaphores:

- **up** – release the semaphore
- **down** – get the semaphore (can block)
- **down_interruptible** – get the semaphore, but return whether we blocked
- **down_trylock** – try to get the semaphore without blocking, otherwise return an error
Semaphores

- optimized assembly code for normal case (\texttt{down()})
  - C code for slower “contended” case (\texttt{__down()})

- \texttt{up()} is easy
  - atomically increment; \texttt{wake_up()} if necessary

- \texttt{uncontended down()} is easy
  - atomically decrement; continue

- \texttt{contended down()} is really complex!
  - basically increment sleepers and sleep
  - loop because of potentially concurrent ups/downs

- still in \texttt{down()} path when lock is acquired
A **rw_semaphore** is a semaphore that allows either one writer or any number of readers (but not both at the same time) to hold it.

Any writer requesting to hold the **rw_semaphore** is blocked when there are readers holding it.

A **rw_semaphore** can be used for serialization only in code that is allowed to block. Both types of semaphores are the only synchronization objects that should be held when blocking.

Writers will not starve: once a writer arrives, readers queue behind it

Increases concurrency; introduced in 2.4
RW Semaphore Operations

- Operations for manipulating semaphores:
  - `up_read` – release a `rw_semaphore` held for read.
  - `up_write` – release a `rw_semaphore` held for write.
  - `down_read` – get a `rw_semaphore` for read (can block, if a writer is holding it)
  - `down_write` – get a `rw_semaphore` for write (can block, if one or more readers are holding it)
More RW Semaphore Ops

- Operations for manipulating semaphores:
  - `down_read_trylock` — try to get a `rw_semaphore` for read without blocking, otherwise return an error
  - `down_write_trylock` — try to get a `rw_semaphore` for write without blocking, otherwise return an error
  - `downgrade_write` — atomically release a `rw_semaphore` for write and acquire it for read (can't block)
A mutex is a data structure that is also used to synchronize access to critical sections or other resources, introduced in 2.6.16.

- Core difference: only 1 owner, while semaphores can have multiple owners
- Historically, semaphores have been used in the kernel, but now mutexes are encouraged, unless counting feature is really required
- As of 2.6.26, major effort to eliminate semaphores completely, and may eventually disappear
- Replace remaining instances with completions
Why Mutexes?

Pros
- Simpler (lighter weight)
- Tighter code
- Slightly faster, better scalability
- No fastpath tradeoffs
- Debug support – strict checking of adhering to semantics (if compiled in)

Cons
- Not the same as semaphores
- Cannot be used from interrupt context
- Owner must release
Mutex Operations

- Operations for manipulating mutexes:
  - `mutex_unlock` – release the mutex
  - `mutex_lock` – get the mutex (can block)
  - `mutex_lock_interruptible` – get the mutex, but allow interrupts
  - `mutex_trylock` – try to get the mutex without blocking, otherwise return an error
  - `mutex_is_locked` – determine if mutex is locked
Real-Time Mutexes

- Implement priority inheritance to solve priority inversion
  - `task_struct->prio` will be adjusted to the highest of priorities of waiters
  - `task_struct->prio` will be set back to its normal priority
Completions

- Higher-level means of waiting for events
- Optimized for contended case

```c
init_completion       // replaces sema_init
complete             // replaces up
wait_for_completion  // replaces down
wait_for_completion_interruptible
wait_for_completion_timeout
wait_for_completion_interruptible_timeout
```
For serialization that is not performance sensitive, the big kernel lock (BKL) was used

- This mechanism is historical and should generally be avoided.
- The function `lock_kernel` gets the big kernel lock.
- The function `unlock_kernel` releases the big kernel lock.
- The function `kernel_locked` returns whether the kernel lock is currently held by the current task.
- The big kernel lock itself is a simple lock called `kernel_flag`.

The Big Kernel Lock (BKL)
Read-Copy-Update Locks

- RCU is an alternative to a readers-writer lock
  - Extremely low overhead
  - Wait-free reads
Read-Copy-Update Locks

Diagram:

1. Readers?
2. list_del_rcu()
3. wait for readers
4. free()
There are four readers reading on an old version of a RCU protected data, which is removed but not reclaimed. So the grace period must extend to the point that the last readers finish its reading on it. The readers, which begin their reading at the time after the grace period begins, will not see the old version of the removed object; they have nothing with the grace period.

There is a trick on the classic RCU to determine when to really reclaim the old versions of the data structure. Since it is not allowed to explicitly block or sleep in the classic RCU, RCU determine the finish of existing readers by CPU context switch, which means, if each CPU executes at least one context switch, it is guaranteed that all the existing critical sections are quitted, and then the old version can be reclaimed safely.

2.3 (Maintain (Multiple (Version of Recently Updated Objects)

Readers may see different versions of a RCU protected data structure when reads and updates occur concurrently, this is depending on the time of the readers begin their critical section. Examples are showed in our presentation, so it won't be repeated here.

3. (Characteristics and Usage of RCU

Key points need to know:

1. In this paper, we are mainly discussing the implementation and characteristic of RCU under the non-COMPRESS! Linux kernel. There are several approaches of Linux kernel, but when we use RCU primitives, the primitives would automatically disable the!
Read-Copy-Update Locks

**READER**
- `rcu_dereference()`
- `rcu_assign_pointer()`
- `rcu_read_lock()`
- `rcu_read_unlock()`

**UPDATER**
- `call_rcu()`
- `synchronize_rcu()`

**RECLAIMER**
- `preempt-disable only if preemptible kernel`

**Notes**
- `rmb` only on DEC Alpha
- `wmb`
- `mutex for concurrent updaters`