- **Q1: What is synchronization?**
  - Synchronization is the coordination of concurrent operations so that the interleaving of their execution follows a well-defined ordering—as opposed to being non-deterministic, potentially introducing erroneous program behaviour at runtime.

- **Q2: When do we need synchronization?**
  - Given two operations p1 and p2 which are executed concurrently and have a dependency (e.,g., p2 consumes the result produced by p1) such that p1 must always "happen before" p2, synchronization mandates that $t_1(i) < t_2(i)\ \forall\ i \in [1, n]$ where $t_1(i)$ is the time when p1 completes its i-th execution instance and $t_2(i)$ is the time when p2 starts its i-th execution instance. In other words, synchronization is necessary to serialize all executions of two (or more) interdependent concurrent operations so that they adhere to a well-defined ordering of execution.

- **Q3: What goes wrong without synchronization?**
  - Without synchronization, there is no mandate on the execution interleaving of concurrent operations, and this may lead to erroneous program behavior.
  - **Race conditions** are among the most notorious software errors, which occur when the execution interleaving of concurrent operations leads to erroneous program behavior.
  - Race conditions manifest mostly due to **data races**, **semantic ordering errors**, or **weak memory consistency models**.
    - **Data races**: A program contains a data race *iif* two different threads (1) access the same memory location concurrently, (2) at least one of these accesses is a write and at least one of the accesses is not atomic, and (3) neither happens before the other—i.e., there is no "happens-before" relationship. Any such data race results in undefined program behavior and may lead to a data race error. (See the ISO/IEC 9899:2011(C11), Section 5.1.2.4/25, on multi-threaded executions and data races.)

Race condition due to data race

```c
int total = 0;

void *add(void *arg) {
  for (int i = 0; i < 1e6; ++i)
    ++total;
  return NULL;
}

void main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Total-1: %d\n", total);

  total = 0;
  pthread_create(&t1, NULL, add, (void *) NULL);
  pthread_join(t1, NULL);
  pthread_create(&t2, NULL, add, (void *) NULL);
  pthread_join(t2, NULL);
  printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:
 1159: push %rbp                  # Save base pointer to stack
 115a: mov  %rsp, %rbp            # Set up new stack frame
 115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
 1161: movl $0x0, -0x4(%rbp)      # i = 0
 1168: jmp  117d <add+0x24>       # for-loop start

                                  Data race!
 116a: mov  0x2ebc(%rip), %eax    #  %eax ← total
 1170: add  $0x1, %eax            #  %eax += 1
 1173: mov  %eax, 0x2eb3(%rip)    #  total ← %eax

 1179: addl $0x1, -0x4(%rbp)      # i += 1
 117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
 1184: jle  116a <add+0x11>       # for-loop jump
 1186: mov  $0x0, %eax            # rval = %eax
 118b: pop  rbp                   # Restore stack
 118c: ret                        # Return to caller
```

```
➜ git:(master) ✗ ./counter
Total-1: 1011367
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1011367
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1028085
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1011197
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1018502
Total-2: 2000000

➜ git:(master) ✗ ./counter
Total-1: 1013853
Total-2: 2000000
```

- **Semantic ordering errors**: A program contains a semantic ordering error if, at runtime, there exists some feasible interleaving of its concurrent operations that leads to unintended, erroneous program behavior.

Benign race condition due to semantic ordering error (w/o data race)

```c
void* func1(void* arg) {
  printf("1\n");
  return NULL;
}
void* func2(void* arg) {
  printf("2\n");
  return NULL;
}
int main(void) {
  pthread_t t1, t2;

  pthread_create(&t1, NULL, func1, NULL);
  pthread_create(&t2, NULL, func2, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return 0;
}
```

```
➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2

➜ concurrency git:(master) ✗ ./nosync
1
2
```

- Semantic ordering errors are race conditions that do not necessarily involve a data race: Even when there is no access to the same memory location, if there is no synchronization mechanism to mandate the execution interleaving of concurrent operations, then a race condition may occur as a result of undefined program behavior.
- Undefined program behavior is usually, but not always, problematic: What will the following program print?
- The above program's undefined behavior does not seem too harmful; however, there have been cases where a similar pattern of undefined program behavior had detrimental consequences.
- A notorious example is the Therac-25 computerized radiation therapy machine, designed to treat cancer patients by delivering precisely calibrated doses of radiation: Between 1985 and 1987, at least six patients (at the Kennestone Regional Oncology Center, Marietta, GE, USA) received massive overdoses of radiation, leading to severe injuries and deaths. (See here for more details.)
- **How did this happen?**
  - Due to the lack of proper synchronization in the software controlling the machine, an unexpected input by the operator controlling the machine could lead to an ordering error where the high-power radiation beam mode was set (p1) after the filter control was determined and finalized (p2): That is, for the operations p1 and p2 with an implicit "happens-before" relationship, there could be cases where $t_1(1)$ [mode setting] > $t_2(1)$ [filter setting dependent on mode setting] did not hold.
  - Indeed, the software had more than one bugs that could lead to erroneous behavior: As shown below, there is a data race on the "mode" flag (and a secondary data race on the "input_finalized" flag) as well as a logical error because changing the "mode" flag must not be allowed after the "input_finalized" flag was set.

Serious race condition due to semantic ordering error (w/ data race)

```c
int mode = 0;              // 1: Low-power; 2: High-power
int filter_engage = 0;     // 0: Filter not engaged; 1: Filter engaged
int input_finalized = 0;   // Has operator finished input?

void *filter_control(void *arg) {
  while (!input_finalized) {
    sched_yield();
  }
  usleep(100);
  if (mode == 2)
    filter_engage = 1;
  else
    filter_engage = 0;
  return NULL;
}

void beam_activate() {
  if (mode == 2 && filter_engage == 0)
    printf("🧯 Treatment with high-power beam and no filter in place\n");
  else
    printf("🟢 Safe setup\n");
}

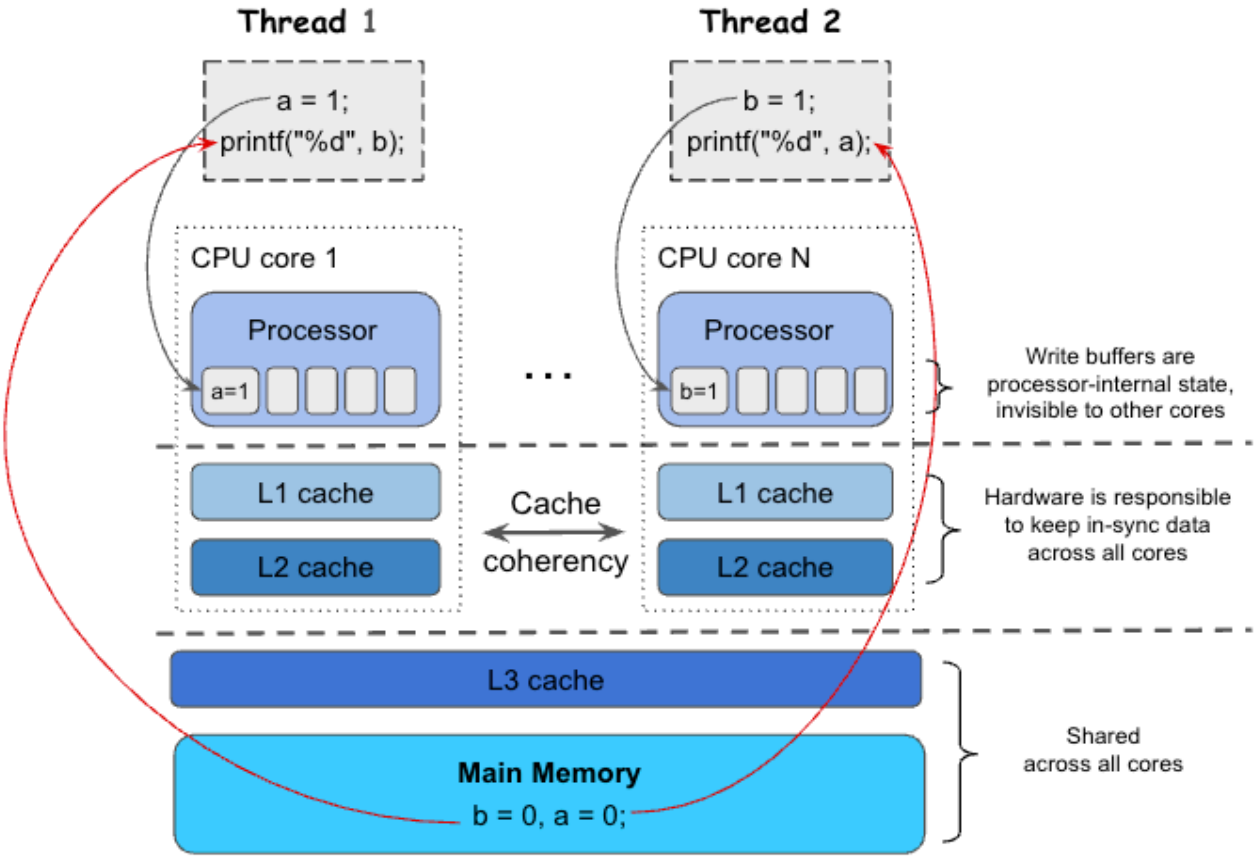
int main(void) {
  pthread_t filter_control_t;
  pthread_create(&filter_control_t, NULL, filter_control, NULL);
  usleep(100);  // Time window 1:  operator does  initial setup
  mode = 1;
  input_finalized = 1;
  usleep(100); // Time window 2: operator does  final edits
  mode = 2;
  pthread_join(filter_control_t, NULL); // Control logic completed
  beam_activate();
}
```

```
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🟢 Safe setup.
➜ concurrency git:(master) ✗ ./therac25
🧯 Treatment with high-power beam and no filter in place :-(
```

- **Weak memory consistency models:** In order to hide store latency and avoid processor stalls, modern multi-processor CPUs use write buffers to internally hold write operations until the memory system can process them—that is, until the cache line has read-write coherence permissions. This hardware-level optimization introduces a memory model (i.e., a set of allowed behaviors) that violates the well-defined semantics of sequential consistency and, without synchronization, it allows unforeseen reorderings of operations that may lead to erroneous program behavior at runtime.
  - **Sequential consistency (SC)** [easy to reason; impractically slow]
    - Every load from a memory address gets its value from the last store before it to the same address in global memory. (See "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," by Leslie Lamport on Sept. 1979.)
    - To achieve SC the hardware must wait for the effects of each instruction to become visible on all cores before starting the next instruction. (The first level of shared memory is the L3 cache with an overhead of at least 30 cycles for access time.)

- **Total Store Order (TSO)** [fast; more difficult to reason: departs from SC]
  - Modern multi-core CPUs implement cache coherency protocols so that core-local caches (e.g., L1 and L2 caches) are coherent across all cores, and no processor uses outdated data. Therefore, for a store to be committed to the core-local caches, it first needs to wait for the cache lines to be invalidated on all other cores.
  - To avoid this latency, modern CPUs use processor-internal First-in First-out (FIFO) write buffers to hold committed store operations before they complete (i.e., before being written to the cache).
  - This hardware-level optimization introduces a memory model that violates the semantics of SC, because, although it enforces most of the ordering conditions for a memory model to be considered sequentially consistent, it allows "store → load" reorderings between different cores.
  - Assume variables a and b are zero-initialized: can the following program print "00"?

Example of how TSO violates SC semantics and leads to race conditions



- **Q4: How do we design synchronization primitives?**
  - Recall what we are trying to achieve: Ensure that, despite all sources of non-determinism, the execution interleaving of concurrent operations does not lead to unforeseen, erroneous program behavior at runtime.
  - Enforcing an explicit interleaving ordering among all concurrent operations of, say, multi-threaded programs is impossible without control on the dispatching of asynchronous events (such as interrupts) that hint the OS kernel to preempt a blocked execution context (e.g., a process or thread that is waiting for I/O) and execute another one that is ready to run.

- Even if the OS kernel were to give us such control, reasoning about all possible scheduler interleavings would be impractically hard because the state space is enormous. (Completely disabling preemption and running each task to completion could make the state space tractable, but a system that does not support concurrency would appear prohibitively non-responsive, and thus unpleasant to use.)
- A practical compromise to design correct concurrent programs would be to, somehow, obtain a guarantee that some "happens-before" relationship exists among the execution interleaving of their concurrent operations at runtime. That is, that no concurrent operation starts in the middle of another, leaving the previous one in a partial or incomplete state.
    - This guarantee may seem trivial; yet, it addresses one of the main sources of race conditions—namely, data races. Recall the definition of a data race: "...at least one concurrent memory access is not atomic, and neither happens before the other…"
    - Therefore, if we establish that among concurrent operations involving accesses on a shared memory location, one of them every time executes to completion before any other starts, then, by definition, no data race exists among none of them.
    - Although we do not explicitly enforce a particular "happens-before" ordering in the interleaving of operations, it is guaranteed that one such ordering exists.
- **[Mutual exclusion / Atomicity]** The fundamental design principle followed to implement programs whose concurrent operations are guaranteed to satisfy some "happens-before" interleaving ordering upon execution is called mutual exclusion.
    - Conceptually, it is as simple as respecting the following principle: Insofar as any thread operates on a shared resource (e.g., a memory location), no other thread is allowed to execute any operation on the same shared memory location.
    - Technically, the software primitive required to provide mutual exclusion is called **atomicity**: all concurrent operations, insofar as they start execution, it is guaranteed to execute to completion without being interrupted in an incomplete or partial state; or, don't start execution until it is guaranteed to execute to completion.
    - Atomicity requires hardware support: That is, **architecture-specific atomic instructions** that bundle together more than one memory-related operation, which are guaranteed to be executed as a whole to completion.
    - **Atomicity of individual instructions** (not useful for mutual exclusion; good info)
        - Most modern architectures guarantee that aligned writes of word-sized values (e.g., 32-bit or 64-bit) are atomic, and no other threads will ever see a "half-written" value, even if multiple writes occur concurrently.
        - However, if the operation isn't atomic, writing unaligned values (e.g., crossing a cache line or word boundary) may result in word tearing.
    - **Atomicity of composite instructions** (see the C11 standard for atomics)
        - x86_64 atomic instructions
            - **xadd [mem], reg**; Atomically adds the value or reg at [mem], and returns the old value of [mem]. (Solves data race; example-1 / Q3.)
            - **xchgb reg, [mem]**; Atomically exchange the values: assign the value of reg to [mem], and return the old value of [mem].

        - **cmpxchg [target], %rbx**; Compare and conditionally exchange.
            - If [target] == %rax then [target] := %rbx and %zf := 1, else rax := [target] and %zf = 0
    - aarch64 atomic instructions
        - **ldadd %w0, %w1, [target];** Atomic addition.
            - Atomic ([target] += %w0 and %w1 := old [target])
        - **cas %w0, %w1, [target];** Compare and conditionally swap.
            - w0 := [target]; if [target] == w0 then [target] := %w1
- Atomicity guarantees mutual exclusion for the memory-related instructions bundled together and solves data races among them, but does not allow us to express realistic operations because the bundled instructions are just very few.
- What can we do? Use atomic instructions as a low-level primitive to build a higher-level primitive (called locking) appropriate for mutual exclusion on arbitrarily long sequences of instructions (called critical sections).

- **[Critical sections / Locking]** A critical section is a sequence of instructions that access some shared state (such as a common memory location or a system's resource) and ought to be executed by only one thread at a time. In concurrent programs, critical sections prevent data races and ensure correct synchronization between threads.
    - We design critical sections with the following <u>requirements</u> in mind
        - **Safety (mutual exclusion):** At most one thread in the critical section.
        - **Liveness (progress):** If multiple threads attempt to enter the critical section, one must be allowed to proceed.
        - **Bounded wait (starvation-free)**: If a thread is waiting to enter the critical section, it must eventually enter.
    - We design critical sections with the following <u>desirable properties</u>
        - **Efficiency**: Don't unnecessarily waste resources while waiting; instead, voluntarily yield the CPU. That is, avoid busy waiting.
        - **Performance**: The overhead of entering and exiting the critical section is small relative to the work being done within it.
        - **Fair**: All threads must wait approximately the same outside of the critical section.
    - We implement critical sections using a software primitive called **locking:** A token-based synchronization scheme with the following properties
        - (1) While a thread cannot obtain the token, it waits outside the critical section, trying to get the token.
        - (2) When a thread gets the token, it enters the critical section and executes the corresponding instructions while everyone else waits outside the critical section.
        - (3) When a thread finishes executing instructions in the critical section, it releases the token for anyone else to get it.
- Critical sections and locking enable us to write multithreaded programs that take the most out of modern multiprocessor CPUs by dividing jobs into two parts: a critical section part that must be executed serialized, as if the hardware were designed for uniprogramming, and the remainder of the job that can be executed concurrently with support for multitasking.

Using locking to provide mutual exclusion on the critical section

```c
int total = 0;

void *add(void *arg) {

 for (int i = 0; i < 1e6; ++i) {
  pthread_mutex_lock(&l);
  ++total;
  pthread_mutex_unlock(&l);
 }

 return NULL;
}

void main() {
 pthread_t t1, t2;

 pthread_create(&t1, NULL, add, (void *) NULL);
 pthread_create(&t2, NULL, add, (void *) NULL);
 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 printf("Total-1: %d\n", total);

 total = 0;
 pthread_create(&t1, NULL, add, (void *) NULL);
 pthread_join(t1, NULL);
 pthread_create(&t2, NULL, add, (void *) NULL);
 pthread_join(t2, NULL);
 printf("Total-2: %d\n", total);
}
```

```
➜ obdjump -d ./counter

0000000000001159 <add>:
  1159: push %rbp                  # Save base pointer to stack
  115a: mov  %rsp, %rbp            # Set up new stack frame
  115d: mov  %rdi, -0x18(%rbp)     # *arg = %rdi
  1161: movl $0x0, -0x4(%rbp)      # i = 0
  1168: jmp  117d <add+0x24>       # for-loop start
  ..........................................................
  118e: lea  0x2eeb(%rip),%rax
  1195: mov  %rax,%rdi
  1198: call 1070 <pthread_mutex_lock@plt>     } Get the lock
  ..........................................................
  116a: mov  0x2ebc(%rip), %eax  #  %eax ← total
  1170: add  $0x1, %eax          #  %eax += 1
  1173: mov  %eax, 0x2eb3(%rip)  #  total ← %eax   } Critical section
  ..........................................................
  11ac: lea  0x2ecd(%rip),%rax
  11b3: mov  %rax,%rdi
  11b6: call 1040 <pthread_mutex_unlock@plt>   } Release the lock
  ..........................................................
  1179: addl $0x1, -0x4(%rbp)      # i += 1
  117d: cmpl $0xf423f, -0x4(%rbp)  # loop counter compare
  1184: jle  116a <add+0x11>       # for-loop jump
  1186: mov  $0x0, %eax            # rval = %eax
  118b: pop  rbp                   # Restore stack
  118c: ret                        # Return to caller
```

- **Q5: How do we implement locks?**
  - What we are trying to implement is an API that supports the following operations
    - Initialize a shared lock variable
    - Acquire (lock) the lock variable before entering the critical section
    - Release (unlock) the lock after exiting the critical section
  - Locks are a shared resource themselves, and therefore, in implementing them, we need
    - **A lock operation before entering the critical section:** Atomically checks the state of the lock, and if it is free, sets its state to indicate that the lock has been acquired.
    - **An unlock operation after exiting the critical section:** Resets the state of the lock to indicate that it has been released, and does so, in total program order—that is, the unlock operation is sequentially consistent with respect to all memory operations that precede it, to ensure all critical-section operations become visible before the lock appears released.

- The canonical way to implement a portable lock operation is with a macro called **test_and_set**, which atomically reads the value of a memory address and also write a value to it, in one go.
    - **Example test_and_set implementation in x86**

```
static inline uint8_t test_and_set(uint8_t *lock) {
    uint8_t old = 1;
    // xchgb: Exchanges 8-bit values (b = byte)
    // +q:  Tells the compiler that old is both an input and an output in a general register.
    // +m: Memory operand, also read-write.
    // "memory" clobber: Tells the compiler not to reorder memory around this inline
    __asm__ volatile (
        "xchgb %0, %1"
        : "+q" (old), "+m" (*lock)
        :
        : "memory"
    );
    return old;
}
```

- The canonical way to implement a portable unlock operation is with a class of macros that leverage architecture-specific instructions, called memory barriers, which explicitly prevent compiler- and processor-dependent reordering of memory operations.
    - **x86 memory barrier instructions** (relevant Linux kernel macros)
        - **mfencle**: Prevents reordering of loads and stores, and ensures that all memory reads and writes before the mfence, complete before any that follow.
        - **lfence:** Prevents reordering of loads before loads, and ensures all prior loads complete before any subsequent load.
        - **sfence:** Prevents reordering of stores before stores, and ensures all prior stores complete before any subsequent store.
    - **aarch64 memory barrier instructions** (relevant Linux kernel macros)
        - **DMB**: Prevents any memory reordering.
        - **DSB**: Prevents any memory reordering and waits for instruction completion.
        - **ISB**: Prevents instruction pipeline reordering
- With the above, we can implement a primal, but safe (i.e., guaranteed mutual exclusion), version of a lock, known as **a raw spin lock**.
    - **Raw spinlock implementation**

```
typedef uint8_t raw_spinlock_t; // Define a simple spinlock type
raw_spinlock_t lock = 0;          // 0 = unlocked, 1 = locked

void acquire(raw_spinlock_t *lock) {
        preempt_disable();
        while (test_and_set(lock)) ;  // Spin while the lock is not free
}

void release(raw_spinlock_t *lock) {
        __asm__ volatile("mfence" ::: "memory");  // Ensure memory ordering
        *lock = 0;
        preempt_enable();

}
```

- Q6: What are the most common types of locks?

- Depending on whether the locking implementation continuously tries to acquire the lock regardless of whether it is available or not, there are two types of locks: **spin locks** (i.e., those that spin until acquiring the target lock) and **sleeping locks** (i.e., those that voluntarily preempt themselves and release the processor when the target lock in not free).
- **Spin locks:** The raw spin lock implementation shown before is the most simplistic—yet correct—version of locks that is typically used as a primitive to build higher-level locking mechanisms addressing performance and fairness shortcomings.
  - Raw spinlocks have no explicit fairness guarantee: A thread may acquire the lock multiple times, preventing others from making fair progress.
  - Not appropriate for unicore CPUs: The thread spinning to get an unavailable lock unnecessarily consumes processor cycles (busy-waiting).
  - Lots of traffic over the memory bus (especially when there are multiple spinners).
  - Simple to implement: One memory location for arbitrarily many processors.
  - Easy to address a few of their above shortcomings as long as (1) the critical section is small (such that the overhead of context switching would be more) and (2) no sleep occurs while holding the lock.
    - Disable preemption before trying to acquire the lock ⇒ solves potential for deadlocks
    - Use a simple counter to keep track of whose thread the turn is to get the lock next ⇒ solves fairness
  - Sample spinlock implementation from the Linux kernel

    spinlock(...) // lock API
    raw_spinlock(...) // wrapper
    _raw_spinlock(...) // wrapper
    __raw_spinlock(...) // disables pre-emption
    do_raw_spinlock(...) // invokes arch-specific implementation
    arch_spinlock(...) // invokes queued spinlock
    queued_spinlock(...) // wrapper
    cmpxchg_acquire(...) // get the lock if free
    queued_spinlock_slowpath(...) // or enqueue
  - The actual implementation uses a queue to solve the fairness issue of raw spinlocks, and also disables preemption on the local processor.
  - Since preemption is disabled, a thread sleeping while holding a spin lock will not only hog the local processor but will also prevent progress of other threads (running on other processors) trying to acquire the lock. Furthermore, spinlocks may be used with local interrupts disabled, and sleeping while holding a spinlock may break real-time guarantees, making the system appear "not responsive;" or, even worse, cause deadlock.
- **Sleeping locks:** Contrary to spin locks, sleeping locks voluntarily yield the processor when the target lock cannot be acquired because it is not free.
  - Sleeping locks are appropriate for mutual exclusion on longer critical sections and also allow the use of functions that may potentially sleep.
  - Voluntarily yielding the processor when the target lock is not free solves the busy-waiting issue, but introduces the requirement to schedule back in the right execution context when/if the requested lock becomes available in the future. Otherwise, the implementation is not starvation-free.

- Internally, the implementation of sleeping locks uses a queue to keep track of blocked (sleeping) execution contexts waiting to be scheduled back in when the target lock becomes available.
- The internal state of sleeping locks (e.g., the queue keeping track of who is waiting on the lock to become available) must be protected by a spinlock—and this is how "spinlocks are used as a low-level primitive to implement higher-level locking mechanisms."
- The most typical sleeping lock in the Linux kernel is what is called a **mutex**, and is implemented here.

Example mutex implementation

```
// Define a mutex type
typedef struct mutex_t {
  int lock = 0;
  raw_spinlock_t guard = 0;
  queue_t queue = NULL;
 } mt;

void acquire(mutex_t *lock) {
  spin_lock(mt->guard)

  // Lock is being held; voluntarily yield the processor
  if (!mt-> lock == 0) {
     enqueue(mt->queue, self)
     spin_unlock(mt->guard)
     sched_yield();
  } else {
     mt->lock = 1
     spin_unlock(mt->guard)
   }
}

void release(mutex_t *lock) {
   spin_lock(mt->guard)

   // Release the lock and let other know
   mt->lock = 0
   if (!queue_empty(mt->queue)) {
     wake_up(dequeue(mt->queue))
  }
  spin_unlock(mt->guard)
}
```

- Q7: Hybrid user-space/kernel-space locks in Linux?

- All the lock implementations discussed so far require involvement of the OS kernel, and therefore their use from user space will neccessarily incur the additional overhead of performing syscalls.
- **Can we avoid this?** Recall why / when the kernel's involvement is necessary
    - In the case of spinlocks, the kernel needs to keep track of the spinners so that the implementation has some minimum fairness guarantees.
    - In the case of sleeping lock, the kernel needs to keep track of the preempted (sleeping) threads in order to wake them up accordingly when the lock becomes available.
- The key observation is that when there is no contention on a target lock and the requesting thread does not need to be put on a wait queue, then, the kernel's involvement is not necessary.
- A hybrid user-space/kernel-space could be implemented such that it atomically test and modifies in user-space a value indicating that the lock was acquired if it were free; and only fall back to kernel-space functionality when contention arises and the requesting thead must be preempted.
- The futex is special Linux primitive that allows user-space programs to sleep based on the value of a user-space memory address without forcing a kernel-space lock object. (See "Fast Userlevel Locking in Linux," by H. Franke and R. Russell at Ottawa Linux Symposium, 2002.)
    - Drastically cuts the overhead compared to using a purely syscall-based locking primitive.
    - With futex, the actual lock state is in userspace memory (atomic_int, etc.)
    - The kernel only needs to be aware of this memory address, and only check for a value change—which requires not special functionality or per-task state to be tracked.
- The two most important operations exposed by the futex primitive are futex_wait and futex_wake.
    - **futex_wait(addr, val)** blocks while the value at the user-space memory address addr equals to the integer value val.
    - **futex_wake(addr, num)** wakes up up to num threads blocked waiting for a value changed on the value stored at the user-space memory address addr.
    - Since addr. is a virtual address of a shared variable, many processes may map the same shared memory at different virtual addresses and the futex implementation cannot rely on addr. to correlate futex objects with the corresponding user-space virtual addresses. Therefore, internally, the futex implementation converts addr into a (physical page frame, offset) pair to uniquely identify futexes across processes, since the underlying physical location backing the shared variable would be unique system-wide.

Example futex, implementation and use

```c
struct futex_q {
  struct task_struct *task;
  struct plist_node list;
  int *uaddr;
};

struct futex_hash_bucket {
  spinlock_t lock;
  struct plist_head chain;
};

static struct futex_hash_bucket *futex_hash(unsigned long addr);

// If the value at addr is val, put the current thread to sleep
int futex_wait(int *uaddr, int expected) {
  int val;
  struct futex_hash_bucket *hb = hash_futex(uaddr);

  spin_lock(&hb->lock);
  val = atomic_read(uaddr);
  if (val != expected)
    return -EAGAIN;

  struct futex_q q = {
    .task = current,
    .uaddr = uaddr
  };
  enqueue(&hb->queue, &q);
  set_current_state(TASK_INTERRUPTIBLE);
  spin_unlock(&hb->lock);
  schedule(); // Sleep until woken

  spin_lock(&hb->lock);
  dequeue(&hb->queue, &q);
  spin_unlock(&hb->lock);
  return 0;
}

// Wake up nr_wake  threads waiting on addr
int futex_wake(int *uaddr, int nr_wake) {
  struct futex_hash_bucket *hb = hash_futex(uaddr);
  int woken = 0;

  spin_lock(&hb->lock);
  list_for_each_entry(q, &hb->queue, list) {
    if (q->uaddr == uaddr) {
      wake_up_process(q->task);
      woken++;
      if (woken >= nr_wake)
        break;
    }
  }
  spin_unlock(&hb->lock);
  return woken;
}
```

```c
typedef struct {
  atomic_int lock; // 0 = unlocked, 1 = locked
} mutex_t;

void mutex_init(mutex_t *m) {
  atomic_store(&m->lock, 0);
}

void mutex_lock_slow(mutex_t *m) {
  int expected;

  while (1) {
    expected = 0
    futex_wait(&m->lock, 1);  // Wait  until lock is released
    if (atomic_compate_exchange_strong(&m->lock, &expected, 1))
      return;
  }
}

void mutex_lock_fast(mutex_t *m) {
  int expected = 0;

  if (atomic_compate_exchange_strong(&m->lock, &expected, 1))
    return;

  while (1) {
    expected = 0;
    futex_wait(&m->lock, 1);  // Wait  until lock is released
    if (atomic_compate_exchange_strong(&m->lock, &expected, 1))
      return;
  }
}

void mutex_unlock(mutex_t *m) {
  atomic_store(&m->lock, 0);
  futex_wake(&m->lock, 1);  // Wake one waiter
}
```

```
➜ git:(master) ✗ ./mutex_futex 1000000 1
Using 1 threads to perform 1000000 lock-unlock operations
contented: 0, uncontented: 1000010, total: 1000010
Median overhead of slow vs fast futex-based lock: +119.7% (±11.0% 95% CI)

➜ git:(master) ✗ ./mutex_futex 100000 4
Using 4 threads to perform 1000000 lock-unlock operations
contented: 8541135, uncontented: 31458875, total: 4000010
Median overhead of slow vs fast futex-based lock: +78.4% (±2.5% 95% CI)
```

- **Q8: What is fine-grained locking?**
  - Where are we with locks, so far?
    - Putting to sleep a thread trying to lock an unavailable lock and waking it up when the target lock becomes available addresses the busy-waiting problem.
    - Waking up the appropriate thread is implemented by maintaining a per-lock queue of waiting threads, which guarantees fair treatment among competing threads.
    - When the cost of context switching is more than the cost of the commands within the critical section, the use of a spin lock is preferable instead of a sleeping lock
    - When there is no contention (i.e., the lock is free), keep the fast path in user space to avoid making an unnecessary user-to-kernel mode switch (and back).
  - Spin locks and sleeping locks are coarse-grained locks in the sense that they give the requesting thread no ability to express whether the target lock will be obtained for reading or for writing.
  - The semantics of the operation within the critical section can help decrease contention if, say, for example, some data is read more often than written because multiple tasks wish to search on a data structure and do not necessarily wish to mutate it.
  - **A read-write spin lock (rwlock)** is the most typical fine-grained lock, which allows multiple tasks to hold it in a "read" state, or one task to hold it in a "write" state.
    - Readers do not need to serialize access with each other: If only readers exist, the path is lock-free with minimal contention to mark the "read" state of the lock..
    - Writers serialize access with other writers as well as with readers: If one or more writers exist, there is contention among writers with each other as well as with readers.

Example read-write spin lock, implementation and use

```
typedef struct {
  spinlock_t write_lock;
  volatile int readers;
} rw_spinlock_t;

void rwlock_init(rw_spinlock_t *lock) {
  lock->write_lock = 0;
  lock->readers = 0;
}

void read_lock(rw_spinlock_t *lock) {
  while (1) {
    while (lock->write_lock)  // Wait until no writer is holding the lock
          ;

    // Use  lock xadd to atomically increment the reader's count
    __sync_fetch_and_add(&lock->readers, 1);

    // Re-check in case a writer acquired the lock, and backtrack
    if (lock->write_lock) {
      __sync_fetch_and_sub(&lock->readers, 1);
      continue;
    }

    break;                    // Acquired read lock successfully
  }
}

void read_unlock(rw_spinlock_t *lock) {
  __sync_fetch_and_sub(&lock->reader_count, 1);
}

void write_lock(rw_spinlock_t *lock) {
   // Spin until we get the lock
  while (test_and_set(&lock->write_lock)) ;

  // Wait for readers to drain
  while (lock->reader_count > 0) ;
}

void write_unlock(rw_spinlock_t *lock) {
   __asm__ volatile("sfence" ::: "memory");
  lock->write_lock = 0;
}
```

```
// Define a read-write spinlock type
static rw_spinlock_t my_lock;
rwlock_init(&lock);

int value = 123;

void reader(void)
{
  struct my_data *p;

  read_lock(&my_lock);              // Acquire read lock
  printk("Value = %d\n", value);    // Shared  access
  read_unlock(&my_lock);            // Release lock
}

void writer(void *arg)
{
  write_lock(&my_lock);     // Acquire write lock
  value = *(int*)  arg;     // Exclusive access
  write_unlock(&my_lock);   // Release lock
}


int main() {
 pthread_t  t1, t2, t3, t4;
 int val = 10;

 pthread_create(&t1, NULL, reader, (void *) NULL);
 pthread_create(&t2, NULL, writter, (void *) &val);
 val += 1;
 pthread_create(&t3, NULL, writter, (void *)&val);
 pthread_create(&t4, NULL, reader, (void *) NULL);

 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 pthread_join(t3, NULL);
 pthread_join(t4, NULL);
}
```

- **Q9: How is lockless synchronization implemented in the Linux kernel?**
  - Fine-grained read-write locks decrease contention because, when only readers exist, they don't block each other and can all acquire the read lock concurrently. However, some contention still remains, and as the number of readers increases, performance degrades.
    - There is a global counter for readers: __sync_fetch_and_add(&lock->readers, 1) targets the same memory location ⇒ On multi-core or NUMA systems, this causes cache-line bouncing and leads to slowdowns as reader threads scale up.
    - One writer can block all readers: If a writer holds the lock, all incoming readers spin on while (lock->write_lock) ⇒ contending on a shared variable.
    - Atomic operations still serialize under the hood: Even though readers don't "lock," each atomic add/sub still causes coherence traffic and cache-line invalidations between cores.
  - To further minimize contention, the Linux kernel uses a special synchronization primitive called **Read-Copy-Update (RCU):** RCU allows readers to access data concurrently without locks, while ensuring safe updates by writers. (Detailed description here.)
  - Fundamentally, RCU is not a traditional lock but rather a lockless synchronization mechanism.
    - Readers: rcu_read_lock() and rcu_read_unlock() are extremely lightweight, safe to use in interrupt context and preemption-disabled sections ⇒ Multiple readers can execute entirely concurrently, and the use of thread-local storage prevents cache bouncing.
    - Writers**:** Allocate and initialize a new copy of the data, and then call synchronize_rcu() to wait until all pre-existing readers have finished.
  - In the following toy example, the RCU API enforces correctness on readers' concurrency and memory ordering, but a spinlock must be used to protect writers' concurrency. [Q: Why?]

Example RCU lock, implementation and use

```
#define MAX_THREADS 16

#define  rcu_assign_pointer(p, v) \
  atomic_store_explicit(&(p), (v), memory_order_release)

#define  rcu_dereference(p) \
  atomic_load_explicit(&(p), memory_order_acquire)

// Thread-local index
__thread int rcu_thread_id;

// Reader state array
atomic_int readers_state[MAX_THREADS];

void rcu_read_lock() {
  __sync_fetch_and_set(&readers_state[rcu_thread_id], 1);
}

void rcu_read_unlock() {
  __sync_fetch_and_set(&readers_state[rcu_thread_id], 0);
}

void synchronize_lock() {
 for (inti = 0; i < MAX_THREADS; ++i) {
    // Wait for all reader threads to exit their critical section
    while (__sync_fetch_and_add(&readers_state[i], 0)) {
      sched_yield();
    }
  }
}
```

```
struct data {
  int val;
  struct rcu_head rcu;
};
static struct data __rcu *global_ptr;

raw_spinlock_t writers_lock = SPIN_LOCK_UNLOCKED;

void reader(void)
{
  rcu_read_lock();          // Start read-side critical section
  printk("Value = %d\n", ptr->value);
  rcu_read_unlock();       // End read-side critical section
}

void writer(void)
{
  struct data *old, *new;

  new = kmalloc(sizeof(*new), GFP_KERNEL);
  new->value = 42;

  spin_lock(&writer_lock);                    // Only one writer
  old = rcu_dereference(global_ptr);          // Get current
  rcu_assign_pointer(global_ptr, new);        // Publish new
  synchronize_rcu();                          // Wait for readers
  spin_unlock(&writer_lock);

  kfree(old);   // Free old value safely
}
```

- Q10: What is a deadlock?

- The kernel and the respective system libraries are responsible for the correctness of the locking mechanism implementations, while programmers are responsible for using locks correctly.
- The most common error when using locks is what is called **a deadlock**: an unrecoverable error where all members of some group of entities (e.g., threads or proccesses) wait indefinitely on each other and cannot make progress because each waits for another member of the group, including itself, to take action, such as, for example, releasing a lock.
- There are four necessary preconditions for a deadlock on a shared resource to occur, known as the Coffman conditions. (See "Systems Deadlocks," by E.G.Coffman.)
- Without loss of generality, we describe the four aforementioned preconditions assuming the competing entities are threads.
    - 1.) **Mutual Exclusion:** The shared resource can only be held by one thread at a time.
    - 2.) **No preemption:** Once the shared resource is obtained by a thread, it cannot be taken away from it involuntarily.
    - 3.) **Hold and Wait:** A thread holding a shared resource is also requesting/waiting for additional resources, which are being held by other threads.
    - 4**.) Circular Wait:** A circular wait occurs when a cycle forms in the resource allocation graph. In this scenario, each thread is waiting for a resource that is currently held by another thread, creating a closed loop of dependencies, which prevents any thread in the cycle from progressing and results in a deadlock. Formally, there exists a set of waiting threads, $T = \{T_1, T_2, ..., T_n\}$, such that T_1 is waiting for a resource held by T_2, T_2 is waiting for a resource held by T_3, …, and T_n is waiting for a resource held by T_1.
- **Proactive prevention** (by eliminating one of the four necessary preconditions)
    - No mutual exclusion: No thread gets exclusive access to the shared resource.
    - Enable preemption: No thread can hold the shared resource for more than a given time frame while others are trying to obtain it.
    - Avoid hold and wait: No thread can hold the shared resource while requesting another, and must, instead, try to obtain all the necessary resources at once, at the beginning.
    - Avoid circular waiting: Impose a hierarchical ordering on shared resource acquisition, followed by all threads, under which a shared resource is requested only if all others that precede it in the hierarchy are being held. (In practice, when writing code that acquires multiple locks, developers acquire them in a well-documented hierarchical order.)
- **Reactive prevention** (by detection)
    - In general, verifying the correctness of concurrent programs with respect to deadlocks is a very difficult problem, which has been studied for many decades. The core of its difficulty emanates from the enormous state space formed by all feasible interleavings of multiple threads executing concurrent operations on shared resources.
    - The most practical approach to uncovering deadlocks in programs is to first perform what is called a state space reduction—by considering only the interleaving of interdependent operations as part of the target state space, and ignoring the rest—and then executing the program under test in the reduced state space, hoping to bring it into a deadlock-erroneous state, if such exists. (See "Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem," by P. Godefroid.)
- **Overall**, using locks correctly is non-trivial: Prefer implementations with timeouts (preemption⇒ the user eventually gets some feedback) and follow a well-documented hierarchical acquisition.