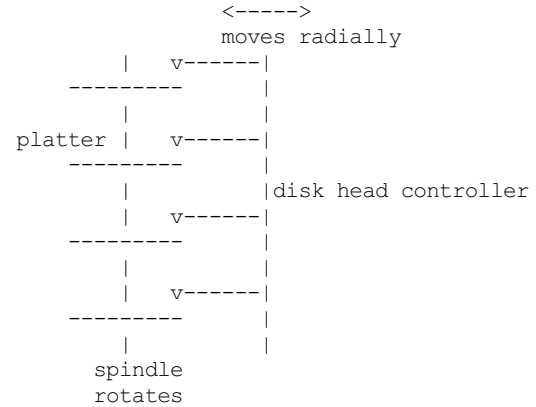
An Example of a Webserver

Consider a webserver that receives an HTTP request (e.g., fetch URL), and replies with the contents of the corresponding HTML file. Also, let's say that the HTML files are stored in the server's disk. Multiple clients can connect to the webserver simultaneously.

Let us first understand the typical hardware characteristics. The webserver is possibly running on a modern processor executing at 1-2 GHz frequency, i.e., each instruction executed locally on the CPU (e.g., register access, L1 cache hit, etc.) takes around 1-2 nanoseconds to complete. Instructions that miss the cache, and thus need to access the main memory take 10-100 nanoseconds. In contrast, a disk access takes roughly 1-10 milliseconds to complete, i.e., around one million instructions can execute in the time it takes to complete one disk access!

Why is a disk access so expensive? As we will also see later in our discussion on file systems, a magnetic disk is a mechanical device organized as a cylindrical spindle, with multiple circular platters stacked one above another. Each platter in the cylinder has concentric tracks, which hold the data as magnetic impressions. To read this data, a magnetic disk has a *disk head* associated with each platter that can move radially and position itself on a particular track. The cylindrical spindle rotates around its central axis so the head can read data off the track, as the spindle rotates.

Notice that reading data involves mechanical radial movement of the disk head arm, and also requires the head to be positioned at the right point within a track, which involves a rotational latency. The time taken for the radial movement of the head to reach the desired track, is called the *seek time*. Typically, average seek times range between 4 milliseconds (for enterprise drives) to 15ms (for mobile drives). Typical disks rotate at 5000 RPM (for mobile drives) to 15000 RPM (for enterprise drives), and so the average rotational latencies range between 2-5 milliseconds. Overall a disk access can take 5-20 milliseconds to complete.



Let's look at the pseudo-code of a webserver:

```
while (1) {
  read message from incoming network queue;
  URL = parse message;
  read URL file from disk;
  write reply on outgoing network queue;
}
```

Here we assume that there is an incoming network queue, where incoming packets are buffered, and read in FIFO order. If multiple clients are accessing the webserver, their requests will get buffered in the incoming queue in arrival order. This webserver picks and serves one request at a time. Because, each request needs to go to the disk device, and assuming each disk request takes 10 milliseconds, we can serve 100 requests per second. Modern webservers serve tens of thousands of requests per second. We can make this system faster by introducing a cache of disk files in memory.

```
while (1) {
  read message from incoming network queue;
  URL = parse message;
  if (requested file is in cache) {
    data = read URL file from disk;
    new.name = URL;      //new is the current cache pointer.
    new.data = data;
    new = new + 1;       //mark the current cache pointer as used.
  }
  write reply on outgoing network queue;
}
```

This server is much better, as we have optimized for the common case where there is locality of access, i.e., the same set of pages are accessed close in time, and most of these accesses can be served from memory.

However, still, if even one request results in a disk access (cache miss), then all the following requests in the network queue will have to wait for a long time (even though those requests could have been served much faster from memory). So, let's use multiple threads, each serving one request.

```
for (i = 0; i < 10; i++) {
  fork(server);
}
```

```
server() {
  while (1) {
    read message from incoming network queue;
    URL = parse message;
    if (requested file is in cache) {                  // A
      data = read URL file from disk;
      //new is the current cache pointer.
      new.name = URL;                                  // B
      new.data = data;                                 // C
      //mark the current cache pointer as used.
      new = new + 1;                                   // D
    }
    write reply on outgoing network queue;
  }
}
```

Each thread executes the server function, but now we have concurrency problems. Firstly, assuming there is only one network queue, multiple server threads will be reading from the same input queue, and need correct synchronization for that. We will be discussing this later. Similarly, the output network queue is shared by multiple server threads. Let's however first focus our attention to the shared cache, accesses to which are made in statements A, B, C, and D.

Many race conditions exist in this code and many bad things can happen. We can have a cache with an empty entry, or a bad entry, or worse still, a page belonging to URL1, cached under URL2.

One way to solve this problem is to use a global cache lock, and instrument the entire critical section with it (statements A, B, C, and D). However, that brings us back to the same problem (which we wanted to solve using multiple threads), which is that even if one thread misses the cache and accesses the disk, all other concurrent threads will have to wait (on the lock acquire statement) for a long time. (Notice that we have just changed the waiting point from the input queue to the lock acquisition). So, a full solution will require fine-grained locking (e.g., lock per file/page, a lock for the disk, etc.), and will need careful lock acquisition for correctness.

Let's now look at the shared network queues. On the incoming queue, data (or packets) is *produced* by a "network thread" (or multiple network threads) and *consumed* by the "server threads". The network thread reads the incoming packets from the network wire and appends it to the queue's buffer (which is consumed in FIFO order by the server thread). Similarly, on the outgoing queue, data is produced by the server threads, and consumed by another set of outgoing network threads. This is a common situation where there is a shared queue and there are producer and consumer threads accessing this queue. This is also commonly known as the *producer-consumer* problem.

```
network() {
  packet = read from wire;
  queue.head = packet;
  queue.head++;
}

server() {
  packet = queue.tail;
  queue.tail++;
  read packet and process request;
}
```

Queues are often implemented as circular buffers, with `head` and `tail` pointers, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global

void produce(char data) {
  if ((head + 1) % MAX  ==  tail) {
    error("producing to full queue!");
    return;
  }
  queue[head] = data;
  head = (head + 1) % MAX;
}

char consume(void) {
  if (tail == head) {
    error("consuming from empty queue!");
    return -1;
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  return e;
}
```

The code above implements `produce()` and `consume()` functions for a queue implemented as a circular buffer. Additions to full queues, and consumption from empty queues are not allowed, in this case.

In a multi-threaded scenario, one of the threads may be consuming from the queue, while another thread may be producing to the queue. Also, if a

consumer tries to consume from an empty queue, it may want to wait till the queue becomes non-empty, and then consume an element. Similarly, if a producer wants to produce to a full queue, it may want to wait till the queue becomes not-full, and then produce the element, in the following way:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global

void produce(char data) {
  if ((head + 1) % MAX  ==  tail) {
    wait for queue to have some empty space;
  }
  queue[head] = data;
  head = (head + 1) % MAX;
}

char consume(void) {
  if (tail == head) {
    wait for queue to have some elements;
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  return e;
}
```

How can a thread wait for a condition to become true? For example, how does a thread wait for the queue to have some empty space? This requires some coordination between the producer and the consumer threads. Such coordination is not possible using locks, as locks only implement mutual exclusion. This is done using a special construct, called *condition variables*.

Just like locks, the condition variables are also variables, usually shared by multiple threads, and they provide functions called `wait()` and `notify`.

```
struct cv;   //condition variable type

void wait(struct cv *, struct lock *);

void notify(struct cv *);
```

(Let's ignore the second argument of the `wait` function, of type `lock`, for some time). The `wait` function causes the calling thread to get suspended (or blocked). It is woken up, only if another thread subsequently calls `notify` on the same condition variable.

In our producer-consumer example, the producer thread may wait on a condition variable, if it finds that the buffer is full. The consumer thread, after consuming an element, may call `notify` to wakeup any sleeping producer thread (as the queue may now be not-full), in the following way:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;

void produce(char data) {
  if ((head + 1) % MAX  ==  tail) {
    wait(&not_full, ...);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_empty);
}

char consume(void) {
  if (tail == head) {
    wait(&not_empty, ...);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_full);
  return e;
}
```

Here, we declared two condition variables, `not_full` and `not_empty`. If the queue is full, the producer thread waits on `not_full`. The consumer thread calls `notify(not_full)` if it consumes an element. The effect of `notify` is to wakeup all threads waiting on that condition variable. If no threads are currently waiting on that condition variable, `notify` does not have any effect. Also, notice that we have left the second argument of the `wait` calls blank for now.

Firstly, this code is incorrect because the threads are concurrently reading shared variables, `queue`, `head`, and `tail`, and so bad interleavings can cause incorrect results. We need to use locks to fix this. Secondly, we do not need to call `notify` on every produce and consume operation. Instead, notify only needs to be called if the queue transitioned from full to not-full in the case of the consumer (or empty to not-empty in the case of the producer). Let's try fixing this code, as follows:

```
char queue[MAX];  //global
```

```
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  if ((head + 1) % MAX  ==  tail) {
    wait(&not_full, ...);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  if (head == (tail + 1) % MAX) {   // the queue just became not-empty
    notify(&not_empty);
  }
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  if (tail == head) {
    wait(&not_empty, ...);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  if ((head + 2) % MAX == tail) {    // the queue just became not-full
    notify(&not_full);
  }
  release(&qlock);
  return e;
}
```

Notice that now, if a producer starts waiting on the condition variable, the system will deadlock, as it has acquired the lock and so no other thread will be able to access the queue and so the producer will never get notified. A better solution may perhaps be to release the lock before calling wait, and reacquire it afterwards, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  if ((head + 1) % MAX  ==  tail) {
    release(&qlock);
    wait(&not_full, ...);
    acquire(&qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_full);
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  if (tail == head) {
    release(&qlock);
    wait(&not_empty, ...);
    acquire(&qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_empty);
  release(&qlock);
  return e;
}
```

This code is also incorrect. Let's see what can happen: it is possible that the producer thread checks the queue, finds it full, and so releases the lock, and is about to call `wait()`. However, before the producer calls `wait()`, the consumer thread, can acquire the lock, consume an element, notify `not_full`, and release the lock. In this case, the notify call will be lost, as the producer has not called wait yet. When the producer calls wait after this, it will keep sleeping forever, as the consumer will not call notify again, resulting in a deadlock. This is also called the *lost-notify* problem.

The lost-notify problem occurred because the release of the lock and the act of going to sleep were not atomic. Hence, the `wait()` function takes an additional argument, `lock`. Internally, `wait()` releases the lock before going to sleep (in an atomic manner), and tries to reacquire the lock after being notified. The wait function ensures that no other thread can call notify between the release of the lock and the act of going to sleep. The (almost) correct code looks like the following:

```
char queue[MAX];  //global
```

```
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  if ((head + 1) % MAX  ==  tail) {
    wait(&not_full, &qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_full);
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  if (tail == head) {
    wait(&not_empty, &qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_empty);
  release(&qlock);
  return e;
}
```

Notice that, now we do not need the calls to `acquire()` and `release()` around the calls to `wait()`, as they are called internally by `wait` itself. In this code, if a producer finds the buffer full, it goes to sleep, releasing the lock atomically. A consumer which consumes an element can then call notify to wakeup the producer. The woken-up producer is now ready to run (though it may not immediately get the CPU). The first thing that the producer does is to try to reacquire the lock (inside wait). Because the lock may be already held by the consumer (notice that the call to `release(&qlock)` is after the call to `notify`), and so it may have to wait again. As soon as the lock is released, the producer may get the lock and can continue to consume the next element.

Semantics of wait() and notify()

`wait(cv, lock)` releases the `lock` and goes to sleep on `cv`. `notify(cv)` wakes up *all* threads that are currently waiting on that cv. After waking up, the `wait()` function tries to reacquire the `lock` before returning.