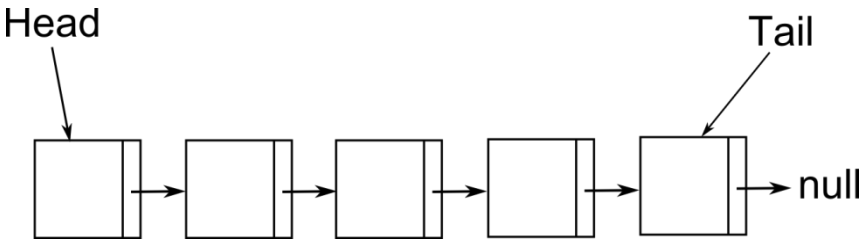
Thread Synchronization Lab

The producer/consumer type of problem is quite common in both the internal operating systems design as well as at the application level. Co-operating processes or one-to-many worker threads are representative cases. With multicore architectures becoming the standard in industry, efficient synchronization in this type of problems is getting more and more important.

In this lab you are required to:
Implement an unbounded multiple producer - multiple consumer concurrent queue in C using **PThreads mutex** locks.

Queue: Structure and Interface



You should implement a queue that:

- Is *linked-list* based
- Is FIFO
- maintains one Head and one Tail pointer for the first and last node respectively

The content of each node needs to be merely an integer and of course any appropriate link to the other nodes needed.
The prototypes of the functions you need to implement are:

- `void initialize_queue(void);`
  Initializes the queue structure and all its appropriate components (pointers, locks etc)
- `void enqueue(int val);`
  Enqueues a new node that contains the value `val` from the Tail.
- `int dequeue(int *extractedValue);`
  Dequeues a node from the Head, storing its value in the `*extractedValue`. Returns 0 if successful and 1 if the queue is found logically empty.

Concurrency and Synchronization

For ensuring correctness of your concurrent queue implementation locking should be used. Specifically *inside* the enqueue and dequeue methods the appropriate lock(s) should be acquired and released when necessary. In this lab we will consider two possible designs that will use one or two locks (more information in the Reporting section).

We suggest using the mutex locks described in the *Pthreads* standard. The man pages for pthreads is the official documentation, but several good tutorials exist in the web also (e.g. http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html). Useful keywords: `pthread_mutex_t`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `PTHREAD_MUTEX_INITIALIZER`.

Dummy node

Since this is going to be a concurrent queue where multiple threads will be enqueueing and dequeueing concurrently, it is useful if your design always considers a dummy node inside the queue – this helps in the case of a logically empty queue. This optimization is actually needed for further improving the performance by using two locks instead of one (cf. Reporting, part 2).

In particular, when initializing the queue, a dummy node (with any `int` value) should be created and the `Head` and `Tail` will point there. Then a `dequeue` operation will consider the queue empty iff the first node's next pointer is `NULL` – thus `Head` or `Tail` should never be null and always point to the dummy node. If the queue is not empty the `dequeue` should store the value of the second node in the `extractedValue` and the first node will be actually disconnected from the queue i.e. the latest logically dequeued node will be always the dummy node.

Testing

Implement in a tester.c file a main() function that will:

1. Initialize the queue
2. Insert 100 nodes (of any value).
3. Spawn *N* threads that will randomly enqueue and dequeue *X* operations in total (*X/N* each).
4. Measure and return the time duration in milliseconds of step 3.
5. *N* should be an input argument, and *X* a constant with value 100000.

For step 3 the use of pthreads is again needed (see previous section). One way to measure the elapsed time is by using the `gettimeofday()` in the beginning and the end of the desired interval. Useful keywords: `pthread_create`, `ptread_join`, `<sys/time.h>`, `struct timeval`.

Reporting

Part 1

Implement the queue in a file concurrent_queue.c using **one** lock.

Test your implementation for *N*=2 and 4. Are there any differences? Why? Describe your results in a pdf file called report.pdf.

Is a deadlock possible to occur? Justify your answer (give an example if yes or a proof otherwise) in the report.

Part 2

Is the above the most efficient implementation? Can it get better?

Implement the queue using **two** locks where appropriate in a separate file concurrent_queue_2locks.c

Again, test your implementation for *N*=2 and 4. Are there any differences? Why? Was there any gain with the second lock? Describe your results in your report.

Is a deadlock possible to occur? Justify your answer (an example if yes or proof otherwise) in your report.

Submission

Submit a compressed archive (tar.gz,zip,rar) including any files needed for your implementation (at least concurrent_queue.c, concurrent_queue_2locks.c, tester.c) and your report.pdf as described above extended with any information that you think is relevant.