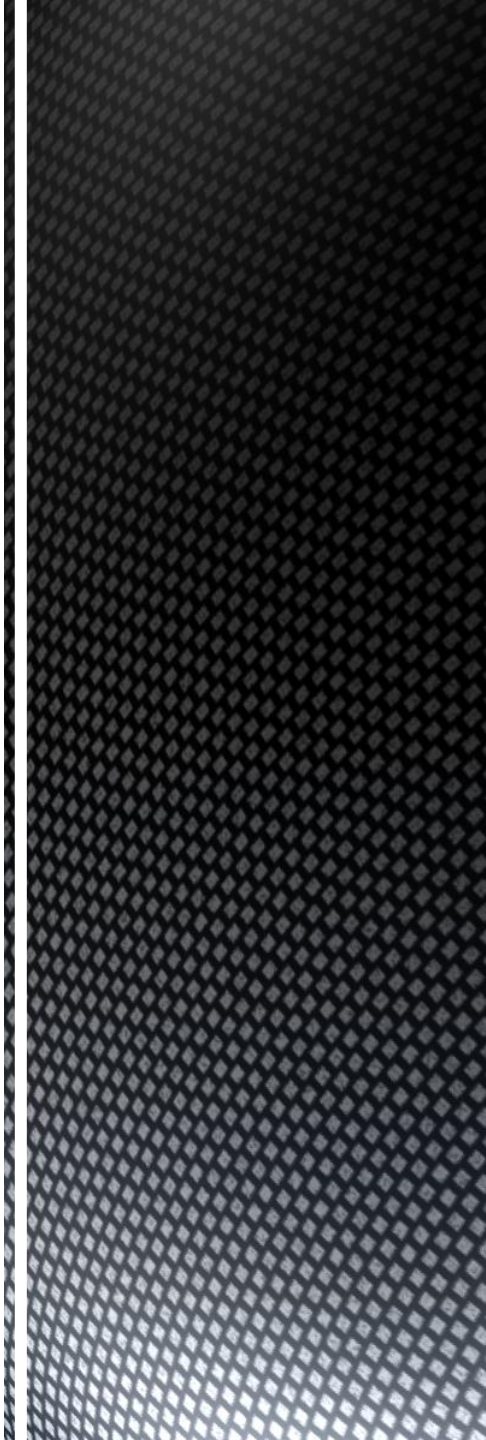
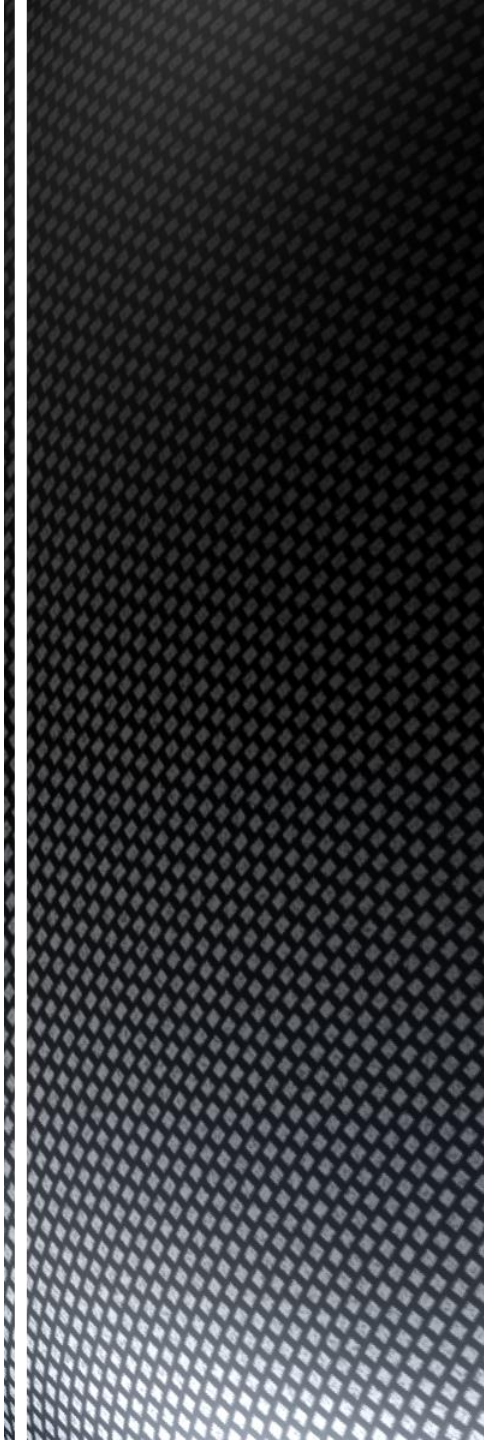
Operating Systems Lab 2: Thread Synchronization and a bit of Pthreads

Yiannis Nikolakopoulos
ioaniko@chalmers.se



Pthreads



Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Slide by Silberschatz, Galvin and Gagne

Pthreads basic API

Creating a thread:

- int pthread_create(pthread_t * thread,
  const pthread_attr_t * attr,
  void *(*start_routine)(void *),
  void *arg);
  - thread: the thread id is returned here
  - attr: thread attributes to be used in the new thread
  - void *(*start_routine): the function that will be run by the thread; it should get only one argument: a void pointer
  - void *arg: pointer to the argument for the start_routine; use a structure for multiple arguments
  - *Returns:* 0 on success, an error number otherwise

Pthreads basic API

Wait for a thread to terminate:

- int pthread_join(pthread_t th,
  void **thread_return);
  - th: the thread that we wait to terminate
  - thread_return: the return location of th

Hello World

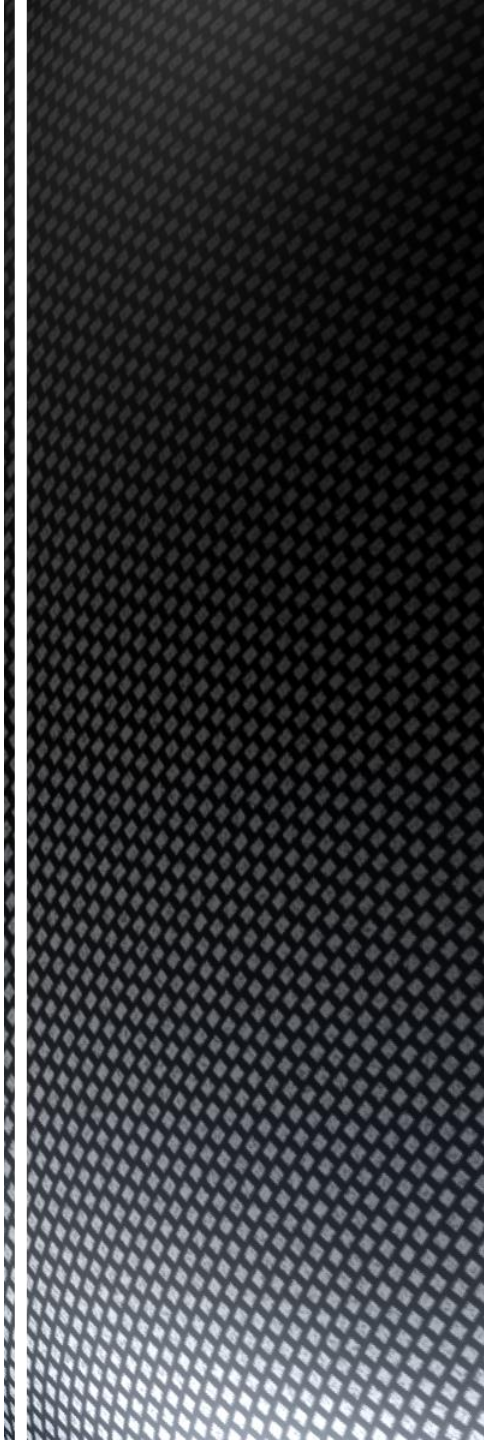
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *dummy_function(void *arg);

main()
{
  pthread_t thread1, thread2;
  const char *msg1 = "Hello ";
  const char *msg2 = "World! ";

  pthread_create( &thread1, NULL, dummy_function, (void *) msg1);
  pthread_create( &thread2, NULL, dummy_function, (void *) msg2);

  /*Wait for threads to finish */
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  exit(0);
}

void *dummy_function(void *arg)
{
  char *message;
  message = (char *) arg;
  printf("%s \n", message);
}
```

NOTES

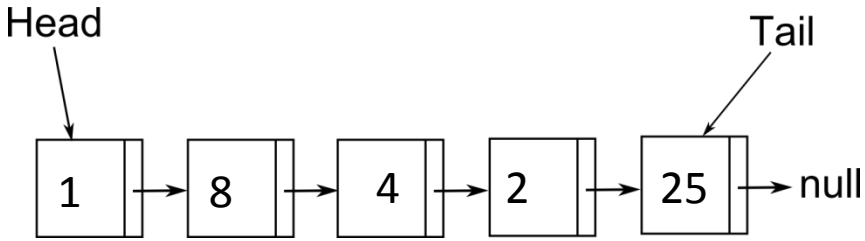
1. #include <pthread.h>
2. Compile with '-pthread' flag

Lab 2
Concurrent Queue

Or the Multiple Producer Multiple Consumer Problem



Queue: Structure and Interface



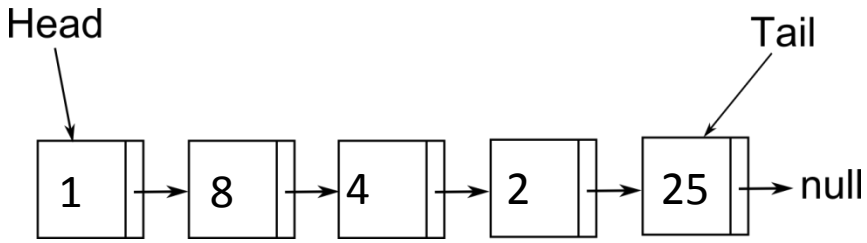
The queue:

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

The goal:

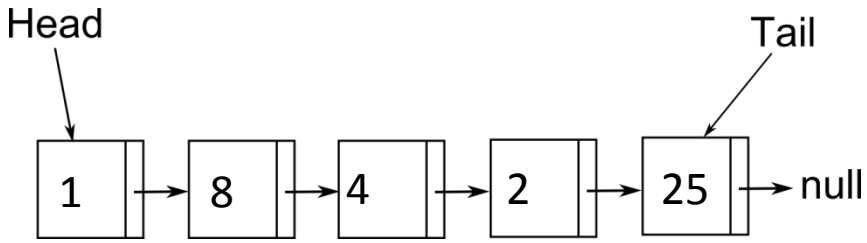
- Mulitple threads to enqueue and dequeue concurrently

Queue: Structure and Interface



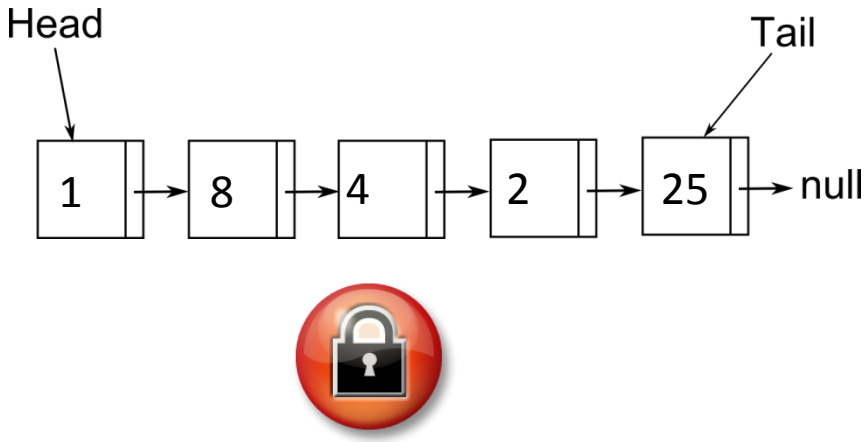
Prototypes:

- void initialize_queue(void);
- void enqueue(int val);
- int dequeue(int *extractedValue);



- Nothing strange so far…
- How to make it concurrent?
- Which are the dangers?
- What can you use?

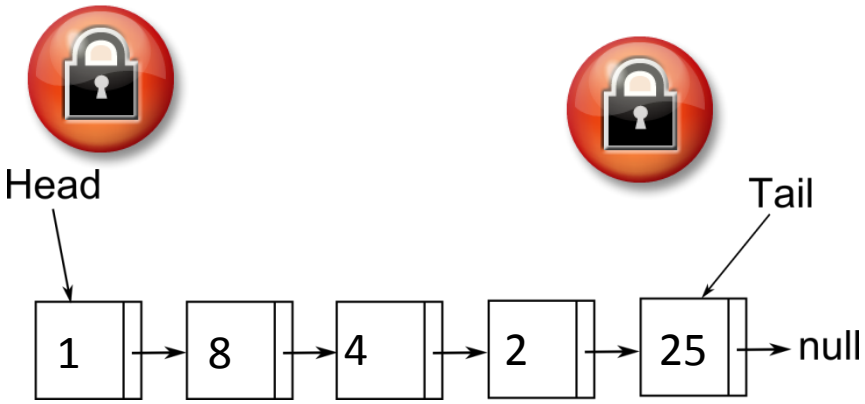
Concurrency



Concurrency

Simple case:

- One lock: acquire before any Head/Tail modification or connection to the list



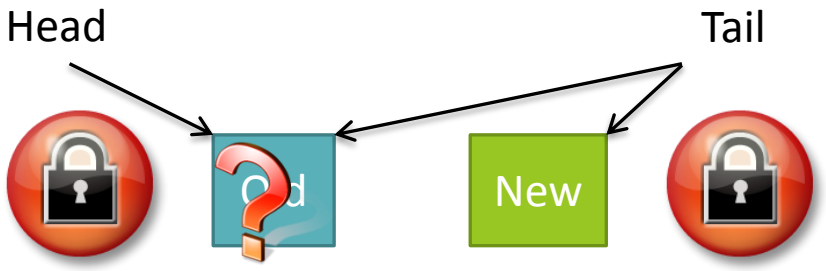
A slightly more complex case:

- Two locks

Generally:

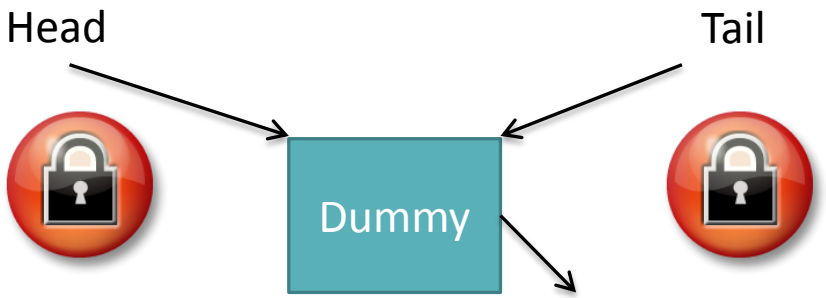
- The locks should be invisible to the "users" of the queue,
  i.e. used inside the enqueue or dequeue methods

Concurrency



- What can happen if the queue is empty or with only one node left?

Concurrency: Dummy Node



Concurrency: Dummy Node

- We will keep always one dummy node in the list (starting from the initialization)
- The queue is (logically) empty when Dummy's next is NULL
- On a dequeue:
  - We return the value of the node next to the Dummy
  - We consider that node the new Dummy (and remove the old one)

Testing and Reporting

Detailed instructions in lab PM, but roughly:

- N threads will randomly enqueue/dequeue each time a fixed number of X operations in total (X/N each)
- Measure how much time this takes for the 2 different queue implementations
- Prove that no deadlock will occur in each of your implementations