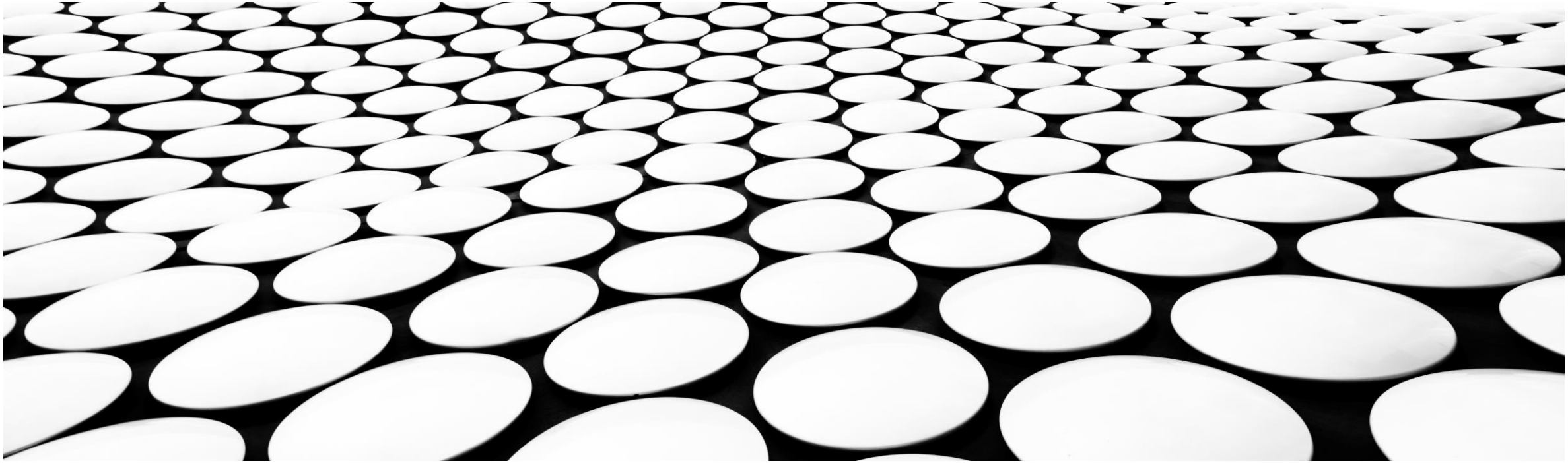
POSIX threads

COEN-317: Distributed Systems
Robert Bruce
Department of Computer Science and Engineering
Santa Clara University



What are POSIX threads?

What is POSIX?

- POSIX stands for "Portable Operating System Interface" [1].
- POSIX defines "defines a standard operating system interface and environment" [1].
- POSIX leads to interoperability and portability in software application development.

What are POSIX threads?

- An Application Programmer Interface (API) of functions for creating threaded applications as defined by *IEEE Standard 1003.1c-1995* [2].
- "pthreads" is a synonym for "POSIX threads".

[1] https://standards.ieee.org/ieee/1003.1/7700/
[2] https://standards.ieee.org/ieee/1003.1c/1393/

Processes and threads

An operating system separates runs applications (processes) in separate, virtualized address spaces.

- This virtualization technique maps the virtual memory into real physical memory [1].
- Each process is executed independently by the operating system [1].
- Memory for each process is protected from other processes within its own virtual address space [1].
- Each process has its own heap and stack space that is separate from other processes [1].

Threads are like processes except in how they are managed by the operating system:

- For an application that utilizes two or more threads, the threads share the same memory in user space [2].
- Caveat: an application developer must be careful with concurrent programming to avoid race-conditions.
- A mutex (binary semaphore) is a signaling mechanism to impose atomic cooperation and synchronization between two or more threads.

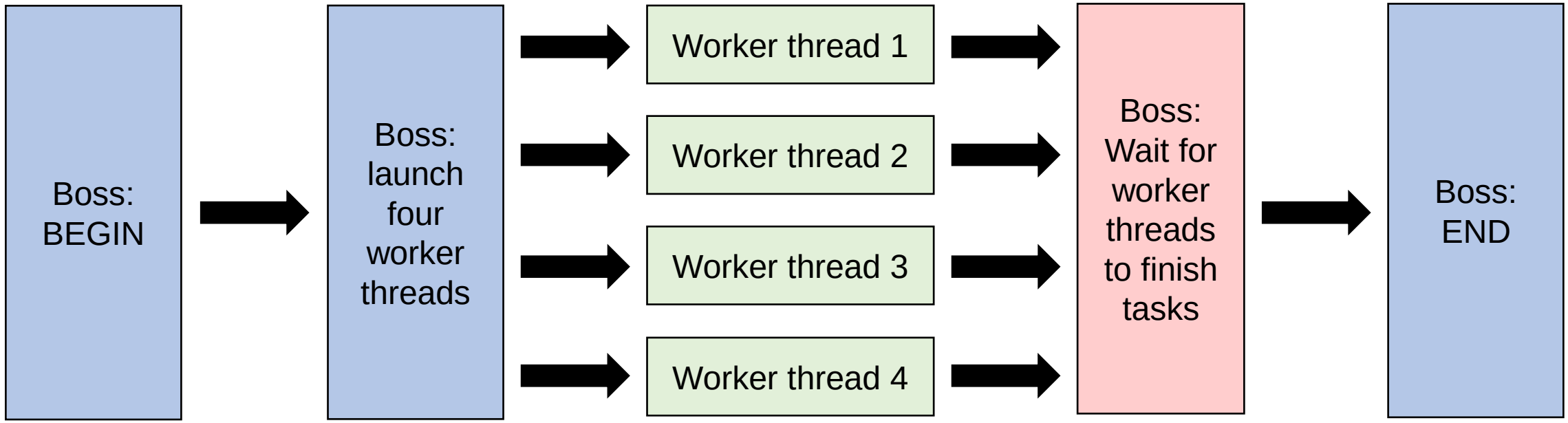
[1] pp. 104-105, *Distributed Systems* (3rd edition) by Maarten van Steen and Andrew S. Tanenbaum.
[2] p. 108, *Distributed Systems* (3rd edition) by Maarten van Steen and Andrew S. Tanenbaum.

POSIX threads example program

The example program, *pthread_example.c* uses a boss-worker thread model:

- The boss thread (also called the *dispatcher*) launches four worker threads then waits for each worker thread to complete its task.
- Each worker thread saves the current date/time then exits.
- After each worker thread is finished, the boss (or dispatcher) thread exits.

POSIX threads: Boss-Worker model



Dissecting the POSIX threads example program (main)

```
int main (int argc, char *argv[])
{
  worker_thread_node worker_thread[4];
  int return_code;
  int i;

  for (i = 0; i < 4; i++)
  {
    bzero (&worker_thread[i].date_time_string[0], sizeof(worker_thread[i].date_time_string));
  }
  return_code = pthread_create (&worker_thread[0].thread_id, NULL, thread_process_0, (void *) &worker_thread[0]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[1].thread_id, NULL, thread_process_1, (void *) &worker_thread[1]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[2].thread_id, NULL, thread_process_2, (void *) &worker_thread[2]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[3].thread_id, NULL, thread_process_3, (void *) &worker_thread[3]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
```

Dissecting the POSIX threads example program (main)

```
int main (int argc, char *argv[])
{
  worker_thread_node worker_thread[4];
  int return_code;
  int i;

  for (i = 0; i < 4; i++)
  {
    bzero (&worker_thread[i].date_time_string[0], sizeof(worker_thread[i].date_time_string));
  }
  return_code = pthread_create (&worker_thread[0].thread_id, NULL, thread_process_0, (void *) &worker_thread[0]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_co
  if (retur
  {
    fprintf
    exit (E
  }
  return_code = pthread_create (&worker_thread[2].thread_id, NULL, thread_process_2, (void *) &worker_thread[2]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[3].thread_id, NULL, thread_process_3, (void *) &worker_thread[3]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
```

The *pthread_create()* function creates a thread and defines the subroutine the thread should run in.

This newly created thread will execute inside the subroutine *thread_process_0*.

Dissecting the POSIX threads example program (main)

```
int main (int argc, char *argv[])
{
  worker_thread_node worker_thread[4];
  int return_code;
  int i;

  for (i = 0; i < 4; i++)
  {
    bzero (&worker_thread[i].date_time_string[0], sizeof(worker_thread[i].date_time_string));
  }
  return_code = pthread_create (&worker_thread[0].thread_id, NULL, thread_process_0, (void *) &worker_thread[0]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[1].thread_id, NULL, thread_process_1, (void *) &worker_thread[1]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  retur
  if (r
  {
    fpr
    exi
  }
  return_code = pthread_create (&worker_thread[3].thread_id, NULL, thread_process_3, (void *) &worker_thread[3]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
```

The *pthread_create()* function creates a second thread and defines the subroutine the second thread should run in.

This second thread will execute inside the subroutine *thread_process_1*.

Dissecting the POSIX threads example program (main)

```
int main (int argc, char *argv[])
{
  worker_thread_node worker_thread[4];
  int return_code;
  int i;

  for (i = 0; i < 4; i++)
  {
    bzero (&worker_thread[i].date_time_string[0], sizeof(worker_thread[i].date_time_string));
  }
  return_code = pthread_create (&worker_thread[0].thread_id, NULL, thread_process_0, (void *) &worker_thread[0]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[1].thread_id, NULL, thread_process_1, (void *) &worker_thread[1]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[2].thread_id, NULL, thread_process_2, (void *) &worker_thread[2]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_c
  if (retu
  {
    fprint
    exit (
  }
```

The *pthread_create()* function creates a third thread and defines the subroutine the third thread should run in.

This third thread will execute inside the subroutine *thread_process_2*.

Dissecting the POSIX threads example program (main)

```
int main (int argc, char *argv[])
{
  worker_thread_node worker_thread[4];
  int return_code;
  int i;

  for (i = 0; i < 4; i++)
  {
    bzero (&worker_thread[i].date_time_string[0], sizeof(worker_thread[i].date_time_string));
  }
  return_code = pthread_create (&worker_thread[0].thread_id, NULL, thread_process_0, (void *) &worker_thread[0]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_create (&worker_thread[1].thread_id, NULL, thread_process_1, (void *) &worker_thread[1]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
  return_
  if (ret
  {
    fprin
    exit
  }
  return_code = pthread_create (&worker_thread[3].thread_id, NULL, thread_process_3, (void *) &worker_thread[3]);
  if (return_code)
  {
    fprintf (stderr, "Error creating thread\n");
    exit (EXIT_FAILURE);
  }
```

The *pthread_create()* function creates a fourth thread and defines the subroutine the fourth thread should run in.

This fourth thread will execute inside the subroutine *thread_process_3*.

Dissecting the POSIX threads example program (main)

```
  return_code = pthread_join (worker_thread[0].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[1].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[2].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[3].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  for (i = 0; i < 4; i++)
  {
    printf ("Thread %d: %s\n", i, worker_thread[i].date_time_string);
  }
  exit (EXIT_SUCCESS);
}
```

Dissecting the POSIX threads example program (main)

```
return_code = pthread_join (worker_thread[0].thread_id, NULL);
if (return_code)
{
  fprintf (stderr, "Error joining thread\n");
  exit (EXIT_FAILURE);
}
return_
if (ret
{
  fprin
  exit
}
return_
if (ret
{
  fprin
  exit
}
return_
if (ret
{
  fprintf (stderr, "Error joining thread\n");
  exit (EXIT_FAILURE);
}
for (i = 0; i < 4; i++)
{
  printf ("Thread %d: %s\n", i, worker_thread[i].date_time_string);
}
exit (EXIT_SUCCESS);
}
```

Four worker threads are running concurrently in their respective subroutines:

- *thread_process_0* (first worker thread)
- *thread_process_1* (second worker thread)
- *thread_process_2* (third worker thread)
- *thread_process_3* (fourth worker thread)

The boss thread in main waits for the first worker thread to finish its task. This is done with the *pthread_join()* function.

Dissecting the POSIX threads example program (main)

```
  return_code = pthread_join (worker_thread[0].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[1].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_jo
  if (return_code)
  {
    fprintf (stderr, "Erro
    exit (EXIT_FAILURE);
  }
  return_code = pthread_jo
  if (return_code)
  {
    fprintf (stderr, "Erro
    exit (EXIT_FAILURE);
  }
  for (i = 0; i < 4; i++)
  {
    printf ("Thread %d: %s\n", i, worker_thread[i].date_time_string);
  }
  exit (EXIT_SUCCESS);
}
```

Three worker threads are running concurrently in their respective subroutines:

- *thread_process_1* (second worker thread)
- *thread_process_2* (third worker thread)
- *thread_process_3* (fourth worker thread)

The boss thread in main waits for the second worker thread to finish its task.

Dissecting the POSIX threads example program (main)

```
  return_code = pthread_join (worker_thread[0].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[1].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[2].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_jo
  if (return_code)
  {
    fprintf (stderr, "Erro
    exit (EXIT_FAILURE);
  }
  for (i = 0; i < 4; i++)
  {
    printf ("Thread %d: %s
  }
  exit (EXIT_SUCCESS);
}
```

Two worker threads are running concurrently in their respective subroutines:

- *thread_process_2* (third worker thread)
- *thread_process_3* (fourth worker thread)

The boss thread in main waits for the third worker thread to finish its task.

Dissecting the POSIX threads example program (main)

```
  return_code = pthread_join (worker_thread[0].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[1].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[2].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[3].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  for (i = 0; i < 4; i++)
  {
    printf ("Thread %d: %s
  }
  exit (EXIT_SUCCESS);
}
```

One worker thread is running in the subroutine:

- *thread_process_3* (fourth worker thread)

The boss thread in main waits for the fourth worker thread to finish its task.

Dissecting the POSIX threads example program (main)

```
  return_code = pthread_join (worker_thread[0].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[1].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_join (worker_thread[2].thread_id, NULL);
  if (return_code)
  {
    fprintf (stderr, "Error joining thread\n");
    exit (EXIT_FAILURE);
  }
  return_code = pthread_jo
  if (return_code)
  {
    fprintf (stderr, "Erro
    exit (EXIT_FAILURE);
  }
  for (i = 0; i < 4; i++)
  {
    printf ("Thread %d: %s\n", i, worker_thread[i].date_time_string);
  }
  exit (EXIT_SUCCESS);
}
```

The boss thread (main) is now in single-process mode. The boss thread outputs the results from the worker threads (i.e. output the date and time) that the worker threads wrote as strings in their respective subroutines.

Dissecting the POSIX threads example program

In this example, the following thread subroutines are all identical:

- *thread_process_0*
- *thread_process_1*
- *thread_process_2*
- *thread_process_3*

Therefore, we'll examine just the first one (*thread_process_0*).

Dissecting the POSIX threads example program (thread_process_0)

```
void * thread_process_0 (void *args)
{
  time_t t;
  struct tm *tmp;
  worker_thread_node *worker_thread;

  worker_thread = (worker_thread_node *) args;
  t = time (NULL);
  tmp = localtime (&t);
  if (tmp == NULL)
  {
    fprintf (stderr, "localtime failed");
    exit (EXIT_FAILURE);
  }
  if (strftime (&worker_thread->date_time_string[0], sizeof(worker_thread->date_time_string), "%B %d, %Y %I:%M:%S %P", tmp) == 0)
  {
    fprintf (stderr, "strftime returned 0");
    exit (EXIT_FAILURE);
  }
  return (EXIT_SUCCESS);
}
```

Dissecting the POSIX threads example program (thread_process_0)

```
void * thread_process_0 (void *args)
{
  time_t t;
  struct tm *tmp;
  worker_thread_node *worker_thread;

  worker_thread = (worker_thread_node *) args;
  t = time (NULL);
  tmp = localti
  if (tmp == NU
  {
    fprintf (st
    exit (EXIT_
  }
  if (strftime (&worker_thread->date_time_string[0], sizeof(worker_thread->date_time_string), "%B %d, %Y %I:%M:%S %P", tmp) == 0)
  {
    fprintf (stderr, "strftime returned 0");
    exit (EXIT_FAILURE);
  }
  return (EXIT_SUCCESS);
}
```

Parameters sent to each thread subroutine are passed in with the void pointer to args (arguments).

Strategy for using POSIX threads

The use of POSIX threads is vital to the COEN-317 programming project.

Master-queue:

The master queue is responsible for assigning, load-balancing, and monitoring the completion of a series of parallel tasks assigned to two or more worker machines.

Worker machine:

The worker machine is responsible for processing a job request as sent by the master queue. Specifically, the worker machine must:

1. Communicate the time to complete the job to the master queue
2. Send back results from task completion (such as images of a face mesh composited on a human face).

Strategy for using POSIX threads: the master queue pipeline

Master queue:

- Thread one listens to socket connection for incoming video files to process. When a job request is received, the job is placed in a local worker queue and marked as "available".
- Thread two monitors the worker machines and the master queue of jobs waiting to be processed. If there is a worker machine available, thread two assigns the job to that worker machine's queue:
    1. Thread two in the master queue assigns the job to a specific worker machine (with a specified IP address) and updates incoming job as "in-progress".
- Thread two continues to monitor all in-progress jobs assigned to worker machines. When a worker machine messages back to the master queue that it has completed the job (i.e. converting a video into a series of still images):
    1. Thread two in the master queue updates the job task with "completed". The information from the worker machine indicates the number of frames processed for a given video identifier (*video_id*).
- Thread three in the master queue looks for video files that have just been converted to still images with status "completed". The third thread is responsible creating a subsequent series of jobs to process each still image from the video identified by *video_id*. The following newly created jobs are added to the master queue. These jobs are assigned to worker machines by thread two (above):
    1. Retrieve the 68 facial data points in the still image with filename: *video_id*.*frame*.png, calculate then draw Delaunay triangles on the still image based on the sixty-eight data points. Lastly, create an output file with the composited mesh on that still image titled: mesh-*video_id*.*frame*.png
- Thread four monitors the master queue to determine if the meshes have been drawn for every still image of a given *video_id*. If this is true, thread four submits the following job to the queue for subsequent processing:
    1. Use FFMPEG to create a video from the still images containing face meshes back into a video based on video_id.

Strategy for using POSIX threads: Worker machine

Worker machine:

- Thread one listens to socket connection for incoming job requests from master queue. When a job request is received, the job is placed in a local worker queue and marked as "available".
- Thread two monitors the local worker queue looking for jobs to complete. If a job exists in the local worker queue and thread two is available (i.e. not processing a job):
    1. Thread two requests a mutex (lock).
    2. If thread two receives the mutex lock, thread two updates the incoming job as "in-progress" by thread two.
    3. Thread two releases the mutex (lock).
    4. When job is completed, thread two updates the incoming job as "completed" by thread two.
    5. Thread two sends output from completed work along with time elapsed (to complete job) back to master queue.

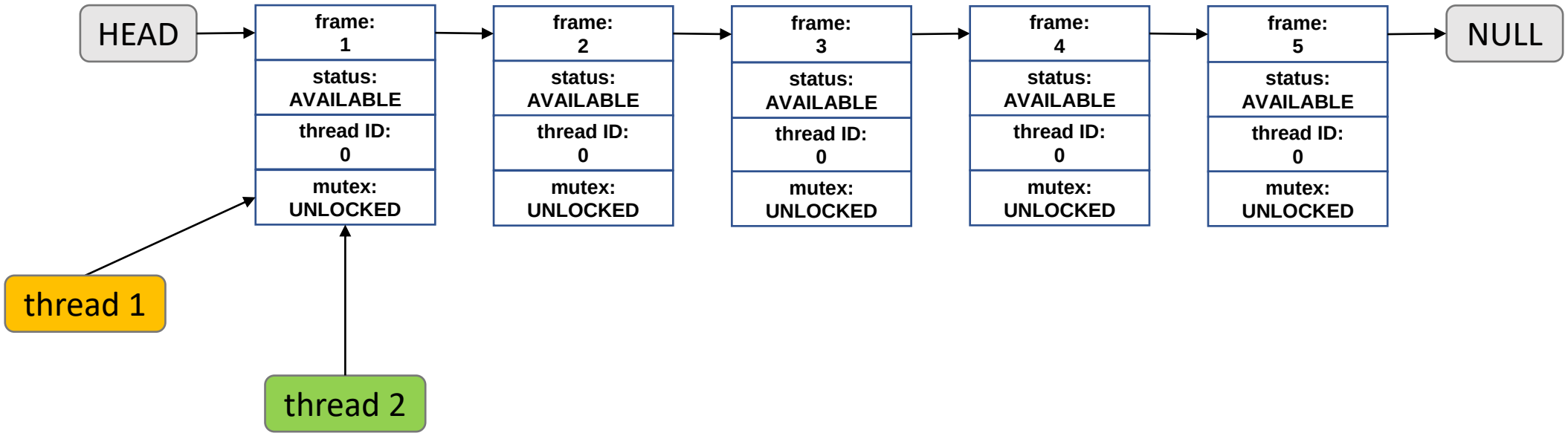
Achieving mutual exclusion in multi-threaded applications: the mutex

Developing a pipeline and coordinating multiple threads to work collaboratively together requires atomic level synchronization.

- The mutex (**MUT**ual **EX**clusion) is a binary semaphore.
- It can be locked or unlocked by only one thread at a time.
- It is an atomic operation that is completed in the kernel of the operating system (i.e. outside of user space).

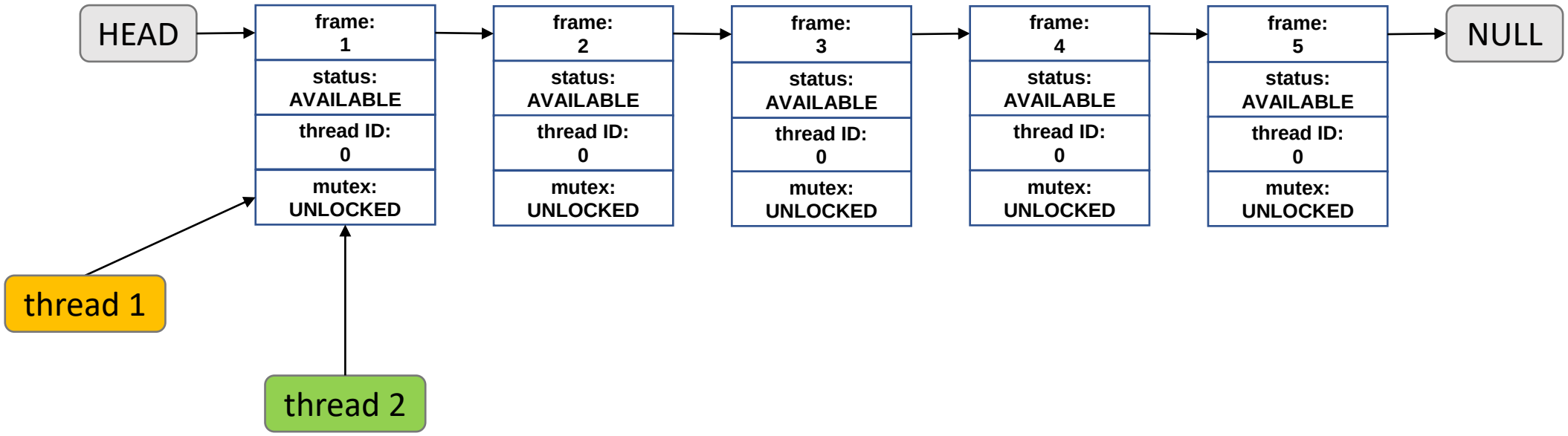
Using the mutex in POSIX threads: a practical example

A simple job queue linked list that is being updated by two or more threads:



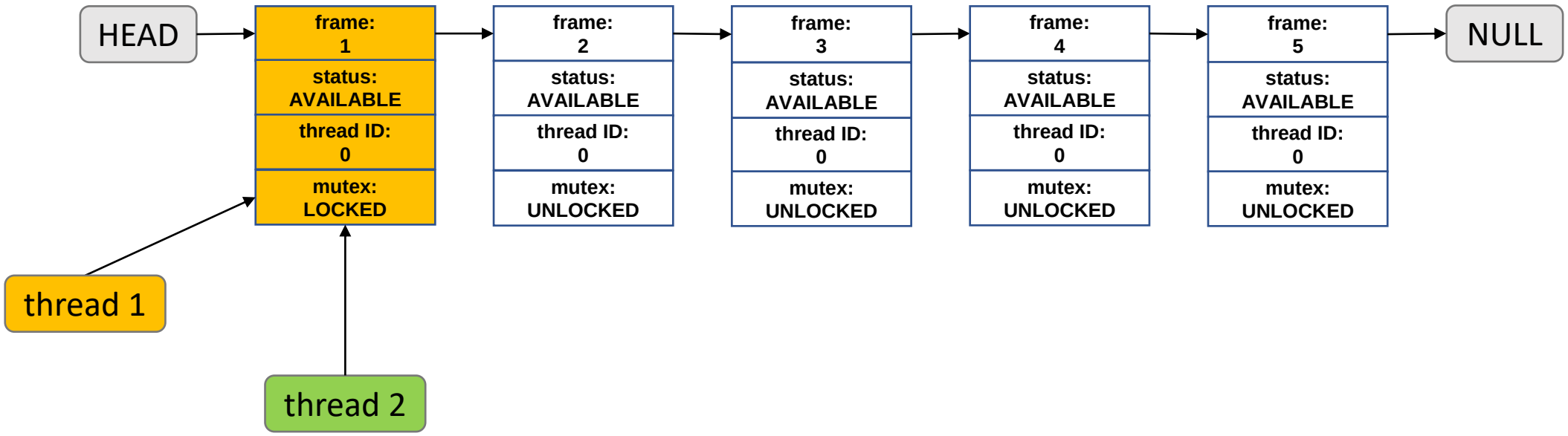
Using the mutex in POSIX threads: a practical example

If the status of an element in the job queue is "AVAILABLE", thread one and thread two request a mutex lock.



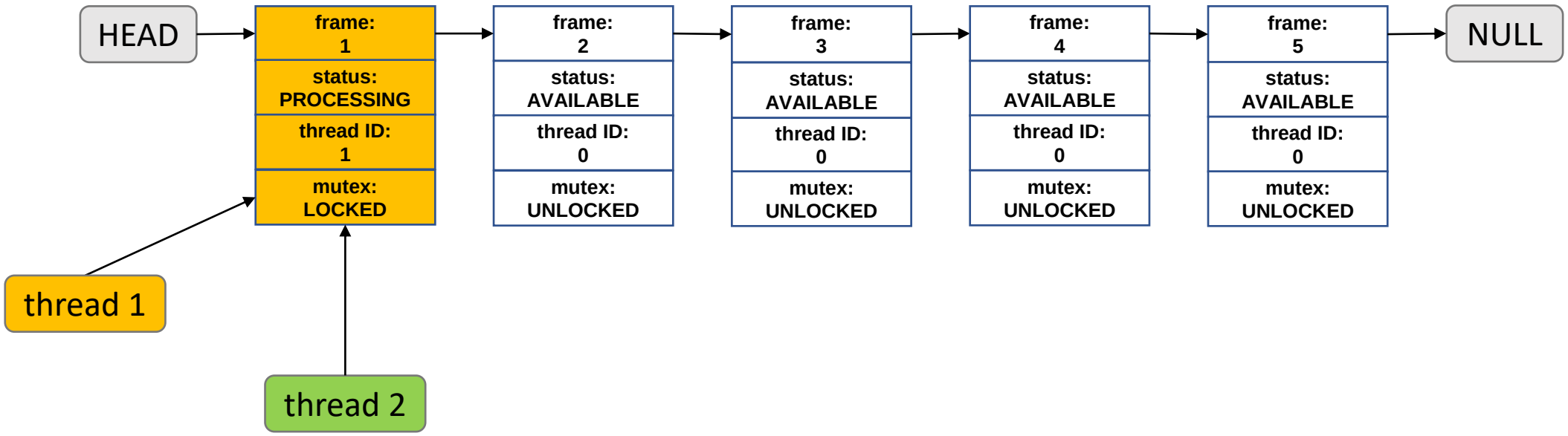
Using the mutex in POSIX threads: a practical example

The lock is granted to thread one. Check the status once again to ensure the element hadn't changed from "AVAILABLE" to "IN-PROCESS" by another thread before the lock was granted.



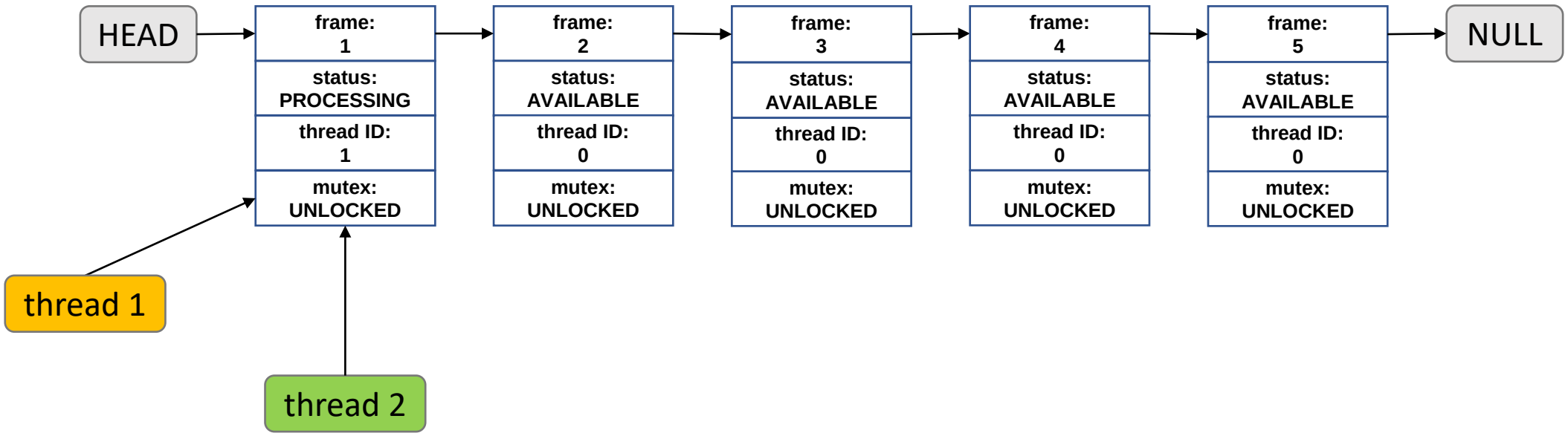
Using the mutex in POSIX threads: a practical example

Update the element in the job queue with status "PROCESSING" and the thread identifier that is processing this thread.



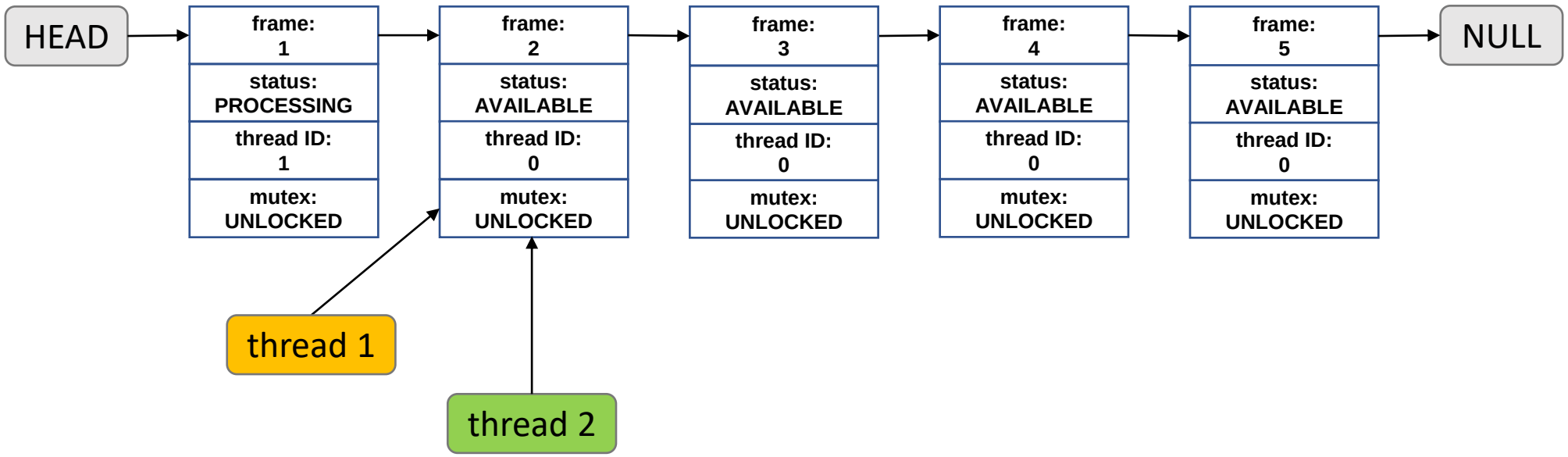
Using the mutex in POSIX threads: a practical example

Release the mutex lock on the individual element of the linked list:



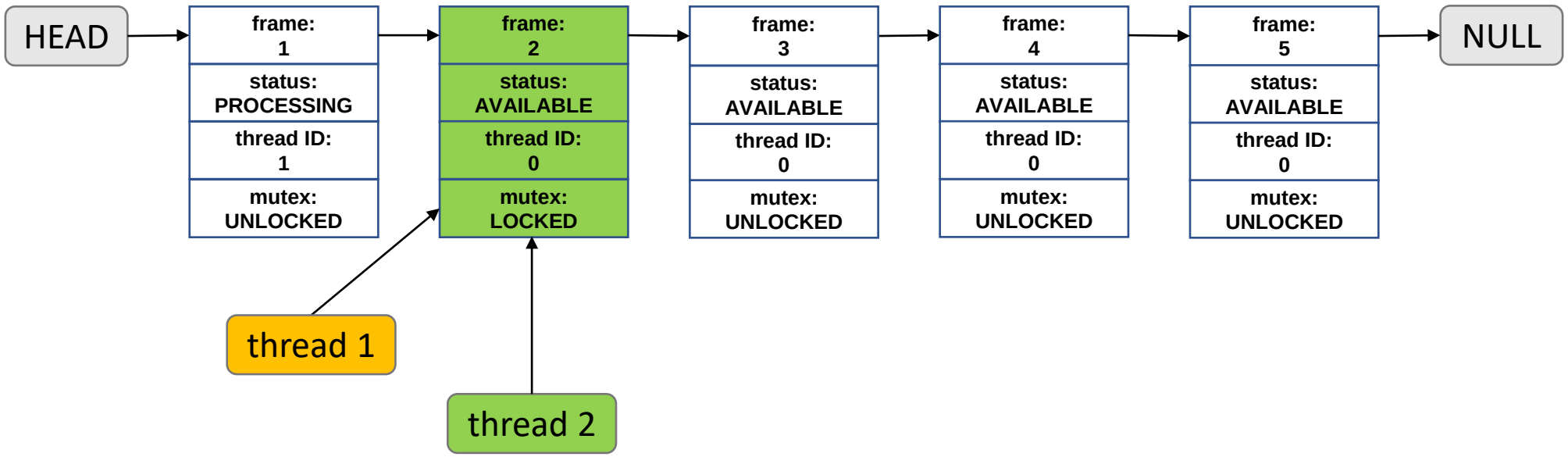
Using the mutex in POSIX threads: a practical example

Thread one and thread two advance to element two of the list. Both threads request the lock.



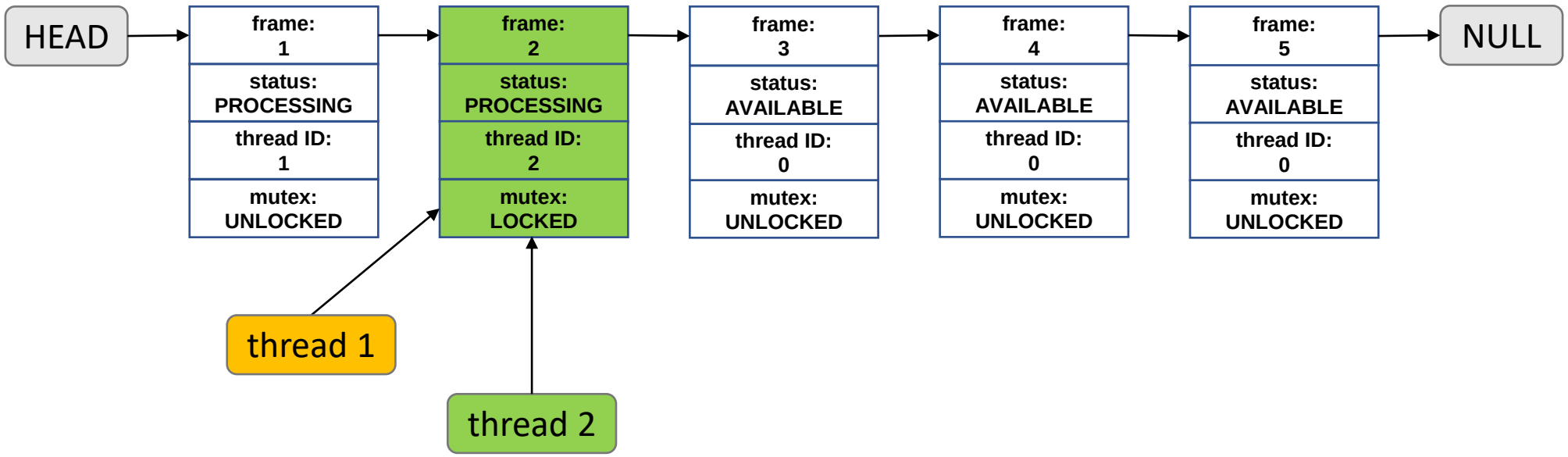
Using the mutex in POSIX threads: a practical example

The lock is granted to thread two. Check the status once again to ensure the element hadn't changed from "AVAILABLE" to "IN-PROCESS" by another thread before the lock was granted.



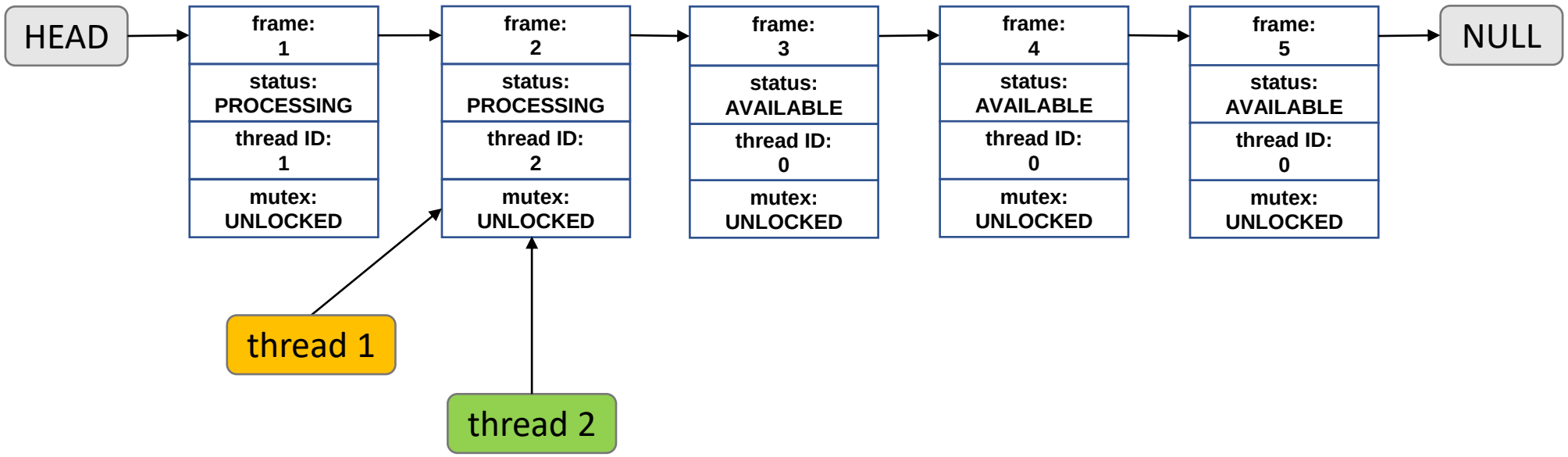
Using the mutex in POSIX threads: a practical example

Update the element in the job queue with status "PROCESSING" and the thread identifier that is processing this thread.



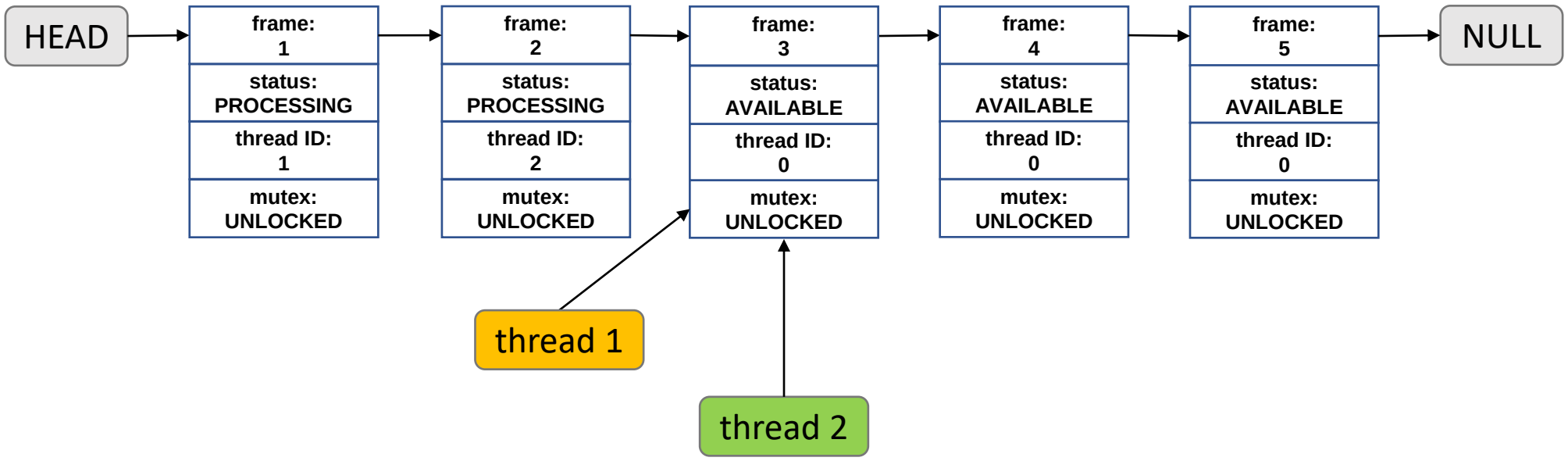
Using the mutex in POSIX threads: a practical example

Release the mutex lock on the individual element of the linked list:



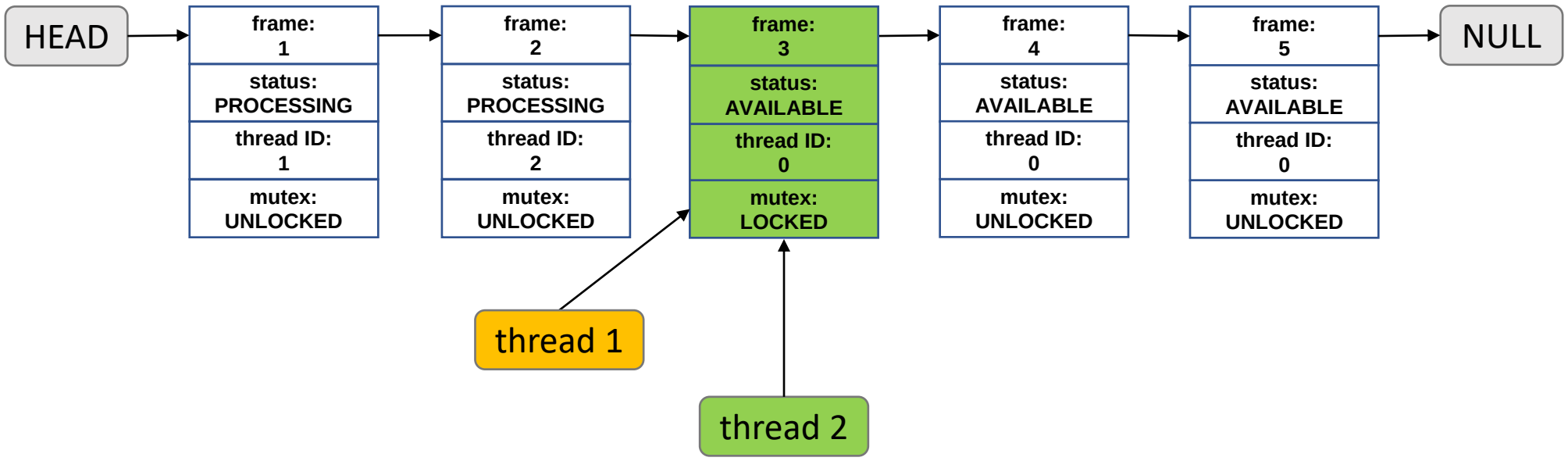
Using the mutex in POSIX threads: a practical example

Thread one and thread two advance to element three of the list. Both threads request the lock.



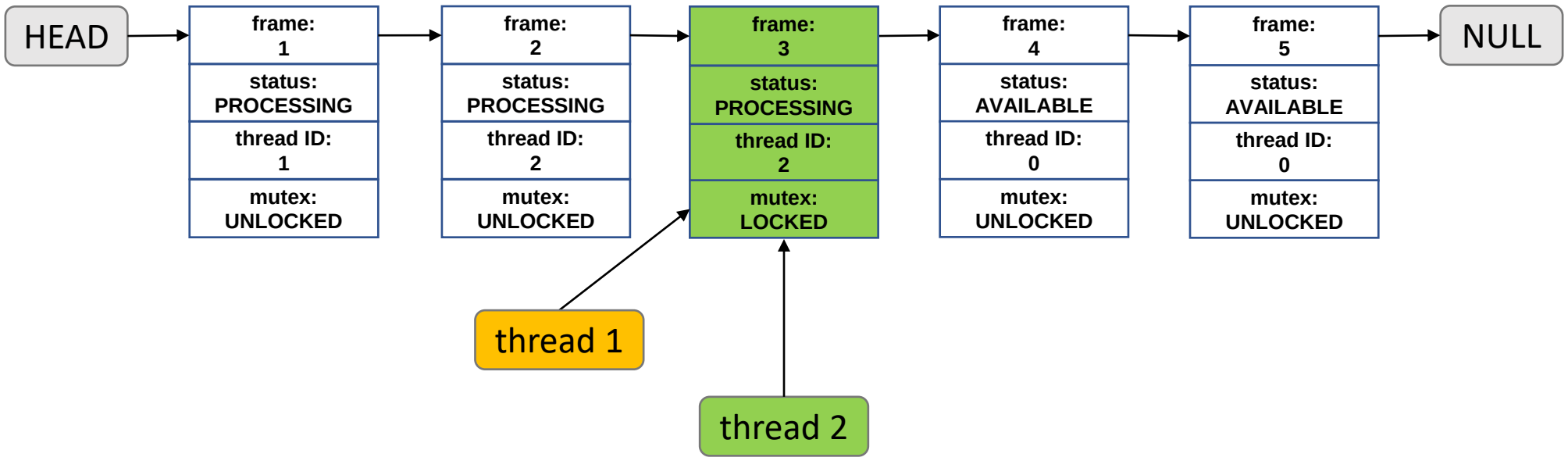
Using the mutex in POSIX threads: a practical example

The lock is granted to thread two. Check the status once again to ensure the element hadn't changed from "AVAILABLE" to "IN-PROCESS" by another thread before the lock was granted.



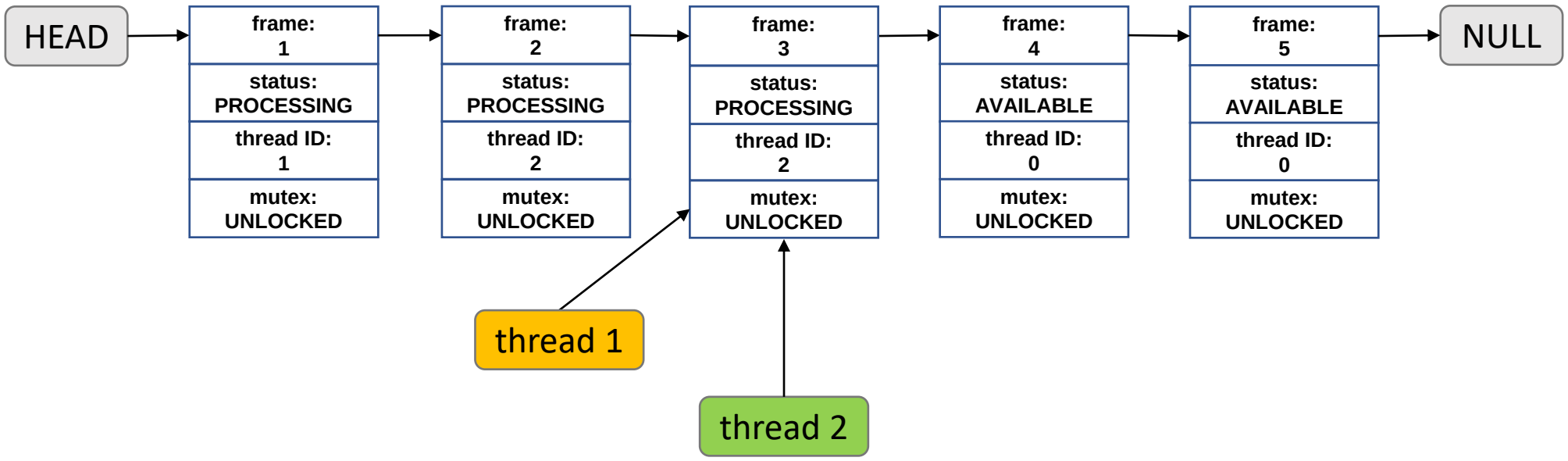
Using the mutex in POSIX threads: a practical example

Update the element in the job queue with status "PROCESSING" and the thread identifier that is processing this thread.



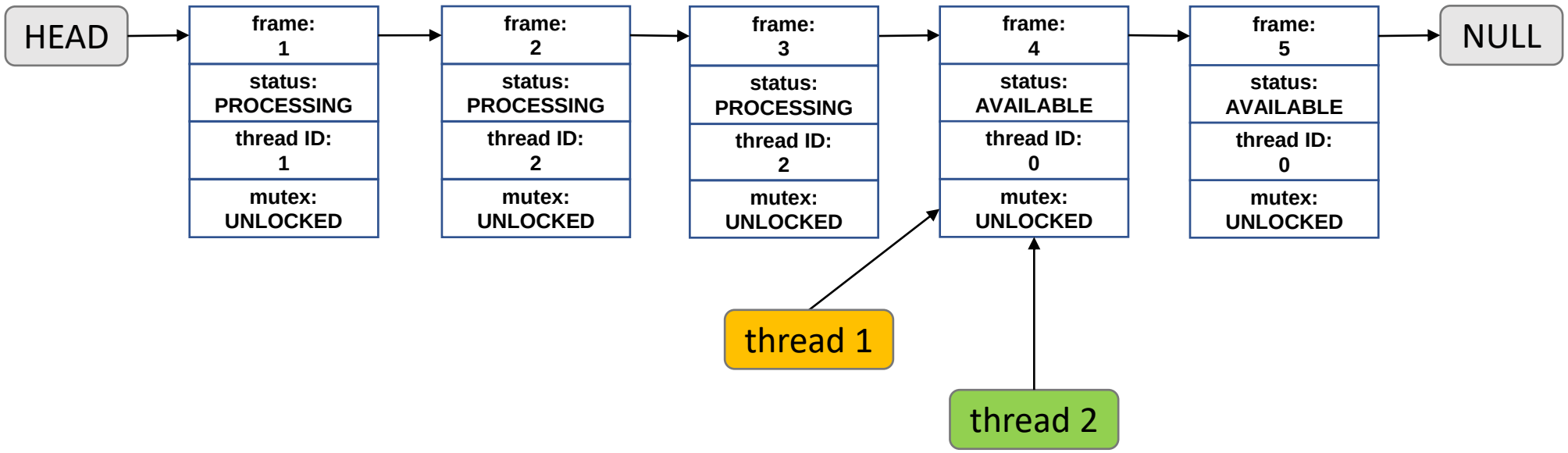
Using the mutex in POSIX threads: a practical example

Release the mutex lock on the individual element of the linked list:



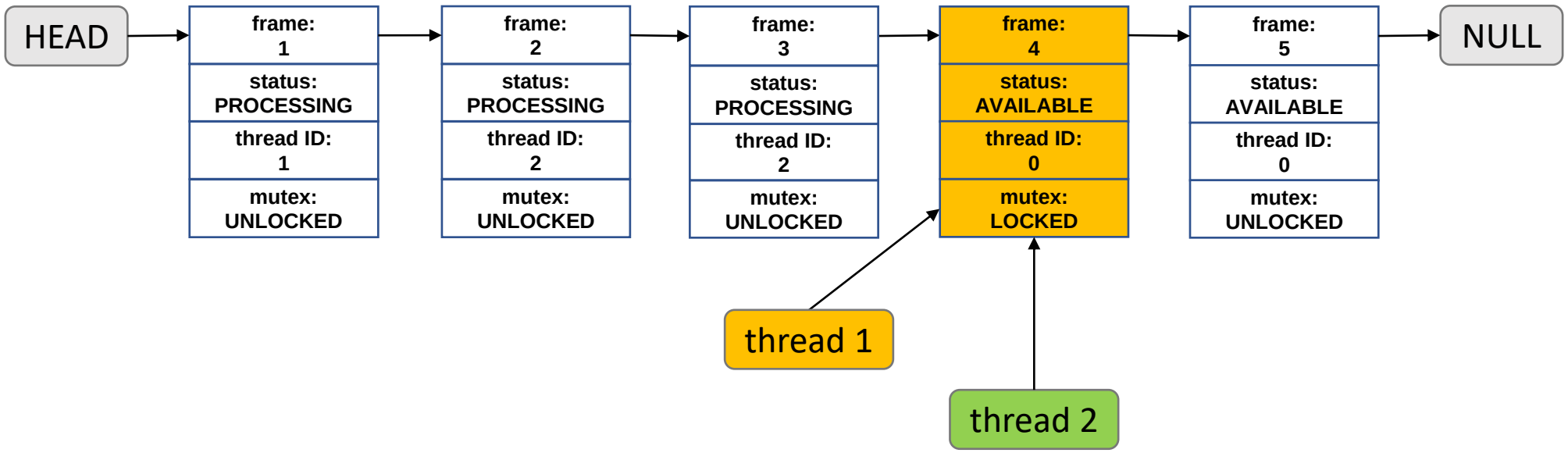
Using the mutex in POSIX threads: a practical example

Thread one and thread two advance to element four of the list. Both threads request the lock.



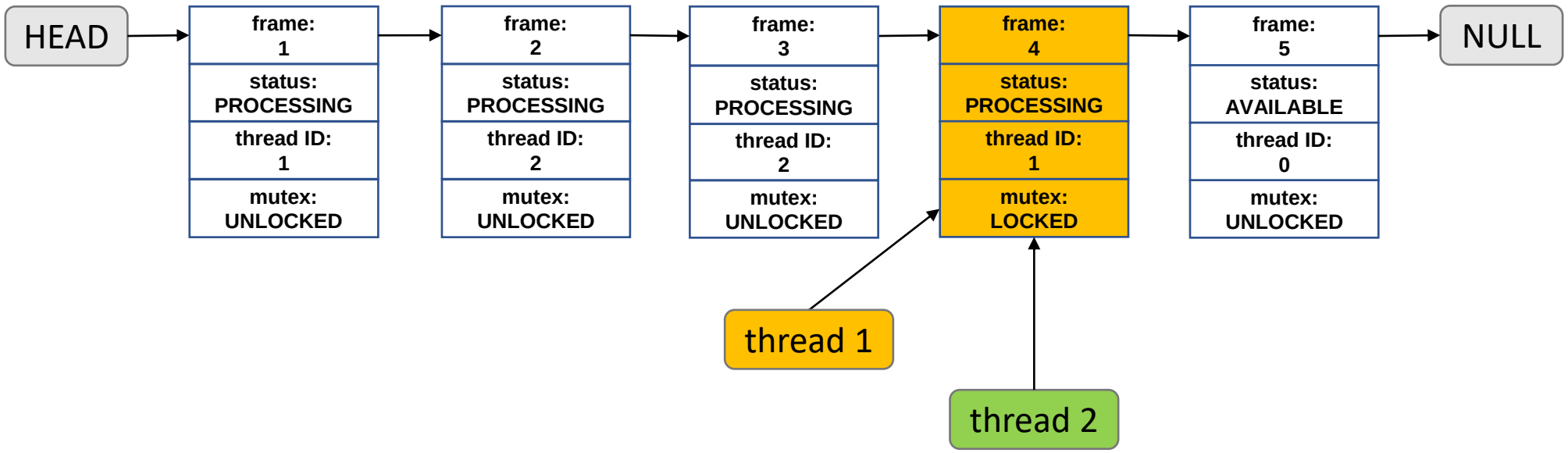
Using the mutex in POSIX threads: a practical example

The lock is granted to thread one. Check the status once again to ensure the element hadn't changed from "AVAILABLE" to "IN-PROCESS" by another thread before the lock was granted.



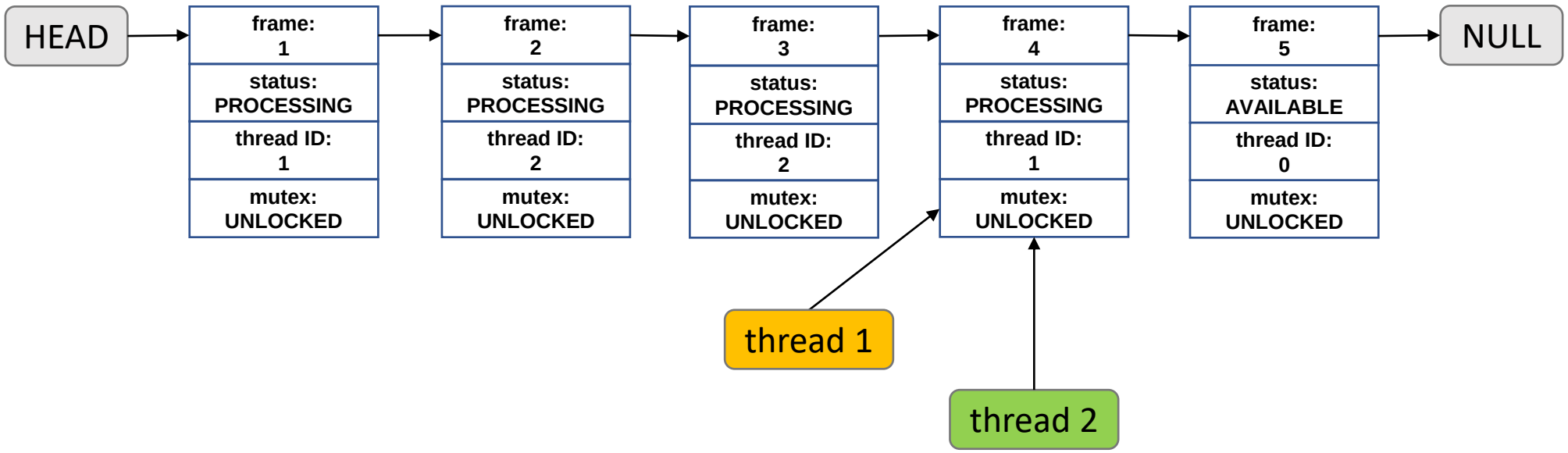
Using the mutex in POSIX threads: a practical example

Update the element in the job queue with status "PROCESSING" and the thread identifier that is processing this thread.



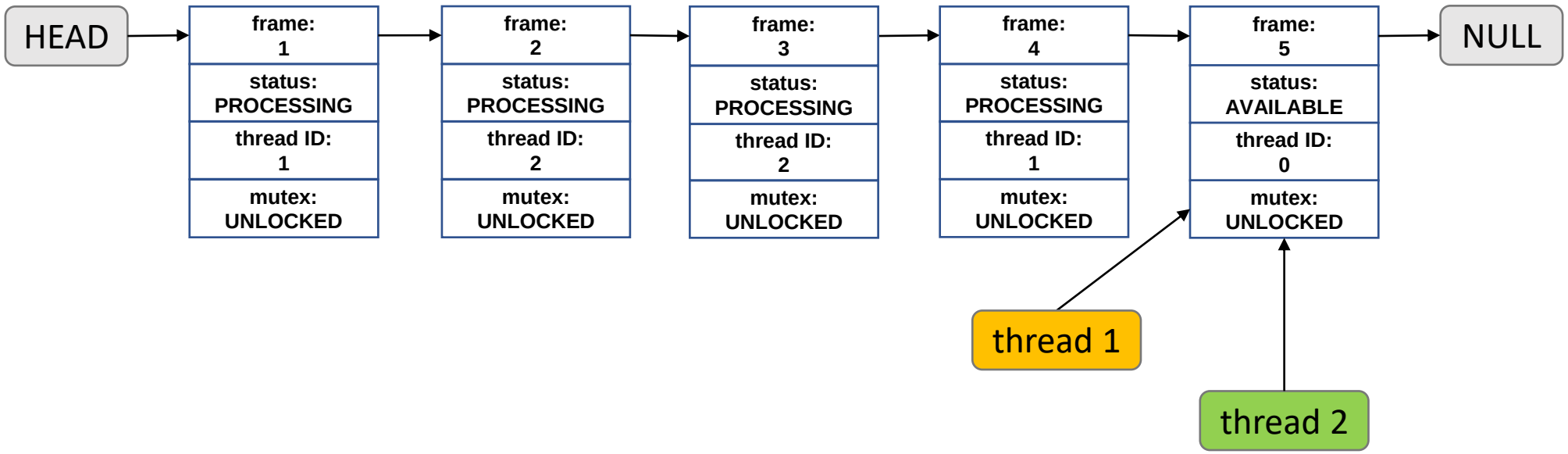
Using the mutex in POSIX threads: a practical example

Release the mutex lock on the individual element of the linked list:



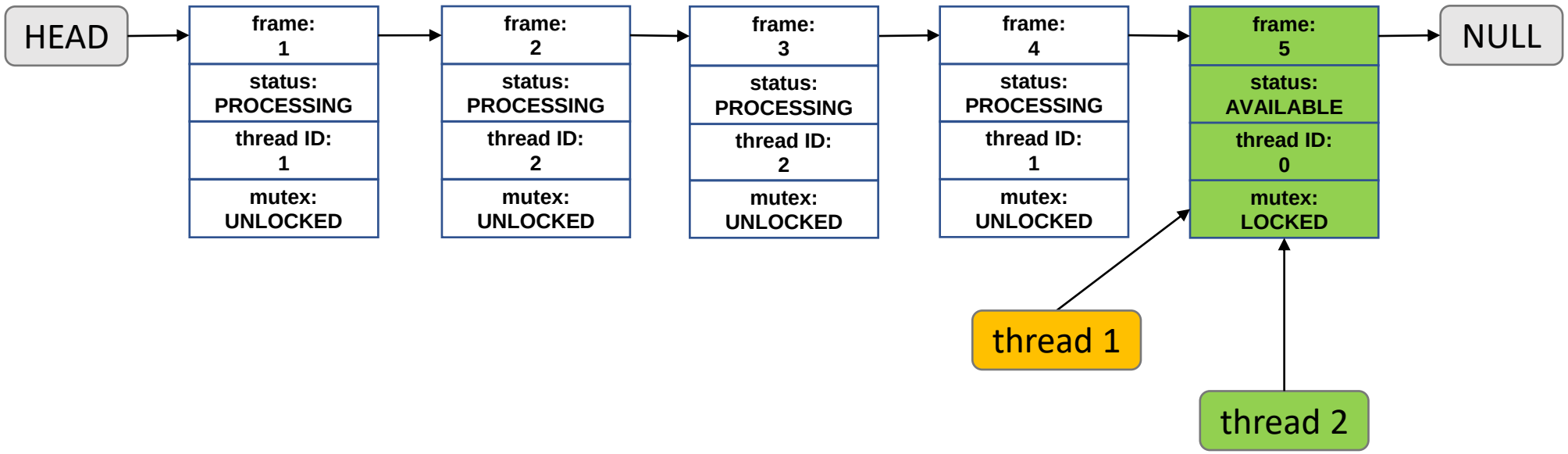
Using the mutex in POSIX threads: a practical example

Thread one and thread two advance to element five of the list. Both threads request the lock.



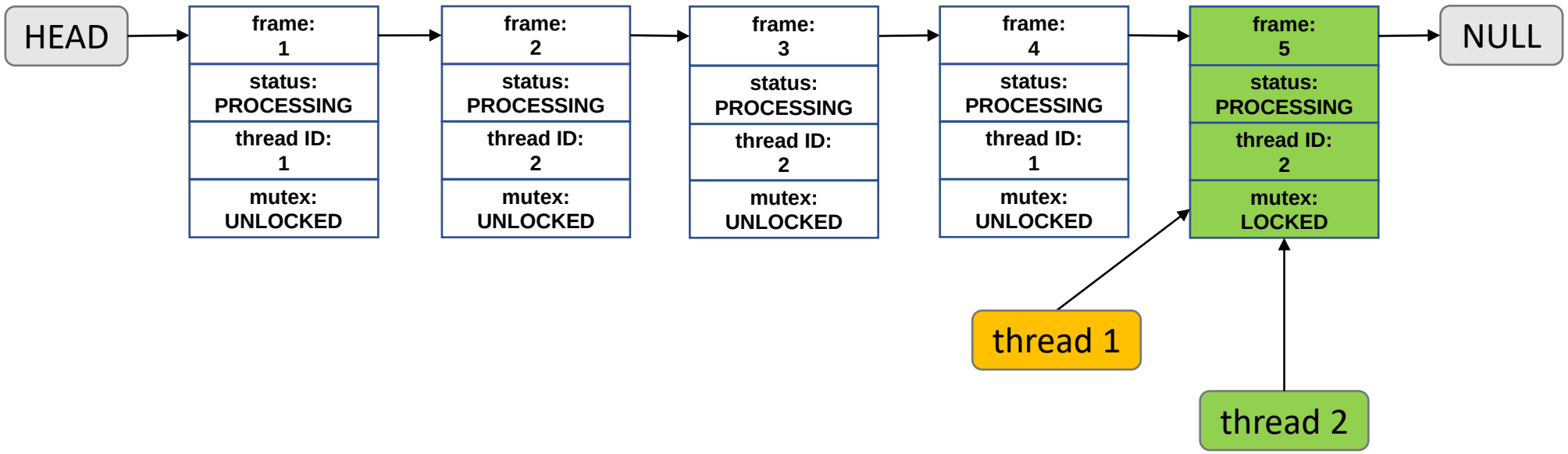
Using the mutex in POSIX threads: a practical example

The lock is granted to thread two. Check the status once again to ensure the element hadn't changed from "AVAILABLE" to "IN-PROCESS" by another thread before the lock was granted.



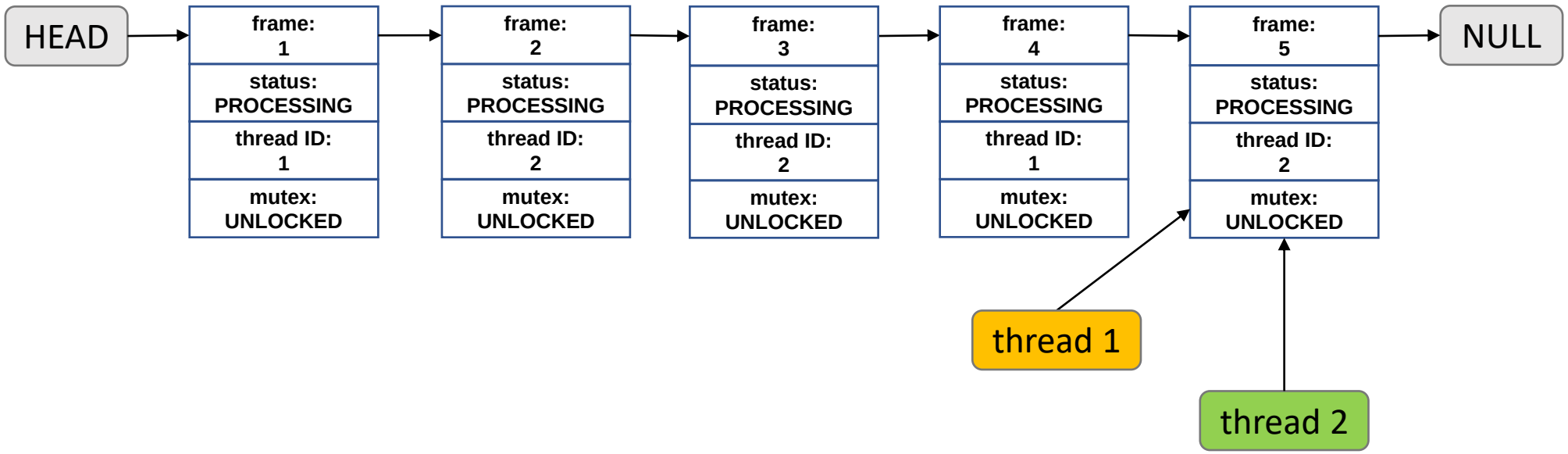
Using the mutex in POSIX threads: a practical example

Update the element in the job queue with status "PROCESSING" and the thread identifier that is processing this thread.



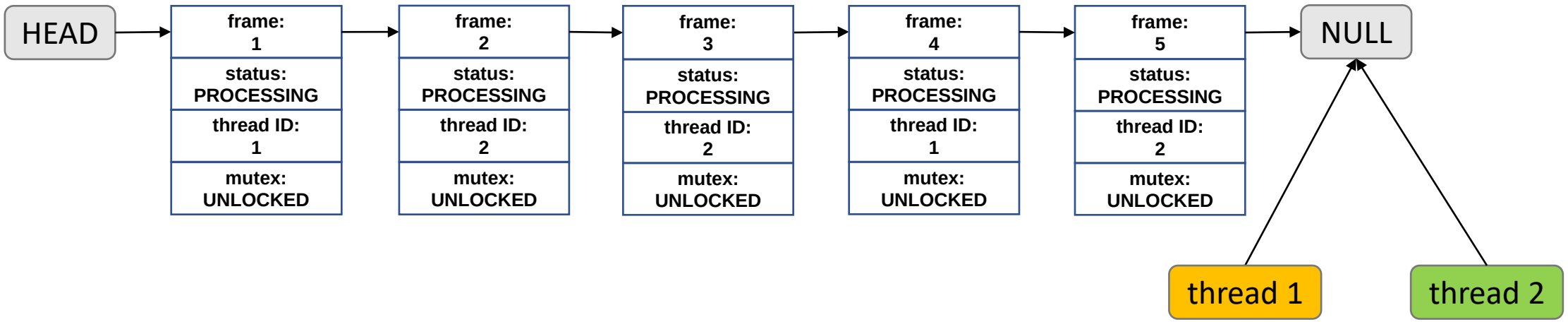
Using the mutex in POSIX threads: a practical example

Release the mutex lock on the individual element of the linked list:



Using the mutex in POSIX threads: a practical example

The threads have reached the end of the list. All elements have been processed.



POSIX thread functions for creating, locking, and unlocking a mutex

The following POSIX thread functions are used with a mutex:

- Initialize a mutex (at beginning of your program): *pthread_mutex_init()*
- Request a mutex lock: *pthread_mutex_lock()*
- Release a mutex lock: *pthread_mutex_unlock()*

For further reading...

I have uploaded example programs to Camino under the files folder then under *Example programs* folder:

- POSIX threads example: pthread_example.c
- POSIX threads mutex example: pthread_mutex_example.c

Two excellent books on POSIX threads:

- *Programming with POSIX Threads* by David R. Butenhof
- *Pthreads Programming: A POSIX Standard for Better Multiprocessing* by Dick Buttlar, Jacqueline Farrell, and Bradford Nichols