IIT Bombay

Lab 3
CS347m Operating Systems

Synchronization Primitives

Yash Sanjeev 180070068

17th April, 2021

Contents

1 Problem Statement

We have to design various synchronization primitives using the `pthreads` library available among the standard libraries for `C`. These primitives should be able to control the execution of threads in the following ways:

1. **Thread Ordering :** We should be able to run the threads in a specified order, provided by the user when initialising the primitive with an array of threads.
2. **Barrier Synchronization :** The threads wait at a barrier, a point in execution where all threads need to reach before any of them executes any further.
3. **Priority Synchronization :** The threads are provided with a priority order. Threads with higher priority must be executed before ones with lower priority.
4. **Starvation Prevention :** None of the threads in this primitive should be denied access to the critical section for a long time. Thus, threads who have starved for longer are given priority.

2 Overview of the Implementation

In all these implementations, we define a structure `order_t` which is initialised with an array of `pthread_t` and any other values that might be needed such as the order of execution or the priority index. We also need a wrapper structure called `wrap` where we provide information like thread identity `tid` and the synchronization primitive we intend to use along with the arguments of the function to be run by the thread.

We need a wrapper function `starter` around the function to be run by the thread. This wrapper is needed to accept the `wrap` structure, lock the thread and force it to sleep until certain conditions are satisfied. When these conditions are satisfied, the thread wakes up and `starter` calls the function that was originally intended to be run by the thread.

In each case, a header file has been created to implement the details of the synchronization primitive. This header file is then included in a file called `driver.c`, where the validity of the implementation is tested with a simple function `simplex` having targeted messages. With these details in mind, I present the different implementation details in the following sections.

3 Thread Ordering

3.1 Header File

The implementation is done in a header file named `ordering.h`. Thread ordering is performed by the struct `order_t` and the arguments for `simplex` are wrapped in the struct `wrap`.

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

typedef struct  __ordered_threads {
/*
  cap - number of threads handled by the primitive
  count - number of threads already executed
  active - flag denoting whether execution has started
  order - array denoting the order of execution
  mutex - common lock for locking all threads
  cond - common conditional variable to check which thread to run
*/
  int cap, count, active;
  int *order;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
} order_t;

typedef struct __wrapper_args {
/*
  tid - identifies each thread uniquely
  seq - the synchronization primitive
  routine - the original function to be run by the thread
  arg - the arguments required by the above routine
*/
  int tid;
  order_t *seq;
  void *(*routine)(void *);
  void *arg;
} wrap;
```

After creating the primitive, we need to initialize it with an array of threads, each running a function with its own set of arguments.

```
void initialize(order_t *seq, int cap, int *order) {
  seq->cap = cap;
  seq->count = 0;      // all threads pending execution
  seq->active = 0;     // execution hasn't started
  seq->order = order;

  assert(pthread_mutex_init(&seq->mutex, NULL) == 0);
  assert(pthread_cond_init(&seq->cond, NULL) == 0);
  printf("Initialized\n");
}
```

Observe the execution flag `active` hasn't been set. That will be done after all threads have been created and user calls the function `execute`. After initialization, we create threads using a custom wrapper `mythread_create`.

```
void mythread_create(order_t *seq, pthread_t *threads, wrap *kwarg) {
  int cap = seq->cap;

  for(int i=0; i<cap; i++) {
    // routine and its argument list already specified by user
    kwarg[i].seq = seq;
    kwarg[i].tid = i;
    pthread_create(&threads[i], NULL, starter, &kwarg[i]);
  }
}
```

The created threads call a wrapper function `starter` which sleeps until its turn for execution arrives. For further execution, it calls the `routine` included in the `wrap` structure provided.

```
void *starter(void *arg) {
  // cast the generic arg list in custom arg wrapper
  wrap *kwarg = (wrap*)arg;

  // extract various attributes from kwarg
  int tid = kwarg->tid;
  order_t *seq = kwarg->seq;

  // original function to be run by the user
```

```
  void *(*routine)(void *) = kwarg->routine;

  pthread_mutex_lock(&seq->mutex);
  // thread number to be executed
  int ind = seq->count;

  while(seq->order[ind] != tid || seq->active == 0)
    pthread_cond_wait(&seq->cond, &seq->mutex);

  // only run if execution started and turn has come
  routine(kwarg->arg);
  seq->count ++;

  // wake all sleeping threads
  pthread_cond_broadcast(&seq->cond);
  pthread_mutex_unlock(&seq->mutex);
  return NULL;
}
```

Now all threads may have been created but execution hasn't started because the attribute `seq->active` is `0`. Now we provide the `execute` function to the user, called after the point where all threads to be run along with the order has been provided.

```
void execute(order_t *seq) {
  pthread_mutex_lock(&seq->mutex);
  printf("Started Execution\n");
  seq->active = 1;

  // wake all sleeping threads
  pthread_cond_broadcast(&seq->cond);
  pthread_mutex_unlock(&seq->mutex);
}
```

At this point, we have started executing all the threads, and the condition variable `seq->cond` ensures that these threads are run in order. All that remains is for the `main` thread to wait for all these threads to execute.

```
void mythread_join(order_t *seq, pthread_t *threads) {
  int cap = seq->cap;
  // join all of the threads
```

```
  for(int i=0; i<cap; i++)
    pthread_join(threads[i], NULL);
}
```

3.2 Driver File

This file aims to validate the implementation of the primitive. We accept the number of threads `cap` to be run by the user. However, the priority array is created using a random permutation of the numbers `[1, cap]` to make testing faster. The permutation is created using the **Fisher-Yates** shuffling algorithm.

```
#include <stdlib.h>
#include "ordering.h"

int* permute(int n) {
  int *r = malloc(n * sizeof(int));

  for(int i=0; i<n; i++)
    r[i] = i;

  for (int i = n-1; i >= 0; --i){
    int j = rand() % (i+1);
    int temp = r[i];
    r[i] = r[j];
    r[j] = temp;
  }

  return r;
}
```

The basic function to be run is `simplex`, which simply prints what thread number is running right now. The `main` function initialized the threads to be run and the wrapper argument structure for each of them.

```
void *simplex(void *arg) {
  int *pos = arg;
  printf("Running thread #%d\n", *pos);
  return NULL;
}
```

```c
int main(){
  int cap;
  printf("Enter the number of threads: ");
  scanf("%d",&cap);

  int arg[cap];
  pthread_t threads[cap];
  int *order = permute(cap);

  for(int i=0; i<cap; i++)
    printf("%d ", order[i]);
  printf("is the random order\n");

  order_t dateko;
  wrap kwarg[cap];
  initialize(&dateko, cap, order);

  for(int i=0; i<cap; i++){
    arg[i] = i;
    kwarg[i].arg = &arg[i];
    kwarg[i].routine = simplex;
 }

  mythread_create(&dateko, threads, kwarg);
  execute(&dateko);
  mythread_join(&dateko, threads);

  return 0;
}
```

We run the code with the following commands on the terminal

```
gcc -Wall -pthread -o driver driver.c
./driver
```

which gives the following output:

```
Enter the number of threads: 8
2 6 5 1 0 3 4 7 is the random order
Initialized
Started Execution
```

```
Running thread #2
Running thread #6
Running thread #5
Running thread #1
Running thread #0
Running thread #3
Running thread #4
Running thread #7
```

Clearly the order of execution is exactly the same as the one specified by the array `int *order`. Thus, we have succeeded in creating a thread ordering primitive.

4 Barrier Synchronization

4.1 Header File

The implementation is done in a header file named `barrier.h`. Thread ordering is performed by the struct `order_t` and the arguments for `simplex` are wrapped in the struct `wrap`.

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

typedef struct  __ordered_threads {
/*
  cap - number of threads handled by the primitive
  barr - flag indicating whether all the threads have reached barrier
  active - flag denoting whether execution has started
  flag - array indicating whether each of the threads has reached
         barrier
  mutex - common lock for locking all threads
  cond - common conditional variable to check which thread to run
*/
  int cap, barr, active;
  int *flag;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
} order_t;
```

```
typedef struct __wrapper_args {
/*
  tid - identifies each thread uniquely
  seq - the synchronization primitive
  routine - the original function to be run by the thread
  arg - the arguments required by the above routine
*/
  int tid;
  order_t *seq;
  void *(*routine)(void *);
  void *arg;
} wrap;
```

After creating the primitive, we need to initialize it with an array of threads, each running a function with its own set of arguments.

```
void initialize(order_t *seq, int cap, int *order) {
  seq->cap = cap;
  seq->active = 0;     // execution hasn't started
  seq->barr = 0;       // all threads haven't reached barrier
  seq->flag = (int *)malloc(cap * sizeof(int));

  for(int i=0; i<cap; i++)
    // i-th thread hasn't reached barrier
    seq->flag[i] = 0;

  assert(pthread_mutex_init(&seq->mutex, NULL) == 0);
  assert(pthread_cond_init(&seq->cond, NULL) == 0);
  printf("Initialized\n");
}
```

Observe the execution flag `active` hasn't been set. That will be done after all threads have been created and user calls the function `execute`. After initialization, we create threads using a custom wrapper `mythread_create`.

```
void mythread_create(order_t *seq, pthread_t *threads, wrap *kwarg) {
  int cap = seq->cap;

  for(int i=0; i<cap; i++) {
    // routine and its argument list already specified by user
```

```
    kwarg[i].seq = seq;
    kwarg[i].tid = i;
    pthread_create(&threads[i], NULL, starter, &kwarg[i]);
  }
}
```

The created threads call a wrapper function `starter` which sleeps until its execution flag `seq->active` is set. For further execution, it calls the `routine` included in the `wrap` structure provided.

```
void *starter(void *arg) {
  wrap *kwarg = (wrap*)arg;

  order_t *seq = kwarg->seq;
  void *(*routine)(void *) = kwarg->routine;

  pthread_mutex_lock(&seq->mutex);
  while(seq->active == 0)
    pthread_cond_wait(&seq->cond, &seq->mutex);

  pthread_mutex_unlock(&seq->mutex);
  routine(arg);
  return NULL;
}
```

Now all threads may have been created but execution hasn't started because the attribute `seq->active` is `0`. Now we provide the `execute` function to the user, called after the point where all threads to be run have been provided.

```
void execute(order_t *seq) {
  pthread_mutex_lock(&seq->mutex);
  printf("Started Execution\n");
  seq->active = 1;

  // wake all sleeping threads
  pthread_cond_broadcast(&seq->cond);
  pthread_mutex_unlock(&seq->mutex);
}
```

All the functions executed in the threads must make a call to the function `barrier`, exactly at the point where they want all threads to reach before any further execution.

This function blocks any thread until all the threads have reached the barrier point.

```
void barrier(order_t *seq, int tid) {
  pthread_mutex_lock(&seq->mutex);
  // thread with id tid has reached the barrier
  seq->flag[tid] = 1;
  int cap = seq->cap;
  int barr = 1;

  for(int i=0; i<cap; i++) {
    if(seq->flag[i] == 0)
      // i-th thread has not reached the barrier
      barr = 0;
  }

  seq->barr = barr;
  while(seq->barr == 0)
    pthread_cond_wait(&seq->cond, &seq->mutex);

  // executed only when all elements of seq->flag are 1
  pthread_cond_broadcast(&seq->cond);
  pthread_mutex_unlock(&seq->mutex);
}
```

At this point, we have started executing all the threads, and the condition variable `seq->cond` ensures that these threads cross the barrier only after all threads have reached it. All that remains is for the `main` thread to wait for all these threads to execute.

```
void mythread_join(order_t *seq, pthread_t *threads) {
  int cap = seq->cap;
  // join all of the threads
  for(int i=0; i<cap; i++)
    pthread_join(threads[i], NULL);
}
```

4.2 Driver File

The driver file contains a function which makes a call to the `barrier` function in the header file above, and prints messages corresponding to the code it executes before and after the barrier point.

```c
#include <stdlib.h>
#include "barrier.h"

void *simplex(void *arg) {
  wrap *kwarg = (wrap *)arg;

  // unwrapping of kwarg
  int *pos = kwarg->arg;
  int tid = kwarg->tid;
  order_t *seq = kwarg->seq;

  printf("Running thread #%d before barrier\n", *pos);
  barrier(seq, tid);
  printf("Running thread #%d after barrier\n", *pos);

  return NULL;
}
```

The number of threads is accepted from the user, and a basic function `simplex` is run in each thread, printing messages before and after it reaches the barrier point.

```c
int main(){
  int cap;
  printf("Enter the number of threads: ");
  scanf("%d",&cap);

  int arg[cap];
  pthread_t threads[cap];

  order_t dateko;
  wrap kwarg[cap];
  initialize(&dateko, cap);

  for(int i=0; i<cap; i++){
    arg[i] = i;
    kwarg[i].arg = &arg[i];
    kwarg[i].routine = simplex;
  }
```

```
  mythread_create(&dateko, threads, kwarg);
  execute(&dateko);
  mythread_join(&dateko, threads);

  return 0;
}
```

We run the code with the following commands on the terminal

```
gcc -Wall -pthread -o driver driver.c
./driver
```

which gives the following output:

```
Enter the number of threads: 8
Initialized
Started Execution
Running thread #5 before barrier
Running thread #2 before barrier
Running thread #7 before barrier
Running thread #6 before barrier
Running thread #3 before barrier
Running thread #1 before barrier
Running thread #0 before barrier
Running thread #4 before barrier
Running thread #4 after barrier
Running thread #7 after barrier
Running thread #2 after barrier
Running thread #5 after barrier
Running thread #3 after barrier
Running thread #1 after barrier
Running thread #6 after barrier
Running thread #0 after barrier
```

Clearly all messages of "execution before the barrier" occur before any message of "execution after the barrier", which means we have succeeded in creating a barrier synchronization primitive.

5 Priority Synchronization

5.1 Header File

The header file is the exact same as `ordering.h` for the thread ordering problem, except for the wrapper function `starter` which is presented below.

```
int max(int *arr, int n) {
  //find max element of an array
  int m = arr[0];

  for(int i=0; i<n; i++){
    if(arr[i] > m)
      m = arr[i];
  }

  return m;
}

void *starter(void *arg) {
  wrap *kwarg = (wrap*)arg;

  int tid = kwarg->tid;
  order_t *seq = kwarg->seq;
  void *(*routine)(void *) = kwarg->routine;

  pthread_mutex_lock(&seq->mutex);
  int priority = seq->priority[tid];
  int cap = seq->cap;

  // sleep if execution flag not set
  while(priority < max(seq->priority, cap) || seq->active == 0)
    pthread_cond_wait(&seq->cond, &seq->mutex);

  // only run if priority is max among remaining threads
  printf("Running thread with priority %d\n", priority);
  routine(kwarg->arg);

  // set priority to 0 after execution
  seq->priority[tid] = 0;
```

```
  // wake up all the sleeping threads
  pthread_cond_broadcast(&seq->cond);
  pthread_mutex_unlock(&seq->mutex);
  return NULL;
}
```

5.2 Driver File

The driver file accepts the number of threads `cap` and max priority `m` allowed from the user, but creates the priority array itself by randomly choosing values in `[0, m]` for a total of `cap` times.

```
#include <stdlib.h>
#include "priority.h"

int* arbitrary(int n, int m) {
  int* r = malloc(n * sizeof(int));
  for(int i=0; i<n; i++)
    r[i] = rand()%m;

  return r;
}
```

The `main` function calls a function `simplex` which simply prints the thread identity.

```
void *simplex(void *arg) {
  int *pos = arg;
  printf("Running thread #%d\n", *pos);
  return NULL;
}

int main(){
  int cap, m;
  printf("Enter number of threads and max priority: ");
  scanf("%d %d", &cap, &m);

  int arg[cap];
  pthread_t threads[cap];
  int* priority = arbitrary(cap, m);
```

```c
  for(int i=0; i<cap; i++)
    printf("(%d %d) ", priority[i], i);
  printf("is the (priority, #thread)\n");

  order_t dateko;
  wrap kwarg[cap];
  initialize(&dateko, cap, priority);

  for(int i=0; i<cap; i++){
    arg[i] = i;
    kwarg[i].arg = &arg[i];
    kwarg[i].routine = simplex;
  }

  mythread_create(&dateko, threads, kwarg);
  execute(&dateko);
  mythread_join(&dateko, threads);

  return 0;
}
```

We run the code with the following commands on the terminal

```
gcc -Wall -pthread -o driver driver.c
./driver
```

which gives the following output:

```
Enter number of threads and max priority: 8 4
(3 0) (2 1) (1 2) (3 3) (1 4) (3 5) (2 6) (0 7) is the (
   priority, #thread)
Initialized
Started Execution
Running thread with priority 3
Running thread #5
Running thread with priority 3
Running thread #3
Running thread with priority 3
Running thread #0
Running thread with priority 2
Running thread #1
```

```
Running thread with priority 2
Running thread #6
Running thread with priority 1
Running thread #4
Running thread with priority 1
Running thread #2
Running thread with priority 0
Running thread #7
```

Clearly threads with higher priority are executed before ones with lower priority. Thus, we have succeeded in creating a priority synchronization primitive.

6 Starvation Prevention

We wish to prevent the starvation of any thread in our structure, by giving priority to threads that have been denied access to the critical section more often. Thus, there is an array maintained internally by the primitive where the priority of a thread is bumped up each time it is denied access to the critical section.

6.1 Header File

The implementation is done in a header file named (ironically) `starvation.h`. The implementation is pretty similar to the case of priority synchronization with the only changes being made in `initialize` function for initialization of the primitive `order_t` and the wrapper function `starter`.

```c
void initialize(order_t* seq, int cap) {
  seq->cap = cap;
  seq->active = 0;
  seq->priority = (int*)malloc(cap * sizeof(int));

  for(int i=0; i<cap; i++)
    // emulate the access denial of threads
    seq->priority[i] = rand()%cap;

  assert(pthread_mutex_init(&seq->mutex, NULL) == 0);
  assert(pthread_cond_init(&seq->cond, NULL) == 0);
  printf("Initialized\n");
}
```

We emulate the denial of threads by giving them random priority at initialization, so that we may test the validity of our implementation. The `starter` function also has a bunch of messages to make validation easier.

```
void *starter(void *arg) {
  wrap *kwarg = (wrap*)arg;

  int tid = kwarg->tid;
  order_t *seq = kwarg->seq;
  void *(*routine)(void *) = kwarg->routine;

  pthread_mutex_lock(&seq->mutex);
  int priority = seq->priority[tid];
  int cap = seq->cap;

  while(priority < max(seq->priority, cap) || seq->active == 0) {
    if(seq->active == 0)
      printf("Thread #%d started with priority %d\n",
              tid, priority);
    else {
      seq->priority[tid] ++;
      priority = seq->priority[tid];
      printf("Thread #%d has been denied access\n", tid);
    }

    pthread_cond_wait(&seq->cond, &seq->mutex);
  }

  printf("Running thread %d with final priority %d\n",
          tid, seq->priority[tid]);
  routine(kwarg->arg);
  seq->priority[tid] = 0;

  pthread_cond_broadcast(&seq->cond);
  pthread_mutex_unlock(&seq->mutex);
  return NULL;
}
```

6.2 Driver File

The driver file is the simplest, it only takes the number of threads to be included in the primitive, and provides it during initialization.

```
#include <stdlib.h>
#include <unistd.h>
#include "starvation.h"

void *simplex(void *arg) {
  int *pos = arg;
  printf("Ran thread #%d\n", *pos);
  return NULL;
}

int main(){
  int cap;
  printf("Enter number of threads: ");
  scanf("%d", &cap);

  int arg[cap];
  pthread_t threads[cap];

  order_t dateko;
  wrap kwarg[cap];
  initialize(&dateko, cap);

  for(int i=0; i<cap; i++){
    arg[i] = i;
    kwarg[i].arg = &arg[i];
    kwarg[i].routine = simplex;
  }

  mythread_create(&dateko, threads, kwarg);
  execute(&dateko);
  mythread_join(&dateko, threads);

  return 0;
}
```

We run the code with the following commands on the terminal

```
gcc -W -pthread -o driver driver.c
./driver
```

which gives the following output

```
Enter number of threads: 8
Initialized
Thread #0 started with priority 7
Thread #1 started with priority 6
Thread #2 started with priority 1
Thread #3 started with priority 3
Thread #4 started with priority 1
Started Execution
Running thread 0 with final priority 7
Ran thread #0
Thread #2 has been denied access
Thread #3 has been denied access
Thread #6 has been denied access
Thread #7 has been denied access
Thread #4 has been denied access
Thread #1 has been denied access
Running thread 5 with final priority 7
Ran thread #5
Thread #4 has been denied access
Thread #3 has been denied access
Thread #2 has been denied access
Running thread 1 with final priority 7
Ran thread #1
Thread #2 has been denied access
Running thread 3 with final priority 5
Ran thread #3
Thread #2 has been denied access
Thread #6 has been denied access
Thread #4 has been denied access
Running thread 7 with final priority 5
Ran thread #7
Thread #4 has been denied access
Running thread 2 with final priority 5
Ran thread #2
Running thread 4 with final priority 5
Ran thread #4
```

```
Running thread 6 with final priority 4
Ran thread #6
```

We can see that `thread #2` started with a priority of `1`, was denied access four times and finally ran with a priority of `5`. This fact can also be verified for other threads. Hence we have successfully avoided the problem of starvation by introducing flexible priority values which update as the threads starve.