Concurrent and Parallel Programming in Shared Memory Systems

Luis Miguel Pinho

July 2022

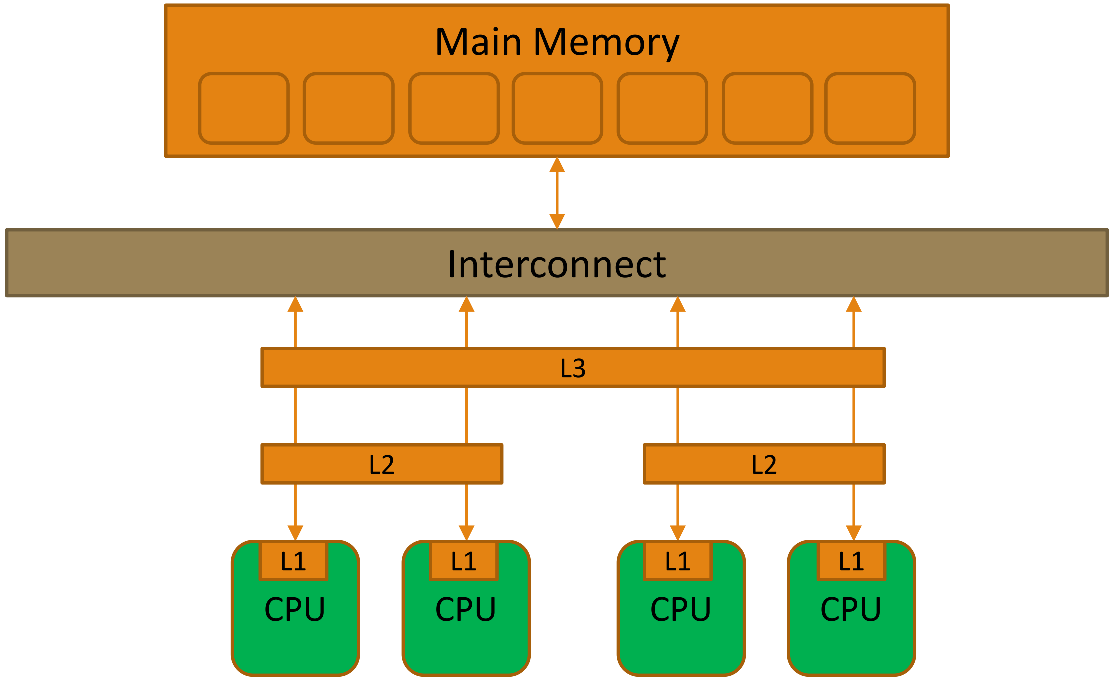
Instituto Superior de Engenharia do Porto

Preliminary reading

- This module requires a basic understanding of computer architecture and the C programming language
- Although not required, preliminary reading of the following online resources help with understanding the concepts that will be discussed in the classes:
  - An Introduction to Parallel Programming, P Pacheco, M. Malensek, University of San Francisco, 2021, available at https://www.cs.usfca.edu/~mmalensek/cs521/schedule/materials/pthreads.pdf
  - Pthread Tutorial, P. Chapin, Vermont Technical College, 2020, available at http://lemuria.cis.vtc.edu/~pchapin/TutorialPthread/pthread-Tutorial.pdf

Shared Memory Systems

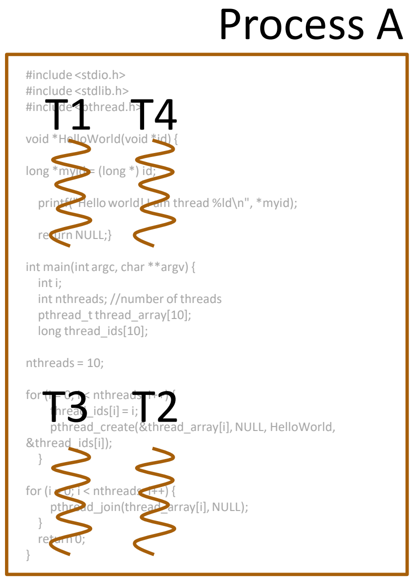
- In this course, we will focus on shared memory systems
  - All cores can access the same physical memory of the system
  - Although it is usually not that simple … later we will discuss the impact of caches



- We will not deal with heterogenous systems or distributed memory systems

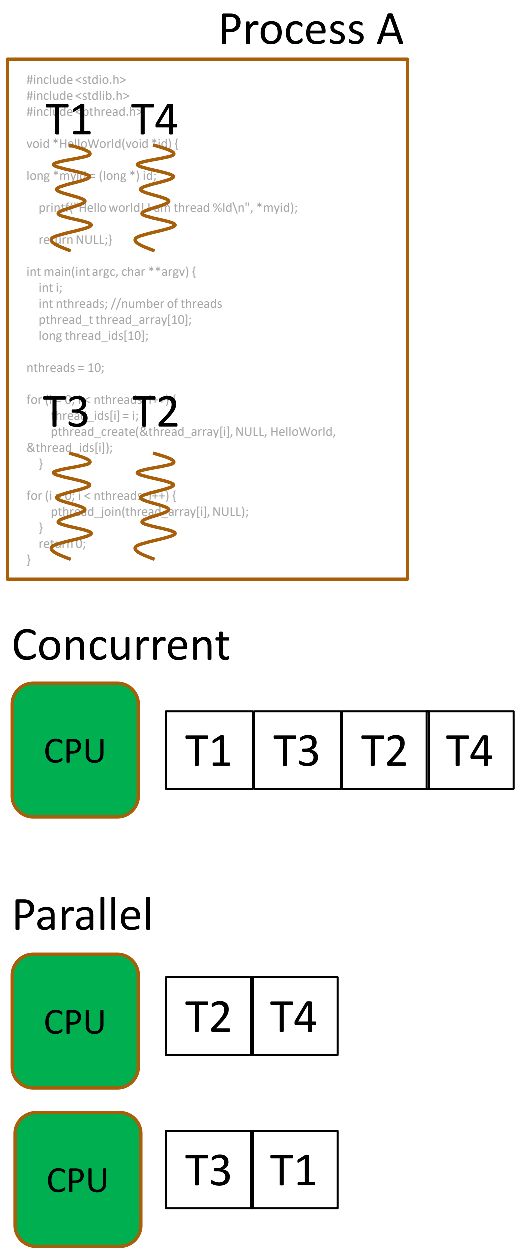
Threads

- Processes are a heavy unit
  - Dedicated memory space, file descriptors, etc.
  - Managing and switching between processes has large overhead
  - Data sharing between process is complex and heavy
- Threads are lightweight units of execution
  - Single process with multiple threads of control
  - Concurrent and parallel activities sharing the same context (code, memory, files, etc.)
  - Allows for less overhead, more efficient sharing, easy to scale
  - Sharing of data is easy, but also easy to make mistakes



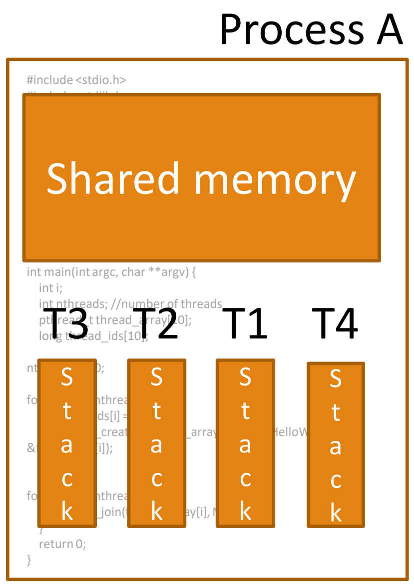
Threads

- Concurrency
  - A way of structuring software, where the functionalities are provided by a set of cooperating executing entities
  - Allows also to model the inherent concurrency in the physical world
  - May execute in parallel or time share the same processor
- Parallelism
  - An implementation feature which allows to execute multiple activities at the same time (multi-core CPUs, GPUs, TPUs)
  - Concurrent executing units are a natural mapping for parallel execution (but not the only one)

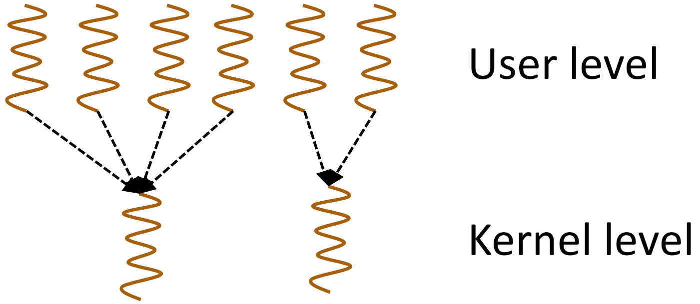


Threads

- Threads can be created/terminated dynamically
- All threads access the same memory area
  - But each thread as its own separate stack (private variables)
- Threads communicate by shared variables (e.g. global variables)
- Threads also use shared variables to synchronize execution



- Threads can be user level or kernel level
  - If user level, all of the thread management is done at user level, the kernel is not aware of the threads
    - Multiple user threads can be mapped to a single kernel entity
  - We will focus in implementations that map one to one



POSIX Threads

- International standard (POSIX: Portable Operating System Interface for UNIX)
  - Specifies the interface (and some behaviour) to operating systems
  - In some cases, specification may be optional (may vs shall)
- POSIX Threads (pthreads)
  - IEEE 1003.1c: Standard extension for threads (C language)
  - Interface to create and synchronize threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_func) (void *), void *arg)
pthread_join(pthread_t thread, void **return_val)

int pthread_mutex_init( pthread_mutex_t *mutex, pthread_muttexattr_t *attr)
int pthread_mutex_destroy( pthread_mutex_t *mutex)

int pthread_cond_wait(pthread_cond_t*cond, pthread_mutex_t*mutex)
int pthread_cond_signal(pthread_cond_t*cond)

// …
```

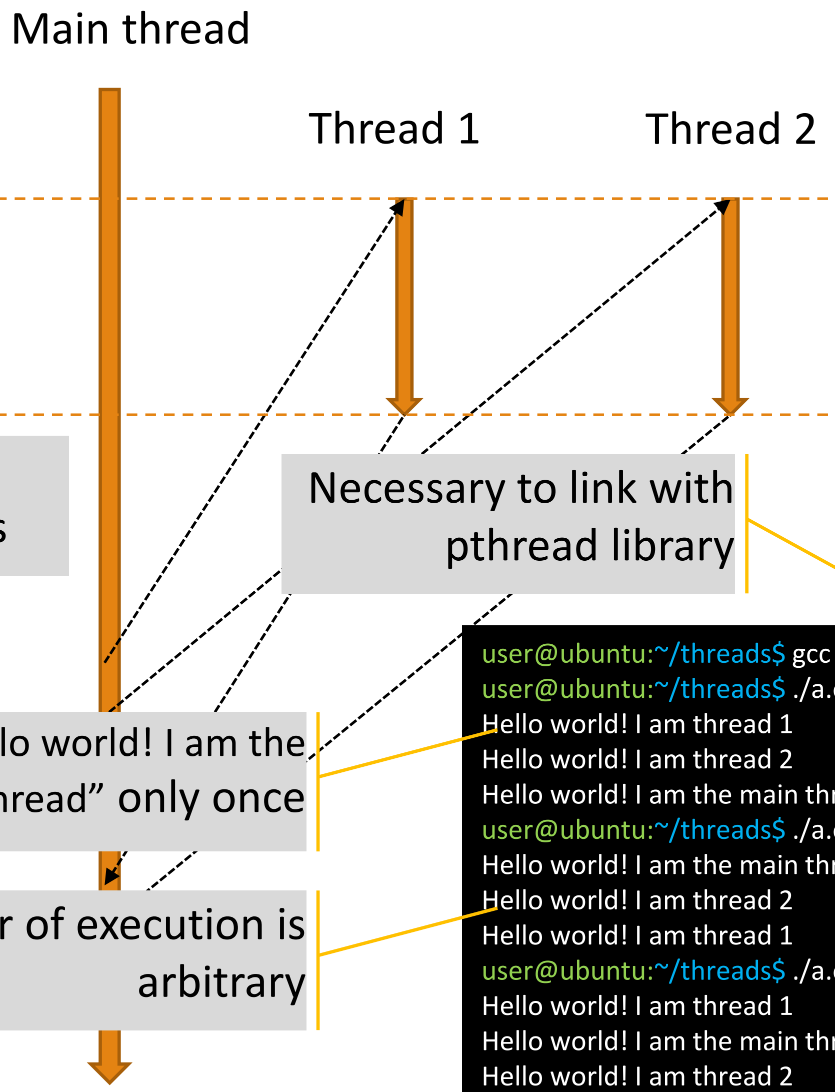
Creating and using threads

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

void* HelloWorld(void* id) {
   long myid = * (long*) id;
   printf("Hello world! I am thread %ld\n", myid);
   return NULL;
}

int main() {
   long id1 = 1, id2 = 2;
   pthread_t thread_id1, thread_id2;

   pthread_create(&thread_id1, NULL, HelloWorld, &id1);
   pthread_create(&thread_id2, NULL, HelloWorld, &id2);

   printf("Hello world! I am the main thread \n");

   pthread_join(thread_id1, NULL);
   pthread_join(thread_id2, NULL);

   return 0;
}
```

Created threads execute only the function, return terminates thread

Creates a new thread of execution, asynchronous

pthread_join suspends the main thread until the joining thread finishes



Necessary to link with pthread library

“Hello world! I am the main thread” only once

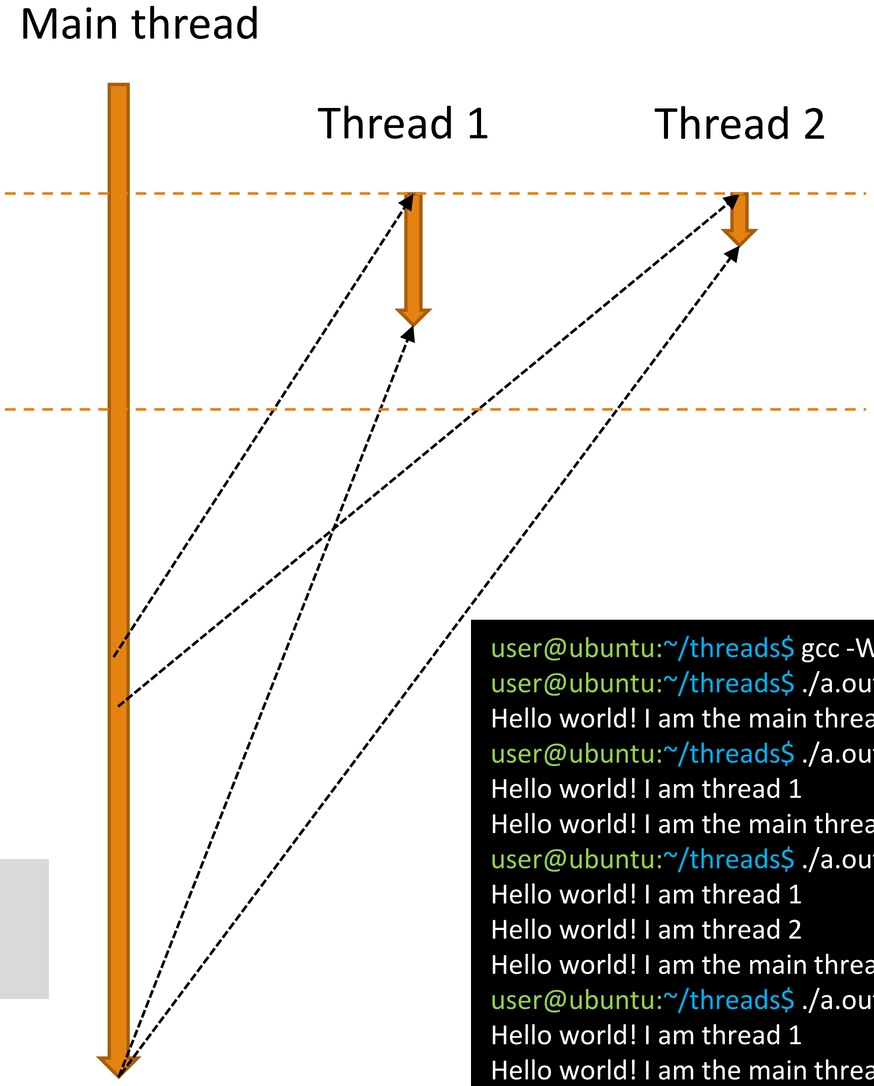
Order of execution is arbitrary

```
user@ubuntu:~/threads$ gcc -Wall th.c -lpthread
user@ubuntu:~/threads$ ./a.out
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am the main thread
user@ubuntu:~/threads$ ./a.out
Hello world! I am the main thread
Hello world! I am thread 2
Hello world! I am thread 1
user@ubuntu:~/threads$ ./a.out
Hello world! I am thread 1
Hello world! I am the main thread
Hello world! I am thread 2
```

Creating and using threads

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

void* hello_world(void* id) {
   long myid = * (long*) id;
   printf("Hello world! I am thread %ld\n", myid);
   return NULL;
}

int main() {
   long id1 = 1, id2 = 2;
   pthread_t thread_id1, thread_id2;

   pthread_create(&thread_id1, NULL, hello_world, &id1);
   pthread_create(&thread_id2, NULL, hello_world, &id2);

   printf("Hello world! I am the main thread \n");

   // pthread_join(thread_id1, NULL);
   // pthread_join(thread_id2, NULL);

   return 0;
}
```

If no join, threads can be abruptly terminated




```
user@ubuntu:~/threads$ gcc -Wall th.c -lpthread
user@ubuntu:~/threads$ ./a.out
Hello world! I am the main thread
user@ubuntu:~/threads$ ./a.out
Hello world! I am thread 1
Hello world! I am the main thread
user@ubuntu:~/threads$ ./a.out
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am the main thread
user@ubuntu:~/threads$ ./a.out
Hello world! I am thread 1
Hello world! I am the main thread
```

Creating and using threads

```
int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_func) (void *),
        void *arg
)
```

pthread_t is an opaque data structure; actual data is implementation defined. A programmer cannot access the internal data.

Specification guarantees that it holds the required data to univocally identify the specific thread.

pthread_create expects the address, so that it can initialize the value

The address of the function (name of function in C). The function needs to accept one argument, a pointer to void, and also return a pointer to void.

A pointer to void can be cast to a pointer of any type.

The actual argument that is passed to the function

Creating and using threads

```
int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_func) (void *),
        void *arg
)
```

Thread attributes, allows to control some of the thread features and behaviour:
- Changing the stack size
- Changing the thread scheduling
- …

Default is usually sufficient (use NULL).

Creating and using threads

```
pthread_join(
   pthread_t thread,
   void **return_val
)
```

The id of the thread to wait

The address of the variable to hold the returned value from the thread function

The return is an address, so it is necessary to give join a double pointer

```
void pthread_exit(void *retval)
```

Terminating the thread can be explicit with pthread_exit, instead of return from function

```
pthread_t pthread_self(void)
```

Allows for a thread to get its identifier

Sharing data

This program calculates in parallel the maximum value of a vector

```
#define SIZE 1000
#define N_THREADS 4

long vector[SIZE];

int main() {

  pthread_t thread_id[N_THREADS];
  long index[N_THREADS];
  void *ret_val;
  long local_max;
  long max;

  fill(vector);

  for(int i = 0; i < N_THREADS; i++)
        index[i] = i;

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, max_func, &index[i]);

  max = vector[0];
  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], &ret_val);
    local_max = (long) ret_val;
    if(local_max > max) max = local_max;
  }
  printf("Max = %ld\n", max);
  return 0;
}
```

N_THREADS are created, each one will search in one split of the vector

The "return" of each thread is compared to the global max value

The max_func function is used by each working thread to calculate a local max value

```
void* max_func(void* index) {
  long my_index = *(long*)index;
  long local_max = vector[my_index*SIZE/N_THREADS];

  for(int i = my_index*SIZE/N_THREADS; i < (my_index +1)*SIZE/N_THREADS; i++)
        if(vector[i] > local_max) local_max = vector[i];

  return (void*) local_max;
}
```

To simplify code, we will be always assuming that SIZE is multiple of N_THREADS

Sharing data

vector is in the heap and shared by all threads

```
#define SIZE 1000
#define N_THREADS 4

long vector[SIZE];

int main() {

  pthread_t thread_id[N_THREADS];
  long index[N_THREADS];
  void *ret_val;
  long local_max;
  long max;

  fill(vector);

  for(int i = 0; i < N_THREADS; i++)
        index[i] = i;

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, max_func, &index[i]);

  max = vector[0];
  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], &ret_val);
    local_max = (long) ret_val;
    if(local_max > max) max = local_max;
  }
  printf("Max = %ld\n", max);
  return 0;
}
```

```
void* max_func(void* index) {
  long my_index = *(long*)index;
  long local_max = vector[my_index*SIZE/N_THREADS];

  for(int i = my_index*SIZE/N_THREADS; i < (my_index +1)*SIZE/N_THREADS; i++)
    if(vector[i] > local_max) local_max = vector[i];

  return (void*) local_max;
}
```

The index vector is in the main thread stack, but still accessible by other threads (using its address)

Returning the local max value is using the functional return value which is copied to the join call

Sharing data

vector is in the heap and shared by all threads

```
#define SIZE 1000
#define N_THREADS 4

long vector[SIZE];

int main() {

  pthread_t thread_id[N_THREADS];
  long index[N_THREADS];
  void *ret_val;
  long local_max;
  long max;

  fill(vector);

  for(int i = 0; i < N_THREADS; i++)
        index[i] = i;

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, max_func, &index

  max = vector[0];
  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], &ret_val);
    local_max = (long) ret_val;
    if(local_max > max) max = local_max;
  }
  printf("Max = %ld\n", max);
  return 0;
}
```

```
void* max_func(void* index) {
  long my_index = *(long*)index;
  long local_max = vector[my_index*SIZE/N_THREADS];

  for(int i = my_index*SIZE/N_THREADS; i < (my_index +1)*SIZE/N_THREADS; i++)
    if(vector[i] > local_max) local_max = vector[i];

  return (void*) local_max;
}
```

RED FLAGS:

1. All threads are accessing variables in parallel. Need to guarantee there are no inconsistencies. In this case it is easy to check:

- Shared variables (vector and index array) are written before threads being created.
- Threads only read from these variables

But it is always the responsibility of the programmer to check

2. Several unchecked conversions between void* and long. These are not portable, need to be checked in each platform. Again, the responsibility of the programmer (-Wall).

Sharing data

```
#define SIZE 1000
#define N_THREADS 4

long vector[SIZE];

int main() {

   pthread_t thread_id[N_THREADS];
   long index[N_THREADS];
   void *ret_val;
   long local_max;
   long max;

   fill(vector);

   for(int i = 0; i < N_THREADS; i++)
         index[i] = i;

   for (int i = 0 ; i<N_THREADS; i++)
      pthread_create(&thread_id[i], NULL, max_func, &i);

   max = vector[0];
   for (int i = 0 ; i<N_THREADS; i++){
      pthread_join(thread_id[i], &ret_val);
      local_max = (long) ret_val;
      if(local_max > max) max = local_max;
   }
   printf("Max = %ld\n", max);
   return 0;
}
```

```
void* max_func(void* index) {
   long my_index = *(long*)index;
   long local_max = vector[my_index*SIZE/N_THREADS];

   for(int i = my_index*SIZE/N_THREADS; i < (my_index +1)*SIZE/N_THREADS; i++)
         if(vector[i] > local_max) local_max = vector[i];

   return (void*) local_max;
}
```

Usual mistake: the variable i is being changed by the main thread. By passing its address to the other threads, the values will be inconsistent

Sharing data

```c
#define SIZE 1000
#define N_THREADS 4

long vector[SIZE];

typedef struct data_type{
  int index;
  int local_max;
}data_type;

int main() {

  pthread_t thread_id[N_THREADS];
  data_type data[N_THREADS];
  int max;

  fill(vector);

  for(int i = 0; i < N_THREADS; i++)
        data[i].index = i;

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, max_func, &data[i]);

  max = vector[0];
  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], NULL);
    if(data[i].local_max > max) max = data[i].local_max;
  }
```

```c
void* max_func(void* data) {
  data_type* my_data = (data_type*)data;
  my_data->local_max = vector[my_data->index*SIZE/N_THREADS];

  for(int i = my_data->index*SIZE/N_THREADS; i < (my_data->index+1)*SIZE/N_THREADS; i++)
        if(vector[i] > my_data->local_max) my_data->local_max = vector[i];

  return NULL;
}
```

Solution using a data structure to share data

Sharing data

```c
#define SIZE 1000
#define N_THREADS 4

long vector[SIZE];

typedef struct data_type{
  int index;
  int local_max;
}data_type;

int main() {

  pthread_t thread_id[N_THREADS];
  data_type data[N_THREADS];
  int max;

  fill(vector);

  for(int i = 0; i < N_THREADS; i++)
        data[i].index = i;

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, max_func, &data[i]);

  max = vector[0];
  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], NULL);
    if(data[i].local_max > max) max = data[i].local_max;
  }
```

```c
void* max_func(void* data) {
  data_type* my_data = (data_type*)data;
  my_data->local_max = vector[my_data->index*SIZE/N_THREADS];

  for(int i = my_data->index*SIZE/N_THREADS; i < (my_data->index+1)*SIZE/N_THREADS; i++)
        if(vector[i] > my_data->local_max) my_data->local_max = vector[i];

  return NULL;
}
```

Eventually you could malloc the return value in the thread function, and pass the pointer to main.

The main thread would be responsible for freeing the data

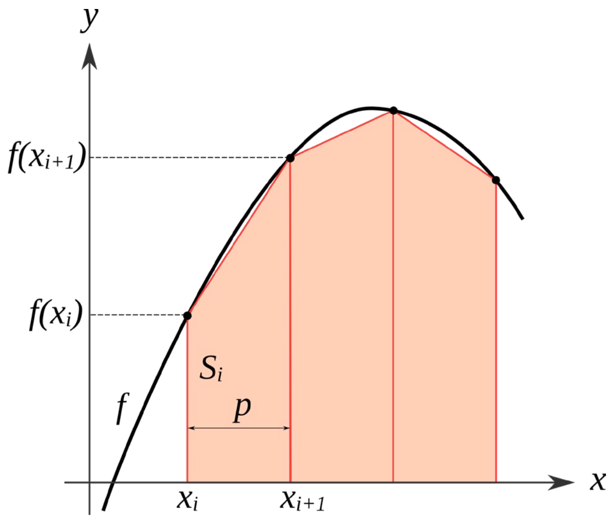
May lead to erroneous behavior, use only when large structures which may be required (or not) at execution

The same with the required structures for the threads. Examples until now, the number of threads is fixed, so no need to allocate dynamic memory.

However, if the number of threads is variable, we may need to allocate one struct per thread.

Hands-on

- Exercise: parallelize the trapezoidal rule to calculate the integral



$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k.$$

Source: wikipedia

```
double integral (double a, double b, int n){
  double x, p, sum=0, integral;

  p=fabs(b-a)/n;

  for(int i=1; i<n; i++){
    x = a + i * p;
    sum=sum + f(x);
  }

  integral=(p/2)*(f(a)+f(b)+2*sum);

  return integral;
}
```

Synchronizing threads

- Race Condition
  - A race condition occurs when two threads simultaneously try to use the same resource (e.g. a shared variable)
  - Remember, it is the programmer responsibility to guarantee it is safe
  - Usually, the part of code which makes the access is called the critical section

```
int ticket_counter;

int get_ticket(){
  int ticket = ticket_counter;
  printf("Issue ticket number %d\n", ticket);
  ticket_counter++;
  return ticket;
 }

void* ticket_func(void* data) {

  int my_ticket = get_ticket();

  printf("My ticket is %d\n", my_ticket);

  return NULL;
}
```

```
int main() {

  pthread_t thread_id[N_THREADS];

  ticket_counter = 0;

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, ticket_func, NULL);

  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], NULL);
  }
```

Thread interrupted between accessing the ticket and increasing the counter value

```
user@ubuntu:~/threads$ ./a.out
Issue ticket number 0
My ticket is 0
Issue ticket number 0
My ticket is 0
Issue ticket number 2
My ticket is 2
Issue ticket number 3
My ticket is 3
Issue ticket number 4
My ticket is 4
Issue ticket number 5
My ticket is 5
Issue ticket number 6
My ticket is 6
Issue ticket number 7
My ticket is 7
Issue ticket number 8
My ticket is 8
Issue ticket number 8
My ticket is 8
```

Synchronizing threads

- Mutual exclusion
  - One thread is the “owner” of the resource, others have to wait

```
pthread_mutex_t mut;
```

Mutex - mutual exclusion control object

```
int pthread_mutex_init( pthread_mutex_t *mutex, pthread_muttexattr_t *attr)
int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Creation and destruction of a mutex

Thread wanting to use the shared resource locks the mutex (or tries to lock)

When finished to use the shared resource, unlocks it allowing other threads to proceed

Synchronizing threads

```
int ticket_counter;
pthread_mutex_t ticket_mutex;


int get_ticket(){
  pthread_mutex_lock(&ticket_mutex);
  int ticket = ticket_counter;
  printf("Issue ticket number %d\n", ticket);
  ticket_counter++;
  pthread_mutex_unlock(&ticket_mutex);
  return ticket;
 }


void* ticket_func(void* data) {
  int my_ticket = get_ticket();

  printf("My ticket is %d\n", my_ticket);

  return NULL;
}
```

```
int main() {

  pthread_t thread_id[N_THREADS];

  ticket_counter = 0;
  pthread_mutex_init(&ticket_mutex, NULL);

  for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, ticket_func, NULL);

  for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], NULL);
  }
  pthread_mutex_destroy(&ticket_mutex);


  return 0;
}
```

Synchronizing threads

- The wrong use of mutual exclusion may cause deadlocks to occur

```
pthread_mutex_t ticket_mutex_1;
pthread_mutex_t ticket_mutex_2;


void* thread_1_func(void* data) {

  pthread_mutex_lock(&ticket_mutex_1);
  pthread_mutex_lock(&ticket_mutex_2);
  // …
  return NULL;
}

void* thread_2_func(void* data) {

  pthread_mutex_lock(&ticket_mutex_2);
  pthread_mutex_lock(&ticket_mutex_1);
  // …
  return NULL;
}
```

(1) Thread 1 locks mutex 1

(4) And blocks in mutex 2

(2) Thread 2 locks mutex 2

(3) And blocks in mutex 1

Deadlock

Both threads are blocked waiting for the other thread to release the mutex

Rule of thumb, always acquire the locks in the same order. May not be always possible.

Synchronizing threads

- Mutexes implement synchronization through controlling access to data
- Conditional variables allow to synchronize threads on the value of data
  - An object that allows a thread to suspend until a certain condition occurs
  - When the event or condition occurs, another thread can signal the release of the suspended thread
  - A conditional variable is a shared resource, so a mutex needs to be used
  - Note that it is the role of the programmer to implement the logic behind the synchronization

Synchronizing threads

```
pthread_cond_t cond_obj;

int pthread_cond_init(  pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);


int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec*abstime);


int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t*cond);
```

Mutex associated with wait

A thread calling wait needs to have the lock in the mutex. When the thread suspends, the lock is released (allowing other threads to use the lock).

When the suspended thread is released, it will be given the mutex again (only released when mutex is available)

Signal will release one of the suspended threads

Broadcast will release all, which need to compete for the resource

Synchronizing threads

```
// thread 1

if (elements == 0) 1
   pthread_cond_wait(&cond_var,      ) 5


// thread 2

elements ++; 2
if(elements > 0) 3
   pthread_cond_signal(&cond_var); 4
```

Simple logic

Thread 1 will check if there are elements to process, if not, waits

Thread 2 adds elements and signals thread 1 to continue

This will not work

Elements is a shared variable, but it is not protected.

If execution order is the one shown, signal is lost, as there is no thread waiting

Synchronizing threads

```
// thread 1
pthread_mutex_lock(&mux);
if (elements == 0)
   pthread_cond_wait(&cond_var, &mux );
pthread_mutex_unlock(&mux);

// thread 2
pthread_mutex_lock(&mux);
elements ++;
if(elements > 0)
   pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mux);
```

Using a lock guarantees that the concurrent updates and test of the elements shared variable are protected

If thread suspends in the wait, the OS unlocks the mutex, which allows thread 2 to access the critical section

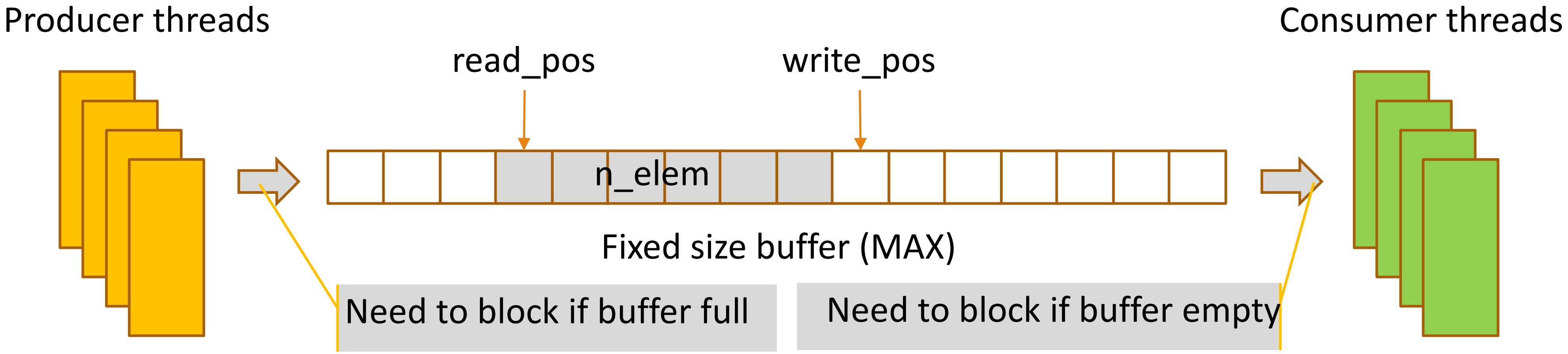
When thread 2 unlocks the mutex, thread 1 can resume from the wait with the mutex locked again

Synchronizing threads

```
// thread 1
pthread_mutex_lock(&mux);
while (elements == 0)
   pthread_cond_wait(&cond_var, &mux);
pthread_mutex_unlock(&mux);

// thread 2
pthread_mutex_lock(&mux);
elements ++;
if(elements > 0)
   pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mux);
```

Spurious wakeups may break the logic. In between the time when the condition variable was signalled and when the waiting thread finally ran, another thread may have changed the condition. And in multicores, signals may turn into broadcasts, which will wake all threads.
Therefore, whenever a condition wait returns, the thread should re-evaluate to determine whether it can safely proceed.

Example: bounded buffer

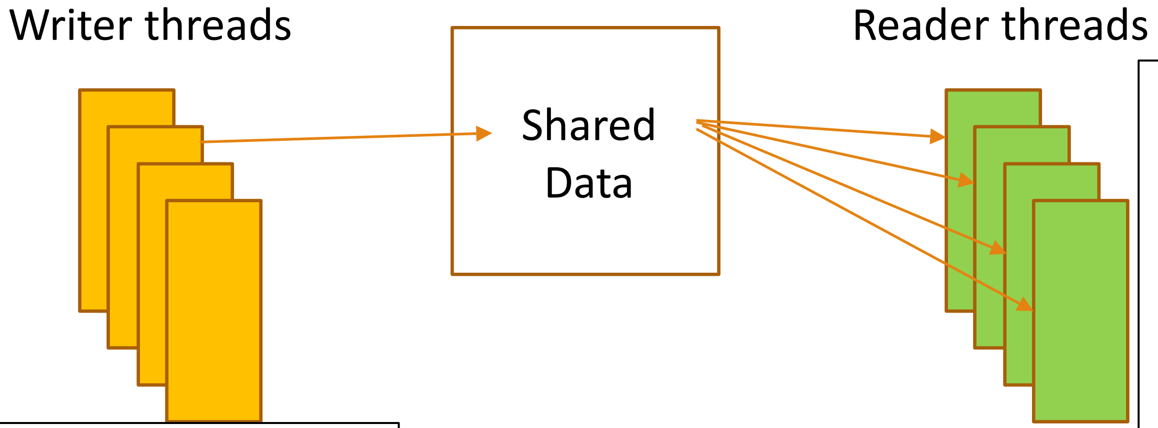



```
void write(int elem){
  pthread_mutex_lock(&buffer_mux);
  while(n_elem == MAX)
     pthread_cond_wait(&not_full, &buffer_mux);
  buffer[write_pos++] = elem;
  if(write_pos == MAX) write_pos = 0;
  n_elem++;
  if(n_elem == 1)
     pthread_cond_signal(&not_empty);
  pthread_mutex_unlock(&buffer_mux);
}
```

```
void read(int *elem){
  pthread_mutex_lock(&buffer_mux);
  while(n_elem == 0)
     pthread_cond_wait(&not_empty, &buffer_mux);
  *elem = buffer[read_pos++];
  if (read_pos == MAX) read_pos = 0;
  n_elem --;
  if(n_elem == MAX-1)
    pthread_cond_signal(&not_full);
  pthread_mutex_unlock(&buffer_mux);
}
```

Example: Reader/Writer

- Classical problem
  - A shared data can be accessed for reading by multiple readers
  - But only one writer at a time (and if there is a writer, there cannot be readers)



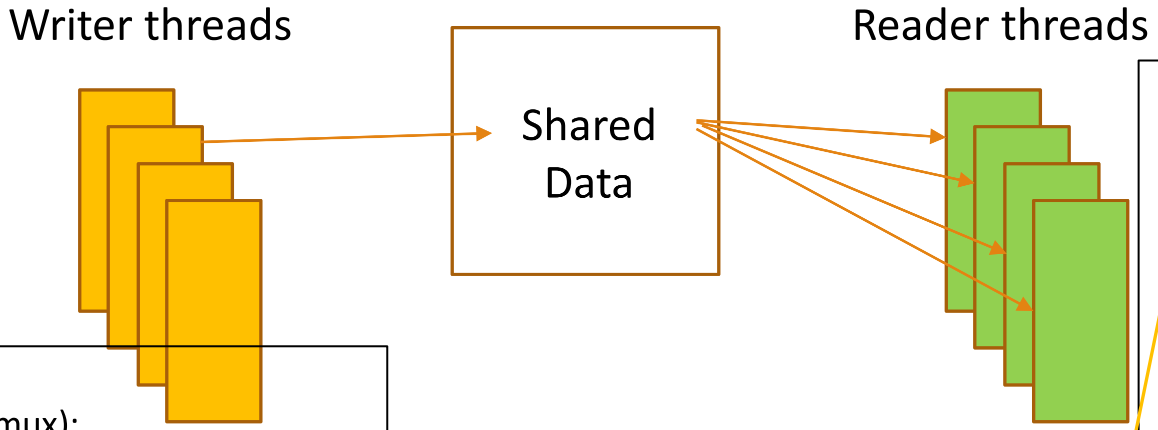

```
void write(int elem){
  pthread_mutex_lock(&writer_mux);
  shared_data = elem;
  pthread_mutex_unlock(&writer_mux);
}
```

```
void read(int *elem){
  pthread_mutex_lock(&reader_mux);
  reader_count++;
  if(reader_count == 1)
     pthread_mutex_lock(&writer_mux);
  pthread_mutex_unlock(&reader_mux);

  *elem = shared_data;

  pthread_mutex_lock(&reader_mux);
  reader_count--;
  if(reader_count == 0)
     pthread_mutex_unlock(&writer_mux);
  pthread_mutex_unlock(&reader_mux);

}
```

The logic is correct, but the implementation is not allowed
Unlocking a mutex if the thread does not have the lock is not allowed in pthread mutexes

Example: Reader/Writer

- Solutions
  - Use a different synchronization primitive (pthreads read write lock, semaphores)
  - Use conditional variables



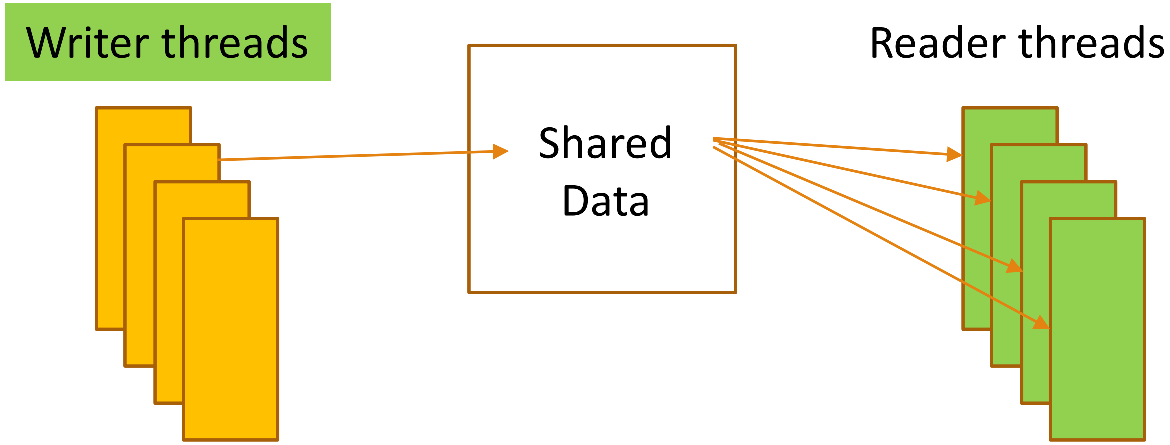

```
void write(int elem){
  pthread_mutex_lock(&shared_mux);
  while (reader_count > 0)
    pthread_cond_wait(&shared_cv, &shared_mux);

  shared_data = elem;

  pthread_mutex_unlock(&shared_mux);
}
```

Note that writers may eventually starve, if readers continue to appear

```
void read(int *elem){
  pthread_mutex_lock(&shared_mux);
  reader_count++;
  pthread_mutex_unlock(&shared_mux);

  *elem = shared_data;

  pthread_mutex_lock(&shared_mux);
  reader_count--;
  if(reader_count == 0)
    pthread_cond_signal(&shared_cv);
  pthread_mutex_unlock(&reader_mux);

}
```

Hands-on

- Change the Reader/Writer implementation to give preference to writers



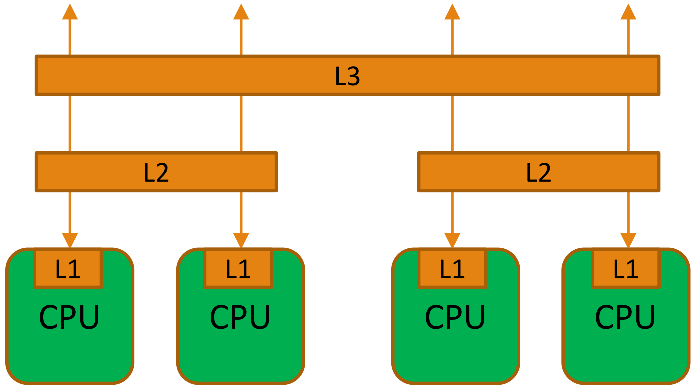
Going further

- As seen, mutex and conditional variables may not be sufficient
  - Other synchronization mechanisms exist, that can be used (even non pthread ones)
    - Read-write locks (Implementations are allowed to favour writers to avoid writer starvation)
    - Barriers - Synchronize a group of threads – all need to reach the barrier, before threads can cross it
    - Semaphores - Mutual exclusion (mutex with a counter), allowing for multiple simultaneous accesses

- There is also the possibility to do busy waiting
  - The thread is spinning in a test (`while (flag!=my_turn);`), no useful work being done until condition false
    - This is a waste of CPU resources, and may cause problems with compiler optimizations (use volatile)
  - It is useful in limited situations where the wait is just a few cycles, being more efficient in this case
    - Or inside the kernel
    - But care must be taken that cache issues do not impair the performance gain

Going further

- Memory hierarchy is not flat
  - Levels of cache memories are used for performance reasons (small fast memory buffers)
- However, cache memory may case performance penalties when concurrent threads are working in shared data
  - A write-miss occurs when a core tries to update a variable that’s not in cache, and it has to access main memory
  - Caches do not work with individual addresses, but with cache lines, coherence is guaranteed: if a value is updated by a core in a cache line, the line is invalidated, and other cores need to reload
  - False sharing may occur if two threads are updating different values which are in the same cache line
  - This is highly dependent of the memory architecture of the processor
- Solutions
  - Pad the data, to guarantee threads do not work in the same cache line (implies wasting memory)
    - Highly architecture dependent, requires fine tuning
  - Copy to local thread data, updating only at the end
    - Copying data in and out may nevertheless also have a penalty
    - And updates are not “immediately” visible



Going further

- Thread safe function calls
  - A function is thread-safe if it can be simultaneously executed by multiple threads providing the correct behaviour
  - Some C library functions are not thread safe, as they use static objects
    - The random number generator random in stdlib.h.
    - The time conversion function localtime in time.h.
    - The string tokenizer strok in string.h
- Previous example of get_ticket

```
int ticket_counter;

int get_ticket(){
  int ticket = ticket_counter;
  printf("Issue ticket number %d\n", ticket);
  ticket_counter++;
  return ticket;
 }
```

```
void* ticket_func(void* data) {
  int my_ticket = get_ticket();
  printf("My ticket is %d\n", my_ticket);
  return NULL;
}
int main() {
  pthread_t thread_id[N_THREADS];
  ticket_counter = 0;
 for (int i = 0 ; i<N_THREADS; i++)
    pthread_create(&thread_id[i], NULL, ticket_func, NULL);
 for (int i = 0 ; i<N_THREADS; i++){
    pthread_join(thread_id[i], NULL);
  }
```

Going further

- All these mechanisms are low level
  - Threads are function handlers which are basically callbacks
  - The programmer is responsible for handling exclusion, and conditional variables explicitly
  - Focused on how to synchronize, instead of focusing on what to synchronize
- Higher-level approaches exist which are much less error prone
  - Threads as language level entities
    - Java thread/runnable class
    - Ada task types and objects
  - The monitor concept provides a construct which integrates the data and the operations in the data, with the mutual exclusion and conditional waiting mechanisms
    - Ada Protected Objects
    - Java Synchronized Objects

Going further

- In Ada, tasks are first level language entities
  - A programmer can declare a task type or variables
  - Task logic and structure is clearly visible in the program structure

```
with Ada.Text_IO; use Ada.Text_IO;
procedure tasking is

   task T1;
   task body T1 is
   begin
      Put_Line("Hello from T1");
   end T1;

   task T2;
   task body T2 is
   begin
      Put_Line("Hello from T2");
   end T2;

begin
   Put_Line("Hello from main");
end tasking;
```

Going further

```
task type Find_Max_Task is
   entry Start(Index: in Positive);
   entry Finish (Result: out Integer);
end Find_Max_Task;

task body Find_Max_Task is
   Range_Start, Range_Finish: Positive;
   Local_Max: Integer;
begin
   accept Start(Index: Positive) do
      Range_Start := (Index - 1) * Size/Number_of_Tasks + 1;
      Range_Finish := Index * Size/Number_of_Tasks;
   end Start;

   Local_Max :=  Vector(Range_Start);
   for I in 1 .. Size loop
      if Vector(I) > Local_Max then
         Local_Max := Vector(I);
      end if;
   end loop;

   accept Finish(Result: out Integer) do
      Result := Local_Max;
   end Finish;
end Find_Max_Task;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Max is
   Size: constant Positive := 10_000;
   Number_of_Tasks: constant Positive := 10;

   Vector: array (Positive range 1 .. Size) of Integer;

   Tasks: array (1 .. Number_Of_Tasks) of Find_Max_Task;

   Max: Integer;
   Local_Max: Integer;
begin
   Fill(Vector);

   for I in 1 .. Number_Of_Tasks loop
      Tasks(I).Start(I);
   end loop;

   Max := Vector(1);
   for I in 1 .. Number_Of_Tasks loop
      Tasks(I).Finish (Local_Max);
      if Local_Max > Max then
         Max := Local_Max;
      end if;
   end loop;
   Put("Max = "); Put(Max); New_Line;
end Max;
```

Going further

- Ada Protected Objects are one of the most powerful implementations of the monitor concept
  - Mutual monitor calls in Java risk deadlock (the “nested monitor” problem). In Ada, with the Ceiling_Locking policy supported, mutual calls across protected objects will not deadlock.
  - In Ada, protected entries combine a condition test with object locking in a way that avoids race conditions. In Java, the programmer must explicitly code the condition wait/notification logic, a more error-prone approach.

```
protected type Bounded_Buffer is
    entry Get (X : out Item);
    entry Put (X : in Item);
  private
    Get_Index : Buffer_Index := 1;
    Put_Index : Buffer_Index := 1;
    Count     : Buffer_Count := 0;
    Data      : Buffer_Array;
  end Bounded_Buffer;
```

```
protected body Bounded_Buffer is
    entry Get (X : out Item) when Count > 0 is
    begin
      X := Data(Get_Index);
      Get_Index := (Get_Index mod Maximum_Buffer_Size) + 1;
      Count := Count - 1;
    end Get;
    entry Put (X : in Item) when Count < Maximum_Buffer_Size is
    begin
      Data(Put_Index) := X;
      Put_Index  := (Put_Index mod Maximum_Buffer_Size) + 1;
      Count := Count + 1;
    end Put;
  end Bounded_Buffer;
```

Advanced reading

- David R. Butenhof, “Programming with POSIX Threads”, Addison-Wesley, 1997, ISBN 978-0201633924
- Thomas Rauber, Gudula Runger, “Parallel Programming For Multicore and Cluster Systems”, Springer, 2010, DOI 10.1007/978-3-642-04818-0
- Alan Burns, Andy Wellings, “Real-Time Systems and Programming Languages - Ada 2005, Real-Time Java and C/Real-Time POSIX”, 4th edition, Pearson, 2009, ISBN 978-0321417459

Credits

- Version 1.0, Luis Miguel Pinho, with inputs from:
  - L.L Ferreira, M.J. Viamonte, Threads, Sistemas de Computadores, 2008
  - L. Nogueira, N. Pereira, L.M. Pinho, Threads, Sistemas de Computadores, 2015
  - P. Pacheco, “An Introduction to Parallel Programming”, 1st Edition, Morgan Kaufmann 2011, ISBN 9780080921440, companion materials, https://booksite.elsevier.com/9780123742605/