Parallel Architectures and Programming (PAP)

POSIX Threads (Pthreads) Programming[1]

Eduard Ayguadé
(eduard@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2019/20-Spring

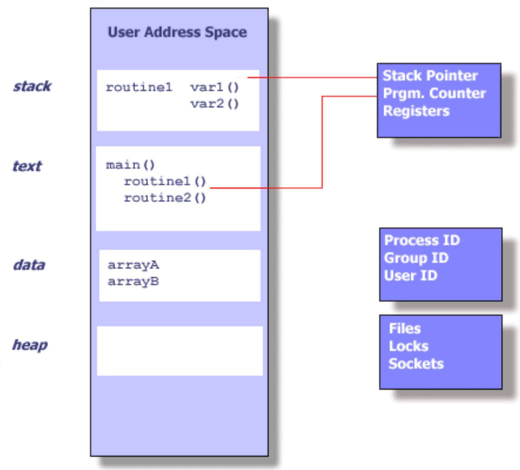
[1] Examples in `/scratch/nas/1/pap0/sessions/pthreads.tar.gz`.

Outline

Processes

- A process is a program in execution, created by OS with a fair amount of overhead
- Each process contains:
  - Executable program
  - Data, heap and stack
  - Execution context, i.e. all information the OS needs to manage the process: PID, UID, GID, registers (including program counter and stack pointer), environment, working directory, file descriptors, shared libraries, inter-process communication mechanisms, ...

Processes


User Address Space
stack
routine1 var1()
var2()
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets

fork system call

A (parent) process can create child processes

- Child processes have a separated but exact copy of the parent's address space
- After a new child process is created, both processes will execute the next instruction following the `fork()` system call

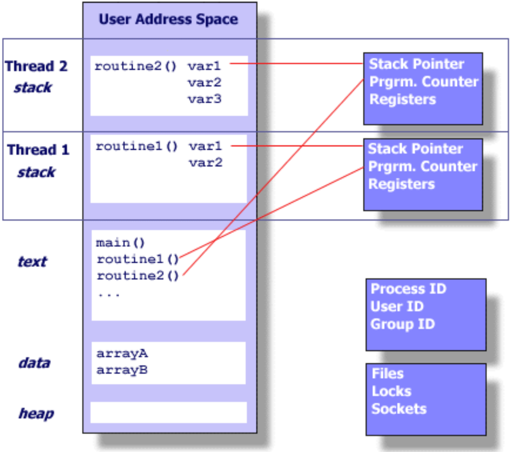
```
void  main(void) {
     pid_t pid = fork();
     if (pid < 0)
          printf("Error creating child process\n);
     else if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}
```

Threads

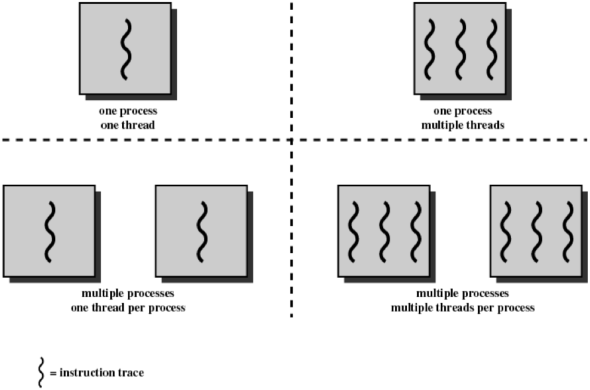
- A thread is an independent stream of instructions inside a process that duplicates only the essential resources for OS to schedule him to run
  - Program counter
  - Stack pointer
  - Other registers

  with some additional information (scheduling properties and thread–specific data)
- Multiple threads can live within the same process, sharing process resources and address space
- Lightweight, i.e. the cost of creating and managing a thread is much less than a process

Threads


User Address Space
Thread 2
stack
routine2() var1
var2
var3
Thread 1
stack
routine1() var1
var2
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets

Processes and threads



What do we need to implement the OpenMP runtime library?

Shared–memory abstraction is key in the definition of OpenMP

- **Processes** do not provide it, **threads** do, in addition there is an issue with the overheads of creating processes vs. threads
- Pthreads – standard thread API (IEEE Std 1003.1)[2]
  - Thread management: create, terminate, join
  - Thread synchronization: barriers, mutexes, condition variables and semaphores
  - Thread-specific data

[2] POSIX threads tutorial at https://computing.llnl.gov/tutorials/pthreads

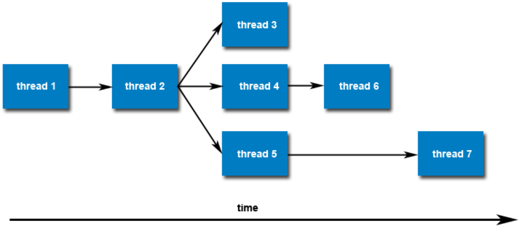
Outline

Thread creation

- Creates a new thread and makes it executable

```
int pthread_create ( pthread_t * thread,
                     pthread_attr_t * attr,
                     void * (*start_routine)(void *),
                     void * arg );
```

- Once created, threads are peers (there is no implied hierarchy) and may create other threads



Thread (and process) termination

- Thread termination

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

  - We will see later how the return value can be used

- Process termination
  - `exit()` called by any thread
  - main returns

Example: Hello world! (not correct)

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

void *PrintHello(void * arg) {
   printf("Hello World!\n");
   pthread_exit(NULL);
}

int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, &PrintHello, NULL);

   return(0); // Not correct because main thread does not wait
              // for the termination of created thread
}
```

Simply creates a thread and terminates.

Thread joining

- Suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated

```
int pthread_join( pthread_t thread,
                  void **value_ptr );
```

  - The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call using `pthread_exit()`
- There is an implicit join when main finishes with `pthread_exit()`, keeping created threads alive

Example: Hello world!

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

void *PrintHello(void * arg) {
   printf("Hello World!\n");
   pthread_exit(NULL);
}

int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, &PrintHello, NULL);

   pthread_join(thread, NULL);
   printf("Thread has terminated\n");
   return(0);
}
```

Example: multiple Hello world!

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define numThreads 5

void *PrintHello(void * arg) {
   printf("Hello World!\n");
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   pthread_t threads[numThreads];
   for (int t=0; t<numThreads; t++)
       pthread_create(&threads[t], NULL, &PrintHello, NULL);

   printf("Done creating threads\n");

   for (int t=0; t<numThreads; t++)
       pthread_join(threads[t], NULL);

   printf("All threads finished\n");
}
```

Example: multiple Hello world! (with return result)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define numThreads 5

void *PrintHello(void * arg) {
   long tid = pthread_self();        // returns the ID of the calling thread
   printf("Hello World!\n");
   pthread_exit((void *) tid);
}

int main(int argc, char *argv[]) {
   pthread_t threads[numThreads];
   for (int t=0; t<numThreads; t++)
       pthread_create(&threads[t], NULL, &PrintHello, NULL);

   void *status;
   for (int t=0; t<numThreads; t++) {
       pthread_join(thread[t], &status);
       printf("Completed join with thread %ld\n", (long)status);
   }
}
```

Passing data to threads ... wrongly

- Consider the following thread creation code

```
void *PrintHello(void *threadId) {
   int * tid = (int *) threadId;
   printf("Hello World! It's me, thread #%d!\n", *tid);
}

int main(int argc, char *argv[]) {
   ...
   for (int t=0; t<numThreads; t++)
       pthread_create(&threads[t], NULL, &PrintHello, (void *)&t);
   ...
}
```

- Why is this wrong?

Passing data to threads ...

- How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling?
  - Make sure that all passed data is thread safe, i.e. that it can not be changed by other threads
- Example 1: The calling thread uses a unique memory position for each thread, ensuring that each thread's argument remains intact throughout the program.
- Example 2: The calling thread uses a unique structure for each thread in order to pass multiple arguments

Passing data to threads ... example 1

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define numThreads 5

void *PrintHello(void *threadId) {
   int * tid = (int *) threadId;
   printf("Hello World! It's me, thread #%d!\n", *tid);
}

int main(int argc, char *argv[]) {
   pthread_t threads[numThreads];
   int argument[numThreads];

   for (int t=0; t<numThreads; t++) {
       argument[t] = t;
       pthread_create(&threads[t], NULL, &PrintHello, (void *) &argument[t]);
   }

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

Passing data to threads ... example 2

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define numThreads 5

typedef struct {
   int  thread_id;
   char *message;
} thread_data;

char *messages[2];

... // code for thread

int main(int argc, char *argv[]) {
   pthread_t threads[numThreads];
   thread_data thread_data_array[numThreads];

   for (int t=0; t<numThreads; ++t ) {
       thread_data_array[t].thread_id = t;
       thread_data_array[t].message = messages[t%2];
       pthread_create (&threads[t], NULL, &PrintHello,  (void *) (thread_data_array + t));
       }

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

Passing data to threads ... example 2 (cont.)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define numThreads 5

typedef struct {
   int        thread_id;
   char *message;
} thread_data;

char *messages[2];
messages[0] = "Hello World!, it's me";
messages[1] = "Bonjour, le monde!, c'est moi";

void *PrintHello(void *threadArgs) {
   thread_data *my_data = (thread_data *) threadArgs;

   int tid = my_data->thread_id;
   char * msg = my_data->message;
   printf("%s thread #%d\n", msg, tid);
}

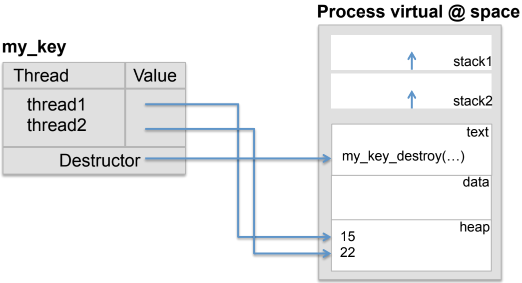
... // code for main
```

Outline

Thread-specific data

- Pthreads allows the definition of global variables (keys) that have different values for different threads. By referencing the same name, different threads may bind different values to the same key
- ```
  pthread_key_t
  ```
  is the type for defining keys in the thread-local storage
- ```
  int pthread_key_create(pthread_key_t *key,
                         void (*destructor)(void*));
  ```
  creates a thread-specific data key visible to all threads in the process. Done only once!

Thread-specific data



- The user–programmed destructor function is called by the system whenever a thread dies, to clean up the per-thread data allocated to it

Thread-specific data

- ```
  void *pthread_getspecific(pthread_key_t key);
  ```
  returns the value currently bound to the specified key on behalf of the calling thread

- ```
  int pthread_setspecific(pthread_key_t key,
                          void *value);
  ```
  associates a thread-specific value with a key

- ```
  int pthread_key_delete(pthread_key_t key);
  ```
  deletes a thread-specific data key previously created with `pthread_key_create`
  ```

Hello World! using thread–specific data (1)

```
#define numThreads 5
pthread_key_t my_key;

void print_message () {
  int * myID = (int *) pthread_getspecific (my_key);
  printf("Hello World! It's me, thread #%d!\n", *myID);
}

void *PrintHello(void * threadId) {
  pthread_setspecific (my_key, threadId);
  // later in the thread code ...
  print_message ();
  pthread_exit(NULL);
}

int main () {
  pthread_t threads[numThreads];
  int argument[numThreads];

  pthread_key_create (&my_key, NULL);

  for (int t = 0; t < numThreads; t++) {
    argument[t] = t;
    pthread_create(&threads[t], NULL, &PrintHello, (void *) &argument[t]);
  }
  ...
  pthread_key_delete(my_key);
}
```

Hello World! using thread–specific data (2)

```
#define numThreads 5

typedef struct {
  int thread_id;
  char message[32];
} perthread_data;

pthread_key_t my_key;

void print_message () {
  perthread_data * mine = pthread_getspecific (my_key);
  printf("%s %d!\n", mine->message, mine->thread_id);
}

void *PrintHello(void * threadId) {
  int id = (int) threadId;
  perthread_data * mine = (perthread_data *) malloc(sizeof(perthread_data));
  mine->thread_id = id;
  if (id%2) strcpy(mine->message, "Hello World!, it's me");
  else strcpy(mine->message, "Bonjour le Monde!, c'est moi");
  pthread_setspecific (my_key, (void *) mine);

  // later in the thread code ...

  print_message ();
  pthread_exit(NULL);
}
```

Hello World! using thread–specific data (2)

```
#define numThreads 5

pthread_key_t my_key;

void my_key_destroy (void * arg) {
  perthread_data * mine = (perthread_data *) arg;
  printf("Destroying key for thread %d\n", mine->thread_id);
  free (mine); // deallocate data
}

int main () {
  pthread_t threads[numThreads];

  pthread_key_create (&my_key, my_key_destroy); // per-thread attribute key

  for (int t = 0; t < numThreads; t++)
    pthread_create(&threads[t], NULL, &PrintHello, (void *) t);

  for (int t = 0; t < numThreads; t++)
    pthread_join(threads[t], NULL);

  pthread_key_delete(my_key);
}
```

Outline

Synchronization objects

- Mutexes
- Barriers[3]
- Built-in functions in gcc for atomic memory accesses
- Condition variables
- Semaphores[3]

[3] Part of POSIX real-time extensions.

Mutex variables

- A 'mutex' is a synchronization object that is used to protect a region of code that only one thread at a time can execute (mutual exclusion[4])
  - A thread 'locks' the mutex when it enters, and 'unlocks' the mutex when it leaves
  - All other threads attempting to gain the lock must wait until the current 'owner' unlocks the mutex – at that point one of them will gain access
- Routines (see arguments in the examples):

```
pthread_mutex_t           // Declare a mutex
pthread_mutex_init()      // Initialize a mutex
pthread_mutex_destroy()   // Destroy a mutex
pthread_mutex_lock()      // Lock a mutex
pthread_mutex_unlock()    // Unlock a mutex
```

[4] Pthreads includes "read-write" locks, which allow multiple readers to acquire a lock if a writer doesn't hold it.

Creating and destroying mutexes

- Two ways to initialize mutexes: statically and dynamically
- If you want to include attributes, you need to use the dynamic version

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;
pthread_mutex_init(&m2, NULL);
...
pthread_mutex_destroy(&m1);
pthread_mutex_destroy(&m2);
```

Using mutexes

```
#include <pthread.h>
#define numThreads 5
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0;

void * run (void * arg) {
    for (int i = 0; i < 100; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
}

int main () {
    pthread_t threads[numThreads];
    for (int t=0; t<numThreads; ++t )
        pthread_create (&threads[t], NULL, &run,  NULL);

    for (int t=0; t<numThreads; t++)
        pthread_join(threads[t], NULL);

    pthread_mutex_destroy (&mutex);
    printf("Counter = %i\n", counter);
}
```

Using mutexes

Also you can use pthread_mutex_trylock, if needed.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

This function attempts to acquire ownership of the mutex specified without blocking the calling thread. If mutex is currently locked by another thread, the call to pthread_mutex_trylock returns an error of EBUSY. A failure of EDEADLK indicates that the mutex is already held by the calling thread[5].

[5] errno.h has to be included n order to understand these error codes

Producer-consumer using mutexes (1)

```
// Mutexes to protect the buffer
pthread_mutex_t producer_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t consumer_lock = PTHREAD_MUTEX_INITIALIZER;
int buffer; // buffer size 1

#define MAX 10 // maximum number to produce

int main() {
    int rc;
    pthread_t producer_thread;
    pthread_t consumer_thread;

    pthread_mutex_lock(&consumer_lock);
    if ((rc= pthread_create(&consumer_thread, NULL, &consumer, NULL)))
        printf("Error creating the consumer thread\n");
    if ((rc= pthread_create(&producer_thread, NULL, &producer, NULL)))
        printf("Error creating the producer thread\n");

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    printf("Done..\n");
}
```

Producer-consumer using mutexes (2)

```
void *producer(void *arg) {
    int number = 0;
    while (1) {
        number++;

        pthread_mutex_lock(&producer_lock);
        printf("Producer : %d\n", number);
        buffer = number;
        pthread_mutex_unlock(&consumer_lock);

        // Stop if MAX has been produced
        if (number == MAX) {
            printf("Producer done.. !!\n");
            break;
        }
    }
}
```

Producer-consumer using mutexes (3)

```
void *consumer(void *arg) {
    int number;
    while (1) {
        pthread_mutex_lock(&consumer_lock);
        // consume (print) the number in the buffer
        number = buffer;
        printf("Consumer : %d\n", number);
        pthread_mutex_unlock(&producer_lock);

        // If the MAX number was the last consumed number, the consumer should stop
        if (number == MAX) {
            printf("Consumer done.. !!\n");
            break;
        }
    }
}
```

Barriers

- A 'barrier' is a synchronization object that forces several cooperating threads to wait at a specific point until all have finished before any one thread can continue
- Routines (see arguments in the example):

```
pthread_barrier_t          // Declare a barrier

pthread_barrier_init( )    // Initialize a barrier
pthread_barrier_destroy( ) // Destroy a barrier

pthread_barrier_wait( )    // Synchronize threads
                           // at the barrier
```

Example: Hello world! using a barrier

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define numThreads 5
pthread_barrier_t barrier;

void *PrintHello(void * arg) {
   printf("Hello World!, it's me thread %d\n", (int) arg);
   pthread_barrier_wait(&barrier);
}

int main() {
   pthread_t threads[numThreads];
   pthread_barrier_init(&barrier, NULL, numThreads+1); // number of threads to wait;
              // NULL for default barrier attributes: per-process private barrier

   for (int t=0; t<numThreads; t++)
       pthread_create(&threads[t], NULL, &PrintHello, (void *) i );
   printf("Done creating threads\n");

   pthread_barrier_wait(&barrier); // the function returns PTHREAD_BARRIER_SERIAL_THREAD
              // for one arbitrary thread synchronized at the barrier; 0 for the rest
   printf("Done executing threads\n");

   pthread_barrier_destroy(&barrier);
}
```

Outline

Atomic operations

- These built–ins perform the operation specified by XXX (`add`, `sub`, `or`, `and`, `xor`, `nand`), and return the value that had previously been in memory pointed by `*ptr`

```
type __sync_fetch_and_XXX (type *ptr, type value)
```

i.e. `tmp = *ptr; *ptr op= value; return tmp;`

- These built–ins perform the operation specified by XXX, and return the new `value`

```
type __sync_XXX_and_fetch (type *ptr, type value)
```

i.e. `*ptr op= value; return *ptr;`

Atomic operations (cont.)

- These built–ins perform an atomic compare and swap. That is, if the current value of `*ptr` is `oldval`, then write `newval` into `*ptr`

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval)
type __sync_val_compare_and_swap (type *ptr, type oldval, type newval)
```

  The "bool" version returns true if the comparison is successful and `newval` was written. The "val" version returns the contents of `*ptr` before the operation

Note: Type in each of the expressions can be one of the following: int, unsigned int, long, unsigned long, long long, unsigned long long.

Memory barrier

- This built–in issues a full memory barrier

```
void __sync_synchronize ()
```

  i.e. enforces an ordering constraint on memory operations issued before and after the memory barrier instruction: operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier

Implicit memory barriers in Pthreads

- Thread creation: any variable value set by a thread prior to `pthread_create` can be seen within the newly created thread. This no longer holds if set after `pthread_create`, even if this operation occurs before the thread starts.
- Thread join: Any variable value set by a thread prior terminating can be seen within the joining thread after successful completion of `pthread_join`.
- Mutex unlock: any variable value set by a thread prior to unlocking a mutex can be seen by any thread that later successfully locks the same mutex. This may no longer hold if another mutex or if no locking at all is used, or if the variable value is set after `pthread_unlock`.

Example using GCC built–in functions

```
#define INC_TO 1000000 // one million ...

int global_int = 0;
int finished = 0;
int towait = 0;

void *thread_routine( void *arg ) {
    int id = (int)arg;

    for (long i = 0; i < INC_TO; i++)
        global_int++;

    if (id == 0) { finished=1; towait = 1; }
    if (id != 0) while(finished==0);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[numThreads];
    for (int i = 0; i < numThreads; i++)
        pthread_create(&threads[i], NULL, thread_routine, (void *) i);

    while (towait == 0);
    for (int i = 0; i < numThreads; i++)
        pthread_join(threads[i], NULL);
    //while (towait == 0));
}
```

Example using GCC built–in functions

```
#define INC_TO 1000000 // one million ...

int global_int = 0;
int finished = 0;

void *thread_routine( void *arg ) {
    int id = (int)arg;

    for (long i = 0; i < INC_TO; i++)
        __sync_fetch_and_add( &global_int, 1 ); // global_int++ has data race

    if (id == 0) { finished=1; towait = 1; __sync_synchronize(); } // fence for consistency
    if (id != 0) while(finished==0) __sync_synchronize (); // fence for consistency

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[numThreads];
    for (int i = 0; i < numThreads; i++)
        pthread_create(&threads[i], NULL, thread_routine, (void *) i);

    while (towait == 0) __sync_synchronize (); //  memory fence for memory consistency
    for (int i = 0; i < numThreads; i++)
        pthread_join(threads[i], NULL);
    //while (towait == 0)); // it would be OK, pthread_join implies memory barrier
}
```

Outline

- This is your turn to explain ...
- ... Pthread condition variables and POSIX semaphores,
- their API and some useful examples to understand how useful they are

Outline

CPU affinity

- Setting CPU affinity of a thread

```
void *thread_func(void *param) {
    cpu_set_t cpuset;
    /* bind process to processor 0 */
    CPU_ZERO(&cpuset); // clears set, so that it contains no CPUs
    CPU_SET(0, &cpuset); // Add CPU cpu to set
    pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);

    /* waste some time so the work is visible with "top" (press 1) or "htop" */
    printf("result: %f\n", waste_time(5000));

    /* bind process to processor 3 */
    CPU_CLR(0, &cpuset); // Remove CPU from set
    CPU_SET(3, &cpuset);
    pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);

    /* waste some more time to see the processor switch */
    printf("result: %f\n", waste_time(5000));

    pthread_exit(NULL);
}
```

- cpu_set_t is a bit mask, one bit per core (processor)

CPU affinity

- Other macros: `CPU_ISSET` to test to see if a CPU is a member of set, `CPU_COUNT` to count the number of CPUs in set, logical operations on sets (`CPU_AND`, `CPU_OR` and `CPU_XOR`) and comparison of two sets (`CPU_EQUAL`).
- The `pthread_getaffinity_np(thread, cpusetsize, *cpuset)` function returns the CPU affinity set of the thread in the buffer pointed to by cpuset.

Parallel Architectures and Programming (PAP)

POSIX Threads (Pthreads) Programming[6]

Eduard Ayguadé
(eduard@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2019/20-Spring

[6] Examples in /scratch/nas/1/pap0/sessions/pthreads.tar.gz.