Programming Shared Memory with POSIX Threads

Background: Processes vs Threads

POSIX threads: Intro

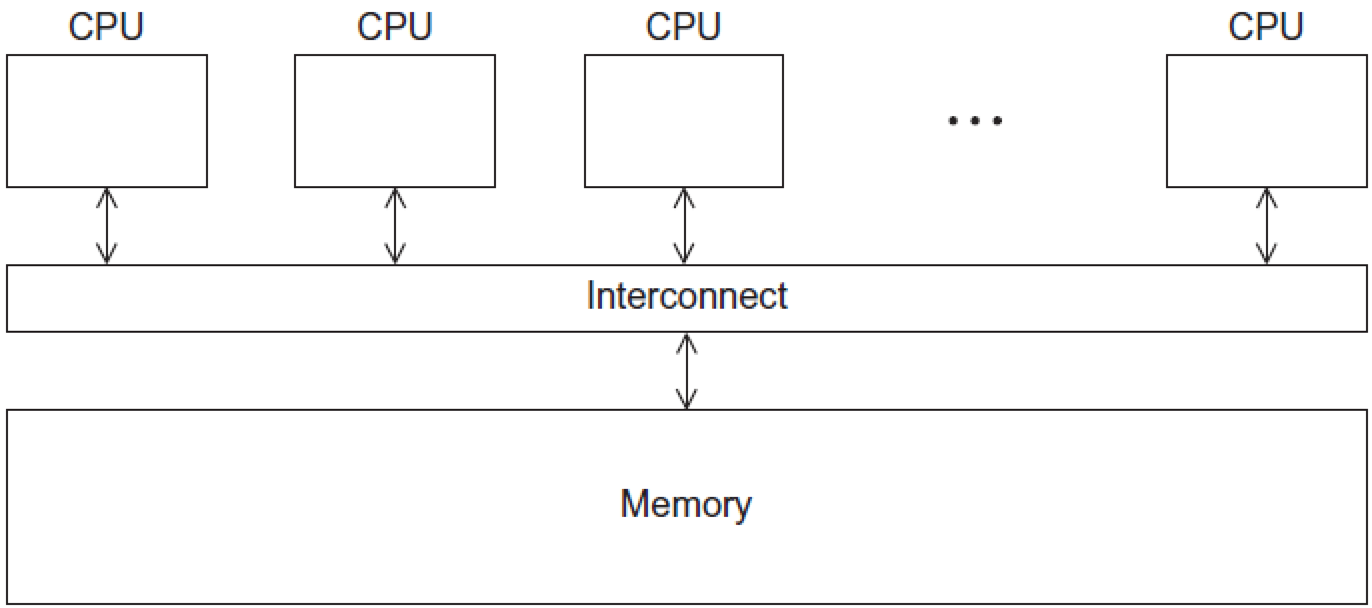
Our first parallel program

Bibliography

- [Pacheco]: Peter Pacheco, Matthew Malensek, Introduction to Parallel Programming, 2nd Edition, Morgan Kaufmann Publisher, March 2020, Chapter 2.1.2, Chapter 4.1, Chapter 4.2, Chapter 4.3
- [Illinois]: System Programming @ University of Illinois, Chapter 6 https://cs341.cs.illinois.edu/coursebook/index.html

A Shared Memory System

Shared-memory programs: At a minimum, it supposes that certain variables are available to multiple “processes”



From [Pacheco]

Processes

- A process is an instance of a running (or suspended) program
- It is managed by the Operating System
- Each process has its private virtual address space (each program thinks that it has exclusive use of main memory)
- Each process has a logical control flow (each program thinks that it has exclusive use of the CPU)

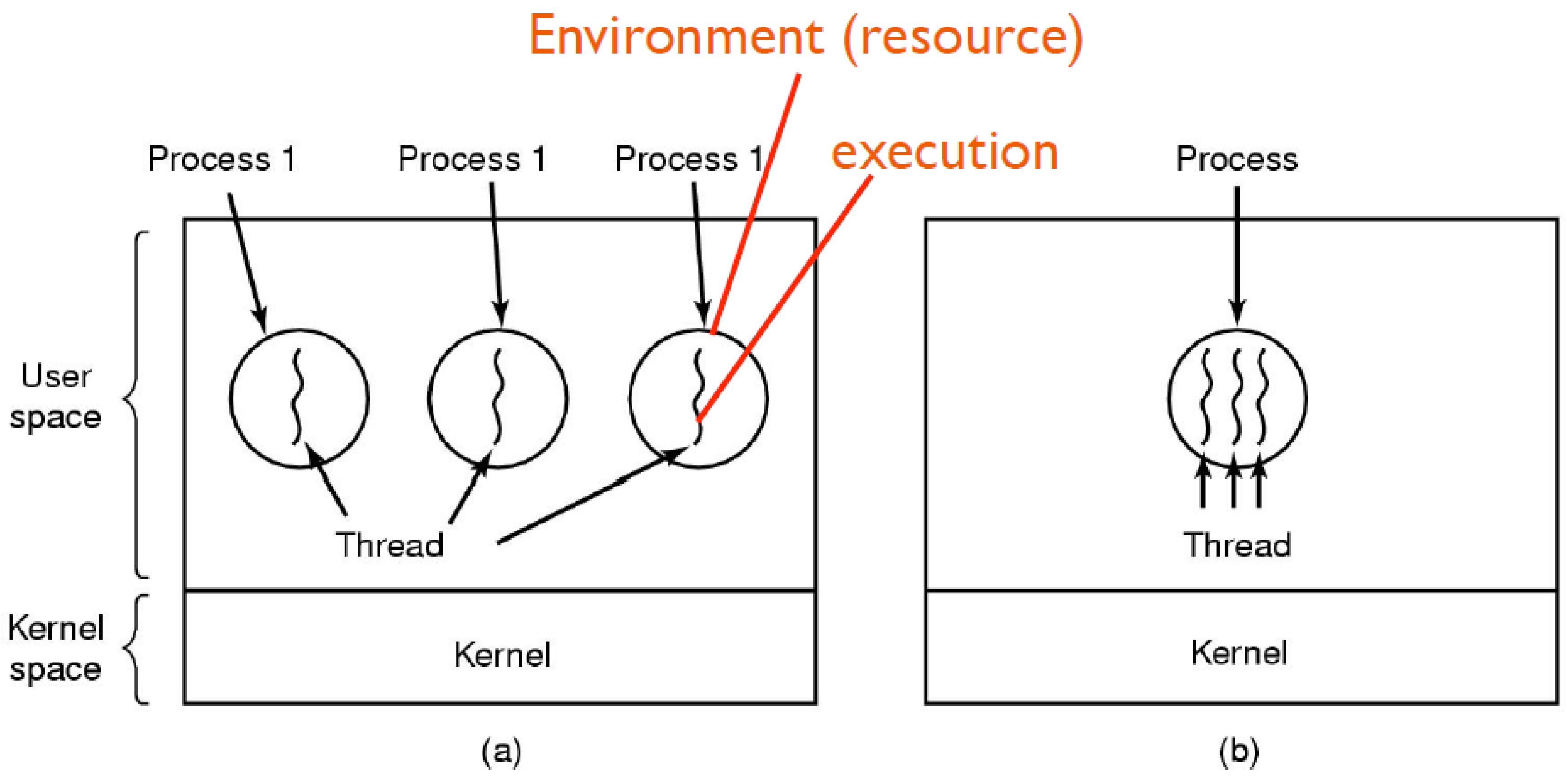
Components of a Process

- A Process consists of:
  - The executable machine language program
  - A block of memory for the stack – keeps track of active function calls
  - A block of memory for the heap – used for memory explicitly allocated by the user program
  - Descriptors of resources that the system has allocated for the process—for example, file descriptors (including stdout, stdin, and stderr)
  - Security information—for example, information about which hardware and software resources the process can access
  - Information about the state of the process, such as whether the process is ready to run or is waiting on a resource, the content of the registers, including the program counter, etc
- In most systems, by default, a process's resources are private: another process can't directly access them unless the operating system intervenes!

Processes and Threads

- Threads are analogous to a “light-weight” process.
  - Creating a process (fork) and executing process context switches is “expensive”
  - Threads as “lightweight” processes: little memory, fast startup
- In a shared memory program a single process may have multiple threads of control.
- **Threads belonging to the same process can share most of the process’s resources:**
  - **Can share**: **memory (heap), file descriptors, sockets**
  - **Must have of their own**: own **program counter** and own **call stack**, so that the threads can execute independently of each other.

Processes and Threads



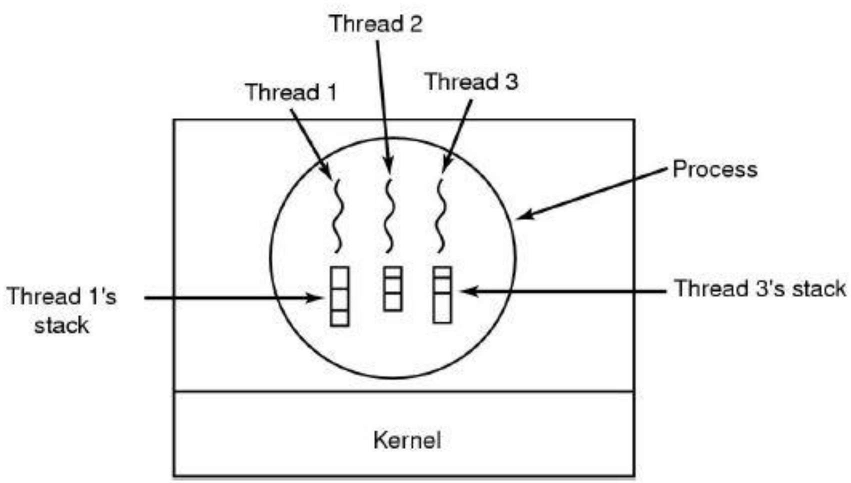
Three processes each with one thread

One process with three threads

From [Illinois CS241]

Thread Specific Resources

- Each thread has its own
  - Thread ID (an integer)
  - Stack, Registers, Program Counter
- Threads in one process can communicate via shared memory (which is the heap block of the containing process)
  - Access to shared memory must be done carefully -> learn good practices of shared memory programming



From [Illinois CS241]

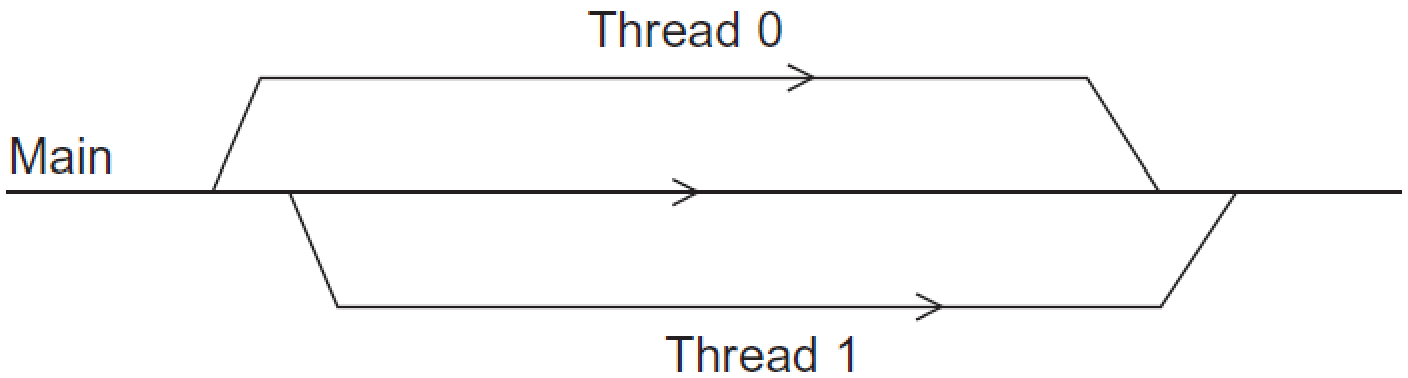
POSIX Threads

- Early on:
  - Each OS had its own thread library/API
  - Difficult to write multithreaded programs
    - Learn a new API with each new OS
    - Modify code with each port to a new OS
- POSIX (IEEE 1003.1c-1995) provided a standard known as pthreads
- A standard for Unix-like operating systems.
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.
- There are implementations of pthreads API for Windows (to ensure portability of threaded applications between Windows and Unix)
- Other Thread implementations:
  - Java Threads (managed by the JVM)
  - Windows Threads

POSIX Threads model

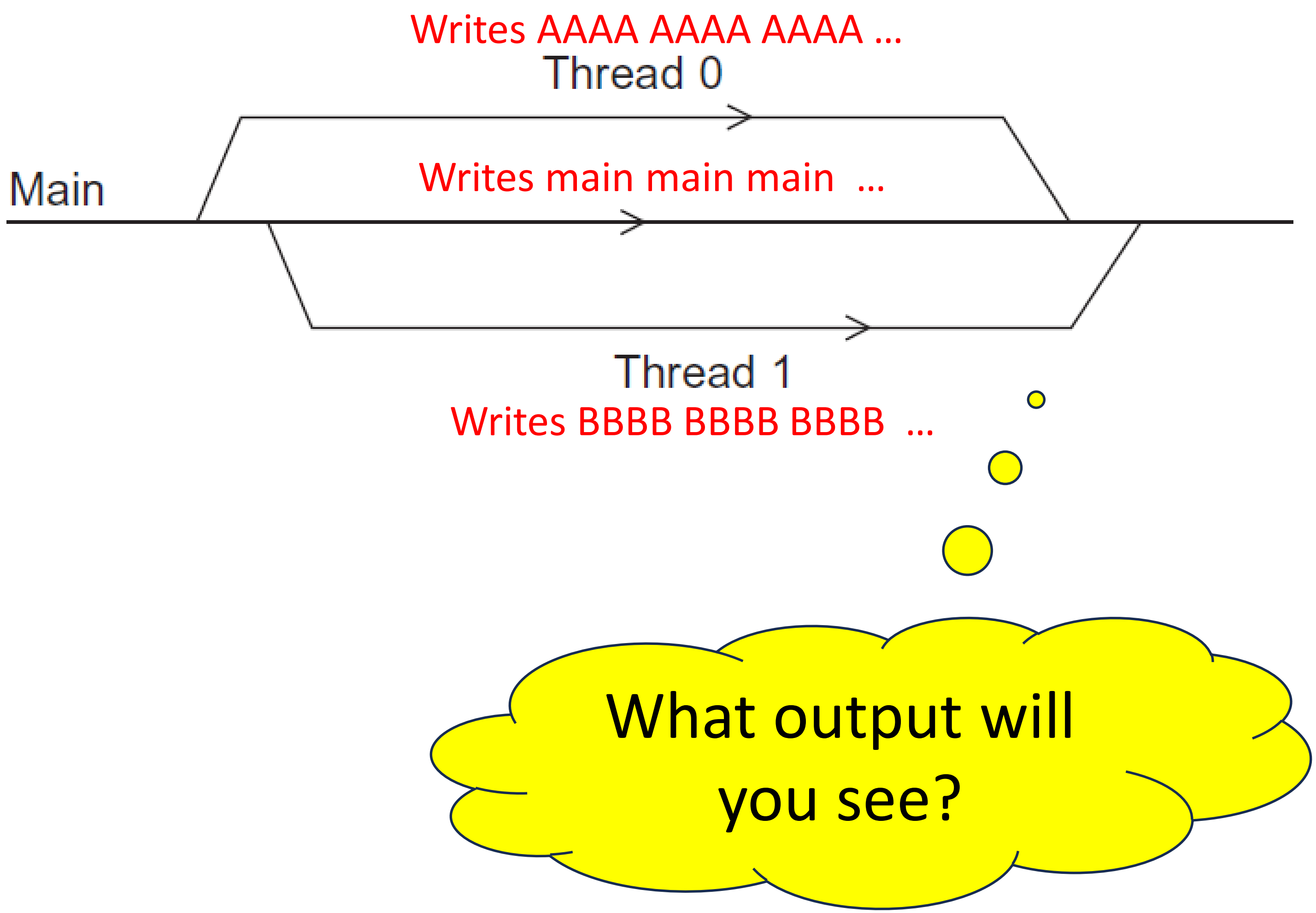
- Pthreads lets the user program decide to explicitly create a new thread at a random moment calling function pthread_create
- The new thread will execute a function (the “thread function”) which is explicitly given as an *argument* to pthread_create
- A call to pthread_create determins a *branch* or *fork* off the parent(main) thread. Multiple calls to pthread_create will result in multiple branches or forks. Operating system will perform context switches between threads (similar to process context switches but more lightweight)
- The parent thread can wait for the completion of a child thread by calling pthread_join
- A thread terminates when:
  - The thread function returns
  - The function pthread_exit is called
- Spawning pthreads follows a fork-join model

Running the Threads



Example: Main thread forks and joins two threads.

First Example: HelloAB



First Example: HelloAB (1)

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

/* Thread function A – does not use argument */
void *HelloA(void *dummy)
{
   for (int i = 0; i < 200; i++)
      printf("AAAA");
   return NULL;
}

/* Thread function B – does not use argument */
void *HelloB(void *dummy)
{
   for (int i = 0; i < 200; i++)
      printf("BBBB");
   return NULL;
}
```

First Example: HelloAB (2)

```
int main(int argc, char *argv[])
{

   pthread_t thread_handleA, thread_handleB;

   pthread_create(&thread_handleA, NULL, HelloA, NULL);

   pthread_create(&thread_handleB, NULL, HelloB, NULL);

   for (int i = 0; i < 200; i++)
      printf("main");

   pthread_join(thread_handleA, NULL);

   pthread_join(thread_handleB, NULL);

   return 0;
}
```

Creating a thread

```
int pthread_create (
    pthread_t*  thread_p ,
    const pthread_attr_t*  attr_p ,
    void*  (*start_routine ) ( void ) ,
    void*  arg_p  ) ;
```

Parameters:

- **thread_p**: (out parameter)
  - Unique thread identifier **returned** from call
  - Must be allocated before calling pthread_create!!!
- **Attr_p**: (in parameter)
  - Attributes structure used to define new thread
  - Use **NULL** for default values
- **start_routine**:
  - Main routine for created thread. The thread will run this function
  - Takes a pointer **(void*)**, returns a pointer **(void*)**
- **arg_p**:
  - Pointer to the argument passed to child thread main routine

Return: zero if success, error number otherwise

Function started by pthread_create

- Acts as the main function of the new thread
- The prototype of this function must be:
  `void*  thread_function ( void*  args_p ) ;`

- void* can be cast to any pointer type in C!

- args_p can point to a list containing one or more values needed by thread_function.
- Similarly, the return value of thread_function can point to a list of one or more values.

Terminating a thread

A thread terminates when:

- The thread function returns
- The function pthread_exit is called

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

- Terminate the calling thread
- Makes the value **retval** available to any successful join with the terminating thread
- Returns: **pthread_exit()** cannot return to its caller!
- Parameters:
- **retval**: Pointer to data returned to joining thread. It must be *a pointer to heap not to the stack!*

Note: If **main()** exits by calling **pthread_exit()** before its threads, the other threads continue to execute. Otherwise, they will be terminated when **main()** finishes.

Waiting for a thread

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

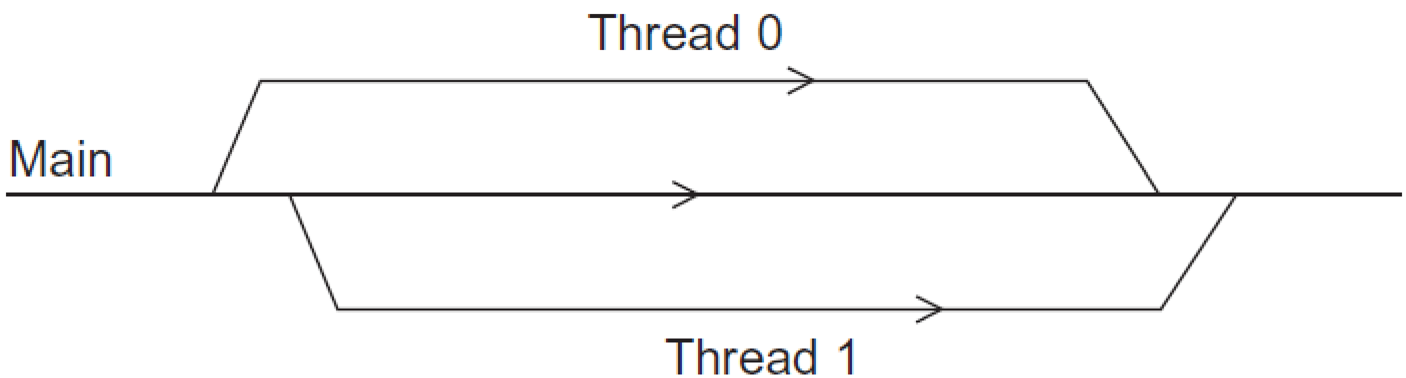
- Suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

- Returns: zero on success, error code on failure
- Parameters
  - **thread**: Target thread identifier
  - **retval**: The pointer passed to **pthread_exit()** by the terminating thread is made available in the location referenced by **retval**

Returned error codes

- All functions return an errorcode>0 if something went wrong.
- Programs should check this and act accordingly (exit, messages,etc)

HelloWorld Example



A global variable (accessible to all threads) **`num_threads`** contains the total number of threads.
The main thread spawns a total of **`num_threads`** threads.
Each thread receives as argument of the thread-function its ***rank***(determined by its position in the order of spawning)
Each thread prints a message: *Hello from thread {rank}out of {num_threads}*
It is a common pattern that a thread receives as argument its rank and acts differently according to its rank !

HelloWorld Example:

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

/* Global variable:  accessible to all threads */
int thread_count;

/* Thread function.
    Receives one argument which is a pointer to its rank.
    All threads get this same function
 */
void *Hello(void *rank)
{
   int my_rank = *(int *)rank;
   printf("Hello from thread %d of %d\n", my_rank, thread_count);
   return NULL;
}
```

HelloWorld Example: (contd.)

```
int main(int argc, char *argv[]) {
   int i;
   pthread_t *thread_handles;
   int *tid;

   thread_count = 10; /* hardcoding value; could read it */

   thread_handles = malloc(thread_count * sizeof(pthread_t));
   tid = malloc(thread_count * sizeof(int));

   for (i = 0; i < thread_count; i++) {
      tid[i] = i;
      pthread_create(&thread_handles[i], NULL, Hello, (void *)&tid[i]);
   }

   printf("Hello from the main thread\n");

   for (i = 0; i < thread_count; i++)
      pthread_join(thread_handles[i], NULL);

   free(thread_handles);
   free(tid);
```

HelloWorld Example v2: *Try* change:

Replace old sequence:

```
for (i = 0; i < thread_count; i++) {
   tid[i] = i;
   pthread_create(&thread_handles[i], NULL, Hello, (void *)&tid[i]);
}
```

With new sequence:

```
for (i = 0; i < thread_count; i++) {
   pthread_create(&thread_handles[i], NULL, Hello, (void *)&i);
}
```

This is NOT working !!!

Because all threads receive as argument a pointer to the same location of the variable i which is continuously changing its content. It is totally unpredictable which value will be there at the moment when each thread gets to extract the value from its argument pointer!

HelloWorld Example v3:

This is OK:

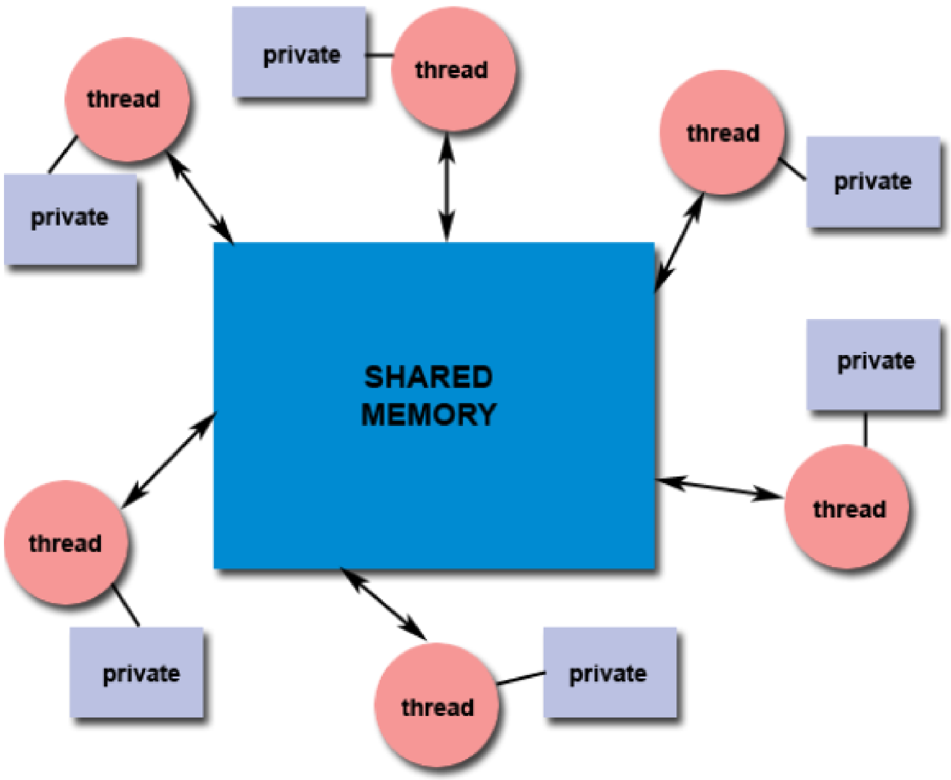
```
 long i;

 // …

 for (i = 0; i < thread_count; i++)
    {
       pthread_create(&thread_handles[i], NULL, Hello, (void *)i);
    }
// …


void *Hello(void *rank)
{
   long my_rank = (long)rank;

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}
```

Source code of Hello examples:

- helloAB.c
- hello1.c
- hello2.c
- hello3.c

Our first parallel program

- Shared memory programming with threads:
  - All threads have access to the same global, shared memory (the global variables of the program)
  - Threads also have their own private data (local variables in thread functions)
  - Programmers are responsible for synchronizing access (protecting) globally shared data.
    - Our first program is embarrassingly parallel and does not need synchronization



Matrix-Vector Multiplication

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

Matrix-Vector Multipl - Serial

```
void Mat_vect_mult(
                   double  A[]  /* in  */,
                   double  x[]  /* in  */,
                   double  y[]  /* out */,
                   int     m    /* in  */,
                   int     n    /* in  */) {
   int i, j;

   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
}
```

Matrix-Vector Multipl - Parallel

- We parallelize by dividing the work among the threads.
- Each element y[i] of the result vector can be computed independently from other elements of the result vector -> they can be computed in parallel
- The maximum possible degree of parallelism is equal to m(the size of the vector y).
- If m is much larger than the number of available cores or processors, it is NOT reasonable to spawn such a big number of threads!
- If T threads can be efficiently supported, then each thread gets to compute m/T elements of the result vector y (assume m is divisible by T)
  - Thread rank (rank in [0…T-1]) will compute elements y[i] where:
  - i>=rank* m/T and i<(rank+1)*m/T
  - Each thread needs all elements of vector x but only its range of rows from matrix A. However, they are used read-only -> we make them all global (shared) variables and need not take any protection measures

Matrix-Vector Multipl - Parallel

```
/* Global variables */
int     thread_count;
int     m, n;
double* A;
double* x;
double* y;

// …

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   // … add code to read m,n,thread_count and allocate A, x, y

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
         Pth_mat_vect, (void*) thread);
```

Matrix-Vector Multipl - Parallel

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i*n+j]*x[j];
   }

   return NULL;
}
```

Source code of Examples:

- Matrix-Vector multiplication: Serial version, Parallel version with POSIX threads, measuring speedup
  speedup_pth_mat_vect.c

Matrix-Vector Multipl - Evaluation

- The parallel program must compute the same correct result as the serial program
- The parallel program should have a shorter runtime TParalel compared with the runtime of the serial version TSerial
- Speedup = TSerial/TParallel

Matrix-Vector Multipl – Evaluation(1)

Measurements performed on a laptop with 8 cores:

For a certain number of parallel threads used, vary the size of the matrix:

Use **2** parallel threads

m=n	**100**	**500**	**1000**	**5000**	**10000**
TSerial	0.000	0.002	0.004	0.073	0.275
TParallel	0.002	0.003	0.004	0.038	0.139
Speedup	0	0.66	1	1.92	1.97

Use **4** parallel threads

m=n	**100**	**500**	**1000**	**5000**	**10000**
TSerial	0.000	0.002	0.004	0.073	0.275
TParallel	0.002	0.003	0.005	0.02	0.074
Speedup	0	0.66	0.8	3.65	3.71

Matrix-Vector Multipl – Evaluation(2)

Measurements performed on a laptop with **8 cores:**

For **a given size of the matrix** (**m=n=10000**), vary the number of parallel threads:

threads	**2**	**4**	**8**	**16**	**100**	**1000**
TSerial	0.275	0.275	0.275	0.275	0.275	0.275
TParallel	0.139	0.074	0.049	0.049	0.051	0.074
Speedup	1.97	3.71	5.61	5.61	5.39	3.71

It is BAD to use more threads than the maximum parallel capacity of the system !

Parallel Performance Conclusions

- Creating and managing threads is an additional overhead for the system
- If the serial workload is small (small size of matrix) then the parallel version does not run faster because of this overhead!
- The speedup obtained by parallelization has as upper limit the number of parallel threads
- Using more threads than the physical degree of parallelism (number of cores/processors) has a negative impact on the speedup