

An Introduction to POSIX threads

Alexey A. Romanenko
arom@ccfit.nsu.ru

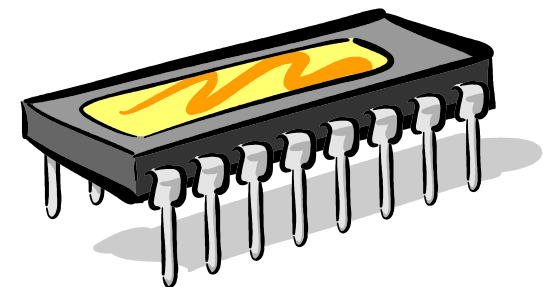
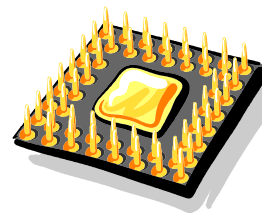
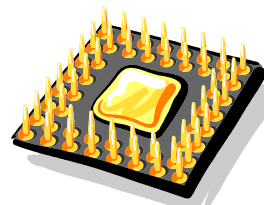
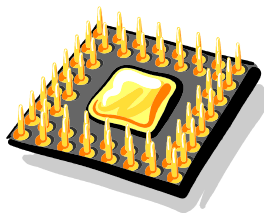
What is this section about?

- POSIX threads overview
- Program compilation
- POSIX threads functions
- etc.

Agenda

- POSIX threads programming model
- POSIX threads overview
 - Launching
 - Threads synchronization
 -

The SMP systems



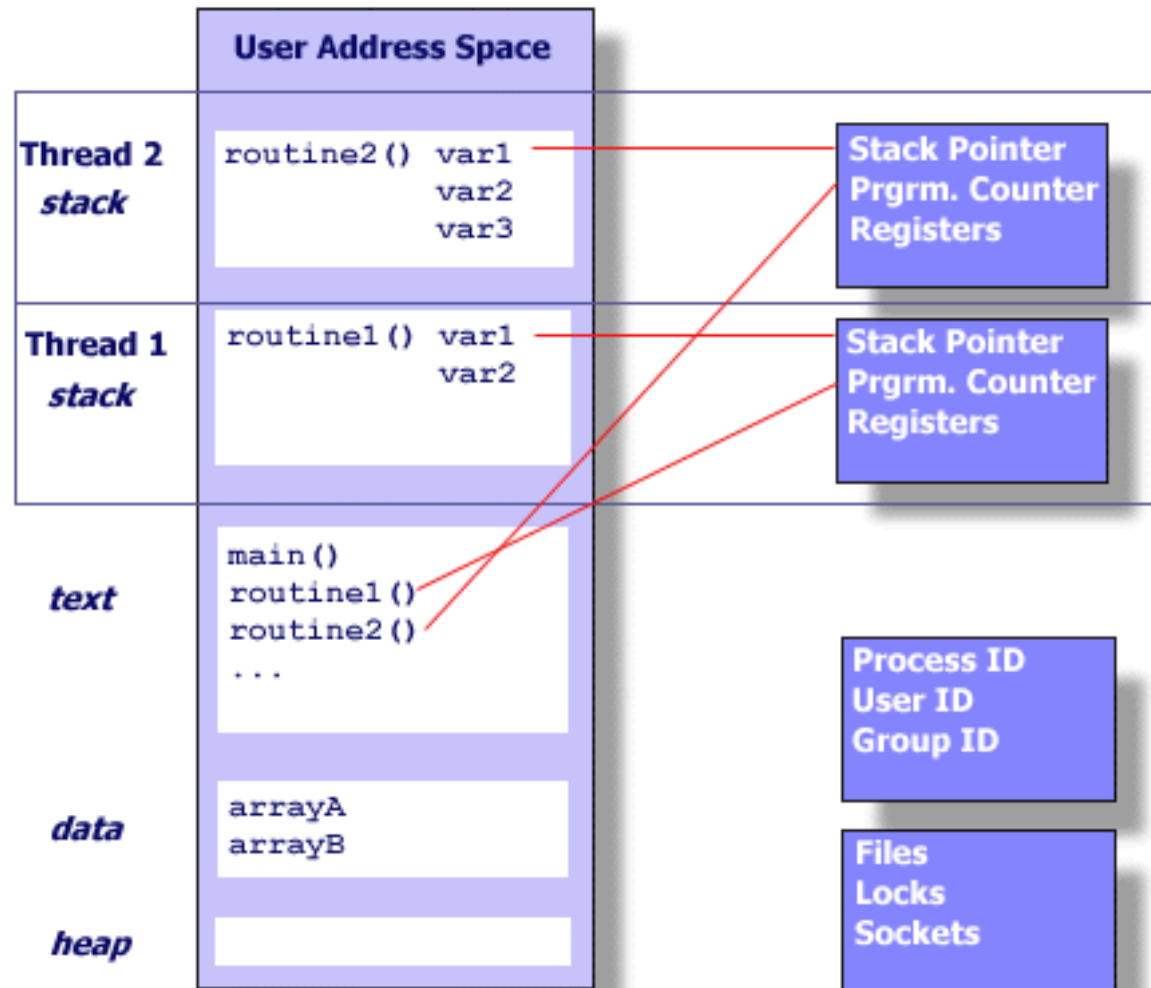
Threads

Threads use and exist within the process resources, yet are able to be scheduled by the operating system and run as independent entities within a process.

A thread can possess an independent flow of control and could be schedulable because it maintains its own.

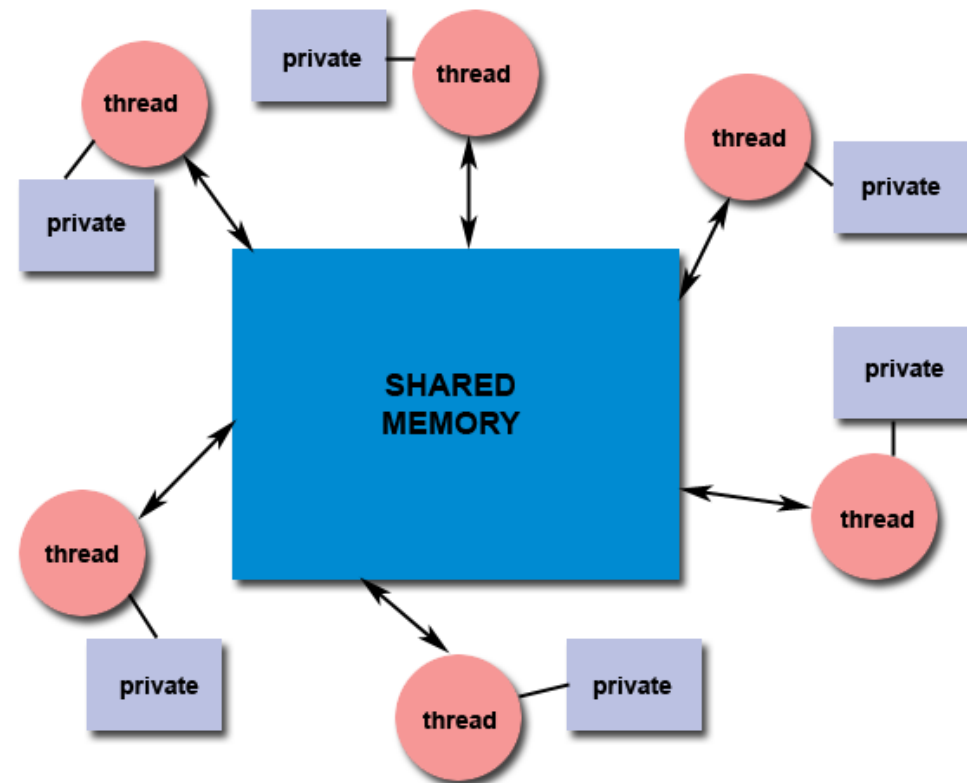
Thread is not a process

Thread



Shared Memory Model

- All threads have access to the same, globally shared, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can be accessed only by the threads that owns
- Explicit synchronization



POSIX threads

- POSIX specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or Pthreads.

Threads attributes

Shared

- process ID
- parent process ID
- process group ID and session ID
- controlling terminal
- user and group IDs
- open file descriptors
- record locks
- file mode creation mask
- current directory and root directory

Distinct

- thread ID (the `pthread_t` data type)
- signal mask (`pthread_sigmask`)
- the `errno` variable
- alternate signal stack (`sigaltstack`)
- real-time scheduling policy and priority

Thread-safe functions

- POSIX.1-2001 requires that all functions specified in the standard shall be thread-safe, except for the following functions:
 - crypt()
 - ctime()
 - encrypt()
 - dirname()
 - localtime()
 - gethostbyname()
 - etc. see specification.

Advantages and Drawbacks of Threads

- Advantages:
 - the overhead for creating a thread is significantly less than that for creating a process (~ 2 milliseconds for threads)
 - multitasking, i.e., one process serves multiple clients
 - switching between threads requires the OS to do much less work than switching between processes

- Drawbacks:
 - not as widely available as longer established features
 - writing multithreaded programs require more careful thought
 - more **difficult to debug** than single threaded programs
 - for single processor machines, creating several threads in a program may not necessarily produce an increase in performance (only so many **CPU cycles** to be had)

POSIX Threads (pthreads)

- IEEE's POSIX Threads Model:
 - programming models for threads in a UNIX platform
 - pthreads are included in the international standards ISO/IEC9945-1
- pthreads programming model:
 - creation of threads
 - managing thread execution
 - managing the shared resources of the process

- main thread:
 - initial thread created when `main()` (in C) or `PROGRAM` (in fortran) are invoked by the process loader
 - once in the `main()`, the application has the ability to create `daughter threads`
 - if the main thread returns, the process terminates even if there are running threads in that process, unless special precautions are taken
 - to explicitly avoid terminating the entire process, use `pthread_exit()`

- thread termination methods:
 - **implicit** termination:
 - thread function execution is completed
 - **explicit** termination:
 - calling `pthread_exit()` within the thread
 - calling `pthread_cancel()` to terminate other threads
- for numerically intensive routines, it is suggested that the application calls p threads if there are p available processors

Sample Pthreads Program in C++ and Fortran 90/95

- The program in C++ calls the `pthread.h` header file. Pthreads related statements are preceded by the `pthread_` prefix (except for semaphores). Knowing how to manipulate pointers is important.
- The program in Fortran 90/95 uses the `f_pthread` module. Pthreads related statements are preceded by the `f_pthread_` prefix (again, except for semaphores).
- Pthreads in Fortran is still **not** an industry-wide standard.


```
1  //*****
2  //   This is a sample threaded program in C++. The main thread creates
3  //   4 daughter threads. Each daughter thread simply prints out a message
4  //   before exiting. Notice that I've set the thread attributes to joinable and
5  //   of system scope.
6  //*****
7  #include <iostream.h>
8  #include <stdio.h>
9  #include <pthread.h>
10
11 #define NUM_THREADS 4
12
13 void *thread_function( void *arg );
14
15 int main( void )
16 {
17     int i, tmp;
18     int arg[NUM_THREADS] = {0,1,2,3};
19
20     pthread_t thread[NUM_THREADS];
21     pthread_attr_t attr;
22
23     // initialize and set the thread attributes
24     pthread_attr_init( &attr );
25     pthread_attr_setdetachstate( &attr, PTHREAD_CREATE_JOINABLE );
26     pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
27
```

```
28     // creating threads
1     for ( i=0; i<NUM_THREADS; i++)
2     {
3         tmp =pthread_create( &thread[i], &attr, thread_function, (void *)&arg[i] );
4
5         if ( tmp !=0 )
6         {
7             cout <<"Creating thread " <<i <<" failed!" <<endl;
8             return 1;
9         }
10    }
11
12    // joining threads
13    for ( i=0; i<NUM_THREADS; i++)
14    {
15        tmp =pthread_join( thread[i], NULL );
16        if ( tmp !=0 )
17        {
18            cout <<"Joining thread " <<i <<" failed!" <<endl;
19            return 1;
20        }
21    }
22
23    return 0;
24 }
25
```

```
54  //*****
55  // This is the function each thread is going to run. It simply asks
56  // the thread to print out a message. Notice the pointer acrobatics.
57  //*****
58  void *thread_function( void *arg )
59  {
60      int id;
61
62      id = *((int *)arg);
63
64      printf( "Hello from thread %d!\n", id );
65      pthread_exit( NULL );
66  }
```

- How to compile:

- in Linux use:

```
> {C++comp}-D_REENTRANT hello.cc -lpthread -o hello
```

- it might also be necessary for some systems to define the `_POSIX_C_SOURCE` (to 199506L)

- Creating a thread:

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

- first argument – pointer to the identifier of the created thread
- second argument – thread attributes
- third argument – pointer to the function the thread will execute
- fourth argument – the argument of the executed function (usually a struct)
- returns 0 for success

- Waiting for the threads to finish:

```
int pthread_join( pthread_t thread, void **thread_return )
```

- main thread will wait for daughter thread *thread* to finish
- first argument – the thread to wait for
- second argument – pointer to a pointer to the return value from the thread
- returns 0 for success
- threads should always be joined; otherwise, a thread might keep on running even when the main thread has already terminated

```
1      !*****
2      ! This is a sample threaded program in Fortran 90/95. The main thread
3      ! creates 4 daughter threads. Each daughter thread simply prints out
4      ! a message before exiting. Notice that I've set the thread attributes to
5      ! be joinable and of system scope.
6      !*****
7      PROGRAM hello
8
9      USE f_pthread
10     IMPLICIT NONE
11
12     INTEGER, PARAMETER :: num_threads =4
13     INTEGER :: i, tmp, flag
14     INTEGER, DIMENSION(num_threads) :: arg
15     TYPE(f_pthread_t), DIMENSION(num_threads) :: thread
16     TYPE(f_pthread_attr_t) :: attr
17
18     EXTERNAL :: thread_function
19
20     DO i =1, num_threads
21         arg(i) =i - 1
22     END DO
23
24     !initialize and set the thread attributes
25     tmp =f_pthread_attr_init( attr )
26     tmp =f_pthread_attr_setdetachstate( attr, PTHREAD_CREATE_JOINABLE )
27     tmp =f_pthread_attr_setscope( attr, PTHREAD_SCOPE_SYSTEM )
28
```

```
29      ! this is an extra variable needed in fortran (not needed in C)
30      flag =FLAG_DEFAULT
31
32      ! creating threads
33      DO i =1, num_threads
34          tmp =f_thread_create( thread(i), attr, flag, thread_function, arg(i) )
35          IF ( tmp /=0 ) THEN
36              WRITE (*,*) "Creating thread", i, "failed!"
37              STOP
38          END IF
39      END DO
40
41      ! joining threads
42      DO i =1, num_threads
43          tmp =f_thread_join( thread(i) )
44          IF ( tmp /=0 ) THEN
45              WRITE (*,*) "Joining thread", i, "failed!"
46              STOP
47          END IF
48      END DO
49
```

```
50      !*****
51      !   This is the subroutine each thread is going to run. It simply asks
52      !   the thread to print out a message. Notice that f_thread_exit() is
53      !   a subroutine call.
54      !*****
55      SUBROUTINE thread_function( id )
56
57      IMPLICIT NONE
58
59      INTEGER :: id, tmp
60
61      WRITE (*,*) "Hello from thread", id
62      CALL f_thread_exit()
63
64      END SUBROUTINE thread_function
```


- How to compile:
 - only available in AIX 4.3 and above:
`>xlf95_r -lpthread hello.f -o hello`
 - the compiler should be thread safe
- The concept for creating and joining threads are the same in C/C++ except that pointers are not directly involved in fortran.
- Note that in fortran some pthread calls are function calls while some are subroutine calls.

Thread Attributes

- detach state attribute:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

- detached – main thread continues working without waiting for the daughter threads to terminate
- joinable – main thread waits for the daughter threads to terminate before continuing further

- contention scope attribute:

```
int pthread_attr_setscope(pthread_attr_t *attr, int *scope);
```

- system scope – threads are mapped one-to-one on the OS's kernel threads (kernel threads are entities that scheduled onto processors by the OS)
- process scope – threads share a kernel thread with other process scoped threads

Threads Programming Model

- **pipeline** model – threads are run one after the other
- **master-slave** model – master (main) thread doesn't do any work, it just waits for the slave threads to finish working
- **equal-worker** model – all threads work

Thread Synchronization Mechanisms

- Mutual exclusion (**mutex**):
 - guard against multiple threads modifying the same shared data simultaneously
 - provides **locking/unlocking** critical code sections where shared data is modified
 - each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

- **Basic Mutex Functions:**

```
int pthread_mutex_init(pthread_mutex_t*mutex, const pthread_mutexattr_t*mutexattr);
```

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

```
int pthread_mutex_unlock(pthread_mutex_t*mutex);
```

```
int pthread_mutex_destroy(pthread_mutex_t*mutex);
```

- a new data type named `pthread_mutex_t` is designated for mutexes
- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the attribute of a mutex can be controlled by using the `pthread_mutex_init()` function
- the lock/unlock functions work in tandem

```
#include <pthread.h>
...
pthread_mutex_t my_mutex; // should be of global scope
...
int main()
{
    int tmp;
    ...
    // initialize the mutex
    tmp = pthread_mutex_init( &my_mutex, NULL );
    ...
    // create threads
    ...
    pthread_mutex_lock( &my_mutex );
        do_something_private();
    pthread_mutex_unlock( &my_mutex );
    ...
    return 0;
}
```

- Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

- Counting Semaphores:
 - permit a limited number of threads to execute a section of the code
 - similar to mutexes
 - should include the `semaphore.h` header file
 - semaphore functions do not have `pthread_` prefixes; instead, they have `sem_` prefixes

- Basic Semaphore Functions:

- creating a semaphore:

- ```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- initializes a semaphore object pointed to by `sem`
    - `pshared` is a sharing option; a value of `0` means the semaphore is local to the calling process
    - gives an initial value `value` to the semaphore

- terminating a semaphore:

- ```
int sem_destroy(sem_t *sem);
```

- frees the resources allocated to the semaphore `sem`
 - usually called after `pthread_join()`
 - an error will occur if a semaphore is destroyed for which a thread is waiting

- semaphore control:

```
int sem_post(sem_t *sem);
```

```
int sem_wait(sem_t *sem);
```

- `sem_post` *atomically* increases the value of a semaphore by 1, i.e., when 2 threads call `sem_post` simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)
- `sem_wait` *atomically* decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
...
sem_t semaphore;    // also a global variable just like mutexes
...
int main()
{
    int tmp;
    ...
    // initialize the semaphore
    tmp = sem_init( &semaphore, 0, 0 );
    ...
    // create threads
    pthread_create( &thread[i], NULL, thread_function, NULL );
    ...
    while ( still_has_something_to_do() )
    {
        sem_post( &semaphore );
        ...
    }
    ...
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```


```
void *thread_function( void *arg )
{
    sem_wait( &semaphore );
    perform_task_when_sem_open();
    ...
    pthread_exit( NULL );
}
```

- the main thread increments the semaphore's count value in the while loop
- the threads wait until the semaphore's count value is non-zero before performing `perform_task_when_sem_open()` and further
- daughter thread activities stop only when `pthread_join()` is called

- Condition Variables:
 - used for communicating information about the state of shared data
 - can make the execution of sections of a codes by a thread depend on the state of a data structure or another running thread
 - condition variables are for **signaling**, not for mutual exclusion; a mutex is needed to synchronize access to shared data

Concluding Remarks

- when making an SMP program, try the simpler approach first:
 - OpenMP directives
 - SMP-enabled libraries
 - automatic SMP capabilities of compilers
- better performance is not guaranteed
- debugging is difficult

- 
- When mixing with MPI, the simplest way is to let only 1 thread handle the communications.
 - So why threads? In some cases, it is the only viable approach.

References

- *Programming with POSIX threads*, by D. Butenhof, Addison Wesley (1997).
- *Beginning Linux Programming*, by R. Stones and N. Matthew, Wrox Press Ltd (1999).
- www.opengroup.org/onlinepubs/007908799/xsh/threads.html
- www.research.ibm.com/actc/Tools/thread.htm
- www.unm.edu/cirt/introductions/aix_xlfortran/xlflr101.htm
- www.emsl.pnl.gov:2080/capabs/mset/training/ibmSP/fthreads/fthreads.html
 - *Programming with Threads*, by G. Bhanot and R. Walkup
 - *Appendix: Mixed models with Pthreads and MPI*, V. Sonnad, C. Tamirisa, and G. Bhanot