1 Controlling Thread Attributes and Synchronization

- Threads and synchronization variables can have several attributes associated with them.
  - Different threads may be scheduled differently (round-robin, prioritized, etc.),
  - They may have different stack sizes, and so on.
  - A synchronization variable such as a mutex-lock may be of different types.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- When creating a thread or a synchronization variable, we can specify the attributes object that determines the properties of the entity.
- Pthreads allows the user to change the priority of the thread.
- Subsequent changes to attributes objects do not change the properties of entities created using the attributes object prior to the change.
- There are several advantages of using attributes objects.

1 It separates the issues of program *semantics and implementation.*
  - Thread properties are specified by the user.
  - How these are implemented at the system level is transparent to the user.
  - This allows for greater portability across operating systems.

2 Using attributes objects improves modularity and readability of the programs.

3 It allows the user to modify the program easily.
  - For instance, if the user wanted to change the scheduling from round robin to time-sliced for all threads,
  - they would only need to change the specific attribute in the attributes object.

- To create an attributes object with the desired properties,

- we must first create an object with default properties and then modify the object as required.

1.1 Attributes Objects for Threads

- **pthread_attr_init**;

```
int
pthread_attr_init (
    pthread_attr_t *attr);
```

- This function initializes the attributes object *attr* to the default values.
- Upon successful completion, the function returns a 0, otherwise it returns an error code.
- The attributes object may be destroyed.
- **pthread_attr_destroy**;

```
int
pthread_attr_destroy (
    pthread_attr_t *attr);
```

- The call returns a 0 on successful removal of the attributes object *attr*.
- Individual properties associated with the attributes object can be changed using the following functions:
- **pthread_attr_setdetachstate** $\Longrightarrow$ to set the detach state
- **pthread_attr_setguardsize_np** $\Longrightarrow$ to set the stack guard size
- **pthread_attr_setstacksize** $\Longrightarrow$ to set the stack size
- **pthread_attr_setstackaddr** $\Longrightarrow$ to set the stack address
- **pthread_attr_setinheritsched** $\Longrightarrow$ to set whether scheduling policy is inherited from the creating thread
- **pthread_attr_setschedpolicy** $\Longrightarrow$ to set the scheduling policy (in case it is not inherited)

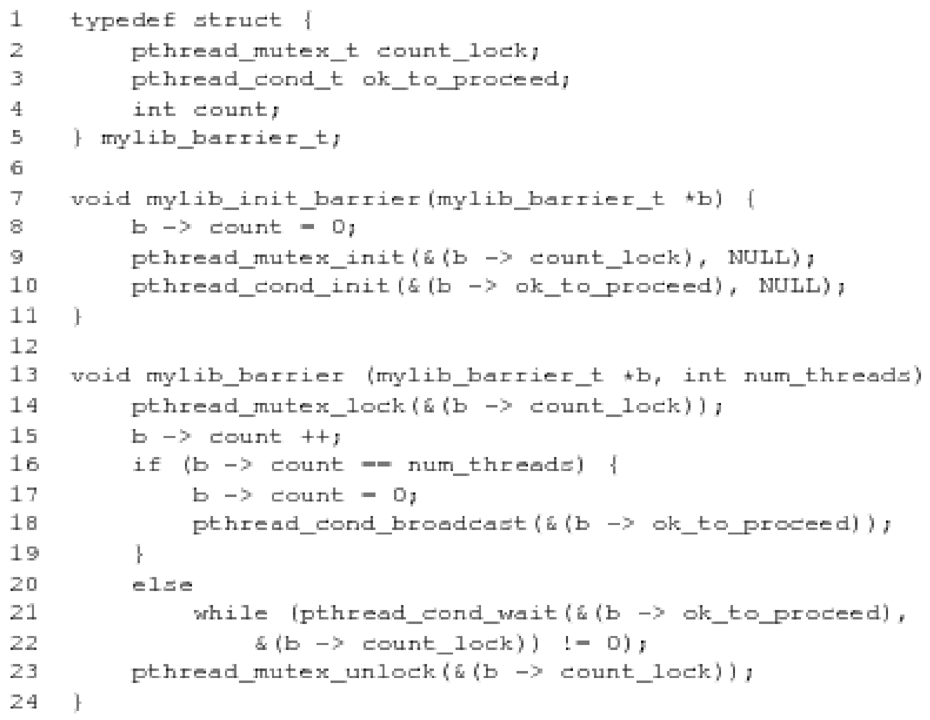
- **pthread_attr_setschedparam** $\Longrightarrow$ to set the scheduling parameters
- **pthread_attr_setprio** $\Longrightarrow$ to set the priority
- **pthread_attr_default, pthread_attr_init**
- For most parallel programs, *default thread properties are generally adequate.*

2 Composite Synchronization Constructs

- While the Pthreads API provides a **basic set of synchronization constructs**, often, there is a need for higher level constructs.
- These higher level constructs can be built using basic synchronization constructs.
- An important and often used construct in threaded (as well as other parallel) programs is a *barrier*.
- A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier.
- Barriers can be implemented using a *counter*, a *mutex* and a *condition variable*.
- A single integer is used to keep track of the number of threads that have reached the barrier.
  - If the *count* is less than the total number of threads, the threads execute a *condition wait.*
  - The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.

The code for accomplishing this is as follows:

- In the above implementation of a barrier, threads enter the barrier and stay until the broadcast signal releases them.
- The threads are released one by one since the mutex *count_lock* is passed among them one after the other.
- The trivial lower bound on execution time of this function is therefore $O(n)$ for n threads.



```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t *b, int num_threads)
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    }
    else
        while (pthread_cond_wait(&(b -> ok_to_proceed),
            &(b -> count_lock)) != 0);
    pthread_mutex_unlock(&(b -> count_lock));
}
```

- This implementation of a barrier can be speeded up using multiple barrier variables.

3 Tips for Designing Asynchronous Programs

- When designing multithreaded applications, it is important to remember that one cannot assume any order of execution with respect to other threads.
- Any such order must be explicitly established using the synchronization mechanisms discussed above: *mutexes*, *condition variables*, and *joins*.
- In many thread libraries, threads are switched at *semi-deterministic* intervals.
- Such libraries (*slightly asynchronous* libraries) are more forgiving of synchronization errors in programs.
- On the other hand, **kernel threads** (threads supported by the kernel) and threads scheduled on multiple processors are less forgiving.

- The programmer must therefore **not make any assumptions** regarding the level of asynchrony in the threads library.
- The following rules of thumb which help minimize the errors in threaded programs are recommended.
- Set up all the requirements for a thread before actually creating the thread. This includes
  - initializing the data,
  - setting thread attributes,
  - thread priorities,
  - mutex-attributes, etc.
- Once you create a thread, it is possible that the newly created thread actually runs to completion before the creating thread gets scheduled again.
- When there is a producer-consumer relation between two threads for certain data items,
- At the producer end, make sure the data is placed before it is consumed and that intermediate buffers are guaranteed to not overflow.
- At the consumer end, make sure that the data lasts at least until all potential consumers have consumed the data.
- This is particularly relevant for stack variables.
- Where possible, define and use group synchronizations and data replication.
- This can improve program performance significantly.
- **While these simple tips provide guidelines for writing error-free threaded programs, extreme caution must be taken to avoid race conditions and parallel overheads associated with synchronization.**

4 OpenMP: a Standard for Directive Based Parallel Programming

- While standardization and support for these threaded APIs has come a long way,
- their use is still predominantly restricted to **system programmers** as opposed to **application programmers**.
- One of the reasons for this is that APIs such as Pthreads are considered to be **low-level primitives**.
- Conventional wisdom indicates that a large class of applications can be efficiently supported by **higher level constructs** (**or directives**)
- which rid the programmer of the mechanics of manipulating threads.
- Such **directive-based languages** have existed for a long time,
- but only recently have standardization efforts succeeded in the form of OpenMP.

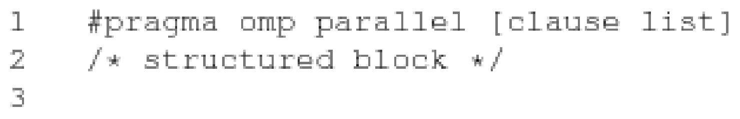
4.1 The OpenMP Programming Model

- OpenMP is an API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- OpenMP directives provide support for **concurrency**, **synchronization**, and **data handling** while avoiding the need for explicitly setting up mutexes, condition variables, data scope, and initialization.
- OpenMP directives in C and C++ are based on the *#pragma* compiler directives.
- The directive itself consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

- OpenMP programs execute serially until they encounter the *parallel* directive.
- This directive is responsible for **creating a group of threads**.

- The exact number of threads can be specified in the directive, set using an environment variable, or at runtime using OpenMP functions.
- The main thread that encounters the *parallel* directive becomes the *master* of this group of threads with id 0.
- The *parallel* directive has the following prototype:

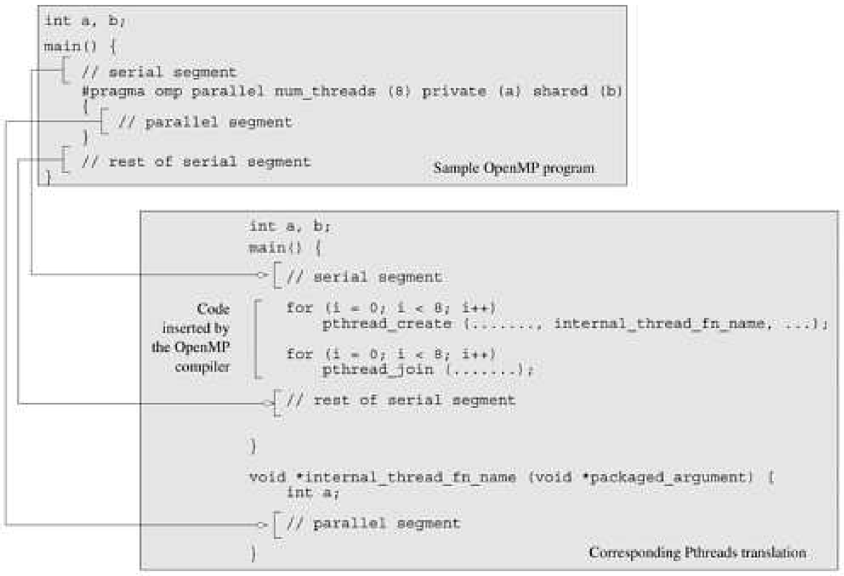


```
#pragma omp parallel [clause list]
/* structured block */

```

- Each thread created by this directive executes the *structured block* specified by the parallel directive.
- It is easy to understand the concurrency model of OpenMP when viewed in the context of the corresponding Pthreads translation.
- In Figure 1, one possible translation of an OpenMP program to a Pthreads program is shown.

- The clause list is used to specify **conditional parallelization**, **number of threads**, and **data handling**.
- **Conditional Parallelization:** The clause *if (scalar expression)* determines whether the parallel construct results in creation of threads.
  - Only one *if* clause can be used with a parallel directive.
- **Degree of Concurrency:** The clause *num_threads (integer expression)* specifies the number of threads that are created by the *parallel* directive.
- **Data Handling:** The clause *private (variable list)* indicates that the set of variables specified is local to each thread.
  - i.e., each thread has its own copy of each variable in the list.
  - The clause *firstprivate (variable list)* is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive.



```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

```
int a, b;
main() {
    // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);
    for (i = 0; i < 8; i++)
        pthread_join (.......);
    // rest of serial segment

}
void *internal_thread_fn_name (void *packaged_argument) {
    int a;
    // parallel segment
}
```

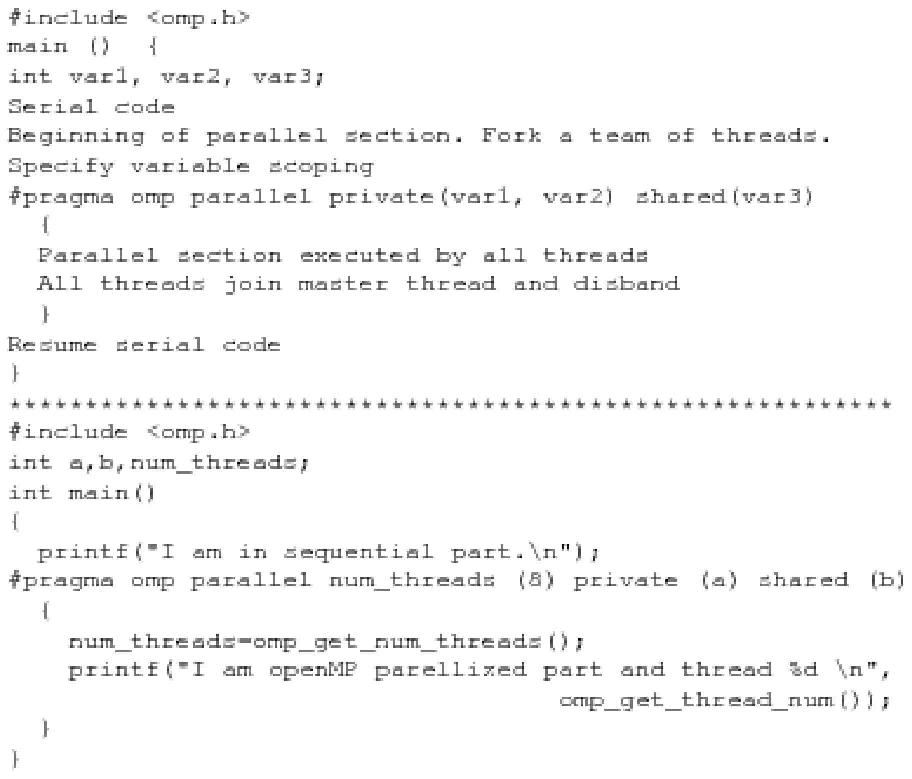
Sample OpenMP program
Code inserted by the OpenMP compiler
Corresponding Pthreads translation

Figure 1: A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

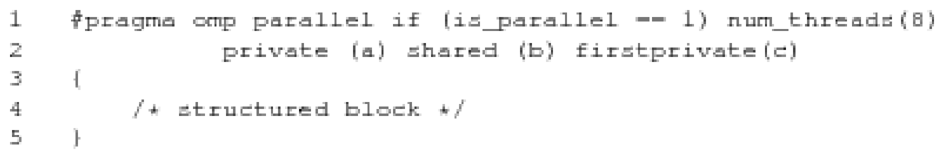
- The clause *shared (variable list)* indicates that all variables in the list are shared across all the threads,
- i.e., there is only one copy. Special care must be taken while handling these variables by threads to ensure serializability.

Using the parallel directive;

- Here, if the value of the variable *is_parallel* equals one, eight threads are created.
- Each of these threads gets private copies of variables a and c, and shares a single value of variable b.
- Furthermore, the value of each copy of c is initialized to the value of c before the parallel directive.
- The clause *default (shared)* implies that, by default, a variable is shared by all the threads.
- The clause *default (none)* implies that the state of each variable used in a thread must be explicitly specified.



```
#include <omp.h>
main ()  {
int var1, var2, var3;
Serial code
Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
  {
  Parallel section executed by all threads
  All threads join master thread and disband
  }
Resume serial code
}
**********************************************************
#include <omp.h>
int a,b,num_threads;
int main()
{
  printf("I am in sequential part.\n");
#pragma omp parallel num_threads (8) private (a) shared (b)
  {
    num_threads=omp_get_num_threads();
    printf("I am openMP parellized part and thread %d \n",
                                 omp_get_thread_num());
  }
}
```



```
  #pragma omp parallel if (is_parallel == 1) num_threads(8)
            private (a) shared (b) firstprivate(c)
  {
      /* structured block */
  }
```

- This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.

- Just as *firstprivate* specifies how multiple local copies of a variable are initialized inside a thread,

- the <u>*reduction* clause</u> specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.

- The usage of the *reduction* clause is *reduction (operator: variable list)*.

- This clause performs a reduction on the scalar variables specified in the list using the *operator*.

- The variables in the list are implicitly specified as being private to threads.

- The *operator* can be one of

```
+, *, -, &, |, ^, &&, and ||.
```

Using the reduction clause;

```
#pragma omp parallel reduction(+: sum) num_threads(8)
 {
 /* compute local sums here */
   }
 /* sum here contains sum of all local instances of sums */
```

- In this example, each of the eight threads gets a copy of the variable *sum*.
- When the threads exit, the sum of all of these local copies is stored in the single copy of the variable (at the master thread).
- **Computing PI** using OpenMP directives (presented a Pthreads program for the same problem).
- The *omp_get_num_threads()* function returns the number of threads in the parallel region
- The *omp_get_thread_num()* function returns the integer id of each thread (recall that the master thread has an id 0).
- The parallel directive specifies that all variables except *npoints*, the total number of random points in two dimensions across all threads, are local.
- Furthermore, the directive specifies that there are eight threads, and the value of sum after all threads complete execution is the sum of local values at each thread.
- A for loop generates the required number of random points (in two dimensions) and determines how many of them are within the prescribed circle of unit diameter.

Note that this program is <u>much easier</u> to write in terms of specifying creation and termination of threads compared to the corresponding POSIX threaded program.

```
/* ******************************************************
 An OpenMP version of a threaded program to compute PI.
 ****************************************************** */

 #pragma omp parallel default(private) shared (npoints) \
                       reduction(+: sum) num_threads(8)
   {
    num_threads = omp_get_num_threads();
     sample_points_per_thread = npoints / num_threads;
     sum = 0;
     for (i = 0; i < sample_points_per_thread; i++) {
  rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
  rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
       if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
           (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
           sum ++;
     }
   }
```