1 Thread Basics: Passing Arguments, Cancellation and Joining

1.1 Passing Arguments to Threads

- **Passing Arguments to Threads**
- The *pthread_create*() function allows the programmer to pass one argument to the thread function.
- For cases where multiple arguments must be passed, this limitation is easily overcome by creating a **structure**.
- This structure contains all of the arguments, and then a pointer is passed to that structure in the *pthread_create*() routine.
- All arguments must be passed by reference and cast to (void *).
- Threads have non-deterministic start-up and scheduling.
- How can you safely pass data to newly created threads?
- **Example:** Demonstrates how to pass a simple integer to each thread.

```
long *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (long *) malloc(sizeof(long));
   *taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
   ...
}
```

Figure 1: Passing single argument to thread function.

- **Example:** Demonstrates how to pass/setup multiple arguments to thread function via a structure.

  Each thread receives a *unique instance* of the structure.

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

Figure 2: Passing multiple arguments to thread function via a structure.

1.2 Thread Cancellation

- **Cancellation.**
- Consider a simple program to evaluate a set of positions in a chess game.
- Assume that there are k moves, each being evaluated by an independent thread.
- If at any point of time, a position is established to be of a certain quality, the other positions that are known to be of worse quality must stop being evaluated.
- In other words, the threads evaluating the corresponding board positions must be canceled.
- Posix threads provide this cancellation feature.
- A thread may cancel itself or cancel other threads.
- **pthread_cancel**.

```
int
pthread_cancel (
    pthread_t   thread);
```

- Here, *thread* is the handle to the thread to be canceled. When a call to this function is made, a cancellation is sent to the specified thread.
- It is not guaranteed that the specified thread will receive or act on the cancellation. Threads can protect themselves against cancellation.
- When a cancellation is actually performed, cleanup functions are invoked for reclaiming the thread data structures.
- The **pthread_cancel** function returns after a cancellation has been sent. The cancellation may itself be performed later.

1.3 Joining and Detaching Threads

- **Joining and Detaching Threads.**
- The main program <u>must wait</u> for the threads to run to completion.
- “Joining“ is one way to accomplish synchronization between threads.
- Function **pthread_join** which suspends execution of the calling thread until the specified thread terminates.

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

- A call to this function waits for the termination of the thread whose id is given by thread.
- A call to this function waits for the termination of the thread whose id is given by thread.
- On a successful call to **pthread_join**, the value passed to **pthread_exit** is returned in the location pointed to by *ptr*.

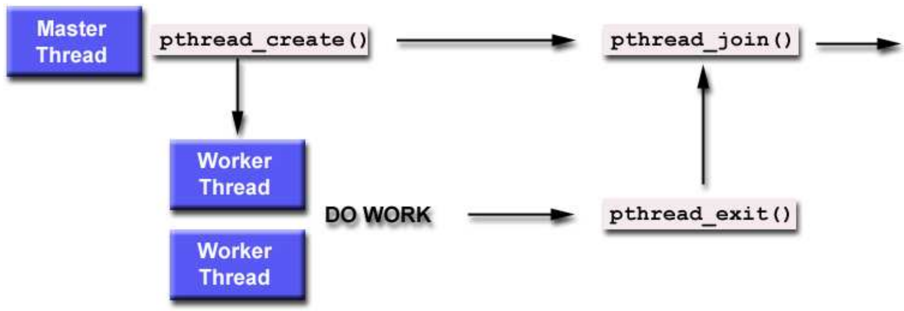

Figure 3: Threads joining.

- On successful completion, **pthread_join** returns 0, else it returns an error-code.
- When a thread is created, one of its attributes defines whether it is **joinable or detached**.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the **attr** argument in the *pthread_create*() routine is used.
- **Detaching**:
- The **pthread_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.
- If a thread requires joining, consider explicitly creating it as joinable (portability).
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state (resources).
- **Reentrant functions** are those that can be safely called when another instance has been suspended in the middle of its invocation.
- All thread functions must be reentrant because a thread can be pre-empted in the middle of its execution.

- If another thread starts executing the same function at this point, a non-reentrant function might not work as desired.

2 Synchronization Primitives in Pthreads

2.1 Mutual Exclusion for Shared Variables

- While communication is implicit in shared-address-space programming,
- much of the effort associated with writing correct threaded programs is spent on **synchronizing concurrent threads** with respect to their data accesses or scheduling.
- Using **pthread_create** and **pthread_join** calls, we can create concurrent tasks.
- These tasks work together to manipulate data and accomplish a given task.
- When multiple threads attempt to manipulate the same data item,
- the results can often be **incoherent** if proper care is not taken to synchronize them.
- Consider the following code fragment being executed by multiple threads.

```
/* each thread tries to update variable best_cost
                                      as follows */
if (my_cost < best_cost)
    best_cost = my_cost;
```

- The variable *my_cost* is thread-local and *best_cost* is a global variable shared by all threads.
- This is an undesirable situation, sometimes also referred to as a **race condition**.
- So called because the result of the computation depends on the race between competing threads.

- To understand the problem with shared data access, let us examine one execution instance of the above code fragment.
- Assume that there are two threads,
- The initial value of *best_cost* is 100,
- The values of *my_cost* are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.
- Depending on which thread executes first, the value of *best_cost* at the end could be either 50 or 75.
- There are two problems here:
  1. non-deterministic nature of the result;
  2. more importantly, the value 75 of *best_cost* is inconsistent in the sense that no serialization of the two threads can possibly yield this result.
- Race condition occurred because the test-and-update operation is an **atomic operation**;
  - i.e., the operation should not be broken into sub-operations.
- Furthermore, the code corresponds to a **critical segment**;
  - i.e., a segment that must be executed by only one thread at any time.
- Many statements that seem atomic in higher level languages such as C may in fact be non-atomic.
  - i.e., a statement of the form $global\_count += 5$ may comprise several assembler instructions and therefore must be handled carefully.
- Threaded APIs provide support for implementing critical sections and atomic operations using **mutex**-locks (mutual exclusion locks).
- Mutex-locks have two states: locked and unlocked.
- At any point of time, **only one thread can lock a mutex lock**.

- A lock is an atomic operation.
  - To access the shared data, a thread must first try to acquire a mutex-lock.
  - If the mutex-lock is already locked, the process trying to acquire the lock is **blocked**.
  - This is because a locked mutex-lock implies that there is another thread currently in the critical section and that no other thread must be allowed in.
  - When a thread leaves a critical section, it must *unlock the mutex-lock* so that other threads can enter the critical section.
- All mutex-locks must be initialized to the unlocked state at the beginning of the program.
- The function **pthread_mutex_lock**;

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

- A call to this function attempts a lock on the mutex-lock *mutex_lock*.
- The data type of a *mutex_lock* is predefined to be *pthread_mutex_t*.
- If the mutex-lock is already locked, the calling thread blocks; otherwise the mutex-lock is locked and the calling thread returns.
- A successful return from the function returns a value 0. Other values indicate error conditions such as deadlocks.
- The function **pthread_mutex_unlock**;

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

- On leaving a critical section, a thread must **unlock the mutex-lock** associated with the section.
- If it does not do so, no other thread will be able to enter this section, typically resulting in a deadlock.
- On calling **pthread_mutex_unlock** function, the lock is relinquished and one of the blocked threads is **scheduled** to enter the critical section.
- The specific thread is determined by the **scheduling policy**.
- if the thread priority scheduling is not implied, the assignment will be left to the native system scheduler and may appear to be more or less **random**.
- **Mutex variables** must be declared with type *pthread_mutex_t*, and must be initialized before they can be used.
- There are two ways to initialize a mutex variable:
  1. Statically, when it is declared. For example: *pthread_mutex_t my-mutex = PTHREAD_MUTEX_INITIALIZER;*
  2. Dynamically, with the **pthread_mutex_init()** routine. This method permits setting mutex object attributes, *attr*.
- If a programmer attempts a **pthread_mutex_unlock** on a previously unlocked mutex or one that is locked by another thread, the effect is undefined.
- The function **pthread_mutex_init**;

```
int
pthread_mutex_init (
    pthread_mutex_t *mutex_lock,
    const pthread_mutexattr_t *lock_attr);
```

- We need one more function before we can start using mutex-locks, namely, a function to initialize a mutex-lock to its unlocked state.
- The mutex is initially unlocked.

- The attributes of the mutex-lock are specified by *lock_attr*.
- If this argument is set to *NULL*, the default mutex-lock attributes are used (normal mutex-lock).
- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can, therefore, lead to **significant performance degradation**.
- It is therefore important to minimize the size of critical sections and to handle critical sections and shared data structures with extreme care.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, *pthread_mutex_trylock*.
- It does not have to deal with queues associated with locks for multiple threads waiting on the lock.
- The function **pthread_mutex_trylock**;

```
int
pthread_mutex_trylock (
    pthread_mutex_t *mutex_lock);
```

- This function attempts a lock on *mutex_lock*.
  - If the lock is successful, the function returns a zero.
  - If it is already locked by another thread, **instead of blocking** the thread execution, it returns a value *EBUSY*.
  - This allows the thread to **do other work** and *to poll the mutex for a lock*.
- Furthermore, **pthread_mutex_trylock** is typically much faster than **pthread_mutex_lock** on typical systems.