Unit 3 – Multi-threaded applications

THREADS

- We will use the term multiprocessor to refer to multi-core processor or multiprocessor systems.

- **Process**: Instance of a computer program running on a machine. While a computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program (e.g.: opening up several instances of the same program often results in more than one process being executed).

- **Thread**: It is an independent functional execution stream that can be scheduled to run as such by the OS.
  - ✓ A thread comprises the machine state necessary to execute a sequence of machine instructions: location of the current instruction, the machine's address and data registers, etc.
  - ✓ You can have many threads sharing an address space, doing different things. On a multiprocessor, the threads in a process can be doing different things simultaneously.
  - ✓ A thread is smaller, faster, and more maneuverable than a traditional process. Once threads are added to an OS, a process becomes just data (address space, files, etc.) plus one or more threads that do something with all the data.
  - ✓ With threads, you can build applications that utilize system resources more efficiently, that run faster on multi-core processor or multiprocessors, and that are easier to maintain.

- A process can be executed by one or many threads. A thread is the part of a process that's necessary to execute code. Thread ≅ "stripped down" process.
- A system can switch between two threads within a process much faster than it can switch between processes. This is mostly because threads within a process share the address space (code, data, stack).

- **Synchronization**: To write a program of any complexity using threads, you'll need to share data between threads, or cause various actions to be performed in some coherent order across multiple threads. To do this, you need to synchronize the activity of your threads.
- **Scheduling**: The act of placing threads onto CPUs, so that they can execute, and of taking them off of those CPUs so that others can run instead.

CONCURRENCY VS. PARALLELISM

- **Concurrency**: It refers to things that *appear* to happen at the same time, but which may occur serially. Concurrent operations may be arbitrarily interleaved so that they make progress independently (one need not be completed before another begins).
  - ✓ Two or more threads may or may not be executing code at the same time, but they are in the middle of it (see Fig. 1(a)). Every multitasking OS has numerous concurrent processes, even though only one could be on a CPU at any given time.

- **Parallelism**: It describes concurrent sequences that proceed simultaneously. True parallelism can only occur on a multiprocessor, but concurrency can occur on both uniprocessors and multiprocessors. Parallelism requires that a program be able to perform two or more computations at once, while concurrency requires only that the programmer be able to pretend that more than two things can happen at once. That is, concurrency is the illusion of parallelism.
  - ✓ Two or more threads actually run at the same time on different CPUs (see Fig. 1(b)). They are of course also running concurrently.
- The vast majority of timing and synchronization issues in multi-threading are issues of concurrency, not parallelism.

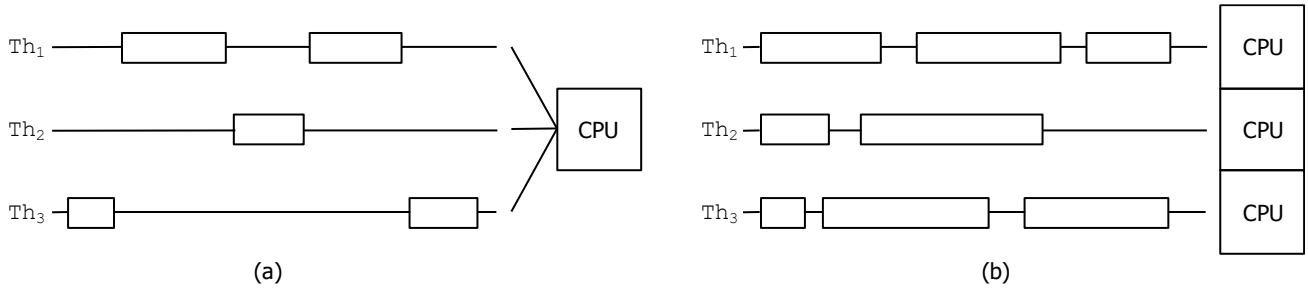

Figure 1. Concurrency vs. Parallelism. (a) Three threads running concurrently on a CPU. (b) Three threads running in parallel on 3 CPUs

MULTITHREADING

- Technique that allows one program to do multiple tasks concurrently.
- There are a wide range of methods to increase the performance of individual CPUs (besides increasing clock speed): long instruction pipelines, superscalar techniques, multi-level caches, compiler optimization techniques (out-of-order execution, predictive branching, VLIW). These methods have their limits. Limiting factors are bus, memory, and peripheral speeds.

- This is where symmetric multi-core processor (SMP) or symmetric multiprocessors come into play. We should write program that take advantage of SMP machines; this means ***threading*** (i.e., multi-threading) although there are other possibilities.
- Benefits of threaded programming model:
  - ✓ Exploitation of program parallelism on multiprocessor hardware.
    - Multithreading allows a process to perform more than one independent computation at the same time.
  - ✓ More efficient exploitation of a program's natural concurrency, by allowing the program to perform computations while waiting for slow I/O operations to complete.
    - Concurrency allows the program to make computation progress while waiting for blocking operations like I/O.
  - ✓ A modular programming model that clearly expresses relationships between independent events within the program.
    - Programming with threads is a good idea even if the code will never run on a multiprocessor. Even single-core processors might experience performance improvement.
- Costs of threading: computing overhead, programming discipline, harder to debug.
- Threads do not necessarily provide the best solution to every programming problem. Threads are not always easier to use, and they don't always result in better performance.

- Definitions for thread libraries: Win32, OS/2, POSIX. Win32 and OS/2 are proprietary and limited to individual platforms. The POSIX specification is intended for all computing platforms. This is a standardized interface for thread implementation.

POSIX THREADS (PTHREADS)

- Standardized programming interface. For UNIX systems, this interface has been specified by the IEEE 1003.1c POSIX (Portable Operating System Interface) standard (1995) and revised in the IEEE 1003.1-2008 and IEEE 1003.1-2017 POSIX standards. Implementations adhering to this standard are referred to as POSIX threads, or Pthreads. Technically we use the term Pthreads API (application programming interface) for the interface specified by the aforementioned standards.
- **Pthreads**: Set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library.
- References:
  - ✓ Blaise Barney, POSIX Threads Programming, Lawrence Livermore National Laboratory.
  - ✓ David R. Butenhof, "Programming with POSIX Threads".

- In shared memory multiprocessor architectures, threads can be used to implement parallelism and/or concurrency. A thread maintains its own Stack Pointer, Register, Scheduling properties (policy, priority), set of pending and blocked signals (UNIX kernel's way of interrupting a running process and informing it that something of interest has occurred), thread specific data.
- Threads within the same process share resources. As a result:
  - ✓ Changes made by a thread to shared system resources (e.g.: closing a file) will be seen by all other threads.
  - ✓ Two pointers having the same value point to the same data.
  - ✓ Reading/writing to the same memory locations is possible, and thus requires explicit synchronization by the programmer.

OVERVIEW

Advantages

- A thread can be created with much less OS overhead than a process.
- There is no intermediate memory copy required, since threads share the same address space within a single process.
- Performance gains and practical advantages:
  - ✓ Overlapping CPU work with I/O: While one tread is waiting for an I/O operation to complete, CPU intensive work can be performed by other threads.
  - ✓ Priority/real-time scheduling: more important tasks can be scheduled to supersede or interrupt lower priority tasks.
  - ✓ Asynchronous event handling: tasks that service events of indeterminate frequency and duration can be interleaved.

Designing Threaded Programs

- On multi-core machines, pthreads are ideally suited for parallel programming.
- For a program to take advantage of pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently.
- Shared memory model: all threads have access to the same global, shared memory. Threads can also have their own private data. Programmers are responsible for synchronizing access (protecting) global shared data.
- Thread-safeness: Application's ability to execute multiple threads simultaneously without affecting shared data.
- The subroutines that comprise the Pthreads API include:
  - ✓ Thread management: creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling, etc).
  - ✓ Mutexes: Routines that deal with synchronization (mutex = mutual exclusion). Mutex functions provide for creating, destroying, locking, and unlocking mutexes.
  - ✓ Condition variables: Routines that address communications between threads that share a mutex. It includes functions to create, destroy, wait, and signal based upon specified variable values.
  - ✓ Synchronization: Routines that manage read/write locks and barriers.

- For convenience, Table I lists the most common pthreads routines, organized by their usage.

TABLE I. SUMMARY OF COMMON PTHREADS DIRECTIVES AVAILABLE IN "PTHREAD.H"

Declare identifiers	`pthread_t pthread`	Thread identifier
	`pthread_mutex_t mutex`	Mutex
	`pthread_cond_t cond`	Condition variable
	`pthread_attr_t attr`	Thread attributes object
	`pthread_mutexattr_t mutex_attr`	Mutex attributes object
	`pthread_condattr_t condattr`	Condition variable attributes object

Create and terminate threads	`int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);`
	`int pthread_exit (void *value_ptr);`
	`int pthread_cancel (pthread_t thread);`	

Join and detach threads	`int pthread_join (pthread_t thread, void **value_ptr);` `int pthread_detach (pthread_t thread);`	
Miscellaneous	`pthread_t pthread_self (void);` `int pthread_equal (pthread_t t1, pthread_t t2);`	
Create and destroy mutexes	`int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *mutex_attr);` `int pthread_mutex_destroy (pthread_mutex_t *mutex);`	

Lock/unlock mutexes	`pthread_mutex_lock (pthread_mutex_t *mutex);`
	`pthread_mutex_trylock (pthread_mutex_t *mutex);`
	`pthread_mutex_unlock (pthread_mutex_t *mutex);`	

Create/destroy condition variables	`int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *condattr);` `int pthread_cond_destroy (pthread_cond_t *cond);`	

Wait/signal on condition variables	`int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);`
	`int pthread_cond_signal (pthread_cond_t *cond);`
	`int pthread_cond_broadcast (pthread_cond_t *cond);`	

CREATING AND USING THREADS

- **Declare an identifier**: in a program a thread is represented by a thread identifier of type `pthread_t`
  `pthread_t thread;`
- **Create a thread**: A thread begins by calling some function that you provide. This is called a ***thread start function***:
  `int pthread_create {pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);`
  - ✓ The ***thread start function*** (`void *(start)(void *)`) expects an argument `arg` of type `void *` and should return an argument of the same type (`void *`).
  - ✓ `pthread_create` returns an identifier referred to by the `thread` argument, by which your code refers to the new thread.
  - ✓ When a C program runs, it begins with the special function `main()`. In a threaded program, this stream of instructions is called the "initial thread" or "main thread". You can do anything within this initial thread that you can do with any other thread (e.g.: determine its own thread identifier, terminate itself).
    - In this initial thread, the ***thread start function*** (`main()`) is called from outside the program (e.g. UNIX systems link your program with an .o file, which initializes the process and then calls `main()`).
  - ✓ ***Detaching***: To ensure that resources used by terminated threads are available to the process, you should always detach each thread you create when you are done with it. Threads terminated but not detached may retain virtual memory, including their stacks and other system resources. Detaching a thread (including a running thread) informs the system that you no longer need that thread and allows the system to reclaim the resources it has allocated to the thread. Note that detaching a running thread will not affect the thread.
    - To create a thread that you will never need to control, you can use the attribute of "detached" when creating it.
    - If you do not want to wait for a thread that you created (and you know you will no longer need to control the thread), you can detach at any time using a detaching function: `int pthread_detach (pthread_t thread);`
      - A thread might detach itself, or any other thread that knows its identifier may detach it any time.
    - If you use the ***join*** function (that specifies a thread), it will automatically detach that thread.

- **Exit thread**: This is done by:
  - ✓ Specifically calling the thread exit function: `int pthread_exit (void *value_ptr);`
  - ✓ Simply returning from the ***thread start function*** when its work is done.

- **Wait for a thread to exit**: If you specifically want to wait for a thread to exit (say you created 20 threads and you can't continue until they are all finished), you can call the ***join*** function. It will also let you know a thread's return value:
  - ✓ `int pthread_join (pthread_t thread, void **value_ptr);`
  - ✓ This ***join*** function will block the caller until the specified thread has terminated, and then it can store the terminated thread's return value (`value_ptr`). It will also automatically detach the specified thread.

EXAMPLES

- **Sample Program I** (exit thread by returning from *thread_start function* and using *join* function):

```
#include <pthread.h>
#include "errors.h"

// thread start function:
void *thread_routine (void *arg) { // The returned value is always NULL
   return arg;                     // it 'returns' arg to satisfy the standard function prototype
}

int main (int argc, char *argv[]) {
   pthread_t thread_id;
   void *thread_result;
   int status;

   status = pthread_create (&thread_id, NULL, thread_routine, NULL);
   if (status != 0) err_abort (status, "Create thread");

   status = pthread_join (thread_id, &thread_result); // waits for thread to complete
   if (status != 0) err_abort (status, "Join thread");

   if (thread_result == NULL) return 0; else return 1;
}
```

  - ✓ ***thread start function***: `void *thread_routine (void *arg)` Argument that is passed: `NULL`
  - ✓ The program creates a thread by calling `pthread_create`, and then waits for it by calling `pthread_join`. Returning from `main` will cause the process to terminate, along with all threads.
  - ✓ When the `pthread_join` completes, the program checks the thread's return value (`thread_result`), to ensure that the thread returned the value it was given. The program exits with 0 (success) if the value is NULL, or with 1 otherwise.

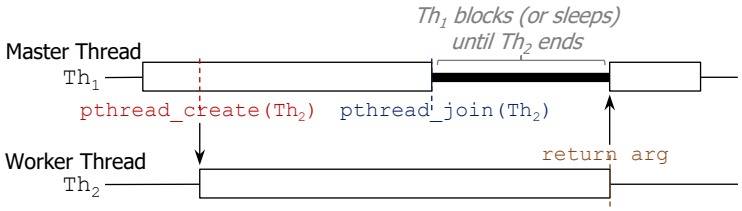

Figure 2. Sample Program I: Master thread creates and joins a thread that uses `return arg` to exit.

  - ✓ When the call to `pthread_join` returns, that thread has been detached and you can't join with it again.

- **Sample Program II** (exit thread via the `pthread_exit` function and <u>not</u> using *join* function):

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5 // 5 threads are created

void *PrintHello(void *threadid) { // thread start function. There is no return value
   long tid;
   tid = (long) threadid;
   printf ("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL); // each thread terminates with a call to pthread_exit
 }

int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;

   for (t=0; t < NUM_THREADS; t++) {
      printf ("In main: creating thread %ld\n", t); // threads do not execute and finish consecutively
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc) { printf ("ERROR; return code from pthread_create() is %d\n", rc); exit(-1); }
   }

   pthread_exit(NULL); // Important. It blocks main to support the threads it created till they're done
}
```

  - ✓ ***thread start function***: `void *PrintHello (void *threadid)` Argument that is passed: `void* t`
    - Note that this function does not specify a return value, and instead the thread uses `pthread_exit` to exit.
  - ✓ Since there is no *join* function, `main()` must explicitly call `pthread_exit()` as the last thing it does (`main()` will block and be kept alive to support the threads it created until they are done). If you don't, all the threads it created will terminate if `main()` finishes before the threads it spawned.
  - ✓ The program creates 5 threads and then waits until they complete. Note that created threads do not necessarily execute and finish in a consecutive fashion.
  - ✓ Program Output:

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
```

```
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

- **Sample Program III** (exit thread via the `pthread_exit` function and not using ***join*** function):

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

void* print_i(void *ptr) { // This function will run concurrently
   printf("a\n");
   printf("b\n");
   pthread_exit (NULL);
}

int main() {
   pthread_t t1;
   int iret1 = pthread_create(&t1, NULL, print_i, NULL); // argument NULL is provided
   printf("c\n");
   return 0;
}
```

  - ✓ ***thread start function***: `void *print_i (void *ptr)` Argument that is passed: `NULL`
  - ✓ This example illustrates the issues you might run into if there is no synchronization mechanism (like the ***join*** function).
  - ✓ A thread is created, but we do not wait for the thread to exit (no ***join***), or include `pthread_exit` in `main` so that it blocks until the thread is done. This leads to unpredictable behavior. For example, if `main` thread completes before the `print_i` thread has, the `print_i` thread will die. Here, `print_i` thread might not print anything, print one letter, or print both.
  - ✓ Program Output (the output is random):

```
c
```

```
c
a
```

```
a
c
```

- Note that `pthread_join()` blocks the calling thread until the specified thread terminates (this is a way to *synchronize* threads). Fig. 3 depicts this, where a thread exits by calling `pthread_exit`.
  - ✓ You don't need to wait for a thread, but if you don't, you will need to somehow ensure the process runs until the thread completes.

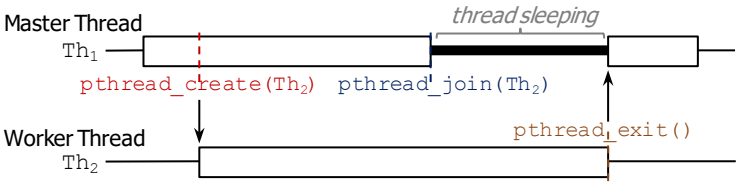

Figure 3. Master thread creates and joins a thread that uses `pthread_exit` to exit.

LIFE CYCLE OF A THREAD

- This encompasses creation to recycling, all through all the scheduling states threads can assume along the way. At any instant, a thread is in one of the four basic states listed in Table II. Fig. 4 shows the relationships between these thread states, and the events that cause threads to move from one state to another.
- The "initial thread" of a process is created when a process is created. All other threads must be explicitly created.

TABLE II. THREAD BASIC STATES

State	Meaning
Ready	Thread is able to run, but it is waiting for a processor. It may have just started, or just been unblocked, or preempted by another thread.
Running	Thread is currently running. On a multiprocessor, there may be more than one running thread in the process.
Blocked	The thread is not able to run because it is waiting for an event (e.g.: a condition variable, to lock a mutex, I/O operation to complete).
Terminated	Thread has terminated by returning from its start function, calling its exit function, or having been cancelled. It was not detached and has not yet been joined. Once detached of joined, it will be recycled.

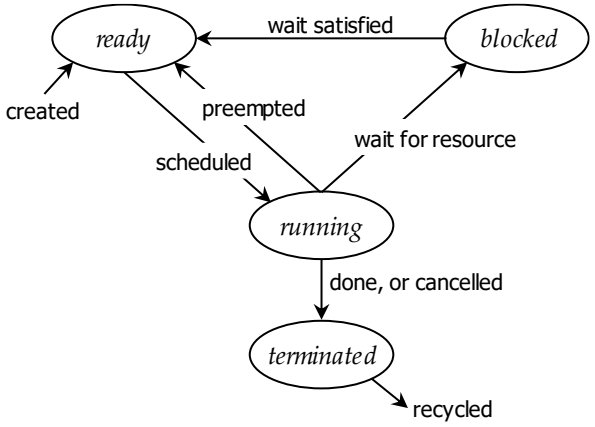

Figure 4. State transitions for a thread.

- A thread is *ready* when it is first created, and whenever it is unblocked (goes from *blocked* to *ready*, so that it is once again eligible to run). *ready* threads are waiting for a processor. Depending on scheduling constraints, it may remain in this state for a long time before executing.

- ✓ Sample program I: the thread running `thread_routine` becomes *ready* during main's call to `pthread_create`.
- ✓ There is no synchronization between a thread return from `pthread_create` and the scheduling of a new thread: a thread may start before the other thread returns (it may even run to completion and terminate before `pthread_create` returns).

- A thread becomes *running* when it was *ready* and a processor selects the thread for execution. In a uniprocessor, this usually means that another thread has become *blocked* or has been preempted; the blocking (or preempted) thread saves its context and restores the context of the next *ready* thread to replace itself. On a multiprocessor, a previously unused processor may execute a *ready* thread without any other thread blocking.
  - ✓ A thread is preempted (goes from *running* to *ready*) if the system reassigns the processor on which it was running.
  - ✓ When the new thread runs, it calls the specified ***thread start function***. It may be preempted by other threads or block itself to wait for external events any number of times.

- A thread becomes *blocked* if it needs a resource that is not available (it attempts to lock a mutex that is currently locked, it waits on a condition variable, it waits for a signal that is not currently pending, it attempts an I/O operation that can't be immediately completed, etc.). When a thread is unblocked, it is made *ready* again (it may execute immediately if the processor is available).
  - ✓ Sample program I: the main thread is blocked at `pthread_join`, to wait for the thread it created to run. If the thread had not already run at this point, it would move from *ready* to *running* when main becomes *blocked*. As the thread runs to completion and returns, the main thread will be unblocked; returning to the *ready* state. When processor resources are available, either immediately or after the thread becomes *terminated*, main will again become *running*, and complete.

- Eventually, a thread completes (it returns from the ***thread start function*** or calls the `pthread_exit` function) or it is cancelled by another thread (`pthread_cancel`). In any case, it is in the *terminated* state. If the thread was already detached, it is immediately recycled. Otherwise the thread remains in the *terminated* state until joined (`pthread_join`) or detached.
  - ✓ Sample program I: the thread that had run `thread_routine` will be recycled by the time the main thread returns from the `pthread_join` call.
  - ✓ Recycling releases any system or process resources that weren't released at termination (storage for thread's return value, the stack, memory used to store register state, etc.).

SYNCHRONIZATION

- To write any kind of concurrent program, you need to share data between threads or cause various actions to be performed in some coherent order across multiple threads. Thus, you must be able to reliably synchronize the different threads.
- Without synchronization, two or more threads will start to change some data at the same time, one will overwrite the other. To avoid this, threads must be able to reliably coordinate their actions.
- Synchronization isn't important just when you modify data. You also need it when a thread needs to read data that was written by another thread (if the order in which data was written matters).
- Approaches:
  - ✓ ***join*** function (`pthread_join`): Basic mechanism. You wait for threads to finish before you keep executing.
  - ✓ **Mutex:** We control the thread access to data. We lock/unlock portion of code that is mutually exclusive (only 1 thread can use it at a time).
  - ✓ **Condition Variable:** Synchronization is based on actual value of data. Used with a mutex lock.

BASIC TERMS

- ***Invariants***: assumptions made by a program, especially about relationship between sets of variables.
  - ✓ Example: queue package.
    - Each queue has a queue header (pointer to the first queued data element).
    - Each data element includes a pointer to the next data element. The data isn't all that is important: the queue package relies on relationships between that data.
      - For example, the queue header must be NULL or contain a pointer to the first queued data element.
      - Each data element must contain a pointer to the next data element, or NULL if it is the last.
      - These relationships are the ***invariants*** of your queue package.
    - When a program encounters a broken invariant (e.g.: it dereferences a queue header containing a pointer to something that is not a valid data element), the program will produce incorrect results or fail immediately.
  - ✓ Most invariants can be broken, and are routinely broken, during isolated areas of code. The trick is to be sure that broken invariants are always repaired before unsuspecting code encounter them.
- ***Critical sections***: areas of code that affect a shared state. When you remove an element from a queue, you can see the code performing the removal as a ***critical section***, or you can see the state of the queue as an invariant.
- ***Predicates***: logical expressions that describe the state of invariants needed by your code. Example: Boolean variable, result of whether a pointer is NULL, result of whether a counter is greater than a threshold, returned value from function.

MUTEXES

- Most general way to synchronize between threads: ensure all memory access to the same (or related) data are "mutually exclusive"; only one thread is allowed to write at a time.
- Mutex (mutual exclusion) variables are the basic *pthreads* synchronization mechanism: they implement thread synchronization and protect shared data when multiple writes occur.
- A mutex provides a single, absolute owner for a specific section of code that affects a shared state (called *critical section*).
  - ✓ Mutex variable: "lock" protecting access to a shared data resource. In *pthreads*, only one thread can lock (or own) a mutex variable at any given time. If a thread locks a mutex, no other thread can own that mutex until the owning thread unlocks that mutex.

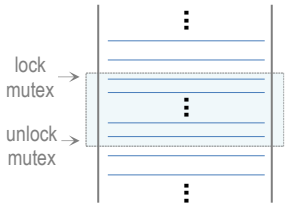

Figure 5. Mutex usage. Only one thread owns the mutex (code portion) at a time.

- Typical sequence in the use of a mutex:
  1. Create and initialize a mutex variable.
  2. Several threads attempt to lock the mutex.
  3. Only one succeeds and that thread owns the mutex. The ones that did not succeed become blocked.
  4. The owner thread performs some set of actions.
  5. The owner unlocks the mutex.
  6. Another thread acquires the mutex and repeats the process.
  7. Finally, the mutex is destroyed.

Usage:

- Declare mutex variables:
  `pthread_mutex_t mutex;`
  - ✓ Note that the mutex variables must be initialized before they can be used:
    - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`. This is static initialization, when it is declared.
    - `int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *mutex_attr)`. Dynamic initialization. This permits setting mutex object attributes, `mutex_attr`.
  - ✓ The mutex is initially unlocked.

- Locking and unlocking mutexes:
  - ✓ `pthread_mutex_lock (pthread_mutex_t *mutex)`: Used by a thread to acquire a lock on the specific mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

- ✓ `pthread_mutex_trylock (pthread_mutex_t *mutex)`: Will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. It may be useful in preventing deadlock conditions.
- ✓ `pthread_mutex_unlock (pthread_mutex_t *mutex)`: Will unlock a mutex if called by the owning thread. This is required after the thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error returns if the mutex is owned by another thread or if the mutex was already unlocked.

- Free (destroy) a mutex object that is no longer needed:
  - ✓ `int pthread_mutex_destroy (pthread_mutex_t *mutex);`

- Fig. 6 shows a timing diagram of three threads (T_1, T_2, T_3) sharing a mutex. During execution, a thread (represented by a line) might be in these different locations:
  - ✓ "Owning the mutex" box: it means that the associated thread owns the mutex.
  - ✓ "Waiting to own a mutex" box: it means the associated thread is blocked and waiting to own the mutex.
  - ✓ Neither box: It means that the associated thread does not own the mutex.

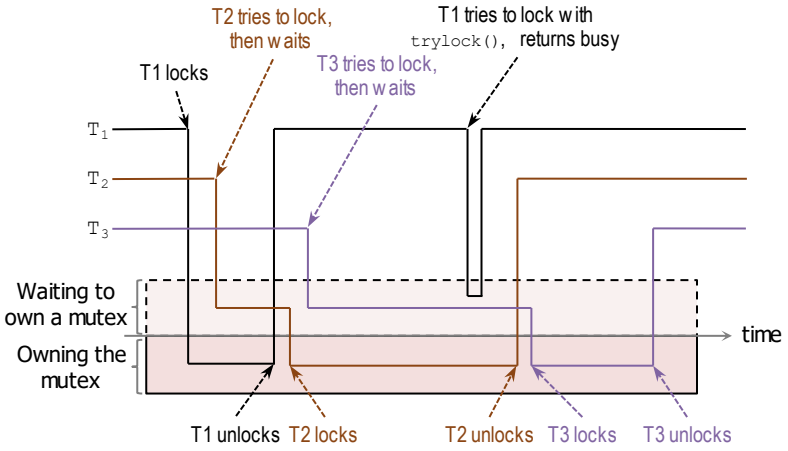

Figure 6. Mutex operation on 3 threads (T_1, T_2, T_3).

- Initially, the mutex is unlocked. T_1 attempts to lock the mutex (`pthread_mutex_lock`); it succeeds immediately (T_1 owns the mutex) as there is no contention. Later, T_2 attempts to lock the mutex and, because the mutex is already locked, T_2 blocks, waiting to own a mutex. Later, T_1 unlocks the mutex, unblocking T_2, which then succeeds in locking the mutex. Later, T_3 attempts to lock the mutex, and blocks. Later, T_1 calls `pthread_mutex_trylock` to try to lock the mutex and, because the mutex is locked, returns immediately with a busy error code. T_2 unlocks the mutex, which unblocks T_3 so that it can lock the mutex. Finally, T_3 unlocks the mutex to complete our example.

- A mutex is usually meant for small portions of code. The larger the portion of the code, the more the application will behave like a sequential program.

- **Example:**

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; // mutex declaration and static initialization

void print(char* a, char* b) { // try commenting/uncommenting the mutex below and look at output
  pthread_mutex_lock(&mutex1); // comment out
  printf ("1: %s\n", a);
  sleep(1);
  printf ("2: %s\n", b);
  pthread_mutex_unlock(&mutex1); // comment out
}

// These two functions (print_i, print_j) will run concurrently.
void* print_i(void *ptr) { print("I am", " in i"); pthread_exit(NULL); }
void* print_j(void *ptr) { print("I am", " in j"); pthread_exit(NULL); }

int main() {
  pthread_t t1, t2;
  int status;

  status = pthread_create(&t1, NULL, print_i, NULL);
  status = pthread_create(&t2, NULL, print_j, NULL);
  status = pthread_join(t1, NULL);
  status = pthread_join(t2, NULL);
  return 0;
}
```

  - ✓ Without a mutex, the two created threads execute concurrently. Though not always the case, the most common execution sequence is: `t1` starts executing, when it hits `sleep(1)`, `t2` will start executing.

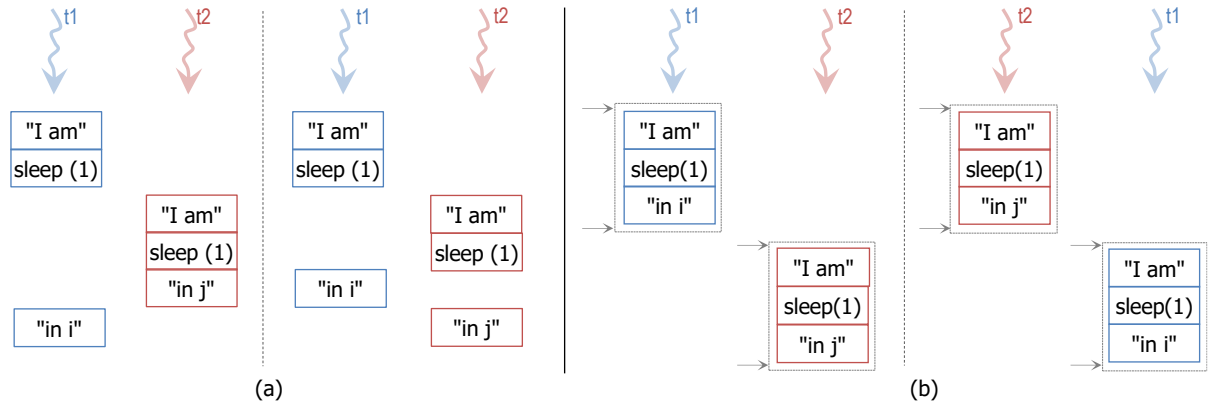

Figure 7. (a) Two execution samples without mutex. (b) Two execution samples with a mutex.

- ✓ With a mutex, the two created threads execute concurrently. However, when a thread (`t1` or `t2`) starts executing, it will lock a portion of code. When that portion of code is completed, it is unlocked, so that the other thread can execute. Usually, `t1` will execute first.

Program Output:
```
1: I am
2:  in i
1: I am
2:  in j
```

Program Output (without mutex, i.e., mutex commented out):
```
1: I am
1: I am
2:  in j
2:  in i
```

- ✓ Note that in this example, we lock/unlock essentially the entire code within a thread. Usually, we only lock a portion of the code within a thread.
- ✓ Even though we do not write on memory on this code, this example illustrates how threads do not execute and finish in a consecutive fashion; thereby creating potential trouble which might require the use of mutexes.

- **Example (dot product):** Length of Vectors: `VECLENxNUMTHREADS`

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
// Source: https://computing.llnl.gov/tutorials/pthreads/
typedef struct {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex */
#define NUMTHRDS 4
#define VECLEN 100000 // This is the length of the vector each thread operates on
   DOTDATA dotstr; // global structure
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

void *dotprod (void *arg) { // thread start function
   int i, start, end, len;
   long offset;
   double tsum, *x, *y;
   offset = (long)arg; // argument passed from main()

   len = dotstr.veclen; x = dotstr.a;        y = dotstr.b;
   start = offset*len;  end = start + len; // [start, end): range on which this thread operates

   tsum = 0; // sum computed by a single thread
   for (i = start; i < end; i++) { tsum += (x[i] * y[i]); } // perform dot product

// Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += tsum;
   printf("Thread %ld did %d to %d:  tsum=%f global sum=%f\n",offset,start,end-1,tsum,dotstr.sum);
   pthread_mutex_unlock (&mutexsum);
   pthread_exit((void*) 0);
}

int main (int argc, char *argv[]) {
   long i;
   double *a, *b;
```

```
    void *status;

    a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
    b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

    for (i=0; i < VECLEN*NUMTHRDS; i++) { a[i]=1;   b[i]=a[i]; } // input data (just 1's)

    dotstr.veclen = VECLEN; dotstr.a = a; dotstr.b = b; dotstr.sum=0; // initializing global variables
    pthread_mutex_init(&mutexsum, NULL); // mutex: dynamic initialization

    // create and join threads
    for (i=0; i < NUMTHRDS; i++) pthread_create(&callThd[i], NULL, dotprod, (void *)i);

    for (i=0; i < NUMTHRDS; i++) pthread_join(callThd[i], &status);

    printf ("Sum =  %f \n", dotstr.sum); // print out results
    free (a); free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit (NULL);
}
```

- ✓ ***thread start function***: `void *dotprod (void *arg)` Argument that is passed: `void *i`
- ✓ The program creates 4 threads, and then waits for all threads to exit by calling `pthread_join` on each thread.
- ✓ The main data (input vectors, vector length, sum) is available to all threads through a global structure. Each thread works on a different part of the data
- ✓ Each thread gets an index i (called **`offset`** locally) and processes over the range $[i \times len\,, (i+1) \times len\, - 1]$, where `len=VECLEN`. Then, it locks a mutex, updates the global variable `dotstr.sum`, and unlocks the mutex.
- ✓ Program Output: The threads may not execute in order. This is the output that we got in one execution instance:

```
thread 0: did 0 to 99999: tsum = 100000  global sum = 100000
thread 1: did 100000 to 199999: tsum = 100000  global sum = 200000
thread 3: did 300000 to 399999: tsum = 100000  global sum = 300000
thread 2: did 200000 to 299999: tsum = 100000  global sum = 400000
```

- ✓ Fig. 8 depicts the threads' operation over time. Here, the order of execution is T_0, T_1, T_3, T_2. Other execution instances are: [T_0, T_1, T_2, T_3], [T_0, T_2, T_1, T_3]. Regardless of the order, the thread id corresponds to the range that it is processed. Each thread performs the sum of products operation over its associated range before attempting to lock a mutex.

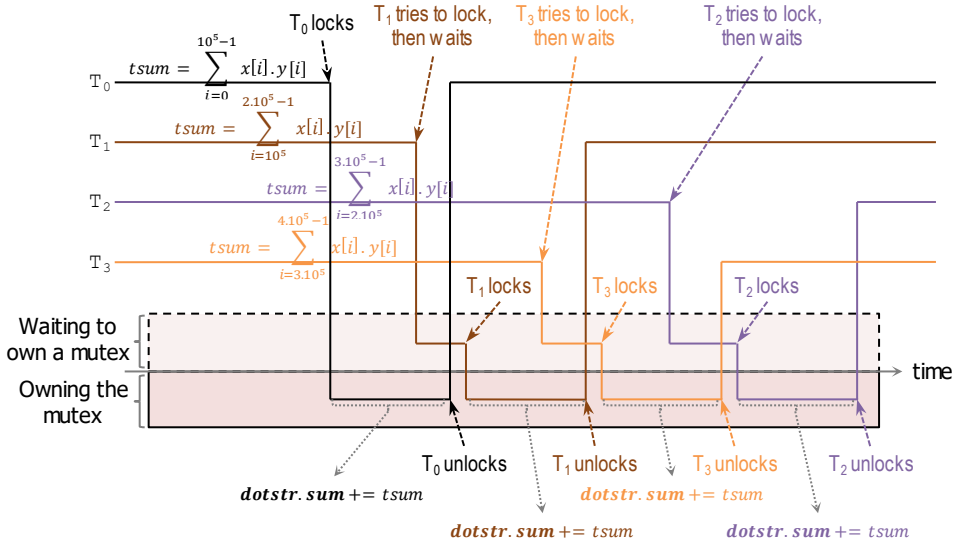

Figure 8. Mutex operation on 4 threads (T_1, T_2, T_3, T_4). The points at which T_1, T_3, T_2 attempt to lock a mutex vary widely. Threads might run on a uniprocessor or a multiprocessor

CONDITION VARIABLES

- Mechanism used for communicating information about the state of shared data such as a queue no longer empty).
  - ✓ Mutexes implement synchronization by controlling thread access to data.
  - ✓ Condition variables allow threads to synchronize based on the actual value of data.
- Without condition variables, the programmer would need to have thread continually polling (possibly in a critical section) to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- Condition variable: signaling mechanism associated with a mutex, always used in conjunction with a mutex lock.
  - ✓ Condition variables are for signaling, they do not provide mutual exclusion.
- Sample sequence for using condition variables for two threads.

TABLE III. REPRESENTATIVE SEQUENCE FOR USING CONDITION VARIABLES

Main Thread: ▪ Declare and initialize global data/variables which require synchronization (e.g.: *count*) ▪ Declare and initialize a condition variable ▪ Declare and initialize an associated mutex ▪ Create threads A and B to do work	
Thread A ▪ Do work up to the point where certain condition must occur (e.g.: *count* reaching a specified value). ▪ Lock associated mutex and check value of global variable. ▪ Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread B. *Note that this call automatically and atomically unlocks the associated mutex variable* so that it can be used by Thread B. ▪ When signaled, wake up. Mutex is automatically and atomically locked. ▪ Explicitly unlock mutex ▪ Continue	**Thread B** ▪ Do work ▪ Lock associated mutex ▪ Change the value of the global variable that Thread A is waiting upon. ▪ Check value of the global Thread A wait variable. If it fulfills the desired condition, signal (`cond_signal` or `cond_broadcast`) Thread A. ▪ Unlock mutex ▪ Continue
Main thread ▪ Join/continue	

Usage:

- Declaration:
  `pthread_cond_t cond`
  - ✓ Note that the condition variables must be initialized before they can be used:
    - `pthread_cond_t cond = PTHREAD_COND_INITILIAZER`. This is static initialization, when it is declared.
    - `int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *condattr)`. Dynamic initialization. The ID of the created condition variable is returned to the calling thread via the condition parameter `cond`. This method allows setting condition variable object attributes, `condattr`.

- Waiting and signaling on condition variables:
  - ✓ `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)`: Blocks the calling thread until the specified condition `cond` is signaled. This routine should be called while `mutex` is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, `mutex` will be automatically locked for use by the thread.
  - ✓ `int pthread_cond_signal (pthread_cond_t *cond)`: Used to signal (or wake up) another thread which is waiting on the condition variable `cond`. It should be called after `mutex` is locked and must unlock `mutex` in order for `pthread_cond_wait` routine to complete.
  - ✓ `int pthread_cond_broadcast (pthread_cond_t *cond)`: It should be used instead of `pthread_cond_signal` if more than one thread is in a blocking wait state.

- Free (destroy) a condition variable that is no longer needed:
  - ✓ `int pthread_cond_destroy (pthread_cond_t *cond);`

- **Example**: Three threads.
  - ✓ Two of them perform the work of updating the *count* variable. If the count reaches a specific value, it signals to the other threads. The third thread waits until the condition is signaled (*count* reaches a specific value) and then wakes up.

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12
// Source: https://computing.llnl.gov/tutorials/pthreads/
int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex; // mutex variable declaration
pthread_cond_t count_threshold_cv; // condition variable declaration

void *inc_count(void *t) { // thread start function
   int i;
   long my_id = (long)t;

   for (i=0; i<TCOUNT; i++) {
      pthread_mutex_lock(&count_mutex);
      count++;

      /* Check the value of count and signal waiting thread when condition is
```

```
      reached.  Note that this occurs while mutex is locked. */
      if (count == COUNT_LIMIT) {
         pthread_cond_signal(&count_threshold_cv);
         printf("inc_count(): thread %ld, count = %d  Threshold reached.\n", my_id, count);
      }
      printf("inc_count(): thread %ld, count = %d, unlocking mutex\n", my_id, count);
      pthread_mutex_unlock (&count_mutex);
      sleep(1); // Do some "work" so threads can alternate on mutex lock
   }
   pthread_exit(NULL);
}

void *watch_count(void *t) { // thread start function
   long my_id = (long)t;

   printf("Starting watch_count(): thread %ld\n", my_id);

   /* Lock mutex and wait for signal. The pthread_cond_wait routine will automatically
   and atomically unlock mutex while it waits. Also, note that if COUNT_LIMIT is reached
   before this routine is run by the waiting thread, the loop will be skipped to prevent
   pthread_cond_wait from never returning. */
   pthread_mutex_lock(&count_mutex);
   while (count < COUNT_LIMIT) {
     pthread_cond_wait (&count_threshold_cv, &count_mutex);
     printf("watch_count(): thread %ld Condition signal received.\n", my_id);
     count += 125;
     printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
   }
   printf("watch_count(): thread %ld Unlocking mutex\n", my_id);
   pthread_mutex_unlock(&count_mutex);
   pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
   int i;
   long t1=1, t2=2, t3=3;
   pthread_t threads[3];
   pthread_attr_t attr;

   pthread_mutex_init (&count_mutex, NULL); // mutex: dynamic initialization
   pthread_cond_init (&count_threshold_cv, NULL); // condition variable: dynamic initialization

   /* For portability, explicitly create threads in a joinable state */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   pthread_create(&threads[0], &attr, watch_count, (void *)t1);
   pthread_create(&threads[1], &attr, inc_count, (void *)t2);
   pthread_create(&threads[2], &attr, inc_count, (void *)t3);

   /* Wait for all threads to complete */
   for (i=0; i<NUM_THREADS; i++) { pthread_join(threads[i], NULL); }
   printf ("main(): Waited and joined with %d threads\n", NUM_THREADS);
   printf ("main(): Final value of count = % d. Done\n", count);

   /* Clean up and exit */
   pthread_attr_destroy(&attr);
   pthread_mutex_destroy (&count_mutex);
   pthread_cond_destroy (&count_threshold_cv);
   pthread_exit(NULL);
}
```

✓ Fig. 9 shows the two functions `watch_count` and `int_count` depicting where the mutex lock and unlock occurs.

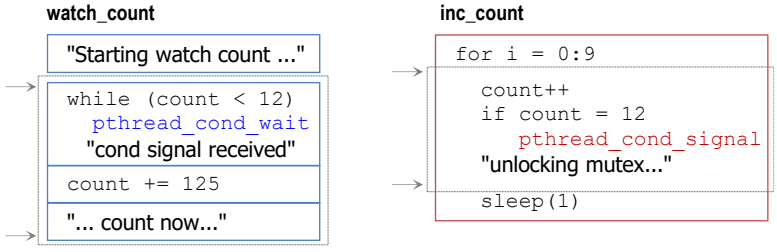

Figure 9. Functions representing the operations of each thread (watch_count: t1, inc_count: t2, t3). Note the mutex lock/unlock. Also note which function issues the signaling and which is waiting for the signal (when the condition is met)

✓ Fig. 10 depicts an execution sample. Note how t1 (the one that has a blocking wait) unlocks the mutex while waiting for the condition event. When that event is met, the mutex in t1 is locked and execution continues normally.

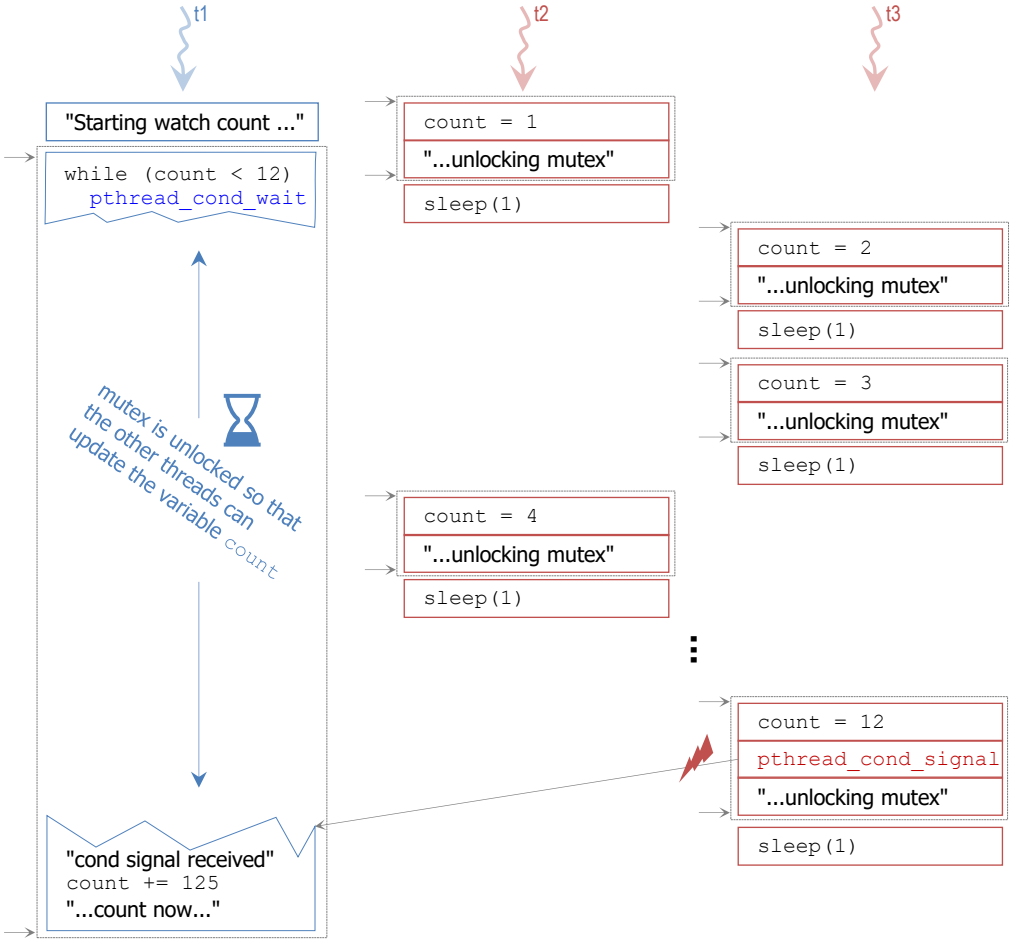

Figure 10. Sample execution of the program. t1 calls pthread_cond_wait and releases the lock while waiting for the other threads to update count. When count reaching the specified value (12), t2 (or t3) issues the condition signal to wake up t1. t1 then locks the mutex and continues normal execution.

✓ Note that even after the condition signal is received, the program might still execute some *count* updating a few more times. This is because the `sleep(1)` instruction in `int_count` allows the other thread (t2 or t3) to execute. Example of execution output:

```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
watch_count(): thread 1 Count= 1. Going into wait...
inc_count(): thread 3, count = 2, unlocking mutex
inc_count(): thread 2, count = 3, unlocking mutex
inc_count(): thread 3, count = 4, unlocking mutex
inc_count(): thread 2, count = 5, unlocking mutex
inc_count(): thread 3, count = 6, unlocking mutex
inc_count(): thread 2, count = 7, unlocking mutex
inc_count(): thread 3, count = 8, unlocking mutex
inc_count(): thread 2, count = 9, unlocking mutex
inc_count(): thread 3, count = 10, unlocking mutex
inc_count(): thread 2, count = 11, unlocking mutex
inc_count(): thread 3, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 3, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received. Count= 12
watch_count(): thread 1 Updating the value of count...
watch_count(): thread 1 count now = 137.
watch_count(): thread 1 Unlocking mutex.
inc_count(): thread 2, count = 138, unlocking mutex
inc_count(): thread 3, count = 139, unlocking mutex
inc_count(): thread 2, count = 140, unlocking mutex
inc_count(): thread 3, count = 141, unlocking mutex
inc_count(): thread 2, count = 142, unlocking mutex
inc_count(): thread 3, count = 143, unlocking mutex
inc_count(): thread 2, count = 144, unlocking mutex
```

```
inc_count(): thread 3, count = 145, unlocking mutex
Main(): Waited and joined with 3 threads. Final value of count = 145. Done.
```

COMPILATION (LINUX)

- To compile programs that include the pthreads library, use the modifier `-lpthread` when invoking `gcc`.
- Examples (dot product program):
  - ✓ `gcc -Wall dotprod.c -o dotprod -lpthread:` It will compile the `dotprod.c` file and generate an output file named `dotprod`. The `-Wall` option will check for all kinds of warnings like unused variables (this is good practice).
    - To execute the output file, use `./dotprod` ↵