POSIX THREADS PROGRAMMING

Dr. Lavika Goel

What is a Thread?

- Thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system
- Its a "procedure" that runs independently from its main program
- Multithreaded Program
  - Imagine a main program that contains a number of procedures
  - Imagine all of these procedures being able to be scheduled to run simultaneously or independently

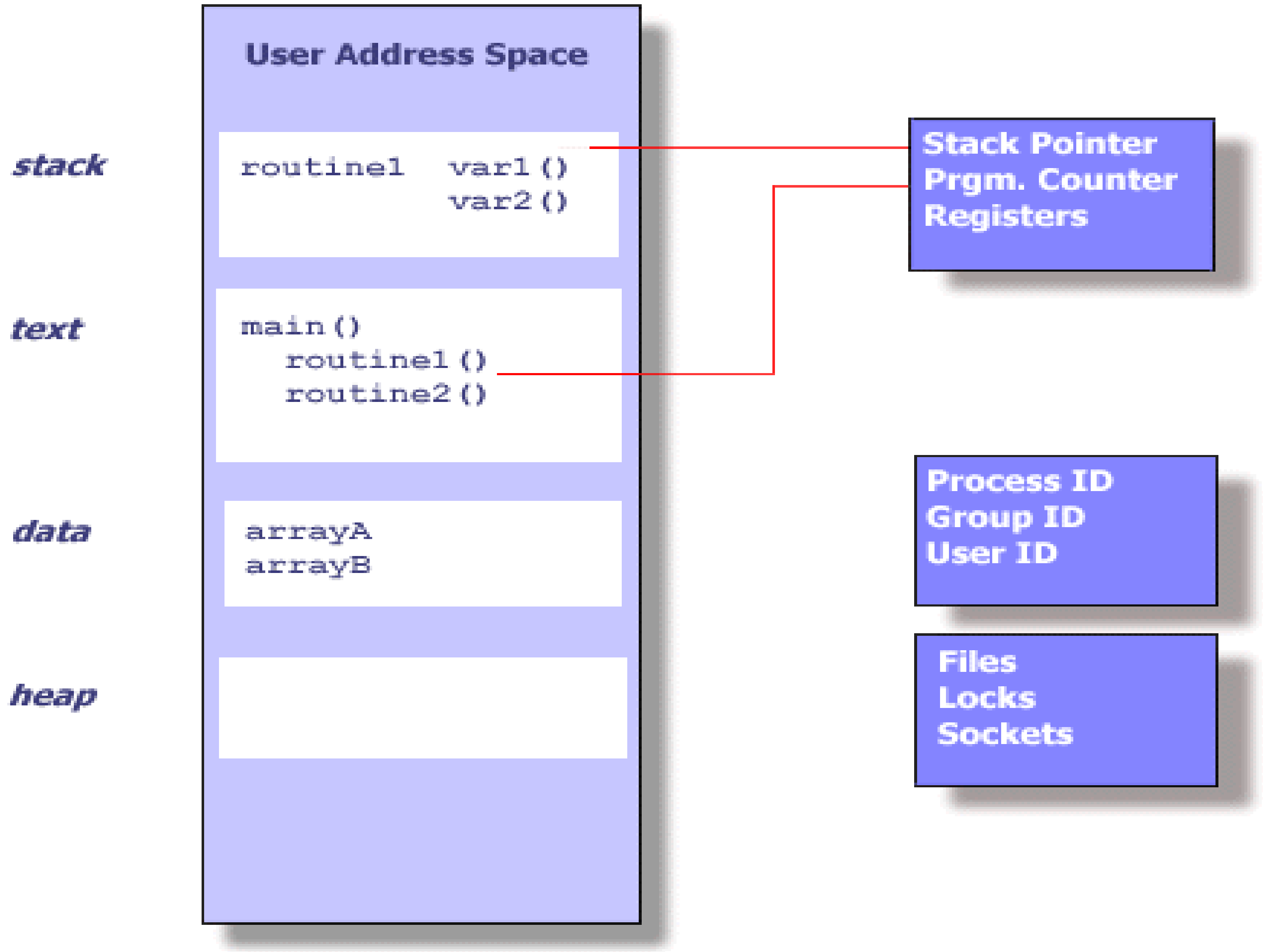
Process

- A process is created by the operating system, and requires a fair amount of "overhead".
- Processes contain information about program resources and program execution state
- Process Information:
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Working directory
  - Program instructions

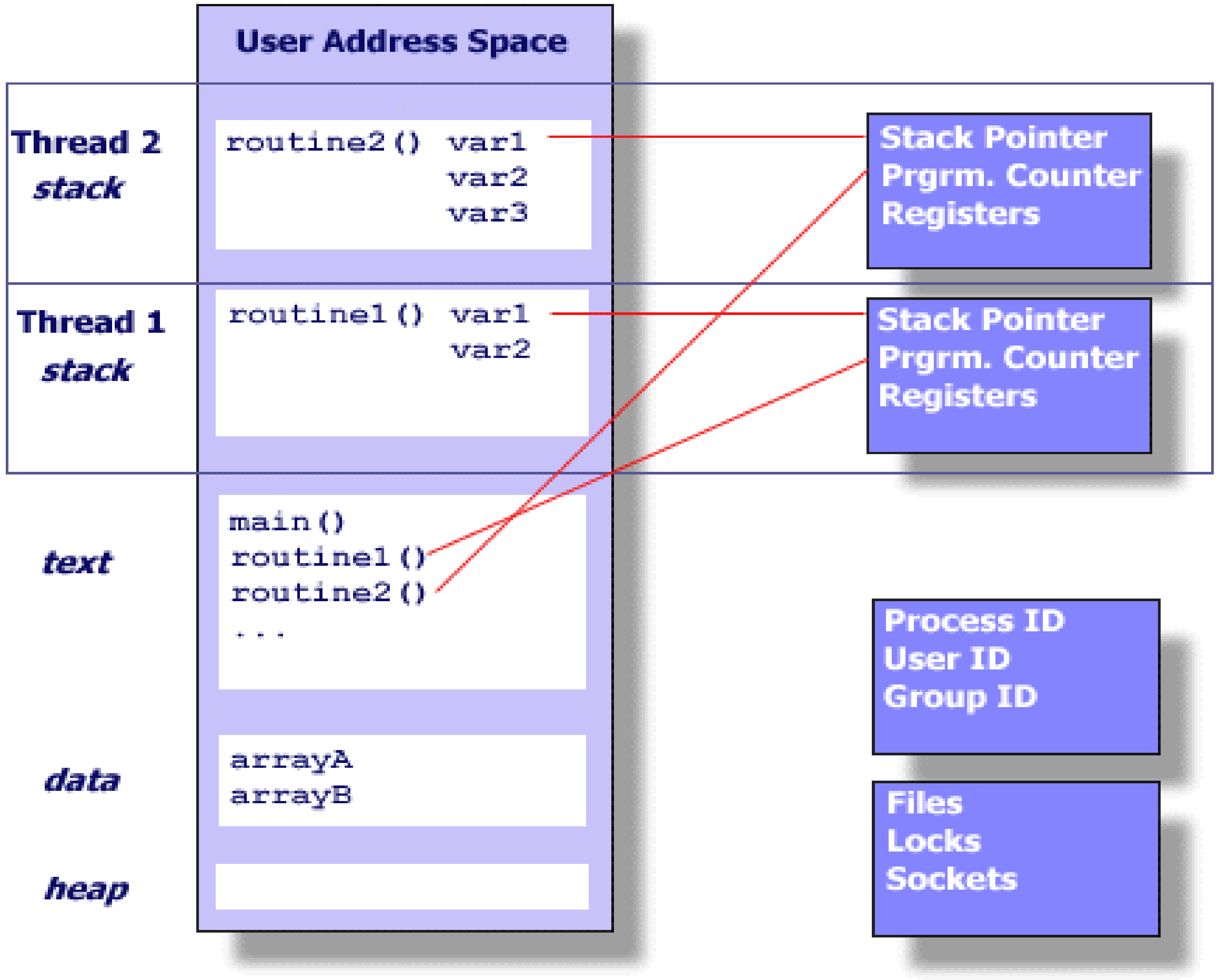
Process

- Process Information:
  - Registers, Stack, Heap
  - File descriptors
  - Signal actions
  - Shared libraries
  - Inter-process communication tools
    - message queues, pipes,
    - semaphores, or shared memory

Unix Process



Threads in a Unix Process



Thread Control

- Threads use and exist within the process resources
- Threads are able to be scheduled by the OS and run as independent entities
- Threads duplicate only the bare essential resources that enable them to exist as executable code
- Thread maintains its own Stack pointer, Registers, Scheduling properties, Set of pending and blocked signals and Thread specific data.

Threads in UNIX

- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable
- May share the process resources with other threads that act equally independently (and dependently)
- Dies if the parent process dies - or something similar
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

What are Pthreads?

- Hardware vendors have implemented their own proprietary versions of threads.
- These implementations differed substantially from each other making portable threaded applications difficult to develop.
- Standardized programming interface was required to take full advantage of the capabilities provided by threads
- IEEE POSIX 1003.1c standard (1995) - Portable Operating System Interface (POSIX)

Pthreads

- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h** header / include file and a thread library
- This library may be part of another library, such as **libc**, in some implementations.

Some useful links:

- standards.ieee.org/findstds/standard/1003.1-2008.html
- www.opengroup.org/austin/papers/posix_faq.html
- www.unix.org/version3/ieee_std.html

Lightweight

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead.
- Managing threads requires fewer system resources than managing processes.
- For example, the following table compares timing results for the **fork()** subroutine and the **pthread_create()** subroutine.
- Timings reflect 50,000 process/thread creations, were performed with the **time** utility, and units are in seconds, no optimization flags.

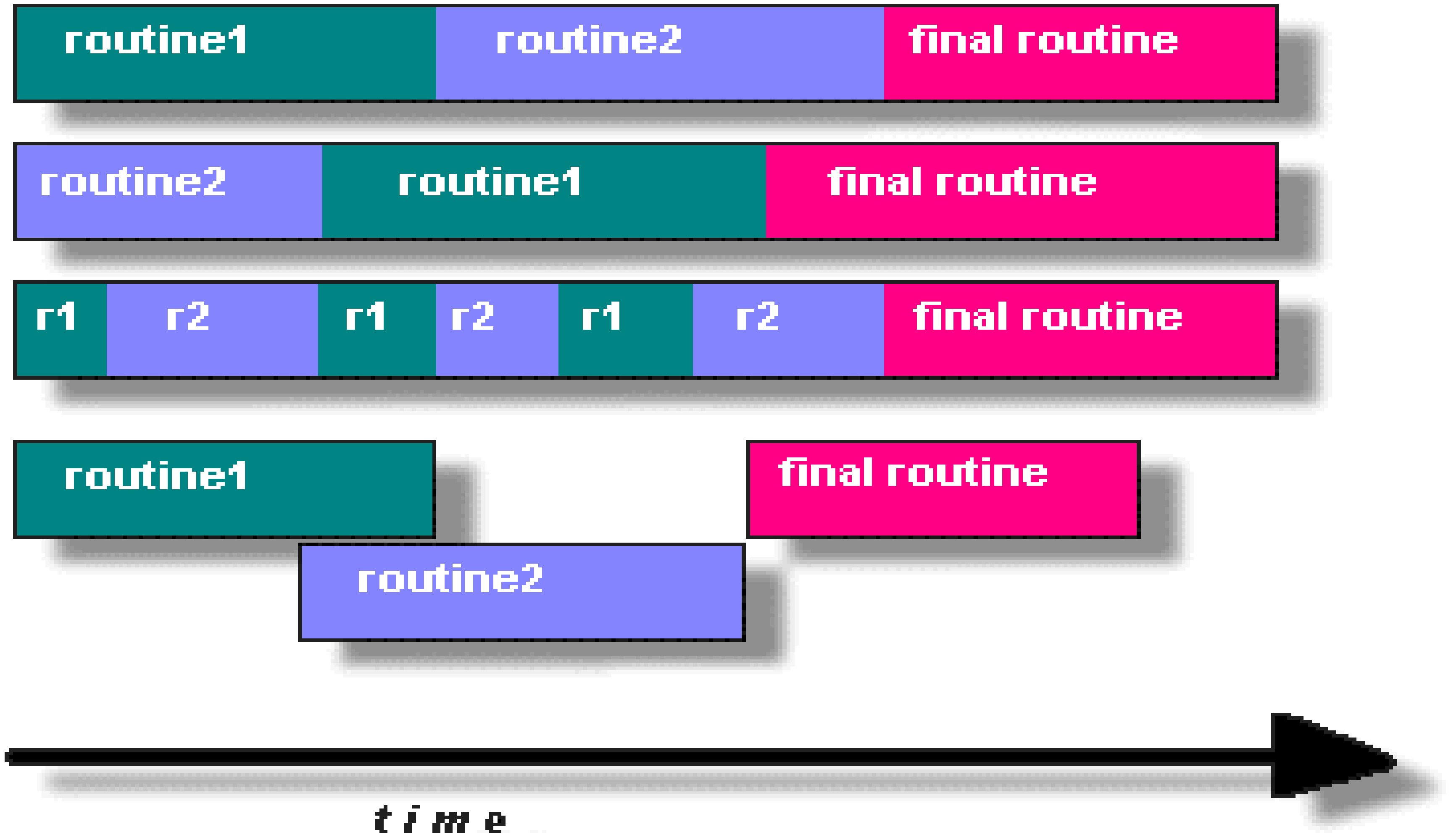
Lightweight

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Concurrent and Parallel Programming

- Problem partitioning
- Load balancing
- Communications
- Data dependencies
- Synchronization and race conditions
- Memory issues
- I/O issues
- Program complexity
- Programmer effort/costs/time

Concurrent (Parallel) Execution

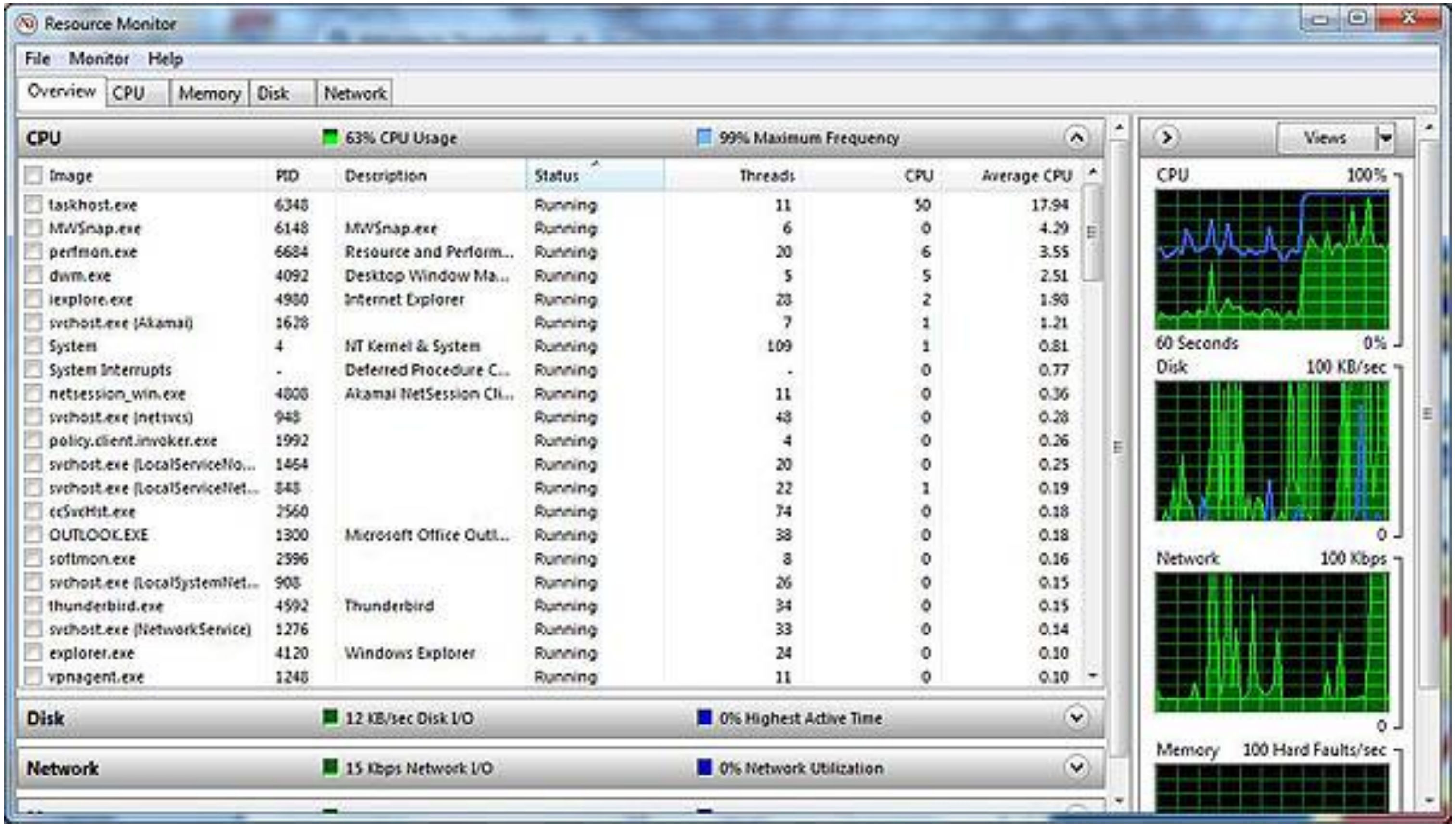


Programs well suited for pthreads

- Work that can be executed, or data that can be operated on, by multiple tasks simultaneously.
- Block for potentially long I/O waits
- Use many CPU cycles in some places but not others
- Must respond to asynchronous events
- Some work is more important than other work (priority interrupts)

Example

- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.
  - Another good example is a modern operating system, which makes extensive use of threads. A screenshot of the MS Windows OS and applications using threads is shown below.



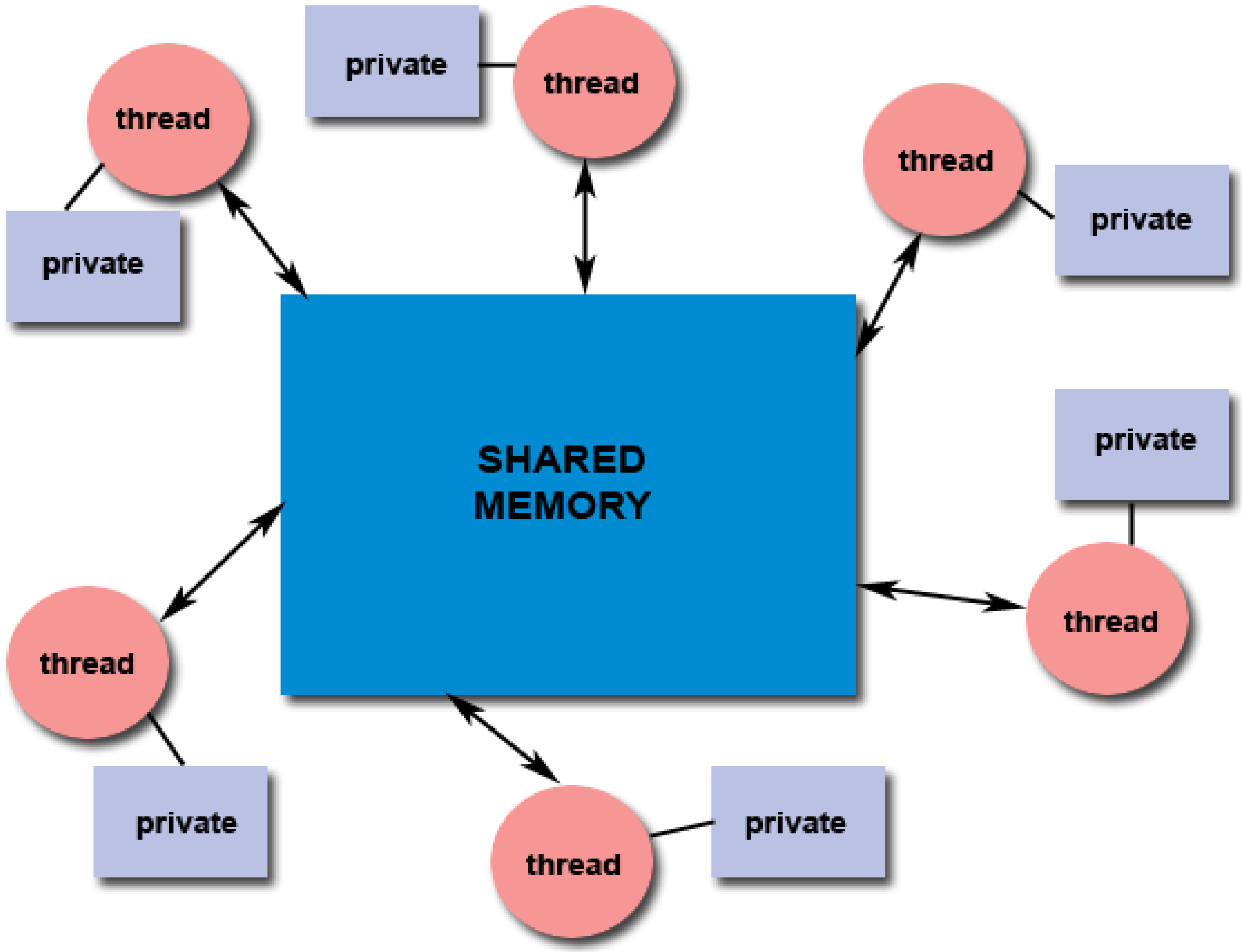
CPU 🟩 63% CPU Usage 🟦 99% Maximum Frequency

Image	PID	Description	Status	Threads	CPU	Average CPU
taskhost.exe	6348		Running	11	50	17.94
MWSnap.exe	6148	MWSnap.exe	Running	6	0	4.29
perfmon.exe	6684	Resource and Perform...	Running	20	6	3.55
dwm.exe	4092	Desktop Window Ma...	Running	5	5	2.51
iexplore.exe	4980	Internet Explorer	Running	28	2	1.98
svchost.exe (Akamai)	1628		Running	7	1	1.21
System	4	NT Kernel & System	Running	109	1	0.81
System Interrupts	-	Deferred Procedure C...	Running	-	0	0.77
netsession_win.exe	4808	Akamai NetSession Cli...	Running	11	0	0.36
svchost.exe (netsvcs)	948		Running	48	0	0.28
policy.client.invoker.exe	1992		Running	4	0	0.26
svchost.exe (LocalServiceNo...	1464		Running	20	0	0.25
svchost.exe (LocalServiceNet...	848		Running	22	1	0.19
ccSvcHst.exe	2560		Running	74	0	0.18
OUTLOOK.EXE	1300	Microsoft Office Outl...	Running	38	0	0.18
softmon.exe	2596		Running	8	0	0.16
svchost.exe (LocalSystemNet...	908		Running	26	0	0.15
thunderbird.exe	4592	Thunderbird	Running	34	0	0.15
svchost.exe (NetworkService)	1276		Running	33	0	0.14
explorer.exe	4120	Windows Explorer	Running	24	0	0.10
vpnagent.exe	1248		Running	11	0	0.10

Disk 🟩 12 KB/sec Disk I/O 🟦 0% Highest Active Time

Network 🟩 15 Kbps Network I/O 🟦 0% Network Utilization

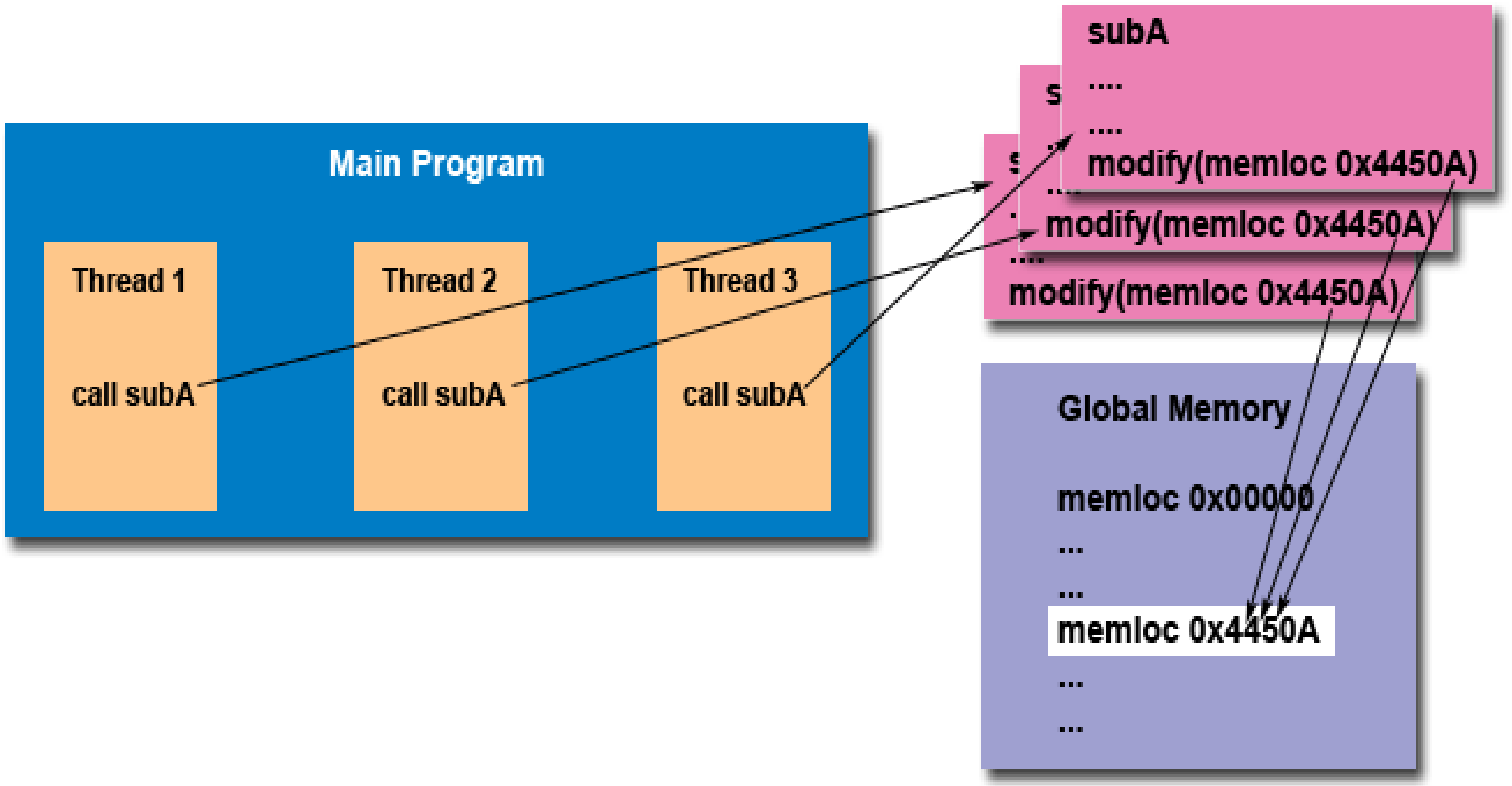
Shared Memory



Shared Memory Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.

Thread Safeness



Thread Safeness

- Thread-safeness refers to an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions
- If an application creates several threads, each of which makes a call to the same library routine:
  - This library routine modifies a global memory.
  - Each thread may try to modify this global memory location at the same time.
  - If the routine does not employ some sort of synchronization constructs, it is not thread-safe.

Pthreads API

- Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995
- Four major Categories
  - **Thread management**: Creating, Detaching, Joining
  - **Mutexes** (mutual exclusion) deals with synchronization: Mutex creation, destroying, locking, unlocking
  - **Condition variables**
  - **Synchronization**: Routines that manage read / write locks and barriers

Pthread Library

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Pthread Library

- The Pthreads API contains around 100 subroutines
- For portability, the **pthread.h** header file should be included in each source file using the Pthreads library.
- The current POSIX standard is defined only for the C language.
- Compile Commands - GNU C

```
gcc -pthread
```

Creating and Managing Threads

- pthread_create (thread, attr, start_routine, arg)
- pthread_exit (status)
- pthread_cancel (thread)
- pthread_attr_init (attr)
- pthread_attr_destroy (attr)

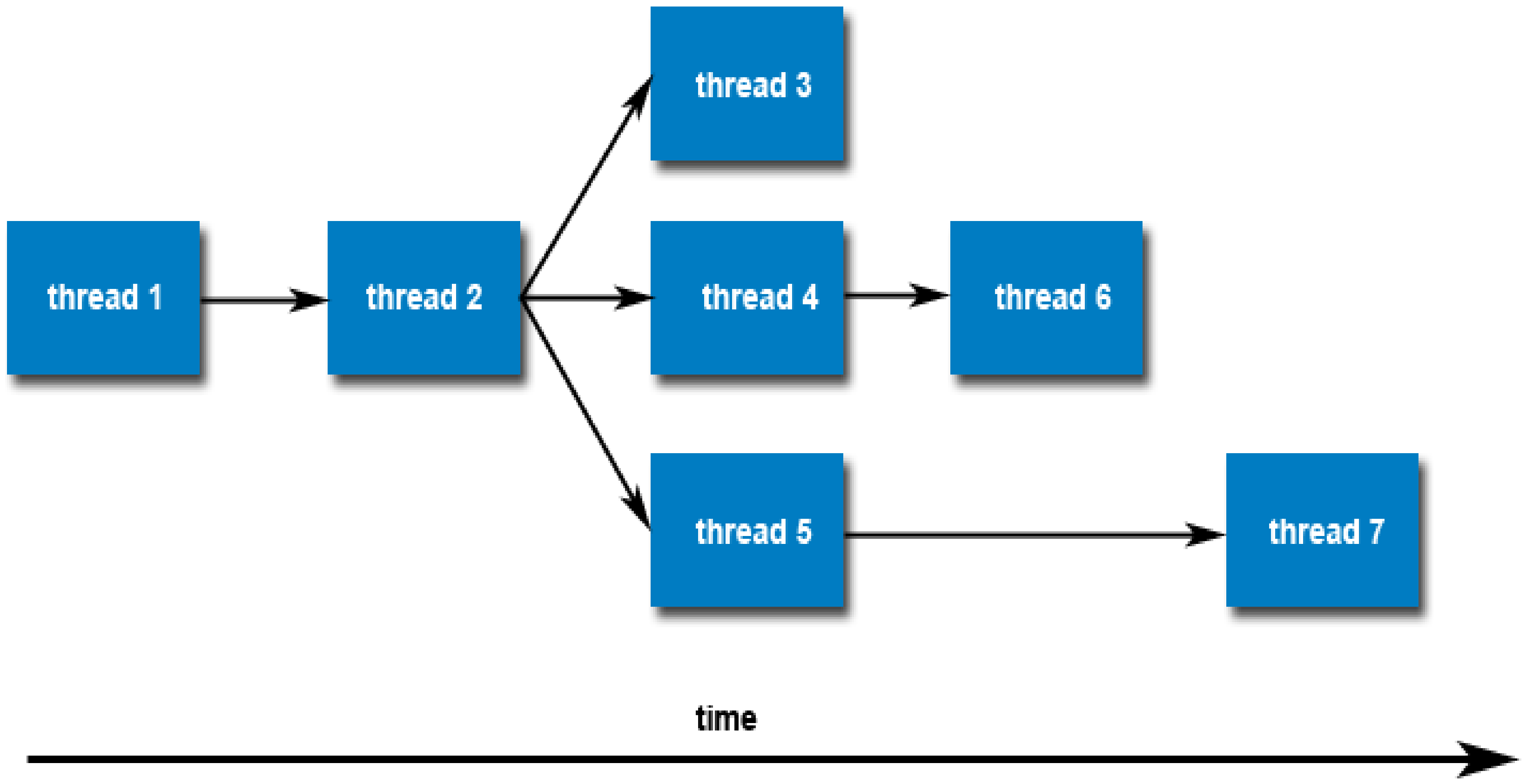
Creating Threads

- main() program comprises a single, default thread
- All other threads must be explicitly created by the programmer
- **`pthread_create`** creates a new thread and makes it executable
- This routine can be called any number of times from anywhere within your code
- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency between threads.

`pthread_create` – Arguments

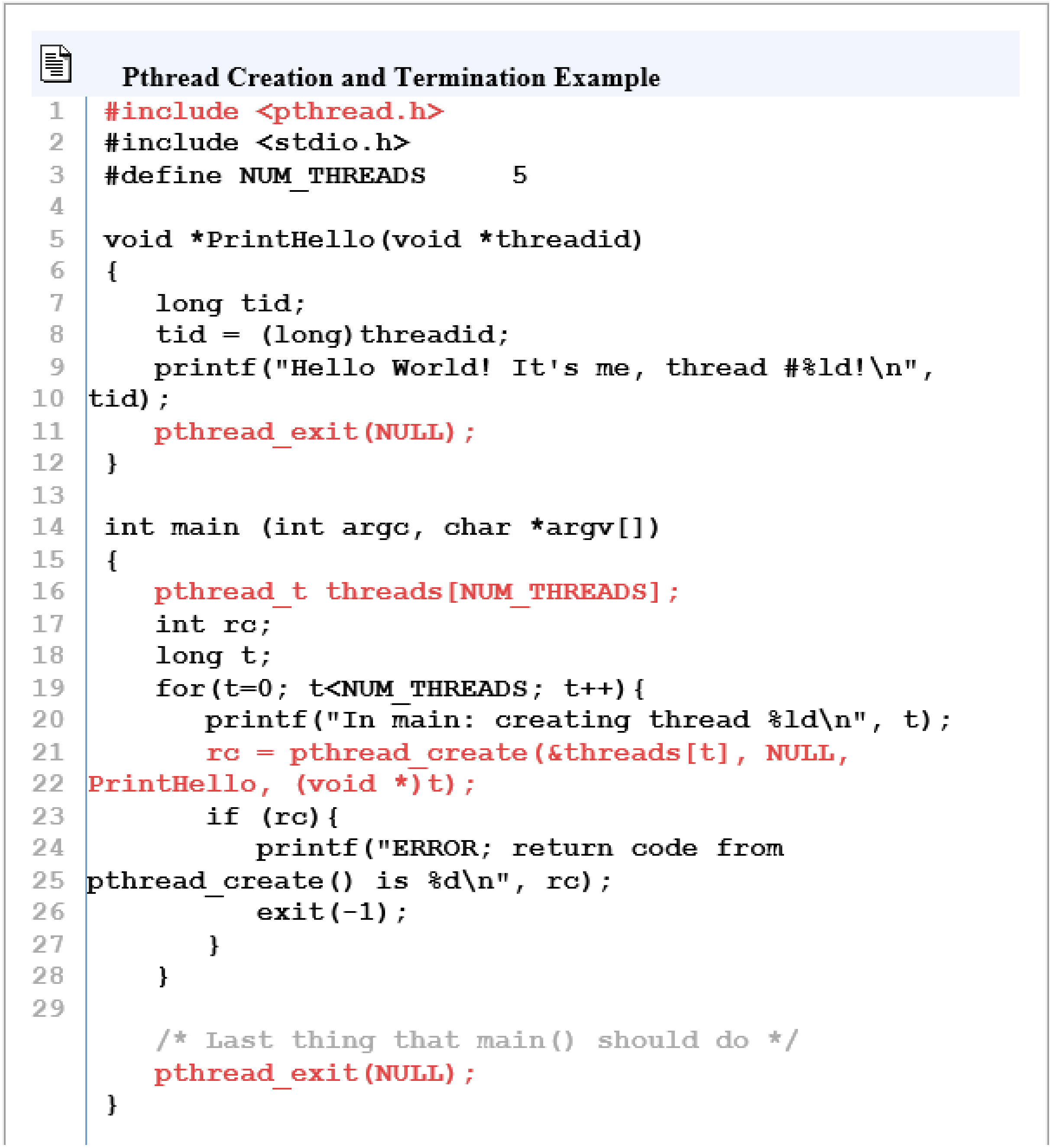
- thread: A unique identifier for the new thread returned by the subroutine.
- attr: An attribute object that may be used to set thread attributes. We can specify a thread attributes object, or NULL for the default values.
- start_routine: C routine that the thread will execute once it is created.
- arg: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed

Creating Threads



Thread creation and termination example

This simple example code creates 5 threads with the **`pthread_create()`** routine. Each thread prints a "Hello World!" message, and then terminates with a call to **`pthread_exit()`**.



Pthread Creation and Termination Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n",
tid);
   pthread_exit(NULL);
}

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);
      rc = pthread_create(&threads[t], NULL,
PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from
pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Thread Attributes

- Thread is created with certain default attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
- **`pthread_attr_init`** & **`pthread_attr_destroy`** are used to initialize / destroy the thread attribute object.
- Other routines are then used to query / set specific attributes in the thread attribute object
- Attributes include:
  - Detached or joinable state
  - Scheduling inheritance, policy, parameters, contention scope
  - Stack size, address, guard (overflow) size

Terminating Threads

- The thread returns normally from its starting routine. Its work is done.
- The thread makes a call to the **`pthread_exit`** subroutine - whether its work is done or not.
- The thread is canceled by another thread via the **`pthread_cancel`** routine.
- The entire process is terminated due to making a call to either the **`exec() or exit()`**
- If main() finishes first, without calling **`pthread_exit`** explicitly itself

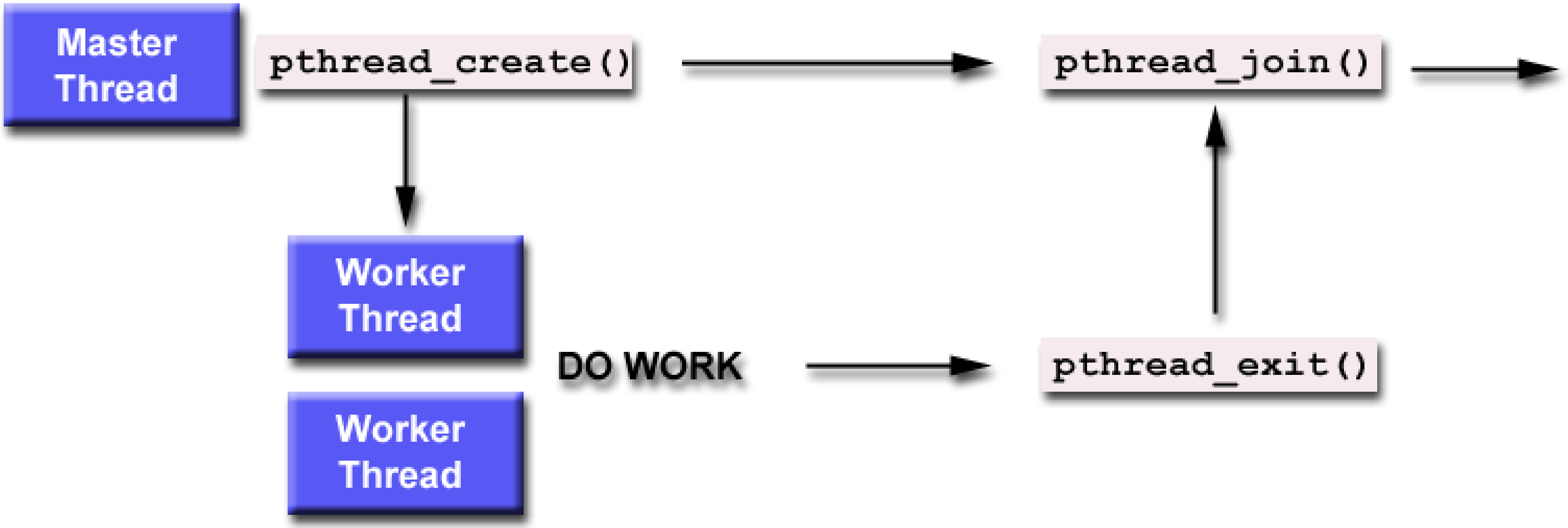
Terminating Threads

- The **`pthread_exit()`** routine allows the programmer to specify an optional termination *status* parameter
- This optional parameter is typically returned to threads "joining" the terminated thread
- In subroutines that execute to completion normally, calling **`pthread_exit()`** can often be dispensed with unless the optional status code needs to be passed back
- Cleanup: the **`pthread_exit()`** routine does not close files; any files opened inside the thread will remain open after the thread is terminated

Joining and Detaching Threads

- pthread_join (threadid, status)
- pthread_detach (threadid)
- pthread_attr_setdetachstate (attr, detachstate)
- pthread_attr_getdetachstate (attr, detachstate)

Joining


Master Thread
pthread_create()
pthread_join()
Worker Thread
Worker Thread
DO WORK
pthread_exit()

Joining

- "Joining" is one way to accomplish synchronization between threads
- The **`pthread_join()`** subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to **`pthread_exit()`**.
- A joining thread can match one **`pthread_join()`** call
- It is a logical error to attempt multiple joins on the same thread.

Joinable?

- 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

Joinable?

- Four Step Process:
  - Declare a pthread attribute variable of the pthread_attr_t data type
  - Initialize the attribute variable with **`pthread_attr_init()`**
  - Set the attribute detached status with **`pthread_attr_setdetachstate()`**
  - When done, free library resources used by the attribute with **`pthread_attr_destroy()`**

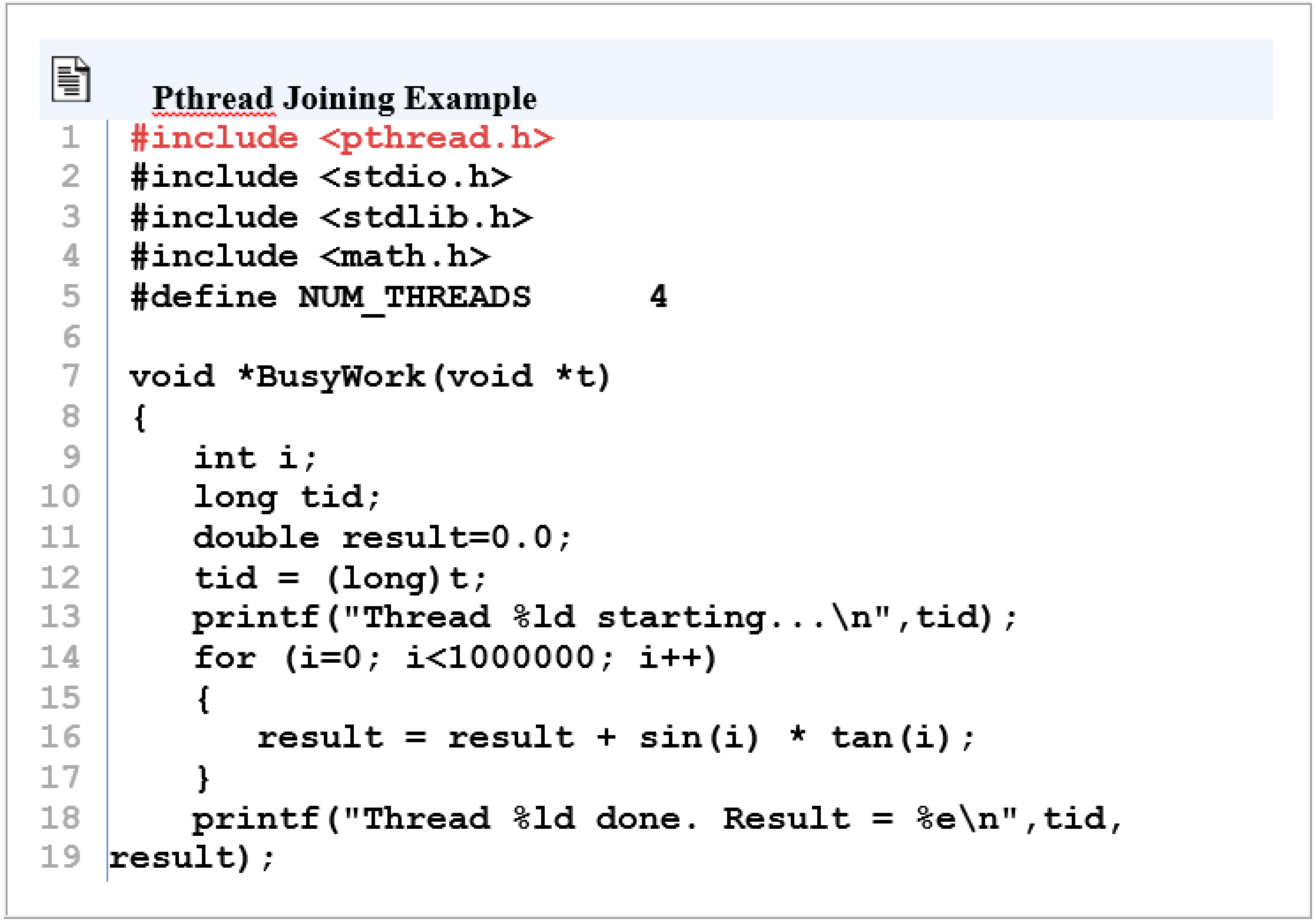
Detaching

- The **`pthread_detach()`** routine can be used to explicitly detach a thread even though it was created as joinable
- There is no converse routine

Example: Pthread Joining

- This example demonstrates how to "wait" for thread completions by using the Pthread join routine.
- Since some implementations of Pthreads may not create threads in a joinable state, the threads in this example are explicitly created in a joinable state so that they can be joined later.

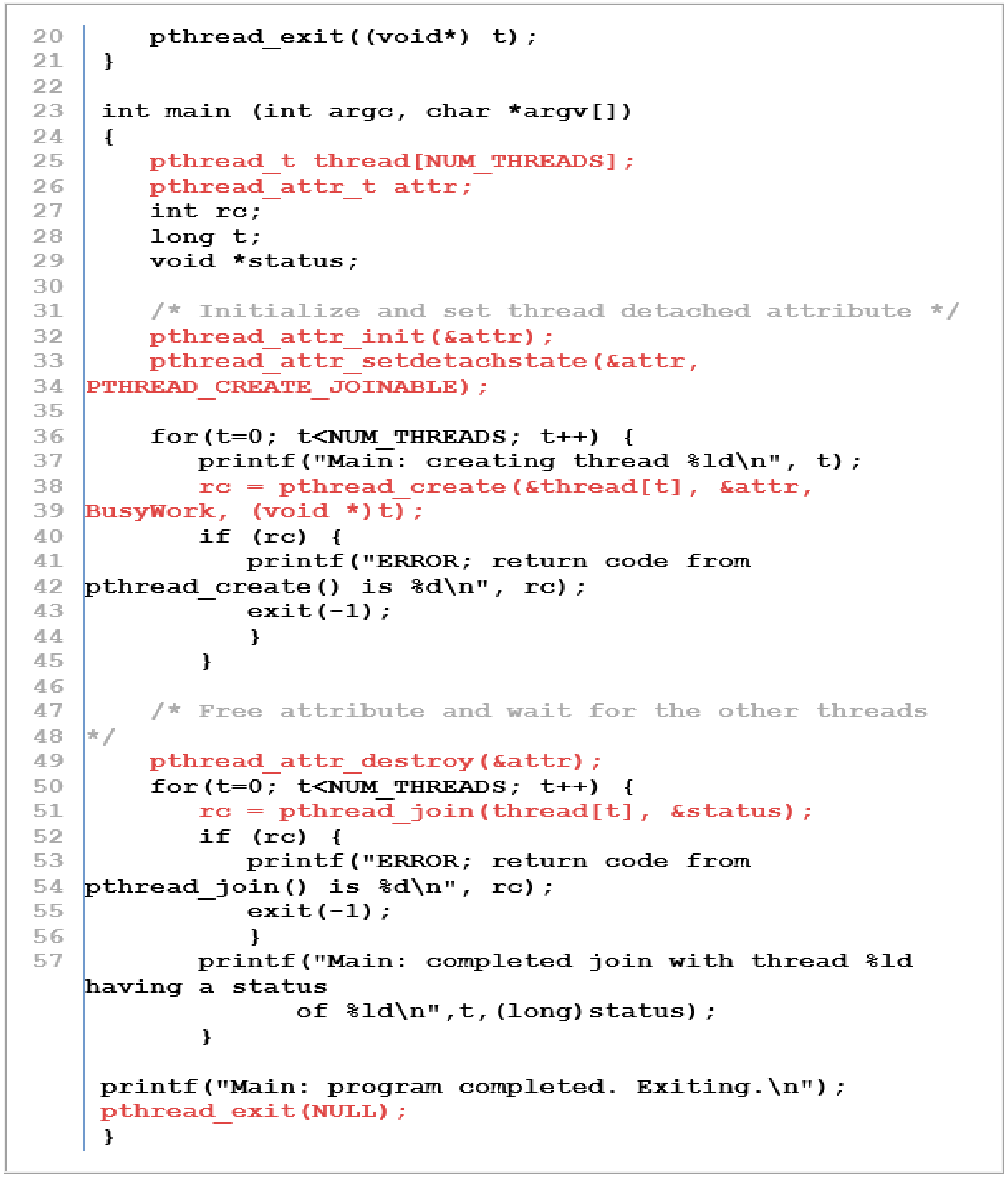
Example



Pthread Joining Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS     4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid,
result);
```

(Contd.)



```
    pthread_exit((void*) t);
 }

 int main (int argc, char *argv[])
 {
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    long t;
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);

    for(t=0; t<NUM_THREADS; t++) {
       printf("Main: creating thread %ld\n", t);
       rc = pthread_create(&thread[t], &attr,
BusyWork, (void *)t);
       if (rc) {
          printf("ERROR; return code from
pthread_create() is %d\n", rc);
          exit(-1);
          }
       }

    /* Free attribute and wait for the other threads
*/
    pthread_attr_destroy(&attr);
    for(t=0; t<NUM_THREADS; t++) {
       rc = pthread_join(thread[t], &status);
       if (rc) {
          printf("ERROR; return code from
pthread_join() is %d\n", rc);
          exit(-1);
          }
       printf("Main: completed join with thread %ld
having a status
             of %ld\n",t,(long)status);
       }

 printf("Main: program completed. Exiting.\n");
 pthread_exit(NULL);
 }
```

Mutex Variables

- Mutex is an abbreviation for "mutual exclusion".
- Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A mutex variable acts like a "lock" protecting access to a shared data resource.

Mutex Variables

- The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time.
- Even if several threads try to lock a mutex only one thread will be successful.
- No other thread can own that mutex until the owning thread unlocks that mutex.
- Threads must "take turns" accessing protected data.

Race Conditions

- Mutexes can be used to prevent "race" conditions.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Mutex

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Create and Destroy Mutex

- `pthread_mutex_init (mutex,attr)`
- `pthread_mutex_destroy (mutex)`
- `pthread_mutexattr_init (attr)`
- `pthread_mutexattr_destroy (attr)`

Create and Destroy Mutex

- 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:
  - Statically, when it is declared

    ```
    pthread_mutex_t mymutex =
    PTHREAD_MUTEX_INITIALIZER;
    ```
  - Dynamically, with the **`pthread_mutex_init()`** routine. This method permits setting mutex object attributes, *attr*
- The mutex is initially unlocked.

Create and Destroy Mutex

- The *attr* object is used to establish properties for the mutex object, and must be of type **`pthread_mutexattr_t`** if used (may be specified as NULL to accept defaults).
- The **`pthread_mutexattr_init()`** and **`pthread_mutexattr_destroy()`** routines are used to create and destroy mutex attribute objects respectively.
- **`pthread_mutex_destroy()`** should be used to free a mutex object which is no longer needed.

Lock and Unlock Mutex

- `pthread_mutex_lock (mutex)`
-
- `pthread_mutex_trylock (mutex)`

- `pthread_mutex_unlock (mutex)`

Lock and Unlock Mutex

- The **`pthread_mutex_lock()`** routine is used by a thread to acquire a lock on the *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()`** will lock a mutex.
- However, if the mutex is already locked, the routine will return immediately with a "busy" error code.
- This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

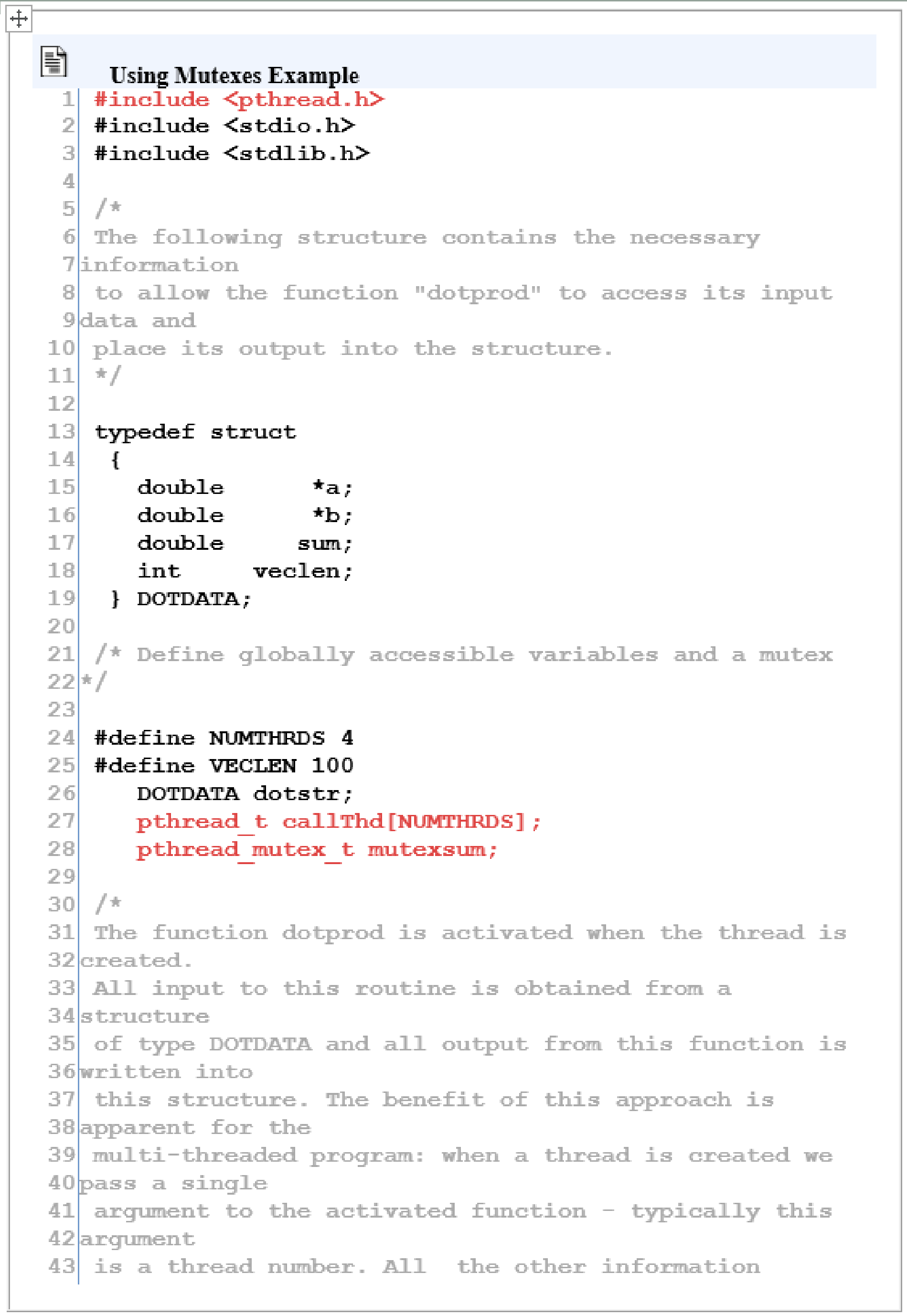
Lock and Unlock Mutex

- **`pthread_mutex_unlock()`** will unlock a mutex if called by the owning thread.
- Calling this routine is required after a thread has completed its use of protected data and other threads need access.
- An error will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread
- Mutexes are akin to a "gentlemen's agreement"
- It is up to the code writer to insure that the necessary threads all make the mutex lock and unlock calls correctly

Example: Using Mutexes

- This example program illustrates the use of mutex variables in a threads program that performs a dot product.
- The main data is made available to all threads through a globally accessible structure.
- Each thread works on a different part of the data.
- The main thread waits for all the threads to complete their computations, and then it prints the resulting sum.

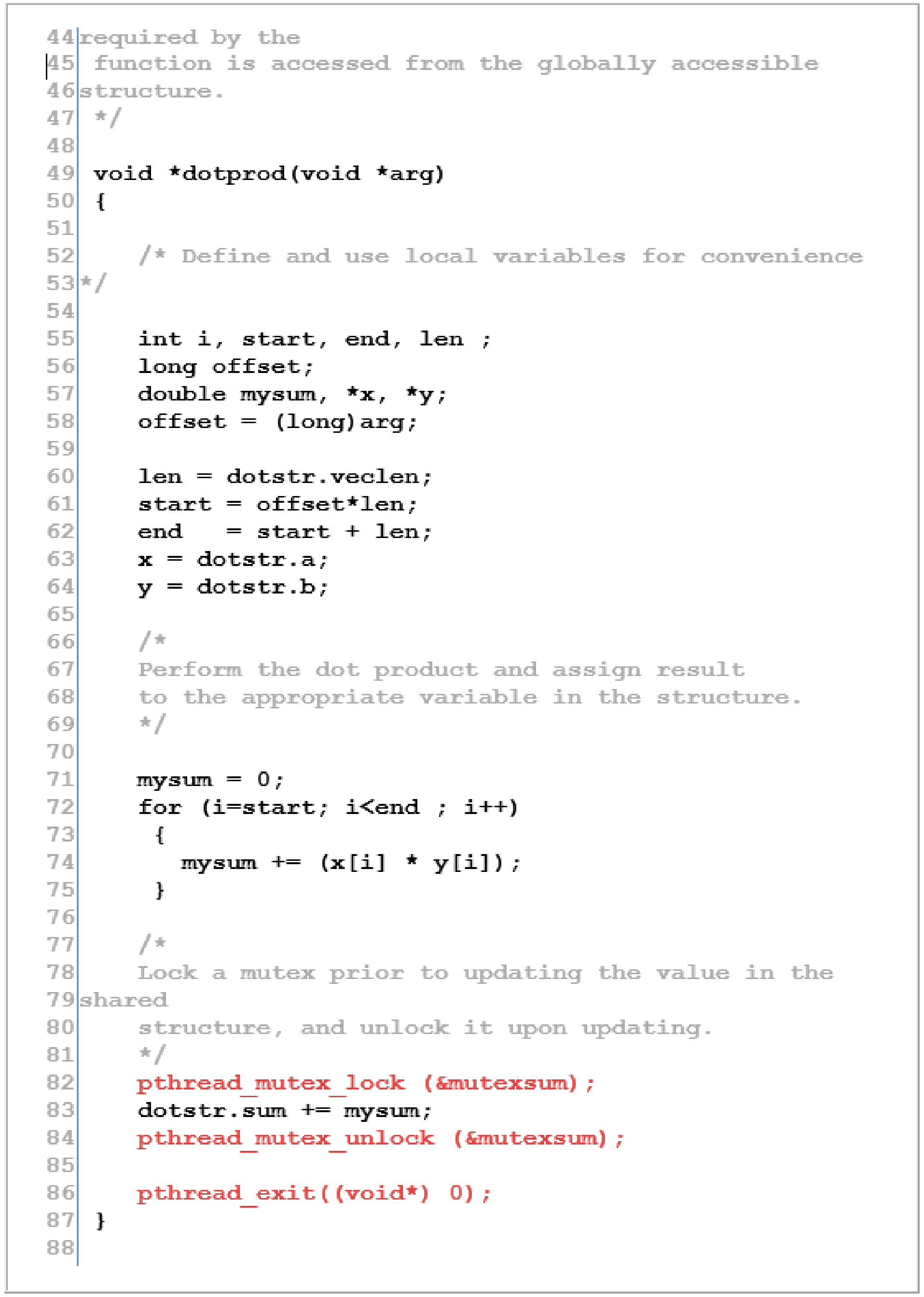
Example



Using Mutexes Example

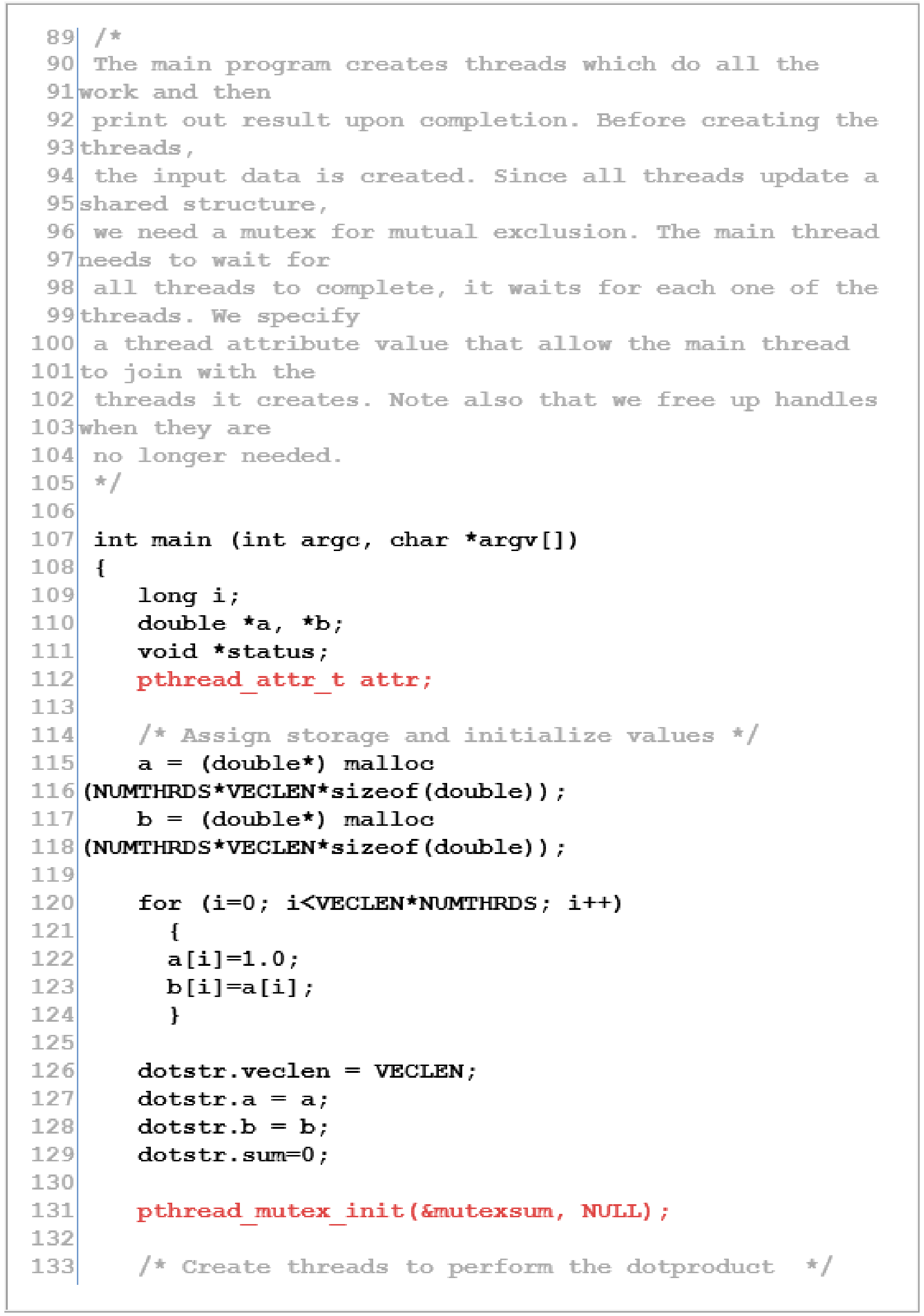
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
The following structure contains the necessary
information
to allow the function "dotprod" to access its input
data and
place its output into the structure.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;

/* Define globally accessible variables and a mutex
*/

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

/*
The function dotprod is activated when the thread is
created.
All input to this routine is obtained from a
structure
of type DOTDATA and all output from this function is
written into
this structure. The benefit of this approach is
apparent for the
multi-threaded program: when a thread is created we
pass a single
argument to the activated function - typically this
argument
is a thread number. All  the other information
```

(Contd. - 1)



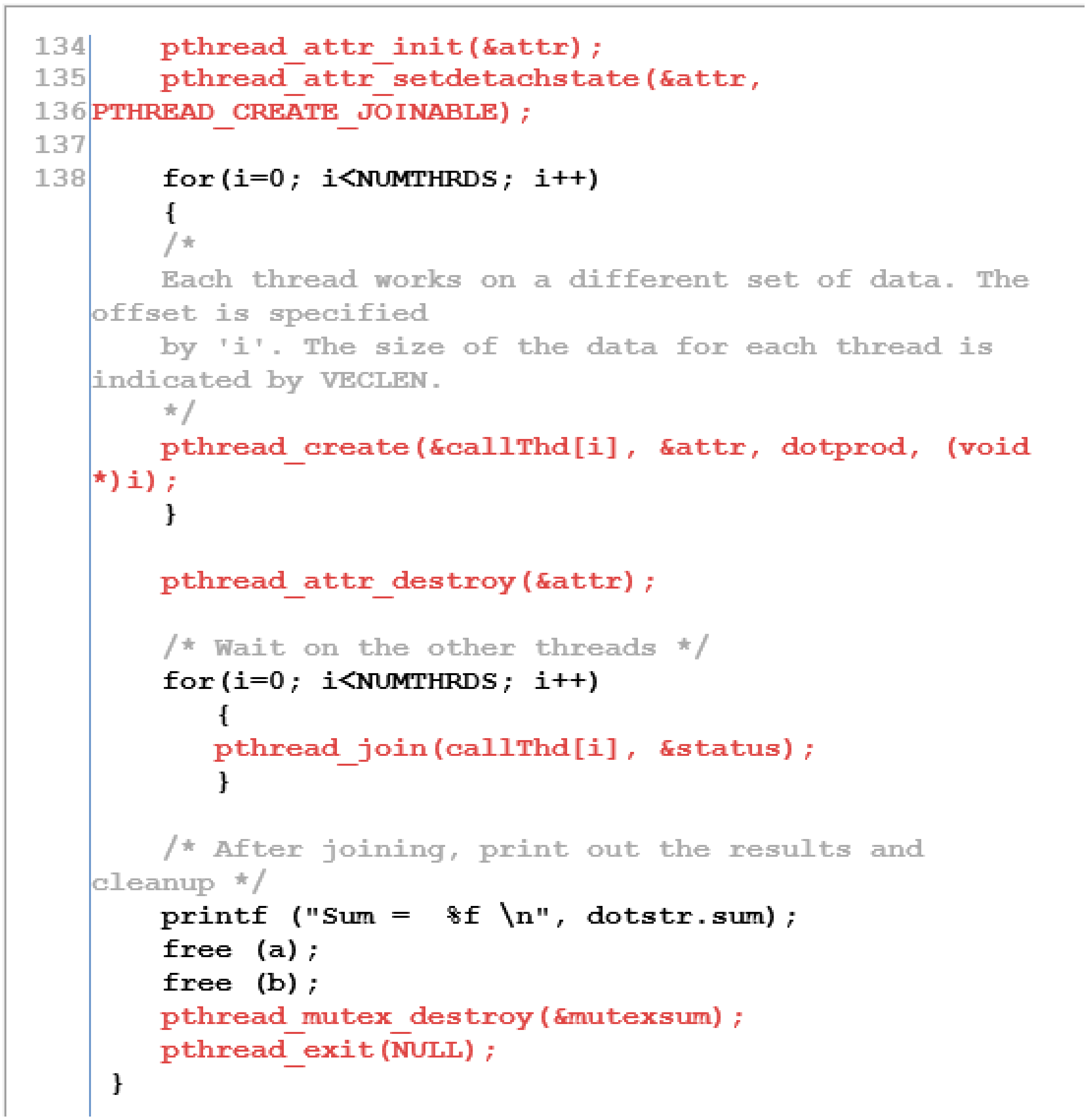
```
required by the
 function is accessed from the globally accessible
structure.
 */

 void *dotprod(void *arg)
 {

    /* Define and use local variables for convenience
*/

    int i, start, end, len ;
    long offset;
    double mysum, *x, *y;
    offset = (long)arg;

    len = dotstr.veclen;
    start = offset*len;
    end   = start + len;
    x = dotstr.a;
    y = dotstr.b;

    /*
    Perform the dot product and assign result
    to the appropriate variable in the structure.
    */

    mysum = 0;
    for (i=start; i<end ; i++)
     {
       mysum += (x[i] * y[i]);
     }

    /*
    Lock a mutex prior to updating the value in the
shared
    structure, and unlock it upon updating.
    */
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
 }
```

(Contd. – 2)



```
/*
The main program creates threads which do all the
work and then
print out result upon completion. Before creating the
threads,
the input data is created. Since all threads update a
shared structure,
we need a mutex for mutual exclusion. The main thread
needs to wait for
all threads to complete, it waits for each one of the
threads. We specify
a thread attribute value that allow the main thread
to join with the
threads it creates. Note also that we free up handles
when they are
no longer needed.
*/

int main (int argc, char *argv[])
{
   long i;
   double *a, *b;
   void *status;
   pthread_attr_t attr;

   /* Assign storage and initialize values */
   a = (double*) malloc
(NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc
(NUMTHRDS*VECLEN*sizeof(double));

   for (i=0; i<VECLEN*NUMTHRDS; i++)
     {
     a[i]=1.0;
     b[i]=a[i];
     }

   dotstr.veclen = VECLEN;
   dotstr.a = a;
   dotstr.b = b;
   dotstr.sum=0;

   pthread_mutex_init(&mutexsum, NULL);

   /* Create threads to perform the dotproduct  */
```

(Contd. – 3)



```
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);

    for(i=0; i<NUMTHRDS; i++)
    {
    /*
    Each thread works on a different set of data. The
offset is specified
    by 'i'. The size of the data for each thread is
indicated by VECLEN.
    */
    pthread_create(&callThd[i], &attr, dotprod, (void
*)i);
    }

    pthread_attr_destroy(&attr);

    /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++)
       {
       pthread_join(callThd[i], &status);
       }

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

Semaphore

- The pthreads library itself does not provide a semaphore
- However, a separate POSIX standard does define them
- The necessary declarations to use these semaphores are contained in semaphore.h
- To define a semaphore object, use

```
    sem_t sem_name;
```

Semaphore API

- **`sem_init`**: Initialize a new semaphore
  - The second argument denotes *how* the semaphore will be shared.
  - Passing zero denotes that it will be shared among **threads** rather than processes.
  - The final argument is the initial value of the semaphore.
- **`sem_destroy`**: Deallocate an existing semaphore.
- **`sem_wait`**: This is the Wait () operation.
- **`sem_post`**: This is the Signal () operation.

sem_init

- `int sem_init (sem_t *sem, int pshared, unsigned int value);`
  - `sem` address of the declared semaphore
  - `pshared` should be 0 (not shared with threads in other processes)
  - `value` the desired initial value of the semaphore

- Return: The return value is 0 if successful.

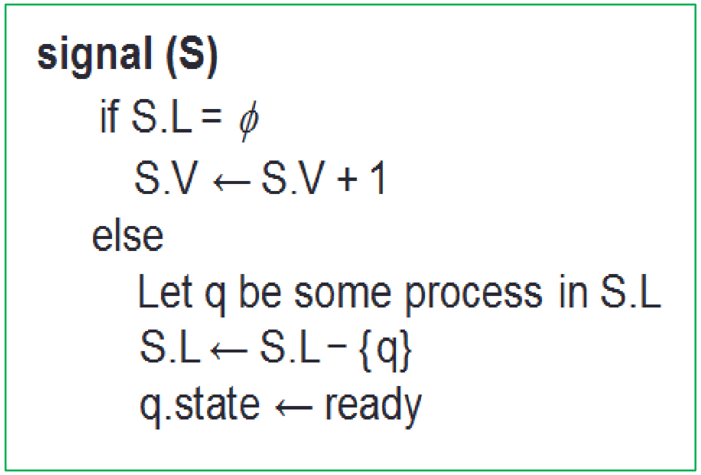
sem_wait

- To wait on a semaphore
- `int sem_wait (sem_t *sem);`
- If the value of the semaphore is negative, the calling process blocks
- One of the blocked processes wakes up when another process calls `sem_post`

```
wait (S)
    if S.V > 0
        $S.V \leftarrow S.V - 1$
    else
        $S.L \leftarrow append(S.L, p)$
        $p.state \leftarrow blocked$
```

sem_post

- To increment the value of a semaphore
- `int sem_post(sem_t *sem);`
- It increments the value of the semaphore
- It wakes up a blocked process waiting on the semaphore, if any.



```
signal (S)
    if S.L = $\phi$
        $S.V \leftarrow S.V + 1$
    else
        Let q be some process in S.L
        $S.L \leftarrow S.L - \{q\}$
        $q.state \leftarrow ready$
```

sem_getvalue

- To find out the value of a semaphore
- `int sem_getvalue (sem_t *sem, int *valp);`
- Gets the current value of sem
- Places it in the location pointed to by valp

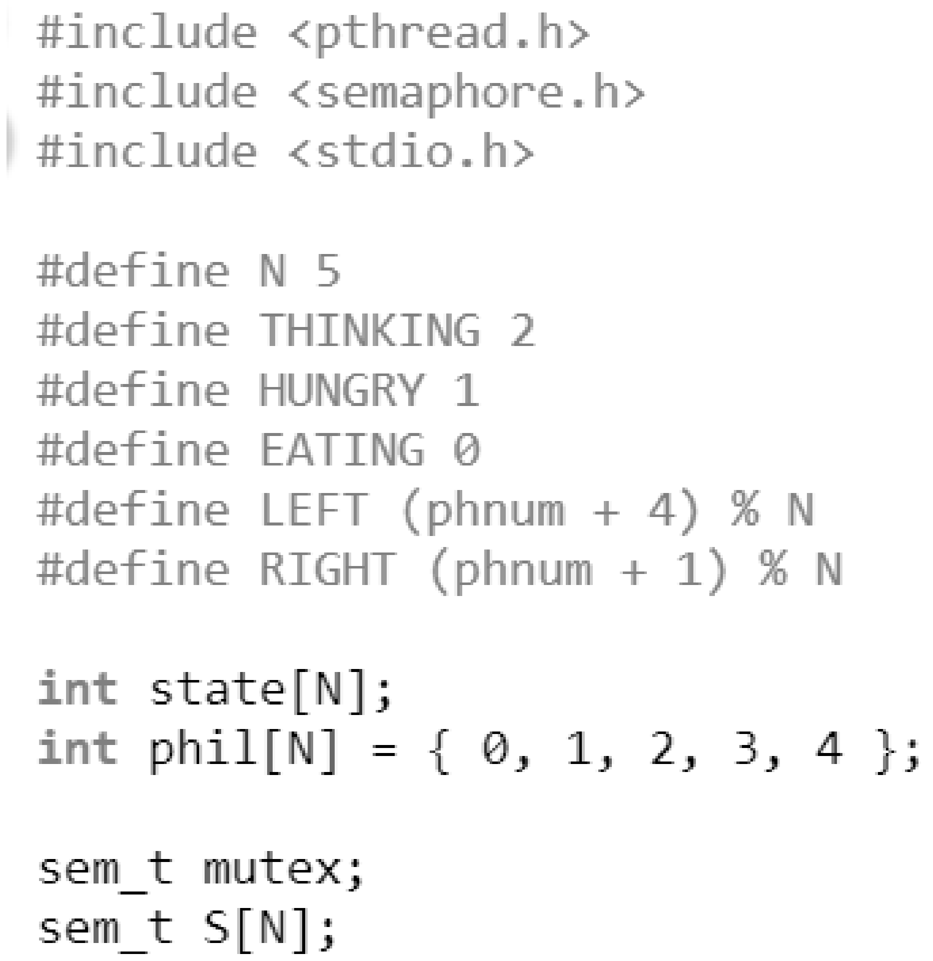
sem_destroy

- To destroy a semaphore
- `sem_destroy(sem_t *sem);`
- Destroys the semaphore
- No threads should be waiting on the semaphore if its destruction is to succeed.

Comparison

State	Pthread	Mutex	Semaphore
Creation	`pthread_create`	`pthread_mutex_init`	`sem_init`
Destroy	`pthread_exit`	`pthread_mutex_destroy`	`sem_destroy`
Waiting	`pthread_join`	-	-
Acquisition	-	`pthread_mutex_lock`	`sem_wait`
Release	-	`pthread_mutex_unlock`	`sem_post`

Example code: Dining Philosopher Problem Using Semaphores

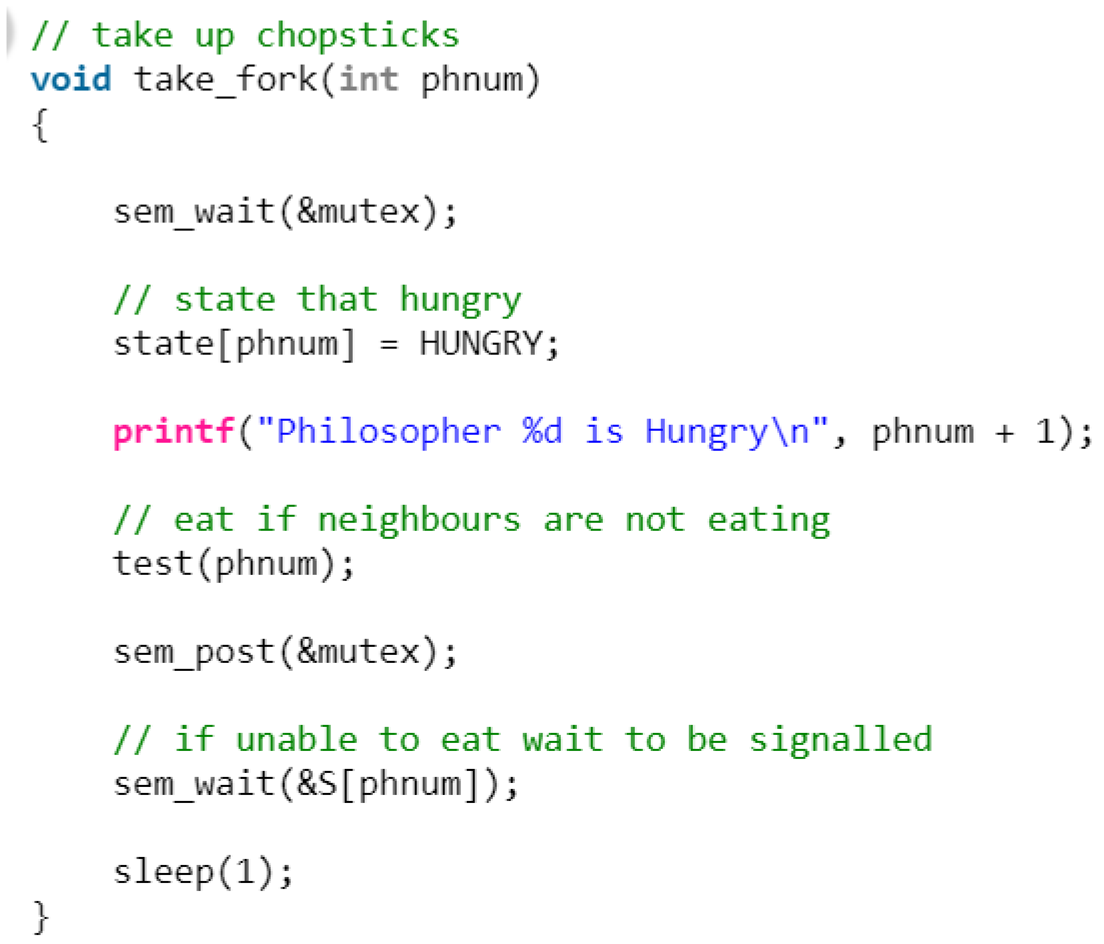


```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];
```

(Contd.-1)

```c
void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n",
                      phnum + 1, LEFT + 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);

        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}
```

(Contd.-2)



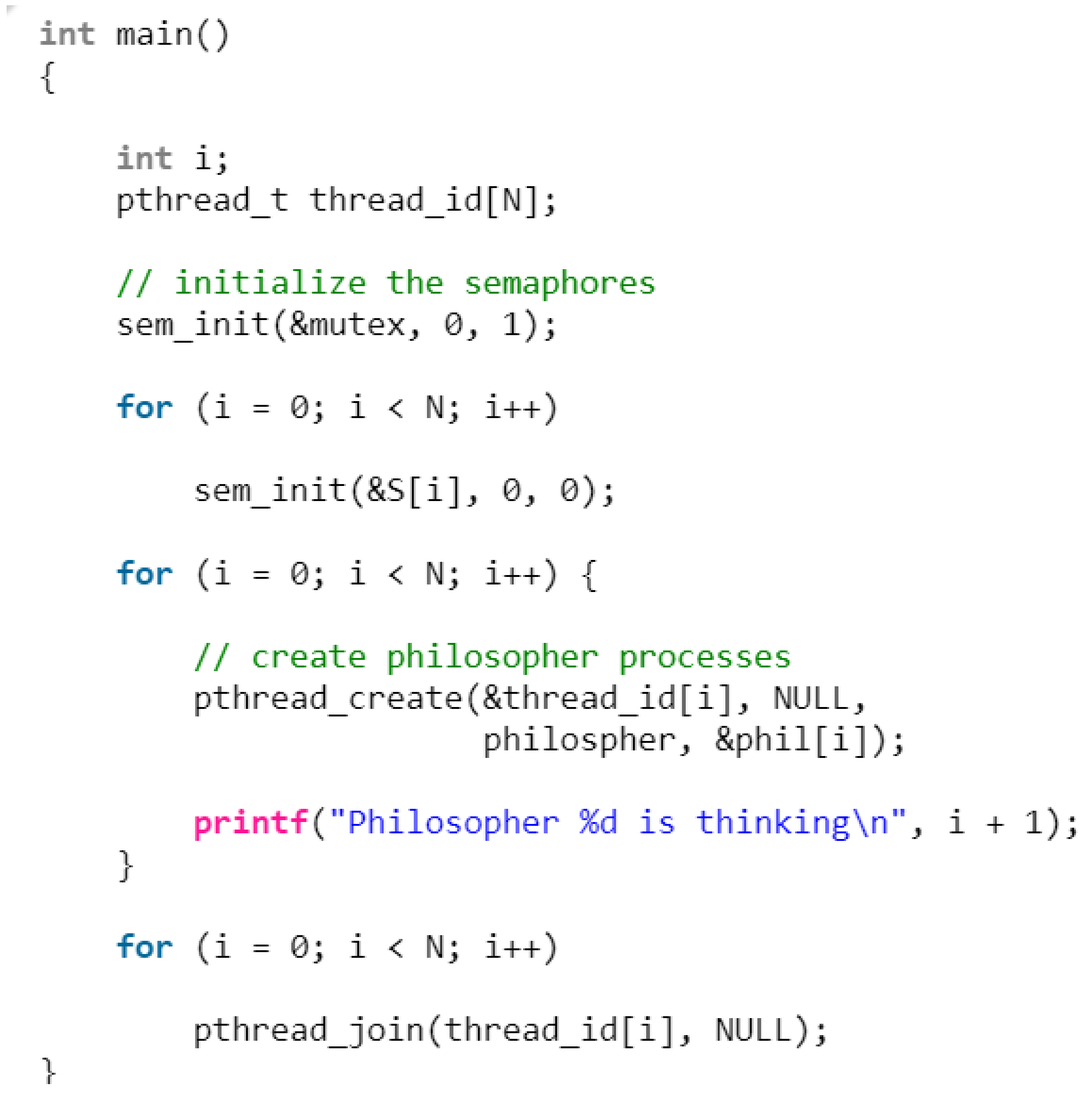
```
// take up chopsticks
void take_fork(int phnum)
{

    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);

    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}
```

(Contd.-3)

```
// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}
```

```
void* philospher(void* num)
{

    while (1) {

        int* i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);
    }
}
```

(Contd.- 4)



```
int main()
{

    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)

        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {

        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                       philospher, &phil[i]);

        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)

        pthread_join(thread_id[i], NULL);
}
```

Condition Variables

- Condition variables provide yet another way for threads to synchronize.
- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.

Creating and Destroying Condition Variables

- `pthread_cond_init (condition,attr)`
- `pthread_cond_destroy (condition)`
- `pthread_condattr_init (attr)`
- `pthread_condattr_destroy (attr)`

Creating and Destroying Condition Variables

- Condition variables must be declared with type **pthread_cond_t**, and must be initialized before they can be used. There are two ways to initialize a condition variable:
  - Statically, when it is declared. For example: **pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;**
  - Dynamically, with the **pthread_cond_init()** routine. The ID of the created condition variable is returned to the calling thread through the *condition* parameter.

Creating and Destroying Condition Variables

- The optional *attr* object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes.
- The attribute object, if used, must be of type **pthread_condattr_t** (may be specified as NULL to accept defaults).
- The **pthread_condattr_init()** and **pthread_condattr_destroy()** routines are used to create and destroy condition variable attribute objects.
- **pthread_cond_destroy()** should be used to free a condition variable that is no longer needed.

Waiting and Signaling on Condition Variables

- `pthread_cond_wait (condition, mutex)`
- `pthread_cond_signal (condition)`
- `pthread_cond_broadcast (condition)`

Waiting and Signaling on Condition Variables

- **pthread_cond_wait()** blocks the calling thread until the specified *condition* is signalled. 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.

Waiting and Signaling on Condition Variables

- The **pthread_cond_signal()** routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after *mutex* is locked, and must unlock *mutex* in order for **pthread_cond_wait()** routine to complete.
- The **pthread_cond_broadcast()** routine should be used instead of **pthread_cond_signal()** if more than one thread is in a blocking wait state.
- It is a logical error to call **pthread_cond_signal()** before calling **pthread_cond_wait()**.

Waiting and Signaling on Condition Variables

- Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:
- Failing to lock the mutex before calling **pthread_cond_wait()** may cause it NOT to block.
- Failing to unlock the mutex after calling **pthread_cond_signal()** may not allow a matching **pthread_cond_wait()** routine to complete (it will remain blocked).

A representative sequence for using condition variables is shown here:

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, 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 Thread-A.
- Unlock mutex.
- Continue

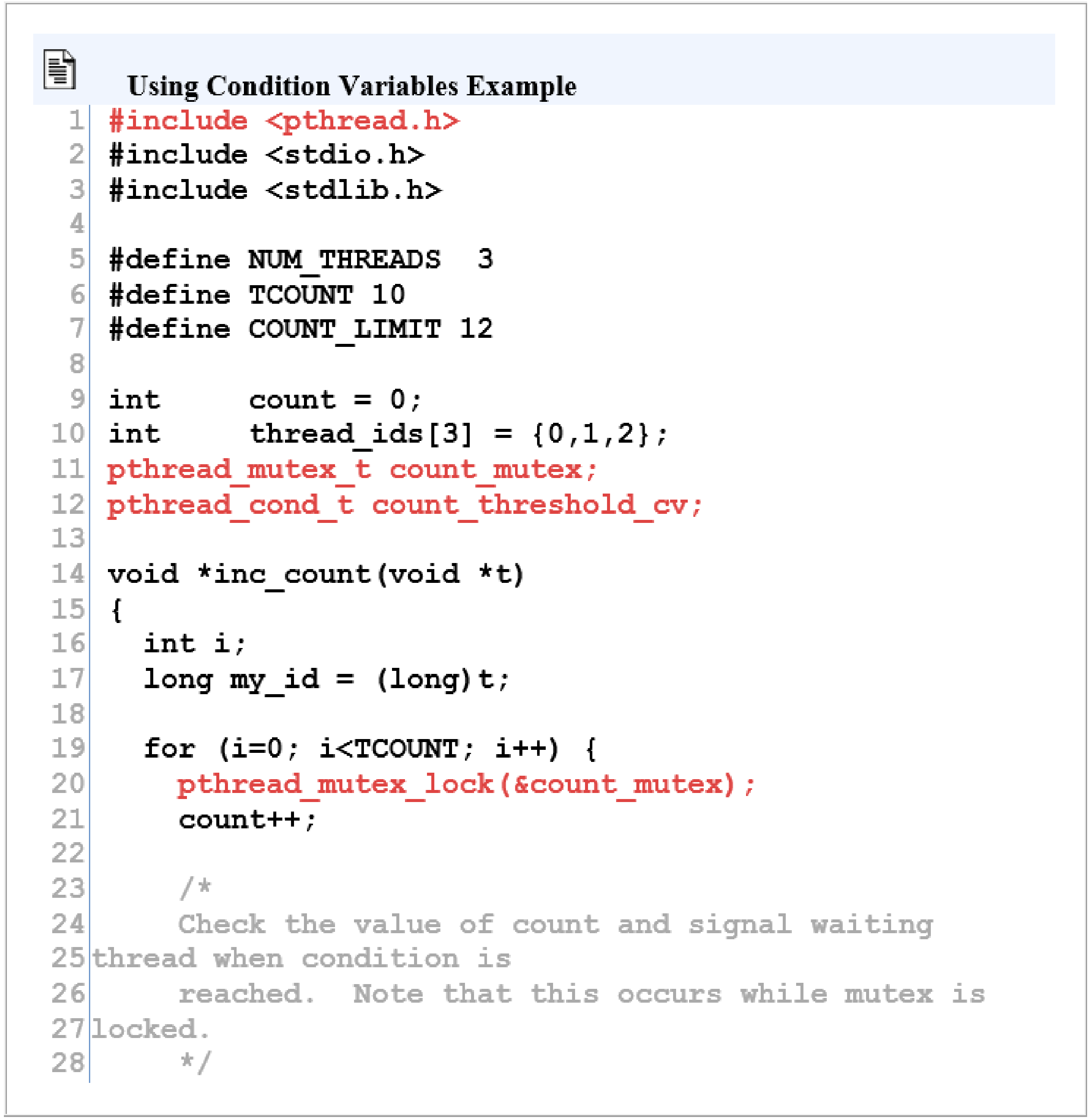
Main Thread

Join / Continue

Example: Using Condition Variables

- This simple example code demonstrates the use of several Pthread condition variable routines.
- The main routine creates three threads.
- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

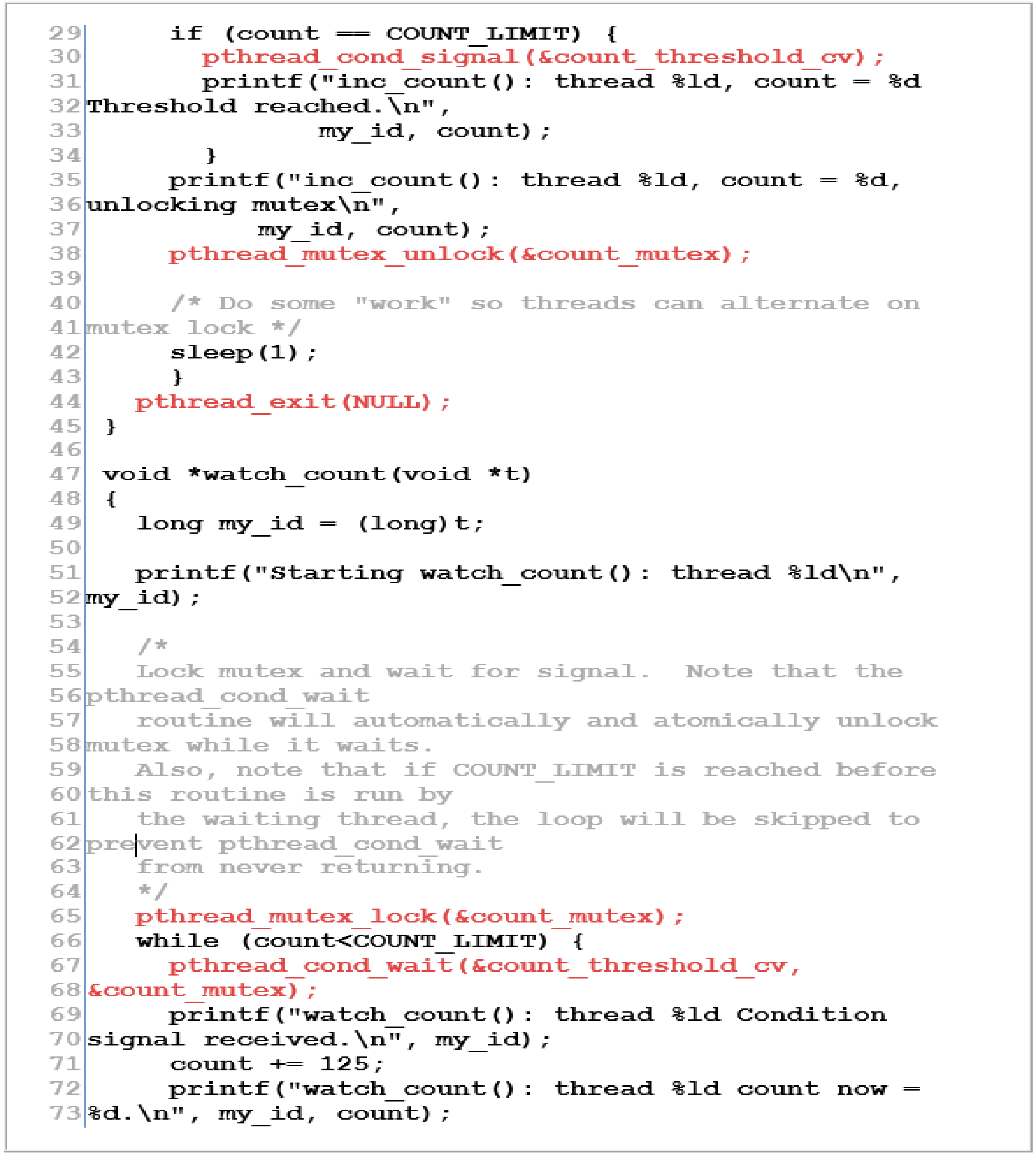
Example



Using Condition Variables Example

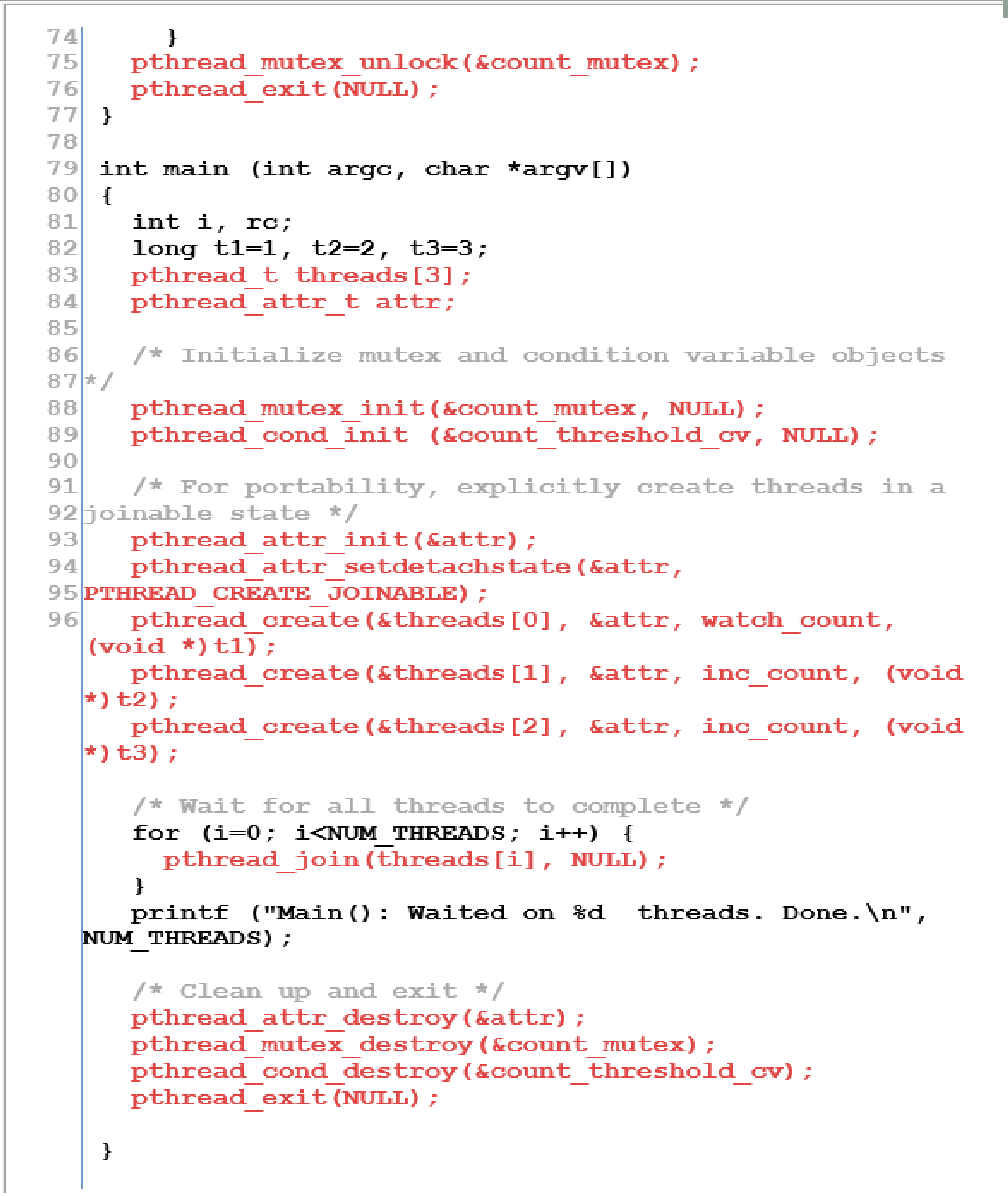
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  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.
    */
```

(Contd.)



```
     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);

     /* Do some "work" so threads can alternate on
mutex lock */
     sleep(1);
     }
   pthread_exit(NULL);
 }

 void *watch_count(void *t)
 {
   long my_id = (long)t;

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

   /*
   Lock mutex and wait for signal.  Note that 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);
```

(Contd.)



```
      }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
 }

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

    /* Initialize mutex and condition variable objects
*/
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init (&count_threshold_cv, NULL);

    /* 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 on %d  threads. Done.\n",
NUM_THREADS);

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

 }
```