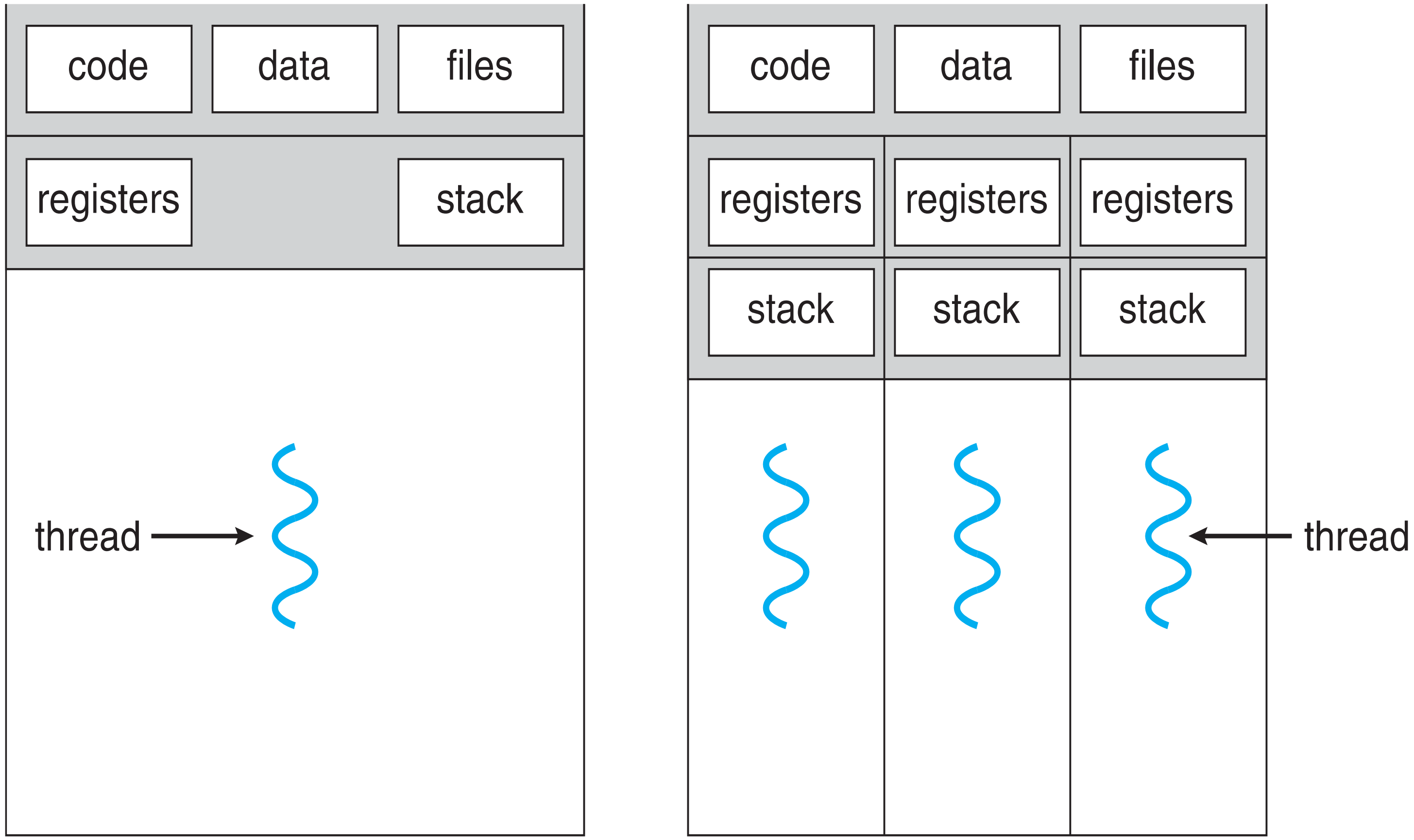
Threads

Threads

- A thread is a basic unit of CPU utilization;
- It comprises a thread ID, a program counter (PC), a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals

```c
#include<stdio.h>
int main()
{
  while(1)
  {
    printf("Hello...\n");
  }
  return 0;
}
```

Check the entry in process list...



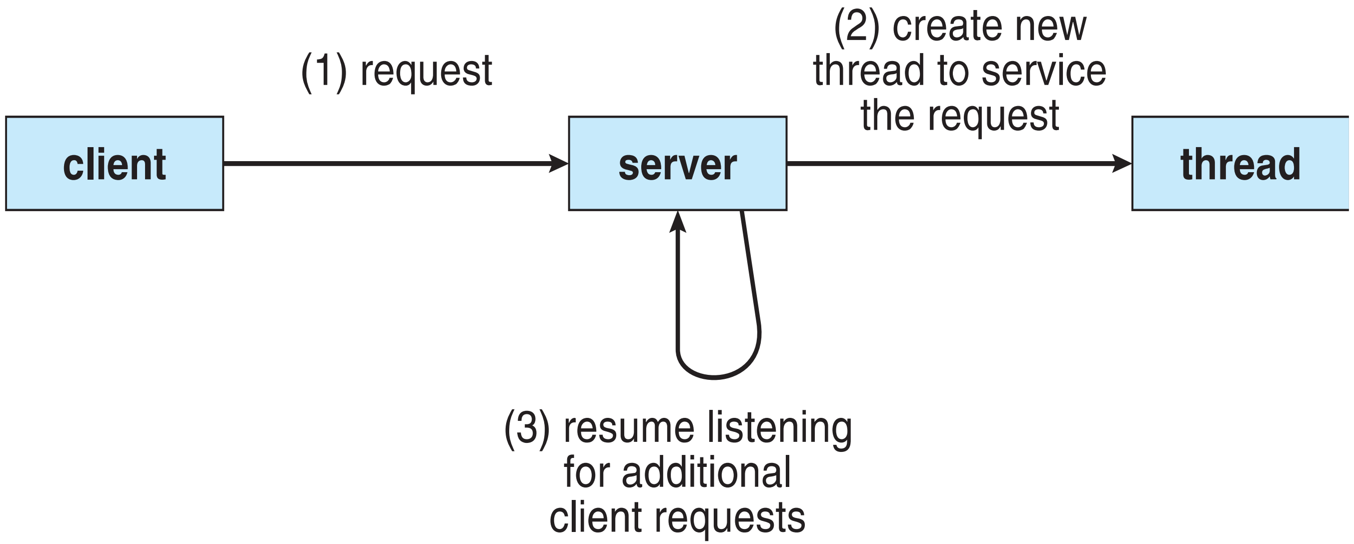
single-threaded process

multithreaded process

Examples

- **Web browser** might have one thread **display images or text** while another thread **retrieves data** from the network.
- A **word processor** may have a thread for **displaying graphics**, another

thread for **responding to keystrokes** from the user, and a third thread for performing **spelling and grammar checking** in the background


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

```
#include<stdio.h>
int main()
{
  while(1)
  {
    printf("Hello...\n");
  }
  return 0;
}
```

Check the entry in process list...

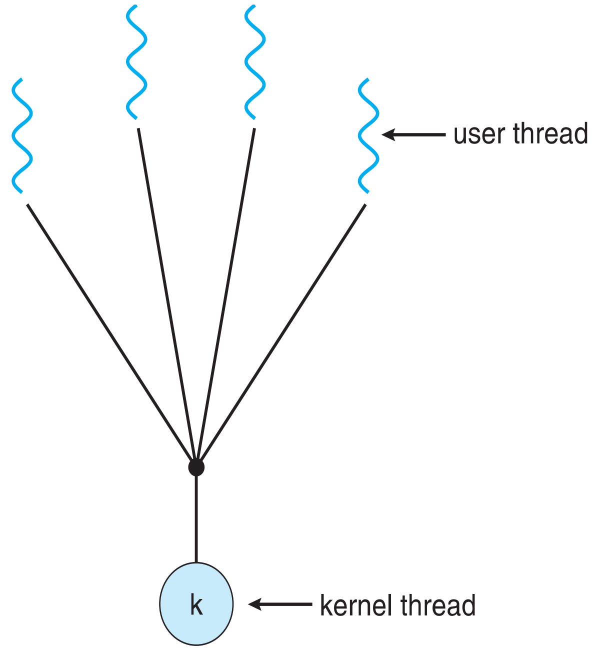
User level vs Kernel level threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

- Many-to-One
- One-to-One
- Many-to-Many

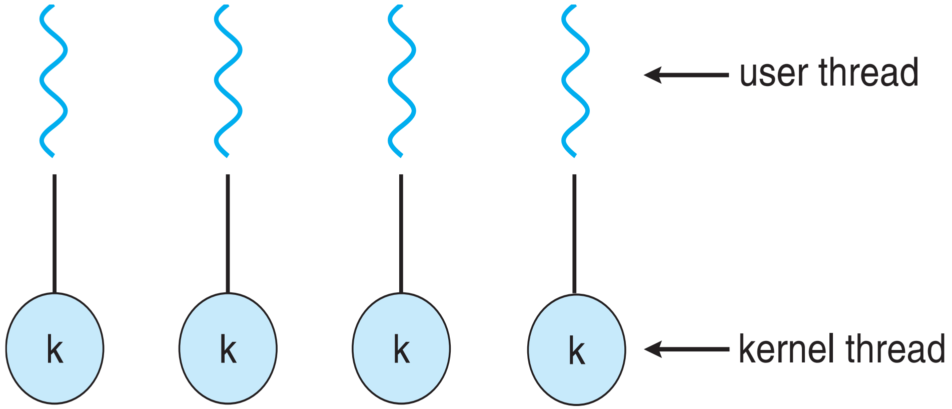
Many-to-One

- **Many user-level threads** mapped to **single kernel thread**
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



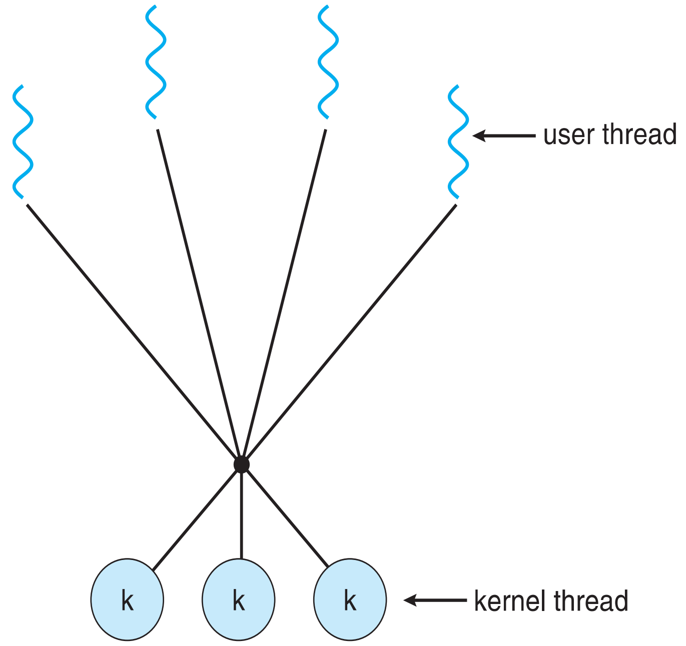
One-to-One

- **Each user-level thread** maps to kernel thread
- Creating a user-level thread creates a kernel thread
- **More concurrency** than many-to-one
- **Number of threads** per process sometimes **restricted** due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- Pthread Library (60+ functions)
  - Thread management: create, exit, detach, join, . .
  - Mutex locks: init, destroy, lock, unlock, . . .
  - Condition variables: init, destroy, wait, timed wait, ...
- Programs must include the file pthread.h
- Programs must be linked with the pthread library (**-lpthread**)

Types: **`pthread[_object]_t`**

Functions: **`pthread[_object]_action`**

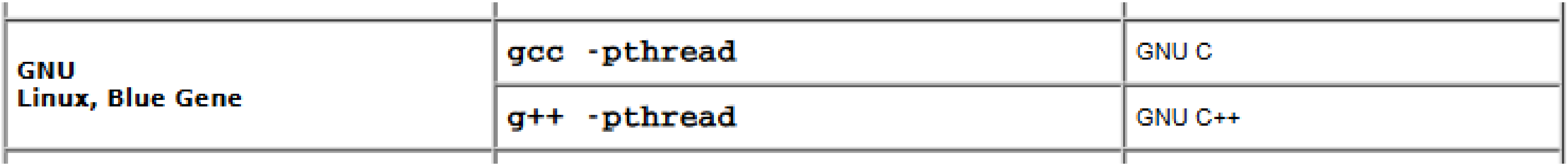
Constants/Macros: **`PTHREAD_PURPOSE`**

Examples:

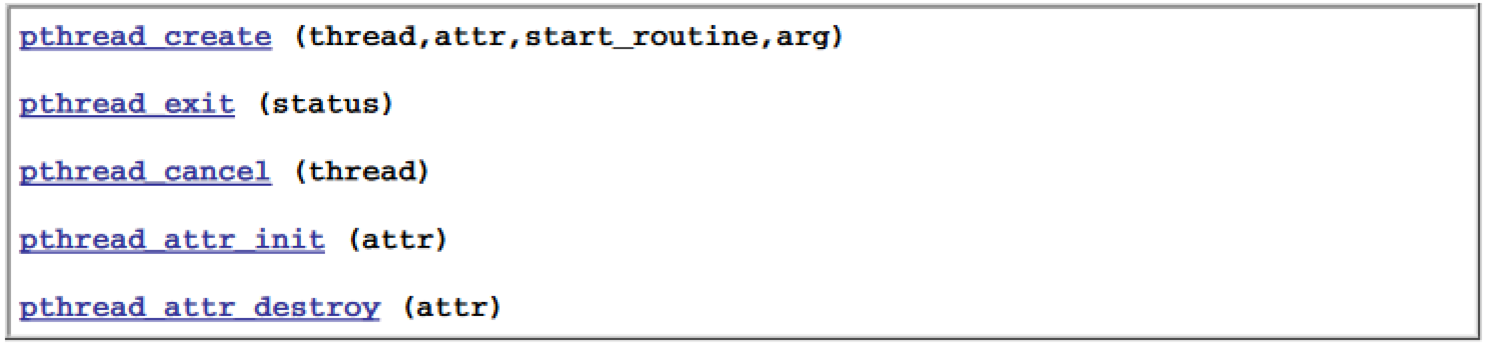
- pthread_t: the type of a thread
- pthread_create(): creates a thread
- pthread_mutex_t: the type of a mutex lock
- pthread_mutex_lock(): lock a mutex
- PTHREAD_CREATE_DETACHED

```
#include<stdio.h>
int main()
{
  while(1)
  {
    printf("Hello...\n");
  }
  return 0;
}
```

Check the entry in process list...



GNU **Linux, Blue Gene**	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++



```
pthread_create (thread,attr,start_routine,arg)

pthread_exit (status)

pthread_cancel (thread)

pthread_attr_init (attr)

pthread_attr_destroy (attr)
```

Creates a new thread

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

- Returns 0 to indicate success, otherwise returns error code...
  - thread: output argument for the id of the new thread
  - attr: input argument that specifies the attributes of the thread to be created (NULL = default attributes)
  - start_routine: function to use as the start of the new thread must have prototype: void * foo(void*)
  - arg: argument to pass to the new thread routine. If the thread routine requires multiple arguments, they must be passed bundled up in an array or a structure

Terminates the calling thread

- void pthread_exit(void *retval);

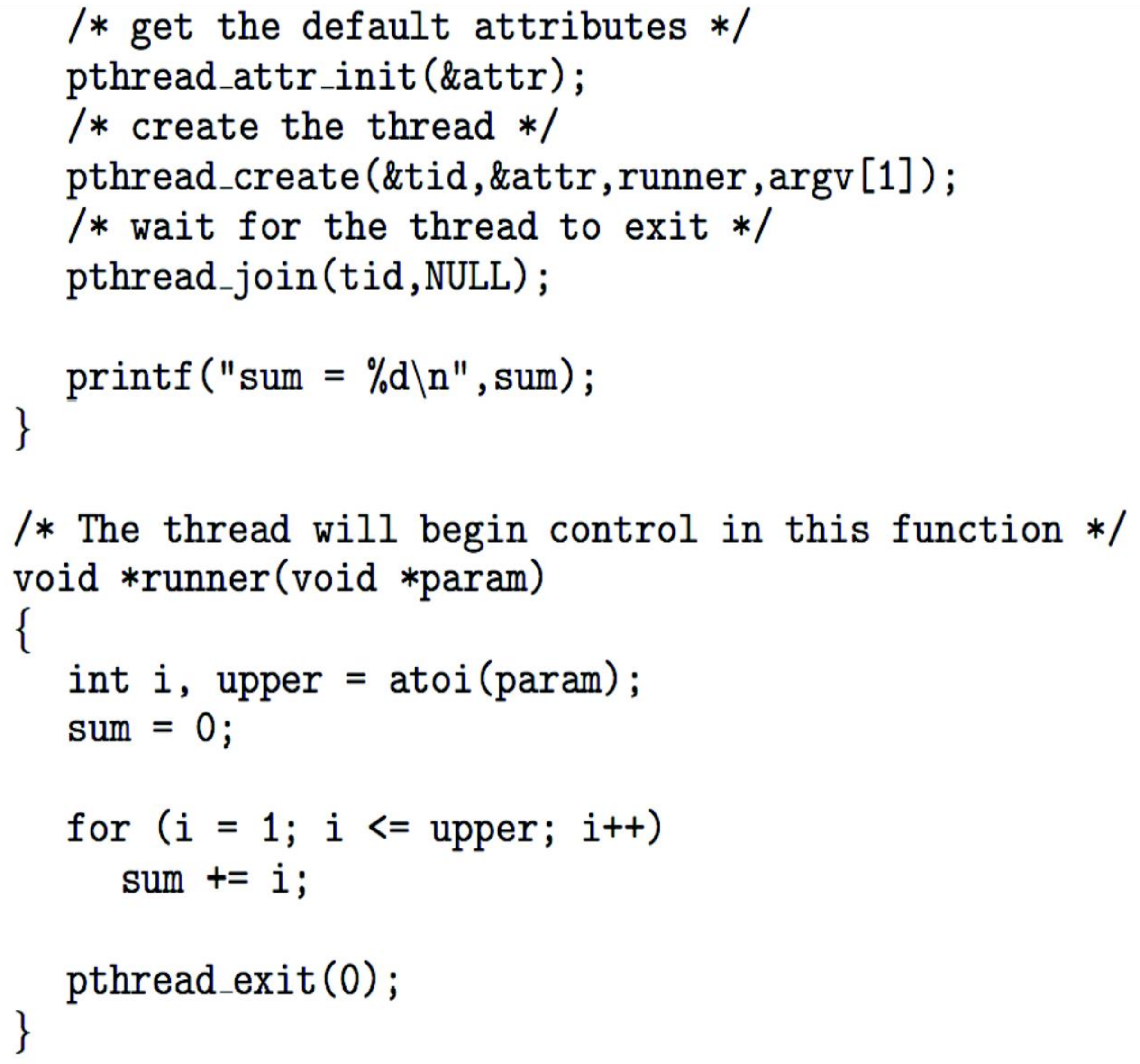
- The return value is made available to another thread calling a pthread_join()
- The **return value** of the function serves as the argument to the (implicitly called) pthread_exit().

- **Causes the calling thread to wait for another thread to terminate**
- int pthread_join(pthread_t thread, void **value_ptr);
  - thread: input parameter, id of the thread to wait on
  - value_ptr: output parameter, value given to pthread_exit() by the terminating thread (which happens to always be a void *)

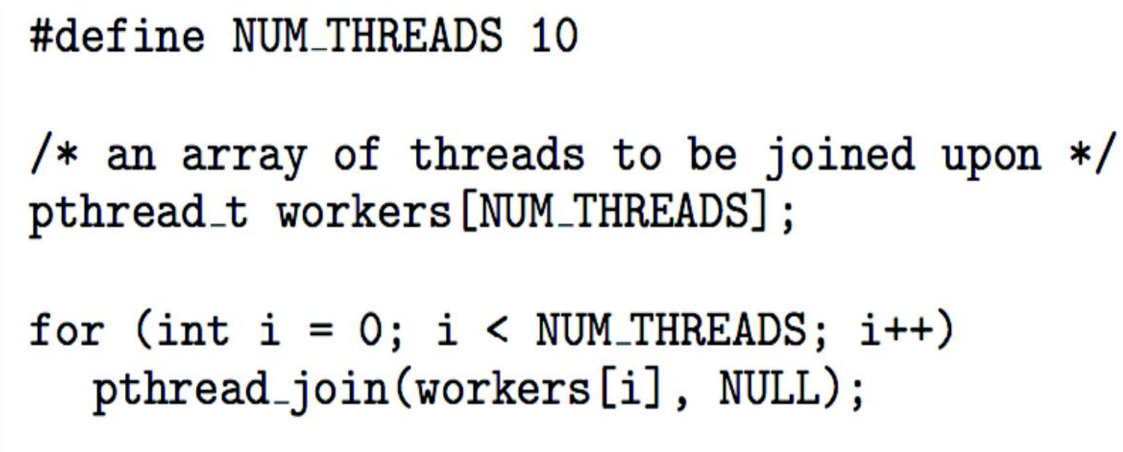
Returns 0 to indicate success, error code otherwise

Multiple simultaneous calls for the same thread are not allowed

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```



```
  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```



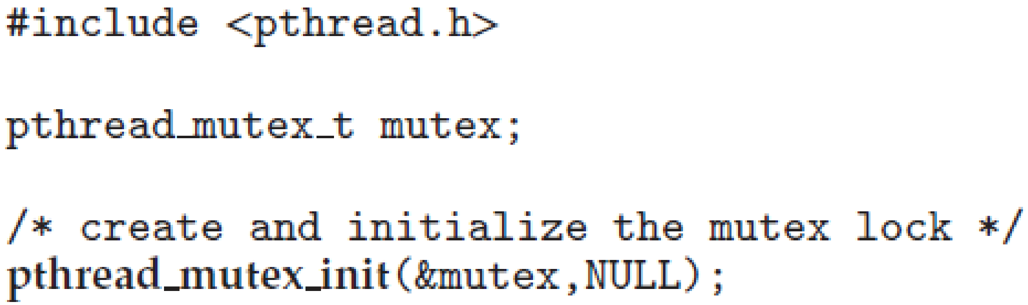
```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

Example

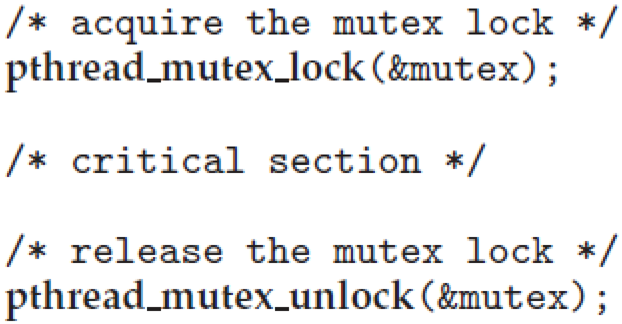
```
#include<unistd.h>
#include<stdio.h>
#include<pthread.h>
int first()
{
 int i;
 for(i=0;;i++)
 {
     printf("\nFirst: %d",i);
     sleep(1);
 }
}
```

```
int main()
{
    pthread_t th;
    int i;
    pthread_create(&th, 0,(void
                *)&first,NULL);
    for(i=0;;i++)
    {
        printf("\nMain: %d",i);
        sleep(1);
    }
    pthread_join(th, NULL);
    return 0;
}
```

Mutex lock



```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```



```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

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

int mails = 0;
pthread_mutex_t mutex;

void* routine() {
    for (int i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&mutex);
        mails++;
        pthread_mutex_unlock(&mutex);
        // read mails
        // increment
        // write mails
    }
}
```

```c
int main(int argc, char* argv[]) {
    pthread_t p1, p2, p3, p4;
    pthread_mutex_init(&mutex, NULL);
    if (pthread_create(&p1, NULL, &routine, NULL) != 0) {
        return 1;
    }
    if (pthread_create(&p2, NULL, &routine, NULL) != 0) {
        return 2;
    }
    if (pthread_create(&p3, NULL, &routine, NULL) != 0) {
        return 3;
    }
    if (pthread_create(&p4, NULL, &routine, NULL) != 0) {
        return 4;
    }
    if (pthread_join(p1, NULL) != 0) {
        return 5;
    }
    if (pthread_join(p2, NULL) != 0) {
        return 6;
    }
    if (pthread_join(p3, NULL) != 0) {
        return 7;
    }
    if (pthread_join(p4, NULL) != 0) {
        return 8;
    }
    pthread_mutex_destroy(&mutex);
    printf("Number of mails: %d\n", mails);
    return 0;
}
```

Condition variables

- There are many cases where a **thread wishes to check** whether a **condition** is true before continuing its execution.
- Example:
  - A parent thread might wish to check whether a child thread has completed.
  - This is often called a `join()`.

A Parent Waiting For Its Child

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

What we would like to see here is:

```
parent: begin
child
parent: end
```

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

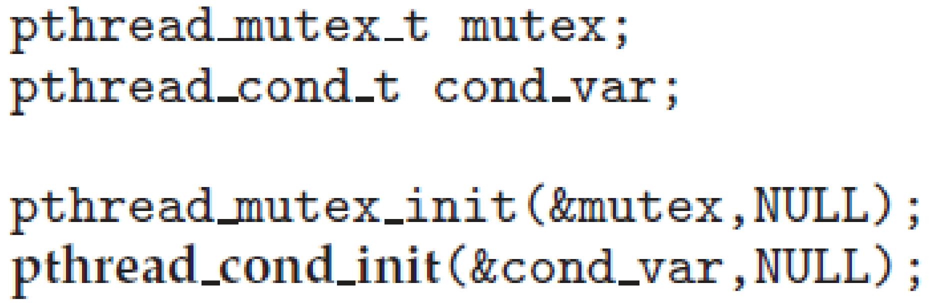
- This is hugely inefficient as the parent spins and **wastes CPU time**.

- **Condition variable**
  - **Waiting** on the condition
    - **An explicit queue** that threads can put themselves on when some state of execution is not as desired.
  - **Signaling** on the condition
    - **Some other thread**, when it changes said state, can wake one of those waiting threads and allow them to continue.

- Declare condition variable

```
pthread cond t c;
```

  - Proper initialization is required.



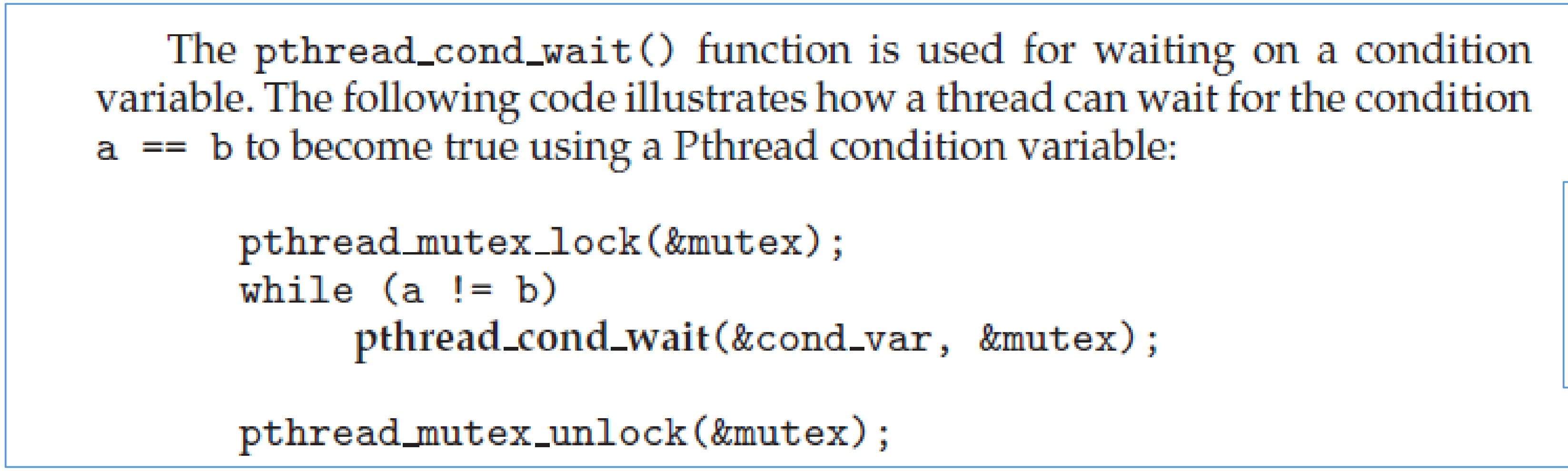
```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

- Operation (the POSIX calls)

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

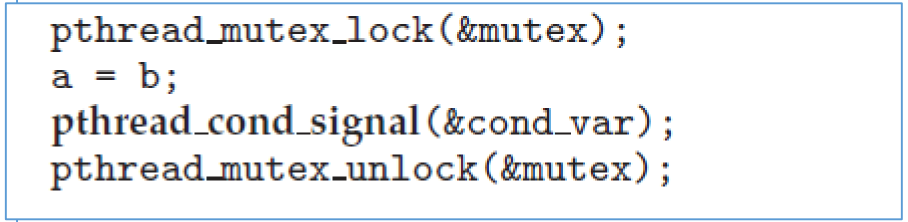
  - The wait() call takes a mutex as a parameter.
    - The wait() call **release the lock** and put the calling thread to sleep.
    - When the thread wakes up, it must **re-acquire the lock**.

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```



The `pthread_cond_wait()` function is used for waiting on a condition variable. The following code illustrates how a thread can wait for the condition `a == b` to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```



```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
        Pthread_mutex_lock(&m);
        done = 1;
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
        printf("child\n");
        thr_exit();
        return NULL;
}

void thr_join() {
        Pthread_mutex_lock(&m);
        while (done == 0)
                Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

(cont.)

```c
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

- **Parent:**
  - Create the child thread and continues running itself.
  - Call into `thr_join()` to wait for the child thread to complete.
    - Acquire the lock
    - Check if the child is done
    - Put itself to sleep by calling `wait()`
    - Release the lock
- **Child:**
  - Print the message "child"
  - Call `thr_exit()` to wake the parent thread
    - Grab the lock
    - Set the state variable `done`
    - Signal the parent thus waking it.

```
void thr_exit() {
        done = 1;
        Pthread_cond_signal(&c);
}

void thr_join() {
        if (done == 0)
                Pthread_cond_wait(&c);
}
```

- The issue here is a subtle **race condition**.
    - The parent calls `thr_join()`.
        - The parent checks the value of `done`.
        - It will see that it is 0 and try to go to sleep.
        - Just before it calls wait to go to sleep, the parent is interrupted and the child runs.
    - The child changes the state variable `done` to 1 and signals.
        - But no thread is waiting and thus no thread is woken.
        - When the parent runs again, it sleeps forever.
- Always hold the lock while signaling

The importance of the state variable **`done`**

```
void thr_exit() {
        Pthread_mutex_lock(&m);
        Pthread_cond_signal(&c);
        Pthread_mutex_unlock(&m);
}

void thr_join() {
        Pthread_mutex_lock(&m);
        Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
}
```

`thr_exit()` and `thr_join()` without variable `done` (it is a broken code)

- Imagine the case where the child runs immediately.
  - The child will signal, but there is no thread asleep on the condition.
  - When the parent runs, it will call wait and be stuck.
  - **No thread will ever wake it.**

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

pthread_mutex_t mutexFuel;
pthread_cond_t condFuel;
int fuel = 0;

void* fuel_filling(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutexFuel);
        fuel += 15;
        printf("Filled fuel... %d\n", fuel);
       pthread_cond_signal(&condFuel);
        pthread_mutex_unlock(&mutexFuel);
     sleep(1);
    }
}
```

```
void* car(void* arg) {
    pthread_mutex_lock(&mutexFuel);
    while (fuel < 40) {
        printf("No fuel. Waiting...\n");
        pthread_cond_wait(&condFuel, &mutexFuel);
        // Equivalent to:
        // pthread_mutex_unlock(&mutexFuel);
        // wait for signal on condFuel
        // pthread_mutex_lock(&mutexFuel);
    }
    fuel -= 40;
    printf("Got fuel. Now left: %d\n", fuel);
    pthread_mutex_unlock(&mutexFuel);
}
```

```
int main(int argc, char* argv[]) {
    pthread_t th[2];
    pthread_mutex_init(&mutexFuel, NULL);
    pthread_cond_init(&condFuel, NULL);
    for (int i = 0; i < 2; i++) {
        if (i == 1) {
            if (pthread_create(&th[i], NULL, &fuel_filling, NULL) != 0) {
                perror("Failed to create thread");
            }
        } else {
            if (pthread_create(&th[i], NULL, &car, NULL) != 0) {
                perror("Failed to create thread");
            }
        }
    }
    for (int i = 0; i < 2; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    pthread_mutex_destroy(&mutexFuel);
    pthread_cond_destroy(&condFuel);
    return 0;
}
```

Return values – with return statement

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

void* roll_dice() {
    int value = (rand() % 6) + 1;
    int* result = malloc(sizeof(int));
    *result = value;
    // printf("%d\n", value);
    printf("Thread result: %p\n", result);
    return (void*) result;
}
```

```
int main(int argc, char* argv[]) {
    int* res;
    srand(time(NULL));
    pthread_t th;
    if (pthread_create(&th, NULL, &roll_dice, NULL) != 0) {
        return 1;
    }
    if (pthread_join(th, (void**) &res) != 0) {
        return 2;
    }
    printf("Main res: %p\n", res);
    printf("Result: %d\n", *res);
    free(res);
    return 0;
}
```

Return values – with pthread_exit statement

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

void* roll_dice() {
    int value = (rand() % 6) + 1;
    int* result = malloc(sizeof(int));
    *result = value;
    sleep(2);
    printf("Thread result: %d\n", value);
    pthread_exit((void*) result);
}
```

```
int main(int argc, char* argv[]) {
    int* res;
    srand(time(NULL));
    pthread_t th;
    if (pthread_create(&th, NULL, &roll_dice, NULL) != 0) {
        return 1;
    }
    // pthread_exit(0);
    if (pthread_join(th, (void**) &res) != 0) {
        return 2;
    }
    printf("Result: %d\n", *res);
    free(res);
    return 0;
}
```

Detach

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

#define THREAD_NUM 2

void* routine(void* args) {
    sleep(1);
    printf("Finished execution\n");
}
```

```
int main(int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
   int i;
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_create(&th[i], NULL, &routine, NULL) != 0) {
            perror("Failed to create thread");
        }
          pthread_detach(th[i]);
    }

    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    //  pthread_exit(0);
}
```

Detach

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

#define THREAD_NUM 2

void* routine(void* args) {
    sleep(1);
    printf("Finished execution\n");
}
```

```
int main(int argc, char *argv[]) {
    pthread_t th[THREAD_NUM];
    pthread_attr_t detachedThread;
    pthread_attr_init(&detachedThread);
    pthread_attr_setdetachstate(&detachedThread, PTHREAD_CREATE_DETACHE

    int i;
    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_create(&th[i], &detachedThread, &routine, NULL) != 0) {
            perror("Failed to create thread");
        }
        // pthread_detach(th[i]);
    }

    for (i = 0; i < THREAD_NUM; i++) {
        if (pthread_join(th[i], NULL) != 0) {
            perror("Failed to join thread");
        }
    }
    pthread_attr_destroy(&detachedThread);
    pthread_exit(0);
}
```