Multithreading model

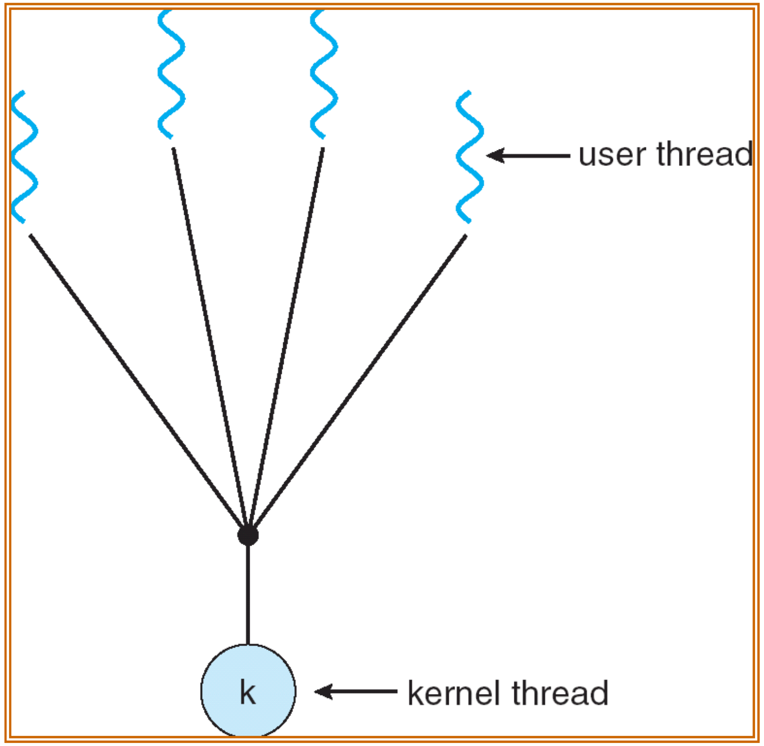
- Many-to-One
- One-to-One
- Many-to-Many

1. A scheduler is not aware of the existence of user threads

2. If a kernel thread is blocked, then its associated user threads are blocked as well.

3. One kernel thread can be allocated to one processor

- Many-to-one model

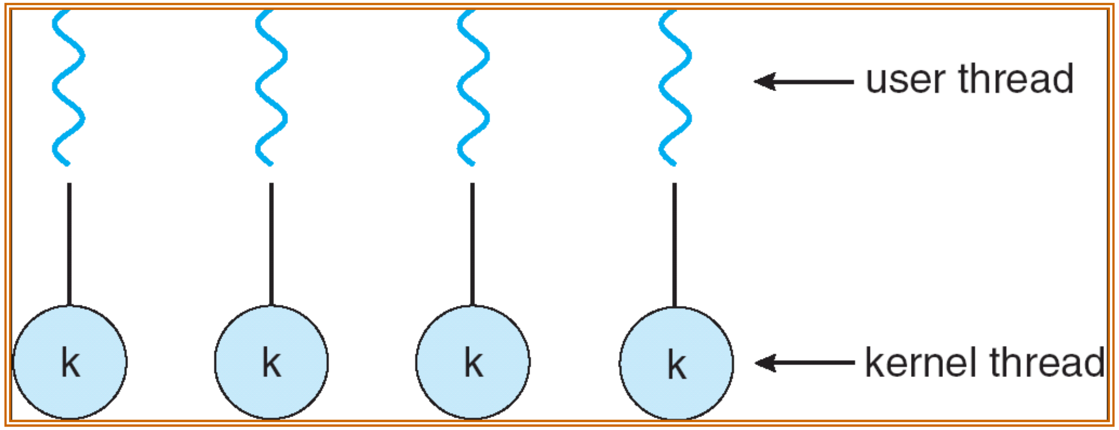


Very fast, small overhead

If one thread calls blocking system call, then the entire process will block

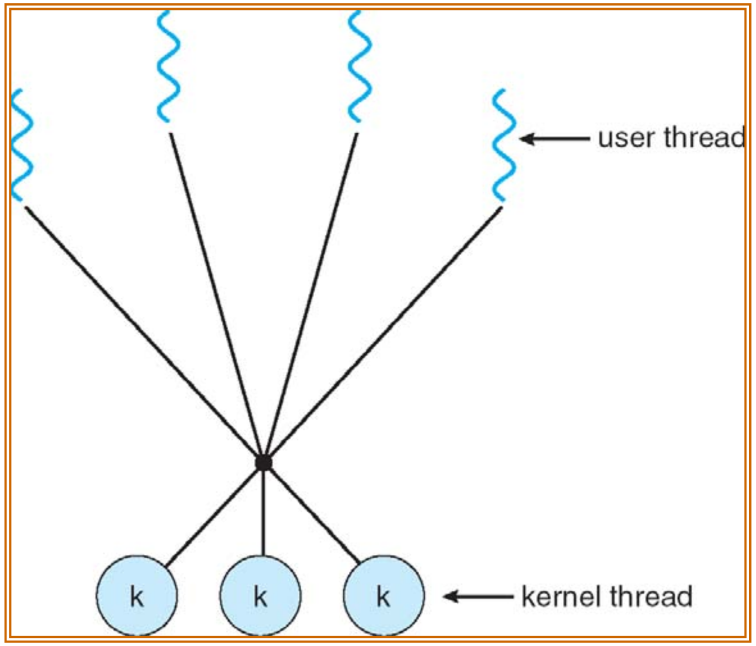
Multiple threads are unable to run in parallel on multiprocessors

One-to-one model



1. Maps each user thread to a kernel thread
2. Another thread can run when a thread makes a blocking system call
3. It allows multiple threads to run in parallel on multiprocessors
4. But creating kernel threads are resource-intensive.

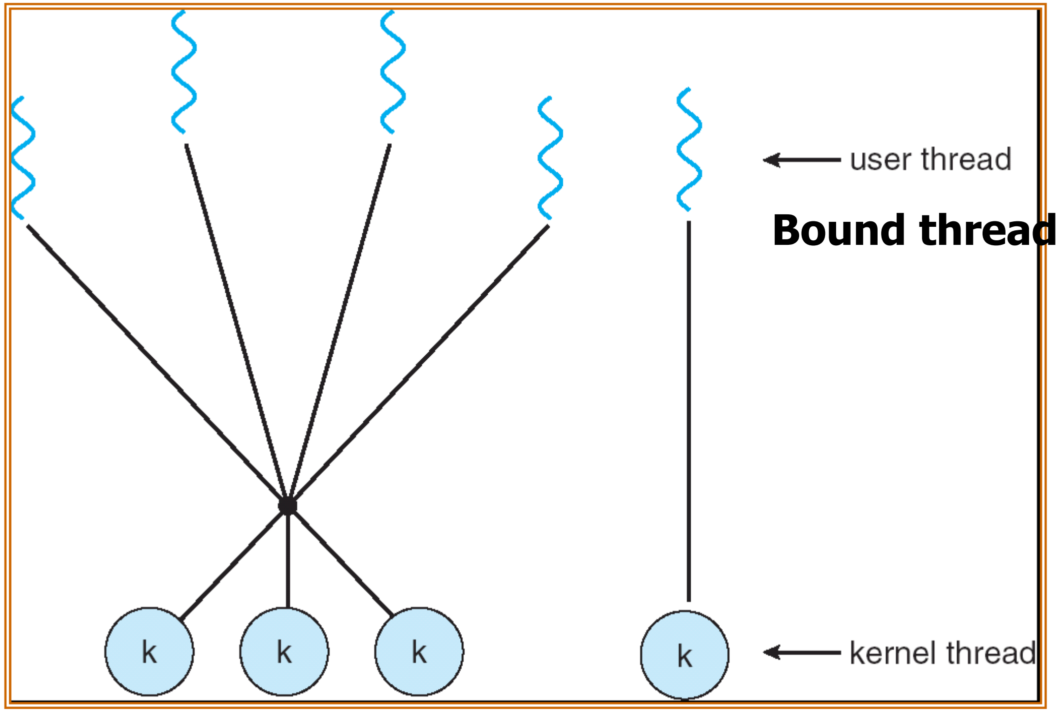
- Many-to-many model
  - Multiplexes many user threads to smaller or equal-number of kernel threads



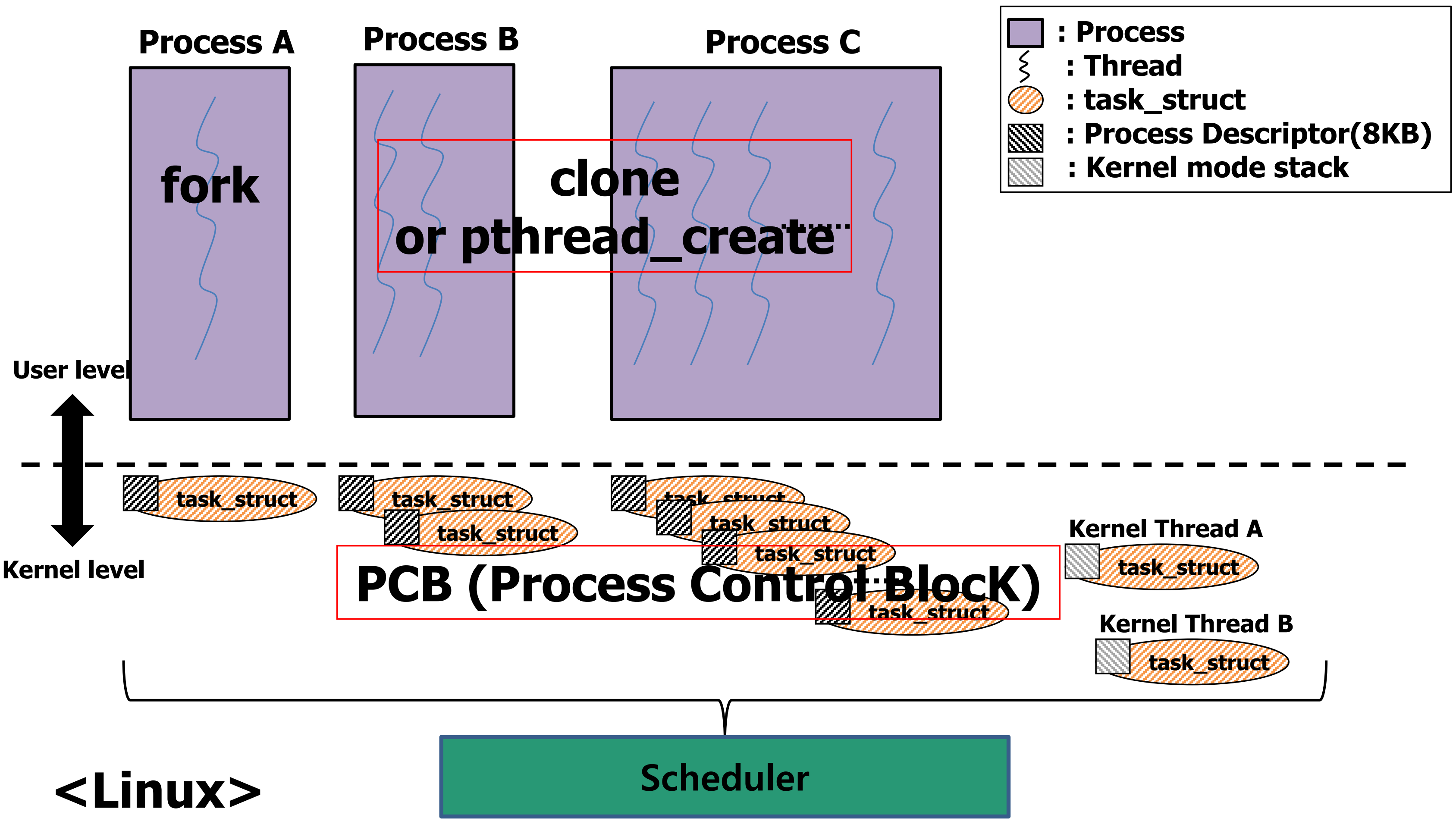
- Many-to-many model
  - Hybrid approach between one-to-one model and many-to-one model
  - Advantages
    - 1. Developers can create as many as user threads as necessary
    - 2. The corresponding kernel threads can run in parallel on multiprocessors
    - 3. When a thread calls a blocking system call, the kernel can schedule another thread

<Scheduler Activation>

- Two-level model
  - Many-to-many model + one-to-one model



Linux and windows use one-to-one model



<Linux>

Thread libraries

- Thread library
  - Provides the programmer an API for creating and managing threads
  - Two methods;
    - To provide a library entirely in user space
      - Invoking a function in the library results in a local function call in user space and not a system call
    - To implement a kernel-level library supported directly by the operating system
      - Invoking a function in the library results in a system call

- Three primary thread libraries:
  - POSIX Pthreads
    - May be provided as either a kernel- or user-level library
  - Win32 threads
    - Provided as a kernel-level library
  - Java threads
    - Implemented using a thread library available on the host system
      - For example, on Windows systems, Java threads use Win32 API

Thread programming example (pthread library)

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

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the thread */

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

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);

   /* now wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d₩n",sum);
}
```

Thread function

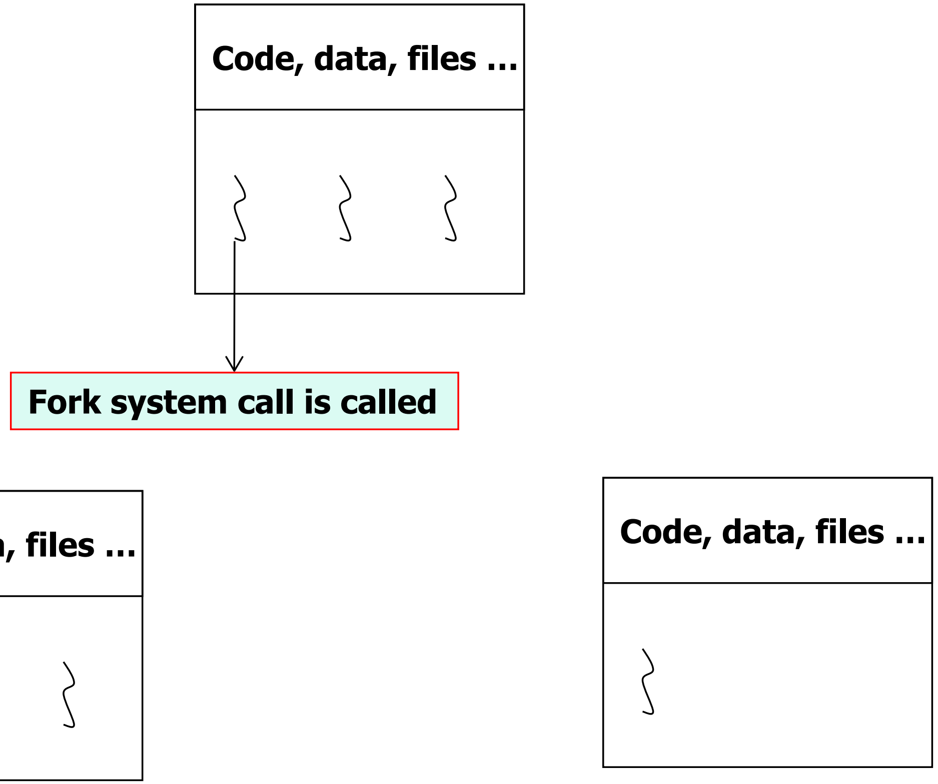
```
void *runner(void *param)
{
        int i, upper = atoi(param);
        sum = 0;

        if (upper > 0) {
                for (i = 1; i <= upper; i++)
                        sum += i;
        }

        pthread_exit(0);
}
```

Threading issues

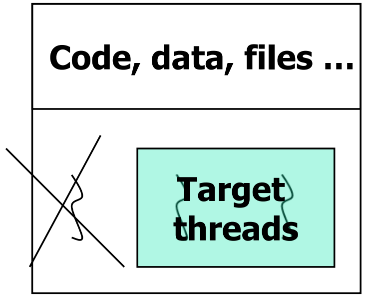
- Semantics of fork() and exec()
  - Does **fork()** duplicate 1) only the calling thread or 2) all threads?
    - If exec() is immediately called,
      - Only the calling thread is duplicated
    - If exec() is not called,
      - all threads are duplicated

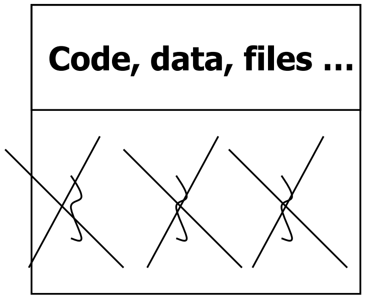


If exec system call is not called

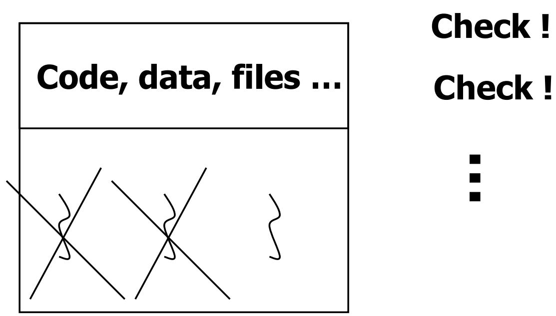
If exec system call is called immediately

- Thread cancellation
  - Terminating a thread before it has finished
    - For example,
      - Often, a web page is loaded using several threads
      - When a user presses a button on a web browser that stops a web page from loading any further.
  - Target thread
    - A thread that is to be cancelled
  - Two general approaches:
    - **Asynchronous cancellation** terminates the target thread immediately
    - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
      - Safe cancellation points





Asynchronous cancellation

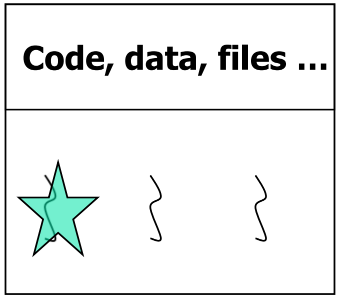


Deferred cancellation

- Signal handling
  - Signals are used in UNIX systems to notify a process that a particular event has occurred;
    - Synchronous signal
      - Examples: Illegal memory access, divide by zero
    - Asynchronous signal
      - By an event external to a running process
  - signal handler
    - A default signal handler
    - A user-defined signal handler

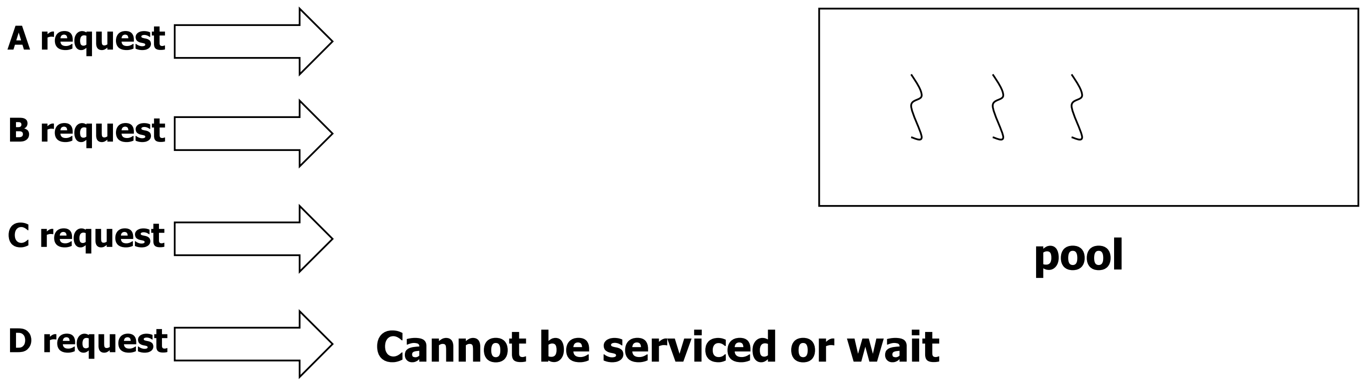
- Where should a signal be delivered ?
  - 1. Deliver the signal to the thread to which the signal applies
  - 2. Deliver the signal to every thread in the process
  - 3. Deliver the signal to certain threads in the process
  - 4. Assign a specific thread to receive all signals for the process
- Examples
  - Division by zero
    - 1.
  - <control><c>
    - 2.
- POSIX API
  - pthread_kill (pthread_t tid, int signal)

A signal is caused by the division by zero.



If the <control><c> button has been pushed

- Thread pools
  - Motivation;
    - Unlimited threads may exhaust system resources
  - Thread pool
    - Only limited number of threads are admitted to the system
    - Create a number of threads in a pool where they wait for a work
  - Advantages:
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool


A request
B request
C request
D request
pool
Cannot be serviced or wait

- Thread-specific data
  - Allows each thread to have its own copy of data
  - Most thread libraries provide some form of support for thread-specific data

Thread programming API

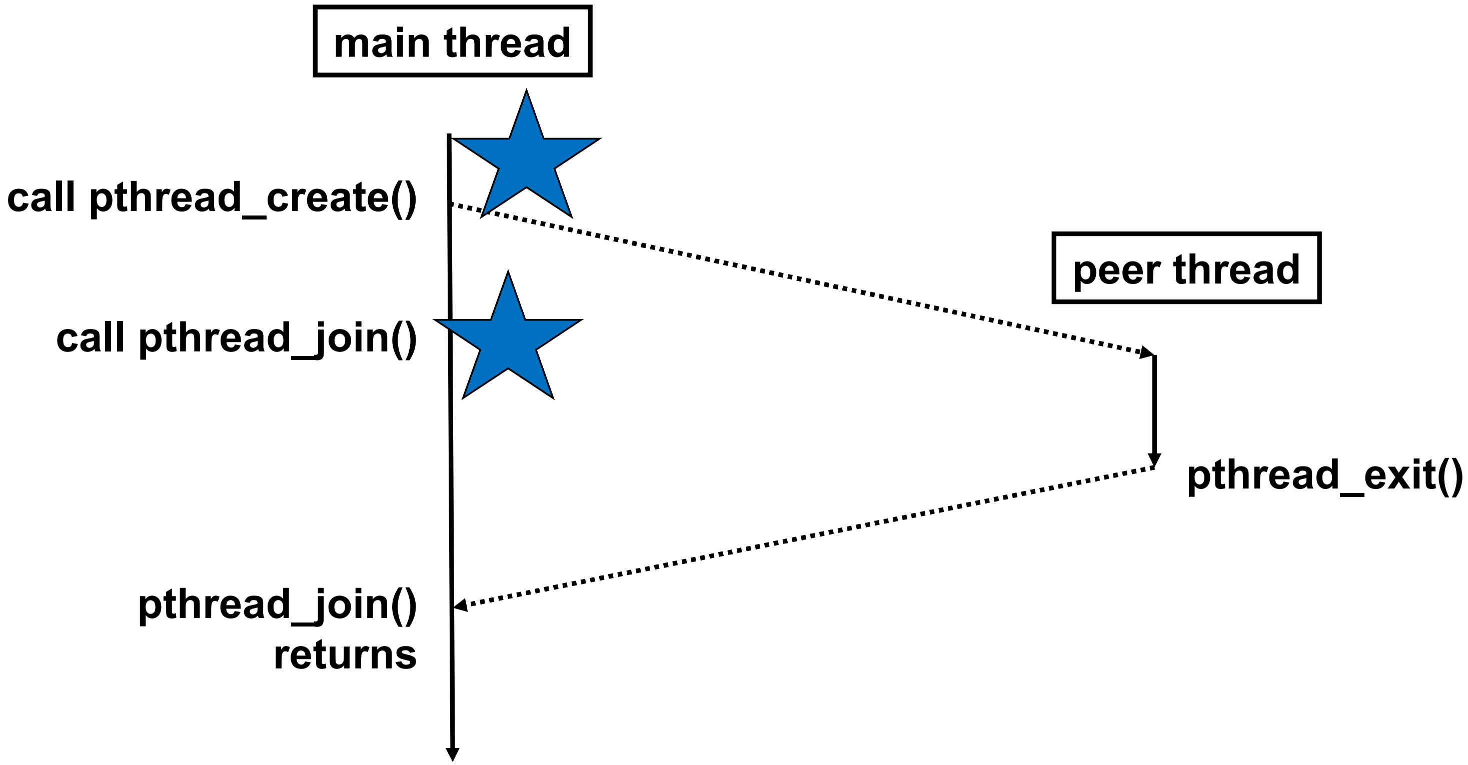
- Thread programming API
  - pthread_create
  - pthread_join
  - pthread_exit

- POSIX thread programming (1)
  - Thread creation
    - 1> Prototype
      - #include <pthread.h>
      - int pthread_create(pthread_t* tid, pthread_attr_t *attr, (void *) f, void *arg);
    - Roles
      - Creates a new thread and runs the thread routine f with an input argument of arg
      - When pthread_create returns, argument tid contains the ID of the newly created thread

- POSIX thread programming (2)
  - Reaping terminated threads
    - Prototype
      - #include <pthread.h>
      - int pthread_join(pthread_t tid, void *thread_return);
    - Roles
      - pthread_join function blocks until thread tid terminates
      - It is similar to wait function but can only wait for a specific thread to terminate

- POSIX thread programming (3)
  - Terminating the threads
    - Prototype
      - #include <pthread.h>
      - int pthread_exit(void *thread_return);
    - Roles
      - Terminating the thread with a return value of thread_return that will be transferred to pthread_join

S/W architecture



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

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */

   /* create the thread */
   pthread_create(&tid,NULL,runner,argv[1]);

   /* now wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d₩n",sum);
}
```

Thread function

```c
void *runner(void *param)
{
        int i, upper = atoi(param);
        sum = 0;

        if (upper > 0) {
                for (i = 1; i <= upper; i++)
                        sum += i;
         }

         pthread_exit(0);
}
```