# Operating System Tutorial

Tutorial 8

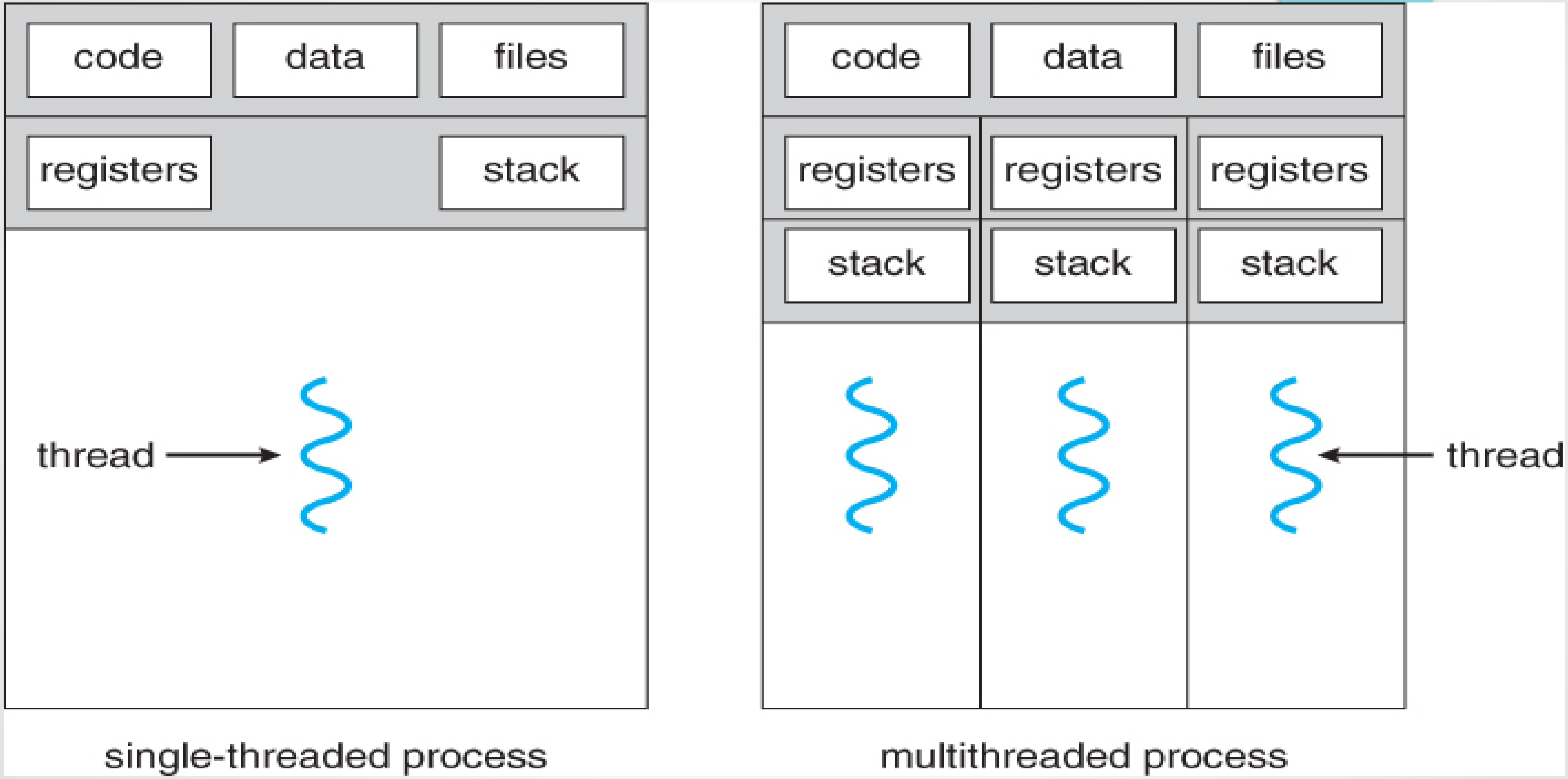
Thread

(pthreads library)

## What is Thread ?

- **Thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.
- Because threads have some of the properties of processes, they are sometimes called lightweight processes.
- A process can have multiple threads, those threads allow multiple executions of streams within the process.

## Thread


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

## PIPE

- PIPE Pipes permit sequential communication from one process to a related process.
- A PIPE permits unidirectional communication.
- PIPE is a temporary file, which has different file-descriptors for reading and writing.
- Data written to the “write end” of the pipe is read back from the “read end” .
- Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

## Motivation

- Traditionally processes are created using the fork() system call.
- Fork() produces a second copy of the calling process.
- Independent processes can execute in parallel.

## Motivation

- This independence provides memory protection and therefore stability.
- But when multiple processes working on the same task/problem or running concurrently (using IPC), has following disadvantages:
- The cost of switching between multiple processes is relatively high.
- There are often severe limits on the number of processes the scheduler can handle efficiently.
- Synchronization variables, shared between multiple processes, are typically slow.

## Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- Ex1: In a word processor, a background thread may check spelling and grammar while a foreground thread processes user input ( keystrokes ), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Ex2: Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.

## Thread Vs Process

- Threads in the same process share:
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id

## Thread Vs Process

- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno

## pthreads

- ‘pthread’ is a very popular API in C/C++ for threading an application also known as POSIX threads.
- It allows us to spawn a new concurrent process flow.
- A thread is spawned by defining a function and it's arguments which will be processed in the thread.
- The purpose of using the POSIX thread library in your software is to execute software faster.

## pthread_create()

***int pthread_create (pthread_t *thread_id,***
***const pthread_attr_t *attributes,***
***void *(*thread_function)(void *),***
***void *arguments);***

Arguments:

- thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h)
- void * (*start_routine) - pointer to the function to be threaded.
- *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

pthread_create( &my_thread, NULL, func1, (void*) message);

## Example

```
void printMsg(char* msg)
{
    printf(“%s\n”, msg);
}

int main(int argc, char** argv)
{
    pthread_t thrdID;
    printf(“creating a new thread\n”);
    pthread_create(&thrdID, NULL, (void*)printMsg, argv[1]);
    printf(“created thread %d\n”. thrdID);
    pthread_join(thrdID, NULL);
    return 0;
}
```

**Compile:** **gcc thread.c -lpthread**

## pthread_exit ()

```
int pthread_exit (void *status);
```

- status is the return value of the thread.
- This routine kills the thread.

## pthread_self()

- A thread can get its own thread id, by calling pthread_self()

  pthread_t pthread_self ();

- Two thread id's can be compared using pthread_equal()

  int pthread (pthread_t t1, pthread_t t2);

- Returns zero if the threads are different threads, non-zero otherwise.

## Thread Synchronization

The threads library provides three synchronization mechanisms:

- joins - Make a thread wait till others are complete (terminated).
- mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- condition variables – Allow you to wait until another thread completes an arbitrary activity.

## pthread_join()

- One Thread can wait on the termination of another by using pthread_join().
- The pthread_join() function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.

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

- The exit status is returned in status_ptr.

## Mutex

- Mutexes have two basic operations, lock and unlock.

```
int pthread_mutex_init (pthread_mutex_t *mut, const pthread_mutexattr_t *attr);
```

- Initialize a mutex object.
- Pass a pointer to the mutex,
- To use the default attributes pass NULL for the second parameter.
- Also can be initilized in following way:

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

## Lock and Unlock Mutex

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

- Locks the mutex :).

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

- Unlocks the mutex :).

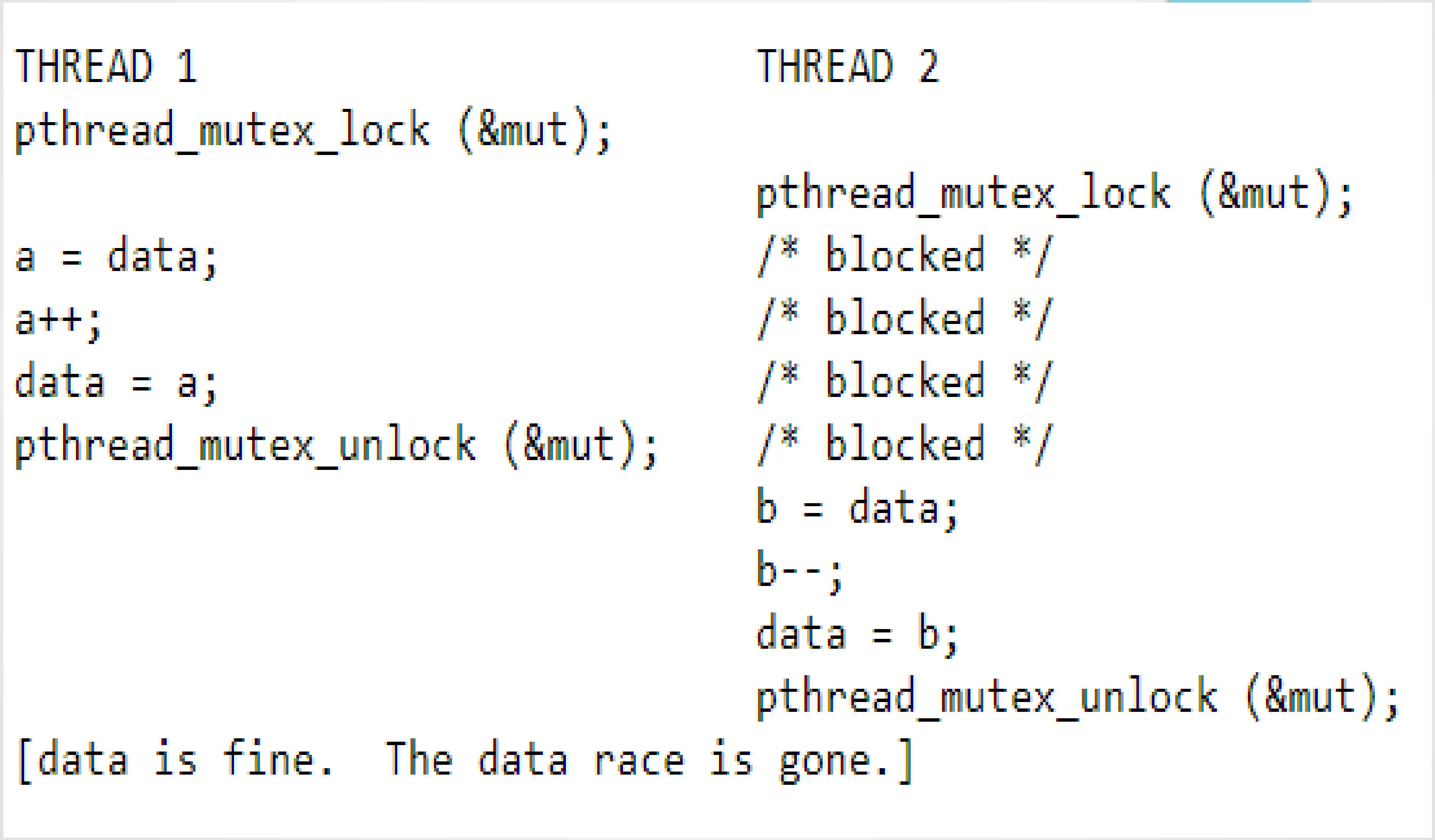
```
int pthread_mutex_trylock (pthread_mutex_t *mut);
```

- Either acquires the lock if it is available, or returns EBUSY.

```
int pthread_mutex_destroy (pthread_mutex_t *mut);
```

- Deallocates any memory or other resources associated with the mutex.

## Example



```
THREAD 1                        THREAD 2
pthread_mutex_lock (&mut);
                                pthread_mutex_lock (&mut);
a = data;                       /* blocked */
a++;                            /* blocked */
data = a;                       /* blocked */
pthread_mutex_unlock (&mut);    /* blocked */
                                b = data;
                                b--;
                                data = b;
                                pthread_mutex_unlock (&mut);
[data is fine.  The data race is gone.]
```

## Task 1

Write a program in which four threads of a process are accessing the same variable.

Each of the thread is modifying the variable in different manner. Display output of each the thread and check. (Correct or not)

## Task 2

Modify above program by using synchronization (i.e. mutex) to ensure the correct execution.

## Task 3

Design a client-server application to check the given number is prime or not. Server should create a new thread for each client request which responds to request.

## Task 4

Modify the socket-based date server program so that the server services each client request in a separate thread.

## Task 5

The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, .... Formally, it can be expressed as:

- $fib_0 = 0,\ fib_1 = 1,\qquad fib_n = fib_{n-1} + fib_{n-2}$

Write a multithreaded program that generates the Fibonacci series which work as follows:

The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that is shared by the threads (an array is probably the most convenient data structure).When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish using the techniques.