Operating Systems 2017/2018 — Part V – Threads

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Slides based on the book

'Operating System Concepts, 9th Edition,

Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley'

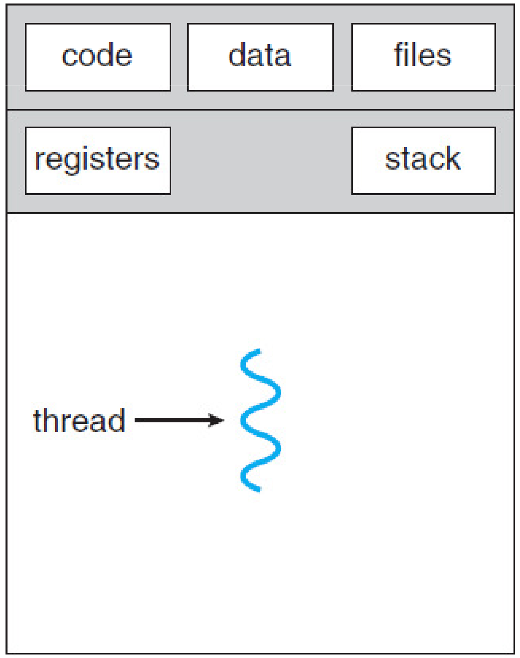
Chapter 3

Thread Concept

- So far, we saw that a process is a running program with a **single thread of execution**, meaning that it can **perform only one task at a time**
- Modern operating systems have extended the process concept to allow a process to have **multiple threads of execution** and thus to **perform more than one task at a time**
  - Most modern applications can be seen as a set of multiple tasks
  - The processing of **independent tasks or asynchronous events** can be interleaved by assigning a separate thread to handle each task or event type

Single and Multithreaded Processes

- **Single-threaded** is the traditional approach of a single thread of execution per process, in which the **concept of a thread is not recognized**
- **Multithreading** refers to the ability of an OS to support **multiple concurrent paths of execution within a single process**



single-threaded process

Main Benefits

- There are 4 main benefits of multithreaded programming:
  - **Performance** – in general it is significantly more time consuming to create and manage processes than threads, thread creation is cheaper than process creation and thread switching has lower overhead than context switching
  - **Resource Sharing** – threads share by default the memory (same address space) and the resources of the process they belong to, while multiple processes have to use complex operating system mechanisms to share memory or other resources
  - **Responsiveness** – multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, especially important for user interfaces
  - **Parallelism** – a single-threaded process can run on only one processor, regardless how many are available in a multiprocessor architecture, where threads may be running in parallel on different processing processors/cores

Performance Implications

- If there is an application that should be implemented as a set of related units of execution, it is far more efficient to do so as a collection of threads rather than a collection of separate processes:
  - It takes far **less time to create a new thread** in an existing process than to create a brand-new process
  - It takes **less time to terminate a thread** than a process
  - It takes **less time to switch between two threads** within the same process than to switch between processes
  - It is **more efficient to communicate between threads** than between processes since communication between processes usually requires the intervention of the kernel

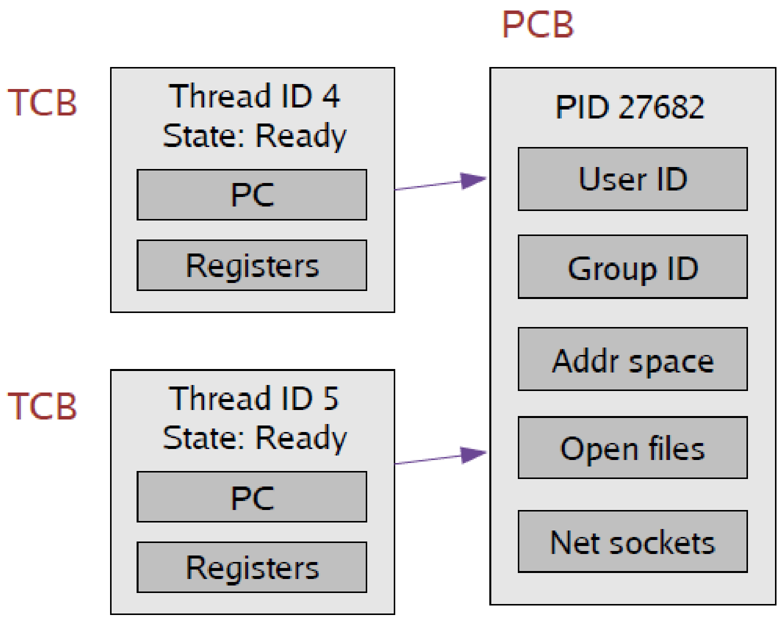
Thread Execution Context

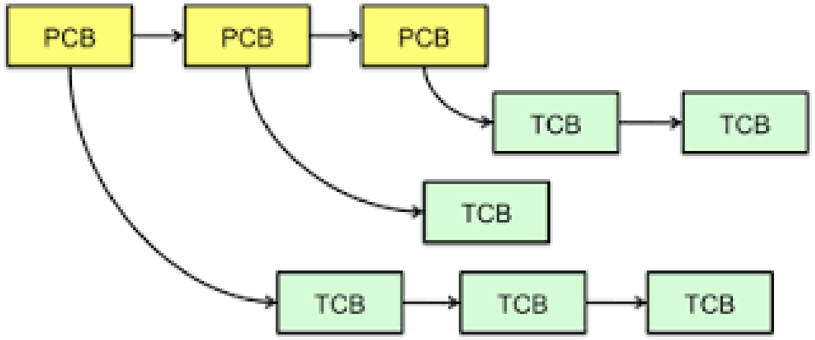
- **Everything within a process is sharable among all the threads** belonging to the process:
  - Text of the executable program
  - Program’s data, heap and stacks
  - Operating system resources, such as open files and signals

- A thread consists of the information necessary to represent an **execution context within a process**, which includes:
  - Thread ID (TID)
  - Scheduling priority and policy
  - Set of registers, including the program counter
  - Stack
  - Thread-specific data (similar to static data, but unique to each thread)

Thread Control Block (TCB)

- The information related to thread execution is stored out of the PCB in **thread control blocks (TCBs)**
  - Each thread has its own TCB, which includes all thread-specific execution information plus a pointer to the corresponding PCB
  - PCB keeps the remaining (non-execution related) management information about the containing process plus a pointer to the list of associated TCBs



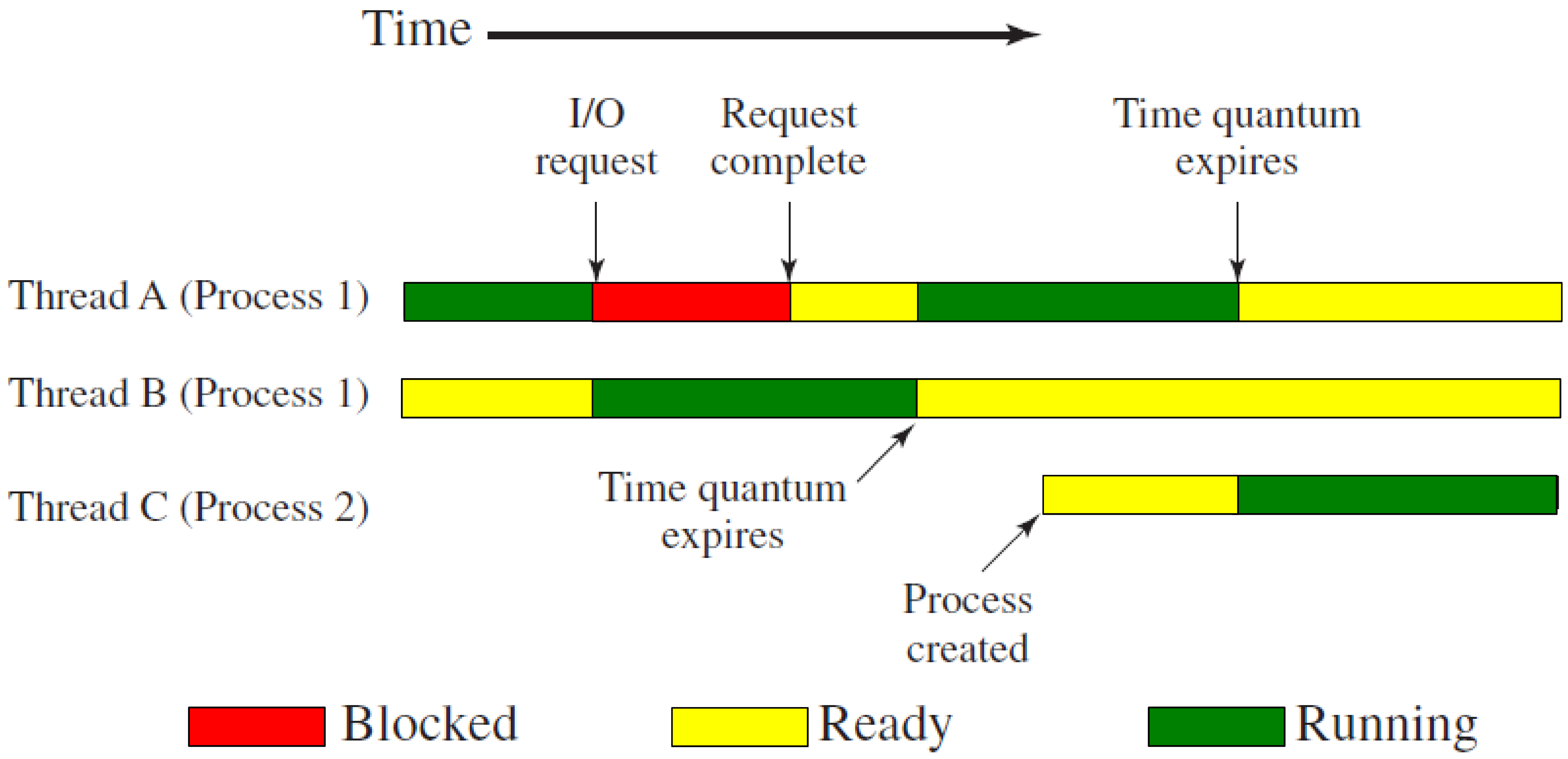


Thread States

- As for processes, the key states for a thread are:
  - **Running**, instructions are being executed
  - **Waiting/Blocked**, the thread is waiting for some event to occur
  - **Ready**, the thread is waiting to be assigned to a processor

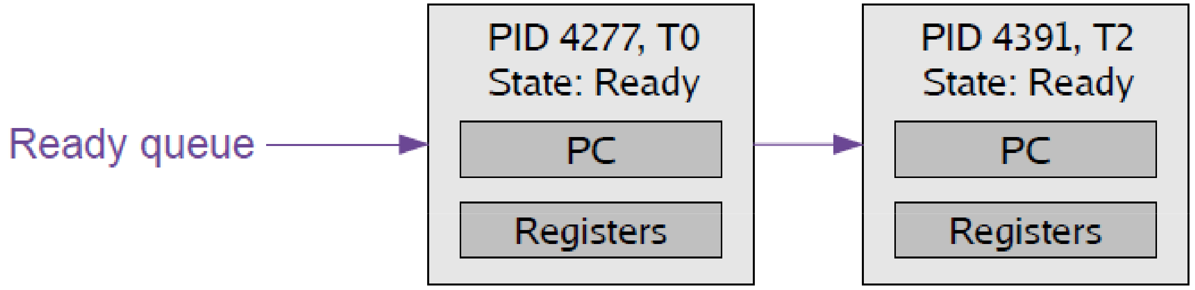
- There are four basic thread operations that can change the thread state:
  - **Spawn** – when a new thread is spawned (the thread is provided with its own register context and stack space and placed on the ready queue)
  - **Block** – when a thread needs to wait for some event to occur (the thread saves its register context and stack pointers)
  - **Unblock** – when the event for which a thread is blocked occurs (the thread is moved to the ready queue)
  - **Finish** – when a thread completes execution (the thread's register context and stack space are deallocated)

Transition Among States



Thread Context Switch

- Scheduling queues now point to TCBs and context switch now requires **saving the context of the current thread to its TCB** and **loading the saved context for the new thread from its TCB**



- Context switch between two threads in the same process:
  - No need to change the memory address-space

- Context switch between two threads in different processes:
  - Must change memory address-space, sometimes invalidating cache or generating page faults

Thread Libraries

- A **thread library** provides a API for creating and managing threads
  - **User-level library (user threads)** – management is done by a user-level threads library and without kernel support. All code and data structures for the library exist in user space, thus invoking a function in the library results in a local function call in user space and not in a system call to the kernel
  - **Kernel-level library (kernel threads)** – supported and managed directly by the operating system. All code and data structures for the library exist in kernel space, thus invoking a function in the library results in a system call to the kernel

Thread Libraries

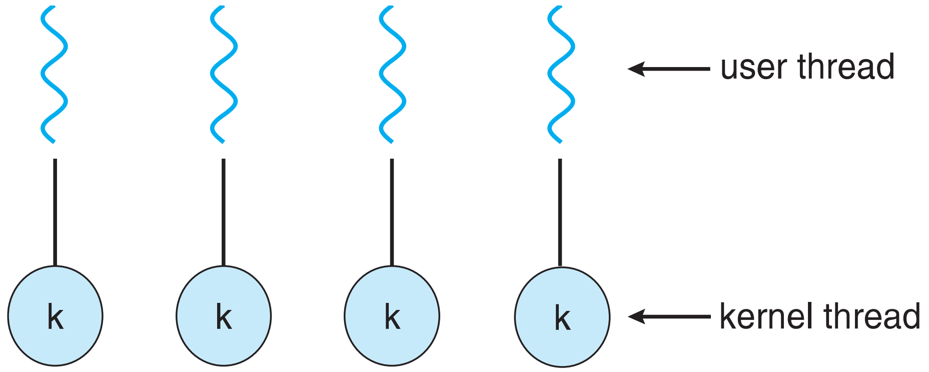
- All contemporary operating systems – including Windows, Linux, Mac OS X and Solaris – support kernel threads

- There are three main thread libraries:
  - **POSIX Pthreads**, provided as either a user-level or a kernel-level library
  - **Windows Threads**, kernel-level library available on Windows systems
  - **Java Threads**, managed by the JVM and typically implemented using the thread library available on the host system

- Ultimately, a relationship must exist between user threads and kernel threads. There are 3 common ways of establishing such a relationship:
  - **One-to-one model**
  - **Many-to-one model**
  - **Many-to-many model**

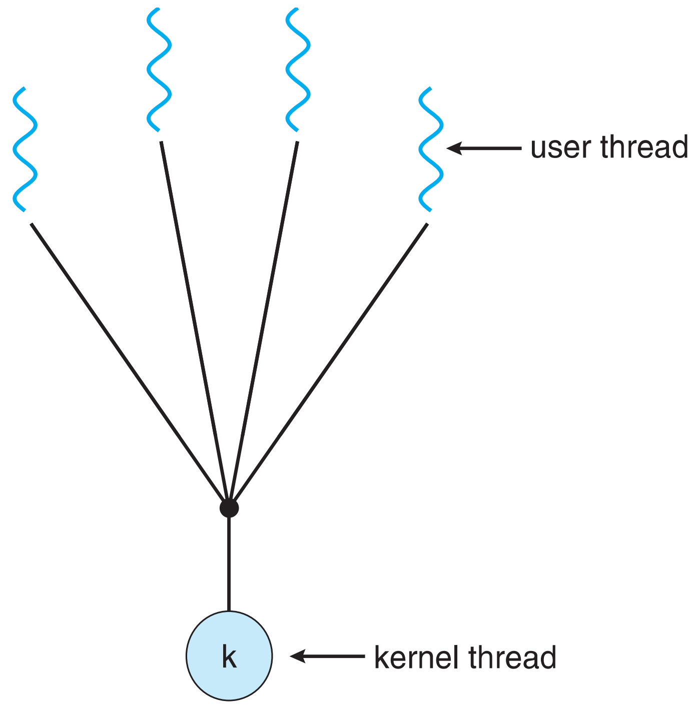
One-to-One Model

- Maps each user-level thread to a kernel thread:
  - Creating a user-level thread creates a kernel thread
  - Competition takes place among all kernel threads in the system
  - When a thread makes a blocking system call, another thread can execute
  - Allows multiple threads to run in parallel on multiprocessor/multicore systems
  - Since the overhead of creating kernel threads can burden the performance of the system, most implementations restrict the number of threads supported by the model
- Examples:
  - **Linux**
  - **Windows NT/XP/2000**
  - **Solaris 9 and later**



Many-to-One Model

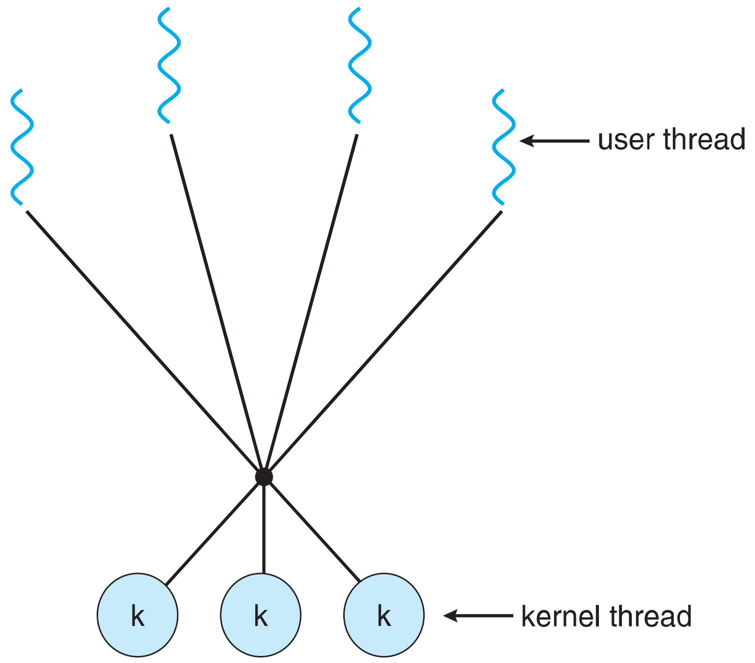
- Maps many user-level threads to one kernel thread (process):
  - Thread management is done by the thread library in user space
  - Competition takes place among the threads within the process
  - One thread blocking causes all to block
  - Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessor/multicore systems



- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads:
  - The number of kernel threads may be specific to either a particular application or machine
  - Developers can create as many user threads, but only the corresponding kernel threads can run in parallel on multiprocessor/multicore systems
  - When a thread makes a blocking system call, the kernel can schedule another thread for execution
- Examples:
  - **Solaris prior to version 9**
  - **Windows ThreadFiber package (NT/2000)**



Thread Creation – Pthreads

- New threads can be created with the function:

```
int pthread_create(pthread_t *tidp, pthread_attr_t *attr,
                   void *(*start_rtn)(void *), void *arg);
```

  - The new thread starts running the **start_rtn()** function with the **arg** argument and sets **tidp** with the thread ID of the newly created thread
  - The **attr** argument can be used to customize various thread attributes, setting it to NULL creates a thread with the default attributes
  - Returns 0 if successful, error number on failure

Thread Termination – Pthreads

- We can terminate threads with the function:

```
void pthread_exit(void *rval_ptr);
```

  - Terminates thread execution with return code **rval_ptr**, without terminating the entire process (the same happens when the thread returns from the start routine)
  - Note however that, if a thread calls **exit()**, then the entire process terminates
  - Returns 0 if successful, error number on failure

Thread Termination – Pthreads

- We can wait for specific threads with the function:

```
int pthread_join(pthread_t tid, void **rval_ptr);
```

  - Waits for a specific thread **tid** to terminate and blocks until the specified thread calls pthread_exit() or returns from its start routine
  - The thread return code is then made available in the **rval_ptr** argument
  - Returns 0 if successful, error number on failure

Thread Example – Pthreads

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

int sum; /* data shared by thread(s) */
void *runner(void *param);


/* threads call this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

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

   pthread_exit(0);
}
```

```
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);
}
```

Semantics of exec() and fork()

- What happens if one thread in a multithreaded process calls the **exec()** or **fork()** system calls?
- The **exec()** system call works as usual and the program specified in the call will **replace the entire process – including all threads**
- The **fork()** system call is typically implemented using two versions:
  - One that creates a **new multithreaded process** by **duplicating all threads in the calling process**
  - Another that creates a **new single-threaded process** by **duplicating only the calling thread**