Threads

Threads

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Processes: Characteristics

- A process may execute other processes through
  - Cloning (UNIX, `fork`)
  - Replacing the current image with another image (UNIX, `exec`)
  - explicit call (Windows, `CreateProcess`)
- A process has
  - Its own address space
  - A single execution thread (a single program counter)

Processes: Limits

- Synchronization and data transfer
  - No cost or minimal for (almost) independent processes
  - High cost for cooperating processes
- Cloning involves
  - A significant increase in the memory used
  - Creation time overhead
- Management of multiple processes requires
  - Scheduling
  - Expensive context switching operations

Standard process = **heavyweight process**
A task with a single thread of execution

From processes to threads

- There are several cases where it would be useful to have
  - Lower creation and management costs
  - A single address space
  - Multiple execution threads within that address space
- Example
  - WEB applications
    - A server must responds quickly to many access requests
    - The requests are submitted at the same time, and require similar processing of the data, etc.

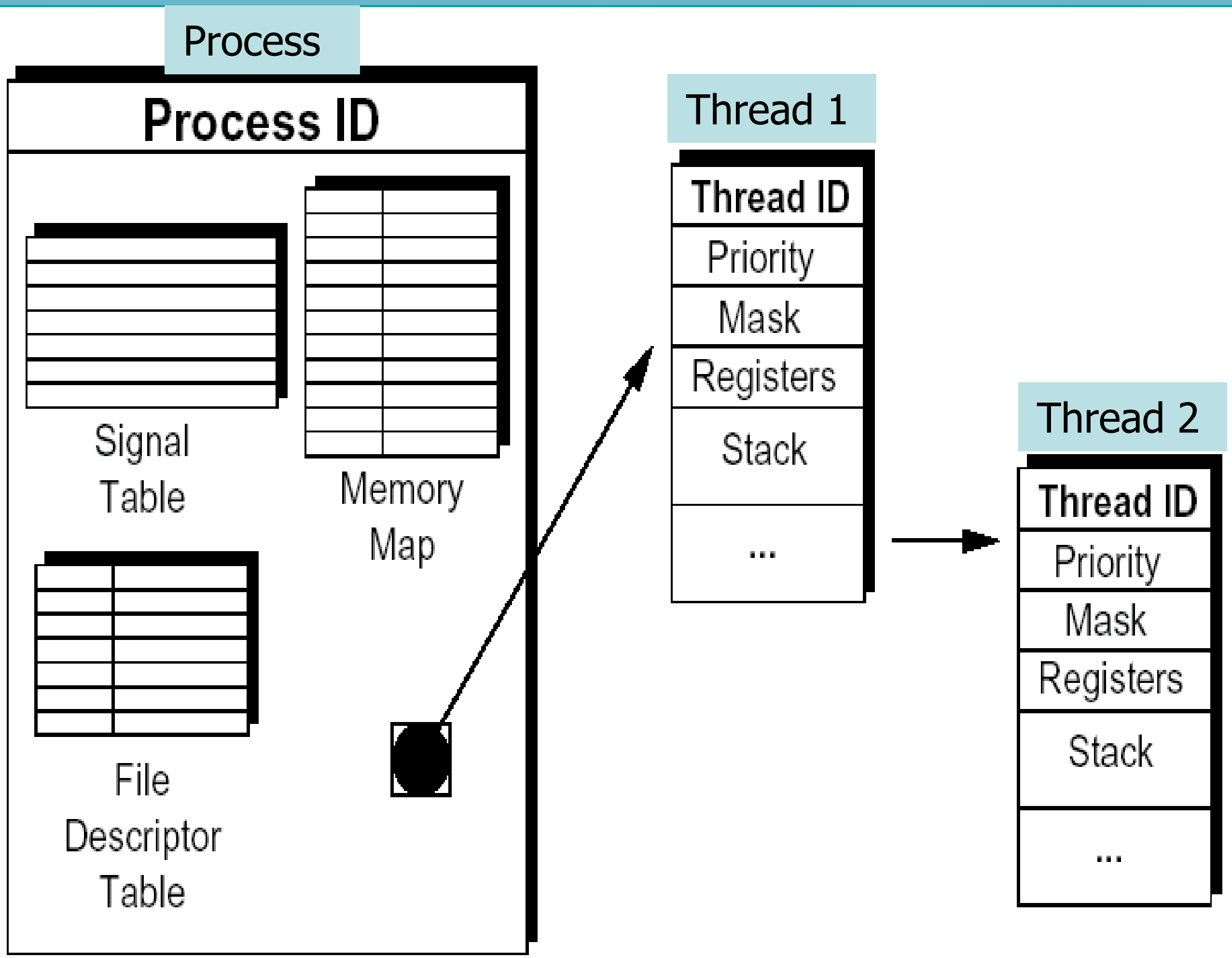
From processes to threads

- The 1003.1c POSIX standard introduces the concept of **threads**
- The thread model allows a program to control multiple different flows of operations that overlap in time.
- Each flow of operations is referred to as a **thread**
- Creation and control over these flows is achieved by making calls to the POSIX Threads API.
- A thread can share its address space with other threads

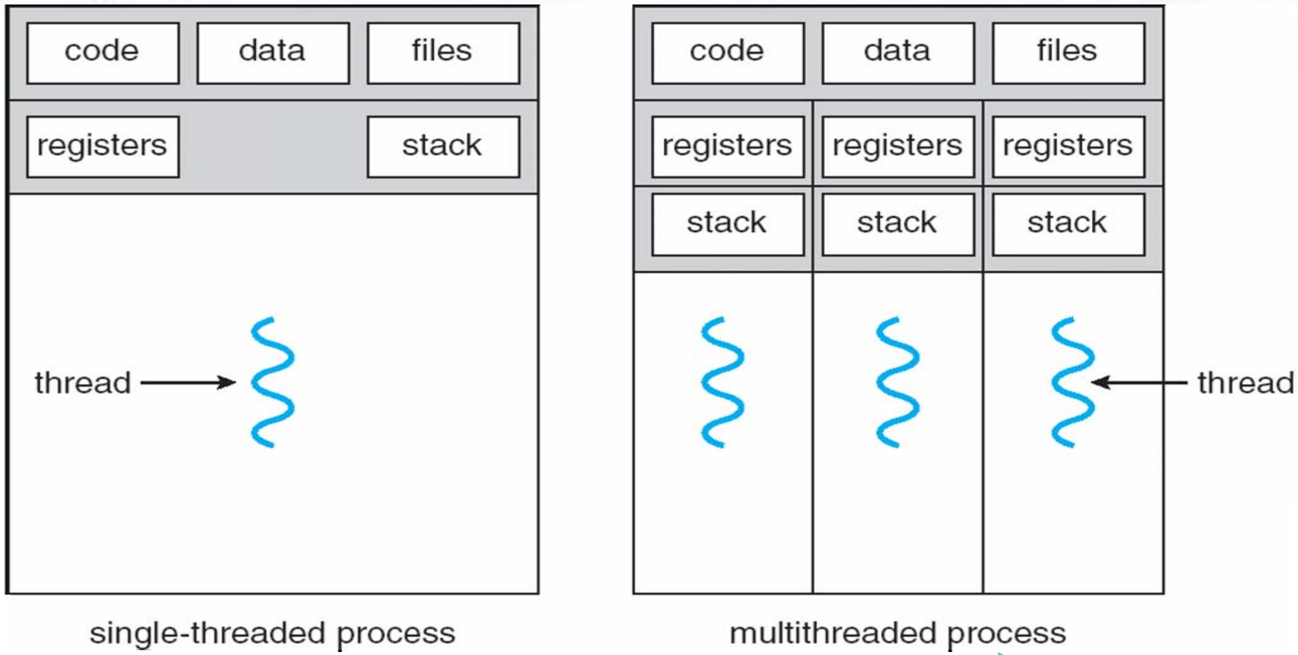
From processes to threads

- The **process** is the owner of the resources that are uses by all its threads
- The **thread** is the basic unit of CPU utilization (and scheduling)
- Thread is also called a **lightweight process**

From processes to threads



From processes to threads



A process with a single thread

A process with three threads
Sharing requires protection !

Threads: Pros

- The use of threads allows
  - Shorter response time
    - Creating a thread it is 10-100 times faster than creating a process
    - Example
      - To create 50000 jobs (**`fork`**) takes 10 seconds (real time)
      - To create 50000 thread (**`pthread_create`**) takes 0.8 seconds (real time)
  - Shared resources
    - Processes can share data only with special techniques
    - Threads share data automatically

Threads: Pros

- Lower costs for resource management
  - Allocate memory to a process is expensive
  - Threads use the same section of code and/or data to serve more clients
- Increased scalability
  - The advantages of multi-threaded programming increase in multi-processor systems
  - In multi-core systems (different calculation units per processor) threads allow easily implementing concurrent programming paradigms based on
    - Task separation (pipelining)
    - Data partitioning (same task on data blocks)

Threads: Cons

- Since threads of the same process run in the same address space they must by synchronized to properly access shared data

Concurrency with threads

- Optimize the following code segment that performs the scalar product of two huge dimension vectors, which have been independently created by two processes or by two threads

```
for (i=0; i<n; i++) {
  v[i] = v1[i] * v2[i];
}
```

- Sharing data would be ineffective for processes, but is inexpensive for threads

Concurrency with threads

Data partition with a divide-and-conquer strategy

```
mult (a, b) {
  for (i=a; i<b; i++)
    v[i] = v1[i] * v2[i]
}
...
CreateThread (mult, 0, n/2);
CreateThread (mult, n/2, n);
```

A thread perform its task on its partition of the data

Care has to be taken to avoid

- the use of non-reentrant procedures
- the use of non-reentrant library functions
- access to common variables, etc.

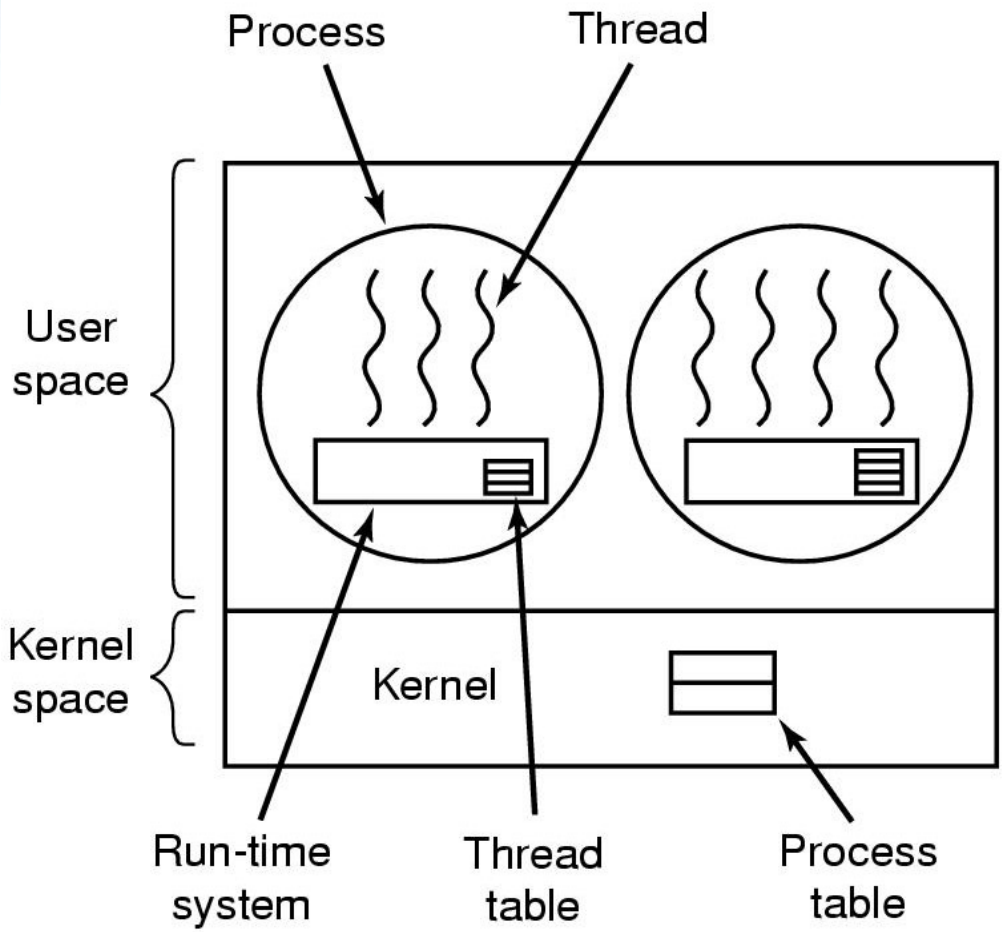
Multithread programming models

- Three multithread programming models exist
  - User-level thread
    - Thread implemented at user-level
    - The kernel is not aware that threads exist
  - Kernel-level thread
    - Thread implemented at kernel-level
    - The kernel directly supports the thread concept
  - Mixed or hybrid solution
    - The operating system provides both user-level and kernel threads

User-level threads

- The thread package is fully implemented in the user space, as a set of functions
  - These function calls, at user level, the standard system libraries
  - Each process has a its own thread table, which is managed by the thread package functions
  - The kernel is **not aware about** threads, it manages only processes

User-level threads



User-level threads

Advantages

- Efficient management because it is carried out at the user level
- Fast context switching between threads of the same task
- Can be implemented on top of any kernel
- Allow the programmer to generate a large number of threads
- Possibly, you can implement your own custom scheduling strategy

User-level threads

❖ Disadvantages

- There is only a single thread running per process task even in a multiprocessor system
- The scheduler maps all the user-level threads of a process to a the single kernel thread of that process
- If a thread makes a blocking system call, all threads of the same process are blocked
- There is no true parallelism

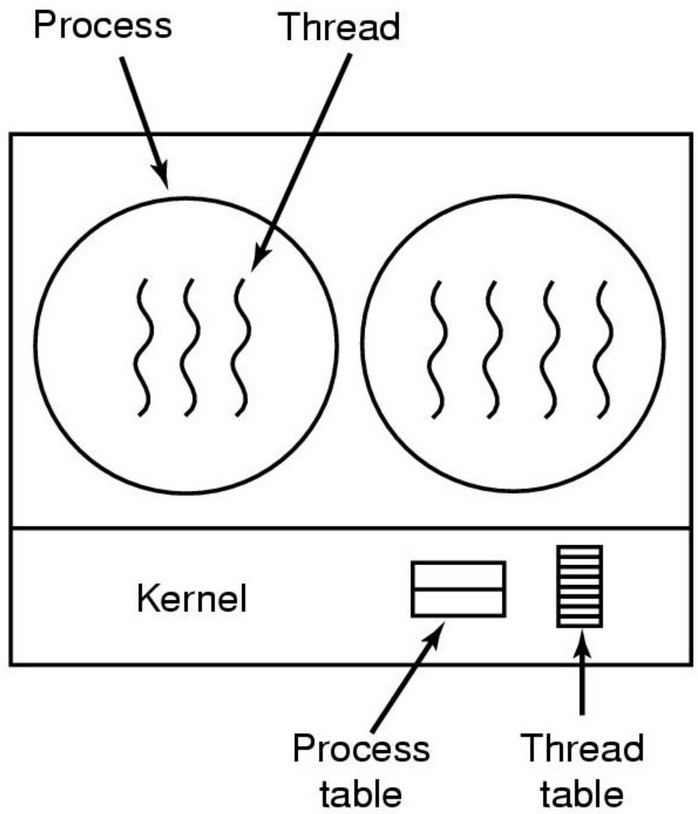
User-level threads

❖ Disadvantages

- There is not automatic scheduling of the threads of a process
- Either a thread explicitly pass CPU control to other threads of the same process,
- or a scheduler is implemented, at user-level, using the kernel timer

Kernel threads

- Threads are directly managed by the kernel



Kernel threads

❖ Advantages

- A ready thread can be scheduled even if another thread of the same process has called a blocking system call
  - If a thread is blocked, the kernel scheduler can select another thread among the ready threads of all processes
- In a multiprocessor system multiple threads of a process can be executed in real concurrency

Kernel threads

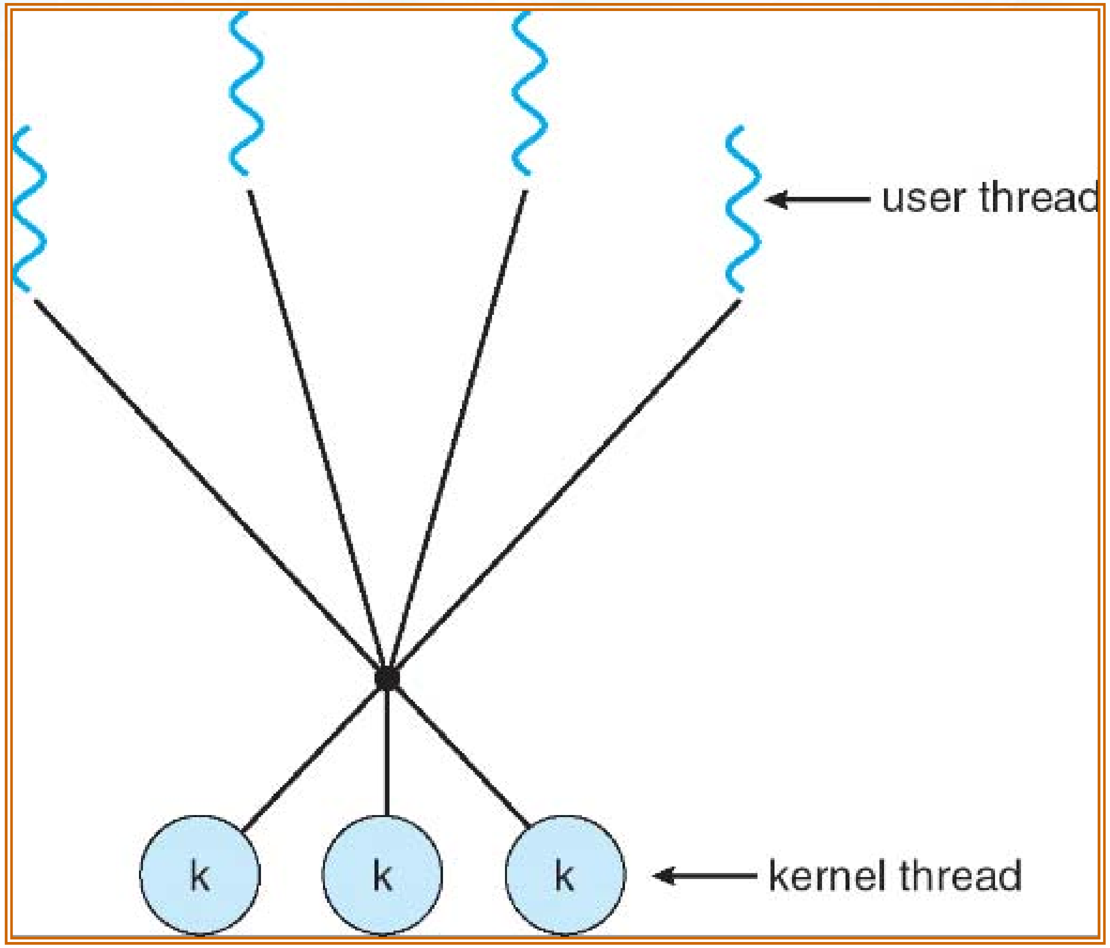
❖ Disadvantages

- Context switching more expensive because it requires passing though the kernel mode, and kernel support
- The maximum number of threads per task/system is limited, due to the use of kernel memory structures

Hybrid implementation

- One of the multi-thread programming problems is to define the relationship between user threads and kernel threads
  - Virtually all modern operating systems have a kernel thread
    - Windows, UNIX, Linux, MAC OS X, Solaris
  - The basic idea is to have m user threads and to map them to n kernel threads
  - Typically, n<m

Hybrid implementation



Hybrid implementation

- The hybrid implementation attempts to combine the advantages of both approaches
  - The user decides the number of its user-level threads, and the number of kernel-threads on which they must be mapped
  - The kernel is aware only of the kernel thread and only manages those threads
  - Each kernel thread can be used in turn by several user threads

Processes and threads coexistence

- Several problems arise due to the coexistence of processes and threads
  - Using the system call fork
    - A fork duplicates only the thread that makes the call, or all the threads of the process?
  - Example
    - P has two threads T1 and T2
    - T1 is waiting on a read, while T2 performs a `fork`
    - Now we have P and his child, both with T1 waiting on a read
    - Which thread T1 will receive data? Only one? Which? Both?

Processes and threads coexistence

- Using the system call **exec**
  - Does the **exec** replace only the calling thread with the new process, or all threads?
  - Some systems provide different versions of the system call **fork**
  - **forkall**, duplicates all process threads
  - **fork1**, duplicates only the calling thread
- Signal management
  - If a process receives a signal which thread will catch it?
  - What happens if multiple threads indicate their interest to catch a signal? Which will handle the signal?