Threads

Threads

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

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 (with kernel intervention)

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 (concurrency) 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 (**scheduled and executed independently**) 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

Process = unit that groups resources
Thread = CPU scheduling unit

Thread = **hlightweight process**

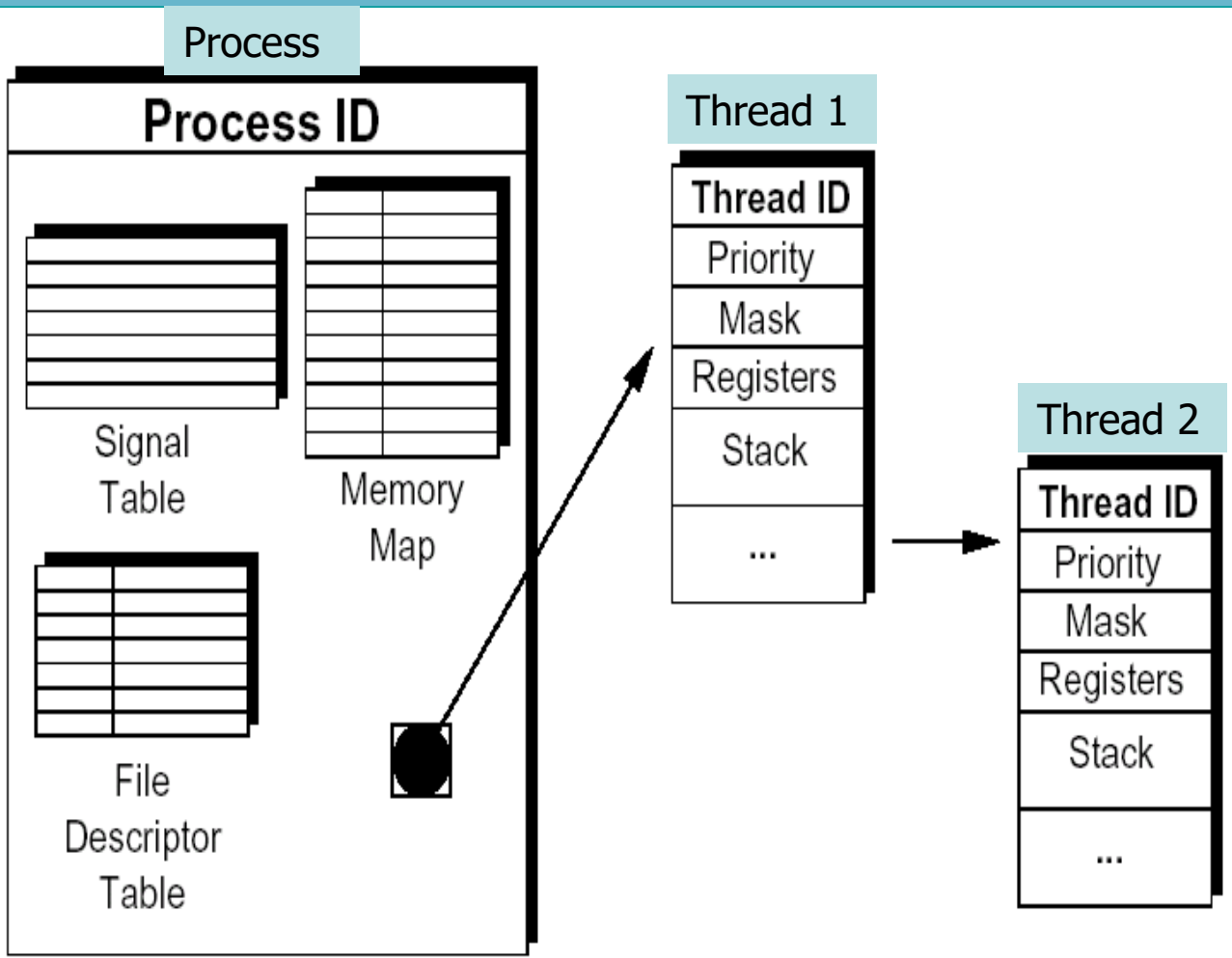
From processes to threads

- Shared data
  - Code section
  - Data section (variables, file descriptors, etc.)
  - Operating system resources (e.g., signals)
    - i.e., shared data: static variables, external variables, dynamics variables (heap)
- Private data
  - Program counter and hardware registers
  - Stack, i.e., local variables and execution history

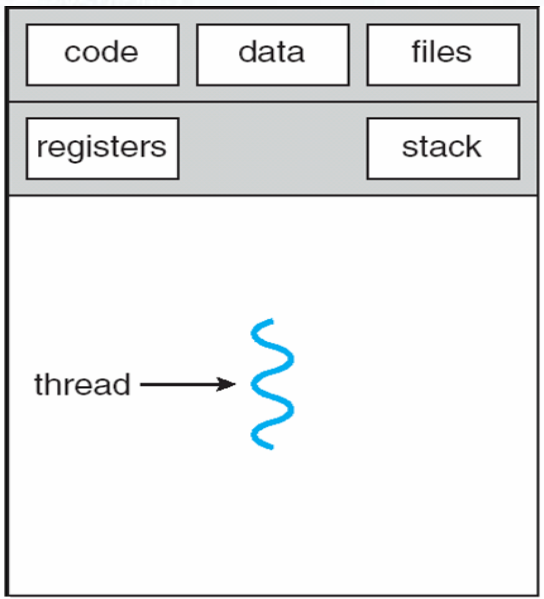
... with the other threads of the same process ...

Obvious, since a thread implies its own flow of execution (within the same process)

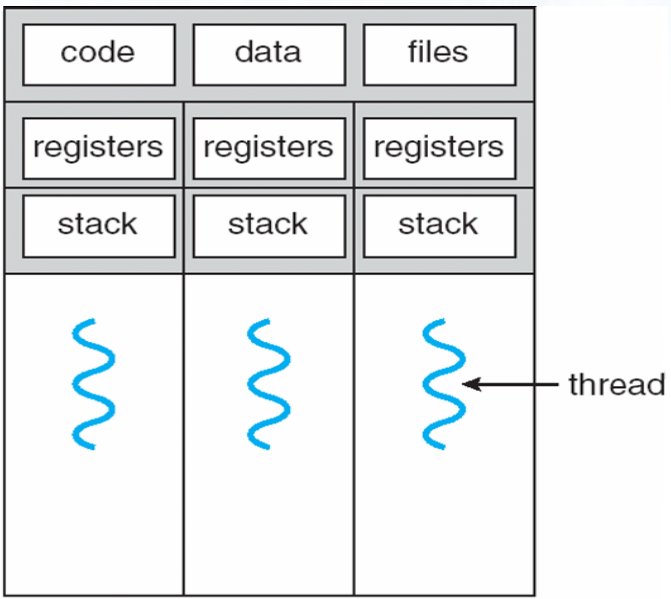
From processes to threads



From processes to threads



single-threaded process



multithreaded process

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
      - The creation of 50000 jobs (`fork`) takes 10 seconds (real time)
      - The creation of 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: Advantages

- 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: Disadvantages

- There is no protection for threads
  - They are executed in the same address space and OS protection is impossible or unnecessary
  - If the threads are not synchronized, access to shared data is not thread safe
- There is not a parent-child hierarchical relationship between threads
  - To the creating thread is normally returned the identifier of the created thread, but this does not imply a hierarchical relationship
  - All threads are "equal"

Concurrency with threads

- Optimize the following code segment that performs the scalar product of four huge dimension vectors (v1, v2, v3, v4)

```
for (i=0; i<n; i++) {
  v[i] = v1[i] * v2[i] + v3[i] * v4[i];
}
```

With processes, data sharing would be expensive and prevent its use

With threads, data sharing is automatic and concurrency is immediate

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] + v3[i] * v4[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
  - Kernel-level thread
    - Thread implemented at kernel-level
    - The kernel directly supports the thread concept
  - User-level thread
    - Thread implemented at user-level
    - The kernel is not aware that threads exist
  - Mixed or hybrid solution
    - The operating system provides both user-level and kernel threads

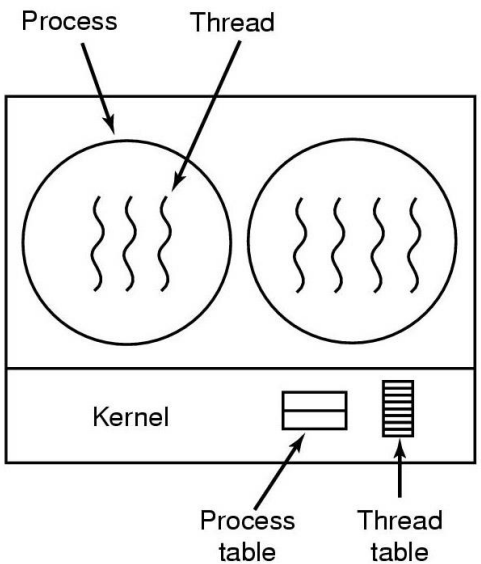
The choice is moderately controversial

Kernel-level threads

- Threads are managed by the kernel
- The OS
  - Manipulates both processes and threads
  - Is aware of the existence of threads
  - Provides adequate support for their handling
    - All the operations on threads (creation, synchronization, etc.) are performed through **system calls**

Kernel-level threads

- The operating system, for each thread, keeps information similar to those it maintains for each process
  - Thread table
  - Thread Control Block (TCB) for each active thread
- Managed information is "global" within the whole OS



Kernel-level threads

- **Advantages**
  - Since the operating system is aware of threads, it can select
    - Thread to schedule among the ready threads of all processes
      - Global view of all threads of all processes
    - Possibly allocating more CPU time to processes with many threads than to processes with few threads

Kernel-level threads

- Effectiveness in applications that perform often blocking calls (e.g., blocking read)
  - Ready threads can be scheduled even if they belong to the same task of a thread that called a blocking system call
  - That is, if one thread blocks, it is always possible to execute another thread in the same process (or in another) because the OS checks all the threads of all the processes
- It allows an effective parallelism
  - Multiple threads can be executed in a multiprocessor system

Kernel-level threads

Already mentioned for processes (motivation for the introduction of threads)

❖ Disadvantages

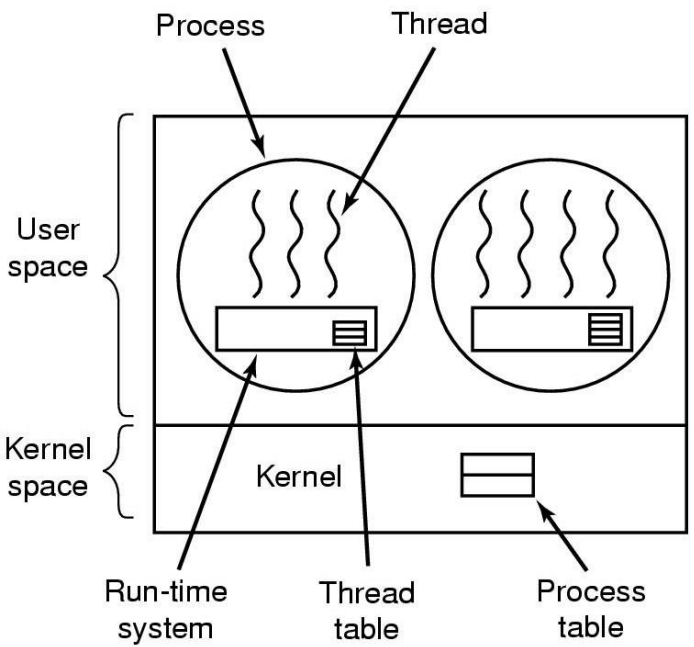
- Due to the transition to kernel mode the management is relatively slow and inefficient
  - Expensive context switching
  - Handling times hundreds of times slower than necessary
- Limitation on the maximum number of threads
  - The OS must control the number of generated threads
- Expensive information management (thread table and TCB)

User-level threads

- The thread package is fully implemented in the user space, as a set of functions
  - The kernel is **not aware about** threads, it manages only processes
  - Threads are managed run-time through a **library**
    - Support by means of a set of functions, called from user-space
    - Thread creation, synchronization, scheduling, etc. do not require kernel intervention
    - Function are used, **not** system calls

User-level threads

- Each process needs a personal table of running threads
  - Needed information is less than the management at the kernel-level
    - Smaller TCBs
    - Local visibility of information (within the process)



User-level threads

- **Advantages**
  - Can be implemented on top of any kernel, even in systems that do not support threads natively
  - Do not require modifications to the OS
  - Efficient management
    - Fast context switching between threads of the same task
    - Efficient data manipulation
    - Hundreds of times faster than kernel threads
  - Allow the programmer to generate the desired number of threads
    - It might be possible to think of scheduling/custom management of threads within each process

User-level threads

Disadvantages

- The operating system does not know the existence of threads
- Inappropriate or inefficient choices can be made
  - The OS could schedule a process whose running thread could do a blocking operation
  - In this case, the whole process could be blocked even if inside it several other threads could be executed

User-level threads

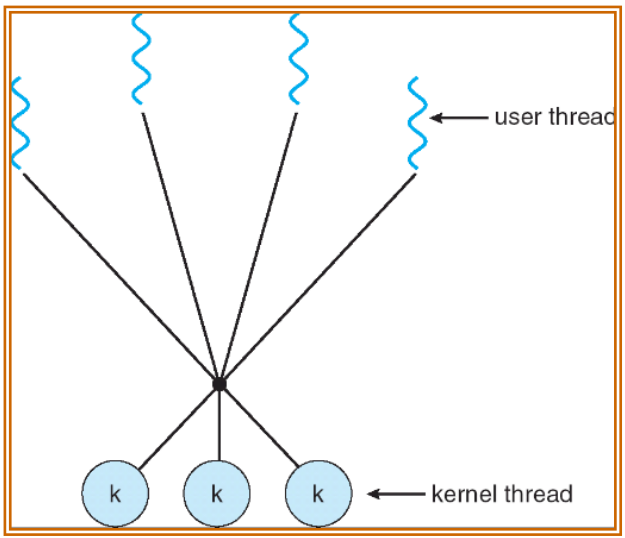
- Information should be communicated between kernel and run-time user-level manager
- Without this communication mechanism
  - Exist only on running thread for each task even in a multiprocessor system
  - There is no scheduling within a single process, i.e., interrupts do not exist within a single process
  - If a running thread does not release the CPU, it cannot be blocked

User-level threads

- The scheduler must map user threads to the single kernel thread
  - If the kernel thread blocks, all the user-level threads are blocked
  - There is no true thread-level parallelism without handling multiple threads at the kernel-level

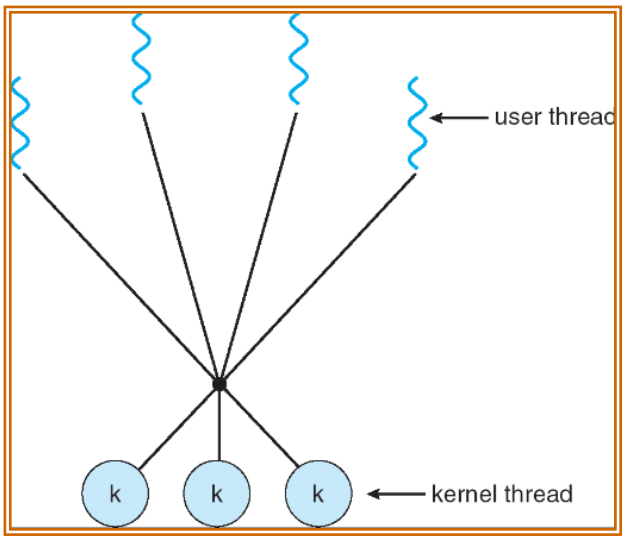
Hybrid implementation

- One of the multi-thread programming problems is to define the relationship between user-level threads and kernel-level threads
- Practically all modern OSs 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

- 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 its 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?
- Signal management
  - If a process receives a signal which thread will catch it?
    - A thread should indicate its interest in handling the signal. What happens if multiple threads indicate their interest to catch a signal? Which will handle the signal?
- Some systems provide different versions of the system call **`fork`**
  - **`forkall`**, duplicates all process threads
  - **`fork1`**, duplicates only the calling thread