1 Threads

1.1 Theads from an OS perspective

A proces consists of two **fundamental units**

- **Resources**: all related resources are grouped together:
  - A logical addres space containing the process image (program, data, heap stack)
  - Files, I/O devices, I/O channels
- **Execution trace**, i.e, an entity that gets exceuted

A process can **share its resources** between **multiple execution traces**. Every thread has its own **execution context** (e.g program counter, stack, registers).
All threads have **access** to the process **shared resources**

- E.g files, one thread opens a file, all threads on the same process can acess the file
- Global variables, memory, etc. Which is needed or synchronisation

Some CPUs (hyperthreaded ones) have direct **hardware support** for **multi-threading**. They can offer up to 8 hardware threads per core.

Similar to processes, threads have:

- **States** and **transitions** (new, running, blocked, ready, terminated)
- **A thread control block**

Threads incur **less overhead** to create/terminate/switch (address space remains the same for threads of the same process)

Processes	Threads
Address space	Program Counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	Local vars
Signals and signal handlers	
Accounting information	

- **Inter-thread communication** is easier/faster than **interprocess** communication (threads share memory by default)
- **No protection boundaries** are required in the address space. Threads are cooperating, belong to the same user, and have a common goal)
- **Synchronisation** has to be considered carfully

1.2 Why use threads

Multiple **related activites** apply to the **same resources**, these resources should be accessible/shared
Processes will often contain multiple **blocking tasks**

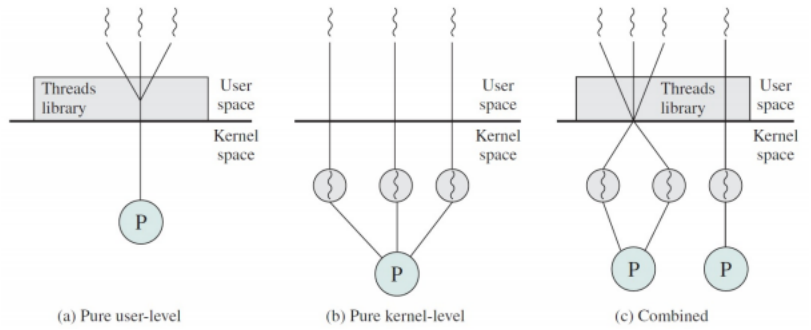
- I/O operations (thread blocks, **interrupt** marks completion)
- Memory access: pages faults are result in blocking

Such activities should be carried out in **parallel/concurrently**. **Application examples**: webservers, make program, spreadsheets, word processors, processing large data volumes

1.3 Thread types

- **User thread N:1**: To make threads **cheap and fast**, they need to be implemented at **user level**. User-Level threads are managed entirely by the **run-time system** (user-level library).The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.User-Level threads are **small and fast**, each thread is represented by a PC, register,stack, and small thread control block.
  - **Advantages**
    - A user-level threads package can be implemented on an Operating System that does not support threads.
    - User-level threads do not require modification to operating system
    - **Simple Representation**: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
    - **Simple Management**: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
    - **Fast and Efficient**: Thread switching is not much more expensive than a procedure call.
  - **Disadvantages**
    - User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, Os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock.
    - Lack of coordination between threads and operating system kernel. Therefore, process as whole gets **one time slice** irrespect of whether process has one thread or 1000 threads within.
    - User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.
- **Kernel thread 1:1**: In this method, the kernel **knows** about and **manages** the threads. **No runtime system is needed** in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. Kernel-Level threads **make concurrency much cheaper** than process because, much less state to allocate and initialize.
  - **Advantages**
    - Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
    - Kernel-level threads are especially good for applications that frequently block.
  - **Disadvantages**
    - The kernel-level threads are **slow and inefficient**. For instance, threads operations are hundreds of times slower than that of user-level threads.
    - Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to **maintain information** about threads. As a result there is significant **overhead** and increased in kernel complexity.
- **Hybrid implementation M:N** maps some M number of application threads onto some N number of kernel entities or "virtual processors".In general, *M:N* threading systems are more **complex** to implement than either kernel or user threads, because changes to **both kernel and user-space code are required**. In the *M:N* implementation, the threading library is responsible for **scheduling** user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls.
  - **Advantages**
    - Can take advantage of multiple CPUs or multiple CPU cores. And if one task blocks, you can create another kernel thread to use the available CPU more efficiently.
    - Less context switches, which increases performance (in the ideal case, where you have as many running threads as you have processors, you may have almost no context switches).
  - **Disadvantages**
    - Possibly bigger latency: if all the threads in the pool are busy and you add new short task, you may wait a long time before it starts executing.

1.4 Comparison



2 Libraries

2.1 Thread management

Thread libraries provide an **API/interface for managing threads**. Can be implemented:

- Entirely in **user space**
- Basedon **system calls**, i.e., rely on the kernel for thread implementations

Examples of thread APIs include **POSIX's PThreads**, Windows Threads and Java threads

- The PThread specification can be implemented as user or kernel threads

2.2 POSIX Threads

POSIX threads are **specification** that **"anyone" can implement**. It definines a set of APIs (function calls, over 60) and what they do