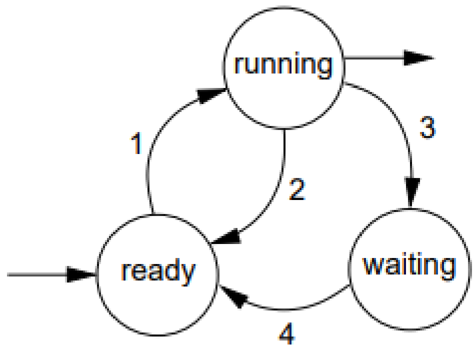
Processing and Scaling

Threads

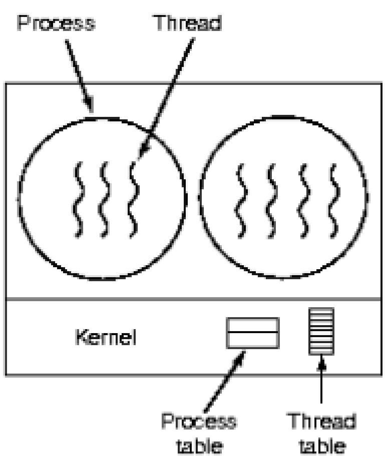
- Abstract the execution of a sequence of instructions
- Threads of a given process may share most resources, except the stack and the processor state
- **Thread-specific information**: State, Processo State (including the SP and PC) and Stack
- Operations like creation/termination and switching on threads of the same process are much more efficient than the same operations on processess

Thread States



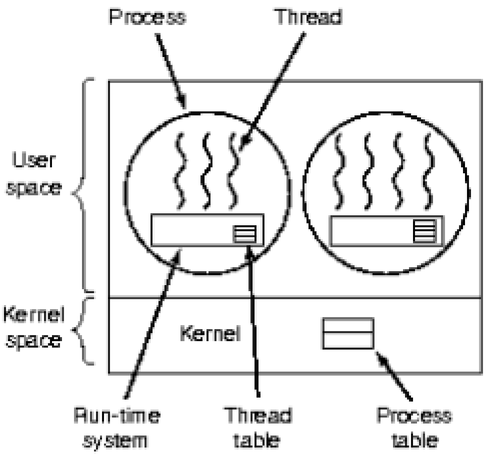
Implementation

Kernel-level Threads



- Implemented directly by the OS
- The kernel supports processes with multiple threads
- OS keeps a *threads table* with information on every thread
- All thread management operations incur a system call

User-level Threads



- Implemented by user-level code (e.g. a library)
- The kernel is not aware of the existence of threads at user-level

Threads' Library

Must provide functions for:

- thread creation/destruction
- thread synchronization
- yield a core to other threads
  Responsible for thread switching and keeps a threads' table
  Wrapper-functions of some systems call that may block have to be modified

Advantages

- The OS needs not support threads
- The kernel is not involved in most operations.

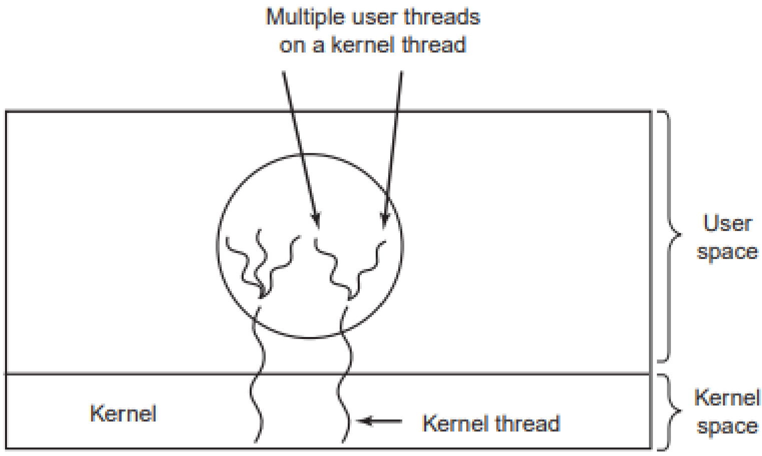
Disadvantages

- Page-fault by one thread will prevent the other threads from running
- Cannot be used to exploit parallelism in multicore architectures

Hybrid Implementation

Idea: multiplex user-level threads on kernel-level threads

- The kernel is not aware of the existence of user-level threads
  - User-level scheduler **gives hints** to the kernel-level schedular
  - Kernel-level scheduler **notifys** the user-level schedular about its decisions
- The library maps the user-level threads to kerner-level treads
  - Nr. user-level threads >> Nr. kernel-level threads

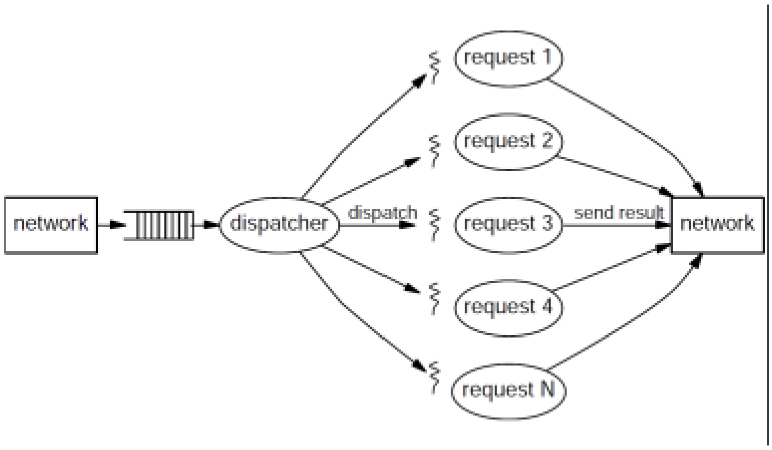


Multi-threaded Server

- Each thread processes a request
- When one thread blocks on I/O → another thread may be scheduled to run in its place

Common pattern:

- **One Dispatcher**: accepts a connection request
- **Several Workers**: process all the requests sent in the scope of a single connection



Boudning threads' resource usage

Thread-Pools

- Allow to bound the number of threads (bound if you use multiple-thread pools to avoid **excessive thread-switch overhead**)
- Avoid thread creation/destruction overhead (with fixed and at least a minimum nr of threads)

Synchronous vs. Asynchronous I/O

Synchronous I/O

- **Blocking**: thread blocks until the operation is completed
- **Non blocking**: the thread never blocks, not even in input operations

Asynchronous I/O

The system call just enqueues the I/O request and returns immediately

- The thread may execute while the requested I/O operation is being executed
- The thread learns about the termination of the I/O operation (polling or via event notification e.g. signals in Unix/Linux)

`poll()`/`epoll()` and Blocking I/O

Scenatio: Witch TCP, servers use one data socket per connection/client

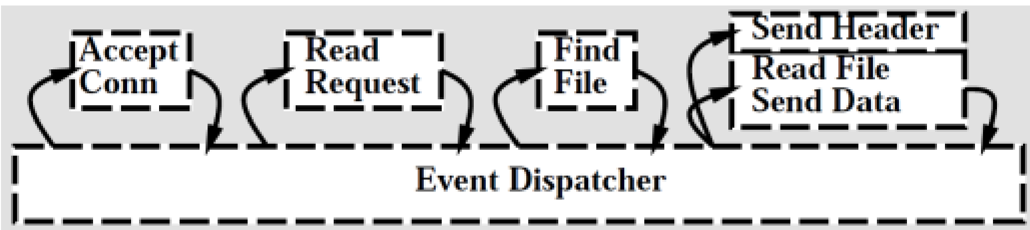
To use fewer threads than data sockets:

- Use `select()`/`poll()`/`epoll()` blocks until:
  - One of the requested events occurs
  - The timeout expires
  - **Note:** doesn't work with regular files (use helper threads for disk I/O)

Operation Termination

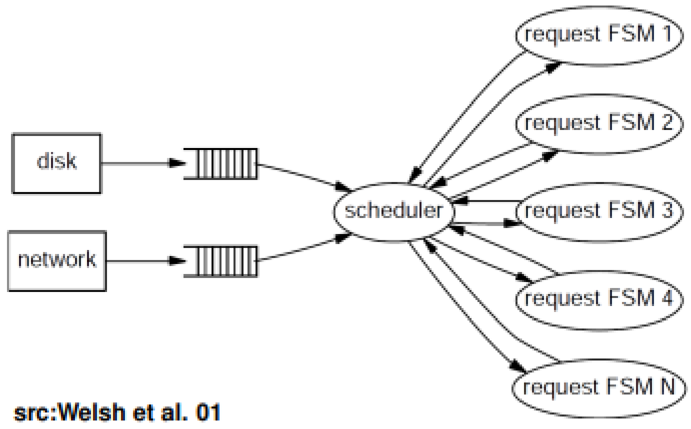
- Polling (`SIGEV_NONE`)`
- Notification:
  - Signal (`SIGEV_SIGNAL`): must register the corresponding handler
  - Function (`SIGEV_THREAD`): executed by a thread created for that purpose

Event-driven Server (State Machine Approach)



- Server executes a loop, in which it:
  - waits for events (usually I/O events)
  - processes these events (sequentially, but may be not in order)
- Blocking is avoided (non-blocking I/O operations)

- Keep FSM for each request



Scalability Issues

- **Data Copying**, expecially in network protocols
  - use buffer descriptors
  - use scatter/gather I/O (`readv()`/`writev()`)
- **Memory allocation**
  - design allocator which can pre-allocate a pool of memory buffers and avoid freeing them
- **Concurrency protocol**
  - avoid sharing
  - locking granularity
    - too coarse: false sharing and unnecessary blocking
    - too fine grained: may lead to deadlocks
  - minimize the duration of critical sections
- **Kernel/protocol tuning**