Processing and Scaling

October 29, 2021

Roadmap

Roadmap

Latency Numbers Everyone Should Know

Threads

Threads Implementation

Multi-threaded Server

Synchronous vs. Asynchronous I/O

Event-driven Server

Latency (2009)

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

src:Jeffrey Dean presentation at LADIS09

Latency (2019)

Table 2.3: Latency numbers that every WSC engineer should know. (Updated version of table from [Dea09].)

Operation	Time
L1 cache reference	1.5 ns
L2 cache reference	5 ns
Branch misprediction	6 ns
Uncontended mutex lock/unlock	20 ns
L3 cache reference	25 ns
Main memory reference	100 ns
Decompress 1 KB with Snappy [Sna]	500 ns
"Far memory"/Fast NVM reference	1,000 ns (1us)
Compress 1 KB with Snappy [Sna]	2,000 ns (2us)
Read 1 MB sequentially from memory	12,000 ns (12 us)
SSD Random Read	100,000 ns (100 us)
Read 1 MB bytes sequentially from SSD	500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network	1,000,000 ns (1 ms)
Read 1 MB sequentially from disk	10,000,000 ns (10 ms)
Disk seek	10,000,000 ns (10 ms)
Send packet California→Netherlands→California	150,000,000 ns (150 ms)

src:Barroso, Hölze and Raganathan 2019

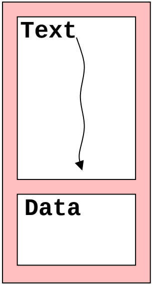
Roadmap

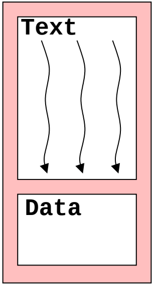
Threads

Threads abstract the execution of a sequence of instructions, i.e. a thread of execution

Simplifying, whereas a process abstracts the execution of a program, a thread abstracts the execution of a function

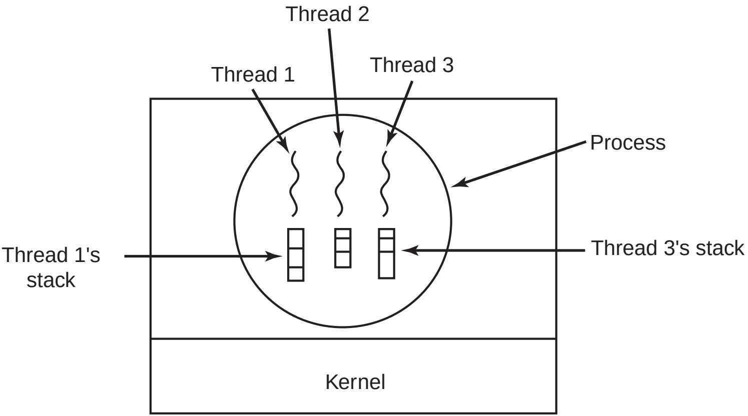
- In today's OSs, a process provides an execution environment for more than one thread.





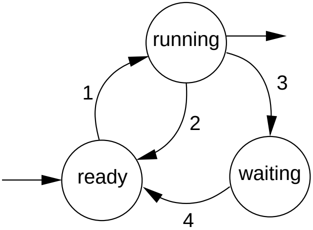
Resource Sharing with Threads

- Threads of a given process may share most resources, except the stack and the processor state:



Thread State

- Like a process, a *thread* may be in one of 3 states:



- Thread-specific information is relatively small:
  - its state (e.g. a process may be blocked waiting for an event
  - the processo state (incluing the `SP` and `PC`);
  - a stack.
- Operations like:
  - creation/termination
  - switching

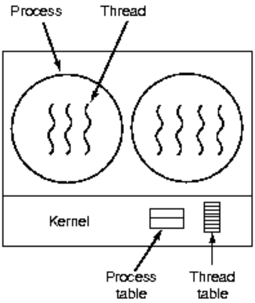
  on threads of the same process are much more efficient than the same operations on processes

Roadmap

Threads Implementation

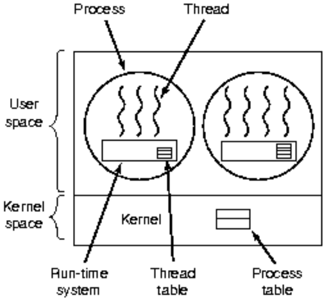
- Threads can be implemented:
  1. directly by the OS (*kernel-level threads*);
  2. by user-level code, e.g. a library, (*user-level threads*).

Kernel-level Threads



- The kernel supports processes with multiple threads:
  - The kernel’s scheduler allocates cores to threads
- The OS keeps a *threads table* with information on every thread
  - Usually a process’ control block points to its own threads’ table
- All thread management operations, such as thread creation, incur a system call

User-level Threads



- The kernel is not aware of the existence of threads at user-level:
  - Threads are implemneted by *user-space* library
  - The OS needs not support threads

User-level Threads' Implementation

- The threads' library must provide functions for:
  - thread creation or destruction;
  - thread synchronization;
  - yield a core to other threads.
- The library is responsible for thread switching and keeps a thread's table
- The wrapper-functions of some system calls that may block, have to be modified
  - To preven other threads from blocking
- Some issues:
  - how to make non-blocking system calls?
  - what about *page-faults*?
  - how to prevent a thread from never yielding a CPU/core?

User-level vs. Kernel-level Threads

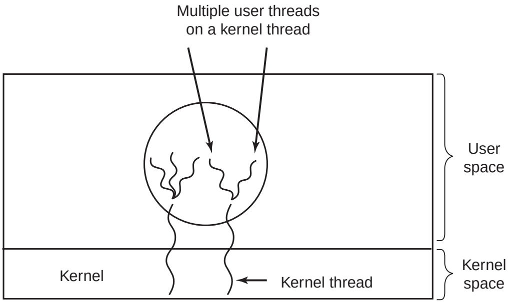
+ The OS needs not support threads
+ The kernel is not involved in most operations. E.g.:
  - Thread creation/destruction
  - Thread switching

– *Page-fault* by one thread will prevent the other threads from running (the single kernel-level thread is put in the WAIT state

– Cannot be used to exploit parallelism in multicore architectures

Hybrid Implementation (m:n)

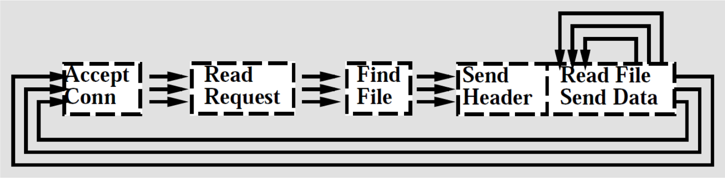
Idea: multiplex user-level threads on kernel-level threads



- The kernel is not aware of the existence of user-level threads. Actually for better results:
  - The user-level scheduler should give hints to the kernel-level scheduler
  - The kernel-level scheduler should notify the user-level kernel about its decisions
- The library maps user-level threads to kernel-level threads
  - The number of user-level threads may be much larger than that of kernel-level threads

Roadmap

Multi-threaded Server



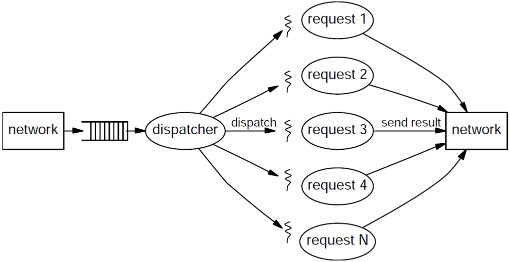
src:Pai et al. 99

- Each thread processes a request (and HTTP 1.0 connection)
- When one thread blocks on I/O
  - Another thread may be scheduled to run in its place.

- A common pattern is:

  One dispatcher thread, which accepts a connection request

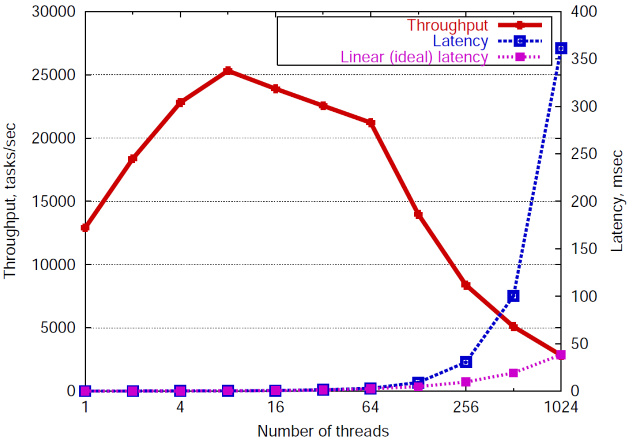
  Several worker threads, each of which processes all the requests sent in the scope of a single connection



src:Welsh et al. 01

Latency

- Same file 8 KB reads (no disk accesss)
- No thread creation
- "4-way 500MHz Pentium III with 2 GB memory under Linux 2.2.14"



src: Welsh et al. 01

Bounding threads' resource usage

Thread-Pools

- Allow to bound the number of threads
- Avoid thread creation/destruction overhead
  - If you use a fixed number of threads
  - Or at least a minimum number of threads, that is large enough
- Supported by several packages
  - E.g `java.util.concurrent` supports both `ThreadPoolExecutor` and `ScheduledThreadPoolExecutor`

Excessive Thread-Switch Overhead

- This arises more often if you use multiple-thread pools
- In this case, you may want to bound the number of active threads
  - E.g. using a counting semaphore.
    - Initialize the counting semaphore with the number of cores (plus some slack) and call:

      `down()` either at a beginning of a task or after a potentially blocking call

      `up()` either at the end of a task or before a potentially blocking call

Roadmap

Latency Numbers Everyone Should Know

Threads

Threads Implementation

Multi-threaded Server

Synchronous vs. Asynchronous I/O

Event-driven Server

Synchronous vs. Asynchronous I/O

Synchronous I/O Can have two modes

Blocking The thread blocks until the operation is completed.

- `write()/send()` system calls may return immediately after copying the data to kernel space and enqueueing the output request

Non blocking The thread does not block

- Not even in input operations: the call returns immediately with whatever data is available at the kernel
- In Unix, all I/O to block devices (and regular files or directories) is blocking, even if you set the `O_NONBLOCK` flag

Asynchronous 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
  - either by polling
  - or via event notification (signals in Unix/Linux)

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

Scenario With TCP, servers use one data socket per connection/client

Question Can we use fewer threads than data sockets?

Answer Yes: use `select()/poll()/epoll()`

```
struct pollfd {
    int   fd;          /* file descriptor */
    short events;      /* requested events */
    short revents;     /* returned events */
};
int poll(struct pollfd *fds, nfds_t nfds, int timeout)
```

`poll()` blocks until:

- One of the requested events, e.g. data input (`POLLIN`), occurs
- The timeout (in ms) expires

Issue This does not work with regular files

- "Regular files shall always poll TRUE for reading and writing."
- A work-around is to use helper threads for disk I/O

POSIX Asynchronous I/O

- POSIX.1b specifies several functions for asynchronous I/O

```
int aio_read(struct aiocb *racbp);
int aio_write(struct aiocb *racbp);
int aio_cancel(int fd, struct aiocb *acbp);
ssize_t aio_return(const struct aiocb *acbp);
int aio_error(const struct aiocb *acbp);
```

- The asynchronous I/O operations are controlled by an AIO control block stucture (`struct aiocb`)

```
struct aiocb {
   int            aio_fildes;
   off_t          aio_offset; /* no file position */
   volatile void *aio_buf;
   size_t         aio_nbytes;
   struct sigevent aio_sigevent; /* on completion */
   int            aio_lio_opcode; /* for list op. */
   int            aio_reqprio; /* AIO priority */
}
```

Asynchronous I/O: Operation Termination

Problem How does the user process learn that the operation has terminated?

Solution There are two alternatives, which are specified in the `sigev_notify` member of the `struct sigevent`:

Polling (`SIGEV_NONE`) The process can invoke `aio_error()`

- It returns `EINPROGRESS` while it has not completed

Notification Here there are also some alternatives

Signal (`SIGEV_SIGNAL`) the signal is specified in field of the `struct sigevent` of of the `struct aiocb` argument

- Process must register the corresponding handler via the `sigaction()` system call

Function (`SIGEV_THREAD`) to be executed by a thread created for that purpose

Asynchronous I/O: `struct sigevent`

```
union sigval {           /* Data passed with notification */
    int     sival_int;          /* Integer value */
    void   *sival_ptr;          /* Pointer value */
};
struct sigevent {
    int          sigev_notify; /* Notification method */
    int          sigev_signo;  /* Notification signal */
    union sigval sigev_value;  /* Data passed with
                                  notification */
    void       (*sigev_notify_function) (union sigval);
                     /* Function used for thread
                        notification (SIGEV_THREAD) */
    void        *sigev_notify_attributes;
                     /* Attributes for notification thread
                      (SIGEV_THREAD) */
    pid_t        sigev_notify_thread_id;
                     /* ID of thread to signal (SIGEV_THREAD_ID)
};
```

Event-based Concurrency with `java.nio` package

Core classes

Channels There are several subclasses

Selector For blocking waiting for more than one I/O event from a selectable channel

Buffers To read/write data from/to channels

Issue `java.nio.channels.FileChannel` is not selectable

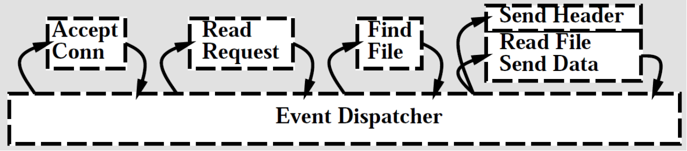
- To avoid blockin on file I/O need to use `java.nio.channels.AsynchronousFileChannel`, which supports asynchronous I/O
  - This is more complicated than non-blocking I/O
  - There is no `java.nio.channels.AsynchronousDatagramChannel`, although one can find references to it on the Web

Getting started with new I/O (NIO) Overview of Java I/O

- Refers to non-blocking I/O as asynchronous I/O, but they are not the same
- For (an even) more practical oriented tutorial you can checkout Java NIO Tutorial

Roadmap

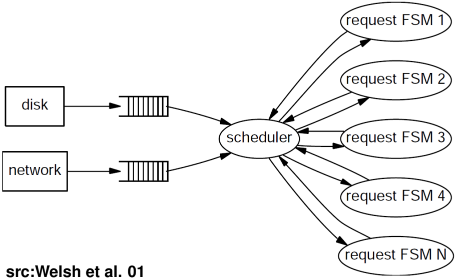
Event-driven Server



src:Pai et al. 99

- The server executes a loop, in which it:
  - waits for events (usuallly I/O events)
  - processes these events (sequentially, but may be not in order)
- Blocking is avoided by using **non-blocking** I/O operations

- Need to keep a FSM for each request
  - The loop dispatches the event to the appropriate FSM
- Known as the state machine approach



src:Welsh et al. 01

Other Scalability Issues

Data copying especially in network protocols

- Use buffer descriptors (similar to `java.nio.buffers`) not simple pointers
- Use scatter/gather I/O, e.g. `readv()/writev()`

Memory allocation default allocator is general purpose

- Design your own
  - Which can pre-allocate a pool of memory buffers
  - Avoid to free those buffers

Concurrency control

- Avoid sharing (if possible, sometimes requires copying)
  - Usually, you cannot avoid sharing memory buffers
- 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

Further Reading

- Ch. 3 of van Steen and Tanenbaum, *Distributed Systems, 3rd Ed.*
  - Section 3.1 *Threads*
- Arpaci-Dusseau & Arpaci-Dusseau, *Event-based Concurrency*, Ch. 33 of OSTEP book
- Pai et al., *Flash: An efficient and portable Web Server*, in 1999 Annual Usenix Technical Conference
- Welsh et al, *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*, in Symposium on Operating Systems, 2001
  - Matt Welsh, *A Retrospective on SEDA*, Blog entry with a critical assessment of SEDA by its designer 10 years later, i.e. in 2010
- Kegel, D. *The C10K problem*