02: Concurrent & Network Programming

Distributed Software Architectures

Stefan Huber <shuber.lba@fh-salzburg.ac.at>

March 22, 2019

Section 1

Concurrent Programming

Processes and threads

Process:

- Multi-tasking: An OS can execute multiple programs at a time.
- A process is an OS abstraction to provide a *virtual processor* with *virtual memory*.
- The OS isolates processes from each other: It gives the illusion of having exclusive access to the processor and memory.
- *Process control block* contains information on:
  - Process Id
  - CPU states: instruction pointer, registers, memory mapping, ...
  - Privilege information: user id, group id, ...
  - Resources: opened files, sockets, pipes, ...
- Processes hierarchy: Every process has a parent process.

Threads:

- Multi-threading: Multiple threads of execution in one process context.
- Threads operate on the process memory and with the process resources.
- Thread safety:
  - Code is thread-safe if, when executed by multiple threads, there is no unintended interference.
  - Global states[1] may break thread safety; concurrent access must be taken into account.

[1] This includes static variables in C functions. Thread-safety requires at least *re-entrancy*.

Process memory layout

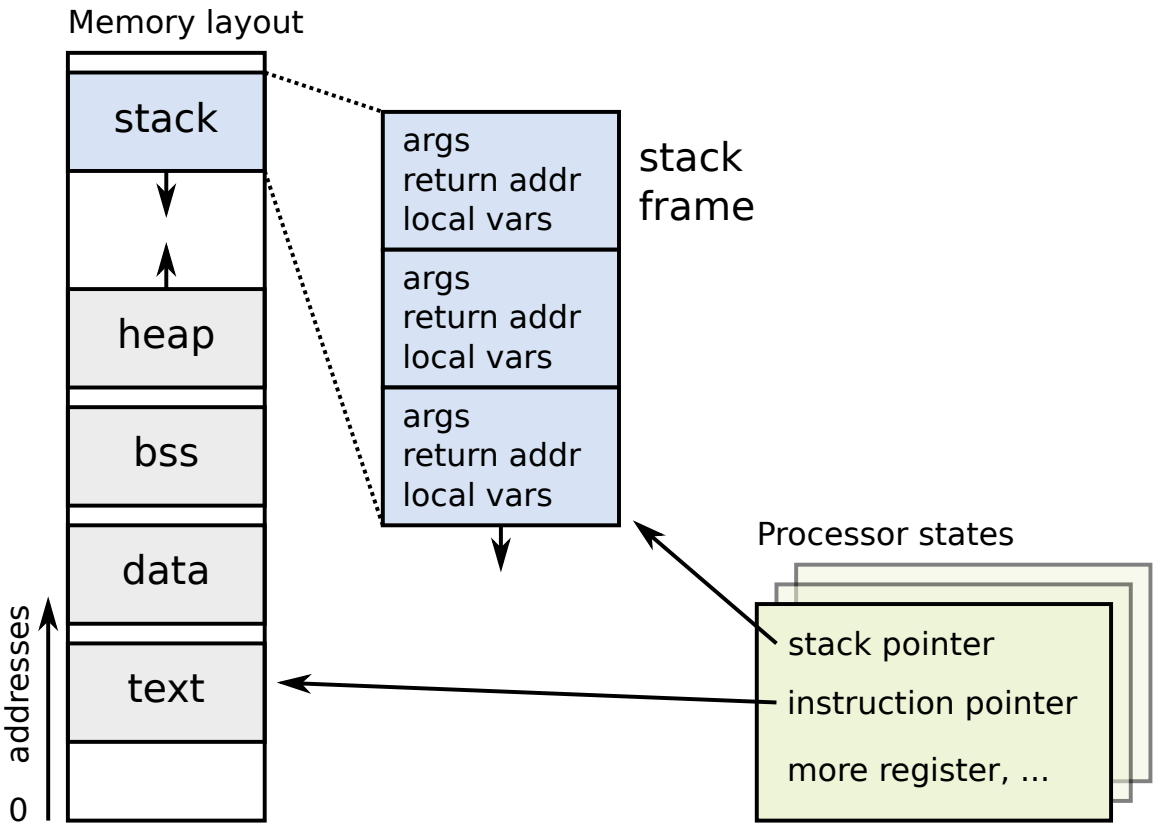


Figure: A simplified process memory layout and the relationship to processor registers. A "thread of execution" refers to the processor states.

Multi-threaded clients

Latencies:

- Communication in a distributed system may involve significant latencies.
- Idea: Hide latencies by doing something else while waiting.

Web browser:

- Has to fetch many resources besides referenced in an HTML document.
- Fetch them in parallel to avoid summing up latencies.
- Displaying content before having all resources fetched.
- Multi-threading simplifies implementation. For instance, each fetch can be implemented with a simple blocking read.
- Multiple connections in parallel may utilize multiple web servers.

Example

Electrolysis[2] is a multi-process architecture of Firefox. The parent process forks 4 child processes to handle web content.[3] Each child process has a plethora of threads running.

[2] `https://wiki.mozilla.org/Electrolysis`

[3] Firefox 66 increases this to 8: `https://bugzilla.mozilla.org/show_bug.cgi?id=1470280`.

Thread-level parallelism

- A modern web browser may create hundreds of threads.
- Most are probably not running most of the time but waiting.
- To what extent are those running in parallel?

Definition (Thread-level parallelism (TLP))

For a total of N threads let c_i be the fraction of time where exactly i threads executed simultaneously. Then we define

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}.$$

- Modern web browsers have a TLP of 1.5 to 2.5.
- Hence, a processor with 2 to 3 cores could be utilized well.
- Moreover, the hundreds of threads are used with software architecture in mind.

Multi-threaded servers

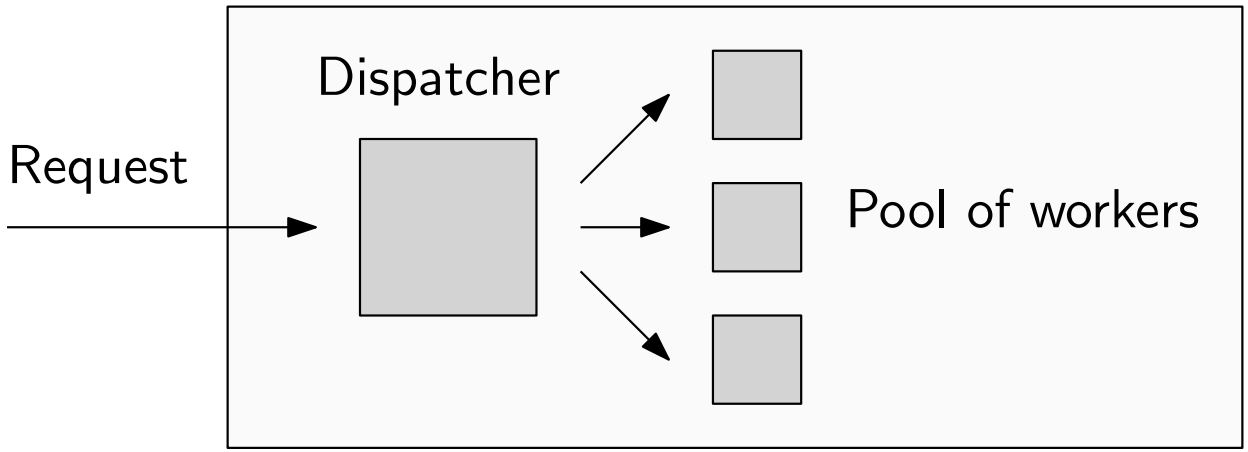
Servers deal with multiple clients, hence multi-threading is typically more important for servers than clients.

- Simpler code.
- Higher performance.

Multi-threaded servers

Example: File server

- Popular architecture: A dispatcher and multiple worker threads.
- Dispatcher thread receives requests from clients.
- Dispatcher distributes work to one of the worker threads.
- Worker thread performs the read with blocking reads on the file system.



Benefits:

- Parallel handling of clients, unlike for a single-threaded implementation with blocking file IO.
- Simpler to implement than managing asynchronous calls with multiple clients.

Benefits of concurrent programming

Performance gains:

- Parallel execution:
  - Hyper-threading
  - Multi-core processors
  - Multiple processors
- IO:
  - Native Command Queing (NCQ) of hard disks, RAID systems.
  - Parallel fetch of web page resources to reduce serialized latencies.
  - (Concurrent IO can easily degrade performance, too!)

Software architecture gains:

- Modularization into concurrent modules as architecture.
- Simpler code, e.g., we can still use blocking IO instead of asynchronous IO.
- Splitting up a process into multiple can improve security:
  - Isolation mechanisms
  - Finer privilege control

Concurrent programming does not imply *parallel (simultaneous) execution*.

Concurrency in Python

Python documentation to concurrency:

- `https://docs.python.org/3/library/concurrency.html`

Modules in the Python Standard Library:

- threading
- multiprocessing
- and more...

Threading module

`Thread` class represents a thread:

- A callable object (e.g., function) can be passed to the constructor.
- The method `start()` starts the thread execution.
- The method `join()` waits until thread terminates.

The Global Interpreter Lock

The standard Python interpeter (CPython) has a Global Interpreter Lock (GIL):

- No two threads can execute Python byte code at the same time.
- Makes access to Python data structures thread-safe.
- However, no parallelism for Python threads within a process!

Lock

The class `Lock` forms synchronization primitive:

- Can have two states: locked, unlocked
- The method `release()` unlocks a Lock.
- The method `acquire()` locks it.
  - If it is unlocked, it gets locked and continues execution.
  - It it is already locked, the calling thread gets blocked until another thread releases it.
- Can be used with `with` statement (context manager).

Side topic: Context Manager

When dealing with all kind of resources we often produce code like this:

```
VAR = EXPR                      # Creating file object, threading.Lock(), etc.
VAR.some_initialization()       # Opening file, aquiring lock, etc.
try:
    some_logic(VAR)             # File I/O, accessing resource, etc.
finally:                        # No matter whether exception has been raised
    VAR.some_cleanup()          # Closing file, releasing lock, etc.
```

This is a typical situation where the *context manager protocol*[4] can help:

```
# Aquire and later release mylock in an exception-safe fashion
with mylock:
    do_something()
```

This translates basically to

```
mylock.__enter__()          # Does lock.acquire()
try:
    do_something()
finally:
    mylock.__exit__()   # Does lock.release()
```

[4] See https://docs.python.org/3/reference/datamodel.html#context-managers and https://www.python.org/dev/peps/pep-0343/.

Barrier

The class `Barrier` makes all threads to wait for each other.

- A barrier is created for a number N of threads.
- The method `wait()` makes the thread wait until N threads called `wait()` and then all threads are released simultaneously.

Event

The class `Event` is a one-bit signal (flag).

- The method `is_set()` returns `True` iff flag is set.
- The method `set()` raises the flag and `clear()` resets it.
- The method `wait()` makes the thread wait until flag is raised.

Multiprocessing module

The `multiprocessing` package provides a similar API to `threading`.

- The class `Process` is like `Thread`.
- The method `start()` and `join()` start execution and wait until termination.
- There is also `Lock()`, `Event()`, `Barrier()`, and the like.

Multiprocessing module

The `multiprocessing` package provides a similar API to `threading`.

- The class `Process` is like `Thread`.
- The method `start()` and `join()` start execution and wait until termination.
- There is also `Lock()`, `Event()`, `Barrier()`, and the like.

But there is more:

- Processes have their own address space; we require *Inter-Process Communication*:
  - The class `Queue` allows to `put()` and `get()` objects.
  - The function `Pipe()` creates a pair of connection object.
  - Shared memory.
- The class `Pool` allows to easily distribute computational load.

```
1 def f(x):
2     return x*x
3
4 if __name__ == "__main__":
5     with multiprocessing.Pool(5) as p:
6         print(p.map(f, [1, 2, 3]))
```

Processes are scheduled by the OS: No GIL between processes.

Section 2

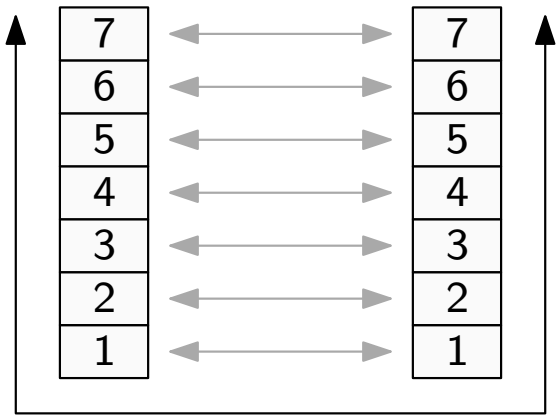
Network Programming

Reminder

Anwesenheitsliste nicht vergessen.

ISO OSI Reference model

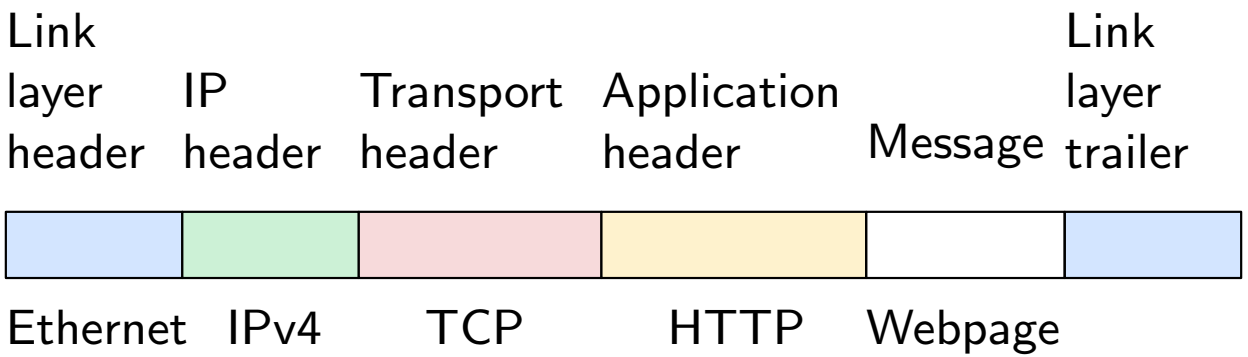
7	Application	High-level APIs, remote file access, resource sharing
6	Presentation	Data translation including encoding, compression, encryption
5	Session	Communication sessions with back-and-forth transmission
4	Transport	Sending data segments and datagrams
3	Network	Routing packets, addressing, traffic control
2	Data link	Data frames between nodes
1	Physical	Bit stream over physical medium



- A *reference model*.
- Each layer dedicated to a specific task.
- Each layer builds upon the layer below, provides a service to the layer above
- Software engineering aspects:
  - *Separation of concerns*: Dedication to specific tasks, inner cohesion.
  - A concrete layer implementation must be *exchangeable*: It must only rely on the *contract* with the layer below and fulfills the *contract* to the layer above.

Internet protocol suite

Application	HTTP, FTP, DNS, IMAP, POP, SMTP, SSH, Telnet, ...
Transport	TCP, UDP, ...
Internet	IPv4, IPv6, ICMP, ...
Link	Ethernet, WiFi, PPP, FDDI, ...



Network tools

host	DNS lookup utility
netstat	Prints network connections, and more
nc	Netcat is the TCP/IP swiss army knife
socat	Socket cat, a mix of nc and cat
curl	A command line data transfer tool by URL.
tcpdump	Dumps traffic on a network
wireshark	An extensive network analyzer for all kind of protocols

Sockets

Berkley sockets:

- *The* API for communication, not only for Internet network communication:
  - A socket is a communication endpoint.
  - The communication happens between two sockets.
- A typical sequence of socket calls looks as follows:

server		**client**
s = socket()		
s.bind()		
s.listen()		b = socket()
	a = s.accept()	b.connect()
	a.send()/recv()	b.send()/recv()
	a.close()	b.close()
s.close()		

- The middle column is typically done repeatedly:
  - In a single-threaded loop.
  - In parallel threads.
  - In a forked process.

Sockets in Python

```python
import socket

# See https://docs.python.org/3/library/socket.html
# AF_INET = "Internet protocol", SOCK_STREAM = TCP, SOCK_DGRAM = UDP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("127.0.0.1", 1200))
s.listen()
a, addrinfo = s.accept()
print("recv: ", a.recv(1024).decode("utf-8"))
a.send(bytes("greetings from server", "utf-8"))
a.close()
s.close()
```

```python
import socket

b = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
b.connect(("127.0.0.1", 1200))
b.send(bytes("greetings from client", "utf-8"))
print("recv: ", b.recv(1024).decode("utf-8"))
b.close()
```

A version for UDP

```python
import socket

# With SOCK_DGRAM we do not use listen() and accept().
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(("localhost", 1200))
print("recv:", s.recv(1024))
```

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# We can use sendto() instead of connect() and send().
s.sendto(bytes("hi!", "utf-8"), ("localhost", 1200))
```

Technical details

Timeouts:

- Operations are by default blocking operations.
- A timeout of x (float) seconds can be set by `socket.socket.settimeout(x)`.
- If `x` is `None` then timeout is infinity.

Privileged ports:

- Ports below 1024 are *privileged*.
- On Unix-like operating systems superuser rights are required.

http.server

The module `http.server` implements a simple (non-production) HTTP server.

- `https://docs.python.org/3/library/http.server.html`

```
import http.server
import threading

# A customized Handler class to process HTTP requests
class Handler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/plain")
        self.end_headers()
        self.wfile.write("Path was {}\n".format(self.path).encode())

if __name__ == "__main__":
    server = http.server.HTTPServer(('localhost', 1080), Handler)
    server_th = threading.Thread(target=server.serve_forever)
    server_th.start()

    input("Press <return> to exit")
    server.shutdown()
```

urllib.request

The module `urllib.requests` implements access to URLs in all detail.

- Authentication, SSL/TLS, redirections, cookies, ...
- `https://docs.python.org/3/library/urllib.request.html`

```
1 import urllib.request
2
3 if __name__ == "__main__":
4     # Does actually a 303 Moved Permanently to https://...
5     with urllib.request.urlopen("http://www.sthu.org/monalisa.txt") as f:
6         print(f.read().decode("utf-8"))
```