Course 2

Understanding Slowness

Slow Web Server

ab - Apache benchmark tool

```
ab -n 500 site.example.com
```

Profiling - Improving the code

Profiling ??? IT ???????Profile ?????????????? Profiling ?????????????????????? Profiling ?????????????????????????????

A profiler is a tool that measures the resources that our code is using, giving us a better understanding of what's going on.

- gprof : For C program
- cProfile : For Python program
- pprofile3 + kcachegrind(GUI) : For Python program
- Flat, Call-graph, and Input-sensitive are integral to debugging
- timeit (python module) : Measure execution time of small code snippets

Parallelizing operations

- Speed Up Your Python Program With Concurrency – Real Python

Python modules

- threading
- asyncio
- future

Concurrency for I/O-bound tasks

Python has two main approaches to implementing concurrency: threading and asyncio.

1. Threading is an efficient method for overlapping waiting times. This makes it well-suited for tasks involving many I/O operations, such as file I/O or network operations that spend significant time waiting. There are however some limitations with threading in Python due to the Global Interpreter Lock (GIL), which can limit the utilization of multiple cores.
2. Alternatively, asyncio is another powerful Python approach for concurrency that uses the event loop to manage task switching. Asyncio provides a higher degree of control, scalability, and power than threading for I/O-bound tasks. Any application that involves reading and writing data can benefit from it, since it speeds up I/O-based programs. Additionally, asyncio operates cooperatively and bypasses GIL limitations, enabling better performance for I/O-bound tasks.

Python supports concurrent execution through both threading and asyncio; however, asyncio is particularly beneficial for I/O-bound tasks, making it significantly faster for applications that read and write a lot of data.

Parallelism for CPU-bound tasks

Parallelism is a powerful technique for programs that heavily rely on the CPU to process large volumes of data constantly. It's especially useful for CPU-bound tasks like calculations, simulations, and data processing.

Instead of interleaving and executing tasks concurrently, parallelism enables multiple tasks to run simultaneously on multiple CPU cores. This is crucial for applications that require significant CPU resources to handle intense computations in real-time.

Multiprocessing libraries in Python facilitate parallel execution by distributing tasks across multiple CPU cores. It ensures performance by giving each process its own Python interpreter and memory space. It allows CPU-bound Python programs to process data more efficiently by giving each process its own Python interpreter and memory space; this eliminates conflicts and slowdowns caused by sharing resources. Having said that, you should also remember that when running multiple tasks simultaneously, you need to manage resources carefully.

Combining concurrency and parallelism

Combining concurrency and parallelism can improve performance. In certain complex applications with both I/O-bound and CPU-bound tasks, you can use asyncio for concurrency and multiprocessing for parallelism.

With asyncio, you make I/O-bound tasks more efficient as the program can do other things while waiting for file operations.

On the other hand, multiprocessing allows you to distribute CPU-bound computations, like heavy calculations, across multiple processors for faster execution.

By combining these techniques, you can create a well-optimized and responsive program. Your I/O-bound tasks benefit from concurrency, while CPU-bound tasks leverage parallelism.

psutil

```
# Installation
pip3 install psutil
```

Usage

```
import psutil


# for checking CPU usage
psutil.cpu_percent()


# For checking disk I/O,
psutil.disk_io_counters()


# For checking the network I/O bandwidth:
psutil.net_io_counters()
```

rsync with python

Use the rsync command in Python

```
import subprocess
src = "<source-path>" # replace <source-path> with the source directory
dest = "<destination-path>" # replace <destination-path> with the destination directory


subprocess.call(["rsync", "-arq", src, dest])
```

Segmentation fault

??????? - ?????????????????? C, C++??

gdb

- `ulimit -c unlimited` : ???? core file ?? unlimited

- `gdb -c <core-file> <program-name>` : ?? core file ???

```
ulimit -c unlimited

gdb -c core example
```

gdb sub-commands

- backtrace : ???????????
- up : ?? backtrace ????????
- list : ???????????
- print : ???????

```
gdb -c core example

....

(gdb) backtrace

....

(gdb) up

...

list

...

print i

...

print argv[0]

...

print argv[1]
```

Revision #27
Created 3 December 2024 10:08:08 by Admin
Updated 17 December 2024 15:40:35 by Admin