Combining asyncio und threads in a single application

PyCon JP 2020 – 28.08.2020
Online

Joining from Düsseldorf, Germany

Marc-André Lemburg :: eGenix.com GmbH

Speaker Introduction

- Marc-André Lemburg
  - Python since 1994
  - Studied Mathematics
  - CEO eGenix.com GmbH
  - Consult as Interim CTO / Senior Software Architect
  - EuroPython Society Chair
  - Python Software Foundation Fellow
  - Python Core Developer
  - Based in Düsseldorf, Germany
  - More: http://malemburg.com

Terminology: Synchronous / Threaded / Asynchronous

- Synchronous
  - All instructions are executed one after another
  - I/O and similar external resources cause execution to wait
  - Timing is not a problem. Everything is deterministic.
  - Problem: Waiting is not an efficient use of resources :-)

Terminology: Synchronous / Threaded / Asynchronous

- Threaded
  - Several synchronous parts of the program run in parallel, using OS threads
  - Execution is controlled by the OS, not the application
  - Threads are often assigned to different CPU cores

  - Problem: Sequence of execution is not necessarily deterministic
  - Problem: Unexpected delays can happen
  - Problem: Sharing data is hard – requires locks
  - Problem: OS overhead

  - Advantage: Efficient use of resources

Terminology: Synchronous / Threaded / Asynchronous

- Asynchronous
  - While some parts of the program wait for e.g. I/O, other parts can continue to run
  - Execution is controlled by the application, not the OS
  - This is not the same as “running in parallel“ (threading)

  - Problem: Sequence of execution is not necessarily deterministic
  - Problem: Unexpected delays can happen
  - Problem: Scope limited to a single core
  - Problem: All parts of the code have to participate

  - Advantage: Efficient use of resources

Python: Global Interpreter Lock (GIL)

- The GIL makes sure that only one thread runs Python byte code at any point in time
  - Only released for I/O or other long running tasks...
  - … and then only if no Python code can be run

```
204  /* Take the GIL.
205
206     The function saves errno at entry and restores its value at exit.
207
208     tstate must be non-NULL. */
209  static void
210  take_gil(PyThreadState *tstate)
211  {
212      int err = errno;
```

https://github.com/python/cpython/blob/master/Python/ceval_gil.h

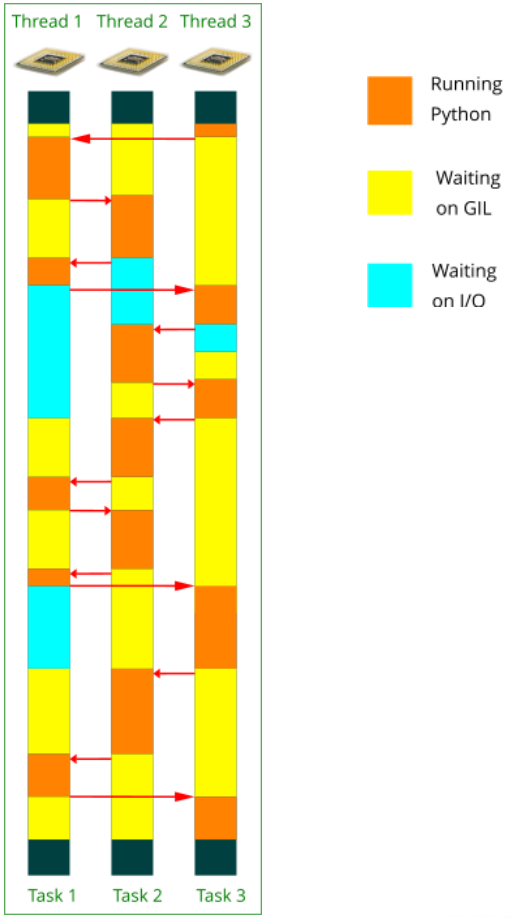
- Threads can only share the Python Interpreter, not use it simultaneously
  - Result: Even if you have multiple Cores in the CPUs, only one thread can run Python byte code
  - All other threads which want to run Python code have to wait

Goal: Use CPUs as efficiently as possible with Python

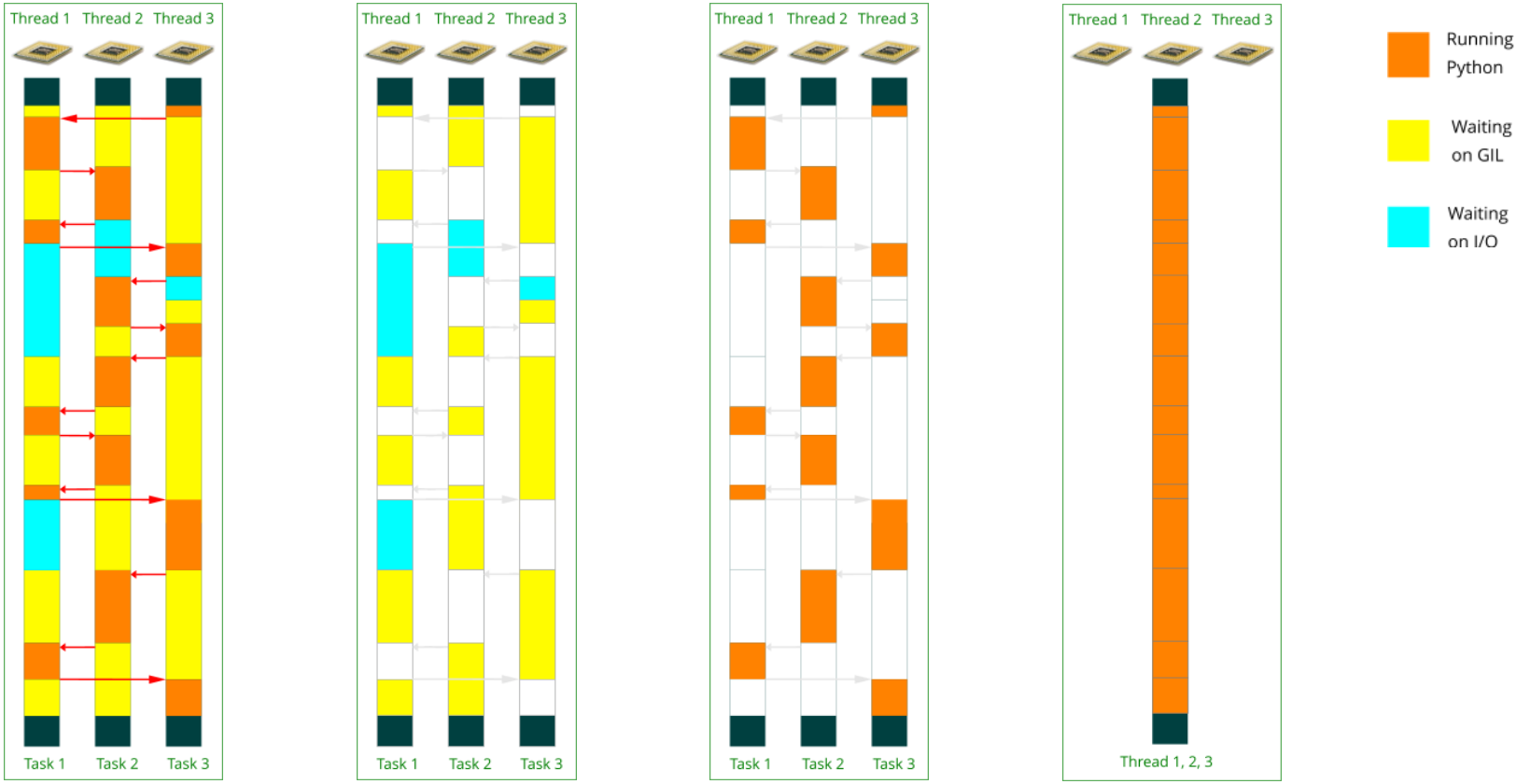
- Examples for asynchronous Python applications
  - Web Server (Tornado, Starlette)
  - Chat Server (Discord)
  - IoT Server (Home Assistant)
- Examples for synchronous / threaded Python applications
  - Database connections
  - Many non-Python tools
  - Embedded third party libraries
- Existing synchronous / threaded applications should remain usable, but with the benefits of using asynchronous execution where possible

Python: Threaded code / multiple cores/threads

- Threaded + multiple cores/threads = Much waiting
  - Threads need to wait for the GIL
    Delays due to I/O
  - Not much parallel work
    (only while doing I/O)

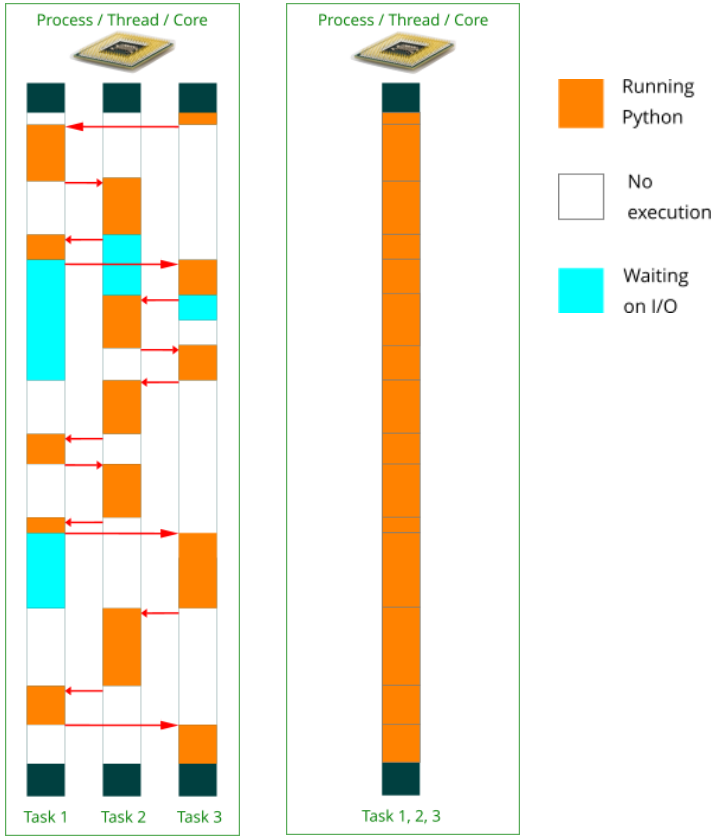


Python: Threaded code / multiple cores/threads



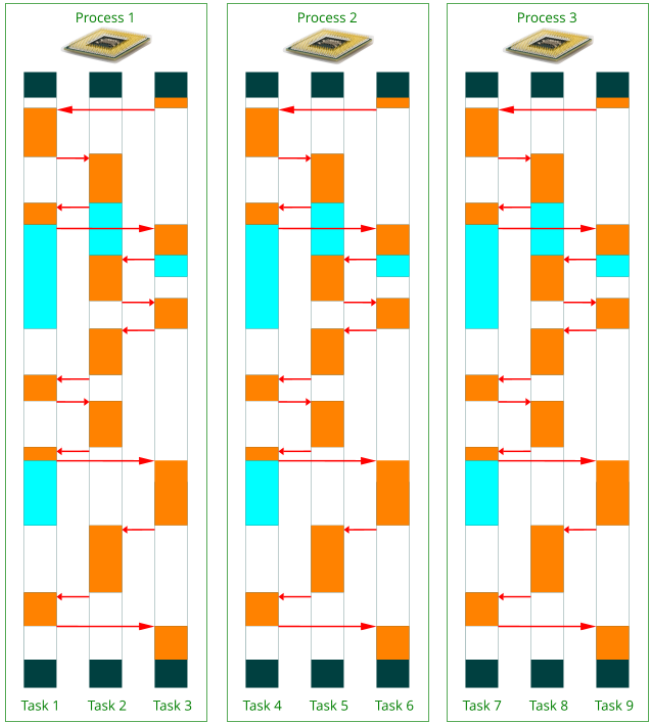
Python: Asynchronous to saturate a single core/thread

- Asynchronous + one thread/process = less waiting, but only one core
  - All application parts have to participate
  - Active passing of control (cooperative)
  - Less overhead compared to threads
  - No parallel work, only simulated
  - More efficient use of the core



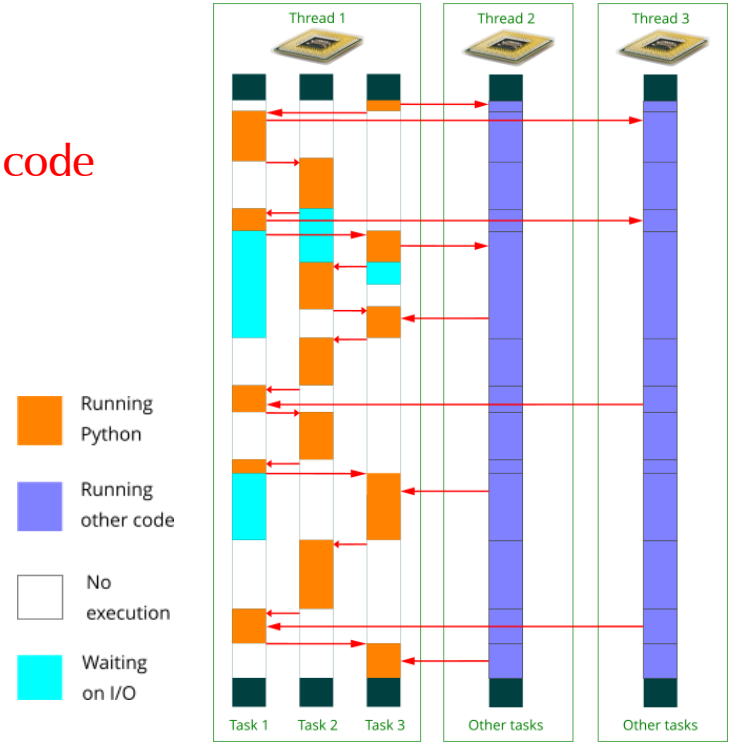
Python: Extend asynchronous to all cores via processes

- Asynchronous + multiple processes = less waiting, all cores
  - All application parts have to participate
  - Active passing of control (cooperative)
  - Needs more RAM
  - Recommendation: 1-2 Processes per Core
  - Partially parallel work
  - More efficient use of the CPU



Python: Saturate other cores with external code

- Asynchronous + multiple threads = less waiting, more cores
  - All application parts have to participate
  - Active passing of control (cooperative)
  - Parallel work between Python / external code
  - More efficient use of the CPU
  - Good approach when embedding calculation intense packages or external tools



async + await: Quick intro

- Coroutines
  - Like “Subroutines”, but routine can internally give up control to the calling function
- New keywords in Python 3.5
  - Make working with Coroutines a lot easier
  - async def task() - defines a Coroutine
  - await io_call() - gives up control, until io_call() responds
- Package asyncio
  - Provides the event loop to run coroutines
  - Many other helpers to run coroutines

async + await: Example

Synchronous

```
1   import asyncio
2   import time
3
4   # Synchron
5   def task_sync(x):
6       print (f'Task sync: {x} working')
7       time.sleep(2)
8       print (f'Task sync: {x} done')
9
10  task_sync('Example 1')
11
12  print ('-'*72)
```

Asynchronous

```
14  # Asynchron
15  async def task_async(x):
16      print (f'Task async: {x} working')
17      await asyncio.sleep(2)
18      print (f'Task async: {x} done')
19
20  # Call task
21  tasks = (task_async('Example 2'),
22           task_async('Example 3'),
23           )
24  async def main():
25      await asyncio.gather(*tasks)
26  asyncio.run(main())
```

async + await: Example

Synchronous

```
1   import asyncio
2   import time
3
4   # Synchron
5   def task_sync(x):
6       print (f'Task sync: {x} working')
7       time.sleep(2)
8       print (f'Task sync: {x} done')
9
10  task_sync('Example 1')
11
12  print ('-'*72)
```

Asynchronous

```
14  # Asynchron
15  async def task_async(x):
16      print (f'Task async: {x} working')
17      await asyncio.sleep(2)
18      print (f'Task async: {x} done')
19
20  # Call task
21  tasks = (task_async('Example 2'),
22           task_async('Example 3'),
23           )
24  async def main():
25      await asyncio.gather(*tasks)
26  asyncio.run(main())
```

async + await: Example

Synchronous

```
1   import asyncio
2   import time
3
4   # Synchron
5   def task_sync(x):
6       print (f'Task sync: {x} working')
7       time.sleep(2)
8       print (f'Task sync: {x} done')
9
10  task_sync('Example 1')
11
12  print ('-'*72)
```

Asynchronous

```
14  # Asynchron
15  async def task_async(x):
16      print (f'Task async: {x} working')
17      await asyncio.sleep(2)
18      print (f'Task async: {x} done')
19
20  # Call task
21  tasks = (task_async('Example 2'),
22           task_async('Example 3'),
23           )
24  async def main():
25      await asyncio.gather(*tasks)
26  asyncio.run(main())
```

Async doesn't like blocking code

- Task objects are run in an Event Loop
  - A task runs until it hits the next await, control then goes back to the Event Loop
  - Only works, if the code collaborates, doesn't unnecessarily blocks and gives back control

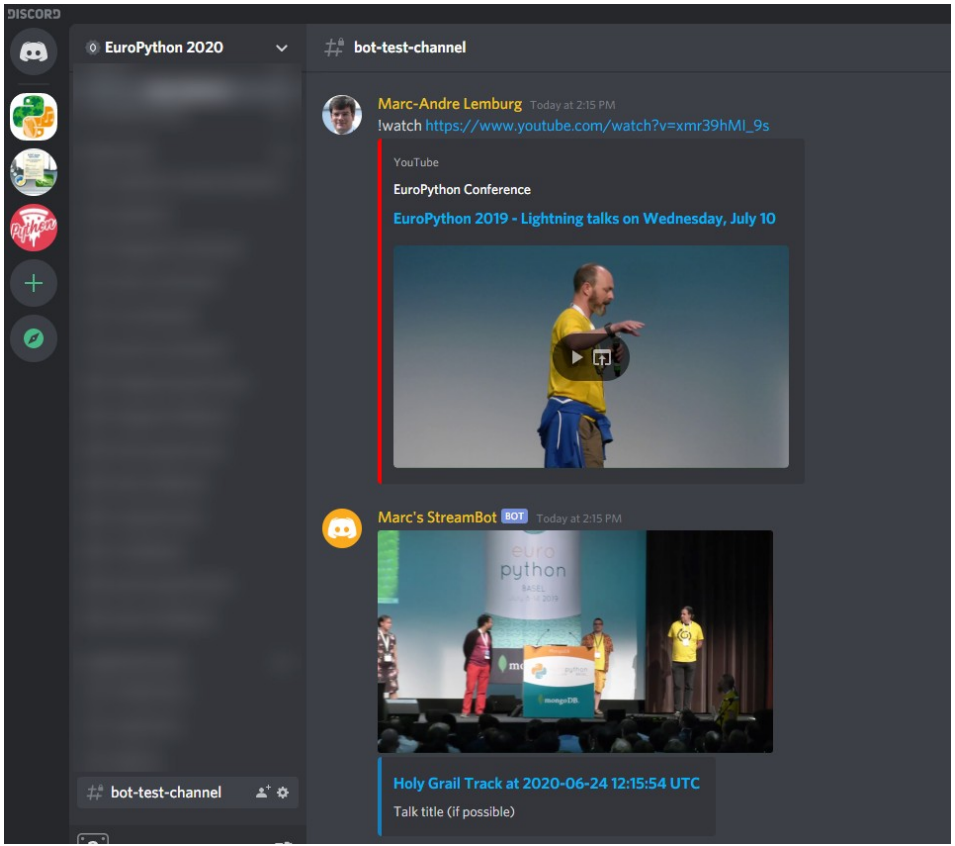
Async doesn't like blocking code

- Blocking Code
  - It is possible to run blocking code in a separate thread, so that it doesn't prevent the Event Loop from continuing with other tasks:

    loop.run_in_executor()

  - concurrent.futures.ThreadPoolExecutor provides such an "Executor"

Example: Discord Bot, using VLC



- Idea: Take snapshots of YouTube streams and send them to Discord as preview (EuroPython 2020)
- Implementation: Discord Bot using Discord.py (async) and python-vlc (threaded)

Discord.py async, VLC sync

```
### Bot

class MyClient(discord.Client):

    # ThreadPoolExecutor used for running VLC players
    thread_executor = None

    # VLC players dict, mapping filename to vlc player instance
    vlc_players = None

    async def on_connect(self):

        # Create ThreadExecutor
        self.thread_executor = concurrent.futures.ThreadPoolExecutor(
            max_workers=MAX_THREADS,
            )

        # Init client vars
        self.vlc_players = {}

    async def on_ready(self):
        print('Logged on as', self.user)

    async def on_message(self, message):
        if message.author == self.user:
            # don't respond to ourselves
            return
```

Discord.py API uses async

We start the VLC clients using a separate thread executor

Discord.py Commands

```
elif command == 'stream':
    if not admin_check(message):
        await channel.send('You need admin rights to run this command.')
        return
    # Start streaming
    if args:
        filename = clean_name(args[0])
    else:
        filename = 'snapshot.png'
    print ('Starting to stream picture %s to %s' % (filename, channel))
    await self.stream_image(channel, filename)

elif command == 'watch':
    if not admin_check(message):
        await channel.send('You need admin rights to run this command.')
        return
    if not args:
        await channel.send('Command needs a URL as argument.')
        return
    url = args[0]
    print ('Starting to stream %s picture to %s' % (url, channel))
    # Start streaming
    await self.stream_url(channel, url)

elif command == 'clear':
    if not admin_check(message):
        await channel.send('You need admin rights to run this command.')
        return
    print ('Clearing channel %s' % channel)
    # Clear all message in the channel
    await channel.purge()
```

Bot commands each start a separate aync method

“watch“ starts the VLC client und the streaming of the snapshots

Start VLC, then loop to send snapshots

```
async def stream_url(self, channel, url, interval=15):
        prev_messages = []
        filename = 'snapshot-%s.png' % clean_name(url)
        if filename not in self.vlc_players:
            player_task = asyncio.create_task(
                self.run_vlc_player(url, filename=filename))
        while True:
            if os.path.exists(filename):
                print ('Sending stream picture %s to %s' % (
                    filename, channel))
                embed = discord.Embed(
                    title='Holy Grail Track at %s' % timestamp(),
                    description='Talk title (if possible)',
                    url='https://zoom.us/',
                    color=1341883,
                    )
                embed.set_image(url='attachment://snapshot.png')
                try:
                    message = await channel.send(
                        embed=embed,
                        file=discord.File(filename))
                    if len(prev_messages) >= 6:
                        # Keep 6 images around
                        oldest_message = prev_messages.pop(0)
                        await oldest_message.delete()

                    prev_messages.append(message)

                except NETWORK_EXCEPTIONS as reason:
                    print ('Ignoring network error: %s' % reason)
            await asyncio.sleep(interval)
```

This is where the VLC client is started as a task

Once started, we run an endless loop send snapshots to the Discord channel

Streaming is done in a separate thread, parallel to the bot

```
def start_vlc_player(self, url):

    player = video_player.VideoPlayer()
    player.play_stream(url)
    player.wait_until_playing()
    return player

async def run_vlc_player(self, url, interval=5, filename='snapshot.png'):

    # Start VLC
    loop = asyncio.get_running_loop()
    player = await loop.run_in_executor(
        self.thread_executor,
        self.start_vlc_player,
        url)
    self.vlc_players[filename] = player

    # Loop and take snapshots
    try:
        while True:
            player.snapshot(filename, width=1920, height=1080)
            await asyncio.sleep(interval)
    finally:
        # XXX Could make this async as well
        player.stop_stream()
        del self.vlc_players[filename]
```

The method .start_vlc_player() is synchronous

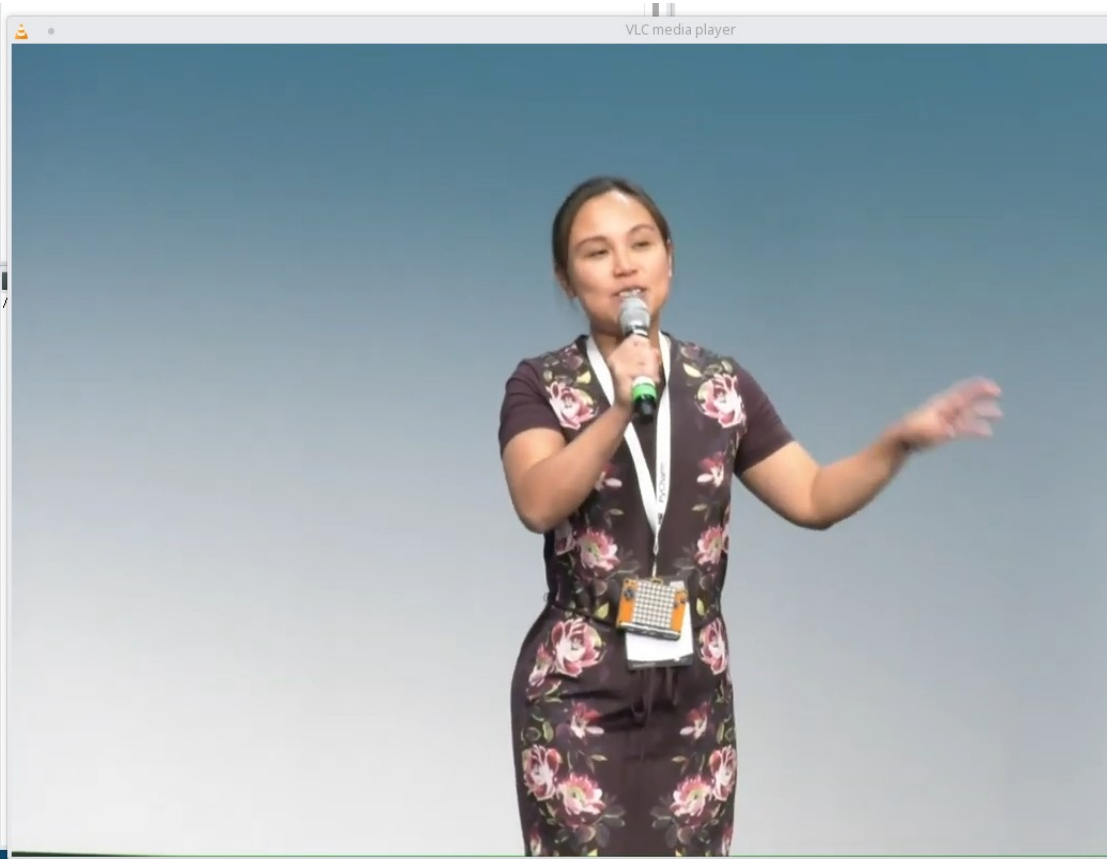
It is run in a separate thread, managed by the .thread_executor

Snapshots are taken regularly after the player has started

Discord Bot (async), using VLC (sync) in the same process

vlc-stream-screenshots : make — Konsole

```
drwxr-xr-x 2 lemburg lemburg      48 May 28 17:57 __pycache__
drwxr-xr-x 7 lemburg lemburg      86 May 15 13:41 pyenv
-rw-r--r-- 1 lemburg lemburg      43 May 15 16:13 requirements.txt
-rw-r--r-- 1 lemburg lemburg      32 May 15 13:50
-rw-r--r-- 1 lemburg lemburg 1148825 Jun 10 21:50 snapshot-httpswwwyoutubecomwatchv2a
0Cv3kXF4w.png
-rw-r--r-- 1 lemburg lemburg 1979725 May 28 19:19 snapshot-httpswww.youtube.comwatchv
uF2GhMAaQQQ.png
-rw-r--r-- 1 lemburg lemburg  761845 Jun 24 14:17 snapshot-httpswwwyoutubecomwatchvxm
r39hMI9s.png
-rw-r--r-- 1 lemburg lemburg  854206 Jun 24 14:10 snapshot.png
-rwxr-xr-x 1 lemburg lemburg     802 May 16 19:36 streamwatcher.py
-rwxr-xr-x 1 lemburg lemburg     777 May 16 11:29 streamwatcher.py~
-rwxr-xr-x 1 lemburg lemburg     775 May 15 16:44 streamwatch.py~
-rw-r--r-- 1 lemburg lemburg    4448 May 28 15:48 video_player.py
-rw-r--r-- 1 lemburg lemburg    4446 May 28 15:39
projects/vlc-stream-screenshots> make run
Logged on as test-streamwatcher#1753
Received message '!clear' from malemburg#7526 on channel bot-test-channel
Clearing channel bot-test-channel
Received message '!watch https://www.youtube.com/watch?v=xmr39hMI_9s' from malemburg#
7526 on channel bot-test-channel
Starting to stream https://www.youtube.com/watch?v=xmr39hMI_9s picture to bot-test-ch
annel
Sending stream picture snapshot-httpswwwyoutubecomwatchvxmr39hMI9s.png to bot-test-ch
annel
Failed to open VDPAU backend libvdpau_nvidia.so: cannot open shared object file: No s
uch file or directory
Sending stream picture snapshot-httpswwwyoutubecomwatchvxmr39hMI9s.png to bot-test-ch
annel
Sending stream picture snapshot-httpswwwyoutubecomwatchvxmr39hMI9s.png to bot-test-ch
annel
Sending stream picture snapshot-httpswwwyoutubecomwatchvxmr39hMI9s.png to bot-test-ch
annel
Sending stream picture snapshot-httpswwwyoutubecomwatchvxmr39hMI9s.png to bot-test-ch
annel
```



Thank you for your attention !

Beautiful is better than ugly.

Contact

eGenix.com Software, Skills and Services GmbH
Marc-André Lemburg
Pastor-Löh-Str. 48
D-40764 Langenfeld
Germany

eMail: mal@egenix.com
Phone: +49 211 9304112
Fax: +49 211 3005250
Web: http://www.egenix.com/