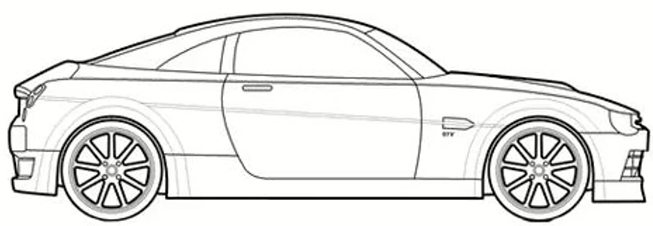
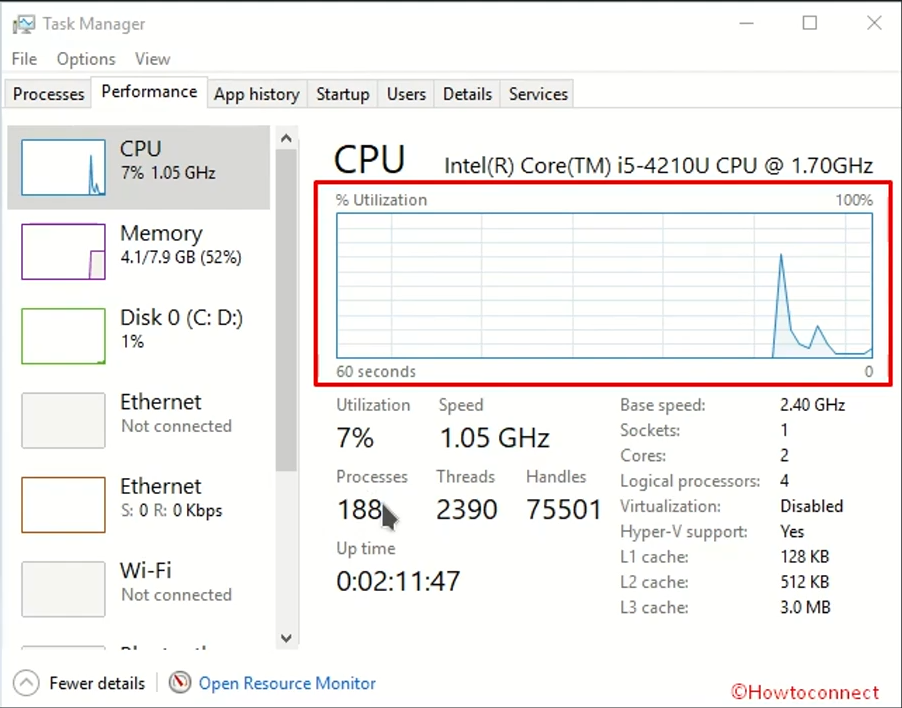
**Intro :**



در ابتدا لازم است با مفهوم process و thread آشنا شویم .شما یک ماشین را درنظر بگیرید که یک مجموعه واحدی میباشد داخل این ماشین ما ابزارها و قطعات مختلف را داریم (مثلاً لاستیک و چراغ ها) به ماشین process و به هرکدام از قطعات یک thread گفته می‌شود

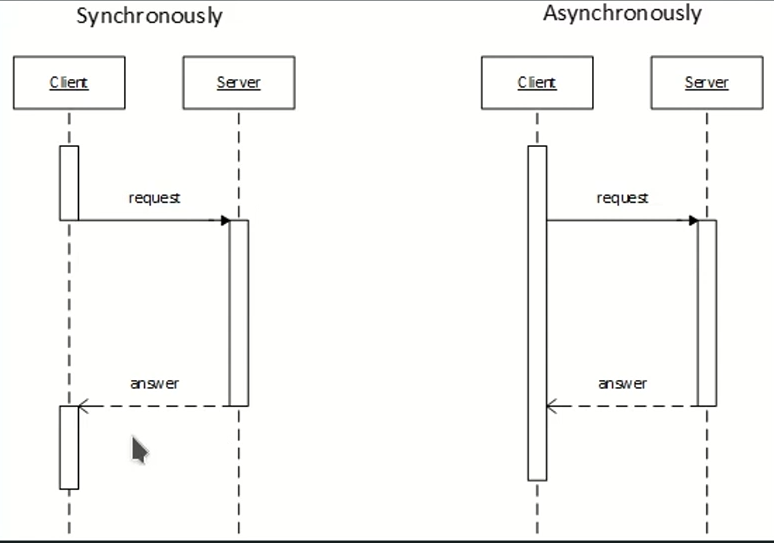
یا مثال دیگر مرورگر firefox یک process میباشد و هرtab این مرورگر که باز میباشد و کار خاصی را انجام میدهد thread گفته می‌شود



در ویندوز میتوانید در task manager تعداد processها و threadها را مشاهده کنید

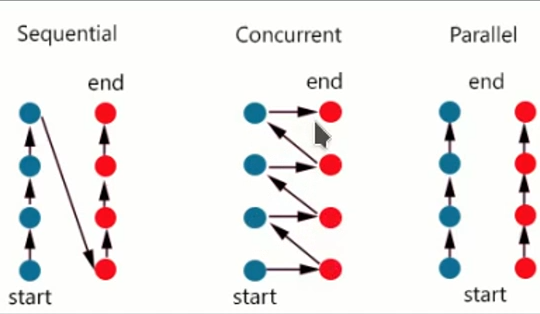
هر process حداقل یک thread را باید داشته باشد

در لینوکس هم میتوانیم با استفاده از دستور htop تعداد processها و threadها را ببینیم (البته process را با Tasks و thread را با thr نمایش میدهد)



در برنامه نویسی sync هنگامیکه client درخواستی را به server ارسال میکند تا زمانی که پاسخ از سرور بیاید باید منتظر بماند و هیچ کار دیگری انجام ندهد

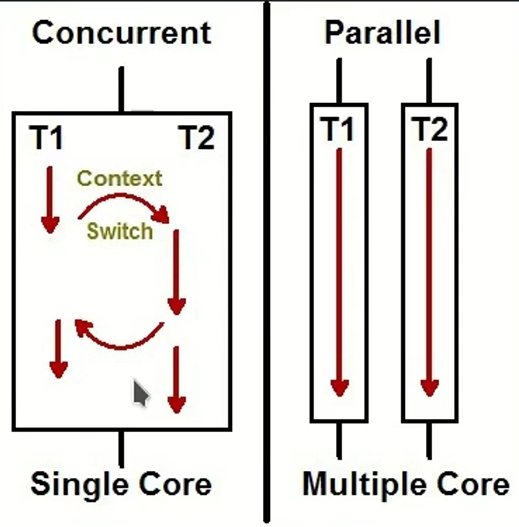
در برنامه نویسی async هنگامیکه client درخواست خود را به server ارسال میکند دیگر لازم نیست منتظر پاسخ server بماند و متواند کارهای دیگر خود را انجام دهد ( مثالا زمان ارسال پیامک احراز هویت)



sequential : مربوط به برنامه نویسی عادی و معمولی میباشد

concurrent : مربوط به برنامه نویسی async میباشد که در پایتون با استفاده از asyncio میتوانیم این مدل برنامه نویسی هایی را انجام دهیم

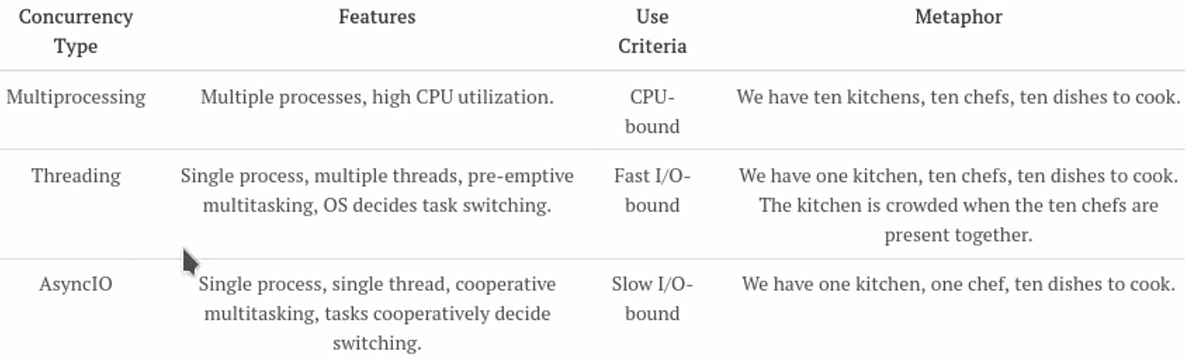
parallel : این مدل برنامه نویسی زمانی است که میخواهیم چندین thread یا process به طور همزمان اجرا شوند multitreading و multiprocessing در پایتون برای این مورد میباشد



زمانی از برنامه نویسی async استفاده می‌شود که بخواهیم از منابع i/o سیستم بهترین استفاده را بکنیم و منتظر نمانیم ( مثلاً میخواهیم درخواستی را به یک دیتابیس ارسال کنیم خب در این حالت i/o سیستم درگیر میباشد و میتوانیم با استفاده از برنامه نویسی async از زمانی که باید منتظر پاسخ دیتابیس باشیم استفاده کنیم) ( IO bound )

زمانی از برنامه نویسی parallel استفاده میکنیم که منابع cpu زیادی در اختیار داشته باشیم دراین حالت میتوانیم با استفاده از هر core cpu چندین process یا thread را بصورت همزمان انجام دهیم (cpu bound )

توجه شود که از multi treading میتوانیم در موارد ( IO bound ) نیز استفاده کنیم در این حالت سیستم عامل تصمیم میگیرد که چگونه بین taskها جابجا شود اما در async این برنامه و taskها میباشند که باعث جابجایی میشوند

کتابخانه asyncio از دو بخش high level و low level تقسیم‌بندی می‌شود که high level برای برنامه نویسان عادی میباشد و low level برای برنامه نویسانی میباشد که framework یا library طراحی میکنند

=======================================================================

**coroutines :**

coroutine همان فانکشن python میباشد منتها میتواند توی بدنه خودش stop بشه ما بریم یک کار دیگه ای رو انجام بدیم بعد برگردیم و ادامه اون function رو بریم جلو

برای ایجاد coroutine قبل از کلمه def باید async نوشته شود در ادامه داخل بدنه فانکشن این قابلیت رو باید داشته باشد که در یک مرحله‌ای stop بشه بره یک کار دیگه ای رو انجام بده اون جایی که ممکن است برنامه stop شود را با await مشخص میکنیم

به اون قسمتی که برنامه متوقف می‌شود IO block گفته می‌شود

async و await که درباره آن صحبت شد ربطی به پکیچ asyncio ندارد و در هسته اصلی پایتون میباشد

async def one(name):  
 print(f“Hello {name}”)

one(“ahmad”)

هنگامیکه کد بالا را اجرا کنیم درواقع اخطار RuntimWarning دریافت میکنیم مبنی بر اینکه هیچ awaitی در coroutine one وجود ندارد

coroutine ها را نمی‌توان بصورت مستقیم اجرا کرد و ۳ روش برای اجرای coroutine وجود دارد :

- run : به وسیله دستور run میتوانیم یک coroutine را اجرا کنیم :

asyncio.run(one(“ahmad”))

توجه داشته باشید که استفاده از run به تنهایی دستورات coroutineها را به صورت async اجرا نمیکند

-create\_task‌ : از این دستور استفاده مکنیم و برای هر اجرای coroutine یک task درست میکنیم و به صورت await اجرا میکنیم ( در حالت دومی که نوشته شده است هنگامیکه ahmad قرار است print شود ۲ ثانیه sleep می‌شود به همین دلیل سریعاً به دنبال اجرای sajjad می‌رود و دوباره همینجا ۲ ثانیه sleep می‌شود به همین خاطر تمام این ۲ ثانیه ها روی هم قرار گرفته‌اند و بعد از اتمام ۲ ثانیه هر دو print بصورت همان اجرا می‌شوند و زمان کلی اجرا ۲ ثانیه می‌باشد )  
تسک های create شده در event loop ثبت می‌شوند

=======================================================================

**cancel :**

در فایل cancel مثالی برای cancel کردن یک task نوشته شده است به این صورت که اگر اجرای یک task بیشتر از ۵ ثانیه طول کشید task به وسیله فانکشن مربوطه cancel می‌شود ( فانکشن مربوطه cancel میباشد و خطای CancelledError را برمیگرداند )

فانکشن done خروجی boolean دارد به این صورت که اگر یک task با موفقیت تمام شود مقدار true و اگر هنوز تمام نشده باشد مقدار false دارد

برنامه‌ای که در حالت اول نوشته شده است به صورت دستی بررسی میکند که مدت زمان اجرای یک تسک چقدر طول کشیده است

اما در حالت دوم از wait\_for استفاده کرده ایم ورودی های این فانکشن دو مقدار میباشد یکی coroutine و دومی مقدار زمان مورد نظر که در این حالت اگر خطای TimeoutError را برمیگرداند

فانکشن cancelled خروجی boolean دارد به این صورت که اگر یک task با موفقیت اجرا شود و cancel نشود مقدار false و اگر cancel شود مقدار true را برمیگرداند

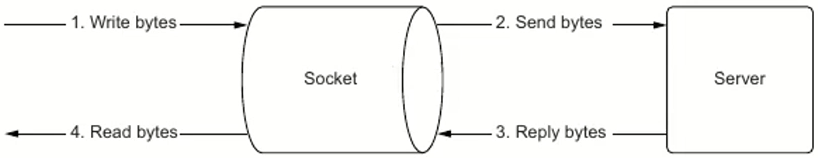
در حالت سوم از cancel شدن تسک به وسیله فانکشن shield جلوگیری میکنیم توجه داشته باشید که برای ادامه ی اجرا شدن تسک باید دوباره صدا زده شود (خط ۵۳ )

=======================================================================

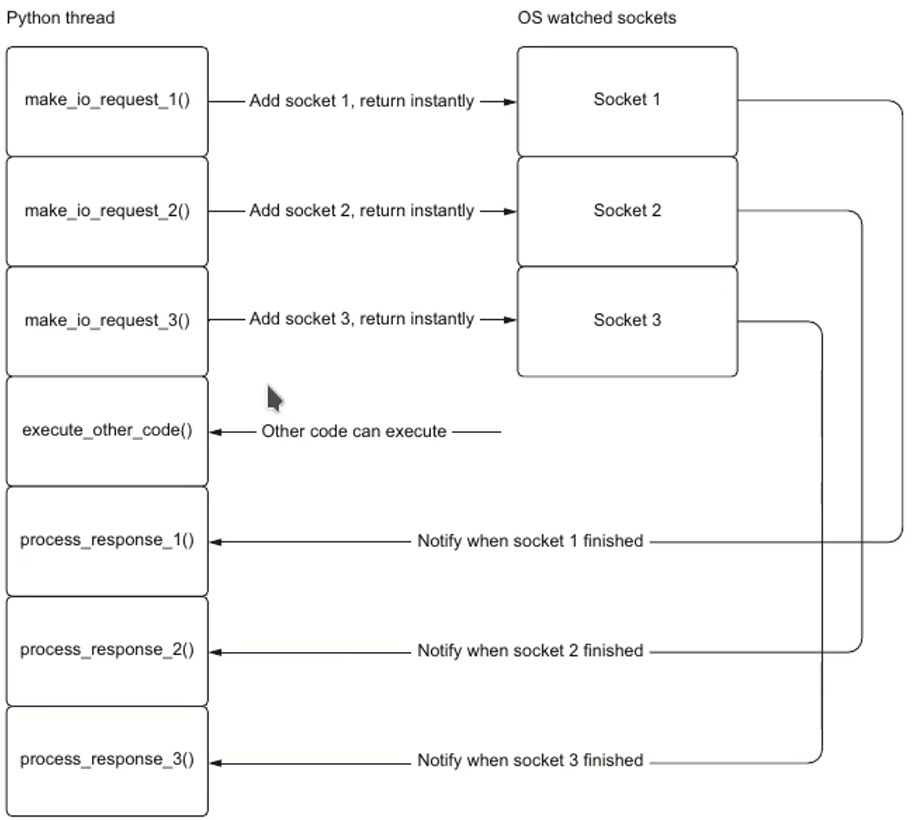
**event loop :**

نکته‌ای که باید به آن توجه کرد این است که ما برنامه نویسان معمولی خیلی خیلی کم بصورت مستقیم از event loopها استفاده میکنیم و خوده پایتون این event loop ها را برای ما مدیریت میکند

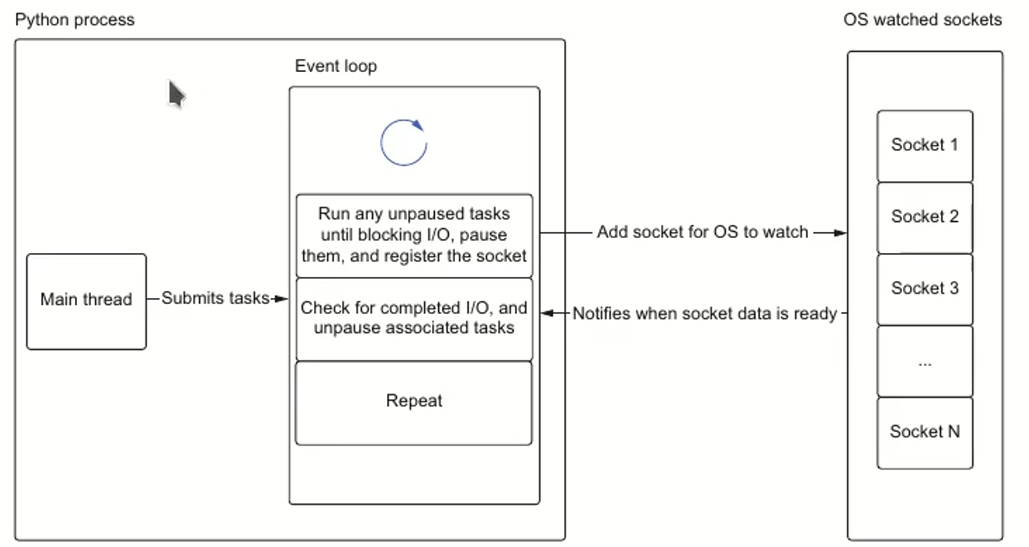
socket چیست :

هنگام ارسال درخواست . درخواست ابتدا وارد سوکت می‌شود . سوکت ها یک لایه انتزاعی هستند که وظیفه آن‌ها دریافت و ارسال اطلاعات هستند

زمانی که ما از async io استفاده میکنیم در‌واقع قسمتی از برنامه درگیر این سوکت ها میباشد

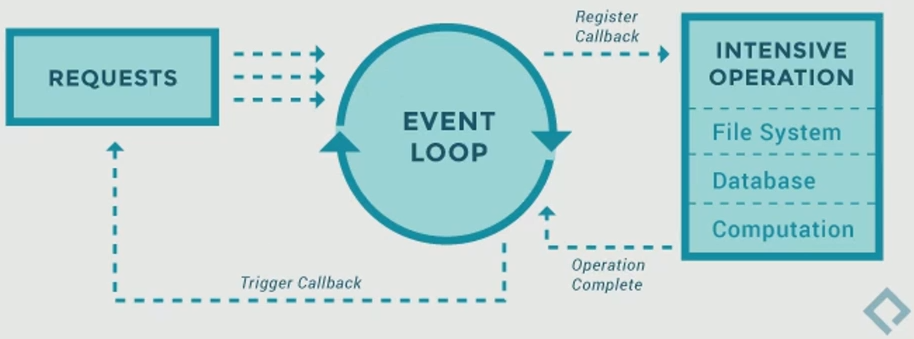


در شکل بالا نشان میدهد که سیستم عامل (os) کنترل کننده سوکت ها میباشد و زمانی که از asyncio استفاده میکنیم python و os با هم باید در ارتباط باشند



event loop ها را به عنوان یک حلقه در نظر بگیرید که دائماً اجرا می‌شود و تسک هایی که ثبت می‌شوند را برای شما اجرا میکند تسک ها در یک صف ذخیره می‌شوند و event loop وظیفه اجرای این تسک ها را دارد

مثالی که میتوان آن را بیشتر بررسی کرد در فایل coroutine.py میباشد که در تابع main دو تسک a و b ساخته می‌شود و در صفی ذخیره می‌شود خب ابتدا a باید اجرا شود event loop تسک a را شروع به اجرا شدن میکند در همین ابتدا به await برمیخورد به همین خاطر تسک a را رها میکند و در انتهای صف قرار میدهد تا منتظر پاسخ نماند و شروع به اجرا کردن تسک b میکند . تسک b هم به await بر میخورد و تسک b هم رها میکند و در انتهای صف مربوطه قرار میدهد ۲ ثانیه بعد پاسخی که تسک a منتظر آن بود آماده می‌شود و میتواند تسک a ادامه یابد به همین خاطر تسک a اجرا و تمام می‌شود و در همین ۲ ثانیه پاسخی که تسک b منتظر آن بود نیز آماده می‌شود بعد از پایان تسک a سریعاً تسک b اجرایش ادامه پیدا میکند و تمام می‌شود جمعا به وسیله asyncio و event loop هر دو تسک در ۲ ثانیه اجرا شدند



=======================================================================

**gather:**

پکیج aiohttp برای ارسال درخواست http به صورت async استفاده می‌شود .

متد gather از شما چند آبجکت awaitable میگیرد و این آبجکت ها را برای شما اجرا میکند این آبجکت ها باید coroutine باشند

متد gather یک پارامتر ورودی دارد به نام return\_exception که به صورت default برابر با False میباشد که اگر یکی از آبجکت های awaitable به exception خورد کل برنامه متوقف می‌شود اما اگر این پارامتر برابر با True شود exceptionی که تولید می‌شود به عنوان پاسخ برگردانده می‌شود

=======================================================================

**Future :**

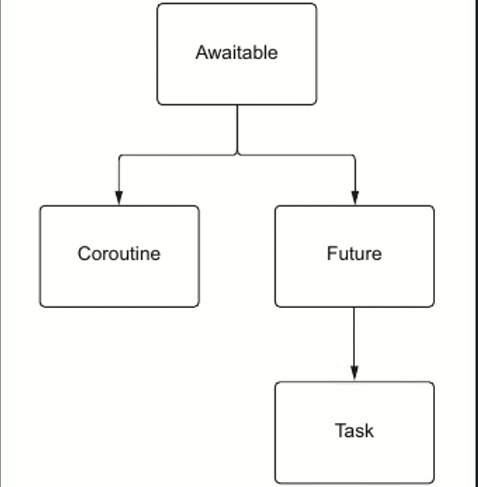
future به object هایی گفته می‌شود که الان مقدار ندارند و انتظار داریم بعداً مقدار دهی بشوند تا زمانی که مقدار ندارند به عنوان انجام نشده در نظر گرفته می‌شوند و زمانی که مقدار دهی شدند به عنوان انجام شده در نظر گرفته می‌شوند

روش پیشنهادی پایتون برای ساخت future این است که از event loopی که الان فعال هست از create\_future استفاده شود ( ()loop.create\_future )

متد done مشخص میکند که future مقدار دهی شده است یا نه

متد set\_result مقداری را به future مربوطه میدهد

متد result مقدار future مربوطه را نمایش میدهد



future یکی از مباحث low level در acyncio میباشند و ما هیچ‌وقت به طور مستقیم از این مبحث استفاده نمیکنیم از تمام تسک هایی که ساخته می‌شوند از Future ارثبری میکنند

=======================================================================

**as\_completed :**

کاری که as\_completed برای ما انجام میدهد این است که تسک های مختلف را برای ما اجرا میکند هرکدام از تسک ها پاسخش آماده شد برای ما برمیگرداند

تفاوتی که با gather دارد این است که gather صبر میکرد تا تمام تسک ها پاسخشان آماده شود و در آخر پاسخ هارا برای ما نمایش میداد

as\_completed هر کدام از objectهای awaitableی که برای آن ارسال کرده‌ایم را به future تبدیل میکند

=======================================================================

**wait :**

وظیفه wait هم انجام دادن تسک ها به ترتیب ( concurrently ) می‌باشد . با این تفاوت که مشخص میکند کدام تسک ها تا الان انجام شده است و کدام تسک ها هنوز تمام نشده‌اند

objectهایی که به wait فرستاده می‌شوند باید از نوع تسک باشند

پارامتر ورودی دیگر به نام return\_when وجود دارد با سه حالتی که: ۱- تمام تسک ها با موفقیت انجام شوند ۲- به اولین خطا برخورد کند ۳- به اولین موفقیت برسد

=======================================================================

**lock :**

در مثالی اولی که زده شد counter یک shared resource میباشد و تمام تسک ها میخواهند روی این counter تغییر ایجاد کنند به همین خاطر زمانی که 0.5 ثانیه متوقف می‌شوند باعث می‌شود مقدار counter در نهایت ۱ باقی بماند

اما در مثال دوم اگر اون 0.5 ثانیه زمانی که متوقف می‌شود را برداریم جواب نهایی ۱۰۰ میباشد

در مثال سوم از lock استفاده کرده‌ایم به این منظور که وقتی کد به خطی رسید که لازم بود قفل شود و اجازه داده نشود تسک دیگر به این قسمت وارد شود مقدار نهایی ما درست می‌شود ( توجه شود که در آخر باید قفل باز شود )

ممکن است زمانی فراموش کنیم که قفل را آزاد کنیم اگر از async with استفاده شود دیگر لازم نیست قطعه کد آزاد سازی قفل را بنویسیم

=======================================================================

**semaphore :**

semaphore هم به ماننده lock عمل میکند با این تفاوت که میتوانیم مشخص کنیم چه تعداد تسک همزمان میتوانند از shared resource استفاده کنند ( توجه داشته باشید که تعداد acquire ها و release ها باهم برابر باشند در این حالت میتوانیم از BoundedSemaphore استفاده کنیم که به مشکل برنخوریم )