Begin Modern Programming with Python

Pyi Soe

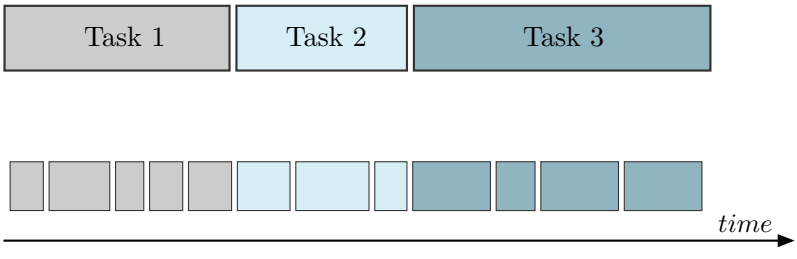
မာတိကာ

အခန်း ၁

Concurrency

Concurrency ဆိုတာ တစ်ချိန်တည်းမှာ အလုပ်တစ်ခုမကကို လုပ်ဆောင်တာလို့ ယေဘူယျ ပြောနိုင်ပါတယ်။ စန္တယားတီးရင်း သီချင်းဆိုနေတာဟာ အလုပ်နှစ်ခုကို concurrent လုပ်နေတာပါ။ ကားမောင်းရင်း ဖုန်းလည်းပြော၊ google map ကနေ လမ်းကြောင်းကြည့်သွားနေတာဟာလည်း concurrent လုပ်နေတာပဲ။ လူတွေဟာ တစ်ချိန်တည်း အလုပ် သုံးလေးမျိုး နိုင်နိုင်နင်းနင်း လုပ်နိုင်စွမ်း ရှိကြပါတယ်။ ဒီအခန်းမှာ လေ့လာကြရမှာကတော့ ဆော့ဖ်ဝဲတွေနဲ့ သက်ဆိုင်တဲ့ concurrency ပါ။ Concurrent ပရိုဂရမ်တွေဟာ နှစ်ခု (သို့) နှစ်ခုထက်ပိုတဲ့ အလုပ် (Task) တွေကို တစ်ချိန်တည်း ဆောင်ရွက်တဲ့ ပရိုဂရမ်တွေ ဖြစ်ပါတယ်။

ဒီစာအုပ်မှာ အခုချိန်ထိ တွေ့ခဲ့တဲ့ ပရိုဂရမ် အားလုံးလိုလိုဟာ sequential ပရိုဂရမ်တွေပါ။ အလုပ်တစ်ခုပြီးမှ နောက်တစ်ခုလုပ်တာကို sequential လို့ ခေါ်တယ်။ Sequential က တစ်ချိန်မှာ အလုပ်တစ်ခုပဲ လုပ်တာ၊ တစ်ခုပြီးမှပဲ နောက်တစ်ခုလုပ်တာ။ ပရိုဂရမ်တစ်ခုအတွက် problem decomposition လုပ်တဲ့အခါ အဓိက main task ကို အလုပ်အခွဲ subtask သုံးခု ခွဲမယ်ဆိုပါစို့။ Sequential ပရိုဂရမ်တစ်ခုဟာ အဲဒီ subtask သုံးခုကို တစ်ခုပြီးမှ တစ်ခု လုပ်ဆောင်မှာပါ (ပုံ ၁.၁ အပေါ် စတုဂံသုံးခု)။ ဒီ subtask တစ်ခုချင်းကို နောက်တစ်ဆင့် သေးငယ်တဲ့ subtask တွေ အဖြစ် ခွဲခြမ်းမယ်ဆိုရင်လည်း တစ်ခုပြီးမှ တစ်ခု လုပ်ဆောင်ရပါမယ်။ ပုံ (၁.၁) အောက်ဘက် စတုဂံလေးတွေက ဒီသဘောတရားကို ဖော်ပြထားတာ ဖြစ်တယ်။

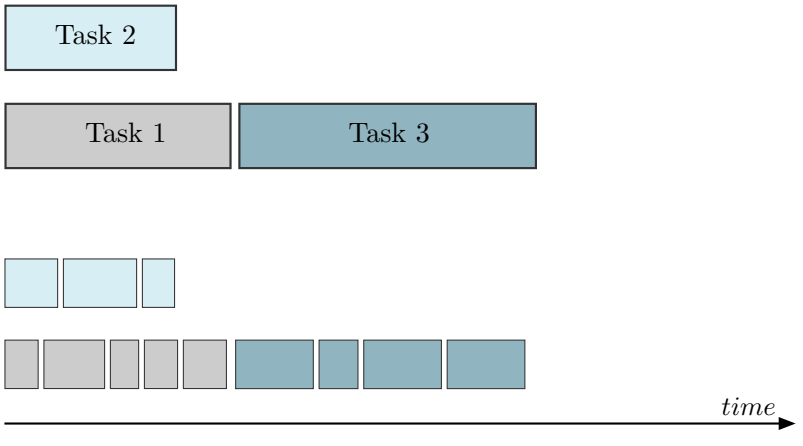


ပုံ ၁.၁ Sequential ပရိုဂရမ် အလုပ်လုပ်ပုံ။ အပေါ်က higher level task သုံးခုရဲ့ sequential သဘောကို ပြတာ။ အောက်ဘက် စတုဂံအသေးတွေက အပေါ်သုံးခုကို တစ်ခုချင်း အသေးစိတ် ထပ်ခွဲကြည့်တဲ့အခါမှာလည်း sequential ပဲဖြစ်နေတဲ့ သဘောကို ပြတာ။

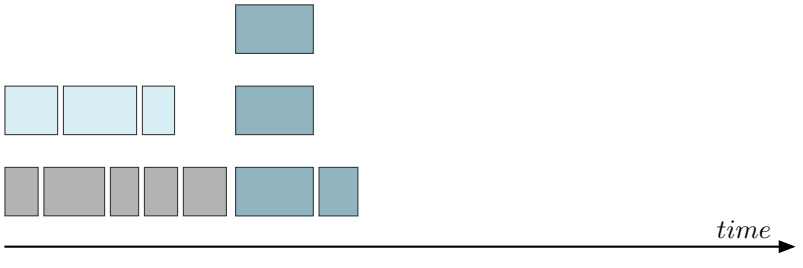
Sequential ပရိုဂရမ်တွေလည်း သူ့နေရာနဲ့သူ အသုံးဝင်ပါတယ်။ ဒါပေမဲ့ တချို့ ပရိုဂရမ်တွေဟာ အလုပ်တွေကို တစ်ချိန်တည်း လုပ်လို့ရအောင် စီစဉ်လို့ရနိုင်မှပဲ အဆင်ပြေ အလုပ်ဖြစ်မဲ့ သဘောရှိတယ်။ ဒီလို ပရိုဂရမ်မျိုးတွေကို concurrent ပရိုဂရမ်လို့ ခေါ်တယ်။ Graphical user interface (GUI)

ပရိုဂရမ်တွေ၊ ဝဘ်အပ်ပလီကေးရှင်းတွေ၊ ဒေတာဘေ့စ်တွေ စတာတွေမှာ concurrency ဟာ မဖြစ်မနေ လိုအပ်တယ်။ GUI ပရိုဂရမ် တစ်ခုဟာ ဂရပ်ဖစ် ပုံဖော်ပေးတာရော ကီးဘုဒ်နဲ့ မောက်စ်ကနေ အသုံးပြုသူ လုပ်လိုက်တဲ့ အဖြစ်အပျက်တွေကိုပါ အတုံ့အလှည့် လုပ်ဆောင်ပေးနေတာပါ။ တစ်ခါတစ်ရံ နောက်ကွယ်မှာ အလုပ်တစ်ခုကို အချိန်အတော်ကြာ လုပ်ဆောင်နေပြီး တစ်ချိန်တည်းမှာ အသုံးပြုသူကို အခြားကိစ္စတွေကို လုပ်ဆောင်လို့ ရနေစေတယ်။ ဥပမာ စာစီစာရိုက် ဆော့ဖ်ဝဲ နောက်ကွယ်မှာ စာလုံးပေါင်းသတ်ပုံနဲ့ သဒ္ဒါစစ်ပေးနေပြီး စာလည်းဆက်ရေးလို့ ရနေတာမျိုး။ ဝဘ်နဲ့ ဒေတာဘေ့စ်လို client-server အပ်ပလီကေးရှင်းတွေမှာတော့ concurrent user အများအပြားရဲ့ request တွေကို ဖြည့်ဆည်းပေးဖို့ လုပ်ဆောင်ရပါတယ်။ ဒီအချက်တွေကို ကြည့်ခြင်းအားဖြင့် ဘာကြောင့် concurrency အရေးပါလဲ ထင်ထင်ရှားရှား မြင်နိုင်မယ် ထင်ပါတယ်။

ပုံ (၁.၂) မှာ task 1 နဲ့ task 2 ဟာ concurrent ဖြစ်တယ်။ ၎င်းတို့ရဲ့ အခွဲ subtask တွေကလည်း concurrent ဖြစ်တာ တွေ့ရပါမယ်။ ဒီသဘောကို အောက်ဘက် စတုဂံ အသေးနှစ်တန်းနဲ့ ပြထားတာ တွေ့နိုင်တယ်။ Higher level task တွေပဲ concurrent ဖြစ်နိုင်တာ မဟုတ်ပါဘူး။ Lower level subtask တွေကိုလည်း လိုအပ်ရင် concurrent လုပ်ဆောင်စေနိုင်တယ်။ ပုံ (၁.၃) မှာ task 3 အခွဲ subtask သုံးခုကို concurrent ဖြစ်တာကို တွေ့ရပါမယ်။ Concurrent ပရိုဂရမ်တစ်ခုမှာ ဘယ် task တွေကို concurrent လုပ်မလဲ၊ ဘယ်ဟာတွေကိုတော့ sequential လုပ်ဆောင်စေမလဲ လိုအပ်သလို စီစဉ်နိုင်ပါတယ်။



ပုံ ၁.၂ Concurrent ပရိုဂရမ် ဥပမာတစ်ခု။ Task 1 နဲ့ task 2 ဟာ concurrent ဖြစ်တယ် (အပေါ်)။ အောက်က စတုဂံ အသေးတွေက task 1 နဲ့ task 2 ရဲ့ subtask တွေ concurrent လုပ်ဆောင်ပုံကို ပြထားတာ။



ပုံ ၁.၃ Task 3 ရဲ့ subtask တချို့ concurrent ဖြစ်နေပုံ။

၁.၀ Concurrency vs. Parallelism

Concurrency အကြောင်း ဖော်ပြတဲ့အခါ *parallelism* ကိုလည်း ချန်ထားခဲ့လို့မရပါဘူး။ အကြောင်းအရာနှစ်ခုက ဆက်နွယ်နေတယ်၊ စကားလုံး တစ်ခုနဲ့တစ်ခု သဘောတရား တူသလိုလို ဖလှယ်သုံးနှုန်းကြတယ် ဆိုပေမဲ့ တကယ်တမ်း တိတိကျကျ စဉ်းစားပြောရင် အဓိပ္ပါယ် မတူကြဘူး။ အသုံးချရတဲ့ ရည်ရွယ်ချက်ရော အခြေအနေပါ မတူတာမို့လို့ concurrent နဲ့ parallel အခြေခံသဘောတရားအားဖြင့် ဘာကွာခြားလဲ ခွဲခြားမြင်ဖို့ အရေးကြီးပါတယ်။

လုပ်စရာ task တွေ အများအပြားရှိမယ်၊ တစ်ခုစီကို အလှည့်ကျ အချိန်နည်းနည်းစီပေးပြီး အားလုံးရှေ့ကို တိုးတက်မှုရှိနေအောင် မျှမျှတတ ခွဲဝေလုပ်ဆောင်ပေးတာဟာ concurrency သဘောတရားဖြစ်တယ်။ ပုံ (၂.၂) မှာ ဒီသဘောတရားကို တွေ့နိုင်ပါတယ်။ Parallelism ဆိုတာကတော့ task တစ်ခု (သို့) task တွေ အများကြီးကို မြန်နိုင်သမျှ မြန်မြန်ပြီးအောင် multi-processor (သို့) multi-core CPU ကို ခွဲပေးပြီး လုပ်ဆောင်စေတာပါ။ Concurrency ဟာ parallelism အတွက် ဘယ်လို အထောက်အကူ ပေးနိုင်လဲဆိုတာ နောက်ပိုင်းမှာ တွေ့ရပါမယ်။

အဆောက်အအုံ နံရံတစ်ခု အုတ်စီတာကို စိတ်ကူးကြည့်ပါ။ ပန်းရန်ဆရာ တစ်ယောက်တည်း ရှိတယ်ဆိုရင် အင်္ဂတေ (အရပ်ခေါ် မဆလာ) နဲ့ အုတ်သယ်တာပါ တွဲလုပ်ရပါမယ်။ အုတ်ခဲ (သို့) အင်္ဂတေ သယ်တဲ့အခါ အုတ်စီတာ ခဏရပ်ရပါမယ်။ တစ်ချိန်တည်းမှာ အုတ်စီတာရော အင်္ဂတေသယ်တာပါ တစ်ပြိုင်နက် လုပ်တာမဟုတ်ဘူး။ ဒါပေမဲ့ နံရံစီတာ တိုးတက်မှုရှိသလို အင်္ဂတေနဲ့ အုတ်သယ်တာကလည်း ရပ်မနေပါဘူး။ ဒါဟာ concurrency သဘောတရား ဖြစ်ပါတယ်။ အလုပ်တွေကို မျှလုပ်ပေးနေတာ။ အာလုံးကလည်း တစ်ချိန်တည်းမှာ တိုးတက်မှု ရှိနေကြတာ တွေ့ရပါမယ်။

ပန်းရန်ဆရာ နောက်ထပ်တစ်ယောက် ထပ်ခေါ်မယ်ဆိုပါစို့။ နံရံတစ်ခုတည်းကို နှစ်ယောက်ဝိုင်းလုပ်ရင် နှစ်ဆပိုပြီး မြန်မြန်ပြီးမယ်။ ဒါမှမဟုတ် တစ်ယောက် နံရံတစ်ဖက် တာဝန်ယူမယ်ဆိုရင် တစ်ချိန်တည်းမှာ နှစ်ခုပြီးမှာပါ။ အုတ်နဲ့ အင်္ဂတေ သယ်ဖို့အတွက်ပါ လူသပ်သပ် ထပ်ခေါ်လိုက်မယ် ဆိုရင်တော့ အလုပ်အများကြီး ပိုတွင်ကျယ်လာနိုင်ပါတယ်။ ဒါဟာ parallelism သဘောတရား ဖြစ်ပါတယ်။

ပရိုဂရမ်တွေကို concurrency နဲ့ parallelism ရှုထောင့်ကနေ အောက်ပါအတိုင်း ၄ မျိုး ခွဲကြည့်နိုင်တယ်။ (အခြေခံ အဆင့်အနေနဲ့ အခုလောက် အသေးစိတ် သိဖို့ လိုချင်မှ လိုမှာပါ၊ ဒါပေမဲ့ နားလည်ထားရင် သဘောတရားပိုင်းဆိုင်ရာ အထောက်အကူ ဖြစ်ပါတယ်)။

- Sequential (တစ်နည်းအားဖြင့် Concurrent လည်းမဟုတ်၊ parallel လည်းမဟုတ်)
- Concurrent ဖြစ်တယ်၊ ဒါပေမဲ့ parallel မဟုတ်
- Parallel ဖြစ်တယ်၊ ဒါပေမဲ့ concurrent မဟုတ်
- Concurrent လည်းဖြစ်တယ်၊ parallel လည်းဖြစ်တယ်

Sequential ပရိုဂရမ်တစ်ခုဟာ task တွေကို တစ်ခုပြီးမှ နောက်တစ်ခု ဆက်လုပ်တယ်။ တစ်ခုမပြီးသေးခင် အခြားတစ်ခုကို မစသေးဘူး။ ဒါကြောင့်မို့ တစ်ချိန်မှာ task တစ်ခုကပဲ တိုးတက်မှုရှိနေမှာပါ။ Concurrent ပရိုဂရမ်တစ်ခုမှာတော့ task အများအပြား လုပ်ဆောင်တယ်။ အားလုံးကို အချိန်ခွဲဝေ မျှလုပ်ပေးတယ်။ Task တစ်ခုစီတိုင်း အချိန်နဲ့အမျှ တိုးတက်မှုရှိနေမယ်။ ဒါပေမဲ့ ဒါဟာ parallel တော့ မဟုတ်သေးဘူး။ အလှည့်ကျ မျှပြီး လုပ်ပေးနေတာ။ CPU တစ်ခုတည်းနဲ့ပဲ task တစ်ခုမက concurrent လုပ်လို့ရတယ်။ အကယ်၍ concurrent ပရိုဂရမ်ကို multi-processor/multi-core CPU ကွန်ပျူတာပေါ် run ရင် concurrent လည်းဖြစ်၊ parallel လည်းဖြစ်တယ် ယူဆရမှာပါ။ မြင်သာအောင် နံရံအုတ်စီတဲ့ ဥပမာနဲ့ ခိုင်းနှိုင်းကြည့်နိုင်တယ်။ ပန်းရံဆရာ တစ်ယောက်တည်းကပဲ အုတ်စီ၊ အင်္ဂတေ/အုတ် သယ်တာက concurrent။ ပန်းရံဆရာက အုတ်စီပြီး အင်္ဂတေ/အုတ် သယ်တာ အခြားတစ်ယောက်က လုပ်ပေးနေရင် concurrent ရော parallel ပါ ဖြစ်တဲ့သဘော။

Parallel တော့ဖြစ်တယ်၊ concurrent မဟုတ်တာလည်း ရှိသေးတယ်။ Task က တစ်ခုတည်းပဲ၊ မြန်မြန်ပြီးအောင် ကွန်ပျူတာစွမ်းအားအရင်းအမြစ် အရေအတွက် (ဥပမာ CPU) များများသုံးပြီး လုပ်တာကို ဆိုလိုတာ။ ကိန်းဂဏန်း အလုံး တစ်သန်းကို ပေါင်းမယ်ဆိုပါစို့။ အလုံးတစ်သန်းကို နှစ်သိန်းခွဲစီ လေးပိုင်း ပိုင်းပြီး တစ်ပိုင်းချင်း ပေါင်းတဲ့အခါ quad-core CPU နဲ့ ဆိုရင် ပုံမှန်ထက် လေးဆနီးပါး ပိုပြီးမြန်မှာပါ။ ဒီလိုမျိုးကို parallel ဖြစ်ပေမဲ့ concurrent မဟုတ်ဘူးလို့ ယူဆပါတယ်။ အုတ်စီတဲ့အလုပ် တစ်ခုတည်းကိုပဲ လေးယောက်ခွဲပြီး လုပ်တာနဲ့ သဘောတရားတူတယ်။ (ဒီကိစ္စမှာ အုတ်ခဲ/အင်္ဂတေ သယ်တဲ့ကိစ္စပါ ထည့်စဉ်းစားရင်တော့ task တစ်ခုမကတော့တဲ့အတွက် concurrent လည်းဖြစ်တယ်၊ parallel လည်းဖြစ်တယ် ယူဆနိုင်မယ်)။

၁.၂ Threads, Process and Concurrency

Operating system ဆိုတာ ကွန်ပျူတာပေါ်မှာ run တဲ့ ပရိုဂရမ်တွေကို စီမံကွပ်ကဲတဲ့ စနစ်လို့ ယေဘူယျ နားလည်နိုင်တယ်။ ကနေ့ခေတ် ကွန်ပျူတာတွေမှာ အပ်ပလီကေးရှင်း ပရိုဂရမ်တွေ အသုံးပြုနိုင်ဖို့ operating system ရှိရပါမယ်။ ကွန်ပျူတာတစ်လုံးမှာ အပ်ပလီကေးရှင်းတွေ အများကြီး ဖွင့်ထားပြီး တစ်ပြိုင်တည်း သုံးလို့ရအောင် operating system က စီမံပေးထားတာ။ Operating system တစ်ခုပေါ်မှာ run နေတဲ့ ပရိုဂရမ်ကို process လို့ခေါ်တယ်။ အပ်ပလီကေးရှင်းတစ်ခုမက တစ်ပြိုင်တည်း သုံးလို့ရတာကို *multitasking* ရတယ်လို့ ပြောလေ့ရှိပြီး အဲဒီလို သုံးလို့ရအောင် ဆောင်ရွက်ပေးနိုင်တဲ့ operating system ကို Multitasking Operating System ခေါ်တယ်။ Multitasking ကို concurrency ပုံစံတစ်မျိုး အနေနဲ့ ယူဆနိုင်တယ်။ Multitasking ဆိုတာလည်း တစ်ချိန်တည်းမှာ အလုပ်တစ်ခုမက လုပ်နေတာပဲလေ။ ဒါပေမဲ့ operating system က multitasking ရအောင် ထောက်ပံ့ပေးထားပြီးသား ဆိုတော့ ပုံမှန်အားဖြင့် ပရိုဂရမ်မာက ထူးထူးထွေထွေ ဘာမှလုပ်စရာမလိုဘူး။

အခု ကျွန်တော်တို့ အဓိကစိတ်ဝင်စားတာက process (run နေတဲ့ ပရိုဂရမ်) တစ်ခုမှာ ပါဝင်တဲ့ task တွေနဲ့ သက်ဆိုင်တဲ့ concurrency ပါ။ Operating system က စီမံကွပ်ကဲတဲ့ process တစ်ခုရဲ့ subtask တွေကို တစ်ပြိုင်တည်း လုပ်ဆောင်ဖို့ *thread* ကို အသုံးပြုရပါတယ်။ Process တစ်ခုဟာ sequential ပဲ လုပ်ဆောင်တဲ့အခါ thread တစ်ခုပဲ လိုအပ်တယ်။ Task တွေကို concurrent လုပ်ဆောင်စေချင်ရင် အဲဒီ task တစ်ခုချင်းကို သီးခြား thread တစ်ခု အနေနဲ့ လုပ်ဆောင်ပေးဖို့ operating system ကို ခိုင်းရပါမယ်။ Programming language တွေမှာ thread ဖန်တီးအသုံးပြုရတာ လွယ်ကူအဆင်ပြေစေမဲ့ ဖီချာတွေ ထည့်သွင်းပေးထားလေ့ရှိပါတယ်။

Python မှာ task နှစ်ခု concurrent လုပ်ဆောင်ဖို့ thread အသုံးပြုတဲ့ အရိုးရှင်းဆုံး ဥပမာ တစ်ခုကို ကြည့်ရအောင်။ ၁ ကနေ ၁၀ ထိ ထုတ်ပေးတဲ့ task နဲ့ A ကနေ J ထုတ်ပေးတဲ့ task နှစ်ခု ရှိတယ် ယူဆပါ။ Python Standard Library မှာ ပါဝင်တဲ့ threading မော်ဂျူး အသုံးပြုပြီး task တစ်ခုကို thread တစ်ခုစီမှာ concurrent run မှာပါ။

```python
import threading
import time

# Task 1: Print numbers from 1 to 10
def print_numbers():
    for i in range(1, 11):
        print(f"{i}, ", end="")
        time.sleep(0.2)  # Simulate some work with a delay

# Task 2: Print letters from A to J
```

```
def print_letters():
    for letter in ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']:
        print(f"{letter}, ", end="")
        time.sleep(0.2)  # Simulate some work with a delay

print("Main thread started.")

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

print("Main thread starts both threads.")
thread2.start()
thread1.start()

# Wait for both threads to finish
thread1.join()
thread2.join()
print()
print("Both tasks completed!")
```

ပရိုဂရမ်တစ်ခု run တဲ့အခါ operating system က process တစ်ခု ဖန်တီးပါတယ်။ (Process ဆိုတာ operating system စီမံမှုအောက်မှာရှိတဲ့ run နေတဲ့ ပရိုဂရမ်ဆိုလို့ ပြောခဲ့တယ်)။ စစချင်းမှာ အဲဒီ process ဟာ ပရိုဂရမ်ညွှန်ကြားချက်တွေကို လုပ်ဆောင်ပေးမဲ့ thread တစ်ခုပဲ ရှိပါမယ်။ *main* thread လို့ ခေါ်တယ်။ Sequential ဖြစ်စေ၊ concurrent ဖြစ်စေ thread တစ်ခုတော့ ရှိကိုရှိရပါမယ်။ ပရိုဂရမ် စစချင်းမှာ အဲဒီ thread က အလုပ်စလုပ်မှာပါ။ main thread လို့ ခေါ်ပါတယ်။ main thread ကနေ အခြား thread တွေ ဖန်တီးပြီး task တွေကို concurrent run နိုင်ပါတယ်။

Thread တစ်ခု လုပ်ဆောင်ပေးရမဲ့ task ကို `target=`*task_name* နဲ့ ညွှန်ပြပေးနိုင်ပါတယ်။ ရှေ့ကဥပမာမှာ thread နှစ်ခုကို ဖန်တီးထားတာ အခုလိုတွေ့ရမှာပါ

```
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)
```

လုပ်ဆောင်စေချင်တဲ့ ဖန်ရှင်နံမည်ကို target အနေနဲ့ ထည့်ပေးထားတယ်။ Thread တစ်ခုက ဘယ် task ကို run ပေးရမလဲ ညွှန်ပြဖို့ ဒီနည်းလမ်းတစ်ခုတည်း ရှိတာတော့ မဟုတ်ဘူး။ အခြားနည်းလမ်းတွေ ရှိပါသေးတယ်။ နောက်ပိုင်းမှာ တွေ့ရပါမယ်။

Thread နှစ်ခု ဖန်တီးထားပြီးပြီ။ ဒါပေမဲ့ thread ဖန်တီးထားရုံနဲ့ အဲဒီ thread က အလုပ်စတာ မဟုတ်ပါဘူး။ `start` ဖန်ရှင် ခေါ်ပေးရပါမယ်

```
thread1.start()
thread2.start()
```

သတိပြုသင့်တာ တစ်ခုက `start` ခေါ်ပြီးတာနဲ့ ချက်ချင်း အလုပ်စလုပ်မယ်လို့လည်း အာမခံချက် မရှိဘူး။ `start` ဟာ thread တစ်ခု အလုပ်စလို့ရပြီဆိုတာ အရိပ်အမြွက် ဖော်ပြတဲ့သဘောလို့ပဲ ယူဆရမယ်။ `start` ခေါ်ပြီးတဲ့အခါ thread က *ready* state ကို ရောက်သွားတယ်။ Thread တွေကို operating system က စီမံတယ်လို့ ပြောခဲ့တာ ပြန်အမှတ်ရပါ။ ဘယ်အချိန်မှာ တကယ် အလုပ်စလုပ်

မလဲ၊ ဘယ်လို အလှည့်ကျ လုပ်ဆောင်စေမလဲ စတာတွေက operating system ထိန်းချုပ်မှုအောက်မှာ ရှိတာဖြစ်တယ်။ ဒါကြောင့် *ready* state မှာ ရှိတဲ့ thread တစ်ခုဟာ operating system က ၎င်းကို အလှည့်ချပေးတဲ့အခါမှပဲ အမှန်တကယ် အလုပ်စလုပ်လို့ရမှာ ဖြစ်တယ်။ လက်ရှိအချိန်မှာ အလုပ်လုပ်နေတဲ့ thread ရဲ့ state ကို *running* state လို့ သတ်မှတ်တယ်။

Concurrent task တွေ တစ်ခုနဲ့တစ်ခု coordinate လုပ် (ပူးပေါင်းဆောင်ရွက်) ဖို့ လိုလေ့ရှိတယ်။ Thread တစ်ခုက အခြား thread တစ်ခု ပြီးတဲ့ထိ စောင့်တာဟာ coordination ပုံစံတစ်မျိုးပဲ။ အခြား coordination ပုံစံတွေလည်း အတော်များများ ရှိတယ်။ ဥပမာ thread တစ်ခုက အခြား thread တစ်ခု ထုတ်ပေးလိုက်တာကို ယူသုံးတာကလည်း coordinate လုပ်တာပဲ။ အခုဥပမာမှာ main thread ဟာ `thread1` နဲ့ `thread2` ပြီးအောင် စောင့်ရပါမယ်။ လက်ရှိ thread က အခြား thread တစ်ခု ပြီးတဲ့ထိ စောင့်မယ်ဆိုရင် `join` ကို ခေါ်ပေးရပါမယ်။

```
# Wait for both threads to finish
thread1.join()
thread2.join()
```

ဒီလိုဆိုရင် `thread1` နဲ့ `thread2` ပြီးတော့မှ main thread ဆက်အလုပ်လုပ်မှာပါ။ ဘယ် thread ပြီးတဲ့ထိ စောင့်ရမလဲ။ အဲ့ဒီ thread ရဲ့ `join` ကို စောင့်နေချင်တဲ့ thread က ခေါ်ပေးရမှာ။

အခုဥပမာကို အကြိမ်အနည်းငယ် run ကြည့်ပါ။ အခု ဒီစာရေးနေရင်း စက်မှာ သုံးခေါက် run တာ အခုလို ထွက်တယ်

```
Main thread started.
Main thread starts both threads.
1, A, 2, B, 3, C, 4, D, 5, E, 6, F, 7, G, 8, H, 9, I, 10, J,
Both tasks completed!


Main thread started.
Main thread starts both threads.
1, A, 2, B, 3, C, 4, D, 5, E, F, 6, G, 7, 8, H, 9, I, 10, J,
Both tasks completed!


Main thread started.
Main thread starts both threads.
1, A, B, 2, C, 3, 4, D, 5, E, F, 6, G, 7, H, 8, I, 9, J, 10,
Both tasks completed!
```

အခု output တွေမှာ သိသိသာသာ ဖြစ်နေတာတစ်ခုက thread နှစ်ခုက ထုတ်ပေးတဲ့ အက္ခရာနဲ့ ဂဏန်းတွေဟာ အစီအစဉ်တစ်ခုနဲ့ ပုံသေအမြဲ မဖြစ်နေဘူး။ တစ်ခေါက်ကို တစ်မျိုးစီ။ ဂဏန်းတွေဖယ်ပြီး အက္ခရာတွေပဲ ကြည့်ရင် A, B, C, D, … အစဉ်တိုင်းဖြစ်တယ်။ အက္ခရာတွေမပါဘဲ ဂဏန်းတွေချည်း ဆိုလည်း အစဉ်အတိုင်းပါပဲ။ ဒါပေမဲ့ thread နှစ်ခုပေါင်း ထုတ်ပေးတဲ့ output အစီအစဉ်ကိုတော့ ခန့်မှန်းလို့ မရဘူး။ ဒါဟာ thread သုံးတဲ့အခါ သတိပြုရမဲ့ အခြေခံကျတဲ့ အချက်တစ်ခု ဖြစ်တယ်။ ဘယ် thread ကို ဘယ်ချိန်မှာ run ဖို့ အလှည့်ချပေးမလဲ၊ ဘယ်လို အလှည့်ကျ စနစ်သုံးမလဲက operating system နဲ့ သက်ဆိုင်တာ။ ပရိုဂရမ်မာအနေနဲ့ အဲဒီ အလှည့်ကျစနစ်ရဲ့ timing ကို ခန့်မှန်းတွက်ဆပြီး လိုချင်တဲ့ အစီအစဉ်ဖြစ်အောင် လုပ်ဖို့ မဖြစ်နိုင်ဘူး။

`join` နဲ့ဆိုင်တဲ့ နှစ်ကြောင်း ကွန်းမန့်ပိတ်ပြီး run ကြည့်ပါ။ Task နှစ်ခု မပြီးသေးဘဲ ပြီးသွားပြီ

ဆိုတဲ့စာသား ထွက်လာတာတွေ့ရပါမယ်။ Main thread က သူ့အလှည့် ရောက်တဲ့အခါ `thread1` နဲ့ `thread2` ပြီးအောင် မစောင့်ဘဲ ဆုံးတဲ့ထိ ဆက်လုပ်သွားတာကြောင့် ဖြစ်တယ်။

```
Main thread started.
Main thread starts both threads.
1, A,
Both tasks completed!
2, B, 3, C, 4, D, E, 5, F, 6, G, 7, H, 8, 9, I, 10, J,
```

၁.၃ Race Conditions

Sequential ဆိုရင် တစ်ခုပြီးမှ နောက်တစ်ခု ဆက်လုပ်လို့ရမှာ။ ဂဏန်းအားလုံး ထုတ်ပေးပြီးမှ အက္ခရာဆက်ထုတ်လို့ရမယ်။ ဒါမှမဟုတ် အက္ခရာတွေပြီးမှ ဂဏန်းတွေ ထုတ်လို့ရမှာပါ။

```
import threading
import time

def print_numbers():
    # same as before

def print_letters():
    # same as before

# Task 3: Print from M1 to M10
def print_seq_of_m():
    for letter in ['M1', 'M2', 'M3', 'M4', 'M5',
                   'M6', 'M7', 'M8', 'M9', 'M10']:
        print(f"{letter}, ", end="")
        time.sleep(0.2)  # Simulate some work with a delay

print("Main thread started.")

# Create two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Start both threads
thread2.start()
thread1.start()

# run task 3 in main thread
print_seq_of_m()

# Wait for both threads to finish
thread1.join()
thread2.join()
print()
print("Both tasks completed!")
```

```
# run task 3 in main thread
print_seq_of_m()

# Start both threads
thread2.start()
thread1.start()
```