Diskrétní simulace za použití knihovny SimPy

Panovský Tomáš

9. února 2026

Obsah

1 Úvod

V této bakalářské práci se zabývám diskrétní simulací za použití knihovny SimPy v Pythonu. Knihovna Simpy umožňuje modelovat procesy, jenž probíhají souběžně, a mohou být zastaveny, nebo pozastaveny na určitou dobu. Praktická část bakalářské práce obsahuje aplikaci simulující průběh hudebního festivalu. Cílem simulace je zjistit, jak se návštěvníci pohybují a kde vznikají fronty, což může pomoci při organizaci reálného festivalu.

2 Systém

str17 Reálnou skutečnost, kterou chceme analyzovat, označujeme jako systém. Analyzovat znamená sledovat, jak se různé části systému chovají v čase, a vyvozovat závěry o efektivitě, kapacitě nebo chování systému.

Na systémy můžeme nahlížet jako diskrétní nebo spojité, přičemž záleží na úhlu pohledu a vlastnostech, které v systému převažují. **Diskrétní systém je takový, u kterého se stavové proměnné mění pouze v diskrétních časových okamžicích.** Příkladem diskrétního systému je hudební festival, kde se stavové proměnné, například počet návštěvníků ve frontě u stánku s pivem, mění pouze tehdy, když návštěvník přijde na řadu, nebo dostane pivo a odejde. Oproti tomu spojitý systém je takový systém, u kterého se stavové proměnné mění plynule v čase, nikoli pouze v diskrétních okamžicích.

Stav systému je definován jako soubor stavových proměnných, které jsou nezbytné k popisu systému v libovolném okamžiku. V simulaci hudebního festivalu mohou být možné stavové proměnné například počet lidí čekajících ve frontách u stánků s občerstvením, počet lidí právě sledujících koncert, nebo čas příchodu dalšího návštěvníka do areálu.

2.1 Komponenty systému

Každý systém lze chápat jako soubor vzájemně propojených prvků, které společně ovlivňují jeho chování. Aby bylo možné systém lépe pochopit, je vhodné rozdělit jej na základní komponenty, které popisují jeho strukturu a dynamiku. Mezi tyto komponenty patří entity, jejich atributy, aktivity a události.

Entita je objekt zájmu v systému. V našem případě mohou být entitami například návštěvníci festivalu, stánky s občerstvením nebo pódium.

Atribut je vlastnost entity, například počet lidí u stánku, nebo informace o tom, zda má návštěvník hlad nebo je unavený.

Aktivita představuje časově omezenou činnost entity, například čekání ve frontě, sledování koncertu nebo nákup jídla.

Událost je okamžitá změna stavu systému. Události dělíme na Endogenní a Exogenní. Endogenní probíhají uvnitř systému a jsou způsobeny chováním jeho komponent, např. návštěvník dokončí konzumaci jídla a odchází od stánku. Exogenní probíhají v prostředí systému a ovlivňují ho zvenčí, například náhlý déšť.

3 Úvod do simulace

str6 Simulace je napodobení systému v čase. Cílem je vytvořit umělou historii stavu daného systému, a následně pozorovat tuto uměle vytvořenou historii za účelem vyvození závěrů o provozních vlastnostech systému, tedy o měřitelných charakteristikách a výkonnosti systému, jako je například délka front u stánků, průměrná doba čekání návštěvníků nebo hustota návštěvníků u pódia.

Chování systému vyvíjejícího se v čase se zkoumá pomocí simulačního modelu. V této práci je simulační model vytvořen v knihovně SimPy. Simulační model poté může být použit k prozkoumání široké škály otázek typu „co by se stalo, kdyby“ týkajících se reálného systému. Například můžeme zkoumat, zda je daný počet stánků s občerstvením dostatečný pro obsloužení všech návštěvníků festivalu bez dlouhých front.

Možné změny systému pak lze nejprve nasimulovat, aby bylo možné předpovědět jejich dopad na výkonnost systému. Simulace může být také použita ke studiu systémů ve fázi návrhu, ještě před jejich samotnou realizací.

4 Model systému

str. 13 **Model je definován jako reprezentace systému za účelem jeho studia.** Pro většinu studií je nutné zvažovat pouze ty aspekty systému, které ovlivňují problém, který je předmětem zkoumání.

Modely lze klasifikovat jako statické nebo dynamické, deterministické nebo stochastické a diskrétní nebo spojité. Statický simulační model reprezentuje systém v určitém časovém okamžiku, zatímco dynamické modely reprezentují systémy, jak se mění v čase. Deterministické simulační modely neobsahují žádné náhodné prvky, a to ani ve vstupních datech, ani v průběhu samotné simulace. Při opakovaném spuštění se stejnými vstupy poskytují vždy totožný průběh i výsledek simulace. Stochastické modely naproti tomu obsahují jeden nebo více náhodných prvků, které mohou vstupovat do simulace jak na jejím začátku, tak v jejím průběhu, například při generování časů událostí nebo rozhodování o chování entit. Díky tomu lépe vystihují systémy, jejichž chování je ovlivněno náhodou. Výsledky takových simulací nejsou jednoznačné, ale mají pravděpodobnostní charakter. Diskrétní a spojité modely jsou definovány obdobně jako u systémů.

Pro studium hudebního festivalu použijeme diskrétní, dynamický a stochastický model. Diskrétní model, protože stavové proměnné se mění pouze v konkrétních okamžicích. Dynamický model, protože sleduje vývoj systému v čase během celé doby trvání festivalu. Stochastický model, protože některé vstupy, například časy příchodů návštěvníků, doba čekání u stánku nebo délka sledování koncertu, jsou náhodné.

5 Simulace v SimPy

Pro realizaci diskrétní simulace jsem zvolil jazyk Python a knihovnu SimPy. Ta je postavena především na generátorech (využívající příkaz `yield`), které umožňují popis aktivit a událostí entit.

5.1 Generátory v Pythonu

V jazyce Python je iterátor objekt, který umožňuje postupné získávání hodnot bez nutnosti mít všechny hodnoty uložené v paměti. Iterátor si pamatuje svůj aktuální stav a při každém volání funkce next() vrací další prvek.

Zvláštní formou iterátorů jsou generátory, které obsahují v těle funkce příkaz `yield`. Příkaz `yield` umožňuje generátoru postupně produkovat jednotlivé prvky

posloupnosti a může teoreticky produkovat i nekonečnou posloupnost dat. Posloupnost zde znamená řadu hodnot, které generátor postupně poskytuje. Příkaz produkující prvek posloupnosti nabývá tvaru: `yield element`. Po provedení příkazu yield se generátor pozastaví a vrátí hodnotu specifikovanou příkazem `yield`. Při dalším volání pokračuje ve vykonávání od místa, kde byl přerušen.

Příklad generátoru:

```
def get_numbers():
   i = 0
   while True:
      yield i
      i = i + 1
```

Napřed pouze vytvoříme iterátor i:

```
i = get_numbers()
```

Další prvek můžeme vyžádat funkcí next. Dalším prvkem posloupnosti bude hodnota určená příkazem yield. Tedy:

```
next(i)
```

V tuto chvíli bude v proměnné i uložena 0. Tělo generátoru se začne vykonávat od pozastaveného místa až po příkaz yield. Vykonávání těla generátoru je pozastaveno na řádku:

```
i = i + 1
```

Popsaným způsobem získáme další hodnoty z generátoru, tedy při dalším volání `next(i)` bude v proměnné `i` 1, poté 2, a tak dále.

V kontextu diskrétní simulace v knihovně SimPy jsou generátory využity k modelování aktivit, které představují chování jednotlivých entit, například návštěvníků festivalu. Každý příkaz `yield` v generátoru odpovídá předání řízení simulátoru a obvykle vrací objekt typu `Event`, který reprezentuje událost, na jejíž dokončení aktivita čeká. Příkladem může být čekání ve frontě u stánku s jídlem, dokončení přípravy jídla či jeho obdržení zákazníkem. Takto lze simulovat souběžné aktivity více entit a stochastické prvky, například náhodné časy příprav nebo příchodů návštěvníků, což odpovídá reálnému chování systému.

5.2 Základní principy SimPy

Základní komponenty SimPy jsou prostředí simulace (jenž je v SimPy označováno jako Environment), události a procesní funkce. Smyslem prostředí je řídit celou simulaci, udržovat simulační čas a organizovat události. Události udávají, kdy má dojít ke změně stavu a společně tvoří časový plán simulace. Procesní funkce jsou python generátorové funkce, které generují události. Smyslem procesních funkcí je popisovat chování entit v simulaci. Na každou procesní funkci můžeme nahlížet jako na aktivity dané entity. Voláním procesní funkce vzniká entita, jejíž chování popisuje iterátor funkcí vrácený. Jeden z argumentů volání procesní funkce je vždy prostředí. Instance procesní funkce běžící v simulačním prostředí v kontextu SimPy a programového řízení simulace nazýváme proces.

Prostředí simulace řídí průběh simulační času a spravuje seznam plánovaných událostí. Simulační čas je čas, ve kterém simulace probíhá, nikoliv reálný čas. V SimPy je tento čas oproti reálnému času bez konkrétní jednotky a probíhá diskrétně. Simulační čas jsou tedy pouze diskrétní hodnoty jako například 0, 1, 2 a podobně. Jednotky času si můžeme určit sami podle toho, co simulujeme. Mohou to být například hodiny, minuty či sekundy. Standardně čas začíná hodnotou 0. Aktuální čas simulace můžeme získat zasláním zprávy `env.now()`. Každá událost má přiřazený čas, ve kterém nastane. Prostředí tedy vždy zpracuje událost naplánovanou na aktuální čas.

Jednotlivé aktivity reprezentované procesní funkcí mohou probíhat nezávisle na sobě. To, že aktivita probíhá nějakou dobu, lze v SimPy simulovat události `environment.timeout(time)`, která procesní funkci na stanovený čas pozastaví. Během doby, kdy je procesní funkce pozastavena, může prostředí vykonávat jiné události naplánované na aktuální čas simulace. K pozastavení procesní funkce dochází ve chvíli, kdy entita provádí časově omezenou aktivitu. Pozastavený proces tedy představuje období, během kterého simulovaná entita vykonává danou aktivitu. Procesní funkce lze pozastavit i dalšími typy událostí, které budou vysvětleny později.

Každá událost v SimPy má prioritu. Priorita je číslo, podle kterého se rozhoduje, která událost se zpracuje dříve, pokud je více událostí naplánovaných na stejný simulační čas. Ve výchozím nastavení mají všechny události stejnou prioritu, lze ji ale ručně nastavit, což umožňuje řídit pořadí zpracování, například aby určitá událost vždy předběhla jinou událost ve stejném časovém okamžiku. Avšak ve většině simulací, včetně této, není potřeba ruční nastatování priorit.

Každá událost má také interní identifikátor, který prostředí používá pro rozlišení dvou událostí se stejným časem a prioritou. Identifikátor se zvyšuje s každou novou událostí, takže simulace ví, která událost byla vytvořena dříve a měla by být zpracována první.

Abychom ilustrovali, jak SimPy pracuje s procesními funkcemi a jak mezi nimi

přepíná pomocí událostí, si představme jednoduchý scénář se dvěma roboty, kteří se pohybují v různých směrech. Každý robot představuje samostatnou entitu se svou vlastní aktivitou, která nějakou dobu trvá. Právě tuto dobu můžeme v SimPy modelovat událostí `env.timeout()`. V následujícím příkladu vznikne zavoláním procesní funkce `robot` entita `Robot A` a entita `Robot B` . Tělo procesní funkce zařídí, že vzniklý robot po náhodně dlouhou dobu půjde směrem vlevo nebo vpravo a poté se na jednu časovou jednotku zastaví. Pro určení doby, jak dlouho se roboti budou pohybovat použijeme modul `random` ze standardní knihovny Pythonu. Prostředí simulace mezi těmito dvěma aktivitami automaticky přepíná vždy ve chvíli, kdy některý z robotů čeká na dokončení události `timeout`.

```
env = simpy.Environment()

def robot(env, name, direction):
    while True:
        print(f"{env.now}: {name} se začal pohybovat směrem {direction}")
        yield env.timeout(random.randint(1, 3))
        print(f"{env.now}: {name} se zastavil a rozhlíží se")
        yield env.timeout(1)

env.process(robot(env, "Robot A", "vpravo"))
env.process(robot(env, "Robot B", "vlevo"))
env.run()
```

Nejprve je vytvořeno prostředí simulace `env`. Následně je definována procesní funkce `robot`, která představuje aktivitu jednoho robota. Procesní funkci je v argumentech předáno simulační prostředí `env`, jméno robota `name` a směr `direction`, kterým robot bude chodit. Prostředí následně pomocí zavolání env.process() zaregistrovalo procesní funkce `robot`, čímž došlo k vytvoření entit robotů. SimPy tak může řídit průběh jejích aktivit prostřednictvím událostí. Nakonec je simulace spuštěna zasláním zprávy `env.run()`. Výstup tohoto příkladu by vypadal následovně:

```
0: Robot A se začal pohybovat směrem vpravo
0: Robot B se začal pohybovat směrem vlevo
2: Robot A se zastavil a rozhlíží se
2: Robot A se začal pohybovat směrem vpravo
3: Robot B se zastavil a rozhlíží se
4: Robot B se začal pohybovat směrem vlevo
.
.
.
```

5.3 Prostředí simulace

Již víme, že simulační prostředí spravuje čas simulace, plánování a zpracování událostí, a poskytuje také prostředky pro postupné provádění simulace. Mimo to SimPy také nabízí simulační prostředí `RealtimeEnvironment`, které simulační čas synchronizuje s reálným časem. To umožňuje například běh simulace paralelně s reálnými událostmi, kdy čas simulace plyne stejně jako skutečný čas, to však v této práci nevyužijeme.

Další funkce prostředí je obsluha spuštění simulace. Simulace v SimPy se spouští zasláním zprávy `env.run()`, kde případný argument této zprávy záleží na tom, kdy má simulace skončit. Simulace obvykle končí po vyčerpání všech událostí, nebo po uplynutí předem stanoveného času. Pokud chceme, aby simulace skončila po vyčerpání všech naplánovaných událostí, necháme zprávu `env.run()` bez argumentu. Speciálním případem takto spuštěné simulace může být nekonečná simulace, a to například když kód procesní funkce běží ve smyčce `while True:`, jako jsme si ukázali v příkladu s roboty. Tato simulace samovolně nikdy neskončí a musíme ji ukončit násilně. V případě, že chceme ukončit simulaci po uplynutí předem stanoveného času zašleme zprávu `env.run()` s argumentem `until=time`, kde `time` je hodnota datového typu `integer`, na které se má simulační čas zastavit, například `env.run(until=10)`. Simulace se zastaví v okamžiku, kdy čas dosáhne hodnoty 10, ale neprovede žádné události naplánované na čas 10. Poslední variantou je spuštění simulace do doby, než nenastane konkrétní událost `env.run(until=event)`.

Prostředí také poskytuje metody pro ruční krokování simulace, to je užitečné zejména pro testování a ladění simulace po jednotlivých krocích. Zaslání zprávy `peek()` vrací čas nejbližší naplánované události, aniž by byla událost vykonána, což umožňuje kontrolu plánovaného průběhu simulace. Zaslání zprávy `step()` zpracuje nejbližší naplánovanou událost a posune aktuální čas simulace na čas této události. Pokud nejsou k dispozici žádné další události, vyvolá metoda výjimku `EmptySchedule`, čímž signalizuje konec simulace.

Zprávy `peek()` a `step()` si ukážeme na následujícím jednoduchém příkladu, kde budeme mít dvě procesní funkce `procesni_funkce_1` a `procesni_funkce_2`, a každá procesní funkce představuje jednoduchou aktivitu, která u první funkce trvá 3 časové jednotky a u druhé 5 časových jednotek.

```
def procesni_funkce_1(env):
    print(f"t = {env.now}: start aktivity_1")
    yield env.timeout(3)
    print(f"t = {env.now}: konec aktivity_1")

def procesni_funkce_2(env):
    print(f"t = {env.now}: start aktivity_2")
```

```
    yield env.timeout(5)
    print(f"t = {env.now}: konec aktivity_2")

env = simpy.Environment()
env.process(procesni_funkce_1(env))
env.process(procesni_funkce_2(env))
```

Následně budeme na střídavě zasílat zprávy `peek()` a `step()`. Připomeňme, že `peek()` vrací čas nejbližší naplánované události, a `peek()` zpracuje nejbližší událost.

```
print(env.peek())
env.step()
print(env.peek())
env.step()
print(env.peek())
env.step()
print(env.peek())
env.step()
print(env.peek())
env.step()
print(env.peek())
env.step()
```

První zaslání zprávy `peek()` vrátí 0, protože start procesů `procesni_funkce_1` a `procesni_funkce_2` je naplánován na čas 0. Po prvním zaslání zprávy `env.step()` se spustí první proces `procesni_funkce1` a vypíše se: `t = 0: start aktivity_1` Proces narazí na `yield env.timeout(3)`, což naplánuje timeout událost v čase 3. Druhé zaslání zprávy `peek()` vrátí 0, protože další nejbližší událost (start druhého procesu `procesni_funkce2`) je stále naplánovaná na čas 0. Druhé zaslání zprávy `env.step()` spustí start druhého procesu `procesni_funkce2` a vypíše se: `t = 0: start aktivity_2`. Proces narazí na `yield env.timeout(5)`, což naplánuje timeout událost v čase 5. Třetí volání `peek()` vrátí 3, protože nejbližší naplánovaná událost je timeout prvního procesu. Třetí zaslání zprávy `env.step()` zpracuje timeout první procesní funkce. Simulační čas se posune na 3 a proces dokončí svou aktivitu: `t = 3: konec aktivity_1`. Čtvrté zaslání zprávy `peek()` ukazuje nejbližší naplánovanou událost, což je tentorkát dokončení procesu `procesni_funkce_1` v čase 3. V SimPy se totiž i ukončení procesu zapisuje jako událost do plánovače. Proto po dokončení těla procesní funkce ještě jedno `step()` finalizuje proces (nastaví jeho stav na finished), ale nevypíše žádný text z těla funkce. Pátá zpráva `step()` tedy nic nevypíše a pouze ukončí první proces. Tímto způsobem bychom pokračovali až do zaslání sedmé zprávy `env.step()`, která by už vyhodila výjimku `EmptySchedule`, protože v plánu simulace nejsou další události, což signalizuje konec simulace.

Výstup by tedy vypadal následovně.

```
0
t = 0: start aktivity_1
0
t = 0: start aktivity_2
3
t = 3: konec aktivity_1
3
5
t = 5: konec aktivity_2
5
```

5.4 Události

SimPy vytváří události prostřednictvím modulu `simpy.events`. Základní třídou pro všechny události je `simpy.events.Event`, která definuje společné vlastnosti a metody, jenž dědí všechny specializované typy událostí. Třídu `Event` lze použít přímo u uživatelsky definovaných událostí, které budou představeny v následující části práce. SimPy dále nabízí několik specializovaných podtříd pro různé účely.

5.4.1 Specializované události

Z předešlých kapitol již víme, že `Timeout` je speciální událost, která je dokončena po uplynutí zadaného času. Používá se pro simulaci aktivity, která trvá určitou dobu. `Initialize` je událost, která signalizuje inicializaci komponenty a využívá obvykle interně v SimPy. Komponentou je například objekt `Resource`, který představuje přístup k omezenému zdroji. O nich se více dozvíme později. `Process` je událost reprezentující běh generátorové funkce. Dokončení této události znamená, že skončila nějaká aktivita. `Condition` je abstraktní třída pro složitější podmínky, které závisí na jiných událostech. `AllOf` je podmíněná událost, která je dokončena až tehdy, když všechny zadané události jsou dokončeny. `AnyOf` je podmíněná událost, která je dokončena, jakmile je dokončena alespoň jedna ze zadaných událostí.

5.4.2 Uživatelsky definované události

Třída `Event` kromě vestavěných událostí umožňuje vytvořit vlastní události mající podobné vlastnosti jako vestavěné. Vlastní událost vznikne vytvořením nové instance třídy `Event`, které předáme prostředí, v němž se simulace odehrává. Vytvoření uživatelské události může vypadat například takto my_event = simpy.Event(env),

kde `env` je prostředí. Uživatelsky definovaná událost je užitečná, když potřebujeme simulovat změnu stavu, na kterou SimPy nemá speciální událost. Typickými příklady jsou čekání na externí signál, specifická reakce systému nebo vlastní podmínky řízení toku simulace.

U uživatelsky definovaných událostí lze explicitně vyvolat jejich nastání. Jakmile událost nastane, může být vyhodnocena buď jako úspěšná, nebo jako neúspěšná. Úspěšné nastání události se vyvolá voláním metody `Event.succeed()`, která může volitelně nést návratovou hodnotu reprezentující výsledek dané aktivity. Tato hodnota je následně předána procesům, které na danou událost čekají.

V případě vzniku chybového stavu během zpracování procesní funkce lze událost vyhodnotit jako neúspěšnou pomocí metody `Event.fail()`. Tato metoda přijímá instanci výjimky, která je následně vyvolána v procesech čekajících na danou událost, čímž je umožněno standardní ošetření chybových stavů.

SimPy dále poskytuje obecný mechanismus pro přeposílání výsledků mezi událostmi prostřednictvím metody `Event.trigger()`, která převezme výsledek i stav jiné události. Tento přístup je vhodný zejména v případech, kdy je požadováno propojení událostí bez nutnosti explicitně rozlišovat jejich úspěšné či neúspěšné vyhodnocení.

5.4.3 Stavy událostí

Každá událost může být v jednom ze tří stavů: může nastat `(not triggered)`, má nastat `(triggered)`, nebo nastala `(processed)`. Každá událost projde těmito stavy přesně jednou a vždy v uvedeném pořadí. Události jsou úzce spjaty s časem simulace, který řídí jejich přechod mezi stavy. Po vytvoření je událost ve stavu `(not triggered)`, to znamená, že událost ještě není aktivována, ale jde pouze o objekt v paměti. Když je událost naplánována na určitý čas, přejde do stavu `triggered` a atribut `Event.triggered` je nastaven na `True`. Událost v tomto stavu čeká na svůj časový okamžik, kdy bude zpracována. Dokud událost není zpracována, lze k ní přidávat callbacky. Událost přejde do stavu `processed` jakmile byla dokončena a všechny callbacky byly spuštěny.

5.4.4 Callbacky

`Callbacky` jsou funkce, které se spustí ve chvíli, kdy je událost dokončena. Každá událost v SimPy si interně udržuje seznam callbacků. Jakmile je událost dokončena, SimPy tento seznam projde a všechny uložené callbacky vykoná. Callbacky lze k události připojit také explicitně přidáním callback funkce do seznamu `callbacks` voláním metody `append()`. Procesní funkce, která čeká na nějakou událost (např. `yield env.timeout(...)`), je automaticky zaregistrována jako jeden z callbacků

této události. Díky tomu se po jejím dokončení procesní funkce znovu aktivuje a pokračuje přesně na místě, kde byla pozastavena. Tento mechanismus umožňuje přirozené střídání aktivit a souběh více entit, aniž by bylo nutné jejich běh řídit manuálně. Callback tedy reaguje na dokončení události. Aktivity, které čekají na událost, jsou jedním typem callbacků. Můžeme si ale callbacky definovat i explicitně, například pokud chceme, aby se po dokončení timeout události spustila nějaká další funkce. Jelikož jsou callbacky nízkoúrovňový mechanismus, který SimPy používá interně, umožňují reagovat na událost i bez definování procesních funkcí.

5.4.5 Příklad uživatelsky definované události s callbacky

Uživatelsky definovanou událost a použití callbacku demonstrujeme, když se vrátíme k příkladu s roboty. Tentorkát ho však upravíme tak, aby roboti šli opačným směrem proti sobě. Jakmile se jejich pozice shodují, je vyvolána uživatelsky definovaná událost signalizující jejich setkání. Na tuto událost je navázán callback, který je spuštěn v okamžiku dokončení události. Tento callback bude mít pouze informativní účel a vypíše informaci o nastání události setkání. Událost setkání pojmenujeme `meeting_event` a callback `meeting_callback`.

```
position = {"Robot A": 0, "Robot B": 5}

def robot(env, name, event):
    while True:

        if position["Robot A"] == position["Robot B"]:
            event.succeed()
            break

        else:
            print(f"{env.now}: {name} je na pozici {position[name]} a začal se pohy

        if name == "Robot A":
            position[name] += 1
        else:
            position[name] -= 1

        yield env.timeout(random.randint(1, 3))
        print(f"{env.now}: {name} se zastavil a rozhlíží se")
        yield env.timeout(1)

def meeting_callback(event):
    print(f"Roboti se potkali na pozici ", position)
```

```
env = simpy.Environment()
meeting_event = env.event()
meeting_event.callbacks.append(meeting_callback)

env.process(robot(env, "Robot A", meeting_event))
env.process(robot(env, "Robot B", meeting_event))

env.run(until=meeting_event)
```

První změnou oproti předchozímu příkladu je, že musíme v globální proměnné `position` uchovávat aktuální pozici robotů, aby bylo možné identifikovat okamžik, kdy se roboti setkají. Dále musíme vytvořit událost setkání, tedy uživatelsky definovanou událost `meeting_event = env.event()`, již dříve zmíněný informativní callback `meeting_callback` a následně ho přidat do seznamu callbacků události pomocí `meeting_event.callbacks.append(meeting_callback)`. Při volání `process` je tentokrát robotům předáno prostředí, název robota, a událost `meeting_event`, simulaci v tomto případě spustíme s podmínkou ukončení při nastání této události.

V procesní funkci `Robot` je klíčový rozdíl v kontrole pozice obou robotů při každém průchodu cyklem `while`. V okamžiku, kdy se budou roboti nacházet na shodné pozici je vyvoláno úspěšné nastání uživatelsky definované události voláním metody `event.succeed()`, čím je simulace ukončena a je spuštěn callback. Pokud roboti nejsou na stejné pozici, začnou se pohybovat a rozhlížet podobně jako v předchozím příkladu, pouze s tím rozdílem, že u robota `robot_A` se pozice o jednu jednotku zvyšuje, a u robota `Robot_B` se pozice o jednu jednotku snižuje.

Výstup příkladu v tomto případě by vypadal následovně:

```
0: Robot A je na pozici 0 a začal se pohybovat
0: Robot B je na pozici 5 a začal se pohybovat
2: Robot A se zastavil a rozhlíží se
3: Robot B se zastavil a rozhlíží se
3: Robot A je na pozici 1 a začal se pohybovat
4: Robot B je na pozici 4 a začal se pohybovat
5: Robot A se zastavil a rozhlíží se
5: Robot B se zastavil a rozhlíží se
6: Robot A je na pozici 2 a začal se pohybovat
Roboti se potkali na pozici  {'Robot A': 3, 'Robot B': 3}
```

5.4.6 Procesy jsou také události

Procesy v SimPy vytvořené pomocí `env.process()` mají příjemnou vlastnost, dají se také považovat za události. To znamená, že proces může čekat na dokončení jiného

procesu pomocí příkazu yield. Jakmile proces, na který se čeká skončí, čekající proces je obnoven a je mu předána návratová hodnota ukončeného procesu.

Jak může proces čekat na dokončení jiného procesu demostrujeme příkladem, v němž entita `teacher` čeká, než entita `student` vypočítá zadaný jednoduchý příklad. Po dokončení výpočtu entity `student` je návratová hodnota `result` předána entitě `teacher`, která ji následně vytiskne na standardní výstup.

```
def student(env, duration):
    yield env.timeout(duration)
    result = 2 + 3
    return result

def teacher(env):
    result = yield env.process(student(env, 3))
    print(result)

env = simpy.Environment()
env.run(env.process(teacher(env)))
```

5.5 Sdílené zdroje

Sdílené zdroje představují objekty, o které soutěží více procesů (entit) současně. Procesy musí čekat ve frontě, pokud je zdroj momentálně obsazen, a mohou ho využít až po jeho uvolnění. Tento princip se hodí pro modelování situací, kdy více procesů chce přistoupit k jednomu zdroji a jeho kapacita je omezená, například pět lidí čekajících na jeden záchod a nebo několik aut čekajících na jednu čerpací stanici. Simpy rozlišuje tři typy zdrojů.

Resources jsou zdroje, které mohou být současně využívány libovolným omezeným počtem procesů. U objektů `resource` je sledováno počet současných uživatelů. Příkladem může být omezený počet toalet.

Containers představují zdroje, ve kterých se pracuje s homogení, nerozlišitelnou hmotou. Procesy do něj mohou dávat nebo brát množství hmoty. U objektů `container` je sledováno množství nějaké hmoty. `Container` může být například sud a procesy z něj mohou brát litry piva.

Stores jsou zdroje, které uchovávají konkrétní objekty. Procesy mohou vložit nebo vybrat jednotlivý objekt, který má své specifické vlastnosti. U objektů `store` se sledují konkrétní objekty, jejich identita, atributy a pořadí, tedy co přesně je dostupné. Příkladem mohou různá být trička v merch stánku.

Všechny zdroje sdílejí stejný princip, zdroj je jakýsi kontejner s omezenou kapacitou. Procesy se mohou pokusit něco do zdroje vložit, případně do něj vstoupit, nebo něco získat, případně z něj odejít. Pokud je zdroj plný nebo prázdný, procesy musí čekat ve frontě. Každý sdílený zdroj typicky obsahuje kapacitu, frontu pro vkládání `put_queue`, frontu pro vybírání `get_queue`, a metody `put()` a `get()`. Metoda `put()` se používá, pokud chce proces něco vložit do zdroje, pokud je zdroj plný, proces musí čekat ve frontě `put_queue`, dokud se místo neuvolní. Metoda `get()` se používá, pokud proces chce něco ze zdroje získat, pokud je zdroj prázdný, proces musí čekat ve frontě `get_queue`, dokud se něco nevloží. Tyto metody nevykonávají akci okamžitě, ale vrací událost, která nastane ve chvíli, kdy je možné požadavek splnit. Proces se klíčovým slovem `yield` pozastaví a pokračuje po splnění požadavku.

5.5.1 Resources

Zdroje `Resources` mohou být využívány omezeným počtem procesů současně. Aby proces směl daný `resource` používat, musí si k němu vyžádat přístup, a to metodou `request()`. Po skončení proces musí zdroj `resource(res)` uvolnit metodou `release(res)`. Parametr `res` je zdroj, který ke kterému chceme přistoupit, nebo ho uvolnit. Volání `request()/release()` je tedy formou volání `put()/get()` u zdroje `resource`. Uvolnění proběhne vždy okamžitě. .

Simpy implementuje `Resources` ve třech variantách. První variantou je základní zdroj `Resource`, jeho parametry jsou prostředí a kapcita, ta musí být vždy kladné celé číslo a ve výchozím nastavení je kapacita rovna jedné, tedy `Resource(env, capacity=1)`. Metoda `Resource.request()` vytváří `request token`, který reprezentuje konkrétní žádost procesu o daný zdroj. Tento token je uložen ve zdroji po celou dobu jeho využívání a je následně předán metodě `release()`, která na jeho základě zdroj uvolní. Uvolnění `Resource` lze provést jednoduchým způsobem, kdy pomocí `resource.request(req)` zažádáme o zdoj `req` a poté čekáme, až bude volný. Následně proběhne nějaká aktivita, a nakonec zdroj uvolníme pomocí `resource.release(req)`.

```
req = resource.request()
yield req
yield env.timeout(1)
resource.release(req)
```

Aby se předešlo situacím, kdy by zdroj zůstal obsazený například v důsledku přerušení procesu nebo vzniku chyby, nabízí SimPy elegantnější řešení v podobě **context manageru**. Ten využívá klíčová slova `with` a `as` a zajišťuje automatické uvolnění zdroje po opuštění příslušného bloku kódu:

```
with resource.request() as req:
    yield req
    yield env.timeout(1)
```

V tomto případě je `request token` automaticky zaregistrován v rámci bloku `with`. Jakmile je blok opuštěn, SimPy zajistí uvolnění zdroje bez nutnosti explicitního volání metody `release()`. Tento mechanismus je funkčně obdobný práci se soubory v jazyce Python, kde konstrukce `with open(...)` zajišťuje automatické uzavření souboru. Z vnitřního pohledu lze chování `context manageru` zjednodušeně přirovnat k následující struktuře:

```
try:
    yield req
    yield env.timeout(1)
finally:
    resource.release(req)
```

Díky tomuto přístupu je zaručeno, že zdroj bude vždy korektně uvolněn, a to i v případě přerušení procesu (`Interrupt`) nebo vzniku výjimky.

Druhou variantou sdíleného zdroje `Resources` je `PriorityResource`. Tato třída je podtřídou základní třídy `Resource` a umožňuje procesům při každé žádosti o zdroj specifikovat prioritu. Požadavky s vyšší prioritou získají přístup ke zdroji dříve než požadavky s nižší prioritou. Priorita je vyjádřena celočíselnou hodnotou, přičemž menší hodnota znamená vyšší prioritu než vyšší hodnota. Například `PriorityResource` s prioritou 1 má vyšší prioritu než `PriorityResource` s prioritou 2. Ve všech ostatních ohledech se `PriorityResource` chová stejně jako běžný `Resource`.

Poslední variantou `Resources` je třída `PreemptiveResource`. Tato třída umožňuje odebrat zdroj procesu, který jej právě využívá. To je užitečné, pokud jsou nové požadavky natolik důležité, že pouhé přednostní zařazení do fronty nestačí a je nutné procesu, který využívá daný zdroj, tento zdroj odebrat. Tento mechanizmus se označuje jako `preemption`, tedy přerušení. Třída `PreemptiveResource` dědí z `PriorityResource` a rozšiřuje metodu `request()` o parametr `preempt`, který je ve výchozím stavu nastaven na True. Pokud je tento parametr nastaven na False (`resource.request(priority=x, preempt=False)`), proces se rozhodne nepřerušovat jiný proces, který zdroj právě využívá. Takový požadavek je nicméně stále zařazen do fronty podle své priority.

Implementace `PreemptiveResource` klade větší důraz na prioritu než na samotné přerušení. To znamená, že preemptivní požadavky nemohou přeskočit požadavek s vyšší prioritou, který již čeká ve frontě.