Producent a konzument

Klasickým návrhovým vzorem paralelního programování je producent-konzument, kde jsou procesy označeny buď jako producenti nebo konzumenti. Producenti jsou odpovědní za přidávání do sdílené datové struktury a konzumenti za odebírání z této struktury. V každém okamžiku může přistupovat pouze jedna strana, buď jediný producent nebo jediný konzument.

Zde uvažujeme příklad se sdílenou frontou, která je chráněna mutexem.

Code

Producer

```
def produce(i, p, d):
p += 1
time.sleep(d)
print(f"Producer {i} has produced {p}")

def store(i, p):
with m:
f.put((i, p))
print(f"Producer {i} has added its {p}-th product")

def producer(n, i):
p = 0
d = random.random() * 2
for j in range(n):
produce(i, p, d)
store(i, p)
time.sleep(random.random() * 2.5)
print(f"Producer {i} is exiting.")
```

Consumer

```
def wait(i):
with m:
while f.empty():
print(f"Consumer {i} is waiting")
c.wait()

def take(i):
with m:
ip, p = f.get()
print(f"Consumer {i} has taken product ({ip},{p})")

def consumer(n, i):
```

```
for j in range(n):
wait(i)
take(i)
time.sleep(random.random() * 2.5)
print(f"Consumer {i} is exiting.")
```

Problém, který se řeší v tomto příkladu, je synchronizace mezi producenty a konzumenty při práci s sdílenou datovou strukturou (ve formě fronty). Bez synchronizace by mohlo dojít k chybám, jako je ztráta dat nebo nesprávné chování programu.

Řešením tohoto problému je použití synchronizačních mechanismů, konkrétně mutexů a podmíněných proměnných. Mutex zajišťuje, že pouze jeden proces může najednou přistupovat ke sdílené datové struktuře, čímž se zabrání závodům a chybám způsobeným konkurenčním přístupem. Podmíněné proměnné umožňují procesům čekat, dokud nejsou splněny určité podmínky, což je užitečné v situacích, kdy chceme, aby konzument čekal na výrobek vytvořený producentem.

Konkrétně v tomto příkladě, kdy konzument čeká na výrobek v frontě, je využita podmíněná proměnná (Condition). Producent signalizuje podmíněnou proměnnou, když přidává výrobek do fronty, což probouzí konzumenta, který čeká. Tímto způsobem konzument neztrácí zdroje a časovou efektivitu tím, že neustále kontroloval prázdnost fronty, ale aktivně čeká na výrobu nového výrobku.

Večeřící filozofové

Problém obědvajících (též večeřících) filozofů je myšlenkový experiment, který se používá k ověření správné činnosti správy procesů. Formuloval jej Edsger Dijkstra v roce 1965. Každý operační systém má správu procesů, což je vlastně algoritmus, který řídí vznik, zánik a plánování procesů (běžících programů) v operačním systému.

Ilustrace problému obědvajících filozofů

Představme si kulatý stůl, na kterém je po obvodu položených 5 talířů. Mezi každými dvěma talíři je jedna čínská hůlka, celkem je jich tedy také 5. Dále si představme, že za tímto stolem sedí pět filozofů, každý za svým talířem. Filozof představuje nějaký proces a může provádět dvě činnosti – buď obědvat (oběd), nebo filozofovat. Aby mohl obědvat, musí si vzít dvě čínské hůlky. Pokud chce filozofovat, nedrží ani jednu hůlku.

Když spolu filozofové (procesy) nekomunikují nebo komunikují nesprávně, může se každý z nich rozhodnout, že vezme například levou hůlku. Teď chce každý z nich vzít pravou hůlku, ale ta je obsazena, takže filozof nemůže ani obědvat, ani filozofovat. Takový stav se nazývá uváznutí (anglicky deadlock). Musí tedy buď počkat, než se hůlka uvolní, nebo hůlku položit a zkusit to později znovu.

Jiným kritickým stavem je vyhladovění (anglicky starvation, česky stárnutí zdrojů), které nastává, když se filozof nedostane po určité době k jídlu (resp. proces ke zdrojům). Nastává například při velmi krátkých intervalech jezení a filozofování.

Vyhladovění by mohlo také nastat nezávisle na deadlocku, pokud je filozof neschopný získat obě hůlky během stejného časového úseku. Například můžeme uvažovat pravidlo, že filozof pustí jednu hůlku z ruky za 5 minut od okamžiku, kdy si ji vezme, a čeká dalších 5 minut na to, aby udělal další pokus najíst se. Tento režim eliminuje možnost deadlocku (systém se vždy může dostat do jiného stavu), ale stále je problémem livelock. Když totiž všichni filozofové položí najednou hůlky a pak si je za 5 minut opět vezmou, situace se opakuje, znovu mají po jedné hůlce všichni filozofové.

Nedostatek volných hůlek je analogií k zamykání sdílených prostředků v reálném počítačovém programování, situace známá, jako souběžnost. Zamykání prostředku je běžná technika, jak zajistit konzistenci daného zdroje v čase. Když zdroj, se kterým program pracuje, už je uzamčen jiným programem, program čeká, dokud není odemčen. Pokud některé programy vyžadují zamykání zdrojů, může nastat deadlock. Závisí tudíž na okolnostech. Například, jeden program potřebuje dva soubory ke zpracování. Když každý ze dvou takových programů uzamkne jeden z těchto souborů, oba budou čekat na to, aby ten druhý uvolnil zdroj, který má k dispozici, což se nikdy nestane.

Obecně je problém obědvajících filozofů běžným abstraktním problémem používaným pro vysvětlení různých otázek, které se objevují v záležitosti vzájemné výlučnosti jako hlavní myšlenky daného problému.

Řešení

Řešení za pomocí přidání číšníka

Jednoduché řešení lze dosáhnout zavedením číšníka u stolu. Číšník určí, kdo si hůlky vezme, co v podstatě vyřeší problém rozhodování. Protože si uvědomuje, které hůlky jsou použity, je schopen rozhodnout a zabránit tak deadlocku. Protože na stole zůstala v případě 5 filozofů ještě jedna hůlka, je zřejmé, že následovat v jezení bude ten filozof, který se stal jejím dočasným majitelem. Tomu pak číšník přisoudil hůlku, kterou zrovna obsluhuje vedle sedící filozof.

Konkrétní řešení: Označíme si filozofy podle hodinových ručiček od A po E. Pokud A a C jedí, tak jsou použité 4 hůlky. B nemá k dispozici žádnou hůlku, pokud D a E mají mezi sebou jednu nevyužitou hůlku. Řekněme, že D chce jíst. Pokud by si filozof tuto hůlku vzal, může nastat deadlock. V našem případě však o tom rozhoduje číšník, který ví, že je třeba počkat na uvolnění další hůlky, aby se později mohl alespoň jeden filozof naobědvat.

Řešení s hierarchií zdrojů

Další jednoduché řešení dostaneme vyhrazením částečného pořadí, nebo hierarchie pro zdroje (v tomto případě hůlky), a zřízením konvence, že všechny zdroje budou dosahované v určitém pořadí a v opačném pořadí uvolněné. V našem případě budou zdroje (hůlky) očíslované od 1 po 5 v nějakém pořadí a každý filozof si vždy vezme nejdříve hůlku s menším číslem a až potom hůlku s větším číslem. Pak vždy položí nejdříve hůlku s vyšším číslem, následně hůlku s menším číslem. Pokud tedy 5 filozofů simultánní zvedne hůlku s menším číslem, tak zůstane na stole hůlka s největším číslem, takže 5. filozof bude bez hůlky. Navíc pouze jeden z filozofů bude mít přístup k oběma hůlkám. Když dojí, pustí obě hůlky, přičemž tu hůlku s nižším číslem pustí dříve, což umožní, aby se najedl filozof sedící vedle něj.

Toto řešení i navzdory vyhýbání se deadlokům není příliš praktické, speciálně v případě, pokud neznáme předem používanou množinu zdrojů. Například pokud program drží zdroje 3 a 5 a potřebuje ještě zdroj 2, musí vypustit zdroj 5, pak 3, aby mohl požádat o 2 a opět požádat o zdroje 3 a 5 v tomto pořadí. Právě proto je tento způsob velmi neefektivní.

Řešení Chandel-Misra

V roce 1984 K. Mani Chandel a J. Misra navrhli jiné řešení problému obědvajících filozofů, aby povolili libovolnému počtu programů (číslovaných P1, ..., Pn) soutěžit o libovolný počet zdrojů (číslovaných R1, ..., Rm). Na rozdíl od Dijkstrova řešení tato označení mohou být libovolná. Uvědomme si, že toto není skutečný problém obědvajících filozofů, protože vyžaduje jejich vzájemnou komunikaci.

1. Pro každý pár filozofů válčících o zdroj vytvoří hůlku a dají ji filozofovi s nižším ID. Každá hůlka může být buď špinavá, nebo čistá. Na začátku je každá hůlka špinavá.

2. V případě, že chce filozof použít množinu zdrojů, musí dostat hůlky od svých soupeřících sousedů. Pro všechny takové hůlky zašle žádanku.

3. Filozof, který obdrží požadavek, si hůlku nechá, pokud je čistá, v opačném případě ji přenechá žádajícímu filozofovi. Předtím, než tuto hůlku zašle filozofovi, ji nejdříve očistí.

4. Potom, co filozof dojí, všechny jeho hůlky jsou špinavé. Pokud nějaký filozof dříve požádal o hůlku, filozof, který ji momentálně vlastní, ji očistí a pošle.

5. Toto řešení umožňuje velké množství paralelních programů a řeší libovolně velký problém s předpokladem, že každé vlákno potřebuje právě jeden zdroj v čase.