Wstęp

Biblioteka obejmuje 3 niskopoziomowe elementy:

- rozproszony mutex
- rozproszona zmienna warunkowa
- rozproszone zmienne współdzielone
  - lista (list)
  - słownik (dictionary)

Dostarcza również klasy bazowej (`MonitorBase`) umożliwiającą implementację monitora. Monitor wykorzystuje elementy wymienione powyżej w sposób możliwie transparentny dla programisty.

Rozproszony mutex (moduł `mutex`)

Rozproszony mutex wykorzystuje rozproszony algorytm wzajemnego wykluczania Lamporta. Przy każdym wysłaniu oraz odebraniu wiadomości aktualizowany jest zegar lamporta (globalny w ramach jednego procesu). Znacznik czasowy jest wysyłany razem z wiadomością.

Każdy mutex utrzymuje kolejkę procesów oczekujących na wejście do sekcji krytycznej oraz tablicę odpowiedzi (wiadomości `acquire_reply`) na prośbę (wiadomość `acquire_request`) od poszczególnych procesów.

Mutex udostępnia 2 metody:

Metoda `acquire()`

- wyślij wiadomość `acquire_request` do wszystkich pozostałych procesów
- dodaj bieżący proces do kolejki (kolejność wynika ze znacznika czasowego)
- czekaj aż zostaną spełnione 2 warunki:
  - otrzymano odpowiedzi (nowsze od prośby) od wszystkich pozostałych procesów
  - bieżący proces jest na początku kolejki

Metoda `release()`

- wyślij wiadomość `release` do wszystkich pozostałych procesów
- usuń bieżący proces z kolejki

W przypadku wielokrotnych wywołań metody `acquire()`, zwiększany jest licznik `acquire_count`. Aby zwolnić mutex metoda `release()` musi być wywołana taką samą liczbę razy aby zredukować licznik do zera.

Mutex reaguje w następujący sposób na odebrane wiadomości:

Odbiór wiadomości `acquire_request`

- dodaj proces nadawcy do kolejki
- wyślij odpowiedź `acquire_reply`

Odbiór wiadomości `acquire_reply`

- zaznacz licznik czasowy odpowiedzi w tablicy

Odbiór wiadomości `release`

- usuń proces nadawcy z kolejki

Zmienna warunkowa (moduł `condition`)

Zmienna warunkowa utrzymuje kolejkę procesów czekających na spełnienie warunku (metoda `wait()`). Tak jak w przypadku kolejki mutexa kolejność wynika ze znaczników czasowych. Każda zmienna warunkowa jest skojarzona z mutexem i może być używana tylko wewnątrz sekcji krytycznej.

Udostępnia 2 metody:

Metoda `wait()`

- wyślij wiadomość `wait` do wszystkich pozostałych procesów
- zwolnij mutex (`mutex.release()`)
- czekaj na odebranie wiadomości `signal`
- zajmij mutex (`mutex.acquire()`)

Metoda `signal()`

- jeśli kolejka jest pusta: zakończ metodę
- wyślij wiadomość `signal` do pierwszego procesu w kolejce
- usuń pierwszy proces z kolejki

Zmienna warunkowa reaguje w następujący sposób na odebranie wiadomości:

Odbiór wiadomości `wait`

- dodaj proces nadawcy do kolejki

Odbiór wiadomości `signal`

- powiadom pozostałe procesy o możliwości usunięcia procesu z kolejki (wiadomość `pop`)
- kontynuuj wykonywanie metody `wait()`

Odbiór wiadomości `pop`

- usuń pierwszy proces z kolejki (jest to proces nadawcy)

Zmienne współdzielone (moduł `shared_variables`)

Dostępne są 2 rodzaje zmiennych współdzielonych: listy oraz słowniki, odpowiadające standardowym strukturom danych w języku Python. Zmienna współdzielona utrzymuje historię modyfikacji i umożliwia synchronizację ze zdalną kopią poprzez rozgłoszenie zmian. Zmienna współdzielona powinna być używana wraz z mutexem aby zapewnić, że tylko jeden proces modyfikuje zmienną w danym momencie. Elementami zmiennych mogą być dowolne obiekty podlegające serializacji za pomocą modułu pickle.

Zmianna współdzielona posiada metodę `apply_changes(changes)`, która przyjmuje listę zmian i wykonuje kolejno operacje na zmiennej w celu synchronizacji stanu. Jej implementacja zależy od rodzaju zmiennej.

Posiada również metodę `apply_pending_changes()`, która aplikuje zbuforowane zmiany (otrzymane od innych procesów) w kolejności wynikającej z ich znaczników czasowych.

Do rozgłoszenia zmian służy metoda `sync()`:

Metoda `sync()`

- wyślij listę zmian do wszystkich pozostałych procesów (wiadomość `sync`)
- wyczyść listę zmian

Zmienna współdzielona reaguje na odbiór wiadomości `sync`, która zawiera w sobie listę zmian.

Odbiór wiadomości `sync`

- dodaj otrzymane zmiany do kolejki `pending_changes` (kolejność wynika ze znacznika czasowego)

Metodę `sync()` należy wywołać będąc jeszcze w sekcji krytycznej, ponieważ inny proces, oczekujący na zajęcie mutexa musi otrzymać zmiany przed wejściem do sekcji krytycznej.

Metodę `apply_pending_changes()` należy wywołać dopiero po otrzymaniu wszystkich wiadomości `sync` od procesów, które poprzednio modyfikowały zmienną. Pewność co do tego można mieć po zajęciu mutexa, ponieważ oznacza to, że otrzymano wszystkie wiadomości `release` od procesów które mogły modyfikować zmienną, a więc również wszystkie wiadomości `sync` (proces wysyła `sync` przed `release`).

Monitor (moduł `monitor_meta`)

Dostępna jest klasa bazowa `MonitorBase`. Tworząc klasę dziedziczącą z tej klasy (np. `Monitor`) programista może w prosty sposób zaimplementować monitor. Każda instancja klasy `Monitor` zostaje powiązana z mutexem w momencie tworzenia.

Obsługa mutexa jest dołącza do każdej zdefiniowanej przez programistę metody klasy `Monitor` z wyjątkiem metody `__init__` (zakłada się też, że programista nie będzie modyfikował metody `__new__`).

Zmienne warunkowe należy utworzyć w metodzie `__init__`, korzystając z pomocniczej metody `condition()`. Metoda ta tworzy odpowiednio opakowaną zmienną warunkową. Opakowanie to jest niezbędne dla zapewnienia obsługi zmiennych współdzielonych przed i po wykonaniu metody `wait()` (która wewnętrznie zwalnia oraz zajmuje mutex).

Zmienne współdzielone należy również tworzyć w metodzie `__init__`, korzystając z metody `shared(data)`. Metoda ta tworzy zmienną współdzieloną odpowiedniego typu i wiąże ją z monitorem. Obsługa wszystkich powiązanych zmiennych współdzielonych jest automatycznie dołączana do metod monitora.

Wątek obsługi zdarzeń (moduł `util`)

Komunikacja związana z monitorem odbywa się w osobnym wątku. Po inicjalizacji monitorów należy ten wątek uruchomić. Najprościej jest wykorzystać menedżer kontekstu `event_loop_thread()`:

```
from monitor.monitor_meta import MonitorBase
from monitor.util import event_loop_thread

class Monitor(MonitorBase):
    pass

m = Monitor()

with event_loop_thread():
    # application code
```

`event_loop_thread()` zadba o uruchomienie wątku przed rozpoczęciem kodu aplikacji i jego zatrzymanie po zakończeniu bloku kodu.