Екзаменаційний білет №4

1. Два основних способи реалізації потоків.

Існує два основних підходи до реалізації багатопоточності в операційних системах:

1. Потоки в користувацькому просторі: У цьому підході керування потоками здійснюється бібліотекою потоків користувацького рівня. Ядро не знає про існування цих потоків і керує ними так, ніби кожен потік є однопотоковим процесом. Перевагою потоків користувацького рівня є те, що їх швидко створювати та керувати ними, оскільки всі операції, такі як створення, планування та синхронізація потоків, відбуваються у просторі користувача без необхідності втручання ядра. Однак, суттєвим недоліком є те, що якщо один потік користувацького рівня виконує операцію блокування, весь процес може бути заблоковано, оскільки ядро бачить лише однопотоковий процес.
2. Потоки в просторі ядра: У цьому підході все керування потоками здійснюється ядром. Ядро повністю знає про існування потоків і відповідає за управління ними. Це означає, що створення, планування та синхронізація потоків відбувається повільніше, оскільки вони пов'язані з системними викликами ядра. Однак головною перевагою потоків на рівні ядра є те, що операційна система може планувати потоки одного процесу на різних процесорах у багатопроцесорній системі. Крім того, якщо один потік у процесі блокується, інші потоки у тому ж процесі можуть продовжувати працювати, оскільки ядро знає про окремі потоки.

Ці два підходи являють собою фундаментальний компроміс між продуктивністю та функціональністю в моделях багатопотоковості. Потоки користувацького рівня мають переваги у продуктивності, але їм бракує деяких функціональних можливостей потоків рівня ядра, наприклад, справжнього паралельного виконання на декількох процесорах.

1. Проаналізувати чи можна в реалізації алгоритму заміщення LRU за допомогою n×n-матриці змінити порядок виконання операцій скідання стовпців та встановлення рядка.

Алгоритм Least Recently Used (LRU) виглядає наступним чином:

Початок обходу сторінок: передбачає ітераційний перегляд сторінок одну за одною та прийняття рішення, чи зберігати їх у наборі (кеші/пам'яті), чи ні.

Якщо набір містить менше сторінок, ніж може вмістити, сторінки додаються до набору доти, доки він не досягне своєї місткості або доки не будуть оброблені всі запити на сторінки. Під час цього процесу ведеться індексна карта для відстеження останніх сторінок, до яких було здійснено доступ.

Якщо набір заповнений, якщо поточна сторінка вже є в наборі, ніяких дій не виконується. В іншому випадку, найменш використовувана сторінка (визначена за допомогою індексного масиву) замінюється поточною сторінкою.

Алгоритм повертає кількість помилок сторінок, що виникли під час роботи.

Припустимо, що ми працюємо з реалізацією алгоритму, що використовує бітову матрицю. Тоді множенням на довільну матрицю nxn, наприклад матрицю перестановок, ми можемо змінити вхідну матрицю сторінок. Таким чином, ми будемо звертатися до сторінок вже у іншому порядку і виконувати подальші кроки що може кардинально змінити результат алгоритму.

1. За допомогою condition-змінних реалізувати систему накопичення потоків, при проходженні якої потік призупиняється, а по досягненні порогового значення кількості потоків, призупинених системою, прихід нового потока призводитиме до випуску одного з них. Хто припускається має визначатися політикою планувальника.

Я наведу приклад на мові Python, використовуючи потоки та змінні умови. Бібліотека потоків Python надає високорівневу, легку для розуміння абстракцію для такого роду завдань.

import threading

class ThreadAccumulator:

def \_\_init\_\_(self, threshold):

self.threshold = threshold

self.current\_count = 0

self.condition = threading.Condition()

def enter(self):

with self.condition:

self.current\_count += 1

print(f"Thread entered, count: {self.current\_count}")

if self.current\_count >= self.threshold:

self.release\_one()

while self.current\_count >= self.threshold:

self.condition.wait()

print("Thread allowed to proceed.")

def release\_one(self):

self.current\_count -= 1

print(f"Releasing one thread, count: {self.current\_count}")

self.condition.notify()

# example

accumulator = ThreadAccumulator(threshold=5)

# Function for threads to execute

def thread\_task():

print("Thread starting")

accumulator.enter()

print("Thread finishing")

# Create and start threads

for \_ in range(10):

threading.Thread(target=thread\_task).start()

У цьому прикладі

Клас ThreadAccumulator інкапсулює логіку для накопичення потоків і звільнення їх на основі порогового значення.

Метод enter - це місце, де потоки призупиняються, доки не буде виконано умову.

При досягненні порогового значення потоки викликають wait у змінній умови, призупиняючи свою роботу.

Коли входить новий потік і поріг перевищено, один потік, що очікує, звільняється.

Метод release\_one зменшує лічильник і повідомляє один потік, що очікує, про необхідність продовження роботи.

Потоки виконують функцію thread\_task, яка представляє роботу потоку. У цьому випадку він просто входить і виходить з ThreadAccumulator.