30.01.2021 Politechnika Wrocławska

Sprawozdanie

do zadania ***Programy symulujące***
z przedmiotu

Systemy operacyjne

Politechnika Wrocławska

Yevchuk Mykhailo,
Cyberbezpieczeństwo, Wydział Elektroniki, PWr

W ramach tego zadania zrealizowałem dwa algorytmy zastępowania stron oraz porównałem ich.

Algorytmy zastępowania stron

Pamięć wirtualna – mechanizm zarządzania pamięcią komputera zapewniający procesowi wrażenie pracy w jednym, dużym, ciągłym obszarze pamięci operacyjnej podczas, gdy fizycznie może być ona pofragmentowana, nieciągła i częściowo przechowywana na urządzeniach pamięci masowej.To mechanizm systemu operacyjnego, który umożliwia wykonywanie procesów, mimo iż nie są one w całości przechowywane w pamięci. Pamięć wirtualna oznacza możliwość posługiwania się znacznie większą pamięcią niż fizyczna pamięć RAM.

Stronicowanie na żądanie - to najczęstszy sposób implementacji pamięci wirtualnej. W skrócie można go określić jako sprowadzanie strony do pamięci operacyjnej tylko wtedy, gdy jest ona potrzebna. Nazywa się to także *procedurą leniwej wymiany*. Takie rozwiązanie zmniejsza liczbę wykonywanych operacji wejścia/wyjścia i zapotrzebowanie na pamięć operacyjną, ponieważ nie sprowadza się niepotrzebnych stron (pojedyncze wywołanie dużego wielofunkcyjnego programu może wymagać sprowadzenia jedynie niewielkiej części jego kodu). Dzięki temu zmniejsza się czas reakcji systemu. Można też obsłużyć większą liczbę użytkowników.

Gdy proces odwołuje się do pamięci, mogą zajść trzy sytuacje:

- odwołanie jest niepoprawne,
- odwołanie jest poprawne i strona jest w pamięci, oraz
- odwołanie jest poprawne, ale strony nie ma w pamięci.

W pierwszym przypadku zlecenie jest odrzucane, w drugim jest po prostu obsługiwane, a w trzecim musi dodatkowo obejmować sprowadzanie żądanej strony z dysku do pamięci.
Gdy wystąpi brak strony a nie będzie już wolnych ramek, trzeba wybrać jedną z ramek i zastąpić obecną w niej stronę żądaną stroną. Dzięki zastępowaniu stron pamięć logiczna może być większa niż pamięć fizyczna.

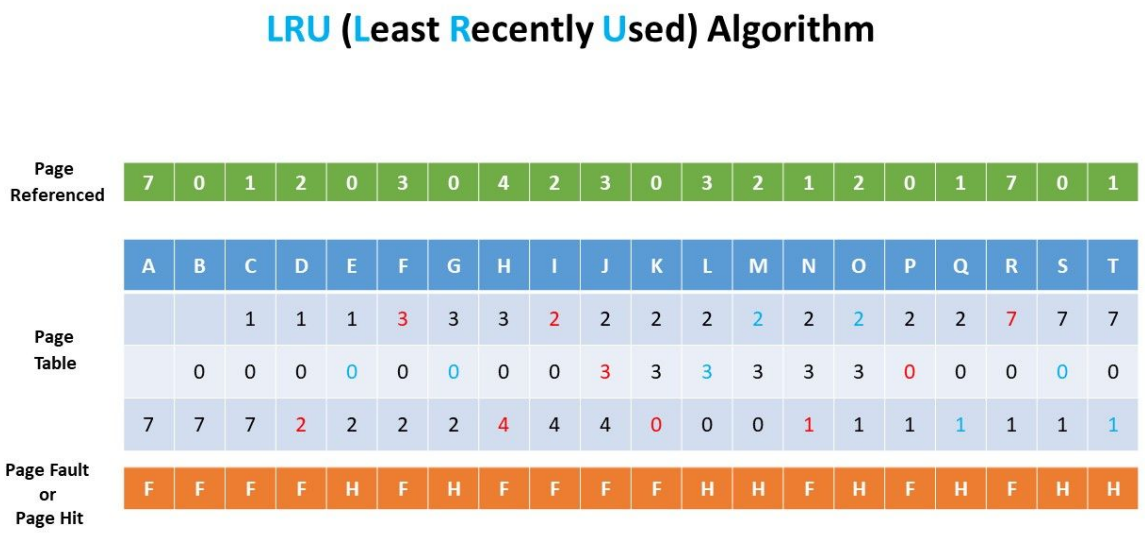
Zastępowanie strony obejmuje następujące czynności:

1. Znalezienie położenia żądanej strony na dysku.
2. Znalezienie strony-ofiary, która ma być usunięta z pamięci.
3. Zapisanie ofiary na dysk -- o ile na dysku nie ma jej wiernej kopii.
4. Oznaczenie w tablicy stron odwołania do ofiary jako niepoprawnego.
5. Wczytanie żądanej strony do zwolnionej ramki.
6. Oznaczenie w tablicy stron odwołania żądanej strony jako poprawnego.

Wybieraniem ofiary zajmują się *algorytmy zastępowania stron*. Jest to jedyne miejsce, gdzie wybór odpowiedniego algorytmu w systemie operacyjnym może wpłynąć na efektywność pamięci wirtualnej.

LRU

LRU (Least-Recently-Used) - czyli "najdawniej używany", jest to algorytm zastępowania stron, w którym ofiarą staje się strona, która nie była używana przez najdłuższy okres.



Implementacja.

Implementację danego algorytmu dokonałem w języku Python.Procesy przedstawiłem jako liczby typu integer, więc realizacja algorytmu jest prosta i jej sposób jest bardzo podobny do rysunku wyżej.

Funkcje `createRandomlyProcessesList(size:int)` oraz `createManuallyProcessesList()` Są to proste funkcje pozwalające na stworzenie list wybranej długości wypełnionych wybraną ilością procesów.Możemy wypełnić listy awtomatycznie lub przez

własnoręczne podawanie danych wybierając odpowiednią funkcję.

```
def createRandomlyProcessesList(size:int, numberOfProcesses:int):
    """ Returns a list with set length (size argument) of randomly generated processes from 1
    to numberOfProcesses argument """
    return [random.randint(1, numberOfProcesses) for i in range(size)]

def createManuallyProcessesList():
    """ Returns a list of processes entered manually by user"""
    numberOfProcesses = int(input("How many processes you want to create: "))
    processesList = []
    for _ in range(numberOfProcesses):
        process = int(input("Enter a process: "))
        processesList.append(process)
    return processesList
```

Funkcja getNumberOfPageFaultsOfProcesses(processesList:list, emptyPageSlots:int):
Ta właśnie funkcja odpowiada za realizację logiki naszego algorytmu.Tworzymy listę pageSlots = [], która będzie reprezentowała ramki pamięci.W pętli *for* przerabiamy każdy osobny proces.Mamy dwa przypadki odnośnie obecności procesu w pamięci cache: może tam być lub jego tam nie ma. Jeżeli nie ma procesu w żadnej z ramek ale mamy ramkę wolną (jeżeli długość odpowiadającej za to listy jest mniejsza niż parameter emptyPageSlots) to możemy dodać ten proces do naszej pamięci cache. Jak nie ma procesu oraz nie ma wolnych ramek pamięci to usuwamy proces najdawniej używany (wychodząc z logiki naszego programu, taki proces zawsze znajduje się na pierwszej pozycji listy czyli ma indeks 0) a potem dodajemy nasz nowy proces.W przypadku jak proces do którego chcemy się odwołać jest w jednej z ramek pamięci - to nie musimy nic dodawać lub usuwać, ale wychodząc z metody implementacji tego algorytmu zakładamy że w liście reprezentującej ramki pamięci najdawniej używany proces jest na początku listy a ostatnio używany proces jest na końcu.Więc usuwamy proces z listy i dodajemy ten sam proces na koniec listy, tym samym ją aktualizujemy.Funkcja zwraca ilość brakujących stron.

```python
def getNumberOfPageFaultsOfProcesses(processesList:list, emptyPageSlots:int):
    """ Returns number of Page faults for given list of processes and number of empty page
    slots """

    #logic of this function is to append incoming processes to pageSlots list by the time it
    will be full (emptyPageSlots variable)
    #when pageSlots will be full, in case if we will get element that is not in this list, we
    will delete first element from it, becouse this element will be automatically last
    recently used
    #and then append the new one
    #in case if element will be it list we will delete it, and immediately append it to list,
    thus we will update out list pf cache

    pageFault = 0
    pageSlots = []
    for process in processesList:
        if process not in pageSlots:
            if len(pageSlots) < emptyPageSlots:
                pageSlots.append(process)
            else:
                del pageSlots[0]
                pageSlots.append(process)

            pageFault += 1
        else:
            pageSlots.remove(process)
            pageSlots.append(process)

    return pageFault
```

Funkcje do zapisu danych do pliku oraz odczu danych z pliku

Także realizowałem osobne proste funkcje dla zapisywania listy procesów do pliku CSV oraz odczytu z takiego pliku.Możemy zapisać listę procesów do pliku po czym, podając jego nazwę możemy te wartości odczytać.Funkcja odczytująca umieści dane na liście i zwróci ją.

```python
def writeProcessesDataToCSVfile(processesList:list, fileName:str="processesDataLRU.csv"):
    """ Writes down all the information about single process to a CSV file given as argument,
    if file wasn't given as argument -
    creates new CSV file named 'processesDataLRU.scv' and writes down information to it """

    CSVcolumns = ["Process ID"]
    processesList = [{CSVcolumns[0]: i} for i in processesList]

    try:
        with open(fileName, "w+") as f:
            writer = csv.DictWriter(f, fieldnames=CSVcolumns)
            writer.writeheader()
            for process in processesList:
                writer.writerow(process)

        print("Successfully written to", fileName)
    except IOError:
        print("Input/Output error!")

def readProcessesDataFromCSVfile(fileName:str):
    """ Reads information about processes from CSV file. Returns a list where every single
    process is a dictionary. """

    try:
        with open(fileName, "r") as f:
            processesList = [{key: value for key, value in row.items()}
            for row in csv.DictReader(f, skipinitialspace=True)]
            print("Successfully read from", fileName)

            return [process["Process ID"] for process in processesList]
    except IOError:
        print("Input/Output error!")
```

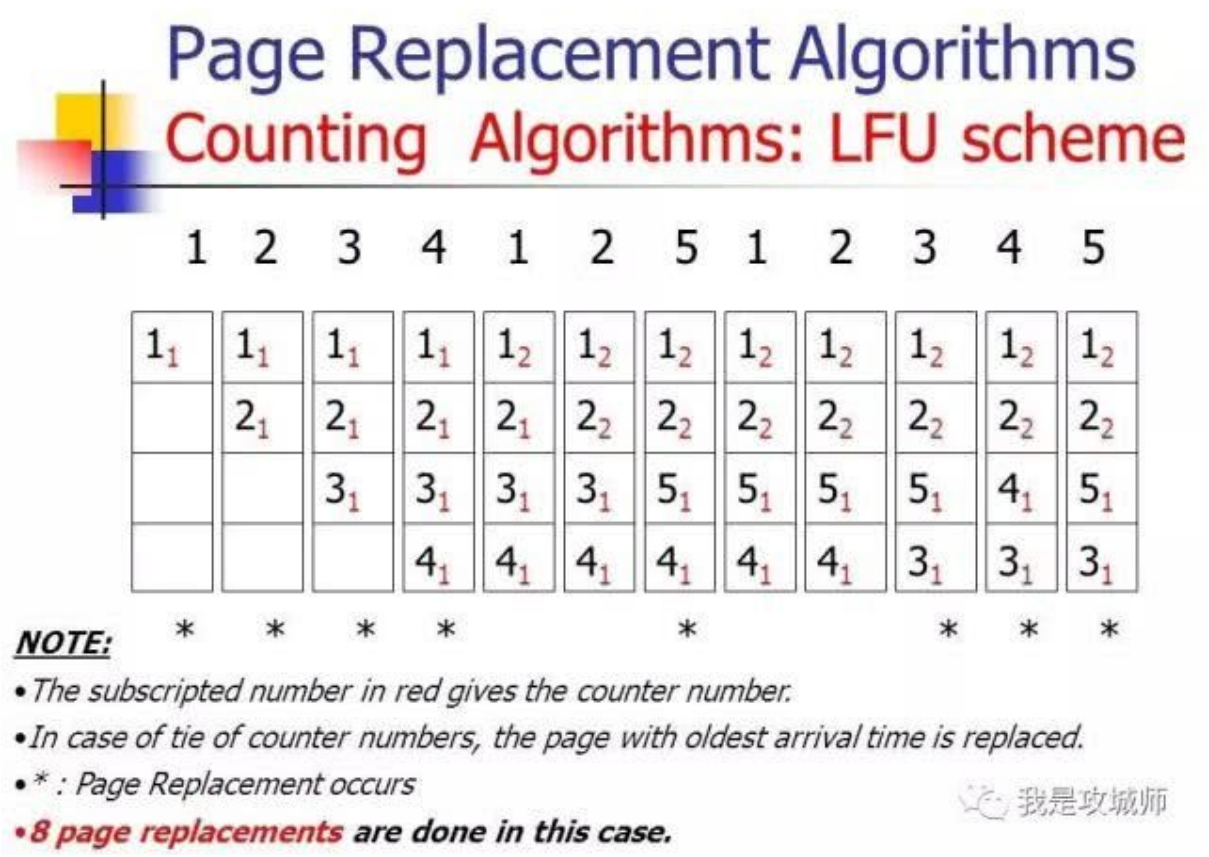
Wyniki:
Algorytm był przetestowany na stu różnych ciągach odwołań. Każde sto ciągów przetestowałem na liczbie ramek pamięci fizycznej dostępnych dla procesów ze zbioru {3, 5, 7}. Każdy z ciągów zawiera 100 elementów, reprezentujących 20 różnych procesów (losowo ustawionych). Oto są wyniki które otrzymałem:

```
For 3 free cache pages average number of page faults is 84.88
Successfully written to processesDataLRU_3.csv
For 5 free cache pages average number of page faults is 76.05
Successfully written to processesDataLRU_5.csv
For 7 free cache pages average number of page faults is 67.04
Successfully written to processesDataLRU_7.csv
```

Te wyniki będziemy mogli porównać z wynikami otrzymanymi z działania następnego algorytmu zastępowania stron LFU .

LFU

LFU (Least-Frequently-Used) - czyli "najrzadziej używany", jest to algorytm zastępowania stron, w którym ofiarą staje się strona, do której było najmniej odwołań. Zakłada się tu, że strony o dużych licznikach odwołań są aktywnie użytkowane i nie należy ich usuwać z pamięci.



Implementacja.

Implementację danego algorytmu, zarówno jak i poprzedniego algorytmu, dokonałem w języku Python.Procesy przedstawiłem jako słowniki, które mają dwie wartości, proces oraz licznik odwołań do niego, więc realizacja algorytmu jest intuicyjna.Wszystkie funkcje dla tworzenia list procesów ręcznie oraz randomowa są takie same jak przy implementacji algorytmu LRU.Funkcje dla zapisu listy procesów do pliku oraz odczytu listy procesów z pliku też są identyczne.Jedyną różnicą jest nazwa pliku który tworzy funkcja zapisująca w przypadku jak nie została podana nazwa pliku przez użytkownika.Jest to:

```
fileName:str="processesDataLFU.csv"
```

W związku z tym uważam za potrzebne tylko przedstawienie funkcji przedstawiającą logikę algorytmu.

Funkcja getNumberOfPageFaultsOfProcesses(processesList:list, emptyPageSlots:int):

Implementacja funkcji głównej jest prawie taka sama jak w przypadku funkcji głównej dla algorytmu LRU. Jedyną różnicą jest przedstawienie procesu jako słownika, ponieważ chcemy liczyć ilość odwołań do każdego procesy znajdującego się w ramkach pamięci.Przed usuwaniem elementu sotrujemy listę względem ilości odwołań oraz usuwamy proces z najmniejszą ilością licznika.

```
def getNumberOfPageFaultsOfProcesses(processesList:list, emptyPageSlots:int):
    """ Returns number of Page faults for given list of processes and number of empty page
    slots """

    #logic of this function is to store processes as a dictionaries with two keys: process ID
    and counter of requests
    #we will store all the processes in pageSlots list, that will represents our cache
    memory, and will have length = emptyPageSlots
    #while our list of processes in cache memory will not we full, we will aappend incoming
    processes to it
    #when pageSlots will be full, in case if we will get element that is not in this list, we
    will delete first element from it,
    #because this element will be automatically least frequently used one (we will sort list
    by request counter of processes before deleting) and then append the new one
    #in case if element will be it list we will increment it's request counter

    pageFault = 0
    pageSlots = []
    for process in processesList:
        if not any(dct["Process ID"] == process for dct in pageSlots):
            if len(pageSlots) < emptyPageSlots:
                pageSlots.append({"Process ID": process,
                                  "Request Counter": 0})
            else:
                pageSlots = sorted(pageSlots, key = lambda i: i["Request Counter"])
                del pageSlots[0]
                pageSlots.append({"Process ID": process,
                                  "Request Counter": 0})

            pageFault += 1
        else:
            for cacheProcess in range(len(pageSlots)):
                if pageSlots[cacheProcess]["Process ID"] == process:
                    pageSlots[cacheProcess]["Request Counter"] += 1


    return pageFault
```

Wyniki:

Sposób testowania został taki sam jak dla algorytmu LRU. Algorytm był przetestowany na stu różnych ciągach odwołań. Każde sto ciągów przetestowałem na liczbie ramek pamięci fizycznej dostępnych dla procesów ze zbioru {3, 5, 7}.

Każdy z ciągów zawiera 100 elementów, reprezentujących 20 różnych procesów (losowo ustawionych). Oto są wyniki które otrzymałem:

```
For 3 free cache pages average number of page faults is 85.62
Successfully written to processesDataLFU_3.csv
For 5 free cache pages average number of page faults is 76.36
Successfully written to processesDataLFU_5.csv
For 7 free cache pages average number of page faults is 66.27
Successfully written to processesDataLFU_7.csv
```

Porównywanie algorytmów zastępowania stron LRU i LFU

Kryteria oceny algorytmów:

- Średnia liczba brakujących stron.

Widzimy że uśrednione wyniki otrzymane przy testowaniu obu algorytmów są praktycznie identyczne.Dla żadnej ilości bramek pamięci ze zbioru {3, 5, 7} **wynik nie różni się więcej niż o 1,15%** (dla ilości bramek 7)**.**