Tema nr. 1

Memory Allocation Simulator – ma_sim.py

Alin Clincea, 221

1. Descriere

Scopul aplicației *ma_sim.py* este de a ilustra câțiva algoritmi de alocare a memoriei*: first fit, best fit* și *worst fit*. Aplicația este scrisă în Python 3.9.4 și poate funcționa atât ca program independent, cât și ca modul importabil.

2. Mod de utilizare

Aplicația poate fi executată dintr-un terminal folosind următoarea comandă în directorul unde se află scriptul *ma_sim.py*:

py -i ma_sim.py [-ff | -bf | -wf]

În funcție de sistemul de operare, executabilul pentru Python poate fi *py* (Windows), *python3* (Linux), *python* (MacOS) etc.

Switch-ul *-i*, deși opțional, totuși foarte recomandat aici, permite executarea interactivă a scriptului: după executarea instrucțiunilor, prompt-ul *py* va rămâne activ, permițând introducerea de comenzi în contextul scriptului deja încărcat în memorie.

Aplicația poate fi pornită cu unul din parametrii *[-ff | -bf | -wf]* - corespunzători algoritmilor de alocare suportați. În lipsa lor, algoritmul implicit este *first fit (-ff)*.

Interacțiunea cu aplicația se realizează prin introducerea de comenzi care pot crea, modifica sau distruge obiectul *Memory*.

Interacțiunea minimală presupune crearea de noi procese prin comenzi de tipul:

m + 200 - semnificând „încearcă alocarea a 200 de unități pentru un proces”

respectiv, ștergerea proceselor existente prin comenzi de tipul:

m - 1 - semnificând „șterge procesul cu *pid 1*, dacă există”

Feedback-ul în cazul comenzilor corect introduse este o reprezentare aproximativă a conținutului memoriei (bar-chart cu gradații 0 - 100 și două culori: verde pentru spațiu liber, roșu pentru spațiu alocat) și un tabel cu descrierea conținutului memoriei și a ratei de succes a alocărilor.

Modificarea algoritmului de alocare a memoriei poate fi făcută și după începerea execuției, prin schimbarea proprietății corespunzătoare a obiectului *Memory*: ex. *m.algorithm = "wf"*.

Crearea de noi obiecte *Memory* se poate realiza prin instrucțiuni de tipul: *n = Memory(20000)*, unde valoarea introdusă reprezintă capacitatea noii memorii alocate (implicit, capacitatea este 1000).

Distrugerea unui obiect *Memory* se poate face folosind comanda standard Python: *del m* (unde m este obiectul).

3. Implementare

3.1 Dependințe

Aplicația importă modulul *sys* (instalat odată cu Python; necesar pentru citirea parametrilor din linia de comandă) și necesită instalarea prealabilă a modulului *colorama* (o bibliotecă Python care permite

modificarea afișării textului în mod consolă, utilizată aici pentru posibilitatea de a schimba culoarea textului), instalabil prin comanda: *pip install colorama*.

3.2 Detalii de implementare

Memoria este reprezentată prin clasa *Memory*, care conține o subclasă *Block*.

Un obiect *Block* reprezintă o secțiune a memoriei. *Block*-ul poate fi proces sau spațiu liber (atributul boolean *.process*). Alte atribute se referă la identificare (*.pid*), dimensiune (*.size*) și adresă de început (*.address*). Adresa finală se poate calcula folosind relația: *af = address + size*.

Conținutul memoriei este stocat ca listă Python de obiecte *Block*, ce include toate operațiile necesare manipulării ei (ex. inserție de elemente, ștergere de elemente).

Operațiile principale ale obiectului *Memory* sunt: adăugarea unui bloc (*add_block()*), ștergerea unui bloc (*remove_block()*) și afișarea memoriei (*print_memory()*). Orice adăugare sau ștergere a unui bloc de memorie este urmată de apelarea funcției de afișare.

Funcția *add_block(size)* este un wrapper care selectează algoritmul intern (apelând funcțiile interne *_ff(), bf()* sau *_wf()*), conform setării atributului *.algorithm*.

Funcția *remove_block(pid)* înlătură un bloc ce reprezintă un proces. După modificarea atributelor blocului, sunt verificate elementele vecine (dreapta-stânga) din lista ce reprezintă conținutul memoriei; dacă un element vecin este tot spațiu liber, se realizează compactarea memoriei într-un singur bloc, cu adresa cea mai mică dintre cele comparate.

Pentru simplificarea utilizării aplicației, operatorii + și - (binari) sunt supraîncărcați în clasa *Memory*, astfel încât:

- alocarea unui proces se face prin:
  m.add_block(size) sau *m + size*
- ștergerea unui proces (și convertirea lui în spațiu liber) se face prin:
  m.remove_block(pid) sau *m – pid*

4. Experimente

Pentru testarea aplicației am scris un alt program (*experiments.py*) care importă modulul *ma_sim* și generează aleatoriu parametri (liste cu dimensiuni de procese, listă cu *pid* de eliminat).

Aceste liste sunt generate cu următoarele specificații:

1. *initial_sizes* și *final_sizes*: liste cu dimensiuni (configurabile) aleatorii de procese;
2. *pids*: listă aleatorie cu *pid*-uri de eliminat, pentru a se crea „găuri" de memorie.

Principiul de funcționare este: alocarea unui număr de procese, crearea de găuri de memorie și realocarea de noi procese pentru a se observa rata de succes a alocărilor.

Am utilizat aceiași parametri în cele trei opțiuni de algoritm de alocare și am calculat rata de succes finală. Fiecare astfel de experiment este repetat de un număr de ori la alegerea utilizatorului, iar rezultatele finale (bazate pe o listă de tuple cu ratele de succes) sunt cumulate corespunzător fiecărui algoritm.