Structuri de date (Seria CB)
Tema 2 - *MiniOS*

Responsabili temă	Cosmin-Dumitru Oprea, Bianca-Mihaela Cauc, Andrei Cherecheşu
Data publicării	**28.03.2019**
Termen predare	**18.04.2019 (ora 23:55)** Se acceptă teme trimise cu penalizare de 10 puncte/zi (din maxim 100 puncte) până la data de **21.04.2019** (**ora 23:55)**
Versiune document	2

1. Introducere

Să ne imaginăm că vrem să ne creăm propriul nostru sistem de operare numit MiniOS. Pentru aceasta vom implementa următoarele două componente:

1. modul pentru gestionarea memoriei
2. modul pentru gestionarea proceselor

1.1 Modulul pentru gestionarea memoriei

Fiecare proces care este creat în cadrul unui sistem de operare are alocată o zonă de memorie de dimensiune fixă şi care este stabilită la crearea procesului. La crearea unui proces, acesta specifică dimensiunea necesară a memoriei, iar acest sistem îi rezervă procesului, dacă este posibil, o zonă de memorie de dimensiunea cerută.

1.2 Modulul pentru gestionarea proceselor

Sistemul nostru de operare va rula pe un procesor cu un singur nucleu, iar pentru a putea rula mai multe procese în cadrul sistemului nostru de operare, avem nevoie de un planificator. Acest modul se va ocupa de crearea şi planificarea pentru rulare a proceselor.

3. Cerinţă

Scopul temei este de a implementa cele două module componente ale sistemului nostru de operare. Vom caracteriza un proces prin următoarele atribute:

- **PID**, identificatorul unic al procesului în cadrul sistemului de operare:
    - este atribuit automat la crearea procesului de către modulul pentru gestiunea proceselor. Valoare PID-ului este un număr natural cuprins între [1 - 32767]
    - PID-ul este atribuit ca fiind cel mai mic număr disponibil
    - dacă un proces a avut PID-ul X iar acesta şi-a terminat execuţia, PID-ul X devine disponibil
- **prioritate**, este un număr natural cuprins intre [0 - 127]
    - trimisă ca parametru la crearea procesului

- **timp de execuţie**, număr natural pozitiv reprezentabil pe 32 de biţi
    - trimis ca parametru la crearea procesului
- **începutul zonei de memorie**
    - este alocată automat de către modulul pentru gestionarea memoriei
- **dimensiunea zonei de memorie**, număr natural pozitiv <= 3 * (1024 ^ 2)
    - trimis ca parametru la crearea procesului

3. Implementare

3.1 Sistemul de gestiune a memoriei

Din cauza alocărilor şi dealocărilor repetate, are loc fragmentarea memoriei, caz ce poate duce la imposibilitatea sistemului de operare de a mai crea procese noi. Astfel, modulul de gestiune a memoriei va avea implementată o funcţionalitate pentru defragmentarea memoriei. Aceasta funcţionalitate va fi folosită automat de fiecare dată când modulul nu reuşeşte să aloce memorie pentru un proces nou. Defragmentarea se va efectua prin mutarea memoriei proceselor în ordinea crescătoare a PID-ului începând de la adresa 0. Astfel, memoria procesul cu PID-ul cel mai mic va fi mutată la adresa 0, memoria procesului cu al doilea cel mai mic PID va fi mutată la adresa 0 + dimensiunea memoriei procesului cu PID-ul cel mai mic, ş.a.m.d. Sistemul nostru de operare va adresa o memorie totală de 4MiB, zona din intervalul [3MiB, 4MiB] fiind rezervată de către sistem, restul memoriei de la [0MiB la 3MiB) fiind folosită pentru memoria proceselor.

[0MiB……3MiB).....4MiB]

Memorie proces PID 3	Memorie proces PID 1	Memorie liberă	Memorie proces PID 2	Memorie liberă	Memorie rezervată de OS

Figura 1: Memoria sistemului înainte de defragmentare

Memorie proces PID 1	Memorie proces PID 2	Memorie proces PID 3	Memorie liberă	Memorie liberă	Memorie rezervată de OS

Figura 2: Memoria sistemului după defragmentare

Sistemul va funcţiona după următoarele reguli:

- va aloca pentru procesul nou creat o zonă de memorie aflată la cea mai mica adresă disponibilă
- în cazul în care nu se poate aloca memorie, se rulează o defragmentare şi se reia procesul; dacă nici după defragmentare nu se poate aloca memorie pentru proces, sistemul de gestiune a memoriei va afişa mesajul `"Cannot reserve memory for PID <PID>.\n"`.
- toată memoria fiecărui proces va fi de tip stivă
- fiecare proces va avea dreptul de a face push / pop pe stiva proprie
- dacă procesul are stiva plină şi face push, sistemul va afişa mesajul `"Stack overflow PID <PID>.\n"`
- dacă procesul are stiva goală şi face pop, sistemul va afişa mesajul `"Empty stack PID <PID>.\n"`

3.2 Sistemul de gestiune a proceselor

Acest sistem are o coadă de aşteptare în care sunt ţinute procesele care aşteaptă eliberarea procesorului pentru a putea rula. Un proces se poate afla în una din următoarele trei stări:

- ***waiting***
    - procesul se află în ***coada de aşteptare***
- ***running***
    - rulează activ pe procesor
- ***finished***
    - procesul se află în ***coada de procese terminate***

Sistemul funcţionează după următoarele reguli:

- are definită o cuantă de timp reprezentând timpul maxim de rulare continuă a unui proces
    - dimensiunea unei cuante este de T milisecunde (ms), T va fi citit din fişierul de intrare.
- procesele din ***coada de aşteptare*** sunt ordonate astfel:
    - descrescător după prioritate
    - crescător după timp de execuţie rămas, în caz de priorităţi egale
    - crescător după PID, în caz de prioritate şi timp de execuţie rămas egale.
- procesele din ***coada finished*** sunt ordonate astfel:
    - procesul care şi-a terminat primul execuţia este primul în coadă
    - procesul care şi-a terminat ultimul execuţia este ultimul în coadă
- se alege pentru rulare primul proces din coada de aşteptare
- după expirarea cuantei de timp a procesului curent:
    - dacă nu mai sunt alte procese în sistem, acesta îşi continuă rularea
    - altfel, procesul este trecut din starea *running* în starea *waiting*
    - primul proces din coada de aşteptare este trecut din starea *waiting* în starea *running,* urmând ca apoi să fie scos din coadă şi să înceapă execuţia
    - este adăugat în coada de aşteptare procesul care a trecut în starea *waiting*.
- dacă un proces îşi termina execuţia:
    - i se eliberează memoria, urmând ca aceasta să devină disponibilă pentru procese noi
    - PID-ul asociat procesului devine disponibil şi poate fi atribuit proceselor create ulterior
    - acesta este mutat în ***coada de procese terminate***
    - primul proces din coada de aşteptare este trecut în starea *running*
- dacă nu mai sunt procese în sistem, va trece spre rulare procesul *Idle (PID = 0)* care este un proces implicit al sistemului de operare. Acest proces rulează până când apare un proces în sistem, caz în care este înlocuit la rulare imediat de acesta.
- la crearea unui proces acesta este inserat în ***coada de aşteptare*** sau este pus în direct în starea *running* în cazul în care în sistem există doar procesul *Idle*.
- procesul *Idle* nu are stivă

4. Descrierea operaţiilor şi a datelor de intrare

Rezolvarea temei va consta în efectuarea unui set de operaţii descrise în fişierul de intrare.

A. Adăugare proces în coada de aşteptare

***Sintaxă*:** `add <mem_size_in_bytes> <exec_time_in_ms> <priority>`

Mod de funcţionare: Creează un proces nou cu atributele date şi îl inserează în coada de aşteptare după următoarele criterii:

- descrescător după prioritate
- crescător după timp de execuţie rămas, în caz de priorităţi egale
- crescător după PID, în caz de prioritate şi timp de execuţie rămas egale.

În caz de succes afişează un mesaj de forma:

```
Process created successfully: PID: <PID>, Memory starts at
<hex_memory_address>.\n
```

Exemplu: `add 1048576 3000 7` */* este creat un proces cu o memorie de 1MB, timp de rulare 3s şi prioritate 7 */*

```
Process created successfully: PID: 1, Memory starts at 0x3c0.
```

Se garantează că la orice moment de timp va fi disponibil cel puţin un PID pentru un proces nou. Dacă în sistem există doar procesul *Idle*, procesul nou creat va fi trecut direct în starea *running*, fără a mai fi adăugat în coada de aşteptare.

B. Determinarea stării unui proces

Sintaxă: `get <PID>`

Exemplu: `get 1023`

Mod de funcţionare: Comanda cauta toate procesele cu identificatorul <PID> şi le afişează in formatul de mai jos:

- proces în starea ***running***:
  `Process <PID> is running (remaining_time = timp_ramas).\n`
- proces în starea ***waiting***:
  `Process <PID> is waiting (remaining_time = timp_ramas).\n`
- proces în starea ***finished***:
  `Process <PID> is finished.\n`
- În cazul în care in sistem nu a existat niciun proces cu PID-ul dat, se va afişa mesajul:
  `Process <PID> not found.\n`

Prima data se verifica procesul din starea ***running***, apoi procesele din ***coada de aşteptare***, iar apoi procesele din ***coada de procese terminate***. Cele doua cozi se vor parcurge de la primul element spre ultimul.

C. Salvarea datelor pe stivă

***Sintaxă*:** `push <PID> <4_signed_bytes_data>`

Exemple: `push 1023 2147483647`
`push 1024 -2147483648`

Mod de funcţionare: Pune pe stiva procesului 4 octeţi. În caz de stivă plină afişează mesajul: `"Stack overflow PID <PID>.\n"` şi nu va adaugă nimic în stivă. În cazul în care nu există niciun proces cu PID-ul dat se va afişa mesajul: `"PID <PID> not found.\n"`.

D. Ştergerea datelor de pe stivă

***Sintaxă*:** `pop <PID>`

Exemplu: `pop 1023`

Mod de funcţionare: Elimină primii 4 octeţi din vârful stivei procesului. În caz de stivă goală afişează mesajul: `"Empty stack PID <PID>.\n".` În cazul în care nu există niciun proces cu PID-ul dat se va afişa mesajul: `"PID <PID> not found.\n"`.

E. Afişarea stivei unui proces

Sintaxă: `print stack <PID>`

Mod de funcţionare: Afişează doar conţinutul plin al stivei în felul următor:
`Stack of PID <PID>: 4_byte1 4_byte2 … 4_byteN.\n`,
unde:
4_byte1 sunt primii 4 octeţi de la baza stivei, iar 4_byteN sunt ultimii 4 octeţi de la vârful stivei.

Exemplu:
`Stack of PID 1023: 5353 4324122 2147483647 -2147483648.`
În caz de stivă goală afişează mesajul: `"Empty stack PID <PID>.\n".`

F. Afişarea cozii de aşteptare

Sintaxă: `print waiting`

Mod de funcţionare: Afişează coada de aşteptare a planificatorului la momentul curent în felul următor:

```
Waiting queue:\n
[(<PID1>: priority = <prioritate1>, remaining_time =
<timp_execuţie_rămas1>),\n
…,
(<PIDN>: priority = <prioritateN>, remaining_time =
<timp_execuţie_rămasN>)]\n,
```

unde PID1 corespunde primului proces aflat în coada de aşteptare a planificatorului, iar PIDN corespunde ultimului proces din coada de aşteptare.

Exemplu:

```
Waiting queue:
[(1452: priority = 72, remaining_time = 100),
(1521: priority = 55, remaining_time = 250),
```

```
(1245: priority = 55, remaining_time = 320),
(1352: priority = 23, remaining_time = 500)]
```

G. Afişarea cozii de procese terminate

Sintaxă: `print finished`

Mod de funcţionare: Afişează coada de procese terminate astfel:

`Finished queue:\n`

`[(<PID1>: priority = <prioritate1>,` `executed_time =` *`<timp_execuţie1>)\n,`*

…,

`(<PIDN>: priority = <prioritateN>,` `executed_time =` *`<timp_execuţieN>)]\n`*,

unde PID1 corespunde primului proces aflat în coada de aşteptare a planificatorului, iar PIDN corespunde ultimului proces din coada de aşteptare.

Exemplu:

```
Finished queue:
[(1452: priority = 72, executed_time = 134),
(1521: priority = 55, executed_time = 14),
(1245: priority = 55, executed_time = 156)]
```

H. Executarea unui număr de unităţi de timp

Sintaxă: `run <număr_unităţi_timp>`

Mod de funcţionare:

Execută următoarele `număr_unităţi_timp` de timp pe procesor.

În cazul în care în sistem nu există niciun proces, se va rula procesul *`Idle`* care nu face nimic.

I. Terminarea execuţiei tuturor proceselor

Sintaxă: `finish`

Mod de funcţionare: Se continuă execuţia până când toate procesele ajung în coada de procese terminate.

La finalul execuţiei, se afişează timpul total necesar pentru terminarea proceselor neterminate în momentul apelării comenzii *finish*:

`Total time: <total_execution_time>\n`

În datele de intrare va exista maxim un apel al comenzii ***finish***, iar după acesta nu vor mai exista alte comenzi.

5. Restricţii şi precizări:

- 1 <= T <= 1000, dimensiunea unei cuante de timp este de maxim o secundă
    - va fi citit de pe prima linie din fişierul de intrare
- 0 <= **linii în fişierul de intrare** <= 10.000
- 0 <= **număr total de procese în sistem** <= 32767
- 1KiB = 1024 Bytes
- 1MiB = 1024 KiB
- `mem_size_in_bytes` este multiplu de 4
- 4 Bytes <= `mem_size_in_bytes` <= 3MiB
- 0 <= `priority` <= 127
- 0 < `exec_time_in_ms < 10.000.000`
- se garantează că datele de intrare vor fi corecte
- programul va fi rulat astfel: **./tema2 in_file out_file**
- comenzile se citesc din fişierul ***in_file***, iar rezultatele se scriu în fişierul ***out_file***
- stivele şi cozile vor fi implementate ca **liste generice simplu înlănţuite**
- **nu** aveţi voie cu **variabile globale**
- **nu** aveţi voie sa iteraţi prin structuri; folosiţi doar operaţiile specifice pentru stive şi cozi
- pentru sortări, afişări se vor folosi **doar stive/cozi auxiliare** (nu se permite iterarea prin stive/cozi)
- 40% din testele de intrare vor respecta următoarele condiţii:
    - timpii de execuţie ai proceselor vor fi multipli ai lui T
    - argumentul comenzii *run* va fi multiplu al lui T

6. Exemple:

Intrare	Ieşire
500 add 1000000 500 10 add 1000000 500 20 add 1000000 500 30 run 500 print finished run 500 print finished	Process created successfully: PID: 1, Memory starts at 0x0. Process created successfully: PID: 2, Memory starts at 0xf4240. Process created successfully: PID: 3, Memory starts at 0x1e8480. Finished queue: [(1: priority = 10, executed_time = 500)] Finished queue: [(1: priority = 10, executed_time = 500), (3: priority = 30, executed_time = 500)]

Explicaţie:

Primul proces care rulează este cel care a fost creat primul deoarece acesta a fost trecut direct in starea `running`. Următorul proces la rulare este cel cu PID=3, acesta având prioritatea mai mare decât cea a procesului cu PID=2.

Intrare	Ieşire
500 add 240 463 23 add 720 1939 10 add 860 1939 10 get 1 print waiting run 500 get 1 get 2 get 3 push 2 15 push 2 17 push 2 18 pop 2 print stack 2 run 1939 get 1 get 2 get 3 run 1000 get 1 get 2 get 3 print waiting print finished finish	Process created successfully: PID: 1, Memory starts at 0x0. Process created successfully: PID: 2, Memory starts at 0xf0. Process created successfully: PID: 3, Memory starts at 0x3c0. Process 1 is running (remaining_time: 463). Waiting queue: [(2: priority = 10, remaining_time = 1939), (3: priority = 10, remaining_time = 1939)] Process 1 is finished. Process 2 is running (remaining_time: 1902). Process 3 is waiting (remaining_time: 1939). Stack of PID 2: 15 17. Process 1 is finished. Process 2 is waiting (remaining_time: 939). Process 3 is running (remaining_time: 963). Process 1 is finished. Process 2 is waiting (remaining_time: 439). Process 3 is running (remaining_time: 463). Waiting queue: [(2: priority = 10, remaining_time = 439)] Finished queue: [(1: priority = 23, executed_time = 463)] Total time: 902

Explicație:

Se adaugă trei procese în sistem a căror memorie se alocă consecutiv la adresele 0 (hex 0x0), 240 (hex 0xf0), 960 (hex 0x3c0). Cele trei procese primesc în ordine valorile 1, 2 şi 3 pentru PID. Primul proces trece direct în starea *running*, fiind singurul proces din sistem la momentul adăugării. Procesele 2 şi 3 sunt în coada de aşteptare, sortate după PID (au prioritatea şi timpul de execuţie egale). Sistemul execută primele 500 de ms, timp în care procesul 1 se termină, iar procesul 2 rulează pentru 37 de ms, rezultând un timp de execuţie rămas de 1902 ms. Se execută următoarele 1939 ms astfel: în primele 500-37=463 ms rulează în continuare procesul 2, după care acestuia i se termina cuanta de timp. Astfel procesul 2 rămâne cu un timp de execuţie de 1902-(463)=1439. În următoarele 500 de ms rulează procesul 3 căruia ii mai rămân de rulat 1939-500=1439. Apoi rulează 500 de ms procesul 2 şi îi mai rămân de rulat 1439-500=939 ms. În următoarele 1939-463-500-500=476 de ms rulează procesul 3 căruia îi mai rămân de rulat 1439-476=963 de ms. În următoarele 1000 de ms procesul 3 rulează 500-476=24 ms până i se termină cuanta de timp, apoi rulează 500 de ms procesul 2 care se termină, iar apoi rulează restul de 476 de ms procesul 3. Astfel procesul procesului 2 îi rămân 939-24-476=439 de ms, iar procesului 3 îi mai rămân de rulat 963-500=463 de ms. La rularea comenzii *finish* mai este nevoie de 439+463=902 ms.

7. Notare:

- **85 de puncte** obţinute pe testele de pe vmchecker;
- **10 puncte**: coding style;
- **5 puncte**: README - va conţine detaliile de implementare a temei, precum şi **punctajul obţinut la teste** (la rularea pe calculatorul propriu);
- **bonus: 20 de puncte** pentru soluţiile care nu au memory leak-uri (bonusul se acordă doar dacă testul a trecut cu succes);
- temele care nu compilează, nu rulează sau obţin punctaj 0 pe vmchecker, indiferent de motive, vor primi punctaj 0;
- se depunctează pentru:
    - warninguri la compilare (trebuie compilat cu -Wall);
    - linii mai lungi de 80 de caractere;
    - folosirea incorectă de pointeri;
    - neverificativa codurilor de eroare;
    - neeliberarea resurselor folosite (trebuie eliberarea memoriei alocate, închiderea fişierelor etc);
    - alte situaţii nespecificate aici, dar considerate inadecvate;

8. Reguli de trimitere a temelor:

- temele vor trebui încărcate atât pe vmchecker (în secţiunea **Structuri de Date Seria CB**) cât şi pe acs.curs.pub.ro, în secţiunea aferentă temei 2.
- arhiva cu rezolvarea temei trebuie să fie **.zip** şi să conţină:
- fişiere surse (fiecare fişier sursă va trebui sa înceapă cu un comentariu de forma:

  / NUME Prenume - grupa */*

- fişier **README**, denumit obligatoriu astfel, care să conţină detalii despre implementarea temei
- fişier **Makefile**, denumit obligatoriu astfel, cu două reguli:
    - **build**, care va compila sursele şi va obţine executabilul cu numele **tema2**
    - **clean**, care va şterge executabilele şi alte fişiere obiect generate
- arhiva trebuie să conţină doar fişierele sursa (inclusiv Makefile şi README); **nu** se acceptă fişiere executabile sau obiect
- dacă arhiva nu respectă specificaţiile de mai sus nu va fi acceptată la upload şi tema nu va fi luata în considerare.