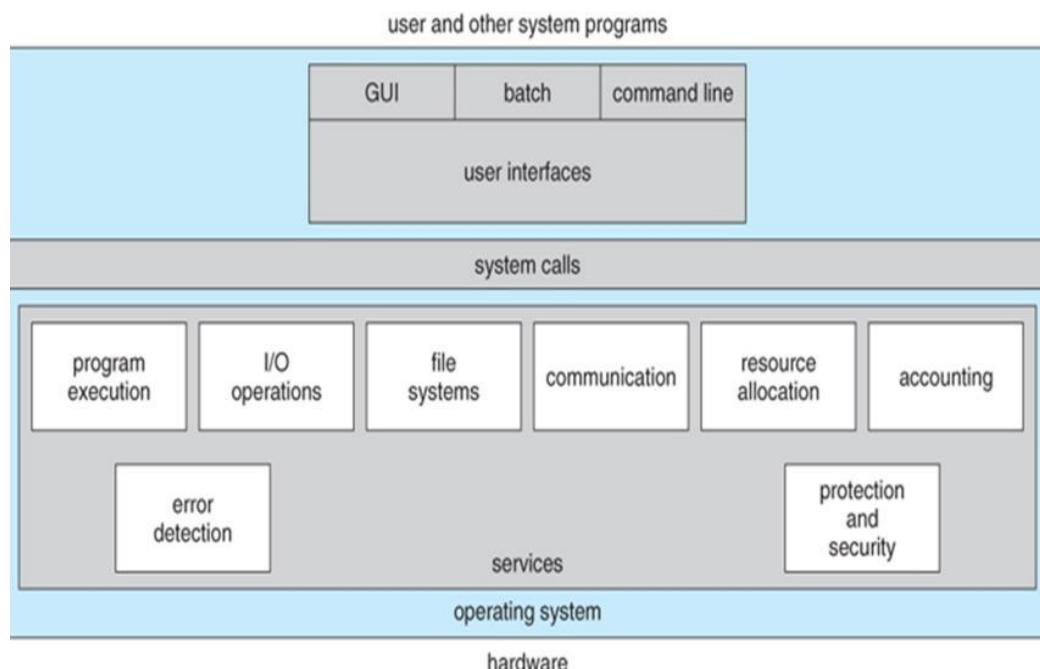


ლექცია 1. შესავალი

გამოთვლითი ტექნიკა გამოჩენის პირველივე დღიდან მუდმივად ვითარდება, იძენს ახალახალ ფიზიკურ კომპონენტებს და შესაძლებლობებს. თანამედროვე კომპიუტერი შედგება ერთი ან რამდენიმე პროცესორისგან, ოპერატორი მეხსიერებისგან, მყარი დისკისგან, მონიტორისგან, შეტანა/გამოტანის სხვადასხვა მოწყობილობისგან და მრავალი პერიფერიული მოწყობილობისგან (პრინტერი, სკანერი და ა.შ.). ფიზიკური კომპონენტების (რესურსების) გარდა გამოთვლითი მანქანა აღჭურვილია ვირტუალური რესურსებით, რომლებიც ხელს უწყობენ ამა თუ იმ ფიზიკური რესურსის ნორმალურ ფუნქციონირებას (დრაივერები) ან ემსახურებიან მომხმარებლის კომფორტულ საქმიანობას (პროგრამები). შედეგად მიიღება რთული სისტემა, რომლის ეფექტური მართვა საკმარისად რთულ ამოცანას წარმოადგენს. ფიზიკური კომპონენტების ეფექტური მართვის მიზნით ყოველი გამოთვლითი მანქანა აღჭურვილია სპეციალური პროგრამული უზრუნველყოფით, რომელსაც **ოპერაციული სისტემა** (operating system, OS) ეწოდება.



ნახ. 1.1. ოპერაციული სისტემის ადგილი პროგრამულ უზრუნველყოფაში

ნახ. 1.1-ზე წარმოდგენილია თანამედროვე კომპიუტერის სტრუქტურა. როგორც ნახატიდან ვხედავთ, ქვედა ნაწილში განთავსებულია აპარატურული უზრუნველყოფა (Hardware). ის შედგება ინტეგრალური სექტორისგან, მყარი დისკისგან, კლავიატურისგან და სხვა მსგავსი ფიზიკური მოწყობილობებისგან. მის ზემოთ განთავსებულია პროგრამული უზრუნველყოფა (Software).

რადგანაც ოპერაციული სისტემა გამოიყენება გამოთვლითი მანქანის შემადგენელი ფიზიკური და ვირტუალური რესურსების ეფექტური მუშაობის უზრუნველსაყოფად, ამიტომ მას უნდა გააჩნდეს აბსოლუტური წვდომა ყველა რესურსზე, ხოლო სამომხმარებლო პროგრამებს კი ასეთი წვდომა არ ჭირდებათ. აუცილებლობის შემთხვევაში მათ უნდა მიმართონ ოპერაციულ სისტემას შესაბამისი მოქმედების განხოციელების მოთხოვნით.

გამოყენებით მანქანაში რესურსებზე არასანქცირებული წვდომის აკრძალვის მიზნით გამოიყენება პროგრამული უზრუნველყოფის ამუშავების ორი რეჟიმი: **ბირთვის** (kernel mode), რომელშიც ამუშავებულია ოპერაციული სისტემა და ოპერაციული სისტემის დამხმარე სპეციფიური პროგრამული უზრუნველყოფები და **მომხმარებლის** (user mode), რომელშიც მუშაობენ სამომხმარებლო პროგრამები. თითოეულ რეჟიმში შესაძლებელია მხოლოდ ამ რეჟიმისთვის დამახასიათებელი ბრძანებების შესრულება.

როგორც ბევრთ აღვნიშნეთ, ოპერაციული სისტემა არის პროგრამული უზრუნველყოფის მნიშვნელოვანი კომპონენტი, რომელიც მუშაობს ბირთვის რეჟიმში. ამ რეჟიმში მას აქვს სრული წვდომა აპარატურულ უზრუნველყოფაზე და, კომპიუტერის დანიშნულებიდან გამომდინარე, შეუძლია შეასრულოს სხვადასხვა სირთულის ნებისმიერი ინსტრუქცია. პროგრამული უზრუნველყოფის დარჩენილი ნაწილი მუშაობს მომხმარებლის რეჟიმში, სადაც შესაძლებელია მხოლოდ შეზღუდული რაოდენობის ინსტრუქციების შესრულება.

სამომხმარებლო ინტერფეისის (user interfaces) პროგრამები განთავსებულია მომხმარებლის რეჟიმში მომუშავე პროგრამული უზრუნველყოფის ზედა ნაწილში და მომხმარებელს საშუალებას აძლევენ აამუშაოს ისეთი გამოყენებითი პროგრამები, როგორიცაა ტექსტური რედაქტორი, ელექტრონული წერილის კითხვის პროგრამა და ა.შ.

მნიშვნელოვანი განსხვავება ოპერაციულ სისტემასა და ჩვეულებრივ პროგრამულ უზრუნველყოფას შორის მდგომარეობს იმაში, რომ მომხმარებელი, რომელიც უკმაყოფილო კონკრეტული პროგრამული უზრუნველყოფის მუშაობით, მაგალითად როგორიცაა, ტექსტური რედაქტორი ან ელექტრონული წერილის კითხვის პროგრამა, შეუძლია ჩაანაცვლოს ის სხვა არსებული პროგრამული უზრუნველყოფით ან დაწეროს საკუთარი პროგრამა, მაგრამ მას არ შეუძლია დაწეროს სისტემური დროის წყვეტის საკუთარი პროგრამული უზრუნველყოფა, რომელიც წარმოადგენს ოპერაციული სისტემის ნაწილს და აპარატურულ დონეზე დაცულია მომხმარებლის მხრიდან რაიმე სახის ცვლილების შეტანისგან.

არსებობენ პროგრამული უზრუნველყოფები (სისტემური პროგრამები), რომლებიც მომხმარებლის რეჟიმში მუშაობის მიუხედავად ასრულებენ ოპერაციული სისტემისთვის განკუთვნილ ამოცანებს ან ასრულებენ გარკვეულ სპეციფიურ ფუნქციებს.

ყველა პროგრამა, რომელიც მუშაობს ბირთვის რეჟიმში წარმოადგენს ოპერაციული სისტემის ნაწილს. ასევე, შესაძლებელია, რომ მომხმარებლის რეჟიმში მომუშავე პროგრამებიც წარმოადგენდნენ ოპერაციული სისტემის ნაწილს ან იმყოფებოდნენ მასთან მჯიდრო კავშირში.

ოპერაციული სისტემა გამოყენებითი პროგრამებისგან განსხვავდება არამხოლოდ მისი ადგილმდებარეობით პროგრამულ უზრუნველყოფაში, არამედ მისი დიდი მოცულობითა და რთული სტრუქტურით. თანამედროვე ოპერაციული სისტემების, როგორიცაა Windows ან Linux ოპერაციული სისტემები, პროგრამული კოდი აქარბებს რამდენიმე ათეულ მილიონ სტრიქონს.

1.1. რა არის ოპერაციული სისტემა?

ოპერაციული სისტემისათვის ზუსტი განსაზღვრების მიცემა შეუძლებელია. შეიძლება ითქვას, რომ ოპერაციული სისტემა არის პროგრამული უზრუნველყოფა, რომელიც მუშაობს ბირთვის რეჟიმში, მაგრამ ეს ყოველთვის არ შეესაბამება სინამდვილეს. ის ფაქტი, რომ ოპერაციული სისტემისათვის ზუსტი განმარტების მიცემა შეუძლებელია განპირობებულია ოპერაციული სისტემის მიერ შესასრულებელი ორი ფუნქციით: სავსებით გაურკვეველი აპარატული უზრუნველყოფის ნაცვლად გამოყენებით პროგრამისტს შესთავაზოს მისთვის გასაგები რესურსების აბსტრაქტული ნაკრები და ეფექტურად მართოს ისინი.

1.1.1. ოპერაციული სისტემა როგორც განზოგადოებული მანქანა

კომპიუტერის არქიტექტურა მანქანურ ბრძანებათა დონეზე საკმაოდ პრიმიტულია, ხოლო მათი გამოყენება სამომხმარებლო პროგრამებში მოუხერხებელი. ეს ძირითადად შეეხება შეტანა/გამოტანის სისტემას. მაგალითად, დისკთან მუშაობა გულისხმობს მისი შიდა მოწყობილობის ელექტრონული კომპონენტების ცოდნას, დისკის ბრუნვისთვის ბრძანებათა შეტანის კონტროლერს, ბილიკების მოძრვას და ფორმატირებას, სექტორების კითხვას და ა.შ. ცხადია, რომ საშუალო დონის პროგრამისტს მოწყობილობათა მუშაობის ყველა ამ თავისებურების გათვალისწინება არ შეუძლია. მისთვის საკმარისია გააჩნდეს მარტივი მაღალდონიანი აბსტრაქცია, მაგალითად, დისკის ინფორმაციული სივრცე წარმოიდგინოს, როგორც ფაილების ნაკრები. ფაილები შეიძლება გაიხსნას მონაცემების წასაკითხად ან

ჩასაწერად. ეს კონცეპტუალურად უფრო მარტივია ვიდრე იმაზე ფიქრი, თუ როგორაა მოწყობილი დისკის ბილიკები და როგორ მუშაობს მისი სატრიალო მოწყობილობა. ანალოგიურად, მარტივი და ცხადი აბსტრაქციით პროგრამისტისგან დამაღლია ყველა არასასურველი წვრილმანი. უფრო მეტიც თანამედროვე ოპერაციულ სისტემებში შეიძლება შეგვექმნას იღუბია ოპერატიული მეხსიერებისა და პროცესორების რაოდენობის ამოუწურაობაზე. ყველაფერ ამას განაგებს ოპერაციულისტებმა.

მაშასადამე, ოპერაციული სისტემა მომხმარებელს შეიძლება წარმოუდგეს როგორც ვირტუალური მანქანა, რომელთანაც საქმის დაჭერა უფრო ადვილია ვიდრე უშეალოდ კომპიუტერის კომპონენტებთან.

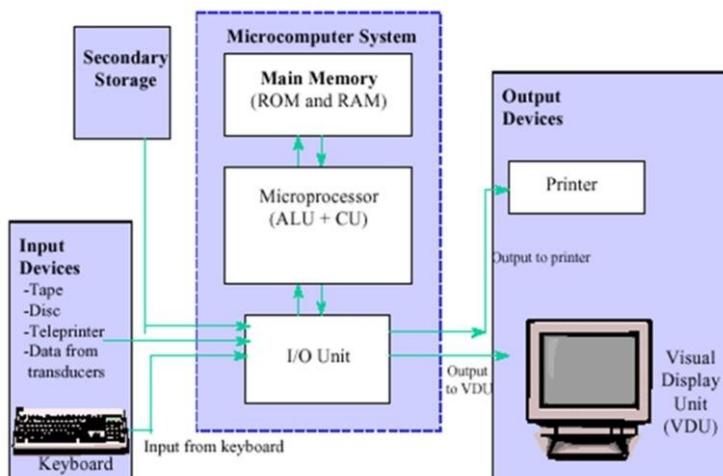
1.1.2. ოპერაციული სისტემა როგორც რესურსების მენეჯერი

როგორც უკვე აღვნიშნეთ ოპერაციული სისტემის ამოცანას წარმოადგენს კომპიუტერის საკმაოდ რთული არქიტექტურის შემადგენელი კომპონენტების ეფექტური მართვა და მათი ოპტიმალური გამოყენება. მაგალითად, წარმოვიდგინოთ რა მოხდება, თუ ერთ კომპიუტერზე მომუშავე რამდენიმე პროგრამა ერთდროულად შეეცდება პრინტერზე საკუთარი შედეგის გამოტანას. ასეთ შემთხვევაში მივიღებთ სხვადასხვა პროგრამის მიერ ქაოტურად დაბეჭდილი სტრიქონებისგან შედგენილ ფურცლებს. ოპერაციული სისტემა ამ ტიპის ქაოსს იცილებს თავიდან მყარ დისკზე პრინტერისთვის გამოყოფილ ადგილას ინფორმაციის ბუფერირების (შენახვის) და ბეჭდვის თანმიმდევრობის ორგანიზების გზით.

მრავალმომხმარებლიანი და მრავალპროგრამული კომპიუტერებისთვის რესურსების მართვის და მათი უსაფრთხოების დაცვის აუცილებლობა კიდევ უფრო ცხადია. შესაბამისად, ოპერაციული სისტემა, როგორც რესურსების მენეჯერი ანხორციელებს პროცესორების, მეხსიერებების და სხვა რესურსების პროგრამებს შორის დალაგებულ და კონტროლირებად განაწილებას.

1.2. კომპიუტერის აპარატული უზრუნველყოფა

ოპერაციული სისტემა მჭიდროდაა დაკავშირებული კომპიუტერის აპარატულ უზრუნველყოფასთან. ის აფართოებს კომპიუტერის მიერ შესასრულებელ ბრძანებათა ნაკრებს და მართავს მის რესურსებს. იმისათვის, რომ ოპერაციული სისტემა ფუნქციონირებდეს ნორმალურად შეფერხების გარეშე საჭიროა პროგრამისტი ერკვეოდეს აპარატურული უზრუნველყოფის მუშაობის პრინციპებში.



ნახ. 1.2. კომპიუტერის კონცეპტუალური წარმოდგენა

ნახ. 1.2-ზე ნაჩვენებია კომპიუტერის კონცეპტუალური წარმოდგენა. პროცესორი, მეხსიერება და შეტანა/გამოტანის მოწყობილობა ერთმანეთთან დაკავშირებულია სისტემური

სალტის (system bus) მეშვეობით, რომლის მეშვეობითაც ისინი ერთმანეთთან ცვლიან მონაცემებს. თანამედროვე კომპიუტერს გააჩნია შედარებით რთული სტრუქტურა და იყენებს რამდენიმე სალტეს.

1.2.1. პროცესორები

პროცესორი ნებისმიერი გამოთვლითი მანქანისათვის წარმოადგენს მნიშვნელოვან კომპონენტს. ის ძირითადი მეხსიერებიდან იღებს ბრძანებებს და ერთიმეორის მიყოლებით ასრულებს მათ. ჩვეულებრივ, პროცესორის მეშაობის ერთი ციკლი მდგომარეობს შემდეგში:

- ძირითადი მეხსიერებიდან შესასრულებლად პირველივე ბრძანებისკითხვა;
- ბრძანების ტიპის და ოპერანდების განსაზღვრა;
- მონაცემების დეკოდირება;
- მიღებული ბრძანების შესრულება.

როგორც ვიცით პროგრამა შედგება ბრძანებების მიმდივრობისგან და, შესაბამისად, პროცესორს უწევს ერთიმეორის მიყოლებით პროგრამის ყველა ბრძანების შესასრულებლად ზემოთ მოყვანილი ეტაპების გამეორება. მიუხედავად იმისა პროგრამა არის თუ არა ამუშავებული პროცესორი არ აჩერებს თავის საქმიანობას. თუ არცერთი პროგრამა არაა ამუშავებული პროცესორი ასრულებს ცარიელ ციკლს.

პროცესორების ყოველ ოჯახს (Intel, AMD, ARM) გააჩნია ბრძანებათა საკუთარი ნაკრები (რომლის შესრულებაც მას შეუძლია). შესაბამისად, ერთი ოჯახის პროცესორებისთვის დაწერილი პროგრამა (ინსტრუქციების ნაკრები) შეუძლებელია შესრულდეს სხვა ოჯახის პროცესორზე. ვინაიდან ბრძანებების და მონაცემების კითხვა (ბრძანების შესრულებასთან შედარებით) ხორციელდება საკმარისად ნელა, ამიტომ ყოველ პროცესორს, ძირითადი ცვლადებისა და შუალედური შედეგების შესანახად, გააჩნია რამდენიმე რეგისტრი (მაღალი სწრაფემედების მქონე მეხსიერება). საერთო დანიშნულების რეგისტრების (რომლებიც ინახება შუალედური შედეგი) დანამატად მრავალ პროცესორს გააჩნია რეგისტრი, რომელიც ხელმისაწვდომია პროგრამისტისთვის. ერთერთ ასეთ რეგისტრს ბრძანებათა მთვლელი ეწოდება. ის შეიცავს ოპერატიულ მეხსიერებაში განთავსებული შემდეგი შესასრულებელი ბრძანების მისამართს. პროცესორისთვის შემდეგი შესასრულებელი ბრძანების მისამართის გადაცემის შემდეგ ხდება ბრძანებათა მთვლელის მნიშვნელობის განახლება და ის უთითებს ახალი შესასრულებელი ბრძანების მისამართს.

კიდევ ერთი სპეციალური რეგისტრი არის ე.ნ. **სტეკის მიმთითებელი**, რომელიც უთითებს, მეხსიერებაში მიმდინარე სტეკის წვეროზე. სტეკი შეიცავს ერთ ფრეიმს (მონაცემთა არე) ყოველი პროცედურისთვის, რომელშიც შესულია და არაა იქიდან გამოსული. პროცედურის სტეკურ ფრეიმში ასევე შედის მისი შესვლის პარამეტრები (ლოკალური და დროებითი ცვლადები), რომელიც არ არიან მოქცეული სტეკში.

ოპერაციულ სისტემას უნდა გააჩნდეს ყველა რეგისტრის მდგომარეობის შესახებ ინფორმაცია. პროცესორის მულტიპლექსირების (სხვადასხვა წყაროდან ინფორმაცია მცირე პორციებად მიღება) დროს ოპერაციულმა სისტემამ შეიძლება ხშირად შეაჩეროს პროგრამის შესრულება, რათა აამუშაოს სხვა პროგრამა. მუშა პროგრამის ყოველი შეჩერებისას ოპერაციული სისტემა ინახავს ყველა რეგისტრის შემცველობას, რათა პროგრამის მუშაობის შემდგომი განახლებისას მოხდეს მისი ამუშავება შეჩერებული ადგილიდან.

1.2.2. მრავალნაკადიანი და მრავალბირთვიანი პროცესორები

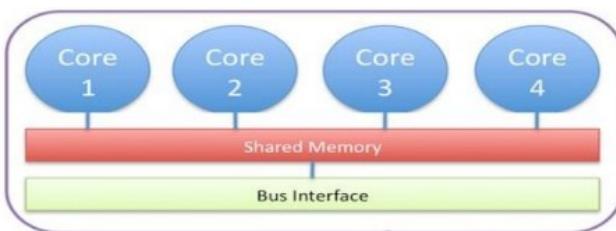
გ.მურის (Gordon Moore, რომელიც არის Intel კომპანიის ერთერთი თანადამფუძნებელი) კანონის თანახმად, ტრანზისტორების რაოდენობა კრისტალში ორმაგდება ყოველ 18 თვეში ერთხელ. ამ კანონს არანაირი კავშირი არ აქვს ფიზიკასთან. ის მიღებულია ემპირიული

დაკვირვების შედეგად: ტრანზისტორების ზომის შემცირებისა და სქემაზე მათი მეტი რაოდენობით განთავსების შესაძლებლობიდან გამომდინარე. ეს კანონი რამდენიმე ათეული წელია რაც მოქმედებს.

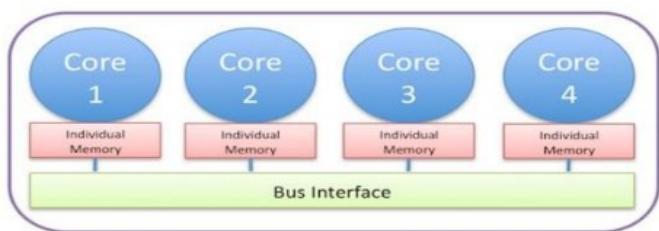
მცირე ზომის არეზე ტრანზისტორების დიდი რაოდენობით განთავსების შესაძლებლობა დამატებით ქმნის ახალ პრობლემებს, რაც დაკავშირებულია ტრანზისტორების მართვასთან. ამ პრობლემის ერთერთ გადაწყვეტას წარმოადგენს მათი განთავსება პროცესორის კრისტალზე დიდი მოცულობის **ქეშ-მეხსიერებად**. ქეშ-მეხსიერების მოცულობის მეტი გაზრდა იძლევა უკუფექტს.

გამოთვლითი მანქანების სწრაფქმედების ამაღლების მიზნით კიდევ ერთ წინგადადგმულ ნაბიჯს წარმოადგენს ფუნქციონალურ ბლოკების დუბლირებასთან ერთად მმართველი ლოგიკის დუბლირება. პროცესორებისთვის დამახაიათებელ ამ თვისებას **მრავალნაკადიანობა** (multi- threading) ეწოდება. ეს ტექნოლოგია პროცესორს საშუალებას აძლევს შეინახოს 2 ნაკადის მდგომარეობა და განახორციელოს მათ შორის გადართვა. მრავალნაკადიანობა არ იძლევა პროგრამების პარალელური დამუშავების შესაძლებლობას. ის იძლევა პროგრამის შესრულების მიმდინარეობის დაჩქარების საშუალებას ნაკადებს შორის გადართვის დროის შემცირების ხარჯე. მრავალნაკადიანობა ზემოქმედებს ოპერაციულ სისტემაზე ვინაიდან, ყოველი ნაკადი მასში წარმოადგენს ცალკე აღებულ პროცესორს.

ა) ბირთვები საერთო მეხსიერებით



ბ) ბირთვები ინდივიდუალური მეხსიერებით



ნახ. 1.3. მრავალბირთვიანი პროცესორების რეალიზაციის სქემა

მრავალნაკადიანი პროცესორების გარდა არსებობს პროცესორები, რომლებიც ერთ კრის- ტალში შეიცავენ 2, 4 ან მეტ სრულფასოვან პროცესორს ანუ ბირთვებს (Core). ასეთი სქემის რეალიზება შესაძლებელია ორი მეთოდით: პროცესორები საერთო მეხსიერებით და პროცესორები ინდივიდუალური მეხსიერებით. ნახ.1.3-ზე ნაჩვენებია ორივე სქემა.

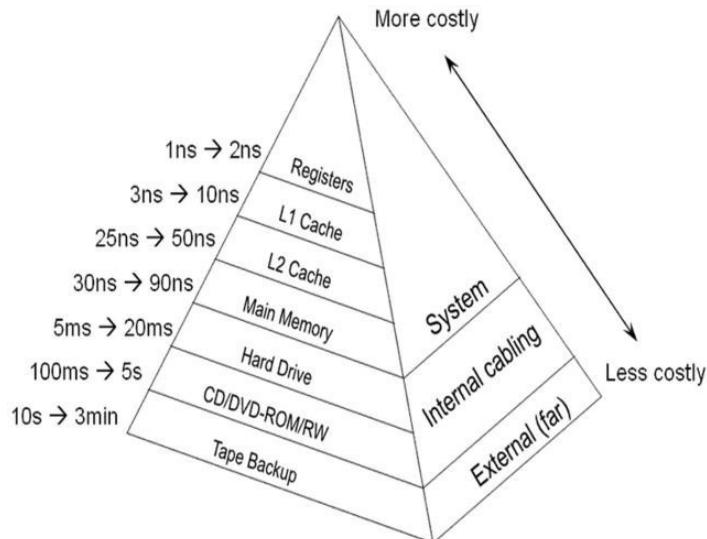
1.2.3. ძირითადი მეხსიერება

კომპიუტერის მეორე ძირითად კომპონენტს წარმოადგენს მეხსიერება (RAM - Random Access Memory. ხშირად მას ფიზიკურ მეხსიერებასაც უწოდებენ). მეხსიერება უნდა იყოს სწრაფი (მუშაობდეს პროცესორის სისწრაფით), საკმარისად დიდი და იაფი ლირებულების. ვერცერთი თანამედროვე ტექნოლოგია ვერ უზრუნველყოფს სამივე მოთხოვნის ერთდროულ დაკმაყოფილებას, ამიტომ გამოიყენება სხვა მიდგომა: მეხსიერება იყოფა ნახ. 1.4-ზე ნაჩვენები იერარქიის სახით. იერარქიის ზედა დონეებს გააჩნია მაღალი სისწრაფე, მცირე მოცულობა და მეხსიერების 1 ბიტზე დიდი ფასი, ვიდრე დაბალი დონის მეხსიერებას.

ზედა დონე შედგება პროცესორის რეგისტრებისგან. რეგისტრები შექმნილია იმავე ტექნოლოგით რაც პროცესორი და სწრაფქმედებითაც არ ჩამოუვარდებიან მას. ისინი იძლევიან 32×32 ბიტი მოცულობის ინფორმაციის შენახვის შესაძლებლობას 32-თანრიგა სისტემისთვის და 64×64 ბიტისა კი - 64-თანრიგა სისტემისთვის.

შემდეგ დონეებზე განთავსებულია ქეშ-მეხსიერებები (L1 და L2). როგორც აღვნიშნეთ პროცესორის რეგისტრების მოცულობა საკმარისად მცირეა (<1 კბ), ამიტომ მასში დიდი რაოდენობით “სიტყვების” განთავსება შეუძლებელია. ქეშ-მეხსიერების გამოყენებით პროგრამის შესრულებისთვის საჭირო “სიტყვები” გარკვეული ნაწილი შეიძლება განთავსებული იქნას ქეშ-მეხსიერებაში. ქეშ-მეხსიერების გამოყენება იმდენად წარმატებული გამოდგა რომ შემუშავებული იქნა მე-3 დონის ქეშ-მეხსიერება (L3), რომლის მოცულობაც რამდენიმეჯერ აღემატება L2 ქეშ-მეხსიერების მოცულობას და თავდაპირველად გამოიყენებოდა მხოლოდ

მძლავრ სერვერებზე. თანამედროვე პროცესორებში ქეშ-მეხსიერებები (L1 და L2) განთავსებულია პროცესორის მიკროსქემაზე, ხოლო L3 კი - მისგარეთ.



ნახ. 1.4. მეხსიერების იერარქია

იერარქიის შემდეგ დონეზე განთავსებულია ოპერატიული მეხსიერება (main memory, ან RAM (random access memory)). ეს არის კომპიუტერის ძირითადი მეხსიერება. ოპერატიულ მეხსიერებაში ხდება ამუშავებული პროგრამების მონაცემების დიდი „პორციების“ განთავსება, საიდანაც პატარ-პატარა პორციებად ეს მონაცემები ხვდება ქეშ-მეხსიერებში. თანამედროვე კომპიუტერებისთვის ოპერატიული მეხსიერების მოცულობა არ წარმოადგენს პრობლემას და ის შესაძლებელია მერყეობდეს რამდენიმე გბ-დან (პერსონალური კომპიუტერი) რამდენიმე ასეულ გბ-მდე (მძლავრ სერვერებში). კომპიუტერის მუშაობის სწრაფემედების ამაღლების მიზნით თანამედროვე ოპერატიულ სისტემებში ოპერატიულ მეხსიერებაში შესაძლებელია ჩატვირთული იყოს არა მხოლოდ ამუშავებული პროგრამის მონაცემები არამედ, ისეთი პროგრამის მონაცემები, რომელიც პერსონალური კომპიუტერის მუშაობის გარკვეულ პერიოდში (რამდენიმე საათი, დღე ან თვე) საერთოდ არ ამუშავებულა. ოპერატიულ მეხსიერებაში ასეთი მონაცემების არსებობა მისი დიდი მოცულობის გამო არ აზრალებს ოპერატიული სისტემის სწრაფემედებას და ხელს უწყობს რაიმე პროგრამის ამუშავების აუცილებლობისას მის სწრაფ ჩატვირთვას.

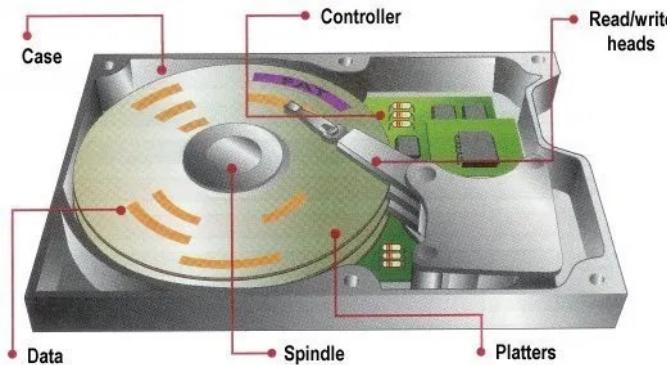
იერარქიის შემდეგ დონეზე განთავსებულია დამხმარე შესანახი მეხსიერება. ამ ტიპის მეხსიერება კომპიუტერში გამოიყენება სხვადასხვა სახის მონაცემების შესანახად. დამხმარე მეხსიერებებს წარმოადგენს იერარქიის შემდეგ დონეზე განთავსებული მეხსიერებებიც. ვინაიდან კომპიუტერში ამ ტიპის მოწყობილობების გამოყენების აუცილებლობა არ არსებობს, ამიტომ მათ მიაკუთვნებენ ინფორმაციის შესახვის გარე მოწყობილობებს.

1.2.4. მყარი დისკი

როგორც უკვე აღვნიშნეთ მეხსიერების იერარქიაში ოპერატიული მეხსიერების შემდეგ დონეზე განთავსებულია დამხმარე შესანახი მეხსიერება - **მყარი დისკი** (hard disk drive - HDD). მყარი დისკის ინფორმაციის შესანახი 1 ბიტის ღირებულება გაცილებით ნაკლებია ოპერატიული მეხსიერებაში ინფორმაციის შესანახი 1 ბიტის ღირებულებაზე. მყარი დისკის მოცულობა რამდენიმე ასეულკვერ ან ათასეულკვერ აღემატება ოპერატიული მეხსიერების მოცულობას, ხოლო ინფორმაციაზე წვდომის სისწრაფე კი ოპერატიულ მეხსიერებასთან მიმართებაში რამდენიმე ათეულკვერ დაბალია. ნახ. 1.5-ზე წარმოდგენილი მყარი დისკის კონსტრუქციის სქემა.

მყარი დისკი შედგება მეტალის რამდენიმე დისკოსგან, რომელიც საკუთარი ღერძის გარშემო აკეთებს 5400, 7200 ან 10800 ბრუნს წუთში. დისკოს ორივე მხარე გამოიყენება ინფორმაციის შესანახად. დისკოს ზედაპირი იყოფა ბილიკებად, ცილინდრებად და

სექტორებად. დისკოდან ინფორმაციის კითხვა ხდება წამკითხავი თავაკის გამოყენებით, რომელსაც უწევს პოზიციონირება დისკობე.



ნახ. 1.5. HDD დისკის კონსტრუქციის სქემა



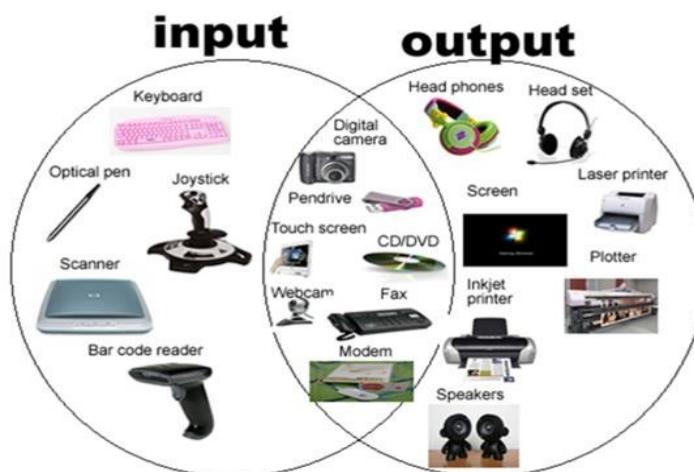
ნახ. 1.6. SSD დისკი

შევნიშნოთ, ბოლო ათი წლის მანძილზე გამოჩნდა და აქტუალურობას არ კარგავს ე.წ. SSD (Solid State Disk) დისკი (ნახ.1.6), რომლებიც სწრაფებით უსწრებენ HDD დისკებს, მაგრამ ჩამოვარდებიან RAM-ს. ერთის მხრივ, რადგანაც SSD დისკი აგებულია ტრანზისტორების გამოყენებით, ამიტომ მონაცემების კითხვისას არ მოითხოვს პოზიციონირების დროს, რაც საკმარისად სწრაფების ხდის მას. მეორე მხრივ, ტრანზისტორებზე დადებული შეზღუდვა საგრძნობლად ამცირებს SSD დისკების სიცოცხლის ხანგრძლივობას. სახელდობრ, მონაცემების შენახვის ყოველ ბიტზე შეზღუდულია კითხვის და ჩაწერის ოპერაციების რაოდენობა - 100000 ოპერაცია.

თანამედროვე კომპიუტერულ სისტემებში მონაცემების შესანახი დამატებითი მეხსიერების როლში იყენებენ ჰიბრიდულ მიდგომას, რომელიც გულისხმობს SSD და HDD დისკების ერთდროულ გამოყენებას. ჰიბრიდული მიდგომის შემთხვევაში SSD დისკები ძირითადად გამოიყენება ოპერაციული სისტემის ინსტალირებისთვის, როთაც ზრდიან ოპერაციული სისტემის სწრაფებისას, ხოლო HDD დისკები კი - გამოიყენება მონაცემების გრძელვადიანი შენახვისთვის.

1.2.5. შეტანა/გამოტანის მოწყობილობა

პროცესორი და მეხსიერება არ არის ერთადერთი რესურსი, რომელსაც მართავს ოპერაციული სისტემა. ის ასევე ურთიერთქმედებს ინფორმაციის შეტანა/გამოტანის მოწყობილობებთან. ნახ. 1.7-ზე ნაჩვენებია შეტანა/გამოტანის მოწყობილობების მრავალფეროვნება.



ნახ. 1.7. შეტანა/გამოტანის მოწყობილობების მრავალფეროვნება

შეტანა/გამოტანის მოწყობილობა შედგება ორი კომპონენტისგან: თვით მოწყობილობა და კონტროლერი. კონტროლერი წარმოადგენს მიკროსქემას ან მიკროსქემების ნაკრებს,

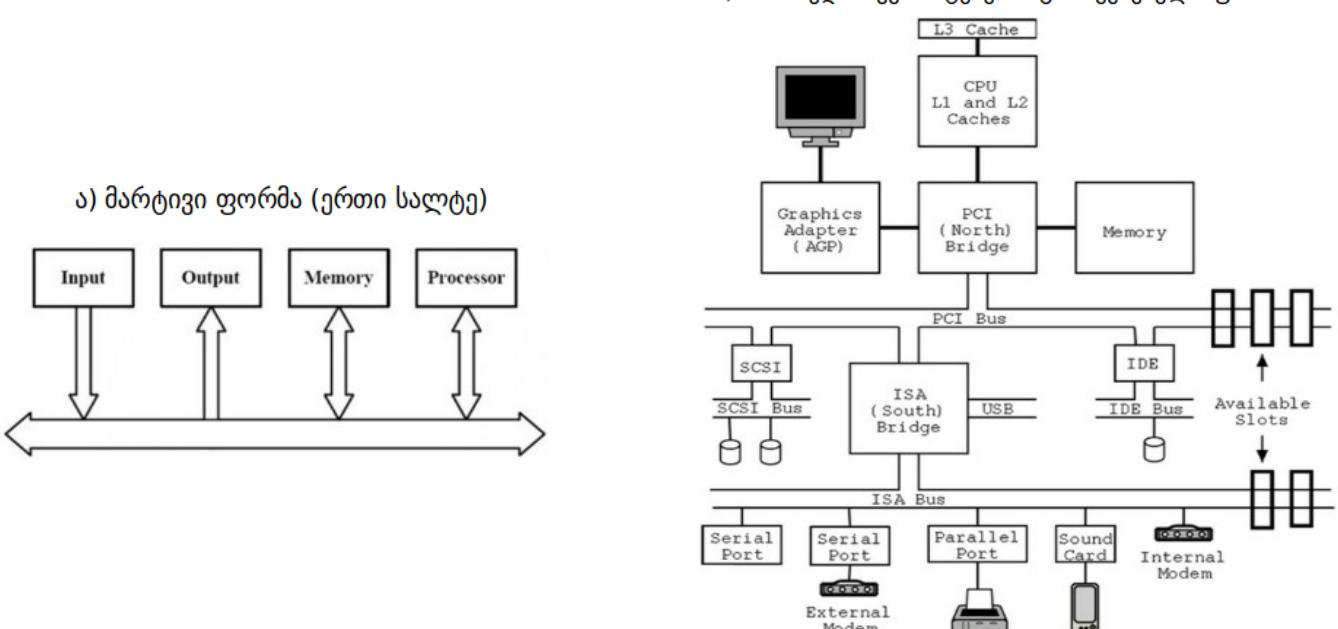
რომლებიც ფიზიკურ დონეზე მართავენ შესაბამის მოწყობილობას. ისინი ოპერაციული სისტემისგან ღებულობენ ბრძანებებს გარკვეული მოქმედებისშესასრულებლად და ასრულებენ მას. სხვადასხვა მოწყობილობების კონტროლერები არიან რთული აგებულების და განსხვავ-დებიან ერთიმეორისგან. ასევე, განსხვავდება მათი ნორმალური ფუნქციონირებისთვის საჭირო პროგრამული უზრუნველყოფები. პროგრამას, რომელიც გამოიყენება კონკრეტული მოწყობილობის ნორმალური ფუნქციონირების უზრუნველსაყოფად, ეწოდება **მოწყობილობის დრაივერი**. მოწყობილობების მწარმოებლები საკუთარი პროდუქტის შემუშავებასთან ერთად ქმნიან მის პროგრამულ უზრუნველყოფას სხვადასხვა ოპერაციული სისტემაში შესაბამისი მოწყობილობის სრულფასოვანი ფუნქციონირების უზრუნველყოფის მიზნით.

მეორე კომპონენტს - მოწყობილობას, გააჩნია მარტივი ინტერფეისი, ვინაიდან საჭიროა ის პასუხობდეს სტანდარტებს. სტანდარტის დაცვა წარმოადგენს აუცილებლობას იმის გამო, რომ უზრუნველყოფილი იყოს სხვადასხვა მოწყობილობის კონტროლერების შეთანხმებული მუშაობა. ოპერაციული სისტემა ხედავს კონტროლერის ინტერფეისს და მისი გამოყენებით მართავს შესაბამის მოწყობილობას.

1.2.6. სალტე

როგორც უკვე არაერთხელ აღვნიშნეთ, კომპიუტერული სისტემა შედგება მრავალი ფიზიკური კომპონენტისგან. ფიზიკურ კომპონენტებს შორის სრულფასოვანი კომუნიკაციის უზრუნველსაყოფად გამოიყენება ე.წ. **სალტე** (bus). სალტე ეს არის კომპიუტერის დედაბლატაზე განთავსებული წვრილი (უხილავი) ხაზების ერთობლიობა, რომელზეც ურთიერთდებ კომპონენტებს შორის გადაიცემა მონაცემების შესაბამისი ელექტრული მუხტები. სალტეში გამოყენებული ხაზების რაოდენობა განსაზღვრავს არქიტექტურის თანრიგიანობას, რაც ნიშნავს, რომ 32-თანრიგა არქიტექტურის შემთხვევაში ხაზების რაოდენობა არის 32, ხოლო 64-თანრიგა არქიტექტურის შემთხვევაში კი არის 64. არქიტექტურის თანრიგიანობა თავის მხრივ განსაზღვრავს ოპერაციული სისტემის თანრიგიანობას, რომელიც შესაძლებელია დაყენდეს შესაბამის არქიტექტურაზე. მაგალითად, 32-თანრიგა არქიტექტურის შემთხვევაში ოპერაციული სისტემა აუცილებლად უნდა იყოს 32- თანრიგა, ხოლო 64-ის შემთხვევაში კი შესაძლებელია გამოყენებული იქნას როგორც 64-თანრიგა (სასურველია) ისე, 32-თანრიგა ოპერაციული სისტემა.

ბ) თანამედროვე სისტემებში გამოყენებული ფორმა



ნახ. 1.8. კომპიუტერში სალტეების ორგანიზაციის სქემა

გამოთვლით მანქანაში შესაძლებელია გვხვდებოდეს რამდენიმე განსხვავებული სალტე როგორიცაა PCI, SCSI, IDE, ISA და სხვა.

ნახ. 1.8 ა)-ბე ნაჩვენებია კომპიუტერში გამოყენებული სალტეების ორგანიზაციის სქემა, რომელიც მრავალი წლის განმავლობაში გამოიყენებოდა. პროცესორის და მეხსიერების სწრაფქმედების ზრდასთან ერთად აუცილებელი გახდა შეცვლილიყო მისი ორგანიზაცია. ნახ.18 ბ) ნაჩვენებია თანამედროვე კომპიუტერებში გამოყენებული სალტეების ორგანიზაციის რთული ფორმა. ნაჩვენებ სისტემაში გამოყენებულია 6 განსხვავებული სალტე, რომელთაგან თითოეულს გააჩნია მონაცემთა გადაცემის საკუთარი სისწრაფე და დანიშნულება.

1.3. ოპერაციული სისტემების ოჯახი

ოპერაციული სისტემების განვითარების ისტორია ითვლის ნახევარ საუკუნეზე მეტს. მთელი ამ ხნის განმავლობაში შეიქმნა მრავალი ოპერაციული სისტემა, მაგრამ მათგან უმეტესმა ვერ მოიპოვა პოპულარობა და დღეისთვის წარმოადგენს ისტორიის საკუთრებას. დავახასიათოთ ზოგიერთი დღეისთვის ფართოდ გავრცელებული მოწყობილობების ოპერაციული სისტემები.

შეინფრეიმების ოპერაციული სისტემები. მეინფრეიმი წარმოადგენს ბაზარზე გამოჩენილ პირველ გამოთვლით მანქანას, რომელსაც გააჩნდა დიდი ზომა და რიგ შემთხვევებში იკავებდა მთელ შენობას. ის ძირითადად განთავსებული იყო დიდ გამოთ- ვლით ცენტრებში. დღეისთვის ამ ტიპის მოწყობილობებად გვევლინება სუპერკომპიუტერები, რომელთა შეძენა ძვირი ღირებულების გამო შეუძლია მხოლოდ მსხვილ ორგანიზაციებს.

მეინფრეიმების ოპერაციული სისტემები უპირატესად ორიენტირებულია დიდი რაოდენობის ამოცანების დამუშავებაზე, რომელთა ერთობლიობა საჭიროებს კოლოსალური რაოდენობის შეტანა/გამოტანის ოპერაციებს. მათ მიერ განხორციელებული სამუშაო იყოფა სამ ტიპად: პაკეტური დამუშავება, ტრანზაქციების დამუშავება და დროის გაყოფის რეჟიმში მუშაობა. ასეთი ოპერაციული სისტემის მაგალითს წარმოადგენს OS/390.

სერვერული ოპერაციული სისტემები. სერვერი არის მძლავრი შესაძლებლობების მქონე კომპიუტერი, რომელიც დამატებით შეიძლება აღჭურვილი იყოს რამდენიმე პროცესორით, მეხსიერების მოდულებით და მყარი დისკებით. ასეთი კომპიუტერების ოპერაციული სისტემის დანიშნულებას წარმოადგენს ქსელის მეშვეობით დაკავშირებული კომპიუტერების მომსახურეობა. ისინი მომხმარებლებს სთავაზობენ აპარატურულ და პროგრამულ რესურსებზე წვდომის შესაძლებლობას. სერვერებს მომხმარებელთათვის შეუძლიათ ბეჭდვის, ფაილების შენახვის ან ვებ-სერვისების შეთავაზება. ასეთი ოპერაციული სისტემების ტიპიურ მაგალითს წარმოადგენს Solaris, FreeBSD, Linux და Windows ოპერაციული სისტემების სერვერული ვერსიები.

მრავლპროცესორული ოპერაციული სისტემები. გამოთვლითი მანქანების განვითარების მთელს ეტაპზე მეტი სიმძლავრის მქონე მანქანების მისაღებად შემუშავებული იყო მრავალი ტექნოლოგია. მათ შორის ზოგიერთი გულისხმობდა რამდენიმე პროცესორის გამოყენებას. რათემაუნდა ერთ მიკროსქემაზე რამდენიმე პროცესორის განთავსება გარკვეულ სირთულეებთანაა დაკავშირებული. ამ სირთულეს ემატება ისიც, რომ ამ შემთხვევაში საჭიროა ყველა პროცესორის მუშაობის კონტროლი, მათ მიერ რესურსების დაკავების კონტროლი და სხვა მრავალი წვრილმანის გათვალისწინება. ასეთი კომპიუტერებისთვის შეიქმნა სპეციალური ოპერაციული სისტემა, რომელიც აღჭურვილი იყო კავშირებისა და სინქრონიზაციის დამატებითი საშუალებებით.

რათემაუნდა ერთ პლატაზე რამდენიმე პროცესორის განთავსება და დამატებითი კავშირის ხაზების გათვალისწინება გამოიწვევს სისტემური პლატის ზომის გაზრდას. დიდი რაოდენობის პროცესორების შემთხვევაში ამ ამოცანის გადაწყვეტა შეიძლება იყოს წარმოუდგენელი. კიდევ ერთი მიდგომა, რომელიც შემოთავაზებული იყო მძლავრი მანქანის მისაღებად მდგომარეობს პროცესორის კრისტალში დამატებით ბირთვების (პროცესორების) ჩამატებაში,

რომლებიც მუშაობენ სრულფასოვანი პროცესორის მსგავსად. ბოლო პერიოდში ფართოდ გამოიყენება ეს ტექნოლოგია. ამ შემთხვევაში პლატაზე დამატებითი ხაზების გაყვანის აუცილებლობა არ არსებობს. ასეთი კომპიუტერის ოპერაციული სისტემა ძალიან წააგავს მრავალ- პროცესორული მანქანების ოპერაციულ სისტემას და საჭიროებს სინქრონიზაციის მექანიზმს.

პერსონალური კომპიუტერების ოპერაციული სისტემები.

მომხმარებელი ყველაზე მეტად იცნობს ამ ტიპის ოპერაციულ სისტემებს. ის შექმნილია მრავალპროგრამულ რეჟიმში მუშაობისთვის. ოპერაციული სისტემის ძირითად ამოცანას წარმოადგენს ინდივიდუალური მომხმარებლის ხარისხიანი მომსახურეობა. ისინი ძირითადად გამოიყენება ტექსტთან სამუშაოდ, ელექტრული ცხრილების შესაქმნელად, გლობალურ ქსელზე წვდომისთვის, გართობისთვის და ა.შ. ასეთი ოპერაციული სისტემების მაგალითის წარმოადგენს Linux, Windows და Macintosh ოპერაციული სისტემები.

რეალური დროის სისტემები. ამ ტიპის ოპერაციული სისტემები გამოიყენება სხვადასხვა ტექნიკური ობიექტების ან ტექნოლოგიური პროცესების სამართავად. ასეთი სისტემები ხასიათდება გარე მოვლენებზე უკიდურესად მისაღები დროითი რეაქციით, რომლის განმავლობაშიც შეიძლება შესრულებული იყოს ობიექტების მმართველი პროგრამა.

სისტემამ უნდა დაამუშაოს შემოსული მონაცემები იმაზე სწრაფად ვიდრე ისინი შემოედინება. ამასთან, შესაძლებელია მონაცემების შემოღინება ხდებოდეს ერთდროულად რამდენიმე წყაროდან. ასეთი მკაცრი შეზღუდვა აისახება რეალური დროის სისტემის არქიტექტურაზე, მაგალითად, მასში შეიძლება არ არსებობდეს ვირტუალური მეხსიერება, რომლის მხარდაჭერამაც პროგრამის შესრულების პროცესში შეიძლება გამოიწვიოს მისი შენელება, რისი წინასწარმეტყველებაც შეუძლებელია.

გარდა ზემოთ ჩამოთვლილი ოპერაციული სისტემებისა, ასევე, არსებობს სხვა ოპერაციული სისტემები, მაგალითად, როგორიცაა „ჭიბის პერსონალური კომპიუტერის“ (PDA - Personal Digital Assistant) ან თანამედროვე მობილური აპარატების (სმარტფონების, ტაბლეტების) ოპერაციული სისტემები, ინტეგრირებული ოპერაციული სისტემები (ტელევიზორების, მანქანების, MP3- პლეერების და ა.შ.), სენსორული კვანძების ოპერაციული სისტემები (დაცვის სენსორული სისტემები) და ა.შ.

1.4. ოპერაციულ სისტემასთან დაკავშირებული ძირითადი ცნებები

მიუხედავად იმისა, რომ ოპერაციული სისტემები შეიძლება შექმნილი იყოს სხვადასხვა მწარმოებლის მიერ და გამოიყენებოდეს სხვადასხვა ტიპის მოწყობილობებში, მათ შორის მაინც არსებობს მსგავსება და საერთო ცნებები, რომლებიც გამოიყენება თითოეულ მათგანში, და აბსტრაქცია, როგორიცაა პროცესი, მისამართების სივრცე, ფაილები, უსაფრთხოება, სისტემური გამოძახება, წყვეტა, განსაკუთრებული შემთხვევა და ა.შ.

პროცესი. ოპერაციულ სისტემაში მნიშვნელოვან როლს თამაშობს პროცესის ცნება. პროცესი თავისი არსით წარმოადგენს პროგრამას შესრულების მომენტში. ყოველ პროცესთან დაკავშირებულია მისიმისამართების სივრცე (მეხსიერების უკრედების მისამარ-თების სიმრავლე), რომლიდანაც ის კითხულობს და რომელშიც წერს მონაცემებს. მისამართების სივრცე შეიცავს შესრულებად პროგრამას, მის მონაცემებსა და სტეკს. გარდა ამისა, ყოველ პროცესთან დაკავშირებულია რესურსების გარკვეული ნაკრები, მათ შორის რეგისტრები (ბრძანებათა მთვლელი და სტეკის მიმთითებელი), გახსნილი ფაილების ცხრილი, დაკავშირებული პროცესების ცხრილი და სხვა მრავალი ინფორმაცია, რომელიც საჭიროა პროცესის შესარულებლად.

ოპერაციულ სისტემას, მასში რამდენიმე პროცესის ერთდროულად ამუშავების შემთხვევაში, უწევს მიიღოს გადაწყვეტილება გარკვეული დროით შეაჩეროს ამა თუ იმ პროცესის საქმიანობა, ხოლო მოგვიანებით განაახლოს ის შეჩერებული მდგომარეობიდან. რაც ნიშნავს, რომ ოპერაციულმა სისტემამ პროცესზე ინფორმაცია უნდა შეინახოს ცხადი ფორმით. მაგალითად, პროცესს წასაკითხად ერთდროულად შეიძლება გახსნილი ჰქონდეს რამდენიმე ფაილი. თითოეულ ამ ფაილთან

დაკავშირებულია მიმთითებელი მიმდინარე მდგომარეობაზე (მომდევნო წასაკითხი ბაიტის ნომერი). პროცესის შეჩერებისას ყველა ეს მიმთითებელი უნდა იქნას შენახული რათა პროცესის ხელახლა ამჟამავებისას ინფორმაციის კითხვა განახლდეს საჭირო ადგილიდან. მრავალ ოპერაციულ სისტემაში ყველა პროცესზე ინფორმაცია (პროცესის მისამართების სირვეები არსებული მონაცემების გამოკლებით) ინახება ე.წ. **პროცესების ცხრილში** და ოპერაციულ სისტემაში არსებული ყოველი პროცესისთვის ის წარმოადგენს სტრუქტურათა მასივს.

პროცესის მისამართების სივრცე. კომპიუტერში ყოველი შესრულებადი პროგრამის შესანახად გამოიყოფა ოპერატიული მეხსიერების გარკვეული ნაწილი. ყველაზე მარტივ ოპერაციულ სისტემებში მეხსიერებაში იტვირთება მხოლოდ ერთი პროგრამა. ასეთ ოპერაციულ სისტემებში ახალი პროგრამის შესასრულებლად საჭიროა მეხსიერებიდან მიმდინარე პროგრამის ამოშლა (ამოტვირთვა) და შემდგომი პროგრამის ჩატვირთვა.

უფრო დახვეწილი ოპერაციული სისტემები იძლევიან მეხსიერებაში ერთდროულად რამდენიმე პროგრამის განთავსების შესაძლებლობას. ასეთ ოპერაციულ სისტემებში იმისთვის, რომ პროგრამებმა მუშაობაში ხელი არ შეეშალონ ერთიმეორეს და ოპერაციულ სისტემას, საჭიროა დაცვის მექანიზმის არსებობა (ანუ თითოეული მათგანისთვის მკაცრად უნდა იყოს განსაზღვრული გამოყოფილი მეხსიერების შუალედები).

კომპიუტერის ოპერატიული მეხსიერების მართვა და დაცვა წარმოადგენს მნიშვნელოვან საკითხს. მეხსიერებასთან დაკავშირებულ მეორე არანაკლებ მნიშვნელოვან საკითხს წარმოადგენს პროცესების მისამართების სივრცის მართვა. ოპერაციულ სისტემაში ჩვეულებრივ ყოველ პროცესს გამოიყოფა მისამართების უჯრედების უწყვეტი მიმდევრობა. უმარტივეს შემთხვევაში მისამართების სივრცის მაქსიმალური მოცულობა არ აღემატება ოპერატიული მეხსიერების მოცულობას.

კომპიუტერებში ძირითადად გამოიყენება 32- ან 64-თანრიგა დამისამართების სქემა, რომელიც შესაბამისად იძლევა 2^{32} (4GB)და 2^{64} (16EB¹²) ბაიტი მოცულობის მისამართების სივრცის ქონის შესაძლებლობას.

ბუნებრივია, ისმის კითხვა რა მოხდება იმ შემთხვევაში, როდესაც პროცესი საჭიროებს იმაზე მეტი მოცულობის მეხსიერებას ვიდრე კომპიუტერის ფიზიკური მეხსიერება? თავიდან მსგავს სიტუაციებში კომპიუტერი დასახულ ამოცანას ვერ უმკლავდებოდა. მოვიანებით გამოჩენდა ვირტუალური მეხსიერების ტექნოლოგია. ამ ტექნოლოგიით ოპერაციული სისტემა მისამართების სივრცის ერთ ნაწილს ინახავს ოპერატიულ მეხსიერებაში, ხოლო მეორე ნაწილს მყარ დისკზე და საჭიროებისამებრ უცვლის მათ ადგილებს. ოპერაციული სისტემა ვირტუალური მისამართების ნაკრების სახით ქმნის მისამართების სივრცის აბსტრაქციას, რომელსაც მიმართავს პროცესი. ვირტუალური მისამართების სივრცე კომპიუტერის ფიზიკური მეხსიერებისგან დამოუკიდებელია და მოცულობითაც შესაძლებელია აქარბებდეს მას.

ფაილი. მეორე მნიშვნელოვანი აბსტრაქცია, რომელიც ყველა ოპერაციულ სისტემაში არის მხარდაჭერილი, არის ფაილური სისტემა. როგორც აღვნიშნეთ ოპერაციული სისტემის ამოცანას წარმოადგენს მომხმარებლისგან დამალოს მყარი დისკის და შეტანა/გამოტანის სხვა მოწყობილობების მუშაობის სპეციფიკა და პროგრამისტს შესთავაზოს მოწყობილობებისგან დამოუკიდებელი, მოხერხებული და გასაგები აბსტრაქტული მოდელი - **ფაილი**, როგორც მყარ დისკზე განთავსებული მონაცემების გარკვეული ერთობლივობა. პროგრამისტისთვის ამ მოდელის შემოღებით მარტივდება მასზე მიმართვა და ზემოქმედება (კითხვა, ჩაწერა, წაშლა).

ოპერაციული სისტემა ფაილების ნაკრების ერთ ჯგუფში გასაერთიანებლად იყენებს **კატალოგის** (ფოლდერი) აბსტრაქციას როგორც მეთოდს. კატალოგის ელემენტი შეიძლება იყოს როგორც ფაილი, ასევე კატალოგი. ყოველ კატალოგს შეუძლია გააჩნდეს საკუთარი ქვეკატალოგი. ასეთნაირად ორგანიზებული სქემა გვაძლევს ფაილურ სისტემის იერარქიულ სტრუქტურას. ოპერაციულ სისტემაში ფაილების და კატალოგების შესაქმნელად, წასაშლელად და მათზე დასაშვები სხვა ოპერაციების განსახორციელებლად გამოიყენება სისტემური გამოძახებები.

მონაცემების შეტანა/გამოტანა. როგორც უკვე აღვნიშნეთ, კომპიუტერი აღჭურვილია შეტანა/გამოტანის სხვადასხვა ფიზიკური მოწყობილობებით (ნახ.1.7). შეტანა/გამოტანის მოწყობილობათა რიცხვს შეიძლება მივაკუთვნოთ კლავიატურა, მონიტორი, პრინტერი და ა.შ. ოპერაციულ სისტემაში ასეთი მოწყობილობების სამართავად გამოიყენება პროგრამათა ქვესისტემა. პროგრამათა ერთი ნაწილი არა დამოკიდებული შეტანა/გამოტანის კონკრეტული მოწყობილობის ტიპები და მათი გამოყენება შესაძლებელია ამ ტიპის უმეტეს მოწყობილობასთან, მეორე ნაწილი კი პირიქით - მაგალითად, მოწყობილობათა დრაივერები გამოიყენება შეტანა/გამოტანის შესაბამის მოწყობილობებთან სამუშაოდ.

უსაფრთხოება. კომპიუტერში ინახება დიდ მოცულობის მნიშვნელოვანი ინფორმაცია (ელექტრონული წერილი, ბიზნეს-გეგმა, საგადასახადო დეკლარაცია და ა.შ.), რომელთა დაცვასა და კონფიდენციალობის შენახვას საჭიროებს მომხმარებელი. უსაფრთხოების მართვა წარმოადგენს ოპერაციული სისტემის კიდევ ერთ ამოცანას. მაგალითად, მან უნდა მისცეს ფაილზე წვდომის შესაძლებლობა მხოლოდ შესაბამისი უფლების მქონე მომხმარებელს.

სისტემის უსაფრთხოების მექანიზმის მუშაობის მაგალითი განვიხილოთ UNIX ოპერაციული სისტემის მაგალითზე. UNIX-ში ფაილს დაცვისათვის ენიჭება 9-თანრიგა კოდი. ეს კოდი შედგება 3 ბიტიანი ველებისაგან. რომელთაგან პირველი სამეული განეკუთვნება მომხმარებელს, მეორე სამეული მომხმარებლის ჯგუფს და მესამე სამეული კი - სხვა დანარჩენ მომხმარებლებს. თითოეულ ველში შესაბამისად არსებობს კითხვის, რედაქტირების და შესრულების უფლების აღმნიშვნელი ბიტი. ამ ბიტებს ეწოდებათ **rwx-ბიტები** (read, write, execute). მაგალითად, კოდი rwxr-x-- აღნიშნავს, რომ ფაილის მფლობელი სარგებლობს ამ ფაილის კითხვის, რედაქტირების და შესრულების უფლებით; მფლობელის ჯგუფი სარგებლობს მხოლოდ ფაილის კითხვის და შესრულების უფლებით; სხვა დანარჩენი მომხმარებელი კი მხოლოდ შესრულების უფლებით. კატალოგისათვის შესრულების უფლება (x) ნიშნავს ძებნის ოპერაციის განხორციელების შესაძლებლობას. დაშვების კოდში რომელიმე სიმბოლოს არარსებობა ნიშნავს მომხმარებლის შეზღუდვას შესაბამისი უფლებით.

ფაილების დაცვის გარდა საჭიროა ოპერაციული სისტემის დაცვა არასასურველი ცვლილებებისგან როგორც მომხმარებლის ისე, ვირუსის მატარებელი პროგრამების მხრიდან.

სისტემური გამოძახება (system calls). როგორც უკვე აღვნიშნეთ კომპიუტერს გააჩნია მუშაობის ორი რეჟიმი: ბირთვი და მომხმარებელი. ბირთვის რეჟიმში შესაძლებელია პრივილეგირებული ბრძანებების შესრულება. აქ ბრძანების შესასრულებლად ხდება პირდაპირ აპარატურაზე მიმართვა. მომხმარებლის რეჟიმში ხორციელდება ნაკლებად პრივილეგირებული ბრძანებების შესრულება. როდესაც მომხმარებლის პროგრამა საჭიროებს გარკვეული პრივილეგირებული ბრძანების შესრულებას ის მიმართავს ოპერაციულ სისტემას შესაბამისი ბრძანებით, რომელსაც სისტემური გამოძახება ეწოდება, და პრივილეგირებულ რეჟიმში ასრულებს საჭირო მოქმედებებს. ყოველ ოპერაციულ სისტემაში მხარდაჭერილია სისტემური გამოძახებების საკუთარი ნაკრები. სისტემური გამოძახებები სამომხმარებლო პროგრამებისთვის წარმოადგენენ ბირთვის რეჟიმში შესვლის წერტილებს. სხვადასხვა ოპერაციულ სისტემებში სისტემური გამოძახებების სხვადასხვა სახელებს ეძახიან, მაგალითად, IBM-ის ოპერაციული სისტემებისთვის ესაა **ექსტრაკოდი**, Unix-მსგავსი ოპერაციული სისტემებისთვის კი სახელი უცვლელია - სისტემური გამოძახება.

სისტემური გამოძახება ეს არის ინტერფეისი ოპერაციულ სისტემასა და გამოყენებით პროგრამას შორის. მათ შეუძლიათ შექმნან, წაშალონ და გამოიყენონ სხვადასხვა ობიექტები. გამოყენებითი პროგრამა სისტემური გამოძახების მეშვეობით მიმართავს ოპერაციულ სისტემას გარკვეული სერვისების მისაღებად. არსებობენ პროცედურათა ბიბლიოთეკები, რომლებიც კომპიუტერის რეგისტრებში ტვირთავენ გარკვეულ პარამეტრებს და ახდენენ პროცესორის წყვეტას, რის შემდეგაც მართვა გადაეცემა ოპერაციული სისტემის ბირთვის შესაბამის ნაწილს, რომელიც ახდენს მიმდინარე გამოძახების დამუშავებას. ასეთი ბიბლიოთეკების არსებობის მიზანს წარმოადგენს სისტემური გამოძახების გადაქცევა პროგრამაში ჩვეულებრივი ფუნქციის გამოძახების მსგავს გამოძახებად.

ძირითადი განსხვავება მდგომარეობს იმაში, რომ სისტემური გამოძახებისას ამოცანა გადადის პრივილეგირებულ ანუ ბირთვის (kernel mode) რეჟიმში. ამიტომ სისტემურ გამოძახებას, აპარატული წყვეტისგან განსხვავებით, ხშირად **პროგრამულ წყვეტას** უწოდებენ.

წყვეტა (interrupt). ეს არის (პროცესორთან მიმართებაში) გარე მოწყობილობის მიერ გენერირებული მოვლენა. აპარატურა აპარატული წყვეტის მეშვეობით ახდენს ცენტრალური პროცესორის ინფორმირებას, რომ მოხდა გარკვეული მოვლენა და საჭიროა დაუყოვნებელი რეაგირება (მაგალითად, მომხმარებელმა დააჭირა ღილაკზე) ან, აცნობებს შეტანა/გამო-ტანის ასინქრონული ოპერაციის დასრულების შესახებ (მაგალითად, დასრულდა დისკიდან მონაცემების კითხვა). აპარატული წყვეტის მნიშვნელოვანი ტიპია - ტაიმერიდან წყვეტა, რომელიც გენერირდება პერიოდულად გარკვეული დროითი შუალედის გასვლის შემდეგ. ტაიმერიდან წყვეტა ოპერაციული სისტემის მიერ გამოიყენება პროცესურის დაგეგმვისას. აპარატული წყვეტის ყოველ ტიპს გააჩნია საკუთარი ნომერი, რომელიც ცალსახად განსაზღვრავს წყვეტის წყაროს. აპარატული წყვეტა ეს არის ასინქრონული მოვლენა ანუ, ის წარმოიშობა იმისგან დამოუკიდებლად დროის მოცემულ მომენტში პროცესორის მიერ კოდის რომელი ფრაგმენტი სრულდება.

განსაკუთრებული შემთხვევა (exception) არის პროგრამის მიერ ბრძანების შესრულების მცდელობის შედეგად წარმოიშობილი მოვლენა, რომელიც გარკვეული მიზებების გამო შეუძლებელია შესრულდეს. ასეთი ბრძანების მაგალითი შეიძლება იყოს არასაკმარისი პრივილეგიის ქონისას შეზღუდულ რესურსზე წვდომის მცდელობა ან მეხსიერების არარსე-ბულ გვერდზე მიმართვა. განსაკუთრებული შემთხვევები შეიძლება დაიყოს ორ ნაწილად: გამოსწორებადი და გამოუსწორებელი. გამოსწორებად განსაკუთრებულ შემთხვევას მიეკუთვნება ისეთი განსაკუთრებული შემთხვევა, როგორიცაა დროის მიმდინარე მომენტისთვის ოპერაციულ მეხსიერებაში საჭირო ინფორმაციის არარსებობა. გამოსწორებადი განსაკუთრებული სიტუაციის გამომწვევი მიზების აღმოფხვრის შემდეგ პროგრამას შეუძლია გააგრძელოს შესრულება.

გამოუსწორებელი განსაკუთრებული შემთხვევა ძირითადად წარმოიშობა პროგრამული შეცდომებისას (მაგალითად, ნულზე გაყოფა). ჩვეულებრივ, ასეთ შემთხვევებში ოპერაციული სისტემა აჩერებს იმ პროგრამის შესრულებას, რომელმაც წარმოშვა გამოუსწორებელი განსაკუთრებული შემთხვევა.

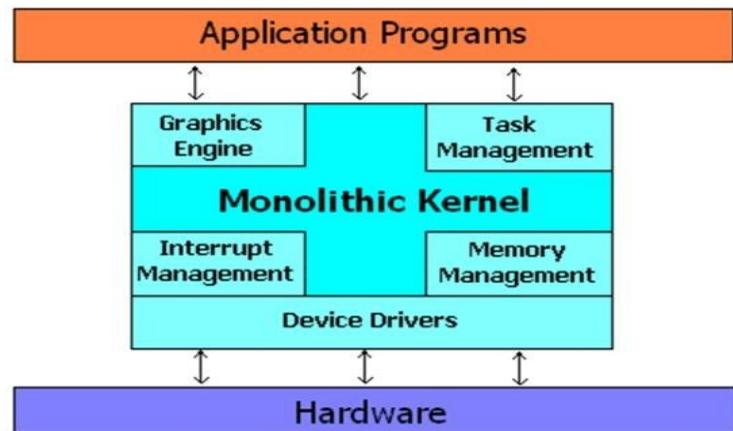
1.5. ოპერაციული სისტემის სტრუქტურა

ოპერაციული სისტემების ნახევარსაუკუნოვანი არსებობის მანძილზე შემუშავებული იყო მრავალი არქიტექტურული მიდგომა (მონოლიტური სისტემა, მრავალდონიანი სისტემა და ა.შ.). თითოეულ მათგანს გააჩნია საკუთარი უპირატესობა და ნაკლოვანება. მათგან ერთმა ნაწილმა ჰქოვა ფართო გამოყენება და განვითარდა, მეორე ნაწილმა კი ვერ ნახა დღის სინათლე. მოკლედ დავახასიათოთ ეს არქიტექტურები.

მონოლიტური სისტემა. მონოლიტური არქიტექტურით ორგანიზებული ოპერაციული სისტემა ფართოდაა გავრცელებული. ამ შემთხვევაში მთლიანი სისტემა მუშაობს როგორც ერთი პროგრამა, ანუ ოპერაციული სისტემის კოდი დაწერილია როგორც პროცედურათა ნაკრები, რომლებიც ერთ დიდ შესრულებად პროგრამაში არის გაერთიანებული (ნახ. 1.9).

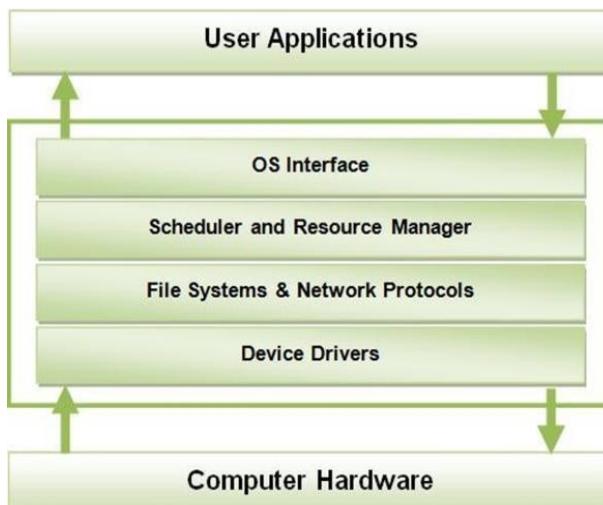
ამ არქიტექტურაში ყოველ პროცედურას შეუძლია გამოიძახოს სხვა პროცედურა, რომელიც გამომძახებელსთვის შეასრულებს სასარგებლო საქმიანობას.

ვინაიდან პროცედურებს გააჩნიათ შეუზღუდავი წვდომა ერთიმეორებე და აპარატურაზე სისტემა მთლიანობაში შეიძლება აღმოჩნდეს არამდგრადი შეცდომის ან ზიანის შემცველი კოდის მიმართ. მონოლიტური არქიტექტურის ბაზაზე აგებულ ოპერაციულ სისტემებს წარმოადგენ OS/360, VMS და Linux ოპერაციულის ისტემები.



ნახ. 1.9. მონოლიტური სისტემის მოდელი

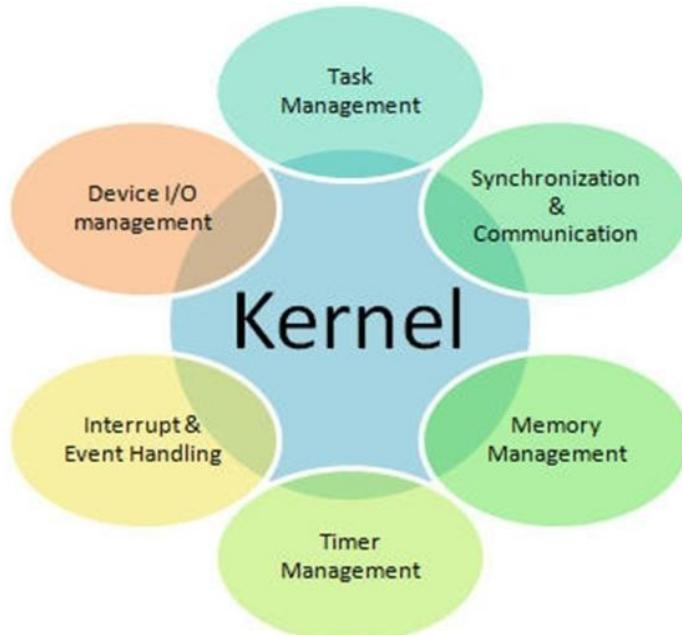
მრავალდონიანი არქიტექტურა. მონოლიტური მიდგომის განზოგადებას წარმოადგენს ოპერაციული სისტემის ორგანიზება დონეების იერარქიის სახით, რომელშიც თითოეული დონეზე თავმოყრილია სისტემაში მსგავსი ფუნქციების განმახორციელებელი კომპონენტები. დონეები ერთიმეორისგან მაღავენ მათ მიერ განსახორციელებელი ამოცანის გადაწყვეტის მექანიზმებს, მაგრამ მეზობელი დონეებს სთავაზობენ მომსახურეობას. ასეთნაირად ორგანიზებულ ოპერაციულ სისტემაში, სხვა დონეებზე ზემოქმედების გარეშე შესაძლებელია, ცალკეული დონეების მოდიფიცირება. დონეებს შორის გადასვლა საჭიროებს შეაღედური ელემენტების გამოყენებას, რაც ამცირებს სისტემის წარმადობას. გარდა ამისა, ვინაიდან მრავალდონიანი არქიტექტურაში დონეები სარგებლობენ ერთიმეორის მომსახურეობით და რესურსებზე წვდომის შეუზღუდავი უფლებით, მონოლიტური მიდგომის მსგავსად, ამ შემთხვევაშიც შეიძლება სისტემა აღმოჩნდეს არამდგრადი შეცდომის ან ზიანის შემცველი კოდის მიმართ. THE (Technische Hogeschool Eindhoven) ოპერაციული სისტემა წარმოადგენს მრავალდონიანი ოპერაციული სისტემის მაგალითს (ნახ. 1.10). მრავალი თანამედროვე ოპერაციული სისტემა, მათ შორის Windows XP და Linux-ი შეიძლება გარკვეული თვალსაზრისით განეკუთვნებოდნენ მრავალდონიან სისტემებს.



ნახ. 1.10. THE ოპერაციული სისტემის დონეები

მიკრობირთვი. ოპერაციული სისტემის შესაქმნელად მრავალდონიანი მიდგომის გამოყენებისას ხშირ შემთხვევებში, პრაქტიულად, შეუძლებელია მომხმარებლის და ბირთვის რეჟიმებს შორის საზღვრის გავლება. ვინაიდან ბირთვის რეჟიმში მუშაობისას შეცდომის ან ზიანის შემცველმა კოდმა შესაძლებელია გამოიწვიოს სისტემის დაზიანება, ამიტომ სასურველია ბირთვის რეჟიმში მუშაობდეს ნაკლები რაოდენობის პროცესები. ამ მიზნით ოპერაციული სისტემების აგების შედგომი ტენდენციები მდგომარეობს სისტემის პროგრამული კოდის უმეტესი ნაწილის მომხმარებლის რეჟიმში გადატანასა და ბირთვის რეჟიმში მომუშავე ნაწილის მნიშვნელოვნად შემცირებაში. მიკრობირთვული არქიტექტურის მიხედვით ოპერაციული სისტემა იყოფა ცალკეულ მოდულებად (ნახ. 1.11). ამ მოდულებიდან მხოლოდ ერთი - **მიკრობირთვი**,

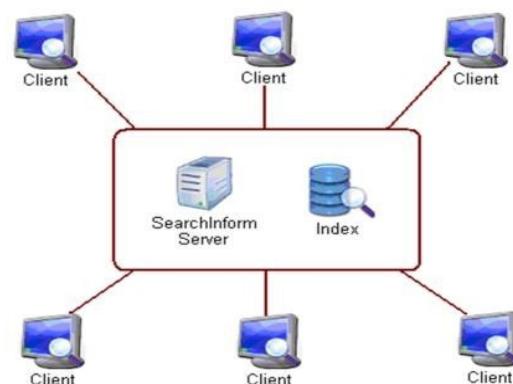
მუშაობს ბირთვის რეჟიმში. ყველა მოდული ერთმანეთთან ურთიერთქმედებს მიკრობირთვის მეშვეობით, რომელიც უზრუნველყოფს პროგრამებს შორის კავშირს, პროცესორის გამოყენების დაგეგმვას, წყვეტის პირველად დამუშავებას, შეტანა/გამოტანის ოპერაციებს და მეხსიერების საბაზო მართვას. მიკრობირთვის ბაზაზე ოპერაციული სისტემის მაგალითს წარმოადგენს Windows -ის ოპერაციული სისტემა.



ნახ. 1.11. მიკრობირთვული სისტემის მოდელი

კლიენტ-სერვერული სისტემა. მიკრობირთვის იდეის გარკვეული ცვლილება გამოიხატება პროცესების ორი კლასის გამოყოფაში: **სერვერი**, რომელიც მომხმარებელს სთავაზობს გარკვეულ მომსახურეობას, და **კლიენტი**, რომელიც სარგებლობს ამ მომსახურეობით. ეს მოდელი ცნობილია, როგორც კლიენტ-სერვერული მოდელი (ნახ. 1.12). კლიენტსა და სერვერს შორის კავშირი ხორციელდება შეტყობინებათა გადაცემის გზით. კლიენტი გარკვეული მომსახურე-ობის მისაღებად ადგენს შესაბამის შეტყობინებას და გადასცემს მას ბირთვს. ბირთვი შემოსულ შეტყობინებას ამუშავებს და პასუხს უბრუნებს გამომგზავნ პროცესს. კლიენტისა და სერვერის ერთ მანქანზე მუშაობისას შესაძლებელია გარკვეული ოპტიმიზაციის განხორციელება (შეტყობინებების გადაცემის მინიმიზირების გზით).

კლიენტ-სერვერული მოდელის არსი მდგომარეობს კლიენტის და სერვერის ლოკალური ან გლობალური ქსელით დაკავშირებულ სხვადასხვა მანქანებზე ამუშავებაში. ვინაიდან კლიენტები და სერვერები ერთიმეორესთან ურთიერთქმედებენ შეტყობინებათა გადაცემის გზით მათვის აუცილებლობას არ წარმოადგენს იმის ცოდნა, თუ როგორ ხორციელდება შეტყობინებების დამუშავება - ხდება ეს ლოკალურად თუ გლობალურად.



ნახ. 1.12. კლიენტ-სერვერული სისტემის მოდელი

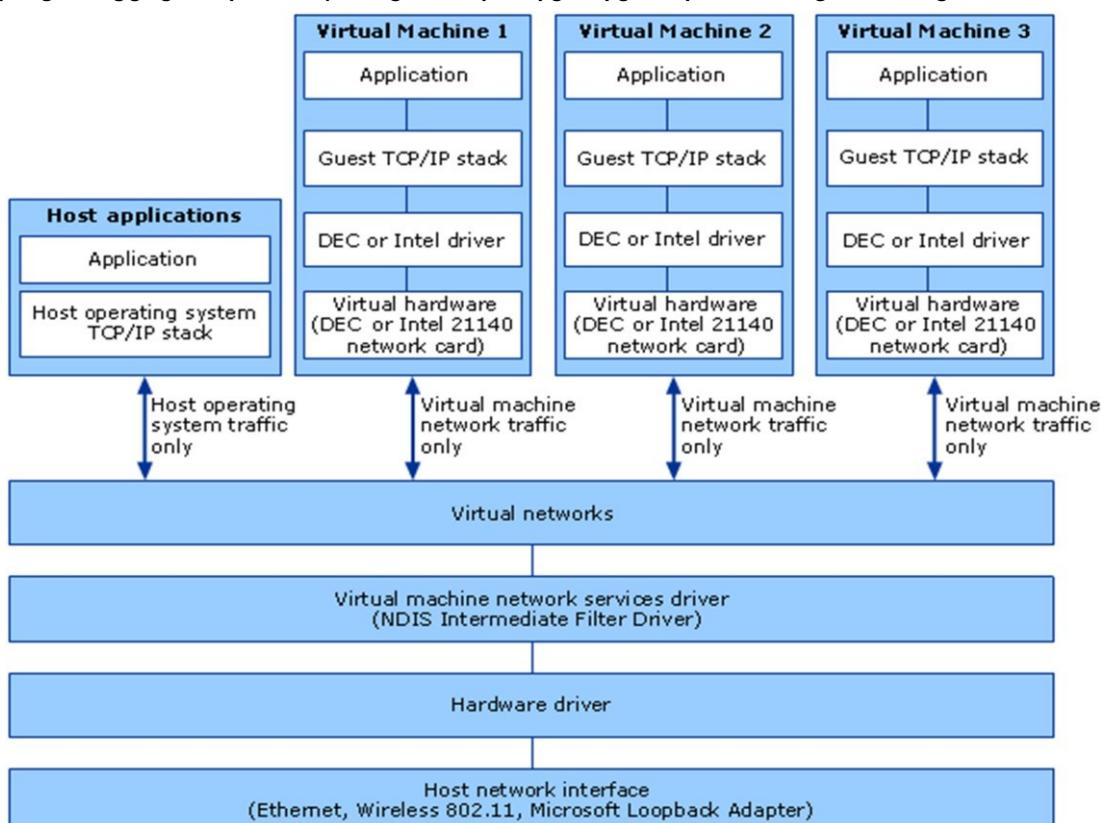
ვირტუალური მანქანა. პირველი გამოთვლითი მანქანის ოპერაციული სისტემები წარმოადგენდნენ უშაალოდ პაკეტური დამუშვების სისტემებს. ასეთ მანქანებზე მომხმარებელს არ

პქონდა საკუთარი ამოცანის შესრულების შესაძლებლობა. მას საკუთარი პროგრამა მიჰქონდა გამოთვლით ცენტრში, სადაც ახერხებდა მის ამუშავებას. მოგვიანებით გამოჩნდა დროის გაყოფის რეჟიმში მომუშავე სისტემები. აქ მომხმარებელს უკვე შეეძლო საკუთარი გამოთვლების განხორციელება, მაგრამ სამუშაოს ასრულებდნენ საკმაოდ დაბალი სიჩქარით და მისი ძირი ღირებულების გამო მანქანა თითქმის მიუწვდომელი იყო მომხმარებლისთვის.

მომხმარებლის მომსახურეობის გაუმჯობესებისა და ხელმისაწვდომობის მიზნით IBM ორგანიზაციამ შეიმუშვა განსხვავებული სისტემა (VM/370), რითაც დროის გაყოფის რეჟიმში მომუშავე მომხმარებელს შესთავაზა (1) მრავალამოცანიანი რეჟიმი და (2) განზოგადებული მანქანა უფრო მოხერხებული ინტერფეისით ვიდრე მოწყობილობებია. ამ სისტემის არსი მდგომარეობდა აღნიშნული ორი ფუნქციისგანანილებაში.

სისტემა, რომელიც ცნობილია როგორც **ვირტუალური მანქანის მონიტორი** (ნახ. 1.13), იგვირთებოდა ჩვეულებრივ აპარატურაზე. ის უზრუნველყოფდა მრავალამოცანიანობას ზედა დონისთვის არა ერთი არამედ რამდენმიმე ვირტუალური მანქანის შეთავაზებით. ოპერაციული სისტემებისგან განსხვავებით ეს ვირტუალური მანქანები არ წარმოადგენდნენ მანქანებს განზოგადოებული არქიტექტურით. მათ არ გააჩნდათ ფაილების მხარდაჭერა. ნაცვლად ამისა ისინი იძლეოდნენ გამოსავალი აპარატურის, ბირთვის და მომხმარებლის რეჟიმის, შეტანა/გამოტანის მოწყობილობების და სხვა კომპონენტების, რომელიც რეალურ მანქანას გააჩნია, სრულ აბსტრაქციას.

ვინაიდან ყოველი ვირტუალური მანქანა იდენტური კომპონენტებისგან კონსტრუირებული მანქანის, ამიტომ თითოეულ მათგანში შეიძლება ჩაიტვირთოს რეალურ მანქანაზე მომუშავე ნებისმიერი ოპერაციული სისტემა. ფიზიკურ მანქანაზე შესაძლებელი იყო ერთდროულად რამდენიმე ვირტუალური მანქანის ამუშავება. შესაბამისი ტექნოლოგიური განვითარების არარსებობის გამო ვირტუალური მაქანების მიღვომამ იმ პერიოდში ვერ ჰქოვა პრაქტიკული გამოყენება და ბოლო პერიოდამდე ხდებოდა მისი იგნორირება.



ნახ. 1.13. ვირტუალური მანქანის მოდელი

პროგრამულ უზრუნველყოფის და ტექნოლოგიების განვითარების ახალმა ტენდენციებმა აქტუალური გახადა ვირტუალური მანქანის თემა. კომპიუტერული ტექნიკის მნარმოებელმა მრავალმა ორგნიზაციამ საკუთარ პროდუქტში დამატა ვირტუალიზაციის შესაძლებლობა. მაღალმნარმოებლურ სერვერებზე ვირტუალიზაცია განიხილება, როგორც მექანიზმი ერთ

მანქანაზე სხვადასხვა სერვერების ერთდროული ამუშვებისა (რიგ შემთხვევაში სხვადასხვა ოპერაციული სისტემებით) და რომელიმე მათგანის მწყობრიდან გამოსვლის (პროგრამული გაუმართაობის) შემთხვევაში დანარჩენი სერვერების ნორმალური ფუნქციონირების გარანტი. პერსონალურ კომპიუტერებზე ვირტუალიზაციის შესაძლებლობა ნიშნავს მომხმარებლის მხრიდან რამდენიმე, მაგალითად, Windows და Linux, ოპერაციული სისტემის ერთდროული ამუშავების შესაძლებლობას. კომპიუტერში ვირტუალიზაციის შესაძლებლობის გამოყენება საჭიროებს გარკვეულ პროგრამულ უზრუნველყოფას. კომპიუტერულ სისტემაში ვირტუალიზაციის შესაძლებლობის რეალიზაცია შესაძლებელია შემდეგი პროგრამების გამოყენებით VMware workstation, Virtual PC, Oracle VM VirtualBox და სხვა.

შერეული (პიძრიდული) სისტემები. ოპერაციული სისტემის აგების ზემოთ განხილულ ყოველ მიღვომას გააჩნია თავისი უპირატესობა და ნაკლოვანება. უმეტეს შემთხვევებში თანამედროვე ოპერაციული სისტემები იყენებენ ამ მიღვომების სხვადასხვა კომბინაციებს. მაგალითად, Linux ოპერაციული სისტემის ბირთვი წარმოადგენს მონოლიტურ სისტემას მიკრობირთვული არქიტექტურის ელემენტებით. ბირთვის კომპილაციისას შესაძლებელია ბირთვის მრავალი კომპონენტის ე.წ. მოდულების დინამიური ჩატვირთვა/ამოტვირთვა. მოდულის ჩატვირთვის მომენტში მისი კოდი იტვირთება სისტემის დონეზე და უკავშირდება ბირთვის დანარჩენ ნაწილს. მოდულის შიგნით შესაძლებელია იყოს გამოყენებული ბირთვის მიერ ექსპორტირებული ფუნქცია.

მიკრობირთვული არქიტექტურის და მონოლიტური ბირთვის ელემეტები ყველაზე მეტადაა შეზავებული Windows NT ბირთვში. თუმცა Windows NT-ს ხშირად უწოდებენ მიკრობირთვული ოპერაციულ სისტემას. Windows NT-ს კომპონენტები განთავსებულია დინამიურ მეხსიერებაში და ერთმანეთთან კავშირს ამყარებენ შეტყობინებათა გადაცემის გზით, რაც სავსებით დამახასიათებელია მიკრობირთვული არქიტექტურისთვის, მაგრამ იმავდროულად ბირთვის ყოველი კომპონენტი მუშაობს მისამართების ერთ სივრცეში და აქტიურად იყენებენ მონაცემთა საერთო სტრუქტურებს, რაც დამახასიათებელია მონოლიტური ოპერაციული სისტემისთვის.

ლექცია 2. პროცესები

ოპერაციული სისტემისთვის ერთერთ მნიშვნელოვან ცნებას წარმოადგენს პროცესის ცნება: აბსტრაქცია, რომელიც აღწერს პროგრამას შესრულების მომენტში. ყველაფერი რაც გამოთვლით სისტემაში ხდება (სისტემის ჩატვირთვა, გამოყენებითი პროგრამების შესრულება, ქსელური კომუნიკაცია და ა.შ.) რეალიზებულია პროცესის ცნების გამოყენებით, ამიტომ მნიშვნელოვანია თავიდანვე გვქონდეს პროცესის კონცეფციაზე სრული წარმოდგენა. პროცესები ერთი ცენტრალური პროცესორის არსებობის შემთხვევაშიც კი იძლევიან ოპერაციების (ფსევდო) პარალელური შესრულების შესაძლებლობას და ფიზიკურ პროცესორს წარმოგვიდგენენ რამდენიმე ვირტუალური პროცესორის სახით.

2.1. პროცესი

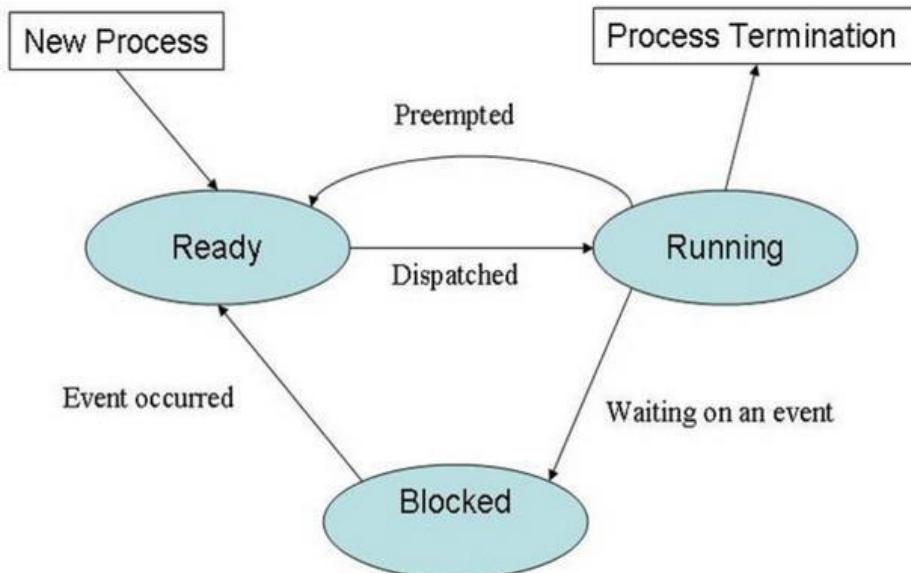
თანამედროვე კომპიუტერები დაკავებული არიან ერთდროულად რამდენიმე ამოცანის შესრულებით. მაგალითად, გლობალური ქსელიდან web-სერვერზე მუდმივად შემოდის მრავალი მოთხოვნა web-გვერდზე განთავსებული ინფორმაციის დათვალიერებაზე, იქიდან მონაცემების გადმოწერაზე ან იქ საკუთარი მონაცემების განთავსებაზე. ხშირად ეს მონაცემები განთავსებულია ლოკალურად (ერთ გამოთვლით მანქანაში არსებულ რამდენიმე მყარ დისკზე ან ლოკალურ ქსელში გაერთიანებული რამდენიმე გამოთვლით მანქანაში გამოყენებულ მყარ დისკებზე) ან ტერიტორიულად სხვადასხვა ადგილას განთავსებული გამოთვლითი მანქანების მყარ დისკებზე. მოთხოვნის შემოსვლისთანავე სერვერი ამოწმებს web-გვერდზე ჩანაწერის არსებობას მის რეგისტრებში. თუ ასეთი ჩანაწერი არსებობს, მაშინ აგზავნის ამ web-გვერდს შემოსულს მოთხოვნაზე პასუხად. წინააღმდეგ შემთხვევაში სერვერი მიმდევრობით მიმართავს ქეშ-მეხსიერებას (L1 -> L2), ოპერატიულ მეხსიერებას და ბოლოს მყარ დისკს სანამ შესაბამის web-გვერდზე ინფორმაცია არ იქნება მოძიებული. ინფორმაციის ძებნის პროცესში შესაძლებელია სერვერზე შემოვიდეს სხვა მრავალი მოთხოვნა. თუ სისტემაში რამდენიმე მყარი დისკია, მაშინ პირველი მოთხოვნის დაკმაყოფილებამდე შემოსული მოთხოვნებიდან ზოგიერთი ან ყველა შესაძლებელია გადამისამართებული იქნას იმავე ან სხვადასხვა მყარ დისკზე. ცხადია, რომ ასეთი პარალელური ოპერაციების განსახორციელებლად და მათ სამართავად საჭიროა გარკვეული მექანიზმები. მოქმედებების პარალელური შესრულება შესაძლებელია პროცესების აბსტრაქციის გამოყენებით.

ახლა განვიხილოთ პერსონალური კომპიუტერის მაგალითი. სისტემის ჩატვირთვასთან ერთად იტვირთება რამდენიმე პროცესი. ასეთი პროცესების რიცხვს შეიძლება განეკუთვნებოდეს პროცესები, რომლებიც ამოწმებენ ფიზიკური მოწყობილობის მაკომპლექტებლების ფუნქციონირებას (პროცესორი, ოპერატიული მეხსიერება, მყარი დისკი და ა.შ.), სისტემური პროცესები, რომლებიც ხელს უწყობენ ოპერაციული სისტემის ნორმალურ ფუნქციონირებას, და გამოყენებითი პროცესები, რომლებიც გარკვეულ სამომხმარებლო პროგრამებს ეხმარებიან დასახული ამოცანის შესრულებაში. მაგალითად, ამოწმებენ ელექტრონულ ფოსტას ახალი წერილის შემოსვლაზე ან ანტივირუსული პროგრამის მიერ ამუშავებული პროცესები, რომლებიც მას ეხმარებიან სისტემაში ვირუსის მატარებელი ინფორმაციის აღმოჩენაში და ა.შ. ასევე, სისტემაში შეიძლება იყოს მომხმარებლის მიერ ინიცირებული პროცესები. მაგალითად, ფაილიდან მონაცემების დასაბეჭდად ან კომპაქტ-დისკზე მონაცემების ჩასაწერად ამუშავებული პროცესები.

მრავალამოცანიან სისტემაში პროცესორი მცირე დროითი შუალედით (დროითი კვანტით) გამოეყოფა თითოეულ პროცესს. მიუხედავად იმისა, რომ დროის ყოველ კონკრეტულ მომენტში პროცესორი შეიძლება გამოეყოს მხოლოდ ერთ პროცესს, დროით კვანტის მოცულობის მიხედვით პროცესორმა შეიძლება 1 წმ.-ის განმავლობაში მოასწროს სისტემაში არსებულ ყველა პროცესთან ან მათ გარკვეულ ჯგუფთან მუშაობა, რის გამოც გვექმნება პროცესების პარალელურად შესრულების ილუზია. ასეთ შემთხვევაში ამბობენ, რომ პროცესები სრულდება ფსევდოპარალელურად. ადამიანისთვის ძნელია სისტემაში პარალელურად მიმდინარე გამოთვლებზე თვალყურისდევნება. ამ მიზნით შემუშავებული იქნა მიმდევრობითი პროცესების კონცეპტუალური მოდელი, რომელიც ამარტივებს პარალელურ გამოთვლებთან მუშაობას.

2.2. პროცესის სიცოცხლის ციკლი

ოპერაციული სისტემა ვალდებულია ყოველ პროცესს შესასრულებლად შესთავაზოს საკმარისი დრო. რეალურად სისტემაში ერთდროულად შეიძლება ამუშავებული იყოს იმდენი პროცესი რამდენიც პროცესორი ან ბირთვია სისტემაში. ჩვენი საუბარი ძირითადად შეეხება ერთპროცესორულ სისტემას. რამდენიმე პროცესორის ან ბირთვის შემთხვევაში მსჯელობა ანალოგიურია იმ განსხვავებით, რომ მრავალპროცესორულ სისტემაში გასათვალისწინებელია დამატებითი სინქრონიზაციის მექანიზმები. რადგანაც მხოლოდ ერთპროცესორულ სისტემებზე ვისაუბრებთ, ამიტომ დროის ნებისმიერ მომენტში სისტემაში შეიძლება სრულდებოდეს მხოლოდ ერთი პროცესი.



ნახ. 2.1. პროცესის მდგომარეობათადიაგრამა

პროცესის სიცოცხლის ციკლი შედგება მიმდევრობითი დისკრეტული მდგომარეობებისგან (process state) (ნახ. 2.1). პროცესის ერთი მდგომარეობიდან მეორეზე გადასვლა შეიძლება პროვოცირებული იყოს სხვადასხვა მოვლენით. ამბობენ, რომ პროცესი იმყოფება შესრულების მდგომარეობაში (running state) ანუ სრულდება, თუ მიმდინარე მომენტში მისთვის გამოყოფილია პროცესორი. ამბობენ, რომ პროცესი იმყოფება მზადყოფნის მდგომარეობაში (ready state), თუ პროცესისთვის პროცესორის გადაცემის შემთხვევაში ის შეძლებს შესრულებას. ამბობენ, რომ პროცესი იმყოფება ბლოკირებულ მდგომარეობაში (blocked state), თუ მის შესასრულებლად საჭიროა გარკვეული მოვლენის დადგომა (მაგალითად, შეტანა/გამოტანის ოპერაციის დასრულება).

სხვადასხვა ოპერაციულ სისტემაში გამოიყენება დამატებითი მდგომარეობები. ჩვენ შემოვიფარგლებით მხოლოდ აღწერილი მდგომარეობებით.

ოპერაციულ სისტემაში დროის ნებისმიერ მომენტში არსებობენ პროცესები, რომელთა ერთი ნაწილი იმყოფება მზადყოფნის მდგომარეობაში, ხოლო მეორე ნაწილი ბლოკირებულ მდგომარეობაში. ამიტომ ოპერაციულ სისტემაში იქმნება მზადყოფნის მდგომარეობაში მყოფი პროცესების ცხრილი (ready list) (რომელსაც მზადყოფნის ცხრილს უწოდებენ) და ბლოკირებული პროცესების ცხრილი (blocked list). მზადყოფნის ცხრილში პროცესები განთავსებულია პრიორიტეტის მიხედვით ისე, რომ შემდეგი პროცესი, რომელიც მიიღებს პროცესორს, განთავსებულია ცხრილის თავში (ანუ უმაღლესი პრიორიტეტის მქონე პროცესი). ბლოკირებული პროცესების ცხრილში პროცესები განთავსებულია დაულაგებელი ფორმით, ვინაიდან ბლოკირებული მდგომარეობიდან გამოსვლა შესაძლებელია მოხდეს გარკვეული მოვლენის დადგომის შემდეგ, რომლის დადგომის დროის განსაზღვრა წინასწარ შეუძლებელია.

2.3. პროცესების მართვა

ვინაიდან ოპერაციული სისტემაში პროცესები სრულდებიან მონაცვლეობით, ამიტომ იმისთვის, რომ შეჩერების და განახლების ოპერაციების შესრულებისას პროცესებმა ხელი არ შეუშალონ ერთიმეორეს, საჭიროა პროცესების ორგანიზებული მართვა. გარდა ამისა, პროცესებს უნდა შეეძლოთ ოპერაციულ სისტემასთან ურთიერთქმედება და ისეთი ელემენტარული მოქმედებების განხორციელება როგორიცაა ახალი პროცესის შექმნა ან ოპერაციული სისტემის ინფორმირება მიმდინარე პროცესის დასრულების თაობაზე.

2.3.1. მდგომარეობებს შორის გადასვლა

მომხმარებელის მიერ რაიმე პროგრამის ამუშავებისას, იქმნება პროცესი, რომელიც თავსდება მზადყოფნის ცხრილში. მზადყოფნის ცხრილში განთავსებული ყოველი პროცესი (მის წინ განთავსებული პროცესებისთვის პროცესორის გამოყოფის ხარჯზე) მიიწევს ცხრილის დასაწყისისკენ. მზადყოფნის ცხრილის სათავეში მოქცეული პროცესისთვის ხელმისაწვდომი ხდება პროცესორი. პროცესორის გამოყოფის შემდეგ პროცესი მზადყოფნის მდგომარეობიდან გადადის შესრულების მდგომარეობაში. პროცესის ერთი მდგომარეობიდან მერეში გადასვლის მომენტს ეწოდება მდგომარეობის შეცვლა (process transition). პროცესისთვის პროცესორის შეთავაზებას ანხორციელებს პროცესორის მმართველი (processor dispatcher) პროგრამა. ოპერაციული სისტემა მართავს პროცესის გადაყვანას ერთი მდგომარეობიდან მეორეში და ამით არეგულირებს პროცესებისთვის პროცესორის გამოყოფის თანაბრობას. იმისთვის, რომ რაიმე პროცესის მიერ არ მოხდეს პროცესორის შემთხვევითი ან მიზანმიმართული დაკავება, ოპერაციული სისტემა სპეციალურ აპარატულ ტაიმერში (interrupting clock) აყენებს დროითი კვანტის (quantum) მნიშვნელობას, რომლის განმავლობაშიც მიმდინარე პროცესს ეძლევა პროცესორის გამოყენების შესაძლებლობა. თუ დროითი კვანტი საკმარისი აღმოჩნდა პროცესის დასრულებისთვის, მაშინ პროცესის დასრულების შემდეგ პროცესორი უბრუნდება ოპერაციულ სისტემას და ის ამ უკანასკნელს გასცემს რიგით შემდეგ პროცესზე. თუ დროითი კვანტი პროცესის დასრულებისთვის არ აღმოჩნდა საკმარისი, მაშინ პროცესი კვლავ გადადის მზადყოფნის მდგომარეობაში და თავსდება მზადყოფნის ცხრილის ბოლოში. შესრულების მომენტში შეტანა/გამოტანის გარკველი ოპერაციის საჭიროებისას პროცესი

წებაყოფლობით თმობს პროცესორს და გადადის ბლოკირებულ მდგომარეობაში. ბლოკირებული მდგომარეობიდან პროცესს შეუძლია გამოსვლა მხოლოდ შეტანა/გამოტანის შესაბამისი ოპერაციის დასრულების შედეგ. ბლოკირებული მდგომარეობიდან გამოსული პროცესი თავსდება მზადყოფნის ცხრილის ბოლოში.

შევნიშნოთ, რომ თუ შესრულების ან ბლოკირებული მდგომარეობიდან პროცესის გამოსვლა დაემთხვა სისტემაში ახალი პროცესის წარმოქმნას, მაშინ მზადყოფნის ცხრილში ჯერ თავსდება ახლად შექმნილი პროცესი, ხოლო შემდეგ შესაბამისი მდგომარეობიდან გამოსული პროცესი.

ამრიგად, ჩვენ განვსაზღვრეთ მდგომარეობებს შორის 4 შესაძლო გადასვლა. პროცესის ამუშავების შემდეგ ის მზადყოფნის მდგომარეობიდან გადადის შესრულების მდგომარეობაში. დროითი კვანტის ამოწურვის შემდეგ პროცესი შესრულების მდგომარეობიდან უკან უბრუნდება მზადყოფნის მდგომარეობას. როცა რაიმე მიზეზით ხდება პროცესის ბლოკირება ის შესრულების მდგომარეობიდან გადადის ბლოკირებულ მდგომარეობაში. და ბოლოს, შესაბამისი მოვლენის დადგომის შემდეგ (რომელსაც ის ელოდებოდა) პროცესი გამოდის ბლოკირებული მდგომარეობიდან და გადადის მზადყოფნის მდგომარეობაში. გარდა ბლოკირებული მდგომარეობისა, რომელსაც ინიცირებს მომხმარებელი, სხვა დანარჩენი მდგომარეობების ინიცირებას ანხორციელებს ოპერაციული სისტემა.

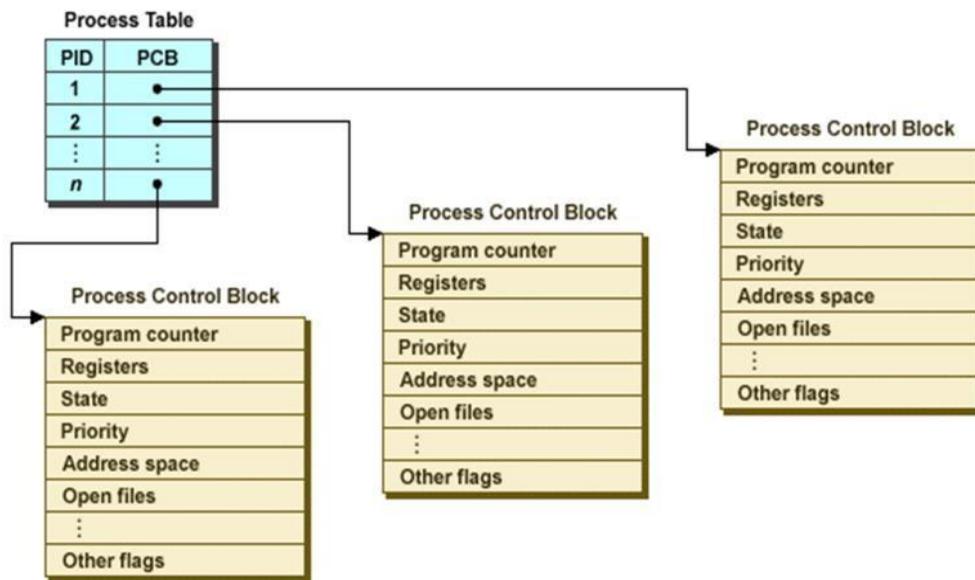
2.3.2. პროცესის მართვის ბლოკი. პროცესის დესკრიპტორი

ახალი პროცესის შექმნისას ოპერაციული სისტემა ახორციელებს მთელ რიგ მოქმედებებს. პირველ რიგში, ყველა პროცესი რაიმე ფორმით უნდა განსხვავდებოდეს ერთიმეორისგან. ამ მიზნით პროცესს ენიჭება უნიკალური იდენტიფიკატორის მნიშვნელობა (PID – process identifier). შემდეგ ოპერაციული სისტემა ქმნის პროცესის მართვის ბლოკს (PCB – process control block), რომელსაც პროცესის დესკრიპტორს (process descriptor) უწოდებენ, და მასში ანთავსებს პროცესის მართვისათვის საჭირო ყველა აუცილებელ ინფორმაციას. პროცესის მართვის ბლოკში შედის შემდეგი ინფორმაცია:

- PID;
- პროცესის მიმდინარე მდგომარეობა (სრულდება, მზადაა თუ ბლოკირებულია);
- ბრძანებათა მთვლელი (program counter), რომელიც განსაზღვრავს თუ რომელი ბრძანება უნდა შესრულდეს შემდეგი;
- პროცესის პრიორიტეტი;
- უფლებები (მონაცემები, რომლებიც განსაზღვრავენ იმ რესურსების ჩამონათვალს რომელზეც ექნება დამვება მიმდინარე პროცესს);
- მიმთითებელი მშობელ პროცესზე (parent process);
- მიმთითებელი შვილ პროცესზე (child process);
- მეხსიერებაში პროცესის მონაცემებზე და ინსტრუქციებზე მიმთითებელი;
- პროცესისთვის გამოყოფილ რესურსებზე მიმთითებელი.

გარდა ამისა, პროცესის მართვის ბლოკში ინახება პროცესორის რეგისტრების შემცველობა, ე.წ. პროცესის შესრულების კონტექსტი (execution process), პროცესის გამოსვლამდე მდგომარეობა შესრულებიდან. პროცესის შესრულების კონტექსტი დამოკიდებულია გამოყენებული სისტემის არქიტექტურაზე და, ჩვეულებრივ, შეიცავს ძირითადი დანიშნულების რეგისტრების (სადაც ინახება პირდაპირ წვდომადი პროცესის მონაცემები) და პროცესის მართვის რეგისტრების

(მაგალითად, პროცესის მისამართების სივრცეზე მიმთითებელი) შემცველობას. პროცესის შესრულების კონტექსტში შენახული ინფორმაციის გამოყენებით ოპერაციულ სისტემას შესრულების მდგომარეობაში გადასული პროცესი შეუძლია აღადგინოს შეჩერებული მდგომარეობიდან.



ნახ. 2.2. პროცესების სისტემური ცხრილი

პროცესის მდგომარეობის შეცვლისას ოპერაციულმა სისტემამ უნდა განაახლოს პროცესზე ინფორმაცია მის მართვის ბლოკში. ჩვეულებრივ, პროცესზე ინფორმაციას ოპერაციული სისტემა ინახავს პროცესების სისტემურ ცხრილში (ნახ. 2.2), რათა დააჩქაროს წვდომა საჭირო ინფორმაციაზე. პროცესის მუშაობის დასრულების შემდეგ ოპერაციული სისტემა ათავისუფლებს პროცესის მიერ დაკავებულ მეხსიერებას, მის მიერ დაკავებულ რესურსებს და პროცესების ცხრილიდან შლის ინფორმაციას პროცესზე. ამის შემდეგ გამოთავისუფლებული რესურსების გამოყენება შეუძლია სისტემაში წარმოქმნილ სხვა პროცესებს.

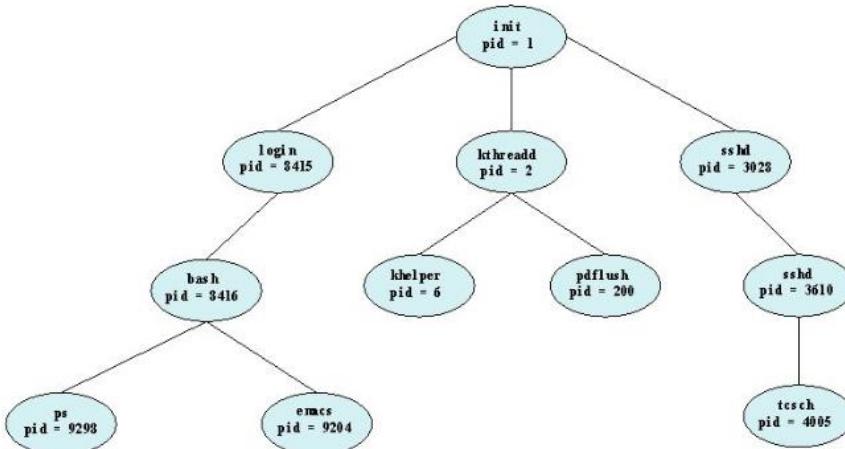
2.3.3. ოპერაციები პროცესებზე

ოპერაციულ სისტემას უნდა გააჩნდეს პროცესებზე გარკვეული ოპერაციების განხორციელების შესაძლებლობა, მათ შორის:

- პროცესის წარმოქმნა;
- პროცესის განადგურება;
- პროცესის შესრულების შეჩერება და განახლება;;
- პროცესის პრიორიტეტის შეცვლა;
- პროცესის ბლოკირება;
- პროცესის არჩევა შესასრულებლად;
- პროცესების ურთიერთქმედების უზრუნველყოფა.

პროცესმა შეიძლება წარმოქმნას ახალი პროცესები. წარმოქმნილ პროცესს შვილი პროცესი (child process) ეწოდება, ხოლო წარმოქმნელ პროცესს კი - მშობელი პროცესი (parent process). ცხადია, რომ ყოველ პროცესს შეიძლება ჰყავდეს მხოლოდ ერთი მშობელი პროცესი და ჰყავდეს საკუთარი შვილი პროცესები. ასეთი მიდგომით მიიღება პროცესების იერარქიული სტრუქტურა (hierarchical process structure) (ნახ. 2.3), რომელშიც ყოველ პროცესს ჰყავს ერთი მშობელი პროცესი. პროცესს შეიძლება ჰყავდეს რამდენიმე შვილი პროცესი. UNIX-მსგავს სისტემაში, მაგალითად, Linux სისტემაში ყოველი პროცესი იქმნება init პროცესის მეშვეობით, რომელიც თავის მხრივ წარმოიქმნება სისტემის ჩატვირთვის

პროცესში. მაგალითად, init პროცესის მიერ იქმნება პროცესები kswapd, xfs, khubd (და სხვა), რომლებიც, შესაბამისად, მართავენ მეხსიერებას, ფაილურ სისტემას და მოწყობილობებს. პროცესის განადგურება გულისხმოს მის მთლიანად წაშლას სისტემიდან. პროცესის დასრულებისას მის მიერ დაკავებული მეხსიერების ნაწილი და რესურსები უბრუნდება ოპერაციულ სისტემას, პროცესზე ინფორმაცია იშლება ყველა სისტემური ცხრილიდან, იშლება პროცესის PCB და ამის შემდეგ პროცესის მიერ დაკავებული მეხსიერება შეიძლება გამოყენებული იქნას სხვა პროცესის მიერ. თუ პროცეს გააჩნია შვილი პროცესები, მაშინ მისი წაშლა სისტემიდან შედარებით რთული ამოცანაა. ზოგიერთ ოპერაციულ სისტემაში პროცესის წაშლასთან ერთად სისტემიდან ავტომატურად იშლება მის მიერ წარმოქმნილი პროცესები, ხოლო ზოგიერთში კი წარმოქმნილი პროცესები რჩებიან სისტემაში და განაგრძობენ არსებობას.



ნახ. 2.3. პროცესების იერარქია

პროცესის პრიორიტეტის შეცვლა გულისხმობს პროცესის PCB-ში პრიორიტეტის მნიშვნელობის შეცვლას. პროცესის დაგეგმვის რეალიზებული მექანიზმის შესაბამისად შესაძლებელია საჭირო გახდეს PCB-ზე მიმთითებლის გადაადგილება პრიორიტეტების სხვა მიმდევრობაზე.

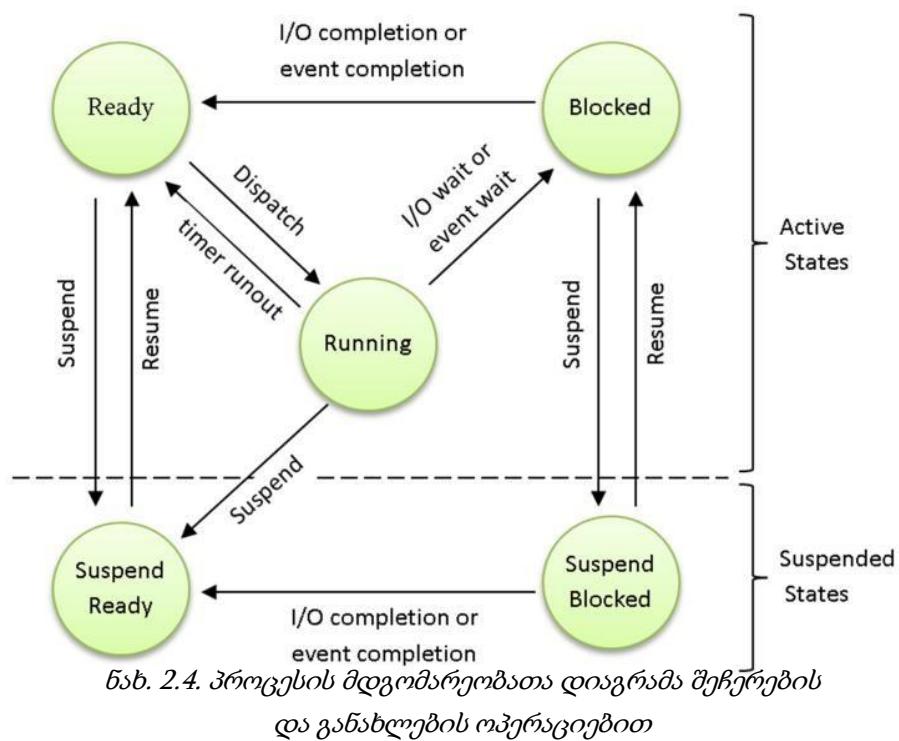
2.3.4. პროცესის შესრულების შეჩერება და განახლება

მრავალ ოპერაციულ სისტემაში მომხმარებელს (შესაბამისი უფლებების ქონის შემთხვევაში) შეუძლია შეაჩეროს პროცესის შესრულება. შეჩერებული (suspended) პროცესი გარკვეული დროით იშლება მზადყოფნის ცხრილიდან, მაგრამ არ იშლება სისტემიდან. ადრე პროცესის შეჩერების მოქმედება იძლეოდა სისტემის დატვირთულობის კორექტირების ან მტყუნებაზე რეაგირების შესაძლებლობას. თანამედროვე კომპიუტერების სწრაფებების ამის გაკეთების საშუალებას არ იძლევა. მიუხედავად ამისა, პროცესის მუშაობით უკმაყოფილო მომხმარებელს მაინც შეუძლია შეაჩეროს პროცესის შესრულება (ავარიული დასრულება (abort)), სანამ არ დარწმუნდება პროცესის ნორმალურ ფუნქციონირებაში. პროცესის შესრულების შეჩერება გამოიყენება სისტემის უსაფრთხოებისთვის საფრთხეების აღმოჩენის მიზნით (მაგალითად, ზიანის შემცველი კოდის აღმოსაჩენად) ან ახალი პროგრამული უზრუნველყოფის ტესტირების მიზნით.

ნახ. 2.4-ზე ნაჩვენებია პროცესის მდგომარეობის დიაგრამა შეჩერებისა და განახლების ოპერაციების გათვალისწინებით. დიაგრამაზე შემოღებულია ორი ახალი მდგომარეობა: „შეჩერებული მზად“ (suspended-ready) და „შეჩერებული ბლოკირებული“ (suspended-blocked).

პროცესის შესრულების შეჩერების ინიციატორი შეიძლება იყოს ან თვითონ პროცესი ან სხვა პროცესი. ერთპროცესორულ სისტემაში შესრულებად პროცესს შეუძლია მხოლოდ საკუთარი შესრულების შეჩერების ინიცირება. არც მზადყოფნის მდგომარეობაში მყოფ და არც ბლოკირებულ მდგომარეობაში მყოფ პროცესს არ შეუძლია გამოიწვიოს შესრულებადი პროცესის შეჩერება.

მრავალპროცესორულ სისტემაში სხვადასხვა პროცესორებზე შესრულებად (შესრულების მდგომარეობაში მყოფ) პროცესებს შეუძლიათ ერთიმეორის შესრულების შეჩერება.

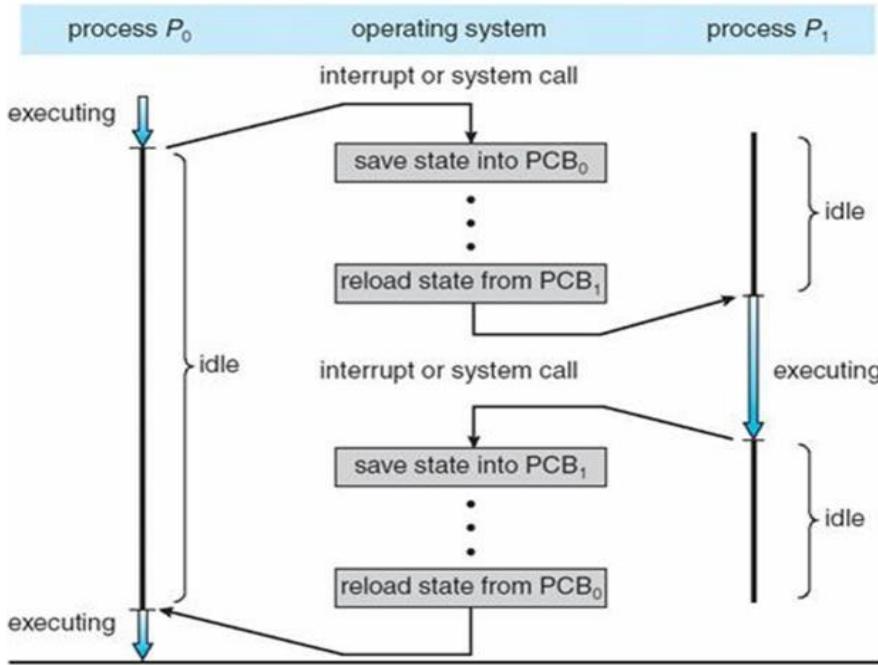


„შეჩერებული მზად“ მდგომარეობაში პროცესი შეიძლება გადავიდეს საკუთარი შესრულების შეჩერების შედეგად ან ამ მდგომარეობაში გადაყვანილი იქნას სხვა პროცესის მიერ. ამ მდგომარეობიდან პროცესს შეუძლია მხოლოდ მზადყოფნის მდგომარეობაში გადასვლა, რომლის ინიცირებაც შეუძლია სისტემაში არსებულ სხვა პროცესს. ბლოკირებული პროცესი სისტემაში არსებული სხვა პროცესის მიერ შეიძლება გადაყვანილი იქნას მდგომარეობაში „შეჩერებული ბლოკირებული“. ამ მდგომარეობიდან პროცესის გამოსვლა შეიძლება ინიცირებული იქნას სისტემაში არსებული სხვა პროცესის მიერ, რომელიც მიმდინარე პროცესს გადაიყვანს მდგომარეობაში ბლოკირებული, ან გარკვეული მოვლენის დადგომის შედეგად, რომელიც მას გადაიყვანს მდგომარეობაში „შეჩერებული მზად“.

2.3.5. კონტექსტის გადართვა

როგორც აღნიშნეთ კონტექსტის გადართვა ეს არის ოპერაცია, რომელიც გულისხმობს პროცესის მდგომარეობის შეცვლას. ოპერაციული სისტემა ახორციელებს კონტექსტის გადართვის ოპერაციას რათა შეაჩეროს ერთი პროცესის შესრულება და პროცესორი შესასრულებლად გამოუყოს სხვა პროცესს. კონტექსტის გადართვამდე საჭიროა ოპერაციული სისტემის ბირთვმა განახორციელოს შესაჩერებელი პროცესის კონტექსტის შენახვა პროცესის მართვის ბლოკში და მომდევნო შესასრულებელი პროცესის PCB-დან მოახდინოს კონტექსტის ჩატვირთვა პროცესორის რეგისტრებში (ნახ. 2.5).

პროცესის კონსტექსტის გადართვა მნიშვნელოვანია მრავალამოცანიანი სისტემისთვის. კონტექსტის გადართვის მომენტში პროცესორს არ შეუძლია სასარგებლო გამოთვლების წარმოება (ასრულებს ცარიელ ციკლს). ის იწვევს პროცესორული დროის ზედნადებ დანახარჯებს, ამიტომ ოპერაციული სისტემა ცდილობს შეამციროს კონტექსტის გადართვაზე დახარჯული დრო.



ნახ. 2.5. კონტექსტის გადართვა

ოპერაციული სისტემა ხშირად მიმართავს პროცესის მართვის ბლოკს (PCB), ამიტომ პროცესორების უმეტეს გააჩნია ჩაშენებული რეგისტრები, რომლებიც უთითებენ პროცესის PCB-ს და ამით აჩქარებენ კონტექსტის გადართვის პროცესს. როდესაც ოპერაციული სისტემა ინიცირებს პროცესის კონტექსტის გადართვას, პროცესორი მის რეგისტრებში არსებულ ინფორმაციას ინახავს მიმდინარე პროცესის PCB-ში. გარდა ამისა, პროცესორი ამარტივებს და აჩქარებს კონტექსტის გადართვის პროცესს ამ მიზნით შემქნილი სპეციალური პროცედურების გამოყენებით.

2.4. წყვეტა

წყვეტა (interrupt) არის მექანიზმი, რომელიც პროგრამულ უზრუნველყოფას აძლევს საშუალებას მოახდინოს შესაბამისი რეგისტრების აპარატურიდან შემოსულ სიგნალებზე. სიგნალების ნაირსახეობის დამუშავებისთვის ოპერაციულ სისტემაში გათვალისწინებულია სპეციალური პროგრამები, ე.წ. **წყვეტის დამმუშავებლები** (interrupt handlers). ოპერაციული სისტემა წყვეტის სიგნალის აღმოჩენის შემთხვევაში დროებით აჩერებს მიმდინარე პროცესის შესრულებას და შემოსულ სიგნალს გადასცემს წყვეტის დამმუშავებელ შესაბამის პროგრამას.

პროცესის ინსტრუქციების შესრულებისას პროცესორმა შეიძლება დააგენერიროს წყვეტის სიგნალი (ამ შემთხვევების საუბარია ე.წ. ხაფუნგზე (trap), რომელიც სინქრონულია (synchronous) პროცესის ოპერაციებთან მიმართებაში). სინქრონულ წყვეტებს ადგილი აქვს, როცა პროცესი ცდილობს აკრძალული ოპერაციების შესრულებას, მაგალითად, როგორიცაა ნულზე გაყოფა, ან მეხსიერების დაცულ მისამართზე მიმართვა.

წყვეტა შესაძლებელია გამოწვეული იყოს პროცესის მიმდინარე ინსტრუქციებისგან დამოუკიდებელი სხვა მოვლენით (ასეთ წყვეტას ასინქრონული (asynchronous) წყვეტა ეწოდება). აპარატურა აგენერირებს ასინქრონულ წყვეტას რათა მოახდინოს პროცესორის ინფორმირება საკუთარი მდგომარეობის შეცვლის თაობაზე. მაგალითად, მომხმარებლის მიერ კლავიატურაზე სიმბოლოს აკრეფისას კლავიატურა აგენერირებს წყვეტის სიგნალს.

წყვეტა არის ყველაზე ნაკლები რესურსმომთხოვნი მექანიზმი პროცესორის ყურადღების მისაპყრობად. წყვეტის ალტერნატივას წარმოადგენს პროცესორისთვის პერიოდულად ყველა

მოწყობილობის შემოწმების მოთხოვნის წაყენება. მსგავსი მიდგომა, რომელიც ცნობილია მიმდევრობითი გამოკითხვის (pooling) სახელით, იწვევს ზედნადებ დანახარჯებს და ზრდის კომპიუტერული სისტემის სირთულეს. წყვეტა პროცესორს ათავისუფლებს მოწყობილობების მუდმივი გამოკითხვისგან.

წყვეტის გამომყენებელი სისტემა შეიძლება იყოს ხშირი გადატვირთვის საფრთხის ქვეშ, თუ სისტემაში ფიქსირდება დიდი რაოდენობით წყვეტები, რომელთა დამუშავებასაც ვერ აუდის. კომპიუტერის ქსელურ ინტერფეისს გააჩნია მცირე მოცულობის მეხსიერება, რომელშიც ინახება ქსელიდან შემოსული მონაცემები. ყოველთვის როცა ქსელიდან შემოდის ახალი მონაცემები ქსელური ინტერფეისი აგენერირებს წყვეტას რომლითაც ატყობინებს პროცესორს დასამუშავებლად მონაცემების მზადყოფნაზე. თუ პროცესორი ვერ იქნება მზად ქსელური ინტერფეისიდან მომავალი მონაცემების დასამუშავებლად, მაშინ ის არ მიიღებს ამ მონაცემებს და ქსელური ინტერფეისის მეხსიერება გადაივსება, მასში განთავსებული მონაცემები კი ნაწილობრივ ან სრულად დაიკარგება. ჩვეულებრივ, სისტემები დასამუშავებელად შემოსული წყვეტების შესანახად იყენებენ მეხსიერების მისამართების მიმდევრობებს. კომპიუტერის პიკური მუშაობის მომენტში შესაბამისი მეხსიერება შეიძლება არ აღმოჩნდეს საკმარისი შემოსული წყვეტების შესანახად და შედეგად ზოგიერთი მათგანი დაიკარგოს.

2.4.1. წყვეტის დამუშავება

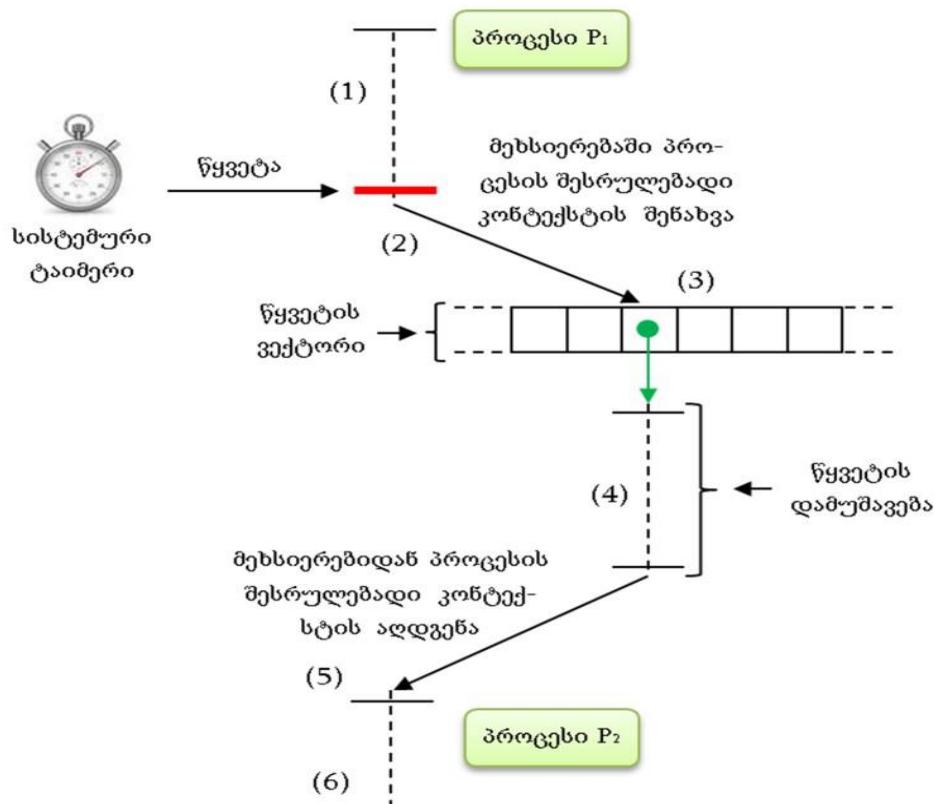
წყვეტის თაობაზე პროცესორის ინფორმირებისათვის კომპიუტერის დედაპლატაზე გაყვანილია სპეციალური ხაზები პროცესორამდე, რომლებშიც გადაიცემა წყვეტის შესაბამისი სიგნალები. წყვეტა შეიძლება გამოწვეული იყოს, მაგალითად, სისტემური ტაიმერების, პერიფერიული პლატების, მოწყობილობების კონტროლერების და ა.შ. მიერ. კონკრეტული მოწყობილობა წყვეტის საჭიროების შემთხვევაში (მაგალითად, დროითი კვანტის ამოწურვა ან შეტანა/გამოტანის ოპერაციის დასრულება) აგენერირებს შესაბამის სიგნალს და გადასცემს მას აღნიშნული ხაზების გამოყენებით. ყველა პროცესორზე წყვეტის დამუშავების მიზნით განთავსებულია ე.წ. წყვეტის კონტროლერი, რომელიც წყვეტაზე შემოსულ სიგნალებს ახარისხებს პრიორიტეტების მიხედვით და უზრუნველყოფს მათ დამუშავებას პრიორიტეტის გათვალისწინებით.

აღვწეროთ პროცესი თუ როგორ ხორციელდება აპარატული წყვეტის დამუშავება:

1. წყვეტის საჭიროების შემთხვევაში მოწყობილობა აგენერირებს შესაბამისი სიგნალს და გადასცემს მას სპეციალურ ხაზებში;
2. ხაზებში გადაცემული სიგნალის აღმოჩნდის (ხაზის გააქტიურების) შემთხვევაში პროცესორი აჩერებს პროცესის მიმდინარე ინსტრუქციის შესრულებას (ანუ აჩერებს პროცესს) და ინახავს რეგისტრებში განთავსებულ ინფორმაციას;
3. პროცესორი მართვას გადასცემს ოპერაციულ სისტემას, რომელიც შემოსული სიგნალის დამუშვების მიზნით ეძებს წყვეტის ვექტორის შესაბამის ელემენს.
4. წყვეტის დამუშავებელი პროგრამა (წყვეტის ტიპზე დამოკიდებულებით) ასრულებს შესაბამის ინსტრუქციებს;
5. წყვეტის დამმუშავებელი პროგრამის დასრულების შემდეგ შეჩერებული პროცესი (თუ ბირთვში არ დაფიქსირებულა კონტექსტის გადართვა) ან სხვა რომელიმე პროცესი გადაყვანილი იქნება შესრულების მდგომარეობაში;
6. განახლებული პროცესი აგრძელებს შესრულებას.

ნახ. 2.6-ზე ნაჩვენებია ოპერაციული სისტემის და აპარატული უზრუნველყოფის

ურთიერთქმედება სისტემური ტაიმერის მიერ გენერირებული წყვეტის დამუშავებისას. სისტემურმა ტაიმერმა დააგენერირა წყვეტა (მაგალითად, პროცესის საჭიროებს შეტანა/გამოტანის ოპერაციას) და მართვა გადაეცა ოპერაციულ სისტემას. ამ შემთხვევაში ხდება P₁ პროცესის შეჩერება და პროცესორის რეგისტრებში არსებული ინფორმაციის შენახვა (2). შემდეგ პროცესორი მართვას გადასცემს წყვეტის დამმუშავებელ პროგრამას (3). წყვეტის დამმუშავებელი პროგრამა გენერირებული სიგნალის მიხედვით ირჩევს წყვეტის ვექტორის შესაბამის ელემენტს და ახდენს წყვეტის დამუშავებას (4). წყვეტის დამმუშავებელი პროგრამის დარსულების მომენტისთვის თუ არ დაფიქსირება კონტექსტის გადართვა, მაშინ მიმდინარე პროცესის (P₁) შესაბამისი ინფორმაცია ჩაიტვირთება პროცესორის რეგისტრებში. წინააღმდეგ შემთხვევაში გამოყენებული პროცესორის მუშაობის დამგეგმავის შესაბამისი ალგო-რითმით შერჩეული პროცესის (P₂) მონაცემები იქნება ჩატვირთული პრო-ცესორის რეგისტრებში (5). შერჩეული პროცესი დაიკავებს პროცესორს და გააგრძელებს შესრულებას (6).



ნახ. 2.6. წყვეტის დამუშავება

2.4.2. წყვეტის კლასები

წყვეტის ნაირსახეობა დამოკიდებულია გამოყენებულ არქიტექტურაზე. ზოგიერთი წყვეტა გამოიყენება თითქმის ყველა არქიტექტურაში. ჩვენი შემდგომი საუბარი შეეხება Intel IA-32 არქიტექტურაში განსაზღვრულ წყვეტებს.

IA-32 სფეციფიკაცია განასხვავებს ორი ტიპის სიგნალს, რომლის მიღებაც შეუძლია პროცესორს: წყვეტა და განსაკუთრებული შემთხვევა. წყვეტა ახდენს პროცესორის ინფორმირებას ამა თუ იმ მოვლენის დადგომის ან მოწყობილობის სტატუსის შეცვლის თაობაზე. IA-32 არქიტექტურაში გათვალისწინებულია პროგრამული წყვეტა, რომელსაც იყენებს პროცესი სისტემური ფუნქციის გამოძახების მიზნით. განსაკუთრებული შემთხვევა ახდენს პროგრამული ან აპარატული შეცდომის შესახებ სიგნალიზირებას. გარდა ამისა, განსაკუთრებული შემთხვევა გამოიყენება კოდის შიგნით

გარკვეულ წერტილის მიღწევისას პროცესის საქმიანობის შესაჩერებლად.

მოწყობილობა, რომელიც აგენერირებს წყვეტას შეტანა/გამოტანის სიგნალის ფორმით და ტაიმერიდან წყვეტის ფორმით, არის შიდა პროცესორთან მიმართებაში. მსგავსი სიგნალები არაა სინქრონული შესრულებად პროცესთან მიმართებაში, ვინაიდან მათი ფორმირება არაფრით არაა დაკავშირებული პროცესორის მიერ შესრულებულ ინსტრუქციებთან. წყვეტის პროგრამირება, მაგალითად, სისტემური ფუნქციის გამოძახება, კი პირიერი, სინქრონულია შესრულებად პროცესთან მიმართებაში. ცხრილ 2.1-ში მოყვანილია წყვეტის ზოგიერთი ტიპები, რომლებიც განსხვავებულია IA 32 არქიტექტურაში.

ცხრილი 2.1. IA 32 სკეციფიკაციაში განსაზღვრული განსხვავებული წყვეტის ტიპები

ტიპი	აღწერა
შეტანა/გამოტანა	შეიძლება გენერირებული იყოს შეტანა/გამოტანის მოწყობილობის მიერ შესაბამისი ოპერაციის დასრულების დროს. ის აინფორმირებს პროცესორს არხის ან მოწყობილობის მდგომარეობის შეცვლის შესახებ
ტაიმერი	სისტემაში შეიძლება არსებობდეს მოწყობილობები, რომლებიც გარკვეული დროითი შუალედის გავლის შემდეგ აგენერირებენ წყვეტას. ასეთი წყვეტა შეიძლება გამოყენებული იქნას სისტემის წარმადობის მონიტორინგის ან სხვა მიზნით. მაგალითად, ოპერაციული სისტემა იყენებს სისტემურ ტაიმერს პროცესისთვის დროითი კვანტის ამოწურვაზე პროცესორის ინფორმირების მიზნით
პროცესორთაშორისი	ამ ტიპის წყვეტა მრავალპროცესორულ სისტემაში იძლევა ერთი პროცესორის მიერ მეორისთვის შეტყობინების გადაცემის საშუალებას

IA 32 არქიტექტურაში განსაზღვრულია განსაკუთრებული შემთხვევების შემდეგი კლასები: **აცდენა** (faults), **ხაფანგი** (traps) და **ავარიული დასრულება** (aborts) (ცხრილი 2.2). აცდენა და ხაფანგი განსაკუთრებული შემთხვევის დამმუშავებელ პროგრამას აძლევს საშუალებას გააგრძელოს პროცესის შესრულება. აცდენა ეს არის შეცდომა, რომელიც შეიძლება კორექტირებული იქნას განსაკუთრებული შემთხვევის დამმუშავებელი პროგრამის მიერ. მაგალითად, ფურცლის აცდენა წარმოიშობა, მაშინ როცა პროცესი ცდილობს მიმართოს მონაცემებს მესიერებაში არარსებულ ფურცელზე. ოპერაციულ სისტემას შეუძლია ამ შეცდომის გამოსწორება მოთხოვნილი მონაცემების მეხსიერებაში გადმოტანით. შეცდომის გამოსწორების შემდეგ პროცესორი აგრძელებს პროცესის შესრულებას იმ ინსტრუქციიდან, რომელმაც წარმოშო განსაკუთრებული შემთხვევა.

ცხრილი 2.2. IA 32 სკეციფიკაციაში განსაზღვრული განსხვავებული განსაკუთრებული

შემთხვევების კლასები

კლასი	აღწერა
აცდენა	შეიძლება გამოწვეული იყოს სხვადასხვა მიზეზით პროგრამის ინსტრუქციების შესრულების მომენტში. მაგალითად, ასეთი შეიძლება იყოს: ნულზე გაყოფა, მონაცემთა არასწორი ფორმატი, აკრძალული მოქმედების შესრულება, აკრძალულ რესურსებზე მიმართვა, მიმართვა დასაშვები შუალედის გარეთ
ხაფანგი	შეიძლება გამოწვეული იყოს გადავსებით გამოწვეული შეცდომით, ან პროგრამი ისეთი ადგილის მიღწევით, რომელიც იწვევს მის შეჩერებას
ავარიული დასრულება	მიიღწევა პროცესორის მიერ ისეთი შეცდომის აღმოჩენის შემთხვევაში, რომლის შემდეგაც პროცესი ვერ შეძლებს გაგრძელებას. მაგალითად, როდესაც განსაკუთრებული სიტუაციის დამმუშავებელი პროგრამის მუშაობისას წარმოიშობა განსაკუთრებული სიტუაცია

ხაფანგი არ განეკუთვნება კორექტირებად შეცდომებს, ის დაკავშირებულია გადავსების ან შეჩერების შემთხვევასთან. მაგალითად, როცა პროცესი გადასცემს პროცესორს ბრძანებას ისეთი გამოსახულების გამოთვლაზე, რომლის შედეგიც სცდება გამოყენებული ცვლადის ტიპის დასაშვები მნიშვნელობის საზღვრებს. ამ შემთხვევაში ოპერაციული სისტემა ატყობინებს პროცესორს გადავსების შეცდომის თაობაზე. განსაკუთრებული შემთხვევის დამმუშავებელ პროგრამის მუშაობის დასრულების შემდეგ პროცესორი აგრძელებს პროცესის შესრულებას განსაკუთრებული შემთხვევის გამომწვევი ინსტრუქციის შემდეგი ინსტრუქციიდან.

ავარიული დასრულება წარმოიშობა, როცა პროცესი არ შეუძლია შესრულების გაგრძელება (მაგალითად, აპარატურის მტყუნების შემთხვევაში) და ოპერაციული სისტემა აჩერებს პროცესის შესრულებას. ამასთან, პროცესორი ვერ გარანტირებს პროცესის შესრულების კონტექსტის საიმედო შენახვას.

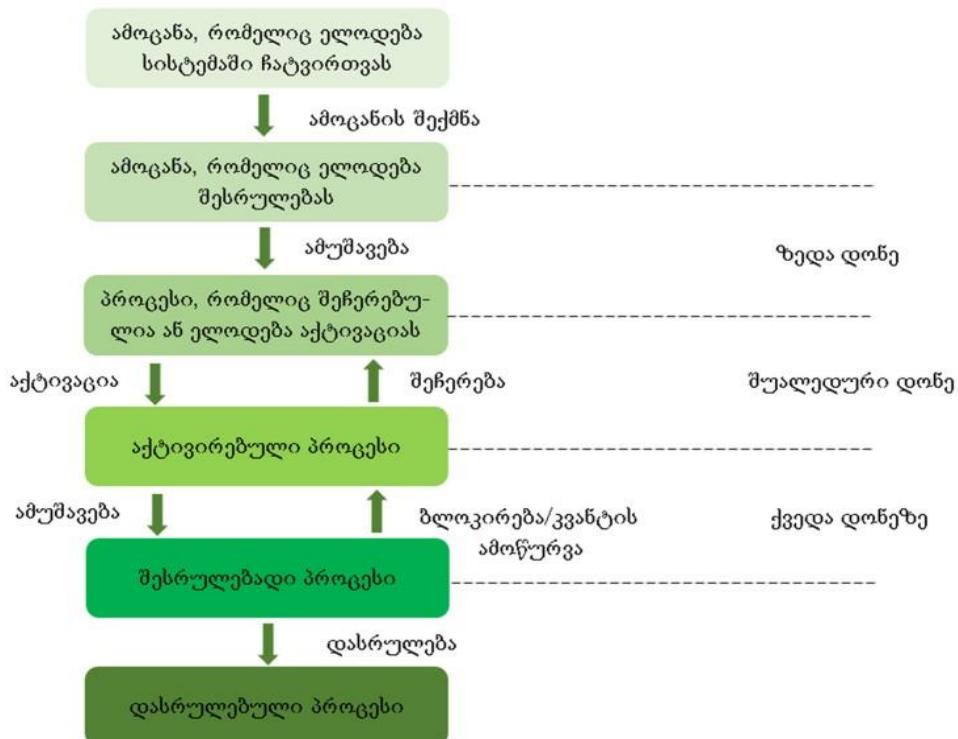
ზოგიერთი წყვეტა შესაძლებელია საჭიროებდეს დაუყოვნებლივ დამუშავებას, მაშინ როცა სხვების დამუშავება შეიძლება გადაიდოს გარკვეული დროითი შუალედით. მაგალითად, კლავიატურის მიერ გამოწვეული წყვეტა მნიშვნელოვანია და ის საჭიროებს დაუყოვნებელ დამუშავებას. ოპერაციულ სისტემაში წყვეტისთვის მნიშვნელობის მისანიჭებლად გამოიყენება პრიორიტეტები. წყვეტების პრიორიტეტების რეალიზება შესაძლებელია როგორც პროგრამულად ისე აპარატულად.

ლუქცია 3. პროცესების დაგეგმვა

მრავალამოცანიანობა ოპერაციული სისტემისთვის იძლევა კომპიუტერული სისტემის რესურსების ეფექტური გამოიყენების შესაძლებლობას. როდესაც ოპერაციულ სისტემას უწევს არჩევანის გაკეთება, თუ რომელ პროცესს გამოუყოს პროცესორი, მან უნდა ისარგებლოს გარკვეული სტრატეგით, რომელსაც პროცესორის მუშაობის დამგეგმვი (processor scheduling policy) ეწოდება. ის განსაზღვრავს პროცესების შესრულების თანმიმდევრობას. დაგეგმვის პოლიტიკა უნდა აკმაყოფილებდეს წარმადობის გარკვეულ კრიტერიუმებს. მაგალითად, როგორიცაა შეეცადოს გაზარდოს პროცესების რაოდენობა, რომლებიც დასრულებას შეძლებენ დროის მცირე შუალედში (წარმადობა), შეეცადოს შეამციროს პროცესების ჩასატვირთად საჭირო დრო (ლოდინი), შეეცადოს არ გაზარდოს პროცესის დასასრულებლად საჭირო (რეალური) დრო.

3.1. დაგეგმვის დონეები

გამოთვლით სისტემაში პროცესორის დაგეგმვის არსებობს რამდენიმე დონე: ზედა, შუალედური და ქვედა (ნახ. 3.1). ზედა დონის დაგეგმვის პოლიტიკა, რომელსაც ხშირად ამოცანის დაგეგმვას (job scheduling) ან გრძელვადიან დეგეგმვას (long-term scheduling) უწოდებენ, განსაზღვრავს, თუ რომელი ამოცანები ჩაეტანება აქტიურ კონკურენციაში სისტემური რესურსების მოპოვებაზე. შექმნილი ამოცანა იქცევა პროცესად ან პროცესების ჯგუფად. დაგეგმვის ზედა დონეზე განისაზღვრება მრავალამოცანიანობის ხარისხი (ანუ დროის მოცემულ მომენტში პროცესების საერთო რაოდენობა სისტემაში). სისტემაში დიდი რაოდენობით პროცესების არსებობა გამოიწვევს სისტემის ყველა რესურსის დაკავებას, რაც მიგვიყვანს სისტემის წარმადობის შემცირებამდე. ამ შემთხვევაში ზედა დონეზე დაგეგმვის პოლიტიკას უნდა შეეძლოს სისტემაში ახალი ამოცანების დროებით გამოჩენის შეზღუდვა, სანამ არ დასრულდება რამდენიმე უკვე არსებული ამოცანა.



ნახ. 3.1. დაგეგმვის დონეები

ზედა დონეზე ამოცანის შექმნის შემდეგ ის გადადის შუალედურ დონეზე (intermediate-level scheduling). ამ დონის პოლიტიკა განსაზღვრავს, თუ რომელი პროცესები გაუწევენ ერთმანეთს კონკურენციას პროცესორის მოპოვებაზე. შუალედური დონის დამგეგმავი ოპერატიულად რეაგირებს სისტემის დატვირთულობის ცვალებადობაზე, პროცესების შესრულების მცირე დროითი შუალედით შეჩერებითა და განახლებით, რათა უზრუნველყოს სისტემის თანაბარი დატვირთულობა.

ქვედა დონეზე დაგეგმვის (low-level scheduling) პოლიტიკა განსაზღვრავს მზა პროცესებიდან რომელს გამოყენებულის გამოთავისუფლებული პროცესორი. მრავალ თანამედროვე სისტემაში გამოყენებულია მხოლოდ შუალედური და დაბალი დონის დამგეგმავები. მაღალი დონის დამგეგმავი გამოიყენება მსხვილ მეინფრეიმულ სისტემებში, რომლებიც ჰაკეტური ამოცანების დამუშავებისთვის გამოიყენება.

ქვედა დონის დაგეგმვის პოლიტიკა ხშირად ანიჭებს პრიორიტეტს (priority), რომელიც მის მნიშვნელოვნობაზე მიუთითებს. რაც უფრო მნიშვნელოვანია პროცესი მით უფრო მეტია ალბათობა იმისა, რომ ის იქნება არჩეული შესასრულებლად. ქვედა დონის დამგეგმავი, რომელსაც დისპეტჩერი (dispatcher) ეწოდება, ანხორციელებს პროცესებისთვის პროცესორის გამოყოფას. ის წამში რამდენჯერმე სრულდება და ამიტომ მუდმივად უნდა იმყოფებოდეს ოპერატიულ მეხსიერებაში.

3.2. დაგეგმვა პრიორიტეტული გაძევებით და მის გარეშე

დაგეგმვის პოლიტიკამ შეიძლება დაუშვას პრიორიტეტული გაძევება ან შეზღუდოს ის. თუ პროცესისთვის პროცესორის გადაცემის შემდეგ მისთვის პროცესორის ჩამორთმევა (დროითი კვანტის ამოწურვამე) შეუძლებელია, მაშინ ამბობენ, რომ მოცემულია დაგეგმვის პოლიტიკა არაპრიორიტეტული გაძევებით (nonpreemptive). წინააღმდეგ შემთხვევაში ამბობენ, რომ მოცემულია დაგეგმვის პოლიტიკა პრიორიტეტული გაძევებით (preemptive). არაპრიორიტეტული გაძევებით დაგეგმვის პოლიტიკის გამოყენებისას პროცესი, რომელიც იკავებს პროცესორს, ნებაყოფლობით აბრუნებს მას დასრულების ან დროითი კვანტის ამოწურვის შემდეგ. პრიორიტეტული გაძევებით დაგეგმვის პოლიტიკის გამოყენებისას კი პირიქით, სისტემაში მაღალპრიორიტეტული პროცესის გამოჩენის შემთხვევაში მიმდინარე პროცესს (იძულებით) ჩამოერთმევა პროცესორი და ის გადაეცემა მაღალპრიორიტეტულ პროცესს.

პრიორიტეტული გაძევებით ალგორითმის გამოყენება საჭიროა ისეთ სისტემებში, რომელშიც გამოყენებულია პრიორიტეტები. მაგალითად, რეალური დროის სისტემებში წყვეტის ერთი სიგნალის დაკარგვამ შეიძლება გამოიწვიოს კატასტროფული შედეგები. ინტერაქტიულ სისტემებში პრიორიტეტული გაძევებით ალგორითმის გამოყენება უზრუნველყოფს გამოხმაურების ოპტიმალურ დროს.

არაპრიორიტეტული გაძევებით ალგორითმის გამოყენებისას ამოცანებს შეიძლება მოუწიოთ დიდი ხნით ლოდინი (დიდი ამოცანების შემთხვევაში), თუმცა ამ შემთხვევაში დამუშავების ციკლების ხანგრძლივობის წინასწარმეტყველება იქნება შესაძლებელი. ვინაიდან ამ შემთხვევაში პროცესორის ჩამორთმევა შესრულებადი პროცესისთვის შეუძლებელია, ამიტომ პროგრამები, რომლებიც შეიძლება შეიცავენ დიდი რაოდენობით შეცდომებს, დასრულებამდე საკუთარი სურვილით არ გამოათავისუფლებენ პროცესორს (ასეთი პროცესი შეიძლება საერთოდ არ დასრულდეს (ჩაიციკლოს)). ასევე, ამ შემთხვევაში დაბალი პრიორიტეტის მქონე პროცესის

შესრულება შეიძლება უსასრულოდ გადაიდოს.

იმისთვის, რომ არ მოხდეს სისტემის (შემთხვევთი ან მიზანმიმართული) მონოპოლური დასაკუთრება, პრიორიტეტული გაძევებით დაგეგმვით სისტემებს უნდა შეეძლოთ პროცესისთვის პროცესორის ჩამორთმევა. პროცესისთვის პროცესორის გამოყოფის შემდეგ თუ პროცესმა ნებაყოფლობით არ დააბრუნა პროცესორი, მაშინ სისტემამ მასში გამოყენებული მექანიზმის ამუშავებით უნდა განახორციელოს პროცესორის იძულებითი ჩამორთმევა და განსაზღვროს ახალი პროცესი, რომელსაც გადასცემს მას.

3.3. პრიორიტეტები

დაგეგმვა ხშირად იყენებს პრიორიტეტებს იმისთვის, რომ განსაზღვროს ამა თუ იმ სიტუაციაში თუ რომელ პროცესს გამოუყოს პროცესორი. პრიორიტეტის გამოყენებით განისაზღვრება პროცესების პირობითი მნიშვნელოვნობა. პროცესისთვის პრიორიტეტის დანიშვნა შესაძლებელია სტატიკურად და დინამიურად.

სტატიკური პრიორიტეტი (static priorities) არ შეიძლება შეიცვალოს. მისი დანიშვნის მექანიზმის რეალიზება სისტემაში ადვილია და იწვევს სისტემის შედარებით ნაკლებ დაყოვნებას. თუმცა ისინი არ რეაგირებენ გარშემო სიტუაციების ცვლილებაზე, რომლებმაც შესაძლებელია განახორციელონ პრიორიტეტების კორექტირება და გამოიწვიონ დაყოვნების შემცირება.

დინამიური პრიორიტეტები (dynamic priorities) რეაგირებენ სიტუაციის ცვალებადობაზე. მაგალითად, სისტემამ შესაძლებელია აამაღლოს პროცესის პრიორიტეტი, თუ მას დაკავებული აქვს მაღალპრიორიტეტული პროცესისთვის საჭირო მნიშვნელოვანი რესურსი, ხოლო მას შემდეგ რაც დაბალპრიორიტეტული პროცესი გამოათავისუფლებს შესაბამის რესურსს მისი პრიორიტეტის მნიშვნელობა დააბრუნოს საწყის მნიშვნელობამდე. დინამიური პრიორიტეტების სქემის რეალიზება, სტატიკურ სქემებთან მიმართებაში, რთულია და იძლევა მეტ დაყოვნებას. თუმცა რიგ შემთხვევებში ასეთი დაყოვნებები სხვადასხვა მიზეზის გამო შეიძლება იყოს გამართლებული.

მრავალმომხმარებლიან სისტემებში ოპერაციული სისტემა ხარისხიანად უნდა მოემსახუროს სხვადასხვა მომხმარებელს. ასეთ სისტემაში მომხმარებლებს უწევთ სხვადასხვა სირთულის და მნიშვნელოვნობის ამოცანის გადაწყვეტა. ყოველი ამოცანა საჭიროებს გარკვეული რესურსების ჩამონათვალს, გადაწყვეტის გარკვეულ სისტრაფეს და ა.შ. ასეთ სისტემებში უნდა არსებობდეს მომხმარებლების პრიორიტეტით განსხვავების და პრიორიტეტული მომხმარებლისთვის რესურსებზე შესაბამისი წვდომის უზრუნველყოფის მექანიზმები. მაგალითად, მომხმარებელთა პრიორიტეტებად დაყოფის მექანიზმად შესაძლებელია გამოყენებული იქნას შესაბამის მომსახურეობაზე გარკვეული საზღაურის (პრიორიტეტის ყიდვა (purchase priority)) დაწესება.

3.4. დაგეგმვის მიზნები

სისტემის შემქმნელებს პროცესების დაგეგმვის მექანიზმების შემუშავებისას უწევთ მრავალი ფაქტორის გათვალისწინება, მათ შორისაა სისტემის ტიპი და მომხმარებლის მოთხოვნილება. რეალური დროის სისტემების დაგეგმვის ალგორითმები უნდა განსხვავდებოდეს ინტერაქტიული სისტემების დაგეგმვის ალგორითმებისგან, ვინაიდან მომხმარებლი შესაბამისი სისტემისგან მოითხოვს რეალურ შედეგს. სისტემის ტიპზე დამოკიდებულებით დამგეგმავს უნდა შეეძლოს:

- სისტემის მაქსიმალური გამტარუნარიანობის ამაღლების უზრუნველყოფა. დაგეგმვის ალგორითმმა უნდა შეძლოს დროის გარკვეულ შუალედში მაქსიმალური რაოდენობის პროცესების მომსახურება;
- ინტერაქტიულ რეჟიმში მომუშავე მაქსიმალური რაოდენობის მომხმარებლებისთვის გამოხმაურების მისაღები დროის გარანტირება;
- უზრუნველყოს სისტემის რესურსების მაქსიმალური თანაბარი დატვირთვა. დაგეგმვის მექანიზმმა უნდა განახორციელოს რესურსების შეძლებისდაგვარად თანაბარი გამოყენება;
- უსასრულო ლოდინის გამორიცხვა. პროცესის შესრულების გადადება არ უნდა ხდებოდეს უსასრულოდ;
- პრიორიტეტების გათვალისწინება. სისტემებში, რომლებშიც მხარდაჭერილია პრიორიტეტი, სისტემამ უნდა უზრუნველყოს პრიორიტეტული პროცესებისთვის უპირატესობის მინიჭება;
- ზედნადები დანახარჯების მინიმიზირება. სისტემა უნდა შეეცადოს პროცესს გამოუყოს მხოლოდ ის რესურსი, რომელიც საჭიროა მიმდინარე მომენტში შესასრულებლად;
- უზრუნველყოს წინასწარმეტყველებადობა. რეალური დროის სისტემებში პროცესის შესრულებისას ხანდახან საჭიროა შედეგის წინასწარმეტყველების შესაძლებლობა.

სისტემამ ამ მიზნების რეალიზება შეიძლება განახორციელოს სხვადასხვა მეთოდებით. რიგ შემთხვევაში დამგეგმავი პროცესის უსასრულო ლოდინს იცილებს მისი „სიძველის“ გათვალისწინებით. პერიოდულად იზრდება პროცესის პრიორიტეტი სანამ ის არ დასრულდება. დამგეგმავმა შეიძლება აამაღლოს სისტემის წარმადობა იმ პროცესებისთვის უპირატესობის მინიჭებით, რომლებიც მარტივად შეიძლება იყოს შესრულებული, ან შეიძლება მისგან გამოთავისულფდება ისეთი რესურსები, რომელთა გამოყოფა სხვა პროცესებისთვის არის შესაძლებელი. ერთერთი ასეთი სტრატეგია გულისხმობს იმ პროცესებისთვის უპირატესობის მინიჭებას, რომლებიც იკავებენ მნიშვნელოვან რესურსებს. მაგალითად, შეიძლება წარმოიშვას სიტუაცია, რომლის დროსაც დაბალპრიორიტეტულ პროცესს დაკავებული აქვს რესურსი, რომელსაც საჭიროებს მაღალპრიორიტეტული პროცესი. თუ ეს რესურსი არ არის განაწილებადი, მაშინ დამგეგმავმა უნდა შესთავაზოს დაბალპრიორიტეტულ პროცესს მის დასასრულებლად საჭირო რესურსები, რათა მან გამოათავისუფლოს მაღალპრიორიტეტული პროცესისთვის საჭირო რესურსი. ამ მიდგომას ეწოდება პრიორიტეტების ინვერსია (priority inversion), ვინაიდან პროცესების მიმართებითი პრიორიტეტი ცვლიან ადგილებს, რომლის მეშვეობითაც მაღალპრიორიტეტული პროცესი ახერხებს მისთვის საჭირო რესურსის მიღებას.

ზემოთ ჩამოთვლილი თვისებებიდან ზოგიერთი წინააღმდეგობაში მოდის ერთიმეორესთან. მაგალითად, ინტერაქტიულ სისტემაში გამოხმაურების მცირე დროის გარანტირება ნიშნავს საკმარისი რაოდენობით თავისუფალი რესურსების ქონას, რომელთა გამოყოფა, საჭიროების შემთხვევაში, შესაძლებელია პროცესებისთვის. მსგავსი სტრატეგია ამცირებს რესურსის გამოყენების კოეფიციენტს (ინტენსივობას). რეალური დროის სისტემებში, სადაც მოითხოვება სისტემის „რეაქტიულობა“ და გამოხმაურების დროის წინასწარმეტყველება, რესურსების გამოყენების ეფექტურობა ნაკლებად მნიშვნელოვანია. სხვა სისტემებში კი პირიქით.

მიუხედავად სისტემებს შორის განსხვავებისა, დაგეგმვის პოლიტიკას მაინც გააჩნია მსგავსი თვისებები:

- სამართლიანობა (fairness). დაგეგმვის პოლიტიკა სამართლიანია, თუ ყველა პროცესს ის ექცევა ერთნაირად და არ ხდება პროცესის შესრულების უსასრულოდ გადადება დამგეგმავის მიზეზით;
- წინასწარმეტყველებადობა (predictability). სისტემის თანაბარი დატვირთულობის

შემთხვევაში პროცესის შესრულება არ უნდა საჭიროებდეს განსხვავებულ დროს სხვადასხვა დროს შესრულებისას;

- **მასშტაბირებადობა (scalability).** სისტემის დატვირთულობის შემთხვევაში დაგეგმვამ არ უნდა დაკარგოს ქმედუნარიანობა.

3.5. დაგეგმვის კრიტერიუმები

იმისთვის, რომ მოხდეს ზემოთ ჩამოთვლილი დაგეგმვის მიზნების რეალიზება, დაგეგმვის მექანიზმა უნდა გაითვალისწინოს პროცესის ყოფაქცევა. გამოთვლით სისტემაში არსებული პროცესი შეიძლება დაკავებული იყოს ინტენსიური გამოთვლებით (processor-bound) ან ინტენსიური შეტანა/გამოტანით (I/O-bound). ცხადია, რომ ინტენსიური გამოთვლების საჭიროების მქონე პროცესი მისთვის პროცესორის გამოყოფის შემთხვევაში ძირითადად დაკავებულია აქტიური გამოთვლებით და შეიძლება ანხორციელებს მცირე რაოდენობის შეტანა/გამოტანის ოპერაციებს. ინტენსიური შეტანა/გამოტანის ოპერაციის საჭიროების მქონე პროცესი კი პირიქით, ძირითადად დაკავებულია შეტანა/გამოტანის ოპერაციების განხორციელებით და შესაძლებელია საჭიროებდეს უმნიშვნელო გამოთვლებს.

დაგეგმვის დისციპლინა უნდა ითვალისწინებდეს სისტემის ტიპს - პაკეტურია ის თუ ინტერაქტიული. პაკეტური პროცესი (batch process) შეიცავს ისეთ ამოცანებს, რომელიც შესასრულებლად არ საჭიროებს ადამიანის ჩარევას. ინტერაქტიული პროცესი (interactive process) კი მუდმივად საჭიროებს მომხმარებლისგან მონაცემების შეტანას. სისტემამ ინტერაქტიული პროცესებისთვის უნდა უზრუნველყოს სწრაფი გამოხმაურება.

3.6. დაგეგმვის ალგორითმები

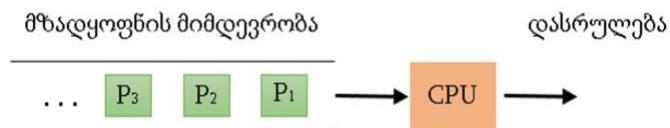
არსებობს დაგეგმვის ალგორითმების მრავალფეროვნება, რომლებიც გამოიყენება სხვადასხვა ტიპის ამოცანების გადასაწყვეტად და ეფექტურობის ხარისხის მისაღწევად. მრავალი მათგანი შესაძლებელია გამოყენებული იქნას დაგეგმვის სხვადასხვა დონეზე. განვიხილოთ ზოგიერთი მათგანი.

3.6.1. FCFS (First Come - First Served)

დაგეგმვის ყველაზე მარტივ ალგორითმს წარმოადგენს ალგორითმი FCFS (ნახ. 3.2). ამ ალგორითმის გამოყენებისას სისტემაში იქმნება „მზაყოფნა“ მდგომარეობაში მყოფი პროცესების მიმდევრობა. სისტემაში წარმოქმნილი ყოველი ახალი პროცესი ამ მიმდევრობაში ემატება ბოლოდან. შესრულებას იწყებს მიმდევრობის თავში მყოფი პირველივე პროცესი. პროცესს პროცესორი გამოყოფა იმ დროითი შუალედით რამდენიც საჭიროა მის შესასრულებლად. პროცესი პროცესორს გამოათავისუფლებს მხოლოდ იმ შემთხვევაში, თუ ის დასრულდა ან გარკვეული მიზეზით (შეტანა/გამოტანის ოპერაციის ლოდინის გამო) მოხდა მისი ბლოკირება. მას შემდეგ რაც სისტემაში დაფიქსირდება მოვლენა, რომელსაც ბლოკირებული პროცესი ელოდებოდა, ის გადადის მდგომარეობაში მზადყოფნა და მიმდევრობას ემატება ბოლოდან.

ამ ალგორითმის უპირატესობა მდგომარეობს იმაში, რომ მისი რეალიზაცია და დაპროგრამება მარტივია. სისტემაში წარმოქმნილი ყოველი პროცესი იმართება ერთი მიმდევრობით. ამ

მიმდევრობიდან შესასრულებლად ახალი პროცესის არჩევა ხორციელდება მიმდევრობის თავიდან, ხოლო სისტემაში ახალი პროცესის წარმოქმნისას ის ემატება მიმდევრობის ბოლოში.



ნახ. 3.2. დაგეგმვა FIFO პრინციპის გამოყენებით

ალგორითმს, ასევე, გააჩნია ნაკლოვანება. დავუშვათ სისტემაში გვაქვს ინტენსიური გამოთვლების საჭიროების მქონე 1 პროცესი, რომელიც ყოველი ამუშავებისას პროცესორთან იმყოფება 1 წმ-ის განმავლობაში და რამდენიმე ინტენსიური შეტანა/გამოტანის საჭიროების მქონე პროცესი, რომელიც პროცესორთან მუშაობს მცირე დროით, მაგრამ საჭიროებს 1000 კითხვის ოპერაციას. ინტენსიური გამოთვლების საჭიროების მქონე პროცესი 1 წმ პროცესორთან დაყოვნების შემდეგ იწყებს მონაცემების კითხვას დისკიდან. შემდეგ იტვირთება მიმდევრობიდან ინტენსიური შეტანა/გამოტანის საჭიროების მქონე პროცესები და ასე გრძელდება სანამ არ დასრულდება ყველა პროცესი. თუკი დავუშვებთ, რომ ინტენსიური შეტანა/გამოტანის საჭიროების მქონე პროცესი დისკიდან მონაცემთა ერთი ბლოკის კითხვას ანდომებს 1 წმ-ს, მაშინ მივიღებთ, რომ ის საკუთარ სამუშაოს დასასრულებლად საჭიროებს 1000 წმ-ს. თუკი გამოვიყენებდით დაგეგმვის ისეთ ალგორითმს, რომელიც შეძლებდა ინტენსიური გამოთვლების საჭიროების მქონე პროცესის ბლოკირებას ყოველ 10 წმ-ში, მაშინ ინტენსიური შეტანა/გამოტანის საჭიროების მქონე პროცესები ნაცვლად 1000 წმ-სა დასრულდებოდნენ 100 წმ-ში, რაც არ გამოიწვევდა სისტემის საგრძნობ შენელებას.

3.6.2. SJF (Short Job First)

ბუნებრივია, რომ თუკი სისტემაში ყველა პროცესი იარსებებდა ერთდროულად და წინასწარ ცნობილი იქნებოდა მათი შესრულებისთვის საჭირო პროცესორული დრო, მაშინ მარტივი იქნებოდა პროცესების დაგეგმვის ამოცანაც. SJF ალგორითმი აგებულია ამ პრინციპით. SJF ალგორითმის გამოყენება გულისხმობს დამგეგმავის მიერ შესასრულებლად პირველი არჩეული იყოს მოკლე (ნაკლები პროცესორული დროის საჭიროების მქონე) ამოცანა. ამ ალგორითმშიც გამოიყენება ერთი მიმდევრობა. მიმდევრობა დალაგებულია ზრდადობით შესასრულებლად საჭირო დროითი შუალედის მიხედვით. განვიხილოთ მაგალითი. ნახ. 3.3 ა)- ზე გამოსახულია 4 პროცესისგან შემდგარი მიმდევრობა, შესასრულებლად საჭირო დროითი შუალედებით 8, 4, 4, 4 დროითი ერთეული შესაბამისად. წარმოდგენილი მიმდევრობით პროცესების შესასრულებლად დაგვჭირდება შემდეგი დროები: A → 8, B → 12, C → 16, D → 20, ხოლო მათი შესრულების საშუალო დრო ვი იქნება $(8 + 12 + 16 + 20) / 4 = 14$ ერთეული.



ნახ. 3.3. დაგეგმვა. პირველი სრულდება მიმდევრობაში გამოჩენილი პირველივე პროცესი (ა). პირველი სრულდება შესასრულებლად ნაკლების დროის საჭიროების მქონე პროცესი (ბ)

თუკი იმავე პროცესების შესრულებას განვახორციელებთ ბ)-ზე წარმოდგენილი მიმდევ-

რობით, მაშინ პროცესების შესასრულებლად დაგჭირდება შემდეგი დროები: A--> 4, B--> 8, C--> 12, D --> 20, ხოლო მათი შესრულების საშუალო დრო კი იქნება $(4 + 8 + 12 + 20) / 4 = 11$ ერთეული.

შევნიშნოთ, შესაძლებელია იმის ჩვენება, რომ, თუ თავიდანვე ცნობილია სისტემაში არსებული ყველა პროცესი და მათი შესრულების დრო, მაშინ SJF ალგორითმი არის ოპტიმალური. განვიხილოთ მაგალითი. ვთქვათ, მოცემულია 5 პროცესი - A – E, შესრულების დროით 2, 4, 1, 1, 1 შესაბამისად და მიმდევრობაში გამოჩენის დროით 0, 0, 3, 3, 3. ცხადია, რომ ამ შემთხვევაში პროცესების შესრულება განხორციელდება წარმოდგენილი თანმიმდევრობით (A – E) და შესაბამისად შესრულების დროის საშუალო იქნება $(2 + 6 + 4 + 5 + 6) / 5 = 4,6$ დროითი ერთეული. თუკი ყველა პროცესი თავიდანვე გვექნებოდა, მაშინ მათი შესრულების თანმიმდევრობა იქნებოდა C D E A B, ხოლო შესრულების საშუალო დრო კი $- (1 + 2 + 3 + 5 + 9) / 5 = 4$ ერთეული.

3.6.3. პრიორიტეტული SJF

SJF ალგორითმისთვის პრიორიტეტის დამატებით მიიღება განსხვავებული ალგორითმი. ამ შემთხვევაშიც იგულისხმება, რომ პროცესების შესრულებისთვის საჭირო დრო წინასწარაა ცნობილი. სისტემაში ყოველი წარმოქმნილი პროცესის შესრულებისთვის საჭირო დრო ედრება მიმდინარე პროცესის დასრულდებისთვის საჭირო დროს. თუ ეს უკანასკნელი მეტია წარმოქმნილი პროცესის შესრულების დროზე, მაშინ მიმდინარე პროცესი წყვეტს შესრულებას და მის ადგილს იკავებს წარმოქმნილი პროცესი. თუკი წარმოქმნილი პროცესის შესრულების დრო მეტია მიმდინარე პორცესის დასრულებისთვის საჭირო დროზე, მაშინ წარმოქმნილი პროცესი ემატება მზადყოფნის ცხრილში მყოფი პროცესების მიმდევრობას და იქ იკავებს შესაბამის ადგილს (პრიორიტეტის მიხედვით). ასეთნაირად რეალიზებული SJF ალგორითმი იძლევა სისტემაში წარმოქმნილი მოკლე დროის საჭიროების მქონე პროცესების დროული შესრულების შესაძლებლობას.

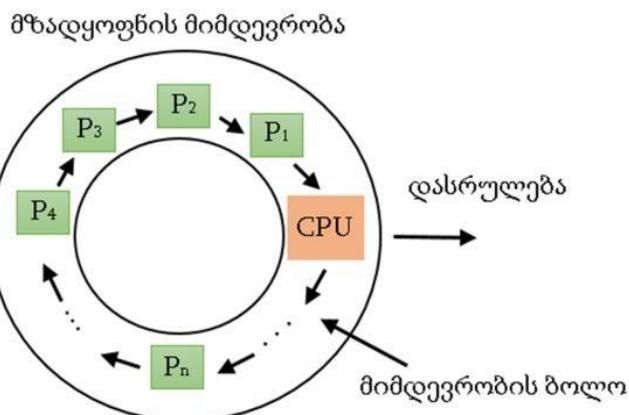
3.6.4. ციკლური დაგეგმვა (RR - Round Robin)

ამოცანების დაგეგმვის ერთერთ ძველ, მარტივ, სამართლიან და ხშირად გამოყენებად ალგორითმს წარმოადგენს ციკლური დაგეგმვის ალგორითმი. ამ შემთხვევაშიც სისტემაში ყოველი წარმოქმნილ პროცესი „მზადყოფნა“ მდგომარეობაში მყოფი პროცესების მიმდევრობას ემატება ბოლოდან. მიმდევრობის ელემენტებისთვის პროცესორის გამოყოფა ხორციელდება გარკვეული დროითი შუალედებით (დროითი კვანტით). თუ დროითი კვანტი პროცესის დასასრულებლად არ იყო საკმარისი, მაშინ მას ჩამოერთმევა პროცესორი. პროცესი გადადის მიმდევრობის ბოლოში (თავიდან იკავებს რიგს) და პროცესორი შესასრულებლად გადაეცემა მიმდევრობის შემდეგ წევრს. თუ დროითი კვანტი საკმარისი აღმოჩნდა პროცესის დასასრულებლად, მაშინ პროცესორი გადაეცემა მიმდევრობის რიგით შემდეგ წევრს. პროცესორის გამოყოფა ხორციელდება ციკლურად მანამ მიმდევრობა შეიცავს ერთ მაინც წევრს.

ციკლური დაგეგმვისთვის მნიშვნელოვან მომენტს წარმოადგენს დროითი კვანტის განსაზღვრა. ერთი პროცესიდან მეორე პროცესზე გადართვა ადმინისტრირების ამოცანების შესასრულებლად (რეგისტრებისა და მეხსიერების რუკის შენახვა, სხვადასხვა ცხრილების განახლება) მოითხოვს გარკვეულ დროით შუალედს. თუ დავუშვებთ, რომ პროცესის გადართვა ადმინისტრირების ამოცანის განსახორციელებლად საჭიროებს 1 ნწმ. და დროითი კვანტი 4 ნწმ.- ია, მაშინ გამოდის, პროცესორის მიერ 4 ნწმ. სასარგებლო სამუშაოში გატარებულ დროს ყოველთვის მოყვება 1 ნწმ. (20%) არამომგებიან სამუშაოში დახარჯული დრო.

პროცესორის ეფექტურად გამოყენების თვალსაზრისით შესაძლებელია კვანტის გაზრდა 100 წწ.-მდე. ამ შემთხვევაში იკარგება მხოლოდ 1%, მაგრამ ასეთი დიდი დროითი კვანტის გამოყენებას შეიძლება მოჰყვეს უარყოფითი შედეგი. განვიხილოთ სერვერული სისტემის მაგალითი, რომელშიც დროის მცირე შუალედში შემოსულია შესრულების სხავდასხვა დროითი შუალედის მქონე 50 მოთხოვნა. ყველა მოთხოვნა შემოსვლისთანავე განთავსდება მიმდევრობაში და თუ პროცესორი არაა დაკავებული ის გამოეყოფა მიმდევრობის პირველივე მოთხოვნას (პროცესს). მეორე პროცესი პროცესორს მიიღებს მხოლოდ 100 წწ.-ის გასვლის შემდეგ და ა.შ. თუკი დავუშვებთ, რომ ყველა პროცესი საკუთარ

კვანტით დროს გამოიყენებს სრულად, მაშინ აღმოჩნდება, რომ მიმდევრობის ბოლო პროცესს მოუწევს 5 მიკრო წმ-ის დალოდება პროცესორის მისაღებად. რაც გამოიწვევს ნებისმიერი მომხმარებლის უკმაყოფი- ლებას. შედეგი განსაკუთრებით არასასურველი იქნება იმ შემთხვევაში, თუკი პროცესი საჭიროებს რამდენიმე წწ.-ს. თუკი ამ შეთხვევაში დროითი კვანტის მნიშვნელობა იქნებოდა პატარა სიდიდე, მაშინ თითოეული მოთხოვნის მომსახურეობა რათქმაუნდა გაუმჯობესდებოდა.



ნახ. 3.4. დაგეგმვა RR პრინციპის

მეორე თავისებურება მდგომარეობს იმაში, რომ თუ კვანტი მეტია პროცესორის ჩართულობის საშუალოზე, მაშინ პროცესები არ განახორციელებენ ხშირ გადართვას. ნაცვლად ამისა, პროცესების უმეტესი დაიკავებს პროცესორს კვანტის ამოწურვამდე (რომელიც გამოიწვევს პროცესის იძულებით გადართვას). იძულებითი წყვეტის არარსებობა ზრდის სისტემის წარმადობას, ვინაიდან პროცესების გადართვა განხორციელდება ლოგიკური აუცილებლობით, ანუ როდესაც მოხდა პროცესის ბლოკირება და ის ვერ აგრძელებს შესრულებას.

მაშასადამე, დროითი კვანტის საკმარისად მცირე სიდიდით განსაზღვრისას, მოხდება პროცესების ხშირი გადართვა და პროცესორის გამოყენების ეფექტურობა მცირდება, ხოლო დიდი მნიშვნელობის მიცემამ კი შეიძლება მიგვიყვანოს იქამდე, რომ მოკლე მოთხოვნებს დასჭირდეთ დიდი ხნით დალოდება.

3.6.5. პრიორიტეტული დაგეგმვა

დაგეგმვისას გარე ფაქტორების გათვალისწინებას მივყავართ პრიორიტეტულ ალგორითმებამდე. პრიორიტეტული დაგეგმვის ალგორითმის არსი მარტივია: სისტემაში გამოჩენილ ყოველ პროცესს ენიჭება გარკვეული რიცხვითი მნიშვნელობა. „მზადყოფნა“ მდგომარეობაში მყოფი პროცესებიდან შესასრულებლად აირჩევა მაღალი პრიორიტეტის მქონე პროცესი.

მიუხედავად იმისა, მიმდინარე მომენტში რამდენი მომხმარებელი სარგებლობს გამოთვლითი

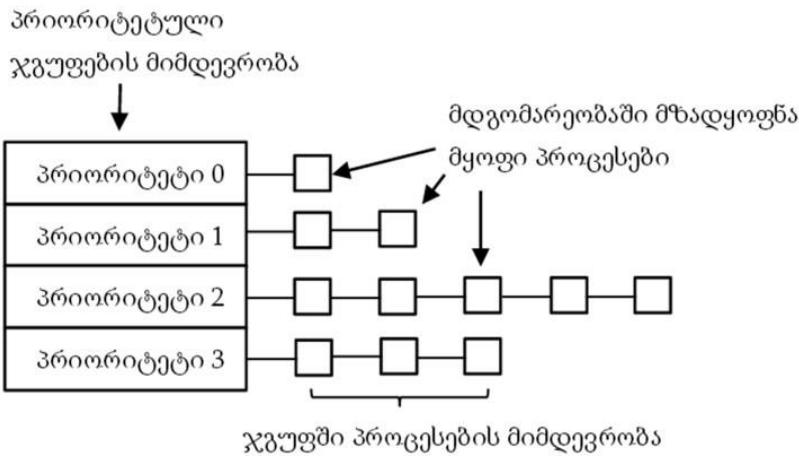
სისტემის მომსახურეობით გამოთვლით სისტემაში შესაძლებელია არსებობდეს რამდენიმე განსხვავებული პრიორიტეტის მქონე პროცესი. მაგალითად, ელექტრონული ფოსტის შემოწმების პროცესს უნდა გააჩნდეს ნაკლები პრიორიტეტი ვიდრე მომხმარებლის მიერ გააქტიურებულ ვიდეო ფაილის შესრულების პროცესს.

იმის გამო, რომ ოპერაციულ სისტემაში მუდმივად არ მოხდეს მაღალპრიორიტეტული პროცესების შესრულება და ამან არ გამოიწვიოს სხვა პროცესების შესრულების დიდი ხნით გადადება ოპერაციულ სისტემაში შემოღებულია მექანიზმი, რომელიც უზრუნველყოფს დაბალ-პრიორიტეტული პროცესების შესრულებასაც. ამ მექანიზმის არსი მდგომარეობს შემდეგში: რამდენჯერაც განხორციელდება მაღალპრიორიტეტული პროცესისთვის პროცესორის გამოყოფა იმდენით შენცირდეს (გაიზარდოს) პროცესის პრიორიტეტი. დამგეგმავი პრიორიტეტის მნიშვნელობის შეცვლას ახორციელებს ტაიმერიდან მიღებული წყვეტის სიგნალის საფუძველზე.

3.6.7. მრავალდონიანი მიმდევრობა (Multilevel Queue)

ზოგიერთ ოპერაციულ სისტემაში გამოიყენება პროცესების დაჯუფება პრიორიტეტის მნიშვნელობის მიხედვით. ამ შემთხვევაში სისტემაში იქმნება მზადყოფნის მდგომარეობაში მყოფი პროცესების იმდენი მიმდევრობა რამდენი პრიორიტეტის განსხვავებული მნიშვნელობა არსებობს პლიუს ერთი, ანუ ყოველ ჯგუფი შეიცავს მზადყოფნის მდგომარეობაში მყოფი პროცესების საკუთარ მიმდევრობას და პრიორიტეტებად დაყოფილი პროცესების ჯგუფები ქმნიან კიდევ ერთ ჯგუფს. პროცესების ასეთი დაყოფის მიმართ შესაძლებელია გამოყენებულ იქნას სხვადასხვა მიდგომა. მაგალითად, დიდი ჯგუფის მიმართ, რომელიც აერთიანებს სხვადასხვა პიორიტეტის მნიშვნელობის ჯგუფებს, შესაძლებელია გამოყენებული იქნას პრიორიტეტული დაგეგმვა, ხოლო პრიორიტეტის ქვეჯგუფებში კი - ზემოთ განხილული ალგორითმებიდან ნებისმიერი. პროცესების შესრულება ხორციელდება შემდეგნაირად: პირველი შესრულდება მაღალპრიორიტეტული ჯგუფის პროცესები. ამის შემდეგ პროცესორი შეიძლება გამოეყოს პრიორიტეტით შემდეგ ჯგუფს მხოლოდ იმ შემთხვევაში, თუ მასში ყველა პროცესი დასრულებულია ან რომელიმე მათგანი იმყოფება ლოდინის მდგომარეობაში. პრიორიტეტული გამევებით დაგეგმვის ალგორიმის გამოყენების შემთხვევაში თუ დაფიქსირდება მაღალპრიორიტეტული პროცესის გამოსვლა ლოდინის მდგომარეობიდან, მაშინ მიმდინარე პროცესს ჩამოერთმევა პროცესორი და გადაეცემა ლოდინის მდგომარეობიდან გამოსულ პროცესს. წინააღმდეგ შემთხვევაში მიმდინარე პროცესი შეძლებს პროცესორთან დარჩენას დროითი კვანტის ამოწურვამდე და მხოლოდ ამის შემდეგ გადაეცემა ლოდინის მდგომარეობიდან გამოსულ პროცესს შესრულების შესაძლებლობა. შემდეგ დაიწყება პრიორიტეტის მნიშვნელობით შემდეგი ჯგუფის პროცესების მიმდევრობის შესრულება და ასე გრძელდება სანამ არ ამოიწურება სისტემაში არსებული ყველა პროცესი ჯგუფებში. აღწერილ ალგორითმს მრავალდონიანი მიმდევრობა ეწოდება (ნახ. 3.5).

მრავალდონიანი მიმდევრობის გამოყენებას გააჩნია თავისი ნაკლოვანება, რაც გამოიხატება შემდეგში: სისტემაში მუდმივად წარმოიქმნება ახალი პროცესები, რომლებიც შესასრულებლად საჭიროებენ სხვადასხვა რესურსებსა და შესრულების განსხვავებულ დროს. შეიძლება აღმოჩნდეს, რომ მაღალპრიორიტეტული პროცესები, რომლებიც მოითხოვენ შესრულების ხანგრძლივ პერიოდს პროცესორს იკავებენ დიდი ხნით და ამ ხნის განმავლობაში არაერთი მაღალპრიორიტეტული პროცესი შეიძლება წარმოიქმნას სისტემაში. შესაბამისად, წარმოიშობა საფრთხე, რომ პროცესორი შეიძლება საერთოდ არ გამოეყოს დაბალპრიორიტეტული ჯგუფის პროცესებს ან ამ ჯგუფის რომელიმე პროცესის, რომელიც შესასრულებლად საჭიროებს ხანგრძლივ პერიოდს, შესრულება გაგრძელდეს უსასრულოდ.



ნახ. 3.5. დაგეგმვის მრავალდონიანი მიმდევრობის ალგორითმი

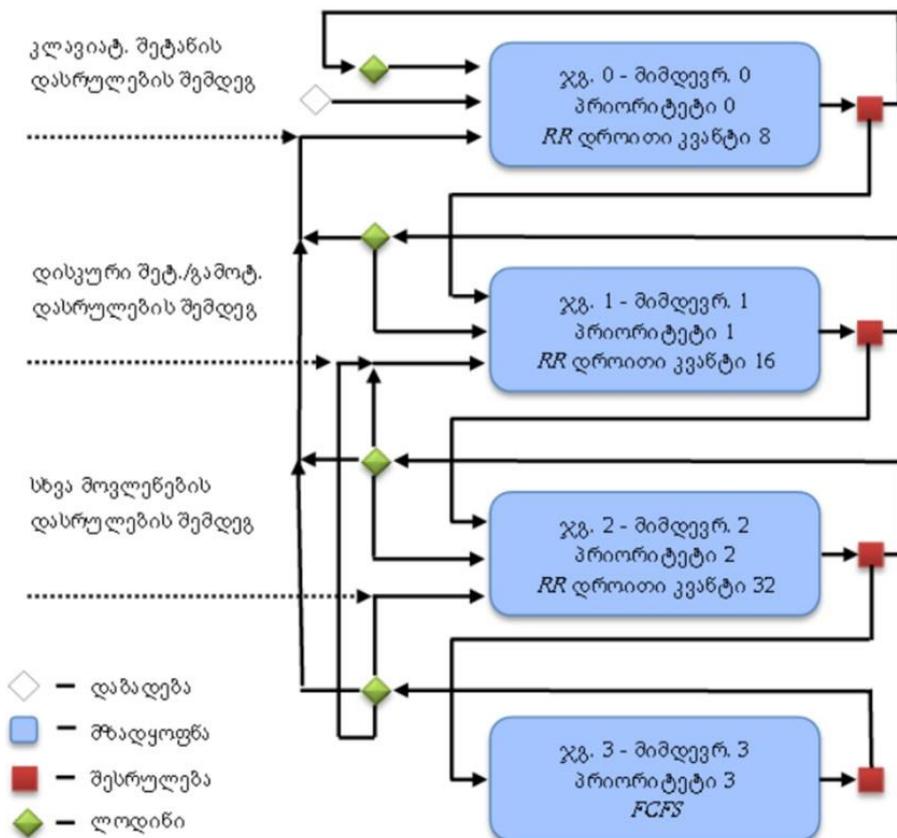
4 პრიორიტეტული ჯგუფით (დაბალი მნიშვნელობა

აღნიშნავს მაღალ პრიორიტეტს)

ასეთი სიტუაციის თავიდან აცილების მიზნით საჭიროა დამატებითი მექანიზმის გათვალისწინება, რომლის მიხედვითაც მოხდება პრიორიტეტის კონკრეტული მნიშვნელობის მქონე ჯგუფის პროცესის, რომელსაც ერთხელ მაინც გამოეყო პროცესორი და არ დაუმთავრებია შესრულება, გადაყვანა ამა თუ იმ ჯგუფში. მრავალდონიან მიმდევრობაში მსგავსი მექანიზმის გათვალისწინებით მიღებულ ალგორითმს ეწოდება მრავალდონიანი მიმდევრობები უკუკავშირით (Multilevel Feedback Queue).

დაგეგმვის „მრავალდონიანი მიმდევრობები უკუკავშირით“ ალგორითმის მუშაობის საილუსტრაციოდ განვიხილოთ მაგალითი, რომელშიც გვაქვს პრიორიტეტის 4 ჯგუფი (ნახ. 3.6). დავუშვათ ჯგუფებისთვის პრიორიტეტის მნიშვნელობა განსაზღვრულია მასში შემავალი პროცესების შესასრულებლად საჭირო დროით. ჯგუფების მიმდევრობისთვის ხორციელდება პრიორიტეტული დაგეგმვა, რაც ნიშნავს, რომ პირველი შესრულდება ჯგუფი 0-ის ყველა პროცესი, შემდეგი შესრულდება ჯგუფი 1-ის ყველა პროცესი და ასე გაგრძელდება სანამ რომელიმე ჯგუფი შეიცავს ერთ მაინც პროცესს. როგორც ნახ. 3.6-დან ჩანს 0-2 ჯგუფებში შიგნით არსებული მიმდევრობებისთვის დაგეგმვა ხორციელდება RR ალგორითმით, ხოლო ჯგუფ 3-ში არსებული მიმდევრობისთვის კი - FCFS ალგორითმით.

სისტემაში წარმოქმნილი ყოველი ახალი პროცესის განთავსება ამა თუ იმ ჯგუფში ხორციელდება იმის მიხედვით, თუ რამდენ დროს საჭიროებს ის შესასრულებლად, ანუ თუ ის შესასრულებლად საჭიროებს 1 – 8 ერთეულს მოხვდება ჯგუფ 0-ში; თუ საჭიროებს 9 – 16 ერთეულს მოხვდება ჯგუფში 1; თუ საჭიროებს 17 – 32 ერთეულს მოხვდება ჯგუფ 2-ში. წინააღმდეგ შემთხვევაში მოხვდება ჯგუფ 3-ში. როგორც უკვე აღვნიშნეთ, ჯგუფი 1-ის პროცესების შესრულება დაიწყება მას შემდეგ რაც ჯგუფ 0-ში არ გვექნება არცერთი პროცესი. ჯგუფი 1-ის რომელიმე პროცესის შესრულებისას, თუ სისტემაში წარმოქმნა რაიმე პროცესი, რომელიც შეიძლება მიეკუთვნებოდეს ჯგუფ 0-ს, მაშინ მიმდინარე პროცესს ჩამოერთმევა პროცესორი და ის გადაეცემა წარმოქმნილ პროცესს. თუკი მოცემული მომენტისთვის პროცესორის ჩამორთმევის შემდეგ მიმდინარე პროცესი შესასრულებლად საჭიროებს 1 – 8 ერთეულს, მაშინ მოხდება მისი გადაყვანა ჯგუფ 0-ში. ანალოგიურად მოხდება სხვა ჯგუფების შემთხვევაშიც. მაგალითად, თუ მოცემულ მომენტში სრულდებოდა ჯგუფი 3-ის რაიმე პროცესი და მას სისტემაში წარმოქმნილი მაღალპრიორიტეტული პროცესის გამო ჩამოერთვა პროცესორი, მაშინ მიმდინარე პროცესის განთავსება რომელიმე ჯგუფში მოხდება იმის მიხედვით, თუ რამდენი დრო დარჩა მას შესასრულებლად: 1 – 8 ერთეულის შემთხვევაში ჯგუფ 0-ში, 9 – 16 ერთეულის შემთხვევაში ჯგუფ 1-ში და ა.შ.



**ნახ. 3.6. პროცესების მიგრაციის სქემა დაგეგმვის მრავალდონიანი
მიმდევრობებში უკუკავშირით. სქემაში არაა ნაჩვენები
პროცესების გაძევება და დასრულება**

მრავალდონიანი მიმდევრობები უკუკავშირით წარმოადგენს პროცესების დაგეგმვისადმი უფრო ზოგად მიდგომას ჩვენს მიერ განხილული მიდგომებიდან. ის რეალიზაციაში საკმაოდ რთულია და ამავდროულად მოქნილიც.

ცხადია, რომ ჩვენს მიერ განხილული დაგეგმვის ალგორითმების გარდა არსებობს სხვა მრავალი ალგორითმიც. მათი კონკრეტული რეალიზაციის სრულად აღწერისთვის საჭიროა მივუთთოთ:

- „მზადყოფნა“ მდგომარეობაში მყოფი პროცესების მიმდევრობების რაოდენობა;
- პრიორიტეტების ჯგუფების მიმდევრობისთვის გამოყენებული დაგეგმვის ალგორითმი;
- პრიორიტეტების ჯგუფების შიგნით გამოყენებული დაგეგმვის ალგორითმი;
- წესი, რომლითაც ხორციელდება სისტემაში წარმოქმნილი პროცესის რომელიმე ჯგუფში განთავსება;
- პროცესების ერთი მიმდევრობიდან მეორეში გადაყვანის წესი.

ლექცია 4. პროცესების ურთიერთქმედება

მიუხედავად იმისა, პროცესები ამუშავებულია თუ არა ერთ ან რამდენიმე გამოთვლით მანქანაზე, მათ ხშირად უწევთ ერთიმეორესთან ურთიერთქმედება (მონაცემების გაცვლა, საერთო რესურსების გამოყენება, ერთიმეორის მიერ მიღებული შედეგების გათვალისწინება საკუთარი სამუშაოს შესრულებისას და ა.შ.).

4.1. ურთიერთქმედი პროცესები

ბუნებრივია ისმის კითხვა: რატომ უნდა ეწეოდნენ პროცესები ერთობლივ საქმიანობას? პროცესების ერთობლივი მუშაობა შესაძლებელია გამოწვეული იყოს რამდენიმე მიზეზით:

- **მუშაობის სიჩქარის ამაღლება.** თუ ურთიერთქმედი პროცესებიდან ერთი რომელიმე ელოდება გარკვეული მოვლენის დადგომას (მაგალითად: შეტანა/გამოტანის ოპერაციის დასრულება ან სხვა პროცესის მიერ შესრულებული სამუშაოს შედეგს), მაშინ სხვა პროცესი შესაძლებელია ეწეოდეს სასარგებლო საქმიანობას, რომელიც მიმართულია საერთო საქმის გადაწყვეტისკენ. მაგალითად, როგორც ვიცით, პროცესების კონცეფციის მიხედვით პროგრამა შეიძლება წარმოვიდგინოთ მიმდევრობით შესრულებადი პროცესების ნაკრების სახით. თუ ამ ნაკრებიდან შესაძლებელია პროცესების გარკვეული რაოდენობის პარალელური შესრულება, მაშინ მრავალპროცესორულ სისტემებში ისინი შეიძლება შესრულდეს სხვადასხვა პროცესორებზე.
- **რესურსების ერთობლივი გამოყენება.** გამოთვლითი მანქანა აღჭურვილია რესურსების შეზღუდული რაოდენობით. ყოველი პროცესი ცდილობს დაიკაოს მის შესასრულებლად საჭირო რესურსი, თუ ეს უკანასკნელი თავისუფალია და არ ათავისუფლებს მას სანამ არ დასრულდება. პროცესის მიერ დაკავებული რესურსის გამოყენება სხვა პროცესს არ შეუძლია. თუ შესაძლებელი იქნებოდა რესურსების ერთობლივი გამოყენება, მაშინ შესაძლებელი იქნებოდა კონკრეტული რესურსის ლოდინის მდგომარეობაში მყოფი პროცესების ნაწილობრივ შესრულება ან დასრულება, რაც გაზრდიდა რესურსის გამოყენების ეფექტურობას. მაგალითად, პროცესებმა იმუშაონ მონაცემთა ერთიდაიმავე დინამიურ ბაზასთან ან საერთო ფაილებთან, გამოიყენონ საერთო მეხსიერების სივრცე, და ამ რესურსებზე შეთანხმებულად განახორციელონ ცვლილებები;
- **მოდულური კონსტრუქცია.** სისტემა შესაძლებელია შედგებოდეს ერთიმეორესთან ურთიერთქმედი დამოუკიდებელი მოდულებისგან. ასეთი სისტემის ტიპიურ მაგალითს წარმოადგენს ოპერაციული სისტემა მიკრობირთვის ბაზაზე, რომელშიც სისტემის შემადგენელი კომპონენტები (მოდულები) წარმოადგენენ ერთიმეორესთან მიკრობირთვის მეშვეობით დაკავშირებულ ურთიერთქმედ პროცესებს;
- და ბოლოს, პროცესების ერთობლივად მუშაობის აუცილებლობა შესაძლებელია გამოწვეული იყოს მომხმარებლის მოხერხებული მუშაობის ორგანიზაციის მიზნით. მაგალითად, როდესაც ის ერთდროულად ცდილობს განახორციელოს პროგრამის რედაქტირება და კომპილირება. ამ შემთხვევაში საჭიროა შეესაბამისი რედაქტორის და კომპილატორის პროცესები შეთანხმებულად ურთიერთქმედებდნენ ერთიმეორესთან.

პროცესებს ერთიმეორესთან ურთიერთქმედება არ შეუძლიათ მონაცემების გაცვლის გარეშე. მიღებულმა მონაცემებმა შესაძლებელია შეცვალოს პროცესის ყოფაქცევა. პროცესებს, თუ ისინი მონაცემების გაცვლის გზით ზემოქმედებენ ერთიმეორის ყოფაქცევაზე, ურთიერთქმედი პროცესები ეწოდებათ. წინააღმდეგ შემთხვევაში, მათ დამოუკიდებელი პროცესები ეწოდებათ.

4.2. მონაცემების გაცვლის საშუალებები

ურთიერთქმედ პროცესებს შორის გადასაცემი მონაცემების მოცულობის და პროცესის ყოფაქცევაზე შესაძლო ზემოქმედების მიხედვით მონაცემების გაცვლის საშუალებები იყოფა სამნაწილად:

- სიგნალური.** გადასაცემი ინფორმაციის მოცულობა მინიმალურია - ერთი ბაიტი. სიგნალური საშუალებით ხდება პროცესორის ინფორმირება გარკვეული მოვლენის დადგომის და მასზე შესაბამისი რეაგირების საჭიროების თაობაზე. ამ შემთხვევაში მონაცემების მიმღები პროცესის ყოფაქცევაზე ზემოქმედება მინიმალურია. ყველაფერი დამოკიდებულია იმაზე, გააჩნია თუ არა მას მიღებული სიგნალის ტიპზე ინფორმაცია და როგორ დაამუშაოს ის. სიგნალზე არასწორმა რეაგირებამ ან მისმა უგულვებელყოფამ შესაძლებელია მიგვიყვანოს გაუთვალისწინებელ შემთხვევამდე;
- არხული.** პროცესებს შორის მონაცემების გაცვლა ხორციელდება ოპერაციული სისტემის მიერ ამ მიზნით სპეციალურად გამოყოფილი კავშირის არხებით. გადასაცემი მონაცემების მოცულობა დამოკიდებულია კავშირის არხის გამტარუნარიანობაზე. გადასაცემი მონაცემების მოცულობის გაზრდით იზრდება სხვა პროცესების ყოფაქცევაზე ზემოქმედების შესაძლებლობა;
- განაწილებადი მეხსიერება.** ორ ან რამდენიმე პროცესს შეუძლია შეთანხმებულად გამოიყენოს მისამართების სივრცის გარკვეული ნაწილი. გადასაცემი მონაცემების მოცულობა დამოკიდებულია მეხსიერების გამოყოფილ ნაწილზე. მონაცემების მაქსიმალური მოცულობით გადაცემა საჭიროებს გარკვეულ სიფრთხილეს, ვინაიდან მან შეიძლება ზეგავლენა იქონიოს სხვა პროცესების ყოფაქცევაზე. განაწილებადი მეხსიერება წარმოადგენს ერთ გამოთვლით სისტემაში პროცესების ურთიერთქმედების საკმაოდ ჩქარ და სუაფრთხო მეთოდს.

ურთიერთქმედ პროცესებს შორის მონაცემების მიღება/გადაცემის პროგრამირება, განაწილებადი მეხსიერების გამოყენებით, ხორციელდება პროგრამირების ენების საშუალებებით, ხოლო სიგნალური და არხული საშუალებები კი საჭიროებენ სპეციალურ სისტემურ გამოძახებებს.

4.3. კომუნიკაციის საშუალებების ლოგიკური ორგანიზაცია

კომუნიკაციის საშუალებების განხილვისას ჩვენი ინტერესის სფეროს წარმოადგენს არა მათი ფიზიკური რეალიზაცია, არამედ ლოგიკური, რომელიც განსაზღვრავს მათი შესრულების მექანიზმებს. მოკლედ აღვწეროთ ისინი.

4.3.1. კავშირის დამყარება

ბუნებრივია ისმის კითხვა: ურთიერთქმედ პროცესებს შორის მონაცემების გასაცვლელად კავშირის საშუალება უშუალოდ პროცესის წარმოქმნასთან ერთად უნდა იყოს გამოყენებული, თუ მონაცემების გაცვლამდე საჭიროა გარკვეული ინიციალიზაცია? მაგალითად, სხვადასხვა პროცესები საერთო მეხსიერების გამოყენების მიზნით ოპერაციულ სისტემას მიმართავენ სპეციალური ფორმით, რომელიც შემდგომ გამოყოფს მას. მაგრამ პროცესებს შორის მონაცემების გადაცემა არ საჭიროებს ინიციალიზირებას.

კავშირის დამყარებასთან მჭიდრო კავშირშია კავშირის საშუალებების გამოყენებისას დამისამართების მეთოდები. ინფორმაციის გადაცემა/მიღებისას რიგ შემთხვევაში საჭიროა ცხადად იყოს მითითებული ინფორმაციის გამგზავნი და მიმღები.

განასხვავებენ დამისამართების ორ მეთოდს: **პირდაპირი და არაპირდაპირი.** პირდაპირი

დამისამართებისას ურთიერთქმედი პროცესები პირდაპირ ურთიერთქმედებენ ერთიმეორესთან. გაცვლის ყოველი ოპერაციისას ცხადად ეთითება იმ პროცესის სახელი ან ნომერი, რომელსაც ეგზავნება მონაცემები, ან რომელმაც უნდა გამოაგზავნოს ისინი. თუ გამგზავნი და მიმღები პროცესი ცხადად უთითებს ურთიერთქმედების მხარეს, მაშინ დამისამართების ასეთ სქემას სიმეტრიული პირდაპირი დამისამართება ეწოდება. წინააღმდეგ შემთხვევაში დამისამართების სქემას ეწოდება ასიმეტრიული პირდაპირი დამისამართება. პროცესების სიმეტრიული პირდაპირი ურთიერთქმედებისას სხვა პროცესს არ შეუძლია ჩაერიოს მათ საქმიანობაში, დაიჭიროს გაგზავნილი მონაცემები ან შეცვალოს ისინი.

არაპირდაპირი დამისამართებისას გამგზავნი პროცესის მიერ გაგზავნილი მონაცემები თავსდება გარკვეულ შუალედურ ობიექტში, საიდანაც მისი აღება შეუძლია მსგავსი მონაცემების საჭიროების მქონე რომელიმე პროცესს. ამასთან, არცერთი მხარე, რომელმაც განათავსა მონაცემები შუალედურ ობიექტში და რომელმაც აიღო ისინი იქიდან, არ საჭიროებს მეორე მხარის იდენტიფიცირებას.

პირდაპირი დამისამართების გამოყენებისას პროცესებს შორის კავშირი კლასიკურ ოპერაციულ სისტემებში მყარდება ავტომატურად, დამატებითი მაინიცირებელი მოქმედებების გარეშე. ამ შემთხვევაში კავშირის საშუალებების გამოსაყენებლად საჭიროა გაცვლაში მონაწილე პროცესების იდენტიფიცირების პროცედურის ცოდნა.

არაპირდაპირი დამისამართების შემთხვევაში შესაძლებელია არც იყოს საჭირო კავშირის საშუალებების ინიციალიზირება. ამ შემთხვევაში პროცესს უნდა გააჩნდეს ინფორმაცია მონაცემების შესანახი შუალედური ობიექტის იდენტიფიკატორზე.

4.3.2. კავშირის საშუალებების ინფორმაციული ვალენტურობა

შემდეგი მნიშვნელოვანი საკითხი არის კავშირის ინფორმაციული ვალენტურობის საკითხი. რამდენი პროცესი შეიძლება ერთდროულად იყოს ასოცირებული კავშირის კონკრეტულ საშუალებასთან? ორ პროცესს შორის რამდენი ასეთი კავშირი შეიძლება იყოს გამოყენებული?

ცხადია, რომ პირდაპირი დამისამართებისას ორ პროცესს შორის მონაცემების გაცვლის მიზნით კავშირის მხოლოდ ერთი ფიქსირებული საშუალება შეიძლება იყოს გამოყებული, და მხოლოდ ეს ორი პროცესი შეიძლება ასოცირდებოდეს მასთან. არაპირდაპირი დამისამართებისას შეიძლება არსებობდეს ორზე მეტი პროცესი, რომლებიც იყენებენ მონაცემების შესანახ ერთსადაიმავე შუალედურ ობიექტს, და ერთზე მეტი შუალედური ობიექტი შესაძლებელია გამოყენებული იყოს ორი პროცესის მიერ.

საკითხების ამ ჯგუფს უნდა მივაკუთვნოთ კავშირის მიმართულობის საკითხი: კავშირი ერთმიმართულებიანია თუ ორმიმართულებიანი? კავშირი ერთმიმართულებიანია, თუ მასთან ასოცირებულ პროცესს კავშირის საშუალება შეუძლია გამოიყენოს მხოლოდ ერთი მიმართულებით: ინფორმაციის მისაღებად, ან მის გადასაცემად. ორმიმართულებიან კავშირში ყოველ პროცესს, რომელიც მონაწილეობს ურთიერთქმედებაში, შეუძლია გამოიყენოს კავშირი მონაცემების მისაღებად და მის გადასაცემად. კომუნიკაციურ სისტემებში ერთმიმართულებიან კავშირს ეწოდება სიმპლექსური, ორ მიმართულებიან კავშირს კი მონაცემების სხვადასხვა მიმართულებით მიმდევრობით გადაცემის შესაძლებლობით - ნახევრად დუპლექსური, ხოლო მონაცემების ორივე მიმართულებით ერთდროული გადაცემის შესაძლებლობით კი დუპლექსური.

4.3.3. მონაცემების გადაცემის თავისებურებაზი

როგორც უკვე აღვნიშნეთ, პროცესებს შორის მონაცემების გადაცემა კავშირის არხის საშუალებით წარმოადგენს საკმაოდ უსაფრთხო მეთოდს განაწილებადი მეხსიერების გამოყენებასთან მიმართებაში და უფრო ინფორმაციულია კომუნიკაციის სიგნალურ საშუალებებთან შედარებით. გარდა ამისა, განაწილებადი მეხსიერება არ შეიძლება გამოყენებული იყოს იმ პროცესების კავშირისთვის, რომლებიც ფუნქციონირებენ სხვადასხვა გამოთვლით სისტემებში. სწორედ ამიტომ პროცესების კომუნიკაციის საშუალებებიდან კავშირის არხმა ფართო გამოყენება ჰქონდა. განვიხილოთ ზოგიერთი საკითხი, რომელიც დაკავშირებულია კავშირის არხული საშუალებების ლოგიკურ ორგანიზაციასთან.

4.3.4. ბუფერი

როგორც ვიცით ურთიერთქმედი პროცესები შეიძლება განთავსებული იყვნენ როგორც ლოკალურ მანქანაზე ისე მოშორებულ მანქანებზე. პროცესებს შორის მონაცემების გაცვლისას ის შეიძლება პირდაპირ იგზავნებოდეს პროცესებს შორის ან თავსდებოდეს რაიმე შუალედურ ობიექტში, ე.წ. **სისტემურ ბუფერში**. ბუფერი არის მეხსიერების უჯრედების (ბაიტების) გარკვეული ერთობლიობა, რომელიც გამოყოფილია ოპერაციული სისტემის მიერ მონაცემების დროებით შენახვის მიზნით. გამოყენებული არქიტექტურის მიხედვით სისტემა შეიძლება საერთოდ არ იყენებდეს ბუფერს.

პროცესებს შორის მონაცემების გაცვლის საჭიროებისას ბუნებრივად ისმის კითხვა: შეუძლია თუ არა კავშირის არხს ერთი პროცესის მიერ გადასაცემი მონაცემების შენახვა სანამ მეორე პროცესი არ მიიღებს მას? ხომ არ ხდება მონაცემების განთავსება შუალედურ ობიექტში? როგორი უნდა იყოს გადასაცემი მონაცემების მოცულობა? სხვა სიტყვებით, რომ ვთქვათ, გააჩნია თუ არა კავშირის არხს ბუფერი და როგორია მისი მოცულობა. აქ შეიძლება გამოიყოს 3 შესაძლებლობა:

- **ბუფერი ნულოვანი მოცულობით ან არ არსებობს.** კავშირის არხში შეუძლებელია მონაცემების შენახვა. ამ შემთხვევაში, მონაცემების გამგზავნი პროცესი, სანამ გააგრძელებდეს შესრულებას, უნდა დაელოდოს მიმღები პროცესის მიერ მონაცემების მიღების დადასტურებას.
- **შემოსაზღვრული მოცულობის ბუფერი.** ბუფერის მოცულობა ფიქსირებულია (ვთქვათ, N ერთეული), ანუ კავშირის არხს შეუძლია შეინახოს მონაცემების არაუმეტეს N ერთეულისა. თუ მონაცემების გადაცემის მომენტში ბუფერში მონაცემების შესანახად საკმარისი ადგილია, მაშინ გადამცემი პროცესი იწყებს მათ გადაცემას, ხოლო, თუ ბუფერი გადავსებულია ან მასში საკმარისი ადგილი არაა, მაშინ გამგზავნი პროცესი ელოდება ბუფერში მონაცემების შესანახად ადგილის გამოთავისუფლებას.
- **შემოუსაზღვრელი მოცულობის ბუფერი.** თეორიულად ეს შესაძლებელია, მაგრამ პრაქტიკულად არარეალიზებადი. თუ პროცესი საჭიროებს მონაცემების გამგზავნას მას არ აინტერესებს ბუფერში არის თუ არა მონაცემების განსათავსებლად საკმარისი ადგილი ან გადაცემული მონაცემები აიღო თუ არა მათი საჭიროების მქონე პროცესმა. ის ბუფერში მონაცემების გადასცემს მათი გამოჩენისთანავე (გადაცემის აუცილებლობის მიუხედავად).

არაპირდაპირი დამისამართების შემთხვევაში კავშირის არხული საშუალების გამოყენებისას ბუფერის მოცულობის ქვეშ იგულისხმება მონაცემების ის მოცულობა, რომელიც შეიძლება შესანახად იყოს განთავსებული შუალედურ ობიექტში.

4.3.5. შეტანა/გამოტანის ნაკადი და შეტყობინება

კავშირის არხის გამოყენებით მონაცემთა გადაცემის არსებობს ორი მოდელი: შეტანა/ გამოტანის ნაკადი და შეტყობინება. ნაკადების მოდელის მეშვეობით ინფორმაციის გადაცემისას ინფორმაციის გადაცემა/მიღების ოპერაციები საზოგადოდ არ ინტერესდებიან მონაცემების შემცველობით. პროცესს, რომელმაც კავშირის არხიდან წაიკითხა 100 ბაიტი, არ აინტერესებს იყო თუ არა წაკითხული მონაცემები გადაცემული ერთი პროცესის მიერ, თუ მათი გადაცემა მოხდა სხვადასხვა პროცესების მიერ 20 ბაიტიან პორციებად. სისტემის მხრიდან რამენაირი ინტერპრეტაციის გარეშე, მონაცემები წარმოადგენ ბიტების უბრალო ნაკადს.

კავშირის არხებით პროცესებს შორის ინფორმაციის გადაცემის საკმაოდ მარტივ მეთოდს წარმოადგენს მონაცემების გადაცემა *pipe* -ით (არხი, მილი). წარმოვიდგინოთ, რომ გამოთვლით სისტემაში გვაქვს გარკვეული უხილავი მილი, რომლის ერთი ბოლოდან პროცესებს შეუძლიათ ინფორმაციის „ჩაყრა“, ხოლო მეორედან კი - შემოსული ნაკადის მიღება. ასეთი მეთოდი ახდენს ნაკადების შეტანა/გამოტანის მოდელის რეალიზებას. ოპერაციულ სისტემაში მილის განთავსების შესახებ ინფორმაციას ფლობს მხოლოდ მისი წარმომქმნელი პროცესი. ეს ინფორმაცია მის მიერ შეიძლება გაზიარებული იყოს მხოლოდ მემკვიდრე პროცესებთან, ანუ მშობელი - შვილი დამოკიდებულებით დაკავშირებულ პროცესებთან (იმ შვილ პროცესებთან, რომლებიც *pipe* არხის წარმოქმნის შემდეგ წარმოიქმნენ). შესაბამისად, კავშირისთვის *pipe*-ის გამოყენება შეუძლიათ მხოლოდ იმ პროცესებს, რომელთაც ყავთ კავშირის მოცემული არხის წარმოქმნელი „წინაპარი“.

თუ *pipe*-ის შემქმნელ პროცესს ექნებოდა სისტემაში სხვა პროცესებისთვის *pipe*-ის გაზიარების შესაძლებლობა, ანუ შექმნილი *pipe* -ის ხილულად გადაქცევის შესაძლებლობა (მაგალითად, ოპერაციულ სისტემაში მისი გარკვეული სახელით დარეგისტრირების გზით) მივიღებდით ობიექტს, რომელსაც FIFO ეწოდება. FIFO-ს გამოყენებით შესაძლებელია სისტემაში ნებისმიერ პროცესებს შორის კავშირის ორგანიზება.

შეტყობინების მოდელში პროცესები გადასაცემ მონაცემებს ადებენ გარკვეულ სტრუქტურას. მონაცემების მთლიან ნაკადს ისინი ყოფენ ცალკეულ შეტყობინებებად, მონაცემებს შორის შეტყობინებების საზღვრის გავლებით გზით. შეტყობინებებს შორის საზღვრის მაგალითს წარმოადგენს ტექსტში წინადადებებს შორის წერტილი ან აბზაცის საზღვრები. გარდა ამისა, გადასაცემ მონაცემებს შესაძლოა დაემატოს მითითება იმაზე, თუ ვის მიერაა გაგზავნილი კონკრეტული შეტყობინება და ვისთვისაა ის განკუთვნილი. შეტყობინების მოცულობა შეიძლება იყოს ფიქსირებული ან ცვლადი. გამოთვლით სისტემებში შეტყობინების გადასაცემად გამოიყენება სხვადასხვა საშუალებები: შეტყობინებათა მიმდევრობები, *socket*-ები და ა.შ.

შეტანა/გამოტანის ნაკადების არხებსა და შეტყობინებათა არხებს ყოველთვის გააჩნიათ სასრული ზომის ბუფერი. მონაცემთა ნაკადების ბუფერის მოცულობაზე საუბრისას მას გამოვსახავთ ბაიტებში, ხოლო შეტყობინებების ბუფერის შემთხვევაში კი მის მოცულობას გამოვსახავთ შეტყობინებების რაოდენობით.

4.3.6. კავშირის არხების საიმედოობა

კავშირის ყველა საშუალების განხილვისას ერთ-ერთ მნიშვნელოვანია მათი საიმედოობის საკითხი.

კომუნიკაციის საშუალებას ეწოდება საიმედო, თუ მონაცემთა გაცვლისას შესრულებულია შემდეგი ოთხი პირობა:

1. მონაცემები არ იკარგება.

2. მონაცემები არ ზიანდება.

3. მონაცემებში არ ჩნდება ზედმეტი ინფორმაცია.

4. მონაცემების გადაცემისას არ ირღვევა გაცვლის თანმიმდევრობა.

ცხადია, რომ განაწილებადი მეხსიერებით მონაცემების გადაცემა წარმოადგენს კავშირის საიმედო საშუალებას. ის, რაც შენაბულია განაწილებად მეხსიერებაში, სხვა პროცესების მიერ იკითხება პირვანდელი სახით (თუ რაიმე მიზეზით არ წაიშალა მეხსიერებიდან, მაგალითად, კომპიუტერის კვების წყაროს გამორთვა). კომუნიკაციის სხვა საშუალებებისთვის, როგორც ამას ვხედავთ ზემოთ მოყვანილი მაგალითებიდან, ეს ყოველთვის არა სამართლიანი.

როგორ ხორციელდება გამოთვლით სისტემაში კომუნიკაციის საშუალების არასაიმედოობასთან გამკვლავება? კავშირის არხით შეტყობინებებით მონაცემების გაცვლის მაგალითზე განვიხილოთ ის საშუალებები, რომლებიც გამოიყენება არასაიმედო გადაცემასთან გამკვლავების მიზნით. ინფორმაციის დაზიანების აღმოსაჩენად ყოველი გადასაცემი შეტყობინება აღჭურვილია გაგზავნილი ინფორმაციისთვის გარკვეული ალგორითმით (მაგალითად, CRC - Cyclic Redundancy Check) გამოთვლილი საკონტროლო ჯამით. შეტყობინების მიღებისას მიმღები იმავე ალგორითმით ითვლის მიღებული მონაცემების საკონტროლო ჯამს და მას ადარებს გამგზავნის მიერ დათვლილ მნიშვნელობას. თუ მონაცემები გადაცემისას არ დაზიანებულა (საკონტროლო ჯამები ერთმანეთის ტოლია), მაშინ დასტურდება მათი მიღება. წინააღმდეგ შემთხვევაში, შემოსული მონაცემები იქნება გაუქმებული (განადგურებული) და გამგზავნს უწევს მონაცემების ხელმეორედ გადაცემა. საკონტროლო ჯამებთან ერთად შესაძლებელია გამოყენებული იყოს გადასაცემი მონაცემების სპეციალური კოდირება, შეცდომის აღმომფხვრელი კოდების მეშვეობით. ასეთი კოდირება იძლევა იმის შესაძლებლობას, რომ მონაცემების გარკეულწილად დაზიანების შემთხვევაში (რომელიც მერყეობს დასაშვებ ფარგლებში) ის შეიძლება აღდგენილი იყოს საწყის მდგომარეობამდე. თუ გარკვეული დროითი შუალედის გასვლამდე არ იქნება მიღებული დადასტურება მონაცემების მიღებაზე კავშირის მეორე ბოლოდან, მაშინ ის შეიძლება ჩაითვალოს დაკარგულად და საჭიროა მისი ხელმეორედ გადაგზავნა. იმისთვის, რომ კავშირის არხის მიმღებ ბოლოზე აცილებული იყოს ერთი და იგივე ინფორმაციის ორჯერადი მიღება, უნდა განხორციელდეს შესაბამისი კონტროლი. ამ მიზნით ხორციელდება გადასაცემი მონაცემების გადანომვრა. გარკვეული ნომრის შეტყობინების მიღებისას, რომლის ნომერიც არ შეესაბამება ლოდინის შეტყობინების ნომერს (მიმდევრობის შემდეგ წევრს), უბრალოდ, ჩაითვლება დაკარგულად და დაელოდება სწორი ნომრის მქონე შეტყობინებას.

მსგავსი მოქმედებები შეიძლება დაეკისროს:

- ოპერაციულ სისტემას;
- პროცესებს, რომლებიც ცვლიან მონაცემებს;
- ერთობლივ სისტემასაც და პროცესებსაც, მათი პასუხისმგებლობის განაწილებით.

4.3.7. კავშირის დასრულება

მონაცემთა გაცვლის საშუალებების შესწავლისას მთავარ საკითხს წარმოადგენს გაცვლის შეწყვეტის საკითხი. აქ საჭიროა ორი მნიშვნელოვანი მომენტის გამოყოფა: მოითხოვება თუ არა პროცესის მხრიდან რაიმე სპეციალური მოქმედება კომუნიკაციის საშუალებების გამოყენების შეჩერებაზე და ზემოქმედებს თუ არა ასეთი შეწყვეტა სხვა პროცესების ყოფაქცევაზე. კავშირის ის საშუალებები, რომლებიც ურთიერთქმედებისთვის არ საჭიროებენ ინიციალიზირების, ასევე, არ საჭიროებენ დამატებით მოქმედებებს კავშირის დასრულებისთვის. თუ კავშირის დამყარება

ითხოვდა გარკვეულ ინიციალიზირებას, მაშინ მისი დასრულებისას საჭირო ხდება მთელი რიგი პროცედურების განხორციელება, მაგალითად, ოპერაციული სისტემის ინფორმირება მასთან ასოცირებული რესურსების გამოთავისუფლების თაობაზე.

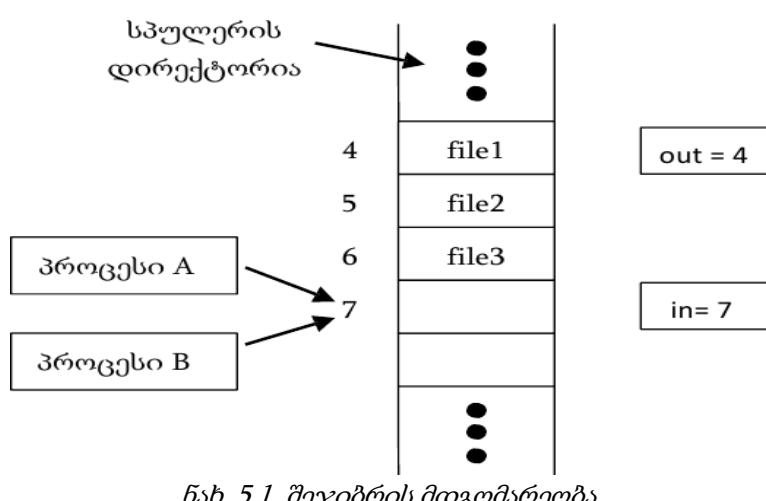
თუ ურთიერთქმედი პროცესები კავშირს წყვეტენ შეთანხმებულად, მაშინ ასეთი შეწყვეტა არ ზემოქმედებს მათ შემდგომ ყოფაქცევაზე. სულ სხვა სურათი გვაქვს ურთიერთქმედი პროცესებიდან ერთ-ერთის მხრიდან შეუთანხმებლად კავშირის გაწყვეტის შემთხვევაში. თუ ურთიერთქმედი პროცესებიდან რამდენიმე იმყოფება მონაცემების მიღების ლოდინის მდგომარეობაში ან შეიძლება გარკვეული დროის შემდეგ ამოჩნდეს ამ მდგომარეობაში და ურთიერთქმედმა ერთ-ერთმა პროცესმა გადაწყვიტა კავშირის გაწყვეტა, მაშინ შეიძლება აღმოჩნდეს, რომ ლოდინის მდგომარეობაში მყოფი პროცესი უსასრულოდ ელოდება (ბლოკირებულია) მონაცემების მიღებას. პროცესის უსასრულოდ ბლოკირების თავიდან აცილების მიზნით ოპერაციული სისტემა ვალდებულია მიიღოს გარკვეული გადაწყვეტილება. ეს გადაწყვეტილება შეიძლება ეხებოდეს ლოდინის მდგომარეობაში მყოფი პროცესის მუშაობის შეწყვეტას, ან მის ინფორმირებას (მაგალითად, წინასწარ განსაზღვრული სიგნალის გადაცემით) კავშირის არხის არარსებობაზე.

4.5. პროცესების სინქრონიზაცია

პროცესები ხშირად ურთიერთქმედებენ ერთმანეთთან (პროცესი შესაძლებელია იყენებდეს სხვა პროცესის ან პროცესების ჯგუფის მიერ მიღებულ შედეგებს, პროცესების ჯგუფში პროცესების ერთი ნაწილი საჭიროებდეს მეორე ნაწილის მიერ დაკავებულ რესურსებს და ა.შ., ამიტომ მნიშვნელოვანია პროცესების ურთიერთქმედების საკითხის შესწავლა).

4.5.1. შეჯიბრის მდგომარეობა

პროცესები შესაძლებელია შეთანხმებულად იყენებდნენ ოპერაციულ სისტემაში არსებულ მონაცემთა საერთო საცავს. ეს საცავი შეიძლება იყოს განთავსებული ოპერატიულ მეხსიერებაში ან წარმოდგენილი იყოს რაიმე ფაილის სახით.



ნახ. 5.1. შეჯიბრის მდგომარეობა

პროცესების ურთიერთქმედების საკითხში უკეთ გასარკვევად განვიხილოთ მაგალითი. განვიხილოთ ბეჭდვის სპულერის კატალოგი. როდესაც პროცესი საჭიროებს რაიმე ფაილის ბეჭდვას ის ათავსებს სპულერის კატალოგში ფაილის სახელს. ოპერაციულ სისტემაში არსებული მეორე პროცესი - **პრინტერის დემონი**, პერიოდულად ამოწმებს სპულერის კატალოგს და იქ

ჩანაწერის (ფაილზე მიმთითებლის) აღმოჩენის შემთხვევაში მიმართავს შესაბამის ფაილს, ბეჭდავს მას და კატალოგიდან შლის ჩანაწერს (ფაილზე მიმთითებელს).

დავუშვათ, სპულერის კატალოგში განთავსებულია გარკვეული რაოდენობის მეხსიერების არები, 1,2,..., რომელთაგან თითოეულში შეიძლება ინახებოდეს ფაილის სახელი. ასევე, დავუშვათ გვაქვს ორი ცვლადი: **in**, რომელიც უთითებს კატალოგში შემდეგ ცარიელ არეზე და **out**, რომელიც უთითებს შემდეგი დასაბეჭდი ფაილის სახელზე. დავუშვათ, დროის გარკვეულ მომენტში 1 – 3 არეები ცარიელია, ხოლო 4 – 6 არეებში კი განთავსებულია დასაბეჭდი ფაილების სახელები (ნახ. 5.1) და თითქმის ერთდროულად ორმა პროცესმა (A და B) გადაწყვიტა საკუთარი მონაცემების განთავსება დასაბეჭდი ფაილების მიმდევრობაში (სპულერის კატალოგში). ამ შემთხვევაში შეიძლება წარმოიქმნას შემდეგი სიტუაცია: A პროცესმა წაიკითხა **in** ცვლადის მნიშვნელობა და ის შეინახა საკუთარ ლოკალურ ცვლადში **next_free_slot** (მომდევნო ცარიელი არე). ამის შემდეგ ტაიმერიდან წყვეტის გამო პროცესორი A პროცესიდან გადაერთო B პროცესზე. B პროცესიც ხედავს **in** ცვლადის მნიშვნელობას და ინახავს მას საკუთარ ლოკალურ ცვლადში (**next_free_slot**). მიმდინარე მომენტისთვის ორივე პროცესი თვლის, რომ საკუთარი მონაცემების განსათავსებლად მათ შეუძლიათ მიმართონ არეს ნომრით 7 (მონაცემების განთავსება სპულერის კატალოგში ხორციელდება ზრდადი მიმდევრობით). ვინაიდან სრულდება B პროცესი, ის ამ არეში ანთავსებს საკუთარ მონაცემებს და **in** ცვლადის მნიშვნელობას ზრდის 1-ით. ამის შემდეგ ის ასრულებს გარკვეულ მოქმედებებს და დროითი კვანტის ამოწურვის გამო პროცესორი გადაერთვება A პროცესზე. (ვგულისხმობთ, რომ დროითი კვანტის განმავლობაში არ განხორციელებულა B პროცესის მიერ დასაბეჭდად გადაცემული ფაილის ბეჭდვა.) ვინაიდან A პროცესი განახლდება იმ ადგილიდან, სადაც მოხდა მისი შეჩერება და მას საკუთარ ლოკალურ ცვლადში ჩაწერილი აქვს მნიშვნელობა 7, ის მიმართავს ამ არეს და იქ განათავსებს საკუთარი ფაილის სახელს, რომელიც გადაეწერება იქ არსებულ სახელს (B პროცესის მიერ ჩაწერილ სახელს). შემდეგ **in** ცვლადის მნიშვნელობას ზრდის 1-ით.

სპულერში შიდა წინააღმდეგობა არ არსებობს, ამიტომ პრინტერის დემონი ვერ დააფიქსირებს მომხდარ ცვლილებას. მომხმარებელი კი, რომელმაც გარკვეული სამუშაოს შესასრულებლად წარმოქმნა B პროცესი, ნაბეჭდი ფორმით ვერასოდეს ვერ მიიღებს მის მიერ განხორციელებული სამუშაოს შედეგს.

სიტუაციას, რომლის დროსაც ორი ან მეტი პროცესი იყენებს საერთო რესურსს და მიღებული შედეგი დამოკიდებულია ბოლოს შესრულებულ პროცესზე ეწოდება **შეჯიბრის მდგომარეობა**.

4.5.2. კრიტიკული სექცია

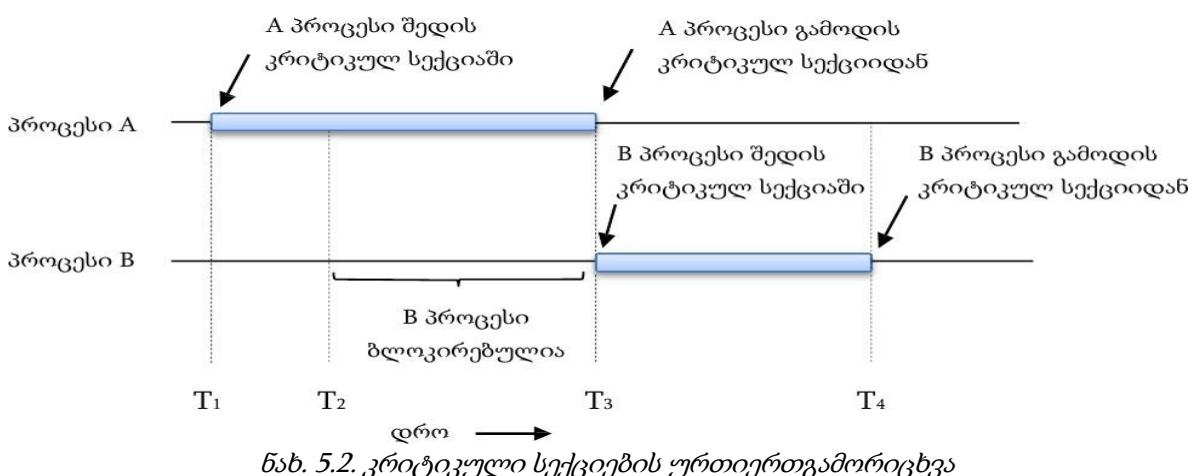
ბუნებრივია ისმის კითხვა: როგორ უნდა ავიცილოთ შეჯიბრის მდგომარეობა? საზოგადოდ, თუ პროცესები იყენებენ „საერთო“ ცნებით დაკავშირებულ რესურსებს, მაშინ შეჯიბრის მდგომარეობის თავიდან აცილების მიზნით საჭიროა განსაზღვრული იყოს მეთოდი, რომელიც გამორიცხავს რამდენიმე პროცესის მხრიდან რესურსებზე ერთდროული წვდომის შესაძლებლობას. სხვა სიტყვებით, რომ ვთქვათ, საჭიროა ურთიერთგამორიცხვის მეთოდი, რომლის მიხედვითაც, თუ რაიმე პროცესი იყენებს სხვა პროცესებთან საზიარო რესურსს, მაშინ სანამ ის არ დაასრულებს ამ რესურსთან მუშაობას სხვა პროცესი ვერ მიიღებს მას. ზემოთ განხილულ მაგალითში პრობლემა მდგომარეობდა იმაში, რომ B პროცესმა იმაზე ადრე მიმართა საზიარო რესურსს (ცვლადს) სანამ A პროცესი დაასრულებდა მასთან მუშაობას. ოპერაციული სისტემების კონსტრუქციის მნიშვნელოვან ნაწილს წარმოადგენს იმ ელემენტარული

მოქმედებების განსაზღვრა, რომლებიც გამოიყენებიან ურთიერთგამორიცხვის მიღწევის მიზნით.

შეჯიბრის მდგომარეობის ფორმულირება შესაძლებელია აბსტრაქტული ფორმით: შესრულებისას პროცესი დაკავებულია გამოთვლებით ან ლოგიკური მოქმედებების შესრულებით, რომელიც შესაძლებელია იწვევდეს შეჯიბრის მდგომარეობას. პროგრამის იმ ნაწილს, რომელშიც გამოიყენება საერთო მეხსიერებაზე წვდომა კრიტიკული სექცია ეწოდება. თუ გვექნებოდა მექანიზმი, რომელიც უზრუნველყოფდა ურთიერთქმედი პროცესებიდან (დროის ნებისმიერ მომენტში) საკუთარ კრიტიკულ სექციაში მხოლოდ ერთი პროცესის არსებობას, მაშინ შევძლებდით შეჯიბრის მდგომარეობის აცილებას.

მიუხედავად იმისა, რომ ეს მოთხოვნა იძლევა შეჯიბრის მდგომარეობის აცილების საშუალებას ის არა საკმარისი იმისთვის, რომ პარალელურმა პროცესებმა სწორად წარმართონ საკუთარი მუშაობა და ეფექტურად გამოიყენონ საზიარო რესურსები. პროცესის მიერ შესრულებული სამუშაო, რომ იყოს მისაღები საჭიროა შემდეგი პირობების შესრულება:

1. ორ პროცესს არ შეუძლია ერთდროულად იმყოფებოდეს საკუთარ კრიტიკულ სექციაში;
 2. არ უნდა არსებობდეს არანაირი წინასწარი მოსაზრება პროცესორების სავარაუდო რაოდენობაზე;
 3. არცერთი პროცესი, რომელიც სრულდება საკუთარი კრიტიკული სექციის გარეთ არ შეიძლება იყოს ბლოკირებული სხვა პროცესის მიერ;
 4. პროცესები უსასრულოდ არ უნდა ელოდებოდნენ საკუთარ კრიტიკულ სექციაში შესვლას.
- ნახ. 5.2-ზე ნაჩვენებია კრიტიკული სექციების ურთიერთგამორიცხვის მაგალითი. A პროცესი შედის საკუთარ კრიტიკულ სექციაში T_1 დროს. T_2 დროს საკუთარ კრიტიკულ სექციაში შესვლას ცდილობს B პროცესი, მაგრამ ვერ ახერხებს იმის გამო, რომ A პროცესი იმყოფება საკუთარ კრიტიკულ სექციაში. ამის შემდეგ ხდება B პროცესის ბლოკირება სანამ A პროცესი არ გამოვა იქიდან. A პროცესის კრიტიკული სექციიდან გამოსვლის შემდეგ B პროცესი გამოდის ბლოკირების მდგომარეობიდან და შედის საკუთარ კრიტიკულ სექციაში T_3 დროს), საიდანაც გამოდის T_4 დროს და ვუბრუნდებით საწყის მდგომარეობას.



4.5.3. ურთიერთგამორიცხვა აქტიური ლოდინით

შემუშავებული იყო ურთიერთგამორიცხვის პრობლემის გადაწყვეტის მრავალი მეთოდი, რომელთაგან თითოეულს გააჩნია საკუთარი უპირატესობა და ნაკლოვანება. განვიხილოთ ზოგიერთი მეთოდი.

4.5.3.1. წყვეტის აკრძალვა

ერთპროცესორული სისტემის შემთხვევაში კრიტიკული სექციის პრობლემის გადაწყვეტა მარტივია. ამ შემთხვევაში შეიძლება პროცეს მიეცეს უფლება განახორციელოს ყველანაირი წყვეტის აკრძალვა და კრიტიკული სექციიდან გამოსვლის შემდეგ კი ჩამოერთვას ის. წყვეტის აკრძალვის შემთხვევაში შეუძლებელი იქნება ტაიმერიდან წყვეტის განხორციელება და შესაბამისად პროცესორიც ვერ შეძლებს სხვა პროცესზე გადართვას. ამ შემთხვევაში არ არსებობს საფრთხე რომელიმე პროცესმა განახორციელოს საკუთარ კრიტიკულ სექციაში შესვლა, მაგრამ შეიძლება წაარმოიშვას სხვა ტიპის პრობლემა. რა შეიძლება მოჰყვეს პროცესის გადაწყვეტილებას აკრძალოს წყვეტა და შემდეგ არ დაუშვას მისი შესაძლებლობა? ამან შესაძლებელია მიგვიყვანოს მთლიანი სისტემის უმოქმედობამდე.

შევნიშნოთ, რომ მრავალპროცესორული სისტემის შემთხვევაში წყვეტის აკრძალვა მოქმედებს მხოლოდ იმ პროცესორზე, რომელსაც შეეხება ბლოკირების ინსტრუქცია. დანარჩენი პროცესორები ჩვეულ რეჟიმში გააგრძელებენ მუშაობას. მიუხედავად იმისა, რომ მრავალპროცესორულ სისტემაში შესაძლებელია ამ მეთოდმა გაამართლოს მაინც დაუშვებელია პროცესისთვის წყვეტის აკრძალვის შესაძლებლობის მიცემა.

4.5.3.2. ცვლადი-ბოქსლომი

ამ მეთოდის გამოყენება გულისხმობს სისტემაში პროცესებისთვის საერთო ცვლადის (ცვლადი-ბოქსლომი) შემოღებას, რომელიც გააკონტროლებს პროცესების შესვლას საკუთარ კრიტიკულ სექციაში. ცვლადი-ბოქსლომის საწყისი ინიციალიზაცია ხდება 0-ვანი მნიშვნელობით, რაც პროცესებისთვის აღნიშნავს, რომ საკუთარ კრიტიკულ სექციაში არ იმყოფება არცერთი პროცესი და ნებისმიერ მათგანს სურვილის შემთხვევაში შეუძლია შევიდეს იქ. პროცესი საკუთარ კრიტიკულ სექციაში შესვლის შემდეგ ცვლის ცვლადი-ბოქსლომის მნიშვნელობას 1- ით, რაც იმის მანიშნებელია, რომ ურთიერთქმედი პროცესებიდან ერთერთი იმყოფება საკუთარ კრიტიკულ სექციაში და, თუ რომელიმე პროცესი საჭიროებს საკუთარ კრიტიკულ სექციაში შესვლას, მაშინ მას მოუწევს დალოდება სანამ მიმდინარე პროცესი არ გამოვა იქიდან. პროცესი საკუთარი კრიტიკული სექციიდან გამოსვლის შემდეგ ცვლად-ბოქსლომს უბრუნებს საწყის მნიშვნელობას (0-ს).

ამ მიდგომასაც გააჩნია ნაკლოვანება, რომლის მსგავსიც უკვე ვნახეთ სპულერის კატალოგის შემთხვევაში. მართლაც, დავუშვათ რაიმე პროცესი ამოწმებს ცვლადი-ბოქსლომის მნიშვნელობას და აღმოაჩინა, რომ ის 0-ის ტოლია. სანამ პროცესი განახორციელებდეს საკუთარ კრიტიკულ სექციაში შესვლას შესაძლებელია მას დამგეგმავმა ჩამოართვას პროცესორი და გადასცა სხვა პროცესს. ახალი პროცესიც შეამოწმებს ცვლადი-ბოქსლომის მნიშვნელობას და ნახავს, რომ ის 0-ის ტოლია, ანუ არცერთი პროცესი არ იმყოფება საკუთარ კრიტიკულ სექციაში. პროცესი შედის საკუთარ კრიტიკულ სექციაში. თუ ახალი პროცესის საკუთარი კრიტიკული სექციიდან გამოსვლამდე პროცესორი დაუბრუნდა პირველ პროცესს, მაშინ რადგანაც ის შესრულებას აგრძელებს შეჩერებული ადგილიდან და შეჩერების დროს ცვლადი-ბოქსლომის მნიშვნელობა იყო 0, ის არ ახდენს ამ მნიშვნელობის გადამოწმებას, შედის საკუთარ კრიტიკულ სექციაში და ცვლადი-ბოქსლომის მნიშვნელობას ზრდის 1-ით. ამრიგად, მივიღეთ, რომ ორი პროცესი ერთდროულად იმყოფება საკუთარ კრიტიკულ სექციაში.

შეიძლება ვიფიქროთ, რომ პრობლემის გადაწყვეტა მარტივია, თუ პროცესს მოვთხოვთ საკუთარ კრიტიკულ სექციაში შესვლამდე ხელმეორედ განახორციელოს ცვლადი-ბოქსლომის მნიშვნელობის შემოწმება. მაგრამ პრობლემის გადაწყვეტილებისთვის ასეთი მოთხოვნაც არაა

საკმარისი. პრობლემა წარმოიშობა იმ შემთხვევაში, როდესაც სხვა პროცესმა შეცვალა ცვლადი-ბოქლომის მნიშვნელობა პირველი პროცესის მიერ ცვლადი-ბოქლომის მნიშვნელობის შემოწმების დასრულებისთანავე.

4.5.3.3. მკაცრი მიმდევრობა

კრიტიკული სექციის პრობლემის გადაწყვეტის შემდეგი მეთოდი გულისხმობს პროცესების წარმოდგენას მკაცრი მიმდევრობის სახით. პროცესები მიმდევრობის მკაცრი დაცვით შედიან საკუთარ კრიტიკულ სექციაში. იქიდან გამოსვლის შემდეგ მიმდევრობის შემდეგ წევრს ეძლევა საკუთარ კრიტიკულ სექციაში შესვლის შესაძლებლობა. ამ მეთოდის საილუსტრაციოდ განვიხილოთ შემდეგი პრაქტიკული ხასიათის მაგალითი. ვთქვათ, მოცემული ორი პროცესი: P₀ და P₁. პროგრამა 5.1-ით წარმოდგენილი ფრაგმენტები განსაზღვრავენ ამ პროცესების მიერ კრიტიკულ სექციაში შესვლის მომენტს.

```
while (TRUE) {
    while (turn != 0); // ცარიელი ციკლი
        critical_region();
    turn = 1; // P1 პროცესს ეძლევა კრიტიკულ
    // სექციაში შესვლის შესაძლებლობა
    noncritical_region();
}
}

a)
```

```
while (TRUE) {
    while (turn != 1); // ცარიელი ციკლი
        critical_region();
    turn = 0; // P0 პროცესს ეძლევა კრიტიკულ
    // სექციაში შესვლის შესაძლებლობა
    noncritical_region();
}
}

b)
```

პროგრამა 5.1. კრიტიკული სექციის პრობლემის პროგრამული გადაწყვეტა; P₀ პროცესი (ა); P₁ პროცესი (ბ) turn ცვლადი განსაზღვრას, თუ რომელი პროცესი უნდა შევიდეს საკუთარ კრიტიკულ სექციაში. საკუთარ კრიტიკულ სექციაში შესვლის წინ ორივე პროცესი ამოწმებს turn ცვლადის მნიშვნელობას. რადგანაც turn = 0, პირველი P₀ პროცესი შედის საკუთარ კრიტიკულ სექციაში, ხოლო P₁პროცესი კი ასრულებს ცარიელ ციკლს. P₀პროცესი საკუთარი კრიტიკული სექციიდან გამოსვლის შემდეგ ზრდის turn ცვლადის მნიშვნელობას 1-ით. რადგანაც უკვე turn = 1, P₁პროცესი გამოდის ცარიელი ციკლიდან და შედის საკუთარ კრიტიკულ სექციაში. იქიდან გამოსვლის შემდეგ turn ცვლადის მნიშვნელობას ხდის 0-ის ტოლად, რითაც P₀ პროცესს საჭიროების შემთხვევაში ეძლევა საკუთარ კრიტიკულ სექციაში შესვლის შესაძლებლობა.

ამ მეთოდის ნაკლოვანება მდგომარეობს იმაში, რომ მიმდევრობაში მყოფი პროცესები ციკლურად შედიან საკუთარ კრიტიკულ სექციაში და, თუ მიმდევრობის ერთი რომელიმე წევრი არ საჭიროებს საკუთარ კრიტიკულ სექციაში ხშირ შესვლას, ხოლო სხვა რომელიმე წევრი კი პირიქით, მაშინ აღმოჩნდება, რომ ამ უკანასკნელს (რომელიც ხშირ შესვლას საჭიროებს) მოუწევს დიდი ხნით ლოდინი სანამ ციკლი დატრიალდება, რაც ეწინააღმდეგება ზემოთ ჩამოყალიბებულ ერთერთ პირობას (ურთიერთქმედი პროცესები არ უნდა ახდენდნენ ერთიმეორის ბლოკირებას).

4.5.3.4. პიტერსონის ალგორითმი

კრიტიკული სექციის პრობლემის გადაწყვეტის ზემოთ ჩამოყალიბებული მეთოდები იძლევიან პრობლემის გადაწყვეტას, მაგრამ ისინი ვერ აკმაყოფილებენ ზემოთ ჩამოყალიბებულ ოთხივე პირობას. 1981 წელს პიტერსონის მიერ შემუშავებული იყო ალგორითმი, რომელმაც დააკმაყოფილა აღნიშნული პირობები (პროგრამა 5.2). პიტერსონის ალგორითმი ეყრდნობა ორი

პროცედურის გამოყენებას. მას აქვს სახე:

```
#define N 2      // პროცესების რაოდენობა
int turn;        // ცვლადი პროცესების რიგითობის განსაზღვრისთვის
int interested[N] = {0, 0};    // ყველა საწყისი მნიშვნელობა 0-ის ტოლია
void enter_region(int process) { // process-ის მნიშვნელობა ტოლია 0-ის ან 1-ის
    int other;           // ცვლადი მეორე პროცესის ნომრისთვის
    other = 1 - process; // პროცესი ავლენს დაინტერესებას საკუთარ კრიტიკულ სექციაში შესვლაზე
    interested[process] = 1;
    turn = process;
    while (turn == process && interested[other] == 1); // ცარიელი ციკლი
}
// პროცესი ტოვებს საკუთარ კრიტიკულ სექციას
void leave_region(int process) {
    // პროცესი გამოდის საკუთარ კრიტიკულ სექციიდან
    interested[process] = 0;
}
```

პროგრამა 5.2. კრიტიკული სექციის გადაწყვეტის პიტერსონის ალგორითმი

საკუთარ კრიტიკულ სექციაში შესვლამდე ყოველი პროცესი იძახებს `enter_region` ფუნქციას და მას პარამეტრის როლში გადასცემს საკუთარ ნომერს (0 ან 1). ეს გამოძახება პროცესს აიძულებს, თუ საჭიროა დაელოდოს საკუთარ კრიტიკულ სექციაში უსაფრთხო შესვლას. საკუთარი კრიტიკულ სექციაში მუშაობის დასრულების შემდეგ ის გამოდის იქიდან და იძახებს ფუნქციას `leave_region`, რითაც მეორე პროცესს აძლევს შესაძლებლობლობას შევიდეს საკუთარ კრიტიკულ სექციაში.

განვიხილოთ ალგორითმის მუშაობის პრინციპი. თავიდან არცერთი პროცესი არ იმყოფება საკუთარ კრიტიკულ სექციაში. შემდეგ პროცესი ნომრით 0 იძახებს ფუნქციას `enter_region`. ის ავლენს დაინტერესებას (`inserted[0] = 1`) კრიტიკულ სექციაში შესვლაზე და `turn = 0`. რადგანაც პროცესი ნომრით 1 ამ მომენტისთვის არ ავლენს დაინტერესებას საკუთარ კრიტიკულ სექციაში შესვლაზე, ამიტომ პროცესი 0 შედის საკუთარ კრიტიკულ სექციაში. თუ პროცესი 1-ი გამოთქვამს სურვილს საკუთარ კრიტიკულ სექციაში შესვლაზე, მაშინ ის იქნება ბლოკირებული სანამ `inserted[0] = 0`, ანუ პროცესი 0 არ გამოიძახებს ფუნქციას `leave_region`.

იმ შემთხვევაში, როდესაც ორივე პროცესი ერთდროულად შეეცდება გამოიძახოს `enter_region` ფუნქცია, ფუნქციის მიერ გათვალისწინებული იქნება ბოლოს შეტანილი მნიშვნელობა. დავუშვათ პროცესი 1 ბოლო შეეცადა შესვლას, მაშინ `turn = 1`. რადგანაც ორივე პროცესმა ერთდროულად გამოიძახა `enter_region` ფუნქცია, ამიტომ `while`-თან ორივე აღმოჩნდება ერთდროულად. პროცესი 0 არ შევა ციკლში და ის აღმოჩნდება საკუთარ კრიტიკულ სექციაში. პროცესი 1 შევა ციკლში და იქიდან ვერ გამოვა სანამ პროცესი 0 არ გამოიძახებს `leave_region` ფუნქციას ანუ არ დატოვებს საკუთარ კრიტიკულ სექციას.

4.5.3.5. აქტივაცია და დეაქტივაცია

პიტერსონის ალგორითმი ქმედუნარიანია, მაგრამ მას მაინც გააჩნია ნაკლოვანება. პრობლემა მდგომარეობს შემდეგში: პროცესი საკუთარ კრიტიკულ სექციაში შესვლის საჭიროების შემთხვევაში ამოწმებს ამ მოქმედების განხორციელების შესაძლებლობას. თუ მას არ შეუძლია კრიტიკულ სექციაში შესვლა ის ასრულებს ცარიელ ციკლს და ელოდება კრიტიკულ სექციაში

შესვლას.

ამ მომენტს არამხოლოდ მივყავართ პროცესორული დროის უსარგებლოდ ხარჯვამდე, არამედ შეიძლება გამოიწვიოს გაურკვეველი შედეგიც. განვიხილოთ ორი პროცესი: H (მაღალპრიორიტეტული) და L (დაბალპრიორიტეტული). მათი მუშაობის დაგეგმვის წესის თანახმად მზადყოფნის მდგომარეობაში მყოფი პროცესების მიმდევრობაში H პროცესი გამოჩენისთანავე დაიკავებს პროცესორს. დავუშვათ დროის მიმდინარე მომენტში, როდესაც L პროცესი იმყოფება საკუთარ კრიტიკულ სექციაში სისტემაში წარმოიქმნა H პროცესი. მაღალი პრიორიტეტის გამო H პროცესი დაიწყებს შესრულებას. თუ აღმოჩნდა, რომ შესრულების მომენტში H გადავიდა აქტიური ლოდინის მდგომარეობაში, მაშინ H-ის მაღალი პრიორიტეტის გამო პირველი უნდა დასრულდეს ის. L ვერასოდეს ვერ გამოვა საკუთარი კრიტიკული სექციიდან, ამიტომ H პროცესს მოუწევს უსასრულო ციკლში ყოფნა.

იმ მიზნით, რომ პროცესების ურთიერთებულებისას არ მოხდეს პროცესორული დროის უსარგებლოდ ხარჯვა, განვიხილოთ გარკვეული პრიმიტივები, რომლებიც ახდენენ სამუშაოს ბლოკირებას სანამ მათთვის არ იქნება ნებადართული კრიტიკულ სექციაში შესვლა. ასეთი პრიმიტივების უმარტივეს ნაკრებს sleep და wakeup. sleep სისტემური გამოძახება ახორციელებს მისი შემსრულებელი პროცესის ბლოკირებას, რომელიც იქნება შეჩერებული სანამ არ მოხდება მისი გააქტიურება სხვა პროცესის მიერ. გააქტიურების wakeup სისტემური გამოძახება ერთ არგუმენტად იყენებს გასააქტიურებელი პროცესის სახელს. sleep და wakeup სისტემური გამოძახებები ერთმანეთთან დასაკავშირებლად დამატებით პარამეტრად იყენებენ მეხსიერების სივრცის ნაწილს.

5.3.6. მომხმარებელი-მწარმოებლის ამოცანა

sleep და wakeup სისტემური გამოძახებების გამოყენების საილუსტრაციოდ განვიხილოთ „მომხმარებელი-მწარმოებლის ამოცანა“ (რომელიც, ასევე, ცნობილია შემოსაზღვრული ბუფერის ამოცანის სახელით). ვთქვათ, ორი პროცესი იყენებს საერთო ბუფერს. ერთი მათგანი ანთავსებს ინფორმაციას ბუფერში, ხოლო მეორე კი ანხორციელებს იქიდან კითხვას (იღებს მონაცემებს). ამ ამოცანაში პრობლემები წარმოიშობა, როდესაც მომხმარებელი ცდილობს წაიკითხოს მონაცემები ცარიელი ბუფერიდან ან როდესაც მწარმოებელი შეეცდება სავსე ბუფერში განათავსოს მონაცემები. პრობლემის გადაწყვეტა პირველ შემთხვევაში მდგომარეობს მომხმარებლის ბლოკირებაში სანამ მწარმოებელი არ განათავსებს ბუფერში ინფორმაციას, მეორე შემთხვევაში კი მწარმოებლის ბლოკირებაში სანამ მომხმარებელი ბუფერიდან არ წაიკითხავს ინფორმაციას.

ერთი შეხედვით პრობლემის გადაწყვეტა მარტივია (პროგრამა 5.3), მაგრამ მას მივყავართ შეჯიბრის მდგომარეობამდე. ბუფერში ჩანაწერების რაოდენობის გასაკონტროლებლად შემოვიდოთ ცვლადი (count). თუ ჩანაწერების მაქსიმალური რაოდენობა უნდა იყოს N, მაშინ მწარმოებლის პროცესმა უნდა შეამოწმოს პირობა count == N. თუ პირობა სრულდება უნდა მოხდეს მწარმოებლის პროცესის ბლოკირება. წინააღმდეგ შემთხვევაში მწარმოებელი შეძლებს ბუფერში ინფორმაციის განთავსებას და გაზრდის count ცვლადს (count++).

მომხმარებლის პროცესიც მუშაობს ანალოგიური წესით: ის ამოწმებს პირობას count == 0. თუ პირობა შესრულდა ხდება მომხმარებლის პროცესის ბლოკირება სანამ მწარმოებელი ბუფერში არ განათავსებს მონაცემებს. თუ პირობა დარღვეულია, მაშინ მომხმარებელი ბუფერიდან იღებს ჩანაწერს და 1-ით ამცირებს count ცვლადს მნიშვნელობას. როგორც მწარმოებელი, ისე მომხმარებელი ამოწმებს, ხომ არ იმყოფება მეორე ბლოკირების მდგომარეობაში. თუ აღმოჩნდა, რომ რომელიმე ბლოკირებულია, მაშინ მეორე ახდენს მის აქტივირებას. პროგრამულ გადაწყვეტას აქვს

სახე:

```
#define N 100 // ბუფერის მოცულობა
int count = 0; // ბუფერშიჩანაწერების რაოდენობა
void producer(void) { // მწარმოებელი
    int item;
    while (TRUE) {
        item = produce_item();
        // ბუფრის გადავსების შემთხვევაში მოხდეს მწარმოებლის ბლოკირება
        if (count == N) sleep();
        insert_item(item); // ბუფერშიჩანაწერის განთავსება
        count++;
        // ბუფერშიჩანაწერის გამოჩენის შემთხვევაში მომხმარებლის გააქტიურება
        if (count == 1)
            wakeup(consumer);
    }
}
void consumer(void) { // მომხმარებელი
    int item; while (TRUE) {
        if (count == 0) sleep(); // თუ ბუფერი ცარიელია მოხდეს ბლოკირება
        item = remove_item(); // ბუფერიდან ჩანაწერის ამოღება
        count--;
        if (count == N - 1)
            wakeup(producer);
        // ბუფერში ცარიელი ადგილის გამოჩენის შემთხვევაში მწარმოებლის აქტივირება
        consume_item();
    }
}
```

პროგრამა 5.3. მომხმარებელი-მწარმოებლის ამოცანის პროგრამული გადაწყვეტა

ამ შემთხვევაში შეჯიბრის მდგომარეობის წარმოშობის მიზეზი შეიძლება იყოს count ცვლადზე თავისუფალი წვდომის შესაძლებლობა. შესაძლებელია წარმოიქმნას შემდეგი სიტუაცია: ბუფერი ცარიელია (count = 0) და მომხმარებელმა ეს დაიმახსოვრა. დამგეგმავი დროებით შეაჩერებს მომხმარებლის პროცესს (ის იქნება „მიძინებული“) და განაახლებს მწარმოებლის პროცესს. მწარმოებელი ბუფერში ანთავსებს მონაცემებს და 1-ით ზრდის count ცვლადის მნიშვნელობას. რადგანაც უკვე count = 1, ის იძახებს wakeup პროცედურას და ახდენს მომხმარებლის პროცესის გააქტიურებას, რომელიც იმყოფებოდა მიძინებულ მდგომარეობაში. ლოგიკურად მომხმარებლის პროცესი არ იმყოფება ბლოკირების მდგომარეობაში, ამიტომ აქტივაციის სიგნალი ზედმეტადაა გამოყენებული. გააქტიურებული მომხმარებელი შესრულების წინ ამოწმებს მთვლელის შენახულ მნიშვნელობას (count = 0) და ისევ უბრუნდება მიმდინარე მდგომარეობას. ადრე თუ გვიან ბუფერი გადაივსება, რაც გამოიწვევს მწარმოებლის პროცესის მიძინებას. შედეგად ორივე პროცესი იქნება შეჩერებული (იქნება მიძინებულ მდგომარეობაში). წარმოქმნილი პრობლემა მდგომარეობს საკუთარი სურვილით მიძინებულ მდგომარეობაში გადასული პროცესის მიმართ wakeup პროცედურის არასწორად გამოყენებაში. პრობლემის გადაწყვეტა საჭიროებს დამატებითი ცვლადის შემოღებას, რომელიც გააკონტროლებს აქტივაციების რაოდენობას. თუ პროცესი საჭიროებს აქტივირებას, მაშინ მისთვის მოდხება wakeup პროცედურის განხორციელება.

4.5.4. სემაფორები

აქტივაციების რაოდენობის დასათვლელად დეიქსტრის მიერ შემოღებული იყო ახალი ტიპი **სემაფორი** (semaphore). თუ სემაფორის მნიშვნელობა 0-ის ტოლია, მაშინ შენახული აქტივაცია არ გვაქვს, წინააღმდეგ შემთხვევაში გვაქვს შესაბამისი რაოდენობის აქტივაცია.

დეიქსტრის მიერ შეთავაზებული იყო ორი ოპერაცია down და up. ის ამოწმებს სემაფორი არის თუ არა 0-ის ტოლი. თუ ის 0-სგან განსხვავებულია, მაშინ მისი მნიშვნელობა მცირდება 1-ით და ხდება შენახული აქტივაციის გააქტიურება. თუ ის 0-ის ტოლია, მაშინ down ოპერაციის შესრულების გარეშე ხორციელდება პროცესის ბლოკირება.

სემაფორის მნიშვნელობის შემოწმება, მისი შეცვლა და პროცესის შეჩერება შეიძლება განხილული იყოს როგორც გაუყოფელი, ატომარული ოპერაცია. მათი ატომარულობა გარანტირებს, რომ სხვადასხვა პროცესები ერთდროულად არ აღმოჩნდებიან საკუთარ კრიტიკულ სექციაში.

up ოპერაცია სემაფორის მიერ დამახსოვრებულ მნიშვნელობას ზრდის 1-ით. თუ სემაფორთან დაკავშირებულია ერთი ან რამდენიმე შეჩერებული პროცესი, რომელთაც შეუძლიათ down ოპერაციის განხორციელება, სისტემა ირჩევს ერთერთ მათგანს და მას აძლევს down ოპერაციის განხორციელების შესაძლებლობას. ამრიგად, სემაფორთან მიმართებაში up ოპერაციის განხორციელების შემდეგ (რომელთანაც დაკავშირებული იყო სემაფორი) მისი მნიშვნელობა იქნება 0, მაგრამ შეჩერებული პროცესების რაოდენობა შემცირდება 1-ით. სემაფორის მნიშვნელობის გაზრდის და შესასრულებლად პროცესის არჩევის ოპერაცია განიხილება, როგორც გაუყოფელი ოპერაცია. არცერთი პროცესი არ შეიძლება იყოს ბლოკირებული up ოპერაციის შესრულების მომენტში, ისევე როგორც არ შეიძლება იყოს ბლოკირებული wakeup() პროცედურის განხორციელებისას.

4.5.4.1 სემაფორების გამოყენებით მწარმოებელი-მომხმარებლის ამოცანის გადაწყვეტა

პროგრამა 5.4-ით ნაჩვენებია სემაფორების გამოყენებით აქტივაციის პრობლემის გადაწყვეტა.

ბუნებრივი იქნებოდა down და up ოპერაციების რეალიზება სისტემური გამოძახებების გამოყე- ნებით, რომ ოპერაციულმა სისტემამ სემაფორის შემოწმების, მისი მნიშვნელობის შეცვლის ან პროცესის შეჩერების აუცილებლობის შემთხვევაში დროებით აკრძალოს ყველანაირი წყვეტა. ვინაიდან ეს მოქმედებები იყენებენ რამდენიმე ბრძანებას, ამიტომ წყვეტის აკრძალვა არ იქნებოდა ზიანის მომტანი. მრავალპროცესორულ სისტემებში საჭიროა სემაფორების დაცვა იმის გარანტირებისთვის, რომ დროის ნებისმიერ მომენტში ის იმუშავებს მხოლოდ ერთ პროცესთან.

```
#define N 100 // ბუფერში ადგილების რაოდენობა
typedef int semaphore; // სემაფორი - int ტიპის ცვლადების სპეციალური ნაირსახეობა
semaphore mutex = 1; // ცვლადი კრიტიკულ სექციაზე დაშვების სამართავად
semaphore empty = N; // ცვლადი ბუფერში ცარიელი ადგილების გასაკონტროლებლად
semaphore full = 0; // ცვლადი ბუფერში დაკავებული ადგილების გასაკონტროლებლად
void producer(void){
    int item;
    while (TRUE) {
        item = produce_item(); // ინფორმაციის გენერირება ბუფერში განსათავსებლად
```

```

        down(&empty);    // ბუფერში ცარიელი ადგილების მთვლელის შემცირება
        down(&mutex);   // კრიტიკულ სექციაში შესვლა
        insert_item(item); // ბუფერში ახალი ინფორმაციის დამატება
        up(&mutex);     // კრიტიკული სექციიდან გამოსვლა
        up(&full);      // ბუფერში დაკავებული ადგილების მთვლელის გაზრდა
    }
}

void consumer(void) { int item;
    while (TRUE) {
        down(&full);    // ბუფერში დაკავებული ადგილების მთვლელის შემცირება
        down(&mutex);   // კრიტიკულ სექციაში შესვლა
        item = remove_item(); // ბუფერიდან ინფორმაციის ამოღება
        up(&mutex);     // კრიტიკული სექციიდან გამოსვლა
        up(&empty);     // ბუფერში ცარიელი ადგილების მთვლელის გაზრდა
        consume_item(item); // ჩანაწერებთან მუშაობა
    }
}

```

პროგრამა 5.4. სემაფორების გამოყენებით მწარმოებელი-მომხმარებლის ამოცანის გადაწყვეტა

სემაფორების მეშვეობით მწარმოებელი-მომხმარებლის ამოცანის გადაწყვეტაში გამოყენებულია სამი სემაფორი: full (რომელიც ითვლის ბუფერში დაკავებული ადგილების რაოდენობას), empty (რომელიც ითვლის ბუფერში ცარიელი ადგილების რაოდენობას) და mutex (რომელიც უზრუნველყოფს, რომ მწარმოებელი და მომხმარებელი ერთდროულად არ აღმოჩნდნენ ბუფერში). თავიდან full = 0, mutex = 1 და empty ბუფერის მოცულობის ტოლია (N). სემაფორს, რომელიც ინიციალიზირეულია 1-ით და გამოიყენება ორი ან რამდენიმე პროცესის საკუთარ კრიტიკულ სექციაში შესვლის გასაკონტროლებლად ეწოდება ორმაგი სემაფორი. ურთიერთბლოკირების გარანტირება შესაძლებელია, თუ ყოველი პროცესი საკუთარ კრიტიკულ სექციაში შესვლამდე განახორციელებს down ოპერაციას, ხოლო იქიდან გამოსვლის შემდეგ კი - აც აპერაციას.

პროგრამა 5.4-ში გამოყენებული სემაფორების გამოიყენება 2 განსხვავებული მიზნით: mutex სემაფორი გამოიყენება ურთიერთბლოკირების პრობლემის გადასაწყვეტად. მის დანიშნულებას წარმოადგენს იმის გარანტირება, რომ დროის ნებისმიერ მომენტში ბუფერზე და კითხვის და ჩაწერის შესაბამის ცვლადებზე ერთდროულად წვდომა არ ქონდეს ერთზე მეტ პროცესს. სხვა სემაფორები გამოიყენება პროცესების სინქრონიზაციისთვის. full და empty გამოიყენება კონკრეტული მოვლენის დადგომა/არდადგომის გარანტირებისთვის. მოცემულ მაგალითში ისინი გარანტირებენ, რომ მწარმოებელი შეწყვეტს საკუთარ შესრულებას, თუ ბუფერი გადავსებულია და მომხმარებელი შეწყვეტს საკუთარ შესრულებას, თუ ბუფერი ცარიელია.

4.5.5. მონიტორები

მიუხედავად სემაფორების მეშვეობით მწარმოებელი-მომხმარებლის ამოცანის მოხერხებული გადაწყვეტისა, სემაფორების გამოყენება პროგრამირებისას მოითხოვს დიდ ყურადღებას და სიფრთხილეს. დავუშვათ, რომ განხილულ მაგალითში შემთხვევით მოხდა პირველად down(&mutex) ოპერაციის შესრულება, ხოლო შემდეგ კი იმავე ოპერაციის შესრულება empty და full სემაფორებისთვის. დავუშვათ, რომ მომხმარებელმა, რომელიც შევიდა საკუთარ

კრიტიკულ სექციაში აღმოაჩინა, რომ ბუფერი ცარიელია. ხდება მისი ბლოკირება და ელოდება შეტყობინების გამოჩენას. მაგრამ მწარმოებელს არ შეუძლია საკუთარ კრიტიკულ სექციაში შესვლა ინფორმაციის გადასაცემად, ვინაიდან ის ბლოკირებულია მომხმარებლის მიერ. ვდებულობთ ურთიერთბლოკირების სიტუაციას.

რთულ პროგრამებში სემაფორების სწორად გამოყენების ანალიზის ჩატარება არც ისე ადვილია. პროგრამების შესრულება არ იძლევა შედეგს, ვინაიდან შეცდომის წარმოშობა დამოკიდებულია ატომარული ოპერაციების მონაცვლეობაზე და შეცდომები შეიძლება იყოს ძნელად გამეორებადი. პროგრამისტის მუშაობის შემსუბუქების მიზნით 1974 წ. ჰორის (Hoare) მიერ შემოთავაზებული იყო უფრო მაღალი დონის მექანიზმი ვიდრე სემაფორები, რომელსაც მონიტორი ეწოდა.

მონიტორი წარმოადგენს მონაცემთა ტიპს, რომელიც შეიძლება ჩადებული იქნას ობიექტზე ორიენტირებულ პროგრამირების ენებში. მონიტორებს გააჩნიათ საკუთარი ცვლადები, რომლებიც განსაზღვრავენ მის მდგომარეობას. ამ ცვლადების მნიშვნელობების შეცვლა შესაძლებელია ფუნქცია-მეთოდის მეშვეობით, რომლებიც მიეკუთვნება მონიტორს. თავის მხრივ, ფუნქცია-მეთოდები შეიძლება იყენებდნენ მხოლოდ მონიტორის შიგნით არსებულ მონაცემებს და საკუთარ პარამეტრებს. მონიტორის სტრუქტურის აღწერა შესაძლებელია შემდეგნაირად:

```
monitor monitor_name{
    // შიდა ცვლადების აღწერა; void m1(...){ ... }
    ...
    void mN(...){ ... }
}
// ცვლადების ინიციალიზაციის ბლოკი;
```

აქ m1,..., mN ფუნქციები წარმოადგენენ მონიტორის ფუნქცია-მეთოდებს, ხოლო ცვლადების ინიციალიზაციის ბლოკი შეიცავს ოპერაციებს, რომლებიც სრულდებიან მხოლოდ ერთხელ: ან მონიტორის შექმნისას ან რომელიმე ფუნქცია-მეთოდის პირველივე გამოძახებისას მის შესრულებამდე.

მონიტორების მნიშვნელოვან თავისებურებას წარმოადგენს ის, რომ დროის ნებისმიერ მომენტში მოცემული მონიტორის შიგნით შეიძლება აქტიური იყოს მხოლოდ ერთი პროცესი. ვინაიდან მონიტორები წარმოადგენენ პროგრამირების ენის განსაკუთრებულ კონსტრუქციებს, კომპილიატორს შეუძლია განასხვაოს მონიტორის ფუნქციის გამოძახება სხვა ფუნქციების გამოძახებისგან და დაამუშაოს ის სპეციალური წესით, მისთვის ურთიერთგამორიცხვის რეალიზაციის პროლოგი/ეპილოგის დამატებით. რადგანაც ურთიერთგამორიცხვის მექანიზმის აგების ვალდებულება ეკისრება კომპილიატორს და არა პროგრამისტს, პროგრამისტის მუშაობა მონიტორების გამოყენებისას საკმაოდ შემსუბუქებულია, ხოლო შეცდომების წარმოქმნის შესაძლებლობა კი მინიმალური.

პროცესების ურთიერთქმედებისას წარმოშობილი ამოცანის გადაწყვეტის რეალიზებისთვის მხოლოდ ურთიერთგამორიცხვა არაა საკმარისი. საჭიროა პროცესების მიღევრობის ორგანიზების ისეთი საშუალებები, როგორიცაა წინა მაგალითში გამოყენებული full და empty სემაფორები. ამ მიზნით მონიტორებში შემოღებულია პირობითი ცვლადის ცნება (*condition variables*), რომლებზეც შესაძლებელია (სემაფორებისთვის down და up ოპერაციების მსგავსი) wait და signal ოპერაციის განხორციელება.

თუ მონიტორის ფუნქციას გარკვეული მოვლენის დადგომამდე არ შეუძლია შემდგომი

შესრულება რომელიმე პირობით ცვლადზე ასრულებს ოპერაციას wait. ამასთან, ხორციელდება wait ოპერაციის შემსრულებელი პროცესის ბლოკირება და სხვა პროცესი ღებულობს მონიტორში შესვლის შესაძლებლობას.

როდესაც ლოდინის მოვლენა დაფიქსირდება, სხვა პროცესი ფუნქცია-მეთოდის შიგნით იმავე პირობით ცვლადზე ახორციელებს ოპერაციას signal. რასაც მივყავართ ადრე ბლოკირებული პროცესის გააქტიურებამდე. თუ ამ ცვლადისთვის რამდენიმე პროცესი ელოდებოდა signal იმისთვის, მაშინ მათ შორის მხოლოდ ერთი ხდება აქტიური. რისი გაკეთებაა შესაძლებელი იმისთვის, რომ მონიტორის შიგნით არ აღმოგვაჩნდეს ორი ერთდროულად აქტიური პროცესი? ჰორის მიერ შემოთავაზებული იყო მექანიზმი, რომლის დროსაც გააქტიურებული პროცესი აჩერებს მის გამააქტიურებელ პროცესს სანამ ეს უკანასკნელი არ დატოვებს მონიტორს. მოგვიანებით ჰანსენის (Hansen) მიერ იყო შემოთავაზებული შემდეგი მექანიზმი: გამააქტიურებელი პროცესი ტოვებს მონიტორს signal იმისთვის შესრულების შემდეგ.

უნდა აღინიშნოს, რომ პირობითი ცვლადებს, დეიქსტრის სემაფორებისგან განსხვავებით, შეუძლიათ წინაისტორიის დამახსოვრება. რაც ნიშნავს, რომ signal ოპერაცია ყოველთვის უნდა შესრულდეს wait ოპერაციის შემდეგ. თუ ოპერაცია signal სრულდება პირობით ცვლადზე, რომელიც არაა დაკავშირებული არცერთ ბლოკირებულ პროცესთან, მაშინ ინფორმაცია მომხდარ მოვლენაზე იქნება უკუგდებული. მაშასადამე, wait ოპერაციის შესრულება ყოველთვის მიგვიყვანს პროცესის ბლოკირებამდე. მომხმარებელი-მწარმოებლის ამოცანის გადაწყვეტა მონიტორების გამოყენებით გამოიყურება შემდეგნაირად:

```

monitor ProducerConsumer {
    condition full, empty;
    int count;
    void put() {
        if(count == N)
            full.wait;
        put_item();
        count++;
        if(count == 1)
            empty.signal;
    }
    void get() {
        if(count == 0)
            empty.wait;
        get_item();
        count--;
        if(count == N-1)
            full.signal;
    }
    { count = 0; }
}

Producer:
while(1) {
    produce_item;
    ProducerConsumer.put();
}

Consumer:
while(1) {

```

```
    ProducerConsumer.get();  
    consume_item;  
}
```

ადვილად შესამოწმებელია, რომ მოყვანილი მაგალითი ნამდვილად წარმოადგენს დასმული ამოცანის გადაწყვეტას.

მონიტორების რეალიზაცია ითხოვს სპეციალური პროგრამირების ენების და მათთვის კომპილიატორების შემუშავებას. მონიტორები გვხვდებიან ისეთ ენებში როგორიცაა Pascal, Java და ა.შ. სისტემური გამოძახებების მეშვეობით მონიტორების ემულაცია ჩვეულებრივი ფართოდ გამოყენებადი პროგრამირების ენებისთვის არც ისე ადვილია, როგორც სემაფორებისთვის. ამიტომ შესაძლებელია კიდევ ერთი მექანიზმის გამოყენება, რომელსაც გაჩნია ურთიერთგამორიცხვის უხილავი საშუალებები - შეტყობინების გადაცემა.

ლექცია 6. ურთიერთბლოკირება

თანამედროვე გამოთვლითი მანქანა აღჭურვილია მრავალი რესურსით (როგორიცაა პროცესორი, მყარი დისკი, მეხსიერება, შეტანა/გამოტანის მოწყობილობები და ა.შ.). ოპერაციულ სისტემაში წარმოქმნილი ყოველი პროცესი მიმართავს მას იმ რესურსის გამოყოფის მოთხოვნით, რომელსაც პროცესი საჭიროებს დასახული ამოცანის გადასაწყვეტად. დროის ნებისმიერ მომენტში კონკრეტული რესურსი შეიძლება გამოეყოს მხოლოდ ერთ პროცესს. რამდენიმე პროცესი შეიძლება მიმართავდეს ოპერაციულ სისტემას ერთიდაიმავე რესურსის გამოყოფაზე. პროცესებისთვის საზიარო რესურსის გამოყოფა შესაძლებელია დაკავშირებული იყოს გარკვეულ პრობლემებთან, ამიტომ ოპერაციული სისტემა პროცესებს გამოუყოფს რესურსებს მხოლოდ მცირე დროითი შუალედით, მათზე დაშვების სრული უფლებებით.

ხშირად პროცესები საკუთარი სამუშაოს შესასრულებლად საჭიროებენ ერთზე მეტ რესურსს. ოპერაციულმა სისტემამ პროცესის მიერ მოთხოვნილი რესურსები შეიძლება გამოეყოს მას მოთხოვნის გაკეთების მიმდევრობის შესაბამისად სათითაოდ ან ერთიანად. ორივე მიდგომას გააჩნია უპირატესობა და ნაკლოვანება. მაგალითად, განვიხილოთ სიტუაცია, რომლის დროსაც სისტემაში გვაქვს ორი პროცესი, P₁, P₂, და ორი რესურსი: სკანერი და დისკური მოწყობოლობა. დავუშვათ, რომ ორივე პროცესი საჭიროებს თითოეულ რესურსს და რესურსების გამოყოფა ხორციელდება პროცესებისთვის თითოთითოდ. დავუთვათ, P₁ პროცესი დაპროგრამირებულია ისე, რომ მან პირველი გამოიყენოს სკანერი და მეორე დისკური მოწყობილობა, ხოლო P₂ პროცესი კი პირიქით. ცხადია, რომ თუ ორივე პროცესი „ერთდროულად“ გვყავს სისტემაში, მაშინ პირველი P₁ პროცესს გამოეყოფა სკანერი, ხოლო P₂ -ს დისკური მოწყობილობა. P₁ პროცესი სკანერთან მუშაობის დასრულების და მისგან მონაცემების მიღების შემდეგ საჭიროებს მეორე რესურსს და აკეთებს შესაბამის მოთხოვნას რესურსის გამოყოფაზე. მაგრამ მისი დაკმაყოფილება იქნება შეუძლებელი, ვინაიდან დისკურ მოწყობილობას იკავებს P₂ პროცესი. ანალოგურად, P₂ პროცესი დისკურ მოწყობილობასთან მუშაობის დასრულების შემდეგ გააკეთებს მოთხოვნას სკანერზე და მისი დაკმაყოფილებაც, ასევე, იქნება შეუძლებელი. ასეთ შემთხვევაში ორივე პროცესი იქნება ბლოკირებული სანამ არ გამოთავისუფლდება მათ მიერ მოთხოვნილი რესურსი, რაც არ მოხდება თუ არ მოხდა ოპერაციული სისტემის მიერ რომელიმე პროცესის შესრულების შეჩერება. წარმოქმნილ სიტუაციას ეწოდება **ურთიერთბლოკირება (deadlock) ან ჩიხი.**

6.1. რესურსები

კომპიუტერულ სისტემას გააჩნია მრავალი რესურსი, რომელთა შეთავაზება მას შეუძლია პროცესებისთვის საკუთარი სამუშაოს განსახორციელებლად. რესურსი შეიძლება იყო აპარატული (მყარი დისკი, პროცესორი, მეხსიერება) ან ინფორმაციული (ფაილები, მონაცემთა ბაზა). როგორც უკვე აღვნიშნეთ, ყოველი რესურსი დროის ნებისმიერ მომენტში შესაძლებელია გამოეყოს მხოლოდ ერთ პროცესს. შესაბამისად, თუ რამდენიმე პროცესი საჭიროებს ერთიდაიმავე რესურსს მათ ამ რესურსის მიღება შეუძლიათ რიგრიგობით. თუ ყოველი რესურსი ოპერაციულ სისტემაში წარმოდგენილი იქნებოდა რამდენიმე ასლის სახით, მაშინ ერთი კონკრეტული რესურსის მომთხოვნი რამდენიმე პროცესის დაკმაყოფილება შესაძლებელი იქნებოდა ერთროულად.

6.1.1. განაწილებადი და გაუნაწილებელი რესურსი

ოპერაციულ სისტემაში წარმოდგენილი ყოველი რესურსი შეიძლება იყოს ორი სახის:

განაწილებადი ან გაუნაწილებელი. პროცესის მიერ დაკავებულ რესურსს, რომლის ჩამორთმევაც მისთვის შესაძლებელია „უმტკივნეულოდ“, მიეკუთვნება განაწილებად რესურსს. ამ ტიპის რესურსის მაგალითს წარმოადგენს მეხსიერება. სისტემაში არასაკმარისი მეხსიერების არსებობის შემთხვევაში ხორციელდება პროცესის სრული ან ნაწილობრივი გადატანა მეხსიერების არიდან დისკუზე, ხოლო მოგვიანებით კი მისი უკან დაბრუნება.

პროცესისთვის გაუნაწილებელი რესურსის ჩამორთმევამ შესაძლებელია გამოიწვიოს პროცესის მიერ შესრულებული მნიშვნელოვანი სამუშაოს დაკარგვა. მაგალითად, პროცესისთვის, რომელიც დისკური მოწყობილობის გამოყენებით ანხორციელებს კომპაქტ-დისკუზე მონაცემების ჩაწერას მის დასრულებამდე დისკური მოწყობილობის ჩამორთმევა დაუშვებელია. რესურსის ჩამორთმევის შემთხვევაში ინფორმაციის ჩაწერა ვერ განხორციელდება და კომპაქტ-დისკუც გახდება უსარგებლო ხელმეორედ გამოყენებისათვის. ურთიერთბლოკირებაში შესაძლებელია მონაწილეობდეს როგორც განაწილებადი, ისე გაუნაწილებელი რესურსი. რესურსების საგულდაგულო გადანაწილების ხარჯზე შესაძლებელია განაწილებადი რესურსებით გამოწვეული ურთიერთბლოკირების აცილება. ამიტომ ყურადღებას გავამახვილებთ მხოლოდ გაუნაწილებელი რესურსებით გამოწვეულ ურთიერთბლოკირებასთან გამკვლავების მეთოდებზე.

საზოგადოდ, რესურსების გამოყენება ხორციელდება მოვლენათა შემდეგი მიმდევრობით:

1. მოთხოვნა;
2. გამოყენება;
3. გამოთავისუფლება.

თუ პროცესის მიერ რესურსზე გაკეთებული მოთხოვნის მომენტში ის მიუწვდომელია (რესურსი დაკავებულია სხვა პროცესის მიერ), მაშინ პროცესს იძულებით უწევს ლოდინის მდგომარეობაში გადასვლა. ზოგიერთ ოპერაციულ სისტემაში მსგავს შემთხვევებში პროცესი გადადის ბლოკირების მდგომარეობაში და როგორც კი რესურსი გახდება ხელმისაწვდომი ის გამოდის იქიდან და იღებს მას. სხვა სისტემებში რესურსის არ გამოყოფა თანდართულია შეცდომის კოდით და შემდგომი მოქმედებების განხორციელება (გააკეთოს რესურსზე მოთხოვნა ხელმეორედ თუ დაელოდოს მას) ეკისრება რესურსის მომთხოვნ პროცესს.

6.2. ურთიერთბლოკირება

ვთქვათ, მოცემული გვაქვს პროცესების რაიმე ჯგუფი.

ამბობენ, რომ პროცესების ჯგუფი იმყოფება **ურთიერთბლოკირების მდგომარეობაში**, თუ ამ ჯგუფის ყოველი პროცესი ელოდება მოვლენას, რომელიც შესაძლებელია გამოიწვიოს მხოლოდ ამავე ჯგუფის რომელიმე სხვა პროცესმა.

ვინაიდან ურთიერთბლოკირების მდგომარეობაში მყოფი პროცესების ჯგუფის ყოველი პროცესი იმყოფება ლოდინის მდგომარეობაში, ამიტომ არცერთს არ შეუძლია გამოიწვიოს რაიმე მოვლენა, რომელმაც შესაძლებელია ლოდინის მდგომარეობიდან გამოიყვანოს ამავე ჯგუფის სხვა რომელიმე პროცესი. შესაბამისად, ჯგუფის პროცესები იმყოფებიან უსარულო ლოდინის მდგომარეობაში. ამ მოდელში იგულისხმება, რომ ჯგუფი შედგება მხოლოდ ტრადიციული (ერთ ნაკადიანი) პროცესებისგან და წყვეტა, რომელმაც შესაძლებელია რომელიმე პროცესი გამოიყვანოს ლოდინის მდგომარეობიდან არ გაგვაჩნია. წყვეტის არარსებობის მოთხოვნა აუცილებელია იმისთვის, რომ რომელიმე პროცესმა რაიმე მიზეზით არ განაახლოს შესრულება.

უმეტეს შემთხვევაში მოვლენა, რომელსაც ელოდება ჯგუფის თითოეული პროცესი არის ამავე

ჯგუფის სხვა პროცესის მიერ დაკავებული რესურსის გამოთავისუფლება. სხვა სიტყვებით, რომ ვთქვათ, ჯგუფის ყოველი წევრი ელოდება სხვა წევრის მიერ დაკავებული რესურსის გამოთავისუფლებას.

შევნიშნოთ, რომ ამ შემთხვევაში რესურსების რაოდენობას, სახეობას და ჯგუფში პროცესების რაოდენობას გადამწყვეტი მნიშვნელობა არ აქვს.

6.2.1. ურთიერთბლოკირების წარმოქმნის აუცილებელი და საკმარისი პირობები

კველევების შედეგად მეცნიერებმა დაადგინეს, რომ ურთიერთბლოკირების წარმოქმნისთვის აუცილებელია და საკმარისი შემდეგი 4 პირობის შესრულება:

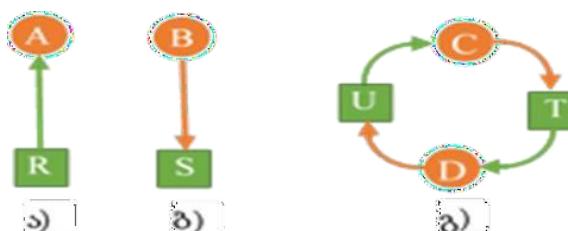
- ურთიერთგამორიცხვის პირობა (mutual exclusion).** დროის ნებისმიერ მომენტში რესურსი ან გამოყოფილია მხოლოდ ერთი პროცესისთვის ან თავისუფალია;
- რესურსის ლოდინის პირობა (hold and wait).** პროცესს, რომელსაც დაკავებული აქვს გარკვეული რესურსი შეუძლია მოითხოვოს ახალი რესურსი;
- გაუნაწილებლობის პირობა (no preemption).** პროცესისათვის მის მიერ დაკავებული რესურსის იძულებითი ჩამორთმევა შეუძლებელია. პროცესმა დაკავებული რესურსი უნდა გამოათავისუფლოს საკუთარი სურვილით;
- ციკლური ლოდინის პირობა (circular wait).** უნდა არსებობდეს ორი ან მეტი პროცესისგან შემდგარი წრიული მიმდევრობა, რომელშიც მიმდევრობის ყოველი წევრი ელოდება მის შემდეგ მდგომი წევრის მიერ დაკავებული რესურსის გამონთავისუფლებას.

ჩამოყალიბებული პირობებიდან ერთერთის დარღვევის შემთხვევაში სისტემაში არ გვაქვს ურთიერთბლოკირების მდგომარეობა.

უნდა აღინიშნოს, რომ ჩამოყალიბებული პირობები არ არღვევს იმ პოლიტიკას, რომელსაც სისტემა ემხრობა ან არა.

6.2.2. ურთიერთბლოკირების მოდელირება

ზემოთ ჩამოყალიბებული პირობები შესაძლებელია მოდელირებული იყოს გრაფთა თეორიის გამოყენებით. ყოველ გრაფს გააჩნია ორი სახის კვანძი: პროცესი, რომელიც გამოსახულია წრის ფორმით, და რესურსი, რომელიც გამოსახულია კვადრატის ფორმით. მიმართული მონაკვეთი რესურსიდან პროცესისკენ აღნიშნავს, რომ პროცესმა მოითხოვა შესაბამისი რესურსი და ის იკავებს მას (ნახ. 6.1. ა)). პროცესიდან რესურსისკენ მიმართული მონაკვეთი აღნიშნავს, რომ პროცესი ელოდება შესაბამის რესურსს და ამის გამო ის ბლოკირებულია (ნახ. 6.1. ბ)). ნახ. 6.1. გ)-ზე კი გამოსახულია შემდეგი სიტუაცია: C პროცესი იკავებს U რესურსს და ელოდება T რესურსს, რომელსაც იკავებს D პროცესი, ხოლო D კი პირიქით, იკავებს T რესურსს და ელოდება U-ს.



ნახ. 6.1. რესურსების განაწილების გრაფი: რესურსი

შევნიშნოთ, რომ კავშირულ სტრუქტურების მოხსენენ (გ); ურთიერთბლობულ კონფლიქტების, რომელიც

გვეხმარება გავერკვეთ რესურსებზე მოთხოვნების მიმდევრობას მივყავართ თუ არა ურთიერთბლოკირებამდე. ის არ იძლევა ურთიერთბლოკირების პრობლემის გადაწყვეტას.

ურთიერთბლოკირების პრობლემის გადასაწყვეტად გამოიყენება შემდეგი მიდგომა:

- პრობლემის იგნორირება;
- პრობლემის აღმოჩენა და აღმოფხვრა;
- პრობლემის დინამიური აცილება (რესურსების გადანაწილება);
- ურთიერთბლოკირების აუცილებელი და საკმარისი პირობებიდან ერთერთის დარღვევა.

6.3. პრობლემის იგნორირება

ურთიერთბლოკირების პრობლემის გადაწყვეტის მარტივი მიდგომა მდგომარეობს პრობლემის იგნორირებაში (პრობლემის არ დანახვა). იმისთვის, რომ მიღებული იქნას გადაწყვეტილება საჭიროა განხორციელდეს ურთიერთბლოკირების წარმოშობის ალბათობის და აპარატურული და პროგრამული უზრუნველყოფის მტყუნებით გამოწვეული ზიანის ალბათობის შეფასება და მათი ერთიმეორისთვის შედარება. მაგალითად, მომხმარებელი, მხოლოდ იმის გამო, რომ სისტემა 5 წელიწადში ერთხელ ხვდება ურთიერთბლოკირების მდგომარეობაში, არ დათანხმდება ისეთი სისტემის გამოყენებას, რომელიც კვირაში ერთი დღე ვერ ფუნქციონირებს ნორმალურად აპარატული ან პროგრამული გაუმართაობით გამოწვეული პრობლემების გამო.

6.4. პრობლემის აღმოჩენა და სისტემის ქმედუნარიანობის აღდგენა

ამ მიდგომის გამოყენებისას სისტემა არ ცდილობს ურთიერთბლოკირების აცილებას. ის უშვებს მის წარმოშობას, ცდილობს მისი წარმოშობის მიზეზების დადგენას და შემდეგ სისტემის ქმედუნარიანობის აღსადგენად ანხორციელებს შესაბამის მოქმედებებს.

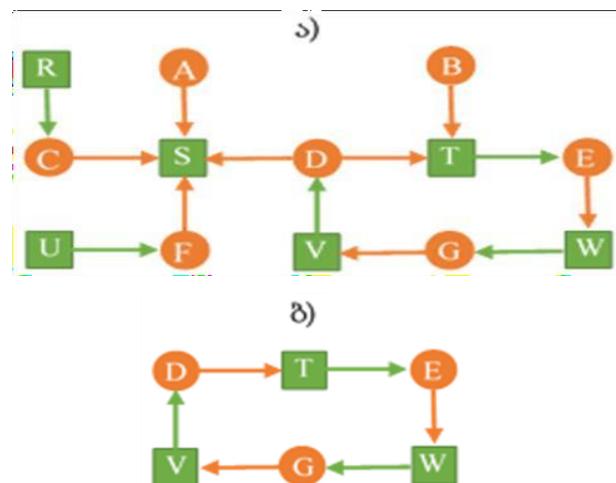
6.4.1. ურთიერთბლოკირების აღმოჩენა

განხილვა დავიწყოთ მარტივი შემთხვევით, როდესაც სისტემაში არსებული ყოველი რესურსი წარმოდგენილია ერთი ასლის სახით. ასეთი სისტემისათვის შესაძლებელია ავაგოთ ნახ. 6.1-ზე გამოსახული გრაფის ანალოგიური გრაფი. თუ გრაფი შეიცავს ერთ ან მეტ ციკლს, მაშინ საქმე გვაქვს ურთიერთბლოკირებასთან. ციკლში მონაწილე ყოველი პროცესი ბლოკირებულია და მათი იქიდან თავის დაღწევა შეუძლებელია. თუ გრაფზე არ აღმოგვაჩნდა ციკლი ეს ნიშნავს არ გვაქვს ურთიერთბლოკირება და პროცესები პრობლემის გარეშე შეძლებენ დასრულებას.

განვიხილოთ მაგალითი. ვთქვათ, მოცემული გვაქვს 7 პროცესი (A - G) და 6 რესურსი (R - W). დავუშვათ რესურსებზე გაკეთებულ მოთხოვნები წარმოდგენილია შემდეგი მიმდევრობის სახით:

1. A პროცესი იკავებს R რესურსს და ითხოვს S რესურსს;
2. B პროცესი არ იკავებს არცერთ რესურსს და ითხოვს T რესურსს;
3. C პროცესი არ იკავებს არცერთ რესურსს და ითხოვს S რესურსს;
4. D პროცესი იკავებს U რესურსს და ითხოვს S და T რესურსს;
5. E პროცესი იკავებს T რესურსს და ითხოვს V რესურსს;
6. F პროცესი იკავებს W რესურსს და ითხოვს S რესურსს;

7. G პროცესი იკავებს V რესურსს და ითხოვს U რესურსს.



ნახ. 6.2. რესურსების გრაფი (ა); რესურსების ციკლი გრაფში (ბ)

ბუნებრივად ისმის კითხვა: იმყოფება თუ არა სისტემა ურთიერთბლოკირების მდგომარეობაში და თუ პასუხი დადებითია, მაშინ რომელი რესურსები მონაწილეობენ ურთიერთბლოკირებაში?

ამ კითხვაზე პასუხს იძლევა ნახ. 6.2. ა)-ზე გამოსახული გრაფი, რომელიც აგებულია მოთხოვნების მიმდევრობის შესაბამისად. როგორც ნახ. 6.2. ა)-დან ვხედავთ, გრაფი შეიცავს ერთ ციკლს. ციკლში მონაწილე პროცესები (D, E, G (ნახ. 6.2 ბ)) იმყოფებიან ურთიერთბლოკირების მდგომარეობაში. A, C, F პროცესები არ მონაწილეობენ ურთიერთბლოკირებაში, ვინაიდან S რესურსი შესაძლებელია (ნებისმიერი მიმდევრობით) გამოყოს ერთერთ მათგანს, რომელიც სამუშაოს დასრულების შემდეგ გამოათავუთლებს მას და სხვა პროცესებსაც შეეძლებათ ამ რესურსის გამოყენება.

მიუხედავად იმისა, რომ გრაფების ცხრილიდან ვიზუალურად ადვილია ურთიერთბლოკირების აღმოჩენა, რეალურ სისტემებში საჭიროა ფორმალური ალგორითმი. განვიხილოთ ბლოკირების აღმოჩენის ყველაზე მარტივი ალგორითმი, რომელიც ამოწმებს რესურსების გრაფს და ციკლის აღმოჩენის შემთხვევაში წყვეტს მუშაობას. თუ გრაფი არ შეიცავს ციკლს, მაშინ ალგორითმი სრულდება გრაფის ბოლომდე შემოწმების შემდეგ. ალგორითმში იგება ერთი დინამიური სტრუქტურა (L), რომელშიც იწერება კვანძების მიმდევრობა და ინიშნება მიმართული მონაკვეთები. ალგორითმის მუშაობის პროცესში მიმართული მონაკვეთის მონიშვნა აღნიშნავს, რომ შესაბამისი გადასვლა შემოწმებულია და ის არ საჭიროებს ხელმეორედ გადამოწმებას.

ალგორითმი შედგება შემდეგი ეტაპებისგან:

1. გრაფზე არსებული ყოველი i-ური კვანძისთვის, რომელსაც გრაფი გამოიყენებს საწყის წერტილად, შესრულდება 2 - 6 ეტაპები;
2. ხორციელდება L ჩამონათვალის ინიციალიზირება და ყველა მიმართულ მონაკვეთს ეხსნება მონიშვნა;
3. მიმდინარე კვანძი L ჩამონათვალს ემატება ბოლოდან და მოწმდება, ხომ არ არის ის ამ ჩამონათვალში. ჩამონათვალში რომელიმე კვანძის ხელმეორედ გამოჩენა ნიშნავს, რომ გრაფი შეიცავს ციკლს და ალგორითმი წყვეტს მუშაობას;
4. მითითებული კვანძისთვის მოწმდება ხომ არ არსებობს კვანძიდან გამომავალი სხვა მიმართული მონაკვეთი. ასეთის არსებობის შემთხვევაში გადავდივართ მე-5 ეტაპზე. წინააღმდეგ შემთხვევაში გადავდივართ მე-6 ეტაპზე;
5. მიმდინარე კვანძიდან მიმართული მონაკვეთის არჩევა ხდება ნებისმიერად. მიმართულ მონაკვეთს ეხსნება მონიშვნა და გადავდივართ ახალ კვანძზე ამ მიმართულებით. ალგორითმი

შესრულებას აგრძელებს მე-3 ეტაპიდან;

6. თუ კვანძი წარმოადგენს საწყის წერტილს ეს ნიშნავს, რომ გრაფი არ შეიცავს ციკლს და ალგორითმი წყვეტს მუშაობას. წინააღმდეგ შემთხვევაში გვაქვს ჩიხი. მიმდინარე კვანძი იშლება და ალგორითმი უბრუნდება იმ კვანძს, რომლიდანაც მიმდინარე კვანძზე მოხდა გადასვლა. შემდეგ ალგორითმი ახალი კვანძისთვის შესრულებას იწყებს მე-3 ეტაპიდან.

ალგორითმი ყოველ კვანძს განიხილავს საწყისი წერტილის როლში, აგებს ხეს და მის შიგნით ეძებს ციკლს. თუ შესრულების მომენტში ალგორითმმა ორჯერ აღმოჩინა რომელიმე კვანძი ეს ნიშნავს, რომ ციკლი აღმოჩენილია და ალგორითმი წყვეტს შესრულებას. ყოველი კვანძისთვის ალგორითმი გადადის ყველა შესაძლო მიმართულებით, რომელსაც უთითებს მიმართული მონაკვეთი (მოთხოვნა). თუ ალგორითმს კვანძიდან რომელიმე მიმართულებით გადასვლა არ შეუძლია (გადასვლა არ ხდება სხვა კვანძზე), მაშინ ის უბრუნდება ერთი დონით მაღლა მყოფ კვანძს. თუ ალგორითმს რომელიმე კვანძთან მიმართებაში შესამოწმებელი არაფერი დარჩა ეს ნიშნავს, რომ კვანძი არ მონაწილეობს ურთიერთბლოკირებაში. თუ ყველა კვანძისთვის ეს პირობა შესრულებულია, მაშინ სისტემა არ იმყოფება ურთიერთბლოკირების მდგომარეობაში.

პრაქტიკული ხასიათის ამოცანაზე განვახორციელოთ ალგორითმის მუშაობის ილუსტრირება. განვიხილოთ ნახ. 6.2. ა)-ზე წარმოდგენილი გრაფი. ალგორითმის მიერ კვანძების განხილვის მიმდევრობა ნებისმიერია. ჩვენს შემთხვევაში კვანძები ავიღოთ შემდეგი მიმდევრობით: R, A, B, C, S, D, T, E და ა.შ. შევნიშნოთ, რომ ციკლის აღმოჩენის შემთხვევაში ალგორითმი წყვეტს მუშაობას.

ალგორითმში კვანძები ავიღოთ მარცხენა ზედა კიდიდან დაწყებული. ალგორითმი პირველ საწყის წერტილად იღებს R კვანძს და ხდება L სტურქტურის ინიციალიზირება ($L=[R]$). რადგანაც R-დან გადასვლა გვაქვს C-ზე, ამიტომ ის აღმოჩნდება L-ში მეორე ელემენტად ($L=[RC]$). C-დან გადასვლა ხდება S-ზე და შესაბამისად ის იქნება L-ში მესამე ელემენტი S ($L=[RCS]$). რადგანაც S-დან არსად არ ხდება გადასვლა, ამიტომ L-ის ფორმირება R კვანძისთვის დასრულებულია და შესაბამისად ვლებულობთ, რომ R ურთიერთბლოკირებაში არ მონაწილეობს. ალგორითმი მეორე საწყის წერტილად ირჩევს C კვანძს. რადგანაც C-დან გვაქვს გადასვლა მხოლოდ S-ზე და იქიდან კი არსად, ამიტომ ამ შემთხვევაში გვაქვს $L=[CS]$, საიდანაც ვასკვნით, რომ C-ც არ მონაწილეობს ურთიერთბლოკირებაში. შემდეგ ნაბიჯზე ალგორითმი საწყის წერტილად ირჩევს B-ს და ანალოგიური მსჯელობით D-მდე მისვლისას გვაქვს $L=[BTEWGVD]$. D-დან გადასვლა შესაძლებელია ორი მიმართულებით (S და T). S-ის მიმართულებით გადასვლის შემთხვევაში, რადგანაც S-დან არცერთი მიმართულებით არ ხდება გადასვლა, ამიტომ ვლებულობთ $L=[BTEWGVDs]$, საიდანაც ვასკვნით, რომ ამ მიმართულებით არ შეიძლება გვქონდეს ურთიერთბლოკირება. D-ს მიმართულებით გადასვლით კი გვაქვს $L=[BTEWGVDt]$. რადგანაც T რესურსი L სტურქტურაში განმეორდა ეს ნიშნავს, რომ ციკლი აღმოჩენილია და ალგორითმი წყვეტს მუშაობას.

იმ შემთხვევაში, როდესაც ოპერაციულ სისტემაში ერთიდაიგივე რესურსი წარმოდგენილია რამდენიმე ასლის სახით გამოიყენება სხვა მიდგომა. ამ შემთხვევაში ი პროცესს შორის (P_1, \dots, P_n) ურთიერთბლოკირების აღმოჩენის ალგორითმში გამოიყენება მატრიცები.

შემოვიღოთ აღნიშვნები: ვთქვათ, თ არის გამოთვლით სისტემაში განსხვავებული რესურსების რაოდენობა და რესურსები გადანომრილია 1-დან m -მდე. E_j -ით ($1 \leq j \leq m$) აღვნიშნოთ j -ური რესურსის ასლების რაოდენობა. E -თი აღვნიშნოთ გამოთვლითი სისტემის რესურსების საერთო რაოდენობა (ანუ ვექტორი $E = (E_1, \dots, E_n)$). A-თი აღვნიშნოთ დროის მიმდინარე მომენტში რესურსების თავისუფალი (ხელმისაწვდომი) ასლების რაოდენობის აღმიშვნელი ვექტორი $A = (A_1, \dots, A_n)$, სადაც A_j -ით ($1 \leq j \leq m$) აღნიშვნულია j -ური თავისუფალი რესურსის რაოდენობა.

დამატებით შემოვიღოთ კიდევ ორი მატრიცა R და C , სადაც R მატრიცის $R_i (1 \leq i \leq n)$ სტრიქონი (ვექტორი) აღნიშნავს P_i პროცესის მიერ დამატებით მოთხოვნილი რესურსების რაოდენობას, ხოლო C მატრიცის $C_i (1 \leq i \leq n)$ სტრიქონი (ვექტორი) კი აღნიშნავს P_i პროცესისთვის დროის მიმდინარე მომენტში გამოყოფილი რესურსების საერთო რაოდენობას.

$$P = (P_1, \dots, P_n) \quad A = (A_1, \dots, A_n) \quad E = (E_1, \dots, E_n)$$

$$R = \begin{pmatrix} R_{11} & \cdots & R_{1m} \\ \vdots & \cdots & \vdots \\ R_{n1} & \cdots & R_{nm} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \cdots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{pmatrix}$$

ცხადია, რომ რესურსებისთვის გვექნება შემდეგი დამოკიდებულება

$$E_i = A_i + \sum_{j=0}^m C_{ij}, (1 \leq i \leq n).$$

ნებისმიერ ორ $X = (X_1, \dots, X_m)$ და $Y = (Y_1, \dots, Y_m)$ ვექტორს შორის შედარების ოპერაცია განვსაზღვროთ შემდეგნაირად:

$$X \leq Y, \text{ მაშინ და მხოლოდ მაშინ, როდესაც } X_j \leq Y_j \text{ ყოველი } j\text{-სთვის } (1 \leq j \leq m).$$

ალგორითმის მუშაობის დაწყებამდე ყველა პროცესი ცხადდება არამარკირებულად. შესრულების პროცესში მოხდება პროცესების მარკირება იმის აღსანიშნავად, რომ ისინი არ მონაწილეობენ ურთიერთბლოკირებაში და შეუძლიათ დასრულება. ალგორითმის დასრულების შემდეგ არამარკირებული პროცესის არსებობა ნიშნავს, რომ ის მონაწილეობს ურთიერთბლოკირებაში. ალგორითმის გამოყენებისას ვითვალისწინებთ ყველაზე უარეს შემთხვევას, რომლის დროსაც პროცესი იკავებს ყველა რესურსს დასრულებამდე.

ურთიერთბლოკირების ალგორითმის მუშაობა იყოფა სამ ეტაპად:

1. ხორციელდება არამარკირებული პროცესის (P_i) ძებნა, რომლისთვისაც შესრულებულია პირობა $R_i \leq A_i$, ანუ დასასრულებლად საჭირო რესურსების რაოდენობა არ აღემატება თავისუფალი რესურსების რაოდენობას;
2. თუ ასეთი პროცესი მოიძებნა A ვექტორით იცვლება C მატრიცის შესაბამისი სტრიქონი და ალგორითმი უბრუნდება ეტაპ 1-ს;
3. თუ ასეთი პროცესი არ არსებობს, მაშინ ალგორითმი ასრულებს მუშაობას.

საზოგადოდ, ალგორითმის მუშაობის პრინციპი მდგომარეობს შემდეგში: პირველ ეტაპზე ალგორითმი ეძებს პროცესებს, რომელთა დაკავშირებით შესაძლებელია სისტემაში არსებული თავისუფალი რესურსების ხარჯზე. შემდეგ პროცესს გამოეყოფა რესურსები და ის იწყებს შესრულებას. პროცესი დასრულების შემდეგ ათავისუფლებს მის მიერ დაკავებულ ყველა რესურსს და მათი გამოყენება შეუძლია სისტემაში არსებულ სხვა პროცესებს. ხდება პროცესის მარკირება. თუ ალგორითმის დასრულების შემდეგ სისტემაში აღმოჩნდება მხოლოდ მარკირებული პროცესები ანუ ყველა პროცესს შეუძლია დასრულება, ეს ნიშნავს, რომ სისტემაში ურთიერთბლოკირების წარმოშობის საფრთხე არ არსებობს. წინააღმდეგ შემთხვევაში დროის ნებისმიერ მომენტში შესაძლებელია წარმოიშვას ურთიერთბლოკირება.

ალგორითმის მუშაობის საილუსტრაციოდ განვიხილოთ მაგალითი. ვთქვათ, მოცემული გვაქვს სამი პროცესი, P_1, P_2, P_3 და 4 რესურსი, X_1, X_2, X_3, X_4 (ნახ. 6.3). რესურსი შეიძლება იყოს ნებისმიერი ჩამოთვლილთაგან: პროცესორი, მყარი დისკი, ფაილები, მეხსიერება და ა.შ.

$X_1 \quad X_2 \quad X_3 \quad X_4$	
$E = (4 \quad 2 \quad 3 \quad 1)$	- სისტემაში არსებული რესურსების საერთო რაოდენობა
$X_1 \quad X_2 \quad X_3 \quad X_4$	
$A = (2 \quad 1 \quad 0 \quad 0)$	- სისტემაში მიმდინარე გომინაციას და მის რესურსების საერთო რაოდენობა
რესურსების მიმდინარე განაწილების გატრიცა	მოთხოვნების გატრიცა
$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$	$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$

ნახ. 6.3. ურთიერთბლოკირების აღმოჩენის ალგორითმის გამოყენების მაგალითი

ალგორითმის შესრულების მომენტში პირველი ხორციელდება იმ პროცესების მოძიება, რომელთა მიერ გაკეთებული მოთხოვნების დაკმაყოფილება არსებული თავისუფალი რესურსების ხარჯზე შესაძლებელია. ალგორითმს არ შეუძლია P_1, P_2 პროცესის დაკმაყოფილება, ვინაიდან მიმდინარე მომენტში დაკავებულია, შესაბამისად, X_3, X_4 რესურსი. მაგრამ შესაძლებელია P_3 პროცესის დაკმაყოფილება. სისტემა რესურსებს გამოუყოფს სწორედ P_3 პროცესს და ის დაიწყებს შესრულებას. P_3 პროცესის დასრულების შემდეგ სისტემაში გვექნება შემდეგი თავისუფალი რესურსები $A = (2, 2, 2, 0)$.

რესურსების ასეთი ნაკრებით უკვე შესაძლებელია P_2 პროცესის მომსახურება. სისტემა დააკმაყოფილებს მის მოთხოვნას და მისი დასრულების შემდეგ გვექნება შემდეგი თავისუფალი რესურსები $A = (4, 2, 2, 1)$, რომელიც საკმარისია P_1 პროცესის მოთხოვნის დასაკმაყოფილებლად. რადგანაც ყველა პროცესს შეუძლია დასრულება შეგვიძლია დავასკვნათ, რომ სისტემაში არ არსებობს ურთიერთბლოკირების წარმოშობის საფრთხე.

განხილულ მაგალითში რესურსების მოთხოვნების გარკვეული შეცვლით შეიძლება მივიღოთ სიტუაცია, რომლის დროსაც პროცესები აღმოჩნდებიან ურთიერთბლოკირების მდგომარეობაში. მაგალითად, P_3 პროცესის მიერ გაკეთებულ მოთხოვნას $A = (2, 1, 0, 0)$ თუ შევცვლით მოთხოვნით $A = (2, 1, 0, 1)$ ადვილი შესამოწმებელია, რომ ამ შემთხვევაში სისტემაში წარმოიშობა ურთიერთბლოკირების მდგომარეობა.

6.4.2. ურთიერთბლოკირებიდან თავის დაღწევა

დავუშვათ ალგორითმა შესრულების შემდეგ აღმოჩინა, რომ პროცესები იმყოფებიან ურთიერთბლოკირების მდგომარეობაში. როგორ უნდა დავაღწიოთ თავი ამ მდგომარეობას? ყოველ ოპერაციულ სისტემაში არსებობს ურთიერთბლოკირების მდგომარეობიდან გამოსვლის და მის შემდეგ აღდგენის გარკვეული მექანიზმი.

ზოგჯერ შესაძლებელია პროცესს დროებით ჩამოერთვას გარკვეული რესურსი და ის გადაეცეს სხვა პროცესს, რომელიც საკუთარი სამუშაოს დასრულების შემდეგ რესურსს დაუბრუნებს მის წინა მფლობელს. პაკეტური დამუშავების სისტემებში უმეტეს შემთხვევაში ამისთვის საჭიროა ოპერატორის ჩარევა.

პროცესისთვის გარკვეული რესურსის ჩამორთმევა მისი ინფორმირების გარეშე და სხვა

პროცესისთვის გადაცემა (რომელიც საკუთარი შესრულების შემდეგ დაუბრუნებს მას) დამოკიდებულია რესურსის ტიპზე. ამ მეთოდით აღდგენა გარკვეულ სირთულეებთანაა დაკავშირებული ან საერთოდ შეუძლებელია. პროცესის არჩევა, რომელსაც შეიძლება ჩამორთვას გარკვეული რესურსი, დამოკიდებულია იმაზე, თუ რამდენად „უმტკივნეულო“ იქნება მისთვის შესაბამისი რესურსის ჩამორთმევა.

აღდგენის შემდეგი მეთოდი შესაძლებელია ჩამოვაყალიბოთ შემდეგნაირად: თუ ოპერაციული სისტემის შემქმნელს ან გამოთვლითი სისტემის ოპერატორს წინასწარ ეცოდინება ურთიერთბლოკირების წარმოშობის ალბათობა, მაშინ მას შეეძლება საკონტროლო წერტილების ორგანიზება. რაც ნიშნავს, რომ პროცესის მდგომარეობა იწერება სპეციალურ ფაილში, რომელიც იძლევა გარკვეული ადგილიდან პროცესის ხელახალი შესრულების (აღდგენის) შესაძლებლობას. საკონტროლო წერტილები შეიცავენ არამხოლოდ მეხსიერების ასლს, არამედ შეიცავენ ინფორმაციას რესურსების მდგომარეობაზე. მაღალი ეფექტურობის მისაღწევად არ უნდა ხდებოდეს ახალი საკონტროლო წერტილების გადაწერა უკვე არსებულზე, არამედ ისინი უნდა ინახებოდნენ ცალკეულ ფაილებად. შესაბამისად, ასეთ სისტემაში ყოველი პროცესისთვის იარსებებს საკონტროლო წერტილების გარკვეული მიმდევრობა. ურთიერთბლოკირების აღმოჩენის შემთხვევაში ძნელი არაა იმის გარკვევა, თუ რომელმა რესურსმა გამოიწვია ის. ურთიერთბლოკირების მდგომარეობიდან გამოსვლის მიზნით პროცესს უწევს იმ საკონტროლო წერტილზე დაბრუნება, რომელიც წინმსწრებია შესაბამისი რესურსის გამოყოფისა. ასეთ შემთხვევაში, აღნიშნული საკონტროლო წერტილის შემდეგ პროცესის მიერ შესრულებული სამუშაო მთლიანად იკარგება და ამ გზის გავლა პროცესს უწევს თავიდან. პროცესი, მისთვის გარკვეული რესურსის ჩამორთმევის შემდეგ, ბრუნდება შესაბამის საკონტროლო წერტილში, ჩამორთმეული რესურსი გამოეყოფა სხვა პროცესს, ხოლო მიმდინარე პროცესს კი ჩამორთმეული რესურსის მისაღებად უწევს ლოდინი.

კიდევ ერთი მეთოდი, რომელიც გამოიყენება ურთიერთბლოკირების მდგომარეობიდან გამოსასვლელად მდგომარეობს ერთი პროცესის ან პროცესთა მთელი ჯგუფის განადგურებაში. განადგურებული შეიძლება იქნას როგორც ციკლში მონაწილე, ისე მის გარეთ მყოფი ნებისმიერი პროცესი. ამ მეთოდის გამოყენების შემთხვევაში სასურველია ისეთი პროცესების შერჩევა, რომელთა შესრულების შეწყვეტა სისტემისთვის იქნება „უმტკივნეულო“, ხოლო მის მიერ გამოთავისუფლებული რესურსები კი საკმარისი სხვა პროცესების დასასრულებლად.

6.5. პრობლემის დინამიური აცილება

ურთიერთბლოკირების აღმოჩენის მიზნით ჩვენ გავაკეთეთ დაშვება, რომ პროცესი თავიდანვე ითხოვს მის შესასრულებლად საჭირო ყველა რესურს და იღებს მათ. მაგრამ უმეტეს ოპერაციულ სისტემებში პროცესების მხრიდან რესურსების მოთხოვნა ხორციელდება თითოთითოდ. სისტემას უნდა შეეძლოს იმის განსაზღვრა შეიძლება თუ არა რესურსის გამოყოფა დაკავშირებული იყოს გარკვეულ საფრთხესთან. თუ რესურსის გამოყოფა არ შეიცავს არანაირ საფრთხეს, მაშინ მისი გამოყოფა შესაძლებელია. წინააღმდეგ შემთხვევაში პროცესს არ გამოეყოფა რესურსი. ამასთან დაკავშირებით ისმის კითხვა: არსებობს თუ არა რაიმე მეთოდი რომლის მეშვეობითაც სისტემას წინასწარ შეუძლია განსაზღვროს ურთიერთბლოკირების წარმოშობის ალბათობა და აიცილოს ის? ამ კითხვაზე პასუხი დადებითია, მაგრამ ურთიერთბლოკირების აცილება შესაძლებელია მხოლოდ წინასწარ ცნობილი გარკვეული ინფორმაციის საფუძველზე.

ურთიერთბლოკირების აცილების ალგორითმი ეყრდნობა უსაფრთხო (საიმედო)

მდგომარეობის კონცეფციას. დროის ნებისმიერ მომენტში სისტემაში არსებული რესურსები იმყოფება გარკვეულ მდგომარეობაში, რასაც გამოვსახავთ A , C , R მატრიცებით.

რესურსების მდგომარეობას ეწოდება **საიმედო**, თუ არსებობს რესურსების გადანაწილების ერთი მაინც ისეთი შესაძლებლობა, რომლის დროსაც ყველა პროცესს შეეძლება დასრულება, მიუხედავად იმისა თუ როგორია რესურსებზე გაკეთებული მოთხოვნები. წინააღმდეგ შემთხვევაში რესურსების მდგომარეობას ეწოდება **არასაიმედო**.

საიმედო მდგომარეობის ილუსტრირება განვიხილოთ ერთი რესურსის შემთხვევაში. ნახ. 6.4-ზე გამოსახულია სიტუაცია, რომლის დროსაც რესურსის საერთო რაოდენობა 10-ის ტოლია. აქედან A პროცესი იკავებს 3 რესურს და ჯამში საჭიროებს 9 რესურს, B იკავებს 2-ს და ჯამში საჭიროებს - 4-ს, ხოლო C იკავებს 2-ს და ჯამში საჭიროებს 7-ს. რესურსების საერთო რაოდენობიდან პროცესების მიერ დაკავებულია 7 რესურსი და თავისუფალია 3 რესურსი.

I	II										
A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	—	—	B	—	—
C	2	7	C	2	7	C	2	7	C	7	7

თავისუფალი: 3 თავისუფალი: 1 თავისუფალი: 5 თავისუფალი: 0 თავისუფალი:

I. გამოყოფილია II. საჭიროებს

ნახ. 6.4. საიმედო მდგომარეობის მაგალითი

ნახ. 6.4-ზე გამოსახული სიტუაცია არის საიმედო, ვინაიდან მასში რესურსების განაწილების ორგანიზება შესაძლებელია ისეთნაირად, რომ ყველა პროცესმა შეძლოს დასრულება. კერძოდ, მიმდინარე მომენტისთვის თავისუფალი არის სამი რესურსი, ხოლო B პროცესი კი დასრულებისთვის საჭიროებს 2 რესურსს. სისტემა პირველი B პროცესს გამოუყოფს საჭირო რაოდენობის რესურსს, რომელიც დასრულების შემდეგ გამოათავისუფლებს მის მიერ დაკავებულ ყველა რესურსს. B პროცესის დასრულების შემდეგ სისტემაში თავისუფალი რესურსების რაოდენობა იქნება 5, რომელიც საკმარისია C პროცესის დასრულებისთვის. C პროცესის დასრულების შემდეგ თავისუფალი რესურსების რაოდენობა იქნება 7, რომელიც რათქმაუნდა საკმარისია A პროცესის დასრულებისთვის.

თუ განხილულ ამოცანაში თავიდან არსებული თავისუფალი რესურსების განაწილება მოხდებოდა შემდეგნაირად: 2 რესურსი B პროცესს და ერთი რესურსი A პროცესს, მაშინ B პროცესის დასრულების შემდეგ სისტემაში თავისუფალი რესურსების რაოდენობა იქნებოდა 4, რომელიც არ აღმოჩნდებოდა საკმარისი არც A და არც C პროცესის დასასრულებლად და მივიღებდით არასაიმედო მდგომარეობას (ანუ წარმოიქმნებოდა ურთიერთბლოკირება).

6.5.1. ბანკირის ალგორითმი

დეისტრის მიერ შემუშავებული იყო ალგორითმი, რომელიც ეყრდნობოდა ბანკირის მოქმედებებს კლიენტისთვის კრედიტის გამოყოფისას და იძლეოდა ურთიერთბლოკირების აცილების შესაძლებლობას. ეს ალგორითმი ცნობილია **ბანკირის ალგორითმის** სახელით და პროცესებისთვის რესურსების გამოყოფა ხორციელდება იმის მიხედვით, რესურსის გამოყოფის შემდეგ იქნება თუ არა შენარჩუნებული საიმედო მდგომარეობა.

ბანკირის ალგორითმის მაგალითად, ერთი რესურსის შემთხვევაში, შევვიძლია განვიხილოთ ნახ. 6.4-ზე წარმოდგენილი მაგალითი. ალგორითმი ყოველ შემოსულ მოთხოვნას განიხილავს შემოსვლის თანმიმდევრობით და ამოწმებს მოთხოვნის დაკმაყოფილებას ხომ არ მივყავართ

არასაიმედო მდგომარეობამდე. თუ რესურსის გამოყოფა არ ცვლის საიმედო მდგომარეობას, მაშინ პროცესისთვის ხდება საჭირო რესურსის გამოყოფა. წინააღმდეგ შემთხვევაში შემოსული მოთხოვნა იქნება გადადებული გვერდზე სანამ მისი დაკმაყოფილება არ მიგვიყვანს საიმედო მდგომარეობამდე ანუ სისტემაში არ გაჩნდება მის დასასრულებლად საკმარისი რესურსი.

ბანკირის ალგორითმის განზოგადება შესაძლებელია ორი ან მეტი რაოდენობის რესურსის შემთხვევაზეც. ნახ. 6.5-ზე გამოსახულია ორი მატრიცა: ა) გამოსახავს 5 პროცესს მათ მიერ დაკავებული რესურსებით და ბ) კი გამოსახავს თითოეული მათგანის დასასრულებლად დამატებით საჭირო რესურსების რაოდენობას. ერთი რესურსის შემთხვევის მსგავსად ამ შემთხვევაშიც პროცესებმა უნდა გააკეთონ მოთხოვნები, რათა სისტემამ შეძლოს გამოსაყოფი რესურსების მართვა. ნახ. 6.5-ზე, ასევე, გამოსახულია სისტემაში არსებული რესურსების საერთო რაოდენობა (E), დაკავებული რესურსები (B), და თავისუფალი რესურსები (A).

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>P₁</td><td>3</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>P₂</td><td>0</td><td>1</td><td>0</td><td>0</td><td></td></tr> <tr><td>P₃</td><td>1</td><td>1</td><td>1</td><td>0</td><td></td></tr> <tr><td>P₄</td><td>1</td><td>1</td><td>0</td><td>1</td><td></td></tr> <tr><td>P₅</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td></tr> </table> <p>შიმდინარე განაწილება</p>	P ₁	3	0	1	1	1	P ₂	0	1	0	0		P ₃	1	1	1	0		P ₄	1	1	0	1		P ₅	0	0	0	0		<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td>P₁</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>P₂</td><td>0</td><td>1</td><td>1</td><td>2</td></tr> <tr><td>P₃</td><td>3</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>P₄</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>P₅</td><td>2</td><td>1</td><td>1</td><td>0</td></tr> </table> <p>დაპატებით საჭირო რესურსები</p>	P ₁	1	1	0	0	P ₂	0	1	1	2	P ₃	3	1	0	0	P ₄	0	0	1	0	P ₅	2	1	1	0
P ₁	3	0	1	1	1																																																			
P ₂	0	1	0	0																																																				
P ₃	1	1	1	0																																																				
P ₄	1	1	0	1																																																				
P ₅	0	0	0	0																																																				
P ₁	1	1	0	0																																																				
P ₂	0	1	1	2																																																				
P ₃	3	1	0	0																																																				
P ₄	0	0	1	0																																																				
P ₅	2	1	1	0																																																				
ა)	ბ)																																																							

$E = (6342)$
 $B = (5322)$
 $A = (1020)$

ნახ. 6.5. ბანკირის ალგორითმი რამდენიმე რესურსის შემთხვევაში

ამ შემთხვევაში ალგორითმის მუშაობა შეიძლება აღვწეროთ შემეგ ეტაპებად:

1. R მატრიცაში ვეძევთ სტრიქონს, რომლის დაკმაყოფილებაც შესაძლებელია A ვექტორით. თუ ასეთი სტრიქონი არ აღმოჩნდა (სისტემაში არა საკმარისი თავისუფალი რესურსი), მაშინ სისტემა იმყოფება ურთიერთბლოკირების მდგომარეობაში, ვინაიდან შეუძლებელია რომელიმე პროცესის მიერ გაკეთებული მოთხოვნის დაკმაყოფილება;
2. ვუშვებთ, რომ პროცესი რომლის შესაბამის სტრიქონიც იქნა არჩეული ითხოვს ყველა რესურსს და ასრულებს საკუთარ სამუშაოს. სამუშაოს დასრულების შემდეგ ის აბრუნებს მის მიერ დაკავებულ რესურსებს, რომლებიც აისახება A ვექტორში;
3. 1 და 2 ეტაპი განმეორდება სანამ ყველა პროცესი არ დასრულდება (საიმედო მდგომარეობა), ან დაგვრჩება პროცესები, რომელთა მოთხოვნის დაკმაყოფილება შეუძლებელია (არასაიმედო მდგომარეობა);

შევნიშნოთ, რომ ბანკირის ალგორითმის გამოყენებისას თუ შესაძლებელია რამდენიმე პროცესის მიერ გაკეთებული მოთხოვნის ერთდროული დაკმაყოფილება, მაშინ მიმდევრობას რომლითაც მოხდება მოთხოვნების დამუშავება არანაირი მნიშვნელობა არ გააჩნია.

6.6. ურთიერთბლოკირების აცილება

როგორ შეიძლება ავიცილოთ ურთიერთბლოკირება რეალურ სისტემებში? როგორც უკვე დავრწმუნდით საზოგადოდ ურთიერთბლოკირების აცილება შეუძლებელია, ვინაიდან ის წინასწარ მოითხოვს მომავალში გასაკეთებელ მოთხოვნებზე ინფორმაციის ქონას. ასეთი ინფორმაციის მიწოდება კი ყოველთვის შეუძლებელია. ამ კითხვაზე პასუხის გასაცემად საკმარისია ურთიერთბლოკირების წარმოშობის აუცილებელი და საკმარისი პირობებიდან ერთერთის დარღვევა.

6.6.1. ურთიერთგამორიცხვის პირობის დარღვევა

საზოგადოდ ურთიერთგამორიცხვის აცილება შეუძლებელია. გარკვეულ რესურსებზე დაშვება უნდა იყოს განსაკუთრებული. მიუხედავად ამისა შესაძლებელია ზოგიერთი რესურსის საერთო სარგებლობის რესურსად გადაქცევა. მაგალითის სახით განვიხილოთ პრინტერი. ცხადია, რომ პრინტერზე შედეგების გამოტანა შეუძლია რამდენიმე პროცესს. ქაოსის აცილების მიზნით ხორციელდება ყველა გამოსატანი მონაცემის დროებითი განთავსება დისკზე, ანუ განაწილებად მოწყობილობაზე. მხოლოდ ერთი სისტემური პროცესი, ე.წ. პრინტერის დემონი, რომელიც პასუხისმგებელია პრინტერის გამონთავისუფლების შესაბამისად დოკუმენტის დასაბეჭდად გამოტანაზე, რეალურად ურთიერთქმედებს მასთან.

სამწუხაროდ ყველა მოწყობილობის და მონაცემების გადაქცევა საზიარო რესურსად შეუძლებელია.

6.6.2. რესურსების ლოდინის პირობის დარღვევა

რესურსების ლოდინის პირობის დარღვევა შესაძლებელია რესურსების მოთხოვნის ორეტაპიანი სტრატეგიის შესრულებით.

- პირველ ეტაპზე პროცესი ითხოვს მის შესასრულებლად საჭირო ყველა რესურსს. სანამ არ მოხდება რესურსების სრულად გამოყოფა პროცესი არ იწყებს შესრულებას;
- თუ მიმდინარე პროცესისთვის პირველი ეტაპით მოთხოვნილი რესურსები დაკავებული იყო სხვა პროცესების მიერ, მაშინ ის ათავისუფლებს მისთვის გამოყოფილ რესურსებს და იმეორებს პირველ ეტაპს.

ამრიგად, ყველა რესურსის დაკავების ერთერთ მეთოდს წარმოადგენს ყველა პროცესს მოვთხოვოთ მათი შესრულებისთვის საჭირო რესურსების მოთხოვნა შესრულების დაწყებამდე. თუ სისტემას შეუძლია პროცესს გამოუყოს მისთვის საჭირო ყველა რესურსი, მას შეუძლია იმუშაოს დასრულებამდე. თუ რესურსებიდან დაკავებულია ერთი მაინც პროცესი დაელოდება მის გამოთავისუფლებას.

მოცემული გადაწყვეტა გამოიყენება პაკეტური დამუშავების სისტემებში, რომელიც მომხმარებლისგან ითხოვს პროგრამისათვის საჭირო ყველა რესურსის მითითებას. როგორც უკვე აღინიშნა, მომავალში საჭირო რესურსების ჩამონათვალის პროგნოზირება ხშირად შეუძლებელია. თუ ასეთი ინფორმაცია არსებობს, მაშინ შესაძლებელია ბანკირის ალგორითმის გამოყენება.

6.6.3. გაუნაწილებლობის პირობის დარღვევა

თუ პროცესებს შეეძლებოდათ ერთიმეორისთვის რესურსების წართმევა და მათი შენარჩუნება დასრულებამდე, მაშინ შესაძლებელი იქნებოდა ურთიერთბლოკირების წარმოქმნის მესამე პირობის დარღვევა. ჩამოვაყალიბოთ მოყვანილი მიდგომის ნაკლოვანება:

- პროცესებისთვის შეიძლება იმ რესურსების ჩამორთმევა, რომელთა მდგომარეობის შენახვა, ხოლო მოგვიანებით მისი აღდგენა მიმდინარე მდგომარეობამდე ადვილია;
- თუ პროცესი გარკვეული პერიოდის განმავლობაში იყენებს განსაზღვრული რაოდენობის რესურსებს და მოხდება ამ რესურსების ან მისი ნაწილის ჩამორთმევა, მაშინ შესაძლებელია დაიკარგოს პროცესის მიერ შესრულებული სამუშაოს მნიშვნელოვანი ნაწილი;
- მოცემული სქემის შედეგი შეიძლება იყოს ზოგიერთი პროცესების დისკრიმინაცია, რომელთაც ყოველთვის ართმევენ რესურსებს.

6.6.4. წრიული ლოდინის პირობის დარღვევა

მოთხოვნათა დაკმაყოფილების ერთერთ მეთოდს წარმოადგენს - რესურსების დალაგება. მაგალითად, შესაძლებელია ყველა რესურსისათვის უნიკალური ნომრის მინიჭება და მოთხოვნა, რომ პროცესებმა რესურსები მოითხოვონ მათი რიგითი ნომრის ზრდის მიხედვით. მაშინ წრიული ლოდინი არ შეიძლება წარმოიშვას. ბოლო მოთხოვნის და ყველა რესურსის გათავისუფლების შემდეგ პროცესს შეიძლება ნება დაერთოს პირველი მოთხოვნის განხორციელებაზე. ცხადია, რომ პრაქტიკულად შეუძლებელია ისეთი მიმდევრობის მოძებნა, რომელიც დააკმაყოფილებს ყველას.

ლექცია 7. ნაკადები (thread)

7.7. ნაკადები (thread)

პროცესის ცნების შემოღებამ შესაძლებელი გახადა პროგრამის ცნების გაყოფა და ამით ოპერაციულმა სისტემამ შეიძინა მრავალპროგრამულობის შესაძლობლობა. ბუნებრივად ისმის კითხვა: შესაძლებელია თუ არა პროცესის ცნების კიდევ უფრო გაყოფა და რა სახის უპირატესობა შეიძლება მოგვცეს ამან? რათქმა უნდა შესაძლებელია პროცესის ცნების გაყოფა და შედეგად მიღებული „მინიპროცესის“, რომელსაც ნაკადი ეწოდება, გამოყენებით შესაძლებელია მთელი რიგი უპირატესობების მიღება. ნაკადების გამოყენების ძირითადი მიზეზი მდგომარეობს მრავალ პროგრამაში სხვადასხვა მოქმედებების პარალელური განხორციელების შესაძლებლობაში, რომელთა ნაწილიც პერიოდულად შეიძლება იყოს ბლოკირებული. პროგრამირების მოდელი მარტივდება მსგავსი პროგრამების გაყოფით პარალელურ რეჟიმში შესრულებადი thread-ის სახით.

პროცესების განხილვის დროს ჩვენ შევხვდით არაერთ არგუმენტს, რომლითაც აიხსნე- ბოდა პროცესების მოდელის გამოყენების უპირატესობა. კერძოდ, ნაცვლად იმისა, რომ გვეფიქრა წყვეტაზე, ტაიმერებზე და კონტექსტის (პროცესების) გადამრთველზე, უმჯობესი იყო გვეფიქრა პარალელურ პროცესებზე. ახალი ელემენტის - thread-ის, დამატებით ჩვენ უკვე შევიძლია ვიფიქროთ პროცესის შიგნით შესრულებადი thread-ის მიერ პროცესისთვის გამოყოფილი მეხსიერების სიცრცის და რესურსების ერთობლივ გამოყენებაზე. ეს შესაძლებლობა განსაკუთრებით მნიშვნელოვანია იმ პროგრამებისთვის, რომლებიც საკუთარი შესრულებისას იყენებენ მხოლოდ ერთ პროცესს.

thread-ის გამოყენების მეორე მიზეზს წარმოადგენს მისი წარმოქმნის და დასრულების სისწრაფე, „მძიმეწონიან“ პროცესთან შედარებით. მრავალ სისტემაში thread-ის წარმოქმნა ხორციელდება 10 – 100 -ჯერ უფრო სწრაფად, ვიდრე პროცესის წარმოქმნა. ეს თვისება განსაკუთრებით სასარგებლოა, როდესაც საჭიროა thread-ების რაოდენობის სწრაფი და დინამიური შეცვლა.

thread-ების გამოყენების მესამე მიზეზი მდგომარეობს სისტემის წარმადობის ამაღლებაში. ერთპროცესორულ სისტემაში პროცესების მუშაობას არ მივყავართ წარმადობის დაჩქარებამდე. ასეთ სისტემებში მნიშვნელოვანი ამოცანის გადაწყვეტისას (რომელიც საჭიროებს შეტანა/გამოტანის ოპერაციებს) შეტანა/გამოტანის ოპერაციის დასრულებამდე პროცესორს უწევს უქმად ყოფნა (ცარიელი ციკლის შესრულება). thread-ების გამოყენებით შესაძლებელია განსახორციელებელი მოქმედებების ორგანიზება დროითი გადაფარვით (თუ რომელიმე სრულდება აქტიურ რეჟიმში, მაშინ მეორე ხორციელდება ფონურ რეჟიმში), რაც გამოიწვევს პროგრამის დაჩქარებას.

thread-ები განსაკუთრებით სასარგებლოა მრავალპროცესორულ სისტემებში, სადაც რეალურად შესაძლებელია პარალელური გამოთვლების განხორციელება.

იმისთვის, რომ გავერკვეთ რაში გამოიხატება thread-ების გამოყენებით მიღებული სარგებელი განვიხილოთ კონკრეტული მაგალითი. განვიხილოთ ტექსტური პროცესორი მაგალითად, Microsoft word). ამ პროგრამას დასახეჭდად წარმოდგენილი დოკუმენტი მონიტორის ეკრანზე გამოაქვს იმ ფორმით რა ფორმითაც შემდგომ ის აღმოჩნდება ფურცელზე. კერძოდ, მონიტორზე ყველა ფურცლის გვერდის ბოლო და ყველა სტრიქონის ბოლო მდებარეობს იმ ადგილას, სადაც ისინი ბეჭდვის შემდეგ აღმოჩნდებიან ფურცლებზე, რათა აუცილებლობის შემთხვევაში მომხმარებელმა ადვილად წაიკითხოს ის ან შეცდომის შემთხვევაში განახორციელოს შესაბამისი ცვლილება.

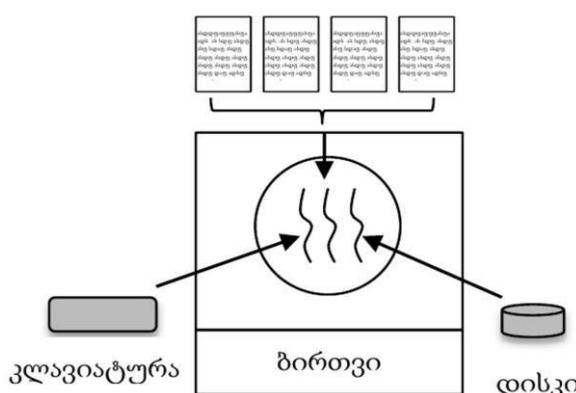
ახლა დავუშვათ, მომხმარებელი დაკავებულია წიგნის წერით. ერთის მხრივ, უმჯობესია მთლიანი წიგნის ქონა ერთი ფაილის სახით, რომელშიც მებნის, გლობალური ცვლილების

განხორციელების ან სხვა საჭირო მოქმედებების შესრულება ადვილია. მეორეს მხრივ, მთლიანი წიგნი შესაძლებელია წარმოდგენილი იყოს თავების მიხედვით სხვადასხვა ფაილის სახით. თუ თითოეული თავი და მასში შემავალი პარაგრაფები წარმოდგენილი იქნებოდა ცალკეული ფაილის სახით, მაშინ მთლიან წიგნთან მუშაობა იქნებოდა მოუხერხებელი. გლობალური ცვლილების განხორციელებისას საჭირო გახდებოდა თითოეული ფაილის ცალცალკე რედაქტირება.

ახლა დავუშვათ, მომხმარებელმა გადაწყვიტა 800 გვერდიან დოკუმენტში ცვლილების შეტანა. კერძოდ, პირველი გვერდიდან ამოშალოს გარკვეული ფრაგმენტი და, მაგალითად, 601-ე გვერდზე განახორციელოს ცვლილება. პირველ გვერდზე შესაბამისი ფრაგმენტის წამლის შემდეგ 601-ე გვერდზე გადასვლის ბრძანების შესრულებამდე ტექსტურმა პროცესორმა უნდა განახორციელოს მთლიანი ტექსტის ფორმატირება, ვინაიდან წინასწარ მისთვის უცნობია, თუ რითი იწყება აღნიშნული გვერდი. წინა გვერდების (1-600) დამუშავების გარეშე კი შეუძლებელი იქნებოდა ამის გაკეთება. გვერდების დამუშავებას და საჭირო გვერდის მონიტორის ეკრაზე გამოტანას შესაძლებელია დასჭირდეს გარკვეული დრო, რაც გამოიწვევს მომხმარებლის უკმაყოფილებას.

აქ სიტუაციის გამარტივება შესაძლებელია thread-ების გამოყენებით. დავუშვათ ტექსტური პროცესორი შექმნილია როგორც ორნაკადიანი პროგრამა, რომელთაგან ერთი ურთიერთქმედებს მომხმარებელთან, მეორე კი დაკავებულია ფონურ რეჟიმში ტექსტის ფორმატირებით. ამ შემთხვევაში მომხმარებელთან ურთიერთქმედების thread-ის მიერ როგორც კი დოკუმენტის პირველი გვერდიდან წაიშლება შესაბამისი ფრაგმენტი, ფონურ რეჟიმში მუშაობას დაიწყებს ტექსტის ფორმატირების thread-ი. სანამ მომხმარებელთან ურთიერთქმედების thread-ი გააკონტროლებს კლავიატურის მიერ განხორციელებულ მოქმედებებს (ხომ არ მოხდა რაიმე სიმბოლოს შეტანა) ფორმატირების thread-მა შესაძლებელია დაასრულოს მთლიანი დოკუმენტის დამუშავება და მომხმარებლის მოთხოვნისთანავე გამოიტანოს საჭირო გვერდი.

ბუნებრივია ისმის კითხვა: თუ პროგრამა დაიყო ორ thread-ად ხომ არ შეიძლება მისთვის ახალი thread-ის დამატება? მრავალი ტექსტური პროცესორი ავტომატურ რეჟიმში პერიოდულად (5-10 წუთი) დისკზე ინახავს დოკუმენტში განხორციელებულ ცვლილებებს იმ მიზნით, რომ არ მოხდეს შესრულებული სამუშაოს დაკარგვა, რაც შესაძლებელია გამოწვეული იყოს პროგრამის ან სისტემის გაუმართავი მუშაობით. ჩვენს შემთხვევაში ტექსტური პროცესორისთვის მესამე thread-ად შეიძლება სარეზერვო სამუშაოს განმახორციელებელი thread-ის დამატება (ნახ. 7.1).



ნახ. 7.1. სამი და ოთხი ნაკადის გამოყენებელი პროცესი

თუ პროგრამა იქნებოდა გათვლილი მხოლოდ ერთი thread-ის შესრულებაზე, მაშინ, მონაცემების დისკზე რეზერვირების thread-ის შესრულების დაწყებიდან დასრულებამდე, მოხდებოდა კლავიატურიდან განხორციელებული ნებისმიერი მოქმედების უგულვებელყოფა და მომხმარებელი ამას სისტემის წარმადობას დააბრალებდა. აქ შეიძლება მოვიქცეთ ისე, რომ

კლავიატურიდან განხორციელებული მოქმედების დროს შეწყდეს დისკვიზე რეზერვირების ნაკადი და მივაღწიოთ მაღალ წარმადობასაც, მაგრამ ეს მიგვიყვანდა წყვეტის გამომყენებელ უფრო რთულ მოდელთან. სამი thread-ის გამომყენებელი მოდელი უფრო მარტივია. პირველი ნაკადი ურთიერთქმედებს მომხმარებელთან, მეორე ახდენს ტექსტის ფორმატირებას, ხოლო მესამე კი პერიოდულად დისკვიზე ინახახავს მონაცემებს.

ცხადია, რომ სამი განსხვავებული პროცესი ერთი პროცესის შიგნით სამი განსხვავებული ნაკადების მსგავსად ვერ იმუშავებდა, ვინაიდან დოკუმენტთან მუშაობას საჭიროებს სამივე ნაკადი. ნაკადები იყენებენ მეხსიერების ერთ ნაწილს და შესაბამისად, ყველა მათგანს გააჩნია წვდომა დოკუმენტზე, რისი გაკეთებაც პროცესების შემთხვევაში გარკვეულ პრობლემებთანაა დაკავშირებული.

7.5. ნაკადების კლასიკური მოდელი

პროცესის მოდელი ეყრდნობა ორ დამოუკიდებელ ცნებას: რესურსებისა და შესრულებების დაჯგუფება. ზოგჯერ უმჯობესია ამ ცნებების განცალკევება, რომლის დროსაც თავს იჩენს thread-ის ცნება.

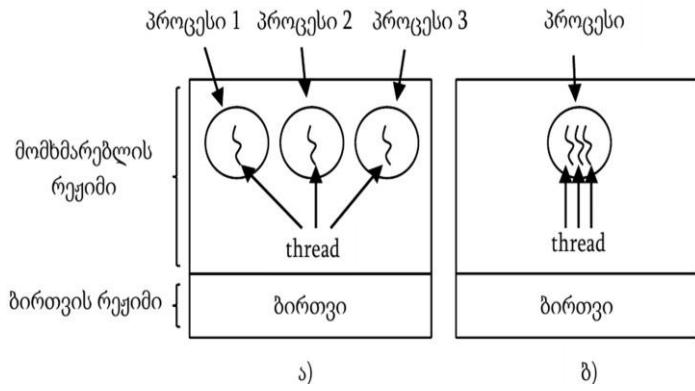
პროცესის განმარტების მიხედვით, ერთის მხრივ, ის წარმოადგენს ურთიერთქმედი რესურსების დაჯგუფების მეთოდს. პროცესს გააჩნია საკუთარი მისამართების სივრცე, რომელშიც განთავსებულია მის შესასრულებლად განკუთვნილი პროგრამული ტექსტი (კოდი) და მონაცემები, და ასევე სხვა რესურსები. რესურსების რიცხვს შეიძლება მივაკუთვნოთ გახსნილი ფაილები, სიგნალები და მათი დამტუშავებლები და ა.შ. ამ რესურსების მართვა საგრძნობლად გამარტივდებოდა თუ შევძლებდით მათ წარმოდგენას ერთი პროცესის სახით. მეორეს მხრივ, პროცესი თავის შიგნით შეიცავს thread-ს. thread-ს გააჩნია ბრძანებათა საკუთარი მთვლელი, რომელიც აკონტროლებს შესასრულებელი ინსტრუქციების თანმიმდევრობას, გააჩნია რეგისტრები, რომლებშიც ინახება მონაცემები და ცვლადები. ასევე, გააჩნია შესრულების პროტოკოლების (წესებისა და პროცედურების) სტეკი, რომელიც ყოველი გამოძახებული პროცედურისთვის შეიცავს ერთ ფრეიმს, რომელსაც არ დაუბრუნებია მართვა. მიუხედავად იმისა, რომ thread-ი არსებობს პროცესის შიგნით (იქმნება, იშლება და ეწევა სასარგებლო საქმიანობას), აუცილებელია thread-ის და პროცესის ცნებების ერთიმეორისგან განსხვავება.

პროცესის მოდელში thread-ები იძლევიან პროცესის ერთიმეორისგან დამოუკიდებელ ნაწილებად დაყოფის შესაძლებლობას, რომელთა შესრულება პარალელურ რეჟიმში შესაძლებელია. ერთი პროცესის შიგნით რამდენიმე thread-ის არსებობა შეიძლება განვიხილოთ როგორც ანალოგი, ერთ გამოთვლით სისტემაში რამდენიმე პროცესის არსებობისა. პირველ შემთხვევაში thread-ები იყენებენ საერთო მეხსიერების სივრცეს და რესურსებს, ხოლო მეორე შემთხვევაში პროცესები ინაწილებენ საერთო ფიზიკურ მეხსიერებას, დისკებს, პრინტერს და სხვა რესურსებს. ვინაიდან thread-ებს გააჩნიათ პროცესების თვისებები ხშირად მათ გამარტივებულ პროცესებსაც უწოდებენ.

ტერმინი მრავალნაკადიანი რეჟიმი გამოიყენება იმ სიტუაციის აღსაწერად, რომლის დროსაც ერთი პროცესის ფარგლებში დაშვებულია რამდენიმე thread-ის შესრულება.

ნახ. 7.2. ა)-ზე ნაჩვენებია სამი პროცესი, რომელთაგან თითოეულს გააჩნია საკუთარი მისამართების სივრცე და ერთი thread-ი, ბ)-ზე კი ნაჩვენებია ერთი პროცესი სამი thread-ით. მიუხედავად იმისა, რომ ორივე შემთხვევაში გვაქვს სამი thread-ი ა)-ზე თითოეული მათგანი

მუშაობს საკუთარ მისამართების სივრცეში, ხოლო ბ)-ზე სამივე იყენებს საერთო მეხსიერების სივრცეს.



ნახ. 7.2. სამი პროცესი თითოეული ერთი thread-ით (ა).

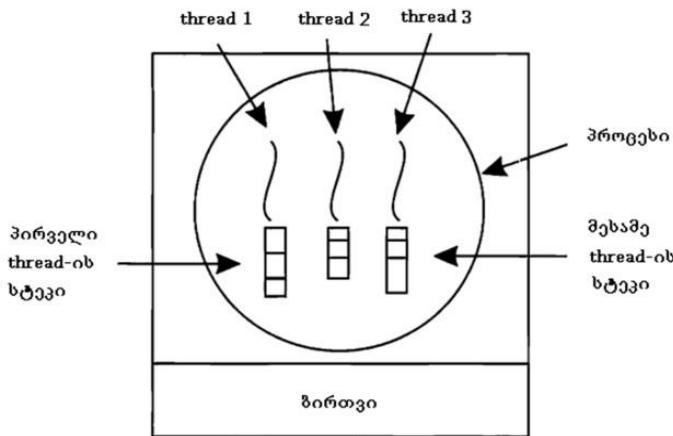
ერთი პროცესი სამი thread-ით (ბ).

როდესაც მრავალნაკადიანი პროცესი სრულდება ერთპროცესორულ გამოთვლით სისტემაში პროცესორი thread-ებს გამოეყოფა მონაცვლეობით და იქმნება thread-ების (კვაზი) პარალელური შესრულების ილუზია.

ერთი პროცესის ფარგლებში სხვადასხვა thread-ები არ სარგებლობენ ისეთი დამოუკიდებლობით, როგორითაც ერთ გამოთვლითი სისტემაში სხვადასხვა პროცესები. პროცესის შიგნით ყველა thread-ი სარგებლობს საერთო მეხსიერების სივრცით და მონაცემებით. ვინაიდან thread-ი სარგებლობს საერთო მეხსიერებაზე შეუზღუდულავი მიმართვის უფლებით, მას შეუძლია მიმართოს სხვა thread-ისთვის განკუთვნილ მონაცემებს, განახორციელოს მათი კითხვა, ჩაწეროს იქ საკუთარი მონაცემები ან წაშალოს იქიდან სხვისი მონაცემები. პროცესებისგან განსხვავებით, რომლებიც შესაძლებელია ერთმანეთის მიმართაც იყვნენ „მტრულად“ განწყობილი, ერთი პროცესის შიგნით არსებული thread-ები ეწევიან საერთო საქმიანობას, ამიტომ მათ შორის არ არსებობს დაცვის მექანიზმები და ასეთი მექანიზმების არსებობის საჭიროებაც არაა.

ტრადიციული (ანუ ერთ thread-იანი) პროცესების მსგავსად, thread-ები უნდა იმყოფებოდნენ შემდეგი მდგომარეობებიდან ერთერთში: სრულდება, ბლოკირებულია, მზადაა შესასრულებლად ან დასრულდა. შესრულების მდგომარეობაში thread-ი იკავებს პროცესორს. ბლოკირებული thread-ი ელოდება შესაბამის მოვლენას. მაგალითად, როდესაც thread-ი ასრულებს კლავიატურიდან კითხვის სისტემურ გამოძახებას ის იქნება ბლოკირებული სანამ კლავიატურიდან არ იქნება რაიმე მონაცემი აკრეფილი. thread-ი შეიძლება ბლოკირებული იყოს რაიმე შიდა მოვლენის ლოდინის გამო. მზადყოფნის მდგომარეობაში მყოფი thread-ი იწყებს შესრულებას, როგორც კი დადგება მისი რიგი. thread-ების გადასვლა მდგომარეობიდან მდგომარეობაზე ანალოგიურია პროცესებისთვის შესაბამისი გადასვლებისა.

შევნიშნოთ, რომ ყოველ thread-ს გააჩნია საკუთარი სტეკი (ნახ. 7.3). ყოველი thread-ის სტეკი შეიცავს თითო ფრეიმს უკვე გამოძახებული, მაგრამ ჯერ არ დასრულებული პროცედურისთვის. ასეთი ფრეიმი შეიცავს პროცედურის ლოკალურ ცვლადებს და გამოძახების დასრულებისას მართვის დასაბრუნებელ მისამართს. მაგალითად, თუ X პროცედურა იძახებს Y პროცედურას, Y კი - Z პროცედურას, მაშინ Z-ის შესრულებისას სტეკში იქნება სამივე პროცედურა. თითოეული thread-ი, როგორც წესი, ასრულებს სხვადასხვა პროცედურას და შესაბამისად, შესრულდება საკუთარ გარემოში. ამიტომ თითოეული thread-ი საჭიროებს საკუთარ სტეკს.



ნახ. 7.3. thread-ი საკუთარი სტეკით

7.6. thread-ების რეალიზაცია

thread-ები რეალიზაციის შესაძლებელია განხორციელდეს როგორც ბირთვის ისე, მომხმარებლის რეჟიმში. ასევე, განიხილება thread-ები რეალიზების კიდევ ერთი მეთოდი, რომელიც წარმოადგენს დავის სგანს და ცნობილია ჰიბრიდული მეთოდის სახელით. ჰიბრიდული მეთოდი გულისხმობს მომხმარებლის და ბირთვის რეჟიმში ერთდროულად thread-ების არსებობას ისე, რომ შენარჩუნებული იყო მათი გამოყენებით მიღებული უპირატესობები.

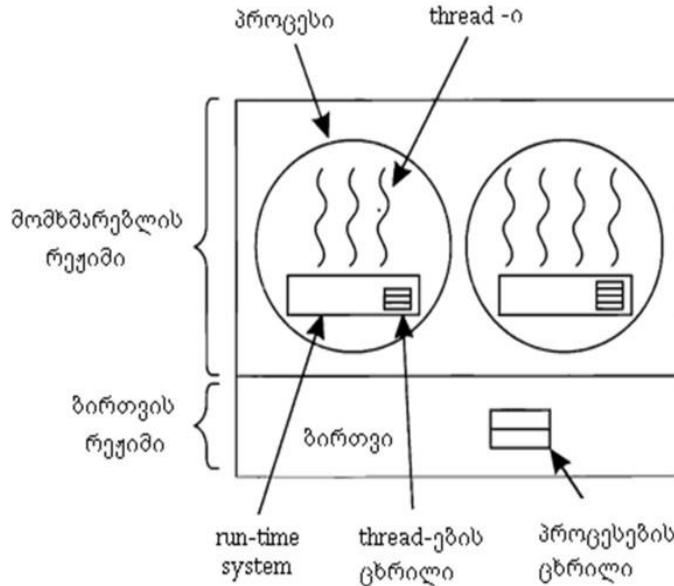
7.6.1. thread-ების რეალიზაცია მომხმარებლის რეჟიმში

thread-ის რეალიზაციის პირველი მეთოდი გულისხმობს thread-ების სრული ნაკრების განთავსებას მომხმარებლის რეჟიმში. ამ ნაკრების არსებობაზე ბირთვს წარმოდგენა არ გააჩნია. ის ურთიერთქმედებს პროცესებთან, რომლებიც მომხმარებლის რეჟიმში წარმოდგენილია thread-ების ნაკრების სახით. thread-ების გამოყენების მნიშვნელოვანი უპირატესობა მდგომარეობს იმაში, რომ მათი რეალიზება შესაძლებელია ერთპროცესორულ სისტემაში, რომელსაც შეიძლება არც გააჩნია thread-ების მხარდაჭერა. ერთპროცესორულ სისტემებში thread-ები რეალიზებულია სპეციალური ბიბლიოთეკების მეშვეობით. სისტემაში thread-ები რეალიზებულია ერთიანი სტრუქტურით (ნახ. 7.4). ისინი სრულდებიან პროგრამების შესრულების სისტემაში (runtime systems), რომელშიც თავმოყრილია thread-ების მმართველ პროცედურათა ნაკრებები (pthread_create, pthread_exit, pthread_join, pthread_yield და ა.შ.).

მომხმარებლის რეჟიმში ყოველ პროცესს გააჩნია thread-ების ცხრილის მხარდაჭერა, რომელშიც ინახება პროცესის შიგნით არსებულ thread-ებზე ინფორმაცია. ის წარმოადგენს ბირთვის რეჟიმში არსებული პროცესების ცხრილის ანალოგს, იმ განსხვავებით, რომ ყოველი thread-ისთვის ის შეიცავს მის ბრძანებათა მთვლელს, სტეკის მიმთითებელს, რეგისტრებს, მდგომარეობას და ა.შ., და პროცესი მას იყენებს thread- ებზე გარკვეული მოქმედებების გასახორციელებლად (წარმოქმნა, წაშლა და ა.შ.). thread-ის წარმოქმნის, მდგომარეობის შეცვლის, ან მის მიერ გარკვეული სამუშაოს შესრულების შემდეგ ხდება thread-ების ცხრილის შესაბამის ჩანაწერის განახლება.

თუ thread-ის მიერ განხორცილებულმა მოქმედებამ შეიძლება გამოიწვიოს მისი ბლოკირება, მაშინ ამ მოქმედების განხორციელებამდე ის იძახებს runtime system-ის შესაბამის პროცედურას,

რომელიც აფასებს ბლოკირების შესაძლებლობას. თუ ბლოკირება გარდაუვალია, მაშინ ის thread-ების ცხრილში ინახავს საკუთარი რეგისტრების შემცველობას და წყდება მისი შესრულება. ამის შემდეგ thread-ების ცხრილიდან რეგისტრებში იტვირთება ახალი thread-ის შესაბამისი ინსტრუქციები და შესრულებას იწყებს ის.



ნახ. 7.4. thread-ების ნაკრები მომხმარებლის რეჟიმში

ერთერთი მნიშვნელოვანი განსხვავება thread-ისა პროცესისგან მდგომარეობს იმაში, რომ, თუ thread-ის მიერ შესრულებული პროცედურა იწვევს მის შეჩერებას გარკვეული დროითი შუალედით, მაშინ გამოყენებული პროცედურა ინახავს ინფორმაცის thread-ის მიმდინარე მდგომარეობაზე და მის მიერ განხორციელებულ სამუშაოზე. პროცედურას, ასევე, შეუძლია გამოიძახოს დამგეგმავი შესასრულებლად შემდგომი thread-ის ასარჩევად. მსგავსი პროცედურები ლოკალურია და მათი გამოყენება უფრო ეფექტურია ვიდრე ბირთვის რეჟიმში შესაბამის გამოძახებების გამოყენება.

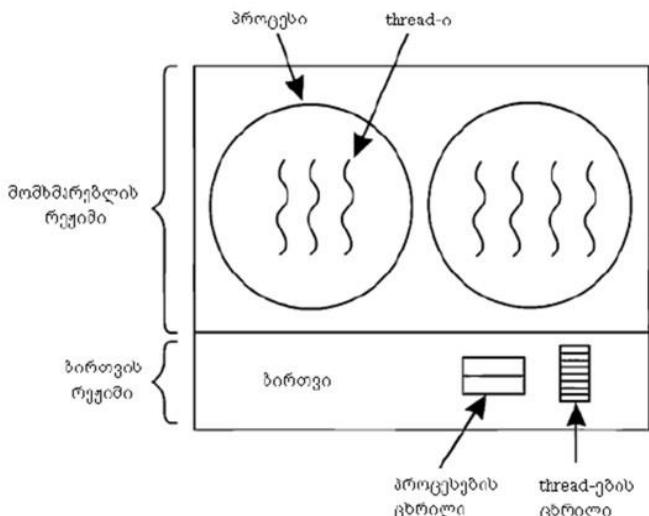
მომხმარებლის რეჟიმში რეალიზებულ thread-ებს, მრავალ უპირატესობასთან ერთად, გაჩნია ნაკლოვანებებიც. ასეთი საკითხების რიცხვს განეკუთვნება შემდეგი: როგორ უნდა მოხდეს thread-ის ბლოკირება; ერთპროცესორულ სისტემაში დროის ნებისმიერ მომენტში პროცესორთან იმყოფება ერთი thread-ი. ერთი პროცესის ფარგლებში არ არსებობს ტაიმერიდან წყვეტა. როგორ უნდა მოხდეს thread-ისთვის პროცესორის ჩამორთმევა, თუ მან უსასრულოდ გადაწყვიტა შესრულება; არ არსებობს thread-ების სიგნალებით ინფორმირების მექანიზმი.

7.6.2. ნაკადების რეალიზაცია ბირთვში

როგორც ნახ. 7.5-ზეა ნაჩვენები, აქ უკვე მომხმარებლის რეჟიმში გამოყენებული runtime system-ის ან მისი მსგავსი პროგრამების გამოყენების აუცილებლობა არ არსებობს. ასევე, ყოველ პროცესში არ არის thread-ების ცხრილი. ნაცვლად ამისა ბირთვში არის thread-ების ცხრილი, რომელშიც მოცემულია სისტემაში არსებული ყველა thread-ი. როდესაც thread-ი საჭიროებს ახალი thread-ის წარმოქმნას ან სხვა thread-ზე გარკვეულ ზემოქმედებას ის მიმართავს ბირთვს, რომელიც შედგომ ანხორციელებს შესაბამის მოქმედებებს და აახლებს thread-ების ცხრილს.

ბირთვში არსებულ thread-ების ცხრილი შეიცავს მომხმარებლის რეჟიმში ანალოგიური ცხრილის ველების მსგავს ველებს (რეგისტრები, მდგომარეობა და ა.შ.). ეს ინფორმაცია წარმოადგენს ქვესიმრევლეს იმ ინფორმაციისა, რომლის მხარდაჭერაც გააჩნია ბირთვს ტრადი-

ციული პროცესებთან მიმართებაში.



ნახ. 7.5. thread-ების ნაკრები ბირთვის რეგისტრი

thread-ების ბლოკირებისთვის რეალიზებული სისტემური გამოძახებების მსგავსი გამოძახებები ბირთვის რეგისტრი საჭიროებენ შედარებით მეტ დანახარჯებს, ვიდრე მომხმარებლის რეგისტრი განხორციელებული მსგავსი გამოძახებები. როდესაც ხორციელდება thread-ის ბლოკირება ბირთვს საკუთარი გადაწყვეტილებით შეუძლია აამუშაოს მიმდინარე პროცესის სხვა thread-ი ან ნებისმიერი სხვა პროცესის thread-ი. მომხმარებლის რეგისტრი რეალიზებული thread-ების შემთხვევაში კი runtime system-ი ამუშავებს იმავე პროცესის thread-ებს სანამ მას არ ჩამოერთმევა პროცესორი.

ვინაიდან ბირთვში thread-ების წარმოქმნა და განადგურება მოითხოვს მნიშვნელოვან დანახარჯებს, ზოგიერთი სისტემები წარმოქმილ სისტუაციასთან გამკვლავების მიზნით იყენებს მეთოდს, რომელიც გულისხმობს დასრულებული thread-ის ხელმეორედ გამოყენებას. დასრულებული thread-ი ბირთვიდან არ იშლება. ნაცვლად ამისა ის ინიშნება, როგორც შესასრულებლად გამოუსადეგარი, მაგრამ ამით მისი სტრუქტურა ბირთვში არ ირღვევა. მოგვიანებით როდესაც საჭირო გახდება ახალი thread-ის წარმოქმნა, ნაცვლად ახლის წარმოქმნისა ხდება ძველი thread-ის გააქტიურება, რაც იძლევა წარმოქმნისთვის საჭირო დროის შემცირების შესაძლებლობას. ამ მეთოდის რეალიზება შესაძლებელია მომხმარებლის რეგისტრი, მაგრამ ამის აუცილებლობა არ არსებობს ვინაიდან მომხმარებლის რეგისტრი ამ მეთოდით thread-ის წარმოქმნისთვის დახარჯულ დროში შესაძლებელია განხორციელდეს thread-ის ჩვეულებრივი წარმოქმნა.

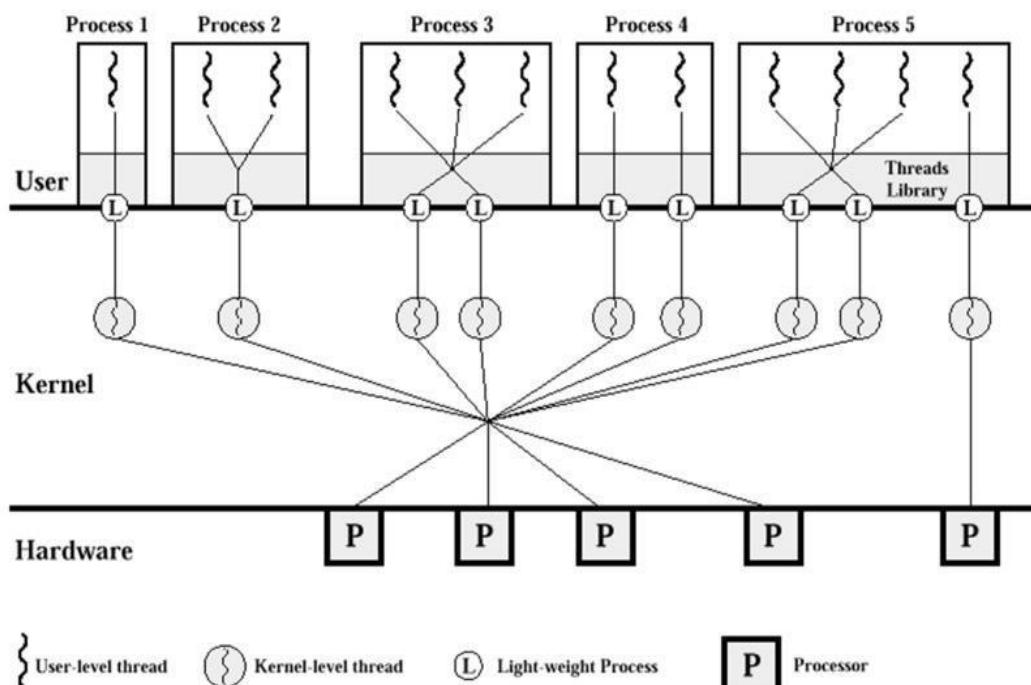
მიუხედავად იმისა, რომ ბირთვის რეგისტრი წარმოქმნილი thread-ები გვაძლევენ მრავალ პრობლემასთან გამკვლავების საშუალებას, მაინც რჩება პრობლემები, რომელთა გადაწყვეტის მიმართ ისინი უძლურნი არიან. მაგალითად, მრავალნაკადიანი პროცესის გაყოფისას წარმოქმნილ პროცესს ეყოლება იმდენი thread-ი, რამდენიც საწყისს, თუ ეყოლება მხოლოდ ერთი? უმეტეს შემთხვევებში საუკეთესო გამოსავალი დამოკიდებულია იმაზე, თუ რომელი პროცესის შესრულებაა დაგეგმილი. თუ ის აპირებს exec სისტემური გამოძახების შესრულებას, მაშინ მიზანშეწონილი იქნებოდა ერთი thread-ის არსებობა. თუ ის გააგრძელებს საწყისი პროცესის შესრულებას, მაშინ კი ყველა thread-ის არსებობა. მეორე პრობლემას წარმოადგენს სიგნალები. როგორც ადრე აღვნიშნეთ სიგნალები ეგზავნება პროცესებს და არა thread-ებს (კლასიკურ მოდელში). რომელმა thread-მა უნდა დაამუშავოს შემოსული სიგნალი? იქნებ საჭიროა წინასწარ thread-ებმა დაარეგისტრირონ საკუთარი სურვილი გარკვეული სიგნალების დამუშავებაზე? რა მოხდება იმ შემთხვევაში, თუ ერთიდაიმავე სიგნალის დამუშავებაზე სურვილი გამოთქვა ორმა ან მეტმა thread-მა? რომელს უნდა გადაეცეს ის

დასამუშავებლად? ეს არის პრობლემების მცირეოდენი ჩამონათვალი, რომლებიც დაკავშირებულია thread-ებთან.

7.6.3. thread-ების ჰიბრიდული რეალიზაცია

thread-ების რეალიზაციის შემდეგ მეთოდს წარმოადგენს მომხმარებლის და ბირთვის რეჟიმში thread-ების გამოყენებით მიღებული უპირატესობების გადატანა ახალ მეთოდში. ერთერთი მეთოდი, რომელიც ნაჩვენებია ნახ. 7.6-ზე, გულისხმობს ბირთვის რეჟიმში რამდენიმე thread-ის ყოლას, რომლებიც მომხმარებლის რეჟიმში წარმოდგენილი იქნება thread -ების გარკვეული ნაკრებების სახით. thread-ების ასეთნაირად რეალიზების შემთხვევაში პროგრამისტს შეუძლია განსაზღვროს, თუ რამდენი thread -ი დასჭირდება მას ბირთვის რეჟიმში და რამდენ ნაწილად შეუძლია გაყოს თითოეული მომხმარებლის რეჟიმში. ეს მეთოდი სარგებლობს მაქსიმალური მოქნილობით.

ასეთი მიდგომისას ბირთვისთვის ცნობილია მხოლოდ ბირთვის რეჟიმში არსებულ thread -ებზე ინფორმაცია. thread -ებთან მუშაობა ხორციელდება იმ რეჟიმის შესაბამისად, რომელ რეჟიმსაც მიეკუთვნება ის.



ნახ. 4.6. thread-ების ჰიბრიდული რეალიზაცი

ლექცია 8. კომპიუტერის მეხსიერების ორგანიზაცია

ოპერაციული სისტემის მნიშვნელოვან ამოცანას წარმოადგენს ეფექტურად მართოს გამოთვლითი მანქანის რესურსები, გამოყენებით პროგრამებს შესთავაზოს შესრულების გარემო და მონაცემების შესანახი სივრცე. პროგრამის მონაცემები დროის სხვადასხვა მომენტში შეიძ- ლება განთავსებული იყოს კომპიუტერულ სისტემაში გამოყენებულ სხვადასხვა მეხსიერებაში. გამოყენებითი პროგრამის ამუშავებამდე მის შესასრულებლად აუცილებელი მონაცემები განთავსებულია ინფორმაციის შესანახ მოწყობილობაზე, ხოლო მისი ამუშვების შემდეგ ეს მონაცემები გადადის ფიზიკურ მეხსიერებაში. პროგრამის შესრულების შესაბამისად მისი მონაცემები თავსდებიან პროცესორთან ახლოს ქეშ-მეხსიერებაში და რეგისტრებში.

ოპერაციულ სისტემას უწევს მომხმარებლის პროცესებსა და ოპერაციული სისტემის კომპონენტებს შორის მეხსიერების გადანაწილების ამოცანის გადაწყვეტა. ამ საქმიანობას ეწოდება მეხსიერების მართვა. მეხსიერება გამოთვლითი სისტემისთვის წარმოადგენს მნიშვნე- ლოვან რესურსს, რომელიც საჭიროებს დიდი ყურადღებით მართვას. ოპერაციული სისტემის ნაწილს, რომელიც ანხორციელებს მეხსიერების მართვას, მეხსიერების მენეჯერი ეწოდება.

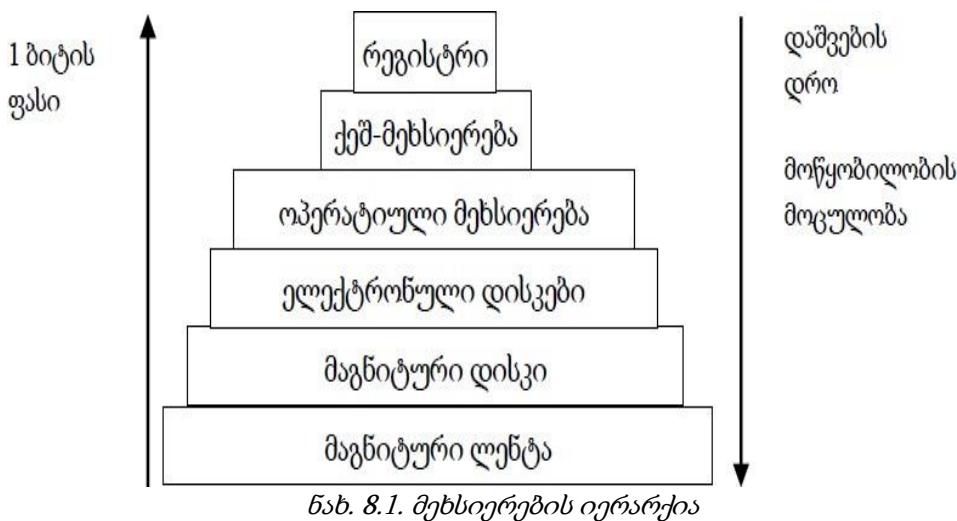
8.1. მეხსიერების ორგანიზაცია

გამოთვლითი სისტემის გამოჩენის პირველივე დღიდან მეხსიერება წარმოადგენს მის ერთერთ ძვირადღირებულ კომპონენტს. მიუხედავად იმისა, რომ ოპერატიული მეხსიერება სწრაფად იაფდება მისი ღირებულება გარე მეხსიერებასთან მიმართებაში მაინც რჩება მაღალი. გარდა ამისა, თანამედროვე ოპერაციული სისტემები და გამოყენებითი პროგრამები საჭიროებენ უფრო მეტი მოცულობის მეხსიერებას ვიდრე ადრე.

მეხსიერების სწრაფქმედება და მოცულობა შემოიფარგლება ფიზიკის და ეკონომიკის კანონებით. ყველა ელექტრონული მოწყობილობა მონაცემებს გადასცემს ელექტრული სიგნალების მეშვეობით. არსებობს ელექტრული სიგნალების გავრცელების ზღვრული სიჩქარე. რაც უფრო დიდია მანძილი ორ დაშორებულ წერტილს (ტერმინალს) შორის მით უფრო მეტი დროა საჭირო მათ შორის მონაცემების გადასაცემად. მეორეს მხრივ, პროცესორების აღჭურვა პროცესორის სწრაფქმედების შესაბამისი მოთხოვნების დამმუშავებელი დიდი მოცულობის მეხსიერებით საკმარისად გაზრდის გამოთვლითი სისტემის ღირებულებას.

გამოთვლითი სისტემების და ტექნოლოგიების განვითარებასთან ერთად მიმდინარეობდა გამოთვლით სისტემის სწრაფქმედებასა და ღირებულებას შორის შუალედის მოძიების სამუშაოები რა დროსაც გამოჩნდა მეხსიერების იერარქიული სტრუქტურა (ნახ. 8.1). მეხსიერების იერარქიის თავში განთავსებულია ყველაზე სწრაფქმედი მეხსიერება, ნაკლები მოცულობითა და ინფორმაციის შესანახ ბიტზე მაღალი ღირებულებით, ხოლო იერარქიის ბოლო დონეზე განთავსებულია ნელი, იაფი ღირებულების და დიდი მოცულობის მეორადი მეხსიერების მოწყობილობები.

რეგისტრი - არის სისტემაში გამოყენებული ყველაზე ძვირადღირებული და სწრაფქმედი მეხსიერება. ის დამზადებულია იმავე ტექნოლოგიით რაც პროცესორები და სწრაფქმედებითაც არ ჩამოუვარდება მას. შემდეგ დონეზე განთავსებულია ქეშ-მეხსიერება. მისი სწრაფქმედება იზომება დროითი დაყოვნებებით - შუალედით, რომელიც საჭიროა მონაცემების გადასაცემად.



როგორც წესი, დროითი დაყოვნებები იზომება ნანოწამებში (ნწ) ან პროცესორის ციკლით. მაგალითად, Pentium 4 პროცესორისთვის პირველი დონის (L1) ქეშ-მეხსიერების დაყოვნება შეადგენს პროცესორის 2 ციკლს. მეორე დონის (L2) ქეშ-მეხსიერების დაყოვნება შეადგენს პროცესორის დაახლოებით 10 ციკლს. დღეისთვის პირველი და მეორე დონის ქეშ-მეხსიერებები ინტეგრირებულია პროცესორზე და, შესაბამისად, შეუძლიათ ისარგებლონ მონაცემების მაღალსიჩქარიანი გადაცემებით. პირველი დონის ქეშ-მეხსიერების მოცულობა შეადგენს რამდენიმე ათეულ ან ასეულ კბ-ს, ხოლო მეორე დონის ქეშ-მეხსიერების მოცულობა კი რამდენიმე ასეული კბ-დან რამდენიმე მბ-დდე შიძლება მერყეობდეს. მძლავრ პროცესორებზე შესაძლებელია გამოყენებული იყოს მესამე დონის (L3) ქეშ-მეხსიერება, რომელიც სწრაფქმედებით ჩამოუვარდება L2-ს, მაგრამ ოპერატიულ მეხსიერებასთან შედარებით გააჩნია მაღალი სწრაფქმედება.

იერაქიის შემდეგ დონეზე განთავსებულია ოპერატიული (ძირითადი, ფიზიკური ან უბრალოდ) მეხსიერება. ოპერატიული მეხსიერება პროცესორისთვის მონაცემების გადასაცემად იყენებს პროცესორის სისწრაფესთან შედარებით დაბალი სისტემურ სალტეს (system bus) და შესაბამისად დაყოვნება ამ შემთხვევაში მაღალია. თანამედროვე არქიტექტურებში ოპერატიული მეხსიერების დაყოვნება შეადგენს რამდენიმე ასეულ პროცესორის ციკლს. მისი მოცულობა მერყეობს რამდენიმე ასეული მბ-დან რამდენიმე ასეულ გბ-მდე (ტერაბიტი მძლავრ სერვერებში). რეგისტრები, ქეშ-მეხსიერება და ოპერატიული მეხსიერება დამოკიდებულია ელექტროენერგიაზე და, შესაბამისად, ელექტროენერგიის უცარი გამორთვის შემთხვევაში იკარგება მათში არსებული მონაცემები.

მეხსიერების შემდეგ დონეებზე განთავსებულია მეხსიერების მეორადი (გარე) მოწყობილობები. ელექტრული დისკები, მაგნიტური დისკები და მაგნიტური ლენტა წარმოადგენენ გამოთვლით სისტემაში მონაცემების შენახვის დიდი მოცულობის და დაბალი ღირებულების მოწყობილობებს. ამ ტიპის მოწყობილობებში ინფორმაციაზე წვდომა ქეშ-მეხსიერებასთან შედარებით ხორციელდება რამდენიმე ასეულ ათასჯერ ნაკლები სისწრაფით. შენახვის მეორადი მოწყობილობების უპირატესობა მდგომარეობს იმაში, რომ ისინი არ არიან დამოკიდებული ელექტროენერგიაზე და უცარი გამორთვის შემთხვევაში მათზე არსებული ინფორმაცია არ იკარგება. მეორე უპირატესობას წარმოადგენს ის, რომ მათ მონაცემების შესანახად გააჩნიათ დიდი მოცულობა.

ოპერაციული სისტემის ამოცანას წარმოადგენს მეხსიერების იერარქიის მართვა. პროცესორის ეფექტურად გამოყენების მიზნით სისტემამ შესასრულებლად უნდა აირჩიოს ისეთი პროცესი, რომელიც შესანახი მოწყობილობიდან არ ელოდება მონაცემებს ან მას მსგავსი მონაცემები მიღებული აქვს და მზადაა შესასრულებლად. იერარქიის მუშაობის პრინციპი მდგომარეობს

შემდეგში. როდესაც დამგეგმავი ირჩევს პროცესს შესასრულებლად, მისი მონაცემები უნდა განთავსდეს პროცესორთან ახლოს. პროცესთან დაკავშირებული ინსტრუქციები და მონაცემები იტვირთება რეგისტრებში. თუ რეგისტრებში საჭირო ინფორმაცია ვერ მოიძებნა ინფორმაციის მოძიება გრძელდება ქეშ-მეხსიერებებზე. თუ საჭირო ინფორმაცია არსებობს იქ, მაშინ ის იქნება გადატანილი რეგისტრებში. წინააღმდეგ შემთხვევაში ინფორმაციის მოძიება გაგრძელდება შემდგომ დონეებზე სანამ ის არ იქნება მოძებნილი. ინფორმაციის მოძებნის შემდეგ კი ხორციელდება მისი გადატანა დაბალი დონიდან მაღალ დონეებზე. წინააღმდეგ შემთხვევაში ოპერაციული სისტემა აჩერებს პროცესის შესრულებას.

მეხსიერების მენეჯერის ამოცანას წარმოადგენს პროცესებს შორის მეხსიერების გადანაწილების საკითხი (ერთპროგრამულობა ან მულტიპროგრამულობა), გააკონტროლოს დროის მიმდინარე მომენტში მეხსიერების რა ნაწილია დაკავებული და რა ნაწილი თავისუფალი, აუცილებლობის შემთხვევაში თუ მეხსიერებაში პროცესისა და მისი მონაცემების განსათავსებლად არაა საკმარისი ადგილი, მაშინ იქიდან ამოიღოს (წაშალოს) გარკვეული პროცესი და მისი მონაცემები, ხოლო მის ადგილას განათავსოს ახალი პროცესი და მისი მონაცემები.

8.2. ლოკალურობა

აღმოჩნდა, რომ ასეთი იერარქიული ორგანიზაციისას მეხსიერების დონეზე დაშვების სიჩქარის შემცირებით მცირდება მასზე მიმართვის სიხშირე. გასაღებ როლს აქ თამაშობს რეალური პროგრამების თვისებები, რომელთაც შეუძლიათ გარკვეული პერიოდის განმავლობაში იმუშაონ მცირე რაოდენობის მისამართების ნაკრებთან. ეს დაკვირვებადი ემპირიული თვისება ცნობილია, როგორც ლოკალურობის ან მიმართვების ლოკალიზაციის პრიციპი.

ლოკალურობის პირობა დამახასიათებელია არამხოლოდ ოპერაციული სისტემის ფუნქციონირებისთვის არამედ ზოგადად მისი ხასიათისათვის. ოპერაციული სისტემის შემთხვევაში ლოკალურობის ახსნა შესაძლებელია, თუ გავითვალისწინებთ როგორ იწერება პროგრამები და ინახება მონაცემები, ანუ დროის გარკვეულ მომენტში მონაცემთა შემოსაზღვრულ ნაკრებთან მუშაობს პროგრამის გარკვეული ფრაგმენტი. კოდის ამ ნაწილისა და მონაცემების განთავსება მეხსიერებაში შესაძლებელია სწრაფი დაშვებით. შედეგად, მეხსიერებაზე რეალური დროითი დაშვება განისაზღვრება ზედა დონეზე დაშვების დროით, რაც განაპირობებს იერარქიული სქემის გამოყენების ეფექტურობას.

პროცესორის ქეში წარმოადგენს აპარატურის ნაწილს, ამიტომ ოპერაციული სისტემის მენეჯერი დაკავებულია ინფორმაციის განაწილებით კომპიუტერის ძირითად და გარე მეხსიერებებს შორის. ზოგიერთ სქემებში მონაცემთა ნაკადები ოპერატიულ და გარე მეხსიერებებს შორის რეგულირდება პროგრამისტის მიერ, თუმცა ეს დაკავშირებულია პროგრამისტის მხრიდან დროის დანაკარგთან. ამიტომ პროგრამისტები ცდილობენ მსგავსი საქმიანობა დააკისრონ ოპერაციულ სისტემას.

ძირითად მეხსიერებაში არსებულ მისამართებს, რომლებიც უთითებენ ფიზიკურ მეხსიერებაში მონაცემების რეალურ ადგილმდებარეობას, ეწოდებათ **ფიზიკური მისამართები**. ფიზიკური მისამართების სიმრავლეს, რომელთანაც მუშაობს პროგრამა, ეწოდება **ფიზიკური მისამართების სივრცე**.

8.3. ლოგიკური მეხსიერება

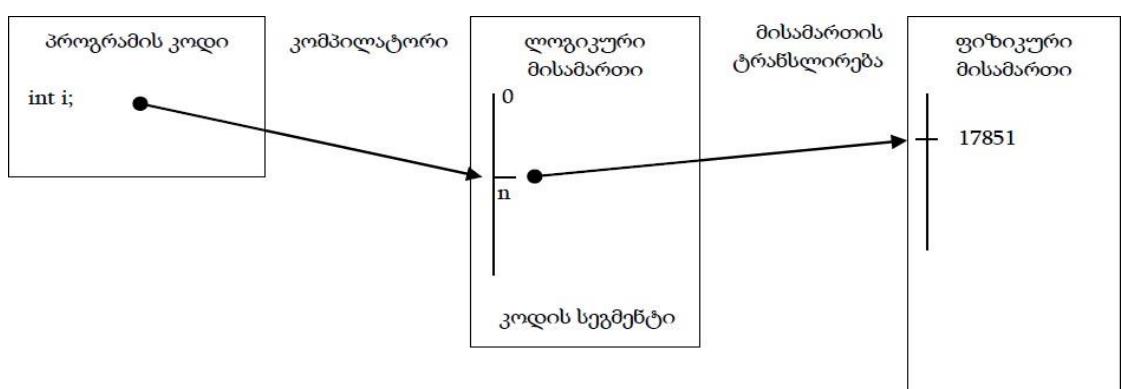
მეხსიერების აპარატული ორგანიზაცია უჯრედების წრფივი ერთობლიობის სახით არ

შეესაბამება პროგრამისტის წარმოდგენას იმაზე, თუ როგორაა ორგანიზებული პროგრამების და მონაცემების შენახვა. მრავალი პროგრამა წარმოადგენს ერთმანეთისგან დამოუკიდებლად შექმნილი მოდულების ნაკრებს. ხანდახან პროცესის შემადგენლობაში შემავალი ყველა მოდული ერთმანეთის მიყოლებით (უწყვეტი მიმდევრობით) თავსდება მეხსიერებაში, რითაც ქმნიან მისამართების წრფივ სივრცეს. ხშირად მოდულები თავსდებიან მეხსიერების სხვადასხვა მიღდამოში და გამოიყენებიან სხვადასხვაგვარად.

მისამართებს, რომლებსაც მიმართავს პროცესი, განსხვავდება იმ მისამართებისგან, რომლებიც რეალურად არსებობენ ოპერატიულ მეხსიერებაში. ყოველ კონკრეტულ შემთხვევაში პროგრამების მიერ გამოყენებადი მისამართები შესაძლებელია წარმოდგენილი იყვნენ სხვადასხვა მეთოდებით (მაგალითად, მისამართები გამოსავალ ტექსტში არის სიმბოლური). კომპილიატორი ამ სიმბოლურ მისამართებს აკავშირებს ცვლად მისამართებთან. პროგრამის მიერ გენერირებულ მსგავს მისამართს უწოდებენ **ლოგიკურ მისამართს** (ვირტუალური მეხსიერების მხარდაჭერ სისტემებში მას კირტუალურ მისამართს უწოდებენ). ყველა ლოგიკური (ვირტუალური) მისამართების სიმრავლეს ეწოდება ლოგიკური (ვირტუალური) მისამართების სივრცე.

8.4. მისამართების დაკავშირება

ლოგიკური და ფიზიკური მისამართების სივრცეები არც ორგანიზაციით და არც მოცულობით არ შეესაბამებიან ერთიმეორებს. ლოგიკური მისამართების სივრცის მაქსიმალური მოცულობა განისაზღვრება პროცესორის თანრიგიანობით და რიგ შემთხვევებში გაცილებით აჭარბებს თანამედროვე სისტემებში ფიზიკური მისამართების სივრცის მოცულობას. შესაბამისად პროცესორს და ოპერაციულ სისტემას უნდა შეეძლოს პროგრამის კოდში მიმართვების (მისამართების) ასახვა რეალურ ფიზიკურ მისამართებზე, რომელიც შეესაბამება პროგრამის მიმდინარე მდებარეობას ძირითად მეხსიერებაში. მისამართების ასეთ ასახვას მისამართების ტრანსლირება ან მისამართების დაკავშირება ეწოდება (ნახ. 8.2).



ნახ. 8.2. ლოგიკური მეხსიერების ფორმირება და მისი დაკავშირება ფიზიკურთან

პროგრამაში ოპერატორის მიერ წარმოქმნილი ლოგიკური მისამართების დაკავშირება ფიზიკურთან უნდა განხორციელდეს ოპერატორის შესრულების დაწყებამდე ან მისი შესრულების მომენტში. ამრიგად, ინსტრუქციებისა და მონაცემების დაკავშირება მეხსიერებასთან შესაძლებელია გაკეთდეს შემდეგ ეტაპებზე:

- კომპილაციის ეტაპი - როდესაც ცნობილია მეხსიერებაში პროცესის ზუსტი ადგილმდებარეობა, მაშინ უშუალოდ გენერირდება ფიზიკური მისამართი. პროგრამის საწყისი მისამართის შეცვლისას საჭიროა მისი კოდის ხელახლი კომპილირება.

- **ჩატვირთვის ეტაპი** - თუ კომპილაციის ეტაპზე უცნობია ინფორმაცია სადაა განთავსებული პროგრამა, კომპილატორი აგენერირებს გადატანად კოდს. ამ შემთხვევაში საბოლოო დაკავშირების გადადება ხდება ჩატვირთვის მომენტამდე. თუ საწყისი მისამართი იცვლება, საჭიროა მხოლოდ კოდის გადატვირთვა შეცვლილი სიდიდის გათვალისწინებით.
- **შესრულების ეტაპი** - თუ პროცესი შესაძლებელია გადაადგილებული იყოს შესრულების დროს მეხსიერების ერთი მიდამოდან მეორეში, დაკავშირება გადაიდება შესრულების ეტაპამდე. ამ შემთხვევაში სასურველია სპეციალიზირებული მოწყობილობების (მაგალითად, გადაადგილების რეგისტრების) არსებობა. მათი მნიშვნელობა ემატება პროცესის მიერ გენერირებულ ყოველ მისამართს. უმეტესი თანამედროვე ოპერაციული სისტემა ანხორციელებს მისამართების ტრანსლირებას შესრულების ეტაპზე, ამისთვის სპეციალური აპარატურული მექანიზმების გამოყენებით.

8.5. მეხსიერების მართვის სისტემა

მეხსიერების ეფექტური გამოყენების კონტროლის უზრუნველსაყოფად ოპერაციულმა სისტემამ უნდა განახორციელოს შემდეგი ფუნქციები:

- პროცესის მისამართების სივრცის ასახვა ფიზიკური მეხსიერების კონკრეტულ მიდამოზე;
- კონკურენტ პროცესებს შორის მეხსიერების განაწილება;
- პროცესის მისამართების სივრცეზე დაშვების კონტროლი;
- ოპერატიულ მეხსიერებაში საკმარისი ადგილის არარსებობის შემთხვევაში პროცესების გადატანა მეორად მეხსიერებაზე;
- თავისუფალი და დაკავებული მეხსიერების სივრცის აღრიცხვა.

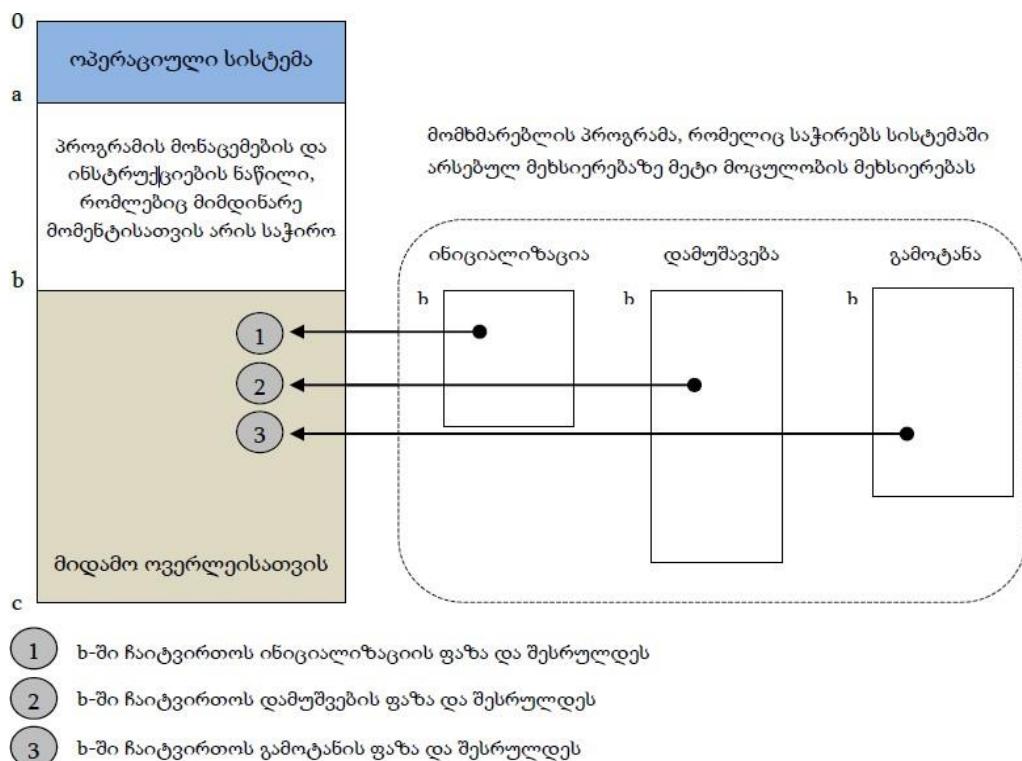
ქვემოთ განხილულია მეხსიერების მართვის კონკრეტული სქემები. ყოველი სქემა ემყარება მართვის გარკვეულ იდეოლოგიას, ასევე, ალგორითმებს, მონაცემთა სტრუქტურებს და დამოკიდებულია გამოყენებული სისტემის არქიტექტურულ თავისებურებებზე.

8.6. მეხსიერების ორგანიზაცია უწყვეტი ბლოკების მიმდევრობით

ადრეულ გამოთვლით მანქანასთან მუშაობა შეეძლო მხოლოდ ერთ მომხმარებელს და მისთვის ხელმისაწვდომი იყო გამოთვლითი სისტემის ყველა რესურსი. პროგრამების განთავსება მეხსიერებაში ხორციელდებოდა შემდეგნაირად: სისტემაში გამოჩენილი ყოველი პროგრამა იკავებდა იმ მოცულობის მეხსიერებას, რომელიც მის შესასრულებლად იყო საჭირო. ამასთან, მეხსიერების გამოყოფა ხორციელდებოდა უწყვეტი ბლოკების მიმდევრობით. თუ მეხსიერებაში პროგრამის განსათავსებლად არ იყო საკმარისი ადგილი, მაშინ ის ელოდებოდა შესაბამისი ადგილის გამოჩენას. ასეთი ორგანიზაციისას პროგრამებს დიდი ხნით უწევდათ დალოდება შესაბამისი სივრცის გამოთავისუფლებამდე. ამ პრობლემას ხშირად ემატებოდა ისიც, რომ შესაძლებელია პროგრამა ითხოვდა იმაზე მეტი მოცულობის მეხსიერებას ვიდრე გამოთვლით სისტემაში არსებობდა. მოგვიანებით გამოჩნდა მექანიზმი ე.წ. **ოვერლეის სტრუქტურა**, რომელმაც შესაძლებელი გახადა დიდი პროგრამების დაყოფა შედარებით მცირე ზომის ნაწილებად. ამ ნაწილების განთავსება ძირითად მეხსიერებაში და შემდგომ მათი შესრულება პრობლემას არ წარმოადგენდა.

8.6.1. ოვერლეი სტრუქტურა

ოვერლეი სტრუქტურის გამოყენებით პროგრამისტი პროგრამას ყოფდა ლოგიკურ ბლოკებად (ნახ. 8.3). როდესაც პროგრამა არ საჭიროებდა ძირითად მეხსიერებაში განთავსებულ გარკვეულ ბლოკს ის იქიდან შლიდა მას და მის ნაცვლად ითხოვდა მიმდინარე მომენტისთვის საჭირო ბლოკს.



ნახ. 8.3. ოვერლეი სტრუქტურა

ერთის მხრივ, ოვერლეი სტრუქტურა იძლეოდა ძირითადი მეხსიერების გაფართოების შესაძლებლობას. მეორეს მხრივ, კი მისი პროექტირება საჭიროებდა საკმაოდ დიდ დროს და პროგრამისტისგან ითხოვდა სტრუქტურის მოწყობის, მისი კოდის, მონაცემების და ოვერლეი სტრუქტურის აღმწერი ენის ცოდნას. ამ მიზეზებით ოვერლეი სტრუქტურის გამოყენება შემოიფარგლება მცირე მოცულობის ოპერატიული მეხსიერების მქონე კომპიუტერებით.

8.6.2. დაცვა ერთმომხმარებლიან სისტემებში

მომხმარებლის პროცესმა შეიძლება მიმართოს ოპერაციული სისტემის მიერ დაკავებულ მეხსიერების სივრცის ნაწილს და მიზანმიმართულად ან შემთხვევით დაზიანოს მისი კოდი. ოპერაციული სისტემის კოდის დაზიანების შემთხვევაში კი ვერც იპერაციული სისტემა და, შესაბამისად, ვერც პროცესი ვერ შეძლებს ნორმალურ ფუნქციონირებას. თუ პროცესი შეეცდება მიმართოს ოპერაციული სისტემის მიერ დაკავებული ოპერატიული მეხსიერების ნაწილს, მაშინ მომხმარებელს უნდა შეეძლოს მისი დანახვა და პროცესის კოდში ცვლილების შეტანა (შეცდომის აღმოფხვრა). დაცვის არარსებობის შემთხვევაში პროცესი მარტივად შეძლებს ამის გავეთებას.

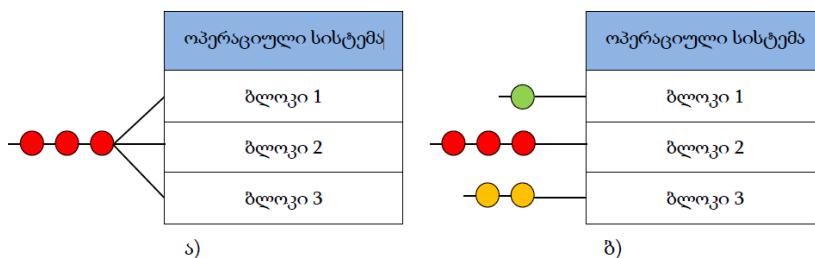
ერთმომხმარებლიან სისტემებში მეხსიერების უწყვეტ ბლოკებად ორგანიზაციით დაცვის უზრუნველყოფა შესაძლებელია პროცესორზე განთავსებული ერთი რეგისტრით - **საზღვრის რეგისტრით**, რომლის შემცველობის შეცვლაც შეუძლია მხოლოდ პრივილეგირებულ ინსტრუქციებს. ამ რეგისტრში ინახება ოპერატიული მეხსიერების ის მისამართი საიდანაც ხორციელდება მომხმარებლის პროცესების განთავსება. პროცესის ყოველი მიმართვისას ოპერატიულ მეხსიერებაზე მოწმდება საზღვრის მისამართის მნიშვნელობა. თუ პროცესი მიმართავს საზღვრის

მისამართის მნიშვნელობაზე ნაკლები მნიშვნელობის მისამართს, მაშინ მისი მოთხოვნა იქნება უარყოფილი.

8.6.3. მეხსიერების ორგანიზება ფიქსირებული მოცულობის ბლოკებად

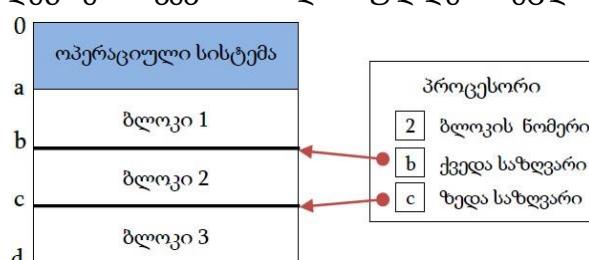
მეხსიერების ორგანიზაციის უმარტივეს სქემას წარმოადგენს მისი დანაწილება ფიქსირებული მოცულობის ბლოკებად. პროცესის ფიზიკური და ლოგიკური მისამართების დაკავშირება ხორციელდება კონკრეტულ ბლოკში ჩატვირთვის ეტაპზე, ხანდახან კომპილაციის ეტაპზეც. ყოველ ბლოკს შეიძლება გააჩნდეს პროცესების საკუთარი მიმდევრობა. ასევე, შესაძლებელია ყოველი ბლოკისთვის არსებობდეს გლობალური მიმდევრობა (ნახ. 8.4).

მეხსიერების მართვის ქვესისტემა აფასებს შემოსული პროცესის მოცულობას, ირჩევს მისთვის შესაბამის ბლოკს, ანხორციელებს პროცესის ჩატვირთვას ამ ბლოკში და მისამართების ტრანსლირებას. ამ სქემის ცხადი ნაკლოვანება მდგომარეობს იმაში, რომ ერთდროულად შესრულებადი პროცესების რაოდენობა შემოსაზღვრულია დანაწილებების რაოდენობით. მეორე არსებით ნაკლოვანებას წარმოადგენს ის, რომ შემოთავაზებული სქემა ძლიერ "იტანჯება" შიდა ფრაგმენტაციით (მეხსიერების ნაწილის დაკარგვით, რომელიც გამოყოფილია პროცესისთვის, მაგრამ ეს უკანასკნელი ვერ იყენებს მას). ფრაგმენტაცია წარმოიშობა იმის გამო, რომ პროცესი სრულად ვერ იკავებს მისთვის გამოყოფილ დანაწილებას ან იმიტომ, რომ ზოგიერთი დანაწილება საკმაოდ მცირეა მომხმარებლის შესრულებადი პროგრამის განსათავსებლად.



ნახ. 8.4. სქემა ფიქსირებული დანაწილებით: (ა)-პროცესების საერთო მიმდევრობით; (ბ)-პროცესების განსხვავებული მიმდევრობებით

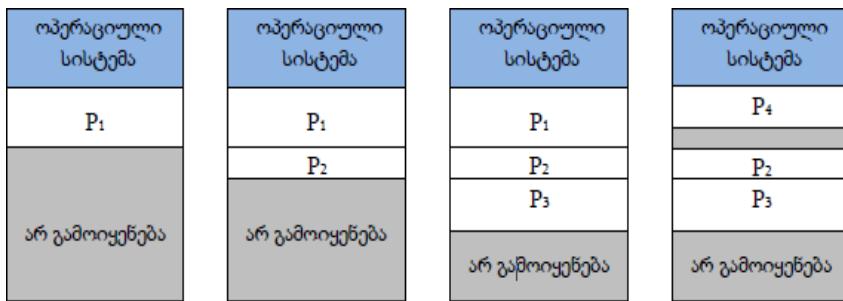
მეხსიერების ორგანიზების სქემის გართულებასთან ერთად საჭიროა მისი დაცვის მექანიზმების განსაზღვრა. ერთმომხმარებლიან სისტემებში საჭირო იყო ოპერაციული სისტემის დაცვა მხოლოდ მომხმარებლის ერთი პროცესისგან. მრავალპროგრამულ სისტემებში უკვე საჭიროა ოპერაციული სისტემის დაცვა პროგრამებისგან და, ასევე, პროგრამების დაცვა ერთიმეორისგან. უწყვეტი მიმდევრობით მეხსიერების ორგანიზაციით სისტემებში დაცვა ხორციელდება ორი საბაზო: ზედა და ქვედა რეგისტრით (პროგრამის განთავსების შუალედთან მიმართებაში), და ზღვრული რეგისტრით (ნახ. 8.5). როდესაც პროცესი მიმართავს მეხსიერებას, მაშინ სისტემა ამოწმებს მიმართვა არის თუ არა მოქცეული საბაზო რეგისტრში მითითებულ მნიშვნელობებს შორის. თუ მოთხოვნილი მისამართი თავსდება აღნიშნულ შუალედში, მაშინ პროცესის მოთხოვნა დაკმაყოფილდება. წინააღმდეგ შემთხვევაში ის დასრულდება შეცდომით.



ნახ. 8.5. მეხსიერების დაცვა მულტი-პროგრამულ სისტემებში

8.6.4. მეხსიერების ორგანიზაცია ცვლადი ზომის ბლოკებად

მეხსიერების ფიქსირებული ზომის ბლოკებად ორგანიზაციის მეთოდებისთვის დამახასიათებელია რესურსების არაეფექტური გამოყენება. მაგალითად, პროცესის განსათავსებლად ბლოკს შეიძლება გააჩნდეს მცირე მოცულობა, ან იყოს საკმარისად დიდი და მეხსიერების დიდი ნაწილი იკარგებოდეს. ცხადია, რომ ოპტიმალური გადაწყვეტა მდგომარეობს პროცესისთვის ზუსტად იმ მოცულობის მეხსიერების გამოყოფაში, რომელიც მას სჭირდება შესრულებისთვის. მეხსიერების ორგანიზაციას ამ ფორმით ეწოდება **მეხსიერების ორგანიზაცია ცვლადი მოცულობის ბლოკებად**. მეხსიერების ორგანიზაციის ასეთი მეთოდისას თავიდან მეხსიერება არაა დაყოფილი ბლოკებად. სისტემაში წარმოქმნილ ყოველ პროცესს მეხსიერება გამოეყოფა მკაცრად განსაზღვრული მოცულობით (ნახ. 8.6). მეხსიერების ორგანიზაციის ასეთი სქემა იტანჯება გარე ფრაგმენტაციით, რაც ნიშნავს შემდეგს: პროცესის დასრულების შემდეგ მეხსიერება თავისუფლდება და მის ადგილს იკავებს სხვა პროცესი (მას ცალკე გამოეყოფა მკაცრად განსაზღვრული მოცულობის ბლოკი). ახალი პროცესის განთავსებისას პროცესმა სრულად შეიძლება ვერ დაიკავოს აღნიშნული ბლოკი და მეხსიერების ის ნაწილი, რომელიც დარჩა ახალი პროცესის მიერ საჭირო სივრცის ათვისების შემდეგ შესაძლებელია არ აღმოჩნდეს საკმარისი რაიმე პროცესის განსათავსებლად. ასეთ შემთხვევაში მსგავსი ნაწილების ჯამური მოცულობა შესაძლებელია იყოს საკმარისი ერთი ან რამდენიმე პროცესის განსათავსებლად, მაგრამ მათი მცირე მოცულობისა და დიდი რაოდენობის გამო ეს შეუძლებელია.



ნახ. 8.6. პროცესებს შორის მეხსიერების განაწილების დინამიკა

გარე ფრაგმენტაციის პრობლემის ერთერთ გადაწყვეტას წარმოადგენს კუმშვის ინდიკატორი (თავისუფალი) მიდამოების გადაადგილება ზრდადი (კლებადი) მისამართების მხარეს ისე, რომ მთლიანმა თავისუფალმა მეხსიერებამ წარმოქმნას მისამართების უწყვეტი სივრცე. ამ მეთოდს ზოგჯერ უწოდებენ გადაადგილებადი დანაწილებების მეთოდს. კუმშვა არის ძვირადღირებული პროცედურა. კუმშვის ოპტიმალური სტრატეგიები საკმარისად მნელია და ისინი ხორციელდებიან სხვა მისამართებზე გადატანა/დაბრუნების კომბინაციებით.

8.8. მეხსიერების მართვის სტრატეგიები

მეხსიერების მართვის სტრატეგიები იქმნება ძირითადი მეხსიერების ოპტიმალური გამოყენების მიზნით. ამ სტრატეგიებს ყოფენ სამ ნაწილად:

1. ჩატვირთვის სტრატეგია;
2. განთავსების სტრატეგია;
3. ჩანაცვლების სტრატეგია.

ჩატვირთვის სტრატეგია განსაზღვრავს, თუ როდის უნდა განხორციელდეს ოპერატიულ მეხსიერებაში ახალი პროგრამის მონაცემებისა და ინსტრუქციების ჩატვირთვა. ეს სტრატეგია იყოფა

ორ ნაწილად: ჩატვირთვა მოთხოვნის საფუძველზე და წინასწარი ჩატვირთვა. ბოლო პერიოდამდე გამოიყენებოდა სტრატეგია ჩატვირთვა მოთხოვნის საფუძველზე, რომელიც გულისხმობს ჩასატვირთი პროგრამის მონაცემების და ინსტრუქციების გადატანას შენახვის მეორადი მოწყობილობიდან ოპერატიულ მეხსიერებაში მას შემდეგ რაც მასზე გაკეთდება მოთხოვნა. ამ სტრატეგიისთვის უპირატესობის მინიჭების მიზეზი იყო გამოთვლით სისტემებში დიდი მოცულობის ოპერატიული მეხსიერების არარსებობა. ასეთ შემთხვევაში ოპერატიულ მეხსიერებაში სხვადასხვა პროგრამების მონაცემებისა და ინსტრუქციების განთავსება გამოიწვევს მის უსარგებლოდ დაკავებას. გამოთვლით სისტემებში დიდ მოცულობით ოპერატიული მეხსიერების გამოჩენამ შესაძლებელი გახადა წინასწარი ჩატვირთვის სტრატეგიის გამოყენება, რამაც საგრძნობლად გაზარდა პროცესორის სწრაფქმედება.

ოპერატიულ მეხსიერებაში პროგრამის მონაცემების და ინსტრუქციების განთავსების აუცილებლობისას განთავსების სტრატეგია განსაზღვრავს, თუ სად უნდა განხორციელდეს მათი განთავსება. განთავსების სტრატეგია იყოფა სამ ნაწილად: პირველი შესაფერისი, საუკეთესოდ შესაფერისი და ნაკლებად შესაფერისი. პირველი შესაფერისი სტრატეგიის მიხედვით ახალი პროგრამის მონაცემებისა და ინსტრუქციების განთავსება ხორციელდება მოცულობით შესაფერის პირველივე დანაყოფში. საუკეთესოდ შესაფერისი სტრატეგიის მიხედვით პროგრამის მონაცემებისა და ინსტრუქციების განთავსება ხორციელდება ძირითადი მეხსიერების ისეთ დანაყოფში, რომელშიც მათი განთავსების შემდეგ რჩება მცირე მოცულობის ადგილი. ნაკლებად შესაფერისი სტრატეგიის შემთხვევაში პროგრამის მონაცემებისა და ინსტრუქციების განთავსება შესაძლებელია განხორციელდეს ძირითადი მეხსიერების ისეთ დანაყოფში, რომელშიც საკმარისი ადგილი იქნება ახალი პროგრამის მონაცემებისა და ინსტრუქციების განსათავსებლად. ოპერატიულ მეხსიერებაში მონაცემების განთავსებამდე ხორციელდება მისი დაყოფა გარკვეული მოცულობის ბლოკებად. თუ ბლოკები ფიქსირებული მოცულობისაა, მაშინ მათ ფურცლები ეწოდებათ. წინააღმდეგ შემთხვევაში მათ სეგმენტებს უწოდებენ.

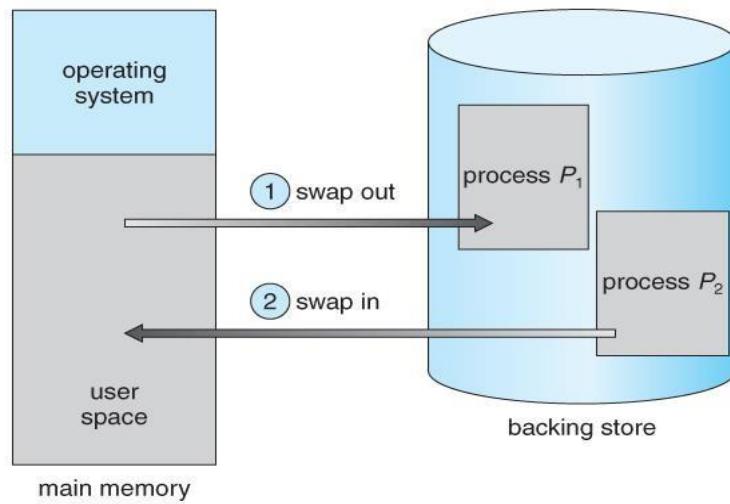
თუ ძირითად მეხსიერებაში ახალი პროგრამის მონაცემების და ინსტრუქციების განსათავსებლად საკმარისი ადგილი არაა, მაშინ იქიდან უნდა მოხდეს გარკვეული პროგრამის მონაცემების და ინსტრუქციების წაშლა. თუ რომელი მონაცემები და ინსტრუქციები უნდა წაიშალოს ოპერატიული მეხსიერებიდან განსაზღვრავს ჩანაცვლების სტრატეგია.

8.9. სვოპინგი (swapping)

მულტიპროგრამულ სისტემებში მეხსიერების ორგანიზაციის ამ დრომდე განხილულ ყველა სქემაში პროცესი იმყოფება ოპერატიულ მეხსიერებაში მის დასრულებამდე. ასეთი მიდგომის ალტერნატივას წარმოადგენს სვოპინგის გამოყენება, რომლის დროსაც პროცესი არ საჭიროებს მისი შესრულების ბოლომდე იმყოფებოდეს ძირითად მეხსიერებაში.

სვოპინგის გამოყენებელ ზოგიერთ სისტემებში ძირითად მეხსიერებაში ერთდროულად იმყოფება მხოლოდ ერთი პროცესი (ნახ. 8.7). ეს პროცესი სრულდება იმ დრომდე სანამ მას შეუძლია პროცესორის გამოყენება (მაგალითად, შეტანა/გამოტანის ოპერაციის განხორციელებამდე). თუ პროცესის მუშაობა ჩერდება, მაშინ სისტემა მეხსიერებას და პროცესორს გადასცემს სხვა პროცესს. შედეგად სისტემა მიმდევრობით გადასცემს პროცესებს მეხსიერებას. როდესაც პროცესი ათავისუფლებს რესურსს ხორციელდება მისი მონაცემების გადატანა ძირითადი მეხსიერებიდან შესანახ მოწყობილობაზე და მის ადგილს (ოპერატიულ მეხსიერებაში) იკავებს ახალი პროცესის მონაცემები. მისი დასრულების ან შეჩერების შემთხვევაში სისტემას პროცესის მონაცემები გადააქვს

შესანახ მოწყობილობაზე.



ნახ. 8.8. სვოპინგი

მოგვიანებით გამოჩნდნენ სვოპინგის უფრო რთული სქემები, რომლებშიც ოპერატიულ მეხსიერებაში შესაძლებელი იყო მეტი რაოდენობის პროცესების განთავსება. ასეთ სისტემებს, თუ მეხსიერებას საჭიროებს ახალი პროცესი, ოპერატიული მეხიერებიდან იქ არსებული პროცესი გადააქვთ შესანახ მოწყობილობაზე.

მეხსიერების ორგანიზაციის ასეთი სქემები შეიძლება განვიხილოთ, როგორც წინამორბედი თანამედროვე სისტემებში გამოყენებული ვირტუალური მეხსიერებისა.

ლუქცია 9. ფაილური სისტემა

მომხმარებლისთვის ფაილი ცნობილია, როგორც შესანახ მოწყობილობაზე განთავსებული, კონკრეტულ ინფორმაციასთან დაკავშირებული მეხსიერების უჯრედების ერთობლიობა, რომელზეც შესაძლებელია გარკვეული დასაშვები მოქმედებების განხორციელება. ფაილები ძირითადად განთავსებულია მეხსიერების მეორად მოწყობილობაზე, მაგრამ მიუხედავად ამისა პროგრამის შესრულების მომენტში ხდება მათი გადატანა პირველად მეხსიერებაში (ოპერატიულ მეხსიერება, ქეშ-მეხსიერება, რეგისტრები).

როგორც ვიცით, გამოთვლით სისტემაში მონაცემები ინახება მხოლოდ ორობითი ფორმით, რომლებთანაც უშუალო მუშაობა დაკავშირებულია დიდ სირთულეებთან. პროცესების და ვირტუალური მეხსიერების შემდეგ ფაილი არის გამოთვლით სისტემაში გამოყენებული კიდევ ერთი აბსტრაქცია. ფაილის აბსტრაქციის გამოყენებით ოპერაციული სისტემა ქმნის ინტერფეისს მომხმარებელსა და შენახვის მოწყობილობას შორის, რითაც მომხმარებლისგან მაღალ შესანახ მოწყობილობაზე მონაცემების პროცესირების (შენახვა, დამუშავების) თავისებურებებს.

9.1. მონაცემთა იერარქია

გამოთვლით სისტემაში არსებული ინფორმაცია ინახება გარკვეული იერარქიის სახით. ამ იერარქიის ყველაზე დაბალ დონეზე განთავსებულია ბიტები. ნებისმიერ გამოთვლით სისტემაში მონაცემების წარმოსადგენად გამოიყენება ორობითი კოდებით წარმოდგენილი ბიტების ერთობლიობა. N ბიტისგან შემდგარ მიმდევრობაში შესაძლებელია 2^N მიმდევრობის მიღება. განსხვავებული იერარქიის შემდეგ დონეზე განთავსებულია ფიქსირებული ზომის ბიტები (მაგალითად, ბაიტები, სიმბოლოები და სიტყვები). სიტყვა წარმოადგენს ბიტების მიმდევრობას, რომელთა ერთდროული დამუშავებაც შეუძლია პროცესორს. მაგალითად, 32-თანრიგა პროცესორისთვის სიტყვა შედგება 4 ბაიტისგან, ხოლო 64-თანრიგა პროცესორისთვის კი - 8 ბაიტისგან.

სიმბოლო ეს არის ბიტების ერთობლიობა, რომლებიც გამოიყენებიან ციფრების, ალფავიტის ასოების, სასვენი ნიშნების და სპეციალური სიმბოლოების აღსანიშნავად. მრავალ სისტემაში სიმბოლო შედგება 8 ბიტისგან, ამიტომ ასეთ სისტემებში სიმბოლოთა სიმრავლე შედგება 2^8 (= 256) სიმბოლოსგან. დღეისთვის ფართოდაა გავრცელებული სიმბოლოთა სამი ნაკრები: ASCII, EBCDIC და Unicode.

ASCII კოდში სიმბოლოები ინახება 8-ბიტიანი ფორმატით, ამიტომ ამ ნაკრებში მხოლოდ 256 სიმბოლოა. ამ შეზღუდვის გამო მასში მხარდაჭერილი არა სიმბოლოთა საერთაშორისო ნაკრები. EBCDIC სტანდარტი ძირითადად გამოიყენება IBM მიერ წარმოებულ მეინფრეიმებში. მასში სიმბოლო წარმოდგენილია 8 ბიტით ანუ ASCII სტანდარტის მსგავსად ამ სტანდარტშიც გამოიყენება მხოლოდ 256 სიმბოლო.

Unicode ეს არის საერთაშორისო სტანდარტი, რომელიც ფართოდ გამოიყენება ინტერნეტ სივრცეში და ისეთ პროგრამებში, რომელთაც გააჩნიათ სხვადასხვა ენების მხარდაჭერა. მასში თავმოყრილია მსოფლიოში თითქმის ყველა ქვეყნის ალფავიტის სიმბოლოები. Unicode-ში სიმბოლოების წარმოდგენისთვის გამოიყენება სამი 8-, 16- და 32-ბიტიანი ფორმატი. 8-ბიტიანი ფორმატი, რომელიც ასევე ცნობილია UTF-8 კოდირების სახელით, შეიცავს მხოლოდ ASCII სტანდარტში წარმოდგენილ სიმბოლოებს. 16- და 32-ბიტიანი ფორმატები კი გამოიყენება მეტი რაოდენობის სიმბოლოების გასაერთიანებლად.

იერარქიაში შემდეგი დონეებია: **ველი**, ანუ **სიმბოლოთა ჯგუფი** (მაგალითად, სახელი); **ჩანაწერი**, ანუ ველების ნაკრები (მაგალითად, სტუდენტზე ჩანაწერი შესაძლებელია

შეიცავდეს შემდეგ ველებს: იდენტიფიკატორი, სახელი, მისამართი და ა.შ.); **ფაილი**, ანუ ურთიერთდაკავშირებული ჩანაწერების ერთობლიობა. ამ იერარქიის თავში განთავსებულია ფაილური სისტემა.

9.2. ფაილები

როგორც უკვე აღვნიშნეთ, ფაილი წარმოადგენს მეხსიერების მისამართების ნაკრებს, რომლებზეც შესაძლებელია გახორციელდეს შემდეგი ოპერაციები:

- **შექმნა** (create file) - ახალი ფაილის შექმნა;
- **გახსნა** (open file) - ფაილის გამზადება მასზე დასაშვები მოქმედებების (კითხვა/ჩაწერა) განსახორციელებლად;
- **დახურვა** (close file) - ფაილის შემდგომ გახსნამდე მასზე დასაშვები მოქმედებების შეზღუდვა;
- **განადგურება** (destroy file) - ფაილის წაშლა;
- **კოპირება** (copy file) - ფაილის შიგთავსის ასლის გადატანა სხვა ფაილში;
- **ასახვა** (list file) - ფაილის შიგთავსის მონიტორის უკანზე ან ფურცელზე გამოტანა.

მონაცემთა ცალკეული ელემენტებზე, რომლებიც შენახულია ფაილებში, შესაძლებელია შემდეგი ოპერაციების განხორციელება:

- **კითხვა** (read data) - ფაილიდან მონაცემების კითხვა პროცესის მეხსიერებაში;
- **ჩაწერა** (write data) - პროცესის მეხსიერებიდან მონაცემების ჩაწერა ფაილში;
- **განახლება** (update) - ფაილში არსებული მონაცემეთა ელემენტების რაოდენობის შეცვლა;
- **ჩასმა** (insert data) - ფაილში ახალი ელემენტების ჩამატება;
- **წაშლა** (delete data) - ფაილიდან მონაცემთა ელემენტების წაშლა.

ფაილები ხასიათდებიან შემდეგი მახასიათებლებით:

- **მოცულობა** (size of file) - ფაილში შენახული მონაცემების მოცულობა;
- **ადგილმდებარეობა** (location of file) - ფაილის ადგილმდებარეობა შესანახ მოწყობილობაზე ან ფაილური სისტემაში;
- **დაშვების უფლებები** (accessibility of file) - ფაილზე წვდომის უფლებები;
- **ტიპი** (type) - ფაილის ტიპი. მაგალითად შესრულებადი ფაილი პროცესისთვის შეიცავს შესასრულებელ ინსტრუქციებს;
- **შეცვლადობა** (volatility of file) - ფაილში შენახულ მონაცემებში ცვლილების შეტანის სიხშირე;
- **აქტიურობა** (activity of file) - დროის მოცემულ შუალედში ფაილის ჩანაწერებზე მიმართვის სიხშირე.

ფაილი შეიძლება შედგებოდეს ერთი ან რამდენიმე ჩანაწერისგან. ფიზიკური ჩანაწერი ან ფიზიკური ბლოკი ეს არის შესანახ მოწყობილობაზე ჩაწერილი ან მისგან წაკითხული ინფორმაციის ერთეული. ლოგიკური ჩანაწერი ან ლოგიკური ბლოკი ეს არის მონაცემთა ნაკრები, რომელიც პროგრამების მიერ შეიძლება გაგებული იყოს როგორც ინფორმაციის ერთეული. თუ ფიზიკური ჩანაწერი შეიცავს ზუსტად ერთ ლოგიკურ ჩანაწერს, მაშინ ამბობენ, რომ ის შედგება დია მონაცემებისგან (unblocked records). წინააღმდეგ შემთხვევაში ამბობენ, რომ ჩანაწერი შედგება ჩაკეტილი მონაცემებისგან (blocked records).

9.3. ფაილური სისტემა

ფაილური სისტემა ახდენს ფაილების ლოგიკურ დალაგებას და განსაზღვრავს მათში შენახულ მონაცემებზე დაშვების უფლებებს. ფაილური სისტემის ფუნქციებია:

- ფაილების მართვა (file management) - ანხორციელებს ფაილების შენახვის მექანიზმის რეალიზებას, მათზე მიმართვას, მათ დაყოფას და უზრუნველყოფს მათ დაცვას;
- დამხმარე შესანახი მოწყობილობების მართვა (auxiliary memory management) - შესანახ მოწყობილობაზე ფალებისთვის სივრცის გამოყოფა;
- ფაილების მთლიანობა (file integrity) - გარანტირებს, რომ ფაილში შენახული ინფორმაცია არ დაიკარგება;
- დაშვების მეთოდები (access methods) - მეთოდები, რომლებიც გამოიყენება ფაილში შენახულ მონაცემებზე წვდომისთვის.

ფაილური სისტემა ძირითადად დაკავებულია შესანახ მეორად მოწყობილობასთან მუშაობით, მაგრამ მას ასევე შეუძლია მიმართოს მონაცემების სხვა მატარებელზე (მაგალითად, ოპერატიულ მეხსიერებაში) არსებულ ფაილებს.

ფაილური სისტემა მომხმარებელს საშუალებას აძლევს შექმნას, ცვლილებები განახორციელოს და წაშალოს ფაილი. ისინი ასევე უნდა იძლეოდნენ ფაილების სტრუქტურირების შესაძლებლობას იმ მეთოდებით, რომელიც მოხერხებული იქნება პროგრამებისთვის და ფაილებს შორის მონაცემების გაცვლისთვის. ფაილის მფლობელმა სხვა მომხმარებლებს უნდა შესთავაზოს საკუთარ ფაილზე დაშვების უფლებები. ფაილების გაზიარების მექანიზმა უნდა განახორციელოს კონტროლირებადი დაშვების სხვადასხვა ტიპები (მაგალითად, კითხვის უფლება, ჩაწერის უფლება, შესრულების უფლება და მათი სხვადასხვა კომბინაციები).

ფაილური სისტემას უნდა გააჩნდეს აპარატული დამოუკიდებლობა - მომხმარებელს ფაილზე მიმართვა რთული ფიზიკური სახელების ნაცვლად უნდა შეეძლოს სიმბოლური სახელებით. სიმბოლური სახელი ეს არის ლოგიკური, მომხმარებლისთვის მოსახერხებელი სახელი. სიმბოლური სახელები სისტემას საშუალებას აძლევენ მომხმარებელს შესთავაზოს მისი მონაცემების ლოგიკური ორგანიზაციის და მათზე ოპერაციების განხორციელების მარტივი შესაძლებლობა. ფიზიკური მისამართი კი არის ფაილის ფიზიკური ადგილმდებარეობა შესანახ მოწყობილობაზე. ფიზიკური წარმოდგენა კი დაკავშირებულია შესანახ მოწყობილობაზე მონაცემების განთავსებასთან და მონაცემებთან მუშაობისას მოწყობილობის სპეციფიკაციასთან. ფაილურმა სისტემამ მომხმარებლისგან უნდა დამალოს მოწყობილობის მუშაობის თავისებურებები და ის, თუ როგორ წარმოიდგინება მონაცემები შესანახ მოწყობილობაზე.

მონაცემების შემთხვევითი ან მიზანმიმართული დაკარგვის აცილების მიზნით ფაილურ სისტემას უნდა გააჩნდეს მონაცემების სარეზერვო ასლების შექმნის საშუალებები, რომლებიც იძლევიან მონაცემების ასლების ჭარბი რაოდენობით შექმნის შესაძლებლობას, და აუცილებლობის შემთხვევაში ასლებისგან ორიგინალის აღდგენის საშუალებები. გარდა ამისა, ფაილური სისტემას ისეთ გარემოში, სადაც წყდება მნიშვნელოვანი ამოცანები (მაგალითად, თავდაცვით სისტემებში, საბანკო სისტემებში), უნდა შეეძლოს მონაცემების დაცვა არასანქცირებული დაშვებისგან და მონაცემთა გაცვლისთვის გააჩნდეს შიფრაციისა და დეშიფრაციის საშუალებები. ეს საშუალებები გადაცემულ მონაცემებს ხელმისაწვდომს ხდიან მხოლოდ იმ მომხმარებლებისთვის, რომელთაც გააჩნიათ დეშიფრაციის გასაღები.

9.3.1. დირექტორიები

ფაილურ სისტემაში ფაილების ორგანიზებული განთავსებისა და სწრაფი წვდომისთვის

გამოიყენება დირექტორიები. **დირექტორია** ეს არის ფაილი, რომელიც შეიცავს ჩამონათვალს მასში შემავალ ფაილებსა და დირექტორიებზე და ინფორმაციას ფაილურ სისტემაში მათი ლოგიკური ადგილმდებარეობის შესახებ. ჩვეულებრივი ფაილებისგან განსხვავებით დირექტორიები არ ინახავენ მონაცემებს. ცხრილ 9.1-ში ნაჩვენებია დირექტორია ფაილის ტიპიური ველები.

ცხრილი 9.1. ველები, რომლებსაც შეიცავდეს დირექტორია

ველი	აღწერა
სახელი	ფაილის სახელის შემცველი სიმბოლური სტრიქონი
ადგილმდებარეობა	ფაილურ სისტემაში ფაილის განთავსების ლოგიკური მისამართი (მაგალითად, სრული ან მიმართუებითი სახელი)
მოცულობა	ფაილის მიერ დაკავებული ბაიტების რაოდენობა
ტიპი	ფაილის დანიშნულების აღნიშვნელი სიმბოლო (მაგალითად, მონაცემების ფაილი ან დირექტორია)
მიმართვის დრო	ფაილზე ბოლო მიმართვის დრო
შეცვლის დრო	ფაილში განხორციელებული ბოლო ცვლილების დრო
შექმნის დრო	ფაილის შექმნის დრო

9.3.2. ერთდონიანი ფაილური სისტემები

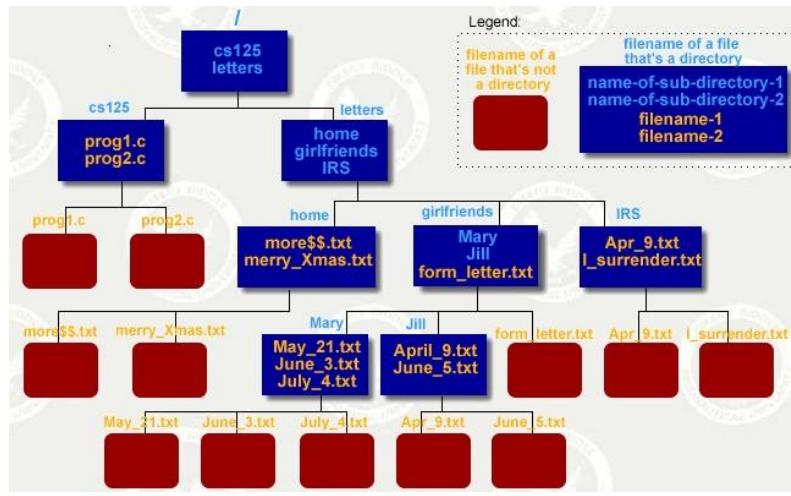
ფაილური სისტემის ყველაზე მარტივ რეალიზაციას წარმოადგენს ერთდონიანი ანუ **ბრტყელი ფაილური სისტემა**. ასეთ სისტემებში ყველა ფაილი ინახება ერთ დირექტორიაში. დირექტორის შიგნით ფაილების სახელები უნდა იყოს უნიკალური. ვინაიდან სხვადასხვა პროგრამების მიერ გამოყენებული ფაილების სახელები (რომლებიც შეიცავენ განსხვავებულ მონაცემებს) შესაძლებელია იყვნენ იდენტური, ამიტომ ამ ტიპის ფაილურმა სისტემამ ვერ ჰქოვა ფართოდ გავრცელება.

9.3.3. იერარქიულად სტრუქტურირებული ფაილური სისტემები

პროგრამებისთვის მისაღებია ფაილური სისტემის იერარქიული სტრუქტურა (ნახ. 9.1). **საბაზო** (root) დირექტორია განთავსებულია იერარქიის თავში და აღნიშნავს ფაილური სისტემის დასაწყისს. ნახ. 9.1-ზე საბაზო დირექტორია უთითებს სხვადასხვა მომხმარებლის დირექტორიაზე. მომხმარებლის დირექტორია კი თავის მხრივ უთითებს მასში შემავალ ფაილებზე ჩანაწერს. ყოველი ასეთი ჩანაწერი უთითებს ფაილურ სისტემაში ფაილის ლოგიკურ ადგილმდებარეობას.

ფაილის სახელი უნდა იყოს უნიკალური მხოლოდ მომხმარებლის დირექტორიის შიგნით. იერარქიულად სტრუქტურირებულ ფაილურ სისტემაში ყოველ დირექტორიას შეიძლება გააჩნდეს მრავალი ქვედირექტორია, მაგრამ მხოლოდ ერთი მშობელი დირექტორია. ფაილის სრულ სახელს ფაილურ სისტემაში წარმოადგენს გზა საბაზო დირექტორიიდან დანიშნულების ფაილამდე.

მრავალი ფაილური სისტემა ორგანიზებულია იერარქიული სტრუქტურით. თითოეულში ფაილამდე გზის აღსანიშნავად შესაძლებელია გამოყენებული იყოს სხვადასხვა აღნიშვნები. მაგალითად, Windows-ის სისტემაში საბაზო დირექტორიის აღსანიშნავად გამოიყენება ლათინური ალფავიტის სიმბოლოები C, D, და A.შ., ხოლო UNIX-მსგავს სისტემებში საბაზო დირექტორიის აღსანიშნავად კი გამოიყენება სპეციალური სიმბოლო '/'. ამ სისტემებში ასევე განსხვავებული ფაილების სრული სახელის ჩანაწერებიც.



ნახ. 9.1. იერარქიული ფაილური სისტემა

9.3.4. მიმართებითი სახელი

მრავალ ფაილურ სისტემაში ფაილამდე გზის ნავიგაციის გამარტივების მიზნით მხარდაჭერილია **სამუშაო დირექტორიის** (working directory) კონცეფცია. სამუშაო დირექტორია (რომელიც Windows და UNIX სისტემაში აღინიშნება სიმბოლოთი '.') მომხმარებელს საშუალებას აძლევს არ გამოიყენოს ფაილის სახელი დაწყებული საბაზო დირექტორიიდან.

დავუშვათ მიმდინარე სამუშაო დირექტორია არის /letters (ნახ. 9.1), მაშინ მიმართებითი სახელი (relative path) /letters/home/more\$\$.txt ფაილამდე იქნება /home/more\$\$.txt. ფაილის მიმართებითი სახელის გამოყენება იძლევა მისი სრული სახელის ჩანაწერის შემცირების საშუალებას. როდესაც ფაილზე მიმართვა ხორციელდება მიმართებითი სახელით ფაილური სისტემა ამ სახელს ავსებს სრულ სახელამდე (სახელი საბაზო დირექტორიიდან დანიშნულების ფაილამდე) და ამ ფორმით პოულობს მის ადილმდებარეობას ფაილური სისტემის ლოგიკურ ხეზე. მხოლოდ ამის შემდეგ ხორციელდება ფაილზე შესაბამისი მოქმედებები.

9.3.5. ლინკი (link)

ლინკი არის დირექტორიაში განთავსებული ჩანაწერი, რომელიც უთითებს სხვა დირექტორიაზე ან დირექტორიაში განთავსებულ ფაილზე. მომხმარებელი სისტემაში ნავიგაციის გამარტივების მიზნით ხშირად იყენებს ლინკს. სისტემაში ლინკი შეიძლება არსებობდეს ორი ტიპის: ლოგიკური და ფიზიკური.

ლოგიკური ლინკი წარმოადგენს დირექტორიაში არსებულ ჩანაწერს, რომელიც უთითებს სხვა დირექტორიაში განთავსებულ ფაილამდე გზას. UNIX-მსგავს სისტემაში ის ცნობილია სიმბოლური ლინკის, Windows-ში - მალმბობის (shortcut), ხოლო MAC OS-ში კი alias-ის სახელით. ფაილური სისტემა მითითებულ ფაილამდე გზას იკვლევს ლინკში არსებული მისამართით.

ფიზიკური ლინკი კი ეს არის დირექტორიაში არსებული ჩანაწერი, რომელიც უთითებს შესანახ მოწყობილობაზე ფაილის განთავსების ფიზიკურ მისამართს. ამ შემთხვევაში ფაილური სისტემა ფაილს პოულობს მისი ფიზიკური მისამართით.

ვინაიდან ფიზიკური ლინკი პირდაპირ უთითებს ფიზიკური მისამართების ბლოკებს, ამიტომ შესანახ მოწყობილობაში უწყვეტი მიმდევრობით განთავსებული მეტი მოცულობის თავისუფალი სივრცის მიღების მიზნით მონაცემთა ბლოკების გადაჯგუფების ოპერაციის (დეფრაგმენტაცია) განხორციელების შედეგად ფაილმა შესაძლებელია შეიცვალოს მისი ფიზიკური მისამართი. შედეგად ფიზიკური ლინკი გახდება უსარგებლო. ამ პრობლემის თავიდან აცილების მიზნით

სისტემაში არსებულ ყოველ ფაილში შეიძლება ჩაიწეროს ინფორმაცია მასზე გაკეთებულ ფიზიკურ ლინკზე. ჩანაწერის გაკეთება იძლევა დეფრაგმენტაციის ოპერაციის განხორციელების შემდეგ ფაილის ფიზიკური ლინკის განახლების შესაძლებლობას.

ვინაიდან ლოგიკურ ლინკში ინახება ფაილურ სისტემაში ფაილის ლოგიკური ადგილ-მდებარეობა, რომელიც დეფრაგმენტაციის ოპერაციის განხორციელებით არ იცვლება, ამიტომ ლოგიკური ლინკის განახლების აუცილებლობა არ არსებობს. ლოგიკური ლინკი გახდება უსარგებლო, თუ ფაილი, რომელსაც ის უთითებს გადაადგილებული იქნება ახალ დირექტორიაში ან მას შეეცვლება სახელი.

9.3.6. მეტამონაცემები (metadata)

უმეტეს ფაილურ სისტემაში მომხმარებლების მონაცემებთან და დირექტორიებთან ერთად ინახება სპეციალური მონაცემები (მაგალითად, შესანახ მოწყობილობაზე თავისუფალი ბლოკების რაოდენობა, ინფორმაცია ფაილზე ბოლოს განხორციელებულ ცვლილებაზე და ა.შ.), რომლებსაც მეტამონაცემები ეწოდებათ. ისინი უზრუნველყოფენ ფაილური სისტემის მთლიანობას და მომხმარებლის მხრიდან მათზე უშუალო ზემოქმედება შეუძლებელია. სანამ ფაილური სისტემა შეძლებდეს შესანახ მოწყობილობაზე მონაცემების განთავსებას საჭიროა მისი ფორმატირება.

მრავალი ფაილური სისტემა შეიცავს ე.წ. **სუპერბლოკს**, რომელშიც ინახება ფაილური სისტემის მთლიანობის უზრუნველყოფისთვის აუცილებელი ინფორმაცია. სუპერბლოკი შეიძლება შეიცავდეს:

- ფაილური სისტემის იდენტიფიკატორს, რომელიც ცალსახად განსაზღვრავს გამოყენებული ფაილური სისტემის ტიპს;
- ინფორმაცია ფაილურ სისტემაში ბლოკების რაოდენობაზე;
- ინფორმაცია თავისუფალი ბლოკების ადგილმდებარეობაზე;
- ინფორმაცია საბაზო დირექტორიის ადგილმდებარეობაზე;
- ფაილური სისტემის ბოლო შეცვლის დრო და თარიღი;
- ინფორმაცია ფაილური სისტემის შემოწმების აუცილებლობაზე.

სუპერბლოკის დაზიანების ან წაშლის შემთხვევაში ფაილურმა სისტემამ შეიძლება ვერ მიმართოს ფაილში განთავსებულ მონაცემებს, რაც თავის მხრივ მიგვიყვანს მომხმარებლის მონაცემების დაზიანებამდე. მონაცემების დაზიანების ან დაკარგვის რისკის შემცირების მიზნით ფაილურ სისტემაში ინახება სუპერბლოკის ასლების ჭარბი რაოდენობა, რომლებიც გადანაწილებულია შესანახ მოწყობილობაზე განთავსებულ სხვადასხვა ბლოკებში. ფაილურ სისტემაში სუპერბლოკის უმნიშვნელო დაზიანების შემთხვევაში სუპერბლოკი სისტემის მიერ იცვლება ერთერთი სარეზერვო ასლით.

9.3.7. ფაილური დესკრიპტორი

ფაილის გახსნის ოპერაციის შესრულებისას ოპერაციული სისტემა ეძებს ფაილის ადგილმდებარეობას ფაილურ სისტემაში. ერთსადაიმავე ფაილის მრავალჯერადი მიმართვის საჭიროებისას ფაილურ სისტემაში ფაილის ძებნის და ფაილის მრავალჯერადი გახსნის ოპერაციების ხელახალი საჭიროების თავიდან აცილების მიზნით ოპერაციული სისტემა გახსნილ ფაილზე ინფორმაციას ინახავს ოპერატიულ მეხსიერებაში ე.წ. **ღია (გახსნილი) ფაილების ცხრილში** შესაბამისი ჩანაწერის გაკეთების გზით. ფაილის გახსნის ოპერაცია აბრუნებს არაუარყოფით რიცხვით მნიშვნელობას - **ფაილურ დესკრიპტორს**, რომელიც წარმოადგენს ღია ფაილების

ცხრილში შესაბამის ფაილზე ჩანაწერის ნომერს. ფაილზე განსახორციელებელი ყოველი შემდეგი მოქმედება ხორციელდება ფაილური დესკრიპტორის გამოყენებით.

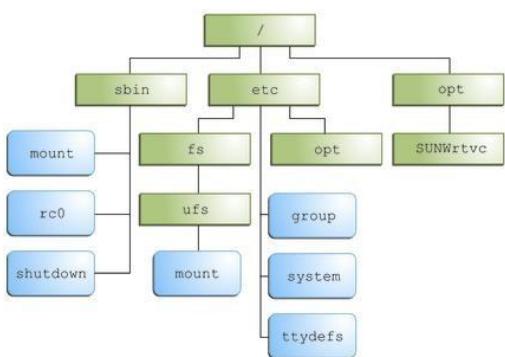
ღია ფაილების ცხრილი ასევე შეიცავს ფაილის მართვის ველებს. ამ ველებში ინახება ფაილის სამართავად გამოყენებადი ინფორმაცია ე.წ. ფაილის ატრიბუტები. ფაილის ატრიბუტები შეიძლება განსხვავდებოდნენ სისტემიდან სისტემამდე. მაგალითად, ის შეიძლება შეიცავდეს შემდეგ ინფორმაციას:

- ფაილის სიმბოლური სახელი;
- ფაილის ადგილმდებარეობა შესანახ მოწყობილობაზე;
- საორგანიზაციო სტრუქტურა (მაგალითად, მიმდევრობითი ან ნებისმიერი დაშვების ფაილი);
- ინფორმაცია შესანახ მოწყობილობაზე;
- ინფორმაცია ფაილის ტიპზ;
- ინფორმაცია ფაილის ხასიათზე (დროებითი თუ მუდმივი);
- ფაილის შექმნის დრო და თარიღი;
- ფაილზე მიმართვების მთვლელი (მაგალითად, ფაილზე განხორციელებული კითხვის ოპერაციების რაოდენობა);

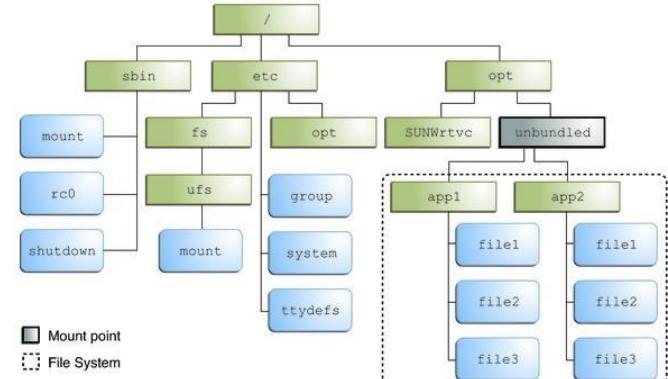
ჩვეულებრივ ფაილის სამართავი მონაცემები ინახება შესანახ მოწყობილობაზე. ფაილის გახსნის ოპერაციის განხორციელებისას ეს მონაცემები შესანახი მოწყობილობიდან სისტემას გადააქვს ოპერატიულ მეხსიერებაში.

9.3.8. ფაილური სისტემის მიერთება (mounting file system)

ხშირად მომხმარებელი საჭიროებს ისეთ მონაცემებზე დაშვებას, რომლებიც არ წარმოადგენენ მიმდინარე (ოპერაციული სისტემის) ფაილური სისტემის ნაწილს (მაგალითად, DVD დისკზე განთავსებული მონაცემები, USB მოწყობილობის ფაილური სისტემა). ამიტომ ფაილურ სისტემაში გათვალისწინებულია სხვა ფაილური სისტემის ძირითადზე მიერთების შესაძლებლობა. ფაილური სისტემის მიერთების ოპერაცია სხვადსხვა ფაილურ სისტემებს მოაქცევს ერთიანი სახელების სივრცეში - ფაილების ნაკრებში, რომლებზეც მიმართვა შეუძლია ყველა ფაილურ სისტემას. გაერთიანებული სივრცე მომხმარებელს საშუალებას აძლევს გამოიყენოს სხვადასხვა მოწყობილობაზე განთავსებული ინფორმაცია, თითქოს ისინი წარმოადგენენ მიმდინარე ფაილური სისტემის ნაწილს.



ა) ფაილური სისტემა ახალი ფაილური მიერთებამდე



ბ) ფაილური სისტემა ახალი ფაილური სისტემის სისტემის მიერთების შემდეგ

ნახ. 9.3. ფაილური სისტემის მიერთების ოპერაცია

ფაილური სისტემის მიერთების ბრძანება მიმდინარე ფაილურ სისტემაში ახალი ფაილური სისტემისთვის გამოყოფს დირექტორიას ე.წ. მიერთების წერტილს (დირექტორიას). Windows

ოპერაციული სისტემის ადრეულ ვერსიებში მიერთების ოპერაცია იყო ბრტყელი, რაც ნიშნავს, რომ ახალ ფაილურ სისტემას სახელად ენიჭებოდა ლათინური ალფავიტის ასოები და ისინი თავსდებოდნენ სტრუქტურის ერთსადაიმავე დონეზე.

UNIX-მსგავს ოპერაციული სისტემებისა და Windows NTFS 5.0-ის შემდეგი ვერსიების ფაილური სისტემები იძლევიან მიმდინარე ფაილურ სისტემაში ახალი ფაილური სისტემის ნების-მიერ ადგილას განთავსების შესაძლებლობას. მიერთების დირექტორიაში არსებული მიმდინარე ფაილური სისტემის მონაცემები ამ დირექტორიაში ახალი ფაილური სისტემის არსებობის განმავლობაში იქნება დამალული. მისი მოხსნის (սპოუნ) შემდეგ კი ისევ იქცევიან ხილულად.

ფაილური სისტემა მიერთების დირექტორიებს მართავს ე.წ მიერთების ცხრილის მეშვეობით. ამ ცხრილში ინახება ინფორმაცია მიერთების წერტილის მისამართზე და მოწყობილობაზე, რომელზეც ისაა განთავსებული. მიმდინარე ფაილური სისტემა ახალი ფაილური სისტემიდან მონაცემების კითხვის საჭიროებისას მიერთების დირექტორიამდე იყენებს ამ დირექტორიამდე სრულ მისამართს. მიერთების დირექტორიიდან დანიშნულების ფაილამდე გზის გასარკვევად ის მიმართავს მიერთების ცხრილს, საიდანაც არკვევს მოწყობილობის და მიერთებული ფაილური სისტემის ტიპს და ახალ ფაილურ სისტემაში ფაილის სრულ მისამართს, რომელსაც ამატებს მიერთების დირექტორიამდე სრულ მისამართს.

ახალი ფაილური სისტემის მოხსნის ბრძანებას მიმდინარე ფაილური სისტემიდან ამოაქვს მიერთებული ფაილური სისტემის მონაცემები და მიერთების ცხრილიდან იშლება შესაბამისი ჩანაწერი.

9.4. ფაილების ორგანიზაცია

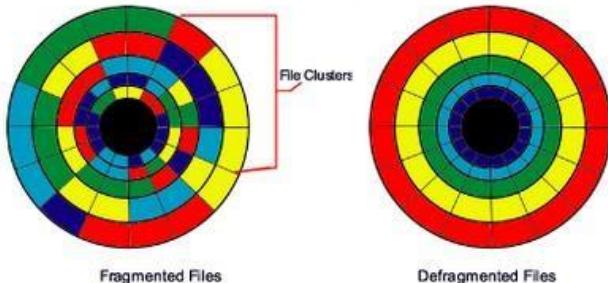
ფაილების ორგანიზაცია ეწოდება შესანახ მოწყობილობაზე მათი განთავსების წესს. ფაილების ორგანიზაციის არსებობს შემდეგი მეთოდები:

- **მიმდევრობითი (sequential)** - შესანახ მოწყობილობაზე ჩანაწერის განთავსება ხორციელდება მათი ფიზიკური მიმდევრობით. ასეთი ორგანიზაცია დამახასიათებელია მაგნიტურლენტაზე შენახული ფაილებისთვის, ანუ მიმდევრობითი დაშვების მოწყობილობებისთვის;
- **პირდაპირი (direct)** - პირდაპირი წვდომის შესანახ მოწყობილობაზე ჩანაწერებზე დაშვება შესაძლებელია მათი ფიზიკური ადგილმდებარეობის მიხედვით. გამოყენებითი პროგრამა ჩანაწერებს შესანახ მოწყობილობაზე ანთავსებს იმ მიმდევრობით, რომელიც მისთვისაა მოსახერხებელი;
- **ინდექსირებული მიმდევრობა (indexed sequential)** - შესანახ მოწყობილობაზე ჩანაწერები თავსდება ლოგიკური მიმდევრობით, გასაღების გამოყენებით, რომელიც ინახება თითოეულ ჩანაწერში. სისტემაში მხარდაჭერილია ინდექსები, რომლებიც უთითებენ გარკვეულ ძირითად ჩანაწერებს. ინდექსირებული მიმდევრობის ჩანაწერებზე დაშვება შესაძლებელია მათი გასაღებების მიმდევრობით, ან პირდაპირ სისტემაში შექმნილი ინდექსების მეშვეობით;
- **დაყოფილი (partitioned)** - ასეთი ტიპის ორგანიზაციისას ფაილი პრაქტიკულად იყოფა რამდენიმე ქვეფაილად. თითოეული ქვეფაილის მისამართი ინახება ფაილის დირექტორიაში. დაყოფილი ფაილები გამოიყენება პროგრამების ბიბლიოთეკების ან მაკროსების შესანახად (ამ მიზეზით ორგანიზაციის ასეთ ტიპს ბიბლიოთეკურსაც უწოდებენ);

9.5. ფაილების განთავსება

შესანახ მოწყობილობაზე ფაილების განთავსების და დაკავებული სივრცის გამოთავისუფლების პრობლემა გარკვეულწილად წააგავს მრავალპროგრამულ გარემოში მეხსიერების მართვის პრობლემას. ფაილების შექმნისა და წაშლის სიხშირის მიხედვით შესანახი მოწყობილობა უფრო და უფრო ხდება ფრაგმენტირებული, ხოლო ფაილების მონაცემები კი შესანახ მოწყობილობაზე განთავსებულია არამეზობელ ბლოკებში (მიმოფანტულია) (ნახ. 9.4). ამან შესაძლებელია ზეგავლენა იქონიოს ოპერაციული სისტემის წარმადობაზე.

File clusters, fragmented files, defragmented files



ნახ. 9.4. ფაილური სისტემის მაგალითი ფრაგმენტაციამდე და ფრაგმენტაციის შემდეგ

ვინაიდან პროცესები ხშირად მიმართავენ ფაილის მიმდევრობით ნაწილებს, ამიტომ უმჯობესი იქნება, თუ ფაილის მონაცემები განთავსდება უწყვეტი მიმდევრობით, რაც გაზრდის მათზე დაშვების და ზოგადად ოპერაციული სისტემის მუშაობის სისწრაფეს. მაგალითად, ოპერაციული სისტემა ფაილში გარკვეული ინფორმაციის ძებნისას ხშირად იყენებს სკანირების ოპერაციას რათა საჭიროების შემთხვევაში შეეძლოს წინა ან მომდევნო ჩანაწერზე გადასვლა. სკანირების ოპერაცია უნდა ხორციელდებოდეს პოზიციონირების აუცილებელი ოპერაციის მინიმალური დროითი დანახარჯით.

ვინაიდან ხშირ შემთხვევებში ფაილის მოცულობა იცვლება დროის მიხედვით, ხოლო მომხმარებლისთვის კი წინასწარ უცნობია ფაილის სავარაუდო მოცულობა, ამიტომ ფაილისთვის უწყვეტი მისამართების სივრცის გამოყოფის სისტემები იცვლება უფრო დინამიური ფრაგმენტირებული მისამართების სივრცის გამოყოფის სისტემებით.

9.5.1. ფაილების განთავსება უწყვეტი მიმდევრობით

ამ ტიპის ფაილურ სისტემაში შენახვის მოწყობილობაზე ფაილების შენახვა ხორციელდება მისამართების უწყვეტი მიმდევრობით. მომხმარებელი, რომელიც საჭიროებს ახალი ფაილის შექმნას წინასწარ უთითებს ფაილის სავარაუდო მოცულობას. თუ შესანახ მოწყობილობაზე ფაილის განსათავსებლად ვერ მოიძებნა საჭირო მოცულობის მისამართების უწყვეტი მიმდევრობა, მაშინ ფაილი არ შეიქმნება.

ფაილების უწყვეტი მიმდევრობით განთავსების მეთოდის უპირატესობა მდგომარეობს იმაში, რომ ერთიმეორის მიყოლებით ლოგიკურად განთავსებული ჩანაწერები შესანახ მოწყობილობაზე ფიზიკური მისამართებითაც განთავდებიან იმავე მიმდევრობით. შესაბამისად, მათზე დაშვება უფრო სწრაფია ვიდრე სისტემებში, რომლებიც მონაცემების განთავსებას ახორციელებენ ფრაგმენტირებული ფიზიკური მისამართების სივრცეებში. ასეთ სისტემებში მონაცემების მოძიებაც გამარტივებულია, ვინაიდან დირექტორიებში ინახება ინფორმაცია ფაილის მხოლოდ დასაწყისის და დასასრულის მისამართებზე.

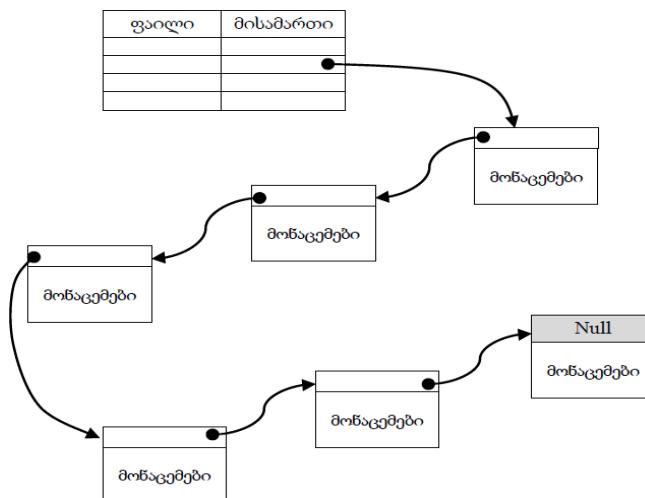
ასეთი სისტემების ნაკლოვანება კი მდგომარეობს გარე ფრაგმენტაციის წარმოშობაში. ამასთან, თუ ფაილის მოცულობა გაიზრდება და მისთვის ადრეულ ეტაპზე გამოყოფილი სივრცე არაა

საკმარისი მის უწყეტი მისამართებით განსათავსებლად, მაშინ საჭირო იქნება ფაილისთვის საკმარისი მოცულობის უწყეტი მისამართების სივრცის მოძიება. შესაფერისი სივრცის მოძიების შემდგომ ფაილის ახალ სივრცეში გადასატანად საჭირო იქნება დიდი რაოდენობით შეტანა/გამოტანის ოპერაციის განხორციელება, რაც გამოიწვევს ოპერაციული სისტემის წარმადობის შენელებას. შეიძლება ჩავთვალოთ, რომ ამ პრობლემისთვის გვერდის ავლა შესაძლებელია თავიდანვე ფაილისთვის დიდი მოცულობის მითითებით, მაგრამ ამ შემთხვევაში აღმოჩნდება, რომ მისამართების სივრცე არასწორადაა გადანაწილებული.

9.5.2. ფაილის განთავსება დაკავშირებული სიებით

უმეტეს ფაილურ სისტემაში ფაილების განთავსება რეალიზებულია მისამართების ფრაგმენტი-რებული სივრცეებით. ფრაგმენტირებული სივრცეებით განთავსების ერთერთ რეალიზაციას წარმოადგენს დაკავშირებული სექტორების ჩამონათვალის გამოყენება. ასეთი რეალიზაციისას დირექტორიაში არსებული ყოველი ჩანაწერი უთითებს შემნახველ მოწყობილობაზე ფაილის მიერ დაკავებულ პირველივე სექტორზე. სექტორის დანაყოფები შეიცავს ფაილის მონაცემებს. ვინაიდან ფაილის მონაცემები შესაძლებელია განთავსებული იყოს რამდენიმე სექტორში, ამიტომ კითხვა/ჩაწერის ოპერაციის განხორციელებისას დისკის წამკითხავმა თავაკმა შესაბამისი ჩანაწერის მოძიების მიზნით უნდა განახორციელოს მიმართვა ფაილის მიერ დაკავებულ ყველა სექტორზე.

ფაილების ფრაგმენტირებული განთავსების სისტემა აგვარებს ფაილების უწყეტი განთავსების სისტემისთვის დამახასიათებელ პრობლემას, მაგრამ მას გააჩნია საკუთარი ნაკლოვანებები. ვინაიდან ასეთი ორგანიზაციისას ფაილის ჩანაწერები შესანახ მოწყობილობაზე შესაძლებელია განთავსებული იყოს არამეზობელ სექტორებში, ამიტომ ჩანაწერებზე პირდაპირი და მიმდევრობითი ოპერაციების განხორციელებას შესაძლებელია დასჭირდეს პოზიციონირების მრავალი ოპერაცია. გარდა ამისა სექტორების ჩამონათვალში მიმთითებლების დამახსოვრების აუცილებლობა ამცირებს თითოეულ სექტორში შესანახი მონაცემების მოცულობას.



ნახ. 9.5 ფაილების განთავსება დაკავშირებული სიებით

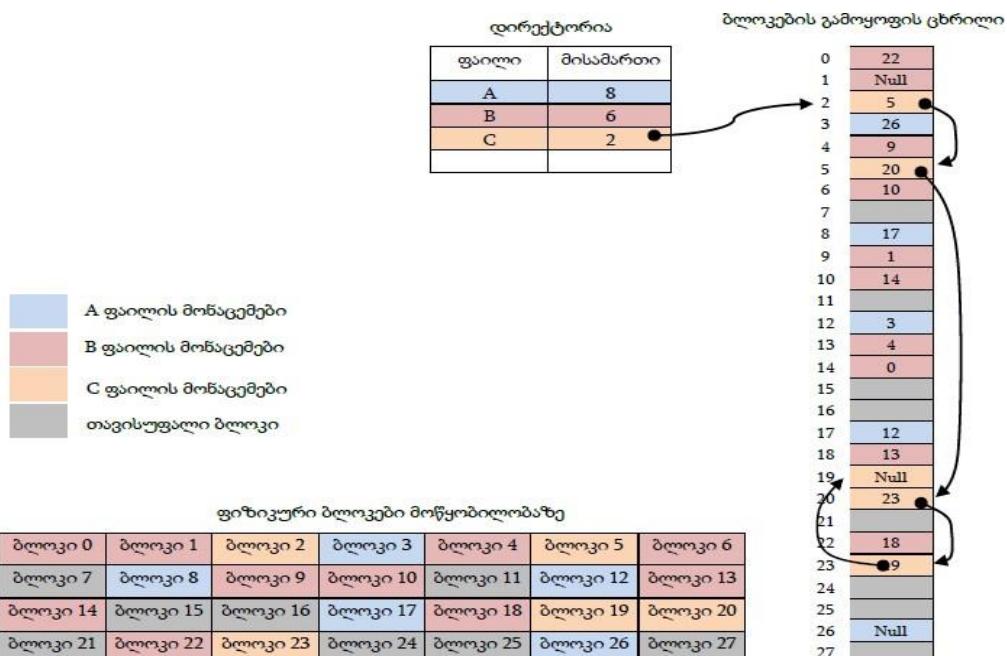
ერთერთი სქემა, რომელიც გამოიყენება შესანახ მოწყობილობასთან მუშაობის ეფექტურობის გაზრდისა და ფაილებზე მიმართვებისას ზედნადები დანახარჯების შესამცირებლად არის ბლოკური განთავსების სქემა. ამ სქემის მიხედვით მონაცემების შენახვისთვის ცალკეული სექტორის გამოყოფის ნაცვლად ხდება სექტორების გარკვეული უწყეტი მიმდევრობის, ბლოკების გამოყოფა. სისტემა ფაილების მონაცემების განსათავსებლად გამოყოფს მაქსიმალურად ახლოს მყოფ ბლოკებს. ფაილის მონაცემებზე ყოველი მიმართვა საჭიროებს შესაბამისი ბლოკის და ბლოკის შიგნით სექტორის ცოდნას. ბლოკების დაკავშირებისას (გადაბმისას) დირექტორიაში განთავსებული

ყოველი ჩანაწერი უთითებს ფაილის მიერ დაკავებულ პირველ ბლოკზე (ნახ. 9.5). ყოველი ფაილის თითოეული ბლოკი შედგება ორი დანაყოფისგან: მონაცემები და შემდეგ ბლოკზე მიმთითებელი. მინიმალურ ერთეული რომლის გამოყოფაც შესაძლებელია ფაილის მონაცემების შესანახად არის ერთი ბლოკი. საჭირო ჩანაწერის მოსამებნად აუცილებელია ბლოკების სკანირება პირველიდან იმ ბლოკამდე, რომელშიც აღმოჩნდება ჩანაწერი. ბლოკების არამეზობლად განთავსებისას შესაბამისი ჩანაწერის მოძებნა რთულდება, რაც აისახება სისტემის წარმადობაზე. ბლოკებში ჩანაწერის ჩამატების ან წამლის ოპერაციები დაიყვანება ბლოკის შიგნით მიმთითებლის შეცვლამდე.

ბლოკის მოცულობამ შესაძლებელია არსებითი ზეგავლენა იქონიოს სისტემის წარმადობაზე. დიდი მოცულობის ბლოკებმა შეიძლება მიგვიყვანოს საგრძნობ შიდა ფრაგმენტაციამდე. თუმცა დიდი ბლოკების შემთხვევაში ფაილზე დაშვების შეტანა/გამოტანის ოპერაციების რაოდენობა მცირდება. მცირე ზომის ბლოკების შემთხვევაში კი შეიძლება აღმოჩნდეს, რომ ფაილი განთავსებულია საკმარისად დაშორებულ ბლოკებში, რომლებზეც მიმართვა შესაძლებელია აისახოს სისტემის წარმადობაზე. პრაქტიკაში გამოიყენება ბლოკები მოცულობით 1 – 8 კბ. თანამედროვე ოპერაციულ სისტემებში გამოიყენება 4 კბ ზომის ბლოკები.

9.5.3. ცხრილური ფრაგმენტირებული განთავსება

ფაილური სისტემა, რომელშიც ფაილების განთავსება რეალიზებულია ცხრილური ფრაგმენტირებული ფორმით საჭირო მონაცემებზე დაშვებისთვის, პოზიციონირების ოპერაციათა რაოდენობის შემცირების მიზნით, ინახავს მიმთითებელს ფაილის ბლოკებზე (ნახ. 9.6). მაგალითად, C ფაილის პირველი ბლოკის ნომერი არის 2. ბლოკების გამოყოფის ცხრილის შესაბამის ბლოკში (2) არსებული რიცხვი (5) გამოიყენება შემდეგი ბლოკის მისათითებლად და ა.შ. ცხრილში ჩანაწერი, რომელიც შეესაბამება ფაილის მიერ დაკავებულ ბოლო ბლოკს შეიცავს მნიშვნელობას Null.



ნახ. 9.6. ფაილების ფრაგმენტირებულ ცხრილური განთავსება

ვინაიდან მიმთითებლები, რომლებშიც განთავსებულია მონაცემები, ინახება ცენტრალიზებულად, ამიტომ შესაძლებელია მათი ჩატვირთვა ქეშ-მეხსიერებაში და სწრაფად განსაზღვრა იმ ბლოკების ჯაჭვისა, რომლებშიც ინახება მონაცემები. ბოლო ჩანაწერის მოძებნა საჭიროებს ყველა მიმთითებლის დათვალიერებას ბლოკების გამოყოფის ცხრილში. დაშვების დროის შემცირების მიზნით ეს ცხრილი დისკზე უნდა ინახებოდეს უწყვეტი მიმდევრობით და იყოს ქეშირებული

ოპერატიულ მექსიერებაში. თუ ფაილური სისტემა შედგება ბლოკების მცირე რაოდენობისგან, მაშინ ამის გაკეთება არ იქნება რთული. მაგალითად, დისკება მოცულობით 1,44 მბ 1 კბ მოცულობის ბლოკებად დაყოფის შემთხვევაში შეიცვას 1440 ბლოკს, რომლებზეც მიმართვა შესაძლებელია 11 ბიტიანი ინდექსებით ($1440 < 2^{11} = 2048$). შესაბამისი ცხრილის მოცულობა ტოლია ბლოკების რაოდენობა გამრავლებული ერთი ბლოკის ინდექსის სიგრძეზე - განხილულ შემთხვევაში დაახლოებით 2000 ბაიტი.

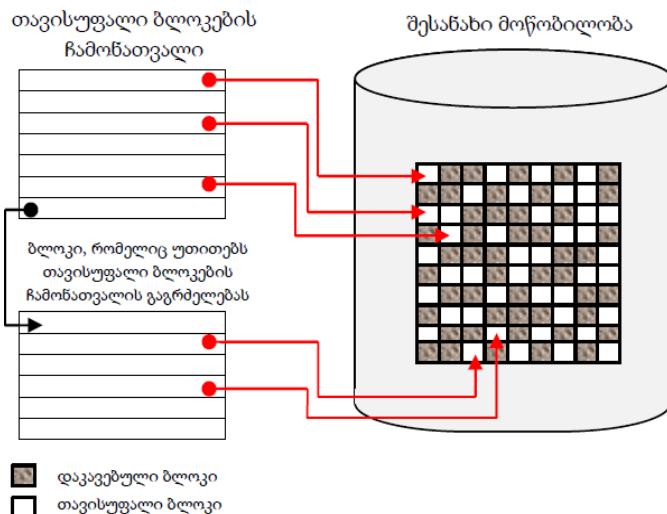
დიდი რაოდენობის ბლოკების შემცველი ფაილური სისტემისთვის ბლოკების განთავსების ცხრილის და ბლოკების ცხრილის მოცულობა იქნება საკმაოდ დიდი. მაგალითად, 200 გბ დისკი დაყოფილი 4 კბ ბლოკებად შეიცვას 50000000 ბლოკს, რომელთაგან თითოეულზე წვდომა შესაძლებელი იქნება 26 ბიტიანი ინდექსით. ამ შემთხვევაში ბლოკების განთავსების ცხრილის მოცულობა იქნება 160 მბ-ზე მეტი. წინა მაგალითში ბლოკების გამოყოფის ცხრილი იკავებდა მხოლოდ 2 ბლოკს ($2000 < 2 \text{ კბ}$). ამ შემთხვევაში ის დაიკავებს რამდენიმე ათას ბლოკს, რამაც შესაძლებელია მიგვიყვანოს ფრაგმენტაციამდე. თუ ფაილის ბლოკები იქნება მიმოფანტული შესანახ მოწყობილობაზე, მაშინ ამ ბლოკებზე ცხრილის ჩანაწერებიც იქნება მიმოფანტული. ამიტომ სისტემას შეიძლება დასჭირდეს ცხრილის რამდენიმე ბლოკის ჩატვირთვა მეხსიერებაში, რამაც შესაძლებელია საგრძნობლად შეამციროს სისტემის რეაქციის დრო. გარდა ამისა ასეთი დიდი ცხრილის ქეშირებამ შესაძლებელია დაიკავოს ოპერატიული მეხსიერების მნიშვნელოვანინაწილი.

9.6. თავისუფალი სივრცის მართვა

დროის მიმდინარეობასთან ერთად შესაძლებელია იცვლებოდეს ფაილის მოცულობა და ოპერაციულმა სისტემამ ფაილის მონაცემების შესანახად უნდა გამოყოს თავისუფალი ბლოკი. ოპერაციული სისტემა მიმდინარე დროში თავისუფალ და დაკავებულ ბლოკებზე ინფორმაციის სამართვად იყენებს თავისუფალი ბლოკების ჩამონათვალს (სიას) (დაკავშირებული ბლოკების ჩამონათვალი, რომელშიც მითითებულია ბლოკის მისამართი ფიზიკურ მოწყობილობაზე). თითოეულ ბლოკში ბოლო ჩანაწერი უთითებს ჩამონათვალის შემდეგ ბლოკზე. ბოლო ბლოკში ბოლო ჩანაწერი უთითებს Null-მიმთითებელზე იმის აღსანიშნავად, რომ ის არის ბოლო თავისუფალი ბლოკების ჩამონათვალში. როდესაც სისტემა საჭიროებს ფაილისთვის ახალი ბლოკის გამოყოფას ის მიმართავს თავისუფალი ბლოკების ჩამონათვალს, ირჩევს პირველივე თავისუფალ ბლოკს, პოულობს მის მისამართს შესანახ მოწყობილობაზე და იქ ანთასებს მონაცემებს, ხოლო თავისუფალი ბლოკების ჩამონათვალიდან შლის ბლოკზე შესაბამის ჩანაწერს.

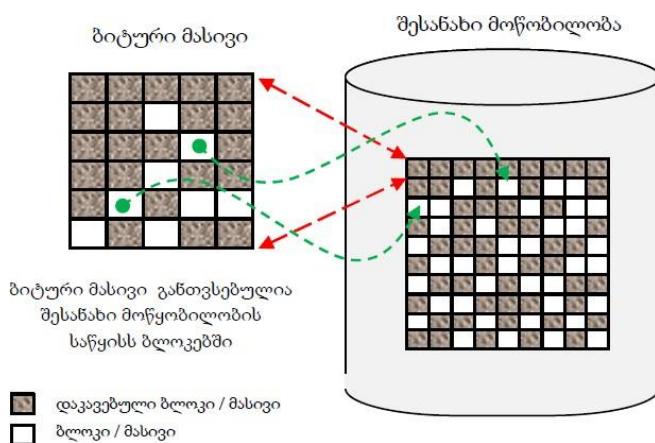
ჩვეულებრივ ფაილური სისტემა მონაცემების განსათავსებლად, თავისუფალი ბლოკების ჩამონათვალიდან გამოყოფს, სათავესთან ახლოს განთავსებულ ბლოკს, ხოლო გამოთავისუფლებულ ბლოკს ანთავსებს ამ ჩამონათვალის ბოლოში. ჩამონათვალის დასაწყისზე და დასასრულზე მიმთითებელი შესაძლებელია ინახებოდეს ფაილური სისტემის სუპერბლოკში. თავისუფალი ბლოკის მოძებნა შესაძლებელია მიმთითებლის ერთი გადასვლით. ახალი ბლოკის დამატება ასევე მოითხოვს ჩამონათვალში ახალი მიმთითებლის ჩამატებას. ამიტომ ასეთი მიდგომა თავისუფალი ბლოკების ჩამონათვალის მხარდაჭერისთვის არ ითხოვს დამატებით ზედნადებ დანახარჯებს. თუმცა ფაილების შექმნის და წაშლის მაღალი ინტენსივობის შემდგომ შესანახი მოწყობილობა უფრო და უფრო იქნება ფრაგმენტირებული და ჩამონათვალის მეზობელი ჩანაწერები შესაძლებელია არ უთითებდნენ მეზობელ ბლოკებზე, ანუ ბლოკები იყვნენ საკმარისად დაშორებული ერთიმეორისგან, რაც გაზრდის მონაცემებზე დაშვების დროს. ამ მიდგომის ალტერნატივად სისტემამ შესაძლებელია განახორციელოს უწყვეტი ბლოკების მოძიება თავისუფალი ბლოკების

ჩამონათვალში ან განახორციელოს ჩამონათვალის დახარისხება, მაგრამ ორივე მიღებული მოითხოვს საკმაო ზედნადებ დანახარჯებს.



ნახ. 9.7. სივრცის მართვა თავისუფალი ბლოკების ჩამონათვალის გამოყენებით

თავისუფალი ბლოკების მართვის კიდევ ერთ მეთოდს წარმოადგენს ბიტური მასივის გამოყენება (ნახ. 9.8). მასივში ყოველი ბიტი შესაბამება შემნახველ მოწყობილობაზე ერთ ბლოკს. თუ ბიტის მნიშვნელობა 1-ის ტოლია, მაშინ მოწყობილობაზე შესაბამისი ბლოკი დაკავებულია. წინააღმდეგ შემთხვევაში ის თავისუფალია ახალი მონაცემების ჩასაწერად. ბიტური მასივი ჩვეულებრივ იკავებს რამდენიმე ბლოკს. ბიტური მასივის გამოყენების ერთერთი უპირტესობა თავისუფალი ბლოკების ჩამონათვალის გამოყენებასთან მიმართებაში მდგომარეობს იმაში, რომ ფაილურ სისტემას სწრაფად შეუძლია განსაზღვროს ბლოკების მიმდევრობა ქმნის თუ არა მისამართების უწყვეტ სივრცეს. მისი ნაკლოვანება კი მდგომარეობს იმაში, რომ თავისუფალი ბლოკების მოსაძებნად საჭიროა მასივის მთლიანი დათვალიერება. ასეთი დათვალიერება დაკავშირებულია ზედნადებ დანახარჯებთან, მაგრამ რიგ შემთხვევებში შესაძლებელია მათი უგულვებელყოფაც (თანამედროვე პროცესორების სისწრაფე გაცილებით აღემატება შეტანა/გამოტანის მოწყობილობის სისწრაფეს).



ნახ. 9.8. თავისუფალი სივრცის მართვა ბიტური მასივის გამოყენებით

9.7. ფაილზე დაშვების კონტროლი

ხშირად მომხმარებელი საკუთარ კომპიუტერში ინახავს მნიშვნელოვან ინფორმაციას (მაგალითად, საკრედიტო ბარათის ნომერი, ფინანსური მონაცემები, პაროლები და ა.შ.) და ფაილურ

სისტემას უნდა გააჩნდეს მექანიზმები, რომლითაც უზრუნველყოფს ფაილის მონაცემებზე დაშვების კონტროლს. ამ მიზნით ოპერაციული სისტემა აღჭურვილია ფაილზე დაშვების სხვადასხვა მეთოდებით. აღვწეროთ ზოგიერთი მათგანი.

9.7.1. დაშვების ბიტური მატრიცა

ფაილზე დაშვების მართვის ერთერთ მეთოდს წარმოადგენს დაშვების კონტროლისთვის ორგანზომილებიანი მასივის შექმნა (ნახ. 9.9), რომელშიც ჩამოთვლილია სისტემაში არსებული ყველა მომხმარებლი და ფაილი. მატრიცის a_{ij} ელემენტი განსაზღვრავს მომხმარებელს გააჩნია თუ არა შესაბამის ფაილზე დაშვება. თუ $a_{ij} = 1$, მაშინ მომხმარებელი სარგებლობს ფაილზე დაშვების შესაბამისი უფლებით. წინააღმდეგ შემთხვევაში ($a_{ij} = 0$) მას ეს უფლება არ გააჩნია. სისტემაში მომხმარებლების და ფაილების დიდი რაოდენობის შემთხვევაში იზრდება მატრიცის მოცულობა და რთულდება მისი მართვა.

მომხმარებელი	ფაილი									
	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	0	0	1
4	1	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	1	1	1	1	0	0	0	0	1	1
10	1	1	0	0	1	1	0	0	0	1

ნახ. 9.9. დაშვების კონტროლის მატრიცა

შევნიშნოთ, რომ ასეთი მატრიცის გამოყენებისას მომხმარებელს ფაილზე გააჩნია სრული დაშვება ან დაშვება საერთოდ არ გააჩნია. ასეთი მატრიცული კონცეფცია რომ იყოს გამოყენებადი პრაქტიკული თვალსაზრისით მასში უნდა იყოს ასახული დაშვების განსხვავებული უფლებების აღმნიშვნელი კოდები - მხოლოდ კითხვა, მხოლოდ რედაქტირება, მხოლოდ შესრულება და მათი კომბინაციები, რაც რათქმაუნდა კიდევ უფრო გაართულებს მის მართვას.

9.7.2. დაშვების უფლებების მართვა მომხმარებელთა კლასების მიხედვით

არსებობს მეთოდი, რომელიც შესანახ მოწყობილობაზე ითხოვს ნაკლები მოცულობის დისკურსივრცეს ვიდრე დაშვების კონტროლის მატრიცა. ეს მეთოდი ეყრდნობა მომხმარებელთა კლასების ცნებას. მომხმარებელთა კლასიფიკაცია ფაილზე დაშვების უფლებების მიხედვით ხორციელდება შემდეგნაირად:

- **მფლობელი (owner)** - ფაილის შემქმნელი მომხმარებელი. ამ ტიპის მომხმარებელს ფაილზე გააჩნია განუსაზღვრელი უფლებები. მას შეუძლია საკუთარ ფაილზე განსაზღვროს სხვა მომხმარებლების დაშვების უფლებები;
- **განსაზღვრული მომხმარებელი (specified user)** - მომხმარებელი, რომელსაც ფაილის მფლობელმა მიანიჭა დაშვების უფლებების გარკვეული პრივილეგიები;
- **ჯგუფი (group)** ან **პროექტი (project)** - მომხმარებლები ხშირად მიეკუთვნებიან საერთო პროექტზე მომუშავე ადამიანთა ჯგუფს. ამ შემთხვევაში ჯგუფის ყველა წევრს შეიძლება გააჩნდეს პროექტთან დაკავშირებულ სხვა მომხმარებლების მონაცემებზე დაშვების სრული უფლებები;

- **საყოველთაო (public)** - მრავალი სისტემა იძლევა საყოველთაო ფაილის შექმნის შესაძლებლობას, რომელზეც დაშვება შეეძლება სისტემაში შემოსულ ნებისმიერ მომხმარებელს. გაჩუმების წესით, ჩვეულებრივ ასეთ ფაილებზე შესაძლებელია კითხვის და შესრულების ოპერაციების განხორციელება.

დაშვების კონტროლის მონაცემები შესაძლებელია ინახებოდეს ფაილის მართვის ბლოკის ნაწილის სახით და იკავებდეს დისკურსი სივრცის უმნიშვნელო ნაწილს.

9.7.3. მონაცემებზე დაშვების მეთოდები

რამდენიმე პროცესი შესაძლებელია ერთდროულად ითხოვდეს შესანახ მოწყობილობის სხვადასხვა ბლოკებში განთავსებულ მონაცემებს და მოთხოვნების დასაკმაყოფილებლად უნდა შესრულდეს პოზიციონირების ნელი ოპერაციები. იმისთვის, რომ სწრაფად განხორციელდეს შეტანა/გამოტანაზე შემოსული მოთხოვნების დაკმაყოფილება ოპერაციულ სისტემაში შესაძლებელია გამოყენებული იქნას სხვადასხვა მეთოდები.

თანამედროვე ოპერაციულ სისტემებში გამოყენებულია დაშვების სხადასხვა მეთოდები. ხშირად ამ მეთოდებს ყოფენ ორ ნაწილად: **დაშვების საბაზო მეთოდები (basic access methods)** და **მიმდევრობითი დაშვების მეთოდები (queued access methods)**.

მიმდევრობითი დაშვების მეთოდები სარგებლობს მეტი უპირატესობით ვიდრე საბაზო დაშვების მეთოდები. მიმდევრობითი დაშვების მეთოდები გამოიყენება იმ შემთხვევაში, როდესაც წინასწარ შესაძლებელია განისაზღვროს თუ რა მიმდევრობით განხორციელდება ჩანაწერების დამუშავება. ეს მეთოდები ახორციელებენ ჩანაწერების წინასწარ ბუფერირებას და შეტანა/გამოტანის ოპერაციების დაგეგმვას. ბუფერირების გზით ცდილობენ მიმდინარე ჩანაწერის დამუშავების დასრულებამდე გამზადებული იყოს შემდეგი დასამუშავებელი ჩანაწერი. ოპერატიულ მეხსიერებაში ერთდროულად ინახება რამდენიმე ჩანაწერი რაც იძლევა შეტანა/გამოტანის ოპერაციების გაერთიანების და ამით სისტემის წარმადობის ამაღლების საშუალებას.

დაშვების საბაზო მეთოდები გამოიყენებიან იმ შემთხვევაში, როდესაც წინასწარ შეუძლებელია იმის განსაზღვრა თუ როგორი მიმდევრობით განხორციელდება ჩანაწერების დამუშავება, განსაკუთრებით ნებისმიერი მიმდევრობით დაშვების დროს. გარდა ამისა, არსებობს მრავალი შემთხვევა (მაგალითად, მონაცემთა ბაზების პროგრამებში), რომლის დროსაც გამოყენებითი პროგრამები ითხოვენ ჩანაწერებზე დაშვების კონტროლს წინასწარ ბუფერირების გარეშე. დაშვების საბაზო მეთოდები ჩანაწერების კითხვასა და რედაქტირებას ახორციელებენ შესანახ მოწყობილობაზე ფიზიკური ბლოკებით. ჩანაწერების გაერთიანება ბლოკებად და ფიზიკური ბლოკების დაყოფა ჩანაწერებად (საჭიროების შემთხვევაში) ხორციელდება შესაბამისი პროგრამის მიერ.

9.8. მონაცემების მთლიანობის დაცვა

როგორც უკვე აღვნიშნეთ მომხმარებელი კომპიუტერში ინახავს მისთვის მნიშვნელოვან მონაცემებს. სისტემის გაუმართავმა მუშაობამ, სტიქიურმა უბედურებამ ან არაკეთილმოსურნე ადამიანმა შეიძლება საფრთხე შეუქმნას ამ მონაცემებს. ასეთი მოვლენების შედეგი შეიძლება იყოს კრიტიკული ხასიათის მატარებელი. ოპერაციული სისტემები და მონაცემების შენახვის სისტემები უნდა იყვნენ მდგრადი მტყუნებამედეგობის მიმართ იმ თვალსაზრისით, რომ ითვალისწინებდნენ მსგავსი მოვლენის შესაძლებლობას და მომხმარებელს სთავაზობდეს მონაცემების დაკარგვის შემდგომ აღდგენის შესაძლებლობას.

9.8.1. მონაცემების სარეზერვო ასლები და აღდგენა

მრავალ ოპერაციულ სისტემაში ინფორმაციის დაკარგვის თავიდან აცილების მიზნით რეალიზებულია ინფორმაციის ჭარბი რაოდენობით სარეზერვო ასლების შექმნის და ინფორმაციის დაკარგვის შემდგომ ამ სარეზერვო ასლებით მათი აღდგენის საშუალებები. გარდა ამისა, ამ სარეზერვო ასლების შექმნის და მათი მეშვეობით შემდგომი აღდგენის მექანიზმებით შესაძლებელია სისტემის დაცვა მომხმარებლის მიერ განხორციელებული მოქმედებებისგან. დაცვის ფიზიკური საშუალებები ეს არის მონაცემების დაცვის პირველი დონე. ასეთი მეთოდები (მაგალითად, საკუტები და დაცვის სისტემები) იძლევიან ფიზიკური მოწყობილობის (კომპიუტერის) დაცვას არასანქცირებული დაშვებისგან. ვინაიდან ელექტრონურგიის უეცარმა გამორთვამ შესაძლებელია გამოიწვიოს ოპერატიულ მეხსიერებაში არსებული შეუნახავი ინფორმაციის დაკარგვა, ამიტომ სისტემა უნდა იყოს დაცული ელექტრონურგიის უეცარი გამორთვისგან, რაც გულისხმობს კომპიუტერის უზრუნველყოფას კვების უწყვეტი წყაროთი.

სტიქიურმა უბედურებებმა (მაგალითად, ხანძარმა, წყალდიდობამ და ა.შ.) შესაძლებელია გაანადგუროს მომხმარებლის მონაცემები. ამ მიზნით გარკვეული ორგანიზაციები ქმნიან სპეციალურ სერვერებს, რომლებიც განთავსებულია გეოგრაფიულად დაშორებულ ტერიტორიულ ერთეულებზე და მომხმარებელს სთავაზობს სერვერებზე საკუთარი მონაცემების შენახვის შესაძლებლობას.

მონაცემების დაკარგვის აცილების მიზნით ფართოდ გამოიყენება სარეზერვო ასლების პერიოდული შექმნის მეთოდი. შესანახი მოწყობილობის ფიზიკური სარეზერვო ასლის შექმნის ოპერაცია გულისხმობს მოწყობილობის სარეზერვო ასლის შექმნას ბიტურ დონეზე. რიგ შემთხვევაში ხორციელდება მხოლოდ დაკავებული ბლოკების კოპირება (სარეზერვო ასლის შექმნა). სისტემაში ფიზიკური სარეზერვო ასლის შექმნის ორგანიზება მარტივია, მაგრამ ის არ ინახავს ინფორმაციას ფაილური სისტემის ლოგიკურ სტრუქტურაზე. ფაილური სისტემის მონაცემები, სისტემის არქიტექტურაზე დამოკიდებულებით, შესაძლებელია ინახებოდეს სხვადასხვა ფორმატში. ფიზიკური სარეზერვო ასლებით განსხვავებული არქიტექტურის სისტემებში მონაცემების აღდგენა შეიძლება იყოს რთული. გარდა ამისა ვინაიდან ფიზიკური სარეზერვო ასლის შექმნა არ ითვალისწინებს ფაილური სისტემის ლოგიკურ სტრუქტურას, ამიტომ სარეზერვო ასლები უნდა შეიცავდნენ ფაილურ სისტემას სრულად.

ლოგიკური სარეზერვო ასლი ინახავს ფაილური სისტემის მონაცემებს და მის ლოგიკურ სტრუქტურას. ლოგიკური სარეზერვო ასლის შექმნის ოპერაცია იხედება დირექტორიის სტრუქტურაში და განსაზღვრავს თუ რომელი ფაილების ასლის შექმნაა საჭირო, ხოლო შემდეგ მას ანთავსებს სარეზერვო ასლის შენახვის მოწყობილობაზე (მაგალითად, მაგნიტური ლენტა, CD ან DVD) სტანდარტული, ხშირად შეკუმშული არქივის ფორმით. ვინაიდან სარეზერვო ასლები მონაცემებს ინახავნ სტანდარტული ფორმით დირექტორიების სტრუქტურის სახით, ამიტომ ისინი ოპერაციულ სისტემებს აძლევენ საკუთარი ფაილების სხვადასხვა ფორმატებით წაკითხვის და მონაცემების სარეზერვო ასლების მეშვეობით აღდგენის შესაძლებლობას. გარდა ამისა, ლოგიკური სარეზერვო ასლები იძლევიან ცალკეული ფაილების აღდგენის შესაძლებლობას, რაც საჭიროებს ფაილური სისტემის აღდგენისთვის საჭირო დროზე ნაკლებ დროს. ვინაიდან ლოგიკური სარეზერვო ასლები კითხულობენ ფაილური სისტემის მიერ წარმოდგენილ მონაცემებს, ამიტომ მას შეიძლება გამორჩეს გარკვეული მონაცემები (მაგალითად, დამალული ფაილები ან მეტა-მონაცემები), რომლებიც იქნებოდა შენახული ბიტური სარეზერვო ასლის შექმნისას. სტანდარტულ ფორმატში ფაილების შენახვა ზედნადები დანახარჯების გამო შესაძლებელია იყოს არაეფექტური, რომელიც დაკავშირებულია ფაილის მიმდინარე და არქივის ფორმატებს შორის გადაყვანასთან.

ინკრემენტული სარეზერვო ასლის შექმნა ეს არის ლოგიკური სარეზერვო ასლის შექმნა,

რომლის დროსაც ხორციელდება ფაილური სისტემის იმ მონაცემებისთვის სარეზერვო ასლების შექმნა, რომლებიც წინა სარეზერვო ასლის შემდეგ შეიქმნენ ან განიცადეს გარკვეული ცვლილება. სისტემას შეუძლია შეინახოს ინფორმაცია სარეზერვო ასლის შექმნის შემდეგ იმ ფაილებზე, რომლებიც წარმოიქმნენ ან განიცადეს ცვლილება და ეს მონაცემები პერიოდულად შეინახოს სარეზერვო ასლებში, რაც ითხოვს სრულ სარეზერვო ასლის შექმნაზე ნაკლებ დროს და რესურსს. პერიოდული ინკრემენტული სარეზერვო ასლის შექმნით შესაძლებელია მონაცემების დაკარგვის აცილება.

სემინარი 1. UNIX ოპერაციული სისტემა

სასწავლო კურსის მსვლელობისას სემინარული მეცადინეობის მასალა იღუსტრირებული იქნება UNIX ოპერაციული სისტემის ერთერთი ნაირსახეობის (Linux) მაგალითზე, თუმცა ჩვენი საუბარი არ შეეხება მხოლოდ ამ სისტემის თავისებურებებს.

Linux ოპერაციული სისტემის ბირთვი წარმოადგენს მონოლიტურ სისტემას. მისი ბირთვის კომპილაციისას შესაძლებელია ბირთვის მრავალი კომპონენტის (მოდულის) დინამიური ჩატვირთვა/ამოტვირთვა. ჩატვირთვის მომენტში მისი კოდი შესასრულებლად გადადის პრივილეგირებულ რეჟიმში და უკავშირდება ბირთვის დანარჩენ ნაწილებს. მოდულის შიგნით შესაძლებელია გამოყენებული იყოს ბირთვის მიერ ექსპორტირებული ნებისმიერი ფუნქცია.

1.1. სისტემური გამოძახება და ბიბლიოთეკა libc

როგორც უკვე აღვნიშნეთ, UNIX ოპერაციული სისტემის ძირითად და მუდმივად ფუნქციონირებად ნაწილს წარმოადგენს მისი ბირთვი. სხვა (სისტემური ან გამოყენებითი) პროგრამები ბირთვთან ურთიერთქმედებენ სისტემური გამოძახებების (system call) მეშვეობით, რომლებიც პროგრამებისთვის წარმოადგენენ ბირთვზე დაშვების წერტილებს. სისტემური გამოძახებები არის სპეციალურად შექმნილი პროცედურები, რომლებსაც სისტემური ან გამოყენებითი პროგრამები გარკვეული დროით გადაჰყავთ პრივილეგირებულ რეჟიმში. ამ რეჟიმში პროგრამები ღებულობენ წვდომას ისეთ რესურსებზე, რომლებიც მომხმარებლის რეჟიმში მათთვის მიუწვდომელი იყო.

სისტემური გამოძახების შესასრულებლად საჭირო მანქანური ბრძანებები განსხვავდება მანქანიდან მანქანამდე. ასევე, განსხვავდება სხვადასხვა ოპერაციულ სისტემაში გამოყენებული სისტემური გამოძახებების ნაკრებები. C ენის პროგრამისტის თვალსაზრისით სისტემური გამოძახების გამოყენება არაფრით არ განსხვავდება C ენის სტანდარტული ANSI ბიბლიოთეკის რომელიმე ფუნქციის გამოყენებისგან, მაგალითად, როგორიცაა სტრიქონებთან სამუშაო ფუნქციები: strlen(), strcpy() და ა.შ. UNIX-ის სტანდარტული ბიბლიოთეკა libc უზრუნველყოფს ყოველი სისტემური გამოძახების C ინტერფეისს. ამას მივყავართ იქამდე, რომ ყოველი სისტემური გამოძახება პროგრამისტისთვის წააგავს C ენის ფუნქციას.

სისტემურ გამოძახება შეიძლება დასრულდეს წარმატებით ან წარუმატებლად. წარუმატებლად დასრულების აღსანშნავად უმეტეს სისტემურ გამოძახებაში გამოიყენება მნიშვნელობა -1, ხოლო წარმატებული დასრულების აღსანიშნავად კი რაიმე არაუარყოფითი მნიშვნელობა. სისტემური გამოძახებები, რომლებიც აბრუნებენ მიმთითებელს, შეცდომის შემცველი სიტუაციის იდენტიფიცირებისთვის, იყენებენ მნიშვნელობას NULL. შეცდომის ზუსტ მიზეზებში გასარკვევად C-ინტერფეისი გვთავაზობს <errno.h> ფაილში განსაზღვრულ გლობალურ ცვლადს - errno, მის შესაძლო მნიშვნელობებთან და მათ მოკლე აღწერებთან ერთად. შევნიშნოთ, რომ errno ცვლადის გაანალიზება საჭიროა შეცდომის შემცველი სიტუაციის წარმოშობისთანავე, ვინაიდან წარმატებით დასრულებული სისტემური გამოძახება არ ცვლის მის მნიშვნელობას. პროგრამის შესრულებისას სტანდარტული შეცდომაზე სიმბოლური ინფორმაციის მისაღებად შესაძლებელია გამოყენებული იქნას სტანდარტული UNIX-ფუნქცია perror().

```
perror() ფუნქციის პროტოტიპი  
#include<stdio.h>  
void perror(char *str);  
ფუნქციის აღწერა
```

`perror()` ფუნქცია გამოიყენება შეცდომაზე შეტყობინების გამოსატანად, რომელიც შეესაბამება გამოტანის სტანდარტული ნაკადის შეცდომის დაფიქსირებისას `errno` სისტემური ცვლადის მნიშვნელობას. ფუნქცია ბეჭდავს მთლიანად `str` სტრიქონს (თუ `str` პარამეტრი არ უდრის `NULL`-ს), ორწერტილს, ჰარის და წარმოშობილი შეცდომის შესაბამისი შეტყობინების ტექსტს, ახალ ხაზზე გადასვლის სიმბოლოთი ('\'').

1.2. ფაილის სრული და მიმართებითი სახელები

ყველა ოპერაციული სისტემისთვის ფაილის ცნება წარმოადგენს ერთერთ მნიშვნელოვან ცნებას. UNIX-მსგავს ოპერაციულ სისტემებში ფაილის ცნების მნიშვნელობა უფრო გამოჩანს ვიდრე Windows-ის ოპერაციულ სისტემებში. ყველაფერი რაც ხდება UNIX-მსგავს ოპერაციულ სისტემებში რეალიზებულია ფაილის ცნებაზე დამოკიდებულებით. სანამ ფაილურ სისტემას და ფაილს დაწვრილებით განვიხილავდეთ ფაილებთან დაკავშირებული ზოგიერთი ცნების განსასაზღვრავად შეგვიძლია დავკავშიროთ ფაილზე ზოგადი ცნებით: ფაილი ეს არის მყარ დისკზე განთავსებული ერთ მონაცემებთან დაკავშირებული მექსიერების მისამართების ერთობლიობა (ნაკრები).

ისევე, როგორც სხვა ოპერაციულ სისტემებში, UNIX-მსგავს ოპერაციულ სისტემაში ყოველი ფაილი გაერთიანებულია ხისებრ ლოგიკურ სტრუქტურაში. ფაილები შეიძლება გაერთიანდნენ დირექტორიებში ან კატალოგებში¹. არ არსებობს ფაილი, რომელიც არ შედის არცერთ დირექტორიაში. დირექტორიები თავის მხრივ შეიძლება წარმოადგენდნენ სხვა დირექტორიის ქვედირექტორიებს. არსებობს დირექტორია, რომელშიც არ შედის არც ფაილი და არც ქვედირექტორია (ცარიელი დირექტორია) (ნახ. 1.1). ყველა დირექტორიას შორის არსებობს ერთადერთ დირექტორია, რომელიც არ წარმოადგენს სხვა დირექტორის ქვედირექტორიას, ასეთ დირექტორიას საბაზო ან `root` დირექტორია ეწოდება. UNIX ოპერაციულ სისტემაში არსებობს 5 ტიპის ფაილი. ჩვენ ძირითადად განვიხილავთ ორს:

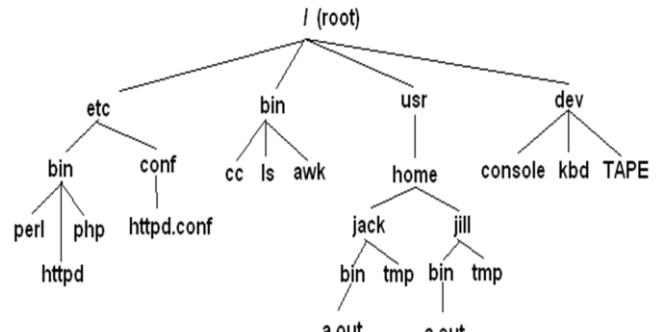
ჩვეულებრივი ანუ რეგულარული ფაილი (რომელიც შეიძლება შეიცავდეს პროგრამის მონაცემებს, შესრულებად კოდს, მომხმარებლის მონაცემებს და ა.შ.) და დირექტორია.

ყოველ ფაილს უნდა გააჩნდეს სახელი. სხვა ოპერაციულ სისტემების მსგავსად UNIX ოპერაციულ სისტემაში ფაილისთვის სახელის მინიჭებისას არსებობს გარკვეული შეზღუდვები. POSIX სტანდარტში UNIX ოპერაციული სისტემისთვის სისტემური გამოძახების ინტერფეისი შეიცავს მხოლოდ სამ ცხად შეზღუდვას:

- სახელის სიგრძე არ უნდა აღემატებოდეს სისტემაში გათვალისწინებულ სიგრძეს (Linux-სთვის - 255 სიმბოლო);
- არ შეიძლება `NULL` სიმბოლოს გამოყენება;
- არ შეიძლება `'/'` სიმბოლოს გამოყენება².

აკრძალული სიმბოლოების რიცხვს ასევე შეიძლება მივაკუთვნოთ სიმბოლოები: `"", "?", "\",` `"\\" და "\\"`.

ნახ. 1.1. ფაილური სისტემის სტრუქტურის მაგალითი



¹ Windows-ის ოპერაციულ სისტემაში გამოიყენება ცნება ფოლდერი

² ერთადერთ გამონაკლისს წარმოადგენს საბაზო დირექტორია, რომლის სახელიცაა `'/'`. სწორედ ის წარმოადგენს ერთადერთ ფაილს, რომელსაც ფაილურ სისტემაში გააჩნია უნიკალური სახელი

სისტემაში არსებულ ფაილებს უნდა გააჩნდეთ სახელი, რომელიც უნიკალური იქნება მხოლოდ შესაბამისი დირექტორიის ფარგლებში. UNIX ოპერაციულ სისტემაში ფაილთან დაკავშირებით გვაქვს ფაილის სრული და მიმართებითი სახელის ცნება. როგორც ნახ. 1.1-დან ჩანს „jack“ და „jill“ დირექტორიაში შედის ერთიდაიმავე სახელის მქონე სხვადასხვა დირექტორია („bin“), რომლებიც თავის მხრივ შეიცავენ ერთიდაიმავე სახელის მქონე სხვადასხვა ფაილს („a.out“). jack დირექტორიის შესაბამისი a.out ფაილისთვის ამოვწეროთ ყველა იმ დირექტორიის სახელი, რომლებიც მდებარობენ ფაილური სისტემის ლოგიკურ ხეზე საბაზო დირექტორიიდან დაწყებული ამ ფაილამდე (/usr/home/jack/bin/a.out). მსგავს მიმდევრობაში პირველი ყოველთვის იქნება საბაზო დირექტორიის სახელი, ხოლო ბოლო დანიშნულების ფაილის სახელი. ეს სახელები, გარდა საბაზო დირექტორიისა და მის შემდეგ პირველი დირექტორიის სახელისა, ერთმანეთისგან გამოვყოთ სიმბოლოთი ‘/’ („/usr/home/jack/bin/a.out“). მიღებული ჩანაწერი ცალსახად განსაზღვრავს ფაილის ადგილმდებარეობას ფაილური სისტემის ლოგიკურ ხეზე. სწორედ ასეთ ჩანაწერს ეწოდება ფაილის სრული სახელი.

ფაილის სრული სახელი შეიძლება შეიცავდეს დირექტორიების საკმაო რაოდენობას და იყოს საკმარისად გრძელი, ასეთ სახელებთან მუშაობა კი ყოველთვის არაა მოხერხებული. ოპერაციულ სისტემაში მომუშავე ყოველი პროგრამისთვის, ბრძანებათა ინტერპრეტატორის³ ჩათვლით, რომელიც ასრულებს შეტანილ ბრძანებებს და გამოაქვს ახალი ბრძანების შეტანის ველი, ფაილური სისტემის ლოგიკურ ხეზე გამოიყოფა ერთერთი დირექტორია, რომელიც ინიშნება მოცემული პროგრამისთვის მიმდინარე ანუ სამუშაო დირექტორიად. იმის გაგება, თუ რომელი დირექტორია წარმოადგენს ბრძანებათა ინტერპრეტატორისთვის მიმდინარე დირექტორიას შესაძლებელია pwd ბრძანებით.

pwd ბრძანების სინტაქსისი

```
pwd
```

ბრძანების აღწერა

pwd ბრძანებას გამოაქვს მიმდინარე დირექტორიის სახელი მუშა ბრძანებათა ინტერპრეტატორისთვის.

მიმდინარე დირექტორიის ცოდნით შესაძლებელია ფაილური სისტემის ხეზე მისგან დანიშნულების ფაილამდე გზის გადება. კვანძების (დირექტორიების) მიმდევრობა, რომლებიც შეგვხვდება ამ გზაზე ჩავწეროთ შემდეგი სახით: კვანძი, რომელიც შეესაბამება მიმდინარე დირექტორიას ამ ჩანაწერში არ შევა; საბაზო კატალოგის მიმართულებით მოძრაობისას შემხვედრ ყოველ დირექტორიას აღვნიშნავთ სიმბოლოთი ‘..’, ხოლო მისგან მოძრაობისას შემხვედრ დირექტორიებს კი ჩავწერთ. ამ ჩანაწერში სხვადასხვა კვანძები გამოვყოთ სიმბოლოთი ‘/’. მიღებულ ჩანაწერს ეწოდება ფაილის მიმართებითი სახელი. ფაილის მიმართებითი სახელი იცვლება მიმდინარე დირექტორიის შეცვლისას. მაგალითად ჩვენს მაგალითში (ნახ. 1.1), თუ მიმდინარე დირექტორია არის „/usr/home/jill“, მაშინ „/usr/home/jack/bin/a.out“ ფაილისთვის მიმართებითი მისამართი იქნება „.../..../usr/home/jack/bin/a.out“⁴, ხოლო, თუ მიმდინარე დირექტორია არის „/usr/home/jack“, მაშინ მისი მიმართებითი სახელი იქნება „bin/a.out“.

³ ბრძანებათა ინტერპრეტატორი არის პროგრამა, რომელიც ინტერაქტიულ რეჟიმში მუშაობს და მომზმარებელს აძლევს ბრძანებების შესრულების საშუალებას. მაგალითად, Linux-ში ასეთი პროგრამა terminal

⁴ შევნიშნოთ, რომ მოცემულ შემთხვევაში, მოცემული /usr/home/jack/bin/a.out ფაილისთვის მიმართებითი სახელის მისაღებად აუცილებლობას არ წარმოადგენ მიმართებით სახელში ფაილამდე სრული სახელი გამოვიყენოთ („.../..../usr/home/jack/bin/a.out“). რადგანაც jill და jack დირექტორიებისთვის home წარმოადგენს „მშობელ დირექტორიას“, ამიტომ მიმართებით სახელის მისაღებად საჭიროა ერთი დონით ასვლა და სასურველ დირექტორიამდე გზის გადება, ანუ /usr/home/jack/bin/a.out ფაილისთვის მიმართებითი სახელი იქნება „jack/bin/a.out“

1.2.1. მომხმარებლის საშინაო დირექტორია

სისტემაში დარეგისტრირებული ყოველი მომხმარებლისთვის იქმნება სპეციალური დირექტორია (Desktop), რომელიც მისი სისტემაში რეგისტრაციის შემდეგ იქცევა მისთვის მიმდინარე დირექტორიად. ამ დირექტორიამ მიიღო სახელწოდება მომხმარებლის საშინაო დირექტორია. `pwd` ბრძანების მეშვეობით შესაძლებელია იმის გარკვევა, თუ რომელია მომხმარებლის საშინაო დირექტორია.

1.2.2. უნივერსალური ცნობარი - ბრძანება `man`

აუცილებელია ყველა ოპერაციულ სისტემას გააჩნდეს საკუთარი საცნობარო მასალა, რომელშიც აღწერილი იქნება ბრძანებები, მათი სინტაქსისი, ოფციები და სხვა დამატებითი ინფორმაცია, რომელიც ხელშემწყობი იქნება ოპერაციულ სისტემასთან მუშაობისას. Windows-ის ოპერაციული სისტემისთვის ასეთი საცნობარო მასალის შემცველი არის უტილიტი `help`⁵. UNIX-მსგავს აპერაციული სისტემებში გამოიყენება უტილიტი `man`. მისი გამოყენებით შესაძლებელია ამომწურავი ინფორმაციის მიღება სასწავლო კურსის მსვლელობისას გამოყენებულ ბრძანებებზე და სისტემურ გამოძახებებზე, მათი გამოყენების სინტაქსისზე, პარამეტრებზე და ოფციებზე, რომლებიც დამატებით შესაძლებლობებს მატებენ მათ.

შევნიშნოთ, რომ ამ კურსის მსვლელობისას გამოყენებულ ბრძანებები და სისტემური გამოძახებები არ იქნება გამოყენებული სრულად (შესაძლებლობების თვალსაზრისით). ამ ბრძანებებზე ამომწურავი ინფორმაციის მიღება შესაძლებელია UNIX Manual-ით. UNIX Manual-ში ინფორმაციის დიდი ნაწილი ხელმისაწვდომია ინტერაქტიული ფორმით `man` უტილიტის გამოყენებით.

`man` უტილიტის გამოყენება ადვილია, მისი სინტაქსისა

`man NAME`

სადაც `NAME` - ეს არის ის ბრძანება, უტილიტი, სისტემური გამოძახება, ბიბლიოთეკური ფუნქცია ან ფაილი, რომელზეც ვსაჭიროებთ დამატებითი ინფორმაციის მიღებას. ბრძანების შესრულებით მიღება მისი დანიშნულების და გამოყენების აღმწერი გვერდი, რომელიც გამოტანილი იქნება ტერმინალის შიგნით. თუ აღმწერი ინფორმაცია რამდენიმე გვერდისგან შედგება, მაშინ გვერდების გადასაფურცლად გამოიყენება ღილაკი `<space>`, წინა გვერდზე დასაბრუნებლად - კომბინაცია `<ctrl> + `, ხოლო ინფორმაციის დათვალიერების რეჟიმიდან გამოსასვლელად კი - ღილაკი `<q>`. გვერდის დათვალიერების რეჟიმიდან გამოსვლის შემდეგ ისევ ვუბრუნდებით ტერმინალს და შესაძლებელია ინტერაქტიულ რეჟიმში ბრძანებების შეყვანა.

თანამედროვე ოპერაციულ სისტემებში თავმოყრილია დიდი მოცულობის ინფორმაცია. ამ ინფორმაციაში შესაძლებელია მეორდებოდეს გარკვეული (ბრძანების ან სისტემური გამოძახების) სახელი განსხვავებული კონტექსტით. სასურველ ბრძანებაზე ამომწურავი ინფორმაციის მიღების მცდელობისას შეიძლება მიღებული იქნას ინფორმაცია, რომელიც პასუხობს საძიებო ობიექტს და, ასევე, ინფორმაცია, რომელიც კონტექსტიდან გამომდინარე სცდება ინტერესის სფეროს. დიდი მოცულობის ინფორმაციაში ძებნის ოპერაციის გაადვილების მიზნით მთელი ინფორმაცია დაყოფილია ჯგუფებად. ჯგუფებში ინფორმაცია გაერთიანებულია ბრძანებების, დანიშნულების და ფუნქციების მიხედვით, მაგალითად, სისტემური გამოძახებები, ბიბლიოთეკური ფუნქციები, ბირთვის ბრძანებები და ა.შ.

ინფორმაციის დაყოფა ჯგუფებად შესაძლებელია განსხვავდებოდეს UNIX-მსგავს სისტემების სხვადასხვა ვერსიებს შორის. მაგალითად, Linux-ში ინფორმაცია დაყოფილია

⁵ Windows 7-დან დაწყებული დამატებით შეიძლება გამოყენებული იქნას უტილიტი `get-help`

ჯგუფებად:

1. შესრულებადი ფაილები ან ინტერპრეტატორის ბრძანებები;
2. სისტემური გამოძახებები;
3. ბიბლიოთეკური ფუნქციები;
4. სპეციალური ფაილები (ჩვეულებრივ მოწყობილობათა ფაილები);
5. სისტემური ფაილების ფორმატები და მიღებული შეთანხმებები;
6. თამაშები;
7. მაკროპაკეტები და უტილიტები - ისეთი როგორიცაა man;
8. სისტემური ადმინისტრატორის ბრძანებები;
9. ბირთვის ქვეპროგრამები (არასტანდარტული განყოფილება).

საჭირო ბრძანებაზე ამომწურავი ინფორმაციის მისაღებად დამატებით სასურველია იმის ცოდნა თუ რომელ ჯგუფს მიეკუთვნება შესაბამისი ბრძანება. იმისთვის, რომ გავარკვიოთ როგორაა დაყოფილი მთელი ინფორმაცია ჯგუფად საჭირო man ბრძანება გამოვიყენოთ ფორმით: man man, ხოლო იმისთვის, რომ საჭირო ბრძანებაზე მივიღოთ ამომწურავი ინფორმაცია შესაბამისი ჯგუფიდან man ბრძანება უნდა გამოვიყენოთ ფორმით:

man GROUP NAME

სადაც GROUP არის შესაბამისი ჯგუფის სახელი, ხოლო NAME კი არის სასურველი ბრძანების სახელი.

1.3. ფაილური სისტემის უმარტივესი ბრძანებები

როგორც ზემოთ აღვნიშნეთ, ყოველ პროგრამას ოპერაციულ სისტემაში გამოეყოფა სამუშაო დირექტორია. ნებისმიერი მოქმედება (როგორიცაა ახალი ფაილის შექმნა, ფაილის რედაქტირება და ა.შ.), რომელიც იქნება განხორციელებული ამ პროგრამის მიერ ყოველგვარი ცვლილების გარეშე რეალიზებული იქნება მხოლოდ პროგრამის სამუშაო დირექტორიაში. მაგალითად, გრაფიკულ რეჟიმში ტექსტური რედაქტორით ფაილის შექმნისას ფაილი შეიქმნება ტექსტური რედაქტორის სამუშაო დირექტორიაში, თუ მომხმარებელი არ შეცვლის მას. ხშირად ბრძანებათა ინტერპრეტატორთან მუშაობისას შესაძლებელია მოვარის სხვადასხვა ბრძანებების შესრულება სამუშაო დირექტორიისგან განსხვავებულ საბაზო დირექტორიის სხვა მხარეს განთავსებულ დირექტორიაში. ამ შემთხვევაში ბრძანებების შესრულება მოითხოვს ყოველ ბრძანებაში მითითებული იყოს საჭირო დირექტორიის სრული ან მიმართებითი სახელი. ბრძანების ამ ფორმით გამოყენება ართულებს მას. ბრძანების გამოყენების გამარტივება შესაძლებელია თუ შესაბამის დირექტორიას გადავაქცევთ სამუშაო დირექტორიად. ამ შემთხვევაში საკმარისი იქნება ბრძანებაში მითითებული იყოს შესაბამისი ფაილის მხოლოდ სახელი, რაც ამარტივებს ბრძანების ჩაწერას.

სამუშაო დირექტორიად ფაილური სისტემის ნებისმიერი დირექტორიის გადასაქცევად გამოიყენება ბრძანება cd (change directory). მისი სინტაქსია

cd DIRNAME

სადაც DIRNAME⁶ არის იმ დირექტორიის სრული ან მიმართებითი სახელი, რომელიც უნდა იქცეს სამუშაო დირექტორიად.

ხშირად ჩვენ ვსაჭიროებთ ვიცოდეთ თუ რა ფაილებია განთავსებული ამა თუ იმ

⁶ შევნიშნოთ, რომ სადაც შეგვხვდება სიტყვა DIRNAME, მის ქვეშ ვიგულისხმებთ დირექტორიის სრულ ან მიმართებით სახელს

დირექტორიაში, როგორია ფაილებზე დაშვების უფლებები, არის თუ არა იქ დამალული ფაილები და ა.შ. დირექტორის შიგთავსის (ხილული დირექტორიების და ფაილების) დასათვალიერებლად გამოიყენება ბრძანება ls (list), მისი სინტაქსისია

ls DIRNAME

სადაც DIRNAME არის იმ დირექტორის სრული ან მიმართებითი სახელი, რომლის შიგთავსის დათვალიერებასაც ვაპირებთ.

ტერმინალში შესრულებულ ls ბრძანებას, ოფციისა და ფაილის სახელის მითითების გარეშე, გამოაქვს სამუშაო დირექტორიაში განთავსებული ხილული ფაილების ჩამონათვალი. -a ოფციით ბრძანებას გამოაქვს მითითებულ ფაილში არსებული ხილული და დამალული⁷ ფაილების ჩამონათვალი, ხოლო -l (ლ) ოფციით კი - დირექტორიაში განთავსებული ფაილების ჩამონათვალი მათივე ატრიბუტებით: დაშვების უფლებები, მფლობელი, ჯგუფი, ფაილის სახელი და ა.შ.

ბრძანებათა ინტერპრეტატორიდან შესაძლებელია ტექსტური მონაცემების შემცველი ფაილის შიგთავსის დათვალიერება მის გაუხსნელად. ამ მიზნით გამოიყენება ბრძაბება cat, მისი სინტაქსისია

cat FILENAME⁸

სადაც FILENAME ტექსტური მონაცემების შემცველი ფაილია⁹. cat ბრძანებით შესაძლებელია ერთზე მეტი ფაილის შიგთავსის გამოტანა. ამ შემთხვევაში საჭიროა cat ბრძანებას მივუწეროთ ფაილის სახელები იმ მიმდევრობით თუ როგორი მიმდევრობით გვინდა ისინი გამოჩნდნენ ტერმინალში.

გარდა ფაილის შიგთავსის დათვალიერებისა cat ბრძანებით ასევე შესაძლებელია ახალი ფაილის შექმნა და მისი რედაქტირების რეჟიმში გადასვლა ან რაიმე ფაილებში არსებული ტექსტური მონაცემების ერთ ფაილში ჩაწერა. ამ მიზნით საჭიროა მონაცემთა ნაკადის შეტანის შესაბამისი სიმბოლოს ‘>’ გამოყენება. ბრძანების სინტაქსისია

cat > FILENAME

cat FILENAME1 FILENAME2 ... FILENAMEN > FILENAME

პირველი ბრძანების შესრულების შედეგად შეიქმნება ფაილი - FILENAME და გადავდივართ მისი რედაქტირების რეჟიმში. თუ ასეთი სახელის მქონე ფაილი არსებობდა, მაშინ მასში არსებულ მონაცემებზე გადაეწერება კავიატურიდან შეტანილი მნიშვნელობები. რედაქტირების რეჟიმიდან გამოსასვლელად გამოიყენება კომბინაცია <Ctrl> + <D>. მეორე ბრძანების შესრულების შედეგად FILENAME1 FILENAME2 ... FILENAMEN ფაილებში განთავსებული მონაცემები, მიმდევრობის შენარჩუნებით, ჩაიწერება FILENAME ფაილში. ამასთან, ფაილებში არსებული მონაცემების გამოტანა ტერმინალის ეკრანზე არ მოხდება.

მონაცემთა ნაკადის შეტანის სიმბოლოს გამოყენებით, ასევე, შესაძლებელია ნებისმიერი ბრძანების (რომელიც იძლევა ტექსტურ შედეგს) შესრულების შედეგის ჩაწერა ფაილში (ტერმინალში გამოტანის გარეშე). ამ შემთხვევაში, მაგალითად, ls -al ბრძანების შესრულების შედეგის ფაილში FILENAME ჩასაწერად, ბრძანება უნდა შესრულდეს ფორმით

⁷ სისტემაში დამალული ფაილების სახელები იწყება სიმბოლოთი ‘?’ (წერტილი). ასეთი ფაილი შეიძლება შექმნილი იყოს სისტემის ან გამოყენებითი პროგრამის მიერ

⁸ შეენიშნოთ, რომ სადაც შეგვხვდება სიტყვა FILENAME, მის ქვეშ ვიგულისხმებთ ფაილის სრულ ან მიმართებით მისამართს

⁹ თუ დასათვალიერებელი ფაილი დიდი მოცულობისაა, მაშინ ტერმინალში გამოტანილი იქნება მისი ბოლო გვერდი. ასეთ ფაილებთან სამუშაოდ გამოიყენება ბრძანება more

ls -al > FILENAME.

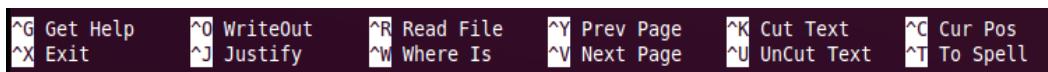
შევნიშნოთ, რომ თუ ფაილი FILENAME ბრძანების შესრულების მომენტში არ არსებობდა, მაშინ ის ავტომატურად შეიქმნება. თუ ფაილი არსებობდა, მაშინ იქ არსებულ მონაცემებს გადაეწერება ბრძანების შესრულების შედეგი.

ახალი ფაილის შექმნა ტერმინალიდან ასევე შესაძლებელია nano ტექსტური რედაქტორის გამოყენებით, რომელიც გადაგვიყვანს მისი რედაქტირების რეჟიმში. მისი სინტაქსისია

nano FILENAME

თუ ფაილი FILENAME სახელით უკვე არსებობდა შესაბამის დირექტორიაში, მაშინ გადავალთ მისი რედაქტირების რეჟიმში. nano ტექსტური რედაქტორის ქვედა ნაწილში განთავსებულია მართვის ღილაკების ჩამონათვალი (ნახ. 1.2), რომელთა გამოყენებითაც <Ctrl> ღილაკთან კომბინაციით შესაძლებელია სხვადასხვა მოქმედებების განხორციელება. მაგალითად, <Ctrl> + <X> კომბინაციით შესაძლებელია დოკუმენტის შენახვა და რედაქტირების რეჟიმიდან გამოსვლა.

ნახ. 1.2. nano ტექსტური რედაქტორის მართვის ღილაკები



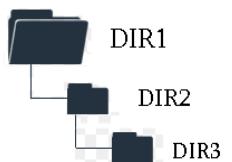
ფაილურ სისტემაში ახალი დირექტორის შესაქმნელად გამოიყენება ბრძანება mkdir (make directory), მისი სინტაქსისია

mkdir DIRNAME.

mkdir ბრძანების მეშვეობით ასევე შესაძლებელია რამდენიმე დირექტორის ერთდროული შექმნა. ამასთან, შესაძლებელია რამდენიმე ჩალაგებული დირექტორის ანუ იერარქიის შექმნა, რაც ნიშნავს შემდეგს: თუ სამუშაო დირექტორიაში (/home/student) არ გვაქვს არცერთი დირექტორია და გვინდა შევქმნათ მაგალითად, 3 დონით ჩალაგებული დირექტორია (DIR1/DIR2/DIR3) საჭიროა ტერმინალში ბრძანება შევასრულოთ შემდეგი მიმდევრობით

mkdir DIR1 DIR1/DIR2 DIR1/DIR2/DIR3

ანუ მიმდევრობით უნდა შევქმნათ ჯერ DIR1, შემდეგ DIR1/DIR2 და ბოლოს DIR1/DIR2/DIR3 დირექტორია.



ერთი დირექტორიიდან მეორეში ფაილის (ან ფაილების) გადასაკოპირებლად (copy-paste) გამოიყენება ბრძანება cp (copy). ის, ასევე, გამოიყენება ერთი დირექტორის (ან დიერექტორიების) სხვა დირექტორიაში რეკურსიული¹⁰ გადაკოპირებისთვის. მისი სინტაქსისია

cp FILENAME DESTINNAME

cp FILENAME1 FILENAME2 ... FILENAMEN DESTINNAME

cp -r SDIRNAME DESTINNAME

cp -r DIR1 DIR2 ... DIRN DESTINNAME

სადაც SDIRNAME (source directory) საწყისი დირექტორიაა, რომლიდანაც უნდა მოხდეს ფაილების გადაკოპირება, DESTINNAME (destination directory) დანიშნულების დირექტორია, სადაც უნდა მოხდეს მონაცემების გადაკოპირება, -r ოფცია გამოიყენება რეკურსიული გადაკოპირებისთვის.

¹⁰ რეკურსიული კოპირება ნიშნავს დირექტორის გადაკოპირებას მასში არსებული ქვედირექტორიებითა და ფაილებით

ფაილის ერთი დირექტორიიდან მეორეში გადასაადგილებლად (cut-paste) გამოიყენება ბრძანება mv (move). მისი სინტაქსისა

```
mv SOURCEFILE DESTFILE
```

```
mv FILENAME1 FILENAME2 ... FILENAMEN DESTINNAME
```

თუ პირველ ბრძანებაში ორივე მისამართი უთითებს ერთიდაიმავე დირექტორიას, მაშინ მოხდება SOURCEFILE ფაილისთვის სახელის გადარქმევა და ახალი სახელი იქნება DESTFILE. სხვა შემთხვევაში მოხდება ფაილის ერთი დირექტორიიდან მეორეში გადაადგილება. მეორე ბრძანებით ხდება FILENAME1 FILENAME2 ... FILENAMEN ფაილების გადაადგილება DESTINNAME დირექტორიაში. ასევე, mv ბრძანებით, დამატებითი ოფციის გამოყენების გარეშე, შესაძლებელია ერთი დირექტორიის (ან დირექტორიების) გადაადგილება მეორე დირექტორიაში.

დირექტორიიდან ფაილის (ან ფაილების) წასაშლელად გამოიყენება ბრძანება rm (remove). მისი სინტაქსისა

```
rm FILENAME1 FILENAME2 ... FILENAMEN
```

დირექტორიიდან მთლიანი ქვედირექტორიის წასაშლელად საჭიროა -r ოფციის გამოყენება, ანუ საჭიროა რეკურსიული წაშლის განხორციელება ფორმით

```
rm -r DIR1 DIR2 ... DIRN
```

შევნიშნოთ, რომ ზემოთ გამოყენებულ ბრძანებებში ფაილის სრული ან მიმართებითი სახელები შეიძლება უთითებდნენ როგორც ერთიდაიმავე დირექტორიის, ისე სხვადასხვა დირექტორიის ფაილებს.

1.4. ფაილების სახელების შაბლონი

ოპერაციულ სისტემაში სასურველი ფაილის მოძებნა შეიძლება გართულდეს რამდენიმე მიზეზის გამო: ფაილის სახელი უცნობია, მისი ადგილმდებარეობა უცნობია და ა.შ. ასეთ შემთხვევაში ოპერაციულ სისტემაში უნდა არსებობდეს ე.წ. მეტასიმბოლოგის მხარდაჭერა, რომლებიც ძებნის ოპერაციის გამარტივებას გამოიწვევენ. ასეთი მეტასიმბოლოგი შეიძლება გამოყენებულიქნას არამხოლოდ ძებნის მოქმედების განხორციელებისას, არამედ ფაილების მიმართ სხვადასხვა ბრძანებების გამოყენების დროსაც. ხშირად ამ მეტასიმბოლოგის გამოყენებით ბრძანებებთან გამოსაყენებელ სიმბოლოთა კომბინაციას შაბლონს უწოდებენ. მეტასიმბოლოგი, რომლებიც ამარტივებენ ამა თუ იმ ბრძანების გამოყენებას მოცემულია შემდეგ ცხრილში

* -	შეესაბამება ყველა სიმბოლოს ჰათვლით;
? -	შეესაბამება მხოლოდ ერთ სიმბოლოს;
[...] -	შეესაბამება კვადრატულ ფრჩილებში მითითებულ ნებისმიერ სიმბოლოს ან ალფავიტის ასოს, რომლებიც ერთიმეორისგან შეიძლება გამოყოფილი იყოს სიმბოლოთი ‘,’ ან ‘-’ (თუ ეთითება უწყვეტი შუალედი)

მაგალითად, *.c შაბლონს აკმაყოფილებს მიმდინარე დირექტორიის ყველა ის ფაილი, რომლის გაფართოებაც არის .c. [a-d]* შაბლონს აკმაყოფილებს მიმდინარე დირექტორიის ყველა ის ფაილი, რომლის სახელიც იწყება a, b, c, d სიმბოლოებიდან ნებისმიერით. არსებობს ერთი შეზღუდვა * მეტასიმბოლოს გამოყენებაზე ფაილის სახელის დასაწყისში, მაგალითად, *c შაბლონის შემთხვევაში. ასეთი შაბლონებისთვის იმ ფაილების სახელები, რომლებიც იწყება სიმბოლოთი ‘.’ ითვლება, რომ ისინი არ აკმაყოფილებენ შაბლონს.

1.5. მომხმარებელი და მისი ჯგუფი

მომხმარებელს ოპერაციულ სისტემაში მუშაობა შეუძლია მხოლოდ მასში გარკვეული სახელით დარეგისტრირების შემდეგ. ოპერაციულ სისტემას არ შეუძლია ლოგიკური სახელებით მანიპულირება, ამიტომ სისტემაში დარეგისტრირებულ ყოველ მომხმარებელს ენიჭება საკუთარი საიდენტიფიკაციო ნომერი UID¹¹ (user identifier).

სისტემაში დარეგისტრირებული მომხმარებელი აუცილებლად საჭიროებს ოპერაციული სისტემის გარკვეულ რესურსზე მიმართვას. მომხმარებლები შეიძლება იყვნენ სხვადასხვა პრივილეგიებით და თანაბრად არ საჭიროებენ რესურსებზე წვდომას. ოპერაციულ სისტემაში მომხმარებლები პრივილეგიების მიხედვით ერთიანდება სხვადასხვა ჯგუფებში და ღებულობენ რესურსებზე შესაბამის წვდომას. ოპერაციული ჯგუფების ერთიმეორისგან განსხვავების მიზნით იყენებს ჯგუფის იდენტიფიკატორს GID¹² (group identifier).

მომხმარებლის იდენტიფიკატორის - UID, და მისი ჯგუფის იდენტიფიკატორის - GID, მნიშვნელობის მისაღებად შესაბამისად გამოიყენება სისტემური გამოძახებები getuid() და getgid().

getuid() და getgid() სისტემურ გამოძახებათა პროცესი

```
#include<sys/types.h>
#include<unistd.h>
uid_t getuid(void);
gid_t getgid(void);
```

სისტემურ გამოძახებათა აღწერა

getuid სისტემური გამოძახება აბრუნებს მიმდინარე პროცესის მომხმარებლის იდენტიფიკატორს.

getgid სისტემური გამოძახება აბრუნებს მიმდინარე პროცესის მომხმარებლის ჯგუფის იდენტიფიკატორს.

uid_t და gid_t არის C ენის მთელრიცხვა ტიპის სინონიმი.

UNIX-მსგავს სისტემაში განასხვავებენ სამი კატეგორიის მომხმარებელს: მომხმარებელი, რომელიც რეგისტრირებულია სისტემაში გარკვეული სახელით, ჯგუფი, რომელსაც ის მიეკუთვნება და სხვა მომხმარებლები, რომლებიც მიმდინარე მომხმარებლის ჯგუფს არ მიეკუთვნებიან. თითოეული კატეგორიის აუცილებლად საჭიროებს გარკვეულ დაშვების უფლებებს.

1.6. ფაილზე დაშვების უფლებები

გამოთვლით სისტემას შეიძლება ჰყავდეს ბევრი მომხმარებელი. თითოეული მომხმარებელი ინახავს მისთვის მნიშვნელოვან სხვადასხვა ფაილებს (იურიდიული, ფინანსური და ა.შ.). ოპერაციულ სისტემას უნდა შეეძლოს მომხმარებლისთვის მნიშვნელოვანი ფაილების დაცვა არასანქცირებული დაშვებისგან. ამ მიზნით UNIX-მსგავს სისტემებში განასხვავებენ დაშვების სამუფლებას:

- კითხვის უფლება - r (read);
- რედაქტირების უფლება - w (write);
- შესრულების უფლება - x (execute).

მონაცემების შემცველი ფაილებისთვის ამ უფლებების არსი ემთხვევა მათ შინაარსს. დირექტორიებისთვის ის რამდენადმე განსხვავებულია. დირექტორიისთვის კითხვის უფლება ნიშნავს ამ დირექტორიაში არსებული ფაილების სახელების კითხვას. ვინაიდან დირექტორიისათვის უფლება „შესრულება“ ყოველგვარ აზრს მოკლებულია (ისევე როგორც არაშესრულებადი რეგულარული ფაილისთვის), მისთვის შესრულების უფლება იცვლის მნიშვნელობას:

¹¹ გარკვეული რიცხვითი მნიშვნელობა, რომელიც უნიკალურია სისტემის მასშტაბით

¹² გარკვეული რიცხვითი მნიშვნელობა, რომელიც უნიკალურია სისტემის მასშტაბით

ამ უფლების ქონა იძლევა დირექტორიაში შემავალ ფაილებზე დამატებითი ინფორმაციის მიღების შესაძლებლობას (მათი მოცულობა, მფლობელი, შექმნის თარიღი და ა.შ.). ამ უფლების გარეშე შეუძლებელია დირექტორიაში არსებული ფაილის შემცველობის დათვალიერება, რედაქტირება და შესრულება. შესრულების უფლება დირექტორიისთვის საჭიროა იმისთვის, რომ მისი გადაქცევა შესაძლებელი იყოს მიმდინარე დირექტორიად. ეს უფლება საჭიროა მისკენ მიმავალ გზაზე განთავსებული ყველა დირექტორიისათვის. დირექტორიისათვის ჩაწერის უფლება იძლევა მისი შემცველობის შეცვლის შესაძლებლობას: ფაილის შექმნასა და წაშლას, მათი სახელის გადარქმევას. აღვნიშნოთ, რომ ფაილის წასაშლელად საკმარისია იმ დირექტორი-ისთვის შესრულების და ჩაწერის უფლების ქონა, რომელშიც შედის ფაილი, მიუხედავად თვითონ ფაილზე დაშვების უფლების ქონისა.

ფაილზე დაშვების უფლებების დასათვალიერებლად გამოიყენება ls ბრძანება ოფციით -l (ლ).

ფაილურ სისტემაში წარმოქმნილი ყოველი ფაილისათვის ინახება მისი მფლობელისა და მფლობელის ჯგუფის სახელები. შევნიშნოთ, რომ მფლობელთა ჯგუფის სახელი აუცილებელი არაა ემთხვეოდეს ფაილის შემქმნელის ჯგუფის სახელს. გამარტივებულად შეიძლება ჩაითვალოს ის, რომ Linux ოპერაციულ სისტემაში ახალი ფაილის შექმნისას მის მფლობელად ითვლება მომხმარებელი, რომელმაც შექმნა ის, ხოლო მფლობელთა ჯგუფად კი - ის ჯგუფი, რომელსაც მიეკუთვნება მომხმარებელი. ფაილის მფლობელს ან სისტემურ ადმინისტრატორს შეუძლია შეცვალოს ფაილის მფლობელი ან მფლობელთა ჯგუფი chown და chgrp ბრძანებების გამოყენებით შესაბამისად.

chown ბრძანების სინტაქსისი

```
chown owner FILENAME1 FILENAME2 ... FILENAMEN
```

ბრძანების აღწერა

chown ბრძანების მეშვეობით ხდება ფაილის მფლობელის ჯგუფის შეცვლა.

owner პარამეტრით შესაძლებელია ახალი მფლობელის მოცემა როგორც სიმბოლური ფორმით,

მფლობელის ფაილის სახელი username, ან რიცხვითი მნიშვნელობა, როგორც მისი UID.

FILENAME1 FILENAME2 ... FILENAMEN პარამეტრები კი - იმ ფაილების სახელებია, რომელთაც ეცვლებათ მფლობელები. მათ ნაცვლად შესაძლებელია შაბლონის გამოყენებაც

chgrp ბრძანების სინტაქსისი

```
chgrp group FILENAME1 FILENAME2 ... FILENAMEN
```

ბრძანების აღწერა

chgrp ბრძანების მეშვეობით ხდება ფაილის მფლობელის ჯგუფის შეცვლა.

group პარამეტრით შესაძლებელია ფაილის მფლობელთა ჯგუფის მოცემა როგორც სიმბოლური ფორმით, ფაილის მფლობელთა ჯგუფის სახელი, ასევე რიცხვითი მნიშვნელობით, როგორც მისი GID.

FILENAME1 FILENAME2 ... FILENAMEN პარამეტრები კი - იმ ფაილების სახელებია, რომელთაც ეცვლებათ მფლობელთა ჯგუფი. მათ ნაცვლად შესაძლებელია შაბლონის გამოყენებაც

შევნიშნოთ, რომ შესაძლებელია ბრძანების შესრულება საჭიროებდეს ადმინისტრატორის უფლებებს. ადმინისტრატორის სახელით ბრძანების შესასრულებლად საჭიროა ჩვეულებრივი ფორმით აკრეფილი ბრძანების წინ მიეთითოს სიტყვა tsudo. ბრძანების ასეთნაირად შესრულების შემთხვევაში სისტემა მოითხოვს ადმინისტრატორის პაროლის შეყვანას.

ფაილის მფლობელის და ჯგუფის შეცვლასთან ერთად შესაძლებელია ფაილზე დაშვების უფლებების შეცვლა. ამ მიზნით გამოიყენება ბრძანება chmod.

chmod ბრძანების სინტაქსისი

chmod [who] { + | - | = } [perm] FILENAME1 FILENAME2 ... FILENAMEN

ბრძანების აღწერა

chmod ბრძანების მეშვეობით ხდება ერთ ან რამდენიმე ფაილზე უფლებების შეცვლა.

who პარამეტრი განსაზღვრავს რომელი კატეგორიის მომხმარებლებისთვის ხდება დაშვების უფლებების შეცვლა. ის შეიძლება შეიცავდეს ერთ ან რამდენიმე სიმბოლოს:

a - ყველა ტიპის მომხმარებლის უფლებების მომართვა. თუ პარამეტრი who არაა მითითებული, მაშინ გაჩუმებით გამოიყენება a. დაშვების უფლებების მომართვისას ამ მნიშვნელობით მოცემული უფლებების მომართვა ხდება ფაილის შექმნის ნიღაბის მნიშვნელობის გათვალისწინებით;

u - ფაილის მფლობელისთვის დაშვების უფლებების მომართვა;

g - ფაილის მფლობელის ჯგუფში შემავალი მომხმარებლებისთვის დაშვების უფლებების მომართვა;

o - ყველა დანარჩენი მომხმარებლებისთვის დაშვების უფლებების მომართვა.

ოპერაცია, რომელიც სრულდება დაშვების უფლებებზე მომხმარებელთა მითითებული კატეგორიისთვის, განისაზღვრება ერთერთით შემდეგი სიმბოლოებიდან:

+ - დაშვების უფლებების დამატება;

- - დაშვების უფლებების გაუქმება;

= - დაშვების უფლებების შეცვლა, ე.ი. ყველა არსებულის გაუქმება და ჩამოთვლილის დამატება.

თუკი perm პარამეტრი არაა განსაზღვრული, მაშინ დაშვების ყველა არსებული უფლება იქნება უარყოფილი. perm პარამეტრი განსაზღვრავს დაშვების იმ უფლებებს, რომლებიც იქნებიან დამატებული, უარყოფილი ან მომართული შესაბამისი ბრძანების სანაცვლოდ. ის წარმოადგენს შემდეგი სიმბოლოების ან ერთერთი მათგანის კომბინაციას:

r - უფლება კითხვაზე;

w - უფლება რედაქტირებაზე;

x - უფლება შესრულებაზე.

FILENAME1 FILENAME2 ... FILENAMEN პარამეტრები არის იმ ფაილების სახელები, რომელთათვისაც ხდება დაშვების უფლებების შეცვლა. სახელების ნაცვლად შესაძლებელია გამოიყენებული იქნას მათი შაბლონებიც

ახალი ფაილის შექმნისას ოპერაციული სისტემა მას ანიჭებს დაშვების გარკვეულ უფლებებს გაჩუმებით. ისმის კითხვა: რითი სარგებლობს ოპერაციული სისტემა ფაილისათვის უფლებების მინიჭებისას? ამ მიზნით ის იყენებს ფაილის შექმნის ნიღაბს იმ პროგრამისთვის, რომელიც ქმნის ფაილს.

1.7. მიდინარე პროცესის ფაილების შექმნის ნიღაბი

მიმდინარე პროცესის ფაილის შექმნის ნიღაბი (umask) გამოიყენება open() და mknod() სისტემური გამოძახებების მიერ ახლად შექმნილი ფაილისათვის ან FIFO-სთვის დაშვების საწყისი უფლებების მოსამართად. ფაილის შექმნის ნიღაბის უმცროსი 9 ბიტი შეესაბამება მომხმარებლის (რომელმაც შექმნა ფაილი), ჯგუფის (რომელსაც ის მიეკუთვნება), და სხვა დანარჩენი მომხმარებლის დაშვების უფლებებს, როგორც ეს ნაჩვენებია ქვემოთ რვაობითი ჩანაწერის ფორმით:

0400 – ფაილის შექმნელი მომხმარებლისთვის კითხვის უფლება;

0200 – ფაილის შექმნელი მომხმარებლისთვის ჩაწერის უფლება;

0100 – ფაილის შექმნელი მომხმარებლისთვის შესრულების უფლება;

0040 – ფაილის შექმნელი მომხმარებლის ჯგუფისთვის კითხვის უფლება;

0020 – ფაილის შექმნელი მომხმარებლის ჯგუფისთვის ჩაწერის უფლება;

0010 – ფაილის შექმნელი მომხმარებლის ჯგუფისთვის შესრულების უფლება;

- 0004 – სხვა დანარჩენი მომხმარებლებისთვის კითხვის უფლება;
- 0002 – სხვა დანარჩენი მომხმარებლებისთვის ჩაწერის უფლება;
- 0001 – სხვა დანარჩენი მომხმარებლებისთვის შესრულების უფლება;

რომელიმე ბიტის 1-ის ტოლი მნიშვნელობით ჩაწერა ახლად შექმნილი ფაილისთვის კრძალავს დაშვების უფლების შესაბამის ინიციალიზაციას. ფაილის შექმნის ნიღაბის მნიშვნელობა შეიძლება შეიცვალოს umask() სისტემური გამოძახების ან umask ბრძანების მეშვეობით. ფაილის შექმნის ნიღაბი მემკვიდრეობით გადადის შვილპროცესზე fork() სისტემური გამოძახების საშუალებით ახალი პროცესის წარმოქმნისას და შედის პროცესის სისტემური კონტექსტის უცვლელ ნაწილში exec() სისტემური გამოძახებისას. ამ მემკვიდრეობის შედეგად umask ბრძანების გამოყენებით ნიღაბის შეცვლა ზემოქმედებს ყველა პროცესზე, რომლებიც წარმოიქმნებიან ბრძანებათა გარსით, ახლად შექმნილი ფაილის დაშვების ატრიბუტებზე.

პროგრამა-ბირთვის ნიღაბის მიმდინარე მნიშვნელობის შეცვლა ან მისი დათვალიერება შესაძლებელია umask ბრძანების მეშვეობით. მაგალითად, იმისთვის, რომ გავანულოთ სისტემის მიერ ახალი შესაქმნელი ფაილისთვის გაჩუმებით გადაცემული მნიშვნელობები საჭიროა ფაილის შექმნამდე umask ბრძანება გამოვიყენოთ ფორმით umask(0).

umask ბრძანების სინტაქსისი

umask [value]

ბრძანების აღწერა

umask ბრძანება განკუთვნილია ბრძანებათა ბირთვის ფაილის ნიღაბის შესაქმნელად ან მისი მიმდინარე მნიშვნელობის დასათვალიერებლად. პარამეტრის გარეშე ბრძანებას გამოაქვს ფაილის შექმნის ნიღაბის მომართული მნიშვნელობა რვაობითი ფორმით. ახალი მნიშვნელობის მოსამართად ის მოიცემა როგორც value პარამეტრი რვაობითი ფორმით.

1.8. პროგრამული კოდი, მისი კომპილაცია და შესრულება

სასწავლო კურსის მსვლელობისას პროგრამული კოდს დავწერთ C++ ენაზე და შესაბამისად კოდისთვის გამოვიყენებთ cpp გაფართოების ფაილს. ჩვენს მიერ პროგრამული კოდი შეიძლება აკრეფილი იყოს უშუალოდ ტერმინალიდან nano ტექსტური რედაქტორის გამოყენებით ან ტექსტურ რედაქტორში kate (ნახ. 1.3), რომელშიც სამუშაო ველი იყოფა ორ ნაწილად. პირველ ნაწილში შესაძლებელია უშუალოდ პროგრამული კოდის აკრეფა, ხოლო მეორე ნაწილში გამოტანილია პროგრამაში ინტეგრირებული ბირთვის რეჟიმი, საიდანაც შესაძლებელია ბრძანების დაკომპილირება და შემდგომ მისი შესრულება.

UNIX-მსგავს სისტემებში პროგრამის კომპილიაციისთვის გამოიყენება ინტეგრირებული კომპილიატორები, როგორიცაა gcc, cc, g++ და ა.შ. სასწავლო კურსის მსვლელობისას ჩვენ გამოვიყენებთ g++ კომპილიატორს. g++ კომპილატორის გამოყენებით კოდის (main ფუნქციის) შემცველი ფაილის კომპილაცია შესაძლებელია ფორმით

g++ CODEFILE

სადაც CODEFILE პროგრამული კოდის შემცველი ფაილია. უნდა აღინიშნოს, რომ ფაილის კომპილაციისას g++ კომპილატორი კომპილაციის შედეგის ფაილისთვის

ნახ. 1.3. kate პროგრამის სამომხმარებლო ინტერფეისი

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
    pid_t a = fork();
    if (a == -1)
    {
        printf("პროცესი ვერ შეიქმნა");
        exit(EXIT_FAILURE);
    }
    else
    {
        if (a == 0) // შვილი პროცესი
    }
}

```

student@ubuntu:~/Desktop\$ cd /home/student/Desktop\$

გაჩუმების წესით აგენტირებს სახელს a.out. ეს სახელი იქნება დაგენერირებული g++ კომპილატორის მიერ კომპილირებული ყველა ფაილისთვის მიუხედავად იმისა, არიან თუ არა ეს ფაილები ერთ დირექტორიაში განთავსებული. ერთ დირექტორიაში განთავსებული რამდენიმე ფაილის დაკომპილირების შემთხვევაში დირექტორიაში დარჩება ბოლოს კომპილირებული ფაილის შედეგი, სხვა ფაილების კომპილაციის შედეგი იკარგება. იმისთვის, რომ არ მოხდეს სხვადასხვა ფაილების კომპილაციისას შესაბამისი შედეგის ფაილების ერთიმეორეზე გადაწერა სასურველია ფაილის კომპილაციისას მისთვის პროგრამისტის მიერ იყოს შერჩეული შესაბამისი სახელი. ამ მიზნით g++ კომპილატორიში გამოიყენება ოფცია -o (ო). ამისთვის g++ კომპილატორი უნდა გამოვიყენოთ ფორმით

```
g++ CODEFILE -o OUTNAME
```

სადაც OUTNAME არის კომპილაციის შედეგად მიღებული ფაილის სახელი. შევნიშნოთ, რომ კომპილაციის შედეგის ფაილისთვის აუცილებელობას არ წარმოადგენს “.out“ გაფართოების მიწერა.

კომპილირებული ფაილის შესრულება შესაძლებელია ფორმით

```
./OUTNAME
```

სემინარი 2. პროცესები Unix მსგავს სისტემებში

2.1. პროცესები და მისი კონტექსტი

ნებისმიერი ოპერაციული სისტემისათვის პროცესის ცნება თამაშობს მნიშვნელოვან როლს. ყველაფერი რაც გამოთვლით სისტემაში ხდება აგებულია პროცესების ცნების აბსტრაქციაზე. სისტემის ჩატვირთვიდან დაწყებული მისი მუშაობის დასრულებამდე. თანამედროვე ოპერაციულ სისტემებში ამ ცნებაზე დამოკიდებულებით არის რეალიზებული სხვადასხვა ტექნოლოგია, როგორიცაა ფაილური სისტემა, ვირტუალური მეხსიერება და ა.შ. ყველაზე მეტად პროცესის ცნების მნიშვნელობა ოპერაციული სისტემისთვის შეიძლება დანახული იქნას UNIX -მსგავს ოპერაციულ სისტემებთან მუშაობისას. ოპერაციულ სისტემაში არსებული ბირთვის და მომხმარებლის რეჟიმის მუშაობა ემყარება პროცესის ცნებას (სისტემური გამოძახებების შესრულება, სიგნალების გენერირება, მეხსიერების მართვა, განსაკუთრებული სიტუაციის დამუშავება, სხვადასხვა სერვისები).

სისტემაში წარმოქმნილი ყოველი პროცესი შესასრულებლად იყენებს სისტემაში არსებულ რესურსებს (მეხსიერება, პროცესორი, ფაილური სისტემის მიერ შეთავაზებული სამსახურები, შეტანა/გამოტანის მოწყობილობა და ა.შ.). ის შესასრულებელი სამუშაოს მიხედვით შეიძლება სრულდებოდეს როგორც მომხმარებლის (user-mode), ასევე, ბირთვის (kernel mode) რეჟიმში. მომხმარებლის რეჟიმში პროცესს შეუძლია განახორციელოს არაპრივილეგირებული საქმიანობა (მაგალითად, ტექსტურ რედაქტორში ტექსტის აკრეფა). მომხმარებლის რეჟიმიდან (შესაბამის სისტემური გამოძახების შესრულებით) ის გადადის ბირთვის რეჟიმში, სადაც მისთვის ნებადართულია პრივილეგირებული ბრძანებების შესრულება და ღებულობს შესაბამის მომსახურეობას (კითხულობს მონაცემებს, აწარმოებს გამოთვლებს და ა.შ.). რადგანაც პროცესი შეიძლება საჭიროებდეს ორივე რეჟიმში შესრულებას, ამიტომ მასთან ასოცირებულ ინფორმაციას ყოფენ ორ ნაწილად: მომხმარებლის კონტექსტი და ბირთვის კონტექსტი. მომხმარებლის კონტექსტის ქვეშ იგულისხმება ყველა ის ინფორმაცია, რომელიც საჭიროა მომხმარებლის რეჟიმში პროცესის შესასრულებლად, ესენია

- ინიცირებადი უცვლელი მონაცემები (მუდმივები);
- ინიცირებადი ცვლადი მონაცემები (ყველა ცვლადი, რომელთაც საწყისი მნიშვნელობა ენიჭება კომპილაციის ეტაპზე);
- არაინიცირებადი უცვლელი მონაცემები (ყველა სტატიკური ცვლადი, რომელთაც საწყისი მნიშვნელობა არააქვთ მინიჭებული კომპილაციის ეტაპზე);
- მომხმარებლის სტეკი;
- მონაცემები, რომლებიც განთავსებული არიან დინამიურად განაწილებად მეხსიერებაში, და ა.შ.

„ბირთვის კონტექსტი“ ცნების ქვეშ გაერთიანებულია სისტემური და რეგისტრული კონტექსტი. ბირთვის კონტექსტში გამოყოფენ ბირთვის სტეკს, რომელიც გამოიყენება პროცესის მუშაობისას ბირთვის რეჟიმში, და ბირთვის მონაცემებს, რომლებიც ინახება პროცესის მართვის ბლოკის (PCB) ანალოგიურ სტრუქტურებში. ბირთვის მონაცემებში შედის:

- მომხმარებლის იდენტიფიკატორი (UID);
- მომხმარებლის ჯგუფის იდენტიფიკატორი (GID);
- პროცესის იდენტიფიკატორი (PID);
- მშობელი-პროცესის იდენტიფიკატორი (PPID), და ა.შ.

2.2. პროცესის იდენტიფიკაცია

ყოველი ოპერაციულ სისტემა სისტემაში წარმოქმნილი პროცესების იდენტიფიცირებისთვის საჭიროებს გარკვეულ მექანიზმს. სხვადასხვა ოპერაციულ სისტემაში შეიძლება იდენტიფიკატორის როლში გამოყენებულ იქნას რიცხვითი მნიშვნელობა ან სიმბოლური სახელები ან მათი კომბინაცია. UNIX მსგავს სისტემებში წარმოქმნილი ყოველი პროცესი უნიკალური იდენტიფიკატორის როლში ენიჭება რიცხვითი მნიშვნელობა - PID (process identifier). სხვადასხვა სისტემებში პროცესის იდენტიფიცირებისათვის გამოყოფილი რიცხვითი მნიშვნელობები მერყეობს 0-დან გარკვეულ მაქსიმალურ მნიშვნელობამდე. მაგალითად, Intel-ის ტიპის 32-თანრიგა პროცესორის ბაზაზე Linux სისტემაში პროცესების იდენტიფიცირებისთვის გამოყოფილია შუალედი [0, 2³²-1]. სისტემაში პროცესისთვის საიდენტიფიკაციო ნომრის მინიჭებისას უნიკალობის შენარჩუნების მიზნით ყოველ ახალ პროცესს სისტემა რიცხვით მნიშვნელობას ანიჭებს ზრდადი მიმდევრობით (ანუ ბოლოს წარმოქმნილი პროცესის საიდენტიფიკაციო ნომერს + 1). პროცესის დასრულების შემდეგ მის მიერ დაკავებული ნომერი (ID) თავისუფლდება და მისი გამოყენება შესაძლებელია სისტემაში წარმოქმნილი ახალი პროცესის იდენტიფიცირებისათვის. იმის გამო, რომ ოპერაციულ სისტემაში არ დაირღვეს პროცესებისათვის საიდენტიფიკაციო ნომრების მინიჭების პროცესი სისტემაში დასრულებული პროცესებიდან გამონთავისუფლებული საიდენტიფიკაციო ნომრის მნიშვნელობის ხელახალი გამოყენება პროცესებისათვის ხდება მხოლოდ მას შემდეგ რაც მიღწეული იქნება იდენტიფიცირებისთვის გამოყოფილი ნომრების მაქსიმალური მნიშვნელობა. ამის შემდეგ სისტემაში საიდენტიფიკაციო ნომრის მნიშვნელობის მინიჭება პროცესებისთვის იწყება მინიმალური თავისუფალი ნომრიდან და გრძელდება ზრდადი მიმდევრობით.

პროცესის და მისი მშობელი პროცესის იდენტიფიკატორის მნიშვნელობის მისაღებად გამოიყენება სისტემური გამოძახებები getpid და getppid. მათი პროტოტიპები და მონაცემთა შესაბამისი ტიპები განსაზღვრულია სისტემურ ფაილებში <sys/types.h> და <unistd.h>.

```
#include<unistd.h>
#include<sys/types.h>
pid_t getpid(void); // პროცესის იდენტიფიკატორი
pid_t getppid(void); // მშობელი პროცესის იდენტიფიკატორი
```

2.3. პროცესის სიცოცხლის ციკლი

ოპერაციულ სისტემაში წარმოქმნილი პროცესი თავისი არსებობის მანძილზე (წარმოქმნიდან დასრულებამდე) გადის რამდენიმე დისკრეტულ მდგომარეობას. სხვადასხვა ოპერაციულ სისტემაში დისკრეტულ მდგომარეობათა რაოდენობა შეიძლება იყოს განსხვავებული. ნახ. 2.1-ზე ნაჩვენებია UNIX მსგავს ოპერაციულ სისტემებში პროცესების მდგომარეობის მოკლე დიაგრამა.

სისტემაში წარმოქმნილი ყოველი პროცესი იმყოფება მდგომარეობაში „დაბადება“. მას შემდეგ რაც ის აღიჭურვება მიმდინარე მომენტისთვის შესასრულებლად საჭირო რესურსებით ის გადადის მდგომარეობაში „მზადყოფნა“ და იკავებს რიგს პროცესორის მომსახურეობის მისაღებად. როგორც კი პროცესისთვის ხელმისაწვდომი გახდება პროცესორი ის მდგომარეობიდან „მზადყოფნა“ გადადის მდგომარეობაში „შესრულება“. როგორც ნახ. 2.1-დან ჩანს, მდგომარეობა „შესრულება“ დაყოფილია ორ ნაწილად: „შესრულება ბირთვის რეჟიმში“ და „შესრულება მომხმარებლის“

„რეჟიმში“. პროცესი მდგომარეობაში „შესრულება მომხმარებლის რეჟიმში“ ასრულებს მომხმარებლის გამოყენებით ინსტრუქციებს. მდგომარეობაში „შესრულება ბირთვის რეჟიმში“

სრულდება ოპერაციული სისტემის ბირთვის ინსტრუქციები მიმდინარე პროცესთან მიმართებაში (მაგალითად, სისტემური გამოძახების ან წყვეტის დამუშავებისას). მდგომარეობიდან „შესრულება“ პროცესი შეიძლება გამოვიდეს რამდენიმე მიზეზით:

- დროითი კვანტის ამოწურვის გამო.** ამ შემთხვევაში პროცესი გადადის მდგომარეობაში „მზადყოფნა“, იკავებს რიგს და ელოდება პროცესორის გამონთავისუფლებას;
- საჭირო რესურსის არარსებობა.** თუ პროცესი საჭიროებს გარკვეულ რესურსს (მეხსიერებას, შეტანა/გამოტანის მოწყობილობას, სხვა პროცესის მიერ განხორციელებული გამოთვლების შედეგი, მონაცემები და ა.შ.), რომელიც საჭიროა მისი შემდგომი შესრულებისთვის, მაშინ ოპერაციულ სისტემას ის გადაჰყავს მდგომარეობაში „ლოდინი“;
- დასრულება.** პროცესი დასრულების შემთხვევაში გადადის მდგომარეობაში „დასრულდა“. ოპერაციული სისტემა ანთავისუფლებს დასრულებული პროცესის მიერ დაკავებულ ყველა რესურს და მიღებულ შედეგს ანთავსებს მეხსიერების წინასწარ განსაზღვრულ მისამართზე იმ დრომდე სანამ მას არ მოითხოვს წარმომქმნელი პროცესი (მშობელი) ან არ მოხდება სისტემის გადატვირთვა.

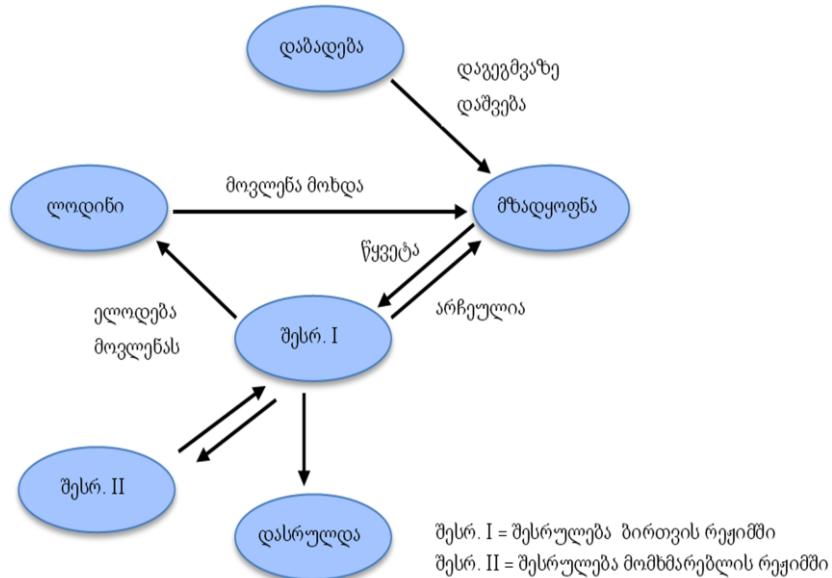
სისტემაში ამა თუ იმ პროცესის მიერ რესურსის გამონთავისუფლების ან შეტანა/გამოტანის მოწყობილობიდან წყვეტის წარმოქმნის შემდეგ სისტემა ამოწმებს „ლოდინი“ მდგომარეობაში მყოფ პროცესებს. იქ ეძებს პროცესს, რომელიც შესაძლებელია ელოდება გამონთავისუფლებულ რესურსს ან სისტემაში დაფიქსირებულ მოვლენას. ასეთი პროცესის არსებობის შემთხვევაში (რამდენიმე პროცესის შემთხვევაში პროცესი აირჩევა ნებისმიერად) პროცესი იღებს საჭირო რესურს ან მონაცემებს და თუ ის საკმარისია მისი შესრულების გაგრძელებისთვის, მაშინ პროცესი „ლოდინი“ მდგომარეობიდან გადადის მდგომარეობაში „მზადყოფნა“ და იკავებს რიგს. სხვა შემთხვევაში ელოდება სხვა საჭირო მოვლენის დადგომას.

2.4. პროცესების იერარქია

UNIX ოპერაციულ სისტემაში არსებული ყველა პროცესი გარდა ერთი პროცესისა (იდენტიფიკატორით 0, რომელიც წარმოადგენს სისტემის ძირითად პროცესს) შეიძლება წარმოქმნილი იყოს მხოლოდ რაიმე სხვა პროცესით. UNIX-მსგავს სისტემებში სხვა დანარჩენი პროცესის წარმომქმნელის როლში შეიძლება გამოდიოდეს პროცესი ნომრით 0 ან 1.

პროცესი, რომელიც წარმოქმნის ახალ პროცესს მშობელი პროცესი ეწოდება, ხოლო წარმოქმნილ პროცესს კი - შვილი პროცესი. თავის მხრივ, შვილი პროცესს შეუძლია წარმოქმნას საკუთარი შვილი პროცესები და ა.შ. ამ ფორმით პროცესების წარმოქმნისას იქმნება იერარქიული ხილები სტრუქტურა, რომელსაც პროცესების იერარქიული ხე ეწოდება. UNIX სისტემაში არსებული ყოველი პროცესი ერთმანეთთან დაკავშირებულია დამოკიდებულებით

ნახ. 2.1. პროცესის მდგომარეობის დიაგრამა



შესრ. I = შესრულება ბირთვის რეჟიმში

შესრ. II = შესრულება მომხმარებლის რეჟიმში

მშობელი - შვილი. როდესაც მშობელი პროცესი შვილ პროცესზე ადრე ასრულებს საკუთარ სიცოცხლეს (გადაგადის მდგომარეობაში „დასრულდა“) შვილი პროცესის PCB-ში, იერარქიის მთლიანობის შენარჩუნების მიზნით, მშობელი პროცესის იდენტიფიკატორის მნიშვნელობის (PPID – parent process identifier) როლში (ნაცვლად მისი რეალური მნიშვნელობისა) იწერება მნიშვნელობა 1, რომელიც შეესაბამება init¹ პროცესის იდენტიფიკატორს. რითაც ფაქტიურად init პროცესი „იშვილებს დაობლებულ“ პროცესს.

2.5. UNIX-ში პროცესის წარმოქმნა

ყოველი ოპერაციული სისტემა პროცესის შესაქმნელად საჭიროებს გარკვეულ მექანიზმს. Windows-ის სისტემაში ამ მიზნით გამოიყენება ფუნქცია createprocess. ამ ფუნქციას ჩვენ არ გამოიყენებთ და ამიტომ დავკამაყოფილდებით მხოლოდ მისი პროტოტიპის მოყვანით. ფუნქციის პროტოტიპს აქვს სახე:

```
BOOL CreateProcess(
    LPCWSTR     pszImageName,
    LPCWSTR     pszCmdLine,
    LPSECURITY_ATTRIBUTES  psaProcess,
    LPSECURITY_ATTRIBUTES  psaThread,
    BOOL         fInheritHandles,
    DWORD        fdwCreate,
    LPVOID       pvEnvironment,
    LPWSTR       pszCurDir,
    LPSTARTUPINFO          psiStartInfo,
    LPPROCESS_INFORMATION pProcInfo
);
```

UNIX მსგავს ოპერაციულ სისტემებში პროცესის წარმოსამნელად გამოიყენება სისტემური გამოძახება fork(). წარმოქმნილი პროცესი პრაქტიკულად წარმოადგენს მშობელი პროცესის სრულ ასლს. პროგრამისტის მიერ ცხადად თუ არ იქნა განსაზღვრული პროცესების საქმიანობა, მაშინ მშობელი და შვილი პროცესი გამოიყენებს ერთიდაიმავე რესურსებს და მონაცემებს და შესაბამისად დაკავდება ერთიდაიმავე საქმით.

fork() სისტემური გამოძახების პროტოტიპი

```
#include<unistd.h>
#include<sys/types.h>
pid_t fork(void);
```

სისტემური გამოძახების აღწერა

სისტემური გამოძახება fork() გამოიყენება UNIX ოპერაციულ სისტემაში ახალი პროცესის წარმოსამნელად. წარმოქმნილ პროცესს მშობელ-პროცესთან მიმართებაში ეცვლება შემდეგი პარამეტრების მნიშვნელობები:

- პროცესის იდენტიფიკატორი;
- მშობელი პროცესის იდენტიფიკატორი;
- დრო, დარჩენილი SIGALRM სიგნალის მისაღებად;
- მშობელი პროცესისთვის განკუთვნილი სიგნალები არ მიუვათ შვილ პროცესებს.

fork სისტემური გამოძახების წარმატებით დასრულების შემთხვევაში ის აბრუნებს ორ მნიშვნელობას: პირველი მშობელ პროცესში და მეორე წარმოქმნილ პროცესში. ახალი პროცესის წარმოქმნისას წარმოქმნილ პროცესში სისტემური გამოძახება აბრუნებს მნიშვნელობას 0, ხოლო მშობელ პროცესში კი - შვილი პროცესის იდენტიფიკატორის ტოლ მნიშვნელობას. რაიმე

¹ init არის სისტემური პროცესი, რომლის სიცოცხლის ხანგრძლივობაც განსაზღვრავს ოპერაციული სისტემის ფუნქციონირების დროს

მიზეზით fork სისტემური გამოძახების წარუმატებლად² დასრულების შემთხვევაში სისტემური გამოძახება მის მაინიცირებელ პროცესში აბრუნებს -1 -ის ტოლ მნიშვნელობას.

რადგანაც fork სისტემური გამოძახება წარმატებით დასრულების შემთხვევაში აბრუნებს ორ განსხვავებულ მნიშვნელობას, ამიტომ ჩვენ შეგვიძლია მშობელი და შვილი პროცესების მუშაობის დაგეგმვის მიზნით გამოვიყენოთ if-else კონსტრუქცია უშუალოდ fork სისტემური გამოძახების გამოყენების შემდეგ. fork სისტემური გამოძახების მუშაობის პრინციპიდან გამომდინარე ორჯერ შესრულდება მის შემდეგ მომავალი პროგრამული კოდი ანუ, შესრულდება if-else კონსტრუქციის ორივე ნაწილი: ერთხელ მშობლისთვის და ერთხელ შვილისთვის³. fork სისტემური გამოძახების გამოყენებით ახალი პროცესის წარმოქმნისა და მშობელი და შვილი პროცესის სამუშაოს დაგეგმვის შესაბამისი პროგრამული ფრაგმენტი გამოიყურება შემდეგნაირად:

```
pid_t pid = fork();
if(pid == -1){ // შეცდომა, პროცესი არ შეიქმნა
    ...
} else if (pid == 0) { // მშობელი
    ...
} else { // შვილი
    ...
}
```

2.6. პროცესის დასრულება

C ენაზე დაწერილი პროგრამა კორექტულად შეიძლება დასრულდეს ორი მეთოდით: პირველი, როცა ის სრულდება return ოპერატორის გამოყენებით; მეორე მეთოდი გამოიყენება პროგრამის ნებისმიერ ადგილას პროცესის დასრულების საჭიროებისას. ამისათვის გამოიყენება C ენის სტანდარტული ბიბლიოთეკის ფუნქციებიდან exit() ფუნქცია. ამ ფუნქციის შესრულებისას ხდება შეტანა/გამოტანის ნაწილობრივ შევსებული ბუფერის გასუფთავება შესაბამისი ნაკადების დახურვით, რის შემდეგაც ინიცირდება პროცესის მუშაობის შეწყვეტის და „დასრულდა“ მდგომარეობაში გადამყვანი სისტემური გამოძახება.

ფუნქციიდან დაბრუნება მიმდინარე პროცესში არ ხდება და, შესაბამისად, ფუნქციაც არაფერს არ აბრუნებს.

exit() ფუნქციის პარამეტრის მნიშვნელობა გადაეცემა ოპერაციული სისტემის ბირთვს და შემდეგ შეიძლება მიიღოს იმ პროცესმა, რომელმაც წარმოქმნა შვილი პროცესი.

exit() ფუნქციის პროტოტიპი

```
#include<stdlib.h>
void exit(int status);
```

ფუნქციის აღწერა

exit() ფუნქცია გამოიყენება პროცესის კორექტული დასრულებისთვის. ამ ფუნქციის გამოყენების შემდეგ ხდება შეტანა/გამოტანის ყველა ნაწილობრივ შევსებული ბუფერის გასუფთავება შესაბამისი ნაკადების დახურვით (ფაილების, pipe, FIFO, სოკეტების).

status - პროცესის კოდის დასრულების - პარამეტრის მნიშვნელობა გადაეცემა ოპერაციული სისტემის ბირთვს და შემდეგ შეიძლება მიღებული იქნას დასრულებული პროცესის წარმომქმნელი პროცესის მიერ.

² სისტემური გამოძახება წარუმატებლად შეიძლება დასრულდეს რამდენიმე მიზეზით: სისტემაში მიღწეულია პროცესების მაქსიმალური რაოდენობა ან მეხსიერებაში არაა ახალი პროცესის განსათავსებლად საკმარისი ადგილი

³ პროცესების შესრულების თანმიმდევრობა წინასწარ უცნობია

შევნიშნოთ, რომ main() ფუნქციაში ბოლო ინსტრუქციაზე მიღწევით ხდება exit ფუნქციის არაცხადი გამოძახება პარამეტრის მნიშვნელობით 0 (exit(0)).

თუ შვილი პროცესი საკუთარ სამუშაოს ამთავრებს მშობელ პროცესზე ადრე და მშობელ პროცესს ცხადად არ მიუთითებია, რომ ის საჭიროებს ინფორმაციის მიღებას დასრულებული პროცესის სტატუსზე, მაშინ დასრულებული პროცესი სისტემიდან არ იშლება და რჩება მდგომარეობაში „დასრულდა“ სანამ ან დასრულდება მშობელი პროცესი ან იმ დრომდე, სანამ მშობელი პროცესი არ მიიღებს მასზე ინფორმაციას. „დასრულდა“ მდგომარეობაში მყოფ პროცესებს UNIX მსგავს ოპერაციულ სისტემაში ეწოდებათ **ზომბი პროცესები** (zombie).

დასრულებულ პროცესზე ინფორმაციის მისაღებად მშობელ პროცესს შეუძლია გამოიყენოს სისტემური გამოძახება waitpid ან მისი შემოკლებული ფორმა wait. wait სისტემური გამოძახების შესრულებით ხორციელდება მშობელი პროცესის ბლოკირება იმ დრომდე სანამ არ დასრულდება წარმოქმნილი პროცესი. წარმოქმნილი პროცესის დასრულების შემდეგ მშობელ პროცესს გადაეცემა ინფორმაცია შვილი პროცესის სტატუსზე და მართვა უბრუნდება მშობელ პროცესს. wait სისტემური გამოძახებისგან განსხვავებით waitpid სისტემური გამოძახებით შესაძლებელია მშობელი პროცესის შესრულების ბლოკირება კონკრეტული იდენტიფიკატორის მქონე პროცესის დასრულებამდე.

wait() და waitpid() სისტემური გამოძახება

```
#include<sys/types.h>
#include< sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

სისტემური გამოძახების აღწერა

waitpid() სისტემური გამოძახებით ხდება მიმდინარე პროცესის ბლოკირება სანამ ან არ დასრულდება მის მიერ წარმოქმნილი pid იდენტიფიკატორის მქონე პროცესი.

- თუ pid > 0, მაშინ ხდება pid იდენტიფიკატორით პროცესის დასრულებაზე დალოდება
- თუ pid = 0, მაშინ ველოდებით იმ ჯგუფის ყველა პროცესის დასრულებას, რომელსაც მშობელი პროცესი მიეკუთვნება.
- თუ pid = -1, მაშინ ველოდებით ნებისმიერი წარმოქმნილი პროცესის დასრულებას.
- თუ pid < -1, მაშინ ველოდებით ნებისმიერი პროცესის ან პროცესების ჯგუფის დასრულებას, რომლის იდენტიფიკატორიც pid-ის აბსოლუტური მნიშვნელობის ტოლია.

თუ სისტემურმა გამოძახებამ აღმოაჩინა წარმოქმნილი დასრულებული პროცესი pid სპეციფიკური პარამეტრით, მაშინ ეს პროცესი იშლება სისტემიდან, ხოლო status პარამეტრით მითითებულ მისამართზე ინახება მის დასრულებაზე ინფორმაცია. status პარამეტრი შეიძლება მოცემული იყოს მნიშვნელობით NULL იმ შემთხვევაში, თუ ამ ინფორმაციას ჩვენთვის არააქვს მნიშვნელობა.

wait სისტემური გამოძახება წარმოადგენს waitpid სისტემური გამოძახების კერძო შემთხვევას პარამეტრების მნიშვნელობით: pid = -1, options = 0.

2.7. C ენაში main() ფუნქციის პარამეტრები

C ენაზე დაწერილი პროგრამის შესრულება იწყება main() ფუნქციის გამოძახებით. მისი სრული პროტოტიპი გამოიყურება შემდეგნაირად:

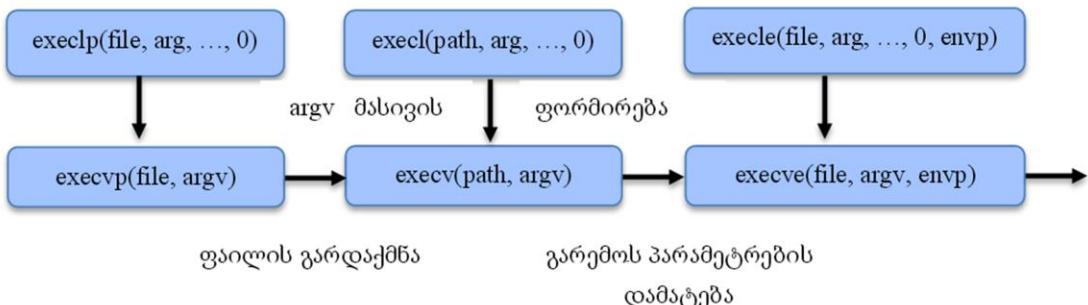
```
int main(int argc, char *argv[]);
```

სადაც პირველი პარამეტრი გამოსახავს ბრძანებათა ინტერპრეტატორის ველში აკრეფილი არგუმენტების რაოდენობას, ხოლო მეორე - არგუმენტებზე მიმთითებლების მასივს.

როდესაც ბირთვი ასრულებს C ენაზე დაწერილ პროგრამას main ფუნქციის გამოძახებამდე ის ასრულებს სპეციალურ პროცედურას. ამ პროცედურის მისამართი ეთითება შესრულებადი პროგრამის ფაილში, როგორც შესვლის წერტილი. პროცედურის მისამართს განსაზღვრავს გამოყენებული კომპილერის მიერ გამოძახებული კავშირების რედაქტორი. წინასწარი ჩატვირთვის პროცედურა ბირთვისგან ღებულობს ბრძანება-თა ველის პარამეტრებს და ცვლადი გარემოს მნიშვნელობებს. მხოლოდ ამის შემდეგ ხდება მიმართვა main ფუნქციაზე.

პროგრამა გარდა ბრძანებათა ინტერპრეტატორის პარამეტრებისა, ასევე, იღებს ცვლადი გარემოს პარამეტრების ჩამონათვალს. ბრძანებათა ინტერპრეტატორის პარამეტრების მსგავსად ცვლადი გარემოს პარამეტრებიც წარმოადგენენ მიმთითებელთა მასივს, რომელთაგან თითოეული უთითებს სიმბოლურ სტრიქონზე. სისტემაში არსებული გლობალური ცვლადით - environ, შესაძლებელია ცვლადი გარემოს პარამეტრების მნიშვნელობების მითითება. ნახ. 2.2-ზე გამოსახულია 5 პარამეტრიანი ცვლადი გარემოს მაგალითი.

ნახ. 2.3. *exec()* სისიტემური გამოძახების შესასრულებლად სხვადასხვა ფუნქციების ურთიერთკავშირი



ცვლადი გარემოს პარამეტრების მისათითებლად main ფუნქცია შეიცავდა მესამე არგუმენტს. სამ არგუმენტიანი main ფუნქცია გამოიყენებოდა უმეტეს UNIX მსგავს სისტემაში. ამ შემთხვევაში main ფუნქციის პროტოტიპის გააჩნია სახე

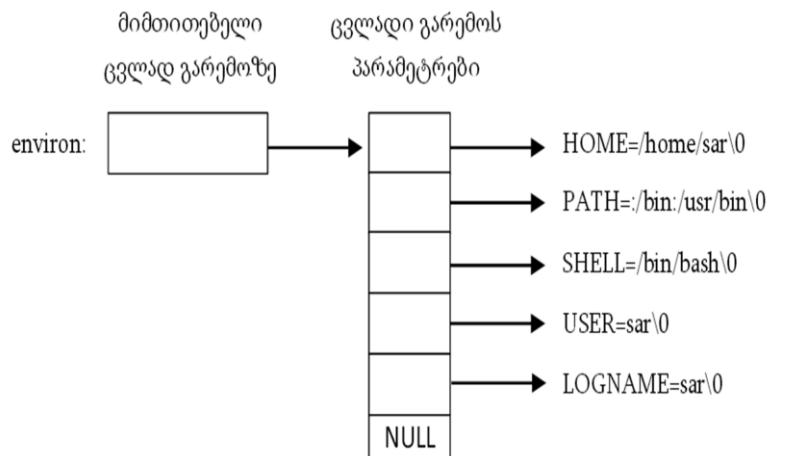
```
int main(int argc, char *argv[], char *envp[]);
```

ISO C სტანდარტიზაციით მიღებული შეთანხმების შედეგად main ფუნქციაში არ გამოიყენება მესამე არგუმენტი და ცვლად გარემოს პარამეტრების გადაცემა ხორციელდება გლობალური envirion ცვლადის გამოყენებით. ცვლადი პარამეტრების მნიშვნელობების ასეთნაირად გადაცემის მეთოდი არაფრით არ განსხვავდება main ფუნქციის მესამე არგუმენტით იმავე მნიშვნელობების გადაცემისგან. ამის გამო, POSIX სტანდარტშიც შენარჩუნებულია ორ არგუმენტიანი main ფუნქცია.

2.8. პროცესის მომხმარებლის კონტექსტის შეცვლა

პროცესის მომხმარებლის კონტექსტის შესაცვლელად გამოიყენება სისტემური გამოძახება exec(), რომლის გამოძახებაც უშუალოდ არ შეუძლია მომხმარებელს. exec() სისტემური გამოძახება

ნახ. 2.2. 5 პარამეტრიანი ცვლადი გარემო



ცვლის მიმდინარე პროცესის მომხმარებლის კონტექსტს რომელიმე შესრულებადი ფაილის შემცველობით და აყენებს პროცესორის რეგისტრების საწყის მნიშვნელობას (მათ შორის აყენებს პროგრამულ მთვლელს ჩასატვირთი პროგრამის საწყისზე). ეს სისტემური გამოძახება საკუთარი მუშაობისთვის მოითხოვს შესასრულებელი ფაილის სახელის, ბრძანებათა ველის არგუმენტების და გარემოს პარამეტრების მოცემას. სისტემური გამოძახების განსახორციელებლად პროგრამისტს შეუძლია გამოიყენოს ერთერთი შემდეგი ფუნქციებიდან: execl(), execvp(), execl() და execv(), execle(), execve(), რომლებიც ერთმანეთისგან განსხვავდებიან exec() სისტემური გამოძახების მუშაობისთვის აუცილებელი პარამეტრების მიწოდების მეთოდებით. მითითებული ფუნქციების ურთიერთკავშირი გამოსახულია ნახ. 2.3-ზე.

პროცესის მომხმარებლის კონტექსტის შეცვლის ფუნქციის პროტოტიპი

```
#include<unistd.h>
int execlp(const char *file, const char *arg0, ... , const char *argN, (char *) NULL);
int execvp(const char *file, char *argv[]);
int exect(const char *path, const char *arg0,... , const char *argN, (char *) NULL);
int execv(const char *path, char *argv[]);
int execle(const char *path, constchar *arg0, ... , const char *argN, (char *) NULL, char *envp[]);
int execve(const char *path, char *argv[], char *envp[]);
```

ფუნქციის აღწერა

ახალი პროგრამის ჩასატვირთად მიმდინარე პროცესის სისტემურ კონტექსტში გამოიყენება ურთიერთდაკავშირებული ფუნქციების ოჯახი, რომლებიც ერთმანეთისგან განსხვავდებიან პარამეტრების წარდგენის ფორმით. პარამეტრი

- file უთითებს იმ ფაილის სახელს, რომელიც უნდა იქნას ჩატვირთული;
- path უთითებს ჩასატვირთ ფაილამდე სრულ გზას.
- arg0, ..., argN წარმოადგენს ბრძანებათა ველის არგუმენტებზე მიმთითებლებს. შევნიშნოთ, რომ arg0 პარამეტრი უნდა უთითებდეს ჩასატვირთი ფაილის სახელზე.
- argv წარმოადგენს ბრძანებათა ველის პარამეტრებზე მიმთითებლების მასივს. მასივის საწყისი ელემენტი უნდა უთითებდეს ჩასატვირთი პროგრამის სახელზე, ხოლო მასივი უნდა მთავრდებოდეს ელემენტით, რომელიც შეიცავს მიმთითებელს NULL.
- envp არგუმენტი წარმოადგენს ცვლადი გარემოს პარამეტრებზე მიმთითებლების მასივს, რომელიც მოცემულია სტრიქონის სახით „ცვლადი = სტრიქონი“. ამ მასივის ბოლო ელემენტი უნდა შეიცავდეს მიმთითებელს NULL.

ვინაიდან ფუნქციის გამოძახება არ ცვლის მიმდინარე პროცესის სისტემურ კონტექსტს, ჩატვირთული პროგრამა მისი ჩამტვირთავი პროცესისგან მიიღებს შემდეგ ატრიბუტებს:

- პროცესის იდენტიფიკატორი;
- მშობელი-პროცესის იდენტიფიკატორი;
- პროცესის ჯგუფის იდენტიფიკატორი;
- სეანსის იდენტიფიკატორი;
- დრო, SIGALRM სიგნალის წარმოქმნამდე;
- მიმდინარე სამუშო დირექტორია;
- ფაილების შექმნის ნიდაბი;
- მომხმარებლის იდენტიფიკატორი;
- მომხმარებლის ჯგუფის იდენტიფიკატორი;
- სიგნალების ცხადი იგნორირება;
- ღია ფაილების ცხრილი;

გამოძახების განმახორციელებელი ფუნქციიდან პროგრამაში მნიშვნელობის დაბრუნება არ ხდება შესრულების წარმატებულად დასრულების შემთხვევაში და მართვა გადაეცემა ჩატვირთულ პროგრამას. წარუმატებელი შესრულების შემთხვევაში გამოძახების მაინიცირებელ პროგრამაში ბრუნდება უარყოფითი მნიშვნელობა

ვინაიდან პროცესის სისტემური კონტექსტი exec()-ის გამოძახებისას პრაქტიკულად უცვლელი რჩება, მომხმარებლისთვის სისტემური გამოძახების მეშვეობით დაშვებული პროცესის ატრიბუტების უმეტესი (UID, GID, PID, PPID და სხვა) ახალი პროგრამის ამუშავების შემდეგაც არ იცვლება.

მნიშვნელოვანია fork() და exec() სისტემურ გამოძახებებს შორის განსხვავების დაჭერა. fork() წარმოქმნის ახალ პროცესს, რომლის მომხმარებლის კონტექსტიც ემთხვევა მომხმარებლის მშობელი-პროცესის კონტექსტს. სისტემური გამოძახება exec() ცვლის მიმდინარე პროცესის მომხმარებლის კონტექსტს, მაგრამ არ წარმოქმნის ახალ პროცესს.

2.9. მაგალითები

- fork სისტემური გამოძახების გამოყენებით შევქმნათ ახალი პროცესი. მშობელი პროცესი დაელოდოს შვილი პროცესის დასრულებას და შემდეგ გააგრძელოს შესრულება.

```
#include<stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){
    pid_t a; // ცვლადი fork-ის მიერ დაბრუნებული მნიშნელობისთვის
    int status; // ცვლადი შვილი პროცესის მდგომარეობისთვის
    if ((a = fork()) == -1) {
        perror("პროცესი არ შეიქმნავ");
        exit(EXIT_FAILURE4);
    }

    if (a == 0){ //შვილი პროცესი
        cout << "შვ-იდენტ. " << (int) getpid() << "\t, მშ-იდენტ. ", (int) getppid() << endl;
    } else { //მშობელი პროცესი
        wait(&status); //მოხდა მშობელი პროცესის შეჩერება შვილი დასრულებამდე
        cout << "\n მშობელი პროცესი აგრძელებს შესრულებას\n";
        cout << "შვ-იდენტ. " << (int) getpid() << "\t, მშ-იდენტ. ", (int) getppid() << endl;
    }
    exit(EXIT_SUCCESS);
}
```

prog.2.1.c

- exec() სისტემური გამოძახების გამოყენებით დავგეგმოთ პროცესის საქმიანობა: შვილმა პროცესი დააკომპილიროს წინა პროგრამა (ფაილი prog.2.1.c), ხოლო მშობელმა პროცესმა კი შეასრულოს კომპილაციის შედეგის ფაილს (prog.2.1.out).

⁴ EXIT_FAILURE და EXIT_SUCCESS არის stdlib.h ბიბლიოთეკაში განსაზღვრული სპეციალური ცვლადები, რომლებიც შესაბამისად ღებულობენ რაიმე არანულოვან ან ნულოვან მნიშვნელობას

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){
    int status;           // ცვლადი შვილი პროცესის სტატუსის განსასაზღვრავად
    pid_t a = fork();
    if (a == -1){
        perror("პროცესი არ შეიქმნა\n");
        exit(EXIT_FAILURE);
    }
    if (a != 0){ // მშობელი პროცესი
        wait(&status);      // მშობელი ელოდება შვილის დასრულებას
        // მშობელი-პროცესი ახდენს 2.1.out ფაილის შესრულებას, რომელიც
        // წარმოადგენს 2.1.c ფაილის კომპილაციის შედეგს
        execl ("prog.2.1.out", "./2.1.out", NULL);
    } else { // შვილი-პროცესი ახდენს 2.1.c ფაილის კომპილაციას
        execl("/bin/gcc", "gcc", "prog.2.1.c", "-o", "prog.2.1.out",NULL);
    }
    exit(EXIT_SUCCESS);
}

```

prog.2.2.c

3. დაწერეთ პროგრამა რომელშიც მშობელ პროცესს ეყოლება 2 შვილი პროცესი და ერთერთ მათგანს ეყოლება საკუთარი შვილი პროცესი. "შვილიშვილმა" განახორციელოს სამუშაო დირექტორიაში 3 ქვედირექტორიის შექმნა, ხოლო მეორე შვილმა განახორციელოს ამ დირექტორიიებიდან ორის მესამეში გადატანა

```

#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){
    // ცვლადები პროცესის მიმდინარე სტატუსისთვის
    int st1, st2, st3;
    // ცვლადების პროცესების შექმნის შესამოწმებლად
    pid_t pro1, pro2, pro3;
    // შევქმნათ პირველი შვილი პროცესი
    if ((pro1 = fork()) == -1){
        cout << "პირველი პროცესი არ შეიქმნა\n";
        exit(EXIT_FAILURE);
    } else { // პირველი პროცესი შეიქმნა
        if (pro1 == 0) { // შვილმა უნდა შექმნას საკუთარი შვილი

```

```

if ((pro2 = fork()) == -1){ // შვილიშვილი არ წარმოიქმნა
    cout << "შვილიშვილი პროცესი არ წარმოიქმნა!\n";
    exit(EXIT_FAILURE);
} else { // შვილიშვილი შეიქმნა. დავგეგმოთ მისი სამუშაო
    if (pro2 == 0) {
        cout << "შვილიშვილი პროცესი!\n";
        execl(argv[1], argv[2], argv[3], argv[4], argv[5], NULL);
    } else { // დავგეგმოთ პირველი შვილი პროცესის სამუშაო
        cout << "პირველი პროცესი!\n";
        // პირველი პროცესი ვაიძულოთ დაელოდოს შვილის დასრულებას
        wait(&st3);
    }
}
} else { // მშობელი პროცესი. ვაიძულოთ ის დაელოდოს პირველი შვილის
// დასრულებას და შემდეგ შექმნას მეორე შვილი პროცესი
wait(&st2);
// შევქმნათ მეორე შვილი პროცესი
if ((pro3 = fork()) == -1) {
    cout << "მეორე შვილი პროცესი არ შეიქმნა!\n";
    exit(EXIT_FAILURE);
} else { // მეორე შვილი პროცესი შეიქმნა
    if (pro3 == 0){ // მეორე შვილი პროცესი
        // დავგეგმოთ მეორე შვილი პროცესის სამუშაო
        cout << "მეორე შვილი პროცესი!\n";
        execl(argv[6], argv[7], argv[3], argv[4], argv[5], NULL);
    }
    else { // დავგეგმოთ მშობელი პროცესის სამუშაო
        // მშობელი ვაიძულოთ დაელოდოს მეორე შვილის დასრულებას
        cout << "მშობელი პროცესი!\n";
        wait(&st1);
    }
}
}
}
exit(EXIT_SUCCESS);
}

```

prog.2.3.c

პროგრამაში გამოყენებული `argv[1]`, `argv[2]` და ა.შ. პარამეტრები უთითებენ კლავიატურიდან შეტანილ მნიშვნელობებს. კერძოდ, რადგანაც ჩვენ ჯერ გვინდა შევქმნათ რამდენიმე დირექტორია, ხოლო შემდეგ მოვახდინთ მათი გადაადგილება ერთერთ დირექტორიაში, ამიტომ პროგრამის შესასრულებლად ბრძანების შესრულება უნდა მოხდეს შემდეგი ფორმით

ბრძანება	<code>./prog.2.3</code>	<code>/bin/mkdir</code>	<code>mkdir</code>	<code>DIR1</code>	<code>DIR2</code>	<code>DIR</code>	<code>/bin/mv</code>	<code>mv</code>
არგუმენტები	<code>argv[0]</code>	<code>argv[1]</code>	<code>argv[2]</code>	<code>argv[3]</code>	<code>argv[4]</code>	<code>argv[5]</code>	<code>argv[6]</code>	<code>argv[7]</code>

სემინარი 3. პროცესების დაგეგმვა

გამოთვლითი სისტემა აღჭურვილია მრავალი რესურსით. ოპერაციულ სისტემაში წარმოქმნილი ყოველი პროცესი მიმართავს მას შესასრულებლად საჭირო რესურსების გამოყოფაზე მოხოვნით. ოპერაციული სისტემის ყველაზე მნიშვნელოვან რესურსს წარმოადგენს პროცესორი, რომელიც თანაბრად უნდა გამოეყოს სისტემაში წარმოქმნილ ყველა პროცესს. იმისთვის, რომ ოპერაციულმა სისტემამ უზრუნველყოს პროცესორის გამოყენების თანაბრობა ის მიმართავს მასში არსებულ პროგრამას, ე.წ. **პროცესორის დამგეგმვას**, რომელიც განსაზღვრავს თუ რა თანმიმდევრობით და რა დროითი შუალედით უნდა მოხდეს პროცესებისთვის პროცესორის გამოყოფა.

ყოველი პროცესი, რომელიც იქმნება პროგრამის შესრულების მომენტში, შესასრულებლად საჭიროებს გარკვეულ დროით შუალედს. ოპერაციულ სისტემაში წარმოქმნისას ეს დროითი შუალედი შესაძლებელია არ იყოს საკმარისი პროცესის რეალური შესრულებისთვის¹. შეიძლება ამის არსებობდეს სხვადასხვა მიზეზები: სისტემის დატვირთულობა, გამოყენებული ალგორითმი და ა.შ. ოპერაციული სისტემის მთავარ ამოცანას წარმოადგენს მაქსიმალურად მიაახლოვოს პროცესის შესასრულებლად საჭირო დრო იმ დროსთან², რომელიც დასჭირდება მას შესასრულებლად სისტემაში წარმოქმნის შემდეგ. რაც უფრო ნაკლებია ზემოხსენებულ ორ დროს შორის სხვაობა მით უფრო კარგია გამოყენებული აგორითმი.

ოპერაციული სისტემების არსებობის პირველი დღიდან შეიქმნა პროცესორის დაგეგმვის მრავალი ალგორითმი. ნაწილი ამ ალგორითმებიდან დღეს არ გამოიყენება და წარმოადგენს ისტორიის საკუთრებას, მეორე ნაწილმა კი განიცადა გარკვეული ცვლილება და აქტიურად გამოიყენება ამა თუ იმ სისტემაში (ტრანზაქციების, დროის გაყოფის, ინტერაქტიულ და ა.შ.).

შევნიშნოთ, რომ ოპერაციულ სისტემაში წარმოქმნილი ყოველი პროცესი ძირითადად დაკავებულია ორი ტიპის საქმიანობით: **აქტიური გამოთვლებით და აქტიური შეტანა/გამოტანით**. ორივე სახის საქმიანობის საჭიროების მქონე პროცესების შესრულებისას შეიძლება რამდენიმეჯერ მოხდეს მათი შეჩერება მონაცემების შეტანა/გამოტანის მიზნით, რაც ზრდის პროცესის შესასრულებლად საჭირო დროს. სიმარტივისთვის ქვემოთ განხილულ ალგორითმებში ვიგულისხმებთ, რომ პროცესი დასრულდება მის შესასრულებლად მითითებულ დროში და ის ერთდროულად არაა დაკავებული ორივე საქმიანობით.

განვიხილოთ ეს ალგორითმები

3.1. FCFS (First Come – First Served)

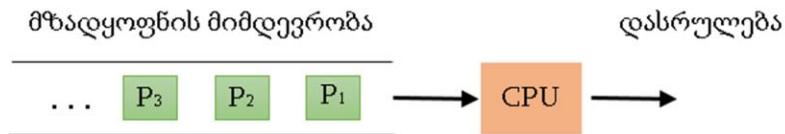
დაგეგმვის ყველაზე მარტივ ალგორითმს წარმოადგენს ალგორითმი FCFS (პირველი შემოვიდა - პირველი მომსახურდება) (ნაბ. 3.1). ამ ალგორითმის გამოყენებისას სისტემაში იქმნება მზაყოფნის ცხრილი, რომელშიც პროცესები თავსდებიან სისტემაში წარმოქმნის თანმიმდევრობით. სისტემაში წარმოქმნილი ყოველი ახალი პროცესი ამ მიმდევრობაში ემატება ბოლოდან. შესრულებას იწყებს ცხრილის თავში მყოფი პირველივე პროცესი. პროცესისთვის პროცესორის გამოყოფა ხდება მის შესასრულებლად საჭირო დროითი შუალედით. პროცესისგან პროცესორის გამონთავისუფლება მოხდება მხოლოდ მისი დასრულების ან გარკვეული მიზეზით (შეტანა/გამოტანის ოპერაციის ლოდინის გამო) მისი ბლოკირების შემთხვევაში. ბლოკირებული პროცესი სისტემაში მის გასაგრძელებლად საჭირო მოვლენის (მაგალითად, გამონთავისუფლდა

¹ შესრულება, რომლის დროსაც ოპერაციულ სისტემაში არსებული ყოველი რესურსი ხმარდება მიმდინარე პროცესს და ამ რესურსების ჩამორთმევა მისთვის არ ხდება

² შესასრულებლად საჭირო პროცესორული დრო

შეტანა/გამოტანის მოწყობილობა) დადგომის შემდეგ (თუ ის საკმარისია) გამოდის ბლოკირების მდგომარეობიდან და თავსდება მზადყოფნის ცხრილის ბოლოში.

ნახ. 3.1. დაგეგმვა FCFS პრინციპის გამოყენებით



განვიხილოთ მაგალითი. ვთქვათ პროცესები მოცემულია შემდეგი ცხრილით

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4

შევნიშნოთ, რომ შემდეგში თუ პროცესების შესრულებასთან ერთად არ იქნება მითითებული შესრულების დრო, მაშინ ვიგულისხმებთ, რომ ყველა ჩამოთვლილი პროცესი ერთდროულად გვაქვს სისტემაში. ერთი პროცესორის შემთხვევაში რეალურად ოპერაციულ სისტემაში ერთდროულად შეუძლებელია წარმოიქმნას ერთზე მეტი პროცესი.

რადგანაც ყველა პროცესი გვაქვს სისტემაში, ამიტომ შესასრულებლად მათი არჩევა მოხდება მათი ინდექსების მიხედვით - პირველი P₁ პროცესი, მეორე P₂ პროცესი და ა.შ. რადგანაც FCFS ალგორითმი ყოველ პროცესს შესასრულებლად გამოუყოფს საჭირო დროს სრულად, ამიტომ პროცესების შესრულებისთვის გვექნება შემდეგი ცხრილი

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
დასრულებისთვის საჭირო დრო	5	12	15	21	25
ლოდინის დრო	0	5	12	15	21

უფრო დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁	+	+	+	+	+																				
P ₂	-	-	-	-	-	+	+	+	+	+	+	+													
P ₃	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+										
P ₄	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+					
P ₅	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	

სადაც სიმბოლო '+' აღნიშნავს პროცესორზე პროცესის შესრულების მომენტს, ხოლო სიმბოლო '-' კი ლოდინს. ალგორითმის შესაფასებლად ჩვენ დაგვჭირდება დავითვალოთ ლოდინის („მინუსიების რაოდენობა“) და შესასრულებლად საჭირო (მისი შექმნიდან დასრულებამდე დროების საშუალო.

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

$$\text{ლოდინის საშუალო } \text{არის } \rightarrow (0+5+12+15+21)/5 = 10.6$$

$$\text{შესრულების დროის საშუალო } \rightarrow (5+12+15+21+25)/5 = 15.6$$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 10.6 დროითი ერთეულით ლოდინი, ხოლო შესრულებას კი დასჭირდა 15.6 დროითი ერთეული.

თუ განხილულ ამოცანაში შემოვიღებთ პროცესის გამოჩენის (შექმნის) დროს, მაშინ ამოცანა უმნიშველოდ გართულდება. ვთქვათ, პროცესების ცხრილს აქვს სახე

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
გამოჩენის დრო	1	4	0	7	2

რადგანაც ამ შემთხვევაში ჩვენი ცხრილი დამატებით შეიცავს პროცესების გამოჩენის დროს, ამიტომ პროცესები მზადყოფნის ცხრილში ადგილს დაიკავებენ გამოჩენის დროის მიხედვით. პირველი P₃ პროცესი, მეორე P₁ პროცესი და ა.შ. ამ მიმდევრობით მოხდება მათი შესრულებაც.

პროცესების შესრულების დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
P ₁		-	-	+	+	+	+	+																		
P ₂				-	-	-	-	-	-	-	-	-	+	+	+	+	+	+								
P ₃	+	+	+																							
P ₄							-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	
P ₅			-	-	-	-	-	-	+	+	+	+														

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

ლოდინის საშუალო არის $\rightarrow (2+8+0+12+6)/5 = 5.6$

შესრულების დროის საშუალო $\rightarrow (7+15+3+18+10)/5 = 8.6$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 5.6 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 8.6 დროითი ერთეული.

როგორც ვხედავთ ამ შემთხვევაში მიიღწევა უკეთესი შედეგი.

დამოუკიდებელი სამუშაო

1. დაგეგმვის FCFS ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების და ლოდინის დროის საშუალო, თუ ყველა პროცესი სისტემაში არსებობს ერთდროულად.

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	6	1	5	2	8

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	8	10	7	2	15

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1

2. დაგეგმვის FCFS ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების და ლოდინის დროის საშუალო, თუ პროცესები სისტემაში გამოჩნდნენ სხვადასხვა დროს:

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	2	8
გამოჩენის დრო	0	2	10	5	1

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	8	10	7	2	15
გამოჩენის დრო	2	3	0	5	10

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1
გამოჩენის დრო	2	0	3	7	5

3.2. SJF (Short Job First)

როგორც ზემოთ განხილული ამოცანების შემთხვევაში ვნახეთ, ოპერაციულ სისტემაში შესასრულებლად ნაკლები დროის საჭიროების მქონე პროცესს შეიძლება მოუწიოს დიდი დროით ლოდინი შესასრულებლად. თუ შესასრულებლად ნაკლები დროის საჭიროების მქონე პროცესები შესრულდებოდნენ თავიდან და შემდეგ მეტი დროის საჭიროების მქონე პროცესები მაშინ, ცხადია, რომ ლოდინის და შესრულების დროების საშუალო უფრო ნაკლები იქნებოდა.

ოპერაციულ სისტემისთვის ხშირად უცნობია პროცესების შესასრულებლად საჭირო დრო. თუ იარსებებდა პროცესების შესასრულებლად საჭირო დროის დადგენის რაიმე მექანიზმი, მაშინ პროცესების დაგეგმვის ამოცანა ბევრად გამარტივდებოდა. SJF ალგორითმი (პირველი მოკლე ამოცანა) აგებულია ამ პრინციპით. SJF ალგორითმის გამოყენება გულისხმობს, რომ წინასწარ ცნობილია პროცესების შესრულების დრო და მასში გამოყენებულ მზადყოფნის ცხრილში პროცესები განთავსებულია მათი შესრულების დროის მიხედვით - ნაკლები პროცესორული დროის საჭიროების მქონე პროცესი განთავსებულია ცხრილის თავში. დამგეგმავის მიერ შესასრულებლად პირველი აირჩევა ცხრილის თავში განთავსებული პროცესი. სისტემაში წარმოქმნილი ყოველი პროცესი ფასდება მისი შესრულების დროით და მზადყოფნის ცხრილში შესრულების დროის მიხედვით თავსდება შესაბამის ადგილას.

განვიხილოთ მაგალითი:

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4

რადგანაც ყველა პროცესი თავიდანვე გვაქვს სისტემაში, ამიტომ დამგეგმავის მიერ მათი შესარულებლად არჩევა მოხდება შესრულების დროის მიხედვით - პირველი P₃ პროცესი, მეორე P₅ პროცესი და ა.შ. რადგანაც SJF ალგორითმი ყოველ პროცესს გამოუყოფს მის შესასრულებლად საჭირო დროს, ამიტომ პროცესების შესრულებისთვის გვექნება შემდეგი დეტალიზირებული ცხრილი

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁	-	-	-	-	-	-	-	+	+	+	+	+													
P ₂	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	
P ₃	+	+	+																						
P ₄	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+							
P ₅	-	-	-	+	+	+	+																		

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

$$\text{ლოდინის საშუალო } \text{არის} \rightarrow (7+18+0+12+3)/5 = 8$$

$$\text{შესრულების დროის საშუალო} \rightarrow (12+25+3+18+7)/5 = 13$$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 8 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 13 დროითი ერთეული.

როგორც ვხედავთ SJF ალგორითმი FCFS ალგორითმთან შედარებით (თუ პროცესები თავიდანვე იარსებებენ სისტემაში) იძლევა უკეთეს შედეგს.

თუ განხილულ ამოცანას დავამატებთ გამოჩენის დროს ის გარკვეულწილად გართულდება. ვთქვათ, გვაქვს პროცესების შემდეგი ცხრილი

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
გამოჩენის დრო	1	4	0	7	2

რადგანაც ამ შემთხვევაში ჩვენი ცხრილი დამატებით შეიცავს პროცესების გამოჩენის დროს, ამიტომ პროცესები მზადყოფნის ცხრილში ადგილს დაიკავებენ შესრულების დროის მიხედვით. პირველი P₃ პროცესი. მიუხედავად იმისა, P₁ პროცესი სისტემაში შეიქმნა P₅ პროცესზე ადრე ის შესასრულებლად საჭიროებს P₅ პროცესის შესასრულებლად საჭირო დროზე მეტს, ამიტომ მეორე სისტემაში შესრულდება P₅ პროცესი და ა.შ.

პროცესების შესრულების დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁		-	-	-	-	-	-	+	+	+	+	+													
P ₂				-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	
P ₃	+	+	+																						
P ₄							-	-	-	-	-	-	+	+	+	+	+	+							
P ₅			-	+	+	+	+																		

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

$$\text{ლოდინის საშუალო } \text{არის} \rightarrow (6+14+0+5+1)/5 = 5.2$$

$$\text{შესრულების დროის საშუალო } \rightarrow (11+21+3+11+5)/5 = 10.2$$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 5.2 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 10.2 დროითი ერთეული.

დამოუკიდებელი სამუშაო

1. დაგეგმვის SJF ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების და ლოდინის დროის საშუალო, თუ ყველა პროცესი სისტემაში არსებობს ერთდროულად:

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	6	7	1	2	8

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	10	7	2	3

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	15	2	1

2. დაგეგმვის SJF ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების და ლოდინის დროის საშუალო, თუ პროცესები სისტემაში გამოჩნდნენ სხვადასხვა დროს:

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	2	8
გამოჩენის დრო	0	2	2	5	1

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	8	10	7	2	15
გამოჩენის დრო	2	3	0	5	0

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1
გამოჩენის დრო	2	0	3	7	5

3.3. პრიორიტეტული SJF

პრიორიტეტის შემოღებით სისტემაში შესაძლებელია პროცესების დაყოფა პრიორიტეტების ჯგუფების მიხედვით. პრიორიტეტის მნიშვნელობა შეიძლება იყოს გარკვეული მთელი რიცხვითი მნიშვნელობა. ის უპირატესობაში აყენებს გარკვეულ პროცესებს სხვა პროცესებთან მიმართებაში. პრიორიტეტის დანიშვნა შესაძლებელია მოხდეს ორი მეთოდით: სტატიკურად და

დინამიურად. სტატიკური პრიორიტეტი ინიშნება პროგრამისტის ან სისტემური ადმინისტრატორის მიერ. მისი შეცვლა პროცესის დასრულებამდე შეუძლებელია. დინამიური პრიორიტეტი ინიშნება სისტემის მიერ გამოყენებული ალგორითმით და ის შესაძლებელია გარკვეული მოვლენის დადგომის შემდეგ (მაგალითად, პროცესი დასრულდა) შეიცვალოს.

პრიორიტეტის გამოყენებელი ალგორითმი შეიძლება იყოს ორი სახის: **ალგორითმი პრიორიტეტული გაძევებით** და **ალგორითმი პრიორიტეტული გაძევების გარეშე**. პრიორიტეტული გაძევების გარეშე (**გაუძევებელი**) ალგორითმის გამოყენებისას სისტემაში ახალი მაღალპრიორიტეტული პროცესის წარმოქმნის შემთხვევაში მიმდინარე პროცესი, რომელსაც დაკავებული აქვს პროცესორი (მიუხედავად იმისა როგორია მისი პრიორიტეტი წარმოქმნილ პროცესთან მიმართებაში), აგრძელებს შესრულებას და მხოლოდ მისი დასრულების შემდეგ შეეძლება მაღალპრიორიტეტულ პროცესს შესრულება. პრიორიტეტული გაძევებით (გაძევებადი) ალგორითმის გამოყენებისას სისტემაში ახალი მაღალპრიორიტეტული პროცესის წარმოქმნის შემთხვევაში მიმდინარე პროცესი, რომელსაც დაკავებული აქვს პროცესორი (თუ მისი პრიორიტეტი წარმოქმნილ პროცესთან მიმართებაში დაბალია), იქნება შეჩერებული სისტემის მიერ და შესრულებას დაიწყებს მაღალპრიორიტეტულ პროცესი.

SJF ალგორითმი შეიძლება იყოს როგორც გაძევებადი ისე გაუძევებელი. მასში პრიორიტეტი შეიძლება განსაზღვრული იყოს როგორც სტატიკურად (გარკვეული რიცხვითი მნიშვნელობის მინიჭებით), ასევე დინამიურად (შესრულების დროის მიხედვით³).

განვიხილოთ მაგალითები.

1. გაუძევებელი SJF ალგორითმი სტატიკური პრიორიტეტით (გამოჩენის დროის გარეშე)

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
პრიორიტეტი	0	3	4	2	1

შევნიშნოთ, რომ ამ შემთხვევაში ალგორითმი გაძევებადი იქნება თუ გაუძეველი არსებითი მნიშვნელობა არ გააჩნია, ვინაიდან ყველა პროცესი თავიდანვე გვაქვს სისტემაში.

რადგანაც გვაქვს პრიორიტეტული ალგორითმი, ამიტომ ამ შემთხვევაში მათი შესრულება მოხდება პრიორიტეტის მნიშვნელობის მიხედვით. თუ გვექნებოდა ერთნაირი პრიორიტეტის მქონე ორი ან მეტი პროცესი, მაშინ ფიქსირებული პრიორიტეტის ფარგლებში უპირატესობა მიენიჭებოდა მათი შესრულების დროს. განსახილველ მაგალითში პრიორიტეტის მნიშვნელობა არ მეორდება, ამიტომ ისინი შესრულდებიან ერთიმეორის მიყოლებით პრიორიტეტის მნიშვნელობის მიხედვით. შესაბამისად პროცესები შესასრულებლად არჩეული იქნება მიმდევრობით პირველი P₁ პროცესი, მეორე P₅ პროცესი და ა.შ., ხოლო პროცესების შესრულებისთვის გვექნება შემდეგი დეტალიზირებული ცხრილი

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁	+	+	+	+	+																				
P ₂	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+	+			
P ₃	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	
P ₄	-	-	-	-	-	-	-	-	+	+	+	+	+	+											
P ₅	-	-	-	-	-	+	+	+	+																

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

$$\text{ლოდინის საშუალო არის } \rightarrow (0+15+22+9+5)/5 = 10.2$$

³ ნაკლები შესრულების დრო მაღალი პრიორიტეტი

შესრულების დროის საშუალო $\rightarrow (5+22+25+15+9)/5 = 15.2$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 10.2 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 15.2 დროითი ერთეული.

2. გაუძვებელი SJF ალგორითმი სტატიკური პრიორიტეტით (გამოჩენის დროით)
ვთქვათ, გვაქვს პროცესების შემდეგი ცხრილი

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
გამოჩენის დრო	1	4	0	7	2
პრიორიტეტი	0	3	4	2	1

რადგანაც ალგორითმი გაუძვებელია, ამიტომ პროცესი, რომელიც დაიკავებს პროცესორს არ გამოათავისუფლებს მას დასრულებამდე, მიუხედავად იმისა მისი შესრულების მომენტში გამოჩნდა თუ არა მაღალპრიორიტეტული პროცესი. პროცესის დასრულების შემდეგ პროცესორს იკავებს მოცემული მომენტისთვის არსებული მაღალპრიორიტეტული პროცესი. ამრიგად, პროცესები შესრულდებიან მიმდევრობით: რადგანაც სისტემაში პირველი გამოჩნდა P₃ პროცესი, ამიტომ ის დაიწყებს შესრულებას და პროცესორს არ გამოათავისუფლებს სანამ არ დასრულდება. მისი დასრულების (3 დროითი ერთეულის) შემდეგ სისტემაში უკვე გვაქვს ორი პროცესი (P₁ და P₅). რადგანაც P₁ პროცესს აქვს მაღალი პრიორიტეტი (0), ვიდრე P₅ პროცესს (1), ამიტომ შესრულებას დაიწყებს P₁ პროცესი. და ა.შ.

პროცესების შესრულების დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁		-	-	+	+	+	+	+																	
P ₂					-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	
P ₃	+	+	+																						
P ₄							-	-	-	-	-	+	+	+	+	+	+								
P ₅			-	-	-	-	-	-	+	+	+	+													

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

ლოდინის საშუალო არის $\rightarrow (2+14+0+5+6)/5 = 5.4$

შესრულების დროის საშუალო $\rightarrow (7+21+3+11+10)/5 = 8.6$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 5.4 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 8.6 დროითი ერთეული.

3. გაძვებადი SJF ალგორითმი სტატიკური პრიორიტეტით (გამოჩენის დროით)

ვთქვათ, გვაქვს პროცესების შემდეგი ცხრილი

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
გამოჩენის დრო	1	4	0	7	2
პრიორიტეტი	0	3	4	2	1

რადგანაც ალგორითმი გაძვებადია, ამიტომ პროცესი, რომელიც დაიკავებს პროცესორს გამოათავისუფლებს მას (დასრულებამდე) მხოლოდ სისტემაში მაღალპრიორიტეტული პროცესის გამოჩენის შემთხვევაში. პროცესორს დაიკავებს მაღალპრიორიტეტული პროცესი და დაიწყებს შესრულებას. ამრიგად, პროცესები შესრულდებიან მიმდევრობით: რადგანაც სისტემაში პირველი გამოჩნდა P₃ პროცესი, ამიტომ ის დაიწყებს შესრულებას. 1 დროითი ერთეულის შემდეგ სისტემაში გამოჩნდება პროცესი P₁, რომელსაც გააჩნია მაღალი პრიორიტეტი.

P₃ პროცესი დროებით შეწყვეტს შესრულებას და შესრულებას გააგრძლებს P₁ პროცესი. 2 დროითი ერთეულის შემდეგ სისტემაში გამოჩნდება P₅ პროცესი, რომელსაც P₁ პროცესზე დაბალი პრიორიტეტი გააჩნია, ამიტომ შესრულებას გააგრძლებს P₁ პროცესი და ა.შ.

პროცესების შესრულების დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁		+	+	+	+	+																			
P ₂					-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	+		
P ₃	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+
P ₄								-	-	-	+	+	+	+	+	+									
P ₅			-	-	-	-	+	+	+	+															

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

ლოდინის საშუალო არის $\rightarrow (0+12+22+3+4)/5 = 8.2$

შესრულების დროის საშუალო $\rightarrow (5+19+25+9+8)/5 = 13.2$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 8.2 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 13.2 დროითი ერთეული.

4. გაძევებადი SJF ალგორითმი დინამიური პრიორიტეტით (გამოჩენის დროით). პრიორიტეტის როლში აღებული იქნება პროცესისთვის მიმდინარე მომენტში შესასრულებლად საჭირო დრო.

ვთქვათ, გვაქვს პროცესების შემდეგი ცხრილი

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
გამოჩენის დრო	1	4	0	7	2

რადგანაც პრიორიტეტის როლში აღებულია შესრულების დრო, ამიტომ დროის ყოველ მომენტში შესაძლებელია იცვლებოდეს პროცესის მიმდინარე პრიორიტეტი და პროცესის შესასრულებლად არჩევისას დამგეგმავმა უნდა გაითვალისწინოს ის. ასევე, გაძევებადობის გამო სისტემაში წარმოქმნილი მაღალპრიორიტეტული პროცესი ჩაანაცვლებს შესრულებად დაბალ-პრიორიტეტულ პროცესს. ამრიგად, პროცესები შესრულდებიან მიმდევრობით: რადგანაც სისტემაში პირველი გამოჩენდა P₃ პროცესი, ამიტომ ის დაიწყებს შესრულებას. მაგალითის მიხედვით პროცესი P₁ შესასრულებლად საჭიროებს ყველაზე ნაკლებ დროს. ამიტომ ის დაიკავებს პროცესორს და არ გამოათავისუფლებს მას სანამ არ დასრულდება. 3 დროითი ერთეულის შემდეგ სისტემაში გვაქვს ორი პროცესი (P₁ და P₅). შესრულების დროის მიხედვით P₅ პროცესი საჭიროებს ნაკლებ დროს, ამიტომ ის უპირატესია (მაღალპრიორიტეტულია) და დაიწყებს შესრულებას და ა.შ.

პროცესების შესრულების დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁		-	-	-	-	-	-	+	+	+	+	+													
P ₂					-	-	-	-	-	-	-	-	-	-	-	-	-	-	+	+	+	+	+	+	
P ₃	+	+	+																						
P ₄								-	-	-	-	-	+	+	+	+	+	+							
P ₅			-	+	+	+	+	+																	

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

ლოდინის საშუალო არის $\rightarrow (6+14+0+5+1)/5 = 5.2$

შესრულების დროის საშუალო $\rightarrow (11+21+3+11+5)/5 = 10.2$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 5.2 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 10.2 დროითი ერთეული.

დამოუკიდებელი სამუშაო

1. დაგეგმვის პრიორიტეტული SJF ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების დროის საშუალო, თუ ყველა პროცესი სისტემაში არსებობს ერთდროულად (მაღალ პრიორიტეტს შეესაბამება ნაკლები რიცხვითი მნიშვნელობა):

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	2	8
პრიორიტეტი	1	0	2	2	1

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1
პრიორიტეტი	2	0	3	7	5

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	7	2	1	10	11
პრიორიტეტი	0	1	3	7	5

2. დაგეგმვის პრიორიტეტული SJF ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების დროის საშუალო, თუ პროცესები სისტემაში გამოჩნდნენ სხვადასხვა დროს:

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	2	8
გამოჩენის დრო	0	2	8	3	1
პრიორიტეტი	1	0	2	2	1

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1
გამოჩენის დრო	0	5	7	1	2
პრიორიტეტი	2	0	3	7	5

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	7	2	1	10	11
გამოჩენის დრო	1	5	1	0	3
პრიორიტეტი	0	1	3	7	5

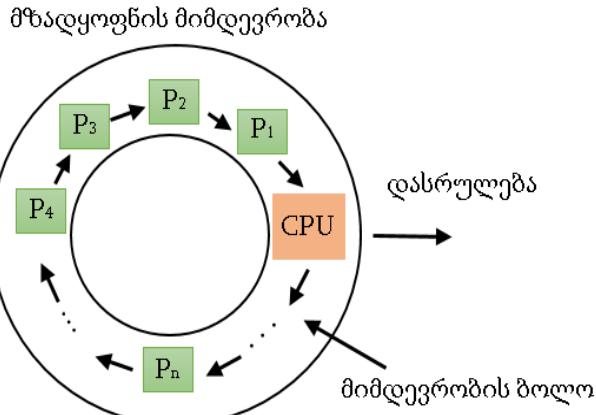
3. დაგეგმვის პრიორიტეტული გამევებადი SJF ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების დროის საშუალო, თუ პროცესები სისტემაში გამოჩნდნენ სხვადასხვა დროს. პრიორიტეტის როლში აიღეთ პროცესის შესასრულებლად საჭირო მიმდინარე დრო:

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	2	5
გამოჩენის დრო	0	3	8	2	1

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1
გამოჩენის დრო	0	5	7	1	4
პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	7	2	1	10	11
გამოჩენის დრო	1	5	1	0	3

3.4. ციკლური დაგეგმვა (RR - Round Robin)

ყველა ზემოთ განხილულ ალგორითმში თუ პროცესი მიზანმიმართულად ან უნებლიერ ხელში ჩაიგდებს პროცესორს და ჩაიციკლება, მაშინ იქიდან გამოსვლა საჭიროებს დამატებით მექანიზმს, რომელსაც ალგორითმი არ ითვალისწინებს. გარდა ამისა, ისინი მუშაობენ მხოლოდ ერთ პროცესთან სანამ ეს უკანასკნელი არ დასრულდება და ნებაყოფლობით არ დაუთმობს პროცესორს სხვა პროცესს.



ნახ. 3.2. დაგეგმვა RR ალგორითმის გამოყენებით

პროცესორის დაგეგმვის ერთერთ ძველ, მარტივ, სამართლიან და ხშირად გამოყენებად ალგორითმს წარმოადგენს ციკლური დაგეგმვის ალგორითმი (ნახ. 3.2), რომელიც გულისხმობს პროცესებისთვის პროცესორის გამოყოფას გარკვეული დროითი შუალედით (დროითი კვანტით). ალგორითმში მხარდაჭერილია მზადყოფნის ერთი ცხრილი, რომელშიც მზადყოფნის მდგომარეობაში მყოფი პროცესები ემატება ბოლოდან. პროცესი, რომელსაც შესასრულებლად არ ეყო დროითი კვანტი, იძულებით ჩამოერთმევა პროცესორი და ის გადაეცემა შემდეგ შესასრულებელ პროცესს, ხოლო მიმდინარე პროცესი გადაინაცვლებს მზადყოფნის ცხრილის ბოლოში. თუ პროცესი ჩაეტია დროით კვანტში (დასრულდა), მაშინ ის ნებაყოფლობით აბრუნებს პროცესორს, რომელიც გადაეცემა შემდეგ შესასრულებელ პროცესს. პროცესორის გამოყოფა ხდება ციკლურად პროცესებისთვის სანამ მზადყოფნის ცხრილში არსებობს ერთი მაინც პროცესი.

განვიხილოთ მაგალითი. ვთქვათ, დროითი კვანტი 2-ის ტოლია.

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4

რადგანაც ყველა პროცესი გვაქვს სისტემაში, ამიტომ დამგეგმავის მიერ მათი შესარულებლად არჩევა მოხდება მათი მიმდევრობით დროითი კვანტით 2 - პირველი P₁ პროცესი, მეორე P₂ პროცესი და ა.შ.

პროცესების შესრულებისთვის გვექნება შემდეგი დეტალიზირებული ცხრილი

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁	+	+	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	-	+						
P ₂	-	-	+	+	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	+	+	+	-	+	
P ₃	-	-	-	-	+	+	-	-	-	-	-	-	+												
P ₄	-	-	-	-	-	-	+	+	-	-	-	-	-	+	+	-	-	-	-	-	+	+	+		
P ₅	-	-	-	-	-	-	-	-	+	+	-	-	-	-	-	-	+	+							

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

$$\text{ლოდინის საშუალო არის } \rightarrow (16+17+13+19+16)/5 = 16.2$$

შესრულების დროის საშუალო $\rightarrow (21+23+16+25+20)/5 = 21$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 16.2 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 21. დროითი ერთეული.

თუ განხილულ ამოცანას დავამატებთ გამოჩენის დროს ის გარკვეულწილად გართულდება. ვთქვათ, გვაქვს პროცესების შემდეგი ცხრილი

პროცესი	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	6	4
გამოჩენის დრო	1	4	0	7	2

რადგანაც ამ შემთხვევაში ჩვენი ცხრილი დამატებით შეიცავს პროცესების გამოჩენის დროს, ამიტომ პროცესების გამოჩენის და შესრულებიდან პროცესის გამოსვლა უნდა გაკონტროლდეს საგულდაგულოდ.

შევნიშნოთ, რომ თუ სისტემაში პროცესის წარმოქმნასთან ერთად შესრულებად პროცესს ამოეწურა დროითი კვანტი, მაშინ ვინაიდან შესრულებადი პროცესი საჭიროებს კონტექსტის შენახვას ჩავთვლით, რომ მზადყოფნის ცხრილში პირველი ჩაიწერება წარმოქმნილი პროცესი, ხოლო შემდეგ პროცესი, რომელმაც დროითი კვანტის ამოეწურვის გამო იძულებით დატოვა პროცესორი. პროცესების მიმდევრობა, რომლითაც მოხდება დამგეგმავის მიერ მათი შესრულება, შეირჩევა შემდეგნაირად: რადგანაც სისტემაში პირველი გამოჩნდა P₃ პროცესი, ამიტომ ის დაიწყებს შესრულებას. 1 ერთეულის სისტემაში გამოჩნდება P₁ პროცესი. ის მოექცევა მზადყოფნის ცხრილის ბოლოში (მოცემული მომენტისთვის მიმდევრობას ექნება სახე - P₃ P₁). 2 ერთეულის გასვლის შემდეგ მიმდინარე პროცესს იძულებით ჩამოერთმევა პროცესორი და სისტემაში გამოჩნდება ახალი პროცესი P₅. რადგანაც P₃ პროცესის საჭიროებს კონტექსტის შენახვას, ამიტომ ჯერ მიმდევრობის ბოლოში მოექცევა P₅ პროცესი, ხოლო შემდეგ რადგანაც P₃ პროცესი უნდა შესრულდეს კიდევ 1 დროითი ერთეულით, ის გადმოინაცვლებს მიმდევრობის ბოლოში. მოცემული მომენტისთვის მიმდევრობას ექნება სახე - P₁ P₅ P₃.

პროცესების შესრულების დეტალიზირებულ ცხრილს ექნება სახე

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
P ₁	-	+	+	-	-	-	-	-	+	+	-	-	-	-	-	-	+								
P ₂				-	-	-	+	+	-	-	-	-	-	-	+	+	-	-	-	+	+	+	-	-	
P ₃	+	+	-	-	-	-	+																		
P ₄								-	-	-	-	-	-	+	+	-	-	-	+	+	-	-	+	+	
P ₅			-	-	+	+	-	-	-	-	-	-	-	+	+										

ამრიგად, ლოდინის და შესრულების დროისთვის შესაბამისად გვექნება:

ლოდინის საშუალო არის $\rightarrow (12+14+4+11+7)/5 = 9.8$

შესრულების დროის საშუალო $\rightarrow (17+21+7+17+11)/5 = 14.6$

ანუ თითოეულ პროცესს პროცესორის მისაღებად საშუალოდ მოუწია 9.8 დროითი ერთეულით ლოდინი, ხოლო მის შესრულებას დასჭირდა 14.6 დროითი ერთეული.

დამოუკიდებელი სამუშაო

1. დაგეგმვის ციკლური ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების დროის საშუალო, თუ ყველა პროცესი სისტემაში არსებობს ერთდროულად:

პროცესი (კვანტით 3)	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	15	10	5	2	15

პროცესი (კვანტით 1)	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1

პროცესი (კვანტით 2)	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	10	1	5	10	11

2. დაგეგმვის ციკლური ალგორითმის გამოყენებით დაითვალეთ პროცესების შესრულების დროის საშუალო, თუ პროცესები სისტემაში გამოჩნდნენ სხვადასხვა დროს:

პროცესი (კვანტით 2)	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	5	7	3	2	8
გამოჩენის დრო	0	2	2	5	1

პროცესი (კვანტით 2)	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	8	10	7	2	15
გამოჩენის დრო	2	3	0	5	0

პროცესი (კვანტით 2)	P ₁	P ₂	P ₃	P ₄	P ₅
შესრულების დრო	16	7	8	2	1
გამოჩენის დრო	2	0	3	7	5

სემინარი 4. UNIX პროცესების ურთიერთქმედების ორგანიზაცია

პროცესები, რომლებიც იქმნებიან სხვადასხვა პროგრამების მიერ ემსახურებიან გარკვეულ მიზანს. წარმოქნილი (შვილი) პროცესი ასრულებს წარმომქმნელი (მშობელი) პროცესის მიერ მისთვის დაკისრებულ ამოცანას (შეტანა/გამოტანის ოპერაციები ან გარკვეული გამოთვლები). შვილი პროცესის დასრულების შემდეგ მშობელმა პროცესმა უნდა იცოდეს ინფორმაცია შვილი პროცესის მიერ შესრულებულ სამუშაოზე. გარდა ამისა შვილი პროცესი მისთვის დაკისრებული ამოცანის შესრულების მომენტში შეიძლება მშობლისგან საჭიროებდეს გარკვეული მონაცემების მიღებას ან მშობლისათვის გარკვეული მონაცემების გადაცემას. ასეთი კომუნიკაციის საჭიროების აუცილებლობისას ოპერაციულ სისტემას აუცილებლად უნდა გააჩნდეს კომუნიკაციის განხორციელების შესაძლებლობა. კომუნიკაციის საშუალებებს შორის ყველაზე მეტად გამოიყენება კავშირის არხი, რომელიც უზრუნველყოფს პროცესების საკმაოდ უსაფრთხო და ინფორმაციულ ურთიერთქმედებას.

კავშირის არხით მონაცემების გადაცემის არსებობს ორი მეთოდი: შეტანა/გამოტანის ნაკადი და შეტყობინება. გამოყენების თვალსაზრისით მათ შორის ყველაზე მარტივია ნაკადების მოდელი, რომელშიც ინფორმაციის მიღება/გადაცემის ოპერაციას საერთოდ არ აინტერესებს რა სახის ინფორმაციის მატარებელია ის. კავშირის არხში მთლიანი ინფორმაცია განიხილება, როგორც ბიტების ნაკრები, რომელსაც არ გააჩნია შიდა სტრუქტურა.

4.1. ფაილთან მუშაობა

ინფორმაციის გადაცემა ნაკადების მეშვეობით შესაძლებელია განხორციელდეს არამხოლოდ პროცესებს შორის, არამედ პროცესსა და შეტანა/გამოტანის მოწყობილობას შორის (მაგალითად, პროცესსა და დისკს შორის), რომელზეც მონაცემები წარმოიდგინება ფაილების სახით. ვინაიდან ფაილებთან ნაკადებით მუშაობისას გამოყენებადი სისტემური გამოძახებები წააგავს პროცესების ნაკადებით ურთიერთქმედებისას გამოყენებად სისტემურ გამოძახებებს, ამიტომ თავდაპირველად განვიხილოთ ფაილებსა და პროცესებს შორის ნაკადებით ინფორმაციის გაცვლის მექანიზმი.

პროგრამირების ყველა ენას გააჩნია ფაილებთან მუშაობის სტანდარტული ბიბლიოთეკური ფუნქციები, რომელთა მეშვეობითაც პროგრამისტს შეუძლია განახორციელოს გარკვეული მოქმედებები ფაილებზე. მაგალითად, C ენისთვის ფაილთან სამუშაო ფუნქციებია fopen(), fread(), fwrite(), fprintf(), fscanf(), fgets() და ა.შ. თითოეული მათგანი გამოიყენება ფაილზე გარკვეული მოქმედების განსახორციელებლად. მაგალითად, fgets() ფუნქცია გამოიყენება ფაილიდან სიმბოლოების წასაკითხად, fscanf() ფუნქცია კი ანხორციელებს ინფორმაციის შეტანას, რომელიც შეესაბამება მითითებულ ფორმატს, და ა.შ. ნაკადების მოდელის მიხედვით ოპერაციები, რომელიც განსაზღვრულია შეტანა/გამოტანის სტანდარტულ ბიბლიოთეკაში, არ წარმოადგენენ ნაკადურ ოპერაციებს, ვინაიდან თითოეული მათგანი მოითხოვს გადასაცემი მონაცემების სტრუქტურას.

UNIX ოპერაციული სისტემაში ეს ფუნქციები წარმოადგენენ სისტემურ გამოძახებებზე დანამატს - სერვისულ ინტერფეისს, რომლებიც ანხორციელებენ ნაკადებით ინფორმაციის გაცვლის პირდაპირ ოპერაციებს პროცესებსა და ფაილებს შორის და არ საჭიროებენ იმის ცოდნას თუ რას შეიცავს ის.

4.2. ფაილური დესკრიპტორი

პროცესებზე საუბრისას აღვნიშნეთ, რომ ყოველ პროცესს სისტემაში გამოყოფილი აქვს მეხსიერების საკუთარი სივრცე, რომელშიც ინახება პროცესთან დაკავშირებული ყველა საჭირო

ინფორმაცია (პროგრამული ფაილები, შესასრულებლად საჭირო ფაილები, ცვლადები და ა.შ.), და რესურსები. პროცესის მიერ გამოყენებულ ფაილებზე ინფორმაცია შედის მის სისტემურ კონტექსტში და ინახება მის მართვის ბლოკში - PCB. გადმოცემის სიმარტივისთვის შეგვიძლია დავუშვათ, რომ UNIX ოპერაციულ სისტემაში ინფორმაცია იმ ფაილებზე, რომელზედაც პროცესი ნაკადებით ანხორციელებს მონაცემების გაცვლის ოპერაციას, ნაკადების კავშირის არზე ინფორმაციასთან ერთად (რომელიც აკავშირებს პროცესს სხვა პროცესებთან და შეტანა/გამოტანის სხვა მოწყობილობებთან) ინახება გარკვეულ მასივში, რომელიც შეესაბამება შეტანა/გამოტანის გარკვეულ ნაკადს, ეწოდება ფაილური დესკრიპტორი ამ ნაკადისთვის. ამრიგად, ფაილური დესკრიპტორი წარმოადგენს პატარა არაუარყოფით რიცხვს, რომელიც მიმდინარე პროცესისთვის (დროის მოცემულ მომენტში) განსაზღვრავს შეტანა/გამოტანის გარკვეულ მოქმედ არხს. ზოგიერთი ფაილური დესკრიპტორი ნებისმიერი პროგრამის დაწყების ეტაპზე ასოცირდება შეტანა/გამოტანის სტანდარტულ ნაკადთან. მაგალითად, ფაილური დესკრიპტორი 0 შეესაბამება მონაცემთა შეტანის სტანდარტულ ნაკადს, 1 - მონაცემთა გამოტანის სტანდარტულ ნაკადს, 2 კი - შეცდომის გამოტანის სტანდარტულ ნაკადს. ინტერაქტიულ რეჟიმში ნორმალური მუშაობისას მონაცემთა შეტანის სტანდარტული ნაკადი პროცესს აკავშირებს კლავიატურასთან, ხოლო მონაცემების და შეცდომების გამოტანის სტანდარტული ნაკადი კი - ტერმინალის ეკრანთან.

4.3. ფაილებთან მუშაობა

`open()` სისტემური გამოძახება. ფაილური დესკრიპტორი გამოიყენება სისტემური გამოძახებისთვის შეტანა/გამოტანის ნაკადის აღმწერი პარამეტრის როლში, რომელიც ამ ნაკადზე ანხორციელებს ოპერაციებს. ამიტომ სანამ განვახორციელებდეთ ფაილიდან კითხვას და მასში ჩაწერას ფაილზე ინფორმაცია უნდა განვათავსოთ გახსნილი ფაილების ცხრილში და განვსაზღვროთ შესაბამისი დესკრიპტორი. ამისთვის გამოიყენება ფაილის გახსნის პროცედურა, რომელსაც ანხორციელებს სისტემური გამოძახება `open()`.

`open()` სისტემური გამოძახება flag-ების ნაკრებს იყენებს იმისთვის, რომ მოახდინოს იმ ოპერაციების განსაზღვრა, რომელთა გამოყენება ფაილის მიმართ იგულისხმება მომავალში, ან რომლებიც უნდა იყვნენ შესრულებულნი უშუალოდ ფაილის გახსნის მომენტში. flag-ების ყველა შესაძლო ნაკრებიდან განვიხილავთ მხოლოდ რამდენიმეს: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT` და `O_EXCL`. პირველი სამი flag-ი არის ურთიერთგამომრიცხავი: ერთერთი მათგანის გამოყენება აუცილებელია და მისი არსებობა გამორიცხავს დანარჩენი ორის არსებობის შესაძლებლობას. ეს flag-ები აღწერენ ოპერაციათა ნაკრებს, რომლებიც მომავალში ფაილის წარმატებული გახსნის შემთხვევაში ფაილებზე იქნება დაშვები: მხოლოდ კითხვა, მხოლოდ რედაქტირება, კითხვა და რედაქტირება. როგორც ვიცით ყოველ ფაილს ყავს სამი ტიპის მომხმარებელი, რომლებიც ამ ფაილზე სარგებლობს გარკვეული უფლებებით. თუ ფაილი მოცემული სახელით არსებობს დისკზე და მასზე დაშვების უფლებები იმ მომხმარებლისთვის, რომლის სახელითაც მუშაობს მიმდინარე პროცესი, არ ეწინააღმდეგება მოთხოვნილ ოპერაციათა ნაკრებს, მაშინ ოპერაციული სისტემა ახდენს ფაილური ცხრილის სკანირებას დასაწყისიდან ბოლომდე პირველი ცარიელი ელემენტის მოსაძებნად, ავსებს მას და აბრუნებს ამ ელემენტის ინდექსს ფაილის დესკრიპტორის როლში. თუ ფაილი დისკზე არაა, არ ყოფნის უფლებები ან გახსნილი ფაილების ცხრილში არ არსებობს ცარიელი ელემენტი, მაშინ წარმოიქმნება შეცდომის შემცველი სიტუაცია.

იმ შემთხვევაში, როდესაც ვუშვებთ, რომ ფაილი დისკზე შეიძლება არ არსებობს და საჭიროა მისი შექმნა, flag-ი ოპერაციათა ნაკრებისთვის უნდა იყოს გამოყენებული O_CREAT flag-თან ერთად. თუ ფაილი არსებობს, მაშინ ყველაფერი ხორციელდება ზემოთ აღწერილი სცენარით. თუ ფაილი არ არსებობს, მაშინ თავიდან ხორციელდება ფაილის შექმნა, სისტემური გამომახების პარამეტრებში მითითებული უფლებებით. ოპერაციათა ნაკრების შესაბამისობის შემოწმება, გამოცხადებულ დაშვების უფლებებით, შესაძლებელია არც განხორციელდეს (როგორც მაგალითად, Linux-ში).

იმ შემთხვევაში, როდესაც ჩვენ ვითხოვთ, რომ ფაილი დისკზე არ არსებობდეს და იყოს შექმნილი მისი გახსნის მომენტში, flag-ი ოპერაციათა ნაკრებისთვის O_CREAT და O_EXCL flag-ებთან უნდა გამოიყენებოდეს კომბინირებულად.

open სისტემური გამომახების პროტოტიპი

```
#include <fcntl.h>
int open(char *path, int flags);
int open(char *path, int flags, int mode);
```

სისტემური გამომახების აღწერა

open სისტემური გამომახება გამოიყენება ფაილის გახსნის ოპერაციისთვის და მისი განხორციელებისას აბრუნებს გახსნილი ფაილის ფაილურ დესკრიპტორს (პატარა არაუარყოფით მთელ რიცხვს), რომელიც სხვდასხვა სისტემური გამომახების მიერ გამოიყენება ამ ფაილზე დასაშვები ოპერაციების განსახორციელებლად.

path პარამეტრი წარმოადგენს მითითებელს იმ სტრიქონზე, რომელიც შეიცავს გასახსნელი ფაილის მიმართებით ან სრულ სახელს.

flags პარამეტრი განსაზღვრავს გასახსნელ ფაილზე დაშვების უფლებებს. მას შეუძლია მიიღოს შემდეგი სამი მნიშვნელობიდან მხოლოდ ერთი:

- O_RDONLY - მხოლოდ კითხვა;
- O_WRONLY - მხოლოდ რედაქტირება;
- O_RDWR - კითხვა და რედაქტირება;

ამ მნიშვნელობების კომბინირება „ბიტური ან ('|)“ ოპერაციით შესაძლებელია ერთი ან რამდენიმე flag-თან:

- O_CREAT - თუ ფაილი მითითებული სახელით არ არსებობს და საჭიროა მისი შექმნა;
- O_EXCL - გამოიყენება O_CREAT flag-თან ერთად. მათი ერთობლივი გამოყენებისას და მითითებული სახელის ფაილის არსებობისას არ ხდება ფაილის გახსნა და წარმოიშობა შეცდომის შემცველი სიტუაცია.
- O_NDELAY - კრძალავს პროცესის გადაყვანას ლოდინის მდგომარეობაში ფაილის გახსნის ოპერაციის ან ამ ფაილზე შემდგომი ნებისმიერი ოპერაციისას.
- O_APPEND - ფაილის გახსნისას და ჩაწერის ყოველი ოპერაციის შესრულების წინა (თუ ის დაშვებულია) ფაილის მიმდინარე პოზიციის მიმთითებელი ყენდება ფაილის ბოლოზე.
- O_TRUNC - თუ ფაილი არსებობს მის მოცულობას ამცირებს 0-მდე, ფაილის არსებული ატრიბუტების შენახვით გარდა შესაძლებელია ფაილზე ბოლო დაშვებისა და მისი ბოლო მოდიფიცირებისა.

გარდა ამისა, UNIX ოპერაციული სისტემის ზოგიერთ ვერსიებში შესაძლებელია გამოიყენებოდეს flag-ების დამატებითი მნიშვნელობები:

- O_SYNC - ფაილში ჩაწერის ნებისმიერი ოპერაცია იქნება ბლოკირებული იმ დრომდე სანამ ჩაწერილი ინფორმაცია ფიზიკურად არ იქნება განთავსებული შესაბამის hardware დონეზე.
- O_NOCTTY - თუ ფაილის სახელი განეკუთვნება ტერმინალურ მოწყობილობას, ის არ იქცევა პროცესის მმართველ ტერმინალად, იმ შემთხვევაშიც კი, თუ ამ დრომდე მას

არ გააჩნდა მმართველი ტერმინალი.

mode პარამეტრი ახალი ფაილის შექმნისას მისთვის აყენებს სხვადასხვა კატეგორიის მომხმარებელთათვის დაშვების უფლებებს. ის აუცილებელია, თუ მოცემულ flag-ებს შორის არის flag-ი O_CREAT და წინააღმდეგ შემთხვევაში შეიძლება იქნას გამოტოვებული. ეს პარამეტრი გამოისახება, როგორც შემდეგი რვაობითი მნიშვნელობების ჯამი.

- 0400 - მომხმარებელი სარგებლობს კითხვის უფლებით;
- 0200 - მომხმარებელი სარგებლობს რედაქტირების უფლებით;
- 0100 - მომხმარებელი სარგებლობს შესრულების უფლებით;
- 0040 - მომხმარებლის ჯგუფი სარგებლობს კითხვის უფლებით;
- 0020 - მომხმარებლის ჯგუფი სარგებლობს რედაქტირების უფლებით;
- 0010 - მომხმარებლის ჯგუფი სარგებლობს შესრულების უფლებით;
- 0004 - სხვა მომხმარებელი სარგებლობს კითხვის უფლებით;
- 0002 - სხვა მომხმარებელი სარგებლობს რედაქტირების უფლებით;
- 0001 - სხვა მომხმარებელი სარგებლობს შესრულების უფლებით.

ფაილის შექმნისას რეალურად დასაყენებელი დაშვების უფლებები მიიღება mode პარამეტრის სტანდარტული კომბინაციისაგან და მიმდინარე პროცესის ფაილის შექმნის ნიდაბის umask მნიშვნელობისაგან, კერძოდ, ისინი ტოლია mode = umask. FIFO ტიპის ფაილების გახსნისას სისტემურ გამოძახებას გააჩნია ყოფაქცევის გარკვეული თავისებურება სხვა ტიპის ფაილების გახსნასთან შედარებით. თუ FIFO-ს გახსნა ხდება მხოლოდ ჩასაკითხად და არაა დასმული flag-ი O_NDELAY, მაშინ ხდება სისტემური გამოძახების განმახორციელებელი პროცესის ბლოკირება სანამ რომელიმე სხვა პროცესი არ გახსნის FIFO-ს ჩასაწერად. თუ O_NDELAY flag-ი დასმულია, მაშინ ბრუნდება FIFO -სთან ასოცირებული ფაილური დესკრიპტორის მნიშვნელობა. თუ FIFO გაიხსნა მხოლოდ ჩასაწერად და არაა დასმული flag-ი O_NDELAY, მაშინ ხდება სისტემური გამოძახების განმახორციელებელი პროცესის ბლოკირება სანამ რომელიმე სხვა პროცესი არ გახსნის FIFO-ს ჩასაკითხად. თუ O_NDELAY flag-ი დასმულია, მაშინ წარმოიშობა შეცდომის შემცველი სიტუაცია და ბრუნდება მნიშვნელობა -1.

დასაბრუნებელი მნიშვნელობა

სისტემური გამოძახება ნორმალურად დასრულების შემთხვევაში აბრუნებს გახსნილი ფაილისთვის ფაილური დესკრიპტორის მნიშვნელობას, ხოლო შეცდომის წარმოქმნის შემთვევაში მნიშვნელობას -1.

სისტემური გამოძახება read() და write(). როგორც ზემოთ აღვნიშნეთ open() სისტემური გამოძახება ნორმალური დასრულების შემთხვევაში აბრუნებს ფაილურ დესკრიპტორს, რომლის გამოყენებაც არის შესაძლებელი ფაილზე სხვადასხვა (კითხვის, რედაქტირების და ა.შ.) ოპერაციების განსახორციელებლად. ფაილიდან მონაცემების წასაკითხად გამოიყენება სისტემური გამოძახება read(), ხოლო ფაილში მონაცემების ჩასაწერად კი - write().

read და write სისტემურ გამოძახებათა პროტოტიპი

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

სისტემურ გამოძახებათა აღწერა

სისტემური გამოძახება write და read გამოიყენება ინფორმაციის შეტანა/გამოტანის ოპერაციების განსახორციელებლად ფაილებში ან მეხსიერებაში

fd პარამეტრი წარმოადგენს ფაილურ დესკრიპტორს ადრე შექმნილი კავშირის არხისთვის, რომლის მეშვეობითაც განხორციელდება ინფორმაციის გადაცემა ან მიღება.

addr პარამეტრი წარმოადგენს მეხსიერების სივრცის მისამართს, რომლიდანაც დაწყებული იქნება

ინფორმაცია წაკითხული ან ინფორმაცია განთავსებული.

nbytes პარამეტრი განსაზღვრავს write სისტემური გამოძახებისთვის ბაიტების იმ რაოდენობას, რომელთა განთავსებაც სისტემურმა გამოძახებამ განახორციელა მეხსიერების სივრცეში addr მისამართიდან დაწყებული, ხოლო read სისტემური გამოძახებისთვის კი განსაზღვრავს ბაიტების იმ რაოდენობას, რომელთა კითხვა განახორციელა სისტემურმა გამოძახებამ მეხსიერების სივრციდან addr მისამართიდან დაწყებული.

დასაბრუნებელი მნიშვნელობა

წარმატებული დასრულების შემთხვევაში სისტემური გამოძახება აბრუნებს რეალურად გაგზავნილი ან მიღებული ბაიტების რაოდენობას. შევნიშნოთ, რომ ეს მნიშვნელობა შეიძლება არ ემთხვევოდეს nbytes პარამეტრის მოცემულ მნიშვნელობას და იყოს მასზე ნაკლები, მონაცემთა გადაცემისას დისკზე ან კავშირის არხში საკმარისი ადგილის არარსებობის ან მისი მიღებისას ინფორმაციის არარსებობის გამო. წარუმატებელი დასრულების ან გადაცემისას შეცდომის წარმოქმნის შემთხვევაში სისტემური გამოძახება აბრუნებს უარყოფით მნიშვნელობას.

ფაილებთან მუშაობისას ყოფაქცევის თავისებურებანი. ფაილებთან მუშაობისას ხორციელდება ფაილიდან ინფორმაციის კითხვა ან იქ მისი ჩაწერა, ფაილის მიმთითებლის მიმდინარე მდგომარეობიდან დაწყებული. მიმთითებლის მნიშვნელობა იზრდება რეალურად ჩაწერილი ან წაკითხული ბაიტების რაოდენობით. ფაილიდან ინფორმაციის წაკითხვისას ის არ იკარგება ფაილიდან. თუ სისტემური გამოძახება read აბრუნებს მნიშვნელობას 0, ეს ნიშნავს, რომ ფაილიდან ინფორმაცია წაკითხულია ბოლომდე.

ნაკადების ოპერაციის დასრულების შემდეგ პროცესმა უნდა შეასრულოს შეტანა/გამოტანის ნაკადის დახურვის ოპერაცია, რომლის დროსაც მოხდება კავშირის არხის ბუფერის სრულად გასუფთავება, გათავისუფლდება ოპერაციული სისტემის ის რესურსები, რომლებსაც იკავებდა პროცესი, ფაილური დესკრიპტორების ცხრილის შესაბამისი ელემენტი. ამ მოქმედებებზე პასუხისმგებელია სისტემური გამოძახება close(). უნდა აღინიშნოს, რომ პროცესის მუშაობის დასრულებისას exit() ფუნქციის ცხადი თუ არაცხადი გამოძახებისას შეტანა/გამოტანის გახსნილი ნაკადები დახურვა ხორციელდება ავტომატურად.

close სისტემური გამოძახების პროტოტიპი

```
#include <unistd.h>
int close(int fd);
```

სისტემური გამოძახების აღწერა

close სისტემური გამოძახება გამოიყენება ფაილებთან მუშაობის ოპერაციების კორექტული დასრულებისთვის, რომლებიც აღიწერებიან ოპერაციულ სისტემაში ფაილური დესკრიპტორების მეშვეობით.

დასაბრუნებელი მნიშვნელობა

სისტემური გამოძახება ნორმალური დასრულების შემთხვევაში აბრუნებს მნიშვნელობას - 0 და შეცდომის წარმოქმნის შემთხვევაში მნიშვნელობას -1.

განვიხილოთ მაგალითი. შევქმნათ პროცესი, რომელშიც მასივში არსებული მონაცემების ჩაწერა მოხდება პროგრამით შექმნილ ფაილში.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(){
    int fd; // ცვლადი ფაილური დესკრიპტორის მნიშვნელობისთვის
```

```

size_t s1, s2; // ცვლადები მონაცემების ზუსტად გადაცემის შესამოწმებლად
char mass[] = "Operating System";
s2 = sizeof(mass);
// გავანულოთ შესაქმნელი ფაილისთვის ოპერაციული სისტემის მიერ გაჩუმებით
// დანიშვნადი დაშვების უფლებები
(void) umask(0);
// სამუშაო დირექტორიაში შევმნქათ test.txt ფაილი (უფლებებით 0765)
fd = open("test.txt", O_WRONLY | O_CREAT, 0765);
// შევამოწმოთ fd ფაილური დესკრიპტორის მნიშვნელობა
if (fd < 0){ // თუ ფაილი არ შეიქმნა, მასში მონაცემებს ვერ ჩავწერთ
    printf("Can't open file\n");
    exit(EXIT_FAILURE);
}
// ინფორმაციის ჩაწერა კავშირის არხში
s1 = write(fd, mass, s1);
// თუ მონაცემების ჩაწერისას სისტემური გამოძახება წარუმატებლად დასრულდა ან
// მონაცემები სრულად არ იქნა ჩაწერილი, მაშინ ვასრულებთ პროგრამას
if (s1 != s2) {
    printf("Can't write all massing\n");
    exit(EXIT_FAILURE);
}
close(fd); //დავხუროთ კავშირის არხი
return 0; }
პროგრამა 04-1.c. open(), write() და close() სისტემური გამოძახების გამოყენება
აკრიფეთ, დააკომპილირეთ და შეასრულეთ პროგრამა.

```

4.4. კავშირის არხი

მიუხედავად იმისა, პროცესები შეიძლება სრულდებოდნენ ერთ ან რამდენიმე გამოთვლით მანქანაზე, მათ ხშირად უწევთ ერთიმეორესთან ურთიერთქმედება, საერთო რესურსების გამოყენება, ერთიმეორის მიერ მიღებული შედეგების გათვალისწინება საკუთარი სამუშაოს შესრულებისას და ა.შ.

პროცესებს ერთიმეორესთან ურთიერთქმედებენ მონაცემების გაცვლის გზით. მონაცემების გაცვლის არსებობს სამი მექანიზმი:

- სიგნალები;
- კავშირის არხები
- განაწილებადი მეხსიერება.

მიმდინარე სემინარულ მეცადინეობაზე განვიხილავთ პროცესებს შორის მონაცემების გაცვლის კავშირის საშუალებებს.

კავშირის არხებით პროცესებს შორის მონაცემების გაცვლა ხორციელდება ოპერაციული სისტემის მიერ ამ მიზნისთვის სპეციალურად გამოყოფილი კავშირის არხებით. გადასაცემი მონაცემების მოცულობა დამოკიდებულია კავშირის არხის გამტარუნარიანობაზე. გადასაცემი მონაცემების მოცულობის გაზრდით იზრდება სხვა პროცესების ყოფაქცევაზე ზემოქმედების შესაძლებლობა;

4.4.1. კავშირის არხი pipe

UNIX ოპერაციულ სისტემაში მონაცემთა ნაკადების მოდელის გამოყენებით სხვადასხვა პროცესებს შორის ან ერთი პროცესის შიგნით ინფორმაციის გადაცემის ყველაზე მარტივ მეთოდს

წარმოადგენს pipe (არხი, მილი).

pipe შევიძია წარმოვიდგინოთ როგორც ოპერაციული სისტემის მეხსიერების მისამართების სივრცეში განთავსებული სასრული მოცულობის „უზილავი მილი“, რომლის შემავალ და გამომავალ ბოლოებზე მიმართვა ხორციელდება სისტემური გამოძახებების მეშვეობით. pipe ხშირად ორგანიზებულია წრიული ბუფერის სახით. ბუფერში კითხვის და ჩაწერის ოპერაციების შესრულებისას გადაადგილდება შემავალი და გამომავალი ნაკადების შესაბამისი ორი მიმთითებელი. ამასთან, გამომავალ ნაკადს არ შეუძლია გაასწროს შემავალ ნაკადს და პირიქით. ოპერაციული სისტემის შიგნით ასეთი წრიული ბუფერის ახალი ეგზემპლარის შესაქმნელად გამოიყენება სისტემური გამოძახება pipe().

pipe სისტემური გამოძახების პროტოტიპი

```
#include<unistd.h>
```

```
int pipe(int *fd);
```

სისტემური გამოძახების აღწერა

სისტემური გამოძახება pipe ემსახურება ოპერაციული სისტემის შიგნით pipe-ის შექმნას.

fd პარამეტრი წარმოადგენს მიმთითებელს ორი მთელი პარამეტრისაგან შემდგარ მასივზე.

სისტემური გამოძახების წარმატებით დასრულების შემდეგ მასივის პირველ ელემენტში - fd[0] - შეტანილი იქნება ფაილური დესკრიპტორი, რომელიც შეესაბამება pipe-ის გამოსატანი მონაცემების ნაკადს და იძლევა მხოლოდ კითხვის ოპერაციის განხორციელების შესაძლებლობას, ხოლო მასივის მეორე ელემენტში - fd[1] - შეტანილი იქნება ფაილური დესკრიპტორი, რომელიც შეესაბამება მონაცემების შესატან ნაკადს და იძლევა მხოლოდ ჩაწერის ოპერაციის განხორციელების შესაძლებლობას.

დასაბრუნებელი მნიშვნელობა

სისტემური გამოძახება აბრუნებს 0-ის ტოლ მნიშვნელობას წარმატებული დასრულების შემთხვევაში და შეცდომის წარმოქმნის შემთხვევაში კი -1-ს.

მუშაობის პროცესში სისტემური გამოძახება ახდენს ბუფერში მეხსიერების მიდამოს და მიმთითებლის გამოყოფის ორგანიზებას და შემავალ და გამომავალ ნაკადებზე შესაბამისი ინფორმაცია შეაქვს ფაილური დესკრიპტორების ცხრილის ორ ელემენტში, რითაც ყოველ pipe-თან აკავშირებს ორივე ფაილურ დესკრიპტორს. ერთერთი მათგანისთვის დაშვებულია მხოლოდ pipe-დან კითხვის ოპერაცია, ხოლო მეორისთვის მხოლოდ pipe-ში ჩაწერის ოპერაცია. ამ ოპერაციების განსახორციელებლად შესაძლებელია გამოყენებულ იქნას იგივე read() და write() სისტემური გამოძახებები, რაც ფაილებთან მუშაობის შემთხვევაში. ბუნებრივია, რომ მონაცემთა ნაკადის შეტანის ან/და გამოტანის ოპერაციების დასრულების შემდეგ საჭიროა შესაბამისი ნაკადის დახურვა და სისტემური რესურსების გამოთავისუფლება close() სისტემური გამოძახების გამოყენებით. უნდა აღინიშნოს, რომ თუ pipe-ის გამოყენებელი ყველა პროცესი ხურავს მათთან ასოცირებულ ფაილურ დესკრიპტორს, მაშინ ოპერაციული სისტემა ახდენს pipe-ის ღიკვიდირებას. ამრიგად, pipe -ის სიცოცხლის ხანგრძლივობა სისტემაში არ შეიძლება აღემატებოდეს მასთან მომუშავე პროცესების სიცოცხლის ხანგრძლივობას.

მნიშვნელოვანი განსხვავება pipe-სა და ფაილს შორის მდგომარეობს იმაში, რომ ინფორმაცია pipe-დან იშლება წაკითხვისთანავე და ხელმეორედ მისი წაკითხვა შეუძლებელია.

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<unistd.h>
```

```
#include<sys/types.h>
```

```
int main(){
```

```
    int fd[2]; // ორელემენტიანი მასივი ფაილური დესკრიპტორისთვის
```

```

size_t s1, s2; // ცვლადები მონაცემთა ზუსტად გადაცემის შესამოწმებლად
char mass[] = "Operating System!";
s2 = sizeof(mass);
char re_mass[s2];
// შევქმნათ კავშირის არხი pipe-ი
if(pipe(fd) < 0) {
    printf("Can't create pipe\n");
    exit(EXIT_FAILURE);
}
// pipe არხში შევიტანოთ mass მასივის მონაცემები
s1 = write(fd[1], mass, s2);
// შევამოწმოთ რამდენად სწორად მოხდა მონაცემების ჩაწერა
if(s1 != s2){
    printf("Can't write all massing\n");
    exit(EXIT_FAILURE);
}
// pipe არხიდან re_mass მასივში წავიკითხოთ მონაცემები
s1 = read(fd[0], re_mass, s2);
// შევამოწმოთ რამდენად სწორად მოხდა მონაცემების ჩაწერა
if(s1 != s2) {
    printf("Can't read massing\n");
    exit(EXIT_FAILURE);
}
printf("%s\n", re_mass); // დავტეჭდოთ re_mass მასივი
close(fd[0]); // დავხუროთ გახსნილი კავშირის არხები
close(fd[1]);
return 0;
}

```

პროგრამა 04-2.c. pipe-ის წარმოქმნა სისტემაში

აკრიფეთ პროგრამა, დააკომპილირეთ და შეასრულეთ ის.

4.4.2. მემკვიდრე პროცესებს შორის ერთმიმართულებიანი კავშირის ორგანიზება

ცხადია, რომ თუ pipe-ის უპირატესობა შემოფარგლული იქნებოდა ერთი პროცესის შიგნით კოპირების ფუნქციით, რომლის განხორციელებისას მონაცემებს მოუწევდათ ოპერაციულ სისტემის „შემოვლა“, მაშინ ის არ იქნებოდა აღნიშვნის ღირსი. fork() სისტემური გამოძახებისას მემკვიდრე პროცესებს გადაცემათ ფაილური დესკრიპტორის მნიშვნელობა, რომელიც exec() სისტემური გამოძახების განხორციელებისას შედის პროცესის სისტემური კონტექსტის განუყოფელ ნაწილში. ეს გარემოება საშუალებას გვაძლევს pipe-ის მეშვეობით განვახორციელოთ ინფორმაციის გადაცემა იმ მემკვიდრე პროცესებს შორის, რომელთაც ყავთ pipe-ს წარმომქმნელი წინაპარი.

განვიხილოთ მაგალითი, რომელშიც ”მემკვიდრე“ პროცესებს შორის pipe არხის მეშვეობით მონაცემების გაცვლის მიზნით დემონსტრირებულია ერთმიმართულებიანი კავშირის ორგანიზება.

```

#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h> int main(){

```

```

int fd[2], pr, st;
size_t s1, s2;
char mass[] = "Operating System!";
s2 = sizeof(mass); char re_mass[s2];
// შევქმნათ კავშირის არხი pipe-ი
if(pipe(fd) < 0){
    printf("Can't create pipe\n");
    exit(EXIT_FAILURE);
}
// შევქმნათ ახალი პროცესი
if((pr = fork()) < 0){
    printf("Can't fork child\n");
    exit(EXIT_FAILURE);
}
else {
    if(pr != 0) { // მშობელ-პროცესი
        close(fd[0]); // დავხუროთ მონაცემების კითხვის არხი
        s1 = write(fd[1], mass, s2); // pipe არხში ჩავწეროთ მონაცემები
        if(s1 != s2){
            printf("Can't write all massing\n");
            exit(EXIT_FAILURE);
        }
        close(fd[1]); // დავხუროთ მონაცემების ჩაწერის არხი
        printf("Parent exit\n");
    }
    else { // შვილი-პროცესი
        close(fd[1]); // დავხუროთ მონაცემების ჩაწერის არხი
        s1 = read(fd[0], re_mass, s2); // კავშირის არხიდან წავიკითხოთ მონაცემები
        if(s1 != s2){
            printf("Can't read massing\n");
            exit(EXIT_FAILURE);
        }
        printf("%s\n", re_mass); // დავგეჭდოთ წაკითხული სტრიქონი
        close(fd[0]); // დავხუროთ მონაცემების კითხვის არხი
    }
}
return 0;
}

```

პროგრამა 04-3.c. ერთმიმართულებიანი კავშირი მშობელ- და შვილ-პროცესს შორის აკრიფტეთ პროგრამა, დააკომპილირეთ და შეასრულეთ ის.

4.4.3. მემკვიდრე პროცესებს შორის ორმიმართულებიანი კავშირის ორგანიზება

pipe-ი გამოიყენება მხოლოდ ერთმიმართულებიანი ანუ სიმპლექსური კავშირისთვის. წინა მაგალითში pipe-ის მეშვეობით ორმიმართულებიანი კავშირის ორგანიზების შემთხვევაში შესაძლებელია წარმოქმნილიყო შემდეგი სიტუაცია: მშობელი პროცესი კავშირის არხიდან წაიკითხავდა მის მიერვე ჩაწერილ მონაცემებს (თითქოს შვილმა განახორციელა მათი ჩაწერა), ხოლო შვილი პროცესი კი საერთოდ ვერ მიიღებდა ვერაფერს. ერთი pipe-ით ორმიმართულებიანი კავშირის ორგანიზებისთვის საჭიროა პროცესების სინქრონიზაციის სპეციალური საშუალებები. უფრო მარტივი მეთოდი მემკვიდრე პროცესებს შორის ორმიმართულებიანი

კავშირის ორგანიზაციისა მდგომარეობს ორივე მიმართულებით თითო pipe-ის გამოყენებაში.

შევნიშნოთ, რომ ზოგიერთ UNIX-მსგავს სისტემაში (მაგალითად, Solaris-ში) რეალიზებულია სრული დუპლექსური pipe. ასეთ სისტემებში ორივე ფაილური დესკრიპტორისთვის, რომელიც ასოცირდება pipe-თან, ნებადართულია კითხვის და ჩაწერის ოპერაცია.

4.4.4. კავშირის არხი FIFO

როგორც უკვე აღვნიშნეთ, ოპერაციულ სისტემაში pipe-ის განთავსებაზე ინფორმაციას ფლობს მხოლოდ მისი წარმომქმნელი პროცესი და ეს ინფორმაცია ინახება ფაილური დესკრიპტორების ცხრილში. ვინაიდან სისტემაში ახალი პროცესის წარმოქმნისას ახალ პროცესს მემკვიდრეობით გადაეცემა ფაილური დესკრიპტორების მნიშვნელობები, ამიტომ pipe-ის გამოყენება შეუძლიათ მხოლოდ pipe-ის წარმომქმნელ და მის მემკვიდრე პროცესებს. UNIX ოპერაციულ სისტემაში არსებობს pipe-ის გამოყენებით არამონათესავე პროცესებს შორის მონაცემების გადაცემის შესაძლებლობა, მაგრამ მისი რეალიზება საკმაოდ რთულია და ჩვენ მას არ შევეხებით.

UNIX-მსგავს ოპერაციულ სისტემაში ნაკადებით ნებისმიერი პროცესების ურთიერთქმედებისთვის გამოიყენება კავშირის არხი, რომელსაც FIFO ეწოდება. FIFO ყველაფრით წარმომადგენერირდება გარდა ერთი გამონაკლისისა: ბირთვის მისამართების სივრცეში FIFO -ს განთავსებაზე და მის მდგომარეობაზე მონაცემების მიღება პროცესებს შეუძლიათ არა მემკვიდრე კავშირებით არამედ ფაილური სისტემიდან. ამ მიზნით FIFO -ს შექმნისას დისკზე იქმნება სპეციალური ტიპის ფაილი (.fifo), რომლის გამოყენებითაც პროცესები ცვლიან მონაცემებს ერთმანეთთან. პროცესების ურთიერთქმედების დასრულების შემდეგ FIFO არხი წყვეტს ფუნქციონირებას, ისევე როგორც pipe -ის შემთხვევაში, სპეციალური ტიპის ფაილი კი სისტემაში რჩება. აღნიშნული ფაილის გამოყენება მომავალში პროცესებს შორის ურთიერთქმედებისთვის შესაძლებელია და პროცესების ურთიერთქმედების საჭიროებისას არ არსებობს აუცილებლობა შეიქმნას ახალი fifo გაფართოების ფაილი. ასეთ შემთხვევაში პროცესების ურთიერთქმედებამდე საჭიროა მათ იცოდნენ (სპეციალურად ამ მიზნისთვის შექმნილ) fifo გაფართოების ფაილამდე სრული ან მიმართებითი სახელი. FIFO -ს შესაქმნელად გამოიყენება სისტემური გამოძახება mknod() (UNIX მსგავსი სისტემების ზოგიერთ ვერსიებში დამატებით არსებობს ფუნქცია mkfifo()).

mknod სისტემური გამოძახების პროტოტიპი

```
#include<sys/stat.h>
#include<unistd.h>
int mknod(char *path, int mode, int dev);
```

სისტემური გამოძახების აღწერა

FIFO არხის შესაქმნელად გამოიყენება mknod სისტემური გამოძახება. ვინაიდან ამ სისტემურ გამოძახებას გამოვიყენებთ მხოლოდ კავშირის არხის შექმნის საილუსტრაციოდ ქვემოთ მოყვანილი აღწერა არა სრული. ჩვენ ასევე არ განვიხილავთ პარამეტრების გადაცემის ყველა შესაძლო ვარიანტს.

path პარამეტრი წარმოადგენს მიმთითებელს იმ სტრიქონზე, რომელიც შეიცავს ფაილის სრულ ან მიმართებით სახელს (სპეციალური ტიპის ფაილი) და FIFO იყენებს პროცესებს შორის მონაცემების გაცვლის მიზნით. FIFO -ს შექმნისას მსგავსი სახელისა და გაფართოების ფაილი არ უნდა არსებობდეს იმ დირექტორიაში, სადაც ეს ფაილი იქმნება. წინააღმდეგ შემთხვევაში არ მოხდება FIFO -ს შექმნა.

Mode პარამეტრი აყენებს FIFO-ზე სხვადასხვა კატეგორიის მომხმარებელთა დაშვების უფლებების

ატრიბუტებს. ეს პარამეტრი მოიცემა S_IFIFO ოფციისა და დაშვების უფლებების კომბინირებით ბიტური „ან“ ოპერაციით. FIFO-სთვის დაშვების უფლებები განისაზღვრება შემდეგი რვაობითი მნიშვნელობების კომბინაციით:

- 0400 – FIFO-ს შემქმნელი მომხმარებლისთვის დაშვებულია კითხვის უფლება;
- 0200 – FIFO-ს შემქმნელი მომხმარებლისთვის დაშვებულია ჩაწერის უფლება;
- 0040 – FIFO-ს შემქმნელი მომხმარებლის ჯგუფისთვის დაშვებულია კითხვის უფლება;
- 0020 – FIFO-ს შემქმნელი მომხმარებლის ჯგუფისთვის დაშვებულია ჩაწერის უფლება;
- 0004 – სხვა მომხმარებლებისთვის დაშვებულია კითხვის უფლება;
- 0002 – სხვა მომხმარებლებისთვის დაშვებულია ჩაწერის უფლება.

Dev პარამეტრი არაა არსებითი ჩვენს შემთხვევაში და ამიტომ ჩვენ მას ყოველთვის მოვცემთ მნიშვნელობით 0.

დასაბრუნებელი მნიშვნელობა

FIFO-ს წარმატებით შექმნისას სისტემური გამოძახება აბრუნებს მნიშვნელობას 0, ხოლო

წარუმატებელ შემთხვევაში კი უარყოფით მნიშვნელობას.

mkfifo ფუნქციის პროტოტიპი

```
#include<sys/stat.h>
#include<unistd.h>
int mkfifo(char *path, int mode);
```

ფუნქციის აღწერა

ზოგიერთ UNIX-მსგავს სისსტემაში FIFO-ს შექმნა დამატებით შესაძლებელია mkfifo ფუნქციის გამოყენებით.

path - პარამეტრი წარმოადგენს გარკვეულ დირექტორიაში შესაქმნელი სპეციალური ტიპის ფაილის სრულ ან მიმართებით სახელზე მიმთითებელს. ამ შემთხვევაშიც აუცილებელია, რომ fifo გაფართოების ფაილის შექმნისას მსგავსი სახელის ფაილი არ არსებობდეს შესაბამის დირექტორიაში.

mode პარამეტრი აყენებს FIFO-ზე დაშვების უფლებებს სხვადასხვა კატეგორიის მომხმარებლებისთვის.

დასაბრუნებელი მნიშვნელობა

FIFO-ს წარმატებით შექმნისას ფუნქცია აბრუნებს მნიშვნელობას 0, ხოლო წარუმატებელ შემთხვევაში კი უარყოფით მნიშვნელობას.

მნიშვნელოვანია გვახსოვდეს FIFO-ს ტიპის ფაილი არ იქმნება დისკზე ინფორმაციის განსათავსებლად, რომელიც იწერება სახელდებულ pipe-ში. ეს ინფორმაცია თავსდება ოპერაციული სისტემის მისამართების სივრცის შიგნით, ხოლო ფაილი წარმოადგენს მხოლოდ სანიშნეს, რომელიც ქმნის მისი განთავსების გადამისამართებებს.

4.4.5. open() გამოძახების თავისებურება FIFO-ს გახსნისას

FIFO-სთან მუშაობისას read() და write() სისტემურ გამოძახებებს გააჩნიათ იგივე თავისებურებები, რაც pipe-ის შემთხვევაში. open() სისტემურ გამოძახებისას გვაქვს გარკვეული ცვლილებები, რაც გამოწვეულია მისი შესრულებადი პროცესების ბლოკირების შესაძლებლობით. თუ FIFO იხსნება მხოლოდ ჩასაკითხად და O_NDELAY flag-ი არაა მითითებული, მაშინ ხდება სისტემური გამოძახების შემსრულებელი პროცესის ბლოკირება მანამ, სანამ სხვა რომელიმე პროცესი არ გახსნის FIFO-ს ჩასაწერად. თუ O_NDELAY flag-ი მითითებულია ბრუნდება FIFO-სთან ასოცირებული ფაილური დესკრიპტორის მნიშვნელობა. თუ FIFO იხსნება მხოლოდ ჩასაწერად და O_NDELAY flag-ი არაა მითითებული, მაშინ ხდება სისტემური გამოძახების შემსრულებელი პროცესის ბლოკირება მანამ, სანამ სხვა რომელიმე პროცესი არ გახსნის FIFO-ს ჩასაკითხად. თუ O_NDELAY flag-ი მითითებულია წარმოიშობა შეცდომა და

ბრუნდება მნიშვნელობა -1. open() სისტემური გამოძახების პარამეტრებში O_NDELAY flag-ის მითითებით FIFO-ს გამხსნელ პროცესს ნაკადიდან მონაცემების კითხვის და მასში ჩაწერის მომდევნო ოპერაციების შესრულებისას ეკრალება ბლოკირება.

```
#include<fcntl.h>
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/stat.h>
#include<sys/wait.h>
#include<sys/types.h>
int main(){
    int fd, pr, st, FIFO;
    size_t s1, s2;
    char mass[] = "Operating System!"; // გადასაცემი მასივი
    s2 = sizeof(mass);
    char re_mass[s2];
    char name[]="file fifo"; // კავშირის ფაილის სახელი
    (void) umask(0); // გავანულოთ დაშვების საწყისი უფლებები
    FIFO = mknod(name, S_IFIFO | 0754, 0) // შექმნათ კავშის არხი FIFO
    if(FIFO < 0){
        printf("შეუძლებელია FIFO-ს შექმნა\n");
        exit(EXIT_FAILURE);
    }
    if( (pr = (int) fork()) == -1){ // პროცესის შექმნა
        printf("შეუძლებელია პროცესის წარმოქმნა\n");
        exit(EXIT_FAILURE);
    } else {
        if(pr != 0){
            // გავხსნათ FIFO არხი მონაცემების ჩასაწერად
            fd = open(name, O_WRONLY);
            if( fd < 0){
                printf("შეუძლებელია FIFO-ს გახსნა\n");
                exit(EXIT_FAILURE);
            }
            s1 = write(fd, mass, s2); // FIFO არხში ჩავწეროთ მონაცემები
            if(s1 != s2){
                printf("სრულად არ ჩაიწერა მონაცემები\n");
                exit(EXIT_FAILURE);
            }
            close(fd); // დავხურო კავშირის არხი
            wait(&st); // მშობელი ელოდება შვილის დასრულებას
        } else {
            // გავხსნათ FIFO არხი მონაცემების წასაკითხად
            fd = open(name, O_RDONLY);
            if( fd < 0){
                printf("შეუძლებელი FIFO-ს გახსნა\n");
                exit(EXIT_FAILURE);
            }
            // FIFO არხიდან წავიკითხოთ მონაცემები
        }
    }
}
```

```

s1 = read(fd, re_mass, s2);
if(s1 != s2){
    printf("ინფორმაცია სრულად ვერ იქნა წაკითხული \n");
    exit(EXIT_FAILURE);
}
printf("\n\t%os\n",re_mass); // გადაცემული მონაცემების ბეჭდვა
close(fd); // დავხურო კავშირის არხი
}
exit(EXIT_SUCCESS);
}

```

პროგრამა 04-4.c. FIFO-ს გამოყენებით ერთმიმართულებიანი კავშირი მშობელ- და შვილ-პროცესს შორის

აკრიფეთ პროგრამა, დააკომპილირეთ და შეასრულეთ ის.

შევნიშნოთ, რომ ამ პროგრამის ყოველი შემდეგი შესრულება მიგვიყვანს შეცდომამდე, რაც გამოწვეული იქნება პროგრამის შესრულების ყოველ ჯერზე სამუშაო დირექტორიაში fifo გაფართოების სპეციალური ფაილის არსებობით. პრობლემის გადაწყვეტა მოცემულ შემთხვევაში წარმოადგენს მისი შესრულების წინ ამ ფაილის წაშლა დირექტორიიდან, ან პირველი შესრულების შემდეგ განხორციელდებს პროგრამის ტექსტის რედაქტირება: იქიდან წაიშალოს ის ნაწილი, რომელიც დაკავშირებულია FIFO არხის წარმოქმნასთან ან მშობელ პროცესს მოვთხოვოთ, რომ სანამ დასრულდებოდეს წაშალოს FIFO არხი.

სემინარი 5. SystemVIPC საშუალებები. განაწილებადი მეხსიერება

წინა სემინარზე ჩვენ გავეცანით პროცესებს შორის კავშირის არხებით მონაცემების გადაცემის მექანიზმებს, კერძოდ, pipe-ს და FIFO-ს. კავშირის არხებით მონაცმების გაცვლის ეს მექანიზმები საკმაოდ ადვილია გამოყენებაში, მაგრამ გააჩნიათ მთელი რიგი ნაკლოვანებები:

- კითხვისა და ჩაწერის ოპერაციები არ აანალიზებენ გადასაცემი მონაცემების შემცველობას. ისინი არ ინტერესდებიან მიღებული ინფორმაცია გადაცემულია ერთი წყაროდან თუ ის სხვადასხვა წყაროდან შემოვიდა;
- პროცესებს შორის მონაცემების გადაცემისას საჭიროა კოპირების მინიმუმ ორი ოპერაცია: მონაცემების გადამცემი პროცესის მისამართების სივრციდან სისტემურ ბუფერში და სისტემური ბუფერიდან მიმღები პროცესის მისამართების სივრცეში.
- ინფორმაციის გაცვლის მომენტში გადამცემი და მიმღები პროცესები ერთდროულად უნდა არსებობდნენ სისტემაში. შეუძლებელია გადამცემმა პროცესმა კავშირის არხში განათავსოს მონაცემები და დასრულდეს, ხოლო გარკვეული დროის შემდეგ კი მიმღებმა პროცესმა განახორციელოს ინფორმაციის კითხვა არხიდან.

5.1. SystemVIPC-ის ცნება

კავშირის არხებით ინფორმაციის გადაცემის აღწერილმა ნაკლოვანებებმა საჭირო გახადა შექმნილიყო პროცესებს შორის ინფორმაციის გადაცემის სხვა მექანიზმები. ამ მექანიზმების ნაწილმა, რომელიც პირველად გამოჩნდა UNIX SystemV-ში და შემდეგ პრაქტიკულად UNIX ოპერაციული სისტემის ყველა თანამედროვე ვერსიაში, მიიღო სახელწოდება **SystemVIPC** (IPC - inter process communications). SystemVIPC ჯგუფში შედის: შეტყობინებათა მიმდევრობები, განაწილებადი მეხსიერება და სემაფორები. პროცესების ურთიერთქმედების ორგანიზაციის ეს საშუალებები დაკავშირებულია არამხოლოდ წარმოშობის ზოგადობით, არამედ მსგავსი ოპერაციების (მაგალითად, სისტემაში რესურსების გამოყოფა და გამოთავისუფლება) შესრულებისას მათ გააჩნიათ მსგავსი ინტერფეისი.

5.1.1. SystemVIPC სახელების სივრცე

SystemVIPC-დან კავშირის ყველა საშუალება, ისევე, როგორც ჩვენს მიერ განხილული pipe და FIFO, წარმოადგენს კავშირის საშუალებას არაპირდაპირი დამისამართებით. როგორც ზემოთ აღვნიშნეთ, არამონათესავე პროცესებს შორის კავშირის არხის მეშვეობით ურთიერთქმედების ორგანიზაციისთვის არაპირდაპირი დამისამართებისას საჭიროა, რომ კავშირის ამ საშუალებას გააჩნდეს სახელი. pipe-ისთვის სახელის არქონა მონათესავე პროცესებს შესაძლებლობას აძლევს მიიღონ ინფორმაცია სისტემაში მის ადგილმდებარეობასა და მდგომარეობაზე მხოლოდ მემკვიდრე კავშირებით. ფაილურ სისტემაში FIFO-სთვის სახელის მინიჭების (სისტემაში დარეგისტრირების) შესაძლებლობა არამერკვიდრე პროცესებს საშუალებას აძლევს მიიღონ ეს ინფორმაცია ფაილური სისტემის ინტერფეისის მეშვეობით.

რაიმე სახის ობიექტების შესაძლო სახელების სიმრავლეს ეწოდება შესაბამისი ტიპის ობიექტების სახელების სივრცე. FIFO-სთვის სახელების სივრცეს წარმოადგენს ფაილურ სისტემაში ფაილების შესაძლო სახელების სიმრავლე. SystemVIPC-სთვის სახელების ასეთ სივრცეს წარმოადგენს მონაცემთა გარკვეული მთელმნიშვნელობიანი რიცხვების (key_t გასაღებების) სიმრავლე. ამასთან, პროგრამისტს არ შეუძლია პირდაპირ მიანიჭოს გასაღების მნიშვნელობა, ეს მნიშვნელობა მოიცემა გასაშუალოებით: ფაილურ სისტემაში არსებული გარკვეული ფაილის სახელის და პატარა მთელი რიცხვის კომბინაციით. მაგალითად, კავშირის

საშუალების ასლის ნომრით.

გასაღების მნიშვნელობის მიღების ასეთი მეთოდი დაკავშირებულია შემდეგ მოსაზრებასთან:

- თუ პროგრამისტს ექნება კავშირის საშუალებისთვის საიდენტიფიკაციო ნომრის მინიჭების შესაძლებლობა, მაშინ გამორიცხული არაა, რომ ორმა პროგრამისტმა შემთხვევით გამოიყენოს ერთიდაიგივე რიცხვითი მნიშვნელობა. ამ შემთხვევაში, სხვადასხვა პროცესები შეეცდებიან არასანქცირებულად გამოიყენონ კომუნიკაციის საშუალებები, რამაც შეიძლება მიგვიყვანოს პროცესების არასტანდარტულ ყოფაქცევამდე. ამიტომ გასაღების მნიშვნელობის მირითად კომპონენტს წარმოადგენს იმ ფაილის სრული სახელის გარდაქმნა რიცხვით მნიშვნელობაში, რომელზეც პროცესებს გააჩნიათ წვდომა. ამ მიზნით პროგრამისტს შეუძლია გამოიყენოს საკუთარი სპეციფიური ფაილი, მაგალითად, შესრულებადი ფაილი, რომელიც დაკავშირებულია ურთიერთქმედ პროცესებიდან ერთერთთან. შევნიშნოთ, რომ პროცესების ურთიერთქმედების განმავლობაში არ უნდა ხდებოდეს გასაღების ფორმირებისთვის გამოყენებული ფაილის ადგილმდებარეობის შეცვლა;
- გასაღების მნიშვნელობის მეორე კომპონენტი გამოიყენება იმისთვის, რომ პროგრამისტს ჰქონდეს შესაძლებლობა ფაილის ერთსა და იმავე სახელს დაუკავშიროს კავშირის საშუალების ერთზე მეტი ასლი. ასეთი კომპონენტის როლში შესაძლებელია გამოდიოდეს შესაბამის ასლის რიგითი ნომერი.

ორი კომპონენტისგან გასაღების მნიშვნელობის მიღება ხორციელდება ფუნქციით **ftok()**.

ftok() ფუნქციის პროტოტიპი

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *pathname, char proj_id);
```

ფუნქციის აღწერა

ftok ფუნქცია გამოიყენება არსებული ფაილის სახელის პატარა მთელ რიცხვში გარდაქმნისთვის, მაგალითად, კავშირის საშუალების ეგზემპლარის რიგითი ნომრის SystemVIPC გასაღებში.

pathname პარამეტრი უნდა იყოს მიმთითებელი არსებულ ფაილზე, რომელზეც წვდომა გააჩნია **ftok** ფუნქციის გამომახებელ პროცესს.

proj_id - ეს არის პატარა მთელი რიცხვი, რომელიც ახასიათებს კავშირის საშუალების ასლს. გასაღების გენერაციის შეუძლებლობის შემთხვევაში ფუნქცია აბრუნებს უარყოფით მნიშვნელობას წინააღმდეგ შემთხვევაში აბრუნებს გენერირებული გასაღების მნიშვნელობას.

key_t მონაცემთა ტიპი წარმოადგენს 32-თანრიგა მთელ რიცხვს.

აღვნიშნოთ გასაღების მისაღებად ფაილის გამოყენების სამი მნიშვნელოვანი მომენტი:

- გასაღების გენერირებისთვის საჭიროა ფაილურ სისტემაში არსებული რეალური ფაილის სახელი, რომელზეც ინფორმაციის გამცვლელ პროცესებს ექნება კითხვის უფლება;
- გასაღების გენერირებისთვის გამოყენებული ფაილი არ უნდა იცვლიდეს ადგილმდებარეობას პროცესებს შორის ურთიერთქმედების დასრულებამდე;
- გასაღების გენერირებისთვის ფაილის სახელის მითითება არ ნიშნავს, რომ კავშირის საშუალებით გადასაცემი ინფორმაცია განთავსებული იქნება ამ ფაილში. ფაილის სახელის მითითება სხვადასხვა პროცესებს საშუალებას აძლევს დააგენერირონ იდენტური გასაღებები და ინფორმაციის გაცვლისთვის გამოიყენონ ოპერაციული სისტემის მიერ სპეციალურად ამ მიზნისთვის გამოყოფილი მისამართების სივრცე.

5.1.2. SystemVIPC დესკრიპტორი

როგორც ადრე აღვნიშნეთ, ოპერაციული სისტემა პროცესთან დაკავშირებულ ფაილებსა და კავშირის არხებზე ინფორმაციას ინახავს ფაილური დესკრიპტორების ცხრილში. სისტემური გამოძახება, რომლიც ანხორციელებს კავშირის არხით მონაცემების გადაცემას, პარამეტრის როლში იყენებს ფაილური დესკრიპტორების ცხრილის შესაბამის ჩანაწერს (ფაილურ დესკრიპტორს). ფაილური დესკრიპტორების გამოყენება პროცესების შიგნით იძლევა კავშირის არხის იდენტიფიცირების და ფაილებთან მუშაობისთვის უკვე არსებული ინტერფეისის გამოყენების შესაძლებლობას. პროცესის დასრულებისას მას მივყავართ კავშირის არხის ავტომატურ დახურვამდე რითაც აიხსნება კავშირის არხებით ინფორმაციის გადაცემის ზემოთ აღწერილი ნაკლოვანებები.

SystemVIPC-ის კომპონენტების რეალიზაციისას მიღებული იყო სხვა კონცეფცია. ერთერთი მათგანი მდგომარეობს შემდეგში: ოპერაციული სისტემის ბირთვი ინფორმაციას სისტემაში გამოყენებულ ყველა SystemVIPC საშუალებებზე ინახავს მომხმარებლის კონტექსტის გარეთ. თუ პროცესი საჭიროებს კავშირის საშუალების გამოყენებას, მაშინ ის ღებულობს წარმოქმნილი ან არსებული კავშირის საშუალების მაიდენტიფიცირებელ არაურყოფით მნიშვნელობას (დესკრიპტორს) და სარგებლობს ამ მნიშვნელობის შესაბამისი კავშირის საშუალებით. დესკრიპტორის მნიშვნელობა სისტემურ გამოძახებას, რომელიც SystemVIPC საშუალებებზე ანხორციელებს შემდგომ ოპერაციებს უნდა გადაეცემოდეს პარამეტრის სახით.

მსგავსი კონცეფცია იძლევა ნაკადების მეშვეობით კავშირის საშუალებების ერთერთ ყველაზე მნიშვნელოვანი ნაკლოვანების (ურთიერთქმედი პროცესების ერთდროულად არსებობის) აცილების შესაძლებლობას, მაგრამ იმავდროულად საჭიროებს განსაკუთრებულ სიფრთხილეს იმისთვის, რომ პროცესმა, რომელმაც მიიღო ინფორმაცია მათ ნაცვლად არ გაითვალისწინოს ძველი ინფორმაციის, რომელიც შემთხვევით დარჩა კომუნიკაციის მექანიზმში.

5.2. განაწილებადი მეხსიერება

პროცესებს შორის კავშირის ორგანიზაციის უზრუნველსაყოფად *SystemVIPC* კავშირის საშუალებები წინასწარ ითხოვენ ინიციალიზირებად მოქმედებებს. გარკვეული გასაღებით განაწილებადი მეხსიერების მიდამოს შექმნისთვის ან უკვე არსებულ განაწილებად მიდამოზე დაშვებისთვის გამოიყენება სისტემური გამოძახება *shmget()*. ახალი განაწილებადი მეხსიერების შესაქმნელად *shmget()* სისტემური გამოძახების გამოყენების არსებობს ორი მეთოდი:

- სტანდარტული მეთოდი.** სისტემურ გამოძახებაში გასაღების მნიშვნელობის როლში ყენდება *ftok()* ფუნქციის მიერ გარკვეული ფაილის სახელისთვის და განაწილებადი მეხსიერების მიდამოს ასლის ნომრისთვის გენერირებული მნიშვნელობა. flag-ების როლში ყენდება შესაქმნელ სეგმენტზე დაშვების უფლებების და *IPC_CREAT* flag-ის კომბინაციები. თუ სეგმენტი მოცემული გასაღებისთვის არ არსებობს, მაშინ სისტემა შეცდება მის შექმნას დაშვების მითითებული უფლებებით. თუ ის არსებობს, მაშინ ვიღებთ მის დესკრიპტორს. flag-ების ამ კომბინაციისთვის შესაძლებელია *IPC_EXCL* flag-ის დამატება. ეს flag-ი გარანტირებს სისტემური გამოძახების ნორმალურ დასრულებას მხოლოდ იმ შემთხვევაში, თუ სეგმენტი მართლა შეიქმნა (ანუ ის ადრე არ არსებობდა), თუ სეგმენტი არსებობდა, მაშინ სისტემური გამოძახება დასრულდება შეცდომით და <errno.h> ფაილში აღწერილი *errno* სისტემური ცვლადის მნიშვნელობა შეიცვლება *EEXIST*-ით (*errno* = *EEXIST*).
- არასტანდარტული მეთოდი.** გასაღების მნიშვნელობის როლში ეთითება სპეციალური მნიშვნელობა *IPC_PRIVATE*. *IPC_PRIVATE* მნიშვნელობის გამოყენებას ყოველთვის მივყავართ განაწილებადი მეხსიერების ახალი სეგმენტის შექმნის მცდელობამდე

მითითებული დაშვების უფლებებით და გასაღებით, რომელიც არ ემთხვევა არცერთი უკვე არსებული სეგმენტის გასაღების მნიშვნელობას და შეუძლებელია მიღებული იყოს *ftok()* ფუნქციის მეშვეობით. ამ შემთხვევაში ხდება *IPC_CREAT* და *IPC_EXCL* flag-ების მნიშვნელობების იგნორირება.

shmget() სისტემური გამოძახების პროტოტიპი

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

სისტემური გამოძახების აღწერა

shmget სისტემური გამოძახება გამოიყენება განაწილებადი მეხსიერების სეგმენტზე დაშვებისთვის და მისი წარმატებით დასრულების შემთხვევაში ის აბრუნებს ამ სეგმენტისთვის *System VIPC* დესკრიპტორს¹.

key პარამეტრი წარმოადგენს *System VIPC* გასაღებს სეგმენტისთვის, ანუ ფაქტიურად მის სახელს *System VIPC* სახელების სივრციდან. ამ პარამეტრის მნიშვნელობის როლში შეიძლება გამოყენებული იყოს *ftok()* ფუნქციის მეშვეობით მიღებული გასაღების მნიშვნელობა ან სპეციალური მნიშვნელობა *IPC_PRIVATE*. *IPC_PRIVATE* მნიშვნელობის გამოყენებას ყოველთვის მივყავართ განაწილებადი მეხსიერების ახალი სეგმენტის შექმნის მცდელობამდე გასაღებით, რომელიც არ ემთხვევა არსებული სეგმენტების გასაღების მნიშვნელობებს და არ შეიძლება მიღებული იყოს *ftok()*-ის მეშვეობით.

size პარამეტრი განსაზღვრავს არსებული ან შესაქმნელი სეგმენტის მოცულობას ბაიტებში. თუ სეგმენტი მითითებული გასაღებით უკვე არსებობს, მაგრამ მისი მოცულობა არ ემთხვევა size პარამეტრით მითითებულ მნიშვნელობას წარმოიშობა შეცდომა.

shmflg პარამეტრი გამოიყენება მხოლოდ განაწილებადი მეხსიერების ახალი სეგმენტის შექმნისას და განსაზღვრავს მომხმარებლების სეგმენტზე დაშვების უფლებებს, ასევე ახალი სეგმენტის შექმნის აუცილებლობას და სისტემური გამოძახების ყოფაქცევას შექმნის მცდელობისას. ის წარმოადგენს შემდეგი მნიშვნელობების გარკვეულ კომბინაციას (ბიტური ოპერაციით ან ('|')):

- *IPC_CREAT* – თუ სეგმენტი მითითებული გასაღებისთვის არ არსებობს, მაშინ ის უნდა შეიქმნას;
- *IPC_EXCL* – გამოიყენება *IPC_CREAT* flag-თან ერთად. მათი ერთობლივი გამოყენების და მითითებული გასაღებით სეგმენტის არსებობისას, არ ხორციელდება სეგმენტზე დაშვება და წარმოიშობა შეცდომის შემცველი სიტუაცია, ამასთან *<errno.h>* ფაილში აღწერილი *errno* ცვლადი ღებულობს მნიშვნელობას *EEXIST*;
- 0400 – სეგმენტის შემქმნელი მომხმარებლისთვის ნებადართულია კითხვა;
- 0200 – სეგმენტის შემქმნელი მომხმარებლისთვის ნებადართულია ჩაწერა;
- 0040 – სეგმენტის შემქმნელი მომხმარებლის ჯგუფისთვის ნებადართულია კითხვა;
- 0020 – სეგმენტის შემქმნელი მომხმარებლის ჯგუფისთვის ნებადართულია ჩაწერა;
- 0004 – სხვა მომხმარებლისთვის ნებადართულია კითხვა;
- 0002 – სხვა მომხმარებლისთვის ნებადართულია ჩაწერა.

დასაბრუნებელი მნიშვნელობა

სისტემური გამოძახება წარმატებით დასრულებისას განაწილებადი მეხსიერების სეგმენტისთვის აბრუნებს *System VIPC* დესკრიპტორის მნიშვნელობას და -1-ს შეცდომის წარმოქმნის შემთხვევაში.

შექმნილ განაწილებად მეხსიერებაზე წვდომა ხორციელდება მისი დესკრიპტორით, რომელსაც აბრუნებს სისტემური გამოძახება *shmget()*. უკვე არსებულ მიღამოზე დაშვება

¹ მთელი არაუარყოფითი მნიშვნელობა, რომელიც ცალსახად ახასიათებს გამოთვლითი სისტემის შიგნით სეგმენტს და მომავალში გამოიყენება მასთან სხვა ოპერაციებისას

შეიძლება განხორციელდეს შემდეგი ორი მეთოდით:

- თუ ვიყენებთ პროცესისთვის გენერირებულ გასაღებს, მაშინ `shmget()` გამოძახების გამოყენებით შეგვიძლია მივიღოთ განაწილებადი მეხსიერების დესკრიპტორი. ამ შემთხვევაში არ შეიძლება flag-ების შემადგენელ ნაწილად `IPC_EXCL` flag-ის მნიშვნელობის მითითება, ხოლო გასაღების მნიშვნელობა კი არ შეიძლება იყოს `IPC_PRIVATE`. ხდება დაშვების უფლებების იგნორირება, ხოლო მიდამოს მოცულობა უნდა ემთხვეოდეს მისი შექმნისას მითითებულ მოცულობას.
- ან შეგვიძლია ვისარგებლოთ იმითი, რომ *SystemVIPC* დესკრიპტორი ნამდვილია ოპერაციული სისტემის ფარგლებში და გადავცეთ მისი მნიშვნელობა განაწილებადი მეხსიერების შექმნელი პროცესებიდან მიმდინარე პროცესს.

დესკრიპტორის მიღების შემდეგ საჭიროა განაწილებადი მეხსიერების მიდამოს განთავსება მიმდინარე პროცესის მისამართების სივრცეში. ეს ხორციელდება `shmat()` სისტემური გამოძახების მეშვეობით. ნორმალურად დასრულების შემთხვევაში ის აბრუნებს მიმდინარე პროცესის მისამართების სივრცეში განაწილებადი მეხსიერების მისამართს. შემდგომი დაშვება ამ მისამართზე ხორციელდება პროგრამირების ენის ჩვეულებრივი საშუალებებით.

shmat() სისტემური გამოძახების პროტოტიპი

```
#include<sys/types.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg);
```

სისტემური გამოძახების აღწერა

`shmat` სისტემური გამოძახება გამოიყენება პროცესის მისამართების სივრცეში განაწილებადი მეხსიერების სეგმენტის განსათავსებლად.

`shmid` პარამეტრი წარმოადგენს განაწილებადი მეხსიერების სეგმენტისთვის *SystemVIPC* დესკრიპტორს, ანუ `shmget()` სისტემური გამოძახების მიერ დაბრუნებულ მნიშვნელობას.

`shmaddr` პარამეტრის როლში ვიყენებთ მნიშვნელობას `NULL`, რითაც ოპერაციულ სისტემას ეძლევა შესაძლებლობა განათავსოს განაწილებადი მეხსიერება პროცესის მისამართების სივრცეში.

`shmflg` პარამეტრისთვის გამოვიყენებთ ორ მნიშვნელობას: 0 - კითხვის და ჩაწერის ოპერაციების განსახორციელებლად ან `SHM_RDONLY` - თუ გვინდა მისგან მხოლოდ კითხვა. ამასთან პროცესს უნდა გააჩნდეს სეგმენტზე დაშვების შესაბამისი უფლება.

დასაბრუნებელი მნიშვნელობა

სისტემური გამოძახება წარმატებით დასრულების შემთხვევაში აბრუნებს განაწილებადი მეხსიერების მისამართს პროცესის მისამართების სივრცეში და -1-ს შეცდომის წარმოქმნის შემთხვევაში.

პროცესს განაწილებადი მეხსიერების გამოყენების დასრულების შემდეგ შეუძლია საკუთარი მისამართების სივრციდან ამოშალოს განაწილებადი მეხსიერება. განაწილებადი მეხსიერების ამოსაშლელად გამოიყენება `shmdt()` სისტემური გამოძახება. შევნიშნოთ, რომ `shmdt()` სისტემურ გამოძახება პარამეტრის მნიშვნელობის როლში ითხოვს პროცესის მისამართების სივრცეში განაწილებადი მეხსიერების მიდამოს დასაწყისის მითითებას, ანუ მნიშვნელობას, რომელიც დაბრუნა `shmat()` სისტემურმა გამოძახებამ, ამიტომ საჭიროა მოცემული მნიშვნელობის შენახვა განაწილებადი მეხსიერების გამოყენების მთელს პერიოდში.

shmdt() სისტემური გამოძახების პროტოტიპი

```
#include<sys/types.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

სისტემური გამოძახების აღწერა

shmdt სისტემური გამოძახება გამოიყენება მიმდინარე პროცესის მისამართების სივრციდან განაწილებადი მეხსიერების სეგმენტის ამოსაშლელად.

shmaddr პარამეტრი წარმოადგენს განაწილებადი მეხსიერების სეგმენტის მისამართს ანუ, მნიშვნელობას რომელიც დააბრუნა სისტემურმა გამოძახებამ *shmat()*.

დასაბრუნებელი მნიშვნელობა

სისტემური გამოძახება აბრუნებს 0-ს ნორმალურად დასრულებისას და -1-ს შეცდომის წარმოქმნის შემთხვევაში.

განაწილებადი მეხსიერების ორგანიზების საილუსტრაციოდ განვიხილოთ მაგალითი. ვთქვათ, გვაქვს ორი პროცესი, რომლებიც იყენებენ განაწილებად მეხსიერებას. განაწილებად მეხსიერებაში განვათავსებთ სამელემენტიან მასივს, რომლის პირველ ორ ელემენტში ჩაიწერება, შესაბამისად, პირველი და მეორე პროცესის (ცალცალკე) შესრულებათა რაოდენობა, ხოლო მესამეში მათი შესრულებათა საერთო რაოდენობა.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(){
    int *mass; /* მიმთითებელი განაწილებად მეხსიერებაზე */
    int shm_id; /* ცვლადი განაწილებადი მეხსიერების სეგმენტის იდენტ. */
    int flag = 1; /* მასივის ელემენტების ინიციალიზაციის აუცილებლობის flag-ი */
    /* ფაილი, რომელსაც გავიყენებთ SystemVIPC გასაღების ფორმირებისთვის */
    char name[] = "prog-5.1.c";
    key_t key;
    /* ფაილისთვის SystemVIPC გასაღების გენერირება */
    if ((key = ftok(name,0)) < 0){
        perror("შეუძლებელია გასაღების გენერირება\n");
        exit(EXIT_FAILURE);
    }
    /* განაწილებადი მეხსიერების შემქნა */
    shm_id = shmget(key, 3*sizeof(int), IPC_CREAT | IPC_EXCL | 0644);
    if (shm_id < 0){
        /* შეცდომა გამოწვეულია განაწილებადი მეხსიერების არსებობით */
        if (errno == EEXIST){
            /* არსებული განაწილებადი მეხსიერების დესკრიპტორის მიღება */
            shm_id = shmget(key, 3*sizeof(int), 0);
            if(shm_id < 0){
                perror("შეუძლებელია განაწილებადი მეხსიერების მოძებნა\n");
                exit(EXIT_FAILURE);
            }
            flag = 0;
        }
    }
```

```

else { /* შეცდომა გამოწვეულია გაურკვეველი მიზეზით */
    perror("შეუძლებელია განაწილებადი მეხსიერების შექმნა\n");
    exit(EXIT_FAILURE);
}
/* მასივის განთავსება განაწილებად მეხსიერებაში */
mass = (int *) shmat(shm_id, NULL, 0);
if(mass == (int *)(-1)) {
    perror("შეუძლებელია მასივის განთავსება განაწილებად მეხსიერებაში\n");
    exit(EXIT_FAILURE);
}
if (flag){
    mass[0] = 1; /* პროცესი 1-ის შესრულებათა დასათვლელად */
    mass[1] = 0; /* პროცესი 2-ის შესრულებათა დასათვლელად */
    mass[2] = 1; /* ორივე პროცესის ჯამური შესრულებების დასათვლელად */
} else {
    mass[0] += 1;
    mass[2] += 1;
}
printf("\tპროცესი 1\tპროცესი 2\tპროცესი 3\n");
printf("\t %d\t\t %d\t\t %d\n\n", mass[0], mass[1], mass[2]);
/* პროცესის მისამართების სივრციდან განაწილებადი მეხსიერების წაშლა */
if (shmctl(mass) < 0){
    perror("შეუძლებელია განაწილებადი მეხსიერების წაშლა\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}

```

prog-5.1.c

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main(){
    int *mass; /* მიმთითებელი განაწილებად მეხსიერებაზე */
    int shm_id; /* ცვლადი განაწილებადი მეხსიერების სეგმენტის იდენტ. */
    int flag = 1; /* მასივის ელემენტების ინიციალიზაციის აუცილებლობის flag-ი */
    /* ფაილი, რომელსაც გავიყენებთ SystemVIPC გასაღების ფორმირებისთვის */
    char name[] = "prog-5.1.c";
    key_t key;
    /* ფაილისთვის SystemVIPC გასაღების გენერირება */
    if (key = ftok(name,0) < 0){

```

```

perror("შეუძლებელია გასაღების გენერირება\n");
exit(EXIT_FAILURE);
}

/* განაწილებადი მეხსიერების შემქნა */
shm_id = shmget(key, 3*sizeof(int), IPC_CREAT | IPC_EXCL | 0644);
if (shm_id < 0){
    /* შეცდომა გამოწვეულია განაწილებადი მეხსიერების არსებობით */
    if (errno == EEXIST){
        /*არსებული განაწილებადი მეხსიერების დესკრიპტორის მიღება */
        shm_id = shmget(key, 3*sizeof(int), 0);
        if(shm_id < 0){
            perror("შეუძლებელია განაწილებადი მეხსიერების მოძებნა\n");
            exit(EXIT_FAILURE);
        }
        flag = 0;
    }
    else { /* შეცდომა გამოწვეულია გაურკვეველი მიზეზით */
        perror("შეუძლებელია განაწილებადი მეხსიერების შექმნა\n");
        exit(EXIT_FAILURE);
    }
}
/* მასივის განთავსება განაწილებად მეხსიერებაში */
mass = (int *) shmat(shm_id, NULL, 0);
if(mass == (int *)(-1)){
    perror("შეუძლებელია მასივის განთავსება განაწილებად
მეხსიერებაში\n");
    exit(EXIT_FAILURE);
}
if (flag){
    mass[0] = 0; /* პროცესი 1-ის შესრულებათა დასათვლელად */
    mass[1] = 1; /* პროცესი 2-ის შესრულებათა დასათვლელად */
    mass[2] = 1; /* ორივე პროცესის ჯამური შესრულებების დასათვლელად */
} else {
    mass[1] += 1;
    mass[2] += 1;
}
printf("\tპროცესი 1\tpროცესი 2\tპროცესი 3\n");
printf("\t %d\t %d\t %d\n", mass[0], mass[1], mass[2]);
/* პროცესის მისამართების სივრციდან განაწილებადი მეხსიერების წაშლა */
if (shmdt(mass) < 0){
    perror("შეუძლებელია განაწილებადი მეხსიერების წაშლა\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}

```

ეს ორი პროგრამა ძალიან წააგავს ერთიმეორებს. ისინი განაწილებად მეხსიერებას იყენებენ თითოეული პროგრამის შესრულების რაოდენობის და მათი შესრულების საერთო რაოდენობის შესანახად. განაწილებადი მეხსიერების სეგმენტში განთავსებულია სამელემენტიანი მასივი. მასივის პირველი ელემენტი გამოიყენება, როგორც პროგრამა 1-ის შესრულების მთვლელი, მეორე ელემენტი - პროგრამა 2-ის შესრულების მთვლელი, ხოლო მესამე ელემენტი კი მათი ჯამისთვის. დამატებითი ნიუანსი პროგრამაში წარმოიშობა განაწილებადი მეხსიერების შექმნისას მასივის ელემენტების ინიციალიზაციის აუცილებლობით. ამისთვის გვჭირდება, რომ პროგრამებმა შეძლონ განსხვავება შემთხვევისა როდის შექმნეს მათ ის და შემთხვევისა, ის რომ არსებობდა. განსხავებას ვაღწევთ დასაწყისში `shmget()` სისტემური გამოძახების გამოყენებით flag-ებით `IPC_CREAT` და `IPC_EXCL`. თუ გამოძახება დასრულდება წარმატებით, მაშინ შეიქმნება განაწილებადი მეხსიერება. თუ გამოძახება დასრულდა შეცდომით და `errno` ცვლადმა მიიღო მნიშვნელობა `EEXIST`, მაშინ ეს ნიშნავს, რომ განაწილებადი მეხსიერება უკვე არსებობს და შეგვიძლია მივიღოთ მისი IPC დესკრიპტორი იმავე გამოძახების გამოყენებით flag-ის ნულოვანი მნიშვნელობის მითითებით.

5.3. განაწილებადი მეხსიერების ყოფაქცევა

მნიშვნელოვან საკითხს წარმოადგენს განაწილებადი მეხსიერების სეგმენტის ყოფაქცევა სხვადასხვა სისტემური გამოძახებების შესრულებისას, კერძოდ, `fork()` და `exec()` სისტემური გამოძახებებისა და `exit()` ფუნქციის შესრულებისას.

`fork()` სისტემური გამოძახების შესრულებისას შვილი პროცესი მემკვიდრეობით იღებს ინფორმაციას მშობელი პროცესისაგან მის მისამართების სივრცეში განთავსებულ ყველა განაწილებადი მეხსიერების მიდამოზე.

`exec()` სისტემური გამოძახებისა და `exit()` ფუნქციის შესრულებისას პროცესის მისამართების სივრცეში განთავსებული განაწილებადი მეხსიერების მიდამოები იშლება იქიდან, მაგრამ არსებობას განაგრძობენ ოპერაციულ სისტემაში.

ამოცანა 1. დაწერეთ პროგრამა, რომელშიც პროცესი შექმნის ორ შვილ პროცესს. პირველი შვილი პროცესი ფაილიდან წაიკითხავს `N` მთელ მნიშვნელობას და ჩაწერს განაწილებად მეხსიერებაში, ხოლო მეორე შვილი პროცესი დაითვლის ამ მონაცემთა საშუალო არითმეტიკულს და დაბეჭდავს

```
#include <fstream>
#include <iostream>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/wait.h>
#include <sys/types.h>
using namespace std;
int main(){
    int N = 10; // ფაილში ელემენტების რაოდენობა
    int *arrow; // ამ მისამართზე განთავსდება განაწილებადი მეხსიერება
```

```

int shmid; // ცვლადი განაწილებადი მეხსიერების დესკრიპტორისთვის
int st1, st2; // სტატუსის ცვლადები
pid_t p1, p2; // ცვლადების პროცესის შექმნის შესამოწმებლად
key_t key; // ცვლადი გასაღების მნიშვნელობისთვის
// მოვახდინოთ გასაღების გენერაცია SheMem.cpp ფაილისთვის და შემოწმება
key = ftok("prog-5.3.cpp", 0);
if (key == -1) {
    cout << "გასაღები მითითებული ფაილისთვის არ შეიქმნა.\n";
    exit(EXIT_FAILURE);
}
// გენერირებული გასაღებისთვის შევქმნათ საჭირო მოცულობის განაწილებადი მეხსიერება
shmid = shmget(key, N*sizeof(int), IPC_CREAT | IPC_EXCL | 0600);
if (shmid == -1){ // თუ განაწილებადი მეხსიერება არ შეიქმნა
    if (errno == EEXIST){ // ასეთი განაწილებადი მეხსიერება ხომ არ არსებობს?
        shmid = shmget(key, N*sizeof(int), 0); // გამოვიყენოთ არსებული განაწილებადი მეხსიერება
        if (shmid == -1){
            cout << "ვერ მოხერხდა არსებული განაწილებადი მეხსიერების დესკრიპტორის მიღება.\n";
            exit(EXIT_FAILURE);
        }
    }
}
else {
    cout << "განაწილებადი მეხსიერება არ შეიქმნა სხვა მიზეზით.\n";
    exit(EXIT_FAILURE);
}
// შევქმნათ პროცესი
p1 = fork();
if (p1 == -1){
    cout << "პირველი შვილი პროცესი არ შეიქმნა.\n";
    exit(EXIT_FAILURE);
}
if (p1 == 0){ // პირველი შვილი პროცესი
    // პირველი შვილი პროცესის მისამართების სივრცეს მივაბათ განაწილებადი მეხსიერება
    arrow = (int*)shmat(shmid, NULL, 0);
    if (arrow == (int*)(-1)){
        cout << "ვერ მოხერხდა განაწილებადი მეხსიერების მიერთება"
        << " პროცესის მისამართების სივრცეზე.\n";
        exit(EXIT_FAILURE);
    }
    int tmp; // დროებითი ცვლადი ფაილიდან წაკითხული მონაცემების ჩასაწერად
    ifstream Input("data.in"); // შევქმნათ მონაცემთა ნაკადი
    for (int i = 0; i<N; i++){
        Input >> tmp;
        arrow[i] = tmp;
    }
}

```

```

Input.close(); // დავხუროთ მონაცემთა ნაკადი
// განაწილებად მეხსიერებაში მთელი მნიშვნელობების ჩაწერის შემდეგ პროცესის
// მისამართების სივრციდან ამოვშალოთ განაწილებადი მეხსიერება
if (shmdt(arrow) == -1)
    cout << "არ მოხდა განაწილებადი მეხსიერების წაშლა.\n";
    // პირველი პროცესის დასრულდა
}
else {
    wait(&st1); // მშობელი პროცესი ელოდება პირველი შვილის დასრულებას
    // შევქმნათ პროცესი
    p2 = fork();
    if (p2 == -1){
        cout << "მეორე შვილი პროცესი არ შეიქმნა.\n";
        exit(EXIT_FAILURE);
    }
    if (p2 == 0){ // მეორე შვილი პროცესი
        // მეორე შვილი პროცესის მისამართების სივრცეს მივაბათ განაწილებადი მეხსიერება
        arrow = (int*)shmat(shmid, NULL, 0);
        if (arrow == (int*)(-1)){
            cout << "ვერ მოხერხდა განაწილებადი მეხსიერების მიერთება"
                << " პროცესის მისამართების სივრცეზე.\n";
            exit(EXIT_FAILURE);
        }
        double average = 0; // საშუალო მნიშვნელობის დასათვლელად
        // average ცვლადში ჩავწეროთ განაწილებად მეხსიერებაში არსებული რიცხვების ჯამი
        for (int i = 0; i<N; i++){
            average += arrow[i];
        }
        // პროცესის მისამართების სივრციდან ამოვშალოთ განაწილებადი მეხსიერება
        if (shmdt(arrow) == -1)
            cout << "არ მოხდა განაწილებადი მეხსიერების წაშლა.\n";
        // დავბეჭდოთ საშუალო არითმეტიკულის მნიშვნელობა
        cout << "average = " << average / N << endl;
        // მეორე პროცესის დასრულდა
    }
    else { // მშობელი პროცესი
        wait(&st1); // მშობელი პროცესი ელოდება მეორე შვილის დასრულებას
        // მშობელი პროცესი დასრულდა
    }
}
exit(EXIT_SUCCESS);
}

```

prog-5.3.cpp

ამოცანა 2. დაწერეთ პროგრამა, რომელშიც პროცესი შექმნის ორ შვილ პროცესს. პირველი შვილი პროცესი სტრიქონს ჩაწერს განაწილებად მეხსიერებაში, ხოლო მეორე შვილი პროცესი განაწილებად მეხსიერებაში არსებულ მონაცემებს ჩაწერს ფაილში, ხოლო მშობელი პროცესი ტერმინალში გამოიტანს ფაილის შიგთავსი.

```
#include <fstream>
#include <iostream>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/wait.h>
#include <sys/types.h>
using namespace std;
int main(){
    char str[] = "Operating System."; // ფაილში ელემენტების რაოდენობა
    size_t N = sizeof(str);
    char *arrow; // ამ მისამართზე განთავსდება განაწილებადი მეხიერება
    int shmid; // ცვლადი განაწილებადი მეხსიერების დესკრიპტორისთვის
    int st1, st2; // სტატუსის ცვლადები
    pid_t p1, p2; // ცვლადების პროცესის შექმნის შესამოწმებლად
    key_t key; // ცვლადი გასაღების მნიშვნელობისთვის
    // მოვახდინოთ გასაღების გენერაცია SheMem.cpp ფაილისთვის და შემოწმება
    key = ftok("prog-5.4.cpp", 0);
    if (key == -1) {
        cout << "გასაღები მითითებული ფაილისთვის არ შეიქმნა.\n";
        exit(EXIT_FAILURE);
    }
    // გენერირებული გასაღებისთვის შევქმნათ საჭირო მოცულობის განაწილებადი მეხსიერება
    shmid = shmget(key, N, IPC_CREAT | IPC_EXCL | 0600);
    if (shmid == -1){ // თუ განაწილებადი მეხსიერება არ შეიქმნა
        if (errno == EEXIST){ // ასეთი განაწილებადი მეხსიერება ხომ არ არსებობს?
            shmid = shmget(key, N, 0); // გამოვიყენოთ არსებული განაწილებადი მეხსიერება
            if (shmid == -1){//
                cout << "ვერ მოხერხდა არსებული განაწილებადი მეხსიერების დესკრიპტორის
მიღება.\n";
                exit(EXIT_FAILURE);
            }
        }
    }
    else {
        cout << "განაწილებადი მეხსიერება არ შეიქმნა სხვა მიზეზით.\n";
        exit(EXIT_FAILURE);
    }
}
// შევქმნათ პროცესი
```

```

p1 = fork();
if (p1 == -1){
    cout << "პირველი შვილი პროცესი არ შეიქმნა.\n";
    exit(EXIT_FAILURE);
}
if (p1 == 0){ // პირველი შვილი პროცესი
    // პირველი შვილი პროცესის მისამართების სივრცეს მივაბათ განაწილებადი მეხსიერება
    arrow = (char*)shmat(shmid, NULL, 0);
    if (arrow == (char*)(-1)){
        cout << "ვერ მოხერხდა განაწილებადი მეხსიერების მიერთება"
            << " პროცესის მისამართების სივრცეზე.\n";
        exit(EXIT_FAILURE);
    }
    // ჩავწეროთ განაწილებად მეხსიერებაში სტრიქონი
    for (int i = 0; i<N; i++){
        arrow[i] = str[i];
    }
    // განაწილებად მეხსიერებაში მთელი მნიშვნელობების ჩაწერის შემდეგ პროცესის
    // მისამართების სივრციდან ამოვშალოთ განაწილებადი მეხსიერება
    if (shmdt(arrow) == -1)
        cout << "არ მოხდა განაწილებადი მეხსიერების წაშლა.\n";
    // პირველი პროცესესი დასრულდა
}
else {
    wait(&st1); // მშობელი პროცესი ელოდება პირველი შვილის დასრულებას
    // შევქმნათ პროცესი
    p2 = fork();
    if (p2 == -1){
        cout << "მეორე შვილი პროცესი არ შეიქმნა.\n";
        exit(EXIT_FAILURE);
    }
    if (p2 == 0){ // მეორე შვილი პროცესი
        // მეორე შვილი პროცესის მისამართების სივრცეს მივაბათ განაწილებადი მეხსიერება
        arrow = (char*)shmat(shmid, NULL, 0);
        if (arrow == (char*)(-1)){
            cout << "ვერ მოხერხდა განაწილებადი მეხსიერების მიერთება"
                << " პროცესის მისამართების სივრცეზე.\n";
            exit(EXIT_FAILURE);
        }
        // შევქმნათ მონაცემთა გამოტანის ნაკადი
        ofstream Out("Data.out");
        for (int i = 0; i<N; i++){
            Out << arrow[i];
        }
        Out.close();
    }
}

```

```

// პროცესის მისამართების სივრციდან ამოვშალოთ განაწილებადი მეხსიერება
if (shmctl(shmid, SHM_RND, 0) == -1)
    cout << "არ მოხდა განაწილებადი მეხსიერების წაშლა.\n";
// მეორე პროცესის დასრულდა
}
else { // მშობელი პროცესი
    wait(&st1); // მშობელი პროცესი ელოდება მეორე შვილის დასრულებას
    execl("./bin/cat", "cat", "Data.out", NULL);
    // მშობელი პროცესი დასრულდა
}
}
exit(EXIT_SUCCESS);
}

```

prog-5.4.cpp

კრიტიკული სექციის მკაცრი მიმდევრობის ალგორითმის გამოყენებით ამოვხსნათ კიდევ ერთი ამოცანა.

ვთქვათ, მშობელ პროცესს (main პროცესს) ყავს ორი შვილი (P₁ და P₂), რომელთაგან პირველი პროცესი (P₁) განახორციელებს 10 ელემენტიანი განაწილებადი მეხსიერების შევსებას [1, 100] შუალედიდან შემთხვევითი პრინციპით აღებული მნიშვნელობებით, ხოლო მეორე პროცესი (P₂) ამ მნიშვნელობებს შორის იპოვის უდიდესს და დაბეჭდავს შედეგს. ამასთან, მშობელი პროცესი შემთხვევითი პრინციპით განსაღვრავს თუ რომელმა პროცესმა უნდა განახორციელოს პირველი საქმიანობა.

ამოცანის პირობის თანახმად რადგანაც უნდა მოხდეს P₁ და P₂ პროცესების სინქრონული მუშაობის უზრუნველყოფა, ამიტომ საჭიროა კიდევ ერთი განაწილებადი მეხსიერება, რომელშიც ჩაიწერება იმ პროცესის ნომერი (1 ან 2), რომელიც პირველი დაიწყებს შესრულებას. ამ განაწილებად მეხსიერებაზე წვდომა უნდა ჰქონდეს მშობელ და შვილ პროცესებს, ხოლო 10 ელემენტიან განაწილებად მეხსიერებაზე კი მხოლოდ შვილ პროცესებს. პროგრამის შესაბამის კოდს ექნება სახე:

```

#include <iostream>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
using namespace std;
int main(){
    // შემოვიღოთ აუცილებელი ცვლადები
    int k1, k2, // ორი ცვლადი ორი გასაღებისთვის
    sId1, sId2, // განაწილებადი მეხსიერების იდენტიფიკატორი
    *shm1, *shm2, // განაწილებად მეხსიერებაზე მიმთითებელი
    N = 10; // ელემენტების რაოდენობა განაწილებად მეხსიერებაში
}

```

```

// შევქმნათ პირველი გასაღები განაწილებადი მეხსიერებისთვის
k1 = ftok("prog.cpp", 1);

// შევქმნათ პირველი განაწილებადი მეხსიერება
sId1 = shmget(k1, sizeof(int), IPC_CREAT | 0600);

// მოვახდინოთ პირველი განაწილებადი მეხსიერების მიერთება main-თან
shm1 = (int*)shmat(sId1, NULL, 0);
srand(time(NULL));
// პირველ განაწილებად მეხსიერებაში ჩავწეროთ შემთხვევითი მნიშვნელობა (1 ან 2)
shm1[0] = rand() % 2 + 1; // ეს მნიშვნელობა განსაზღვრავს პროცესის ნომერს

// main-დან გამოვაერთოთ განაწილებადი მეხსიერება რადგანაც მას უკვე არ ვსაჭიროებთ
shmdt(&sId1);

if(fork() == 0){ // შევქმნათ პირველი პროცესი
    // პირველი პროცესთან მივაერთოთ პირველი განაწილებადი მეხსიერება
    shm1 = (int*)shmat(sId1, NULL, 0);

    // რადგანაც პირველ განაწილებად მეხსიერებაში მშობლის (main-ის) მიერ
    // შესაძლებელია თანაბრად ჩაწერილი იყოს როგორც 1-ი ისე 2-ი, ამიტომ თუ
    // განაწილებად მეხსიერებაში შეგვხვდა 2-ი უსასრულოდ უნდა დაველოდოთ
    // მასში 1-ის ტოლი მნიშვნელობის გამოჩენას
    while(true){
        // თუ განაწილებად მეხსიერებაში წერია 2-ის ტოლი მნიშვნელობა, მაშინ უნდა
        // შევიდეთ უსასრულო ციკლში სანამ მნიშვნელობა არ გახდება 1-ის ტოლი
        while(shm1[0] == 2);

        // თუ გამოვცდით უსასრულო ციკლს, ანუ shm1[0] == 1, მაშინ საჭიროა
        // შევქმნათ ახალი გასაღები მეორე განაწილებადი მეხსიერებისთვის
        k2 = ftok("prog.cpp", 2);

        // შევქმნათ მეორე განაწილებადი მეხსიერება
        sId2 = shmget(k2, N * sizeof(int), IPC_CREAT | IPC_EXCL | 0600);

        // თუ განაწილებადი მეხსიერება არ შეიქმნა მისი არსებობის მიზეზით
        if(sId2 == -1 && errno == EEXIST){
            // მაშინ გამოვიყენოთ არსებული განაწილებადი მეხსიერება
            sId2 = shmget(k2, N * sizeof(int), 0);
        }

        // მივაერთოთ პირველი შვილ პროცესს მეორე განაწილებადი მეხსიერება
        shm2 = (int*)shmat(sId2, NULL, 0);
}

```

```

// შევავსოთ განაწილებადი მეხსიერება შემთხვევითი მნიშვნელობებით [1, 100]
// შუალედიდან
for(int i=0;i<N ;i++){
    shm2[i] = rand() % 100 + 1;
}

// გამოვაერთოთ მეორე განაწილებადი მეხსიერება პირველი შვილი პროცესიდან
shmdt(&sId2);

// დავუთმოთ მეორე შვილ პროცესს შესრულების შესაძლებლობა
shm1[0] = 2;
break; // დავასრულოთ უსასრულო ციკლი
}

// პირველი პროცესიდან გამოვაერთოთ პირველი განაწილებადი მეხსიერება
shmdt(&sId1);
} else {
if(fork() == 0){ // შევქმნათ მეორე პროცესი
    // მეორე პროცესთან მივაერთოთ პრველი განაწილებადი მეხსიერება
    shm1 = (int*)shmat(sId1, NULL, 0);
    // რადგანაც პირველ განაწილებად მეხსიერებაში მშობლის (main-ის) მიერ

    // შესაძლებელია თანაბრად ჩაწერილი იყოს როგორც 1-ი ისე 2-ი, ამიტომ თუ
    // განაწილებად მეხსიერებაში შეგვხვდა 2-ი უსასრულოდ უნდა დაველოდოთ
    // მასში 2-ის ტოლი მნიშვნელობის გამოჩენას
    while(true){
        // თუ განაწილებად მეხსიერებაში წერია 1-ის ტოლი მნიშვნელობა, მაშინ უნდა
        // შევიდეთ უსასრულო ციკლში სანამ მნიშვნელობა არ გახდება 2-ის ტოლი
        while(shm1[0] == 1);

        // თუ გამოვცდით უსასრულო ციკლს, ანუ shm1[0] == 2, მაშინ საჭიროა
        // შევქმნათ ახალი გასაღები მეორე განაწილებადი მეხსიერებისთვის
        k2 = ftok("prog.cpp", 2);

        // შევქმნათ მეორე განაწილებადი მეხსიერება
        sId2 = shmget(k2, N * sizeof(int), IPC_CREAT | IPC_EXCL | 0600);

        // თუ განაწილებადი მეხსიერება არ შეიქმნა მისი არსებობის მიზეზით
        if(sId2 == -1 && errno == EEXIST){
            // მაშინ გამოვიყენოთ არსებული განაწილებადი მხსიერება
            sId2 = shmget(k2, N * sizeof(int), 0);
        } else {
            // თუ განაწილებადი მეხსიერების შემქმნელად გვევლინე მეორე პროცესი ეს
            // ნიშნავს, რომ განაწილებადი მეხსიერება ჯერ არაა შევსებული და საჭიროა მის
            // შესავსებად გზა დავუთმოთ პირველ შვილ პროცესს

```

```

shm1[0]=1;
continue; // გავაგრძელოთ ციკლი
}
// მივაერთოთ მეორე შვილ პროცესს მეორე განაწილებადი მეხსიერება
shm2 = (int*)shmat(sId2, NULL, 0);

// მოვახდინოთ განაწილებად მეხსიერებაში განთავსებულ მნიშვნელობებს შორის
// უდიდესი მნიშვნელობის დადგენა
int max = shm2[0];
for(int i = 1; i < N; i++){
    if (max < shm2[i]){
        max = shm2[i];
        break; // შევწყვიოთ ციკლი
    }
}
// გამოვაერთოთ მეორე განაწილებადი მეხსიერება პირველი შვილი პროცესიდან
shmdt(&sId2);

// დავტეჭდოთ მაქსიმალური მნიშვნელობა
cout << "უდიდესი ელემენტი არის - " << max << endl;

// დავუთმოთ პირველ შვილ პროცესს შესრულების შესაძლებლობა
shm1[0] = 1;
break; // დავასრულოთ უსასრულო ციკლი
}
// მეორე პროცესიდან გამოვაერთოთ პირველი განაწილებადი მეხსიერება
shmdt(&sId1);
} else {
    // შვილი პროცესების დასრულებამდე შევაჩეროთ მშობელი პროცესის დასრულება
    sleep(1);
    cout << "მშობელი პროცესი დასრულდა!\n";
}
}
return 0;
}

```

prog-5.5.cpp

სემინარი 6. ურთიერთბლოკირება

თანამედროვე კომპიუტერი აღჭურვილია მრავალი რესურსით. სისტემაში წარმოქმნილი ყოველი პროცესი მიმართავს მას იმ რესურსის გამოყოფის მოთხოვნით, რომელსაც საჭიროებს დასახული ამოცანის გადასაწყვეტად. დროის ნებისმიერ მომენტში კონკრეტული რესურსი შეიძლება გამოყოს მხოლოდ ერთ პროცესს. ხშირად რამდენიმე პროცესი შეიძლება მიმართავდეს ოპერაციულ სისტემას ერთიდაიმავე რესურსის გამოყოფაზე. პროცესებისთვის საზიარო რესურსის გამოყოფა შესაძლებელია დაკავშირებული იყოს გარკვეულ პრობლემებთან, ამიტომ ოპერაციული სისტემა პროცესებს რესურსებს უყოფს მცირე დროითი შუალედით, მათზე დაშვების სრული უფლებებით.

ხშირად პროცესები საკუთარი სამუშაოს შესასრულებლად საჭიროებენ ერთზე მეტ რესურსს. ოპერაციულმა სისტემამ პროცესის მოთხოვნილი რესურსები შეიძლება გამოუყოს მოთხოვნილი მიმდევრობის შესაბამისად სათითაოდ ან ერთიანად. ორივე მიდგომას აქვს საკუთარი უპირატესობა და ნაკლოვანება. პროცესებისთვის რესურსის გამოყოფის შემთხვევაში შეიძლება წარმოქმნას სიტუაცია, რომლის დროსაც შეიძლება გამოიყოს პროცესების ჯგუფი, რომლებიც იკავებენ სისტემის გარკვეულ რესურსს და ამავდროულად ითხოვენ ამავე ჯგუფის სხვა პროცესის მიერ დაკავებული რესურსის გამოყოფას. ვინაიდან პროცესმა შეიძლება არ გამოათავისუფლოს მისთვის გამოყოფილი რესურსი დასრულებამდე, ამიტომ სისტემა ვერ შეძლებს პროცესების ჯგუფის ვერცერთი პროცესის მიერ გაკეთებული მოთხოვნის დაკავილებას. წარმოქმნილ სიტუაციას ეწოდება **ურთიერთბლოკირება (deadlock) ანჩიხი**.

6.1. რესურსები

როგორც უკვე აღვნიშნეთ, კომპიუტერულ სისტემას გააჩნია მრავალი რესურსი, რომელთაც ის სთვაზობს პროცესებს შესასრულებელი სამუშაოს განსახორციელებლად. რესურსი შეიძლება იყო აპარატული (მყარი დისკი, პროცესორი, მეხსიერება) ან ინფორმაციული (ფაილები, მონაცემთა ბაზა).

ოპერაციულ სისტემაში წარმოდგენილი ყოველი რესურსი შეიძლება იყოს ორი სახის: **განაწილებადი ან გაუნაწილებელი**. პროცესის მიერ დაკავებულ რესურსს, რომლის ჩამორთმევაც მისთვის შესაძლებელია „უმტკივნეულოდ“ (შესრულებული მნიშვნელოვანი სამუშაოს დაკარგვის გარეშე), მიეკუთვნება განაწილებად რესურსს. გაუნაწილებელი კი პირიქით. მაგალითად, განაწილებადი რესურსის მაგალითს წარმოადგენს მეხსიერება. სისტემაში არასაკმარის მეხსიერების არსებობის შემთხვევაში ხორციელდება პროცესის სრული ან ნაწილობრივი გადატანა მეხსიერების არიდან დისკზე, ხოლო მოგვიანებით კი მისი უკან დაბრუნება.

ოპერაციულმა სისტემამ გაუნაწილებელი რესურსი შეიძლება გამოუყოს მხოლოდ ერთ პროცესს. პროცესი ასეთ რესურსს დააბრუნებს მხოლოდ დასრულების შემდეგ.

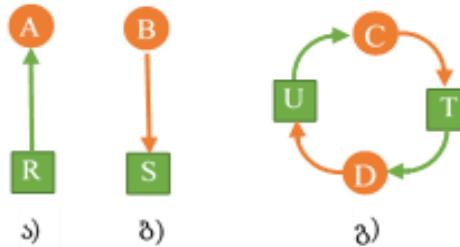
ურთიერთბლოკირების წარმოშობა სისტემაში დაკავშირებულია იმასთან, რომ რესურსი სისტემაში შეიძლება წარმოდგენილი იყოს მხოლოდ ერთი ასლის სახით. თუ რესურსები იქნებოდა წარმოდგენილი რამდენიმე ასლის სახით, რამდენიმე პროცესისთვის შესაძლებელი იქნებოდა ერთიდაიმავე რესურსის გამოყოფა, დააჩქარებდა პროცესების შესრულებას.

6.2. ურთიერთბლოკირების მოდელირება

ოპერაციულ სისტემას გარკვეული ალგორითმების საშუალებით შეუძლია გააალიზოს მოთხოვნათა მიმდევრობა. ანალიზის შედეგად (მივყვართ თუ არა მოთხოვნათა მიმდევრობას ურთიერთბლოკირებამდე) მიღებული დასკვნის შესაბამისად ის ღებულობს გადაწყვეტილებას

გამოყოს თუ არა პროცესებისთვის შესაბამისი რესურსები.

იმისთვის, რომ გავერკვეთ თუ როგორ აკეთებს ამას ოპერაციული სისტემა დაგვჭირდება ავაგოთ მოთხოვნათა მომდევრობის მოდელი გრაფთა თეორიის გამოყენებით. ყოველ გრაფს გააჩნია ორი სახის კვანძი: პროცესი, რომელიც გამოსახულია წრის ფორმით, და რესურსი, რომელიც გამოსახულია კვადრატის ფორმით. მიმართული მონაკვეთი რესურსიდან პროცესისკენ აღნიშნავს, რომ პროცესმა მოითხოვა შესაბამისი რესურსი და ის იკავებს მას (ნახ. 6.1. a)). პროცესიდან რესურსისკენ მიმართული მონაკვეთი აღნიშნავს, რომ პროცესი ელოდება შესაბამის რესურსს და ამის გამო ის ბლოკირებულია (ნახ. 6.1. b)). ნახ. 6.1. გ)-ზე კი გამოსახულია შემდეგი სიტუაცია: C პროცესი იკავებს U რესურსს და ელოდება T რესურსს, რომელსაც იკავებს D პროცესი, ხოლო D პროცესი კი პირიქით იკავებს T რესურსს და ელოდება U-ს.



ნახ. 6.1. რესურსების განაწილების გრაფი: რესურსი დაკავებულია (ა); რესურსის მოთხოვნა (ბ); ურთიერთბლოკირება (გ)

ქვემოთ ჩვენ განვიხილავთ ორ შემთხვევას, როდესაც რესურსები წარმოდგენილია მხოლოდ 1 ასლის სახით და როდესაც რესურსები წარმოდგენილია რამდენიმე ასლის სახით. თითოეული შემთხვევისასთვის განვიხილავთ შესაბამის ალგორითმს.

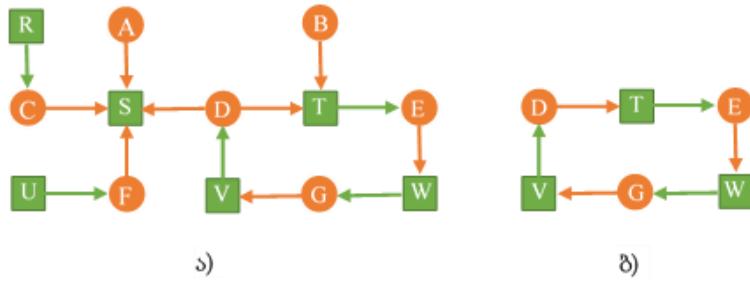
6.3. ურთიერთბლოკირების ალგორითმი

1. რესურსების ერთი ასლი. განხილვა დავიწყოთ მარტივი შემთხვევით, როდესაც სისტემაში არსებული ყოველი რესურსი წარმოდგენილია ერთი ასლის სახით. ასეთი სისტემისთვის შესაძლებელია ავაგოთ ნახ. 6.1-ზე გამოსახული გრაფის ანალოგიური გრაფი. თუ გრაფი შეიცავს ერთ ან მეტ ციკლს, მაშინ საქმე გვაქვს ურთიერთბლოკირებასთან. ციკლში მონაწილე ყოველი პროცესი ბლოკირებულია და მათი იქიდან თავის დაღწევა შეუძლებელია. თუ გრაფში არ გვაქვს ციკლი შესაბამისად არ გვაქვს ურთიერთბლოკირებაც.

განვიხილოთ მაგალითი. ვთქვათ, მოცემული გვაქვს 7 პროცესი (A - G) და 6 რესურსი (R - W). დავუშვათ რესურსებზე გაკეთებულ მოთხოვნები წარმოდგენილია შემდეგი მიმდევრობის სახით:

1. A პროცესი იკავებს R რესურსს და ითხოვს S რესურსს;
2. B პროცესი არ იკავებს არცერთ რესურსს და ითხოვს T რესურსს;
3. C პროცესი არ იკავებს არცერთ რესურსს და ითხოვს S რესურსს;
4. D პროცესი იკავებს U რესურსს და ითხოვს S და T რესურსს;
5. E პროცესი იკავებს T რესურსს და ითხოვს V რესურსს;
6. F პროცესი იკავებს W რესურსს და ითხოვს S რესურსს;
7. G პროცესი იკავებს V რესურსს და ითხოვს U რესურსს.

იმყოფება თუ არა სისტემა ურთიერთბლოკირების მდგომარეობაში და თუ პასუხი დადებითია, მაშინ რომელი რესურსები მონაწილეობენ ურთიერთბლოკირებაში?



ნახ. 6.2. რესურსების გრაფი (ა); რესურსების ციკლი (ბ)

ამ კითხვაზე პასუხს იძლევა ნახ. 6.2. ა)-ზე გამოსახული გრაფი, რომელიც აგებულია მოთხოვნების მიმდევრობის შესაბამისად. როგორც ა)-დან ვხედავთ, გრაფი შეიცავს ერთ ციკლს. ციკლში მონაწილე პროცესები (D, E, G (ნახ. ბ)) იმყოფებიან ურთიერთბლოკირების მდგომარეობაში. A, C, F პროცესები არ მონაწილეობენ ურთიერთბლოკირებაში, ვინაიდან S რესურსი შესაძლებელია (ნებისმიერი მიმდევრობით) გამოყოს თითოეულ მათგანს, სამუშაოს დასრულების შემდეგ კი დააბრუნებს მას და სხვა პროცესებსაც შეეძლებათ ამ რესურსის გამოყენება. (შევნიშნოთ, რომ ამ რესურსის დაუფლება ასევე შეუძლია D პროცესსაც, რომელიც არ გამოათავისუფლებს მას დასრულებამდე.)

მიუხედავად იმისა, რომ გრაფების ცხრილიდან ვიზუალურად ადვილია ურთიერთბლოკირების აღმოჩენა, რეალურ სისტემებში საჭიროა ფორმალური ალგორითმი. განვიხილოთ ბლოკირების აღმოჩენის ყველაზე მარტივი ალგორითმი, რომელიც ამოწმებს რესურსების გრაფს და ციკლის აღმოჩენის შემთხვევაში წყვეტს მუშაობას. თუ გრაფი არ შეიცავს ციკლს, მაშინ ალგორითმი სრულდება გრაფის ბოლომდე შემოწმების შემდეგ. ალგორითმში იგება ერთი დინამიური სტრუქტურა (L), რომელშიც იწერება კვანძების მიმდევრობა და ინიშნება მიმართული მონაკვეთები. ალგორითმის მუშაობის პროცესში მიმართული მონაკვეთის მონიშვნა აღნიშნავს, რომ შესაბამისი გადასვლა შემოწმებულია და ის არ საჭიროებს ხელმეორედ გადამოწმებას.

ალგორითმი შედგება შემდეგი ეტაპებისაგან:

1. გრაფზე არსებული ყოველი i-ური კვანძისთვის, რომელსაც გრაფი გამოიყენებს საწყის წერტილად, შესრულდება 2 - 6 ეტაპები;
2. ხორციელდება L ჩამონათვალის ინიციალიზირება და ყველა მიმართულ მონაკვეთს ეხსნება მონიშვნა;
3. მიმდინარე კვანძი L ჩამონათვალს ემატება ბოლოდან და მოწმდება, ხომ არ არის ის ამ ჩამონათვალში. ჩამონათვალში რომელიმე კვანძის მეორედ გამოჩენა ნიშნავს, რომ გრაფი შეიცავს ციკლს და ალგორითმი წყვეტს მუშაობას;
4. მითითებული კვანძისთვის მოწმდება ხომ არ არსებობს კვანძიდან გამომავალი სხვა მიმართული მონაკვეთი. ასეთის არსებობის შემთხვევაში გადავდივართ მე-5 ეტაპზე. წინააღმდეგ შემთხვევაში გადავდივართ მე-6 ეტაპზე;
5. მიმდინარე კვანძიდან მიმართული მონაკვეთის არჩევა ხდება ნებისმიერად. მიმართულ მონაკვეთს ეხსნება მონიშვნა და გადავდივართ ახალ კვანძზე ამ მიმართულებით. ალგორითმი შესრულებას აგრძელებს მე-3 ეტაპიდან;
6. თუ კვანძი წარმოადგენს საწყის წერტილს ეს ნიშნავს, რომ გრაფი არ შეიცავს ციკლს და ალგორითმი წყვეტს მუშაობას. წინააღმდეგ შემთხვევაში გვაქვს ჩიხი. მიმდინარე კვანძი იშლება და ალგორითმი უბრუნდება იმ კვანძს, რომლიდანაც მიმდინარე კვანძზე მოხდა გადასვლა. შემდეგ ალგორითმი ახალი კვანძისთვის შესრულებას იწყებს მე-3 ეტაპიდან.

ალგორითმი ყოველ კვანძს განიხილავს საწყისი წერტილის როლში, აგებს ხეს და მის შიგნით ემებს ციკლს. თუ შესრულების მომენტში ალგორითმმა ორჯერ აღმოაჩინა რომელიმე კვანძი ეს ნიშნავს, რომ ციკლი აღმოჩენილია და ალგორითმი წყვეტს შესრულებას. ყოველი კვანძისთვის ალგორითმი გადადის ყველა შესაძლო მიმართულებით, რომელსაც უთითებს მიმართული მონაკვეთი (მოთხოვნა). თუ ალგორითმს კვანძიდან რომელიმე მიმართულებით გადასავლა არ შეუძლია (გადასვლა სხვა კვანძზე არ ხდება), მაშინ ის უბრუნდება ერთი დონით მაღლა მყოფ კვანძს. თუ ალგორითმს რომელიმე კვანძთან მიმართებაში შესამოწმებელი არაფერი დარჩა ეს ნიშნავს, რომ კვანძი არ მონაწილეობს ურთიერთბლოკირებაში. თუ ყველა კვანძისთვის ეს პირობა შესრულებულია, მაშინ სისტემა არ იმყოფება ურთიერთბლოკირების მდგომარეობაში.

პრაქტიკული ხასიათის ამოცანაზე განვახორციელოთ ალგორითმის მუშაობის ილუსტრირება. განვიხილოთ ნახ. 6.2. ა)-ზე წარმოდგენილი გრაფი. ალგორითმის მიერ კვანძების განხილვის მიმდევრობა ნებისმიერია. ჩვენს შემთხვევაში კვანძები ავიღოთ შემდეგი მიმდევრობით: R, A, B, C, S, D, T, E და ა.შ. შევნიშნოთ, რომ ციკლის აღმოჩენის შემთხვევაში ალგორითმი წყვეტს მუშაობას.

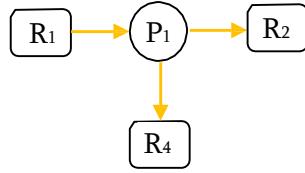
პირველი ალგორითმი საწყის წერტილად იღებს R კვანძს და ხდება L სტურქტურის ინიციალიზირება ($L=[R]$). რადგანაც R-დან გადასვლა გვაქვს C-ზე, ამიტომ ის აღმოჩნდება L-ში მეორე ელემენტად ($L=[RC]$). C-დან გადასვლა ხდება S-ზე და შესაბამისად ის იქნება L-ში მესამე ელემენტი S ($L=[RCS]$). რადგანაც S-დან არსად არ ხდება გადასვლა, ამიტომ L-ის ფორმირება R კვანძისთვის დასრულებულია და შესაბამისად ვლებულობთ, რომ R ურთიერთბლოკირებაში არ მონაწილეობს. ალგორითმი მეორე საწყის წერტილად ირჩევს C კვანძს. რადგანაც C-დან გვაქვს გადასვლა მხოლოდ S-ზე და იქიდან კი არსად, ამიტომ ამ შემთხვევაში გვაქვს L=[CS], საიდანაც ვასკვნით, რომ C-ც არ მონაწილეობს ურთიერთბლოკირებაში. შემდეგ ნაბიჯზე ალგორითმი საწყის წერტილად ირჩევს B-ს და ანალოგიური მსჯელობით D-მდე მისვლისას გვაქვს L=[BTEWGVD]. D-დან ორი მიმართულებითაა შესაძლებელი გადასვლა (S და T). S-ის მიმართულებით გადასვლის შემთხვევაში, რადგანაც S-დან არცერთი მიმართულებით არ ხდება გადასვლა, ამიტომ ვლებულობთ L=[BTEWGVDS], საიდანაც ვასკვნით, რომ ამ მიმართულებით არ შეიძლება გვქონდეს ურთიერთბლოკირება. D-ს მიმართულებით გადასვლით კი გვაქვს L=[BTEWGVDT]. რადგანაც T L-ში გამოჩნდა მეორეჯერ ეს ნიშნავს, რომ ციკლი აღმოჩენილია და ალგორითმი წყვეტს მუშაობას.

ამოცანა 1. იძლევა თუ არა პროცესების მიერ რესურსებზე გაკეთებულ მოთხოვნათა მიმდევრობა საიმედო მდგომარეობას, თუ მოთხოვნათა მიმდევრობა მოცემულია სახით:

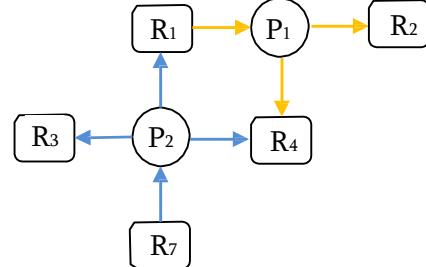
პროცესი	იკავებს	ითხოვს
P ₁	R ₁	R ₂ , R ₄
P ₂	R ₇	R ₁ , R ₃ , R ₄
P ₃		R ₂ , R ₄ , R ₅
P ₄	R ₃	R ₆ , R ₇

განსაზღვროთ მოთხოვნათა მოცემულ მიმდევრობას მივყავართ თუ არა არასაიმედო მდგომარეობამდე. დადებითი პასუხის შემთხვევაში განსაზღვრეთ ურთიერთბლოკირებაში მონაწილე რესურსები.

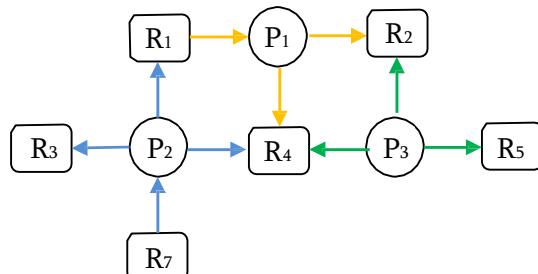
ავაგოთ მოთხოვნების შესაბამისი დიაგრამა: პირველი მოთხოვნისთვის (P_1 იკავებს R₁-ს, ითხოვს R₂ და R₄-ს) გვექნება



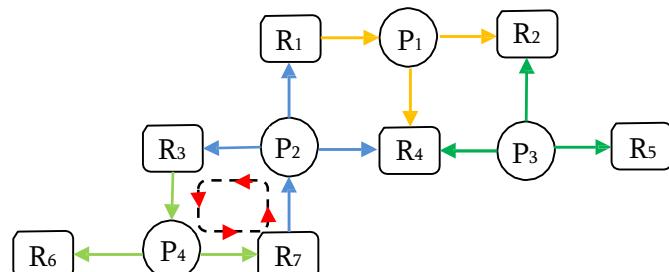
მეორე მოთხოვნისთვის (P_2 იკავებს R_7 -ს, ითხოვს R_1 , R_3 და R_4 -ს) გვექნება



მესამე მოთხოვნისთვის (P_3 ითხოვს R_2 , R_4 და R_5 -ს) გვექნება



მეოთხე, ბოლო მოთხოვნისთვის (P_4 ითხოვს R_3 და R_6 , R_7) გვექნება



რადგანაც დიაგრამის მიხედვით მიმართული მონაკვეთებით შეიკრა წრე ($P_2 \rightarrow R_3 \rightarrow P_4 \rightarrow R_7$), ამიტომ მოთხოვნათა მოყვანილი მიმდევრობა გვაძლევს რესურსების გადანაწილების არასაიმედო მდგომარეობას. ურთიერთბლოკირებაში მონაწილეობს შემდეგი რესურსები: R_3 და R_7 .

მაგალითები. იძლევა თუ არა პროცესების მიერ რესურსებზე გაკეთებულ მოთხოვნათა მიმდევრობა საიმედო მდგომარეობას, თუ მოთხოვნათა მიმდევრობა მოცემულია სახით:

პროცესი	იკავებს	ითხოვს
P_1	R_2	R_1
P_2	R_3	R_1
P_3	R_1	R_2, R_4
P_4	R_4, R_5	
P_5	R_6	R_4, R_5

პროცესი	იკავებს	ითხოვს
P_1		R_1, R_2
P_2		R_1, R_3
P_3	R_1	R_2, R_4
P_4	R_3, R_5	R_4
P_5	R_4	R_5
P_6		R_5

8)	<table border="1"> <thead> <tr> <th>პროცესი</th> <th>იკავებს</th> <th>ითხოვს</th> </tr> </thead> <tbody> <tr><td>P₁</td><td>R₃</td><td>R₁</td></tr> <tr><td>P₂</td><td>R₁</td><td>R₂, R₄</td></tr> <tr><td>P₃</td><td>R₂</td><td>R₅</td></tr> <tr><td>P₄</td><td>R₅</td><td>R₄, R₇</td></tr> <tr><td>P₅</td><td>R₆</td><td>R₃</td></tr> <tr><td>P₆</td><td>R₈</td><td>R₄, R₆, R₇</td></tr> <tr><td>P₇</td><td></td><td>R₇, R₈</td></tr> </tbody> </table>	პროცესი	იკავებს	ითხოვს	P ₁	R ₃	R ₁	P ₂	R ₁	R ₂ , R ₄	P ₃	R ₂	R ₅	P ₄	R ₅	R ₄ , R ₇	P ₅	R ₆	R ₃	P ₆	R ₈	R ₄ , R ₆ , R ₇	P ₇		R ₇ , R ₈	<table border="1"> <thead> <tr> <th>პროცესი</th> <th>იკავებს</th> <th>ითხოვს</th> </tr> </thead> <tbody> <tr><td>P₁</td><td></td><td>R₁, R₂</td></tr> <tr><td>P₂</td><td></td><td>R₁, R₃</td></tr> <tr><td>P₃</td><td></td><td>R₃, R₄</td></tr> <tr><td>P₄</td><td></td><td>R₅</td></tr> <tr><td>P₅</td><td></td><td>R₄</td></tr> <tr><td>P₆</td><td></td><td>R₄, R₆</td></tr> <tr><td>P₇</td><td></td><td>R₅, R₆</td></tr> </tbody> </table>	პროცესი	იკავებს	ითხოვს	P ₁		R ₁ , R ₂	P ₂		R ₁ , R ₃	P ₃		R ₃ , R ₄	P ₄		R ₅	P ₅		R ₄	P ₆		R ₄ , R ₆	P ₇		R ₅ , R ₆						
პროცესი	იკავებს	ითხოვს																																																						
P ₁	R ₃	R ₁																																																						
P ₂	R ₁	R ₂ , R ₄																																																						
P ₃	R ₂	R ₅																																																						
P ₄	R ₅	R ₄ , R ₇																																																						
P ₅	R ₆	R ₃																																																						
P ₆	R ₈	R ₄ , R ₆ , R ₇																																																						
P ₇		R ₇ , R ₈																																																						
პროცესი	იკავებს	ითხოვს																																																						
P ₁		R ₁ , R ₂																																																						
P ₂		R ₁ , R ₃																																																						
P ₃		R ₃ , R ₄																																																						
P ₄		R ₅																																																						
P ₅		R ₄																																																						
P ₆		R ₄ , R ₆																																																						
P ₇		R ₅ , R ₆																																																						
9)	<table border="1"> <thead> <tr> <th>პროცესი</th> <th>იკავებს</th> <th>ითხოვს</th> </tr> </thead> <tbody> <tr><td>P₁</td><td>R₁</td><td>R₂</td></tr> <tr><td>P₂</td><td>R₂</td><td>R₃</td></tr> <tr><td>P₃</td><td>R₄</td><td>R₁</td></tr> <tr><td>P₄</td><td>R₃</td><td>R₅</td></tr> <tr><td>P₅</td><td></td><td>R₃, R₅, R₇</td></tr> <tr><td>P₆</td><td>R₆</td><td>R₄</td></tr> <tr><td>P₇</td><td>R₅</td><td>R₆, R₇</td></tr> </tbody> </table>	პროცესი	იკავებს	ითხოვს	P ₁	R ₁	R ₂	P ₂	R ₂	R ₃	P ₃	R ₄	R ₁	P ₄	R ₃	R ₅	P ₅		R ₃ , R ₅ , R ₇	P ₆	R ₆	R ₄	P ₇	R ₅	R ₆ , R ₇	<table border="1"> <thead> <tr> <th>პროცესი</th> <th>იკავებს</th> <th>ითხოვს</th> </tr> </thead> <tbody> <tr><td>P₁</td><td>R₁</td><td>R₂</td></tr> <tr><td>P₂</td><td>R₄</td><td>R₁</td></tr> <tr><td>P₃</td><td>R₂</td><td>R₃</td></tr> <tr><td>P₄</td><td>R₆</td><td>R₄</td></tr> <tr><td>P₅</td><td></td><td>R₂, R₄, R₅, R₇</td></tr> <tr><td>P₆</td><td>R₃</td><td>R₅</td></tr> <tr><td>P₇</td><td>R₇</td><td>R₆</td></tr> <tr><td>P₈</td><td>R₅</td><td>R₈</td></tr> <tr><td>P₉</td><td>R₈</td><td>R₇</td></tr> </tbody> </table>	პროცესი	იკავებს	ითხოვს	P ₁	R ₁	R ₂	P ₂	R ₄	R ₁	P ₃	R ₂	R ₃	P ₄	R ₆	R ₄	P ₅		R ₂ , R ₄ , R ₅ , R ₇	P ₆	R ₃	R ₅	P ₇	R ₇	R ₆	P ₈	R ₅	R ₈	P ₉	R ₈	R ₇
პროცესი	იკავებს	ითხოვს																																																						
P ₁	R ₁	R ₂																																																						
P ₂	R ₂	R ₃																																																						
P ₃	R ₄	R ₁																																																						
P ₄	R ₃	R ₅																																																						
P ₅		R ₃ , R ₅ , R ₇																																																						
P ₆	R ₆	R ₄																																																						
P ₇	R ₅	R ₆ , R ₇																																																						
პროცესი	იკავებს	ითხოვს																																																						
P ₁	R ₁	R ₂																																																						
P ₂	R ₄	R ₁																																																						
P ₃	R ₂	R ₃																																																						
P ₄	R ₆	R ₄																																																						
P ₅		R ₂ , R ₄ , R ₅ , R ₇																																																						
P ₆	R ₃	R ₅																																																						
P ₇	R ₇	R ₆																																																						
P ₈	R ₅	R ₈																																																						
P ₉	R ₈	R ₇																																																						
3)	<table border="1"> <thead> <tr> <th>პროცესი</th> <th>იკავებს</th> <th>ითხოვს</th> </tr> </thead> <tbody> <tr><td>P₁</td><td>R₁, R₂</td><td></td></tr> <tr><td>P₂</td><td>R₃</td><td>R₁, R₂</td></tr> <tr><td>P₃</td><td>R₅</td><td>R₂, R₄</td></tr> <tr><td>P₄</td><td></td><td>R₃, R₅</td></tr> <tr><td>P₅</td><td>R₄, R₆</td><td></td></tr> <tr><td>P₆</td><td></td><td>R₅, R₆</td></tr> </tbody> </table>	პროცესი	იკავებს	ითხოვს	P ₁	R ₁ , R ₂		P ₂	R ₃	R ₁ , R ₂	P ₃	R ₅	R ₂ , R ₄	P ₄		R ₃ , R ₅	P ₅	R ₄ , R ₆		P ₆		R ₅ , R ₆																																		
პროცესი	იკავებს	ითხოვს																																																						
P ₁	R ₁ , R ₂																																																							
P ₂	R ₃	R ₁ , R ₂																																																						
P ₃	R ₅	R ₂ , R ₄																																																						
P ₄		R ₃ , R ₅																																																						
P ₅	R ₄ , R ₆																																																							
P ₆		R ₅ , R ₆																																																						

2. რესურსების მრავალი ასლი. იმ შემთხვევაში, როდესაც ოპერაციულ სისტემაში ერთიდაიგივე რესურსი წარმოდგენილია რამდენიმე ასლის სახით გამოიყენება სხვა მიღომა. ამ შემთხვევაში ი პროცეს შორის (P_1, \dots, P_n) ურთიერთბლოკირების აღმოჩენის ალგორითმში გამოიყენება მატრიცები.

შემოვიღოთ აღნიშვნები: ვთქვათ, თ არის გამოთვლით სისტემაში განსხვავებული რესურსების რაოდენობა და რესურსები გადანომრილია 1-დან m -მდე. E_j -ით ($1 \leq j \leq m$) აღვნიშნოთ j -ური რესურსის ასლების რაოდენობა. E -თი აღვნიშნოთ გამოთვლითი სისტემის რესურსების საერთო რაოდენობა (ანუ ვექტორი $A = (A_1, \dots, A_m)$). A -თი აღვნიშნოთ დროის მიმდინარე მომენტში რესურსების თავისუფალი (ხელმისაწვდომი) ასლების რაოდენობის აღმნიშვნელი ვექტორი $A = (A_1, \dots, A_m)$, სადაც A_j -ით ($1 \leq j \leq m$) აღნიშნულია j -ური თავისუფალი რესურსის რაოდენობა. დამატებით შემოვიღოთ კიდევ ორი მატრიცა R და C , სადაც R მატრიცის R_i ($1 \leq i \leq n$) სტრიქონი (ვექტორი) აღნიშნავს P_i პროცესის მიერ დამატებით მოთხოვნილი რესურსების საერთო რაოდენობას, ხოლო C მატრიცის C_i ($1 \leq i \leq n$) სტრიქონი (ვექტორი) კი აღნიშნავს P_i პროცესისთვის დროის მიმდინარე მომენტში გამოყოფილი რესურსების საერთო რაოდენობას.

$$P = (P_1, \dots, P_n) \quad A = (A_1, \dots, A_m) \quad E = (E_1, \dots, E_m)$$

$$R = \begin{pmatrix} R_{11} & \cdots & R_{1m} \\ \vdots & \cdots & \vdots \\ R_{ln} & \cdots & R_{nm} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \cdots & \vdots \\ C_{ln} & \cdots & C_{nm} \end{pmatrix}$$

ცხადია, რომ რესურსებისთვის გვექნება შემდეგი დამოკიდებულება

$$E_i = A_i + \sum_{j=1}^m C_{ij} \quad (1 \leq i \leq n)$$

ნებისმიერ ორ $X = (X_1, \dots, X_m)$ და $Y = (Y_1, \dots, Y_m)$ ვექტორს შორის შედარების ოპერაცია განვსაზღვროთ შემდეგნაირად: $X \leq Y$, მაშინ და მხოლოდ მაშინ, როდესაც $X_i \leq Y_i$ ყოველი i -სთვის $1 \leq i \leq m$.

ალგორითმის მუშაობის დაწყებამდე ყველა პროცესი ცხადდება არამარკირებულად. შესრულების პროცესში მოხდება პროცესების მარკირება იმის აღსანიშნავად, რომ ისინი არ მონაწილეობენ ურთიერთბლოკირებაში და შეუძლიათ დასრულება. ალგორითმის დასრულების შემდეგ არამარკირებული პროცესის არსებობა ნიშნავს, რომ ის მონაწილეობს ურთიერთბლოკირებაში. ალგორითმის გამოყენებისას ვითვალისწინებთ ყველაზე უარეს შემთხვევას, რომლის დროსაც პროცესი იკავებს ყველა რესურსს დასრულებამდე.

ურთიერთბლოკირების ალგორითმის მუშაობა იყოფა სამ ეტაპად:

1. ხორციელდება არამარკირებული პროცესის (P_i) ძებნა, რომლისთვისაც შესრულებულია პირობა $C_i < A_i$, ანუ დასასრულებლად საჭირო რესურსების რაოდენობა არ აღემატება თავისუფალი რესურსების რაოდენობას;
2. თუ ასეთი პროცესი მოიძებნა A ვექტორით იცვლება C მატრიცის შესაბამისი სტრიქონი და ალგორითმი უბრუნდება ეტაპ 1-ს;
3. თუ ასეთი პროცესი არ არსებობს, მაშინ ალგორითმი ასრულებს მუშაობას.

საზოგადოდ, ალგორითმის მუშაობის პრინციპი მდგომარეობს შემდეგში. პირველ ეტაპზე

ალგორითმი ეძებს პროცესებს, რომელთა დაკმაყოფილებაც შესაძლებელია სისტემაში არსებული თავისუფალი რესურსების ხარჯზე. შემდეგ პროცესს გამოყოფა რესურსები და ის იწყებს შესრულებას. პროცესი დასრულების შემდეგ ანთავისუფლებს მის მიერ დაკავებულ ყველა რესურსს და მათი გამოყენება შეუძლია სისტემაში არსებულ სხვა პროცესებს. ხდება პროცესის მარკირება. თუ ალგორითმის დასრულების შემდეგ სისტემაში აღმოვაჩნდა მხოლოდ მარკირებული პროცესები ანუ ყველა პროცესს შეუძლია დასრულება, ეს ნიშნავს, რომ სისტემაში ურთიერთბლოკირების წარმოშობის საფრთხე არ არსებობს. წინააღმდეგ შემთხვევაში დროის ნებისმიერ მომენტში შესაძლებელია წარმოიშვას ურთიერთბლოკირება.

ალგორითმის მუშაობის საილუსტრაციოდ განვიხილოთ მაგალითი.

ამოცანა 2. პროცესების მიერ რესურსებზე გაკეთებული მოთხოვნის მიხედვით გაარკვიეთ არის თუ არა მოთხოვნათა მიმდევრობა საიმედო. თუ მიმდევრობა არასაიმედოა, მაშინ მიუთითეთ რომელი რესურსები მონაწილეობენ ურთიერთბლოკირებაში.

პროცესების მიერ რესურსებზე გაკეთებული მოთხოვნების საფუძველზე დიაგრამის აგების მეთოდი სამართლიანი იმ შემთხვევაში, როდესაც რესურსები სისტემაში წარმოდგენილია ერთი ეგზემპლარის სახით. თუ სისტემაში ყოველი რესურსები წარმოდგენილია რამდენიმე ეგზემპლარის სახით, მაშინ მოთხოვნათა საიმედოობის გასარკვევად საჭიროა სხვა მეთოდი ძიება.

განვიხილოთ მაგალითი. ვთქვათ, სისტემაში მოცემული გვაქვს 4 პროცესი, $P = (P_1 P_2 P_3 P_4)$ და 4 რესურსი, $E = (6, 8, 5, 8)$, მათ შორის მიმდინარე მომენტისთვის თავისუფალი რესურსები გვაქვს შემდეგი რაოდენობით $A = (2, 2, 0, 0)$. დავუშვათ პროცესებისთვის მიმდინარე მომენტში გამოყოფილი რესურსების მატრიცას და მოთხოვნილი რესურსების მატრიცას შესაბამისად აქვს სახე

$$R = \begin{pmatrix} 1 & 2 & 1 & 3 \\ 3 & 2 & 2 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 2 \end{pmatrix} \quad \text{და} \quad C = \begin{pmatrix} 1 & 6 & 2 & 1 \\ 1 & 0 & 2 & 1 \\ 2 & 2 & 0 & 0 \\ 0 & 2 & 1 & 0 \end{pmatrix}.$$

შევნიშნოთ, რომ გამოყოფილი და მოთხოვნილი რესურსების მატრიცაში სვეტი აღნიშნავს ნუმერაციის შესაბამის რესურსს და თუ რა რაოდენობითაა ის პროცესისთვის გამოყოფილი ან მის მიერ მოთხოვნილი, ხოლო სტრიქონი აღნიშნავს ნუმერაციის შესაბამის პროცესს და თუ რა რაოდენობითაა მისთვის სხვადასხვა რესურსები გამოყოფილი ან მოთხოვნილი. მაგალითად, მეორე სვეტი აღნიშნავს A_2 რესურსს, ხოლო მეოთხე სტრიქონი კი P_4 პროცესს.

თავისუფალი რესურსების მიხედვით მხოლოდ P_3 პროცესის მოთხოვნის დაკმაყოფილებაა შესაძლებელი. მისი დასრულების შემდეგ თავისუფალი რესურსების რაოდენობა i -ება $A = (2 \ 3 \ 1 \ 0)$. რესურსების ამ რაოდენობით მხოლოდ P_4 პროცესის მოთხოვნის დაკმაყოფილებას შევძლებთ, რომლის დასრულების შემდეგ თავისუფალი რესურსების რაოდენობა i -ება $A = (2 \ 4 \ 2 \ 2)$. დარჩენილი ორი პროცესიდან თავიდან შესაძლებელი იქნება P_2 პროცესის მოთხოვნის დაკმაყოფილება ხოლო შემდეგ კი P_1 პროცესის მიერ გაკეთებული მოთხოვნის დაკმაყოფილება. მაშასადამე, მოთხოვნათა მოცემული მიმდევრობა იძლევა საიმედო მდგომარეობას, ხოლო თანმიმდევრობა რომელიც გვაძლევს საიმედო მდგომარეობას არის შემდეგი: $P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_1$.

თუ პროცესების მიერ რესურსების მოთხოვნათა მატრიცაში P_2 პროცესს მოვთხოვთ A_3

რესურსი მოითხოვოს 3 ერთეული, მაშინ მივიღებთ არასაიმედო მდგომარეობას.

მაგალითები. პროცესების მიერ რესურსებზე გაკეთებული მოთხოვნების მიხედვით განსაზღვრეთ სისტემა რჩება თუ არა საიმედო მდგომარეობაში. დადებითი პასუხის შემთხვევაში განსაზღვრეთ მოთხოვნათა დაკმაყოფილების თანმიმდევრობა.

$$s) \quad A = \begin{pmatrix} 1 & 2 & 0 & 2 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 2 & 3 & 2 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 \\ 3 & 4 & 1 & 1 & 3 \\ 0 & 2 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 2 & 3 & 2 & 0 & 0 \\ 0 & 1 & 2 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 2 & 1 & 0 \end{pmatrix}$$

$$b) \quad A = \begin{pmatrix} 2 & 1 & 3 & 0 & 2 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 0 & 2 & 0 & 1 \\ 0 & 3 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 0 \\ 0 & 2 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 0 & 1 & 2 & 0 & 0 \\ 0 & 1 & 2 & 1 & 1 \\ 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 1 & 0 \end{pmatrix}$$

$$g) \quad A = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 2 & 2 & 0 & 0 & 2 & 2 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 2 & 1 & 1 & 0 & 0 \\ 2 & 0 & 2 & 5 & 0 & 2 \end{pmatrix} \quad R = \begin{pmatrix} 2 & 2 & 3 & 0 & 0 & 2 \\ 1 & 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 2 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 2 & 3 & 3 \end{pmatrix}$$

$$q) \quad A = \begin{pmatrix} 1 & 1 & 1 & 2 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 2 & 1 & 3 & 1 & 0 \\ 2 & 1 & 0 & 0 & 2 & 3 \\ 1 & 2 & 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 0 & 1 & 0 \\ 2 & 0 & 2 & 2 & 0 & 1 \end{pmatrix} \quad R = \begin{pmatrix} 2 & 2 & 3 & 0 & 3 & 3 \\ 0 & 0 & 2 & 3 & 0 & 2 \\ 5 & 4 & 2 & 0 & 2 & 1 \\ 4 & 1 & 0 & 2 & 2 & 3 \\ 1 & 0 & 1 & 2 & 0 & 0 \end{pmatrix}$$

$$j) \quad A = \begin{pmatrix} 1 & 2 & 1 & 0 & 1 & 2 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 2 & 1 & 3 & 1 & 0 & 2 & 4 \\ 2 & 2 & 0 & 2 & 3 & 0 & 0 \\ 0 & 3 & 2 & 3 & 0 & 3 & 2 \\ 2 & 0 & 1 & 0 & 2 & 1 & 0 \\ 1 & 1 & 2 & 2 & 3 & 0 & 1 \\ 0 & 1 & 2 & 1 & 0 & 2 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 0 & 2 & 0 & 0 & 1 & 2 & 0 \\ 5 & 3 & 9 & 7 & 5 & 4 & 8 \\ 3 & 2 & 0 & 1 & 0 & 3 & 3 \\ 0 & 0 & 9 & 4 & 3 & 8 & 5 \\ 2 & 2 & 5 & 3 & 1 & 3 & 3 \\ 2 & 2 & 0 & 5 & 3 & 6 & 5 \end{pmatrix}$$

$$3) \quad A = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 2 & 1 & 0 \\ 1 & 1 & 2 & 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 2 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 & 0 & 2 & 1 \end{pmatrix} \quad R = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 2 & 2 & 0 \\ 2 & 1 & 1 & 3 & 0 & 3 & 2 \\ 4 & 2 & 2 & 0 & 2 & 3 & 2 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 3 & 1 & 0 & 2 \end{pmatrix}$$

სემინარი 7. ნაკადი (thread) UNIX-მსგავს ოპერაციულ სისტემებში

პირველი გამოთვლითი მანქანები წარმოადგენდნენ ერთპროგრამულ გამოთვლით სისტემებს, რომლებიც ვერ უზრუნველყოფდნენ რესურსების გადანაწილებას პროგრამებს შორის და პროგრამები სრულდებოდნენ მათი შემოსვლის მიმდევრობით. ასეთ სისტემებში გამოთვლითი სისტემის წარმადობის ამაღლების და რესურსების თანაბარი დატვირთვის მიმართულებით პირველი წინგადადგმული ნაბიჯი იყო პროცესების მოდელის (აბსტრაქციის) შემოღება. პროცესების მოდელის შემოღებით შესაძლებელი გახდა ერთპროგრამული გამოთვლითი სისტემის გადაქცევა მრავალპროგრამულ სისტემად, რამაც საგრძნობლად გაზარდა გამოთვლითი სისტემის წარმადობა. თანამედროვე გამოთვლითი სისტემები აღჭურვილნი არიან საკმარისად სწრაფი პროცესორებით, რომლებიც მაღალი სისწრაფით ასრულებენ გამოთვლებს, და მრავალი ახალი რესურსით, რომლებიც დამატებით შესაძლებლობებს მატებენ გამოთვლით სისტემას. მიუხედავად პროცესორის გაზრდილი წარმადობისა და რესურსების გაზრდილი რაოდენობისა პროცესების მოდელი მაინც არ იძლევა ამ უპირატესობის სრულად გამოყენების შესაძლებლობას. პროცესები იკავებენ სისტემის რესურსებს მონოპოლურად და მათ არ უზიარებენ სისტემაში არსებულ სხვა პროცესებს, მიუხედავად იმისა იყენებს თუ არა მათ. პროცესების მოდელის შემდგომ განზოგადებას, რომელმაც შესაძლებელი გახადა თანამედროვე გამოთვლით სისტემებში ჩადებული უპირატესობის გამოყენება, წარმოადგენს ნაკადების (thread-ების) მოდელი.

thread-ის მოდელის შემოღებით შესაძლებელი გახდა პროცესის დაყოფა “ურთიერთ-დამოუკიდებელი” ცნებით დაკავშირებულ კომპონენტებად (thread-ად) - მინი პროცესებად, რომლებიც ერთი მიზნისთვის არიან შექმნილი და იძლევიან პროგრამის შესრულების დაჩქარების საშუალებას. პროცესებისგან განსხვავებით thread-ები არ საჭიროებენ კონტექსტის გადართვას და იძლევიან პროცესის მიერ დაკავებული რესურსების "მეტნაკლებად" თანაბარი ან სრულად დატვირთვის შესაძლებლობას.

სხვადასხვა ოპერაციულ სისტემაში ნაკადები შეიძლება იყოს რეალიზებული განსხვავებული მექანიზმებით. ოპერაციულ სისტემაში ნაკადი შეიძლება რეალიზებული იყოს არსებული სამი მოდელიდან ერთერთის გამოყენებით: ნაკადები მომხმარებლის რეჟიმში, ნაკადები ბირთვის რეჟიმში და ჰიბრიდული მოდელი (მომხმარებლის და ბირთვის რეჟიმში ერთდროული რეალიზაცია).

POSIX სტანდარტი განსაზღვრავს UNIX-მსგავს სისტემებში ნაკადებთან მუშაობის ინტერფეისს. ჩვენ მოკლედ გავეცნობით ზოგიერთ ფუნქციებს, რომლებიც იძლევიან პროცესის thread-ებად დაყოფის და მათი ყოფაქცევის მართვის შესაძლებლობას POSIX სტანდარტის შესაბამისად. POSIX სტანდარტის დამაკმაყოფილებელ thread-ებს უწოდებენ **POSIXthread-ს** ან **pthread-ს**.

ნაკადების მოდელის მიხედვით ყოველი პროცესი შეიცავს მინიმუმ ერთ ნაკადს. თუ პროცესის დაყოფა შეუძლებელია ერთზე მეტ ნაკადად, მაშინ მას ტრადიციული პროცესი ეწოდება. ამ შემთხვევაში ორივე ცნება „პროცესი“ და „ნაკადი“ იდენტურია. რადგანაც სისტემაში არსებულ ყოველ პროცესს გააჩნია იდენტიფიკატორი, ამიტომ პროცესის მსგავსად ნაკადი იდენტიფირებისთვის საჭიროებს უნიკალურ ნომერს - thread-ის იდენტიფიკატორი. რადგანაც ნაკადი იქმნება პროცესის შიგნით, ამიტომ მისი იდენტიფიკატორის მნიშვნელობა უნდა იყოს უნიკალური პროცესის შიგნით. ნაკადის იდენტიფიკატორის მნიშვნელობის მისაღებად გამოიყენება ფუნქცია `pthread_self()`. პროცესის შიგნით წარმოქმნილ პირველ `thread-ს`, ამ

პროცესის მმართველი (ძირითადი) **thread**-ი ეწოდება.

thread_self() ფუნქციის პროტოტიპი

```
#include <pthread.h>
pthread_t pthread_self(void);
```

ფუნქციის აღწერა

pthread_self ფუნქცია აბრუნებს მიმდინარე thread-ის იდენტიფიკატორის მნიშვნელობას.

მონაცემთა ტიპი pthread_t წარმოადგენს C ენის მთელმნიშვნელობიანი ტიპის ერთერთ სინონიმს

7.1. thread-ის წარმოქმნა, მიერთება და დასრულება

როგორც ზემოთ აღვნიშნეთ, UNIX-ში ტრადიციული პროცესი შეიცავს მხოლოდ ერთ ნაკადს. POSIX სტანდარტის მხარდაჭერის შემთხვევაში პროგრამა ამუშავდება, როგორც პროცესი ერთი მმართველი thread-ით. ასეთი პროგრამის შესრულება არაფრით არ განსხვავდება ტრადიციული პროცესის შესრულებისგან, სანამ ის არ წარმოქმნის ახალ thread-ებს. ახალი thread-ის წარმოსაქმნელად გამოიყენება ფუნქცია pthread_create. წარმოქმნილ ნაკადს გააჩნია წვდომა პროცესის მისამართების სივრცეზე და რესურსებზე, რომელსაც ის მემკვიდრეობით იღებს გამომძახებელი thread-ისგან.

pthread_create ფუნქციის პროტოტიპი

```
#include <pthread.h>
int pthread_create(pthread_t *thr, pthread_attr_t *attr, void *(*start_routine)(void ), void *arg);
```

ფუნქციის აღწერა

pthread_create ფუნქცია გამოიყენება მიმდინარე პროცესის შიგნით ახალი thread-ის წარმოსაქმნელად.

thr - პარამეტრი უთითებს მეხსიერებაში გამოყოფილ იმ მისამართს, სადაც pthread_create ფუნქციის წარმატებით დასრულების შემთხვევაში მოხდება წარმოქმნილი thread-ის იდენტიფიკატორის მნიშვნელობისგანთავსება;

attr - პარამეტრის მეშვეობით წარმოქმნილი thread-ისთვის მოხდება გარკვეული ატრიბუტების მინიჭება. თუ ამ პარამეტრად ავიდებთ მნიშვნელობას NULL, მაშინ წარმოქმნილ thread-ს სისტემის მიერ ატრიბუტები დაენიშნება გაჩუმების წესით;

მესამე პარამეტრის მეშვეობით ხორციელდება წარმოქმნილი thread-ისთვის შესასრულებელი ინსტრუქციების გადაცემა, ანუ ინსტრუქციები, რომელთა შესასრულებლადაც შეიქმნა thread-ი;

arg - პარამეტრის მეშვეობით საჭიროების შემთხვევაში ხორციელდება thread-ისთვის გარკვეული ინფორმაციის გადაცემა.

დასაბრუნებელი მნიშვნელობა

ნორმალური დასრულების შემთხვევაში ფუნქცია აბრუნებს მნიშვნელობას 0-ს და ახალი thread-ის იდენტიფიკატორს ანთავსებს იმ მისამართზე, რომელსაც უთითებს thr პარამეტრი. შეცდომის შემთხვევაში აბრუნებს დადებით მნიშვნელობას, რომელიც განსაზღვრავს <errno.h> ფაილში აღწერილ შეცდომის კოდს.

პროცესის შიგნით რამდენიმე thread-ის წარმოქმნისას წინასწარ შეუძლებელია იმის განსაზღვრა, თუ რომელი thread-ი შესრულდება პირველი.

thread-ის შექმნის შემდეგ საჭიროა ის მიუერთდეს მმართველ thread-ს (main-ს). მიერთების გარეშე ის ვერ გასწევს პროცესისთვის სასარგებლო საქმანობას. thread-ის მიერთებისთვის გამოიყენება pthread_join ფუნქცია.

pthread_join ფუნქციის პროტოტიპი

```
#include <pthread.h>
int pthread_join(pthread_t thr, void **status_addr);
```

ფუნქციის აღწერა

pthread_join ფუნქცია ახდენს გამომძახებელი thread-ის ბლოკირებას გამოძახებული thread-ის (იდენტიფიკატორით thr) დასრულებამდე. ბლოკირების მოხსნის შემდეგ status_addr მისამართზე განთავსებულ მიმთითებელშიჩაიწერება დასრულებული thread-ის მიერ დაბრუნებული მნიშვნელობა. თუ არ გვაინტერესებს რა მნიშვნელობა დააბრუნა thread-მა, მაშინ ამ პარამეტრის როლში შესაძლებელია NULL მნიშვნელობის გამოყენება.

დასაბრუნებელი მნიშვნელობა

ფუნქცია აბრუნებს მნიშვნელობას 0 ნორმალურად დასრულების შემთხვევაში. შეცდომის შემთხვევაში ბრუნდება დადებითი მნიშვნელობა (და არა უარყოფითი, როგორც უმეტეს სისტემურ გამოძახებებში და ფუნქციებში), რომელიც განსაზღვრავს <errno.h> ფაილში აღწერილ შეცდომის კოდს.

პროცესის შიგნით არსებული thread-ებიდან რომელიმეს მიერ exit ფუნქცია გამოძახებას მივყავართ მთლიანი პროცესის დასრულებამდე. thread-ის დასრულება ისე, რომ პროცესი არ დასრულდეს შესაძლებელია შემდეგნაირად:

- thread-მა შეიძლება დააბრუნოს მართვა ამუშავებული პროცედურიდან (return ოპერატორით). ამ პროცედურის დასაბრუნებელი მნიშვნელობა იქნება thread-ის დასრულების კოდი;
- thread-ი შესაძლებელია იძულებით დაასრულოს იმავე პროცესის სხვა thread-მა;
- thread-მა შეიძლება გამოიძახოს ფუნქცია pthread_exit().

pthread_exitფუნქციისპროტიპი

```
#include <pthread.h>
void pthread_exit(void *status);
```

ფუნქციის აღწერა

pthread_exit ფუნქცია გამოიყენება მიმდინარე პროცესის thread-ის დასრულებისთვის. ფუნქცია არასოდეს არ ბრუნდება მის გამომძახებელ thread-ში.

status პარამეტრში ჩაწერილი მნიშვნელობა შეიძლება გამოყენებული იყოს სხვა thread-ის მიერ, რომელიც ელოდებოდა thread-ის დასრულებას.

thread-ების მუშაობის საილუსტრაციოდ განვიხილოთ მაგალითები.

მაგალითი 1. დავწეროთ პროგრამა, რომელშიც შეიქმნება კლავიატურიდან შეტანილი მნიშვნელობის შესაბამისი რაოდენობის thread-ი, რომელთაგან თითოეული დაითვლის გადაცემული პარამეტრის შესაბამის ფაქტორიალს და დაბეჭდავს.

```
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
using namespace std;
// დავწეროთ ფუნქცია, რომელსაც შეასრულებს thread-ი
void *FACT(void* arg){
    int el = (int) arg;
    int fact = 1;
    for (int i = 2; i <= el; i++)
        fact *= i;
    cout << el << "-ის ფაქტორიალი არის - " << fact << endl;
    pthread_exit(NULL);
}
int main(){
```

```

int N, err, i;
cout << "შემოიტანეთ thread-ების რაოდენობა ";
cin >> N;
pthread_t mass[N]; // მასივი thread-ის იდენტიფიკატორისთვის
for (i = 0; i<N; i++){
    err = pthread_create(&mass[i], NULL, FACT, (void*)i); // შევქმნათ thread-ი
    if (err) { // შევამოწმოთ შეიქმნა თუ არ thread-ი
        cout << "შეცდომა! thread " << i << "-ი არ შეიქმნა." << endl;
        exit(EXIT_FAILURE);
    }
}
for (i = 0; i<N; i++){ // მივაერთოთ thread-ი ძირითად thread-ს
    err = pthread_join(mass[i], NULL);
    if (err){ // შევამოწმოთ მოხდა თუ არ thread-ის მიერთება
        cout << "შეცდომა! thread " << i << "-ი არ მიერთდა." << endl;
        exit(EXIT_FAILURE);
    }
}
exit(EXIT_SUCCESS);
}

```

prog-7.1.cpp

რადგანაც დასაკომპილირებელი პროგრამა შეიცავს thread-ებს, ამიტომ მისი კომპილირებისთვის საჭიროა დამატებით -pthread ოფციის გამოყენება, ანუ კომპილაციისთვის ტერმინალში ბრძანება უნდა აიკრიფოს ფორმით

g++ -pthread prog-7.1.cpp

(-o prog-7.1.out - სტანდარტული სახელის შეცვლის შემთხვევაში).

მაგალითი 2. დავწეროთ პროგრამა, რომელშიც thread-ი დაითვლის გადაცემული მასივის ელემენტების ჯამის საშუალო არითმეტიკულს და დაბეჭდავს. მასივი შეავსეთ შემთხვევითი ელემენტებით შუალედიდან [15, 61].

```

#include <ctime>
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
using namespace std;
const int N = 10; // მასივის ელემენტების რაოდენობა
// დავწეროთ thread-ის მიერ შესასრულებელი ფუნქცია
void* average(void* arg){
    int *array = (int*) arg;
    double sum = 0;
    for (int i = 0; i<N; i++)
        sum += array[i];
    cout << sum / N << endl;
    pthread_exit(NULL);
}

```

```

int main(){
    int m[N], err;
    pthread_t thid;
    srand(time(NULL));
    for (int i = 0; i < N; i++){ // შევასვოთ მასივი
        m[i] = rand() % 47 + 15;
    }
    cout << endl;
    err = pthread_create(&thid, NULL, average, (void*)&m);
    if (err){
        cout << "thread-ი არ შეიქმნა\n";
        exit(EXIT_FAILURE);
    }
    err = pthread_join(thid, NULL);
    if (err){
        cout << "thread-ი არ მიერთდა\n";
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}

```

prog-7.2.cpp

7.2. sem_post() და sem_wait() ფუნქცია

როგორც უკვე აღვნიშნეთ ნაკადები პროცესის შიგნით შეიძლება რამდენიმე thread-ის შექმნა. თითოეულ მათგანს შეუძლია გამოიყენოს პროცესისთვის გამოყოფილი რესურსები (ფაილები, მეხსიერება და ა.შ.). თუ პროცესის შიგნით შექმნილი ორი ან რამდენიმე thread-ი ერთმანეთთან ურთიერთქმედებისას იყენებენ მეხსიერების საერთო სივრცეს, მაშინ აღნიშნულ სივრცეზე მიმართვა უნდა ხორციელდებოდეს შეთანხმებულად. ურთიერთქმედი ნაკადების სინქრონიზაციისთვის შესაძლებელია გამოვიყენოთ ფუნქციები sem_post() და sem_wait() .

sem_post() და sem_wait() ფუნქციის პროტოტიპი

```
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

ფუნქციის აღწერა

sem_post ფუნქცია ითვლის აქტივაციათა რაოდენობას და მთვლელის მნიშვნელობას ზრდის 1- ით, ხოლო sem_wait ფუნქცია კი პირიქით, თუ აქტივაციათა რაოდენობა ერთზე მეტია, მაშინ მეხსიერების საერთო ნაწილის გამოთავისუფლების შემთხვევაში thread-ს გამოუყოფს მას და აქტივაციათა რაოდენობას ამცირებს 1-ით.

დასაბრუნებელი მნიშვნელობა

ორივე ფუნქცია აბრუნებს 0-ის ტოლ მნიშვნელობას წარმატებულად დასრულების შემთხვევაში. წარუმატებლობის შემთხვევაში აბრუნებს -1-ის ტოლ მნიშვნელობას და errno ცვლადში იწერება შეცდომის აღმნიშვნელი კოდის მნიშვნელობა.

მაგალითი 3 (მომხმარებელი მწარმოებლის ამოცანა). მომხმარებელი მწარმოებლის ამოცანა მდგომარეობს შემდეგში: ორი ურთიერთქმედი thread-ი (მომხმარებელი და მწარმოებელი) იყენებს მეხსიერების ერთიდაიმავე სივრცეს. თუ მეხსიერების სივრცე ცარიელია მწარმოებელი

მასში წერს ინფორმაციას, ხოლო თუ გადავსებულია, მაშინ ელოდება მასში ადგილის გამოთავისუფლებას. მომხმარებელი კი პირიქით, თუ მეხსიერების სივრცე ცარიელია, მაშინ ელოდება მასში ინფორმაციის გამოჩენას. ამოცანის პროგრამულ გადაწყვეტას აქვს შემდეგი სახე

```
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
// მეხსიერების სივრცე, რომელსაც მიმართავს ორივე ნაკადი
// შესაბამისად მონაცემების ჩასაწერად და წასაკითხად
const int N =1024;
char buf[N];
sem_t len; // სინქრონიზაციის ცვლადი
void * read1(void * a){ // ფუნქცია პირველი thread-ისთვის
    char * b = (char *) a;
    cout << "შემოიტანეთ სტრიქონი: " << endl;
    for (int i=0; i<N; i++) {
        sem_post(&len); // ფუნქცია ზრდის აქტივაციათა რაოდენობას
        cin >> b;
    }
    pthread_exit(NULL);
}
void * write1(void * a){ // ფუნქცია მეორე thread-ისთვის
    char * b = (char *) a;
    for (int i=0; i<N; i++) {
        sem_wait(&len); // ფუნქცია ამცირებს აქტივაციათა რაოდენობას
        cout << b;
    }
    pthread_exit(NULL);
}
int main(){
    int err1, err2;
    pthread_t th1, th2;
    err1 = pthread_create(&th1,NULL,read1, (void *)&buf); // გევემნათ #1 thread-ი
    if (err1){
        cout << "შეცდომა! შეცდომის კოდი " << err1 << endl;
        exit(EXIT_FAILURE);
    }
    err2 = pthread_create(&th2,NULL,write1, (void *)&buf); // გევემნათ #2 thread-ი
    if (err2){
        cout << "შეცდომა! შეცდომის კოდი " << err2 << endl;
        exit(EXIT_FAILURE);
    }
    err1 = pthread_join(th1,NULL); // მივაერთოთ #1 thread-ი ძირითადს
    if (err1){
        cout << "შეცდომა! შეცდომის კოდი " << err1 << endl;
        exit(EXIT_FAILURE); }
```

```

err2 = pthread_join(th2,NULL); // მივაერთოთ #2 thread-ი ძირითადს
if (err2){
    cout << "შეცდომა! შეცდომის კოდი " << err2 << endl;
    exit(EXIT_FAILURE);
}
return 0;
}

```

prog-7.3.cpp

მაგალითი 4. დაწერეთ პროგრამა, რომელშიც გვეყოლება ორი შვილი პროცესი. პირველი შვილი პროცესი შექმნის ერთ thread-ს, რომელიც განაწილებად მეხსიერებაში განათავსებს N (=11) შემთხვევით მთელ მნიშვნელობას შუალედიდან [17, 53]. მეორე პროცესი ორ thread-ს, რომელთაგან ერთი დაითვლის წაკითხული მასივის ლუწინდექსიანი ელემენტების ჯამს, ხოლო მეორე კი მასივის 3-ის ჯერადი ელემენტების საშუალო არითმეტიკულს და დაბეჭდავს.

```

#include <iostream>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <pthread.h>
#include <unistd.h>
using namespace std;
const int N = 10; // მასივში ელემენტების რაოდენობა
void* fill(void* arg);
void* sumEv(void* arg);
void* avOdd(void* arg);
int main(){
    int *array, err1, err2, st1, st2, shmid;
    pid_t p1, p2;
    pthread_t th1, th2, th3;
    key_t key;
    if ((key = ftok("th5.cpp", 0)) == -1) {
        cout << "არ შეიქმნა გასაღები.\n";
        exit(EXIT_FAILURE);
    }
    shmid = shmget(key, 44, IPC_CREAT | 0600);
    if (shmid == -1) {
        cout << "არ შეიქმნა განაწილებადი მეხსიერება.\n";
        exit(EXIT_FAILURE);
    }
    if ((p1 = fork()) == -1){
        cout << "არ შეიქმნა პირველი შვილი პროცესი.\n";
        exit(EXIT_FAILURE);
    }
    if (p1 == 0){ // პირველი შვილი პროცესის დასაწყისი

```

```

// პროცესის მისამართების სივცეს მივაბათ განაწილებადი მეხსიერება
array = (int*)shmat(shmid, NULL, 0);
if (array == (int*)(-1)) {
    cout << "არ მოხდა განაწილებადი მეხსიერების მიერთება.\n";
    exit(EXIT_FAILURE);
}

// შევქმნათ ნაკადი 1
err1 = pthread_create(&th1, NULL, fill, (void*)&array);
if (err1) {
    cout << "არ მოხდა 1 thread-ის შექმნა.\n";
    exit(EXIT_FAILURE);
}

// ძირითად thread-ს მივუერთოთ thread-o 1
err1 = pthread_join(th1, NULL);
if (err1) {
    cout << "არ მიერთდა 1 thread-o.\n";
    exit(EXIT_FAILURE);
}

// პროცესის მისამართების სივრციდან ამოვშალოთ განაწილებადი
მეხსიერება
if (shmdt(array) == -1)
    cout << "1არ მოხდა განაწილებადი მეხსიერების წაშლა.\n";
}// პირველი შვილი პროცესის დასასრული
else {
    wait(&st1);
    if ((p2 = fork()) == -1){
        cout << "არ შეიქმნა მეორე შვილი პროცესი.\n";
        exit(EXIT_FAILURE);
    }
    if (p2 == 0){ // მეორე შვილი პროცესის დასაწყისი
        sleep(2);
        // პროცესის მისამართების სივცეს მივაბათ განაწილებადი მეხსიერება
        array = (int*)shmat(shmid, NULL, 0);
        if (array == (int*)(-1)) {
            cout << "არ მოხდა განაწილებადი მეხსიერების მიერთება.\n";
            exit(EXIT_FAILURE);
        }
        // შევქმნათ ნაკადი 2
        err1 = pthread_create(&th2, NULL, sumEv, (void*)&array);
        if (err1) {
            cout << "არ მოხდა 2 thread-ის შექმნა.\n";
            exit(EXIT_FAILURE);
        }
        // შევქმნათ ნაკადი 3
        err2 = pthread_create(&th3, NULL, avOdd, (void*)&array);
        if (err2) {

```

```

cout << "არ მოხდა 3 thread-ის შექმნა.\n";
exit(EXIT_FAILURE);
}
// მირითად thread-ს მივუერთოთ thread-ი 2
err1 = pthread_join(th2, NULL);
if (err1) {
    cout << "არ მიერთდა 2 thread-ი.\n";
    exit(EXIT_FAILURE);
}
// მირითად thread-ს მივუერთოთ thread-ი 3
err2 = pthread_join(th3, NULL);
if (err2) {
    cout << "არ მიერთდა 3 thread-ი.\n";
    exit(EXIT_FAILURE);
}
// პროცესის მისამართების სივრციდან ამოვშალოთ განაწილებადი
მებსიერება
if (shmdt(array) == -1)
    cout << "არ მოხდა განაწილებადი მებსიერების წაშლა.\n";
} // მეორე შვილი პროცესის დასასრული
else { // მშობელი პროცესი
    wait(&st2);
    cout << "პროგრამა წარმატებით დასრულდა.\n";
}
}
exit(EXIT_SUCCESS);
}

// ფუნქცია განაწილებადი მებსიერების შესავსებად
void* fill(void* arg){
    int *m = (int*)arg;
    for (int i = 0; i<N; i++){
        m[i] = rand() % 37 + 17;
        cout << m[i] << " ";
    }
    cout << endl;
    pthread_exit(NULL);
}

// ფუნქცია მასივის ლუწინდექსიანი ელემენტების დაჯამებისთვის
void* sumEv(void* arg){
    int* m = (int*)arg;
    int sum = 0;
    for (int i = 0; i<N; i += 2){
        cout << m[i] << " ";
        sum += m[i];
    }
}

```

```

cout << endl;
cout << "ღუნქცია მასივის 3-ის ჯერადი ელემენტების ჯამია " << sum << endl;
pthread_exit(NULL);
}

// ფუნქცია მასივის 3-ის ჯერადი ელემენტების საშუალო
// არითმეტიკულის დაათვლელად
void* avOdd(void* arg){
    int* m = (int*)arg;
    int count = 0;
    double sum = 0;
    for (int i = 0; i<N; i++){
        cout << m[i] << " ";
        sum += m[i];
        if (m[i] % 3 == 0)
            count++;
    }
    cout << endl;
    cout << "3-ის ჯერადი ელემენტების საშუალო არის " << sum << endl;
    pthread_exit(NULL);
}

```

7.3. mutex

thread-ების სინქრონული მუშაობის უზრუნველსაყოფად semaphore-ების მსგავსად გამოიყენება mutex-ი. thread-ების სინქრონული მუშაობის უზრუნველყოფის აუცილებლობა დგება იმ შემთხვევაში, როდესაც ორი ან რამდენიმე thread-ი საჭიროებს მეხსიერების საერთო ნაწილზე წვდომას (საკუთარ კრიტიკულ სექციაში შესვლას). კრიტიკულ სექციაში thread-ის შესვლის შემდეგ საჭიროა მოხდეს კრიტიკული სექციის ჩაკეტვა (შეიძლება მეხსიერების საერთო ნაწილზე წვდომა). ამ მიზნით გამოიყენება სპეციალური ფუნქცია pthread_mutex_lock. მას შემდეგ რაც thread-ი კრიტიკულ სექციაში დაასრულებს მუშაობას საჭიროა კრიტიკული სექციის გახსნა სხვა მსურველთათვის (მოიხსნას შეზღუდვა საერთო ნაწილზე წვდომაზე). ამ მიზნით გამოიყენება სპეციალური ფუნქცია pthread_mutex_unlock. სინქრონიზაციის მიზნით ორივე ფუნქცია იყენებს სპეციალურ (pthread_mutex_t ტიპის) გლობალურ ცვლადს. ასეთი ცვლადი აუცილებლად უნდა იყოს ინიციალიზირებული. ცვლადის ინიციალიზირება შესაძლებელია ორი მეთოდით:

- სტატიკური - ცვლადის ინიციალიზირებისთვის გამოიყენება სპეციალური მუდმივა PTHREAD_MUTEX_INITIALIZER ;
- დინამიური - ცვლადის ინიციალიზირებისთვის გამოიყენება სპეციალური ფუნქცია pthread_mutex_init. ამ შემთხვევაში აუცილებელია ინიციალიზირების მომენტის შემოწმება, ვინაიდან pthread_mutex_init ფუნქციის წარუმატებლად დასრულების შემთხვევაში არაინიციალიზირებული ცვლადის გამოყენება მიგვიყვანს შეცდომამდე. mutex-თან მუშაობის დასრულების შემდეგ საჭიროა შესაბამის ცვლადს მოეხსნას ინიციალიზირება. ამ მიზნით გამოიყენება ფუნქცია pthread_mutex_destroy.

mutex-ის გამოყენების საილუსტრაციოდ განვიხილოთ მაგალითი.

მაგალითი 5. დაწერეთ პროგრამა, რომელშიც გვეყოლება ორი thread-ი. პირველი thread-ი შეავსებს მასივს N (=10) შემთხვევით მთელი მნიშვნელობით შუალედიდან [17, 66]. მეორე thread-

ი დაითვლის მასივის ელემენტების საშუალო არითმეტიკულს და დაბეჭდავს.

```
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
using namespace std;
const int N = 10;
pthread_mutex_t lock;
// ფუნქცია პირველი thread-ისთვის
void* fill(void* arg){
    // მოვახდინოთ კრიტიკული სექციის ჩავეტვა
    pthread_mutex_lock(&lock);
    cout << "პირველი thread-ი" << endl;
    int* m = (int*)arg;
    for (int i = 0; i<N; i++)
        m[i] = rand() % 50;
    cout << "პირველი thread-ი დასრულდა." << endl;
    // მოვახდინოთ კრიტიკული სექციის გახსნა
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
// ფუნქცია მეორე thread-ისთვის
void* ave(void* arg){
    // მოვახდინოთ კრიტიკული სექციის ჩავეტვა
    pthread_mutex_lock(&lock);
    cout << "მეორე thread-ი" << endl;
    int* m = (int*)arg;
    double sum = 0;
    for (int i = 0; i<N; i++)
        sum += m[i];
    cout << "საშუალო არის " << sum / N << endl;
    cout << "მეორე thread-ი დასრულდა." << endl;
    // მოვახდინოთ კრიტიკული სექციის გახსნა
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
int main(){
    int mass[N];
    int x;
    pthread_t t1, t2;
    // მოვახდინოთ mutex-ის ცვლადი ინიციალიზირება
    x = pthread_mutex_init(&lock, NULL);
    if (x){
        cout << "არ მოხდა mutex-ის ინიციალიზაცია.\n";
        exit(EXIT_FAILURE);
    }
}
```

```

// შევქმნათ thread-ი 1 და 2
pthread_create(&t1, NULL, fill, (void*)&mass);
pthread_create(&t2, NULL, ave, (void*)&mass);
// ძირითად მიზურობით მოვლა მონაცემებს განათავსებს
pthread_join(t1, NULL);
pthread_join(t2, NULL);
// mutex-ის ცვლადს მოვუხსნათ ინიციალიზაცია
pthread_mutex_destroy(&lock);
exit(EXIT_SUCCESS);
}

```

მაგალითი 6. დაწერეთ პროგრამა, რომელშიც გვეყოლება ორი შვილი პროცესი. პირველ შვილი პროცესს ეყოლება საკუთარი ორი thread-ი, რომელთაგან პირველი ფაილიდან წაიკითხვას მონაცემებს მასივში, ხოლო მეორე მასივში არსებულ მონაცემებს განათავსებს pipe არხში. მეორე შვილ პროცესსაც ეყოლება საკუთარი ორი ნაკადი, რომელთაგან პირველი pipe არხიდან წაიკითხავს მონაცემებს, ხოლო მეორე შექმნის ფაილს და ამ ფაილში ჩაწერს მასივში არსებულ მონაცემებს. პირველ შვილ პროცესში thread-ებს შორის სინქრონიზაცია გააკეთეთ semaphore-ის გამოყენებით, ხოლო მეორეში კი mutex-ის გამოყენებით.

```

#include <fcntl.h>
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <semaphore.h>
using namespace std;
sem_t se;
pid_t p1, p2;
const int N = 5;
char m1[N], m2[N];
pthread_mutex_t mu;
int fd[2], st1, st2;
pthread_t t1, t2, t3, t4;
// ფუნქციების thread-ების შესასრულებლად
void* read1(void* arg);
void* read2(void* arg);
void* write1(void* arg);
void* write2(void* arg);
// ძირითადი thread-ი
int main(){
    // შევქმნათ pipe არხი
    if (pipe(fd) == -1) {
        cout << "pipe არხი არ შექმნა.\n";
        exit(EXIT_FAILURE);
}

```

```

}

// შევქმნათ პირველი შვილი პროცესი
p1 = fork();
if (p1 == -1){
    cout << "პირველი შვილი პროცესი არ შეიქმნა.\n";
    exit(EXIT_FAILURE);
}
if (p1 == 0) { // პირველი შვილი პროცესი
    // შევქმნათ thread-ი 1 და 2
    pthread_create(&t1, NULL, read1, (void*)&m1);
    pthread_create(&t2, NULL, write1, (void*)&m1);
    // thread-ი 1 და 2 მივაერთოთ ძირითად thread-თან
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
else {
    wait(&st1);
    // შევქმნათ მეორე შვილი პროცესი
    p2 = fork();
    if (p2 == -1){
        cout << "მეორე შვილი პროცესი არ შეიქმნა.\n";
        exit(EXIT_FAILURE);
    }
    if (p2 == 0) { // მეორე შვილი პროცესი
        // მოვახდინოთ mutex-ისთვის ცვლადის ინიციალიზირება
        if (pthread_mutex_init(&mu, NULL)) {
            cout << "არ მოხდა mu ცვლადის ინიციალიზირება.\n";
            exit(EXIT_FAILURE);
        }
        // შევქმნათ thread-ი 3 და 4
        pthread_create(&t3, NULL, read2, (void*)&m2);
        pthread_create(&t4, NULL, write2, (void*)&m2);
        // thread-ი 3 და 4 მივაერთოთ ძირითად thread-თან
        pthread_join(t3, NULL);
        pthread_join(t4, NULL);
        // mutex-ის ცვლადს მოვუხსნათ ინიციალიზაცია
        pthread_mutex_destroy(&mu);
    }
    else {
        wait(&st2);
        cout << "პროგრამა წარმატებით დასრულდა.\n";
    }
}
exit(EXIT_SUCCESS);
}

void* read1(void* arg)

```

```

{
    char* ch = (char*)arg;
    // გავხსნათ ფაილი
    int FD = open("data.in", O_RDONLY);
    if (FD == -1){
        cout << "წასაკითხი ფაილი არ გაიხსნა.\n";
        exit(EXIT_FAILURE);
    }
    // ფაილიდან წავიკითხოთ მონაცემები
    sem_post(&se);
    size_t s1 = read(FD, ch, N);
    if ( s1 != N ){
        cout << "მონაცემების კითხვისას დაფიქსირდა შეცდომა.\n";
        exit(EXIT_FAILURE);
    }
    close(FD);
    pthread_exit(NULL);
}

void* write1(void* arg)
{
    close(fd[0]);
    char* ch = (char*) arg;
    sem_wait(&se);
    // მონაცემების ჩაწეროთ pipe არხში
    size_t s1 = write(fd[1], ch, N);
    if ( s1 != N ){
        cout << "მონაცემების ჩაწერისას დაფიქსირდა შეცდომა.\n";
        exit(EXIT_FAILURE);
    }
    close(fd[1]);
    pthread_exit(NULL);
}

void* read2(void* arg){
    // მოვახდინოთ კრიტიკული სექციის ჩაკეტვა
    pthread_mutex_lock(&mu);
    char* ch = (char*) arg;
    close(fd[1]);
    // pipe არხიდან წავიკითხოთ მონაცემები
    size_t s1 = read(fd[0], ch, N);
    if (s1 != N){
        cout << "მონაცემების კითხვისას დაფიქსირდა შეცდომა.\n";
        exit(EXIT_FAILURE);
    }
    close(fd[0]);
    // მოვახდინოთ კრიტიკული სექციის გახსნა
    pthread_mutex_unlock(&mu);
}

```

```

pthread_exit(NULL);
}

void* write2(void* arg){
    // მოვახდინოთ კრიტიკული სექციის ჩაპეტება
    pthread_mutex_lock(&mu);
    char* ch = (char*) arg;
    (void)umask(0);
    // შევქმნათ ფაილი
    int FD = open("data.out", O_WRONLY | O_CREAT, 0600);
    if (FD == -1){
        cout << "ჩასაწერი ფაილი არ შეიქმნა და გაიხსნა.\n";
        exit(EXIT_FAILURE);
    }
    // ფაილში ჩავწეროთ მონაცემები
    size_t s1 = write(FD, ch, N);
    if (s1 != N){
        cout << "მონაცემების ჩაწერისას დაფიქსირდა შეცდომა.\n";
        exit(EXIT_FAILURE);
    }
    close(FD);
    // მოვახდინოთ კრიტიკული სექციის გახსნა
    pthread_mutex_unlock(&mu);
    pthread_exit(NULL);
}

```