Algoritmi vremenskog upravljanja u operacijskim sustavima

Dumančić, Lucija

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET

ZAVRŠNI RAD

ALGORITMI VREMENSKOG UPRAVLJANJA U OPERACIJSKIM SUSTAVIMA

Lucija Dumančić

Split, rujan 2022.

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET

ZAVRŠNI RAD

ALGORITMI VREMENSKOG UPRAVLJANJA U OPERACIJSKIM SUSTAVIMA

Lucija Dumančić

Mentor: Doc. dr. sc. Jelena Nakić

Split, rujan 2022.

Temeljna dokumentacijska kartica

Završni rad

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku
Ruđera Boškovića 33, 21000 Split, Hrvatska

ALGORITMI VREMENSKOG UPRAVLJANJA U OPERACIJSKIM SUSTAVIMA

Lucija Dumančić

SAŽETAK

Cilj završnog rada je implementacija i usporedba algoritama vremenskog upravljanja.. Na početku teorijskog dijela rada, opisani su procesi u operacijskim sustavima i vremensko upravljanje. Osim implementiranih algoritama, opisani su i ostali algoritmi vremenskog upravljanja u operacijskim sustavima. Na kraju teorijskog dijela rada navedeni su i općeniti i specifični ciljevi za pojedine operacijske sustave. U praktičnom dijelu radu implementirani su algoritmi *First-Come First-Served, Shortest Job Next, Round Robin* i *Priority Scheduling.* Programski dio rada je izrađen u okruženju Google Colaboratory koje podržava programski jezik Python. Navedeni algoritmi su analizirani i uspoređeni na temelju prosječnog vremena obrade i prosječnog vremena čekanja. U privitku se nalazi programski kod za svaki od navedenih algoritama.

Ključne riječi: proces, vremensko upravljanje, algoritmi, First-Come First-Served, , Shortest Job Next, Round Robin, Priority Scheduling, vrijeme obrade, vrijeme čekanja

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 42 stranice, 13 grafičkih prikaza, 0 tablica i 6 literaturnih navoda. Izvornik je na hrvatskom jeziku.

Mentor: **Doc. dr. sc. Jelena Nakić,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: **Doc. dr. sc. Jelena Nakić,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ines Šarić-Grgić, *asistent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Lucija Bročić, *asistent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: **rujan 2022.**

Basic documentation card

Thesis

University of Split
Faculty of Science
Department of Computer Science
Ruđera Boškovića 33, 21000 Split, Croatia

OPERATING SYSTEMS SCHEDULING ALGORITHMS

Lucija Dumančić

ABSTRACT

This thesis deals with the implementation and comparison of operating systems scheduling algorithms. At the beginning of the theoretical part of the work, processes in operating systems and scheduling are described. In addition to the implemented algorithms, other operating systems scheduling algorithms are also described. At the end of the theoretical part of the paper, both general and specific goals for individual operating systems are listed. In the practical part of the work, the First-Come First-Served, Shortest Job Next, Round Robin and Priority Scheduling algorithms were implemented. The programming part of the work was created in the Google Colaboratory environment, which supports the Python programming language. The mentioned algorithms were analyzed and compared based on average processing time and average waiting time. The program code for each of the mentioned algorithms is in the attachment.

Key words: process, scheduling, algorithms, First-Come First-Served, Shortest Job Next, Round Robin, Priority Scheduling, turnaround time, waiting time

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of: 42 pages, 13 figures, 0 tables and 6 references

Original language: Croatian

Mentor: **Jelena Nakić, Ph.D.** *Associate Professor of Faculty of Science, University of Split*

Reviewers: **Jelena Nakić, Ph.D.** *Associate Professor of Faculty of Science, University of Split*

Ines Šarić-Grgić, *Assistant of Faculty of Science, University of Split*

Lucija Bročić, *Assistant of Faculty of Science, University of Split*

Thesis accepted: **September 2022.**

Sadržaj

1. Uvod

Iz znatiželje razumijevanja niza događaja koji rezultiraju kvalitetnim radom računala i brzim prebacivanjem s jednom programa na drugi došla sam na ideju proučavanja algoritama za vremensko upravljanje u operacijskim sustavima. Na kolegiju *Operacijski sustavi* stekla sam osnovno znanje o procesima, a ostali programski kolegiji služili su kao podloga za izradu praktičnog dijela završnog rada.

Cilj ovog rada je obrada najpopularnijih algoritama za vremensko upravljanje te njihova analiza i usporedba.

Rad je podijeljen u dva poglavlja. U prvom poglavlju opisana je teorijska podloga o procesima u operacijskim sustavima te vremensko upravljanje. Vremensko upravljanje podijeljeno je u četiri odlomka, opisujući algoritme u tri vrste operacijskih sustava. U drugom poglavlju opisana je implementacija algoritama te način na koji su testirani. U privitku se nalazi cijeli programski kod za svaki implementirani algoritam.

Na kraju rada se nalazi osvrt na analizu i usporedbu implementiranih algoritama.

2. Procesi u operacijskim sustavima

Prema [1], proces je jedna od najosnovnijih apstrakcija koje operacijski sustav pruža korisnicima. Neformalna definicija procesa je program u izvršavanju. Korisnik često želi pokrenuti više od jednog programa istovremeno. Tipičan sustav može naizgled izvoditi desetke procesa u isto vrijeme. Operacijski sustav stvara ovu iluziju virtualizacijom procesora. Pokretanjem jednog procesa, zatim njegovim zaustavljanjem i pokretanjem drugog, i tako dalje, operacijski sustav može stvoriti iluziju da postoje mnogi virtualni procesori dok zapravo postoji samo jedan ili nekoliko fizičkih procesora.

2.1. Upravljanje procesima

U većini sustava procesi se mogu izvršavati istovremeno, a mogu se stvarati i brisati dinamički. Shodno tome, sustavi moraju osigurati sistem za stvaranje i završetak procesa.

Tijekom izvođenja, proces može stvoriti nekoliko novih procesa putem sistemskog poziva. Proces koji kreira novi proces naziva se *roditeljski proces,* a novi procesi nazivaju se *djeca* tog procesa. Svaki od novih procesa može zauzvrat stvoriti druge procese, tvoreći *stablo procesa*. Većina operacijskih sustava identificira procese prema jedinstvenom identifikatoru procesa, koji je obično cijeli broj. Općenito, proces treba određene resurse (procesorsko vrijeme, memoriju, datoteke, ulazno/izlazne uređaje) da bi izvršio svoj zadatak. Proces koji je dijete može dobiti svoje resurse direktno iz operacijskog sustava ili može biti ograničen na podskup resursa nadređenog procesa.

Proces se završava kada završi sa izvršavanjem svoje posljednje faze te traži operacijski sustav da ga izbriše. Do brisanja procesa može doći i u drugim okolnostima. Proces može uzrokovati prekid drugog procesa. Roditelj može prekinuti izvršavanje jednog od svoje djece procesa iz različitih razloga:

- Dijete proces je prekoračilo korištenje nekih resursa koji su mu dodijeljeni.
- Zadatak dodijeljen djetetu više nije potreban.
- Roditelj izlazi iz sustava, a operacijski sustav ne dopušta procesima djeci da nastave izvršavanje ako njihov roditelj prekine.

Neki sustavi ne dopuštaju postojanje djeteta ako je njegov roditelj prestao sa izvršavanjem, iz nekog razloga. [2]

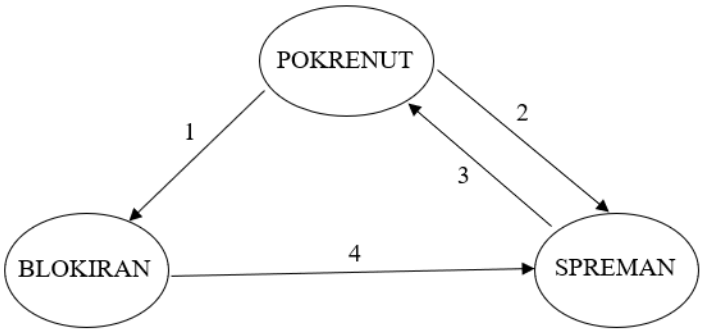
2.2. Stanja procesa

Procesi mogu imati 3 stanja, a to su:

- pokrenut (eng. *running*)
- spreman (eng. *ready*)
- blokiran (eng. *blocked*).

Pokrenuti proces trenutno koristi procesor. Spreman proces je pripravan za pokretanje, ali je privremeno zaustavljen zbog izvođenja drugog procesa. Blokiran proces se ne može pokrenuti sve dok se ne dogodi neki vanjski događaj. [1]

Na slici 1 su prikazana četiri međukoraka. U međukoraku 1 nema ulaznog podatka pa se proces ne može pokrenuti. U međukoraku 2 raspoređivač trenutnom procesu oduzima procesor koji je dodijeljen nekom drugom procesu. U međukoraku 3 raspoređivač vraća procesor procesu koji je bio aktivan prije trenutnog procesa. U međukoraku 4 proces je pokrenut zbog vanjskog događaja.



Slika 1 Stanja procesa

2.3. Vremensko upravljanje

Upravljanje procesorom se bitno razlikuje u ovisnosti o tome radi li se o jednokorisničkom sustavu ili sustavu sa višestrukim izvršavanjem. U jednokorisničkom sustavu, procesor je zauzet samo kada korisnik izvršava posao, u svim ostalim trenucima je u stanju mirovanja. Upravljanje procesorom u ovom okruženju je jednostavno. Kod računala sa višestrukim izvršavanjem očekivano je da se više procesa istovremeno natječe za procesorske resurse, tj.

vrijeme. Procesor mora biti dodijeljen svakom procesu na pravedan i učinkovit način, što može biti izazovno. [3]

Postavlja se pitanje što je dobro vremensko upravljanje procesima. Različiti algoritmi za vremensko upravljanje imaju različita svojstva, stoga izbor određenog algoritma može dati prednost jednoj klasi procesa u odnosu na drugu. Prema [3] postoji nekoliko kriterija za odabir:

- *Maksimizirati propusnost* pokretanjem što je više moguće procesa u određenom vremenu. Propusnost se mjeri brojem procesa koji se dovrše po jedinici vremena.
- *Minimizirati vrijeme odziva* poboljšanjem interaktivnih zahtjeva. U interaktivnim sustavima vrijeme obrade možda nije najbolji kriterij. Općenito je ograničeno brzinom izlaznog uređaja.
- *Minimizirati vrijeme obrade* brzim premještanjem cijelih poslova u sustav i izvan njega. To se može postići tako da se prvo pokrenu svi skupni procesi. Skupni procesi mogu se grupirati tako da se izvode učinkovitije od interaktivnih procesa.
- *Minimizirati vrijeme čekanja* premještanjem procesa iz reda čekanja što je brže moguće.
- *Maksimizirati učinkovitost procesora* održavajući procesor zauzetim cijelo vrijeme.
- *Osigurati pravednost za sve procese* dajući svim procesima jednaku količinu procesorskog i ulazno/izlaznog vremena.

U prvim generacijama računala raspoređivanje je bilo trivijalno učitavanjem niza (eng. *batch*) zadataka. Današnja računala se temelje na principu podjele vremena (eng. *time-sharing*), iako i dalje postoje sustavi koji kombiniraju oba načina. Kod sustava u kojima je procesorsko vrijeme vrijedan resurs, raspoređivač može činiti veliku razliku u dojmu brzine izvršavanja i zadovoljstva korisnika.

2.3.1. U serijskim sustavima

Serijski ili skupni (eng. *batch*) sustavi su sustavi kod kojih korisnici ne komuniciraju izravno s računalom. Korisnik pripremi program na nekom uređaju koji je izvan mreže (npr. USB), a zatim ga prenosi računalnom operatoru. [2] Nakon toga operator sortira programe u grupe prema zajedničkom svojstvu kako bi ubrzao proces. Serijski sustavi imaju svoje prednosti i mane. Jedna od prednosti serijskog sustava je što se proces može odvijati bez prisutnosti osobe, u slučaju da se radi o poslu koji zahtjeva više vremena. [1] Druga prednost je što ne

zahtjeva posebno računalno sklopovlje (eng. *hardware*) i sustavnu podršku za unos podataka. Mana serijskog sustava je teže uklanjanje grešaka te manjak interakcije između korisnika i operacijskog sustava. Također, ukoliko se pojavi greška, ostali zadani poslovi se zaustavljaju i čekaju otklanjanje greške. U izvršavanju procesa je jako malo prekida ili ih uopće nema. Zahtjev ispisa na računalu je jedan od primjera takvog serijskog posla. Važan pojam koji se veže uz serijske sustave je pojam serijske obrade poslova koja se odnosi na rad računala kroz red ili seriju odvojenih programa bez ručnog posredovanja.

Algoritmi raspoređivanja u serijskim sustavima su:

- *First-Come First-Served (FCFS)*
- *Shortest Job First (SJF)*
- *Shortest Remaining Time Next (SRTN)*

U sljedećim odlomcima objašnjen je svaki od navedenih algoritama.

Prioritet po redu dospijeća - *FIRST-COME FIRST-SERVED (FCFS)*

First-Come First-Served (FCFS) je najjednostavniji neprekidajući algoritam raspoređivanja koji se temelji na principu *First In First Out* (FIFO). Na početku reda nalazi se prvi proces koji se izvršava bez prekida, a svaki sljedeći proces stavlja se na kraj reda te čeka svoj red za izvršavanje. Ukoliko je neki proces blokiran, stavlja se na kraj reda. Svaki proces se izvršava puno vrijeme trajanja bez prekida. Ovaj algoritam ima najdulje vrijeme obrade. [3] Prednosti ovog algoritma su jednostavno izvršavanje i praćenje procesa te jednostavna implementacija. Međutim, mane ovog algoritma su dulje vrijeme izvršavanja i ulazno/izlazni uređaji se ne koriste tijekom izvođenja. Procesi koji nisu vremenski zahtjevni predugo čekaju u redu na izvršavanje, a algoritam prednost daje procesima koji su povezani s procesorom, a ne s ulazno/izlaznim uređajima.

Prioritet po vremenu izvršavanja - *SHORTEST JOB FIRST (SJF)*

Shortest Job First (SJF) je jednostavni neprekidajući algoritam koji bira onaj proces koji je najmanje vremenski zahtjevan. Ako su dva procesa jednako vremenski zahtjevna, onda se primjenjuje FCFS algoritam i odabire se onaj proces koji je prvi došao. U odnosu na ostale algoritme, ovaj algoritam ima optimalno vrijeme izvršavanja. SJF algoritam raspoređivanja

je dokazano optimalan, jer daje minimalno prosječno vrijeme čekanja za dati skup procesa. [2] Postoji i prekidajuća verzija ovog algoritma, pod nazivom *Shortest Remaining Time Next (SRTN).*

Prednosti SJF algoritma su što uvijek izvršava proces koji je najmanje vremenski zahtjevan te je prosječno vrijeme čekanja kraće u odnosu na prethodno opisani algoritam. Ukoliko postoji puno kratkih poslova, dulji poslovi su zaustavljeni pa je to jedna od mana ovog algoritma. Druga mana je što vrijeme izvršavanja određenog procesa treba biti poznato unaprijed.
Najvažnija razlika između SJF i FCFS algoritama je prosječno vrijeme čekanja koje je kod SJF algoritma bitno kraće.

Prioritet po prvom najkraćem preostalom vremenu - *SHORTEST REMAINING TIME NEXT (SRTN)*

Kao što je već navedeno, ovaj algoritam je prekidajuća verzija SJF algoritma. Proces koji je najmanje vremenski zahtjevan se izvršava sve dok ne dođe sljedeći proces, bez obzira na njegovu vremensku zahtjevnost. Proces koji se izvršavao staje te se izračuna ukupno vrijeme trajanja *(*eng. *total time)*, a zatim se odabire najkraći proces koji se tada izvršava sve do pojave sljedećeg procesa. U trenutku kada se ukupno vrijeme trajanja izjednači sa trajanjem procesa koji dolazi zadnji, tj. kada svi procesi postanu dostupni, onda se počinje primjenjivati SJF algoritam. Budući da je ovo prekidajuća vrsta algoritma, vrijeme čekanja je smanjeno u odnosu na SJF algoritam. Neke od prednosti ovog algoritma su kratko vrijeme odziva i jako brzo izvršavanje kratkih poslova. S druge strane, ovaj algoritam jako opterećuje procesor i zbog prebacivanja se smanjuje vremenska ušteda. [2]

2.3.2. U interaktivnim sustavima

Interaktivni sustavi pružaju bolje vrijeme obrade od serijskih sustava, ali su sporiji od sustava u stvarnom vremenu. Uvedeni su kako bi zadovoljili zahtjeve korisnika kojima je bila potrebna brza obrada kada otklanjaju pogreške u svojim programima. Operacijski sustavi zahtijevali su razvoj programske podrške za dijeljenje vremena (eng. *time-sharing software*) koji bi svakom korisniku omogućio izravnu komunikaciju s računalnim sustavom putem naredbi unesenih s terminala. Interaktivni operacijski sustav pruža neposrednu

povratnu informaciju korisniku, a vrijeme odziva se može mjeriti u minutama i sekundama. [3]

Ovakav sustav korisniku omogućuje jednostavno upravljanje sustavom i procjenu rezultata provedenih aktivnosti radi postizanja određenih ciljeva. Osim primjene na računalima, interaktivni sustavi našli su svoju primjenu i u drugim uređajima poput mobitela i navigacijskih sustava.

Algoritmi raspoređivanja u interaktivnim sustavima su:

- *Round Robin (RR)*
- *Priority Scheduling (PS)*
- *Shortest Process Next (SPN)*
- *Guaranteed Scheduling*
- *Lottery Scheduling*
- *Fair-Share Scheduling*

U sljedećim odlomcima objašnjen je svaki od navedenih algoritama.

Kružno posluživanje - *ROUND ROBIN (RR)*

Kružno posluživanje zasniva se na FCFS algoritmu, s tim da je dodano vremensko ograničenje u korištenju procesora. Ovaj algoritam je jedan od najčešće korištenih algoritama raspoređivanja jer je najjednostavniji i najpravedniji. Vremenski interval unutar kojeg se proces ima pravo izvršiti naziva se kvant i dodjeljuje se svakom procesu. Kvant se kreće između od 10 do 100 milisekundi ili od 1 do 2 sekunde. Jedna od prednosti ovog algoritma je jednostavna implementacija. Ukoliko se proces ne završi u zadanom vremenskom intervalu, proces se prekida, a procesor se dodjeljuje nekom drugom procesu. Proces koji je iskoristio svoj vremenski interval se postavlja na kraj reda. [3] Upravo zbog toga je važno da raspoređivač održi listu procesa koji su pokretljivi. Svaki proces dobije jednak dio procesora. Mana ovog algoritma je duljina kvanta. Naime, zamjenom procesa gubi se vrijeme na administraciju kao što su punjenje registara, memorijsko mapiranje, osvježavanje tablica i lista te pražnjenje i ponovno punjenje predmemorije. Problem se također javlja ako je vremenski interval dulji od vremena za izvršavanje procesa. Promjena procesa i smanjenje učinkovitosti procesora dogodit će se ukoliko je vremenski interval prekratak. [1] S druge strane, ukoliko je vremenski interval predug može doći do slabijeg

odgovora na vremenski kraće procese. Prema [2] 80% procesorske obrade treba biti kraće od vremenskog intervala. Prosječno vrijeme čekanja duže je u odnosu na ostale algoritme te uzrokuje veće opterećenje procesora.

Posluživanje sa prioritetima – *PRIORITY SCHEDULING (PS)*

Posluživanje sa prioritetima temelji se na raspoređivanju prema razinama prioriteta. Svaki proces dobije prioritet koji je podložan promjenama. Procesi se raspoređuju uz pomoć FCFS algoritma. Ukoliko dva procesa imaju isti prioritet prednost ima onaj proces koji je prvi došao. Jedna od mana ovog algoritma su procesi s visokim prioritetom koji se mogu izvršavati neodređeno dugo. U jako opterećenom računalnom sustavu, stalan tok procesa višeg prioriteta može spriječiti da proces niskog prioriteta dobije procesor. Rješenje problema neodređene blokade nisko-prioritetnih procesa je starenje (eng. *aging*). Starenje je tehnika postupnog povećanja prioriteta procesa koji dugo čekaju u sustavu. Tim postupkom bi čak i proces s početnim niskim prioritetom imao najviši prioritet u sustavu te bi bio izvršen. [2] Drugi način za rješavanje problema je dodjeljivanje vremenskog intervala maksimalne duljine svakom procesu. Nakon isteka vremenskog intervala zaustavlja se trenutni proces i sljedeći proces se počinje izvršavati. Također, mana ovog algoritma je mogućnost odgađanja procesa s nižim prioritetom zbog procesa s visokim prioritetom koji zauzmu veliki dio procesora.

Prioriteti mogu biti definirani interno ili eksterno. Interno definirani prioriteti koriste neku mjerljivu veličinu ili količine za izračunavanje procesa. Na primjer, vremenska ograničenja, broj otvorenih datoteka i zahtjevi za memorijom korišteni su u računalnim prioritetima. Eksterni prioriteti postavljeni su kriterijima izvan operacijskog sustava, kao što je važnost procesa. [2]

Prednost najkraćeg procesa – *SHORTEST PROCESS NEXT (SPN)*

Ovaj algoritam nastao je na temelju SJF algoritma za raspoređivanje te se procesi raspoređuju prema trajanju procesa čime se minimizira vrijeme čekanja. Proces s najkraćim trajanjem se izvodi do kraja, bez prekida. Problem je kako odrediti koji od pripravnih procesa je najkraći. Nakon svakog pokretanja procesa računa se ukupno vrijeme (eng. *total time*) te se uz pomoć njega odabire sljedeći proces s najkraćim trajanjem. [4]

Garantirano raspoređivanje – *GUARANTEED SCHEDULING*

Garantirano raspoređivanje temelji se na ravnomjernoj raspodijeli procesora koji se obeća svakom procesu na početku. Međutim, algoritam mora postupno prilagođavati prioritete kako bi ispoštovao dano obećanje. Onaj proces koji ima najmanji omjer iskorištenog i dobivenog vremena ima prioritet pri raspoređivanju procesa za izvršavanje. Ovaj algoritam je dobro zamišljen, ali ga je tako teško realizirati. [4]

Lutrijsko raspoređivanje – *LOTTERY SCHEDULING*

Lutrijsko raspoređivanje temelji se na teoriji vjerojatnosti budući da svaki proces dobije jednu karticu, tzv. srećku. Za svaki vremenski interval, na slučajan način, dobije se skup kartica. Iz toga proizlazi zaključak da je vrijeme na procesoru proporcionalno broju kartica (srećki) koji dobije svaki proces. Međutim, oni procesi koji imaju viši prioritet dobiju više kartica (srećki). Svaki proces mora dobiti barem jednu karticu (srećku) kako bi napredovao. Algoritam za raspoređivanje na slučajan način odabire jednu karticu (srećku) te proces koji ima tu karticu (srećku) dobije pravo na resurs. Budući da neki procesi imaju veći broj kartica (srećki) oni imaju i veću mogućnost dobitka. Ukoliko se završi ili doda novi proces, to se na podjelu procesora održava proporcionalno, bez obzira koliko kartica ima pojedini proces. [4]

Algoritam pravilne raspodjele – *FAIR-SHARE SCHEDULING*

Ovaj algoritam, kao što mu i ime kaže, radi na principu ravnomjerne raspodjele procesora među svim korisnicima ili skupinama korisnika. [4] Ukoliko je aktivno 5 korisnika koji u istom trenutku izvršavaju po jedan proces, tada raspoređivač svakom korisniku dodjeljuje 20% procesora. Dodijeljeni dio procesora se ne mijenja ako neki korisnik pokrene još jedan proces. U tom slučaju svaki proces koristi po 10% procesora od mogućih 20%. Međutim, pojavom novog korisnika mijenja se dodijeljeni dio procesora i tada svaki korisnik dobije približno 16.7% procesora. Druga mogućnost je podjela korisnika u skupine pri čemu se procesor dijeli prema broju skupina. Nakon toga se dodijeljeni dio procesora dijeli unutar skupine.

Pretpostavimo da postoje 4 skupine korisnika. Tada svaka skupina dobije po 25% procesora.

Ako se prva skupina sastoji od 2 korisnika onda svaki korisnik dobije 12.5% procesora. Ako se druga skupina sastoji od 3 korisnika onda svaki korisnik dobije približno 8.3% procesora. Ako se treća skupina sastoji od 4 korisnika onda svaki korisnik dobije 6.25% procesora. Ako se četvrta skupina sastoji od 5 korisnika onda svaki korisnik dobije 5% procesora.

2.3.3. U sustavima u realnom vremenu

Sustavi stvarnog vremena su bazirani na kvaliteti te im je vrijeme od apsolutne važnosti. Podaci se obrađuju u trenutku dolaska u sustav te postoji vremensko ograničenje unutar kojeg se ti podaci moraju obraditi. Ukoliko se ne poštuju ograničenje to će dovesti do pada sustava. Postoje dvije vrste ovog sustava – striktni i popustljivi sustavi. [4] Striktni sustavi imaju stroge zahtjeve jer oni jamče da će osjetljivi procesi biti dovršeni na vrijeme. Primjer takvih sustava su roboti u automobilskoj industriji, monitori za nadzor pacijenata na jedinicama intenzivnog liječenja, autopiloti u zrakoplovnoj industriji i slično. [3] Popustljivi sustavi su manje restriktivni. Naime, oni osiguravaju da će osjetljivi procesi dobiti prioritet u odnosu na druge procese te da će taj prioritet zadržati sve dok se proces ne izvrši. Takvi sustavi se koriste u komercijalnim operativnim sustavima.

2.3.4. Ciljevi algoritma

Svaki od opisanih algoritama karakterističan je prema određenim ciljevima. Ciljevi serijskih sustava su propusnost, vrijeme proračuna i iskoristivost procesora. Propusnost se odnosi na maksimalni broj poslova po satu, vrijeme između početka i završetka nastoji se minimalizirati, a procesor se nastoji održavati zaposlenim cijelo vrijeme. [4]

Ciljevi interaktivnih sustava su vrijeme odziva i razmjernost. Razmjernost se odnosi na ispunjenje korisnikovih očekivanja, a vrijeme odziva na brzi odziv na korisnikov zahtjev. [4]

Ciljevi sustava stvarnog vremena su poštivanje rokova i predvidljivost. Poštivanjem rokova nastoji se spriječiti gubitak podataka, a kod predvidljivost se nastoji izbjeći gubitak kvalitete u multimedijskim sustavima. [4]

Također, postoje ciljevi koji su zajednički svim sustavima, a to su jednakost, poštivanje pravila sustava i ravnoteža. Jednakost se odnosi na ravnomjernu raspodjelu procesora, a ravnoteža se odnosi na održavanje svih dijelova sustava zaposlenima. [4]

3. Implementacija algoritama

Ideja praktičnog dijela je testirati implementirane algoritme u različitim uvjetima. Programski dio projekta je izrađen u okruženju Google Colaboratory koje podržava programski jezik Python. Za implementaciju su odabrana 4 najpopularnija algoritma: *First-Come First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR)* i *Priority Scheduling (PS)*. Praktični dio, točnije kod, nalazi se u privitku. Na početku rada korisnik iz izbornika odabire koji algoritam želi pokrenuti te zatim može unijeti željeni broj procesa. Neovisno o izboru algoritma od korisnika se traži da unese jedinstveni ID za svaki proces. Parametri koji su zajednički svim algoritmima su vrijeme dolaska (eng. *arrival time*) i vrijeme izvršavanja (eng. *burst time*). Vrijeme dolaska je vrijeme koje je potrebno procesu da bude spreman da se izvrši, odnosno da prijeđe u stanje *spreman*, i stane u red za izvršavanje. S druge strane, vrijeme izvršavanja označava koliko vremena je procesu potrebno da se izvrši. Kod RR i PS algoritama postoje dodatni parametri koje korisnik treba unijeti. Vremenski interval (eng. *quant time*) je parametar usko vezan uz RR algoritam i označava interval za izvršavanje procesa. Za PS algoritam je potrebno dodatno za svaki proces unijeti prioritet. Nakon unosa potrebnih parametara korisniku se ispisuje rezultat. Računaju se prosječno vrijeme obrade (eng. *turnaround time*) i vrijeme čekanja (eng. *waiting time*). Vrijeme obrade označava vrijeme koje je potrebno procesu od ulaska u red za izvršavanja do njegovog izvršenja. Vrijeme čekanja je ukupno vrijeme koje proces provede dok čeka spreman u redu za izvršavanje. Redoslijed izvršavanja procesa je prikazan nizom koji predstavlja vremensku crtu.

3.1. Testiranje

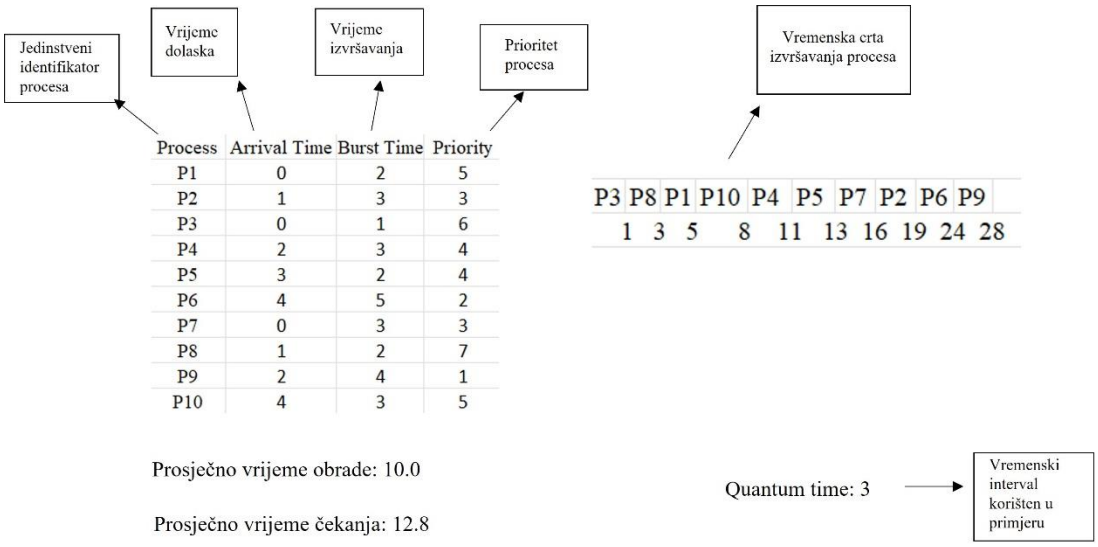
Implementirani algoritmi su podvrgnuti zanimljivim situacijama kojima se mogu najbolje okarakterizirati. Parametri na osnovu kojih se radi usporedba algoritama su: prosječno vrijeme obrade i prosječno vrijeme čekanja. Iako ovi parametri ne određuju je li algoritam dobar ili ne, mogu biti interesantni za analizirati kroz različite testne podataka. U nekim primjerima su korišteni isti testni podaci s ciljem da se prikaže kako različiti algoritmi djeluju u istim situacijama. U ostalim primjerima testni podaci su prilagođeni konkretnom algoritmu. Uz same rezultate algoritama navode se komentari i opažanja.

Vrijeme obrade se računa kao razlika vremena završetka (eng. *completion time*) i vremena dolaska. [1] Vrijeme završetka je vrijeme kada se proces prestaje izvršavati, što znači da je proces završio svoje vrijeme izvršavanja i da je u potpunosti izvršen.

Vrijeme čekanja se mjeri razlikom vremena obrade i vremena izvršavanja.

Za svaki skup procesa računa se prosječno vrijeme obrade i prosječno vrijeme čekanja.

U nastavku je svaki od implementiranih algoritama obrađen posebno. Rezultati primjera su prikazani tablicom. Slika 2 je ogledni pokazatelj glede lakšeg razumijevanja primjera. Tablica lijevo je početno stanje procesa, tj. testni podaci, čiji su stupci detaljnije objašnjeni. S desne strane je rezultat izvršavanja algoritma, odnosno tablica koja predstavlja vremensku crtu izvršavanja procesa. Niz brojeva u donjem dijelu tablice predstavlja trenutke u kojima se procesor prebacivao s jednog procesa na drugi (eng. *context switch*).



Slika 2 Ogledni primjer

Prioritet po redu dospijeća - *FIRST-COME FIRST-SERVED (FCFS)*

Kroz testiranje je primijećeno da je prosječno vrijeme čekanja uglavnom jako dugo. Problem ovog algoritma je što kraći procesi moraju čekati na one duže. Dolazi do učinka konvoja jer svi drugi procesi čekaju da jedan veliki proces izađe iz procesora. Ovaj učinak rezultira manjim korištenjem procesora nego što bi to bilo moguće korištenjem SJF algoritma. [2]

Na slici 3 je primjer FCFS algoritma gdje svi procesi imaju isto vrijeme dospijeća, odnosno 0. Slika 4 je također primjer sustava u kojem pojedini procesi imaju isto vrijeme dospijeća.

U takvim slučajevima prvi se izvršava onaj proces koji ima manji ID. Slika 5 je klasični primjer algoritma gdje su sva vremena dospijeća različita i izvršavanje ide po redu dolaska.

Process	Arrival Time	Burst Time
P1	0	2
P2	0	3
P3	0	1
P4	0	3
P5	0	2
P6	0	5
P7	0	3
P8	0	2
P9	0	4
P10	0	3

P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	
2	5	6	9	11	16	19	21	25		28

Prosječno vrijeme obrade: 14.2

Prosječno vrijeme čekanja: 11.4

Slika 3 First-Come First-Served

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	0	1
P4	2	3
P5	3	2
P6	4	5
P7	0	3
P8	1	2
P9	2	4
P10	4	3

P1	P3	P7	P2	P8	P4	P9	P5	P6	P10	
2	3	6	9	11	14	18	20	25		28

Prosječno vrijeme obrade: 11.9

Prosječno vrijeme čekanja: 9.1

Slika 4 First-Come First-Served

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	5	1
P4	8	3
P5	2	2
P6	6	5
P7	4	3
P8	3	2
P9	9	4
P10	7	3

P1	P2	P5	P8	P7	P3	P6	P10	P4	P9	
2	5	7	9	12	13	18	21	24	28	

Prosječno vrijeme obrade: 9.4

Prosječno vrijeme čekanja: 6.6

Slika 5 First-Come First-Served

Prioritet po vremenu izvršavanja - *SHORTEST JOB FIRST (SJF)*

Kako je već spomenuto SJF algoritam je dokazano optimalan, pružajući minimalno prosječno vrijeme čekanja za dati skup procesa. Premještanje kratkog procesa prije dugog više smanjuje vrijeme čekanja kratkog procesa nego što povećava vrijeme čekanja dugog procesa. Kao posljedica, prosječno vrijeme čekanja se smanjuje. [2] SJF ima očitu prednost jednostavnosti i nešto manje očitu, već spomenutu, prednost da je vrijeme čekanja korisnika u sustavu manje nego u bilo kojem sustavu (uključujući i FCFS sustav) koji ne iskorištava poznata vremena rada. Jasno je da dugotrajni procesi trpe više u SJF sustavu nego u FCFS sustavu. Stoga, smanjenje u prvom trenutku vremena čekanja dolazi po cijenu povećanja u drugom trenutku (ili odstupanja). [5]

Na slici 6 prikazan je izgled vremenske crte izvršavanja procesa, na istim testnim podacima kao i u prethodnom primjeru, koristeći SJF algoritam. Primjer na slici 7 prikazuje slučaj kada se među procesima nalaze oni koji imaju isti AT ili BT. Kada dva procesa imaju isto vrijeme dolaska, kako i sam naziv algoritma kaže, prvi se izvršava onaj proces koji ima manje vrijeme izvršavanja. U slučaju kada dva procesa imaju isto vrijeme izvršavanja djeluje se po principu FCFS algoritma, odnosno izvršava se onaj koji je prije stigao u red za izvršavanje.

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	5	1
P4	8	3
P5	2	2
P6	6	5
P7	4	3
P8	3	2
P9	9	4
P10	7	3

P1	P5	P8	P3	P2	P7	P10	P4	P9	P6	
2	4	6	7	10	13	16	19	23	28	

Prosječno vrijeme obrade: 8.3

Prosječno vrijeme čekanja: 5.5

Slika 6 Shortest Job First

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	0	1
P4	2	3
P5	3	2
P6	4	5
P7	0	3
P8	1	2
P9	2	4
P10	4	3

P3	P1	P8	P5	P7	P2	P4	P10	P9	P6
1	3	5	7	10	13	16	19	23	28

Prosječno vrijeme obrade: 10.8

Prosječno vrijeme čekanja: 8.0

Slika 7 Shortest Job First

Kružno posluživanje - *ROUND ROBIN (RR)*

Prosječno vrijeme čekanja često je jako dugo. Izvedba RR algoritma uvelike ovisi o veličini vremenskog intervala za izvođenje (eng. *Quantum time*). S jedne strane imamo ekstrem gdje je vremenski interval iznimno velik pa je RR algoritam isti kao FCFS algoritam. Drugi ekstrem je ako je vremenski interval iznimno malen (1 milisekunda), RR pristup se naziva dijeljenje procesora. Vrijeme obrade također ovisi o veličini vremenskog intervala. Prosječno vrijeme obrade skupa procesa se ne poboljšava nužno kako se povećava vremenski interval izvršavanja. Općenito, prosječno vrijeme obrade može se poboljšati ako većina procesa obavi svoje izvršavanje u jednom vremenskom intervalu. [2]

Kod Round Robin algoritma prikazane su dvije različite situacije sa istim procesima. Slika 8 je primjer gdje je vremenski interval 2. Vremenska crta kod tog primjera je dosta duža, jer se mnogi procesi nisu uspjeli izvršiti odjednom u zadanom vremenskom intervalu. Samim time i prosječno vrijeme čekanja je dugo. Drugi primjer je na slici 9 kad je vremenski interval povećan na 3. Vremenska crta je kraća i vrijeme čekanja je bolje jer se većina procesa izvršila u jednom vremenskom intervalu.

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	0	1
P4	2	3
P5	3	2
P6	4	5
P7	0	3
P8	1	2
P9	2	4
P10	4	3

P1	P3	P7	P2	P8	P4	P9	P5	P6	P10	P7	P2	P4	P9	P6	P10	P6	
2	3	5	7	9	11	13	15	17	19	20	21	22	24	26	27	28	

Prosječno vrijeme obrade: 15.4 Quantum time: 2

Prosječno vrijeme čekanja: 12.6

Slika 8 Round Robin

Process	Arrival Time	Burst Time
P1	0	2
P2	1	3
P3	0	1
P4	2	3
P5	3	2
P6	4	5
P7	0	3
P8	1	2
P9	2	4
P10	4	3

P1	P3	P7	P2	P8	P4	P9	P5	P6	P10	P9	P6	
2	3	6	9	11	14	17	19	22	25	26	28	

Prosječno vrijeme obrade: 12.6 Quantum time: 3

Prosječno vrijeme čekanja: 9.8

Slika 9 Round Robin

Posluživanje sa prioritetima – PRIORITY SCHEDULING (PS)

Kao što je objašnjeno u teorijskom dijelu, kada proces dođe u red čekanja, njegov prioritet se uspoređuje sa prioritetom procesa koji se trenutno izvršava. Budući da se ovdje radi o prekidajućem posluživanju sa prioritetom, algoritam će prekinuti procesor ako je prioritet novopristiglog procesa veći od prioriteta trenutnog procesa koji se izvršava. Ako se dogodi da dva procesa imaju jednak prioritet postupa se prema FCFS algoritmu. [2] Nije precizno određeno što predstavlja visok, a što nizak prioritet. U nekim sustavima niska brojčana vrijednost predstavlja nizak prioritet, dok je u nekim sustavima to visok prioritet. Konkretno u ovim primjerima veće brojčane vrijednosti znače i veći prioritet. Posluživanje sa prioritetom može dati sporije vrijeme izvršavanja nekim korisnicima.

Na slici 10 je prikazan najjednostavniji primjer posluživanja sa prioritetom. Svi procesi su dostupni u isto vrijeme, a prvi na izvršavanje ide proces s najvećim prioritetom. Slučaj gdje procesi imaju isti prioritet je obrađen na slici 11. U tom primjeru su se procesi, osim po prioritetu, izvršavali i po vremenu dolaska.

Process	Arrival Time	Burst Time	Priority
P1	0	10	3
P2	0	1	1
P3	0	2	4
P4	0	1	5
P5	0	5	2

P4	P3	P1	P5	P2	
1	3	13	18	19	

Prosječno vrijeme obrade: 7.0

Prosječno vrijeme čekanja: 10.8

Slika 10 Priority Scheduling

Process	Arrival Time	Burst Time	Priority
P1	0	2	5
P2	1	3	3
P3	0	1	6
P4	2	3	4
P5	3	2	4
P6	4	5	2
P7	0	3	3
P8	1	2	7
P9	2	4	1
P10	4	3	5

P3	P8	P1	P10	P4	P5	P7	P2	P6	P9	
1	3	5	8	11	13	16	19	24	28	

Prosječno vrijeme obrade: 10.0

Prosječno vrijeme čekanja: 12.8

Slika 11 Priority Scheduling

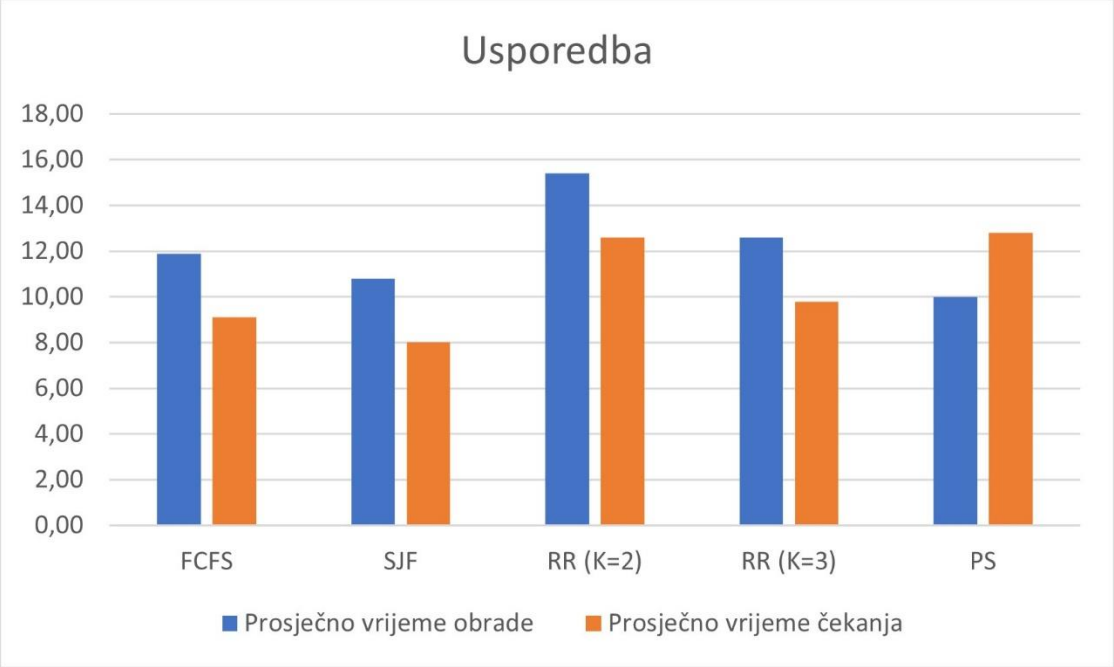
3.2. Osvrt

Testiranjem algoritama uslijedili su različiti rezultati i opažanja. Konačna opažanja su da je FCFS algoritam bolji ako se radi o procesima koji imaju malo vrijeme izvršavanja. Ako procesi dolaze istovremeno u sustav, bolji je SJF algoritam. U slučaju podešavanja željenog prosječnog vremena čekanja RR je najbolji izbor. [5] PS algoritam se temelji na prioritetu u kojem se proces s najvišim prioritetom pokreće prvi, što dovodi do problema neodređenog izvršavanja procesa s višim prioritetom. Preporuča se da svaka vrsta simulacije za bilo koji algoritam vremenskog upravljanja ima ograničenu točnost. Jedini način da se procijeni algoritam za vremensko upravljanje je da se isprogramira te stavi u operacijski sustav. Samo tada se ispravna radna sposobnost može mjeriti u sustavima u stvarnom vremenu. [6]

Na samom kraju na slici 13 je prikazana grafička usporedba sva 4 algoritma. Budući da je jedan testni skup procesa obrađen kroz sve algoritme, moguće je grafički prikazati usporedbu prosječnog vremena obrade i prosječnog vremena čekanja. Testni podaci na temelju kojih je napravljena usporedba su prikazani na slici 12.

Process	Arrival Time	Burst Time	Priority
P1	0	2	5
P2	1	3	3
P3	0	1	6
P4	2	3	4
P5	3	2	4
P6	4	5	2
P7	0	3	3
P8	1	2	7
P9	2	4	1
P10	4	3	5

Slika 12 Testni skup procesa



Slika 13 Grafički prikaz usporedbe

4. Zaključak

Nakon implementacije algoritama i provedene usporedbe povlači se zaključak da ne postoji savršeni algoritam za vremensko upravljanje. Kao što je opisano u samom radu, algoritmi imaju svoje prednosti i nedostatke. Naime, svaki od algoritama zadovoljava potrebe korisnika u određenoj situaciji. Također, značajke algoritma uvelike ovise i o samom operacijskom sustavu te o opterećenosti procesora. S napretkom i razvojem tehnologije, postoji mogućnost da će ovi algoritmi biti zamijenjeni poboljšanim verzijama prilagođenima korisničkim potrebama i zahtjevima suvremenog doba.

Najviše implementacija na ovu temu nalazi se u programskom jeziku C jer on pruža najrealniju simulaciju stvarnog operacijskog sustava. Iako je manje zastupljeno, u ovom radu algoritmi su implementirani u programskom jeziku Python kako bi se pokazalo da je i na taj način moguće dobiti kvalitetne povratne informacije i rezultate.

U ovom radu izostavljen je parametar iskorištenosti procesora (eng. *CPU utilization*) zbog kompleksnosti implementacije. Ovaj rad se može proširiti dodavanjem navedenog parametra koji može biti još jedna zanimljiva stavka na temelju kojih se mogu uspoređivati algoritmi za vremensko upravljanje.

Literatura

[1] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2014). *Operating systems: Three easy pieces*. Arpaci-Dusseau Books, Inc.

[2] Silberschatz, A., Galvin, P. B., & Gagne, G. (2004). *Operating System Concepts.*

[3] McHoes, A., & Flynn, I. M. (1991). *Understanding operating systems*.

[4] Zaharija, G. (2022). *Operacijski sustavi (materijali s predavanja)*

[5] Coffman Jr, E. G., & Kleinrock, L. (1968, April). Computer scheduling methods and their countermeasures. In *Proceedings of the April 30--May 2, 1968, spring joint computer conference* (pp. 11-21).

[6] Singh, P., Singh, V., & Pandey, A. (2014). Analysis and comparison of CPU scheduling algorithms. *International Journal of Emerging Technology and Advanced Engineering*, *4*(1), 91-95.

Skraćenice

FIFO	*First In First Out*	prvi ušao prvi izašao
FCFS	*First-Come First-Served*	prioritet po redu dospijeća
SJF	*Shortest Job First*	prioritet po vremenu izvršavanja
SRTN	*Shortest Remaining Time Next*	prioritet po prvom najkraćem preostalom vremenu
RR	*Round Robin*	kružno posluživanje
PS	*Priority Scheduling*	posluživanje sa prioritetom
SPN	*Shortest Process Next*	prednost najkraćeg procesa
AT	*Arrival Time*	vrijeme dolaska
BT	*Burst Time*	vrijeme izvršavanja

Privitak

Programski kod za algoritam *First-Come First-Served*:

```
class FCFS:
    def processData(self, no_of_processes):
        process_data = []
        for i in range(no_of_processes):
            temporary = []
            process_id = int(input("Unesite ID procesa: "))

            arrival_time = int(input(f"Unesite Arrival Time z
a Proces {process_id}: "))

            burst_time = int(input(f"Unesite Burst Time za Pr
oces {process_id}: "))

            temporary.extend([process_id, arrival_time, burst
_time])
            process_data.append(temporary)
        FCFS.schedulingProcess(self, process_data)

    def schedulingProcess(self, process_data):
        process_data.sort(key=lambda x: x[1])

        """Sort according to Arrival Time"""

        start_time = []
        exit_time = []
        s_time = 0
        for i in range(len(process_data)):
            if s_time < process_data[i][1]:
                s_time = process_data[i][1]
            start_time.append(s_time)
            s_time = s_time + process_data[i][2]
            e_time = s_time
            exit_time.append(e_time)
            process_data[i].append(e_time)
        t_time = FCFS.calculateTurnaroundTime(self, process_d
ata)
```

```
        w_time = FCFS.calculateWaitingTime(self, process_data
)
        FCFS.printData(self, process_data, t_time, w_time)

    def calculateTurnaroundTime(self, process_data):
        total_turnaround_time = 0
        for i in range(len(process_data)):
            turnaround_time = process_data[i][3] - process_da
ta[i][1]
            '''
            turnaround_time = completion_time - arrival_time
            '''
            total_turnaround_time = total_turnaround_time + t
urnaround_time
            process_data[i].append(turnaround_time)
        average_turnaround_time = total_turnaround_time / len
(process_data)
        '''
        average_turnaround_time = total_turnaround_time / no_
of_processes
        '''
        return average_turnaround_time

    def calculateWaitingTime(self, process_data):
        total_waiting_time = 0
        for i in range(len(process_data)):
            waiting_time = process_data[i][4] - process_data[
i][2]
            '''
            waiting_time = turnaround_time - burst_time
            '''
            total_waiting_time = total_waiting_time + waiting
_time
            process_data[i].append(waiting_time)
        average_waiting_time = total_waiting_time / len(proce
ss_data)
        '''
        average_waiting_time = total_waiting_time / no_of_pro
cesses
        '''
        return average_waiting_time
```

```
    def printData(self, process_data, average_turnaround_time
, average_waiting_time):
        print(f'Prosjecno Turnaround Time: {average_turnaroun
d_time}')
        print(f'Prosjecno Waiting Time: {average_waiting_time
}')
```

Programski kod za algoritam *Shortest Job First*:

```
class SJF:

    def processData(self, no_of_processes):
        process_data = []
        for i in range(no_of_processes):
            temporary = []
            process_id = int(input("Unesite ID procesa: "))
            arrival_time = int(input(f"Unesite Arrival Time z
a Proces {process_id}: "))
            burst_time = int(input(f"Unesite Burst Time za Pr
oces {process_id}: "))
            temporary.extend([process_id, arrival_time, burst
_time, 0, burst_time])
            '''
            '0' is the state of the process. 0 means not exec
uted and 1 means execution complete
            '''
            process_data.append(temporary)
        SJF.schedulingProcess(self, process_data)

    def schedulingProcess(self, process_data):
        start_time = []
        exit_time = []
        s_time = 0
        sequence_of_process = []
        process_data.sort(key=lambda x: x[1])
        '''
        Sort processes according to the Arrival Time
        '''
        while 1:
            ready_queue = []
```

```
            normal_queue = []
            temp = []
            for i in range(len(process_data)):
                if process_data[i][1] <= s_time and process_d
ata[i][3] == 0:
                    temp.extend([process_data[i][0], process_
data[i][1], process_data[i][2], process_data[i][4]])
                    ready_queue.append(temp)
                    temp = []
                elif process_data[i][3] == 0:
                    temp.extend([process_data[i][0], process_
data[i][1], process_data[i][2], process_data[i][4]])
                    normal_queue.append(temp)
                    temp = []
            if len(ready_queue) == 0 and len(normal_queue) ==
 0:
                break
            if len(ready_queue) != 0:
                ready_queue.sort(key=lambda x: x[2])
                '''
                Sort processes according to Burst Time
                '''
                start_time.append(s_time)
                s_time = s_time + 1
                e_time = s_time
                exit_time.append(e_time)
                sequence_of_process.append(ready_queue[0][0])
                for k in range(len(process_data)):
                    if process_data[k][0] == ready_queue[0][0
]:
                        break
                process_data[k][2] = process_data[k][2] - 1
                if process_data[k][2] == 0:        #If Burst
Time of a process is 0, it means the process is completed
                    process_data[k][3] = 1
                    process_data[k].append(e_time)
            if len(ready_queue) == 0:
                if s_time < normal_queue[0][1]:
                    s_time = normal_queue[0][1]
                start_time.append(s_time)
                s_time = s_time + 1
```

```
                e_time = s_time
                exit_time.append(e_time)
                sequence_of_process.append(normal_queue[0][0]
)
                for k in range(len(process_data)):
                    if process_data[k][0] == normal_queue[0][
0]:
                        break
                process_data[k][2] = process_data[k][2] - 1
                if process_data[k][2] == 0:        #If Burst
Time of a process is 0, it means the process is completed
                    process_data[k][3] = 1
                    process_data[k].append(e_time)
        t_time = SJF.calculateTurnaroundTime(self, process_da
ta)
        w_time = SJF.calculateWaitingTime(self, process_data)
        SJF.printData(self, process_data, t_time, w_time, seq
uence_of_process)

    def calculateTurnaroundTime(self, process_data):
        total_turnaround_time = 0
        for i in range(len(process_data)):
            turnaround_time = process_data[i][5] - process_da
ta[i][1]
            '''
            turnaround_time = completion_time - arrival_time
            '''
            total_turnaround_time = total_turnaround_time + t
urnaround_time
            process_data[i].append(turnaround_time)
        average_turnaround_time = total_turnaround_time / len
(process_data)
        '''
        average_turnaround_time = total_turnaround_time / no_
of_processes
        '''
        return average_turnaround_time

    def calculateWaitingTime(self, process_data):
        total_waiting_time = 0
        for i in range(len(process_data)):
```

```
            waiting_time = process_data[i][6] - process_data[
i][4]
            '''
            waiting_time = turnaround_time - burst_time
            '''
            total_waiting_time = total_waiting_time + waiting
_time
            process_data[i].append(waiting_time)
        average_waiting_time = total_waiting_time / len(proce
ss_data)
        '''
        average_waiting_time = total_waiting_time / no_of_pro
cesses
        '''
        return average_waiting_time

    def printData(self, process_data, average_turnaround_time
, average_waiting_time, sequence_of_process):
        process_data.sort(key=lambda x: x[0])
        '''
        Sort processes according to the Process ID
        '''

        print(f'Prosjecno Turnaround Time: {average_turnaroun
d_time}')
        print(f'Prosjecno Waiting Time: {average_waiting_time
}')
        print(f'Niz procesa: {sequence_of_process}')
```

Programski kod za algoritam *Round Robin*:

```
class RoundRobin:

    def processData(self, no_of_processes):
        process_data = []
        for i in range(no_of_processes):
            temporary = []
            process_id = int(input("Unesite ID procesa: "))
```

```
            arrival_time = int(input(f"Unesite Arrival Time z
a Proces {process_id}: "))

            burst_time = int(input(f"Unesite Burst Time za Pr
oces {process_id}: "))

            temporary.extend([process_id, arrival_time, burst
_time, 0, burst_time])
            '''
            '0' is the state of the process. 0 means not exec
uted and 1 means execution complete

            '''
            process_data.append(temporary)

        quantumTime = int(input("Unesite Quantum Time: "))

        RoundRobin.schedulingProcess(self, process_data, quan
tumTime)

    def schedulingProcess(self, process_data, quantumTime):
        start_time = []
        exit_time = []
        executed_process = []
        ready_queue = []
        s_time = 0
        process_data.sort(key=lambda x: x[1])
        '''
        Sort processes according to the Arrival Time
        '''
        while 1:
            normal_queue = []
            temp = []
            for i in range(len(process_data)):
                if process_data[i][1] <= s_time and process_d
ata[i][3] == 0:
                    present = 0
                    if len(ready_queue) != 0:
                        for k in range(len(ready_queue)):
                            if process_data[i][0] == ready_qu
eue[k][0]:
```

```
                                present = 1
                    '''
                    The above if loop checks that the next pr
ocess is not a part of ready_queue
                    '''
                    if present == 0:
                        temp.extend([process_data[i][0], proc
ess_data[i][1], process_data[i][2], process_data[i][4]])
                        ready_queue.append(temp)
                        temp = []
                    '''
                    The above if loop adds a process to the r
eady_queue only if it is not already present in it
                    '''
                    if len(ready_queue) != 0 and len(executed
_process) != 0:
                        for k in range(len(ready_queue)):
                            if ready_queue[k][0] == executed_
process[len(executed_process) - 1]:
                                ready_queue.insert((len(ready
_queue) - 1), ready_queue.pop(k))
                    '''
                    The above if loop makes sure that the rec
ently executed process is appended at the end of ready_queue
                    '''
                elif process_data[i][3] == 0:
                    temp.extend([process_data[i][0], process_
data[i][1], process_data[i][2], process_data[i][4]])
                    normal_queue.append(temp)
                    temp = []
            if len(ready_queue) == 0 and len(normal_queue) ==
 0:
                break
            if len(ready_queue) != 0:
                if ready_queue[0][2] > quantumTime:
                    '''
                    If process has remaining burst time great
er than the Quantum Time, it will execute for a time period e
qual to Quantum Time and then switch
                    '''
                    start_time.append(s_time)
```

```
                    s_time = s_time + quantumTime
                    e_time = s_time
                    exit_time.append(e_time)
                    executed_process.append(ready_queue[0][0]
)
                    for j in range(len(process_data)):
                        if process_data[j][0] == ready_queue[
0][0]:
                            break
                    process_data[j][2] = process_data[j][2] -
 quantumTime
                    ready_queue.pop(0)
                elif ready_queue[0][2] <= quantumTime:
                    '''
                    If a process has a remaining burst time l
ess than or equal to Quantum Time, it will complete its execu
tion
                    '''
                    start_time.append(s_time)
                    s_time = s_time + ready_queue[0][2]
                    e_time = s_time
                    exit_time.append(e_time)
                    executed_process.append(ready_queue[0][0]
)
                    for j in range(len(process_data)):
                        if process_data[j][0] == ready_queue[
0][0]:
                            break
                    process_data[j][2] = 0
                    process_data[j][3] = 1
                    process_data[j].append(e_time)
                    ready_queue.pop(0)
            elif len(ready_queue) == 0:
                if s_time < normal_queue[0][1]:
                    s_time = normal_queue[0][1]
                if normal_queue[0][2] > quantumTime:
                    '''
                    If process has remaining burst time great
er than the Quantum Time, it will execute for a time period e
qual to Quantum Time and then switch
                    '''
```

```
                    start_time.append(s_time)
                    s_time = s_time + quantumTime
                    e_time = s_time
                    exit_time.append(e_time)
                    executed_process.append(normal_queue[0][0
])
                    for j in range(len(process_data)):
                        if process_data[j][0] == normal_queue
[0][0]:
                            break
                    process_data[j][2] = process_data[j][2] -
 quantumTime
                elif normal_queue[0][2] <= quantumTime:
                    '''
                    If a process has a remaining burst time l
ess than or equal to Quantum Time, it will complete its execu
tion
                    '''
                    start_time.append(s_time)
                    s_time = s_time + normal_queue[0][2]
                    e_time = s_time
                    exit_time.append(e_time)
                    executed_process.append(normal_queue[0][0
])
                    for j in range(len(process_data)):
                        if process_data[j][0] == normal_queue
[0][0]:
                            break
                    process_data[j][2] = 0
                    process_data[j][3] = 1
                    process_data[j].append(e_time)
        t_time = RoundRobin.calculateTurnaroundTime(self, pro
cess_data)
        w_time = RoundRobin.calculateWaitingTime(self, proces
s_data)
        RoundRobin.printData(self, process_data, t_time, w_ti
me, executed_process)

    def calculateTurnaroundTime(self, process_data):
        total_turnaround_time = 0
        for i in range(len(process_data)):
```

```
            turnaround_time = process_data[i][5] - process_da
ta[i][1]
            '''
            turnaround_time = completion_time - arrival_time
            '''
            total_turnaround_time = total_turnaround_time + t
urnaround_time
            process_data[i].append(turnaround_time)
        average_turnaround_time = total_turnaround_time / len
(process_data)
        '''
        average_turnaround_time = total_turnaround_time / no_
of_processes
        '''
        return average_turnaround_time

    def calculateWaitingTime(self, process_data):
        total_waiting_time = 0
        for i in range(len(process_data)):
            waiting_time = process_data[i][6] - process_data[
i][4]
            '''
            waiting_time = turnaround_time - burst_time
            '''
            total_waiting_time = total_waiting_time + waiting
_time
            process_data[i].append(waiting_time)
        average_waiting_time = total_waiting_time / len(proce
ss_data)
        '''
        average_waiting_time = total_waiting_time / no_of_pro
cesses
        '''
        return average_waiting_time

    def printData(self, process_data, average_turnaround_time
, average_waiting_time, executed_process):
        process_data.sort(key=lambda x: x[0])

        print(f'Prosjecno Turnaround Time: {average_turnaroun
d_time}')
```

```
        print(f'Prosjecno Waiting Time: {average_waiting_time
}')

        print(f'Niz procesa: {executed_process}')
```

Programski kod za algoritam *Priority Scheduling*:

```
class Priority:

    def processData(self, no_of_processes):
        process_data = []
        for i in range(no_of_processes):
            temporary = []
            process_id = int(input("Unesite ID procesa: "))

            arrival_time = int(input(f"Unesite Arrival Time z
a Proces {process_id}: "))

            burst_time = int(input(f"Unesite Burst Time za Pr
oces {process_id}: "))

            priority = int(input(f"Unesite Priority za Proces
 {process_id}: "))

            temporary.extend([process_id, arrival_time, burst
_time, priority, 0, burst_time])
            '''
            '0' is the state of the process. 0 means not exec
uted and 1 means execution complete
            '''
            process_data.append(temporary)
        Priority.schedulingProcess(self, process_data)

    def schedulingProcess(self, process_data):
        start_time = []
        exit_time = []
        s_time = 0
        sequence_of_process = []
        process_data.sort(key=lambda x: x[1])
```

```
        '''
        Sort processes according to the Arrival Time
        '''
        while 1:
            ready_queue = []
            normal_queue = []
            temp = []
            for i in range(len(process_data)):
                if process_data[i][1] <= s_time and process_d
ata[i][4] == 0:
                    temp.extend([process_data[i][0], process_
data[i][1], process_data[i][2], process_data[i][3],
                                 process_data[i][5]])
                    ready_queue.append(temp)
                    temp = []
                elif process_data[i][4] == 0:
                    temp.extend([process_data[i][0], process_
data[i][1], process_data[i][2], process_data[i][4],
                                 process_data[i][5]])
                    normal_queue.append(temp)
                    temp = []
            if len(ready_queue) == 0 and len(normal_queue) ==
 0:
                break
            if len(ready_queue) != 0:
                ready_queue.sort(key=lambda x: x[3], reverse=
True)
                start_time.append(s_time)
                s_time = s_time + 1
                e_time = s_time
                exit_time.append(e_time)
                sequence_of_process.append(ready_queue[0][0])
                for k in range(len(process_data)):
                    if process_data[k][0] == ready_queue[0][0
]:
                        break
                process_data[k][2] = process_data[k][2] - 1
                if process_data[k][2] == 0:       #if burst t
ime is zero, it means process is completed
                    process_data[k][4] = 1
                    process_data[k].append(e_time)
```

```
            if len(ready_queue) == 0:
                normal_queue.sort(key=lambda x: x[1])
                if s_time < normal_queue[0][1]:
                    s_time = normal_queue[0][1]
                start_time.append(s_time)
                s_time = s_time + 1
                e_time = s_time
                exit_time.append(e_time)
                sequence_of_process.append(normal_queue[0][0]
)
                for k in range(len(process_data)):
                    if process_data[k][0] == normal_queue[0][
0]:
                        break
                process_data[k][2] = process_data[k][2] - 1
                if process_data[k][2] == 0:        #if burst
time is zero, it means process is completed
                    process_data[k][4] = 1
                    process_data[k].append(e_time)
        t_time = Priority.calculateTurnaroundTime(self, proce
ss_data)
        w_time = Priority.calculateWaitingTime(self, process_
data)
        Priority.printData(self, process_data, t_time, w_time
, sequence_of_process)

    def calculateTurnaroundTime(self, process_data):
        total_turnaround_time = 0
        for i in range(len(process_data)):
            turnaround_time = process_data[i][6] - process_da
ta[i][5]
            '''
            turnaround_time = completion_time - arrival_time
            '''
            total_turnaround_time = total_turnaround_time + t
urnaround_time
            process_data[i].append(turnaround_time)
        average_turnaround_time = total_turnaround_time / len
(process_data)
        '''
```

```
        average_turnaround_time = total_turnaround_time / no_
of_processes
        '''
        return average_turnaround_time

    def calculateWaitingTime(self, process_data):
        total_waiting_time = 0
        for i in range(len(process_data)):
            waiting_time = process_data[i][6] - process_data[
i][2]
            '''
            waiting_time = turnaround_time - burst_time
            '''
            total_waiting_time = total_waiting_time + waiting
_time
            process_data[i].append(waiting_time)
        average_waiting_time = total_waiting_time / len(proce
ss_data)
        '''
        average_waiting_time = total_waiting_time / no_of_pro
cesses
        '''
        return average_waiting_time

    def printData(self, process_data, average_turnaround_time
, average_waiting_time, sequence_of_process):
        process_data.sort(key=lambda x: x[0])
        print(f'Prosjecno Turnaround Time: {average_turnaroun
d_time}')
        print(f'Prosjecno Waiting Time: {average_waiting_time
}')
        print(f'Niz procesa: {sequence_of_process}')
```