Gimnazija “Vuk Karadžić”, Trstenik

PREDMET

Operativni Sistemi i Računarske Mreže

MATURSKI RAD

Implementacija First Come First Served, Shortest Job First i Longest Job First planera procesa u programskom jeziku C

Profesor: Dejan Končar

Učenik: Dušan Trišić

Maj, 2024.

SADRŽAJ

1. Uvod

Operativni sistemi su temelj modernih računarskih sistema, a planiranje procesa predstavlja ključnu funkciju unutar njihovog delovanja. Planiranje procesa je vitalni mehanizam operativnog sistema koji određuje način raspoređivanja i izvršavanja procesa na računaru. Sa sve kompleksnijim zahtevima modernih aplikacija i tehnološkim napretkom, efikasno upravljanje procesima postaje sve važnije kako bi se osigurala optimalna iskorišćenost resursa sistema i brza reakcija na korisničke zahteve. U ovom maturskom radu ću istraživati planiranje procesa u operativnim sistemima, analizirati različite strategije planiranja kao i algoritme koji se koriste za raspoređivanje procesa. Takođe, pričaću o uticaju planiranja procesa na performanse sistema, kao i izazovima s kojima se operativni sistemi suočavaju u efikasnom upravljanju procesima. Kroz sve navedene aspekte, cilj ovog rada je pružiti dublje razumijevanje planiranja procesa u operativnim sistemima i njegovog uticaja na performanse i stabilnost računarskih sistema u savremenom digitalnom okruženju.

2. Procesi

Program koji se izvršava naziva se proces. Da bi se izvršavao, procesu će biti potrebni određeni resursi - poput CPU-a, memorije, datoteka i U/I uređaja - da bi obavio svoj zadatak. Ovi resursi se dodeljuju procesu ili kada se proces kreira ili tokom njegovog izvršavanja.

Sistemi se sastoje od skupa procesa: procesi operativnog sistema izvršavaju sistemski kod, a korisnički procesi izvršavaju korisnički kod. Svi se ti procesi mogu istovremeno izvršavati.

Operativni sistem je odgovoran za nekoliko važnih aspekata upravljanje procesima:

- stvaranje i brisanje procesa
- raspoređivanje procesa
- obezbeđivanje mehanizama za sinhronizaciju, komunikaciju i razrešavanje dead lock-a za procese

3. Planeri procesa

Primitivni operativni sistemi rade na samo jednom jezgru - samo jedan proces se izvrsava u svakom trenutku. Algoritmi koji kontrolisu redosled i resurse (procesorsko vreme, memorije, ...) se zovu planeri procesa.

Postoji nekoliko različitih algoritama planiranja procesa, koji određuju redosled izvršavanja procesa i raspodelu resursa. Neke od najčešćih vrsta algoritama planiranja uključuju FCFS (First-Come, First-Served), SJF (Shortest Job First), Round Robin, Priority Scheduling itd. Svaki od ovih algoritama ima svoje prednosti i mane, i odabir odgovarajućeg algoritma zavisi od specifičnih zahteva sistema i prioriteta.

Algoritmi koje sam ja implementirao su jako prosti i spadaju u statičke planere procesa (prioritet se ne menja tokom izvršavanja).

Prvi i najprostiji algoritam za planiranje procesa jeste First Come First Served (FCFS) planer. Ovaj planer raspoređuje procese na osnovu vremena njihovog dolaska - prvi proces koji zatraži procesor će ga i dobiti, dok će svi ostali čekati dok se taj proces ne završi. Implementacija FCFS algoritma je relativno jednostavna jer koristi strukturu reda, gde se procesi dodaju na kraj reda kada stignu, a uklanjaju sa početka reda kada dobiju procesor i završe svoje izvršavanje. Ovaj pristup ne zahteva kompleksne algoritme ili dodatnu logiku za određivanje prioriteta ili preusmeravanje procesa, što ga čini lako razumljivim i jednostavnim za implementaciju.

Međutim, iako FCFS algoritam ima svoje prednosti, kao što je nemogućnost deadlock-a (situacije u kojoj se procesi međusobno blokiraju čekajući resurse koje drže drugi procesi), on nije mnogo efikasan. Jedan od glavnih problema ovog algoritma je tzv. "konvoj efekat" (convoy effect), gde duži procesi mogu značajno uticati na vreme čekanja kraćih procesa. Na primer, ako neki proces koji traje veoma dugo zatraži procesor, on će blokirati sve ostale procese koji su stigli nakon njega, što može dovesti do velikih kašnjenja i neefikasnog korišćenja procesora.

Još jedan problem FCFS algoritma je nemogućnost prilagođavanja prioritetima procesa. Svi procesi se tretiraju jednako bez obzira na njihovu važnost ili vremenske zahteve, što može biti nepovoljno u sistemima gde su neki procesi kritični i zahtevaju brže izvršenje. Također, ukoliko proces koji traje "večno" ili vrlo dugo zatraži procesor, on će monopolizirati resurse sistema, uzrokujući da svi ostali procesi čekaju neodređeno dugo. Ovo može biti posebno problematično u interaktivnim sistemima gde korisnici očekuju brzi odziv na svoje zahteve.

Iako FCFS planer može biti adekvatan za jednostavne i predvidive operativne sisteme, njegova upotreba u složenijim ili visoko interaktivnim sistemima može dovesti do značajnih performansnih problema. Zbog ovih nedostataka, često se koriste napredniji algoritmi za planiranje procesa koji mogu dinamičnije raspoređivati resurse i bolje upravljati različitim potrebama procesa unutar sistema.

Naredni algoritam za planiranje procesa je malo kompleksniji, ali je još uvek prilično primitivan i jednostavan. Ovaj algoritam se naziva Shortest Job First (SJF) i raspoređuje procese na osnovu njihovog trajanja. Osnovna ideja iza SJF algoritma je da se procesi koji zahtevaju najkraće vreme izvršavanja obrađuju prvi, dok ostali čekaju na svoj red. Da bi SJF algoritam funkcionisao, potrebno je unapred znati koliko dugo svaki proces zauzima procesor u najgorem slučaju, što može biti izazov u stvarnim scenarijima gde je teško precizno predvideti vreme izvršavanja svakog procesa.

SJF algoritam može biti vrlo efikasan u smanjenju prosečnog vremena čekanja procesa, jer obrada kraćih procesa može značajno smanjiti ukupno vreme koje procesi provode čekajući na izvršenje. Na primer, ako imamo niz procesa sa vremenima izvršavanja 2, 4, 1 i 3 jedinice vremena, SJF će prvo obraditi proces koji traje 1 jedinicu, zatim 2 jedinice, pa 3, i na kraju proces koji traje 4 jedinice vremena. Ovaj pristup može optimizovati ukupnu efikasnost korišćenja procesora i smanjiti prosečno vrijeme čekanja svih procesa.

Međutim, SJF algoritam ima i svoje nedostatke. Jedan od glavnih problema je potreba za poznavanjem tačnog vremena izvršavanja procesa unapred, što nije uvijek moguće ili praktično u stvarnim sistemima. Još jedan značajan problem je tzv. "problem gladovanja" (starvation), gde dugotrajni procesi mogu biti stalno odlagani ako stalno dolaze kraći procesi. To može rezultirati situacijom u kojoj dugotrajni procesi nikada ne dođu na red za izvršenje, jer su uvijek preteknuti kraćim procesima.

Ovaj problem može biti posebno izražen u sistemima gde postoji velika količina kratkih procesa. Ako sistem stalno prima nove kratke procese, dugotrajni procesi će neprestano biti odgađani, što može dovesti do ozbiljnih performansnih problema i nepredvidljivog ponašanja sistema. Da bi se ublažio ovaj problem, ponekad se koristi varijanta SJF algoritma poznata kao Shortest Remaining Time First (SRTF), koja omogućava preemptivno raspoređivanje procesa, tj. prekidanje trenutno izvršavanog procesa ako stigne novi proces sa kraćim vremenom izvršavanja.

Uprkos svojim nedostacima, SJF algoritam može biti vrlo efikasan u specifičnim situacijama gde je moguće precizno predvideti vreme izvršavanja procesa i gde postoji relativno stabilan broj kratkih i dugotrajnih procesa. U praksi, često se koristi zajedno sa drugim algoritmima kako bi se postigla ravnoteža između efikasnosti i pravičnosti u raspoređivanju procesa.

Poslednji algoritam, koji po mnogome liči na prethodni, jeste Longest Job First. Umesto da procese sortira od najkraćeg do najdužeg, on ih sortira obrnuto - od najdužeg ka najkraćem. Ima sličan problem kao i njegov sličnoimenik, samo što u ovom slučaju na red ne dolaze kratki procesi.

4. Implementacija

Izvorni kod celog projekta se može videti na linku - *“https://github.com/dusan3sic/maturski”*.

Implementacija procesa u programskom jeziku C. Svaki proces je instanca strukture Task, koji ima id (u nekim literaturama i pid) i execution_time - vreme potrebno za izvršavanje procesa.

```
typedef struct {
    int id;
    int execution_time;
} Task;
```

Procesi se čuvaju u jednoj povezanoj listi koja je standardno implementirana pomoću strukture Node i Scheduler.

```
typedef struct Node {
    Task task;
    struct Node *next;
} Node;

typedef struct {
    Node *front;
    Node *rear;
    int size;
} Scheduler;
```

Planer se inicijalizuje pomoću funkcije initScheduler koja postavlja pokazivače koji pokazuju na prvi, poslednji član i veličinu liste.

```
void initScheduler(Scheduler *scheduler) {
    scheduler->front = NULL;
    scheduler->rear = NULL;
    scheduler->size = 0;
}
```

Naredna funkcija - addTask() je drugačija u zavisnosti od algoritma. U FCFS algoritmu, ova funkcija samo dodaje proces na kraj linkovane liste, dok u druga dva algoritma ga ubacuje na mesto u zavisnosti od dužine trajanja procesa. Ubacivanje na “neko” mesto u linkovanoj listi se postiže postavljanjem pointera koji pokazuje na sledeći član u listi. Pošto u druga dva slučaja moramo da pronađemo odgovarajući član, moramo da koristimo while petlju da bismo prošli kroz članove liste. Algoritam vrši i par provera pre pretrage i modifikacije da ne bi dolazilo do grešaka.

```
//FCFS
void addTask(Scheduler *scheduler, Task task) {
   Node *newNode = (Node*)malloc(sizeof(Node));
   if (newNode == NULL) {
       return;
   }
   newNode->task = task;
   newNode->next = NULL;

   if (scheduler->rear == NULL) {
       scheduler->front = newNode;
       scheduler->rear = newNode;
   } else {
       scheduler->rear->next = newNode;
       scheduler->rear = newNode;
   }
   scheduler->size++;
}
```

```
    //LJF, SJF je isti kao LJF samo je obrnut znak
    void addTask(Scheduler *scheduler, Task task) {
        Node *newNode = (Node*)malloc(sizeof(Node));
        if (newNode == NULL)         return;
        newNode->task = task;
        newNode->next = NULL;
        if (scheduler->rear == NULL) {
            scheduler->front = newNode;
            scheduler->rear = newNode;
        } else {
            Node *currentNode = scheduler->front;
            Node *previousNode = NULL;
             while (currentNode != NULL && currentNode->task.execution_time
< task.execution_time) {
                previousNode = currentNode;
                currentNode = currentNode->next;
            }
            if (previousNode == NULL) {
                newNode->next = scheduler->front;
                scheduler->front = newNode;
            } else {
                newNode->next = currentNode;
                previousNode->next = newNode;
            }
            if (newNode->next == NULL) {
                scheduler->rear = newNode;
            }
        }
        scheduler->size++;
    }
```

Funkcija getNextTask() je ista za sva tri algoritma, s obzirom da se sledeći proces uvek nalazi na početku povezane liste. Pre nego što se vrati trenutni proces, on se brise iz povezane liste, tako da sledeći po redu bude prvi.

```
Task getNextTask(Scheduler *scheduler) {
    if (scheduler->front == NULL) {
        Task emptyTask = {0, 0, 0};
        return emptyTask;
    }

    Node *frontNode = scheduler->front;
    Task nextTask = frontNode->task;

    scheduler->front = frontNode->next;
    free(frontNode);

    if (scheduler->front == NULL) {
        scheduler->rear = NULL;
    }

    scheduler->size--;
    return nextTask;
}
```

Postoji i par linux shell fajlova koji služe za pokretanje programa. Postoji pripremiZaBuild.sh skripta koja nam pomaže pri odabiru algoritma koji koristimo za planer. Postoji run.sh skripta koja pokrece bin fajl, koji je generisan pomoću komande make i konfiguracije koja se nalazi u Makefile-u.

5. Literatura

Operativni sistemi - "*Operating System Concepts" - Abraham Silberschatz, Peter B. Galvin, Greg Gagne - 2008*

OS-dev - *"Writing a Simple Operating System —from Scratch" - Nick Blundell - 2010*