Operacijski sustav

Kadak, Filip

Sveučilište Josipa Jurja Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Studij
Sveučilišni prijediplomski studij Matematika i računarstvo

Operacijski sustav

Završni rad

Mentor:

izv. prof. dr. sc.
Domagoj Matijević

Komentor:

Bartol Borozan

Student:

Filip Kadak

Osijek, 2024

Sadržaj

1 | Uvod

Kada pričamo o programiranju računala, najčešće pričamo o nekom od jezika visoke razine, time zanemarujemo mnoge detalje koje računalo izvodi u pozadini. Kako bi izvršili jednostavan zadatak poput ispisivanja "Hello world" poruke na zaslon, u pozadini se mora izvršiti niz zadataka. Moramo pronaći mjesto u memoriji računala za alociranje prostora za naš string, pratiti poziciju na ekranu gdje ispisujemo tekst i učitati neki font s kojim prikazujemo tekst. Ovdje već ima velik broj operacija koje su zahtjevne za izvesti. Ako naš program zahtjeva i neki ulaz od korisnika, trebamo uvesti i komunikaciju našeg programa sa hardverom poput tipkovnice za unos teskta.

Sa navedenim problemima i mnogim drugim koji se bave sa komunikacijom različitih dijelova računala, bavi se operacijski sustav. Operacijski sustav sadrži implementacije funkcija koje compiler ubacuje u naš kod i time omogućava korištenje abstraktnih funkcija poput print("Hello world") da mi ne moramo brinuti o njihovim implementacijama. Ove funkcije moraju biti dobro optimizirane i sigurne za korištenje. Za njihovu implementaciju koriste se razne strukture podataka i algoritmi kojima elegantno i efikasno možemo riješiti navedene probleme.

U ovom radu ćemo implementirati neke od najbitnijih funkcija operacijskog sustava na pojednostavljenom modelu računala poznatom kao Hack računalo [1]. Bavit ćemo se implementacijom jednostvnih matematičkih operacija, alociranjem i oslobađanjem memorije, crtanjem na ekran, strukturom podataka string za operacije nad tekstom, te komunikacijom s tipkovnicom za unos podataka.

2 | Matematika

Prva klasa koju ćemo implementirati jest Math klasa, klasa kojom ćemo obavljati jednostavne matematičke operacije poput množenja, dijeljenja i računanja korijena. Ove operacije se u računalu izvršavaju ogroman broj puta te je bitno implementirati ih efikasno. Cilj nam je implementirati algoritme koji ovise o broju znamenki, a ne o njihovoj vrijednosti, jer vrijednost može rasti eksponencijalno. Primjer ovoga ćemo vidjeti u funkciji množenja. Ove operacije mogu se implementirati u hardveru ili softveru. Zbrajanje, a time i oduzimanje, su najčešće implementirani u hardveru, njih nećemo ovdje pokriti. O hardverskoj implementaciji drugih operacija, poput množenja, odlučuje proizvođač ovisno o cijeni i dobitku na brzini izvršavanja u usporedbi sa softverskom implementacijom.

2.1 Množenje

Naivna implementacija množenja bila bi niz zbrajanja u petlji:

```
for i = 1 ... y sum = sum + x
```

Kompleksnost ovog algoritma je $O(n)$ gdje je n vrijednost varijable y, ako pretpostavimo da y može biti 64-bitni broj, čak i sa milijunima operacija na sekundu, jedan poziv ovakvog množenja trebao bi stotine godina da se izvrši. Stoga, trebamo pronaći algoritam koji ne ovisi o vrijednosti brojeva koje množimo. Algoritam koji ćemo opisati i implementirati ovisi o broju znamenki binarnog zapisa broja. U gornjem primjeru, to bi bilo upravo 64. Kompleksnost algoritma je ponovno $O(n)$, no sa puno manjom maksimalnom vrijednosti broja n. Algoritam je baziran na činjenici da vrijedi sljedeća jednakost:

$$123 * 456 = 100 * 456 + 20 * 456 + 3 * 456.$$

U binarnom zapisu, ovo se nadalje pojednostavljuje jer množimo samo sa 0 ili 1, te je cijeli algoritam samo niz zbrajanja i pomicanja.

```
      1011     (binarni zapis broja 11)
    × 1110     (binarni zapis broja 14)
    ======
      0000     (1011 × 0)
     1011      (1011 × 1, pomaknuto jednu poziciju ulijevo)
    1011       (1011 × 1, pomaknuto dvije pozicije ulijevo)
+ 1011         (1011 × 1, pomaknuto tri pozicije ulijevo)
=========
 10011010      (binarni zapis broja 154)
```

Algoritam 1 Algoritam za množenje

```
1: function MULTIPLY(x, y)
2:     sum = 0
3:     shiftedx = x
4:     for i = 0 to n − 1 do
5:         if (i-th bit of y == 1) then
6:             sum = sum + shiftedx
7:         shiftedx = shiftedx * 2
8:     return sum
```

Naknadno, redak

```
shiftedx = shiftedx * 2;
```

Možemo zapisati kao operaciju pomaka ulijevo ili zbrajanjem broja sobom samim. Obje operacije su implementirane hardverski te bi bile efikasnije nego pomicanje pomoću množenja.

2.2 Dijeljenje

Algoritam za dijeljenje također ima vrijeme izvršavanja $O(n)$ gdje je n broj znamenki binarnog zapisa broja koji dijelimo. Baziran je na sljedećem uvidu:

$$480/17 = 2 * (480/34) = 2 * (2 * (480 * 68)) = ...$$

U svakom koraku, pomnožit ćemo dijeljitelja sa 2 dok god je on manji od dijeljenika. Maksimalan broj puta koliko ovo možemo ponoviti je upravo broj znamenki dijeljenika u binarnom zapisu.

Algoritam 2 Algoritam za dijeljenje

```
1: function DIVIDE(x, y)
2:     if y > x then
3:         return 0
4:     q = DIVIDE(x, y + y)
5:     if (x − (2 × q × y)) < y then
6:         return q + q
7:     else
8:         return q + q + 1
```

Kako bi dani algoritam radio i za negativne brojeve, moramo ga malo modificirati. Jednostavno, moramo pamtiti predznake brojeva prije ulaska u rekurziju te nastaviti sa apsolutnim vrijednostima. Ovu informaciju možemo spremiti u boolean varijable, te na kraju rekurzije provjeriti njihove vrijednosti i vratiti rezultat ili promijeniti predznak rezultata.

2.3 Korijen

Kako na modelu računala na kojem radimo nemamo floating point brojeve, implementiramo jednostavan algoritam za pronalazak cjelobrojnog korijena. Ovaj algoritam koristi ranije implementirano množenje za računanje kvadrata broja. U Hack računalu radimo sa 16-bitnim brojevima pa je zato broj n u Algoritmu 3 jednak 16, dakle krećemo od 2^7 pa uspoređujemo sa brojem čiji korijen tražimo, ako je kvadrat broja manji od traženog broja, dodamo ga, u suprotnom nastavimo na idući. Ovo ponavljamo dok ne dođemo do 2^0. Vrijeme izvršavanja ovog algoritma je također $O(n)$, gdje je n broj znamenki u binarnom zapisu.

Algoritam 3 Algoritam za računanje korijena

```
1: function SQRT(x)
2:     for i = (n/2 − 1) to 0 do
3:         if (y + 2^i)^2 ≤ x then
4:             y ← y + 2^i
5:     return y
```

3 | Unos i prikaz podataka

Bitan zadatak operacijskog sustava je komunikacija ulaznih i izlaznih uređaja sa računalom. Kako bi koristili računalo, moramo mu nekako slati svoje zahtjeve, te dobiti neku povratnu informaciju ili rezultat natrag. Kao primjer ove komunikacije, implementirat ćemo klasu za tipkovnicu koja će učitavati unos, te klase potrebne za prikaz tog unosa na ekranu. Potrebno nam je imati klasu koja će primati i obrađivati tekst, klasu za osnovne mogućnosti grafike i iscrtavanja oblika na ekranu, te konačno, klasu koja će ih ujediniti i omogućiti nam unos i prikaz teksta, simbola i brojeva.

3.1 Keyboard klasa

Način na koji ćemo implementirati tipkovnicu je da označimo adresu u memoriji koja će sadržavati trenutno pritisnutu tipku. Kada pročitamo vrijednost te adrese u memoriji, ona će biti upravo ASCII vrijednost trenutno pritisnute tipke. Ako nijedna tipka nije pritisnuta, biti će 0. Na nama je da minimiziramo ili uklonimo nesigurnosti koje se mogu pojaviti kod unosa podataka. Ako očekujemo ulaz koji se sastoji od više znakova, korisnik bi trebao moći slobodno brisati i dodavati nove znakove prije nego potvrdi svoj unos. Također, ne možemo znati koliko će korisnik dugo držati tipku pritisnutu niti koliko vremena je potrebno između unosa dva znaka. Stoga ćemo biti obzirni pri implementaciji potrebnih funkcija.

3.1.1 Unos znaka

Kada očekujemo unos znaka, ulazimo u petlju. Kao njen uvjet, čekamo da adresa u memoriji na koju smo postavili tipkovnicu sadrži vrijednost različitu od 0. Čim se vrijednost promijeni, spremimo ju. Zatim ulazimo u novu petlju koja blokira nastavak izvršavanja dok korisnik ne pusti tipku.

```
1 while (Keyboard.keyPressed() = 0) {}
2 let key = Keyboard.keyPressed();
3 while (~(Keyboard.keyPressed() = 0)) {}
```

3.1.2 Unos niza znakova

Kako bi postigli unos od više znakova, ponavljamo ovu funkciju dok korisnik ne potvrdi unos. U našem slučaju, to se dogodi kada je pritisnuta tipka enter, to jest, kada zaprimimo znak za prelazak u novu liniju. Također, ovdje ćemo se brinuti da korisnik može brisati znakove ukoliko dođe do pritiska krive tipke.

```
1 while (~(key = String.newLine())) {
2   let key = Keyboard.readChar();
3
4   if (~(key = String.newLine())) {
5     if (key = String.backSpace()) {
6       do str.eraseLastChar();
7     }
8     else {
9       do str.appendChar(key);
10    }
11  }
12 }
```

3.2 String klasa

Trebat će nam jednostavna klasa za obrađivanje teksta. U nju ćemo spremati ulaz sa tipkovnice te dalje ga uređivati po potrebi. Naša klasa će sadržavati niz u koji ćemo dodavati znakove, kapacitet niza te trenutnu duljinu teksta sadržanog u nizu.

3.2.1 Brisanje i dodavanje znakova

Funkcija brisanja znaka iz niza će jednostavno smanjiti trenutnu duljinu teksta za 1. Znakove u nizu koji se nalaze nakon trenutne duljine nećemo smatrati dijelom teksta.

Dodavanje znaka će prvo provjeriti je li duljina manja od maksimalnog kapaciteta. Ukoliko je, povećamo duljinu i dodamo ga na kraj niza. Ukoliko je niz pun, imamo izbor kako ćemo to riješiti, najjednostavniji pristup je jednostavno vratiti grešku ili ništa ne napraviti. Bolji pristup bi bio povećati kapacitet niza pa dodati znak.

3.2.2 Pretvaranje brojeva u tekst i obratno

Kako je ulaz sa tipkovnice primljen u obliku teksta, implementirat ćemo jednostavne algoritme za pretvaranje teksta u broj i obrnuto.

Algoritam 4 Pretvaranje brojeva u tekst

```
1: function INT_TO_STRING(val)
2:     last_digit ← val mod 10
3:     c ← char that represents last_digit
4:     if val < 10 then
5:         return c
6:     else
7:         return INT_TO_STRING(val / 10).append_char(c)
```

Pretvaranje znamenke u znak je jednostavno, na vrijednost znamenke dodamo vrijednost znamenke 0 iz ASCII tablice. Dodatno, trebamo rukovati sa negativnim brojevima. Ukoliko je vrijednost val manja od nule, prvo dodajemo znak '-' u naš niz, zatim nastavimo pretvorbu sa apsolutnom vrijednosti broja.

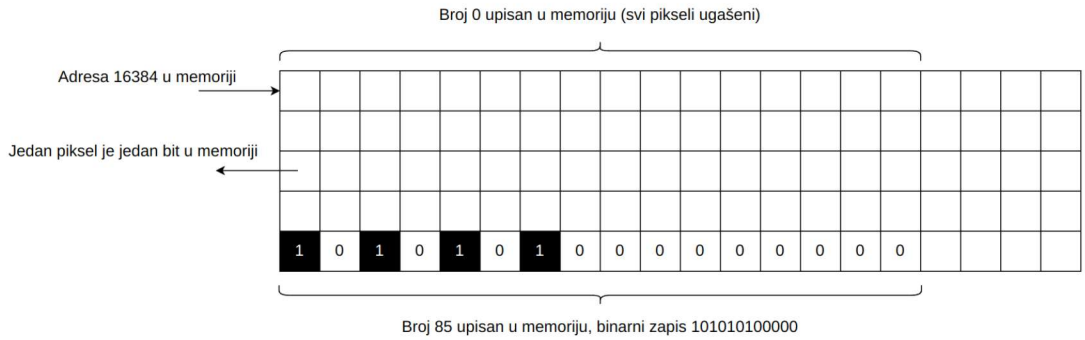
Algoritam 5 Pretvaranje teksta u broj

```
1: function STRING_TO_INT(str)
2:     val ← 0
3:     for i = 0 to str.length() − 1 do
4:         d ← value of digit at str[i]
5:         val ← val × 10 + d
6:     return val
```

Ponovno moramo uvesti poseban slučaj za negativne brojeve. Ukoliko je prvi znak '-', petlja će krenuti od drugog znaka te na kraju pretvorimo vrijednost val u negativan broj.

3.3 Screen klasa

Označimo dio memorije kao naš prikaz. Koristit ćemo adresu 16384 kao početak našeg zaslona. On će biti 512 piksela širok i 256 piksela visok. Ako na neku od adresa koje smo dodijelili zaslonu upišemo broj, vidjet ćemo njegov binarni zapis u obliku crnih i bijelih točaka na tom dijelu zaslona.



Slika 3.1: Zaslon u memoriji

3.3.1 Bojanje piksela

Prvo ćemo definirati funkciju koja će obojati jedan bit kako bi korisniku crtanje na ekran bilo intuitivno, ova funkcija će primati x i y koordinate piksela. Točka $(0, 0)$ biti će gornji lijevi kut zaslona, dok će točka $(511, 255)$ biti donji desni kut zaslona. Da bi uključili ovaj bit, moramo pronaći tu adresu u memoriji to radimo formulom.

$$address = (y * 32) + (x/16);$$

Kako zaslon gledamo kao niz brojeva, a definirali smo veličinu kao 512x256, da pronađemo redak, moramo množiti sa brojem 32. On predstavlja koliko 16-bitnih brojeva stane u red dug 512 bitova. Nadalje, kako se red sastoji od 16 bitnih brojeva, stupac pronalazimo dijeljenjem x koordinate sa 16.

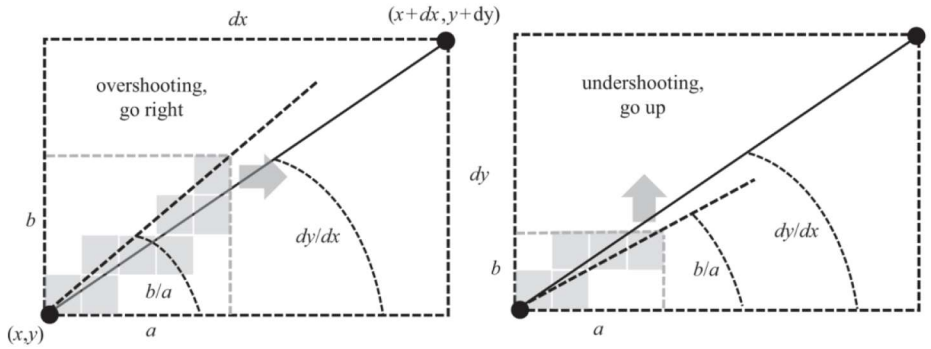
Kada smo pronašli broj koji sadrži piksel koji želimo uključiti, moramo odrediti koji bit u njemu želimo promijeniti. Računamo $x\%16$, ovaj broj će nam upravo dati redni broj bita kojeg tražimo. Da bi uključili ili isključili neki bit, prvo uzimamo broj koji ima nule na svim pozicijama osim tražene, to će biti $2^{x\%16}$

Ako želimo postaviti taj bit na 1, odnosno crnu boju, koristimo bitwise ili operaciju. Ona će uključiti bit bez da mijenja vrijednost ostalih bitova. Ukoliko želimo postaviti bit na 0, prvo koristimo bitwise ne operaciju da dobijemo inverznu vrijednost, a zatim bitwise i operaciju da ugasimo bit bez promijene ostalih bitova na toj adresi.

Funkcijom bojanja piksela dalje možemo definirati kompleksnije grafičke funkcije. Ovdje ćemo opisati funkcije za crtanje linija od točke a do točke b, te ćemo pomoću nje implementirati crtanje kvadrata i kruga.

3.3.2 Crtanje linije

Za crtanje linije, koristit ćemo petlju koja u svakoj svojoj iteraciji računa kut od početne točke do trenutne točke. Ovaj kut ćemo usporediti sa kutem od početne točke do krajnje točke i rezultatom odrediti hoće li iduča točka biti gore, dolje, lijevo ili desno od trenutne.



Slika 3.2: Prikaz algoritma za crtanja linije, [1]

Pri implementaciji, da bi ubrzali izvršavanje algoritma, razliku ćemo ažurirati svakim prolazom kroz petlju i time svesti uvjet na jedno zbrajanje ili oduzimanje, umjesto dijeljenja.

```
while (~(Math.abs(a) > dx) & ~(Math.abs(b) > dy)) {
  do Screen.drawPixel(x1 + a, y1 + b);
  if (diff < 0) {
    if (x2 < x1) {
      let a = a - 1;
    }
    else {
      let a = a + 1;
    }
    let diff = diff + dy;
  }
  else {
    if (y2 < y1) {
      let b = b - 1;
    }
    else {
      let b = b + 1;
    }
  let diff = diff - dx;
  }
}
```

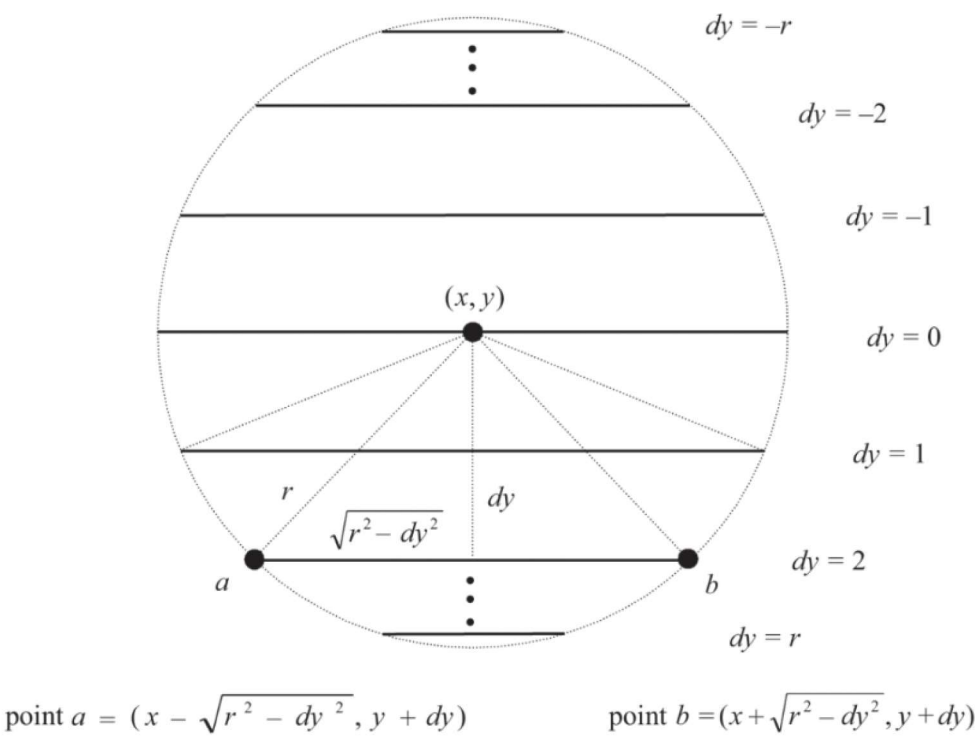
Naknadno, primijetimo poseban slučaj crtanja uspravnih i vodoravnih crta. Ukoliko je x ili y koordinata početne i krajnje točke jednaka, možemo koristiti posebnu funkciju koja je u tom slučaju znatno brža. Ova optimizacija će nam pomoći pri crtanju kvadrata i kruga.

3.3.3 Crtanje naprednijih oblika

Prilikom crtanja kvadrata, želimo crtati niz okomitih linija te pomicati se za jedan stupac u svakom prolazu.

```
let a = 0;
while (~((x1 + a) > x2)) {
  do Screen.drawLine((x1 + a), y1, (x1 + a), y2);
  let a = a + 1;
}
```

Pri crtanju kruga, kao argumente primamo središte i polumjer kružnice. Tada ulazimo u petlju koja će se ponavljati 2r puta. U svakom prolazu spustit ćemo se za jedan redak te izračunati širinu kruga u tom retku pomoću pravokutnog trokuta koji spaja središte, rub kružnice i udaljenost od središta do trenutnog retka.



Slika 3.3: Prikaz algoritma za crtanja kruga, [1]

```
let dy = -r;
let a = 0;
while (~(dy > r)) {
  let root = Math.sqrt((r * r) - (dy * dy));
  do Screen.drawLine(x - root, y + dy, x + root, y + dy);
  let dy = dy + 1;
}
```

3.4 Output klasa

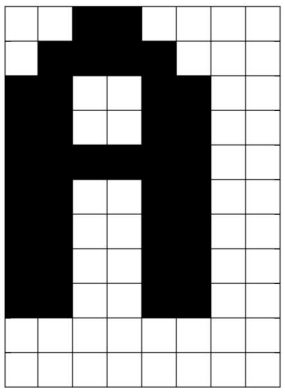
Sada imamo funkcije za primanje i obradu teksta te osnove grafičkog prikaza. U ovoj klasi ćemo ih objediniti i, uz definiranu mapu znakova koja će predstavljati naš font, omogućit ćemo ispis znakova na ekran. Za početak ćemo odlučiti neka pravila za naš ispis znakova. Da bi pojednostavili zadatak, odredit ćemo da je svaki znak ispisan unutar bloka širine 8 piksela i visine 11 piksela. Nadalje, želimo imati barem jedan stupac i barem jedan redak potpuno praznim kao razmak između znakova pri ispisu.

Kada smo dizajnirali svoj znak, spremit ćemo ga u sljedećem obliku: za svaki znak u svom fontu, napravit ćemo niz od 12 brojeva. Ovaj niz će na prvom mjestu imati ASCII vrijednost tog znaka te 11 brojeva zapisanih u binarnom obliku sa bitom najmanje vrijednosti na lijevom mjestu. Ovih 12 brojeva zajedno čine naš željeni znak.

Kao primjer, pogledajmo slovo A. Ono ima ASCII vrijednost 65, te će naš niz za slovo A izgledati ovako:

```
65
12: 00110000
30: 01111000
51: 11001100
51: 11001100
63: 11111100
51: 11001100
51: 11001100
51: 11001100
51: 11001100
 0: 00000000
 0: 00000000
```

Kada ga nacrtamo na zaslon, izgledat će ovako:



Slika 3.4: Prikaz mape slova A

Za ispisivanje slova, nećemo primati poziciju već ćemo pratiti poziciju prethodno napisanog slova za izračunavanje pozicije idućeg, koristeći sljedeću formulu:

$$address = 16384 + (cursor\_y * 11 * 32) + (cursor\_x/2);$$

Za redak, množimo redak teksta sa širinom retka, a zatim sa visinom znakova. Za stupac, primjetimo da u jedan 16-bitni broj možemo upisati dva slova, dakle, podijelimo poziciju s 2 da dobijemo adresu. Da bi ispisali znak, moramo na izračunatu adresu upisati prvi broj iz naše mape znakova za njega. Naknadno, moramo pripaziti na to upisujemo li znak na parni ili neparni stupac. Naime, kako u jedan 16-bitni broj stanu dva znaka širine 8 bita, morat ćemo pomaknuti bitove iz naše mape udesno za 8, tj, pomnožiti trenutni broj sa 2^8.

```
1  while (i < 11) {
2    let row = map[i];
3
4    if (cursor_x & 1) {
5      let row = Memory.peek(address) | (row * 256);
6    }
7
8    do Memory.poke(address, row);
9
10   let address = address + 32;
11   let i = i + 1;
12 }
```

Na kraju pomičemo cursor za jedan znak udesno, ili ako smo na kraju retka, pomičemo ga na početak idućeg retka. Pripazimo na slučaj kada smo na zadnjem znaku u zadnjem retku našeg zaslona. Ovdje možemo jednostavno postaviti cursor ponovno na $(0, 0)$ ili implementirati scrolling funkciju.

4 | Upravljanje memorijom

Preostalo nam je implementirati sustav za upravljanje memorijom, te poneke sistemske funkcije. Upravljanje memorijom je bitan zadatak operacijskog sustava da bi sustav mogao efikasno dodjeljivati memoriju kada je to potrebno. Osim toga, implementirat ćemo funkcije za čekanje i zaustavljanje izvršavanja programa.

4.1 Sys klasa

Sys klasa se sastoji od jednostavnih ali bitnih funkcija za rad sustava.

Prva funkcija je init funkcija. Ona će se pozvati pri pokretanju i pozvat će init funckije svih drugih klasa našeg operacijskog sustava.

Iduća je wait funkcija koja će zaustaviti izvršavanje programa na određeno vrijeme. U nekim programskim jezicima ovo je takozvana sleep funkcija. Kako nemamo dohvaćanje vremena u našem sustavu, implementirat ćemo ovu funkciju pomoću petlje. U svakom prolazu kroz petlju odradimo jednu operaciju zbrajanja - povećavanje našeg brojača za 1. Na nama je da odredimo broj operacija koje naš sustav može izvršiti u jednoj milisekundi te prilagodimo broj ponavljanja petlje. Ovako implementirana wait funkcija nije prenosiva jer svaki sustav ima drugačije performanse. Kada smo odredili koliko operacija možemo izvršiti u jednoj milisekundi, dodajemo vanjsku petlju koja će se vrtiti onoliko puta koliko milisekundi želimo čekati

```
1  while( i < duration ) {
2    let j = 0;
3
4    // Broj ponavljanja ove petlje ovisi o racunalu
5    // Potrebno je podesiti prije koristenja
6    while( j < 190 ) {
7      let j = j + 1;
8    }
9
10   let i = i + 1;
11 }
```

Zatim, halt funkcija koja će zablokirati program dok ga ne ugasimo u potpunosti. Ona je jednostavna beskonačna petlja.

Konačno, imamo error funkciju koja će ispisati grešku i broj koji predstavlja grešku koja se dogodila.

4.2 Memory klasa

Klasa Memory zadužena je za svo upravljanje memorijom sustava. Pomoću nje dohvaćamo vrijednosti sa adrese i upisujemo vrijednosti na danu adresu, alociramo blokove memorije za nizove i slične strukture podataka, oslobađamo alociranu memoriju te brinemo da blokovi memorije nisu fragmentirani.

4.2.1 Peek i poke funkcije

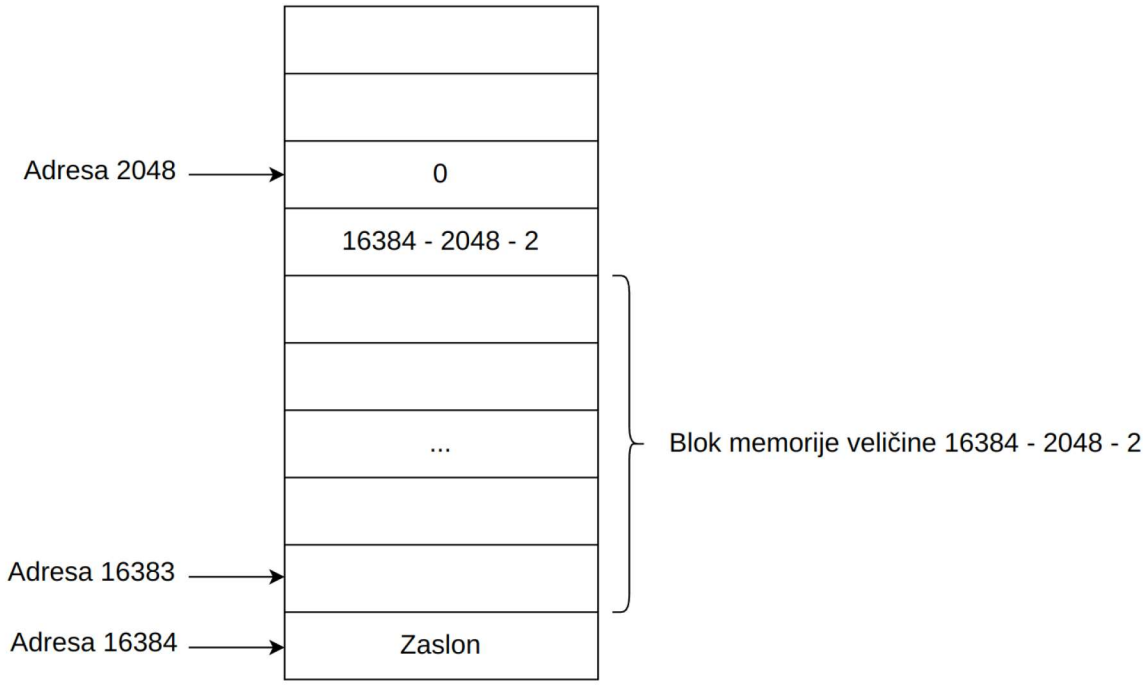
Za početak, trebamo jednostavne peek i poke funkcije za dohvaćanje vrijednosti sa adrese i unošenje podataka na adresu. Ovdje ćemo koristiti trik koji nam dozvoljava programski jezik Jack u kojem je implementirana klasa. Naime, ako napravimo varijablu tipa Array, ona je referenca na adresu, ako onda promijenimu njenu vrijednost bez da ju prvo dereferenciramo, imamo direktan pristup toj adresi u memoriji.

```
function int peek(int address) {
  Array memory;
  let memory = 0;
  return memory[address];
}
```

```
function void poke(int address, int value) {
  Array memory;
  let memory = 0;
  let memory[address] = value;
  return;
}
```

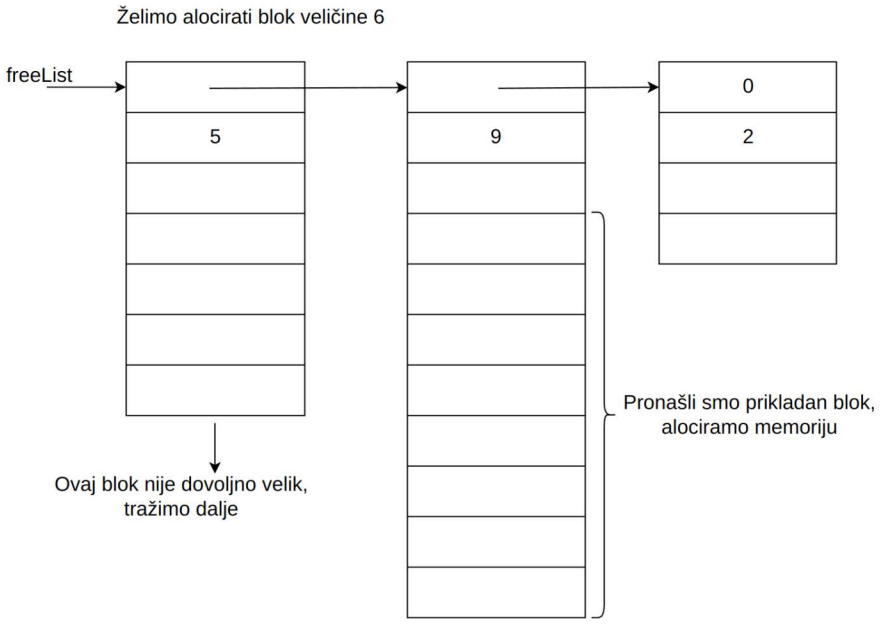
4.2.2 Alokacija i oslobađanje memorije

Kako bi pratili slobodnu memoriju na računalu, organizirat ćemo memoriju u povezanu listu. Svaki element liste na početku će sadržavati adresu idućeg te veličinu dostupne memorije koju on sadrži. Za početak, cijeli RAM ćemo označiti kao jedan element, a adresu idućeg postavljamo na 0 što nam govori da nema idućeg elementa. Veličinu ćemo postaviti na cijeli heap. Ovaj element ćemo unijeti na adresu 2048 što ćemo nazvati bazom heapa. Pokazivač na prvi element povezane liste nazivamo freeList i spremamo unutar klase Memory.

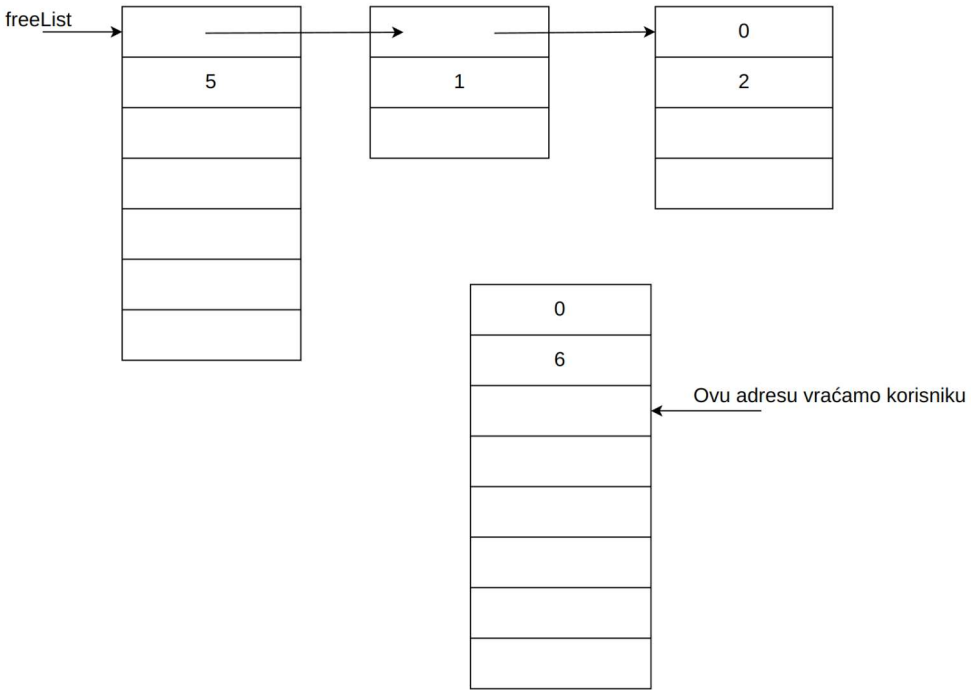


Slika 4.1: Prikaz memorije pri inicijalizaciji Memory klase

Kada želimo alocirati memoriju prolazimo kroz povezanu listu dok ne pronađemo prvi element koji je dovoljno velik da sadrži količinu memorije koju je korisnik zatražio plus dva dodatna mjesta za pokazivač na idući element i veličinu. Veličinu oduzimamo od veličine bloka i korisniku vraćamo pokazivač na prvi element memorije.



Slika 4.2: Primjer alokacije, pronalazak slobodne memorije

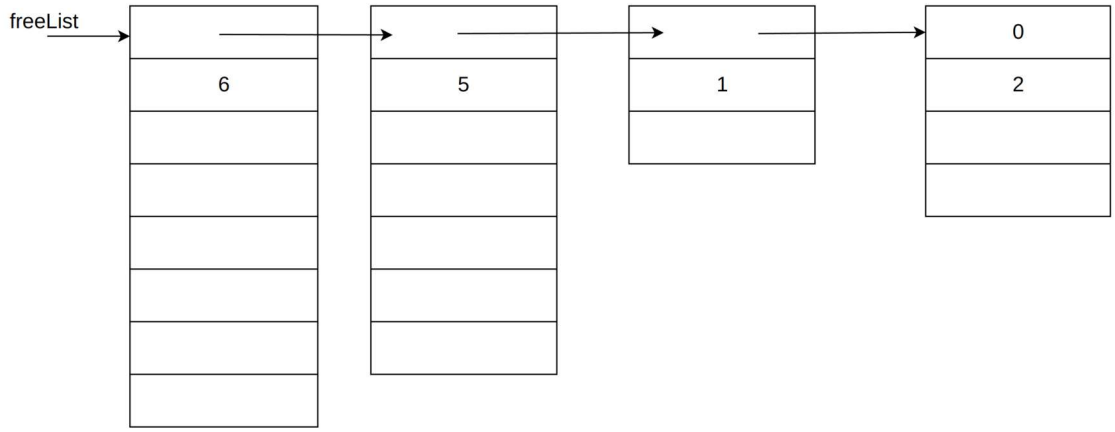


Slika 4.3: Primjer alokacije, uspješna alokacija

```
let curr = freeList;

while (~(curr = 0)) {
  let next = memory[curr];
  let curr_size = memory[curr + 1];

  if (~(curr_size < (size + 2))) {
    let memory[curr + 1] = curr_size - (size + 2);

    let object = curr + curr_size - size;
    let memory[object] = 0;
    let memory[object + 1] = size;

    return object + 2;
  }

  let curr = next;
}
```

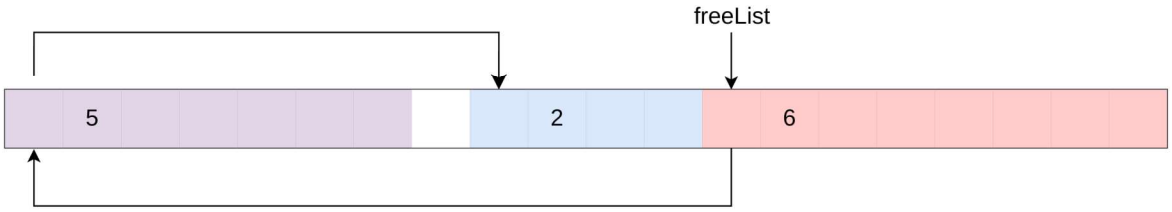
Kada želimo osloboditi memoriju, postavljamo oslobođeni blok kao prvi element povezane liste i postavljamo adresu prethodnog prvog elementa kao adresu idućeg.

```
function void deAlloc(Array o) {
  let memory[o - 2] = freeList;
  let freeList = o - 2;
}
```



Slika 4.4: Primjer oslobađanja memorije

Primjetimo da je, iako smo oslobodili memoriju, došlo do fragmentacije. Iako su prvi i treći element na slici 4.4 u memoriji jedan pokraj drugog, u našoj povezanoj listi oni su dva nepovezana elementa.



Slika 4.5: Primjer fragmentacije memorije

Za kraj želimo implementirati funkciju za defragmentaciju memorije. Ona će prolaziti kroz listu slobodnih blokova te za svaki izračunati na kojoj adresi blok završava te dodamo 1 da dobijemo prvu adresu nakon tog elementa. Zatim ponovno prolazimo kroz cijelu listu i ako ijedan blok sadrži tu adresu kao sljedeći element, pronašli smo dva uzastopna bloka te ih možemo spojiti u jedan. Nakon spajanja ponavljamo ovu operaciju. Algoritam je kompleksnosti $O(n^3)$. Na nama je da odredimo kada želimo pozivati funkciju defragmentacije. Mi ćemo ju pozivati iz deAlloc() funkcije kada ostanemo bez memorije.

```
let curr = freeList;
let updated = false;
while (~(curr = 0)) {
  let search = freeList;
  let curr_size = memory[curr + 1];

  while (~(search = 0)) {
    if ((curr + 2 + curr_size) = search) {
      let search_size = memory[search + 1];
      let memory[curr + 1] = curr_size + search_size + 2;

      if (~(prev = 0)) {
        let memory[prev] = memory[search];
      }

      let updated = true;
    }
    else {
      let prev = search;
    }

    let search = memory[search];
  }

  if (~updated) {
    let curr = memory[curr];
  }
}
```

Ovime smo se pobrinuli da je slobodna memorija uvijek spojena koliko je god to moguće.

4.3 Array klasa

Array klasa služi kao sučelje za funkcije alociranja i oslobađanja memorije. Ona omogučuje korisniku da definira blok memorije kao tip podataka. Funkcije Array.new() i Array.dispose() samo su pozivi na Memory.alloc() i Memory.deAlloc(), ali za razliku od Memory klase, Array klasu možemo koristiti kao varijablu u našem kodu. Pomoću Array klase možemo kreirati mnoge druge složene strukture podataka poput String klase koju smo ranije implementirali.

Literatura

[1] N. Nisan, S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles, Second Edition*, The MIT Press, Massachusetts, 2021.

Sažetak

U ovom radu se bavimo implementacijom osnovnih funkcija operacijskog sustava na pojednostavljenom modelu računala. Implementiramo osnovnu matematiku, upravljanje memorijom, unos i prikaz teksta i brojeva te osnovnom grafikom.

Ključne riječi

operacijski sustav, nand2tetris, upravljanje memorijom, rukovanje unosom, prikaz teksta, prikaz brojeva, osnovna grafika, matematičke operacije, simulacija os-a, edukacijski operacijski sustavi

Operating system

Summary

In this paper, we address the implementation of basic operating system functions on a simplified computer model. We implement basic mathematics, memory management, input and display of text and numbers, as well as basic graphics.

Keywords

operating system, nand2tetris, memory management, input handling, text display, number display, basic graphics, mathematical operations, os simulation, educational operating systems

Životopis

Rođen sam 1999. godine u Sisku. Srednju školu završavam u Kutini 2018. te se iste godine upisujem na preddiplomski studij matematike i računarstva na Odjelu za matematiku u Osijeku. Kao član tima natjecateljskog programiranja na Odjelu za matematiku, osvojio sam jednom drugo mjesto i dvaput treće mjesto na STEM Games natjecanju u programiranju, dva puta sam bio u top 100 na IEEEXtreme natjecanju i sudjelovao sam na CERC natjecanju.