Predavanji 10-11

Delo s pomnilnikom

Iztok Savnik, FAMNIT

Maj, 2024.

Literatura

- John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapter 7)
- Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapters 3 and 8)
- Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall, Inc., 1988.

Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Uvod

- O predstavljenih temah je bilo že govora vendar ne podrobno in ne na organiziran način
- Kako in kje v pomnilniku so shranjene spremenljivke?
  - Statične in dinamične spremenljivke
  - Statično dodeljevanje in dodeljevanje pomnilnika v kopici
- Implementacijski pogled na definicijsko območje in življensko dobo spremenljivk
  - Aktivacijski zapisi in alokacija na skladu
  - Lokalne in globalne spremenljivke
  - Kakp dostopati do globalnih spremenljivk?

Uvod

- Kako implementiramo verige funkcijskih klicev?
  - Aktivacijski zapisi na skladu
  - Strukture, ki jih kreirajo aktivacijski zapisi
- Ročno in avtomatsko zaseganje skladovnega pomnilnika
  - O tem ni bilo veliko govora...
  - pomnilnik za objekte, zapise, polja, sezname, itd.
  - Puščanje pomnilnika, viseče reference in drugi hrošči
- Strategije in algoritmi za delo s kopico
  - Cena za avtomatsko delo s pomnilnikom
  - Čiščenje pomnilnika

Čas vezave

- Vezava je asociacija med dvemi stvarmi kot sta ime in stvar, ki jo imenujemo
- Čas vezave je čas v katerem je vezava kreirana
  - Bolj splošno, čas v katerem so narejene odločitve o implementaciji
  - Vezava med vprašanjem in odgovorom
- Pomembni časi vezave v programskem sistemu
  - Čas prevajanja: Preslikava med visokonivojskimi gradniki v strojno kodo in v statičen načrt podatkov v pomnilniku
  - Čas povezovanja kode: Ime v nekem modulu referencira objekt v drugem modulu; vezava ni dokončana do časa povezovovanja kode
  - Čas izvajanja: Povezava spremenljivke z vrednostjo in druge odločitve med izvajanjem programa

Čas vezave

- Zgodniji čas vezave se pogosto asociira z večjo učinkovitostjo; kasnejši čas vezave se asociira z večjo fleksibilnostjo.
  - Programski jeziki, ki so osnovani na prevajalniku so običajno bolj učinkoviti kot prog. jeziki, ki temeljijo na tolmaču, ker uporabljajo zgodnejše odločitve
  - Npr. tolmač mora analizirati deklaracije vsakič, ko se program požene
- Izraza statičen in dinamičen se v splošnem uporabljata za opis stvari, ki zgodijo pred in med izvajanjem programa

Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Življenska doba in upravljanje s pomnilnikom

- Pomembno je razlikovati med imeni in objekti na katere se ime sklicuje
- Pomembni dogodki v življenski dobi objekta in vezave
  - Kreacija objekta
  - Kreacija vezave
  - Referenciranje spremenljivk, podprogramov in tipov, ki uporabljajo vezave
  - Deaktivacija in reaktivacija vezave, ki je začasno neuporabna
  - Brisanje vezave
  - Brisanje objekta

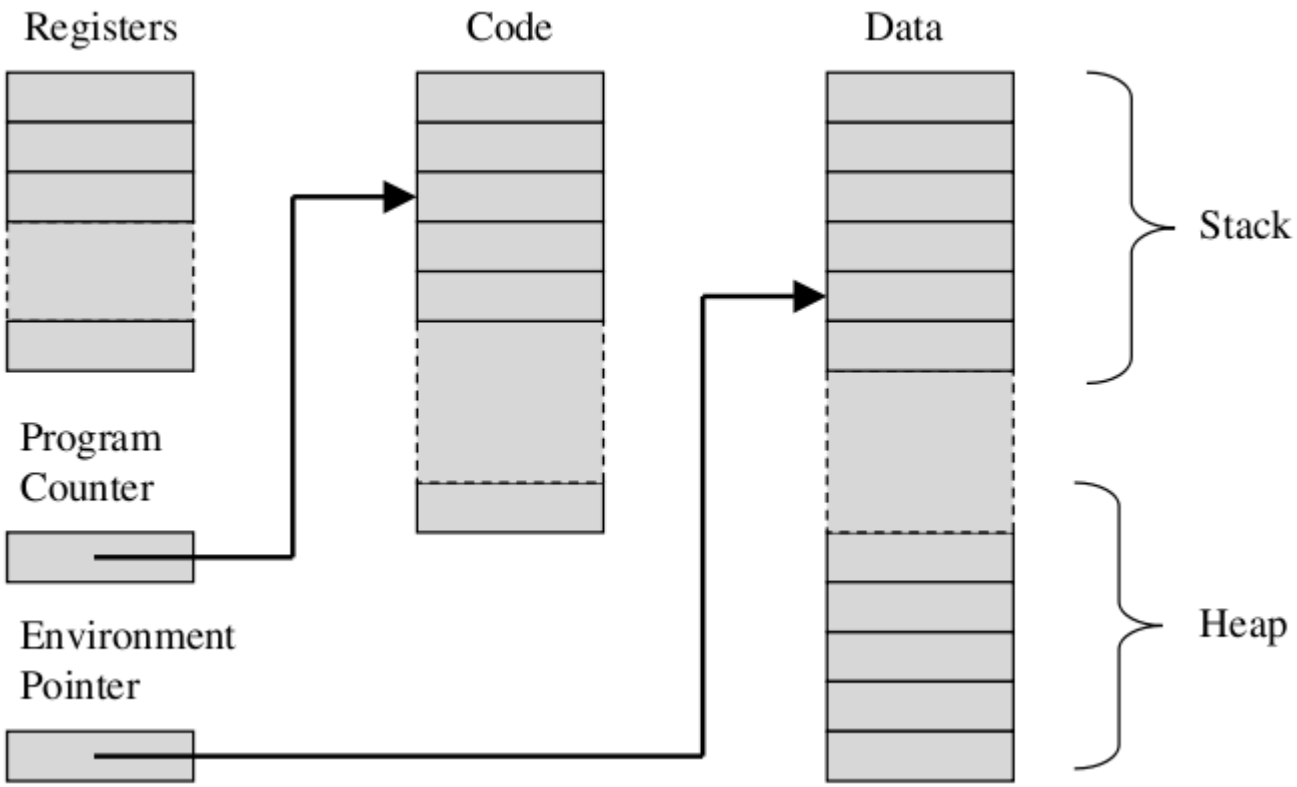
Življenska doba

- Časovno obdobje med kreacijo in destrukcijo vezave imena-objekta se imenuje življenska doba vezave.
- Čas med kreacijo in destrukcijo objekta se imenuje življenska doba objekta.
  - Objekt lahko zadrži vrednost in ga še vedno lahko uporabimo tudi po tem, ko povezano ime ni več dostopno
    - Spremenljivko podamo kot parameter funkciji po referenci
    - Fortran, var v Pascal, ali ‘&’ v C++
  - Življenska doba vezave imena z objektom je daljša kot življenska doba objekta
    - Imamo hrošča v programu—viseča referenca

Metode za upravljanje pomnilnika

- Življenska doba objekta se v splošnem ujema z naslednjimi tremi metodami za upravljanje pomnilnika
  1. Statičnim objektom dodelimo absolutni naslov, ki se ohrani med izvajanjem programa.
  2. Skladovni objekti so zaseženi in sproščeni po principu FIFO. Zaseganje in sproščanje objektov je tesno povezano s klici podprogramov.
  3. Objekti v kopici so lahko zaseženi in sproščeni kadarkoli. Ti objekti zahtevajo uporabo bolj splošenega (in dražjega) alokacijskega algoritma.

Pomnilnik programa



Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Statična alokacija

- Globalne spremenljivke so predstavljene s statičnimi objekti, s katerimi se shranjujejo tudi druge vrste objektov.
- Drugi statični objekti:
  - Inštrukcije, ki predstavljajo strojno kodo programa
  - Spremenljivke, ki so lokalne v nekem podprogramu vendar zadržijo vrednost med večimi klici podprograma
  - Numerične konstante in konstante, ki so nizi znakov. Primera: A = B/14.7 in printf("hello, world\n")
  - Tabele uporabljene v času izvajanja za razhroščevanje, dinamično preverjanje tipov, čiščenje pomnilnika, delo z izjemami in še za druge namene

Statična alokacija

- Statično zaseženi objekti se pogosto shranijo v zaščitenem delu pomnilnika uporabljenem samo za branje
- Statični aktivacijski zapisi
  - Shranjevanje spremenljivk v blokih in podprogramih
  - En aktivacijski zapis za en podprogram
  - V danem času je lahko aktiven en sam klic podprograma
  - Lokacija aktivacijskega zapisa je določena v času prevajanja
  - Enostavno in zelo hitro

Statična alokacija

- Statični aktivacijski zapis shrani:
  - Lokalne spremenljivke in začasne vrednosti
  - Argumente podprogramov, naslov vrnitve, vrnjena vrednost
  - Referenca na aktivacijski zapis okolja iz katerega je poprogram bil klican
- Problemi statičnih aktivacijskih zapisov
  - Ne moremo uporabljati rekurzije
    - Fortran je imel rekurzijo šele v kasnejših verzijah
  - Več-nitno programiranje se tudi ne more implementirati statično

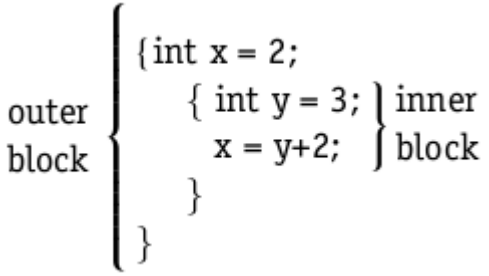
Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Skladovna alokacija

- Aktivacijski zapis ali skladovni okvir je zasežen, ko se aktivira blok ali podprogram
  - Naravno gnezdenje blokov in klicov podprogramov omogoča enostavno alokacijo prostora za lokalne objekte na skladu
- Vzdrževanje sklada je odgovornost klicne sekvence podprograma
  - Koda, ki jo izvede klican podprogram tik pred in tik po klicu (prolog/epilog)
- Dostop do višjih imenskih prostorov
  - Veriga aktivacijskih zapisov sledi strukturi blokov in podprogramov

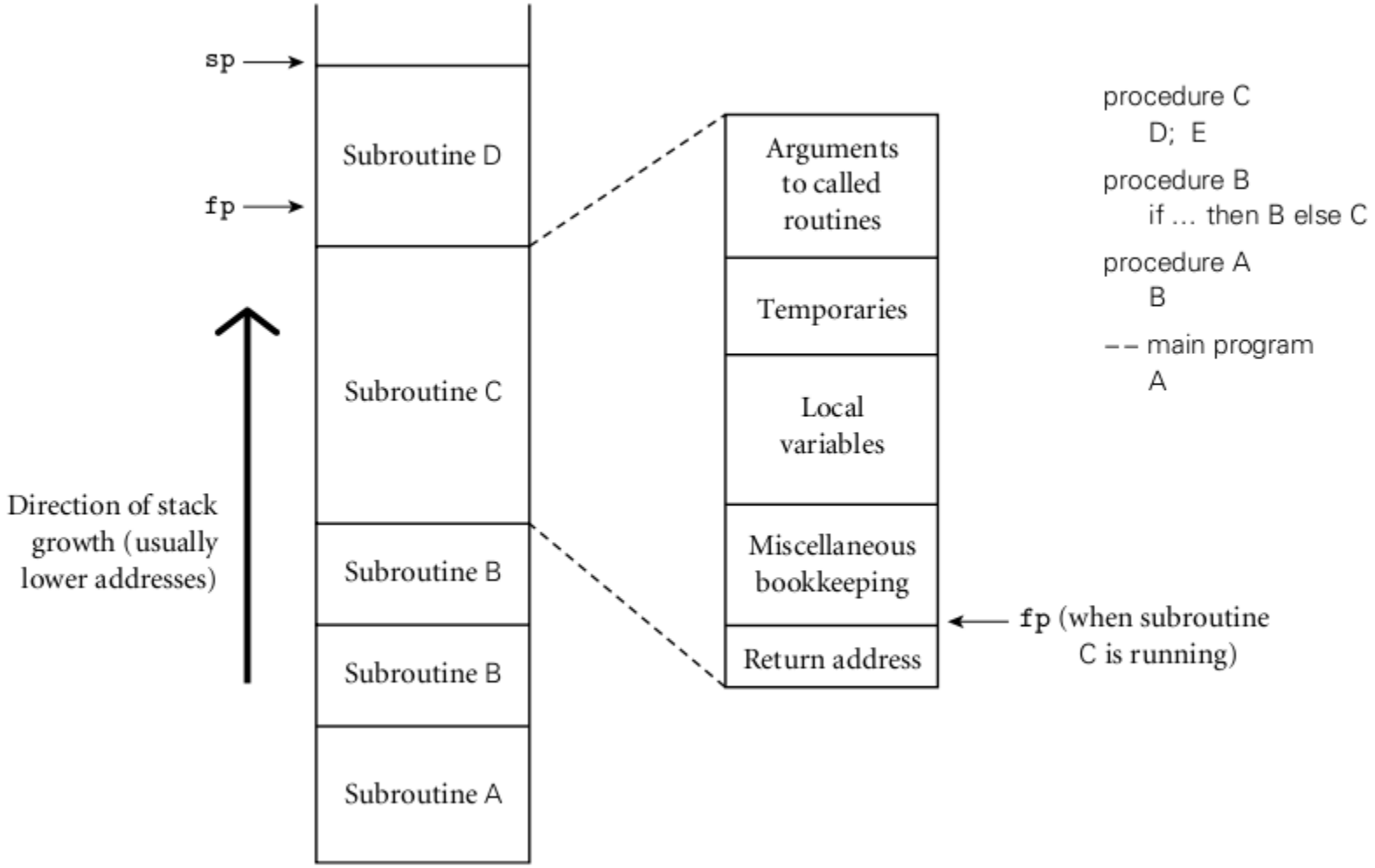
Blokovno strukturirani jeziki



- Mehanizmi za upravljanje pomnilnika so povezani s strukturo blokov
  - Spremenljivka deklarirana v bloku je lokalna
  - Spremenljivko definirano v nadrejenem bloku imenujemo globalna spremenljivka
- Lastnosti blokovno strukturiranih jezikov
  - Nove sprem. definiramo kjerkoli v bloku
  - Blok je lahko vgnezden ne sme se pa delno prekrivati
  - Po klicu se alocira pomnilnik za spremenljivke deklarirane v bloku
  - Po zaključku podprograma se (nekatere ali tudi vse) spremenljivke sprosti

Iz prejšnjih predavanj...

Skladovna alokacija pomnilniškega prostora za podprogram



Pomnilnik za spremenljivke bloka

- Mehanizmi za upravljanje pomnilnika za tri vrste spremenljivk
- Lokalne spremenljivke
  - Shranjene na skladu v aktivacijskih zapisih bloka
- Parametri
  - Parametri podprograma so shranjeni v aktivacijskih zapisih
- Globalne spremenljivke
  - Dostop preko verige aktivacijskih zapisov postavljenih na sklad pred aktivacijo trenutnega bloka

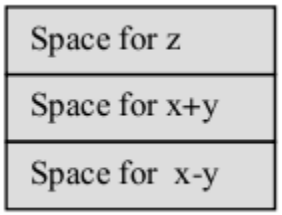
Primer

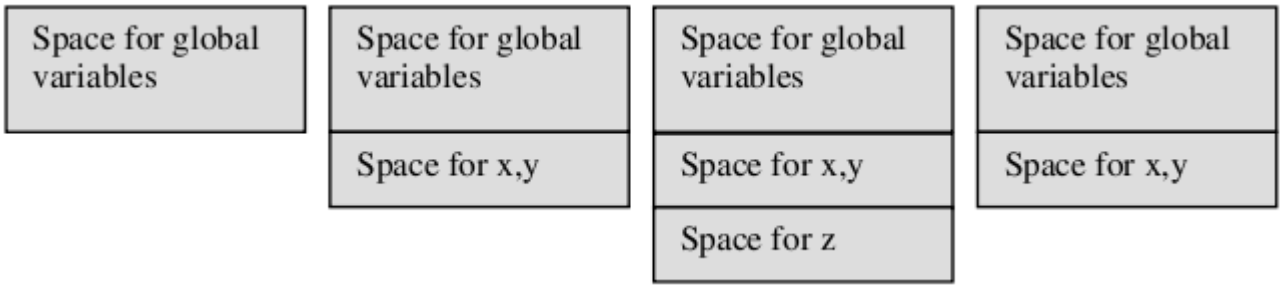
```
{ int x=0;
  int y=x+1;
      { int z=(x+y)*(x-y);
        };
};
```

- Ko se začne izvajati zunanji blok se da na sklad aktivacijski zapis, ki vsebuje prostor za x in y
- Ko se začne izvajati notranji blok se kreira na skladu aktivacijski zapis, ki vsebuje prostor za z
- Po tem, ko je vrednost spremenljivke z izračunana se aktivacijski zapis s spremenljivko z izbriše iz sklada
- Ko se zaključi zunanji blok se vzame iz sklada aktivacijski zapis, ki vsebuje spremenljivki x in y

Vmesni rezultati

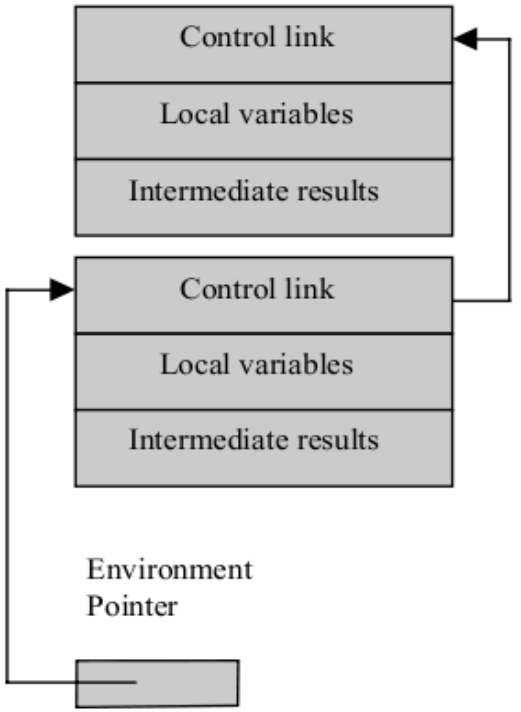
```
{ int z = (x+y)*(x-y);
}
```





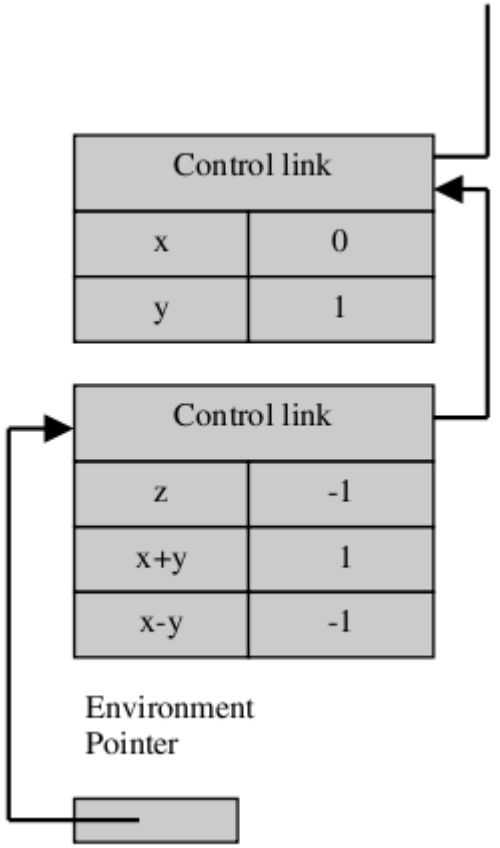
Kontrolni kazalec

- Različni aktivacijski zapisi imajo različno velikost
  - Operacije, ki dajejo in jemljejo zapise iz sklada shranjujejo kazalec na predhodni aktivacijski zapis
- Kontrolni kazalec (dinamični kazalec)
  - Kazalec na vrh predhodnega aktivacijskega zapisa
- Ko se doda nov aktivacijski zapis
  - Kontrolni kazalec novega aktivacijskega zapisa se postavi na prejšnjo vrednost kazalca na skladovno okolje
  - Kazalec na skladovno okolje se popravi



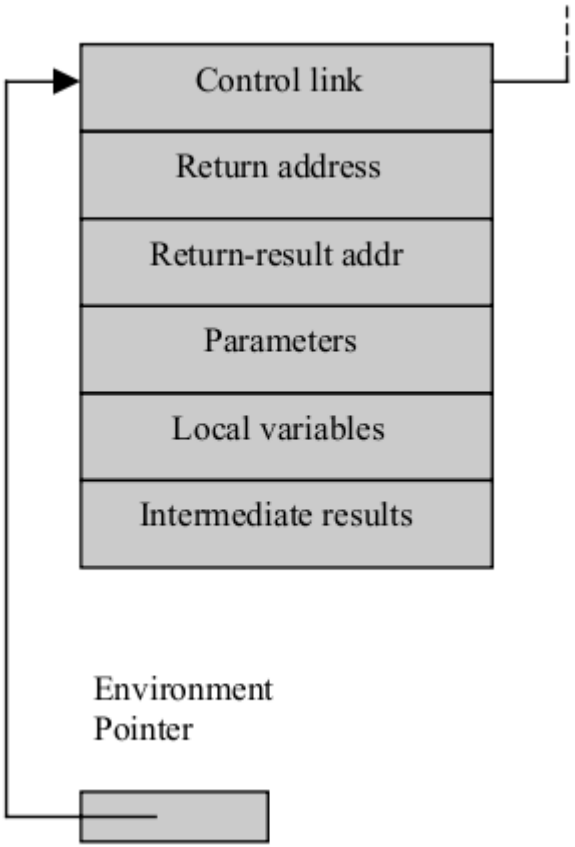
Primer

```
{ int x=0;
  int y=x+1;
      { int z=(x+y)*(x-y);
        };
        };
```



Aktivacijski zapisi za funkcije

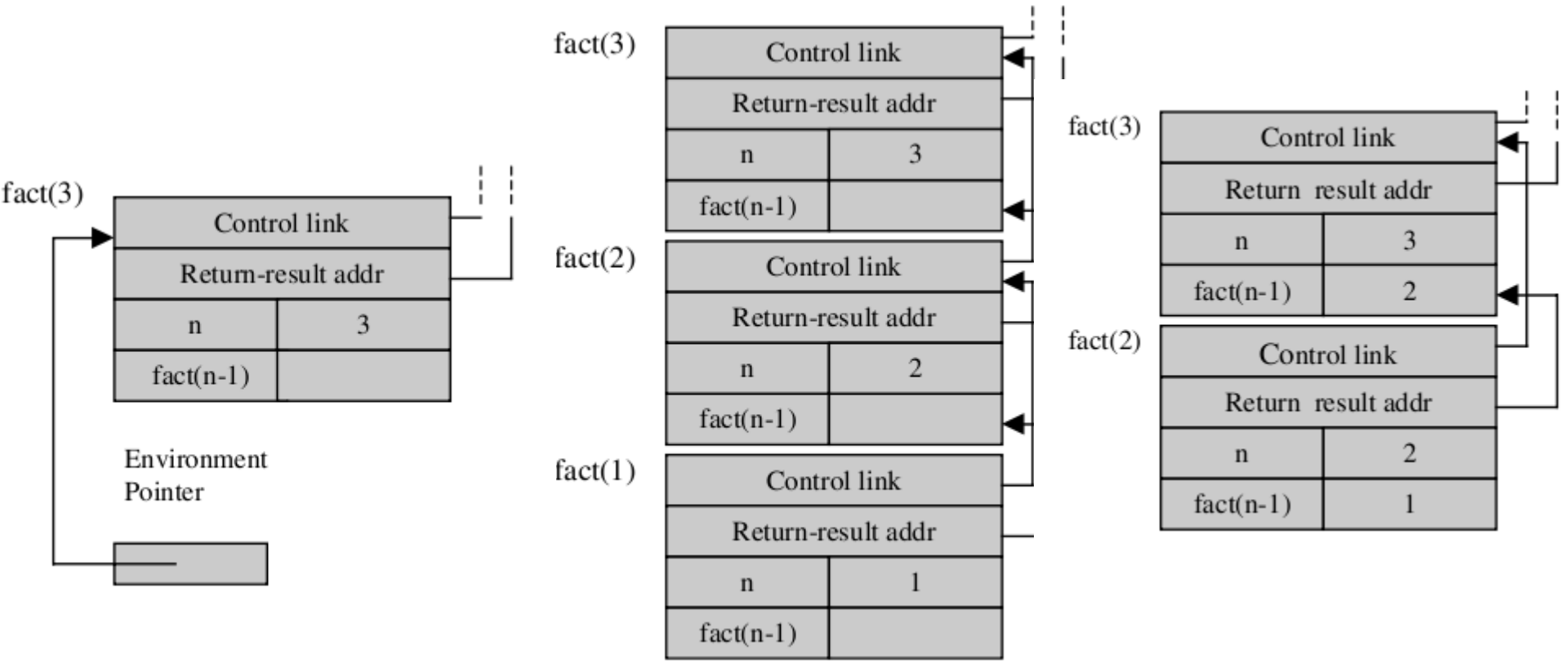
- Kontrolni kazalec (dinamičen kazalec) kaže na predhoden aktivacijski zapis na skladu
- Kazalec dostopa (statični kazalec) je kazalec na strukturno nadrejen blok
- Naslov vrnitve, ki vsebuje naslov na prvo inštrukcijo, ki se bo izvedla po zaključitvi funkcije
- Naslov rezultata, ki vsebuje naslov spremenljivke kamor se shrani rezultat funkcije
- Dejanski parameteri funkcije
- Lokalne spremenljivke, ki so definirane znotraj funkcije
- Začasne spremenljivke, ki shranjujejo vmesne rezultate v formulah



Primer

```
fun fact(n) = if n <= 1 then 1 else n * fact(n-1);
```

- Aktivacijski zapisi se dodajajo in brišejo iz programskega sklada med izvajanjem funkcije factorial



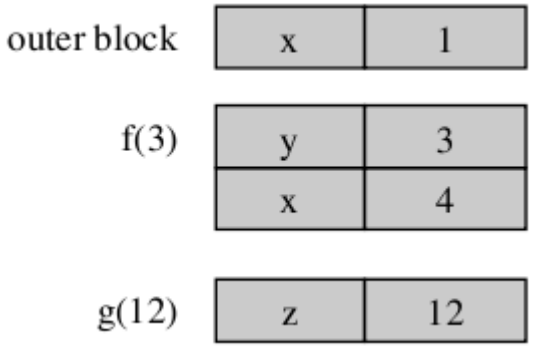
Globalne spremenljivke

- Identifikator x se uporablja znotraj telesa funkcije vendar ni definiran znotraj funkcije
- Do globalne spremenljivke x dostopamo z iskanjem aktivacijskega zapisa kjer je x definirana.
- Imamo dva pristopa k iskanju deklaracije globalnega identifikatorja
  - Statično definicijsko območje
    - Globalni identifikator je identifikator z imenom, ki je definirano v najbližjem strukturno nadrejenem bloku (kazalec dostopa)
  - Dinamično definicijsko območje
    - Globalni identifikator je identifikator, ki je definiran v najbližjem aktivacijskem zapisu gledano iz vrha proti dnu sklada (kontrolni kazalec)

Statično in dinamično def. območje

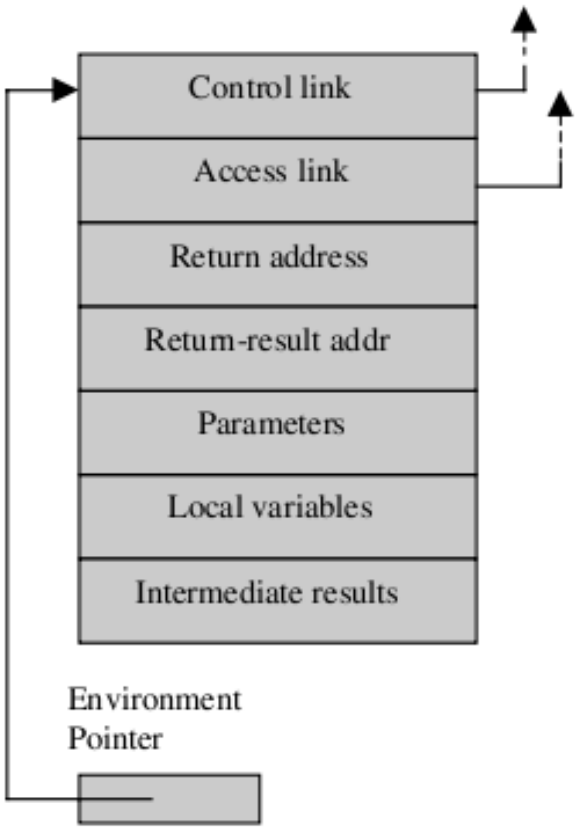
- Razlika med statičnim in dinamičnim def. območjem
  - Iskanje delaracij z uporabo statičnega def. območja uporabja statično (nespremenljivo) razmerje med bloki tekstovnega zapisa programa
  - Iskanje deklaracij z dinamičnim def. območjem je izdelano na osnovi sekvence klicev, ki ustreza dinamičnem izvajanju programa
- Example:

```
int x=1;
function g(z) = x+z;
function f(y) = {
        int x = y+1;
        return g(y*x)
};
f(3);
```



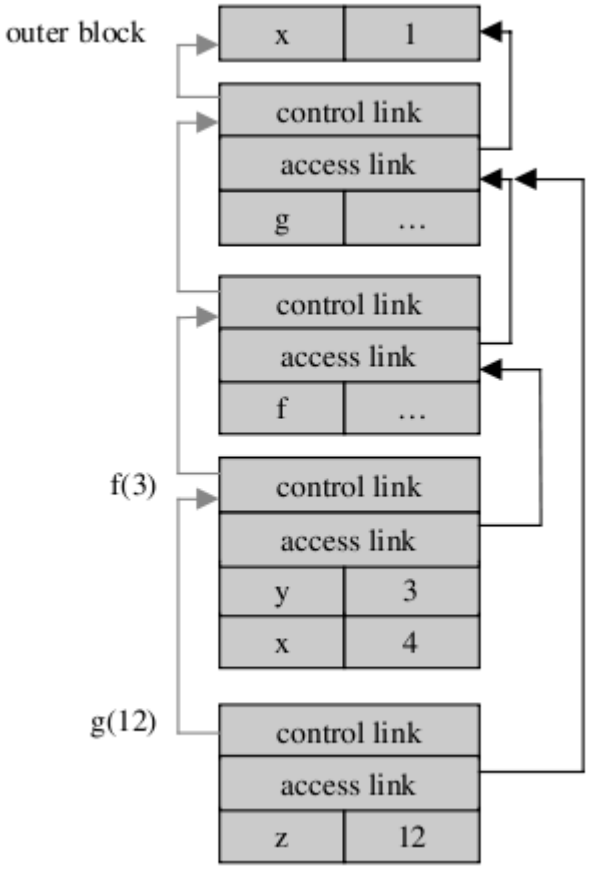
Kazalec dostopa

- Kako implementirati statično definicijsko območje?
- Kazalec dostopa (statični kazalec) aktivacijskega zapisa kaže na aktivacijski zapis najbližjega nadrejenega bloka programa
- Vrinjeni bloki ne potrebujejo dodatnega kazalca za dostop, ker je najbližji nadrejen blok tisti, ki je bil zadnji vstavljen.
- Kazalec dostopa bo v splošnem kazal na različen blok od kontrolnega kazalca



Primer

```
int x=1;
function g(z) = x+z;
function f(y) = {
        int x = y+1;
        return g(y*x)
};
f(3);
```



- Deklaracija g se pojavi znotraj definicijskega območja z deklaracijo x
  - Kazalec dostopa deklaracije g kaže na aktivacijski zapis za deklaracijo x
- Deklaracija f je tudi znotraj definicijskega območja deklaracije g
  - Kazalec dostopa deklaracije f kaže na aktivacijski zapis deklaracije g
- Klica f(3) in g(12) zahtevata dostop do aktivacijskih zapisov za def. območja funkcij s telesoma f in g, po vrsti

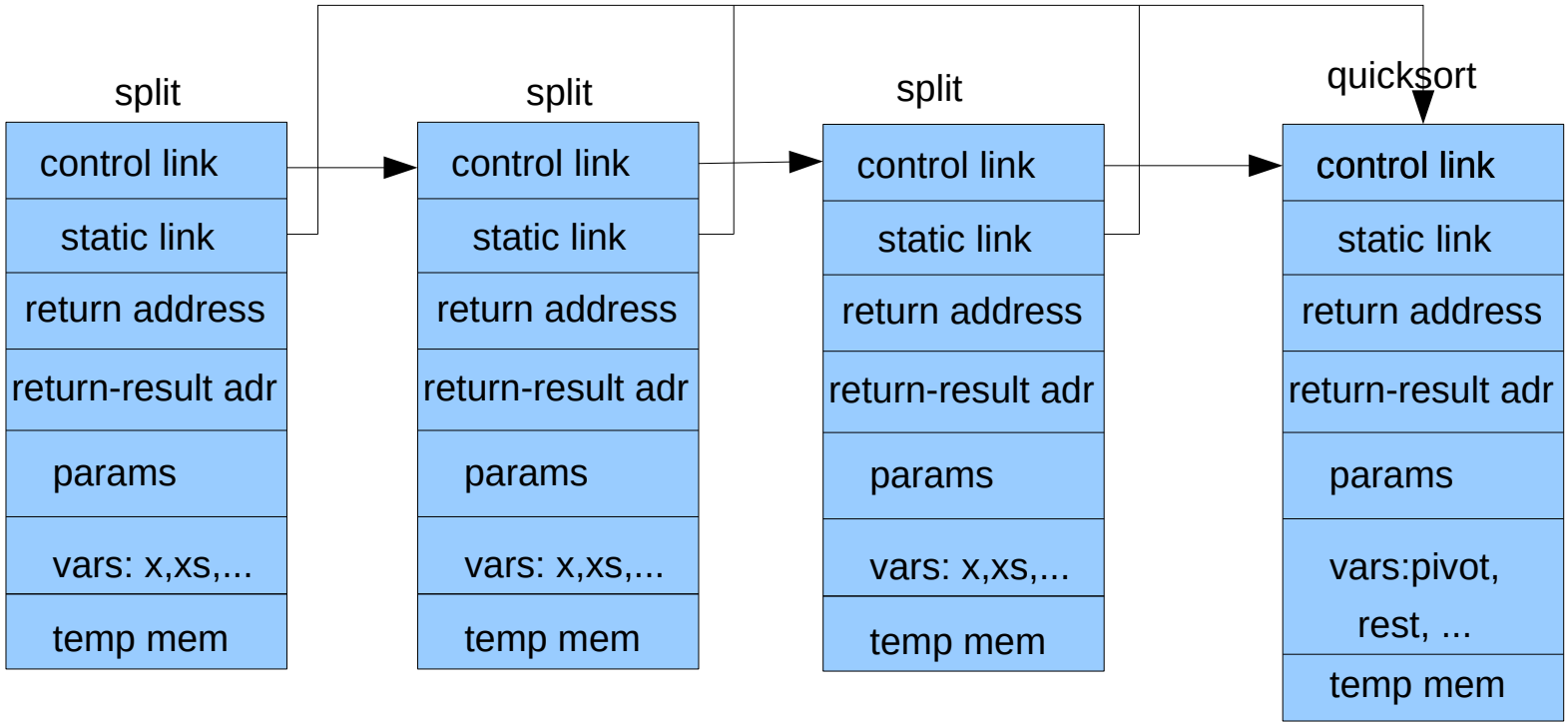
Kontrolni kazalec in kazalec dostopa

- Povzetek:
  - Kontrolni kazalec je kazalec na aktivacijski zapis prejšnjega bloka
  - Kazalec dostopa je kazalec na aktivacijski zapis najbližjega nadrejenega bloka v tekstovnem zapisu
  - Kontrolni kazalec je odvisen od dinamičnega obnašanja programa
  - Kazalec dostopa je odvisen samo od statične strukture programa
  - Kazalci dostopa so uporabljeni za iskanje lokacij globalnih spremenljivk v jezikih s statičnim def. območjem

Primer: quicksort

```
# let rec quicksort = function
      [] -> []
    | pivot::rest ->
        let rec split = function
            [] -> ([],[])
          | x::tail ->
              let (below, above) = split tail
              in
                if x<pivot then (x::below, above)
                else (below, x::above)
        in let (below, above) = split rest
            in quicksort below @ [pivot] @ quicksort above;;
val quicksort : 'a list -> 'a list = <fun>
```

Primer: quicksort

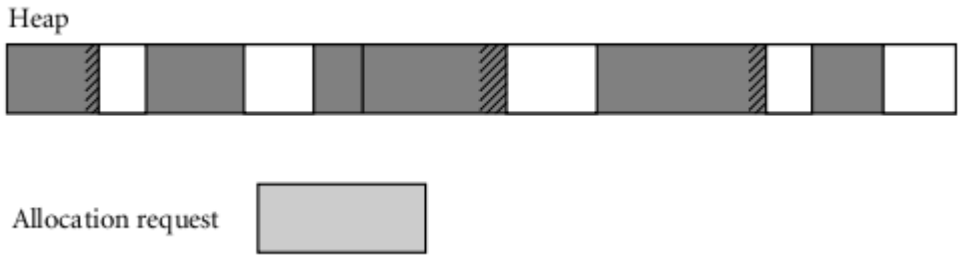


Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Alokacija pomnilnika iz kopice

- Kopica je področje pomnilnika kjer so lahko bloki pomnilnika zaseženi in sproščeni kadarkoli med izvajanjem programa
- Kopica je potrebna pri dinamični alokaciji delov neke povezane podatkovne strukture
  - Znakovni nizi, seznami in množice; velikost se lahko spreminja med popravljanjem stukture
  - Polja, zapisi, objekti, rekurzivne podatkovne strukture, ...
- Strategije za upravljanje pom. prostora kopice
  - Ravnotežje med hitrostjo in prostorom
  - Interna in eksterna fragmentacija



Algoritmi za delo s kopico

- Enojno povezan seznam—seznam prostih blokov
  - Bloki kopice, ki niso uporabljeni
  - Začetno ima seznam en sam blok, ki obsega celotno kopico
  - Zahtevek po alokaciji pregleda seznam in poišče blok primerne velikosti
  - Algoritem prvega ujemanja
  - Algoritem najboljšega ujemanja
  - Nepotreben del najdenega bloka se vrne nazaj v kopico, če ni manjši od nekega minimuma
  - Sicer preostanek predstavlja interno fragmentacijo

Enojno povezan seznam

- Pričakovali bi, da najboljše ujemanje da boljši rezultat vendar to ni res
- Časovna kompleksnost
  - Najboljše ujemanje ima višjo ceno alokacije kot prvo ujemanje
    - Vedno gre preko vseh kandidatov
    - V novejših aplikacija lahko imamo velike količine blokov!
  - Koncept “trenutni” blok v algoritmu prvega ujemanja
    - Potuje krožno po seznamu; na koncu gre na začetek
- Vsi algoritmi so linearni glede na število prostih blokov
  - V najslabšem primeru je potrebno pregledati vse bloke

Enojno povezan seznam

- Prostorska kompleksnost
  - Prvo ujemanje se obnaša boljše kot najboljše ujemanje
  - Rezultat najboljšega ujemanja je večje število zelo majhnih blokov (ostankov)
    - Interna ali eksterna fragmentacija?
  - Rezultat je odvisen od porazdelitve velikosti zaseženih blokov
    - Porazdelitev je odvisna od tipa aplikacije

Enojno povezan seznam

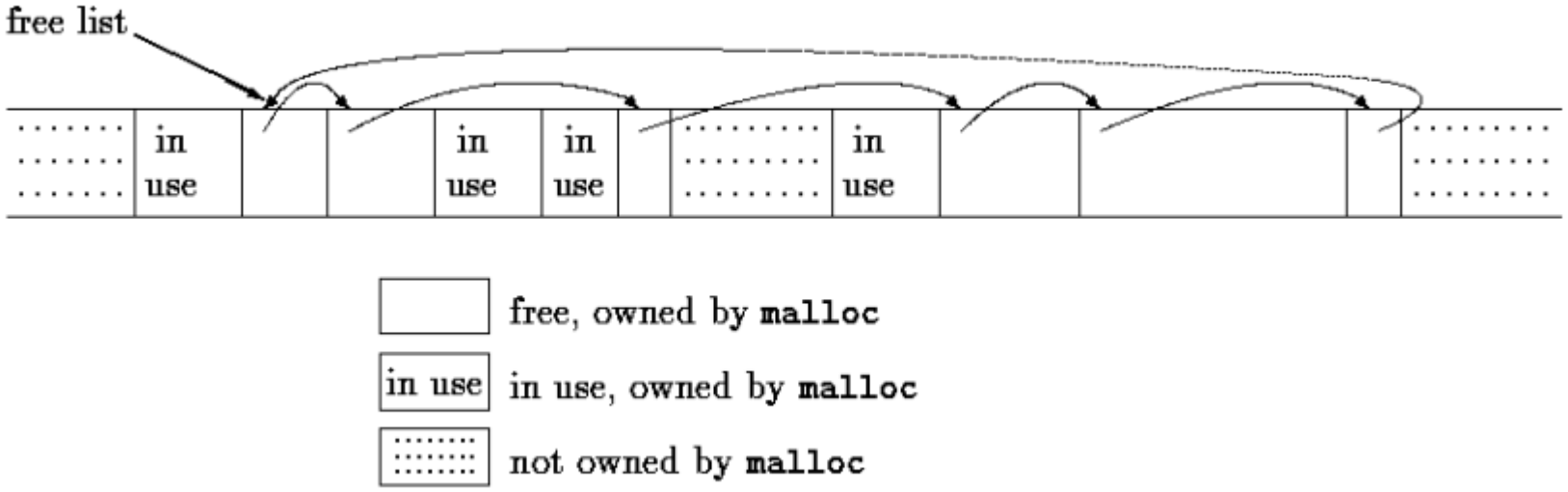
- Kako zmanjšati ceno algoritma?
  - Vzdrževanje ločenih seznamov prostih blokov po različnih velikostih
  - Če bloka ne najdemo v ustreznem seznamu potem pogledamo v seznam z večjimi bloki
    - Preostanek se shrani v ustrezen seznam (manjših blokov)
  - Ceno lahko zmanjšamo na konstanto
- Poglejmo si kopico v program. jeziku C
  - Potem bomo pregledali še nekatere rešitve, ki uporabljajo več seznamov

Kopica v C

- C začetno vsebuje implementacijo kopice osnovano na seznamu prostih blokov
  - Klici malloc() in free() se lahko pojavijo v poljubnem vrstnem redu
  - malloc() pokliče operacijski sistem, ko potrebuje več pomnilnika
    - Pom. prostor malloc() torej ni nujno zvezen
  - Prazen prostor je urejen v seznam prostih blokov
  - Vsak blok vsebuje velikost, kazalec na naslednji blok in sam prostor
  - Bloki so urejeni po naraščajočem vrstnem redu pomnilniških naslovov
    - Zadnji blok (najvišji naslov) kaže na prvega (najnižji naslov)

Kopica v C

- Za vsako zahtevo se pregleduje seznam dokler ne najdemo zadosti velik blok, t.j., “prvo ujemanje”
- Če je blok prevelik se razdeli, zasežen blok se vrne uporabniku in preostali pomnilnik ostane v seznamu prostih blokov

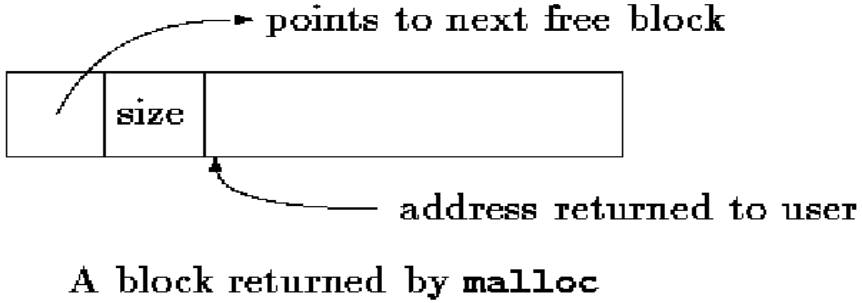


Kopica v C

- Shranjeni objekti morajo biti primerno “poravnani”
  - Nabolj restriktivni tip se lahko shrani na določenem naslovu potem se lahko shranijo tudi drugi tipi
  - Na nekaterih arhitekturah je najbolj restriktiven tip double, na drugih zadostuje int ali long

```
typedef long Align;    /* for alignment to long boundary */
union header {         /* block header */
   struct {
        union header *ptr; /* next block if on free list */
        unsigned size;       /* size of this block */
   } s;
   Align x; /* force alignment of blocks */
};
typedef union header Header;
```

Kopica v C



A block returned by `malloc`

- Zahtevana velikost v zlogih se zaokroži na večkratnik enote, ki je velikost glave bloka
  - Vsak alociran blok vsebuje najprej glavo (ena enota) in potem sledijo podatki (recimo n enot)
- Iskanje novega prostega bloka se začne tam kjer je bil prej najden zadnji blok (`freep`)
  - Ta strategija ohranja seznam homogen
  - Če najdemo prevelik blok se uporabniku vrne del repa
  - Kazalec vrnjen uporabniku kaže na začetek alociranega bloka (takoj za glavo)

malloc()

```
static Header base;       /* empty list to get started */
static Header *freep = NULL;     /* start of free list */
/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
  Header *p, *prevp;
  Header *morecore(unsigned);
  unsigned nunits;

  nunits = (nbytes+sizeof(Header)-1)/sizeof(header) + 1;
  if ((prevp = freep) == NULL) {
     /* no free list yet */
     base.s.ptr = freeptr = prevptr = &base;
     base.s.size = 0;
  }
```

```
  for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
     if (p->s.size >= nunits) {         /* big enough */
        if (p->s.size == nunits)        /* exactly */
           prevp->s.ptr = p->s.ptr;
        else {                          /* allocate tail end */
          p->s.size -= nunits;
          p += p->s.size;
          p->s.size = nunits;
        }
        freep = prevp;
        return (void *)(p+1);
     }
     if (p == freep)      /* wrapped around free list */
        if ((p = morecore(nunits)) == NULL)
           return NULL;    /* none left */
     }
}
```

morecore()
free()

```
#define NALLOC 1024  /* minimum #units to request */
/* morecore: ask system for more memory */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;
    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)   /* no space at all */
      return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

```
/* free: put block ap in free list */
void free(void *ap)
{
    Header *bp, *p;
    bp = (Header *)ap – 1;    /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break; /* freed block at start or end of arena */
    if (bp + bp->size == p->s.ptr) {   /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->size == bp) {  /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Dinamični bazeni kopice

- Kopica je razdeljena v “bazene”, enega za vsako vnaprej določeno velikost
  - Zahteva se zaokroži na naslednjo določeno velikost
  - Razdelitev v področja je lahko statična ali dinamična
- Dva mehanizma za dinamične bazene
  - Sistem kolegov, Fibonacci kopica
- Sistem kolegov (angl. buddy sys.)
  - Velikosti blokov so potence števila dva
  - Potebujemo blok velikosti 2^k
    - Ga ni ⇒ blok velikosti 2^{k+1} se razdeli na dva dela
    - En del (“kolega”) se da v seznam prostih

Dinamični bazeni kopice

  - Ko je blok sproščen se združi s “kolegom”, če je ta prost
- Fibonacci heap
  - Velikosti blokov so Fibonaccijeva števila
  - Malce bolj kompleksno
  - Vodi do malce manjše fragmentacije
- Problemi z eksterno fragmentacijo
  - Zmožnost zadovoljevanja zahtev do kopice lahko sčasoma pojenja
  - Vedno se je mogoče izmisliti sekvenco zahtev, ki jih ni možno zadovoljiti (medtem pa obstaja zadosti prostora)
  - Kompaktiranje kopice s premikanjem obstoječih blokov
    - Popraviti je potrebno še vse reference na premaknjene obj.

Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Ročno/avtomatsko delo s pomnilnikom

- Eksplicitno (ročno) delo s pomnilnikom
  - Eksplicitna alokacija in sproščanje objektov
  - Programer ima totalno kontrolo nad delom s pomnilnikom
  - C, C++, Pascal, ...
- Avtomatsko delo s pomnilnikom
  - Prevajalnik in izvajalni sistem (run-time system) delata s pomnilnikom
  - Čiščenje pomnilnika
  - Java, Haskell, Erlang, Python, Perl, ...

Eksplicitno delo s pomnilnikom

- Program alocira pomnilniške bloke nad katerimi ima popolno kontrolo
  - Po tem, ko bloki niso več uporabni so sproščeni
- Običajen malloc-free cikel dobro znan v PJ C
- Problemi:
  - Kazalec je “izgubljen” in pomnilnik pušča—zmogljivost se zmanjšuje in pomnilnik se na koncu napolni
  - Objekt je pomotoma sproščen in dobimo viseči kazalec
- Programer mora biti zelo pazljiv, da načrta vse alokacije pomnilnika v parih

Primer

```
void function_which_allocates() {
    // allocate an array of 50 floats
    float* a = new float[50];
     // additional code making use of ’a’
     // return to caller, having forgotten
     // to delete the memory we allocated
}
```

```
int main() {
    function_which_allocates();
    // the pointer ’a’ no longer exists, and therefore cannot be freed,
    // but the memory is still allocated. a leak has occurred.
    int* p = new(1024);
    int* q = p;
    delete p; // q is dangling pointer by now
    // main continues
    *q = 2048; // memory corruption: write to garbage memory
    delete q; // memory corruption: double free of memory
}
```

Eksplicitno delo s pomnilnikom

- Vračanje pomnilnika je velikokrat avtomatično
  - Ob vrnitvi funkcije se sprostijo vse spremenljivke, parametri
- Disciplinirano zaseganje/vračanje pomnilnika vodi do učinkovitih programov
  - V realnosti so vsi hitri programi napisani v PJ, ki omogočajo eksplicitno delo s pomnilnikom
  - Zmogljivost PJ, ki uporablja avtomatsko delo s pomnilnikom je primerljiva z ekspl. delom samo če je zadosti pomnilnika.
  - Prog. jeziki s čišenjem pomnilnika so za velikostni razred počasnejši od jezikov brez čiščenja spom.
  - Če je pomnilnik problem, so jeziki z eksplicitno kontrolo vedno boljši

Vsebina

1. Čas vezave
2. Življenska doba in upravljanje pomnilnika
3. Statično dodeljevanje pomnilnika
4. Dodeljevanje skladovnega pomnilnika
5. Dodeljevanje pomnilnika iz kopice
6. Eksplicitno delo s pomnilnikom
7. Čiščenje pomnilnika

Čiščenje pomnilnika

- Alokacija pomnilnika se vedno proži z neko specifično operacijo v programu
  - Kreiranje objekta, dodajanje v seznam, prireditev niza z večjo dolžino, ...
- Sproščanje lahko naredimo na dva načina
  - Eksplicitno v nekaterih jezikih
    - Npr., C, C++, in Pascal
  - Drugi prog. jeziki sprostijo zasežene objekte implicitno
    - Po tem, ko se ne da več dostopat do njih preko spremenljivk
    - Jezik vsebuje sistem za čiščenje pomnilnika, ki identificira in sprosti nedostopne objekte

Čiščenje pomnilnika

- Jeziki s čiščenjem pomnilnika
  - Večina funkcijskih in skriptnih jezikov zahteva čiščenje pomnilnika (Lisp,Haskell,Ocaml,Go,Scala,C#,itd.)
  - Precej imperativnih jezikov (Modula-3, Java, in verzija C) vsebuje čiščenje pomnilnika
- Argumenti za eksplicitno delo s pomnilnikom
  - Enostavnost implementacije PJ
    - Uporaba podsistema za čiščenje pomnilnika naredi implementacijo PJ precej bolj kompleksno.
  - Hitrost izvajanja programov
    - Celo zelo dobro narejeni čistilci pomnilnika uporabijo precej časa za čiščenje v nekaterih programih

Čiščenje pomnilnika

- Argumenti v prid čiščenju pomnilnika
  - Napake pri ročnem delu s pomnilnikom so med najbolj pogostimi in najdražjimi napakami v realnih sistemih
  - Objekt je prehitro sproščen
    - Program lahko dostopa do viseče reference
    - Dostop do pomnilnika, ki se ne uporablja več za dan objekt
  - Objekt ni sproščen po uporabi
    - V programu “pušča pomnilnik”; hitro zmanjka pomnilnika
    - Napake pri dealokaciji se zelo težko odkrijejo

Čiščenje pomnilnika

- Avtomatično čiščenje pomnilnika je zaželjeno v večini programskih jezikov
  - To je zaključek tako načrtovalcev PJ kot tudi programerjev.
  - Velikokrat nočemo porabiti več dni za razhroščevanje programa ampak bi želeli imeti rezultat takoj.
  - Cena čiščenja pomnilnika se kompenzira s hitrejšo strojno opremo
  - Realnost?
- V večih primerih ni možno implementirkati učinkovitega sistema brez eksplicitne kontrole nad zaseganjem pomnilnika
  - Večina prevajalnikov, operacijskih sistemoiv, podatkovnih sistemov ... je napisanih v C ali C++

Štetje referenc

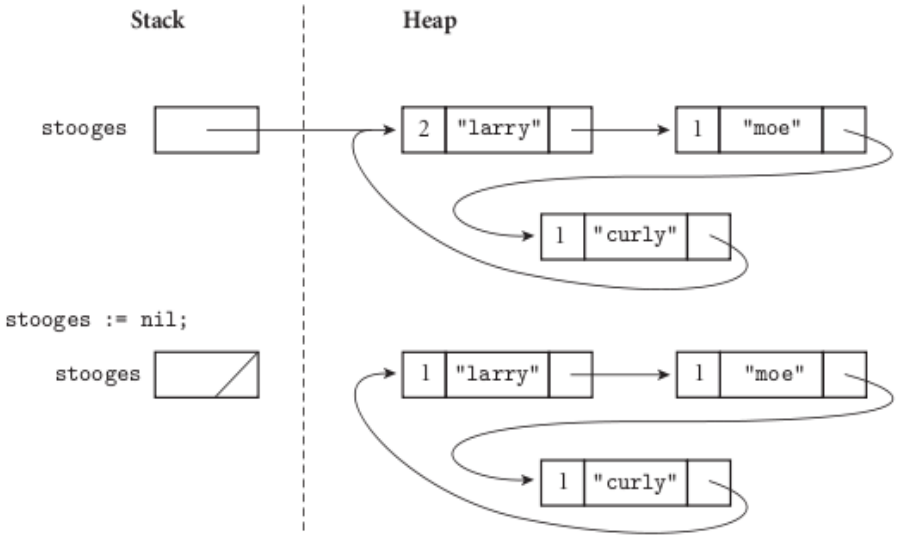
- Kdaj objekt ni več uporaben?
  - Ne obstaja kazalec na objekt
- Enostavna rešitev
  - Shrani števec kazalcev, ki referencirajo objekt v samem objektu
    - Začetna vrednost števca je 1
  - Ko je kazalec a prirejen kazalcu b
    1. dec_rc(object(b))
    2. Make assignment b := a
    3. inc_rc(object(a))
  - Ob zaključku podprograma
    - Epilog klicne sekvence zmanjša števce referenc vsem objektom na katere so kazale lokalne spremenljivke in parametri

Štetje referenc

- Ko je števec referenc enak 0 se objekt lahko sprosti
  - Izvajalni sistem rekurzivno zmanjšuje števec za vse objekte, ki so referencirani iz objekta, ki se je sprostil
- Kaj potrebujemo za štetje referenc?
  - Prog. jezik mora znati identificirati lokacijo vsakega kazalca
    - Katere besede v objektu ali skladovnem okvirju predstavljajo kazalce?
    - Uporabimo lahko deskriptorje tipov (odmiki komponent), ki jih generira prevajalnik
  - V spošnem velja, da prog. jeziki s strogimi tipi lahko uporabljajo takšne algoritme
    - Obstajajo tudi rešitve za jezike, ki nimajo strogih tipov

Štetje referenc

- Problem je definicija “uporabnega objekta”.
  - Objekt je neuporaben tudi, če obstajajo reference na objekt
  - Štetje referenc ne izloči objekte v krožnih strukturah
- Veliko jezikov uporablja ŠR za nize var. dolžine
  - Ti ne vsebujejo referenc
- Perl uporablja ŠR za vse dinamično alocirane podatke
  - Programmer mora prerezati cikle



Označi-in-pometi

- Boljša definicija uporabnega objekta
  - Do objekta dostopamo preko verige legalnih kazalcev, ki se začnejo z imenovanim objektom
  - Krožno referencirani objekti ne ostanejo v pomnilniku
- Uporabne objekte odkrijemo z rekurzivnim preiskovanjem kopice.
  - Začnemo z vsemi “zunanjimi” kazalci (precej drago)
- Označi-in-pometi
  - Klasičen mehanizem za identifikacijo blokov pomnilnika, ki niso več uporabni.
  - Ko se velikost prostega prostora v kopici zmanjša pod minimum se začnejo izvajati naslednji trije koraki.

Označi-in-pometi

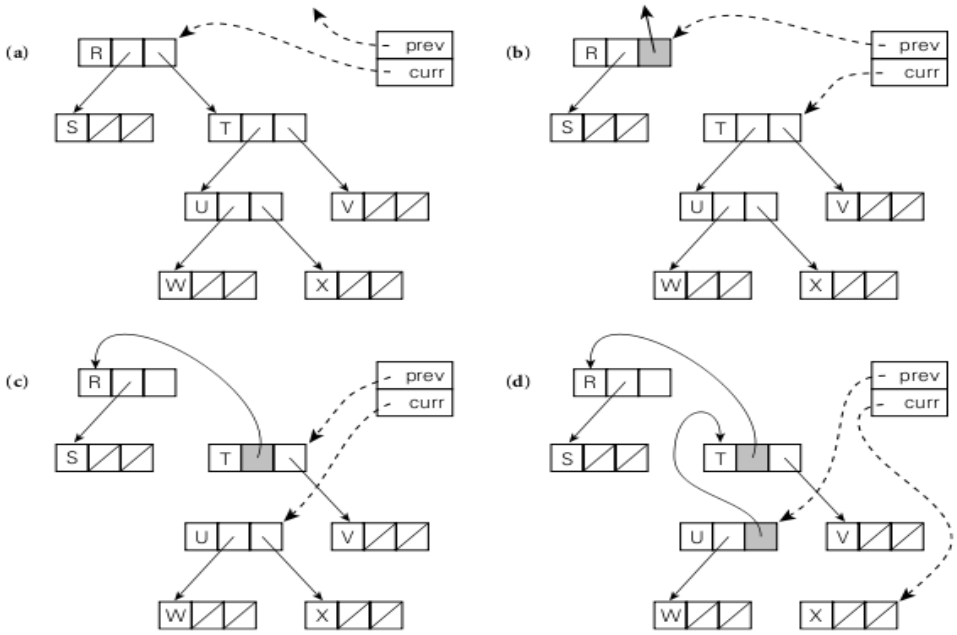
1. Zbiralec se najprej sprehodi po kopici in označi vse bloke kot neuporabne
2. Zbiralec začne s kazalci izven kopice in rekurzivno preiskuje vse povezane podatkovne strukture ter jih označuje kot uporabne
3. Zbiralec se še enkrat sprehodi po kopici in premakne vse bloke, ki so označeni kot neuporabni v seznam prostih blokov

Označi-in-pometi

- Problemi algoritma
  - Za vsak uporabljen blok moramo vedeti kje se začne in kje konča
  - Vsak blok mora na začetku vsebovati velikost bloka in značko, ki pove ali je prost ali uporabljen
  - Zbiralec mora v koraku (2) biti sposoben poiskati kazalce vsebovane v vsakem bloku
    - Rešitev: vstavi kazalec na deskriptor tipa nekje v začetek bloka

Izboljšave Označi-in-pometi

- Obrnjen kazalec
  - Rekurzivno preiskovanje kopice uporablja sklad
    - Pomnilnik v kopici lahko uporabimo za sledenje poti
  - Med preiskovanjem poti do nekega bloka zbiralec obrača kazalce nazaj
    - Zbiralec si zapomni trenutni blok in blok od koder je prišel
  - Iskanje se lahko implementira brez sklada



Izboljšave Označi-in-pometi

- Ustavi-in-kopiraj
  - Zmanjšamo zunanjo fragmentacijo s kompaktiranjem pomnilnika
  - Izločimo koraka (1) in (3)
  - Kopico razdelimo na dve regiji enake velikosti
    - Zaseganje pomnilnika se dogaja v prvem delu
  - Vsak dosegljiv blok se prepiše v drugo polovico kopice tako da ni več zunanje fragmentacije
    - Stara kopija se označi kot “uporabna”
    - Kazalci na star blok se popravijo tako da kažejo na nov blok
  - Ko zbiralec zaključi so vsi uporabni bloki shranjeni v drugem delu kopice
    - Prvi del kopice je prazen!
    - Zbiralec zamenja prvo in drugo polovico

Generacijsko zbiranje

- *Opazka*: večina zaseženih objektov živi zelo kratek čas
- Kopica je razdeljena na več področij (pogosto dve)
  - Ko zmanjkuje prostora zbiralec najprej pregleda najmlajšo regijo (“vrtec”)
    - Zelo verjetno ima največi del smeti v kopici
  - Če v najmlajši regiji ni zadosti prostora potem zbiralec gre iskati prostor v naslednjo regijo (po starosti)
  - V najslabšem primeru zbiralec pregleda kompletno kopico
- V večini primerov je iskanje omejeno samo z velikostjo najmlajše regije

Generacijsko zbiranje

- Objekt, ki preživi nekaj zbiranj (pogosto eno) je promoviran v naslednjo starejšo regijo
  - Reminiscenca Ustavi-in-prepiši
  - Promocija bloka spremeni kazalec na blok; vsi kazalci morajo biti popravljeni
  - Za vsako prireditev kazalca prevajalnik pripravi kodo, ki preveri ali je nova vrednost stari-v-novi kazalec
  - Ta mehanizem prirejanja je poznan kot pisalna meja (angl. write barrier)