



Operační systémy 2

Aplikace a práce s pamětí

Petr Krajča



Katedra informatiky
Univerzita Palackého v Olomouci

- aplikace pracují s pamětí jinak než OS \Rightarrow stránky vs. objekty
- zásobník a halda (heap)
- rozdílné požadavky i rozhraní
 - C: `malloc`, `free`, `realloc`, ...
 - C++: operátory `new`, `new[]`, `delete`, ...
 - Java: `new`
 - LISP & Scheme: `cons`, `vector`, ..., `lambda`?
- není záležitost OS \Rightarrow běhové prostředí (JVM, CLR), standardní knihovna (libc – `dldmalloc`, `ptmalloc`, `Hoard`, `TCMalloc` [Google], `jemalloc` [FreeBSD])
- lze vyměnit (mohou být specifické pro OS i jednotlivé aplikace)
- záleží na nárocích (e.g., typické velikosti objektů, počtu vláken, tolerovaná režie, rychlost, míra fragmentace)
- potřeba řešit cache, lokalitu dat, TLB, atd.

- ptmalloc v.3 – knihovna v C
- založená na „Doug Lea's Malloc” (dlmalloc, který je datovaný až do roku 1987)
- přidává podporu pro více vláken a SMP
- „optimalizovaný” pro běžné použití
- dokumentace \Rightarrow zdrojový kód
- rozhraní poskytující operace malloc, free, realloc, ...
- získává souvislý blok paměti přes volání OS brk/sbrk (mění velikost datového segmentu o daný počet bytů)

```
void * simple_malloc(int size) {  
    return sbrk(size);  
}
```
- uvolnění přes sbrk(-velikost) \Rightarrow jednoduchý, ale fragmentuje paměť
- (velké bloky) případně mmap (mapování anonymní paměti, /dev/zero)

- každé vlákno může mít svůj alokátor
- malloc přiděluje menší kousky paměti
- alokuje po blocích zakrouhlených na dvojnásobek slova (slovo = 32/64 bitů, podle platformy) tj. 8B (minimálně)
- některé platformy přímo vyžadují zarovnání paměti
- navíc hlavička
 - velikost předchozího bloku (pokud je volný)
 - velikost celého bloku
 - příznak, jestli je předchozí blok volný (uložen ve velikosti – spodní bit)
- každý blok paměti začíná na sudém násobku slova
- vrácená paměť začíná na sudém násobku slova
- nejmenší alokovatelný blok paměti je 4 slova (na i386 to je 16 B)
- není rozdíl mezi malloc(1) a malloc(6)!

```

chunk-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        | velikost predchoziho bloku (pokud P = 1)          |
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+ |P|
        | velikost celeho bloku (size)                      1| +--+
mem-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      |                                                         |
      +-      (size - sizeof(size_t)) bytu                    -+
      |                                                         |
      +-      pouzitelne pameti                                -+
      |                                                         |
      +-                                                         -+
      :                                                         |
      :                                                         |
chunk-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      |                                                         :

```

- udržuje si informace o volných blocích
- volné bloky evidovány
 - do velikosti 256 B v 32 frontách (oboustraný spojový seznam), každá pro jednu velikost
 - větší bloky v 16 stromech (trie), každý pro bloky o velikosti $2^n - 2^{n+1}$, pro $n = \langle 8, 23 \rangle$; (uspořádané podle velikosti)
 - speciální strom pro ostatní (větší) bloky
- při zavolání free
 - ověří se, jestli je následující blok volný (a případně se sloučí)
 - ověří se, jestli je předchozí blok volný (a případně se sloučí)
 - zařadí se do příslušného seznamu/stromu
- nikdy nejsou dva volné bloky vedle sebe
- malloc se nejdříve snaží najít první vhodný blok, který byl uvolněn
- pokud malloc vybírá z volných bloků, vybírá z následujících strategií
 - v seznamech: FIFO
 - ve stromech: best-fit

```

chunk-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        | Size of previous chunk                                     |
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
'head:' | Size of chunk, in bytes                                     |P|
mem->   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        | Forward pointer to next chunk in list                     |
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        | Back pointer to previous chunk in list                     |
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
        | Unused space (may be 0 bytes long)                         .
        .                                                             .
        .                                                             |
nextchunk-> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
'foot:' | Size of chunk, in bytes                                     |
        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

```

realloc(ptr, size)

- pokud je v požadovaném bloku místo, nemění velikost
- jinak kopíruje data do nového bloku a starý uvolní
- není dobrý nápad:

```
int i = 0;
char * buf = malloc(sizeof(char));
while ((c = getc(stdin)) != EOF) {
    buf = realloc(buf, (++i) * sizeof(char));
    buf[i] = c;
}
```

Další související operace

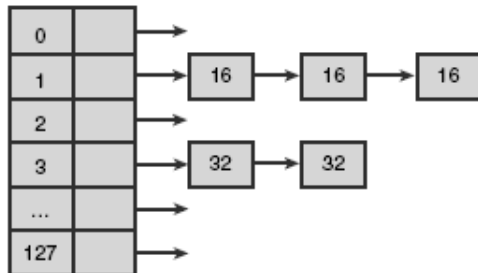
- mallopt – nastavení vlastností alokátoru
- posix_memalign – získá paměť zarovnanou na stránky

- API + (Frontend) + Backend; jeden proces může mít několik heapů
- viz SolRu p. 731
- každý objekt obsahuje hlavičku (8 B) jako v ptmalloc

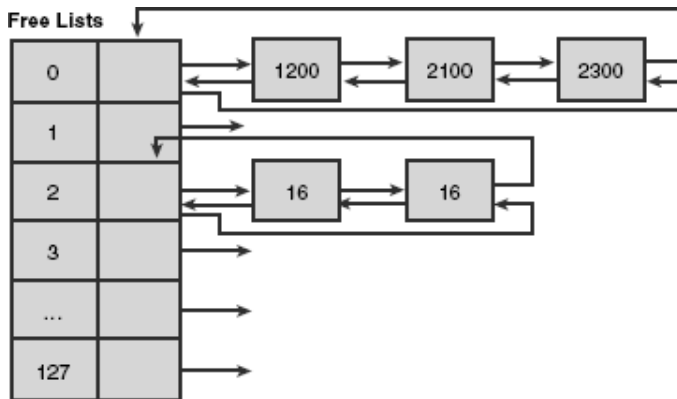
Frontend

- optimalizace alokací menších bloků
- „přichystané bloky k použití“
- LAL (Look-Aside-Lists): 127 seznamů s objekty velikosti $n \times 8 \text{ B}$
- LF (Low Fragmentation): jako LAL, ale až pro objekty do 16 kB (větší objekty \Rightarrow větší granularita)
 - objekty 1 B-256 B: zarovnané na 8 B
 - objekty 257 B-512 B: zarovnané na 16 B
 - objekty 513 B-1024 B: zarovnané na 32 B
 - objekty 1025 B-2048 B: zarovnané na 64 B
 - ...
- používá se LAL i LF (od Vista implicitní)

Look Aside Table



- podobný ptmalloc
- velké bloky se alokují přímo přes VM
- pokud není k dispozici malý blok, rozdělí se větší

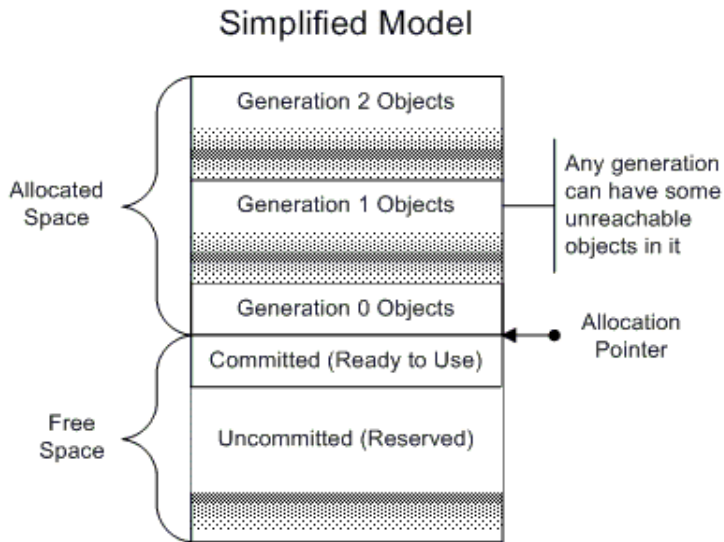


- **Garbage Collector**: John McCarthy vyvinul pro LISP (1958)
- renesance (**main stream**) v průběhu devadesátých let (**Java**)
- **zjednodušení vývoje aplikací na úkor režie počítače**
- jinak přítomna v ostatních jazycích i dřív
- **centrum pozornosti** \Rightarrow **Java, .NET, skriptovací jazyky**
- GC lze doplnit i do C/C++
- Boehm(-Demers-Weiser) **garbage collector** (okamžitá náhrada GC_MALLOC)

- každý objekt obsahuje počítadlo, kolik na něj odkazuje objektů
- pokud je na něj vytvořený/zrušený odkaz, zvýší/sníží se počítadlo o 1
- v případě, že je počítadlo 0 \implies objekt je uvolněn
- problémy
 - režie (paměť i CPU)
 - cyklické závislosti
- Delphi, PHP, Python, COM
- ARC (automatic reference counting) – ObjectiveC, Swift (počítání referencí řeší překladač)

- objekty rozděleny do tří barev/množin (černé, šedé, bílé)
- na počátku:
 - bílé – kandidáti na uvolnění
 - šedé – objekty odkazované z „kořenů“ (určené k prověření)
 - černé – nemají odkazy na bílé objekty
- algoritmus:
 - 1 vezme se jeden šedý objekt a je přebarven černě
 - 2 z něj odkazované bílé objekty přebarveny šedě
 - 3 opakuje se, dokud existují šedé objekty

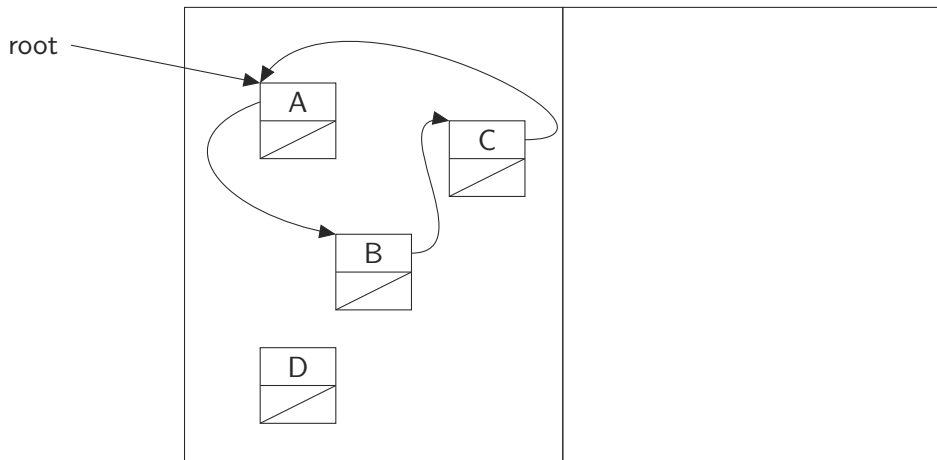
- přesunující vs. nepřesunující (moving/non-moving)
 - lze přeskupit objekty, aby vytvořily souvislý blok
 - režie na přeskupení/snížení fragmentace \Rightarrow rychlejší alokace
- generační
 - předpokládá se, že krátce žijící objekty budou žít krátce
 - paměť rozdělena na generace \Rightarrow nad staršími objekty nemusí probíhat sběr příliš často
- přesné vs. konzervativní
 - existují informace o uložení ukazatelů? (JVM)
 - alternativně lze určit, jestli daná část objektu je ukazatel (Boehm GC)
- inkrementální vs. stop-the-world
 - problém se zastavením běhu programu, aby mohlo dojít k „úklidu“ (změny odkazů)
 - inkrementální redukuje výše zmíněný problém
 - možné vylepšit paralelním zpracováním



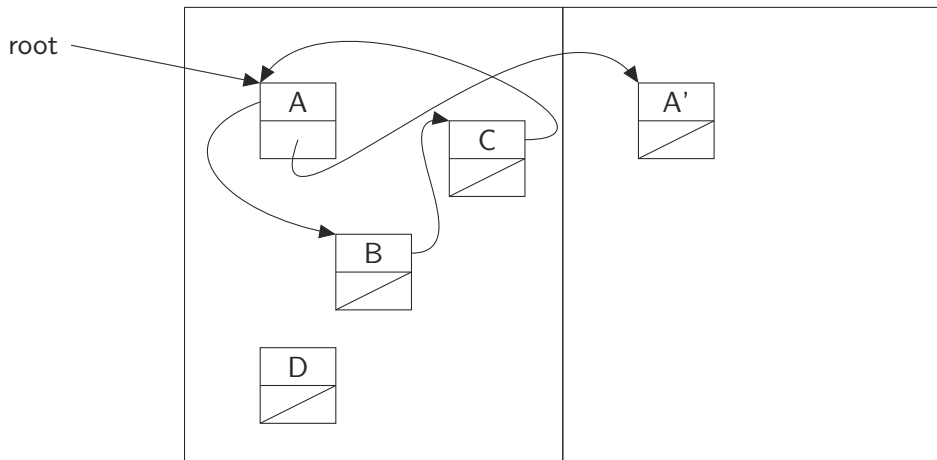
- populární typ GC
- každý objekt má příznak, jestli se používá (nastavný na 0)
- v průběhu „úklidu“
 - objekty odkazované ze zásobníku, statických proměnných (popř. registrů) \Rightarrow označeny jako používané
 - objekty odkazované z označených objektů \Rightarrow označeny
 - neoznačené objekty uvolněny
- problém s (vnější) fragmentací
- špatná práce s cache

- mark-and-sweep – problém s fragmentací
- alternativní řešení:
- paměť rozdělena na dva stejně velké bloky (aktivní, neaktivní)
- při úklidu se objekty přesunují z aktivní části do neaktivní (nové místo)
- po přesunu všech živých objektů dojde k prohození aktivní a neaktivní části
- problém: přeuspořádání ukazatelů
- výhoda: odstranění fragmentace + pracuje se jen s živými objekty (cache, TLB)
- kombinace obou: mark-and-compact

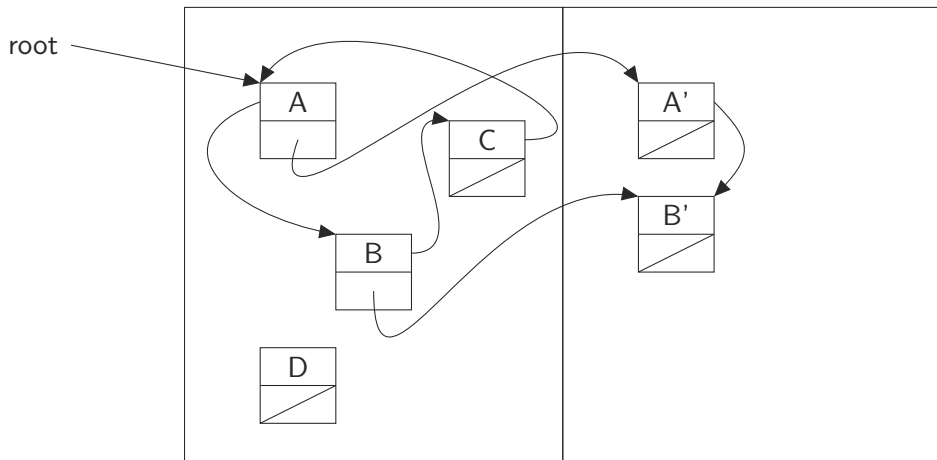
Kopírující GC (ilustrace)



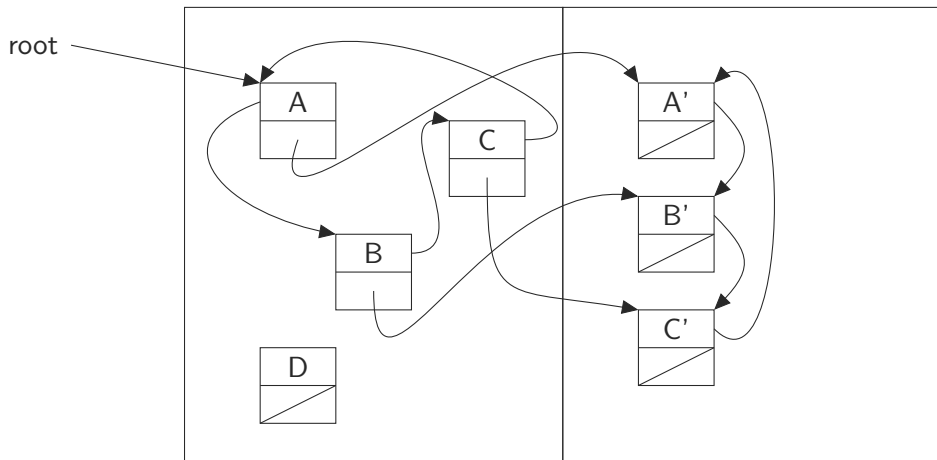
Kopírující GC (ilustrace)

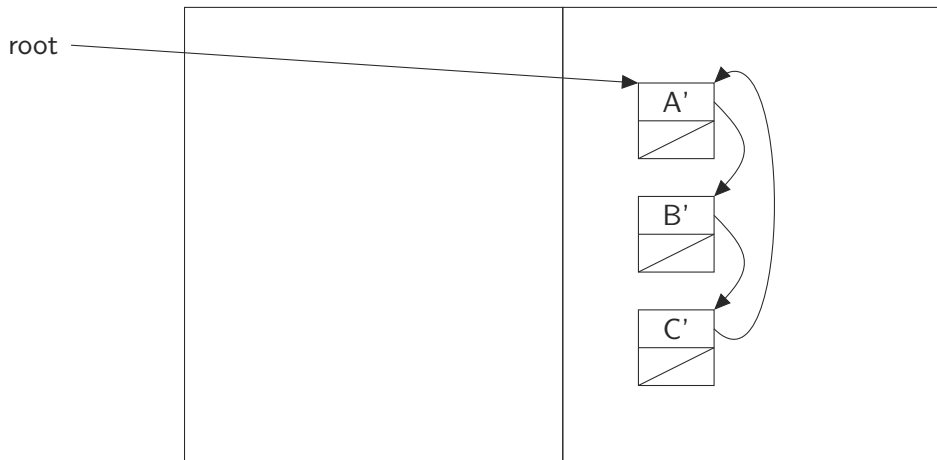


Kopírující GC (ilustrace)



Kopírující GC (ilustrace)





Java

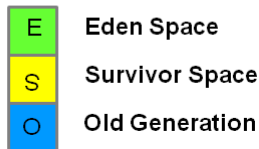
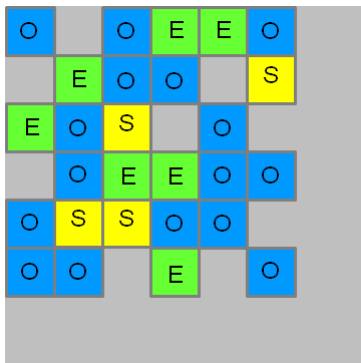
- young generation
 - eden
 - 2x survivor space (vždy jeden volný)
- tenured generation
- PermGen

Alokace

- objekty alokovány v edenu
- dojde-li k zaplnění edenu, spustí GC nad generací Young
- objekty, jež jsou v survivor space delší dobu, jsou přesunuty do tenured generation
- objekty, které jsou živé v edenu a survivor space, jsou přesunuty do volného survivor space

G1 (garbage first)

- paměť rozdělena na menší úseky s podobným významem jako v předchozí variantě
- v mark fázi jsou označeny živé objekty a určí se zaplnění jednotlivých úseků
- úseky, kde je málo živých objektů, jsou uvolněny přesunutím do zaplněnějších úseků
- varianta mark-and-compact, eliminuje přesuny objektů, přesuny je možné dělat paralelně



.NET

- tři generace (0, 1, 2)
- objekty alokovány vždy v 0 a postupně jsou přesouvány výš
- varianta mark-and-compact

Finalizace

- samostatná fronta objektů ke zrušení (v případě .NETu krok po označení živých objektů)
- metoda `finalize()` – volána před uvolněním objektu z paměti.
- není zaručeno, kdy bude daná metoda vůbec zavolána (v JVM nemusí být zavolána vůbec)
- raději nepoužívat

Manuální

- režie (CPU): malloc
- uniklá paměť (Valgrind)
- používání uvolněné paměti, dvojité uvolnění
- možná finalizace objektů
- neexistuje problém se stop-the-world

Automatická

- rychlejší: malloc
- pomalejší uvolnění paměti (někdy může být výhodné uvolňovat objekty naráz)
- paměť může taky „unikat“ (statické proměnné)
- problém s finalizací objektů
- nedeterminismus

- práce se zásobníkem je rychlejší než s haldou
- hodnotové typy C++/C#; možné ovlivnit umístění (zásobník \times heap)
- např. v Javě místo vytvoření objektu nelze kontrolovat (Heap?)

Escape analysis

- analýza, jestli daný objekt/ukazatel může „uniknout“ z daného kontextu (např. funkce)
- záležitost překladače
- možnost převést alokaci na heapu na zásobník
- možnost rozbít objekt hodnot na primitivní datové typy

Regiony/Arény/Pooly

- region/aréna – alokuje se velký blok paměti
- přiděluje se místo z tohoto bloku (může být požadovaná jednotná velikost objektu)
- předpokládá se, že objekty budou uvolněny společně
- menší režie na alokaci i uvolnění než v případě malloc/free

Pozor na předčasné optimalizace!!!