Operační systémy 2

Aplikace a práce s pamětí

Petr Krajča

Katedra informatiky
Univerzita Palackého v Olomouci

18. říjen, 2010

Motivace

- aplikace pracují s pamětí jinak než OS $\Longrightarrow$ 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 $\Longrightarrow$ běhové prostředí (JVM, CLR), standardní knihovna (libc – dlmalloc, 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.

Alokátor v GNU/(Linuxu): ptmalloc

- 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 $\Longrightarrow$ zdrojový kód
- rozhraní poskytující operace `malloc`, `free`, `realloc`, ...
- získává souvislý blok paměti přes volaní OS `brk/sbrk` (mění velikost datového segmentu o daný počet bytů)

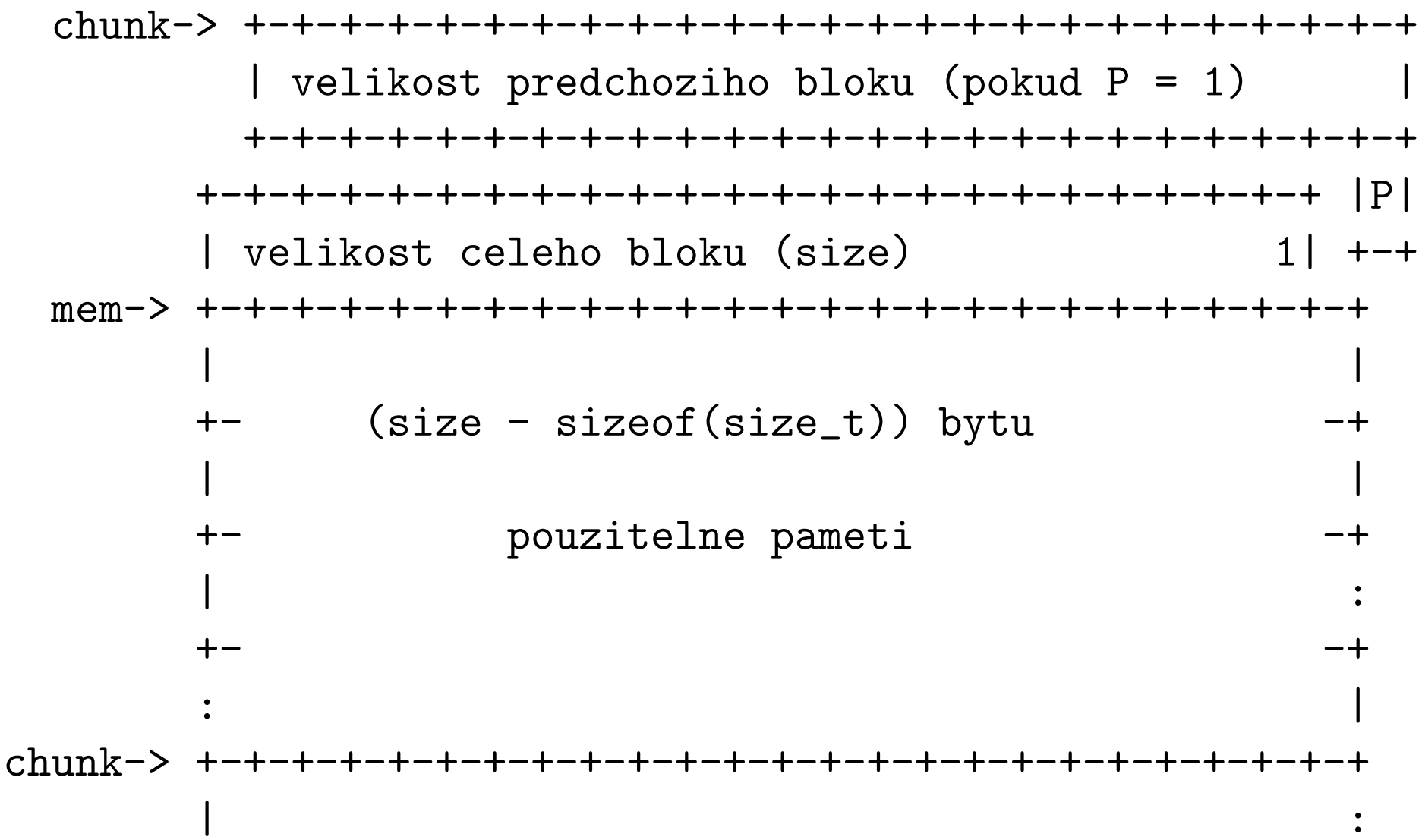
```
void * simple_malloc(int size) {
    return sbrk(size);
}
```

- uvolnění přes `sbrk(-velikost)` $\Longrightarrow$ jednoduchý, ale fragmentuje paměť
- (velké bloky) případně `mmap` (mapování anonymní paměti, `/dev/zero`)

ptmalloc: základní vlasnosti (1/5)

- 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; u nově alokovaného bloku je vždy 1)
- každý kus paměti začíná na sudém násobku slova
- vrácená paměť začíná na sudém násobku slova
- nejmenší alokovatelný kus paměti na i386 je (16 B), i.e., 12 B pro data
- není rozdíl mezi `malloc(1)` a `malloc(6)`!

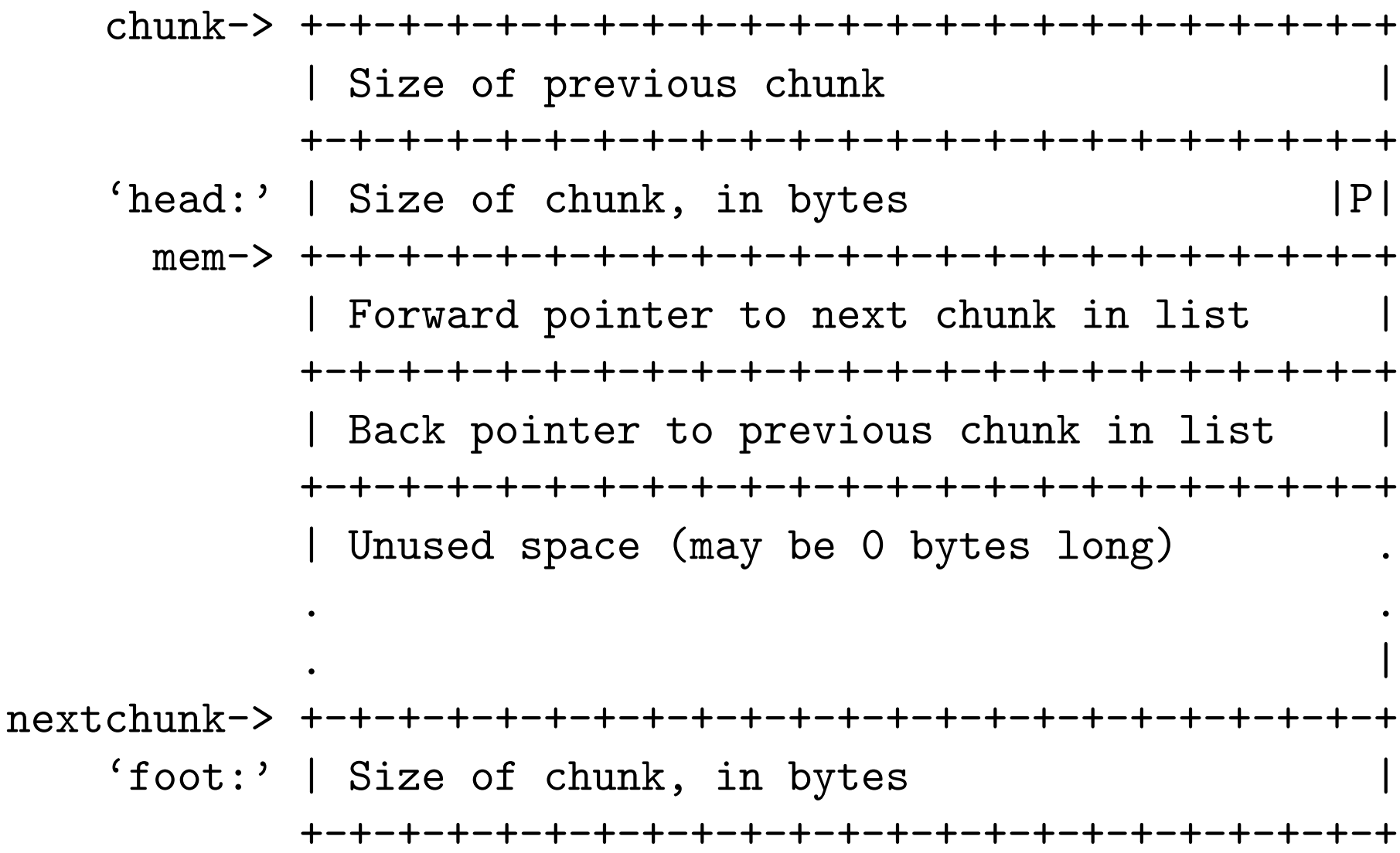
ptmalloc: alokovaný blok (2/5)



ptmalloc: uvolnění paměti a `free` (3/5)

- udržuje si informace o volných blocích
- `malloc` se nejdřív snaží najít první vhodný blok, který byl uvolněn
- 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
- pokud `malloc` vybírá z volných bloků, vybírá z následujících strategií
  - v seznamech: FIFO
  - ve stromech: best-fit

ptmalloc: uvolněný blok do 256 B (4/5)



ptmalloc: ostatní operace (5/5)

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

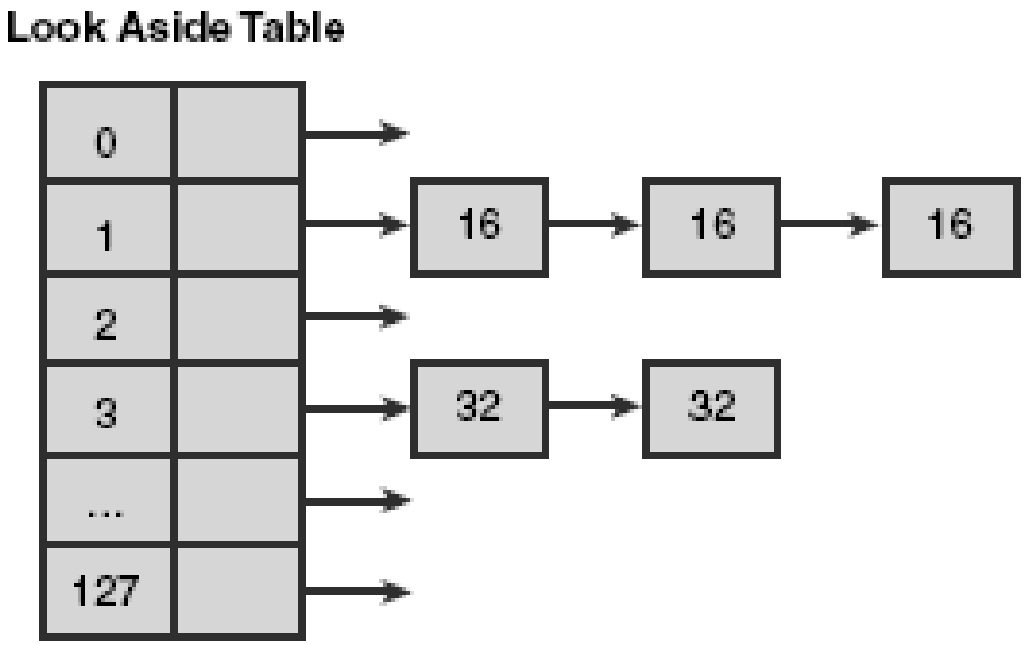
Alokátor ve Windows (1/3)

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

Frontend

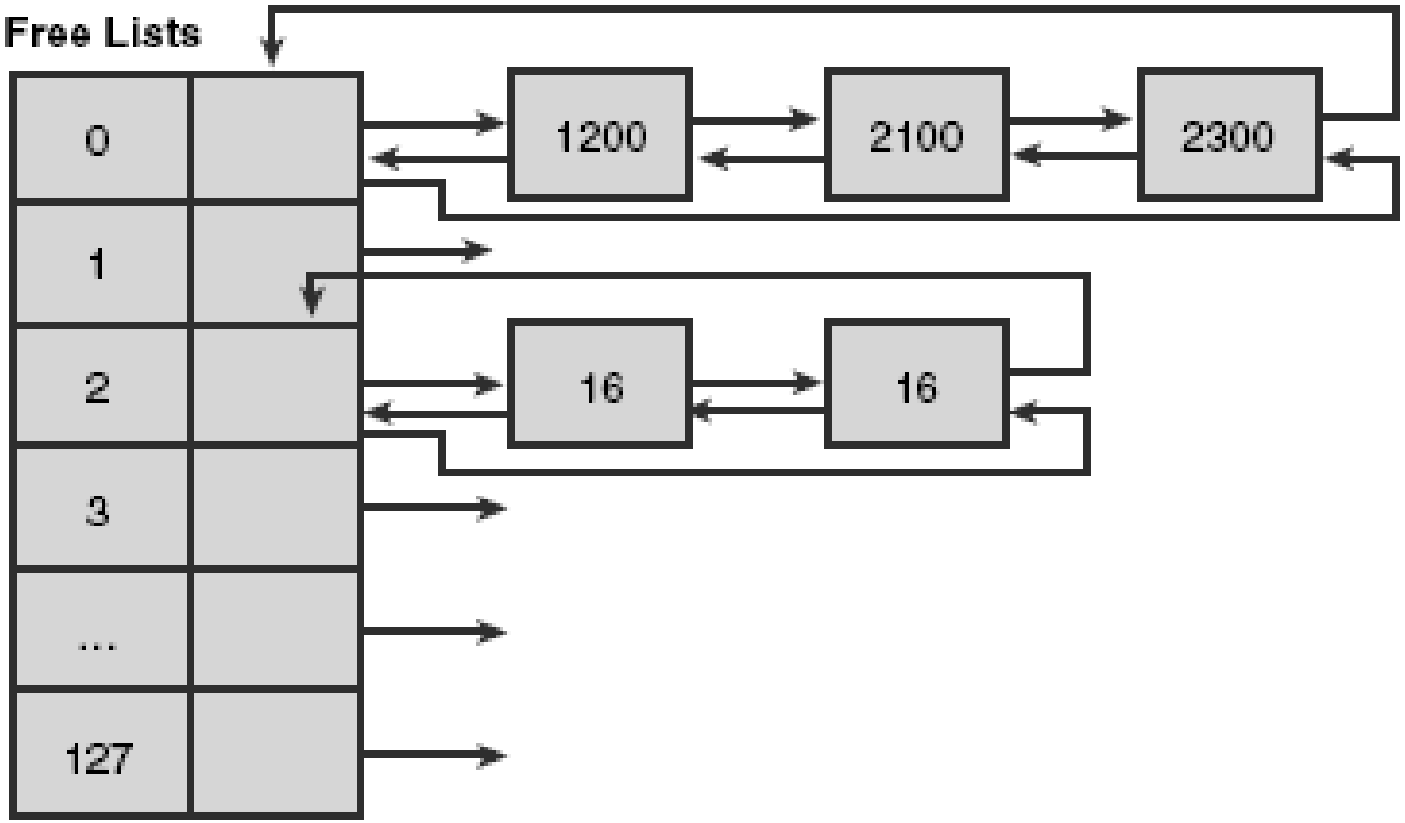
- optimalizace alokací menších bloků
- ,,přichystané bloky k použití”
- LAL (Look-Aside-Lists): 127 seznamů s objekty velikosti $n \times 8$ B
- LF (Low Fragmentation): jako LAL, ale až pro objekty do 16 kB (větší objekty $\Longrightarrow$ větší granularita)
  - objekty 1 B-256 B: zarovnány na 8 B
 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

Alokátor ve Windows (2/3)



Alokátor ve Windows: backend (3/3)

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



Automatická správa paměti

- 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 $\Longrightarrow$ Java, .NET, skriptovací jazyky
- GC lze doplnit i do C/C++
- Boehm(-Demers-Weiser) garbage collector (okamžitá náhrada `GC_MALLOC`)

Počítání odkazů

- každý objekt obsahuje počítadlo, kolik na něj odkazuje objektů
- pokud je na něj vytvořený/zrušný odkaz, zvýší/sníží se počítadlo o 1
- v případě, že je počítadlo 0 $\Longrightarrow$ objekt je uvolněn
- problém: režie (paměť i CPU); cyklické závislosti
- Delphi, PHP, Python, COM

Mark and Sweep

- populární typ GC
- každý objekt má příznak jestli se používá (nastavný na 0)
- v průběhu ,,úklidu"
  - objekty odkazované ze zásobníku, statických proměnných (popř. registrů) $\Longrightarrow$ označeny jako používané
  - objekty odkazované z označených objektů $\Longrightarrow$ označeny
  - neoznačené objekty uvolněny

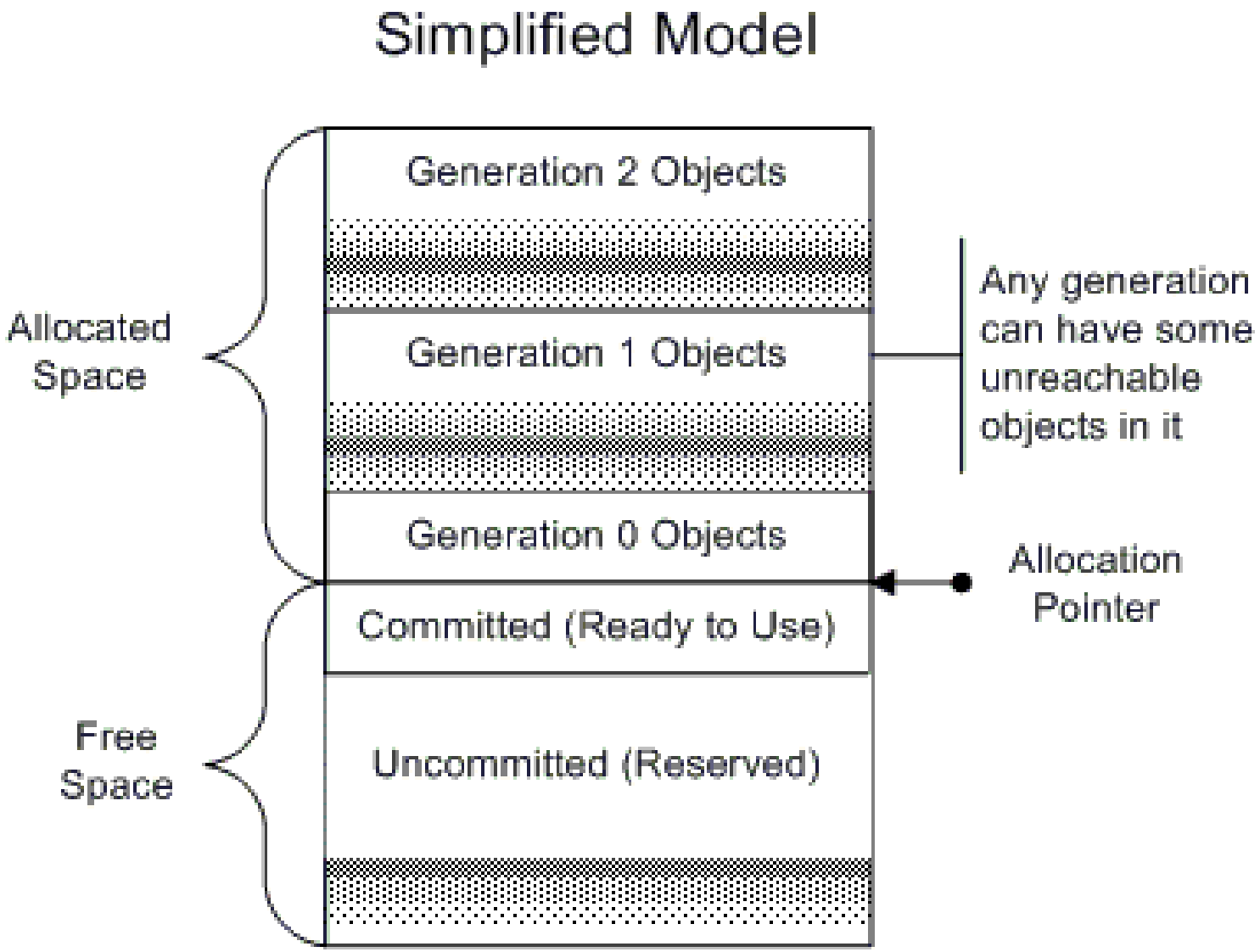
Abstrakce: tři barvy

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

Varianty GC

- přesunující vs. nepřesunující (moving/non-moving)
  - lze přeskupit objekty, aby vytvořily souvislý blok
  - režie na přeskupení/snížení fragmentace$\Longrightarrow$ rychlejší alokace
- generační
  - předpokládá se, že krátce žíjící objekty budou žít krátce
  - paměť rozdělena na generace $\Longrightarrow$ 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 redukují výše zmíněný problém
  - možné vylepšit paralelním zpracováním

Generační GC: ilustrace



Kopírující GC

- 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

Manuální vs. automatická správa paměti

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

Zásobník, Halda, Regiony, Pooly, atd.

- práce se zásobníkem je rychlejší než s haldou
- hodnotové typy C++/C#; možné ovlivnit umístění (zásobník×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!!!

Atomický přístup do paměti

- vícevláknové aplikace/víceprocesorové počítače (+ out-of-order provádění operací)
- obecně přístupy do paměti nemusí být atomické (záležitost CPU)
- klíčové slovo `volatile` – často záleží na překladači; zarovnání přístupů
- *memory barriers* umožňují vynutit si synchronizaci (záležitost CPU)

Atomické operace

- Compare-and-Swap (CAS): ověří jestli se daná hodnota rovná požadované a pokud ano, přiřadí ji novou (CMPXCHG)
- Fetch-and-Add: vrátí hodnotu místa v paměti a zvýší jeho hodnotu o jedna (XADD)
- Load-link/Store-Conditional (LL/CS): načte hodnotu a pokud během čtení nebyla změněna, uloží do ní novou hodnotu
- umožňují implementovat synchronizační primitiva (mutex, kritické sekce, etc.)

Transakční paměť (1/3)

- atomické operace umožňují implementovat bezzámkové (lock-free) datové struktury $\Longrightarrow$ netriviální
- souběžný přístup k paměti $\Longrightarrow$ vývoj vícevláknových aplikací komplikovaný
- vlákna vs. procesy
- problémy se synchronizací (zámky nejsou reentrantní)
- ale: databázové systémy zvládají paralelně pracovat s daty bez vážnějších problémů!!1!
- rozdělení programu na části, které jsou provedeny ,,atomicky" (ACI)

```
void transfer(Account a1, Account a2, int amount) {
    atomic {
        a1.balance += amount;
        a2.balance -= amount;
    }
}
```

Transakční paměť (2/3)

- změny se neprovádí přímo, ale ukládají do logu, v případě ,,commitu" se ověří, jestli došlo ke kolizi
- zjednodušení vývoje – na úkor režie
- omezení: transakci musí být možné provést víckrát

```
void transfer(Account a1, Account a2, int amount) {
    atomic {
        a1.balance += amount;
        a2.balance -= amount;
        launchTheMissiles();
    }
}
```

- možné explicitně ovládat transakce `retry`, `orElse`
- problém: jak se vypořádat s hodnotami mimo transakční paměť (např. existující knihovny)

Transakční paměť (3/3)

- různé varianty a implementace
- podpora HW pro transakční paměť $\Longrightarrow$ omezená až žádná (processor ROCK? [SUN])
- podporat na straně softwaru (Software Transactional Memory – STM)
- podpora jako knihovny/rozšíření Javy/C#/C++
- výzkum Haskell [Microsoft], Fortress/DSTM [SUN]
- jazyky zaměřené na paralelní zpracování, např. Clojure