Alokace paměti pro procesy

* paměť pro programy vytvořené v imperativních programovacích jazycích je poskytována následovně:
  - paměť pro statické proměnné je vyhrazena při spuštění programu
  - lokální proměnné procedur a funkcí jsou alokovány na zásobníku (viz předmět KIV/FJP)
  - dynamická alokace paměti je realizována v oblasti zvané hromada (heap) pomocí služeb operačního systému

* procesy běží (v současných OS) ve virtuální paměti
  - na 32 bitovém systému fy Intel může každý proces adresovat 2 nebo 3 GB virtuální paměti (zbytek do 4 GB je vyhrazen pro OS)
  - procesy nemohou mít mapovánu všechnu paměť, kterou mohou adresovat (protože ani disky nejsou neomezeně velké)
  - po spuštění procesu je vytvořeno mapování (tabulka stránek) pouze pro kód procesu, pro statické proměnné a pro počáteční zásobník
  - při požadavku procesu na zvětšení hromady musí proces požádat operační systém o namapování další paměti
  - proces nežádá o paměť přímo, ale používá k tomu knihovní fce - tzv. alokátory paměti

Explicitní správa paměti

* v jazycích jako je C, C++ nebo Pascal musí být paměť alokována a uvolňována explicitně (fce malloc() a free() v C, klíčová slova new a delete v C++)
  - alokátor paměti spravuje hromadu, alokace paměti je prováděna např. metodou first fit
  - pokud není možné alokovat paměť ze současné hromady, alokátor požádá o mapování dalšího úseku virtuální paměti, část nového úseku přidělí
* problémy s explicitní alokací/dealokací
  - poměrně snadné, pokud je paměť používána pouze uvnitř jedné fce
  - problém rozsáhlejších programů - uvolnit paměť ve chvíli, kdy už není zapotřebí - vede k chybám dlouhodobě běžících programů
  - proto mechanismy pro automatickou správu paměti

Čítání referencí

* čítání referencí (angl. reference counting) používáno vysokoúrovňovými programovacími jazyky, jako např. Perl nebo Python
* implementace:
  - všechny datové struktury obsahují položku, která obsahuje informaci o počtu referencí (odkazů, ukazatelů) na tuto datovou strukturu
  - při vytvoření lokálního odkazu (např. při předání parametru proceduře) se počet odkazů o jeden zvýší, při zrušení lokálního odkazu (např. při ukončení procedury) se sníží
  - při snížení počtu odkazů se testuje, zda je výsledek 0; pokud je, paměť obsazená datovou strukturou se uvolní, protože jí už nikdo nepoužívá
  - počty referencí udržuje automaticky interpret programovacího jazyka

* výhody
  - programátor se nemusí zabývat dealokací paměti
  - jednoduchá implementace
* nevýhody
  - každé vytvoření a zrušení odkazu stojí čas
  - pokud je datová struktura cyklická, nebude uvolněna ani v případě, že na ní již neexistuje žádný odkaz (”mrtvé cykly”)

Garbage collection

* garbage collection (česky: sbírání smetí; dále jen GC) je automatická detekce a uvolnění paměti, která již není odkazována
* běžně implementováno samostatným vláknem, které je spuštěno, pokud dostupná paměť poklesne pod určitý limit

* pro správnou činnost potřebuje GC rozumět obsahu datových struktur - proto bývá nejčastěji součástí virtuálních strojů (JVM, CLR)

* výhoda:
 - nemusíte se zabývat dobou života dat a objektů (ani cyklické odkazy nejsou problém)
* nevýhody:
 - správa paměti je plně v režii virtuálního stroje, nemáte příliš možností jí ovlivnit (kromě explicitního spuštění GC); o dealokaci objektu se ani nedozvíte
 - GC se může aktivovat v nevhodných okamžicích (problém zvláště pro RT aplikace)
 - nepoužívaná paměť může stále ještě zůstat alokována, pokud nenastavíte nepoužívaný odkaz na null

Algoritmus mark-and-sweep
....

* paměť je procházena ve dvou průchodech
 - první průchod - vyhledává dostupnou paměť
  . vyhledávání začíná od dat, která jsou s jistotou procesu dostupná - lokální proměnné procedur a funkcí, globální proměnné (tzv. kotevní objekty)
  . pokud data obsahují odkazy (ukazatele), postupuje dále na odkazovaná data
  . všechny nalezené objekty jsou označeny (např. nastaven bit ”objekt je dostupný”)
 - druhý průchod - neoznačená alokovaná paměť je považována za ”smetí” a je uvolněna

* hlavní nevýhoda algoritmu mark-and-sweep: práce GC způsobuje nárazově citelné zpomalení systému

Baker collector
....

* jeho inkrementální verze řeší hlavní nevýhodu algoritmu mark-and-sweep (Baker 1978)
 - paměť rozděluje do dvou částí, nazvaných semiprostory
 - jeden z nich je označen jako aktivní, jsou v něm vytvářeny všechny nové objekty
 - jednou za čas GC označí jako aktivní druhý semiprostor
  . před otevřením druhého semiprostoru projde původní semiprostor podobným mechanismem jako při mark-and-sweep, ale místo označení objekt evakuuje do nového semiprostoru
  . na místě původního objektu zanechá tzv. náhrobní kámen s novou adresou objektu (pro případ, že by se na objekt někdo chtěl odkazovat)
  . na konci průchodu zůstane v původním semiprostoru pouze smetí, které může být zrušeno
 - výsledkem je čistý semiprostor => umožňuje rychlou alokaci
 - pokud dovolíme nové alokace před úplnou evakuací starého semiprostoru, musíme ihned přesunout všechny případné objekty odkazované z nově alokovaného objektu
* výsledný algoritmus může být inkrementální => řeší problém nárazové aktivace algoritmu mark-and-sweep

* nevýhoda: neustále přesouvá objekty, což je drahé
* řeší generační GC - např. v CLR (.NET) je třígenerační GC
 - mnoho objektů má krátkou dobu života, některé ale ne
 - při přesunu objektu je zvětšeno číslo generace objektu, pokud objekt přežije N generací, je přesunut do privilegované oblasti, kde není (tak často) zpracováván GC

❇