2.5. DYNAMICKÁ ALOKACE PAMĚTI

VIDĚLI JSME, ŽE při práci s odkazy nemusí být datové struktury uloženy v paměti sekvenčně; několik různých tabulek se tak může v oblasti společného fondu nezávisle zvětšovat a zmenšovat. V našich úvahách jsme ale vždy mlčky předpokládali, že všechny uzly mají stejnou velikost – tedy že každý z uzlů obsahuje jistý pevně daný počet paměťových buněk.

Pro celou řadu aplikací můžeme najít vhodný kompromis a pro všechny struktury použít skutečně uzly o jednotné velikosti (viz například cvičení 2). Pokud bychom jednoduše vzali maximální velikost, znamenalo by to u menších uzlů plýtvání; proto raději bereme menší velikost uzlů a používáme něco jako klasickou *filozofii spojové paměti*: „Pokud tu není místo na potřebné informace, dáme je někam jinam a sem vložíme jen odkaz.“

Pro celou řadu jiných aplikací je ale jednotná velikost uzlů nevhodná; často potřebujeme v jedné společné paměťové oblasti mít uzly o různé velikosti. Jinak řečeno: potřebujeme algoritmy pro rezervaci a uvolňování paměťových bloků proměnné velikosti z většího celkového fondu, přičemž každý z bloků je tvořen jistým spojitým intervalem paměťových pozic. Těmto technikám se obvykle říká algoritmy *dynamické alokace paměti*.

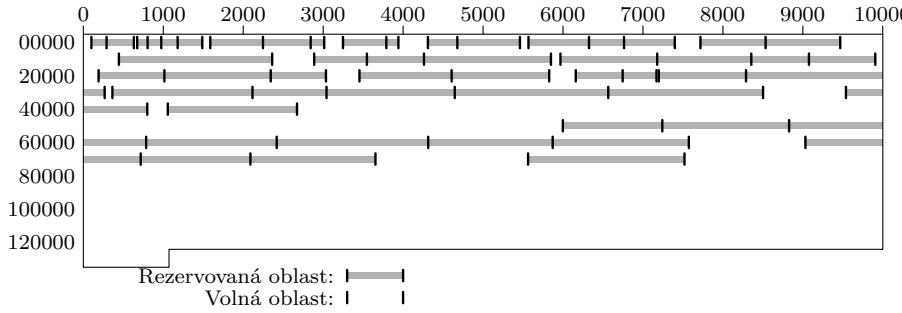
Někdy, například v simulačních programech, potřebujeme častěji dynamickou alokaci uzlů o poměrně malé velikosti (třeba 1–10 slov); jindy, například v operačních systémech, pak pracujeme častěji s velkými bloky informací. Tyto dva různé pohledy vedou také k poněkud odlišným přístupům k dynamické alokaci, i když obě metody mají hodně společného. Pro jednotnost názvosloví budeme tedy v této části textu označovat množinu spojitých paměťových pozic spíše pomocí pojmů *blok* a *oblast*, nikoli „uzel“.

Kolem roku 1975 začali někteří autoři nazývat fond dostupné paměti „halda“ (též hromada, heap). V této sérii knih budeme ale toto slovo používat jen v klasičtějším slova smyslu, který souvisí s prioritními frontami (viz část 5.2.3).

A. Rezervace. Na obrázku 42 vidíme typickou *mapu paměti*, nazývanou někdy „šachovnice“ (checkerboard), která znázorňuje aktuální stav nějakého paměťového fondu (pool). V tomto případě je paměť rozdělena do 53 bloků, které jsou „rezervované“ neboli obsazené, a dále do 21 bloků, které jsou „volné“ neboli „dostupné“, tedy nevyužité. Po nějaké době činnosti dynamické alokace paměti bude paměť počítače vypadat zřejmě jinak. Naším prvním problémem je odpovědět na dvě otázky:

a) Jak bude toto dělení dostupného prostoru reprezentováno uvnitř počítače?
b) Je-li dána takováto reprezentace dostupného prostoru, jaký je vhodný algoritmus, který nalezne spojitý blok n volných pozic a rezervuje je?

Odpověď na otázku (a) samozřejmě zní, že je potřeba někde udržovat *seznam* dostupného prostoru; ten budeme téměř vždy udržovat přímo v *samotném* volném prostoru. (Výjimkou je alokace místa v diskovém souboru nebo v jiné paměti s nejednotnou dobou přístupu, ke které je proto vhodné udržovat adresář volného místa odděleně.)



Obr. 42. Mapa paměti

To znamená, že dostupné segmenty paměti můžeme *spojit dohromady*: první slovo každé volné oblasti paměti bude obsahovat velikost daného bloku a adresu další volné oblasti. Jednotlivé volné bloky spojíme v rostoucím nebo klesajícím pořadí velikosti, v pořadí paměťových adres anebo v jiném, v podstatě náhodném pořadí.

Uvažujme například obrázek 42, který ilustruje paměť se 131 072 slovy, adresovanými pomocí čísel od 0 do 131 071. Pokud bychom propojili dostupné bloky v pořadí paměťových pozic, stačila by jedna proměnná `AVAIL`, která by ukazovala na první volný blok (v tomto případě by se `AVAIL` rovnalo 0), a další bloky bychom reprezentovali takto:

pozice	`SIZE`	`LINK`	
0	101	632	
632	42	1488	
⋮	⋮	⋮	[17 podobných položek]
73654	1909	77519	
77519	53553	Λ	[speciální značka pro poslední odkaz]

Pozice 0–100 tak tvoří první dostupný blok, za rezervovanými oblastmi 101–290 a 291–631 podle obrázku 42 se nachází další volný prostor v pozicích 632–673 atd.

Co se týče otázky (b), potřebujeme-li vyhradit spojitou oblast n slov, musíme určitě najít nějaký blok $m \geq n$ dostupných slov a jeho velikost zmenšit na $m - n$. (Navíc, pokud je $m = n$, musíme blok odstranit ze seznamu volného místa.) Bloků o n a více buňkách může být i několik, a naskýtá se tedy otázka, *kterou* z oblastí nakonec vybrat?

Na tuto otázku se nabízejí dvě základní odpovědi: buďto můžeme použít *metodu best-fit* (nejlepší vhodný blok), nebo *metodu first-fit* (první volný blok). V prvním případě se rozhodneme vybrat oblast s m buňkami, kde m je nejmenší nalezená hodnota větší nebo rovna n. K tomu někdy musíme nejprve prohledat celý seznam dostupného prostoru. Metoda first-fit vybírá oproti tomu jednoduše první vhodnou oblast, která má $\geq n$ slov.

Z historického pohledu se metoda best-fit po několik let hojně používala; přirozeně se zdá být vhodnou metodou, protože si tím větší dostupné oblasti

ušetříme na později, až je budeme potřebovat. Vůči této metodě můžeme ale vznést několik námitek: Je poměrně pomalá, protože při ní musíme provést dosti zdlouhavé hledání; jestliže nejlepší vhodný blok není oproti prvnímu vhodnému podstatně lepší, nestojí tato námaha s hledáním za to. Ještě horší ale je, že u metody best-fit se zvyšuje počet velmi malých bloků, což je obvykle nežádoucí. Za jistých situací je technika first-fit prokazatelně lepší než best-fit; představme si například, že máme k dispozici jen dvě volné oblasti paměti o velikosti 1300 a 1200 slov, a uvažujme dále požadavky bloků o velikosti 1000, 1100 a 250 slov:

požadavek paměti	*dostupné oblasti, first-fit*	*dostupné oblasti, best-fit*
—	1300, 1200	1300, 1200
1000	300, 1200	1300, 200
1100	300, 100	200, 200
250	50, 100	není

(1)

(Ve cvičení 7 nás ale čeká opačný příklad.) Podstatný závěr je, že žádná z metod zjevně nevítězí nad druhou, a proto můžeme doporučit jednodušší metodu first-fit.

Algoritmus A (*Metoda first-fit*). Nechť `AVAIL` ukazuje na první dostupný blok paměti a předpokládejme, že každý dostupný blok s adresou `P` má dvě pole: `SIZE(P)`, počet slov v bloku, a `LINK(P)`, odkaz na další dostupný blok. Poslední ukazatel je roven Λ. Algoritmus vyhledá a rezervuje blok `N` slov, nebo oznámí chybu.

A1. [Inicializace.] Přiřaďte `Q` $\leftarrow$ `LOC(AVAIL)`. (V celém algoritmu pracujeme se dvěma ukazateli, `Q` a `P`, které jsou obvykle svázány podmínkou `P` $=$ `LINK(Q)`. Dále předpokládáme, že `LINK(LOC(AVAIL))` $=$ `AVAIL`.)

A2. [Konec seznamu?] Přiřaďte `P` $\leftarrow$ `LINK(Q)`. Pokud je `P` $= \Lambda$, algoritmus končí neúspěšně; v paměti není volný spojitý blok o `N` slovech.

A3. [Je `SIZE` dostatečná?] Pokud je `SIZE(P)` $\geq$ `N`, jděte na A4; jinak přiřaďte `Q` $\leftarrow$ `P` a vraťte se na krok A2.

A4. [Rezervace `N`.] Přiřaďte `K` $\leftarrow$ `SIZE(P)` $-$ `N`. Pokud je `K` $= 0$, přiřaďte `LINK(Q)` $\leftarrow$ $\leftarrow$ `LINK(P)` (a tím odstraňte prázdnou oblast ze seznamu); jinak přiřaďte `SIZE(P)` $\leftarrow$ `K`. Algoritmus končí úspěšně, protože rezervoval oblast délky `N` počínaje pozicí `P` $+$ `K`. ∎

Tento algoritmus je jasný na první pohled. Mírnou změnou jeho strategie můžeme ale dosáhnout významného urychlení. Toto zlepšení je velmi důležité a pro čtenáře bude jistě potěšením, pokud mu tajemství neprozradíme a bude smět na řešení přijít sám (viz cvičení 6).

Algoritmus A můžeme využít při alokaci paměti jak s malým, tak i velkým `N`. Uvažujme ale nyní na chvíli, že nás zajímají především *velké* hodnoty `N`. Všimněte si nyní, co se v algoritmu stane, pokud je `SIZE(P)` rovno `N` $+ 1$: dostaneme se do kroku A4 a zmenšíme `SIZE(P)` na 1. Jinými slovy, vytvořili jsme dostupný blok o velikosti 1; tento blok je natolik malý, že je prakticky k ničemu a že jen zbytečně „zaneřádí“ paměť. Lepší by bylo rezervovat celý blok `N` $+ 1$ slov a jedno

přebývající slovo neušetřit; často je skutečně vhodné několik slov paměti ztratit a nemuset se zabývat nepodstatnými detaily. Podobné úvahy platí i pro bloky `N` + `K` slov, pokud je `K` velmi malé.

Povolíme-li možnost rezervace o něco více než `N` slov, budeme si muset navíc pamatovat, kolik slov jsme skutečně rezervovali, a při pozdějším uvolnění bloku uvolnit celou množinu `N` + `K` slov. Tato dodatečná režie znamená, že musíme obsadit určitý prostor v *každém* bloku, ale systém bude efektivnější jen za jistých okolností a za velmi těsné shody; strategie se tedy nezdá být příliš atraktivní. Zavedení speciálního *řídícího slova* na první pozici každého bloku s proměnnou velikostí se ale ukazuje výhodné ještě z dalších důvodů a obvykle je docela rozumné zapisovat pole `SIZE` do prvního slova všech bloků, ať už volných nebo rezervovaných.

V souladu s těmito dohodami můžeme krok A4 upravit takto:

A4′. [Rezervace $\geq$ `N`.] Přiřaďte `K` $\leftarrow$ `SIZE(P)` $-$ `N`. Pokud je `K` $< c$ (kde c je malá kladná konstanta a vyjadřuje množství paměti, které jsme ochotni obětovat v zájmu úspory času), přiřaďte `LINK(Q)` $\leftarrow$ `LINK(P)` a `L` $\leftarrow$ `P`. Jinak přiřaďte `SIZE(P)` $\leftarrow$ `K`, `L` $\leftarrow$ `P` + `K`, `SIZE(L)` $\leftarrow$ `N`. Algoritmus končí úspěšně, protože rezervoval oblast délky `N` nebo více, počínaje pozicí `L`.

Doporučená hodnota konstanty c je okolo 8 až 10, i když pro srovnání této hodnoty s jinými není k dispozici příliš mnoho teoretických či empirických důkazů. U metody best-fit je test podmínky `K` $< c$ ještě *důležitější* než u metody first-fit, protože je zde vyšší pravděpodobnost vzniku malých bloků (s menšími hodnotami `K`); pro činnost algoritmu je ale vhodné, aby byl počet dostupných bloků co nejmenší.

B. Uvolnění. Nyní uvažujme opačný problém: Jak do seznamu volného (dostupného) místa vrátit bloky, které již nepotřebujeme?

Na pohled je lákavé vyřešit problém pomocí techniky uvolňování paměti (garbage collection) z části 2.3.5; můžeme tedy jednoduše nedělat nic, dokud nevyčerpáme veškerý volný prostor, a potom prohledat všechny momentálně obsazené oblasti a vytvořit nový seznam `AVAIL`.

Myšlenku uvolňování paměti nelze ale doporučit pro všechny aplikace. Za prvé totiž potřebujeme dosti „disciplinovaně“ pracovat s ukazateli, máme-li garantovat, že všechny momentálně obsazené oblasti snadno najdeme, a tato disciplína v uvažovaných aplikacích často chybí. Za druhé, jak jsme již viděli, uvolňování paměti je při téměř zaplněné paměti dosti pomalé.

Uvolňování paměti ale není uspokojivé ještě z jednoho důvodu, a sice kvůli jevu, se kterým jsme se v předchozím výkladu této techniky nesetkali: Předpokládejme, že máme dvě sousední oblasti paměti, které jsou obě volné, ale vzhledem k filozofii uvolňování paměti není jedna z nich (naznačena stínováním) v seznamu `AVAIL`.

(2)

V tomto diagramu jsou tmavě stínované oblasti úplně vlevo a úplně vpravo nedostupné (obsazené). Nyní rezervujeme část volné oblasti:

(3)

Proběhne-li v této chvíli uvolňování paměti, máme dvě oddělené volné oblasti:

(4)

Hranice mezi dostupnou a rezervovanou oblastí se často „zakopávají“ natrvalo a s postupem času se tato situace výrazně zhoršuje. Pokud bychom ale přijali jinou filozofii a vraceli bloky do seznamu `AVAIL` ihned po jejich uvolnění a pokud bychom *slučovali sousední volné oblasti*, změnilo by se schéma (2) na

(5)

a po dalších dvou krocích bychom dostali

(6)

což je podstatně lepší než (4). Díky tomuto jevu zanechává technika uvolňování paměti za sebou paměť více rozdrobenou, než by měla být.

Pro potlačení tohoto nepříjemného jevu můžeme spolu s uvolňováním paměti využívat takzvané *zhuštění paměti*, při kterém přesuneme všechny rezervované bloky do spojité oblasti pozic, takže po dokončení operace uvolňování paměti jsou i všechny dostupné bloky sloučeny. Alokační algoritmus je nyní ve srovnání s Algoritmem A zcela triviální, protože máme vždy dostupný pouze jeden ucelený blok. V této technice je sice časově náročné kopírování všech obsazených pozic a změna hodnoty příslušných polí s odkazy, ale při disciplinované práci s ukazateli to můžeme zvládnout s rozumnou mírou efektivity; v každém bloku musí být k dispozici volné pole odkazu, které využijeme v algoritmech uvolňování. (Viz cvičení 33.)

Protože mnohé aplikace požadavkům reálné proveditelnosti uvolňování paměti nevyhovují, podíváme se nyní na metody pro vracení bloků paměti do seznamu volného místa. Jediný problém v těchto metodách představuje zmíněné zhuštění: dvě sousední volné oblasti musíme sloučit do jedné. Při uvolnění oblasti obklopené z obou stran volnými bloky musíme dokonce sloučit všechny tři popsané oblasti. *Takto může být paměť v dobré rovnováze, a to i když různé oblasti paměti rezervujeme a uvolňujeme po poměrně dlouhou dobu.* (Jako důkaz tohoto tvrzení se podívejte na níže uvedené „pravidlo 50 procent“.)

Problém je stanovit, jestli jsou oblasti po obou stranách navráceného bloku momentálně volné, a pokud jsou volné, provést odpovídající aktualizaci seznamu `AVAIL`. Druhá operace je o něco obtížnější, než by se zdálo.

Prvním řešením uvedených problémů je udržovat seznam `AVAIL` v pořadí rostoucích paměťových pozic.

Algoritmus B (*Uvolňování se seřazeným seznamem*). Tento algoritmus pracuje s předpoklady Algoritmu A a s dodatečným předpokladem, že je seznam `AVAIL` seřazen podle paměťových pozic (tedy pokud `P` ukazuje na dostupný blok a `LINK(P)` $\neq \Lambda$, je `LINK(P)` $>$ `P`), a přidává uvolněný spojitý blok `N` buněk

začínající na pozici P0 do seznamu AVAIL. Přirozeně předpokládáme, že žádná z těchto buněk N není dosud mezi dostupnými (volnými).

B1. [Inicializace.] Přiřaďte Q ← LOC(AVAIL). (Viz poznámky v kroku A1 výše.)

B2. [Posun P.] Přiřaďte P ← LINK(Q). Pokud je P = Λ nebo pokud je P > P0, jděte na B3; jinak přiřaďte Q ← P a opakujte krok B2.

B3. [Kontrola horní hranice.] Pokud je P0 + N = P a P ≠ Λ, přiřaďte N ← ← N + SIZE(P) a dále přiřaďte LINK(P0) ← LINK(P). Jinak přiřaďte LINK(P0) ← P.

B4. [Kontrola dolní hranice.] Pokud je Q + SIZE(Q) = P0 (předpokládáme, že

$$\texttt{SIZE(LOC(AVAIL))} = 0,$$

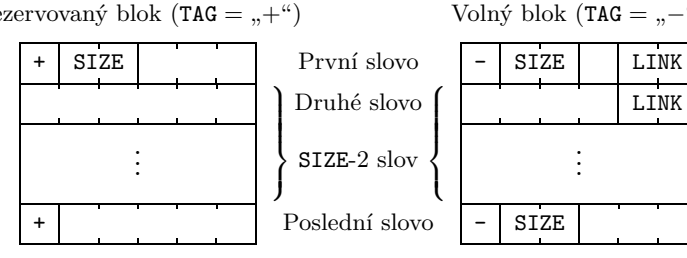
takže pro Q = LOC(AVAIL) tento test vždy selže), přiřaďte SIZE(Q) ← ← SIZE(Q) + N a LINK(Q) ← LINK(P0). Jinak přiřaďte LINK(Q) ← P0, SIZE(P0) ← N. ∎

Kroky B3 a B4 provádějí požadované zhuštění a vycházejí z toho, že ukazatele Q < P0 < P definují počáteční pozice tří po sobě jdoucích dostupných oblastí.

Čtenář jistě snadno nahlédne, že pokud neudržujeme seznam AVAIL v pořadí paměťových pozic, musíme zhuštění paměti provést metodou „hrubé síly“, a tedy přistoupit k úplnému prohledání seznamu AVAIL; v Algoritmu B je tato operace zkrácena na prohledání průměrně zhruba *poloviny* seznamu AVAIL (v kroku B2). Ve cvičení 11 si ukážeme úpravu Algoritmu B, se kterou stačí prohledat průměrně jen asi třetinu seznamu. Je ale jasné, že nad dlouhým seznamem AVAIL jsou všechny tyto metody podstatně pomalejší, než by bylo záhodno. Existuje nějaký jiný způsob rezervace a uvolňování paměťových oblastí, při kterém bychom seznam AVAIL nemuseli tolik prohledávat?

Nyní se podíváme na metodu, která potlačuje veškeré prohledávání při navracení paměti; jak si ukážeme ve cvičení 6, po vhodné úpravě může dokonce potlačit téměř veškeré prohledávání při rezervaci. Technika využívá pole TAG na obou koncích každého bloku a pole SIZE v prvním slově bloku; při rezervaci rozumně velkých bloků paměti je tato režie zanedbatelná, i když u bloků malé průměrné velikosti může být až zbytečně velkou. U jiné metody, kterou si popíšeme ve cvičení 19, stačí jediný bit v prvním slově každého bloku, pouze za cenu mírného prodloužení běhu a o něco komplikovanějšího programu.

Každopádně ale předpokládejme, že nám jisté řídící informace navíc nevadí a že budeme chtít dosáhnout především výraznější úspory času v Algoritmu B při dlouhém seznamu AVAIL. Popsaná metoda předpokládá, že mají jednotlivé bloky následující tvar:



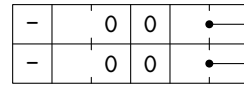
(7)

Myšlenkou následujícího algoritmu je udržování obousměrně propojeného seznamu AVAIL, ve kterém můžeme jednotlivé položky pohodlně odstraňovat z libovolných náhodných částí. Pole TAG na každém konci bloku slouží k řízení procesu zhuštění, protože z něj snadno poznáme, jestli jsou volné i oba sousední bloky.

Obousměrné propojení zajistíme známým způsobem, kdy odkaz LINK v prvním slově bude ukazovat na další volný blok v seznamu a odkaz LINK ve druhém slově ukazuje zpět na předchozí blok; pokud je tedy P adresa dostupného bloku, platí vždy

$$\texttt{LINK(LINK(P)} + 1) = \texttt{P} = \texttt{LINK(LINK(P} + 1)). \tag{8}$$

Pro zajištění správných „hraničních podmínek“ definujeme ještě záhlaví seznamu:



(9)

Rezervační algoritmus typu first-fit napíšeme u této techniky podobně jako Algoritmus A, takže jej v textu neuvádíme (je ve cvičení 12). Podstatnou novinkou je totiž u této metody především uvolnění bloku, které proběhne v podstatě za pevný čas.

Algoritmus C (*Uvolnění s hraničními značkami*). Předpokládejme, že bloky paměťových pozic mají tvar (7) a dále že seznam AVAIL je obousměrně propojený, jak jsme si řekli před chvílí. Tento algoritmus umístí do seznamu AVAIL blok pozic počínaje adresou P0. Pokud je fond volného prostoru umístěn v pozicích od m_0 do m_1 včetně, algoritmus pro jednoduchost předpokládá

$$\texttt{TAG(}m_0 - 1\texttt{)} = \texttt{TAG(}m_1 + 1\texttt{)} = \text{„+“}.$$

C1. [Kontrola dolní hranice.] Pokud je TAG(P0 − 1) = „+“, jděte na C3.

C2. [Odstranění dolní oblasti.] Přiřaďte P ← P0−SIZE(P0 − 1) a potom přiřaďte P1 ← LINK(P), P2 ← LINK(P + 1), LINK(P1 + 1) ← P2, LINK(P2) ← P1, SIZE(P) ← SIZE(P) + SIZE(P0), P0 ← P.

C3. [Kontrola horní hranice.] Přiřaďte P ← P0 + SIZE(P0). Pokud je TAG(P) = = „+“, jděte na C5.

C4. [Odstranění horní oblasti.] Přiřaďte P1 ← LINK(P), P2 ← LINK(P+1), LINK(P1+1) ← P2, LINK(P2) ← P1, SIZE(P0) ← SIZE(P0) + SIZE(P), P ← P + SIZE(P).

C5. [Přidání do seznamu AVAIL.] Přiřaď SIZE(P − 1) ← SIZE(P0), LINK(P0) ← ← AVAIL, LINK(P0 + 1) ← LOC(AVAIL), LINK(AVAIL + 1) ← P0, AVAIL ← ← P0, TAG(P0) ← TAG(P − 1) ← „−“. ∎

Jednotlivé kroky Algoritmu C jsou přímým důsledkem rozložení paměti podle (7); ve cvičení 15 nás čeká o něco delší, ale zároveň i o něco rychlejší algoritmus. V kroku C5 je výraz AVAIL zkratkou za LINK(LOC(AVAIL)), jak vidíme v (9).

C. Systém dvojic. Nyní se podíváme na jiný přístup k dynamické alokaci paměti, který je vhodný pro dvojkové počítače. Metoda potřebuje pro každý

blok jeden bit režie, přičemž všechny bloky u ní musí mít délku 1, 2, 4, 8, 16 atd. slov. Nemá-li určitý blok délku přesně 2^k slov pro vhodné celé k, zvolíme nejblíže vyšší mocninu 2 a podle toho alokujeme i nevyužitý prostor.

Myšlenkou této metody alokace paměti je udržovat samostatné seznamy dostupných bloků o každé velikosti 2^k, $0 \le k \le m$. Celý fond paměťového prostoru se skládá z 2^m slov, které mají adresy od 0 do $2^m - 1$. Na začátku je k dispozici celý blok 2^m slov. Později, pokud potřebujeme rezervovat blok 2^k slov a pokud není k dispozici žádný blok této velikosti, *rozdělíme* větší dostupný blok do dvou rovných částí; nakonec se takto objeví blok správné velikosti 2^k. Velký blok rozdělíme na dvě části o stejné velikosti (poloviny původního bloku), které nazýváme *dvojice*, buddies. Jakmile jsou později oba dílčí bloky volné, sloučíme je opět do jediného bloku; v procesu tak můžeme pokračovat do nekonečna, dokud nevyčerpáme veškerý volný prostor.

Klíčovým momentem praktické použitelnosti této metody je, že pokud známe adresu bloku (tedy paměťovou pozici jeho prvního slova) i jeho velikost, známe také adresu druhého bloku z dvojice. S blokem o velikosti 16, jenž začíná na binární pozici 101110010110000, tvoří například dvojici blok se začátkem na 101110010100000. Všechno si vysvětlíme na činnosti algoritmu, kde *adresa bloku o velikosti* 2^k *je násobkem* 2^k. Jinými slovy, adresa ve dvojkovém (binárním) zápisu má vpravo nejméně k nul. Toto pozorování snadno zdůvodníme indukcí: Platí-li pro všechny bloky o velikosti 2^{k+1}, platí jistě i po rozdělení bloku na poloviny.

To znamená, že například blok o velikosti 32 má adresu tvaru $xx\ldots x00000$ (kde za x reprezentují nuly nebo jedničky); po jeho rozdělení mají nově vytvořené bloky adresy $xx\ldots x00000$ a $xx\ldots x10000$. Obecně nechť $\text{buddy}_k(x) =$ je adresa druhého bloku z dvojice k bloku o velikosti 2^k a adrese x; platí

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, & \text{je-li } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{je-li } x \bmod 2^{k+1} = 2^k. \end{cases} \tag{10}$$

Tuto funkci snadno vypočteme pomocí operace XOR neboli „výlučné nebo“ (nazývané někdy „selektivní doplněk“ či „součet bez přenosu“), která je běžnou součástí dvojkových počítačů; viz cvičení 28.

Systém dvojic využívá v každém bloku jednobitové `TAG`:

$$\begin{aligned} &\texttt{TAG(P)} = 0, && \text{pokud je blok s adresou } \texttt{P} \text{ rezervovaný;} \\ &\texttt{TAG(P)} = 1, && \text{pokud je blok s adresou } \texttt{P} \text{ volný.} \end{aligned} \tag{11}$$

Kromě tohoto pole `TAG`, které je definováno ve všech blocích a do kterého nesmí uživatelé při rezervaci bloků zasahovat, obsahují dále *dostupné* neboli volné bloky dvě pole odkazů `LINKF` a `LINKB`, jež tvoří obvyklé odkazy vpřed a vzad z obousměrně propojeného odkazu; navíc bloky obsahují pole `KVAL`, které definuje hodnotu k určující velikost 2^k. Níže uvedené algoritmy využívají pozice tabulky `AVAIL[0]`, `AVAIL[1]`, ..., `AVAIL[`m`]`, které slouží jako záhlaví seznamů volných paměťových bloků o velikosti 1, 2, 4, ..., 2^m. Tyto seznamy jsou obousměrně propojené, takže záhlaví seznamu obsahuje jako obvykle dva ukazatele (viz část

2.2.5):

$$\begin{aligned}&\texttt{AVAILF}[k] = \texttt{LINKF(LOC(AVAIL}[k]\texttt{))} = \text{odkaz na konec seznamu } \texttt{AVAIL}[k];\\ &\texttt{AVAILB}[k] = \texttt{LINKB(LOC(AVAIL}[k]\texttt{))} = \text{odkaz na čelo seznamu } \texttt{AVAIL}[k].\end{aligned} \tag{12}$$

Na začátku, před alokací prvního bloku, platí

$$\begin{aligned}\texttt{AVAILF}[m] = \texttt{AVAILB}[m] &= 0,\\ \texttt{LINKF(0)} = \texttt{LINKB(0)} &= \texttt{LOC(AVAIL}[m]\texttt{)},\\ \texttt{TAG(0)} = 1, \qquad \texttt{KVAL(0)} &= m\end{aligned} \tag{13}$$

(což znamená jeden dostupný blok o délce 2^m, který začíná na pozici 0) a

$$\texttt{AVAILF}[k] = \texttt{AVAILB}[k] = \texttt{LOC(AVAIL}[k]\texttt{)}, \qquad \text{pro } 0 \le k < m \tag{14}$$

(to znamená prázdné seznamy dostupných bloků délky 2^k pro všechna $k < m$).

Na základě tohoto popisu systému dvojic již čtenář může snadno sám navrhnout vhodné algoritmy pro rezervaci a uvolnění paměťových oblastí a nemusí se rovnou dívat na níže uvedené řešení. Všimněte si, jak relativně snadno se dají bloky v rezervačním algoritmu rozpůlit.

Algoritmus R (*Rezervace v systému dvojic*). Tento algoritmus nalezne a rezervuje blok o 2^k pozicích, nebo oznámí chybu, a to při výše popsaném systému uspořádání paměti s dvojicemi.

R1. [Nalezení bloku.] Nechť j je nejmenší celé číslo z intervalu $k \le j \le m$, pro které je $\texttt{AVAILF}[j] \neq \texttt{LOC(AVAIL}[j]\texttt{)}$, tedy pro které seznam dostupných bloků o velikosti 2^j není prázdný. Pokud takové j neexistuje, algoritmus končí neúspěšně, protože v paměti není žádný dostupný blok odpovídající velikosti k danému požadavku.

R2. [Odstranění ze seznamu.] Nechť $\texttt{L} \leftarrow \texttt{AVAILF}[j]$, $\texttt{P} \leftarrow \texttt{LINKF(L)}$, $\texttt{AVAILF}[j] \leftarrow \texttt{P}$, $\texttt{LINKB(P)} \leftarrow \texttt{LOC(AVAIL}[j]\texttt{)}$, a $\texttt{TAG(L)} \leftarrow 0$.

R3. [Je potřeba dělení?] Pokud je $j = k$, algoritmus končí (našli jsme dostupný blok začínající adresou L a rezervovali jsme jej).

R4. [Dělení.] Zmenši j o 1. Pak přiřaď $\texttt{P} \leftarrow \texttt{L} + 2^j$, $\texttt{TAG(P)} \leftarrow 1$, $\texttt{KVAL(P)} \leftarrow j$, $\texttt{LINKF(P)} \leftarrow \texttt{LINKB(P)} \leftarrow \texttt{LOC(AVAIL}[j]\texttt{)}$, $\texttt{AVAILF}[j] \leftarrow \texttt{AVAILB}[j] \leftarrow \texttt{P}$. (Tím se rozdělí jeden velký blok a nepoužitá polovina se vloží do seznamu $\texttt{AVAIL}[j]$, který byl prázdný.) Vraťte se na krok R3. ∎

Algoritmus S (*Uvolnění v systému dvojic*). Tento algoritmus vrátí blok o 2^k pozicích, které začínají adresou L, do volné paměti, a to při výše popsaném systému uspořádání paměti s dvojicemi.

S1. [Je k dispozici blok z dvojice?] Přiřaďte $\texttt{P} \leftarrow \text{buddy}_k(\texttt{L})$. (Viz (10).) Pokud je $k = m$ nebo pokud je $\texttt{TAG(P)} = 0$, případně pokud je $\texttt{TAG(P)} = 1$ a $\texttt{KVAL(P)} \neq k$, jděte na S3.

S2. [Sloučení s blokem z dvojice.] Přiřaďte

$$\texttt{LINKF(LINKB(P))} \leftarrow \texttt{LINKF(P)}, \qquad \texttt{LINKB(LINKF(P))} \leftarrow \texttt{LINKB(P)}.$$

(Tím odstraníme blok P ze seznamu $\texttt{AVAIL}[k]$.) Potom přiřaďte $k \leftarrow k + 1$, a pokud je $\texttt{P} < \texttt{L}$, přiřaďte $\texttt{L} \leftarrow \texttt{P}$. Vraťte se na S1.

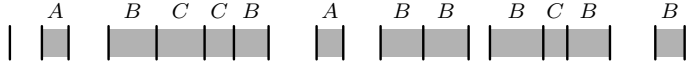
S3. [Uložení do seznamu.] Přiřaďte `TAG(L)` $\leftarrow 1$, `P` $\leftarrow$ `AVAILF[`k`]`, `LINKF(L)` $\leftarrow$ `P`, `LINKB(P)` $\leftarrow$ `L`, `KVAL(L)` $\leftarrow k$, `LINKB(L)` $\leftarrow$ `LOC(AVAIL[`k`])`, `AVAILF[`k`]` $\leftarrow$ `L`. (Tím vložíme blok `L` do seznamu `AVAIL[`k`]`.) ▮

D. Porovnání metod. Matematická analýza těchto algoritmů dynamické alokace paměti se ukazuje být dosti obtížná, ale jeden zajímavý jev se analyzuje poměrně snadno; je to takzvané „pravidlo 50 procent“:

> *Jestliže Algoritmy A a B průběžně voláme takovým způsobem, že je systém v rovnováze s průměrně N rezervovanými bloky, z nichž každý má stejnou pravděpodobnost, že bude uvolněn jako další v pořadí, a kde veličina* `K` *v Algoritmu A nabývá nenulových hodnot (nebo obecněji hodnot $\geq c$ jako v A4′) s pravděpodobností p, pak průměrný počet dostupných bloků je přibližně roven $\frac{1}{2}pN$.*

Toto pravidlo vyjadřuje přibližnou délku seznamu `AVAIL`. Pokud je veličina p blízká 1 – což nastává, pokud je c velmi malé a pokud se velikosti ostatních bloků jen málokdy rovnají – máme zhruba stejně dostupných bloků jako obsazených; odtud také označení „pravidlo 50 procent“.

Zmíněné pravidlo není těžké odvodit. Uvažujme následující mapu paměti:



Rezervované bloky jsou zde rozděleny do tří kategorií:

A: po jejich uvolnění se počet dostupných bloků o jedničku sníží,
B: po jejich uvolnění se počet dostupných bloků nezmění,
C: po jejich uvolnění se počet dostupných bloků o jedničku zvýší.

Nyní nechť N je počet rezervovaných bloků a nechť M je počet volných (dostupných) bloků; dále nechť A, B a C jsou počty bloků jednotlivých popsaných typů. Platí

$$\begin{aligned} N &= A + B + C \\ M &= \tfrac{1}{2}(2A + B + \epsilon) \end{aligned} \qquad (15)$$

kde $\epsilon = 0$, 1 nebo 2, podle podmínek pro horní a dolní hranici.

Předpokládejme nyní, že N je v podstatě konstantní, ale že A, B, C a ϵ jsou náhodné veličiny, které po uvolnění bloku dosahují stacionárního rozdělení a po alokaci bloku (trochu jiného) stacionárního rozdělení. Průměrná změna veličiny M při uvolnění bloku je rovna průměrné hodnotě $(C - A)/N$; průměrná změna M při alokaci bloku je pak $1 - p$. Podle předpokladu rovnovážného stavu je průměrná hodnota $C - A - N + pN$ rovna nule. Potom je ale průměrná hodnota $2M$ rovna pN plus průměrné hodnotě ϵ, protože podle (15) je $2M = N + A - C + \epsilon$. Odtud již vyplývá pravidlo 50 procent.

Náš předpoklad, podle něhož každé odstranění probíhá nad náhodným rezervovaným blokem, platí v případě, že je doba života bloku náhodnou veličinou s exponenciálním rozdělením. Na druhé straně, budou-li mít všechny bloky zhruba stejnou dobu života, předpoklad bude neplatný; John E. Shore poukázal, že bloky typu A mají tendenci více „stárnout“ než bloky typu C, pokud mají alokace a uvolnění charakter zhruba FIFO, protože posloupnost

sousedních rezervovaných bloků bude mít pořadí od nejmladšího k nejstaršímu a naposledy alokovaný blok není téměř nikdy typu A. Tím v paměti vzniká menší počet dostupných bloků a výsledkem je ještě rychlejší zpracování, než kolik by vyplývalo z pravidla 50 procent. [Viz *CACM* **20** (1977), 812–820.]

Podrobnější informace o pravidle 50 procent viz D. J. M. Davies, *BIT* **20** (1980), 279–288; C. M. Reeves, *Comp. J.* **26** (1983), 25–35; G. Ch. Pflug, *Comp. J.* **27** (1984), 328–333.

Kromě tohoto zajímavého pravidla jsou již naše znalosti ohledně rychlosti algoritmů dynamické alokace paměti založeny téměř výhradně na experimentech metodou Monte Carlo. Pro čtenáře bude zajímavé provést si vlastní simulační experimenty a zaměřit je na konkrétní algoritmy alokace paměti pro konkrétní počítač i cílovou aplikaci nebo třídu aplikací. Také autor provedl před napsáním této části textu několik takových experimentů (a skutečně pozoroval pravidlo 50 procent, ještě než pro ně nalezl regulérní důkaz); podívejme se nyní na metody těchto experimentů i jejich výsledky.

Základní simulační program pracoval zhruba takto, přičemž zpočátku byla proměnná `TIME` nulová a veškerá paměť byla volná, tedy dostupná:

P1. Posunout `TIME` o 1.

P2. Uvolnit všechny bloky v systému, které mají být podle plánu uvolněny při aktuální hodnotě `TIME`.

P3. Vypočítat dvě veličiny S (náhodná velikost) a T (náhodná doba života), při vhodném rozdělení pravděpodobnosti a pomocí metod kapitoly 3.

P4. Rezervovat nový blok délky S, který bude uvolněn v okamžiku (`TIME` $+ T$). Návrat na P1. ▮

Při hodnotách `TIME`, které byly celistvým násobkem 200, byly dále vypsány podrobné statistiky o rychlosti algoritmů rezervace a uvolnění. Do každé dvojice testovaných algoritmů vstupovala stejná posloupnost hodnot S a T. Jakmile hodnota `TIME` převýšila 2000, obvykle již systém dosáhl víceméně stabilního stavu, který podle všech náznaků trval do nekonečna. V závislosti na celkovém množství dostupné paměti a na rozdělení veličin S a T v kroku P3 ale alokační algoritmus někdy selhal a nepodařilo se mu najít potřebné volné místo; tím byl simulační experiment ukončen.

Nechť C je celkový počet dostupných paměťových pozic a nechť $\bar{S}$ a $\bar{T}$ označují průměrné hodnoty S a T v kroku P3. Snadno nahlédneme, že očekávaný počet dostupných slov paměti v libovolném daném okamžiku je roven $\bar{S}\bar{T}$, pro dostatečně velké `TIME`. Jestliže bylo v experimentech $\bar{S}\bar{T}$ větší než zhruba $\frac{2}{3}C$, došlo obvykle k přetečení paměti, a to často ještě před požadavkem rezervace C slov. Při malé velikosti bloku ve srovnání s C se paměť zaplnila na více než 90 procent, ale pokud velikost bloků mohla překračovat $\frac{1}{3}C$ (přičemž současně nabývala i mnohem menších hodnot), program měl tendenci považovat paměť za „plnou“, i když z ní bylo obsazeno méně než $\frac{1}{2}C$ pozic. Empirické zkušenosti jasně ukazují, že pro efektivní činnost dynamické alokace paměti *není vhodné používat bloky větší než* $\frac{1}{10}C$.

Důvod tohoto chování snáze pochopíme na základě pravidla 50 procent: Dosáhne-li systém rovnovážného stavu, v němž je velikost průměrného volného bloku f menší než velikost průměrného obsazeného bloku r, můžeme očekávat příchod nesplnitelného požadavku, pokud v systému nemáme pro nouzové situace k dispozici jeden velký volný blok. V saturovaném systému, který nepřetéká, platí tedy $f \geq r$, přičemž je $C = fM+rN \geq rM+rN \approx (p/2+1)rN$. Celková velikost obsazené paměti je proto $rN \leq C/(p/2+1)$; při $p \approx 1$ nedokážeme využít více než jen zhruba 2/3 všech paměťových buněk.

Experimenty byly prováděny se třemi typy rozdělení velikosti S:

(*S1*) celé číslo vybrané rovnoměrně z intervalu od 100 do 2000;

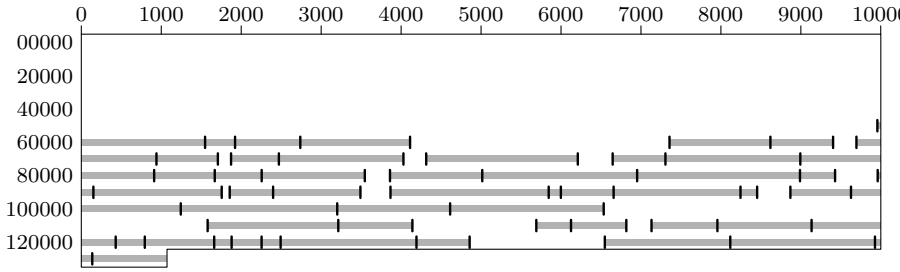
(*S2*) velikosti (1, 2, 4, 8, 16, 32), zvolené s pravděpodobností ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{32}$);

(*S3*) sizes (10, 12, 14, 16, 18, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 500, 1000, 2000, 3000, 4000) zvolené se stejnou pravděpodobností.

Časová veličina T obsahovala obvykle náhodné celé číslo s rovnoměrným rozdělením z intervalu 1 až t, pro pevné $t = 10$, 100 nebo 1000.

Při dalších experimentech bylo T v kroku P3 voleno s rovnoměrným rozdělením z intervalu od 1 do $\min\bigl(\lfloor\frac{5}{4}U\rfloor, 12500\bigr)$, kde U je počet časových jednotek zbývajících do dalšího plánovaného uvolnění nějakého momentálně rezervovaného bloku v systému. Toto rozdělení simuluje chování ve stylu „téměř-LIFO“: Pokud za T volíme vždy hodnoty $\leq U$, degeneruje systém alokace paměti do jednoduché zásobníkové operace, ke které nejsou potřeba žádné složité algoritmy. (Viz cvičení 1.) Při uvedeném rozdělení je zvolené T větší než U ve zhruba 20 procentech případů, takže téměř vždy, i když ne úplně, se dostáváme na obyčejnou zásobníkovou operaci. Algoritmy jako A, B a C se u tohoto rozdělení chovají podstatně lépe než obvykle; jen málokdy, pokud vůbec, obsahoval celý seznam `AVAIL` více než dvě položky, zatímco rezervovaných bloků bylo zhruba 14. Na druhé straně algoritmy systému s dvojicemi byly v tomto rozdělení pomalejší, protože u operace zásobníkového typu prováděly dělení a slučování bloků výrazně častěji. Odvození teoretických vlastností tohoto časového rozdělení se ukazuje být dosti obtížným (viz cvičení 32).

Na obrázku 42, uvedeném ze začátku této části textu, jsme viděli konfiguraci paměti při hodnotě `TIME` = 5000, rozdělením velikostí (*S1*) a rovnoměrným rozdělením času v množině $\{1, \ldots, 100\}$, přičemž jsme používali metodu first-fit jako ve výše popsaných Algoritmech A a B. Pravděpodobnost p, která vstupuje do „pravidla 50 procent“, byla u tohoto experimentu v podstatě rovna 1, takže se dalo očekávat, že dostupných bloků je proti rezervovaným zhruba polovina; obrázek 42 ukazuje ve skutečnosti 21 dostupných a 53 rezervovaných bloků. To ovšem pravidlo 50 procent nevyvrací: při `TIME` = 4600 bylo například 25 dostupných a 49 rezervovaných bloků. Konfigurace z obrázku 42 neukazuje nic jiného, než že i pravidlo 50 procent podléhá běžným statistickým odchylkám. Počet dostupných bloků spadá obvykle do intervalu od 20 do 30, zatímco počet rezervovaných bloků se pohybuje od 45 do 55.



Obr. 43. Mapa paměti při použití metody best-fit (porovnej s obrázkem 42, který ukazuje metodu first-fit, a obrázkem 44, na němž je systém dvojic, obojí při stejné sekvenci požadavků na paměť)

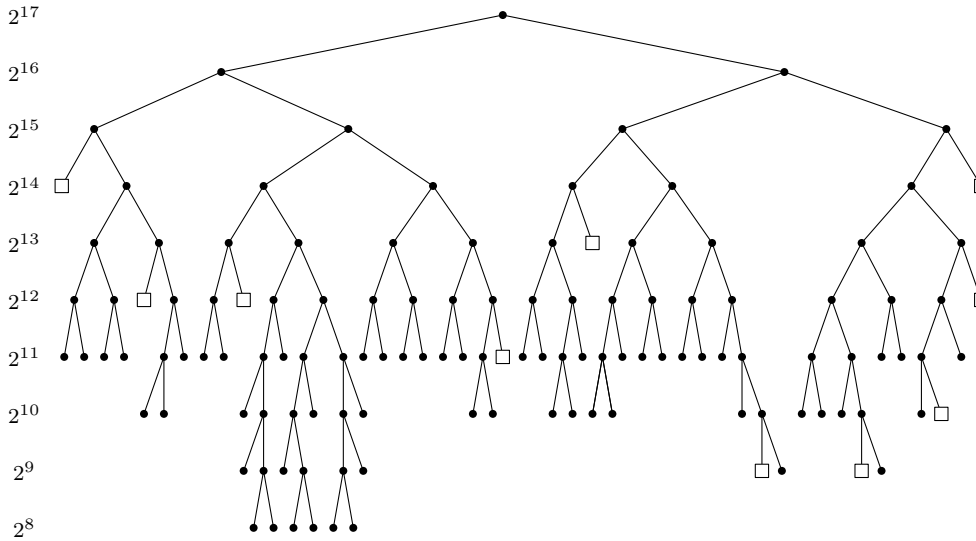
Obrázek 43 ukazuje konfiguraci paměti při *stejných datech jako na obrázku 42*, ale s metodou best-fit místo metody first-fit. Konstantu c v kroku A4′ jsme položili rovnu 16, abychom se tím zbavili malých bloků, v důsledku čehož se pravděpodobnost p zmenšila zhruba na 0,7 a dostupných oblastí bylo méně.

Po změně rozdělení času z původních 1 až 100 na 1 až 1 000 jsme dostali situaci přesně analogickou k obrázkům 42 a 43, přičemž všechny odpovídající veličiny se dostaly zhruba na 10násobek. V ekvivalentu k obrázku 42 bylo například 515 rezervovaných a 240 volných bloků a v ekvivalentu k obrázku 43 pak 176 volných bloků.

Ve všech experimentech s porovnáním obou metod se algoritmus first-fit ukázal být lepším než best-fit. Také po vyčerpání paměti zůstala metoda first-fit „naživu“ ve většině případů déle než best-fit, než došlo k přetečení paměti.

Na stejná data, z nichž jsme vytvořili obrázky 42 a 43, jsme uplatnili také systém dvojic, jehož výsledkem byl obrázek 44. Zde jsme všechny velikosti bloků z intervalu od 257 do 512 považovali za 512, všechny mezi 513 a 1024 jsme zvedli na 1024 atd. Průměrně to znamenalo rezervaci více než čtyř třetin požadované paměti (viz cvičení 21); systém dvojic pracuje samozřejmě lépe při rozdělení velikosti například podle (*S2*) než třeba podle (*S1*). Všimněte si, že na obrázku 44 jsou volné bloky o velikosti 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13} a 2^{14}.

Simulací systému dvojic jsme zjistili, že pracuje lépe, než bychom očekávali. Je zřejmé, že v systému dvojic budou někdy k dispozici dvě sousední oblasti o stejné velikosti, které nesloučíme do jedné (pokud samy netvoří dvojici); na obrázku 44 ale žádná taková situace nevzniká a v praxi je fakticky dosti vzácná. K přetečení paměti došlo až při jejím obsazení z 95 procent, což je překvapivě dobrá bilance alokací. Navíc dělení bloků v Algoritmu R a jejich slučování v Algoritmu S bylo nezbytné jen málokdy; strom zůstal většinou u své podoby z obrázku 44 a dostupné bloky se vyskytovaly na nejčastěji používaných úrovních. Některé matematické výsledky, jež dávají nahlédnout do tohoto chování, stejně jako i nejnižší úroveň stromu, popsali P. W. Purdom, Jr. a S. M. Stigler, *JACM* **17** (1970), 683–697.



Obr. 44. Mapa paměti při použití systému dvojic (stromová struktura indikuje dělení jistých velkých bloků do dvojic o poloviční velikost, čtverečky označují volné bloky)

Dalším překvapením bylo vynikající chování Algoritmu A po úpravě popsané ve cvičení 6; v průměru bylo potřeba provést jen 2,8 testů velikosti dostupného bloku (při rozdělení velikosti (*S1*) a při rovnoměrném rozdělení časů od 1 do 1000), přičemž ve více než polovině případů stačilo naprosté minimum, tedy jediná iterace. Tyto hodnoty platily navzdory přítomnosti zhruba 250 dostupných bloků. Stejný experiment s neupraveným Algoritmem A ukázal v průměru 125 iterací (což znamenalo prozkoumat průměrně polovinu seznamu `AVAIL`); 200 a více iterací bylo zaznamenáno ve zhruba 20 procentech případů.

Toto chování neupraveného Algoritmu A můžeme ve skutečnosti předvídat, protože je důsledkem pravidla 50 procent. V rovnovážném stavu bude část paměti s druhou polovinou rezervovaných bloků obsahovat také druhou polovinu volných bloků; do této části budeme zasahovat v polovině případů uvolnění bloku, a proto s ní musí pracovat i polovina alokací, aby byla zachována rovnováha. Stejné zdůvodnění platí i v případě, že polovinu nahradíme jiným zlomkem. (Tato pozorování popsal J. M. Robson.)

Mezi níže uvedenými cvičeními najdete programy pro počítač `MIX`, které v důsledku výše popsaných úvah realizují dvě nejdůležitější metody: (i) systém značkování hranic, v úpravách ze cvičení 12 a 16, a (ii) systém dvojic. Zde jsou přibližné výsledky:

	Doba pro rezervaci	Doba pro uvolnění
Systém značkování hranic:	$33 + 7A$	18, 29, 31 nebo 34
Systém dvojic :	$19 + 25R$	$27 + 26S$

Zde přitom $A \geq 1$ je počet iterací potřebných při hledání dostatečně velkého dostupného bloku, $R \geq 0$ je počet případů rozdělení bloku na dva (počáteční

rozdíl $j - k$ v Algoritmu R) a $S \geq 0$ je počet případů opětovného sloučení dvojic bloků v Algoritmu S. Ze simulačních experimentů vyplývá, že za uvedených předpokladů ohledně rozdělení velikosti (*S1*) a času voleného mezi 1 a 1000 můžeme v průměru brát $A = 2,8$; $R = S = 0,04$. (Po náhradě rozdělení času za výše popsaný systém „téměř-LIFO“ dostáváme průměrné hodnoty $A = 1,3$; $R = S = 0,9$.) Z toho vyplývá, že obě metody jsou dosti rychlé, přičemž na počítači `MIX` je o něco rychlejší systém dvojic. Nezapomeňte také, že pokud není velikost bloků omezena jen na mocniny dvou, vyžaduje systém dvojic zhruba o 44 procent více místa v paměti.

Odpovídající odhady času pro algoritmy uvolňování paměti (garbage collection) a zhuštění podle cvičení 33 představují okolo 104 jednotek času na vyhledání volného uzlu; přitom předpokládáme, že se uvolňování paměti spouští při zaplnění paměti přibližně z poloviny a že uzly mají průměrnou délku 5 slov, se 2 odkazy na uzel. Pro a proti k metodě uvolňování paměti jsme probírali v části 2.3.5. Není-li paměť příliš zatížena a jsou-li splněny příslušné omezující podmínky, je uvolňování paměti a její zhuštění velmi efektivní; například na počítači `MIX` je metoda uvolňování paměti rychlejší než obě ostatní, pokud dostupné položky nikdy nezabírají více než zhruba jednu třetinu celkového místa v paměti a pokud jsou uzly relativně malé.

Při splnění předpokladů, na nichž jsme stavěli uvolňování paměti, je nejlepší strategií rozdělit dostupný fond paměti na dvě poloviny a veškeré alokace provádět sekvenčně v jedné polovině. Nepotřebné bloky nebudeme okamžitě uvolňovat, nýbrž jednoduše počkáme až do zaplnění momentálně aktivní poloviny paměti; potom zkopírujeme veškerá aktivní data do druhé poloviny a současně odstraníme všechny díry mezi bloky, a to metodou jako ve cvičení 33. Při přechodu z jedné poloviny do druhé můžeme také upravit velikost každého z polovičních fondů.

Zmíněné simulační techniky jsme aplikovali také na některé jiné algoritmy pro alokaci paměti. Tyto jiné metody byly ale ve srovnání s algoritmy v této části tak špatné, že se o nich zmíníme jen velmi stručně:

a) Pro každou velikost jsme udržovali samostatný seznam `AVAIL`. Jediný volný blok jsme podle potřeby rozdělili na dva menší bloky, ale nikdy jsme je neslučovali zpět. Mapa paměti se postupně fragmentovala na menší a menší části, až dosáhla hrozivé podoby; takovéto příliš jednoduché schéma je téměř ekvivalentní samostatným alokacím v disjunktních oblastech paměti, přičemž každé velikosti bloku je vyhrazena jedna oblast.

b) Provedli jsme pokus o alokaci ve dvou úrovních: paměť byla rozdělena do 32 velkých sektorů. Rezervaci velkých bloků složených z 1, 2 nebo 3 sousedních sektorů (větší se vyskytovaly jen zřídka) jsme prováděli alokační metodou hrubé síly; každý z velkých bloků jsme pak rozdělili na menší podle paměťových požadavků, až v aktuálním velkém bloku nezbylo žádné místo, a potom jsme pro další alokace vyhradili další velký blok. Každý velký blok jsme vraceli do volného místa až po uvolnění *veškerého* jeho prostoru. V této metodě vždy došlo velmi rychle k vyčerpání volné paměti.

I když tato konkrétní metoda alokace ve dvou úrovních se při autorových simulačních experimentech ukázala jako nevyhovující, za jiných okolností (které se ovšem v praxi nevyskytují příliš často) může být alokace ve více úrovních přínosná. Pokud například nějaký velký program pracuje na několik etap, víme, že každý podprogram potřebuje jen uzly určitého typu. V některých programech může být také žádoucí používat pro různé třídy uzlů různé alokační strategie. Tuto myšlenku zónové alokace s možností uplatnění různých strategií v jednotlivých zónách a s možností uvolnění celé zóny najednou rozebírá Douglas T. Ross v článku *CACM* **10** (1967), 481–492.

Další empirické výsledky k dynamické alokaci paměti najdete v článcích B. Randell, *CACM* **12** (1969), 365–369, 372; P. W. Purdom, S. M. Stigler a T. O. Cheam, *BIT* **11** (1971), 187–195; B. H. Margolin, R. P. Parmelee a M. Schatzoff, *IBM Systems J.* **10** (1971), 283–304; J. A. Campbell, *Comp. J.* **14** (1971), 7–9; John E. Shore, *CACM* **18** (1975), 433–440; Norman R. Nielsen, *CACM* **20** (1977), 864–873.

***E. Distribuovaná alokace.** Pokud je rozdělení velikosti bloků předem známé a pokud je pravděpodobnost výběru každého z bloků za nejbližší uvolněný stejná, bez ohledu na okamžik jeho alokace, můžeme využít jinou techniku, která dosahuje podstatně lepšího využití paměti než dosud popisované univerzální techniky; vycházíme z doporučení, jež popsali E. G. Coffman, Jr., a F. T. Leighton [*J. Computer and System Sci.* **38** (1989), 2–35]. Jejich „metoda distribuované alokace“ (distributed-fit) spočívá v rozdělení paměti do zhruba $N + \sqrt{N} \lg N$ částí (slotů), kde N je požadovaný maximální počet bloků obsluhovaných v ustáleném stavu. Každý slot má pevnou velikost, i když různé sloty mohou mít různou velikost; podstatné je, že každý daný slot má pevné hranice a že je buďto prázdný, nebo obsahuje jeden alokovaný blok.

Prvních N slotů v Coffmanově a Leightonově schématu rozložíme podle předpokládaného rozdělení velikosti, zatímco posledních $\sqrt{N} \lg N$ slotů bude mít maximální velikost. Uvažujeme-li například rovnoměrné rozdělení velikosti bloků od 1 do 256 a očekáváme-li současnou obsluhu $N = 2^{14}$ bloků, rozdělíme paměť do $N/256 = 2^6$ slotů o velikosti 1, 2, ..., 256, za nimiž následuje „oblast přetečení“ s $\sqrt{N} \lg N = 2^7 \cdot 14 = 1792$ bloky o velikosti 256. Jakmile se systém přiblíží plné kapacitě, bude podle předpokladů obsluhovat N bloků o průměrné velikosti $\frac{257}{2}$, které zabírají $\frac{257}{2} N = 2^{21} + 2^{13} = 2\ 105\ 344$ pozic; toto množství paměťového prostoru jsme alokovali prvním N slotům. Kromě toho jsme si dali stranou dalších $1792 \cdot 256 = 458\ 752$ pozic, které obslouží náhodný rozptyl velikostí; tato dodatečná režie znamená $O(N^{-1/2} \log N)$ z celkového prostoru, nikoli konstantní násobek N jako u systému dvojic, takže pro $N \to \infty$ je zanedbatelná. V našem příkladu znamená ovšem zhruba 18 % z celkové alokace.

Sloty je třeba uspořádat podle velikosti, takže menší sloty jsou před většími. Za tohoto uspořádání můžeme bloky alokovat pomocí metody first-fit nebo best-fit. (V tomto případě jsou díky uspořádání slotů obě metody ekvivalentní.) Podle našich předpokladů je pak konečný výsledek ten, že při zpracování nového

alokačního požadavku zahájíme hledání v podstatě na náhodném místě mezi prvními N sloty a pokračujeme až do nalezení prázdného slotu.

Pokud je počáteční slot v každém hledání opravdu náhodný od 1 do N, nemusíme příliš často vstupovat do oblasti přetečení. Skutečně, potřebujeme-li vložit právě N položek počínaje od náhodného slotu, dojde k přetečení v průměru jen $O(\sqrt{N}\,)$-krát. Důvod je takový, že je tento algoritmus srovnatelný s hašováním pomocí algoritmu lineárního prohledávání (Algoritmus 6.4L), který vykazuje stejné chování, pouze hledání prázdné buňky pokračuje od N k 1 a nepřechází do oblasti přetečení. Z analýzy Algoritmu 6.4L ve Větě 6.4K vyplývá, že po vložení N položek je průměrný posuv každé z nich od hašované adresy roven $\frac{1}{2}(Q(N)-1) \sim \sqrt{\pi N/8}$; z kruhové symetrie snadno nahlédneme, že je tento průměr stejný jako průměrný počet případů, kdy hledání pokračuje ze slotu k do $k+1$, pro každé k. Přetečení odpovídá v metodě distribuované alokace hledání s přechodem ze slotu N do slotu 1, ale naše situace je ještě lepší, protože odstraněním přechodu přes počátek zabráníme jistému zahlcení. V průměru dojde tedy k méně než $\sqrt{\pi N/8}$ případům přetečení. Tato analýza nebere v úvahu odstraňování, při němž jsou předpoklady Algoritmu 6.4L zachovány jen tehdy, pokud při odstranění bloku vloženého mezi počáteční pozicí a alokovanou pozicí přesuneme bloky směrem zpět (viz opět Algoritmus 6.4R); tímto přesunem ale opět zvyšujeme pravděpodobnost přetečení. Naše analýza nebere v úvahu také důsledky přítomnosti více než N bloků současně; k tomu může dojít, pokud předpokládáme, že doba příchodu mezi bloky je zhruba jedna N-tina doby pobytu. Pro více než N bloků musíme rozšířit analýzu Algoritmu 6.4L, ale Coffman s Leightonem dokázali, že oblast přetečení nebude téměř nikdy potřebovat více než $\sqrt{N}\lg N$ slotů; pravděpodobnost dosažení konce je menší než $O(N^{-M})$, pro všechna M.

V našem příkladu není rozdělení počátečního slotu při hledání v rámci alokace rovnoměrné mezi sloty 1, 2, ..., N, nýbrž je rovnoměrné mezi sloty 1, 65, 129, ..., $N-63$, protože slotů o jedné stejné velikosti je vždy $N/256 = 64$. Díky této odchylce od náhodného modelu uvažovaného v předchozím odstavci je ale pravděpodobnost přetečení ještě menší, než jsme uvažovali. Při porušení předpokladů ohledně rozdělení velikosti bloků a doby obsazení nemůžeme samozřejmě říci nic.

F. Přetečení. Co udělat, když v paměti není dost místa? Dejme tomu, že nám přišel požadavek třeba na spojitý blok o n slovech, přičemž všechny dostupné bloky jsou malé. Při prvním výskytu této situace je obvykle k dispozici více než n volných pozic, které ale nejsou spojité; po zhuštění paměti, tedy po přesunu některých obsazených pozic a sloučení volných, můžeme pokračovat v práci. Zhuštění je ale pomalé a musíme při něm disciplinovaně pracovat s ukazateli; navíc, v drtivé většině případů, kdy metoda first-fit narazí na vyčerpání paměťového prostoru, dojde záhy k vyčerpání veškerého volného místa, bez ohledu na prováděné zhuštění. Psát program pro zhuštění paměti obvykle nemá příliš velký smysl, jen za jistých speciálních okolností v souvislosti s uvolňováním paměti, jak si řekneme ve cvičení 33. Očekáváme-li přetečení, můžeme využít některé

metody pro odstraňování položek z paměti a pro jejich ukládání na externí paměťové zařízení a poté je podle potřeby za pomoci zvláštních opatření opět přesouvat zpět. Pro veškeré programy, které pracují s dynamickou pamětí, to ale znamená poměrně přísná omezení ohledně povolených odkazů do jiných bloků a pro efektivní činnost za těchto podmínek je obvykle nutný speciální počítačový hardware (který například při nepřítomnosti dat vyvolá přerušení nebo provede automatické stránkování).

Dále musíme vytvořit vhodnou rozhodovací proceduru, v níž určíme, které bloky jsou nejvhodnějšími kandidáty pro odstranění. Jednou z možností je udržovat obousměrně propojený seznam rezervovaných bloků a při každém přístupu v něm daný blok posunout směrem k začátku seznamu; bloky jsou tak efektivně seřazeny podle okamžiku posledního přístupu a blok na konci seznamu budeme odstraňovat jako první. Podobných výsledků dosáhneme i jednodušším způsobem, kdy rezervované bloky umístíme do kruhového seznamu a do každého bloku doplníme bit „nedávno použitý“; do něj při každém přístupu k bloku zapíšeme 1. Jakmile potřebujeme nějaký blok odstranit, začneme posunovat ukazatel po kruhovém seznamu a vynulujeme všechny bity „nedávného použití“, dokud nenajdeme blok, který nebyl od posledního vstupu ukazatele do této části kruhu použitý.

J. M. Robson ukázal [*JACM* **18** (1971), 416–423], že u strategií dynamické alokace paměti, které nikdy neprovádějí relokaci rezervovaných bloků, nelze zaručit efektivní využívání paměti; vždy budou existovat patologické případy, kdy metoda selhává. I když třeba omezíme bloky na velikost 1 a 2, může dojít k přetečení paměti při jejím zaplnění z pouhých $\frac{2}{3}$, bez ohledu na konkrétní alokační algoritmus! Na Robsonovy zajímavé výsledky se podíváme ve cvičeních 36–40 a ve cvičeních 42–43, kde ukazuje, že metoda best-fit má oproti metodě first-fit velmi špatný nejhorší případ.

G. Doporučená literatura. Vyčerpávající a kriticky pojatý přehled technik dynamické alokace paměti, který vychází z mnohem rozsáhlejších zkušeností, než jaké měl autor v době vzniku materiálu k dispozici, sestavili Paul R. Wilson, Mark S. Johnstone, Michael Neely a David Boles, *Lecture Notes in Computer Science* **986** (1995), 1–116.

CVIČENÍ

1. [*20*] Jaké zjednodušení můžeme provést v algoritmech rezervace a uvolňování z této části textu, přicházejí-li požadavky na paměť vždy v pořadí „LIFO“, tedy pokud je rezervovaný blok uvolněn až po uvolnění všech bloků, které byly rezervovány po něm?

2. [*VM23*] (E. Wolman.) Dejme tomu, že chceme zvolit pevnou velikost uzlu pro položky o proměnné délce, a dejme tomu, že při délce uzlu k a délce položky l znamená uložení položky $\lceil l/(k-b) \rceil$ uzlů. (Číslo b je zde konstantou a znamená, že v každém uzlu obsahuje b slov řídící informace, jako je odkaz na další uzel.) Pokud je průměrná délka l položky rovna L, při jaké hodnotě k bude průměrný objem obsazené paměti minimální? (Předpokládejte, že průměrná hodnota $\big(l/(k-b)\big) \bmod 1$ je rovna $1/2$, pro libovolné pevné k a pro pohyblivé l.)

3. [*40*] Simulací na počítači porovnejte při alokaci paměti metody best-fit, first-fit, a *worst-fit*; poslední popsaná metoda volí vždy „nejhorší vhodný“ blok, tedy největší dostupný blok. Je mezi těmito metodami nějaký významný rozdíl v obsazení paměti?

4. [*22*] Napište program pro `MIX`, který implementuje Algoritmus A; dbejte přitom o rychlé zpracování vnitřního cyklu. Předpokládejte, že pole `SIZE` je $(4:5)$, pole `LINK` je $(0:2)$ a $\Lambda < 0$.

▸ **5.** [*18*] Dejme tomu, že v Algoritmu A víme, že je `N` vždy větší nebo rovno 100. Bude rozumné provést v upraveném kroku A4′ přiřazení $c = 100$?

▸ **6.** [*23*] (*Next fit.*) Po opakovaném volání Algoritmu A budou mít bloky o malé velikosti `SIZE` silnou tendenci zůstávat na začátku seznamu `AVAIL`, takže při hledání bloku délky `N` a vyšší bude často nutné prohledávat dosti velkou část seznamu. Všimněte si například, jak se na obrázku 42 velikosti bloků směrem od začátku paměti ke konci zvětšují, a to jak rezervovaných, tak i volných. (Seznam `AVAIL` byl při přípravě obrázku 42 seřazen podle pozice, jak vyžaduje Algoritmus B.) Umíte navrhnout vhodnou úpravu Algoritmu A takovou, že (a) krátké bloky se nebudou kumulovat v žádné oblasti a že (b) seznam `AVAIL` budeme moci opět uchovávat v rostoucích paměťových pozicích, jak požaduje Algoritmus B?

7. [*10*] V příkladu (1) jsme si ukázali, že metoda first-fit může být někdy výrazně lepší než best-fit. Uveďte podobný příklad, kdy je naopak best-fit lepší než first-fit.

8. [*21*] Ukažte, jak pomocí jednoduché úpravy změnit Algoritmus A z metody first-fit na best-fit.

▸ **9.** [*26*] Jakými způsoby je možné navrhnout rezervační algoritmus s metodou best-fit, pokud v něm nemáme prohledávat celý seznam `AVAIL`? (Přemýšlejte, jak potřebné hledání co nejvíce zkrátit.)

10. [*22*] Ukažte, jak upravit Algoritmus B s tím, že uvolní spojitý blok `N` buněk počínaje pozicí `P0`, aniž bychom předpokládali, že je každá z těchto `N` buněk momentálně obsazená; předpokládejte, že uvolňovaná oblast se ve skutečnosti může překrývat s několika již volnými bloky.

11. [*M25*] Ukažte, že vylepšením Algoritmu A z odpovědi na cvičení 6 můžeme dosáhnout také mírného zlepšení Algoritmu B, jenž zkrátí průměrnou délku hledání z poloviny délky seznamu `AVAIL` na jednu třetinu. (Předpokládejte, že uvolněné bloky se do seřazeného seznamu `AVAIL` vkládají na náhodná místa.)

▸ **12.** [*20*] Upravte Algoritmus A s tím, že bude používat dohody o značkování hranic (7)–(9), že v něm bude upravený krok A4′ z textu a že v něm budou začleněny úpravy ze cvičení 6.

13. [*21*] Napište program pro `MIX`, který bude implementací algoritmu ze cvičení 12.

14. [*21*] Jak by se změnil Algoritmus C a algoritmus ze cvičení 12, pokud by (a) pole `SIZE` nebylo zapsáno do posledního slova volného bloku, respektive (b) pokud by pole `SIZE` nebylo zapsáno do prvního slova rezervovaného bloku?

▸ **15.** [*24*] Ukažte, jak urychlit Algoritmus C za cenu mírného prodloužení programu, přičemž v každém ze čtyř případů vymezených znaménky `TAG(P0 − 1)` a `TAG(P0 + + SIZE(P0))` (plus nebo minus) nebudeme měnit více odkazů, než kolik je nezbytně nutné.

16. [*24*] Napište program pro `MIX`, který bude implementací Algoritmu C s úpravami podle cvičení 15.

17. [*10*] Jaký bude obsah veličin `LOC(AVAIL)` a `LOC(AVAIL)`$+1$ ve schématu (9), pokud není k dispozici žádný volný blok?

▸ **18.** [*20*] Obrázky 42 a 43 jsme dostali ze stejných dat a v podstatě ze dvou stejných algoritmů (Algoritmu A a B), až na to, že na obrázku 43 jsme Algoritmus A upravili a namísto metody first-fit jsme v něm uplatnili metodu best-fit. Proč takto na obrázku 42 vznikla velká dostupná oblast ve *vyšších* paměťových pozicích, zatímco na obrázku 43 je velká volná oblast v *nižších* pozicích?

▸ **19.** [*24*] Uvažujte bloky paměti ve formátu (7), ale bez polí `TAG` a `SIZE`, které původně byly v posledním slově bloku. Dále předpokládejte, že pro opětovné uvolnění rezervovaného bloku používáme následující jednoduchý algoritmus: `Q` $\leftarrow$ `AVAIL`, `LINK(P0)` $\leftarrow$ `Q`, `LINK(P0` $+1$`)` $\leftarrow$ `LOC(AVAIL)`, `LINK(Q` $+1$`)` $\leftarrow$ `P0`, `AVAIL` $\leftarrow$ `P0`, `TAG(P0)` $\leftarrow$ „$-$“. (Tento algoritmus se nijak nestará o slučování sousedních oblastí.)

Navrhněte rezervační algoritmus podobný Algoritmu A, který během prohledávání seznamu `AVAIL` provede i potřebné sloučení sousedních volných bloků a současně zabrání zbytečné fragmentaci paměti, popsané ve schématech (2), (3) a (4).

20. [*00*] Proč je v systému dvojic vhodné mít seznamy `AVAIL[`k`]` obousměrně propojené, a ne je držet jako prosté lineární seznamy?

21. [*VM25*] Zkoumejte poměr a_n/b_n, kde a_n je součet prvních n členů řady $1+2+{}$ $+4+4+8+8+8+8+16+16+\cdots$, a b_n je součet prvních n členů řady $1+2+3+{}$ $+4+5+6+7+8+9+10+\cdots$, pro n jdoucí do nekonečna.

▸ **22.** [*21*] Text opakovaně upozorňuje, že v systému dvojic je možné používat jen bloky o velikosti 2^k, přičemž ve cvičení 21 jsme si ukázali, že tím mohou vzniknout výrazně vyšší paměťové nároky. Potřebujeme-li ale v systému dvojic blok o 11 slovech, proč bychom nemohli vyhledat blok o 16 slovech a rozdělit jej na část s 11 slovy a dva volné bloky o velikosti 4 a 1?

23. [*05*] Jaká je dvojková adresa druhého bloku z dvojice k bloku o velikosti 4 a adrese 011011110000? Čemu se adresa bude rovnat, pokud má mít blok velikost 16?

24. [*20*] Podle algoritmu v textu nemá největší blok (o velikosti 2^m) druhý blok v dvojici, protože reprezentuje veškerou paměť. Bylo by správné definovat $\text{buddy}_m(0) = {}$ $= 0$ (aby tak blok tvořil dvojici se sebou samým) a potom v kroku S1 netestovat podmínku $k = m$?

▸ **25.** [*22*] Kriticky posuďte následující myšlenku: „Dynamická alokace paměti v systému dvojic nemůže při praktické situaci nikdy rezervovat blok o velikosti 2^m (protože tím by se zaplnila celá paměť) a obecně platí maximální velikost 2^n, nad kterou se již žádný blok nebude rezervovat. Proto je plýtváním času začínat s tak velkými dostupnými bloky a slučovat dvojice v Algoritmu S, pokud má sloučený blok velikost větší než 2^n.“

▸ **26.** [*21*] Vysvětlete, jak systém dvojic využít pro dynamickou alokaci paměti v pozicích 0 až `M` -1 i v případě, že `M` není ve tvaru 2^m, jak text vyžaduje.

27. [*24*] Napište program pro `MIX`, který implementuje Algoritmus R, a stanovte dobu zpracování.

28. [*25*] Předpokládejme, že `MIX` je dvojkový počítač a že má definovaný nový operační kód `XOR` (v souladu s notací části 1.3.1): „C $= 5$, F $= 5$. Pro každou bitovou pozici v M, která je rovna 1, se odpovídající bitová pozice v registru A změní na doplněk (tedy z 0 na 1 nebo naopak); znaménko rA zůstává nezměněné. Doba provádění je $2u$.“

Napište program pro `MIX`, který je implementací Algoritmu S, a stanovte dobu provádění.

29. [*20*] Obešli bychom se v systému dvojic bez značkového bitu v každém rezervovaném bloku?

30. [*M48*] Analyzujte průměrné chování Algoritmů R a S, pokud jsou dána rozumná rozdělení sekvence požadavků na paměť.

31. [*M40*] Je možné navrhnout systém alokace paměti analogický k systému dvojic, který namísto mocnin dvou bude používat Fibonacciho posloupnost? (Můžeme tedy začít s F_m dostupnými slovy a rozdělit dostupný blok o F_k slovech do dvou bloků v dvojici s délkami F_{k-1} a F_{k-2}.)

32. [*VM46*] Stanovte $\lim_{n\to\infty} \alpha_n$, pokud existuje, kde α_n je střední hodnota t_n v náhodné posloupnosti definované takto: Jsou-li dány hodnoty t_k pro $0 \le k < n$, nechť t_n je vybráno rovnoměrně z množiny $\{1, 2, \ldots, g_n\}$, kde

$$g_n = \lfloor \tfrac{5}{4} \min\bigl(10000,\, f(t_{n-1} - 1),\, f(t_{n-2} - 2),\, \ldots,\, f(t_0 - n)\bigr) \rfloor,$$

a $f(x) = x$ if $x > 0$, $f(x) = \infty$ if $x \le 0$. [*Poznámka:* Jisté omezené empirické testy ukazují, že α_n bude přibližně rovno 14, tento odhad ale zřejmě není příliš přesný.]

▸ **33.** [*28*] (*Uvolňování paměti a zhuštění.*) Předpokládejme, že paměťové pozice 1, 2, ..., `AVAIL` − 1 slouží jako paměťový fond pro uzly různé velikosti, které mají následující tvar: První slovo uzlu `NODE(P)` obsahuje pole

`SIZE(P)` = počet slov v `NODE(P)`,
`T(P)` = počet polí odkazů v `NODE(P)`; `T(P)` < `SIZE(P)`,
`LINK(P)` = speciální pole odkazu, které slouží jen při uvolňování paměti.

Bezprostředně za uzlem `NODE(P)` následuje v paměti uzel `NODE(P + SIZE(P))`. Dále předpokládejme, že z polí v `NODE(P)` slouží jako odkazy na jiné uzly jen pole `LINK(P + 1)`, `LINK(P + 2)`, ..., `LINK(P + T(P))` a že každé z těchto polí odkazů obsahuje buďto Λ, nebo adresu prvního slova jiného uzlu. Nakonec předpokládejme, že je v programu ještě jedna proměnná odkazu s názvem `USE`, která ukazuje na jeden z uzlů.

Navrhněte algoritmus, který (i) stanoví všechny uzly přístupné přímo nebo nepřímo z proměnné `USE`, (ii) přesune tyto uzly do paměťových pozic od 1 do `K` − 1, pro vhodné `K`, a změní všechny odkazy tak, že strukturální vztahy zůstanou zachovány, a (iii) přiřadí `AVAIL` ← `K`.

Uvažujme například takovýto obsah paměti, kde `INFO(L)` označuje obsah pozice `L`, kromě `LINK(L)`:

1: `SIZE` = 2, `T` = 1	6: `SIZE` = 2, `T` = 0	`AVAIL` = 11,
2: `LINK` = 6, `INFO` = A	7: `CONTENTS` = D	`USE` = 3.
3: `SIZE` = 3, `T` = 1	8: `SIZE` = 3, `T` = 2	
4: `LINK` = 8, `INFO` = B	9: `LINK` = 8, `INFO` = E	
5: `CONTENTS` = C	10: `LINK` = 3, `INFO` = F	

Algoritmus musí tento obsah transformovat na

1: `SIZE` = 3, `T` = 1	4: `SIZE` = 3, `T` = 2	`AVAIL` = 7,
2: `LINK` = 4, `INFO` = B	5: `LINK` = 4, `INFO` = E	`USE` = 1.
3: `CONTENTS` = C	6: `LINK` = 1, `INFO` = F	

34. [*29*] Napište program pro `MIX`, který implementuje algoritmus ze cvičení 33, a stanovte dobu zpracování.

35. [*22*] Porovnejte metody dynamické alokace paměti z této části textu s technikami pro sekvenční seznamy o proměnné velikosti uvedenými na konci části 2.2.2.

▸ **36.** [*20*] Jistá restaurace v kalifornském Hollywoodu má 23 míst k sezení v řadě. Hosté přicházejí samostatně po jednom nebo v páru a usměvavá hosteska je usadí. Dokažte, že hosty je možné vždy usadit okamžitě, bez rozdělení jakéhokoli páru, pokud žádný samotný host (který nepřišel ve dvojici) nebude usazen na místo číslo 2, 5, 8, ..., 20 a pokud v restauraci nikdy není víc než 16 hostů současně. (Pár hostů opouští restauraci společně.)

▸ **37.** [*26*] Pokračujeme-li ve cvičení 36, dokažte, že při 22 místech již hosteska nemůže hosty vždy obsloužit: ať zvolí jakoukoli strategii, vždy bude možné dospět do situace, kdy do restaurace vstupuje dvojice přátel a hostů je jen 14, ale nejsou volná žádná dvě sousední místa.

38. [*M21*] (J. M. Robson.) Problém restaurace popsaný ve cvičeních 36 a 37 můžeme zobecnit a popsat tak nejhorší možný případ jakéhokoli algoritmu dynamické alokace paměti, který nikdy nerelokuje rezervované bloky. Nechť $N(n, m)$ je nejmenší množství paměti takové, že libovolnou posloupnost požadavků na alokaci a uvolnění je možné obsloužit bez přetečení, a to za předpokladu, že všechny velikosti bloků jsou $\le m$ a že celkové množství požadovaného prostoru nikdy nepřekročí n. Ve cvičeních 36 a 37 jsme dokázali, že $N(16, 2) = 23$; stanovte přesnou hodnotu $N(n, 2)$ pro všechna n.

39. [*VM23*] (J. M. Robson.) V notaci ze cvičení 38 ukažte, že $N(n_1 + n_2, m) \le \le N(n_1, m) + N(n_2, m) + N(2m - 2, m)$; a tedy že pro pevné m existuje limita $\lim_{n\to\infty} N(n, m)/n = N(m)$.

40. [*VM50*] Pokračujme ve cvičení 39; stanovte $N(3)$, $N(4)$ a $\lim_{m\to\infty} N(m)/\lg m$, pokud existuje.

41. [*M27*] Úkolem tohoto cvičení je uvažovat nejhorší možný případ obsazení paměti v systému s dvojicemi. Zvláště špatný případ nastává například tehdy, pokud začneme s prázdnou pamětí a postupujeme takto: Nejprve rezervujeme $n = 2^{r+1}$ bloků délky 1, které umístíme do pozic od 0 do $n - 1$; potom pro $k = 1, 2, \ldots, r$, uvolníme všechny bloky, jejichž počáteční pozice nejsou dělitelné 2^k, a rezervujeme $2^{-k-1}n$ bloků délky 2^k, jež umístíme do pozic $\frac{1}{2}(1 + k)n$ až $\frac{1}{2}(2 + k)n - 1$. Při tomto postupu obsadíme $1 + \frac{1}{2}r$-krát více paměti, než kolik skutečně potřebujeme.

Dokažte, že nejhorší možný případ nemůže být podstatně horší než tento: Jestliže ve všech požadavcích žádáme bloky o velikosti 1, 2, ..., 2^r a jestliže celkový objem prostoru požadovaný v libovolném okamžiku nikdy nepřekročí n, kde n je násobkem 2^r, v systému dvojic nikdy nedojde k přetečení paměti o velikosti $(r + 1)n$.

42. [*M40*] (J. M. Robson, 1975.) Nechť $N_{\rm BF}(n, m)$ je množství paměti, které potřebujeme, aby u metody alokace best-fit ze cvičení 38 zaručeně nedocházelo k přetečení. Najděte útočnou strategii, se kterou ukážete, že je $N_{\rm BF}(n, m) \ge mn - O(n + m^2)$.

43. [*VM35*] Pokračujme ve cvičení 42 a položme $N_{\rm FF}(n, m)$ za množství paměti potřebné u metody first-fit. Najděte vhodnou defenzivní strategii, pomocí které ukážete, že je $N_{\rm FF}(n, m) \le H_m n/\ln 2$. (Nejhorší případ metody first-fit není tedy příliš vzdálený od nejlepšího možného nejhoršího případu.)

44. [*M21*] Předpokládejme, že distribuční funkce $F(x) =$ (pravděpodobnost, že má blok velikost $\le x$) je spojitá. Například pro $a \le x \le b$ je $F(x)$ rovno $(x - a)/(b - \\ - a)$, pokud velikosti nabývají rovnoměrného rozdělení mezi a a b. Uveďte vzorec, který vyjadřuje velikosti prvních N slotů, které musí být vytvořeny u metody distribuované alokace.