VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

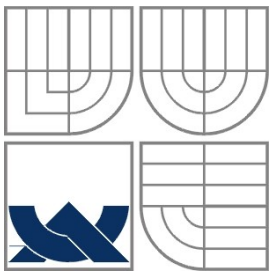
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMICKÉ PŘIDĚLOVÁNÍ PAMĚTI V TINYOS

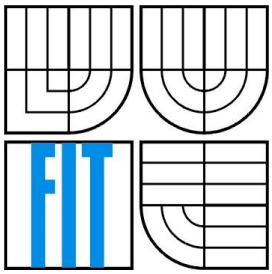
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JÁN KRIŽANSKÝ
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMICKÉ PŘIDĚLOVÁNÍ PAMĚTI V TINYOS

DYNAMIC MEMORY ALLOCATION IN TINYOS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — JÁN KRIŽANSKÝ
AUTHOR

VEDOUCÍ PRÁCE — Ing. HORÁČEK JAN
SUPERVISOR

BRNO 2010

Abstrakt

Táto práce porovnáva po stránce teoretické i praktické různé způsoby dynamického přidělování paměti v operačním systému TinyOS určeném pro jednoduchá programovatelná zařízení. U jednotlivých způsobů byly sledovány různé aspekty. Mezi hlavní patří efektivita využití zdrojů, rychlost alokace a použitelnost z programátorského hlediska. Práce zároveň rozšiřuje možnosti alokace paměti na externí flash paměť. Ve stručnoti také vysvětluje princIpy dynamické alokace paměti a její aplikace v prostředí jednoduchých programovatelných zařízení s omezenými zdroji.

Klíčová slova

dynamické přidělování paměti, TinyOS, nesC, knihovna, malloc, realloc, free, Micaz, jednoduchá programovatelná zařízení, WSN, flash, modul

Abstract

This thesis aims for a theoretical and practical comparison of two different approaches to dynamic memory management in TinyOS operating system designated for simple programmable devices. For each of the approaches we were observing different aspects, among others the effectivity of usage of resources, allocation speed and usability from the programmers point of view. Thesis also extends the capabilities of dynamic allocation onto external flash memory. It shortly explains the principles behind dynamic memory allocation and it's application in the area of simple programmable devices with limited resources.

Keywords

dynamic memory allocation, TinyOS, nesC, library, malloc, realloc, free, Micaz, simple programmable devices, WSN, flash, module

Citace

Ján Križanský: Dynamické přidělování paměti v TinyOS. Brno, 2010, bakalářská práce, FIT VUT v Brne.

Dynamické přidělování paměti v TinyOS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Horáčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ján Križanský
19.5.2010

Poděkování

Poďakovanie patrí vedúcemu práce Ing. Janovi Horáčkovi za jeho technické rady a rady pri vypracovaní písomnej správy.

Obsah

Úvod

Dynamická správa pamäte je jedným zo základných prvkov moderných operačných systémov. Dá sa povedať, že v čase uvedenia tejto techniky sa jednalo o nevyhnutný krok vo vývoji informačných technológií, ktorý umožnil operačným systémom lepšie využívať cenné pamäťové zdroje výpočtových zariadení a tým umožnil beh omnoho náročnejších programov bez nutného zvyšovania hardwarových (HW) požiadavkov. Ak sa dnes pozrieme na väčšinu moderných operačných systémov (OS) zistíme, že s výnimkou vysoko špecializovaných alebo experimentálnych OS, už dnes všetky poskytujú možnosť dynamickej alokácie pamäte. Dôležitosť tejto vlastnosti potvrdzuje aj fakt, že sa táto technika stala neoddeliteľnou súčasťou mnohých vyšších programovacích jazykov (C, C++, Java, PHP) a to až do takej miery, že bola jej obsluha úplne zautomatizovaná a jej správa už není súčasťou samotnej programátorskej činnosti (Java, PHP) [1].

A tak niet divu, že táto dlhodobo uznávaná praktika (v dnešnej dobe ju už môžme dokonca nazvať dogma) sa rozširuje do menej známych oblastí informačných technológií. Jednou z takých oblastí sú senzorové siete. Ide o množiny malých, bezdrôtovo komunikujúcich prenosných zariadení, ktoré práve kôli svojej veľkosti a zachovaniu relatívne nízkej ceny sa nevyznačujú veľkými HW prostriedkami. No aj napriek tomu sú prípady a problémy, ktoré si vyžadujú dynamickú správu pamäte aj na takýchto zariadeniach. Bližšie si o senzorových sieťach povieme v kapitole 1, kde sa oboznámime s ich možnosťami aj s ich obmedzeniami.

Naším cieľom teda bude nájsť cestu, ako využiť techniku dynamickej alokácie na týchto zariadeniach a prípadne porovnať niekoľko rôznych prístupov s prihliadnutím na rôzne požiadavky programátorov aplikácií pre tieto zariadenia. Zároveň by mal tento dokument slúžiť ako technická dokumentácia pre programátorov, ktorí by sa rozhodli tieto riešenia využiť vo svojich aplikáciách a na príkladoch ukázať ich vzorové použitie.

V prvých kapitolách si teda predstavíme prostredie v ktorom sa budeme pohybovať. Hneď v prvej kapitole si v krátkosti uvedieme princípy fungovania senzorových sietí a zariadení, ktoré tieto siete vytvárajú. Jedná sa totiž o akýsi hybrid univerzálnych a zároveň dosť špecializovaných zariadení, ktoré sa vyznačujú malými rozmermi, dlhou životnosťou a relatívne nízkou cenou. Tieto vlastnosti ale zároveň znamenajú mnoho technických obmedzení a špecifických riešení, s ktorými sa v tejto časti oboznámime.

Jedným z takýchto špecifík je účelovo zameraný operačný systém TinyOS vytvorený za pomoci rovnako účelovo zameraného programovacieho jazyka nesC. Obe tieto technológie (alebo ich nazvime prostredia) sú pevne späté so samotnou platformou senzorových zariadení. Obzvlášť teda jazyk nesC, ktorý bol vytvorený s cieľom odtieniť HW periférie týchto zariadení a dokonale využiť ich modulárne vlastnosti. Ako si neskôr ukážeme, je toho docielené za pomoci možno trochu netradičného princípu linkovania modulov, kedy samotný modul (ani prográtor, ktorý modul vyvinul)

nepozná konkrétne komponenty, s ktorými bude v rámci aplikácie komunikovať. Tým je umožnená väčšia flexibilita využitia modulov a možnosť vytvárať aplikácie už samotným spájaním komponent, čo je u iných programovacích jazykov neobvyklé.

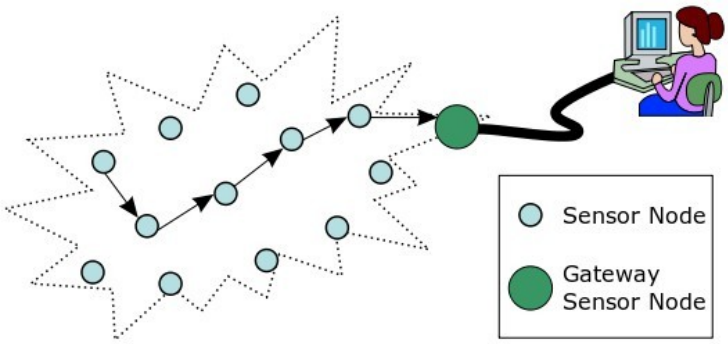
Z týchto princípov teda čerpá i operačný systém TinyOS, ktorý, ako to už vyplýva zo samotného názvu, má svoju výhodu hlavne vo svojej veľkosti. Nejedná sa ale o samostatne bežiaci systém, ktorý by obsluhoval beh procesov. Naopak aplikácia a operačný sytém tvoria jeden program, ktorý je jedinou obslužnou rutinou na zariadení a prevažne vykonáva opakovane istú konkrétu činnosť (zber a vyhodnocovanie dát) až do vypnutia zariadenia. Zariadenie sa tak prostredníctvom aplikácie stáva špecializované na vykonávanie jednej činnnosti.

V kapitole 4 sa už budeme venovať nášmu problému a to vytvoreniu modulu pre TinyOS, ktorý nám doplní chýbajúcu časť OS, umožňujúcu jednoduchú prácu s celým pamäťovým priestorom – dynamickú alokáciu pamäte. Popíšeme si moduly, ktoré som pre jednoduchosť nazval TMalloc a TMallocFlash a následne ich za pomoci niekoľkých testovacích aplikácií porovnáme aby me vedeli určiť silné a slabé stránky oboch riešení.

V závere zhodnotíme dosiahnuté výsledky a navrhneme možnosti využitia modulov. Koniec koncov jednou z motivácií pre vytvorenie týchto modulov je ich reálne využitie v aplikáciách vyvíjaných na Ústave inteligentných systémov na tejto fakulte.

1 Bezdrôtové senzorové siete

Pre začiatok si ale popíšme, čo to vlastne bezdôtové senzorové siete sú. Bezdrôtové senzorové siete (angl. Wireless Sensor Networks – WSN) sú skupiny zariadení, schopné vzájomne komunikovať prostredníctvom rádiového spojenia. Vzniká tak sieť malých zariadení, v ktorej prostredníctvom krokového prenosu informácie zo zariadenia na zariadenie je možné prenášať informáciu na veľké vzdialenosti a plochy [obr. 1.1] . Môžme si tento prenos informácie vysvetliť na situácii, keď sa na spoločenkej akcii stratí dieťa. Táto udalosť vyvolá šírenie správy o stratenom dieťati všetkými smermi až do doby, kým sa dieťa nenájde. Ako vidíme, informácia sa prenáša na krátku vzdialenosť medzi mnohými členmi skupiny (uzlami) no dokážu pokryť pomerne veľkú plochu zdieľanou informáciou.



Obr. 1.1 - Schéma komunikácie uzlov vrámci senzorovej siete
(Sensor Node - senzorový uzol, Gateway Sensor Node - riadiaci uzol)
Komunikácia prebieha od konkrétneho uzlu krokovo cez okolité uzly až k ústrednému uzlu, ktorý komunikuje so systémom na spracovanie dát

V senzorových sieťach sú uzly úplne sebestačné. Dokážu tak fungovať bez zásahu človeka a to pomerne dlhú dobu. Nová IDTechEx správa "Wireless Sensor Networks 2010-2020" [2] špecifikuje, že zariadenie by malo byť schopné bežať nezávisle po dobu 20 rokov. Aj keď tento cieľ zatiaľ nie je štandardne dosiahnuteľný, je jedným z hlavných zameraní vo vývoji senzorových zariadení dosiahnutie tohto cieľa, napríklad využitím technológií, ktoré by boli schopné získavať energiu pre zariadenia v priebehu jeho života.

Toho by sa malo docieliť vývojom a zdokonalením MEMS súčiastok [sl1], ktoré by boli na základe vybraných fyzikálnych princípov (elektorindukcia, fotoindukcia...) schopné generovať energiu a ukladať ju na podobných MEMS kondenzátoroch pre neskoršie využitie [6].

Okrem zdroja energie obsahujú zariadenia v bezdrôtových senzorových sieťach taktiež senzory, ktoré im umožňujú získavať informácie z prostredia, v ktorom sa nachádzajú. Dokážu tak sledovať stav prostredia, rovnako ako svoj stav (polohu, pohyb) a sú schopné tento stav reportovať komunikáciou s ostatnými zariadeniami v sieti až k akémusi hlavnému uzlu, ktorý túto informáciu

vyhodnocuje a následne je schopný spätne komunikovať so zariadeniami v sieti. Konrétne senzory použité na tom ktorom zariadení samozrejme závisia od zamerania siete, ale spomeňme aspoň niekoľko ich typov. Jedná sa napríklad o akcelerometer, pozičný senzor, teplotný senzor, senzor zloženia ovzdušia, mikrofón, senzor magnetického poľa atď.

Senzorové siete sú neoficiálne treťou generáciou senzorových zariadení, ktorá nasleduje aktívne RFID (Radio Frequency IDentification) zariadenia a RTLS (Real Time Location Systems). Ako sme už spomínali, zariadenia tejto tretej generácie by mali byť úplne sebestačné. Ďalej by taktiež mali byť schopné vzdialenej obsluhy, diagnózy a opravy. Tie najlepšie zariadenia už dokonca umožňujú vzdialene zmeniť aj vlastný program, bežiaci na zariadení. [3]

1.1 Praktické aplikácie

Senzorové siete ako technológia je zatiaľ aj napriek dlhodobému vývoju stále v rannom štádiu vývoja a na masové multioborové nasadenie ešte stále čakajú. Využitia tejto technológie sú zatiaľ skôr experimentálneho typu. Skutočný boom senzorových zariadení sa očakáva v najbližších rokoch a čiastočne závisí od vyriešenia niektorých technických nedostatkov, ako je napríklad aj samotná životnosť zariadenia v závislosti od zdroja energie. [3] Ich hlavné zameranie bude ale vyplývať z ich podstaty snímania okolitého prostredia. Ich využitie sa očakáva napr. v armáde pre jednoduché sledovanie nepriateľského územia, pričom rozmiestnenie senzorových zariadení by bolo v podstate iba otázkou vyslovene "rozsypania" množtva zariadení po sledovanom území. Avšak veľký priestor pre uplatnenie čaká senzorové siete aj v civilných podmienkach. Už teraz existuje sieť senzorov rozmiestnených v švajčiarskych alpách, ktoré sledujú zmeny teploty permafrostu a umožňujú tak pripraviť modelové situácie pre prípad, že by došlo k uvoľneniu skalných masívov z dôvodu topenia permafrostu. Projekt nesie názov ***Permasense*** (http://www.permasense.ch/).

To je iba jeden príklad zo širokej škály možných využití, medzi aké budú taktiež patriť zabezpečovanie vozidiel, kontrola kvality vodných rezerv, monitorovanie zariadení, sledovanie klimatických zmien alebo monitorovanie seizmických oblastí.

1.2 Senzorové zariadenia - uzly

Ako me už spomenuli, senzorové siete sú tvorené sústavou malých senzorových zariadení schopných vzájomne komunikovať prostredníctvom rádiového spojenia. V krátkosti si teda popíšeme tieto zariadenia.

Rozmerovo sa zariadenia pohybujú v jednotkách centimetrov v závislosti od typu a výrobcu. Malé rozmery sú ich veľkou výhodou v praktickom využití, pretože je tak možné ich použiť a pripevniť na ľubovoľné miesto. V tomto smere je veľmi výstižný anglický názov pre tieto zariadenia "mote" (prekl. omrvinka).

Zariadenia sa po technickej stránke skládajú z niekoľkých celkov:

- MEMS senzory [sl1]
  - Fyzikálne - magnetický, svetelný, zvukový
  - Chemické - CO, Chemické zbrane
  - Biologické - baktérie, vírusy, proteíny
- Integrované obvody
  - AD prevodník
  - Rádio
  - Výpočtová jednotka
- Ochranný kryt
- Zdroj napätia
  - Pasívny - solárny, vibračný
  - Aktívny - batéria, RF indukcia

Na trhu je veľa druhov komerčne vyrábaných senzorových zariadení spoločne s ďalšími prvkami potrebnými pre vybudovanie celej senzorovej siete. My sme pre naše potreby využili zariadenie MICAz [obr 1.2] vyrábané spoločnosťou Crossbow (http://www.xdow.com/). Podrobnú technickú špecifikáciu zariadenia nájdete v prílohe B.

Obr 1.2 - Zariadenie MICAz od firmy Crossbow

2 Programovacie prostredie

V úvode sme naznačili niečo o programovacom prostredí, v ktorom budeme pracovať. Ide o operačný systém zvaný TinyOS v kombinácii s jazykom nesC odvodeným z jazyka C, ktorý bol špeciálne navrhnutý pre vytváranie modulov a aplikácii na tomto operačnom systéme. Ich uplatnenie síce nie je obmedzené, no aj napriek tomu sa takmer výhradne využívajú v senzorových zariadeniach. Aby bolo možné vytvoriť modul pracujúci efektívne na pomerne nízkej úrovni OS, je nutné nadobudnúť hlbšie

pochopenie princípov, využívaných v tomto prostredí. Ako uvidíme, kombinácia TinyOS a nesC predstavuje princípy, ktoré nie sú úplne bežné v programátorskej praxi. Preto aj koncept vytvárania modulov a aplikácií je o niečo náročnejší a vyžaduje od programátora tak trocha "thinking outside the box" - iný myšlienkový prístup, než je v bežnej praxi využívaný.

2.1 TinyOS

S príchodom technológie senzorových sietí vznikla potreba vytvoriť operačný systém umožňujúci obsluhu zariadení v týchto sieťach. S riešením prišli v roku 1999 vedci z univerzity UC Berkley a navrhli nový operačný systém TinyOS, ktorý bol postavený na niekoľkých kľúčových bodoch:

- Malá veľkosť
- Nízka spotreba energie
- Možnosť behu aplikácií rôzneho typu
- Abstrakcia HW zariadení
- Použiteľnosť na širokej škále senzorových zariadení

Systém TinyOS tak uviedol nové princípy, z ktorých niektoré si pre zaujímavosť popíšeme:

aplikácie tvorené komponentami - aby tvorcovia docielili odtienenie HW zariadení, navrhli systém komponent (*components*), ktoré reprezentujú jak HW komponenty, tak SW moduly. Komponenty sú zapúzdrené bloky kódu, ktoré navonok komunikujú prostredníctvom rozhraní (*interfaces*). Sú tak plne zameniteľné a pri konkrétnom použití aplikácie na vybranom senzorovom zariadení nezáleží na tom, či je komponenta HW charakteru, alebo je nahradená SW modulom v prípade, že HW komponenta na zariadení nie je prítomná.

tasks (úlohy) - súčasťou operačného systému je fronta, do ktorej sa zaradzujú úlohy. Rieši sa tak konkurenčnosť, ktorá môže vznikať v dôsledku obsluhy prerušení. Jedná sa o FIFO frontu.

split-phase princíp - ďalším krokom pre odtienenie HW komponent bolo zavedenie split-phase volania metód. Je totiž známe, že veľa HW komponent nie je schopných vrátiť výsledok požiadavku hneď pri zavolaní, a potrebujú určitý čas na jeho spracovanie. Split-phase princíp rozdeľuje volanie na dve fázy. V prvej fáze sa zašle požiadavok na komponentu. Tá ale nevráti priamo výsledok. Miesto toho zaradí do fronty úloh úlohu na obslúženie požiadavku. Po dokončení obsluhy komponenta oznámi, že obsluha bola dokončená vyvolaním udalosti. Na komunikáciu teda slúžia vždy dve metódy, ktoré sú súčasťou komunikačného rozhrania.

TinyOS podlieha neustálemu vývoju a rovnako, ako sa rozširujú možnosti senzorových zariadení, rozširuje sa aj funkcionalita podporovaná v TinyOS. Modul pre dynamickú správu pamäte bol naimplementovaný a testovaný na verzii TinyOS 2.1.0. V dobe písania tohto dokumentu vyšla

novšia verzia 2.1.1, ktorá rozširuje podporu TinyOS pre ďalšie senzorové zariadenia a prináša podporu mnohých sieťových protokolov ako aj IPv6 sieťovej vrtvy.

2.2 nesC

Jazyk nesC, ako sme už spomenuli, je odvodený od jazyka C. Syntax jazykov je takmer totožná, pričom nesC kôli svojej špecifickej logike zavádza vlastné syntaktické prvky a kľúčové slová, ktoré si popíšeme neskôr.

Najpodstatnejší a najviditeľnejší rozdiel medzi jazykmi C a nesC ale spočíva v programovej štruktúre. Programy, vytvorené v jazyku C sa skladajú zo zdrojových kódov obsahujúcich funkcie a premenné, rozdelených do súborov, ktoré sú kompilované samostatne a nakoniec sú zlinkované do jedného celku - výslednej aplikácie. Naproti tomu programy, písané v jazyku nesC sú tvorené komponentami, ktoré sú v samostatných súboroch prepojené za pomoci konfigurácií. Výsledná aplikácia je kompilovaná ako celok. Komponenty aplikácie sú navzájom prepojené za pomoci rozhraní, ktoré jasne určujú obojsmernú komunikáciu medzi modulmi.

Jazyk nesC z dôvodu využívania na senzorových zariadeniach musí riešiť aj problém konkurencie požiadavkov. Podstata senzorových zariadení totiž spočíva v tom, že zariadenie, ktoré je prevažnú dobu v nečinnosti, reaguje na podnety zaznamenávané z okolia spustením obslužnej procedúry a to v závislosti od senzoru, ktorý podnet vyvolal. To má ale za následok, že požiadavok na spustenie obslužnej procedúry môže prísť v okamžik, keď iný proces práve beží v pamäti. Tento problém rieši jazyk nesC za pomoci tzv. split-phase volania úloh, kde zavolanie úlohy nemusí mať za následok okamžité vykonanie celej úlohy. Miesto toho sa úloha uloží do fronty úloh a vykoná sa až po tom, ako sa spracujú skôr zadané úlohy. Pre tieto účely poskytuje nesC tri typy funkcií:

- event (udalosť)
- command (príkaz)
- task (úloha)

pričom pre udalosti a príkazy platí, že môžu byť volané z prerušení, vyvolaných napríklad senzormi, ale len v prípade, pokiaľ sú zadefinované ako *async*, teda asynchronné. Spracovanie prerušení za pomoci asynchronnej udalosti alebo príkazu má okamžitú prednosť pred aktuálne bežiacim kódom, čo ale môže spôsobiť koflikty a nekonzistenciu dát v priebehu behu programu (napr. ak práve bežiaci proces pracuje s rovnakými dátami, s ktorými pracuje aj obsluha prerušenia). Týmto komplikáciám umožňuje jazyk nesC zabrániť s pomocou využitia bloku *atomic*.

```
atmic {
    if (lock == false) {
        lock = true;
    }
}
```

Kód uzavretý v sekcii atomic nemôže byť prerušený a je tak zaručená konzistencia dát po celý beh kódu v tomto bloku. Blok atomic má však aj istú nevýhodu. Počas behu kódu v tomto bloku sú všetky vyvolané prerušenia automaticky zahodené. Preto sa jeho využitie obmedzuje na čo najmenší kus kódu.

Aj v našom prípade budeme musieť riešiť problém s konkurenciou o prístup k datám a to vo funkciách umožňujúcich zápis a čítanie nad dynamicky alokovaným blokom pamäte. Riešenie tohto problému si bližšie popíšeme v časti 4.2 Návrh modulu.

2.3 TOSSIM

TOSSIM je simulátor diskrétnych udalostí pre TinyOS senzorové siete. Vývojár aplikácie môže namiesto kompilovania pre nejaké zariadenie skompilovať kód pre framework TOSSIM ktorý beží na PC a podrobiť program extenzívanemu testovaniu v kontrolovanom prostredí s možnosťou opakovaného behu programu. TOSSIM slúži hlavne na simuláciu procesov prebiehajúcich v TInyOS systéme a nie je vhodný pre simulovanie skutočného sveta. Nedokáže teda reálne simulovať čas behu aplikácie ani napríklad omeškania či straty paketov pri rádiokomunikácii.

Simulátor je ovládaný prostredníctvom jazyka python pomocou funkcií k tomu vyhradených. Poskytuje širokú škálu nástrojov, ktoré sú potrebné pre ladenie aplikácií ako napr. meranie času (iba orientačne v závisloti na počte prevedených procesorových inštrukcií), výpis chybových hlásení a samozrejme spúšťanie udalostí. Umožňuje vytvoriť kanály pre notifikačné/chybové hlásenia a následne umožňuje vypisovať iba vybrané kanály na štandardný výstup, čoho sme využili a vytvorili sme kanál pomenovaný TMalloc prostredníctvom ktorého vypisujeme stavové notifikácie z modulov TMalloc aj TMallocFlash. Pri vývoji aplikácie s použitím jedného z týchto modulov stačí v testovacom pythonovskom skripte pridať metódu `addChanel("TMalloc", sys.stdout)` čím tento kanál presmerujeme na štandardný výstup.

Simulátor alo neumožňuje simuláciu externej pamäte, čo v našom prípade budeme potrebovať. Na jej simuláciu preto použijeme pomocný modul VirtualBlock ktorého autorom je Ing. Jan Horáček. Ten poskytuje rovnaké rozhrania ako štandardný modul Block v TinyOS. Súbory použité pre simuláciu externej pamäte si popíšeme v sekcii Súborová štruktúra [4.5.1].

2.4 Vývojové prostredie

Pre zjednodušenie práce a skrátenie doby potrebnej na nastavenie testovacieho prostredia pripravili tvorcovia TinyOS predkonfigurovanú verziu linuxu nazvanú XubunTOS postavenú na distribúcii Xubuntu ktorá je dostupná ako virtuálny systém. Systém XubunTOS je dostupný na stiahnutie na adrese http://sing.stanford.edu/tinyos/dists/. Vývoj a testovanie modulu teda prebiehalo vo virtuálnom

systéme XubunTOS 2.1 spúšťanom za pomoci prostredia VMware. Virtuálny systém obsahuje plne nakonfigurovanú verziu TinyOS spolu s testovacím frameworkom TOSSIM.

3 Dynamická správa pamäte

Popísali sme si prostriedky, s ktorými budeme pracovať, rovnako ako prostredie, pre ktoré budeme vyvíjať. Teraz sa zameriame na samotný problém a úlohu, ktorú sa snažíme vyriešiť. Popíšeme si veľmi stručne princíp dynamickej správy pamäte, pripomenieme si pojmy, ktoré sú s touto problematikou spojené, vysvetlíme si jej uplatnenie v TinyOS a samotný postup implementácie, ktorý sme zvolili. Vzhľadom na to, že technika dynamickej správy pamäte patrí medzi základné znalosti programátora, nezávisle od jazyka alebo platformy na ktorej vyvíja, nebudeme zachádzať do prílišných detailov pri popisovaní tejto techniky.

3.1 Základný princíp

Dynamickým prideľovaním pamäte nazývame postup, keď určitý kus pamäťového bloku, nazývaný hromada *(heap)* je vyhradený a prerozdeľovaný bežiacim procesom na vyžiadanie. Z pravidla si veľkosť priradeného priestoru špecifikuje samotný proces a pokiaľ existuje voľný blok o požadovanej veľkosti, dostane proces ukazateľ *(handle)* na tento blok, s ktorým môže následne pracovať. Tento proces sa nazýva alokácia *(allocation)* Po ukončení práce s blokom, keď už tento blok nie je pre proces naďalej potrebný, má proces možnosť vrátiť blok na hromadu. Táto operácia sa zase nazýva uvoľnenie *(free)*. V niektorých prípadoch môže nastať situácia, keď proces potrebuje navýšiť veľkosť alokovaného bloku. V tom prípade požiada o realokáciu *(reallocation)* bloku na novú veľkosť. Správu dynamickej pamäte má štandardne na starosti operačný systém, ktorý musí interne udržiavať informácie o alokovaných blokoch, t.j. o ich umiestnení a veľkosti.

V bodoch si ešte zhrnieme výhody a nevýhody dynamickej alokácie.

- Výhody:
  - možnosť definovať veľkosť potrebnej pamäte až za behu programu
  - efektívnejšie využitie pamäte
- Nevýhody:
  - zvýšená réžia pri alokovaní/uvoľňovaní z pohľadu procesorového času
  - extra využitá pamäť pre uchovanie metadat

3.2 Hromada *(heap)*

Aby mohol systém sprostredkovať možnosť dunamickej alokácie, musí si najprv vyhradiť blok pamäťového priestoru, ktorý bude prerozdeľovať. Veľkosť tohto bloku, rovnako ako jeho umiestnenie je závislé od operačného systému a samotnej fyzickej veľkosti pamäte. Podstatným

faktom však je, že hromada má svoju konečnú veľkosť. Jedným z programátorských dogmat pri práci s dynamickou pamäťou je, že každý dynamicky alokovaný blok pamäte by mal byť pred ukončením programu uvoľnený. Pokiaľ tomu tak nie je, môže dochádzať k únikom pamäte a vo výsledku môže spôsobiť úplne zahltenie hromady. Novodobé operačné systémy sa však dokážu s týmto faktom vysporiadať a dokážu pamäť automaticky recyklovať.

3.2.1 Garbage collector

Jedná sa o súčasť konkrétneho programovacieho jazyka (moederné jazyky ako C#, JAVA), ktorá sleduje použitie ukazateľov na každý alokovaný blok pamäte. Ukazateľ totiž môže byť rôzne kopírovaný a duplikovaný. Pri ukončení behu programu sa garbage collector postará o automatické uvoľnenie všetkých alokovaných blokov, čím uľahčuje prácu programátora, ktorý sa tak nemusí únikmi pamäte zaoberať.

3.2.2 Súkromné hromady

Alternatívnym riešením k systémovej dynamickej pamäti sú súkromné hromady. Sú súčasťou konkrétneho programu, napríklad vo forme samostatnej knižnice alebo modulu. Ich fungovanie je založené na vyhradení veľkého, staticky alebo dynamicky naalokovaného bloku pamäte, ktorý si modul pripraví na začiatku behu programu. Tento priestor je naďalej vrámci procesu využívaný ako hromada a modul sám spravuje prideľovanie a uvoľňovanie priestoru na tejto hromade. Výhodou tohto prístupu je, že sa nemusíme báť o úniky pamäte na úrovni operačného sytému. Stačí na konci programu uvoľnit celý tento blok naraz alebo, pokiaľ vznikol statickou alokáciou, nechať uvoľnenie na operačnom systéme. Tento prístup sme zvolili aj pri implementácii modulu pre TinyOS.

3.3 Komplikácie s tým spojené

Niektoré komplikácie, spojené s dynamickou alokáciou sme si už spomenuli. Okrem extra pamäte, potrebnej na uchovavanie metadat a zvýšenej réžie sú tu však aj ďalšie problémy. Nastávajú pri použití manuálneho systému práce s alokovanóu pamäťou, tak ako to robí jazyk C. Problém vychádza z toho, že pri naalokovani priestoru nie je vrátená konkrétna premenná s ktorou by programátor mohol štandardne pracovať, ale miesto toho je vrátený ukazateľ do pamäte, ktorý ukazuje na konkrétnu pozíciu. To ale znamená, že akonáhle je procesu pridelený niektorý blok pamäte, nie je možné s daným priestorom nijak pracovať až do doby, keď program explicitne uvoľní daný blok, alebo sa program ukončí a vráti tak všetku alokovanú pamäť na hromadu. To má ale nevyhnutne za následok defragmentáciu, ku ktorej dochádza pri striedavom obsadzovaní a uvoľňovaní blokov rôznej veľkosti. V tomto sa dynamická alokácia zásadne líši od tej statickej, pri ktorej sa pamäť obsadzuje

vždy na vrchu zásobníku *(stack)* a rovnako sa z vrchu zásobníka uvoľňuje. Práca so statickou pamäťou je teda lineárna.

Samozrejme aj s týmto problémom sa dá vyporiadať, ale stojí to určité prostriedky. Tento problém sa dá obísť vytvorením akejsi virtuálnej medzivrstvy. Programy tak nedostávajú ukazateľe priamo do pamäte, ale iba do virtuálneho bloku, kde každá adresa určitého bloku je mapovaná na ľubovoľnú adresu bloku v reálnej pamäti. Táto medzivrstva môže byť sprostredkovaná operačným systémom alebo samotným jazykom. V každom prípade však tento prístup vyžaduje vytvorenie relačnej tabuľky, ktorá:

a) zväčšuje priestor potrebný pre metadata [sl4]

b) spomaľuje prácu s dynamickou správou pamäte

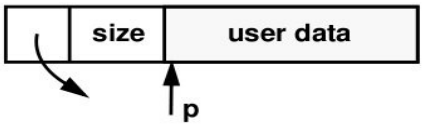
Ako si neskôr objasníme, pre naše riešenie nie je táto technika vhodná ale v jednom z riešení sme použili istú variáciu tejto virtuálnej medzivrstvy, ale značne zjednodušenú.

3.4 Rozloženie a mapovanie blokov

Základným pravidlom, podľa ktorého sa alokátory [sl2] delia je spôsob uchovávania a práce s voľnými blokmi pamäte, možnosť ich členenia a následného zlučovania. Pri spravovaní neobsadených blokov sa využíva niekoľko obecne známych postupov. Cieľom každého z prístupov je skrátiť čas potrebný na alokáciu a uvoľnenie pamäťového bloku, no každý z nich sa tohto cieľa snaží dosiahnuť iným spôsobom. Pri výbere vhodného prístupu musíme zvážiť niekoľko faktorov ako objem metadat potrebný pre uchovávanie stavu voľných blokov, rýchlosť práce s metadatami a dôležité je aj zhodnotiť predpokládané využitie a požiadavky aplikácií z hľadiska veľkosti jednotlivo alokovaných blokov. Pretože sa pohybujeme v priestore senzorových sietí, kde sú zdroje značne obmedzené, budeme sa zameriavať na spôsob, ktorý nespotrebovavá veľa prostriedkov.

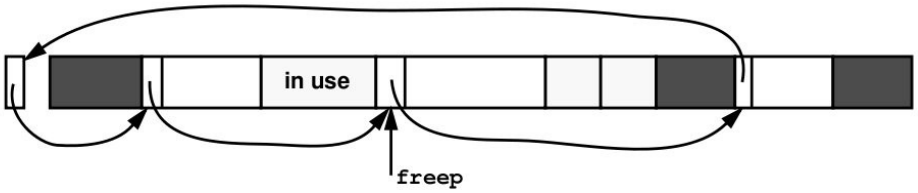
3.4.1 Zreťazený zoznam blokov

Jedná sa o postup, keď sa voľná pamäť skladá z blokov rôznej veľkosti, pričom každý z týchto blokov obsahuje hlavičku. Hlavička je štruktúra obsahujúca dve informácie. Veľkosť celého bloku (bez samotnej hlavičky) a ukazateľ na ďalší voľný blok [obr 3.1] (niekedy sa využíva i obojstranne zviazaný zoznam, ktorý udržiava aj ukazateľ na predošlý blok). Posledný voľný blok ukazuje spätne na prvý blok a uzatvára tak zoznam do smyčky. Vzniká tak zreťazený zoznam blokov rôznych veľkostí. Bloky v zozname sú zoradené tak, že na seba bloky ukazujú vždy vo vzostupnom poradí, aby bolo možné jednoduché zlučovanie voľných blokov.



Obr. 3.1 Jeden voľný blok pamäte

Na úplnom začiatku je celá hromada jedným veľkým blokom s jedinou hlavičkou, ktorá rekurzívne ukazuje sama na seba. V priebehu práce s dynamickou pamäťou sa tento blok fragmentuje na množstvo menších blokov, pričom každý z nich má vlastnú hlavičku s informáciami [obr. 3.2].



Obr. 3.2 Schéma zoznamu voľných blokov

Pokiaľ alokujeme nový blok o určitej veľkosti, prechádzame postupne celý zoznam voľných blokov. Ak nájdeme blok o vhodnej veľkosti, rozdelíme ho na dve časti, pričom časť o veľkosti, ktorú alokujeme, umiestnime na koniec tohto bloku. Tým nám stačí zmeniť hodnotu veľkosti v úvodnej hlavičke. Pri následnom uvoľňovaní použitého bloku prevedieme zlúčenie s voľnými blokmi, s ktorými tento blok súsedil (ak také existujú) do jedného celistvého bloku. Musíme pritom dbať na zachovnie náväznosti zoznamu vo vzostupnom poradí.

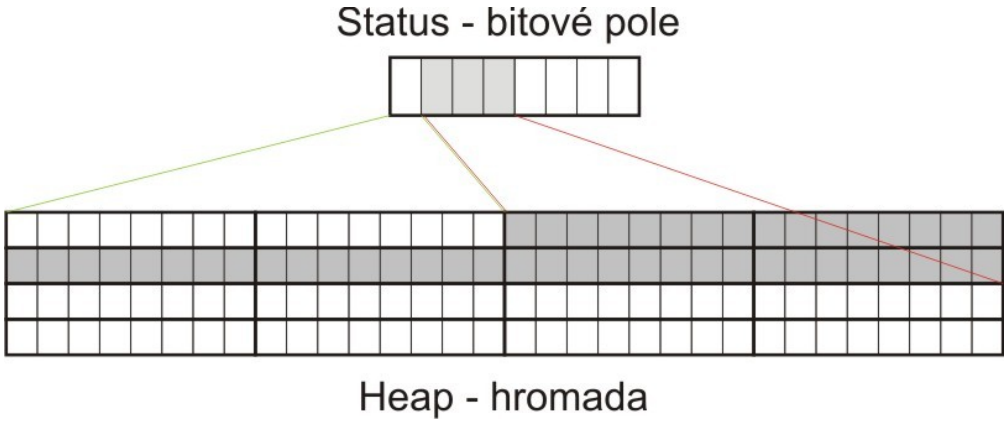
Tento prístup však má jeden podstatný nedostatok. Pokiaľ totiž programátorskou chybou dôjde k pretečeniu alokovaného pamäťového bloku, poškodia sa tým metadata ďalšieho bloku a môže tak dôjsť k úplnej devastácii dát uložených na hromade.

3.4.2 Zreťazený zoznam blokov rovnakej veľkosti

Jedná sa o obdobu prvého postupu, ale pamäť je už v úvode rozdelená na štruktúry rovnakej veľkosti ktorých súčasťou je aj informácia o naledujúcom voľnom bloku. Veľkosť alokovaných blokov je tak v každom prípade násobkom veľkosti jednej takejto štruktúry. Aby nenastával problém s ukládaním veľkých datových typov, je zvykom, že sú tieto štruktúry zarovnané podľa veľkosti najväčšieho datového typu poskytovaného systémom - tzv. omedzujúceho datového typu (napr. omedzujúci datový typ double potrebuje pre uloženie 8B - preto každá štruktúra začína na pozícii, deliteľnej 8) Pri uvoľňovaní takejto pamäte nedochádza k zlučovaniu do väčších blokov, ale naopak k opätovnému rozkúskovaniu na menšie časti. Táto technika je ale pomerne nevýhodná a takmer nepoužívaná.

3.4.3 Bitová mapa

U tohto postupu je pamäť rozdelená na bloky o rovnakej veľkosti. Nejedná sa však už o zreťazený zoznam. Tentokrát je informácia o stave jednotlivých blokov udržiavaná mimo samotnú hromadu. Slúži k tomu štruktúra v ktorej každému bloku hromady - alokačnej jednotke [sl3] - zodpovedá jeden stavový bit, ktorý je fyzicky namapovaný na konkrétnu adresu alokačnej jednotky. Ten teda nadobúda jeden zo stavov voľný/obsadený. Nevýhodou tohto prístupu je, že vyžaduje extra pamäťový priestor mimo hromadu, ktorý uchováva stav jednotlivých alokačných jednotiek v bitovej mape. Zároveň vyhľadávanie voľného priestoru je o niečo časovo náročnejšie než tomu je u zreťazeného zoznamu. Na druhú stranu však fakt, že je informácia o obsadenosti mimo hromadu, značne znižuje pravdepodobnosť poškodenia dat.



Obr. 3.3 Schéma mapovania súboru nezávislých blokov na bitovú mapu
(1 bit v bitovej mape = 16B na hromade)

Vo svojej implementácii dynamickej alokácie som zvolil tento prístup. Okrem spomínaného faktu má pre náš modul uschovanie informácií o obsadenosti mimo hromadu ešte jednu výhodu. Pri použití flash pamäte totiž čítanie dát z flash vedie k značnému omeškaniu a vyžaduje si použitie split-phase funkcií. Pri hľadaní voľného bloku by sme teda museli niekoľkonásobne načítať stavy jednotlivých blokov a po nájdení bloku by sme navyše museli zapisovať zmeny, a to všetko cez funkcie typu plit-phase. Pokiaľ ale máme informácie o alokačných jednotkách mimo hromadu, môžme previesť celú operáciu alokácie a uvoľnenia bloku bez nutnosti na hromadu pristupovať (či už kôli čítaniu, alebo zápisu).

3.5 Algoritmy správy pamäte

Rovnako ako pre rozloženie alokačných blokov existuje niekoľko prístupov aj pre vyhľadávanie vhodného voľného priestoru k alokácii. Znova ide o maximalizáciu efektivity alokačných algoritmov a každý prítup sa toho snaží dosiahnuť iným spôsobom. [13][11]

3.5.1 First Fit

Tento postup je najjednoduchší a využíva sa v prevažnej väčšine nástrojov pre správu dyn. pamäte. U tohto postupu sa postupne prechádza voľnými blokmi pamäte pričom sa pokaždé porovnáva veľkosť voľného bloku s požadovanou veľkosťou na alokáciu. Pokiaľ je taký dostatočne veľký blok nájdený, automaticky sa použije a proces prehľadávania sa zastaví. V tomto smere je teda algoritmus pomerne rýchly. Neprihliada ale na pomer veľkosti voľného bloku a požadovanej veľkosti a tak má za následok značnú fragmentáciu hromady, ktorá následne spomaľuje alokačný proces. Obzvlášť vysoká fragmentácia nastáva v úvodnej časti hromady, ktorá je po určitom čase rozkúskovaná na množstvo malých blokov.

Na druhú stranu je však tento prístup nenáročný jak zo strany množstva metadat, tak aj zo strany množstva potrebného zdrojového kódu pre jeho implementáciu. Preto a pre jeho obľúbenosť sme tento prístup zvolili aj pri implementácii naších modulov.

3.5.2 Best Fit

Druhý postup, ktorý je z 90% totožný z predchádzajúcim je nazvaný best fit. Pri tomto postupe sa už berie v potaz aj pomer veľkosti voľného bloku a veľkosti požadovaného bloku. Pri prechádzaní voľnými blokmi sa prechádza celá voľná pamäť (teda všetky voľné bloky) a hľadá sa blok, ktorý po obsadení zanechá najmenej voľného priestoru - teda pomer veľkostí je najnižší. Tento prístup so sebou prináša mierne navýšenie počtu potrebných operácií pre nájdenie vhodného bloku. Umožňuje však lepšie využiť priestor na hromade a nevyžaduje podobne ako prvý prístup žiadne extra metadata. Z reálneho testovania však bolo zistené, že ani tento prístup nie je efektívnejší vo všetkých prípadoch.

Uveďme si príklad. Na hromade sa nachádzajú dva bloky o veľkosti 30B a 20B. Systém obdrží požiadavok na alokovanie blokov o veľkosti 15B, 15B a 20B. Pri použití princípu best fit nebudeme mať voľný priestor, kam by sme mohli 20B blok umiestniť, zatiaľ čo s použitím first fit dokonale využijeme voľný priestor. Výber postupu teda nie je vôbec jednoznačný a podľa slov Knutha, pokiaľ sa blíži vyčerpanie pamäte, eventuálne pamäť dôjde, nezávisle od toho, ktorý prístup zvolíme [14]. Preto je vhodné vybrať si najjednoduchšie implementovateľný alogritmus - first fit.

3.5.3 Good Fit

Princíp good fit je voľnejšia verzia princípu best fit. Dôsledné dodržiavanie princípu best fit je totiž veľmi náročné na réžiu. Namiesto hľadania ideálneho bloku sa použije voľný blok, ktorého veľkosť približne zodpovedá požadovaniej veľkosti. Približne v tomto prípade znamená, že rozdiel veľkostí je vrámci istej pevne danej medze, napr. do +20%. Ak teda chceme alokovať blok o veľkosti 100B tak použijeme prvý voľný blok o veľkosti menšej alebo rovnej 120B.

3.5.4 Worst Fit

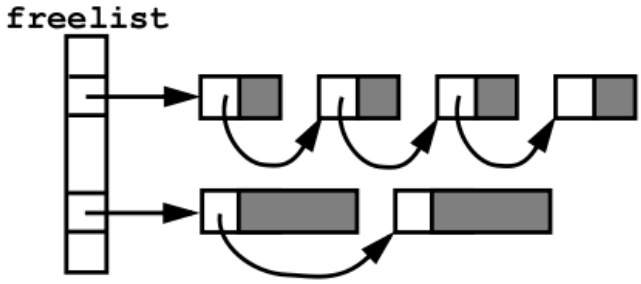
Ide o opačný prístup, keď sa pre alokovanie použije aktuálne najväčší voľný blok. Voľné bloky sú uchovávané v zozname zoradené od najväčšieho bloku. Cieľom tohto postupu je rovnomerne ukrajovať zo všetkých neobsadených častí, no v reálnom použití sa jedná o veľmi neefektívny spôsob, ktorý vedie k prehnanej fragmentácii hromady. Tento algoritmus sa prakticky nepoužíva.

3.5.5 Next fit

Tento prístup je rozšírením algoritmu first fit a jeho cieľom je zamedziť nadmernej fragmentácii v úvodnej časti hromady. Tá je spôsobená tým, že pri hľadaní vhodného bloku sa voľné bloky prechádzajú vždy od začiatku zoznamu. Next fit tento problém rieši tak, že pri každom následujúcom hľadaní sa pokračuje od posledne použitého voľného bloku a tak je celá hromada využívaná a rozdeľovaná rovnomerne.

3.5.6 Quick Fit (Segregated fit)

Tretí prístup sa špecializuje na rýchlosť operácie vyhľadania voľného bloku. Ide o kombináciu prvých dvoch prístupov pričom bloky sú rozdelené do niekoľkých skupín podľa veľkosti a pri požiadavku na alokáciu bloku sa použije vyhľadávanie typu First Fit ale iba nad skupinou, ktorá obsahuje bloky odpovedajúcej veľkosti. Tento prístup ale navyše vyžaduje rozšírené metadata a nie je vhododný pri použití rozdelenia hromady za pomoci bitovej mapy [3.4.3]. Vyžaduje totiž pravidelné preskupovanie a úpravy zoznamov voľných blokov, čo je u tohto rozdelenia hromady pomerne náročná operácia.



Obr. 3.4 Rozdelenie voľných blokov do niekoľkých zoznamov podľa veľkosti

3.6 Fragmentácia

Efektivita využitia pamäťového priestoru je priamo závislá od vzniklej fragmentácie. Niektoré zo spomínaných algoritmov majú práve za cieľ znižovanie fragmentácie vhodným výberom umiestnenia alokovaného bloku. Fragmentácia ale rovnako závisí od schopnosti alokátoru vhodne rozdeľovať a spájať voľné bloky, prípadne predvídať chovanie programu (napr. pokiaľ je známe, ako sa bude dynamická správa využívať, je možné algoritmus v danom smere optimalizovať).

- **Externá fragmentácia** - vzniká z dôvodu zaokrúhľovania alokovaných blokov na určitú výhodnú veľkosť. Napr. z dôvodu uchovávania niektorých datových typov (*double*) je nutné aby blok vždy začínal na pozícii deliteľnej 8. Alebo v našom prípade, keď máme minimálnu veľkosť alokovateľnej jednotky určenú na 16B, je to zaokrúhľovanie veľkosti bloku na násobky 16. Tým ale vzniká na konci bloku nevyužitý priestor, ktorý je možné využiť až po tom, ako bol celý blok zrecyklovaný. V tomto prípade však výhody plynúce zo zaokrúhľovania prevyšujú nevýhodu externej fragmentácie a pri vhodnej voľbe veľkosti alokačnej jednotky nie je strata pamäti v jej dôsledku tak markantná. Prevažne sa pre veľkosť alokačných jednotiek volia hodnoty rovné mocninám 2 (8, 16, 32, 64). Programátor, ktorý pracuje s takýmto alokátorom a plánuje nadmerne využívať dynamické prideľovanie pamäte by si mal dopredu zistiť, aká bude veľkosť alokačnej jednotky u ním zvolenej správy pamäte na konkrétnej architektúre, aby mohol efektívne využiť celý pamäťový priestor.

- **Interná fragmentácia** - je prirodzeným výsledkom často sa opakujúceho obsadzovania a uvoľňovania blokov o rôznych veľkostiach a pri štandardných alokátoroch [sl2] (poskytujúcich priamy prístup do pamäte) nie je možné sa jej vyhnúť. Jediný spôsob ako ju obmedziť je vhodným výberom umiestnenia alokovaných blokov. V tomto smere je asi najlepšou voľbou algoritmus best fit [3.5.2] hoci aj tento môže v určitých prípadoch zlyhať (ako sme si uviedli na príklade).

  Ak ale vytvárame alokačný mechanizmus, ktorý nebude programu umožňovať priamy prístup do pamäte, dá sa tento problém vyriešiť pomerne jednoducho kompresiou (defragmentáciou) dát na hromade. S kompresiou je spojená značná réžia potrebná na kopírovanie dát z miesta na miesto, preto by mala byť použitá ako posledná možnosť, ak nastane stav, že už nie je možné nájsť voľný blok o dostatočnej veľkosti. [14]

3.6.1 Kompresia

Jedná sa o zhustenie dat na hromade tak, aby sa všetky fragmentované časti voľnej pamäte spojili v jeden celistvý blok, najčastejšie na konci hromady. Obsadené bloky sa pritom prechádzajú jeden za druhým a ich obsah je presunutý tesne za ten predchádzajúci. Pri tom však dochádza k zmene adresy každého z presúvaných blokov, preto tento postup nie je možný u alokátorov s priamym prístupom do pamäte. Všetky ukazatele na hromadu v programe by sa totiž stali nevalidnými, následkom čoho by došlo k úplnému zničeniu uložených dat.

Kompresia však môže byť problematická aj v prípade, keď je ako hromada využívaná pamäť s pomalým prístupom. Práve práca s flash pamäťou, ktorú implementujeme v jednom z našich modulov patrí k takýmto prípadom. Presuny veľkého množstva dát vrámci externej pamäte je časovo

neporovnateľne náročnejšie než sú štandardné operácie na zariadeniach, pre ktoré implementujeme (napr. čítanie dát zo senzorov) a počas tohto procesu by sa mohol stav zariadenia zmeniť natoľko, že by už operácia nebola aktuálna (napr. že by sme z dôvodu kompresie stratili niekoľko záznamov zo senzoru, ktoré sme vlastne mali do alokovanej pamäte zapisovať). [14]

3.7 Vyčerpanie pamäte

Pokiaľ program, pracujúci s dynamicky alokovanou pamäťou obsahuje chybu spôsobujúcu úniky pamäte, alebo začne neočakávane využívať dynamickú pamäť extenzívnejšie než bolo pri jeho imlementácii plánované, začne sa priestor na hromade postupne zmenšovať, až dôjde k úplnému vyčerpaniu pamäte. Úplné vyčerpanie ale v našom prípade ale nemusí (a v 99% prípadov ani neznamená), že na hromade už nie je žiadne voľné miesto. V skutočnosti môže byť hromada využitá iba z časti, ale program alokátora požiadal o blok, ktorý svojou veľkosťou presahuje každý z aktuálne voľných blokov pamäte.

Správne implementovaný systém správy pamäte musí byť schopný tento stav riadne spracovať a poskytnúť o tom programu náležitú informáciu. Samotý alokátor by nemal riešiť tento problém ukončením ani iným inváživnym spôsobom. Bežiaci program totiž môže byť schopný na túto udalosť reagovať a uvoľniť nepoužívané bloky k ďalšiemu použitiu. Štandardom u jednoduchých alokátorov je vrátenie prázdnej hodnoty *NULL*, ktorú by mal programátor pri každom pokuse o alokáciu alebo realokáciu testovať. Pri realokácii nesmie dôjsť k poškodeniu naposledy používaného bloku. Ak pracujeme s alokátorom nevracajúcim ukazatele do pamäte, mal by sa chybový stav zasielať iným spôsobom, napríklad cez parameter predávaný odkazom. U modulu TMallocFlash, ktorý práve takýmto spôsobom pracuje môže byť táto informácia predávaná ako parameter funkcie a to vďaka princípu split-phase [2.1].

4 Modul TMalloc a TMallocFlash

Objasnili sme si jak prostredie tak samotný problém, ktorý sme sa snažili vyriešiť a komplikácie spojené s riešením tohto problému. Prejdime teda k popisu samotného modulu a spôsobu, akým sa nám podarilo dosiahnuť cieleného výsledku.

4.1 Zámer

Ako sme v úvode spomenuli, a taktiež ako to vyplýva zo zadania práce, snažili sme sa vytvoriť dva riešenia správy dynamickej pamäte. Rôzne aplikácie totiž majú rôzne požiadavky a tak sme chceli navrhnúť riešenia, z ktorých si bude môcť vývojár konkrétnej aplikácie vybrať v závislosti od toho, či

bude jeho požiadavkom rýchlosť alebo veľkosť ponúkaného priestoru na prácu. Prvé riešenie pracuje výlučne s pamäťou RAM, na ktorej má vyhradený blok o vopred špecifikovaniej veľkosti. Pracovne sme tento modul nazvali TMalloc. U tohto modulu sme sa zamerali na jednoduchosť rozhrania a rýchlosť samotného procesu správy dynamickej pamäte. Druhým riešením je modul operujúci jak na pamäti RAM tak aj na časti flash pamäte, ktorá je súčasťou senzorového zariadenia MICAz. U tohto riešenia nám šlo hlavne o rozšírenie veľkosti hromady na niekoľkonásobok pamäte RAM a umožniť tým prácu s podstatne väčším množstvom dat. Celková veľkosť hromady je 64kB a je rozdelená na oboch typoch pamäte. Pracovne sme tento modul nazvali TMallocFlash. Rovnako ako moduly sme pre jednoduchosť pomenovali aj rozhrania, ktoré tieto moduly poskytujú.

4.2 Návrh

Popíšme si základné myšlienky, na ktorých sme stavali pri vytváraní modulu:

- jednoduchosť rozhrania
- zachovať podobnosť s obecne známým rozhraním dyn. alokácie z jazyka C
- čo najvyššie využitie pamäťového priestoru pri zachovaní rozumnej miery réžie
- využitie možností poskytovaných použitým prostredím (jayzka NesC)

4.2.1 Konvencie

Aby sme zjednodušili samotnú implementáciu modulu, stanovili sme si na začiatku niekoľko konvencií, ktoré vrámci modulu platia.

1. Granularita 16B - minimálny alokovateľný blok má veľkosť 16B. Tento, na prvý pohľad neefektívny prístup nám následne ušetrí veľa priestoru v sekcii metadat. Každý požiadavok na alokovanie bloku sa teda zaokrúhli smerom hore na násobok 16B, aj keď tento priestor nebude využitý.
2. Veľkosť alokovaného bloku sa uchováva v násobkoch 16B - vyplýva to z prvej konvencie ako efektívny spôsob ukladania informácie o bloku.
3. Obsah bloku hneď po alokácii nie je definovaný - podobne ako v jazyku C neprebieha po alokácii naplnenie bloku nulovými hodnotami. Nie je tak pevne dané, aké data sa na alokovanom bloku môžu nachádzať.
4. Počet alokovateľných blokov je obmedzený na 100 - počet alokovateľných blokov musí byť pevne daný. Podrobnejšie si tento bod popíšeme v sekcii metadata[4.2.3].
5. Veľkosť alokovaného bloku musí byť uchovateľná v 1B - táto konvencia má za cieľ zefektívniť prácu s metadatami. Podrobnejšie si tento bod popíšeme taktiež v sekcii metadata.

4.2.2 Úložište dat

Naše moduly fungujú na princípoch tzv. súkromnej hromady. Pri tomto prístupe sme si v module vytvorili statické pole o pevnej, vopred konštantne definovanej veľkosti. Fyzická veľkosť tohto poľa by mala byť navrhnutá tak aby využívala čo najviac z nevyužitého pamäťového priestoru na RAM. My sme túto veľkosť bloku v pamäti RAM u testovacieho programu určili na 4096B.

4.2.3 Metadata

K uchovaniu stavu dynamickej pamäte a alokovaných blokov je potreba niekoľko typov metadat. Samozrejmosťou je uchovávanie veľkosti alokovaného bloku. Už na začiatku sme si zvolili konvenciu, ktorá stanovuje, že veľkosť alokovaného bloku bude uchovávaná v násobkoch minimálnej alokovateľnej jednotky. To nám môže pri plnom využití maximálneho počtu alokovateľných blokov vo výsledku ušetriť až 100B pretože pri využití tohto postupu sme schopní uložiť informáciu iba na 8 bitoch pamäte. V oboch moduloch sme použili trochu odlišný spôsob pre ukladanie tejto veľkosti. V module TMalloc sme zvolili prístup, kde informácia o veľkosti bloku je súčasťou samotného bloku a to hneď na prvom bajte toho istého bloku. Tento prístup je známy napr. z ukladania informácií o poliach v jazyku C. U modulu TMallocFlash sme použili odlišnú metódu. Veľkosť bloku je ukládaná v samostatnej štruktúre, ktorá zároveň uchováva aj ďalšie informácie o alokovanom bloku. U tohto modulu sa totiž informácia o veľkosti bloku bude používať v podstatne väčšej miere ako u prvého modulu, a to obzvlášť kôli metódam write a read. Uloženie do štruktúry s priamym prístupom je preto oveľa rýchlejšia. Nevýhodou ale je, že tieto metadata sú alokované staticky a teda obsadzujú miesto v pamäti počas celého behu programu a znižujú tak efektivitu využívania pamäte v porovnaní s modulom TMalloc.

Porovnanie možstva pamäte potrebného pre uloženie informácie o veľkosti blokov:

TMalloc: $M_1 = bl\_n$

TMallocFlash: $M_2 = 100$

Pomer: $M_2/M_1 = 100/bl\_n$

kde `bl_n` je celkový počet naalokovaných blokov. Je možné vidieť, že pokiaľ budeme v programe používať jeden veľký alokovaný blok, bude efektivita ukladania tejto informácie až 100-násobne vyššia u prvého prítupu. To ale nie je úplne pravda. Prístup ukladania informácie na prvý bajt bloku totiž môže pôsobiť značný nárast nevyužitého alokovaného priestoru. Stane sa tak v prípadoch, keď bude veľkosť alokovaného bloku deliteľná veľkosťou alokovateľnej jednotky. V tom prípade musíme kôli uloženiu jedného extra bajtu zvačšiť alokovaný blok o celú alokovateľnú jednotku - 16B. Vznikne nám tak 15B nevyužitého priestoru. V najhoršom prípade teda môže nastať táto situácia:

$$M_1 = bl\_n + (bl\_n * 15)$$

kde (*bl_n* * 15) je nevyužitá alokovaná pamäť, čo pri využití maximálneho počtu alokovateľných jednotiek zmení pomer obsadeného priestoru na:

$$M_1 = 100 * 16$$

$$M_2 = 100$$

Pomer: $M_2/M_1 = 100/1600$

a v tomto prípade je ukladanie informácie do statickej štruktúry 16-násobne efektívnejšie. Nedokážeme povedať, ako často nastáva tento stav, keď program alokuje blok presne o veľkosti deliteľnej 16B, pretože je to individuálne program od programu. Skutočné porovnanie efektivity využitia pamäte závisí teda hlavne od konkrétneho programu ktorý s dyn. pamäťou pracuje.

Aby sme pochopili, prečo sme zvolili rôzne prístupy u obidvoch modulov, musíme vysvetliť princíp práce s hromadou. Tie si vysvetlíme za chvíľu v sekcii Princíp alokácie [4.2.4]. Pred tým ale spomenieme poslednú súčasť metadat. Aby sme totiž mohli efektívne určiť pozíciu, kde v pamäti sa nachádza voľný blok o potrebnej veľkosti, musíme si uchovávať stav obsadenosti jednotlivých alokovateľných jednotiek. K tomu nám slúži extra premenná. Jedná sa o bitové pole, ktoré odzrkadľuje fyzické rozloženie alokovateľných blokov na hromade. Každý blok je v bitovom poli reprezentovaný samostatným bitom a ten môže nadobúdať jeden z dvoch stavov - voľný / obsadený. Celkovo teda bitové pole zaberá priestor o veľkosti 4096/16/8 = 32B. U modulu TMallocFlash musíme navyše uchovávať dve takéto polia, kde jedno mapuje časť hromady nachádzajúcu sa na pamäti RAM a druhé mapuje časť nachádzajúcu sa na pamäti flash. Celková veľkosť hromady odpovedá veľkosti adresovateľného poľa pomocou 16B ukazateľov, teda 64kB. To znamená bitové pole o veľkosti 65536/16/8 = 512B. Modul TMallocFlash kôli odlišnému prístupu k práci s blokmi potrebuje ku svojmu fungovaniu ešte ďalšie metadata v celkovom objeme 400B (informácie o využívaní bloku, fyzická pozícia bloku na hromade).

Okrem týchto informácií sú k správe potrebné ešte ďalšie metadata, ktoré však objemom nie sú až tak podstatné a sú u oboch modulov zhodné.

Na záver tejto časti si teda porovnajme objem metadat potrebných u každého z dvoch modulov.

	TMALLOC	TMALLOCFLASH
Veľkosti blokov	1B - 100B	100B
Ďalšie info o blokoch	200B	400B
Obsadenost pamäte	32B	512B
Ostatné metadata	~20B	~20B
	253B - 353B	1032B

veľkosť blokov - uschovanie informácie o veľkosti všetkých alokovaných blokov.

ďalšie info o blokoch - informácie o pozícii blokov, časový príznak posledného použitia.

obsadenosť pamäte - veľkosť bitového poľa označujúceho stav každej alokačnej jednotky.

ostatné metadata - odkladacie premenné, buffer, pomocné ukazatele

Pozorujeme teda markantný rozdiel v množstve metadat u oboch modulov.

4.2.4 Princíp alokácie

Princíp alokácie vo svojej podstate nie je veľmi komplikovaný. Pokiaľ to hodne zjedodušíme, dá sa alokácia popísať niekoľkými krokmi. Prvým krokom je vyhľadanie bloku voľnej pamäte. K urýchleniu tohto procesu nám slúži práve spomínané bitové pole obsadenosti. Po tom, ako takýto blok nájdeme, zistíme jeho adresu (skutočnú v prípade TMalloc alebo relatívnu v prípade TMallocFlash). Uložíme adresu a informácie o veľkosti bloku ako metadata. Nakoniec vrátime index, s ktorým môže program ďalej pracovať.

Realokácia je vo svojej podstate len rozšírená alokácia. Jediná zaujímavá situácia nastáva pri realokácii na veľkosť väčšiu ako bola predošlá veľkosť bloku (v skutočnosti je to prevažný spôsob využitia realokácie). Postup je zložený z alokácie bloku o novej veľkosti, následnom skopírovaní dat zo starého bloku a uvoľnenie starého bloku. Posledným krokom zaktualizovanie metadat s informáciami o tomto bloku.

Tieto princípy sú totožné pre oba moduly. Doteraz sme popisovali oba dva moduly hlavne v paralelách a sústredili sme sa prevažne na spoločné črty. V princípe fungovania sú však jednotlivé prístupy k implementácii dosť odlišné. Ako, to si popíšeme v následujúcich dvoch sekciách.

4.3 TMalloc

Jedným z mojich cieľov u tohto modulu bolo zachovanie rovnakej syntaxe a sémantiky ako v jazyku C, vzhľadom k blízkej príbuznosti jazykov C a nesC. Poskytované rozhranie modulu TMalloc by teda malo obsahovať funkcie na dynamickú správu pamäte známé z jazyka C: *tmalloc*, *trealloc* a *tfree.* Programátorovi sme chceli poskytnúť priamy prístup do pamäte k alokovaným blokom cez ukazateľ. Jedná sa teda o rovnaký prístup, aký poskytuje funkcia *malloc* jazyka C. Pretože funkcie *realloc* a *free* pracujú priamo s ukazateľom, ktorý predáva do funkcie programátor (respektíve samotný program), je pri tomto prístupe praktické uložiť metadata na začiatok daného alokovaného bloku a programu predáme adresu na následujúci bajt pamäte. Zoznam adries alokovaných blokov sa interne uchováva, aby bolo možné kontrolovať a validovať uvoľňovanie prípade realokáciu týchto blokov. Aj napriek tomu, že sa snažíme o minimalistický návrh v prípade tohto prvého modulu, považujem za vhodné kontrolovať tieto operácie a ukazatele s ktorými pracujú. Znamená to však extra priestor pre metadata (položka "ďalšie info o blokoch" v porovnaní objemu dat pre metadata [4.2.3]) a zároveň istá extra réžia.

4.3.1 Interface TMalloc

Uživateľské rozhranie modulu TMalloc je zamerané na jednoduchosť. Poskytuje tri štandarné funkcie *talloc, trealloc* a *tfree*. Všetky tri funkcie sú implementované ako jednoduché funkcie typu *command* a ich výsledok je dostupný hneď po zavolaní v návratovej hodnote. Ich syntax je rovnaká ako u funkcií *malloc* a *realloc* jazyka C a neúspech je špecifikovaný návratovou hodnotou typu NULL. Funkcia *tfree* končí vždy úspešne.

4.4 TMallocFlash

Pri tomto module sa návrh musel radikálne zmeniť. Je to zapríčinené použitím rozhrania pre prácu s pamäťou flash o ktorom sa ešte zmienime neskôr. U tohto modulu teda už nedávame programátorovi možnosť pracovať priamo s ukazateľom do pamäte, pretože hromada už nie je celá adresovateľná prostredníctvom ukazateľov. Miesto toho sú informácie o každom alokovanom bloku uložené ako štruktúra, ktorá drží informáciu o relatívnej pozícii bloku, jeho veľkosti a využívaní. Všetky metadata o blokoch sú teda uložené v poli takýchto štruktúr a programátor po zavolaní funkcie *talloc* a úspešnom vyhradení bloku dostane späť iba index do tohto poľa. Celá práca s alokovaným blokom teda prebieha výhradne cez rozhranie poskytované modulom za pomoci tohto indexu. Týmto docielime odtienenie rozdielu medzi pamäťou RAM a flash z pohľadu programátora. Komunikácia s modulom sa tým značne zozložiťuje, ale je to zároveň jediná možnosť ako pracovať s pamäťou flash.

Teraz, keď sme si stručne popísali základné princípy za alokáciou v oboch moduloch, je pochopiteľné, prečo je výhodnejšie ukladať u modulu TMalloc metadata priamo do alokovaného bloku zatiaľ čo v module TMallocFlash do staticky alokovanej štruktúry. Vďaka tomu nemusíme pri alokovaní a uvoľňovaní pristupovať na externú pamäť.

4.4.1 Flash a rozhrania BlockRead/BlockWrite

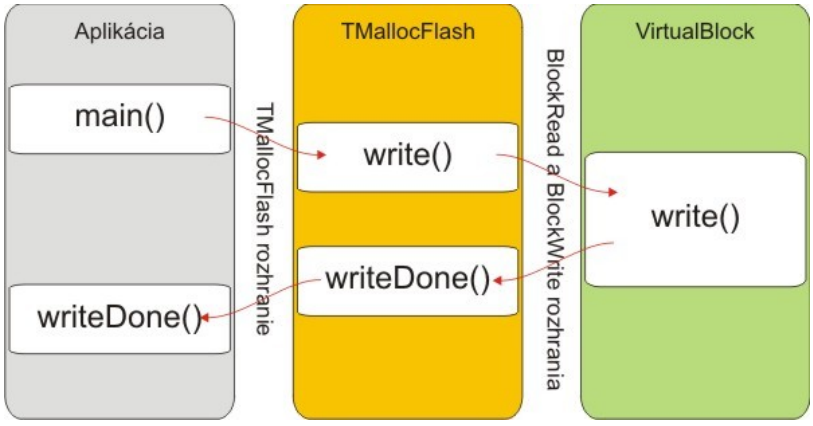
Práca s externou pamäťou v TinyOS je technicky náročnešia na pochopenie. Má totiž niekoľko prístupov. Externá pamäť je používaná troma rôznymi spôsobmi. V závislosti od napojenia bloku v konfigurácii programu špecifikujeme, ktorú časť externého bloku budeme využívať akým spôsobom. Umožňuje nám to vlastnosť NesC nazvaná *parametrizované rozhrania*.

Externá pamäť, nazývaná v TinyOS ako Block Storage je za pomoci parametrizovaného rozhrania rozdelená na dva priestory z ktorých každý ponúka svoje vlastné rozhranie za pomoci ktorého sa dá s týmto priestorom pracovať. Prvý priestor umožňuje zápis iba pridávaním na koniec (*append-only*) a streamované čítanie. Využíva sa prevažne pre ukladanie dát zo senzorov. Druhý priestor umožňuje náhodný prístup (*random-access*) a môže byť voľne využívaný na ľubovoľné operácie. Tento priestor my využijeme na rozšírenie hromady pre dynamickú pamäť. Rozhrania, ktoré tento modul poskytuje sú rozdelené na rozhranie pre zápis *BlockWrite* a rozhranie pre čítanie

BlockRead. Funkcie, ktoré tieto rozhrania obsahujú sú však takmer výlučne split-phase z dôvodu omeškania externej pamäte v porovnaní s operáciami prebiehajúcimi v systéme. To ale pre náš modul pôsobí značné problémy. Je totiž nutné udržiavať vrámci modulu informácie o aktuálne prevádzanej operácii s externou pamäťou medzi zavolaním a potvrdením operácie a hlavne informácie za akým účelom bola daná operácia prevedená. Zápis dát na externú pamäť totiž môže byť vyvolaný niekoľkými rôznymi procesmi (realokácia bloku na flash pamäti, presun bloku vrámci flash pamäte a rovnako aj samotný zápis dát vyvolaný aplikáciou) ale o každom ukončenom zápise nás rozhranie informuje vyvolaním udalosti *writeDone* respektíve *syncDone* teda na 3 rôzne dôvody zápisu nás rozhranie upozorní protredníctvom jedinej funkcie. Musíme teda zhodnotiť, ktorý z možných algoritmov zápis do pamäte vyvolal a pokračovať vo vykonávaní (resp. dokončení) daného procesu. Toto je však obecný problém split-phase funkcií a jeho riešenie býva často dosť ťažkopádne.

4.4.2 Interface TMallocFlash

Rozhranie TMallocFlash je v porovnaní s rozhraním TMalloc značne rozšírené. Okrem štandardných funkcií *tmalloc, trealloc* a *tfree* poskytuje aj funkcie pre čítanie a zápis *tread* a *twrite*. Ako sme už uviedli, je to zapríčinené tým, že prístup do flash pamäte nie je možný za pomoci jednoduchých ukazateľov. Tieto dve funkcie tak tvoria abstraktnú vrstvu nad rozhraniami BlockRead a BlockWrite a zároveň ich značne zjednodušujú. Navyše je však rozhranie rozšírené o párovú funkciu ku každej zo spomínaných funkcií, okrem funkcie *tfree*. Rovnako aj tento jav je zapríčinený rozhraniami komunikujúcimi s pamäťou flash. Všetky funkcie v rozhraniach BlockRead a BlockWrite sú totiž **split-phase**. To znamená, že výsledok požiadavku na pamäť flash nie je dostupná hneď po odoslaní požiadavku. Pokiaľ teda chceme zapúzdriť funkciu, ktorá je implementovaná ako split-phase, musíme taktiež vytvoriť split-phase funkciu.



Obr. 4.1 Diagram priebehu volania zapúzdrenej funkcie typu split phase

Každému z týchto split-phase príkazov teda odpovedá udalosť oznamujúca ukončenie požadovanej operácie. Nevýhodou je zložitejšia integrácia rozhrania do implementovaného programu, pretože, ako vyplýva zo špecifikácie jazyka NesC, každá udalosť používaného rozhrania

musí mať svoju definíciu v implementovanom programe/module, nezávisle od toho, či bude odpovedajúci *command* v rámci kódu použitý.

Aby sme však vyvážili tieto nevýhody, musíme poukázať aj na kladné stránky tohto rozhrania. Jedným z výrazných výhod je zabezpečenie práce s ukazateľmi. Tým, že programu nie je známa adresa alokovaných blokov, nepoužíva program priamy prístup do pamäte cez ukazatele. To zabraňuje aby nastala situácia, keď sa program pokúsi zapísať alebo čítať data cez nevalidný pointer. Toto je častý problém s ktorým sa potýkajú programátori pri práci s dynamickou pamäťou a po programátorovi to vyžaduje kontrolu stavu a obsahu ukazateľov na dynamicky alokované bloky. Modul rovnako kontroluje rozmedzie alokovaného bloku a neumožňuje zápis ani čítanie dát mimo hranice konkrétneho jedného bloku.

4.5 Štruktúra modulu

Ako sme v úvode spomenuli, programy v TinyOS sa zvyčajne skládajú z niekoľkých súborov. Tie sú logicky rozdelené podľa obsahu na *modul*, *rozhranie* a *konfiguráciu*. Rovnakým pravidlám podliehajú aj moduly systému.

4.5.1 Súborová štruktúra

Pri pomenovávaní súborov platia v TinyOS isté zaužívané konvencie, ktoré ale nijak neovplyvňujú fungovanie programu/modulu. Znak "P" na konci názvu znamená, že sa jedná o privátny modul, ku ktorému by program nemal priamo pristupovať. Znak "M" zase značí, že sa jedná o modul, ktorý je určený k využívaniu v iných programoch. Znak "C" znamená, že ide o komponentu a slúži k odlíšeniu komponent a rozhraní. Rozhranie má rovnaký názov ako komponenta, ktorá ho implementuje ale bez kocnového C. Každý z modulov ktoré sme implementovali pozostáva z troch základných súborov.

- modul - Obsahuje implementáciu všetkých funkcií a metod rozhrania. Nachádza sa v súboroch *TMallocP.nc* respektíve *TMallocFlashP.nc*.
- komponenta - Obaľuje privátny modul a definuje napojenie na ostatné používané komponenty a moduly cez ich rozhrania. Nachádza sa v súbore *TMallocC.nc* respektíve *TMallocFlashC.nc*.
- rozhranie - Poskytuje kompilátoru informácie o komunikačnom rozhraní pre konkrétny modul, prostredníctvom ktorého následne kompilátor namapuje jednotlivé volania metód medzi komponentami. Jeho názov by mal byť totožný s názvom komponenty, na ktorú sa viaže, ale bez koncového C. V našom prípade ide o súbory *TMalloc.nc* a *TMallocFlash.nc*.

Toto sú základné súčasti oboch modulov. Modul TMallocFlash ale pro svoje fungovanie potrebuje ešte hlavičkový súbor *TMallocFlash.h* obsahujúci definície konštant a štrukturovaných datových typov.

- ďalšie požadované súbory - Modul TMallocFlash pre svoje fungovanie potrebuje ešte niekoľko ďalch súborov, ktoré nie sú priamo súčasťou modulu, ale poskutyjú nevyhnutné informácie pre komunikáciu modulu s pamäťou flash. Štandardne sú tieto súbory obsiahnuté priamo v sytéme. Týmito súbormi sa potrebujeme zaoberať iba v prípade, že chceme aplikáciu testovať v simulátore (TOSSIM), ktorý štandardne nepodporuje simuláciu komunikácie s externou pamäťou. Ide o súbory *StorageBlock.nc*, *StorageVolumes.h* a *Storage_chip.h*.

  Posledný súbor je XML špecifikácia veľkosti bloku s náhodným prístupom na externej pamäti [4.4.1]. Názov a obsah súboru závisí od platformy, pre ktorú programátor vyvíja konkrétnu aplikáciu:

  - volumes-at45db.xml - pre platformu MICA
  - volumes-stm25p.xml - pre platformu Telos
  - volumes-pxa27xp30.xml - pre platformu Intelmote (imote)

4.5.2 Programová štruktúra

Z programového hľadiska ide o dvojvrstvovú architektúru, kde spodnú vrstvu vytvára privátny modul implementujúci všetky funkcie rozhrania a nad ňou sa nachádza vrstva tvorená komponentou obsahujúcou konfiguráciu. Tá definuje prepojenie privátneho modulu s komponentou obsluhujúcou externú pamäť prostredníctvom rozhraní BlockRead a BlockWrite. Tieto štandardne oddelené rozhrania pre prístup k externej pamäti modul zapúzdruje do vlastného celistvého rozhrania umožňujúceho jak čítanie, tak i zápis.

4.5.3 Nasadenie a použitie

Aby sme mohli modul využívať, je potreba pri kompilácii nastaviť inkludovanie súborov modulu. V rámci Makefile nastavíme cestu pre inkludovanie prostredníctvom príznaku `PFLAGS=-I[Cesta].` Vhodným a preferovaným riešením je umiestnenie modulu medzi ostatné moduly sytému do adresára `tinyos-2.1.0/tos/lib/tmalloc` aby bol dostupný všetkým implementovaným aplikáciám. Príznak pre nastavenie vrámci Makefile pre toto umiestnenie je `PFLAGS=-I$(TOSDIR)/lib/tmalloc`.

Ďalším krokom je napojenie aplikácie na rozhranie v konfigurácii. Jedná sa o jednoduché neparametrizované rozhranie:

```
configuration AplikaceAppC {...}
implementation {
    components TMallocC;
    AplikaceC.TMalloc -> TMallocC->TMalloc;
}
```

Príklad 4.1: Vzorová konfigurace pro modul **TMalloc**

```
configuration AplikaceAppC {...}
implementation {
    components TMallocFlashC;
    AplikaceC.TMallocFlash -> TMallocC->TMallocFlash;
}
```

Príklad 4.2: Vzorová konfigurace pro modul **TMallocFlash**

Ďalej je treba priložiť k aplikácii (alebo vytvoriť) súbor `volumes<chipname>.xml` s nastavením rozdelenia externej pamäti na bloky, ktorý by mal obsahovať špecifikáciu bloku FLASHBLOK a to v následujúcom formáte:

```
<volume_table>
    <volume size="61439" name="FLASHBLOCK" />
</volume_table>
```

Príklad 4.3: Vzorový obsah súboru volumes<chipname>.xml

5 Porovnanie riešení

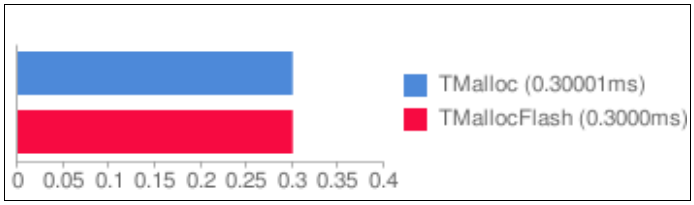
Už z princípu fungovania modulov sa dá povedať, že je jasné, že medzi modulmi budú podstatné výkonnostné rozdiely, hlavne z hľadiska časovej náročnosti a využitia procesorového času. Pre použitie modulov v aplikáciách je ale potrebné určiť jak veľké tieto výkonnostné rozdiely sú a v akých situáciách výkonnosť nadmerne klesá alebo naopak kedy modul pracuje optimálne. Pokiaľ dokážeme určiť problématické situácie, dokážeme zvoliť postupy pre optimalizáciu použitia a navrhnúť tipy, pomocou ktorých by sme dokázali moduly lepšie využiť.

5.1 Testovacia aplikácia

Pre testovanie sme navrhli niekoľko testovacích miniaplikácií. Ide o jednoduché programy založené prevažne na mnohonásobne opakovanej operácii využívajúcej špecifickú skupinu funkcií rozhrania alokátoru. Cieľom je zistiť ako sa ktoré zo súčastí alokátoru chovajú pri stresovom využití.

Test č. 1:

Aplikácia testuje úplne základnú schopnosť modulu alokovať a uvoľňovať blok pamäte o konštantnej veľkosti. Aplikácia vykoná 30 000 opakovaných cyklov, kde v každom cykle požiada o alokovanie bloku o veľkosti 2B a následne zažiada o jeho uvoľnenie [obr. 5.1].

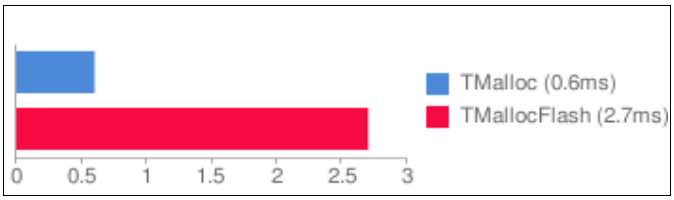


Obr. 5.1 Porovnanie výsledkov testu 1 (ms)

Vidíme, že v tomto prípade sa výsledky vôbec nelíšia. Algoritmus, ktorý obsluhuje testované operácie je vo svojej podstate úplne rovnaký u oboch implementácií a neprebieha žiadna komunikácia s externou pamäťou.

Test č. 2:

V tomto teste pridáme v každom cykle k alokácii ešte ukladanie dát do pamäte. Data sú načítané z virtuálneho senzoru generujúceho náhodné hodnoty. Vygenerované hodnoty sú 16b čísla. Opäť prevedieme 30 000 opakovaní [obr. 5.2].



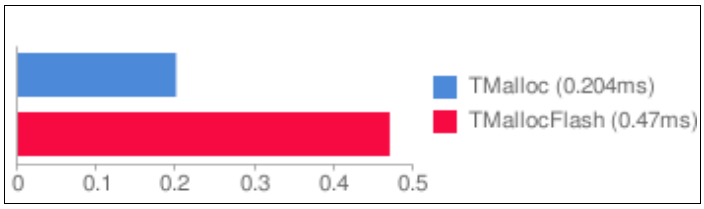
Obr. 5.2 Porovnanie výsledkov testu 2 (ms)

Tentokrát už vidíme rozdiel medzi jednotlivými modulmi. V tomto teste sa už prejavuje vplyv split-phase funkcií v module TMallocFlash.

Test č. 3:

Tentokrát zmeníme postup a uvoľňovanie pamäte necháme až na záver behu programu. Najprv vykonáme 80 cyklov pričom v každom prevedieme alokáciu 46B do ktorých následne načítame 23 náhodných čísel [obr. 5.3]. Zvolili sme veľkosť 46B pretože chceme zaplniť súvisle priestor na

hromade a po pridáni 1B na začiatok bloku kam sa ukladá veľkosť bloku nám zostane jeden voľný bajt do 48B, čo je veľkosť troch alokačných jednotiek.

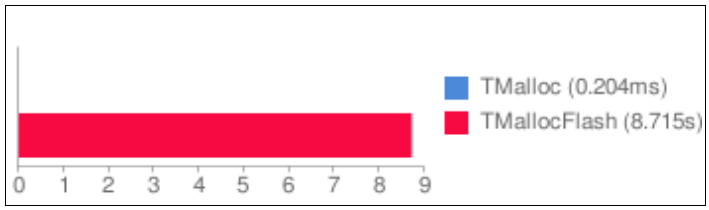


Obr. 5.3 Porovnanie výsledkov testu 3 (ms)

Výsledok je dosť podobný predošlému testu. Podstatný fakt ale je, že doposiaľ sme pracovali iba s oblasťou hromady uloženou na RAM. Preto sa zatiaľ u modulu TMallocFlash neprejavuje omeškanie spôsobené komunikáciou s externou pamäťou.

Test č. 4:

V tomto teste skúsime porovnať vplyv omeškania pri komunikácii s externou pamäťou. Zopakujeme teda test 3 ale pred tým naplníme celú časť hromady umiestnenú na pamäti RAM aby sme prinútili modul zapisovať data na externú pamäť. Data načítané z virtuálneho senzoru teda budú zapisované na flash a každý zápis vyžaduje niekoľko volaní na rozhranie BlockWrite [obr. 5.4].



Obr. 5.4 Porovnanie výsledkov testu 4 (s)

Ako vidíme, rozdiel je obrovský. Omeškanie spôsobené niekoľkonásobnou komunikáciou s flash v každom cykle je markantné. Zároveň si ale musíme uvedomiť, že sa jedná o testovanie za pomoci virtuálneho modulu na testovacom prostredí a vrámci testovacieho modulu máme možnosť nastaviť najmenšiu hodnotu omeškania pri každom dotaze na externú pamäť na 1ms, čo pri tisícoch opakovaní spôsobí ohromné omeškanie. Na reálnom zariadení by omeškanie určite nebolo tak veľké, no stále by bolo oveľa väčšie než u modulu TMalloc.

5.2 Zhodnotenie riešení

Z testovania môžme pozorovať, že obidva moduly sa po určitú dobu chovajú veľmi podobne a časová náročnosť je priamo úmerná počtu vykonávaných operácií. Ovšem po zaplnení hromady vyčlenenej na RAM dochádza k značnému spomaleniu u modulu TMallocFlash. K tomu je treba pripočítať aj fakt, že časť hromady vyčlenenej na RAM je o niečo väčšia u modulu TMalloc pretože toto riešenie nevyžaduje až také množstvo metadat (iba cca 40% v porovnaní z TMallocFlash).

Je preto potrebné pri návrhu aplikácie, ktorá by mala využívať niektorý z týchto modulov, naozaj zhodnotiť potreby aplikácie na pamäťový priestor a porovnať, či nebude v danom prípade výhodnejšie hlbšie prepracovať uvoľňovanie pamäte a použiť modul TMalloc, než automaticky použiť modul rozširujúci hromadu na externú pamäť.

6 Záver

Cieľom mojej bakalárskej práce bolo navrhnúť a implementovať dva riešenia dynamickej správy pamäte pre operačný systém TinyOS, pričom jedno z riešení malo rozširovať priestor pre alokáciu aj na externú pamäť v podobe pamäte typu flash. Moduly boli vyvíjané s cieľom následného využitia v aplikáciách, na ktorých pracujú členovia Ústavu inteligentních systémů na fakulte informačných technológií.

Pri implementácii som sa nechal inšpirovať modulom TinyAlloc, ktorý už problematiku dynamickej alokácie pre TinyOS istým spôsobom rieši. Modul má ale niekoľko nedostatkov. Predovšetkým nelogické využívanie split-phase funkcií a prácu s dvojitými ukazateľmi do pamäte. Pokúsili sme sa teda navrhnúť alternatívne riešenie, ktoré tieto ale aj iné postupy rieši trocha iným prístupom. Ponúkli sme programátorom jednoduchšie rozhranie, ktoré pracuje s ukazateľmi priamo do pamäte, aj keď sme pri tom museli obetovať možnosť komprimácie dát. Tú sme ale následne využili u druhého modulu, ktorý už túto vlastnosť umožňoval.

Implementované moduly, obzvlášť riešenie s rozšírením hromady na externú pamäť si určite nájde uplatnenie vo fakultou vyvíjaných aplikáciách. Modul samotný by sa mal ideálne v budúcnosti ďalej rozvíjať. Medzi návrhy na rozšírenie by som zaradil možnosť špecifikácie veľkosti alokačnej jednotky pri kompilácii (aktuálne pevne stanovená na 16B), aby bolo možné využiť väčší priestor na externej pamäti. Do budúcna bude vhodné lepšie prepracovať logiku odkladania blokov na flash a predovšetkým ich následné obnovovanie na internú pamäť systému. Taktiež ešte vidím možnosť na optimalizáciu metadat u modulu TMallocFlash, ktoré momentálne zaberajú pomerne veľa priestoru na internej pamäti [4.2.3].

Celkovo však môžem zhodnotiť, že sa mi požadovaného cieľa podarilo dosiahnuť a že som pripravil modul systému, ktorý ušetrí mnoho programátorskej práce a pomôže posunúť vývoj aplikácii pre túto zaujímavú platformu o nejaký krok vpred.

Literatúra

[1] *Java memory management* [online]. URL: <http://www.javaworld.com/javaworld/javaqa/1999-08/04-qa-leaks.html> [cit. 1999-08-27]

[2] *Bezdrátové senzorové sítě nové generace* [online]. URL: <http://pandatron.cz/?1280&bezdratove_senzorove_site_nove_generace> [cit. 2010-02-26]

[3] HARROP, Peter - DAS, Raghu. *Wireless sensor networks 2009 - 2019* [online]. URL: <http://www.idtechex.com/research/reports/wireless_sensor_networks_2009_2019_000212.asp> [cit. 2010-04-14]

[4] *TinyOS 2.0.2 Documentation* [online], URL <http://www.tinyos.net/tinyos-2.x/doc/> [cit. 2007-07-30]

[5] LEVIS, P. *TinyOS programming*. Cambridge University Press; 1. edition, 2009. Dokument dostupný na URL <http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf> [cit. 2010-04-15]

[6] LEWIS, F. L. Wireless Sensor Networks. In *Smart Environments: Technologies, Protocols, and Applications*, New York : John Wiley, 2004. Kapitola 2, s. 13-47.

[7] LEVIS, P. - LEE, N. *TOSSIM: A Simulator for TinyOS Networks* [online]. URL <http://www.eecs.berkeley.edu/~pal/pubs/nido.pdf>, Ver. 1.1, [cit. 2003-09-17]

[8] BREWER, E. - GAY, D. - LEVIS, P. - von BEHREN, R. *NesC Overview (and Summary of Changes*) [online]. URL <http://webs.cs.berkeley.edu/retreat-6-02/NesC-v0.6.pdf>, Ver 0.6, [cit. 2002-06-17]

[9] GAY, D. - LEVIS, P. aj. The nesC Language: A Holistic Approach to Networked Embedded Systems . *ACM SIGPLAN notices* [online]. URL: <http://www.eecs.berkeley.edu/~pal/pubs/nesc-pldi03.pdf> [cit. 2003-06-11], Intel Research Technical Report IRB-TR-02-019, ISSN 1523-2867.

[10] KERNIGHAN, B. W. - RITCHIE, D. M. *The C Programming Language.* 2. vydání, ISBN 0-13-110370-9, 272s, Prentice-Hall PTR 1988. Dostupný na URL: <http://cgip.inf.unideb.hu/eng/rtornai/Kernighan_Ritchie_Language_C.pdf> [cit. 2010-04-18]

[11] *The Memory Management Reference* [online], URL: <http://www.memorymanagement.org>, Rvenbrook Limited, 2001, [cit. 2010-04-14].

[14] WIER, C. - NOBLE, J. *Small Memory Software: Patterns for systems with limited memory* [online]. URL: <http://www.smallmemory.com/>, [cit. 2010-05-10]

[13] *Operating systems (CS/CPE 408)* [online], URL: <http://www1bpt.bridgeport.edu/sed/projects/cs503/Spring_2001/kode/os/> [cit. 2010-04-14]

[14] KNUTH, D. *Fundamental Algorithms.* 3. vydání, Addison-Wesley, 1997, ISBN 0-201-89683-4. Section 2.5: Dynamic Storage Allocation.

Zoznam príloh

Prílohy

A: Slovník pojmov a skratiek

[sl1] **MEMS** - (microelektormechanical systems) Sú to miniatúrne zariadenia kombinujúce elektrické a mechanické komponenty [6]. V senzorových zariadeniach sa využívajú na výrobu elektrickej energie potrebnej pre fungovanie zariadenia.

[sl2] **alokátor** - systém pre správu dynamicky alokovanej pamäte, ktorý umožňuje iba manuálnu prácu s dynamickou pamäťou, zvyčajne za pomoci funkcií malloc, realloc, free.

[sl3] **alokáčná jednotka** - ide o najmenší alokovateľný blok na hromade. V implementovaných moduloch má veľkosť 16B. Každý alokovaný blok má interne veľkosť vyjadrenú v násobkoch alokačnej jednotky.

[sl4] **metadata** - dáta ktoré popisujú iné dáta.

B: Technická špecifikácia platformy MICAz

Processor/Radio Board	MPR2400CA	Remarks
Processor Performance		
Program Flash Memory	128K bytes	
Measurement (Serial) Flash	512K bytes	> 100,000 Measurements
Configuration EEPROM	4K bytes	
Serial Communications	UART	0-3V transmission levels
Analog to Digital Converter	10 bit ADC	8 channel, 0-3V input
Other Interfaces	Digital I/O,I2C,SPI	
Current Draw	8 mA	Active mode
	< 15 µA	Sleep mode
RF Transceiver		
Frequency band[1]	2400 MHz to 2483.5 MHz	ISM band, programmable in 1 MHz steps
Transmit (TX) data rate	250 kbps	
RF power	-24 dBm to 0 dBm	
Receive Sensitivity	-90 dBm (min), -94 dBm (typ)	
Adjacent channel rejection	47 dB	+ 5 MHz channel spacing
	38 dB	- 5 MHz channel spacing
Outdoor Range	75 m to 100 m	1/2 wave dipole antenna, LOS
Indoor Range	20 m to 30 m	1/2 wave dipole antenna
Current Draw	19.7 mA	Receive mode
	11 mA	TX, -10 dBm
	14 mA	TX, -5 dBm
	17.4 mA	TX, 0 dBm
	20 µA	Idle mode, voltage regular on
	1 µA	Sleep mode, voltage regulator off
Electromechanical		
Battery	2X AA batteries	Attached pack
External Power	2.7 V - 3.3 V	Molex connector provided
User Interface	3 LEDs	Red, green and yellow
Size (in)	2.25 x 1.25 x 0.25	Excluding battery pack
(mm)	58 x 32 x 7	Excluding battery pack
Weight (oz)	0.7	Excluding batteries
(grams)	18	Excluding batteries
Expansion Connector	51-pin	All major I/O signals

Notes

[1] 5 MHz steps for compliance with IEEE 802.15.4/D18-2003.

Specifications subject to change without notice

C: Programová dokumentácia

Popíšeme spustenie testovacích aplikácií.

Obsah CD

Priložené CD obsahuje elektronickú verziu písomnej správy a zdrojové kódy modulov a testovacích aplikácií. Štruktúra súborov na CD je rozdelená následne:

`/Bakalarska_prace/Pisemna_zprava.pdf`

`/Bakalarska_prace/Modul/tmalloc/` – zdrojové kódy modulu

`/Bakalarska_prace/Modul/aplikace/` – zdrojové kódy testovacích aplikácií

Spustenie test. aplikácií

Aplikácie sú na CD uložné spolu s Makefile, ktorým je možné aplikácie skompilovať. Postup pre spustenie aplikácie v testovacom prostredí je nasledujúci:

- skopírujeme aplikáciu do adresára tinyos-2.1.0/apps/
- modul skopírujeme doadresára tinyos-2.1.0/tos/lib/
- presunieme a do adresára s aplikáciou
- skompilujeme aplikáciu za pomoci príkazu >make micaz sim
- spustíme skript test.py za pomoci príkazu >./test.py
- aplikáciu po dobehnutí explicitne ukončíme pomocou klávesovej skratky Ctrl+C

Návod predpokladá, že máme pripravené testovacie prostredie TinyOS + TOSSIM na platforme Linux.