Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-13428-48035

Bc. Martin Vojtko

MODULÁRNY OPERAČNÝ SYSTÉM PRE VNORENÉ SYSTÉMY

Diplomová práca

Študijný program:	Počítačové a komunikačné systémy a siete
Študijný odbor:	9.2.4 Počítačové inžinierstvo
Miesto vypracovania:	Ústav počítačových systémov a sietí, FIIT STU v Bratislave
Vedúci práce:	doc. Ing. Tibor Krajčovič PhD.

máj 2012

Zadanie diplomovej práce

Meno študenta: **Bc. Martin Vojtko**
Študijný program: Počítačové a komunikačné systémy a siete
Študijný odbor: Počítačové inžinierstvo

Názov práce: **Modulárny operačný systém pre vnorené systémy**

Samostatnou výskumnou a vývojovou činnosťou v rámci predmetov Diplomový projekt I, II, III vypracujte diplomovú prácu na tému, vyjadrenú vyššie uvedeným názvom tak, aby ste dosiahli tieto ciele:
Všeobecný cieľ:
Vypracovaním diplomovej práce preukážte, ako ste si osvojili metódy a postupy riešenia relatívne rozsiahlych projektov, schopnosť samostatne a tvorivo riešiť zložité úlohy aj výskumného charakteru v súlade so súčasnými metódami a postupmi študovaného odboru využívanými v príslušnej oblasti a schopnosť samostatne, tvorivo a kriticky pristupovať k analýze možných riešení a k tvorbe modelov.
Špecifický cieľ:
Vytvorte riešenie, zodpovedúce návrhu textu zadania, ktorý je prílohou tohto zadania. Návrh bližšie opisuje tému vyjadrenú názvom. Tento opis je záväzný, má však rámcový charakter, aby vznikol dostatočný priestor pre Vašu tvorivosť.
Riaďte sa pokynmi Vášho vedúceho.
Pokiaľ v priebehu riešenia, opierajúc sa o hlbšie poznanie súčasného stavu v príslušnej oblasti alebo o priebežné výsledky Vášho riešenia alebo o iné závažné skutočnosti, dospejete spoločne s Vaším vedúcim k presvedčeniu, že niečo v texte zadania a/alebo v názve by sa malo zmeniť, navrhnite zmenu. Zmena je spravidla možná len pri dosiahnutí kontrolného bodu.

Miesto vypracovania: Ústav počítačových systémov a sietí, FIIT STU v Bratislave
Vedúci práce: **doc. Ing. Tibor Krajčovič, PhD.**

Termíny odovzdania:
podľa harmonogramu štúdia platného pre semester, v ktorom máte príslušný predmet (Diplomový projekt I, II, III) absolvovať podľa Vášho študijného plánu
Predmety odovzdania:
V každom predmete dokument podľa pokinov na www.fiit.stuba.sk v časti: home> Informácie o> štúdiu> organizácia štúdia> diplomový projekt

V Bratislave dňa 21. februára 2011

doc. Ing. Pavel Čičák, PhD.
poverený vedením ústavu

Návrh zadania diplomovej práce

Finálna verzia[1]

Študent:

Meno, Priezvisko, tituly:	Martin Vojtko, Bc.
Študijný program:	Počítačové a komunikačné systémy a siete
Kontakt:	martin.vojtkom@gmail.com

Vedúci projektu:

Meno, Priezvisko, tituly:	doc. Ing. Tibor Krajčovič, PhD.
Pracovisko, adresa:	Ústav počítačových systémov a sietí FIIT STU, Ilkovičova 3, 842 16 Bratislava 4
Kontakt:	tkraj@fiit.stuba.sk

Projekt:

Názov:	Modulárny operačný systém pre vnorené systémy
Miesto vypracovania:	Ústav počítačových systémov a sietí FIIT STU, Bratislava
Oblasť problematiky:[2]	Operačné systémy pre vnorené systémy

Text zadania:

Navrhnite a implementujte vybrané časti modulárneho operačného systému pre vnorené systémy, ktorý bude spravovať nad ním spustené procesy.
Predmetom analýzy bude spracovanie prístupov jednotlivých častí jadra modulárneho operačného systému a ich použitia v praxi, analýza použitia štatistických meraní výkonnosti modulov. Analýza súčasných operačných systémov pre vnorené systémy.
Predmetom návrhu bude návrh vybraných častí modulárneho operačného systému, ktoré boli predmetom analýzy a návrh štatistického modulu.
Predmetom implementácie bude modulárny operačný systém pre vnorený systém a testovacia aplikácia pre hostiteľský počítač. Jednotlivé časti operačného systému implementujte viacerými prístupmi, tak aby tvorili moduly. Operačný systém bude modulárne rozšíriteľný o štatistické spracovanie, tak aby sa dali analyzovať jednotlivé kombinácie modulov z hľadísk réžie jadra operačného systému, dodržania limitov procesov reálneho času, férovosti spúšťania procesov, komunikácie medzi procesmi. Porovnajte modulárny operačný systém s inými operačnými systémami. Na otestovanie implementovaného modulárneho operačného systému použite vhodnú vývojovú dosku.

150-200 slov, ktoré opisujú problém v kontexte súčasného stavu vrátane motivácie a smerov riešenia

[1] Veľkosť jednotlivých polí pre vypĺňanie nemožno meniť. Návrh zadania vytlačiť obojstranne na jeden list papiera.

[2] Identifikácia oblasti v rámci odboru štúdia, na ktorú sa projekt primárne viaže

Literatúra:

- Tanenbaum, A.S.: Modern Operating systems 2nd edition, Prentice Hall, 2001. ISBN 0-13-031358-0
- Tanenbaum, A.S and Woodhull, A.S.: Operating systems, Design and Implementation 3nd edition, PrenticeHall, 2006. ISBN 0-13-142938-8
- Štefanovič, J.: Základy operačných systémov, STU, 2007. ISBN 978-80-227-2586-6

2-3 vedecké zdroje, každý na samostatnom riadku a vo formáte zodpovedajúcom bibliografickým odkazom podľa normy STN ISO 690, ktoré sa viažu k téme zadania a preukazujú výskumnú povahu problému a jeho aktuálnosť (uveďte všetky potrebné údaje na identifikáciu zdroja, pričom uprednostnite vedecké príspevky v časopisoch a medzinárodných konferenciách)

Vyššie je uvedený návrh diplomového projektu, ktorý vypracoval Bc. Martin Vojtko, konzultoval a osvojil si ho doc. Ing. Tibor Krajčovič, PhD. a súhlasí, že bude takýto projekt viesť v prípade, že bude pridelený tomuto študentovi.

V Bratislave dňa 18.11.2010

Podpis študenta

Podpis vedúceho projektu

Vyjadrenie garanta predmetu Diplomový projekt I, II, III

Návrh zadania schválený: áno / ~~nie~~[3]

Dňa: 21.2.2011

Podpis garanta

[3]Nehodiace sa prečiarknite

ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Počítačové a Komunikačné Systémy a Siete

Autor: Bc. Martin Vojtko
Diplomová práca: Modulárny operačný systém pre vnorené systémy
Vedúci diplomovej práce: doc. Ing. Tibor Krajčovič PhD.
máj 2012

Predmetom tohto projektu bol návrh a implementácia vybraných konceptov operačných systémov pre vnorené systémy. V práci sme analyzovali jednotlivé koncepty operačných systémov a uviedli sme výber troch operačných systémov z danej oblasti, ktoré sme považovali za najpodobnejšie našej predstave o vnorenom operačnom systéme.

Na základe analýzy sme špecifikovali požiadavky na nový operačný systém, ktorý sme nazvali *Modulárny operačný systém*. Súčasťou špecifikácie sa stal aj hrubý návrh jadra a periférie operačného systému.

Vychádzajúc zo špecifikácie sme navrhli jednotlivé moduly OS. V návrhu sme sa venovali správe pamäti, riadeniu procesov, plánovaniu procesov, komunikácií procesov, riadeniu prístupu, riadeniu spotreby energie, riadeniu vstupov a výstupov, štatistike a periférii OS.

V implementačnej fáze projektu sem vybrali niektoré časti operačného systému, ktoré sme implementovali. Výsledkom je fungujúce jadro OS pozostávajúce zo Správcu pamäte, Správcu procesov, Plánovania procesov, Riadenia prístupu a štatistiky. Modul plánovania procesov sme implementovali viacerými prístupmi. Silnými stránkami implementovaného OS sa stali modulárnosť, konfigurovateľnosť a prenositeľnosť.

Na zbieranie štatistických dát sme implementovali aplikáciu, ktorá zbiera, spracuje a zobrazuje informácie odosielané štatistickým modulom cez sériové rozhranie.

Operačný systém sme otestovali pomocou jednoduchej programovej štruktúry. Výsledkom testovania je aj porovnanie niektorých konfigurácií operačného systému a porovnanie s analyzovaným operačným systémom.

ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Computer and Communication Systems and Networks

Author: Bc. Martin Vojtko
Master's Thesis: Modular Embedded Operating System
Supervisor: doc. Ing. Tibor Krajčovič PhD.
2012, May

The objective of this project is a proposal and an implementation of parts of an embedded operating system. Part of project was analysis of the operating systems concepts and an analysis of chosen embedded operating systems.

In the work we specified requirements, based on the analysis, to new OS that we call *The Modular Operating System*. In the specification is rough proposal, where we specifies a kernel and peripheries.

From the specification we proposed modules of the OS. We proposed Memory Management Module, Task Management Module, Scheduler, Inter-process Communication Module, Access Control Module, Power Management Module, statistical module, I/O Management Module and peripheries of the OS.

In the implementation phase we chose some parts of the OS from the proposal that we implemented. Results of the implementation are Memory Management Module, Task Management Module, Scheduler, Inter-process Communication Module, Access Control Module, statistical module. We implemented the Scheduler by several approaches. Strengths of new OS are modularity, portability and configurability.

We implemented an application witch catches, processes and displays information sent by the OS through serial interface.

The operating system was tested in several configurations by simple program structure. Result of the tests is comparison between several configurations of the OS.

Obsah

1 Úvod

Operačné systémy asi najlepšie poznáme z oblasti personálnych počítačov. Každý personálny počítač má nainštalovaný nejaký OS. Pravdaže personálne počítače nie sú jedinou doménou operačných systémov. Operačné systémy sa používajú aj v obrovských sálových počítačoch, v serveroch alebo aj v maličkých platobných kartách a v neposlednom rade vo vnorených systémoch. Práve vnorené operačné systémy sú predmetom tejto práce a jej cieľom je navrhnúť a implementovať nový vnorený operačný systém s dôrazom na jeho modulárnosť. Vnorené operačné systémy sú v súčastnosti na vzostupe a stávajú sa zaujímavou oblasťou vývoja. V tejto súvislosti sa možno pýtať: "Prečo by mali také operačné systémy pracovať aj vo vnorenom systéme? Však to s ohľadom na zložitosť hardvéru vnoreného systému nie je potrebné!". Dôvodom prečo zaviesť operačný systém je napríklad zvýšenie úrovne abstrakcie návrhu programov, umožnenie paralelného behu viacerých procesov a najmä minimalizovanie spotreby energie. Nie je vždy pravda, že vnorený systém je jednoduché zariadenie, niektoré systémy dosahujú ba aj presahujú výkonové kapacity personálnych počítačov a v takýchto systémoch sa operačný systém určite oplatí. Na trhu existuje množstvo vnorených operačných systémov či už komerčných alebo voľne šíriteľných. Preto sa čitateľ môže pýtať: "Prečo implementovať ďalší?". Dôvod prečo implementovať nový operačný systém spočíva v možnosti kúskom prispieť k zlepšeniu poznania novým pohľadom a v poznaní a skúsenostiach, ktoré budú nadobudnuté počas priebehu projektu. Ďalšia otázka, ktorá si vyžaduje odpoveď vyplýva už z názvu práce "Prečo modulárny?". Odpoveď je jednoduchá, operačné systémy nie sú navrhované s dôrazom na modulárnosť, a preto v prípade nasadenia na rôzne výkonné systémy a architektúry, vznikajú problémy pri nasadení. Modulárny operačný systém bude navrhnutý tak, aby sa dal ušiť na mieru širokému spektru architektúr.

Dokument je členený na šesť častí. V prvej časti, analýze, sa venuje rozboru súčastného stavu v oblasti. V analýze sú operačné systémy rozobrané vo všeobecnosti a osobitne najmä pre vnorené operačné systémy. V druhej časti, špecifikácii, je upriamený pohľad na vymedzenie vlastností, ktoré bude modulárny operačný systém spĺňať. Zároveň je v tejto časti uvedený aj hrubý návrh architektúry operačného systému. Tretia časť je venovaná návrhu Modulárneho operačného systému. V tejto časti sa na základe analýzy a špecifikácie upresňuje finálna podoba MOS. Štvrtá časť, implementácia, opisuje spôsob realizácie vybraných častí MOS. Piata časť práce dokumentuje spôsob akým bol MOS testovaný. Posledná časť finálne zhodnocuje prácu vykonanú počas celého obdobia a uvádza nápady na ďalší rozvoj MOS.

1.1 Pojmy a skratky

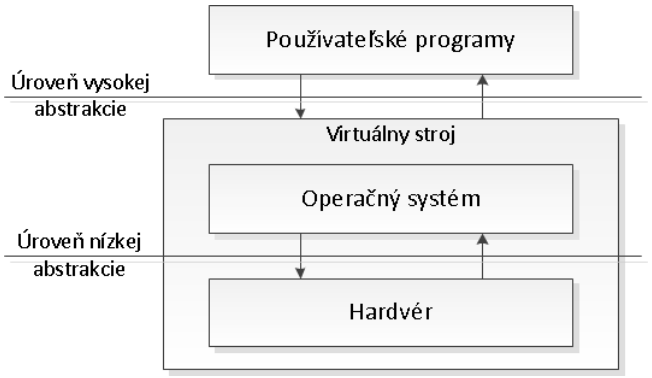
- **Blokovaná úloha (Blocked task)** - Je úloha, ktorá pri pokuse o vstup do kritickej oblasti je na určený časový úsek neplánovaná. Po uplynutí časového úseku je znovu plánovaná.
- **Pozastavená úloha (Suspend task)** - Je úloha, ktorá pri pokuse o získanie respektíve odoslanie dát bola neúspešná z dôvodu, že odosielateľ respektíve prijímateľ nie je pripravený. Úloha nie je plánovaná kým dáta nie sú pripravené.
- **Job** - V oblasti mainframe operačných systémov je job úloha alebo skupina úloh, ktoré sa vykonávajú na procesore v čase výpočtového útlmu úloh reálneho času. V kontexte MOS je to úloha, ktorá nebola reálne vytvorená a plánovaná (len odložená) z dôvodu nedostatku výpočtových alebo pamäťových kapacít.
- **Kontrolný blok úlohy TCB** - (Task control Block) Záznam v tabuľke spustených procesov. Často označovaný aj ako **Hlavička úlohy**.
- **Kontrolný blok programu PCB** -(Program control Block) Záznam v tabuľke programov. Často označovaný aj ako **Hlavička programu**
- **Modulárny operačný systém (MOS)** - Vnorený operačný systém navrhnutý s dôrazom na modulárnosť.
- **Obálka volania** - Platformovo závislá funkcia zabezpečujúca bezpečné a chránené prepnutie procesora do chráneného režimu a spustenie volania, ktoré obaľuje.
- **Orientovaný sled** - Postupnosť vrcholov a hrán patriacich grafu G. Vrcholy sa v slede striedajú s hranami. Ak medzi dvoma vrcholmi v slede je hrana, tak táto hrana je incidentná k obom vrcholom a platí, ak je vrchol v slede pred hranou tak hrana z vrcholu vychádza a do vrcholu, ktorý je za ňou, vchádza.
- **Procesný graf** - Graf reprezentujúci stav procesov systému v čase, ktorého vrcholy sú reprezentujú procesy a hrany reprezentujú vzťahy medzi procesmi.
- **Programový graf** - Graf reprezentujúci systém z hľadiska všetkých možných vzťahov medzi procesmi programov.
- **Úloha/proces reálneho času, RT úloha** - (real-time task) Úloha, ktorej dôležitým faktorom správneho vykonávania je dodržanie časových limitov vykonávaných operácií.
- **Vnorený operačný systém (VOS)** - Operačný systém navrhnutý špeciálne pre prácu na platformách vnorených systémov.

2 Analýza

Vyvinutie operačných systémov bolo prirodzenou reakciou na neustále narastajúcu zložitosť výpočtových systémov. Kód navrhnutý na riadenie vstupno/výstupných zariadení sa v minulosti neustále recykloval a vylepšoval. Takýto kód sa sústreďoval do procedúr a knižníc, ktoré sa linkovali k hlavnému programu, aby sa zvýšila úroveň abstrakcie, a tým aj zjednodušilo programovanie zariadení. Výkonnosť hardvéru postupne rástla a prišla potreba jeho efektívnejšieho využitia. Jeden proces už nestačil efektívne pokryť výpočtové kapacity zariadení. Viacej procesov na výpočtovom systéme podmienilo vznik riadenia prístupu. Ak proces nepotreboval momentálne pracovať, tak uvoľnil priestor inému. Prirodzeným krokom vývoja bolo vytvorenie samostatného procesu, ktorý prevzal riadenie prístupu ostatných procesov k zariadeniu, prvý operačný systém.

2.1 OS ako rozšírenie hardvéru a riadenie zdrojov

Operačný systém rozširuje a riadi pridružený hardvér. Rozšírenie hardvéru je potrebné vnímať ako zvýšenie abstrakcie prístupu k hardvéru, nie ako pridanie novej funkcionality. Na nízkej úrovni abstrakcie je programátor prinútený zaoberať sa zložitým časovaním zariadenia, registrami a riadiacimi signálmi. Operačný systém túto zložitosť zapúzdruje do formy volaní procedúr s jednoduchým komunikačným rozhraním. To zložité obstará operačný systém. Operačný systém tvorí spolu s hardvérom virtuálny stroj (obr. 2.1). [1, 2]



Obr. 2.1: Operačný systém ako rozšírenie hardvéru

Riadenie zdrojov je dôležitou úlohou operačného systému. Hardvérové zariadenia sú komplexné jednotky obsahujúce pamäť, zbernice, V/V rozhrania atď. Úlohou OS je poznať tieto zariadenia, poznať ich aktuálny stav a riadiť ich. Zariadenia sú prideľované jednotlivým procesom podľa určených pravidiel časového a priestorového multiplexu, pretože

jedno zariadenie môže byť pridelené v určenom časovom úseku najviac jednému procesu (časový multiplex). Pravidlá priestorového multiplexu sa zavádzajú v prípade, že v systéme existuje viacej zariadení rovnakého typu, ku ktorým chcú procesy pristúpiť. [1]

Podmienkou časového multiplexu je, že v pridelenom časovom úseku smie byť zariadenie pridelené práve jednému procesu. Po vypršaní času sa zariadenie odoberie procesu a pridelí sa inému. Ak pridelené časové úseky sú tak malé, že nie sú postrehnuteľné ľudským vnímaním vzniká dojem, že v jednom okamihu sa paralelne vykonáva viacero procesov. Takýto paralelizmus sa nazýva nepravý. Pravý paralelizmus je naopak skutočné paralelné vykonávanie viacerých procesov na viacerých prostriedkoch. [3]

2.2 Koncepty OS

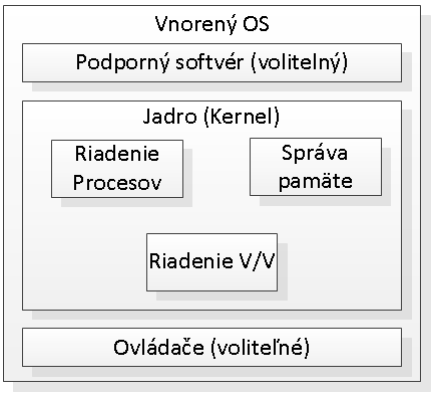
Najzákladnejšie koncepty v oblasti návrhu a implementácie OS sú v určitej miere vlastné každému operačnému systému, nezáleží či je OS navrhovaný pre personálne počítače alebo vnorené aplikácie. Najzákladnejšími konceptami, ktoré sú vlastné skoro každému OS sú [1]:

- Proces
- Uviaznutie
- Správa pamäte
- Vstup a výstup (V/V)
- Súborový systém
- Bezpečnosť
- Interpreter

Každý operačný systém pozostáva z jadra (kernel) a pridružených vrstiev, ktoré ho rozširujú. Jadro štandardného vnoreného operačného systému má za úlohu [4]:

- Riadiť procesy
- Spravovať pamäť
- Spravovať Vstupno výstupné zariadenia

Jadro pokrýva štyri spomínané koncepty a to procesy, uviaznutia, správu pamäte a V/V. Ostatné koncepty sú zahrnuté do iných vrstiev operačného systému. Niektoré jadrá vnorených operačných systémov bývajú minimalizované do takej miery, že sa zaoberajú len správou procesov a uviaznutí (obr. 2.2).



Obr. 2.2: Architektúra OS

2.2.1 Proces

Objektom záujmu každého operačného systému je proces. Operačný systém spravuje procesy a stará sa, aby mal každý proces možnosť pristúpiť k hardvérovým prostriedkom.

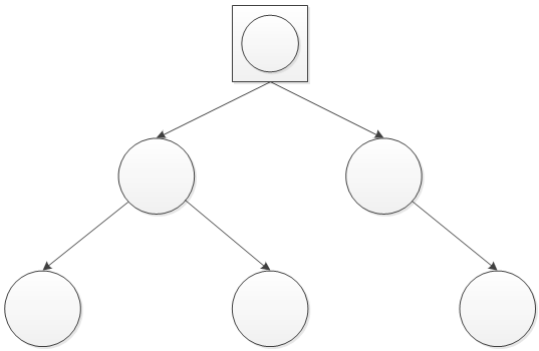
Proces je program vykonávaný na procesore. Jeden program smie byť reprezentovaný v systéme viacerými procesmi na rôznych alebo rovnakých etapách vykonania. Každý proces má pridelený priestor v pamäti, s ktorým smie pracovať. Pamäť štandardného procesu pozostáva z časti uloženej v hlavnej pamäti a časti uloženej v registroch procesora. Obsah hlavnej pamäti rozdeľujeme na kódovú časť a dátovú časť, ktorá sa delí na statickú a dynamickú. V registroch sú ukladané informácie aktuálneho stavu vykonávania procesu, (programové počítadlo, pointer na vrchol zásobníka, registre, atď.) ktorý spolu s hlavičkou procesu tvorí kontext procesu. Ak proces nie je práve vykonávaný tak jeho kontext sa nachádza v tabuľke spustených procesov nazývanej aj tabuľka spustených úloh (obr. 2.3). Jeden záznam v tabuľke sa nazýva kontrolný blok úlohy (task control block, TCB)[1]

Obr. 2.3: Organizácia pamäte procesu

Kontext procesu zachytáva aktuálny stav procesu. Počas prepnutia úlohy je kontext procesu odkladaný do tabuľky spustených úloh alebo do zásobníka daného procesu. Odloženie závisí od implementácie operačného systému ale aj procesora. Niektoré procesory majú inštrukciu riadiacu odloženie kontextu úlohy do zásobníka úlohy čo značne urýchľuje prepnutie úlohy. Po spätnom prepnutí úlohy sa proces znovu obnoví do pôvodného stavu prečítaním kontextu z TCB alebo zo zásobníka. Hlavička procesu obsahuje informácie o prístupových právach, priorite, čísle procesu, pridelenej pamäti, rodičovskom procese atď. [3]

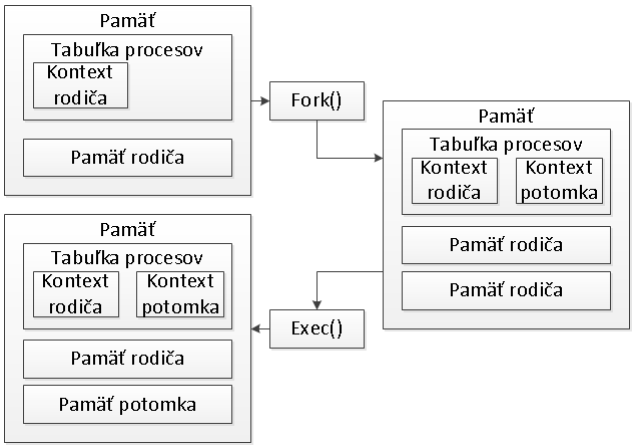
Procesy tvoria hierarchiu rodičov a detí. Takáto hierarchia sa s výhodou reprezentuje grafom. Pre potreby tejto práce bola navrhnutá konvencia založená na teórií grafov, ktorá vhodne popisuje vzťahy medzi procesmi a bude použitá pre opis procesného a programového modelu Modulárneho operačného systému z hľadiska riadenia procesov (obr. 3.2). Konvencia je podrobnejšie vysvetlená v časti Špecifikácia.

Po inicializácii jadra vnoreného operačného systému sa štandardne spustí inicializačný proces, ktorý spúšťa svojich potomkov a potomkovia svojich potomkov. Na obrázku 2.4 je príklad grafu procesov, ktoré sú spustené v systéme. Na vrchole štruktúry je inicializačný proces, ktorý spúšťa dve deti, tie spúšťajú svoje deti.



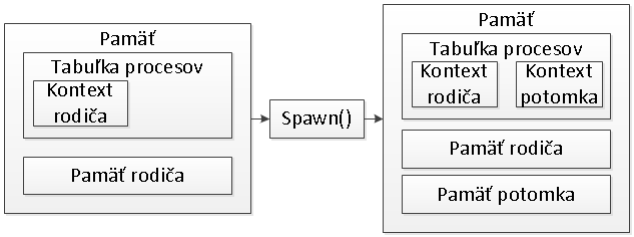
Obr. 2.4: Štruktúra procesov, predok a potomok

Na to, aby sa proces mohol vykonávať musí byť najprv vytvorený. Na vytvorenie procesu vo vnorenom systéme existujú dva modely. Model fork/exec odvodený zo štandardu IEEE/ISO POSIX 1003.1 a model spawn odvodený z fork/exec modelu. Tieto modely sú si veľmi blízke, pretože jeden je odvodený od druhého, ale existujú v nich odlišnosti. Oba modely majú spoločnú inicializačnú časť počas, ktorej sa vytvorí kontext nového procesu a alokuje sa pamäťový priestor procesu. Modely sa líšia v spôsobe alokovania pamäťového priestoru. Podľa modelu fork/exec rodičovský proces volaním fork() vytvorí kópiu svojho pamäťového priestoru pre dcérsky proces. Exec() volanie následne prepíše pôvodný obsah kópie obsahom dcérskeho procesu (obr. 2.5). Výhodou fork/exec modelu je možnosť jednoduchého vloženia inicializačných dát medzi volaniami fork() a exec(). Po volaní fork() dcérsky proces beží ešte v zdrojovom kóde rodičovského procesu. Príkladom je jednoduché vytvorenie komunikačného vzťahu medzi procesmi. [4]



Obr. 2.5: Model fork/exec

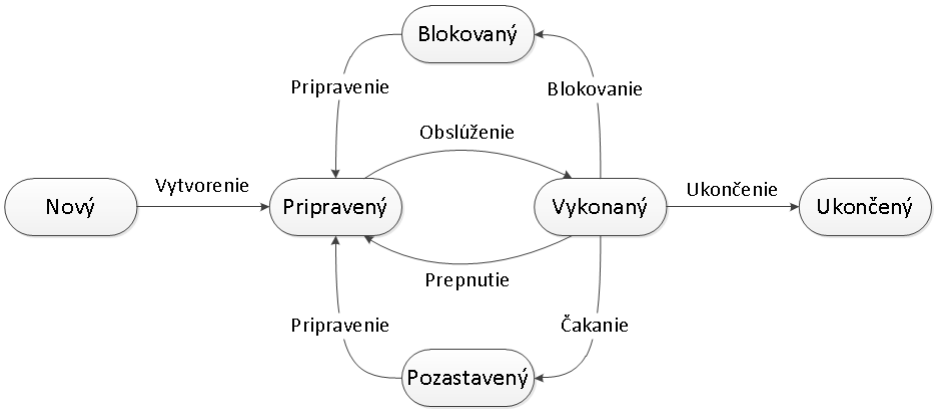
Spawn model vytvorí nový pamäťový priestor pre dcérsky proces od začiatku, čo umožní aby dcérsky proces bol okamžite pripravený na vykonávanie (obr. 2.6). Výhodou je rýchle pripravená úloha ale napríklad vytvorenie komunikačného vzťahu medzi dcérskym a rodičovským procesom je komplikovanejšie. [4]



Obr. 2.6: Model spawn

Operačný systém štandardne poskytuje pre procesy možnosti rôznych stavov spustenia. Proces môže byť *vytvorený, pripravený, vykonávaný, pozastavený, blokovaný a ukončený* (obr. 2.7). Stav *pozastavený* sa s výhodou používa v prípade, že proces čaká na dáta z externého zdroja alebo od iného procesu. V prípade, že je proces *pozastavený* nezaraďuje sa do spúšťacej rady a tým nezaberá prostriedky. Proces je zaradený do spúšťacej rady keď je zdroj pripravený. Stav *blokovaný* je na rozdiel od stavu *pozastavený* zaradený do spúšťacej rady po určenom časovom úseku, napríklad ak čaká proces na vstup do kritickej oblasti. V stave *vykonávaný* môže byť štandardne práve jeden proces. V stave *pripravený* sú procesy, ktoré majú pripravené všetky dáta a čakajú len na pridelenie procesora.

Radením pripravených procesov do poradia spúšťania sa zaoberá špeciálna časť jadra nazvaná plánovač procesov (scheduler). Podrobnejšie je plánovač vysvetlený v časti Jadro OS.



Obr. 2.7: Stavový automat procesu

2.2.2 Uviaznutie

Uviaznutie (deadlock) nastáva v prípadoch, keď procesy pristupujú k rovnakým prostriedkom ako je napríklad miesto v pamäti alebo V/V zariadenie. Tieto prostriedky procesy obsadzujú a vzájomne si bránia vo vykonaní. Uviaznutie vzniká aj pri čakaní na správu od iného procesu napríklad chybným návrhom. [3]

Na to, aby uviaznutie mohlo vzniknúť, musí splniť podmienky uviaznutia [1]:

- **Vzájomné vylúčenie** (mutual exclusion) - každý zdroj je pridelený práve jednému procesu.
- **Obsadenie a čakanie** - obsadenie jedného zdroja a čakanie na druhý.
- **Nepreempcia** - uvoľnenie prostriedku až po skončení práce s nim.
- **Čakanie v kruhu** - proces čaká na zdroj, ktorý obsadil druhý proces, ten čaká na zdroj od tretieho procesu až nakoniec posledný proces čaká na zdroj, ktorý obsadil prvý.

Problém uviaznutia sa môže odstrániť ak jedna z podmienok uviaznutia nie je splnená. V praxi je ale obtiažne dosiahnuť nesplnenie čo i len jednej podmienky. Je potrebné zaviesť možnosť odstránenia uviaznutia v prípade jeho vzniku a predchádzať uviaznutiu dodatočným zabezpečovacím mechanizmom.

Najjednoduchším a často aj najefektívnejším riešením uviaznutia je pštrosí algoritmus, ktorý predpokladá, že žiadne uviaznutie nemôže nastať. Takýto algoritmus je vinikajúci z hľadiska réžie jadra, pretože nepridáva žiaden výpočtový čas. V prípade vzniku uviaznutia sa ale systém už nedokáže zotaviť. Pštrosí algoritmus je vhodné používať len v prípade, že je štruktúra procesov navrhnutá tak, aby k uviaznutiu nemohlo dôjsť, teda nie je splnená niektorá z podmienok uviaznutia.

Odstrániť už vzniknuté uviaznutie je možné zrušením jedného z procesov, ktoré tvoria uviaznutie, na báze výberu obete s kritériom čo najmenšej straty na systéme, odložením procesu a odobraním jeho prostriedkov alebo zrušením preempcie (obsadenia) prostriedku procesu, ktorý ho obsadil.

Iným riešením je použitie metodiky predchádzania vzniku uviaznutia. Príkladom je riadený prístup k prostriedkom cez vyhradený proces. K prostriedku smie pristúpiť len vyhradený proces, s ktorým komunikujú procesy, ktoré chcú pristupovať na prostriedok.

2.2.3 Správa pamäte

Pamäť je možno najdôležitejšie zariadenie vo výpočtovom systéme. V pamäti je uložený zdrojový kód a dáta procesu. Pamäť potrebujú k svojej činnosti všetky procesy ale i operačný systém. Pamäť vo forme registrov alebo zásobníkov majú všetky V/V zariadenia a aj samotný procesor. Preto na správu pamäte je kladený veľký dôraz. Správa pamäte je potrebná z dôvodu riadenia prideľovania pamäte procesom, ale aj z dôvodu jej nedostatku. Bolo by neefektívne ale najjednoduchšie mať toľko pamäte koľko potrebujú všetky procesy spolu s operačným systémom, ale s ohľadom na plytvanie zdrojov a nemožnosť predikcie množstva potrebnej pamäte to nie je možné.

Keď píšeme o pamäti, myslíme tým hlavnú pamäť, teda pamäť, ktorá je priamo alebo prostredníctvom cache pripojená k procesoru a o registroch patriacich k zariadeniam a procesoru. Pamäť na úrovni úložísk (pevný disk) je už predmetom súborových systémov.

Rozlišujeme dve triedy správ pamäte. Metódy z prvej triedy načítavajú procesy z úložiska (pevný disk) a po ukončení činnosti odkladajú proces do úložiska (stránkovanie, swapovanie). Metódy druhej triedy majú správu pamäte jednoduchšiu.

Existujú systémy, ktoré majú v jeden časový okamih práve jeden proces v pamäti. Počas prepnutia kontextu je úlohou správy pamäte proces v pamäti odložiť a načítať nový. Operačné systémy s viacerými procesmi v hlavnej pamäti musia dbať na to, aby procesy nepresiahli svojimi pamäťovými nárokmi do priestoru iného procesu. Najvyššiu úroveň dosahujú operačné systémy s virtuálnou pamäťou. Virtuálna pamäť zabezpečuje možnosť vytvorenia procesov aj mimo hlavnej pamäte. V prípade, že sú takéto procesy potrebné sú premiestnené stránkovacím mechanizmom do hlavnej pamäte s výberom obete.

2.2.4 Správa vstupov a výstupov

Každý výpočtový systém musí mať rozhrania na interakciu so svojim okolím. Výpočtový systém zbiera dáta a prijíma príkazy, ktoré spracúva a vracia späť okoliu vo forme správ alebo príkazov. Úlohou operačného systému je spravovať komunikačné rozhrania. V prvom rade riadi prístup k týmto prostriedkom, pretože jeden prostriedok môže v časovom úseku obsadiť najviac jeden proces.

Okrem správy V/V rozhraní býva operačný systém rozšírený o ovládače, ktoré vedia pracovať s rozhraním a bývajú navrhnuté tak, že len tieto ovládače vedia pristúpiť k rozhraniu, čím sa zabraňuje vznikom uviaznutí.

2.3 Triedenie OS

Podľa veľkosti výpočtového systému delíme operačné systémy na: [1]

- **Mainframe** - navrhnuté na správu obrovského počtu V/V a vykonávanie veľkého počtu procesov naraz v jednom čase.
- **Serverové** - navrhnuté na obstarávanie veľkého počtu používateľov.
- **Multiprocesorové** - navrhnuté na efektívne rozdelenie záťaže na viacero CPU.
- **Personálne** - navrhnuté na prácu s jedným používateľom.
- **Real-Time** - navrhnuté s dôrazom na dodržanie časových limitov vykonania procesov.
- **Vnorené** - navrhnuté na aplikovanie v malých výpočtových systémoch. Často krát spojený s Real-Time požiadavkami.
- **Smart Card** - navrhnuté na riadenie v extrémne malých výpočtových systémov ako sú napríklad platobné karty alebo mobilné SIM karty.

Podľa riadenia procesov delíme operačné systémy na: [4]

- **Kooperatívne** - riadiace prepínanie procesov na základe systémových volaní umiestnených v procese. Výhodou prístupu je bezpečné ukončenie zápisu viac bajtových informácii pretože k prepnutiu dôjde až po ukončení zápisu. Čas vykonania je závislý od potrieb procesu, kontext procesu je malý lebo stačí len poznať vrchol zásobníku. Nevýhodou je nebezpečenstvo uviaznutia, zacyklenia úlohy po chybe vykonania programu. V prípade, že v procese nie je volanie, tak k prepnutiu dôjde až po ukončení vykonania procesu.
- **Preemtívne** - riadiace prepínanie procesov na základe časovo riadeného prerušenia. Proces smie byť vykonávaný, kým mu neuplynie čas, potom je vyvolané prepnutie procesov. Výhodou takejto realizácie je, že nemôže dôjsť k uviaznutiu jedného procesu z dôvodu chyby behu programu alebo z dôvodu dlhého behu výpočtu. Procesy smú byť lineárne programované. Nevýhodou je nebezpečenstvo nedokončenia zápisu alebo prenosu viac bajtových informácii, čas prepnutia nie je závislý od potrieb procesu, operačný systém nevie, ktoré premenné sú aktuálne používané, preto musia byť odložené do kontextu procesu.

- **Kombinované** - prepínanie procesov je riadené časovo riadeným prerušením aj systémovým volaním. Kombinujú sa výhody oboch metód.

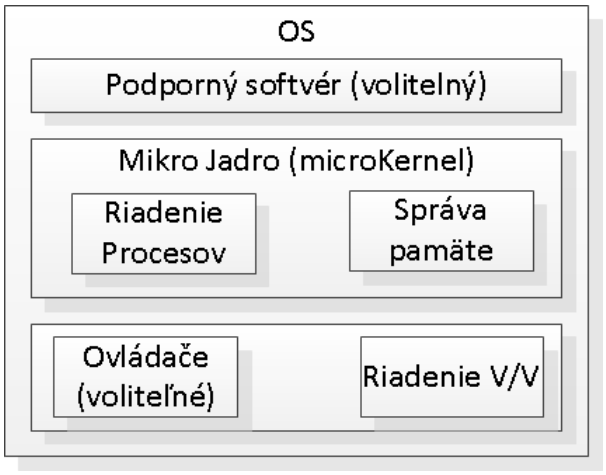
Podľa štruktúry delíme operačné systémy na: [1, 4]

- **Monolitické** - navrhnuté ako množstvo procedúr bez štruktúry. Každá procedúra vie volať akúkoľvek inú procedúru. Výhodou takejto štruktúry je malá réžia systému, pretože prepojenie zariadenia s procesom je na úrovni volania jednej procedúry. Nevýhodou tejto organizácie je veľmi obtiažne rozširovanie funkcionality alebo zlepšovanie vlastností OS. Aj malý zásah do systému spôsobí veľké zmeny.

- **Monolitické-modulárne** - sú systémy rozvrhnuté do monolitických modulov. (obr. 2.2) Koncept zjednodušuje ladenie a úpravu systému.

- **Vrstvové** - systémy majú štruktúru usporiadanú do vrstiev medzi ktorými je vytvorený systém rozhraní. Vrstva poskytuje služby vrstve nad ňou samou a žiada služby od vrstvy pod ňou samou. Výhodou takéhoto konceptu je jednoduchá úprava systému. Nevýhodou je pribúdajúca réžia na každej vrstve systému, ktorá môže vo vnorených systémoch byť kľúčová.

- **microkernel** - návrh OS s čo najjednoduchším jadrom spravujúcim len procesy a pamäť. (obr. 2.8) Podtriedou mikrokernel operačných systémov sú nanokernel systémy, ktoré majú v jadre implementované len riadenie procesov.



Obr. 2.8: Architektúra OS

2.4 Jadro operačného systému

Štandardné jadro vnoreného operačného systému má za úlohu riadiť procesy, spravovať pamäť a V/V. Okrem týchto úloh môže byť jadro rozšírené o ďalšie úlohy v závislosti od

potrieb používateľa. V tejto sekcii budú popísané len tie najzákladnejšie.

2.4.1 Riadenie procesov

Riadenie procesov zahŕňa prepínanie procesov a ich zoraďovanie podľa určitých kritérií. V súčastnosti má veľké percento procesorov hardvérové prepínanie procesov, a tým sa operačné systémy sústredia na implementovanie čo najefektívnejšieho algoritmu na zoraďovanie procesov podľa kritérií. Kritériami zoraďovania môžu byť priorita procesu, práva procesu, čas strávený na procesore, počet potomkov, atď. Podľa kritérií možno identifikovať metódy zoraďovania procesov [3]:

- **Plánovania podľa poradia (round Robin)** - radiaca procesy v poradí v akom bol uložený do tabuľky pripravených procesov. Výhodou tejto metódy je férovosť, pretože zaručuje, že každý proces bude obslúžený a nebude predbehnutý inými procesmi. V prípade existencie procesov, ktoré potrebujú byť vykonané čo najrýchlejšie, to ale môže spôsobovať problémy so splňaním časových limitov.

- **Plánovania podľa kritéria** - riadiaca procesy podľa nejakého kritéria, napríklad priority. Výhodou metódy je rýchlejšie vykonávanie dôležitejších procesov. Nevýhodou je nárast réžie potrebnej na výpočet poradia úlohy.

- **Kombinácie** - predchádzajúcich dvoch metód. Príkladom môže byť zaraďovanie úloh podľa priority do viacerých radov, ktoré sú následne radené metódou round Robin.

2.4.2 Správa pamäte

Správa pamäte má za úlohu prideľovať pamäť novým procesom a zabezpečovať aby tieto nezasahovali do pamäte iných procesov. Pamäť môže byť pridelená staticky alebo dynamicky. Správa pamäte si pamätá priestor, ktorý je ešte voľný, a prideľuje ho procesom nasledujúcimi algoritmami [3]:

- **prvý vyhovujúci (first fit)** - procesu sa pridelí prvý úsek rovnaký alebo väčší ako požadovaná veľkosť.

- **nasledujúci vyhovujúci (next fit)** - rovnako ako *prvý vyhovujúci* len sa hľadá od miesta posledného vloženia.

- **najlepšie vyhovujúci (best fit)** - procesu sa nájde úsek, ktorý najlepšie vyhovuje požadovanej veľkosti.

- **najhoršie vyhovujúci (worst fit)** - opak *najlepšie vyhovujúceho*.

- **rýchlo vyhovujúci (quick fit)** - úseky radené do viacerých radov podľa veľkosti. Vyberá sa prvý úsek z rady, ktorá vyhovuje.
- **Buddy algoritmus** - založený na báze delenia úsekov na polovice kým daný úsek najlepšie nepasuje.

Z hľadiska rozdelenia pamäte poznáme [3]:

- **pevné rozdelenie** - pamäť je rozdelená na rovnaké úseky, ktoré sú pridelené procesom. Keďže proces nemá vždy rovnakú veľkosť stránka nie je vždy využitá a vzniká interná fragmentácia.
- **variabilné rozdelenie** - pamäť je pridelená podľa potrieb procesu. Medzi procesmi vznikajú úseky, ktoré sú malé na to, aby sa tam zmestil ďalší proces a vzniká externá fragmentácia. túto fragmentáciu je možné odstrániť defragmentáciou (zreorganizovaním procesov v pamäti).
- **Stránkovanie** - pamäť je rozdelená na rovnako veľké úseky, stránky. Procesu sú pridelené stránky podľa potreby. Výhoda oproti pevnému rozdeleniu je menšia úroveň internej fragmentácie.
- **segmentovanie** - pamäť rozdelená do segmentov, tak že segment môže obsahovať len jeden druh informácií procesu (inštrukcie, statické dáta, dynamické dáta, privátne dáta...).

2.4.3 Správa vstupov a výstupov

Správa V/V je dôležitým elementom jadra operačného systému. Úlohou správy systému je riadiť pristupovanie zariadení a výmenu dát medzi zariadením a procesom. Vieme definovať tri typy komunikácie závislé od návrhu komunikačného protokolu zariadenia a okolia. [4]:

- **Synchrónna komunikácia** - prebiehajúca na báze presnej, časovanej komunikácie. Druh komunikácie výhodný najmä z hľadiska úspor energie, pretože zariadenie sa môže vypínať, kým nepríde synchronizačný signál z interných hodín procesora.
- **Semisynchrónna komunikácia** - prebiehajúca na báze výmeny synchronizačných signálov. Zariadenie nevie kedy presne začne komunikácia ale vie, že začiatok komunikácie je determinovaný synchronizačným signálom, prerušením. Táto metóda má rovnaké výhody ako synchrónna naviac sa nevyžaduje presné časovanie, tým sa môžu vypnúť interné hodiny procesora čím klesne spotreba.
- **Asynchrónna komunikácia** - prebiehajúca na báze výmeny synchronizačných znakov. Zariadenie počúva pridelenú linku a hľadá na nej synchronizačný znak. Metóda nevýhodná najmä z dôvodu, že zariadenie musí byť neustále aktívne.

2.5 Štatistické merania

Jedným zo stanovených cieľov je implementovanie štatistického modulu. Úlohou tohto modulu je vytvárať obraz o efektivite a zlepšení MOS voči súčasným VOS. Štatistický modul by mal byť schopný merať priepustnosť operačného systému, času vykonávania jednotlivých procesov, času aktívneho vykonávania procesov, réžiu jadra, dĺžka aktívnosti hardvérových prostriedkov, počty spustených procesov, dodržania limitov procesov reálneho času, analyzovanie poradia spúšťania procesov, komunikácie medzi procesmi.

Štatistický modul bude mať v jadre výrazný vplyv na réžiu jadra, pretože bude merať veľké množstvo informácií. Dôležité je, aby jadro bolo schopné merať tak, aby do štatistík nezanieslo nejakú informáciu o sebe, čo nebude také jednoduché, pretože jeho réžia ovplyvní napríklad komunikáciu s okolím. Preto bude vhodné navrhnúť štatistický modul tak, že počas jedného merania sa bude sledovať len nejaká podmnožina z množiny veličín.

2.6 Prehľad vnorených operačných systémov

Pre účely porovnávania s MOS boli zvolené operačné systémy FreeRTOS, AvrX a TinyOS.

2.6.1 AvrX

Výber tohto operačného systému ovplyvnila predchádzajúca skúsenosť. Nevýhodou tohto operačného systému je jeho neprehľadná dokumentácia. Operačný systém je navrhnutý výlučne pre procesory Atmel AVR. Výhodou a zároveň aj nevýhodou je, že jadro je implementované v jazyku symbolických inštrukcií, a teda možno predpokladať veľkú hustotu kódu ale prenositeľnosť na iné procesory je značne komplikovaná. AvrX má v závislosti od konfigurácie 500-700 slov kódu a je navrhnuté ako knižnica rutín.[5]

AvrX je preemtívny a prioritne riadený operačný systém. Plánovač úloh je realizovaný šestnásť-úrovňovým radom. Každý rad obsahuje úlohy s rovnakou prioritou. Každá úloha je pritom do svojho radu vložená na jeho koniec (tzv. metóda "round-robin"). Réžia jadra, pri taktovaní procesora 10 MHz a frekvencii prepínania úloh 10 kHz, vyžaduje priemerne 20 percent procesorového času. AvrX má implementované semafory vhodné pre synchronizáciu úloh alebo riadenie vylúčenia (tzv. "mutual exclusion"). Volania využívajúce semafory majú blokujúci aj neblokujúci charakter. Väčšina neblokujúcich volaní má svoj ekvivalent aj v používateľom definovaných prerušeniach. AvrX implementuje rad časovačov na riadenie časovaných udalostí nastaviteľných volaním jadra.

Prepnutie úlohy trvá 92 cyklov. Počas prepnutia sa odložia všetky registre procesora (32 registrov, SREG, PC) do zásobníka úlohy. Po prepnutí úlohy sú všetky registre inicializované na nulu. Každá úloha má svoj vlastný zásobník, na ktorom sa odkladá kontext úlohy počas

prepnutia a vnorenie do funkcií. Jadro má svoj vlastný zásobník. Ukazovateľ na vrchol zásobníka, priorita a umiestnenie v rade sú odložené v kontrolnom bloku úlohy (TCB). TCB má veľkosť 6B. Úloha môže nadobúdať stavy: *inicializácia*, *spustená (running)*, *pozastavená (suspended)*, *blokovaná (blocked)*, *pripravená (ready)*.

Semafory nadobúdajú stavy *obsadený (pend)* , *čakajúci (wait)* a *voľný (done)*. Ak je semafor v stave *obsadený* pre konkrétnu úlohu, pre inú úlohu je stav semafora *čakajúci*. Na semafor smie čakať viac úloh, pričom tieto úlohy sú blokované.

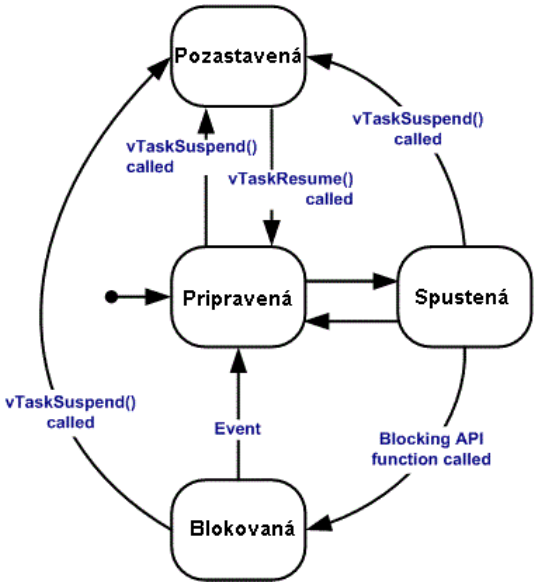
Softvérové časovače reprezentujú implementáciu čakania na udalosť. Každý časovač je identifikovaný kontrolným blokom časovača s veľkosťou 6B. Každý časovač je po jeho vytvorení zaradený na koniec radu časovačov. V kontrolnom bloku časovača je uchovávaná 16b hodnota reprezentujúca dobu, po uplynutí ktorej časovač vyprší. Táto hodnota je vypočítaná ako rozdiel používateľom zadaného oneskorenia a súhrnného oneskorenia určeného predchádzajúcimi časovačmi.

Medziprocesná komunikácia je realizovaná pomocou radov správ. Každý rad správ je definovaný kontrolným blokom správy (MCB). Správa môže byť do radu zaradená procesom alebo používateľom definovaným prerušením. Jednu správu vie prijímať viac procesov. Každá správa je spolu so semaforom a ukazovateľom na ďalšiu správu zabalená do obálky.

2.6.2 FreeRTOS

Tento operačný systém bol zvolený kvôli jeho popularite v oblasti návrhu jednoduchých vnorených systémov. FreeRTOS je real-time operačný systém navrhnutý pre menšie vnorené systémy. Podporuje 27 architektúr najmä z rodiny ARM7 a ARM Coretex-m3. FreeRTOS je jadro, ktoré môže byť nastavené do kooperatívneho, preemptívneho alebo hybridného módu. Prevažná časť OS je implementovaná v jazyku C. Časti kódu závislé od architektúry procesora sú napísané v jazyku symbolických inštrukcií. Jadro dosahuje veľkosť 4-9 kB, podporuje tvorbu štandardných procesov (úloh) a špeciálnych co-rutín (co-routines). Synchronizácia úloh je zabezpečená prostredníctvom semaforov, zámok (mutex) a radov. OS má možnosť nastavenia detekcie pretečenia zásobníka. FreeRTOS nemá softvérové obmedzenie počtu úloh a počtu priorít úloh. Okrem ponúkaných vlastností je operačný systém podporený veľmi kvalitnou dokumentáciou, ukážkovými riešeniami a návodmi.[6]

Úlohy (procesy) môžu byť v stavoch *bežiaca (running)*, *pripravené (ready)*, *blokované (blocked)* a *zastavené (suspended)*. Na obrázku 2.9 sú zobrazené povolené prechody stavov úloh.



Obr. 2.9: Prípustné prechody úloh stavmi [6]

Plánovač operačného systému zabezpečuje pridelenie procesorového času úlohe s najvyššou prioritou pred úlohami s nižšou prioritou. Množstvo priorít sa nastavuje v konfiguračnom súbore pred kompiláciou operačného systému. Každá úloha má svoj vlastný zásobník. Úloha je implementovaná používateľom ako jednoduchá funkcia, (obr. 2.10) zvyčajne s nekonečnou slučkou v jej tele. Funkcii sa prostredníctvom parametra dajú na vstup pripojiť rôzne dáta. Úloha je vytvorená a ukončená volaním funkcií *xTaskCreate()* a *xTaskDelete()*.

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        ...
    }
}
```

Obr. 2.10: Rámec úlohy [6]

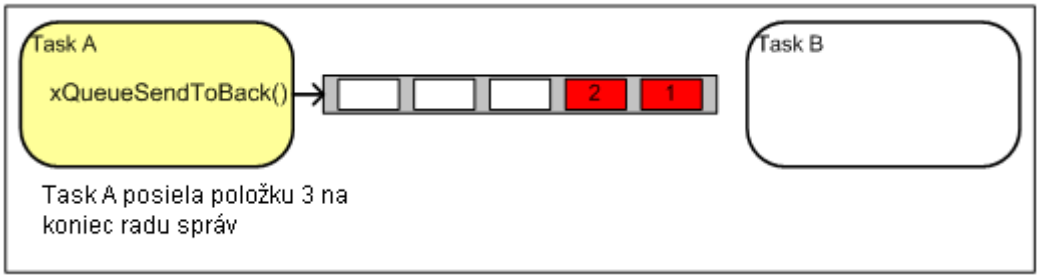
OS implementuje *Idle Task*, ktorá je automaticky spustená jadrom po inicializácií. Jej úlohou je čistenie pamäte po skončených úlohách. Počas vykonávania *Idle Task* je spúšťaná funkcia *Idle Task Hook*. Táto funkcia je vhodná na spúšťanie úloh s rovnakou prioritou

ako *Idle Task*. *Idle Task Hook* sa používa najmä na spúšťanie Co-rutín alebo prepínanie procesora do úsporného módu.

Pomocou takzvanej *Co-rutiny* je možné implementovať systém s menšou spotrebou pamäte pretože všetky co-rutiny majú spoločný zásobník.

Medziprocesná komunikácia je zabezpečená prostriedkami ako sú *rady (queue)*, *zámky (mutex)* a *semafory*.

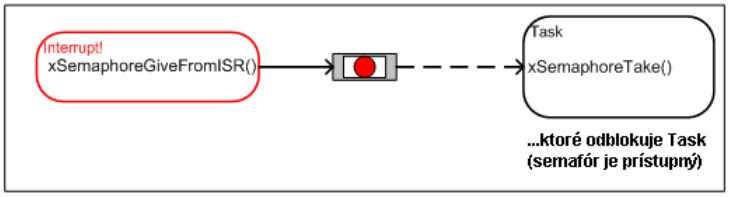
Rady sa najčastejšie používajú na prenos informácií medzi dvoma úlohami alebo medzi úlohou a prerušením (Obr. 2.11). Počas inicializácie radu medzi úlohami sa určí veľkosť jedného bloku dát, maximálny počet blokov v rade a blokovací čas. Najčastejšie sú dáta uložené v rade ako kópia. Dáta sa tým ochránia pred prepísaním. Ak je potrebné ušetriť priestor v pamäti je možné vložiť do radu ukazovateľ na dáta. Blokovací čas určuje počet tikov procesora počas, ktorých je úloha blokovaná. Úloha je blokovaná ak chce čítať z prázdneho radu alebo ak chce zapisovať do plného radu. Ak sa maximálny počet blokov dát v rade nastaví na "1" tak rad plní aj synchronizačnú úlohu.



Obr. 2.11: Príklad radu medzi dvoma úlohami [6]

FreeRTOS implementuje viacero druhov semaforov a zámok. Semafory a zámky sú použiteľné na synchronizáciu medzi dvoma úlohami alebo prerušením a úlohou.

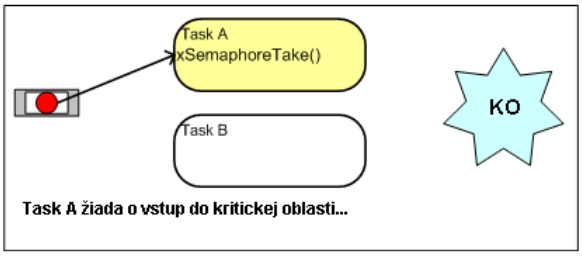
- **Binárne semafory** sa používajú pre prístup k zariadeniam a pre synchronizačné úlohy. Binárny semafor sa od zámky odlišuje v tom, že neimplementuje dedenie priority. Tým sa binárne semafory stávajú vhodnejšími na realizáciu synchronizácie (Obr. 2.12). Semafory majú nastaviteľnú dobu blokovania úlohy. Blokované úlohy na semafore sú zoradené na základe priority.



Obr. 2.12: Semafor ako prostriedok synchronizácie: prerušenie a úloha [6]

- **Štandardné semafory** majú nastaviteľnú hladinu, počet úloh, ktoré môžu vstúpiť do kritickej oblasti. Používajú sa na počítanie udalostí. Semafor je inicializovaný na stav *zamknutý*, teda na nulu. Po príchode udalosti sa zvýši hodnota semaforu a po spracovaní udalosti úlohou sa jeho hodnota zníži. Druhou možnosťou ako použiť štandardný semafor je riadenie prístupu ku zdrojom. V takomto prípade je semafor nastavený na nejakú počiatočnú hodnotu "N". Ak úloha chce pristúpiť ku zdroju tak zníži hodnotu semaforu. Ak hodnota semaforu dosiahne nulu žiaden zdroj nie je voľný a úloha je blokovaná.

- **Zámky** narozdiel od binárneho semaforu implementujú dedenie priority, preto sú zámky vhodnejšie pre riadenie prístupu k zariadeniu. Zámka vystupuje ako token. Na to aby úloha mohla pristúpiť k zdroju musí si vziať token (Obr. 2.13).



Obr. 2.13: Zámok vystupujúci ako token prístupu k zdroju [6]

- **Rekurzívne zámky** umožňujú viacnásobné obsadenie zámky vlastníkom. Ak si úloha obsadí zámku N-krát, tak ostatné úlohy k zámke nemôžu pristúpiť pokým úloha neuvoľní zámku N-krát. Zámky na rozdiel od semaforov nemôžu byť použité v používateľom definovaných prerušeniach.

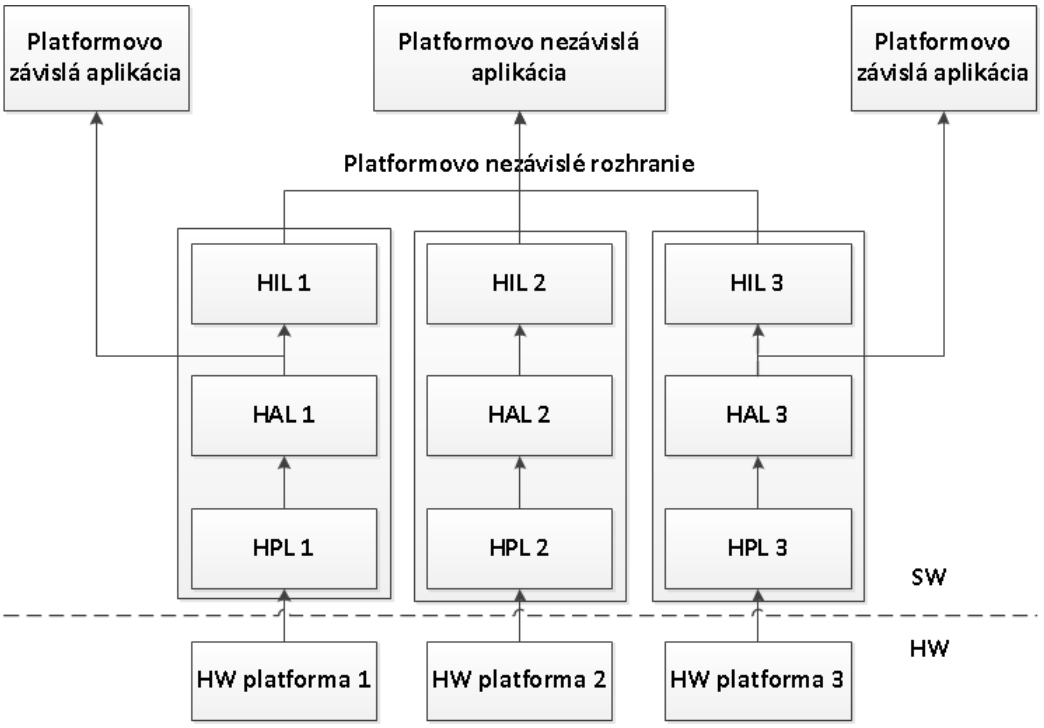
Mimo jadra je implementovaná možnosť softvérových časovačov vo forme služby (deamon). Softvérové časovače sa používajú na časovanie spúšťania úloh. Časovač po vypršaní zavolá *callback* funkciu, v ktorej je možné definovať spúšťanú úlohu.

FreeRTOS okrem štandardných úloh poskytuje služby sledovania behu úloh posielaním dát na digitálny alebo analógový výstup (možné pripojenie analyzátora). Meria štatistiku trvania vykonania úlohy v absolútnom čase, percentuálny podiel absolútneho času a času stráveného na procesore. Operačný systém podporuje správu pamäte a pre ARM Coretex M3 jadrá aj MPU (memory protection unit).

2.6.3 TinyOS

Operačný systém implementovaný v programovacom jazyku nesC, ktorý je dialektom jazyka C. NesC rozširuje štandardné C o formu modulárnosti implementovaním mechanizmu rozhraní medzi komponentami systému.

Hlavnou myšlienkou TinyOs je zavedenie architektúry hardvérovej abstrakcie (Hardware Abstraction Architecture skr. HAA) (Obr. 2.14). Hardvérová abstrakcia zvyšuje znovupoužiteľnosť (prenositeľnosť) zdrojového kódu a zjednodušuje návrh aplikácií skrývaním hardvérovej zložitosti. Naopak hardvérová abstrakcia je v konflikte s výkonnosťou a energetickou efektivitou senzorových sietí. Preto HAA hľadá kompromis medzi efektivitou a jednoduchosťou návrhu aplikácií.[7]



Obr. 2.14: Architektúra hardvérovej abstrakcie HAA

HAA je organizovaná do troch vrstiev. Každá vrstva má definovanú svoju úlohu a je závislá od rozhrania definovaného nižšou vrstvou. Narozdiel od dvojvrstvových riešení použitých v iných VOS ako je napríklad WindowsCE, troj-úrovňová štruktúra poskytuje väčšiu mieru platformovej nezávislosti. HAA umožňuje pripojenie používateľskej aplikácie na rozhranie najvyššej a strednej vrstvy abstrakcie.

Najnižšou vrstvou abstrakcie je vrstva Prezentačná (Hardware Presentation Layer skr. HPL). Táto vrstva je umiestnená priamo nad hardvérom a prezentuje možnosti hardvéru voči vyšším vrstvám. Vrstva pristupuje k hardvéru ale aj prijíma prerušenia od hardvéru. Prezentačná vrstva tvorí čitateľnejšie rozhranie voči vyšším vrstvám a skrýva zložitosť.

Strednou vrstvou abstrakcie je Adaptačná vrstva (Hardware Adaptation Layer skr. HAL). Adaptačná vrstva reprezentuje jadro architektúry HAA. Zapúzdruje zložitosť HPL. Na rozdiel od HPL smie mať HAL stavový charakter. Adaptačná vrstva je stále platformovo špecifická s cieľom najlepšieho využitia vlastností hardvéru.

Najvyššou vrstvou abstrakcie je vrstva rozhrania (Hardware Interface Layer skr. HIL).

HIL zapúzdruje platformovo špecifickú HAL do platformovo nezávislej reprezentácie pre platformovo nezávislé aplikácie.

Všetky súčasti operačného systému sú implementované v nesC ako komponent. Každý komponent má svoje komunikačné rozhranie. Plánovač ako komponent prijíma komponenty úloh. Plánovač je implementovaný ako FIFO rad. V rade smie byť maximálne 255 úloh, pričom každý typ úlohy smie byť v rade práve raz. Úloha sa môže zaradiť do radu znova tým, že sa sama odloží do radu počas vykonania alebo v prípade, ak bola vytvorená systémom.

3 Špecifikácia

Operačné systémy navrhnuté pre vnorené aplikácie spĺňajú okrem základných požiadaviek (spoľahlivosť, presnosť, efektívnosť a férovosť) štandardných operačných systémov aj podmienky vyššej odolnosti voči poruchám behu programu. Zväčša pracujú s obmedzenejšími výkonovými, energetickými a pamäťovými kapacitami. Preto musia byť navrhnuté s dôrazom na šetrenie energie a s tým spojenou minimalizáciou réžie jadra.

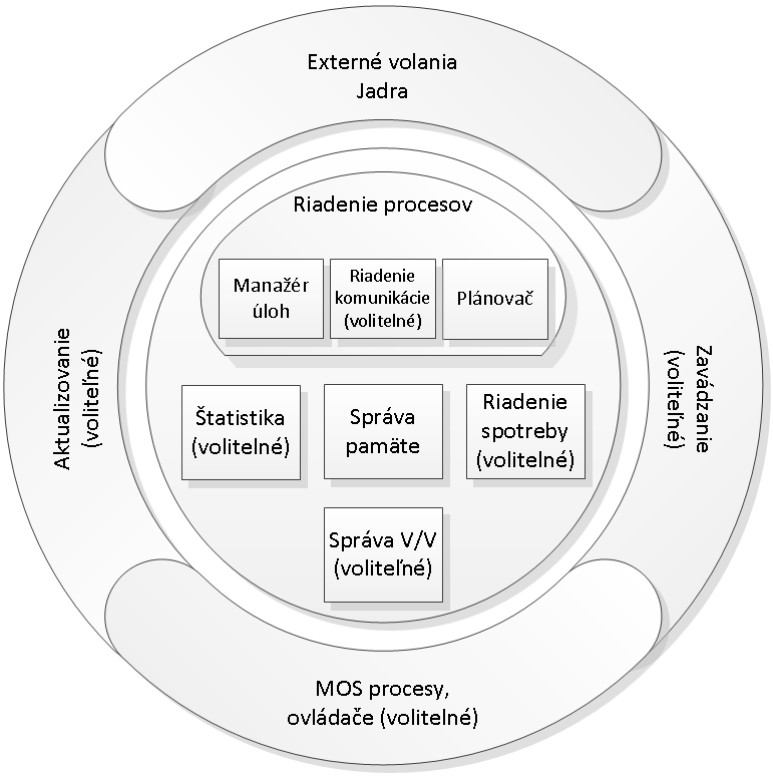
3.1 Požiadavky na OS

Modulárny operačný systém bude navrhnutý tak, aby spĺňal požiadavky:

- modulárnosti na úrovni jadra a pridružených súčastí OS. Modulárnosť zabezpečí jednoduchú výmenu modulov podľa potreby a možností hardvéru alebo štruktúry procesov.
- riadenia spotreby energie s dôrazom na zvýšenie výdrže systému na úkor výkonu. Spotreba energie bude regulovateľná podľa vlastností zdroja. V prípade napájania z elektrickej siete alebo obnoviteľného zdroja bude možné regulovať spotrebu s ohľadom na zvýšenie výkonu na úkor výdrže systému. Riadenie spotreby energie závisí od možností procesora.
- univerzálnosti s možnosťou prenesenia na rôzne hardvérové architektúry. Prispôsobiteľnosť je základným úspechom každého operačného systému.
- obsluhy real-time procesov bez záruky dodržania časových limitov.
- merateľnosti z pohľadu porovnávania výkonnosti a efektívnosti s inými operačnými systémami.
- rekonfigurácie počas behu operačného systému z hľadiska pridávania nových programov a dopĺňania kódu programom, ktoré nemajú priradené spustené procesy.

3.2 Návrh štruktúry operačného systému

Modulárny operačný systém bude navrhnutý ako monolitický modulárny systém. Operačný systém bude pozostávať z jadra a z periférie. (obr. 3.1).



Obr. 3.1: Architektúra operačného systému

Na úrovni jadra budú navrhnuté moduly:

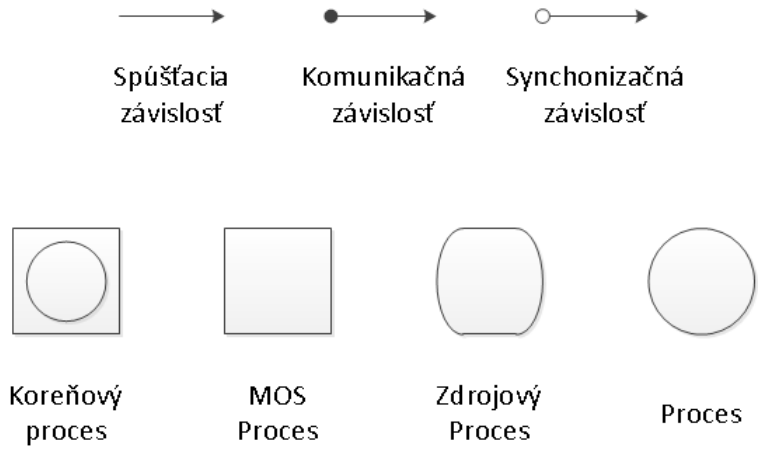
- **procesného riadenia**, ktorý bude zabezpečovať prepínanie procesov (manažér úloh), plánovanie procesov (plánovač, scheduler) a riadenie komunikácie medzi procesmi. Moduly plánovača budú navrhnuté viacerými metódami podľa rôznych kritérií ako sú úspora réžie jadra, úspora pamäte, zvýšenie výkonu a férovosť. Procesné riadenie bude povinnou časťou jadra.
- **správy vstupov a výstupov**, ktorý bude zabezpečovať správu prístupu k hardvérovým zariadeniam.
- **riadenia spotreby**, ktorý bude zabezpečovať riadenie spotreby energie a je navrhnutý podľa úrovne pridelených kompetencií. Od najnižšej úrovne (vypínanie neaktívnych zariadení) až po najvyššiu úroveň (zasahovanie do riadenia procesov a celého systému aktívnym uspaním).
- **správy pamäte**, ktorý bude spravovať pridelené pamäťové prostriedky. Návrh modulu bude pracovať len s hlavnou pamäťou. Systém s pevným diskom (swaping, paging) nebude realizovaný. Správa pamäte bude povinnou časťou jadra.
- **štatistiky**, ktorý bude zameraný na meranie priemerne stráveného času procesov na procesore, priemernej dĺžky vykonávania procesu a počtu aktívnych zariadení.

Na úrovni periférie budú implementované externé volania jadra operačného systému ako sú:

- **volania externého prostredia** - volania jadra súvisiace napríklad s riadením spotreby.
- **volania ovládačov** - obsahujúci MOS procesy spravujúce pridružené zariadenie.
- **volania zavádzania** - schopný zavedenia nového programu do pamäte počas behu systému.
- **volania aktualizácie** - schopného zmeny kódu programu počas behu systému.

3.3 Programový a procesný model

Pre potreby modulárneho operačného systému je zavedená konvencia značenia procesov a vzťahov medzi nimi na báze, grafov. Na obrázku 3.2 sú znázornené tri druhy hrán a štyri druhy vrcholov, ktoré sú identifikované v systéme. Konvencia je vhodná na simuláciu vzťahov medzi procesmi v čase ale aj vzťahov procesov všeobecne podľa návrhu štruktúry programového vybavenia. Na reprezentovanie stavu systému v čase budeme používať grafovú štruktúru, ktorá bude reprezentovaná procesným grafom (obr. 3.4). Na reprezentovanie programovej štruktúry systému budeme používať grafovú štruktúru reprezentovanú programovým grafom (obr. 3.3). Všeobecne platí, že v procesnom grafe sa bude vyskytovať podmnožina procesov a závislostí programového grafu, pričom procesy a závislosti sa môžu opakovať. Spúšťacia závislosť (šípka) znázorňuje vzťah medzi spúšťacím, rodičov-



Obr. 3.2: Konvencia

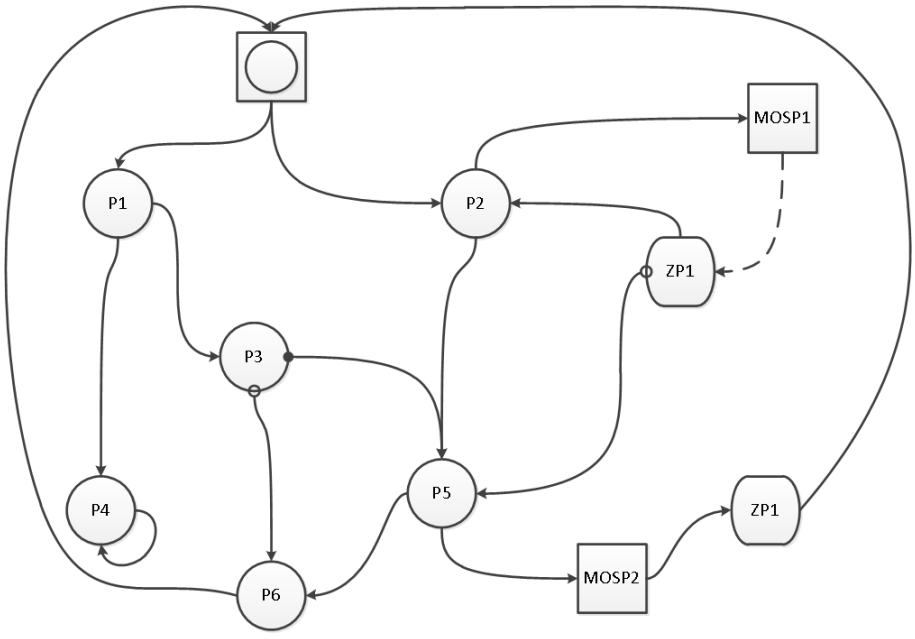
ským procesom (na začiatku šípky) a spúšťaným, dcérskym procesom (na konci šípky). V konfigurácií programového systému znázorňuje závislosť možnosť procesu priradeného jednému programu spustiť proces priradený druhému programu. Stav systému v čase znázorní závislosť aktuálny vzťah rodič a potomok.

Komunikačná závislosť (šípka s plným krúžkom) znázorňuje jednosmerný vzťah výmeny dát medzi dvoma procesmi v rovnakom slede ako pri spúšťacej závislosti. Synchronizačná závislosť je špeciálnym prípadom komunikačnej závislosti znázorňujúca vzťah čakania procesu, do ktorého vstupuje závislosť na signál od procesu, z ktorého závislosť vystupuje.

Modulárny operačný systém bude po inicializácií jadra systému spúšťať jeden inicializačný proces nazývaný koreňovým. Koreňový proces slúži pre používateľa na prípravu prvých procesov, ktoré budú v systéme spustené.

Okrem štandardných procesov, ktorých predkom je koreňový proces budú v systéme existovať procesy, ktoré boli spustené zásahom z okolitého prostredia. Procesy, ktorých predok nie je koreňový proces nazývame zdrojový proces. Zdrojový preto lebo ich spúšťačom bol nejaký zdroj ako je napríklad sériové rozhranie alebo systém prerušenia. Takýmto procesom vieme do systému zaviesť prvok asynchrónnosti. Niektoré informácie sa dostávajú do systému len v predom neurčených intervaloch a nie je potrebné, preto pre ne udržovať proces v hlavnom toku.

Neštandardnými procesmi v systéme sú procesy modulárneho operačného systému, MOS procesy. Tieto procesy sú procesmi operačného systému, ktoré budú štandardnou výbavou alebo doplnením funkcionality systému. Takéto procesy slúžia ako vrstva nad rozhraniami systému.

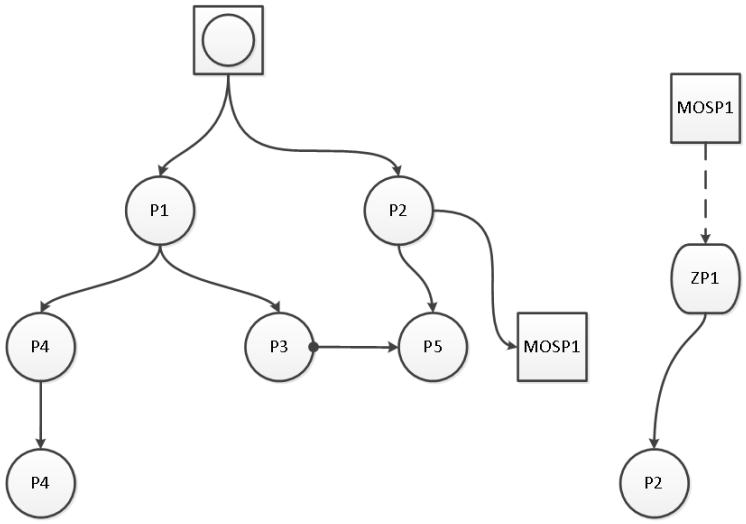


Obr. 3.3: Príklad programového grafu

Na obrázku 3.3 je príklad programového grafu kde sú zobrazené prípady ako sa môže znovu inicializovať koreňový proces, príklady synchronizačnej a komunikačnej závislosti, príklady spúšťacej závislosti (špeciálne pre proces P4), vznik zdrojového procesu a prístup na

MOS proces. Dôležité je upozornit', že spúšt'acia závislost' medzi MOSP1 a ZP1 je špeciálnym prípadom spúšt'ania, kedy je proces spustený externým zásahom.

Na obrázku 3.4 je príklad procesného grafu, ktorý je reprezentáciou spomínaného programového grafu v konkrétnom čase. MOSP1 proces je v grafe zobrazený dvakrát aj keď v systéme bude spustený len raz pretože to vyplýva z návrhu modulárneho operačného systému. V tomto prípade je proces rozdelený pre zdôraznenie nezávislosti ZP1 procesu od zbytku procesov.



Obr. 3.4: Príklad procesného grafu

3.4 Riadenie procesov

Informácie o každom procese spustenom na systéme budú uložené v tabuľke procesov. Jeden záznam tabuľky môže obsahovat': identifikáciu procesu, adresu začiatku pridelenej pamäte dát, veľkost' pridelenej pamäte dát, stav pamäte dát, stav registrov, ukazovateľ na vrchol zásobníka, programové počítadlo atď.

Úlohy budú prepínané po uplynutí času pridelenom úlohe operačného systému. MOS bude preemptívny. Manažér úloh bude mat' za úlohu riadit' životný cyklus procesov (vytvorenie, prepnutie, zrušenie). Manažér úloh bude mat' jednotlivé funkcionality nastaviteľné pred kompiláciou.

Plánovanie úloh bude realizované pomocou štruktúry spájaný zoznam. Ak je úloha pripravená na vykonávanie, tak sa vloží do zoznamu čakajúcich úloh na procesor. V prípade veľkého množstva úloh na plánovanie (nedostatok voľnej pamäte) sa nové úlohy nebudú vytvárat'. Bude možné nastavit' odloženie vytvorenia úlohy do takzvaného Jobu. Vloženie do zoznamu úloh bude realizované rôznymi metódami. Používateľ si pred kompilovaním MOS zvolí metódu plánovania.

Synchronizácia a komunikácia medzi procesmi bude realizovaná prostredníctvom toku dát (stream). Tok dát bude mať práve jedného odosielateľa (sender) a práve jedného príjemcu (receiver). Odosielateľ sa bude musieť registrovať na odosielanie dát danému príjemcovi. Príjemca si bude viesť záznamy o všetkých registrovaných odosielateľoch. Príjemca potvrdí spojenie s odosielateľom jeho výberom zo zoznamu registrovaných odosielateľov. Dáta medzi odosielateľom a príjemcom budú prenášané v štruktúre rad (queue) ako odkaz na dáta. Komunikácia bude realizovaná pomocou blokujúcich aj neblokujúcich volaní. Pomocou toku dát sa bude realizovať aj synchronizácia procesov.

3.5 Správa pamäte

Informácie o každom programe uloženom v pamäti systému budú uložené v tabuľke programov. Jeden záznam tabuľky môže obsahovať: identifikáciu programu, adresu začiatku kódu programu, veľkosť kódu programu, veľkosť potrebnej pamäte dát, počet spustených procesov.

Správca pamäte bude mať za úlohu prideľovať pamäť novým úlohám a prideľovať pamäť už spusteným úlohám. V prípade, že už nie je dostatok priestoru na vytvorenie novej úlohy, je požiadavka na vytvorenie úlohy odložená. Požiadavka môže byť realizovaná, ak budú dostupné zdroje v pamäti. Procesy s rovnakým rodičovským programom budú mať pridelenú rovnakú pamäť programu a vlastnú dynamickú dátovú pamäť. Statická pamäť dát nebude uvažovaná.

3.6 Správca V/V zariadení

Úlohou správcu V/V zariadení bude riadiť prístup k zariadeniam za pomoci *MOS procesov*, ktoré budú voliteľnou súčasťou modulárneho operačného systému. V nastavení bez *MOS procesov* bude správca pamäte obsluhovať prístup k zariadeniam prostredníctvom volaní. Procesy budú pristupovať k zariadeniam prostredníctvom volania a zariadenie bude oslovovať proces volaním uloženým v prerušení. S *MOS procesmi* bude každé alebo individuálne zvolené zariadenie obalené Prezentačnou vrstvou tvoriacou *MOS proces*. K *MOS procesu* budú ostatné procesy pristupovať pomocou vytvoreného toku dát.

3.7 Riadenie spotreby energie

Voliteľným modulom MOS bude riadenie spotreby energie. Tento modul bude rozširovať funkcionalitu ostatných modulov. Bude riadený prostredníctvom externých volaní. Externým volaním sa môže nastaviť mód plánovania úloh a prepínania úloh. Počas prepnutia môže napríklad manažér úloh vyradiť procesor z činnosti jeho uvedením do úsporného režimu, a

tým predĺžiť výdrž systému. Môže sa zmeniť spôsob vykonávania plánovania obmedzením vykonania úloh len na tie s najvyššou prioritou. Ostatné úlohy sú uvedené do pozastaveného stavu.

3.8 Štatistika

Štatistický modul modulárneho operačného systému bude zaznamenávať históriu vykonania procesov. Bude viesť štatistiku každého procesu a jadra. Predmetom štatistiky bude trvanie vykonania úlohy celkovo a na procesore. Zároveň bude vedená aj štatistika jednotlivých pripojených zariadení k procesoru. Prostredníctvom *MOS procesu* bude jednoznačne zistiteľná dĺžka aktivity zariadenia, a teda nepriamo sa bude dať odvodiť spotreba energie.

3.9 Periféria operačného systému

Okrem spomínaných *MOS procesov* bude operačný systém mať možnosť úpravy už vložených programov. Vložený program sa jednoducho nahradí novou verziou programu. Počas behu bude možné do pamäte procesora vkladať aj nové programy. Vkladanie programu sa bude realizovať prostredníctvom procesu, ktorý bude štandardne spustený a plánovaný v operačnom systéme. Po uložení programu bude proces štandardne ukončený.

4 Návrh

Táto kapitola sa podrobnejšie zaoberá návrhom jednotlivých modulov navrhovaného operačného systému.

4.1 Riadenie procesov

Ústrednou časťou jadra MOS bude riadenie procesov. Riadenie zabezpečí väčšinu práce súvisiacej s procesmi. Súčasťou riadenia procesov budú moduly prepínania procesov, plánovania procesov a riadenia komunikácie medzi procesmi. Cieľovým objektom Riadenia procesov budú programy a procesy.

4.1.1 Program

Program predstavuje zdrojový kód pre proces. Každý program bude uvádzaný v systéme svojou hlavičkou PCB. Hlavička bude obsahovať záznamy o:

- začiatku programu,
- konci programu,
- počiatočnej veľkosti dátového priestoru,
- informáciu o počte spustených procesov a
- počiatočnej priorite nového procesu.

Hlavičky procesov budú ukladané v zozname programov.

4.1.2 Proces

Proces spustený v MOS bude mať hlavičku TCB uloženú v tabuľke spustených procesov, ktorú bude spravovať manažér úloh. Obsahom hlavičky procesu bude:

- **ukazovateľ na vrchol zásobníka**, ktorý bude nepovinnou súčasťou hlavičky. Vrchol zásobníka štandardne ukazuje na aktuálny prvok, ktorý bol naposledy vložený do zásobníka.
- **ukazovateľ na dno zásobníka**, ktorý bude povinnou súčasťou hlavičky. Štandardne ukazuje na začiatok zásobníka pričom vrchol zásobníka nemôže byť nižšie ako dno. Môže slúžiť ako ochrana pred podtečením zásobníka.

- **ukazovateľ na koniec zásobníka**, ktorý bude povinnou súčasťou hlavičky. Používa sa na ochranou pred pretečením zásobníka.
- **ukazovateľ na haldu**, ktorý bude nepovinnou súčasťou hlavičky. Bude odkazovať na pamäť pre alokovanie premenných procesu.
- **kontext procesu**, ktorý bude nepovinnou súčasťou hlavičky. Počas prepnutia úlohy bude možné kontext procesu odložiť do zásobníka alebo do hlavičky. Ak je kontext v zásobníku, tak hlavička procesu je menšia. Niektoré procesory majú špeciálnu inštrukciu, ktorá celý kontext odloží do zvoleného zásobníka. Na druhú stranu proces stráca časť svojho pamäťového priestoru, a vzniká problém, ak na zásobníku nie je dosť miesta na odloženie kontextu, čo si vyžiada dodatočné ošetrenie problému.
- **priorita**, ktorá bude nepovinnou súčasťou hlavičky, v závislosti od výberu spôsobu plánovania úloh.
- **prepožičaná priorita**, ktorá bude nepovinnou súčasťou hlavičky, v závislosti od implementácie prepožičania priority. Ak úloha s vysokou prioritou čaká na dáta od úlohy s nízkou prioritou, tak úloha s vysokou prioritou prepožičiava svoju prioritu úlohe s nižšou prioritou.
- **identifikátor (PID)**, ktorý bude povinnou súčasťou hlavičky. Reprezentuje číslo identifikujúce proces a zároveň aj program, ktorý je vykonávaný.
- **odkaz na zoznam registrovaných odosielateľov**, ktorý bude nepovinnou súčasťou hlavičky. Súvisí s komunikáciou medzi procesmi.

Procesu bude pridelený úsek pamäte obsahujúci časť určenú pre zásobník, a v závislosti od nastavenia MOS, aj časť určenú pre haldu. Statické dáta proces nebude môcť obsahovať, ak procesor nemá bázové registre. Staticky pridelený priestor dát bude programátor musieť nahradiť dynamickou alokáciou. Globálne premenné tiež nebudú použiteľné.

Okrem štandardných používateľských procesov bude MOS implementovať špeciálne typy procesov:

- **koreňový proces**, ktorý sa spustí vždy po inicializácii MOS. V tomto procese používateľ deklaruje, ktoré procesy majú byť spustené po inicializácii. V štandardných OS by sa tento proces dal prirovnať shellu ale nie doslovne, pretože používateľ nebude mať možnosť zadávať príkazy operačnému systému priamo (len v prípade pridania modulu používateľského rozhrania). Používateľ bude mať možnosť zmeniť koreňový proces úpravou jeho zdrojového kódu, a tým zmeniť spúšťané procesy. Koreňový proces bude nastaviteľný podla potreby tak, aby sa znovu spustil v predom definovanom intervale alebo aby ho spustil používateľský proces prípadne by sa spustil len po inicializácii systému.

- **MOS proces**, ktorý zapúzdri zariadenie pripojené k procesoru. K zariadenú bude možné pristúpiť len prostredníctvom tohto procesu. Tieto procesy budú čiastočne platformovo závislé a budú tvoriť prezentačnú vrstvu pre ostatné procesy. Môžu byť použité na sledovanie, zápis alebo čítanie zariadenia. MOS procesy budú voliteľné a budú sa dať vypnúť v prípade, že by neboli potrebné alebo ich používateľ nechce použiť. Budú špeciálne navrhnuté na minimálne zaťaženie systému. Aktivujú sa len v prípade volania z používateľského procesu alebo jadrom (prerušenie na zariadení). Aktívne zostanú len v prípade, ak existujú dáta na spracovanie. S týmito procesmi bude možné komunikovať prostredníctvom dátového toku.
- **zavádzací proces**, ktorý bude slúžiť na vkladanie nových programov do pamäte.

4.1.3 Manažér úloh

Manažér úloh bude spravovať všetky spustené procesy v systéme. Hlavičky procesov budú uložené v tabuľke procesov. Tabuľka procesov bude uložená v pamäti dát procesora. Okrem tabuľky procesov bude manažér úloh spravovať aj tabuľku programov. Manažér úloh bude riadený pomocou volaní jadra operačného systému. Pomocou volaní bude možné čítať povolené informácie o konkrétnej úlohe, ale nebude možné aby jedna úloha zasahovala do nastavení druhej úlohy.

Hlavnými úlohami manažéra úloh bude riadenie prepnutia procesu, vytvorenie nového procesu a zrušenie skončeného procesu. Prepnutie procesu je závislé od procesora, a preto bude realizované podľa možností daného procesora v jazyku symbolických inštrukcií. Vytvorenie nového procesu bude realizované volaním spawn() a bude závisieť od zostávajúceho množstva pamäte a množstva spustených procesov.

Po príchode požiadavky na vytvorenie nového procesu manažér úloh vykoná nasledujúce operácie:

1. skontroluje, či má voľný priestor pre proces v tabuľke procesov (ak je nastavené obmedzenie počtu spustených úloh). Ak priestor nie je, tak žiadosť sa v závislosti od výberu používateľa bude evidovať, vznikne odložená úloha (job) alebo sa žiadosť zahodí.

2. skontroluje, či neexistujú odložené žiadosti úloh (Joby). Ak existujú odložené žiadosti, tak novú žiadosť odloží a vytvorí úlohu, ktorá bola odložená (prípadne sa porovnajú priority žiadostí na základe počiatočnej priority úloh). Táto operácia bude voliteľnou pre používateľa.

3. podá žiadosť o vytvorenie priestoru správcovi pamäte. V prípade, že správca nemá dostatok priestoru, tak bude žiadosť v závislosti od výberu používateľa evidovaná alebo sa zahodí.

4. vytvorí proces v prípade, že priestor v pamäti dát je.

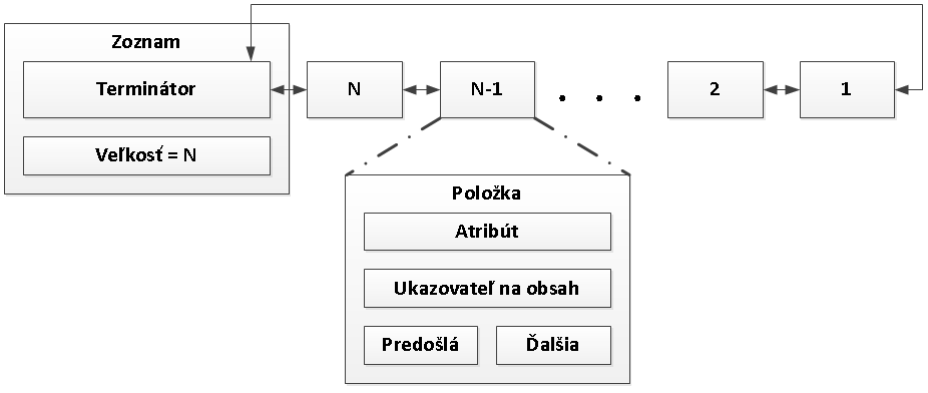
5. zaradí proces do plánovania.

Ak proces ukončil svoju činnosť manažér informuje správcu pamäte, že sa uvolnila pamäť daného procesu a odstráni hlavičku z tabuľky procesov. Manažér kontroluje, či neexistujú nejaké joby, žiadosti o vytvorenie nového procesu, ak áno tak sa job spracuje a vytvorí sa z neho proces. Táto možnosť bude voliteľná pre používateľa.

4.1.4 Plánovač úloh

Plánovač úloh bude rozhodovať o poradí plánovania procesov. Procesy budú usporiadané v štruktúre spájaný zoznam.

Dátová štruktúra spájaný zoznam (list) bude používaná najmä na prácu s úlohami (prístupná bude aj pre používateľa). Dátová štruktúra bude navrhnutá, tak aby bolo možné nové položky vkladať do zoznamu s ohľadom na prioritu položky alebo vkladať na koniec zoznamu bez ohľadu na prioritu položky. Spájaný zoznam bude obsahovať záznam o svojej veľkosti a terminátor. Terminátor bude vždy v zozname prítomný, a ak bude zoznam prázdny, tak bude ukazovať sám na seba, teda bude tvoriť kruhovú logickú štruktúru (obr. 4.1). Terminátor sa nebude započítavať do dĺžky zoznamu. Maximálna dĺžka zoznamu bude pevne stanovená počas kompilácie operačného systému (môže byť aj neobmedzená). Do zoznamu sa budú vkladať položky (items). Položka bude štruktúra tvorená odkazom na dáta, napríklad na TCB procesu. Bude obsahovať atribút, ktorý sa bude používať napríklad ako priorita, bude obsahovať ukazovateľ na predchádzajúci prvok v zozname a nasledujúci prvok v zozname.



Obr. 4.1: Štruktúra spájaného zoznamu a položky

Nad spájaným zoznamom budú implementované operácie starajúce sa o:

- inicializáciu spájaného zoznamu,

- vkladanie a vyberanie položiek spájaného zoznamu,
- hľadanie položiek v spájanom zozname,
- vyhľadanie prvej respektíve poslednej položky spájaného zoznamu a
- zistenie plnosti respektíve prázdnosti spájaného zoznamu.

Z hľadiska metód plánovania bude plánovač implementovaný týmito prístupmi:

- **plánovanie bez priority** vkladaním na koniec zoznamu pripravených procesov.
- **plánovanie s prioritou** vkladaním do zoznamu pripravených procesov podľa priority s možnosťou pridania funkcie prepožičiavania priority.
- **plánovanie s viacerými prioritnými zoznamami pripravených procesov** vkladaním procesu podľa jeho priority na koniec zoznamu určenej priority s možnosťou pridania funkcie prepožičiavania priority.

Modulárny operačný systém bude spracovávať aj procesy reálneho času, ale bez záruky dodržania časového limitu.

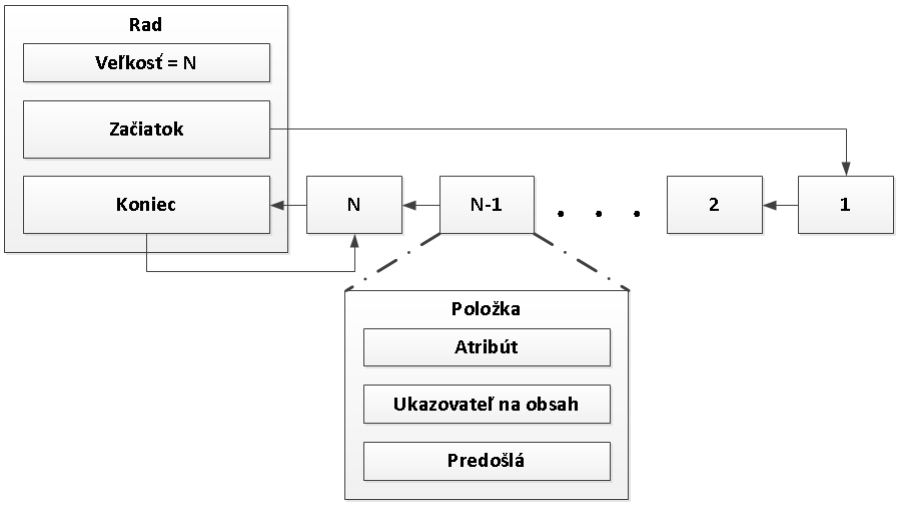
4.1.5 Riadenie komunikácie

Komunikácia medzi procesmi bude zabezpečená pomocou dátového toku. Dátový tok bude pozostávať zo štruktúry rad a ďalších riadiacich informácií.

Štruktúra rad (queue) (obr. 4.2) bude používaná ako jednosmerný kanál, v ktorom sa prenášajú dáta. Obsahom štruktúry rad bude aktuálna veľkosť, začiatok radu a koniec radu. Ak bude rad prázdny, tak koniec radu bude ukazovať na začiatok radu a naopak. Položky budú vkladané do radu vždy na koniec. Položka radu bude obsahovať atribút využiteľný pri synchronizácií, ukazovateľ na prenášaný obsah a ukazovateľ na ďalšiu položku v rade. Začiatok a koniec radu budú trvalé a nebudú sa započítavať do dĺžky radu. Dĺžka radu bude pevne stanovená počas kompilácie operačného systému.

Rad bude implementovaný z dôvodu nižšej spotreby pamäte ako spájaný zoznam a z dôvodu jednoduchších operácií nad radom. Nad radom budú implementované operácie starajúce sa o:

- inicializáciu radu,
- vkladanie a vyberanie položiek radu,
- vyhľadanie prvej respektíve poslednej položky radu a
- zistenie plnosti respektíve prázdnosti radu.

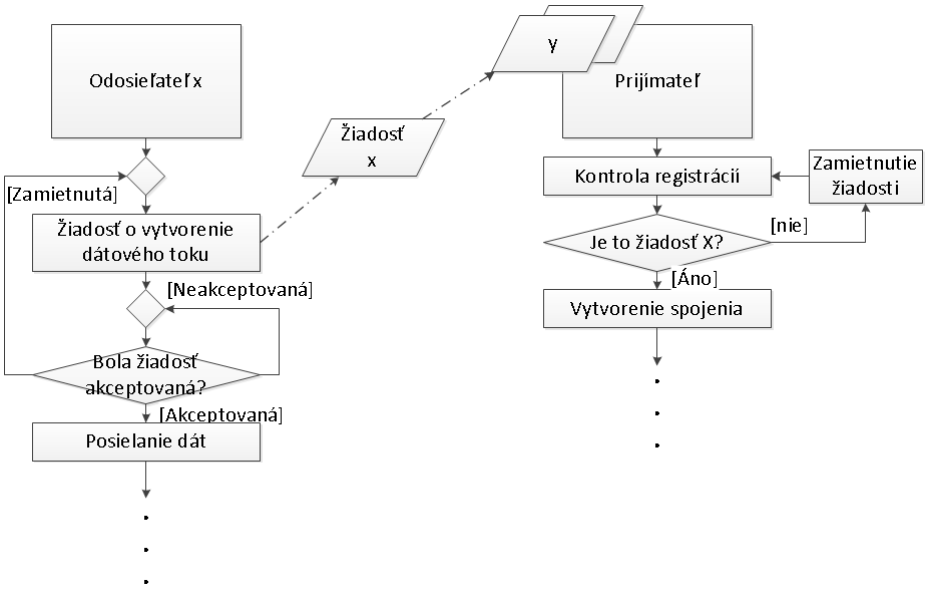


Obr. 4.2: Štruktúra radu a položky

Okrem radu bude hlavička toku dát obsahovať aj identifikátor registrovaného odosielateľa, stav dátového toku, priestor na uloženie pozastaveného odosielateľa, priestor na uloženie pozastaveného príjemcu a priestor pre zámku. Dátový tok bude mať jedného odosielateľa a jedného príjemcu.

Odosielateľ sa bude registrovať u príjemcu žiadosťou o vytvorenie spojenia. Žiadosť sa zaradí (podľa priority alebo na koniec) do zoznamu čakajúcich odosielateľov na pridelenie dátového toku. Odkaz na zoznam čakajúcich registrácií bude mať každý proces vo svojej hlavičke (TCB). Ak je zoznam čakateľov plný, tak daný žiadateľ bude odmietnutý. V prípade, že príjemca nechce prijímať žiadne dáta, tak registrovaný odosielatelia budú musieť čakať. Dátový tok je otvorený až po akceptovaní registrácie príjemcom. Na strane príjemcu sa bude dať rozhodnúť, či chce komunikovať s daným odosielateľom. Ak príjemca nechce komunikovať s odosielateľom, tak daná registrácia sa zamietne. Príjemca akceptovaním registrácie povolí odosielateľovi odosielať dáta (obr. 4.3).

Odosielanie dát bude implementované blokujúcimi aj neblokujúcimi volaniami. Odosielateľ bude pozastavený, ak sa pokúsi odoslať dáta do plného radu blokujúcim volaním. Odosielateľ bude mať možnosť poslať dáta a čakať na potvrdenie blokujúcim volaním. Príjemca bude prijímať dáta blokujúcim aj neblokujúcim volaním. V prípade pokusu o čítanie dát z prázdneho radu blokujúcim volaním bude príjemca pozastavený, kým nie sú pripravené dáta. Ak bude príjemca blokovaný a do radu budú vložené dáta, tak príjemca bude odblokovaný. Rovnako ak bude blokovaný odosielateľ a z radu budú odobrané dáta, tak odosielateľ bude odblokovaný.



Obr. 4.3: Príklad vytvorenia používateľom navrhnutej komunikácie

4.2 Správca pamäte

Správca pamäte bude viesť informácie o využití pamäte programu, pamäte dát a pamäte procesov.

Pamäť programov, dát a premenných bude realizovaná ako spájaný zoznam úsekov. Zoznam bude obsahovať informáciu o nasledujúcom úseku, veľkosti úseku a obsadenosti úseku. Na základe volania malloc() bude správca pamäte prideľovať pamäť na úrovni premenných procesu alebo jadra. Na základe volania tmalloc() bude správca prideľovať pamäť na úrovni procesov. Na základe volania pmalloc() bude správca prideľovať pamäť programom. Vkladanie programu do pamäte bude realizované pomocou nato určeného zavádzacieho procesu.

Správca pamäte bude prideľovať pamäť buď novému procesu alebo premennej procesu. Preto správca bude poznať minimálne informácie o voľných úsekoch pamäte dát nepridelenej žiadnemu procesu. Na žiadosť o vytvorenie nového procesu (tmalloc()) správca vyhľadá vhodný úsek pamäte, ktorý vyhradí pre proces.

V prípade, že proces požaduje alokovať pamäťový priestor pre svoju premennú(malloc()), správca pamäte bude prideľovať pamäť dvoma spôsobmi, podľa toho ako bude realizovaná pamäť dát procesu. Ako bolo spomenuté každý proces má mať pridelenú statickú a dynamickú pamäť dát. Statická pamäť dát bude v procesoch bežiacich pod MOS vylúčená, ak procesor nebude mať bázové registre. Dynamická pamäť je zložená zo zásobníka a haldy. Zásobník je pamäť, ktorá je obsadzovaná bez pričinenia programátora, napríklad volaním funkcie. Halda je pamäť, ktorá sa obsadzuje s pričinením programátora, napríklad alokovanie

priestoru pre ukazovateľ.

Z tohoto dôvodu je možné uvažovať dve metódy realizácie pridelenia pamäťového priestoru premennej procesu:

- Buď bude halda spoločnou pre všetky procesy, a teda správcovi budú stačiť dva zoznamy voľných úsekov pamäte. Jeden pre výber pamäte pre proces s menšou granularitou veľkosti úseku. Druhý pre výber pamäte pre premennú s väčšou granularitou veľkosti úseku, alebo

- každý proces bude mať svoju vlastnú haldu vo svojom pamäťovom priestore. Súčasťou hlavičky procesu teda bude aj ukazovateľ na zoznam voľných úsekov pamäte. Správca pamäte bude prideľovať pamäť na základe priloženého zoznamu voľných úsekov pamäte z hlavičky procesu.

4.3 Správca V/V zariadení

Správca V/V zariadení bude spravovať každé alebo časť zariadení na procesore, ktoré komunikujú s okolím alebo sú internou súčasťou procesora (časovač). Správca V/V zariadení bude tvorený množinou blokujúcich a neblokujúcich volaní, prostredníctvom ktorých bude možné pristúpiť k požadovanému zariadeniu. Vybrané zariadenia budú obalené platformovo závislou prezentačnou vrstvou realizovanou v jazyku symbolických inštrukcií, s ktorou bude komunikovať konkrétne volanie. Správca V/V zariadení bude vykonávať aj správu prerušení generovaných zariadeniami.

Namiesto volaní bude môcť MOS riadiť zariadenia pomocou *MOS procesov*. Ak chce používateľský proces komunikovať so zariadením, musí sa registrovať ako odosielateľ na konkrétnom *MOS procese*. Naopak ak chce zariadenie komunikovať s konkrétnym používateľským procesom, *MOS proces* sa musí registrovať na používateľskom procese ako odosielateľ. Správca V/V zariadení bude v takomto prípade tvoriť vrstvu nad zariadeniami, ktorá je platformovo závislá. S touto vrstvou budú pracovať MOS procesy, ktoré budú čiastočne platformovo závislé.

4.4 Riadenie spotreby energie

Riadenie spotreby energie bude priamou súčasťou modulov správy V/V zariadení a riadenia procesov. Riadenie spotreby je platformovo závislé a zväčša bude implementované v jazyku symbolických inštrukcií. Správa V/V zariadení bude vypínať respektíve uvádzať do úsporného režimu zariadenia v prípade, že nie sú potrebné. Riadenie procesov bude uvádzať do úsporného režimu procesor v prípade nízkeho alebo žiadneho počtu spustených procesov.

4.5 Riadenie prístupu

Moduly MOS alebo používateľské procesy, ktoré potrebujú aby prístup do niektorých zariadený alebo úsekov kódu bol chránený pred vstupom viacerých procesov, budú môcť použiť služby riadenia prístupu pomocou semaforov a zámok. Riadenie prístupu bude platformovo závislé. Ak procesor má podporu inštrukcie kontroly a zápisu premennej (TSL), tak riadenie prístupu bude navrhnuté s použitím tejto inštrukcie. V prípade, že procesor takúto inštrukciu nepodporuje, tak kontrola a nastavenie premennej bude chránené zakázaním a po vykonaní opätovným povolením prerušenia.

Pri pokuse o vstup do kritickej oblasti, ktorá je už obsadená bude žiadateľ o vstup pozastavený alebo blokovaný. Blokovanie respektíve pozastavenie žiadateľa bude závisieť od výberu typu volania.

Výstup z kritickej oblasti jednoducho otvorí zámku respektíve zníži hodnotu semafora.

4.6 Periféria Operačného systému

Okrem jadra operačného systému bude operačný systém implementovať najmä *MOS procesy*, ktoré, ako bolo spomenuté, budú obaľovať pripojené zariadenia procesora.

Na periférii operačného systému bude implementovaný aj mechanizmus zavádzania a aktualizácie programov. Táto možnosť je vhodná najmä, ak je nutné udržať operačný systém v chode bez potreby jeho opätovnej inicializácie. Mnohé operačné systémy na takéto možnosti neprihliadajú, čo brzdí ich použitie v plných prevádzkach. Aktualizáciou programu sa bude rozumieť výmena programov, ktoré už sú zavedené v programovej pamäti. Zavedenie úplne nového programu bude jednoduchšou verziou aktualizácie, kedy sa len na voľné miesto v pamäti programov umiestni nový program.

Žiadosť o zavedenie nového programu spustí zavádzací proces, ktorý sa plánuje na procesor. Proces bude žiadať správcu pamäte o pridelenie miesta pre nový program. V prípade, že miesto nie je, je žiadosť zamietnutá a proces zavádzač končí. V prípade kladnej odozvy správcu pamäte je nový program uložený do pamäte dát a je vytvorený nový záznam do tabuľky programov.

Žiadosť o aktualizáciu programu spustí zavádzací proces, ktorý požiada o výmenu programu s daným identifikačným číslom. Proces, rovnako ako pri zavedení nového programu, vykoná zápis programu do pamäte programov. Pôvodný program bude mať v hlavičke programu nastavený príznak, ktorý odkazuje na záznam v hlavičke nového programu, čím sa zabezpečí spúšťanie nových procesov pod novou verziou programu. Ak pôvodný program už nemá žiadnu spustenú inštanciu, tak sa jeho záznam v tabuľke programov prepíše novým záznamom a jeho priestor v pamäti programov sa uvoľní.

Možnosť zavádzania do pamäte programov bude umožnená len, ak procesor podporuje

zápis do pamäte programu za behu systému.

4.7 Výber platformy

Modulárny operačný systém bude testovaný na procesore rodiny ARM7 prípadne aj na procesore z rodiny Atmel AVR.

Z kategórie procesorov ARM7 bude použitá vývojová doska Sam7-p256 od firmy Olimex [8] s procesorom AT91SAM7S256 [9]. Vývojový kit má 256 kB pamäte programu na čipe typu flash a 64 kB pamäte dát na čipe typu ram. Vývojová doska má vyvedené dve sériové rozhrania, má možnosť vyvedenia sériového rozhrania na výpis ladiacich správ, USB 2.0, 10 bitový analógovo-digitálny prevodník, I2C rozhranie, SPI rozhranie, 3x časovač, 4x PWM, watchdog WDT, DMA (priami prístup k pamäti pre všetky zariadenia), dve tlačidlá, konektor na SD kartu. Možnosť práce v 60 MHz frekvencii. Procesor má vyvedené všetky piny na doske. Programovanie a ladenie prostredníctvom JTAG.

5 Implementácia

Táto kapitola prezentuje implementované časti MOS na platforme ARM7tdmi. V čase publikovania tejto práce sú implementované a funkčne otestované dátové štruktúry spájaný zoznam a rad. Implementovaný a funkčne otestovaný je modul správy pamäte, plánovania úloh, medziprocesnej komunikácie, manažmentu procesov, riadenia prístupu a štatistiky. Implementované sú platformovo závislé časti operačného systému. MOS má rozvrstvený zdrojový kód podľa modulov do samostatných hlavičkových a zdrojových súborov. Platformovo nezávislé časti MOS sú implementované v programovacom jazyku C. Platformovo závislé časti MOS sú implementované z časti v programovacom jazyku C a z časti v jazyku symbolických inštrukcií.

5.1 Štruktúra zdrojových kódov MOS

Vlastnosti MOS sa nastavujú prostredníctvom konfiguračného súboru *config.h*. V tomto súbore je zoznam konfiguračných premenných riadiacich spôsob kompilácie MOS. Platformovo závislé definície typov premenných sú uložené v hlavičkovom súbore *port.h*. Platformovo závislé časti zdrojového kódu sú sústredené v zdrojových súboroch *port.c, main.c* a *asmport.S*.

V súbore *port.c* sú definované funkcie inicializujúce časovanie prepnutia úlohy, funkcie riadiace vstup a výstup z kritickej oblasti a pomocné funkcie súvisiace so štatistickým meraním.

V súbore *main.c* sa nachádza funkcia main, v ktorej sa inicializujú jednotlivé moduly MOS. V tomto súbore je implementovaná spúšťacia sekvencia MOS.

V súbore *asmport.S* sú uložené obálky volaní MOS, ktoré vyžadujú prepnutie do chráneného módu jadra. Nachádza sa tu platformovo závislá inicializácia spustenia MOS a odloženie a načítanie kontextu počas prepnutia úlohy.

Platformovo nezávislé časti MOS sú sústredené v súboroch *ipc.c, taskMNG.c, memMNG.c, list.c, queue.c* a *mos.c*. V súbore *ipc.c* sú funkcie riadiace medziprocesnú komunikáciu.

V súbore *taskMNG.c* sú funkcie súvisiace s inicializáciou MOS, plánovaním, vytváraním, ukončovaním a prepínaním úloh. Sú tu aj funkcie súvisiace so štatistickým meraním.

V súbore *memMNG.c* sú funkcie súvisiace s prideľovaním pamäte procesom a premenným. Sú tu aj funkcie súvisiace so štatistickým meraním.

V súboroch *list.c* a *queue.c* sú funkcie úzko súvisiace so štruktúrou spájaný zoznam respektíve štruktúrou rad.

V súbore *mos.c* je uložená definícia koreňového procesu MOS. Do tohto súboru môžu používatelia vkladať svoje definície procesov a definovať pracovnú náplň koreňového procesu.

Okrem spomínaných súborov sú súčasťou zdrojových kódov MOS aj súbory *CStartup.S* a *Cstartup_SAM7.c*. Tieto súbory sú prevzatými súbormi od spoločnosti Atmel. Slúžia na nízkoúrovňovú inicializáciu platformy ARM7tdmi.

5.2 Dátové štruktúry spájaný zoznam a rad

Spájaný zoznam bol implementovaný, tak ako bol navrhnutý v kapitole 4. Návrh (obr. 4.1). Zoznam je realizovaný ako štruktúra obsahujúca záznam o aktuálnej veľkosti zoznamu a terminátor, ktorý je typu *položka*. Štruktúra *položka* obsahuje záznam atribútu, ukazovateľ na obsah položky typu void, ukazovatele na predchádzajúcu a nasledujúcu položku typu *položka*. Zoznam nemá implementované blokovanie procesov. Nad spájaným zoznamom sú implementované operácie:

- L_init(zoznam) má na vstupe ukazovateľ na štruktúru zoznamu. Táto funkcia inicializuje veľkosť zoznamu na "0". Ukazovatele na nasledujúcu a predchádzajúcu položku v terminátore nastaví tak, aby ukazovali na terminátor.
- L_push(zoznam, položka), L_insert(zoznam, položka) riadia vkladanie do zoznamu. Funkcia L_push() vloží položku na koniec zoznamu. Funkcia L_insert() vloží položku do zoznamu na základe hodnoty atribútu. Po vložení položky do zoznamu vracajú funkcie ukazovateľ na vkladanú položku. Ak je zoznam plný, funkcie vracajú hodnotu NULL.
- L_pop(zoznam) vyberie zo zoznamu prvú položku a jej hodnotu vráti na výstup. Ak je zoznam prázdny tak vráti hodnotu NULL.
- L_isFull(zoznam), L_isEmpty(zoznam) sú makrami a vracajú "1" ak je zoznam plný resp. prázdny a "0" ak nie je zoznam plný resp. prázdny.
- L_first(zoznam), L_last(zoznam) sú makrami a vracajú ukazovateľ na prvú resp. poslednú položku, na ktorú ukazuje terminátor.
- L_search(zoznam, dáta), L_searchAtt(zoznam, atribút) vyhľadávajú položku na základe hodnoty ukazovateľa na dáta resp. na základe hodnoty atribútu. Použitie týchto funkcií bude ešte predmetom rozhodovania.
- L_delete(položka) danú položku, nájdenú funkciou L_search(), odpojí z radu. Táto funkcia bude tiež predmetom skúmania jej použiteľnosti.

Rad bol implementovaný, tak ako bol navrhnutý v kapitole 4. Návrh (obr. 4.2). Rad je realizovaný ako štruktúra obsahujúca záznam o aktuálnej veľkosti radu, položku konca

a položku začiatku radu. Položka obsahuje záznam atribút, ukazovateľ na prenášané dáta typu void, ukazovateľ na nasledujúcu položku v rade. Rad nemá implementované blokovanie úloh. Nad radom sú implementované operácie:

- Q_init(rad) má na vstupe ukazovateľ na štruktúru radu. Táto funkcia inicializuje veľkosť radu na "0" a položku začiatku radu nastaví tak aby ukazovala na položku konca a naopak.

- Q_push(rad, položka) riadi vkladanie do radu. Funkcia vloží položku na koniec radu. Po vložení položky do radu funkcia vracia ukazovateľ na vkladanú položku. V prípade, ak je rad plný, funkcia vracia hodnotu NULL.

- Q_pop(rad) vyberie prvú položku z radu a vráti ju na výstupe. Ak je rad prázdny tak vráti hodnotu NULL.

- Q_isFull(rad), Q_isEmpty(rad) sú makrami a vracajú "1" ak je rad plný resp. prázdny a "0" ak nie je rad plný resp. prázdny.

- Q_first(rad), Q_last(rad) sú makrami a vracajú ukazovateľ na prvú resp. poslednú položku, na ktorú ukazuje začiatok resp. koniec.

5.3 Správca pamäte

Správca pamäte bol implementovaný podľa návrhu. Správca pamäte spravuje:

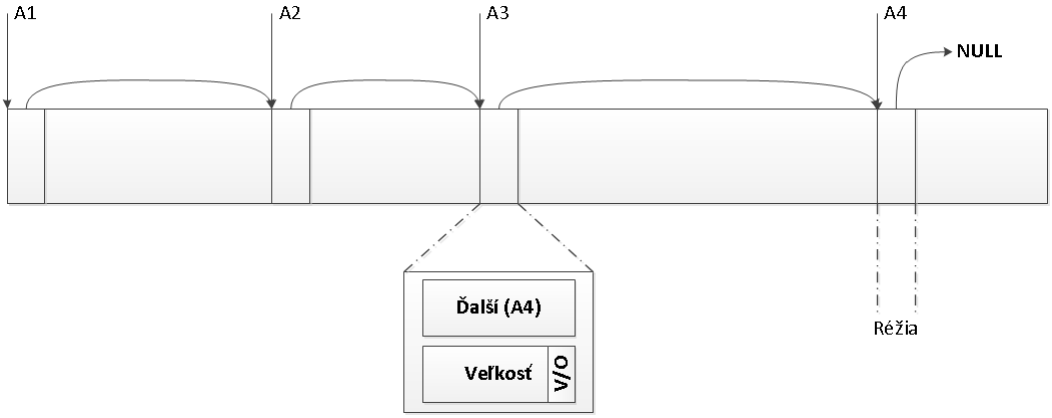
- Zásobník jadra (stack). Priestor pre odkladanie premenných jadra. Jeho veľkosť sa nastavuje v konfiguračnom súbore. Začiatok a koniec zásobníka je uložený v statických premenných, ktoré sú naplnené počas inicializácie správcu.

- Haldu jadra (heap). Priestor na alokovanie pamäte premenným jadra (napr. hlavičky úloh a procesov). Ak je MOS nastavený tak, že procesy nemajú svoju internú haldu, je tu alokovaný priestor aj pre premenné procesov. Počiatok haldy jadra je uložený v statickej premennej a je nastavený počas inicializácie správcu pamäte.

- Pamäť procesov. Vymedzuje priestor na alokovanie pamäte pre spúšťané procesy. Počiatok pamäte procesov je uložený v statickej premennej a je nastavený počas inicializácie správcu pamäte.

- Halda procesu. Vymedzuje priestor pre alokovanie premenných procesov. Halda procesu sa generuje počas vytvorenia procesu len vtedy, ak je konfigurovaná v konfiguračnom súbore.

Prideľovanie pamäte je realizované pomocou metódy *prvý vyhovujúci* (first fit). Základným kameňom metódy je udržiavanie zoznamu úsekov pamäte. Úsek je štruktúra, ktorá má svoju adresu určenú pozíciou v pamäti, má ukazovateľ na ďalší úsek, má informáciu o dĺžke úseku a má informáciu o stave úseku. Informácie o stave a veľkosti úseku sú v našej implementácii zlúčené do jednej položky. Požadovaná veľkosť pridelenej pamäte je zarovnaná na 4B (32-bitové systémy). Z tohto dôvodu sú dva najmenej významné bity veľkosti pridelenej pamäte vždy nulové. Najmenej významný byt je preto použitý ako bit indikujúci obsadenosť úseku. Ak je bit nulový, úsek je voľný, a ak je bit jednotkový, tak úsek je obsadený. Po definovanom úseku nasleduje pamäť, ktorá je pod jeho správou (Obr. 5.1).



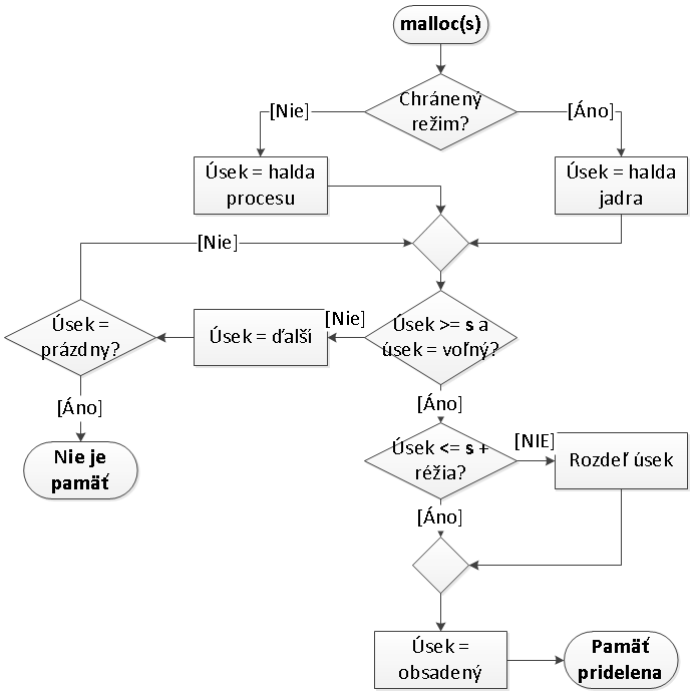
Obr. 5.1: Zoznam úsekov pamäte a štruktúra úsek

Počas inicializácie sa podľa konfiguračného súboru nastaví pamäť haldy a pamäť procesov. Obe pamäte spočiatku obsahujú jediný úsek, ktorý nemá suseda a jeho veľkosť je nastavená na konfigurovanú veľkosť zmenšenú o réžiu úseku. Rovnaký postup je vykonaný po pridelení pamäte pre haldu procesu.

Správca pamäte disponuje tromi funkciami, ktoré spravujú pamäť. Sú to funkcie:

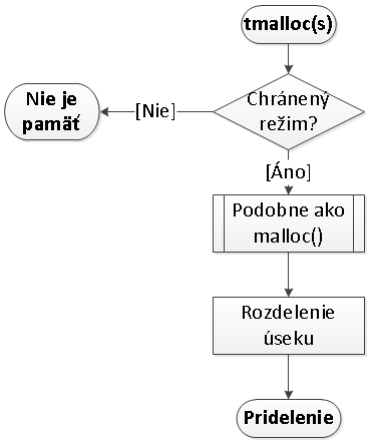
- malloc(veľkosť),
- tmalloc(veľkosť) a
- free(premenná).

Funkcia *malloc()* prideľuje pamäť pre premenné procesov aj jadra. Ak je procesor v privilegovanom režime, tak sa prideľuje pamäť v halde jadra, inak je prideľovaná pamäť v halde procesu. Počas prideľovania pamäte sa najprv hľadá úsek rovnaký alebo väčší ako je požadovaná veľkosť (veľkosť je zaokrúhlená nahor na násobok štyroch bajtov). Ak sa taký úsek nájde a je väčší ako požadovaná veľkosť zväčšený o réžiu, daný úsek sa rozdelí na dva. Prvý bude mať požadovanú veľkosť a druhý zostávajúcu veľkosť. Následne sa daný úsek nastaví na obsadený (obr. 5.2).



Obr. 5.2: Priebeh vykonania funkcie *malloc()*

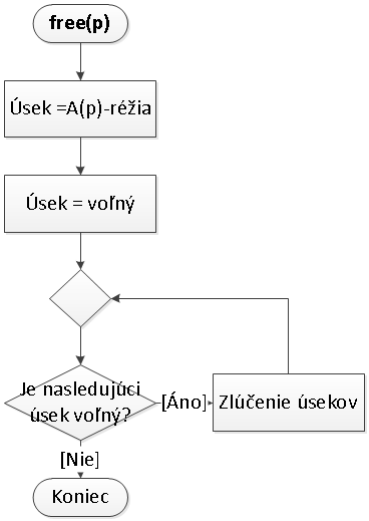
Funkcia *tmalloc()* prideľuje pamäť pre nové procesy. Funkcia sa správne vykoná len ak je procesor v chránenom režime. *Tmalloc()* funguje na rovnakom princípe ako funkcia *malloc()*. Rozdiel je v zaokrúhlení (nahor) požadovanej veľkosti pamäte na konfigurovateľný násobok (napr. 256 B) a v pridelení úseku. Už vybraný úsek sa pred pridelením rozdelí na dva skoro rovnako veľké úseky. Prvý úsek sa stáva haldou, ktorá bude o réžiu menšia ako druhý úsek, ktorý sa stáva zásobníkom (obr. 5.3).



Obr. 5.3: Priebeh vykonania funkcie *tmalloc()*

Funkcia *free()* uvoľňuje pridelenú pamäť. Funkcia funguje rovnako pre všetky druhy pamätí MOS. Vstup funkcie je premenná. Adresa úseku pridelenej pamäte sa vypočíta ako

rozdiel adresy pridelenej vstupnej premennej a réžie úseku. Po identifikovaní úseku sa daný úsek uvoľní. Po uvoľnení úseku sa zisťuje, či nasledujúci úsek nie je tiež voľný. Ak je voľný, tak dané úseky sa zlúčia do jedného. Táto operácia sa opakuje, kým sa nenájde prvý obsadený úsek alebo sa neskončí pamäť. Takýmto riešením sa čiastočne zamedzuje fragmentácii pamäte (obr. 5.4).



Obr. 5.4: Priebeh vykonania funkcie *free()*

5.4 Riadenie procesov

V časti riadenie procesov sme implementovali modul Plánovač, Manažér úloh a Riadenie komunikácie.

5.4.1 Plánovač

Plánovanie úloh bolo implementované troma rôznymi metódami. Zdrojový kód plánovača je v súbore taskMNG.c. Spoločným prvkom všetkých troch druhov plánovača je staticky uložený spájaný zoznam (v prípade viacúrovňového plánovača pole spájaných zoznamov), do ktorého sa ukladajú pripravené úlohy. Tento zoznam je viditeľný len v súbore taskMNG.c. Používateľ nevie priamo zasiahnuť do procesu plánovania. Spájaný zoznam je inicializovaný počas inicializácie MOS. Plánovanie riadia tri funkcie:

- funkcia *SCH_init()* inicializuje spájaný zoznam.
- funkcia *schedule()* vkladá úlohu do spájaného zoznamu. Ak je zoznam plný, vracia chybovú hodnotu.

- funkcia *getTask()* vyberá úlohu zo zoznamu. Ak je zoznam prázdny, vracia chybovú hodnotu.

Plánovanie *Round Robin*

Tento druh plánovania používa vyššie uvedenú štruktúru spájaný zoznam na ukladanie úloh pripravených a čakajúcich na pridelenie procesorového času. Úlohy sú vkladané do zoznamu podľa poradia v akom prichádzajú, princípom prvá dnu prvá von (FIFO).

Prioritné plánovanie

V prioritnom plánovaní sa používa predom definovaná priorita úlohy. Priorita úlohy je určená počas vytvorenia úlohy. Úlohy sú vkladané do zoznamu podľa výšky priority. Prvou sa stane úloha s najvyššou prioritou.

Viac úrovňové plánovanie

Viacúrovňové plánovanie je založené na princípe ukladania úloh s prioritou, spadajúcou do rovnakej triedy, do samostatného spájaného zoznamu. Naše prioritné plánovanie vytvára $n = \frac{k}{2}$ spájaných zoznamov, kde k je najvyššia možná priorita.

Úloha je vložená do radu $n = \frac{k}{2}$, kde k je priorita úlohy. Výhodou takéhoto riešenia je rýchla realizácia delenia dvomi pomocou bytového posunu doprava. Úloha je vkladaná do vybraného spájaného zoznamu vždy na koniec.

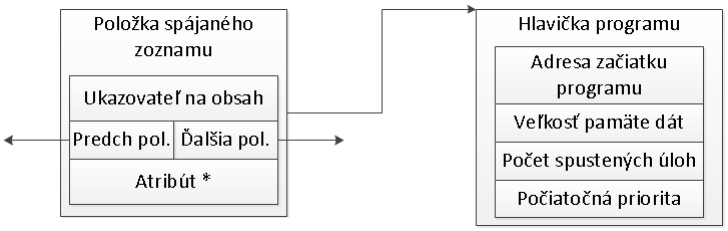
Pri výbere úloh z plánovača sa postupne prechádza od prioritne najvyššieho spájaného zoznamu po prioritne najnižší.

5.4.2 Manažér úloh

Úlohou modulu Manažér úloh je spravovať hlavičky uložených programov (PCB), hlavičky spustených procesov (TCB) a aktuálne bežiacu úlohu (running). Hlavičky sa sústredia do dvoch samostatných spájaných zoznamov. Sú to zoznamy Tabuľka hlavičiek programov (PCBT) a Tabuľka hlavičiek procesov (TCBT). Spájané zoznamy sú statickými premennými MOS, ktoré sa nastavujú počas inicializácie MOS. Manažér úloh poskytuje rozhranie pre vkladanie nových hlavičiek programov a spúšťanie, prepínanie a ukončovanie procesov. Aktuálne bežiaca úloha je uložená v statickej premennej.

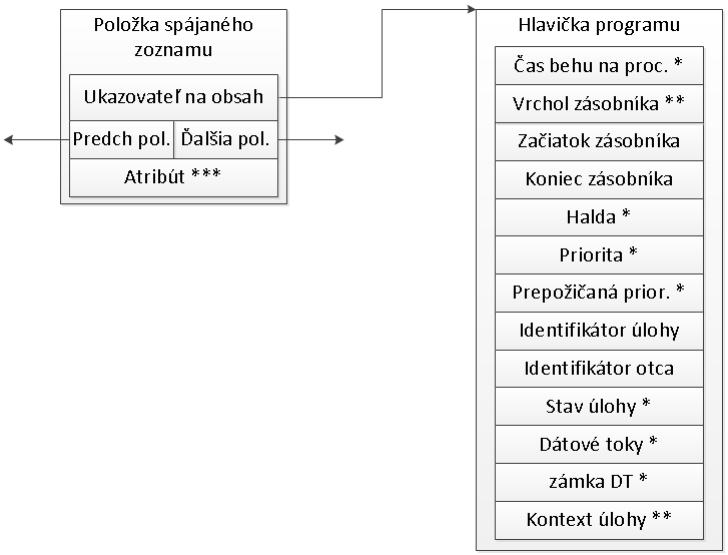
Hlavička programu je štruktúra, ktorá udržuje informácie o uloženom programe v programovej pamäti procesora. Každá hlavička programu spolu so štruktúrou položky, ktorá ju zapuzdruje, má svoj alokovaný priestor v halde jadra. Na obrázku 5.5 je zobrazená štruktúra hlavičky programu spolu so zapuzdrujucou položkou zoznamu. Atribút položky (označený *) nadobúda počas inicializácie hlavičky hodnotu identifikátora programu. Identifikátor jednoznačne identifikuje každý program. Adresa začiatku programu odkazuje na počiatočnú

adresu programu v pamäti programov. Veľkosť pamäte dát určuje, aký veľký úsek pamäte z pamäte procesov bude pridelený novej inštancii programu. Počet spustených úloh udáva aktuálny počet bežiacich procesov, ktoré sú inštanciami daného programu. Počiatočná priorita určuje akú bude mať proces prioritu po jeho vytvorení.



Obr. 5.5: Štruktúra PCB a zapuzdrujúca položka zoznamu

Hlavička procesu je štruktúra, ktorá udržuje informácie o spustenom procese. Každá hlavička procesu spolu so štruktúrou položky, ktorá ju zapuzdruje, má svoj alokovaný priestor v halde jadra. Na obrázku 5.6 je zobrazená štruktúra hlavičky programu spolu so zapuzdrujucou položkou zoznamu. Prvky štruktúry označené * sú voliteľnými. Ich prítomnosť v štruktúre závisí od konfigurácie MOS. Prvky označené ** sa navzájom vylučujú. Podľa konfigurácie sa v hlavičke nachádza buď kontext úlohy alebo len vrchol zásobníka, v ktorom je odložený kontext. Atribút položky (označený ***) nadobúda počas inicializácie hlavičky hodnotu priority, ak je nakonfigurovaná. Stav úlohy uvádza aktuálny stav, v ktorom sa úloha nachádza (pripravená, pozastavená, ukončená...). Prvok Dátové toky odkazuje na spájaný zoznam, v ktorom sa ukladajú žiadosti o komunikáciu. Zámka indikuje, či je alebo nie je obsadená kritická oblasť podávania žiadostí.



Obr. 5.6: Štruktúra TCB a zapuzdrujúca položka zoznamu

Skryté funkcie

Modul používa pre používateľa neprístupné funkcie *MOSP_init()* a *initContext()*. Prvá funkcia slúži na inicializovanie koreňového procesu MOS. Druhá funkcia inicializuje kontext každej novej úlohy.

Verejné volania

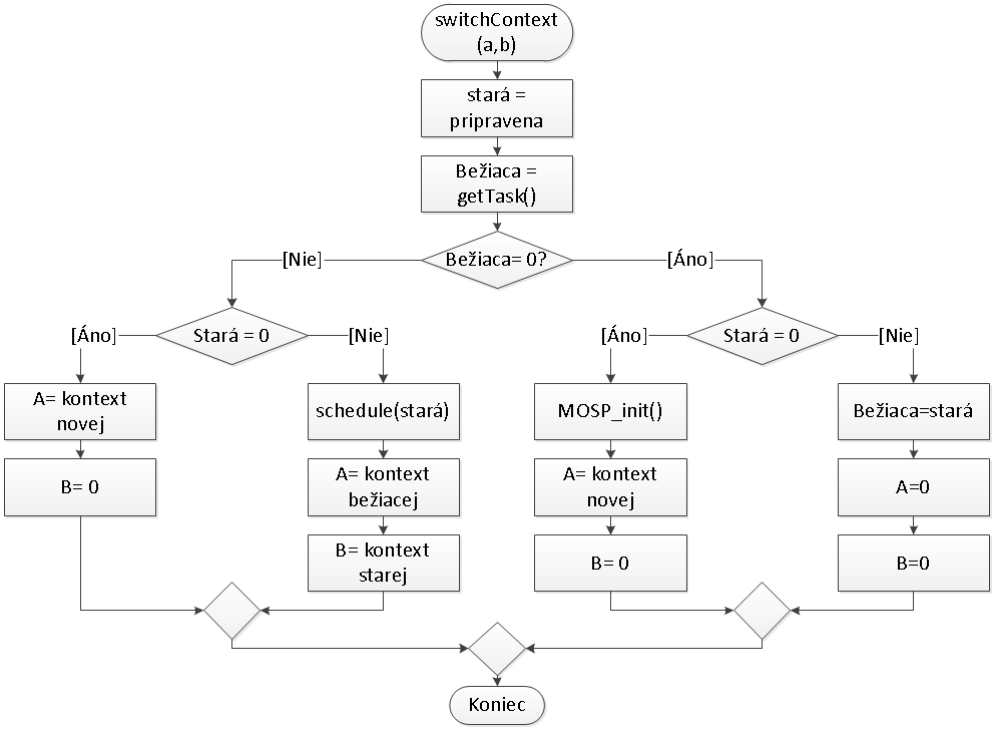
Modul ponúka pre používateľa volania: *TMNG_initStreams(), TMNG_pushStream(), TMNG-_popStream(), TMNG_taskPid(), TMNG_taskHeap()* a *TMNG_taskPriority()*. Tieto volania nevyžadujú prepnutie do chráneného režimu, a preto sa vykonávajú v kontexte procesu, ktorý ich volal.

- TMNG_initStreams() inicializuje spájaný zoznam, v ktorom sa ukladajú položky žiadostí o komunikáciu. Tiež inicializuje zámku, ktorá riadi vstup do kritických oblastí.
- TMNG_pushStream(pid, dátový tok) slúži na vloženie žiadosti do spájaného zoznamu. Pid určuje, ktorej úlohe sa žiadosť podáva. Dátový tok je ukazovateľom na žiadosť. Ak daná úloha neexistuje, spájaný zoznam je plný alebo sa nepodarilo vstúpiť do KO, volanie vracia chybovú hodnotu.
- TMNG_popStream() vyberá zo zoznamu žiadostí adresovaných danej úlohe jednu žiadosť. V prípade, že nebol inicializovaný spájaný zoznam žiadostí, vracia chybovú hodnotu. Ak iná úloha už obsadila kritickú oblasť, vracia chybovú hodnotu. Ak je daný zoznam prázdny, vracia chybovú hodnotu.
- TMNG_taskPid() vracia identifikátor práve bežiacej úlohy.
- TMNG_taskHeap() vracia odkaz na haldu práve bežiacej úlohy. Využíva sa v alokovaní pamäte pre premennú procesu.

Skryté volania

MOS využíva volania, ktoré vyžadujú prepnutie procesora do chráneného režimu ale nemajú obálku volania (nie sú prístupné používateľovi). Volanie *_TMNG_init()* inicializuje TCBT (tabuľku hlavičiek programov), PCBT (tabuľku hlavičiek procesov) a nastaví bežiacu úlohu.

Volanie *switchContext(nová, stará)* je prvkom, ktorý zabezpečuje odloženie aktuálne bežiacej úlohy do zoznamu pripravených úloh a načítanie novej úlohy zo zoznamu pripravených úloh. Na obrázku 5.7 je zobrazený priebeh vykonania volania. Na vstupe volania sú umiestnené ukazovatele na miesto v pamäti, kde bude po ukončení volania umiestená výsledná adresa kontextov starej a novej úlohy. S týmto výsledkom pracuje funkcia odkladajúca kontext. Ak aktuálne bežiaca úloha je v stave pozastavená, tak sa neplánuje a načíta sa len nová úloha z pripravených úloh.



Obr. 5.7: Volanie switchTask()

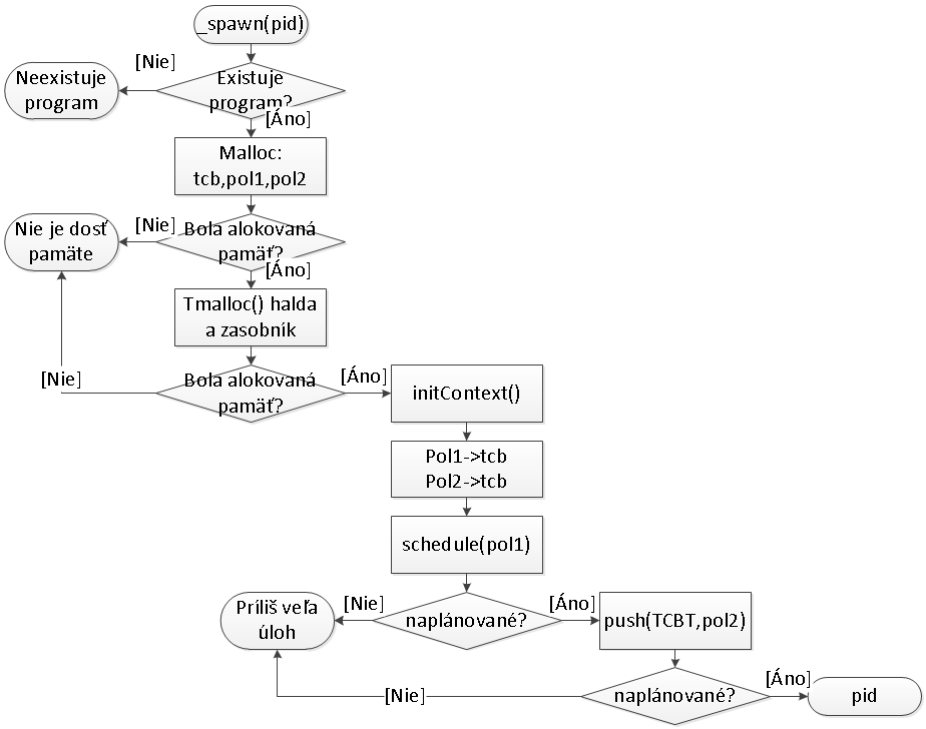
Chránené volania

Modul poskytuje aj volania, ktoré vyžadujú prepnutie procesora do chráneného režimu ale sú prístupné pomocou obálky volania používateľovi. Sú to volania *_spawn(), _taskEnd()* a *_newProgram()*.

Volanie *_newProgram()* vykoná vytvorenie hlavičky pre už zavedený program. Volanie je implementované tak, aby nebolo možné vytvoriť dve hlavičky programu s rovnakým identifikátorom programu. Nie je implementovaný zavádzač, preto nevznikla potreba mať rovnaké identifikátory. Po overení, že neexistuje daný program sa alokuje priestor pre hlavičku a položku zoznamu, ktorá ju zapúzdri. Ak nastane problém s nedostatkom pamäte, volanie skončí neúspechom. Po úspešnom vložení programu sa hlavička zapúzdri do položky zoznamu, tá sa vloží do PCBT. Ak nie je dosť miesta v PCBT, volanie končí neúspechom.

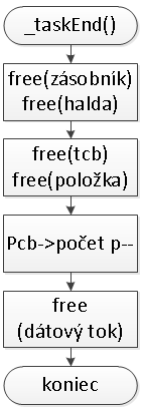
Volanie *_spawn(pid)* vykoná vytvorenie a naplánovanie novej úlohy v MOS. Vstupom volania je Pid programu, z ktorého má byť vytvorená inštancia. Volanie overí, či program daného pid existuje, ak neexistuje volanie končí s chybou. Nasleduje vytvorenie hlavičky procesu a vytvorenie položiek pre TCBT a zoznam pripravených úloh. Ak nie je dostatok pamäte pre vytvárané, prvky volanie skončí s chybou. Po vytvorení položiek sa inicializuje hlavička procesu. Po inicializácii hlavičky sa zapúzdri do položky a vloží sa do zoznamu pripravených úloh. Ak je zoznam plný, tak volanie končí s chybou. Hlavička sa tiež zapúzdri do druhej položky a vloží do TCBT. Ak tu nastane problém s plnosťou TCBT volanie skončí

s chybou. Ak všetko prebehne bez problémov, volanie vráti identifikátor novej úlohy (obr. 5.8).



Obr. 5.8: Volanie _spawn()

Volanie *_taskEnd()* vykoná bezpečné ukončenie úlohy. Počas výkonu volania sa uvoľní celá pamäť alokovaná pre proces a alokovaná v halde jadra. Ak je MOS skompilovaný aj so štatistickým modulom, tak sa hlavička v TCBT neuvoľňuje aby sa ponechali informácie o danej úlohe (obr. 5.9).



Obr. 5.9: Volanie _taskEnd()

5.4.3 Riadenie komunikácie

Riadenie komunikácie je implementované ako spomínaný dátový tok. Dátový tok je štruktúra, ktorá obsahuje ukazovateľ na rad, do ktorého sa budú posielať dáta, identifikátor procesu, ktorý vystupuje ako odosielateľ, stav dátového toku, položky na odloženie blokovaného odosielateľa a príjemcu a zámok. Nad dátovým tokom sú implementované operácie:

- senderConnect(PID) na základe identifikátora úlohy vytvorí žiadosť o spojenie a vloží ju do zoznamu čakajúcich žiadostí. Funkcia vracia *handle*, ktorý je typu void*. V prípade ak je zoznam registrácií plný, tak funkcia vracia NULL. Pridávanie do zoznamu registrácií je chránené zámkou, aby sa zabránilo násobnému prístupu odosielateľov.

- receiverConnect() vyberie registráciu zo zoznamu a inicializuje spojenie s odosielateľom. Ak je zoznam prázdny, funkcia vracia NULL. Odoberanie zo zoznamu je chránené rovnakou zámkou ako pri pridávaní do zoznamu.

- send(dáta, handle), sendAck(dáta, handle), notify(dáta, handle) slúžia na odosielanie dát príjemcovi. Funkcie send() a sendAck() sú blokujúce. Ak je rad správ plný, tak odosielateľ je pozastavený, kým sa v rade neuvoľní miesto. Funkcia sendAck() prikladá k správe aj mechanizmus potvrdenia prijatia správy. Funkcia nastaví atribút položky na nenulovú hodnotu a pozastaví odosielateľa, kým daná správa nie je potvrdená príjemcom. Funkcia notify() je neblokujúci ekvivalent funkcie send() a vracia chybovú hodnotu, ak je rad správ plný.

- receive(dáta, handle), read(dáta, handle) slúžia na prijímanie dát poslaných odosielateľom. Funkcia receive() je blokujúca, teda ak príjemca chce čítať z prázdneho radu, je pozastavený, kým do radu nepríde správa. Funkcia read() je neblokujúcim ekvivalentom funkcie receive(), ktorý vráti chybovú hodnotu, ak je rad prázdny. Obe funkcie v prípade prečítania správy s potvrdením odblokujú odosielateľa, čím je potvrdené prijatie správy.

- senderPid(handle) vráti na základe *handle* dátového toku identifikátor odosielateľa. Táto funkcia sa dá použiť, ak chce príjemca spätne odosielať dáta odosielateľovi vytvorením opačného dátového toku.

- receiverDisconnect(handle) spôsobí zmenu stavu toku dát na stav odpojený, čím odosielateľ už nemôže posielať ďalšie dáta. Táto funkcia vráti "1" ak sú v rade ešte nejaké dáta. Ak v rade nie sú žiadne dáta, tak priestor alokovaný radom sa uvoľní a funkcia vráti "0".

- streamDestroy(handle) uvoľní pamäťový priestor dátového toku.

Niektoré detaily riadenia komunikácie budú ešte prepracované. Problémom môže byť nadviazanie komunikácie, v ktorej bude iniciátor komunikácie príjemca.

5.5 Riadenie Prístupu

V riadení prístupu sme implementovali ochranu vstupu a výstupu do kritickej oblasti pomocou zámok. V prípade, že chce používateľ chrániť úsek kódu a s nim spojený prístup k dátam alebo prostriedkom, môže použiť volania *_beginCritical()* a *endCritical()*, ktoré sú platformovo závislé a nachádzajú sa v súbore *port.c*. Implementovali sme aj funkcionalitu pozastavenia úlohy pomocou volaní *suspendItem()* a *unSuspendItem()*, ktoré sa nachádzajú v súbore *taskMNG.c*.

Zámka je realizovaná ako ukazovateľ na hodnotu. Ak je hodnota nenulov,á zámka je otvorená, inak je zámka uzamknutá.

Volanie *_beginCritical(*zámka)* je zabalené volaním *beginCritical()*. Volanie sa teda nevykoná, ak nie je procesor v privilegovanom režime, ktorý je neprerušiteľný. Volanie testuje obsah ukazovateľa. Ak je obsah nulový (zamknutý), tak volanie vracia hodnotu jeden inak nastaví obsah na nulu a vracia nulu. Volanie *endCritical()* môže byť vykonané hocikedy. Nastaví zámku na hodnotu uvoľnená.

Volanie *suspendItem(**položka)* zabezpečí pozastavenie plánovania úlohy, kým nie je znovu rozhodnuté, že úloha môže byť znovu plánovaná. Počas vykonania volania sa zapúzdrujúca položka zoznamu odloží do priestoru, na ktorý ukazuje vstupná premenná. Nastaví sa stav úlohy v hlavičke na pozastavená. Po vykonaní týchto operácií volanie vkročí do cyklu, v ktorom čaká kým nepríde prepnutie úlohy. Podmienkou cyklu je stav úlohy rovný pozastavený.

Volanie *unSuspendItem(**položka)* zabezpečí zmenu stavu úlohy naspäť do stavu pripravená. Na vstupe volania je odkaz na položku, ktorá bola odložená volaním *suspendItem()*. Volanie danú položku naplánuje pomocou volania *schedule()* a skončí.

Po prepnutí kontextu do úlohy, ktorá bola predtým pozastavená sa nastaví programové počítadlo na adresu vo volaní *suspendItem()*. Tu naposledy úloha čakala, kým nenastane prepnutie v cykle. Podmienka cyklu sa nesplní, pretože sa zmenil stav úlohy z *pozastavená* na *pripravená*, a teda volanie *suspendItem()* skončí a úloha môže pokračovať v štandardnom vykonávaní.

5.5.1 Obálka volania

Obálka volania zabezpečuje prístup k volaniu, ktoré vyžaduje aby procesor bol počas jeho výkonu v chránenom režime. Ak chce používateľ vo svojom procese spúšťať chránené volanie tak musí zavolať obálku volania, ktorá má rovnaký názov ako chránenie volanie. Napríklad ak

chceme vyvolať nový proces, tak zavoláme obálku *spawn(pid)*, v ktorom je zabezpečené prepnutie do chráneného režimu, vykonanie volania *_spawn(pid)* a prepnutie do používateľského režimu. Obálky sú platformovo závislé riešenie. Na platforme ARM7tdmi sú realizované pomocou inštrukcie *swi*.

5.6 Štatistický modul

Štatistický modul je rozložený do súborov *taskMNG.c, memMNG.c, port.c* a *asmport.s*. Jeho úlohou je posielanie informácií o stave jednotlivých meraných veličín na hostiteľský počítač. Implementovali sme posielanie dát prostredníctvom sériového rozhrania. Štatistický modul je implementovaný tak, aby mal minimálny vplyv na réžiu pri behu MOS v režime jadra. Modul pozostáva zo skupiny funkcií, ktoré sa volajú priamo z procesov. Našou myšlienkou bolo púšťanie týchto funkcií najmä z koreňového procesu. Drvivá väčšina funkcií je vykonávaná v používateľskom móde procesora, čím sa zabezpečuje malé ovplyvňovanie behu systému. Výhodou riešenia je, že štatistický modul posiela iba surové dáta, čím sa minimalizuje réžia na strane MOS. Výpočtová a prezenčná fáza sa realizuje v hostiteľskej aplikácií na hostiteľskom počítači.

5.6.1 Sériová komunikácia

Komunikačná časť štatistického modulu je umiestnená v súbore *port.c* pozostáva z troch funkcií:

- *DBGU_init()* inicializuje sériové rozhranie na strane Vnoreného systému (BaudRate 115200, bez parity, 1 stop bit, 8 dátových bitov, bez kontroly). Nastavené bolo ladiace sériové rozhranie procesora at91sam7s256.
- *DBGU_sendCmd(príkaz)* odosiela číslo príkazu záväzné pre hostiteľskú aplikáciu.
- *DBGU_send(*dáta,veľkosť)* odosiela získané dáta hostiteľskej aplikácii. Najprv sa odošle počet bajtov, koľko má aplikácia prijať, a potom sa odošlú po bajtoch dáta.

5.6.2 Odosielané dáta

Funkcie, ktoré realizujú prípravu a odosielanie dát, sú:

- *TMNG_statistics_PCBT()* realizuje odoslanie všetkých informácií z tabuľky programov.
- *TMNG_statistics_TCBT()* realizuje odoslanie všetkých informácií z tabuľky úloh. Ak je nastavený režim MOS s meraním štatistiky, tak sa hlavičky ukončených úloh neodstraňujú z tabuľky úloh. Súčasťou funkcie je aj odoslanie využitia haldy prostredníctvom funkcie *MMNG_statistics_taskHeap()*.

- *MMNG_statistics_kernelHeap(prikaz)* realizuje odoslanie stavu využitia haldy jadra alebo pamäte programov. Ak je príkaz odoslanie haldy, tak sú odosielané dáta: počet obsadených úsekov haldy a ich sumárna veľkosť, počet neobsadených úsekov haldy a ich sumárna veľkosť. Ak je príkaz odoslanie pamäte programu, tak sú odoslané dáta: počet obsadených úsekov pamäte programu a ich sumárna veľkosť, počet neobsadených úsekov pamäte programu a ich sumárna veľkosť.

- *MMNG_statistics_taskHeap(adresa)* realizuje rovnakú funkciu ako predchádzajúca funkcia s tým rozdielom, že realizuje výpočet na základe vstupnej adresy.

- *_MMNG_statistics_kernelStack()* realizuje odoslanie vrcholu, konca a začiatku zásobníka jadra. Táto funkcia je jedinou, ktorá musí byť vykonaná v chránenom režime jadra. Funkcia má svoju obálku volania.

Štatistický modul zasahuje do prepnutia kontextu tým, že ukladá do hlavičky úlohy aj počet prepnutí, počas ktorých daná úloha už beží. Tým sa zanáša malá réžia do prepnutia kontextu. Je to potrebné z dôvodu potreby merania času strávenom na procesore.

5.7 Platformovo závislé časti MOS

MOS sme implementovali na platforme ARM7tdmi konkrétne na procesore at91sam7s256. Tento procesor podporuje niekoľko režimov jadra, z ktorých sme využili *User mód, Fast Interrupt mód, Interrupt mód* a *Supervisor mód*. Ďalšie informácie o jednotlivých vlastnostiach procesora možno nájsť v katalógových listoch a manuáloch od výrobcu procesora [9, 10, 11].

5.7.1 Prepnutie do chráneného režimu

Prepnutie do chráneného režimu a s ním spojené obálky volaní sme realizovali pomocou inštrukcie *swi* (obr. 5.10). Táto inštrukcia, ktorej parameter je číslo volania, vyvolá prepnutie

```
.section .init, "ax"
    .global spawn
    .func   spawn

spawn:
        swi 0x01                //spustenie supervisor módu pre úlohu
        mov pc,lr               //návrat do procesu

    .size   spawn, . - spawn
    .endfunc
```

Obr. 5.10: Obálka volania spawn

do chráneného režimu. V chránenom režime sa vykonáva vyhľadanie volania podľa jeho čísla vo funkcií *SWI_Handler_Entry* (obr. 5.11). Ak sa nenájde volanie, chránený režim sa ukončí

bez vykonania akejkoľvek zmeny. Ak sa volanie nájde, tak sa vykoná a po jeho skončení sa znova procesor vráti do používateľského režimu. Počítadlo adresy inštrukcie je nastavené na adresu ďalšej inštrukcie po inštrukcii *swi*. Kód *SWI_Handler_Entry* sme prebrali a upravili podľa manuálu [11].

```
SWI_Handler_Entry:
    msr     cpsr,#0xd3
    STMFD   sp!, {r4,r5, lr}            // odloženie registrov a link registra

    LDR     r4, [lr, #-4]               // načítanie čísla poslednej inštrukcie SWI xy
    BIC     r4, r4, #0xff000000         // číslo volania ma len 24 bitov
    CMP     r4, #MAX_SWI                // overenie či dané volanie vobec existuje
    mov     lr, pc                      // nastav návrat sem
    LDRLS   pc, [pc, r4, LSL #2]        // skoč na volanie ak r4<MAX_SWI
    B       SwiEnd                      // ak to nie je volanie systému skoč na prepnutie ulohy

/* Jump-Table */
SwiTableStart:
    .word swi_dis_IF                    // 0
    .word _spawn                        // 1
    .word _taskEnd                      // 2
    .word _newProgram                   // 3
    .word _beginCritical                // 4
#ifdef EN_STATMODUL                     // ak je definovaný štatistický modul
    .word _MMNG_statistics_kernelStack  // 5
#endif
SwiTableEnd:
    .set MAX_SWI, ((SwiTableEnd-SwiTableStart)/4)-1

SwiEnd:
    ldmia SP!,{r4,r5, pc}^              // návrat na volanie
```

Obr. 5.11: Nájdenie konkrétneho volania

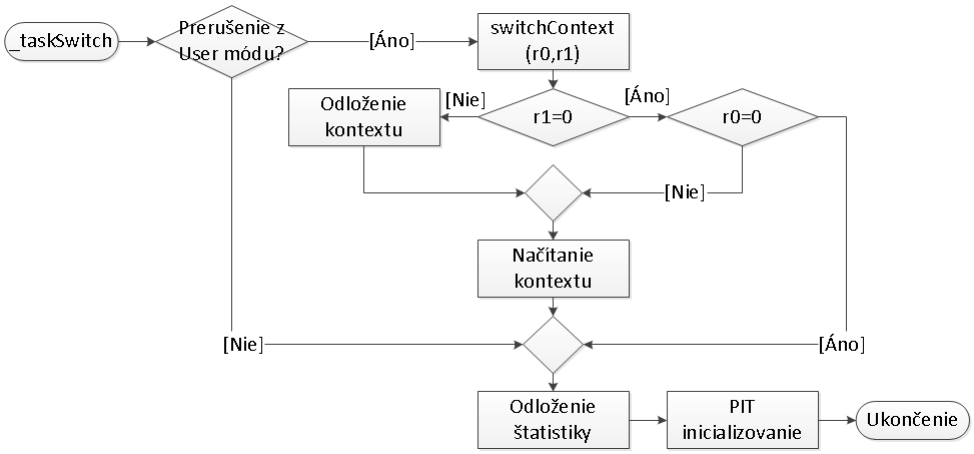
5.7.2 Prepnutie kontextu

MOS sme implementovali preemptívne, a preto na prepnutie úlohy potrebujeme používať nejaký spôsob časovania. Na časovanie prepnutia používame špeciálny časovač *Periodic interval timer* (PIT). Tento časovač je zložený z 20b časovača, ktorý má nastaviteľnú úroveň pretečenia a z 12b počítadla pretečení. Ak časovač pretečie, tak vyvolá prerušenie, ktoré sme nastavili tak, aby preplo procesor do *Fast interrupt* módu. Prepnutie spôsobí spustenie funkcie *FIQ_Handler_Entry* (5.12), ktorá sa nachádza v súbore Cstartup.S od firmy Atmel. *FIQ_Handler_Entry* na základe adresy, ktorá je uložená v registri *R0* procesor skočí na výkon požadovanej funkcie. Po vyvolaní prerušenia PIT sa zavolá funkcia *_taskSwitch*, ktorá nie je prístupná používateľovi. Táto funkcia, napísaná v jazyku symbolických inštrukcií, overuje, či prepnutie bolo volané z režimu *User*, ak nie, prepnutie sa nevykoná. Po overení sa volá funkcia *switchContext()*, ktorá vráti dve adresy, na ktorých sa nachádza kontextový zásobník. Prvá adresa je zásobník, do ktorého sa odloží kontext starej úlohy a druhá je kontextový zásobník, z ktorej sa načíta kontext novej úlohy. Po vykonaní odloženia a načítania kontextu

```
FIQ_Handler_Entry:

    mov         r9 , r0                                 //odloženie použitých registrov
    mov         r10, r4
    mrs         r4,spsr

    ldr         r0 , [r8, #AIC_FVR]                     //r8 je base aic
    msr         CPSR_c,#I_BIT | F_BIT | ARM_MODE_SVC    //prepnutie do swi

    stmfd       sp!, { r1-r3,r5, r12, lr}               //odloženie registrov do zasobnika

    mov         lr, pc                                  //nastavenie lik reg
    bx          r0                                      //skok do rutiny

    ldmia       sp!, { r1-r3,r5, r12, lr}               //načítanie odložených registrov zo z.

    msr         CPSR_c, #I_BIT | F_BIT | ARM_MODE_FIQ   //prepnutie do FIQ

    msr         spsr,r4                                 //načítanie použitých registrov
    mov         r4, r10
    mov         r0 , r9

    subs pc,lr,#4                                       //prepnutie do User a navrat z fiq
```

Obr. 5.12: Fast interrupt entry

je inicializovaný PIT, a ak je zapnutý štatistický modul, tak sa odloží do kontextu starej úlohy hodnota počítadla pretečení z PIT (obr. 5.13).



Obr. 5.13: Odloženie kontextu, prepnutie úlohy, načítanie kontextu

5.8 Podporná aplikácia na hostiteľskom počítači

Implementovali sme konzolovú aplikáciu, ktorá sleduje sériové rozhranie hostiteľského počítača. Aplikácia bola implementovaná v jazyku C a bola otestovaná len na operačnom systéme Windows 7.

Aplikácia po spustení inicializuje sériový port. Po inicializácií vykonáva nekonečnú slučku, v ktorej sa čaká na príkaz od sledovaného vnoreného zariadenia s MOS. Po identifikovaní príkazu spúšťa spracovanie a výpis informácií prijatých cez sériový port.

6 Testovanie

V tejto kapitole uvádzame postup a výsledky testovania nami implementovaného modulárneho operačného systému.

6.1 Testovacia zostava

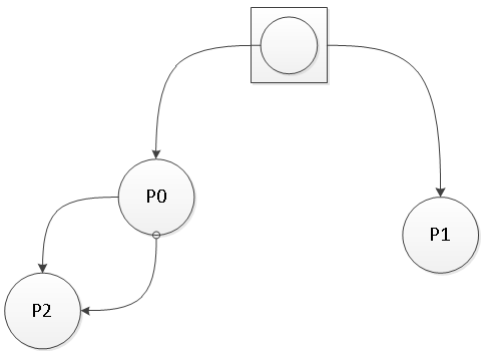
MOS bol testovaný na vývojovej doske *sam7s256* od firmy *Olimex* [8]. Štatistické meranie bolo realizované na notebooku s operačným systémom Windows 7 a interným sériovým rozhraním. Vývojová doska bola programovaná a ladená pomocou programátora *ARM-USB-OCD* od firmy *Olimex* [12].

Z hľadiska programového vybavenia bol na hostiteľskom počítači nainštalovaný:

- kompilátor *arm-none-eabi-gcc* verzia 4.6.2 inštalovaný v balíku *yagaro* a *yagaro tools* [13]
- openOCD server [14]
- ovládače pre *ARM-USB-OCD* programátor [12, 15]
- nami implementovaná aplikácia na čítanie sériovej komunikácie

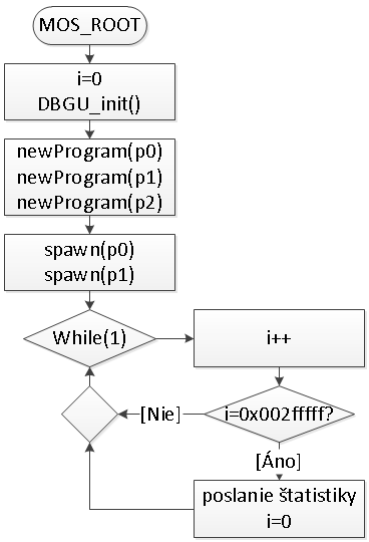
6.2 Testovaná programová štruktúra

Pre overenie funkčnosti MOS sme navrhli jednoduchú programovú štruktúru zobrazenú na obrázku 6.1.



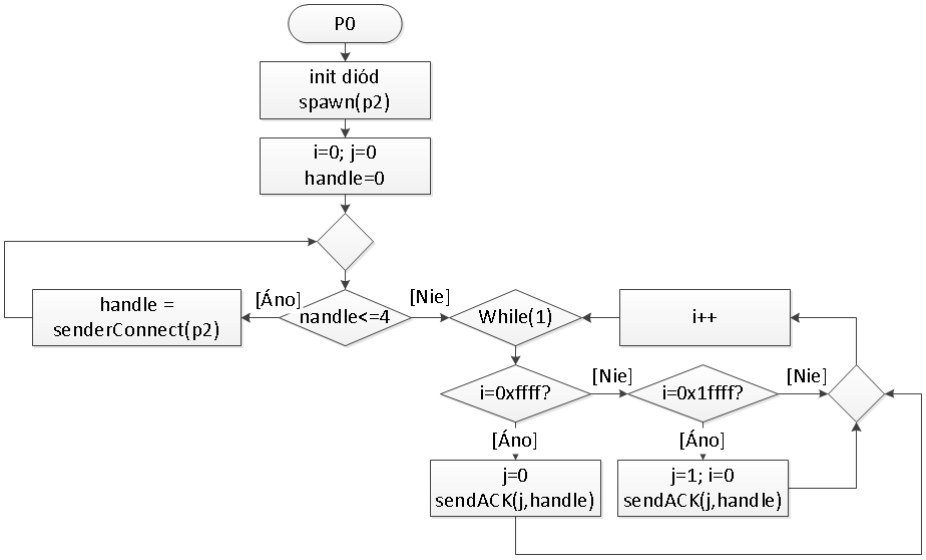
Obr. 6.1: Programový graf testovaného systému

Na obrázkoch 6.2-6.5 sú diagramy priebehu jednotlivých programov spustených na MOS. Koreňový proces vytvorí hlavičky programov a spúšťa procesy p0 a p1. Po spustení procesov plní funkciu štatistického procesu.



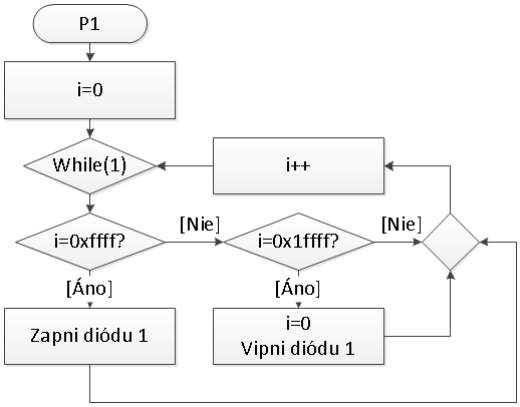
Obr. 6.2: Priebeh koreňového procesu

Po spustení koreňového procesu sa najprv spustí proces P0 6.3. Proces P0 inicializuje port vývojovej dosky s pripojenými diódami, spúšťa proces p2 a v nekonečnej slučke odosiela dáta s potvrdením procesu P2.



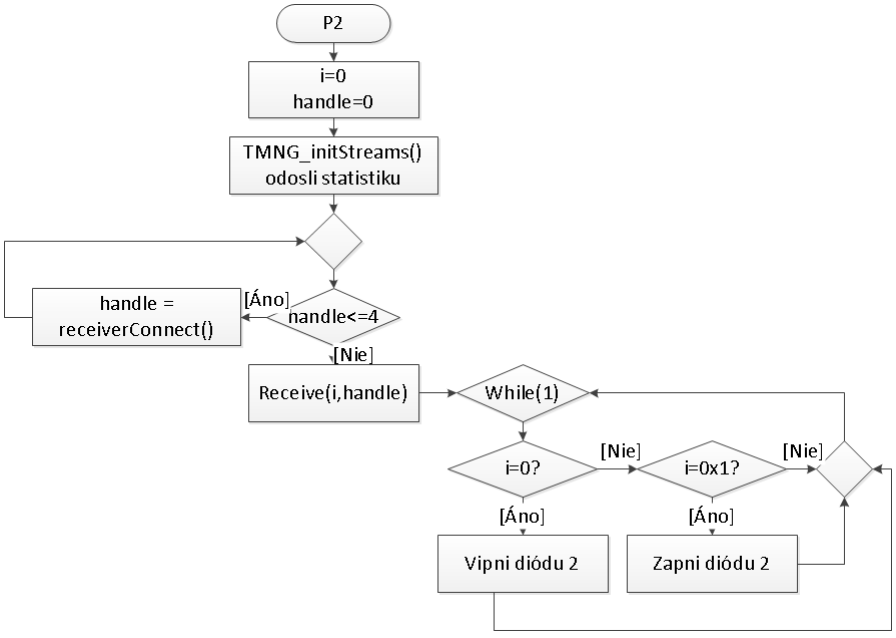
Obr. 6.3: Priebeh procesu p0

Ako posledný sa vytvorí proces P1, ktorý beží samostatne. Proces P1 zapína a vypína v nekonečnej slučke diódu na vývojovej doske 6.4.



Obr. 6.4: Priebeh procesu p1

Proces P2 inicializuje zoznam čakateľov na komunikáciu a inicializuje spojenie, o ktoré žiadal proces P0. Následne prijíma správy od procesu P2. Ak dostane hodnotu "1" tak zapne diódu a ak "0" tak diódu vypne.



Obr. 6.5: Priebeh procesu p2

6.3 Testované konfigurácie MOS

MOS sme testovali v režime s povolenou medziprocesnou komunikáciou a s možnosťou pozastavenia vykonania úloh. V tejto časti uvedieme testy vykonané v režime plánovania Round-Robin s odložením kontextu do hlavičky úlohy alebo do zásobníka úlohy a v režime plánovania s prioritou s odložením kontextu do zásobníka.

6.3.1 Plánovanie Round-Robin

Pri testovaní s plánovaním boli v pamäti programu nasledovné programy (zobrazená hlavička programu) (6.6). V hlavičke programu je uvedená počiatočná adresa programu, veľkosť pamäte dát, počet spustených úloh a počiatočná priorita. V čase výpisu stavu hlavičiek boli všetky úlohy už spustené.

```
Programs in memory: 4
Program 1:
        Begin          : 0x10248c
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 7

Program 2:
        Begin          : 0x102540
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 4

Program 3:
        Begin          : 0x102674
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 5

Program 4:
        Begin          : 0x1026e4
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 7
```

Obr. 6.6: Výstup testovania - hlavičky programov

Odloženie kontextu do zásobníka

MOS sme nechali bežať približne 3 minúty. Výstupom z behu je sumarizácia z obrázku 6.7. V sumári je uvedená aj priorita a prepožičaná priorita, ktorá pre tento prípad nie je smerodajná. Každej úlohe je možné identifikovať otca okrem úlohy 1, ktorá je koreňová. Každá úloha má uvedenú spotrebu pamäte haldy a stav zásobníka. Okrem sumáru úloh je tu uvedená aj spotreba pamäte jadra. Úloha 3 (P2) ukazuje, že bola na procesore podstatne menej ako ostatné úlohy. Je to zapríčinené jej čakaním na správu od úlohy 2 (P0).

```
Tasks in memory: 4
Task 1:
        ProcTime(m:s): 1: 5.1995
        Stack        : end: 0x200564 -> Top: 0x2005ec <- start: 0x200664
        pid          : 1000,  father: 0
        state        : 0
        priority     : 7
        lend Pr      : 7
        Used chunks  : 0
        Free chunks  : 1
        Used memory  : 0(0) / 248(8) B
Task 2:
        ProcTime(m:s): 1: 5.0896
        Stack        : end: 0x20076c -> Top: 0x200814 <- start: 0x20086c
        pid          : 1001,  father: 1000
        state        : 0
        priority     : 4
        lend Pr      : 7
        Used chunks  : 3
        Free chunks  : 2
        Used memory  : 76(24) / 248(16) B
Task 3:
        ProcTime(m:s): 0: 0.0895
        Stack        : end: 0x200974 -> Top: 0x2009fc <- start: 0x200a74
        pid          : 1002,  father: 1001
        state        : 1
        priority     : 7
        lend Pr      : 4
        Used chunks  : 2
        Free chunks  : 1
        Used memory  : 40(16) / 248(8) B
Task 4:
        ProcTime(m:s): 1: 5.1863
        Stack        : end: 0x200b7c -> Top: 0x200c2c <- start: 0x200c7c
        pid          : 1003,  father: 1000
        state        : 0
        priority     : 5
        lend Pr      : 7
        Used chunks  : 0
        Free chunks  : 1
        Used memory  : 0(0) / 248(8) B

ProcTime Total (m:s) : 3: 15.5649
-----------------------------------------
Kernel Stack :  End 0x20fda0 -> Top 0x20ff7c <- Begin 0x20ffa0
-----------------------------------------

Kernel heap data memory:

        Used chunks  : 20
        Free chunks  : 1
        Used memory  : 624(160) / 1032(8) B
-----------------------------------------

Kernel heap process memory:

        Used chunks  : 8
        Free chunks  : 1
        Used memory  : 2080(64) / 63812(8) B
```

Obr. 6.7: Výstup testovania - hlavičky procesov a využitie pamäte (odloženie do zásobníka)

Odloženie kontextu do hlavičky

MOS sme nechali bežať približne 3 minúty. Výstupom z behu je sumarizácia z obrázku 6.8. Ako vidno vo využití zásobníka, je tu menšia spotreba vo všetkých úlohách oproti odloženiu kontextu do zásobníka. Úloha 3 (P2) má tak málo procesného času pretože väčšinu času bola pozastavená. Čakala na správu od úlohy 2 (P0).

```
Tasks in memory: 4
Task 1:
        ProcTime(m:s): 1: 5.7583
        Stack         : end: 0x200564 -> Top: 0x20062c <- start: 0x200664
        pid           : 1000,  father: 0
        state         : 0
        priority      : 7
        lend Pr       : 7
        Part of Context:  PC -> 0x1005f0  CPSR -> 0x60000010  LR -> 0x101f3c
        Used chunks   : 0
        Free chunks   : 1
        Used memory   : 0(0) / 248(8) B
Task 2:
        ProcTime(m:s): 1: 5.6538
        Stack         : end: 0x20076c -> Top: 0x200854 <- start: 0x20086c
        pid           : 1001,  father: 1000
        state         : 0
        priority      : 4
        lend Pr       : 7
        Part of Context:  PC -> 0x102644  CPSR -> 0x80000010  LR -> 0x102614
        Used chunks   : 3
        Free chunks   : 2
        Used memory   : 76(24) / 248(16) B
Task 3:
        ProcTime(m:s): 0: 0.0903
        Stack         : end: 0x200974 -> Top: 0x200a3c <- start: 0x200a74
        pid           : 1002,  father: 1001
        state         : 1
        priority      : 7
        lend Pr       : 4
        Part of Context:  PC -> 0x101db4  CPSR -> 0x20000010  LR -> 0x1022e4
        Used chunks   : 2
        Free chunks   : 1
        Used memory   : 40(16) / 248(8) B
Task 4:
        ProcTime(m:s): 1: 5.7643
        Stack         : end: 0x200b7c -> Top: 0x200c6c <- start: 0x200c7c
        pid           : 1003,  father: 1000
        state         : 0
        priority      : 5
        lend Pr       : 7
        Part of Context:  PC -> 0x102670  CPSR -> 0x80000010  LR -> 0x1001c8
        Used chunks   : 0
        Free chunks   : 1
        Used memory   : 0(0) / 248(8) B

ProcTime Total (m:s) : 3: 17.2667
-----------------------------------------
Kernel Stack :  End 0x20fda0 -> Top 0x20ff7c <- Begin 0x20ffa0
-----------------------------------------

Kernel heap data memory:

        Used chunks   : 20
        Free chunks   : 1
        Used memory   : 880(160) / 1032(8) B
-----------------------------------------

Kernel heap process memory:

        Used chunks   : 8
        Free chunks   : 1
        Used memory   : 2080(64) / 63812(8) B
```

Obr. 6.8: Výstup testovania - hlavičky procesov a využitie pamäte (odloženie do hlavičky)

6.3.2 Plánovanie s prioritou

Na obrázku 6.9 je uvedený stav programov uložených v pamäti programov a počet spustených úloh v danom procesore pre daný program.

```
Programs in memory: 4
Program 1:
        Begin          : 0x10254c
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 7

Program 2:
        Begin          : 0x102600
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 5

Program 3:
        Begin          : 0x10271c
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 5

Program 4:
        Begin          : 0x10278c
        DataL          : 512
        Num of Tasks   : 1
        Start priority: 7
```

Obr. 6.9: Výstup testovania - hlavičky programov

Na obrázku 6.10 je sumár stavu úloh po uplynutí približne troch minút. Ako vidno najväčšiu prevahu má úloha 1, pretože má najvyššiu prioritu. Priorita je daná súčtom priority a prepožičanej priority. Ostatné úlohy majú rovnakú prioritu po súčte s prepožičanou prioritou. Úloha 3 ako aj v minulých meraniach vykazuje nízke percento plánovania pretože čaká na správu od úlohy 2.

```
Tasks in memory: 4
Task 1:
        ProcTime(m:s): 1: 39.2648
        Stack        : end: 0x200564 -> Top: 0x2005ec <- start: 0x200664
        pid          : 1000,  father: 0
        state        : 0
        priority     : 7
        lend Pr      : 7
        Used chunks  : 0
        Free chunks  : 1
        Used memory  : 0(0) / 248(8) B
Task 2:
        ProcTime(m:s): 0: 49.5437
        Stack        : end: 0x20076c -> Top: 0x200814 <- start: 0x20086c
        pid          : 1001,  father: 1000
        state        : 0
        priority     : 5
        lend Pr      : 7
        Used chunks  : 3
        Free chunks  : 2
        Used memory  : 76(24) / 248(16) B
Task 3:
        ProcTime(m:s): 0: 0.0694
        Stack        : end: 0x200974 -> Top: 0x2009fc <- start: 0x200a74
        pid          : 1002,  father: 1001
        state        : 1
        priority     : 7
        lend Pr      : 5
        Used chunks  : 2
        Free chunks  : 1
        Used memory  : 40(16) / 248(8) B
Task 4:
        ProcTime(m:s): 0: 49.6131
        Stack        : end: 0x200b7c -> Top: 0x200c2c <- start: 0x200c7c
        pid          : 1003,  father: 1000
        state        : 0
        priority     : 5
        lend Pr      : 7
        Used chunks  : 0
        Free chunks  : 1
        Used memory  : 0(0) / 248(8) B

ProcTime Total (m:s) : 3: 18.4910
-----------------------------------------
Kernel Stack :  End 0x20fda0 -> Top 0x20ff7c <- Begin 0x20ffa0
-----------------------------------------

Kernel heap data memory:

        Used chunks  : 20
        Free chunks  : 1
        Used memory  : 624(160) / 1032(8) B
-----------------------------------------

Kernel heap process memory:

        Used chunks  : 8
        Free chunks  : 1
        Used memory  : 2080(64) / 63812(8) B
```

Obr. 6.10: Výstup testovania - hlavičky procesov a využitie pamäte (odloženie do zásobníku)

6.4 Prepnutie úlohy

Prepnutie kontextu od vyvolania trvalo v režime Round-robin 14,3 mikro sekundy pri taktovaní procesora 48 MHz. Čo je 14,3 percentná réžia pri taktovaní prepnutia 10 KHz. Pri ostatných plánovaniach je prepnutie v ideálnom prípade rovnako dlhé ale ak nastáva usporiadanie úloh, tak prepnutie sa predlžuje. Samotné odloženie a načítanie kontextu trvá 3,3 mikrosekundy pri rovnakom taktovaní 48 MHz. Výpočet trvania prepnutia bol vykonaný počas režimu ladenia odčítaním hodnoty z PIT.

6.5 Využitie pamäte dát

Plánovač samostatne zaberá 17 B (freeRTOS 236 B [16]) + 16 B za každú plánovanú úlohu.

Úloha v minimálnej konfigurácií zaberá 30 B + 16 B (freeRTOS 64 B [16]). V maximálnej konfigurácií minie 116 B + 16 B.

Rad minie 25 B (freeRTOS 76 B [16]) dát + 12 B za každú novú položku.

6.6 Využitie pamäte programu

Minimálna konfigurácia zaberie približne 5,5 kB pamäte programu. V maximálnej konfigurácii zaberie približne 9,5 kB pamäte programu.

7 Zhodnotenie

Projekt spĺnil všetky stanovené ciele. Výstupom projektu je fungujúce jadro modulárneho operačného systému. Jadro operačného systému je výlučne preemptívne a pozostáva zo správy pamäte, správy procesov, plánovača, medziprocesnej komunikácie a riadenia prístupu. MOS je rozšírený o štatistický modul, ktorým je možné odosielať dáta pre externú aplikáciu. Implementovali sme externú aplikáciu, ktorá spracuje prichádzajúce dáta zo sériového rozhrania.

Počas práce na projekte sme sa stretli s mnohými problémami, ktoré sme vyriešili. Príkladom bol problem s viditeľnosťou funkcií, ktoré sme implementovali. Niektoré verejné funkcie sme chceli použiť len v rámci modulov MOS. Problém sme nakoniec ošetrili overením, či bol procesor prepnutý do chráneného režimu. Problém nastal aj pri realizácii prepnutia kontextu. Prepnutie je riadené prerušením, ktoré je mapované do FIQ módu. Problémom bolo občasné prerušenie aj v chránenom režime procesora, keďže nebolo zakázané FIQ prerušenie. Toto sme ošetrili tým, že hneď po prepnutí do chráneného režimu sme zakázali prerušenie FIQ. Dodatočne sme implementovali overovanie, či FIQ prerušenie nastalo výlučne v používateľskom móde procesora. Nie menej dôležitým problémom bola potreba prepnutia procesora do chráneného režimu s možnosťou pokračovania v kóde ale v novom režime bez možnosti zneužitia používateľom. Toto sme vyriešili overením adresy odkiaľ bola požiadavka zadaná. Ak požiadavka je zadaná z adresy patriacej kódu MOS, tak dané prepnutie bude vykonané. Zásadným problémom pri návrhu bola identifikácia najvhodnejšieho začiatku implementácie. Po identifikovaní najvhodnejšieho začiatku, ktorým sa stal modul Správa pamäte, nabrala implementácia rýchly spád. Počas implementácie sme sa riadili modulárnym a prírastkovým vývojom. Získali sme zručnosť s jazykom symbolických inštrukcií pre platformu ARM, čo považujeme za cennú skúsenosť.

Počas písania tohto dokumentu sme si uvedomili zopár nejasností a nezmyselností v implementácií MOS. MOS nemá implementovaný mechanizmus časovaného odloženia výkonu úlohy Blokovací mechanizmus. Tento nedostatok sa odráža pri plánovaní s prioritou, keď úloha, ktorá by mohla čakať na pridelenie prostriedku zaberá priestor plánovania a nedovoľuje vykonanie ostatných úloh, ktoré majú nižšiu prioritu.

Napriek týmto maličkostiam môžeme zhodnotiť, že MOS je z pamäťovej (programovej aj dátovej) stránky efektívnejší ako freeRTOS, čo je pre nás veľkým úspechom. Rozdelenie kódu podľa platformovej závislosti zefektívňuje a urýchľuje prenášanie na iné platformy. Na prenesenie na inú platformu stačí implementovať platformovo závislý kód.

Do budúcnosti by sme chceli pracovať na zdokonaľovaní MOS. Uvedomili sme si, že implementácia štruktúr spájaného zoznamu a zároveň radu značne zvyšuje nároky na veľkosť

kódu. Navyše pri plánovaní úloh by postačovala na uloženie úlohy aj štruktúra rad. Zároveň sme si počas testovania všimli malú časť kódu, ktorá je platformovo závislá ale je v súbore, ktorý nie je platformovo závislý, čo môže predĺžiť čas prenesenia MOS na iné platformy.

V konečnom zhodnotení hodnotíme projekt ako veľmi úspešný. Pre nás práca na projekte znamená obrovskú skúsenosť a množstvo nových vedomostí v oblasti vnorených operačných systémov, práci s rodinou procesorov ARMv4, zručností s jazykom symbolických inštrukcií a jazykom C.

Literatúra

[1] A. S. Tanenbaum, *Modern Operating Systems (2nd Edition) (GOAL Series)*. Prentice Hall, 2001.

[2] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, 2006.

[3] J. Štefanovič, *Základy operačných systémov*. STU, 2007.

[4] J. J. Labrosse, J. Ganssle, and e. a. Robert Oshana, *Embedded Software: Know It All (Newnes Know It All)*. Newnes, 2007.

[5] L. Barello, *AvrX Real Time Kernel*, 2007. http://www.barello.net/avrx/.

[6] *The FreeRTOS Project*, 2011. http://www.freertos.org/.

[7] *TinyOS*, 2011. http://www.tinyos.net.

[8] *Datasheet, SAM7-S256 development board, Users Manual*, 2008.

[9] *Datasheet, AT91SAM7S256*, 2010.

[10] *Technical Reference Manual, ARM7TDMI (Rev 3)*, 2001.

[11] *User Guide, ARM Software Development Toolkit V. 2.50*, 1998.

[12] *Datasheet, ARM USB JTAG DEBUGGER*, 2006.

[13] M. Fischer, *YAGARTO, Yet another GNU ARM toolchain*, 2012. http://www.yagarto.de/.

[14] *User Guide, Open On-Chip Debugger: OpenOCD*, 2012.

[15] *Installing ARM-USB-OCD - rev.G drivers for Windows 7*, 2010.

[16] *FreeRTOS FAQ - Memory Usage and Boot Times*, 2012. http://www.freertos.org/FAQMem.html.

A Digitálne médium

- **/dokument** - obsahuje digitálnu formu tohto dokumentu vo formáte PDF.
- **/MOS** - obsahuje implementované časti modulárneho operačného systému.
- **/podporná dokumentácia** - obsahuje katalógové listy a manuály použitých hardvérových komponentov.
- **/podporný softvér** - obsahuje softvér a ovládače, ktoré boli použité pri implementovaní a testovaní MOS.

B Vývojárska príručka

Inštalácia vývojového prostredia

Na priloženom digitálnom nosiči je uložený softvér potrebný na rozbehnutie systému. Dostupnú inštaláciu sme používali na operačnom systéme Windows 7. Na ostatných OS nevieme zaručiť funkčnosť.

1. Rozbaľte .zip súbor *OpenOCD_OnlinePackage_MV.zip* uložený v priečinku */podporný softvér* na digitálnom médiu. Odporúčame, aby balíček bol rozbalený do krátkej cesty a aby cesta neobsahovala medzery.

2. V priečinku */ARM-USB-OCD-DRIVER*, ktorý je v rozbalenom balíčku sú uložené ovládače. Postupujte podľa návodu *How to install OpenOCD on Windows 7.pdf* uloženého v tomto priečinku.

3. Pridajte do premennej prostredia *Path* priečinok */openOCD*, ktorý sa nachádza v balíčku

4. Nainštalujete *YAGARTO* umiestnený v */podporný softvér*. Postupujte podľa pokynov inštalátora. Nezabudnite zaškrtnúť uloženie do premenných prostredia. Najlepšie je nainštalovať do priečinku kde ste rozbalili *OpenOCD_OnlinePackage_MV*.

5. Nainštalujte *YAGARTO-tools* umiestnený v */podporný softvér*. Postupujte podľa pokynov inštalátora. Najlepšie je nainštalovať do priečinku kde ste rozbalili *OpenOCD_OnlinePackage_MV*.

Ďalší manuál k openOCD nájdete v priečinku */podporná dokumentácia* pod názvom *openOCD.pdf*

Inštalácia portfólia MOS

Portfólio MOS je uložené v priečinku */MOS*. Skopírujte tento priečinok kam potrebujete. Pre spustenie ladenia MOS otvorte príkazový riadok nastavený do priečinku *mos/microcon/at91sam7s*. Zadajte príkaz *make debug*. Spustíte tým kompiláciu MOS a program gdb (ladenie). Zadaním príkazu *make all* kompilujete zdrojové súbory v danej konfigurácií. Príkaz vylistuje zdrojový kód a sumarizáciu. Zadaním príkazu *make program* spustíte naprogramovanie skompilovaného MOS do vývojovej dosky sam7-p256. Príkazom *make clean* vyčistíte kompiláciu z počítača.

Prenesenie na iné platformy

Ak chcete preniest' MOS na inú platformu je potrebné:

- vytvorit' kompilačný súbor, v ktorom sa kompilujú jednotlivé súbory MOS.
- implementovat' inicializačný .S súbor ak je potrebné inicializovat' zásobníky.
- implementovat' súbor *main.c* V tomto súbore sa inicializuje podpora jazyku C. V tomto súbore sa inicializuje MOS. Tu sa volajú funkcie *MMNG_init(), _TMNG_init(), InitTimedTaskSwitch(), MOSP_start()*. Poradie musí byt' zachované.
- implementovat' funkciu *InitTimedTaskSwitch()*, v ktorej sa inicializuje časovač prepnutia.
- implementovat' funkciu *MOSP_start()*, ktorá spúšt'a MOS.
- implementovat' obálky volaní a zabezpečit' chránený režim pred používateľom.
- implementovat' odloženie kontextu, v ktorom sa musí zavolat' funkcia *switchContext()*.

C Používateľská príručka

Konfigurácia MOS

MOS sa konfiguruje cez konfiguračný súbor *config.h* V tomto súbore používateľ nastaví požadované parametre MOS.

Vlastné programy

Program tvorí svoj program ako funkciu bez vstupného a výstupného parametra. Táto funkcia môže ale nemusí mať v sebe nekonečnú slučku. Okrem funkcie používateľ vytvorí v súbore MOS.h alebo priloží vo vlastnom súbore makro s definovaním PID programu. Používateľ priloží deklaráciu funkcie. Každý program má mať svoj vlastný jedinečný identifikátor. Programu, sa počas spustenia MOS, musí vytvoriť hlavička programu pomocou volania newProgram(PID,meno funkcie,priorita,veľkosť dát). Toto volanie je najlepšie umiestniť do koreňového procesu.

Koreňový proces

V súbore MOS.c je funkcia *MOS_ROOT*. V tele funkcie odporúčame najprv vytvoriť hlavičky všetkých programov v MOS. Následne používateľ spúšťa prvý používateľský proces pomocou volania spawn(pid). Pre potreby merania MOS odporúčame tu uviesť v slučke odosielanie štatistických meraní.

Volania MOS

- T_Pid TMNG_taskPid() - vracia pid aktuálnej úlohy
- void * TMNG_taskHeap() - vracia haldu aktuálnej úlohy
- T_ListAttribute TMNG_TaskPriority() - vracia prioritu aktuálnej úlohy
- T_UInt TMNG_InitStreams() - inicializuje dátovú komunikáciu aktuálnej úlohy
- void * TMNG_popStream() - výber žiadosti adresovanej aktuálnej úlohe
- void taskEnd() - ukončenie úlohy
- T_Pid spawn(T_ListAttribute pid) - vytvorenie novej úlohy založenej na programe daného pid

- T_PCB newProgram(T_ListAttribute pid, void * begin, T_ListAttribute priority, T_Long len) - vytvorenie hlavičky pre novy program
- void DBGU_init() - inicializovanie sériového prepojenia pre potreby štatistického merania
- void TMNG_statistics_PCBT() - odoslanie štatistiky programov cez sériové rozhranie
- void TMNG_statistics_TCBT() - odoslanie štatistiky úloh cez sériové rozhranie
- void* malloc(T_UInt size) - alokovanie priestoru na halde
- void free(void * pointer) - uvolnenie alokovaného piestoru na halde
- void MMNG_statistics_kernelHeap(T_Char cmd) - odoslanie štatistiky haldy jadra/procesnej pamäte
- void MMNG_statistics_kernelStack() - odoslanie štatistiky využitia zásobníka
- T_IPCHandle senderConnect(T_Pid pidReceiver) - pripojenie k danej úlohe žiadosťou
- T_IPCHandle receiverConnect() - pripojenie k žiadateľovi
- T_UInt send(void *message, T_IPCHandle handle) - blokujúce odoslanie správy
- T_UInt sendAck(void *message, T_IPCHandle handle) - blokujúce odoslanie správy s potvrdením
- T_UInt notify(void *message, T_IPCHandle handle) - neblokujúce odoslanie správy
- T_Pid senderPid(T_IPCHandle handle) - zistenie pid úlohy, ktorá odosiela
- T_UInt receive(void **message, T_IPCHandle handle) - blokujúce prijatie správy
- T_UInt read(void **message, T_IPCHandle handle) - neblokujúce prijatie správy
- T_UInt receiverDisconnect(T_IPCHandle handle) - odpojenie od odosielateľa
- T_UInt senderDisconnect(T_IPCHandle handle) - odpojenie od prijímateľa
- T_UInt streamDestroy(T_IPCHandle handle) - zrušenie komunikačného toku
- T_UInt streamState(T_IPCHandle handle) - navrátenie stavu komunikačného toku