ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE
PROVOZNĚ EKONOMICKÁ FAKULTA

Katedra Informačních Technologií

BAKALÁŘSKÁ PRÁCE

Operační systémy pro zpracování úloh v reálném čase

Vít Zahrádka

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Operační systémy pro zpracování úloh v reálném čase" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 30.3.2011 _____

Poděkování

Na tomto místě bych rád poděkoval Ing. Jiřímu Vaňkovi, Ph.D. za cenné rady, pokyny a ochotu při vedení této bakalářské práce.

Operační systémy pro zpracování úloh v reálném čase

Souhrn

Obsahem bakalářské práce je prezentace principů tvorby procesů v reálném čase a metod, které nabízí standard pro programování úloh v reálném čase POSIX.4. Práce zachycuje problematiku operačních systémů pro zpracování úloh v reálném čase. Pozornost je soustředěna zejména na prostředí operačního systému QNX, jeho mechanismy podporující tvorbu procesů pracujících v reálném čase a doporučené postupy pro jejich programování.

Klíčová slova

reálný čas, operační systémy, programování, standard POSIX, QNX, RTOS, meziprocesová komunikace, mikrojádro

Real-time operating systems

Summary

The thesis presents basic principles of creating real-time processes and tools provided by POSIX.4 real-time extensions standard. It focuses especially on real-time operating systems and concentrates particularly on system QNX, its mechanisms for programming of real-time processes and finally defines policy to create real-time process.

Keywords

real-time, operating systems, programming, POSIX standards, QNX, RTOS, inter-process communication, microkernel

Obsah

Seznam obrázků a tabulek

1. Úvod

Operační systémy jsou součást počítače, na kterou jsme si mnohdy již zvykli natolik, že už ani nevnímáme drobné problémy, kterými nás překvapí. Uživatel počítače si již zvykl počkat nějakou tu vteřinu na reakci provozované aplikace, je-li jeho operační systém zrovna vytížen a ani to nevnímá jako událost, se kterou by mohly přijít opravdové nenapravitelné škody.

Opačná situace však panuje mezi aplikacemi, které obsluhují například důležitá zařízení na letištích, v chemických továrnách a atomových elektrárnách. Často i zdržení v řádech zlomku vteřiny by u takových aplikacích mohlo mít za následek smrt stovek lidí a nenapravitelné škody na životním prostředí. A to je důvod, proč se problematikou operačních systémů pro zpracování úloh v reálném čase[1] zabývá po celém světě mnoho lidí.

Od operačních systémů pro zpracování úloh v reálném čase očekáváme, že umožní aplikacím, které jsou na něm provozovány reagovat na podněty *okamžitě*, avšak to není z mnoha důvodů proveditelné – operační systém nemůže přerušit prioritní systémovou úlohu, aby umožnil aplikaci reagovat na vyvstanuvší situaci. Základem zpracování úloh v reálném čase proto není umět reagovat na události dostatečně rychle, ale *spolehlivě dostatečně rychle*. Nastane-li v jaderném reaktoru situace, která bude vyžadovat korekci aplikace, budeme vyžadovat, aby aplikace na tuto situaci *vždy* zareagovala dostatečně včas. Pozdní reakce by v tomto konkrétním případě potom mohla mít opravdu fatální následky. Při programování aplikací v reálném čase tedy musíme umět navrhnout takový systém, který se spolehlivě vypořádá s časovými omezeními během náhodných asynchronních událostí. Ač se toto tvrzení zdá jednoduché, jeho realizace je natolik složitá, že dokázala zaměstnat mnoho teoretiků i vývojářů a zaplnit mnoho stran knih.

1 V dalším textu můžeme pro termín „operační systém pro zpracování úloh v reálném čase“ použít zkratku RTOS, která vychází z anglického termínu *real-time operating system*

2. Cíl práce a metodika

V současné době neexistuje stručný přehled metodiky programování procesů v reálném čase a tato práce si klade za cíl uvedenou situaci změnit. Hlavním cílem řešení je proto srovnání programovacích technik používaných při běžném programování s technikami operačních systémů pro zpracování úloh v reálném čase a stanovení doporučení, jimiž by se potenciální programátoři aplikací pracujících v reálném čase měli řídit.
Dílčími cíli dále jsou:

- osvětlit specifika operačních systémů pro zpracování úloh v reálném čase
- stanovit nejčastější typy úloh, které jsou v prostředí operačních systémů pro zpracování úloh v reálném čase řešeny
- na vybraném operačním systému demonstrovat specifika operačních systémů pro zpracování úloh v reálném čase
- srovnat programování aplikací v reálném čase s programováním běžných aplikací
- dosáhnout hlavního cíle práce, stanovit vhodné programovací techniky a návyky při programování úloh v reálném čase
- formulovat soubor doporučení pro tvorbu aplikačních procesů pracujících v reálném čase

2.1 Metodika

V kapitole „Přehled řešené problematiky“ budou nejprve rozebrána specifika operačních systémů pro zpracování úloh v reálném čase a uvedeny nejčastější úlohy zpracované pomocí RTOS. Práce se zaměří nejprve na požadavky, které jsou na tyto systémy kladeny, a ty uvede do souvislosti s ekvivalentními operačními systémy nepodporujícími práci v reálném čase. Stranou zájmu nezůstane ani teoretické pozadí problematiky RTOS.

V kapitole „Studie vybraného operačního systému“ bude podrobně rozebrán vybraný operační systém pro zpracování úloh v reálném čase a podroben zkoumání jeho možností i úskalí. V podkapitole „Architektura“ bude vysvětlena koncepce vybraného systému a rozebrány možnosti jeho architektury. Zmíněna budou i specifika vybraného operačního systému a nástroje, jichž tento systém využívá.

„Tvorba aplikací pro práci v reálném čase“ bude řešit problematiku programování aplikací v reálném čase. Rozebíráno bude, jaká kritéria je třeba splnit, aby se naše aplikace chovala správně, nenarušovala koncepci systému a přitom plně využívala jeho možnosti? Jako poslední bude diskutováno použití vhodných programátorských technik a návyků v prostředí vybraného operačního systému pro zpracování úloh v reálném čase.

Výsledkem bude formulace doporučení pro tvorbu aplikačních procesů pracujících v reálném čase.

3. Přehled řešené problematiky

V přehledu řešené problematiky bude blíže rozebrán fenomén operačních systémů pro zpracování úloh v reálném čase.

Operační systémy pro zpracování úloh v reálném čase jsou *deterministické* operační systémy, u nichž *lze jednoznačně určit maximální rychlost jejich reakce* na podnět. Je-li pak nahlíženo na RTOS z tohoto hlediska, je nutno říci, že charakteristiky jako rychlost, velikost nebo typ jádra, případně další možnosti systému jsou nepodstatné.

O RTOS tvrdíme, že jsou deterministické, protože operace jsou v těchto systémech prováděny ve fixovaných, předem určených časech nebo časových intervalech a zároveň reakce na přerušení probíhá tak, že systém je schopen obsluhy všech požadavků v daném čase.

Obecně lze systém reálného času definovat jako informační systém, který zpracovává asynchronní vstupy a produkuje odpovědi v pevně stanoveném čase. Doba, kterou má systém k dispozici pro provedení úlohy, je známa předem. Na počtu vstupů a jimi vyvolané pracovní zátěži systému přitom nezáleží. Volnější definice, která je blízká skutečným systémům, zní:

"Systém reálného času je takový informační systém, který zpracovává asynchronní vstupy a produkuje odpovědi v rozhodnutém a ohraničeném čase."[8]

Tato definice odráží skutečnost, že potřebný čas může být vypočítán ze zátěže systému. Navíc musí být tento čas ohraničený, což odpovídá pojmu nejdelší možná doba odezvy.

Oproti běžným operačním systémům jsou RTOS specifické zejména v charakteristikách, které jim umožňují minimalizovat čas potřebný pro přijetí a dokončení

úlohy. Ve smyslu nároků na garanci odezvy systému na podnět lze rozlišovat mezi tzv. *měkkými* a *tvrdými* RTOS[2].

Tvrdé RTOS mají velké nároky na rychlost a spolehlivost reakce na podnět. Tvrdé RTOS se typicky používají v situacích, kdy pozdní nebo dokonce žádná reakce může mít fatální následky. Běžně se vyskytují v palubních počítačích letadel nebo v řídících počítačích chemických továren a atomových elektráren.

Abstraktní definice systému hard RT využívá pojmu využitelnosti odezvy na podnět:

„Praktická využitelnost hodnoty výpočtové aktivity klesá u systému hard RT k nule v okamžiku, kdy je dosažena nebo překročena požadovaná doba odezvy."[8]

Vysvětlit tuto abstraktní definici lze například tak, že požadavky na tvrdý RTOS nutí systém vždy splnit zadaný časový termín. Tvrdému RTOS jsou tedy zadány časové krajní meze, které musí splnit, aby nenastala katastrofická selhání, jako je ztráta životů nebo zařízení.

Tvrdé RTOS je tedy možno definovat jako:

„… systém u nějž je garantováno dokončení naprogramované reakce na podnět ve známém konečném čase.”[9]

Oproti tvrdým jsou měkké RTOS v nárocích na včasnou odezvu tolerantnější. Zpravidla se používají pro úlohy, ve kterých zpoždění provedení úlohy nemá fatální následky, ale může způsobit degradaci výsledku úlohy (např. ztráta výkonu). Typicky lze měkké RTOS použít jako řídící počítače automobilů, výrobních linek, či automatizovaných provozů nebo pro široké spektrum simulací a her. Měkké RTOS lze definovat jako:

„… systém u nějž je naprogramovaná reakce na podnět téměř vždy dokončena ve známém konečném čase.”[9]

Volně zpracováno podle [1], [8], [9]

2 Výrazy měkký a tvrdý RTOS vycházejí z anglických výrazů "soft" a "hard".

3.1 Specifika RTOS

V této kapitole budou diskutována specifika RTOS a jejich úskalí i výhody. [1], [9]

3.1.1 Přepínání úloh

RTOS používají nejčastěji dva druhy přepínání úloh:

- Přepínání řízené událostmi (*event-driven task switching*) provede přepnutí úlohy pouze v případě, že se o výpočetní čas zažádá proces s vyšší prioritou.
- Sdílení času (*time-sharing*) přepíná úlohy jako reakci na časové přerušení nebo na událost. Ve výsledku tedy přepíná úlohy mezi sebou podstatně častěji, než by bylo třeba, nicméně díky tomu umožňuje hladší zpracování více úloh najednou.

Rané procesory strávily příliš mnoho času při zbytečném přepínání úloh oproti dnešním[3] procesorům, a to bylo také důvodem, proč původní se RTOS snažily minimalizovat plýtvání výpočetním časem na zbytečné přepínání úloh.

3.1.2 Plánování procesů

V plánování procesů potom RTOS používají zejména tři základní stavy procesů: „proces je spuštěný na CPU“ (*running*), „proces je připravený pro běh na CPU“ (*ready*) a „proces je blokovaný“ - čekající na vstup nebo výstup (*blocked*). Vzhledem k omezení procesoru bývají zpravidla úlohy po většinu času ve stavu *ready* nebo *blocked* - procesor umí v daném okamžiku zpracovávat pouze jednu úlohu.

Obvykle je datová struktura plánovače úloh navržena tak, aby minimalizovala nejdelší možný čas, který proces může strávit v kritické sekci plánovače, během níž jsou preemptivní vlastnosti potlačeny a v některých případech dokonce potlačena i přerušení. Výběr datové struktury plánovače ovšem závisí i na maximálním počtu úloh, které mohou být v seznamu úloh připravených pro zpracování.

V případě, že v seznamu nefiguruje vždy více než několik úloh, potom se jeví optimálním řešením obousměrný spojový seznam. Pokud seznam obsahuje pouze několik položek, ovšem občas se stává, že jich obsahuje i více, potom by měl seznam být řazen podle priority úkolů. Vyhledávání v takto seřazeném seznamu potom nevyžaduje

3 Například 20 MHz procesor 68000 (používaný na konci osmdesátých let 20. stol.) přepínal úlohy průměrně v čase 20 mikrosekund. Oproti tomu, 100 MHz ARM CPU (z roku 2008) přepne úlohu za méně než 3 mikrosekundy.

procházení celé struktury. Vkládání požadavku potom vyžaduje pouze projít strukturu seznamu a vyhledat buď jeho konec, nebo úlohu s menší prioritou, před níž se vkládaný požadavek zařadí.

Je třeba dát pozor, aby při tomto vyhledávání nebyla narušena preempce[4] systému. Delší a důležitější části úloh by měly být rozděleny do menších logických útvarů a řešeny zvlášť. Pokud pak nastane přerušení, které změní stav prioritního procesu na "připraven ke zpracování" během vkládání méně prioritní úlohy do seznamu připravených úloh, prioritní úloha by měla být do seznamu vložena okamžitě, předtím, než je do seznamu zanesena ta méně prioritní, a okamžitě spuštěna.

V komplexnějších RTOS mohou úlohy vyžadující zpracování v reálném čase sdílet systémové zdroje s úlohami, které zpracování v reálném čase nevyžadují. Seznam připravených úloh se proto může libovolně roztáhnout. V takových systémech je zpracování seznamu připravených procesů pomocí struktury spojového seznamu nevhodné.

3.1.3 Meziprocesová komunikace

Operační systémy zpracovávající několik (i mnoho) úloh najednou se musí vypořádat i se sdílením dat a hardwarových zdrojů mezi jednotlivými úlohami. Avšak simultánní přístup dvou nebo více procesů ke stejným systémovým zdrojům je v naprosté většině nebezpečnou záležitostí. Výsledky takového používání stejných zdrojů mohou být nekonzistentní a nepředvídatelné. Problém vzájemné komunikace řeší úlohy způsoby popsanými následujícími podkapitolami.

3.1.3.1 Dočasné blokování přerušení

Obecně operační systémy nedovolují uživatelům blokovat přerušení - uživatelské programy by tím získaly kontrolu nad procesorovou jednotkou po neomezenou dobu. Moderní procesory dokonce nedovolují programům v uživatelském módu blokovat přerušení, protože je považuje za klíčové pro fungování operačního systému. Nicméně,

[4] Preempcí nazýváme takovou funkčnost operačního systému, která umožní přerušit právě prováděnou úlohu, aniž by byla vyžadována v tomto směru aktivní spolupráce této úlohy, a nahradit ji jinou úlohou. Přerušená úloha je pomocí stejného mechanismu opět obnovena a dokončena. Preempce blízce souvisí s tzv. multitaskingem (zpracování více úloh na procesoru) a změnou kontextu (angl. context switching).

mnoho vestavěných systémů[5] a RTOS dovolují aplikacím fungovat v módu jádra[6], aby zvýšily účinnost a efektivitu systémových volání a daly aplikacím více kontroly nad prostředím, ve kterém fungují bez zásahu operačního systému.

V aplikacích běžících na jednoprocesorových systémech v módu jádra a schopných maskování přerušení je často dobrým přístupem deaktivace (dočasná či úplná) přerušení - zabraňuje simultánnímu přístupu ke stejným systémovým zdrojům, které proces sám využívá. Během maskování přerušení má proces jako jediný práva na používání procesoru, a tudíž žádný jiný proces nemá přístup ke sdíleným zdrojům - sdílené zdroje, které však nesmí být používány více procesy najednou (nazývané i *kritické sekce*) jsou tedy chráněny. Jakmile pak proces přestane využívat kritických sekcí, měl by opět odmaskovat čekající přerušení - ta se v tu chvíli aktivují. Dočasné maskování přerušení může být použito pouze v případech, kdy nejdelší možná doba přístupu ke kritické sekci nepřesahuje maximální požadovanou hodnotu zpoždění přerušení (bude rozebráno v kapitole Zpoždění přerušení). Pokud bude však doba přístupu ke kritické sekci delší, maximální hodnota zpoždění přerušení bude neúnosně navýšena. Metoda blokování přerušení se používá pouze v případě, kdy se v kritické sekci vyskytuje pouze několik řádek instrukcí, které neobsahují iterace. Tato metoda je ideální pro ochranu bitových registrů ovládaných různými úlohami.

3.1.3.2 Binární semafory

Jsou-li kritické sekce delší než jen několik řádek instrukcí, případně obsahují smyčky, potom se musí i RTOS uchýlit k používání mechanismů běžných operačních systémů, jako jsou právě semafory a operačním systémem řízená meziprocesová komunikace. Takové mechanismy zahrnují systémová volání a při ukončení obvykle vyvolávají i kód systémového dispečera - jinými slovy zahrnují stovky instrukcí ke spuštění, zatímco blokování přerušení bývá někdy otázka i jedné instrukce. Ovšem není to dobrá volba, je-li

[5] Pro vestavěné systémy existuje anglický termín *embedded system*. Vestavěné systémy jsou jednoúčelové systémy zabudované do ovládaného zařízení, optimalizované pro konkrétní účely. Příkladem zařízení s vestavěným operačním systémem mohou být mobilní telefony, PDA, bankomaty, kalkulačky a mnoho dalších zařízení.

[6] Mód jádra je také často nazýván systémovým módem. Je to privilegovaný mód spouštění procesů. Je-li proces spuštěn v módu jádra, předpokládá se, že tento proces je důvěryhodný a nemůže poškodit systém. Proto je mu povolen přístup k celé paměti a umožněny spouštět jakékoli instrukce.

třeba zajistit fungování dlouhých kritických sekcí. Přerušení nemohou být blokována delší čas, aniž bychom zvýšili hodnotu zpoždění přerušení.

Binární semafor se nachází v jednom ze stavů: odemčeno a zamčeno. Je-li semafor uzamčen, úloha musí čekat, dokud není semafor odemčen. Binární semafor se řadí k algoritmům, které zajišťují výhradní používání systémových zdrojů, které jsou zkratkou nazývány mutex (z anglického *mutual exclusion*, tedy vzájemné odepření vstupu). K běžným praktikám pro užívání semaforů se řadí nastavení maximálního času čekání (tzv. *timeout*) na odemčení semaforu. Ovšem semafor skýtá i problémy při používání, jako například inverze priority a tzv. uváznutí (*deadlock*).

Při inverzi priority čeká úloha s vyšší prioritou na uvolnění semaforu, který používá úloha s nižší prioritou. Běžným řešením potom bývá spouštět úlohu, která v současné době zamkla semafor na nejvyšší prioritě čekajícího procesu. Ovšem zkomplikuje-li se nám situace ještě více, nemá ani tento přístup požadovaný účinek. Zmíněnou situací je například stav, ve kterém bude proces A čekat na semafor, který má zamčený proces B čekající na semafor, který ovšem je zamčen procesem C. Ošetření vícenásobné dědičnosti bez ztráty stability v cyklech je potom dosti problematické.

K uváznutí dojde, zamknou-li dvě úlohy (nebo více) semafor bez stanoveného času vypršení požadavku a čekají na uvolnění semaforu té druhé úlohy. Takové čekání je ve výsledku nekonečné - vytváří se cyklická závislost. Nejjednodušší uváznutí nastává, budou-li dvě úlohy střídavě zamykat dva semafory ve vzájemně odlišném pořadí. Uváznutím se snaží systém zabránit svým návrhem, případně pomocí semaforů, které za určitých podmínek dávají kontrolu nad semaforem úlohám s vyšší prioritou.

3.1.3.3 Zasílání zpráv

Dalším přístupem ke sdílení procesů je zasílání zpráv v organizovaném schématu zasílání správ. Podle tohoto paradigmatu je sdílený zdroj samotný řízený právě jedním procesem (úlohou, vláknem), který od ostatních přijímá požadavky na manipulaci se sdíleným zdrojem. I když je v tomto případě chování v reálném čase méně řízné, než například u semaforů, jednoduchý systém zasílání zpráv zpravidla zabrání většině rizik spojených s uváznutím. Nicméně jim není schopen zabránit zcela. Inverze priorit se může objevit, pracuje-li úloha na zprávě s nižší prioritou a ignoruje zprávu s vyšší prioritou nebo zprávu pocházející nepřímo od úlohy s vyšší prioritou ve své frontě zpráv ke zpracování. Nastat

může však i uváznutí - typicky pokud například dvě úlohy čekají vzájemně na odpověď druhé úlohy.

3.2 Úlohy vhodné pro RTOS

Obecně RTOS poskytují možnosti pro vytvoření deterministického systému. Samozřejmě, k tomu, aby bylo možno takový systém vytvořit, nepotřebujeme nutně RTOS, avšak realizace systému pak bude čistě na nás, bez podpory mechanismů, které nám umožňují RTOS.

Extrémní determinismus není zpravidla v operačních systémech nutný. RTOS se však mohou jevit výhodnější pro nejrůznější úlohy v porovnání s běžnými operačními systémy. RTOS a zvláště pak vestavěné operační systémy mohou nabídnout stabilní prostředí, ovladače různých zařízení, systém souborů, síťové služby nebo jiné systémové služby v příznivější variantě než běžné operační systémy.

Pro RTOS jsou vhodné zejména úlohy vyžadující jistotu včasné reakce - tedy úlohy pro tzv. *hard real-time* systémy. Mezi ty se řadí zejména takové, u kterých selhání nebo pozdní zpracování vstupů může mít za následek nenapravitelné ztráty na životech či životním prostředí.

3.3 Kvantifikace operačních systémů pro zpracování úloh v reálném čase

Bude-li existovat několik RTOS, které vyhovují požadavkům aplikace na operační systém, potom veškerá další specifika těchto systémů vytváří rozdíl mezi nimi. Většinou jsou v takovém případě třeba kvalitativní či kvantitativní charakteristiky a měřitelné parametry. Volně zpracováno podle [2], [9]

3.3.1 Čas odezvy

Důležitou veličinou pro ohodnocení RTOS je právě čas odezvy. Výkonné RTOS prodlužují svou teoretickou minimální odezvu pouze nepatrně. Je třeba připomenout, že hodnoty uvedených veličin, které jsou zde srovnávány, nepředstavují průměrné hodnoty. Vždy se bude jednat o nejhorší možné časy odezvy.

Následující podkapitoly budou rozebírat typické parametry náležející do této kategorie.

3.3.1.1 Zpoždění přerušení

Zpoždění přerušení (angl. *interrupt latency*) je nazýván časový úsek, který uplyne od požadavku na přerušení do jeho obsloužení. Jednotlivé RTOS však mohou skutečný čas obsloužení přerušení oproti očekávání prodlužovat. Větší hodnota zpoždění přerušení může být způsobena implementací RTOS v oblasti ovladačů přerušení nebo v jiných důležitých částech implementace RTOS.

3.3.1.2 Čas překlopení

Důležitým kvantifikátorem RTOS je časový interval, za který je systém reagovat na hardwarovou (např. přerušení) nebo systémovou událost (např. zařazení nové položky do struktury připravených procesů[7]) a nastavit stav procesu s nejvyšší prioritou na "spuštěný". V případě hardwarových přerušení bývá zpravidla prioritní úloha ta, která přerušení zpracovává. Tomuto času se říká kritický čas odezvy[8], nebo též čas překlopení[9] (angl. *flyback time*).

3.3.1.3 Čas potřebný pro přepnutí kontextu

Touto charakteristikou je myšlen čas potřebný pro synchronní přepnutí z kontextu jednoho procesu (úlohy, vlákna) do kontextu jiného procesu. V anglické literatuře je označován jako *context switch time*.

3.3.2 Výkyvy

Dobrý RTOS je mimo jiné charakterizován i nízkou hodnotou výkyvů v čase odezvy. Mezi faktory, které významně ovlivňují chování systému, s ohledem na výkyvy v čase odezvy patří přiřazování priorit procesům (úlohám, vláknům), přiřazování priorit přerušení, délka a počet kritických zón (tedy použití nesimultánně sdílených zdrojů), vzájemné meziprocesové interakce přes sdílené zdroje a použití algoritmů nebo strategií pro redukci výkyvů.[10]

---

7 Dobře navržené RTOS potřebují k zařazení nové položky do seznamu 3 až 20 instrukcí, a dalších 5 až 30 instrukcí na obnovení statutu procesu s nejvyšší prioritou.

8 z anglického termínu *critical response time*

9 z anglického termínu *fly-back time*

10 Problematika výkyvů je velmi obsáhlá a její podrobnější vysvětlení ponecháme např. na [9]

3.3.3 Velikost

Mnohdy je důležitým kritériem pro rozhodování o vhodnosti systému i jeho velikost (resp. velikost paměti, kterou operační systém zabere pro své fungování). Důležitá je tato veličina zejména u vestavěných systémů, kde hraje roli cena paměťových bloků obsluhovaného systému.

3.3.4 Spolehlivost

Jisté detaily v návrhu činí jisté systémy vnitřně spolehlivější, než jiné. Dobrým příkladem může být dynamická alokace - lze se v ní setkat jak s nespolehlivostí, tak i s nepředpověditelností odezvy při určitých alokačních operacích. Čistě statické návrhy tato vnitřní omezení nemají.

3.3.5 Synchronizační primitivy

Dostupnost a různorodost systémových primitiv (tedy atomických systémových operací) je také důležitý faktor k posouzení. Je-li totiž k dispozici vhodný systémový nástroj, nebude třeba vyvíjet vlastní. Přidanou hodnotou je pak i snazší integrace externího kódu do prostředí RTOS.

Dobrým příkladem mohou být funkce operačního systému umožňující práci s časovači. Není třeba vyvíjet vlastní funkce pro jejich nastavování a obsluhu, jsou-li k dispozici volání podporovaná přímo operačním systémem - zejména jsou-li otestovaná a fungující.

3.4 Standardy POSIX

Problémem operačních systémů obecně bývá, že každý z nich zpravidla používá vlastní proprietární API[11]. POSIX[12] v této situaci roztříštěného trhu vytváří standard, který umožňuje alespoň částečně systémy unifikovat. [2], [7]

3.4.1 POSIX a operační systémy

Standard POSIX.1 (IEEE 1003.1 - 1988) se zaobírá mechanismy operačního systému, které by měly vést k lepší přenositelnosti mezi jednotlivými systémy a určuje základní služby jádra. Mezi ně řadí vytváření a správu procesů, signály, výjimky operací s plovoucí

[11] z anglického termínu *Application Programming Interface* neboli rozhraní pro programování aplikací

[12] zkratka *Portable Operating System Interface (for Unix)*

čárkou, chyby segmentace a paměti, nepřípustné instrukce, chyby sběrnic, časovače, operace se souborovým systémem, roury, rozhraní I/O portů a jejich správa, knihovny jazyka C a spouštěče procesů. Jedná se o rozsáhlé téma, které je mimo rámec této práce. (více informací lze najít v [7])

3.4.2 POSIX a reálný čas

Mladší norma POSIX.4 (IEEE 1003.1b - 1993) se zabývá rozšířením mechanismů normy POSIX.1 na použití v reálném čase. Řeší otázky prioritního plánování, signálů v reálném čase, času a časovačů, semaforů, mechanismů pro zasílání zpráv, sdílené paměti, asynchronních a synchronních vstupů a výstupů a rozhraní pro uzamčení paměti.

4. Studie vybraného operačního systému

Následující kapitola bude z hlediska použití v programování aplikací pracujících v reálném čase rozebírat operační systém QNX.

Systém QNX je relativně rozšířený, stále vyvíjený a podporovaný operační systém nasazovaný pro řešení široké palety úkolů. Velmi zajímavé je jeho použití jako vestavěného systému - to ostatně poslední dobou převažuje. Samotní vývojáři systému i programátorská komunita se podílí na propracované dokumentaci systému a jeho skvělém popisu. Z výše uvedených důvodů padla při rozhodování volba právě na tento RTOS.

Tento operační systém pro práci v reálném čase je založen na systému UNIX a splňuje normu POSIX. Jeho jádro není jako u většiny operačních systémů monolitické - tedy není to jednolitý blok starající se o všechny systémové služby. Místo toho QNX tvoří mikrojádro a soustava procesů zajišťující vyšší systémové služby. Podrobněji bude téma architektury QNX rozebráno v samostatné kapitole.

4.1 Architektura QNX

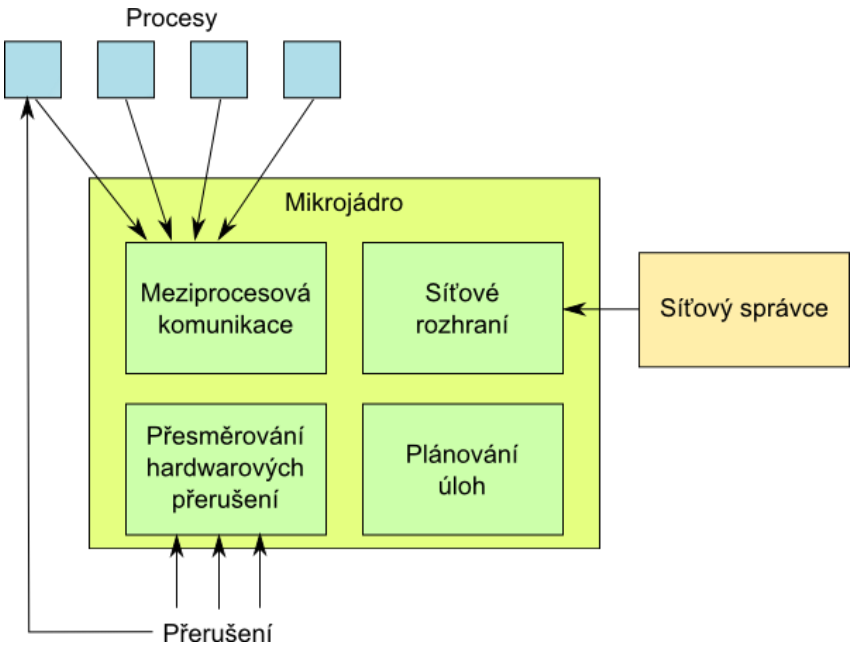
Již od počátku byla architektura systému QNX velmi podobná té současné - středobodem systému bylo velmi malé mikrojádro, obklopené vzájemně spolupracujícími procesy, které zajišťují přístup k souborovému systému a různým druhům zařízení. Tato architektura se prokázala být natolik robustní, že byla nasazována na systémech vyžadujících rychlou reakci na podnět a zároveň maximální flexibilitu. [3], [4], [5]

4.1.1 Mikrojádro

Nejprve je třeba vysvětlit termín *mikrojádro*: mikrojádrem bude myšleno takové jádro operačního systému, které sdružuje pouze minimální potřebnou sadu systémových funkcí. Obecně těmito funkcemi jsou zpravidla správa paměti, podpora plánování procesů a meziprocesová komunikace. Ostatní služby bývají u tohoto typů jádra zpravidla řešeny uživatelskými procesy. Monolitické jádro, oproti tomu, řeší všechny systémové služby uvnitř samotného jádra.

Mikrojádro systému QNX v sobě zahrnuje pouze čtyři základní služby - meziprocesovou komunikaci (IPC - z anglického *interprocess communication*), základ pro síťovou komunikaci, plánování procesů (*process scheduling*) a řízení přerušení (*interrupt dispatching*).

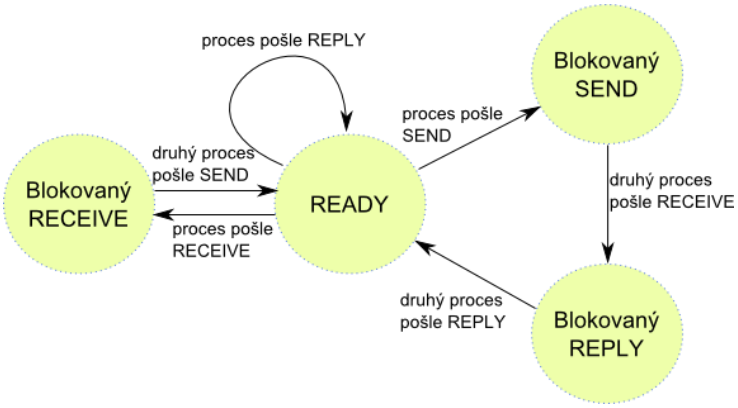
Obrázek 1: Schéma mikrojádra



Zdroj: [5], upraveno

Meziprocesová komunikace (dále už je IPC) je v systému QNX realizována pomocí blokujících příkazů *Send*, *Receive* a *Reply*. Blokující jsou příkazy nazývány, protože po odeslání požadavků *Send* a *Receive* blokují další běh procesu. Proces je opět odblokován po přijetí odpovědi dotazovaného procesu. Schematicky lze stavy procesu zobrazit následujícím nákresem:

Obrázek 2: Následnost stavů procesů



Zdroj: [5], upraveno

Procesům mohou být zprávy doručovány i v pořadí podle priority místo toho, aby jim byly doručovány v pořadí, v jakém byly procesu doručeny - to pak má za následek rychlejší zpracování prioritních úloh. Zasíláním zpráv podle priority je možné se vyhnout *problému inverze priority*[13], ke kterému může dojít v systémech se zasíláním zpráv s fixní prioritou.

Primitivy zasílání zpráv podporují zasílání zpráv ve více částech. Zprávy zasílané jedním procesem druhému tedy nemusí zabírat souvislý blok paměti. Místo toho mohou být údaje o faktickém umístění fragmentů zprávy v paměti obsaženy jak na straně zasílajícího, tak na straně přijímajícího procesu v MX tabulce. Výsledkem je zasílání zpráv nezatěžující systém zbytečným kopírováním

Do mikrojádra je navíc implementována i základní síťová komunikace zprostředkovaná síťovým správcem (bude diskutován v samostatné kapitole). Pokud je služba síťového správce (*Network manager*) dostupná, je napojena přímo na mikrojádro a umožňuje mu komunikaci s procesy spuštěnými jiným mikrojádrem v lokální síti. Díky tomuto typu komunikace je pak libovolnému procesu v síti transparentně přístupný jakýkoli jiný proces v téže síti.

Primitivy plánování procesů systému QNX odpovídají specifikaci POSIX.4 (rozšíření pro použití v reálném čase). Systém QNX poskytuje plně preemptivní, přepínání kontextu podle priority s plánováním *round robin, FIFO* a *adaptivním plánováním*.

[13] problém inverze priority je scénář plánování procesů, ve kterém úloha s vyšší prioritou je nepřímo přepnuta úlohou s nižší prioritou tak, že se chovají, jakoby se jejich priority přehodily

Aby QNX splňoval požadavky POSIX a konvence UNIX, mohou být mikrojádrem volitelně spuštěny další procesy, které tuto funkcionalitu umožňují. Tyto procesy jsou nazývány správci zdrojů a starají se o procesy, souborový systém nebo jiná I/O zařízení. V minimální formě tak systém umožňuje spustit pouze mikrojádro, správce procesů a aplikační procesy.

4.1.2 Správce procesů

První a jediný povinně spouštěný manažerský proces je správce procesů[14]. Umožňuje vytvářet procesy, vést údržbu procesů[15], spravovat paměť, spravovat dědičnost prostředí procesů (jak pro lokální, tak i pro procesy v lokální síti) a správu cest[16]. Právě správa cest musí být správcem procesů zajišťována, protože na rozdíl od monolitických systémů, které vždy pracují se systémem souborů, u QNX je systém souborů nepovinný.

Dokud se nespustí ostatní správci zdrojů, jsou veškeré dotazy na cestu směrovány správci procesů a každý následně spouštěný správce následně registruje cestu, kterou bude spravovat. Ovšem, správce procesů má ten důležitý úkol spravovat celý strom prefixů, pomocí kterého lze vystopovat správce zdrojů, který řeší dotazy na danou cestu.

Po spuštění správce souborového systému (`Fsys`) a správce zařízení (`Dev`) pak může strom prefixů vypadat například takto:

Tabulka 1: Příklad stromu prefixů spravovaných správcem procesů

`/`	systém souborů na disku (`Fsys`)
`/dev`	systém zařízení (`Dev`)
`/dev/hd0`	hrubý diskový svazek (`Fsys`)
`/dev/null`	null device (`Dev`)

Zdroj: [5], str. 5

Otevře-li tedy proces soubor, zašle rutina *open()* jméno souboru správci procesů, který ho vyhodnotí a předá správci, jehož nejdelší registrovaný prefix odpovídá jménu souboru. Pokud by bylo například vyžadováno otevření `/dev/tty0`, bude nejdelším odpovídajícím prefixem `/dev`. Požadavek na otevření bude předán správci zařízení `Dev`.

[14] z anglického termínu *process manager*

[15] v literatuře zmiňovaná i jako tzv. *process accounting*

[16] v literatuře zmiňovaný i jako tzv. *pathname management*

Oproti tomu, cestě `/usr/vz` bude jako nejdelší odpovídat prefix `/` a požadavek bude předán správci souborového systému `Fsys`.

Správce procesů na každém počítači v síti udržuje vlastní hierarchický strom prefixů. Ovšem, z pohledu jednotlivých správců procesů může být pohled na jednotlivé cesty v místní síti značně odlišný. Každopádně, cesty začínající znakem `/` jsou vyhodnocovány vždy v lokální prefixové tabulce. Unikátní jména v síti umožňují specifikovat absolutní umístění zdroje procesům v celé síti. *Aliasing prefixů* může zase mapovat část jmenného prostoru prefixů na správce zdrojů uzlu v místní síti.

To lze například využít v terminálu bez disku, který spouští systém ze sítě - u takového terminálu bude žádoucí, aby kořen systému souborů byl mapován na vzdálený uzel, resp. na jeho správce souborového systému. Bude-li však třeba volat *open()* na `/dev`, i nadále bude zpracován místním správcem zařízení `Dev`, zatímco volání open() na soubory budou řešeny prefixovou tabulkou předdefinovaného vzdáleného uzlu. Důsledek výše uvedených skutečností je, že každý proces v místní síti může přistupovat ke každému zdroji v systému souborů libovolného uzlu.

Tím, že se zodpovědnost za části běhu systému přesouvá na spouštěné procesy (správce) vzniká zajímavá vlastnost systému QNX - správce zdrojů nejsou součástí jádra a jsou spouštěny v základu podobně jako uživatelské procesy, mohou být přidávány a odebírány dynamicky za běhu systému, aniž by bylo třeba měnit nastavení jádra. Tato flexibilita dává možnost nastavitelnosti systému potřebám aplikací.

I když se tato architektura jádra systému zdá být neefektivní, testy výkonu v porovnání s monolitickým systémem UNIX jasně hovoří ve prospěch přepínání kontextu a meziprocesové komunikace mikrojádra. Výkonnost hardwaru tak může být znatelně lépe využita.

Transparentnost tohoto systému jmenných prostorů logicky staví spouštění vzdáleného a spouštění lokálního procesu na stejnou úroveň. Adresování cest jednotlivými správci procesů v síti se nekonfliktně slévá - v chování prostoru jmen nedochází k překryvům.

4.1.3 Správce souborového systému

Správce souborového systému[17] (dále jen `Fsys`) je správce zdrojů, který zprostředkovává systému QNX souborový systém podle specifikací normy POSIX. Implementuje diskovou strukturu používající bitmapu pro alokaci volného místa na disku a přístup seznamu rozsahů k organizaci dat na disku.

`Fsys` vykonává synchronní zápis životně důležitých systémových datových struktur a tím umožňuje disku pohodlně překonat případný výpadek napájení. Katastrofickým stavům zabraňuje i sada značek zapisovaných v datových strukturách. Pomocí nich je pak možné v krajních situacích disk znovu přestavět do funkční podoby.

Vícevláknová architektura správce `Fsys` umožňuje i paralelní zpracování více požadavků najednou (například vstup a výstup na RAM disku nebo paměti cache, zatímco ostatní vlákna jsou blokována čekáním na fyzický vstup/výstup). Tento paralelismus se týká i ovladače samotného zařízení - je-li zařízení schopno zpracovat více čekajících požadavků na vstup/výstup, potom i ovladač sám umožňuje zpracovávat požadavky v jakémkoli pořadí.

Mohlo by se zdát, že meziprocesová komunikace systému QNX bude nadměrně zatěžovat souborový systém neustálým kopírováním a přesouváním datových bloků, avšak ve skutečnosti nejsou tyto operace vůbec třeba. Primitivy umožňující zasílání zpráv o více částech pomocí MX tabulek umožňují správci `Fsys` mapovat souvislé bloky vyžadované aplikacemi pomocí nesouvislých bloků cache paměti ve správci `Fsys`.

Proces správce souborového systému `Fsys` může být spuštěn z řádky i na počítači bez disku s připojením do místní sítě a ovladače zařízení mohou být dynamicky připojovány na `Fsys`. Naopak, není-li již nadále třeba služeb poskytovaných správcem Fsys, je možné jej i včetně ovladačů odstranit z paměti.

4.1.4 Správce zařízení

Správce zařízení[18] (dále označovaný jako `Dev`) zprostředkovává řízení nejrůznějších prostředků podle norem POSIX s několika rozšířeními nutnými pro zajištění komunikace v reálném čase. Stejně jako `Fsys`, `Dev` může být podle libosti spuštěn a jeho ovladače připojeny, případně později odstraněn z paměti, pomine-li potřeba jeho služeb.

[17] z anglického termínu *filesystem manager*

[18] z anglického termínu *device manager*

`Dev` je schopen pracovat s rychlostmi až 115Kbaudů při použití běžného hardwaru s neinteligentními UART[19] zařízeními, to zejména díky nízkému zpoždění (latenci) přerušení poskytovaného mikrojádrem. S využitím inteligentních komunikačních karet s vysokou hodnotou šířky pásma je pak možné nakonfigurovat i komunikační server.

4.1.4.1 Podpora ovladačů zařízení

QNX umožňuje systémová volání dávající uživatelským procesům možnost připojit dostatečně privilegovaný uživatelský proces k příslušnému vektoru přerušení uvnitř jádra. Připojený handler (též obsluha) potom bude jádrem volán jako důsledek fyzického přerušení. Díky tomu, že existuje jako uživatelský proces má pak možnost přístupu k adresovému prostoru procesu jako reakci na přerušení. Jakmile je handler spuštěn, může buď oživit proces, s kterým sdílí kód, nebo být navrácen jádru. Jednotlivé ovladače zařízení pak z tohoto chování čerpají výhody, které jim umožňují kumulovat zprávy v zásobníku spravovaném správcem `Dev`, a vzbudit jej pouze v případě, že se vyskytne předdefinovaná důležitá událost (jako naplnění počtu znaků na terminálu, událost nové řádky nebo vypršení čekací lhůty).

Vyskytuje-li se obsluha přerušení mimo samotné jádro, uživatel může snadno přidávat nebo ubírat handlery přerušení (a ovladače zařízení, které je obsahují) na běžícím systému. Vnitřní řešení přerušení v jádru je schopné vypořádat se s vnořením a sdílením přerušení aniž by jakkoli negativně ovlivnilo samotnou uživatelskou obsluhu přerušení. Podpora externích handlerů přerušení tvoří základ schopnosti jednotlivých správců zdrojů dosáhnout stejného výkonu jako správa zdrojů v monolitickém jádře.

4.1.4.2 Možnosti rozšíření

Velká výhoda ovladačů existujících na úrovni uživatelských procesů je, že vývoj rozšíření systému se nijak funkčně neliší od vývoje uživatelských procesů. Není proto nutné oproti monolitickým jádrům používat specializované nástroje na sledování a odstraňování chyb jádra při případných změnách v kódu služeb. Postačí použití běžného nástroje pro sledování chyb (tzv. *debuggeru*). Vzhledem k tomu, že i správci zdrojů nebo ovladače

[19] z anglického termínu *universal asynchronous receiver-transmitter,* hardware překládající data mezi sériovou a paralelní formou

mohou být za běhu systému přidávány podle potřeby, stává se nutnost opětovného sestavení a následného spuštění jádra zcela bezpředmětnou.

4.1.5 Síťový správce

Jak již bylo v předešlém textu zmíněno, síťový správce[20] (v dalším textu už jen `Net`) je přímo propojen s mikrojádrem. Je-li pak mikrojádro vyzváno k poslání zprávy od procesu, umístěném v místním uzlu k procesu běžícím na jiném uzlu v síti, předá správci `Net` přes vzájemně sdílené rozhraní ukazatel na zprávu. Obdobně pak `Net` může obdržet zprávu od jiných mikrojader a stejným způsobem ji předat místnímu mikrojádru. V podstatě tvoří společně mikrojádra propojená síťovými správci jedno velké mikrojádro. A jelikož všechny systémové služby (vytváření procesů, debugging, vstup a výstup) jsou zprostředkovány pomocí zasílání zpráv přes mikrojádro, výsledkem je síť strojů, které se navenek chovají jako jediný počítač.

Typickým příkladem může být spuštění příkazu:

```
ls /usr/vz | grep abc | wc
```

Ten spustí jeho jednotlivé části na různých procesorech v síti - deskriptory zprostředkované správcem `Proc` způsobí, že roury v příkazu přesměrují data sítí k příslušnému počítači.

Obdobně jako `Fsys` nebo `Dev` i `Net` může být libovolně spouštěn z řádky nebo ukončen, aniž by ovlivnil celkový běh systému. I tento správce má svou sadu ovladačů a podporuje připojení více síťových ovladačů k správci `Net`. Stane-li se, že Net zjistí, že spojení s uzlem je možné realizovat pomocí více než jednoho síťového ovladače, automaticky bude vyrovnávat provoz v jednotlivých ovladačích. Chování síťového rozhraní je možno případně manuálně korigovat s použitím řádkových voleb při spuštění správce `Net`.

Používání více variant propojení jednotlivých síťových uzlů vede k vyšší propustnosti sítě a vyšší toleranci k chybovým stavům - nelze ho než doporučit. Navíc není třeba aplikační procesy jakkoli přizpůsobovat, aby byly schopny těchto výhod více připojení využívat. O vše se postará síťový správce.

[20] z anglického *Network manager*

Net je schopen použití nenákladné sériové linky jako nouzového nebo záložního řešení pro případ výpadku sítě. Pohotovost takového řešení se projeví při použití ve vysokých hodnotách rychlosti (`Dev` je schopen až 115 Kbaud), datové kompresi a povolení použití vyrovnávací paměti obsahující klientský systému souborů. Obdobně lze tohoto řešení použít v případech přetížení místní sítě, ke které může dojít například, probíhá-li replikace dat mezi dvěma souborovými servery. Při použití výše popsaných síťových služeb prezentovaných pod názvem FLEET[21] je možné přidat soukromou linku zabezpečující pouze tuto replikaci, která nebude zatěžovat ostatní provoz v síti. V případě, že jsou servery navíc fyzicky blízko u sebe, je možné použít i nekonvenčních řešení jako přemostění pomocí SCSI nebo DMA[22] mezi sběrnicemi. Zajímavá pak mohou být i řešení používající služeb VLAN[23], díky kterým lze například vytvořit distribuovaný víceprocesorový systém. Tím možnosti použití správce `Net` ovšem nekončí.

4.1.6 Údržba

Běžným problémem monolitických systémů bývá sdílený prostor v paměti, o který se všechen kód jádra dělí. Ovšem v QNX jsou stanovena pouze rozhraní, pomocí kterých jednotlivé komponenty systému komunikují pomocí mechanismů meziprocesové komunikace. Každá komponenta navíc využívá pouze vlastní prostor adres - chyby v jedné komponentě se tedy případně nepromítnou do chodu celého systému. Architektura systému navíc umožňuje jednoduchou správu i přidávání dalších komponent.

[21] FLEET je zkratka pro *Fault-tolerant Load-balancing Efficient Extensible Transparent*, tedy chyby tolerující, vyrovnávající zátěž, efektivní, rozšiřitelný a transparentní.

[22] z anglického termínu *Direct Memory Access*, tedy přímý přístup k paměti

[23] Pozor, v tomto případě zkratka VLAN znamená *Very Local Area Network*.

5. Tvorba aplikací pro práci v reálném čase

Aplikace pracující v reálném čase vždy interaguje s vnějším světem - vyčítá aktuální hodnoty na vnějším zařízení, reaguje na tyto hodnoty, zpracovává vstup uživatele a kontroluje odezvu na předchozí akce. Typickým příkladem může být řídící systém proudového letounu, který kontroluje stav paliva, tlak v palivových nádržích, tah a optimální spotřebu proudových motorů, reakce autopilota na vnější podmínky a konečně i zásahy samotných pilotů do řízení. Volně zpracováno podle [2], [6], [10]

5.1 Proces v reálném čase

Účelem aplikace pro práci v reálném čase je reagovat na množství různých vstupů "dostatečně včas" různými výstupy. Tyto výstupy pak velmi často ovlivňují hodnoty příští sady vstupů. Struktura aplikace zpracovávající vstupy v reálném čase pak může být různá:

- Jeden proces zpracovávající vše (Obsluha všech vstupů i výstupů je realizována uvnitř jednoho procesu. Počet procesů se sice zbytečně nenavýší, ale proces se mnohdy stává zbytečně složitým. Údržba takového procesu může být velmi obtížná a pouhá změna hardwaru může přivodit pád aplikace.)

- Mnoho specializovaných procesů (Každý vstup i výstup, každá celistvá úloha aplikace je realizována vlastním procesem. Složitost jednotlivých úloh se snižuje, avšak vyvstává otázka koordinace úloh jednotlivých procesů. Nadto, každý proces zabírá jisté systémové prostředky a proto mimo jiné není možné vytvořit nekonečný počet procesů. I když bude ve skutečnosti stačit méně procesů, zpravidla se v tomto extrémním případě počet procesů nebezpečně blíží maximální hodnotě počtu procesů.)

- Použití rozumného množství procesů (Kompromis mezi dvěma předchozími alternativami. Jeden proces už může zvládat více úkolů, avšak je nutno dát pozor, aby složitost jednotlivých procesů nepřerostla únosnou mez. Ta totiž mnohdy ukrývá chyby, což je pro kritické RTOS nepřípustné.)

- Emulace více procesů pomocí signálů (Obsluha přerušení, neboli handler může být vnímán i jako nezávislý asynchronní tok předávání kontroly nad vykonáváním úloh uvnitř procesu. Předání je pak vždy vynuceno jako odpověď na odpovídající signál přerušení. Je-li možné navázat vstupy a výstupy procesu na signály přerušení,

potom vlastně existuje rozumný mechanismus jak emulovat chování více procesů, aniž bychom systém ve skutečnosti více procesy zatížili.)

- Použití vláken (Vlákna představují způsob kterým umožnit řešení více úloh v rámci jediného procesu, resp. jeho prostoru adres. Pomocí vláken je možno efektivně řešit více souvisejících úloh.)

5.1.1 Proces v nekonečné smyčce

Koncept procesu v nekonečné smyčce vychází z běžného algoritmu neustálého opakování kontroly portu na vstup a zpracování případného vstupu. V nejjednodušším případě může mechanismus procesu vypadat například takto:

```
while (1)  {
     /* nacteni udaju z portu */
     /* zpracovani nactenych udaju */
     /* vyslani odpovedi */
}
```

Tato koncepce procesu v reálném čase je nazývána *cyklické spouštění*[24]. Je možné ji využít zejména pro jednoduché aplikace v reálném čase, zvláště pak v takových, kde nároky na odezvu nejsou příliš omezující a kde úlohy běžící ve smyčce na sebe vzájemně navazují.

Problémem tohoto pojetí také je, že se nekonečná smyčka bude opakovat v co nejrychleji po sobě následujících iteracích, přičemž na systému mohou běžet i další úlohy, které však tímto přichází o výpočetní čas. Většinou je požadováno spouštět smyčku na nějaké frekvenci, za předpokladu synchronizovaných úloh ve smyčce. Ve skutečnosti však není možno spouštět cyklus ani v takové frekvenci, v jaké bychom chtěli. Nejvhodnějším se jeví použití nejmenší možné frekvence, při které bude program ještě hladce fungovat.

Dalším problémem je však i samotná synchronizace úloh uvnitř smyčky - ve skutečnosti je tím myšleno, že je třeba, aby se jednotlivé úlohy ve smyčce spouštěly na harmonických frekvencích. Bude-li například úkolem vytvořit proces vyčítající z klávesnice údaje 60x za vteřinu a zobrazující je na obrazovku 30x za vteřinu, bude hlavní smyčka spouštěna na frekvenci 60Hz a údaje bude zobrazovat pouze v každé druhé

[24] V anglicky psané literatuře pak *cyclic executive approach*.

smyčce. Horší situace nastane, budou-li frekvence přijímání a zobrazování vzájemně nesoudělné (neharmonické) - bude potom třeba zvýšit frekvenci spouštění smyčky a to v mnoha případech není možné nebo vhodné. Další problémy pak přijdou se zvětšující se komplexností úloh ve smyčce - složitější úlohy je pak mnohdy třeba rozdělit tak, aby byly vykonávány i během několika iterací.

Ladění procesu s cyklickým spouštěním je ošemetnou záležitostí, která zahrnuje přesouvání úloh, jejich dělení, zpracování přes několik iterací a optimalizaci pro konkrétní hardware.

5.1.2 Emulace multitaskingu pomocí obsluhy signálů

Alternativou k nekonečným smyčkám může být obsluha signálů. POSIX signál je softwarovou analogií hardwarového přerušení - obdrží-li proces signál, okamžitě opustí úlohu, kterou vykonává a spustí obslužnou funkci (*handler*), který signálu přísluší. Po dokončení zpracování signálu se proces vrátí k rozdělané úloze.

I když se při použití této koncepce může zdát, že se proces chová jako multitasking, není to bohužel pravda. Handler signálu totiž nemůže být synchronizován s dalšími úlohami simulovanými signálem - jiné úlohy totiž ve skutečnosti ani neexistují. A zablokuje-li se handler signálu, zablokuje se i celá aplikace. Signály mohou nicméně být úspěšně využity v mnoha úlohách reagujících na externí vstupy.

5.1.3 Zpracování úlohy více procesy

Není nutné potýkat se s problémy koncepce jediné smyčky a multitasking emulovat, když systém tuto službu nabízí. Řešená úloha se nám pak rozpadá na několik nezávislých smyček, které řeší pouze svou část úlohy. Mezi výhody tohoto zpracování patří v první řadě jednoduchost a tedy i robustnost, dále pak i lepší přenositelnost, škálovatelnost, modularita a bezpečnost.

Ovšem každý proces, který takto bude pro potřeby řešení úlohy vytvořen, zabírá své místo v paměti, zaměstnává systém při plánování procesů a hlavně, musí být koordinován s ostatními procesy, které zpracovávají úlohu. Mechanismy, jimiž je řízen život procesu, jsou popsány normou POSIX.1 a nejsou touto prací diskutovány.

5.1.4 Signály

Signály jsou nedílnou součástí multitaskingu v prostředí UNIX/POSIX systémů. Signály se používají pro *ošetření výjimek, upozornění procesu na asynchronní událost, abnormální ukončení procesu, emulaci multitaskingu* a *meziprocesovou komunikaci*.

Jak již bylo řečeno, signál je softwarový ekvivalent přerušení, nebo výskytu výjimky. Dostane-li proces signál, znamená to, že se stalo něco, co vyžaduje pozornost procesu. Procesu bude předán signál v případě, že dojde k výjimce (synchronně generované signály - následující okamžitě po něčem, co proces provede), stane-li se událost asynchronní k spuštěnému procesu (vypršení časovače, dokončení vstupu nebo výstupu, přerušení běhu procesu stiskem CTRL-C apod.), nebo rozhodne-li se jiný proces explicitně signál sledovanému procesu poslat.

Jako možnost použití byla sice uvedena meziprocesová komunikace, avšak ve skutečnosti jsou pro ni signály nevhodné. Jsou pomalé, limitované, mechanismus jejich implementace vhodně neřeší řazení signálů a samozřejmě, přeruší běh procesu způsobem, který je třeba vhodně ošetřit. Nicméně v některých případech mohou být signály pro meziprocesovou komunikaci použitelné. Pro jejich použití v meziprocesové komunikaci mluví jejich rozšířenost na systémech UNIX a koncepce asynchronnosti (signál může proces obdržet, zatímco vykonává úkol) - proces obdrží signál okamžitě a nezávisle na aktuálně zpracovávaném úkolu. Proti jejich použití však mluví nutnost připravit proces na výskyt signálu kdekoli v běhu procesu, skutečnost, že signály nám kazí deterministickou povahu našeho procesu a jejich nižší výkonnost oproti synchronní komunikaci.

Použití signálů v asynchronní komunikaci s sebou přináší i další úskalí:

- nedostatek různých druhů signálů pro použití v komunikaci - norma POSIX.1 definuje pouze dva uživatelské signály SIGUSR1 a SIGUSR2
- neexistuje řazení signálů do fronty - dostane-li proces signál ve chvíli, kdy obsluhuje předchozí signál, překryjí data nového signálu data toho starého
- pořadí doručování signálů není stanoveno
- informační obsah signálů je minimální
- asynchronnost je mnohdy problémem

Se signály lze naložit třemi následujícími způsoby: signály je možno ignorovat, blokovat nebo obsloužit. Všechny způsoby nakládání se signály řeší norma POSIX.1.

5.1.4.1 Rozšíření signálů pro reálný čas

Norma POSIX.4 ponechává v platnosti signály určené normou POSIX.1 a definuje novou sadu signálů. Díky tomu může zdrojový kód napsaný podle běžných standardů fungovat, zatímco využití nových mechanismů přináší výhody pro reálný čas.

Nové signály mohou být zpracovány ve frontě a přenášet více informací nastavením bitu SA_INFO ve struktuře `sigaction`. Handlery obsluhující tyto signály pak musí dostávat více parametrů - mimo normou POSIX.1 definovaného identifikátoru signálu `signum` i položky `data` a `extra`. `Extra` odkazuje na kontext počítače v době, kdy nastal signál a `data` umožňuje předat signálem jistá data navíc. Navíc jsou tyto signály předávány pomocí funkce `sigqueue`, nikoli `kill`.

Rozšíření v reálném čase navíc umožňují i řazení signálů do fronty a jejich zpracování podle priority, kde coby priorita vystupuje číslo signálu - nižší číslo bude mít vyšší prioritu.

Avšak i použití mechanismu rozšířených signálů nezbavuje signály nevýhod pro použití v komunikaci. Signály i nadále zůstávají příliš pomalé pro doručení, asynchronnost této komunikace způsobuje problémy s robustností a nízká schopnost přenosu dat (signál může nést jen asi 32bitů dat). Avšak asynchronnost je i velkou výhodou, stejně jako nízké nároky na nastavování, adresnost signálů a málo restrikcí v použití.

5.2 Koordinace procesů, zprávy, sdílená paměť a synchronizace

Komunikací, nebo koordinací procesů bude vždy myšlena meziprocesová komunikace. Tu je možné realizovat předáváním zpráv, sdílenou pamětí, semafory, mutexy[25], zámky čtení a zápisu nebo příznaky[26]. V dalším textu opět nebude rozebírána komunikace specifikovaná normou POSIX.1, ale její rozšíření pro reálný čas.

5.2.1 Řazení zpráv v reálném čase

Řady zpráv fungují v jádru podobně jako roury (*pipes*), existuje u nich vždy zapisující a přijímající strana. Nicméně, řada zpráv musí být samozřejmě více strukturovaná, vzhledem k tomu, že obsahuje mechanismus řízení priority a samotné diskrétní zprávy.

[25] zkrácený tvar anglického výrazu *mutual exclusion*, tedy *vzájemné vyloučení*, avšak v dalším textu bude používán i nadále výraz *mutex*

[26] v anglické literatuře označovány jako *flags*

Řadu je třeba vytvořit a otevřít, stejně jako u zápisu do souboru, přičemž je třeba i definovat její vlastnosti, jako je jméno fronty, maximální velikost fronty čekajících zpráv, a jiné její parametry.

Zprávy v řadě jsou popsány cestou, která se nápadně podobá cestě v systému souborů. Posílání a čtení je pak uskutečněno mechanismem, který se opět nápadně podobá zápisu nebo čtení do souboru. Komplikací nicméně zůstává například maximální délka fronty - přesáhne-li délka řady zpráv maximální počet, poslání zprávy selže.

Další zajímavostí systému posílání zpráv v reálném čase je priorita zprávy, která umožňuje poslat například „nouzovou“ zprávu, která předběhne v odbavení ostatní, běžné zprávy. Jako jiný příklad lze ovšem uvést i nastavení vyšší priority pro systémové zprávy (oproti aplikačním), které umožňuje hladší běh systému. Je-li však serverový proces úloha na nízké prioritě, mohou nastat problémy. Potom musíme využít jiných mechanismů.

Při používání řazení zpráv je třeba dbát na to, abychom po ukončení práce s řadou zavolali „úklidové“ funkce - řady mohou využít souborového systému a nebude-li použito příslušných funkcí, mohou data řady zůstat v souborovém systému, mnohdy i neviditelná pro příkazy souborového systému. Z tohoto důvodu je třeba ve fázi ukončování procesu volat funkce pro vyčištění fronty - proces může být opětovně spouštěn zvláště během vývoje procesu.

5.2.2 Sdílená paměť a mapování souborů

Sdílejí-li dva procesy paměťový blok, je tento blok mapován adresovými prostory obou procesů. Zapíše-li jeden proces údaj do sdílené paměti, druhý proces může okamžitě pracovat s novými údaji. Je ovšem třeba dávat pozor při práci se sdílenými daty - např.: dvě simultánní operace na sdíleném spojovém seznamu mohou mít za následek škody na datové struktuře. Mapování souborů je zabezpečeno pomocí *mmap* - tedy stejným mechanismem jako mapování sdílené paměti.

Sdílený objekt je potřeba na počátku používání vytvořit a inicializovat. Po ukončení práce je pak nutné takový objekt odstranit. Aby však bylo možné takový objekt využívat, je třeba na něj v rámci daného procesu vytvořit odkaz. Po skončení práce s ním je třeba tento odkaz opět odstranit.

5.2.3 Synchronizace procesů

Aby bylo možné úspěšně použít mechanismy sdílení paměti a mapování souborů, je třeba synchronizovat přístup ke sdíleným položkám. V současnosti existují tucty řešení, které tuto funkčnost umožňují.

Mezi základní způsoby jak řídit přístup pro sdílené zdroje jsou semafory. Procesy mohou na semaforu provádět operace *wait* (čekat) nebo *post* (zaslat). Pošle-li proces *wait* na semafor, jehož čítač je kladný, pokračuje a čítač sníží o jedničku (pravděpodobně na nulu). Je-li však čítač na nule, proces je nucen čekat, dokud jiný proces na semafor nepošle *post*, který čítač o jedničku opět navýší.

Pro synchronizaci lze využít i *mutexy*, zmíněné již dříve. Mutex je možné si přestavit jako zámek. Proces, který chce využívat zdroj, může převzít vlastnictví mutexu a zamknout ho. Aby zdroj zamčený pomocí mutexu bylo možno opět použít, je třeba jej otevřít.

5.3 Plánování, čas a uzamykání paměti

Plánování a časování úzce souvisí s problematikou programování v reálném čase. Plánování se zabývá několika vzájemně se překrývajícími se kategoriemi, jimiž jsou zpravidla úkoly:

- zajistit, aby se nějaká úloha vykonala v určeném čase
- zajistit, aby se úloha vykonala před jinou úlohou
- zajistit, že se úloha neúmyslně neopozdí
- zaručit plánovaní úloh

Rozhraní POSIX.4 pro reálný čas umožňuje definovat způsob plánování procesů třemi strategiemi: FIFO (tedy first-in-first-out), round robin a jinou strategii, zpravidla individuálně zpracovanou každou implementací RTOS.

5.3.1 Časovače

Otázka času je zpracována normou POSIX.1 a v této práci nebude uvedena. Tato kapitola bude věnována spíše časovačům, které jsou definovány normou POSIX.4. Díky mechanismům definovaným touto normou je možné dosáhnout vyššího časového rozlišení než na běžných systémech UNIX, a to až na nanosekundy - ovšem v případě podporuje-li tyto mechanismy i hardware. To je ovšem většinou nemožné - jedinou úlohou systému by

pak muselo být obnovování hodnoty času. Často je tedy nutné spokojit se s hodnotami s přesností zhruba na setinu sekundy (i když POSIX.4 stanovuje hranici dokonce na hodnotě 50Hz, tedy přesnost na padesátinu sekundy).

Časovače jsou obecně funkce, které nastaví čítač na nějakou hodnotu, která se následně automaticky snižuje. V okamžiku, kdy dosáhne nuly, je vyvoláno přerušení nebo signál, který je doručen procesoru, případně procesu. Časovač umožňuje procesoru zpracovávat během měření času jiné úlohy, čímž snižuje režii procesu a zvyšuje jeho výkon.

Časovače je možno vytvářet podle potřeby a nastavovat na různé hodnoty. Od standardních časovačů se liší vyšším časovým rozlišením a dynamičností.

5.3.2 Uzamykání paměti

Předešlé kapitoly pojednávají o tom, jak vykonat úkoly včas, avšak existují mnohé překážky, které tomu brání - a nejsou tím myšlena omezení hardwaru nebo systému. Překážkou ve snaze o běh procesu v reálném čase může být i jen samotná technologie virtuální paměti, resp. stránkování či rovnání segmentované paměti (*swapping*).

Stránkování má však velké nároky na čas a navíc není možné předem rozhodnout o tom, jaká data stránkovat a jaká nikoliv. Může se pak stát, že je třeba mít data dostupná v časech v řádu mikrosekund, avšak systém je zprostředkuje až po uplynutí několika milisekund, v horším případě i později.

Tomuto chování systému je možné se vyhnout použitím mechanismu uzamčení paměti nebo její části pro stránkování, který nám umožňuje norma POSIX.4.

5.4 Vstupy a výstupy v reálném čase

Vstupy a výstupy[27] myslíme veškerou komunikaci procesu s okolním světem - a procesy v reálném čase pracují zejména se vstupy a výstupy, reagují na podněty okolí, hledají a realizují vhodné řešení situace.

V reálném čase může být třeba:

- shromažďovat data z jednoho nebo více zařízení
- ovládat výstup na zařízení
- pořizovat záznam událostí (*logging*)

[27] V dalším textu mohou být vstupy a výstupy označeny zkratkou I/O z anglického *input/output*

- přehrávat nebo zaznamenávat multimédia
- provádět databázové operace
- řešit uživatelský vstup a výstup

Procesy v reálném čase mají zpravidla vyšší požadavky na komunikaci se vstupy a výstupy. Rozhodně je třeba zajistit, aby vstupy a výstupy neblokovaly chod procesu. V mnoha případech je také vyžadováno potvrzení o bezpečném dokončení vstupu, zejména v databázových systémech. A konečně, každá aplikace v reálném čase očekává od svých I/O subsystémů rychlý a předpověditelný výkon.

Běžné systémy UNIX mají model I/O, který je silný, jednoduchý a aplikovatelný na jakoukoli periferii. Jeho úskalí však spočívá v nedostatku precizní kontroly - volání v systémech UNIX blíže nedefinovaným způsobem zařídí to, o co proces zažádá, avšak nikdy nelze říci jakým přesně způsobem. Například volání `write` umožňuje minimální kontrolu nad faktickým stavem dat. V konkrétním čase nelze zjistit, zda jsou data přenášena k hardwaru, či uložena ve vyrovnávací paměti čekající na zkompletování. Není ani kontrola nad tím jak dlouho bude systém blokován zpracováním požadavku na I/O. UNIX celkově není synchronizovaný (synchronizace I/O a zařízení je považována za ukončenou pouze je-li zařízení patřičně aktualizován), je synchronní (proces čeká na dokončení I/O) a geometrii jeho souborů je nemožné, či obtížné řídit. Pro specifická zařízení pak používá volání `ioctl`, které však není standardizované, umožňuje pouze operace definované ovladačem a jako volání jádra má vysoké nároky na režii. Vzhledem k tomu, že RT aplikace potřebují se vstupy a výstupy pracovat neustále, přesahují jejich požadavky možnosti systémů UNIX. Mnohé systémy UNIX však obsahují ještě jiné mechanismy využitelné pro synchronní I/O nebo synchronizované I/O, které jsou sice nedokonalé avšak použitelné pro aplikace v reálném čase. Tyto mechanismy však nejsou opět standardizovány.

5.4.1 Synchronizace I/O

Aplikace v reálném čase potřebují pro účely vyrovnávání se s chybami, případně pro potřeby obnovení předchozího stavu deterministicky vědět, jaká data jsou ve vyrovnávací paměti a jaká bezpečně na disku. Standard POSIX.4 pro podporu těchto aplikací stanovuje dvě vrstvy synchronizace: datovou integritu (na úrovni obsahu souboru

bez *metadat*[28], která se zřídka mění) a plnou (souborovou) integritu. Ovšem, zůstává otázka, zda je opravdu vhodné a žádoucí tyto mechanismy využít - s jejich pomocí jsme schopni dosáhnout deterministického chování synchronizace, avšak za cenu výhod vyrovnávacího zásobníku (tzv. *bufferu*). Ztráta výhod použití vyrovnávací paměti se může stát velkou překážkou výkonnosti aplikace.

5.4.2 Asynchronní I/O

Systémy UNIX obecně používají synchronní komunikace - při čtení nebo zápisu musí proces vždy čekat na dokončení vstupů nebo výstupů. RTOS oproti tomu umožňují asynchronní I/O - požadavek na čtení nebo zápis se zařadí do fronty požadavků a proces může v mezičase, kdy se I/O realizuje vykonávat jinou část úlohy. Po ukončení I/O je pak proces upozorněn - například signálem.

5.4.3 Deterministický I/O

RT aplikace v mnoha případech potřebují zajistit co nejrychlejší průběh I/O a zároveň znát délky časových úseků, které I/O zabere. Jinými slovy potřebují, aby časy I/O byly předvídatelné.

Problémem je v tomto případě mnohdy i organizace dat na disku - načítaná data mohou být rozeseta v nesouvislých blocích na disku a systém pak stráví mnoho času vyhledáváním fragmentů dat. Souvisejícím problémem pak je i dynamická alokace bloků při rozšiřování datového úložiště na disku. Aplikace v reálném čase vyžadují, aby operační systém alokoval potřebné bloky předem a nezdržoval chod procesu dynamickou alokací uprostřed běhu úlohy v reálném čase.

Východiskem může být aplikování mechanismů využívaného operačního systému pro tvorbu souvislého datového úložiště (je-li takový mechanismus k dispozici), optimalizace parametrů souborového systému nebo použití hrubého zápisu na disk (tedy zápisu na binární úrovni). Hrubý zápis na disk je však obecně považován za nebezpečný postup a neměl by být raději využíván.

[28] metadata jsou informace „navíc“, informující například o času poslední modifikace dat, uživatelských právech apod.

5.5 Algoritmy a reálný čas

V mnohých případech není třeba přemýšlet nad koncepcí procesu, rozumným využitím zdrojů, trápit se limity použitých datových struktur a hledat optimální algoritmy procházení a vyhledávání. Mnoho programátorů v minulosti řešilo alespoň podobné problémy v reálném čase a tak je možné směle využít jejich prověřená řešení, které snadno přizpůsobíme našim potřebám. Příkladem může být například algoritmus implementující tzv. *garbage collector* (algoritmus spravující paměť - určuje, které bloky jsou již nevyužívané a připravuje je pro opětovné použití) v reálném čase.

6. Výsledky a diskuse

Přínosem této práce je soubor doporučení autora pro tvorbu aplikačních procesů pracujících v reálném čase. Doporučení jsou zaměřena na následující problémové oblasti: potřeba reálného času, volba architektury a způsobu komunikace a doporučené programovací techniky.

6.1 Potřeba reálného času

Otázka, zda je třeba aplikaci vůbec vytvářet tak, aby pracovala v reálném čase, je mnohdy stejně důležitá jako metodika tvorby takových procesů. Je třeba se zamyslet, zda bude mít koncepce procesu v reálném čase vůbec smysl při řešení úlohy. Postačující může být běžný přístup k programování aplikací.

Důležitou otázkou v diskuzi je také, zda má smysl pro úlohu, kterou řešíme využít možností operačního systému pro práci v reálném čase. Jak už bylo řečeno, tyto specializované operační systémy jsou zpravidla využity úlohami s vysokými nároky na garanci včasné reakce na podnět. Avšak mechanismy RTOS mohou být přitažlivé i pro ostatní úlohy, takže při jejich řešení padne volba na jeden z debatovaných systémů. Typické může být například použití systému pro zpracování úloh v reálném čase kvůli jeho malé velikosti, nenáročnosti, síťovému rozhraní či eleganci meziprocesové komunikace. Limitující většinou bývá cena takového systému.

6.2 Volba architektury a systému komunikace

Aplikaci je třeba navrhnout podle potřeby jako jeden proces, nebo jako více procesů, které řeší specializované úkoly v rámci aplikace. Jednoznačně nelze říci, při jakých příležitostech je která architektura aplikace výhodnější - vyplývá to z potřeb konkrétní

řešené úlohy. Při použití více procesů je pak důležité zvolit vhodný typ komunikace - nejvýhodnější se v tomto případě jeví mechanismy pro asynchronní komunikaci, které dovolují procesům vykonat části svých úkolů i během probíhající komunikace. V mnohých případech to však nemusí být žádoucí přístup. Komunikační mechanismus je každopádně třeba volit opatrně, abychom nenarušili chování procesu.

Aby proces zachoval svou funkčnost v reálném čase, je třeba dávat pozor, aby nebyl neúměrně závislý na synchronní komunikaci, jejíž použití může dočasně, v horším případě i trvale zablokovat chod tohoto procesu. Bude-li použita synchronní komunikace, je třeba ošetřit a minimalizovat stavy, ve kterých je proces blokován.

6.3 Doporučené programovací techniky

Doporučit lze používat mechanismů podporovaných operačním systémem, který jsme si zvolili. Tyto mechanismy nemusí být standardní - jejich použitím se sice zhorší přenositelnost aplikace, avšak ta v mnoha případech není požadovaným cílem aplikace.

Není-li žádoucí použít specifických mechanismů zvoleného systému, je vždy dobrou volbou sáhnout po standardním řešení - tedy použít mechanismů definovaných normou POSIX.4. Je ovšem třeba mít se na pozoru před případnými nevýhodami těchto mechanismů a případně je umět ošetřit.

Pří programování procesu v reálném čase, je třeba myslet v první řadě na determinismus úkolů tohoto procesu - úkoly by měly být vykonávané předpověditelným způsobem a v předpověditelném čase. Není žádoucí využívat pomalých algoritmů, avšak ani algoritmů s velikým rozdílem minimálního a maximálního trvání. Dobrá volba je použít již dříve vymyšlený, používaný a ověřený algoritmus v reálném čase.

Opatrnost musíme zachovávat i při práci s periferiemi - vstupy a výstupy jsou alfou a omegou programování v reálném čase a jejich špatné použití může mít za následek nedeterministickou povahu celého procesu.

6.3.1 Vstupy a výstupy

Jak již bylo řečeno v předchozích kapitolách, programování aplikací v reálném čase se zabývá hlavně prací se vstupy a výstupy, které však zároveň mohou být limitujícím faktorem pro hladký běh procesů v reálném čase.

Jednoznačným doporučením se proto v tomto případě jeví použití oddělených procesů zajišťujících komunikaci se vstupy a výstupy. Nelze-li podobné architektury

procesu využít, je vhodné s periferiemi zajistit asynchronní komunikaci a zamyslet se nad potřebou synchronizace.

Důležité při programování vstupů a výstupů v reálném čase je zajistit předpověditelnost chování komunikace se vstupem a výstupem. Kritické je pak ošetření stavů, kdy je periferie odpojená, nekomunikuje, přeruší spojení nebo zprostředkuje nesmyslná data, kde je třeba využít kontroly podmínek *vypršení* (*timeout*) komunikace.

6.3.2 Datové struktury

Při návrhu procesů v reálném čase je třeba dbát, aby použité datové struktury nadměrně časově nezatěžovaly samotný proces. Kritickými operacemi jsou v tomto případě *vyhledávání* a *řazení*, zvláště pak bude-li proces pracovat s velkým počtem dat.

Podle potřeby je nutné rozhodnout se pro výhodnou datovou strukturu - určujícími faktory v tomto případě budou: předpokládané průměrné *zatížení datové struktury* a *operace*, které na datové struktuře budou prováděny. Při volbě struktury budou pak více než zajímavé údaje o průměrné a maximální přístupové nebo vyhledávací době, očekávané vyhledávací době, výsledky pravděpodobnostní analýzy a další údaje. Vysvětlení těchto metod však překračuje rámec této práce.

6.3.3 Iterace a cykly

U procesů v reálném čase je třeba dobře promyslet i iterativní a cyklické algoritmy, které v nejhorších případech mohou skončit v nekonečné smyčce. Opět je třeba se zamyslet nad předpokládaným zatížením iterativních operací a v případě jejich zbytečného zatížení se rozhodnout místo nich implementovat složitější, avšak determinované algoritmy.

6.3.4 Testování

Testování je důležitou součástí všech typů programování a rozhodně by nemělo být zanedbáváno právě při tvorbě aplikací v reálném čase. Je nutné, aby procesy aplikace byly podrobeny vyčerpávajícímu testování - zejména pak podrobit procesy i testům krizových scénářů, nebo vysokému zatížení. Výsledky testování by měly být vhodným podkladem pro případné optimalizace, které mohou být vzhledem k vyžadované spolehlivosti a výkonu důležité.

7. Závěr

Aplikace pracující v reálném čase jsou součástí našeho života, ať už o nich víme, či nikoli. Nalezneme je ukryté v systémech automobilů a letadel, v obslužných systémech a automatických linkách nebo v řídících systémech továren a podniků.

Reálný svět obklopující nás je však velkým úskalím pro aplikace v reálném čase. Aby tyto aplikace mohly bezchybně fungovat, je třeba mnohdy i operačních systémů pro zpracování úloh v reálném čase. Ty díky své architektuře zabraňují nejrůznějším prodlevám, které by během zpracování důležitého úkolu mohly nastat a mít případně za následek nenapravitelné škody.

Díky mechanismům operačního systému pro zpracování úloh v reálném čase však nedokážeme odvrátit všechna rizika. Procesy pracující v reálném čase totiž potřebují specifický přístup. Je třeba, aby bylo jednoznačně jasné, jak dlouho bude program trvat a to za jakýchkoli podmínek. Tomu je pak třeba přizpůsobit zpracování úlohy.

Důležité pro programování aplikací v reálném čase jsou standardy POSIX, které většinu problémů komunikace v reálném čase řeší. Díky nim jsou aplikace implementované s jejich pomocí mimo jiné i přenositelnější na jiné operační systémy řídící se normami POSIX.

Práce ukázala specifika operačních systémů pro zpracování úloh v reálném čase a zmínila nejčastější nasazení těchto systémů v praxi. Následně byla jejich specifika demonstrována na systému QNX. Rozebrala se i problematika programování procesů pracujících v reálném čase a bylo provedeno srovnání s programováním běžných procesů. Byly určeny vhodné programovací techniky pro tvorbu těchto procesů a na jejich základě pak byl formulován soubor doporučení pro tvorbu aplikačních procesů pracujících v reálném čase, který je hlavním přínosem této práce.

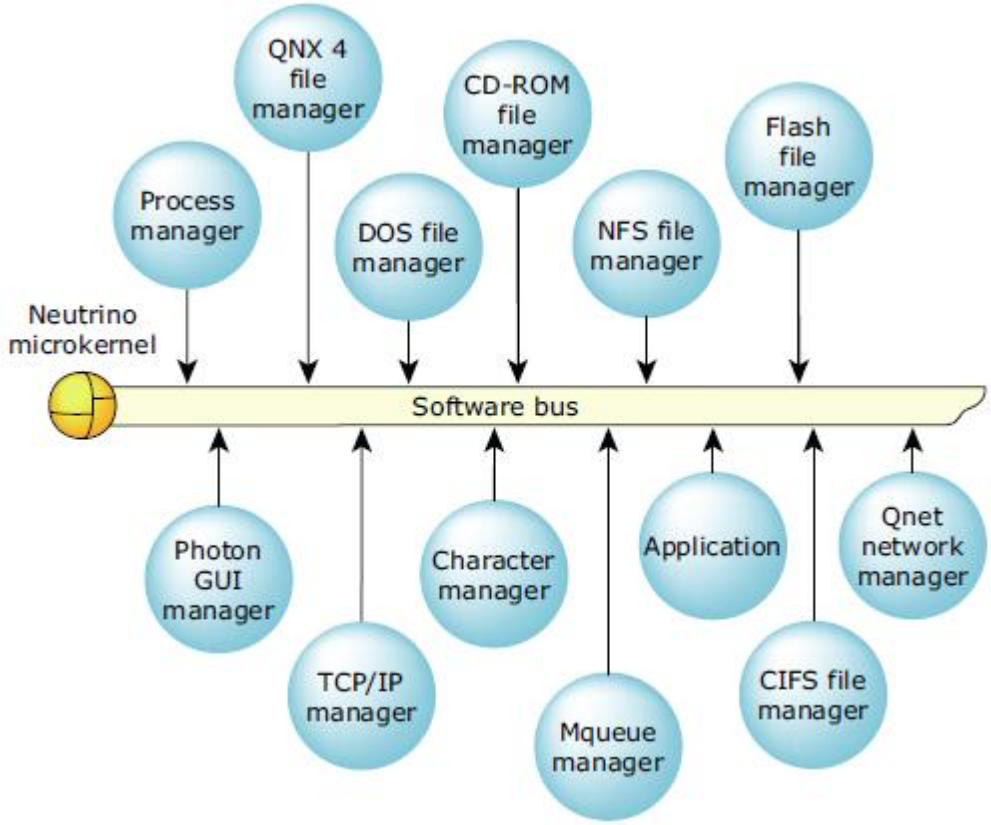
Vzhledem k tomu, že v budoucnu je možné očekávat snahu o automatizaci ve stále více rostoucím počtu odvětví. Proto můžeme očekávat i další rozvoj operačních systémů pro zpracování úloh v reálném čase i programovacích technik, které s vývojem aplikací v reálném čase souvisí.

8. Seznam použitých zdrojů

1. ABBOT, Doug: Linux for embedded and real-time applications.
   1. vyd, Amsterdam: Newnes, 2003. 255 s. ISBN 0-7506-7546-2
2. GALLMEISTER, Bill O.: POSIX.4 : Programming for the real world.
   1. vyd, Sebastopol : O`Reilly & Associates, 1995. 548 s. ISBN 1-56592-074-0
3. QNX Product documentation [on-line].
   Ottawa, Canada: QNX Software Systems Ltd., 2001-2008.
   (HTML, PDF) < http://www.qnx.com/developers/docs/ >
4. QNX Software Systems Ltd:
   QNX User`s guide, Utilities Reference and Utilities Reference Supplement.
   Kanata, Canada: QNX Software Systems Ltd., 1993. 4 sv.
5. HILDEBRAND, Dan: Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures: An architectural overview of QNX.
   Seattle, USA: 1992. ISBN 1-880446-42-1
6. KRTEN, Rob: The QNX Cookbook: recipes for programmers.
   1. vyd, Kanata, Canada: Parse software devices, 2003. 411 s. ISBN 0-9682501-2-2
7. Standard IEEE 1003.1 - 1980 a IEEE 1003.1b - 1993 [on-line]. 1980-2004.
   (HTML) < http://standards.ieee.org/develop/wg/POSIX.html >
8. ŠUMBERA, Petr:
   Porovnávací test operačních systémů reálného času [on-line]. 2000.
   (HTML) < http://www.odbornecasopisy.cz/index.php?id_document=27836 >
9. Dokumentace operačního systému ChibiRT [on-line].
   (HTML) < http://www.chibios.org/dokuwiki/doku.php?id=start >
10. GERLER, Oliver: QNX 4 OS, CE4218: Real-time systems [on-line].
    Limerick, Ireland: University of Limerick, 2000.
    (PDF) < http://www.rockus.at/gerler/writings/QNX.pdf >
11. QNX Neutrino RTOS: System Architecture [on-line].
    Kanata, Canada: QNX Software Systems Ltd., 2010
    (PDF) < http://www.qnx.com/download/download/20970/sys_arch.pdf >

9. Přílohy

Příloha A: Architektura QNX Neutrino



Zdroj: [11], str. 8