VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PROBLEMATIKA PŘECHODU OD JEDNOJÁDROVÉ K VÍCEJÁDROVÉ IMPLEMENTACI OPERAČNÍHO SYSTÉMU

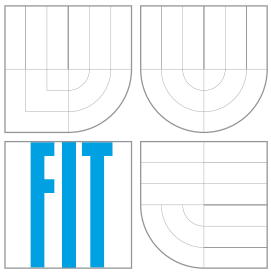
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN MATYÁŠ

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

PROBLEMATIKA PŘECHODU OD JEDNOJÁDROVÉ K VÍCEJÁDROVÉ IMPLEMENTACI OPERAČNÍHO SYSTÉMU

ISSUE OF MIGRATING FROM SINGLE-CORE TO MULTI-CORE IMPLEMENTATION OF OPERATING SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN MATYÁŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOSEF STRNADEL, Ph.D.

BRNO 2014

Zadání diplomové práce/16408/2013/xmatya00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2013/2014

Zadání diplomové práce

Řešitel: **Matyáš Jan, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Problematika přechodu od jednojádrové k vícejádrové implementaci operačního systému**
Issue of Migrating from Single-Core to Multi-Core Implementation of Operating System
Kategorie: Operační systémy

Pokyny:

1. Vytvořte přehled současných vícejádrových (MC) platforem a jejich vlastností. Shrňte principy související s realizací základních částí operačních systémů (OS) navržených pro jednojádrové (1C) prostředí.
2. Po dohodě s vedoucím zvolte konkrétní MC platformu a OS navržený pro 1C prostředí; jejich hlavní rysy zdokumentujte.
3. Shrňte obecné problémy související s přechodem od 1C k MC realizaci OS a diskutujte jejich možná řešení.
4. Diskutujte změny v OS z bodu 2 umožňující využít k činnosti OS a aplikací jádra dostupná v MC z bodu 2.
5. Realizujte s vedoucím dohodnuté změny z bodu 4 s cílem jejich minimálního dopadu na API, konstrukci OS a vysokou škálovatelnost.
6. Vhodně ověřte a vyhodnoťte schopnost změněného OS či jeho částí, určených vedoucím, běžet v MC prostředí.
7. Srovnejte výpočetní režie variant původního 1C-OS změněného MC-OS.

Literatura:

- Peter, S.: Resource Management in a Multicore Operating System. Disertační práce č. 20664, ETH, Zurich, 2012. 156 s. Dostupné mj. z http://e-collection.library.ethz.ch/view/eth:6270
- Další literatura dle pokynů vedoucího.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Strnadel Josef, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2013
Datum odevzdání: 28. května 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Tato práce se zabývá úpravou µC/OS-II pro běh na vícejádrovém procesoru, konkrétně na Zynq 7000 All Programmable SoC, který obsahuje dvě jádra architektury ARM Cortex-A9. Jsou v ní diskutováný potřebné změny a s tím spojené možné problémy při této transformaci.

Abstract

This thesis discuss necessary changes needed in order to run µC/OS-II on multicore processor, mainly Zynq 7000 All Programmable SoC which uses two ARM Cortex-A9 cores. Problems that arise during this transition are also discussed.

Klíčová slova

µC/OS-II, ZedBoard, ARM, Cortex-A9, Zynq, vícejádrový procesor, paralelismus, boot, bootloader, AMP, SMP, real-time, operační systém, plánovač, přerušení

Keywords

µC/OS-II, ZedBoard, ARM, Cortex-A9, Zynq, multicore processor, parallelism, boot, bootloader, AMP, SMP, real-time, operating system, scheduler, interrupt

Citace

Jan Matyáš: Problematika přechodu od jednojádrové k vícejádrové implementaci operačního systému, diplomová práce, Brno, FIT VUT v Brně, 2014

Problematika přechodu od jednojádrové k vícejádrové implementaci operačního systému

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, Ph.D.

.....
Jan Matyáš
28. května 2014

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce, panu Ing. Josefu Strnadelu, Ph.D., za odborné vedení, jeho ochotu a zapůjčení vývojového kitu ZedBoard. Dále potom všem, kteří se podíleli na mé cestě vedoucí k této práci.

Obsah

1 Úvod **3**

2 Související témata **4**
2.1 Historický vývoj . . . 4
2.2 Operační systém . . . 5
2.2.1 Jádro OS . . . 6
2.2.2 Operační systémy na PC . . . 10
2.2.3 Operační systémy na embedded zařízeních . . . 11
2.3 Hardware . . . 11
2.3.1 ARM . . . 12
2.4 Boot . . . 13
2.4.1 Platforma kompatibilní s IBM PC . . . 13
2.4.2 Boot s jádry ARM . . . 15
2.4.3 Firmware . . . 16

3 Od jedno k více jádrům **17**
3.1 Různé formy paralelismu . . . 18
3.1.1 Paralelismus na úrovni instrukcí . . . 19
3.1.2 Víceprocesorová a vícejádrová architektura . . . 21
3.2 Změny v operačním systému . . . 25
3.2.1 Plánování . . . 25
3.2.2 Paměť . . . 26
3.2.3 Jádro . . . 27
3.3 Aplikace . . . 27

4 Zvolená platforma a RTOS **28**
4.1 Popis Zedboardu . . . 28
4.2 Zynq-7000 All Programmable SoC . . . 29
4.2.1 Vývojové prostředí . . . 29
4.2.2 Programové možnosti . . . 30
4.2.3 Bootovací konfigurace . . . 32
4.2.4 Další vlastnosti . . . 33
4.3 µC/OS-II . . . 33
4.3.1 Inicializace systému . . . 34
4.3.2 Procesy . . . 34

5 Provedené změny **39**
5.1 Systémová konfigurace 39
5.2 Probuzení druhého jádra 39
5.3 AMP konfigurace 41
5.4 SMP konfigurace 42

6 Závěr **46**

Kapitola 1

Úvod

Lidstvo je od nepaměti hnáno touhou poznání, zdokonalování, ulehčování si své práce, ziskem zdrojů a překonávání výzev. Všem těmto činnostem napomáhá, bez pochyby revoluční vynález, počítač. Protože se ukázal být skvělým pomocníkem, jeho použití se rozšiřovalo. Tím vnikala další potřeba jeho zdokonalování, což umožnilo nové využití. Vznikl tak začarovaný kruh, z kterého aktuálně není cesta ven. Protože jsme ale limitování fyzikálními zákony, nelze stále dokola zlepšovat jednu a tutéž vlastnost. Proto se oblast úprav dotýká každé části počítače.

Jednou z nich je paralelismus. A to buď v rámci jednoho jádra, nebo také zvyšováním jejich počtu. Tento postup se ostatně uplatňuje i v ostatních odvětvích. Potřebujeme-li uvařit jídlo pro sto lidí, můžeme přidat pomocníky do kuchyně a zvětšit hrnce, nebo využít více kuchyní zároveň. To ale nebude fungovat vždy – může dojít k situaci, kdy v kuchyni již nebude k hnutí, nebo nebude logisticky možné zásobovat všechny zúčastněné kuchyně. Totéž nastává ve světě počítačů – v určitou dobu se jednotlivé aplikace budou natolik zdržovat, že jejich paralelní vykonávání již nebude efektivní. Případně nebudeme schopni dodávat data z paměti jednotlivým jádrům, protože nám nebude postačovat kapacita jejich propojení.

I tak je ale třeba využít paralelismus v co nejvyšší a zároveň efektivní míře, protože se jedná o velmi účinný nástroj zrychlení výpočtu. Řízení takového systému je ale netriviální problém a je třeba mu přizpůsobit kromě hardwaru, také software. Důležitou částí zde proto hraje operační systém, který musí umět pracovat se všemi dostupnými prostředky. Změny, které je třeba provést jsou diskutovány v této práci spolu s experimentální úpravou jednoho z nich.

Kapitola 2

Související témata

Následující podkapitoly slouží k seznámení s pojmem operační systém, jeho vlastnostmi a využitím a jako obecný úvod. Čtenář znalý tématu je může přeskočit.

2.1 Historický vývoj

Jako počátky informatiky se v některých případech uvádí i doby před počátkem našeho letopočtu, kdy vznikaly první počítací stroje. Tyto stroje byly posléze zapomínány, znovu objevovány a upravovány, ale pro mou práci nepřináší nic zajímavého. Až ve 20. století začaly vznikat programovatelné počítače. Postupem času byly vylepšovány a zrychlovány, až nastala situace, kdy lidská obsluha způsobovala výrazné omezení maximální rychlosti těchto počítačů. Začaly vznikat různé automatizační mechanizmy obsluhy, např. různé podavače, které spouštěly jednotlivé programy automaticky za sebou tak, že mezi nimi nevznikaly prodlevy.

Protože tehdy, stejně jako dnes, existovaly různé architektury, výrobci se snažili ulehčit práci programátorům i sobě za pomocí funkcí, které zajišťovaly často používané operace. Programátoři nemuseli psát základní rutiny pro obsluhu počítače a výrobci měli méně práce s řešením problémů při nesprávném použití jejich výrobku. Takto vznikaly první knihovny a ovladače, na kterých jsou dnes postaveny operační systémy. Se zvyšující se rychlostí výpočtů také brzo přestala stačit rychlost periférií a vznikaly prodlevy způsobené čekáním na dokončení vstupních nebo výstupních operací. Tento čas se mohl využít pro obsloužení programu, který v danou chvíli vyžadoval pouze procesorový čas pro samotný výpočet. Vznikla tedy myšlenka přepínání procesů v době, kdy jeden čeká na dokončení pomalé v/v operace. Toto byly první primitivní plánovače – čekal-li proces na dokončení operace, byl blokován a v mezičase byl vykonán jiný program ve frontě. Tímto přístupem vznikly do té doby nevídané problémy. Najednou nebyl počítač vyhrazen pouze jednomu programu. Běžící programy se mohly navzájem ovlivňovat a číst nebo měnit cizí data. Kromě problémů způsobených nekorektním chování jednotlivých programů, ať už např. zastavením celého výpočtu z důvodu chyby, nebo přístupu do cizí paměti, vznikal také problém reálné versus použité paměti při běhu programu nebo linkování. Tyto problémy bylo třeba vyřešit, a tím ulehčit práci programátorům a vývojářům. Začaly se hledat cesty, jak od sebe jednotlivé programy oddělit, aby jeden o druhém nevěděly a nemohly se nijak ovlivnit.

Začaly vznikat různé systémy řízení přístupu do paměti a byl zaveden pojem virtuální paměť. S rostoucím výkonem počítačů rostly i nároky na ně. Jeden počítač zvládal obsloužit více uživatelů najednou, rostla jejich paměť a uživatelé měli svá data uložená v počítači

a vyžadovali určitou míru soukromí. Bylo potřeba, aby „někdo“ řídil přístup k těmto datům. Tím „někdo“ se stal operační systém.

Postupem času se požadavky na operační systém měnily a vyvíjely. Například s příchodem počítačů typu workstation nebylo potřeba obsluhovat více uživatelů najednou, za to byla důležitá co nejmenší režie vlastního systému. To vše vedlo k velké rozmanitosti operačních systémů, která dnes již ve světě pc není tak markantní. Tehdy byla způsobená mimo jiné velkým množstvím používaných platforem, ke kterým výrobci dodávali svůj vlastní operační systém. Dnes jsou již operační systémy natolik složité, že jejich vývoj by vyžadoval obrovské množství prostředků. I ve světě embedded zařízení se postupně přechází na architekturu obsahující univerzální jádro (jako je ARM) s případným rozšířením nutným pro požadovanou funkcionalitu. Vznikají tak různé SoC[1], na které lze s minimem námahy naportovat jádro Linuxu nebo jiného vhodného operačního systému s požadovanými vlastnostmi. Tím se sníží potřeba rozsáhlých úprav jádra systému a jsou sníženy náklady a čas potřebný k vývoji zařízení.

V dnešní době spotřební ekonomiky, kdy cena hardwaru klesá, jeho výkon stoupá a jako největší položka vývoje se jeví čas programátora a doba vývoje zařízení. Protože jsou ale požadavky na zařízení stále vyšší a vývojové cykly kratší, je třeba vzhledem k těmto rozcházejícím se požadavkům zajistit maximální míru unifikace a možnost znovupoužití již hotového. To vše klade vysoké nároky na vývoj operačních systémů, které se stávají nedílnou součástí všech zařízení, které používáme.

2.2 Operační systém

První operační systémy sloužily jako virtuální vrstva nad hardwarem a poskytovaly základní systémové služby. V dnešní době se pod pojmem operační systém často očekává kompletní systém s grafickou nadstavbou a doplňkovými programy, která umožní uživateli provádět nejrůznější operace ihned po instalaci. Proto velká většina dnešních operačních systémů obsahuje nejrůznější programy pro práci se soubory, grafikou, videem, internetem a další. Navzdory tomu, že tyto programy mívají pouze základní funkcionalitu, uživatel jejich přítomnost očekává.

Operační systémy se podle typu odezvy dělí na:

- Běžné
- Real-time – ty se dále dělí na:
  - Soft real-time,
  - Firm real-time a
  - Hard real-time

Oproti běžným operačním systémům musí mít real-time operační systémy garantovanou maximální dobu odezvy pro svou činnost. Tyto systéme se používají tam, kde je nutná garantovaná reakční doba např. z důvodu bezpečnosti. Tyto systémy se používají nejčastěji u různých systémů řízení – např. v automobilech (ABS, Airbagy, ...), elektrárnách, letadlech, armádních strojích aj.

Dělení na soft, firm a hard real-time se provádí podle závažnosti následků, který by byly způsobeny překročením nejpozdější možné odezvy. V případě vážných následků (ohrožení

[1] System on a chip

zdraví člověka) se jedná o hard real-time. Pokud by došlo k materiálním škodám, jedná se o firm systém a pokud dojde pouze k degradaci výkonu systému, jedná se o soft real-time operační systém. Častý omyl u pojmu real-time operační systém je očekávaná okamžitá odezva. Toto ale není pravda. Real-time operační systém má garantovanou maximální reakční dobu, ale tato doba se specifikuje – může trvat i několik minut nebo hodin.

2.2.1 Jádro OS

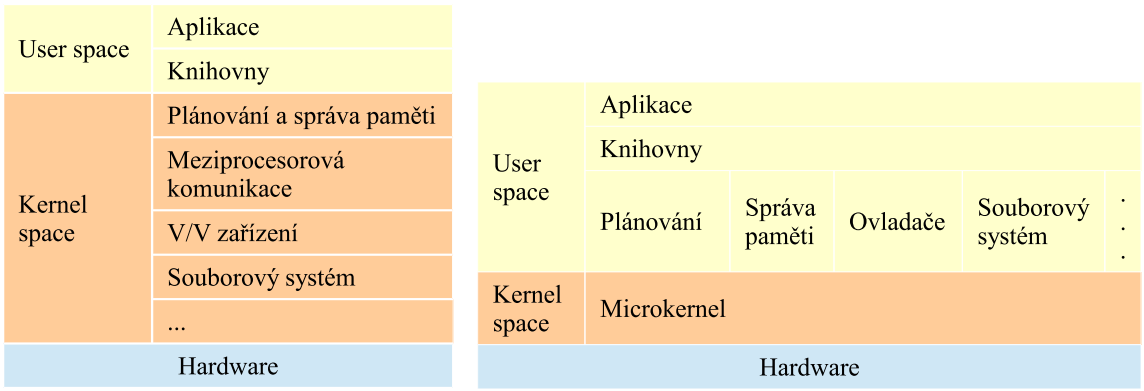
Klíčová část operačního systému se nazývá jádro, nebo anglickým kernel. Jádro je ta část operačního systému, která vytváří, spolu s ovladači, virtuální vrstvu nad hardwarem a poskytuje služby využívané programy. Nedílnou součástí těchto služeb je vytváření nových procesů, jejich plánování, přidělování a správa paměti, obsluha přerušení, synchronizace a meziprocesorová komunikace. Z hlediska mé práce a obecně embedded světa je zajímavé pouze jádro, protože to řeší veškeré problémy spojené s přechodem na víceprocesorovou architekturu a poskytuje v podstatě vše, co se od operačního systému v embedded světě očekává. Ostatní programy již používají služby jádra, na jehož implementace je závislá využitelnost všech vymožeností dnešního hardwaru, jako například vícejádrového CPU. Na druhou stranu, aby mohl program využít daný hardware naplno, musí být s tímto požadavkem také vyvíjen a využívat dané jádro a hardware naplno, což také vyžaduje speciální přístup k návrhu daného programu.

Obecně operační systém pracuje ve dvou režimech - privilegovaný (tzv. kernel space) a neprivilegovaný (tzv. user space). Zjednodušeně řečeno, v privilegovaném režimu běží jádro a v neprivilegovaném běží další procesy. Tot rozdělení se mírně liší na základě architektury jádra.

Architektura jádra

Z pohledu základní struktury se architektura jádra dělí na [21]:

- Monolitické jádro (např. Linux),
- Mikro jádro (např. QNX) a
- Hybridní jádro (např. Windows NT)



(a) Monolitické jádro (b) Mikro jádro

Obrázek 2.1: Typy jader

Monolitické jádro běží celé v jednom adresovém prostoru (kernel space), tj. veškeré datové struktury jsou sdílené napříč celým jádrem. Toto uspořádání je výhodné z hlediska výkonosti, ale způsobuje problémy při údržbě, hledání chyb a je třeba jej vždy kompilovat jako celek. Naproti tomu mikro jádro se skládá z nezávislých modulů, které mezi sebou navzájem komunikují. Každý modul má svou paměť a běží v user space. Toto je výhodné zejména pro jednoduchou údržbu, jednodušší upgrade za běhu systému a vyšší odolnost vůči chybám – jednotlivé moduly lze za určitých okolností restartovat bez vlivu na zbylé moduly. Návrh mikro jádra je ale složitější – je třeba vhodně zvolit rozhraní mezi jednotlivými moduly, protože pozdější změna může zahrnovat rozsáhlé změny napříč moduly. Také je zde kritické rozdělení mezi kernel a user space a optimalizace pro daný hardware. První verze mikro jádra nebyly cíleně optimalizovány a měly velký problém s výkonností způsobené mimo jiné častým přechodem mezi user a kernel space a častým přepínáním kontextů. Druhá generace poté musela dokázat, že mikro jádra dokáží konkurovat monolitickým jádrům, což se jí povedlo [21]. Hybridní jádro je kombinace dvou předchozích a má za cíl jednodušší návrh s vyšším výkonem a jednoduchou údržbu.

Různé operační systémy měly různé cíle a proto se jejich vývoj i vlastnosti liší. Zajímavou diskuzi na toto téma je možné nalézt například v tzv. Tanenbaum – Torvalds debatě [13] z roku 1992.

Plánovače

Plánování, tj. přidělování hardwarových prostředků, je jedna z nejdůležitějších operací prováděných jádrem. Pro výběr nového procesu ke spuštění existuje několik mechanizmů, které se liší svými vlastnostmi. Mezi hlavní vlastnosti daných mechanizmů patří:

- Propustnost – kolik procesů se stihne vystřídat v daném čase
- Latence
  - Doba k prvnímu spuštění nového procesu a
  - Doba potřebná k jeho dokončení
- Spravedlivost
- Čekací doba – doba, než se proces dostane ze stavu připraven do stavu běžící

Tyto vlastnosti jsou navzájem protichůdné a proto každý plánovací mechanizmu implementuje určitý kompromis. Jedná-li se o real-time aplikaci je navíc potřeba dodržení nejpozdějších termínů dokončení pro každý proces a přerušení, což jsou pro ně kritické vlastnosti. Dále se často u real-time aplikací vyskytují periodické úlohy a proto pro ně existují speciální plánovací mechanizmy se zaměření na tento typ úloh.

Existuje několik různých technik plánování, přičemž nejzákladnější dělení je na:

- Preemptivní – Proces může být přerušen kdykoliv během svého běhu.
- Kooperativní – Proces nemůže být přerušen kdykoliv, ale pouze v případě, kdy tak dá sám najevo (např. systémovým voláním).
- Nepreemptivní – Běžící proces nelze přerušit. Další proces může být spuštěn až když současný skončí.

U běžných pc operačních systému se dnes používají preemptivní plánovače, které za pomoci časovače dostávají předem definované časové intervaly a navzájem se střídají, což umožňuje velmi dobrou latenci, kterou uživatel očekává. Samozřejmě preemptivní plánování také umožňuje běžícímu procesu vzdát se procesoru. Ať už kvůli čekání na v/v operaci, nebo proto, že již v danou chvíli nepotřebuje další výpočetní výkon.

Kooperativní plánování může být výhodné zejména tam, kde se pracuje se sdílenými zdroji a z důvodu synchronizace a zamykání zdrojů by docházelo ke zbytečnému přepínání kontextu – další proces by stejně nezískal zámek a byl by znovu přerušen, což by znamenalo zbytečné zdržení při přepínání kontextů. Je-li ale v systému velké množství procesů s dlouhými nepřerušitelnými úseky, může být latence systémů neúnosná. Navíc je tento typ plánování háklivý na chyby v aplikaci. Pokud se aplikace zacyklí, nikdy se nevzdá procesoru a systém je třeba restartovat.

Nepreemptivní plánovač není běžně využíván, protože se hodí pouze pro speciální případy, ale v těch může dosahovat lepších výsledků, než preemptivní plánovače [12].

Mezi základní plánovací mechanismy patří:

- First in, first out (FIFO) – jedná se o jednoduchý plánovací mechanismus, kdy se procesy spouští v pořadí, v jakém přišly do fronty připravených požadavků
- Pevně stanovené priority – procesy jsou spouštěny na základě priority, kdy je každá priorita přiřazena maximálně jednomu procesu
- Víceúrovňové – existuje více front, do kterých se řadí procesy podle jejich priorit a z nich jsou dále spouštěny např. pomocí FIFO metody
- Nejkratší nejdříve – je vybrán proces, který bude hotov nejdříve; toto vyžaduje znalost nebo odhad potřebné doby k dokončení
- Round-robin – každému procesu je přidělen předem stanovený čas a všechny procesy se cyklicky střídají
- Manuální plánování – programátor předem definuje, jak budou procesy plánovány; zde je potřeba dopředu znát počet a typy úloh a proto se používá pouze v embedded světě ve speciálních případech
- „Úplně spravedlivý plánovač[2]“ (CFS [18]) – snaží se procesorový čas spravedlivě rozdělit mezi všechny procesy

Jednotlivé mechanismy lze samozřejmě různě kombinovat.

Při použití plánování s prioritami je třeba dát pozor vyhladovění, zejména pokud se jedná o pevné priority. V případě vysoké náročnosti procesů s vysokou prioritou nemusí být ty s nízkou prioritou nikdy obslouženy. Toto je důležité zejména u real-time aplikací s periodickými úlohami. Pokud ale známe dobu vykonání, případně nejhorší dobu vykonání a periodu jednotlivých úloh, můžeme přesně spočítat využití procesoru a tak ověřit správnou funkčnost systému.

Datové struktury v jádře

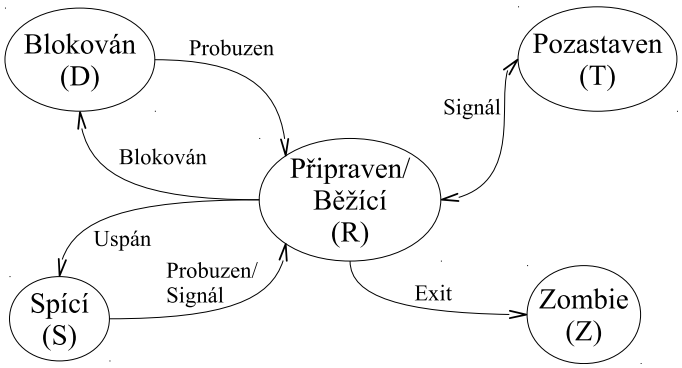
Protože jádro přiděluje prostředky, potřebuje si někde uchovávat jejich přehled a stav. K tomu používá různorodé datové struktury. Mezi ně se řadí např. struktury pro otevřené

[2] Completely Fair Scheduler

soubory, síťové, paměťové a další zařízení, informace o stránkách a virtuální paměti, semaforech, soketech, čítačích a časovačích, procesech, struktury plánovače úloh a mnohé další.

Každá z uvedených (i neuvedených) struktur je postavená na míru potřebám. Např. plánovač v Linux (aktuálně CFS) potřebuje řadit procesy podle času a proto používá red-black tree, což je v podstatě obarvený binární strom.

Mezi důležité datové struktury se řadí také Task Control Block (TCB). Do této struktury jsou ukládány informace o procesu, informace o jeho stavu v případě přepnutí kontextu a informace pro plánování.



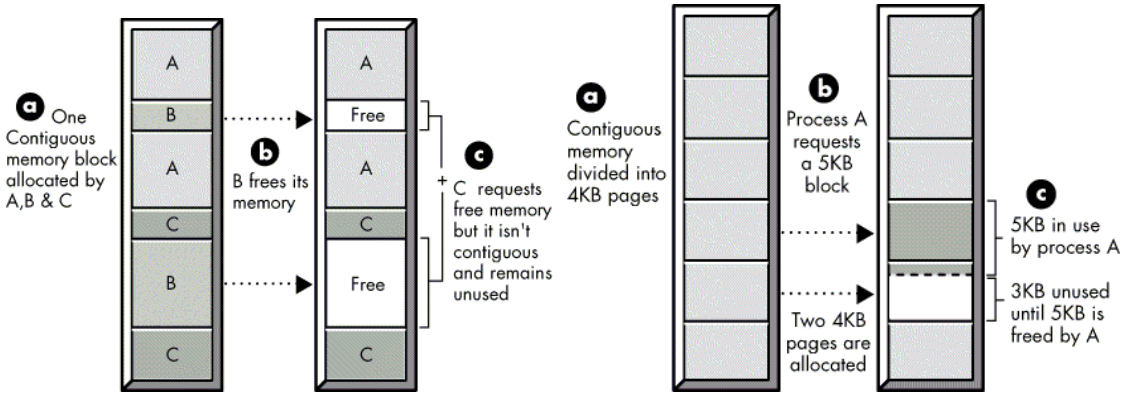
Obrázek 2.2: Stavy proces v Linuxu

Správa paměti

Jádro přiděluje procesům paměť na základě jejich požadavků. Většinou při tom používá virtuální paměť za pomoci stránkování a/nebo segmentace. Takto lze zajistit, aby o sobě procesy navzájem nevěděly a nemohly se ovlivňovat. Také lze za pomoci swapování reálně přidělit procesům více operační paměti, než je fyzicky přítomno. Swapování může být pro uživatele neznatelné, ale v případě paměťově náročných procesů může způsobit citelnou ztrátu výkonu.

Pro přidělování paměti existují v zásadě dva přístupy. Buďto se procesu přidělí jím požadovaná část paměti kontinuálně za sebou, nebo se mu přidělí blok paměti předem definované velikosti. První případ způsobuje externí fragmentaci, kdy mezi alokovanými bloky vznikají různě velké uvolněné bloky, které nelze ideálně využít. To proto, že mají „špatnou“ velikost pro další přidělení, nebo se dále dělí na menší až vznikne příliš malé a nepoužitelné místo. Druhý případ způsobuje interní fragmentaci, kdy pokud proces požaduje 20B paměti a přidělují se pouze bloky o velikosti 8B, 16B a 32B, je mu přidělen blok 32B a 12B zůstává nevyužito.

Po delším používání proto klesá reálně využitelná paměť. Existuje několik způsobů, jak tyto nepříjemné vlastnosti odstranit. Jedním z nich je přidělování paměti procesům co nejméně kontinuálně, čímž dojde k narušení jednotné struktury a tím i k omezení problému. Dalším je třeba defragmentace, která je ale časově a výkonově náročná. Použití stránkování může také vzhledem k jeho funkčnosti omezit externí fragmentaci v hlavní paměti a interní fragmentaci v pamětech blíže k procesoru.



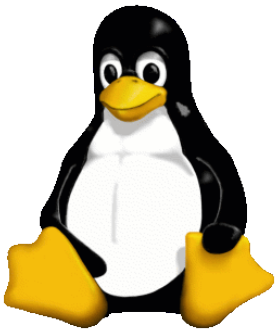
(a) Externí (b) Interní

Obrázek 2.3: Paměťová fragmentace [16]

2.2.2 Operační systémy na PC

Mezi nejúspěšnější operační systémy na pc dnes pravděpodobně patří Microsoft Windows, Unix a z něj odvozené Linux, FreeBSD a Mac OS X. V dnešní době se k nim snaží připojit Google s jeho cloudovým Chrome OS, ale zatím se mu příliš nedaří. Všechny tyto operační systémy cílí na širokou škálu uživatelů a jejich potřeby. Proto jsou navrženy pro univerzální použití na nejrozšířenějších hardwarových platformách.

Vznik Linuxu



Obrázek 2.4: Linux maskot

Protože Linux dnes patří mezi oblíbené operační systémy, je aktivně vyvíjen a jeho zdrojový kód je všem k dispozici, zmíním zde jeho stručnou historii. Protože Linux byl vyvíjen přesně podle požadavků jeho uživatelů a vývojářů v průběhu času tak, jak jim přikládali priority, reflektuje vyvíjející a měnící se požadavky na operační systém.

Linux začal vznikat na začátku 90. let. V té době již existovaly jiné operační systémy (např. první verze Unixu vznikla již na začátku 70. let a již od roku 1988 existovala norma POSIX[3]), mohl Linux využít již z praxe známé a ověřené postupy. První verze Linuxového jádra (určená pro procesor i386) byla vytvořena, v té době studentem Helsinské univerzity, Linusem Torvaldsem. Setkala se s úspěchem a do tohoto projektu začaly přispívat další lidé, což postupem času vedlo k Linuxu, jak jej známe dnes.

- Verze 1
  Když se po třech letech vývoje dostal do verze 1.0, podporoval jedno jádrový procesor na bázi i386, využíval virtuální paměť, šlo na něm zprovoznit grafickou nadstavbu X Window System a byl to kompletně samostatný systém. Později přibyla podpora procesorů Alpha, SPARC a MIPS.

- Verze 2
  Verze 2 přinesla podporu pro víceprocesorové systémy formou velkého zámku. Ten

[3]Portable Operating System Interface [24]

byl odstraně ve verzi 2.2, spolu s podporou dalších procesorů a novými filesystémy. Verze 2.4 byla rozšířená o podporu např. USB, ISA Plug and Play, přibyla podpora dalších procesorů a později i podpora LVM, RAID, ext3 a další. Od této verze již začaly být běžné kompletní change logy ke každé verzi a lze je nalézt spolu s kódem na stránkách https://www.kernel.org/pub/linux/kernel/. Verze 2.6 přinesla další rozšíření a vylepšení podporovaných architektur, podporu další souborových systémů, PAE, integraci ALSA atd.

- Verze 3
  Verze 3.0 nepřinesla nic převratného. Změna na verzi tři proběhla spíše z praktických důvodů a poblíž dvacátého výročí vzniku Linuxu [25]. V dalších verzích dochází k postupnému vylepšování a přidávání ovladačů a postupnému sbližování s Androidem, který vznikl odštěpením od verze 2.6. Ve verzi 3.8 došlo k odstranění podpory procesoru, pro který Linux vznikl – Intel 386. Verze 3.14 přináší podporu pro plánování s deadline, což zlepšuje základní real time plánování oproti dosavadním možnostem.

2.2.3 Operační systémy na embedded zařízeních

Protože embedded zařízení jsou často zaměřeny na velmi úzké spektrum operací, je zde rozmanitost výrazně vyšší. Existuje zde širší platforma rozdílných operačních systémů, které si navzájem konkurují, a procesory s různými instrukčními sadami, optimalizovanými pro úzce zaměřené aplikace. Toto vede k výraznému zvýšení účinnosti takového zařízení a snížení požadavku na hardware, primárně na paměť a rychlost procesoru. Mezi jednu výraznou skupinu patří systémy reálného času. Na tyto operační systémy jsou kladeny speciální požadavky zajišťující jejich časovou předvídatelnost za každých okolností. Pro tyto operační systémy existují také certifikace, které bývají vyžadovány při jejich aplikace např. v armádě, energetice, dopravních prostředcích nebo jiných kritických řídících systémech. Mezi real time operační systémy patří například LynxOS, VxWorks, QNX, RTLinux, FreeRTOS a nebo µC/OS-II.

2.3 Hardware

Znalost hardwaru je pro vývoj jádra operačního systému klíčová, protože jádro plně využívá hardware, zatímco programy využívají služby jádra. S periferiemi bývají dodávány ovladače, ale procesor a paměťový systém je zcela v režii jádra. V průběhu času došlo k přechodu od 4 bitových čipů až po dnešní 64 bitové. Spolu s touto změnou přichází stále větší potřeba paměti a protože její rychlost a cena jdou důležité parametry, došlo k vzniku hierarchie paměti do různých stupňů s rozdílnou cenou, velikostí a rychlostí. Spolu s touto změnou vznikaly různé techniky jak docílit co nejvyšší rychlosti a schopnosti obsloužit všechny programy dle jejich potřeby. Začaly se používat techniky segmentování, stránkování, swapování a vznikl pojem virtuální paměť. Procesory začaly používat superskalární technologie, out of order výpočty atd., které, ač na první pohled neznatelné uživateli, mohou přinášet komplikace při synchronizaci více procesů nebo vláken. Ačkoliv se tyto změny dotýkaly převážně operačních systémů pro servery a pc, dnešní doba levných a výkonných embedded zařízeními je postupně tlačí i do tohoto segmentu. Zde jsou eskalovány do nových rozměrů dalšími požadavky, jako je například real time implementace.

V průběhu času docházelo a stále dochází ke zlepšování používaného hardwaru, ať už se jedná o procesor, paměť, nebo používané v/v periferie. Z historického hlediska byly

procesory rozšiřovány o nové bloky, sloužící pro specifické operace, jako např. floating point operace. Operační systém se musel vyvíjet tak, aby využíval tyto nové bloky a tím dosáhl co možná nejvyššího výkonu. To samozřejmě způsobuje jeho bobtnání a spolu s novými a rozmanitějšími periferiemi roste i počet nutných ovladačů dodávaných spolu s jádrem. Navíc existuje požadavek na zpětnou kompatibilitu...

V embedded světě tento problém spíše neexistuje, protože pro každou platformu je třeba provést speciální port daného systému. Existuje totiž velkém množství nejrůznějších µkontroléru, SoC, čipů FPGA[4], atd. Obzvláště v dnešní době, kdy se od embedded zařízení očekává funkčnost pc jako je připojení na internet, video a audio, voice funkce a další jsou hojně využívané nejrůznější SoC a FPGA čipy. Vzhledem k hardwarové složitosti takovéhoto systému je pak velmi vhodné využít již ověřené kusy kódu některého z existujících operačních systémů. Tento systém je ale třeba napasovat na použitou platformu a její konkrétní implementaci. Naštěstí většina operačních systémů na toto pamatuje a jeho velká většina je napsána ve vyšším programovacím jazyce a je třeba naportovat pouze některé části OS, které jsou závislé na hardwaru, jako je inicializace, práce s paměťovým systémem, rutiny pro vstup, opuštění, zakázání a povolení přerušení atp.

2.3.1 ARM

Tato práce bude stavět převážně na architektuře ARM[5] (konkrétně ZedBoard, který je popsán v kapitole 4.1) a proto zde uvedu její detailnější popis.

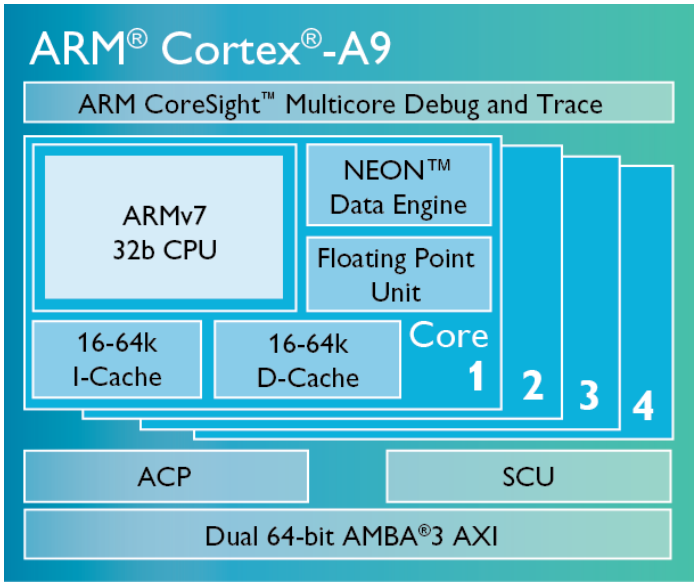
ARM je skupina instrukčních sad založená na principu RISC, zajišťující její jednoduchost (ve srovnání s architekturami CISC, jako např. x86) a umožňuje jí dosáhnout vyšší účinnosti a nižší ceny. Byla vyvinuta britskou firmou Acorn Computer již v 80. letech minulého století. Později se firma rozdělila na menší, z nichž jednou byla ARM Holdings, která se do dneška stará o vývoj a prodej ARM architektury [2]. V dnešní době existují tři „profily" s rozdílnými instrukčními sadami. Každá je určena pro různé aplikace. Tyto řady jsou [1]:

- profil A pro aplikační použití (struktura je naznačena na obrázku 2.5)
- profil M pro použití jako µkontrolér
- profil R pro real time aplikace

Tyto řady se dále dělí podle použitých rozšíření a konkrétního designu daného jádra. Také u ARMu, stejně jako u jiných došlo v nedávné době k rozšíření na 64-bitovou architekturu, čímž se této platformě otevřely nové trhy a dnes již proniká i do serverů a výkonných pc, kde je zajímavá svou nižší spotřebou ve srovnání s architekturou x86. Dokonce i firma Microsoft uvedla Windows RT pro architekturu ARM.

ARM jádra jsou licencována jako IP[6] a firma jako taková nevyrábí fyzické výrobky. Firma umožňuje licencovat jejich implementaci jádra (ať už ve formě před nebo po syntetizaci), nebo také ARM architektury jako takové, kde implementace jádra leží plně na bedrech nakupující společnost. Takto vyvinuté jádro musí samozřejmě plně splňovat danou ARM specifikaci.

[4] Field-programmable gate array
[5] Advanced RISC Machines
[6] Intellectual property – je prodávána znalost, ne fyzický výrobek



Obrázek 2.5: Více jádro ARM Cortex A9 [4]

Základní vlastnosti

- ARM je load/store architektura s 32 bitovými instrukcemi, později rozšířenými o 16 bitové Thumb instrukce
- Neobsahuje predikci skoku, ale umožňuje využít podmíněné vykonávání
- Obsahuje pipeline – starší implementace používaly tří stupňovou, novější mají pipeline hlubší.
- Umožňuje jednoduché rozšíření o koprocesory
- Obsahují debug rozhraní

Dnes architektura ARM obsahuje mnoho variací s různými rozšířeními a instrukčními sadami A32, A64 a T32 (Thumb) ve verzi 1 a 2, což z ní dělá rozsáhlou a komplexní rodinu. Kvůli tomu je, hlavně pro začátečníka, složitější se v ní vyznat. Nabízí ale dobré vlastnosti pro široké použití a to jí dopomohlo k dnešní vysoké popularitě.

2.4 Boot

Po spuštění počítače se velká většina komponent včetně periférií nachází buď v náhodném, nebo výchozím stavu. Ať tak či tak, je třeba provést inicializace a nastavit jednotlivé komponenty do námi požadovaného stavu. Prvotní inicializace se provádí v rámci tzv. boot procesu. Tento proces je závislý na použité platformě, komponentách a programu nebo systému.

2.4.1 Platforma kompatibilní s IBM PC

Na nejrozšířenější pc platformě IBM PC s procesory architektury x86 probíhá boot v několika krocích.

BIOS

První z nich je vykonání programu BIOS[7], případně na novější verzích jeho inovovaná a výrazně vylepšená verze UEFI[8]. BIOS je dodáván spolu se základní deskou a jedná se v podstatě o velmi primitivní operační systém, který provede POST[9] a inicializaci hardwaru, „spustí“ program nazývaný bootloader (podrobnosti níže) a poskytuje mu velmi primitivní rutiny pro základní obsluhu dostupných zařízení, případně jejich podmnožině, jsou-li použity jemu neznáme rozšiřující periferie.

BIOS je načítán z paměti umístěné na základní desce, ze které se po restartu procesoru načítají první instrukce k vykonání. BIOS také poskytuje rozhraní pro jeho konfiguraci, v rámci kterého se nastavuje mj. pořadí prohledávání připojených paměťových médií, na kterých se hledá boot sektor. Pokud nedojde k nalezení zařízení obsahujícího boot sektor, vyhlásí BIOS chybu. Pokud je takové zařízení nalezeno, načte se z něj do paměti na adresu 0x7C00 prvních 512 B a skočí na zmíněnou adresu. Poté už pouze poskytuje níže zmíněné rutiny. Tyto rutiny jsou volány za pomocí přerušení s parametry předanými v definovaných registrech. Rutiny poskytované BIOSem [20]:

- Přerušení 10h – Rutiny pro obsluhu grafického výstupu
- Přerušení 11h – Vrací systémové informace
- Přerušení 12h – Vrací informace o paměti
- Přerušení 13h – Nízko úrovňové diskové operace
- Přerušení 14h – Obsluha sériového portu
- Přerušení 15h – Různé systémové rutiny
- Přerušení 16h – Obsluha klávesnice
- Přerušení 17h – Obsluha tiskárny
- Přerušení 1Ah – Rutiny reálného času a obsluhy PCI

Další informace lze nalézt např. v [8], pro UEFI [26] a celý proces startu počítače (a mnoho dalšího) je popsáno např. v [22].

Zavaděč

Zavaděč, anglicky bootloader, je program, který slouží k nahrání jádra operačního systému do paměti. Pokud to jádro očekává, předává taká parametry, jako např. velikost a rozložení paměti.

U IBM PC modelu 5150 bylo pro zavaděč vyhrazeno 512 B v paměti. Protože 512 B je málo, umisťuje se zde pouze technický zavaděč, který provede načtení hlavní části zavaděče. Toto rozdělení vydrželo dodnes, ale dnes existují mnohem vyspělejší zavaděče s grafickým rozhraním a možností konfigurace. Tyto zavaděče umožňují např. volit mezi různými operačními systémy, měnit parametry předávané jádru atd. Zavaděče se umisťují nejčastěji hned za boot sektor, kde bývá nevyužité místo před začátkem prvního oddílu. Jedná-li se

[7] Basic Input/Output System
[8] Unified Extensible Firmware Interface
[9] Power-on self-test – provede základní kontrolu hardwaru

o systém s UEFI, případně o paměťové médium s GPT[10], jsou detaily boot procesu mírně odlišné.

Protože jádro je již umístěno na partition se souborovým systémem, je třeba, aby zavaděč dokázal s tímto souborovým systémem pracovat. To ale není jeho jediná pokročilá schopnost. Protože rutiny poskytované BIOSem jsou velmi primitivní, bývá práce s nimi také pomalá. Proto moderní zavaděče obsahují některé ovladače a jsou schopny používat hardware daleko efektivněji než samotný BIOS, který musí poskytovat zpětnou kompatibilitu.

Poté co zavaděč nahraje jádro operačního systému, případně také výchozí ram disk a do paměti přidá parametry pro jádro, skočí na první instrukci jádra a jeho práce končí.

2.4.2 Boot s jádry ARM

Protože procesory s jádrem ARM mohou mít různé rozhraní a nemají ustálenou platformu jako je IBM PC compatible pro procesory x86, neexistuje zde definovaný bootovací proces pro celé zařízení jako takové, ale je třeba dodržet kroky pro inicializaci ARM jádra. Zbytek již závisí na použitém hardwaru.

Protože na rozdíl od PC je zařízení s jádrem ARM dodáváno kompletní včetně softwaru, je na výrobci, aby zajistil jeho korektní funkcionalitu. I pro tyto zařízení existují zavaděče (jako např. U-Boot), které výrazně usnadňují zprovoznění operačního systému a vyžadují minimální úsilí při portování, protože již obsahují podporu nejčastějších procesorů.

Inicializace ARM systému se obecně skládá ze dvou kroků [1]:

1. Inicializace prostředí, jako např. vektory přerušení, zásobník, v/v zařízení
2. Inicializace prostředí a knihoven pro běh aplikací napsaných ve vyšším programovacím jazyce, jako je jazyk C

Postup inicializace se samozřejmě liší, pokud dochází ke spuštění operačního systému, nebo pouze embedded aplikace. Protože operační systém provede velkou část inicializace a konfigurace sám podle svých potřeb, zatímco pro embedded aplikaci je třeba znát její požadavky a těm se přizpůsobit.

Po resetu procesor začne vykonávat kód na adrese 0x0 a je třeba skočit na inicializační kód, který musí:

- Nastavit vektory přerušení
- Inicializovat paměťový systém
- Inicializovat registry s ukazateli na zásobník
- Inicializovat potřebné v/v periférie
- Nastavit mód procesoru, je-li to třeba
- Nastavit stav procesoru, je-li to třeba

Následně je již možné používat programy napsané ve vyšším jazyce.

V ARM knihovně existuje tzv. Boot Monitor, který provádí potřebné operace a lze jej využít. Samozřejmě je potřeba jej mírně upravit, aby vyhovoval použité platformě. Více se lze dočíst v kapitole „Boot Monitor configuration“ v [1].

[10] GUID Partition Table

2.4.3 Firmware

Označení firmware se používá u embedded zařízení. Je to souhrnné označení pro software daného zařízení bez ohledu na to, jaká je jeho reálná struktura. Dříve se velmi často používal speciální software vyvinutý pouze pro dané zařízení, ale dnes se stále častěji setkáváme s tím, že dané zařízení obsahuje operační systém a v něm běžící aplikaci řídící jeho chod.

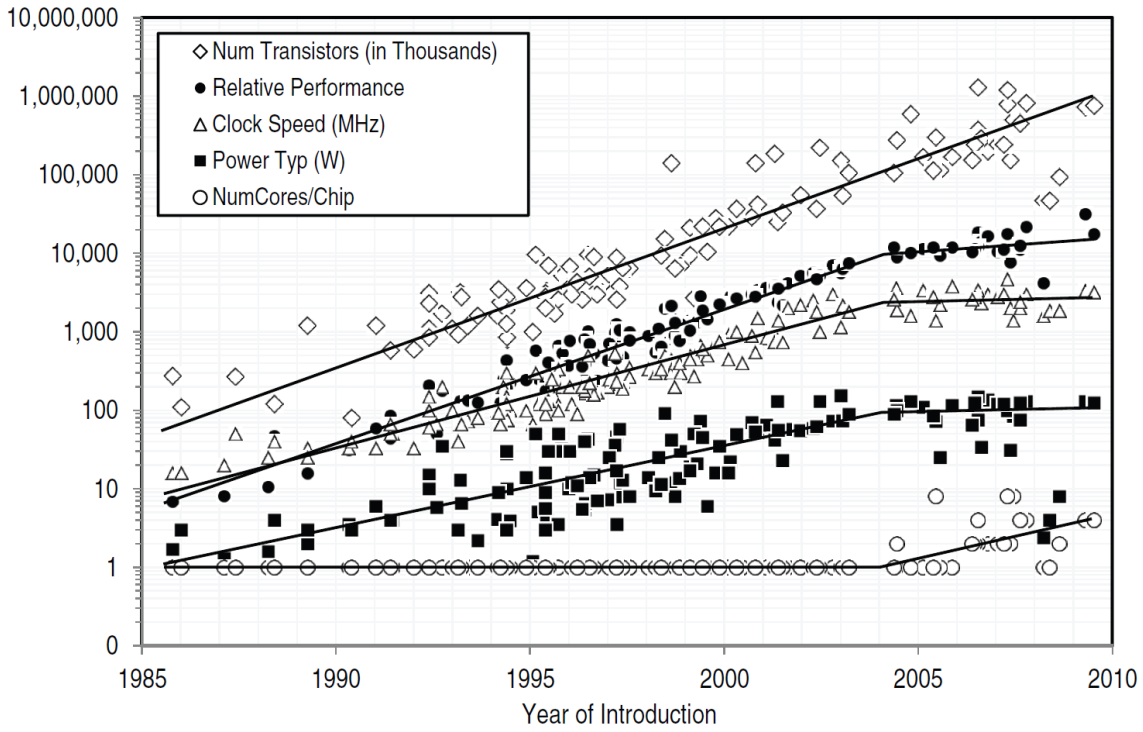
Ať je struktura firmwaru jakákoliv, pro uživatel až na výjimky neexistuje možnost jeho úpravy a v případě jeho updatu se provádí za pomocí jediného souboru, který bývá zpracován samotným firmwarem – nejčastěji jeho prostým nahráním do interní paměti a restartem zařízení, po jehož proběhnutí již naběhne nový systém. Tím je zajištěna jednoduchá obsluha a vysoká odolnost vůči nechtěným chybám.

U některých zařízení je firmware stálý po celou dobu životnosti zařízení, protože neumožňují jeho update (případně pouze v servise). Proto u něj bývá prováděno rozsáhlejší testování a jsou u něj vyšší požadavky na jeho kvalitu. S rozšiřujícími se „connected“ zařízeními se stále častěji vyvíjí systémy umožňující tzv. Over-the-air (OTA) aktualizace – je-li vydán nový firmware, je automaticky stažen z internetu a zařízení je aktualizováno.

Kapitola 3

Od jedno k více jádrům

Jako v každé oblasti, i v návrhu architektury se člověk snaží dosahovat stále lepších výsledků a proto je neustálý tlak na zvyšování výkonu. K tomu existuje několik cest. Jednak se stále zlepšují staré technologie (jako např. změna materiálu pro dosažení lepších fyzikálních vlastností a tím snížení příkonu, nebo minimalizace) a zároveň se vyvíjí nové technologie a postupy. S postupem času se ale čím dál více blížíme fyzikálním limitům, které platí pro momentálně používané materiály využívající křemíkové tranzistory.



Obrázek 3.1: Historický vývoj procesorů [9]

Z pohledu procesorů se mezi jinými uplatňovala cesta zvyšování pracovní frekvence. Touto cesto ale nešlo jít navždy, protože se zvyšuje odpadní teplo. Příkon digitálního CMOS obvodu je:

$$P = \alpha \cdot C \cdot V^2 \cdot f \tag{3.1}$$

kde P je příkon, α je faktor aktivity, C je kapacita obvodu, V je napětí a f je frekvence. Protože převážná většina příkonu je přeměněna na teplo, je z rovnice jasné, že každé zvýšení frekvence způsobí násobení příkonu a tedy i odpadního tepla. A protože materiál a chladící soustava je schopna odvést maximálně určité množství tepla, nevede tudy cesta do nekonečna. Další limitující faktor je rychlost pohybu elektronů, kterou také neovlivníme.

Pro co nejnižší příkon jsou dnes tyto parametry ovlivňovány také za běhu. Nejčastěji je snižováno napětí (DVS[1]) a spolu s ním i frekvence, protože při snížení napětí dojde ke zvýšení doby změny stavu procesoru a tím i jeho maximální frekvence. Při snížení napětí jsme navíc odměněni dvakrát, protože se jedná o druhou mocninu. Další variantou je využívání speciálních nízkopříkonových stavů procesoru (DPM[2]), které vypínají různé části procesoru a šetří tak energii. U těch je problémem režie spojená s přechodem mezi stavy. Parametr α lze také ovlivnit. Jeho změna je vedlejším efektem změny frekvence, ale lze jej upravit i přímo. Pro jeho snížení je potřeba zajistit menší počet změn z log. 0 na log. 1 a naopak. K tomu je třeba znát detailně hardwarovou implementaci procesoru a adekvátně upravit kód. Jednou z prováděných úprav je např. změna kódování adres na sběrnici. Při této změně se použije např. Grayův kód, kde po sobě jdoucí adresy mají vždy pouze jeden bit odlišný. Na rozdíl od běžného binárního kódování je při zvyšování adresy aktivita malá a konstantní, protože se vždy mění pouze jeden bít, na rozdíl od jednoho až všech u běžného binárního kódu.

V grafu na obrázku 3.1 lze vidět data z let 1985 – 2010 pro počet tranzistorů, výkon, frekvenci, příkon a počet jader na čipu. Z křivek je krásně vidět závislost frekvence a příkonu spolu se změnou jejich sklonu při dosažení hraničních hodnot. Ve stejnou dobu se pro změnu začíná uplatňovat paralelismus.

Paralelismus přináší řešení zmíněných problémů, protože za relativně malou cenu můžeme získat výrazné navýšení výkonu. Výhody paralelismu jsou ale silně degradovány, pokud daná aplikace umožňuje minimální, nebo dokonce žádné možnosti k jeho využití. Na výpočet možného zrychlení existuje rovnice Amdahlova zákonu:

$$S = \frac{1}{F + \frac{1-F}{N}} \tag{3.2}$$

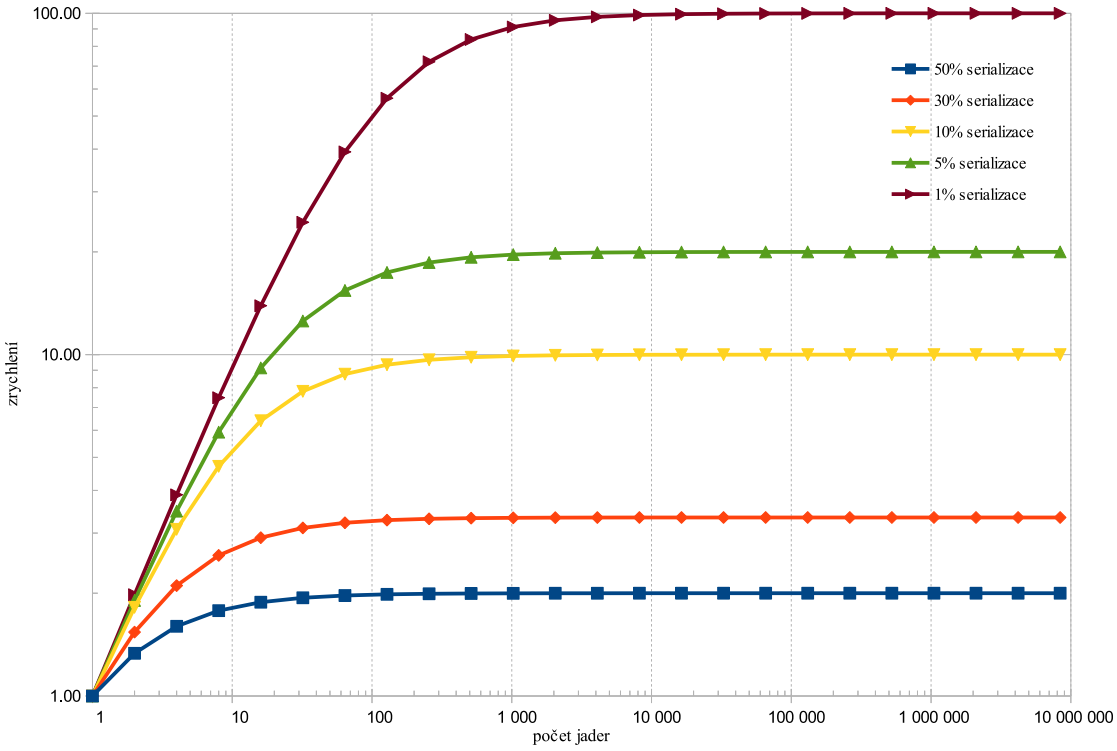
kde S je zrychlení, N je počet procesorů/jader a F je faktor určující část aplikace, kterou nelze vykonat paralelně. Z grafu 3.2 lze vidět, že i malé procento serializace způsobuje velkou ztrátu potencionálního zrychlení – i když je sériově vykonáno pouze jedno procento aplikace, zrychlení nikdy nepřesáhne 100.

3.1 Různé formy paralelismu

V dnešní době využíváme různé formy paralelismu, mezi něž patří:

- SIMD, MISD, MIMD[3]
- Zřetězení
- Superskalarita
- Vykonání instrukcí mimo pořadí

[1] Dynamic Voltage Scaling
[2] Dynamic Power Management
[3] single/multiple instruction, single/multiple data



Obrázek 3.2: Amdahlův zákon

- Hyper-threading
- Vícejádrová architektura
- Víceprocesorová architektura

Prvních pět jsou různé formy paralelismu na úrovni instrukcí (ILP[4]) a poslední dvě jsou již čistě paralelní přístupy. Všechny výše uvedené lze samozřejmě kombinovat a dnešní procesory toho plně využívají.

3.1.1 Paralelismus na úrovni instrukcí

SIMD, MISD, MIMD

Mezi prvními paralelními technologiemi se objevilo SIMD [10]. Jeho využití je ale velmi specifické a dnes se nejvíce využívá při zpracování různých signálu, jako např. videa a audia. U nich je totiž potřeba provádět tytéž operace nad velkým množstvím podobných a navzájem nezávislých dat. Typickým příkladem jsou DSP procesory a grafické karty. Ale i běžné procesory obsahují instrukce pro SIMD, kdy se např. místo 32 b proměnné pracuje s čtyřmi 8 b proměnnými apod. Jejich procentuální využití v běžném procesoru je ale malé a proto se jedná o rozšíření architektury a ne její jádro, na rozdíl od DSP a grafických karet.

MIMD se vyskytuje v menší míře, ale stále se vyskytuje u různých VLIW procesorů. Zde je velmi důležitá práce překladače, který musí být schopen dostupné prostředky plně

[4]instruction level parallelism

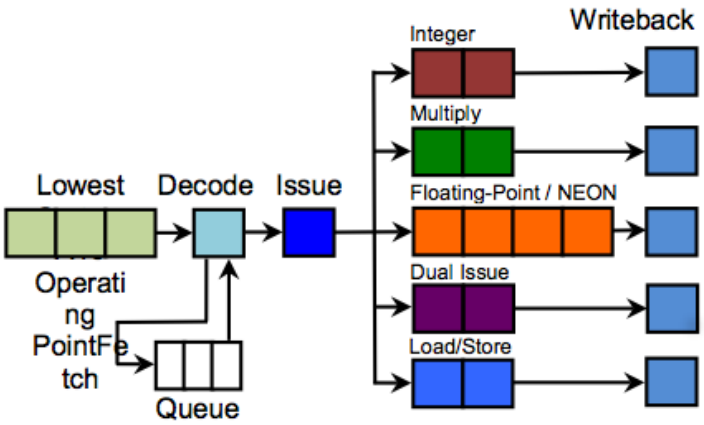
využít. Toto dokáže jen tehdy, když z poskytnutého kódu dokáže vyextrahovat co nejvíce navzájem nezávislých instrukcí a ty „poskládat“ do jedné velké instrukce.

MISD technologie není příliš rozšířená, protože neexistuje mnoho aplikací pro její využití.

Zřetězení

Zřetězení je v podstatě velmi jednoduchá technologie rozložení velkého problému na několik menších. Procesor vykonává každou instrukci v několika krocích. Navzájem lze vykonávat několik instrukcí, kdy se každá nachází v jiném kroku. Tím dojde ke zkrácení doby mezi dokončením dvou instrukcí za předpokladu, že je již rozpracováno více instrukcí (tj. pipeline je již naplněná). Konkrétní rozdělení instrukcí na menší kroky se liší pro různé procesory. Na obrázku 3.3 je zobrazeno rozdělení pro ARM Cortex A7.

U této technologie je velkou penalizací potřeba naplnit frontu a proto mohou způsobovat podmíněné skoky zpomalení výpočtu. Aby se tomu předcházelo, byla vyvinuta technologie predikce skoku, která má za úkol odhadnout cíl skoku, aby nedocházelo k zahození již rozpracovaných instrukcí.



Obrázek 3.3: ARM Cortex A7 pipeline [3]

Superskalarita

Tato technologie se snaží naplno využít různé funkční bloky procesoru (jako je sčítačka nebo násobička) tím, že je možné vykonávat několik instrukcí navzájem za předpokladu, že každá používá jiný funkční blok a instrukce nejsou navzájem závislé. Veškeré operace spojené s využitím superskalarity jsou prováděny na úrovni procesoru a ten proto musí být schopen vyhodnotit závislosti mezi instrukcemi, což je zásadní rozdíl proti MIMD, kde jsou tyto závislosti vyhodnocovány při překladu. Je jasné, že čím více funkčních bloků procesor má, tím větší potenciál superskalarita poskytuje.

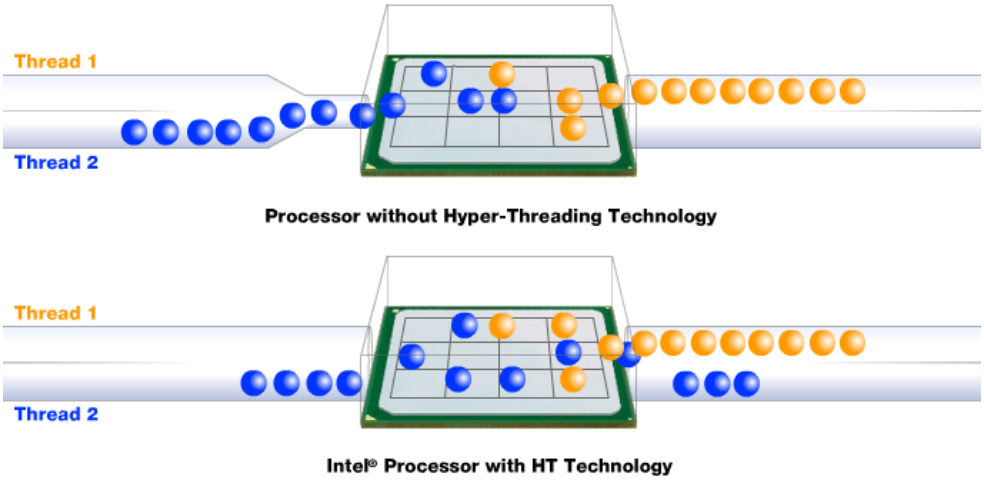
Sureskalarita může být dále vylepšena za pomocí out of order execution, která dále prohloubí možnosti využití více bloků.

Vykonání instrukcí mimo pořadí

Protože pořadí příkazů poskytnuté programátorem (a následně překladačem) nemusí být z pohledu použité architektury ideální, byl vyvinut postup, jak vykonávat instrukce v jiném, než poskytnutém pořadí. Samozřejmě za předpokladu, že výsledek bude stále stejný. Veškerá snaha spojená s out of order execution leží na bedrech procesoru, který musí být schopen vyhodnotit závislosti instrukcí a zajistit, že nedojde ke změně výsledků. To sebou obnáší složitou vyhodnocovací logiku navíc a proto není tato technologie dostupná u jednodušších procesorů.

Hyper-threading

Další z technologií snažící se využít procesor na maximum je Hyper-threading [11]. Při ní se u každého jádra procesoru zdvojí řídící část. Procesor se potom tváří jako dvou jádrový a umožňuje vykonávat dva procesy najednou. Tyto logické jádra sice sdílí funkční bloky, ale protože vykonávané procesy jsou datově nezávislé a s velkou pravděpodobností obsahují různé druhy instrukcí, lze jednotlivé instrukce spouště najednou na různých funkčních blocích procesoru. To, podobně jako u superskalarity, umožňuje zvýšit využití obvodů procesoru a tím zvýšit výkon.



Obrázek 3.4: Hyper-threading [11]

3.1.2 Víceprocesorová a vícejádrová architektura

Vzhledem k minimalizaci je vícejádrová architektura přirozeným vývojem. Protože se uvolňuje místo na čipu, je jednoduché přidat další jádro nebo paměť. To s sebou nese samozřejmě problémy spojené s paralelními architekturami, jako např. synchronizace nebo paměťová koherence.

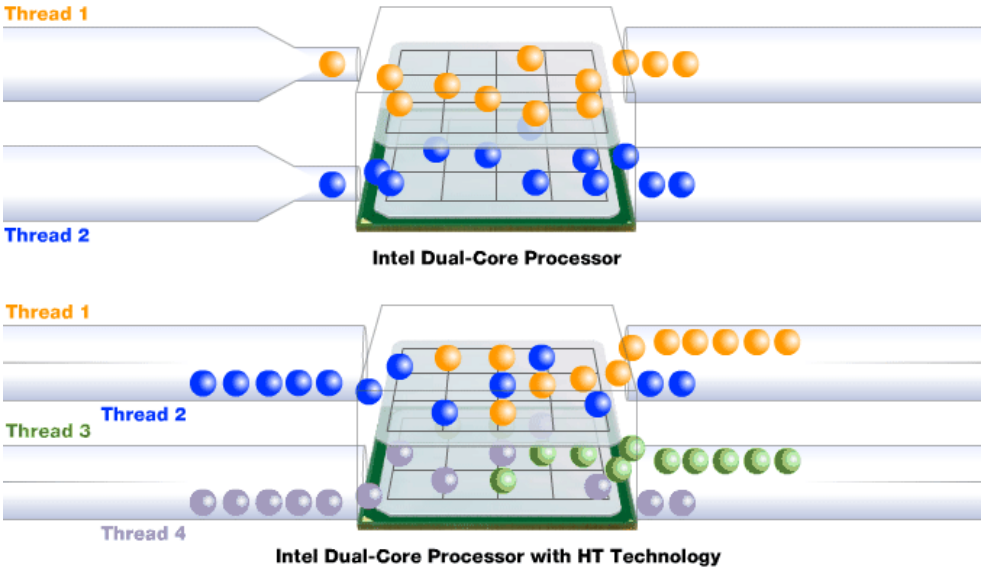
Paměťová koherence je jeden za zásadních problémů pro škálovatelnost a řeší ji několik různých algoritmů, které se snaží s minimálními dopady na výkon zajistit softwarově transparentní řešení. Mezi ně patří různé odposlouchávací a jiné mechanizmy, které zneplatňují nebo aktualizují svou kopii dat, jako např. write-once, Firefly, MSI, MESI, MOSI, MOESI[5], nebo různé adresářové systémy. Druhým velkým problémem je sdílená paměť,

[5]Modified (M), Owned (O), Exclusive (E), Shared (S) a Invalid (I)

která většinou umožňuje maximálně jedno čtení nebo zápis za jednotku času a proto jsou požadavky jader serializovány.

Víceprocesorová architektura je v podstatě totéž, jako vícejádrová, ale jedná se o více samostatných procesorů. Každý z těchto procesorů může být samozřejmě vícejádrový. U dnešních superpočítačů se počet procesorů pohybuje v řádech deseti tisíců s až 16 jádry. Takovéto superpočítače už se samozřejmě nenachází společně v jednom boxu na jedné desce, ale spíše v jedné hale. Proto je mezi nimi velmi rychlá propojující síť a celá architektura je vyvinuta na míru spolu se softwarem.

U víceprocesorových systémů existují dvě základní skupiny - Symetrická (SMP[6]) a asymetrická (AMP[7]) vícejádrová architektura. U vícejádrových systému se jedná v podstatě vždy o SMP.



Obrázek 3.5: Dvou jádro bez a s Hyper-threadingem

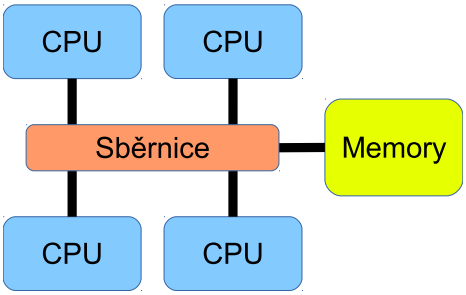
Na více jádrových a procesorových architekturách je také třeba mírně upravit boot proces. Nejčastějším řešením je inicializace systému pouze jedním jádrem, zatímco ostatní čekají na synchronizační signál. Proto tyto architektury umožňují každému jádru zjistit svůj typ (bootstrap nebo aplikační), případně číslo, podle kterého se dále rozhoduje. Aplikační jádra jsou později uvolněna nebo probuzena k běhu operačním systémem.

SMP

Jak už napovídá název symetrická, jedná se o takovou architekturu, kde jsou všechna jádra stejná a jsou připojeny k jedné hlavní paměti s rovným přístupem k v/v zařízením a přerušením. Z pohledu operačního systému jsou si všechny jádra rovny, ačkoliv např. přerušení jsou standardně obsluhována pouze jedním z důvodu paměťové lokality. Na SMP se také nejčastěji spouští pouze jeden operační systém, který obsluhuje všechny jádra. Na osobních počítačích se většinou vyskytují jednotky jader a je zde v silné míře používán multitasking.

[6] Symmetric multiprocessing
[7] Asymmetric multiprocessing

Proto se často stává, že jedna aplikace běží na jednom jádru a tomu je přizpůsobeno i plánování, kdy je většinou cíl nemigrovat aplikace mezi jádry, opět z důvodu paměťové lokality. Samozřejmě i zde existují aplikace schopné využít všechny dostupné jádra naplno a je jim to umožněno. Stačí programovat aplikace s více procesy a/nebo více vlákny.



Obrázek 3.6: SMP struktura

Vícejádrová a víceprocesorová architektura samozřejmě vyžaduje patřičně schopný operační systém. Například v Linuxu bylo třeba zajistit ochranu sdílených zdrojů (paměť, zařízení, ...) pomocí zámků, upravit boot proces, plánovač a obsluhu přerušení. Ve verzi 2.6 byl použit plánovač O(1) s migrací procesů mezi jádry, později nahrazený plánovačem CFS.

AMP

U asymetrických, na rozdíl od symetrických, existuje jistá disproporce mezi jednotlivými jádry. Ať už v přístupu do paměti (NUMA[8]), přístupu k v/v zařízením, nebo vlastní architekturou jader nebo procesorů. Na AMP se často spouští více operačních systémů, kdy každý je přizpůsoben pro své jádro nebo jádra a navzájem mezi sebou komunikují. Toto je výhodné zejména proto, že se může výrazně lišit instrukční sada a zaměření jednotlivých procesorů a lze tak dosáhnout vyšší výkonosti než s univerzálním operačním systémem.

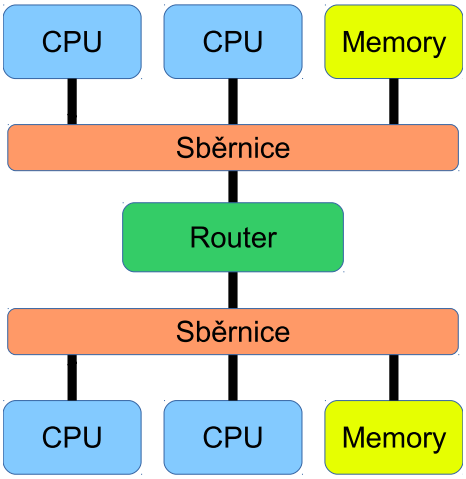
AMP se objevilo dříve než SMP, protože se jednalo o jednoduchou cestu, jak zvýšit výkon počítače a příliš při tom nekomplikovat architekturu a operační systém rovným přístupem ke všem procesorům. Také byla tato cesta levnější, než přidání dalšího počítače. Druhý procesor zde byl využíván pouze jako pomocná výpočetní síla a proto neměl a nepotřeboval např. přístup k periferiím.

Dnes se AMP vyskytuje např. u superpočítačů, které jsou složeny z jednotlivých clusterů s vlastními paměťmi a jedná se tudíž o NUMA. Tyto clustery jsou propojený rychlou sítí a každý cluster má vlastní instanci operačního systému. Do budoucna se AMP v rozsáhlejší míře pravděpodobně prosadí v mobilních zařízení, kde bude kombinovat vysoký výkon s dlouhou výdrží při malém zatížení za pomoci různých procesorů a jejich selektivnímu používání na základě požadavků uživatele. Příkladem této technologie je „big.LITTLE Processing“ od firmy AMR [3].

Propojovací sítě

Protože jednotlivé jádra potřebují navzájem komunikovat a přistupovat ke sdíleným zdrojům, jako např. paměť, je třeba je navzájem propojit. K tomu existují různé typy sítí, kdy každá přináší rozdílné vlastnosti. Obecně lze sítě rozdělit podle použití pro malý počet

[8] Non-uniform memory access



Obrázek 3.7: NUMA struktura – dva clustery propojené pomocí sítě

(jednotky, max. desítky) jader a rozsáhlé systémy (stovky až tisíce jader). Malé systémy lze dále dělit na propojení na čipu (vícejádrový procesor) a propojení na desce (víceprocesorový systém).

Každá ze zmíněných skupin většinou používá různé techniky propojení zejména z důvodu složitosti implementace a tím i ceny. Mezi sledované parametry u těchto sítí patří zpoždění při komunikaci různých uzlů a šířka pásma. Tyto parametry je třeba uvažovat jak na nevytíženém, tak především na vytíženém systému. S těmito parametry souvisí také pravidelnost a symetrie propojovací sítě, množství možných cest, diametr a další parametry. Jedná se v podstatě o stejné přístupy, jaké se používají u síťových propojení a teorie grafů. Na úrovni rozsáhlých systémů se již jedná přímo o síťové propojení mezi jednotlivými clustery.

Mezi používané propojovací sítě patří například:

- 2D mříž
- 2D torus
- Hyper kostka
- Síť motýlek
- Benešova síť
- Kružnice
- Křížový přepínač
- HyperTransport
- a mnohé další

Celý problém propojovacích sítí a jejich vlastností je velmi rozsáhlý a není předmětem této práce. Proto zde není do podrobna rozebírán.

Meziprocesorová komunikace

Výměna dat mezi jádry může probíhat buď za pomoci sdílené paměti, nebo za pomocí posílání zpráv. Obě metody mají své výhody a nevýhody.

Při využití sdílené paměti nejsou data duplikována a všichni vidí a pracují nad stejnými daty v paměti. Je ale třeba zajistit, aby nedošlo k vícenásobnému přístupu k datům, např. za pomoci semaforu. Některé architektury umožňují vícenásobné čtení dat, to ale může způsobovat vyhladovění procesů pokoušejících se o zápis a je třeba tomu přizpůsobit použité algoritmy.

Na druhou stranu posílání zpráv vyžaduje duplikaci dat u každého účastníka komunikace a data nemusí být aktuální, protože je třeba je explicitně zaslat ostatním účastníkům. Není ovšem nutné data zamykat proti vícenásobnému přístupu a celá komunikace je explicitní a lze ji jednoduše sledovat. Také lze komunikaci provádět asynchronně a bez zásahu procesoru. Zároveň je ale třeba počítat s vyšší režií způsobou potřebou provedení komunikace, na rozdíl od pouhého zápisu dat do paměti a jejich okamžitému zpřístupnění ostatním komunikujícím.

V případě explicitní potřeby lze obě metody softwarově (a případně i hardwarově, je-li k tomu uzpůsoben) simulovat nehledě na reálnou implementaci systému, ačkoliv to sebou nese další dodatečnou režii.

3.2 Změny v operačním systému

Jak už bylo diskutováno na začátku této kapitoly, rozumné využití paralelismu je možné pouze při minimálním množství sériově vykonávané činnosti. V případě systémů s mnoha procesy toho lze lehce docílit paralelním vykonávání těchto procesů. Toto se používá u serverů a osobních počítačů, kde je jen minimální množství programů schopno využít paralelismu v dostatečné míře. Další příkladem jsou smartphony a tablety, které se dnes výrazně přibližují osobním počítačům.

U vestavěných systému je situace trochu složitější, protože jednoduché systémy s jednou úlohou nejsou schopny paralelismus využít. Jedná-li se o složitější systém, který zpracovává data z více zdrojů, případně jedná-li se o systém zpracovávající velké množství dat je situace příznivější.

Dnes se paralelismus nejvíce využívá u superpočítačů, které zpracovávají ohromné množství dat a u serverových farem, na kterých běží obrovské množství nezávislých procesů.

3.2.1 Plánování

Na základě přístupu k plánování napříč procesory lze plánování dělit na:

- Plánování s migrací
  - S přesunem procesů
  - Bez přesunu procesů
- Plánování bez migrace

Plánování bez migrace neumožňuje přesun úloh ani procesů mezi jednotlivými procesory nebo jádry. Naproti tomu plánování s migrací ano, přičemž se ještě dále dělí na verzi s přesunem procesů, kde není migrace procesů nijak limitována a na verzi bez migrace procesů.

U té platí, že je-li proces dané úlohy spuštěn na jistém jádře, běží vždy na tomto jádře, dokud neskončí.

Plánování bez migrace může být výhodné pro real-time aplikace v případě, že dopředu známe veškeré úlohy, jejich čas příchodu do systému a dobu trvání. Lze totiž úlohy manuálně rozdělit mezi jádra tak, aby byly optimálně využité.

U plánování s migrací je třeba brát v potaz paměťovou lokalitu, protože v případě přesunutí procesu na jiné jádro může dojít k výraznému zatížení paměťového systému způsobeného zpětným zápisem dat z cache jednoho jádra a jejich načítání do cache druhého jádra. Z toho důvodu se většinou plánovače snaží umisťovat procesy stále na jedno jádro a až v případě výraznějšího rozdílu v zatížení jader dochází k přesunu některých procesů.

Totéž platí pro procesy obsluhy přerušení. Je-li přerušení určitého zařízení obsluhované jedním jádrem, není dobré při příštím přerušení téhož zařízení provádět obsluhu na jiném jádře. Například u síťového rozhraní mohou být příchozí data přijímány v rámci více přerušení a při případné migraci mezi jádry by bylo nutné přenášet již přijatá data mezi jednotlivými jádro podle toho, které zrovna provádí obsluhu daného přerušení a kompletaci dat. Proto přerušení nejčastěji obsluhuje pouze jedno jádro, případně jsou různé zařízení přiděleny různým jádrům.

Problém ovšem nastává v případě, kdy je některé jádro výrazně zatíženo obsluhou přerušení, protože s tímto zatížením nepočítá plánovač. Považuje toto jádro za nezatížené a snaží se mu přidělit více procesů, než je schopno reálně obsloužit.

Plánování real-time aplikací přináší další specifické problémy a některé mechanizmy real-time plánování pro více procesorů jsou uvedeny např. v [23].

3.2.2 Paměť

Z pohledu paměti nejsou přímé zásahy kvůli použití systému na vícejádrovém procesoru nutné. Většina systémů by fungovala dále, ale z důvodu uvedených níže by došlo pouze k minimálnímu zrychlení a je proto třeba optimalizovat některé metody práce s pamětí. Samozřejmě také záleží na použité architektuře, protože pokud nemá např. hardwarovou podporu zajištění koherence paměti, je třeba toto zajistit za pomoci softwaru, což si vyžaduje výrazný zásah při práci s pamětí.

Jediným vzniklým problémem je reálné soutěžení o prostředky, které na jednoprocesorovém systému existuje také, ale jeho vliv na výkon je prakticky nulový z důvodu serializace. Pravděpodobnost přerušení procesu v kritické sekci a naplánování dalšího soutěžícího procesu je u jednojádrových systémů malá. Naproti tomu u vícejádrového systému je pravděpodobnost současného běhu soutěžících procesů mnohem vyšší.

Další s tím spojeným problémem je paměťová lokalita. Např. při soutěžení dvou a více procesů o zámek na standardní jedno jádrové implementaci, může docházet k neustálém požadavku o exkluzivní kopii dat zámku jednotlivými jádry při pokusu o „test and set“ instrukci. To způsobí vysoké zatížení paměťového systému a výraznou degradaci výkonu. Tento problém existuje hlavně u aktivního čekání, které se používá u krátkých kritických sekcí. Proto se používá optimalizovaná implementace, kdy se před pokusem o „test and set“ instrukcí nejprve pouze čte hodnota zámku. Tím dojde ke změně dat zámku na sdílené a každé další čtení již není spojeno s přesunem dat mezi jádry. Poté, co jsou data zámku změněna (zámek byl uvolněn), dojde k zneplatnění dat u ostatních jader, ty si je aktualizují a pokusí se zámek získat.

Problémy spojené s cache jsou dále ovlivněny strukturou architektury. Např. v případě, že jádra sdílí cache druhé úrovně, může být problém lokality dat méně palčivý, než v případě

```
...
while(TestAndSet(&lck, 0));
Critical_Section
Release(&lck);
...
```

```
...
do {
        while(Test(&lck));
        if(TestAndSet(&lck, 0))
                continue;
        Critical_Section
        Release(&lck);
        break;
} while(true);
...
```

Kód 3.1: Aktivní čekání bez/s optimalizací pro vícejádrovou architekturu s cache

dvou nezávislých procesorů, které sdílí pouze hlavní paměť. Optimalizace ale povede v obou případech ke zvýšení výkonu, takže je tak jako tak vhodné ji provést.

3.2.3 Jádro

Úpravy jádra jsou potřeba zejména kvůli zajištění zdrojů před vícenásobným přístupem v případě, že dojde k zavolání jádra na více procesorech najednou. K tomu se používají klasické techniky paralelního programování, případně lze omezit rutiny jádra pouze na jeden procesor a tím zajistit maximálně jeden přístup ke sdíleným proměnným v čase.

Z pohledu periferií je ochrana prostředků zajištěna jádrem, takže u ovladačů není třeba speciální úpravy. Co se týče obsluhy přerušení, je potřeba zajistit, aby bylo obslouženo maximálně jednou. To může být zajištěno buď hardwarově, nebo softwarově. U architektury ARM je toto zajištěno přímo v hardwaru. V případě, kdy je více jader přerušeno najednou, každé dostane jeden vektor z fronty aktivních přerušení. Pokud je v danou chvíli vektorů méně než žádajících jader, dostanou zbývající jádra neplatný vektor přerušení [1].

3.3 Aplikace

Při programování aplikací není třeba speciálního přístupu, pokud programátor nevyžaduje paralelní výpočet. V opačném případě musí vyvinout vícevláknovou nebo víceprocesorovou aplikaci se vším všudy.

U překladače je situace trochu složitější. Aby byly využity některé ILP vlastnosti procesorů (zejména SIMD), je třeba kód speciálně připravit. To vyžaduje analýzu závislosti v kódu a mezi proměnnými a případné optimalizace. Dnešní překladače jsou ale v tomto ohledu připraveny a na programátoru je pouze volba velikosti optimalizace a následná kontrola korektnosti výsledku, zejména v případě, kdy se jedná o paralelní implementaci. Optimalizace překladače totiž mohou způsobit zpřeházení některých instrukcí a tím narušení kritických sekcí.

Kapitola 4

Zvolená platforma a RTOS

Jako zvolená platforma byl vybrán vývojový kit ZedBoard a real time operační systém µC/OS-II.

4.1 Popis Zedboardu

ZedBoard[1] je postaven kolem Zynq-7000 All Programmable SoC. Jeho hlavní vlastnosti jsou:

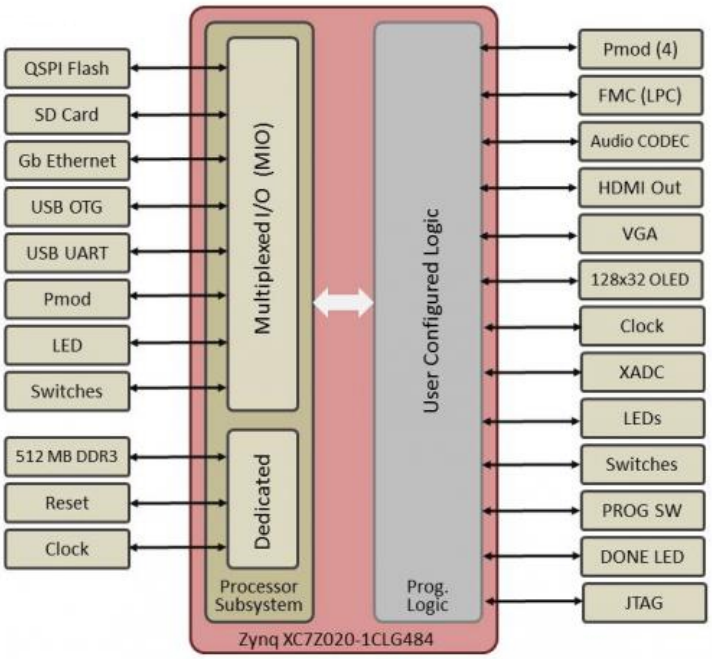
- Xilinx Zynq-7000 All Programmable SoC XC7Z020-CLG484-1
- Paměť
  - 512 MB DDR3
  - 256 Mb Quad-SPI Flash
  - 4 GB SD karta
- Onboard USB-JTAG programování a ladění
- 10/100/1000 Ethernet
- USB OTG 2.0 a USB-UART
- PS & PL V/V rozšíření (FMC, Pmod Compatible, XADC)
- Několik video výstupů (1080p HDMI, 8-bit VGA, 128 x 32 OLED)
- I^2S Audio Codec

Obrázek 4.1 zobrazuje blokový diagram kitu, obrázek 4.2 potom vlastní kit s popisky.

Jedná se o low-cost vývojový kit se širokým záběrem aplikace. Obsahuje dvě jádra ARM Cortex A9 a vše potřebné pro zprovoznění různých operačních systémů.

Deska plošných spojů (DPS) je v případě ZedBoardu desetivrstvá, přičemž čtyři jsou použity pro rozvod napájení a zbylých šest pro rozvod logických signálů. Protože DPS je pouze desetivrstvá, je uspořádaná ve formě hvězdy, aby bylo možné rozvést potřebných sedm napěťových okruhů v pouze dvou vrstvách (další dvě rozvádí zem) a zároveň zachovat co nejnižší množství šumu [5].

[1] Zynq Evaluation and Development Board



Obrázek 4.1: Blokový diagram ZedBoardu [6]

DPS také obsahuje několik přepínačů za pomocí kterých lze měnit konfiguraci Zynqu, jako například výběr bootovací paměti, nebo nastavení některých periferií. Detaily lze nalézt v [7].

Zynq na ZedBoardu lze programovat poskytnutým USB-JTAG rozhraním, případně lze použít přímo vyvedené JTAG rozhraní a obejít tak Digilent USB High Speed JTAG Module, který na desce poskytuje převod USB na JTAG. Alternativní možnost je načtení programu z externích pamětí.

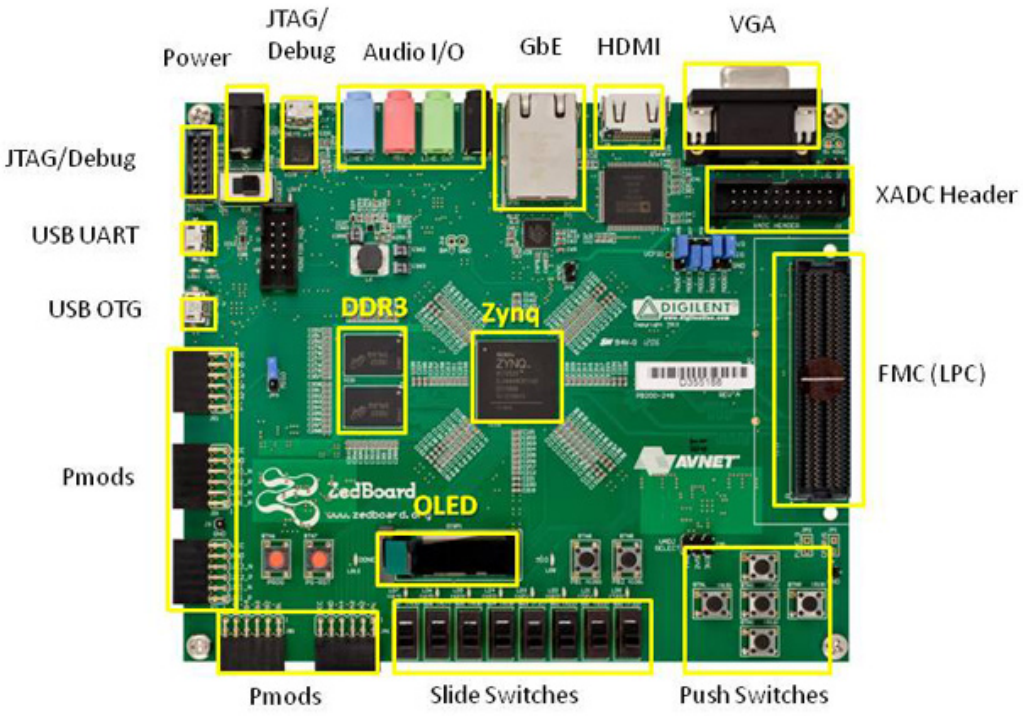
Další vlastnosti ZedBoardu kopírují jeho SoC, kterým je Zynq-7000, jehož detailní popis je uveden v kapitole 4.2.

4.2 Zynq-7000 All Programmable SoC

Zynq 7000 obsahuje dvě jádra Cortex A9 a kromě některých integrovaných rozhraní obsahuje také FPGA část, kterou lze využít pro rozšíření funkčnosti za pomoci IP a nebo vlastního designu. Jednoduchý diagram toho SoC je na obrázku 4.3. Xilinx u svého SoC používá rozdělení na dvě části [27]. První je „Processing System“ (PS) – aplikační procesor (APU), paměťové rozhraní, v/v periférie a jejich konfigurovatelné propojení. Druhým pak „Programmable Logic“ (PL) – FPGA část. Toto rozdělení budu dále používat také.

4.2.1 Vývojové prostředí

Pro vývoj na svých architekturách dodává Xilinx balík vývojových programů. Ty umožňují bezproblémový vývoj včetně využití FPGA části čipu. Protože Zynq je složitý čip s různorodými požadavky na vývoj, poskytuje Xilinx několik různých programů, přičemž každý je použit na specifickou část návrhu nebo vývoje.



Obrázek 4.2: Přední pohled na ZedBoard

Mezi programy v balíku ISE Design Suite je např. Project Navigator, který je rozcestníkem celého projektu, PlanAhead, který slouží k mapování v/v pinů, Xilinx Synthesis Technology pro syntetizaci VHDL nebo Verilogu, Design Suite pro schematický pohled a návrh obvodu, Platform Studio pro návrh embedded procesorů, Software Development Kit pro softwarový vývoj, ModelSim a ISim pro simulace obvodu a další podpůrné programy.

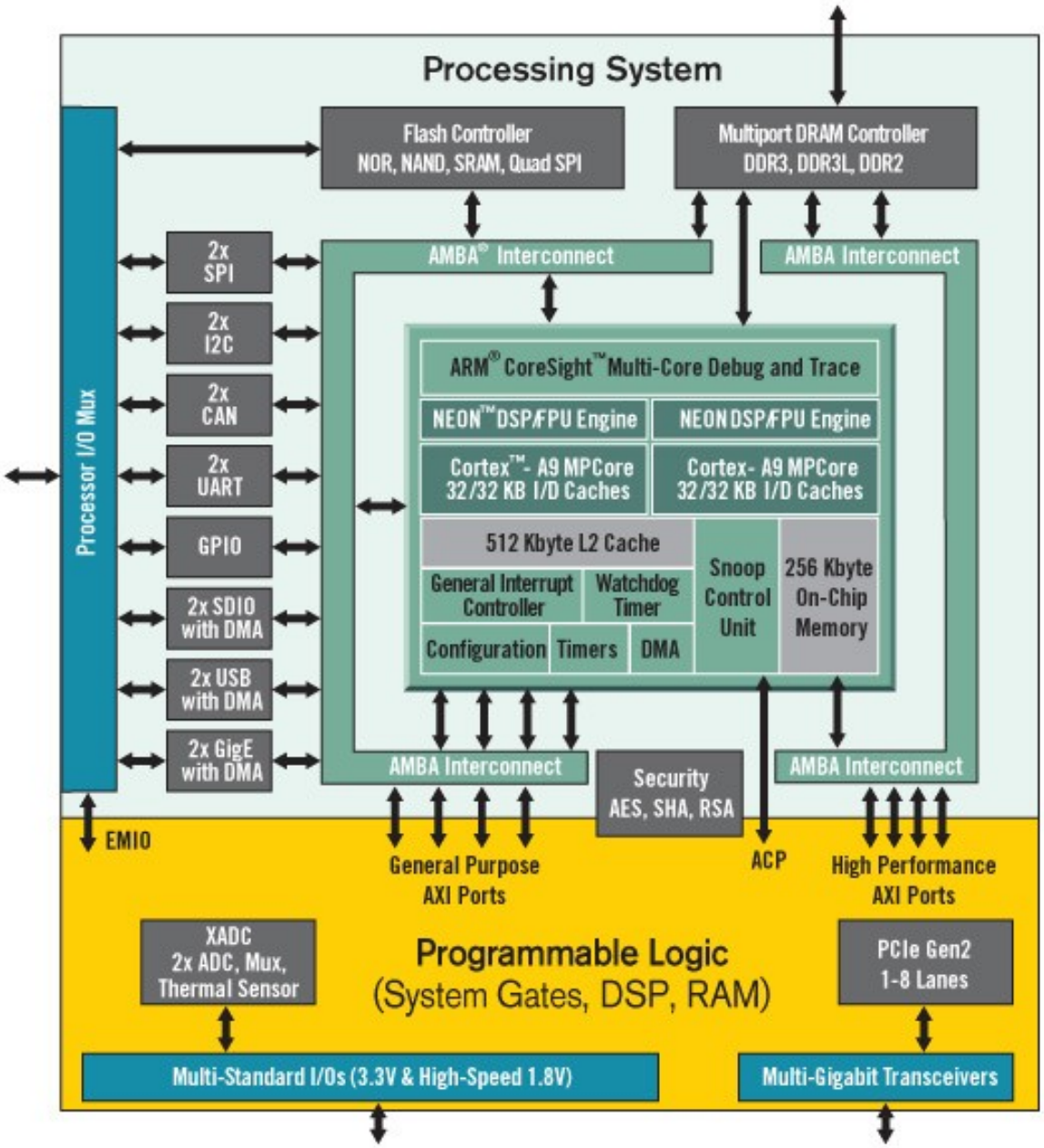
Pro nové SoC byl vyvinut Vivado Design Suite, který má být výrazným posunem v rychlosti návrhu, jednoduchosti použití a systémové integrace. Tento produkt podporuje zařízení série 7 (S7, V7, Zynq), C syntézu a má nové algoritmy pro syntézu, rozmisťování a routování.

Oba balíky se dodávají v různých variantách, které se liší dodávanými programy a tím i možnostmi. Samozřejmě i cena za licenci je rozdílná, přičemž lze získat zkušební licenci a základní balík (Web Pack) je dostupný zdarma.

Pro programování a ladění lze použít JTAG rozhraní, ale vzhledem k možným bootovacím konfiguracím (dále v 4.2.3) není nutné jej využít, protože lze bootovat z externích pamětí. Samozřejmě ale jeho využití poskytuje větší komfort při vývoji.

4.2.2 Programové možnosti

Protože Zynq obsahuje dvě ARM jádra a umožňuje měnit zapojení v/v pinů, lze jej nakonfigurovat buď jako AMP nebo SMP systém. To umožňuje, v režimu AMP, např. běh



Obrázek 4.3: Diagram Zynqu 7000 [28]

klasického operačního systému na jednom jádře a RTOS na druhém jádře s tím, že běžný OS bude obsluhovat většinu periferii (síť, audio a video, USB, ...) a na RTOS poběží kritické řídící aplikace. Vzájemná komunikace mezi těmito systémy může být zajištěna za pomoci sdílené paměti, nebo za pomoci zpráv. V režimu SMP se potom jedná o klasickou vícejádrovou architekturu, na které může běžet libovolný OS.

Další možností je tzv. „Bare-Metal" systém, kdy není použit žádný OS, ale na procesoru běží přímo (většinou jednoduchá) aplikace. Tím je možno získat dodatečné prostředky, které by jinak potřeboval pro svůj běh OS a případně snížit komplexnost celého systému.

Protože Zynq obsahuje také PL část, lze jej navíc volitelně rozšířit např. o další jádro a tím opět změnit konfiguraci a možnosti systémů. Ze zmíněných možných konfigurací je vidět, že Zynq si lze z velké části přizpůsobit a vyhovuje tak širokému spektru aplikací. Totéž platí i pro bootování, které má několik různých konfigurací.

4.2.3 Bootovací konfigurace

Zynq umožňuje jak zabezpečený, tak nezabezpečený boot. Ty se liší v možných konfiguracích a v chování systému.

V obou případech je jako první vykonán kód BootROM, který se nachází v paměti na čipu (OCM). Z bezpečnostních důvodů je jako první po restartu spuštěno jádro 0, které začne vykonávat BootROM a jádro 1 je drženo ve stavu čekání na událost (WFE). Také rozhraní JTAG je zakázáno. Úkolem tohoto kódu je nakonfigurovat systém a zkopírovat další kód do paměti na čipu. Tímto kódem může být buď „First Stage Boot Loader" (FSBL), nebo jiný inicializační kód. Volitelně může být další kód (po BootROM) spouštěn přímo z Quad-SPI nebo NOR zařízení, jedná-li se o nezabezpečený boot.

V případě zabezpečeného bootu kód BootROM zkopíruje FSBL/uživatelský kód do OCM a provede jeho rozšifrování a ověření na základě nastavené konfigurace. Nedojde-li k chybě, začne jej vykonávat. Naproti tomu u nezabezpečeného bootu kód BootROM vypne všechny zabezpečovací části (jako AES engine v PL), povolí JTAG rozhraní a poté spouští/kopíruje další kód podle nastavené konfigurace. Tu lze volit za pomoci pull-up/down rezistorů na příslušných pinech.

Dále spuštěný kód již má plnou kontrolu systému a další akce jsou již zcela na něm. Jedinou podmínkou je, že v případě nezabezpečeného bootu již nelze přejít zpět do zabezpečeného. Kód BootROM již dále není běžící aplikaci přístupný. Aplikační kód (ať už FSBL, nebo jakýkoliv jiný) již také může provést volitelnou konfiguraci PL části.

Jádro jedna se i dále nachází ve WFE stavu a pro jeho probuzení je potřeba provést další akce. Tou hlavní je změna vektoru přerušení pro událost, který se nachází na adrese 0xFFFFFFF0. Po restartu se zde z bezpečnostních důvodů nachází adresa instrukce WFE. Poté již lze jádro probudit za pomoci instrukce SEV, kterou provádí jádro 0. Tato událost je doručena všem jádrům a tím dojde k probuzení jádra 1.

Zdroje kódu pro boot

Je jich pět:

- NAND
- NOR
- SD karta

- Quad-SPI
- JTAG

Přičemž první čtyři jsou tzv. master módy a JTAG je tzv. slave mód, který je možný pouze v nezabezpečeném bootu a je na externím hostu, aby nahrál další kód do OCM. PS je v té době v idle režimu. Další omezení ohledně (ne)zabezpečeného režimu byly zmíněny výše.

Veškeré další detaily včetně např. času potřebného k vykonání BootROM, nebo detailu (de)šifrování apod. lze nalézt v [27].

4.2.4 Další vlastnosti

Mezi další vlastnosti procesu patří např. jednotka správy paměti (MMU), cache a s ní spojená Snoop Control Unit (SCU), TrusZone, Quality of Service (QoS) kontrolér, DMA kontrolér, čítače, watchdog a další kontroléry pro jednotlivé periferie. Podle konkrétní verze procesoru mohou být dostupné další rozhraní, jako např. PCI Express. Každý z nich je možné nakonfigurovat a používat.

Jednotlivé detaily zde nebudu popisovat, protože to není pro mou práci nezbytně nutné. Některé konkrétní detaily související s konfigurací RTOS budou zmíněny dále v kapitole 4.3. Veškeré možnosti nastavení lze dohledat v již zmíněné dokumentaci [27], jejichž 1 836 stránek vypovídá o komplexnosti této řady SoC.

4.3 µC/OS-II

µC/OS-II [17] je preemptivní real time operační systém, který má čistě prioritní plánovač. Byl vypuštěn do světa v roce 1999 a je stále aktivně vyvíjen. Od roku 2009 existuje také novější verze µC/OS-III. Její kořeny jsou v µC/OS-II, ale jedná se o zcela nový systém, který používá Round robin plánování, má méně limitací a podobné hardwarové nároky. Aktuálně, na rozdíl od µC/OS-II, ještě není dokončen proces certifikace pro některá použití. Jeho předchůdcem je µC/OS, původně nazýván µCOS, jehož kořeny sahají do roku 1992 [14].

Systém umožňuje až 256 (dříve pouze 64) úrovní priorit (priorita 0 je nejvyšší), přičemž minimálně dvě (resp. jedna, nejsou-li použity statistické funkce) jsou rezervovány pro vlastní operační systém. Protože každý proces musí mít jedinečnou prioritu, je jejich celkový počet limitován na 256. Uživateli je dostupných nejvyšších 254. µC/OS-II je dostupný ve formě zdrojových kódů pro velké množství různých procesorů, kontrolérů a DSP. Lze jej portovat na procesory od 8 b do 64 b.

Zdrojový kód µC/OS-II je z převážné většiny napsán v ANSI C s výjimkou souborů sloužících pro portování na použitou architekturu, které jsou napsány v Assembleru. Část, která je napsána v C je nezávislá na použité architektuře. Další dva soubory slouží ke konfiguraci systému a jeho vlastností za pomoci podmíněného překladu a konstant. Jeho paměťové nároky se pohybují mezi 5 kB a 24 kB podle zvolené konfigurace.

µC/OS-II podporuje paměťovou ochranu i management a každý proces má vlastní zásobník, jehož velikost lze volit pro každý proces zvlášť. Pro jednodušší ladění obsahuje µC/OS-II monitorovací nástroje, s pomocí kterých lze zjistit, jak velký zásobník daný proces potřebuje.

Na následujících řádcích jsou detailněji popsány některé detaily µC/OS-II. Detailní popis vlastností, konfigurace a návodu k portování lze nalézt v [15].

4.3.1 Inicializace systému

Aplikace využívající µC/OS-II musí dodržet daný postup při startu systému, aby byly korektně inicializovány všechny procesy a celý systém Tento proces se skládá z funkcí OSInit(), nastavení vektoru přerušení pro obsluhu přepínání kontextu, vytvoření procesů za pomoci OSTaskCreate(...)/OSTaskCreateExt(...) a startu vlastního systému za pomoci OSStart().

Funkce OSInit musí být volána jako první funkce systému. Ta vytvoří dva systémové procesy – proces „idle“, který běží, pokud není připraven žádný další proces a statistický proces, který počítá zatížení procesoru.

OSTaskCreate má čtyři povinné parametry. Prvním je ukazatel na adresu vytvářeného procesu, druhým je ukazatel na data pro vytvářený proces. Třetím parametrem je adresa vrcholu zásobníku pro daný proces a poslední je priorita procesu. Funkce OSTastCreateExt je rozšířenou verzí OSTaskCreate umožňující zadání velikosti zásobníku, kontrolu jeho konce, jeho inicializaci vymazáním, rozšíření TCB atd.

Funkce OSStart předá řízení µC/OS-II, který vybere proces s nejvyšší prioritu a spustí jej. Z této funkce nedojde nikdy k návratu.

4.3.2 Procesy

Procesy mohou být dvou typů – nekonečný (nekonečná smyčka) nebo jednorázový, který se po svém běhu smaže, případně dojde k jeho smazání jiným procesem. Každý proces se nachází v jednom ze čtyř, resp. pěti stavů:

- Spící – proces v paměti, který není dostupný µC/OS-II (tj. po OSTaskCreate, OSTaskCreateExt nebo OSTaskDel)
- Připravený – proces připraven k běhu čekající na naplánování
- Běžící – běžící proces
- Čekající – čekání na zprávu, semafor, tick atd.
- Přerušený (ISR) – nastane-li přerušení, je běžící proces přepnut do tohoto stavu

Kompletní graf možných přechodů je uveden na obrázku 4.4.

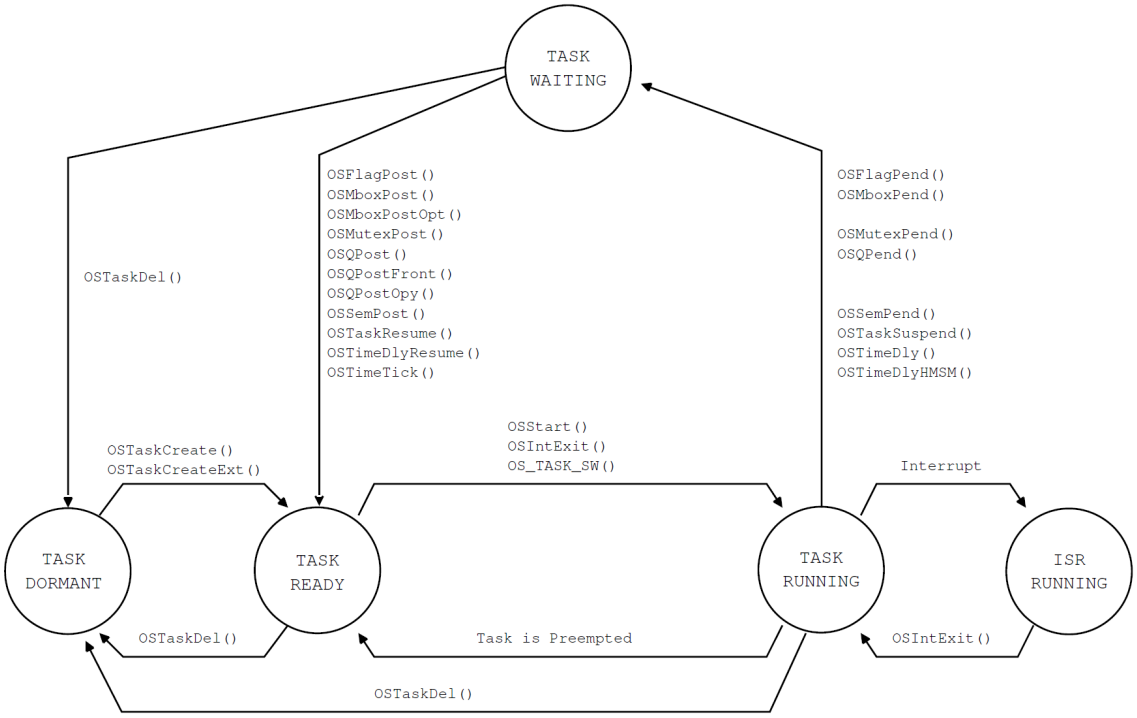
Jak již bylo zmíněno dříve, každý proces má jednu jedinečnou prioritu, která je volena při vytváření procesu. Ta se ale může dynamicky změnit, je-li tato možnost povolena v konfiguraci systému. To může nastat po zavolání funkce OSTaskChangePrio(...), nebo automaticky jako důsledek blokování připraveného procesu s vyšší prioritou čekajícího na semafor blokovaný procesem s nižší prioritou. V takovém případě dojde ke zdědění priority vyššího procesu procesem s nižší prioritou.

Datová struktura TCB

Data jednotlivých procesů jsou uloženy v TCB[2]. Ten v závislosti na konfiguraci obsahuje:

- Adresu vrcholu zásobníku procesu
- Prioritu
- Aktuální stav procesu

[2] Task Control Block



Obrázek 4.4: Graf přechodu mezi stavy procesů v µC/OS-II [15]

- Flag značící, je-li proces připraven k běhu
- Data pro kontrolu zásobníku (velikost, adresu konce)
- Zvolené vlastnosti zásobníku
- Uživatelský pointer (pro rozšíření TCB)
- Uživatelské id procesu
- Ukazatele obousměrně vázaného seznamu TCB procesů
- Proměnnou držící hodnotu zbývající doby požadovaného uspání procesu
- Ukazatel na kontrolní blok událostí
- Ukazatel na zprávu poslanou procesu
- Flag značící provedený požadavek na smazání procesu
- Další pomocné hodnoty

Pro rychlejší odezvu má µC/OS-II připraven seznam s volnými TCB (v počtu zvoleného maximálního množství procesů − počet již vytvořených), z kterého při vytváření nového procesu jeden vyjme a naplní požadovanými hodnotami. To zajišťuje deterministickou dobu vytváření nového procesu. Tentýž postup se používá u další proměnných v systému jako jsou události, semafory a podobně.

Přepínání kontextu

Přepnutí kontextu se skládá za tří hlavních kroků. Prvním je nalezení procesu, který má být spuštěn. Jedná-li se o aktuálně běžící proces, není třeba žádných dalších akcí. Není-li to aktuálně běžící proces, je třeba provést další dva kroky, kterými jsou uložení kontextu starého procesu a obnovení kontextu procesu, který má být spuštěn.

Nový proces ke spuštění je v případě µC/OS-II vybírán na základě priorit. Postup výběru se skládá z několika kroků. Pro rychlejší vyhledávání připraveného procesu s nejvyšší prioritou jsou všechny procesy rozděleny do skupin (nastiňuje obrázek 4.5). Každá skupina má jeden společný bit značící, je-li alespoň jeden proces ve skupině připraven k běhu. Dále existuje pole všech procesů, kde je uveden flag jejich připravenosti. Zjištění připraveného procesu poté probíhá nejprve zjištěním skupiny a poté konkrétního procesu za pomoci mapovacího pole, které na základě hodnoty X vrací hodnotu nejnižšího bitu v log. 1. Toto je provedeno nejprve pro zjištění skupiny, poté procesu uvnitř skupiny a následně je dopočítána priorita daného procesu. Konkrétní kód pro zjištění připraveného procesu je uveden v 4.2.

```
OSRdyGrp = OSMapTbl[prio >> 3];
OSRdyTbl = OSMapTbl[prio & 0x07];
```

Kód 4.1: Přidání procesu do seznamu připravených procesů

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

Kód 4.2: Nalezení připraveného procesu s nejvyšší prioritou

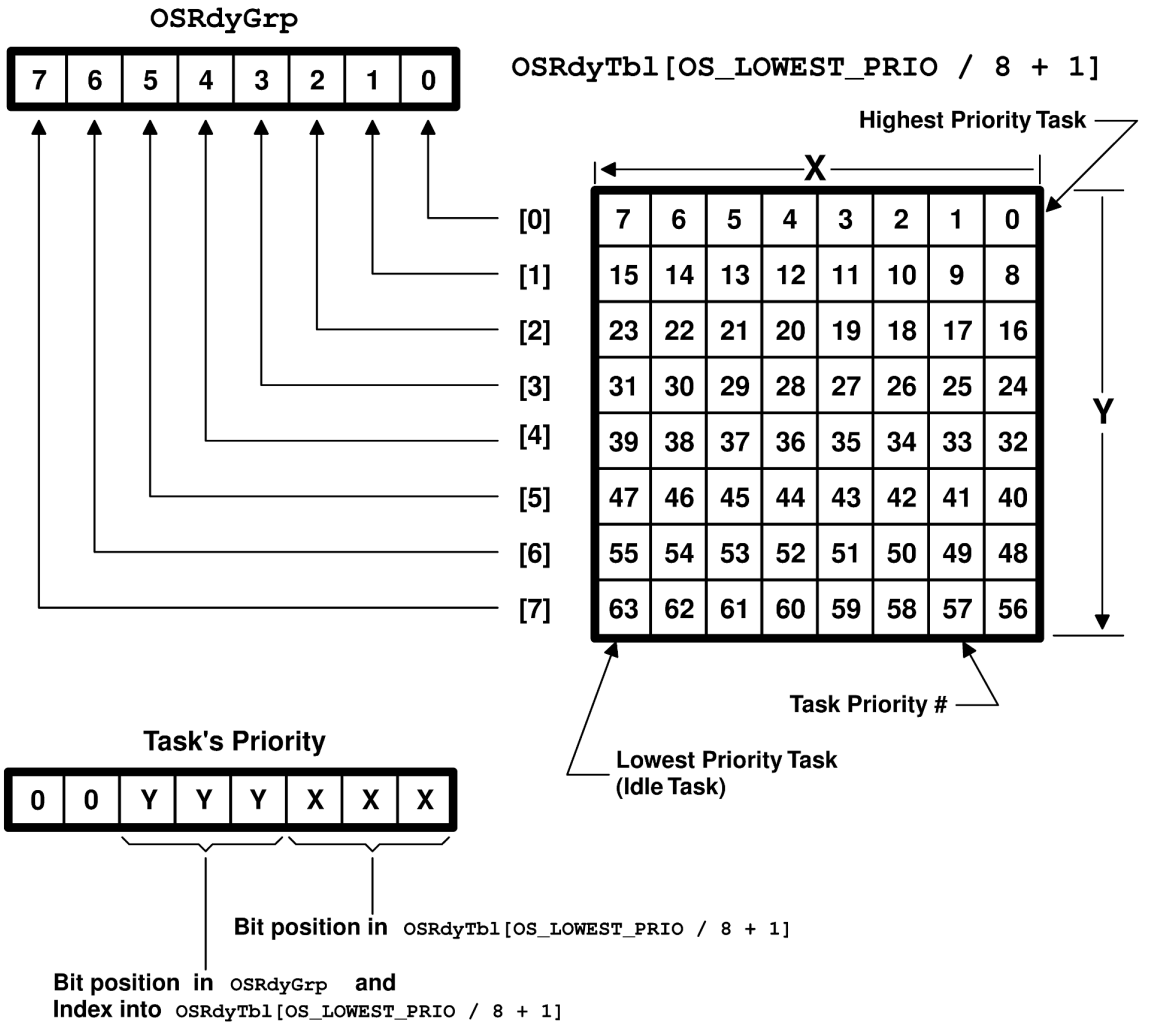
Má-li dojít k přepnutí kontextu, je třeba uložit kontext starého procesu. To je provedeno uložením hodnot stavových a datových registrů aktuálně běžícího procesu na zásobní a uložení vrcholu zásobníku do TCB. Tím je uložen kontext přerušovaného procesu.

K vlastnímu přepnutí kontextu je využíváno softwarové přerušení, případně jeho simulace. Každý proces má v TCB uložen vrchol svého zásobníku, který obsahuje jako poslední záznamy hodnoty stavových a datových registrů procesoru před jeho přerušením. Proto stačí obnovení vrcholu zásobníku, stavových a datových registrů a použití volání návratu z přerušení, které provede skok na potřebnou adresu.

Meziprocesorová komunikace

K meziprocesorové komunikace lze použít buď fronty zpráv, nebo poštovní schránky. Fronta zpráv může být buď FIFO, nebo LIFO a jedná se v podstatě o zřetězené poštovní schránky umožňující uchovat pouze jednu zprávu. Jak fronty zpráv, tak poštovní schránky poskytují blokující implementaci vkládání/získávání zpráv, přičemž lze volitelně specifikovat timeout a tím získat neblokující variantu. Při získávání zpráv dostává vždy přednost proces s nejvyšší prioritu i v případě, že se do fronty dostal jako poslední.

To je způsobeno prioritním plánováním. Je-li fronta prázdná, řadí se zde procesy, které jsou blokovány. Jakmile nějaký proces vloží zprávu do fronty, je k běhu připraveno více procesů, ale je vybrán ten s nejvyšší prioritou. Totéž nastává například v případě čekání na semafor, kdy je po uvolnění semaforu vybrán proces s nejvyšší prioritou a tomu je umožněn vstup do kritické sekce.



Obrázek 4.5: Struktura seznamu připravených procesů v µC/OS-II

Kritická sekce

Jako každý jiný operační systém i µC/OS-II potřebuje zabezpečit své proměnné proti vícenásobnému přístupu. K tomu používá makro OS_ENTER_CRITICAL() a k němu párové OS_EXIT_CRITICAL(). Toto makro je jednou ze základních prvků systému, které je třeba nakonfigurovat v rámci portování systému. µC/OS-II podporuje tři různé metody přístupu do kritické sekce (KS):

1. Zakázání/povolení přerušení
   Toto je nejjednodušší varianta, ale má velkou nevýhodu – po opuštění kritické sekce je přerušení vždy povoleno, nehledě na stav při vstupu do kritické sekce.

2. Uložení masky přerušení na zásobník a její následné obnovení
   Před vstupem do kritické sekce je uloženo aktuální nastavení přerušení na zásobní a přerušení je zakázáno. Při opouštění kritické sekce je nastavení přerušení obnoveno ze zásobníku. Tím je vyřešen problém první metody, ale je třeba dát pozor na situaci, kdy dochází ke změně zásobníku.

3. Uložení masky přerušení do lokální proměnné
   Postup je stejný jako u metody 2, ale stav se ukládá do lokální proměnné. Tím odpadá problém druhé metody, ale je třeba, aby jsme měli přístup k masce přerušení z C kódu.

Tento systém zamykání zajišťuje to, že služby jádra µC/OS-II poskytují reentrativní implementaci a proto se není třeba obávat vícenásobného volání těchto služeb. Toto už ale neplatí v případě vícejádrového systému, kdy i v případě zakázaného přerušení může docházet k přístupu k systémovým proměnným na více jádrech současně. V této situaci dochází k porušení kritické sekce a proto je třeba na těchto systémech použít jinou metodu zabezpečení kritických sekcí. Jednou z nich by mohlo být využití sdílené proměnné a atomické instrukce „test and set“.

Za pomocí tohoto jednoduchého zámku kritických sekcí jsou poté implementovány složitější mechanizmy zamykání. V případě µC/OS-II to jsou semafory a mutexy.

Synchronizace procesů

Pro synchronizaci procesů lze v µC/OS-II využít události, zprávy (ať už frontu, nebo schránku), semafory nebo mutexy. Pro všechny tyto typy událostí existuje společný kontrolní blok událostí (ECB[3]), do kterého jsou uloženy potřebné informace.

Stejně jako v případě TCB je každé události přidělen jeden ECB, které jsou navzájem propojené do listu. Také pro mapování a vyhledávání procesů čekajících na událost je použit stejný postup jako u TCB (strana 34).

[3]Event Control Block

Kapitola 5

Provedené změny

Pro využití µC/OS-II na kitu ZedBoard je třeba upravit operační systém, ale také správně nakonfigurovat samotný ZedBoard a boot proces.

U boot procesu je třeba zajistit správné načtení všech částí kódu na jejich místa v paměti a spuštění jádra na inicializačním kódu. U ZedBoaru je třeba správně nastavit propojení mezi jádry a periferiemi a případně také nastavit kontrolér přerušení. Důležité je také zajistit korektní práci sdílených prostředků, zejména pak L2 cache.

Úpravy samotného µC/OS-II již nejsou příliš složité, protože se jedná o relativně jednoduchý systém a velká část je provedena v rámci inicializačního kódu. Samotné jádro systému proto nevyžaduje tolik změn.

5.1 Systémová konfigurace

Pro konfiguraci celého systému je nejjednodušší v Xilinx Platform Studio vybrat správnou platformu a nechat si vygenerovat výchozí nastavení. V pozdějších verzích je ZedBoard již k dispozici a případné změny lze provádět úpravou výchozího nastavení. Tento postup je jednodušší než kompletní ruční vytváření mapování a také při něm nevzniká tolik chyb.

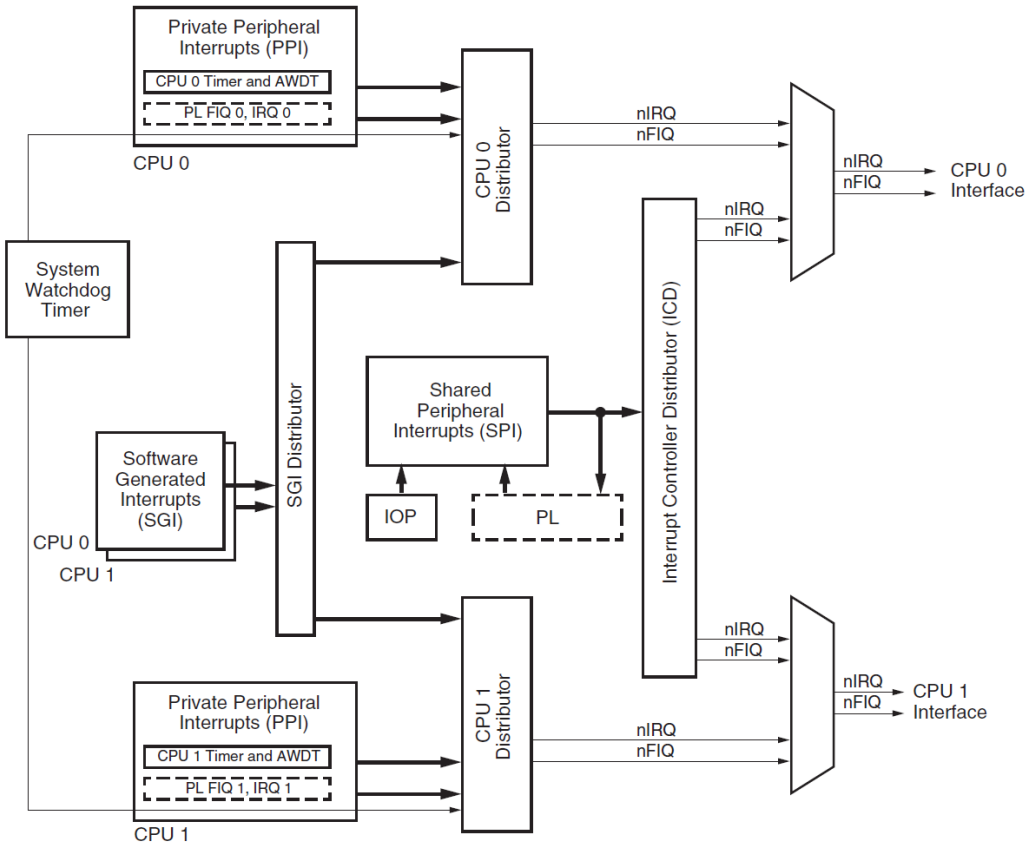
Další konfigurace je provedena v rámci pre-booot a boot procesu, jenž je proveden v rámci kódu v tzv. „Board Support Package“ (BSP). Tento kód lze získat přímo od Xilinxu a obsahuje také ovladače na nejnižší vrstvě pro periferie na čipu.

Důležitou částí konfigurace je také nastavení přerušovacího subsytému, který v případě Zynqu umožňuje několik různých nastavení. Přerušení jsou doručována za pomoci čtyř distributorů, kdy každý obsluhuje část systému. Vyžaduje-li to aplikace, lze volitelně v omezené míře za pomoci PL části „přepojovat“ přerušení mezi jednotlivými distributory. Diagram subsystému je zobrazen na obrázku 5.1. Na něm je vidět, že lze jednotlivé přerušení směřovat buď na jedno, nebo na obě jádra. Toho lze s výhodou využít pro různé potřeby AMP a SMP konfigurace.

5.2 Probuzení druhého jádra

Jak již bylo zmíněno v kapitole 4.2.3 proces probouzení druhého jádra by měl být jednoduchý a skládat se ze dvou kroků:

- Zapsání adresy s kódem pro druhé jádro na adresu 0xFFFFFFF0
- Vyvolání události (instrukce SEV)



Obrázek 5.1: Blokový diagram přerušovacího subsystému [27]

Toto ovšem nefunguje, pokud je nastaven JTAG mód bootování. Protože se ale jedná o triviální chybu, která neovlivní běžné aplikace (JTAG u nich není používán), je třeba se s ní vyrovnat pouze při vývoji. Jedna z variant je nastavení bootování z SD karty, která ale není přítomna. Systém poté přejde do záložního režimu, kdy dojde k zastavení systému a čekání na instrukce z JTAG rozhraní, není-li zakázáno (v případě zabezpečeného bootu). V tomto případě je druhé jádro správně zaparkováno a zmíněný problém již nevznikne. Zároveň docílíme stejného stavu systému, jako při použití JTAG boot módu.

Druhou variantou je za pomocí JTAG rozhraní zapsat přeložený kód obslužné rutiny na adresu 0xFFFFFF00, která zajistí stejnou funkcionalitu - tj. kontrola adresy uložené na 0xFFFFFFF0 na nenulovou hodnotu a případný skok na ni. V případě, že se adrese 0xFFFFFFF0 nachází nulová hodnota, přechází procesor zpět do stavu čekání na událost. Kód této rutiny je zobrazen v 5.1. Poté stačí nastavit registr PC jádra 1 na adresu 0xFFFFFF00, na které se tato rutina nachází a spustit jej. Skript pro debugger (xmd) je k nalezení na přiloženém CD, nebo na webu firmy Xilinx (soubor stub.tcl).

```
mvn     r0, #15
mov     r1, #0
str     r1, [r0]
wfe
ldr     r2, [r0]
cmp     r2, r1
beq     0xc
mov     pc, r2
```

Kód 5.1: Rutina držící jádro ve stavu čekání na instrukci

Chování této rutiny (v případě, že je prováděna v rámci BootROM) bylo upraveno v novějších revizích čipu, kde se pro změnu nečeká na nenulovou hodnotu, ale na hodnotu, která se nerovná hodnotě 0xFFFFFF2C.

Další problém může způsobit spouštění Linuxu, který na novější revizi čipu provádí úplný reset procesoru, což způsobí probuzení druhého jádra a jeho start na adrese 0x0. Je proto třeba jej odchytit a případně poslat zpět do stavu čekání na událost. Možný kód je zobrazen v 5.2. Na totéž je třeba pamatovat, pokud provádíme softwarový reset procesoru v rámci vlastní aplikace.

```
        @ get core number from coprocessor
        mrc     p15,0,r1,c0,c0,5
        and     r1, r1, #0xf
        cmp     r1, #0
        @ core 0 can continue
        beq     OKToRun
EndlessLoop0:
        @ code for core 1 - depends on the need
        @ could be waiting for sync with core 1
        @ and new address for branch
        wfe
        b       EndlessLoop0

OkToRun:
        @ code for core 0
```

Kód 5.2: Kód pro odchycení jádra 1

5.3 AMP konfigurace

V této konfiguraci lze spustit například dvě instance µC/OS-II, Linux a µC/OS-II nebo jinou kombinaci různých OS nebo nezávislých aplikací. Protože se jedná o nezávislý běh různých operačních systémů, jsou zásahy do nich minimální.

Nutné úpravy

První úprava se týká linkeru, u kterého je třeba správně nastavit adresu počátku kódu, protože každý OS běží v jiné části paměti. Zynq také umožňuje konfiguraci „viditelnosti“

paměti a proto ji lze přidělit jednotlivým jádrům v rámci konfigurace a tím zajistit její ochranu.

Dále je třeba specifikovat sdílenou paměť pro případnou komunikaci mezi systémy. Také je třeba dát pozor na to, který ze systémů bude řídit sdílenou cache L2 a s ní spojenou jednotku SCU, protože by se mohly navzájem ovlivňovat. Jenda z variant je zneplatňování pouze L1 cache v jednom ze systémů, zatímco druhý pracuje i s L2 cache a má kontrolu nad SCU. Druhá varianta je vytvoření komunikačního kanálu mezi systémy, kde při práci s L2 cache je jeden ze systémů master a druhý slave. Tyto změny lze provést např. v rámci BSP, je-li použita Xilinxem dodávaná verze.

Dále je potřeba pohlídat chování FSBL, který musí nahrát na správné místa v paměti dodaný kód. K tomu lze vytvořit boot image, který kromě FSBL, aplikační a jiných datových souborů obsahuje také konfigurační vektor pro PL část. FSBL poté provede nastavení PL dodaným vektorem, načte datové soubory a spustí vykonávání kódu na jádře 0. Tento kód poté musí v rámci inicializace provést nastavení adresy pro jádro 1 a jeho vlastní probuzení.

Aktuální verze FSBL poskytnutá Xilinxem v rámci vzorových řešení toto umožňuje. Stačí tedy připravit zbylé soubory a vygenerovat bootovací image.

Změny v samotném µC/OS-II nejsou v tomto případě v podstatě nutné. Jedině v případě přístupu ke sdíleným periferiím je třeba implementovat potřebné zámky za pomoci sdílené paměti, případně pouze předávat požadavky druhému systému k vyřízení. Další vhodnou změnou by mohla být úprava vstupu do kritické sekce, která se provádí za pomoci přerušení. V závislosti na zvolené konfiguraci přerušovacího subsystému je vhodné nahradit zakazování přerušení pro celý systém pouze zakázáním přerušení pro dané jádro, nejsou-li ostatní přerušení doručovány jádru, na kterém µC/OS-II běží.

Dopad na µC/OS-II

Z výše uvedených změn je zřejmě vidět, že dopad na samotný systém jsou minimální, případně žádné, není-li potřeba využívat sdílené periferie. Hrubý výpočetní výkon (dané instance systému) se nezměnil, protože stále běží na jednom jádře. Jedinou změnou je sdílení systémové sběrnice a paměti, kde může docházet ke zpomalení vlivem systému běžícího na druhém jádře.

5.4 SMP konfigurace

V tomto případě běží µC/OS-II na obou jádrech zároveň. Proto jsou již potřebné změny OS rozsáhlejší. Kromě kritické sekce je třeba také upravit plánovač a zajistit správnou obsluhu přerušení.

Nutné úpravy

Nejdůležitější úpravou je úprava kódu pro vstup do kritické sekce. Zde již nestačí jednoduché zakázání přerušení, ale je třeba použít sdílenou proměnnou jako zámek. Architektura ARM poskytuje dvě možnosti, jak zajistit atomickou operaci zápisu a tak implementovat potřebný zámek.

První je instrukce swap (SWP pro 32 b, SWPB pro 8 b). Ta zajistí atomickou záměnu hodnot v registru a paměti. Lze tedy implementovat klasický zámek „test and set“. Tato instrukce je ale v novějších verzích ARM architektur (ARMv6 a pozdější) považována za zastaralou a není doporučeno jí používat. Může totiž způsobit výrazné zpomalení systému,

protože blokuje přerušení a systémovou sběrnici. To má dopad zejména na systémech s pomalým přístupem do paměti.

Proto byly v architektuře ARMv6 (a pozdějších) přidány nové instrukce „Load/Store exclusive“ (LDREX/STREX), které zajišťují potřebnou funkcionalitu s minimálním dopadem na výkon. Při použití instrukce LDREX dojde k přečtení dat z požadované adresy spolu s jejím označením jako „exclusive“. Následné použití instrukce STREX zajistí zapsání na danou adresu pouze v případě, že je dané paměťové místo označeno jako „exclusive“. Pokud ne, je to signalizování v jednom ze tří registrů, které používá instrukce STREX. Pokud dojde k jinému přístupu k tomuto paměťovému místu, je tag „exclusive“ odstraněn a instrukce STREX selže. Tímto způsobem lze implementovat mj. jednoduchý zámek bez blokace sběrnice nebo přerušení.

Vstup do kritické sekce byl tedy rozšířen o zámek za pomoci sdílené proměnné a instrukcí LDREX a STREX. Kód použitého zámku je uveden v 5.3. U výstupu z kritické sekce stačí jednoduchá instrukce STR, která nastaví sdílenou proměnnou jako uvolněnou. Je-li tato operace provedena mezi instrukcemi LDREX a STREX, dojde také k zneplatnění tagu „exclusive“ a kód provádějící pokus o vstup jej bude opakovat. V mém případě k tomu ale nedojde, protože je vstup umožněn pouze jednomu procesu a před použitím instrukce se kontroluje, je-li zámek dostupný. Pokud ne, instrukce STREX se neprovádí. Stále ale může nastat současný pokus o přístup oběma jádry. V tom případě získá zámek to jádro, které první provede instrukci STREX. Ta zároveň zneplatní tag „exclusive“ a proto pokus druhého jádra selže. Protože druhé jádro v danou chvíli nemůže nic dalšího dělat, cyklicky čeká na uvolnění zámku. Druhou variantou čekání by bylo provedení instrukce pro uspání nebo pozastavení jádra a jeho následné probuzení (např. instrukce WFE + SEV) druhým jádrem po uvolnění zámku.

Při vstupu do kritické sekce se stále zakazuje přerušení, aby zbytečně nedocházelo k blokování procesu v kritické sekci a zbytečnému zápasu o zámek. Také se tím eliminuje možnost deadlocku v případě nevhodné blokace procesů s nižší prioritou.

Při úpravě plánování se nabízelo několik variant. Jednou z nich je fixní plánování priorit. To šlo zajistit pevným rozdělením priorit mezi jádra. Například spodní polovina by vždy běžela na jádře 0 a horní na jádře 1 (s výjimkou procesu idle, který by mohl běžet na obou). Při tomto rozdělení by bylo na programátoru, jak procesy rozdělí. Případně by je šlo později přerozdělit za pomoci dynamické změny priority.

Další variantou je dynamické plánování, kdy v jeho nejjednodušší variantě poběží na jednom jádře připravený proces s nejvyšší prioritou a na druhém jádře ten s druhou nejvyšší prioritou. Tato varianta nebere v potaz paměťovou lokalitu, což by mohlo být v určitých případech na škodu.

Plánovačů pro víceprocesorové prostředí ale existuje větší množství (např. [23]) a bylo by vhodné vybrat námi požadovaný pro danou aplikaci. Mnou zmíněné varianty se dle mého názoru nejvíce blíží původnímu prioritnímu plánování použitém v µC/OS-II.

Implementace obou je jednoduchá. U první stačí pracovat vždy pouze s půlkou všech procesů na každém jádře. Jako implementace s nejmenším dopadem se jeví zdvojení proměnných pro vyhledávání připraveného procesu a jejich důsledné rozlišování na základě priority procesu, se kterým pracujeme tak, aby byl proces vždy připraven pouze v rámci „té správné“ skupiny proměnných. Poté stačí při plánování procházet pouze skupinu patřící jádru, pro které se plánování provádí.

U druhé varianty si lze implementaci ulehčit tak, že proces aktuálně běžící na druhém jádru dočasně odstraníme z fronty připravených požadavků a vyhledáme další připravený. Tím dostaneme dva s nejvyšší prioritou. Poté vrátíme dočasně odstraněný proces zpět

(protože stále běží) a spustíme nově vybraný na aktuálním jádře. Je ovšem třeba ošetřit případ, kdy není žádný další proces připraven a má být spuštěn proces idle na obou jádrech.

Ani jedna ze zmíněných variant ale není příliš dobře škálovatelná a proto by bylo vhodné celé plánování přepracovat. Tato změna by již byla rozsáhlejší a s největší pravděpodobností by došlo k narušení konstantní doby změny připravenosti procesu a plánování.

Co se týče přerušení, lze jej nechat vykonat buď na libovolném jádře, nebo je omezit pouze na jedno jádro s tím, že přerušení vytvořené speciálně pro jádro X budou vždy obsloužena na tomto jádře. V případě vykonávání na obou jádrech by bylo vhodné pouvažovat, není-li potřeba dodatečná ochrana některých sdílených zdrojů.

Dopad na µC/OS-II

V tomto případě jsou změny rozsáhlejší a mohly by mít větší vliv. Zejména může nastávat blokace jednoho jádra druhým, na což by měly pamatovat použité aplikace. Celková reakční doba systému by ale neměla vzrůst nad dvojnásobnou dobu, protože i dva současně příchozí požadavky na vstup do kritické sekce by měly být vyřízeny sekvenčně.

Je ale třeba počítat s tím, že skoro každé systémové volání obsahuje vstup do kritické sekce. Pokud by běžící aplikace intenzivně používaly služby jádra, mohla by se objevit vyšší latence, způsobená „načítáním“ zpoždění. V tom případě by bylo vhodné zjemnit zámek pro různé systémové zdroje a tím omezit možnost konfliktu.

```
        @ lock variable - OSJMACSSpnLck
        @ lock unlocked def - OS_JMA_CSLCK_UNLOCKED
        @ lock locked def - OS_JMA_CSLCK_LOCKED
        @ store ret addr
        STMDB           SP!, {LR}
        @ load lck addr
        LDR                     R1, =OSJMACSSpnLck
isLocked:
        @ exclusive read
        LDREXB          R2, [R1]
        @ unlocked?
        CMP                     R2, #OS_JMA_CSLCK_UNLOCKED
        @ if not equal, it is locked
        BLNE isLocked
        @ otherwise exclusive str -> get lock
        STREXBEQ        r2, r0, [r1]
        @ check if success (0 == success)
        CMP                     r2, #0
        BLNE            isLocked
        @ we have aquired lock
        @ data memory barier
        DMB
        @ ret addr restore
        LDM                     SP!, {LR}
        BX                      LR
```

```
        @ load lck addr
        LDR             R1, =OSJMACSSpnLck
        MOV             R0, #OS_JMA_CSLCK_UNLOCKED
        @ data memory barier
        DMB
        @ Unlock
        STRB    R0, [R1]
        BX              LR
```

Kód 5.3: Přidaný kód ke vstupu/výstupu do/z kritické sekce

Kapitola 6

Závěr

Práce byla zaměřená na úpravu operačního systému pro vícejádrovou architekturu. Jako použitý systém byl vybrán µC/OS-II běžící na vývojovém kitu ZedBoard. Tato problematika je rozsáhlá a konkrétní řešení výrazně závisí na použité architektuře. Proto je práce v některých částech spíše obecná.

V rámci úvodních kapitol jsem diskutoval obecné důvody, které vedou k intenzivnímu využívání vícejádrových a víceprocesorových architektur. Dále pak obecné problémy spojené s přechodem k těmto paralelním systémů, a to jak z pohledu hardwaru, tak zejména softwaru.

Na vybraném operačním systému jsem provedl základní změny nutné pro jeho provoz na vícejádrové architektuře, konkrétně dvoujádrovém procesoru ARM. V rámci úprav jsem se věnoval dvou hlavním konfiguracím - AMP a SMP. U SMP konfigurace se nabízí větší množství možných úprav v závislosti na požadovaných vlastnostech. Také se jedná se rozsáhlejší zásahy do systém, než v případě AMP konfigurace.

Jako vhodné pokračování se jeví zjemnění použitého zámku u SMP konfigurace a tím potencionální snížení reakční doby při vyšším zatížení systému. Protože se jednalo o real-time operační systém, bylo by vhodné ověřit, zda nepřišel o své důležité vlastnosti, zejména pak determinismus. Dále by bylo vhodné navrhnout experiment, který by umožnil změřit možné zrychlení pro typickou sadu aplikací, nebo nalézt vhodný benchmark. Spolu s touto informací by bylo zajímavé zjistit, zda-li došlo k měřitelnému zvýšení požadovaného výpočetního výkonu systémem.

Z výše zmíněného je zřejmé, že bych mohl práci výrazně rozšířit a stále se držet tématu i zadání. To se bohužel zejména z časových důvodů nepovedlo. I tak se ale jednalo o práci velmi inspirativní, která může sloužit jako jednoduchý návod k procesu přechodu k paralelním architekturám.

Literatura

[1] ARM Holdings: ARM Architecture Reference Manual. http://infocenter.arm.com/help/index.jsp, 2014, online.

[2] ARM Holdings: ARM Company milestones. http://www.arm.com/about/company-profile/milestones.php, 2014, online.

[3] ARM Holdings: big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php, 2014, online.

[4] ARM Holdings: Cortex-A9 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php, 2014, online.

[5] Avnet: ZedBoard PCB and Decoupling Design Rev. C.1. http://www.zedboard.org/sites/default/files/documentations/Zedboard%20Decoupling%20121001.pdf, 01.10.2012, online.

[6] Avnet: ZedBoard. http://www.zedboard.org/product/zedboard, 11.02.2014, online.

[7] Avnet: ZedBoard Hardware User's guide. http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf, 27.01.2014, online.

[8] Compaq; Phoenix; Intel: BIOS Boot Specification. http://www.phoenix.com/resources/specs-bbs101.pdf, 11.01.1996, online.

[9] Fuller, S. H.; Lynette I. Millett, E. C. N.: *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011, ISBN 9780309159517, 185 s.
URL http://www.nap.edu/openbook.php?record_id=12980

[10] Hennessy, J. L.; Patterson, D. A.: *Computer Architecture, Fifth Edition: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2011, ISBN 978-0123838728, 856 s.

[11] Intel Corporation: Intel Hyper-Threading Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, Naposledy navštíveno 01.05.2014, online.

[12] Jayachandran, P.; Abdelzaher, T.: The Case for Non-Preemptive Scheduling in Distributed Real-Time Systems. https://www.ideals.illinois.edu/bitstream/handle/2142/11332/The%20Case%20for%20Non-Preemptive%20Scheduling%20in%20Distributed%20Real-Time%20Systems.pdf?sequence=2, 2007, online.

[13] Kolektiv autorů: The Tanenbaum-Torvalds Debate. http://oreilly.com/catalog/opensources/book/appa.html, 1992, online.

[14] Labrosse, J. J.: *µC/OS the Real-Time Kernel*. Cmp Books, 1992, ISBN 978-0879304447, 266 s.

[15] Labrosse, J. J.: *MicroC/OS-II: The Real-Time Kernel*. Lawrence, Kan: CMP Books, druhé vydání, 6 2002, ISBN 978-1578201037, 648 s.

[16] Lee, R.: SERVER MEMORY: Understanding Memory Fragmentation in NetWare Servers. http://support.novell.com/techcenter/articles/ana19950407.html, 01.04.1995, online.

[17] Micrium: µC/OS-II. http://micrium.com/rtos/ucosii/overview/, 2014, online.

[18] Molnar, I.: CFS Scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, 23.09.2013, online.

[19] Peter, S.: Resource Management in a Multicore Operating System. http://e-collection.library.ethz.ch/eserv/eth:6270/eth-6270-02.pdf, 2012, online.

[20] Phoenix: PhoenixBIOS 4.0 User's manual. http://www.esapcsolutions.com/ecom/drawings/PhoenixBIOS4_rev6UserMan.pdf, 22.06.2000, online.

[21] Roch, B.: Monolithic kernel vs. Mircokernel. http://www.davepowell.org/media/teaching/operating-systems/handouts/COMP354-Lec02-KernelComparisons.pdf, 2004, online.

[22] Šimůnek, J.: *Úvod do problematiky osobních počítačů IBM PC: MS DOS*. Klub 602, 1991, ISBN 978-8070620724, 218 s.

[23] Strnadel, J.: Plánování úloh v systémech RT - IV: víceprocesorové prostředí. *Automa*, ročník 19, č. 1, 2013: s. 44–46, ISSN 1210-9592.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9974

[24] The Open Group: POSIX bacgrounder. http://www.opengroup.org/austin/papers/backgrounder.html, Naposledy navštíveno 16.04.2014, online.

[25] Torvalds, L.: Linux 3.0-rc1. https://lkml.org/lkml/2011/5/29/204, 29.05.2011, online.

[26] UEFI Forum: Unified Extensible Firmware Interface Forum. http://www.uefi.org/, 2014, online.

[27] Xilinx: Zynq-7000 All Programmable SoC Technical Reference Manual. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 11.02.2014, online.

[28] Xilinx: Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/, 2014, online.