České Vysoké Učení Technické v Praze
Fakulta Elektrotechnická
Katedra Řídicí Techniky

DIPLOMOVÁ PRÁCE

Operační systém QNX

Jaroslav Pajer květen 2003

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 21. května 2003. Jaroslav Pajer

Poděkování

Tato práce by nevznikla bez pomoci, cenných rad, připomínek a podnětů vedoucího diplomové práce Ing. Zdeňka Šebka a také Ing. Pavla Píši.

Můj vděk patří taky všem, kteří mě při práci podporovali i jinak než odbornými radami.

Abstrakt

Tato diplomová práce se zabývá realtimovým operačním systémem QNX. Měla by sloužit jako seznamovací materiál pro další uživatele operačního systému. Práce je zaměřena na strukturu operačního systému, implementaci mikrojádra a především na služby poskytované mikrojádrem. Práce obsahuje několik ukázkových programů psaných v jazyce C, které demonstrují vlastnosti operačního systému. Dále obsahuje aplikaci, která vizualizuje a řídí otáčky stejnosměrného motorku pomocí PWM. Tato aplikace byla vytvořena v objektově orientovaném prostředí Photon Application Builderu. V práci je také diskutována vhodnost použití operačního systému QNX pro řízení v reálném čase. Operační systém QNX je vybaven grafickým rozhraním Photon microGUI, které má jen minimální nároky na paměť a je plně konfigurovatelné.

Abstract

This project deal with realtime operating system named QNX. The project should serve as instrumental material for the next users of operating system. The project is devoted to structure of operating system, implementation of microkernel and especially services provided by microkernel. This project contains a few sample programs implemented in C that illustrate characteristics of operating system. This project further includes application which visualizes and controls a rotation speed of direct-current motor by force of PWM. The application was created in object oriented environment of Photon Application Builder. In the project is also discussed propriety of using operating system QNX for realtime control. Operating system QNX provide graphical interface called Photon microGUI which has only minimal claims on memory and it is fully configurable.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Tato diplomová práce se zabývá realtimovým operačním systémem QNX. Základem celého operačního systému je mikrojádro zvané Neutrino. Díky QNX Neutrinu je systém velmi spolehlivý, dostupný a deterministický. Mikrojádro obsahuje jen málo služeb. Mezi ně patří například vlákna, zprávy, semafory, signály a plánování. Neutrino je dynamicky rozšiřitelné o další služby, které jsou zprostředkovány speciálními volitelnými procesy (např. systém souborů, síť, POSIXové zprávy front a ovladače zařízení).

QNX podporuje následující systémy souborů: Image, RAM, QNX4, DOS, CD ROM, Flash, NFS (Network File System), CIFS (Common Internet File System), Linux Ext2 a Virtual filesystem. Dobrá je také podpora sítě. Integrovány jsou protokoly TCP/IP, PPP, DHCP, ICNP, UDP, nativní komunikační protokol Qnet, atd. Operační systém je vybaven grafickým rozhraním Photon microGUI, které má jen minimální nároky na paměť a je plně konfigurovatelné. Na ploše se nacházejí dva panely. Jeden se nachází ve spodní části obrazovky a má stejnou funkci jako panel z Windows (zobrazuje spuštěné aplikace), druhý se nachází v pravé části obrazovky a obsahuje zástupce pro systémové programy, utility a konfigurační nástroje.

Druhá kapitola se zabývá filozofií operačního systému. Stručně popisuje návrhářský záměr, vhodnost QNX pro embedded systémy pracující v reálném čase, architekturu mikrojádra, architekturu sítě a meziprocesní komunikaci. Třetí kapitola se věnuje implementaci Neutrina a službám Neutrina, jako jsou vlákna, signály, časovače, hodiny, obsluhy přerušení, atd. Čtvrtá kapitola obsahuje několik ukázkových prográmků psaných v jazyce C, které demonstrují použití POSIXových frontových zpráv a služeb Neutina, jako jsou bariéry, semafory a mutexy. V Páté kapitole se zabývám vizualizací a řízením otáček stejnosměrhého laboratorního motorku pomocí PWM. Tato aplikace byla vytvořena v objektově orientovaném prostředí Photon Application Builderu. K diplomové práci je přiložen CD ROM obsahující vlastní diplomovou práci ve formátu doc a pdf, dokumenty a zdrojové kódy.

2 Filozofie QNX

2.1 Návrhářský záměr

Hlavním cílem QNX je dodat otevřený systém s POSIX API v robustní a odstupňované formě, vhodné pro široký rozsah systémů (od malých systémů až po high-end systémy s distribuovaným výpočetním výkonem).

Robustní architektura je nezbytná kvůli aplikacím pracujících v kritických podmínkách, OS pružně a zcela využívá MMU (Management Memory Unit).

2.2 POSIXový OS, QNX a embedded systémy

POSIXový operační systém je příliš velký a proto nevhodný pro embedded systémy.

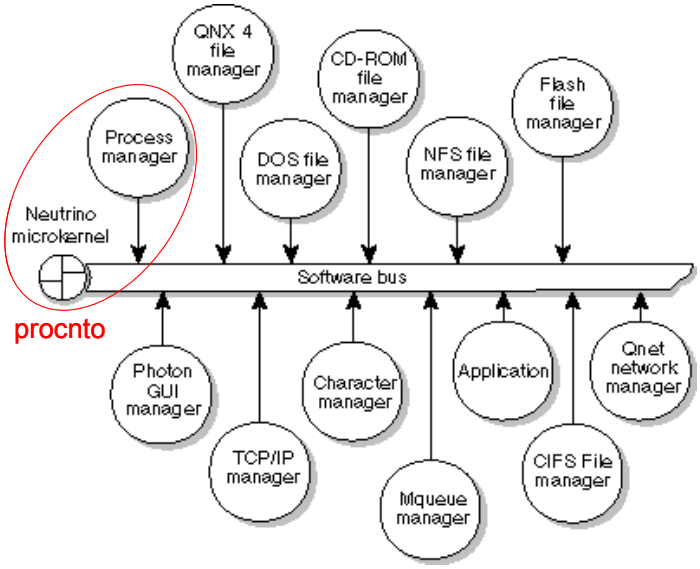
QNX nemá UNIXovou architekturu a díky architektuře jádra dodává standardní POSIX API ve zmenšené formě pro realtimeové embedded systémy a dle požadavků je lze postupně rozšiřovat. QNX je ideální OS pro embedded realtimeové aplikace. I v takto malé formě poskytuje základní služby jako multitasking, vlákna, prioritně řízené preemptivní rozvrhování a rychlé přepínání kontextu.

QNX dosahuje jedinečného stupně účinnosti, modularity a jednoduchosti díky dvěma základním principům:

- architektuře mikrojádra
- meziprocesní komunikaci založené na zprávách

2.3 Architektura mikrojádra

Mikrojádro je strukturováno jako malé jádro, které poskytuje minimální služby užívané skupinou zvláštních spolupracujících procesů. Tyto procesy zajišťují vyšší stupeň funkčnosti operačního systému. Mikrojádro postrádá systémy souborů a mnoho dalších služeb, které jsou poskytovány pomocí speciálních procesů.



Obrázek 2.1: Architektura mikrojádra

Skutečným cílem není vytvořit pouze malé jádro, ale vytvořit modulární operační systém. Velikost jádra je pak pouze vedlejším účinkem tohoto přístupu. Mikrojádro poskytuje služby meziprocesní komunikace, které slouží jako pojítko operačního systému. Výkon a flexibilita těchto služeb určuje výkon výsledného OS.

Jádro OS QNX zajišťuje kompletní ochranu paměti, ne pouze pro uživatelské aplikace, ale také pro komponenty OS (ovladače zařízení, systémy souborů, atd.).

Neutrino na rozdíl od vláken není rozvrhováno. Procesor vykoná kód jádra pouze v případě explicitního volání jádra nebo v odpovědi na hardwarové přerušení.

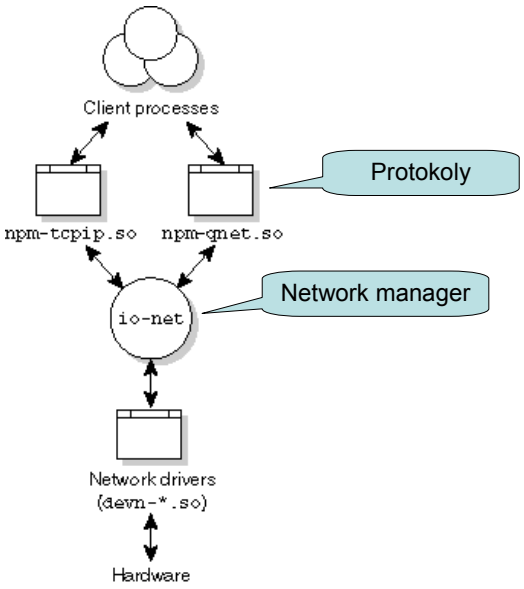
Všechny služby OS QNX kromě služeb zajišťovaných modulem *procnto* jsou zpracovávány pomocí standardních procesů, proto je velmi snadné rozšířit OS o další služby. Systémové procesy jsou v podstatě nerozeznatelné od některých uživatelských procesů, protože uživatelské procesy používají stejný API a vhodně privilegované uživatelské procesy mají k dispozici i stejné služby jádra.

2.4 Síťová distribuce mikrojádra

LAN (Local Area Network) poskytuje ve své nejjednodušší formě mechanismus pro sdílení souborů a periferních zařízení mezi několik vzájemně propojených počítačů. QNX jde daleko za tento jednoduchý koncept a zahrnuje celou síť do jediného, homogenního souboru prostředků.

Jakékoliv vlákno běžící na počítači připojeném v síti může přímo využívat prostředky jiného počítače. Z aplikačního hlediska nejsou žádné rozdíly mezi lokálními a vzdálenými prostředky. K používání vzdálených prostředků se nemusí do aplikace vkládat žádné speciální příslušenství.

Uživatel může spouštět aplikace na libovolném počítači v síti, pokud vlastní patřičné oprávnění.



Obrázek 2.2: Architektura sítě

2.5 Meziprocesní komunikace

Meziprocesní komunikace umožňuje navrhovat aplikace jako sadu spolupracujících procesů, kde každý proces obsluhuje jednu přesně stanovenou část z celku. QNX poskytuje jednoduché, ale výkonné schopnosti meziprocesní komunikace, které značně zjednoduší práci na vyvíjených aplikacích.

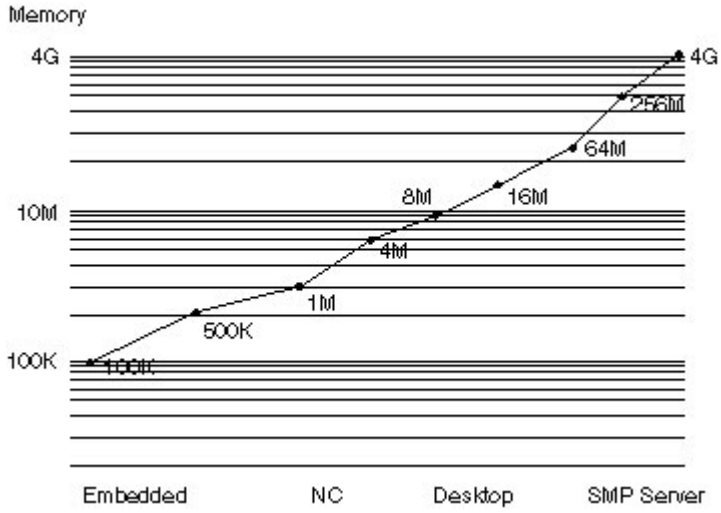
QNX byl první komerční OS svého druhu, který používá message passing (zasílání zpráv) jako základní prostředek meziprocesní komunikace. Díky kompletní integraci zasílání zpráv do celého systému je OS QNX tak jednoduchý a výkonný.

Message passing neslouží pouze k posílání dat mezi procesy, ale také k synchronizaci procesů. Operační systém nepřidává žádná zvláštní data do obsahu posílaných zpráv. Data ve zprávách mají význam pouze pro odesílatele a příjemce.

3 Popis mikrojádra Neutrino

3.1 Implementace Neutrina

Od počátku je na QNX vyvíjen tlak z obou konců výpočetního spektra, jednak od vestavěných systémů (embedded systems) s omezenou pamětí a jednak od špičkových SMP (Symmetrical Multi-Processing) počítačů s gigabajty operační paměti. Neutrino vyhovuje těmto oběma zdánlivě neslučitelným požadavkům a je možné jeho další rozšíření do jiných implementací operačního systému.



Obrázek 3.1: Oblasti použití OS QNX a jeho nároky na operační paměť

Někteří vývojáři se domnívají, že je Neutrino pro svou velikosti a výkonu implementováno pouze v assembleru. Ve skutečnosti je převážně implementováno v jazyce C. Malé velikosti a vysokého výkonu lze dosáhnout spíše pomocí rafinovaných algoritmů a datových struktur, než pomocí optimalizačních metod.

V Neutrinu jsou implementovány rysy POSIXového jádra, které jsou užívány v systémech pracujících v reálném čase, spolu s nezbytnými QNX službami posílání zpráv. POSIXové rysy, které nejsou implementovány v mikrojádru (např. soubor a zařízení I/O), jsou poskytovány nepovinnými procesy a sdílenými knihovnami.

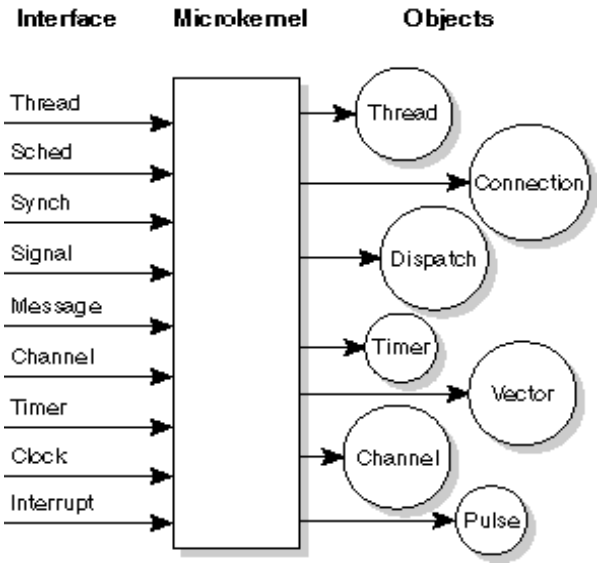
Postupně došlo k redukci kódu mikrojádra, který implementuje potřebné volání jádra. Díky objektům definovaných v nejnižší vrstvě jádra se kód stal více specifický, dovolující opětovné použití kódu (např. sdružování různých druhů POSIXových signálů, realtimeových signálů, a QNX pulsů do společných datových struktur a kódů, které manipulují s těmito strukturami).

POSIXové realtimové a vláknové rozšíření

Neutrino implementuje většinu realtimeových a vláknových služeb přímo v mikrojádru, proto jsou tyto služby dostupné i bez přítomnosti přídavných modulů OS.

Některé profily definované POSIXem naznačují, že tyto služby budou přítomny bez nutného požadování procesního modelu. Neutrino poskytuje přímou podporu vláken a process manager rozšiřuje tuto funkčnost na procesy obsahující více vláken.

Na Obrázku 3.2 jsou vidět základní objekty definované v nejnižší vrstvě Neutrina a rutiny, které s nimi manipulují.



Obrázek 3.2: Mikrojádro Neutrino

3.2 Služby Neutrina

- threads
- message passing
- signals
- clocks
- timers
- interrupt handlers
- semaphores
- mutual exclusion locks (mutexes)
- condition variables (condvars)
- barriers.

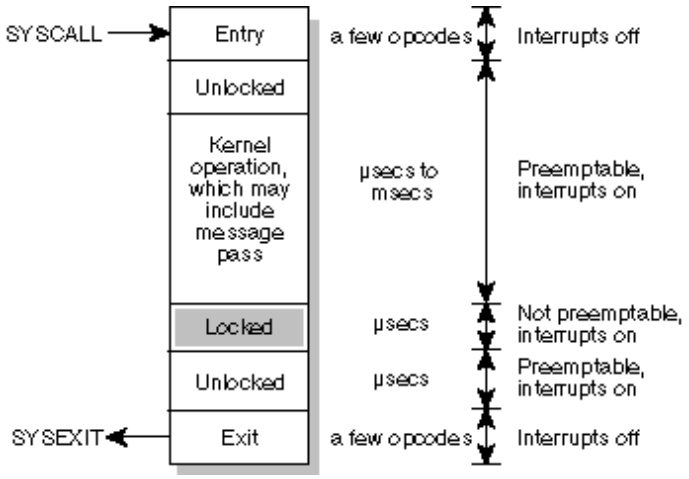
Neutrino je zcela preemptivní, lze přerušit i posílání zpráv mezi procesy. Posílání zprávy bude pokračovat v místě kde došlo k přerušení.

Minimální složitost Neutrina napomáhá k nalezení nejdelší nepreemptivní kódové cesty skrz jádro, ačkoli krátký kód může představovat složitý trasovací problém. Služby určené k implementaci byly vybírány na základě krátké doby výpočtu. Operace vyžadující důležité úkony (např. zavádění procesů) byly přiděleny externím procesům či vláknům, kde úsilí vstoupit do kontextu tohoto vlákna je bezvýznamné ve srovnání s prací prováděnou uvnitř vlákna při obsluhování požadavku.

Rigorózní použití tohoto pravidla, které dělí funkčnost mezi jádro a externí procesy popírá, že mikrojádro OS QNX musí mít vyšší režijní čas než monolitické jádro. Velmi

rychlé přepnutí kontextu je zanedbatelné vůči obsluze žádosti zprostředkované mezi procesy pomocí message passing.

Obrázek 3.3 ukazuje detaily preempce (nuceného přerušení) pro non-SMP jádro (x86 implementace). SMP verze jádra má více operačního kódu (část instrukcí strojového jazyka, která určuje, jaký druh činností počítač přímo provede).



Obrázek 3.3: Podrobný popis preempce

Přerušení je zakázáno pouze po velmi krátkou dobu (typicky stovky nanosekund).

3.2.1 Vlákna a procesy

Vlákna umožňují současné provádění několika algoritmů v aplikaci. Neutrino používá POSIXový vláknový model, který definuje proces obsahující jedno nebo více vláken.

Vlákno si můžeme představit jako minimální „jednotku“ plánovanou a vykonávanou v jádře. Proces může být chápán jako „úložiště“ vláken, definující adresní prostor ve kterém budou vlákna vykonávána. Proces vždy obsahuje nejméně jedno vlákno.

Následující *pthreads* (POSIXová vlákna) knihovní funkce nejsou zahrnuty v žádných funkcích jádra:

- *pthread_attr_destroy()*
- *pthread_attr_getdetachstate()*
- *pthread_attr_getinheritsched()*
- *pthread_attr_getschedparam()*
- *pthread_attr_getschedpolicy()*
- *pthread_attr_getscope()*
- *pthread_attr_getstackaddr()*
- *pthread_attr_getstacksize()*
- *pthread_attr_init()*
- *pthread_attr_setdetachstate()*
- *pthread_attr_setinheritsched()*
- *pthread_attr_setschedparam()*
- *pthread_attr_setschedpolicy()*
- *pthread_attr_setscope()*

- *pthread_attr_setstackaddr()*
- *pthread_attr_setstacksize()*
- *pthread_cleanup_pop()*
- *pthread_cleanup_push()*
- *pthread_equal()*
- *pthread_getspecific()*
- *pthread_setspecific()*
- *pthread_testcancel()*
- *pthread_key_create()*
- *pthread_key_delete()*
- *pthread_once()*
- *pthread_self()*
- *pthread_setcancelstate()*
- *pthread_setcanceltype()*

Tabulka 3.1: Přehled volání POSIXových vláken, která jsou implementována v Neutrinu

POSIX call	Microkernel call	Description
pthread_create()	*ThreadCreate()*	Vytvoří nové vlákno.
pthread_exit()	*ThreadDestroy()*	Zruší vlákno.
pthread_detach()	*ThreadDetach()*	Odpojí vlákno (nemusí být připojeno).
pthread_join()	*ThreadJoin()*	Připojí vlákno čekající na exit status.
pthread_cancel()	*ThreadCancel()*	Zruší vlákno v příštím místě zrušení.
N/A	*ThreadCtl()*	Změní typické chování vlákna.
pthread_mutex_init()	*SyncTypeCreate()*	Vytvoří mutex.
pthread_mutex_destroy()	*SyncDestroy()*	Zruší mutex.
pthread_mutex_lock()	*SyncMutexLock()*	Zamkne mutex.
pthread_mutex_trylock()	*SyncMutexLock()*	Podmíněně zamkne mutex.
pthread_mutex_unlock()	*SyncMutexUnlock()*	Odemkne mutex.
pthread_cond_init()	*SyncTypeCreate()*	Vytvoří condition variable.
pthread_cond_destroy()	*SyncDestroy()*	Zruší condition variable.
pthread_cond_wait()	*SyncCondvarWait()*	Čeká na condition variable.
pthread_cond_signal()	*SyncCondvarSignal()*	Pošle signální condition variable.
pthread_cond_broadcast()	*SyncCondvarSignal()*	Pošle broadcastovou cond. variable.
pthread_getschedparam()	*SchedGet()*	Získává parametry a druh plánování vlákna.
pthread_setschedparam()	*SchedSet()*	Nastavuje parametry a druh plánování vlákna.
pthread_sigmask()	*SignalProcMask()*	Zkontroluje nebo nastavuje signální masku vlákna.
pthread_kill()	*SignalKill()*	Pošle signál danému vláknu.

Neutrino a správce procesů může být nakonfigurován, aby umožňoval koexistenci vláken a procesů (jak definuje POSIX). Procesy jsou vzájemně paměťově chráněny, každý proces může obsahovat jedno nebo více vláken, která sdílí adresní prostor procesu.

3.2.1.1 Atributy vlákna

Ačkoliv vlákna uvnitř procesu sdílí vše uvnitř adresního prostoru procesu, každé vlákno má ještě svá vlastní data. V některých případech jsou tyto data chráněna uvnitř jádra (např. *tid* – thread ID), zatímco v jiných případech jsou nechráněna v adresním prostoru procesu (např. každé vlákno má svůj vlastní zásobník).

Příklady vlastních dat:

- *tid* – Každé vlákno je identifikováno číslem typu integer (thread ID), začíná od jedné a je unikátní uvnitř procesu.
- register set – Každé vlákno má vlastní programový čítač PC (Program Counter), ukazatel zásobníku SP (Stack Pointer) a další specifické registry procesoru
- stack – každé vlákno je vykonáváno na vlastním zásobníku, který je uložen v adresním prostoru procesu.
- signal mask – Každé vlákno má svojí signálovou masku.
- thread local storage – Vlákno má systémem definovanou datovou část TSL (Thread Local Storage), která slouží k uložení „per-thread“ informací (tj. *tid*, *pid*, bázová adresa zásobníku, errno a thread-specific key/data vazby). TSL nemusí být zpřístupněn přímo uživatelskou aplikací. Vlákno může mít uživatelsky definovaná data asociována s klíčem.
- cancellation handlers – callback funkce, které jsou vykonány po skončení vlákna.

Thread-specific data, která jsou implementována v *pthread* knihovně a uložena v TSL, poskytují mechanismus pro spojení globálního klíče procesu (typ integer) s unikátní per-thread hodnotou. Abychom mohli používat thread-specific data, musíme nejprve vytvořit nový klíč a poté mu přiřadit unikátní hodnotu (per-thread). Hodnota může být např. číslo typu integer nebo ukazatel na dynamicky alokovanou datovou strukturu. Klíč pak vrací přiřazenou per-thread hodnotu.

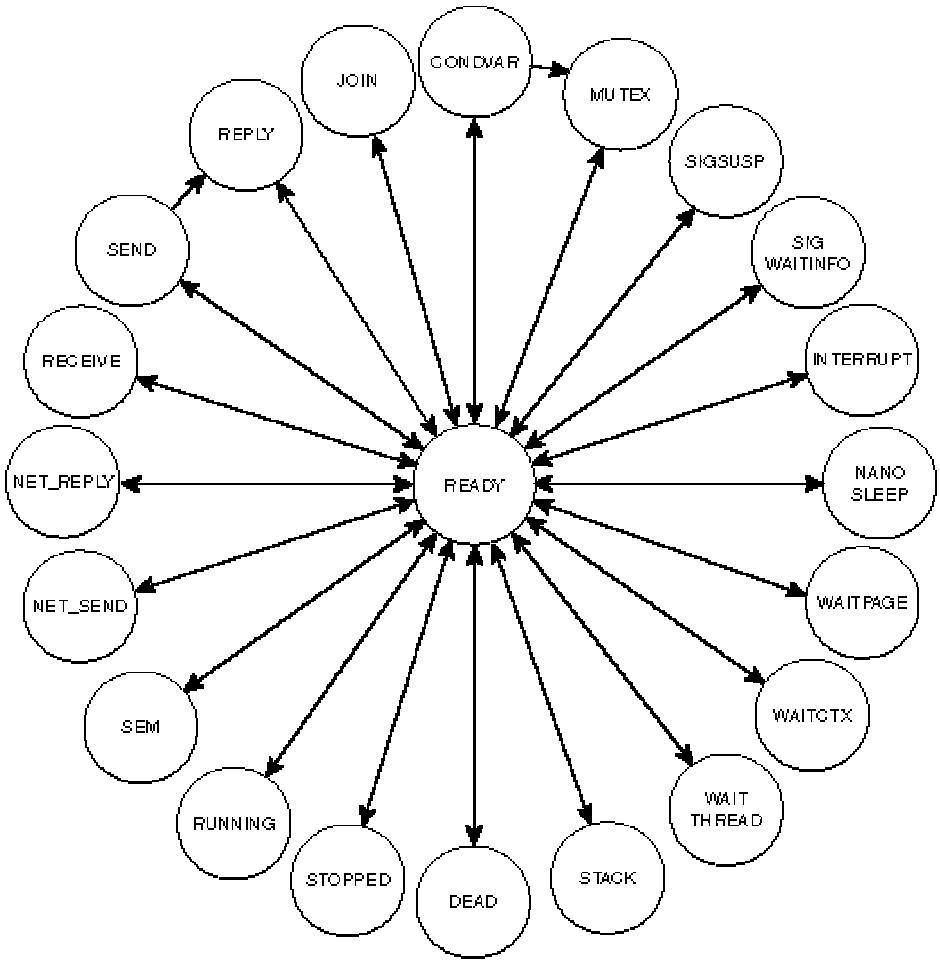
Tabulka 3.2: Funkce pracující s thread-specific daty

Function	Description
pthread_key_create()	Vytvoří klíč s destrukční funkcí.
pthread_key_delete()	Zruší klíč.
pthread_setspecific()	Sváže hodnotu s klíčem.
pthread_getspecific()	Vrátí hodnotu svázanou s klíčem.

3.2.1.2 Stavový automat vlákna

Počet vláken uvnitř procesu se může dosti měnit, dochází k dynamickému vytváření a rušení vláken. Vytvoření vlákna (*pthread_create()*) zahrnuje alokaci a inicializaci nutných prostředků uvnitř adresního prostoru procesu (např. zásobník vlákna) a začátek provádění vlákna.

Ukončení vlákna (*pthread_exit(), pthread_cancel()*) zahrnuje zastavení vlákna a uvolnění alokovaných prostředků. Při provádění vlákna se vlákno nachází většinou ve stavu „ready“ nebo „blocked“. Další stavy lze vidět z následujícího stavového automatu vlákna.



Obrázek 3.4: Stavy vlákna

CONDVAR - Vlákno je zablokováno na condition variable (tj. volalo *pthread_condvar_wait()*).

DEAD - Vlákno skončilo a čeká se na připojení dalšího vlákna.

INTERRUPT - Vlákno je blokováno čekáním na přerušení (tj. volalo *InterruptWait()*).

JOIN – Vlákno je blokováno čekáním na připojení dalšího vlákna (tj. volalo *pthread_join()*).

MUTEX – Vlákno je zablokováno na mutexu (tj. volalo *pthread_mutex_lock()*).

NANOSLEEP – Vlákno je krátký časový interval v klidovém stavu (tj. volalo *nanosleep()*).

NET_REPLY – Vlákno čeká na odpověď, která bude doručena pomocí sítě (tj. volalo *MsgReply*()*).

NET_SEND – Vlákno čeká na puls nebo signál ze sítě (tj. *MsgSendPulse()*, *MsgDeliverEvent()*, or *SignalKill()*).
READY – Vlákno čeká dokud neskončí provádění vlákna se stejnou nebo vyšší prioritou.
RECEIVE – Vlákno je blokováno příjmem zprávy (tj. volalo *MsgReceive()*).
REPLY – Vlákno je blokováno odpovědí na zprávu (tj. volalo *MsgSend()*, a server obdržel zprávu).
RUNNING – Vlákno je právě vykonáváno procesorem.
SEM – Vlákno čeká na semafor (tj. volalo *SyncSemWait()*).
SEND – Vlákno je blokováno posláním zprávy (tj. volalo *MsgSend()*, ale server ještě nepřijal zprávu).
SIGSUSPEND – Vlákno je blokováno čekáním na signál (tj. volalo *sigsuspend()*).
SIGWAITINFO – Vlákno je blokováno čekáním na signál (tj. volalo *sigwaitinfo()*).
STACK – Vlákno čeká na alokaci virtuálního adresního prostoru pro zásobník (rodič volal *ThreadCreate()*).
STOPPED – Vlákno je blokováno čekáním na signál SIGCONT.
WAITCTX – Vlákno čeká na non-integer (např. floating point) kontext až bude platný.
WAITPAGE – Vlákno čeká na alokaci fyzické paměti pro virtuální adresu.
WAITTHREAD – Vlákno čeká až jeho následník dokončí proceduru vytváření (tj. volalo *ThreadCreate()*).

3.2.1.3 Plánování vláken

Provádění aktivního vlákna je vždy dočasně pozastaveno, když nastane volání funkce jádra, výjimka nebo hardwarové přerušení. K přeplánování dojde vždy, když se změní stav některého vlákna. Vlákna jsou rozvrhována celkově na všech procesech (nezáleží na tom, uvnitř kterého procesu se vlákno nachází).

Vykonávání pozastaveného vlákna bude pokračovat, ale plánovač musí přepnout kontext z jednoho vlákna na druhé vždy, když je aktivní vlákno:

- blokováno – Vlákno musí čekat na výskyt některé události (čekání na mutex, atd.). Blokované vlákno je odebráno z fronty připravených vláken a vlákno s nejvyšší prioritou je spuštěno. Když je blokované vlákno následně odblokováno, je umístěno na konec fronty připravených vláken se stejnou prioritou.

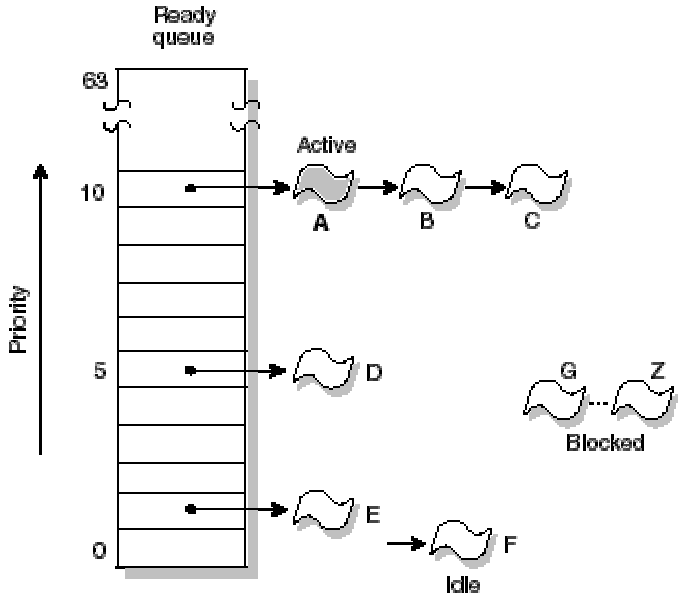
- přerušeno – Ve frontě připravených vláken je vlákno s vyšší prioritou. Přerušené vlákno se zařadí na začátek fronty připravených vláken se stejnou prioritou a vlákno s větší prioritou se spustí.

- uvolněno – Aktivní vlákno dobrovolně uvolní procesor (*sched_yield()*) a je zařazeno na konec fronty připravených vláken se stejnou prioritou. Vlákno s největší prioritou se spustí.

Prioritní plánování

Každému vláknu je přiřazena priorita. Plánovač vybírá následující vlákno ke spuštění podle priority z vláken, která jsou připravena.Vybráno je vlákno s nejvyšší prioritou.

Z Obrázku 3.5 je vidět, že vlákna A až F jsou připravena. Zbylá vlákna G až Z jsou blokována. Vlákno A je právě spuštěno. Vlákna A, B a C mají stejnou prioritu, v jakém pořadí budou spouštěny záleží na použitém plánovacím algoritmu.



Obrázek 3.5: Prioritní plánování vláken

Každé vlákno může mít prioritu od 1 do 63 (největší priorita), nezávisle na strategii plánování. Speciální *Idle* vlákno má prioritu 0 a vždy je připraveno ke spuštění. Vlákna standardně dědí prioritu po svých rodičovských vláknech.

Vlákno má skutečnou a efektivní prioritu, plánování se řídí efektivní prioritou. Vlákno může samo měnit obě priority současně. Normálně je efektivní priorita stejná jako skutečná priorita.

Připravená vlákna jsou ve frontě uspořádána podle priority. Fronta připravených vláken je implementována jako 64 oddělených front, každá pro jednotlivou prioritu. Tyto fronty jsou typu FIFO a první vlákno ve frontě s nejvyšší prioritou je vybráno ke spuštění.

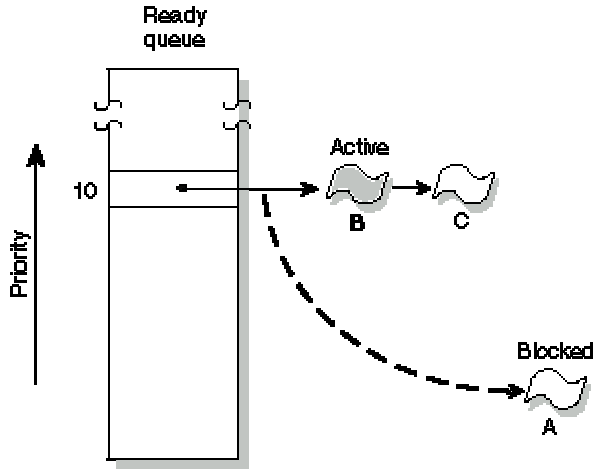
Plánovací algoritmy

QNX poskytuje tyto plánovací algoritmy:

- FIFO plánování
- round-robin plánování
- adaptivní plánování
- sporadické plánování.

FIFO a round-robin plánovací algoritmy se používají pouze v případě, kdy jsou připravena dvě a více vlákna se stejnou prioritou. Adaptivní a sporadické metody však používají „rozpočet“ pro vykonávání vláken. Ve všech případech, pokud se vlákno s vyšší prioritou stane připraveným, všechna vlákna s nižší prioritou budou okamžitě přerušena.

Na Obrázku 3.6 tři připravená vlákna se stejnou prioritou. Jestliže dojde k zablokování vlákna A, začne se vykonávat vlákno B.



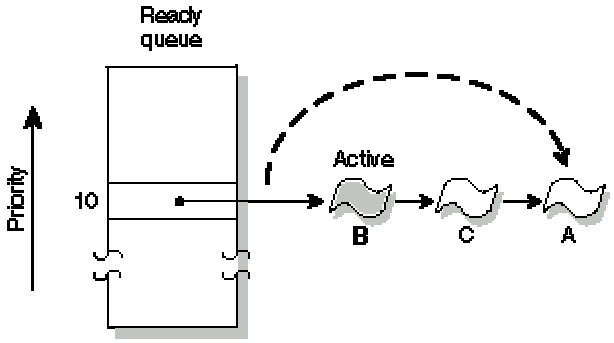
Obrázek 3.6: FIFO plánování vláken

Vlákno dědí plánovací algoritmus po svém rodičovském procesu a může požádat o jeho změnu.

FIFO plánování – vybrané vlákno se vykonává tak dlouho, dokud dobrovolně nepředá řízení (např. vlákno je zablokováno) nebo není přerušeno vláknem s vyšší prioritou.

Round-robin plánování - vybrané vlákno se vykonává tak dlouho, dokud dobrovolně nepředá řízení nebo není přerušeno vláknem s vyšší prioritou nebo nevyprší jeho *timeslice*.

Na Obrázku 3.7 je vidět, že po vypršení *timeslice* vlákna A dojde k aktivaci vlákna B.

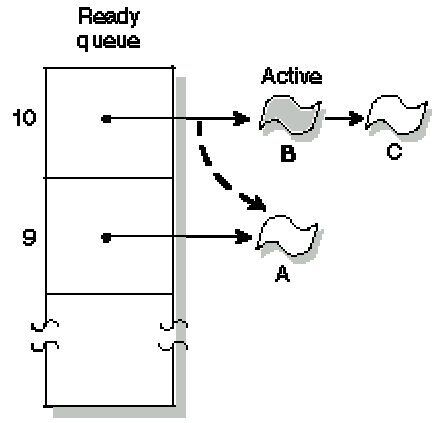


Obrázek 3.7: Round-robin plánování

Timeslice je časový interval přidělený každému procesu. Jakmile vláknu vyprší jeho timeslice, vlákno je přerušeno a dalšímu připravenému vláknu se stejnou prioritou je předáno řízení. Timeslice je čtyřnásobek periody hodin (*ClockPeriod()*).

Adaptivní plánování – Jestliže vláknu vyprší jeho timeslice (tj. není zablokováno), dekrementuje se priorita vlákna. Pokud dojde k zablokování vlákna, je okamžitě vláknu přiřazena původní priorita. Priorita vlákna se však může snížit pouze o jednu úroveň od původní priority.

Obrázek 3.8 ukazuje, že po vypršení timeslice vlákna A dojde k dekrementaci jeho priority a spuštění vlákna B.



Obrázek 3.8: Adaptivní plánování

Adaptivní plánování se málokdy používá pro realtimeové řídicí systémy.

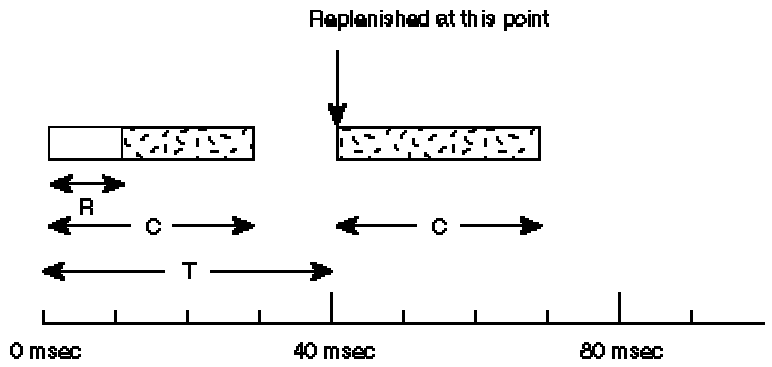
Sporadické plánování – Stejně jako u FIFO plánování se vlákno vykonává dokud nedojde k jeho zablokování nebo přerušení vláknem s vyšší prioritou a stejně jako u adaptivního plánování dochází ke snižování priority, ale u sporadického plánování máme přesnější kontrolu nad chováním vlákna.

Priorita vlákna se může dynamicky měnit mezi normální (*běží na popředí*) a nízkou (*běží na pozadí*) prioritou. Následující parametry slouží ke změně chování sporadického plánování:

Počáteční rozpočet (C) – Časový interval, ve kterém je vykonáváno vlákno s normální prioritou (N) předtím, než je jeho priorita snížena na nízkou prioritu (L).

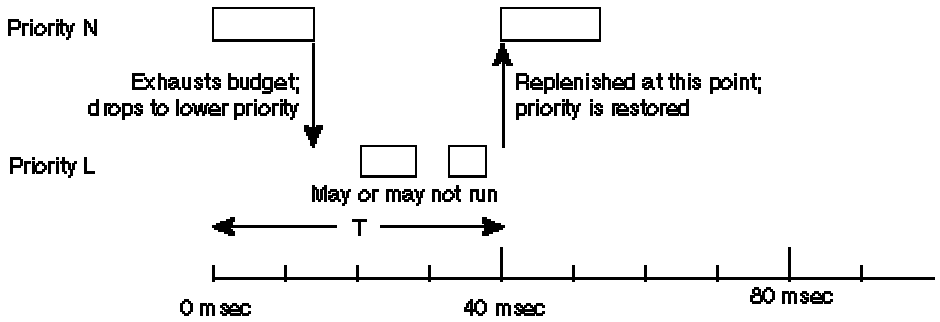
Perioda doplnění rozpočtu (T) – Časový interval během kterého je vláknu dovoleno spotřebovat svůj rozpočet. POSIXová implementace používá tuto hodnotu také jako ofset od času, kdy se vlákno stane připravené.

Na Obrázku 3.9 je vidět počáteční rozpočet (C), který je spotřebováván spuštěným vláknem a je periodicky doplněn (po době T). Rozpočet, který byl spotřebován (R), než došlo k zablokování vlákna bude doplněn v některém pozdějším čase (např. 40 ms), poté co se vlákno stane poprvé připraveným k vykonávání.



Obrázek 3.9: Periodické doplnění rozpočtu

Vlákno s normální prioritou N bude vykonáváno po dobu definovanou počátečním rozpočtem C. Po uplynutí této doby bude priorita vlákna snížena na nízkou prioritu L, dokud nenastane doplnění rozpočtu. Předpokládejme systém, kde vlákno nebude nikdy zablokováno ani přerušeno.

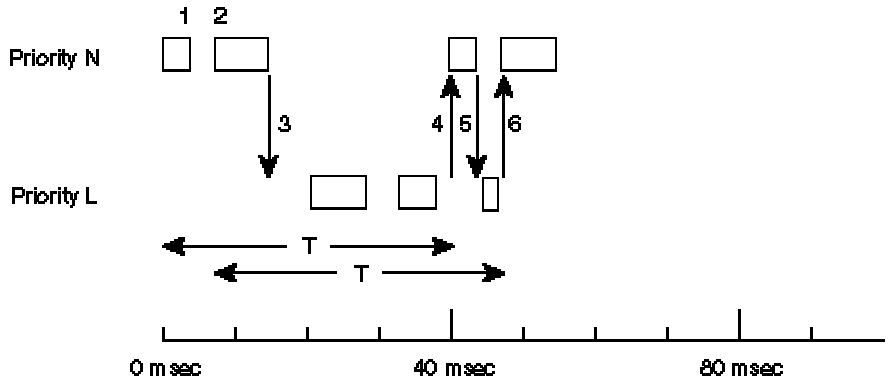


Obrázek 3.10: Snížení priority vlákna

Vlákno s nízkou prioritou L (na pozadí) může nebo nemusí být spuštěno v závislosti na prioritách dalších vláken v systému.

Jakmile nastane doplnění, je priorita vlákna zvýšena na normální prioritu. U správně nakonfigurovaného systému je zajištěno, že během každé doby T bude vlákno spuštěno maximálně po dobu C. Tím je zaručeno, že vlákno spuštěné s prioritou N spotřebuje pouze C/T*100 procent systémových prostředků.

Na Obrázku 3.11 je znázorněno vlákno, které má rozpočet C = 10 ms, který spotřebovává během každé doby doplnění T = 40 ms.



Obrázek 3.11: Vícenásobné doplnění rozpočtu

1. Vlákno je po 3 ms zablokováno. Za 40 ms od začátku spuštění vlákna dojde k operaci doplnění 3 ms (tj. vyprší první doba doplnění).
2. Vlákno je opět spuštěno po 3 ms, začíná druhá perioda doplnění (T). Vlákno je vykonáváno zbylých 7 ms.
3. Vlákno běží bez zablokování 7 ms, čímž vyčerpá svůj rozpočet a jeho priorita se sníží na nízkou prioritu (L), kde může nebo nemusí být vykonáváno. Doplnění 7 ms nastane po uplynutí druhé doby doplnění (v 46. ms).
4. Ve 40. ms má dojít k doplnění 3 ms, proto je vláknu vrácena normální priorita (N).
5. Vlákno spotřebovalo 3 ms svého rozpočtu a jeho priorita je opět snížena na nízkou prioritu.
6. Ve 46. ms dojde k doplnění 7 ms a priorita vlákna je opět zvýšena na normální prioritu.

Perioda vlákna bude oscilovat mezi normální (N) a nízkou (L) prioritou.

3.2.1.4 Manipulace s prioritou a plánovacími algoritmy

Priorita vlákna se může během vykonávání měnit, buď jí mění přímo vykonávané vlákno a nebo vlákno s vyšší prioritou pomocí vyslané zprávy. Můžeme také vybrat plánovací algoritmus, který bude jádro používat pro dané vlákno.

Tabulka 3.3: POSIXová a microkernelová volání, která slouží k manipulaci s prioritou a plánovacími algoritmy

POSIX call	Microkernel call	Description
sched_getparam()	*SchedGet()*	Získá prioritu vlákna.
sched_setparam()	*SchedSet()*	Nastaví prioritu vlákna.
sched_getscheduler()	*SchedGet()*	Získá plánovací algoritmus.
sched_setscheduler()	*SchedSet()*	Nastaví plánovací algoritmus.

3.2.2 Synchronizační služby

Tabulka 3.4: Synchronizační služby Neutrina

Synchronization service	Supported between processes	Supported across a QNX LAN
Mutexes	Yes	No
Condvars	Yes	No
Barriers	No	No
Sleepon locks	No	No
Reader/writer locks	No	No
Semaphores	Yes	Yes (named only)
FIFO scheduling	Yes	No
Send/Receive/Reply	Yes	Yes
Atomic operations	Yes	No

V Tabulce 3.4 jsou uvedeny synchronizační služby, které jsou přímo implementovány v Neutrinu, kromě:

- bariér, sleepon locků a reader/writer locků, které jsou vytvořeny z mutexů a condvars
- atomic operations, které jsou buď implementovány přímo v procesoru a nebo emulovány v jádru.

3.2.2.1 Mutual exclusion locks

Mutual exclusion locks (mutexy) jsou nejjednodušší synchronizační služby Neutrina. Používají se k zajištění exklusivního přístupu k datům sdílených mezi vlákny (kritická oblast). Funkce *pthread_mutex_lock()* se používá k získání mutexu a funkce *pthread_mutex_unlock()* k jeho uvolnění.

Mutex může získat v danou chvíli pouze jedno vlákno. Vlákna pokoušející se získat obsazený mutex budou zablokována dokud nedojde k uvolnění mutexu. Po uvolnění mutexu získá mutex zablokované vlákno s největší prioritou, čímž dojde k jeho odblokovaní a stane se novým vlastníkem mutexu.

Existuje neblokující funkce *pthread_mutex_trylock()*, která zjistí zdali je daný mutex volný. Pro dosažení největšího výkonu by měl být čas vykonávání kritické oblasti co nejkratší. Pokud chceme zablokovat vlákno uvnitř kritické oblasti měli bychom použít condvar.

Dědění priorit

Jestliže vlákno s vyšší prioritou než má vlastník mutexu se pokouší získat obsazený mutex, pak dojde ke zvýšení efektivní priority vlastníka mutexu na prioritu zablokovaného vlákna s vyšší prioritou čekajícího na mutex. Když vlastník mutexu uvolní mutex dojde ke snížení jeho priority na skutečnou prioritu. Tento mechanismus zajistí, že zablokované vlákno s vyšší prioritou bude čekat na mutex co nejkratší možný čas a také řeší klasický problém inverze priorit.

3.2.2.2 Condition variables

Condition variable (condvar) se používá k zablokování vlákna uvnitř kritické oblasti dokud není splněna patřičná podmínka. Podmínka může být libovolně složitá a je nezávislá na condvar. Při implementaci monitoru je nezbytné použít condvar spolu se získáváním mutexu.

Condvar podporují tyto operace:

- wait (*pthread_cond_wait()*)
- signal (*pthread_cond_signal()*)
- broadcast (*pthread_cond_broadcast()*).

Poznámka: Condvar signál nemá nic společného s POSIXovým signálem.

Níže je uveden typický příklad použití condvar:

```
pthread_mutex_lock( &m );
. . .
while (!arbitrary_condition) {
    pthread_cond_wait( &cv, &m );
    }
. . .
pthread_mutex_unlock( &m );
```

V tomto příkladu je získán mutex před testováním podmínky, což zajistí, že pouze toto vlákno má během svého vykonávání přístup k podmínce. Zatímco je podmínka splněna je vlákno zablokováno a čeká dokud některé jiné vlákno nevykoná signal nebo broadcast na condvar.

Cyklus while je nutný ze dvou důvodů. Za prvé POSIX nemůže garantovat, že nenastane falešné vzbuzení vlákna (např. více-procesorové systémy). Za druhé když jiné vlákno změní podmínku, tak potřebujeme opětovným testem zajistit, že změna odpovídá našemu kritériu. Když je čekající vlákno zablokováno, přidružený mutex je automaticky uvolněn pomocí *pthread_cond_wait()*, aby mohlo jiné vlákno vstoupit do kritické oblasti.

Vlákno které vykoná signal odblokuje vlákno s největší prioritou, které čeká na condvar. Zatímco broadcast odblokuje všechna vlákna čekající na condvar. Přidružený mutex je automaticky získán odblokovaným vláknem s nejvyšší prioritou. Vlákno po projití kritickou oblastí musí uvolnit mutex.

Existuje také verze condvar s timeoutem (funkce *pthread_cond_timedwait()*). Čekající vlákno může být odblokováno po vypršení timeoutu.

3.2.2.3 Bariéry

Bariera je synchronizační mechanismus, který umožňuje zachytit několik spolupracujících vláken (např. počítání s maticemi), přinutit je čekat v daném místě dokud všichni nedokončí výpočet. Poté může kterékoliv vlákno pokračovat v další činnosti.

Nejprve vytvoříme bariéru pomocí funkce *pthread_barrier_init()*:

```
#include <pthread.h>

int
pthread_barrier_init (pthread_barrier_t *barrier,
                      const pthread_barrierattr_t *attr,
                      unsigned int count);
```

Tato funkce vytvoří objekt bariéra na adrese dané ukazatelem na objekt (*barrier*) s atributy zadanými v *attr*. Parametr *count* udává počet vláken, které musí volat funkci *pthread_barrier_wait()*.

Poté co je vytvořen objekt bariéra, každé vlákno může zavolat funkci *pthread_barrier_wait()*, čímž naznačuje, že dokončil potřebné výpočty:

```
#include <pthread.h>

int pthread_barrier_wait (pthread_barrier_t *barrier);
```

Když vlákno zavolá funkci *pthread_barrier_wait()*, zablokuje se dokud počet vláken volajících tuto funkci není roven parametru *count* definovaného při vytváření bariéry. Pokud funkci zavolá potřebný počet vláken, všechna tato vlákna budou ve stejném okamžiku odblokována.

Tabulka 3.5: Funkce manipulující s bariérou

Function	Description
pthread_barrierattr_init()	Vytvoří objekt atributů.
pthread_barrierattr_destroy()	Zruší objekt atributů.
pthread_barrierattr_getpshared()	Získá hodnotu atributů.
pthread_barrierattr_setpshared()	Nastaví hodnotu atributů.
pthread_barrier_init()	Vytvoří bariéru.
pthread_barrier_destroy()	Zruší bariéru.
pthread_barrier_wait()	Čekání na bariéře.

3.2.2.4 Sleepon locks

Sleepon locky jsou velmi podobný condvars s několika málo rozdíly. Stejně jako condvars mohou být použity k zablokování vlákna (*pthread_sleepon_lock()*), dokud není splněná patřičná podmínka. Ale na rozdíl od condvars, které musí být alokovány pro každé kontrolované podmínky, sleepon locky multiplexují svojí funkčnost přes jediný mutex a dynamicky alokovaný condvar bez ohledu na počet kontrolovaných podmínek. Tyto sleepon locky jsou vytvořeny podle sleepon locků používaných obvykle v UNIXových jádrech.

3.2.2.5 Reader/writer locks

Reader/writer locky se používají v případě, že více vláken čte data z datové struktury a nejvýše jedno vlákno data zapisuje. Pokud vlákna žádají o zamčení dat z důvodů jejich čtení (*pthread_rwlock_rdlock()*), je všem vláknům vyhověno. Ale když hodlají zamknout data pro zápis (*pthread_rwlock_wrlock()*), je jejich žádost zamítnuta dokud všechna aktuální vlákna, která mají zamčená data pro čtení, je neodemknou (*pthread_rwlock_unlock()*).

Jestliže více vláken hodlá zapisovat do chráněné datové struktury, jsou tato vlákna prioritně řazena do fronty čekajících vláken a je všem žádostem o zápis dat vyhověno nehledě na žádosti vláken o čtení dat ani na jejich priority.

Existují také neblokující funkce *pthread_rwlock_tryrdlock()* a *pthread_rwlock_trywrlock()*. Tyto funkce vrací buď úspěšný pokus o zamknutí nebo status ukazující, že nemohlo dojít k zamknutí. Reader/writer locky nejsou implementovány přímo v jádře, ale jsou vytvořeny z mutexů a condvars poskytovaných jádrem.

3.2.2.6 Semafory

Semafory jsou další běžnou synchronizační službou, která umožňuje vláknům „poslat“ (*sem_post()*) semafor nebo „čekat“ (*sem_wait()*) na semafor a tím řídit jejich probuzení nebo uspání. Funkce *sem_post()* inkrementuje semafor a funkce *sem_wait()* ho naopak dekrementuje. Jestliže vlákno dekrementuje semafor s kladnou hodnotou, nebude vlákno zablokováno. Dekrementuje-li nekladný semafor, je vlákno zablokováno, dokud jiné vlákno neinkrementuje daný semafor.

Podstatný rozdíl mezi semafory a ostatními synchronizačními primitivy je v tom, že semafory jsou „async safe“ a mohou s nimi zacházet signal handlery (např. signal handler pomocí semaforu probudí spící vlákno).

Další užitečnou vlastností semaforů je jejich definice funkčnosti mezi procesy. Ačkoliv mutexy také fungují mezi procesy, je v POSIXovém vláknovém standardu tato schopnost považována za nepovinnou a nemusela by být přenositelná na jiné systémy. Pro synchronizaci vláken v jednom procesu jsou mutexy účinnější než semafory.

Užitečnou variantou semaforů jsou pojmenované semafory, které využívají resource manager, což dovoluje použití semaforů mezi procesy na vzdálených počítačích připojených k počítačové síti. Tyto semafory jsou ovšem pomalejší než nepojmenované semafory.

3.2.2.7 Synchronizace pomocí plánovacích algoritmů

Pokud vybereme POSIXový FIFO plánovací algoritmus, můžeme garantovat, že žádná dvě vlákna se stejnou prioritou nebudou současně vykonávat kritickou oblast na non-SMP (jednoprocesorovém) systému. FIFO plánování zajistí, že všechna vlákna se stejnou prioritou poběží, pokud budou naplánována, dokud dobrovolně neuvolní procesor jinému vláknu. Uvolnění procesoru může nastat, když se vlákno zablokuje při požadování služby jiného procesu nebo když je doručen signál.

Tento způsob synchronizace nemůžeme použít u víceprocesorových systémů, protože na každém procesoru by mohlo běžet jedno vlákno.

3.2.2.8 Synchronizace pomocí message passing

Meziprocesní služby Neutrina posílání zpráv Send/Receive/Reply (popsány později) implementují implicitní synchronizaci blokujícího charakteru. Tyto služby jsou také synchronizační a meziprocesní primitiva (jiná než pojmenované semafory, které jsou vybudovány na vrcholu posílání zpráv), která se dají použít po síti.

3.2.2.9 Synchronizace pomocí atomických operací

V některých případech chceme vykonat krátkou operaci (jako je zvětšení proměnné) s garancí, že operace bude vykonána atomicky (tj. operace nebude přerušena jiným vláknem nebo obsluhou přerušení).

Neutrino poskytuje následující atomické operace:

- přičtení hodnoty
- odečtení hodnoty
- smazání bitů
- nastavení bitů
- změna bitů.

Tyto atomické operace jsou dostupné po vložení C hlavičkového souboru `<atomic.h>`.

3.2.2.10 Implementace synchronizačních služeb

Tabulka 3.6: Funkce mikrojádra a POSIXové funkce implementující synchronizační služby

Microkernel call	POSIX call	Description
SyncTypeCreate()	*pthread_mutex_init()*, *pthread_cond_init()*, *sem_init()*	Vytvoří objekt pro mutex, condvar a semafor.
SyncDestroy()	*pthread_mutex_destroy()*, *pthread_cond_destroy()*, *sem_destroy()*	Zruší synchronizační objekt.
SyncCondvarWait()	*pthread_cond_wait()*, *pthread_cond_timedwait()*	Zablokování na condvar.
SyncCondvarSignal()	*pthread_cond_broadcast()*, *pthread_cond_signal()*	Probudí vlákno zablokované na condvar.
SyncMutexLock()	*pthread_mutex_lock()*, *pthread_mutex_trylock()*	Získání mutexu.
SyncMutexUnlock()	*pthread_mutex_unlock()*	Uvolnění mutexu.
SyncSemPost()	*sem_post()*	Pošle semafor.
SyncSemWait()	*sem_wait()*, *sem_trywait()*	Čeká na semafor.

3.2.3 Meziprocesní komunikace Neutrina

Meziprocesní komunikace (**I**nter **P**rocess **C**ommunication) hraje podstatnou roli při transformaci Neutrina z realtimeového embedded jádra na rozsáhlý POSIXový operační systém. K Neutrinu jsou přidávány různé služby ve formě speciálních procesů. IPC je jakési „lepidlo“, které spojí tyto komponenty do jednoho fungujícího celku.

Ačkoliv je message passing základní formou IPC v QNX a Neutrinu, jsou k dispozici i další formy IPC, které jsou vybudovány právě na základě message passing. Strategií je vytvořit jednoduché a robustní IPC služby, které mohou být naladěny na vykonávání po zjednodušené kódové cestě v mikrojádře.

Srovnávací test porovnávající higher-level IPC služby (jako jsou roury a FIFO implementované na základě message pasing) s jejich monolitickými protějšky, ukazuje srovnatelný výkon.

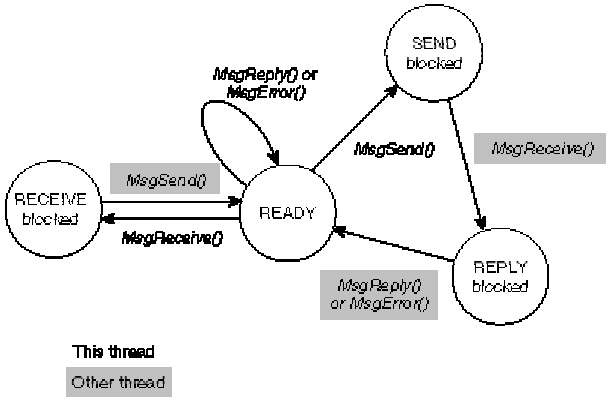
Tabulka 3.7: IPC služby Neutrina

Service:	Implemented in:
Message-passing	kernel
Signále	kernel
POSIX message queues	external process
Shared memory	kernel
Pipes	external process
FIFOs	external process

Message passing implementovaný ve funkcích *MsgSend()*, *MsgReceive()* a *MsgReply()*, je synchronní a kopíruje data. Nyní se blíže seznámíme s oběma vlastnostmi.

3.2.3.1 Synchronní message passing

Vlákno, které posílá zprávu (*MsgSend()*) jinému vláknu (které může být uvnitř jiného procesu) bude zablokováno dokud cílové vlákno nepřijme zprávu (*MsgReceive()*), nezpracuje zprávu a nevykoná *MsgReply()*. Jestliže vlákno zavolá funkci *MsgReceive()* a nečeká na vyřízení dříve poslané zprávy, bude zablokováno dokud jiné vlákno nezavolá funkci *MsgSend()*.



Obrázek 3.12: Stavový automat message passingu

Toto vnitřní blokování synchronizuje vykonávání vlákna posílajícího zprávu, protože žádost na poslání dat způsobí, že posílající vlákno bude zablokováno a přijímající vlákno bude rozvrhováno pro vykonávání. To se stane bez požadování jádra o explicitní práci na rozhodnutí, které další vlákno poběží (jako v případě většiny jiných forem IPC). Vykonání a přesun dat je prováděn přímo z jednoho kontextu do druhého.

Message passing neimplementuje řazení dat do fronty, protože řazení může být implementováno v přijímajícím vláknu. Posílající vlákno je kdykoli připravené čekat na odpověď, proto nemusíme vytvářet samostatné, explicitní blokovací volání pro čekání na odpověď.

Zatímco operace posílání a přijímání zpráv jsou blokující a synchronní, funkce *MsgReply()* a *MsgError()* blokující nejsou. Protože klientské vlákno je vždy zablokováno čekáním na odpověď, není zapotřebí dalších synchronizačních služeb, tudíž blokující *MsgReply()* není potřeba. Tento přístup dovolí serveru odpovědět klientovi a pokračovat v činnosti, zatímco jádro a nebo síťový kód asynchronně pošle data klientskému vláknu a označí ho připraveným pro vykonávání.

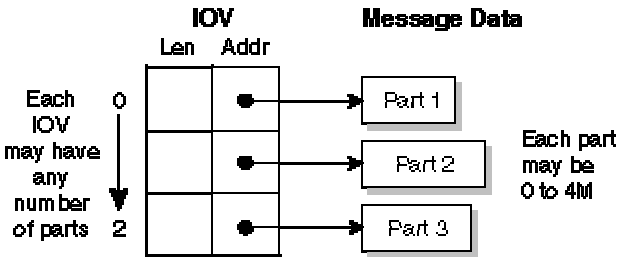
Funkce *MsgReply()* se používá pro vrácení žádného nebo několika bajtů klientskému vláknu. Funkce *MsgError()* vrací pouze status klientskému vláknu. Obě funkce odblokují klientské vlákno čekající na odpověď.

3.2.3.2 Kopírování zpráv

Protože message passing kopíruje zprávu přímo z adresního prostoru jednoho vlákna do adresního prostoru druhého vlákna bez použití bufferu, rychlost doručení zprávy záleží na šířce pásma operační paměti. Neutrino nepřidává žádná zvláštní data do obsahu zprávy, vzájemně definovaná data ve zprávě mají význam pouze pro odesílatele a příjemce.

Message passing podporuje přenos zprávy složené z více částí a to opět z jednoho adresního prostoru do druhého bez použití bufferu. Místo toho odesílající a přijímající vlákno specifikuje vektorovou tabulku, která ukazuje na odesílané a přijímané části zprávy v paměti. Velikost různých částí zprávy může být rozdílná pro odesílatele a příjemce.

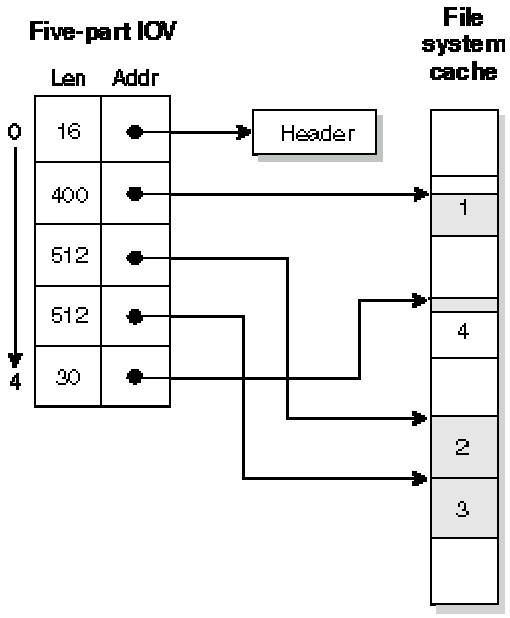
Přenos více částí zprávy umožňuje posílat zprávy, které mají hlavičku oddělenou od dat a to bez potřeby kopírování dat k vytvoření souvislé zprávy. Navíc, pokud je základní datová struktura kruhový buffer, skládající se ze tří částí, hlavičky a dvou oddělených částí, bude zpráva poslána jako jediná atomická zpráva. Hardwarovým ekvivalentem tohoto konceptu by mohl být DMA přenos, mající schopnost rozptýlit a shromáždit data (*scatter/gather*).



Obrázek 3.13: Přenos zprávy složené z oddělených částí

Přenosy více částí zprávy jsou také široce používány ve filesystémech. Při čtení jsou data kopírována přímo z filesystémové vyrovnávací paměti do aplikace používající zprávu s jednou částí pro reply status a *n* částí pro data. Každá datová část ukazuje do vyrovnávací paměti a kompenzuje skutečnost, že bloky vyrovnávací paměti nejsou souvislé, se začátkem nebo koncem čtení uvnitř bloku.

Obrázek 3.14 ukazuje čtení zprávy, skládající se z pěti částí o celkové velikosti 1454 B, z vyrovnávací paměti o velikosti bloku 512 B.



Obrázek 3.14: Čtení zprávy z vyrovnávací paměti

Protože zpráva je výlučně kopírována mezi adresními prostory (raději než manipulací se stránkovou tabulkou), může být jednoduše uložena na zásobníku místo ve speciálním bloku stránkované paměti. V důsledku toho může být mnoho knihovních rutin, které implementují API mezi klientskými a serverovými procesy, jednoduše implementováno bez použití zvláštních IPC volání pro alokaci paměti.

3.2.3.3 Jednoduché zprávy

Pokud se zpráva skládá pouze z jedné části, poskytuje Neutrino funkce, které pracují s ukazatelem ukazujícím přímo do bufferu bez použití IOV (Input Output Vector). V tomto případě je počet částí zprávy nahrazeno velikostí zprávy, na kterou je přímo ukazováno.

Tabulka 3.8: Funkce implementující poslání zprávy a odpovědi na zprávu

Function	Send message	Reply message
MsgSend()	Simple	simple
MsgSendsv()	Simple	IOV
MsgSendvs()	IOV	simple
MsgSendv()	IOV	IOV

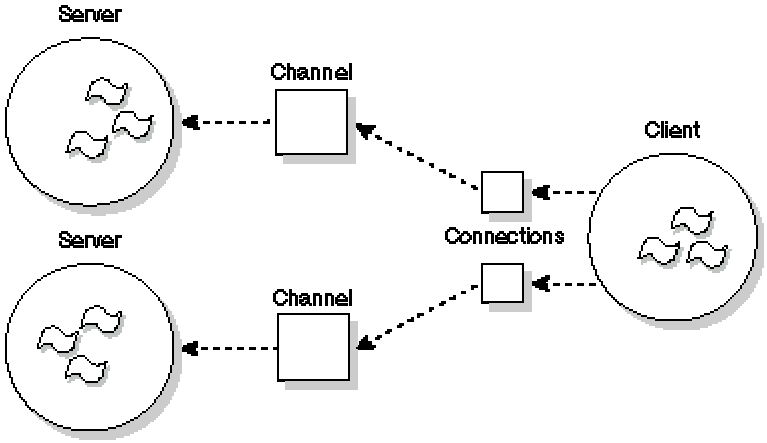
Ostatním funkcím, které pracují přímo se zprávou stačí jednoduše odebrat poslední písmeno „v“ v jejich názvu.

Tabulka 3.9 : Funkce implementující přímé a IOV zprávy

IOV	Simple direct
MsgReceivev()	*MsgReceive()*
MsgReceivePulsev()	*MsgReceivePulse()*
MsgReplyv()	*MsgReply()*
MsgReadv()	*MsgRead()*
MsgWritev()	*MsgWrite()*)

3.2.3.4 Komunikační kanál

V Neutrinu je lepší posílat zprávy komunikačním kanálem, než přímo z jednoho vlákna do druhého. Vlákno, které hodlá přijímat zprávy, nejprve vytvoří kanál a jiné vlákno, které mu hodlá poslat zprávu, se musí nejdříve k danému kanálu připojit. Pokud jsou všechna vlákna v procesu připojena ke stejnému kanálu, pak je toto připojení sdíleno mezi všechna vlákna. Kanály a připojení jsou uvnitř procesu označeny identifikátorem typu integer. Připojení je mapováno přímo do file deskriptoru (tj. connection ID).



Obrázek 3.15: Připojení ke komunikačnímu kanálu

Tabulka 3.10: Funkce implementující práci s komunikačním kanálem

Function	Description
ChannelCreate()	Vytvoří komunikační kanál k přijímání zpráv.
ChannelDestroy()	Zruší komunikační kanál.
ConnectAttach()	Vytvoří připojení ke komunikačnímu kanálu pro posílání zpráv.
ConnectDetach()	Zruší připojení ke komunikačnímu kanálu.

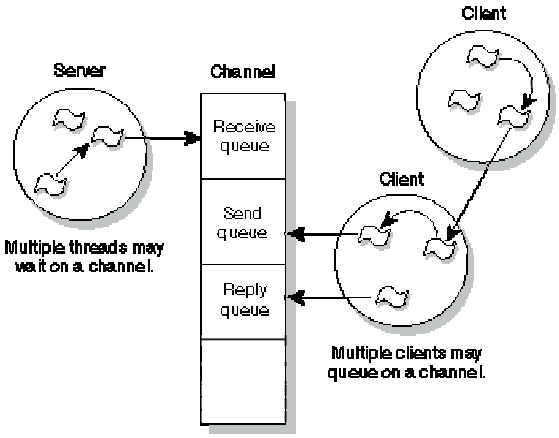
Ukázka implementace serveru pomocí událostní smyčky, ve které dochází k příjmu a zpracování zpráv:

```
chid = ChannelCreate(flags);
SETIOV(&iov, &msg, sizeof(msg));
for(;;) {
        rcv_id = MsgReceivev( chid, &iov, parts, &info );

        switch( msg.type ) {
        /* Perform message processing here */
        }

        MsgReplyv( rcv_id, &iov, rparts );
}
```

Komunikační kanál má tři fronty:

- jednu frontu pro vlákna čekající na zprávy
- jednu frontu pro vlákna, která poslala zprávu, ale zpráva nebyla dosud přijata
- jednu frontu pro vlákna, která poslala zprávu, která byla přijata, ale ještě nebyla doručena odpověď.

Vlákna čekající ve frontách jsou zablokována (tj. RECEIVE-, SEND- nebo REPLY-blocked).



Obrázek 3.16: Fronty komunikačního kanálu

Pulsy

Neutrino podporuje, kromě synchronních Send/Receive/Reply služeb, také neblokující zprávy pevné délky. Tyto zprávy se nazývají *pulsy* a obsahují 1B kódu a 4B dat. Pulsy se často používají jako oznamovací mechanismus uvnitř obsluhy přerušení.

Dědění priorit

Serverový proces přijímá zprávy podle jejich priorit. Jakmile vlákna běžící v serverovém procesu přijmou zprávu, zdědí prioritu posílajícího vlákna, ale nezdědí plánovací algoritmus. Toto dědění priorit řeší problém inverse priorit.

3.2.3.5 Implementace message passing API v Neutrinu

Tabulka 3.11 Funkce implementující message passing API

Function	Description
MsgSend()	Pošle zprávu a zablokuje se dokud neobdrží odpověď.
MsgReceive()	Čeká na zprávu.
MsgReceivePulse()	Čeká na puls.
MsgReply()	Pošle odpověď na zprávu.
MsgError()	Pošle error status jako odpověď na zprávu.
MsgRead()	Čte další data z přijmuté zprávy.
MsgWrite()	Zapisuje další data do odpovědi na zprávu.
MsgInfo()	Získá informace o přijaté zprávě.
MsgSendPulse()	Pošle puls.
MsgDeliverEvent()	Doručí událost klientovi.
MsgKeyData()	Klíčová zpráva kontrolující zabezpečení.

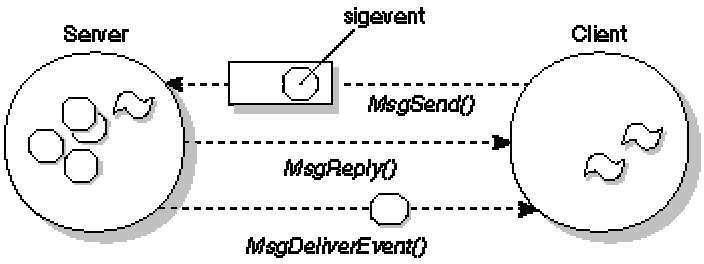
3.2.3.6 Události

Významným pokrokem při návrhu Neutrina je implementace podsystému obsluhujícího události. Tento podsystém zahrnuje veškeré oznamovací metody používané v Neutrinu. Výhoda tohoto přístupu je v tom, že schopnosti využívané výlučně jednou oznamovací metodou se mohou stát dostupné pro další metody (např. QNX aplikace může použít stejné služby řazení POSIXových realtimových signálů do fronty i na UNIXové signály). Což může zjednodušit robustní implementaci signál handlerů v aplikacích.

Tři možné zdroje událostí:

- vlákno volající funkci jádra *MsgDeliverEvent()*
- obsluha přerušení
- vypršení časovače.

Existuje několik rozdílných typů událostí: QNX pulsy, přerušení, různé formy signálů a události umožňující odblokovat zablokované vlákno.



Obrázek 3.17: Klient posílá serveru sigevent

3.2.3.7 Signály

Neutrino definuje 64 signálů, z toho 32 standardních POSIXových signálů, 16 POSIXových realtimových signálů a 8 speciálních signálů. Neutrino rozšiřuje POSIXový mechanismus doručování signálů, signály mohou být kromě procesů doručeny přímo vláknům.

Tabulka 3.12: Implementace signálů

Microkernel call	POSIX call	Description
SignalKill()	*kill()*, *pthread_kill()*, *raise()*, *sigqueue()*	Pošle signál procesní skupině, procesu nebo vláknu.
SignalAction()	*sigaction()*	Přiřazuje akci k signálu.
SignalProcmask()	*sigprocmask()*	Mění signálovou masku vlákna.
SignalSuspend()	*sigsuspend()*, *pause()*	Blokuje dokud signál nevyvolá signál handler.
SignalWaitinfo()	*sigwaitinfo()*	Čeká na signál a vrací o něm informace.

Originální POSIXová specifikace definuje operace pouze na procesech. V mnohovláknovém procesu platí následující pravidla:

- Signálové akce jsou udržovaný na procesní úrovni. Jestliže vlákno ignoruje nebo zachytí signál, ovlivní to všechny vlákna v procesu.
- Signálová maska je udržovaná na vláknové úrovni. Jestliže vlákno blokuje signál, ovlivní to pouze toto vlákno.
- Signál poslaný vláknu, který nelze ignorovat, bude doručen pouze danému vláknu.
- Signál poslaný procesu, který nelze ignorovat, bude doručen prvnímu vláknu, které nemá signál blokovaný. Pokud všechna vlákna mají signál blokovaný, signál bude zařazen do fronty procesu dokud některé vlákno neignoruje nebo neodblokuje signál. Jestliže vlákno ignoruje signál, bude signál odstraněn z fronty procesu. Jestliže vlákno odblokuje signál, bude mu doručen signál z fronty procesu.

Když je signál doručen procesu s mnoha vlákny, musí se prohledat tabulka vláken a najít vlákno, které neblokuje daný signál. V praxi se používá maskování signálů u všech vláken kromě jednoho, které je určeno pro jejich obsluhu. Ke zvýšení účinnosti doručování signálů přispívá ukládání vlákna, které naposledy přijalo signál. Jádro se pokaždé pokouší doručit signál nejprve uloženému vláknu.

POSIXový standard obsahuje koncept frontovaných realtimových signálů. Neutrino podporuje volitelné frontování některých signálů, ale ne realtimových. Každý signál může mít přiřazen 8 bitový kód a 32 bitovou hodnotu, což se velmi podobá pulsům. Jádro využívá této podobnosti a používá společný kód pro zpravování signálů a pulsů. Číslo signálu (*signo*) je mapováno na prioritu pulsu pomocí vztahu _SIGMAX - *signo*. Důsledkem toho je prioritní doručování signálů takové, že signál s nižším čílsem má vyšší prioritu.

3.2.3.8 Speciální signály Neutrina

Speciální signály nemohou být ignorovány ani chyceny. Pokus o volání POSIXové funkce *sigaction()* nebo funkce jádra *SignalAction()* selže (nelze měnit speciální signály) s errorem EINVAL. Navíc tyto signály jsou stále blokovány a mají povoleno frontování. Pokus o odblokování těchto signálů pomocí POSIXové funkce *sigprocmask()* nebo funkce jádra *SignalProcmask()* bude ignorován.

Tabulka 3.13 Rozmezí jednotlivých signálů

Signal range	Description
1 ... 56	56 POSIXových signálů (obsahují tradiční UNIXové signály)
41 ... 56	16 POSIXových realtimových signálů (SIGRTMIN až SIGRTMAX)
57 ... 64	8 speciálních signálů Neutrina (SIGSPECIALMIN až SIGSPECIALMAX)

Níže je uvedena část kódu zajišťující chování běžného signálu jako speciálního signálu. Speciální signály šetří programátory od psaní tohoto kódu a chrání signál od neúmyslných změn v jeho chování.

```
sigset_t *set;
struct sigaction action;

sigemptyset(&set);
sigaddset(&set, signo);
sigprocmask(SIG_BLOCK, &set, NULL);

action.sa_handler = SIG_DFL;
action.sa_flags = SA_SIGINFO;
sigaction(signo, &action, NULL);
```

Následující kód je vhodný pro synchronní oznamování s použitím funkcí *sigwaitinfo()* nebo *SignalWaitinfo()*. Vykonávání kódu bude zablokováno, dokud nebude přijat osmý speciální signál.

```
sigset_t *set;
siginfo_t info;

sigemptyset(&set);
sigaddset(&set, SIGSPECIALMAX);
sigwaitinfo(&set, &info);
printf("Received signal %d with code %d and value %d\n",
        info.si_signo,
        info.si_code,
        info.si_value.sival_int);
```

Protože speciální signály jsou stále blokovány, program nemůže být přerušen ani zrušen, jestliže je speciální signál doručen mimo funkci *sigwaitinfo()*. Ke ztrátě signálů nemůže dojít díky jejich řazení do fronty.

Tyto signály byly navrženy k tomu, aby řešily běžné IPC požadavky, kdy server hodlá oznámit klientovi, že má pro něj dostupná data. Server použije k oznámení funkci *MsgDeliverEvent()*, která může poslat buď puls a nebo signál. Pulsy jsou preferovány v případě, že klient může být také serverem pro další klienty.

3.2.3.9 POSIXové fronty zpráv

POSIX definuje sadu neblokujících prostředků posílání zpráv známých jako fronty zpráv. Fronty zpráv jsou pojmenované objekty stejně jako roury. Prioritní fronta zpráv má bohatší strukturu než roura a tím nabízí větší aplikační kontrolu nad komunikací. POSIXové fronty jsou implementovány v Neutrinu pomocí zvláštního resource manageru nazývaného `mqueue`.

Podle přesného POSIXového výkladu bychom měli vytvářet fronty zpráv, které začínají lomítkem (/) a neobsahují žádné další. Neutrino rozšiřuje POSIXový standard na podporu názvů front, které mohou obsahovat více lomítek.

Tabulka 3.14: Funkce implementující fronty zpráv

Function	Description
Mq_open()	Otevře nebo vytvoří frontu zpráv.
Mq_close()	Zavře frontu zpráv.
Mq_unlink()	Zruší frontu zpráv.
Mq_send()	Pošle zprávu do fronty.
Mq_receive()	Čte zprávu z fronty.
Mq_notify()	Oznámí volajícímu procesu, která zpráva je dostupná ve frontě.
Mq_setattr()	Nastaví parametry fronty.
Mq_getattr()	Získá parametry fronty.

3.2.3.10 Sdílená paměť

Sdílená paměť nabízí největší dostupnou šířku pásma IPC. Přístup do sdílené paměti není sám o sobě synchronizovaný, proto se často používají synchronizační primitiva. Vhodná synchronizační primitiva jsou jak semafory, tak mutexy. Obecně jsou mutexy účinnější než semafory.

Sdílená paměť a message passing

Kombinací sdílené paměti a message passingu vznikne IPC, která nabízí:

- velmi vysoký výkon (sdílená paměť)
- synchronizaci (message passing)
- transparentnost (message passing).

Místo toho, aby si klient se serverem posílali všechna data pomocí message passingu, klient pošle serveru odkaz na oblast sdílené paměti a server může přímo číst a zapisovat data do sdílené paměti. Jednoduchou sdílenou paměť nemohou však využívát procesy na různých počítačích připojených do sítě. Server může využívat sdílenou paměť pro lokální klienty, ale pro vzdálené klienty musí všechna data posílat pomocí message passingu.

Vytvoření objektu sdílené paměti

Vlákna v procesu sdílí paměť tohoto procesu. Pokud chceme sdílet paměť mezi procesy, musíme nejprve vytvořit oblast sdílené pamětí a poté jí namapovat do adresního prostoru procesu.

Tabulka 3.15: Funkce pracující se sdílenou pamětí

Function	Description
shm_open()	Otevře nebo vytvoří oblast sdílené paměti.
close()	Zavře oblast sdílené paměti.
mmap()	Namapuje oblast sdílené paměti do adresního prostoru procesu.
munmap()	Odmapuje oblast sdílené paměti z adresního prostoru procesu.
mprotect()	Změní zabezpečení sdílené paměťové oblasti.
msync()	Synchronizuje paměť s fyzickou pamětí.
shm_ctl()	Přidělí sdíleného paměťového objektu speciální atributy.
shm_unlink()	Zruší oblast sdílené paměti.

POSIXová sdílená paměť je implementována v Neutrinu pomocí proces manageru (`procnto`). Funkce v Tabulce 3.15 jsou implementovány jako zprávy pro `procnto`.

Velikost nově vytvořeného objektu sdílené paměti je nulová. K nastavení patřičné velikosti se používá funkce *ftruncate()* nebo *shm_ctl()*.

3.2.3.11 Roury a fronty FIFO

Abychom mohli používat roury a fronty FIFO, musí být v QNX spuštěn patřičný resource manager (`pipe`).

Roury

Roura je nepojmenovaný soubor, který slouží jako jednosměrný I/O kanál mezi dvěma nebo více spolupracujícími procesy. Jeden proces zapisuje data do roury a ostatní procesy čtou data z roury. `Pipe` manager se stará o bufferování dat. Velikost bufferu je definována v hlavičkovém souboru `<limits.h>` jako PIPE_BUF. Roura je odstraněna, pokud je jeden z konců roury uzavřen.

Typické použití roury je spojení výstupu jednoho programu se vstupen druhého programu. Například v shellu:

```
ls | more
```

přesměruje pomocí roury standardní výstup z `ls` utility na standardní vstup utility `more`.

Tabulka 3.16: Vytvoření roury

Pokud chceme	Použijeme
vytvořit rouru v shellu.	pipe symbol ("`|`").
vytvořit rouru v programu.	funkce *pipe()* nebo *popen()*.

Fronty FIFO

Fronty jsou v podstatě stejné jako roury kromě toho, že fronty jsou trvalé pojmenované soubory uložené v adresářích filesystému.

Tabulka 3.17: Vytvoření a zrušení fronty FIFO

Pokud chceme	Použijeme
vytvořit frontu FIFO v shellu.	`mkfifo` utility.
vytvořit frontu FIFO v programu.	funkci *mkfifo()*.
zrušit frontu FIFO v shellu.	`rm` utility.
zrušit frontu FIFO v programu.	funkce *remove()* or *unlink()*.

3.2.4 Časovače a hodiny

3.2.4.1 Časovače

Neutrino poskytuje úplnou funkčnost POSIXových časovačů. Časovače jsou nenákladným prostředkem v jádru, protože jsou rychle vytvořeny i obslouženy.

POSIXový model časovače je docela bohatý, umožňuje nastavit dobu vypršení časovače:

- absolutně
- relativně
- cyklicky.

Tabulka 3.18: Implementace časovačů

Microkernel call	POSIX call	Description
TimerAlarm()	*alarm()*	Nastaví procesní alarm.
TimerCreate()	*timer_create()*	Vytvoří časovač.
TimerDestroy()	*timer_delete()*	Zruší časovač.
TimerGettime()	*timer_gettime()*	Vrací zbývající čas do vypršení časovače.
TimerGetoverrun()	*timer_getoverrun()*	Vrací počet přetečení časovače.
TimerSettime()	*timer_settime()*	Spustí časovač.
TimerTimeout()	*sleep()*, *nanosleep()*, *sigtimedwait()*, *pthread_cond_timedwait()*, *pthread_mutex_trylock()*, *intr_timed_wait()*	Nastaví timeout pro některé blokující stavy.

3.2.4.2 Hodiny

Tabulka 3.19: Funkce implementující manipulaci s hodinami

Microkernel call	POSIX call	Description
ClockTime()	*clock_gettime()*, *clock_settime()*	Získá nebo nastaví systémový čas.
ClockAdjust()	N/A	Synchronizace hodin.
ClockCycles()	N/A	Vrací aktuální hodnotu volně běžícího 64-bitového čítače cyklů.
ClockPeriod()	*clock_getres()*	Získá nebo nastaví periodu hodin.
ClockId()	*clock_getcpuclockid()*, *pthread_getcpuclockid()*	Vrací parametr, který byl použit ve funkci *ClockTime()* jako `clockid_t`.

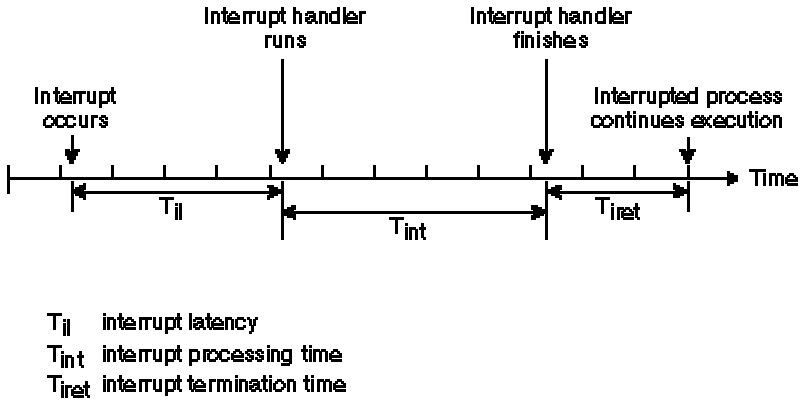
3.2.5 Interrupt handling

V realtimových systémech je velmi důležité minimalizovat dobu od výskytu externí události do začátku vykonávání kódu uvnitř vlákna, které obsluhuje danou událost. Nás budou zajímat dvě zpoždění a sice interrupt latency a scheduling latency.

3.2.5.1 Interrupt latency

Interrupt latency je doba od vzniku přerušení do vykonání první instrukce ovladače zařízení obsluhujícího přerušení. Operační systém povoluje přerušení téměř pocelou dobu jeho běhu, takže interrupt latency je typicky zanedbatelný. Ale určitá kritická sekce kódu vyžaduje dočasné zakázání přerušení. Maximální dobu zakázání přerušení obvykle definuje nejhorší případ interrupt latency, který je u QNX velmi malý.

Obrázek 3.19 ukazuje případ, kdy je hardwarové přerušení zpracováno pevně stanovenou obsluhou přerušení.



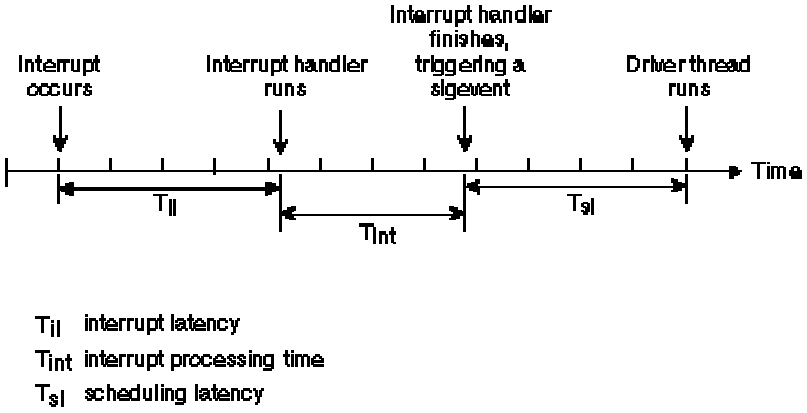
Obrázek 3.18: Obsluha přerušení jednoduše skončí

Interrupt latency (T*il*) v Obrázku 3.19 reprezentuje *minimální* zpoždění, které nastane, pokud v době výskytu přerušení bylo povoleno přerušení. Nejhorší případ interrupt latency bude toto zpoždění *plus* nejdelší čas, ve kterém Neutrino nebo běžící proces Neutrina zakázali přerušení procesoru.

3.2.5.2 Scheduling latency

V některých případech musí obsluha hardwarového přerušení nízké úrovně rozvrhovat spuštění vlákna vyšší úrovně. To se děje tak, že obsluha přerušení vrací událost, která je doručena patřičnému vláknu.

Scheduling latency je doba od vykonání poslední instrukce obsluhy přerušení do provedení první instrukce obslužného vlákna. Tato doba obvykle představuje čas potřebný na uložení kontextu vykonávaného vlákna a obnovení kontextu obslužného vlákna. Ačkoliv je scheduling latency větší než interrupt latency, je rovněž udržována na malé hodnotě.

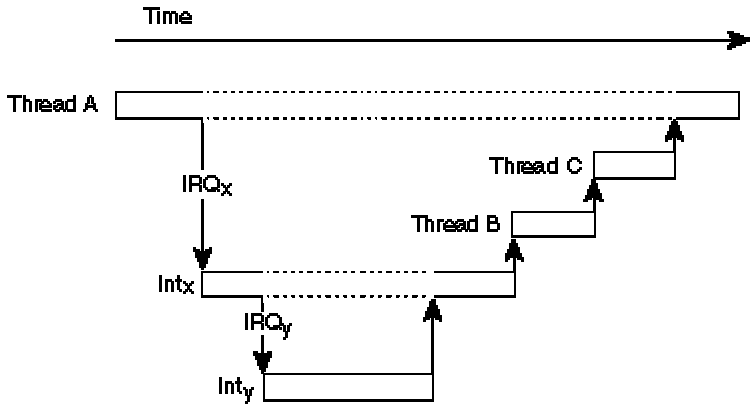


Obrázek 3.19: Obsluha přerušení vrací událost

3.2.5.3 Vnořená přerušení

Tento mechanismus je u QNX plně podporován. V předchozích případech jsem popisoval nejjednodušší a nejběžnější situaci, kdy nastalo pouze jedno přerušení. Vykonávání obsluhy odmaskované přerušení, bude přerušeno, nastane-li dalšího přerušení.

Na Obrázku 3.21 je vidět běžící vlákno A. Přerušení IRQx způsobí spuštění obsluhy přerušení Intx, které je přerušeno přerušením IRQy a jeho obsluhou Inty. Inty vrací událost, která způsobí spuštění vlákna B a Intx vrácí událost, která spouští vlákno C.



Obrázek 3.20: Vnořená přerušení

3.2.5.4 Implementace obsluhy přerušení

QNX dovoluje přiřadit každému přerušení více obsluh přerušení.

Tabulka 3.20: Funkce implementující interrupt handling API

Function	Description
InterruptAttach()	Připojí lokální funkci k vektoru přerušení.
InterruptAttachEvent()	Připojí událost k přerušení.
InterruptDetach()	Odpojí přerušení s ID, které vrátila fce *InterruptAttach()* nebo *InterruptAttachEvent()*.
InterruptWait()	Čeká na přerušení.
InterruptEnable()	Povolí hardwarové přerušení.
InterruptDisable()	Zakáže hardwarové přerušení.
InterruptMask()	Maskuje hardwarové přerušení.
InterruptUnmask()	Odmaskuje hardwarové přerušení.
InterruptLock()	Zamkne hardwarové přerušení (používá se u SMP systémů).
InterruptUnlock()	Odemkne hardwarové přerušení (používá se u SMP systémů).

4 Programy demonstrující vlastnosti QNX

Zdrojové kódy jsou psány v jazyce C. K jejich psaní jsem používal textový editor, který v grafickém prostředí nese název *ped*. Tento editor umožňuje snadný přechod ze psaní kódu jedné funkce na jinou funkci, protože si udržuje seznam definovaných funkcí. Požadovanou funkci stačí pouze vybrat v rozbalovacím menu ve spodní části editoru.

Pro překlad zdrojového kódu na spustitelný soubor jsem používal překladač *qcc*. Informace o parametrech příkazu *qcc* lze zjistit v shellu (terminál) příkazem *use qcc*. Programy se spouští z příkazové řádky shellu pomocí ./ a název spustitelného souboru.

4.1 Bariéry

Bariéry slouží jako synchronizační mechanismus. Pokud na bariéře čeká patřičný počet vláken (v mém případě 3), jsou všechna vlákna naráz uvolněna a mohou pokračovat ve své činnosti. Jako první začne čekat na bariéře hlavní vlákno, poté vlákno2 a jako třetí vlákno1, které způsobí jejich uvolnění. Pro kontrolu správné funkčnosti je v důležitých okamžicích spolu s komentářem vypsán také čas.

4.1.1 Zdrojový kód

```
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <sys/neutrino.h>
#include <unistd.h>

pthread_barrier_t      bariera;     // definice bariery

void *vlakno1(void *zadny)
{
   time_t cas;

   time(&cas);
   printf("vlakno1() zacalo v %s\n", ctime(&cas));
   sleep(20);                       // vlakno ceka 20 sekund
   time(&cas);
   printf("vlakno1() ceka na bariere od %s\n", ctime(&cas));
   pthread_barrier_wait(&bariera);  // cekani na bariere
   time(&cas);
   printf("vlakno1() pokracuje v cinnosti od %s\n", ctime(&cas));
}

void *vlakno2(void *zadny)
{
   time_t cas;

   time(&cas);
   printf("vlakno2() zacalo v %s\n", ctime(&cas));
   sleep(10);                       // vlakno ceka 10 sekund
   time(&cas);
   printf("vlakno2() ceka na bariere od %s\n", ctime(&cas));
   pthread_barrier_wait(&bariera);  // cekani na bariere
```

```
   delay(10);                        // vlakno ceka 10 ms
   time(&cas);
   printf("vlakno2() pokracuje v cinnosti od %s\n", ctime(&cas));
}

main()
{
   time_t cas;
   pthread_t IDv2;
   int    IDv1;

   // vytvoreni bariery
      pthread_barrier_init(&bariera, NULL, 3);

      // vytvoreni vlaken
   IDv1 = ThreadCreate(NULL, vlakno1, NULL, NULL);      // QNX
   pthread_create(&IDv2, NULL, vlakno2, NULL);          // POSIX

   time(&cas);
   printf("main() ceka na bariere od %s\n", ctime(&cas));
   pthread_barrier_wait(&bariera);      // cekani na bariere
   time(&cas);
   printf("main() pokracuje v cinnosti od %s\n", ctime(&cas));
   sleep(1);

   getchar();
   ThreadDestroy(IDv1, -1, NULL);       // zruseni vlaken
   ThreadDestroy(IDv2, -1, NULL);
}
```

4.2 Semafory a mutexy

Tento program demonstruje použití semaforu a mutexu. Každé vytvořené vlákno představuje voliče, který jde nejprve za plentu a poté k volební urně. Za každou plentou a u volební urny se může nacházet pouze jeden volič. Plenty jsou reprezentovány semaforem a volební urna mutexem.

4.2.1 Zdrojový kód

```
#include <stdio.h>
#include <sys/neutrino.h>
#include <semaphore.h>
#include <pthread.h>

sem_t        plenta;             // definice semaforu
pthread_mutex_t  urna;           // definice mutexu

// funkce volic predstavuje volice jdouciho za plentu a k urne
void *volic(int *n)
{
   int k = *n;
   while (1)
   {
      sem_wait(&plenta);         // dekrementace semaforu
```

```
        printf("Volic %d je za plentou.\n", k);
        sleep(k+3);                    // vlakno ceka k+3 sekund
        pthread_mutex_lock(&urna);     // zamknuti mutexu
        sem_post(&plenta);             // inkrementace semaforu
        printf("Volic %d dava obalku do urny.\n", k);
        sleep(k+1);
        pthread_mutex_unlock(&urna);  // odemknuti mutexu
    }
}

main()
{
    const  pocet_volicu = 5;       // definice poctu volicu
    const  pocet_plent = 3;        // definice poctu plent
    int IDv[pocet_volicu];         // definice identifikatoru vlaken
    int    i;
    int    *p_i;

    if (sem_init(&plenta,0,pocet_plent)== -1)  // vytvoreni semaforu
        printf("Nepodarilo se vytvorit semafor.\n");

    if (pthread_mutex_init(&urna, NULL) == -1) // vytvoreni mutexu
        printf("Nepodarilo se vytvorit mutex.\n");

    p_i = &i;
    for(i = 1; i <= pocet_volicu; i++)
        // vytvoreni vlaken
             IDv[i] = ThreadCreate(NULL, (void *)volic, p_i, NULL);

    getchar();
    for(i = 1; i <= pocet_volicu; i++)
        ThreadDestroy(IDv[i], -1, NULL);        // zruseni vlaken
    sem_destroy(&plenta);                       // zruseni semaforu
    pthread_mutex_destroy(&urna);               // zruseni mutexu
}
```

4.3 Pojmenované semafory

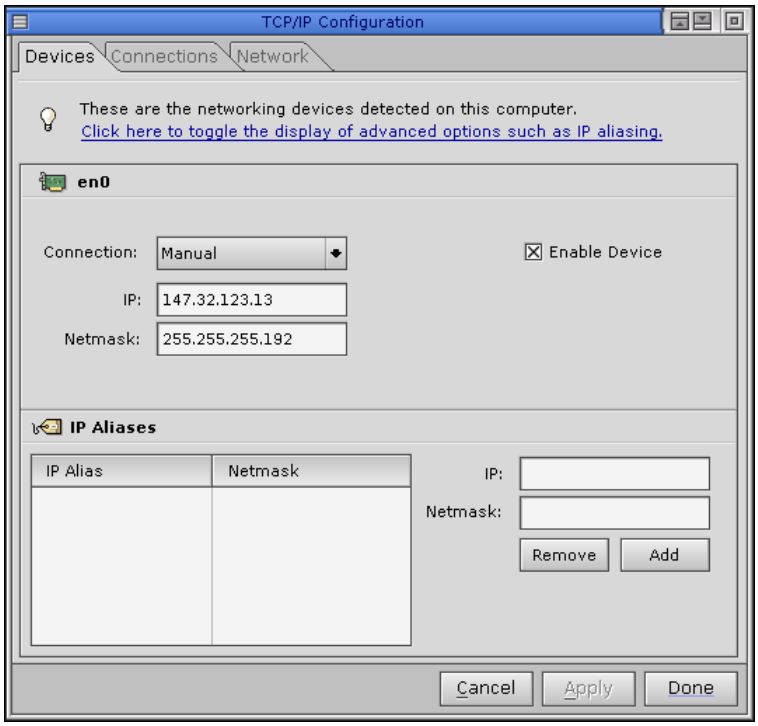
Tento příklad je podobný předchozímu příkladu. Zachována je myšlenka voliče, který jde za plentu a k volební urně. Ovšem místo normálního semaforu a mutexu jsou použity pojmenované semafory. Manipulace s pojmenovanými semafory probíhá pomocí nativního komunikačního protokolu zvaného *Qnet*.

Existují dva programy, jeden serverový a druhý klientský. Serverový program vytvoří na lokálním počítači pojmenované semafory a jeho funkce je shodná s předchozím programem. Klientský program na vzdáleném počítači se připojí k pojmenovaným semaforům na lokálním počítači a bude je využívat stejně jako serverový program. Celé si to můžeme přestavit jako dvě různé fronty voličů, kteří používají stejné plenty i stejnou volební urnu.

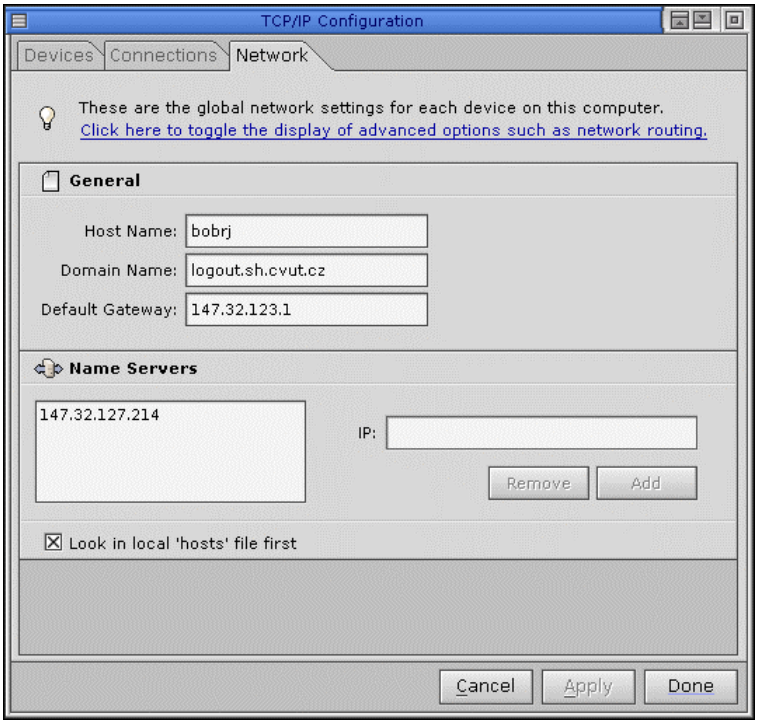
4.3.1 Konfigurace sítě

Konfiguraci sítě lze provést v grafickém prostředí Photon pomocí rozbalovacího menu v pravé části obrazovky nebo kliknout na tlačítko Launch v levém dolním rohu. V menu

Configure stačí kliknout na položku Network a zobrazí se konfigurační okno se třemi záložkami Devices, Connections a Network.

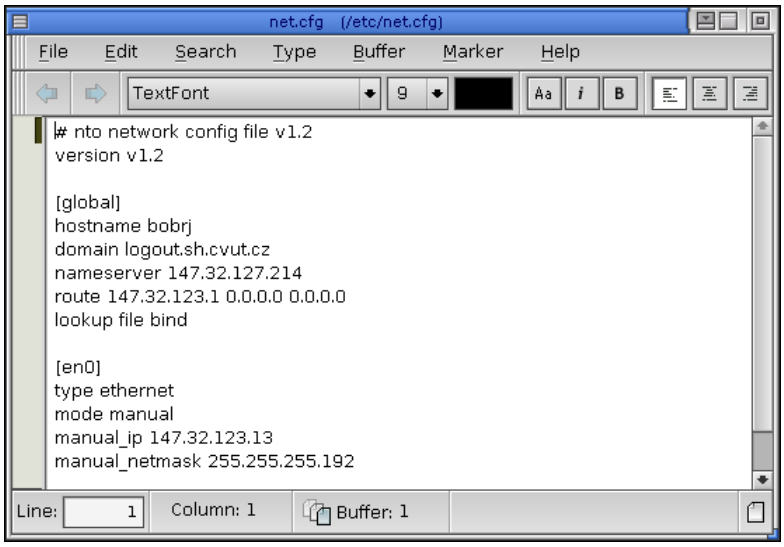


Obrázek 4.1: Konfigurace sítě – záložka Device



Obrázek 4.2: Konfigurace sítě – záložka Network

Konfiguraci sítě můžeme také provést v textovém souboru *net.cfg* v adresáři */etc/*.



Obrázek 4.3: Konfigurace sítě - textový soubor

4.3.2 Zdrojový kód

Serverový program:

```
#include <mqueue.h>
#include <stdio.h>
#include <sys/neutrino.h>
#include <semaphore.h>
#include <sys/stat.h>

sem_t  *sem_plenta, *sem_urna;    // ukazatele na semafory plenta a urna

// funkce volic predstavuje volice jdouciho za plentu a k urne
void *volic(int *n)
{
   int k = *n;
   while (1)
   {
      sem_wait(sem_plenta);      // dekrementace semaforu plenta
      delay(10);                 // hodnota udava dobu v [ms]
      printf("Volic %d je za plentou.\n", k);
      sleep(k+3);                // hodnota udava dobu v [s]
      sem_wait(sem_urna);        // dekrementace semaforu urna
      sem_post(sem_plenta);      // inkrementace semaforu plenta
      printf("Volic %d dava obalku do urny.\n", k);
      sleep(k+1);
      sem_post(sem_urna);        // inkrementace semaforu urna
   }
}

main()
{
```

```
   unsigned int pocet_volicu_s = 3;     // pocet serverovych volicu
   unsigned int pocet_plent = 2;        // pocet plent
   int    IDv[pocet_volicu_s];          // identifikatory vlaken
   int    i, *p_i;
   const char    *plenta = "/plenta";   // nazvy semaforu
      const char *urna = "/urna";

   // vytvoreni semaforu plenta
   sem_plenta = sem_open(plenta, O_CREAT, S_IRWXO, pocet_plent);
      printf("sem_plenta = %d\n", *sem_plenta);

   // vytvoreni semaforu urna
   sem_urna = sem_open(urna, O_CREAT, S_IRWXO, 1);
      printf("sem_urna = %d\n", *sem_urna);

   p_i = &i;
   // vytvoreni vlaken
   for(i = 1; i <= pocet_volicu_s; i++)
      IDv[i] = ThreadCreate(NULL, (void *)volic, p_i, NULL);

   getchar();
   for(i = 1; i <= pocet_volicu_s; i++)
      ThreadDestroy(IDv[i],-1,NULL);   // zruseni vlaken
   sem_close(sem_plenta);        // zavreni pojmenovanych semaforu
   sem_close(sem_urna);
   sem_unlink(plenta);           // zruseni   pojmenovanych semaforu
   sem_unlink(urna);
}
```

Klientský program:

```
#include <mqueue.h>
#include <stdio.h>
#include <sys/neutrino.h>
#include <semaphore.h>
#include <sys/stat.h>

sem_t  *sem_plenta, *sem_urna;    // ukazatele na semafory planta a urna

// funkce volic predstavuje volice jdouciho za plentu a k urne
void *volic(int *n)
{
   int k = *n;
   while (1)
   {
      sem_wait(sem_plenta);      // dekrementace semaforu plenta
      delay(10);                 // hodnota udava dobu v [ms]
      printf("Volic %d je za plentou.\n", k);
      sleep(k+3);                // hodnota udava dobu v [s]
      sem_wait(sem_urna);        // dekrementace semaforu urna
      sem_post(sem_plenta);      // inkrementace semaforu plenta
      printf("Volic %d dava obalku do urny.\n", k);
      sleep(k+1);
      sem_post(sem_urna);        // inkrementace semaforu urna
   }
}
```

```
main()
{
   unsigned int pocet_volicu_s = 3;     // pocet serverovych volicu
   unsigned int pocet_volicu_c = 3;     // pocet klientskych volicu
   int    IDv[pocet_volicu_c];          // identifikatory vlaken
   int    i, *p_i;
   // definice cest ke vzdalenym semaforum
   const char   *plenta = "/net/UNKNOWN_.felk.cvut.cz/dev/sem/plenta";
   const char *urna = "/net/UNKNOWN_.felk.cvut.cz/dev/sem/urna";

      // pripojeni ke vzdalenemu semaforu plenta
   sem_plenta = sem_open(plenta, O_EXCL);
      printf("sem_plenta = %d\n", *sem_plenta);

   // pripojeni ke vzdalenemu semaforu urna
   sem_urna = sem_open(urna, O_EXCL);
      printf("sem_urna = %d\n", *sem_urna);

   p_i = &i;
   // vytvoreni vlaken
      for(i = (pocet_volicu_s + 1); i <= (pocet_volicu_s +
   pocet_volicu_c); i++)
      IDv[i] = ThreadCreate(NULL, (void *)volic, p_i, NULL);
   getchar();
   // uvolnění prostredku a zruseni vlaken
   for(i = (pocet_volicu_s + 1); i <= (pocet_volicu_s +
   pocet_volicu_c); i++) {
      ThreadDetach(IDv[i]);
      ThreadDestroy(IDv[i],-1,NULL);
   }
}
```

4.4 Fronty zpráv

Následující příklad ukazuje použití fronty zpráv. Všechny zobrazované informace jsou nejprve zaslány do fronty a poté pomocí funkce `Vypis` čteny z fronty a vypisovány na obrazovce. Funkce `Makacenko` představuje dělníka, který vezme lopatu, určitou dobu pracuje, vrací lopatu, určitou dobu odpočívá, atd. Lopaty jsou reprezentovány semaforem. Každý dělník vlastní pracovní povolení shodné s identifikátorem vlákna, které představuje dělníka.

4.4.1 Zdrojový kód

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <mqueue.h>

#define   POCET_LOPAT         2
#define   POCET_MAKACENKU     3
#define   POCET_POLOZEK    6
```

```
typedef struct {
   int    Delnik;
   char   Zprava[80];
} Polozka;

int    makacka[POCET_MAKACENKU]   =  {5000, 3000, 4000};
int    leharos[POCET_MAKACENKU]   =  {1000, 2000, 1500};

pthread_t    seznam_prac_povoleni[POCET_MAKACENKU] = {1,2,3};

sem_t  Semafor;                // definice semaforu
mqd_t  Fronta;                 // definice fronty
unsigned int msg_prio = 10;

char
   *makacenkove[POCET_MAKACENKU+1]={"Jarda","Pepa","Franta","Mistr"};

Polozka      Hlaseni;

void   *Makacenko (void *nepouzity)
{
   pthread_t    pracovni_povoleni;
   int       index, vysledek;
   Polozka      S_Hlaseni;

   pracovni_povoleni = pthread_self();  // identifikator vlakna
   delay(10);
   for (index = 0; index < POCET_MAKACENKU; index++)
   {
      if (pracovni_povoleni == seznam_prac_povoleni[index]) {
         break;
      }
   }

   S_Hlaseni.Delnik = POCET_MAKACENKU;
   sprintf(S_Hlaseni.Zprava, "A, pan makacenko %s racil prijit do
      prace!\n", makacenkove[index] );
   // odeslani zprávy
   mq_send(Fronta, (char *)&S_Hlaseni, sizeof(Polozka), msg_prio);

   S_Hlaseni.Delnik = index;
   while (1)
   {
      // zkousi dekrementovat semafor (neblokujici)
      vysledek = sem_trywait(&Semafor);
      if (vysledek != 0) {          // pokud nejde dekrementovat
         sprintf(S_Hlaseni.Zprava, "To je pohodicka, neni s cim
            delat.\n");
         mq_send(Fronta, (char *)&S_Hlaseni, sizeof(Polozka),
            msg_prio);

         sem_wait(&Semafor);        // dekrementuje semafor
      }
      sprintf(S_Hlaseni.Zprava, "Grr, makam jako cernej. To je
         drina.\n");
```

```
        mq_send(Fronta, (char *)&S_Hlaseni, sizeof(Polozka),
           msg_prio);
        delay(makacka[index]);

        sprintf(S_Hlaseni.Zprava, "Hura, padla!\n");
        mq_send(Fronta, (char *)&S_Hlaseni, sizeof(Polozka),
           msg_prio);

        sem_post(&Semafor);        // inkrementuje semafor

        sprintf(S_Hlaseni.Zprava, "Ted si dam pauzicku.\n");
        mq_send(Fronta, (char *)&S_Hlaseni, sizeof(Polozka),
           msg_prio);

        delay(leharos[index]);
    }
}

void *Vypis(void *nepouzity)
{
    unsigned int msg_prio;
    Polozka      R_Hlaseni;
    while(1)
    {
        mq_receive(Fronta, (char *)&R_Hlaseni, sizeof(Polozka),
           &msg_prio);                    // precteni zpravy
        printf("%s: ", makacenkove[R_Hlaseni.Delnik]);
        printf(R_Hlaseni.Zprava);
    }
}


main()
{
    int       index;
    pthread_t    vypisID;
    const char       *nazev_fronty = "/Vypisy";
    struct mq_attr   param_fronty;
    mode_t    mode;

    param_fronty.mq_maxmsg = POCET_POLOZEK;    // parametry fronty
    param_fronty.mq_msgsize = sizeof(Polozka);
    if ((Fronta = mq_open(nazev_fronty, O_RDWR | O_CREAT, mode,
        &param_fronty)) == -1){    // vytvoreni fronty

        printf("Nepodarilo se vytvorit FRONTU ZPRAV!\n");
        exit();
    }

    // vytvoreni vlakna fce Vypis
    pthread_create(&vypisID, NULL, Vypis, NULL);

    // vytvoreni semaforu
    if (sem_init(&Semafor, 0, POCET_LOPAT) == -1) {
        Hlaseni.Delnik = POCET_MAKACENKU;
        sprintf(Hlaseni.Zprava, "Hura! Chlapi, neprivezli lopaty!\n");
        mq_send(Fronta, (char *)&Hlaseni, sizeof(Polozka), msg_prio);
```

```
        delay(10);
        pthread_detach(vypisID);      // zruseni vlakna fce Vypis
        mq_close(Fronta);             // zavreni fronty
        mq_unlink(nazev_fronty);      // zruseni fronty
        exit();
    }

    for (index = 0; index < POCET_MAKACENKU; index++) {
       pthread_create(&(seznam_prac_povoleni[index]),   NULL,   Makacenko,
              NULL);                     //    vytvoreni    vlaken    fce
       Makacenko
    }

    getchar();
    for (index = 0; index < POCET_MAKACENKU; index++) {
       // zruseni vlaken fce Makacenko
       pthread_detach(seznam_prac_povoleni[index]);
       Hlaseni.Delnik = index;
       sprintf(Hlaseni.Zprava, "Kaslu na to, jdu dom.\n");
       mq_send(Fronta, (char *)&Hlaseni, sizeof(Polozka), msg_prio);
    }
    sem_destroy(&Semafor);            // zruseni semaforu
    Hlaseni.Delnik = POCET_MAKACENKU;
    sprintf(Hlaseni.Zprava, "Hm, chlapi se na to vybodli, tak odvezte
       ty lopaty. Dneska koncime.\n");
    mq_send(Fronta, (char *)&Hlaseni, sizeof(Polozka), msg_prio);
    delay(10);
    pthread_detach(vypisID);          // zruseni vlakna fce Vypis
    mq_unlink(nazev_fronty);          // zruseni fronty
}
```

5 Řízení otáček stejnosměrného motorku

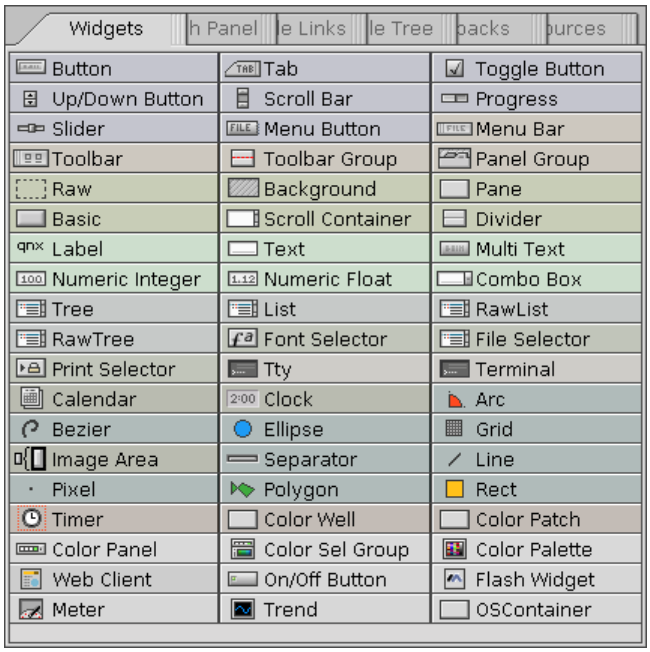
Cílem je řídit otáčky stejnosměrného motorku pomocí pulsně šířkové modulace. Jedná se o laboratorní přípravek, který se připojuje k počítači pomocí paralelního portu. Otáčení je snímáno inkrementálním čidlem. Pokud se změní stav signálů inkrementálního čidla, dojde k vyvolání přerušení na paralelním portu (IRQ 7). Během jedné otáčky motorku je přerušení vyvoláno 16-krát.

5.1 Vývojové prostředí

Řídicí aplikaci jsem vytvářel v objektově orientovaném prostředí Photon Application Builderu (PhAB). Spustíme ho kliknutím na tlačítko Launch v levé dolní části obrazovky a v rozbalovacím menu Development vybráním položky Builder.

5.1.1 Objekty PhAB

PhAB je vývojový nástroj s několika předdefinovanými okny a dostatečným množstvím objektů zvaných Widgets. Každý objekt nese název třídy do které patří, pokud chceme s objektem pracovat musíme mu přiřadit vlastní název. Tento objekt je pak dostupný díky identifikátoru ABW_název objektu.

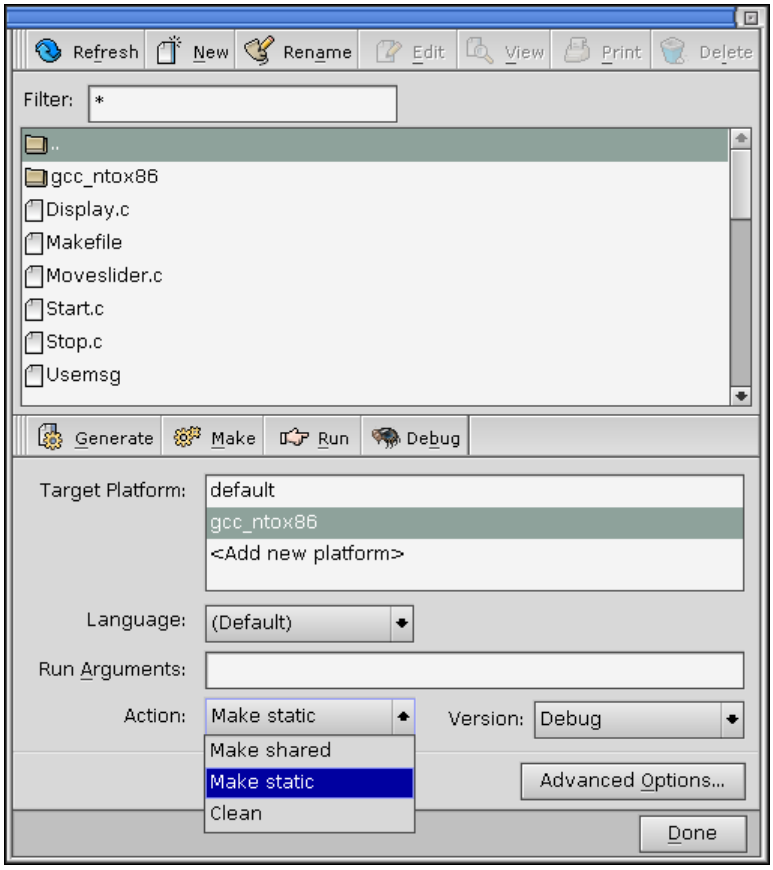


Obrázek 5.1: Předdefinované objekty

5.1.2 Překlad kódu

Okno pro překlad vytvářené aplikace vyvoláme v hlavním menu PhAB kliknutím na Application a vybráním položky Build + Run … nebo stiskem funkční klávesy F5. Nejprve kliknutím na tlačítko Generate vygenerujeme kód hlavního okna Abmain.c a několik dalších souborů, jako např. hlavičkového souboru Abdefine.h definujícího identifikátory pojmenovaných objektů, atd. Poté vybereme platformu, na které bude

aplikace provozována. Kliknutím na tlačítko Make se vykoná překlad jednotlivých callbackových funkcí, přilinkování knihoven (sdílené nebo statické) a vytvoření spustitelného souboru (normální verze nebo verze pro ladění).



Obrázek 5.2: Kompilace vytvářené aplikace

V prostředí PhAB je možné vytvořit hlavičkový soubor pro definici globálních proměnných, např. globals.h. Tento soubor je automaticky vkládán do nově vytvářených callbackových funkcí.

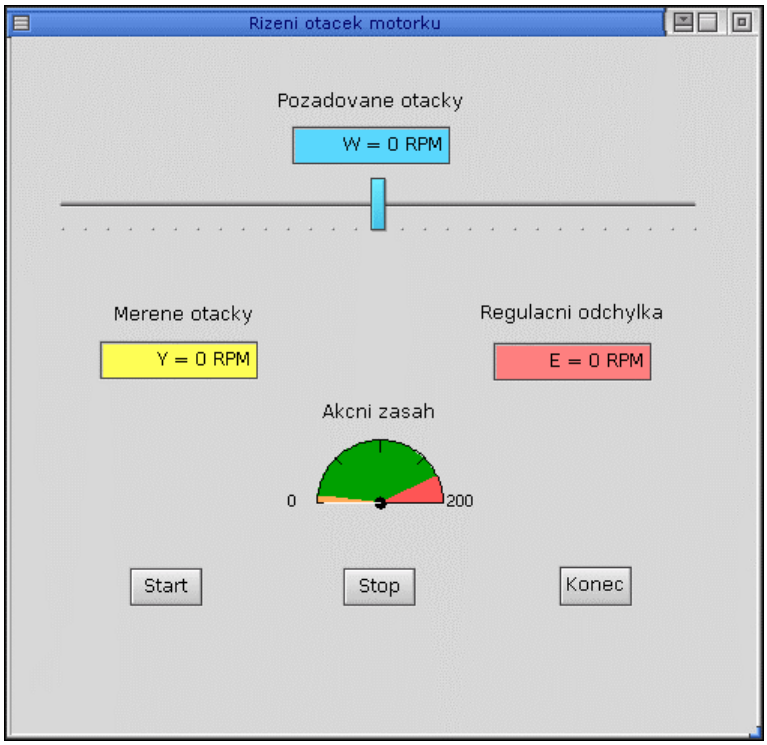
5.2 Vlastní implementace

5.2.1 Popis aplikace

Aplikaci lze spustit buďto přímo z prostředí PhAB v okně pro překlad, kliknutím na tlačítko Run (viz. Obrázek 5.2), nebo z příkazové řádky shellu ./ a název spustitelného souboru.

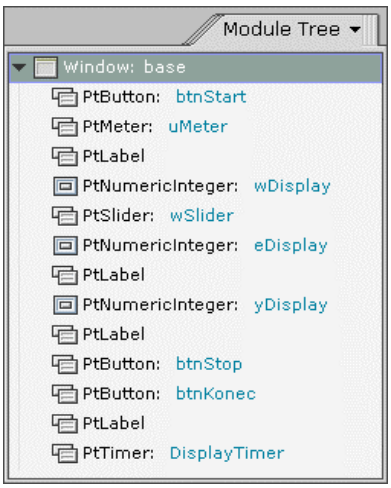
Pohybem jezdce v horní části aplikace měníme požadované otáčky motorku v obou směrech otáčení. Stiskem tlačítka Start se spustí vlastní regulace otáček motorku. Stiskem tlačítka Stop dojde k ukončení regulace a zastavení motorku. Požadované otáčky, měřené otáčky a regulační odchylky jsou zobrazovány v otáčkách za minutu (RPM). Akční zásah zobrazuje délku aktivní části periody PWM v milisekundách. Stiskem tlačítka Konec ukončíme aplikaci.

Na Obrázku 5.3 je vidět vlastní řídicí aplikace stejnosměrného motorku.

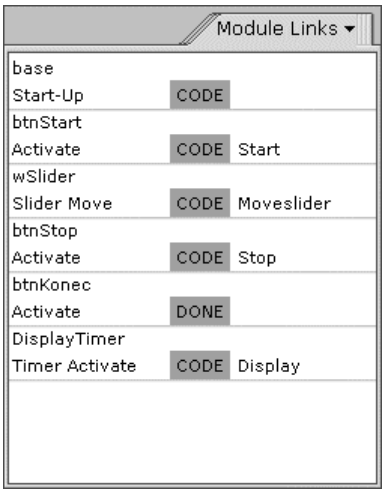


Obrázek 5.3: Řídicí aplikace

Na Obrázku 5.4 jsou uvedeny objekty a jejich názvy použité v aplikaci.



Obrázek 5.4: Použité objekty

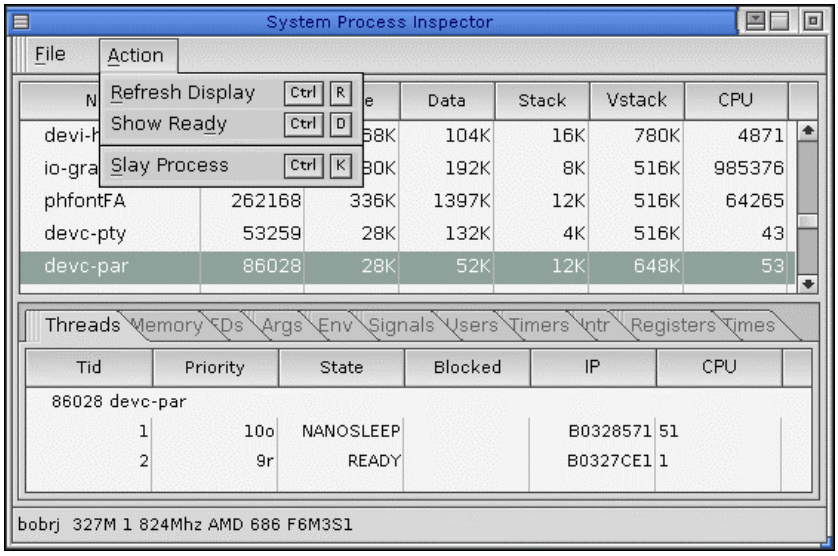


Obrázek 5.5: Callbackové funkce

Obrázek 5.5 ukazuje přiřazení callbackových funkcí k patřičným akcím prováděných na objektech.

5.2.2 Regulační obvod

Před spuštěním vlastní regulace otáček motorku je nutné zrušit ovladač paralelního portu, který zhruba každých 30 sekund nuluje 5. bit (pokud uvažujeme bity 1 až 8) řídicího portu 37Ah, který povoluje přerušení. V Shellu stačí napsat příkaz *slay devc-par* nebo kliknout na tlačítko Launch v levém dolním rohu obrazovky a v rozbalovacím menu Utilities vybrat položku System Information. Ve spuštěné aplikaci vybereme ze seznamu procesů proces, který chceme zrušit (v našem případě devc-par).

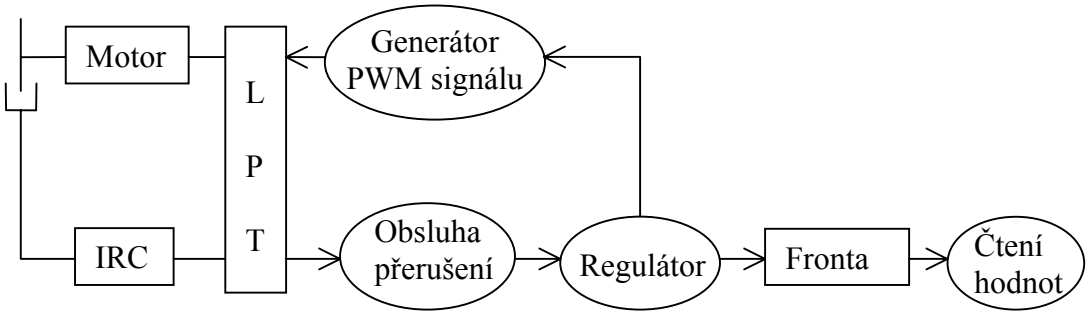


Obrázek 5.6: Zrušení ovladače paralelního portu

Regulační obvod se skládá ze tří vláken a sice obsluhy přerušení, regulátoru a generátoru PWM signálu. Pokud nastane přerušení na paralelním portu, obsluha přerušení inkrementuje patřičnou proměnnou.

Regulátor obsahuje periodicky spouštěný časovač, který po svém vypršení pošle událost typu puls. Po obdržení této události se spočítají aktuální otáčky, regulační odchylka a akční zásah. Tyto hodnoty jsou poslány do fronty zpráv. Z fronty zpráv jsou hodnoty čteny pomocí dalšího samostatně běžícího vlákna. V grafické aplikaci běží také periodicky spouštěný časovač, který při svém vypršení spouští callbackovou funkci, která zobrazuje hodnoty přečtené z fronty zpráv.

Generátor PWM signálu mění šířku aktivní části periody signálu podle velikosti akčního zásahu.



Obrázek 5.7: Regulační obvod

5.2.3 Zdrojové kódy

Globální hlavičkový soubor globals.h:

```
/* Header "globals" for engine Application */

#include <Pt.h>
#include <sys/neutrino.h>
#include <pthread.h>
#include <hw/inout.h>
#include <sys/netmgr.h>
#include <time.h>
#include <errno.h>
#include <mqueue.h>

#define   byte   unsigned char

/* definice struktury zobrazovanych dat */
typedef struct {
   int    yValue;
   int    eValue;
   int    uValue;
} Polozka;

/* global variables */
extern int       regulace_id;
extern int       InterruptAttach_id;
extern int       W;
extern int       int_thread_id;
extern int       pwm_id;
extern int       Vypis_id;
extern mqd_t     Fronta;
extern Polozka   R_Polozka;
```

Callback funkce Moveslider.c:

```
/*************************************************/
/* Callback funkce Moveslider od slideru         */
/*                     AppBuilder Photon Code Lib */
/*                         Version 2.01           */
/*************************************************/

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Local headers */
#include "ablibs.h"
#include "abimport.h"
#include "proto.h"
#include "globals.h"

int
Moveslider(  PtWidget_t  *widget,  ApInfo_t  *apinfo,  PtCallbackInfo_t
   *cbinfo )
{
```

```
    int    *Value;

    /* eliminate 'unreferenced' warnings */
    widget = widget, apinfo = apinfo, cbinfo = cbinfo;

    /* zobrazeni a nastaveni zadane hodnoty */
    PtGetResource(widget, Pt_ARG_GAUGE_VALUE, &Value, 0);
    PtSetResource(ABW_wDisplay, Pt_ARG_NUMERIC_VALUE, *Value, 0);
    W = *Value;

    return( Pt_CONTINUE );
}
```

Callback funkce Start.c:

```
/*************************************************/
/* Callback funkce Start od tlacitka Start       */
/*                    AppBuilder Photon Code Lib */
/*                        Version 2.01           */
/*************************************************/

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Local headers */
#include "ablibs.h"
#include "abimport.h"
#include "proto.h"
#include "globals.h"

#define   IRQ7         7         // preruseni od paralelniho portu
#define   period    200          // [msec.]
#define   MY_PULSE_CODE_PULSE_CODE_MINAVAIL
#define   interval_mereni  0.2   // [sec.]
#define   Kfast        0.3       /* konstanty regulatoru */
#define   Kslow        0.2
#define   MAX_RPM      6862.5    /* maximalni otacky motoru */

int       regulace_id;
int       InterruptAttach_id;
int       W;                     // zadana hodnota
volatile unsigned   count;
struct sigevent  event_int;      // udalost preruseni
unsigned     active;             // aktivni cast periody
unsigned     pocet_preruseni;
byte         Dir;                // smer otaceni
int       int_thread_id;
int       pwm_id;
int       Vypis_id;
mqd_t        Fronta;
Polozka      R_Polozka;          // definovano v globals.h

/*******************************************************************/
/************** Obsluha preruseni od paralelniho portu ***************/
```

```
/*********************************************************************/
const struct sigevent *isr_handler(void *arg, int id)
{
   InterruptMask(IRQ7, InterruptAttach_id);
   if (++count == 1)
   {
         count = 0;
         return(&event_int);
   }
   else
   {
         InterruptUnmask(IRQ7, InterruptAttach_id);
         return(NULL);
   }
}

void *int_thread(void *arg)
{
   byte      port, value;

   ThreadCtl(_NTO_TCTL_IO, 0);   // privilegia pro praci s porty

   value = in8(0x37A); // cteni ridiciho portu 37Ah
   value &= 0x0DF;     // nastaveni vyst. port 378h pro zapis
   value |= 0x10;      // povoleni preruseni
   out8(0x37A, value); // zapis hodnoty na port 37Ah

   InterruptAttach_id = InterruptAttach(IRQ7,&isr_handler,NULL,0,0);
   setprio(0, 22);     // nastaveni priority
   count = 0;
   pocet_preruseni = 0;
   while(1)
   {
      InterruptWait(0, NULL);    // ceka na udalost preruseni
      ++pocet_preruseni;

      InterruptUnmask(IRQ7, InterruptAttach_id);
   }
}
/*********************************************************************/

/*********************************************************************/
/******************* Generovani PWM signalu **************************/
/*********************************************************************/
void *pwm(void *arg)
{
   unsigned     T_on;

   ThreadCtl(_NTO_TCTL_IO, 0);
   setprio(0, 20);
   Dir = 0x00;
   while(1)
   {
      T_on = active;
      out8(0x378, Dir);
      delay(T_on);
      out8(0x378, 0x00);
```

```
        delay(period - T_on);
    }
}
/*********************************************************************/

/*********************************************************************/
/* Fce zajistujici vypocet otacek motorku, odchylky a akcniho zasahu */
/*********************************************************************/
void *regulace(void *arg)
{
    struct sigevent  event;       // udalost typu puls
    struct itimerspec   itime;
    timer_t      timer_id;
    int       chid, rcvid;
    struct _pulsepulse;
    float        E, U, otacky;
    Polozka      S_Polozka;

    setprio(0, 21);
    chid = ChannelCreate(0);      // vytvori komunikacni kanal

    /* nastaveni udalosti typu puls */
       event.sigev_notify = SIGEV_PULSE;
    event.sigev_coid = ConnectAttach(ND_LOCAL_NODE, 0, chid,
    _NTO_SIDE_CHANNEL, 0);
    event.sigev_priority = getprio(0);
    event.sigev_code = MY_PULSE_CODE;

    /* vytvoreni casovace - po vyprseni posle puls */
    timer_create(CLOCK_REALTIME, &event, &timer_id);

    /* nastaveni doby prvniho vyprseni casovace */
    itime.it_value.tv_sec = 0;
    itime.it_value.tv_nsec = interval_mereni*1000000000;
    /* nastaveni doby opakovaneho vyprseni casovace */
    itime.it_interval.tv_sec = 0;
    itime.it_interval.tv_nsec = interval_mereni*1000000000;
    timer_settime(timer_id, 0, &itime, NULL);

    U = 0;
    pocet_preruseni = 0;

    while (1) {
        rcvid = MsgReceive(chid, &pulse, sizeof(pulse), NULL);
        if (rcvid == 0) { /* prisel puls od casovace */
            if (pulse.code == MY_PULSE_CODE) {
                /* vypocet otacek motorku*/
                otacky = (60./16.)*pocet_preruseni/interval_mereni;
                pocet_preruseni = 0;

                /* smer otaceni motoru */
                if (W < 0) {
                    Dir = 0x01;            // zaporne otacky
                    otacky = -otacky;
                    }
                else   {
                    Dir = 0x02;            // kladne otacky
```

```
            };

            /* odstraneni cukani motorku pri nulove zadane
         hodnote (delay(0) nestaci) */
            if (W == 0) Dir = 0x00;

            /* vypocet regulacni odchylky */
            E = W - otacky;
            if (W < 0) E = -E;

            /* vypocet teoretickeho akcniho zasahu */
            if (abs(W) < 600) {
                            U += Kslow * E * period / MAX_RPM;
                        }
            else {
                            U += Kfast * E * period / MAX_RPM;
                        };
            if (W < 0) E = -E;

                        /*  skutecny  akcni  zasah  (osetreni  max.  a
                          zaporneho akcniho zasahu) */
            if (abs(U) > period) {
               active = period;
               U = period;
            }
            else {
               active = abs(U);
               //U = abs(U);
            };
            /* poslani zobrazovanych hodnot do fronty */
            S_Polozka.yValue = (int) otacky;
            S_Polozka.eValue = (int) E;
            S_Polozka.uValue = active;
            mq_send(Fronta, (char *)&S_Polozka, sizeof(Polozka),
NULL);
         }
      }
   }
}
/*******************************************************************/

/*******************************************************************/
/***************** Cteni zobrazovanych dat z fronty ******************/
/*******************************************************************/
void *Vypis(void *arg)
{
   unsigned     msg_prio;

   while(1) {
      mq_receive(Fronta,              (char              *)&R_Polozka,
sizeof(Polozka),&msg_prio);
   }
}
/*******************************************************************/


/*******************************************************************/
```

```
/******************* Callback funkce Start **************************/
/********************************************************************/
int
Start( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
   long         *btnStart_F, *btnStop_F;
   const char       *nazev_fronty = "/Vypis";
   mode_t    mode;
   struct mq_attr   param_fronty;
   int       errvalue;

   /* eliminate 'unreferenced' warnings */
   widget = widget, apinfo = apinfo, cbinfo = cbinfo;

   /* nastaveni parametru fronty */
   param_fronty.mq_maxmsg = 10;
   param_fronty.mq_msgsize = sizeof(Polozka);
   errno = EOK;
   /* vytvoreni fronty zprav */
   if ((Fronta = mq_open(nazev_fronty, O_RDWR | O_CREAT, mode,
      &param_fronty)) == -1) {
      errvalue = errno;
      printf( "The error generated was %d\n", errvalue );
      printf( "That means: %s\n", strerror( errvalue ) );
      printf("Nepodarilo se vytvorit FRONTU ZPRAV!\n");
      exit(EXIT_FAILURE);
   }
   /* inicializace udalosti preruseni */
   SIGEV_INTR_INIT(&event_int);

   /* vytvoreni vlaken */
   Vypis_id = ThreadCreate(NULL, Vypis, NULL, NULL);
   int_thread_id = ThreadCreate(NULL, int_thread, NULL, NULL);
   pwm_id = ThreadCreate(NULL, pwm, NULL, NULL);
   regulace_id = ThreadCreate(NULL, regulace, NULL, NULL);

   /* zmena atributu tlacitka Start */
   PtGetResource(widget, Pt_ARG_FLAGS, &btnStart_F, 0);
   *btnStart_F = *btnStart_F | Pt_BLOCKED & ~Pt_SELECTABLE;
   PtSetResource(widget, Pt_ARG_FLAGS, *btnStart_F, 0);

   /* zmena atributu tlacitka Stop */
   PtGetResource(ABW_btnStop, Pt_ARG_FLAGS, &btnStop_F, 0);
   *btnStop_F = *btnStop_F & ~Pt_BLOCKED | Pt_SELECTABLE;
   PtSetResource(ABW_btnStop, Pt_ARG_FLAGS, *btnStop_F, 0);

   return( Pt_CONTINUE );
}
/********************************************************************/
```

Callback funkce Stop.c:

```
/*************************************************/
/* Callback funkce Stop od tlacitka Stop         */
/*                    AppBuilder Photon Code Lib */
/*                        Version 2.01           */
```

```
/***********************************************/

/* Standard headers */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/* Local headers */
#include "ablibs.h"
#include "abimport.h"
#include "proto.h"
#include "globals.h"

int
Stop( PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo )
{
   long      *btnStart_F, *btnStop_F;

   /* eliminate 'unreferenced' warnings */
   widget = widget, apinfo = apinfo, cbinfo = cbinfo;

   /* zruseni vlaken */
   ThreadDetach(regulace_id);
   ThreadDestroy(regulace_id, -1, NULL);

   InterruptDetach(InterruptAttach_id);
   ThreadDetach(int_thread_id);
   ThreadDestroy(int_thread_id, -1, NULL);

   ThreadDetach(pwm_id);
   ThreadDestroy(pwm_id, -1, NULL);

   ThreadDetach(Vypis_id);
   ThreadDestroy(Vypis_id, -1, NULL);
   mq_close(Fronta);

   /* zastaveni motorku */
   ThreadCtl(_NTO_TCTL_IO, 0);
   out8(0x378, 0x00);

   /* zmena atributu tlacitka Stop */
   PtGetResource(widget, Pt_ARG_FLAGS, &btnStop_F, 0);
   *btnStop_F = *btnStop_F | Pt_BLOCKED & ~Pt_SELECTABLE;
   PtSetResource(widget, Pt_ARG_FLAGS, *btnStop_F, 0);

   /* zmena atributu tlacitka Start */
   PtGetResource(ABW_btnStart, Pt_ARG_FLAGS, &btnStart_F, 0);
   *btnStart_F = *btnStart_F & ~Pt_BLOCKED | Pt_SELECTABLE;
   PtSetResource(ABW_btnStart, Pt_ARG_FLAGS, *btnStart_F, 0);

   /* vynulovani zobrazovanych hodnot */
   R_Polozka.yValue = 0;
   R_Polozka.eValue = 0;
   R_Polozka.uValue = 0;

   return( Pt_CONTINUE );
```

```
}
```

Callback funkce Display.c:

```
/*****************************************************/
/* Callback funkce Display od cacovace DisplayTimer */
/*                       AppBuilder Photon Code Lib */
/*                           Version 2.01           */
/*****************************************************/

/* Local headers */
#include "ablibs.h"
#include "globals.h"
#include "abimport.h"
#include "proto.h"

int
Display(PtWidget_t *widget, ApInfo_t *apinfo, PtCallbackInfo_t *cbinfo)
{
   /* eliminate 'unreferenced' warnings */
   widget = widget, apinfo = apinfo, cbinfo = cbinfo;

   /*zobrazeni otacek motorku, regulacni odchylky a akcniho zasahu*/
   PtSetResource(ABW_yDisplay, Pt_ARG_NUMERIC_VALUE,
   R_Polozka.yValue, 0);
   PtSetResource(ABW_eDisplay, Pt_ARG_NUMERIC_VALUE,
   R_Polozka.eValue, 0);
   PtSetResource(ABW_uMeter, Pt_ARG_METER_NEEDLE_POSITION,
   R_Polozka.uValue, 0);

   return( Pt_CONTINUE );
}
```

6 Závěr

Předložená diplomová práce se již na začátku zabývá vhodností použití operačního systému QNX v realtimových aplikacích. Ze studie [4] vyplívá, že OS QNX je vhodný pro řízení v reálném čase. Dále se zabývá strukturou operačního systému, implementací mikrojádra a podrobněji popisuje služby mikrojádra. Vlastní vypracování se skládá ze dvou hlavních částí.

V první části jsou uvedeny zdrojové kódy, které demonstrují použití POSIXových frontových zpráv a některých synchronizačních služeb. Jedná se o bariéry, semafory, mutexy a pojmenované semafory, které používají nativní komunikační protokol qnet. U pojmenovaných semaforů jsou také ukázány dva možné způsoby konfigurace sítě.

Ve druhé části jsem řešil úlohu řízení otáček stejnosměrného laboratorního motorku pomocí pulsně šířkově modulovaného signálu. Otáčení motorku bylo snímáno inkrementálním čidlem. Při změně stavu IRC čidla se generovalo přerušení na paralelním portu. Za jednu otáčku motorku se vygeneruje 16 přerušení, takže se nejedná o příliš přesné měření otáček, ale pro školní účely postačuje.

První problém, který nastal, bylo právě přerušení od paralelního portu. Po určité době spuštění zkušebního programu se mi nedařilo zachytit žádné další přerušení. Z počátku jsem nevěděl, kde mám hledat příčinu problému. Zkusil jsem tedy přerušení od časovače, ale tam tento problém nenastal. Pak jsem testoval zdali je stále povoleno přerušení od paralelního portu a zjistil jsem, že po určité době dojde k jeho zakázání. Teď stačilo přijít na to, co zakazuje přerušení. Byl to ovladač paralelního portu devc-par, který periodicky zakazoval přerušení. Po jeho zrušení fungovalo vše bez problémů.

Poté se vyskytly ještě další větší či menší problémy ve vývojovém prostředí Photon Application Builderu, které jsem zdárně vyřešil a zachytil jsem je v textu pro případné začínající uživatele PhAB.

Literatura

[1] *QNX* [online]. <http://www.qnx.com>.

[2] *Navrhování systémů s výpočetní technikou* [online]. <http://dce.felk.cvut.cz/nst>.

[3] *Počítače pro řízení* [online]. < http://dce.felk.cvut.cz/por>.

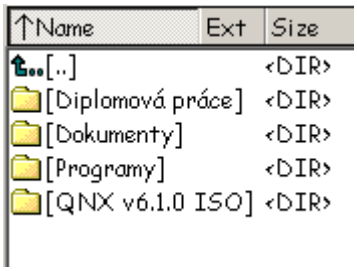
[4] Dedicated Systems Experts. *QNX RTOS v6.1 Evaluation Report*. September 2001, Doc no.: DSE-RTOS-EVA-012. <http://www.dedicated-systems.com>.

[5] Ocera. *Real-Time Operating Systems Analysis*. August 2002. <D1.pdf>.

[6] Herout P. *Učebnice jazyka C*. 3. vyd. České Budějovice: KOPP, 1999. ISBN 80-85828-21-9.

Příloha A

Obsah přiloženého CD ROMu



Obrázek A.1: Struktura přiloženého CD ROMu

Jednotlivé adresáře obsahují:

Diplomová práce – Tato diplomová práce ve formátu .doc a .rtf

Dokumenty – Materiály použité k vypracování diplomové práce

Programy – Demonstrační programy a aplikace, která řídí otáčky motorku

QNX v6.1.0 ISO – Bootovací ISO soubor s RTOS QNX v6.1.0