Univerzita Pardubice

Fakulta ekonomicko-správní

Modelování činnosti vláken v operačním systému

Bc. Martin Ibl

Diplomová práce

2011

Univerzita Pardubice
Fakulta ekonomicko-správní
Akademický rok: **2010/2011**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Martin IBL**
Osobní číslo:	**E09832**
Studijní program:	**N6209 Systémové inženýrství a informatika**
Studijní obor:	**Informatika ve veřejné správě**
Název tématu:	**Modelování činnosti vláken v operačním systému**
Zadávající katedra:	**Ústav systémového inženýrství a informatiky**

Zásady pro vypracování:

Osvojení Petri sítě jako modelovacího nástroje.
Popis činnosti vybraných vláken v multiprocesním operačním systému.
Popis činnosti vybraných vláken ve vícejádrovém procesu.
Modelování této činnosti pomocí Petri sítě.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] BRESHEARS, Clay. The Art of Concurrency. 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O?Reilly Media, Inc., 2009. 303 s. ISBN [3] 978-0-596-52153-0.

[2] HAAS, Peter J. Stochastic Petri Nets. New York : Springer-Verlag New York, Inc., 2002. 529 s. ISBN 0-387-95445-7.

[3] MARKL, Jaroslav. Petriho sítě [online]. Ostrava : VŠB - Technická univerzita Ostrava , 1998 [cit. 2010-06-24]. Dostupné z WWW: <http://www.cs.vsb.cz/markl/pn/>.

[4] SILBERSCHATZ, Abraham ; GALVIN, Peter Baer; GAGNE, Greg. Operating systems concepts. John Wiley, 2003. 978 s.

[5] STALLINGS, William. Operational Systems. Prentice Hall, 2005. 822 s.

[6] TANENBAUM, Andrew Stuart. Modern operating Systems. Prentice Hall, 1992. 728 s.

[7] VORÁČKOVÁ, Šárka; PĚNIČKA, Martin ; VESELÝ, Jaroslav. Úvod do modelování procesů Petriho sítěmi. Vyd. 1. Praha : ČVUT v Praze, 2008. 126 s.

Vedoucí diplomové práce: **prof. Ing. Jan Čapek, CSc.**
Ústav systémového inženýrství a informatiky

Datum zadání diplomové práce: **4. října 2010**

Termín odevzdání diplomové práce: **6. května 2011**

doc. Ing. Renáta Myšková, Ph.D.
děkanka

L.S.

doc. Ing. Jiří Křupka, Ph.D.
vedoucí ústavu

V Pardubicích dne 4. října 2010

Prohlašuji:
Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.
Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 3. 5. 2011

Martin Ibl

PODĚKOVÁNÍ

Touto cestou bych rád poděkoval panu prof. Ing. Janu Čapkovi, CSc., za poskytnuté cenné rady a odborné vedení při zpracovávání této práce.

ANOTACE

Práce se zaměřuje na popis a modelování činnosti vláken v moderních operačních systémech. Součástí práce je osvojení Petri sítí jako modelovacího nástroje s důrazem na barvené Petri sítě. Samotné modelování je realizováno pomocí barvených Petri sítí a zaměřuje se na stavový, behaviorální a kontextuální pohled na činnosti vláken.

KLÍČOVÁ SLOVA

vlákna, modelování, operační systém, Petri sítě, CPN Tools, scheduling, synchronizace, souběžnost, deadlock, race condition, kritická sekce, Standard ML

TITLE

Modeling of activities of threads in the operating system

ANNOTATION

The work focuses on description and modeling of activities of threads in modern operating systems. A part of this work is the adoption of Petri nets as a modeling tool with emphasis on Coloured Petri Nets. The actual modeling is done by using colored Petri nets and focuses on state, behavioral and contextual view of the threads activities.

KEYWORDS

Threads, Modeling, Operating system, Petri nets, CPN Tools, Scheduling, Synchronization, Concurrency, Deadlock, Race condition, Critical region, Standard ML

Obsah

Seznam obrázků

Seznam tabulek

Seznam příloh (DVD)

Příloha č. 1 – CPN Tools. Zdroj [14]

Příloha č. 2 – Základní stavový model vláken. Zdroj vlastní

Příloha č. 3 – Časovaný stavový model vláken. Zdroj vlastní

Příloha č. 4 – Model vláken na uživatelské úrovni. Zdroj vlastní

Příloha č. 5 – Model algoritmu First-Come First-Served. Zdroj vlastní

Příloha č. 6 – Model algoritmu Shortest Job First. Zdroj vlastní

Příloha č. 7 – Model algoritmu Round Robin. Zdroj vlastní

Příloha č. 8 – Model obecného algoritmu prioritního plánování. Zdroj vlastní

Příloha č. 9 – Model metody join. Zdroj vlastní

Příloha č. 10 – Model metody sleep. Zdroj vlastní

Příloha č. 11 – Model metod enter a exit (zamykání). Zdroj vlastní

Příloha č. 12 – Model metod pulse a wait (signalizace). Zdroj vlastní

Příloha č. 13 – Model kontextu vláken. Zdroj vlastní

Příloha č. 14 – Verifikační výstupy. Zdroj vlastní

Úvod

V letech minulých bylo zvyšování výkonu procesorů primárně dosahováno pomocí zvyšování jeho frekvence. Postupem času se však ukázalo, že křemíkový základ pro tvorbu čipů má své limity, resp. tento fakt byl známí již od začátku, ale v poslední době se začal značně projevovat. Hlavním problémem ve zvyšování frekvencí procesorů byla jejich spotřeba elektrické energie a s tím spojená vysoká hřejivost. Tento problém se od začátku řešil miniaturizací architektury, tj. snížit fyzickou velikost jednotlivých tranzistorů, tím zajistit nižší spotřebu elektrické energie a zároveň umožnit zvýšení jejich počtu na čipu. Nástupem nové technologie (na základě standardního inovačního procesu hlavních výrobců procesorů) se začalo jít cestou zvyšováním počtu výpočetních jader procesoru. Tímto faktem se umožnilo využít potenciál současných víceprocesorových operačních systémů (dále OS) na běžných desktopových počítačích. Moderní víceprocesorové OS umožňují na úrovní jádra paralelizovat proces (paralelizovatelný) do jednotlivých vláken, která jsou řešena jednotlivými procesory, resp. jádry. Před nástupem vícejádrových procesorů nebylo možné v klasických jednoprocesorových počítačích tento potenciál využít. Z tohoto důvodu se v současné době široce rozšiřuje paralelní programování, díky němuž je možné jednotlivě tvořené aplikace strukturalizovat na autonomní úseky (vlákna), a tím umožnit maximální využití potenciálu paralelního zpracování.

Hlavním cílem této práce je popsat a následně namodelovat činnost vláken ve víceprocesorových operačních systémech. Toto modelování bude provedeno pomocí Petri sítě. Součástí práce je ověření navržených modelů pomocí verifikačních nástrojů modelovacího nástroje.

Práce je členěno do 8 kapitol. První kapitola pojednává o základní problematice operačních systémů, která je důležitá z hlediska popisu a modelování vláken. V druhé kapitole je nastíněn základní koncept modelování souběžných systémů, vč. problematiky Petri sítí, jako modelovacího nástroje. Třetí kapitola je věnována vysvětlení všech důležitých struktur a konceptů barvených Petri sítí. Součástí kapitoly jsou názorné příklady, na kterých jsou vysvětlovány jednotlivé zákonitosti. Čtvrtá kapitola je koncipována jako úvod do problematiky vláken, tj. jsou zde definovány základní pohledové úhly na problematiku vláken, vč. standardního stavového modelu. Pátá kapitola pojednává o základních plánovacích algoritmech, které řídí přidělování vláken dostupným procesorům/jádrům. Součástí kapitoly je porovnání jednotlivých algoritmů na základě simulace navržených modelů. Šestá kapitole se věnuje základním synchronizačním nástrojům a problémům, které mohou způsobit nekonzistentnost. V sedmé kapitole je nastíněn kontextuální pohled na vlákna z hlediska interakce s procesem a operační pamětí. Poslední kapitola ilustruje příklad využití verifikačních nástrojů CPN Tools při ověřování správnosti navržených modelů.

1 Operační systém

Všeobecně se dá operační systém (dále OS) považovat za logickou vrstvu mezi hardwarem a uživatelem počítače. Na tuto vrstvu lze zpravidla nahlížet dvěma na první pohled nesourodými způsoby [1]:

- operační systém jako hardwarová nástavba,
- operační systém jako správce zdrojů.

1.1 Operační systém jako hardwarová nástavba

Operační systém lze považovat za hardwarové rozhraní na vyšší úrovni abstrakce. Jednotlivé hardwarové prvky jsou primárně ovládány sadou instrukcí, která je z hlediska programování velice nevhodná. OS tedy umožňuje vytvořit instrukční sadu vyššího řádu, která zefektivňuje práci s hardwarovými prvky. Bez OS by každý programátor musel programovat např. pohyb hlaviček v pevném disku, rychlost točení DVD mechaniky, komunikaci po sběrnicích nebo organizaci operační paměti. V takovém případě by každý programátor musel znát instrukční sady všech hardwarových prvků a zároveň jejich funkční principy. I pokud by programátor takovýto program vytvořil, nefungoval by nikde jinde, než na tom počítači kde byl programován (popř. na absolutně stejném počítači). [1]

Operační systém tedy umožňuje univerzálním způsobem obsluhovat jednotlivé hardwarové prvky a nabídnout jednotné rozhraní (**API – Application Programming Interface**) pro uživatele počítače (programátory). V souvislosti s tímto pohledem lze OS považovat za tzv. **virtuální stroj**, který lze jednodušeji programovat. [1]

1.2 Operační systém jako správce zdrojů

Moderní počítače se skládají z procesorů, operačních pamětí, pevných disků, myší, klávesnic, tiskáren a řady dalších zařízení. Operační systém umožňuje spravovat jednotlivá (sdílená) zařízení takovým způsobem, aby nedocházelo k nepožadovaným výsledkům. Sdílení zařízení se nazývá **multiplexing** a může probíhat dvěma způsoby [1]:

- sdílení z hlediska **času**,
- sdílení z hlediska **místa**.

Sdílení (**multiplexing**) z hlediska **času** je např. přidělování jednoho procesoru více procesům, sdílení tiskárny více uživateli apod. Operační systém tedy zajišťuje přidělování jednotlivých zdrojů, tak aby jednotlivým uživatelům (procesům) bylo vyhověno (v konečném čase). [1]

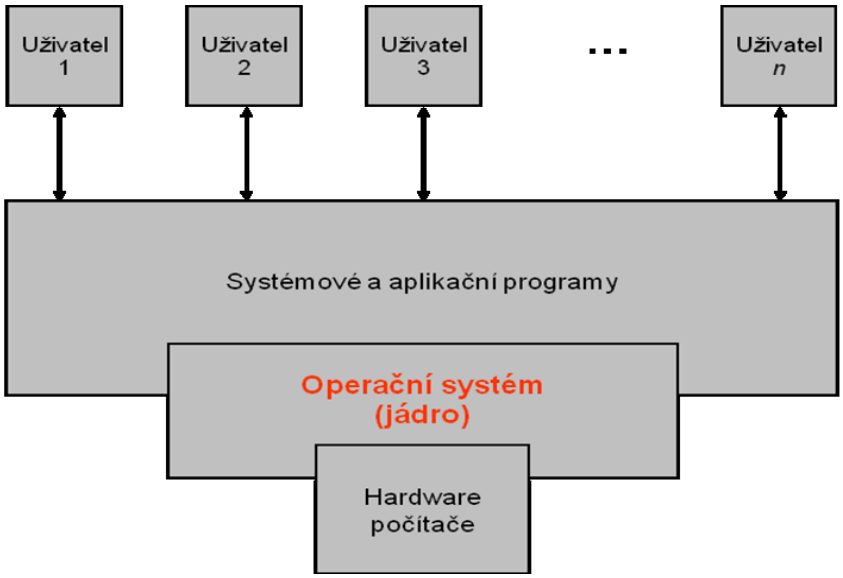
Druhým typem sdílení (**multiplexing**) je sdílení podle **místa**. Typickým příkladem je správa operační paměti, která umožňuje koexistenci velkého počtu procesů bez vzájemné kolize. Dalším příkladem sdílení podle místa může být např. správa pevného disku. [1]

Operační systém tedy zajišťuje správu zdrojů, která je **spravedlivá** (každý uživatel dostane svůj podíl času/místa) a **bezpečná** (nedojde k neočekávaným kolizím, např. přepis paměti). [1]

1.3 Základní koncepce a vývoj OS

V průběhu let se koncept operačního systému značně vyvíjel. V počátku (50. léta 20 století) se jednalo pouze o řadu instrukcí, které umožnily zautomatizovat opakující se činnosti na prvních počítačích. Zpracovávat se však dala pouze jedna úloha současně. Jednalo se zpravidla o obrovské počítače, které zpracovávali úlohy po dávkách. Postupem času (60. léta 20 století) se začalo využívat tzv. **multiprogramování** (**multiprogramming**), které umožňovalo uchovávat v paměti více úloh najednou (sdílení podle **místa**). Pokud tedy jedna úloha čekala na vstupně výstupní operaci, mohla další úloha ihned začít pracovat. Tímto způsobem se zamezilo prodlení z důvodu čekání na vstupní data (např. čas, než si počítač data načte do paměti). Dalším krokem ve vývoji OS byl tzv. **timesharing** (sdílení **času**). **Timesharing** umožňoval interaktivní přístup více uživatelů pomocí terminálů. Před využitím **sdílení času** (**timesharing**) musel programátor např. čekat půl dne, než se dokončí nějaká úloha, aby si mohl vyzkoušet (ověřit, zkompilovat) svůj program. V současné době již drtivá většina OS (např. Windows, Linux, Mac OS) navíc také obsahuje grafické uživatelské rozhraní (**GUI – Graphical User Interface**), které zvyšuje uživatelskou interaktivnost. [1][2]

Většina moderních OS je koncipována jako malá sada instrukcí (**microkernel**), která obsluhuje pouze nejzákladnější funkce (správu paměti, přidělování procesoru, správa vstupně výstupních zařízení apod.). Veškeré systémové služby poté běží jako nezávislé procesy, které využívají služeb jádra. Takovýto princip se zpravidla označuje jako architektura typu **Klient/Server**. Umístění OS v počítačové hierarchii znázorňuje Obrázek 1. [3]

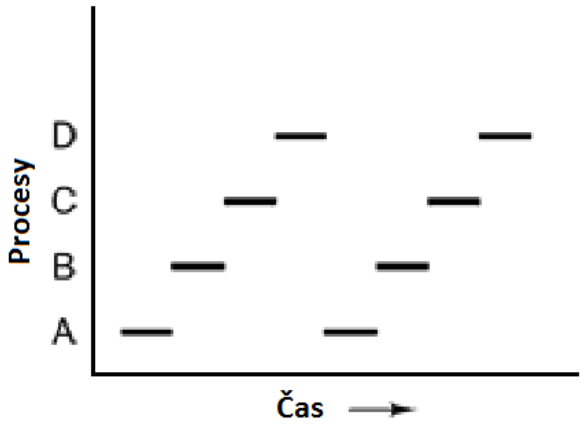


Obrázek 1 - Operační systém v počítačovém modelu. Zdroj [3]

1.4 Procesy

Každý systémový objekt v počítači je z hlediska OS vnímán jako proces. Procesem se rozumí jakýkoli spuštěný (načtený v operační paměti) objekt (program, služba). Proces si lze představit jako kontejner, kterému je přidělena část operační paměti a vykonává nějakou činnost (program). Každý takovýto kontejner se nachází v nějakém kontextu vůči hardwaru a OS. Tento kontext, tedy vyjadřuje stav daného procesu v určitý časový okamžik vůči svému okolí. [3]

Hlavním účelem procesů je možnost jejich přepínání na hardwarové úrovni, tj. dynamicky využívat systémové zdroje podle potřeby, resp. maximalizovat využívání těchto zdrojů. Jak již bylo jednou zmíněno, toto přepínání se nazývá **multiprogramming** a umožňuje efektivně využívat systémové zdroje (např. pokud jeden proces **čeká** na **vstupně výstupní** operaci, jiný proces může být zpracováván). Obrázek 2 znázorňuje přepínání procesů **A**, **B**, **C** a **D** na jednom procesoru. [1]



Obrázek 2 - Multiprogramming. Zdroj [1]

Multiprogramming je tedy vlastnost OS, která umožňuje prokládat jednotlivé programy (procesy) za účelem **maximalizace výkonu**. Na druhou stranu podobný termín **multitasking** představuje podobnou logiku, avšak s jiným cílem. **Multitasking** si klade za cíl vytvořit „**iluzi**“ běhu více programů (procesů) současně pro **uživatele** (**interaktivnost**). **Multitasking** je zajišťován pomocí **multiprogramování (multiprogramming)** a **sdílení času** (**timesharing**). [1]

1.4.1 Spouštění a ukončování procesů

Každý proces musí být nějakým způsobem vytvořen (inicializován) a ukončen (terminován). Proces může vzniknout na základě čtyř hlavních událostí [1]:

- **systémová inicializace** (vznik při startu systému),
- **vznik na základě systémového volání jiného procesu** (běžící proces může vytvořit nový proces),
- **uživatelský požadavek** (příkazová řádka, poklepání na ikonu),
- **inicializace na základě dávky** (předchozí dávka byla zpracována, tudíž může být načtena nová).

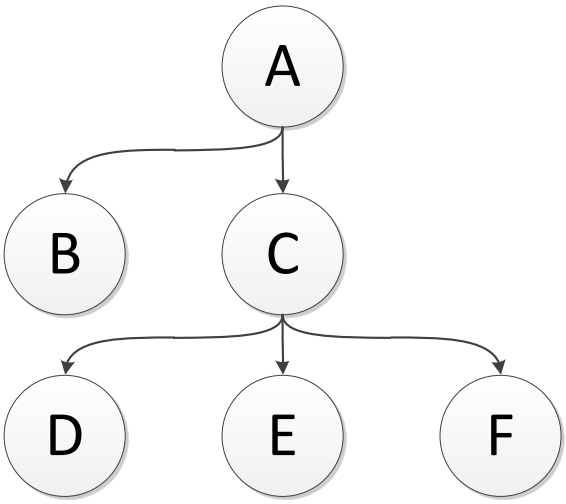
Ukončování procesů může být uvozováno některou z následujících událostí [1]:

- **klasické ukončení** (proces vykonal všechnu práci nebo jej uživatel ukončil),
- **chybové ukončení** (proces se ukončil, jelikož nedostal požadované zdroje pro další činnost ve vhodné formě),
- **fatální chybové ukončení** (proces se ukončil na základě chyby v kódu, tzv. „bug“),
- **ukončení jiným procesem** (jiný proces ukončil činnost procesu systémovým voláním).

Klasické a chybové ukončení je volitelné, tj. řídí je uživatel nebo sám proces. Ostatní typy zakončení jsou prováděny nezávisle na ukončovaném procesu, či uživateli (řídí je OS).

S vytvářením a ukončováním jednotlivých procesů je spojena jejich hierarchie. Hierarchie je tvořena podle principu rodič-potomek, tj. proces, který vytvořil proces (potomka) je jeho rodičem. Každý potomek může mít další potomky (čímž se stává jejich rodičem) apod. Takovýmto způsobem je tvořen tzv. hierarchický strom procesů. [4]

Obrázek 3 ilustruje příklad hierarchie, kde proces A (rodič) má dva potomky (procesy B a C). Dále potomek C má své vlastní potomky procesy D, E a F.



Obrázek 3 - Hierarchie procesů. Zdroj vlastní

Hierarchie procesů je významná, jelikož pokud je ukončen rodič, ukončí se i všichni jeho potomci. Dále je znalost hierarchie důležitá pro případnou meziprocesní komunikaci. [1][4]

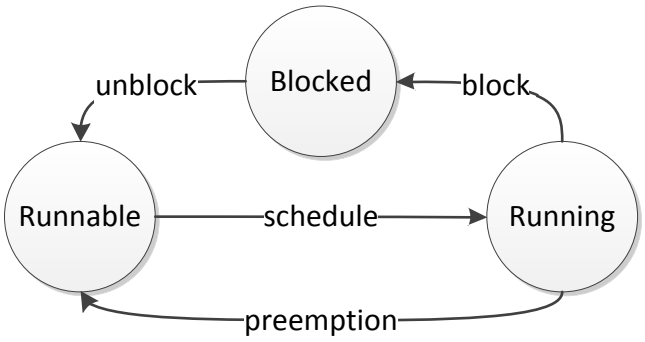
1.4.2 Stavy procesů

Mimo výše zmíněné stavy (nový a ukončený proces) existují další stavy, které reprezentují aktuální status procesu z hlediska přidělení procesoru.

V základu jsou rozlišovány 3 typy stavů [1]:

- **Running** (proces běží, tj. je mu přidělen procesor),
- **Runnable** (proces je připraven a čeká na přidělení procesoru),
- **Blocked** (proces není schopen běhu, jelikož čeká na nějakou vstupně výstupní událost).

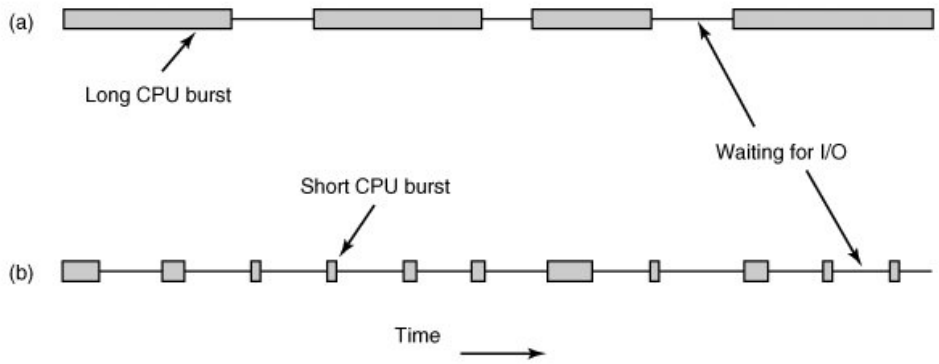
První dva stavy jsou podobné a označují situaci, kdy proces požaduje procesor. Liší se pouze v tom, zda jim je nebo není procesor skutečně přidělen. Oproti tomu stav „**Blocked**“ vyjadřuje neschopnost běhu procesu, dokud není splněna blokovací podmínka. Obrázek 4 ilustruje možné přechody mezi těmito třemi stavy.



Obrázek 4 - Stavy procesů. Zdroj vlastní

Ze stavu „**Runnable**“ je možné přejít pouze do stavu „**Running**“. Struktura, která rozhoduje o přidělení procesoru (přechod mezi stavy „**Runnable**“ a „**Running**“) se nazývá **scheduler** a jeho činnost bude podrobněji popsána v kapitolách 1.6 a 5. Na okraj je možné zmínit, že akce „**schedule**“ a „**preemption**“ jsou prováděny na **úrovni jádra** a proces o nich vůbec **neví**. Přechod do stavu „**Blocked**“ je uvozován nedostatečností nějakého vstupního zdroje (např. čekání na data, na splnění logické podmínky apod.). Pokud jsou podmínky, jejichž nedostatečnost vyvolalo blokování splněny, je proces opět odblokován („**unblock**“) a připraven k přidělení procesoru („**Runnable**“). Situace, že bude proces zablokován, je řízena přímo vnitřní strukturou programu procesu, popř. jeho rodičem.

Téměř každý proces lze rozdělit na dvojici alternujících dávek. Jedná se o dávku procesoru (**CPU brust**) a vstupně výstupní dávku (**I/O burst**). Tyto dávky v podstatě reprezentují střídání stavů „**Running**“ a „**Blocked**“. Z hlediska poměru času stráveným počítáním („**Running**“) a čekáním na vstupně výstupní událost („**Blocked**“), lze poté proces klasifikovat jako tzv. **CPU-Bound** nebo **I/O-Bound** proces. **CPU-Bound** proces (viz Obrázek 5(a)) je příznačný tím, že většinu času stráví počítáním, resp. využíváním procesoru (ve stavu „**Running**“). Jeho **vstupně výstupní** (**I/O**) operace se odehrávají zřídka. Takovéto procesy zpravidla reprezentují různé vědecko-výzkumné výpočty, při nichž proces počítá až do doby, než potřebuje nová data. Odlišným případem jsou tzv. **I/O-Bound** procesy (viz Obrázek 5(b)), které většinu času tráví čekáním na vstupně výstupní operaci. Jedná se typicky o procesy, které interagují s uživateli. V tomto případě tedy většinu vstupně výstupních operací představují různé uživatelské události (např. zmáčknutí tlačítka). Zároveň tyto procesy zpravidla nepočítají žádné složité úlohy, tudíž je jejich doba přidělení procesoru (**CPU burst**) relativně krátká. [4]



Obrázek 5 - Chování procesů - (a) CPU-Bound vs. (b) I/O-Bound. Zdroj [4]

1.4.3 Implementace procesu

Každý proces obsahuje datovou strukturu, která nese informace o jeho stavu vůči procesoru, operační paměti a souborovému systému. Takováto struktura se často označuje jako **PCB** (Process

Control Block) a uchovává všechny potřebné údaje o kontextu. Při každé změně stavu (viz Obrázek 4) proces aktualizuje **PCB**. Tímto způsobem je vždy zachován poslední stav procesu a není problém činnost aktuálně zpracovávaného procesu přerušit a poté pokračovat ze stejného stavu. [1]

1.5 Vlákna

V moderních operačních systémech každý proces obsahuje minimálně jedno řídící vlákno (dále vlákno), které vykonává samotný program, tj. zajišťuje práci s procesorem. Proces jako takový se zpravidla zabývá dvěma činnostmi [1]:

- spravuje zdroje,
- provádí samotný program.

Jak již bylo zmíněno, každý proces má **vlastní adresový prostor** v operační paměti. Zde jsou uchovávány všechny zdroje aktuálně využívané daným procesem. Základní obsah adresového prostoru tvoří **program**, **data** a **ostatní zdroje**. Mezi ostatní zdroje se řadí např. otevřené soubory či další procesy (potomci). Proces tedy usnadňuje práci se všemi těmito zdroji, jelikož je **zapouzdřuje** do **kompaktní** formy. [4]

Druhou činností procesu je samotná řídící a výpočetní činnost, která je vykonávána vláknem. **Vlákna** jsou tedy ty části procesu, které provádějí činnosti **spojené** s **procesorem** počítače. Pokud operační systém nepodporuje vlákna (např. UNIX), poté každý proces zajišťuje i interakci s procesorem, tj. vlákna na úrovni jádra nejsou definována (nejsou uvažována). [5]

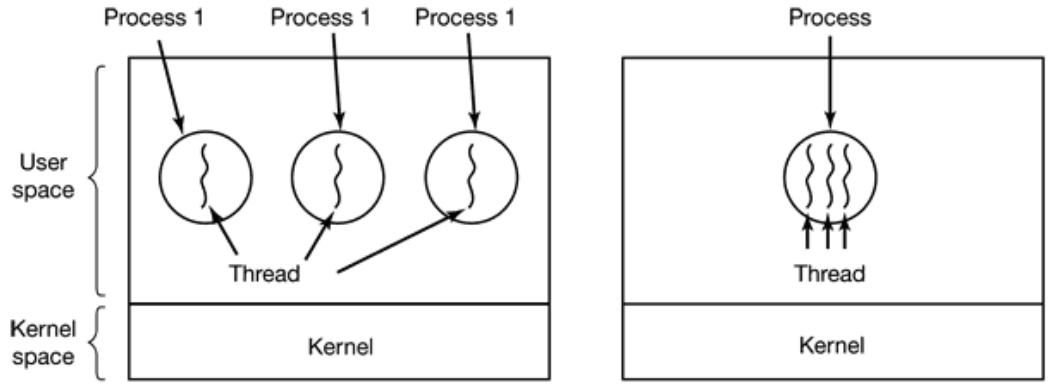
1.5.1 Vlákno versus proces

Pokud OS podporuje **multithreading**, může každý proces obsahovat více než jedno vlákno. Více vláken se používá pokud [3]:

- program lze vykonat paralelně (zvýšení výkonnosti),
- program používá GUI a zároveň zpracovává data (zajištění interaktivnosti).

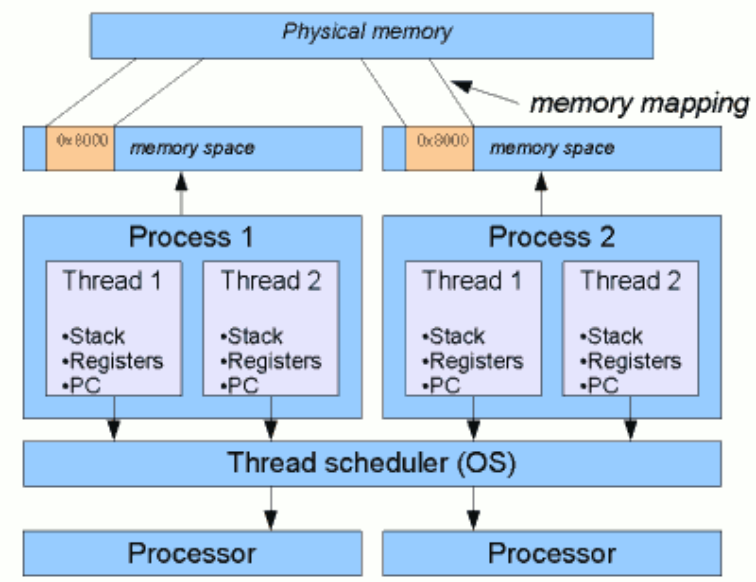
Pokud by nebyl **multithreading** využit, musí proces běžet sekvenčně a vykonávat svůj program, dokud nedojde k jeho zablokování nebo dokončení jeho činnosti. Pokud by takovýto proces běžel na víceprocesorovém počítači, resp. vícejádrovém procesoru, byl by v jeden časový okamžik vždy zpracováván pouze na **jednom** procesoru, resp. jádru. Podobná věc platí pro interaktivnost vůči uživateli. Pokud jednovláknový proces představující uživatelskou aplikaci začne zpracovávat data (např. ověřování uživatelské autentizace) aplikace přestane odpovídat („zamrzne“).

Oba tyto „nedostatky“ jsou řešeny pomocí **vícevláknových** procesů. Používání více vláken však sebou nese i mnohé problémy s jejich **synchronizací**, a tudíž vždy nastává otázka, co je výhodnější, zda **výkon** nebo **transparentnost** (problém synchronizace je podrobněji probrán v kapitole 6). Hlavním problémem při používání vláken je fakt, že sdílejí stejný adresový prostor. Obrázek 6 ilustruje tři vlákna v různé souvislosti. Vlevo tři procesy obsahují jedno vlákno. Vpravo jeden proces obsahuje tři vlákna.



Obrázek 6 - Vlákno versus proces. Zdroj [1]

Obrázek 7 znázorňuje podrobnější vazbu mezi vlákny a procesy. Jsou zde vyobrazeny dva procesy (**Process 1** a **Process 2**), které obsahují po dvou vláknech (**Thread 1** a **Thread 2**). Každé vlákno obsahuje zásobník (**Stack**), výpis registrů (**Registers**) a programový čítač instrukcí (**PC** – **Program Counter**). Směrem vzhůru (**memory space**) každý proces komunikuje s operační pamětí (každý má dedikovanou svou část). Směrem dolů se nachází **Thread scheduler** (plánovač vláken), který přiděluje jednotlivá vlákna dostupným **procesorům/jádrům**.

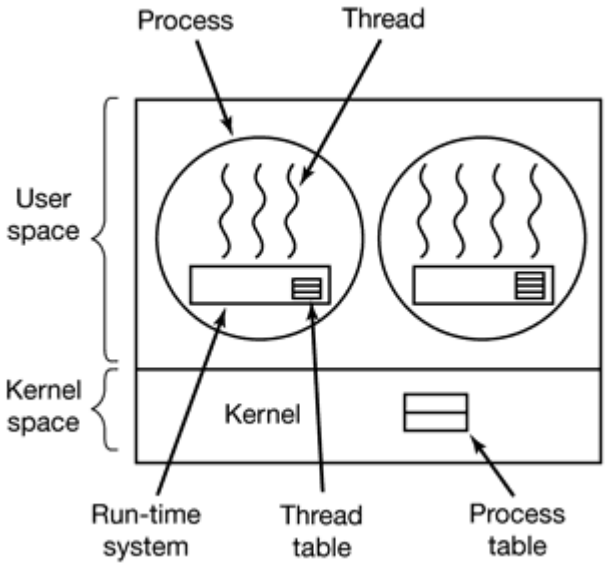


Obrázek 7 - Vlákno versus proces - Podrobněji. Zdroj [6]

Obecně lze konstatovat, že procesy se využívají při práci s různými daty. Vlákna jsou naopak využita, pokud se pracuje se stejnými daty, s nimiž je třeba vykonat řadu činností (nejlépe paralelně, pokud dovoluje hardware). Z tohoto důvodu je možné při zpracovávání vláken využít potenciál současných vícejádrových procesorů. [1]

1.5.2 Vlákna na uživatelské úrovni

Vlákna na uživatelské úrovni jsou signifikantní tím, že řídí vlastní činnost sama. Jádro OS o nich vůbec neví, a tudíž je nemůže řídit. Jelikož všechna řídící činnost probíhá uvnitř procesu, není nutné při zpracovávání různých vláken přepínat kontext. To vede k velice snadnému a hlavně rychlému přepínání mezi vlákny. Každý proces obsahuje vlastní tabulku vláken (viz Obrázek 8), která uchovává všechny potřebné údaje o vláknech (např. priorita, stav apod.). Na úrovni jádra je pouze procesní tabulka (**PCB**), která nese informace o procesech. Hlavní výhodou vláken na uživatelské úrovni je fakt, že mohou běžet i na systémech, které vlákna nepodporují (např. UNIX). [4]

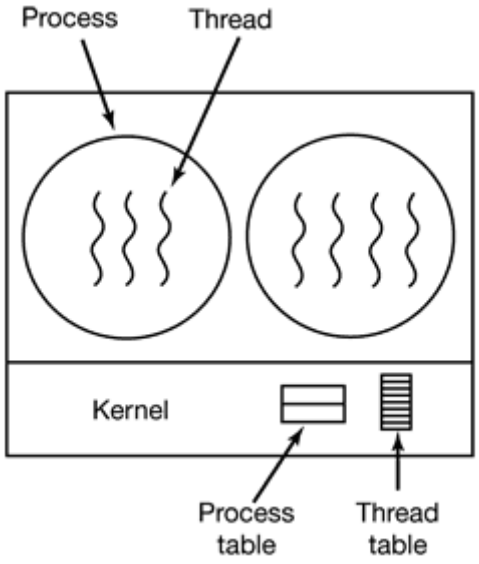


Obrázek 8 - Vlákna na uživatelské úrovni. Zdroj [1]

Hlavní nevýhodou těchto vláken je fakt, že pokud daný proces obdrží systémové volání (např. přerušení) je zablokován celý proces. Zároveň nelze využít **multiprocessing** (paralelní zpracování).

1.5.3 Vlákna na úrovni jádra

Vlákna na úrovni jádra jsou rozdílná v tom, že proces neobsahuje tabulku vláken. Kontext vláken všech procesů je ukládán do vláknové tabulky na úrovni jádra (viz Obrázek 9). Pokud tedy dojde k systémovému volání, je zablokováno pouze vlákno, jehož se to týká a může začít zpracovávání dalšího vlákna z téhož procesu. Pokud proces neobsahuje další připravená vlákna (ve stavu „**Runnable**“), je zpracováno vlákno z jiného procesu. [4]

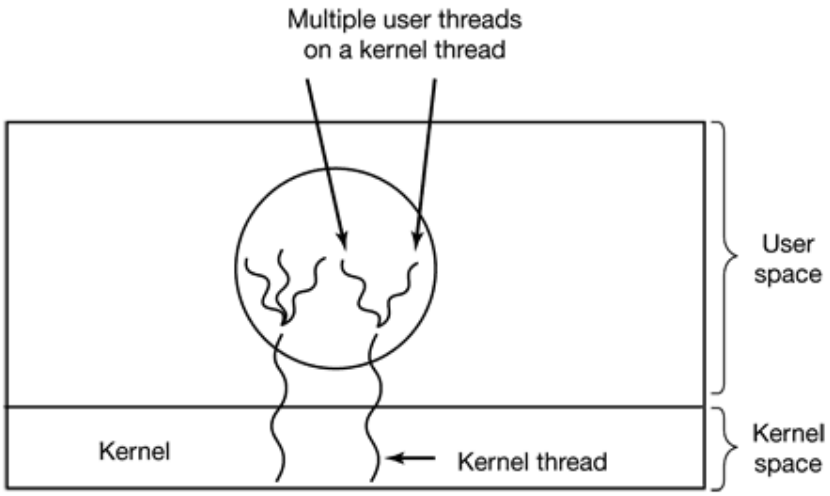


Obrázek 9 - Vlákna na úrovní jádra. Zdroj [1]

Hlavní nevýhodou tohoto typu vláken je výkon (na jednoprocesorových strojích), jelikož každý přechod mezi vlákny vyvolá přepínání kontextu. Na druhou stranu je možné využít **paralelní** zpracování na hardwarové úrovni (**multiprocessing**). [4]

1.5.4 Hybridní vlákna

Výhody a nevýhody obou výše zmíněných typů vláken je možné zkombinovat podle konkrétní situace do tzv. hybridních vláken. Hybridní vlákna představují skutečnost, že OS podporuje vlákna na úrovni jádra, avšak zároveň je využita implementace vláken na uživatelské úrovni. Tato situace je znázorněna na Obrázek 10. [4]



Obrázek 10 - Hybridní vlákna. Zdroj [1]

1.6 Plánování procesů a vláken (scheduling)

V operačním systému, který umožňuje **multiprogramming** je běžné, že několik procesů požaduje přidělení procesoru ve stejný okamžik. Tedy existuje více procesů ve stavu „**Runnable**“, než je fyzický počet **procesorů**. Pokud takováto situace nastane, je nutné pomocí vhodného **algoritmu** vybrat, kterému **procesu/vláknu** bude **procesor/jádro** přidělen a kterému nikoliv. Součást operačního systému, která provádí toto rozhodnutí, se nazývá **scheduler** a celý proces rozhodování **scheduling**. [1]

Výše zmíněný typ plánování (**scheduling**) je formálně nazýván operační plánování (**Short Term Scheduling**), které zajišťuje přidělování procesů/vláken procesorům/jádrům. Dále systém zajišťuje taktické plánování (**Mid Term Scheduling**), které řídí odkládání procesů z operační paměti (**swapping**) a strategické plánování (**Long Term Scheduling**), které spravuje vytváření nových procesů. Taktické a strategické plánování determinují tzv. **stupeň multiprogramování** (regulují počet procesů v operační paměti). [1][3]

Jelikož je z hlediska plánování vláken důležitý pouze operační plánovač (**Short Term Scheduler**), bude v následujícím uvažován pouze tento a označován všeobecně pojmem **scheduler**.

Výběr vhodného algoritmu přidělování procesoru/jádra v první řadě závisí na typu operačního systému, tj. k čemu je daný systém určen. Z hlediska problematiky plánování procesů a vláken jsou rozlišovány tři typy operačních systémů [1]:

- dávkové systémy,
- interaktivní systémy,
- realtimové systémy.

Dávkové systémy jsou zpravidla navrženy pro zpracovávání velkého množství dat po jednotlivých dávkách. Není zde kladen důraz na interaktivnost, jelikož zde nejsou přímí uživatelé čekající na rychlou odpověď. Lze využít nepreemptivní algoritmy, nebo preemptivní algoritmy s dlouhou dobou mezi přerušeními (preempce viz kapitola 1.6.1). [1]

V **interaktivních systémech** tvoří hlavní roli uživatelé, kteří požadují rychlou odezvu rozhraní a celkovou uživatelskou přívětivost. Preemptivní algoritmy jsou podmínkou pro správný běh takovýchto systémů. [1]

Realtimové systémy jsou signifikantní požadavkem na zpracování všech procesů v předem určené době. Nesplnění těchto termínů může vést k fatálním důsledkům (např. systémy podpory života v nemocnicích) nebo ke ztrátě informace (např. přehrávání videa). [1]

Na každý z těchto systémů jsou kladeny různé požadavky, avšak některé mají společné. Těmito společnými požadavky jsou **spravedlivost** a **vyrovnanost**. Spravedlivostí se rozumí stav, kdy každý proces/vlákno má rovnocennou šanci pro získání procesoru/jádra (nedochází k **stárnutí** procesů/vláken). Vyrovnanost představuje fakt, že všechny části (prvky) systému jsou plně vytíženy. Dílčí požadavky různých typů OS ilustruje Tabulka 1. [1]

Tabulka 1 - Specifické požadavky v různých typech OS. Zdroj [1]

Typ OS	Požadavek	Popis
Dávkové systémy	Propustnost	Maximální počet úloh za hodinu
	Doba obrátky	Minimalizovat čas mezi podáním a ukončením dávky
	Využití procesoru	Udržet procesor vytížený po celou dobu práce
Interaktivní systémy	Doba odezvy	Rychle reagovat na uživatelské požadavky
	Proporcionalita	Splňovat požadavky uživatelů
Realtimové systémy	Dodržování termínů	Aby nedošlo ke ztrátě dat
	Předvídatelnost	Vyhnout se zhoršení kvality v multimediálních OS

1.6.1 Plánovací algoritmy

Plánovací algoritmy zajišťují plnění požadavků různých typů operačních systémů. Primárně se tyto algoritmy dělí na **preemptivní** a **nepreemptivní**. **Preemptivní** algoritmy umožňují stanovit fixní dobu práce procesu/vlákna. Pokud proces/vlákno do konce uplynutí této doby neukončí svou činnost, je **plánovačem** (**scheduler**) přerušen a přejde zpět do stavu „**Runnable**“. Otázkou zůstává jak dlouhé **quantum** zvolit, jelikož příliš dlouhé zpomaluje odezvu systému a naopak příliš krátké zpomaluje systém v důsledku častého přepínání kontextu. **Nepreemptivní** algoritmy umožňují zpracovávanému procesu/vláknu běžet, dokud se sám nezablokuje (např. čekáním na vstupně výstupní operaci). Tyto algoritmy jsou zpravidla vhodné pro **neinteraktivní** systém. [1][4]

Plánovací algoritmy se liší podle výše zmíněných typu operačního systému a specifických požadavků na ně kladených, viz Tabulka 2.

Tabulka 2 - Příklady plánovacích algoritmů. Zdroj [1]

Typ systému	Název algoritmu	Typ algoritmu	Popis algoritmu
Dávkové systémy	First-Come First-Served	Nepreemptivní	Dávka, která první požádala o procesor, je první zpracována
	Shortest Job First	Nepreemptivní	Nejkratší dávka je zpracována první
	Shortest Remaining Time Next	Preemptivní	Preemptivní verze předchozího algoritmu
Interaktivní systémy	Round Robin Scheduling	Preemptivní	Každý proces dostane procesor na dobu pevné délky, samotný výběr probíhá podle příchodu procesu
	Priority Scheduling	Preemptivní	Každému procesu je přidělena priorita, proces s vyšší prioritou má vždy přednost před nižší
Realtimové systémy	Earliest Deadline First	Preemptivní	Každý proces je definován požadovaným začátkem a koncem zpracování. Proces, jehož termín konce (deadline) je nejblíže, bude zpracován jako další.

Některé z těchto algoritmů budou podrobněji popsány a namodelovaný v kapitole 5.

1.6.2 Plánování vláken

Pokud proces obsahuje více než jedno vlákno, je možné zapojit do uvažování plánování vláken. Prvně je nutné odlišit plánování na uživatelské a kernelové úrovni.

Vlákna na **uživatelské úrovni** nevyužívají služeb jádra, a tudíž o nich **scheduler** vůbec neví. Pokud je procesu s uživatelskými vlákny přidělen procesor na dané **quantum** jsou jednotlivá vlákna zpracovávána dle **plánovače** (**scheduler**) uvnitř daného procesu. Tedy vlákno běží tak dlouho jak požaduje (nehrozí přerušení ze strany OS).

Na druhou stranu vlákna na úrovni jádra jsou „**viditelná**“ systémových **plánovačem** (**scheduler**), a tudíž mohou být přidělena různým procesorům/jádrům nezávisle na procesu, kterému patří. Hlavní rozdíl mezi vlákny na uživatelské a kernelové úrovni je výkon. Přepínání mezi uživatelskými vlákny zabere pár strojových instrukcí. U vláken na kernelové úrovni se musí přepínat celý kontext, což zabere více času i systémových zdrojů. Naopak hlavní nevýhodou uživatelských vláken je fakt, že při blokování vlákna je automaticky zablokován celý proces. [1]

2 Modelování a Petri sítě

Základní možností jak popsat činnosti vláken v operačním systému je jejich modelování. Jelikož vlákna sdílejí systémové zdroje (procesor, paměť přidělenou procesu), existuje velké množství různých interakcí a možností souběžné činnosti, jež jsou nutné synchronizovat, popř. naplánovat. Z tohoto důvodu je kladen velký důraz na vhodný modelovací nástroj, který umožňuje modelovat (popř. simulovat) souběžnou činnost.

2.1 Modelování souběžných systémů

Souběžné systémy se zpravidla vyznačují velkou složitostí (existuje astronomické množství různých stavů systémů). Modelování souběžných systémů spočívá v nalezení všech důležitých vzorů interakcí mezi autonomními částmi a tím zajistit logickou správnost modelu na dané rozlišovací úrovni. Souběžnými systémy mohou být např. [7]:

- nukleární elektrárna,
- řízení leteckého provozu,
- bankovní systém,
- počítačové sítě,
- nemocniční zařízení pro podporu života.

U takovýchto systémů je požadováno, aby pracovali správně od samého počátku jejich nasazení, tj. není zde prostor pro průběžné testování a zdokonalování za běhu. Složitost těchto systémů vede k nutnosti využívat nástroje, které umožňují testování a „debugging“ již ve fázi modelování, tj. před samotnou implementací a nasazením. [8]

Modelování jako takové je univerzální technika, která může být využita v široké škále aktivit v průběhu vývoje systému. Model je abstraktní reprezentace, se kterou může být manipulováno např. pomocí počítačového nástroje. Používání modelů umožňuje zkoumat chování systému. [7]

Modely umožňují [8]:

- porozumět navrhovanému systému (může vést, k zjednodušení modelu),
- zlepšit správnost navrhovaného systému na základě simulace,
- zkoumáním zlepšit kompletnost (úplnost) a korektnost (správnost).

2.2 Petri sítě

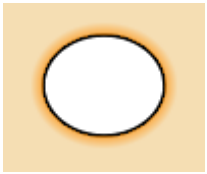
Petri sítě jsou **formalizovaný grafický** modelovací jazyk, který umožňuje modelovat systémy obsahující **souběžnost**. Tento nástroj byl vyvinut během 60 let, když jej německý matematik **C**. **A**. **Petri** zavedl ve své disertační práci „**Kommunikation mit Automaten**“ (1962). [9]

2.2.1 Základní pojmy

Základní **Petri síť** je tvořena následujícími objekty [10]:

- místa (**Places**),
- přechody (**Transitions**),
- orientované hrany (**Arcs**),
- kapacity (**Capacity Indications**),
- váhy (**Weights**),
- počáteční značení (**Initial Marking**).

Místo reprezentuje stav jevu (činnosti, procesu apod.). Základní vlastností místa je možnost uchovávat **token**. Token reprezentuje různé skutečnosti v závislosti na typu **Petri sítě**. Základní smysl tokenu je kvantifikovat počet výskytů nějakého jevu v daném stavu (místě). Místo je graficky reprezentované kolečkem, resp. oválem, viz Obrázek 11. U standardních P/T sítí je možné omezit kapacitu místa, tj. kolik tokenů se může v daném místě maximálně nacházet. [10] Ve zbývající částí této práce budou při modelování považovány pojmy „místo“ a „stav“ za totožné.



Obrázek 11 - Místo. Zdroj vlastní

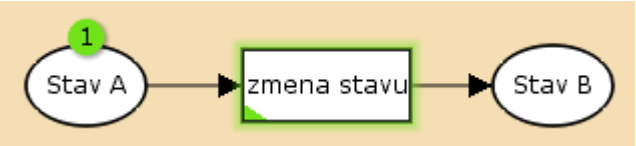
Přechod je graficky reprezentován pomocí obdélníku, popř. čtverce (viz Obrázek 12). Přechod reprezentuje změnu stavu mezi dvěma místi, tj. přechod tokenu z jednoho stavu do jiného. [10]

Obrázek 12 - Přechod. Zdroj vlastní

Propojení mezi místem a přechodem je realizován pomocí **orientované hrany** (viz Obrázek 13). Klasické P/T sítě umožňují ohodnotit hranu přirozeným číslem (váhou), které reprezentuje násobnost dané hrany. Pomocí orientované hrany lze spojovat **pouze** místa a přechody.

Obrázek 13 - Orientovaná hrana. Zdroj vlastní

Ukázka aplikace přechodu za použití dvou míst a dvou orientovaných hran je znázorněn na Obrázek 14. Počáteční značení reprezentuje přítomnost jednoho tokenu v místě „**Stav A**“. Samotný přechod probíhá přenosem tokenu z místa „**Stav A**“ přes přechod „**zmena stavu**“ do místa „**Stav B**“. **Token** reprezentuje **dynamickou** složku **Petri sítí** a může být reprezentován např. bodem, číslicí, vektorem hodnot nebo objektem v závislosti na typu **Petri sítě**.



Obrázek 14 - Aplikace přechodu. Zdroj vlastní

2.3 Vlastnosti Petri sítí

Vlastnosti **Petri sítí** představují množinu základních charakteristik konkrétního modelu.

2.3.1 Bezpečnost (Safeness)

Bezpečnost sítě je spojena převážně s modelováním **binárních** systémů. **Petri síť** lze považovat za bezpečnou, pokud žádné místo v libovolném stavu sítě nepřevýší hodnotu 1. [9][11]

2.3.2 Omezenost (Boundedness)

Omezenost sítě představuje skutečnost, že všechna místa mají pevně danou kapacitu. Jedná se tedy o rozšíření bezpečnosti o fakt, že kapacita místa je omezena limitem „**k“**, tj. místo nemůže obsahovat více než „**k**“ tokenů. Poté je možné o takové sítí konstatovat, že je **k**-omezenou **Petri sítí**. Omezenost se často využívá při modelování zdrojů. [8][9]

2.3.3 Konzervativnost (Conservation)

Konzervativnost sítě reprezentuje fakt, že se počet tokenů v sítí nemění, tj. nové tokeny ani nevznikají, ani nezanikají. Tato vlastnost je hojně využívána při modelování různých strategií přidělování zdrojů v systému. [9][11]

2.3.4 Živost (Lifeness)

Živost je jednou z nejdůležitějších vlastností, která je na modelovaných systémech zkoumána. Živost zkoumá, zda v sítí může nastat tzv. **deadlock** (uváznutí). **Deadlock** sítě je stav, kdy žádný přechod

není aktivní. Situaci, kdy síť neobsahuje žádné aktivní přechody, se dále označuje jako mrtvé značení, popř. mrtvá síť. Zkoumání živosti je důležité např. při analýze různých algoritmů pro zajištění bezpečné spolupráce dvou procesů (např. debugging). [8][9]

2.3.5 Dosažitelnost (Reachability)

Dosažitelnost reprezentovaná stromem dosažitelnosti reprezentuje různé stavy značení, které může síť nabývat. Jedná se tedy o množinu všech stavů sítě a jejich propojení. [8][9]

2.4 Historie a typy Petri sítí

Postupem časy, se základní koncept **Petri** sítí rozšířil do různých vývojových větví. Vznikly různé typy **Petri** sítí, které byly vhodné pro modelování různých problémů.

2.4.1 Časované sítě

Časované Petri sítě jsou rozšířením klasických **Petri** sítí o možnost zkoumat vnitřní čas simulace. Analýza simulačního času umožňuje rozšíření práce s modelem o možnost načasování jednotlivých situací (změn stavů). Typicky se časují přechody přidělením kladného celého čísla, které vyjadřuje časovou náročnost uskutečnění daného přechodu. V některých případech lze časovat i samotný token, viz kapitola 3.4. [9][10]

2.4.2 Stochastické sítě

Příbuzným modelem k časovaným sítím jsou **stochastické Petri** sítě, které umožňují ohodnocení přechodů stochastickým atributem, reprezentujícím délku trvání operace. Tyto sítě se uplatňují při modelování a analýze výkonnosti systému. [9][12]

2.4.3 Barvené sítě

Zavedení **barvených Petri** sítí bylo motivováno snahou odstranit některé nevýhody klasických (P/T) sítí. **P/T Petri** sítě poskytují efektivní primitiva pro popis synchronizace paralelních procesů, avšak složitější datové manipulace se v nich popisují hůře. [8][9]

Detailní popis **barvených Petri sítí** a jejich **konstrukcí** obsahuje následující kapitola.

3 Barvené Petri sítě (Colored Petri Nets)

Barvené Petri sítě (dále **CPN**) byly vyvinuty na **Aarhus University** v **Dánsku** roku 1979. První verze **CPN** byla publikována v disertační práci **Kurta Jensena** roku 1981. Barvené sítě rozšiřují standardní **P/T Petri sítě** o [7][8]:

- možnost **diverzifikace** tokenů pomocí **množin barev** (**Color Sets**),
- přidání **inskripcí**, které umožňují aplikovat konstrukce programovacího jazyka (proměnné, funkce) na hrany a přechody,
- deklarování vlastních funkcí a proměnných pomocí jazyka Standard ML.

CPN je tedy kombinací klasických **P/T Petri sítí** a **programovacího** jazyka, což umožňuje modelovat složitější systémy na libovolném stupni abstrakce se zachováním dostatečné přesnosti vztahů. Mezi zastoupením programovacího a grafického jazyka je obecně nepřímá úměra, což umožňuje zkombinovat výsledný model podle požadavků na **přesnost**, **přehlednost** a **komplexnost**.

Součástí vývoje na dánské univerzitě je i vývojové prostředí (**IDE - Integrated Development Enviroment**), které umožňuje pracovat s **běžnými (P/T)**, **časovanými**, **hierarchickými** a **barvenými Petri** sítěmi. Jedná se o nástroj **CPN Tools**, který umožňuje **editaci**, **simulování** a **analyzování** (**verifikaci**) barvených **Petri** sítí. Dalšími možnostmi **CPN Tools** jsou [14]:

- generování stavového (částečného nebo úplného) prostoru (strom dosažitelnosti),
- analýza generovaného prostoru ve smyslu ohraničenosti, živosti atd.,
- úprava designu (barvy, popisky, vodící čáry),
- kontrola syntaxe kódu.

Instalační soubor nástroje **CPN Tools** je obsažen v příloze č. 1.

3.1 Deklarace datových struktur

Nástroj **CPN Tools** využívá jazyk **CPN ML**, který je rozšířením jazyka **Standard ML** [13]. Tento jazyk se využívá k deklarování datových struktur a samotné inskripci jednotlivých prvků sítě (míst, přechodů a hran). [14]

3.1.1 Deklarace množin barev

Množiny barev (Color Sets – dále **CS)** v **CPN Tools** reprezentují analogii k datovým typům z klasických programovacích jazyků. CS se dělí na jednoduché a složené. Přehled jednoduchý CS obsahuje Tabulka 3. Každý identifikátor CS může obsahovat libovolnou sekvenci **znaků**, **čísel**, **podtržítek** a **apostrofů**. Identifikátor však musí **vždy začínat písmenem** (malým či velkým). [14]

Tabulka 3 - Jednoduché množiny barev. Zdroj [15]

Název	Označení	Příklady
Celá čísla (Integers)	int	-2, -1, 0, 1, 3
Řetězce (Strings)	string	„ahoj“, „zbkd_dk“
Pravdivostní hodnoty (Booleans)	bool	true, false
Jednotka (Unit)	unit	()

Příklad definice deklarace standardních **CS** může vypadat následovně:

- **colset** CISLO = int;
- **colset** RETEZEC = string;
- **colset** ANO_NE = bool;
- **colset** JEDNOTKA = unit;

CS typu **unit** reprezentuje bezrozměrnou veličinu, která nemá přímou interpretaci. U této CS je sledován pouze počet jejího výskytu. Jedná se v podstatě o náhradu bezrozměrného tokenu z klasických **P/T Petri** sítí.

Jednoduché **CS** typu **int** a **string** mohou být omezeny pomocí konstrukce **with**, za kterou následuje přípustný výčet hodnot, resp. interval. Příklad využití deklarace CS využitím konstrukce **with** může být například interval celých čísel 1 – 10, jehož definice by byla následující:

- **colset** INTERVAL = int **with** 1..10;

Podobným způsobem lze omezit interval znaků, který může obsahovat **CS** typu **string**:

- **colset** CAPITALS = string **with** „A“..“Z“;

Speciálním využitím jednoduchých **CS** jsou **enumerativní CS**, které umožňují deklarovat výčet hodnot (např. dny v týdnu, názvy měsíců apod.). Hodnoty výčtu mohou obsahovat jakýkoli alfanumerický řetězec (splňující podmínky pro psaní identifikátorů):

- **colset** STATUS = **with** POVOLENO |ZAMITNUTO|NEROZHODNUTO;

Složené CS jsou definovány na **základě jednoduchých CS** pomocí konstruktorů [15]:

- product,
- record,
- union,
- list,
- subset.

Použitím těchto konstruktorů je možné deklarovat vlastní složené CS, např.:

- **colset** CISLOxRETEZEC = **product** CISLO * RETEZEC;
- **colset** ZAZNAM = **record** poradi:CISLO * data:RETEZEC;
- **colset** PACKET = **union** Data:ZAZNAM + Ack:ACKPACK;
- **colset** PACKETS = **list** PACKET;

3.1.2 Deklarace proměnných

Proměnné v CPN Tools jsou uvozovány klíčovým slovem **var** a definicí **množiny barev** (viz předchozí kapitola). **Proměnné** jsou v **průběhu simulace** (aplikace přechodů) **inicializovány** na konkrétní hodnoty v závislosti na **kontextu**. [14]

Příkladem proměnné může být:

- **var** PROMENNA : CISLO;
- **var** PROMENNA2: ZAZNAM;

První proměnná je **CS** CISLO, resp. integer. Druhá proměnná (PROMENNA2) je komplexního typu, jelikož její typ je složená CS.

3.1.3 Deklarace konstant

Konstanty mohou obsahovat libovolný typ **CS** (unit, bool, int, string, tuple, list, record), bez přímé deklarace. Konstanty jsou uvozovány klíčovým slovem **val**. Příkladem konstanty mohou být:

- **val** DELKA = 10;
- **val** ROZMERY = [1,2,4];
- **val** STAV = [5, true, „xyz“];

Konstanty je vhodné používat např. v deklaracích **CS** a **funkcí**. Konstanty zajišťují **konzistentnost** deklarací, pokud se některé hodnoty vyskytují pravidelně. Příkladem mohou být parametry modelu, které mohou být definovány na jednom místě a kdykoli změněny podle potřeby. [14]

3.1.4 Deklarace funkcí

Funkce umožňují přenést **složitější** výrazy do **deklaračních** oblastí a tím **zjednodušit** čtení a údržbu modelu. Další výhodou funkcí je jejich **znuvupoužitelnost**. Funkce jsou deklarovány pomocí klíčového slova **fun**. Každá funkce může obsahovat libovolný počet vstupních parametrů, které jsou v těle funkce transformovány na výstupy (hodnoty, které funkce vrací). [15]

Funkce může obsahovat konstrukci let-in-end, která rozděluje tělo funkce na 2 částí [14]:

- za klíčovým slovem „**let**“ následují deklarace lokálních konstant či funkcí,
- za klíčovým slovem „**in**“ následují logické konstrukce a návratové hodnoty,
- klíčové slovo „**end**“ uvozuje konec funkce.

Příklady s použitím těchto konstrukcí znázorňuje např. Obrázek 33 v kapitole 5. Seznam nejpoužívanějších funkcí, které nabízí **CPN Tools** shrnuje Tabulka 4.

Tabulka 4 - Přehled základních funkcí CPN Tools. Zdroj [14]

Název funkce	Popis
e::l	připojí element „e“ jako hlavu seznamu „l“
hd l	vrací hlavu seznamu „l“
tl l	vrací seznam „l“ bez hlavy
length l	vrací délku seznamu „l“
foldr f z l	vrací f(e1,f(e2,…,f(en,z) …), kde „l“ = [e1, e2, …, en]

3.2 Inskripce

Inskripce neboli **vpisování** proměnných, podmínek, funkcí, množin barev a dalších konstrukcí programovacího jazyka **CPN ML** na místa, přechody a hrany.

3.2.1 Inskripce místa

Každé místo v CPN Tools může obsahovat 3 typy inskripce [14]:

- **množinu barev** (povinná inskripce),
- **inicializační seznam tokenů** (volitelné),
- **název místa** (volitelné).

Obrázek 15 ilustruje příklad místa, kterému je přidělena CS **THREAD** a zároveň je pojmenováno „**Unstarted**“. Názvy jednotlivých míst musejí být unikátní.

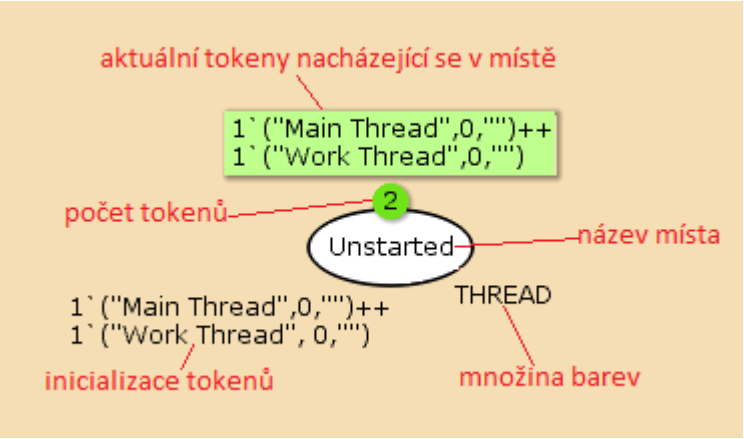
Obrázek 15 - Inskripce místa - Množina barev a název místa. Zdroj vlastní

Obrázek 16 ilustruje příkladovou deklaraci množiny barev **THREAD**, která je složenou CS z jednoduchých CS.

```
colset NAMEOFTHREAD = string;
colset TYPEOFTHREAD = int;
colset OTHERTHREAD = string;
colset THREAD = product NAMEOFTHREAD * TYPEOFTHREAD * OTHERTHREAD;
```

Obrázek 16 - Příklad deklarace množin barev. Zdroj vlastní

Nejdůležitější součást inskripce místa je samotná definice seznamu **tokenů**. Všechny tokeny musejí odpovídat deklaraci CS, např. musí obsahovat řetězec, integer a opět řetězec v tomto pořadí (viz Obrázek 16). Příklad místa se všemi typy inskripce znázorňuje Obrázek 17.

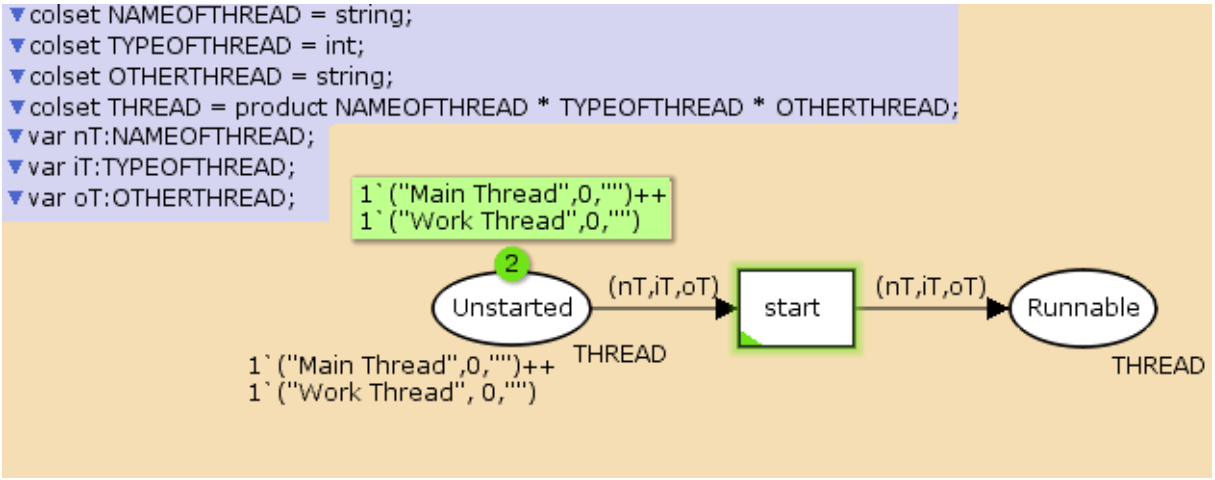


Obrázek 17 - Plná inskripce místa. Zdroj vlastní

Obsah každého tokenu začíná definicí počtu výskytů daného tokenu, tj. celé kladné číslo. Následuje znak „`“ a následně v závorce množina parametrů oddělených čárkami. Struktura parametrů **musí** odpovídat deklaraci **CS** (pozn. řetězce jsou psány v uvozovkách). [14]

3.2.2 Inskripce hrany

Hrana může obsahovat pouze jednu inskripci. Tato inskripce může obsahovat proměnné, které však musejí odpovídat CS v místě, z kterého vychází, resp. do kterého míří. Proměnné mohou nabývat i typů nižšího řádu u složených CS, avšak musejí zachovat pořadí, CS a počet členů struktury této nižší úrovně. Obrázek 18 ilustruje inskripci orientovaných hran jako rozšíření příkladu z kapitoly 3.2.1. [14]



Obrázek 18 - Inskripce orientovaných hran. Zdroj vlastní

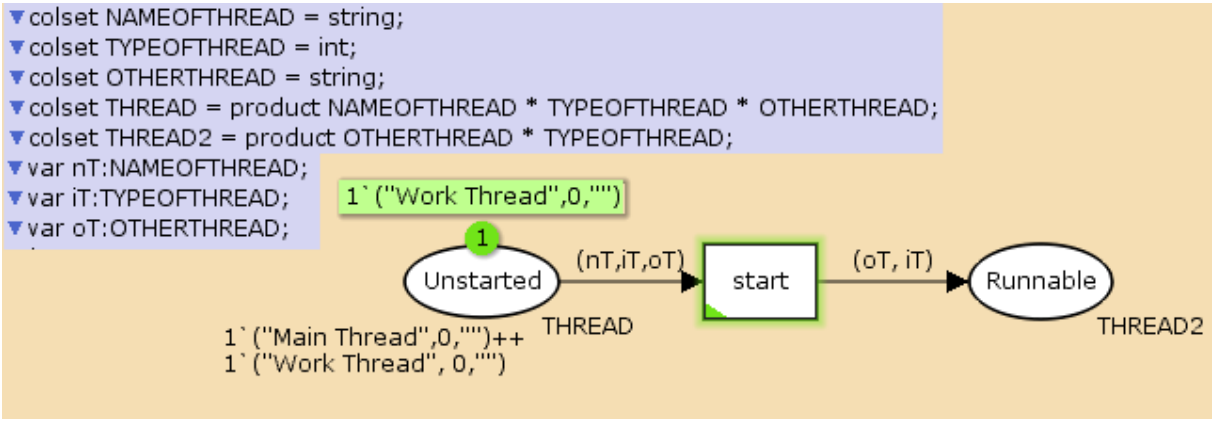
Z obrázku vyplývá, že každá hrana obsahuje 3 proměnné:

- „**nT**“ přenáší první parametr tokenu CS **THREAD**, tj. CS **NAMEOFTHREAD**,
- „**iT**“ přenáší druhý parametr tokenu CS **THREAD**, tj. CS **TYPEOFTHREAD**,
- „**oT**“ přenáší třetí parametr tokenu CS **THREAD**, tj. CS **OTHERTHREAD**.

Jelikož místo „**Runnable**“ je stejného typu jako místo původní („**Unstarted**“), musí inskripce hrany vycházející z přechodu obsahovat stejnou strukturu proměnných jako hrana vstupující do přechodu. Pokud by bylo druhé místo jiného typu než první, musela by se provést úprava proměnných, popř. pouhá restrukturalizace logiky.

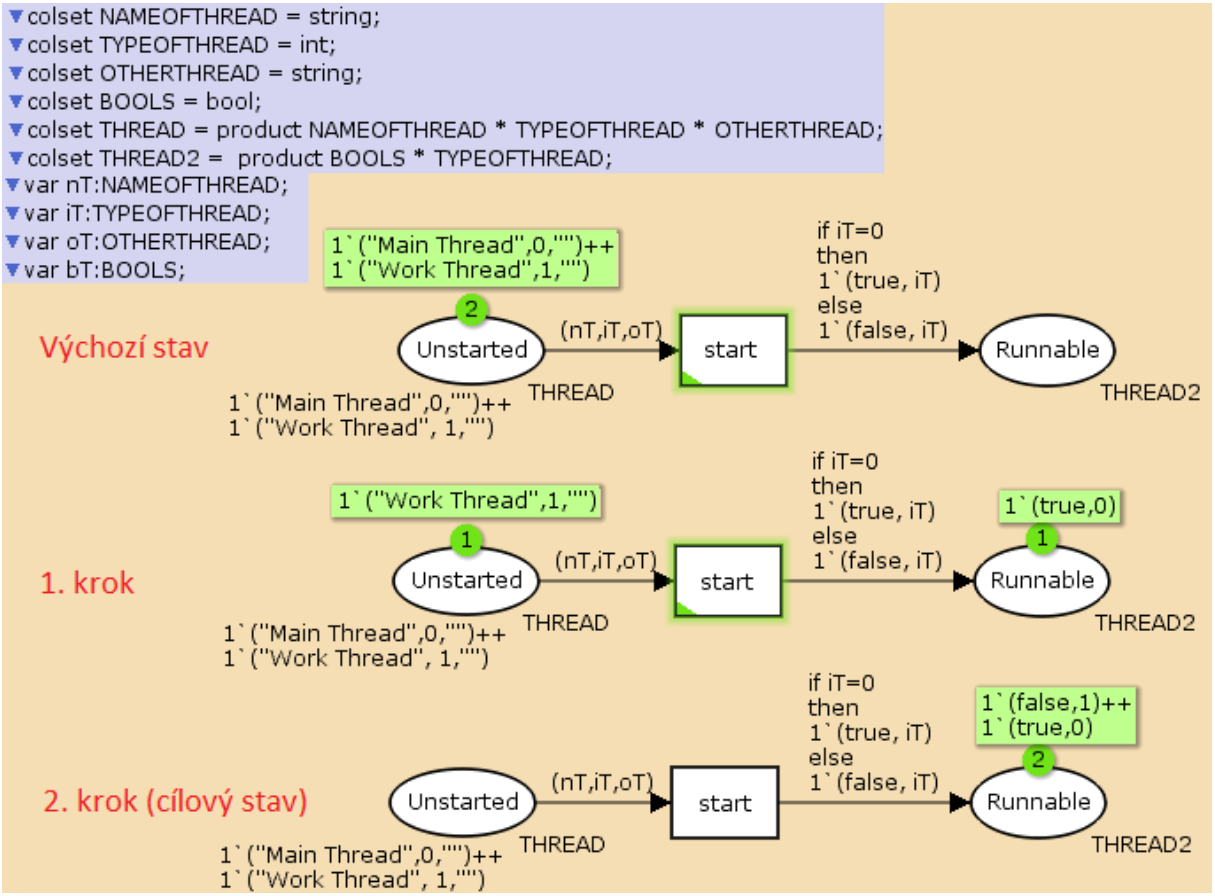
Obrázek 19 znázorňuje příklad kdy druhé místo je jiného typu než první. Případná restrukturalizace spočívá v redukci, či prohození jednotlivých proměnných. Pokud je cílové místo značně jiného typu, tj. obsahuje CS, které se ve výchozím místě nevyskytují, je třeba dané proměnné přetypovat, popř. vytvořit zcela nové hodnoty pro odpovídající proměnné (např. pomocí funkce nebo podmínky).

Obrázek 20 ilustruje proces přechodu tokenů z místa „**Unstarted**“ do místa „**Runnable**“. Jelikož cílové místo je jiného typu a zároveň obsahuje CS, která není ve výchozím místě obsažena (**boolean**). Z tohoto důvodu je na hraně vedoucí z přechodu definována podmínka, která přiřadí této nové logické proměnné hodnotu **true**, pokud je hodnota proměnné „**iT**“ rovna **0**. Pokud je hodnota proměnné „**iT**“ různá od **0** je automaticky nastavena logická proměnná na **false**. Obrázek dále vykresluje přechod a transformaci tokenů z výchozího do cílového místa.



Obrázek 19 - Inskripce orientovaných hran - Restrukturalizace. Zdroj vlastní

V prvním kroku je přenesen token **("MainThread", 0, "")** přes přechod **„start“** a následně transformován předem zmíněnou podmínkou na nový token **(true, 0)**. Tento nový token si zachovává pouze třetinovou informaci z původního tokenu a navíc vytváří novou. Ve druhém kroku je převeden druhý token **("WorkThread", 1, "")** stejným způsobem jako v kroku 1. Po této operaci nastal cílový stav, jelikož všechny tokeny byly přeneseny a žádný další přechod není aktivní.



Obrázek 20 - Inskripce orientovaných hran - Restrukturalizace podmínkou. Zdroj vlastní

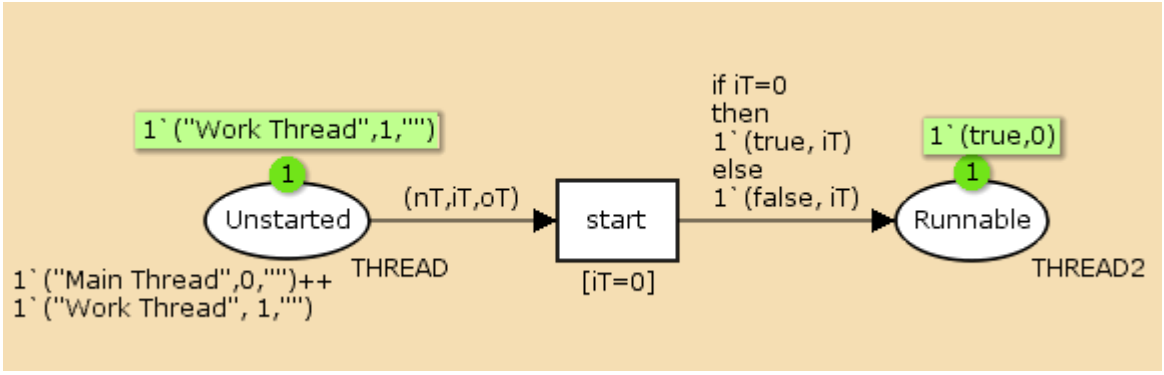
3.2.3 Inskripce přechodu

Každý přechod může obsahovat 5 různých typů volitelných inskripcí. **První** inskripcí je **pojmenování** daného přechodu, které slouží pouze jako mnemotechnická pomůcka stejně jako pojmenování místa. [14]

Druhým typem je inskripce **Guard** neboli **strážní podmínka**, která slouží jako zámek přechodu. Pokud je strážní podmínka splněna, přechod je aktivní, jinak nikoliv. Testovat se mohou libovolné hranové proměnné pomocí následujících syntaktických elementů [14]:

- je rovno (**=**),
- není rovno (**<>**),
- menší nebo rovno (**<=**),
- větší nebo rovno (**>=**),
- menší (**<**),
- větší (**>**),
- a také (**andalso**),
- nebo jiné (**orelse**).

Obrázek 21 znázorňuje využití strážní podmínky na přechodu „**start“.** Z obrázku je patrné, že tato síť je mrtvá, jelikož strážní podmínka **nepropustí** žádné jiné tokeny, než ty, jejichž hodnota druhého atributu je rovna **nule**.



Obrázek 21 - Inskripce přechodu - Strážní podmínka. Zdroj vlastní

Strážní podmínka lze konstruovat i pomocí **if-then-else** konstrukce, avšak musí být zajištěno, aby část **then** a část **else** obsahovala (vracela) pouze logické hodnoty (boolean). Např. podmínka „**[if x=5 then y=6 else z=7]**“ je validní, ale podmínka **„[if x=5 then 6 else 7]“** nikoliv, jelikož bloky **then** a **else** nevrací **boolean**, ale **integer**.

Třetí inskripce umožňuje nadefinovat zpoždění daného přechodu. Zpoždění je uvozováno textem **@+**, za který je možné dosadit celé kladné číslo, které reprezentuje hodnotu daného zpoždění. Lze také použít **if-then-else** konstrukci, která vrací kladný **integer**, popř. využít vlastní nadefinované funkce (která také vrací kladný integer). Pokud není zpoždění explicitně nadefinováno je výchozí hodnota rovna **nule** (**@+0**). [14]

Čtvrtou inskripcí u přechodu je možnost vložit celý kus vlastního kódu, který je uvozován deklarací vstupů a výstupů. Tato inskripce má strukturu [15]:

- **input(),**
- **output(),**
- **action().**

Blok **input()** je volitelný a může obsahovat **vstupní** hodnoty, které vstupují do bloku **action()**. Blok **input()** může obsahovat v závorce **n-tici** vstupních proměnných, např. input(a,b,c,d), kde proměnné a, b, c, d vstupují do bloku **action**() jako vstupy.

Blok **output()** je **obdobou** bloku **input()**, tj. také je volitelný a také může obsahovat n-tici **výstupních** hodnot.

Blok **action()** obsahuje jakýkoli řídící kód v jazyku Standard ML, který je vykonán pokaždé, když je daný přechod aktivován. [14]

Posledním typem inskripce je priorita daného přechodu. Priorita je vyjádřena celým kladným číslem. Pokud

3.3 Hierarchické strukturování CPN

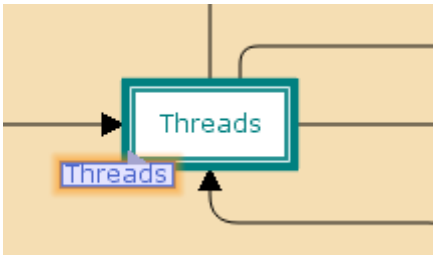
CPN Tools umožňuje využít **hierarchickou abstrakci** pro **zjednodušení** sítě. Tím je možné rozdělit model do **modulů**, resp. **submodelů**. Každý modul může obsahovat další submodel atd. Moduly mají následující význam [8]:

- umožňují využít **re-use** u často se opakujících struktur,
- umožňují **rozdělit** složitý model na snadno pochopitelné a řiditelné části se správně definovaným rozhraním.

CPN, které využívají hierarchickou strukturu, se nazývají **hierarchické barvené Petri sítě** (**HCPN**). [15]

3.3.1 Tvorba modulu

Každý modul je v modelu reprezentován pomocí speciálního **přechodu**, který se v síti chová jako standardní přechod (je mezi dvěma místy). Obrázek 22 znázorňuje příklad přechodu, který tvoří samostatný modul do/z kterého vedou klasické hrany z klasických míst.



Obrázek 22 - Hierarchické CPN. Zdroj vlastní

3.3.2 Definice rozhraní

Uvnitř modulu (subsystému) je nutné vůči nadřazené síti specifikovat rozhraní. Rozhraním jsou v modulu místa, která do modulu v nadřazené síti směřují, resp. z něho vycházejí. Vstupní místa musejí být v modulu označena jako „**In**“, výstupní místa jako „**Out**“ a vstupně výstupní místa jako „**I/O**“. [8][14]

Příklad na hierarchickou abstrakci CPN je možné nalézt v kapitolách 4.2.2 a 6.3.

3.4 Časované CPN

Časované CPN umožňují oddělit **reálny** čas od **simulačního** času. **Reálný** čas vnímá člověk, který **pracuje** s daným modelem, tj. provádí simulaci. Naopak **simulačním** časem je možné **zohlednit časovou náročnost** jednotlivých přechodů, popř. načasovat jednotlivé tokeny podle potřeby. Je to tedy pouze symbolická interpretace časových závislostí mezi jednotlivými stavy sítě. [14]

3.4.1 Časování tokenů

Každému tokenu může být přiděleno číslo (přirozené nebo reálné číslo), které je nazýváno **časová značka**. Pokud token obsahuje takovouto značku je označován jako **časovaný token** (**timed token**) a musí být sdružený s časovanou CS. Časovanou CS lze vytvořit pomocí klíčového slova „**timed**“ za deklaraci dané CS (viz Obrázek 23).

```
colset timedINT = int timed;
```

Obrázek 23 - Deklarace časované CS. Zdroj vlastní

3.4.2 Časový čítač

Simulátor **CPN** obsahuje **čítač**, který zobrazuje **aktuální simulační čas**. Časovaný token není dostupný, pokud obsahuje časovou značku, která je větší než aktuální **simulační** čas (**časový čítač**). Jestliže taková situace nastane a žádný jiný přechod není aktivní, musí časový čítač inkrementovat čas na časovou značku časově nejbližšího tokenu a tím umožnit jeho přechod. [14]

Časový čítač může využívat simulační čas typu „**real**“ nebo **diskrétní** simulační čas (**integer**). Pokud je jeden z těchto časů zvolen, celá síť bude obsahovat pouze časové značky daného typu (nelze kombinovat). [8]

3.4.3 Časování hran

Každou časovanou **CS** lze na hraně inkrementovat o libovolné celé kladné číslo. Toho lze docílit pomocí klíčového výrazu **@+** přidaného za **inskripci hrany** a doplněním požadovaného času za plus. Při časování hran je možné využít podmínky, proměnné či funkce. Toto však platí pouze pro výstupní hrany. Vstupní hrany představují preemptivní posun času. Časová značka tokenu v místě, z kterého vede hrana s preemptivním posunem je aktivní dříve (o hodnotu preemptivního posunu). [14]

3.4.4 Časování přechodů

Poslední možností zohlednění času v **CPN** je časování **přechodů**. Časování přechodů umožňuje využít uživatelsky definované funkce, **random** funkci, časové funkce a distribuční funkce. [14]

3.4.5 Časové funkce

Časové funkce umožňují zpřístupnit simulační čas v průběhu simulace. Tím může být využit k tvorbě logických podmínek, transformován do proměnných apod. Nejdůležitější funkcí je výraz **time()**, který vrací hodnotu aktuálního času simulace. [14]

3.5 Verifikace modelu CPN

CPN Tools umožňuje vygenerovat stavový prostor sítě a pomocí něj verifikovat různé předpoklady kladené na model. Stavovým prostorem se rozumí množina všech možných stavů sítě a jejich propojení. CPN Tools umožňuje generování jak plného, tak částečného stavového prostoru. Součástí generování stavového prostoru je i jeho automatická analýza, která vyhodnocuje standardní vlastnosti, kladené na Petri sítě (některé již částečně definované v kapitole 2.3) [8][15]:

- Dosažitelnost (**Reachibility**) – Nad vygenerovaným stavovým prostorem, lze provádět dotazování, ve smyslu hledání cesty mezi dvěma stavy sítě. Pokud mezi stavem „A“ a stavem

„B“ existuje cesta po směru orientovaných hran, znamená to, že stav „B“ je dosažitelný ze stavu „A“ (opačně to však platit nemusí).

- Ohraničenost (**Boundedness**) – U všech míst v síti je sledován maximální a minimální výskyt tokenů.
- Domácí značení (**Home Properties**) – Reprezentuje stavy sítě, které jsou dosažitelné z kteréhokoliv dosažitelného stavu sítě.
- Živost (**Liveness**):
    - Mrtvé značení (**Dead Markings**) – Pokud síť (stav sítě) neobsahuje žádné aktivní přechody, jedná se o tzv. mrtvou síť, tj. rozmístění tokenů v takovéto síti se nazývá mrtvé značení.
    - Mrtvé přechody (**Dead Transitions**) – Jestliže některý přechod neuvozuje změnu ve stavovém prostoru všech stavů, tj. nikdy nemůže být aplikován, označuje se jako mrtvý přechod.
    - Živé přechody (**Life Transitions**) – Pokud existuje přechod, jehož aktivace je dosažitelná z jakéhokoli stavu sítě, je označen jako živý přechod.
- Spravedlivost (**Fairness**)
    - **Impartial** – Přechod lze označit jako „**Impartial**“ pokud je součástí cyklu ve stavovém prostoru.

Některé z výše zmíněných vlastností budou použity v kapitole 8, která se zabývá verifikací jednotlivých modelů navržených v následujících kapitolách.

4 Základní činnosti vláken

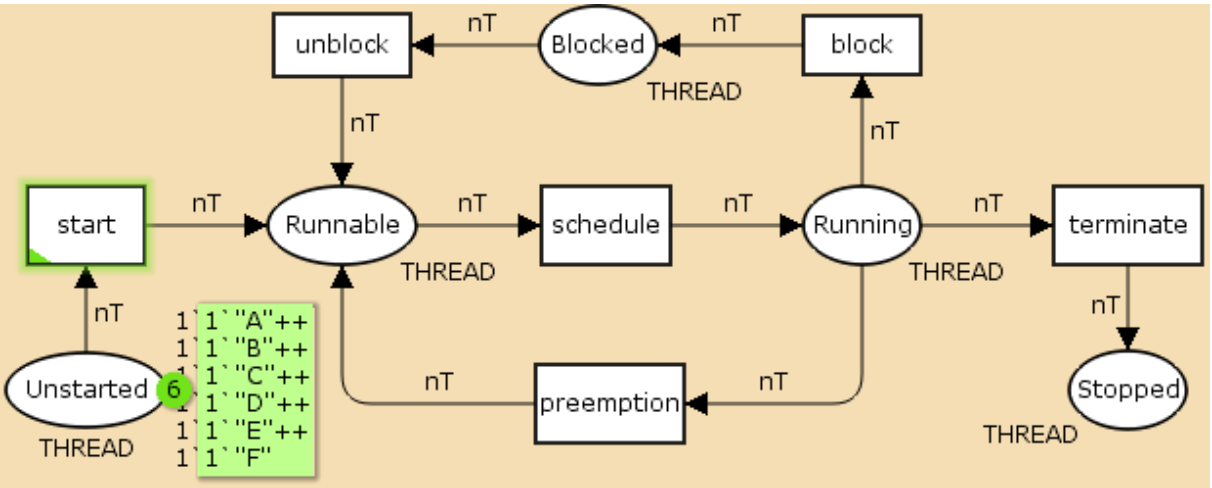
Se znalostí základních konstrukcí **CPN** je nyní možné lépe popsat jednotlivé problematické oblasti spojené s činností vláken v OS. Na činnosti vláken lze nahlížet třemi odlišnými pohledy:

- stavový pohled,
- behaviorální pohled,
- kontextuální pohled.

Všechny tyto pohledy budou probrány v této a následujících třech kapitolách a budou modelovány pomocí **CPN**, resp. **CPN Tools** s využitím výchozího stavového modelu vláken.

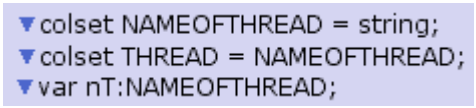
4.1 Stavový model vláken

V první kapitole byl vysvětlen základní procesní stavový model. Stavový model vláken je s tímto modelem ekvivalentní (viz Obrázek 4 v kapitole 1.4.2), jelikož vlákno je v podstatě ta část procesu, která se stará o výpočetní záležitosti (komunikuje s procesorem). V kapitole 3 byl při vysvětlování základů CPN použit příklad, který bude nyní upraven a rozšířen do základního stavového modelu vláken (viz Obrázek 24). Základní stavový model je obsažen v příloze č. 2.



Obrázek 24 - CPN - Stavový model vláken. Zdroj vlastní

Deklarace **CS** a **proměnných** odpovídajících tomuto modelu je znázorněna na Obrázek 25. Model je složen z 5 míst CS typu **THREAD**, jež obsahuje jeden parametr **NAMEOFTHREAD**, který je CS typu **string**. Každá hrana modelu nese proměnnou **nT**, která přenáší již zmíněný parametr vlákna mezi místy.



Obrázek 25 - CPN - Stavový model vláken - Deklarace. Zdroj vlastní

Ze základního stavového modelu vláken vyplývají následující otázky:

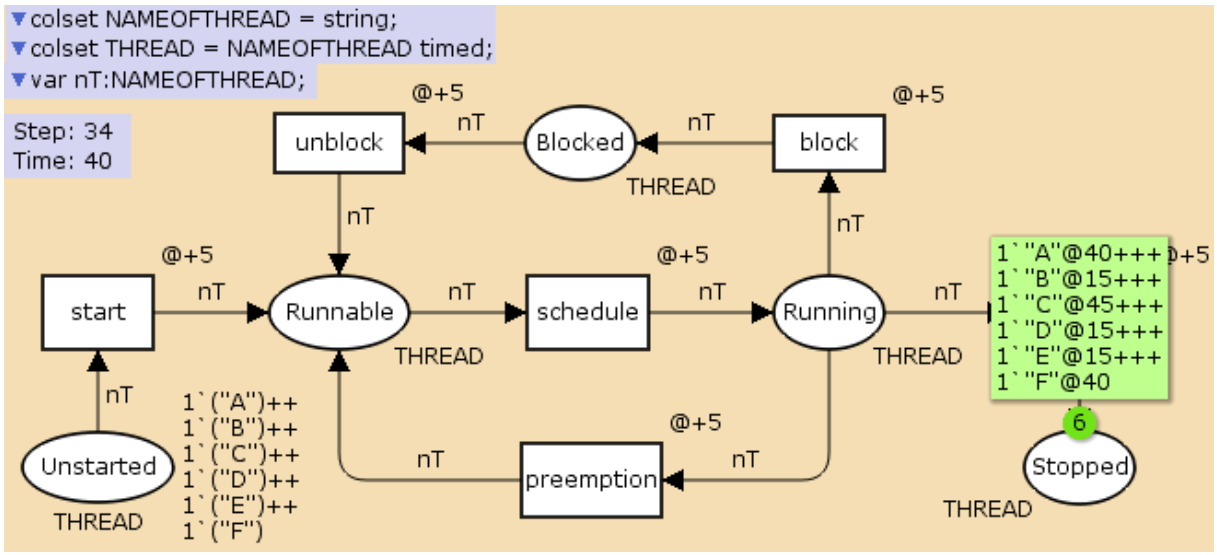
- **Proč došlo ke změně stavu?**
- **Kdy došlo ke změně stavu?**
- **Jakým způsobem byla změna provedena?**

Možné odpovědi na první otázku shrnuje Tabulka 5. Všechny tyto příčiny (kromě rozhodování plánovače - scheduler) jsou v reálném světě přímo závislé na konkrétní situaci. Jsou tedy ve smyslu modelování náhodnou veličinou s určitým rozdělením pravděpodobnosti s hlediska času a logickou podmínkou s hlediska kauzality. Kauzalita jednotlivých změn je podrobněji probrána v kapitole 6.

Tabulka 5 - Možné příčiny přechodů mezi stavy v modelu vláken. Zdroj vlastní

Název přechodu	Možné příčina aktivace
start	- vlákno bylo spuštěno, jelikož byl spuštěn proces, - jiné vlákno vytvořilo a poté spustilo toto vlákno.
schedule	- scheduler tak rozhodl
preemption	- scheduler tak rozhodl
block	- vlákno začalo čekat na nějakou vstupně výstupní operaci - vlákno bylo zablokováno jiným vláknem - vlákno se zablokovalo samo
unblock	- nastala vstupně vstupní operace - jiné vlákno toto vlákno odblokovalo - uplynul čas, na který bylo vlákno zablokováno
terminate	- vlákno dokončilo všechnu svoji práci - jiné vlákno ukončilo toto vlákno

Zařazení časového hlediska do modelu lze odpovědět na druhou otázku (Kdy došlo ke změně stavu?). Pro tento účel lze využít časované CPN, které umožňují **generovat náhodné** časy s předem definovaného **rozdělení pravděpodobnosti**. Lze samozřejmě použít čistě **deterministicky** zvolené časy, které jsou např. typu **integer**. Obrázek 26 ilustruje možný stav (konečný stav) časované sítě po **34 krocích** simulace, která se nachází v simulačním **čase** 40. Z obrázku je patrné, že každý přechod byl deterministicky ohodnocen časovým zpožděním s hodnotou **5** (**@+5**). Takovýmto způsobem lze nyní empiricky volit různá zpoždění a tím zvýšit vypovídací schopnost modelu. Model znázorněný na Obrázek 26 je obsažen v příloze č. 3. Využití náhodně generovaných časů s různými rozděleními pravděpodobností je použito v následující kapitole věnované plánování (**scheduling**).



Obrázek 26 - CPN - Časovaný stavový model vláken. Zdroj vlastní

Poslední otázka kladená na stavový model vláken (Jakým způsobem byla změna provedena?) zkoumá důvod, proč daná změna proběhla, tak jak proběhla. Tato otázka se týká především přechodů „**schedule**“ a „**preemption**“, které řídí systémový **scheduler**.

4.2 Typy vláken

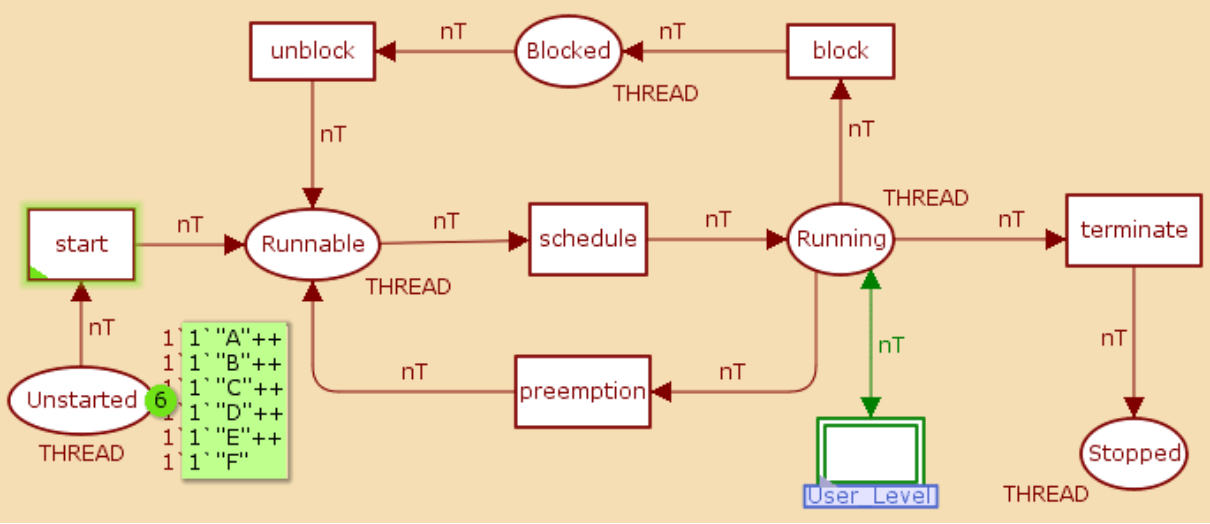
Z hlediska modelování je nutné odlišit různé způsoby implementace vláken. V jistých směrech se mohou jednotlivé typy vláken jevit jako ekvivalentní, proto je nutné předem specifikovat jejich rozdílnou interpretaci při různých úrovních abstrakce a úhlech pohledu.

4.2.1 Vlákna na úrovni jádra

Model na Obrázek 24 lze považovat za typový přiklad vláken na úrovni jádra, jelikož zde může vlákno nabývat stavů „**Runnable**“ a „**Running**“. Tento fakt, je důležitý, jelikož tyto dva stavy determinují použití **multiprogramování** (přechod „**schedule**“), resp. **sdílení času** (přechod „**preemption**“). Vlákna jsou v tomto modelu přidělována procesoru/jádru podle plánovacího algoritmu OS (**scheduler**).

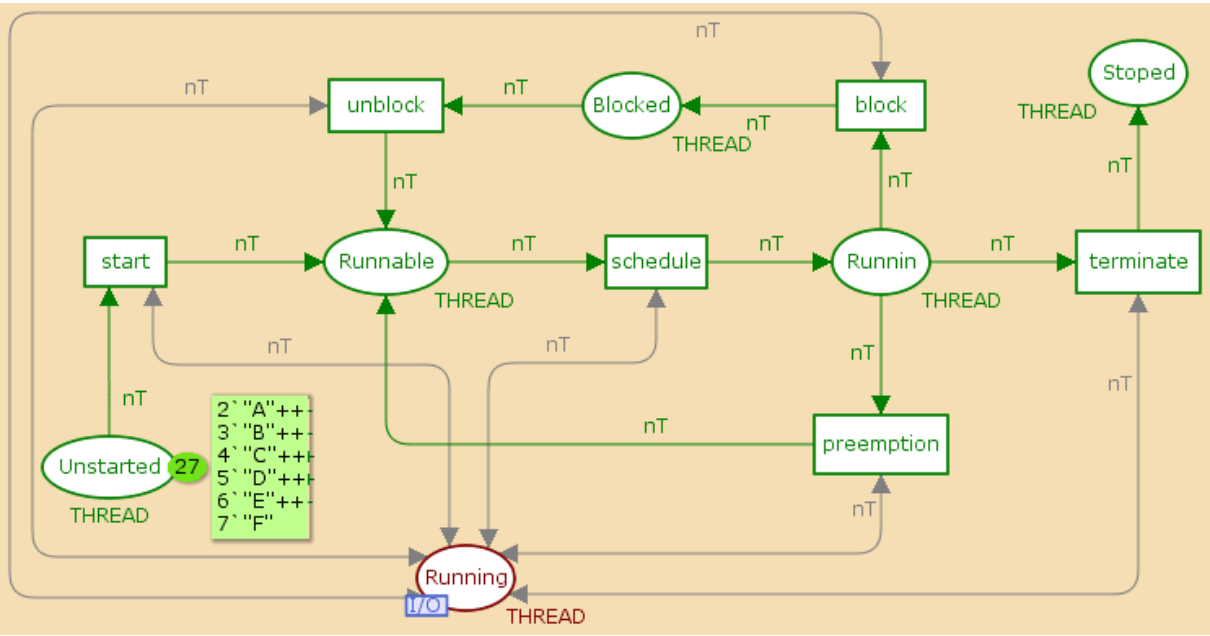
4.2.2 Vlákna na uživatelské úrovni

U vláken na uživatelské úrovni je situace poněkud komplikovanější, jelikož závisí na konkrétním případu využití. Na některých OS mohou být uživatelská vlákna mapována **1:1** (jde tedy o kernelová vlákna), na jiných OS mohou být čistými uživatelskými vlákny (mapování **N:1**) a někdy mohou být implementována jako tzv. hybridní vlákna (mapování **N:M**). Obrázek 27 ilustruje základní stavový model vláken na úrovni jádra s drobnou úpravou. Speciální přechod, který je vyobrazen zelenou barvou, reprezentuje modul (submodel) uživatelských vláken. Samotný modul obsahuje téměř ekvivalentní strukturu s modelem vláken na úrovni jádra (viz Obrázek 28).



Obrázek 27 - CPN - Vlákna na uživatelské úrovni - Základní síť

Hlavní odlišností je samotná interpretace jednotlivých stavů (míst) a přechodů mezi nimi. První významným rozdílem je fakt, že vlákna na uživatelské úrovni mohou měnit své stavy, pouze pokud je jejich odpovídající vlákno na úrovni jádra ve stavu/místě „**Running**“. Toto je zajištěno pomocí obousměrných hran, které vedou z místa „**Running**“ do všech přechodů v modulu „**User-Level**“. Obrázek 28 dále znázorňuje fakt, že každé vlákno na úrovni jádra obsahuje několik uživatelských vláken (např. vlákno A obsahuje dvě vlákna na uživatelské úrovni apod.). Z tohoto vyplývá, že se jedná o implementaci typu **N:M**, tj. hybridní vlákna.

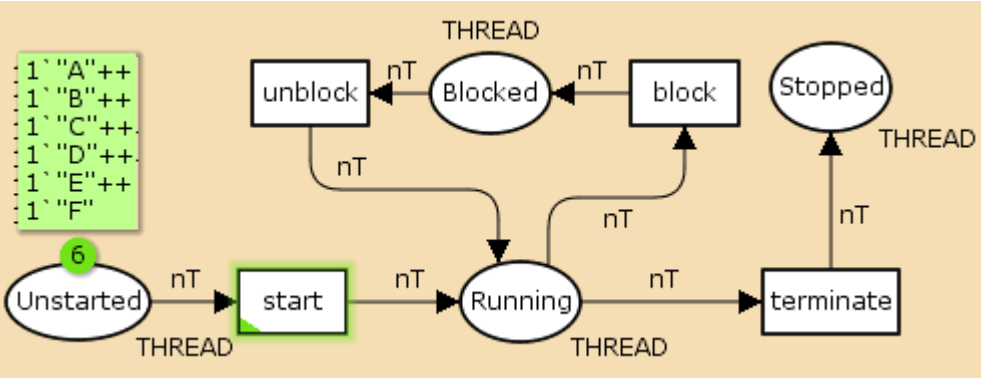


Obrázek 28 - CPN - Vlákna na uživatelské úrovni - User-Level. Zdroj vlastní

Jednotlivé přechody a místa v modulu „**User-Level**“ představují logické stavy uživatelských vláken (a jejich změny), které řídí plánovač (**scheduler**) vláknové knihovny dané implementace. Pokud se tedy

např. uživatelské vlákno nachází ve stavu „**Runnin**“ neznamená to, že je mu skutečně přidělen **procesor/jádro** apod. Pokud je kernelové vlákno ukončeno (přejde přes přechod „**terminate**“), ukončí se i všechna jeho uživatelská vlákna, bez ohledu na jejich stav. Model vláken na uživatelské úrovni je obsažen v příloze č. 4.

V následujících kapitolách budou **uvažována pouze** vlákna na úrovni jádra, resp. uživatelská vlákna s mapováním **1:1**. V některých situacích bude navíc úmyslně abstrahováno od přechodů „**schedule**“ a „**preemption**“. Obrázek 29 reprezentuje model, který abstrahuje od samotného přidělování procesoru/jádra. Z tohoto modelu vyplývá skutečnost, že pokud je vlákno v místě „**Running**“ je z **uživatelského** hlediska aktivní.



Obrázek 29 - Zjednodušený stavový model. Zdroj vlastní

Tento model bude využit v kapitole 6 věnované synchronizaci.

5 Scheduling

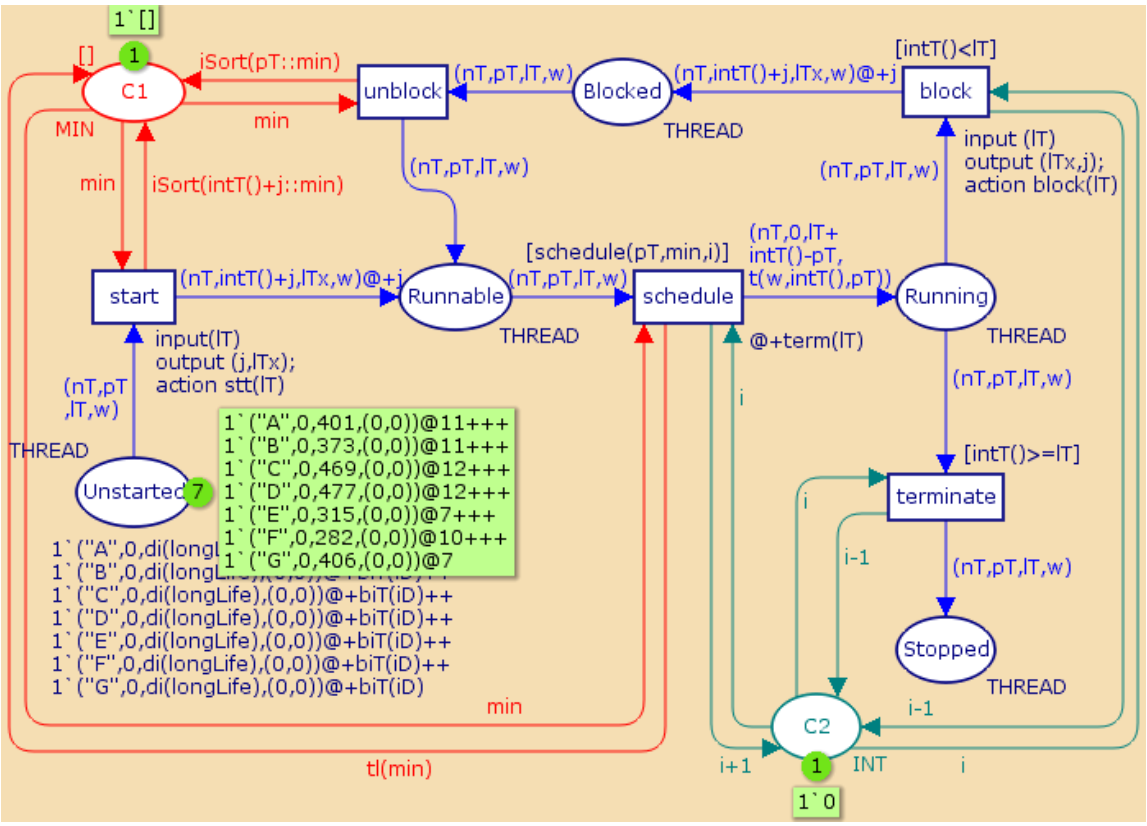
Scheduling, resp. **plánování vláken** je jedna z mála oblastí, která lze věrně namodelovat, jelikož je založeno na řadě primitivních algoritmů, které řídí přidělování a odebírání vláken procesoru/jádru. Jednotlivé algoritmy jsou modelovány s uvažováním vláken na úrovni jádra.

5.1 First-Come First-Served (FCFS)

Nejzákladnějším algoritmem je **First-Come First-Served** (dále **FCFS**), který reprezentuje klasickou **FIFO** (First In First Out) frontu. Jedná se o **nepreemptivní** algoritmus, který řadí příchozí vlákna do místa (stavu) „**Runnable**“ podle jejich příchozího času. Při přidělování procesoru jsou poté prioritně upřednostněna vlákna, která přišla do fronty první.

Základní model plánovacího algoritmu **FCFS** je zobrazen na Obrázek 30 (model, viz příloha č. 5). Model je barevně rozlišen do 3 částí:

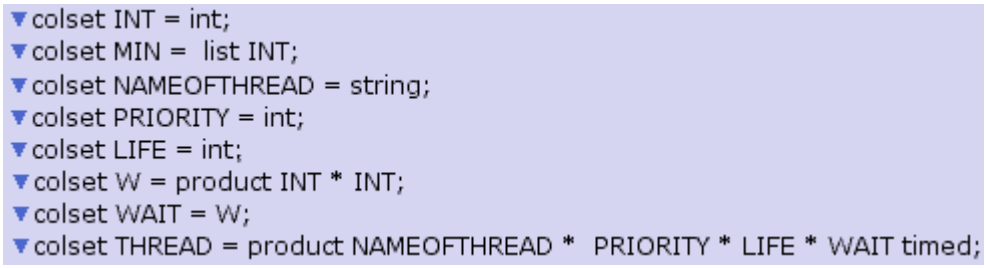
- **modrá** část představuje klasickou **nepreemptivní stavovou strukturu** (přechody = modrá barva, místa = tmavě modrá barva),
- **červená** část realizuje **frontu** s časy příchodů seřazenou vzestupně,
- **tmavě zelená** část reprezentuje kontrolní blok **řídící přidělování procesorů, resp. jader**.



Obrázek 30 - CPN - Scheduling - FCFS - Výchozí stav. Zdroj vlastní

5.1.1 Deklarace struktur modelu algoritmu FCFS

Každá část modelu má svou vlastní **CS** místa. Obrázek 31 znázorňuje deklaraci všech **CS** v tomto modelu.



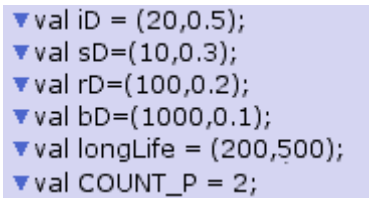
Obrázek 31 - CPN - Scheduling - FCFS - Deklarace CS. Zdroj vlastní

Místo „**C2**“ nese CS **INT**, která je typu **integer**. Místo „**C1**“ nese CS **MIN**, která je typu **list** CS **INT**. Ostatní místa jsou stejného typu CS **THREAD**, která je složena ze tří parametrů, a to **NAMEOFTHREAD** (typu **string**), **PRIORITY** (typu **integer**), **LIFE** (typu **integer**) a **WAIT** (typu CS **W** s dvěma parametry typu **integer**). Zároveň je CS **THREAD časovaná**.

Jak již z deklarace parametrů vlákna vyplývá, je každé vlákno definováno svým názvem (**NAMEOFTHREAD**), prioritou (**PRIORITY** - v tomto případě čas příchodu do místa „**Runnable**“), dobou, která musí být „**odpracována**“ než může vlákno přejít do stavu „**Stopped**“ (**LIFE**) a parametrem **WAIT**, který shromažďuje **statistické údaje** při **simulaci** (**doba čekání** vlákna a **počet přidělení** procesoru/jádru).

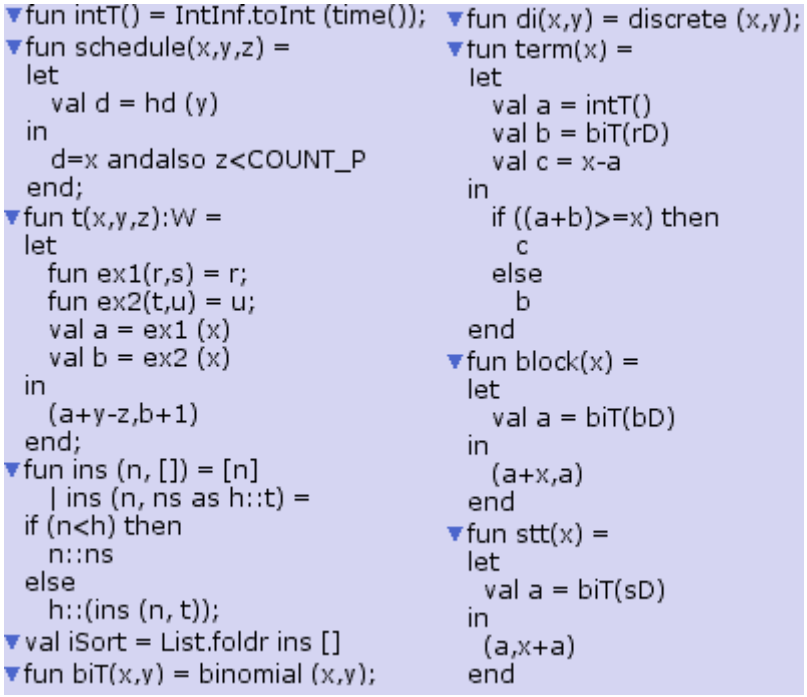
Důležitou součástí modelu jsou jeho **simulační parametry** (konstanty), které ilustruje Obrázek 32. Většina těchto konstant slouží jako **vstup** pro funkce generující náhodná čísla podle konkrétního rozložení pravděpodobnosti. Jediná konstanta, jež neslouží jako vstupní parametr funkce je **COUNT_P**, která představuje počet procesních jednotek (procesorů nebo jader). Ostatními konstantami jsou:

- **iD** (initDelay) – specifikuje zpoždění při vytvoření vlákna, resp. čas kdy vlákno bude připraveno ke spuštění,
- **sD** (startDelay) – specifikuje zpoždění při startu, resp. doba mezi okamžikem vytvoření vlákna a jeho spuštění,
- **rD** (runDelay) – specifikuje délku běhu vlákna, resp. čas, po jehož uplynutí bude vlákno zablokováno,
- **bD** (blockDelay) – specifikuje dobu, po kterou bude vlákno zablokováno,
- **longLife** – specifikuje požadovaná doba práce vlákna.



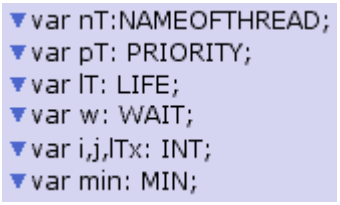
Obrázek 32 - CPN - Scheduling - FCFS - Deklarace konstant. Zdroj vlastní

Konstanty „**iD**“, „**sD**“, „**rD**“ a „**bD**“ jsou používány jako **parametry** funkce „**biT**“ (viz Obrázek 33), která vrací náhodné celé číslo s **binomického** rozdělení pravděpodobnosti. Konstanta **longLife** je využita jako parametr funkce „**di**“, která vrací náhodné celé číslo s **diskrétního** rozdělení pravděpodobnost. Funkce „**bi**“ je v modelu použita pouze jednou a to pro počáteční inicializaci délky práce (parametr **LIFE** vlákna) vláken. Ostatní funkce slouží pro správu přechodů z **hlediska času** a **aktivace** a budou podrobněji probrány v následující subkapitole věnované popisu činnosti modelu.



Obrázek 33 - CPN - Scheduling - FCFS - Deklarace funkcí. Zdroj vlastní

Poslední důležitou deklarací pro počáteční znalost modelu jsou proměnné (viz Obrázek 34). Proměnné „**nT**“, „**pT**“ a „**lT**“ na sebe váží odpovídající parametry vlákna při přechodu mezi místy (stavy).



Obrázek 34 - CPN - Scheduling - FCFS - Deklarace proměnných. Zdroj vlastní

Pomocné proměnné „**i**“, „**j**“ a „**lTx**“ jsou CS typu **INT**. Proměnná „**i**“ je v modelu použita jako čítač (**counter**) stavu zpracovávaných vláken v místě „**C2**“. Proměnná „**j**“ slouží ke zpracovávání časových zpoždění pro potřeby místa „**C1**“. Proměnná „**lTx**“ je ekvivalence k proměnné „**lT**“. Slouží pro **odlišení** vstupů a výstupů v kódových segmentech přechodů „**start**“ a „**block**“.

Jedinou strukturovanou proměnnou v modelu je proměnná „**min**“, která na sebe váže obsah místa „**C1**“ (seznam příchozích časů).

5.1.2 Popis činnosti vláken v modelu algoritmu FCFS

Jak již bylo zmíněno v kapitole 4, je nutné definovat **kdy**, **jak** a **proč** se uskutečnil daný přechod mezi dvěma místy sítě. Z definice časovaných CPN vyplývá, že token, jehož časová značka je větší než simulační čas, **není** v místě **aktivní** (není s ním možné pracovat). Tento fakt je důležitý při **definování významu časové značky** tokenu v místě. Obrázek 30 ilustruje situaci, kdy **7 vláken** (tokenů) je umístěno v místě „**Unstarted**“. Každé vlákno má přiřazeno náhodně generované čísla z **binomického** rozdělení pravděpodobnosti s parametry konstanty „**iD**“, tj. jedná se o hodnoty z rozsahu **0-20** se střední hodnotou **10**. Jelikož minimální značka má hodnotu **7** je simulační čas přizpůsoben této hodnotě (do této doby nelze provést žádnou akci) a umožní provést přechod vláken „**E**“ a „**G**“ do stavu „**Runnable**“.

Při průchodu přes přechod „**start**“ jsou vykonány následující akce:

- Vykoná se funkce „**sttr**“ s parametrem „**lT**“ – tato funkce vrací dvojici hodnot, které jsou přiřazeny do proměnných „**j**“ a „**lTx**“. Proměnná „**j**“ obsahuje náhodně generované číslo s **binomického** rozdělení pravděpodobnosti s parametry konstanty „**sD**“. Proměnná „**lTx**“ představuje původní vstupní parametr „**lT**“ rozšířený o hodnotu proměnné „**j**“.
- Aktualizuje se hodnota v místě „**C1**“ – proměnná „**min**“ (reprezentující seznam časů) vstupující do přechodu je rozšířena o hodnotu aktuálního času simulace (funkce „**intT**“) vč. proměnné „**j**“. Do místa „**C1**“ je vrácen setříděný seznam pomocí funkce „**iSort**“ (**intersection sort**) [16][17]. Třídění je realizováno z důvodu nalezení minimální hodnoty v seznamu (první hodnota je vždy nejmenší).

Do místa „**Runnable**“ je tedy přeneseno vlákno, které bylo modifikováno:

- **Priorita** se nastaví na hodnotu časové značky v místě „**Runnable**“, tj. simulační čas, při němž bude vlákno aktivní a **připravené** na **přidělení procesoru/jádra**. Stejnou hodnotu obsahuje i seznam (token) v místě „C1“.
- Aktualizovaný čas **dokončení** se inkrementuje o dobu, kterou vláknu trvalo spuštění (rozdíl mezi vytvořením a spuštěním vlákna).
- **Časová značka** se **rozšíří** o hodnotu proměnné „**j**“.

Pokud je čas simulace **roven** (nebo **větší**) nejmenší časové značky v místě „**Runnable**“ je aktivován přechod „**schedule**“. Samotný přechod sebou nese následující činnosti:

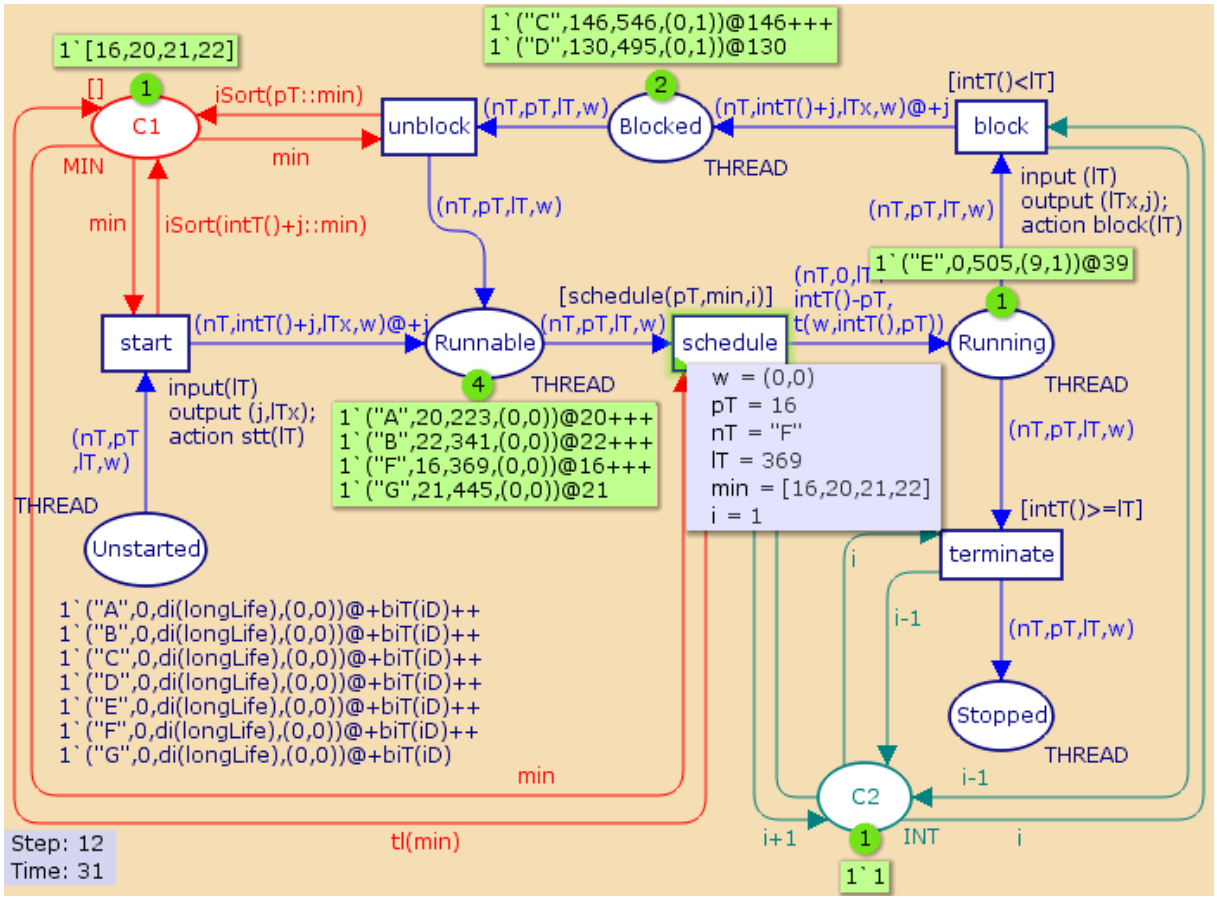
- Aby byl přechod aktivní, musí být splněna strážní podmínka. Strážní podmínka obsahuje funkci „**schedule**“, která vrací logickou hodnotu **true** nebo **false**. Funkce „**schedule**“ přejímá jako vstupní parametry proměnné „**pT“**, „**min“** a „**i“** a v těle porovnává:
    - zda je aktuální počet zpracovávaných vláken (v místě „**Running**“) **menší** než konstanta **COUNT_P** (zaručuje, že nebude přidělen **procesor** (**jádro**), který **neexistuje**),
    - zda je **první** člen v seznamu (**min**) roven hodnotě proměnné „**pT**“ (zaručuje, že bude vždy vybráno vlákno, jehož časová značka v místě „**Runnable**“ je nejmenší – **tedy celý princip algoritmu FCFS**).
- Přechod přiřazuje vláknu zpoždění definované funkcí „**term**“, která používá proměnnou „**lT**“ jako vstupní parametr. Funkce „**term**“ zajišťuje, aby náhodně generované číslo (pomocí konstanty „**rD**“ a funkce „**biT**“) nebylo větší než zbývající požadovaná doba práce vlákna (resp. požadované dokončení práce vlákna). Pokud by takto generované číslo zvýšilo časovou značku nad plánovanou dobu práce, je zvoleno číslo, které reprezentuje zbývající čas práce vlákna.
- Při uskutečnění přechodu je **inkrementována** hodnota (o **1**) tokenu v místě „**C2**“ pomocí proměnné „**i**“.
- Při uskutečnění přechodu je první člen seznamu (**min**) odstraněn pomocí funkce **tl** (vrací seznam bez hlavy).

Obrázek 35 ilustruje síť po **12** krocích. Simulační čas je **31**, což znamená, že všechna vlákna v místě „**Runnable**“ jsou aktivní. V místě „**C1**“ jsou jednotlivé časy těchto vláken setříděny podle velikostí. Nejdříve tedy přišlo vlákno s časem **16**, tj. vlákno „**F**“ bude přiděleno procesoru/jádru jako další. Hodnoty jednotlivých proměnných při uskutečnění přechodu tohoto vlákna jsou znázorněny v okénku pod přechodem „**schedule**“.

Do místa „**Running**“ tedy dorazí vlákno s následujícími změnami:

- Priorita vlákna je vynulována (není nutné ji dále sledovat).
- Celková doba požadované práce vlákna je inkrementována o hodnotu **rozdílu** mezi aktuálním **časem** simulace při uskutečnění přechodu a časem kdy bylo vlákno aktivní v místě „**Runnable**“. Je tedy přičtena hodnota, kterou vlákno čekalo ve stavu „**Runnable**“.
- Časová značka vlákna se zvýšila o hodnotu, kterou vrátila funkce **term**.
- Parametr **WAIT** je nastaven pomocí funkce „**t“** (k prvnímu atributu parametru **WAIT** je přičten **čas čekání** a druhý parametr je inkrementován o hodnotu **1**). Obrázek 35 znázorňuje např. vlákno „**E**“, které v místě „**Running**“ obsahuje hodnotu tohoto parametru **(9,1)**, z čehož vyplývá, že dané vlákno zatím čekalo **9** jednotek simulačního času a bylo přiděleno procesoru pouze **jednou**.

Pokud simulační čas umožňuje aktivitu vlákna v místě „**Running**“, může toto vlákno pokračovat jednou z možných cest, a to přes přechod „**terminate**“ nebo přes přechod „**block**“. Exkluzivita výběru je zaručena pomocí strážních podmínek těchto dvou přechodů, jelikož jsou navzájem antagonistické.



Obrázek 35 - CPN - Scheduling - FCFS – Příklad. Zdroj vlastní

Pokud je **splněna** podmínka pro přechod přes přechod „**terminate**“ je vlákno ukončeno. Strážní podmínka tohoto přechodu je **splněna**, pouze pokud je čas simulace **roven** (nebo větší) **požadovanému času dokončení práce vlákna**. Jestliže se vlákno ukončí (přejde přes přechod „**terminate**“) je automaticky **dekrementována** hodnota (o **1**) tokenu v místě „**C2**“ pomocí proměnné „**i**“.

Pokud není strážní podmínka na přechodu „**terminate**“ splněna, je **automatický plněna** strážní podmínka na přechodu „**block**“. Při přechodu vlákna přes přechod „**block**“ jsou prováděny následující akce:

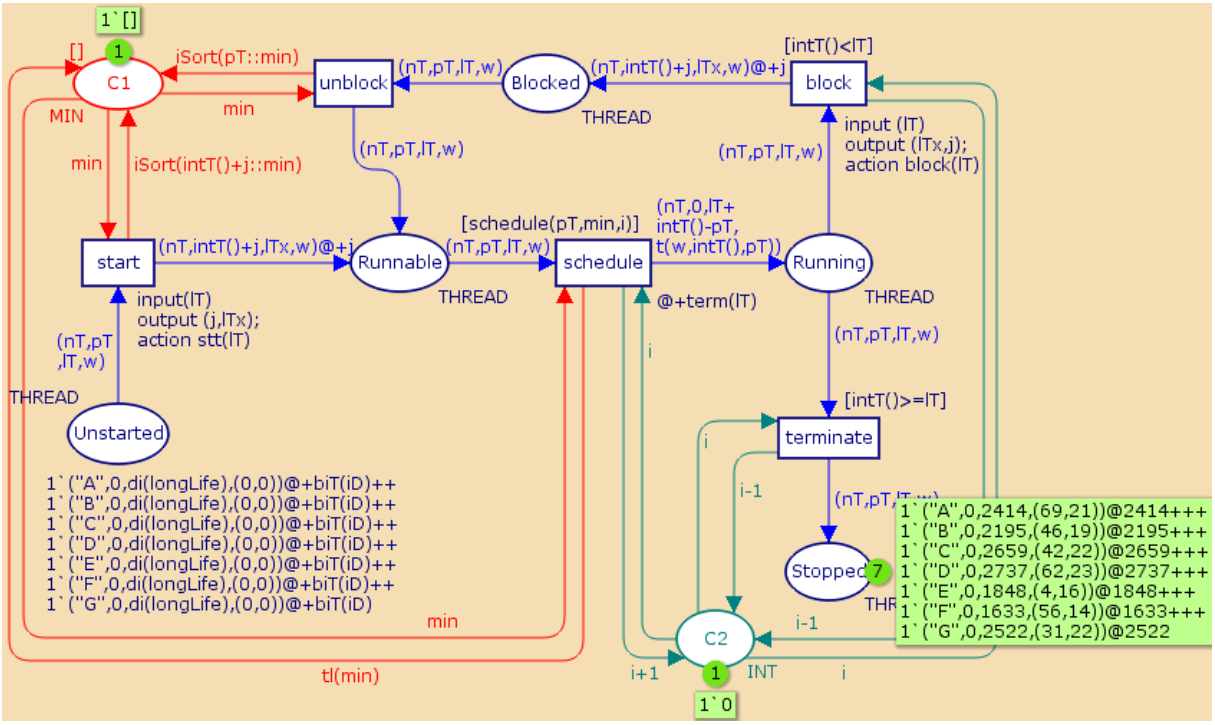
- Vykoná se funkce „**block“**, která je **funkčním ekvivalentem** funkce „**strr“** z přechodu „**start“**. Jediným rozdílem je skutečnost, že je v těle funkce pracováno s konstantou „**bD“** (namísto „**sD“**).
- Pomocí proměnné „**i**“ je **dekrementována** hodnota (o **1**) tokenu v místě „**C2**“.

Do místa „**Blocked**“ tedy přichází vlákno s následujícími modifikovanými parametry:

- **Priorita** vlákna je nastavena na čas **odblokování** (čas odblokování je shodný s časem kdy bude vlákno aktivní ve stavu „**Runnable**“, jelikož přechod „**unblock**“ nepřidává žádné časové zpoždění).
- Celkový čas práce vlákna je zvýšen o dobu blokování (**lTx** = **lT**+**j**).

Pokud je simulační čas vhodný pro aktivaci vlákna v místě „**Blocked**“, je vlákno přeneseno do místa „**Runnable**“ **bez** jakékoliv **modifikace**. Přechod „**unblock**“ navíc přidá hodnotu priority do seznamu v místě „**C1**“ (včetně **ex-post setřídění**).

Obrázek 36 znázorňuje cílový stav modelu algoritmu **FCFS**. Všechna vlákna se nacházejí v místě „**Stopped**“, místo „**C1**“ obsahuje prázdný seznam a místo „**C2**“ neevokuje přidělení procesoru („**i**“ = 0). Z cílových hodnot parametrů vláken je možné hodnotit efektivitu algoritmu z hlediska doby čekání (viz subkapitola 5.5). Z obrázku je např. patrné, že vlákno „**E**“ čekalo nejkratší dobu (4 jednotky simulačního času) a bylo přiděleno procesoru/jádru 16 krát.



Obrázek 36 - CPN - Scheduling - FCFS - Cílový stav. Zdroj vlastní

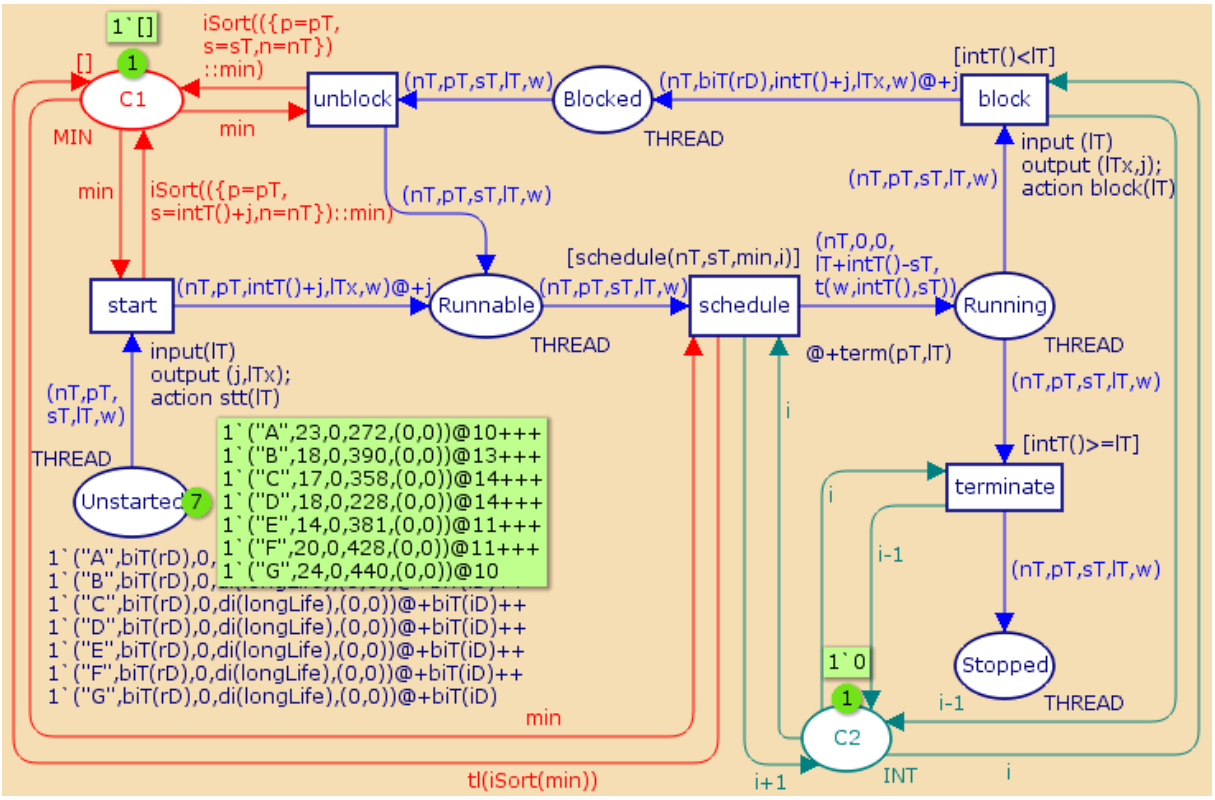
5.2 Shortest Job First (SJF)

Dalším z řady standardních nepreemptivních plánovacích algoritmů je **Shortest Job First** (dále **SJF**), který stejně jako **FCFS** využívá klasickou **FIFO** frontu. Na rozdíl od **FCFS** jsou zde však vlákna řazena podle **délky očekávané doby** přidělení procesoru/jádra. Jsou tedy vždy upřednostněna ta vlákna, která budou využívat procesor nejkratší dobu (před tím než dojde k blokování – např. čekání na vstupně výstupní operaci).

Základní **model** algoritmu **SJF** (viz Obrázek 37) je **obdobou** modelu **FCFS**, a proto budou **popisovány** pouze **odlišnosti** těchto dvou modelů. Celý model lze nalézt v příloze č. 6.

5.2.1 Deklarace struktur modelu algoritmu SJF

Obrázek 37 naznačuje, že struktura parametrů vláken se nepatrně odlišuje od původní struktury v modelu **FCFS**. Změna spočívá v přidání jednoho parametru, který přejímá funkci **priority** z modelu **FCSF** (prioritou v modelu **FCSF** je **čas příchodu**). Původní parametr **PRIORITY** bude nyní nést hodnotu předpokládané doby přidělení procesoru/jádra. **Deklarace** CS, které byly **modifikovány**, či **přidány** znázorňuje Obrázek 38 (zbylé deklarace CS jsou totožné, viz Obrázek 31 v kapitole 5.1.1).



Obrázek 37 - CPN - Scheduling - SJF - Výchozí stav. Zdroj vlastní

Nově tedy každé vlákno obsahuje 5 parametrů, které představují:

- jméno vlákna (**NAMEOFTHREAD**),
- prioritu vlákna, tedy předpokládaný čas, na který bude vláknu přidělen procesor/ jádro (**PRIORITY**),
- čas příchodu do místa „**Runnable**“ - odpovídající parametru PRIORITY v modelu algoritmu FCFS (**STAMP**),
- doba **celkové práce**, kterou **musí** vlákno „**odpracovat**“, než může přejít do míst „**Stopped**“ (**LIFE**),
- celková doba čekání vlákna a počet přidělení procesoru/jádra (**WAIT**).

Zároveň byla upravena deklarace seznamu „**MIN**“, který představuje frontu vláken setříděných podle priority. Nyní je každý prvek seznamu deklarovaný jako záznam (**record**), který se skládá z parametrů „**p**“, „**s**“ a „**n**“, které zaznamenávají důležité parametry vlákna (prioritu, čas příchodu do místa „**Runnable**“ a jméno vlákna).

```
colset STAMP = int;
colset THREAD = product NAMEOFTHREAD *  PRIORITY * STAMP* LIFE * WAIT timed;
colset S = record p:PRIORITY * s:STAMP * n:NAMEOFTHREAD;
colset MIN =  list S;
```

Obrázek 38 - CPN - Scheduling - SJF - Deklarace CS. Zdroj vlastní

Všechny parametry (**konstanty**) modelu zůstávají shodné s **FCFS** deklarací (viz Obrázek 32 v kapitole 5.1.1). S ohledem na nový parametr jednotlivých vláken, je nutné deklarovat odpovídající proměnnou „**sT**“ (viz Obrázek 39), která ponese hodnotu tohoto parametru (**STAMP**).

```
var sT: STAMP;
```

Obrázek 39 - CPN - Scheduling - SJF - Deklarace proměnných. Zdroj vlastní

Nejvíce změn bylo provedeno při **deklarování** jednotlivých **funkcí**, které nyní musí **splňovat odlišný** přístup **SJF** algoritmu. Obrázek 40 ilustruje všechny funkce, které rozšiřují či modifikují deklaraci funkcí z modelu algoritmu **FCFS** (viz Obrázek 33 v kapitole 5.1.1).

```
fun extr ({p,s,n}) = p;
fun extrS ({p,s,n}) = s;
fun extrN({p,s,n}) = n;
fun sor (x,y) =
  let
    val a = intT()
    val b = extrS x
    val c = extrS y
    val d = extr x
    val e = extr y
  in
    if b>a then
      false
    else if b<=a andalso c>a then
      true
    else if d = e then
      b<c
    else
      d<e
  end;
fun ins (n, []):MIN = [n]
    | ins (n, ns as h::t):MIN =
  if sor(n,h) then
    n::ns
  else
    h::(ins (n, t));
val iSort= List.foldr ins [];
fun schedule(u,x,y,z) =
  let
     val d = extrN(hd(iSort(y)))
  in
    d=u andalso z<COUNT_P
  end;
fun term(x,y) = let
    val a = intT()
    val b = y-a
  in
    if ((a+x)>=y) then
      b
    else
      x
  end;
```

Obrázek 40 - CPN - Scheduling - SJF - Deklarace funkcí. Zdroj vlastní

5.2.2 Popis činnosti vláken v modelu algoritmu SJF

Činnosti vláken v modelu algoritmu SJF jsou téměř ekvivalentní s modelem algoritmu FCFS, proto zde budou uvedeny pouze odlišnosti, které jsou významné pro tento algoritmus.

Jak již bylo výše zmíněno třetí parametr vlákna (**STAMP**) reprezentuje časovou značku vlákna v místě „**Runnable**“, tedy simulační čas, při kterém bude vlákno připraveno pro přidělení procesoru/jádra. Tento parametr slouží **pouze** pro potřeby funkce **schedule**, jejíž význam bude probrán níže. Tento parametr, přenášený proměnnou „**sT**“ se chová **stejným** způsobem (nese stejnou informaci) jako parametr **PRIORITY** v modelu algoritmu **FCFS**.

Druhý parametr vlákna (PRIORITY) obsahuje ve výchozím stavu (při inicializaci místa „**Unstarted**“) hodnotu generovanou pomocí funkce „**biT**“ dle parametrů konstanty „**rD“** z **binomického** rozdělení

pravděpodobnosti. Tento parametr tedy nese hodnotu předpokládané doby práce vlákna (na jak dlouho bude mít přidělen procesor/jádro před vstupně výstupní operací). Jelikož se jedná o prioritní parametr, je při každém přechodu přes přechody „**start**“ a „**unblock**“ uchováván (spolu s dalšími parametry) jako záznam v utříděném seznamu v místě „**C1**“. Pokaždé když je vlákno zablokováno je hodnota tohoto parametru nově generována (viz hrana mezi přechodem „**block**“ a místem „**Blocked**“).

Samotná realizace algoritmu **SJF** je uskutečněna na přechodu „**schedule**“. Strážní podmínka tohoto přechodu obsahuje stejnojmennou funkci „**schedule**“, která zaručuje, že bude vybráno vlákno, které odpovídá prvnímu záznamu setříděného seznamu v místě „**C1**“ přenášený proměnnou „**min**“. Zároveň je samozřejmě kontrolován fakt, zda je dostupný volný procesor/jádro.

Jelikož funkce vždy kontroluje pouze první člen v seznamu („**C1**“), musí být definována speciální funkce (sada funkcí), která seznam utřídí do formy odpovídající danému plánovacímu algoritmu. U algoritmu **FCFS** stačilo seznam utřídit podle velikosti (vzestupně) prioritního parametru. U algoritmu **SJF** toto není možné, jelikož by se mohlo stát, že vlákno s nejkratší předpokládanou dobou práce (přidělení procesoru/jádra) bude mít časovou značku v místě „**Runnable**“ větší než aktuální čas simulace. Toto vlákno však ještě není aktivní, a tudíž by nemohl být přechod uskutečněn a nastal by **deadlock**. Proto je nutné třídit pouze záznamy seznamu, jejichž časová značka je menší nebo rovna aktuálnímu čas a ostatní záznamy ponechat na „zadních“ místech. Tuto řadící podmínku realizuje funkce „**sor**“, která pracuje následujícím způsobem:

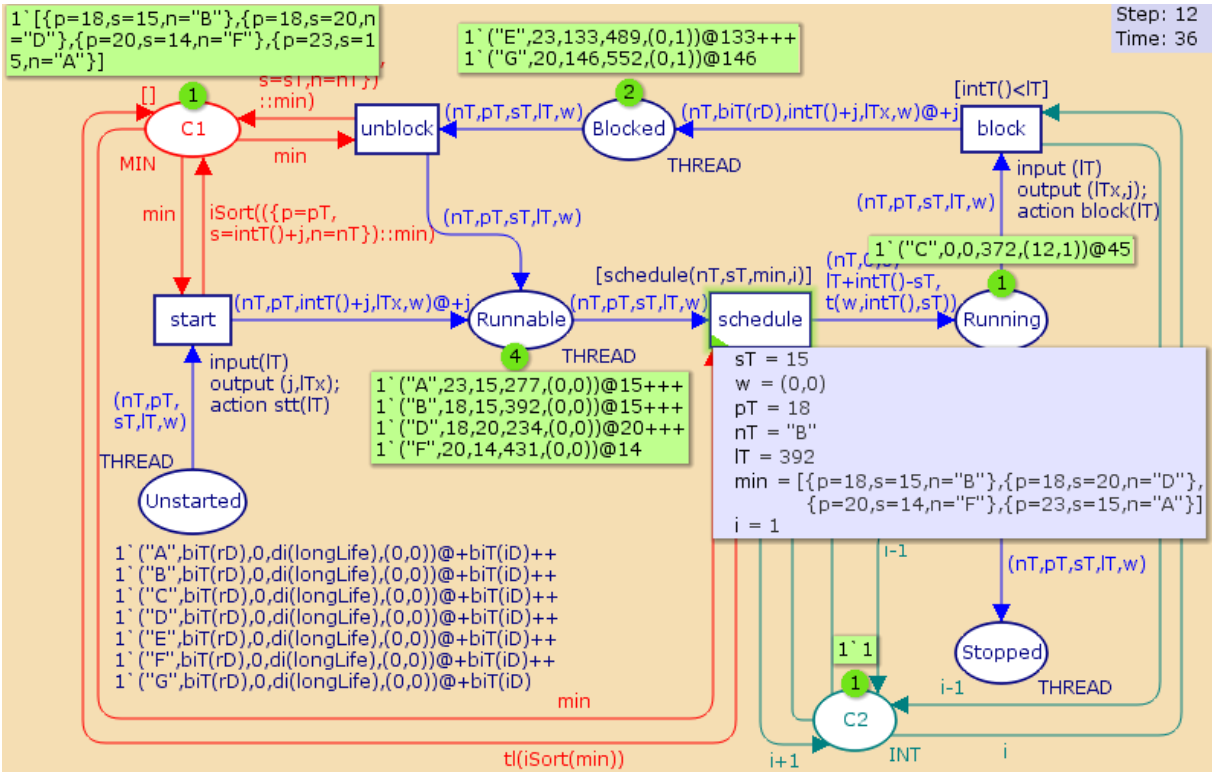
- Pokud je časová značka aktuálně porovnávaného záznamu větší než aktuální čas simulace, je automaticky vyhodnocen jako větší než jakýkoli jiný záznam.
- Pokud je časová značka aktuálně porovnávaného záznamu menší než aktuální čas simulace:
    - Záznam je automaticky menší, pokud časová značka záznamu, vůči němuž je porovnáváno, je větší, než aktuální čas simulace.
    - Pokud je časová značka druhého záznamu také menší nebo rovna aktuálnímu času simulace, jsou porovnávány jejich hodnoty (v případě, že jsou priority totožné, je rozhodováno podle druhého parametru – STAMP).

Rozhodovací tabulka (viz Tabulka 6) přehledně shrnuje řadící filozofii. Výrazy „**sn**“ a „**sh**“ reprezentují druhé parametry (STAMP) aktuálního a předchozího záznamu.

Tabulka 6 - Rozhodovací tabulka funkce třídící podmínky „sor“. Zdroj vlastní

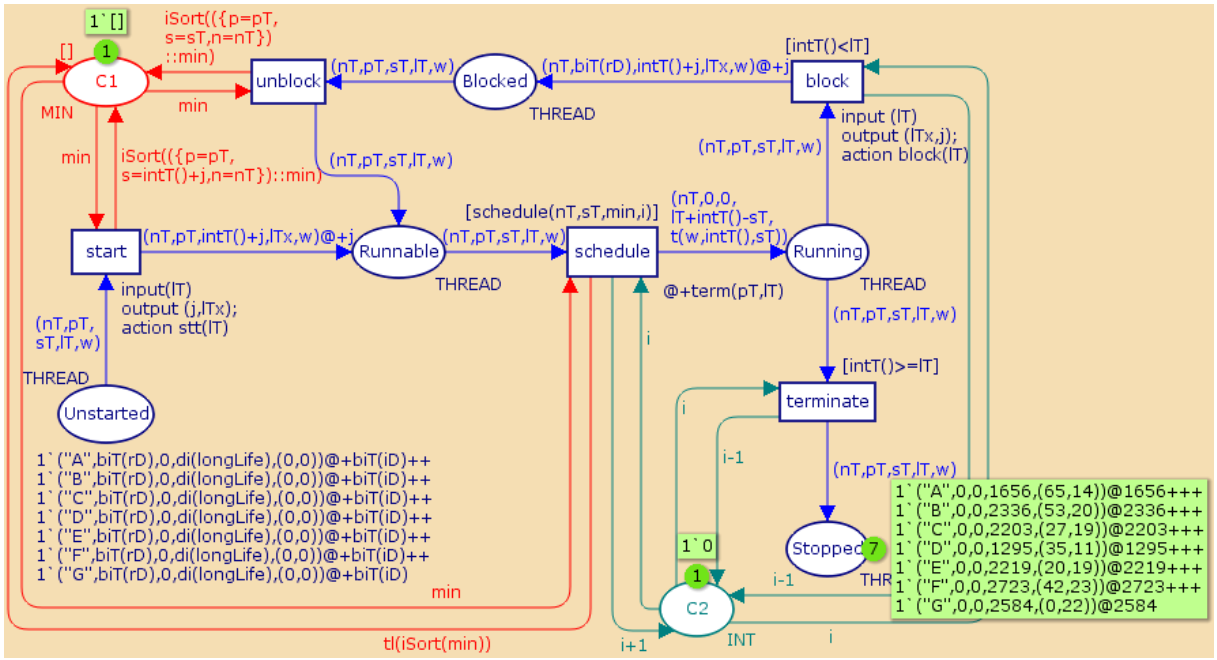
Aktuální záznam (n)	Operátor	Předchozí záznam (h)	Návratová hodnota funkce
n > aktuální čas	and	h > aktuální čas	false
n > aktuální čas	and	h <= aktuální čas	false
n <= aktuální čas	and	h > aktuální	true
n <= aktuální čas	and	h <= aktuální čas	If n=h then sn<sh else n<h

Obrázek 41 ilustruje model algoritmu **SJF** v simulačním čase 36 (po provedení 12 kroků). V místě „**Runnable**“ jsou připravena **4 vlákna** („A“, „B“, „D“ a „F“). První záznam v seznamu v místě „**C1**“ obsahuje odkaz na vlákno „**B**“, které má totožnou předpokládanou dobu práce jako vlákno „**D**“, avšak přišlo do místa „**Runnable**“ jako **první** (v simulačním čase 15).



Obrázek 41 - CPN - Scheduling - SJF - Příklad. Zdroj vlastní

Cílový stav modelu algoritmu SJF je ekvivalentní s modelem algoritmu FCFS (viz Obrázek 42).



Obrázek 42 - CPN - Scheduling - SJF - Cílový stav. Zdroj vlastní

5.3 Round Robin (RR)

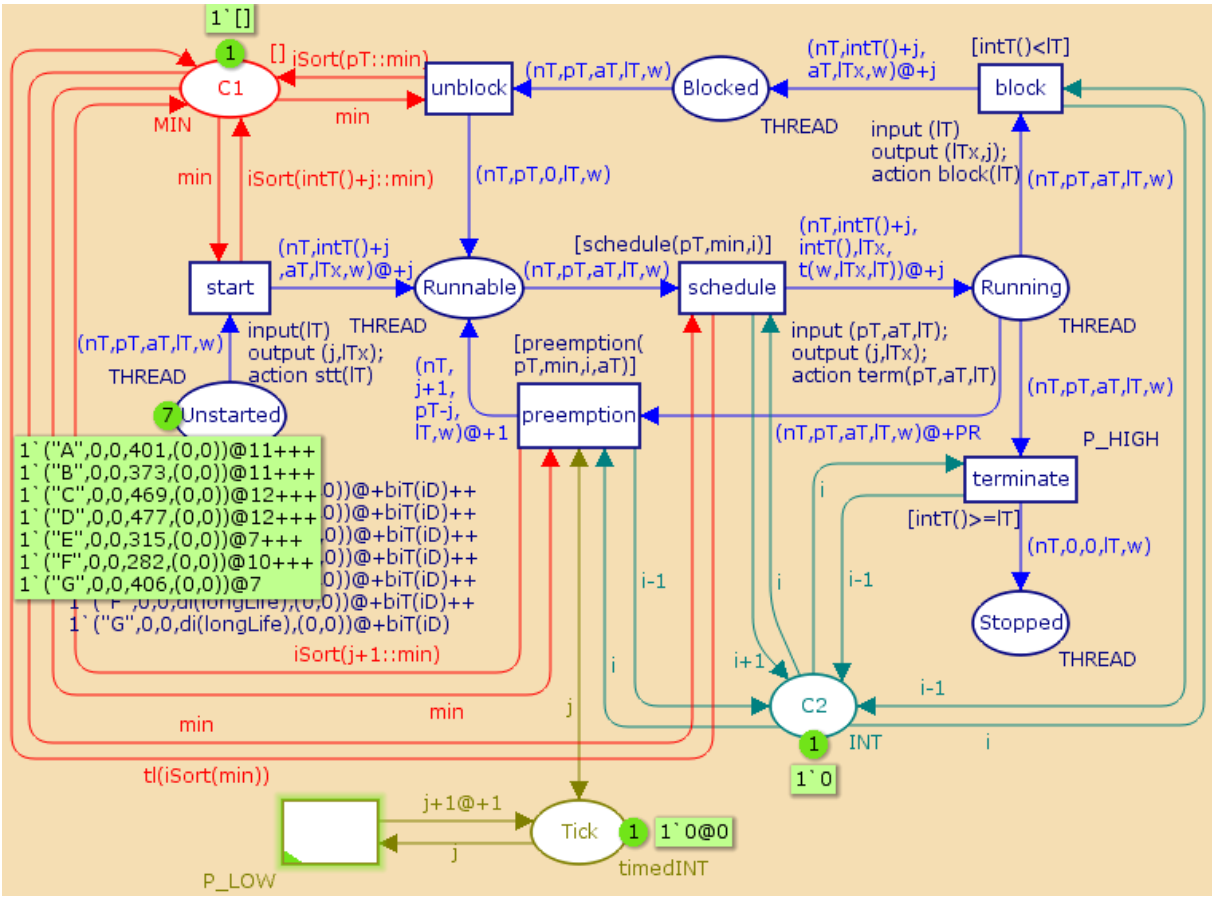
Nejjednodušším a zároveň nejspravedlivějším **preemptivním** plánovacím algoritmem je **Round Robin** (dále **RR**). Jedná se o algoritmus, který každému vláknu přidělí konstantní **quantum**, po jehož uplynutí je vláknu procesor/jádro odebrán. Jednotlivá vlákna jsou v místě „**Runnable**“ řazena podle času příchodu. Jedná se tedy o rozšíření algoritmu **FCFS** o preempci. Celý model lze nalézt v příloze č. 7.

5.3.1 Deklarace struktur modelu algoritmu RR

Model algoritmu **RR** je obdobou modelu algoritmu **FCFS** (viz Obrázek 43). Pouze ho rozšiřuje o přechod „**preemption**“, který vrací vlákno z místa „**Running**“ zpět do místa „**Runnable**“. Dále je vláknům přidělen jeden nový parametr **ACT (integer)**, který může nést dva typy informací:

- Na hranách mezi místem „**Runnable**“ a přechody „**schedule**“ a „**preemption**“ reprezentuje dobu, kterou vláknu zbývá dokončit (před vstupně výstupní operací) po preempci. Pokud vlákno nepřišlo do místa „**Runnable**“ přes přechod „**preemption“** je hodnota této proměnné nulová.
- Na hranách mezi místem „**Running**“ a přechody „**schedule**“ a „**preemption**“ reprezentuje čas, kdy byl vláknu skutečně přidělen procesor/jádro.

Deklaraci tohoto parametru, vč. modifikovaná deklarace vlákna znázorňuje Obrázek 44.



Obrázek 43 - CPN - Scheduling - RR - Výchozí stav. Zdroj vlastní

Ostatní parametry vlákna si ponechávají stejnou deklaraci jako model algoritmu FCFS (viz Obrázek 31 v kapitole 5.1.1).

```
colset ACT = int;
colset THREAD = product NAMEOFTHREAD * PRIORITY  * ACT * LIFE * WAIT timed;
```

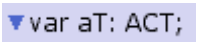
Obrázek 44 - CPN - Scheduling - RR - Deklarace CS. Zdroj vlastní

Pro potřeby preempce byly nově vytvořeny 2 simulační parametry (konstanty), které představují délku **quanta** (**QUANTUM**) a pomocný parametr (**PR**), viz Obrázek 45.

```
val QUANTUM = 10;
val PR =  999999;
```

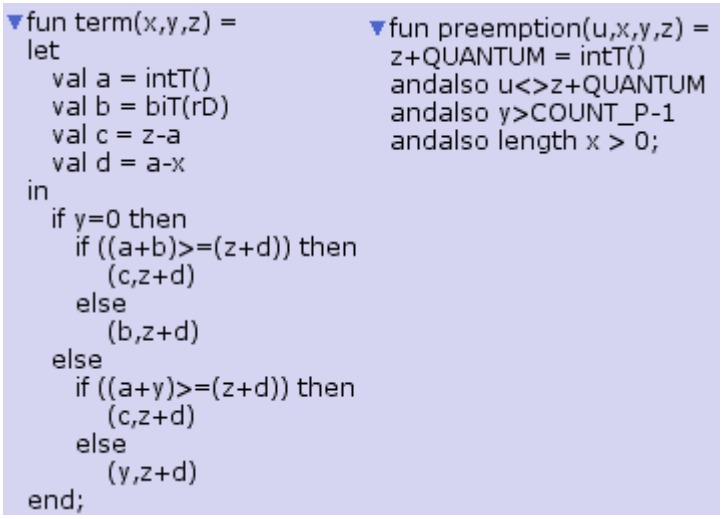
Obrázek 45 - CNP - Scheduling - RR - Deklarace konstant. Zdroj vlastní

Jelikož byl přidán nový parametr vlákna (**ACT**), musí být deklarována proměnná, která ho bude přenášet mezi místy. Tato proměnná se v tomto modelu nazývá „**aT**“ a její deklaraci ilustruje Obrázek 46



Obrázek 46 - CPN - Scheduling - RR - Deklarace proměnných. Zdroj vlastní

Pro nově vytvořený přechod „**preemption**“ byla vytvořena stejnojmenná strážní podmínka, která zajišťuje samotnou preempci (s pomocí místa „**Tick**“). Zároveň byla modifikována funkce „**term**“, která řídí přidělování času procesoru jednotlivým vláknům (viz Obrázek 47).



Obrázek 47 - CPN - Scheduling - RR - Deklarace funkcí. Zdroj vlastní

5.3.2 Popis činnosti modelu algoritmu RR

Činnost odlišná od algoritmu **FCFS**, začíná v kódovém segmentu přechodu „**schedule**“. Zde je využita funkce „**term**“, která přejímá parametry proměnné „**pT**“ (čas, kdy bylo vlákno připraveno pro přidělení procesoru/jádra), „**aT**“ (doba, kterou vláknu ještě zbývá odpracovat z aktuálního přidělení procesoru) a „**lT**“ (celková předpokládaná doba práce vlákna). Pokud je hodnota proměnné „**aT**“ **nulová**, tj. vlákno přišlo do místa „**Runnable**“ **jinou** cestou, než přes přechod „**preemption**“, tak funkce pracuje stejným způsobem jako u algoritmu **FCFS** (čas přidělení procesoru/jádru, je generován a poté přidělen vláknu). Pokud je však hodnota proměnné „**aT**“ nenulová, znamená to, že vláknu byl „násilně“ odňat procesor/jádro, a tudíž byl čas přidělení zkrácen o **quantum** (pokud samozřejmě není předpokládaná doba delší než čas celkové práce vlákna). V takovém případě tedy funkce „**term**“ nastaví čas práce vlákna na hodnotu proměnné „**aT**“.

Jelikož je hodnota konstanty „**PR“** nastavena na dostatečně velkou hodnotu bude značení **„@+PR**“ na výstupní hraně představovat preempci času na nulu. Jinak řečeno, všechny vlákna v místě „**Running**“ budou moci přejít přes tuto hranu i pokud jejich časová značka bude větší než aktuální čas simulace. Vlákno je však přes tuto hranu přeneseno pouze pokud je splněna **strážní** podmínka na přechodu „**preemption**“.

Než bude vysvětlena samotná strážní podmínka, je nutné osvětlit princip pomocného místa „**Tick**“. Toto místo obsahuje časovaný token CS **timedINT**, který je při každé úrovni času **zvýšen** o **1**. Nepojmenovaný přechod nalevo od místa „**Tick“** je nastavena **nízká priorita** z důvodu toho, aby byly vždy **upřednostněny** ostatní přechody sítě a až nakonec bude realizována inkrementace času v místě „**Tick**“ o **1**. Pokud by nebyl čas zvyšován takto „manuálně“ nedocházelo by k preempci jelikož, pokud v daný čas není již aktivní žádný přechod, je simulační čas modelu inkrementován na nejnižší hodnotu časové značky tokenu, který v tento čas může aktivovat nějaký přechod.

Samotná strážní podmínka na přechodu „**preemption**“ nese stejnojmennou funkci, která umožní preempci, pokud jsou splněny všechny následující podmínky zároveň:

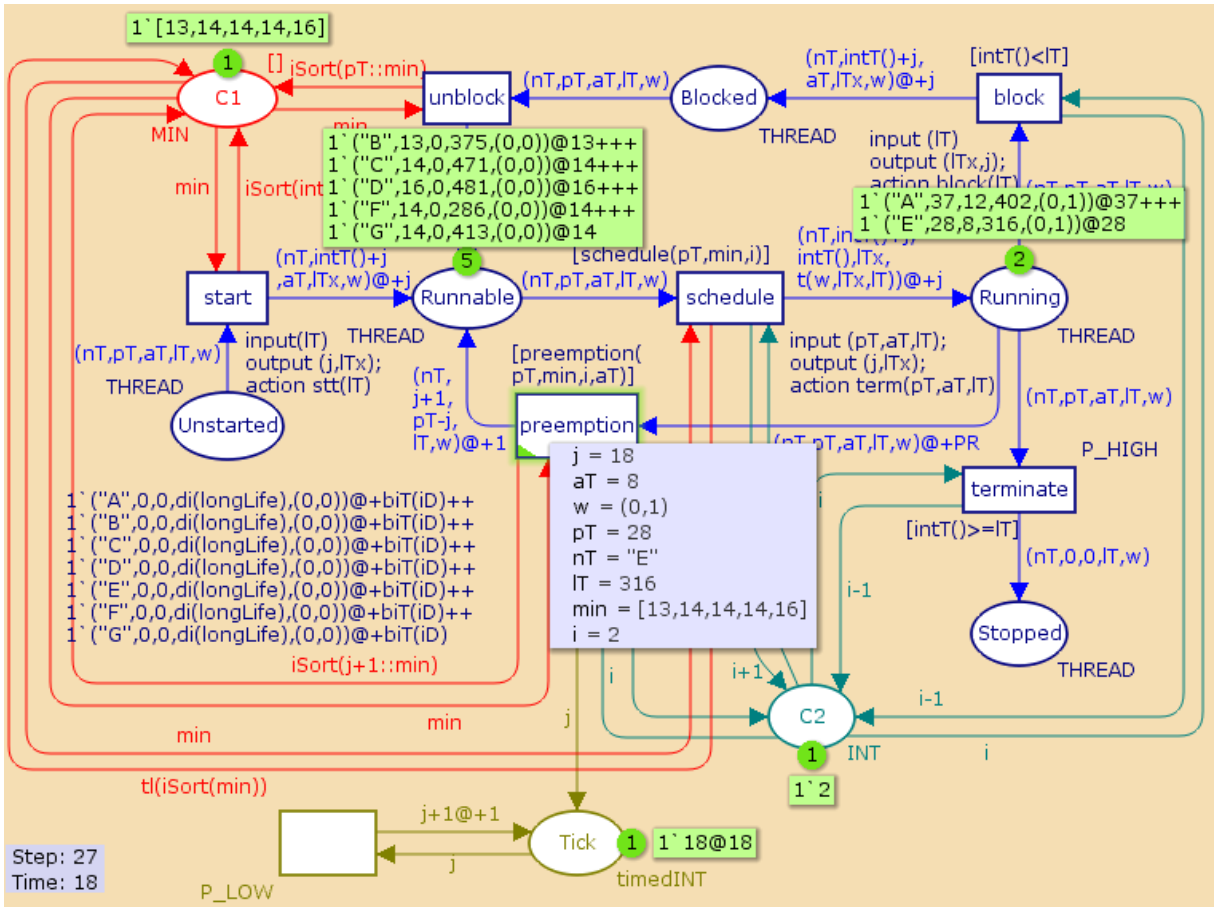
- aktuální simulační čas je roven začátku práce vlákna posunutý o hodnotu konstanty **QUANTUM**,
- čas dokončení vlákna není shodný s dobou zahájení posunutou o hodnotu konstanty **QUANTUM** (toto zabraňuje preempci vláken, které dokončí svou práci, přesně na konci přiděleného **quanta**),
- všechny procesory/jádra **jsou** obsazeny,
- existuje alespoň jedno vlákno v místě „**Runnable**“ (shodné s počtem prvků v místě „**C1**“).

Preempce se tedy uskuteční u všech vláken, která v daný časový okamžik všechny tyto podmínky splňují.

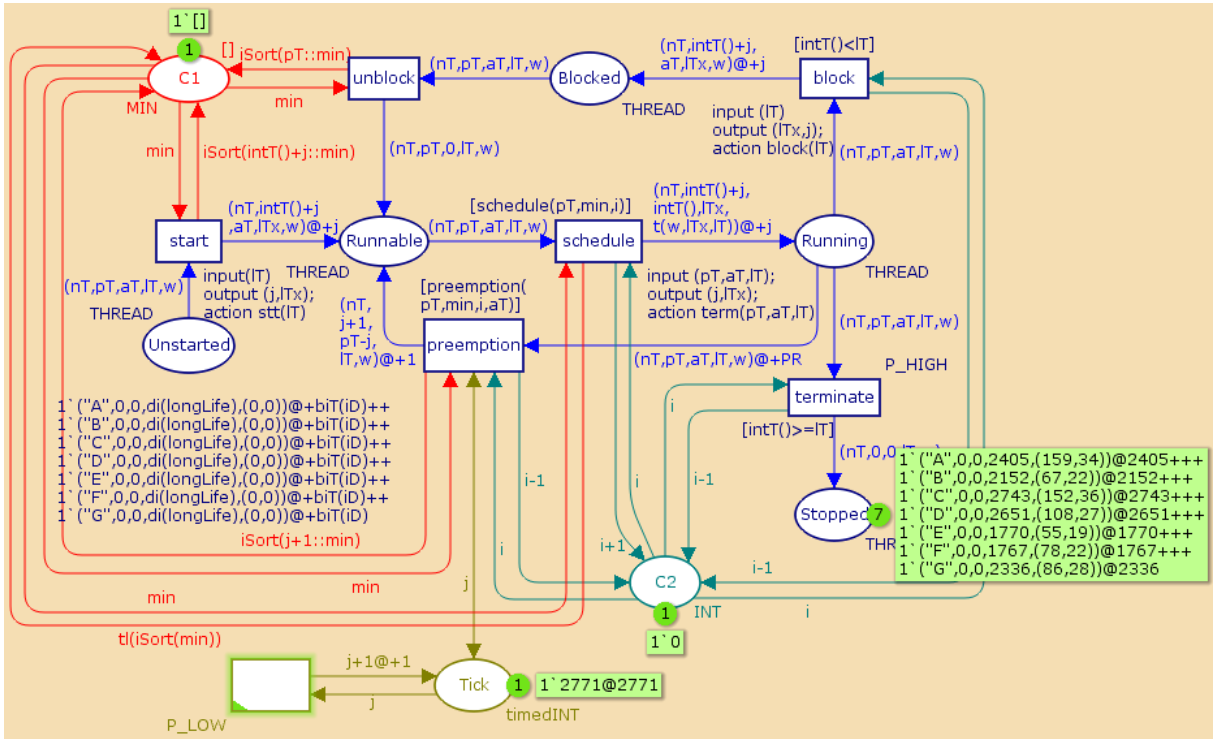
Přechodem přes přechod „**preemption**“ je tedy každému vláknu nastavena časová značka aktuálního času (pomocnou proměnou „**j**“ z místa „**Tick**“) zvýšena o **1** (reprezentující režii přepnutí kontextu). Na stejnou hodnotu je nastaven i parametr vlákna **PRIORITY**. Hodnota parametru **ACT** je nastavena na rozdíl hodnot časů plánovaného ukončení práce vlákna a aktuálního času preempce. Tento parametr, tedy nese hodnotu zbývající práce.

Obrázek 48 ilustruje situaci, kdy vlákno „**E**“ bude odebráno procesoru/jádru. V simulační čas **18** je totiž toto vlákno přesně **10 simulačních jednotek** v místě „**Running**“ (parametr **ACT** = **8**), což odpovídá délce definovaného **quanta**.

Cílový stav modelu algoritmu **RR** znázorňuje Obrázek 49. Jelikož však síť obsahuje nekonečný cyklus (místo „**Tick**“ a pomocný přechod nalevo o něj), nejedná se o mrtvé značení.



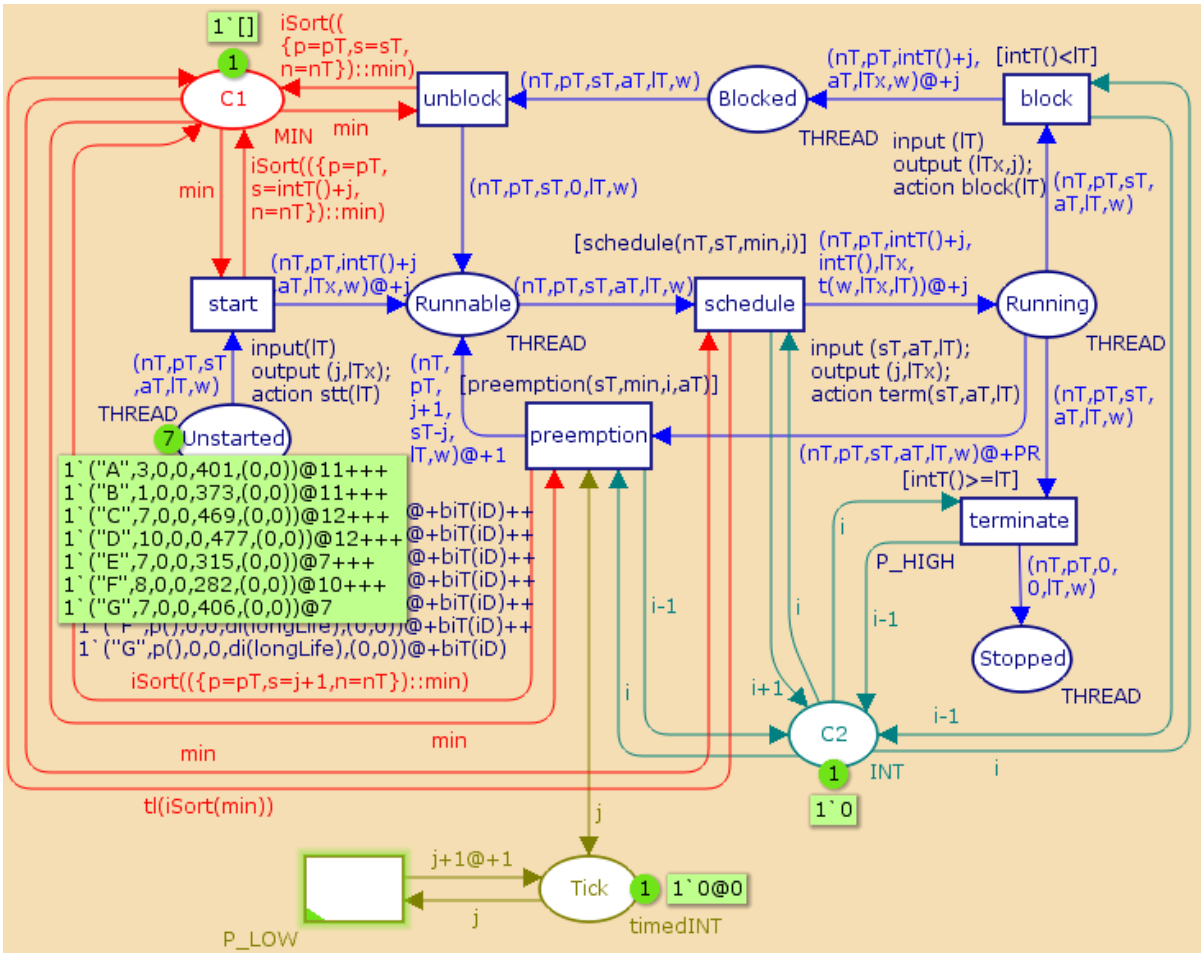
Obrázek 48 - CPN - Scheduling - RR - Příklad preempce. Zdroj vlastní



Obrázek 49 - CPN - Scheduling - RR - Cílový stav. Zdroj vlastní

5.4 Prioritní plánování

Posledním modelovaným algoritmem, je obecný model **prioritního plánovaní**, které využívá **víceúrovňové** fronty typu **RR**. Tento model je v podstatě aplikace **preemptivního** mechanizmu definovaného v předchozí subkapitole na model algoritmu **SJF** s tím, že priorita nebude předpokládaná doba práce, ale obecné **prioritní číslo**. Obrázek 50 ilustruje navržený model algoritmu prioritního plánování s pomocí víceúrovňových front (model, viz příloha č. 8).



Obrázek 50 - CPN - Scheduling - Prioritní plánování. Zdroj vlastní

Jak již bylo zmíněno tento model je aplikací preemptivního mechanizmu na model algoritmu **SJF**. Jediná úprava spočívá v interpretaci parametru „**PRIORITY**“, který nyní reprezentuje celé kladné číslo z rozsahu **1 – 10** (**vyšší** číslo znárodňuje **vyšší** prioritu). Při inicializaci sítě je pomocí funkce „**ran**“ vláknu automaticky přiřazeno číslo z toho intervalu. Úprava deklarace CS **PRIORITY**, jakožto i nová deklarace funkce „**p**“ ilustruje Obrázek 51.

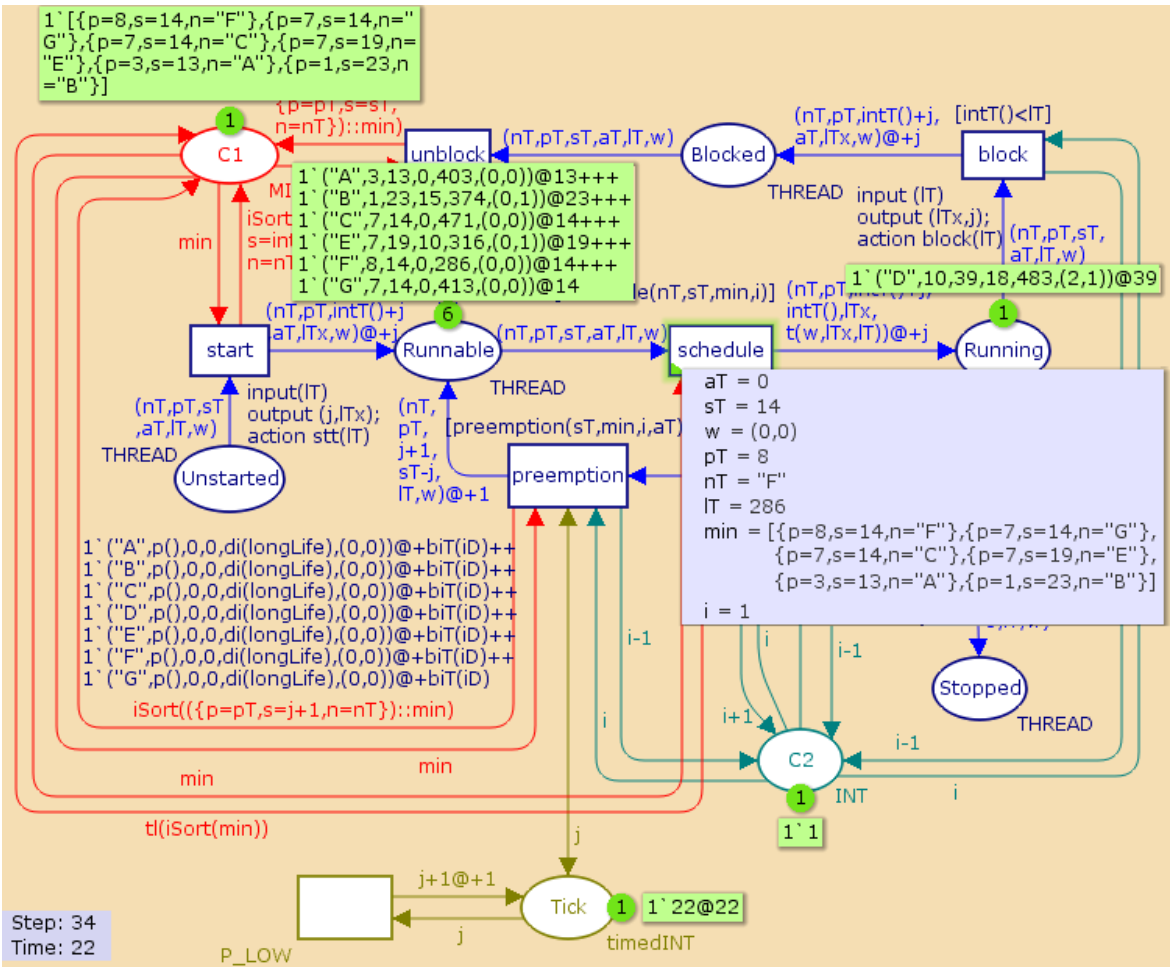
Výsledně jsou tedy vlákna řazena v místě „**Runnable**“, resp. v místě „**C1**“ podle priority. Vlákna se stejnou **prioritou** jsou řazena podle **času příchodu**, resp. časové značky v místě „**Runnable**“. Tímto je

zajištěno, že bude vždy vybráno vlákno, které má největší prioritu a zároveň na úrovni své priority přišlo nejdříve.

```
colset PRIORITY = int with 1..10;
fun p() = PRIORITY.ran();
```

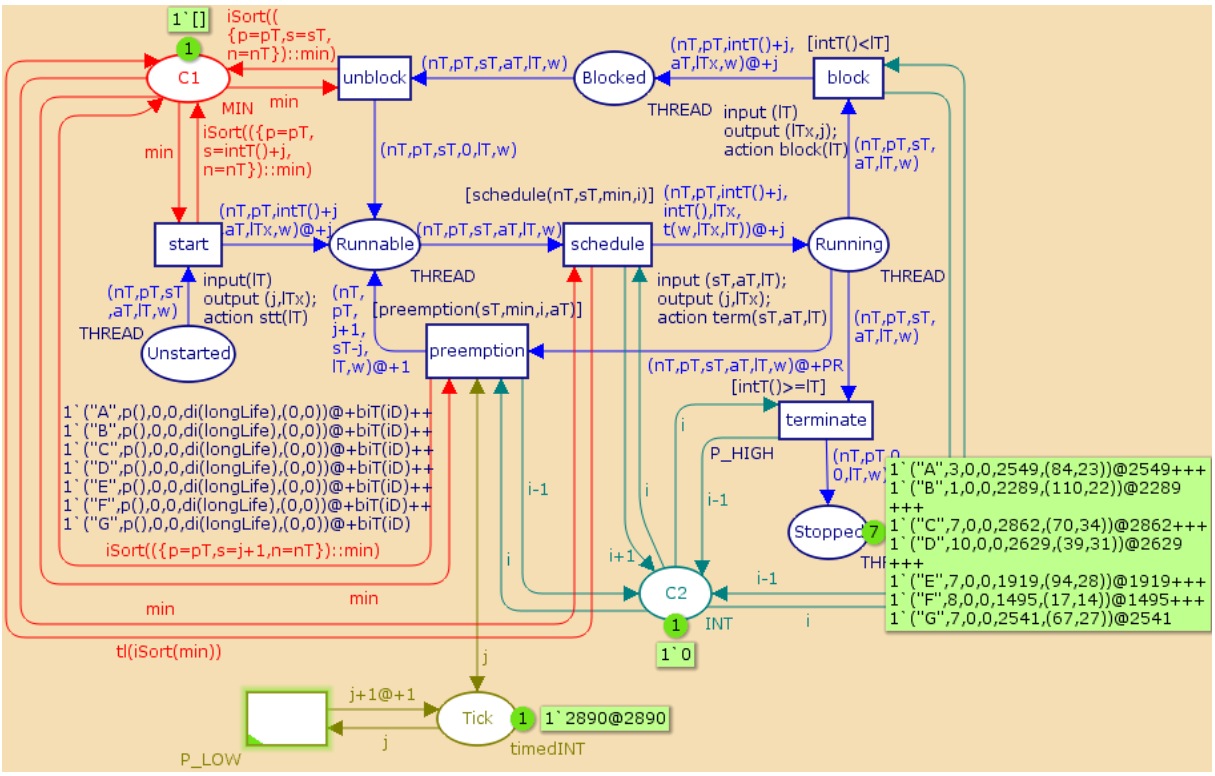
Obrázek 51 - CPN - Scheduling - Prioritní plánování - Deklarace. Zdroj vlastní

Obrázek 52 znázorňuje stav sítě v simulačním čase **22** (po **32** simulačních krocích). V místě „**Runnable**“ je připraveno **5** vláken („**A**“, „**C**“, „**E**“, „**F**“ a „**G**“). Vlákno „**B**“ obsahuje časovou značku s hodnotou **23**, tudíž ještě není aktivní. Z místa „**C1**“ je patrné, že nejvyšší prioritu (první záznam v seznamu) vlastní vlákno „**F**“, tudíž půjde přes přechod „**schedule**“ jako další. Ze seznamu v místě „**C1**“ dále vyplývá, že vlákna „**G**“, „**C**“ a „**E**“ mají stejnou prioritu. Z tohoto důvodu jsou tato vlákna dále řazena podle časové značky v místě „**Runnable**“. Pokud mají vlákna shodnou prioritu i tuto časovou značku, je vybráno vlákno na základě konkrétní realizace řadícího algoritmu (viz vlákna „**G**“ a „**C**“). Jedná se však pouze o **zevšeobecnění**, jelikož model nepracuje s **reálným** časem (reálně nemohou 2 vlákna přijít do stavu „**Runnable**“ ve shodný čas).



Obrázek 52 - CPN - Scheduling - Prioritní plánování - Příklad. Zdroj vlastní

Cílový stav modelu algoritmu prioritního plánování ilustruje Obrázek 53. Stejně jako u modelu algoritmu RR se však nejedná o mrtvé značení, jelikož síť obsahuje zacyklení.



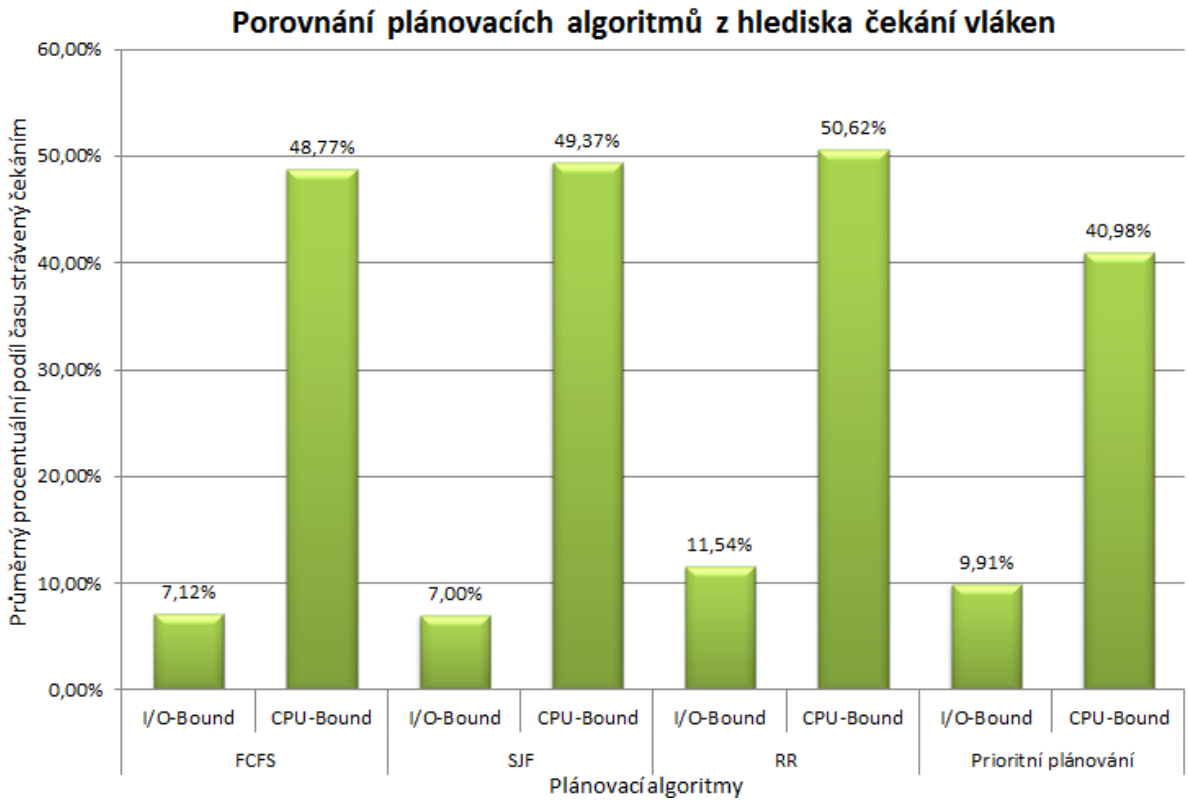
Obrázek 53 - CPN - Scheduling - Prioritní plánování - Cílový stav. Zdroj vlastní

Do modelu by mohlo být dále implementován mechanizmus, který by umožnil dynamickou úpravu priorit např. v závislosti na době čekání vlákna (**aging**) nebo automaticky zvýhodňovat vlákna, která byla právě odblokována. Všechny tyto mechanizmy zabraňují stárnutí vláken (**starvation**), které je hlavním problém při dodržování požadavku na spravedlivost plánování. Modelování těchto mechanizmů je však již nad rámec této práce.

5.5 Porovnání plánovacích algoritmů

Všechny modely navržené v této kapitole obsahují speciální proměnnou „**w**“ (CS **WAIT**), která zajišťuje sběr statistických údajů v průběhu simulace. Těmito statistickými údaji je celková doba čekání vlákna (tj. čas strávený v místě „**Runnable**“) a počet přidělení procesoru/jádra. Na základě provedení pěti testovacích simulací pro každý algoritmus, bylo zjišťováno, jak dlouho z celkové doby života vlákno čeká, tj. procentuální podíl čekání vlákna vůči celkové době jeho života. Testování probíhalo pro dvě odlišná nastavení simulačních parametrů. Prvně byla uskutečněna simulace s původními simulačními parametry (viz Obrázek 32 v kapitole 5.1.1). Poté byly hodnoty simulačních parametrů „**sD**“ a „**bD**“ prohozeny a simulace byla zopakována. Původní simulační parametry reprezentují tzv. **I/O-Bound** vlákno, jelikož doba blokování je delší než doba běhu vlákna. Prohozením

hodnot parametrů „**sD**“ a „**bD**“ vlákno reprezentuje tzv. **CPU-Bound** vlákno. Výsledky provedených simulací shrnuje Obrázek 54.



Obrázek 54 - Porovnání plánovacích algoritmů. Zdroj vlastní

Z obrázku vyplývá, že z hlediska doby čekání, jsou jednotlivé algoritmy relativně podobné. Naopak velký rozdíl je patrný mezi čekáním u vláken typu **I/O-Bound** a **CPU-Bound**. Tyto výsledky jsou však pouze **ilustrační**, jelikož závisí na konkrétním nastavení simulačních parametrů (rozdělení pravděpodobnosti, doba práce, velikost quanta apod.), a proto se mohou dosahované hodnoty výrazně měnit v konkrétních situacích.

V této kapitole byly modelovány základní plánovací algoritmy používané v různých typech OS. Modelování probíhalo na úrovní operačního systému, tj. byla uvažována kernelová vlákna. Stejné algoritmy by samozřejmě mohly být využity i u vláken na uživatelské úrovni (s odlišnou interpretací stavů a vztahů), avšak s podmínkou, že nikdy nepoběží paralelně (COUNT_P = 1).

6 Synchronizace

Synchronizace vláken se zabývá základními koncepty mezivláknové komunikace a strukturami, které umožňují bezpečnou souběžnou činnost více vláken. Popis jednotlivých struktur a postupů je v této kapitole realizován na základě terminologie **programovacího jazyka C#** [18]. Tím je docílena jednotná **interpretovatelnost** a případná **dohledatelnost** jednotlivých synchronizačních struktur.

Úkolem této kapitoly je namodelovat logiku synchronizačních struktur a celkového chování interakce mezi vlákny. Zároveň zde bude **abstrahováno** od plánovací logiky systémového plánovače (**scheduler**).

6.1 Jednoduché blokování

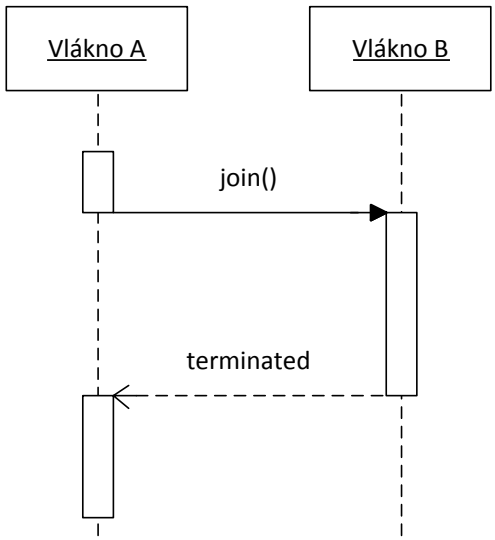
Základními synchronizačními nástroji, jsou různé blokovací konstrukce, které umožňují pozdržet práci jednoho vlákna vůči jinému. Blokování je možné generalizovat na následující tři skutečnosti:

- vlákno čeká (je zablokováno), než jiné vlákno dokončí práci (**join**),
- vlákno je zablokováno na určitý časový úsek (**sleep**),
- vlákno čeká (je zablokováno), než dostane signál od jiného vlákna (**wait**).

Pokud nastane jedna s těchto situací, je stav vlákna změněn na „**WaitSleepJoin“**. Tento stav odpovídá klasickému stavu „**Blocked**“, který byl definován v kapitole 4. V dalším textu budou tedy tyto dva stavy považovány za totožné.

6.1.1 Metoda join

Metoda **join** umožňuje zablokovat vlákno do doby, než jiné vlákno dokončí svou činnost (přejde do stavu „**Stopped**“). Poté je vlákno odblokováno a opět pokračuje ve své činnosti. Obrázek 55 ilustruje případ kdy „**Vlákno A**“ aplikuje metodu **join** vůči „**Vláknu B**“. Tím se činnost „**Vlákna A**“ přeruší (jeho stav se změní na „**WaitSleepJoin“**) dokud není „**Vlákno B**“ ukončeno.



Obrázek 55 - Metoda join. Zdroj vlastní

S využitím znalostí z kapitoly 3, lze tuto metodu modelovat jako souběžnou činnost několika nezávislých vláken. Jednotlivá vlákna jsou pojmenována velkými písmeny anglické abecedy, tj. **„A“ – „Z“**. Dalším parametrem vlákna je index, který podrobněji specifikuje jeho stav. Posledním parametrem je název vlákna, na jehož ukončení zablokované vlákno čeká.

Seznam a popis jednotlivých indexů vláken obsahuje Tabulka 7. Metoda **join** pracuje s prvními třemi indexy, tj. 0 – 2, a posledním indexem (index 5). Ostatní indexy budou využity v navazujících částech této kapitoly. Model metody **join** lze nalézt v příloze č. 9.

Tabulka 7 - Seznam indexů vláken. Zdroj vlastní

Index	Popis
0	vlákno je nově vytvořené
1	vlákno běží
2	vlákno je zablokované, čeká na dokončení jiného vlákna (**join**)
3	vlákno je zablokované, spí (**sleep**)
4	vlákno je zablokované, čeká na signál (**wait**)
5	vlákno je zastaveno

Obrázek 56 znázorňuje deklaraci výše zmíněných parametrů vlákna:

- jméno vlákna (CS **NAMEOFTHREAD**) typu **řetězec**,
- index, resp. typ vlákna (CS **TYPEOFTHREAD**) typu **integer**,
- jméno vlákna, vůči kterému byla aplikována metoda **join** (CS **OTHERTHREAD**) typu **řetězec**.

```
colset NAMEOFTHREAD = string;
colset TYPEOFTHREAD = int;
colset OTHERTHREAD = string;
colset THREAD = product NAMEOFTHREAD *  TYPEOFTHREAD  * OTHERTHREAD;
var nT:NAMEOFTHREAD;
var iT: TYPEOFTHREAD;
var oT:OTHERTHREAD;
var i: INT;
var t1:STRING;
```

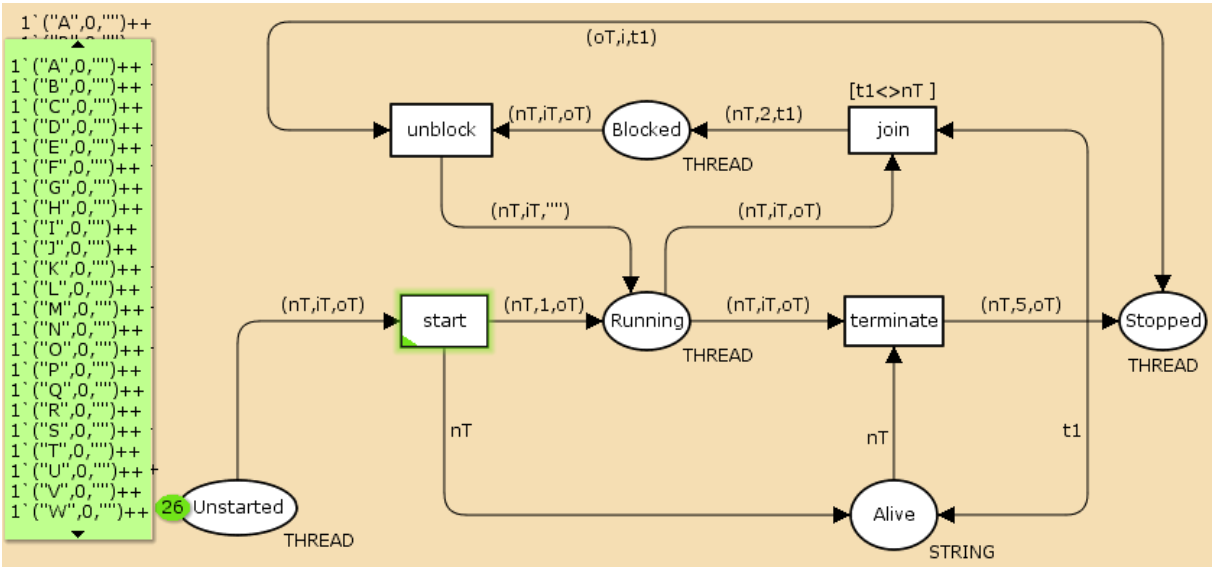
Obrázek 56 - CPN - Metoda join - Deklarace. Zdroj vlastní

Dále obrázek znázorňuje deklaraci samotného vlákna, jakožto produkt těchto tří parametrů (**THREAD**). Posledních 5 proměnných (uvozovaných klíčovým slovem **var**) slouží pro potřeby simulace modelu:

- proměnná **nT** (name of thread) reprezentuje jméno vlákna pohybující se mezi místy,
- proměnná **iT** (index of thread) reprezentuje index vlákna pohybující se mezi místy,
- proměnná **oT** (other thread) reprezentuje název „druhého“ vlákna pohybující se mezi místy,
- proměnné **i** a **t1** nemají žádný specifický význam, slouží pro potřeby modelu jako pomocné proměnné.

Na základě výše deklarovaných konstrukcí byl vytvořen CPN model, viz Obrázek 57. Ve výchozím stavu tento model obsahuje 26 aktivních tokenů v místě „**Unstarted**“ (nespuštěná vlákna).

Z tohoto výchozího stavu se mohou vlákna přemístit přes přechod „**start**“ do místa „**Running**“ a zároveň přidat záznam do místa „**Alive**“. Přechod „**start**“ reprezentuje, jak již název napovídá spuštění daného vlákna, s čímž je spojena změna indexu na hodnotu 1 (vlákno běží), tj. nachází se ve stavu „**Running**“. Místo „**Alive**“ uchovává jmenný seznam všech vláken, které se nenacházejí ve stavu „**Unstarted**“ nebo „**Stopped**“.



Obrázek 57 - CPN - Metoda join - Výchozí stav. Zdroj vlastní

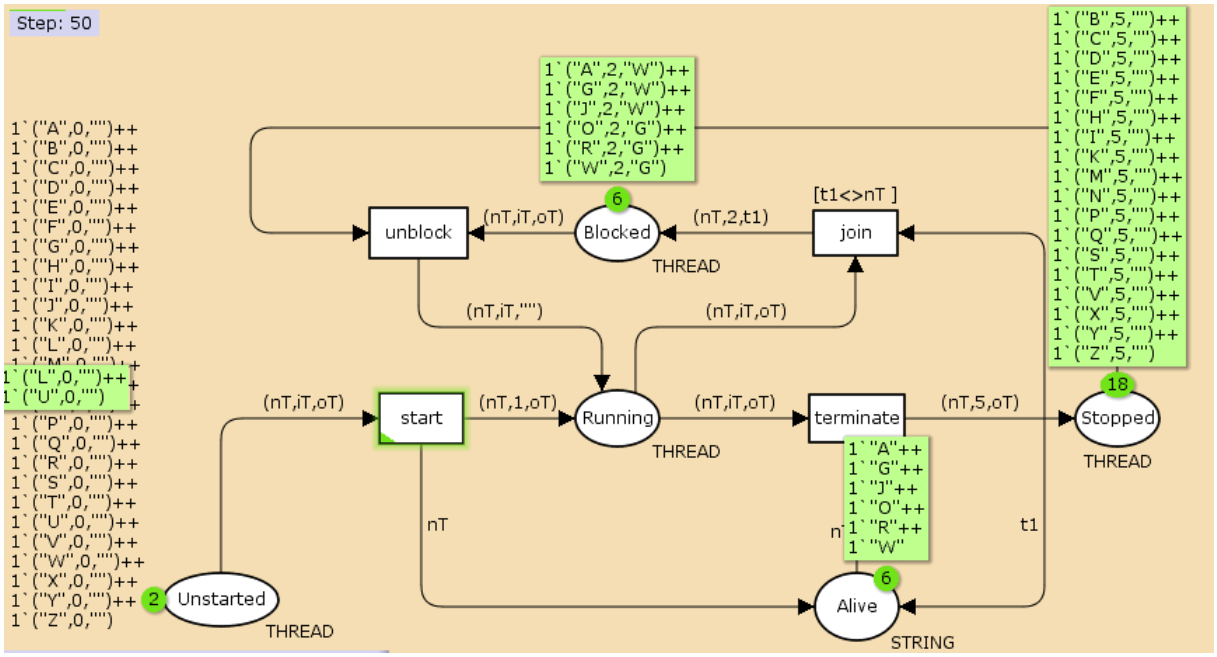
Ze stavu „**Running**“ může vlákno přejít přes přechod „**terminate**“ a tím se ukončit. Zároveň je z místa „**Alive**“ odebrán **právě** ten záznam, který odpovídá ukončovanému vláknu. Nedílnou součástí ukončování vlákna je změna jeho indexu na hodnotu 5.

Druhou a pro metodu **join** zajímavější cestou z místa „**Running**“ je přechod „**join“**, který reprezentuje vykonání této metody daným vláknem. Strážní podmínka **[t1<>nT]** aktivuje daný přechod pouze, pokud náhodně vybrané vlákno z místa „**Alive**“ není shodné s názvem aktuálního vlákna. Tato podmínka zajišťuje skutečnost, že vlákno nemůže aplikovat metodu **join** samo na **sebe**. Pokud je tato podmínka **splněna** je daný přechod **aktivní**, avšak to **nezaručuje** jeho **uskutečnění**. Stále je zde šance, že cesta povede přes přechod „**terminate**“.

Poslední akcí v síti je přechod z místa „**Blocked**“ do místa „**Running**“ přes přechod „**unblock**“. Tento přechod je aktivní pouze, pokud některé z vláken, na které se čeká v místě „**Blocked**“ není již **ukončena**, a tudíž jestli není možné přejít do místa „**Running**“. Tento fakt je zajištěn pomocí proměnné „**oT**“.

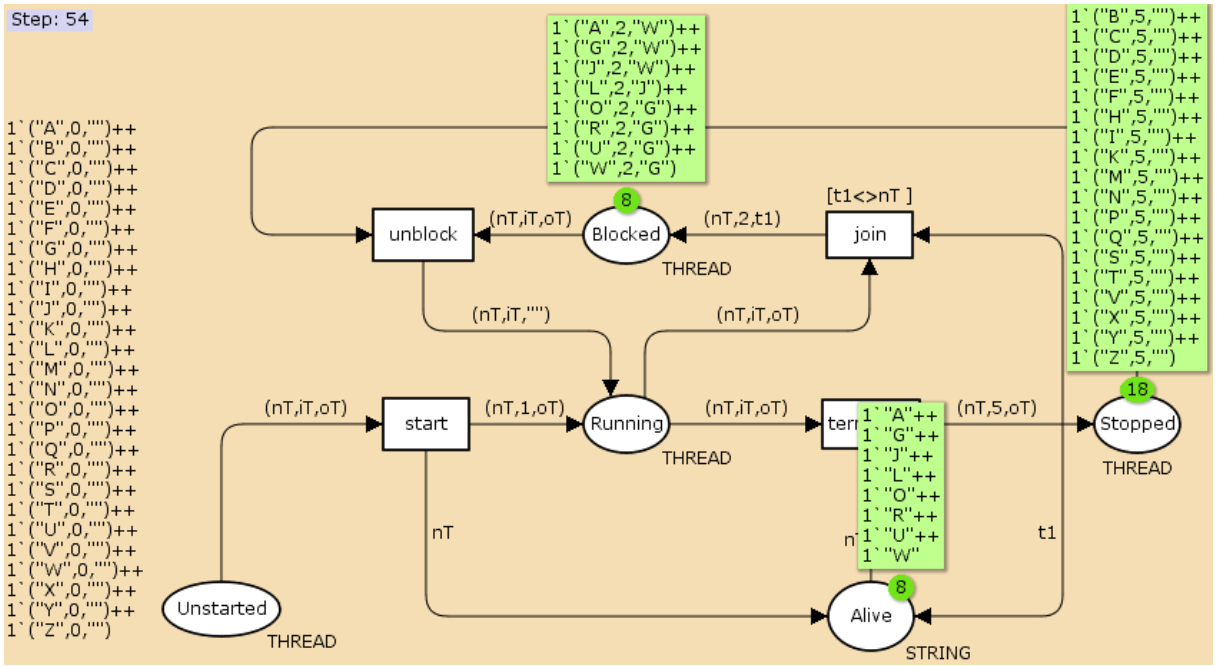
Zdvojené šipky mezi místem „**Stopped**“ a přechodem „**unblock**“, stejně jako mezi místem „**Alive**“ a přechodem „**join**“, představují situaci, kdy **token** z místa „**neodejde**“, pouze je použit pro potřeby aktivace přechodu.

Obrázek 58 ilustruje **možný stav** po 50 krocích simulace. Je patrné, že **2** vlákna ještě nebyla spuštěna (nachází se ve stavu „**Unstarted**“), **6** vláken **aplikovalo** metodu **join** a **čekají** na ukončení požadovaných vláken**.** V místě „**Alive**“ se nachází **6** názvů vláken, která jsou spuštěná. Posledních **18** vláken je již **zpracováno** a nachází se ve stavu „**Stopped**“.



Obrázek 58 - CPN - Metoda join - Po 50 krocích. Zdroj vlastní

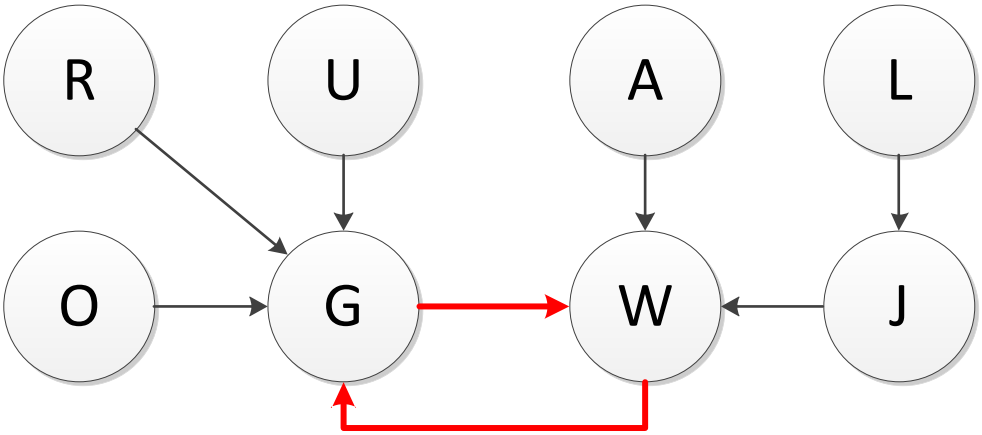
Po provedení dalších **4** kroků simulace jsou všechny přechody neaktivní, tudíž jde o mrtvou síť (viz Obrázek 59). Z obrázku je patrné, že ne všechna vlákna byla řádně ukončena a stále se nacházejí v místě „**Blocked**“. U těchto vláken nastala situace zvaná **deadlock (uváznutí)**.



Obrázek 59 - CPN - Metoda join - Mrtvá síť. Zdroj vlastní

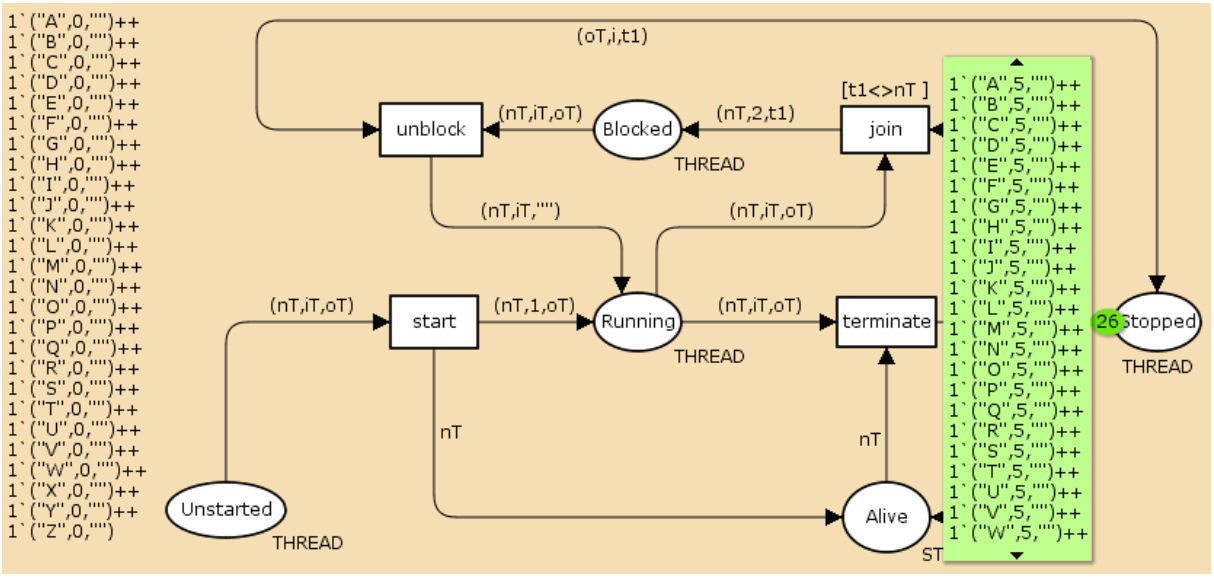
Deadlock v tomto případě vyjadřuje situaci, kdy daná vlákna vytvořila cyklus. Obrázek 60 vykresluje tuto skutečnost pomocí návazností mezi jednotlivými vlákny, tj. orientovaná hrana vyjadřuje vztah

„vlákno čeká na“ (např. vlákno „**O**“ čeká na vlákno „**G**“). Mezi vlákny „**G**“ a „**W**“ je patrné **zacyklení** (červeně vyznačeno), které vedlo k **uváznutí** (deadlock).



Obrázek 60 - Metoda join - Deadlock. Zdroj vlastní

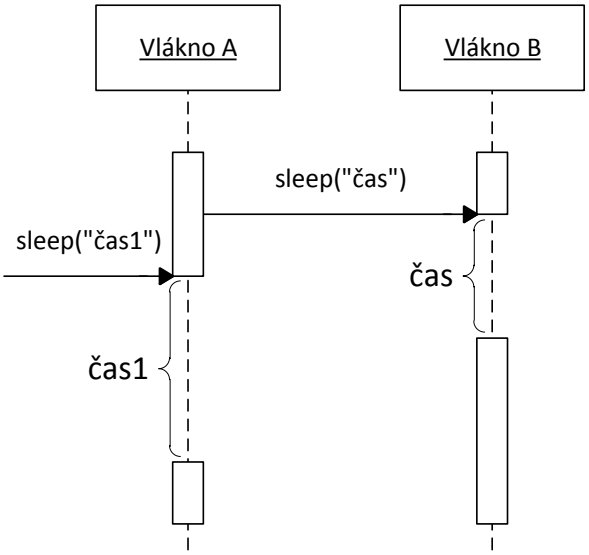
Tento problém by se dal z modelu odstranit pomocí vhodně nadefinované podmínky, která by neaktivovala přechod „**join**“ pokud by se již záznam z místa „**Alive**“ nacházel jako název některého vlákna v místě „**Blocked**“. Toto by však vedlo k velkému zkreslení reality, tudíž není využito. Metoda **join** je prakticky využívána logickým způsobem podle programátorského uvážení, tj. každý programátor musí sám zajistit, aby metodu **join** nepoužil stylem, který by způsobil **deadlock**. Jelikož je v tomto modelu metoda **join** aplikována náhodně, není možné se **zacyklení** vyhnout. Podrobněji bude tento problém probrán v kapitole 8 věnované verifikaci modelu pomocí nástrojů CPN Tools. Obrázek 61 znázorňuje požadovaný (cílový) stav sítě.



Obrázek 61 - CPN - Metoda join - Cílový stav. Zdroj vlastní

6.1.2 Metoda sleep

Další z blokovacích synchronizačních struktur je metoda **sleep**, která umožňuje „**uspat**“ vlákno na předem definovanou dobu. Na rozdíl od metody **join** může vlákno zavolat metodu **sleep samo na sebe**. Princip metody **sleep** je znázorněn na Obrázek 62.



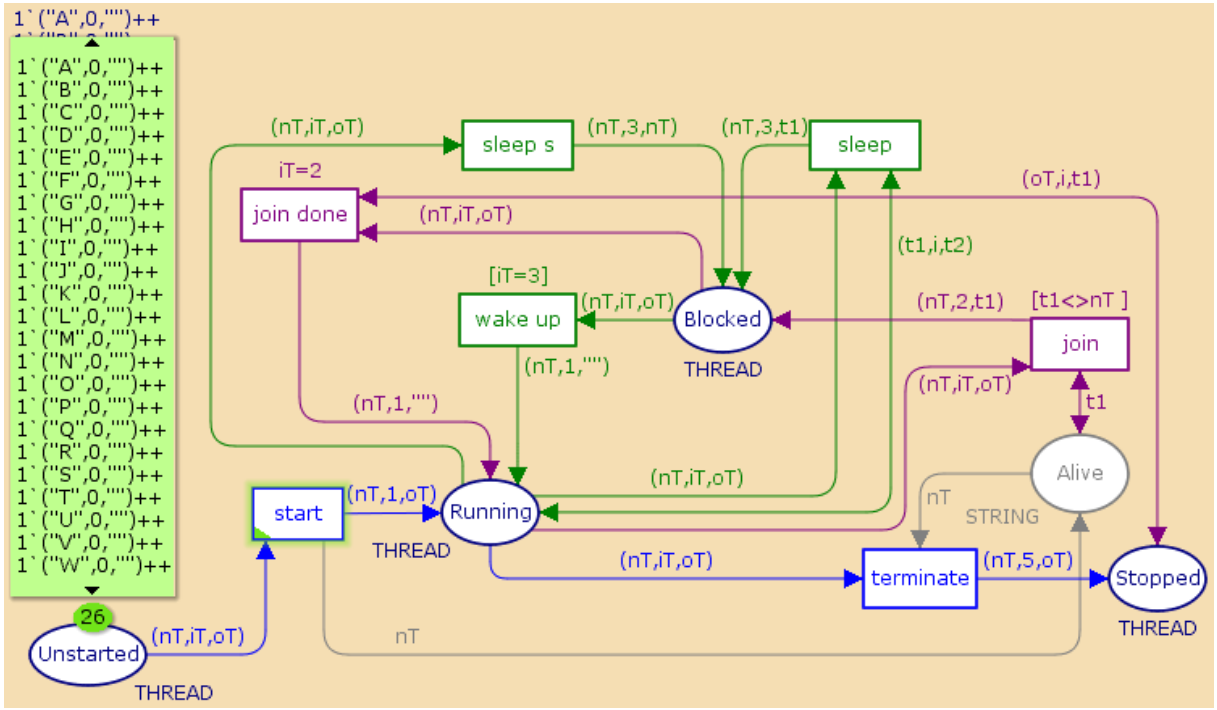
Obrázek 62 - Metoda sleep("čas"). Zdroj vlastní

Modelování metody **sleep** je obdobné metodě **join** a je zakomponováno do modelu metody **join** (model, viz příloha č. 10). **Deklarace** všech struktur zůstala **zachována**. Jedinou změnou bylo rozšíření sady pomocných proměnných o proměnnou „**t2**“ (viz Obrázek 63).

```
var t1,t2:STRING;
```

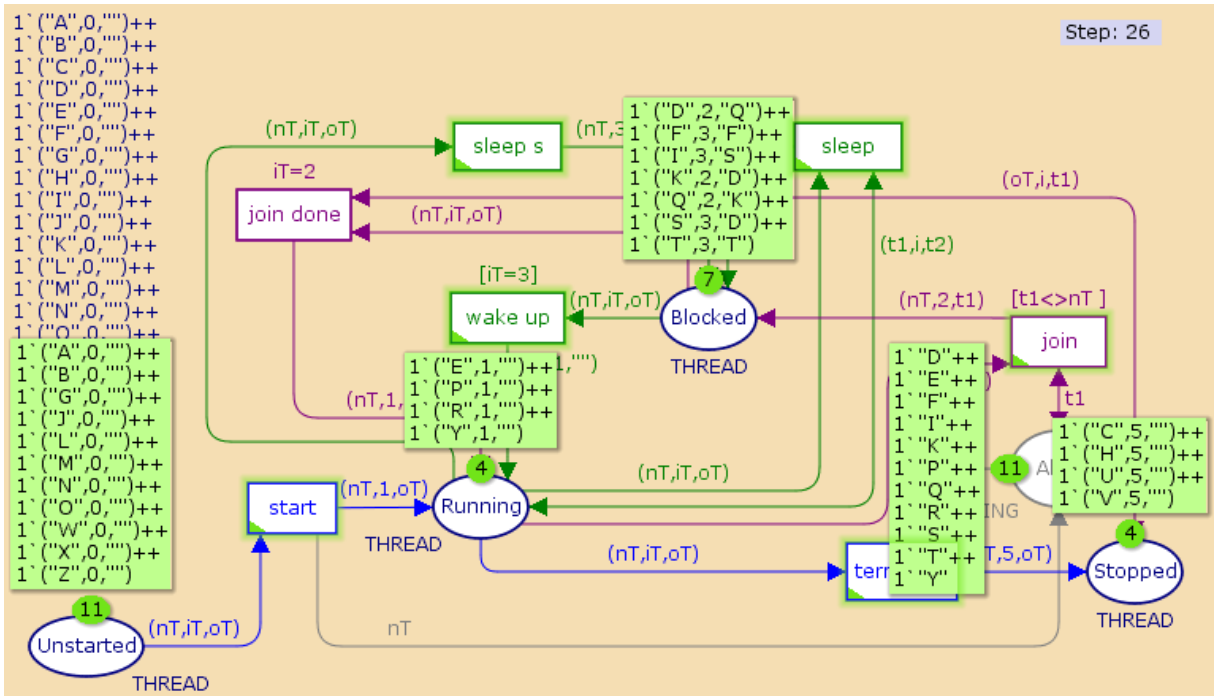
Obrázek 63 - CPN - Metoda sleep - Deklarace proměnných. Zdroj vlastní

Samotná úprava modelu spočívá v přidání přechodů „**sleep**“ a „**sleep s**“, které reprezentují jinou cestu z místa „**Running**“ do místa „**Blocked**“ (viz Obrázek 64). Uskutečnění přechodu „**sleep**“ představuje situaci, kdy je vlákno uspáno jiným vláknem nacházejícím se v místě „**Running**“. Naopak realizace přechodu „**sleep s**“ vyjadřuje situaci, kdy se vlákno uspí samo. Součástí realizace obou přechodu je změna indexu vlákna na hodnotu **3** (viz Tabulka 7).



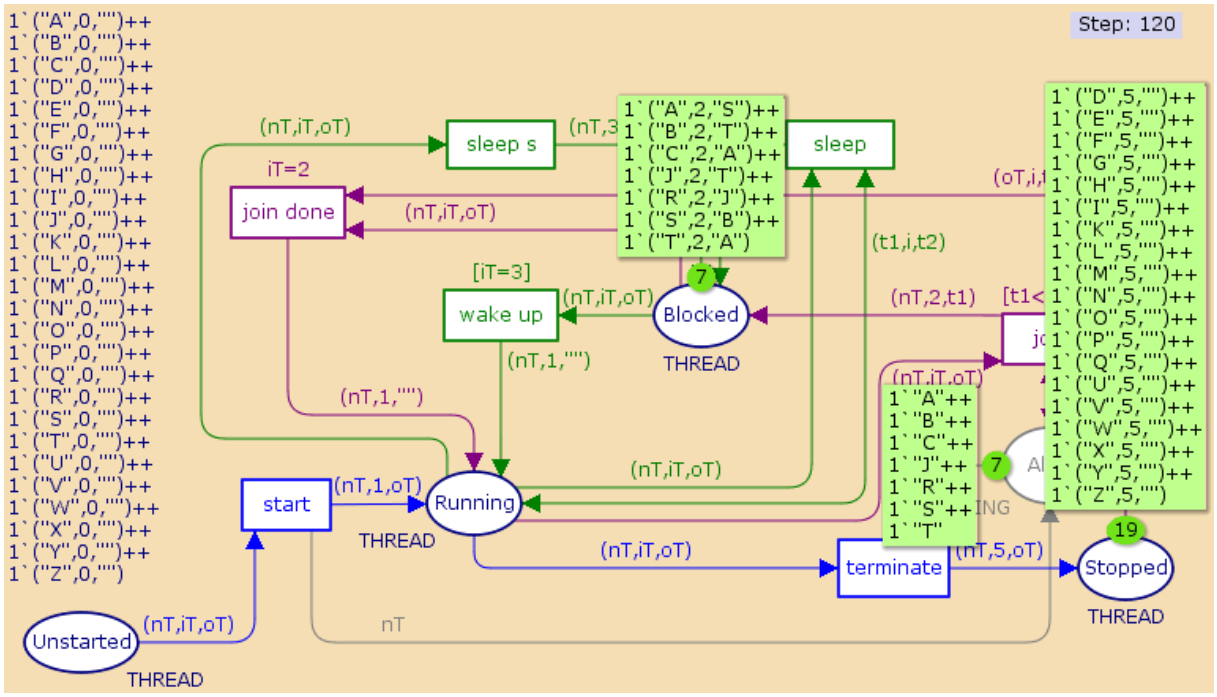
Obrázek 64 - CPN - Metoda sleep - Výchozí stav. Zdroj vlastní

Po provedení 26 kroků je patrné, že místo „**Blocked**“ obsahuje vlákna s indexy **2** a **3**, tj. vlákna byla zablokována na základě metod **join** a **sleep** (viz Obrázek 65).



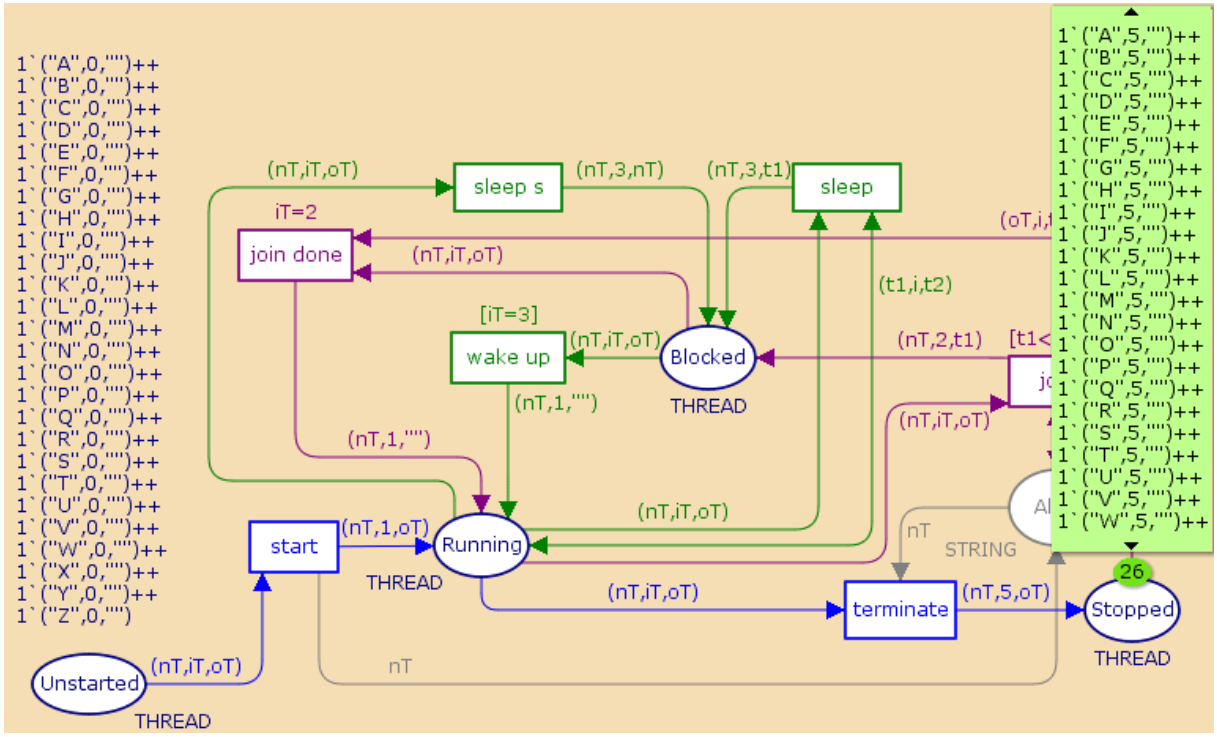
Obrázek 65 - CPN - Metoda sleep - Příklad. Zdroj vlastní

Cílový stav mrtvé sítě (viz Obrázek 66) je obdobou mrtvého značení s předchozí části této kapitoly, tj. některá vlákna **mohou** na základě metody **join** vytvořit **cyklus** a vyvolat **deadlock**. Vlákna zablokovaná metodou **sleep** vyvolat **deadlock** nemohou, jelikož **nejsou závislá** na jiných vláknech.



Obrázek 66 - CPN - Metoda sleep - Mrtvá síť. Zdroj vlastní

Cílový stav (přesun všech vláken do místa „**Stopped**“) sítě znázorňuje Obrázek 67.



Obrázek 67 - CPN - Metoda sleep - Cílový stav. Zdroj vlastní

6.2 Zamykání

Další skupinou synchronizačních struktur jsou **zamykací konstrukce**. Jelikož jednotlivá vlákna využívají sdílené zdroje (sdílení podle místa, viz kapitola 1), musí být explicitním způsobem zajištěno, aby nedocházelo k neočekávaným skutečnostem (**nekonzistentnost**). Problémy se sdílenými zdroji mohou nastat v souvislosti s dvěma situacemi:

- počítač podporuje **multiprocessing** (paralelní zpracování),
- prováděná operace **není atomická** (může dojít k přerušení).

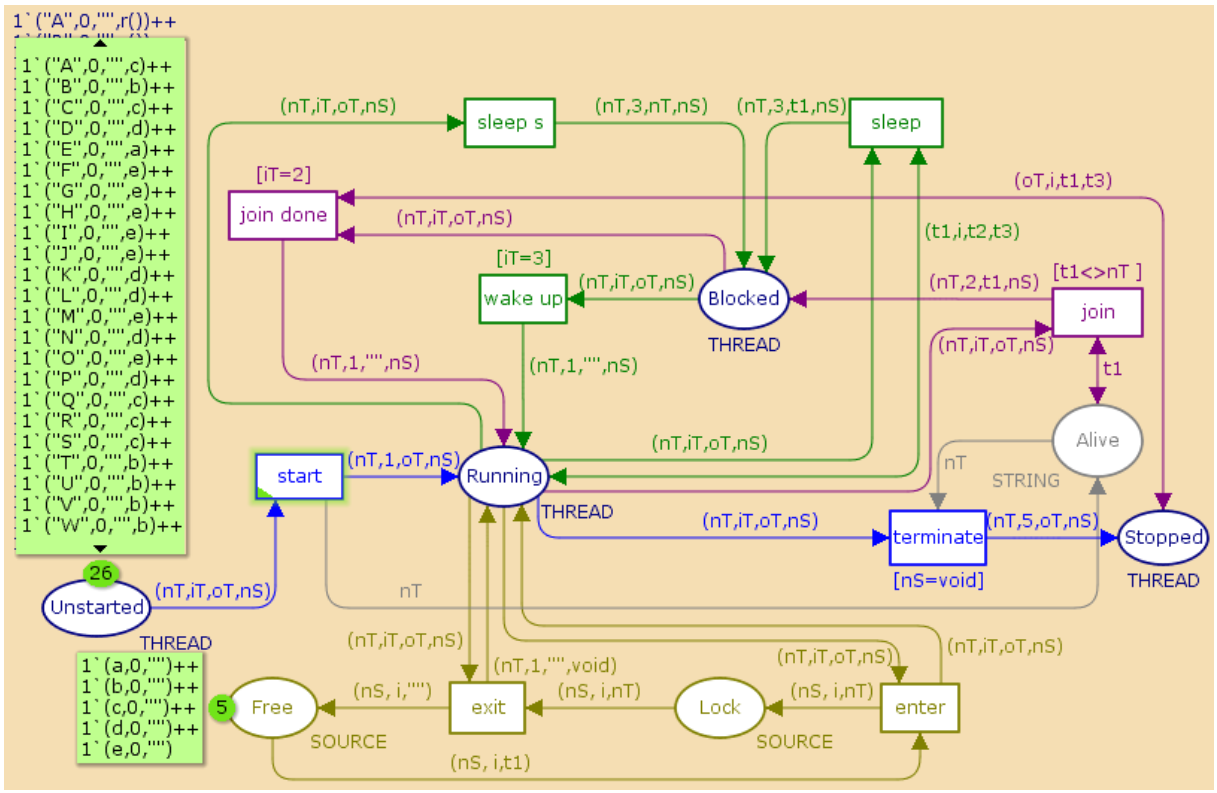
Paralelní zpracování sebou nese řadu problémů spojených se zajištěním vzájemné výlučnosti při využíváním sdílených zdrojů. Tyto problémy jsou zpravidla řešeny pomocí aktivního čekání, zákazem přerušení apod. S tímto souvisí skutečnost, že v moderních počítačích (využívajících preempci) nejsou operace prováděny atomicky. Jelikož může přerušení nastat v podstatě kdykoliv, musí být sdílený zdroj uzamknut na celou dobu, kdy jej vlákno využívá (i pokud bylo přerušeno). Část kódu, který pracuje se sdílenými zdroji, se nazývá „**Kritická sekce**“. Na druhou stranu, samotný fakt, že může dojít k nekonzistentnosti, se označuje jako „**Race condition**“.

Zamykání bude vysvětleno na konstrukci vyšších programovacích jazyků zvané „**Monitor**“. Jedná se o obdobu nízko-úrovňového **Semaforu** (synchronizační nástroj OS). **Monitor** je mocný nástroj, pomocí něhož lze v podstatě synchronizovat jakoukoli činnost vláken [18]. Třída Monitor se skládá z následujících metod:

- **enter**,
- **exit**,
- **wait**,
- **pulse**.

První dvě metody se zabývají zamykáním, resp. metoda **enter** uvozuje vstup do kritické sekce a metoda **exit** uvozuje její konec. Zbylé metody budou probrány v následující části této kapitoly věnované signalizaci.

Princip metod **enter** a **exit** znázorňuje Obrázek 68. Jedná se v podstatě o rozšíření modelu z předchozí části věnované jednoduchým blokovacím metodám (model, viz příloha č. 11).



Obrázek 68 - CPN - Zamykání - Metody enter a exit - Výchozí stav. Zdroj vlastní

Každé vlákno obsahuje nový parametr CS **NAMEOFSHARE**, který je deklarovaný jako výčet výrazů (**enumerátor**) „**a**“, „**b**“, „**c**“, „**d**“, „**e**“ a „**void**“. Pomocí funkce „**r**“ je poté v inicializačním značení místa „**Unstarted**“ tento parametr nastaven na jeden z těchto výrazů (vyjma „**void**“). Jednotlivé výrazy reprezentují název zdroje, který dané vlákno požaduje. Výraz „**void**“ naopak představuje fakt, že vlákno již žádný zdroj nepožaduje. Funkce „**r**“ zajišťuje, že v inicializačním značení budou všechna vlákna požadovat nějaký zdroj. Jednotlivé zdroje (CS **SOURCE**) jsou definovány pomocí svého názvu (CS **NAMEOFSHARE**), číselného parametru (CS **INT** – bude využit v následující subkapitole) a jménem vlákna, které ho vlastní (CS **NAMEOFTHREAD**). Deklaraci upravených či nových struktur znázorňuje Obrázek 69.

```
colset NAMEOFSHARE = with a|b|c|d|e|void;
colset THREAD = product NAMEOFTHREAD *  TYPEOFTHREAD  * OTHERTHREAD * NAMEOFSHARE;
colset SOURCE = product NAMEOFSHARE * INT *  NAMEOFTHREAD;
var nS: NAMEOFSHARE;
var t3:NAMEOFSHARE;
fun r() =
  let
     val x = NAMEOFSHARE.ran();
  in
    if x = void then
      r()
    else
      x
  end;
```

Obrázek 69 - CPN - Zamykání - Deklarace. Zdroj vlastní

Celá činnost zamykání je prováděna v místě „**Running**“. Zároveň je úmyslně abstrahováno od blokování vláken, která čekají na zdroj uzamknutý jiným vláknem. Toto blokování zpravidla probíhá na nižší úrovni abstrakce, a proto je součástí stavu „**Running**“. Z místa „**Running**“ může vlákno pokračovat pěti různými cestami:

- Pokud je splněna strážní podmínka na přechodu „**terminate**“ může být vlákno ukončeno. Tím je automaticky odebrán záznam tohoto vlákna z místa „**Alive**“. Vlákno však nemůže skončit dříve, než dokončí práci se sdíleným zdrojem, který mu byl náhodně přidělen funkcí „**r**“. Tento fakt, je splněn, pokud parametr CS **NAMEOFSHARE** obsahuje výraz „**void**“.
- Cesty přes přechody „**join**“ a „**sleep**“ jsou realizovány stejným způsobem jako v předchozí části této kapitoly.
- Pokud vlákno požaduje zdroj a ten je volný (v místě „**Free**“), může přejít přes přechod „**enter**“, tím vstoupit do kritické sekce a zdroj uzamknout (přesunout do místa „**Lock**“). Součástí uzamykání je nastavení názvu vlákna jako třetí parametr (CS **NAMEOFTHREAD**) zdroje v místě „**Lock**“.
- Jestliže vlákno vlastní zdroj (tento zdroj je uzamčen v místě „**Lock**“ a obsahuje název vlákna jako svůj třetí parametr), může přejít přes přechod „**exit**“, čímž opustí kritickou sekci a zdroj může být přidělen jinému vláknu. Tímto přechodem je automaticky vymazán záznam o vlastnictví zdroje (již zmíněný třetí parametr zdroje). Vláknu je zároveň nastavena hodnota třetího parametru (CS **NAMEOFSHARE**) na výraz „**void**“.

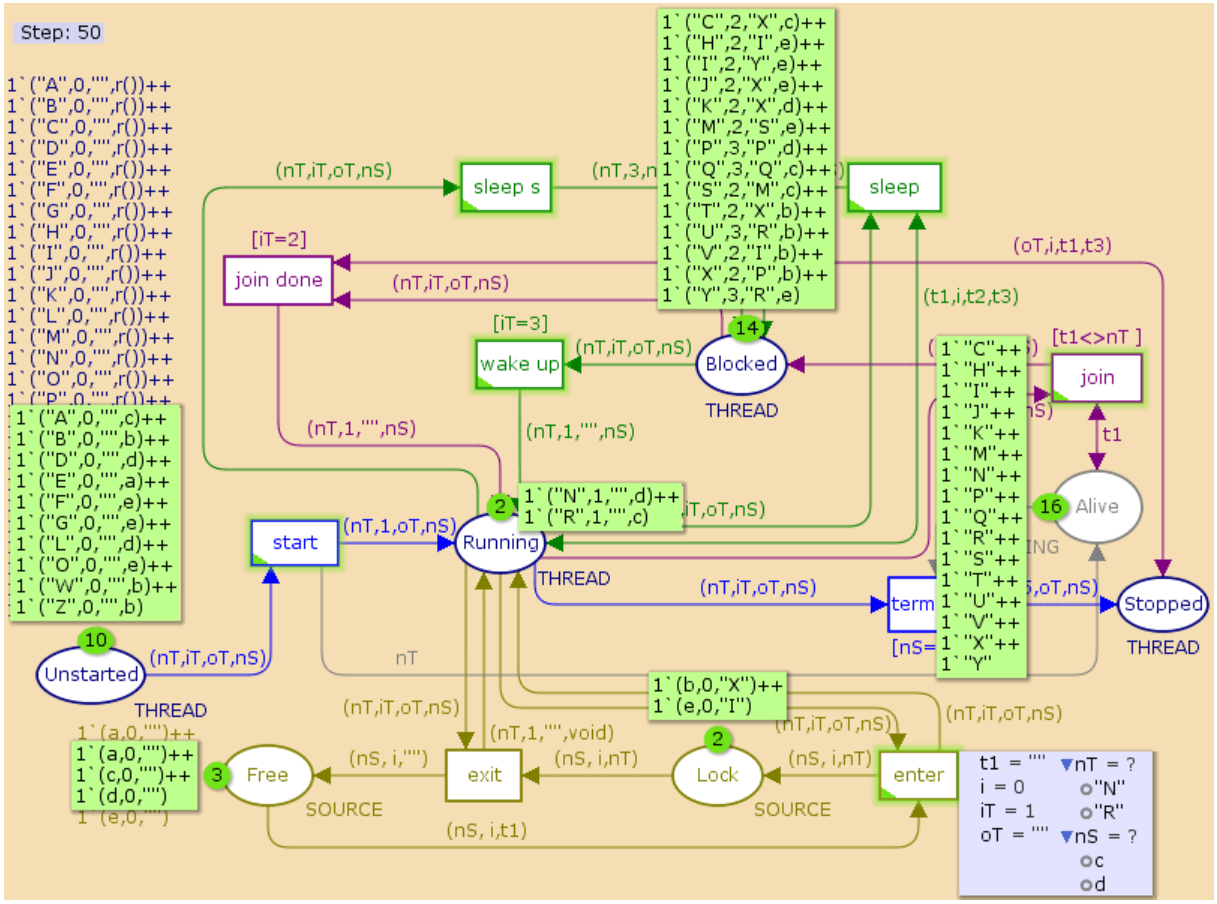
Pomocí tohoto mechanizmu je zaručeno, že žádný zdroj nebude používán, pokud jej drží jiné vlákno.

Obrázek 70 ilustruje příklad sítě po 50 krocích simulace. Z obrázku vyplývá následující situace:

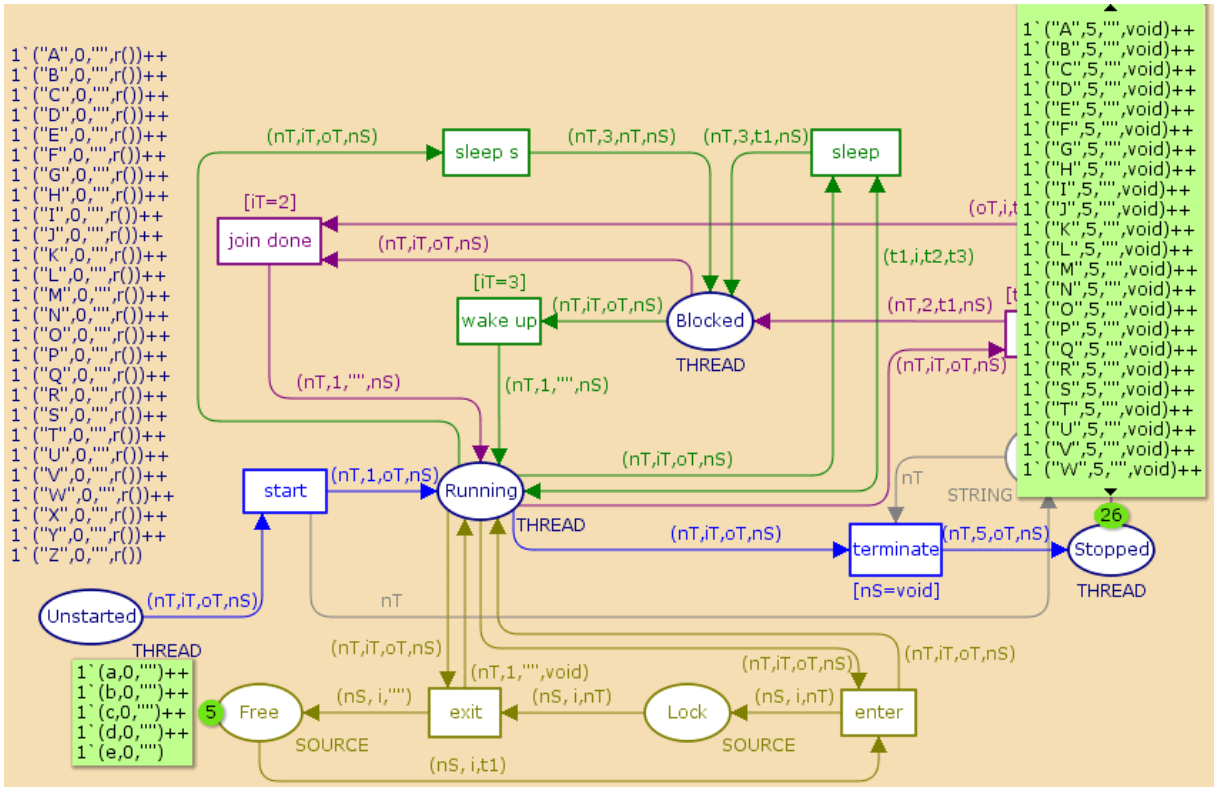
- 10 vláken zatím nebylo spuštěno (nacházejí se v místě „**Unstarted**“),
- 10 vláken je zablokováno („**Blocked**“) na základě metody **join** (index=**2**) a 4 vlákna pomocí metody **sleep** (index=**3**),
- 2 vlákna běží („**Running**“),
- 16 vláken je aktivních („**Alive**“ = „**Blocked**“ + „**Running**“),
- 2 zdroje jsou uzamčeny (v místě „**Lock**“) vlákny „X“ a „I“,
- 3 zdroje jsou volné (v místě „**Free**“).

Další akcí v síti bude uskutečnění přechodu „**enter**“, tj. vlákno „**N**“, resp. „**R**“ vstoupí do kritické sekce a tím uzamkne zdroj „**c**“, resp. „**d**“.

Jelikož model využívá metodu **join**, může nastat **deadlock**. Cílový stav sítě ilustruje Obrázek 71.



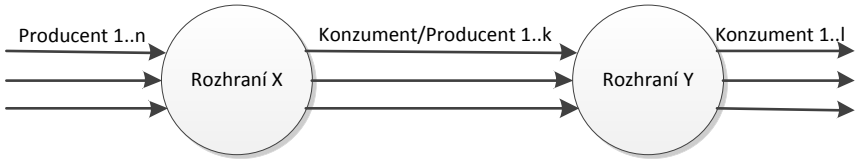
Obrázek 70 - CPN - Zamykání - enter a exit - Příklad. Zdroj vlastní



Obrázek 71 - CPN - Zamykání - enter a exit - Cílový stav. Zdroj vlastní

6.3 Signalizace

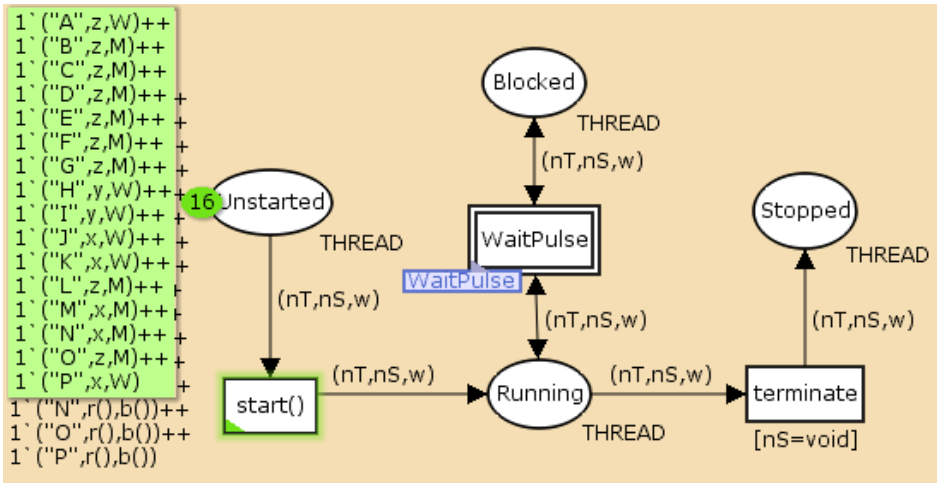
Signalizace se používá v případech, kdy spolu vlákna komunikují. Jedná se tedy o situaci, kdy jedno vlákno vytváří výstup, který je vstupem dalšího vlákna. Takovýmto způsobem může být definována celá procesní činnost programu. Problém nastává, pokud existují prodlevy mezi vlákny z hlediska kontinuity. Z tohoto důvodu musí některá vlákna čekat na svoje vstupní data (aktivně), čímž je zbytečně spotřebováván strojový čas počítače. Z tohoto důvodu se používá signalizace, která umožňuje zablokovat čekající vlákna do doby, než bude potřebný vstup dostupný. Vlákna, jejichž výstupy jsou použity, jako vstupy jiných vláken se označují jako „**Producenti**“. Naopak, vlákna, která tyto výstupy využívá jako své vstupy, jsou označovány jako „**Konzumenti**“. Obrázek 72 ilustruje příklad návaznosti vláken. Producenti zde představují vlákna, která vytvářejí výstupy pro „**Rozhraní X**“. Skrze toto rozhraní tyto výstupy přejímají vlákna označená jako „**Producenti/Konzumenti**“ jako své vstupy a přetvářejí je na výstupy pro „**Rozhraní Y**“. Z „**Rozhraní Y**“ převezmou tyto data finální konzumenti.



Obrázek 72 - Návaznost vláken. Zdroj vlastní

Operační systém obsahuje základní sadu nízko-úrovňových signalizačních nástrojů jako např. Semafor, Mutex apod. V následujícím bude vysvětlen princip metod **pulse** a **wait** třídy **Monitor**. Metody **pulse** a **wait** umožňují pomocí mezivláknové signalizace koordinovat blokování/odblokování v závislosti na plnění blokovací podmínky. Pokud např. vlákno čeká na vstup (výstup jeho producenta), může na sebe zavolat metodu **wait** a tím se zablokovat. Vlákno se opět odblokuje, pokud jiné vlákno (jeho producent) zavolá metodu **pulse**. Tento princip je u konzumenta zpravidla realizován pomocí nekonečného cyklu, jehož ukončení je uvozováno podmínkou, která určuje, zda jsou vstupy připraveny, či nikoliv. V těle cyklu je volána metoda **wait**, čímž je vlákno zablokováno a neprovádí se nekonečný cyklus. Jelikož blokovací podmínka reprezentuje operaci nad sdíleným zdrojem (vstupem), musí se celá operace nacházet v **kritické sekci**. Pokud producent připraví svůj výstup, zavolá metodu **pulse**, která odblokuje jedno vlákno (jeho **konzumenta**).

Pro znázornění činnosti metod **pulse** a **wait** je pro zpřehlednění využita **hierarchická** abstrakce v **CPN**, resp. **HCPN**. Obrázek 73 představuje základní síť, která je tvořena standardními stavy vláken. Výchozí značení obsahuje 16 vláken („A“ – „P“). Celý model je obsažen v příloze č. 12.



Obrázek 73 - CPN - Signalizace - pulse a wait - Základní síť. Zdroj vlastní

Pro potřeby modelu signalizace byl definován nový parametr vlákna (CS **WORK**), který reprezentuje jeho postavení, tedy zda se jedná o producenta (**P**), konzumenta (**C**), či obojí (**PC**). Parametr, který nese název zdroje (CS **NAMEOFSHARE**) pracuje na stejném principu jako v předchozí části věnované zamykání. Jediná úprava spočívá ve změně samotných výrazů, tak aby odpovídaly současné situaci, tj. názvy zdrojů budou nabývat pouze výrazů „**x**“, „**y**“ a výraz pro nepřidělený zdroj „**void**“. V tomto případě se pod pojmem zdroj míní **sdílené rozhraní** mezi vlákny. Dvojice těchto parametrů (**NAMEOFSHARE** a **WORK**) je přidána jako nový parametr vlákna (CS **TYPE**), viz Obrázek 74. Pro zjednodušení modelu se zbývající parametry modelu neuvažují, tj. vlákno se skládá pouze ze svého názvu (CS **NAMEOFTHREAD**) a typu (CS **TYPE**).

```
colset NAMEOFSHARE = with x|y|void;
colset WORK = with P|C|PC;
colset TYPE = product NAMEOFSHARE * WORK;
colset THREAD = product NAMEOFTHREAD * TYPE;
```

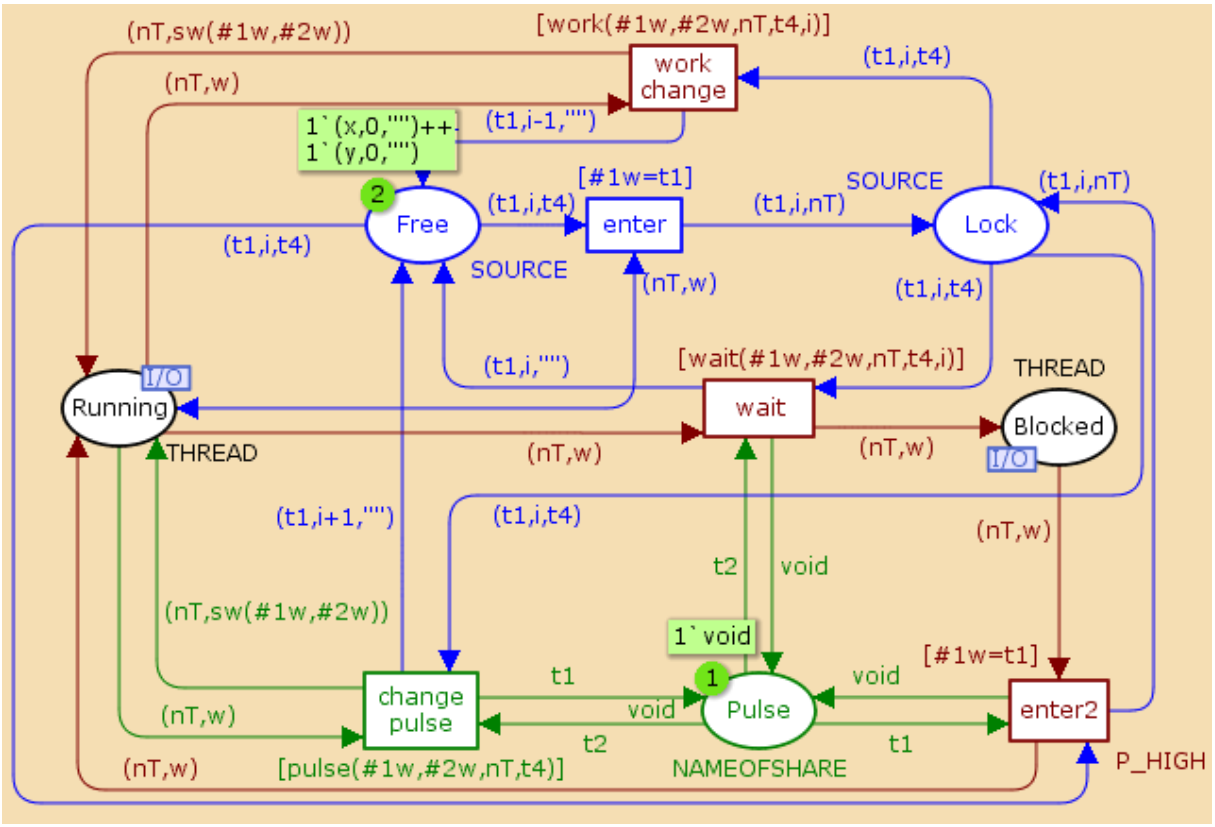
Obrázek 74 - CPN - Signalizace - pulse a wait - Deklarace CS. Zdroj vlastní

V inicializačním značení místa „**Unstarted**“ jsou pomocí funkce „**r**“ nastaveny (pseudonáhodným způsobem) parametry CS **TYPE** (viz Obrázek 75). Tato funkce automaticky přiřazuje „**konzumentům**“ rozhraní „**y**“ a „**producentům**“, resp. „**konzumentům/producentům**“ rozhraní „**x**“. Přechod „**WaitPulse**“ reprezentuje samostatný modul (submodel), jehož struktura je znázorněna na Obrázek 76. Samotný zdroj, deklarovaný stejným způsobem jako v předchozí části věnované zamykaní, reprezentuje sdílené rozhraní mezi producenty a konzumenty. První parametr představuje označení tohoto rozhraní (buď „**x**“ anebo „**y**“). Druhým parametrem zdroje (rozhraní) je blokovací podmínka, která je nastavována producentem a kontrolována konzumentem. V tomto případě blokovací podmínka reprezentuje počet dostupných vstupů/výstupů. Poslední parametr značí název vlákna, které se nachází v kritické sekci daného rozhraní.

```
fun pulse(a,b,c,d) =
  if b=PC andalso a=y andalso c=d then
     true
  else if b=P andalso a=x andalso c=d then
     true
  else
     false
fun work(a,b,c,d,e) =
  if b=PC andalso a=x andalso c=d andalso e<>0 then
     true
  else if b=C andalso a=y andalso c=d andalso e<>0 then
     true
  else
     false
fun wait(a,b,c,d,e) =
  if b=PC andalso a=x andalso c=d andalso e=0 then
     true
  else if b=C andalso a=y andalso c=d andalso e=0 then
     true
  else
     false
```

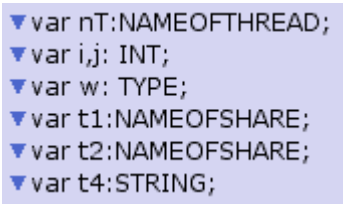
```
fun sw(a,b) =
  if b=PC andalso a=x then
       (y,b)
  else if b=PC then
       (x,b)
  else
       (a,b)
fun r() =
  let
     val b = WORK.ran();
  in
     if(b=C) then
        (y,b)
     else
        (x,b)
  end;
```

Obrázek 75 - CPN - Signalizace - pulse a wait - Deklarace funkcí. Zdroj vlastní



Obrázek 76 - CPN - Signalizace - pulse a wait - Modul „WaitPulse“. Zdroj vlastní

Typ vlákna je mezi místy přenášen pomocí proměnné „**w**“. Všechny proměnné používané v modelu signalizace znázorňuje Obrázek 77.



Obrázek 77 - CPN - Signalizace - pulse a wait - Deklarace proměnných. Zdroj vlastní

Modul „**WaitPulse**“ je vizuálně rozčleněn na tři nezávislé úseky:

- vstup a výstup z **kritické sekce** (modrá část),
- metoda **wait** (červeno-hnědá část),
- metoda **pulse** (zelená část).

Každá akce v modelu musí být uvozována vstupem do kritické sekce (přechod „**enter**“). Tím je rozhraní přidruženo vláknu, které jej uzamklo. Pokud vlákno vlastní dané rozhraní, může vůči němu uplatnit metodu **wait**, nebo **pulse** v závislosti na svém typu.

Z místa „**Running**“ může tedy prvně vlákno přejít pouze přes přechod „**enter**“, čímž uzamkne požadované rozhraní (odpovídající záznam z místa „**Free**“ přejde do místa „**Lock**“). Pokud má vlákno uzamčené rozhraní, může v závislosti na typu provést následující akce:

- Pokud je vlákno „**producent**“:
  - Může přejít přes přechod „**change pulse**“, čímž **inkrementuje** hodnotu (o **1**) blokovací podmínky v jeho rozhraní (uzamknutém). Aktivaci tohoto přechodu realizuje funkce „**pulse**“, která umožní průchod pouze vláknu, které je buď „**producent**“, který požaduje rozhraní „**x**“, nebo „**konzument/producent**“, který požaduje rozhraní „**y**“. Zároveň je kontrolován fakt, že je požadovaný zdroj (rozhraní) volný.
  - Uskutečněním přechodu „**change pulse**“ je zároveň do místa „**Pulse**“ umístěn název rozhraní, jehož blokovací podmínka byla inkrementována.
- Pokud je vlákno „**konzument**“:
  - Pokud je hodnota blokovací podmínky větší než **0** (zjištěno pomocí funkce „**work**“), tj. existují nějaká vstupní data, může vlákno přejít přes přechod „**work change**“, čímž **dekrementuje** hodnotu (o **1**) blokovací podmínky v jeho rozhraní a zároveň dané rozhraní „odemkne“ (opustí **kritickou sekci**, tj. přesune odpovídající záznam z místa „**Lock**“ do místa „**Free**“).
  - Pokud je hodnota blokovací podmínky rovna **0** (zjištěno pomocí funkce „**wait**“), tj. vlákno nemůže zpracovávat požadované vstupy, je realizován přechod „**wait**“, čímž se vlákno přesune do stavu „**Blocked**“. Součástí přechodu „**wait**“ je i přesun

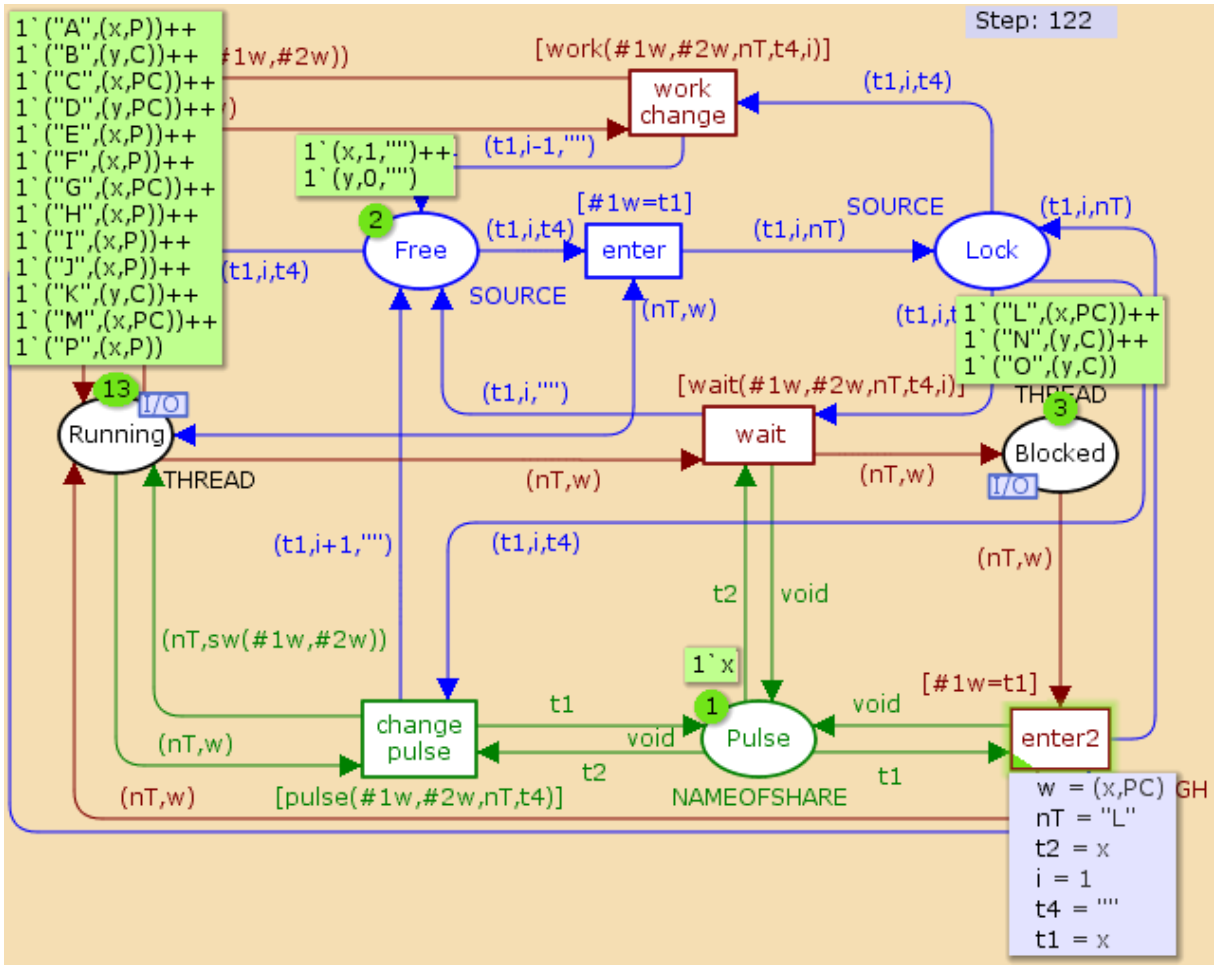
odpovídajícího záznamu z místa „**Lock**“ do místa „**Free**“ (opuštění **kritické sekce**). Zároveň je hodnota tokenu v místě „**Pulse**“ nastavena na „**void**“. Tím je zaručeno, že nebude využit „**prošlý**“ záznam z místa „**Pulse**“.

- Jestliže je vlákno v místě „**Blocked**“ a zároveň je hodnota tokenu v místě „**Pulse**“ nastavena na název vláknem požadovaného rozhraní (které je zároveň v místě „**Free**“), může být uskutečněn přechod „**enter2**“, čímž je vlákno odblokováno (přesunuto do místa „**Running**“) a zároveň mu je přiřazeno (uzamknuto) požadované rozhraní.

- Pokud je vlákno „**konzument/producent**“:
  - Z výchozího stavu se tento typ vlákna chová jako typický „**konzument**“, který požaduje splnění blokovací podmínky na rozraní „**x**“.
  - Po aplikaci přechodu „**work change**“ se vlákno pomocí funkce „**sw**“ začne chovat jako „**producent**“, který vykonává přechod „**change pulse**“ vůči rozhraní „**y**“. Po vykonání přechodu „**change pulse**“ je pomocí stejné funkce („**sw**“) nastaveno vlákno zpět na „**konzumenta**“.

Tímto mechanizmem je namodelován princip metod **pulse** a **wait** třídy **Monitor**. Z důvodu všeobecnosti interakce mezi „**producenty**“ a „**konzumenty**“ (popř. „**konzumenty/producenty**“) není v modelu definována ukončovací podmínka, tj. mechanizmus, který oznámí jednotlivým vláknům, že mají přestat „**konzumovat**“, resp. „**produkovat**“. Na základě tohoto zjednodušení, tedy nebude nikdy aktivní přechod „**terminate**“. Z hlediska verifikace modelu se tedy jedná o mrtvý přechod (**Dead Transition**).

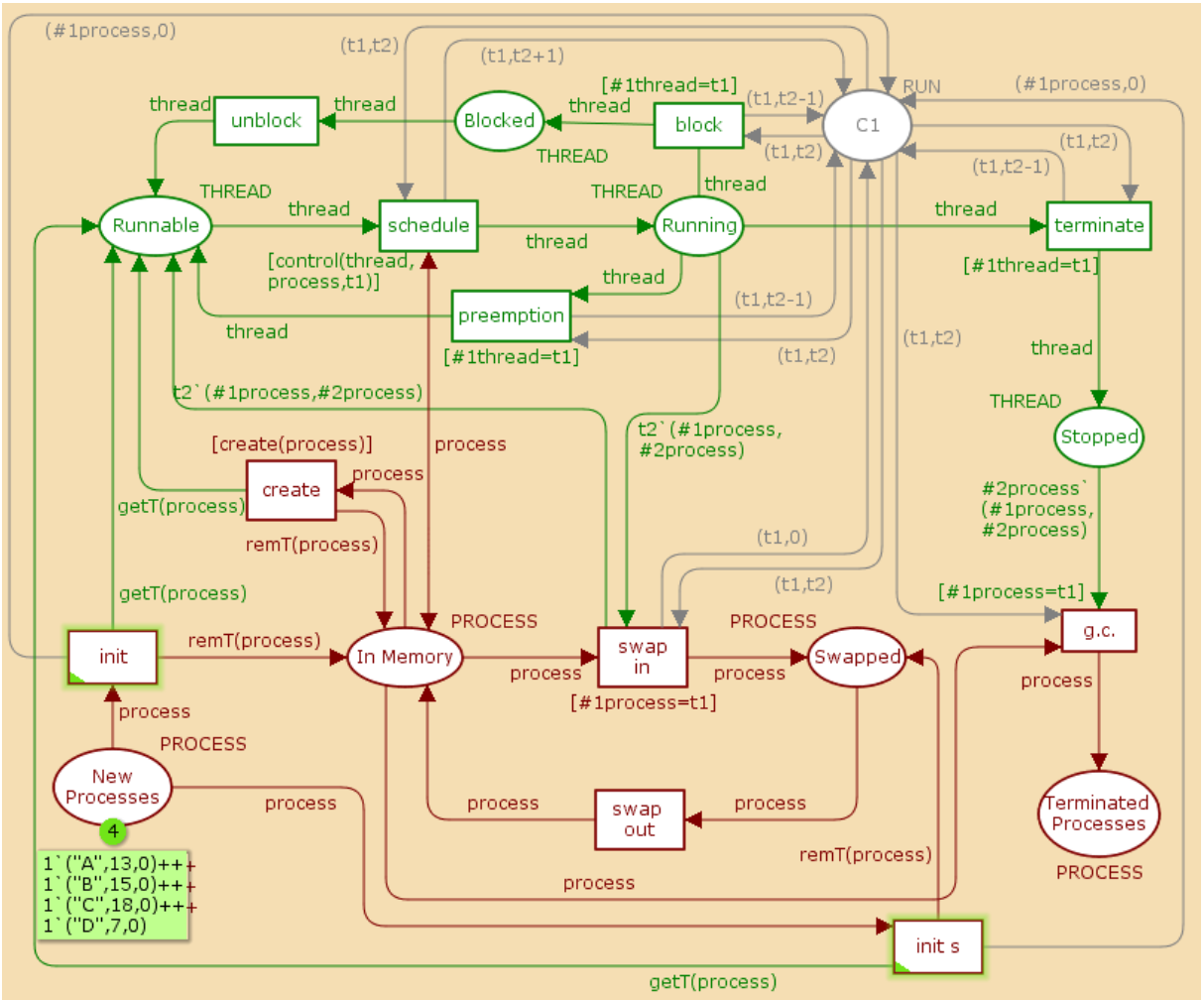
Obrázek 78 znázorňuje příklad modelu metod pulse a wait po 122 krocích simulace. V místě „**blocked**“ se nacházejí 3 vlákna. Dvě vlákna jsou typu „**konzument**“ požadující vstupní data z rozhraní „**y**“ a jedno vlákno je typu „**konzument/producent**“, které požaduje vstup z rozhraní „**x**“. V místě „**Free**“ se nacházejí obě rozhraní, avšak pouze rozhraní „**x**“ obsahuje jednu jednotku dat (vstupu pro jeho „**konzumenty**“). Z tohoto důvodu bude vlákno „**L**“ odblokováno a přesunuto zpět do místa „**Running**“.



Obrázek 78 - CPN - Signalizace - pulse a wait - Příklad. Zdroj vlastní

7 Vlákna v kontextu procesního modelu

V této kapitole bude modelován poslední pohled definovaný v kapitole 4. **Kontextuálním** pohledem se v tomto směru rozumí **interakce** vláken s objekty **hierarchicky nadřazenými**. Tímto nadřazeným objektem je samozřejmě **proces**, který sdružuje využívaná **vstupně výstupní** zařízení a přidělenou **operační paměť** do jednoho **kompaktního** univerzálního celku. Jak již bylo jednou zmíněno, vlákna z hlediska kontextu interagují na jedné straně s **procesorem/jádrem** (viz kapitola 5) a na druhé straně s **procesem**, který rozšiřuje možné stavy vláken o **swapování** (**Mid Term Scheduling**, viz kapitola 1.6). Obrázek 79 znázorňuje model rozšířeného stavového modelu, který rozlišuje stavy, které se týkají **procesů** (červeno-hnědá část) a které **vláken** (zelená část). Model je obsažen v příloze č. 13.



Obrázek 79 - CPN - Vlákna v kontextu procesního modelu - Výchozí stav. Zdroj vlastní

Místa s přidělenou CS **PROCESS** jsou deklarována jako trojice parametrů (viz Obrázek 80):

- CS **NAMEOFPROCESS** – obsahuje název vlákna (**string**),
- CS **TTOGO** – reprezentuje deklaraci počtu vláken procesu (**integer** z rozsahu <0,20>),

- CS **NUMBEROFTHREADS** – představuje aktuální počet spuštěných vláken procesu (**integer**).

Jednotlivá vlákna (CS **THREAD**) jsou deklarována jako dvojice parametrů:

- CS **NAMEOFPROCESS** – obsahuje označení procesu, který vlákno vlastní (**string**),
- CS **TTOGO** – obsahuje max. počet vláken svého procesu (**integer** z rozsahu <0,20>).

Model dále obsahuje jedno kontrolní místo „**C1**“, které je deklarováno jako CS **RUN** (dvojice parametrů CS **NAMEFOPROCESS** a CS **INT**).

```
colset NAMEOFPROCESS = string;
colset TTOGO = int with 1..20;
colset NUMBEROFTHREADS = int;
colset PROCESS = product NAMEOFPROCESS  * TTOGO * NUMBEROFTHREADS;
colset THREAD =  product NAMEOFPROCESS * TTOGO;
colset RUN = product NAMEOFPROCESS * INT;
```

Obrázek 80 - CPN - Vlákna v kontextu procesního modelu - Deklarace CS. Zdroj vlastní

Jednotlivá vlákna a procesy jsou přenášena přes své hrany (barevně odlišené) pomocí proměnných „**thread**“ a „**process**“ (viz Obrázek 81). Model dále obsahuje dvě pomocné proměnné, které přenášejí parametry vláken z kontrolního místa „C1“, jehož princip bude vysvětlen v následujícím.

```
var thread: THREAD;
var process: PROCESS;
var t1: NAMEOFPROCESS;
var t2: INT;
```

Obrázek 81 - CPN - Vlákna v kontextu procesního modelu - Deklarace proměnných. Zdroj vlastní

V inicializačním značení místa „**New Processes**“ jsou definovány 4 různé procesy („A“, „B“, „C“ a „D“). Každému procesu je pomocí funkce „**r**“ pseudonáhodným způsobem vygenerováno číslo z CS **TTOGO** (viz Obrázek 82), které reprezentuje celkový počet vláken tohoto procesu. Obrázek 83 znázorňuje toto inicializační značení pod místem „**New Processes**“.

Z výchozího stavu (viz Obrázek 79) může proces aplikovat přechod „**init**“, čímž se vytvoří (pomocí funkce „**getT**“) jedno vlákno (daného procesu) v místě „**Runnable**“. Proces samotný přejde do místa „**In Memory**“, který reprezentuje fyzickou přítomnost procesu v operační paměti. Ještě předtím mu je však pomocí funkce „**remT**“ inkrementována (o 1) hodnota parametru **NUMBEROFTHREADS**. Součástí realizace přechodu „**init**“ je vytvoření nového záznamu v místě „**C1**“, který ponese **název** vytvářeného procesu (jako první parametr) a **nulovou** hodnotu druhého parametru. Druhá cesta z místa „**New Processes**“ vede přes přechod „**init s**“, který pracuje na stejném principu jako přechod „**init**“, pouze s tím rozdílem, že proces přejde do místa „**Swapped**“ (namísto „**In Memory**“).

Vlákna z místa „**Runnable**“ mohou pokračovat přes přechod „**schedule**“ pouze pokud je splněna strážní podmínka tohoto přechodu, resp. funkce „**control**“. Přechod „**schedule**“ je aktivní, jestliže se proces daného vlákna nachází v operační paměti (v místě „**In Memory**“). Pokud je přechod uskutečněn, je automaticky inkrementována hodnota (o **1**) druhého parametru odpovídajícího záznamu v místě „**C1**“. V místě „**In Memory**“ může proces pomocí přechodu „**create**“ vytvářet nová vlákna (pomocí funkcí „**getT**“ a „**remT**“). Funkce „**create**“ zaručuje, že nebude vytvořeno více vláken, než deklarovány počet (CS **TTOGO**).

```
fun r() = TTOGO.ran();
fun remT(x) =
  let
     fun a1(r,s,t) = r;
     fun a2(r,s,t) = s;
     fun a3(r,s,t) = t;
     val a = a1(x);
     val b = a2(x);
     val c = a3(x);
  in
    (a,b,c+1)
  end;
fun getT(x) =
  let
     fun a1(r,s,t) = r;
     fun a2(r,s,t) = s;
     val a = a1(x)
     val b = a2(x)
  in
    (a,b)
  end;
fun control(x,y,z) =
  let
     fun a1(r,s,t) = r;
     fun a2(r,s) = r;
     val a = a1(y);
     val b = a2(x);
  in
    a=b andalso b = z
  end;
fun create(x) =
  let
     fun a1(r,s,t) = s;
     fun a2(r,s,t) = t;
     val a = a1(x);
     val b = a2(x);
  in
     a<>b
  end;
```

Obrázek 82 - CPN - Vlákna v kontextu procesního modelu - Deklarace funkcí. Zdroj vlastní

Z místa „**Running**“ může vlákno pokračovat 4 různými cestami:

- přes přechod „**block**“ vlákno přejde do místa „**Blocked**“,
- přes přechod „**terminate**“ vlákno přejde do místa „**Stopped**“,
- přes přechod „**preemption**“ vlákno přejde do místa „**Runnable**“,
- přes přechod „**swap in**“ vlákno, resp. vlákna přejdou do místa „**Runnable**“.

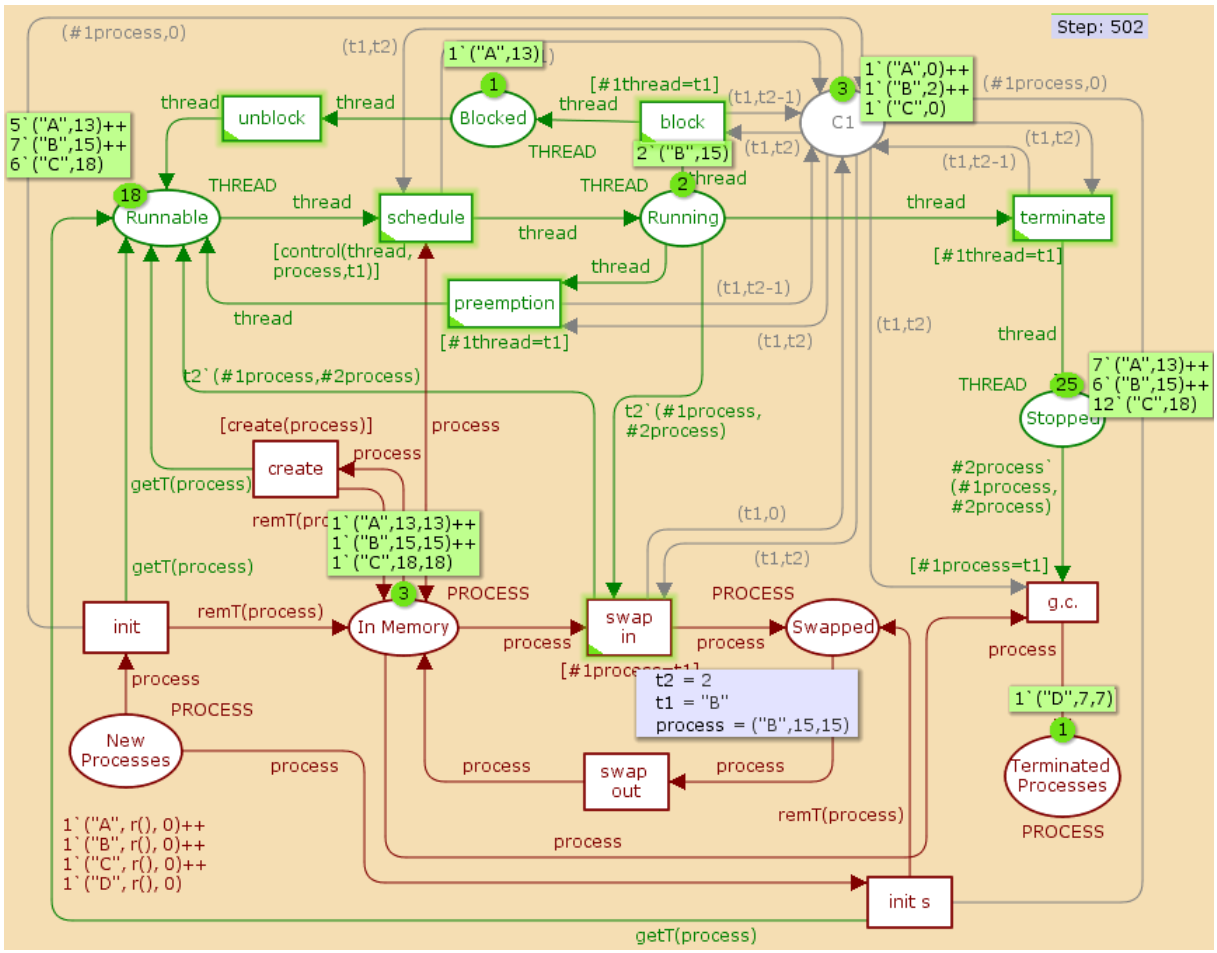
Součástí realizace prvních tří přechodů („**block**“, „**terminate**“ a „**preemption**“) je dekrementace hodnoty (o 1) druhého parametru (CS **INT**) odpovídajícího tokenu v místě „**C1**“.

Při uskutečnění přechodu „**swap in**“ je však situace odlišná. Aplikací přechodu „**swap in**“ je proces přesunut z místa „**In Memory**“ do místa „**Swapped**“. Pokud se proces nenachází v operační paměti, nemohou běžet (nacházet se v místě „**Running**“) žádná jeho vlákna. Z tohoto důvodu je využit odpovídající token v místě „**C1**“, jehož druhý parametr určuje počet vláken daného procesu, která se nacházejí v místě „**Running**“. Součástí realizace přechodu „**swap in**“ je tedy přesun všech případných (<0,NUMBEROFTHREADS>) vláken z místa „**Running**“ do místa „**Runnable**“.

Aktivace přechodů „**unblock**“ a „**swap out**“ není omezena žádnou podmínkou. Přechod „**unblock**“ uvozuje odblokování vlákna (přesun z místa „**Blocked**“ do místa „**Runnable**“). Přechod „**swap in**“ přenáší procesy z místa „**Swapped**“ do místa „**In Memory**“.

Pokud se proces nachází v místě „**In Memory**“ a zároveň jsou všechna vlákna daného procesu ukončena (v místě „**Stopped**“), může být realizován přechod „**g.c.**“. Uskutečněním tohoto přechodu se proces přesune do místa „**Terminated Processes**“ a zároveň jsou odstraněna všechna vlákna z místa „**Stopped**“, která odpovídají danému procesu. Současně je vymazán záznam daného procesu z místa „**C1**“.

Obrázek 83 ilustruje příklad značení sítě (modelu) po 502 simulačních krocích.



Obrázek 83 - CPN - Vlákna v kontextu procesního modelu - Příklad. Zdroj vlastní

Z obrázku vyplývají následující skutečnosti:

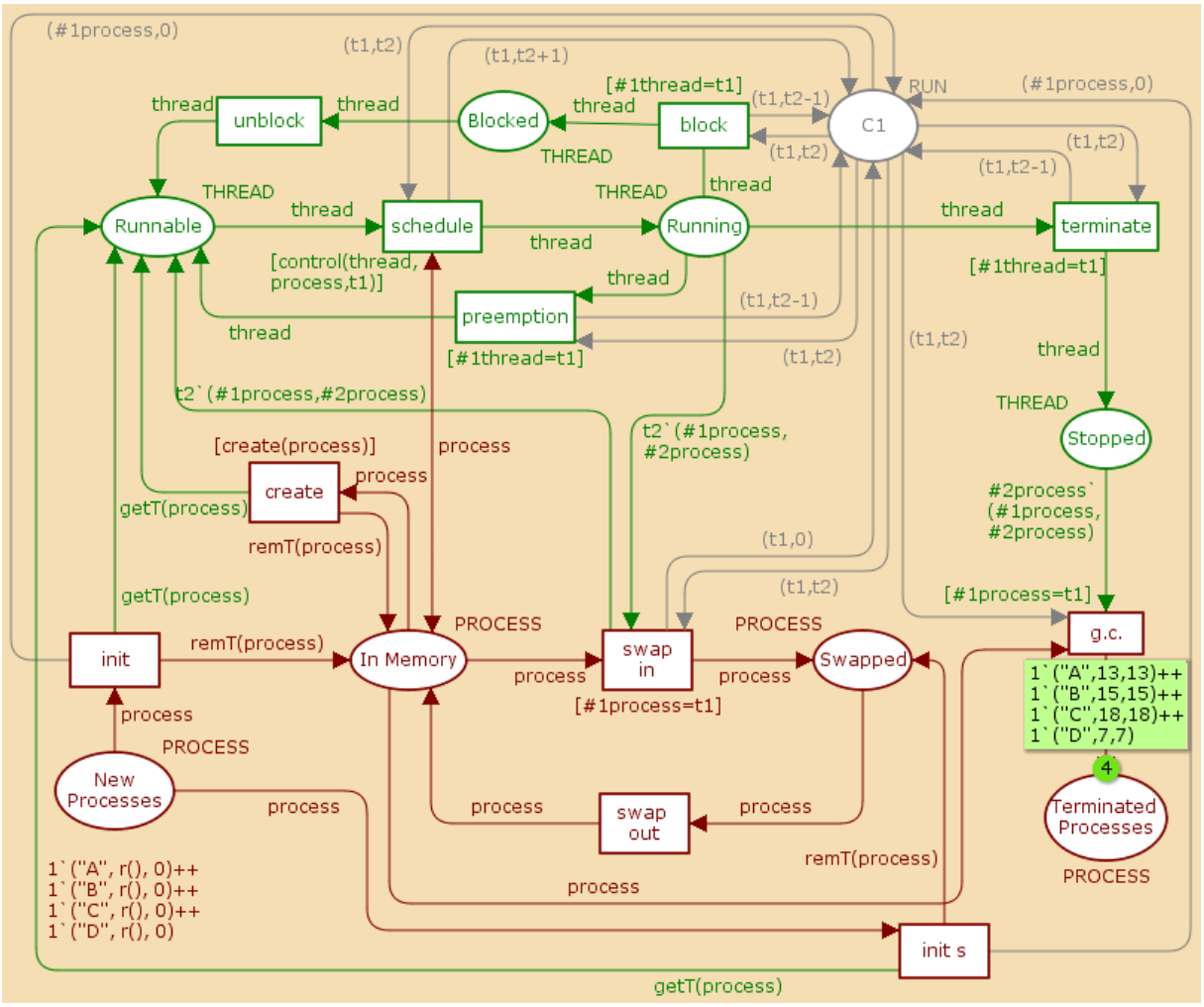
- proces **D** obsahoval **7** vláken, avšak nyní je již ukončen,
- proces **A** se nachází v operační paměti (místo „**In Memory**“), obsahuje **13** vláken, z nichž:
  - **5** se nachází v místě „**Runnable**“,

- **1** je zablokováno (v místě „**Blocked**“),
- **7** je již ukončeno (v místě „**Stopped**“).

- Proces **C** se též nachází v operační paměti („**In Memory**“), obsahuje **18** vláken, z nichž:
  - **6** se nachází v místě „**Runnable**“,
  - **12** je již ukončeno (v místě „**Stopped**“).
- Proces **B** je také v operační paměti („**In Memory**“) a obsahuje **15** vláken, z nichž:
  - **7** se nachází v místě „**Runnable**“,
  - **2** běží (v místě „**Running**“),
  - **6** je již ukončeno (v místě „**Stopped**“).

Z obrázku dále vyplývá, že proces **B** bude odložen na pevný disk („**Swapped**“). Součástí této akce („**swap in**“) bude přesun obou vláken daného procesu z místa „**Running**“ do místa „**Runnable**“.

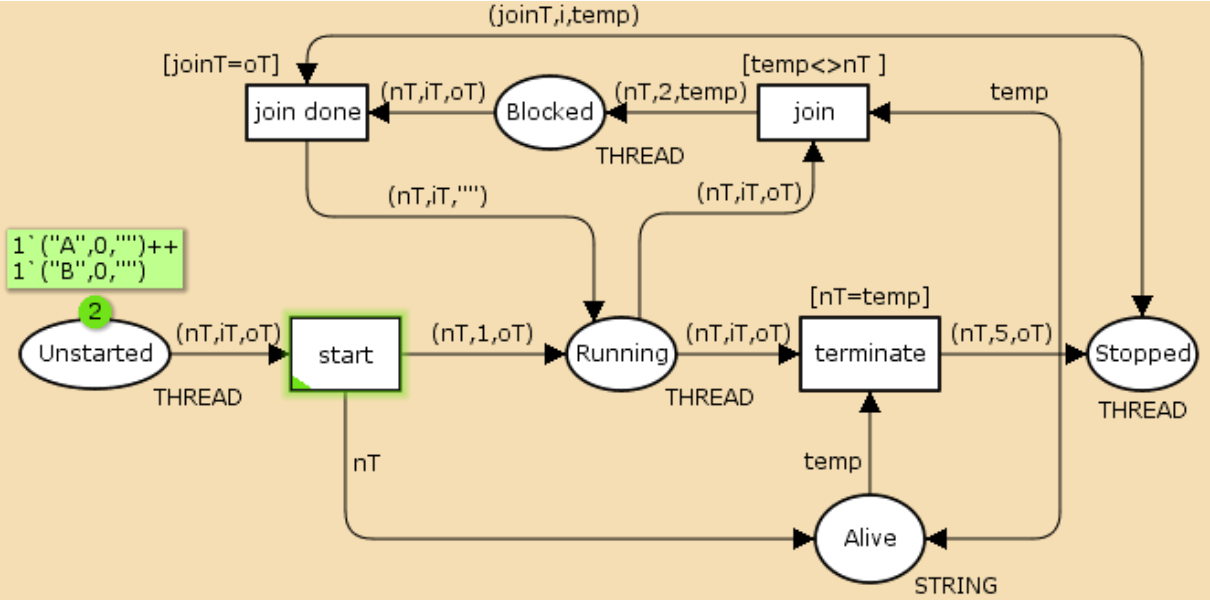
Cílový (požadovaný) stav sítě znázorňuje Obrázek 84.



Obrázek 84 - CPN - Vlákna v kontextu procesního modelu - Cílový stav. Zdroj vlastní

8 Verifikace navržených modelů

Pokud je model navržen, může být verifikován z hlediska předpokladů na něj kladených. V kapitole 6.1.1 byl ustanoven předpoklad, že vlákna na základě aplikace metody **join** mohou vytvořit cyklus, který způsobí **deadlock**. Zmíněný model zobrazuje Obrázek 85. Jedná se o ekvivalentní model z kapitoly 6.1.1. Z důvodu jednoduchosti jsou uvažována pouze **dvě** vlákna („**A**“ a „**B**“).



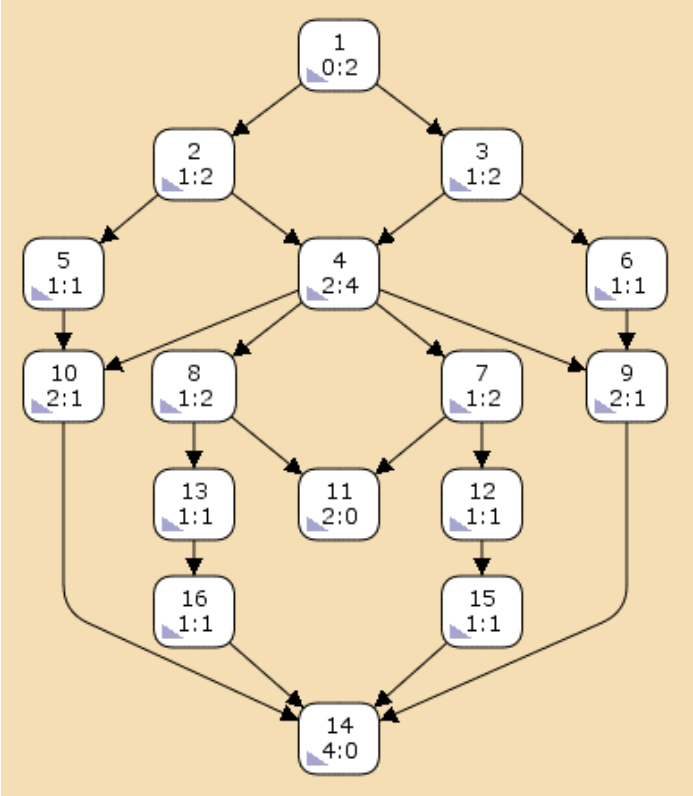
Obrázek 85 - CPN - Model metody join. Zdroj vlastní

8.1 Stavový prostor

Obrázek 86 znázorňuje vygenerovaný stavový graf výše zmíněného modelu. Ze stavového grafu (graf dosažitelnosti) vyplývá, že uzly č. **11** a č. **14** neobsahují výstupní hrany. Tyto uzly reprezentují mrtvé značení sítě. Zatímco uzel č. **14** představuje požadovaný stav, tj. všechna vlákna jsou v místě „**Stopped**“, uzel č. **11** znamená **deadlock**. S rostoucím počtem vláken by počet mrtvých značení narůstal exponenciální řadou, s ohledem na možné kombinace vláken v místě „**Blocked**“. Růst složitosti stavového prostoru s rostoucím počtem vláken znázorňuje Tabulka 8.

Tabulka 8 - Velikost stavového prostoru v závislosti na počtu vláken. Zdroj vlastní

Počet vláken	Počet stavů	Počet orientovaných hran	Počet mrtvých značení
1	3	2	1
2	16	22	2
3	128	306	12
4	1354	4956	120
5	17792	90580	1520



Obrázek 86 - Stavový graf modelu metody join. Zdroj vlastní

8.2 Vlastnosti modelu

Součástí generování stavového prostoru je souhrnná statistika obsahující všechny důležité informace o vlastnostech sítě (viz Obrázek 87). Z obrázku vyplývá, že stavový graf obsahuje **16** uzlů a **22** orientovaných hran. Dále jde o kompletní stavový prostor (**Full**) a každé místo obsahovalo maximálně 2 vlákna a minimálně žádné (**Boundedness Properties**). Posledním údajem je živost sítě (**Liveness Properties**), z které vyplývá, že síť obsahuje **2 mrtvá značení** (**11**,**14**).

```
Statistics
----------

  State Space
     Nodes:  16
     Arcs:   22
     Secs:   0
     Status: Full

  Scc Graph
     Nodes:  16
     Arcs:   22
     Secs:   0
```

```
Boundedness Properties
----------------------

  Best Integer Bounds
                                  Upper      Lower
     Threading'Alive 1            2          0
     Threading'Blocked 1          2          0
     Threading'Running 1          2          0
     Threading'Stopped 1          2          0
     Threading'Unstarted 1        2          0

Liveness Properties
-------------------

  Dead Markings
     [11,14]
```

Obrázek 87 - Statistika sítě. Zdroj vlastní

8.3 Verifikace ostatních modelů

Mimo výše zmíněný model metody **join** byly verifikovány všechny vytvořené modely. Tabulka 9 shrnuje výsledky verifikace jednotlivých modelů.

Tabulka 9 - Výsledky verifikace modelů. Zdroj vlastní

Model	Kapitola	Výsledek verifikace
Stavový model – vlákna na úrovni jádra	4.2.1	Jedno mrtvé značení **odpovídající cílovému stavu**
Stavový model – uživatelská vlákna	4.2.2	Více mrtvých značení (ukončení vlákna na úrovni jádra zachová původní stavy jeho uživatelských vláken) **odpovídacích požadovanému stavu** (všechna vlákna na úrovni jádra byla ukončena)
First-Come First-Served	5.1	**Jedno cílové** značení s různými časy (více mrtvých značení ve stavovém prostoru z důvodu využití časovaných Petri sítí)
Shortest Job First	5.2	**Jedno cílové** značení s různými časy (více mrtvých značení ve stavovém prostoru z důvodu využití časovaných Petri sítí)
Round Robin	5.3	**Jedno cílové** značení s různými časy (více mrtvých značení ve stavovém prostoru z důvodu využití časovaných Petri sítí)
Prioritní plánování	5.4	**Jedno cílové** značení s různými časy (více mrtvých značení ve stavovém prostoru z důvodu využití časovaných Petri sítí)
Metoda join	6.1.1	**Více** mrtvých značení (viz **výše**)
Metoda sleep	6.1.2	Jedno mrtvé značení **odpovídající cílovému stavu** (při neuvažování metody join)
Zamykání (metody enter a exit)	6.2	Jedno mrtvé značení **odpovídající cílovému stavu** (při neuvažování metody join)
Signalizace (metody pulse a wait)	6.2	Nekonečný stavový prostor, **nelze** určit
Vlákna v kontextu procesního modelu	7	Jedno mrtvé značení **odpovídající cílovému stavu**

Z výše uvedené tabulky vyplývá, že pouze model metody **join** vykazuje přítomnost **uváznutí** (**deadlock**). Speciálním případem je model využitý v kapitole 6.2 věnované signalizaci. Jak bylo specifikováno v dané kapitole, tento model neobsahuje ukončovací podmínku, a proto není možné tento model verifikovat (obsahuje nekonečný stavový prostor). Modely algoritmů „**Round Robin**“ a „**Prioritní plánování**“ obsahují také nekonečný stavový prostor. Tento fakt je způsobený přítomností místa „**Tick**“, které neustále inkrementuje hodnotu simulačního času o 1. Z tohoto důvodu obsahuje stavový graf jednu nekonečnou větev, která však není z důvodu verifikace důležitá. Při dostatečně velkém stavovém prostoru (částečném) vykazují tyto modely stejné výsledky jako ostatní plánovací algoritmy, tj. **jedno cílové značení** s různými časy. Ostatní modely dle verifikačních výstupu **CPN Tools** obsahují plné (**Full**) stavové prostory a zároveň nevykazují přítomnost **uváznutí** (**deadlock**)

Verifikační výstupy CPN Tools pro jednotlivé modely jsou obsaženy v příloze č. 14.

Závěr

Vlákna v moderních víceprocesorových operačních systémech hrají významnou roli při využívání různých paralelních technik (algoritmů) pro zvýšení výkonnosti zpracování složitých **vědecko-výzkumných** výpočtů. Mimo tuto oblast je využívání vláken nedílnou součástí všech „**user-friendly**“ uživatelských aplikací, u kterých je kladen velký důraz na jejich interaktivnost. Jelikož se vlákna vyznačují svým prioritním zaměřením na paralelní zpracování, musel být vybrán vhodný nástroj pro jejich modelování. Z důvodu vysoké složitosti všech interakcí, které vlákna mohou vykonat, byl zvolen nástroj **CPN Tools**, který pracuje s **barvenými Petri** sítěmi.

Jednotlivé činnosti vláken byly rozděleny na tři základní pohledy, které komplexně popisují odlišné úhly nahlížení na vlákna při různých úrovních abstrakce a požadavků z hlediska logiky. Prvním takovýmto pohledem je stavový pohled, který nahlíží na jednotlivá vlákna z hlediska jejich stavů a přechodů mezi nimi. V souvislosti s různými přechody mezi stavy vláken, byly definovány jednotlivé příčiny, které mohou vést k těmto změnám. Jedna z nejdůležitějších příčin změn stavů vláken je rozhodování systémového plánovače (**scheduler**), který přiděluje jednotlivá vlákna volným procesorům/jádrům. V souvislosti s touto problematikou byla namodelována činnost dvou preemptivních a dvou nepreemptivních algoritmů, která vysvětluje základní předpoklady na **spravedlivou** činnost vláken (snaha o rovnocennou šanci všech vláken při přidělování procesoru/jádra). Další pohled na činnosti vláken byl spojen s jejich chováním (behaviorální pohled), resp. vzájemnou interakcí. V této souvislosti byl modelován princip základních synchronizačních nástrojů, které řeší problémy spojené se souběžnou činností vláken (problém kritické sekce, signalizace, uspávání apod.). Jednotlivé synchronizační nástroje zajišťují **bezpečnost** (transparentnost a konzistenci) mezivláknové komunikace. Posledním modelovaným pohledem na činnost vláken byl jejich kontext vůči objektům, s kterými vlákna interagují, tj. procesy. Začleněním procesů do standardního stavového modelu bylo umožněno propojit všechny části základního vybavení počítačů, tj. procesoru (vlákna), operační paměti (proces) a vstupně výstupních zařízení (proces).

Poslední část této práce byla věnována využití verifikačních nástrojů **CPN Tools** při ověřování správnosti navržených modelů. Na základě všech těchto skutečností je možné konstatovat, že cíl práce byl splněn.

Použitá literatura

[1] TANENBAUM, Andrew S. *Modern Operatin Systems : 3rd Edition*. New Jersey : Prentice Hall, 2008. 1104 s. ISBN 978-0-13-6006633-2.

[2] TANENBAUM, Andrew S. *Structured Computer Organization : Fourth Edition*. New Jersey : Prentice Hall, 1998. 669 s. ISBN 978-0130959904.

[3] SILBERSCHATZ, Abraham; GALVIN, Peter Baer; CAGNE, Greg. *Operating System Concepts : Seventh Edition*. [s.l.] : John Wiley & Sons, Inc., 2005. 921 s. ISBN 0-471-69466-5.

[4] TANENBAUM, Andrew S. *Operating Systems Design and Implementation : Third Edition*. New Jersey : Prentice Hall, 2006. 1080 s. ISBN 978-0-13-142938-3.

[5] STALLINGS, William . *Operating Systems : Internals and Design Principles*. 5th Edition. [s.l.] : Prentice Hall, 2004. 832 s. ISBN 978-0131479548.

[6] *Javamex* [online]. 2011 [cit. 2011-04-22]. How threads work. Dostupné z WWW: <http://www.javamex.com/tutorials/threads/how_threads_work.shtml>.

[7] JENSEN, Kurt. *Transactions on Petri Nets and Other Models of Concurrency III*. Berlin : Springer-Verlag, 2009. 288 s.

[8] JENSEN, Kurt; KRISTENSEN, Lars M. *Coloured Petri Nets : Modeling and Validation of Concurrent Systems*. Berlin : Springer, 2009. 384 s. ISBN 978-3-642-00283-0.

[9] ČEŠKA, Milan, et al. *Petriho sítě : Studijní opora* [online]. Brno : VUT v Brně, 2009 [cit. 2011-04-21]. Dostupné z WWW: <http://www.fit.vutbr.cz/study/courses/PES/public/Pomucky/PES_opora.pdf>.

[10] MARKL, Jaroslav. Petriho sítě [online]. Ostrava: VŠB - Technická univerzita Ostrava, 1998 [cit. 2010-06-24]. Dostupné z WWW: < http://www.cs.vsb.cz/markl/pn/> .

[11] DIAZ, Michel. *Petri nets :Fundamental Models, Verification and Applications*. Hoboken : John Wiley & Sons, Inc., 2009. 585 s. ISBN 978-1-84821-079-0.

[12] HAAS, Peter J. Stochastic Petri Nets. New York: Springer-Verlag New York, Inc., 2002. 529 s. ISBN 0-387-95445-7.

[13] MILNER, Robin, et al. *The Definition of Standard ML*. Massachusetts Institute of Technology : The MIT Press, 1997. 128 s. ISBN 0-262-63181-4.

[14] CPN Tools [online]. 2010 [cit. 2011-01-09]. Dostupné z WWW: <http://cpntools.org/>.

[15] JENSEN, Kurt; CHRISTENSEN, Søren; KRISTENSEN, Lars M. *CPN Tools State Space Manual* [online]. Aarhus : University of Aarhus, 2002 [cit. 2011-04-17]. Dostupné z WWW: <http://cpntools.org/_media/documentation/manual.pdf>.

[16] CORMEN, Thomas H., et al. *Introduction to Algorithms*. Second Edition. The Massachusetts Institute of Technology : The MIT Press, 2001. 984 s. ISBN 0-262-03293-7.

[17] BRESHEARS, Clay. The Art of Concurrency. 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., 2009. 303 s. ISBN [3] 978-0-596-52153-0.

[18] ALBAHARI, Joseph; ALBAHARI, Ben. C# 4.0 in a Nutshell : Fourth Edition. Sebastopol : O'Reilly Media, Inc., 2010. 1056 s. ISBN 978-0-596-80095-6.

Seznam zkratek

API	-	Application Programming Interface (Aplikační rozhraní pro programování)
CPN	-	Coloured Petri Nets (Barvené Petri Sítě)
CS	-	Colour Set (Barvová Množina)
FCFS	-	First-Come First-Served
GUI	-	Graphical User Interface (Grafické uživatelské rozhraní)
HCPN	-	Hierarchy Coloured Petri Nets (Hierarchické Barvené Petri Sítě)
IDE	-	Integrated Development Enviroment (Integrované Vývojové Prostředí)
OS	-	Operating System (Operační Systém)
RR	-	Round Robin
SJF	-	Shortest Job First