Semestrální práce z předmětu KIV/OS

Single-task výpočet

Tomáš Linhart

A22N0055P

Západočeská univerzita v Plzni
LS 2020 / 2021

Obsah

1. Zadání

- realizováno na platformě Raspberry Pi Zero
- vytvoříte právě jeden user-space task, který bude zaštiťovat níže popsaný výpočet
- task po spuštění vypíše na UART základní informace o práci (číslo zadání a název, jméno řešitele a os. číslo) a napíše, co očekává za vstup
- task čte z UARTu parametry a vstupní data v textové podobě, každé zadání je ukočeno zalomením řádku (dle platformy)
- jako první vstup bude celé číslo, které označuje časový rozestup t_delta hodnot časové řady v minutách - tedy vlastně po jakých číslech se krokuje na ose X
- jako další vstup bude celé číslo, které označuje predikční okénko t_pred - tedy parametr modelu, viz níže
- jako každý další vstup bude číslo s plovoucí desetinnou tečkou nebo příkaz "stop"nebo "parameters"
- task provede po každém přijatém čísle fitting modelu $y(t + t\_pred) = A * b(t) + B * b(t) * (b(t) - y(t)) + C$, kde $b(t)$ spočítáte jako: $b(t) = D/E * dy(t)/dt + 1/E * y(t)$
  - A,B,C,D,E jsou parametry modelu (modelů)
  - dy(t)/dt je v sice derivace, vzhledem k povaze hodnot stačí aproximovat diferencí dělenou číslem 1.0/(24.0*60.0*60,0), je-li t v sekundách
  - fakticky jde o zjednodušené modely pro predikci hladin glukózy v intersticiální tekutině (zde poslouží pouze jako aplikační výpočet pro demonstraci)
  - můžete použít metodu nejmenších čtverců, nebo za bonusové body implementovat jednoduchý genetický/evoluční algoritmus
    - chcete-li minimální přesnost G/EA, inicializujte až polovinu jeho populace s předchozími výsledky získanými pomocí nejmenších čtverců (proč, viz příští rok na KIV/PPR)
  - volitelně můžete na UART vypisovat průběh výpočtu

- jako odpověď na přijaté číslo task vypíše novou predikci $y(t + t\_pred)$ dle nově spočtených parametrů, popř. řetězec NaN, pokud ještě není možné predikovat
- UART terminál musí být během výpočtu rozumně responzivní
  - kdykoliv probíhá výpočet, musí ho být možné zastavit zadáním příkazu "stop"(bez uvozovek)
  - pokud je výpočet ukončen, predikce je spočtena dle starých parametrů
  - pokud použijete metodu, která by byla výrazně rychlá, uměle ji zpomalte a zpomalení vhodně zdokumentujte - abychom mohli testovat responzivitu UART kanálu během výpočtu
- příkaz "parameters"vypíše na UART hodnoty parametrů A až E v nějaké rozumné formě
- pokud zrovna neprobíhá výpočet, zařízení minimalizuje spotřebu el. energie např. instrukcí WFI
- správu paměti řešte dynamicky (halda s podporou v OS), task si vyžádá paměť při své inicializaci (během zpracování a výpočtu již nebudou probíhat alokace z haldy)
- vstupy jsou pochopitelně rozumně validovány, pokud je zadán nevalidní vstup, task uživatele patřičně upozorní

2. Analýza

Prvotní analýza směřovala na to, jaké součásti bude možné využít z dodaného základu systému RTOS (https://github.com/MartinUbl/KIV-RTOS/) a jaké součásti nebudou využity. Prozkoumáním kódu bylo zjištěno, že systém obsahuje už naimplementováno mnoho důležitých součástí systému - plánovač, správu paměti kernel haldy, implementaci ovladačů pro souborový systém a další. Na druhou stranu některé ovladače, jako například ovladač sběrnice I2C, jsou pro tuto práci nadbytečné, avšak pro možné budoucí využití jsou ponechány jako součást projektu.

V práci bude využito již naimplementovaného systémového procesu, který provede uspání systému pomocí instrukce *wfe*. Plánovač na tento task přepne ve chvíli, kdy bude uživatelský task blokovaný, to znamená že bude čekat na instrukce z UART sběrnice.

V kernelu je primárně nutné doimplementovat čtení pomocí UART sběrnice a správu haldy uživatelského procesu.

Čtení z UART sběrnice je implementováno jako obsluha přerušení. Během této obsluhy je přečtený vstupní znak z registru a uložen do kruhového zásobníku. Při volání funkce Read ovaldače UART sběrnice už je vrácena pouze hodnota z tohoto zásobníku a neprovádí se čtení z registru. Pro účely této práce je podporována pouze miniUART sběrnice, označena jako UART0.

Paměť na haldě procesu bude implementována pomocí funkce malloc, respektive sbrk v kernelu. Funkce malloc pomocí systémového přerušení zavolá funkci sbrk, která provádí alokaci stránek a jejich namapování k příslušnému procesu. Funkci malloc už je vrácena pouze virtuální adresa na začátek alokované paměti.

Pro předpovídání hodnot bude využito genetického algoritmu. Program si ukládá vstupní hodnoty zadané uživatelem. Při fittingu parametrů je vypočítána předpověď v aktuálním čase pomocí minulých vstupních hodnot. Tato předpovězená hodnota a reálná hodnota vložená uživatelem je porovnávána ve fitness funkci jednotlivých chromozomů. Model s nejlepší fitness funkcí je použit pro předpověď další hodnoty a uložen pro další iteraci. Prvotní model je vygenerován náhodně. V případě, že dojde k přesné shodě předpovězené a reálné hodnoty, je fitting model předčasně ukončen.

3. Popis implementace

Pro implementaci čtení ze sběrnice UART bylo nejdříve nutné připravit kruhový zásobník. Tato jedndoduchá třída ukládá znaky do bufferu o velikosti 128 znaků. V případě že dojde k přetečení kruhového zásobníku, tak se přepisují starší data. Užitečnou vlastností je funkce read_until, která vrátí část textu od začátku bufferu do specifikovaného znaku. Pokud takový znak nenajde, vrátí celý text zpět do zásobníku. Tato funkce je hlavně využita při čtení celé řádky v třídě Read_Utils.

Jak už bylo zmíněno v analýze, ukládání do kruhového zásobníku UART sběrnice je realizováno jako obsluha přerušení. Toto přerušení je nutné zapnout pomocí již implementovaného řadiče přerušení. Během obsluhy dojde k uložení znaku do kruhového zásobníku, ze kterého je možné hodnoty vybrat voláním standardní funkce Read(). Aby k samotnému ovladači mohly přistupovat i uživatelské procesy, je zpřístupněn pomocí ovladače souborového systému a je dostupný voláním standardní funkce *open("DEV:uart/0", NFile_Open_Mode::Read_Write);*. Aby uživatel viděl znaky, které zadává, tak je při přijetí nového znaku tento samý znak okamžitě vypsán zpět na UART sběrnici.

Funkce malloc využívá systémového volání, jehož obsluha je implementována v souboru *swi.h*. O samotné alokování paměti se stará funkce sbrk uvnitř kernelu. Ta si pro daný proces zjistí, kolik paměti z již je alokované (z předchozích volání) a případně alokuje další stránky danému procesu. V případě, že není třeba stránku alokovat, vrátí pouze adresu začátku paměti. V druhém případě provede alokaci dostatečného počtu stránek a ty pomocí MMU přiřadí do tabulky stránek daného procesu na virtuální adresu. Virtuální prostor procesu začíná na adrese 0x21000000. Aktuální verze správce uživatelské haldy nepodporuje uvolnění paměti.

Jak již bylo zmíněno, tak pomocná třída Read_Utils slouží pro čtení celé řádky ze vstupu z UARTu. Jelikož ale z UARTu lze pouze číst jednotlivé znaky, má tato třída vlastní kruhový zásobník, do kterého si ukládá dočasné přečtené znaky. Nad tímto kruhovým zásobníkem teprve může volat funkci $read\_until('\backslash n', buf)$, která nám vrátí celou řádku. Součástí funkce pro čtení řádky je parametr, který určuje, zda je volání blokující, či ne. Pokud je volání blokující, je celý proces uspán, dokud nepřijde na UART sběrnici nový znak. Poté se provede kontrola, zda už v zásobníku je celá řádka a pokud není, je proces opět uspán. Toto blokování umožní předání procesoru jiným procesům, případně lze celé zařízení uspat pro snížení spotřeby.

Samotný genetický algoritmus je implementovaný v uživatelské úloze. Program nejdříve načítá vstupy pro interval krokování a dobu předpovědi. Tyto hodnoty jsou validovány - v případě že uživatel zadá nekorektní data, je

vyzván, aby je zadal správně. Po zadání proces vypočítá nutný počet vstupů pro možnost trénovat genetický algoritmus pomocí vzorce $ceil(t\_pred/t\_delta)$. Proces poté ve smyčce čeká na zadání příkazu nebo číselné hodnoty. Při přijetí číslené hodnoty spustí trénování nového modelu a vypíše novou předpověď. Během trénování modelu je postupně vytvořeno 200 nových generací. Pokud fitness funkce některého z chromozomů genetického algoritmu je dostatečně malá, je trénování zastaveno a daný chromozom je považován za optimální. Uživatel je o nalezení optimálního chromozomu informován výpisem do konzole. Po vytvoření jedné generace je poté vždy provedeno neblokující čtení z UARTu a jsou zpracovány případné příchozí příkazy.

4. Řešené problémy

Během psaní práce se vyskytly zvláštní problémy, kdy například použití # include z jednoho souboru způsobilo data abort, avšak ten samý # include načtený v jiném souboru data abort nezpůsobil. Tento problém jsem řešil s cvičícím, který poté našel problém v nulování BSS sekce a dodal opravu.

Druhým problémem je, že při emulaci pomocí QEMU při zaslání nové řádky na UART přichází pouze znak $\backslash r$, avšak už nezasílá znak $\backslash n$. Tento problém se mi nepodařilo vyřešit, při čtení z kruhového zásobníku tedy nejdříve kontroluji zaslání znaku $\backslash n$ a pokud ten nedorazí, tak kontroluji zaslání znaku $\backslash r$, který poté nahradím znakem $\backslash n$.

Během testování v qemu bylo zjištěno, že výpočet nového modelu byl velmi rychlý a bylo téměř nemožné vyzkoušet zastavení výpočtu pomocí příkazu stop. Po vytvoření nové generace je přidán prázdný for cyklus, který výpočet zpomalí.

5. Testování

Testování probíhalo především lokálně za využití emulátoru qemu. Během testování byly zadávány validní i nevalidní vstupy a ověřováno, že vypočítaná předpověď se pohybuje v předvídatelných mezích a program pracuje podle zadání.

Během testování na reálném zařízení se bohužel program nepodařilo spustit. Program byl uložený na SD kartě spolu s dalšími nutnými soubory pro spuštění. Raspberry Pi při startu probliklo, poté se ale bohužel na UART sběrnici nepřenášela žádná data. Měřením byla zjištěna téměř nulová spotřeba elektrické energie, což pravděpodobně znamená uspání RPi.

6. Závěr

Výsledkem je upravený vzorový systém KIV/RTOS poskytnutý cvičícím, který plně podporuje přenos dat pomocí UART sběrnice a výpočet předpovědi hodnoty hladiny cukru pomocí genetického algoritmu. Výpočet probíhá v samostatném procesu(tasku), který běží v uživatelském režimu. Přenos dat mezi kernelem a uživatelským procesem probíhá pomocí standardních volání. Součástí práce je též možnost alokace paměti na haldě pro každý proces.

Práci se bohužel nepodařilo zprovoznit na reálném zařízení.