KIV/OS Single task výpočet

Jiří Trefil A22N0060P

Úvod do problematiky

Problém spočíval ve vytvoření user-space tasku, který predikuje hladinu glukózy v krvi v čase. Samotný program musel být realizován na platformě Raspberry Pi Zero. Požadavky na task byly následující - uživatel přes uart zadává parametry pro predikční model. První dva parametry od uživatele, t_delta a t_pred určují časový skok "naměřených" respektive predikční okénko.

Například pro t_delta = 5 je časový rozdíl měřených hodnot 5 vteřin. Pro predikční okénko například 15 nás zajímají hladina glukózy v krvi v čase t_delta+t_pred, tedy v čase t=20. Tato schopnost predikce je velice užitečná, může totiž osobu informovat o tom, jestli neupadne do šoku, pokud spaluje příliš energie, nebo se pomocí této predikce dá řídit další zařízení.

Po zadání dvou časových údajů může uživatel přes uart zadávat číselné hodnoty či příkazy. Číselné hodnoty simulují naměřenou hodnoty v čase t, pro které se vypočítá predikce v čase t+t_delta (pokud je to možné, viz následující kapitola o analýze problému). Příkazy, kterými může uživatel kontrolovat běh modelu jsou následující : stop,parameters. Stop jednodušše zastaví aktuální výpočet a bude čekat na další vstup od uživatele. Příkaz parameters vypíše parametry modelu, viz kapitola o analýze problému.

Toto je v podstatě celý problém, který se snažím vyřešit – komunikace přes UART s uživatel a predikce hodnot.

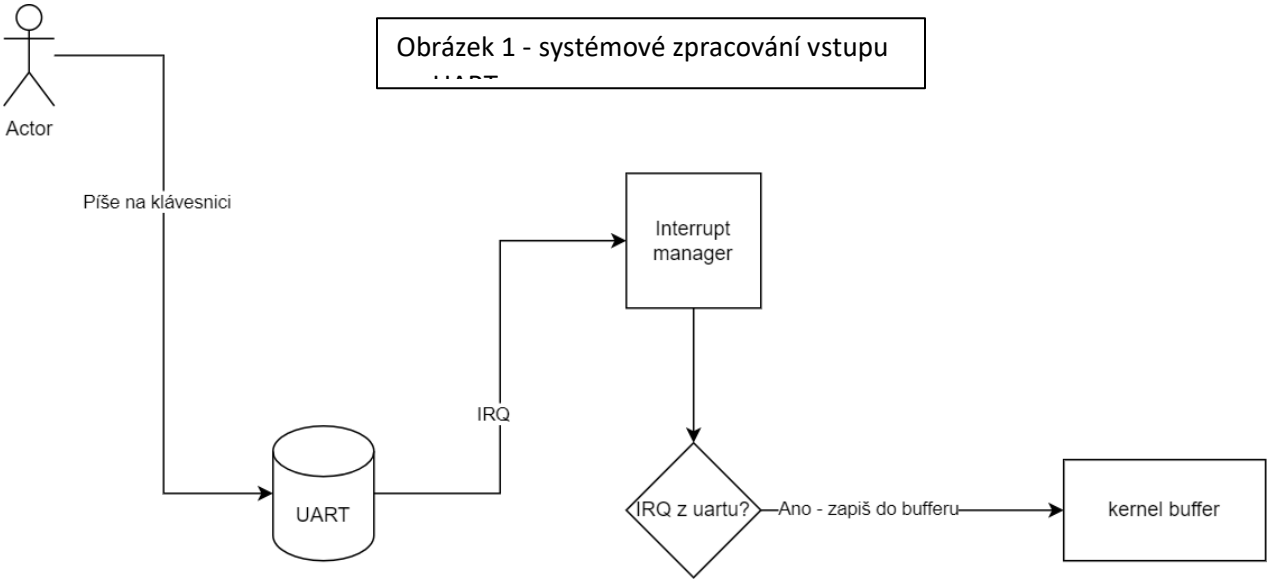
Analýza problému

Samotný problém vytvoření tasku jsem rozdělil na 3 větší části: **uart komunikce**, **alokace paměti tasku**, **vytvoření modelu**. První dva problémy v sobě samozřejmě také obsahující problém komunikace tasku a kernelu respektive za pomocí systémových volání žádat službu jádra, v mém případě čtení/zápis z/na UART kanál a alokace haldy pro task.

UART komunikace

Komunikace přes samotný UART(respektive mini-UART) spočívala ve vytvoření ovladače pro toto zařízení. Následně "namountovat" periferii do filesystému pro snažší ovládání (ve smyslu vytvoření jakéhosi api pro komunikaci s periferiemi pro tasky, které umožňuje snadné čtení/zápis/zavření/otevření zařízení z hlediska volání těchto služeb). Následovně je nezbytné povolit IRQ přerušení z UARTU, aby bylo vůbec možné zpracovat vstup od uživatele. Poslední krok je vytvoření cyklického bufferu v jádru systému, do kterého se budou zapisovat znaky z UART kanálu, tedy vstup uživatele. Samotný buffer zařízení je schopen udržet pouze 1 byte, tedy není možné jej dost dobře použít - docházelo by ke ztrátě dat. Pro shrnutí - je nutné naprogramovat ovladač zařízení. Samotné zařízení povolit, nastavit příslušné registry pro čtení/zápis, zapnout přerušení ze zařízení a vytvořit cyklický buffer, do kterého se bude za

zařízení zapisovat, aby nedocházelo ke ztrátě dat. Na obrázku 1 je vyzobrazeno zpracování uživatelského vstupu z UARTu a zapsání do bufferu.



Obrázek 1 - systémové zpracování vstupu

Alokace paměti pro task

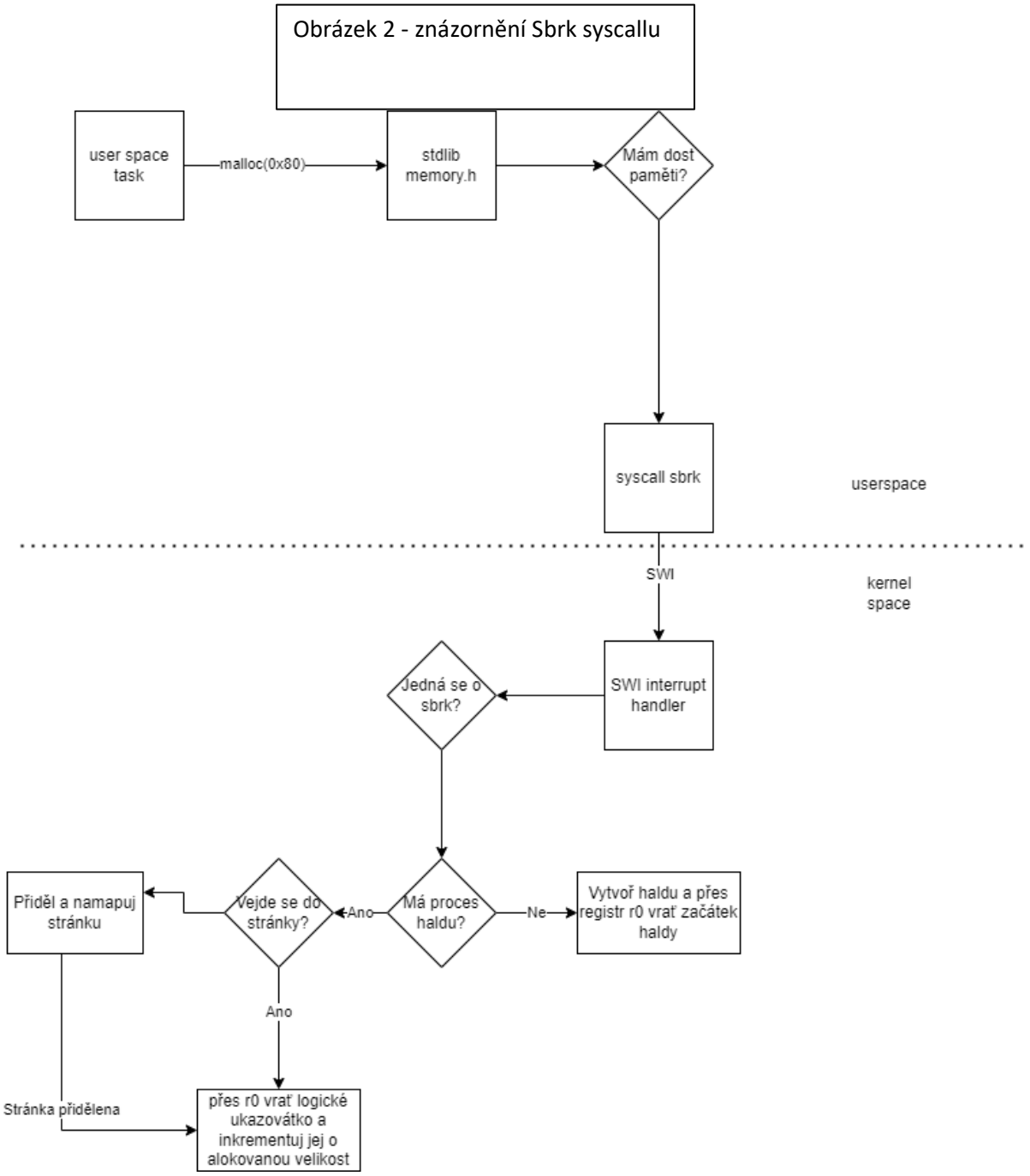
Uživatelský task nezbyteně potřebuje nějaké místo - paměť - kde může alokovat struktury, případně cokoliv, co potřebuje. Task nutně nepotřebuje používat haldu, přidělit tedy každému tasku nějaké místo, označit jej za haldu a zabrat tím hned stránku paměti není tedy chtěné řešení. Je tedy nezbytně nutné implementovat jakýsi ekvivalent sbrk systémového volání pro linuxové prostředí. Myšlenka je jednoduchá - proces zavolá stdlib funkci malloc respektive new, obojí možné a obojí vlastně pouze vrací adresu volné paměti, kterou má proces přidělenou. Správce paměti procesu v stdlibu se podívá, zda má volnou paměť, pokud ji nemá volnou, musí jádro systému požádat o další. Samotný správce vždy žádá víc, než nutně potřebuji, tedy požádá si o “velký” koláč a něj postupně ukrajuje, v mé implementaci žádá o 0x21000 B. Důvod je prostý, systémové volání je drahé - musíme přepnout kontext, uložit registry na zásobník, provést obsluhu, obnovit registry, přepnout zase zpátky kontext a skočit na návratovou adresu, což je drahá operace, toto chceme minimalizovat.

Pokud jádru přijde systémové volání Sbrk, pak můžou nastat v principu 2 scénáře.

1. scénář - proces ještě nemá místo na haldě => připrav procesu místo na haldě (halda procesu začíná od virtuální adresy 0x20000 v mé implementaci). Alokuj stránku, velikost stránky je 1 MB, a namapuj tuto stránku (protože stránka fyzicky v paměti může být kdekoliv) na adresu 0x20000. Následovně procesu vrátíme začátek jeho haldy, tedy adresu 0x20000.
2. scénář - proces již haldu přidělenou má a žádá o další paměť. Zde se jádro podívá, zda má proces pořád místo v již přidělené paměti (stránce). Tato operace spočívá v porovnání logického ukazovátka a fyzického ukazovátka. Logické ukazovátko je jakýsi relativní off set od začátku haldy až na konec již obsazené paměti. Fyzické ukazovátko je zarážka, která hraničí konec adresovatelné paměti procesem. Tedy pokud nastane situace, že by logické ukazovátko “vyskočilo” za fyzické ukazovátko, musíme alokovat další stránku a namapovat ji **souvisle** za již alokované stránky. Pokud tato situace nenastane, tedy logické ukazovátko “nevyskočí” za fyzické

ukazovátko, vrátíme adresu, na kterou právě ukazuje a pouze jej posuneme o velikost, která je alokována.

Syscall je znázorněn na obrázku 2.

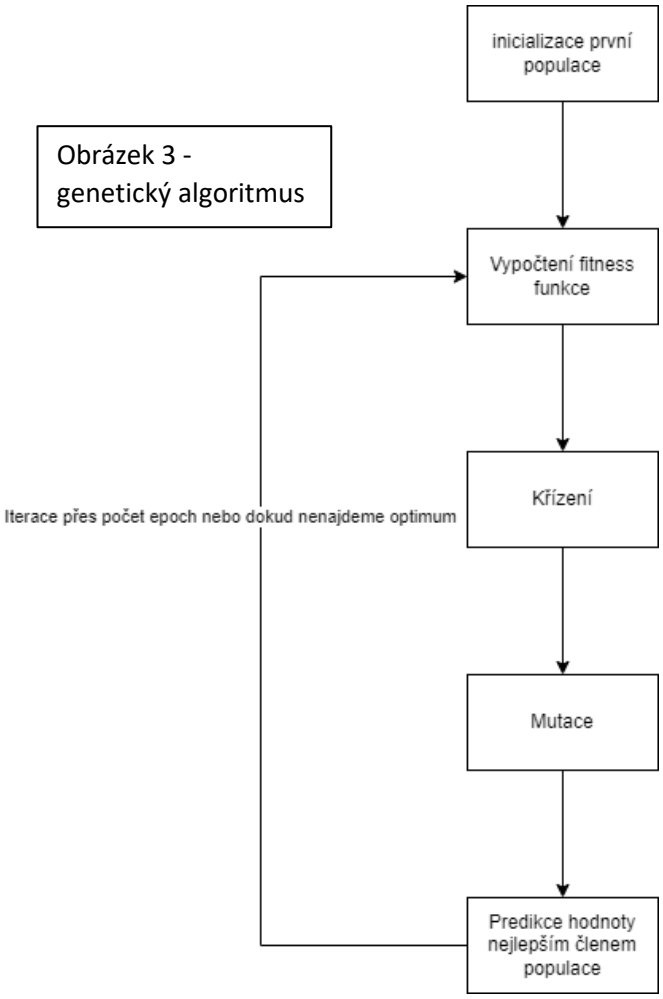


Vytvoření modelu

Samotný model pro predikci glukózy v krvi, tedy y(t+t_delta) je možné implementovat vícerozpůsoby. Například metodou nejmenších čtverců nebo mnou zvoleným genetickým algoritmem, primárně z

důvodé mé matematické inkompetence a taky trochu z důvodu toho, že jsou genetické algoritmy cool. Myšlenka genetického algoritmu je zobrazena na obrázku 3. Největší boj byla zvolení nějaké rozumné fitness funkce, která určuje "správnost řešení". Prvotní myšlenka byla vypočítat chybu člena populace následovně: $fitness = \left| \frac{1}{m} \sum \left(y - y_{predicted} \right) \right|$ kde m je počet vzorků, přes které byla chyba počítána. V podstatě je to aritmetický průměr rozdílu hodnot. Tato funkce vlastně fungovala, nýbrž zde může dojít ke kompenzaci chyby, což není chtěné a také může fitness hodnota být záporná a to je také nechtěné. Při úpravě fitness funkce do následující podoby, vlastně chyba metody nejmenších čtverců,

$fitness = \frac{1}{m} \sum \left(y - y_{predicted} \right)^2$ se chyby nekompenzují a vždy budou kladné, ušetříme si tedy absolutní hodnotu.



Obrázek 3 - genetický algoritmus

Implementace

Implementaci také rozdělním do tří částí, každá bude adresovat jiný problém, který byl zmíněn v analýze.

Implementace ovladače a bufferů

Ovladač ve své podstatě obaluje registry zařízení a poskytuje api pro komunikaci se zařízením. Ovladač UARTu je zde vlastně softwarová vrstva nad registry, která sdružuje práci s nimi do funkcí, které "vystrkuje" ven a které může jádro operačního systému volat. Například Read, Write.

Ovladač poskytuje 4 hlavní důležité funkce. Open, Close, Read, Write. Z názvu je asi jasné, co která dělá, přiblížím jejich implementaci, která je ve své podstatě pouze nastavování bitů registrů.

Funkce Open tedy otevírá UART - samotné spočívá v nastavení bitů vícero registrů. Nejdříve je nutné rezervovat si GPIO piny 14, 15 (GPIO – general purpose input output registry). Tyto piny slouží pro read/write. Následně je nutné nastavit samotné registry mini UARTu, konkrétně registr IIR, který umožní uartu generovat IRQ přerušení (v BCM manuálu jsou registry prohozeny, tedy IER je vlastně IIR a obráceně, viz errata). Následně nastavíme control registr (CNTL) na hodnotu 3 (0b101), tedy Read/Write. Posléze nastavím GPIO piny 14,15 na alternativní funkci 5, což odpovídá UART kanálu. Pokud všechny tyto operace proběhnout bez problému, UART kanál je aktivní.

Ovladač tedy slouží primárně na čtení/zápis z uartu, nečekaně, čtení z uartu userspace taskem je vlastně implementováno "přes ruku". V jádru existuje cyklický buffer velikosti 128 byteů, do kterého se zapisují znaky z UARTu. Userspace task potom čte z tohoto bufferu. Samotný UART udrží pouze 1 byte, tedy nejde z něj samotného dost dobře číst (neblokujícím způsobem, docházelo by ke ztrátě dat). Pro samotné psaní z tasku na UART již buffer potřeba není (bylo by to ale asi čistší řešení).

Po implementaci ovladače a kernelové bufferu bylo možné implementovat buffer na uživatelské straně. Jakožto jádro nechceme implementovat věci jako read_line či podobné, to je o úroveň abstrakce výše. V jádru chceme pouze jednoduché operace - přečti char, překopíruj někam chary. Operace jako read_line tedy necháme na userspace bufferu. Tato implementace UART bufferu zašťituje dva buffery. Jeden buffer se podává jádru se systémovým voláním a jádro do něj zapisuje charaktery. Do druhého bufferu se tyto přečtené znaky vkládají a je možné nad ním dělat operace. Jeden tedy slouží jako "přepravka dat" z kernelu do userspaceu a druhý už slouží jako opravdový buffer.

Jak je v zadání avizováno, uživatelský vstup končí cr+lf, v případě qemu pouze cr. Buffer umožnuje čekat na vstup, tedy blokující čtení řádky aktivním čekáním a neblokující čtení řádky. Aktivní čekání vlastně pouze hledá řádku v bufferu, tedy sekvenci byteů ukončenou znakem \r. Pokud ji nenajde, tak zavolá instrukci wfi, která raspi uspí - redukuje spotřebu energie. Super na WFI je ten fakt, že IRQ signál zařízení vzbudí, takže víme, že se můžeme znovu podívat, protože bylo něco napsáno v UARTu (jiné IRQ v rámci OS neběhá). Blokující čtení umožňuje předání očekávaného typu vstupu – int, string, float. Pokud by uživatel zadal vstup, který by nebyl očekávaným typem, task bude na vstup daného do bufferu stále čekat.

Implementace alokátoru paměti

Správce paměti pro proces je implementován v stdlibu. Samotná třída (Heap_Manager v mé implementaci) zakrývá systémová volání sbrk a free (není implementována obsluha, ale samotné volání implementováno je). Třída si také interně uchovává ukazovátka, podobně jako to dělá jádro u procesu. Tedy kolik má maximálně volné paměti (physical break ekvivalent), kde aktuálně v paměti stojí (logical break ekvivalent) a následně konstantu – 0x21000 – kolik paměti si vyžádá na jedno sbrk zavolání. Z této paměti potom postupně "ukrajuje", aby nemuselo pořád volat jádro systému (přepínání kontextu je drahé). Tato samotná třída je odstíněna a zabalena do hlavičkového souboru v stdlibu – memory.h -

který přepisuje operátor new, respektive new[] a vrací tak dostupnou paměť. Hlavičkový soubor také implementuje funkci malloc známnou z jazyka C – v podstatě funkce dělá to stejné, co přepsaný operátor new - vrátí adresu volného místa.

Implementace modelu

Vytvoření genetického algoritmu už vlastně byla ta “”snadná”” část problému. Algoritmus je implementován klasickým učebnicovým předpisem. Vznikne náhodná první generace – tato generace učiní predikce a ohodnotí se jejich správnost, tedy zavolá se fitness funkce pro každého člena generace. Následně se členové populace seřadí (klasickým quicksort algoritmem). Nejlepší z populace (tedy ten s nejnižší fitness hodnotou – fitness hodnota zde totiž označuje chybu predikce od reality, tedy čím nižší, tím vlastně lepší (možná trochu proti intuitivní)) se označí jako alpha – parametry tohoto jedince budou použity pro predikci hodnoty. Pak nastane fáze, kterou by František Fuka označil jako průměrný páteční večer, tedy genetická párty neboli křízení. Jako pravděpodobnost křízení - nebo lépe poměr křízení jsem stanovil 20%. Tedy smetánka populace (nejlepších 20% jedinců) se skříží a zplodí potomky. Zbylých 80% staré populace je vykoseno a nahrazeno nově vygenerovanými členy populace. Při křízení dvou rodičů se vygenerují dvě náhodná čísla, jedno v intervalu <0,4> a druhé v intervalu <0,1>. První číslo označuje, kolik parametrů se zdědí z rodiče (zbytek bude randomizován) a druhé číslo říká, od kterého rodiče budou parametry zděděny.

Poslední krok toho algoritmu je křížení. Zde pouze vygenerujeme náhodné číslo v intervalu <0,4> odpovídající parametrům A až E a parametr označený tímto číslem randomizujeme. Tímto chceme zajistit, aby generace byla vždycky dostatečně odlišná a nějaké lokální optimum tam nepřežívalo moc dlouho. Toto je v podstatě celý výpočet. Tyto kroky opakujeme do počtu epoch, což je nastavitelný parametr nebo dokud nenajdeme optimální řešení (tedy fitness alphy bude 0 - perfektní predikce dat).

Fitness funkce je vlastně řízena parametry od uživatele, t_delta a t_pred. Pro každou predikovanou hodnotu y(t+t_pred) musíme počkat, dokud uživatel neposkytne hodnotu v čase t, která odpovídá této hodnotě. Lidsky řečeno, pokud je t_delta = 5 a t_pred = 15, pak správnost predikce v čase 15 můžeme vypočítat až po 3. vstupu od uživatele (3*5 = 15), do té doby není možné cokoliv jakkoliv predikovat, model tedy bude vracet NaN řetězec.

Responzivita UART kanálu

Kanál musí být responzivní i během výpočtu. Responzivitu jsem zařídil pomocí “checkpointování”, tedy po zpracování člena populace se dotáži, zda něco není na UARTu. Tímto je uživateli poskytnuta možnost úplně výpočet zastavit příkazem “stop”.

Výpočet je uměle zpomalen, aby bylo možné rozumně ozkoušet renspozivitu.

Řešené problémy

Asi největší problém, na který jsem narazil a musel jsem počkat, až jej vyřešení někdo lepší bylo nedeterministické chování userspace tasku, když nebyla vynulována bss sekce. Program potom "divně" padal. Další trochu větší problém byla implementace jakéhosi bufferu pro komunikaci s uartem rozumným způsobem. Tedy ve smyslu, že mě vlastně v tasku nezajímají jednotlivé chary, které uživatel píše, ale zajímá mě až ucelený vstup, tedy sekvence byteů ukončená \r (v qemu). Na to jsem potřeboval vytvořit userspace buffer, který jádru opakovaně tento buffer "podává" a chce od něj, aby jej naplnilo znaky z uartu. V tomto bufferu potom musím vyhledat sekvenci byteů ukončenou \r, až toto můžu prohlásit za uživatelský vstup a předat jej dále na zpracování.

Největší problém mi tvořilo vytvoření userspace správce haldy, přitom je to cca 50 řádků kódu (primárně kvůli tomu, že používám inkrementální alokaci). Problémem tedy bylo kolem toho korektně "obmotat" hlavu.

Problém, který je možná vázaný na qemu a nevyřešil jsem jej je "korektní" čtení z uartu při IRQ přerušení. V normálním případě (když se aktivně čeká, jestli na uartu něco není podle podle LSB registru LSR) se nad uartem dá čekat. V případě IRQ ale tento bit bohužel není nastaven, takže dost dobře nevím, že mi vlastně přišlo něco z uartu, protože ten bit tam není nastavený. Jediné možné řešení zpracování IRQ z uartu je potom pouze prostě přečíst ten IO registr. Přečtení z IO registru také nastaví přerušení za obsloužené, což je trochu zvláštní. To považuji za buďto velký deficit qemu nebo jsem špatně pochopil, jak má fungovat uart IRQ.

Samotné vytvoření modelu bylo vlastně relativně ok, jediné, co by chtělo silně vylepšit je můj generátor náhody, který má tu pověstnou entropii domova důchodců. Asi by se vlastně dalo říct, že celá semestrální práce pro mě byla dost těžká na uchopení, jak myšlenkově, tak technicky, ale oštítkoval bych ji jako velice přínosnou.

Testování

Testoval jsem to pouze na qemu emulátoru. Testováním jsem primárně ozkoušel různé nevalidní vstupy a validní vstupy, tedy zda se program prostě nesesype na špatný vstup. Dal jsem vlastně pouze otestoval korektní funkčnost při korektních vstupech a přidal umělé zpomalení výpočtu pro otestování responzivity uart kanálu na příkazy od uživatele.

Při rozbíhání systému jsem nenarazil na větší problémy. Velký problém jsem tedy objevil v rozběhnutí gdb pro debuggování, to se mi nakonec podařilo, ale úplně jsem nebyl moc moudrý z toho, jak tento nástroj dobře použít, tak jsem jej úplně opustil.

Závěr

Problém se mi snad dostatečně dobře podařilo vyřešit. Model je schopný do určitě míry predikovat úroveň glukózy v krvi z dostupných datových vzorků. Určitě by se task dal optimalizovat i vylepšit.

Systémový obal je snad dobře naprogramován a měl by správně obsluhovat systémové volání userspace tasků.

Obtížnost semestrální práce bych asi vyšvihl na úroveň 10/10, asi ne úplně z důvodu toho, že to bylo “hodně kódu na psání” nebo něco takového, ale jelikož je to úplně jiná příchuť programování než na kterou jsem osobně zvyklý, objevují se vlastně úplně jiné problémy, člověk musí jinak přemýšlet atp. Přesto mi implementování malinké podmnožiny operačního systému dost otevřelo oči, jednak jsem tedy zjistil, jak je to vlastně složité, druhak se ve vývoji os objevují jakési myšlenky~ paradigmata, které se mohou přenést i na vývoj “obyčejného” softwareu a použít je pro optimalizaci.