FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Operační systémy
Single task výpočet

KIV/OS

Pavel Třeštík
A22N0137P

4. prosince 2022

1 Úvod

Cílem práce je implementovat velmi jednoduchý operační systém (dále jen OS), na kterém poběží jeden uživatelský proces (dále jen task). OS má být cílen na platformu Raspberry Pi Zero. Task komunikuje pomocí rozhraní UART a jeho úkolem je predikovat hladinu glukózy v intersticiální tekutině. OS musí tasku zajistit přístup k UART, poskytnout paměť k provádění predikce a zajistit, že task nemůže zasahovat do paměti, která mu nepatří. Pokud task zrovna neprovádí výpočet, tak se procesor přepne do úsporného režimu.

2 Analýza

Pro predikci task využívá genetického algoritmu. Genetický algoritmus je ale téměř nemožné implementovat bez použití generátoru náhodných čísel. Ten je v algoritmu použit alespoň pro mutaci prvků, ale také je vhodné ho použít pro náhodnou inicializaci chromozomů (prvků algoritmu).

Pro task tedy musí být minimálně možné pracovat s UARTem, generovat náhodná čísla a přidělovat paměť, kterou si task alokuje při své inicializaci.

Raspberry Pi Zero (dále jen RPi) je osazeno System on a Chip[1] Broadcom BCM2835, který poskytuje práci s periferiemi, kterými je RPi osazeno. Mezi tyto periferie patří např. Serial Peripheral Interface (SPI), Pulse Width Modulation (PWM), časovače a další.

RPi nabízí plnohodnotný UART, ale i verzi miniUART skrze Auxilární koprocesor (AUX). UART je poskytnut na General Purpose Input Output (GPIO) rozhraní a proto je třeba implementovat ovladač pro alespoň tyto GPIO, UART a AUX.

Generátor náhodných čísel je možné implementovat pomocí PWM.

Pro ochranu zařízení, tedy aby uživatel neměl neomezenou kontrolu, CPU běžně poskytují různé režimy. RPi poskytuje 7 režimů, z nichž 6 je privilegovaný (např. System (SYS), IRQ, FIQ, SVC...) a 1 je neprivilegovaný (User (USR)). Tato práce má zajistit alespoň základní ochranu paměti a celkově zařízení, takže potřebujeme umět přepínat mezi USR režimem a privilegovanými režimy. V *nixových systémech je procesor v USR režimu a pokud potřebuje přístup do privilegovaného režimu, tak je typicky volána

[1] `https://en.wikipedia.org/wiki/System_on_a_chip`

knihovní funkce, která vyvolá přerušení, čímž přepne procesor do privilegovaného režimu. U *nixů je se vším navíc zacházeno jako se souborem, takže základní ovládání všech zdrojů je dáno jednoduchým rozhraním s funkcemi Open, Close, Read, Write. Pro jednoduchost implementace je tuto vhodný způsob jak v této práci řešit používání periferií a přepínání mezi privilegovanými a neprivilegovaným režimem.

3 Implementace

Práce je implementována nad poskytnutou kostrou KIV-RTOS[2]. V této kostře jsou již připraveny některé ovladače, souborový systém, přepínání procesů, plánovač a příkladové uživatelské procesy. Z připravených věcí tato práce ale nevyužije zdaleka všechny.

3.1 Kernel

Kostra již obsahuje ovladač pro UART, GPIO, AUX, filesystem (FS), procesy, generování náhodných čísel a obsluhy přerušení. Některé z těchto ovladačů bylo třeba rozšířit.

3.1.1 UART

Protože je využit miniUART, tak potřebujeme AUX ovladač a GPIO. Tyto dva ovladače už naštěstí není třeba rozšiřovat. Samotný UART ovladač ale bylo třeba rozšířit. Do ovladače byla dodělána funkce čtení UART zařízení pomocí přerušení, které je potřeba pro volání skrz FS.

Při vyvolání přerušení je obsah FIFO registru, který je poskytnut, přepsán do kruhového bufferu. Kruhový buffer je implementován tak, že do pole fixní délky zapisuje a při dosažení konce přetéká zpět na začátek. V bufferu jsou dva indexy - psací a zapisovací. Původní implementací bylo, že psací index nesmí „předehnat“ čtecí o celou délku pole, protože to by přepsalo dosud nepřečtená data. To ale bylo změněno a v odevzdané verzi je toto umožněno. Důvodem je, že pokud načítáme vstup po řádce, tak při zadání vstupu, který je delší než délka bufferu, by do pole nikdy nebyl vložen konec řádku a tím by nikdy nedošlo ke přečtení a „vyprázdnění“ bufferu.

[2] `https://github.com/MartinUbl/KIV-RTOS`

Přijaté znaky jsou také vypsány zpět na konzoli, aby uživatel viděl, co za vstup zadal.

3.1.2 Procesy a paměť

Při vytváření procesu se alokuje tabulka stránek a do této tabulky byly zavedeny dvě stránky pro kód procesu a jeho stack. Memory management unit (MMU) pracuje s touto tabulkou a převádí virtuální adresy na fyzické. Toto již bylo implementováno, ale byla potřeba to rozšířit.

Rozšířením je, že do struktury procesu byl předán pointer na tabulku stránek, aby se do ní mohly zavést nové stránky, které si alokuje uživatelský proces. Pokud se uživatelský proces pokusí zasáhnout do paměti, která mu nepatří, tak systém ukončí proces a končí v cyklu. Alokace paměti je tedy taková, že uživatelský proces volá `malloc`, který vyvolá přerušení s nově zavedeným kódem. Obsluha tohoto přerušení volá `sbrk`, což je funkce, která vrátí pointer na alokovanou paměť. Pokud se vyžádaná paměť vejde do již alokované paměti, tak `sbrk` nealokuje novou stránku. Pokud `sbrk` přiřazuje nové stránky, tak alokuje po jedné, protože používáme 1MB stránky, což je poměrně velký blok. Pokud je vyžádaná paměť větší než paměť stránky samotné, tak `sbrk` alokuje stránky, dokud nemá dostatek paměti.

V kostře je implementován jednoduchý plánovač procesů. Ačkoliv má na RPi běže pouze jediný už. task, tak je vhodné plánovač zachovat, protože plánovač přepíná procesor do úsporného režimu, pokud nemá co plánovat. Toto je přesně chování, které se pro tuto práci hodí. Implementaci plánovače nebylo třeba rozšiřovat. K využití přepnutí do úsporného režimu stačí zablokovat běžící task. Pokud probíhá výpočet, tak je UART rozhraní čteno bez blokování - tedy pokud na zařízení UART nic není, tak proces pokračuje ve své činnosti. Pokud ale task zrovna nepočítá, tak se UART čte blokujícím způsobem. Toho je v práci dosaženo voláním `wait()` nad UART rozhraním. `wait()` zařídí uspání procesu, který ho volá do doby, dokud z žádaného zařízení nepřijde interrupt, který skrz FS volá `notify()`.

3.2 Userspace task

Na začátku procesu je vypsána uvítací hláška, která sděluje informace o práci a instrukce k použití. Následně jsou načteny 2 celočíselné hodnoty (`t_delta` a `t_pred`) a po jejich načtení je alokována paměť potřebná k provádění predikcí užitím genetického algoritmu (GEA). Poté začne hlavní smyčka pro-

gramu, z které by task už nikdy neměl vyskočit. Již bylo zmíněno, že se predikuje hladina glukózy v intersticiální tekutině. Tato predikce je počítána modelem $y(t + t\_pred) = A * b(t) + B * b(t) * (b(t) - y(t)) + C$, kde $b(t)$ se počítá jako: $b(t) = D/E * dy(t)/dt + 1/E * y(t)$. Derivaci $dy(t)/dt$ v práci stačí aproximovat diferencí (t_delta) dělenou číslem $1.0/(24.0 * 60.0 * 60, 0)$, je-li t v sekundách.

Hlavní smyčka programu vypadá zhruba následovně. Nejdříve načítá vstup pomocí blokujícího čtení. Pokud je přijatý vstup číslo, tak začne výpočet. Výpočet provádí 100 generací, které výše popsanou rovnicí počítají fitness funkci jednotlivých chromozomů generace a tím ladí parametry `A-E`. Po ohodnocení generace se vytvoří nová generace ze stávající. Nejlepších 10% chromozomů je zachováno. Dalších přibližně 80% generace je zaplněno křížením chromozomů. Zbytek nové generace je doplněn nejlepšími chromozomy. 40% odzadu nové generace je mutováno, proto je doplnění generace prováděno nejlepšími chromozomy. Po ohodnocení generace a vytvoření nové generace se vracíme do hlavního cyklu, kde je tentokrát bez blokování přečteno rozhraní UART. Pokud je přijata řádka, je vstup parsován a na základě vstupu je provedena akce. Pokud přijde vstup „stop“ - tedy příkaz k přerušení výpočtu, tak se ukončí probíhající výpočet a vypočte se predikce použitím poslední generace výpočtu, který doběhl kompletně. Pokud přijde příkaz „stop“ kdykoliv, kdy se nepočítá, tak se pouze vypisuje chybové hlášení. Pokud na vstupu je číslo a výpočet již probíhá tak se také vypisuje chybové hlášení. Pokud je na vstupu řetězec „parameters“ tak se vypíší parametry momentálně počítané generace. Tento příkaz funguje kdykoliv v hlavním cyklu a teoreticky je i možné vypisovat parametry, jak se mění během výpočtu. Pokud na vstupu nic není, tak cyklus pokračuje počítáním další ze 100 generací, nebo pokud byla dopočítána 100tá generace, tak se přejde do stavu blokujícího čekání na vstup.

4 Řešené problémy

V práci bylo řešenou pouze několik menších problémů. Jeden takovýto problém byl již popsán v implementaci UART a tím je čtení řádky z kruhového bufferu, pokud bychom bufferu neumožňovaly přetéct o celou délku pole. Řešením bylo psát do bufferu, dokud není zapsán celý vstup bez ohledu na délku zapsaných dat. Pokud by buffer přetekl a nebo byla zahozena část zprávy, která by se do bufferu nevešla, už je celkem nepodstatné, protože v

obou případech je vstup neúplný.

Druhým problémem byly statické proměnné ve třídě `Chromosome`. Tyto proměnné v tasku nefungovali a při debugování bylo zjištěno, že program končí ve smyčce chyby práce s pamětí. Není známa příčina, ale tento problém byl vyřešen předěláním statických proměnných na globální proměnné, pro které byl udělán namespace `Chromosome_NS`.

Třetím problémem bylo otevření generátoru náhodných čísel, který také končil ve smyčce chyby práce s pamětí. Důvodem bylo, že při inicializaci (otevření) generátoru se povolovalo přerušení pro něj, ale alespoň v qemu bylo zjištěno, že adresa povolení přerušení je nepřístupná. U tohoto problému nebylo vyřešeno proč je adresa nepřístupná, ale pro tuto práci generátor funguje i bez přerušení. Přerušení generátoru se nepovolují.

5 Testování

Práce byla vyvíjena na emulátoru `qemu` a debugována připojením debuggeru `gdb` na emulátor. Task byl testován pouze manuálně, ale díky jeho jednoduchosti by toto testování mělo být dostatečné. Funkčnost kernelu ve své podstatě testuje task jako takový, protože z tasku se volají kernelové funkce, které když nefungovali, tak byly debugovány a opraveny.

Byl i neúspěšný pokus práci zprovoznit na hardware. Problémem se zprovozněním na hardware je to, že se nepovedlo navázat UART komunikaci. Bylo ujištěno, že klient používá stejné nastavení, tedy baud rate 115200 a žádný paritní bit. Kvůli nedostatku času nebylo zprovoznění na hardware prioritou.

6 Závěr

Práce staví na kostře KIV-RTOS, kde již byla předpřipravená velká část kernelu a také zde byly příkladové userspace tasky. Do kernelu bylo doimplementováno čtení UART rozhraní a alokace paměti pro uživatelský proces. Následně byly vytvořeny obdoby standardních funkcí, které byly potřeba pro task. Mezi tyto funkce patří například `ftoa`, `fgets`, ale i třída `Random` pro generování náhodných čísel.

Poté byl naimplementován samotný task, který použitím genetického algoritmu ladí parametry predikčního modelu. Práce by měla splňovat všechny požadované body zadání.