Memory manager

Termín odevzdání:	**09.05.2021 23:59:59**
Hodnocení:	**30.0000**
Max. hodnocení:	**30.0000** (bez bonusů)
Odevzdaná řešení:	1 / 60
Nápovědy:	0 / 0

Úkolem je realizovat funkce a třídy implementující zjednodušenou správu virtuální paměti v OS.

Cílem úlohy je hlubší pochopení problematiky virtuální paměti a její správy v OS. Pro účely této úlohy však byla celá problematika drasticky zjednodušena. Celé programování probíhá na úrovni C a C++ a není potřeba pracovat s privilegovanými instrukcemi CPU (tedy jedná se o čistě user-space program, není potřeba znalost assembleru):

CPU
: bude implementováno jako C++ třída `CCPU`, respektive její podtřída. Simulovaný proces bude volat metody této třídy, metody zprostředkují čtení/zápis do paměti.

Proces
: bude simulován jako vlákno. Vlákna bude vytvářet Vaše implementace. Pro každý vytvořený "proces" zároveň vytvoříte instanci `CCPU`, instance bude překládat adresy pro tento "proces".

Adresy
: Náš user-space program samozřejmě nemůže změnit chování adres na reálném CPU. V simulovaných procesech proto budeme pracovat s adresami "uvnitř" simulovaného procesu, tyto adresy budeme předávat odpovídajícím metodám třídy `CCPU`, které adresy přeloží na "opravdové" adresy a zprostředkují zápis/čtení. Simulované adresy "uvnitř" simulovaného procesu budou tedy "virtuální" adresy, adresy v našem programu budou vlastně "fyzické".

Adresní prostor simulovaného procesu
: V user-space simulaci nelze snadno zařídit simulaci celého adresního prostoru simulovaných "procesů". Budeme simulovat pouze část adresního prostoru - haldu simulovaného procesu. Halda bude mít velmi primitivní strukturu - simulovaný proces si pomocí funkcí bude moci alokovat požadovanou velikost haldy (zadá požadovaný počet stránek), simulovaná halda bude adresovaná simulovanými virtuálními adresami v rozsahu 0 až `počet_stránek_haldy * 4096 - 1`.

Paměť
: Při inicializaci dostane funkce adresu počátku a velikost bloku paměti, se kterým hospodaří. Tato paměť má velikost řádově několik desítek MiB. To odpovídá situaci, kterou má na startu reálný OS. Z této paměti budete uspokojovat požadavky simulovaných procesů (do jejich adresního prostoru namapujete části této paměti). Kromě této paměti má Vaše implementace k dispozici ještě několik stovek KiB paměti dostupné přes `new/malloc` a zásobník (zásobníky vláken). Takto dostupná paměť je malá, rozhodně nestačí pro požadované alokace. Reálný OS samozřejmě možnost volat `new/malloc` nemá, v tomto ohledu se jedná o zjednodušení.

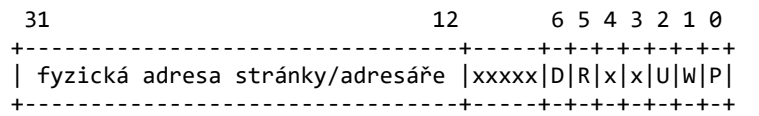
Spuštění simulace
: Simulace se spustí zavoláním Vaší funkce `MemMgr`, požadované rozhraní je v přiloženém souboru. Funkce dostane pomocí parametrů spravovaný blok paměti (ukazatel na jeho počátek a jeho velikost v počtiu stránek, každá stránka je veliká 4 KiB). Dále dostane ukazatel na funkci, tato funkce bude spuštěna jako první "proces" - obdoba procesu init. Funkce zinicializuje Vaše interní struktury pro správu paměti, vytvoří instanci simulovaného procesoru a spustí předanou funkci. Zatím ještě není potřeba vytvářet nová vlákna - init poběží v hlavním vláknu. Init samozřejmě začne vytvářet další simulované procesy, pro vytvoření je v rozhraní `CCPU` příslušné volání. Toto volání bude obslouženo Vaší implementací (vytvoření další instance `CCPU`, přidělení paměti, vytvoření vlákna, ...). Init samozřejmě někdy skončí (volání se vrátí z předané funkce), stejně tak skončí i ostatní procesy. Hlavní vlákno zpracovávající proces init počká na dokončení všech "procesů", uklidí Vámi alokované prostředky a vrátí se z `MemMgr`.

Počet simulovaných procesů
: Počet vytvářených procesů je omezen na `PROCESS_MAX=64`. Tedy kromě procesu init může najednou existovat nejvýše 63 dalších "procesů". Procesy se ale mohou ukončovat a další vznikat. Tedy celkem může vzniknout mnoho (neomezeně) procesů, ale v jeden okamžik jich poběží nejvýše 64. Toto je potřeba zohlednit při ukončování vláken, aby se systém nezahltil.

Vytváření nových procesů
: Nový "proces" se vytváří voláním metody `NewProcess`, tuto metodu budete implementovat v potomku třídy `CCPU`. Metoda dostává parametrem ukazatel na funkci, která bude spuštěna v novém "procesu" - vláknu. Dále je parametrem netypový ukazatel, který předáte jako inicializační parametr (stejně jak to dělá `pthread_create`. Posledním parametrem je příznak, zda nově vzniklý "proces" bude mít ve svém adresním prostoru zcela "prázdno" nebo zda získá kopii obsahu adresního prostoru svého rodiče (chování podobné funkci `fork`).

Základem řešení je třída `CCPU`. Tato třída zjednodušeně simuluje chování procesoru `i386` při překladu adres. Část metod je implementovaná (v testovacím prostředí a v přiloženém zdrojovém kódu). Vaším úkolem bude od této třídy odvodit potomka a v něm implementovat metody, které jsou v `CCPU` abstraktní. Rozhraní třídy je navrženo takto:

- Konstruktor `CCPU(memStart, pageTableRoot)`. Konstruktor inicializuje členské proměnné podle parametrů. Parametr `memStart` udává "opravdový" ukazatel na počátek bloku paměti, který byl simulaci předán při volání `MemMgr`. Druhým parametrem je rámec stránky, kde je umístěn adresář stránek nejvyšší úrovně (toto nastavení stránkování bude použito pro přepočet adres v tomto simulovaném CPU).
- Destruktor můžete v odvozené třídě implementovat, pokud budete potřebovat uvolňovat alokované prostředky.
- Metoda `GetMemLimit` zjistí, kolik stránek má alokovaných proces, pro který je používána tato instance `CCPU`. Tato metoda je abstraktní, její implementace v odvozené třídě je Váš úkol.
- Metoda `SetMemLimit` nastaví paměťový limit (v počtu stránek) pro tento proces. Metoda může být použita jak pro zvětšení, tak pro zmenšení paměťového prostoru simulovaného procesu. Návratovou hodnotou je `true` pro úspěch, `false` pro neúspěch (např. není dostatek paměti pro alokaci). Implementace je Váš úkol.
- Metoda `NewProcess` vytvoří nový simulovaný proces (vlákno). Parametrem volání je adresa funkce spouštěné v novém vlákně, její parametr a příznak `copyMem`. Význam prvních parametrů je zřejmý. Třetí parametr udává, zda má být nově vzniklému "procesu" vytvořen prázdný adresní prostor (hodnota `false`, `GetMemLimit` v novém procesu bude vracet 0) nebo zda má získat paměťový obsah jako kopii paměťového prostoru stávajícího procesu (true). Úspěch je signalizován návratovou hodnotou `true`, neúspěch `false`. Metodu budete implementovat v odvozené třídě.
- Metoda `ReadInt` přečte hodnotu typu `uint32_t` ze zadané simulované virtuální adresy. Návratovou hodnotou je hodnota `true` pro úspěch nebo hodnota `false` pro selhání. Selháním je např. pokus o čtení mimo hranice alokovaného adresního prostrou (reálný OS by v takové situaci vyvolal signál "Segmentation fault", simulace bude reagovat takto mírně). Metoda je kompletně implementovaná v dodané třídě, Vaše implementace ji nebude nijak měnit. Pro zjednodušení předpokládáme pouze zarovnaný přístup (zadaná virtuální adresa je násobek 4, tedy celé čtení se odehraje v jedné stránce).
- Metoda `WriteInt` zapíše hodnotu typu `int` na zadanou simulovanou virtuální adresu. Návratová hodnota je `true` pro úspěch nebo `false` pro neúspěch. Metoda je opět hotová v dodané třídě. Opět předpokládáme pouze virtuální adresy jako násobky 4, tedy opět celý zápis proběhne v jedné stránce.
- Metoda `virtual2physical` přepočítává simulovanou adresu ("virtuální" adresa v procesu) na adresu "fyzickou". Metoda je implementovaná v dodané třídě. Implementace odpovídá chování procesoru i386 pro základní variantu stránkování (2 úrovně adresářů stránek, 4KiB stránka, 1024 odkazů v adresáři stránek). Vaše implementace nebude tuto metodu nijak měnit. Budete ale muset odpovídajícím způsobem vyplnit adresáře stránek, aby je metoda správně zpracovala. V reálném OS je tato funkce "ukryta" v HW procesoru.
- Metoda `pageFaultHandler` je vyvolána, pokud při přepočtu adres dojde k chybě - výpadku stránky. Může se jednat o skutečnou chybu (neoprávněný přístup) nebo o záměr (odložení stránky na disk, implementace strategie copy-on-write). Metoda je vyvolána s parametry přepočítávané virtuální adresy a příznaku, zda se jedná o čtení nebo zápis. Návratovou hodnotou metody je `true` pokud se podařilo odstranit příčinu výpadku stránky (např. načtení stránky z disku) nebo `false` pro indikaci trvalého neúspěchu (přístup k paměti mimo alokovaný rozsah). Pokud je vráceno `true`, je přepočet adres zopakován, pro navrácenou hodnotu `false` je přepočet ukončen a programu je vrácen neúspěch (ReadInt/WriteInt vrátí `false`). Implicitní chování vrací vždy `false`, toto chování postačuje pro základní implementaci. Pokud se ale rozhodnete implementovat strategii copy-on-write, budete muset tuto metodu v podtřídě změnit. Pozor, pokud trvale vracíte `true` a neodstraníte příčinu výpadku, skončí program v nekonečné smyčce. Reálný HW signalizuje výpadek stránky nějakým přerušením, obsluha přerušení odpovídá této metodě. Pokud na reálném HW obsluha přerušení neodstraní příčinu výpadku a požaduje opakování přepočtu adres, může též dojít k zacyklení nebo k přerušení typu double-fault.
- Konstanta OFFSET_BITS udává počet bitů použitých pro adresaci uvnitř stránky (zde 12 bitů).
- Konstanta PAGE_SIZE udává velikost stránky v bajtech (4096 B).
- Konstanta PAGE_DIR_ENTRIES udává počet záznamů v adresáři stránek (zde 1024).
- Položka adresáře stránek má 4 bajty (32 bitů). Její struktura je:



  - Konstanta ADDR_MASK obsahuje masku, která přiložená k položce adresáře stránek zachová pouze adresu stránky/adresu podřízeného adresáře stránek.
  - Konstanta BIT_DIRTY je nastavena CPU v případě zápisu do stránky (v obrázku bit D).
  - Konstanta BIT_REFERENCED je nastavena CPU v případě čtení ze stránky (bit R v obrázku).
  - Konstanta BIT_USER určuje, zda ke stránce má přístup i uživatel (1) nebo pouze supervisor (0). Pro naší simulaci bude potřeba bit vždy nastavit na 1 (bit U v obrázku).
  - Konstanta BIT_WRITE určuje, zda lze do stránky zapisovat (1) nebo pouze číst (0), bit W v obrázku.
  - Konstanta BIT_PRESENT určuje, zda je stránka přítomná (1) nebo odložená na disk/nepřístupná (0), bit P v obrázku.
  - Zbývající bity nejsou v naší simulaci použité. Reálný procesor jimi řídí cache (write-through/write-back/cache-disable), execute-disable a další (i486+).

Odevzdávejte zdrojový kód s implementací funkce `MemMgr`, podtřídy `CCPU` a dalších podpůrných tříd/metod. Za základ řešení použijte zdrojový soubor `solution.cpp` z přiložené ukázky. Pokud zachováte bloky podmíněného překladu, můžete zdrojový soubor `solution.cpp` rovnou odevzdávat na Progtest.

Všimněte si, že pojmenování Vaší podtřídy `CCPU` není vůbec důležité. Instance této třídy vyrábí výhradně Vaše část implementace a testovací prostředí pracuje pouze s polymorfním rozhraním předka.

Nepokoušejte se pro datové stránky/adresáře stránek používat paměť mimo přidělený blok paměti předaný při volání `MemMgr`. Implementace `CCPU` v testovacím prostředí kontroluje, zda jsou použité adresy stránek/adresářů stránek z rozsahu spravované paměti. Pokud jsou mimo, řešení bude odmítnuto.

Základní verze programu musí umět spouštět procesy bez kopírování adresního prostoru (parametr `copyMem` bude v těchto testech vždy `false`). Takové řešení neprojde nepovinným a bonusovým testem, tedy bude hodnoceno méně body.

Řešení, které bude správně (ale neefektivně) zpracovávat parametr `copyMem`, projde i nepovinným testem a dostane nominální hodnocení 100% bodů. Pro zvládnutí posledního (bonusového) testu je potřeba správně a hlavně efektivně kopírovat obsah adresního prostoru volajícího do nově vzniklého procesu (je-li to požadováno). Prosté kopírování nestačí, nebude mít k dispozici dost paměti. Správné řešení bude muset použít techniku copy-on-write.

Pro testování využijte přiložený archiv s několika připravenými testy. Tyto testy (a některé další testy) jsou použité v testovacím prostředí. Dodané testy dále ukazují použití požadovaných funkcí/tříd.

Ve zdrojových kódech přiloženého archivu je vidět seznam dostupných hlavičkových souborů. Všimněte si, že STL není dostupná. Reálný OS také nemá STL k dispozici (není new/delete), navíc byste si museli implementovat vlastní alokátory. Pro vytváření vláken použijte pthread rozhraní, C++11 thread API není v této úloze k dispozici.

SPOILER - jak na to?

- Vaše implementace si musí držet informaci o jednotlivých stránkách, zda jsou volné nebo používané. Na to se hodí nejlépe pole, jeho velikost je dokonce fixní (počet spravovaných stránek). Pro rychlé vyhledávání volné stránky je vhodné nad tímto polem vybudovat stromovou strukturu.
- Datová struktura popisující volné/alokované stránky by měla tyto informace držet uložené ve spravovaném bloku paměti (měla by si na to alokovat několik stránek). Pokud budete "podvádět" a informace o volných/alokovaných stránkách si uložíte mimo tyto stránky (alokujete si další paměť pomocí new/malloc), riskujete, že nebudete mít k dispozici dost paměti.
- Vaše implementace si musí držet seznam běžících procesů (pole o 64 položkách).
- Pro každý simulovaný proces budete muset alokovat jeho vlastní adresář stránek. Není potřeba jej alokovat celý, na to není dostatek paměti. Alokujte pouze hlavní adresář stránek a podle potřeby jej doplňujte o adresáře druhé úrovně a vlastní stránky.
- Implementace `SetMemLimit` bude upravovat adresář stránek volajícího "procesu". Před vlastní úpravou si zkontrolujte, zda máte pro požadovanou změnu dost volných stránek. Pokud ne, vraťte neúspěch a na adresářích stránek procesu nic neměňte. Neúspěch při alokaci paměti nemůže ponechat nastavení stránkování v nekonzistentním stavu.
- Vytvoření nového procesu je vlastně založení vlákna + vytvoření adresáře stránek. Trochu problematické může být zkopírování adresního prostoru volajícího do nově vzniklého procesu. Tato operace může být pomalá, navíc bude paměťově velmi náročná. Lze ji vyřešit pomocí principu copy-on-write. Při takové implementaci budete ale muset implementovat `pageFaultHandler`.
- Pro průběžné odstraňování vláken můžete použít atribut `PTHREAD_CREATE_DETACHED`, musíte ale pak nějakým Vaším synchronizačním mechanismem (podmíněná proměnná) synchronizovat hlavní vlákno, aby bylo jasné, kdy je možné simulaci ukončit.
- Nezapomeňte, že se jedná o program s více vlákny, tedy přístupy ke sdíleným prostředkům je potřeba zamykat.

Vzorová data:	**Download**

☐ **Referenční řešení**

1	**06.05.2021 20:20:01**	**Download**
Stav odevzdání:	Ohodnoceno	
Hodnocení:	30.0000	

- **Hodnotitel: automat**
  - Program zkompilován
  - Test 'Jeden proces (init)': Úspěch
    - Dosaženo: 100.00 %, požadováno: 100.00 %
    - Celková doba běhu: 0.132 s (limit: 15.000 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Sekvencni spousteni procesu': Úspěch
    - Dosaženo: 100.00 %, požadováno: 90.00 %
    - Celková doba běhu: 0.481 s (limit: 14.868 s)
    - CPU time: 0.554 s (limit: 14.868 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %

    - Test 'Paralelni spousteni procesu': Úspěch
        - Dosaženo: 100.00 %, požadováno: 90.00 %
        - Celková doba běhu: 0.157 s (limit: 14.387 s)
        - CPU time: 0.419 s (limit: 14.314 s)
        - Úspěch v závazném testu, hodnocení: 100.00 %
    - Test 'Paralelni procesy + kopie adresniho prostoru': Úspěch
        - Dosaženo: 100.00 %, požadováno: 70.00 %
        - Celková doba běhu: 0.037 s (limit: 10.000 s)
        - CPU time: 0.095 s (limit: 10.000 s)
        - Úspěch v nepovinném testu, hodnocení: 100.00 %
    - Test 'Paralelni procesy + copy-on-write': Neúspěch
        - Dosaženo: 0.00 %, požadováno: 100.00 %
        - Celková doba běhu: 0.118 s (limit: 10.000 s)
        - CPU time: 0.181 s (limit: 10.000 s)
        - Neúspěch v bonusovém testu, hodnocení: Bonus nebude udělen
        - Nesprávný výstup
    - Celkové hodnocení: 100.00 % (= 1.00 * 1.00 * 1.00 * 1.00)
- Celkové procentní hodnocení: 100.00 %
- Celkem bodů: 1.00 * 30.00 = 30.00

SW metriky:

	Celkem	Průměr	Maximum	Jméno funkce
Funkce:	**19**	--	--	--
Řádek kódu:	**217**	**11.42 ± 6.42**	**26**	`CProcess::NewProcess`
Cyklomatická složitost:	**38**	**2.00 ± 1.17**	**5**	`CProcess::SetMemLimit`