Struktura jąder systemów operacyjnych

Lista zadań programistycznych

Wersja z 19 czerwca 2020

Wprowadzenie

Rozwiązania zadań programistycznych muszą ograniczać się do plików w katalogach wskazanych w treści. Do synchronizacji należy korzystać wyłącznie z kolejek [1, §4], powiadomień [1, §9] i instrukcji atomowych. Bez wcześniejszej konsultacji z prowadzącym zajęcia **zabronione** są: modyfikacja pliku «`FreeRTOSConfig.h`», wyłączanie przerwań «`taskENTER_CRITICAL`» i wyłączanie wywłaszczania «`vTaskSuspendAll`». Rozwiązania używające kolejek do realizacji semaforów są również zabronione!

Przygotowanie rozwiązania do sprawdzania

Rozwiązania będą zbierane i oceniane przy pomocy systemu *GitHub Classroom*. Dla każdego zadania prowadzący dostarczy odnośnik do wygenerowania prywatnej wersji repozytorium, w którym należy umieścić rozwiązanie. Po utworzeniu prywatnej kopii repozytorium trzeba naprawić usterkę *GitHub Classroom* (problemy z klonowaniem repozytoriów używających `git-lfs`) wydając następujący szereg poleceń:

```
git clone https://github.com/cahirwpz/FreeRTOS-Amiga.git
cd FreeRTOS-Amiga
git lfs fetch --all
git lfs push --all git@github.com:ii-ask/ZADANIE-STUDENT.git
```

Rozwiązanie należy umieścić w gałęzi «`master`». W gałęzi «`feedback`» musi się znajdować **niezmodyfikowana** wersja gałęzi «`master`» repozytorium `FreeRTOS-Amiga`[1], która służyła za podstawę rozwiązania.

Kod w repozytorium `FreeRTOS-Amiga` jest nadal w fazie intensywnego rozwoju. Czasami będzie zachodziła potrzeba zsynchronizowania repozytorium głównego z repozytorium rozwiązania studenta. W katalogu swojego repozytorium należy wykonać poniższy szereg poleceń, będąc przygotowanym na rozwiązywanie konfliktów:

```
git checkout feedback
git pull https://github.com/cahirwpz/FreeRTOS-Amiga.git
«tu należy rozwiązać potencjalne konflikty»
git push
git checkout master
git merge feedback
```

Prowadzący będzie oceniał zmiany plików w prośbie o połączenie (ang. pull request) o nazwie «`Feedback`». Inne prośby będą przez sprawdzającego ignorowane. W zakładce „Files changed” mają być widać **wyłącznie** zmiany, które są częścią rozwiązania i zostały przygotowane przez studenta. Rozwiązania z nadmiarowymi plikami lub zbędnym kodem będą kierowane do poprawki.

Rozwiązanie zostanie odrzucone przez sprawdzarkę, jeśli wykonanie polecenia «`bash verify-format.sh`» zawiedzie. Żeby automatycznie sformatować kod należy wykonać polecenie «`make format`», przy czym narzędzie «`clang-format`» musi być co najmniej w wersji siódmej.

[1] `https://github.com/cahirwpz/FreeRTOS-Amiga`

Styl programowania

Pamiętaj, że kod Twojego rozwiązania będzie czytał śmiertelnik – im dłuższy i bardziej zagmatwany kod napiszesz, tym sprawdzający będzie bardziej krytyczny. Jeśli potrzebujesz skomentować przepływ sterowania w swoim programie, to znaczy że jest on zbyt zawiły i **należy go uprościć**. Komentarze mają sens przy opisywaniu struktur danych, funkcji pełnionych przez procedury oraz strategii synchronizacji zadań. Proszę stosować się do poniższych wytycznych stanowiących elementy dobrej kultury programisty języka `C`:

- używaj minimalnych środków do osiągnięcia celu,
- minimalizuj użycie dynamicznego przydziału pamięci,
- ograniczaj poziom zagnieżdżenia instrukcji sterujących – nie więcej niż 3 otwarte nawiasy «{»,
- właściwie nazywaj stałe, procedury, zmienne i typy,
- unikaj używania magicznych stałych – nadaj im właściwą nazwę,
- preferuj użycie procedur «`static inline`» w miejsce makrodefinicji,
- dziel długie procedury na mniejsze – procedury powyżej 40 wierszy są nieczytelne,
- wydzielaj powtarzający się kod do podprocedur,
- ograniczaj zasięg widoczności zmiennych i procedur globalnych słowem kluczowym «`static`»,
- minimalizuj globalnie utrzymywany stan programu,
- do przekazywania złożonego stanu do procedur używaj struktur i tagowanych unii,
- procedury będą mniej skomplikowane, jeśli wybierz odpowiednią reprezentację danych w pamięci,
- ograniczaj potrzebę obsługi przypadków brzegowych, na przykład przy pomocy `sentinel value`[2],
- niech procedury robią jedną rzecz, ale dobrze – nie pisz kodu generycznego, jeśli nie ma takiej potrzeby,
- po ukończeniu zadania zadaj sobie pytanie „Czy tego nie da się zrobić mniejszą liczbą linii kodu?”.

Wskazówka: Zapoznaj się z wytycznymi stylu programowania dla jądra `Linux`[3] i `FreeBSD`[4].

Uwagi techniczne

1. Podobnie jak w systemie `Unix`, we `FreeRTOS` procedury obsługi przerwań nie mogą zmieniać kontekstu. Wywłaszczanie jest rozpatrywane w trakcie powrotu z przerwania na podstawie wartości zmiennej globalnej «`xNeedRescheduleTask`».
2. Struktura rekordu aktywacji pułapki (ang. trap frame) i przerwania (ang. interrupt frame) są zmiennego rozmiaru i zależą od typu pułapki i wersji procesora (należy rozpatrywać 68000 i 68010).
3. Komunikaty diagnostyczne należy wyświetlać **wyłącznie** po umieszczeniu makrodefinicji «`DEBUG`» na początku pliku źródłowego. Domyślnie program ma ich nie wypisywać. Zalecaną metoda drukowania komunikatów jest posługiwanie się makrem «`debug`» zdefiniowanym poniżej:

```
#ifdef DEBUG
#include <stdio.h>
#define debug(...) printf(__VA_ARGS__)
#else
#define debug(...)
#endif
```

4. Masz do dyspozycji podzbiór procedur z biblioteki standardowej języka `C` zawartych w katalogu «`libc`». Implementując ich odpowiedniki wykonujesz zbędną pracę – szanuj swój czas! Zapoznaj się z plikami nagłówkowymi «`cdefs.h`», «`ctype.h`», «`stdlib.h`», «`stdio.h`», «`string.h`» i «`strings.h`».
5. W bloku kontrolnym zadania jest miejsce na dokładnie jedną 32-bitową wartość powiadomienia. To bardzo ograniczony zasób, zatem stosuj go rozsądnie. Jeśli zadanie może obsługiwać wiele zdarzeń to z każdym z nich skojarz jeden bit powiadomienia. Powiadomienia nie sprawdzają się w pewnych scenariuszach o czym można przeczytać w [1, §9.1].
6. Kolejki mogą służyć do przekazywania danych, w tym struktur i unii, przez wartość lub przez referencję. Drugi wariant jest zalecany w przypadku wątków jądra przesyłających duże ilości danych [1, §4.5], ale najpierw należy przeanalizować **prawo własności** do przekazywanej pamięci.

[2] `https://en.wikipedia.org/wiki/Sentinel_value`
[3] `https://www.kernel.org/doc/html/latest/process/coding-style.html`
[4] `https://www.freebsd.org/cgi/man.cgi?query=style&sektion=9`

Zadanie 1 (2). Użytkownik dostarczył program używający instrukcji, które nie są zaimplementowane w procesorze m68000. Mimo wszystko chcemy, by program wykonywał się poprawnie, nawet kosztem efektywności. Problemem jest procedura «rand» z pliku «rand.c», która zawiera instrukcje «divsl» i «muls» opisane odpowiednio w [3, 4-92] i [3, 4-135]. Kodowanie binarne tych instrukcji zostało podane poniżej.

```
divsl.l #127773,%d0,%d2   # 4c7c 2800 0001 f31d
muls.l  #16807,%d0        # 4c3c 0800 0000 41a7
muls.l  #-2836,%d2        # 4c3c 2800 ffff f4ec
```

Używając jako szkieletu rozwiązania przykładowego programu «instemul» uzupełnij procedurę obsługi pułapek «vPortTrapHandler». Procedura powinna wykryć instrukcje w formacie «divsl.l #imm,Dx,Dy» oraz «muls.l #imm,Dx» i emulować ich wykonanie. Po otrzymaniu wyniku należy odpowiednio zmodyfikować kontekst przerwanego programu przechowywany pod wskaźnikiem «frame». Jeśli nie rozpoznano niewłaściwej instrukcji albo wykryto inny błąd wykonania, to należy wywołać «vPortDefaultTrapHandler».

Emulacja instrukcji powinna korzystać z procedur znajdujących się w katalogu «libc/gen»[5]. Należy pamiętać o prawidłowym ustawieniu rejestru kodów warunkowych CCR będącego częścią rejestru SR. W przypadku wykrycia błędu dzielenia przez zero należy zgłosić odpowiedni wyjątek procesora – w takim przypadku licznik instrukcji w ramce pułapki (ang. trap frame) powinien wskazywać na instrukcję dzielenia.

Zadanie 2 (3). Zaprogramuj wątek sterownika obsługi terminala znakowego. Punktem startowym powinien być plik «tty.c» należący do przykładowego programu «console». Komunikacja z terminalem przebiega przy pomocy zestawu procedur działających na obiektach plikopodobnych zdefiniowanych w «include/file.h». Sterownik terminala ma implementować poniższe procedury należące do interfejsu «FileOps_t», który wzorowano na uniksowym fileops opisanym w file(9).

```
File_t *TtyOpen(void);
void TtyClose(File_t *f);
long TtyRead(File_t *f, void *buf, size_t nbyte);
long TtyWrite(File_t *f, const void *buf, size_t nbyte);
```

Sterownik ma być zaimplementowany jako **jeden** wątek, z którym procedury «TtyRead» i «TtyWrite» komunikują się przy pomocy kolejek systemu FreeRTOS. Wątek sterownika ma odpowiadać za odświeżanie zawartości ekranu i wczytywanie danych z klawiatury. Należy go utworzyć przy pierwszym otwarciu pliku terminala, a zakończyć po obniżeniu wartości licznika referencji «usecount» do 0. Wątek musi zainicjować dostarczony sterownik klawiatury «drivers/keyboard.c» przy pomocy «KeyboardInit». Przed zakończeniem działania wątek musi zabić sterownik klawiatury przy pomocy «KeyboardKill». Sterownik należy zaprogramować tak, żeby z terminala mógł na raz korzystać więcej niż jeden wątek, a operacje odczytu i zapisu były atomowe (niepodzielne), tj. należy wymieniać dane z terminalem całymi paczkami, a nie pojedynczymi znakami.

W trakcie obsługi żądania od «TtyWrite» należy pamiętać o prawidłowym wyświetlaniu położenia kursora. Gdyby kursor miał wyjechać za ostatni wiersz, to należy przewinąć zawartość ekranu o jeden wiersz do góry.

Wczytywanie danych przy pomocy «TtyRead» powinno zachowywać się jak read na uniksowym urządzeniu terminala znakowego działającego w trybie kanonicznym, tj. ma działać buforowanie i prymitywna metoda edycji linii. Znaczy to, że «TtyRead» ma zwrócić wartość, jeśli bufor wejściowy został zapełniony lub pojawił się w nim znak końca linii. Zakładamy, że $\texttt{MAX\_CANON} = 80$, czyli tyle ile szerokość ekranu w znakach. Do zadań edycji linii należy obsługa znaków specjalnych ERASE (kasuje ostatni znak, domyślnie «Backspace») i KILL (kasuje całą linię, domyślnie «Ctrl+U»).

Uwaga: Nie można dopuścić do głodzenia funkcji odbierającej znaki od przerwania klawiatury!

Zauważ, że wątek sterownika terminala może w jednej chwili czekać na wejście z klawiatury, albo żądanie zapisu z «TtyWrite», zatem będzie potrzebne użycie powiadomień. Zdecydowana większość logiki obsługi terminala powinna być zawarta w obrębie wątku sterownika. Użytkownik interfejsu plikowego ma jedynie wysyłać żądania do sterownika i oczekiwać odpowiedzi. Kod wykonywany w procedurze obsługi przerwania klawiatury musi być bardzo krótki, w szczególności nie może obsługiwać logiki związanej z dostarczaniem znaku do użytkownika interfejsu plikowego.

[5] Kompilator używa ich w trakcie generowania kodu, aby zastąpić nieistniejące instrukcje 32-bitowego mnożenia i dzielenia.

Program testowy korzystający z «TtyRead» i «TtyWrite» ma przyjmować dwa polecenia:

- «c n rgb» zmieniającą kolor palety o numerze «n» na «rgb», gdzie n, r, g i b to cyfry szesnastkowe,
- «d start end» drukującą (tak jak polecenie «od -t xC») zawartość pamięci od adresu start do adresu end, które są podane jako liczby szesnastkowe.

Wskazówka: Więcej na temat interakcji z urządzeniem terminala można przeczytać w [2, §62].

Zadanie 3 (5). Komputer Amiga 500 obsługuje dyskietki dwustronne podwójnej gęstości[6] o pojemności 880KiB. Dyskietka posiada 160 ścieżek, każda po 11 sektorów, z których każdy przechowuje 512-bajtów danych użytkownika. Sektory posiadają 32-bajtowy nagłówek i są zakodowane przy użyciu kodowania MFM[7]. Detalami związanymi z obsługą kontrolera stacji dyskietek, dekodowania ścieżek i sektorów zajmuje się kod w plikach «drivers/floppy.c» i «drivers/floppy-mfm.c».

Program «tools/fsutil.py» służy do tworzenia obrazu dyskietki z prostym systemem plików tylko do odczytu. Umożliwia ono również drukowanie zawartości obrazu dyskietki oraz wyłuskiwanie z niego plików.

Struktura systemu plików przechowywanego na dyskietce jest następująca. Pierwsze dwa sektory, zaczynając numerację od 0, przechowują kod rozruchowy (ang. boot block) z pliku «bootloader.asm» oraz numer sektora początkowego i rozmiar pliku wykonywalnego do załadowania. Kolejne sektory przechowują katalog główny, przy czym system plików nie wspiera podkatalogów. Dalej mamy plik wykonywalny ładowany przez bootloader i sektory zawierające dane pozostałych plików. Szczegóły zapisu systemu plików zawarto w komentarzu na początku pliku «fsutil.py».

Twoim zadaniem jest zaprogramowanie wątku obsługi prostego systemu plików bazując na źródłach w katalogu «examples/floppy». Podany poniżej interfejs musi komunikować się z wątkiem «vFileSysTask» przy pomocy komunikatów typu «FsMsg_t».

```
bool FsMount(void);
int FsUnMount(void);
const DirEntry_t *FsListDir(void **base_p);
File_t *FsOpen(const char *name);
```

Po inicjacji sterownika procedurą «FsInit» programista powinien zamontować nośnik posługując się «FsMount». Na początku należy załadować cały katalog główny do pamięci operacyjnej – wystarczy wczytać rozmiar katalogu i wszystkie jego wpisy dwoma operacjami dyskowymi. Proszę nie modyfikować struktury «DirEntry_t». Po zamonowaniu nośnika będzie można wydrukować jego zawartość przy pomocy «FsListDir». Będzie można również otwierać pliki «FsOpen» o podanej nazwie, przy czym zwracany obiekt interfejsu plikowego musi implementować procedury «FsRead», «FsSeek» i «FsClose». Procedura «FsUnMount» zwraca wartość oznaczającą liczbę otwartych plików i jeśli liczba ta wynosi zero, to nastąpiło odmontowanie nośnika.

Wskazówka: Zauważ, że należy zapobiec wyścigom między «FsUnMount», a «FsListDir» i «FsOpen».

Zakładamy, co następuje:

- użytkownik komputera nie wyciągnie dyskietki z napędu bez odmontowania nośnika,
- nośnik nie posiada błędnych sektorów,
- wywołanie «FsOpen» tworzy nową instancję otwartego pliku niezależną od reszty otwartych plików,
- zadanie może posiadać wiele otwartych plików, ale nie może współdzielić ich instacji z innymi zadaniami,
- nie ma ograniczenia górnego, poza rozmiarem pamięci operacyjnej, na liczbę otwartych plików,
- może istnieć wiele instacji otwartych plików «File_t», które odnoszą się do tego samego obiektu na dysku – jest to możliwe w wyniku wielokrotnego wywołania «FsOpen» z tą samą nazwą pliku,
- nie ma potrzeby reprezentacji obiektów dyskowych w pamięci chyba, że zamierzasz zaimplementować zaawansowaną metodę buforowania plików.

Wskazówka: Struktura otwartego pliku «File_t» odpowiada file(9), a obiektu dyskowego vnode(9).

[6] https://en.wikipedia.org/wiki/List_of_floppy_disk_formats#Logical_formats
[7] https://en.wikipedia.org/wiki/Modified_frequency_modulation

Implementacja systemu plików będzie wymagała komunikacji z wątkiem sterownika kontrolera dyskietki, którego interfejs widnieje w pliku «floppy.h». Sterownik posiada kolejkę żądań typu «FloppyIO_t». Żeby odczytać zakodowaną ścieżkę z zakresu 0 do 159, należy przydzielić na nią miejsce w pamięci typu CHIP[8] przy pomocy «AllocTrack». Po wykonaniu żądania sterownik umieszcza komunikat w kolejce zwrotnej «replyQueue». Oznacza to, że sterownik **nie musi** spełniać żądań w kolejności ich przychodzenia.

Po sprowadzeniu ścieżki z nośnika do pamięci jeszcze nie znamy pozycji sektorów – ich początki wyznaczysz procedurą «DecodeTrack». Każdy sektor posiada 32-bajtowy nagłówek i jest zapisany w kodowaniu MFM. Żeby zdekodować dane wybranego sektora należy użyć procedury «DecodeSector».

Zauważ, że użytownik procedury «FileRead» może czytać dane z pliku małymi paczkami, np. po kilkanaście bajtów. W takim przypadku wczytywanie ścieżki z dyskietki przy każdej operacji odczytu z pliku jest wyjątkowo nieefektywne. Żeby tego uniknąć należy dodać wsparcie dla buforowania zawartości plików. Łączna ilość pamięci zarezerowanej na bufory i ścieżki nie może przekraczać 64KiB. W uproszczonym wariancie wystarczy buforować kilka ścieżek, w bardziej złożonym można buforować zdekodowane sektory. Zakodowana ścieżka zajmuje 12800 bajtów pamięci CHIP, a po zdekodowaniu zajmuje 5.5KiB, tj. 11 sektorów po 512 bajtów.

Do swojego rozwiązania należy dołączyć test, w którym uruchomisz kilka wątków, z których każdy w pętli będzie wykonywał co następuje:

1. Otwórz losowy plik, którego nazwę otrzymano w wyniku skanowania katalogu.
2. Czytaj z pliku kilkaset bajtów co sekundę, aż do osiągnięcia końca pliku.
3. Zamknij plik, idź do punktu 1.

Dodatkowo należy uruchomić wątek interaktywny, który korzystając z portu szeregowego udostępni linię poleceń, w której będzie można wydawać następujące polecenia:

- «open name» otwiera plik o nazwie name i zwraca jego numer od 0 do 9,
- «close num» zamyka plik o numerze num,
- «file num» przełącza plik na którym wykonujemy operacje na num,
- «read n» wczytuje n bajtów z danego pliku i drukuje je na terminal w postaci podobnej do wyjścia z polecenia «hexdump -C», przy czym w pierwszej kolumnie należy umieścić pozycję kursora pliku,
- «seek off whence» ustawia kursor pliku zgodnie z semantyką procedury «FsSeek» i podaje bieżącą wartość kursora pliku; parametr off może być liczbą ujemną, a whence to opcjonalny ciąg znaków «cur» (domyślnie), «set», «end».
- «ls» drukuje zawartość katalogu głównego, w każdym wierszy wydruku ma widnieć nazwa pliku, sektor początkowy, długość w bajtach oraz rodzaj pliku (wykonywalny albo zwykły),

Program «make» w trakcie przetwarzania każdego z celów «run-rom», «debug-rom», «run-floppy» i «debug-floppy» tworzy obraz dyskietki w formacie Amiga Disk File[9]. Domyślnie w obrazie znajduje się plik wykonywalny i program rozruchowy. Do obrazu należy dodać testowe pliki, co można osiągnąć dopisując wybrane pliki do zmiennej «ADF-EXTRA» w pliku «Makefile».

Zadanie 4 (6). Przykład zawarty w katalogu «usermode» prezentuje bardzo proste podejście do realizacji procesów uniksopodobnych dla systemu FreeRTOS. Procesory z rodziny m68k mogą działać w trybie nadzorcy (ang. supervisor), tj. jądra, i użytkownika. Instrukcja pułapki «trap» o wybranym numerze może posłużyć do implementacji wywołań systemowych. Niestety procesor 68000 nie posiada MMU, zatem nie będzie możliwości realizacji odrębnych przestrzeni adresowych – taką możliwość daje dopiero procesor 68030 i późniejsze. Uruchomienie wielu programów we wspólnej przestrzeni adresowej będzie wymagało użycia relokowalnych plików wykonywalnych – w tym celu zostanie wykorzystany format AmigaHunk[10].

Zadanie konstruuje strukturę procesu «Proc_t», która trzyma zasoby zarządzane przez jądro i używane w przestrzeni użytkownika, tj.: stos, listę segmentów programu, tablicę deskryptorów plików oraz rozruchowy stan rejestrów. Procedura «TaskSetProc» służy do skojarzenia struktury procesu z zadaniem, co umożliwia łatwe uzyskanie dostępu do bloku kontrolnego procesu, przy użyciu procedury «TaskGetProc», również w kontekście procedury obsługi pułapki lub przerwania. Pliki wykonywalne przystosowane do uruchomienia

[8] Transfery DMA potrafią korzystać wyłącznie z pamięci dostępnej dla układów specjalizowanych.
[9] https://en.wikipedia.org/wiki/Amiga_Disk_File
[10] http://amiga-dev.wikidot.com/file-format:hunk

w przestrzeni użytkownika, czyli «ushell.exe» (prosta powłoka) i «ucat.exe» (uniksowy program cat), zostały włączone do pliku «usermode.elf» jako tablice bajtów. Można z nich skonstruować obiekt plikopodobny przy pomocy procedury «MemoryOpen». Procedura «ProcLoadImage» ładuje z pliku do pamięci segmenty pliku wykonywalnego, dokonuje relokacji, po czym ustala początkową wartość rejestru PC. W tablicy deskryptorów plików, przy pomocy procedury «ProcFileInstall», zainstalowano plik urządzenia portu szeregowego jako standardowe wejście i wyjście. Przed rozpoczęciem wykonania nowego programu należy przy pomocy «ProcSetArgv» wkopiować na stos użytkownika parametry programu i ustalić początową wartość rejestru SP. Żeby rozpocząć wykonywanie programu trzeba wywołać procedurę «ProcEnter», która zapamięta powrotny kontekst jądra, a następnie przełączy sterowanie do przestrzeni użytkownika (przy pomocy szczypty asemblera w pliku «trampoline.S»). Żeby zakończyć wykonanie programu jądro musi wywołać procedurę «ProcExit», które przywraca kontekst procedury «ProcEnter» tak, by ta wróciła zamiast przechodzić do trybu użytkownika.

Obsługa wywołań systemowych jest zawarta w pliku «sysent.c». Dostępny kod źródłowy zawiera wyłącznie implementację exit(2). Listę numerów wywołań systemowych zawarto w pliku «syscall.h». Po stronie przestrzeni użytkownika mamy procedury, które zaimplementowano w pliku «sysapi.c». Służą one do wzywania wywołań systemowych i część z nich należy uzupełnić.

Zadanie jest podzielone na dwie czeście. W pierwszej zajmij się implementacją obsługi wywołań systemowych służących do obsługi plików, tj. open, close, read, write. Przygotuj kilka obiektów plikopodobnych odnoszących się do ciągów znaków i nadaj im nazwy, które pozwolą na ich otwieranie z przestrzeni użytkownika. Innymi słowy skonstruuj tablicę par (nazwa, obiekt), której użyjesz w trakcie implementacji wywołania open. Wykorzystaj program «ucat.exe» do testów.

Wskazówka: Istnieje możliwości debugowania programu w przestrzeni użytownika przy pomocy gdb. Trzeba się dowiedzieć pod jakie adresy zostały załadowane sekcje programu. Będąc uzbrojonym w taką wiedzę, można skojarzyć symbole z załadowaną sekcją .text przy pomocy polecenia «add-symbol-file».

W drugiej części zadania należy zaimplementować uproszczone wersje wywołań systemowych vfork(2), wait(2) i execv(3) tak, by pozwolić na uruchomienie i korzystanie z programu «ushell.exe», w szczególności wywoływanie programu cat. Przeczytaj uważnie podane podręczniki systemowe, żeby dowiedzieć się jak wystartować program w nowym procesie. Na pewno będzie trzeba zablokować wykonanie procesu rodzica do momentu wykonania execv przez proces potomny. Nasz system nie implementuje zmiennych środowiskowych, stąd wystarczy nam execv zamiast execve(3). Wywołanie wait czeka na dowolny proces potomny, jeśli takowy istnieje, po czym zwraca jego pid i kopiuje jego kod wyjścia pod ustalony adres. Dla prostoty można założyć, że proces może zakończyć działanie tylko i wyłącznie, jeśli nie ma żadnych potomków – żywych lub martwych. Druga część zadania będzie cięższa – przed przystąpieniem do implementacji przemyśl dokładnie synchronizację między procesami. Być może w obrębie procedury «ProcEnter», a przed powiadomieniem rodzica, będzie trzeba wyłączyć przerwania przy pomocy «ulPortSetIPL». Do skopiowania kontekstu wywołania vfork z procesu rodzica do procesu potomnego należy użyć procedury «CloneUserCtx». Ponieważ proces potomny zostanie uruchomiony na stosie rodzica, to może okazać się koniecznym zachowanie stanu stosu przed rozpoczęciem wykonywania procesu potomnego i jego przywrócenie po wywołaniu execv.

UWAGA: Wszelkie zmiany w plikach źródłowych «shell.c» i «cat.c» należy konsultować z prowadzącym zajęcia.

Literatura

[1] „Mastering the FreeRTOS™ Real Time Kernel”
https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

[2] „Linux Programming Interface: A Linux and UNIX® System Programming Handbook”
Michael Kerrisk; No Starch Press; 2010

[3] „Motorola M68000 Family Programmer's Reference Manual”
https://www.nxp.com/docs/en/reference-manual/M68000PRM.pdf