NOTATKA: PROCESY

Proces

- abstrakcja działającego programu
- w każdym systemie wieloprogramowym procesor szybko przełącza się pomiędzy procesami, poświęcając każdemu z nich po kolei dziesiątki albo setki milisekund. Chociaż w dowolnym momencie procesor realizuje tylko jeden proces, w ciągu sekundy może obsłużyć ich wiele, co daje iluzję współbieżności

Model procesów

- całe oprogramowanie możliwe do uruchomienia na komputerze jest zorganizowane w postaci zbioru procesów sekwencyjnych
- proces jest egzemplarzem uruchomionego programu włącznie z bieżącymi wartościami licznika programu, rejestrów i zmiennych
- każdy proces ma swój własny wirtualny procesor CPU, oczywiście w rzeczywistości procesor fizyczny przełącza się od procesu do procesu (wieloprogramowość)
- przykład: w pamięci komputera w trybie wieloprogramowym działają cztery programy. Cztery niezależne od siebie procesy – każdy ma swój własny przepływ sterowania (własny logiczny licznik programu). Oczywiście jest tylko jeden fizyczny licznik programu, dlatego kiedy działa wybrany proces, jego logiczny licznik jest zapisywany w logicznym liczniku programu umieszczonym w pamięci. W dłuższym przedziale czasu nastąpi postęp we wszystkich procesach, jednak w danym momencie działa tylko jeden proces.
- jeden rdzeń – jeden proces, więc więcej rdzeni w procesorze/procesorów – więcej procesów działa naraz

Tworzenie procesów

- inicjalizacja systemu
- uruchomienie wywołania systemowego tworzącego proces przez działający proces (UNIX: fork)
- żądanie usera utworzenia nowego procesu
- zainicjowanie zadania wsadowego (systemy wsadowe w komputerach mainframe)

Kończenie działania procesów

- normalne zakończenie pracy (dobrowolnie)
- zakończenie pracy w wyniku błędu (dobrowolnie)
- błąd krytyczny (przymusowo)
- zniszczenie przez inny proces (przymusowo)

Stany procesów

- wykonywany – rzeczywiste korzystanie z procesora w danym momencie
- zablokowany – proces nie może działać do momentu, w którym wydarzy się jakieś zewnętrzne zdarzenie (proces nie może działać nawet wtedy, gdy procesor nie ma innego zajęcia)
- gotowy – proces może działać, ale jest tymczasowo wstrzymany, żeby inny proces mógł działać

Przejścia między stanami realizowane są przez program szeregujący (część sysopka).

Implementacja procesów

- w sysopku występuje tabela procesów, w której każdemu z procesów odpowiada jedna pozycja – bloki zarządzania procesami, które zawierają:
    - kontekst procesora (zawartość rejestrów, SP, PC)
    - informacje dla planisty (zużycie procesora, priorytet)
    - stan procesu
    - identyfikatory, uprawnienia
    - obraz pamięci (opis stanu przestrzeni adresowej)
    - informacje rozliczeniowe (pomiar zużycia zasobów, profilowanie)
    - uchwyty do używanych zasobów (pliki, IPC)

NOTATKA: WĄTKI

Wątek

- procesy są wykorzystywane do grupowania zasobów, wątki są podmiotami zaplanowanymi do wykonywania przez procesor
- zawiera licznik programu, który śledzi to, jaka instrukcja będzie wykonywana w następnej kolejności
- posiada rejestry zawierające jego bieżące robocze zmienne
- ma do dyspozycji stos zawierający historię działania – po jednej ramce dla każdej procedury, której wykonywanie się rozpoczęło, ale jeszcze nie zakończyło
- wielowątkowość (wątki) działa tak samo jak wieloprogramowość (procesy) - procesor przełącza się w szybkim tempie pomiędzy wątkami, dając iluzję, że wątki działają równolegle
- różne wątki procesu nie są tak niezależne, jak rożne procesy – wszystkie posługują się tą samą przestrzenią adresową, więc współdzielą te same zmienne globalne. Jeden wątek może odczytać, zapisać, a nawet wyczyścić stos innego wątku (nie ma zabezpieczeń).
- procesy potencjalnie należą do różnych użytkowników i mogą być dla siebie wrogie – proces zawsze należy do jednego użytkownika, który przypuszczalnie stworzył wiele wątków, a zatem powinny współpracować, a nie walczyć ze sobą
- stany: działający, zablokowany, gotowy, zakończony

Komponenty procesu (wspólne dla wątków w procesie)	**Komponenty wątku (prywatne dla każdego wątku)**
przestrzeń adresowa (zmienne, kod)	licznik programu
zmienne globalne	rejestry
otwarte pliki	stos
procesy-dzieci	stan
sygnały i procedury obsługi sygnałów	
informacje dotyczące statystyk	

L11.Z1. Czym różni się **przetwarzanie równoległe** (ang. *parallel*) od **przetwarzania współbieżnego** (ang. *concurrent*)? Czym charakteryzują się **procedury wielobieżne** (ang. *reentrant*)? Podaj przykład procedury (a) wielobieżnej, ale nie **wątkowo-bezpiecznej** (ang. *thread-safe*), (b) na odwrót. Kiedy w jednowątkowym procesie uniksowym może wystąpić współbieżność?
Wskazówka: Rozważ procedury obsługi sygnałów.

Intuicja: współbieżne - żongler niby obsługuje wiele piłeczek, ale w ręce ma tylko jedną, równoległe - kilku żonglerów

przetwarzanie równoległe (parallel) – działanie wielu procesów jednocześnie dzięki wielu rdzeniom w procesorze lub wielu procesorom. Wtedy system operacyjny nie musi już "oszukiwać" dzieląc czas jedynego procesora między wiele procesów, lecz może wykonywać każdy z nich na innym procesorze (przynajmniej dopóki wystarczy procesorów).

Procesor wielordzeniowy to procesor z kilkoma jednostkami wykonawczymi ("rdzeniami"). Procesory te różnią się od procesorów superskalarnych, gdyż mogą wykonywać jednocześnie instrukcje pochodzące z różnych ciągów instrukcji; w przeciwieństwie do tych drugich, które co prawda również mogą w pewnych warunkach wykonywać kilka instrukcji jednocześnie, ale pochodzących z jednego ciągu instrukcji (w danej chwili wykonują tylko jeden wątek).

przetwarzanie współbieżne (concurrent) – procesor przełącza się w szybkim tempie pomiędzy procesami lub wątkami. Odbywa się to w ten sposób, że proces otrzymuje pewien kwant czasu (rzędu milisekund), w czasie którego korzysta z procesora. Gdy kwant skończy się, system operacyjny "przełącza" procesy: odkłada ten, który się wykonywał na później i zajmuje się innym. Ponieważ przełączanie odbywa się często, więc użytkownicy komputera mają wrażenie, że komputer zajmuje się wyłącznie ich procesem.

Co więcej, rozwiązując pewien problem, użytkownik może uruchomić "jednocześnie" dwa procesy, które ze sobą w pewien sposób współpracują. Jest to powszechnie stosowane na przykład w środowisku systemu operacyjnego Unix, który umożliwia wykonywanie programów w potoku, np.: ls -l | grep moje | more. Wszystkie trzy programy wchodzące w skład takiego potoku wykonują się współbieżnie, z tym że wyniki generowane przez pierwszy z nich są podawane na wejście drugiego, wyniki drugiego - na wejście trzeciego itd.

funkcje wielobieżne (reentrant) – funkcja jest wielobieżna, jeśli wykonywanie jej może zostać przerwane (poprzez przerwanie lub wywołanie innej funkcji wewnątrz ciała funkcji), a potem może ona zostać ponownie wywołana zanim poprzednie wywołanie zostanie zakończone. Po zakończeniu drugiego wywołania, można wrócić do przerwanego, a wykonywanie go może bezpiecznie kontynuować. Funkcje wielobieżne można więc wywoływać rekurencyjnie. Funkcje takie nie mogą korzystać z static or global non-constant data i wołać non-reentrant routines.

funkcje wątkowo-bezpieczne (thread-safe) – funkcja jest thread-safe, jeśli może być wywoływana jednocześnie przez wiele wątków, nawet jeśli wywołania używają współdzielonych danych, a dostęp do nich jest możliwy dla dokładnie jednego wątku w danym czasie (shared data are serialized).

Każda funkcja, która nie korzysta ze static data i innych dzielonych zasobów na pewno jest thread-safe. Używanie zmiennych globalnych jest thread-unsafe. Np. wątek B może przeczytać kod błędu odpowiadający błędowi spowodowanemu przez wątek A. (Rozwiązanie: przypisanie każdemu wątkowi własnych, prywatnych zmiennych globalnych – każdy wątek będzie miał własną kopię zmiennej errno i innych zmiennych globalnych, co pozwoli na uniknięcie konfliktów).

Funkcje wielobieżne również mogą być wywoływane jednocześnie przez wiele wątków, ale tylko gdy każde wywołanie używa swoich własnych danych, stąd wielobieżna nie musi być thread safe.

Czy w jednowątkowym procesie uniksowym może wystąpić współbieżność?

W jednowątkowym procesie uniksowym może wystąpić współbieżność - poprzez model maszyny stanowej. Jeden wątek kontroluje wiele współbieżnych wykonań. Trzeba każdemu symulować stos, używać nieblokujących wywołań systemowych (np. aio_read).

Przykład: serwer www. Załóżmy, że wątki nie są dostępne, ale projektanci systemu uznali obniżenie wydajności spowodowane istnieniem pojedynczego wątku za niedopuszczalne. Jeśli jest dostępna nieblokująca wersja wywołania systemowego read, możliwe staje się trzecie podejście (nie: wielowątkowy i jednowątkowy serwer www). Jeżeli żądanie może być obsłużone z pamięci podręcznej, to dobrze, a jeśli nie, inicjowana jest nieblokująca operacja dyskowa. Serwer rejestruje stan bieżącego żądania w tabeli, a następnie pobiera następne zdarzenie do obsługi. Może to być żądanie nowej pracy albo odpowiedź dysku dotycząca poprzedniej operacji. Jeśli to jest żądanie nowej pracy, rozpoczyna się jego obsługa. Jeśli jest to odpowiedź z dysku, właściwe informacje są pobierane z tabeli i następuje przetwarzanie odpowiedzi. W przypadku nieblokujących dyskowych operacji IO odpowiedź zwykle ma postać sygnału lub przerwania.

Podaj przykład procedury wielobieżnej, ale nie wątkowo-bezpiecznej i na odwrót

Example 1: not thread-safe, not reentrant

```
/* As this function uses a non-const global variable without any precaution, it is neither reentrant
nor thread-safe. */
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;

    // hardware interrupt might invoke isr() here!!
    *y = t;
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
```

Example 2: thread-safe, not reentrant

```
/* We use a thread local variable: the function is now thread-safe but still not reentrant (within he
   same thread). */

__thread int t;

…
```

Example 3: not thread-safe, reentrant

```
/* We save the global state in a local variable and we restore it at the end of the function.
The function is now reentrant but it is not thread safe. */

int t;

void swap(int *x, int *y) {
    int s;
    s = t; // save global variable
    t = *x;
    *x = *y;

// hardware interrupt might invoke isr() here!
    *y = t;
    t = s; // restore global variable
}
```

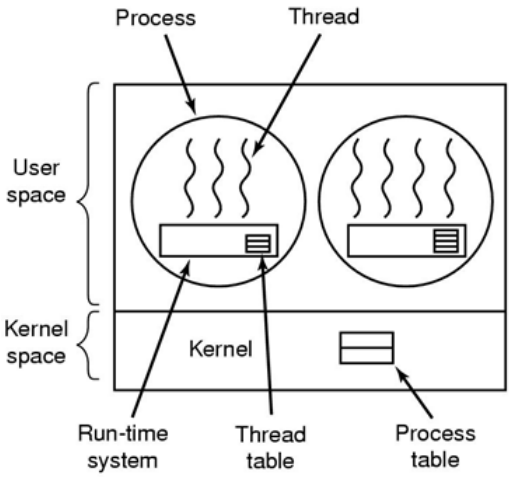
po wywołaniu swap zmienna t wraca do poprzedniego stanu (więc można w trakcie swap wywołać jeszcze jedno swap, które przywróci stary kontekst po swoim zakończeniu) ale nie jest thread-safe, bo inny nasz wątek może zostać wywłaszczony, kolejny wątek zacznie wykonywać swap, zmieni zmienną t, a następnie sam zostanie wywłaszczony i wrócimy do pierwszego

Example 4: thread-safe, reentrant

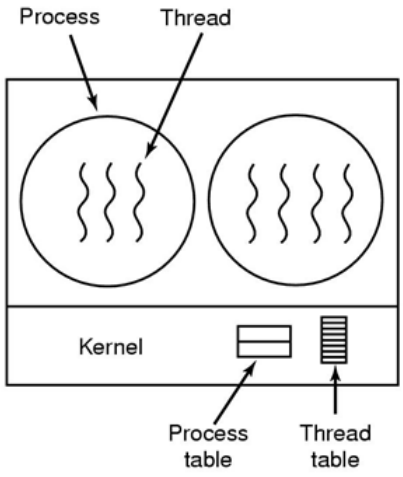
```
/* We use a local variable: the function is now thread-safe and reentrant, we have ascended to higher
plane of existence.  */

void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
// hardware interrupt might invoke isr() here!
    *y = t;
}
```

L11.Z2. Rozważamy **wątki przestrzeni jądra** (ang. *kernel-level threads*). Czym różni się **przełączanie kontekstu** od **przełączania trybu pracy**? Gdzie jądro zachowuje kontekst wątku w trakcie przejścia z przestrzeni użytkownika do przestrzeni jądra? Czemu przełączanie między wątkami należącymi do różnych procesów jest bardziej kosztowne niż w obrębie tego samego procesu? Które z zasobów wymienionych w tabeli 2.4 (§2.1.6) należą do wątku, a które do procesu?

L11.Z3. Wymień przewagi wątków przestrzeni jądra nad **wątkami przestrzeni użytkownika** (ang. *userlevel threads*) zwanych dalej włóknami. Czemu biblioteka ULT musi kompensować brak wsparcia jądra z użyciem **opakowań funkcji** (ang. *wrapper*)? Jakie zdarzenia wymuszają przełączenie kontekstu między włóknami? Co dzieje się w przypadku kiedy włókno spowoduje błąd strony?



A user-level threads package.



A threads package managed by the kernel.

wątki przestrzeni jądra – jądro wie o istnieniu wątków i nimi zarządza

- środowisko wykonawcze w każdym z procesów nie jest wymagane
- jądro dysponuje tabelą wątków
- kiedy wątek chce utworzyć nowy wątek lub zniszczyć istniejący, wykonuje wywsysa, który następnie realizuje te operacje poprzez aktualizację tabeli wątków na poziomie jądra
- zalety:
  - wątki nie muszą dobrowolnie oddawać CPU, zostaną wywłaszczone
  - można rozdzielać wątki na wiele procesorów, w ULT nie
  - planista na wątkach już jest, nie musi być zaimplementowany osobno
  - page fault – jeśli wystąpi, to jądro może łatwo sprawdzić, czy proces zawiera inne wątki możliwe do uruchomienia. Jeśli tak, to uruchamia jeden z nich w oczekiwaniu na przesłanie z dysku wymaganej strony.
- wady:
  - większy koszt wszystkich wywołań, dlatego w przypadku dużej liczby operacji zarządzania wątkami (tworzenia, niszczenia itd.) ponoszone koszty obliczeniowe są wysokie (ale niektóre systemy robią recykling wątków, wykorzystując je ponownie)
  - co gdy wielowątkowy proces wykona wywołanie fork? Czy nowy proces będzie miał tyle wątków, ile ma stary, czy tylko jeden? Zależy od tego, do czego będzie służył nowy proces. Jeśli zamierza użyć wywsysa exec do uruchomienia nowego programu, to jeden wątek. Jeśli ma kontynuować działanie, to wszystkie wątki.
  - sygnały – sygnały są przesyłane do procesów, a nie do wątków. Który wątek ma obsłużyć nadchodzący sygnał? Można rejestrować zainteresowanie określonym sygnałem przez wątek, ale co jeśli dwa wątki zainteresują się tym samym sygnałem?

wątki przestrzeni użytkownika – umieszczenie pakietu wątków w całości w przestrzeni użytkownika, jądro nic o nich nie wie.

- z punktu widzenia jądra procesy, którymi zarządza, są jednowątkowe
- wątki działają na bazie środowiska wykonawczego – kolekcji procedur, które nimi zarządzają (pthread_create, pthread_exit, pthread_join, pthread_yield)

Tabela wątków

- każdy proces potrzebuje swojej prywatnej tabeli wątków (analogiczna do tabeli procesów, śledzi właściwości na poziomie wątku – licznik programu, wskaźnik stosu, rejestry, stan itp.)
- tabela wątków jest zarządzana przez środowisko wykonawcze – kiedy wątek przechodzi do stanu gotowości lub zablokowania, informacje potrzebne do jego wznowienia są zapisywane w tabeli wątków (analogia do tabeli procesów)
- kiedy wątek wykona operację, która może spowodować jego lokalne zablokowanie, np. oczekuje, aż inny wątek w tym samym procesie wykona jakaś pracę, wykonuje procedurę ze środowiska wykonawczego. Procedura ta sprawdza, czy wątek musi być przełączony do stanu zablokowania. Jeśli tak, to zapisuje rejestry wątku (tzn. własne) w tabeli wątków, szuka w tabeli wątku gotowego do działania i ładuje rejestry maszyny zapisanymi wartościami odpowiadającymi nowemu wątkowi.

Zalety i wady rozwiązania

- zalety:
  - można zaimplementować na sysopku, który nie obsługuje wątków – wątki są implementowane za pomocą biblioteki
  - jeśli maszyna posiada instrukcję zapisującą wszystkie rejestry oraz inną instrukcję, która je wszystkie ładuje, przełączanie wątków można przeprowadzić za pomocą zaledwie kilku instrukcji – jest to o wiele szybsze od wykonywania rozkazu pułapki do jądra.
    Procedura zapisująca stan wątku oraz program szeregujący są po prostu lokalnymi procedurami, zatem wywołanie ich jest znacznie bardziej wydajne od wykonania wywołania jądra (nie jest potrzebny rozkaz pułapki, nie trzeba przełączać kontekstu, nie trzeba opróżniać pamięci podręcznej).
  - każdemu procesowi można przypisać własny, spersonalizowany algorytm szeregowania
- wady:
  - blokujące wywołania systemowe blokują wszystkie wątki. Przypuśćmy, że wątek czyta z klawiatury, zanim zostanie wciśnięty jakikolwiek klawisz. Zezwolenie wątkowi na wykonanie wywołania systemowego jest niedopuszczalne, ponieważ spowoduje to zatrzymanie wszystkich wątków. Trzeba pamiętać, że jednym z podstawowych celów korzystania z wątków jest umożliwienie wszystkim wątkom używania wywołań blokujących, a przy tym niedopuszczenie do tego, by zablokowany wątek miał wpływ na inne. W przypadku blokujących wywsysów trudno znaleźć łatwe rozwiązanie pozwalające na spełnienie tego celu.
  - programiści, ogólnie rzecz biorąc, potrzebują wątków w aplikacjach, w których wątki blokują się często – np. w wielowątkowym serwerze WWW. Wątki te bezustannie wykonują wywsysy.

Wszystkie wywołania systemowe można zmienić na nieblokujące (np. odczyt z klawiatury zwróciłby 0 bajtów, gdyby znaki nie były wcześniej zbuforowane), ale wymaganie zmian w sysopku jest be.

Czemu biblioteka ULT musi kompensować brak wsparcia jądra z użyciem opakowań funkcji?

ULT musi rekompensować brak wsparcia jądra w postaci opakowania funkcji, ponieważ inaczej wywołania systemowe blokowałyby cały proces, ponieważ jądro nic nie wie o jego wątkach. Dlatego np. w Linux można najpierw sprawdzić, czy syscall, który chcemy wykonać, zablokuje. Można to robić za pomocą funkcji select.

opakowanie (wrapping) – opakowanie wywsysa funkcją biblioteczną, która sprawdza, czy można je bezpiecznie (bez zablokowania procesu) wykonać (ewentualnie zmienia wątek, zamiast wykonać wywołanie).

Jakie zdarzenia wymuszają przełączanie kontekstu między włóknami?

Inny problem z pakietami obsługi wątków na poziomie użytkownika: jeśli wątek zacznie działać, to żaden inny wątek w tym procesie nie zacznie działać, dopóki pierwszy wątek nie zrezygnuje dobrowolnie z procesora. W obrębie pojedynczego procesu nie ma przerwań zegara. Rozwiązanie: zlecenie środowisku wykonawczemu żądania sygnału zegara (przerwania) co sekundę w celu przekazania mu kontroli. Takie rozwiązanie jest toporne i trudne do zaprogramowania. Okresowe przerwania z wyższą częstotliwością nie zawsze są możliwe, a jak już są, to są drogie obliczeniowo.

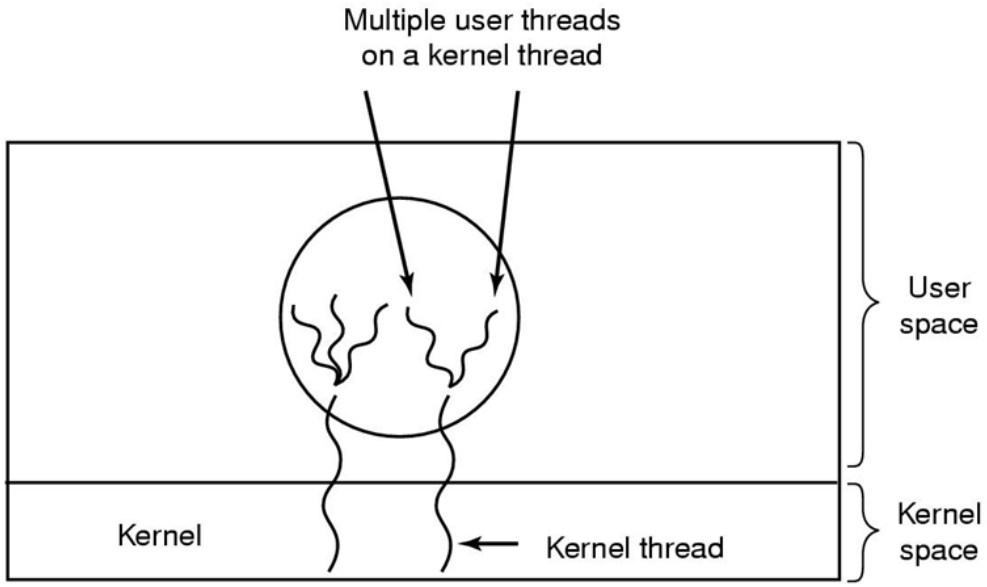
Co się dzieje w przypadku, kiedy włókno spowoduje błąd strony?

braki stron pamięci (page faults) – występuje, gdy program wywoła lub skoczy do instrukcji, której nie ma w pamięci. Wtedy sysopek jest zmuszony do pobrania brakującej instrukcji wraz z jej sąsiadami z dysku.
Podczas gdy potrzebna instrukcja jest wyszukiwana i wczytywana, proces pozostaje zablokowany. Jeśli wątek spowoduje warunek braku strony, jądro, które nawet nie wie o istnieniu wątków, blokuje cały proces do czasu zakończenia dyskowej operacji IO. Robi to, mimo że nie ma przeszkód, by inne wątki działały.

Procedury z pakietu PThreads

- pthread_create – utworzenie nowego wątku
- pthread_exit – zakończenie wątku wywołującego
- pthread_join – wywołuje wątek oczekujący na zakończenie pracy innego
- pthread_yield – zwolnienie procesora w celu umożliwienia działania innemu wątkowi
- pthread_attr_init – utworzenie i zainicjowanie struktury atrybutów wątku
- pthread_attr_destroy – usunięcie struktury atrybutów wątku

L11.Z4. Opisz hybrydowy model wątków bazujący na **aktywacjach planisty** i pokaż, że może on łączyć zalety KLT i ULT. Posługując się artykułem *An Implementation of Scheduler Activations on the NetBSD Operating System* wyjaśnij, jakie **wezwania** (ang. *upcall*) dostanie planista przestrzeni użytkownika gdy: zwiększymy lub zmniejszymy liczbę **wirtualnych procesorów**, wątek zostanie zablokowany lub odblokowany w jądrze, proces otrzyma sygnał.



Multiplexing user-level threads onto kernel-level threads.

model hybrydowy

- jednym ze sposobów połączenia zalet zarządzania wątkami na poziomie usera i jądra jest użycie wątków na poziomie jądra, a następnie zwielokrotnienie niektórych lub wszystkich wątków jądra na wątki na poziomie użytkownika
- wątki poziomu usera po kolei korzystają z wątku poziomu jądra
- programista może określić, ile wątków jądra chce wykorzystać oraz na ile wątków poziomu usera ma być zwielokrotniony każdy z nich – elastyczność
- jądro jest świadome istnienia wyłącznie wątków poziomu jądra i tylko nimi zarządza
- niektóre spośród tych wątków mogą zawierać wiele wątków poziomu usera, stworzonych na bazie wątków jądra
- wątki poziomu usera są tworzone, niszczone i zarządzane identycznie, jak wątki na poziomie usera działające w systemie operacyjnym bez obsługi wielowątkowości

aktywacje planisty – sposób będący próbą poprawy prędkości zarządzania wątkami na poziomie jądra bez konieczności rezygnacji z ich innych zalet

- cel: naśladowanie funkcji wątków jądra, ale z zapewnieniem lepszej wydajności i elastyczności (to cechy, które charakteryzują pakiety zarządzania wątkami zaimplementowane w user space)
    - wątki usera nie powinny wykonywać specjalnych, nieblokujących wywołań systemowych lub sprawdzać wcześniej, czy wykonanie określonych wywsysów jest bezpieczne
    - kiedy wątek zablokuje się na wywsysie lub page faulcie, powinien mieć możliwość uruchomienia innego wątku w ramach tego samego procesu, jeśli jakiś jest gotowy do działania
- uniknięcie niepotrzebnych przejść pomiędzy user space a kernel space – jeśli np. wątek zablokuje się w oczekiwaniu na to, aż inny wątek wykona działania, nie ma powodu informowania o tym jądra. Środowisko wykonawcze user space'u może samodzielnie zablokować wątek synchronizujący i zainicjować nowy.
- kiedy ten mechanizm jest wykorzystywany, jądro przypisuje określoną liczbę procesorów wirtualnych do każdego procesu i umożliwia środowisku wykonawczemu przestrzeni użytkownika na przydzielanie wątków do procesorów. Mechanizm ten może być również wykorzystywany w systemie wieloprocesorowym, w którym zamiast procesorów wirtualnych są procesory fizyczne. Liczba procesorów wirtualnych przydzielonych do procesu zazwyczaj wynosi jeden, ale proces może poprosić o więcej, a także zwrócić procesory, których już nie potrzebuje.

wezwania (upcall) – (podstawowa zasada działania aktywacji planisty) jeśli jądro dowie się o blokadzie wątku (np. z powodu uruchomienia blokującego wywsysa lub page fault) to powiadamia o tym środowisko wykonawcze procesu. W tym celu przekazuje na stos w postaci parametrów numer zablokowanego wątku oraz opis zdarzenia, które wystąpiło. Powiadomienie może być zrealizowane dzięki temu, że jądro uaktywnia środowisko wykonawcze znajdujące się pod znanym adresem początkowym. Jest to mechanizm w przybliżeniu analogiczny do sygnałów na Uniksie.

Upcalls are the interface used by the scheduler activations system in the kernel to inform an application of a scheduling-related event. An application that makes use of scheduler activations registers a procedure to handle the upcall, much like registering a signal handler. When an event occurs, the kernel will take a processor allocated to the application (possibly preempting another part of the application), switch to user level, and call the registered procedure.

To perform an upcall, the kernel allocates a new virtual processor for the application and begins running a piece of application code in this new execution context. The application code is known as the upcall handler, and it is invoked similarly to a traditional signal handler. The upcall handler is passed information that identifies the virtual processor that stopped running and the reason that it stopped. The upcall handler can then perform any user-level thread bookkeeping and then switch to a runnable thread from the application's thread queue.

procesory wirtualne – virtual processors are mapped to available logical processors in the physical computer and are scheduled by the Hypervisor software to allow you to have more virtual processors than you have logical processors

vCPU na którym wątek się zablokował jest nieaktywny i kernel to wtedy zapamiętuje - chodzi natomiasto o to, żeby liczba aktywnych vCPU była wciąż taka sama, więc kernel dodaje procesowi dodatkowy vCPU w "zamian" za ten który jest zablokowany. Gdy dany wątek już się odblokuje, to wątek który się w danym czasie wykonywał jest wywłaszczony i upcall handler w tym vCPU decyduje który wątek uruchomić (w szczególności może przywrócić wywłaszczony wątek) i "usuwa" dodatkowy vCPU, więc sumarycznie liczba vCPU pozostaje taka sama.

Jakie wezwania dostanie planista, gdy

- zwiększymy lub zmniejszymy liczbę wirtualnych procesorów
    - SA_UPCALL_NEWPROC - notifies the process of a new processor allocation
    - SA_UPCALL_PREEMPTED notifies the process of a reduction in its processor allocation
- wątek zostanie zablokowany lub odblokowany w jądrze
    - A_UPCALL_BLOCKED notifies the process that an activation has blocked in the kernel
    - SA_UPCALL_UNBLOCKED this upcall notifies the process that an activation which previously blocked
- proces otrzyma sygnał
    - SA_UPCALL_SIGNAL -this upcall is used to deliver a POSIX-style signal to the process. If the signal is a synchronous trap, then event is 1, and sas points to the activation which triggered the trap. For asynchronous signals, event is 0. The arg parameter points to a siginfo_t structure that describes the signal being delivered.

L11.Z5. Wyjaśnij, jak przy pomocy **multipleksowania wejścia-wyjścia** (ang. *I/O multiplexing*) i **nieblokujących** wywołań systemowych zbudować jednowątkowy serwer `TCP` obsługujący współbieżnie wiele połączeń sieciowych. Zapoznaj się notatkami do §6 książki „*Unix Network programming, Volume 1*”. Przeanalizuj kod serwera usługi `echo` wykorzystującego wywołanie `poll(2)`. Opisz znaczenie flag zawartych w polach «`events`» i «`revents`» struktury `pollfd`.

http://www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch06lev1sec2.html

Jak działa ten serwerek z linka?
https://github.com/shichao-an/unpv13e/blob/master/tcpcliserv/tcpservpoll01.c
1. Inicjalizacja zmiennych
    a. Szczególnie tablicy client[] do której na początku dodany jest deskryptor listenera -> stąd będzie wiadomo czy ktoś nowy się połączył
2. W nieskończonej pętli polling eventów
    a. Sprawdzanie czy nie ma nowego klienta. Jeśli jest to dodanie deskryptora jego gniazda do tablicy klientów.
    b. Sprawdzenie w pętli dla każdego klienta czy przesłał nam jakieś dane.
        i. Jeśli tak, to następuje synchroniczny read a potem synchroniczny write do socketa.
    c. Obsługa błędów
        i. Przerwanie połączenia

L11.Z6. Najpowszechniej implementowane wątki przestrzeni jądra wprowadzają do programów dodatkowy stopień złożoności. Co dziwnego może się wydarzyć w wielowątkowym procesie uniksowym gdy:

- wołamy funkcję fork, aby utworzyć podproces
  fork klonuje tylko jeden wątek (ten, który go wywołał), a ignoruje pozostałe. Cała pamięć zostanie przekopiowana, w tym blokady (mutex, semafor) używane tylko przez pozostałe wątki, co może prowadzić do wycieków pamięci lub zablokowania wątku na którejś blokadzie – blokad nie da się już zwolnić, a dzielone dane (np. malokowa sterta) mogą być uszkodzone.

  pthread_atfork – niech wątek, który chce zrobić fork, założy wszystkie mutexy w każdej sekcji krytycznej w każdej bibliotece. Następnie po forku pierwotny wątek i skopiowany wątek zwolnią wszystkie mutexy w swoich przestrzeniach adresowych. Problemy: no ale czekanie na to, aż uda nam się złapać wszystkie mutexy, może trwać długo. Możemy któryś pominąć. Podczas tej próby może wystąpić deadlock.

- użytkownik wciska kombinację «CTRL+C» i w rezultacie jądro wysyła «SIGINT» do procesu
  signal mask – sygnał może być zablokowany, co znaczy że nie będzie dostarczony, zanim go nie odblokujemy. Każdy wątek w procesie ma swoją niezależną maskę sygnałów, która jest zbiorem aktualnie blokowanych sygnałów.

  Każdy wątek ma swoją własną maskę sygnałów, ale handler sygnału jest per proces. Sygnał trafi do któregoś wątku (arbitralnie wybranego), tylko raz. Jeśli któryś wątek ustawi handler dla SIGINT, to może on zostać wywołany w kontekście któregokolwiek wątku. Jeśli jakiś wątek uzna, że ignorujemy dany sygnał, inny może cofnąć ten wybór poprzez przywrócenie poprzednich ustawień.

- czytamy w wielu wątkach ze zwykłego pliku korzystając z jednego deskryptora pliku
  Deskryptor pliku jest dzielony przez wątki. Jeden wątek może zamknąć plik, z którego korzysta inny. Może zamienić wskaźnik (lseek) na miejsce w tym pliku. Inny problem: w środowisku równoległym oba wątki mogą próbować pisać w tym samym miejscu jednocześnie.

  rozwiązanie:
  pread, pwrite - czytanie/pisanie w pliku na podanym offsecie. Pozwalają wielu wątkom wykonywanie IO na tym samym pliku bez wpływu na zmiany w przesunięciu pliku przez inne wątki.

- modyfikujemy zmienną środowiskową funkcją putenv
  putenv – zmienia lub dodaje zmienną środowiskową. Nie musi być reentrant.

  ENV to tablica wskaźników. putenv tylko dodaje wskaźnik do tej tablicy. Jak nadpiszemy miejsce na które pokazywał nasz nowy env, to słabo. Jednoczesne wywołanie getenv (niekoniecznie przez nas, być może przez jakieś funkcje biblioteczne) i putenv może powodować segfault.

- wątki intensywnie korzystają z menadżera pamięci z użyciem procedury malloc
  Gdy różne wątki alokują często na zamianę małe kawałki pamięci, malloc upycha je obok siebie, a zatem w cache będą siedziały w jednej linii. Następnie, jeżeli dwa wątki równolegle (na dwóch CPU) korzystają ze swojej pamięci, to mamy false sharing:

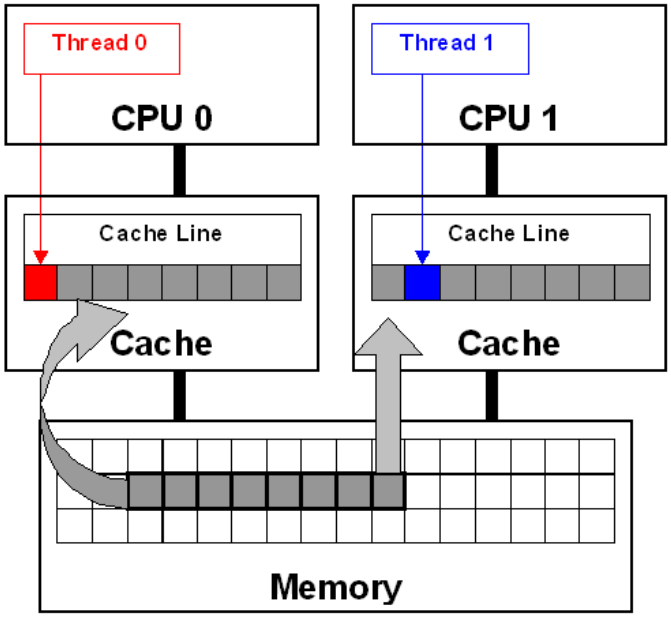
  Polega to na tym, że każdy zapis do cache przez jeden wątek/procesor, powoduje inwalidację tej linii dla pozostałych (pomimo że tak naprawdę używają rozłącznych kawałków).

  In symmetric multiprocessor (SMP) systems, each processor has a local cache. The memory system must guarantee cache coherence. False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces an update, which hurts performance.

  False sharing is a well-known performance issue on SMP systems, where each processor has a local cache. It occurs when threads on different processors modify variables that reside on the same cache line, as illustrated in Figure 1. This circumstance is called false sharing because each thread is not actually sharing access to the same variable. Access to the same variable, or true sharing, would require programmatic synchronization constructs to ensure ordered data access.

```
01 double sum=0.0, sum_local[NUM_THREADS];
02 #pragma omp parallel num_threads(NUM_THREADS)
03 {
04  int me = omp_get_thread_num();
05  sum_local[me] = 0.0;
06
07  #pragma omp for
08  for (i = 0; i < N; i++)
09  sum_local[me] += x[i] * y[i];
10
11  #pragma omp atomic
12  sum += sum_local[me];
13 }
```

There is a potential for false sharing on array sum_local. This array is dimensioned according to the number of threads and is small enough to fit in a single cache line. When executed in parallel, the threads modify different, but adjacent, elements of sum_local (the source line shown in red), which invalidates the cache line for all processors.



L11.Z7. Wątki nie są panaceum na problemy z wydajnością oprogramowania na **maszynach wieloprocesorowych ze współdzieloną pamięcią** (ang. *Shared Memory Processors*). Wymień warunki jakie musi spełniać architektura programu, by stosowanie wątków było uzasadnione (§4.3)? Co ogranicza wydajność programów używających wątków? Jakie problemy z efektywnym użyciem wątków napotkali twórcy silnika gry Half-Life 2 i jak je rozwiązali?