L1.Z1. Wyjaśnij różnice między **powłoką** (ang. *shell*), **systemem operacyjnym** i **jądrem systemu operacyjnego** (ang. *kernel*). W tym celu dobierz kilka przykładów powszechnie wykorzystywanego oprogramowania. Jakie są główne zadania systemu operacyjnego z punktu widzenia programisty?

powłoka - interpreter poleceń użytkownika kierowanych do kernela

- jest podstawowym interfejsem pomiędzy userem a sysopkiem
- powłoka, bo jest nakładką na kernel
- nie jest częścią systemu operacyjnego
- powłoka wykorzystuje emulator terminala jako std IO
- przykłady powłok linuksowych: sh shell i bash shell (domyślna powłoka w niektórych dystrybucjach linuksa)
- dwa rodzaje:
    - powłoka tekstowa - interpreter poleceń uruchamiany w trybie tekstowym, potocznie zwany konsolą
    - powłoka graficzna - ma zwykle postać menadżera plików, kontrolowana za pomocą myszy. Eksplorator – domyślna powłoka systemu MS Windows

system operacyjny - oprogramowanie działające w trybie jądra
SO spełniają dwie niepowiązane ze sobą funkcje:

- dostarczają programistom aplikacji (i programom aplikacyjnym) czytelnego, abstrakcyjnego zbioru zasobów będących odpowiednikami sprzętu
- zarządzają tymi zasobami sprzętowymi
    - pamięć (fizyczna, tablica stron)
    - procesor (translacja adresów, przerwania, cache, TLB)
    - urządzenia IO
    - kanały DMA

Sysopek jako rozszerzona maszyna:
Architektura większości komputerów na poziomie języka maszynowego (zestaw instrukcji, organizacja pamięci, IO i struktura magistral) jest prymitywna i niewygodna do programowania. Jednym z głównych zadań systemu operacyjnego jest ukrywanie sprzętu i dostarczanie programom (programistom) wygodnych, czytelnych, eleganckich i spójnych abstrakcji do wykorzystania (np. wykonywanie operacji IO na dysku – nie trzeba na poziomie sprzętowym, system operacyjny zawiera sterowniki dysku, które zapewniają interfejs do odczytu i zapisu bloków na dysku bez wchodzenia w szczegóły).

Prawdziwymi klientami systemu operacyjnego są programy aplikacyjne. Użytkownicy posługują się abstrakcjami dostarczanymi przez interfejs użytkownika – powłokę wiersza polecenia lub interfejs graficzny.

Sysopek jako menedżer zasobów:
Zadaniem sysopka jest zapewnienie uporządkowanego i kontrolowanego przydziału procesorów, pamięci i urządzeń IO pomiędzy różne programy rywalizujące o te zasoby; śledzenie informacji na temat tego, które programy korzystają z jakich zasobów

jądro systemu operacyjnego (kernel)

- znajduje się bezpośrednio nad warstwą sprzętową
- udostępnia interfejs do korzystania z hardware w postaci wywołań systemowych – syscalli (wywołania te mogą być wołane z poziomu usera, kiedy wymagane są działania w kernel mode – dostęp do wszystkich zasobów)
- dostarcza środowiska uruchomieniowego dla programów

<u>Jakie są główne zadania systemu operacyjnego z punktu widzenia programisty?</u>

- zarządza pamięcią (przydzielanie jej, ochrona dostępu do jej fragmentów, pamięć wirtualna)
- zapewnia ładną abstrakcję zasobów i zarządza nimi efektywnie
- zapewnia interakcję z userem

L1.Z2. Na podstawie dokumentacji wymień składowe **pakietu** deb ze szczególnym uwzględnieniem zawartości pliku control. Porównaj zarządzanie zainstalowanym oprogramowaniem z użyciem pakietów i instalatorów znanych z systemów nieuniksowych. Weź pod uwagę proces pobierania, weryfikacji, instalacji, konfiguracji i odinstalowania oprogramowania.

pakiet - skompresowane archiwum zawierające oprogramowanie, umożliwiające jego łatwą oraz szybką instalację, aktualizację lub dezinstalację

Wymień składowe pakietu deb ze szczególnym uwzględnieniem zawartości pliku control

Przykłady plików deb można znaleźć w /var/cache/archives/.

```
dpkg-deb -I package.deb - show control
dpkg-deb -c package.deb - list files which will be installed
ar tv package.deb       - list content
ar x package.deb        - extract files
```

Zbadajmy przykładowo plik parted_1.4.24-4_i386.deb:

```
$ ar tv parted_1.4.24-4_i386.deb
rw-r--r-- 0/0      4 Mar 28 13:46 2002 debian-binary
rw-r--r-- 0/0   1386 Mar 28 13:46 2002 control.tar.gz
rw-r--r-- 0/0  39772 Mar 28 13:46 2002 data.tar.gz
```

- debian-binary – zawiera informacje o wersji formatu deb - "2.0\n"
- control.tar.gz - zawiera cztery pliki:
  - control - zawiera dodatkowe informacje o pakiecie
  - md5sums - sumy kontrolne dla każdego pliku w data.tar.gz
  - postinst, prerm - zajmują się usuwaniem starych documentation files i dodawaniem linków z doc do share/doc
- data.tar.gz - archiwum zawierające wszystkie pliki, które muszą być zainstalowane wraz z ich ścieżkami; musi być ostatnim plikiem w archiwum deb

```
$ cat control
Package: parted
Version: 1.4.24-4
Section: admin
Priority: optional
Architecture: i386
Depends: e2fsprogs (>= 1.27-2), libc6 (>= 2.2.4-4), libncurses5 (>= \
5.2.20020112a-1), libparted1.4 (>= 1.4.13+14pre1), libreadline4 (>= \
4.2a-4), libuuid1
Suggests: parted-doc
Conflicts: fsresize
Replaces: fsresize
Installed-Size: 76
Maintainer: Timshel Knoll <timshel@debian.org>
Description: blablabla
```

- pola obowiązkowe:
  - package - nazwa pakietu; po tej nazwie inne pakiety będą się odnosić do tego pakietu w dependencjach
  - version - wersja pakietu; wersja pakietu, ma określony format i porządek - określenie minimalnej wersji w dependencjach
  - maintainer - email osoby, która stworzyła pakiet lub odpowiada za jego utrzymanie (kontakt w razie problemów z pakietem)
  - description - opis pakietu

- pola rekomendowane:
  - section - „typ" pakietu, np. utils, net, mail, text, x11, etc.
  - priority – określa priorytet pakietu w stosunku do systemu jako całości, np. required, standard, optional, extra
  - essential - (tak/nie) określa, czy pakiet jest wymagany do prawidłowego działania systemu, dpkg nie pozwoli na usunięcie, jeśli pakiet jest essential
  - architecture – architektura, na którą pakiet jest zbudowany, lub „all", jeśli jest niezależny od architektury
  - suggests - inne przydatne pakiety powiązane z tym
  - homepage - strona domowa oprogramowania dostarczanego z pakietem
  - depends – pakiety, które są wymagane do zainstalowania tego pakietu
  - replaces – pakiety, które ten zastępuje; pozwala nadpisać pliki innego pakietu
  - conflicts – pakiety, z którymi ten konfliktuje, np. zawierając pliki o tej samej nazwie; menedżer pakietów nie pozwoli, żeby konfliktujące pliki zainstalowały się jednocześnie

Porównaj zarządzanie zainstalowanym oprogramowaniem z użyciem pakietów i instalatorów znanych z systemów nieuniksowych

- Unix
    - pobieranie: pakiet z repozytorium pakietów, pobieranie tylko niezainstalowane zależności
    - weryfikacja: pakiety zweryfikowane przez twórców dystrybucji. + MD5
    - instalacja: instalacja pakietu i niezainstalowanych zależności, pakiet “wie”, jak się zainstalować
    - odinstalowanie: pakiet “wie”, jak się odinstalować
- Android-style
    - pobieranie: "apka" ze sklepu, wszystkie zależności są pobierane
    - weryfikacja: apki zweryfikowane przez właściciela sklepu (Google, Amazon, Samsung itd.) + aplikacje podpisane certyfikatem + SHA-256
    - instalacja: menedżer umie zainstalować apkę
    - odinstalowanie: menedżer umie odinstalować apkę
- Windows-style
    - pobieranie: brak zcentralizowanego systemu dystrybucji, wszystkie zależności zawarte w instalatorze lub użytkownik sam musi zadbać o ich spełnienie
    - weryfikacja: programy zweryfikowane przez użytkownika
    - instalacja: użytkownik podejmuje decyzje podczas procesu instalacji
    - odinstalowanie: użytkownik uruchamia aplikacje która wie, jak odinstalować program

Źródła: składowe pakietu, szczegóły control

rys historyczny

Pierwsza generacja

- maszyny prymitywne, wykonanie najprostszych obliczeń zajmowało kilka sekund
- ta sama grupa osób zajmowała się projektowanie, budową, programowaniem, obsługą i konserwacją każdej maszyny
- programowanie wyłącznie w języku maszynowym lub poprzez tworzenie obwodów elektrycznych łączonych za pomocą tysięcy kabli na specjalnych tablicach programowych

Tryb działania programisty:

- zapisanie się na wiszącej na ścianie liście zgłoszeń w celu uzyskania bloku czasu
- pójście do pokoju z maszyną i włączenie swojej tablicy programowej do komputera

Dzięki wprowadzeniu kart perforowanych na początku lat pięćdziesiątych procedura uległa usprawnieniom – zamiast używać tablic programowych, można było teraz pisać programy na kartach i odczytywać je z nich.

Druga generacja

- wprowadzenie tranzystorów, komputery stały się mniej zawodne (wcześniej 20 tysięcy lamp elektronowych, każda mogła się spalić w trakcie obliczeń)
- podział personelu ze względu na pełnione funkcje (projektant, konstruktor, operator itd.)

Proces uruchomienia zadania:

- programista pisał program na papierze w języku Fortran lub w asemblerze, dziurkował go na kartkach
- przynosił zbiór kart do pokoju wprowadzania danych, przekazywał operatorowi
- kiedy komputer skończył wykonywanie zadania, operator szedł do drukarki, oddzierał wynik działania programu i zanosił do pokoju wyników, skąd programista go później odbierał
- operator brał jeden z zestawów kart, które zostały przyniesione do pokoju wprowadzania danych i wczytywał go
- jeśli był potrzebny kompilator Fortrana, operator brał go z szafy i wczytywał do komputera

Obserwacja: dużo czasu marnotrawiono na chodzenie po pokoju komputerowym

Systemy wsadowe:

- IBM 1401 – czytanie kart, kopiowanie taśm, drukowanie wyników; IBM 7094 – obliczenia
- programista przynosi karty do 1401 (czytnik kart)
- 1401 wczytuje plik zadań na taśmę magnetyczną, którą trzeba było przenieść do pokoju komputerowego
- tam taśmę montowano w napędzie taśm
- operator ładował specjalny program („sysopek”), który odczytywał pierwsze zadanie z taśmy i je uruchamiał, 7094 wykonuje obliczenia, po zakończeniu każdego z zadań sysopek automatycznie wczytywał następne zadanie z taśmy i zaczynał je uruchamiać
- zamiast drukowania wynik był zapisywany na drugiej taśmie
- po zakończeniu przetwarzania wsadu operator wyjmował taśmy wejściową i wyjściową, wymieniał taśmę wejściową na następny wsad i przynosił taśmę wyjściową do komputera 1401 w celu wydrukowania go

Trzecia generacja

- problem: na początku lat 60 większość producentów utrzymywała po dwie, niezgodne ze sobą linie produktów: komputery naukowe (takie jak 7094) oraz komputery znakowe (takie jak 1401)
- droga sprawa, poza tym użytkownicy chcieli dużej maszyny, która byłaby zdolna do uruchamiania ich starych programów, tyle że szybciej

- IBM próbuje rozwiązać ten problem, produkuje serię System/360 – pierwsza linia komputerów, w której zastosowano układy scalone
- problem: największa zaleta tych komputerów to jednocześnie największa wada – komputery musiały być wydajne w wielu różnych zastosowaniach; dużo kolidujących ze sobą wymagań – wielki kod, dużo błędów, dużo nowych wersji poprawiających stare (ale też zawierające błędy)
- OS/360 i mu podobne spopularyzowały wieloprogramowość, której nie było w drugiej generacji

L1.Z3. Czym jest **zadanie** w **systemach wsadowych**? Jaką rolę pełni **monitor**? Na czym polega **planowanie zadań**? Zapoznaj się z rozdziałem „*System Supervisor*" dokumentu IBM 7090/7094 IBSYS Operating System. Wyjaśnij znaczenie poleceń **języka kontroli zadań** (ang. *Job Control Language*) użytych na rysunku 3 na stronie 13. Do jakich zastosowań używa się dziś systemów wsadowych?
Wskazówka: Bardzo popularnym systemem realizującym szeregowanie zadań wsadowych jest SLURM .

Czym jest zadanie w systemach wsadowych? Jaką rolę pełni monitor? Na czym polega planowanie zadań?

zadanie - program lub zbiór programów, który dla danych wejściowych produkował dane wyjściowe (rozp. Od karty $JOB)

system wsadowy – idea ich działania polegała na pobraniu pełnego zasobnika zadań w pokoju wprowadzania danych i zapisaniu ich na taśmie magnetycznej za pomocą mniejszego komputera (IBM 1401 czytał karty, kopiował taśmy i drukował wyniki, ale nie nadawał się do obliczeń). Do obliczeń wykorzystywano większe i droższe komputery, np. IBM 7094.

monitor – protoplasta systemu operacyjnego; odczytywał kolejne zadania z taśmy wejściowej (interpretował control language) i je uruchamiał (eliminowało to konieczność obsługi pojedynczych programów przez człowieka). W monitorze były sterowniki, IO, zegar systemowy, ochrona pamięci.

planowanie zadań – ustalanie kolejności wykonywanych zadań. Na początku do komputera wrzucało się „paczkę z zadaniami", a jego planowanie to było zwykłe FIFO. Potem wprowadzono sortowanie np. na podstawie priorytetów zapisanych w JCL.

Wyjaśnij znaczenie poleceń języka kontroli zadań

język kontroli zadań – prymitywny język poleceń dla monitora opisujący zadanie
- $JOB - nowe zadanie, określa czas działania w minutach
- $FORTRAN – załadowanie kompilatora języka Fortran z taśmy systemowej, bezpośrednio za nią następował program do skompilowania
- $LOAD - załaduj kod z taśmy
- $RUN - uruchom program
- $END - zakończ zadanie

- $IBSYS - System Supervisor jest wołany do pamięci rdzenia (core storage) i otrzymuje kontrolę
- $RELEASE – causes the unit assigned to the specified system unit function to be released from the function
- $IBEDT – System Supervisor woła System Editor do core storage z System Library Unit i przekazuje mu kontrolę
- $EXECUTE – precyzuje, który subsystem ma przeczytać i zinterpretować następne karty (po karcie execute)
- $STOP - określa koniec stosu zadań

Do jakich zastosowań używa się dziś systemów wsadowych?

Wiele dzisiejszych systemów wsadowych automatyzuje zadania, dzięki czemu interakcja z człowiekiem nie jest wymagana, chyba że coś pójdzie nie tak. Przykłady:
- payroll system: systemy wsadowe są idealne do tworzenia listy płac pracowników, wykonywanie obliczeń potrzebnych do wypłat pracownikom
- zarzadzanie ogromnymi bazami danych (np. w systemach rezerwacji miejsc lotniczych)
- w bankach transakcje o danej godzinie – są zbierane te wsady transakcji
- SLURM (dawniej *Simple Linux Utiliy for Resource Management*) to system kolejkowy (resource manager, job scheduler) działający np. na maszynach w ICM (centrum obliczeniowe UW) służących do obliczeń naukowych na komputerach dużej mocy

Źródła: slurm

L1.Z4. Jaka była motywacja do wprowadzenia **wieloprogramowych** systemów wsadowych? W jaki sposób wieloprogramowe systemy wsadowe wyewoluowały w systemy z **podziałem czasu** (ang. *time-sharing*)? Podaj przykład systemu **interaktywnego**, który nie jest wieloprogramowy.

wieloprogramowe systemy wsadowe – rozwiązanie polegające na podzieleniu pamięci na kilka części i umieszczeniu w każdej z nich osobnego zadania. Podczas gdy jedno zadanie oczekiwało na zakończenie operacji IO, drugie mogło korzystać z procesora

systemy z podziałem czasu – odmiana systemów wieloprogramowych, w których każdy użytkownik posiadał podłączony do komputera terminal, który zapewniał użytkownikom interaktywną obsługę komputera. Podobnie jak system wieloprogramowy system ten może wykonywać wiele zadań jednocześnie. Istotną różnica między tymi systemami jest to, że system z podziałem czasu nie czeka na przestój jakiegoś programu, aby uruchomić inny, ale uruchamia wiele zadań jednocześnie. My - użytkownicy - mamy wrażenie, że zadania te wykonują się jednocześnie, ale jest to tylko nasze złudzenie. Tak naprawdę procesor przełącza się pomiędzy poszczególnymi zadaniami, ale przełączenia te następują tak szybko, że użytkownicy mogą na bieżąco współpracować z wszystkimi zadaniami w trybie rzeczywistym.

system interaktywny – pozwala wpływać użytkownikowi na SO podczas pracy innego programu

Jaka była motywacja do wprowadzenia wieloprogramowych systemów wsadowych?

- W komputerach 7094, w których bieżące zadanie było wstrzymywane do czasu zakończenia operacji z taśmą lub innej operacji IO, procesor główny pozostawał bezczynny do momentu zakończenia tej operacji.
- W przypadku obliczeń naukowych intensywnie wykorzystujących procesor operacje IO nie są częste, zatem nie powodowało to znaczącego marnotrawstwa czasu. W przypadku komercyjnego przetwarzania danych czas oczekiwania związany z IO wynosił często 80-90% całkowitego czasu, więc trzeba było coś z tym zrobić, by uniknąć tak wielkiego stopnia bezczynności drogiego procesora głównego.
- Pojawiło się rozwiązanie polegające na podzieleniu pamięci na kilka części i umieszczeniu w każdej z nich osobnego zadania. Podczas gdy jedno zadanie oczekiwało na zakończenie operacji IO, drugie mogło korzystać z procesora.
- Bezpieczne przechowywanie w pamięci wielu zadań naraz wymagało specjalnego sprzętu, który chroniłby każde z zadań przed „szpiegowaniem" oraz modyfikowaniem jednego zadania przez drugie, ale komputery z serii 360 oraz inne systemy z trzeciej generacji były wyposażone w taki sprzęt.

W jaki sposób wieloprogramowe systemy wsadowe wyewoluowały w systemy z podziałem czasu (ang. *time-sharing*)?

- Czas pomiędzy złożeniem zadania a otrzymaniem wyników często wynosił kilka godzin. Roszczeniowi programiści chcieli mieć szybciej te wyniki i możliwość debugowania programów.
- Z potrzeby szybkiej odpowiedzi powstała technika podziału czasu – odmiany systemów wieloprogramowych, w których każdy użytkownik posiadał podłączony do komputera terminal.
- Komputer zapewniał szybką, interaktywną obsługę wielu użytkownikom, a także pracę nad złożonymi zadaniami wsadowymi w tle w czasie, w którym w systemach bez podziału czasu procesor był bezczynny.
- Pierwszy komputer z podziałem czasu CTSS zbudowano w MIT.
- Generalnie idea: do przestojów na I/O dochodzą jeszcze przestoje na interakcję usera, który to robił tę interakcję przez terminal.

Podaj przykład systemu interaktywnego, który nie jest wieloprogramowy.

- MS-DOS.
- CP/M na Intel 8080.

L1.Z5. Bardzo ważną zasadą przy projektowaniu oprogramowania, w tym systemów operacyjnych, jest rozdzielenie **mechanizmu** od **polityki**. Wyjaśnij te pojęcia, odnosząc się do powszechnie występujących rozwiązań, np. otwieranie drzwi klasycznym kluczem versus kartą magnetyczną.

polityka – zestaw reguł, które system ma przestrzegać, np. polityka zamykania drzwi. Pytanie: jak zaaranżować elementy?
mechanizm – praktyczna realizacja polityki, np. instalowanie zamków w drzwiach i otwieranie ich kluczem. Pytanie: jak można użyć elementów?

Dla każdej polityki mogą istnieć różne mechanizmy, np. nie tylko kluczem można otworzyć drzwi – także kartą magnetyczną. Separacja mechanizmu od polityki – mechanizm nie powinien narzucać polityki.

Otwieranie drzwi kartą:
Polityka – karta z odpowiednim kodem może otworzyć drzwi.
Mechanizm - zdefiniowane kody otwierają drzwi.

- mechanizm w tym przykładzie nie narzuca polityki, nie definiuje, kto i kiedy ma wchodzić
- łatwa zmiana polityki (wprowadzenie nowych polityk – grupy, klasy dostępu), wymaga zmiany parametrów mechanizmu

Otwieranie drzwi kluczem:
Polityka – drzwi otwierane są przez pasujący klucz.
Mechanizm – drzwi posiadają zamek, który może być otwarty tylko przez pasujący klucz.

- jeśli chcesz zmienić osobę, która może otworzyć drzwi, musisz zmienić zamki i klucze - przy zmianie polityki trzeba zmienić też mechanizm

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	

L4.Z1. Czym różni się **przetwarzanie równoległe** (ang. *parallel*) od **przetwarzania współbieżnego** (ang. *concurrent*)? Czym charakteryzują się **procedury wielobieżne** (ang. *reentrant*)? Podaj przykład a) procedury wielobieżnej, ale nie **wątkowo-bezpiecznej** (ang. *thread-safe*), b) na odwrót. Czy w jednowątkowym procesie uniksowym może wystąpić współbieżność?

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
}
```

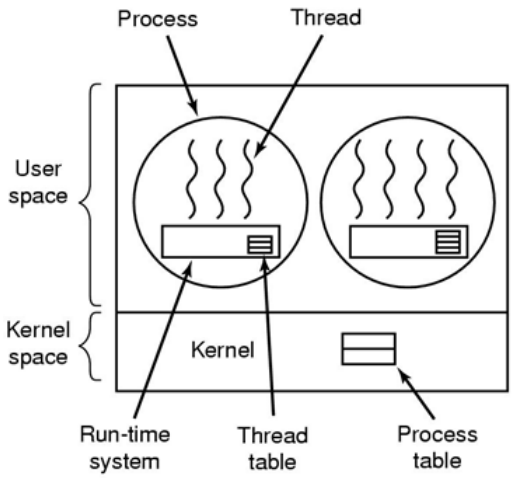
po wywołaniu swap zmienna t wraca do poprzedniego stanu (więc można w trakcie swap wywołać jeszcze jedno swap, które przywróci stary kontekst po swoim zakończeniu) ale nie jest thread-safe, bo inny nasz wątek może zostać wywłaszczony, kolejny wątek zacznie wykonywać swap, zmieni zmienną t, a następnie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

Example 4: thread-safe, reentrant

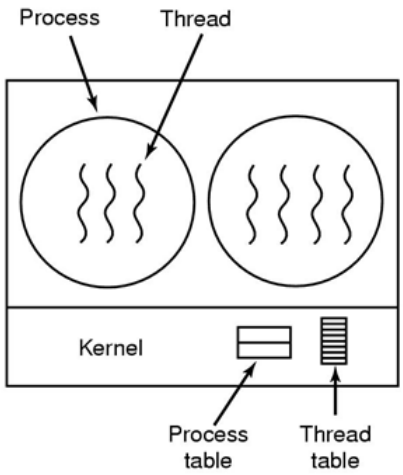
```
/* We use a local variable: the function is now thread-safe and reentrant, we have ascended to higher
plane of existence.  */

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

L4.Z3. Wymień przewagi wątków przestrzeni jądra (ang. *kernel-level threads*) nad **wątkami przestrzeni użytkownika** (ang. *user-level threads*) zwanych dalej włóknami. Czemu biblioteka ULT musi kompensować brak wsparcia jądra z użyciem **opakowań funkcji** (ang. *wrapper*)? Jakie zdarzenia wymuszają przełączanie kontekstu między włóknami? Co się dzieje 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.

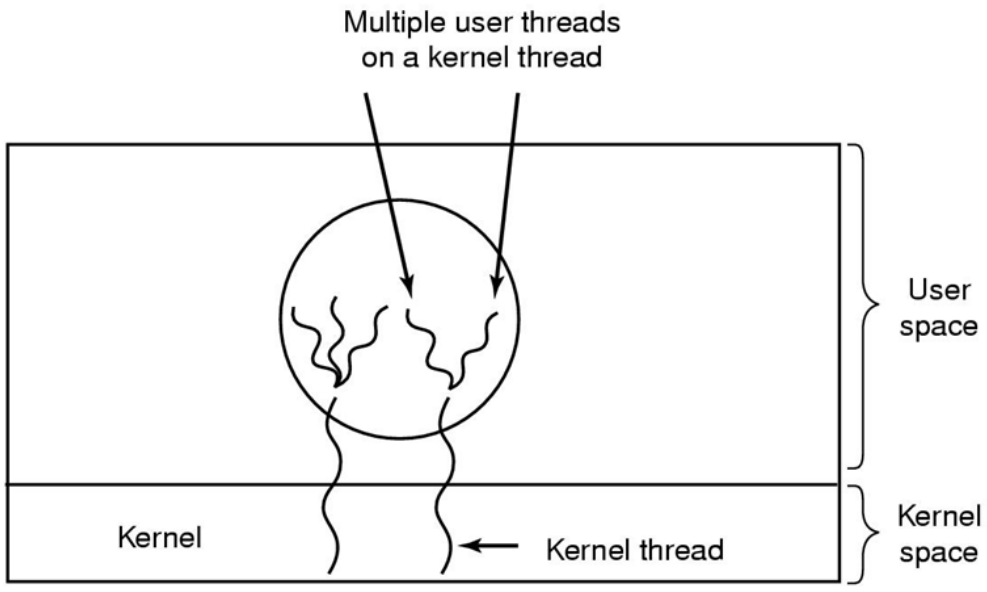
Co się dzieje w przypadku, kiedy włókno spowoduje błąd strony?

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

L4.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.

L4.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.

L12.Z1. Wyjaśnij różnice między **zakleszczeniem** (ang. *deadlock*), **uwięzieniem** (ang. livelock) i **głodzeniem** (ang. *starvation*). W podręcznikach pojęcia te odnoszą się do **zasobów**. Pokaż, że podobne problemy występują w przypadku **przesyłania komunikatów.**

zasoby – wszystko to, co trzeba uzyskać, wykorzystać i zwolnić, np. urządzenia, rekordy danych, pliki itp.
- zasób z wywłaszczaniem – można odebrać procesowi korzystającemu z niego bez skutków ubocznych, np. pamięć
- zasób bez możliwości wywłaszczania – nie można go zabrać bieżącemu właścicielowi bez szkody dla obliczeń, np. nagrywarki Blu-ray

zakleszczenie (*deadlock*) **–** gdy każdy proces jest zablokowany, bo czeka na zdarzenie od innego zablokowanego procesu (np. na zwolnienie jakiegoś zasobu). Żadne ze zdarzeń nigdy nie nastąpi, więc zakleszczenie jest trwałe.

Przykład:
- korki w mieście
- problem ucztujących filozofów: pięciu filozofów siedzi przy stole i każdy wykonuje jedną z czynności: albo je, albo rozmyśla. Stół jest okrągły, przed każdym z nich znajduje się miska ze spaghetti, a pomiędzy każdą sąsiadującą parą filozofów leży widelec, a więc każda osoba ma przy sobie dwie sztuki – po swojej lewej i

prawej stronie. Do zjedzenia potrzebne są dwa widelce. Ryzyko zakleszczenia, gdy każdy z nich zabierze lewy widelec i będzie czekał na prawy (lub na odwrót)

uwięzienie (ang. *livelock*) **–** sytuacja, w której dwa procesy zmieniają swój stan w odpowiedzi na zmianę tego drugiego bez wykonania postępu w pracy. Podobne do zakleszczenia, bo nie ma żadnego progresu, ale różni się tym, że procesy nie są blokowane i nie czekają na żadne zasoby.

Przykład: dwoje małżonków je obiad, ale mają jedną łyżkę. Każde jest bardzo miłe i daje drugiemu łyżkę, jeśli ta osoba jeszcze nie jadła.

głodzenie (ang. *starvation*) **–** sytuacja, w której dany proces jest pomijany przez planistę i mimo że ma możliwość działania, to nie ma dostępu do procesora lub współdzielonego zasobu.

Przykład: w systemie, w którym jeden z procesów ma do wydrukowania duży plik za każdym razem, gdy drukarka będzie wolna, system będzie próbował znaleźć proces, który ma najmniejszy plik do wydrukowania. W przypadku ciągłego napływu procesów z małymi plikami, procesom chcącym wydrukować duży plik nigdy nie zostanie przydzielona drukarka. Zagłodzą się na śmierć – będą w nieskończoność odraczane, mimo że nie są zablokowane.

komunikat – zawiera nagłówek identyfikujący proces wysyłający i odbierający oraz dane.
przesyłanie komunikatów – pomiędzy dwoma procesami o rozłącznych przadrach obejmuje ich kopiowanie z pamięci do pamięci; gdy między wątkami, to przadr jest wspólny

zakleszczenie komunikacyjne

- do innego rodzaju zakleszczeń dochodzi w systemach komunikacyjnych (np. sieciach), w których dwa lub więcej procesy komunikują się ze sobą poprzez przesyłanie komunikatów. W popularnym układzie proces A wysyła komunikat z żądaniem do procesu B, a następnie blokuje się do czasu, aż proces B zwróci komunikat z odpowiedzią. Przypuśćmy, że komunikat z żądaniem został utracony. Proces A blokuje się w oczekiwaniu na odpowiedź. Proces B blokuje się w oczekiwaniu na żądanie wykonania jakiejś operacji. Mamy zakleszczenie.
- przeciwdziałanie:
    - nie da się przeciwdziałać poprzez żądanie zasobów (ponieważ ich nie ma), nie da się także uniknąć ich przez uważne szeregowanie (nie ma takich momentów w czasie, kiedy można by opóźnić żądanie)
    - inna technika: limity czasu (ang. *timeouts*). Za każdym razem, gdy wysyłany jest komunikat wymagający odpowiedzi, jednocześnie uruchamia się licznik czasu. Jeśli licznik czasu dojdzie do zera, zanim nadejdzie odpowiedź, nadawca komunikatu zakłada, że komunikat został utracony i wysyła go ponownie.
    - jeśli pierwszy komunikat nie został utracony, a jedynie odpowiedź jest opóźniona, to zamierzony odbiorca może otrzymać komunikat dwa lub więcej razy, a konsekwencje mogą być niepożądane (np. system bankowości elektronicznej i komunikat zawierający instrukcje dokonania płatności

L12.Z2. Wymień cztery warunki konieczne do zaistnienia zakleszczenia. W jaki sposób programista może **przeciwdziałać** zakleszczeniom (ang. *deadlock prevention*)? Których z proponowanych rozwiązań nie implementuje się w praktyce i dlaczego?

Warunki konieczne do zaistnienia zakleszczenia
1. wzajemne wykluczenie – w danym czasie tylko jeden proces może korzystać z danego zasobu
2. wstrzymanie i oczekiwanie (*hold and wait*) – proces może trzymać dostęp do zasobów, czekając na otrzymanie dostępu do innych zasobów
3. brak wywłaszczania (*no preemption*) – zasoby przydzielone wcześniej nie mogą być przymusowo zabrane procesom. Muszą być jawnie zwolnione przez proces, który jest ich właścicielem.
4. cykliczne czekanie (*circural wait*) – łańcuch procesów, w którym każdy proces trzyma co najmniej jeden zasób potrzebny innemu procesowi

1-3: *może* zdarzyć się deadlock
1-4: *jest* deadlock

Zapobieganie zakleszczeniom (*deadlock prevention*)
1. wzajemne wykluczenie – w przypadku danych można nadać im status tylko do odczytu, tak aby procesy mogły jednocześnie z nich korzystać. Zapis zawsze musi być dokonywany tylko przez jeden proces.

2. trzymaj i czekaj (*hold and wait*) – wymuszenie na procesach, aby zamawiały zasoby tylko wtedy, gdy nie są w posiadaniu innych zasobów. Średnio w porządku, bo wiele procesów przed rozpoczęciem działania nie wie, ile i jakich zasobów będzie potrzebować. Dwa sposoby:
    a. wymaganie od procesu żądania wszystkich zasobów przed rozpoczęciem wykonywania. Jeśli wszystko jest dostępne, proces otrzyma wszystko, czego potrzebuje, i będzie mógł działać do końca. Jeśli jakiś zasób jest niedostępny, żaden zasób nie zostanie przydzielony i proces musi czekać.
    b. wymaganie od procesu zwolnienia wszystkich przetrzymywanych zasobów przed zamówieniem nowego

3. brak wywłaszczania (*no preemption*) - jeśli procesowi przydzielono drukarkę i jest on w trakcie drukowania swojego wyjścia, przymusowe zabranie mu drukarki ze względu na to, że potrzebny ploter jest niedostępny, będzie czasami niemożliwe, a na pewno trudne. Aby uniknąć tej sytuacji, można doprowadzić do wirtualizacji niektórych zasobów. Buforowanie wyjścia drukarki na dysk i zezwolenie na dostęp do fizycznej drukarki tylko demonowi eliminuje zakleszczeni związane z drukarką (tworzy możliwość zakleszczenia z powodu wyczerpania się miejsca na dysku). Nie wszystkie zasoby można jednak wirtualizować w taki sposób, np. rekordy w bazie danych lub tablice wewnątrz systemu operacyjnego.

4. cykliczne czekanie (*circural wait*)
    a. możemy zastosować regułę, zgodnie z którą w określonym momencie proces jest uprawniony tylko do jednego zasobu. Jeśli potrzebuje kolejnego, musi zwolnić poprzedni. Dla procesu, który wymaga skopiowania dużego pliku z taśmy na drukarkę, takie ograniczenie jest nie do zaakceptowania.
    b. inny sposób: nadanie każdemu zasobowi określony, unikatowy numer porządkowy (liczbę naturalną) i wymuszenie na procesach zamawiania zasobów, według rosnącej numeracji. Oznacza to, że proces, który ma już w swoim posiadaniu zasób nr 4, może zająć zasób nr 5, ale już nie może zająć zasoby nr 3, a nawet kolejnego egzemplarza zasobu nr 4. Alternatywnie można wymagać, aby proces posiadający zasób nr 4 zwolnił go przed zamówieniem zasobu nr 3.

Niech zapis *{P} I {Q}* oznacza, że formuły *P* i *Q* są prawdziwe odpowiednio przed i po wykonaniu instrukcji *I*. Formuły te nazywamy kolejno *warunkami wstępnymi* (ang. *preconditions*) i *warunkami końcowymi* (ang. *postconditions*). Zauważ, że z racji wywłaszczania dla programu {P1} I1 {Q1}; {P2} I2 {Q2} nie musi zachodzić Q1 ≡ P2!
UWAGA! W kolejnych zadaniach należy jawnie opisywać globalny stan przy pomocy formuł logicznych.

L12.Z3. W poniższym programie występuje **sytuacja wyścigu** (ang. *race condition*) na współdzielonej zmiennej «tally». Wyznacz jej najmniejszą i największą możliwą wartość. Dyrektywa «parbegin» rozpoczyna współbieżne wykonanie procesów.

```
1 const int n = 50;
2 shared int tally = 0;
3
4 void total() {
5   for (int count = 1; count <= n; count++)
```

```
6     tally = tally + 1; /* to samo co tally++ */
7 }
8
9 void main() { parbegin (total(), total()); }
```

Maszyna wykonuje instrukcje arytmetyczne wyłącznie na rejestrach – tj. kompilator musi załadować wartość zmiennej «tally» do rejestru, przed wykonaniem dodawania. Jak zmieni się przedział możliwych wartości zmiennej «tally», gdy wystartujemy *k* procesów zamiast dwóch? Odpowiedź uzasadnij.

2 podprocesy

Dolna granica = 2

podproces 1	podproces 2	wartość tally	komentarz
load (tally, reg1)		0	reg1 ma wartość 0 (początkową wartość tally)
	load (tally, reg2)	0	
	inc (reg2)	0	
	store (reg2, tally)	1	
	…	49	podproces 2 kontynuuje pracę aż do przedostatniej iteracji pętli
inc (reg1)		49	
store (reg1, tally)		1	wartość 0 trzymana w reg1 zostaje zinkrementowana i zapisana do tally
	load (tally, reg2)	1	
load (tally, reg1)		1	w obu rejestrach wartość tally wynosi 1
inc (reg1)		1	
store (reg1, tally)		2	
…		50	podproces 1 wykonuje pętlę do końca i kończy swoje działanie
	inc (reg2)	50	
	store (reg2, tally)	2	podobnie jak wcześniej, wartość 1 została zachowana w reg2, zinkrementowana i zapisana do tally

Wszystkie zakończyły podprocesy kończą swoje działanie i wartość końcowa tally wynosi 2.

Górna granica = 100

podproces 1	podproces 2	wartość tally	komentarz
	load (tally, reg2)	0	reg1 ma wartość 0 (początkową wartość tally)
	inc (reg2)	0	
	store (reg2, tally)	1	
	…	50	podproces 2 kończy się
load (tally, reg1)		50	podproces 2 zaczyna pracę
inc (reg1)		50	
store (teg1, tally)		51	
…		100	podproces 1 kończy się

k podprocesów
Dolna granica = 2

podproces 1	podproces 2	podprocesy (3 … k)	tally	komentarz
load (tally, reg1)			0	reg1 ma wartość 0 (początkową wartość tally)
	load (tally, reg2)		0	
	inc (reg2)		0	
	store (reg2, tally)		1	
	…		49	podproces 2 kontynuuje pracę aż do przedostatniej iteracji pętli
		…	x	podprocesy (3 … k) wykonują się do końca, modyfikując dowolnie zmienną tally
inc (reg1)			x	
store (reg1, tally)			1	wartość 0 trzymana w rejestrze 1 zostaje zinkrementowana i zapisana do tally, rezultat pracy podprocesów (3 … k) nie ma znaczenia
	load (tally, reg2)		1	
load (tally, reg1)			1	
inc (reg1)			1	
store (reg1, tally)			2	
…			50	podproces 1 wykonuję pętlę do końca i kończy swoje działanie
	inc (reg2)		50	
	store (reg2, tally)		2	podobnie jak wcześniej, wartość 1 została zachowana w reg2, zinkrementowana i zapisana do tally

Wszystkie zakończyły podprocesy kończą swoje działanie i wartość końcowa tally wynosi 2.

Górna granica = 50k
Podobnie jak w przykładzie dla 2 podprocesów, każdy podproces wykonuję się bez zamrożenia stanu i przełączenia się na inny podproces:

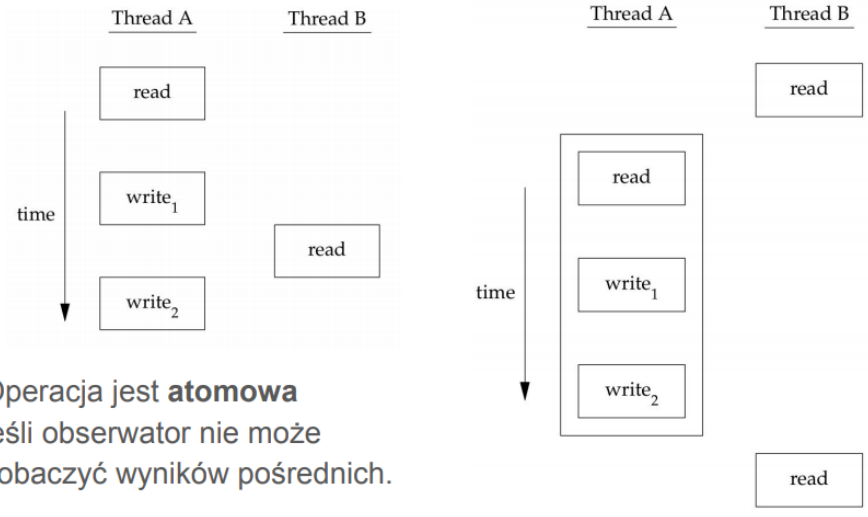
- podproces 1 wykonuję się od początku do końca (tally = 50)
- podproces 2 wykonuje się od początku do końca (tally = 2 * 50).
- …
- podproces k wykonuje się od początku do końca (tally = k * 50).

Dlaczego jest granica 50k to maksimum? Każdy podproces wykonuje maksymalnie 50 inkrementacji, które mogę zwiększyć o 1 zmienną tally. Poprocesów jest k, więc 50k to maksimum.

L12.Z4. Rozważmy poniższy kod ze slajdów do wykładu. Zakładamy, że kolejka «queue» przechowuje do *n* elementów. Wszystkie operacje na kolejce są **atomowe** (ang. *atomic*). Startujemy po jednym wątku wykonującym kod procedury «producer» i «consumer». Procedura «sleep» usypia wołający wątek, a «wakeup» budzi wątek wykonujący daną procedurę. Wskaż przeplot instrukcji, który doprowadzi do
(a) błędu wykonania w linii 6 i 13
(b) zakleszczenia w liniach 5 i 12.

```
1  def producer():
2    forever:
3      item = produce()
4      if queue.full():
5        sleep()
6      queue.push(item)
7      if not queue.empty():
8        wakeup(consumer)

9  def consumer():
10   forever:
11     if queue.empty():
12     sleep()
13   item = queue.pop()
14   if not queue.full():
15     wakeup(producer)
16   consume(item)
```



operacje atomowe – obserwator operacji nie może zobaczyć wyników pośrednich. Atomiczność gwarantuje nam brak przerwań sprzętowych, sygnałów, izolację od innych wątków i procesów.

(a) błędu wykonania w linii 6 i 13

- Queue: 0, producer: **2** consumer: 10
- Queue: 0, producer: **3** consumer: 10
- Queue: 0, producer: **4** consumer: 10
- Queue: 0, producer: **6** consumer: 10
- Queue: **1**, producer: **7** consumer: 10 if jest true, nastepne wykona 8
- Queue: **1**, producer: 7 consumer: **11-13**
- Queue: 0, producer: 7 consumer: **14-16, 11,12**
- Queue: 0, producer: **8** consumer: 12 tu budzi consumer, gdy kolejka pusta
- Queue: 0, producer: 8 consumer: **13**

(b) zakleszczenia w liniach 5 i 12

- Queue: 0, producer: 2 consumer: **11** if jest true, nastepny bedzie sleep
- Queue: 0, producer: **3,6,7** consumer: 11
- Queue: **1**, producer: **3,6,7** consumer: 11
- Queue: **2**, producer: **3,6,7** consumer: 11
- …
- Queue: **n**, producer: **3,4** consumer: 11 nastepny bedzie sleep
- Queue: n, producer: 4 consumer: **12** idzie spac, mimo ze kolejka jest pelna
- Queue: n, producer: **5** consumer: 12 tez zasypia

L12.Z5. Przeczytaj rozdział 2 artykułu „*Beautiful concurrency*”. Na podstawie poniższego przykładu wyjaśnij czemu złożenie ze sobą poprawnych współbieżnych programów używających blokad nie musi dać poprawnego programu (tj. „*locks are not composable*”). Jak poprawić procedurę transfer? Czemu według autorów artykułu blokady nie są dobrym narzędziem do strukturyzowania współbieżnych programów?

Cel: napisz procedurę transferującą pieniądze z jednego konta bankowego do drugiego. Oba konta są trzymane w pamięci, nie ma operacji na bazach danych. Procedura musi poprawnie działać w programie wykonywanym współbieżnie, żaden wątek nie powinien widzieć pośrednich stanów pomiędzy opuszczeniem konta 1 a dotarciem na konto 2.

```
1  class Account {
2    int balance;
3    synchronized void withdraw(int n) { balance -= n; }
4    synchronized void deposit(int n) { balance += n; }
5  }
```

synchronized przy withdraw powoduje, że na czas trwania withdraw nałożona jest blokada
niepuci, bo we współbieżnym programie wątek 2 może zobaczyć operacje pomiędzy przerzuceniem pieniędzy z from do to. To, że są synchronized wiele nam nie daje, ponieważ pomiędzy wywołaniami from i to jest przerwa.

```
7  void transfer(Account from, Account to, int amount) {
8    from.lock(); to.lock();
9    from.withdraw(amount);
10   to.deposit(amount);
11   from.unlock(); to.unlock();
12 }
```

transfer jest podatny na zakleszczenia. Przykład: dwa wątki transferują pieniądze w przeciwnych kierunkach na dwa te same konta A, B. Jeden zakłada blokadę na konto A, drugi na B. Jeden wątek nie może przesłać pieniędzy na A, drugi na B, bo są zablokowane.

```
if from < to
then { from.lock(); to.lock(); }
else { to.lock(); from.lock(); }
```

Proponowane rozwiązanie: ustalamy blokady z pewnym liniowym porządkiem. Zadziała, gdy wiemy, ile blokad jest potrzebnych i musi być to znane z góry. Przykład: sytuacja - from.withdraw bierze pieniądze z from2, jeśli na from nie ma środków. Zanim nie przeczytamy from, nie wiemy, czy na from2 trzeba założyć blokadę. Może na założenie blokady na from2 jest już za późno, skoro składamy je w ustalonym porządku.

Blokady i zmienne warunkowe nie wspierają programowania modularnego – procesu budowania dużych programów poprzez sklejanie ze sobą mniejszych. Blokady to uniemożliwiają. Np. nie możemy użyć naszych (poprawnych) implementacji withdraw i deposit w niezmienionym stanie do implementacji transfer. Musimy ujawnić nasz protokół blokowania, abstrakcja nie jest zachowana.

L12.Z6. Poniżej znajduje się propozycja programowego rozwiązania problemu wzajemnego wykluczania dla dwóch procesów. Znajdź kontrprzykład, w którym to rozwiązanie zawodzi. Okazuje się, że nawet recenzenci renomowanego czasopisma „*Communications of the ACM*" dali się zwieść.

```
1  shared boolean blocked [2] = { false, false };
2  shared int turn = 0;
3
4  void P (int id) {
5    while (true) {
6      blocked[id] = true;
7      while (turn != id) {
8        while (blocked[1 - id])
9          continue;
10       turn = id;
11     }
12     /* put code to execute in critical section here */
13     blocked[id] = false;
14   }
15 }
16
17 void main( ) { parbegin (P(0), P(1)); }
```

Idea: blocked == true oznacza chęć wejścia do sekcji krytycznej. Po uruchomieniu P proces ustawia wartość blocked na true (L6). Jego tura nadejdzie (L10) dopiero wtedy, gdy uzyskamy pewność, że drugi proces nie jest zainteresowany sekcją krytyczną, skończył w niej pracę (L8). Przechodzimy do sekcji krytycznej (L12) i po wykonaniu pracy porzucamy zainteresowanie sekrytem (L13).

Kontrprzykład:

- stan początkowy:
  - blocked = {false, false}
  - turn = 0
- P1 wykonuje się
  - wchodzi do pętli L7 „while (turn != id)" (0 != 1)
  - nie wchodzi w do pętli L8 „while (blocked[1 - id])" (blocked[0] == false)
  - wstrzymuje wykonanie przed L10 „turn = id", zostaje wywłaszczony
- stan:
  - blocked = {false, true}
  - turn = 0
- P0 wykonuje się
  - nie wchodzi do pętli L7 „while (turn != id)" (0 != 0)
  - L12: zaczyna wykonywać sekcję krytyczną, zostaje wywłaszczony
- stan:
  - blocked = {true, true}
  - turn = 0
- P1 wznawia wykonanie
  - zaczyna od L10 „turn = id", wychodzi z pętli
  - L12: wchodzi w sekcję krytyczną

Mechanizm nie zawsze działa dobrze, ponieważ pozwala, żeby dwa procesy były w sekcji krytycznej.