Struktura systemów operacyjnych

systemy wsadowe – 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.

podstawowe funkcje jądra
- zarządzanie zasobami (pamięcią, urządzeniami I/O)
- komunikacja między procesami
- planowanie (planista – planuje, dyspozytor – wykonuje)
- interakcja z systemem plików

przestrzeń użytkownika i jądra - podział przadra:
- przadr użytkownika - przestrzeń w której wykonują się programy
- przadr jądra - miejsce, w którym wykonują się procedury jądra

mechanizm vs. polityka
- polityka – zestaw reguł, które system ma przestrzegać. Pytanie: jak zaaranżować elementy?
- mechanizm – praktyczna realizacja polityki. Pytanie: jak można użyć elementów? Zmiana polityki nie powinna zmieniać mechanizmu (np. polityka: do pokoju hotelowego mają dostęp jednak inne osoby – dalej możemy dać nowym osobom klucze lub nadać uprawnienia ich kartom magnetycznym)

implementacja wywołań systemowych
1. zapisz kontekst
2. znajdź kod procedury - wektor syscalli
    a. albo robimy software interrupt i potem dopiero szukamy w wektorze syscalli (MIPS tak ma chyba); na x86: INT $0x80
    wskaźnik na wektor syscalli w PCB
    b. albo mamy instrukcję syscall która nie wykorzystuje mechanizmu przerwań
3. sprawdź poprawność argumentów
4. skopiuj argumenty (copy-to-kernel())
5. obsłuż
6. skopiuj wynik (copy-to-user())
7. ustawienie errno
8. userret - odpala schedulera i sprawdza sygnały
9. odtworzenie kontekstu
10. powrót

jądro monolityczne – wszystko w jednej binarce, więc wszystko ma dostęp do wszystkiego. Wady: jak coś mało ważnego się wysypie, to wszystko leży. Zalety: lepszy performance. Linux jest monolityczny.

mikrojądro i procesy usługowe – w przestrzeni jądra znajduje się minimum:
- scheduler
- zegar
- obsługa przerwań
- IPC

Reszta żyje jako osobne procesy w user space (np. file system) – **serwery**. Przeciwne wady/zalety.

struktura warstwowa – organizacja jądra jako hierarchii warstw. Idea: każda warstwa może odwoływać się do funkcji warstw niższych, ale nie wyższych

moduły jądra i sterowniki
- moduły - pliki .o ładowane do przestrzeni jądra podczas runtime - nie trzeba przekompilowywać jądra
- sterownik – kod w jądrze do obsługi urządzenia zewnętrznego

wywoływanie procedur vs. przekazywanie komunikatów –

- w jądrze monolitycznym możemy po prostu zawołać każdą procedurę - niski narzut, słabe bezpieczeństwo
- w mikrojądrach chcemy przekazywać komunikaty do różnych serwerów świadczących usługi

warstwa abstrakcji sprzętu – interfejs dla wyższych warstw systemu

jądro hybrydowe – struktura mikrojądra, ale zaimplementowane jako monolityczne – w jądrze jest wszystko, co potrzeba, ale serwery są w przestrzeni jądra

Zadanie 1 (3). Architektura systemu operacyjnego.

T Awaria sterownika w jądrze monolitycznym może doprowadzić do załamania się systemu.

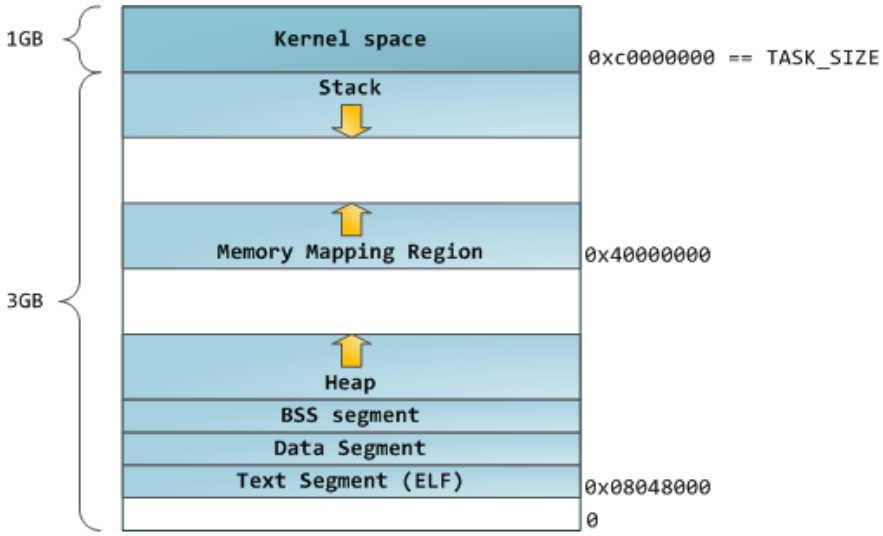
N Jądro monolityczne nie umożliwia doładowywania sterowników w trakcie działania systemu.

N Mikrojądro zawiera w sobie algorytmy szeregowania wątków.

N Jądro jest wywłaszczalne, jeśli przerwania mogą się zagnieżdżać.

Procesy

przestrzeń adresowa procesu



zasoby procesu i PCB

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	

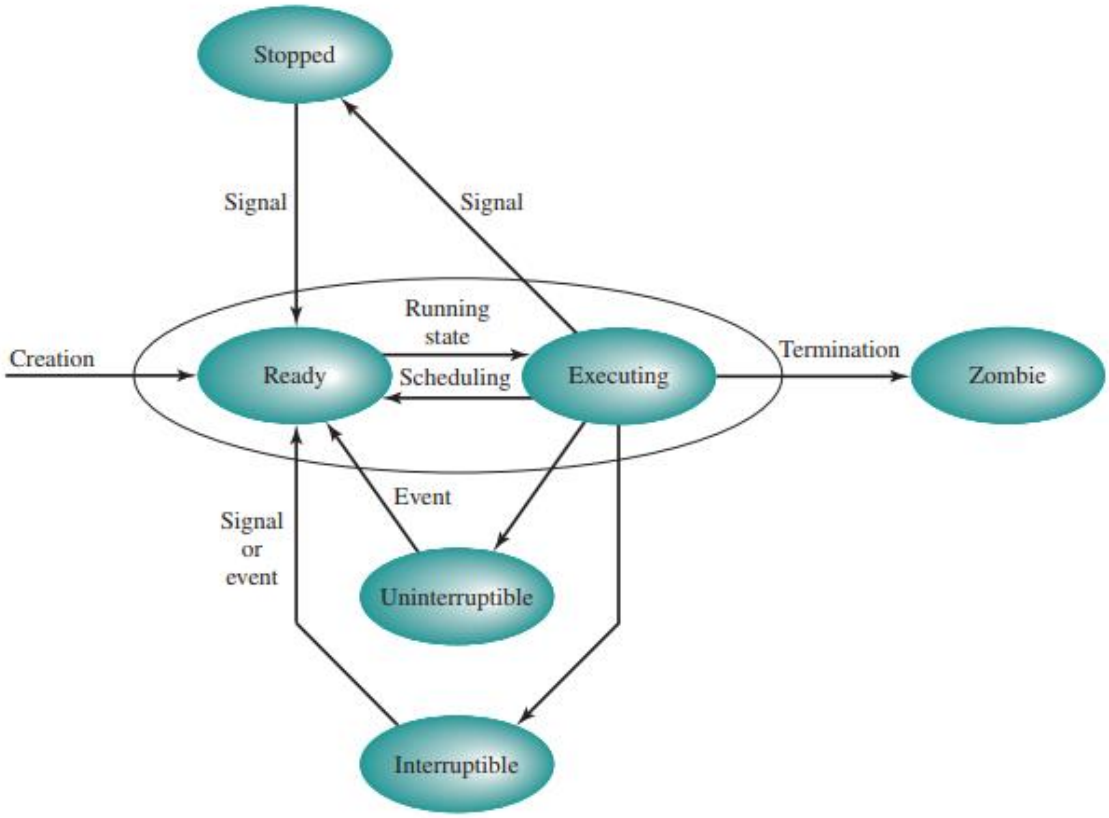
Process Control Block (PCB) – struktura jądra, która przechowuje informacje o procesie i zasobach, z których korzysta:

- 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)

stan procesu i przejścia między stanami

- **Running**: This state value corresponds to two states. A Running process is either executing or it is ready to execute.
- **Interruptible**: This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
- **Uninterruptible**: This is another blocked state. The difference between this and the Interruptible state is that in an Uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
- **Stopped**: The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
- **Zombie**: The process has been terminated but, for some reason, still must have its task structure in the process table.



górna i dolna połówka - dolna połówka jądra - procedury obsługi przerwań, górna połówka - cała reszta

przełączanie kontekstu, trybu pracy procesora i przestrzeni adresowych

Przełączanie procesów

Przełączanie procesów polega na przełączeniu przestrzeni adresowych, wstrzymaniu wirtualnego czasomierza T_1, wymianie kontekstu procesora, wznowieniu czasomierza T_2.

1. Wątek może zostać **uśpiony** (przez zawołanie `cv_wait`) w wyniku wykonania w jądrze **operacji blokującej**.
2. W trakcie przetwarzania wywołania systemowego lub procedury obsługi przerwania może zostać wybudzony wątek o wyższym **priorytecie**.
3. W wyniku upłynięcia kwantu czasu (przerwanie zegarowe) należy **wstrzymać** dany wątek.

sygnały i ich obsługa

funkcje sygnałów:
- komunikacja międzyprocesowa
- tłumaczenie wyjątków
- procesora i pułapek
- sytuacje wyjątkowe
- zarządzanie procesami

SIGKILL i SIGSTOP nie da się zignorować
SIGUSR1/2 – do zdefiniowania
SIGABRT – wysyłany, żeby zakończyć proces i force a core dump
SIGSEGV – proces odwołał się do nieprawidłowego adresu pamięci

Sygnały: wysyłanie

- **synchroniczne** → związane z wykonaniem instrukcji
- **asynchroniczne** → zdarzenia zewnętrzne: budzik, przerwanie (CTRL+C), zakończenie procesu potomnego

SO tłumaczy wyjątki i pułapki na sygnały synchroniczne. W tym przypadku jądro **wysyła sygnał** do wątku.

Sygnały asynchroniczne to prymitywna metoda komunikacji między programami i zgłaszanie zdarzeń przez jądro. Wątki mogą wysyłać sygnały do procesów!

Sygnały: doręczanie

Jądro ma kilka opcji: zignorować sygnał, zakończyć proces, wywołać **procedurę obsługi sygnału** (ang. *signal handler*), której kod jest wykonywany przez jeden z wątków procesu.

Kiedy jądro sprawdza, czy należy doręczyć sygnał?

1. Przed wejściem w stan uśpienia.
2. Po wyjściu ze stanu uśpienia.
3. Przed powrotem do przestrzeni użytkownika `userret(9)`.

Jeśli sygnały mogą przerywać sen, to mówimy wtedy o **śnie przerywalnym** (ang. *interruptible sleep*).

sen płytki i głęboki

(1) Sen przerywalny (interruptible) VS nieprzerywalny (uninterruptible)
Kontekst: wątek czeka na jakieś zdarzenie.
(1A) Sen przerywalny -- wątek można zbudzić wysłaniem mu sygnału, zanim to zdarzenie się stanie
(1B) Nieprzerywalny -- nie można wybudzić wcześniej wątku sygnałem, ani niczym, generalnie wątek obudzi się dopiero gdy stanie się to, na co czeka

(2) Sen płytki (bounded, ograniczony) VS głęboki (unbounded, nieograniczony)
Kontekst: znowu wątek czeka na jakieś zdarzenie
(2A) Sen głęboki, tzn. nieograniczony: to zdarzenie jest zewnętrzne od komputera, nie wiadomo kiedy się stanie, np. czekamy aż użytkownik wciśnie guzik na klawiaturze, albo czekamy na jakiś "warunek"
(2B) Sen płytki, tzn. ograniczony: wiemy że jeśli damy trochę czasu procesorowi, to to zdarzenie na pewno się stanie (nie czekamy na nic zewnętrznego) -- przykład: czekamy aż inny wątek skończy obliczenia, i wiemy że ten inny wątek nie będzie czekał na wciśnięcie klawisza na klawiaturze

tworzenie procesów i kończenie ich działania

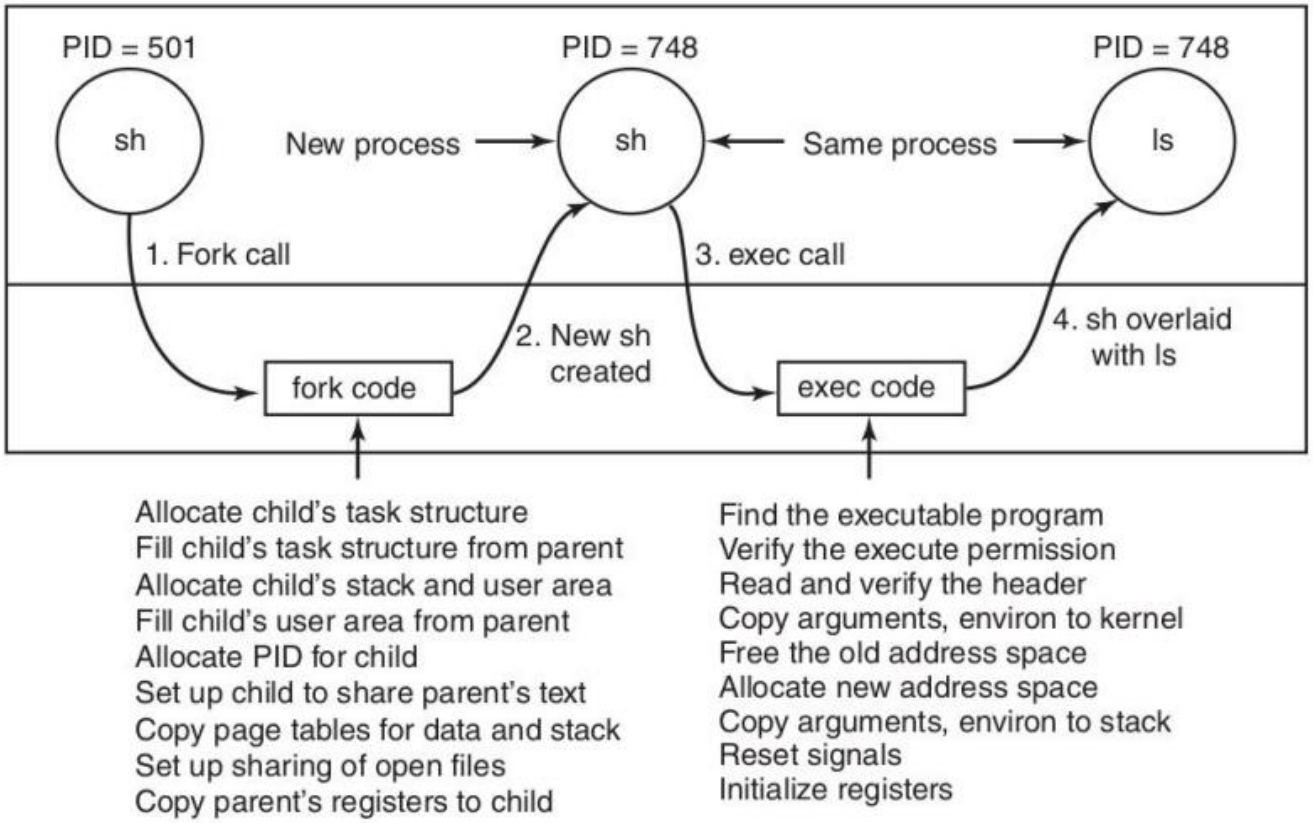
Tworzenie procesów
- inicjalizacja systemu
- uruchomienie wywołania systemowego tworzącego proces przez działający proces (UNIX: fork)
- żądanie usera utworzenia nowego procesu
- zainicjowanie zadania wsadowego (systemy wsadowe w komputerach mainframe)

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

1. Gdy proces umiera (np. po sygnale kill) wysyła SIGCHLD do rodzica
2. Rodzic musi zwolnić slot dziecka. W międzyczasie dziecko zostaje zombie - po to, aby proces mógł odczytać ewentualne statystyki
3. Jeżeli zabijemy proces rodzica zanim zwolni slot dziecka, dziecko zostaje jako zombie - nie można wysłać sygnału do procesu zombie, bo formalnie ten proces już nie istnieje
4. sieroty przygarnia systemd/init i je zabija (ale nigdzie nie jest powiedziane że od razu).

fork:

UNIX: Uruchamianie programów



hierarchia procesów

Relacja rodzic ↔ potomek

W systemach uniksowych tak, ale w *WinNT* opcjonalnie.

Czytanie identyfikatorów: `getpid`, `getppid`, `getpgrp`.

Wszystkie procesy poza `init` muszą mieć rodzica. Co się stanie jeśli rodzic umrze? Ktoś musi przygarnąć **sieroty**.

Zakończonego procesu nie można od razu usunąć. Co gdy rodzic chce zobaczyć zużycie zasobów lub **kod wyjścia**? Proces najpierw przechodzi do stanu `ZOMBIE`.

Możliwe grupowanie procesów w **zadania** (ang. *job*).

tożsamość i uprawnienia procesów

W systemach wieloużytkownikowych wymagana **kontrola dostępu** (ang. *access control*) do zasobów. Procesy mają **tożsamość** (ang. *identity*) użytkownika, z reguły tego który je utworzył. Tożsamość należy potwierdzić przy pomocy mechanizmu **uwierzytelniania** (ang. *authentication*).

Proces widzi zasoby w obrębie przydzielonych **przestrzeni nazw** → system plików. Prosi jądro o przydzielenie zasobu → otwarcie pliku. Jeśli proces ma odpowiednie **uprawnienia** (ang. *permissions*) to dostaje **uchwyt** do zasobu.

Katalog roboczy procesu → wyróżniony element przestrzeni nazw.

Globalna przestrzeń nazw dla zasobów → system plików.

Tożsamość procesu: **użytkownik**, **grupa podstawowa**, **grupy rozszerzone**. Wyróżniony przez jądro użytkownik **`root`** pełni rolę administratora.

Proces dziedziczy zestaw **umiejętności** (`capabilities`), których dobrowolnie może się zrzekać → bezpieczeństwo.

Prezentacja narzędzi `getent` i `id`!

Baza danych dostępna przez usługę `NSS`. Zestaw narzędzi systemowych do modyfikacji bazy danych np.: `useradd`.

śledzenie wykonania procesów

Z użyciem wywołania `ptrace` można:

- podłączyć się do istniejącego procesu by go odpluskwiać
- czytać i modyfikować stan procesu (rejestry, pamięć)
- przechwytywać sygnały wysyłane do procesu
- nasłuchiwać na zdarzenia (`fork`, `exec`)
- śledzić wywołania systemowe
- przełączyć wykonywanie programu w tryb krokowy

Wykład:

Proces to środowisko uruchomieniowe składające się: z przestrzeni adresowej, załadowanego programu, przydzielonej pamięci, otwartych plików i innych zasobów. Proces można zasiedlić niezależnymi kontekstami wykonania instrukcji - wątkami.

Wywłaszczanie: wstrzymywanie i wznawianie programu, bez jego wiedzy i bez wpływu na jego poprawność. Nie pozwala zdominować czasu procesora przez błędnie działający program (np. nieskończona pętla).

Multipleksowanie: metoda koordynowania dostępu do zasobów współdzielonych → wypożyczanie na kwant czasu.

Zadanie 2 (3). O procesach ogólnie.

- [T] Obsługa wywołania systemowego wymaga zmiany trybu pracy procesora.
- [N] Jądro może zmienić stan procesu z `READY` na `BLOCKED`.
- [N] Rekord procesu przechowuje kontekst procesora.
- [T] Mechanizm śledzenia procesów pozwala jednemu procesowi czytać komórki pamięci należące do innego procesu.

Zadanie 4 (3). Procesy i wątki w **systemach uniksowych**.

- [N] Proces można zawsze zakończyć poprzez wysłanie sygnału `SIGTERM`.
- [N] Sierota to proces pozbawiony wszystkich zasobów z wyjątkiem rekordu procesu.
- [T] Proces, który przeszedł w płytki (ograniczony czasowo) sen, można wybudzić sygnałem.
- [N] Proces zawsze posiada tożsamość użytkownika, który go utworzył.

Zadanie 5 (3). Programowanie z użyciem procesów **systemu uniksowego**.

- [T] Proces utworzony wywołaniem `fork` dziedziczy po swoim rodzicu otwarte połączenia sieciowe.
- [T] Żeby uruchomić nowy proces ładowany z pliku należy użyć pary wywołań systemowych `fork` i `execve`.
- [N] Jeśli nastąpi powrót z procedury, od której zaczęło się wykonanie procesu, to proces ten zakończy się poprawnie.
- [N] Zmienne środowiskowe są przechowywane w pamięci współdzielonej między spokrewnionymi ze sobą procesami.

Zadanie 22 (10). Opisz strukturę danych wykorzystywaną przez jądro systemu uniksopodobnego do zarządzania przestrzenią wirtualną procesu i algorytm obsługi błędnego dostępu do pamięci. Jak algorytm odróżnia błąd strony od błędu programisty. Jakich danych wymaga ta procedura?

W PCB proces przechowuje listę segmentów, z których każdy zawiera pierwszy i ostatni adres wirtualny, uprawnienia dostępu `rwx` i wskaźnik na obiekt wspierający. Obiektem wspierającym może być pamięć anonimowa lub plik dyskowy.

Algorytm obsługi błędu strony dysponuje adresem wirtualnym, który spowodował wyjątek procesora, oraz rodzaj dostępu: odczyt/zapis/wykonanie. Jeśli adres nie należy do żadnego segmentu to błąd programisty, koniec. Jeśli strony nie ma w pamięci, to musimy przydzielić ramkę i wyzerować ją (pamięć anonimowa) lub załadować z dysku (odwzorowanie pliku w pamięć), koniec. Jeśli uprawnienia dostępu nie zgadzają się z uprawnieniami segmentu to błąd programisty, koniec. Jeśli segment jest `read-write`, ale strona jest `read-only`, to należy obsłużyć kopiowanie przy zapisie, tj. skopiować ramkę i podczepić w zadane miejsce z uprawnieniami do zapisu, koniec.

Użyj pojęć: ramka, segment, uprawnienia dostępu, obiekt wspierający, pamięć anonimowa, odwzorowanie pliku, kopiowanie przy zapisie.

Wątki

wykonywanie współbieżne a równoległe

- przetwarzanie równoległe (parallel) – działanie wielu procesów jednocześnie dzięki wielu rdzeniom w procesorze lub wielu procesorom.
- 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.

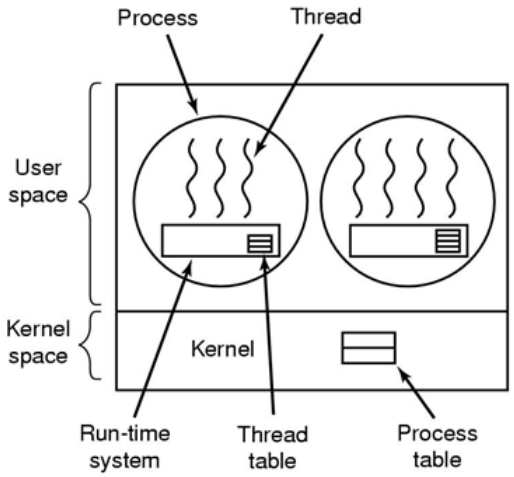
modele programowania współbieżnego: maszyny stanów, współprogramy, wątki

- maszyny stanów - możemy mieć jeden wątek i w nim napisać maszynę stanów, która będzie realizować współbieżność. Jak coś się wywali to wszystko leży. Ciężko zdefiniować wszystkie stany poprawnie. Ogólnie ciężkie w implementacji.
- współprogramy - kooperacyjne wątki. programista zarządza zmianą wątku. gdy pagefault w jednym wątku to wszystkie zablokowane
- wątki - no jakto wątki

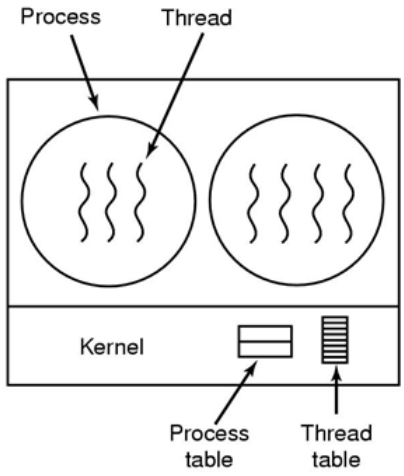
motywacja stojąca za używaniem wątków - jak się jeden wątek wywali, to nie wywala się cały program. Lepsze ustrukturyzowanie programów. W czasie gdy jeden wątek się zablokuje, to inne mogą kontynuować. Wykorzystanie wielu CPU.

problemy w programowaniu z użyciem wątków - niedeterminizm, trzeba umieć dobrze zakładać odpowiednie blokady, przewidzieć wyścigi i deadlocki. Ciężko reprodukować błędy.

porównanie wątków przestrzeni jądra i przestrzeni użytkownika



A user-level threads package.



A threads package managed by the kernel.

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.

Czasami zwane też włóknami (ang. fiber), zielonymi wątkami (ang. green threads) lub współprogramami (ang. coroutines).

\+ przełączanie wątków nie wymaga przejścia do jądra
\+ aplikacja wie jak efektywnie planować wykonanie wątków
\+ wielowątkowość kooperacyjna (mniej problemów z synchronizacją)
\+ niezależne od systemu operacyjnego
− większość wywołań systemowych blokuje
− jeden ULT potrafi wstrzymać wykonanie pozostałych w procesie
− błąd strony blokuje wszystkie wątki
− trzeba zaprogramować środowisko wykonawcze (ang. *runtime library*)

Biblioteka ULT zawiera opakowanie (ang. *wrapper*) blokujących wywołań systemowych oraz zarządzanie i przełączanie wątków.

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?

procedury wielobieżne i wątkowo bezpieczne

- 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).

aktywacje planisty

Biblioteka ULT udostępnia środki synchronizacji i wątki blokują się w przestrzeni użytkownika. Potrzebna obsługa akcji blokujących w jądrze.

Aplikacja zgłasza jądru liczbę wirtualnych procesorów, których chce używać. Kiedy wirtualny procesor się blokuje aplikacja dostaje wezwanie (ang. *upcall*) – coś jak obsługa sygnału. Wezwanie wzywa procedurę planowania środowiska uruchomieniowego i informuje o zmianie stanu wirtualnych procesorów. Tablica zdarzeń typów:

```
SA_UPCALL_(NEWPROC | PREEMPTED | BLOCKED | UNBLOCKED | SIGNAL)
```

… i do tego zapisany kontekst przerwanego wątku.

lokalna przestrzeń wątków (ang. thread local storage) – stos wątku, zmienne oznaczone __local (dzięki temu thread-safe)

problemy z wątkami w systemach uniksowych

Sygnały Z synchronicznymi jest prosto (`SIGSEGV`, `SIGFPE`) bo idą bezpośrednio do wątku. Co się stanie jeśli do procesu przyjdzie sygnał asynchroniczny (`SIGINT`, `SIGHUP`)? Który go obsłuży?

Fork Co się dzieje z pozostałymi wątkami przy klonowaniu? Tylko aktywny przechodzi! Lądujemy w nowej przestrzeni adresowej z założonymi blokadami, których nie ma kto zwolnić!
Można próbować z `pthread_atfork`…

I/O Co jeśli dwa wątki czytają z tego samego pliku? W jakiej kolejności wykonują się operacje? Używać `pread` i `pwrite`!

Zadanie 3 (3). O wątkach ogólnie.

N Przełączanie kontekstu między wątkami należącymi do dwóch różnych procesów jest równie szybkie co przełączanie wątków należących do tego samego procesu.

N Mikrowątki (włókna) nie posiadają własnego stosu.

N Wątki użytkownika (ang. *ULT*) mogą się synchronizować z użyciem muteksów udostępnionych przez jądro.

N Lokalna przestrzeń wątku (ang. *TLS*) jest niedostępna dla innych wątków w danym procesie.

Współbieżność i synchronizacja

sytuacja wyścigu, zagłodzenie, uwięzienie, zakleszczenie

- wyścig (race condition) - sytuacja w której wynik zależny jest od kolejności wykonania operacji
- zagłodzenie (starvation) - wiele wątków działa, ale istnieje podzbiór, który nie jest dopuszczony do zasobów
- uwięzienie (livelock) - wątki coś robią, np. przerzucają między sobą blokadę (ustępują sobie), ale żaden nie postępuje w wykonaniu
- zakleszczenie (deadlock) - wątki czekają na siebie nawzajem
    - brak odbierania zasobów
    - hold and wait
    - cykl
    - wzajemne wykluczenie (tylko jeden ma blokadę)
- sekcja krytyczna – fragment programu korzystający ze współdzielonych zasobów
- operacja atomowa – obserwator nie może zobaczyć wyników pośrednich operacji

wpływ ziarnistości blokad na wydajność programów

- rywalizacja o blokady (lock contention) – kiedy zadanie oczekuje na zwolnienie (release) blokady założonej (acquire) przez inne zadania
- narzut wydajnościowy (lock overhead) – czas, jaki zadanie spędza na wykonywanie akcji założenia lub zwolnienia blokady
- ziarnistość (granularity) – określa ilość chronionych danych – jak długo zadanie wykonuje się z założoną blokadą?

- ziarnistość blokad duża (coarse-grained) – sumarycznie niski narzut, ale wysoka rywalizacja
- ziarnistość blokad mała (fine-grained) – sumarycznie wysoki narzut, ale niska rywalizacja

problem odwrócenia priorytetów i jego rozwiązanie
problem: gdy H i L współdzielą zasób, L założył blokadę, skończył się jego czas – wykonuje się H, ale H musi czekać na blokadzie, którą założył L. H zasypia. Wykonują się wątki pośrednie między H i L – L nie zwolni blokady, bo nie dostanie procesora.

rozwiązanie: propagacja priorytetów. Tymczasowo podbijmy priorytet L, L się wykona i zwolni blokadę, H znowu będzie mógł działać.

muteksy zwykłe i rekurencyjne
m.lock(); - albo wezmę, albo spanko i potem wezmę
m.lock(); - undefined behaviour

rekurencyjny zlicza, ile razy zrobiono na nim lock i ile razy unlock

lock, lock, unlock -> niezdjęty

mutex – synchronizacja wątków

semafory zliczające

semafor – ile wątków może wejść do sekcji. post/wait może zrobić ktokolwiek, mutex zwolnić tylko ten, kto go założył.
wait – cnt()—
wartość 0 -> zadanie idzie spać

post – cnt()++
wartość 0 -> wybudzamy wątek

nazwane – procesy
nienazwane – wątki

semafor binarny – 0/1 – muteks bez właściciela

zmienne warunkowe
ze zmienną warunkową zawsze związany jest mutex
wait – idź spać
signal – budzi 1 wątek
broadcast – wybudź wszystkie wątki, które teraz śpią

idąc spać ściąga mutex, budząc się bierze mutex

blokady wirujące
while(zasób zajęty) continue;

kiedy ok? w kodzie obsługi przerwań, bo mutex by zablokował cały wątek, a nie procedurę obsługi przerwania

blokady adaptacyjne
przez chwilę wirująca, potem zasypia – mutex, który przez chwilę się pętli, jak mija czas, to idzie spać

mutexy domyślnie są adaptacyjne

bariery synchronizacyjne
wszystkie zadania muszą skończyć się w tym samym czasie. parametr – ile wątków musi zrobić lock, aby się zwolniła

Zadanie 6 (3). O synchronizacji ogólnie.

T Jeśli dużo wątków często synchronizuje się na małej liczbie blokad, to zachodzi rywalizacja o blokady.

T Implementacja zmiennej warunkowej nie przechowuje wartości logicznej sprawdzanego predykatu.

N Blokada wirująca to wydajna implementacja sekcji krytycznej w systemach jednoprocesorowych.

N Wątek jest głodzony, jeśli postępuje z obliczeniami, ale system ciągle odrzuca jego żądania o dostęp do zasobu.

Zadanie 8 (3). Synchronizacja i komunikacja w **systemach uniksowych**.

T `POSIX mutex` musi być blokowany i zwalniany przez ten sam proces lub wątek.

N Wątek, który oczekiwał na zmiennej warunkowej, po wybudzeniu wchodzi jako pierwszy do monitora.

T Pamięć współdzieloną między procesami można zaimplementować z użyciem wywołania `mmap`.

N Przesyłanie komunikatów z użyciem potoków jest zawsze atomowe — nie występuje fragmentacja danych.

Zadanie 18 (10). Bariera to narzędzie synchronizacyjne służące do synchronizacji grup k wątków. Po zatrzymaniu się co najmniej k wątków na barierze (procedura «wait») w jednym kroku opuszcza ją dokładnie k wątków. Wątki przychodzą do bariery i opuszczają ją w tej samej kolejności. Po jednokrotnym użyciu bariera wraca do stanu nominalnego. Sumaryczna liczba wątków korzystających z bariery może być dużo większa niż k. Poniższa implementacja zawiera co najmniej jeden błąd. Wskaż stan, w którym funkcja «wait» zadziała nieprawidłowo.

```
1  fun barrier@init(int k):
2    turnstile1 = new Semaphore(0)
3    turnstile2 = new Semaphore(0)
4    mutex = new Semaphore(1)
5    count = 0
6    n = k
7
8  fun barrier@wait():
9    mutex.wait()
10     count += 1
11     if count == n:
12       turnstile1.signal(n)
13   mutex.signal()
14   turnstile1.wait()
15   mutex.wait()
16     count -= 1
17     if count == 0:
18       turnstile2.signal(n)
19   mutex.signal()
20   turnstile2.wait()
```

Weźmy $k+1$ wątków t_i numerowanych od 1. W tabelkę poniżej wpisz sekwencję zdarzeń, która prowadzi do osiągnięcia nieprawidłowego stanu.

wątek	po wierszu	stan
t_1	8	aktywny
$t_1 \ldots t_{k+1}$	13	aktywny
$t_1 \ldots t_k$	14	aktywny
t_{k+1}	14	zablokowany
$t_1 \ldots t_k$	20	zablokowany

UWAGA! Stan wątku należy zapisać następująco: zablokowany, aktywny.

W stanie niedozwolonym wartość semafora «turnstile1» wynosi -1, «turnstile2» wynosi -k a wartość zmiennej «count» jest równa 1. Poniżej należy słownie scharakteryzować nieprawidłowy stan:

$t_1 \ldots t_k$ powinny opuścić barierę, ale nigdy nie tego zrobią, nawet jeśli przyjdą nowe wątki t_i, gdzie $i > k+1$.

UWAGA! Ujemna wartość semafora oznacza liczbę wątków oczekujących na semaforze.

Komunikacja

pamięć dzielona

Pamięć dzielona (lokalnie)

Rodzic może utworzyć blok pamięci dzielonej wywołaniem mmap z flagą **MAP_SHARED** i utworzyć podproces. Ograniczone!

Uniwersalne rozwiązanie? POSIX.1 *shared memory* (shm_overview)
Nazwana pamięć dzielona (shm_open) istnieje póki nie zostanie usunięta (**shm_unlink**) albo do restartu. Początkowo długość zero, trzeba określić z użyciem **ftruncate**. Zasób plikopodobny (deskryptor pliku) odwzorowany w pamięci wywołaniem **mmap**.

Można również efektywnie dzielić pamięć przez plik → znów **mmap**.

Można też tworzyć anonimowe pliki odwzorowane w pamięci memfd_create i przesyłać je między procesami (o tym za chwilę).

przekazywanie komunikatów

Lokalne → kopiowanie danych między procesami: potoki, gniazda domeny uniksowej, skrzynki pocztowe, …

Rozproszone → protokoły sieciowe IPv4 lub IPv6: gniazda TCP, UDP, SCTP, RAW, …; zdalne wywołania procedur, ...

potoki
połączenie standardowego wyjścia jednego procesu ze standardowym wejściem drugiego

nienazwane – PIPES:
-pipes are accessed only through the process which share same ancestors
-jednokierunkowe
-trwają do czasu sterminowania korzystającego z nich procesu

nazwane – FIFO:
-nazwa jest zapisana w systemie plików, więc dowolne procesy mogą z nich korzystać
-istnieją nawet po śmierci procesu-twórcy
-dwukierunkowe

gniazda
Dwukierunkowa metoda komunikacji lokalnej lub sieciowej

Tcp	Udp
Połączeniowy	Bezpołączeniowy
Cel: niezawodność, nie musi być szybko	Cel: szybko, wydajnie, nie trzeba niezawodnie
Handshake: SYN, SYN-ACK, ACK	Nie ma przywitania (bezpołączeniowy)

Zadanie 7 (3). O komunikacji ogólnie.

N Potoki umożliwiają jedno- i dwukierunkową komunikację między procesami na różnych maszynach.

N Transmisja z użyciem gniazd strumieniowych dzieli dane na paczki zwane datagramami.

T Używając protokołu bezpołączeniowego przy każdym odbiorze pakietu dostajemy adres nadawcy.

N Zdalne wywołanie procedur dopuszcza przekazywanie listy otwartych połączeń jako argumentu funkcji.

IPC	Inter-Process Communication
UID	User Identifier
PCB	Process Control Block
RPC	Remote Procedure Call
HAL	Hardware Abstraction Layer
VM	Virtual machine/memory
FS	File system
COW	Copy on write
PID	Process identifier
PGID	Process group identifier
PPID	Process parent identifier
STAT	Process state
ULT	User-level threads
KLT	Kernel-level threads
TLS	Thread-local Storage
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
JCB	Job Control Language
POSIX	Portable Operating System Interface for UNIX
Mutex	Mutual exclusion