AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Systemy operacyjne
Wykład 05

Wersja 2024

dr inż. Marek Wilkus **http://home.agh.edu.pl/~mwilkus**
Wydział Inżynierii Metali i Informatyki Przemysłowej
AGH Kraków

Na podstawie programu opracowanego przez dr inż. Krzysztofa Wilka

AGH

Synchronizacja procesów

- Proces producenta z wykorzystaniem licznika:

```
while (1) {
  ...
  produkuj (jednostka, do nast_p);
 ...

  while (licznik==n) do {} //oczekuj bez produkcji
 bufor[we]=nast_p;
 we=(we+1)%n;
 licznik++;

}
```


AGH

Synchronizacja procesów

- Proces konsumenta – z licznikiem:

```
while (1) {

   while (licznik==0) do {} //czekaj bez przetwarzania
   nast_k=bufor[wy];
   wy=(wy+1)%n;
   licznik--;
  ...
  przetwarzaj (jednostka z nast_k);
  ...

}
```


AGH

Synchronizacja procesów - przykład

- Wartość licznika wynosi 5.
  Po tym czasie producent wyprodukował 1 jednostkę, konsument przetworzył również 1 jednostkę.
- Ile wynosi licznik?

AGH

Synchronizacja procesów - przykład

1) P: rejestr1=licznik //r1=5
2) P: rejestr1++; //r1=6
3) K: rejestr2=licznik //r2=5
4) K: rejestr2--; //r2=4
5) P: licznik=rejestr1; //l=6
6) K: licznik=rejestr2 //l=4

Licznik wynosi 4

5') K: licznik=rejestr2 //l=4
6') P: licznik=rejestr1 //l=6

Licznik wynosi 6.

Bez synchronizacji możemy nigdy nie uzyskać 5.

AGH

Szkodliwa rywalizacja – race condition

- Jeżeli kilka procesów współbieżnie wykorzystuje i modyfikuje te same dane, to wynik działań może zależeć od kolejności w jakiej następował dostęp do danych. Następuje wtedy szkodliwa rywalizacja.
- **Sekcja krytyczna**
  Każdy ze współpracujących procesów posiada fragment kodu w którym następuje zmiana wspólnych danych. Jest to **sekcja krytyczna** procesu.
  Jednym z zadań synchronizacji jest zapewnienie sytuacji, w której gdy jeden proces jest w swojej sekcji krytycznej to inne nie mogą wówczas wejść do swoich sekcji krytycznych.
  Stąd każdy proces musi prosić (w sekcji wejściowej) o pozwolenie na wejście do swojej sekcji krytycznej.

Budowa procesu z sekcją krytyczną:

```
while (1) {
  ...
  sekcja wejściowa
      sekcja krytyczna
  sekcja wyjściowa
  ...
}
```


Warunki poprawnego działania sekcji krytycznej

- **Wzajemne wykluczanie**: Jeżeli proces działa w swojej sekcji krytycznej, to żaden inny proces nie działa w swojej (celem zachowania spójności).
- **Postęp**: Tylko procesy nie wykonujące swoich reszt (wszystkiego poza sekcją krytyczną i sekcjami w/w) mogą kandydować do wejścia do sekcji krytycznych i ten wybór nie może być odwlekany w nieskończoność (celem m.in. uniknięcia zbędnego blokowania).
- **Ograniczone czekanie**: Musi istnieć graniczna ilość wejść innych procesów do ich sekcji krytycznych po tym gdy jakiś proces zgłosił chęć wejścia do swojej i zanim otrzymał na to pozwolenie (celem nie zagłodzenia procesu).

Przykładowy algorytm synchronizacji

- Zmienne **wspólne**:
  int znacznik[2];
  int numer; //0..1
- Na początku znacznik[0]=znacznik[1]=0
- Odpowiedni fragment procesu *i* (proces konkurencyjny ma numer *j*):

```
while (1) {
. . .
  znacznik[i]=1;              //zgłaszam chęć wejścia do sekcji!
  numer=j;
  while (znacznik[j]==1 and numer==1) do {} //czekaj na wejście - inny zgłosił
        sekcja krytyczna
 znacznik[i]=0;                    //koniec, zwalniam sekcję
        reszta
. . .
}
```


Rozkazy niepodzielne

- Są to sprzętowe rozkazy składające się z kilku kroków, ale muszą być wykonywane nieprzerwanie, np.:

```
bool testuj_i_ustal(bool* znak) {
      bool tmp=*znak;
      *znak=true;
      return tmp;
}
```

- Przykładowe użycie:

```
while (1) {
. . .
while (testuj_i_ustal(wspolna)) do {}
      sekcja krytyczna
wspolna=false;

      reszta
. . .
}
```


Rozkazy niepodzielne - po co?

- Pomiędzy wykryciem zwolnionego dostępu do sekcji krytycznej a wejściem nastąpiło przerwanie wstrzymujące wejście do sekcji.
- Inny proces też zauważa zwolnienie sekcji.
- Dwa procesy wchodzą jednocześnie do sekcji krytycznej.

- Systemy jednoprocesorowe: Blokujemy przerwania.
- Systemy wieloprocesorowe: Blokujemy przerwania, alokację rdzeni, przełączanie zadań na rdzenie… (wiele różnych algorytmów)

Semafory

- Są to sprzętowe zmienne całkowite, do których dostęp możliwy jest tylko przez dwie niepodzielne operacje:
  - czekaj(S): while (S<=0) {}
    S--;
  - sygnalizuj(S): S++;

Czekaj, aż będzie można zdekrementować!

- Zastosowanie:

```
semafor wspolna;
while (1) {
 czekaj(wspolna);
    sekcja krytyczna
 sygnalizuj(wspolna);
    reszta
}
```

Przykład:

- Instrukcja S2 w procesie P2 musi być wykonana po zakończeniu wykonywania instrukcji S1 w procesie P1.

```
S1;
sygnalizuj(sync);

czekaj(sync);
S2;
```


Semafory z blokowaniem procesu

- Unika się "zapętlenia" procesu przed semaforem.
- Proces zamiast aktywnie czekać, umieszczany jest w kolejce związanej z danym semaforem i "usypiany".
- Operacja **sygnalizuj** wykonana przez inny proces "budzi" proces oczekujący i umieszcza go w kolejce procesów gotowych do wykonania.

```
type semaphore = record {
        int wartość;
        L[] list of processes;
}
```


Semafory z blokowaniem

```
czekaj(S):
        S.wartość--;
        if (S.wartość<0) {
                dołącz dany proces do listy;
                blokuj;
        }

sygnalizuj(S):
        S.wartość++;
        if (S.wartość<=0) {
                usuń proces  P z listy;
                obudź(P);
        }
```

Jeżeli wartość<0, to abs(wartość) – **liczba procesów czekających na ten semafor!**

Semafory a systemy wieloprocesorowe

- W systemach jednoprocesorowych niepodzielność operacji „czekaj” i „sygnalizuj” można zapewnić poprzez blokadę przerwań na czas wykonywania ich rozkazów.
- W środowisku wieloprocesorowym nie ma możliwości blokowania przerwań z innych procesorów - w takim przypadku wykorzystuje się rozwiązania z sekcji krytycznych - operacje „czekaj” i „sygnalizuj” są sekcjami krytycznymi. Ponieważ ich kody są małe (kilkanaście rozkazów), to zajmowane są rzadko i przypadki aktywnego czekania nie występują często i trwają krótko.

Semafory w UNIX

- man…
  - sem_overview
  - sem_wait
  - sem_post

Problem czytelników i pisarzy

(synchronizacja procesów zapisujących i czytających te same dane)

- Problem 1: Żaden z czytelników nie powinien czekać chyba, że pisarz w tym momencie pisze,
- Problem 2: Jeśli pisarz czeka na dostęp, to żaden nowy czytelnik nie rozpocznie czytania.
- Rozwiązanie:
  Procesy dzielą zmienne:
  semaphore wyklucz, pisanie;
  int liczba_czyt;
- Semafor **pisanie** jest wspólny dla czytających i piszących, obydwa na początku mają wartość 1, a liczba czytelników=0.
- Semafor **pisanie** organizuje wykluczanie piszących, jest również zmieniany przez pierwszego i ostatniego czytającego.

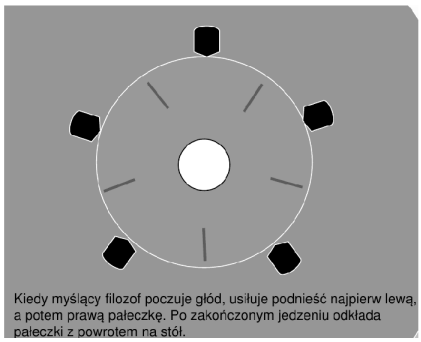
- Proces zapisujący:

```
czekaj(pisanie);
…
zapis
…
sygnalizuj(pisanie);
```

- Proces czytający:

```
czekaj(wyklucz); //tylko 1 proces może działać w tej sekcji!
liczba_czyt++;
if (liczba_czyt==1)
   { czekaj(pisanie); } //może być wewnątrz piszący!
sygnalizuj(wyklucz);  //mogą wchodzić inni czytający!
…
czytanie
czekaj(wyklucz); //znowu sekcja wyłączna
liczba_czyt--;
if (liczba_czyt==0)
   { sygnalizuj(pisanie); } //może ewentualnie wejść piszący
sygnalizuj(wyklucz);
```


Problem ucztujących filozofów



Kiedy myślący filozof poczuje głód, usiłuje podnieść najpierw lewą, a potem prawą pałeczkę. Po zakończonym jedzeniu odkłada paleczki z powrotem na stół.


```
semaphore paleczka [4];

while 1
{
    czekaj(paleczka[i]);
    czekaj(paleczka[i+1 %5]);
    …
          jedzenie();
    sygnalizuj(paleczka[i]);
    sygnalizuj(paleczka[i+1 %5]);
    …
          myslenie();
}
```


Blokada

- A co jeśli każdy z filozofów w jednym momencie poczuje głód i podniesie lewą pałeczkę?
- Zapobieganie:
  - Zostawić jedno wolne miejsce,
  - Pozwolić na podniesienie pałeczki tylko gdy obie są dostępne (sprawdzanie i podnoszenie w sekcji krytycznej),
  - Rozwiązanie asymetryczne: Nieparzysty filozof podnosi najpierw lewą pałeczkę, parzysty zaś prawą.

Region krytyczny

- Jest to konstrukcja służąca do synchronizacji w językach wyższego poziomu.
- Przykładowa składnia:

  region V **when** B **do** S;

  gdzie:

  V – zmienna współdzielona

  Podczas wykonywania instrukcji **S** żaden inny proces nie ma prawa dostępu do zmiennej **V**.

  Jeśli warunek **B** jest prawdziwy, proces może wykonać instrukcję **S**, w przeciwnym wypadku czeka na zmianę **B** oraz na opuszczenie sekcji krytycznej przez inne procesy.

Implementacja

```
var bufor: shared record
          magazyn: array [0..n-1] of jednostka
          licznik,we,wy:integer;
          end;
region bufor when licznik <n do begin
      magazyn[we]:=nast_p;
      we:=we+1 mod n;
      licznik:=licznik+1;
    end;
region bufor when licznik >0 do begin
       nast_k:=magazyn[wy];
       wy:=wy+1 mod n;
        licznik:=licznik-1;
     end;
```

Synchronizacja w systemie Solaris

- Ze względu na implementację procesów czasu rzeczywistego, wielowątkowość i obsługę wielu procesorów, synchronizacja za pomocą sekcji krytycznych nie znalazła zastosowania.
- Zastosowano **zamki adaptacyjne**.

- Zamek rozpoczyna działalność jak standardowy semafor. Jeśli dane są już w użyciu, to zamek wykonuje jedną z dwu czynności:
  - Jeśli zamek jest utrzymywany przez wątek aktualnie wykonywany, to inny wątek ubiegający się o zamek będzie czekać (gdyż aktywny wątek niedługo się zakończy),
  - Jeśli zamek jest utrzymywany przez wątek nieaktywny, to wątek żądający zamka blokuje się i usypia, gdyż czekanie na zamek będzie dłuższe.

Synchronizacja w systemie Solaris

- Zamki adaptacyjne stosuje się, gdy dostęp do danych odbywa się za pomocą krótkich fragmentów kodu (zamknięcie na czas wykonywania co najwyżej kilkuset rozkazów).
- W przypadku dłuższych segmentów kodu stosuje się **zmienne warunkowe**.
- Jeśli zamek jest zablokowany, to wątek wykonuje operację **czekaj** i usypia. Wątek zwalniający zamek sygnalizuje to następnemu z kolejki uśpionych co tamtego budzi.
- **Blokowanie zasobów w celu pisania lub czytania** jest wydajniejsze niż używanie semaforów, ponieważ dane mogą być czytane przez kilka wątków równocześnie, a semafory dają tylko indywidualny dostęp do danych.

Synchronizacja w systemie Linux

- W nowszych wersjach jądra Linux występuje mechanizm synchronizacji zwany **futex** (Fast Userspace muTEX)**.**
- Proces w przestrzeni użytkownika widzi futex jako zmienną (z reguły 32-bit) dostępną przez szybkie, niepodzielne operacje nie wymagające długotrwałych funkcji systemowych podczas czekania.
- Funkcje systemowe używane są gdy rezerwujemy dostęp. Wątek, który potrzebuje dostępu do sekcji krytycznej używa **WAIT(adres,wartosc)**. Taki wątek jest zasypiany jeżeli wartość futexu spod adresu addr jest równa val.
- Gdy inny wątek wykona właściwą pracę, woła **WAKE(adres,wartosc)**. Wówczas wartość jest dekrementowana a jądro przesuwa czekające wątki do kolejki procesów gotowych by kontynuowały pracę.

Futex i więcej wątków

- A co gdy czeka więcej wątków?
- Zjawisko „**owczego pędu”** (thundering herd) - po zwolnieniu zasobów wybudzane jest wiele, wiele wątków i nie wiemy który pierwszy wejdzie do swej sekcji krytycznej.
- Jądro systemu obsługuje **kolejki oczekujących wątków.** Można realizować taką czynność wykorzystując specyficzne wartości **val**. Albo użyć funkcji przenoszących wątki do innych futeksów.

- **REQUEUE -** przyjmuje dwa futeksy i ilość wątków. Jeżeli wartość przechowywana w futeksie jest równa zadanej, budzi podają liczbę wątków, a zadaną część pozostałych przesuwa do nowego futeksu minimalizując zjawisko wybudzania dużej ilości wątków jednocześnie.
- **WAKE_OP -** Budzi wątki w zależności od wyniku zadanej w argumencie operacji. Użyteczne do implementacji zależnych od warunku technik synchronizacji.

Futex - krótka historia

- W Linuksie obecne od dawna, lecz API nie było stabilne.
- Futex2 - zaproponowane przez Valve - możliwości separowania zadań, różne długości zmiennej, warunkowe budzenie procesów.
- W Linuksie stabilne od wersji ok. 5.16 - 2022 r.

Bariera pamięci

- A co jeżeli zmiana kolejności operacji spowoduje szkodliwą rywalizację? Czy stosować kompletne semafory, futeksy, regiony krytyczne?
- Potrzebny jest niskopoziomowy mechanizm, który wymusi kolejność wykonywania operacji.

  Takim rozwiązaniem jest **bariera pamięci**.
- Wymusza konkretną kolejność wykonywanych operacji. **Żeby wykonywać dalej instrukcje, wszystkie wątki muszą wykonać instrukcję bariery.**
- Dzięki temu przy wielu procesorach nie dojdzie do szkodliwej rywalizacji nawet gdy optymalizacja dostępu do pamięci czy po prostu zmiana obciążenia rdzeni próbowałaby zmienić kolejność.
- We współczesnych systemach jest niskopoziomową instrukcją procesora.

Rodzaje barier i ich zastosowania

- **Bariera zapisu** - daje pewność, że operacje zapisu przed barierą zostaną wykonane przed operacjami po wystąpieniu instrukcji bariery.
  - Przydatne gdy któryś rdzeń uparcie buforuje wartość zmiennej w rejestrach.
- **Bariera zależności danych** - Jeżeli popełniamy dwa odczyty, a drugi zależy od wyniku pierwszego (np. drugi korzysta z adresu w pamięci zaczytanego w wyniku pierwszego odczytu), to zastosowanie tej bariery wymusza aktualizację czytanej wartości (np. z innych rdzeni). Inne rdzenie nie odłożą w czasie aktualizacji pierwszej wartości.
- **Bariera odczytu** - Jak wyżej, ale dodatkowo gwarantuje, że wszystkie odczyty sprzed bariery zostaną wykonane przed odczytami po jej użyciu.
- **Bariera ogólnego przeznaczenia** - działa zarówno na operacje odczytu i zapisu - to, co przed barierą wykona się najpierw. Stosowana przy obliczeniach równoległych.

Kiedy stosujemy bariery?

- Kiedy istnieje możliwość, że pojawią się niepożądane interakcje przy pracy dwóch rdzeni, lub rdzenia i urządzenia wejścia-wyjście, nad jednym miejscem w pamięci.
- Kiedy przerwanie (np. to od przełączania kontekstu) spowoduje zmianę alokacji rdzenia przy danym procesie - bariera pamięci umożliwia podjęcie pracy przez drugi rdzeń na aktualnych danych z pamięci. Bez niej aktualne dane pozostałyby w cache lub rejestrach rdzenia, któremu odebrano zadanie.
  - Taki błąd powodował, że w konsoli Nintendo Switch oprogramowanie sprzed wersji 14.0.0 powodowało w niektórych grach "glitche". Komunikacja z GPU odbywa się w dużej mierze z użyciem cache.

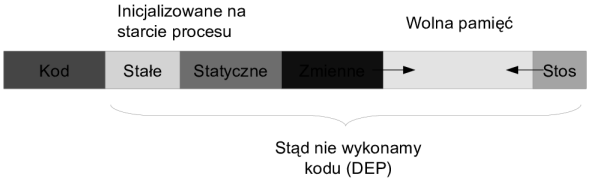
Monitor

- Problem z semaforami:
  - Trudno analizować programy współbieżne i przewidzieć wszelkie możliwe układy rozkazów procesora (przy okazji wychodzą błędy typu SPECTRE).
- **Monitor** stanowi połączenie modułu programistycznego z sekcją krytyczną i jest po prostu modułem zawierającym deklaracje stałych, zmiennych, funkcji i procedur.
- Wszystkie te obiekty, z wyjątkiem jawnie wskazanych funkcji i procedur **są lokalne** w monitorze i nie są widoczne na zewnątrz niego. Wskazane funkcje i procedury (tzw. eksportowane) są widoczne na zewnątrz monitora. Mogą je wywoływać procesy i za ich pośrednictwem manipulować danymi ukrytymi w monitorze.
- Monitor zawiera też kod, który służy do jego inicjacji, na przykład do ustawienia wartości początkowych zmiennych deklarowanych w monitorze.

Monitor (c.d.)

- Co najwyżej jeden proces może być w trakcie wykonania kodu znajdującego się w monitorze.
  Jeśli jakiś proces wywoła procedurę eksportowaną przez monitor i rozpocznie jej wykonanie, to do czasu powrotu z tej procedury żaden inny proces nie może rozpocząć wykonania tej ani żadnej innej procedury/ funkcji monitora.
- O procesie, który wywołał funkcję/procedurę monitora i nie zakończył jeszcze jej wykonania, będziemy mówić, że znajduje się w monitorze. Zatem jednocześnie w monitorze może przebywać co najwyżej jeden proces. Taka definicja narzuca naturalny sposób korzystania ze zmiennych współdzielonych przez procesy: Należy umieścić je w monitorze i dostęp do nich realizować za pomocą eksportowanych procedur i funkcji.
- Programista korzystający w taki właśnie sposób ze zmiennej dzielonej nie musi myśleć o zapewnianiu wyłączności w dostępie do niej - robi to za niego automatycznie sam monitor.

Budowa procesu w pamięci



- Jeżeli bardzo się postaramy, możemy nadpisać stos danymi.
- Kończy się to zazwyczaj błędem „segmentation fault”.

Proces poza kontrolą

- Czy możemy wykonać własny kod na uprawnieniach procesu?
  - Pod kontrolą mamy tylko obszar danych, z którego nie wykonamy żadnego kodu (DEP).
- Co dzieje się podczas wywołania funkcji?
  - Zapisywane są wartości rejestrów i ewentualnie dane pomocnicze.
  - Wskaźnik umieszczany jest **na stosie** (który możemy nadpisać).
  - Następuje skok do funkcji.
- Po funkcji:
  - Następuje zapisanie ewentualnych wyników.
  - **Zdjęcie wartości ze stosu** wywołań i skok powrotny.

Proces poza kontrolą

Jeżeli więc tylko program ma jakikolwiek problem umożliwiający modyfikację jego pamięci, możemy więc **nadpisać mu stos wywołań** i uruchomić dowolny fragment programu na prawach takiego procesu!

- Czy musimy modyfikować kod programu? Niekoniecznie. Polecenia uruchamiające funkcje systemowe, programy, modyfikujące dane - one już są! Musimy je **znaleźć** w obrazie procesu.
- Jak to zrobić?
  - Zmusić program do nadpisania własnego stosu naszym zestawem wywołań,
  - Nadpisać stos wywołań tak, by najbliższy powrót uruchomił inny kod.

Jak to się robi?

- Deasemblowanie programu, w którym szukamy podatności.
- Obejrzenie wszystkiego, co **poprzedza** instrukcję powrotu.
- Znalezienie użytecznych „gadżetów" kończących się pop-nięciem kolejnego adresu ze stosu:
  - Zapis danych z obszaru danych do rejestrów,
  - Wykonanie funkcji systemowej,
  - Przepisywanie danych do obszarów współdzielonych
  - Itd…
- To są **rozkazy** którymi **piszemy nasz kod.**

Jak to się robi? (2)

- Nadpisujemy pamięć programu tak, by nasz kod oparty o powroty dokonkretnych miejsc nadpisał stos (ale nic dalej! - nie chcemy segmentation fault).
- Wymuszamy pierwszy powrót z funkcji (np. kończy się funkcja otwierania pliku).
- Dalej „wraca" się samo.

- Jak nadpisać stos?
  - Przeglądarka obrazków zapisująca do pamięci dane obrazka bez sprawdzenia ich długości,
  - Silnik JavaScript alokujący tablice „na ślepo",
  - Edytor tekstu zapisujący dane dokumentu bez sprawdzenia czy pamięc nie nadpisuje stosu...

Proces poza kontrolą

- Jakie procesy są podatne?
  - Przeglądarki internetowe
  - Środowiska programowania
  - Programy wieloplatformowe z wirtualizacją
  - Itd…. Itd….

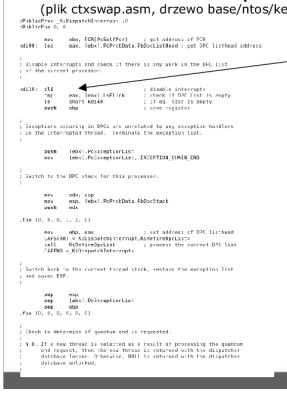
- Tą podatność można łączyć z wykonaniem spekulatywnym (BlindSide, listopad 2020) i nadpisywać stos wywołań kodem, który teoretycznie w ogóle nie powinien się wykonać!

Proces poza kontrolą...

- Jak rozwiązać problem?
  - ASLR - pseudolosowe ustawienie bloków programu w pamięci - trudniej przewidzieć gdzie wyląduje nasz mały kod („gadget") i gdzie ustawić pozycję na stosie wywołań.
    Problem: Jeżeli sprzęt tego nie wspiera, potrzebujemy sprawny generator pseudolosowy.
  - Retpoline - Kontrolowany powrót z funkcji - umożliwia sprawdzenie na poziomie systemu czy nie doszło do naruszenia struktury wykonawczej programu.
    Problem: Co powrót z funkcji praktycznie wołamy funkcje jądra - spadek wydajności.

Kernel Windowsa NT

- Problem został naprawiony tak:
  (plik ctxswap.asm, drzewo base/ntos/ke/$ARCHITECTURE$)



Przełączenie kontekstu możliwe przy powrocie
(tu blokujemy przerwania by się do niego przygotować)

Kernel Windowsa NT

- Problem został naprawiony tak:
  (plik ctxswap.asm, drzewo base/ntos/ke/$ARCHITECTURE$)

```
.fpo (0, 0, 0, 3, 1, 0)

kdi00:  sub     esp, 3*4
        mov     [esp+0], esi            ; save registers
        mov     [esp+4], edi
        mov     [esp+8], ebp
        mov     esi, eax                ; set  next thread address
        mov     edi, febel.PcPrcbData.PbCurrentThread ; get current thread address
        mov     dword ptr [edx].PcFrcbData.PbNextThread, 0 ; clear next thread address
        mov     [edx].PCPrcbData.PtCurrentThread, esi ; set current thread address
        mov     ebp, edi                ; set address of current thread
        mov     byte ptr [edi].ThIdleSwapBlock, 1
        fstCall KiReadyThread           ; ready thread for execution
        CAPSTART ~_KiDispatchInterrupt,SwapContext~
        mov     cl, 1                   ; set NPC interrupt bypass disable
        call    _SwapContext            ; swap context
        CAPTURE ~_KiDispatchInterrupt~
        mov     esp, [esp+0]            ; restore registers
        mov     edi, [esp+4]
        mov     esi, [esp+8]
        add     esp, 3*4
kdi70:  stdRET  _KiDispatchInterrupt    ; return

  . . . . .
```

Zapisujemy rejestry i niezbędne składniki do przełączenia kontekstu...

Następnie porównujemy stan instrukcji powrotu - jeżeli OK - wracamy. Jeżeli nie - proces kończy się z błędem.

PROBLEM: Ataki „speculative" wciąż są możliwe. Rozwiązano to przez ASLR + „drabinka" wywołań.

Jednakże...

- Jeżeli program, który atakujemy korzysta z funkcji systemowych umożliwiających dynamiczne zarządzanie pamięcią, można ograniczyć adres w ramach ASLR do kilkunastu-kilkudziesięciu bitów, a potem to zbruteforce'ować.
  - Atak trwa 40-90 sekund na Core 2 Duo 1.33GHz
- Jeżeli system wkracza na rozkaz „return", wiedząc, że rozkaz „return" to:
  - Znalezienie adresu na szczycie stosu wywołań
  - Zapis do IP tej wartości
  - Powiększenie wartości wskaźnika stosu o rozmiar adresu

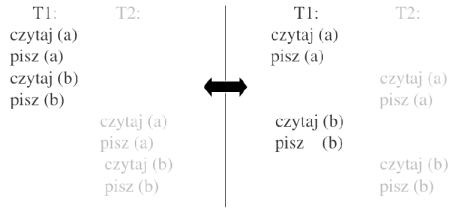
  Można to emulować za pomocą skoku do wartości zamiast ustawiania IP.
- Return Oriented Programming jest jedną z bardziej zaawansowanych technik wykorzystywania podatności w programach.

Transakcje

- **Transakcją** jest zbiór operacji stanowiących logicznie spójną funkcję. Jest to na przykład ciąg operacji czytania lub pisania zakończonych operacją zatwierdzenia lub zaniechania.
- Transakcja zaniechana nie powinna pozostawić śladów w danych, które zdążyła już zmienić.
- Wycofanie transakcji powinno zapewnić odtworzenie stanu sprzed transakcji.
- Spójność danych i ich sprawne odzyskiwanie po np. awarii systemu jest najważniejszą sprawą w bazach danych różnego rodzaju. Jest to też rodzaj synchronizacji danych.

Szeregowanie transakcji

- **Szeregowanie transakcji** polega na takim zaplanowaniu wzajemnego przeplatania się rozkazów z kilku transakcji, aby nie występowały konflikty pisania/czytania tych samych danych



Transakcje - protokół blokowania

- Z każdym obiektem danych kojarzony jest **zamek**, od którego zależy dostęp do danych, np.
  - Jeżeli transakcja dostaje dostęp do obiektu danych w **trybie wspólnym**, to może czytać ten obiekt, ale nie może go zapisywać.
  - Jeśli dostaje obiekt w **trybie wyłącznym**, to wolno jej zarówno czytać jak i zapisywać ten obiekt.
- Każda transakcja musi zamawiać zamek blokujący obiekt w takim trybie, aby mogła wykonać zaplanowaną operację.

Rejestr - odzyskiwanie

- Rejestr to zapis w pamięci trwałej, określający wszystkie zmiany w danych wykonywane podczas transakcji.
  Każdy rekord w rejestrze (logu) zawiera następujące dane:
  - nazwa transakcji,
  - nazwa jednostki danych,
  - stara wartość,
  - nowa wartość,
  - inne dane dotyczące transakcji, np. zaniechanie
- Zanim rozpocznie się wykonywanie transakcji, w rejestrze zapisuje się rekord informujący o rozpoczęciu transakcji.
- Każdy zapis (przez transakcję) poprzedzony jest zapisem odpowiedniego rekordu w rejestrze.
- Gdy dochodzi do zatwierdzenia transakcji, w rejestrze zapisuje się rekord zatwierdzenie.

Rejestr - odzyskiwanie

- Tworzenie rejestru jest pamięcio- i czasochłonne, ale dla bardzo ważnych danych nie jest to cena wygórowana.
- Przy rekonstrukcji danych na podstawie rejestru korzysta się z dwóch procedur:
  - **wycofaj** - odtwarza wszystkie dane uaktualnione przez transakcję T, nadając im stare wartości,
  - **przywróć** - nadaje nowe wartości wszystkim danym uaktualnionym przez transakcję T.
- Transakcja musi być wycofana, jeśli w rejestrze znajduje się rekord rozpoczęcia, a nie ma rekordu zatwierdzenia.
- Transakcja musi być przywrócona, jeśli w rejestrze jest rekord rozpoczęcie oraz rekord zatwierdzenie dla danej transakcji.

Punkty kontrolne

- Odtwarzanie za pomocą rejestru ma pewne wady:
  - Proces przeglądania rejestru jest czasochłonny,
  - Większość transakcji zapisanych w rejestrze odbyła się pomyślnie przed awarią, odtwarzanie ich z rejestru jest dublowaniem pracy.
- Dla przyspieszenia ewentualnego odtwarzania, system organizuje co jakiś czas tzw. punkty kontrolne, w których:
  - Wszystkie rekordy pozostające w tej chwili w pamięci operacyjnej są zapisane w pamięci trwałej (na dysku),
  - Wszystkie zmienione dane, pozostające w pamięci ulotnej, muszą być zapisane w pamięci trwałej,
  - W rejestrze transakcji zapisuje się rekord punkt kontrolny Po awarii przegląda się rejestr od końca.
- Po napotkaniu rekordu punkt kontrolny, przywracanie rozpoczyna się od pierwszej transakcji po nim.

Dziękuję za uwagę