Przełączanie kontekstu w systemie Linux, wersja 2.2.17, na platformie Intel IA32. *

Radosław Pilski

20 grudnia 2000

1 Wstęp

W niniejszym opracowaniu przedstawione jest zagadnienie przełączania kontekstu w systemie operacyjnym Linux, w jądrze 2.2.17. Opis dotyczy realizacji przełączaia kontekstu w 32-bitowej architekturze firmy Intel (w skrócie IA32).

2 Kontekst zadania w architekturze IA32

W architekturze IA32 kontekst zadania (ang. *task*) określony jest przez zawartość pamięci w przestrzeni adresowej procesu oraz przez zawartość rejestrów procesora. Procesory w architekturze IA32 posiadają specjalny **rejestr zadania**: **TR** (ang. *Task Register*). W TR znajduje się selektor segmentu zawierającego strukturę danych określaną jako **segment stanu zadania**: **TSS** (ang. *Task State Segment*). W TSS przechowywane są dane określające kontekst działania zadania:

- zawartość rejestrów ogólnego przeznaczenia (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI)
- zawartość rejestrów segmentowych (CS, DS, ES, SS, FS, GS)
- zawartość rejstru flagowego (EFLAGS)
- wartość wskaźnika instrukcji (EIP)
- selektor TSS poprzedniego zadania
- selektor lokalnej tablicy deskryptorów zadania (LDT)
- fizyczny adres tablicy stron zadania (zawartość rejestru CR3)
- wskaźniki stosu dla poziomów uprzywilejowania od 0 do 2, każdy taki wskaźnik składa się z selektora segmentu (SS0, SS1, SS2) oraz przesunięcia w obrębie segmentu (ESP0, ESP1, ESP2)

*Projekt “Algorytmy + Struktury Danych = Linux”, temat numer 4

- flaga T (pułapka śledzenia, ang. *debug trap*) (przełączenie się do zadania, które ma ustawioną tą flagę powoduje wygenerowanie wyjątku)
- mapa dostępnych przerwań wejścia/wyjścia (określa które porty wejścia/wyjścia są dostępne dla danego procesu)
- mapa przekierowania przerwań

W architekturze IA32 Istnieje możliwość sprzętowego przełączenia kontekstu. Rozwiązanie takie było stosowane we wcześniejszych wersjach jądra. Obecnie jądro używa rozszerzonej struktury TSS do przechowywania kontekstu procesu, samo przełączanie kontekstu realizowane jest jednak programowo. W Linuksie z każdym procesem związana jest struktura `struct task_struct` `[include/linux/sched.h]`. Zawiera ona pole `tss` typu `struct thread_struct`. Struktura `struct thread_struct` odpowiada TSS danego procesu, rozszerzonemu o pola przechowujące zawartość rejestrów śledzenia (ang. *debug registers*), zawartość rejestrów koprocesora oraz informację o trybie wirtualnym v86.

3 Okoliczności, w których dochodzi do przełączenia kontekstu

Przełączenie kontekstu następuje gdy bieżący proces sam zwalnia procesor lub gdy moduł szeregujący stwierdzi, że procesor powinien zostać odebrany aktualnie wykonywanemu procesowi i przydzielony innemu procesowi.

Sytuacje w jakich dochodzi do przełączenia kontekstu mogą być następujące:

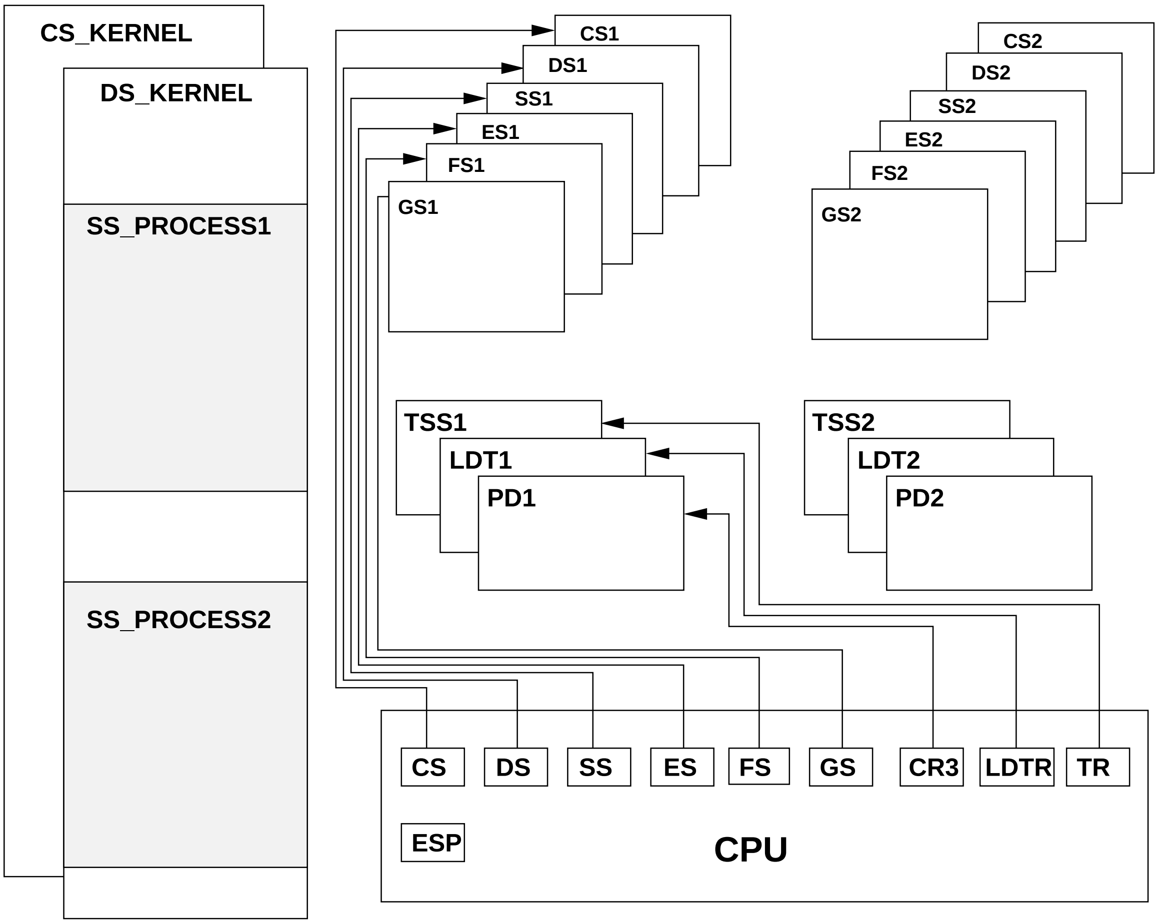
- pojawienie się procesu o priorytecie wyższym niż priorytet bieżącego procesu
- zaśnięcie procesu w oczekiwaniu na zdarzenie systemowe
- wykorzystanie przez bieżący proces przydzielonego mu kwantu czasu procesora
- zakończenie działania procesu

Sprawdzenie, który proces powinien otrzymać procesor jest realizowane przez procedurę `schedule()` `[kernel/sched.c]`. Jest ona wywoływane przed powrotem procesu z przerwania lub funkcji systemowej jeśli globalna zmienna jądra `need_resched` ma wartość niezerową. Jeśli moduł szeregujący ustali, iż procesor powinien zostać przydzielony innemu procesowi niż bieżący, to następuje przełączenie kontekstu. Polega ono na zapisaniu w odpowiednich strukturach systemowych kontekstu bieżącego procesu, odtworzeniu kontekstu procesu, który ma otrzymać procesor, a następnie wznowienie działania tego procesu w miejscu, w którym zostało ono uprzednio przerwane. Dalsza część opracowania zawiera dokładniejszy opis samego procesu przełączania kontekstu.

4 Procedura przełączania kontekstu

Poniżej przedstawiony zostanie przykładowy scenariusz przełączenia kontekstu. W przykładzie tym rozważane są dwa procesy `P1` i `P2`. Bieżącym procesem jest proces `P1`. Wykonywanie procesu `P2` zostało wcześniej wstrzymane i czeka on na wznowienie wykonywani. Proces `P1` wywołuje funkcję systemową. Podczas tego wywołania zarządca procesów decyduje, iż proces P2 powinien zostać wznowiony. Następuje przełączenie kontekstu i wznowienie procesu P2.

Sytuację wyjściową, zaraz przed wyołaniem funkcji systemowej przez proces P1 przedstawia Rysunek 1.



Rysunek 1: Przełączanie kontekstu. Stan systemu przed wywołaniem funkcji systemowej przez proces P1

Na rysunku przedstawione są schematycznie niektóre elementy kontekstu procesów P1, P2 oraz jądra. Prostokąty CS_KERNEL oraz DS_KERNEL reprezentują odpowiednio segment kodu i segment danych jądra. Każdy proces w systemie posiada w segmencie danych jądra swój własny stos używany podczas wywoływania przez dany proces funkcji systemowych oraz podczas obsługi przerwań przychodzących podczas działania danego procesu. Na rysunku prostokąty SS_PROCESS1 oraz SS_PROCESS2 reprezentują stos w segmencie danych jądra odpowiednio dla procesu P1 i procesu P2.

Prostokąty CS1, DS1, SS1, ES1, FS1, GS1 reprezentują segmenty pamięci używane przez proces

P1. Linie prowadzące od nazw rejestrów segmentowych procesora do prostokątów reprezentujących segmenty oznaczają, że w danym rejstrze segmentowym znajduje się selektor odpowiedniego segmentu. Prostokąty `TSS1`, `LDT1`, `PD1` reprezentują odpowiednio segment TSS procesu P1, lokalną tablicę deskryptorów procesu `P1` i katalog stron procesu `P1`. Analogiczne prostokąty z końcowym numerem 2 reprezentują elementy kontekstu procesu `P2`.

W momencie wywołania funkcji systemowej przez proces `P1` następuje przejście z trybu pracy użytkownika do trybu jądra. Przejściu takiemu towarzyszy przełączenie stosów. Z segmentu TSS bieżącego procesu pobierany jest do rejestru SS selektor stosu bieżącego procesu podczas pracy w tryb jądra. Na tym nowym stosie odkładany jest adres (zawartość rejestrów SS i ESP) stosu procesu sprzed wywołania procedury obsługi przerwania. Następnie odkładany jest adres powrotu z procedury obsługi przerwania (zawartość rejestrów CS i EIP) oraz rejestr znaczników (EFLAGS). Zarówno przełączenie stosów, jak i zapamiętanie adresu powrotu jest realizowane sprzętowo.

Następnie rozpoczyna się wykonanie procedury obsługi wywołania systemowego (etykieta `system_call` w pliku `arch/i386/kernel/entry.S`). Najpierw zapamiętywana jest na stosie zawartość rejestru EAX. Następnie wywoływane jest makro `SAVE_ALL`, które umieszcza na stosie wszystkie rejestry ogólnego przeznaczenia oraz rejestry segmentowe ES i DS. Następnie do rejestrów ES i DS ładowany jest selektor segmentu danych jądra.

Stan systemu w tym momencie przedstawia Rysunek 2. Prostokąt `SS1:ESP1` oznacza odłożony na stosie adres stosu procesu `P1`. Prostokąt `CS1:EIP1` reprezentuje adres powrotu z funkcji systemowej, a prostokąt `REGS1` reprezentuje wartości rejestrów procesora odłożone na stos przed wywołaniem właściwej funkcji systemowej. Rejestr CS wskazuje na segment kodu jądra, a segmenty DS, ES i SS wskazują na segment danych jądra. Wskaźnik stosu ESP wskazuje na szczyt stosu procesu `P1` w segmencie danych jądra.

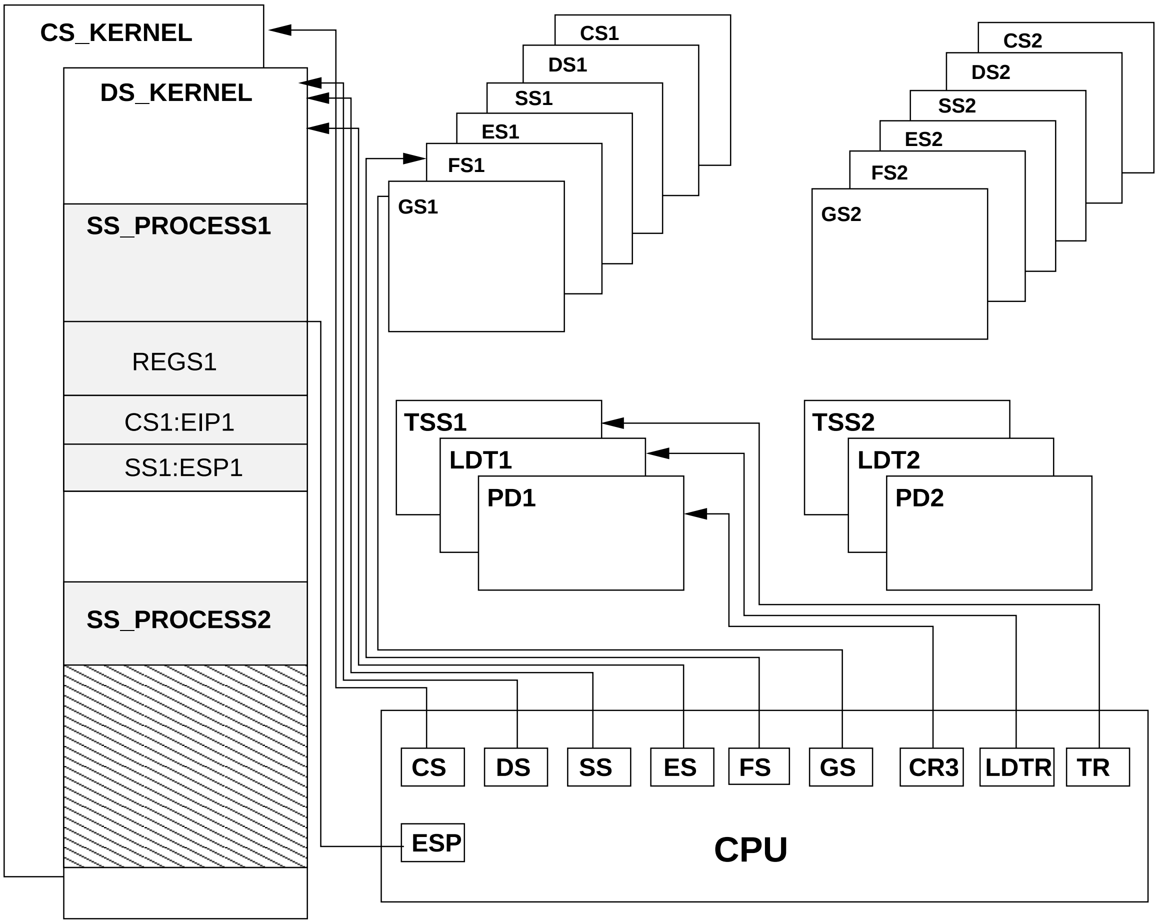
Dalej następuje skok do odpowiedniej funkcji systemowej. Po powrocie z tej funkcji wykonywanie przechodzi do etykiety `ret_from_sys_call`. Tam sprawdzane jest, czy globalna zmienna `need_resched` ma wartość niezerową. Jeśli tak to wywoływana jest procedura szeregująca `schedule`. Procedura szeregująca wybiera proces, który powinien otrzymać procesor.

Tak jak to wcześniej założyliśmy, w naszym przykładzie procesem wybranym przez procedurę `schedule` jest proces P2. Następuje więc przełączenie kontekstu.

Najpierw zwiększany jest licznikprzełączeń kontekstu (`kstat.context_swtch`).

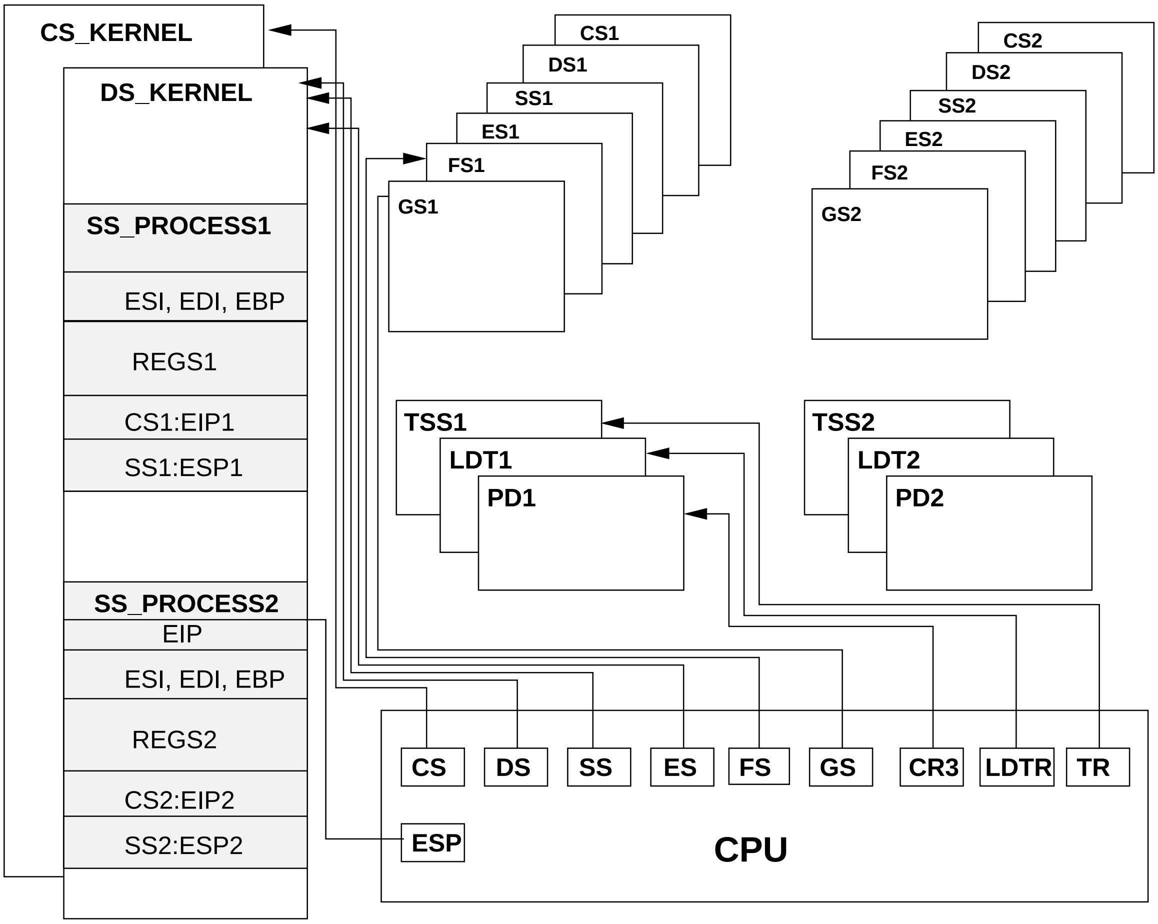
Następnie wywoływane jest makro `switch_to [include/asm-i386/system.h]` gdzie dokonuje się zasadnicze przełączenie kontekstu procesów według następującego schematu:

- Zapisywane są na stosie bieżącego procesu rejestry ESI, EDI i EBP.
- Następuje przełączenie stosów. W TSS bieżącego procesu zapamiętywana jest bieżąca wartość rejestru ESP. W naszym przykładzie do TSS1 zapisana zostanie wartość wskazująca na szczyt stosu `SS_PROCESS1` . Z TSS procesu, który ma otrzymać procesor odtwarzana jest wartość rejestru ESP wskazująca na szczyt stosu tego procesu w segmencie danych jądra. W naszym przykładzie do ESP zostanie załadowana wartość wskazująca na szczyt stosu `SS_PROCESS2` .
- W polu TSS procesu `P1` przechowującym wartość rejestru EIP zapisywany jest adres procedury, która powoduje odtworzenie ze stosu umieszczonych tam wartości rejestrów ESI, EDI oraz EBP, a następnie powrót do procedury `schedule`.
- Na stosie odkładana jest wartość EIP zapamiętana w segmencie `TSS2` (segmencie TSS procesu `P2`). Są dwie możliwe wartości EIP jakie mogą być zapisane w segmencie TSS procesu. Jeżeli



Rysunek 2: Przełączanie kontekstu. Stan po zmianie trybu pracy na tryb jądra i zapamiętaniu na stosie rejestrów procesora.

proces nie był jeszcze wykonywan to wartością EIP zapisaną w jego segmencie TSS jest adres etykiety `ret_from_fork [arch/i386/kernel/entry.S]`. Adres tej etykiety zapisywany jest w TSS procesu zaraz po jego utworzeniu przez funkcję systemową `fork()`. Jeżeli proces był już wykonywany to znaczy, że procesor został mu odebrany w procedurze `schedule()`. W tym przypadku zapisana wartość EIP jest adresem opisanej w poprzednim punkcie procedury, która powoduje odtworzenie ze stosu umieszczonych tam wartości rejestrów ESI, EDI oraz EBP, a następnie powrót do procedury `schedule`. W naszym przykładzie założymy ten drugi przypadek, czyli że proces `P2` był już wykonywany. Stan systemu w tym momencie przedstawiony jest na Rysunku 3. Wskaźnik ESP wskazuje na stos procesu `P2` w segmencie danych jądra. Na rysunku przedstawiona jest też zawartość tego stosu. Dane na stosie procesu `P2` zostały umieszczone w trakcie wywołania funkcji systemowej lub przerwania, podczas którego procesowi `P2` został odebrany procesor.

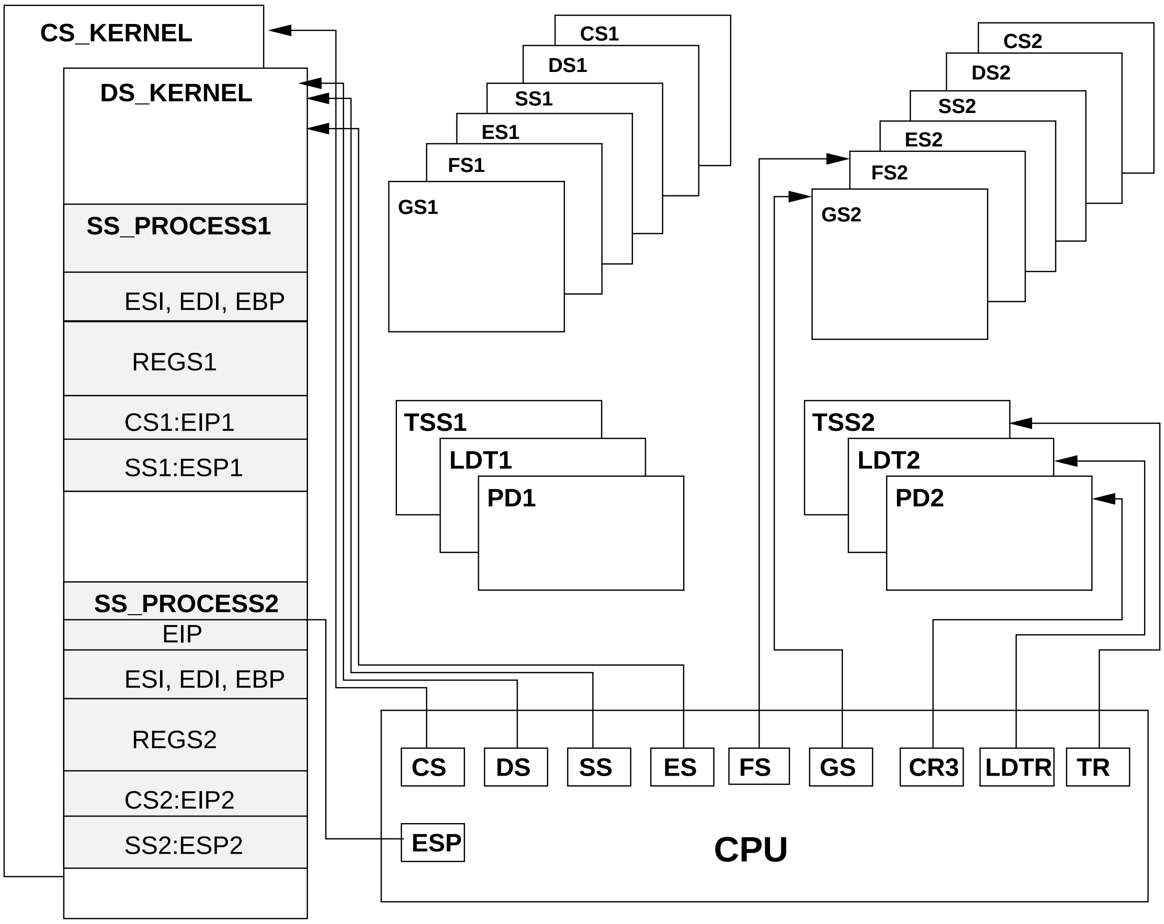


Rysunek 3: Przełączanie kontekstu. Stan po przełączeniu stosów procesów `P1` i `P2`.

- Następnie wykonywany jest skok do procedury `__switch_to(struct task_struct *prev, struct task_struct *next) [arch/i386/kernel/process.c]` , która przełącza pozostałe elementy kontekstu.
  - Najpierw zapisywany jest stan koprocesora matematycznego.
  - Następnie zerowany jest bit zajętości (ang. *busy*) w deskryptorze segmentu TSS procesu, który otrzymuje procesor (nowego procesu). Deskryptor ten znajduje się w tablicy GDT. Bit zajętości oznacza, że dane zadanie jest wykonywane.
  - Do rejestru TR ładowana jest wartość zapamiętana w segmencie TSS nowego procesu.
  - Zawartość rejestrów FS i GS jest zapamiętywana w segmencie TSS starego procesu (procesu, który traci procesor). Wartości tych rejestrów nie zostały zapamiętane na stosie tak jak wartości rejestrów DS i ES, ponieważ jądro nie korzysta z segmentów FS i GS.

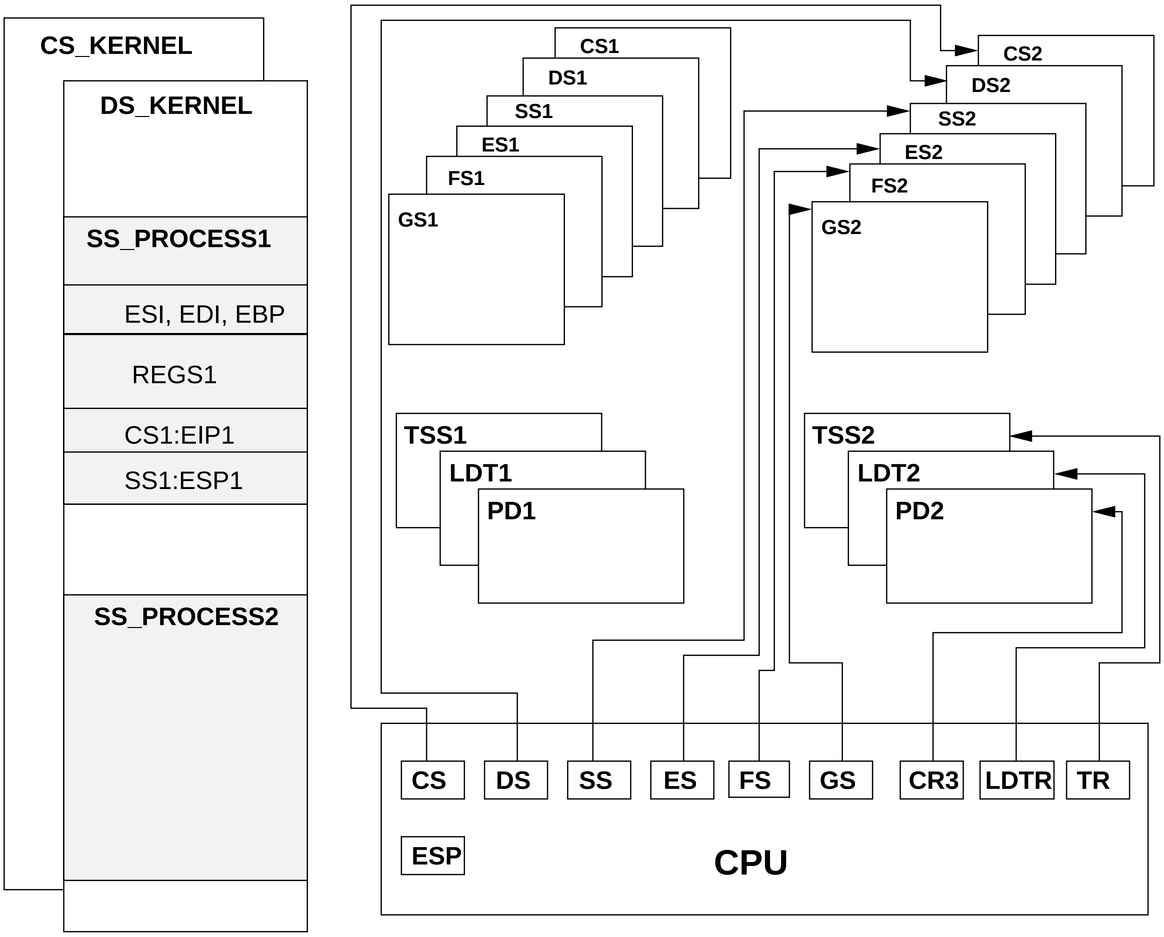
- Z segmentu TSS nowego procesu odtwarzane są wartości rejestrów LDT, CR3, FS i GS
- Jeżeli nowy proces w momencie przerwania jego wykonywania pracował z włączoną opcją śledzenia (ang. *debug*) to odtwarzane są z jego segmentu TSS wartości rejestrów śledzenia (ang. *debug registers*).

Po powrocie z procedury `__switch_to` następuje odtworzenie zapamiętanych na stosie wartości rejestrów ESI, EDI oraz EBP oraz powrót do procedury `schedule` już w kontekście nowego procesu (Rysunek 4).



Rysunek 4: Przełączanie kontekstu. Stan po przełączeniu kontekstu, ale przed wznowieniem wykonywania procesu `P2`.

Następnie makro `RESTORE_ALL [arch/i386/kernel/entry.S]` odtwarza zapisane na stosie rejestry i następuje wznowienie wykonywania procesu `P2` (Rysunek 5).



Rysunek 5: Przełączanie kontekstu. Stan po wznowieniu wykonywania `P2`.

5 Słownik niektórych pojęć i skrótów dotyczących architektury IA32

deskryptor segmentu struktura danych opisująca segment pamięci. Na opis składają się: adres początku oraz rozmiar segmentu oraz informacje opisujące kontrolę dostępu do danego segmentu.

GDT (ang. *Global Descriptor Table*) globalna tablica deskryptorów, jest globalną (używaną przez wszystkie procesy) tablicą deskryptorów segmentów .

LDT (ang. *Local Descriptor Table*) lokalna tablica deskryptorów, jest tablicą deskryptorów segmentów, przy czym każde zadanie może mieć swoją własną LDT.

segment pamięci jest pewnym obszarem pamięci. Segment pamięci określony jest przez adres jego początku oraz przez jego rozmiar. Dla każdego segmentu określone jest dozwolony tryb dostępu do danych w tym segmencie. W architekturze IA32 określone są dwa tryby dostępu do danych w segmencie: tylko do odczytu oraz do odczytu i zapisu. Podział pamięci na segmenty umożliwia sprzętową ochronę pamięci procesu przed niepowołanym dostępem do niej z innych działających w systemie procesów.

selektor jest specjalnym 16 bitowym identyfikatorem segmentu pamięci. Selektor nie wskazuje bezpośrednio na identyfikowany przez niego segment pamięci. Bity selektora od 3 do 15 określają indeks segmentu w jednej z dwóch tablic deskryptorów segmentów: GDT lub LDT. Bit 2 selektora określa, która tablica deskryptorów segmentów ma zostać użyta: 0 oznacza GDT, 1 oznacza LDT. Bity 1 i 0 określają wymagany poziom uprzywilejowania potrzebny aby uzyskać dostęp do danych w danym segmencie.

Bibliografia

[Intel 1] *IA-32 Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture*, 1997, Intel Corporation

[Intel 3] *IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide (Preliminary)*, 1997, Intel Corporation

[ProjLinux] P. Bugalski et al. *Linux Podręcznik: Uruchamianie systemu i niskopoziomowe mechanizmy obsługi procesów*, 1999, wersja elektroniczna: http://rainbow.mimuw.edu.pl/SO/LinuxPodrecznik/