„Systemy czasu rzeczywistego”

„Wprowadzenie do przedmiotu”

Prezentacja jest współfinansowana przez
Unię Europejską w ramach
Europejskiego Funduszu Społecznego w projekcie pt.

„Innowacyjna dydaktyka bez ograniczeń - zintegrowany rozwój Politechniki Łódzkiej - zarządzanie Uczelnią, nowoczesna oferta edukacyjna i wzmacniania zdolności do zatrudniania osób niepełnosprawnych”

Prezentacja dystrybuowana jest bezpłatnie

Dariusz Makowski

Katedra Mikroelektroniki i Technik Informatycznych

tel. 631 2720

dmakow@dmcs.pl

http://neo.dmcs.p.lodz.pl/SCR

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Sprawy formalne

- Informacje ogólne
- Zaliczenie
- Laboratorium z SCR
- Materiały do wykładu i laboratorium

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

Zakres przedmiotu

- Wprowadzenie do przedmiotu
- Systemy operacyjne
- Systemy operacyjne czasu rzeczywistego
- RTEMS – system operacyny czasu rzeczywistego
- Linux jako system czasu rzeczywistego
- Inne systemy czasu rzeczywistego

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

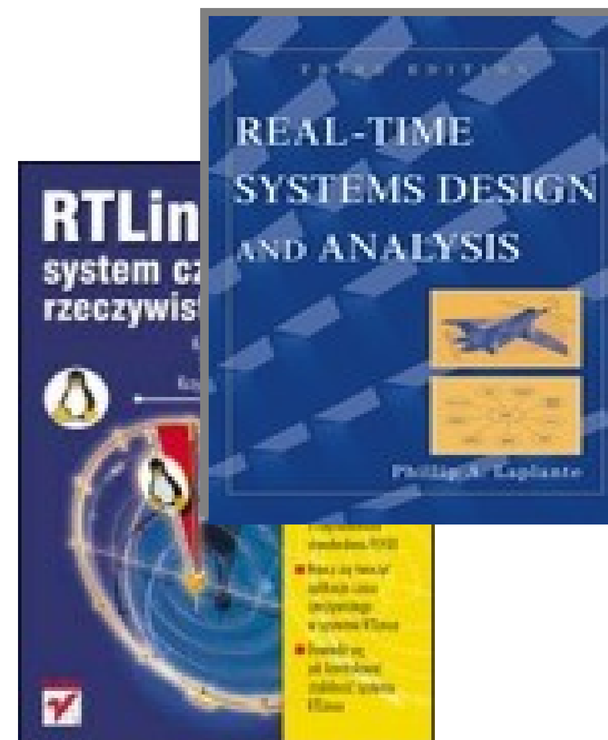
Literatura

Literatura obowiązkowa:

- Materiały wykładowe i laboratoryjne
- RTEMS - http://www.rtems.com/
- Phillip A. Laplante, "Real-Time Systems Design and Analysis", 3rd Edition, May 2004, Wiley-IEEE Press
- K. Lal, T. Rak, K. Orkisz, "RTLinux system czasu rzeczywistego" Helion 2003

Literatura uzupełniająca:

- S. R. Ball, "Embedded Microprocessor Systems" Butterworth-Heinemann 1996



ARM®

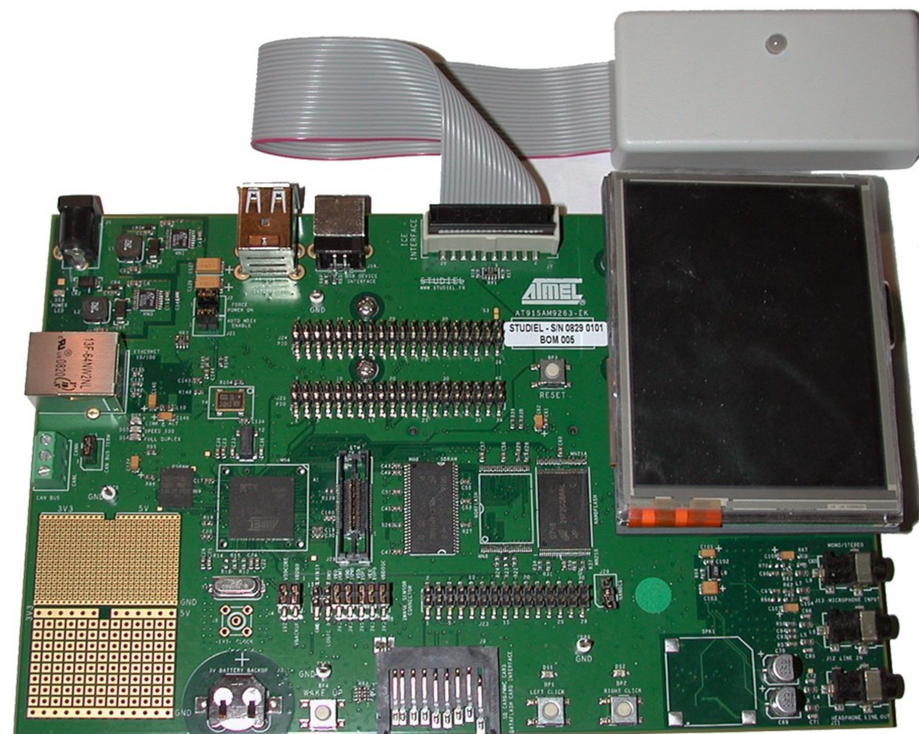
Laboratorium z Systemów czasu rzeczywistego

Laboratorium z Systemów czasu rzeczywistego

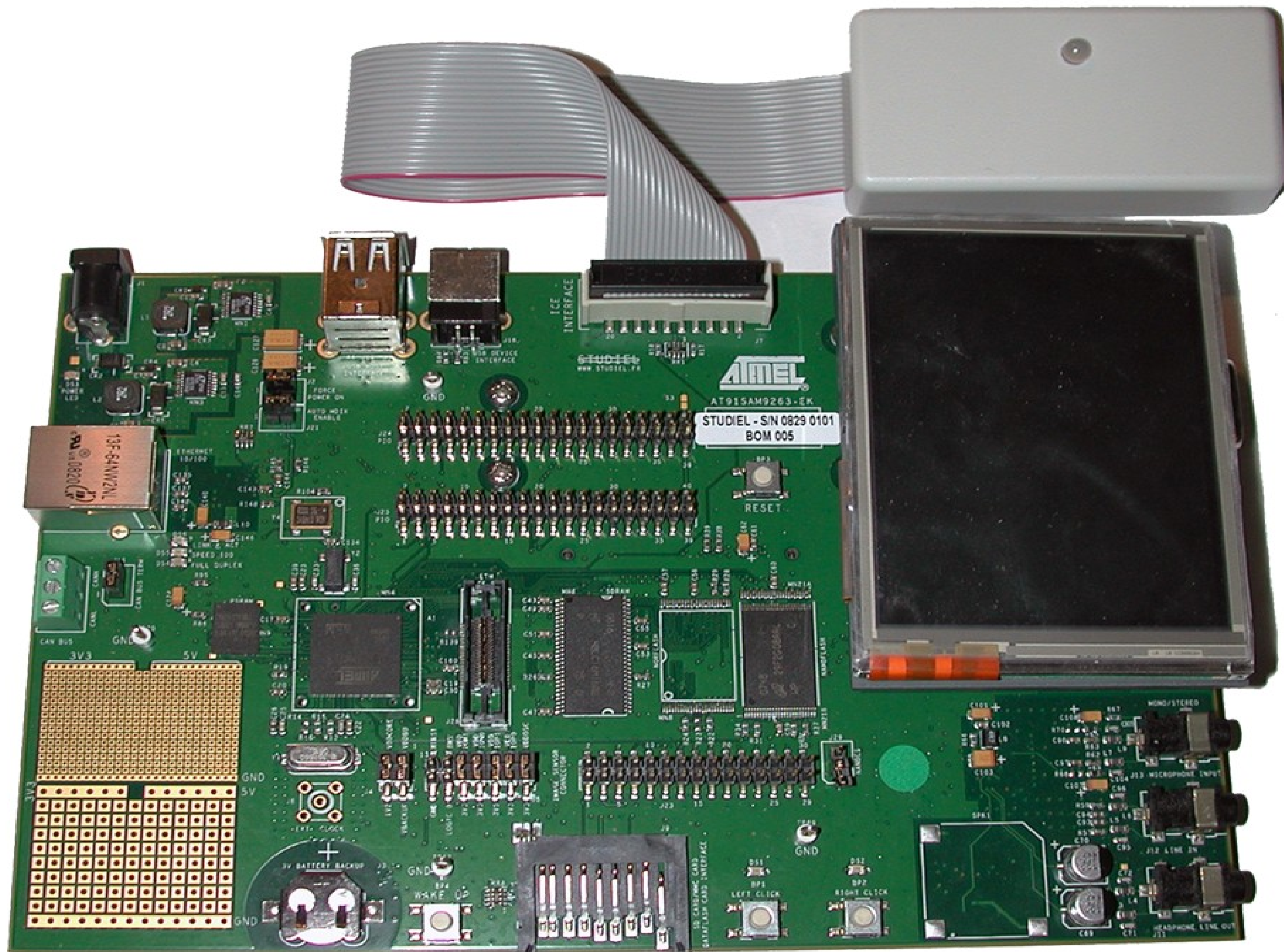
- **Platforma sprzętowa:**
  - Zestaw ewaluacyjny firmy MSC z procesorem ARM AT91SAM9263
- **Środowisko programowe:**
  - Komputer PC pracujący pod kontrolą systemu operacyjnego Linux,
  - Narzędzia GNU (kompilator skrośny, linker, itd...),
  - Debuger GDB,
- **System operacyjny czasu rzeczywistego:**
  - RTEMS.

Zestaw ewaluacyjny firmy MSC (1)

- **Procesor z rdzeniem ARM9TDMI firmy ATMEL:** AT91SAM9263
- **Dostępne pamięci:** 64MB SDRAM, 256MB NAND FLASH, 4 MB DataFlash, FlashCard slots
- **Dostępne interfejsy:** Ethernet 100-base TX, USB FS device, 2 x USB FS Host, CAN 2.0B, EIA RS232,
- **Wyświetlacz:** 3.5" 1/4 VGA TFT LCD z ekranem dotykowym
- **Kodek audio:** AC97 Audio DAC
- **Debug intefece:** JTAG
- **Interfejs do programowania:** JTAG, Free Atmel SAM-BA tools
- **Złącze:** SD/SDIO/MMC card slot
- **Oznaczenie producenta:** AT91SAM9263-STARTUP-PAKET



Zestaw ewaluacyjny firmy MSC (2)



Przykład programu

- Program pierwszy - „Hello World” z użyciem systemu czasu rzeczywistego RTEMS,
- Konfiguracja systemu operacyjnego RTEMS,
- Program drugi - utworzenie nowego wątku (zadania),
- Kompilacja programu,
- Uruchomienie i debugowanie programu.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

RTEMS – pierwszy program

```
#include <bsp.h>                                /* driver prototypes */
#include <stdio.h>
#include <stdlib.h>

rtems_task Init ( rtems_task_argument ignored )
{
        printf( "\n\n*** HELLO WORLD TEST ***\n" );
        printf( "Hello World\n" );
        printf( "*** END OF HELLO WORLD TEST ***\n" );
  /* while(1) {}; */                            /* currently exceptions handled*/
  exit( 0 );                                    /* to avoid exception */
}
```

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego
UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

RTEMS – konfiguracja systemu operacyjnego (1)

```
/* disable clock driver */
#define CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER
/* enable console driver */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
/* define maximum number of tasks */
#define CONFIGURE_MAXIMUM_TASKS          1
/* use RAM-based filesystem  */
#define CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM
/* update Init Table */
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
/* use user-defined Init */
#define CONFIGURE_INIT

#include <rtems/confdefs.h>                    /* configuration table template */
```

RTEMS – program drugi

```
#include <rtems.h>
rtems_task user_application
                    (rtems_task_argument argument)
{
    /* application specific initialization goes here */
  while ( 1 )  {            /* infinite loop */
   /*  APPLICATION CODE GOES HERE
    *  This code will typically include at least one
    *  directive which causes the calling task to
    *  give up the processor.
    */
  }
}
rtems_task init_task( rtems_task_argument ignored )
{
  rtems_id          tid;
  rtems_status_code status;
  rtems_name        name;
  name = rtems_build_name( 'A', 'P', 'P', '1' )
  status = rtems_task_create( name, 1,
                    RTEMS_MINIMUM_STACK_SIZE,
                    RTEMS_NO_PREEMPT,
                    RTEMS_FLOATING_POINT, &tid );
 if ( status != RTEMS_STATUS_SUCCESSFUL ) {
   printf( "rtems_task_create failed with status of
          %d.\n", status );
   exit( 1 );
 }

status = rtems_task_start( tid, user_application, 0 );

 if ( status != RTEMS_STATUS_SUCCESSFUL ) {
   printf( "rtems_task_start failed with status of
           %d.\n", status );
   exit( 1 );
 }

status = rtems_task_delete( SELF );    /* should not
return */

 printf( "rtems_task_delete returned with status of
          %d.\n", status );
 exit( 1 );
}
```

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

RTEMS – konfiguracja systemu operacyjnego (2)

```
/* enable clock driver */
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
/* enable console driver */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
/* define maximum number of tasks */
#define CONFIGURE_MAXIMUM_TASKS          2
/* give name to Init task */
#define CONFIGURE_INIT_TASK_NAME rtems_build_name( 'E', 'X', 'A', 'M' )
/* update Init Table */
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
/* use user-defined Init */
#define CONFIGURE_INIT

#include <rtems/confdefs.h>          /* configuration table template */
```

RTEMS – kompilacja

Zawartość pliku makefile

```
RTEMS_ROOT=/home/lab/scr/RTEMS/rtems
RTEMS_MAKEFILE_PATH=/home/lab/scr/RTEMS/rtems/bin-obj/arm-rtems4.10/c/csb337/make

include $(RTEMS_MAKEFILE_PATH)/Makefile.inc
include $(RTEMS_CUSTOM)
include $(PROJECT_ROOT)/make/leaf.cfg

CPU_CFLAGS += -Wall -g

all:    test
OUTPUT_FILE = hello

test: init.o
	$(CC_FOR_TARGET) -g  $(GCCSPECS) $(CPU_CFLAGS)  -o $(OUTPUT_FILE).elf init.o

init.o: init.c
	$(CC_FOR_TARGET)  $(GCCSPECS) $(CPU_CFLAGS) -c init.c

clean:
	rm -f *.o $(OUTPUT_FILE).elf init.o
```

RTEMS – kompilacja (2)

Ustawienie kompilatora i parametrów kompilacji

```
CC_FOR_TARGET = arm-rtems4.10-gcc --pipe

RTEMS_CPU=arm
RTEMS_CPU_MODEL=at91rm9200

# This is the actual bsp directory used during the build process.
RTEMS_BSP_FAMILY=csb337

#  This contains the compiler options necessary to select the CPU model
#  and (hopefully) optimize for it.
CPU_CFLAGS = -mcpu=arm920 -mstructure-size-boundary=8

# optimize flag: typically -O2
CFLAGS_OPTIMIZE_V = -g
```

Mikroprocesor a system operacyjny

Oprogramowanie mikrokomputerów

Komputer uniwersalny	Złożony komputer wbudowany	Prosty komputer wbudowany
Złożone aplikacje (GUI)	Proste aplikacje	
System Operacyjny	System Operacyjny	
Firmware	Firmware	Aplikacje / Firmware
Hardware	Hardware	Hardware

Komputery osobiste, uniwersalne:

- języki wysokiego poziomu (Asembler, C/C++, Pascal, Java, Basic...)

Komputery wbudowane, sterowniki:

- język niskiego poziomu Asembler,
- języki wysokiego poziomu (C/C++, Basic, Ada).

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

System operacyjny (1)

System Operacyjny (OS, Operating System)

Oprogramowanie, które zarządza zasobami sprzętowymi oraz udostępnia środowisko do wykonywania programów/aplikacji.

System operacyjny wykonuje następujące zadania:

- Udostępnia i rozdziela czas (moc obliczeniową) procesora dla wielu aplikacji,
- Udostępnia mechanizmy do synchronizacji i komunikacji pomiędzy zadaniami,
- Obsługa urządzeń peryferyjnych,
- Zarządzanie zadaniami (wątkami, procesami),
- Zarządzanie pamięcią,
- Udostępnianie systemu plików,
- Komunikacja i transmisja danych.

System operacyjny (2)

Możemy wyróżnić trzy główne elementy systemu operacyjnego:

- **Jądro systemu** wykonujące zadania,
- **Powłoka -** specjalny program umożliwiający komunikację użytkownika z systemem operacyjnym,
- **System plików** - sposób zapisu struktury danych na nośniku.

Jądro systemu składa się z następujących elementów:

- planisty czasu procesora, ustalającego które zadanie i jak długo będzie wykonywane,
- scheduler zadań, odpowiedzialnego za przełączanie pomiędzy uruchomionymi zadaniami,
  - modułu zapewniającego synchronizacje i komunikację pomiędzy zadaniami,
  - modułu obsługi przerwań i zarządzania urządzeniami,
  - modułu obsługi pamięci, zapewniającego przydział i ochronę pamięci.
  - inne funkcje.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Rola jądra w systemie operacyjnym

	Użytkownicy
System operacyjny	Obsługa interfejsu użytkownika
Moduł wykonawczy	Obsługa plików i pamięci masowej
Jądro	Komunikacja międzyprocesowa i synchronizacja
Mikrojądro	Planowanie zadań
Nanojądro	Zarządzanie procesami
	Zasoby sprzętowe

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Dlaczego potrzebny jest system operacyjny ?

- Rosnący poziom złożoności zadań wykonywanych przez współczesne systemy mikroprocesorowe,
- Implementacje skomplikowanych algorytmów przetwarzania danych,
- Programowanie współczesnych urządzeń peryferyjnych ze względu na złożoność sprzętu stało się zadaniem trudnym i czasochłonnym. Brak zunifikowanych interfejsów oraz różnorodność udostępnianych przez producentów urządzeń peryferyjnych sprawia, że kod źródłowy nie jest bezpośrednio przenośny nawet pomiędzy różnymi platformami tego samego producenta.
- Potrzeba użycia ustandaryzowanych sterowników.

Podział systemów operacyjnych

Podział ze względu na planowanie i przydział czasu procesora:

- system operacyjny czasu rzeczywistego (RTOS),
- systemy operacyjne czasowo niedeterministyczne.

Podział ze względu na sposób realizacji przełączania zadań:

- systemy z wywłaszczaniem zadań,
- systemy bez wywłaszczania.

- Podział ze względu na przeznaczenie systemu:
  - otwarte systemy operacyjne,
  - wbudowane systemy operacyjne.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Przykłady systemów operacyjnych

- AmigaOS,
- Apple: DOS, ProDOS, Mac OS, Mac OS X, Mac OS X Server,
- Atari TOS,
- Be: BeOS,
- IBM: OS/2, MFT, MVT, PC-DOS, MVS,
- Microsoft: MS-DOS, DR-DOS, Windows 1.0 – 3.x, 95/98/Me, NT/XP...,
- Novell: NetWare, Novell DOS,
- Unisys: MCP(Master Control Program), OS 2200,
- UNIX: Linux, HP-UX, Mac OS X, Sun Solaris, ...,
- Symbian,
- Palm OS,
- EcomStation,
- ...

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego
UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Przykłady systemów operacyjnych czasu rzeczywistego

- LynxOS
- OS9
- Phoenix-RTOS
- QNX
- Nut/OS
- RTEMS
- RT-Linux
- SenseOS
- VxWorks
- Suse Linux Enterprise Real Time
- MicroC/OS-II.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Pamięć wirtualna

- Pamięć wirtualna to mechanizm komputerowy zapewniający procesowi wrażenie pracy w jednym dużym, ciągłym obszarem pamięci operacyjnej,
- Stronicowanie pamięci – pamięć przydzielana poszczególnym procesom jest podzielona na strony. Przyznanie pamięci wiąże się z przypisaniem pewnej liczby stron dla danego procesu,
- W rzeczywistości pamięć może ulec fragmentacji (pamięć nieciągła) lub fragmenty pamięci mogą się znajdować w różnych pamięciach (różne pamięci SRAM/DRAM, urządzeniach pamięci masowej, np. karty pamięci FLASH, dyski twarde),
- Pamięć wirtualna realizowana jest na zasadzie tłumaczenia adresów. Procesy widzą pamięć ciągłą. Do translacji adresów zwykle wykorzystywana jest jednostka zarządzająca pamięcią MMU (Memory Menagement Unit),
- Pamięć wirtualną Intel zaimplementował po raz pierwszy w procesorze 80286 (tryb chroniony obsługi pamięci).

Układ zarządzania pamięcią (MMU), Motorola 68451

Układ zarządzania pamięcią VI475 "Apple HMMU"

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

Systemy operacyjne dla urządzeń wbudowanych

Systemy z procesorem wyposażonym w jednostkę zarządzania pamięcią (MMU):

Zmodyfikowane odmiany systemu Linux:

- MontaVista Linux,
- Android,
- LynuxWorks,
- RTEMS, itd...

Systemy z procesorem bez układu zarządzania pamięcią:

- µClinux,
- PetaLinux.

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

Systemy operacyjne dla urządzeń wbudowanych

Dlaczego brak układu zarządzania pamięcią jest problemem?

- Brak obsługi pamięci wirtualnej,
- Brak możliwości dynamicznego zmieniania rozmiaru pamięci przydzielonej danemu procesowi,
- Brak sprzętowej ochrony pamięci,
- Brak obsługi obszaru wymiany,
- Brak możliwości przydzielenia obszaru pamięci z różnych źródeł (SDRAM, RAM, HDD, FLASH, itd...),
- Fragmentacja pamięci przy dynamicznej alokacji.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY

Wielozadaniowość

- Współczesne systemy komputerowe pozwalają na załadowanie do pamięci komputera wielu programów i współbieżne ich wykonywanie; najważniejszym pojęciem, związanym ze wszystkimi systemami operacyjnymi, jest proces (zadanie), którym nazywać będziemy każdy program znajdujący się w stanie wykonywania,
- **Wielozadaniowość –** cecha systemu operacyjnego umożliwiająca pseudorównoczesne wykonywanie więcej niż jednego zadania.
- Wielozadaniowość realizowana jest na poziomie jądra systemu operacyjnego. Za przełączenia zadań odpowiedzialny jest planista oraz zarządca procesów.
- Praca równoległa wielu zadań jest pozorna (przy założeniu, że mamy do dyspozycji tylko jeden procesor). Współdzielenie zasobów umożliwia jednoczesne uruchomienie kilku zadań – użytkownik ma wrażenie, że wszystkie zadania są wykonywane równolegle. Jednak należy mieć świadomość, że zadania wykonywane są sekwencyjnie. Ma to wpływ na czas realizacji poszczególnych zadań.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Definicje podstawowe (1)

- **Proces, wątek (process, threat)** – elementarna jednostka pracy, zarządzana przez system operacyjny ubiegająca się o moc obliczeniową procesora. Procesy posiadają własne zasoby (stos, przydzieloną pamięć, otwarte deskryptory urządzeń I/O, pliki), natomiast wątki współdzielą prawie wszystkie zasoby w ramach procesu (np. wspólna pamięć). Proces utożsamiany jest z wykonywanym przez procesor programem, który definiuje jego zachowanie. Z punktu widzenia systemu operacyjnego każdy proces jest skojarzony z logiczną strukturą danych. W dalszej części wykładu używane będzie zamiennie określenie „zadanie" w odniesieniu do procesu. Za zarządzanie procesami odpowiada jądro systemu operacyjnego.
- Program komputerowy jest obiektem pasywnym i nie jest procesem. Proces jest obiektem aktywnym, który korzysta z zasobów dostępnych w systemie i dla którego można wskazać następny rozkaz do wykonania poprzez określenie zawartości wskaźnika instrukcji (licznika rozkazów) procesora.

Definicje podstawowe (1)

- Wykonanie procesu przebiega sekwencyjnie. Procesy mogą znajdować się w jednym z poniższych stanów:
  - Działający,
  - Czekający na zasoby systemu,
  - Przeznaczony do zniszczenia,
  - Proces „zombie”,
  - Proces w trakcie tworzenia.
- Do zarządzania procesami SO udostępnia specjalne funkcje (API).
- Program w formacie wykonywalnym (binarnym) staje się procesem, jeśli system operacyjny załaduje go do pamięci operacyjnej oraz uruchomi.
- Proces składa się z działającego programu oraz struktury danych opisującego jego stan i wykorzystywane zasoby,
- Aktywny proces może tworzyć nowe procesy lub wątki.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Definicje podstawowe (2)

- **Zasoby systemu –** element sprzętowy lub programowy systemu komputerowego. Brak zasobu może zablokować realizację programu. Przykłady zasobów: pamięć, procesor, bufor, itp...,
- **Zarządca procesów (process manager)** – kontroluje procesy w celu efektywnego i bezpiecznego wykorzystania współdzielonych zasobów.
- **Zarządca zasobów (resource manager)** – realizuje przydział zasobów stosownie do żądań procesów aktualnego stanu systemu oraz ogólnosystemowej polityki przydziału.
- **Kolejka zadań (job queue, FIFO)** – zbiór procesów systemu,
- **Kolejka procesów gotowych (ready queue)** – zbiór procesów gotowych do wykonania,
- **Kolejka urządzenia (device queue)** – zbiór procesów czekających na zakończenie operacji wej.-wyj.,
- **Kolejka procesów do synchronizacji** – zbiór procesów blokowanych przez inne zadania lub zasoby procesora. Procesy zwykle oczekują na sygnał synchronizacji, np. kolejka procesów blokowanych semaforem.

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Udział kolejek w planowanie przydziału procesora

Definicje podstawowe (3)

- **Planista (ang. scheduler)** – algorytm szeregowania konkurujących zadań odpowiedzialny za podział czasu procesora (mocy obliczeniowej). Algorytm szeregowania zadań stanowiących pewien zbiór to takie przydzielenie czasu procesora, że każde zadanie jest przetwarzane do momentu zakończenia. Jednakże przetwarzanie danego zadania może być przerwane na bliżej nieokreślony czas.
- **Planista krótkoterminowy (CPU scheduler)** – zajmuje się przydziałem czasu procesora do procesów gotowych (czekających w kolejce ready queue). Planista krótkoterminowy musi działać szybko.
- **Planista średnioterminowy (ang. medium-term scheduler)** – zajmuje się wymianą procesów pomiędzy pamięcią główną a pamięcią zewnętrzną (np. dyskiem).
- **Planista długoterminowy (long-term scheduler, job scheduler)** – zajmuje się ładowaniem nowych programów do pamięci i kontrolą liczby zadań w systemie oraz ich odpowiednim doborem w celu zrównoważenia wykorzystania zasobów. Planista długoterminowy nie musi działać zbyt szybko.

Algorytmy szeregowania zadań (1)

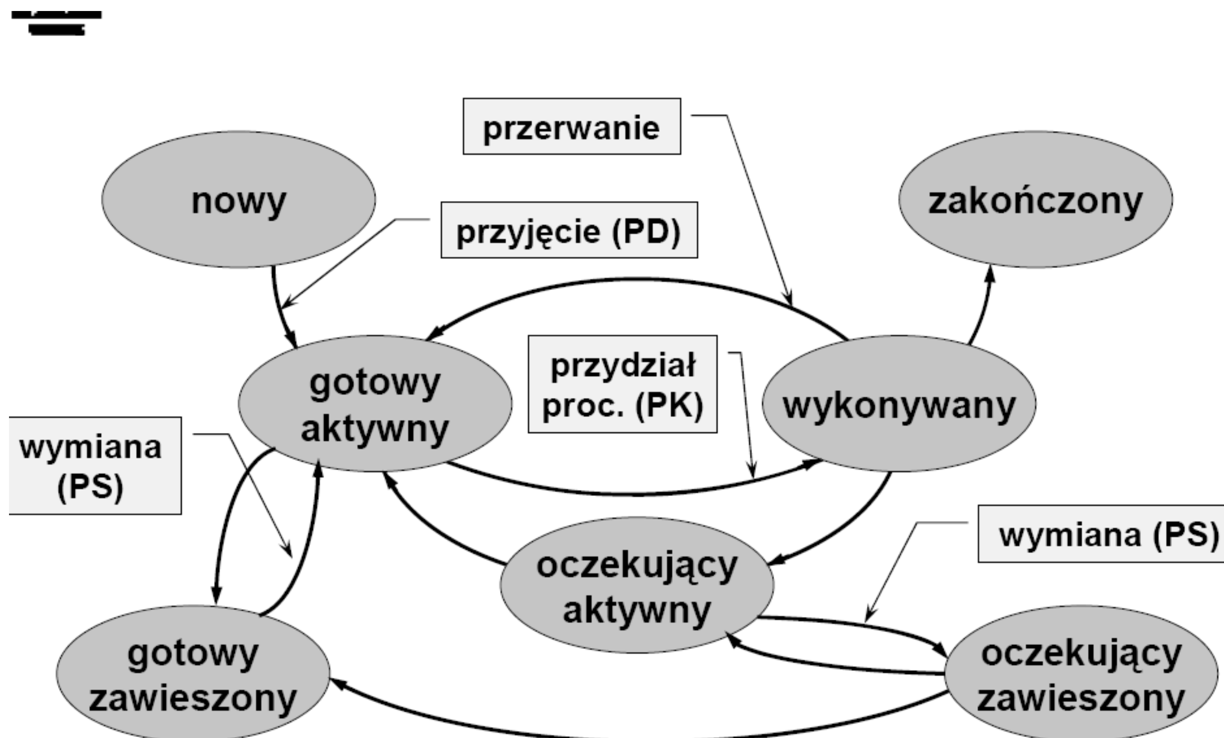
- **FIFO –** zadania wykonywane zgodnie z kolejnością umieszczenia w kolejce. Zadania wykonywane do momentu aż nie zostaną wywłaszczone przez zadanie o wyższym priorytecie lub przez to samo zadanie.
- **Planowanie rotacyjne (round-robin) –** każde z zadań otrzymuje kwant czasu. Po spożytkowaniu swojego kwantu zostaje wywłaszczone i ustawione na końcu kolejki.
- **SJF (shortest job first) –** Najpierw najkrótsze zadanie. Jest algorytmem optymalnym ze względu na najkrótszy średni czas oczekiwania. W wersji z wywłaszczaniem, stosowana jest metoda: najpierw najkrótszy czas pracy pozostałej do wykonania. Algorytm powoduje głodzenie długich procesów.

Algorytmy szeregowania zadań (2)

- **Planowanie priorytetowe –** wybór zadania zależy od przydzielonego priorytetu. Wadą algorytmu jest zagładzanie zadań z niskimi prorytetami. W celu poprawy algorytmu atosuje się postarzanie procesów, polegające na powolnym podnoszeniu priorytetu procesów zbyt długo oczekujących,
- **Planowanie wielopoziomowe –** zadania przypisywane są do kolejek szeregowania w zależności od parametru opisującego każde z zadań jakim w praktyce zwykle jest priorytet. Zadania w danej kolejce są następnie szeregowane określonym algorytmem takim jak na przykład FIFO lub round-robin i kierowane do wykonania. Jeśli w danej kolejce nie ma zadań gotowych do wykonywania, planista ponownie dokonuje analizy kolejki, ale dla zadań o niższym priorytecie. Zwykle możliwa jest zmiana kolejki w której szeregowane jest zadanie, poprzez zmianę priorytetu zadania,
- **Planowanie wieloprocesorowe –** rozdzielanie zadań dla jednakowych lub różnych procesorów/komputerów,
- **Synchronizacja międzyzadaniowa –** ze względu na powiązanie zadań różnymi zasobami a nie tylko procesorem, konieczne jest uwzględnienie aspektu dostępu do tych innych zasobów przez szeregowane zadania.

Cykl życia procesu

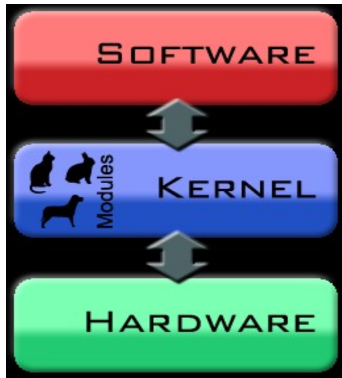
Cykl zmiany procesu



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY



Realizacja jądra systemu operacyjnego

Czy zawsze jest potrzebny system operacyjny ?

- Nie zawsze istnieje potrzeba implementacji złożonego jądra systemu operacyjnego,
- Znaczna część systemów wbudowanych pracuje bez systemu operacyjnego,
- Takie podejście daje możliwość realizacji prostego systemu mikroprocesorowego spełniającego wymagania czasu rzeczywistego,
- Analiza oraz debugowanie takiego programu jest znacznie łatwiejsza niż w przypadku SO,
- Łatwiejsza kontrola uwarunkowań czasowych dla poszczególnych zadań,
- Zalecane jest użycie programu bez systemu operacyjnego dla prostych systemów wbudowanych, gdy jest to możliwe (oszczędność zasobów procesora).

Pseudokernel

Wielozadaniowość może zostać zaimplementowana przy użyciu nieskończonej pętli (ang. Pseudokernel), w której wykonywane są zadania - cykliczne wykonywanie zadań. Takie podejście nie oferuje mechanizmów zarządzania zadaniami oferowanych przez system operacyjny. Stosowane jest często na mikrokontrolerach o niewielkim mocy obliczeniowej oraz niedużej pamięci programu i danych.

Systemy można podzielić na dwie grupy:

- Systemy oparte na nieskończonej pętli,
- Systemy sterowane przerwaniami.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY

Przerwania

Przerwanie (ang. interrupt) – sygnał powodujący zmianę przepływu sterowania. Pojawienie się przerwania powoduje wstrzymanie aktualnie wykonywanego programu i wykonanie przez procesor kodu procedury obsługi przerwania (ang. interrupt handler).

Przerwania dzielą się na dwie grupy:

- **Sprzętowe:**
  - Zewnętrzne – sygnał przerwania pochodzi z zewnętrznego urządzenia peryferyjnego,
  - Wewnętrzne – zgłaszane przez procesor w celu sygnalizacji sytuacji wyjątkowych, tzw. wyjątek (ang. exception),
    - faults (niepowodzenie) – sytuacje, w których aktualnie wykonywana instrukcja powoduje błąd,
    - traps (pułapki) – sytuacja, która nie jest błędem, jej wystąpienie ma na celu wykonanie określonego kodu,
    - aborts – błędy, których nie można naprawić,
  - Niemaskowalne – przerwania, których nie da się wyłączyć,
- **Programowe** – przerwanie wyzwalane jest instrukcja procesora z kodu programu. Dalsze zachowanie procesora jest identyczne jak dla przerwania sprzętowego.

Obsługa przerwań

1. Normalna praca programu – zadania przełączają się w pętli,
2. Zgłoszenie przerwania przez urządzenie peryferyjne,
3. Zachowanie stanu procesora,
   - Wywołanie procedury obsługującej przerwanie
   - Wykonanie kopii rejestrów i danych,
   - Wykonanie procedury obsługi przerwania,
4. Odtworzenie kopii rejestrów i danych,
5. Odtworzenie stanu procesora,
6. Powrót do realizacji zadań w głównej pętli programu.

- Obsługę przerwań należy wykonywać najszybciej jak to jest możliwe.
- Procedura obsługująca przerwania powinna być prosta (np. przepisanie danych do bufora, ustawienie flagi informującej o nadejściu danych, potwierdzenie przerwania).
- Przetwarzanie danych powinno być realizowane w głównej pętli programu.

Pętla z odpytywaniem

Pętla z odpytywaniem (ang. Polled Loop) – program używa nieskończonej pętli. Realizacja poszczególnych zadań zależna jest od zdarzeń (ang. Events), które ustawiają odpowiednie flagi. Jeżeli żadna z flag nie jest ustawiona procesor ciągle sprawdza (odpytuje), czy nie wystąpiło jakieś zewnętrzne zdarzenie.

Inicjalizacja procesora i urządzeń peryferyjnych

```
...
_branch_main:
        ldr     r0, =main
        mov     lr, pc
…
void main (void) {
                for (;;)                                /* wykonuj program w nieskończonej pętli */
                {
                /*      Realizacja zadania 1        */
                        if (data_available){            /* sprawdź, czy są nowe dane */
                                process_data();         /* wykonaj obliczenia */
                                data_available = 0;     /* wyzeruj flagę */
                        }
                /*      Realizacja zadania 2        */

                /*      Realizacja zadania 3        */
                }
}
```

Cykliczna realizacja zadań

Cykliczna realizacja zadań (ang. Cyclic Executives) – umożliwia realizację systemu wielozadaniowego obsługującego krótkie procesy na stosunkowo szybkim procesorze.

```
void main (void) {
        for (;;)                          /* wykonuj program w nieskończonej pętli */
        {
            Realizacja_zadania_1();
            Realizacja_zadania_2();
            Realizacja_zadania_3();
            Realizacja_zadania_3();       /* zadanie 3 wykonywane dwa razy cześciej niż
                                             zadanie 1 i 2 */
            ...
        }
}

Interrupt_Handlers {

}
```

Zadania wykonywane są zgodnie z algorytmem round-robin. Czas poświęcony na realizacje poszczególnych zadań może być zmieniony poprzez ponowne wykonanie tej samej funkcji. W celu imitacji SO można się posłużyć listą wskaźników do funkcji (dołożenie nowej funkcji wymaga umieszczenia nowego wskaźnika na liście).

Cykliczna realizacja zadań - przykład

Przykład pokazuje prosty program realizujący grę „Space Invaders”. Sterowanie odbywa się przy pomocy trzech klawiszy (lewo, prawo, ogień). Synchronizacja pomiędzy zadaniami może zostać zrealizowana przy wykorzystaniu zmiennych globalnych lub zdarzeń. W przykładzie nie wykorzystuję się przerwań. W celu uzyskania płynności gry zadania powinny być krótkie.

```
void main (void) {
        for (;;)                        /* wykonuj program w nieskończonej pętli */
        {
            check_for_keypressed();     /* sprawdź, czy klawisz wciśnięty */
            move_aliens();              /* przesuń statki obcych 1 linię niżej */
            check_for_keypressed();
            check_for_collision();      /* sprawdź, czy nie ma kolizji ze statkiem obcego */
            check_for_keypressed();
            update_screen();            /* odśwież obraz */
        }
}
```

Wady i zalety Polled Loop

Zalety:

- Prosta implementacja, łatwy proces debugowania,
- Zużycie mniejszych zasobów procesora w porównaniu do systemu operacyjnego,
- Przewidywalne zachowanie programu (nie biorąc pod uwagę asynchronicznych przerwań),
- Brak potrzeby przełączania kontekstu.

Wady:

- Główna pętla programu pracuje w trybie blokującym (znaczna część czasu procesora tracona jest na sprawdzanie warunków),
- Brak obsługi asynchronicznych zdarzeń,
- Narzucona kolejność wykonywania zadań,
- Trudności w implementacji nowych zadań,
- Potrzeba „ręcznej” organizacji kodu,
- Sprawdza się dobrze tylko z pojedynczym procesorem i niewielką liczbą obsługiwanych zdarzeń, które nie są od siebie zależne,
- Nie nadaje się do obsługi złożonych systemów.

Wielozadaniowość oparta na współpracy (Cooperative Multitasking)

Cooperative Multitasking (CM) - zadania nadal wykonywane są w głównej pętli programu. Przełączenie zadań następuje w momencie, gdy dane zadanie zakończy pracę (wywołanie funkcji break(), yield()). W przypadku, gdy zadanie nie może kontynuować pracy (brak dostępnych zasobów) oddaje swój czas dla innego zadania, np. oczekuje na nadejście danych (yield, abort, waitfor, or waitfordone).

Przykład 1:

Trzy zadania (process_a - process_c) wykonywane zgodnie z algorytmem R-R. Brak przełączenia kontekstu, zadania mają wspólny stos. Funkcje process_init n() inicjalizują parametry procesów (zmienne wykorzystywane w procesach). Przełączenie funkcji realizowane jest po wykonaniu funkcji break.

Przykład 2:

Cztery zadania (task0 - task3) wykonywane zgodnie z algorytmem R-R. Przełączenie kontekstu realizowane jest po wykonaniu funkcji yield. Każde zadanie ma własny stos. Po przejściu do pracy wielozadaniowej program nie wraca do funkcji main. Rozmiar stosu wynosi 31 (31 słów 24 bitowych) dla każdego zadania. W zadaniach używane są zmienne statyczne, aby zapewnić ich trwałość podczas przełączania zadań.

Cooperative Multitasking – przykład 1

```
void main (void) {
        process_inita();
        process_initb();
        process_initb();

 for (;;) /* wykonuj program w nieskończonej pętli */
  {
    process_a(); /* przełączenie zadań następuje */
    process_b(); /* po wywołaniu instrukcji      */
    process_c(); /* break                        */
  }
}
```

```
void process_a (void){
        for (;;){
                switch (state_a){
                        case 1: phase_a1();
                                break;
                        case 2: phase_a2();
                                break;
                }
        }
}

void process_b (void){
        for (;;){
                switch (state_b){
                        case 1: phase_b1();
                                break;
                        case 2: phase_b2();
                                break;
                }
        }
}
```

Przełączenie kontekstu

- **Przełączenie kontekstu (ang. Context Switching)** - proces zapisywania oraz odtwarzania minimalnej ilości informacji niezbędnych do przywrócenia zadania po jego przerwania w celu obsługi innego zadania lub przerwania.
- Kontekst jest zwykle zapisywany na stosie,
- Kontekst zwykłe zawiera kopię:
  - Rejestrów ogólnego przeznaczenia,
  - Licznika programu,
  - Rejestrów koprocesora,
  - Rejestrów określający stronę pamięci.
- Podczas przełączania kontekstu, w sekcji krytycznej kodu, przerwania są wyłączane.

Przykład realizacji algorytmu CM z przełączeniem kontekstu

```
void main (void) {
/* inicjalizacja systemu, załadowanie uchwytów przerwań */
        init_task1();
        while(1);               /* zadanie IDLE */
}

void Task_1 (void){
        save(context);          /* zapisanie kontekstu poprzedniego zadania */
        task1();                /* przełączenie i uruchomienie zadania 1 */
        restore(context;)       /* odtworzenie kontekstu poprzedniego zadania, przełączenie na
                                   kolejne zadanie */
}

void Task_2 (void){
        save(context);          /* zapisanie kontekstu poprzedniego zadania */
        task2();                /* przełączenie i uruchomienie zadania 2 */
        restore(context;)       /* odtworzenie kontekstu poprzedniego zadania, przełączenie na
                                   kolejne zadanie */
}
```

Cooperative Multitasking – przykład 2

```
/* kopia stosów dla poszczególnych zadań, 31 B */
union{
    uint24_t u24b;                  /* zmienna 24 bit */
    uint8_t u8b[sizeof(uint24_t)];      /* tablica 3 bajtów */
} SavedStack[MaxTask+1][31];
/* kopia wskaźników stosów dla zadań 8/
static uint8_t SavedStackpointer[MaxTask+1];

void main(void){
    InitTask0();
    InitTask1();
    InitTask2();
    InitTask3()                     /* inicjalizacja zadań    */
    StartMultiTask();               /* uruchomienie zadań */
}

void InitTask(static uint8_t TaskNum){
// wyłącz przerwania
/* kopia wskaźnika stosu i adresu powrotu */
    SavedStackpointer[TaskNum]=STKPTR;
    SavedStack[TaskNum][STKPTR&0b11111].u24b=TOS;
    _asm POP _endasm; /* zdejmij adres ze stosu,
                            powrót do main zamiast do task0 */
// włącz przerwania
}
```

```
void task0(void){
    static uint8_t n=0;
    InitTask(0);   /* konfiguracja adresu powrotu*/
    while(1){      /* główna pętla programu       */
        n++;
        yield();     /* przełącz zadanie */
}}

void yield(void){
// wyłącz przerwania
  zapisz wskaźnik stosu dla TaskNum
  zapisz obecny stos dla TaskNum
  zdejmij ostatnią wartość ze stosu – powrót do
zadania TaskNum, a nie do funkcji yield
  przełącz TaskNum na następne zadanie
  przywróć wskaźnik stosu dla TaskNum
  przywróć stos dla TaskNum
// włącz przerwania
}

void StartMultiTask(void){
// wyłącz przerwania
  STKPTR=0;        /* wyzeruj wsk. stosu */
  przywróć wskaźnik stosu dla zadania 0
  przywróć stos dla zadania 0
// włącz przerwania
}
```

Cooperative Multitasking – przykład 2

```
/* kopia stosów dla poszczególnych zadań, 31 B */
union{
    uint24_t u24b;                    /* zmienna 24 bit   */
    uint8_t u8b[sizeof(uint24_t)];    /* tablica 3 bajtów */
} SavedStack[MaxTask+1][31];
/* kopia wskaźników stosów dla zadań 8/
static uint8_t SavedStackpointer[MaxTask+1];

void main(void){
    InitTask0();
    InitTask1();
    InitTask2();
    InitTask3()                  /* inicjalizacja zadań   */
    StartMultiTask();            /* uruchomienie zadań */
}

void InitTask(static uint8_t TaskNum){
// wyłącz przerwania
/* kopia wskaźnika stosu i adresu powrotu */

    _asm POP _endasm; /* zdejmij adres ze stosu,
                         powrót do main zamiast do task0 */
// włącz przerwania
}
```

```
void task0(void){
    static uint8_t n=0;
    InitTask(0);   /* konfiguracja adresu powrotu*/
    while(1){      /* główna pętla programu       */
        n++;
        yield();     /* przełącz zadanie */
}}

void yield(void){
// wyłącz przerwania
  zapisz wskaźnik stosu dla TaskNum
  przełącz TaskNum na następne zadanie
  przywróć wskaźnik stosu dla TaskNum
// włącz przerwania
}

void StartMultiTask(void){
// wyłącz przerwania
  STKPTR=0;        /* wyzeruj wsk. stosu */
  przywróć wskaźnik stosu dla zadania 0
  - przywróć stos dla zadania 0
// włącz przerwania
}
```

Wielozadaniowość oparta na współpracy (Cooperative Multitasking)

Zalety:

- Tania alternatywa dla urządzeń wbudowanych bez systemu operacyjnego,
- Procesor nie marnuje czasu czekając na zdarzenie – może zająć się wykonaniem kolejnego zadania,
- Możliwość implementacji dowolnej liczby procesów (ograniczonych mocą obliczeniową procesora),
- Procesy odpowiedzialne za przełączanie zadań,
- Zmiana kontekstu następuje przez wywołanie funkcji yield().

Wady:

- Brak możliwości wywłaszczania zadań,
- Programista musi zadbać o przełączenie zadania,
- Trudności w obliczeniu czasu reakcji systemu na zdarzenie asynchroniczne,
- Niewłaściwie napisana funkcja może zablokować cały system (brak możliwości przełączenia zadania).

Zadania przełączane przerwaniami

- W przypadku systemów sterowanych przerwaniami zadania są szeregowane na podstawie sygnałów lub zdarzeń generowanych przez przerwania.
- Przełączanie zadań (lub kontekstu) realizowane jest na przerwaniu od timera. Czas przydzielany na realizację poszczególnych zadań określany jest kwantem czasu (ang. quantum or slice time). Kwant czasu powinien być nieco dłuższy od czasu wymaganego na typową interakcję systemu, tak aby możliwe było zakończenie większości zadań bez kosztownego przełączenia kontekstu (zwykłe 10-100 ms).
- Jeżeli procesor wyposażony jest w sterownik przerwań, decyduje on o kolejności realizowanych przerwań (zadań). Jeżeli procesor dysponuje tylko jednym poziomem przerwań funkcja obsługująca przerwania decyduje o kolejności obsługiwanych przerwań (zadań).
- Synchronizacja dostępu do zasobów współdzielonych w ramach procedury obsługi przerwania jest realizowana poprzez blokowanie przerwań. W tym czasie procesor ma ograniczoną możliwość reakcji na zdarzenia. Z tego powodu przebywanie w sekcji krytycznej programu, gdzie blokowane są przerwania, powinno być realizowane najszybciej jak to możliwe.

Przykład realizacji algorytmu CM z przełączeniem kontekstu + IRQ

```
void main (void) {
/* inicjalizacja systemu, załadowanie uchwytów przerwań */
        init();
        while(1);               /* zadanie IDLE */
}

/* Timer generuje przerwania uaktywniające poszczególne zadania */

void int1 (void){
        save(context);          /* zapisanie kontekstu poprzedniego zadania */
        task1();                /* przełączenie i uruchomienie zadania 1 */
        restore(context;)       /* odtworzenie kontekstu poprzedniego zadania */
}
void int2 (void){
        save(context);          /* zapisanie kontekstu poprzedniego zadania */
        task2();                /* przełączenie i uruchomienie zadania 2 */
        restore(context;)       /* odtworzenie kontekstu poprzedniego zadania */
}
void int3 (void){
        save(context);          /* zapisanie kontekstu poprzedniego zadania */
        task3();                /* przełączenie i uruchomienie zadania 3 */
        restore(context;)       /* odtworzenie kontekstu poprzedniego zadania */
}
```

Przykład realizacji algorytmu CM z przełączeniem kontekstu + IRQ

```
void main (void) {
/* inicjalizacja systemu, załadowanie uchwytów przer. */
  init_tasks();            /* inicjalizacja zadań */
  StartMultiTask();        /* uruchomienie zadań */
  while(1);                /* zadanie IDLE */
}
void Task0 (void){
  for(;;) {

        task1();                /* przełączenie i
                                   uruchomienie zadania 1 */
  }
}
void Task2 (void){
  for(;;) {

        task2();                /* przełączenie i
                                   uruchomienie zadania 2 */
  }
}
```

```
// Funkcja działa na przerwaniu od Timera
void yield(void){
// wyłącz przerwania
  save(context);     /* zapisanie kontekstu
                        poprzedniego zadania */
  IncrementTask(); /* zmień nr  zadania */
  restore(context;)     /* odtworzenie kontekstu
                           poprzedniego zadania */
// włącz przerwania
}
void StartMultiTask(void){
// wyłącz przerwania
  ChangeToTask0(); /* uaktywnij zadanie 0 */
  restore(context;)     /* odtworzenie kontekstu
                           zadania 0 */
// włącz przerwania
}
```

Cykliczna realizacja zadań z przerwaniami

Zalety:

- Zadania przypisane do odpowiednich zdarzeń (przerwań),
- Możliwość przydzielenia czasu procesora dla danego zadania (kwantu czasu),
- Możliwość łatwego dodania nowego zadania.

Wady:

- Priorytety zadań uzależnione od sterownika przerwań lub narzucone przez funkcję obsługująca przerwania.

System wielozadaniowy z wywłaszczaniem

Wywłaszczenie (preemption) – przerwanie aktualnie wykonywanego zadania (bez ustalonej z nim zależności) z zamiarem wznowienia jego działania w późniejszym czasie. Zadanie o niższym priorytecie może zostać wywłaszczone przez zadanie o priorytecie wyższym (preemptive-priority system).

- Wyróżnia się systemy o stałych lub zmiennych priorytetach (np. priorytet zadania o niskim priorytecie oczekującego w kolejce może zostać podniesiony, co zapobiega zagłodzeniu procesu przez zadania o wysokich priorytetach).
- Wywłaszczenie zadania określane jest mianem przełączenia kontekstu i jest wykonywane przez algorytm szeregujący (preemptive scheduler).

System wielozadaniowy z wywłaszczaniem

Zalety:

- Blokada jednego zadania nie powoduje zawieszenia całego systemu,
- Umożliwia dokładne określenie czasu kiedy dane zadanie może korzystać z procesora,
- Wywłaszczanie może również dotyczyć całego jądra (Linux),
- Często stosowane w systemach wbudowanych,

Wady:

- Skomplikowana i zasobożerna implementacja,
- Funkcje obsługujące przerwania nie są wywłaszczalne (mogą zablokować cały system),
- Możliwość powstania wyścigów podczas zgłoszenia kilku przerwań o różnych priorytetach,
- Trudny do określenia czas reakcji na sygnały zewnętrze (asynchroniczne).

Proces bezczynności

Proces o najniższym priorytecie (ang. idle process) – proces bezczynności. Procesor znajduje się w stanie bezczynności jeżeli w kolejce procesów gotowych nie ma żadnego zadania. W takim przypadku w kolejce może zostać umieszczony proces idle.

W przypadku systemów sterowanych przerwaniami proces idle może znajdować się głównej pętli programu. Proces bezczynności ma zawsze najniższy priorytet.

```
void main (void) {
/* inicjalizacja systemu, załadowanie uchwytów przerwań */
        init();
        while(1){
                Idle_task();                    /* zadanie IDLE */
        }
}
```

W przypadku systemów mobilnych procesor powinien zostać przełączony do jednego z dostępnych trybów obniżonego poboru mocy.

W przypadku systemu Linux proces idle ma ID równe 0. Nie ma możliwości usunięcia (zabicia) procesu idle.

Sekcja krytyczna

Sekcja krytyczna - w programowaniu współbieżnym fragment kodu programu, w którym korzysta się z zasobu dzielonego.

- Zasoby systemu mikroprocesorowego mogą być wykorzystywane tylko przez jeden proces lub wątek w danej chwili.
- System operacyjny udostępnia mechanizmy pozwalające synchronizować dostęp do zasobów. Dostęp do danego zasobu udostępniany jest tylko dla jednego procesu, pozostałe są wstrzymywane.
- Dostęp do sekcji krytycznej programu powinien być wykonywany szybko, a kod programu operujący na niej powinien być krótki.
- Do blokowania dostępu do sekcji krytycznej wykorzystuje się:
  - Semafory,
  - Muteksy.

Sekcja krytyczna - przykład

```
int dataReceived = 0;
void main (void){
  while (1){
        If (dataReceived){
              ProcessData();
        przerwanie Irq_handler->
              dataReceived = 0;
        }
  }
}
```

```
void Int_handler (void){
        Buffer = data;
        dataReceived = 1;
}
```

Przerwanie spowoduje wpisanie nowych danych do bufora i ustawienie flagi dataReceived. Jednak dane mogą zostać nieprzetworzone ze względu na zjawisko wyścigu. Handler ustawia flagę, która natychmiast jest kasowana w funkcji main.

Inny przykład:

Zadanie 1:

```
int temp1 = counter;
temp1 = temp1 +1;
counter = temp1;
```

Zadanie 2:

```
int temp2 = counter;
temp2 = temp2 +1;
counter = temp2;
```

Semafor

Semafor to mechanizm synchronizacji procesów udostępniany przez jądro SO. Semafory dzielimy na semafory binarne, które mogą przyjmować wartości 0 lub 1 oraz semafory ogólne (wartości dodatnie).

Semafory mogą wpływać na zadania:

- Wstrzymywanie zadania do czasu zwolnienia semafora powiązanego z danym zasobem,
- Wznowienie pracy zadania oczekującego na semaforze,
- Utrzymywanie semafora nawet po zakończeniu zadania, które go utworzyło.

  Do obsługi semaforów wykorzystuje się trzy funkcje:
  - Wait() - funkcja blokująca, czekaj aż zwolni się semafor, zmniejsz wartość semafora – zajęcie semafora,
  - Signal() - zwiększ wartość semafora, zwolnienie semafora,
  - Init() - inicjalizacja semafora wartością (np. 1 – sem. wolny),
- Niewłaściwe użycie semafora może doprowadzić do zakleszczenia dwóch procesów (pierwszy czeka na wyniki drugiego, a drugi na wyniki pierwszego).

Semafor – przykład

```
semafor sem = 1;
void Task1(void){
  for (;;){
    kod_programu
    wait(sem);          /* zajęcie semafora *
      Krytyczna _część_kodu;
    signal (sem);       /* zwolnienie semafora */
    kod_programu
  }
}
```

```
void Task2(void){
  for (;;){
    kod_programu
    wait(sem);          /* zajęcie semafora *
      Krytyczna _część_kodu;
    signal (sem);       /* zwolnienie semafora */
    kod_programu
  }
}
```

Mutex

Mutex (ang. Mutual Exclusion, wzajemne wykluczanie) - element stosowany w programowaniu służący do kontroli dostępu do zasobów niemożliwych do współdzielenia, do ochrony sekcji krytycznych.

Mutex ma dwa możliwe stany: otwarty (nie zajęty przez żaden proces) lub zajęty (przez jeden proces). Mutex nie może być w posiadaniu więcej niż jednego procesu na raz. Proces próbujący zając mutex będący w osiadaniu innego procesu jest zawieszany aż do chwili zwolnienia mutexu przez proces, który go posiadał wcześniej.

Bariery

Bariera (ang. Barrier) - element stosowany w programowaniu służący, podobnie jak semafory i mutexy, do synchronizacji zadań.

- Bariery dostępne są jako funkcje niektórych systemów operacyjnych,
- Bariery umożliwiają synchronizację zadań, które mogą zostać odblokowane globalnie (podobnie jak odbywa się to na wyścigach konnych),
- Dalsza realizacja zadań narzucona jest przez planistę i egzekutora,
- Aktywacja bariery spowoduje ponowne zablokowanie wszystkich zadań.

Sygnały

Sygnał (ang. signal) – zdarzenie wysyłane przez system operacyjny do procesu lub wątku.

- Sygnał, podobnie do przerwania (ISR), posiada uchwyt ASR (Asynchronous Signal Routine) umożliwiający przejęcie sygnału i jego dalszą obsługę.
- Po wygenerowaniu aktywnego sygnału następuje wstrzymanie pracy procesora i wykonanie skoku do ASR,

Porównanie sygnału i przerwania:

- IRQ obsługiwane jest przez procesor, ASR przez system operacyjny,
- ISR nie wpływa na wynik działania danego procesu (może spowodować przełączenie procesu), ASR ma bezpośredni wpływ na wynik zadania lub procesu,
- W przypadku IRQ przesyłany jest numer wektora przerwania do procesora, w przypadku ASR przesyłana jest lista aktywnych sygnałów jak parametr ASR.

Przykład sygnału obsługiwanego pod systemem Linux:

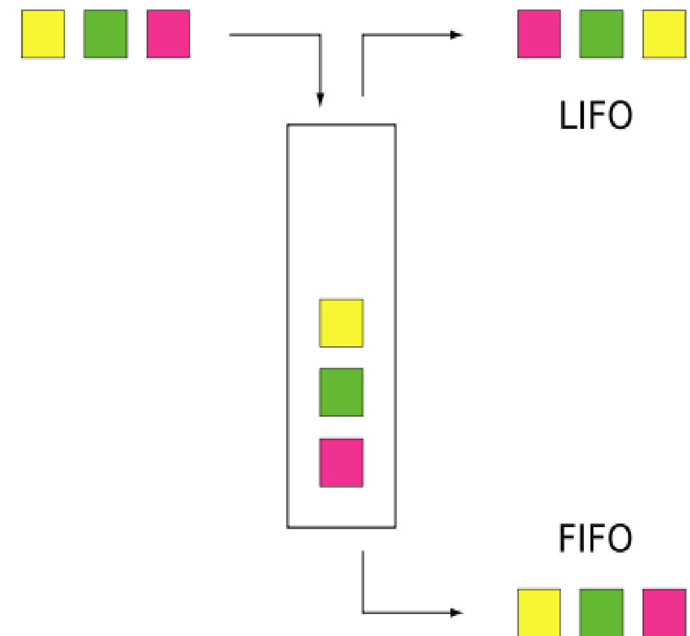
Ctrl + C - wciśnięcie tych klawiszy powoduje wygenerowanie przez system operacyjny sygnału INT (SIGINT) do bieżącego procesu oraz jego zakończenie.

Podobnie można wygenerować sygnał INT z poziomu aplikacji oraz zmienić domyślne zachowanie aplikacji na ten sygnał.

Kolejka wiadomości

Kolejka wiadomości (ang. Message queue, Mailbox) – zmiennej długości bufor wykorzystywany do asynchronicznego przesyłania wiadomości pomiędzy zadaniami oraz funkcjami obsługującymi przerwania.

- Kolejki umożliwiają synchronizację zadań – blokujące oczekiwanie na dane w kolejce,
- SO zwykle udostępnia funkcje umożliwiające sprawdzenie, czy w kolejce znajdują się dane,
- Kolejki realizowane są jako bufory
  - **FIFO (First-in First-out) -** liniowa struktura danych, w której nowe dane dopisywane są na końcu kolejki, a z początku kolejki pobierane są dane do dalszego przetwarzania,
  - **LIFO (Last-in First-Out) -** liniowa struktura danych, w której przetwarzane są dane wprowadzone jako ostatnie.



Zdarzenia

Zdarzenia (ang. Events) – komunikaty przesyłane pomiędzy zadaniami w celu poinformowania o wystąpieniu ważnej sytuacji

- Zdarzenia mogą być generowane przez sprzęt (zdarzenia sprzętowe), system operacyjny (systemowe) lub oprogramowanie (programowe).
- Zdarzenia w postaci specjalnych są przechowywane przez system operacyjny w kolejkach zdarzeń (ang. event queues). Kolejki są zwykle skojarzone z procesami.
- Zdarzenia mogą być przesyłane również jako ustawienie pojedynczego bitu, tzw. flagi. Flagi dostępne dla wszystkich procesów (zmienna globalna).
- Zdarzenia w systemie Linux opisane są przy pomocy struktury:

```
typedef struct {
        int type;                   /* typ zdarzenia*/
        unsigned long serial;       /* liczba ostatnich zleceń przetworzonych przez serwer */
        Bool send_event;            /* true jeśli pochodzi z wywołania SendEvent */
        Display *display;           /* Display z którego zdarzenie zostało odczytane */
        Window window;
    } XAnyEvent;
```