Wykład 5
Przerwania i wywołania systemowe

Porty wejścia-wyjścia

- Intel x86 posiada 65536 portów wejścia-wyjścia, do których dostęp możliwy jest za pomocą specjalnych rozkazów rodziny in* oraz out* (Przestrzeń adresowa we-wy rozłączna z przestrzenią adresową pamięci).
- Sterownik powinien skorzystać z funkcji

```
void request_region(unsigned int from, unsigned int num, const char *name)
void release_region(unsigned int from, unsigned int num)
```

  Aby odpowiednio zarezerwować i zwolnić zakres adresów wejścia wyjścia. Dostępna jest również funkcja `check_region` sprawdzająca, czy zakres adresów wejścia wyjścia jest wolny.

- Odczyt danych z portów przy pomocy makra `inpb(port)` (`inpw` oraz `inpl` dla danych 16- i 32-bitowych). Zapis za pomocą makra `outb(wartość,port)` (`outw` oraz `outl` dla danych 16- i 32-bitowych).
- Makra mają wersje kończące sie na `_p` (np `inpb_p`), które czekają kilka mikrosekund po wykonaniu odczytu bądź też zapisu (niektóre urządzenia mogą mieć problemy, jeżeli dane do portów zapisywane są zbyt szybko).

kanały DMA (na szynie ISA)

- Kanały ISA DMA rezerwujemy/zwalniamy przy pomocy funkcji

```
int request_dma(unsigned int dmanr, const char * device_id)
void free_dma(unsigned int dmanr)
```

Rezerwacja jest głównie przeprowadzana dla celów „grzecznościowych”. Umożliwia wykrycie sytuacji, w której kilka urządzeń

- Do dyspozycji mamy następujące API:

```
void enable_dma(unsigned int dmanr)  // Włącz kanał, transmisja rozpoczęta
void disable_dma(unsigned int dmanr) // Wyłącz kanał
// Ustaw tryb transmisji np. zapis albo odczyt
void set_dma_mode(unsigned int dmanr, char mode)
void set_dma_addr(unsigned int dmanr, unsigned int addr) // Adres początkowy
void set_dma_count(unsigned int dmanr, unsigned int count) // Liczbę bajtów
// Ile bajtów pozostało do przesłania - potencjalnie niebezpieczne
int get_dma_residue(unsigned int dmanr)
```

wraz z wszelkimi ograniczeniami szyny ISA i kontrolera 8237 (adres 24-bitowy, długość bloku <=128KB, blok nie może przekraczać granicy 128KB). Najlepiej alokować bufor przy pomocy kmalloc(GFP_DMA,)

Sygnały (przerwania na poziomie użytkownika)

- Sygnały pozwalają procesowi na reakcję na asynchroniczne zdarznia. Zdarzenia mogą powstać w wyniku akcji procesu, innych procesów lub jądra.
- Każdy sygnał jest reprezentowany przez stałą w formie SIGXXX. Najczęstsze przyczyny zgłaszania sygnałów to:
  - Akcje (często niepoprawne) procesu (SIGFPE, SIGSEGV, SIGILL, ...)
  - Wciśnięcie klawiszy specjalnych (SIGINT – ctrl-c, SIGSTP ctrl-z)
  - Jawne wysłanie sygnału przez inny proces (SIGKILL, dowolny inny sygnał).
- Reakcją na sygnał może być: (a) Wywołanie handlera zdefiniowanego przez użytkownika, (b) Zabicie procesu (zawsze w przypadku SIGKILL) (c) Zabicie procesu i zapisanie pliku core, (d) Wstrzymanie procesu (zawsze w przypadku SIGSTOP) (e) wznowienie wsztrzymanego procesu (SIGCONT), (f) zignorowanie sygnału
- Z wyjątkiem SIGKILL, SIGSTOP i SIGCONT proces może kontrolować reakację na sygnały.
- Sygnał z handlerem ma charakter asynchronicznego przerwania (nie wiemy, kiedy może nadejść) ze wszelkimi tego konsekwencjami dla synchronizacji.
  - Np. w czasie wykonania funkcji malloc otrzymalismy sygnal SIGALARM (timer). Jeżeli teraz jego handler wykona również malloc, to .....
  - Podobnie printf, etc ...

API sygnałów (standard POSIX)

```
struct sigaction {
    void (*sa_handler)();
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void); // nie używane
}
```

- Structura sigaction opisuje reakcję na sygnały:
  - sa_handler może być równy: SIG_DFL (akcja domyślna) SIG_IGN (ignoruj sygnał) albo może wskazywać na adres handlera.
  - sa_flags jest bitową (|) sumą następujących flag SA_NOCLDSTOP (nie generuj sygnał SIHCHLD, gdy proces potomny zostanie przerwany), SA_ONESHOT (po obsłudze jednego sygnału przełącz się na obsługę domyślną, **jest to domyślne zachowanie**) SA_RESTART (nie odtwarzaj akcji domyślnej po otrzymaniu sygnału), SA_NOMASK (handler sygnału może być przerwany przez siebie samego – **nie zalecam**).
  - sa_mask maska (zbiór) sygnałów zablokowanych podczas obsługi tego sygnału przez handler. (Istnieje odrębne API do operacji na zbiorach sygnałów: sigemptyset, sigadset, sigfillset, etc..)

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Zmiana lub/i odczyt (aktualnej reakcji na sygnał signum, w zależności od który ze wskaźników act oraz oldact jest różny od NULL

API sygnałów - c.d.

- Uśpij proces, do momentu pojawienia się sygnału.

```
int pause(void);
```

- Ustaw maskę sygnałów zablokowanych (how – SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK)

```
int  sigprocmask(int  how,  const  sigset_t *set, sigset_t *oldset);
```

- Sprawdź, czy nie są zgłoszone sygnały, których obsługa jest zablokowana.

```
int sigpending(sigset_t *set)
```

- Ustaw sygnał SIGALARM (patrz też setitimer)

```
long alarm(long seconds);
```

- Wyślij sygnał sig procesowi pid

```
int kill(pid_t pid, int sig)
```

Przerwania sprzętowe

- Dzielą się na przerwania szybkie oraz wolne.
- Handler obsługi przerwania wolnego wykonuje się z odblokowanymi przerwaniami.
  - Może być więc przerwany przez inne przerwania.
  - Po obsłudze handlera mogą być wywołane inne funkcje (planista, dolne połowy, kolejki zadań).
  - Bardzo ważnym przykładem jest przerwanie zegarowe.
- Handler przerwania wolnego wywołuje się z zablokowanymi przerwaniami.
  - Dodatkowe czynności nie są wykonywane.
- Rezerwacja przerwania następuje przy pomocy funkcji: `int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char * devname, void *dev_id)`
  - irq – numer przerwania.
  - Handler - adres procedury obsługi przerwania
  - irq flags – szybkie z flagą SA_INTERRUPT, wolne bez tej flagi.

Handler obsługi przerwania

- `void handler(int irq, void * dev_id, struct pt_regs *)`
- Znaczenie parametrów:
  - irq numer przerwania (ten sam handler może obsługiwać kilka przerwań)
  - dev_id jest równy parametrowi dev_id przekazanemu funkcji request_irq.
    - Może zostać wykorzystany do przekazania handlerowi dodatkowych danych.
  - pt_regs przechowuje adres struktury w której zawarte są wartości rejestrów procesora dla przerwanego procesu.
    - Handler obsługi przerwania może sprawdzić, czy przerwany proces wykonywał się w trybie jądra, czy też w trybie użytkownika.
- Zwolnienie przerwania przy pomocy funkcji free_irq.
- Możliwość współdzielenia przerwań na magistrali PCI (flaga SA_SHIRQ)

Obsługa przerwania wolnego - pseudokod

```
irq(irq_num,irq_controller)
{
  SAVE_ALL     // zapisz na stosie wszystkie rejestry procesora.
  ++intr_count;  // zwiększ liczbę zagnieżdżonych przerwań
  sti();          // odblokuj przerwania
  do_irq(irq_num); // wywołaj handler.
  cli();            // zablokuj przerwania
  ACK(irq_controller) // potwierdź kontrolerowi przerwań fakt
                      // obsłużenia tego przerwania.
  --intr_count;
  goto ret_from_sys_call // O tym bedzie za chwile
}
```

- Natychmiast po zgłoszeniu przerwania przerwania są automatycznie blokowane.
- Zakładamy, że do momentu potwierdzenia przerwania kontroler nie zgłosi drugiego przerwania tego samego typu.
- ret_from_sys_call – ewentualne wywołanie planisty, obsługa dolnych połów, sygnałów i kolejek zadań.

Obsługa przerwania szybkiego - pseudokod

```
irq(irq_num,irq_controller)
{
  SAVE_MOST     // zapisz na stosie większość rejestry procesora.
  ++intr_count;  // zwiększ liczbę zagnieżdżonych przerwań
  do_irq(irq_num); // wywołaj handler.
  ACK(irq_controller) // potwierdź kontrolerowi przerwań fakt
                      // obsłużenia tego przerwania.
  --intr_count;
  RESTORE_MOST // odtwórz rejestry procesora
}
```

- Aby maksymalnie przyspieszyć wykonanie handlera nie są zachowane wszystkie rejestry.
- Zauważmy brak skoku do ret_from_sys_cal

Podprogram ret_from_sys_call

- Wywoływany każdorazowo przy powrocie z przerwania wolnego bądź wywołania systemowego.
- Jego zadania: obsługa sygnałów, uruchomienie dolnych połów, uruchomienie planisty.

```
ret_from_sys_call:
    if (intr_count) goto extit_now; //przerwanie wewnątrz przerwania
    if (bh_mask & bh_active) {
        ++intr_count;
        sti();
        do_bottom_half(); // dolna połowa nie może być przerwana
        --intr_count;     // drugą dolną połową.
    }

// przerwany proces działał w trybie jądra
    if (kernel_mode()) goto exit_now;
    if (need_resched) { // need_resched !=0 wywołuje planistę
        schedule();     // przełącz kontekst
        goto ret_from_sys_call;
    }
    if (current->signal & ~current->blocked) {
        do_signal();
    }
    exit_now: RESTORE_ALL; // odtwórz wartości rejestrów
```

Dolne połowy (ang. bottom halves)

- Często niektóre mniej ważne czynności nie muszą być wykonane od razu w procedurze przerwania.
  - Np. przeliczenie priorytetów przy obsłudze przerwania zegarowego.
  - Sytuacje w której mamy do czynienia z szybką transmisją danych.
- W procedurze obsługi przerwania możemy zażądać wykonania dolnej połowy. Jest ona wykonana przy:
  - Powrocie z wolnego przerwania.
  - Powrocie z funkcji systemowej.
- Reguły synchronizacji są następujące:
  - Kod jądra może być przerwany przez dolną połowę.
  - Dolna połowa nie może być przerwana przez inną dolną połowę.
  - Dolna połowa może być przerwana przez (dowolne) przerwanie.
  - Oczywiście można uzywać cli() oraz sti() aby zakazać wykonania przerwań i dolnych połów.

Wykorzystanie dolnych połów

- W systemie istnieją 32 dolne połowy o stałych numerach i procedurach.
- enable_bh(int nr) oraz disable_bh(int nr) włączają i wyłączają dolną połowę.
- mark_bh(int nr) zleca wykorzystanie dolnej połowy.
- Dopuszczalne numery zdefiniowane są w pliku /include/linux/interrupt.h
  - TIMER_BH, KEYBOARD_BH, NET_BH, IMMEDIATE_BH, CONSOLE_BH
- Dolne połowy mają poważne ograniczenia. Głównym jest ich statyczny charakter.
  - Określone adresy procedur.
  - Ograniczona liczba (32)
- W swoich programach możemy wykorzystać IMMEDIATE_BH. Wykorzystywana jest ona przy obsłudze kolejek zadań.

Kolejki zadań – task queues

- Są dynamicznym rozszerzeniem pojęcia dolnych połów.
  - Z dowolnego miejsca w jądrze możemy zlecić wykonanie pewnego kodu w kolejce zadań.
  - Kolejka jest listą funkcji, które mogą być wykonane w późniejszym czasie.

```
struct tq_stuct {
   struct tq_struct *next;
   int sync;

   // Adres funkcji, którą chcemy wywołać
   void (*routine)(void *);

   // Dane tej funkcji
   void *data;
}

// Kolejka zadeklarowana jako wskaźnik na pierwszy element
typedef struct tq_struct *task_queue;

// Zlecenie funkcji do wykonania w kolejce
void queue_task(struct tq_struct *task; task_queue *queue);

// makro DECLARE_TASK_QUEUE(x) deklaruje nową kolejkę
// uruchomienie (i usunięcie) wszystkich funkcji z kolejki
void run_task_queue(task_queue *list);
```

Kolejki zadań – task queues

- Wykorzystanie kolejek zadań:
  - Zadeklarować i zainicjalizować zmienną typu task_struct; pole routine ustawić na adres funkcji, pola data na adres danych które chcemy przekazać.
  - Zlecić zadanie do kolejki (gwarantuje jednorazowe (ang. single shot) wywołanie funkcji, w celu ponownego wykonania należy ponowić zlecenie)
- W Linuxie 2.0 mamy zdefiniowane cztery kolejki.
  - tq_scheduler jest wywoływana z funkcji schedule
  - tq_disk wykorzystywana przez podsystem buforowej pamięci podręcznej
  - tq_timer jest wywoływana przy każdym przerwaniu zegarowym
  - tq_immediate jest wywoływana z dolnej połowy BH_IMMEDIATE.
- Modyfikując jądro na ogół wykorzystujemy kolejki tq_timer i tq_immediate.
  - Aby uruchomić kolejkę tq_immediate należy wykonać funkcję mark_bh(BH_IMMEDIATE).
  - Uruchomienie nastąpi z programu ret_from_sys_call

Przykład - Przerwanie zegara

```
void do_timer(struct pt_regs *regs) {
   ++jiffies;
   ++lost_ticks;
   if (!user_mode(regs))
     ++lost_ticks_system;
   mark_bh(TIMER_BH);
   if (tq_timer)
      mark_bh(TQUEUE_BH); // uruchomienie kolejki tq_timer;
}
```

- Zmienna *jiffies* przechowuje liczbę przerwań zegara od momentu włączenia systemu. Można ją wykorzystać przy aktywnym czekaniu.
  - Stała HZ (100, na procesorach Alpha 1000) przechowuję częstotliwość przerwań zegara.
- Rzeczywisty kod wykonywany jest w dolnej połowie:
  - Zmniejsz pole *counter* w strukturze task_struct o wartość *lost_ticks*;
  - Ustaw *lost_ticks* na zero
  - Jeżeli *counter* przekroczyło zero ustaw zmienną *need_resched* na zero.

- Filozofia Linuksa: obsługując przerwania wykonuj tylko to co naprawdę niezbędne.
  - Pomaga przy dużym obciążeniu systemu

Dygresja – oczekiwanie w jądrze

- Jednym ze sposobów realizacji aktywnego czekania w jądrze (np. w sterowniku urządzenia) jest wstawienie kodu:

```
if (need_resched)
    schedule();
```

  - Wadą tego podejścia jest bezczynne oczekiwanie w pętli i marnowanie kwantu procesora.

- Alternatywne podejście polega na uśpieniu na co najmniej K milisekund:

```
current->state=TASK_INTERRUPTIBLE;
current->timeout=jiffies+K*HZ/1000;
schedule();
```

- Czas odliczany jest z dokładnością przerwań zegarowych (10ms na x86)
- Proces zostanie obudzony po zaprogramowanym czasie.
  - Ale procesor może otrzymać o wiele później !!!
  - O tym decyduje planista.

Wywołania systemowe

- Wywołania systemowe obsługiwane są przez przerwanie programowe 0x80. Numer wywołania służy jako indeks do tablicy sys_call_table przechowującej adresy funkcji obsługujących poszczególne wywołania.

```
int system_call(int num,args)
{
  SAVE_ALL
  if (num>=NR_SYS_CALLS)
     errno=-ENOSYS
  else if (current->flags && PF_TRACESYS) {
     syscall_trace();
     errno=(*sys_call_table[num])(args);
     syscall_trace();
  } else
     errno=(*sys_call_table[num])(args);

  goto ret_from_sys_call;
}
```

Przykłady prostych wywołań

- getpid() zwraca identyfikator danego procesu.

```
asmlinkage int sys_getpid()
{
    return current->pid;
}
```

- pause() wstrzymuje proces do momentu zgłoszenia sygnału.

```
asmlinkage int sys_pause()
{
    current->state=TASK_INTERRUPTIBLE;
    schedule();
    return -ERESTARTNOHAND; // Dopiero po sygnale
}
```

Interfejs funkcji systemowych po stronie programów użytkownika

- Plik ./include/linux/asm/unistd.h

```
// open ma numer wywołania trzy
#define __NR_getpid                    5
_syscall0(int,getpid)


Makro _sycall0 to czarna magia !!!

#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
if (__res >= 0) \
        return (type) __res; \
errno = -__res; \
return -1; \
}
```

Implementacja nowego wywołania systemowego

- Dodać adres funkcji realizującej wywołanie do tablicy (na koniec) w /arch/i386/entry.s i zwiększyć o jeden liczbę w dyrektywie .space

.space (NR_syscalls-189)*4 // zmieniamy 189 na 190

- Wywołanie systemowe może mieć maksymalnie 3 parametry, jeżeli ma więcej, niech przekazuje adres struktury z parametrami (ale pamiętajmy o verify_area !!!).
- Po stronie programów użytkownika musimy wygenerować funkcję biblioteczną wywołującą przerwanie 0x80. Dla funkcji int foo(int a,int b,char *c) z trzema argumentami robimy to w sposób następujący:

```
#define NR_foo 190
_syscall3(int,foo,int,a,int,b,char *,c)
```

- Ale należy pamiętać, że Linuks zaprojektowano w.g. filozofii „Jak najmniej wywołań systemowych”. (Trudno usunąć lub zmienić raz dodane nowe wywołanie systemowe). Przy rozszerzaniu systemu należy korzystać ze sterowników urządzeń, pseudo-systemu plików `proc` itp.