Asembler w kodzie Linuksa

Krzysztof Bonicki, Adam Dąbrowski, Miłosz Dobrowolski, Krzysztof Fajkowski

16 grudnia 2003

Spis treści

1 Wprowadzenie

Prezentacja została podzielona na trzy części:

1. Pierwsza z nich ma na celu zaznajomienie czytelnika ze składnią asemblera oraz przedstawienie w jaki sposób można korzystać z asemblera pisząc kod w C.

2. W drugiej części zostaną zaprezentowane i omówione przykłady zastosowania asemblera w kodzie Linuksa.

3. Trzecia część natomiast, podsumuje zebrane informacje o asemblerze, ze szczególnym uwzględnieniem jego wad i zalet. Podkreśli również kiedy warto korzystać z asemblera.

2 Składnia asemblera

2.1 Wstęp

Najbardziej znaną składnią asemblera jest składnia intelowska, jednakże do umieszczania wstawek asemblera w kodzie Linuksa bardziej przydaje się, przyjęta tam za standardową, składnia AT&T (używana np. przez GCC).
Zakładając, że wiekszość z Was zna podstawy asemblera, ta część prezentacji ograniczy się jedynie do wykazania różnic pomiędzy składnią intelowską a AT&T.

2.2 Porównanie składni AT&T ze składnią Intelowską

Oto wykaz miejsc, w których składnia AT&T odróżnia się od składni intelowskiej:

- Nazwy rejestrów poprzedzane są znakiem "%". Np. żeby odwołać sie do rejestru edx w obu składniach napiszemy odpowiednio:

AT&T	**Intel**
%edx	edx

- Źródło operacji zawsze występuje po lewej stronie, a cel po prawej (odwrotnie niż w przypadku składni intelowskiej). Aby przepisać zawartość rejestru edx do rejestru eax, należałoby napisać:

AT&T	**Intel**
movl %edx, %eax	mov eax, edx
instrukcja źródło, cel	instrukcja cel, źródło

- Stałe i "wartości natychmiastowe" (immediate values) są poprzedzane znakiem "$".
  Liczby w zapisie heksadecymalnym nie mogą być zapisywane w formacie <liczba>h, a

jedynym poprawnym jest zapis 0x<liczba>, przy czym liczb zaczynających się od liter nie trzeba poprzedzać dodatkowym zerem, tak jak w składni intelowskiej. Np.

AT&T	Intel
movl $0xb7, %ecx	mov ecx, 0b7h

- Po nazwie operacji powinien wystąpić znak określający rozmiar danych: “b”, “w” lub “l” (oznaczającym odpowiednio - byte, word, longword=dword). Np.

AT&T	Intel
movw %cx, %dx	mov dx, cx

- Przy adresowaniu pamięci, do której wskazuje rejestr, rejestr powinien być otoczony nawiasami okrągłymi, a nie jak w przypadku składni intelowskiej - kwadratowymi. Np. gdy chcemy przepisać dane z pamięci wskazywanej przez edx, do rejestru eax, napiszemy:

AT&T	Intel
movl (%edx), %eax	mov eax, [edx]

- Dostęp do pamięci o wyliczanym adresie, odbywa się za pomocą następującego wyrażenia:

AT&T	Intel
%segment:przesunięcie(baza, indeks, skala)	segment:[baza + indeks * skala + przesunięcie]

W tym przypadku, stałych nie poprzedzamy znakiem “$”.
Skala może przyjąć tylko wartości 1, 2, 4, lub 8.
Baza i indeks to 32-bitowe rejestry.
Przy czym wszystkie pola są opcjonalne, a jedynym ograniczeniem jest warunek, by wystąpiło chociaż jedno z pary: przesunięcie, baza.
Oto przykłady użycia:

AT&T	Intel
addl 0x1a(%eax, %edx, 0x2), %ebx	add ebx, [eax+edx*2h+1ah]
movl 4(%eax), %ebx	mov ebx, [eax + 4]

- Dostęp do adresów zmiennych statycznych z C odbywa się poprzez użycie znaku podkreślenia:

AT&T	Intel
movl $_nazwaZmiennej, %eax	mov eax, _nazwaZmiennej

Analogicznie dostajemy się do wartości podanych zmiennych (używając składni adresowania pamięci z poprzedniego punktu):

AT&T	Intel
movl _nazwaZmiennej, %eax	mov eax, [_nazwaZmiennej]

- Dalekie skoki oraz wywołania zapisuje się następująco:

AT&T	Intel
lcall/ljmp $section, $offset	call/jmp far section:offset

- podobnie jak i dalekie powroty:

AT&T	Intel
lret $modyfikator_stosu	ret far modyfikator_stosu

2.3 Inne architektury

Warto zauważyć, że zarówno składnia jak i możliwości asemblera są w dużym stopniu uzależnione od architektury komputera, na który pisany jest kod.

Ponieważ prezentacja ta skupia się wokół architektury Intela 80386, warto dla przykładu przytoczyć własności jakiejś innej architektury, w tym przypadku będzie to rodzina Hitachi H8/500.

2.3.1 Hitachi H8/500

- Znaki specjalne:
  - "!" - komentarz jednoliniowy
  - ";" - alternatywny sposób oddzielania instrukcji (oprócz znaku nowej linii)
  - "$" - nie pełni żadnej funkcji (zatem może być używany np. w nazwach)
- Aby uzyskać dostęp do rejestów, można korzystać z przedefiniowanych symboli - 'r0', 'r1', .. , 'r7'.
- Dodatkowo, dostępne są następujące rejestry:
  - cp - code pointer
  - dp - data pointer
  - bp - base pointer

  - tp - stack top pointer
  - ep - extra pointer
  - sr - status register
  - ccr - condition code register

- Wszystkie rejsetry są 16-bitowe.
- Liczby 32-bitowe można reprezentować za pomocą dwóch sąsiednich rejestrów.
- Do adresowania dalekiej pamięci należy używać wskaźników segmentowych (cp - dla licznika programu, dp - dla rejestrów r0-r3, ep - dla r4-r5, tp - dla r6-r7).
- Sposoby adresowania:
  - Rn - Rejestrowe bezpośrednie
  - @Rn - Rejestrowe pośrednie
  - @(d:8, Rn) - Rejestrowe pośrednie z 8-bitowym przesunięciem (ze znakiem)
  - @(d:16, Rn) - Rejestrowe pośrednie z 16-bitowym przesunięciem (ze znakiem)
  - @-Rn - Rejestrowe pośrednie z uprzednim zmniejszeniem
  - @Rn+ - Rejestrowe pośrednie z późniejszym zwiększeniem
  - @aa:8 - 8-bitowy adres bezwzględny
  - @aa:16 - 16-bitowy adres bezwzględny
  - #xx:8 - 8-bitowa stała (immediate)
  - #xx:16 - 16-bitowa stała (immediate)
- Rodzina H8/500 nie posiada sprzętowych liczb zmiennopozycyjnych.
- Rodzina H8/500 nie posiada żadnych dyrektyw specyficznych dla tej architektury.

2.4 Przekazywanie parametrów

Chcąc przekazać parametry do wywołania systemowego, w zależności od liczby parametrów, należy postąpić w jeden z dwóch przedstawionych sposobów:

- Jeśli chcemy przekazać co najwyżej 5 parametrów, należy je umieścić kolejno w następujących rejestrach: ebx, ecx, edx, esi, edi.
- W przypadku przekazywania większej liczby parametrów, należy je umieścić kolejno w ciągłej pamięci, a wskaźnik do niej przekazać w rejestrze ebx.

W obu przypadkach w rejestrze eax umieszczamy numer wywołania systemowego, a następnie uruchamiamy je za pomocą przerwania:

```
int $0x80
```

Poza wyżej wymienionymi, istnieje również możliwość wykonania wywołania systemowego (socket syscall) za pomocą wskazania numeru funkcji (eax), numeru podfunkcji (ebx), oraz wskaźnika do tablicy parametrów (ecx).

2.5 Wstawki asemblerowe

Używanie wstawek asemblerowych w C jest bardzo proste - wystarczy napisać

```
asm ("polecenia_asemblerowe");
```

Ewentualnie, jeśli słowo kluczowe “asm” jest już używane w naszym programie, można użyć składni:

```
__asm__ ("polecenia_asemblerowe");
```

Jeśli chcemy użyć kilku instrukcji asemblerowych w jednym poleceniu “asm”, należy je oddzielić sekwencją “\n\t”. Np.

```
asm ("pushl %eax\n\t"
     "movl $0, %eax\n\t"
     "popl %eax");
```

Przy takim wywoływaniu instrukcji asemblerowych należy pamiętać, że nie wolno nam zmieniać zawartości rejestrów, tzn. po wykonaniu wszystkich zadanych instrukcji zawartość rejestrów musi być dokładnie taka sama jak przed wywołaniem.

2.6 Rozszerzone wstawki asemblerowe

Jeśli chcemy zmieniać zawartość rejestrów, nadać im wartości początkowe, bądź przepisać ich wartości wynikowe do zmiennych, należy użyć alternatywnej składni, która wyprodukuje bardziej optymalny kod, niż jeśli operacje te wykonywalibyśmy samodzielnie:

```
asm ("instrukcje" : wyjscie : wejscie : co_zmieniane);
```

Przy czym pola “wyjście” i “wejście” mają postać listy oddzielanych przecinkami par - ujęta w cudzysłowy asemblerowa nazwa np. rejestru i nazwa zmiennej z C w nawiasach okrągłych.
Dodatkowo w polach wyjściowych teksty w cudzysłowiach są poprzedzane znakiem “=”.
Łączna liczba parametrów nie może przekroczyć 10.
W instrukcjach nazwy rejestrów są poprzedzane “%%”, a nie jak zwykle “%”.
W listach wejściowych i wyjściowych można stosować skrócone nazwy rejestrów:

Skrót	Znaczenie
a	eax / ax / al
b	ebx / bx / bl
c	ecx / cx / cl
d	edx / dx / dl
S	esi / si
D	edi / di
m	pamięć
I	stała wartość (od 0 do 31)
q	jeden z rejestrów eax, ebx, ecx, edx - przydzielany dynamicznie
r	jeden z rejestrów eax, ebx, ecx, edx, esi, edi - przydzielany dynamicznie
g	jeden z rejestrów eax, ebx, ecx, edx, bądź zmienna w pamięci - przydzielane dynamicznie
A	połączone rejestry eax i edx jako 64-bitowy integer (long long)

Dodatkowo GCC numeruje rejestry przydzielane za pomocą “q” i “r”, przydzielając im kolejne numery począwszy od 0. Zatem np. aby w instrukcji wykorzystać pierwszą taką zmienną, napiszemy “%0”, drugą - “%1” itd.
Jeśli zmieniamy wartość jakiejś zmiennej, to w liście zmian należy umieścić napis:

```
"memory"
```

Uwaga: Jeśli wstawka asemblerowa musi być wykonana dokładnie w miejscu, w którym została umieszczona, należy użyć instrukcji:

```
__asm__ __volatile__ (...);
```

Zobaczmy jak całość działa w praktyce:

```
int main(void) {
 int dwa=2, trzy=3, piec=0;

 __asm__ __volatile__ ("addl %2, %1\n"
          "movl %1, %0"
          : "=r"(piec)              // %0 -> piec
          : "r"(dwa), "r"(trzy));  // %1 <- dwa, %2 <- trzy
 // piec == 5
};
```

3 Przykłady zastosowania asemblera w kodzie Linuksa

Teraz zajmiemy się przedstawieniem kilku przykładów użycia asemblera w jądrze linuksa.

3.1 system_call

Szerokie zastosowanie asemblera widzimy w obsłudze wywołań systemowych (sys_call), o czym teraz opowiem:

Najpierw dokonam opisu pomocniczych makrodefinicji, z których korzysta funkcja system_call.

3.1.1 Makro SAVE_ALL:

Makro to służy zachowaniu na stosie wszystkich rejestrów procesora, które mogą zostać użyte przez konkretną procedurę (w naszym przypadku system_call, ale to nie jest jedyne miejsce użycia tego makra). Makro to nie zapisuje natomiast eflags, cs, eip, ss, esp ponieważ one są automatycznie zachowywane przez jednostkę sterowania. Po odłożeniu wszystkiego na stos, makro ładuje na ds i es selektor segmentu danych jądra.

Oto treść makra SAVE_ALL:

```
#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS),%edx; \
        movl %dx,%ds; \
        movl %dx,%es;
```

3.1.2 Makro RESTORE_ALL:

Makro to ładuje do rejestrów wartości zachowane przez makro SAVE_ALL i przekazuje sterowanie do przerwanego programu poprzez wykonanie instrukcji iret.

A oto treść makra:

```
#define RESTORE_ALL     \
        popl %ebx;      \
        popl %ecx;      \
        popl %edx;      \
        popl %esi;      \
        popl %edi;      \
        popl %ebp;      \
        popl %eax;      \
1:      popl %ds;       \
2:      popl %es;       \
        addl $4,%esp;   \
3:      iret;           \
```

3.1.3 Makro GET_CURRENT:

To makro służy do pobrania deskryptora aktualnego procesu. Wykonuje to, pobierając wskaźnik stosu jądra i zaokrągla go do wielokrotności 8KB (ponieważ jak nam wiadomo w strukturze task_union na początku mamy deskryptor procesu, a na końcu od 8KB licząc, stos)

Oto treść makra GET_CURRENT:

```
#define GET_CURRENT(reg) \
        movl %esp, reg; \
        andl $-8192, reg;
```

3.1.4 Funkcja system_call:

Teraz już możemy przejść do właściwej funkcji system_call(). Zostaną opisane najważniejsze z niej kawałki.

- system_call

    1. Na początku funkcji, zachowuje na stosie numer wywołania systemowego oraz część rejestrów procesora poprzez wywołanie makra SAVE_ALL:

```
ENTRY(system_call)
        pushl %eax              # save orig_eax
        SAVE_ALL
```

2. Następnie pobiera deskryptor procesu i zachowuje go w ebx:

```
GET_CURRENT(%ebx)
```

3. Kolejną rzeczą jest sprawdzenie poprawności numeru wywołania systemowego, przekazanego przez proces trybu użytkownika czyli innymi słowy numeru wywoływanej funkcji systemowej. Jeżeli jest on większy lub równy od ilości funkcji systemowych, to procedura skacze do etykiety badsys, gdzie następnie kończy się.

```
cmpl $(NR_syscalls),%eax
jae badsys
```

4. Następnie system_call() sprawdza, czy flaga PT_TRACESYS, która mówi czy wywołania systemowe są śledzone przez program odpluskwiający, jest ustawiona. Jeśli tak to skaczemy do tracesys, gdzie system_call() dwa razy wywołuje funkcję syscall_trace(): raz przed i raz po wywołaniu podprogramu obsługi wywołania systemowego. Funkcja ta zatrzymuje aktualny proces, co pozwala procesowi śledzącemu na zebranie informacji o nim.

```
testb $0x02,ptrace(%ebx)        # PT_TRACESYS
jne tracesys
```

5. No i nareszcie jest wywoływany podprogram obsługi związany z zawartym w eax numerem wywołania systemowego:
   Każda pozycja tabeli rozdzielczej ma 4 bajty długości więc jądro znajduje adres odpowiedniego podprogramu obsługi mnożąc najpierw numer wywołania systemowego przez 4 i dodając adres początkowy tablicy sys_call_table, a następnie wyłuskuje wskaźnik do podprogramu obsługi z pozycji tablicy.

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

6. Po powrocie z programu obsługi, system_call() pobiera zwrócony kod z eax, a następnie zachowuje go w tym miejscu na stosie, gdzie zachowana została wartość rejestru eax z trybu użytkownika (służy do tego proste makro EAX).

```
movl %eax,EAX(%esp)       # save the return value
```

- ret_from_sys_call
  Teraz już możemy wrócić z sys_call'a, czyli kończymy wykonywanie procedury obsługi w bloku ret_from_sys_call.
  Najpierw sprawdzamy zmienne bh_mask i bh_active, aby dowiedzieć się czy są jakieś aktywne, nie zamaskowane dolne połowy (Dolna połowa jest niskopriorytetową funkcją, zazwyczaj związaną z obsługą przerwań, która czeka, aż jądro znajdzie odpowiednią chwilę na jej uruchomienie). Jeśli trzeba wykonać jakieś dolne połowy, wykonywany jest skok pod etykietkę handle_bottom_half.

```
ret_from_sys_call:
        movl SYMBOL_NAME(bh_mask),%eax
        andl SYMBOL_NAME(bh_active),%eax
        jne handle_bottom_half
```

- ret_with_reschedule
  Jeśli nie ma już żadnych dolnych połów to wykonujemy następujące czynności.

  1. Rejestr ebx wskazuje na deskryptor aktualnego procesu. W tym deskryptorze za pomocą funkcji need_resched() sprawdzamy czy jest ustawiona flaga mówiąca, czy należy wykonać schedule().

     ```
     ret_with_reschedule:
             cmpl $0,need_resched(%ebx)
     ```

  2. Jeśli flaga jest ustawiona to należy wykonać reschedule:

     ```
             jne reschedule
     ```

  3. W przeciwnym wypadku idziemy dalej. Sprawdzamy pole sigpending w deskryptorze. Jeżeli jest puste, to aktualny proces wznawia wykonanie w trybie użytkownika. W przeciwnym wypadku kod wykonuje skok do signal_return w celu przetworzenia sygnałów aktualnego procesu. Tam odbywa się sprawdzenie w jakim trybie pracował proces i dalej wywołanie funkcji do_signal, która obsłuży czekające sygnały.

     ```
             cmpl $0,sigpending(%ebx)
             jne signal_return
     ```

- restore_all
  Odtwarzamy wszystkie rejestry, które zapamiętaliśmy przed rozpoczęciem system_call(). Korzystamy ze zdefiniowanego makra RESTORE_ALL

  ```
  restore_all:
          RESTORE_ALL
  ```

- badsys
  Jeżeli numer wywołania systemowego nie jest prawidłowy, to funkcja zachowuje wartość -ENOSYS w tym miejscu stosu, gdzie została zachowana wartość rejestru eax. Do tego miejsca mamy dostęp poprzez makro EAX, które przesuwa się o 18 pozycji względem aktualnego wierzchołka stosu. Następnie wykonywany jest skok do ret_from_sys_call(). Dzięki temu zabiegowi, gdy proces znowu zacznie się wykonywać w trybie użytkownika, znajdzie ujemny kod błędu w eax.

```
badsys:
        movl $-ENOSYS,EAX(%esp)
        jmp ret_from_sys_call
```

- ret_from_intr
  Powrót z przerwania realizowany jest następująco:

  1. Pobieramy adres deskryptora aktualnego procesu i przypisujemy na ebx.

     ```
     ret_from_intr:
             GET_CURRENT(%ebx)
     ```

  2. Następnie używając wartości rejestrów cs i eflags (pobranych dzięki użytecznym makrom EFLAGS i CS), które zostały włożone na stos w czasie wystąpienia przerwania, sprawdzamy czy przerwany proces działał w trybie jądra.

     ```
             movl EFLAGS(%esp),%eax      # mix EFLAGS and CS
             movb CS(%esp),%al
             testl $(VM_MASK | 3),%eax   # return to VM86 mode
                                              or non-supervisor?
     ```

  3. Jeśli nie, to skaczemy do etykietki ret_with_reschedule.

     ```
             jne ret_with_reschedule
     ```

  4. Jeśli natomiast proces przerwany działał w trybie jądra, to znaczy, że wystąpiło zagnieżdżenie przerwań i przerwana ścieżka wykonania jądra jest wznawiana poprzez wykonanie kodu z makrodefinicji RESTORE_ALL (przywrócenie wartości rejestrów sprzed wywołania).

     ```
             jmp restore_all
     ```

- handle_bottom_half

  Jeśli mamy czekające dolne połowy, to wykonujemy wywołanie funkcji do_bottom_half, która rozpoczyna wykonywanie wszystkich aktywnych, nie zamaskowanych dolnych połów. Następnie (po wykonaniu wszystkich dolnych połów) skaczemy do ret_from_intr.

```
handle_bottom_half:
        call SYMBOL_NAME(do_bottom_half)
        jmp ret_from_intr
```

3.2 switch_to

Kolejnym przykładem asemblera w kodzie jądra linuksa jest makro switch_to. Makro to wykonuje przełączenia procesów i jest wywoływane na końcu funkcji schedule().

- Opis makra switch_to: Używane są dwa parametry, oznaczone jako prev (wskaźnik do deskryptora procesu, który ma być uśpiony) i next (wskaźnik do deskryptora procesu, który ma być wykonywany przez procesor.
- Poniżej zamieszczony jest kod makra switch_to wraz z opisem działania:

1. Definiujemy nagłówek funkcji:

```
#define switch_to(prev,next,last) do {                  \
```

2. Na samym początku zachowujemy zawartość rejestrów esi, edi i ebp w stosie trybu jądra prev. Muszą one zostać zachowane, ponieważ kompilator zakłada, że nie zostaną zmienione aż do końca switch_to.

```
    asm volatile("pushl %%esi\n\t"                  \
                 "pushl %%edi\n\t"                  \
                 "pushl %%ebp\n\t"                  \
```

3. Dalej zachowana zostaje zawartość esp w pierwszym parametrze wyjściowym czyli w prev->tss.esp tak, aby pole to wskazywało na wierzchołek stosu trybu jądra prev.

```
                 "movl %%esp,%0\n\t"   /* save ESP */  \
```

4. Następnie ładowany jest pierwszy parametr wejściowy czyli next->tss.esp do esp. Od tej chwili jądro zaczyna operować na stosie jądra next, tak więc właściwie ta instrukcja wykonuje główne przełączenie kontekstu z prev na next. Zmienianie stosu jądra zmienia również aktualny proces, ponieważ adres deskryptora procesu jest ściśle powiązany z adresem stosu trybu jądra.

```
                 "movl %3,%%esp\n\t"  /* restore ESP */  \
```

5. Teraz zachowuje adres oznaczony 1 w prev->tss.eip. Gdy proces usypiany wznowi wykonanie, wykona instrukcję oznaczoną etykietką 1.

```
                 "movl $1f,%1\n\t"    /* save EIP */     \
```

6. Teraz do stosu trybu jądra next, wstawiana jest wartość next->tss.eip.

```
"pushl %4\n\t"      /* restore EIP */   \
```

7. Następnie wykonujemy skok do funkcji __switch_to. Funkcja ta dopełnia to, co zaczęło robić makro switch_to. Opcjonalnie zachowuje zawartość koprocesora matematycznego. Zachowuje zawartość rejestrów segmentacji fs i gs w prev->tss.fs i prev->tss.gs. Zmienia także wskaźnik do lokalnej tablicy deskryptorów i do katalogu stron jeśli trzeba.

```
"jmp __switch_to\n"                         \
```

8. Pozostało nam już tylko przywrócić zawartość rejestrów esi, edi i ebp. Ale zauważmy, że te operacje przywrócenia zostaną wykonane dopiero, gdy planista wybierze prev jako nowy proces do wykonania przez procesor, co wywoła switch_to z prev jako drugim parametrem. Tak więc rejestr esp będzie wskazywał na stos trybu jądra prev.

```
"1:\t"                                      \
"popl %%ebp\n\t"                            \
"popl %%edi\n\t"                            \
"popl %%esi\n\t"                            \
```

9. Podajemy do makra parametry wyjściowe.

```
:"=m" (prev->tss.esp),"=m" (prev->tss.eip), \
 "=b" (last)                                \
```

10. I przekazujemy parametry wejściowe.

```
:"m" (next->tss.esp),"m" (next->tss.eip),   \
 "a" (prev), "d" (next),                    \
 "b" (prev));                               \
} while (0)
```

3.3 io.h

Teraz zajmiemy się jednym z najbardziej użytecznych przykładów kodu asemblera w Linuksie. Mianowicie operacjami czytania i pisania do urządzeń, zdefiniowanymi w /usr/include/asm/io.h. Makro to służy do wygenerowania funkcji odpowiedzialnych za bezpośredni kontakt z urządzeniem w zależności od rozmiaru pobieranych/zapisywanych porcji danych.

Najpierw zdefiniowane widzimy makra:

3.3.1 Makro SLOW_DOWN_IO:

Makro do spowolnienia transmisji z urządzeniem.

1. kiedy mamy ustawiony tryb pracy na zwalnianie poprzez jmp, to pod makrem __SLOW_DOWN_IO będziemy rozumieli kawałek kodu asemblera:

   ```
           jmp     1
   1:      jmp     1
   1:
   ```

   co po prostu odciąży nam transmisję z urządzeniem. W io.h widzimy stosowny kod:

   ```
   #ifdef SLOW_IO_BY_JUMPING
   #define __SLOW_DOWN_IO "\njmp 1f\n1:\tjmp 1f\n1:"
   ```

2. W przeciwnym przypadku pod tym makrem rozumiemy odpowiedni kawałek kodu:

   ```
           outb    %%al, 0x80
   ```

   co jest poprostu pustą instrukcją. I do tego odpowiedni kod w bibliotece:

   ```
   #else
   #define __SLOW_DOWN_IO "\noutb %%al,$0x80"
   #endif
   ```

3.3.2 Makro FULL_SLOW_DOWN_IO:

1. Jeśli mamy tryb REALLY_SLOW_IO, to definiujemy operację FULL_SLOW_DOWN_IO, która jest po prostu czterokrotnym wykonaniem operacji spowolnienia.

   ```
   #ifdef REALLY_SLOW_IO
   #define __FULL_SLOW_DOWN_IO
                   __SLOW_DOWN_IO
                   __SLOW_DOWN_IO
                   __SLOW_DOWN_IO
                   __SLOW_DOWN_IO
   ```

2. Następnie, jeśli nie mamy trybu REALLY_SLOW_IO, jest to jednokrotną operacją spowolnienia.

   ```
   #else
   #define __FULL_SLOW_DOWN_IO
                   __SLOW_DOWN_IO
   #endif
   ```

3.3.3 Makro OUTs:

Teraz zajmiemy się właściwymi funkcjami, które oferuje nam io.h Najpierw przyjrzyjmy się funkcji out(-b, -w, -l):

1. Jako pierwsze definiujemy sobie makro __OUT1(s,x), które przyjmuje dwa parametry:

   - s - jaki mamy rodzaj operacji (b, w, l). Czyli odpowiednia operacja długości byte, word, long (dword).
   - x - typ parametru, char(b) short(w), int(l)

   Tym sposobem mamy zdefiniowany nagłówek funkcji outs, gdzie s może być -b, -w, -l w zależności od tego, jak dużo bajtów chcemy skopiować do urządzenia; value, które jest tym, co chcemy zapisać; i port, który jest adresem portu urządzenia, do którego chcemy zapisać.

   ```
   #define __OUT1(s,x) \
   extern inline void out##s(unsigned x value, unsigned short port) {
   ```

2. Teraz definiujemy sobie treść funkcji out. Składa się ona właściwie tylko z jednej instrukcji:

   ```
           out#s   %s1"0",%s2"1"
   ```

   w zależności od wartości s mamy odpowiednią instrukcję: outb, outw, outl, które zapisują odpowiednio 1, 2 lub 4 kolejne bajty do portu IO. Poza tym, jako źródło tego, co kopiujemy jest zerowy parametr funkcji outs z przedrostkiem, określającym ilość bajtów, na których trzymana jest ta zmienna. Jako port IO, do którego będziemy kopiować jest użyty pierwszy parametr wywołania z przedrostkiem odpowiednim dla wielkości.

   ```
   #define __OUT2(s,s1,s2) \
   __asm__ __volatile__ ("out" #s " %" s1 "0,%" s2 "1"
   ```

3. Teraz pozostaje nam stworzyć definicję funkcji outs zebranej w całości. Najpierw zbieramy definicję tej funkcji w całość dla ciągłego zapisu do urządzenia:

   ```
   #define __OUT(s,s1,x) \
   __OUT1(s,x) __OUT2(s,s1,"w")
               :
               : "a" (value), "Nd" (port)); } \
   ```

   Jak widzimy makro to nazwiemy __OUT, z parametrami:

- s - typ funkcji out (b, w, l)
- s1 - wielkość źródła z którego kopiujemy ("b", "w", ł")
- x - typ źródła (char, short, int)

I najpierw wstawiamy treść deklaracji funkcji __OUT1(s,x), a następnie wstawiamy samą treść funkcji __OUT2(s,s1,"w"), gdzie zauważamy, że port urządzenia będziemy zawsze przedstawiać jako "w". W parametrach wyjściowych nic nie definiujemy, natomiast parametrami wejściowymi są:

- 0: na “eax” - value - czyli to, skąd chcemy zapisać
- 1: na “Nd” - port - czyli port urządzenia, do którego piszemy.

Dla przykładu nasza funkcja kopiująca po bajcie może wyglądać:

```
extern inline void outb(unsigned char value, unsigned short port) {
  __asm__ __volatile__ ("outb %b0,%w1"
            :
            : "a" (value), "Nd" (port));
}
```

4. Jednak definiujemy jeszcze trzy funkcje, u których każdy zapis do urządzenia będzie przerywany pauzami.

```
__OUT1(s##_p,x) __OUT2(s,s1,"w") __FULL_SLOW_DOWN_IO
            :
            : "a" (value), "Nd" (port));} \
```

czyli początek naszych funkcji jest podobny jak wyżej, z taką różnicą, że nazwa funkcji ma sufiks _p, czyli: outb_p, outw_p, outl_p. Następnie wstawiana jest identycznie treść, a po wykonaniu zapisu, wykonujemy jeszcze wcześniej zdefiniowane makro __FULL_SLOW_DOWN_IO.

3.3.4 Makro INs:

Poza instrukcją out, mamy zupełnie do niej analogiczną instrukcję in, w której również występują wszystkie możliwe warianty (b, w, l, b_p, w_p, l_w):

1. Z tą jednak różnicą, że tutaj będziemy zwracali to, co odczytamy typu RETURN_TYPE (o czym za chwilę). Widzimy że w parametrach mamy port urządzenia i zaraz na początku funkcji deklarujemy sobie zmienną, którą zwrócimy jako wynik funkcji.

```
#define __IN1(s) \
extern inline RETURN_TYPE in##s(unsigned short port) {RETURN_TYPE _v;
```

2. Dalej treść funkcji, czyli poleceniem in (b, w, l) kopiujemy odpowiednią ilość bajtów z parametru 1 (portu), na 0 parametr z odpowiednimi przedrostkami.

```
#define __IN2(s,s1,s2) \
__asm__ __volatile__ ("in" #s " %" s2 "1,%" s1 "0"
```

3. Podobnie jak poprzednio składamy funkcje w całość, gdzie najpierw występuje deklaracja funkcji ins, później treść (in...), następnie jako sekcja out mamy zmienną zadeklarowaną w __IN1(s,x), sekcja in to port i zmienna i. Na koniec zwracamy wynik instrukcji ins.

```
#define __IN(s,s1,i...) \
__IN1(s) __IN2(s,s1,"w")
           : "=a" (_v)
           : "Nd" (port) ,##i ); return _v; } \
```

4. I zupełnie analogicznie dla _p, z przerwą po operacji kopiowania.

```
__IN1(s##_p) __IN2(s,s1,"w") __FULL_SLOW_DOWN_IO
           : "=a" (_v)
           : "Nd" (port) ,##i
); return _v; } \
```

3.3.5 Makro INSs:

Teraz pozostały nam jeszcze operacje cykliczne. Czyli kilkukrotne pobranie lub wysłanie danych do urządzenia. Spójrzmy na operację __INS:

1. Opisujemy deklarację funkcji ins(b, w, l), która jako parametry przyjmuje numer portu, miejsce do storowania odebranych danych addr oraz liczbę powtórzeń operacji odczytu.

```
#define __INS(s) \
extern inline void ins##s(unsigned short port,
                          void * addr,
                          unsigned long count) \
```

2. Jako właściwą treść naszej funkcji wielokrotnego odczytu mamy: najpierw instrukcją cld zerujemy flagi, co sprawi, że będziemy zwiększali pozycję w ES:EDI przy zapisie. Następnie mamy instrukcję pętli rep, która wykona instrukcję za nią CX razy. No i na koniec właściwa instrukcja pobierająca z portu urządzenia trzymanego na DX dane i zapisująca do ES:EDI. Teraz zostało już tylko wysłać odpowiednie parametry do tej instrukcji.

   - na EDI trzymamy *addr i tam będziemy zapisywać dane z urządzenia

- na CX trzymamy parametr count i tyle razy pobierzemy dane z urządzenia
- na DX trzymamy numer portu urządzenia

W rezultacie mamy już pełną funkcję inss, która count razy pobierze dane z portu o numerze port.

```
{ __asm__ __volatile__ ("cld ; rep ; ins" #s \
                        : "=D" (addr), "=c" (count)
                        : "d" (port),"0" (addr),"1" (count)); }
```

3.3.6 Makro OUTSs:

Teraz pozostała nam już tylko analogiczna funkcja outs(b, w, l), która sekwencyjnie zapisze kolejne (1, 2, 4) bajtów do urządzenia:

1. Deklarujemy nagłówek funkcji. Funkcja ma nazwę odpowiednio do s (outsb, outsw, outsl). Jako parametry pierwszy dostajemy numer portu urządzenia, później wskaźnik addr do danych, które chcemy przekopiować i ilość kopiowań, które mają się odbyć.

   ```
   #define __OUTS(s) \
   extern inline void outs##s(unsigned short port,
                              const void * addr,
                              unsigned long count) \
   ```

2. Teraz podobnie jak przy odczycie z urządzenia, postępujemy tutaj. Najpierw czyścimy flagi (cld), czyli będziemy się przesuwali zwiększając wskaźnik do ES:EDI. Następnie CX razy wykonujemy out(b, w, l), kopiując odpowiednią ilość danych (1, 2, 4 bajtów) z EDI do DX. I teraz już tylko wystarczy nadać odpowiednie parametry instrukcji asemblera. Czyli na EDI trzymamy *addr z danymi, na CX count, a na DX numer port.

   ```
   { __asm__ __volatile__ ("cld ; rep ; outs" #s \
                           : "=S" (addr), "=c" (count)
                           : "d" (port),"0" (addr),"1" (count)); }
   ```

3.3.7 Tworzenie funkcji:

Tak naprawdę, w tym miejscu pliku znajduje się właściwe utworzenie wszystkich funkcji wejścia/wyjścia do urządzeń, korzystając z wyżej zdefiniowanych makr. tworzymy następujące funkcje:

```
#define RETURN_TYPE unsigned char
__IN(b,"")
#undef RETURN_TYPE
#define RETURN_TYPE unsigned short
__IN(w,"")
#undef RETURN_TYPE
#define RETURN_TYPE unsigned int
__IN(l,"")
#undef RETURN_TYPE
```

Definicje wartości zwracanych przez funkcje w zależności od typu funkcji. Dla inb, inb_p jest to char, i dalej analogicznie. A następnie dla odpowiednio zdefiniowanego RETURN_TYPE tworzymy stosowną funkcję INs.

```
__OUT(b,"b",char)
__OUT(w,"w",short)
__OUT(l,,int)
```

Do wykreowania funkcji outs używamy następujących wywołań, które stworzą: outb dla value char i typu "b", i dalej analogicznie.

```
__INS(b)
__INS(w)
__INS(l)

__OUTS(b)
__OUTS(w)
__OUTS(l)
```

I to samo robimy dla funkcji ins i outs.

3.4 Operacje atomowe w asemblerze AT&T

/asm/atomic.h

Niepodzielność operacji jest najlepszą metodą zapobiegania wyściwgom np. przy implementacji mechanizmów tworzących np. sekcje krytyczne. Operacje niepodzielne, jak wiadomo są czymś co można wykonać bez obawy, że zostaną przerwane zanim wykonają swoje zadanie.
Naturalnie niepodzielne są te instrukcje, które pobierają coś z pamięci raz. Jednak instrukcje które czytają/modyfikują coś w pamięci nie mają zapewnionej niepodzielności. Pomiędzy odczytywaniem i zapisywaniem pamięci inny procesor może przejąć szynę danych. dlatego asembler umożliwia zablokować szynę danych.
Przyjrzyjmy się przykładom z pliku /asm/atomic.h.

```
#include <linux/config.h>
```

config.h includuje następnie autoconfig.h, w którym jest seria deklaracji
Jeśli działamy na systemie wieloprocesorowym

```
#ifdef CONFIG_SMP
```

Definiujemy LOCK, jako asemblerową instrukcję ”lock ;”, zapewniającą niepodzielność instrukcji przy systemie wieloprocesorowym.

```
#define LOCK "lock ; "
```

w przeciwnym przypadku

```
#else
```

Nie będziemy korzystać z tej asemblerowej instrukcji, LOCK będzie pusty (dzięki temu nie musimy wewnątrz programów w C sprawdzać architektury)

```
#define LOCK ""
#endif
```

Definiujemy strukturę, która zapewni nam, że dane w niej trzymane będą pod dokładnie podanym adresem, że kompilator nie będzie nam tutaj "pomagał".

```
typedef struct { volatile int counter; } atomic_t;
```

atomowe odczytanie wartości, następuje poprzez instrukcję :

```
#define atomic_read(v) ((v)->counter)
```

analogicznie ustawienie zmiennej

```
#define atomic_set(v,i) (((v)->counter) = (i))
```

aby atomowo dodać wartość do atomowej wartości używamy funkcji :

```
static __inline__ void atomic_add(int i, atomic_t *v)\\
```

Używamy asemblera :

```
__asm__ __volatile__(

LOCK "addl %1,%0"
:"=m" (v->counter)
:"ir" (i), "m" (v->counter));
}
```

Oczywiście analogicznie odejmowanie :

```
static __inline__ void atomic_sub(int i, atomic_t *v)
{
__asm__ __volatile__(
LOCK "subl %1,%0"
:"=m" (v->counter)
:"ir" (i), "m" (v->counter));
}
```

w kolejnej funkcji

```
static __inline__ int atomic_sub_and_test(int i, atomic_t *v)
{
unsigned char c;

__asm__ __volatile__(
LOCK "subl %2,%0; sete %1"
:"=m" (v->counter), "=qm" (c)
:"ir" (i), "m" (v->counter) : "memory");
return c;
}
```

Dodajemy jeszcze instrukcję *sete %1* która dokonuje nam sprawdzenia, czy odejmowanie się udało (sete) i ustawia odpowiedni parametr (%1) w zależności od wyniku tego sprawdzenia.

Po omówieniu powyższych funkcji widać od razu jak będą wyglądały pozostałe funkcje zawarte w atomic.h :

```
static __inline__ void atomic_inc(atomic_t *v)
static __inline__ void atomic_dec(atomic_t *v)
static __inline__ int atomic_dec_and_test(atomic_t *v)
static __inline__ int atomic_add_negative(int i, atomic_t *v)
static __inline__ int atomic_inc_and_test(atomic_t *v)
```

3.5 Semafory

/asm/atomic.h

Kolejnym miejscem, gdzie używa się asemblera są operacje na semaforach. Przyjrzyjmy się dokładniej funkcjom korzystającym z systemowych semaforów: (semaphore.h)

Struktura opisująca semafory (struct semaphore) składa się z pól :

- count - przechowuje wartość całkowitą. Jeżeli jest większa niż zero, to zasób jest wolny, równa zero oznacza, że zasób jest zajęty i nikt na niego nie czeka, mniejsza niż zero - oznacza ile ścieżek czeka na zajęty zasób.
- wait - przechowuje adres listy kolejki oczekiwania, gdzie oczekują uśpione procesy czekające na zasób.
- waking - Zapewnia, że przy budzeniu procesów, tylko jeden z nich otrzyma zasób.

```
static inline void down(struct semaphore * sem)
{
#if WAITQUEUE_DEBUG
CHECK_MAGIC(sem->__magic);
#endif

__asm__ __volatile__(
"# atomic down operation\n\t"
LOCK "decl %0\n\t"     /* --sem->count */
"js 2f\n"
"1:\n"
".subsection 1\n"
".ifndef _text_lock_" __stringify(KBUILD_BASENAME) "\n"
"_text_lock_" __stringify(KBUILD_BASENAME) ":\n"
".endif\n"
"2:\tcall __down_failed\n\t"
"jmp 1b\n"
".subsection 0\n"
:"=m" (sem->count)
:"c" (sem)
:"memory");
}
```

Warto zacząć od wyjaśnienia słowa kluczowego *volatile*. Używamy go, gdy chcemy aby kod pozostał w nienaruszonym stanie przez kompilator.
Instrukcja *__asm__* jest równoważna instrukcjom *asm* i *__asm* (używana jest dla celów kompatybilności ze starszymi programami - podobnie z volatile)

W zależności od tego jak jest zainicjowana wcześniej już omówiona zmienna LOCK, czyli w zależności od tego, czy działamy na systemie wieloprocesorowym - zapewnia niepodzielność operacji występujścej po nim
tu :

```
decl %0\n\t
```

czyli zmniejszenie parametru z sekcji ”output” - tu :

```
:"=m" (sem->count)
```

- czyli wartości count w strukturze implementującej semafory. Jeśli wartość count >= 0 to zasób jest zajmowany, w przeciwnym wypadku wstawia się go do kolejki oczekujących. (przechodzimy do sekcji drugiej, która wywołuje *__down_failed*)
Użyte tu literki po numerze etykiety oznaczają, czy dana etykieta znajduje się z przodu kodu (literka "f" od forward) czy z tyłu ("b" od back).

```
asm(
".text\n"
".align 4\n"
".globl __down_failed\n"
"__down_failed:\n\t"
#if defined(CONFIG_FRAME_POINTER)
        "pushl %ebp\n\t"
        "movl  %esp,%ebp\n\t"
#endif
        "pushl %eax\n\t"
        "pushl %edx\n\t"
        "pushl %ecx\n\t"
        "call __down\n\t"
        "popl %ecx\n\t"
        "popl %edx\n\t"
        "popl %eax\n\t"
#if defined(CONFIG_FRAME_POINTER)
        "movl %ebp,%esp\n\t"
        "popl %ebp\n\t"
#endif
        "ret"
);
```

Ta z kolei, odkłada rejestry na stos i wywołuje funkcję z C, która właśnie wepchnie nas do kolejki, następnie (to już po obudzeniu zdejmie włożone na stos rejestry i zwróci sterowanie do programu)

```
static spinlock_t semaphore_lock = SPIN_LOCK_UNLOCKED;

void __down(struct semaphore * sem)
{
        struct task_struct *tsk = current;
        DECLARE_WAITQUEUE(wait, tsk);
        tsk->state = TASK_UNINTERRUPTIBLE;
        add_wait_queue_exclusive(&sem->wait, &wait);
```

```
        spin_lock_irq(&semaphore_lock);
        sem->sleepers++;
        for (;;) {
                int sleepers = sem->sleepers;

                /*
                 * Add "everybody else" into it. They aren't
                 * playing, because we own the spinlock.
                 */
                if (!atomic_add_negative(sleepers - 1, &sem->count)) {
                        sem->sleepers = 0;
                        break;
                }
                sem->sleepers = 1;      /* us - see -1 above */
                spin_unlock_irq(&semaphore_lock);

                schedule();
                tsk->state = TASK_UNINTERRUPTIBLE;
                spin_lock_irq(&semaphore_lock);
        }
        spin_unlock_irq(&semaphore_lock);
        remove_wait_queue(&sem->wait, &wait);
        tsk->state = TASK_RUNNING;
        wake_up(&sem->wait);
}
```

Bardzo analogicznie wygląda sprawa z podnoszeniem semafora :

```
static inline void up(struct semaphore * sem)
{
#if WAITQUEUE_DEBUG
CHECK_MAGIC(sem->__magic);
#endif
__asm__ __volatile__(
"# atomic up operation\n\t"
LOCK "incl %0\n\t"     /* ++sem->count */
"jle 2f\n"
"1:\n"
".subsection 1\n"
".ifndef _text_lock_" __stringify(KBUILD_BASENAME) "\n"
"_text_lock_" __stringify(KBUILD_BASENAME) ":\n"
".endif\n"
"2:\tcall __up_wakeup\n\t"
```

```
"jmp 1b\n"
".subsection 0\n"
:"=m" (sem->count)
:"c" (sem)
:"memory");
}
```

I znów - niepodzielnie tym razem zwiększamy wartość pola count w semaforze i jeśli jest dodatnia, to wywołujemy procedurę *__up_wakeup*, która wywołuje procedurę *__up* - ta zaś budzi procesy uśpione kolejce.

```
asm(
".text\n"
".align 4\n"
".globl __up_wakeup\n"
"__up_wakeup:\n\t"
        "pushl %eax\n\t"
        "pushl %edx\n\t"
        "pushl %ecx\n\t"
        "call __up\n\t"
        "popl %ecx\n\t"
        "popl %edx\n\t"
        "popl %eax\n\t"
        "ret"
);



__up(struct semaphore *sem)
{
        wake_up(&sem->wait);
}
```

3.6 Czytanie / pisanie z przestrzeni adresowej procesu

/asm/uaccess.h

Innym ważnym i ciekawym przykładem użycia asemblera jest, sytuacja, gdy proces potrzebuje zapisać lub odczytać coś z przestrzeni adresowej procesu. Służą do tego makrodefinicje get_user i put_user, pozwalające na odczytanie / zapisanie 1,2, lub 4 kolejnych bajtów.

Funkcje te pobierają dwa argumenty (x, ptr) rozmiar zmiennej wskazywanej przez ptr powoduje automatycznie wybranie odpowiedniej funkcji **__get_user_1**, **__get_user_2**, **__get_user_3**, lub **__get_user_4**.

```
#define get_user(x,ptr) \
({ int __ret_gu,__val_gu; \
```

Wielkość wskaźnika

```
switch(sizeof (*(ptr))) { \
```

Wywołujemy funkcję z odpowiednią wielkością, przekazując parametry :

```
case 1:  __get_user_x(1,__ret_gu,__val_gu,ptr); break; \
case 2:  __get_user_x(2,__ret_gu,__val_gu,ptr); break; \
case 4:  __get_user_x(4,__ret_gu,__val_gu,ptr); break; \
default: __get_user_x(X,__ret_gu,__val_gu,ptr); break; \
} \
```

W zależności od wyniku ustawiamy x i zwracamy wartość :

```
(x) = (__typeof__(*(ptr)))__val_gu; \
__ret_gu; \
})

#define __get_user_x(size,ret,x,ptr) \
__asm__ __volatile__("call __get_user_" #size \
```

Wywołujemy odpowiednie **__get_user_** l , gdzie l to przekazana wielkość (size)

```
:"=a" (ret),"=d" (x) \
:"0" (ptr))
```

__get_user_ l omówimy na przykładzie **__get_user_4** :

```
addr_limit = 12
.
.
.
.globl __get_user_4
__get_user_4:
addl $3,%eax
movl %esp,%edx
jc bad_get_user
andl $0xffffe000,%edx
cmpl addr_limit(%edx),%eax
jae bad_get_user
3: movl -3(%eax),%edx
```

```
xorl %eax,%eax
ret

bad_get_user:
xorl %edx,%edx
movl $-14,%eax
ret
```

Przed wywołaniem tego makra, rejestr eax zawiera adres *ptr* pierwszego bajta, który ma zostać przeczytany.
Instrukcje :

```
addl $3,%eax
jc bad_get_user
```

Sprawdzają, czy 4 bajty mają adresy mniejsze niż 4GB. Następnie dokonujemy sprawdzenia, czy są mniejsze niż pole addr_limit.seg dla aktualnego procesu. Pole to jest przechowywane na pozycji 12 w deskryptorze procesu - tę wartość opisuje nam zadeklarowane wcześniej *addr_limit*).

```
movl %esp,%edx
andl $0xffffe000,%edx
cmpl addr_limit(%edx),%eax
jae bad_get_user
```

Drugi raz już odwołujemy się do etykiety bad_get_user, a co ona robi ? zeruje rejestr edx (bo adresy nie były prawidłowe) a na rejestr eax wrzuca kod błędu -EFAULT i kończy pracę.

Jeśli jednak wywołanie było prawidłowe, to funkcja zachowuje dane do przeczytania w edx :

```
movl -3(%eax),%edx
```

Następnie zeruje eax (zwrócone 0 będzie oznaczać poprawne wykonanie makra) i kończy pracę.

```
xorl %eax,%eax
ret
```

put_user zachowuje się dość analogicznie. Dostaje jako parametry kolejno:
(x, ptr) i następnie przy pomocy C wywołuje **put_user_check** , które sprawdza przechowywany na *ptr* adres i wywołuje **put_user_size** które w zależności od przekazanego parametru rozmiaru wskaźnika przekazuje odpowiednie parametry do makra **__put_user_asm** , w którym wykonuje się instrukcje analogicznie do **__get_user** .

Funkcje bez dwóch podkreślników na końcu nie dokonują wcześniejszego sprawdzenia poprawności adresu - korzysta się z tej opcji gdy jądro musi wielokrotnie korzystać z tego samego obszaru w przestrzeni adresowej procesu. Lepiej jest wtedy sprawdzić te adresy przy pierwszym wywołaniu, a potem już używać tego adresu bez konieczności sprawdzania za każdym razem, co naturalnie wydłuża czas trwania operacji.

3.7 Blokady pętlowe

spinlock.h

Blokady pętlowe są mechanizmem, który jest używany do synchronizacji na wieloprocesorowych platformach. Są troszkę podobne do semaforów, ale różnią się tym, ze w przy wielu procesorach, często nie opłaca się przełączać kontekstu - gdyż to sporo kosztuje, lepiej jest jest pozwolić procesowi zachować procesor i poczekać na zasób w tzw. ciasnej pętli.

Będziemy korzystać ze struktury spinlock_t, która składa się z pojedynczego pola *lock* . Może ono przyjmować wartości oznaczające włączenie blokady i jej wyłączenie (1 i 0).

Oczywiście będziemy korzystali również z omówionych wcześniej operacji niepodzielnych, gdyż musimy zabezpieczyć się przed jednoczesnym dostępem do naszej struktury.
Makro **SPIN_LOCK_UNLOCKED** inicjuje nam blokadę do wartości 0 - odblokowana.
spin_lock , które pobiera jako parametr adres blokady (tutaj *slp*) pętlowej i generuje kod :

```
1: lock; btsl $0, slp
```

btsl jest niepodzielną instrukcją, która kopiuje do flagi C (przeniesienie) wartość 0*slp, a później go ustawia.

```
  jnc 3f
```

Jeśli się udało (nie ma przeniesienia), to można przeskoczyć do etykiety 3 - czyli kontynuować proces. Jeśli nie, to wchodzimy do etykiety 2 :

```
2: testb $1,slp
  jne 2b
  jmp 1b
3:
```

Sprawdzamy wartość blokady pętlowej, jeśli nie jest wolna, to skok do etykiety 2. I tak do skutku. Wtedy wracamy do etykiety 1. (**NIE** do 3 - musimy pamiętać o zabezpieczeniu przed jednoczesnym dostępem - inny proces z innego procesora mógł w tym czasie zająć blokadę. Trzeba sprawdzić i ewentualnie zająć zmienić.

Makro **spin_unlock** jak można się domyślać zwalnia blokadę i w zasadzie opiera się na poleceniu :

```
lock; btrl $0, slp
```

które czyści bit blokady pętlowej...

4 Podsumowanie

Czas na podsumowanie. Omówię pokrótce to, co przedstawili koledzy, zwracając uwagę na to, w jakich miejscach znajdujemy asemblera w kodzie Linuksa. Następnie rozważę wady i zalety używania wstawek asemblerowych oraz przyczyny, dla których zostały one umieszczone w pewnych miejscach kodu jądra.

4.1 Omówienie

Co zostało powiedziane?
Kolega omówił składnię AT&T asemblera, porównując ją ze znaną nam składnią intelowską. Zaprezentował też własności architektury rodziny Hitachi H8/500, omawiając występujące w niej znaki specjalne, nazwy rejestrów oraz sposoby adresowania. Można zauważyć, że składnia i możliwości asemblera są w dużym stopniu zależne od architektury komputera, na którym pisany jest kod. Następnie dowiedzieliśmy się, w jaki sposób przekazywane są parametry do wywołania systemowego oraz do czego służą rejestry eax, ebx, ecx, edx, esi, edi i ebp. Została też przedstawiona składnia wstawek asemblerowych w C:

```
asm (...), __asm__ (...) i __asm__ __volatile__ (...)
```

Potem zaprezentowane zostały przykłady zastosowania asemblera w kodzie. Kolega przedstawił operacje atomowe i semafory z kodem asemblera, przydatne przy implementacji mechanizmów współbieżności. Takie mechanizmy w jądrze Linuksa należą do miejsc, gdzie kod asemblera wykorzystywany jest w największym stopniu.
Poznaliśmy także ciekawy przykład użycia asemblera, dotyczący korzystania z przestrzeni adresowej procesu: kod asemblera występuje w makrach get_user i put_user. Następnie kolega pokazał zastosowanie wstawek asemblerowych w implementacji mechanizmów pętlowych, używanych do synchronizacji na platformach wieloprocesorowych. Blokady pętlowe korzystają z omówionych wcześniej operacji niepodzielnych.
Poznaliśmy ważne i rozległe zastosowania asemblera w kodzie Linuksa:

- w obsłudze wywołań systemowych: makrodefinicje SAVE_ALL, RESTORE_ALL i GET_CURRENT oraz liczne fragmenty funkcji sys_call
- w mechanizmie przełączania procesów: makro switch_to
- w fragmentach dotyczących operacji czytania i pisania do urządzeń: operacje określone w /usr/include/asm/io.h, makra SLOW_IO_BY_JUMPING, REALLY_SLOW_IO, OUTs, INs, oraz cykliczne INSs i OUTs.

4.2 Dlaczego mamy asemblera w kodzie Linuksa?

4.2.1 Przewaga nad C

Zaletą używania dobrze, optymalnie napisanego kodu asemblera jest zmniejszenie czasu poświęcanego przez komputer na wykonanie programu. Programy poprawnie napisane w asemblerze

powodują, że komputer wykonuje dane operacje w najbardziej efektowny sposób. Dobre kompilatory C generują co prawda instrukcje asemblera zbliżone efektywnością do tych napisanych przez dobrego programistę w asemblerze, ale niekiedy każda oszczędność czasu procesora ma duże znaczenie.

4.2.2 Miejsca wystąpienia

W pewnych kluczowych miejscach kodu Linuksa asembler używany jest w dużym stopniu. Te miejsca zostały omówione przez nas wcześniej. Dlaczego występuje tam kod asemblera? Zauważmy, że są to fragmenty kodu, których czas wykonania jest krytyczny dla czasu wykonania całych operacji: są wykonywane bardzo często. Ich wykonanie powinno być maksymalnie efektywne. Dlatego zapisane są w asemblerze, co daje optymalność.

4.2.3 Omówienie wad

Programy w asemblerze są trudne do napisania i odczytania, a także podatne na błędy. Pisanie dużych programów w asemblerze jest ciężkie i zajmuje dużo czasu. Powstały program nie jest przenośny, a raczej związany z określoną rodziną procesorów, gdyż składnia i możliwości asemblera w dużym stopniu zależą od architektury. Dlatego dużo lepiej używać niezależnego języka, jak C. Kompilatory C, jak wspomniałem, generują instrukcje asemblera zbliżone do optymalnych.

Bardzo mała część jądra Linuksa jest zapisana w asemblerze. Fragmenty te są napisane tylko dla efektywności i są specyficzne dla poszczególnych procesorów. Podsumujmy teraz zalety i wady asemblera:

4.2.4 Zalety asemblera

- Umożliwia dostęp do zależnych od sprzętu rejestrów i I/O
- Umożliwia kontrolowanie zachowania kodu w krytycznych sekcjach, pozwalając na właściwą synchronizację pracy między wątkami.
- Pozwala na optymalizację, poprzez np. tymczasowe łamanie reguł dotyczących alokacji pamięci i innych konwencji.
- Można napisać ręcznie zoptymalizowany dla konkretnego sprzętu kod.
- Pozwala na całkowitą kontrolę nad zachowaniem kodu

4.2.5 Wady asemblera

- Kod asemblera realizujący pewien program jest długi i trudny do napisania
- Jest podatny na błędy

- Błędy mogą być bardzo trudne do znalezienia
- Powstały program nie jest przenośny na inne architektury
- Napisany kod będzie optymalny tylko dla pewnych implementacji konkretnej architektury.
- Pisząc kod w asemblerze koncentrujemy się na szczegółach, trudno wówczas pisać struktury, które najbardziej przyspieszają działanie programu (np. tablice haszujące, drzewa binarne, czy inne wysokopoziomowe struktury)
- Kompilatory zachowują się wystarczająco dobrze dla typowego kodu, generując instrukcje zbliżone do optymalnych.

4.2.6 Podsumowując

W jądrze Linuksa używa się tylko tyle asemblera, ile naprawdę potrzeba. Niewielkie części napisane w asemblerze są tam tylko dla większej szybkości.