Deluppgift 8 Skapa stack till main

Saker som kan vara bra att känna till

När ett program anropar en funktion sker ofta överföring av information till och från funktionen, så kallad parameteröverföring samt returvärde. Parametrar lagras vanligen på en stack. Varje process (eller tråd om processen är flertrådad) har en stack. I Pintos finns stacken i ett minnesutrymme som startar på adressen PHYS_BASE och växer nedåt (mot lägre minnesadress) varefter mer plats behövs. En pekare, stackpekaren, håller reda på var i minnet stacken slutar just nu. I Pintos finns den i (x86) processorregistret ESP. Den pekaren anger alltså toppen av stacken. Returvärden från funktioner placeras i (x86) processorns EAX register. (Kompilatorn ser till att EAX är ledigt när funktionen anropas.) Antag till exempel att vi har funktionen:

```
int exempel(int a, int b, int c)
{
    return a + b + c;
}
```

Antag vidare att följande anrop existerar:

```
int r = exempel(1, 2, 3);
```

Ett funktionsanrop går nu (i korthet) till enligt följande:

1) Stackpekaren räknas ned för att göra plats för sista argumentet (3).
2) Sista argumentet (3) placeras på adressen stackpekaren anger.
3) Stackpekaren räknas ned för att göra plats för näst sista argumentet (2).
4) Näst sista argumentet (2) placeras på adressen stackpekaren anger.
5) Stackpekaren räknas ned för att göra plats för första argumentet (1).
6) Första argumentet (1) placeras på adressen stackpekaren anger.
7) Stackpekaren räknas ned för att göra plats för adressen till nästa instruktion (efter funktionen).
8) Adressen till nästa instruktion (återhoppsadressen) kopieras till stacken.
9) Funktionen exekverar och räknar ut resultatet av a + b + c.
10) Resultatet (som blir 6) placeras i processorns register EAX.
11) Funktionen returnerar genom att processorn fortsätter exekvera från återhoppsadressen.
12) Stackpekaren räknas upp för att komma tillbaka till ursprungsläget (innan punkt 1).

Argumenten utvärderas och kopieras alltså till stacken i omvänd ordning, sista argumentet först, vilket gör att de hamnar i rätt ordning på stacken, första på lägst adress, nästa på följande adress osv.

Denna procedur gäller för alla funktionsanrop. Även start av programmet (anrop av main) och systemanrop har denna uppbyggnad av stacken, men platsen för återhoppsadressen används ibland till annat (systemanropsnumret i fallet med systemanrop). I PintOS startas ett program genom att programmets “startfunktion” `_start` anropas. Startfunktionen läggs till automatiskt av kompilartorn och fungerar som en "wrapper" till `main`. Den ser ut som följer (i PintOS, se `lib/user/entry.c`):

```
_start(int argc, char* argv[])
{
  exit(main(argc, argv));
}
```

Observera två saker:

- Inne i `_start` kommer de båda parametrarna till `_start`, `argc` och `argv`, **alltid** att läsas från stacken för att kopieras som argument till `main`. De kopieras alltså från stacken till ny position på stacken. Detta sker oavsett om `main` använder sina parametrar eller ej. Om stacken är tom när detta sker kommer processen försöka läsa parametrarna "ovanför" (utanför) sin stack och ett pagefault erhålls.
- `_start` kommer *aldrig* att returnera, eftersom systemanropet `exit` alltid körs innan dess. Detta betyder att återhoppsadressen normalt aldrig kommer att användas. *Dock är det bra om returadressen är `NULL` ur felsäkerhetssynpunkt* (ett specialdesignat eller felaktigt program skulle kunna nå dit).

Funktionen `_start` är speciell på så sätt att den startas av operativsystemet. Eftersom `_start` tar emot två argument (samma som `main`) måste operativsystemet se till att dessa är korrekt initierade på stacken. Detta är lite knepigare än det först verkar. Den andra parametern till `main`, ofta kallad `argv`, är en pekare till en sekvens med teckenpekare. Det är alltså pekare i flera led! `argv` måste naturligtvis peka till en giltig sekvens av teckenpekare, som vardera i sin tur måste peka på en giltig sekvens med tecken.

Alla dessa teckenpekare och tecken måste lagras någonstans. De kan teoretiskt sett lagras var som helst i processens minne, men mest praktiskt brukar vara att lagra dem på stacken, eftersom det är det enda minnet vi har att tillgå omedelbart.

Observera att det är förbjudet att läsa 32-bitars tal eller pekare från en adress som inte är jämnt delbar med fyra. Detta ger vid försök i Solaris "bus error". Detta betyder att man måste vara noga med att placera sådana data på adresser jämnt delbara med fyra när de läggs på stacken. Jämt delbara adresser ger även snabbast minnesaccesser på x86-arkitekturen, så det är viktigt där med.

Data att lagra i `argv` hämtas från kommandoraden. Det första ordet på kommandoraden (programmets namn) lagras i `argv[0]`, det andra ordet i `argv[1]`, o.s.v. Orden separeras med blanka tecken. Antalet ord lagras slutligen i `argc`.

Standardfunktionen för att beräkna längden av C-strängar (pekare till konstanta tecken där sista tecknet är noll-teckenet) heter `strlen` och finns i `<string.h>`. Notera att `strlen` inte räknar med det avslutande noll-tecknet, men att en extra byte lagringsutrymme krävs för detta i alla C-strängar. Standardfunktion för att dela upp en sträng i “tokens” heter `strtok_r` och finns i `<string.h>`. ***Notera att originalsträngen förstörs av strtok_r!*** Ett exempel som skriver ut varje delsträng i originalsträngen `orig`:

```
for (token = strtok_r (orig, " ", &save_ptr); token != NULL;
     token = strtok_r (NULL, " ", &save_ptr))
  printf ("'%s'\n", token);
```

Uppgift

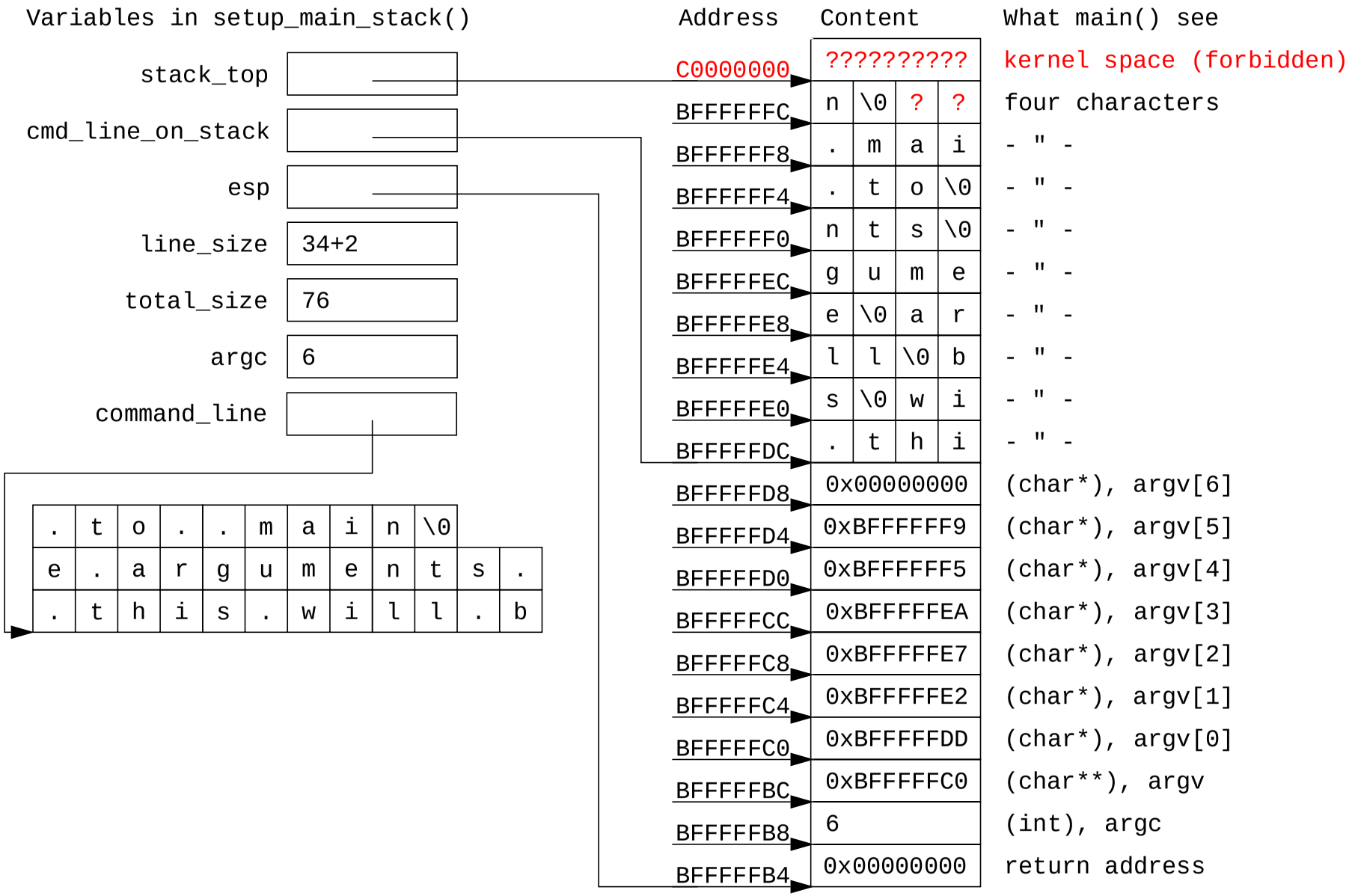
Du skall skriva ett program som läser in en "kommandorad", allokerar minne för en teoretisk stack, och placerar kommandoraden på denna stack så som funktionen `main` förväntar sig. Kopiera filen `setup-argv.c` från kurshemsidan för att få mer ledning. (*Denna kod förs senare över till Pintos.*) Som exempel på hur stacken skall se ut kan du studera följande kommandorad och resultat:

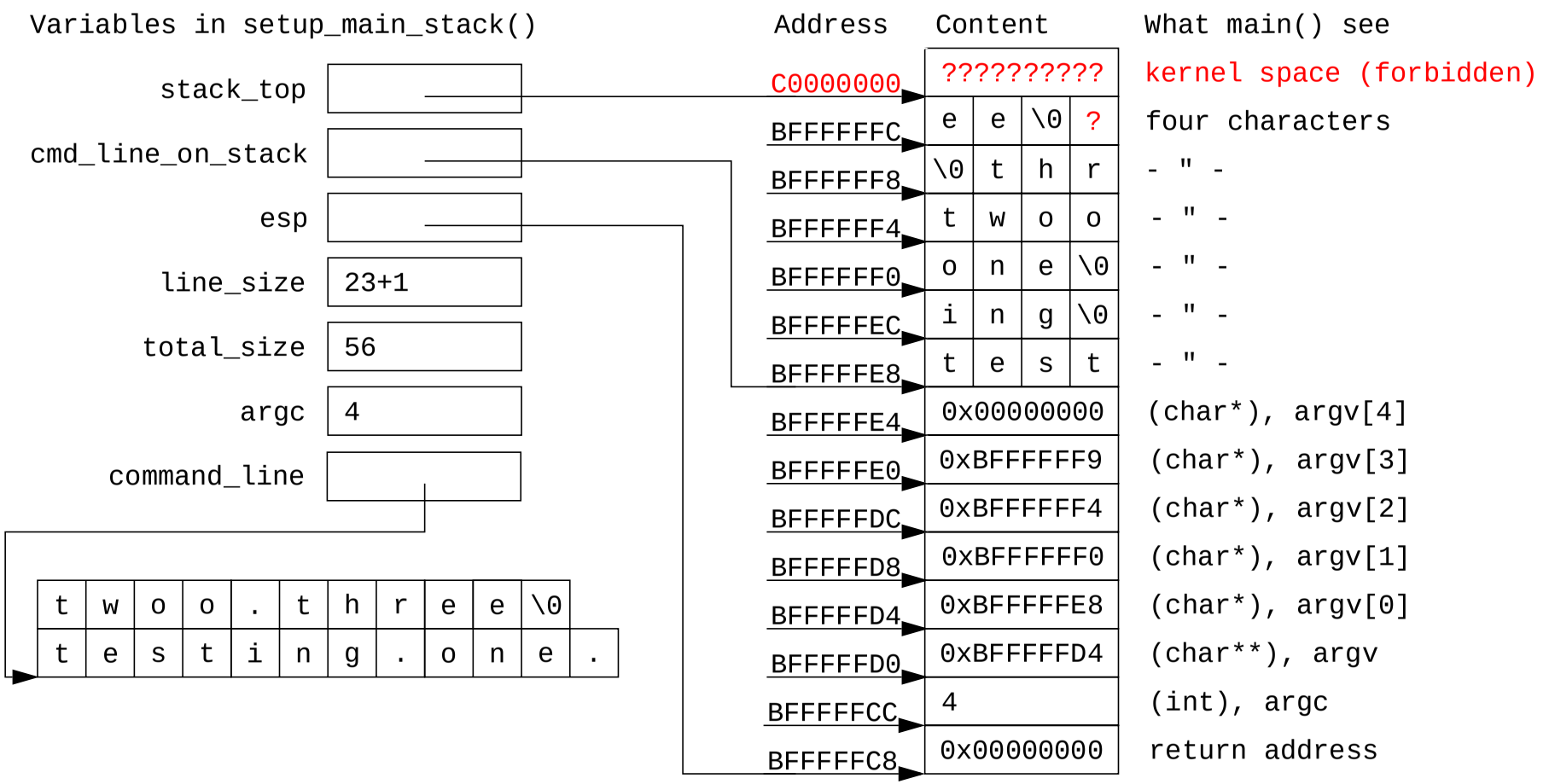
```
Kommandorad: " this will be arguments  to  main " (34 + 1 byte)

Punkt används här för att ange noll-tecknet. '.' => '\0'
Adress och innehåll anges hexadecimalt.

Adress    Innehåll  Datatyp
--------  --------  ---------
C0000000  <PHYS_BASE i pintos kernel space, oläsbar>
BFFFFFFC  n...      char (x4)
BFFFFFF8   mai      char (x4)
BFFFFFF4   to.      char (x4)
BFFFFFF0  nts.      char (x4)
BFFFFFEC  gume      char (x4)
BFFFFFE8  e.ar      char (x4)
BFFFFFE4  ll.b      char (x4)
BFFFFFE0  s.wi      char (x4)
BFFFFFDC   thi      char (x4) <rad STR>
BFFFFFD8  00000000  (char*)
BFFFFFD4  BFFFFFF9  (char*)
BFFFFFD0  BFFFFFF5  (char*)
BFFFFFCC  BFFFFFEA  (char*)
BFFFFFC8  BFFFFFE7  (char*)
BFFFFFC4  BFFFFFE2  (char*)
BFFFFFC0  BFFFFFDD  (char*)  <pekar till 't' på rad STR>
BFFFFFBC  BFFFFFC0  (char**) <pekar till raden ovanför>
BFFFFFB8  6         (int)
BFFFFFB4  00000000  (void (*)(void)) <funktionspekare>
```

OBS! Exemplet bygger på situationen i PintOS precis innan *main* startas. I denna uppgift kommer du se andra adresser. Utskriften av stacken kommer inte heller se ut som ovan, utan endast en byte per rad skrivs ut, dels hexadecimalt (på jämna adresser), och dels som tecken (alla adresser).

Mer exempel ges på kommande sidor. Där är `line_size` angiven med stränglängden plus avrundningen upp till ett tal jämnt delbart med fyra. Står det 21+3 avrundas stränglängden 21 alltså uppåt med tre byte för att använda 24 byte totalt.

Överkurs: Den som vill vara lite mer avancerad kan ta bort dubblerade blanksteg i kommandoraden för att spara ett par byte på stacken. Inte heller det inledande och avslutande blanksteget från kommandoraden behöver lagras på stacken. Att detta ändå lagras beror på att koden blir enklare så.


Variables in setup_main_stack()
stack_top
cmd_line_on_stack
esp
line_size
34+2
total_size
76
argc
6
command_line
. t o . . m a i n \0
e . a r g u m e n t s .
. t h i s . w i l l . b
Address
Content
What main() see
C0000000
??????????
kernel space (forbidden)
BFFFFFFC
n \0 ? ?
four characters
BFFFFFF8
. m a i
- " -
BFFFFFF4
. t o \0
- " -
BFFFFFF0
n t s \0
- " -
BFFFFFEC
g u m e
- " -
BFFFFFE8
e \0 a r
- " -
BFFFFFE4
l l \0 b
- " -
BFFFFFE0
s \0 w i
- " -
BFFFFFDC
. t h i
- " -
BFFFFFD8
0x00000000
(char*), argv[6]
BFFFFFD4
0xBFFFFFF9
(char*), argv[5]
BFFFFFD0
0xBFFFFFF5
(char*), argv[4]
BFFFFFCC
0xBFFFFFEA
(char*), argv[3]
BFFFFFC8
0xBFFFFFE7
(char*), argv[2]
BFFFFFC4
0xBFFFFFE2
(char*), argv[1]
BFFFFFC0
0xBFFFFFDD
(char*), argv[0]
BFFFFFBC
0xBFFFFFC0
(char**), argv
BFFFFFB8
6
(int), argc
BFFFFFB4
0x00000000
return address


Variables in setup_main_stack()
stack_top
cmd_line_on_stack
esp
line_size
23+1
total_size
56
argc
4
command_line
t w o o . t h r e e \0
t e s t i n g . o n e .
Address
Content
What main() see
C0000000
??????????
kernel space (forbidden)
BFFFFFFC
e e \0 ?
four characters
BFFFFFF8
\0 t h r
- " -
BFFFFFF4
t w o o
- " -
BFFFFFF0
o n e \0
- " -
BFFFFFEC
i n g \0
- " -
BFFFFFE8
t e s t
- " -
BFFFFFE4
0x00000000
(char*), argv[4]
BFFFFFE0
0xBFFFFFF9
(char*), argv[3]
BFFFFFDC
0xBFFFFFF4
(char*), argv[2]
BFFFFFD8
0xBFFFFFF0
(char*), argv[1]
BFFFFFD4
0xBFFFFFE8
(char*), argv[0]
BFFFFFD0
0xBFFFFFD4
(char**), argv
BFFFFFCC
4
(int), argc
BFFFFFC8
0x00000000
return address