Deluppgift 4 Använda den dubbellänkade listan i PintOS

OBS! Var försiktig med att klippa och klistra kod. En del tecken konverteras fel. Oftast olika former av citattecken (“ och ” och ‘ och ’). De kan se rätt ut i emcas men kompilatorn förstår ej.

I Pintos finns en länkad lista (src/lib/kernel/list.h). Det är bra att veta hur den används. Den finns tillgänglig att använda utanför Pintos via kurshemsidan. I nedan uppgift skall du använda denna lista till att implementera Erathostenes primtalssåll. Kod för att implementera sållet finns given nedan så du kan koncentrera dig på att skapa den struct som behövs, hur du använder `malloc` och `free` (man -s3c malloc), samt hur listan fungerar. Det är bättre att göra fel i ett litet program utanför Pintos än att göra fel i Pintos.

Saker som kan vara bra att känna till

Listan du skall använda är lite speciell. I grunden är det en dubbellänkad lista. För att undvika specialfall i början används ett “dummy” header-helement, och för att undvika motsvarande specialfall i slutet används ett “dummy” tail-element. Funktioner finns färdiga för att initiera listan, sätta in element först, sist, sorterat eller unikt, iterera genom listan, ta bort element osv. Både headerfilen och implementationsfilen innehåller användbara kommentarer. Det som är speciellt är listelementen som bygger upp listan. De är deklarerade som:

```
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };
```

Som du ser finns alltså en next-pekare och en prev-pekare. Men det finns ingen plats för data. Listan lagrar ingenting. Nu tycker du kanske det är ganska dumt att implementera en lista som inte lagrar någon data. I själva verket är det ett genidrag. Listan är nu helt generell. Istället för att bara kunna lagra ett slags data, av förutbestämnd datatyp, kan den nu lagra vad som helst. Funktionerna som hanterar lista, t.ex. för insättning och borttagning behöver aldrig anpassas till datatypen som lagras, de vet överhuvudtaget inte om att någon data lagras.

För att lagra något i listan gör du “tvärtom” vad som är “normalt”. Istället för att lagra data i varje listelement skall du lagra listelementet tillsammans med din data. Det du sätter in i och får ut från listan är *alltid* listelementet. När du får ut ett listelement **VET** du dock att det ligger tillsammans med din data i minnet eftersom du skapade det så, och kan använda det speciella makrot *list_entry* för att få en pekare till ditt data. Detta fungerar enligt följande exempel.

Exempel på användning av listan med C-teckensträngar

1) Deklaration och initiering som krävs för att skapa och initiera en lista:

```
struct list my_list;
list_init(&my_list);
```

2) För att lagra C-teckensträngar i listan behövs en datatyp som lagrar en C-teckensträng tillsammans med ett listelement. Den datatypen skapas som en struct. Observera namnet på medlemsvariabeln i fetstil. Det variabelnamnet behövs senare, vad du än döpt det till.

```
struct string_elem
{
  char* string;
  struct list_elem elem;
};
```

3) Följande kod allokerar och initierar ett listelement av datatypen från föregående steg.

```
struct string_elem* my_string_elem
  = (struct string_elem*)malloc(sizeof(struct string_elem));
my_string_elem->string= “I rule the Pintos list“
// my_string_elem->elem is initialized when inserted in list
```

4) För att sätta in `my_string_elem` i listan skall du ange adressen till ditt `list_elem`, den medlemsvariabel som angavs i fet stil förut. Du måste även ange adressen till den lista du vill sätta in i.

```
list_push_front(&my_list, &my_string_elem->elem);
```

5) När du hämtar ut data från en lista kommer du att få exekt den adressen som du satte in, dvs en pekare till ditt `list_elem`. Denna kan konverteras till den datatyp som du skapade tidigare (i detta exempel `struct string_elem`) med hjälp av det funktionslika makrot `list_entry`. Makrot `list_entry` tar emot den pekare till `struct list_elem` som du fick ut från listan, och den datatyp som det skall konverteras till (`struct string_elem`). För att utföra adressberäkningarna korrekt behöver makrot även veta hur långt in i din struct som listelementet ligger. Till det behövs återigen namnet på den medlemsvariabel som angavs med fet stil. (Om listelementet ligger t.ex. 8 byte in i din struct, och om listelementet ligger på adressen 864 så startar din struct på 864-8=856.) Kompilatorn ser automatiskt till att beräkningen blir rätt utifrån de argument som skickas till makrot, men givetvis måste **du** veta vilken struct listelementet du får ut från listan är medlemsvariabel i för att kunna ange rätt argument. Följande kodsekvens plockar ut ett listelement och konverterar:

```
struct list_elem* have = NULL;
struct string_elem* want = NULL;
have = list_pop_front(&my_list); /* get and remove from list */
want = list_entry(have, struct string_elem, elem);
/* ... */
free(want); /* ‘want‘ was allocated in step 3 */
```

6) När du tar bort saker ur listan, var noga med att de är bortlänkade ur listan innan du avallokerar minne. Om du tar bort saker inuti en loop som itererar över listan, var noga med att ta tillvara på resultatet från `list_remove`, nästa iteration måste utgå från det värdet eftersom det borttagna inte är giltigt att använda i `list_next`. Följande kod tar bort vissa element ur en lista:

```
for (e = list_begin(&my_list); e != list_end(&my_list); )
{
  struct string_elem *s;
  s = list_entry (e, struct string_elem, elem);

  if (strcmp(s->string, “to remove“) == 0)
  {
    e = list_remove(e); // remove and go to next
    free(s); // needed if ‘s‘ was allocated before insertion
  }
  else
  {
    // e must NOT be removed when you do list_next(e)
    e = list_next(e); // keep and go to next
  }
}
```

7) Listan har en funktion för att sätta in i en sorterad lista. Efterågon data (se tidigare steg) så behövs en funktion som avgör om ett element i listan är mindre än ett annat. Den skall ta in de två listelement som skall jämföras och returnera `true` om det första argumentet är mindre än den andra. En tredje parameter som sällan behövs tillåter att eventuell annan data som behövs i jämförelsen kan skickas med (t.ex. om vi vill sortera en lista med koordinater utifrån varje koordinats avstånd till en fix punkt, där den fixa punkten måste skickas med som extra (`aux`) data). En pekare till jämförelsefunktionen skickas med till insättningsfunktionen genom att helt enkelt ange namnet på jämförelsefunktionenfunktionen. Så här ser jämförelsefunktionen ut för listan i detta exempel (`aux` används ej, så den sätts till `NULL` i anropet):

```
bool less(const struct list_elem* a,
          const struct list_elem* b,
          void* aux)
{
  struct string_elem* sa;
  struct string_elem* sb;
  sa = list_entry (a, struct string_elem, elem);
  sb = list_entry (b, struct string_elem, elem);

  return (strcmp(sa->string, sb->string) < 0);
}

list_insert_ordered(&my_list, &to_insert->elem, less, NULL);
```

Kompilering av flera filer

När ditt program är uppdelat i flera moduler (flera filer) så måste alla implementationsfiler kompileras för att hela programmet skall kunna skapas. Observera att headerfiler *aldrig* skall kompileras. (Headerfiler kommer med vid kompileringen av implementationsfilen eftersom de inkluderas. Om du kompilerar en headerfil av misstag kommer kommer du så småningom att få problem om du inte är noga med att ta bort alla filer av typen *.gch som skapas.)

I denna uppgift behöver du kompilera två filer, listan och huvudprogrammet. Antag att filerna heter list.c och use-list.c, då behövs följande kommando för att förkompilera listan (eftersom du inte skall ändra listan räcker det att kompilera den en gång):

```
gcc -Wall -Wextra -std=c99 -pedantic -g -c list.c
```

Sedan kompileras hela programmet. Här nyttjas den förkompilerade objektfilen `list.o`:

```
gcc -Wall -Wextra -std=c99 -pedantic -g list.o use-list.c
```

Det går även att kompilera allt på en gång (ett kommando), eller att förkompilera även huvudprogrammet. Tänk till själv så kommer du fram till hur. Kompilatorn använder per default “standard” `gnu99` vilket är ISO C99 standarden med några tillägg. I kompileringsraderna ovan instruerar vi kompilatorn att använda strikt ISO C99 standard.

Uppgift

Algoritmen du skall implementera (för att göra något med listan) är given nedan och finns även som fil via kurshemsidan. Fokus i denna uppgift ligger på att öva listhantering, pekarhantering och minneshantering. Ersätt kommentarerna nedan med korrekt listkod:

```
/* TODO: skapa och initera en lista */

for (i = 2; i < N; ++i)
{
  /* TODO: sätt in talet ‘i’ sorterat listan */
}

for (i = 2; i < N; ++i)
{
  for (j = i*2; j < N; j += i)
  {
    struct list_elem *e;
    /* TODO: Stega igenom listan och ta bort varje element
       som är jämt delbart med j. Glöm inte avallokera minne.
     */
  }
}
/* TODO: skriv ut alla tal i listan */
/* TODO: töm listan och avallokera alla element */
```