TD 6 : listes chaînées

Programmation en C (LC4)

Semaine du 5 mars 2007

1 Listes simplement chaînées

▶ **Exercice 1**

```
sliste_t *liste_vide(void) {
    return NULL;
}

int est_vide(sliste_t *liste) {
    return (liste == liste_vide());
}
```

▶ **Exercice 2**

```
#include <stdio.h>

void affiche_liste_iter(sliste_t *liste) {
    while (!est_vide(liste)) {
        printf("%u ", liste->n);
        liste = liste->suivante;
    }
    printf("\n");
}

void affiche_liste_rec(sliste_t *liste) {
    if (!est_vide(liste)) {
        printf("%u ", liste->n);
        affiche_liste_rec(liste->suivante);
    }
    printf("\n");
}
```

▶ **Exercice 3**

```
#include <assert.h>
#include <stdlib.h>

sliste_t *ajoute(unsigned int n, sliste_t *liste) {
    sliste_t *nouvelle_liste = malloc(sizeof(sliste_t));
    nouvelle_liste->n = n;
    nouvelle_liste->suivante = liste;
    return nouvelle_liste;
}

sliste_t *enleve_tete(sliste_t *liste) {
    assert(!est_vide(liste)); /* met fin au programme si liste n'est pas vide */
    sliste_t *nouvelle_liste = liste->suivante;
```

```
    free(liste);
    return nouvelle_liste;
}
```

► **Exercice 4**

```
void libere_liste(sliste_t *liste) {
    while (!est_vide(liste))
        liste = enleve_tete(liste);
}
```

► **Exercice 5**

```
void enleve_element(unsigned int n, sliste_t *liste) {
    sliste_t *precedent = liste_vide();
    sliste_t *nouvelle_liste = liste;
    while (!est_vide(liste) && liste->n != n) {
        precedent = liste;
        liste = liste->suivant;
    }
    if (!est_vide(liste)) {
        if (est_vide(precedent))
            nouvelle_liste = enleve_tete(liste);
        else
            precedent->suivant = enleve_tete(liste);
    }
    return nouvelle_liste;
}

void enleve_element2(unsigned int n, sliste_t *liste) {
    sliste_t *nouvelle_liste = liste;
    sliste_t **precedent = &nouvelle_liste;
    while (!est_vide(liste) && liste->n != n) {
        precedent = &liste->suivant;
        liste = liste->suivant;
    }
    if (!est_vide(liste))
        *precedent = enleve_tete(list);
    return nouvelle_liste;
}
```

2 Stratégie d'allocation mémoire

► **Exercice 6**

```
zone_t zone(unsigned int adr, unsigned int t) {
    return (((adr & 0xffff) << 16) | (t & 0xffff))
}

unsigned int adr_zone(zone_t zone) {
    return ((zone >> 16) & 0xffff);
}

unsigned int taille_zone(zone_t zone) {
    return (zone & 0xffff);
}
```

► **Exercice 7**

```c
void ajoute_zone_libre(unsigned int adr, unsigned int t, memoire_t *mem) {
    mem->zones_libres = ajoute(zone(adr, t), mem->zones_libres);
}

void ajoute_zone_allouee(unsigned int adr, unsigned int t, memoire_t *mem) {
    mem->zones_allouees = ajoute(zone(adr, t), mem->zones_allouees);
}
```

▶ **Exercice 8**

```c
unsigned int utilise_zone_libre(unsigned int t, memoire_t *mem) {
    unsigned int adr_libre;
    sliste_t *liste = mem->zones_libres;
    while (!est_vide(liste)) {
        if (taille_zone(liste->n) >= t) {
            liste->n -= t;
            adr_libre = adr_zone(liste->n);
            liste->n = zone(adr_libre + t, taille_zone(liste->n));
            return adr_libre;
        }
        liste = liste->suivante;
    }
    return TAILLE_MEMOIRE;
}
```

▶ **Exercice 9**

```c
#include <stdio.h>
#include <stdlib.h>

char *alloue_zone(unsigned int t, memoire_t *mem) {
    unsigned int indice = utilise_zone_libre(t, mem);
    if (indice == TAILLE_MEMOIRE) {
        if (mem->taille_occupation + t <= TAILLE_MEMOIRE) {
            indice = mem->taille_occupation;
            mem->taille_occupation += t;
        } else {
            printf("Plus de mémoire.\n");
            exit(EXIT_FAILURE); /* quitte le programme en indiquant l'échec */
        }
    }
    mem->zones_allouees = ajoute(zone(indice, t), mem->zones_allouees);
    return (mem->memoire + indice);
}
```

▶ **Exercice 10**

```c
#include <stdio.h>
#include <stdlib.h>

void libere_zone(char *ptr, memoire_t *mem) {
    unsigned int indice = ptr - mem->memoire;
    sliste_t *liste = mem->zones_allouees;
    sliste_t *precedent = liste_vide();
    while (!est_vide(liste)) {
        if (adr_zone(liste->n) == indice) {
            ajoute_zone_libre(adr_zone(liste->n), taille_zone(liste->n), mem);
            if (est_vide(precedent))
                mem->zones_allouees = enleve_tete(liste);
```

```
            else
                precedent->suivante = enleve_tete(liste);
            return;
        }
        precedent = liste;
        liste = liste->suivante;
    }
    printf("Demande de libération invalide.\n");
    exit(EXIT_FAILURE);
}
```