Devoir noté – Todo-lists : une petite file de priorités en C Version 1.1 du 5 mai

Jean-Cédric Chappelier Merlin Nimier-David

du 5 mai 2020, 17h00, au 28 mai, 23h59.

I. Introduction et instructions générales

Ce devoir noté consiste à écrire une petite application qui gère des listes de tâches (« *TODO lists* ») avec diverses priorités. Nous vous demandons d'écrire tout votre code dans *un seul* fichier nommé `todolist.c` sur la base d'un fichier fourni, à compléter.

Indications : Si un comportement ou une situation donnée n'est pas définie dans la consigne ci-dessous, vous êtes libre de définir le comportement adéquat. On considérera comme comportement adéquat toute solution qui ne viole pas les contraintes données et qui ne résulte pas en un crash du programme.

Instructions :

1. Cet exercice doit être réalisé **individuellement !** L'échange de code relatif à cet exercice noté est **strictement interdit !** Cela inclut la diffusion de code sur le forum ou sur tout site de partage.

   Le code rendu doit être le résultat de *votre propre production*. Le plagiat de code, de quelque façon que de soit et quelle qu'en soit la source sera considéré comme de la tricherie (et peut même, suivant le cas, être illégal et passible de poursuites pénales).

   En cas de tricherie, vous recevrez la note « NA » pour l'entièreté de la branche et serez de plus dénoncés et punis suivant l'ordonnance sur la discipline.

2. Vous devez donc également garder le code produit pour cet exercice dans un endroit à l'accès strictement personnel.

   Le fichier (source !) `todolist.c` à fournir comme rendu de ce devoir ne devra plus être modifié après la date et heure limite de rendu.

3. Veillez à rendre du code *anonyme* : **pas** de nom, ni de numéro SCIPER, ni aucun identifiant personnel d'aucune sorte !

4. Utilisez le site Moodle du cours pour rendre votre exercice.

   Vous avez jusqu'au jeudi 28 mai, 23:59 (heure du site Moodle du cours faisant foi) pour soumettre votre rendu. **Aucun délai supplémentaire ne sera accordé.**

II. Contexte

En première partie, on vous demande d'implémenter une *file d'attente* (« *queue* » en anglais), qui est une structure de données basée sur le principe du *premier entré, premier sorti* (en anglais FIFO, **F**irst **I**n, **F**irst **O**ut), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés.

Dans la seconde partie de l'exercice, on vous demande d'écrire une **file de priorités** (*priority queue* en anglais), qui permet de gérer une file de tâches avec des priorités différentes. Votre programme permet ainsi d'ajouter des tâches (avec leur priorité) à une file de priorités, et de retourner ces tâches dans le bon ordre de priorité.

Les **priorités** possibles pour les tâches sont des nombres entiers positifs (ou nuls). Lorsque les tâches sont retournées de la file de priorités, elles sortent par ordre de priorité (une valeur de priorité plus grande indique une priorité plus élevée).

Le schéma d'implémentation demandé dans ce sujet pour une file de priorités est de créer `n+1` files ordinaires (tableau dynamique), où `n` est le maximum des priorités, et d'enregistrer les tâches dans la file correspondant à leur priorité. Ensuite, lorsque l'on sort une tâche de la file de priorité, on choisit toujours la tâche qui se trouve dans la file (non vide) de priorité la plus élevée. Si la file de priorité la plus élevée est vide, on peut, au choix d'implémentation, soit la supprimer de la liste des files, soit la garder vide pour une future utilisation (mais, dans tous les cas, on remet la valeur de `n` à jour).

Vous devez utiliser la fonction `main()` fournie dans le fichier `todolist.c` et **ne devez pas modifier le code fourni** (au sens : « changer les prototypes » ; vous êtes bien sûr libre d'ajouter vos propres tests et compléter les parties demandées).

III. Implémentation

Le fichier `todolist.c` distribué contient un certain nombre de fonctions, dont certaines que nous vous demandons de compléter. (Commencez peut être par jeter juste un coup d'oeil au fichier fourni pour avoir une idée.)

III.1 Types

Notez que les fonctions fournies utilisent déjà les types que vous allez définir : veillez donc à ce que ces fonctions compilent et fonctionnent comme prévu, sans les modifier.

Files d'attente

Dans un premier temps, créez dans le fichier `todolist.c` fourni, une structure de données `queue_t` pour représenter une file de pointeurs sur des tâches. Elle nécessite deux structures de données complémentaires:

- un type `task_t` (à définir) décrivant de façon *générique* une tâche/un objet à stocker dans la file ; c'est à vous de trouver comment définir ce type pour avoir des tâche/objet *génériques* (regardez le `main()` fourni) ;
- une structure de données à nommer `queue_node_t`, qui représente un élément de la file, et qui contient un pointeur sur une tâche et le prochain élément de la file ;

  Au niveau des de la propriété des tâches : les files ne font donc que les référencer (pointeur) et **ne** sont donc **pas** propriétaires des tâches. Comme illustration, voir le `main()` fourni.

Pour la file elle-même (type `queue_t`), elle doit contenir un moyen d'accéder à la tête de la file (l'élément `queue_node_t` qui a été déposé depuis le plus longtemps dans la file) et un moyen d'accéder au dernier élément de la file (utile pour ajouter une tâche à la file).

Si on prend l'analogie d'une file d'attente à un guichet, `queue_node_t` est un client qui attend (en portant sur lui (l'adresse de) la « tâche » à accomplir/demander) et `queue_t` représente le guichet, qui peut accéder au premier client et au dernier client (il ne voit pas les autres au milieu).

Définissez également :

- le type `task_print`, pointeur sur une fonction pouvant afficher une tâche (de type `task_t`, donc) ;
- une *macro* `NO_TASK` permettant d'indiquer qu'il n'y a pas de tâche.

Files de priorités

Créez ensuite une structure de données `priority_queue_t` pour représenter une file de priorités suivant les principes expliqués dans l'introduction. Cette structure de données contient au minimum (vous être libre d'ajouter d'autres champs si cela vous paraît utile) :

- `max`: un champ de type `priority_t` (à définir) qui représente la priorité maximale en cours dans la file de priorité; `max` sera toujours au minimum égal à `0` ;

- `queues`: les différentes files ordinaires (comme expliqué dans l'introduction) ; elles seront gérées dynamiquement (tableau dynamique), mais à un moment donné, il y en aura donc `max`+1 d'utilisées (et toujours au moins une de disponible, celle de priorité `0`).

Définissez enfin la *macro* `priority_print()` qui affiche tout simplement une `priority_t`.

III.2 Fonctions fournies

Avant de décrire plus bas le travail à faire, nous présentons ici les fonctions déjà fournies (pouvant être utiles).

Fonction `main()`

La fonction `main()` appelle les diverses fonctions (dont celles que vous aurez définies) afin de les tester. Utilisez donc ce `main()` fourni pour faire tous les tests qui vous semblent utiles (typiquement en regardant les résultats produits dans des cas bien spécifiques que vous aurez jugé pertinent de choisir).

Pensez également à tester (par vous-mêmes) chacune de vos fonctions, progressivement.

Fonctions outils

La fonction outil suivante vous est fournie :

- `print_subqueue()` : elle permet d'afficher une des files d'attente qui compose la file d'attente à priorités.

III.3 Fonctions à implémenter

Files d'attente

Pour pouvoir utiliser `queue_t`, créez les fonctions suivantes :

- `queue_init()`, qui initialise une file d'attente (de type `queue_t`) ;
- `queue_push()`, qui ajoute une tâche dans une file d'attente ;
- `queue_pop()`, qui retourne la tâche en tête de la file donnée, selon le principe premier entré, premier sorti ; `queue_pop()` retire également la tâche de la file ;
- `queue_is_empty()`, qui teste si une file est vide (résultat non nul) ou non vide (résultat `0`) ;
- `queue_print()`, qui affiche les éléments d'une file (suivant le format donné dans les exemples de ce sujet) ;
- `queue_clear()`, qui vide une file d'attente.

Utilisez la première partie de la fonction `test()` fournie pour tester votre code. Voici la sortie que votre programme doit produire pour cette première partie du `test()` :

```
Queue example:
pain, confiture, lessive,
pain <-- pop
confiture <-- pop
lessive <-- pop
---
<empty queue>
```

III.3.1 Files de priorités

Pour pouvoir utiliser `priority_queue_t`, implémentez les fonctions suivantes :

- `pri_queue_init()`, qui initialise une file de priorités ;
- `pri_queue_pop()`, qui retourne la première tâche de priorité la plus grande dans une file de priorités (et la supprime de la file) ;
- `pri_queue_push()`, qui ajoute une tâche, en précisant sa priorité, dans une file de priorités ;
- `pri_queue_is_empty()`, qui teste si la file de priorités est vide (résultat non nul) ou non vide (résultat 0) ; une file de priorités est vide si toutes ses files « de base » sont vides ;
- `pri_queue_print()`, qui affiche les éléments d'une file de priorités (en ordre de priorité décroissant, suivant le format utilisé dans les exemples donnés dans ce sujet) ;
- `pri_queue_clear()`, qui vide une file de priorités.
- `pri_queue_delete()`, qui, vide la file de priorités et, si nécessaire, « fait le ménage » en fin de vie d'une file de priorités.

Utilisez (sans la modifier) la seconde partie de la fonction `test()` fournie pour tester votre code. Voici la sortie que votre programme doit produire pour cette seconde partie du `test()` sur le premier tableau de tâches :

```
Priority queue example:
7 : beurre,
6 : <empty queue>
5 : pain, noix,
4 : confiture, miel,
3 : <empty queue>
2 : lessive, sel,
1 : savon,
0 : <empty queue>
beurre <-- pop (max p = 5)
```

```
pain <-- pop (max p = 5)
noix <-- pop (max p = 4)
confiture <-- pop (max p = 4)
miel <-- pop (max p = 2)
lessive <-- pop (max p = 2)
sel <-- pop (max p = 1)
savon <-- pop (max p = 0)
```

IV. Exemple de déroulement

Si tout se passe bien, le `main()` fourni devrait vous donner le résultat suivant. Cet exemple de déroulement complet vous sera également fourni en ligne sous forme d'un fichier texte.

```
======
Queue example:
pain, confiture, lessive,
pain <-- pop
confiture <-- pop
lessive <-- pop
---
<empty queue>
-----
Priority queue example:
7 : beurre,
6 : <empty queue>
5 : pain, noix,
4 : confiture, miel,
3 : <empty queue>
2 : lessive, sel,
1 : savon,
0 : <empty queue>
beurre <-- pop (max p = 5)
pain <-- pop (max p = 5)
noix <-- pop (max p = 4)
confiture <-- pop (max p = 4)
miel <-- pop (max p = 2)
lessive <-- pop (max p = 2)
sel <-- pop (max p = 1)
savon <-- pop (max p = 0)
---
3 : pain, confiture,
2 : <empty queue>
1 : confiture,
0 : <empty queue>
```

```
---
0 : <empty queue>
---
5 : pain,
4 : <empty queue>
3 : confiture,
2 : <empty queue>
1 : <empty queue>
0 : confiture,
---
======
Queue example:
111, 112, 113,
111 <-- pop
112 <-- pop
113 <-- pop
---
<empty queue>
-----
Priority queue example:
7 : 114,
6 : <empty queue>
5 : 111, 117,
4 : 112, 118,
3 : <empty queue>
2 : 113, 116,
1 : 115,
0 : <empty queue>
114 <-- pop (max p = 5)
111 <-- pop (max p = 5)
117 <-- pop (max p = 4)
112 <-- pop (max p = 4)
118 <-- pop (max p = 2)
113 <-- pop (max p = 2)
116 <-- pop (max p = 1)
115 <-- pop (max p = 0)
---
3 : 111, 112,
2 : <empty queue>
1 : 112,
0 : <empty queue>
---
0 : <empty queue>
---
5 : 111,
4 : <empty queue>
```

```
3 : 112,
2 : <empty queue>
1 : <empty queue>
0 : 112,
---
```

V. Quelques conseils pour terminer

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire.

Tout votre code et toutes vos fonctions doivent être robustes tant que faire se peut.

S'il n'y a pas de raison qu'un argument ou membre d'une struct soit modifiable, alors il est souvent préférable de le marquer `const`.

Pensez à tester correctement chacune des fonctionnalités implémentées à **chaque** étape et **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas.