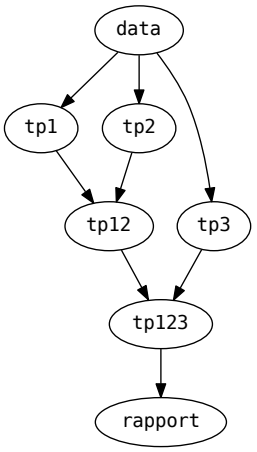
TP à rendre 2

Dans cet exercice, il vous est demandé d'écrire un programme qui utilise la description d'un ensemble de tâches pour les exécuter dans l'ordre prescrit en vérifiant leur bon fonctionnement.

Un graphe de tâche décrit un ensemble de tâches (les sommets du graphe) liées entre elles par des liens de précédence (les arcs du graphe). Le graphe est donc orienté, et doit être acyclique. Les tâches sont des commandes quelconques, et les arcs précisent des contraintes sur l'ordonnancement des tâches : une tâche ne peut s'exécuter que si tous ses prédécesseures ont correctement terminé leur exécution. Dans le fichier de description ci-dessous, on indique pour chaque tâche son nom, les noms de ses prédécesseures, et la commande qui en constitue l'activité. La représentation graphique placée à droite est équivalente, mais elle ne montre pas les commandes associées aux tâches.

```
data:
    test -r notes.data

tp1: data
    $ awk '$2=="tp1"' notes.data > tp1.data

tp2: data
    $ awk '$2=="tp2"' notes.data > tp2.data

tp3: data
    $ awk '$2=="tp3"' notes.data > tp3.data

tp12: tp1 tp2
    $ join -j 1 tp1.data tp2.data > tp12.data

tp123: tp12 tp3
    $ join -j 1 tp12.data tp3.data > tp123.data

rapport: tp123
    $ cut -d ' ' -f 1,3,5,7 tp123.data > rapport.data
```



Un « ordonnanceur » est un programme chargé d'exécuter l'ensemble des tâches du graphe, en respectant les contraintes de précédence. Pour ce faire il doit :

- — créer un nouveau processus chaque fois qu'il veut lancer une tâche, et y exécuter la commande associée à la tâche ; un ordonnanceur ne doit pas créer plus de processus qu'il n'a de tâches à exécuter, et tous les processus exécutant des tâches doivent être des fils directs du processus ordonnanceur ;
- — détecter la fin de chaque tâche et son succès ou son échec, afin de pouvoir déduire les nouvelles tâches prêtes à être lancées (une tâche ne peut être lancée qu'après que tous ses prédécesseures ont été exécutées avec succès) ;
- — s'assurer de ne jamais lancer plus de `nproc` tâches simultanément (le nombre `nproc` est un paramètre de l'ordonnanceur) ;
- — afficher des informations sur chaque tâche, par exemple le numéro de processus après le lancement de la tâche, ou encore le type de fin de tâche (sur `exit()` ou sur un signal) avec les informations afférentes.

Dans l'exemple donné plus haut, au début de l'exécution, seule la tâche `data` est prête à être exécutée : toutes les autres tâches ont des prédécesseures non encore exécutés. Voici par exemple ce qui peut se passer au cours d'une exécution :

- — Si la tâche `data` s'exécute et termine correctement, alors les tâches `tp1`, `tp2` et `tp3` deviennent prêtes.
- — Si on décide d'exécuter par exemple les tâches `tp1` et `tp3` (en même temps éventuellement), et qu'elles se déroulent correctement, aucune nouvelle tâche ne devient prête.
- — La seule possibilité est donc d'exécuter enfin la tâche `tp2`.
- — Après la fin de celle-ci, l'exécution se poursuit, en séquence, par `tp12`, puis `tp123`, puis `rapport`.

Bien sûr, si n'importe laquelle de ces tâches provoque une erreur d'exécution (c'est-à-dire exécute une commande qui ne renvoie pas un code de retour égal à 0), ses successeures ne peuvent pas devenir prêtes. Par exemple, si `tp3` échoue, alors `tp123` ne deviendra jamais prête. L'ordonnanceur s'arrêtera donc, après avoir exécuté `tp12` toutefois. L'exécution va « aussi loin que possible » dans le graphe, mais pas plus loin.

Notez que le principe de l'ordonnanceur est similaire à celui du programme `make`. Mais il y a deux différences principales : premièrement, nous considérons des noms de tâches, *pas* des fichiers, et deuxièmement, nous nous donnons la possibilité d'exécuter plusieurs tâches en même temps (comme le fait `make` avec l'option `-j`).

But du devoir : Vous devez écrire un programme `ordonne.c`, appelé de la façon suivante :

```
./ordonne nproc < fichier.graph
```

où `nproc` est un nombre supérieur ou égal à 1 : c'est le nombre maximum de tâches que l'on peut exécuter en parallèle (à aucun moment il ne doit y avoir plus de `nproc` processus exécutant des tâches simultanément).

Le programme `ordonne` doit exécuter les tâches en respectant les contraintes de précédence, et doit vérifier leur bonne exécution. Il doit aussi, le cas échéant, faire la liste des tâches qui ne peuvent être exécutées parce qu'elles ne seront jamais prêtes. Dans tous les cas, il doit produire sur le canal d'erreur standard des lignes indiquant les événements pertinents de l'exécution : le détail du format à utiliser est décrit dans l'annexe B.

Si l'ordonnanceur parvient à exécuter toutes les tâches sans erreur, alors son code de retour doit être égal à 0. Dans tous les autres cas, son code de retour est 1. En aucun cas l'ordonnanceur ne peut s'arrêter si une tâche lancée est encore en cours d'exécution.

Notez que votre programme doit lire la description d'un graphe de tâches *sur son entrée standard*. Vous n'avez pas besoin de connaître le format exact du fichier de description d'un graphe de tâches. L'archive qui vous est fournie sur Moodle contient les fichiers `taskgraph.c` et `taskgraph.h`, qui vous permettent de lire une telle description et de construire un structure de données utilisable dans votre programme. La structure de données représentant une tâche est décrite dans l'annexe A. L'utilisation des fonctions est illustrée dans le programme `simule.c`. Si vous voulez connaître les détails du format des fichiers, consultez l'annexe C.

Vous trouverez également dans l'archive l'exemple donné plus haut (dans le fichier `notes.graph`, avec un fichier de données appelé `notes.data`). Vous y trouverez aussi un script pour générer aléatoirement de nouveaux graphes (`graph.py`), utilisé dans le *makefile* et surtout dans le script de test.

Bonus (optionnel) : Lorsque votre programme fonctionne correctement, vous pouvez si vous le souhaitez ajouter la fonctionnalité suivante : `ordonne` affiche le temps pris par chaque tâche. Le temps pris par une tâche se décompose en temps *utilisateur* et temps *système*. Ce temps doit être obtenu à l'aide de la primitive `getrusage()`, et affiché à la fin de chaque ligne décrivant la fin d'une tâche.

Programmation : Vous écrirez le programme `ordonne` en langage C en n'utilisant que des primitives systèmes. Vous pouvez utiliser `atoi()` ou `sscanf()` pour convertir l'argument en nombre. Vous pouvez utiliser `fprintf()` et les canaux standard définis dans `stdio.h` pour produire la sortie demandée.

Votre programme doit compiler avec `cc -Wall -Wextra -Werror -Wvla` (utilisez le *Makefile* disponible sur Moodle). Les programmes qui ne compilent pas avec cette commande ne seront pas examinés. Votre programme ne devra pas comporter d'allocation dynamique de mémoire.

Comme d'habitude, un script de test est mis à votre disposition. Pour savoir comment l'utiliser, n'hésitez pas à lire ce script ainsi que le fichier de log généré : ils peuvent vous aider à mettre votre programme au point.

À rendre : Vous devrez rendre sur Moodle un *unique* fichier `ordonne.c` (Moodle sait qui vous êtes, il est inutile d'appeler votre programme `Jean-Claude_Dusse_L2S4_Printemps_2019-2020_ordonne.c`, et il est inutile aussi de rendre un fichier d'un autre nom ou une archive au format-du-jour – il y aura de sévères pénalités sinon).

Ce TP à rendre est individuel. On rappelle que la copie ou le plagiat sont sévèrement sanctionnés.

A Utilisation du code fourni

Nous vous fournissons du code permettant de lire un fichier contenant la description d'un graphe, et d'en construire une représentation en mémoire. Ce code se trouve dans les fichiers `taskgraph.c` et `taskgraph.h`. Ce dernier est documenté. L'archive que vous obtenez sur Moodle contient également un programme d'exemple d'utilisation de ce code : le fichier `simule.c` est une sorte d'ordonnanceur, qui n'exécute rien du tout mais illustre l'utilisation des fonctions qui vous sont fournies : découverte d'une tâche prête, mise à jour à la fin d'une tâche, etc.

Pour exécuter un graphe de tâches, on conserve en mémoire une liste chaînée de tâches, dont chaque élément contient la description d'une tâche. Le tableau suivant liste les attributs d'une tâche, et indique leur valeur initiale pour deux tâches de l'exemple donné plus haut :

Attribut	Tâche `data`	Tâche `tp12`
name	`"data"`	`"tp12"`

cmd	`{ "test",`
	`  "-r",`
	`  "notes.data",`
	`  NULL }`
		`{ "sh",`
		`  "-c",`
		`  "join ... tp12.data",`
		`  NULL }`

nsucc	`3`	`1`

succs	`{ pointeur vers tp1,`
	`  pointeur vers tp2,`
	`  pointeur vers tp3 }`
		`{ pointeur vers tp123 }`

pid	`-1`	`-1`
npred	`0`	`2`

Les pointeurs contenus dans le tableau `succs` sont bien des *pointeurs vers une `struct task`*, et non pas simplement les noms des tâches successeures.

La description d'une tâche contient deux types d'informations :

- les informations *statiques* qui sont purement descriptives : le nom de la tâche (`name`), la commande qui lui est associée (`cmd`), et ses successeurs (`nsucc` et `succs`) ;
- les informations *dynamiques* qui servent à indiquer si la tâche est prête (`npred`), et l'identificateur du processus qui l'exécute (`pid`).

Pendant l'exécution du graphe, les informations statiques ne sont pas modifiées, mais les informations dynamiques varient au cours du temps et de l'exécution des tâches. Par exemple :

- L'attribut `pid` vaut `-1` tant que la tâche n'est pas exécutée. Lorsque la tâche démarre son exécution, `pid` reçoit l'identificateur du processus qui l'exécute ; il garde cette valeur définitivement.
- L'attribut `npred` contient à tout moment le nombre de prédécesseures non encore exécutés correctement. Lorsqu'une tâche est terminée avec succès, votre programme doit décrémenter l'attribut `npred` de tous les successeures de la tâche qui vient de terminer avec succès (ils sont accessibles via l'attribut `succs`).

D'après ces définitions, on peut déduire qu'une tâche est prête lorsque son attribut `npred` vaut `0`, et que son attribut `pid` vaut `-1`. C'est cette condition que recherche la fonction `graph_find_ready()`. Notez que lorsque cette fonction renvoie `NULL`, cela signifie qu'aucune tâche n'est prête, et *pas* que toutes les tâches ont été exécutées.

Votre programme a toute liberté pour modifier les attributs de la partie dynamique. Nous vous recommandons de ne rien modifier dans les attributs de la partie statique (et aux autres attributs non mentionnés ici), et de les utiliser tels qu'ils sont définis.

Le programme `simule.c` illustre l'utilisation du graphe de tâches. Son principe est de chercher une tâche prête, de « faire semblant de l'exécuter » (en tirant au hasard le succès de la tâche), et de mettre à jour le graphe. Et il recommence tant qu'il trouve des tâches prêtes. Quand il s'arrête, il affiche également les tâches qui ne peuvent être exécutées. Son format de sortie est conforme à celui qui vous est demandé (voir annexe B).

B Format de sortie

L'ordonnanceur doit écrire sur son canal d'erreur standard exactement une ligne pour chaque événement important. Ces événements sont :

- lorsqu'une tâche est lancée, la ligne doit être :

  ```
  INIT nom pid
  ```

- lorsqu'une tâche se termine avec succès, la ligne doit être :

  ```
  FINI nom pid
  ```

- lorsqu'une tâche se termine en échouant, il y a deux lignes possibles, selon la nature de l'échec :

  ```
  DEAD nom pid EXIT code
  DEAD nom pid SIGNAL signo
  ```

- à la fin de l'exécution, pour chaque tâche qui ne peut pas être exécutée, l'ordonnanceur doit produire une ligne :

  ```
  STALLED nom
  ```

Dans les descriptions ci-dessus, tout ce qui n'est pas souligné doit apparaître exactement comme indiqué (en lettres majuscules). Pour chaque type de ligne, vous êtes libre d'ajouter ce que vous voulez après les éléments précisés ci-dessus.

Avec l'exemple donné plus haut, les deux listings à gauche et au centre ci-dessous indiquent la sortie du programme avec différents paramètres. Le listing à droite indique la sortie lorsque la commande de la tâche `tp3` échoue.

`./ordonne 1 < notes.graph`

```
INIT data 9813
FINI data 9813
INIT tp1 9814
FINI tp1 9814
INIT tp2 9816
FINI tp2 9816
INIT tp3 9818
FINI tp3 9818
INIT tp12 9820
FINI tp12 9820
INIT tp123 9822
FINI tp123 9822
INIT rapport 9824
FINI rapport 9824
```

`./ordonne 3 < notes.graph`

```
INIT data 10023
FINI data 10023
INIT tp1 10024
INIT tp2 10025
INIT tp3 10026
FINI tp2 10025
FINI tp1 10024
INIT tp12 10030
FINI tp3 10026
FINI tp12 10030
INIT tp123 10032
FINI tp123 10032
INIT rapport 10034
FINI rapport 10034
```

`./ordonne 3 < notes2.graph`

```
INIT data 10193
FINI data 10193
INIT tp1 10194
INIT tp2 10195
INIT tp3 10196
DEAD tp3 10196 EXIT 1
FINI tp1 10194
FINI tp2 10195
INIT tp12 10200
FINI tp12 10200
STALLED tp123
STALLED rapport
```

C Format du fichier de description d'un graphe de tâches

Un fichier de description d'un graphe (extension `.graph` en général) contient une suite de descriptions de tâches. Une description de tâche a la forme suivante :

```
<cible>: <pred_1> ... <pred_n>
    <commande>
```

Ces deux lignes définissent la tâche `<cible>` (un nom). Les noms de tâches `<pred_1>`…`<pred_n>` indiquent les prédécesseures de `<cible>`, et il peut n'y en avoir aucun. Les tâches portant ces noms *doivent* avoir été définies au préalable.

Tous les noms de tâches sont des chaînes de caractères arbitraires. Un nom de tâche ne peut pas contenir d'espace, de tabulation, ou de caractère de fin de ligne.

La ligne `<commande>` peut prendre deux formes distinctes :

— Une commande donnée par la liste des mots qui la constituent. Par exemple :

```
sleep 3
```

est constituée des deux mots `"sleep"` et `"3"`.

— Une commande à destination d'un interpréteur (`sh`), dont le premier mot doit être `$`. Par exemple :

```
$ awk '$2=="tp1"' notes.data > tp1.data
```

est constituée de trois mots : les deux premiers sont `"sh"` et `"-c"`, le troisième est un unique mot contenant tout ce qui suit `$`, c'est-à-dire `"awk ... > tp1.data"`.

Notez que les mots figurant dans la commande ne font l'objet d'aucune interprétation. C'est tout l'objet de la deuxième forme de commande : laissez le *shell* interpréter les tubes, les guillemets, les redirections, etc.

Notez aussi que même si ce format ressemble à celui d'un *makefile*, il est *beaucoup* plus simple : les noms de tâches *ne sont pas* des fichiers ; il ne peut y avoir qu'une seule ligne de commande ; l'ordre des tâches est important ; il n'y a pas de variable, de fonctions, que sais-je encore.