

# TP #4: Sync Lab

Ecole Polytechnique de Montréal

Hiver 2019 — Durée: 2 heures et 10 minutes

## Présentation

Le *Sync Lab* a pour but de vous familiariser avec l'utilisation de mutexes, de sémaphores et de barrières pour synchroniser plusieurs fils d'exécution. **Il est conseillé de lire l'énoncé en entier avant de commencer le TP.**



**Attention:** Ce cours a une politique bien définie en termes de plagiat. L'archive que vous avez téléchargée sur Autolab a été générée pour votre groupe seulement. Il vous est interdit de partager votre code ou le code qui vous a été fourni, que ce soit en ligne (via un dépôt Git public par exemple) ou avec d'autres étudiants. Votre code sera systématiquement vérifié et le plagiat sera sanctionné. Afin d'éliminer tout doute quant à ce qui peut constituer du plagiat et pour connaître les sanctions, veuillez vous référer à la page Moodle du cours.

## Objectifs

A l'issue du TP, vous serez capables de:

- Lancer des fils d'exécutions avec des fonctions à plusieurs paramètres;
- Créer des mutexes, des sémaphores et des barrières de synchronisation;
- Utiliser des mutexes, des sémaphores et des barrières pour synchroniser des tâches en respectant un graphe de dépendances.

## Enoncé

Vous trouverez par la suite les instructions précises pour chacune des questions à résoudre. Pour vous aider à gérer votre temps, nous indiquons pour chaque question avec le symbole 🕒 le temps qu'un élève finissant le TP dans le temps imparti devrait y consacrer. **Ces indications ne sont que des estimations pour que vous puissiez situer votre progression, alors pas de panique si vous dépassez la durée conseillée!**

Question	Contenu	🕒	Barème
1	Lancement de fils d'exécution avec des paramètres	30mn	3.0 pt
2	Synchronisation avec des sémaphores	25mn	3.0 pt
3	Synchronisation avec des sémaphores; gestion des dépendances	40mn	6.0 pt
4	Gestion des défaillances d'un fil d'exécution	35mn	4.0 pt

L'objectif est de simuler l'installation d'un paquet logiciel qui possède plusieurs dépendances: le très célèbre programme `good-1.17`. Vous trouverez le graphe de dépendances complet du programme dans la figure 1. Vous pouvez lire le graphe comme suit: s'il y a une flèche d'un programme à un autre, cela signifie que le premier programme est une dépendance du second programme, et qu'il doit donc être installé avant.

Voici le plan pour installer le paquet `good-1.17`: nous allons lancer simultanément, dans dix fils d'exécution différents, l'installation de chaque paquet (le paquet `good-1.17` ainsi que toutes ses dépendances directes et indirectes). Nous allons ensuite utiliser vos compétences récemment acquises en synchronisation afin de nous assurer que l'ordre d'installation des paquets respecte bien le graphe de la figure 1. Vous serez bien évidemment guidés étape par étape, alors c'est parti!

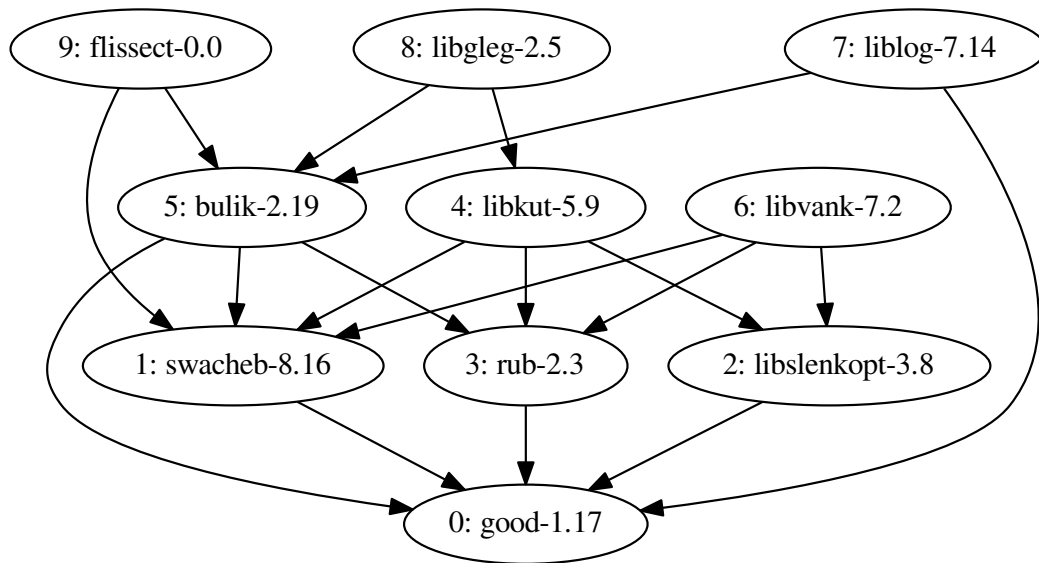


Figure 1: La hiérarchie des paquets logiciels à installer.

Question 1 (Lancement des fils d'exécution) ⌚ environ 30mn

Dans un premier temps, vous allez commencer par créer dix fils d'exécution, dont chacun exécutera la fonction `threadedPackageInstaller` que vous trouverez dans le fichier `installer/libinstaller.h`. Vous n'avez pas à vous soucier maintenant de ce que fait cette fonction, mais notez seulement qu'elle prend deux arguments:

1. Un entier `packageNum`, qui correspond au numéro de paquet (compris entre 0 et 9) indiqué sur la figure 1;
2. Un pointeur vers une structure de type `struct management_data` (vous trouverez la définition dans le fichier `installer/libinstaller.h`). Nous avons initialisé la structure pour vous et vous trouverez le pointeur vers cette structure en paramètre de la fonction `installer` du fichier `installer.c`. Pour le moment, ne vous souciez pas de l'intérêt de cette structure dont vous vous servirez dans les questions suivantes; sachez seulement que tous les fils d'exécution doivent impérativement recevoir un pointeur vers *la même structure*, qui est celle dont nous vous fournissons le pointeur.

🔧 Complétez la fonction `installer` du fichier `installer.c` afin de lancer les dix fils d'exécutions. Au total, la fonction `threadedPackageInstaller` sera donc exécutée dix fois avec un premier paramètre variant de 0 à 9. Vous pourriez avoir besoin de définir une structure pour pouvoir passer correctement les arguments à la fonction `threadedPackageInstaller`; en cas de question, n'hésitez pas à solliciter votre chargé de laboratoire.



**Attention:** La fonction `threadedPackageInstaller` vérifiera qu'elle a bien été appelée depuis dix fils d'exécution différents avec des valeurs différentes de l'entier `packageNum`. Elle utilise notamment une barrière de synchronisation pour s'assurer que les dix appels s'exécutent *en même temps* depuis les différents fils d'exécution. Enfin, la fonction vérifiera que le pointeur vers la structure de type `struct management_data` est correct.

La fonction `threadedPackageInstaller` que nous avons implémentée se charge notamment – en plus d'effectuer des tâches en lien avec la correction automatique – d'appeler la fonction `installPackage` qui est définie dans le fichier `installer.c`, et dont vous écrirez l'implémentation. Cette fonction `installPackage` prend deux arguments – les mêmes que ceux de la fonction `threadedPackageInstaller` – et elle a pour but de:

1. Télécharger le paquet logiciel dont le numéro lui est passé en argument;
2. Installer le paquet logiciel une fois le téléchargement terminé.

Les deux questions qui suivent vous guident pour effectuer chacune des deux tâches.

Question 2 (Téléchargement des paquets) ⌚ environ 25mn

Le but de cette question est de compléter la fonction `installPackage` qui est définie dans le fichier `installer.c` afin de télécharger le paquet logiciel dont le numéro est passé en argument de la fonction. Pour ce faire, vous devrez appeler la fonction `doPackageDownload`, dont vous pouvez voir la signature dans le fichier `installer/libinstaller.h`. Notez bien que le premier argument de cette fonction est une chaîne de caractères; vous devrez donc passer le nom du paquet logiciel en argument, et non son numéro.

La bande passante de notre infrastructure de téléchargement de paquets logiciels étant malheureusement limitée, nous ajoutons une contrainte: la fonction `doPackageDownload` ne pourra être exécutée en parallèle que par 3 fils d'exécutions différents. Cela signifie que vous devrez protéger l'appel à la fonction `doPackageDownload` par un sémaphore partagé par tous les fils d'exécution.

✍️ Complétez les fonctions `initializeManagementData` et `cleanupManagementData` du fichier `installer.c` afin d'initialiser puis de détruire le sémaphore qui vous servira pour protéger l'appel à la fonction de téléchargement. Vous utiliserez pour cela les fonctions de librairie `sem_init`, `sem_destroy`, ainsi que `malloc` pour allouer l'espace mémoire nécessaire pour stocker le sémaphore. Vous stockerez un pointeur vers le sémaphore créé dans le champ `downloadSemaphore` de la structure `md` passée en argument des deux fonctions (vous pouvez voir la définition du type de la structure dans le fichier `installer/libinstaller.h`).

✍️ Complétez ensuite la fonction `installPackage` du fichier `installer.c` comme demandé plus haut. Vous devrez faire un appel à la fonction `doPackageDownload` et protéger cet appel grâce au sémaphore que vous venez de définir, avec les fonctions `sem_wait` et `sem_post`.



**Attention:** Vous devez respecter la contrainte que nous avons définie ci-dessus, à savoir que la fonction `doPackageDownload` ne pourra pas être exécutée simultanément plus de 3 fois. Inversement, nous vous demandons également à ce que la fonction `doPackageDownload` **puisse** être exécutée 3 fois en parallèle dès que possible, sans quoi votre solution ne sera pas acceptée.

Question 3 (Installation des paquets) ⌚ environ 40mn

Dans cette question, vous allez compléter la fonction `installPackage` afin d'installer le paquet logiciel dont le numéro est passé en argument de la fonction. Pour ce faire, vous devrez appeler la fonction `doPackageInstall`, dont vous pouvez voir la signature dans le fichier `installer/libinstaller.h`. Chaque paquet logiciel sera protégé par un sémaphore, dont l'état indiquera s'il est possible d'installer le paquet: si la valeur du sémaphore est à 1, alors le paquet est prêt pour installation.

- vous devrez commencer par initialiser 10 sémaphores différents (un par paquet logiciel) dont la valeur sera initialement à 1 pour les paquets pouvant s'installer immédiatement (ceux qui ne possèdent pas de dépendances), et à 0 pour les autres;
- vous devrez ensuite installer le paquet logiciel passé en argument de la fonction `installPackage` en prenant soin de protéger l'installation par le sémaphore associé au paquet logiciel;
- enfin, une fois l'installation du paquet terminée, il vous faudra déterminer si d'autres paquets sont prêts pour installation, et le cas échéant débloquer les sémaphores correspondants.



**Information:** Nous nous servons dans cette question de sémaphores comme s'il ne s'agissait que de *mutexes*; veuillez donc toujours à ce que vos sémaphores ne prennent que les valeurs 0 ou 1. La raison pour laquelle nous avons privilégié l'utilisation de sémaphores est qu'il n'est pas possible – en théorie – pour un fil d'exécution de déverrouiller un *mutex* qui a été verrouillé par un autre fil d'exécution. Or nous aurons besoin d'exploiter cette possibilité!

✎ Complétez les fonctions `initializeManagementData` et `cleanupManagementData` du fichier `installer.c` afin d'initialiser puis de détruire les sémaphores qui vous serviront pour protéger les appels à la fonction d'installation. Pour chaque paquet logiciel, vous stockerez un pointeur vers le sémaphore créé dans le tableau `canInstallPackage` de la structure `md` passée en argument des deux fonctions – les indices du tableau correspondant aux numéros des paquets logiciels. N'oubliez pas qu'à la fin de l'appel à la fonction `initializeManagementData`, seuls les paquets sans dépendance (et pouvant donc s'installer immédiatement) auront un sémaphore dont la valeur sera à 1.

✎ Complétez ensuite la fonction `installPackage` du fichier `installer.c` comme demandé plus haut. Vous devrez faire un appel à la fonction `doPackageInstall` et protéger cet appel grâce aux sémaphores que vous venez de définir.

✎ Une fois le paquet installé, la fonction `threadedPackageInstaller` fera appel à la fonction `postPackageInstall` du fichier `installer.c`. Complétez cette dernière fonction afin de débloquer correctement les sémaphores des paquets logiciels qui pourraient être prêts pour installation – vous aurez besoin de vous référer à la figure 1. Pour mieux découper votre code, vous pourrez également vouloir utiliser la fonction `wakePackage` du fichier `installer.c`.



**Attention:** Une partie de la note à cette question est consacrée à la correction de votre fonction `postPackageInstall`. Le script d'évaluation procède à un test unitaire exhaustif de votre fonction `postPackageInstall` pour vérifier qu'elle gère correctement tous les ordres possibles d'installation des paquets. **Afin de vous assurer que l'évaluation se déroule correctement, merci de ne pas modifier de variable globale dans la fonction `postPackageInstall`. Votre fonction ne doit agir qu'en fonction des arguments qu'elle reçoit et ne modifier que la structure de données `md` passée en argument.**

#### Question 4 (Gestion des défaillances) environ 35mn


Que se passe-t-il dans le cas où l'un des fils d'exécution défaille pendant le téléchargement ou l'installation d'un paquet? Le risque est que le fil d'exécution ne relâche jamais les verrous qu'il détient, et donc que les paquets dépendant de ce fil d'exécution ne puissent jamais s'installer. Observez ce qu'il se passe quand vous lancez l'installation des paquets mais en faisant volontairement échouer l'installation ou le téléchargement du paquet n°4, par exemple:

```
UNSTABLE=4 ./syncclab 1
```

Nous souhaitons assurer la terminaison du programme même en cas de défaillance d'un fil d'exécution. L'objectif est de faire en sorte qu'en cas de plantage du fil d'exécution associé à un paquet, l'installation de tous les paquets logiciels dépendant directement ou indirectement du paquet défaillant soit annulée. Pour ce faire, vous allez utiliser les fonctions `pthread_cleanup_push` et `pthread_cleanup_pop`. Vous pourrez ainsi définir une fonction qui sera exécutée lorsqu'un fil d'exécution termine inopinément. Cette fonction sera chargée de:

- Faire appel à la fonction `doPackageCleanup` définie dans `installer/libinstaller.h`;
- Terminer à l'aide de la fonction `pthread_cancel` tous les fils d'exécution chargés d'installer des paquets logiciels dépendant du paquet courant. Ces fils d'exécution exécuteront donc à leur tour la fonction désignée par `pthread_cleanup_push`, laquelle exécutera la fonction `doPackageCleanup` et ainsi de suite.

Par effet de cascade, tous les fils d'exécution chargés d'installer soit le paquet défaillant, soit un des paquets dépendant directement ou indirectement de lui, seront terminés après avoir fait appel à la fonction `doPackageCleanup`. En revanche, vous devez faire en sorte que les paquets non affectés par la défaillance soient installés normalement.

 Complétez votre code dans le fichier `installer.c` afin d'obtenir le comportement décrit. Le script d'évaluation testera tous les cas possibles de défaillance et la note dépendra du nombre de cas gérés avec succès.



**Conseil:** Vous aurez besoin pour faire appel à `pthread_cancel` que chaque fil d'exécution ait accès à tous les identifiants (type `pthread_t`) des fils d'exécutions qui sont chargés d'installer les paquets logiciels. Nous vous fournissons à cet effet un tableau `tids` pour stocker ces identifiants dans la structure de type `struct management_data`. Servez-vous en pour enregistrer l'identifiant de chaque fil d'exécution lors des appels à la fonction `pthread_create`.

## Instructions

### Travailler sur le TP

Toutes vos solutions pour ce TP doivent être écrites dans le fichier `installer.c`. **Seul ce fichier sera pris en compte pour évaluer votre travail; il est donc inutile, voire contre-productif, de modifier les autres fichiers que nous vous fournissons!**

### Compiler et exécuter le TP

Nous vous fournissons tous les scripts et librairies pour vous permettre de compiler les sources du TP. Pour compiler le TP initialement et après chacune de vos modifications sur le code source:

```
Console
```

```
$ make
```

lorsque vous vous situez dans le répertoire de base du laboratoire. Si la compilation se déroule sans problème, vous pouvez ensuite exécuter le programme:

```
Console
```

```
$ ./syncclab
```

qui va lancer votre solution pour le TP.

## Evaluer votre travail

Nous vous fournissons les scripts qui vous permettront d'évaluer votre travail autant de fois que vous le souhaitez. Il vous suffit d'exécuter:

```
Console
```

```
$ ./grade.sh
```

pour avoir un rapport détaillé sur votre travail.



**Information:** Les scripts que nous vous fournissons vous donnent une indication mais pas une garantie sur la note finale que vous obtiendrez. Seule l'évaluation par les serveurs d'Autolab sera prise en compte (mais si vous respectez bien les consignes ci-dessus, les deux notes devraient être identiques!).

## Rendre votre travail

Votre travail doit être rendu sur Autolab **avant la fin de cette séance de TP**. Aucune autre forme de remise de votre travail ne sera acceptée. Les retards ou oublis seront sanctionnés comme indiqué sur la page Moodle du cours.

Lorsque vous souhaitez soumettre votre travail – vous pouvez le faire autant de fois que vous le souhaitez pendant la séance –, créez l'archive de remise en effectuant:

```
Console
```

```
$ make handin
```

Cela a pour effet de créer le fichier `handin.tar.gz` que vous devrez remettre sur Autolab. Seul le dernier fichier remis sera pris en compte pour l'évaluation finale.

## Evaluation

Ce TP est noté sur 20 points, répartis comme suit:

- /3.0 pts: Question 1
- /3.0 pts: Question 2

- /6.0 pts: Question 3
- /4.0 pts: Question 4
- /4.0 pts: Clarté du code

Une première note sur 16 points vous est donnée par le script d'auto-évaluation (voir ci-dessus) ainsi que par Autolab. N'hésitez pas à exécuter le script d'auto-évaluation pour connaître le barème détaillé. Les 4 points restants seront évalués par la suite par vos chargés de laboratoire, qui vous feront des commentaires via la plateforme Autolab.

## Ressources

Ce TP vous laisse beaucoup d'autonomie pour programmer votre solution. Le cours constitue une première ressource pour résoudre ce TP. Si vous avez besoin d'informations sur la syntaxe d'une fonction ou d'un programme en particulier, les *manpages* sont une ressource précieuse.

Pour ce TP, les fonctions de type `sem_*` et `pthread_mutex_*`. Vous aurez également besoin des fonctions `pthread_create`, `pthread_join`, `malloc` ainsi que `free`.



**Attention:** Copier puis coller du code tout prêt à partir d'Internet (par exemple depuis Stack Overflow) n'est pas considéré comme du travail original et peut être considéré comme du plagiat. Les *manpages* et les documentations officielles, en revanche, vous aident à apprendre à construire un code par vous-même. Privilégiez cette solution pour améliorer votre apprentissage, et n'hésitez pas à solliciter votre chargé de laboratoire si vous avez une question.