Fiche Ressource n°3 SAE22

Configuration d'un stream audio avec Python et Sounddevice

1. Notion de stream

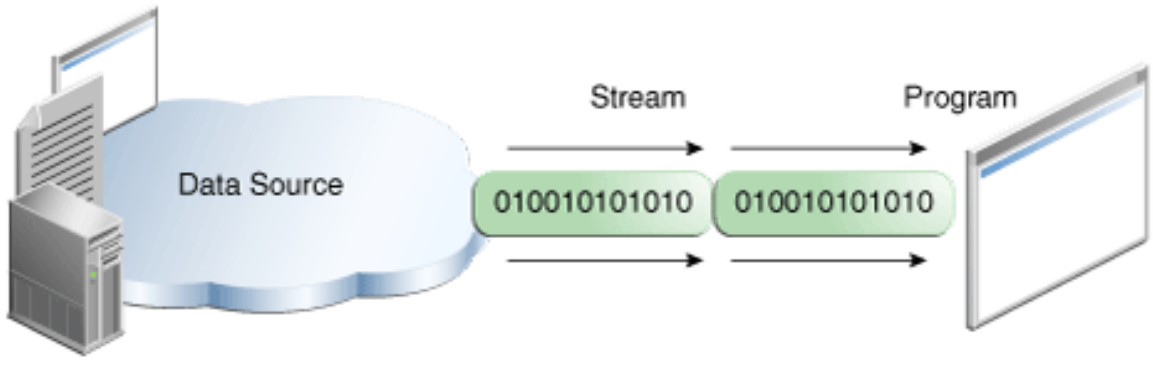
Il existe 2 méthodes pour traiter des données :

- Traiter l'ensemble des données stockée temporairement dans une mémoire dite tampon (buffer) d'un coup. Par exemple on charge un fichier dans une variable stockée en mémoire. On parle de traitement hors ligne ou batch.
- Traiter les données par morceaux « **chunk** », l'ensemble des morceaux transmis formant un flux de données « **stream** ».

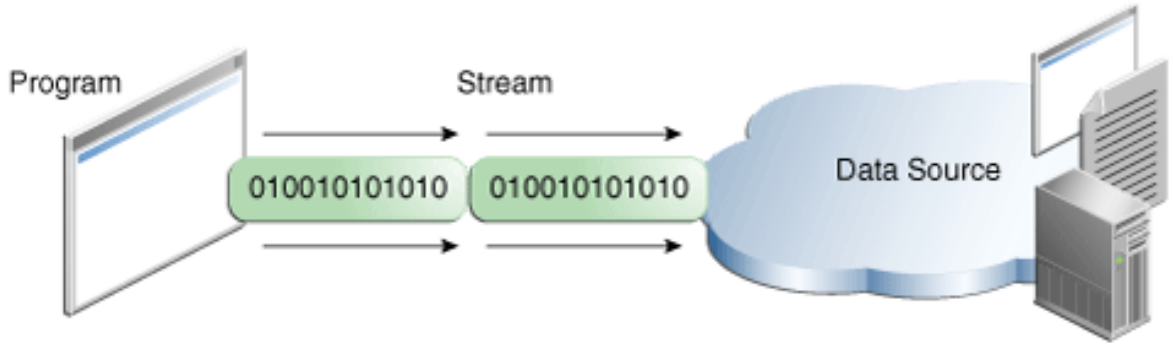
Le traitement des données avec un stream présente 2 avantages :

1. Il permet **d'économiser de la mémoire** puisqu'il n'y a pas à stocker toutes les données mais seulement les morceaux. Il est donc par exemple adapté au traitement de gros fichiers.
2. Il permet de **gagner du temps** puisqu'on n'a pas à attendre la fin de l'envoi des données pour commencer le traitement. Les streams sont donc typiquement utilisés pour le traitement des données en temps réel comme le streaming vidéo qui vous permet de commencer la lecture d'un film sans attendre la fin du téléchargement.

Les streams peuvent être utilisée pour émettre (stream sortant) ou recevoir des données (stream entrant) ou les 2.



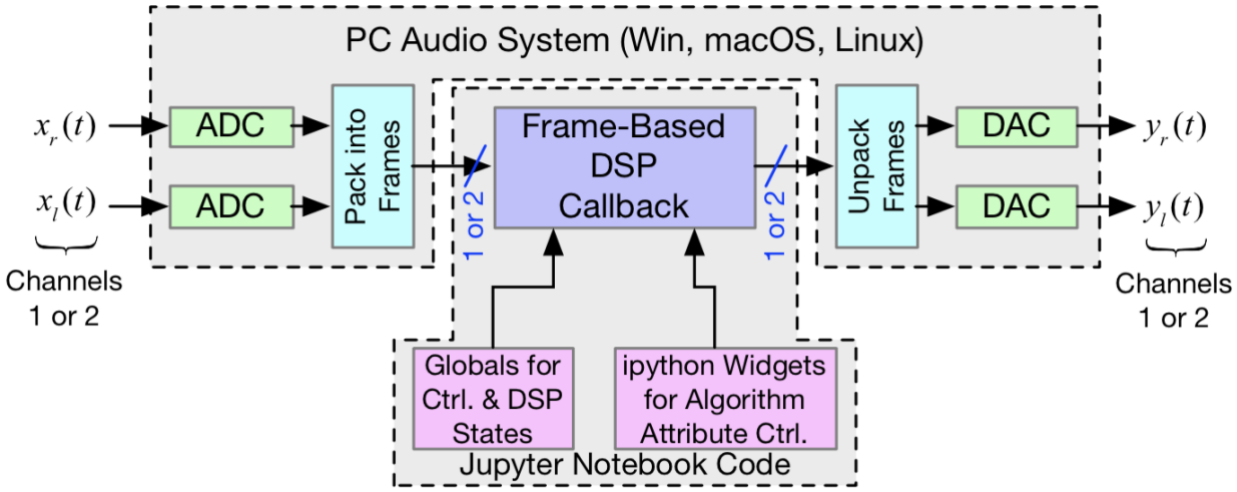
Programme utilisant un stream entrant pour recevoir des données d'un source



Programme utilisant un stream sortant pour générer des données

Dans la SAE on utilisera les streams pour l'émission, et la réception données (signaux audio et signaux modulés AM) via la carte son, soit le micro pour un stream entrant soit les haut-parleurs pour un stream sortant.

L'architecture montrant l'interaction entre la carte son, l'OS et le programme Python que vous mettrez en œuvre est illustré ci-dessous :



Source : Mark Wickert : Real-Time Digital Signal Processing Using pyaudio_helper and the ipywidgets

Pour un **stream entrant**, le signal analogique reçu sur l'entrée analogique du micro de la carte son est d'abord converti en échantillons numériques à la **fréquence d'échantillonnage fe** via son Convertisseur Analogique Numérique CAN ou en anglais ADC « Analog to Digital Converter. On a 1 CAN par voie (gauche ou droite). Les échantillons remplissent une mémoire temporaire ou buffer d'entrée dont la taille est spécifiée par l'utilisateur lors de la création du stream.

Lorsque ce buffer est complètement rempli, le contrôleur de la carte son le notifie au système d'exploitation, qui le notifie au programme d'application afin que celui puisse accéder au bloc audio pour commencer le traitement. En **mode non bloquant,** les échantillons du buffer regroupés dans une **trame** « **frame** » (aussi appelé « **chunk** » : morceau de données) sont envoyés à une fonction spécifique du programme appelé **fonction de callback** dont le nom est spécifié lors de la création du stream. Cette fonction

est en charge du traitement des échantillons et c'est donc dans cette fonction que vous écrirez vos programmes.

Pour un **stream sortant**, quand le buffer de sortie de la carte son est vide, le controleur de la carte son le notifie à l'OS qui l'indique au programme. La fonction de callback est alors en charge de fournir le nouveau morceau « chunk » de données à transmettre au buffer de la carte son. Ces échantillons sont alors transmis sur les haut-parleurs à la fréquence d'échantillonnage indiquée lors de la création du stream via les Convertisseur Numérique Analogique (1 par voie) ou DAC « Digital to Analog Converter ».

2. Premier exemple simple de stream audio avec Sounddevice

La librairie sounddevice permet de créer des streams audio entrant ou sortant via la carte son. Ces données peuvent-être présentées avec une structure de données de type tableau numpy ou sous le format de données brutes « raw ».

On peut créer un stream avec une structure de données numpy avec les fonctions:
- sounddevice.Stream : stream entrant et sortant
- sounddevice.InputStream : stream entrant
- sounddevice.OutputStream : stream sortant

La création d'un stream audio demande de spécifier un certain nombre de paramètres, les plus importants sont :
- **callback** : nom de la fonction de callback (obligatoire si on veut fonctionner en mode non-blocking) quand un chunk/frame est prêt à être reçu (stream entrant) ou émis (stream sortant)
- **samplerate** : la fréquence d'échantillonnage de la carte son
- **channels** : nombre de voies de la carte son : 1 ou 2 pour la stéréo
- **blocksize** : la taille des blocs de données « chunk » en nombre d'échantillons. Par défaut, pour minimiser la latence, le paramètre blocksize = 0, ce qui veut dire que la taille des blocs peut varier si la charge CPU est importante.
- **device** : le numéro (ou les 2 numéros pour un flux entrant/sortant) de la carte son (micro et haut-parleur) à utiliser. Le numéro peut être obtenu avec sounddevice.query_devices().

Ces paramètres ont cependant une valeur par défaut configurée avec l'objet default de sounddevice par exemple default.samplerate pour la fréquence d'échantillonnage. On n'est donc pas obligé de le configurer si on souhaite utiliser la valeur par défaut.

On donne ci-dessous 1 exemple très simple d'utilisation d'un stream entrant qui récupère les données du micro et se contente d'afficher le numéro du chunk et la valeur du premier échantillon :

[89]:
```
nombre_blocs = 0

def print_number(indata, frames, time, status):
    global nombre_blocs
    print(f"Réception du chunk numéro {nombre_blocs} dont le premier élément␣
 ↪vaut {indata[0]}")
    nombre_blocs += 1

stream = sd.InputStream(callback=print_number, blocksize=10000, channels=1)
print('Démarrage du stream')
stream.start()
time.sleep(1)
stream.stop()
stream.close()
print('Fin du stream')
print(f"{nombre_blocs} blocs ont été reçu")
```

```
Démarrage du stream
Réception du chunk numéro 0 dont le premier élément vaut [0.00184224]
Réception du chunk numéro 1 dont le premier élément vaut [0.00621709]
Réception du chunk numéro 2 dont le premier élément vaut [0.00508941]
Fin du stream
3 blocs ont été reçu
```

Dans ce premier exemple on crée un stream avec les caractéristiques suivantes :

- mode non-blocking car on appelle la fonction de callback nommée *print_number*
- Appel de la fonction après réception de blocs de 10000 échantillons sur le micro de la carte son
- Paramètre de la carte son : 1 voie, fréquence d'échantillonnage par défaut

Après création du stream avec `stream = sd.InputStream`, celui-ci est inactif. Il faut donc le démarrer avec `stream.start()` si on veut l'activer. Le stream est lancé dans un thread séparé et pendant ce temps, dans le thread principal, un timer d'1 seconde est lancé avec `time.sleep(1)`. Après ce temps, le stream est arrêté `stream.stop()` puis fermé `stream.close()`. On voit qu'on a reçu 3 blocs ou chunk pendant cet intervalle.

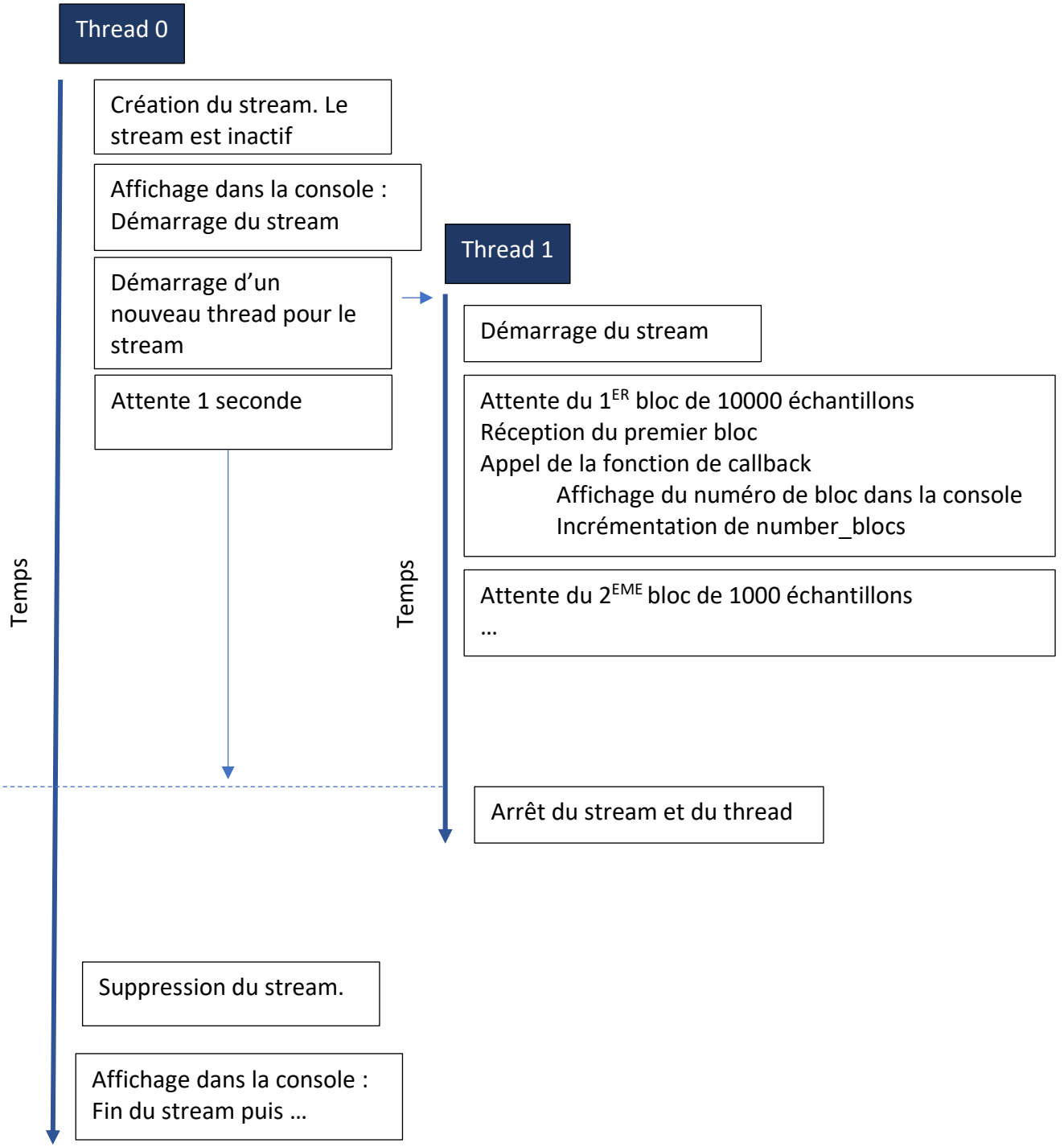
Pendant l'intervalle d'1 seconde, dans le thread 1 gérant le stream, à chaque réception d'un bloc de 10000 échantillons sur le micro de la carte son, le stream va appeler la fonction de callback nommé *print-number*. Cette fonction doit avoir une structure particulière (appelée signature) donnée dans la documentation de sounddevice :

```
callback(indata: ndarray, indata: ndarray, frames: int, time:
CData, status: CallbackFlags) :
```

- *frames* correspond au nombre d'échantillons reçus ou à émettre.

- *indata* et *outdata* sont respectivement les buffers d'entrée et de sortie de la carte son. Il s'agit de tableau numpy bidimensionnel avec une colonne par canal (si on a pris 2 voies pour le stream avec channel = 2) et avec un type de données spécifié par dtype.

```
stream = sd.InputStream(callback=print_number,
blocksize=10000, channels=1)

print('Démarrage du stream')


stream.start()

time.sleep(1)









stream.stop()


stream.close()


print('Fin du stream')
print(f"{nombre_blocs} blocs ont été reçus")
```



Pour un stream entrant, *indata* comprend les données reçues du micro de la carte son. Ici la fonction de callback affiche le numéro de chunk et la valeur du premier échantillon de *indata* :

```
print(f"Réception du chunk numéro {nombre_blocs} dont le premier élément vaut {indata[0]}")
```

Pour un stream sortant, *outdata* doit être rempli par la fonction de callback. Attention on ne peut pas faire une assignation avec outdata = donnees car on doit copier les données dans une autre zone mémoire c'est-à-dire le buffer de la carte son. Il faut donc faire une copie superficielle avec outdata[:] = donnee.

A la place de démarrer, arrêter et stopper manuellement le stream, on peut utiliser le gestionnaire de contexte avec la commande `with`. Le gestionnaire de contexte va effectuer toute ces étapes automatiquement et permet de s'assurer de plus que si une erreur se produit, le stream sera arrêté et fermé correctement (la gestion des erreurs n'est pas prévue dans le programme précédent). Il est donc toujours conseillé d'utiliser le gestionnaire de contexte.

Le même programme avec le gestionnaire de contexte `with` est donné ci-dessous :

[92]:
```
nombre_blocs = 0

def print_number(indata, frames, time, status):
    global nombre_blocs
    print(f"Réception du chunk numéro {nombre_blocs} dont le premier élément␣
 ↪vaut {indata[0]}")
    nombre_blocs += 1

stream = sd.InputStream(callback=print_number, blocksize=10000, channels=1)
print('Démarrage du stream')
with stream :
    time.sleep(1)
print('Fin du stream')
print(f"{nombre_blocs} blocs ont été reçu")
```

```
Démarrage du stream
Réception du chunk numéro 0 dont le premier élément vaut [0.00116605]
Réception du chunk numéro 1 dont le premier élément vaut [0.02694978]
Réception du chunk numéro 2 dont le premier élément vaut [0.01516216]
Fin du stream
3 blocs ont été reçu
```

3. Exemples commentés de diffusion d'un stream audio sur le haut-parleur

La documentation technique de sounddevice donne un certain nombre d'exemples d'utilisation des streams. Une version simplifiée (sans les commandes du module argparse qui permet de gérer les arguments d'une ligne de commande) et commentée de 2 exemples est donnée ci-dessous.

Exemple 1 : réception du signal sur le micro de la carte son et retransmission sur les haut-parleurs

```
fs = 100000

def callback(indata, outdata, frames, time, status):
    if status:
        print(status)
    outdata[:] = indata
stream = sd.Stream(samplerate=fs, callback=callback)
try:
    with stream:
        print('#' * 80)
        print('press Return to quit')
        print('#' * 80)
        input()
except KeyboardInterrupt:
    exit('')
```

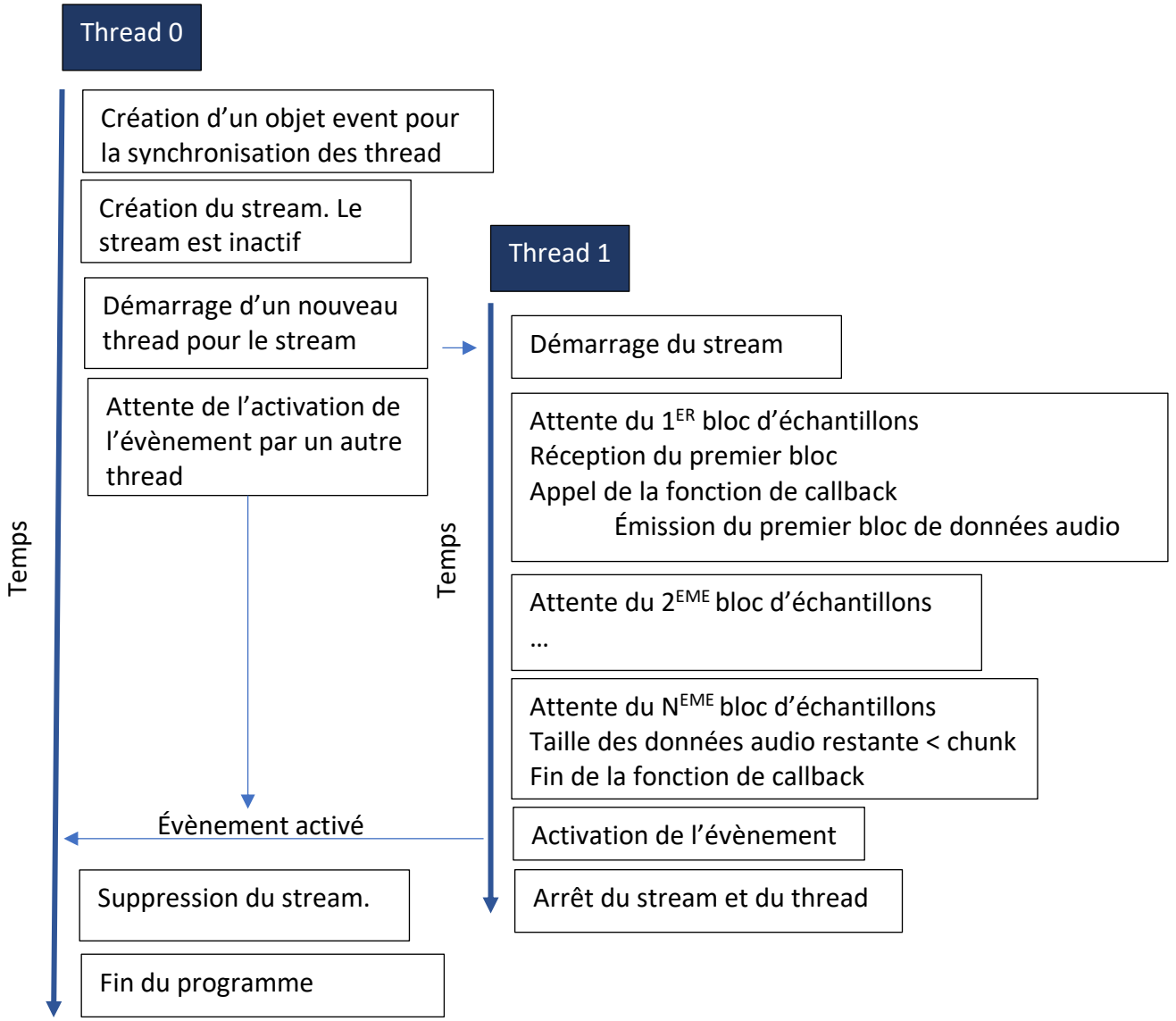
Ce premier exemple assez simple recopie, dans la fonction de callback, les données reçues sur le micro *indata* vers le haut-parleur *outdata* : `outdata[:]= indata`.

Pour arrêter le stream, on affiche dans la console « press Return to quit » et la commande `input()` permet d'attendre que l'utilisateur appuie cette touche Entrée.

Exemple 2 : transmission d'un signal audio issus d'un fichier audio .wav sur les haut-parleurs

```
event = threading.Event()
try:
    data, fs = sf.read('NR4.wav', always_2d=True)
    current_frame = 0

    def callback(outdata, frames, time, status):
        global current_frame
        if status:
            print(status)
        chunksize = min(len(data) - current_frame, frames)
        audio = data[current_frame:current_frame + chunksize]
        outdata[:] = audio
        if chunksize < frames:
            outdata[chunksize:] = 0
            raise sd.CallbackStop()
        current_frame += chunksize

    stream = sd.OutputStream(samplerate=fs, callback=callback,␣
 ↪finished_callback=event.set)

    with stream:
        event.wait()
except KeyboardInterrupt:
    exit('\nInterrupted by user')
```

```
event = threading.event()

stream = sd.OutputStream(samplerate=fs,
callback=callback, finished_callback=event.set)

with stream :


    event.wait()
```



L’objectif de cet exemple est de récupérer le signal audio contenu dans un fichier .wav à l’aide de la librairie soundfile renommée sf, puis de transmettre par morceaux « chunk » ce signal sur la carte son en utilisant un stream audio de sounddevice.

On récupère donc les échantillons du signal audio dans la variable *data* et de la fréquence d'échantillonnage dans la variable fs qui servira de fréquence d’échantillonnage pour la carte son : `data, fs = sf.read('NR4.wav', always_2d=True).`

On utilise une variable globale nommée *current_frame* pour savoir le dernier échantillon du signal audio (et donc de la variable *data*) qui a été transmis lors des appels successifs de la fonction de callback. Cette variable est incrémentée de la taille des données émises *chunksize* à chaque appel de la fonction de callback : `current_frame += chunksize`

Dans la fonction de callback, *frames* correspond à la taille des données attendue par le buffer de la carte son *outdata*. On récupère donc le nombre d’échantillons correspondant dans la variable *data* en commençant à l’échantillon suivant le dernier envoyé lors du précédent appel à la fonction de callback, soit *current_frame* : `audio = data[current_frame:current_frame + chunksize]` qu’on place dans le buffer : `outdata[:] = audio.`

On souhaite arrêter le stream quand le signal audio est terminé c’est-à-dire quand la taille du morceau « chunk » restant à transmettre du signal audio est inférieur à la taille *frames* attendu pour le buffer de sortie de la carte son : `if chunksize < frames:`

Si cette condition est vérifiée, on crée une exception avec la fonction `raised,` cette exception étant `CallbackStop`. L’exception `CallbackStop` continue le stream jusqu'à ce que le buffer généré par la fonction de callback ait été lus, puis l’arrête.

Cependant le buffer *outdata* doit être rempli avec un nombre d’échantillons égal à *frames* . On doit récupérer la taille de ces données : `chunksize = min(len(data) - current_frame, frames)` puis compléter le buffer de la carte son avec des 0 pour avoir un nombre d’échantillons égal à frames : `outdata[chunksize:] = 0.`

L’exception `CallbackStop` n’est pas propagée au thread principal, et le thread principal continue de fonctionner comme si de rien n'était (et n’est donc pas au courant de l’arrêt du stream). On doit donc trouver un moyen pour communiquer avec le thread principal et indiquer la fin du stream.

L'utilisation d'objets event `event = threading.Event()` est un moyen simple de communiquer entre les threads typiquement pour pouvoir synchroniser les opérations entre threads. L’événement crée un indicateur interne que les threads peuvent soit activer avec `set()` soit effacer avec `clear()`. Les autres threads peuvent attendre `wait()` que l’indicateur soit activé pour réaliser une tâche spécifique ce qui synchronise donc les threads.

Ici à la création du thread avec la gestion de contexte `with stream :` on indique au thread principal d’attendre l’activation de l’indicateur par un autre thread :

`event.wait()`. Cet indicateur est activé quand le stream se termine (à cause de l'exception Callbackstop) et le stream se termine par une fonction de callback de fin de stream (`finished_callback`) qui active l'indicateur (`event.set`) :

```
stream = sd.OutputStream(samplerate=fs, device=3, channels=2, callback=callback, finished_callback=event.set)
```

Enfin, le programme est inséré dans une structure de gestion des exceptions `try`, `except` qui permet, si l'utilisateur appuie sur CTRL+C (qui génère l'exception `KeyboardInterrupt`), de stopper le programme avant la fin de la lecture du fichier audio.

Attention : le CTRL+C ne fonctionne pas dans une cellule du notebook, vous utiliserez donc votre IDE préférée pour tester le programme.