TD - Gestion entrées-sorties

Source exploitée : https ://notes.shichao.io/unp/ch6

1 Introduction

Lorsqu'un client gère plus d'un canal d'entrée au même moment, par exemple l'entrée standard `stdin` et un socket, on se retrouve dans une situation où le client est bloqué par la fonction `input` (`fgets` en C), et ne lit donc pas le texte ou la notification de fin reçue par le serveur, provoquant alors une erreur.

On souhaite être notifié lorsqu'une condition d'entrée-sortie est prête : c'est le multiplexage d'entrées-sorties.

Cet outil est utilisé dans les situations suivantes :

— Un client emploie plusieurs descripteurs (un socket et une entrée utilisateur interactive comme l'entrée standard) ;

— Un client est connecté à plusieurs sockets en même temps ;

— Un serveur TCP qui écoute ses clients ;

— Un serveur qui propose simultanément les protocoles TCP et UDP.

On distingue plusieurs modèles d'E/S :

— bloquante ;

— non-bloquante ;

— avec multiplexage ;

— asynchrone.

1.1 Modèle bloquant

Comme expliqué précédemment, le processus est **bloqué** lorsque la fonction `recv` est appelée.

1.2 Modèle non-bloquant

Un socket peut être invoqué en mode non-bloquant. Dans ce cas, le *kernel* reçoit l'ordre de retourner une erreur si une instruction ne peut pas être terminée sans mettre le processus en pause.

L'application effectue ici un *polling* en demandant sans arrêt au *kernel* si des opérations doivent être réalisées. Cela consomme beaucoup de temps CPU, mais on rencontre ce modèle principalement sur des systèmes dédiés à une seule fonction.

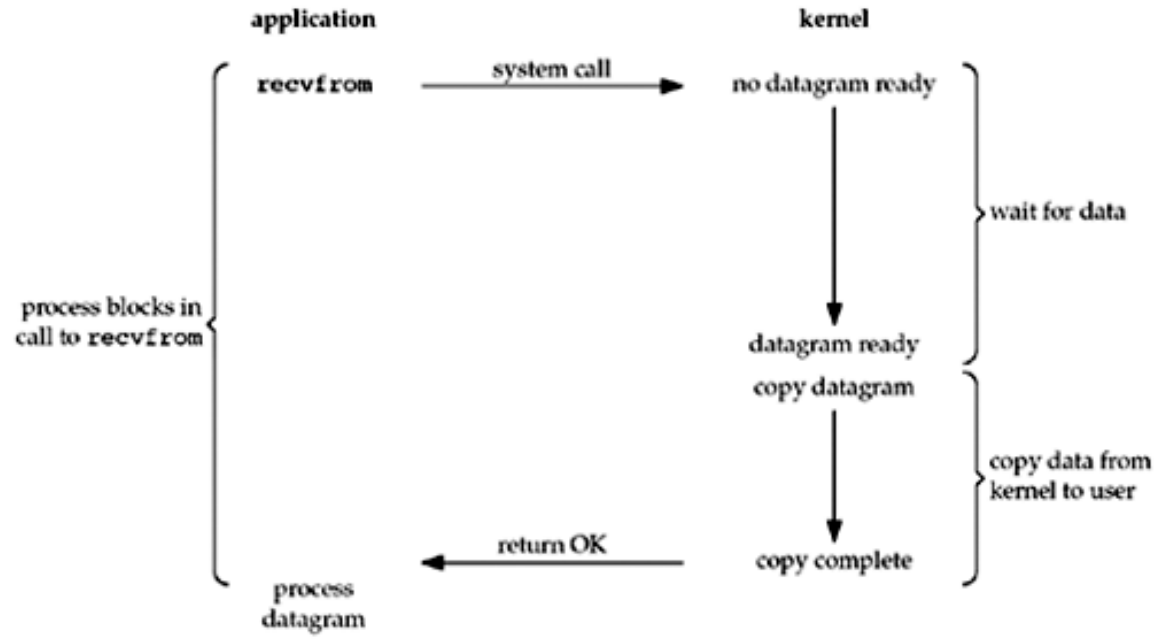

FIGURE 1 – Schéma du modèle bloquant

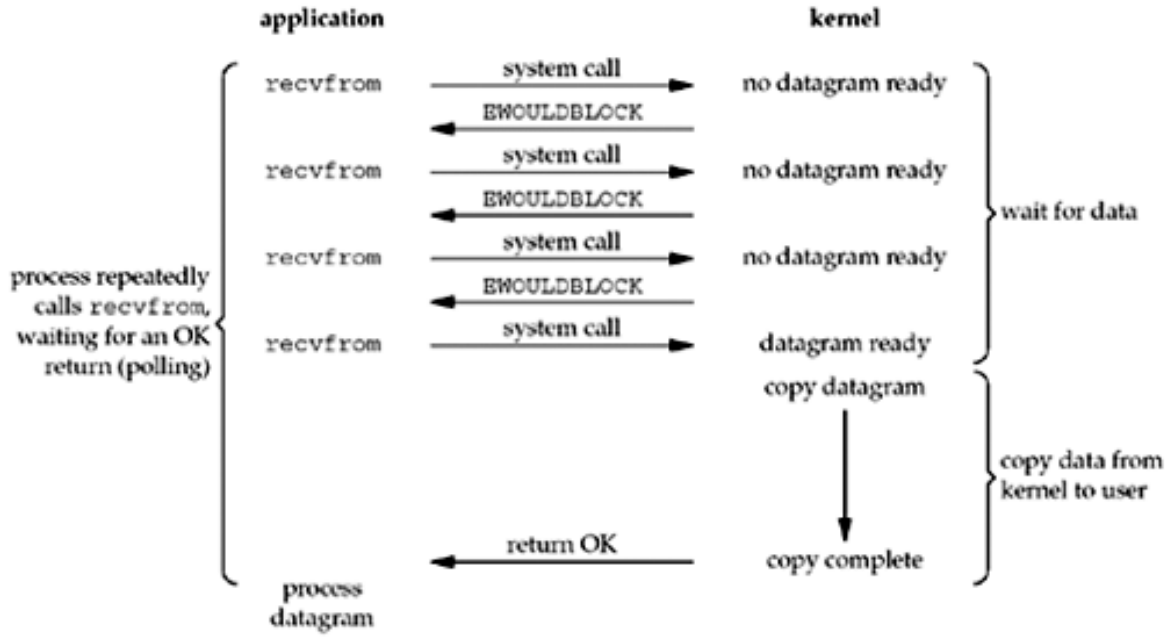

FIGURE 2 – On appelle dans une boucle la fonction `recvfrom` jusqu'à recevoir nos données.

1.3 Multiplexage

La différence notable avec le modèle bloquant est qu'il est possible d'attendre plus d'un descripteur à la fois. Cela est réalisé à partir de la fonction `select`. Cette dernière consiste à demander au *kernel* de signaler le processus qu'un ou plusieurs événements de disponibilité des descripteurs ont été détectés (réception de données, socket prêt à émettre, etc.), ou alors après un certain temps (*timeout*).

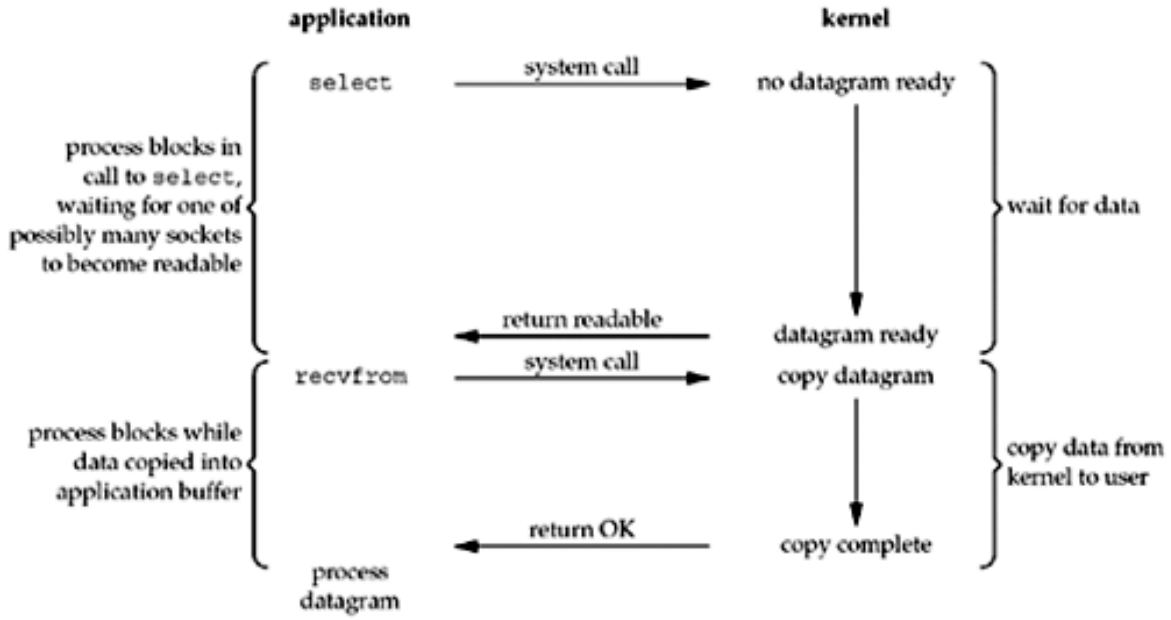

FIGURE 3 – Schéma du modèle multiplexage

Il est également possible d'utiliser plusieurs *threads* pour encapsuler les opérations bloquantes. Ainsi, pour un système de chat, on peut créer deux *threads* : un pour l'entrée clavier, et un pour le socket.

Exercice 1 — Réalisation du chat

Réaliser une application de chat basique. Plusieurs utilisateurs peuvent se connecter simultanément.

Le serveur fonctionnera avec le modèle non-bloquant [1].

1. On structurera légèrement plus nos messages : chaque message transmis contiendra un en-tête qui est un `int` (32 bits, big-endian) qui indique sa taille. Ainsi, le récepteur devra donc appeler une première fois la méthode `recv` en donnant 4 (le nombre d'octets du `int` d'en-tête) en paramètre pour récupérer cette valeur, puis tant que le message n'est pas récupéré en entier, alors on appelle à nouveau `recv`.
   Note : `recv` retourne le nombre d'octets qui ont été reçus.
   Créer deux fonctions `encode_message(mess: str)` et `decode_message(encoded_mess: bytes)`.

   — La première devra retourner le message encodé en `bytes`. La fonction doit ajouter en début de message un `timestamp`, et le message entier aura un en-tête contenant la taille en octets du message.

   — La seconde décode la série de `bytes`, en extrayant l'en-tête contenant le nombre d'octets.

   La librairie `struct` contient des fonctions permettant d'encoder et décoder facilement les types de base. La librairie `datetime` permet de récupérer un timestamp, et de générer automatiquement une chaîne de caractères contenant la date et heure.

1. Un exemple de serveur avec la librairie `selectors` est disponible dans la documentation à l'adresse suivante : https://docs.python.org/3/library/selectors.html

```
# On insère au début du message la taille du message dans un entier
# (I, càd 4 octets) en big indian (>).
structured = struct.pack('>I', len(serialized))
```

2. En partant du code du serveur proposé dans la documentation, apporter des modifications pour que les sockets acceptés soient sauvegardés dans une liste. Lorsqu'un message est réceptionné, la fonction `read` est appelée, et le socket source est donné en paramètre. Le message reçu doit s'afficher dans la console, avec l'adresse du socket, puis il doit être transmis à tous les autres clients connectés. Le message transmis contient dans son en-tête l'heure (timestamp), puis l'adresse de l'émetteur
3. Créer une application client qui permet de se connecter à un tel serveur. L'application sera réalisée à l'aide de threads. Le morceau de code ci-dessous en déclare deux qui vont lancer les fonctions `send_loop` et `receive_loop`.

```
thread_send = threading.Thread(target=send_loop)
thread_send.start()

thread_receive = threading.Thread(target=receive_loop)
thread_receive.start()

thread_send.join()
thread_receive.join()
```

1.4 Asynchrone

Les appels bloquants (qui nécessitent d'attendre) ne le sont plus, et le *kernel* envoie un **signal** défini par l'application lorsque une opération d'entrée-sortie est terminée.

1.4.1 Coroutines

Une coroutine est une série d'instructions, au même titre qu'une *routine* (autre nom de fonction), sauf qu'elle peut être amenée à s'interrompre avant de se terminer. Elle retourne alors des données avant d'être à nouveau appelée. Les générateurs de Python sont typiquement des coroutines : ils contiennent un seul élément qu'ils retournent à chaque appel, puis génèrent le suivant, jusqu'à arriver à la fin.

```
def generateur_base():
    for i in range(4):
        yield i

a = generateur_base()
next(a)
next(a)
next(a)
next(a)
next(a)
```

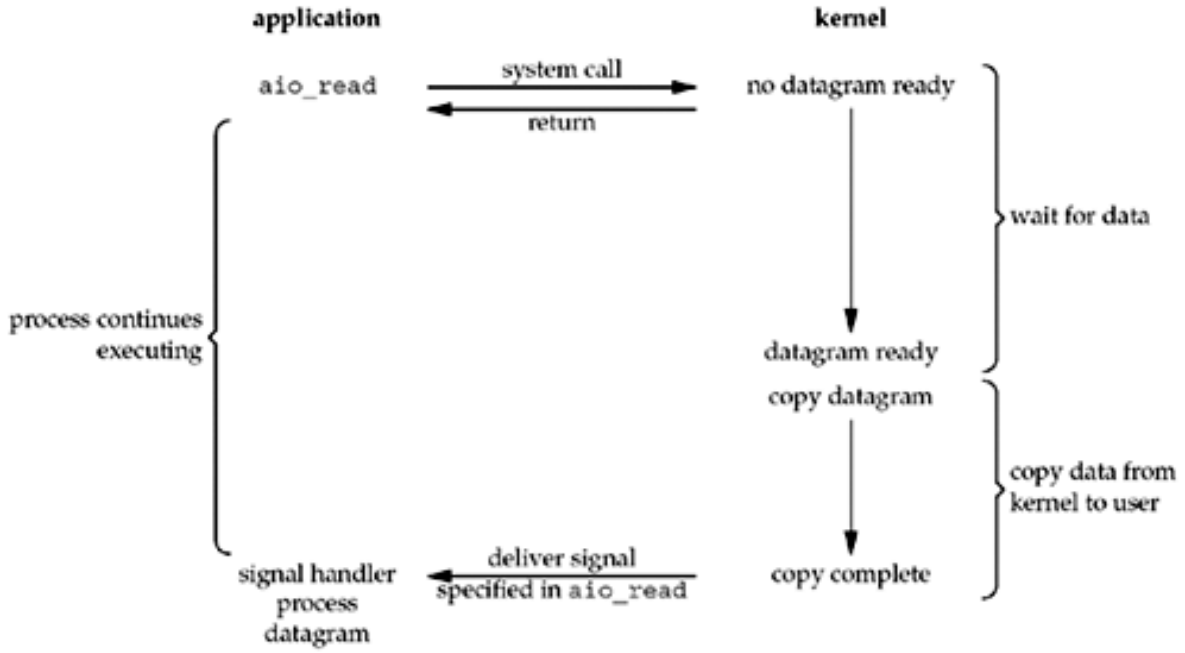

FIGURE 4 – Modèle asynchrone

A chaque appel, la fonction effectue un tour de boucle et retourne la nouvelle valeur par le mot clef `yield`.

Lorsque les éléments sont épuisés, une exception `StopIteration` est envoyée.

Dans Python 3.7+, on déclare des coroutines avec le mot clef `async`.

```
1 import asyncio
2
3 async def main():
4     print('hello')
5     await asyncio.sleep(1)
6     print('world')
7
8 asyncio.run(main())
```

Ce morceau de code affiche *hello*, attend une seconde puis affiche *world*. On remarque qu'il ne suffit pas d'appeler la fonction pour qu'elle s'exécute, mais par l'intermédiaire de la fonction `asyncio.run()`. Celle-ci est le point d'entrée d'un programme asynchrone : elle crée une nouvelle **boucle d'événements** puis la détruit à la fin.

1.4.2 Tâches

Dans Python, une Tâche (`Task`), exécute une coroutine dans une boucle d'événements. Les tâches d'une boucle d'événements s'effectuent de manière concurrente. Le morceau de code ci-dessous s'exécute alors en deux secondes au lieu de trois : les tâches sont créées (presque) simultanément.

```python
async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```