Traitement non-bloquant et asynchrone

INF3173 – Principes des systèmes d'exploitation
Automne 2024

Francis Giraldeau
giraldeau.francis@uqam.ca

Université du Québec à Montréal

Tp3 Thread-pull

ieffect -i data/ -o tmp

marche

-m multi-thread → à implementé

processing.c

↓

process_multi_thread(..)

threadpool create

↳ add_task

↳ join

Semaphor

Sem_t work_busy

Test

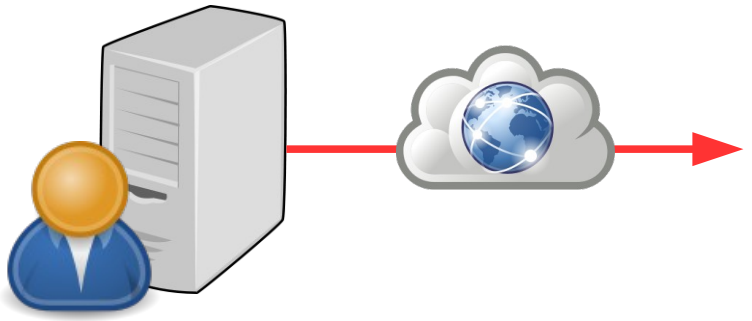
int tid = gettid()

```
int Sem_init (sem_t * sem, int pshared, unsigned int val ){
    pthread_mutex_lock (---)
```

Plan

- Introduction
- Utilisation de fil d’exécution
- Fichiers non-bloquants.
- Appel système multiplexage
- Utilisation de signal
- Application

“Good response time [below 150ms] is the key to user satisfaction” (Tolia et al., 2006)

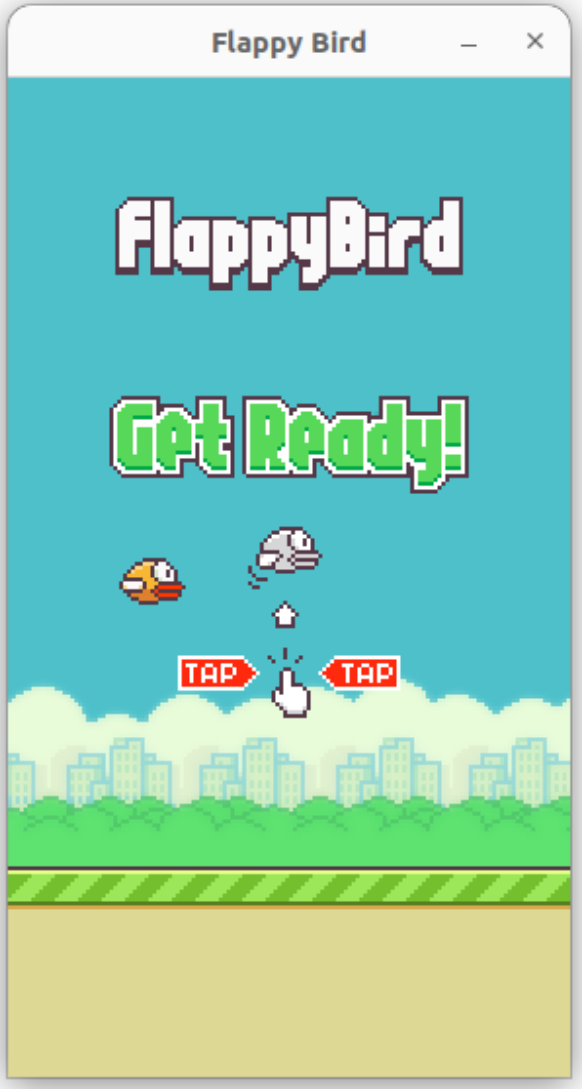


La durée d’un clignement d’oeil est de l’ordre de 100 millisecondes [1]

[1] Source: http://www.ucl.ac.uk/media/library/blinking

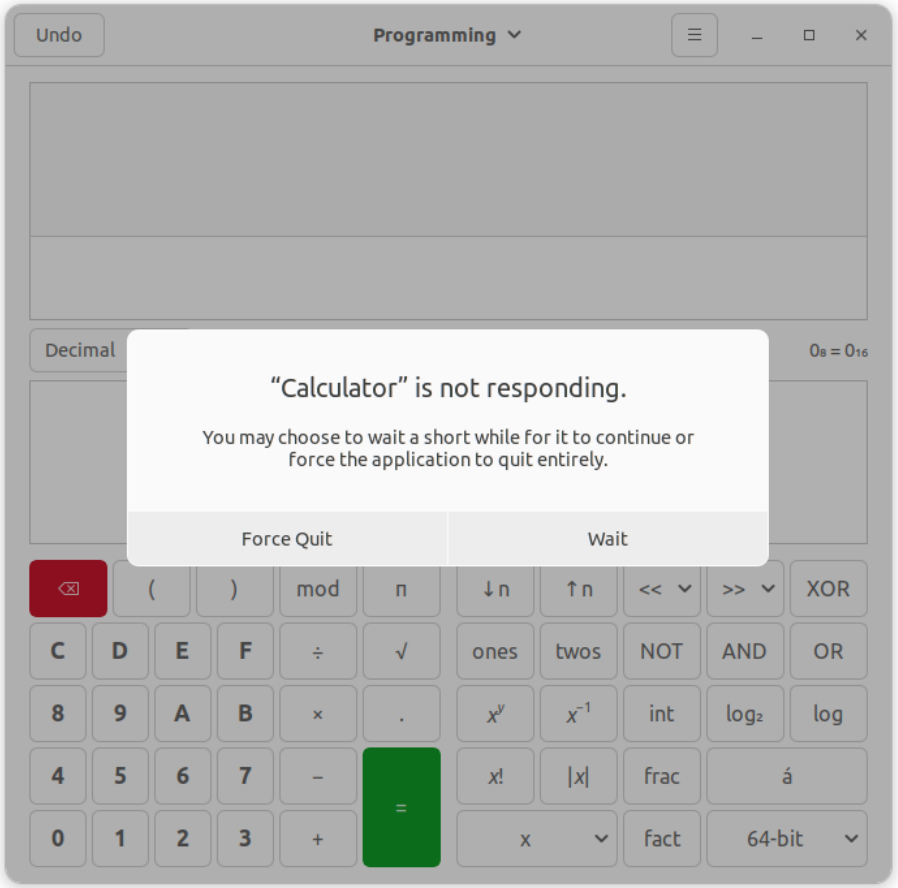
Anatomie d'un jeu

```
FPS = 30 # Fréquence (Frame Per Second)
def FlappyBirdGame(movementInfo):
    # Chargement des images,
    # initialisation, etc.

    # Boucle principale qui se répète
    # tant que le jeu est en marche
    while True:
        # Traitement des événements
        for event in pygame.event.get():
            if event.type == KEYDOWN and ...
                SOUNDS['wing'].play()


        # Mise à jour de la logique du jeu
        # i.e. faire bouger le joueur,
        # détection de collision, etc

        # Dessin à l'écran (rendu)

        pygame.display.update()
        FPSCLOCK.tick(FPS)
```



Source: https://github.com/sourabhv/FlapPyBird

Blocage dans une interface graphique



- La mise à jour de l'interface et la gestion des événement doit se faire continuellement.
- Il ne faut JAMAIS bloquer le fil principal d'une interface graphique.
- Sinon, dégrade l'interactivité et l'utilisateur peut penser que l'application est plantée.

Problématique

- Respecter l'échéance est primordiale pour la fluidité
  - Sinon, le jeu est saccadé et cause frustration
  - Exemple de « soft realtime »
- Temps de rendu < période (frame)
  - 30 FPS : rendu < 32ms
  - 60 FPS : rendu < 16ms
- Le temps de CPU doit être inférieur
  - Incluant tous les processus auxiliaires (i.e. gestion du son)
- **Il faut contrôler les blocages**

Source de blocages

- Faute de page majeur
  - Délai imprévisible : dépend du périphérique!
  - Se produit quand on accède à une adresse
- Réception d’un paquet réseau
  - Le serveur ou la connexion est peut-être lente
  - Par défaut read()/write() pourraient bloquer longtemps sur un socket
- Lecture d’un fichier sur disque avec read()

Stratégies

- Effectuer le travail qui risque de bloquer dans un fil d’exécution séparé
- Utilisation d’entrées-sorties non-bloquant
- Surveillance de plusieurs descripteurs de fichiers simultanément avec délai borné
- En utilisant des signaux

Utilisation d'un fil d'exécution

- Bloquer dans un fil d'exécution n'empêche pas le fil principal de s'exécuter
- Technique classique qui fonctionne bien avec les fichiers ordinaires
- Nécessite de communiquer avec le fil principal (producteur/consommateur)
- L'idéal est de réutiliser les fils d'exécution pour réduire le surcoût
- Ne pas démarrer trop de fils d'exécution, car les fils vont concurrencer pour l'accès au processeur
- Suffisant pour éviter qu'une interface graphique ne bloque pas, mais les transferts se font séquentiellement

Exemple : POSIX aio

- API standard dans la librairie C
- Basé sur des fils d'exécution sous Linux
  - Pourrait changer dans le futur
- Ajout des requêtes dans une file d'attente
  - Structure aiocb (**A**sync **IO C**ontrol **B**lock)
- aio_read() : démarrer une lecture en arrière-plan
- aio_write() : démarrer une écriture en arrière-plan
- aio_error() : déterminer si une requête est terminée
- aio_return() : obtenir le statut d'une requête
- aio_suspend() : attendre qu'il y ait au moins une requête qui se termine

Option O_NONBLOCK

- Modification d'un descripteur avec fcntl()
  - int flags = fcntl(fd, F_GETFL, 0);
  - fcntl(fd, F_SETFL, flags | O_NONBLOCK);
- Rend read() non-bloquant
- Non-prêt : read() ret == -1 && errno == EAGAIN
- On doit réessayer plus tard
  - Mais on peut faire autre chose en attendant!
- En pratique, fonctionne pour socket et pipe
- N'a pas d'effet avec fichiers réguliers

Multiplexage

- Gérer la lecture et l'écriture dans plusieurs fichiers dans un seul fil d'exécution
- Appels systèmes : select(), poll() et epoll()
- Aucune attente ou attente bornée (timeout)
- Tous les descripteurs prêt sont retournés et les utiliser ne bloquera pas
- Exemple : traitement des entrées du clavier et de la souris
- Exemple : téléchargement simultané de plusieurs fichiers, peut se faire dans un seul fil d'exécution
  - Réduction potentielle des changements de contexte coûteux

Multiplexage Linux aio

- API du noyau Linux
- Permet de remplacer POSIX aio, sans utiliser de fil d’exécution
- Le noyau gère la file d’attente
- Disponible depuis Linux 2.5

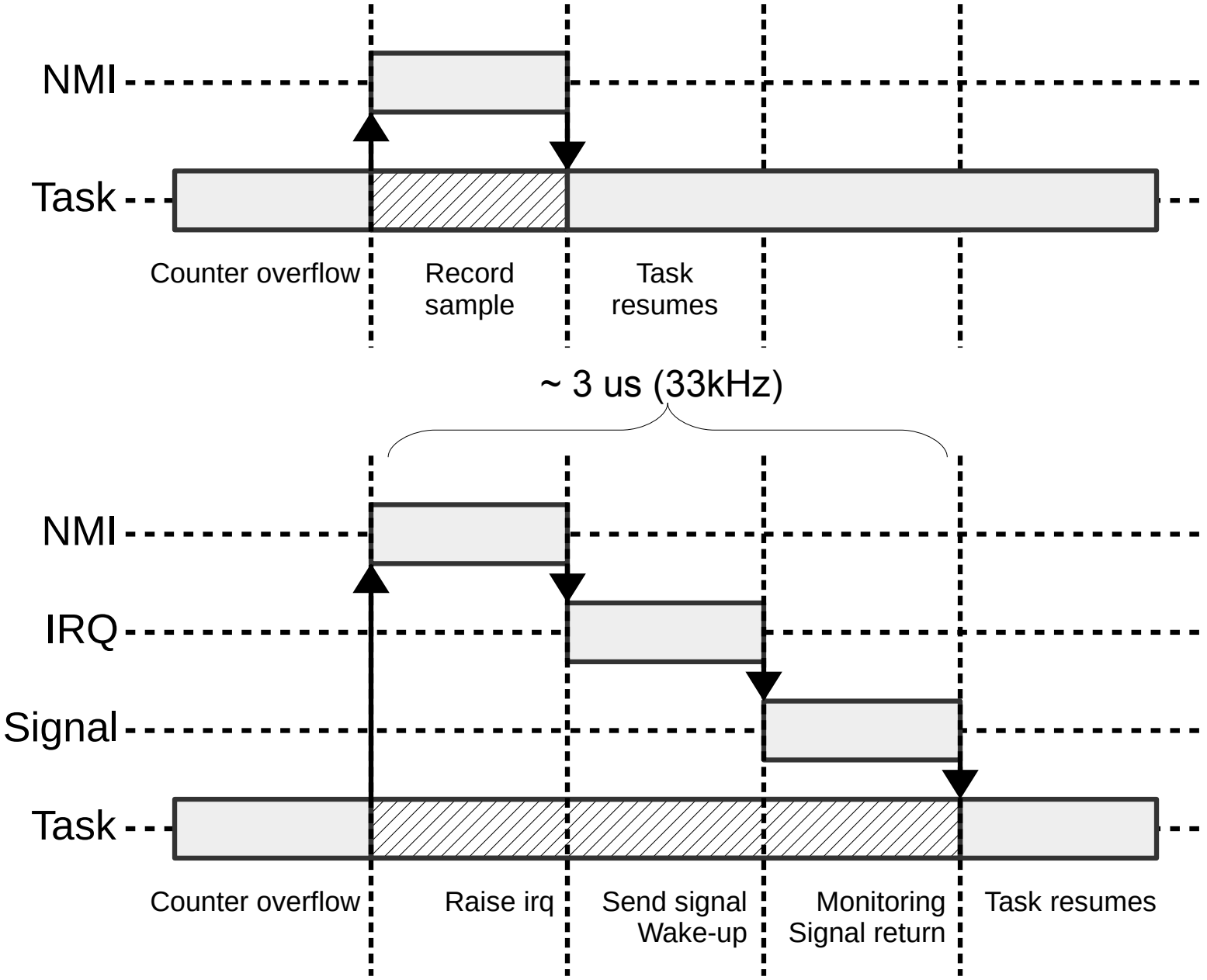
Multiplexage io_uring

- API Linux pour les entrées-sorties haute performance
- Permet d’éviter des copies lorsque ces possible
- Interface récente avec des développements actifs (nécessaite Linux ≥ 5.1)
- Désactivé sur Android et d’autres plaformes par précaution, car plusieurs vulnérabilités ont été trouvées

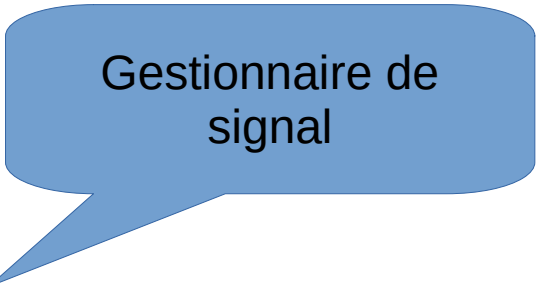
Signal

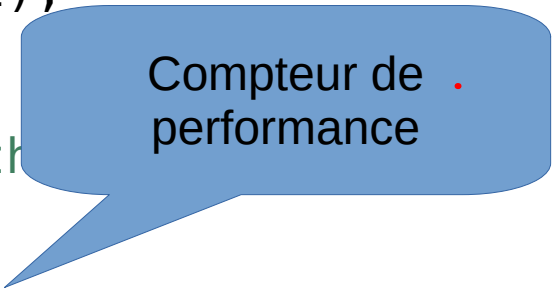
- Définir l’attribut O_ASYNC sur le fichier avec fcntl()
- Le signal SIGIO est émis quand le fichier est prêt pour la lecture/écriture
- Exemple : compteurs de performance

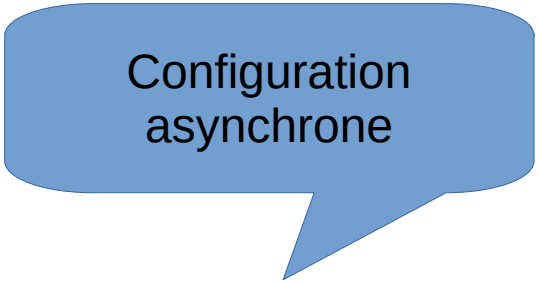
Architecture



```
static int
sampling_do_open(PyObject *obj)
{
    […]
    /* install handler */
    sigact.sa_sigaction = handle_sigio;
    sigact.sa_flags = SA_SIGINFO;
    sigaction(SIGIO, &sigact, &oldsigact);

    /* Open the perf event */
    attr.disabled = 0; /* do not start th
    attr.watermark = 1;
    attr.wakeup_events = 1;
    fd = sys_perf_event_open(&attr, tid, -1, -1, 0);

    /* Configure fasync */
    fown.type = F_OWNER_TID;
    fown.pid = tid;
    fcntl(fd, F_SETOWN_EX, &fown)
    flags = fcntl(ev->fd, F_GETFL);
    fcntl(ev->fd, F_SETFL, flags | FASYNC | O_ASYNC)

    /* enable counter for one shot */
    ioctl(xev->fd, PERF_EVENT_IOC_REFRESH, 1);
    ioctl(xev->fd, PERF_EVENT_IOC_ENABLE, 0);
}
```







Lorsque le signal survient, on prend la pile d'appel pour créer un profile

```
static void
handle_sigio(int signo, siginfo_t *info, void *data)
{
    struct frame tsf[DEPTH_MAX];
    size_t depth = 0;

    depth = __do_traceback(get_top_frame(), tsf, DEPTH_MAX);
    tracepoint(python, traceback, tsf, depth);
    ioctl(fd, PERF_EVENT_IOC_REFRESH, 1);
}
```

Typiquement, on fait simplement mettre un drapeau à 1 pour indiquer qu'un fichier est prêt à être traité

Étude de cas : Python aiohttp

- Librairie permettant de gérer des connexions HTTP de manière asynchrone
- Basé sur epoll
  - Comme select(), mais plus performant
  - epoll_create() : créer un groupe d'attente
  - epoll_ctl() : ajoute, modifie ou enlève des descripteurs à surveiller
  - epoll_wait() : attend qu'un descripteur soit prêt, avec un délai maximum optionnel (timeout)
- Variante epoll_pwait() : permet de désactiver certains signaux temporairement (i.e. SIGINT) pour s'assurer que les requêtes ne soient pas interrompues et se complètent

Python aiohttp serveur

```python
from aiohttp import web
import time

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    time.sleep(1)
    return web.Response(text=text)

app = web.Application()
app.add_routes([web.get('/', handle),
                web.get('/{name}', handle)])

if __name__ == '__main__':
    web.run_app(app)
```

```python
import aiohttp
import asyncio

async def main():

    async with aiohttp.ClientSession() as session:
        async with session.get('http://localhost:8080') as response:

            print("Status:", response.status)
            print("Content-type:", response.headers['content-type'])

            html = await response.text()
            print("Body:", html[:15], "...")

asyncio.run(main())

```

le parent doit être async

Démarrage explicite de la boucle événementielle

Étude de cas: JavaScript async/await

- Basé sur une boucle événementielle
- Attente de tous les descripteurs de fichier, puis exécution des fonctions de rappels liées
- Structure async/await utilise epoll_wait()

```
1  const sleep = (ms) => new Promise(
2      (resolve) => setTimeout(resolve, ms)
3  );
4
5  async function main() {
6    await sleep(3000);
7    console.log("Fin du programme");
8  }
9
10 main();
11
```