Exercices Pratiques et Cas d'Utilisation

1. Pipeline de Traitement Multi-processus

Ce premier exercice illustre l'utilisation combinée des tubes et des processus pour créer un pipeline de traitement de données.

Objectif : Créer un système où trois processus collaborent en chaîne :

1. Le premier processus lit un fichier texte
2. Le deuxième processus convertit le texte en majuscules
3. Le troisième processus compte les occurrences de chaque lettre

Description Détaillée

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>

#define BUFFER_SIZE 4096

void processus_lecteur(int pipe_out, const char* filename) {
    char buffer[BUFFER_SIZE];
    FILE* file = fopen(filename, "r");
    if (!file) {
        perror("Erreur ouverture fichier");
        exit(EXIT_FAILURE);
    }

    size_t n;
    while ((n = fread(buffer, 1, BUFFER_SIZE, file)) > 0) {
        write(pipe_out, buffer, n);
    }

    fclose(file);
    close(pipe_out);
}

void processus_convertisseur(int pipe_in, int pipe_out) {
    char buffer[BUFFER_SIZE];
    ssize_t n;

    while ((n = read(pipe_in, buffer, BUFFER_SIZE)) > 0) {
        for (int i = 0; i < n; i++) {
            buffer[i] = toupper(buffer[i]);
```

```
        }
        write(pipe_out, buffer, n);
    }

    close(pipe_in);
    close(pipe_out);
}

void processus_compteur(int pipe_in) {
    char buffer[BUFFER_SIZE];
    int occurrences[26] = {0};  // Pour les lettres A-Z
    ssize_t n;

    while ((n = read(pipe_in, buffer, BUFFER_SIZE)) > 0) {
        for (int i = 0; i < n; i++) {
            if (isalpha(buffer[i])) {
                occurrences[buffer[i] - 'A']++;
            }
        }
    }

    // Affichage des résultats
    for (int i = 0; i < 26; i++) {
        printf("%c: %d\n", 'A' + i, occurrences[i]);
    }

    close(pipe_in);
}
```

2. Système de Journalisation Multi-processus

Cet exercice illustre l'utilisation des FIFOs et des sémaphores pour créer un système de journalisation centralisé.

Objectif : Permettre à plusieurs processus d'écrire des logs de manière synchronisée dans un fichier central.

Architecture du Système

- Un processus logger central
- Multiples processus clients
- Un FIFO pour la communication

```
struct log_message {
    pid_t pid;           // Identifiant du processus
    time_t timestamp;    // Horodatage
    int level;           // Niveau de log (INFO, WARNING, ERROR)
    char message[256];   // Message
```

```
};

// Code du logger central
void logger_central(const char* fifo_path, const char* log_file) {
    struct log_message msg;
    int fifo_fd = open(fifo_path, O_RDONLY);
    FILE* log_fp = fopen(log_file, "a");

    while (1) {
        ssize_t n = read(fifo_fd, &msg, sizeof(msg));
        if (n == sizeof(msg)) {
            fprintf(log_fp, "[%s] PID-%d: %s\n",
                    level_to_string(msg.level),
                    msg.pid,
                    msg.message);
            fflush(log_fp);
        }
    }
}

// Code client
void log_message(const char* fifo_path, int level, const char* message) {
    struct log_message msg;
    msg.pid = getpid();
    msg.timestamp = time(NULL);
    msg.level = level;
    strncpy(msg.message, message, sizeof(msg.message) - 1);

    int fifo_fd = open(fifo_path, O_WRONLY);
    write(fifo_fd, &msg, sizeof(msg));
    close(fifo_fd);
}
```

3. Gestionnaire de Ressources Partagées

Ce dernier exercice combine mémoire partagée et sémaphores pour gérer un pool de ressources.

Objectif : Implémenter un système où plusieurs processus partagent un ensemble de ressources limitées.

```
struct resource_pool {
    sem_t mutex;          // Protection de la structure
    sem_t resources;      // Compteur de ressources disponibles
    int total;            // Nombre total de ressources
    int available;        // Nombre de ressources disponibles
    int allocated[MAX_PROCESSES];  // Allocation par processus
};
```

```c
void init_resource_pool(struct resource_pool* pool, int count) {
    sem_init(&pool->mutex, 1, 1);
    sem_init(&pool->resources, 1, count);
    pool->total = count;
    pool->available = count;
    memset(pool->allocated, 0, sizeof(pool->allocated));
}

int acquire_resource(struct resource_pool* pool, int count) {
    if (sem_wait(&pool->mutex) == 0) {
        if (pool->available >= count) {
            pool->available -= count;
            pool->allocated[getpid() % MAX_PROCESSES] += count;
            sem_post(&pool->mutex);
            return 0;  // Succès
        }
        sem_post(&pool->mutex);
    }
    return -1;  // Échec
}
```

4. Exercices Supplémentaires

1. Implémentez un système de cache partagé entre processus
2. Créez un pool de workers avec distribution de tâches
3. Développez un système de messagerie asynchrone entre processus