Développement d'un Simulateur de Système de Contrôle de Fabrication avec Robots en C

LOZAHIC LEON - CHU THOMAS

SR70 ARCHITECTURE DES SYSTÈMES D'EXPLOITATION
1ÈRE ANNÉE - FISA
SPÉCIALITÉ INFORMATIQUE

6 novembre 2024 - 30 novembre 2024

SOMMAIRE

Introduction

L'UE SR70 de l'UTBM, encadrée par Mahjoub DRIDI, nous enseigne la programmation système, en mettant l'accent sur le langage C dans un environnement UNIX. En guise d'évaluation finale, un projet a été proposé avec une grande liberté de choix du sujet et la possibilité de travailler en groupe. Nous avons donc choisi de choisir le sujet '**Robots**', décrivant un système industriel de contrôle de fabrication avec des robots. Ce projet a pour objectif de développer un système où plusieurs robots, représentés par des processus, collaborent pour exécuter des tâches spécifiques au sein d'une chaîne de production. L'accent est mis sur la coordination via des mécanismes de communication inter-processus (IPC), comme les files de messages, la mémoire partagée, et les sémaphores. Dans la section suivante, nous présenterons l'architecture globale de notre projet, avant de détailler ses fonctionnalités principales et les défis techniques relevés.

1 Architecture du projet

L'architecture de notre projet repose sur une organisation modulaire des fichiers, chaque fichier ayant un rôle distinct pour refléter la structure d'un système industriel coordonné. Cette organisation permet de séparer clairement les responsabilités entre les différents processus. Voici comment est organisé notre projet :

- **coordinator.c**
  Le fichier central du projet, il représente le **processus coordinateur**, responsable de la gestion et de la supervision globale des tâches et des robots. Le coordinateur contrôle l'attribution des tâches aux robots, surveille leur état et gère les ressources partagées.
- **robot_maker.c**, **robot_painter.c**, **robot_transporter.c**
  Ces fichiers représentent les processus **robots** dédiés, chacun effectuant une étape spécifique du cycle de production :
  - **robot_maker.c** : Responsable de l'assemblage des pièces.
  - **robot_painter.c** : Chargé de peindre les pièces.
  - **robot_transporter.c** : Responsable du transport des pièces.

  Ces fichiers utilisent des sémaphores et des files de messages pour communiquer avec le coordinateur et synchroniser leurs actions.
- **capacity.h**
  Ce fichier contient les constantes définissant la capacité des ressources partagées, comme le nombre de robots, le nombre de pièces dans le système, et les limites des salles (par exemple, salles de peinture, assemblage ou recharge).
- **tasks.h**
  Ce fichier contient les définitions des structures partagées utilisées par tous les processus. Les éléments clés incluent :
  - **Les tâches (`struct Task`)** : Chaque tâche contient des informations sur son état (en attente, en cours, terminé) et son identification.
  - **Les robots (`struct Robot`)** : Informations sur chaque robot, comme son ID, son PID, et son état de disponibilité.
  - **Les sémaphores (`struct Semaphores`)** : Gestion des ressources partagées (salles et files d'attente).
  - **Les messages entre robot et coordinateur (`struct RobotMessage`)** : Message envoyé à un robot pour donner une tâche ou envoyé au coordinateur pour l'informer de la réalisation de celle-ci.

2 Fonctionnalités implémentées

Dans cette section, nous décrivons les différentes fonctionnalités implémentées dans notre projet.

2.1 Gestion des tâches de production

Le coordinateur central est responsable de la gestion des tâches de production, en s'assurant que chaque tâche passe par les étapes d'assemblage, de peinture et de transport. Cette fonctionnalité repose sur une file de tâches partagée et un système de communication inter-processus (IPC) entre le coordinateur et les robots.

2.1.1 Organisation des tâches

Chaque tâche est représentée par une structure partagée (`struct Task`) stockée dans une mémoire partagée. Les tâches passent par plusieurs étapes définies dans la variable `stage` :

- **0** : En attente.
- **1** : Assemblée.
- **2** : Peinte.
- **3** : Transportée (tâche terminée).

La progression des tâches est gérée par le coordinateur, qui attribue chaque tâche au robot disponible en fonction de son état actuel.

```
1 struct Task {
2     int task_id;
3     // 0: Pending
4     // 1: Made
5     // 2: Painted
6     // 3: Transported
7     int stage;
8     int working;
9 };
```

Extrait de code 1 – struct Task (`tasks.h`)

Le coordinateur attribue les tâches aux robots selon leur disponibilité et l'étape en cours de chaque tâche. Une fois attribuée, la tâche est marquée comme `working = 1`, empêchant d'autres robots de travailler dessus.

2.1.2 Traitement des tâches par les robots

Chaque robot reçoit une tâche via une file de messages et utilise des sémaphores pour accéder à des ressources partagées, telles que les salles d'assemblage, de peinture, les docks de transport, de rechargement ou de réparation. Une fois la tâche traitée, le robot met à jour l'état de celle-ci et informe le coordinateur de sa disponibilité pour une nouvelle mission.

```
tasks[message.task_index].working = 1;
log_message("Waiting for workshop...", tasks[message.
    task_index].task_id);
sem_wait(&USED_SEMAPHORE);

log_message("Working...", tasks[message.task_index].task_id);
sleep(2); // Simule le temps de travail
tasks[message.task_index].stage = tasks[message.task_index].
    stage + 1;
tasks[message.task_index].working = 0;

log_message("Made!", tasks[message.task_index].task_id);
```

Extrait de code 2 – Traitement d'une tâche par le robot assembleur (`robot_maker.c`)

2.1.3 Synchronisation et suivi des ressources partagées

La synchronisation est assurée par des sémaphores pour garantir que plusieurs robots ne tentent pas d'accéder simultanément à une même ressource (par exemple, une salle de travail). Le coordinateur surveille l'état de chaque tâche et affiche régulièrement leur progression.

```
void print_tasks_state(const struct Task *tasks) {
    for (int i = 0; i < NUM_TASKS; i++) {
        printf("T%d: S%dW%d / ", tasks[i].task_id, tasks[i].
            stage, tasks[i].working);
    }
    printf("\n");
}
```

Extrait de code 3 – Affichage de l'état des tâches (`coordinator.c`)

```
T1: S3W0 / T2: S3W0 / T3: S3W0 / T4: S3W0 / T5: S3W0 / T6: S3W0 / T7: S3W0 / T8: S3W0 / T9: S2W0 / T10: S1W1 /
Makers                | R0: A0 R1: A1
Painters              | R0: A0 R1: A0 R2: A1
Transporters    | R0: A0
```

FIGURE 1 – Affichage de l'état des tâches dans les logs

La figure 1 illustre l'état des 10 tâches. Les tâches T1 à T8 sont terminées (S3W0), la tâche T9 est peinte et en attente de transport (S2W0), tandis que la tâche T10 est assemblée et en cours de peinture (S1W1). Les lignes "Makers", "Painters" et "Transporters" décrivent les ressources disponibles : deux salles d'assemblage (R0 et R1) où l'une est occupée (R0 : A0), deux salles de peinture occupées sur trois, et la salle de transport unique est également occupée.

2.1.4 Gestion des robots disponibles

Une fois une tâche terminée, le robot informe le coordinateur de sa disponibilité via un message IPC. Le coordinateur met à jour l'état du robot et peut lui attribuer une nouvelle tâche.

```c
void receive_messages(struct Robot robots[], int size) {
    for (int i = 0; i < size; i++) {
        struct RobotMessage rcv_message;
        errno = 0;
        msgrcv(robots[i].queue_id, &rcv_message, sizeof(
           rcv_message), 2, IPC_NOWAIT);
        if (errno != ENOMSG) {
            robots[i].available = rcv_message.recharging == 1
               ? 0 : 1;
        }
    }
}
```

Extrait de code 4 – Mise à jour de la disponibilité des robots

2.1.5 Fin de Production

La production est considérée comme terminée lorsque toutes les tâches atteignent l'étape `stage = 3`. Le coordinateur vérifie périodiquement cet état avant d'arrêter le système.

```
int are_tasks_done(const struct Task *tasks) {
    for (int i = 0; i < NUM_TASKS; i++) {
        if (tasks[i].stage != 3)
            return 0;
    }
    return 1;
}
```

Extrait de code 5 – Vérification de la fin de production

2.2 Coordination entre les robots et le coordinateur (IPC)

Pour ne pas avoir à filtrer les messages du côté de chaque robot, une file de messages est créée pour chaque robot. Ces robots écoutent un message de type 1, qui contient la structure `RobotMessage` avec l'index de la tâche dans laquelle travailler. Les robots ayant chacun attaché le segment de mémoire partagée contenant les tâches, ils peuvent directement affecter un état à la tâche comme précisé précédemment. Les files de messages sont créées par le coordinateur en suivant une convention d'établissement de la clé selon le type de robot et son numéro d'ordre. Ainsi, les robots lancés avec en argument le numéro d'ordre du robot peuvent directement attacher la mémoire partagée et recevoir les messages de la queue qui leur est dédiée.

2.3 Gestion de l'énergie des robots

Chaque robot dispose d'une capacité énergétique limitée, définie par la constante `ENERGY_CAPACITY`. Lorsqu'il exécute une tâche, une quantité fixe d'énergie est consommée (`ENERGY_USED_PER_TASK`), ce qui simule les ressources nécessaires à son fonctionnement. Si l'énergie restante devient insuffisante pour exécuter une nouvelle tâche, le robot interrompt son cycle de travail pour se recharger.

La recharge énergétique suit les étapes suivantes :

— **Vérification de la disponibilité des stations de recharge** : Le robot utilise un sémaphore (`sem_recharge_slots`) pour garantir qu'il accède à une station de recharge disponible, évitant ainsi que plusieurs robots n'occupent simultanément une même station.

— **Simulation du temps de recharge** : Une fois qu'une station est disponible, le robot attend un certain délai (`sleep(5)`), représentant le temps nécessaire pour restaurer complètement sa capacité énergétique.

— **Restauration de l'énergie** : La valeur de l'énergie du robot est réinitialisée à sa capacité maximale (`ENERGY_CAPACITY`).

— **Libération de la station** : Le robot libère le sémaphore pour signaler que la station de recharge est à nouveau disponible pour d'autres robots.

La fonction `repair()` dans le fichier `robot_transporter.c` illustre ce processus de réparation, tandis que la fonction `coin_toss()` gère la probabilité de panne :

```
1  int coin_toss(int probability) {
2      return rand() % 100 < probability;
3  }
4
5  void repair() {
6      log_message("Breaking down, going to repair room...", -1);
7      sem_wait(&semaphores->sem_repair_slots); // Verification
           de la disponibilite d'une salle de reparation
8      sleep(5); // Simulation du temps de reparation
9      sem_post(&semaphores->sem_repair_slots); // Liberation de
           la salle de reparation
10     log_message("Repaired and operational!", -1); //
           Indication que le robot est a nouveau fonctionnel
11 }
```

Extrait de code 6 – Simulation d'une panne et réparation d'un robot

2.4 Simulation des pannes des robots

Chaque robot a une probabilité fixe (`BREAKDOWN_PROBABILITY`) de tomber en panne après l'exécution d'une tâche. Après chaque action, on "lance une pièce" biaisée en fonction de cette probabilité : si on "gagne", le robot tombe en panne, il cesse de fonctionner. Par défaut, cette probabilité est définie à 20% par exemple (ou une autre valeur selon les spécifications du robot). Lorsqu'une panne est détectée, le robot interrompt son travail et est envoyé dans une salle de réparation (si la salle est disponible). Ce processus est géré via un sémaphore pour garantir un accès exclusif aux salles de réparation.

Le fonctionnement de la simulation des pannes est le suivant :

- **Détection de la panne** : Après chaque tâche, une fonction de "pile ou face" (`coin_toss`) est utilisée pour déterminer si le robot tombe en panne. Cette fonction génère un nombre aléatoire et vérifie s'il est inférieur à la probabilité définie.
- **Accès à la salle de réparation** : Si le robot tombe en panne, il utilise un sémaphore (`sem_repair_slots`) pour accéder à une salle de réparation disponible.
- **Simulation du temps de réparation** : Une fois qu'une salle est disponible, le robot attend un délai (`sleep(5)`) pour simuler le processus de réparation.
- **Libération de la salle** : Après réparation, le robot libère la salle en signalant au sémaphore que la ressource est à nouveau disponible.
- **Reprise du travail** : Une fois réparé, le robot est de nouveau opérationnel et prêt à reprendre ses tâches.

```
1  int coin_toss(int probability) {
2      return rand() % 100 < probability;
3  }
4
5  void repair() {
6      log_message("Breaking down, going to repair room...", -1);
7      sem_wait(&semaphores->sem_recharge_slots); // Utilisation
           d'une salle partagee
8      sleep(5); // Simule le temps de reparation
9      sem_post(&semaphores->sem_recharge_slots);
10     log_message("Repaired and operational!", -1);
11 }
```

Extrait de code 7 – Simulation d'une panne et réparation d'un robot

Conclusion

Le projet de fin d'UE nous a permis de voir l'application en pratique, dans un cas plus concret, des mécaniques qui nous ont été enseignées. Nous avons implémenté les fonctionnalités clés d'un simulateur de gestionnaire de robots, telles que la coordination via l'IPC, la gestion des ressources partagées avec des sémaphores, et la simulation de la consommation énergétique et des pannes. Ces éléments constituent les bases essentielles d'un système robuste et ont été réalisés avec succès. Nous n'avons pas pu implémenter toutes les fonctionnalités à 100%, notamment la gestion multi-sites, en raison de contraintes de temps liées aux autres projets des UE du semestre. Cependant, avec plus de temps, certaines extensions auraient pu enrichir le simulateur. Par exemple, la gestion multi-sites aurait ajouté une dimension supplémentaire, simulant des environnements de production collaboratifs. Le projet reste néanmoins pleinement fonctionnel, flexible et nous sommes plutôt satisfaits du résultat.