**Explication du code**

Réutilisation du code du dossier PETRA monopièce. Le cycle pour une pièce reste globalement le même mais de nouveaux éléments ont été ajoutés afin de permettre une circulation multipièce.

**Critères de l’application** : Un processus par pièce et 3 pièces en circulation en même temps. Plutôt que du multiprocessus, nous avons mis en place une application multithreads. Du coup, au lieu d’un processus par pièce, nous avons un thread par pièce.

**Nouvelles variables** :

Pthread\_t tid1, tid2, tid3 et tidm : Variables qui vont contenir les tids des threads 1, 2, 3 (un thread par pièce) et mauvaise (thread pour la séquence de récupération d’une mauvaise pièce).

Pthread\_mutex\_t mutexChariot : Mutex utilisé pour les sections critiques manipulant le chariot. Cela a pour but d’éviter que deux threads manipulent le chariot en même temps.

Pthread\_mutex\_t mutexMauvaise : Mutex utilisé pour la section critique du thread mauvaise afin de bloquer les autres threads durant la récupération d’une mauvaise pièce.

Pthread\_cond\_t cond : Variable de condition utilisée afin de synchroniser un thread pièce avec le thread mauvaise en cas de mauvaise pièce.

Pthread\_mutex\_t mutex : Mutex associé à la variable de condition cond.

Int mutexOwner : Variable dans laquelle est stocké le tid d’un thread pièce s’il est en train d’exécuter le code de la section critique protégée par mutexChariot. Cette variable servira à déterminer quels threads pièce devront recevoir un signal ou non si la section critique d’une mauvaise pièce est en cours.

Int condOwner : Variable dans laquelle est stocké le tid d’un thread pièce s’il est en train d’attendre sur la variable de condition. Cette variable servira à déterminer quels threads pièce devront recevoir un signal ou non si la section critique d’une mauvaise pièce est en cours.

Bool m : Variable locale à chaque thread pièce qui servira à mémoriser si une pièce est bonne ou mauvaise.

**Sections importantes du code** :

Le main :

Le main agit comme un lance thread, c’est-à-dire qu’il se contente de créer les quatres threads nécessaires au contrôle du petra. Trois threads identiques sont lancés (afin de gérer une pièce par thread pour un total de trois pièces en même temps) et un quatrième thread est lancé pour s’occuper de la séquence en cas de mauvaise pièce. Le main se termine ensuite via un pthread\_exit afin de ne pas terminer ses sous-threads.

Le thread pièce (fctThread) :

En premier lieu, le signal SIGUSR1 est armé pour chaque thread pièce afin d’exécuter le Handler correspondant à la réception de ce signal. Ensuite, une boucle infinie est lancée dans laquelle on vérifie à chaque tour que le réservoir de pièce n’est pas vide (s’il est vide on se contente de l’afficher).

Si le réservoir n’est pas vide, une section critique protégée par mutexChariot s’exécute. Le premier thread à verrouiller mutexChariot stocke son tid dans la variable mutexOwner et exécute dans cette section le code pour prendre une pièce et la déposer sur le convoyeur 1 puis attendre deux secondes avant de déverrouiller mutexChariot. Cette section critique a deux buts : permettre de protéger les modifications sur l’actuateur du chariot pour ne pas que deux threads puissent le manipuler en même temps et permettre de creuser un certain écart entre chaque pièce afin qu’il y ait bien trois pièces qui circulent en même temps sur le petra.

Une fois la section critique terminée, mutexOwner est remis à -1 et une vérification sur mutexMauvaise est faite. Si mutexMauvaise était verrouillé pendant la section critique précédente, cela veut dire que les convoyeurs étaient stoppés (voir explication thread mauvaise). Du coup, l’attente de deux secondes s’est effectuée mais la pièce n’a pas bougé dans cet intervalle. Ainsi, dans ce cas, pour être sûr que l’écart entre les pièces est respecté, nous attendons que mutexMauvaise soit relaché (voir explication thread mauvaise) avant de relancer l’attente de 2 secondes.

Le reste du code pour gérer l’acheminement de la pièce est ensuite effectué et deux cas peuvent survenir : soit la pièce est bonne et le tour de boucle se termine, soit la pièce est mauvaise et dans ce cas, un signal est envoyé à la variable de condition (sur laquelle le thread mauvaise attend dès le début) et le thread pièce se met en attente sur cette même variable (tout en stockant son tid dans la variable condOwner) jusqu’à ce que le thread mauvaise renvoie un signal après avoir terminé sa séquence de récupération d’une mauvaise pièce (condOwner est alors remis à -1). Une fois ce dernier signal reçu, le tour de boucle se termine.

Le thread Mauvaise (fctMauvaise) :

Ce thread est le seul à gérer les convoyeurs vu qu’il doit les stopper en cas de mauvaise pièce. Ainsi, au début, une boucle infinie est lancée dans laquelle les deux convoyeurs sont activés et le thread se met en attente sur la variable de condition.

Lorsque le thread est réveillé par un signal sur la variable de condition, celui-ci rentre dans une section critique protégée par mutexMauvaise. Dans cette section critique s’en trouve une autre protégée par mutexChariot afin de ne pas manipuler le chariot s’il est déjà manipulé par un thread pièce.

Avant de verrouiller mutexChariot, les convoyeurs sont désactivés et un signal SIGUSR1 est envoyé aux threads dont le tid n’est pas égal à condOwner et mutexOwner (le thread pièce dont le tid est stocké dans condOwner est déjà en attente sur la variable de condition et le thread pièce dont le tid est stocké dans mutexOwner (s’il y en a un à ce moment-là) attendra que mutexMauvaise soit relaché après avoir fini d’exécuter sa propre section critique). Les threads recevant le signal s’interrompront pour exécuter le Handler correspondant dans lequel ils seront en attente tant que mutexMauvaise est verrouillé. Ainsi, tous les threads pièces sont en attente pendant une séquence de récupération de mauvaise pièce.

Après avoir verrouillé mutexChariot, le code pour récupérer la pièce et la placer dans la poubelle est effectué. Ensuite, mutexChariot et mutexMauvaise sont relachés et un signal est envoyé à la variable de condition (ainsi, tous les processus pièces reprennent leur exécution). Le tour de boucle se termine afin qu’au nouveau tour les convoyeurs soient réactivés.