UTBM

Atelier de production

Projet : LO41

Etienne Maillard
Maxime Study
06/01/2017

Table des matières

Introduction

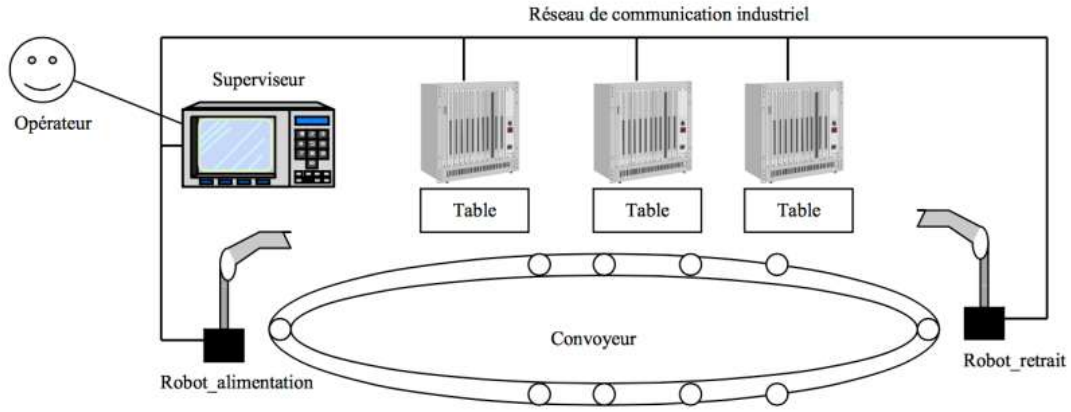
Dans le cadre de l'UV LO41, nous avons pour but de réaliser un atelier de production, seul ou à deux permettant ainsi de mettre en avant nos connaissances sur les threads, sémaphores, mutex et autres outils de synchronisation vu lors de nos séances de travaux pratique. Connaissant la date de rendu le vendredi 06 janvier 2017, nous avons géré notre projet de la manière suivante.

Dans un premier temps, nous avons longuement réfléchis pour se mettre d'accord sur les outils que nous allions mettre en place. De ce fait, nous avons réalisé des schémas et un réseau de pétri pour établir les différents processus et thread composant l'application. Nous avons ensuite réalisé du pseudo code, qui nous a servis de base de codage.

Explication du sujet

Sujet

Soit un atelier de production composé d'un convoyeur, un robot d'alimentation de pièces à usiner, M machines d'usinage et un robot de retrait de pièces usinées. L'ensemble est relié par un réseau de communication (voir figure ci-dessous).



Le superviseur reçoit ses ordres de son environnement représenté ici par un opérateur. On suppose que le convoyeur tourne en permanence et que sa vitesse est choisie de manière adéquate pour que les robots et machines puissent manipuler les pièces sans problème. Quand l'opérateur signale au superviseur l'arrivée d'une nouvelle pièce à usiner (on suppose que chaque pièce est accompagnée d'un code qui spécifie l'opération d'usinage à lui appliquer), le superviseur détermine la machine qui va l'usiner et envoie un ordre au robot d'alimentation pour déposer la pièce sur le convoyeur et à la machine d'usinage concernée pour se préparer.

Le robot d'alimentation effectue l'opération de placement sur le convoyeur au bout de 20 secondes au maximum, s'il n'y arrive pas il génère un signal d'anomalie vers le superviseur qui fait passer le système dans un état de défaillance. Lorsqu'une machine d'usinage retire la pièce qu'elle doit usiner, elle la dépose sur sa table et envoie un signal de libération du convoyeur au superviseur. Si la machine n'a pas retiré la pièce au bout de 50 secondes, le système passe dans un état de défaillance.

Chaque machine d'usinage possède sa propre table d'usinage sur laquelle elle dépose la pièce durant l'opération d'usinage. Lorsqu'une machine d'usinage termine son travail, elle envoie un compte rendu au superviseur et attend de celui-ci un ordre pour déposer la pièce sur le convoyeur. Si une machine n'a pas fini son travail au bout de 10 minutes, le superviseur avertit l'opérateur en déclarant la machine

en panne, mais n'arrête pas le système. Si une pièce arrive et qu'elle nécessite une opération sur une machine déjà déclarée en panne, le superviseur passe dans un état de défaillance. Lorsqu'un compte rendu de fin d'usinage est reçu, le superviseur attend que le convoyeur soit libre, ensuite il envoie un ordre à la machine d'usinage concernée pour déposer la pièce sur le convoyeur et un ordre au robot de retrait pour retirer la pièce afin de l'envoyer au dépôt. Quand la pièce est retirée un compte rendu est renvoyé au superviseur par le robot de retrait. Si le robot de retrait n'a pas fini son travail au bout de 30 secondes, le superviseur fait passer le système dans un état de défaillance. Comme il n'y a qu'un seul convoyeur, le superviseur doit assurer sa bonne utilisation : une seule pièce à la fois sur le convoyeur. On suppose que tous les équipements (robots et machines d'usinage) sont munis de capteurs pour détecter l'arrivée de pièces et leur départ du convoyeur. On suppose également que les robots et machines peuvent manipuler les pièces alors que le convoyeur est en mouvement.

Fonctionnement attendu

Le programme doit être composé d'un makefile permettant de compiler automatiquement les différents fichiers. De plus, il doit être accompagné d'un fichier README expliquant comment lancer le programme.

Problématique

Dans ce sujet, de nombreux points techniques sont à prendre en compte. Tout d'abord, il y a les états de défaillances. Ces états de défaillance doivent être lancés dans différentes circonstance qu'il fallait bien analyser. Le deuxième point est le superviseur. On doit être capable de suivre et détecter le bon déroulement de chaque machines et robots. Le tout doit bien être synchronisé pour éviter de perdre des pièces.

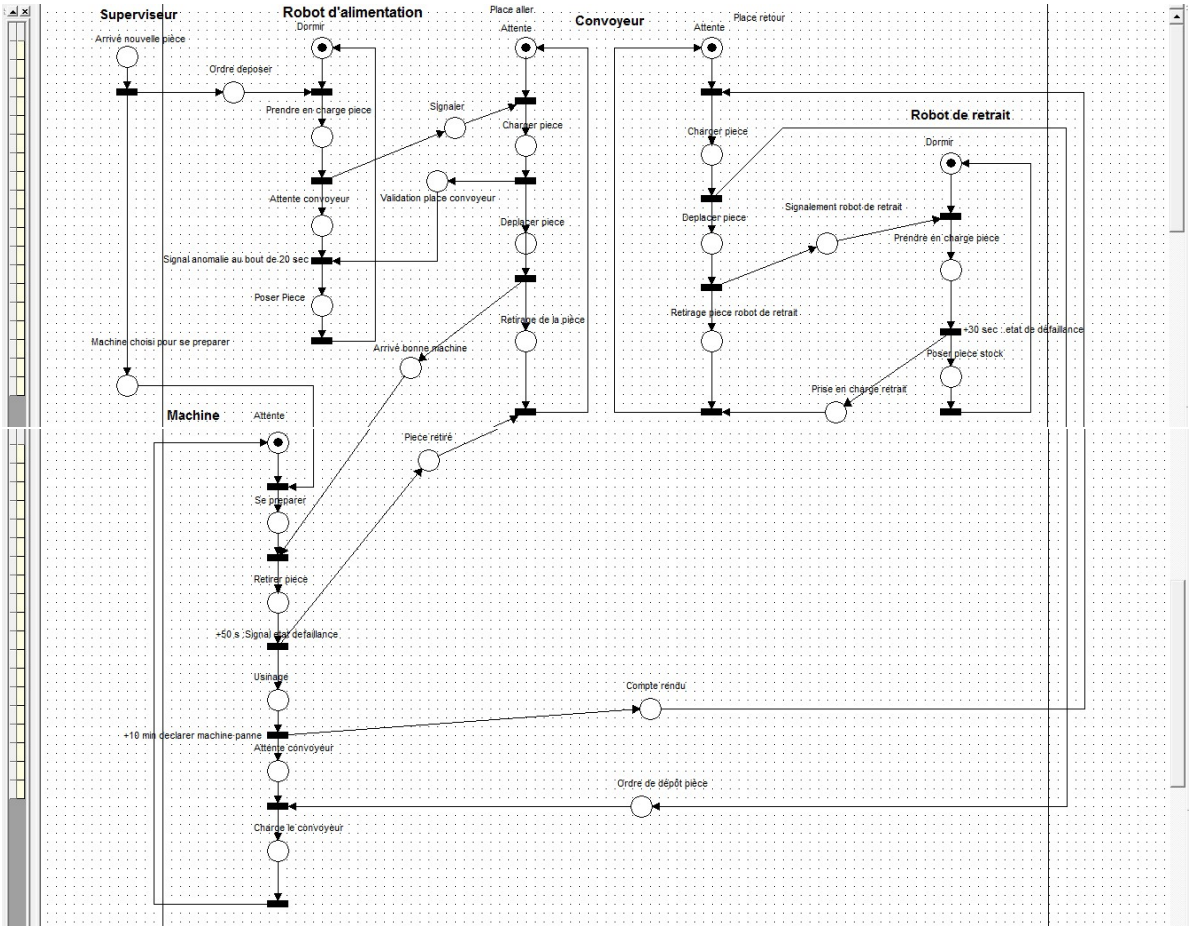
Explication de notre solution

Justification de notre solution

Pour réaliser l'atelier de production, nous avons choisi la solution avec les threads et moniteur. Au départ du projet, la plus grande difficulté était de savoir comment gérer les différents états de défaillance. Après quelques recherches sur internet, nous sommes tombés sur la fonction pthread_cond_timedwait qui permettait de répondre exactement au cahier des charges. De plus, nous avions déjà eu de l'expérience avec les sémaphore et threads en python lors des années précédente. C'est pourquoi nous nous sommes naturellement dirigés vers une application avec des sémaphores et moniteurs.

Conception

Lors de notre phase de conception, nous avons créé le réseau de Pétri suivant :



Le réseau de Pétri est un outil de conception qui nous a permis de bien comprendre et modeliser notre système. Cependant, lors de la mise en place de l'application, nous nous sommes aperçu que certains aspect de ce réseau de Pétri n'étaient pas optimaux pour le fonctionnement du projet.

Pendant la réalisation de notre réseau de Pétri, nous avons mis en place un brainstorming afin de determiner comment réaliser les différentes fonctions de notre application (pseudo code). Ainsi, nous avons pu répartir les différentes tâches de manière à être plus efficace dans notre travail.

Extrait du pseudo code :

```
superviseur :
  -on crée une liste contenant x liste chainé (x = nb type différent)
  -fonction creePiece :
    . crée la structure d'une piece : type, id, usine (bool)
    . on détermine la machine en fonction du type   :    on regarde quelle table a le meme type que la piece
    . on met la piece dans la liste chainé d'attente du type d'opération
```

Mise en place du système

Pour réaliser notre projet, nous avons créé un dépôt git. Cela permet à la fois de partager le code assez facilement et de réaliser du versioning. Notre projet contient différents fichiers et dossiers :

- Un readme.md pour expliquer comment lancer le projet

- Un makefile pour compiler facilement le projet
- Un dossier header qui contient toute les entêtes des fichiers
- Un dossier src qui contient les fichiers sources
- Un dossier object pour les fichiers .o

Nous avons créé un fichier source par élément logique du système :

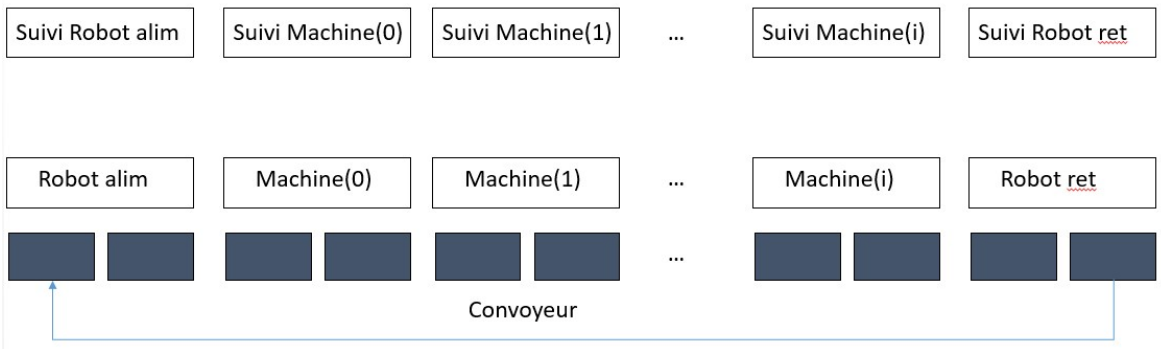
- IHM
- Convoyeur
- Machine
- Main
- Robot
- Superviseur

Chaque fichier contient les éléments qui lui sont propres.

Fonctionnement de notre système

Notre système est constitué de n machines disposant de n opérations. Chaque machine dispose de l'opération n (c'est-à-dire de son numéro) et d'un temps d'opération de i secondes. Le convoyeur est créé en fonction du nombre de machines. Chaque machine et robot disposent de deux emplacements de convoyeur.

Ainsi la taille total du convoyeur (tableau) est de tailleTotal= (nbRobot(2) + nbMachine(i))*2. Le convoyeur fonctionne de manière particulière. Il y a les cases paires et les cases impaires. Les cases paires sont utilisées pour la transmission de pièces entre le robot d'alimentation et les machines et les cases impaires sont utilisées pour la transmission des pièces usinées entre les machines et le robot de retrait. Cette technique nous permet d'avoir toujours de la place sur le convoyeur et de ne pas à savoir si la pièce a été usiné ou non. Le convoyeur possède une vitesse de rotation qui est simulé dans notre projet par une attente en chaque déplacement (chaque case est déplacée vers la droite).

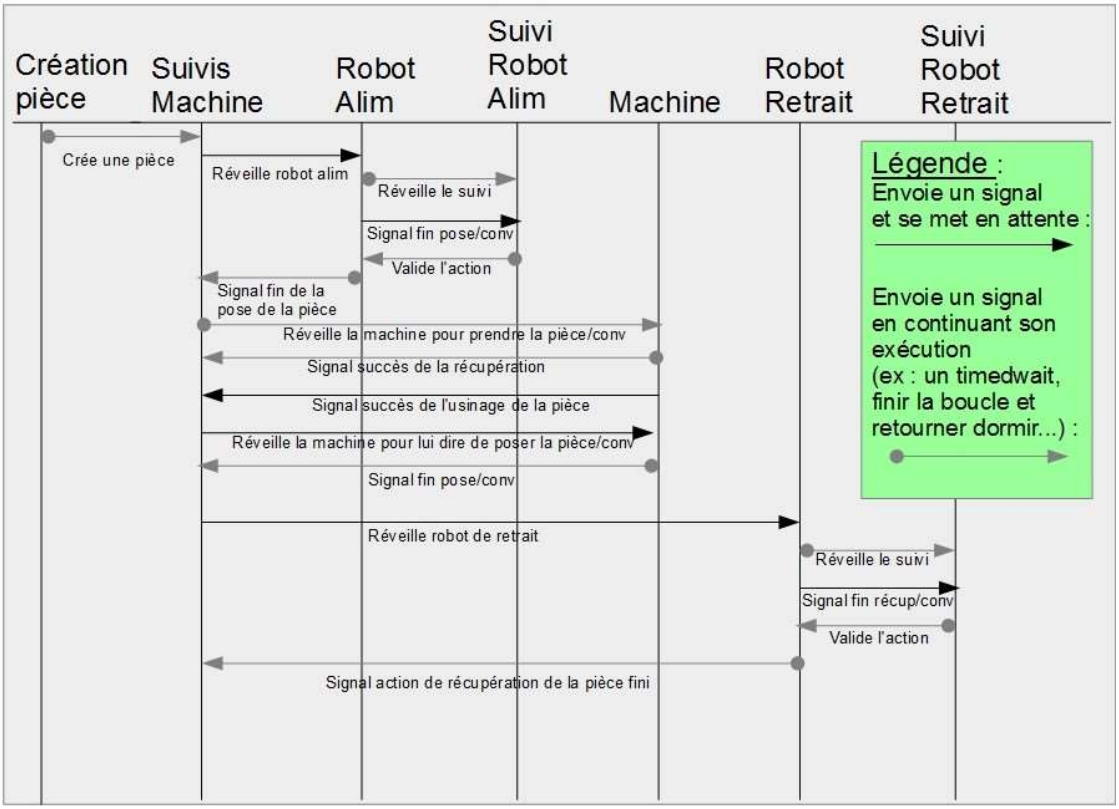


Chaque pièce dispose d'un identifiant num et d'une opération ope. Lors d'un ajout de pièce, la pièce est mise dans la liste d'attente de la machine d'opération de son type d'opération.

```
struct piece {
  int num;
  int ope; //operation d'usinage
};
typedef struct piece piece;
```

Pour réaliser la liste d'attente, nous avons créé une liste chaînée. Ainsi, nous ne sommes pas contraint au nombre de pièces en attente et ce système pourrait permettre d'ajouter des pièces dynamiquement pendant le fonctionnement du système.

Voici comment pourrait être représenté notre application finale sous forme de diagramme de séquence :



Avant toute chose, nous devons mettre en place tous les threads : robot d'alim, robot de retrait, x*machine, convoyeur, mais aussi différents threads qui compose le superviseur : suivis du robot d'alim, suivi du robot de retrait, nombreDeMachine*suivi d'une machine.

Lorsque tous les threads sont créés, (ils se créent l'un à la suite de l'autre et sont stoppé lorsqu'ils sont créés par un lock(&mtx_menu) qui bloque ainsi le menu jusqu'à ce que le dernier thread soit bien initialisé). Une fois que tout est place, nous pouvons ajouter les pièces.

Lorsque nous ajoutons une pièce, nous regardons si la liste d'attente de la machine du type d'opération de la pièce est vide. Si c'est le cas, soit le suivi du robot d'alimentation dort soit il est en train de suivre une pièce.

Pour plus de compréhension, il est préférable d'avoir la fonction threadSuiviMachine du fichier superviseur.c

A l'initialisation, tous les threads sauf le convoyeur se lance et s'endorme au début de leur boucle while.

Ainsi lors de la création de la première pièce, le thread de suivi de la machine se réveil. (cf. image ci-dessous) :

```
120     int test = 0;
121     if (recupererElementEnTete(maListeMachine[ope]->listeAttente) == NULL){
122           test = 1;
123     }
124     maListeMachine[ope]->listeAttente=ajouterEnFin(maListeMachine[ope]->listeAttente, *nouvellePiece);
125
126     if (test == 1){ //si la liste d'attente pour la machine était vide
127           pthread_cond_signal(&(maListeMachine[ope]->dormir)); //on previent qu'il y a une pieces
128           pthread_mutex_unlock(&(maListeMachine[ope]->mutMachine)); //on libere le mutex
129     }
```

Figure 1 : threadSuiviMachine superviseur.c

On bloque ensuite le mutex du robot d'alimentation. Ainsi aucun autre thread de suivi ne pourra accéder au robot d'alimentation tant que le thread qui à la main ne relâche le mutex.

```
244                 pthread_mutex_lock(&MitSurRobotAlim); //on tente de lock pour poser la piece sur le robot d'alim
```

On pose ensuite notre pièce sur le robot d'alimentation puis on envoie un signal au robot d'alimentation pour qu'il se réveil.

Le suivi de la machine s'endort.

Le robot d'alimentation se réveil et envoie un signal à son suivi pour qu'il se réveil. Le robot d'alimentation commence à tenter de poser la pièce sur le convoyeur. Il ne peut poser une pièce seulement si le convoyeur est sur une rotation paire. Il s'endort donc après chaque tentative d'ajout de la pièce sur le convoyeur et se fait réveiller par le convoyeur qui envoie un signal de réveil toutes les deux rotations.

Lorsqu'il se fait réveiller, il regarde s'il y a une pièce dans l'emplacement devant lui sur le convoyeur (index 0). S'il n'y en a pas, il pose la pièce et sort de la boucle. Il avertit aussitôt le suivi du robot d'alimentation pour lui indiquer qu'il a réussi à poser la pièce sur le convoyeur et s'endort (cf. image ci-dessous).

```
while (1){ //on lance la boucle pour retirer la piece
        pthread_cond_wait(&condPose/*2*/,&mutexConvoyeur ); //on attend que ce soit impair
        if (conv[(tailleConv-1)].num != -1){ //impair donc piece usiné

                pieceRobotRetrait = retirerPieceConvoyeur(tailleConv-1); //on retire la piece
                fonctionPrevenirAffichage();
                pthread_mutex_unlock(&mutexConvoyeur);
                op=pieceRobotRetrait.ope;
                break;
        }
        pthread_mutex_unlock(&mutexConvoyeur);
}
```

Figure 2 :fonc_robotRetrait robot.c

En effet le suivi du robot d'alimentation (cf image ci-dessous) après s'être réveillé, s'était à nouveau mis en attente mais nous avons utilisé ici : pthread_cond_timedwait. Cette fonction permet d'endormir un thread un certain temps (donné en paramètre). Si il n'est pas réveillé par un signal avant ce délais, le thread se réveil et sort avec un code différents de 0. Dans ce cas, un message est envoyé

à l'opérateur et le système entier s'arrête. Si au contraire, il se fait réveiller avant le temps impartis (20 secondes), cela veut dire que le signal envoyé par le robot d'alimentation après avoir posé la pièce a été reçu. Le suivi du robot d'alimentation envoie alors un signal au robot d'alimentation pour lui dire que tout s'est bien passé. Le suivi retourne alors s'endormir en recommençant sa boucle.

```
337  void * fonc_SuiviRobotAlim() {
338          pthread_mutex_lock(&mtx_menu);
339          pthread_mutex_unlock(&mtx_menu);
340          struct timeval tp;
341          struct timespec ts;
342          while(1){
343                  pthread_cond_wait(&RobotSuiviAlim,&mutexAlim);
344                  pthread_mutex_unlock(&mutexAlim); //on débloque le mutex
345                  gettimeofday(&tp, NULL);
346                  ts.tv_sec = tp.tv_sec;
347                  ts.tv_nsec = tp.tv_usec*1000;
348                  ts.tv_sec += 20;
349                  //on s'endort le temps que la piece soit posé sur le convoyeur.
350                  //mais si le temps dépasse les 5 secondes de traitement (ts), le SYSTEME passe en défaillant
351                  if(pthread_cond_timedwait(&RobotSuiviAlim,&mutexAlim,&ts) != 0){
352                          pthread_mutex_unlock(&mutexAlim);
353                          printf("DEFAILLANCE ROBOT D'ALIMENTATION!! \nLe systeme passe en défaillance\n");
354                          killThreads(); //tout le systeme est KO
355                          break;
356                  }
357                  pthread_cond_signal(&RobotAlim);
358
359                  pthread_mutex_unlock(&mutexAlim);
360          }
361          pthread_exit(NULL);
362  }
```

Figure 3: fonc_SuiviRobotAlim superviseur.c

De retour au robot d'alimentation qui vient de se réveiller, on signale alors au suivi de la machine qui nous avait réveillés pour indiquer que la pose s'est effectué avec succès. Et le robot retourne s'endormir au début de sa boucle.

Le suivi de machine se réveil et débloque le mutex du robot d'alimentation. Ainsi un autre suivi de machine qui souhaite poser une pièce peut accéder au robot d'alimentation.

Le suivi de la machine réveil à ce moment-là la machine qu'il supervise.

La machine se réveille et scrute son index sur le convoyeur l'arrivé d'une pièce. Le suivi à lui lancé comme dans le suivi du robot d'alimentation un "timedwait" de 15 secondes qui est le temps acceptable pour que la pièce soit retirée du convoyeur. Si ce n'est pas le cas, comme précédemment, l'atelier entier s'arrête.

S'il se fait réveiller avant, il se relance dans un "timedwait" mais cette fois pour le temps d'usinage. En effet, la machine après avoir envoyé un signal après avoir retiré la pièce (de la même manière qu'on ajoute une pièce sur le convoyeur (cf robot d'alim)), on lance directement l'usinage de la pièce, simulé par un sleep.

Une fois l'usinage terminé, on indique au suivi que l'usinage s'est bien réalisé en le réveillant et la machine s'endort. Si il n'avait pas été réveillé avant 10 minutes (temps sujet mais baissé à 20 secondes pour le projet), la machine aurait été passé en défaillance et son thread ainsi que celui du suivi auraient été tué.

Si l'usinage s'est bien déroulé, le suivi se fait réveiller et il lock alors l'accès au robot de retrait (ou si le mutex du robot de retrait est déjà utilisé, il attend qu'il ne le soit plus).

Au moment où il prend la main sur le robot de retrait, il réveille la machine pour qu'elle pose la pièce sur le convoyeur (sans limite de temps). La machine pose alors la pièce sur le convoyeur, réveil son suivi et fini l'exécution de sa boucle puis s'endort à nouveau au début de celle-ci.

Le suivi de la machine réveil alors le robot de retrait pour qu'il s'apprête à retirer la pièce et s'endort.

```
294                 pthread_mutex_lock(&AttentRetrait); //on bloque le robot de retait
```

De la même manière que pour le robot d'alimentation, le robot de retrait effectue les mêmes tâches et lorsqu'il a retiré la pièce, il réveille le suivi de machine.

Toutes les boucles finissent leur exécution et retourne dormir sauf le suivi de la machine qui, s'il y a une pièce en attente dans la liste d'attente de sa machine passe directement à l'étape où il tente de lock le mutex du robot d'alimentation.

IHM

Utilisation

Notre interface homme machine permet d'afficher les différents éléments de l'atelier de production. Notre processus principal lance le menu. Le menu permet de sélectionner les différents modes de notre application :

- Le mode par défaut créé 4 machines et reparties 8 pièces entre chaque machines (2 pièces chacune).
- Le mode personnalisé permet à l'utilisateur de saisir la vitesse du convoyeur, le nombre de machines et le nombre de pièce à usiner pour chaque machine. Malheureusement, on ne peut pas choisir le temps d'usinage pour chaque machine, ni le seuil de tolérance pour les défaillances (temps maximal à ne pas dépasser).
- Le mode défaillance permet de lancer une défaillance du système (il utilise pour base le mode par défaut)

Les temps de défaillance par défaut ont été initialisé à et ne peuvent pas être changé :

  - 20 secondes pour le robot d'alimentation
  - 30 secondes pour le robot de retrait
  - 15 secondes pour que la machine retire la pièce
  - 20 secondes pour que la machine réalise son usinage

Pour terminer l'affichage du fonctionnement du système, il faut envoyer SIGINT avec CTR+C.

Outil de synchronisation utilisée lors du développement de l'IHM.

Lorsque l'utilisateur indique le mode qu'il souhaite réaliser, l'IHM lance la fonction dans un processus fils (fork). L'atelier de production est une boucle (threads daemon). Pour arrêter l'application, nous sommes passés par des signaux. Le thread principale intercepte le CTR-C (SIGINT) et modifie son comportement. Le processus fils (l'atelier à proprement parlé), intercepte également SIGINT pour qu'il puisse réaliser les différentes libérations de mémoire (free, pthread_cond_destroy et pthread_mutex_destroy) afin d'éviter la présence de fuites de mémoire. Après l'envoie de SIGINT, le fils se tue, ce qui permet au père de continuer car il attendait la fin du processus fils avec waitpid.

Pour réaliser l'affichage, nous avons mis en place un thread (daemon) affichage qui affiche tous les pièces du système. Une pièce est initialisé a piece.num =-1 et piece.ope=-1 (opération pas obligatoire

de le remplir mais on préfère mettre une valeur par défaut) dans l'objet (machine, robot ou case du convoyeur) si l'appareil ne contient pas de pièce.

```
100    pieceRobotAlim=pieceVideConv;
101    pieceVideSortie.num=-1;
102    pieceVideSortie.ope=-1;
103    pieceRobotRetrait=pieceVideSortie;
```

Quand une pièce rentre dans une machine ou robot, elle remplace la pièce vide par la pièce qui vient d'être récupérée. De plus, quand la pièce est posé sur le convoyeur, on remplace la pièce du système par la pièce qu'on à récupérer. Lorsque l'on change la pièce, on prévient le thread d'affichage.

```
pieceRobotAlim=pieceVideConv; // on a livré le piece
fonctionPrevenirAffichage();

 9  void fonctionPrevenirAffichage()
10  {
11    /* Dire à l'affichage que j'ai modifié qqch d'important*/
12    pthread_mutex_lock(&mutAffichage);
13    pthread_cond_signal(&condAffichage);
14    pthread_mutex_unlock(&mutAffichage);
15  }
```

La fonction « prévenir affichage » permet de dire au thread affichage qu'un élément a été modifié dans le système.

```
28  void * fonc_afficher()
29  {
30        while(1)
31        {
32          pthread_cond_wait(&condAffichage, &mutAffichage);
33          printf("****************************************\n");
34          int i;
35          int j=0;
```

Grace au système de condition et de sémaphore, le thread n'affiche que lorsque on en a besoin, c'est-à-dire lorsqu'il y a une modification dans les différentes pièces des différents objets (pose ou prise d'une pièce).

Amélioration futur

Une des améliorations de notre système pourrait être de pouvoir ajouter des pièces à la volée pendant le déroulement de l'atelier de production. Une des autres améliorations serait de pouvoir saisir plus de paramètre tel que le temps d'usinage de l'opération et les temps de défaillance.

Conclusion

Lors de ce projet, nous avons utilisé de nombreuses connaissances vues lors de nos séances de TP (processus, threads, signaux, moniteurs…) sur la communication interprocessus. De plus, nous avons dû apprendre de nouvelles notions système pour pouvoir simuler pleinement notre atelier d'usinage (pthread_timedlock, p_cancel…).

Nous avons essayé de reproduire le système de la manière la plus fiable possible par rapport à la réalité, c'est pourquoi nous avons apporté quelques modifications au sujet.

De plus, étant en binôme, nous avons pu mettre en place des outils pour le versionning (git) et gérer la répartition des tâches. Ce projet nous a apporté sur le plan humain, technique et organisationnel.