INF5170 — Programmation parallèle
Examen final (Automne 2000)

Durée: 13h30 – 16h30 (3 heures) **Documentation autorisée:** Toute documentation personnelle.

1. Affirmations fausses à corriger (10 pts)

Chacune des six (6) affirmations suivantes est fausse. Parmi ces six affirmations, *choisissez en cinq (5)* et expliquez *brièvement* pourquoi elles sont fausses.

1. On appelle sémaphore *général* un sémaphore qui peut être utilisé par n'importe quel processus. Quant à un sémaphore *binaire*, il ne peut être utilisé que par deux processus.

2. Un programme est dit `SPMD` lorsque, à chaque instant, tous les processus exécutent la même instruction. De tels programmes sont donc parfaits pour des machines SIMD plutôt que MIMD.

3. Un moniteur est un processus actif qui reçoit, de la part d'autres processus, des requêtes pour accéder à une structure de données privée et qui assure, de façon implicite, que tous les accès seront fait de façon mutuellement exclusive.

4. Il est impossible de réaliser des systèmes clients/serveurs à l'aide de processus communiquant uniquement par l'intermédiaire de messages échanges sur des canaux. C'est pour cette raison que les approche avec RPC et Rendez-vous sont introduites.

5. En Java, un moniteur est obtenu à partir d'une classe en ajoutant le mot réservé `synchronized` devant un nom de méthode. Ainsi, lorsque ce mot clé apparaît devant un nom de méthode `m0`, cela assure alors que la méthode `m0`, pour un objet donné, ne sera jamais exécutée en même temps que n'importe quelle autre méthode s'exécutant sur le même objet.

6. En Pthreads (librairie de *threads* de POSIX), l'opération `sem_wait` correspond à l'opération `P` de la notation d'Andrews, alors que l'opération `sem_post` correspond à l'opération `V`. Par contre, contrairement à l'opération `P`, l'opération `sem_wait` n'est *pas* bloquante — on dit alors qu'elle est *split-phase*.

2. Sémaphore généralisé avec opération explicite d'initialisation (10 pts)

On désire réaliser un sémaphore, mais pour lequel une opération explicite d'initialisation `Init` doit être appelée *avant* que les opérations `P()` et `V()` puissent être utilisées. Si ces dernières sont appelées avant l'appel à `Init`, elles sont alors bloquées jusqu'à ce que l'initialisation avec `Init` soit effectuée. En outre, après le premier appel à `Init`, si d'autres appels à `Init` sont effectués, ces derniers n'ont aucun effet (mais ne bloquent pas).

L'interface de l'opération `Init` est la suivante:

```
Init( int val_init );      /* Valeur initiale du semaphore. */
```

Écrivez un module réalisant un tel sémaphore en utilisant *l'une ou l'autre* des approches suivantes:

1. Un module moniteur;

2. Un module utilisant une approche par rendez-vous.

Note: Un bonus sera accordé si vous développez *correctement* les deux modules.

3. Modules pour un `CompteBancaire` (20 pts)

Un compte bancaire est partagé par un groupe de personnes. Chaque personne peut déposer ou retirer des fonds de ce compte. Le solde courant à un instant donné est la somme de tous les dépots ayant été effectués à date moins la somme de tous les retraits qui ont été acceptés. Ce solde ne peut et ne doit jamais être négatif (pas de marge de crédit). Une opération de retrait doit être *retardée* (mise en attente) si le montant à retirer est supérieur au solde courant.

On veut définir un module logiciel pour réaliser un tel compte bancaire. Les en-têtes des deux opérations exportées par le module seront simplement les suivantes:

```
deposer( float montant );
retirer( float montant );
```

Dans les deux cas, il s'agit donc de procédures qui ne retournent *aucun* résultat. De plus, pour simplifier, il n'y a pas d'opérations pour examiner le solde du compte et un montant est simplement un nombre point-flottant.

Vous devez définir différentes versions de ce module. Plus précisément, vous devez définir deux (2) mises en oeuvre parmi les quatre (4) possibilités suivantes:

1. Un moniteur (chap. 5).
2. Un processus de gestion du compte bancaire avec lequel on ne peut communiquer que par l'intermédiaire de canaux de communications (type `chan`) (chap. 7).
3. Un module utilisable par l'intermédiaire de RPC et utilisant des sémaphores pour assurer l'exclusion mutuelle et les synchronisations conditionnelles (section 8.1)
4. Un module utilisant l'approche par rendez-vous (section 8.2).

Dans chaque cas, il s'agit simplement de définir un module qui gère *un unique* compte bancaire. Évidemment, dans chaque cas, vous devez aussi vous assurer que le comportement est correct en présence de requêtes multiples faites possiblement de façon concurrente.

Note importante: Si plusieurs retraits sont en attente et qu'un dépot est effectué, vous pouvez utiliser *n'importe quelle stratégie* pour sélectionner le ou les retraits qui seront satisfaits; vous n'avez donc pas à respecter l'ordre d'arrivée des retraits ni à tenter de satisfaire le plus grand nombre de retraits.

Note: Un bonus sera accordé si vous développez *correctement* un (1) autre module.

4. Processus concurrents et canaux de communications (10 pts)

Soit le processus `Trier`, dont l'interface et la logique générale sont présentées (en partie, sous forme de pseudo-code) plus bas. Ce processus reçoit en entrée une suite de clés (chaînes de caractères) et de nombres point-flottants (chaque clé est associée à un unique nombre) et produit en sortie la version triée de cette suite (triée en fonction de la clé):

```
chan a_trier(String cle, float val);
chan tries  (String cle, float val);

process Trier {
  String cle;
  float val;

  while (TRUE) {
    receive a_trier( cle, val );
    if (cle == EOS) break;
    ajouter_dans_liste( cle, val );  # On ajoute la cle dans la liste des elements a trier.

  }
  # On trie la liste des elements recus.
  trier_liste();
  # On obtient chaque des elements tries (en ordre) et
  # on les transmet sur le canal de sortie.
  while (liste_pas_vide()) {
    obtenir_prochain_liste( &cle, &val );
    send tries( cle, val )
  }
  send tries( EOS, EOS );
}
```

En utilisant ce processus et les canaux indiqués, écrivez, dans la notation d'Andrews, un programme concurrent (groupe de processus) utilisant uniquement des canaux de communication et permettant de résoudre le problème suivant:

- Le programme reçoit en entrée (sur le canal `entree`) une série de valeurs de la forme suivante: une chaîne de caractères indique le nom d'un employé, laquelle chaîne est suivie du nombre d'heures travaillées pour chacune des journées de la semaine (peuvent être `0.0`). Les noms ne sont pas nécessairement en ordre. De plus, il est possible que plusieurs items d'informations soient présents dans la suite pour un employé donné, et ce à des positions non-consécutives. La suite est terminée lorsqu'une fin de canal (`EOS`) est rencontrée (pour le nom de l'employé).
- Le programme produit en sortie une liste ordonnée des employés avec le nombre total d'heures travaillées durant la période traitée.

Exemple:

- Entrée:

```
EMP2  8.0  8.0  8.0  8.0  8.0
EMP1  7.0  7.0  8.0  9.0  0.0
EMP3  7.0  7.0  8.0  9.0  0.0
EMP1  0.0  0.0  6.0 10.0  6.0
EMP2  8.0  8.0  8.0  8.0  8.0
```

- Sortie:

```
EMP1  53.0
EMP2  80.0
EMP3  31.0
```

5. Canaux de communications (10 pts)

Soit le programme présenté à la figure 1 (p. 5). Écrivez *une* procédure Threaded-C qui va permettre de réaliser les processus `proc[i]` (pour `i` $\neq 0$).

Notes:

- Vous pouvez supposer que le type `ItemChan` est défini comme suit:

```
typedef union {
  int res;

  struct {
    int v1;
    int v2;
  } v1_v2;
} ItemChan;
```

- Vous *devez* aussi supposer que la procédure réalisant les processus `proc[i]` *recevra en argument* les canaux devant être utilisés par le processus (approche dynamique d'association des canaux). N'oubliez pas aussi que, dans la version Threaded-C, le processus maître qui crée ces divers processus doit pouvoir détecter le moment où tous les processus ont complété — un signal de terminaison additionnel est donc nécessaire.

Rappel: Les opérations de manipulation de canaux en Threaded-C ont les en-têtes suivantes:

```
void CHAN_INIT_SYNC   ( CHAN* chan, int max_items, SPTR chan_created );
void CHAN_SEND_SYNC   ( CHAN  chan, ItemChan item, SPTR slot );
void CHAN_SEND_EOC    ( CHAN  chan );
void CHAN_RECEIVE_SYNC( CHAN  chan, ItemChan* item, bool* eoc, SPTR slot );
void CHAN_DESTROY     ( CHAN  chan, SPTR slot );
```

Note: Lorsque la fin de canal est rencontrée (`EOS`), toutes les lectures subséquentes sur ce canal vont elles aussi lire (obtenir) la fin de canal.

[**Bonus**] Que fait ce programme? Quels rôles jouent `n` et `k`? Quelle(s) stratégie(s) de programmation parallèle est(sont) utilisée(s) dans ce programme? Expliquez/justifiez brièvement votre réponse.

```
chan in(int, int);
chan out(int);

process proc[0] {
  int n, k;
  int j;
  int res;

  scanf( "%d", &n );
  scanf( "%d", &k );
  assert( n % k == 0 );

  for [j = 0 to (n/k)-1]
    send in(j*k+1, k);
  send in(EOS, EOS);

  res = 1;
  for [j = 0 to (n/k)-1] {
    receive out(v)
    res *= v;
  }
  printf( "res = %d\n", res );
}

process proc[i = 1 to N] {
  int n;
  int v1, v2;
  int res;

  receive in( v1, v2 );
  while ( v1 != EOS ) {
    res = 1;
    for [j = v1 to v1+v2-1]
      res *= j;
    send out(res);
    receive in( v1, v2 );
  }
}
```

Figure 1: Programme mystère