Les bizarreries de l’isolation des IPC

Rachid Koucha
[Ingénieur développement logiciel]

Le namespace IPC (Inter-Process Communication) isole les mécanismes de communication inter-processus (sémaphores, mémoire partagée et queues de messages) hérités d’Unix Système V et bizarrement seulement une partie de leurs pendants POSIX.

Table of Contents

Avant-propos

Le code source des exemples utilisés dans cet article sont disponibles sur Github : https://github.com/Rachid-Koucha/linux_ns.

Cet article a été publié dans GNU Linux Magazine France n°250 du mois juillet 2021 :

Introduction

Bien que délaissés au profit de leurs pendants POSIX, les IPC Système V (sémaphores, mémoire partagée et queues de messages) sont encore très utilisés dans les applications. Ils ont la particularité de ne pas être identifiés dans le système de fichiers. Ils ont leur propre mécanisme d'identification à base de clé qui fait qu'ils ne peuvent pas être isolés dans un nouveau namespace mount (mount_ns) par exemple. C'est pour cela qu'un namespace dédié s'est imposé : le namespace IPC (ipc_ns).

Les IPC POSIX (**man 7 mq_overview**, **man 7 sem_overview** et **man 7 shm_overview**) s'appuient sur le système de fichiers (de type **mqueuefs** monté sur **/dev/mqueue** pour le premier et de type **tmpfs** monté sur **/dev/shm** pour les deux autres). **Ils peuvent donc être isolés dans un mount_ns**. Cependant, les queues de messages POSIX ont en commun avec les IPC système V, la disponibilité de fichiers dédiés dans **PROCFS** (cf. **man 7 namespaces** au § « IPC namespaces ») :

- **/proc/sys/fs/mqueue** : fichiers associés aux queues de messages POSIX ;

- **/proc/sys/kernel** et **/proc/sysvipc** : fichiers associés aux IPC système V.

Il est nécessaire de virtualiser ces fichiers dans la mesure où ils doivent retourner des informations en fonction du namespace depuis lequel ils sont consultés. D'où la présence des queues de messages POSIX dans l'ipc_ns et l'absence des deux autres.

1 PROCFS

Les entrées dédiées aux IPC dans le système de fichiers **/proc** étant virtualisées, leur contenu est différent d'un ipc_ns à l'autre. Si nous lançons notre programme **sem** dans un terminal et que nous consultons le fichier **/proc/sysvipc/sem**, on voit une entrée associée au sémaphore créé avec la clé 0x12345 (74565 en décimal) :

```
$ ./sem &
[1] 4982
Semaphore id is: 1
$ cat /proc/sysvipc/sem
       key      semid perms      nsems   uid   gid  cuid  cgid      otime      ctime
     74565          1     0          1  1000  1000  1000  1000          0 1583516593
```

Dans un conteneur LXC **busybox** qui a son propre ipc_ns et mount_ns pour le montage de **/proc**, le même fichier est vide car le sémaphore a été créé dans l'ipc_ns initial et non pas celui du conteneur :

```
bbox# cat /proc/sysvipc/sem
       key      semid perms      nsems   uid   gid  cuid  cgid      otime      ctime
```

Nous pouvons aussi mettre en œuvre des configurations différentes d'un ipc_ns à l'autre vu que les paramètres sont exportés et modifiables via **/proc**.

Pour les queues de messages POSIX, les paramètres sont dans **/proc/sys/fs/mqueue** : **msg_default**, **msg_max**, **msgsize_default**, **msgsize_max** et **queues_max**. Ils sont documentés dans **man 7 mq_overview**.

Pour les IPC système V (queues de messages, segments de mémoire partagée et sémaphores), les paramètres sont dans **/proc/sys/kernel** : **msgmax**, **msgmni**, **msgmnb**, **shm_rmid_forced**, **shmall**, **shmmax**, **shmmni** et **sem**. Ils sont documentés dans **man 5 proc**.

Dans un conteneur LXC **busybox**, changeons la valeur de **/proc/sys/kernel/msgmax** pour modifier la taille maximale des messages postés dans les queues de messages système V :

```
# lxc-console -n bbox -t 0
bbox# cat /proc/sys/kernel/msgmax
8192
bbox# echo 1024 > /proc/sys/kernel/msgmax
/bin/sh: can't create /proc/sys/kernel/msgmax: Read-only file system
```

La modification est impossible car le conteneur monte **/proc/sys** en lecture seule par défaut. C'est imposé par le paramètre de configuration **lxc.mount.auto** :

```
# cat /var/lib/lxc/bbox/config
[...]
lxc.mount.auto = cgroup:mixed proc:mixed sys:mixed
[...]
```

La valeur **proc:mixed** signifie certes que **/proc** est monté en lecture/écriture mais il indique aussi que **/proc/sys** est monté en **lecture seule pour des raisons de sécurité**. En effet, ce répertoire contient des données globales et critiques pour le système. Les paramètres des IPC sont donc virtualisés mais sont localisés dans l'arborescence **/proc/sys** qui contient une pléthore de paramètres globaux (non virtualisés). En d'autres termes, c'est un lieu sensible pour la sécurité et la stabilité du système. Pour redéfinir ces paramètres dans un conteneur LXC de type **busybox**, il faut donc modifier sa configuration en changeant **lxc.mount.auto** afin d'autoriser les lectures/écritures dans toute l'arborescence **/proc**. Il faut toutefois garder à l'esprit que la sécurité du système hôte est ainsi mise à mal car les paramètres critiques localisés dans **/proc/sys** deviennent modifiables à partir du conteneur. Voici donc la nouvelle version du fichier **/var/lib/lxc/bbox/config** :

```
# lxc.mount.auto = cgroup:mixed proc:mixed sys:mixed
lxc.mount.auto = cgroup:mixed proc:rw sys:mixed
```

Relançons le conteneur afin qu'il prenne en compte la nouvelle configuration et tentons à nouveau la modification du paramètre **msgmax** :

```
# lxc-stop -n bbox
# ./lxc-start2 bbox
# lxc-console -n bbox -t 0
[...]
bbox# cat /proc/sys/kernel/msgmax
8192
bbox# echo 1024 > /proc/sys/kernel/msgmax
bbox# cat /proc/sys/kernel/msgmax
1024
```

La modification est maintenant autorisée dans le conteneur. Et bien entendu, côté namespace initial (c.-à-d. système hôte), ce fichier n'est pas modifié :

```
$ cat /proc/sys/kernel/msgmax
8192
```

2 Pérennité des identifiants d'IPC

Les manuels n'abordent pas la problématique du changement de namespace lorsque des identifiants d'IPC sont déjà créés et en cours d'utilisation. Nous allons aborder le sujet de manière empirique pour chaque type d'IPC.

2.1 Pérennité des sémaphores

Les références sur les sémaphores **[1]** sont détruites lorsqu'un processus change d'ipc_ns. Il convient de prendre les précautions d'usage pour éviter les risques d'interblocage **[2]**. En effet, si nous considérons le cas classique d'un processus#1 qui prend le contrôle d'un sémaphore (opération **P()**) puis change de ipc_ns (en appelant **unshare()** ou **setns()**) sans libérer le sémaphore (c.-à-d. sans l'opération **V()**) alors qu'un processus#2 dans le namespace de départ attend la libération du sémaphore sur une opération **P()**. Ce dernier restera en attente infinie car le processus#1, en possession d'un identifiant de sémaphore caduc, n'aura plus la possibilité de le libérer (cf. figure 1).

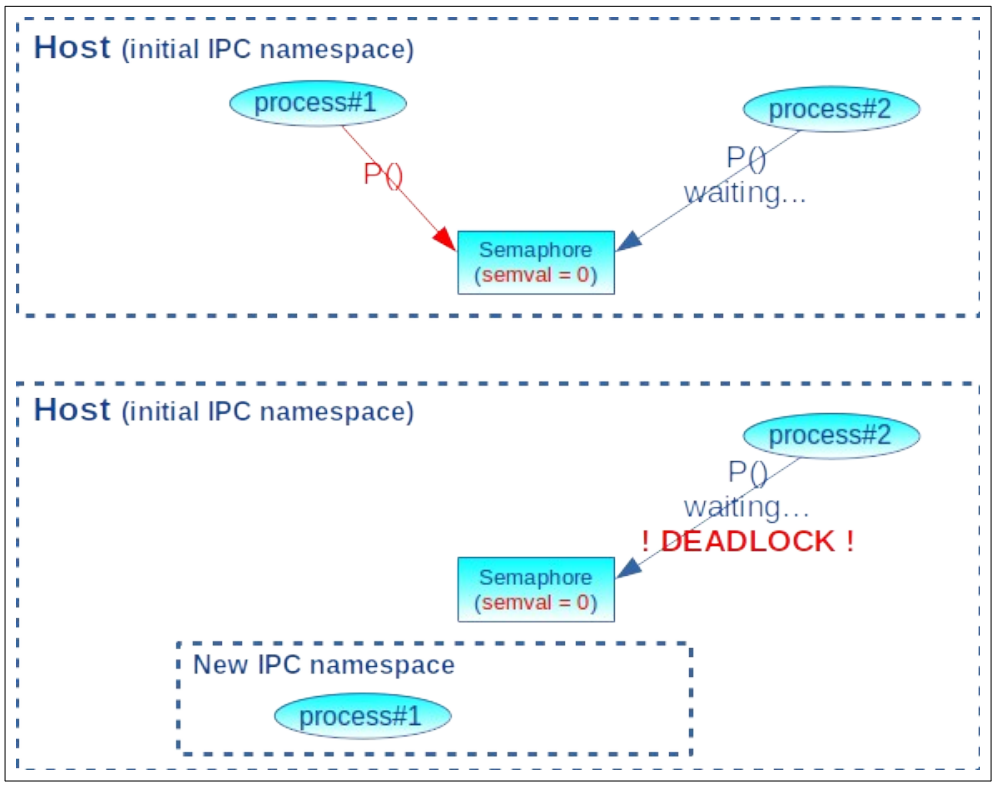

Fig. 1 : Interblocage entre processus sur changement de namespace

Par conséquent, avant un changement d'ipc_ns, le recours à **semadj** s'applique aussi (cf. l'encadré sur « l'ajustement de sémaphore ») ! Le principe consiste à utiliser le drapeau **SEM_UNDO** lors des incrémentations/décrémentations (cf. **man 2 semop**) des sémaphores afin de déclencher dans le noyau une annulation des opérations en cas de disparition inopinée des processus ayant la main dessus.

L'ajustement de sémaphore

L'appel système **semop()** incrémente ou décrémente la valeur **semval** d'un sémaphore. Un champ **semadj** est associé **à chaque processus pour chaque sémaphore**. Lorsque la valeur **semop** passée à **semop()** avec le drapeau **SEM_UNDO** est :

- inférieur à 0 (p. ex. opération **P()** qui fait -1), le champ **semval** est décrémenté de la valeur de **semop** et le champ **semadj** est incrémenté de la valeur absolue de **semop** si **semval** est supérieur ou égal à **semop**. Sinon le processus appelant est mis en attente jusqu'à ce que **semval** soit supérieur ou égal à **semop** ;

- supérieur à 0 (p. ex. opération **V()** qui fait +1), le champ **semval** est incrémenté de la valeur de **semop** et le champ **semadj** est décrémenté de la valeur de **semop**.

En cas de disparition d'un processus avant de lâcher la main sur un sémaphore (p. ex. **semval** = 0), le noyau ajoute la valeur du **semadj** à son champ **semval**. Cela à pour effet d'inverser les opérations effectuées (p. ex. **semval** += 1 si le processus disparaît entre l'appel à **P()** et **V()**) et par conséquent de lâcher la main sur le sémaphore.

Dans le programme d'exemple **semns**, un processus père crée un sémaphore et effectue l'opération **P()** pour prendre la main dessus. Ensuite il engendre un processus fils et attend un premier message de sa part. Ce dernier envoie le message lorsqu'il a récupéré un identifiant sur le sémaphore. Sur réception du message, le père attend une réponse de l'opérateur pour savoir s'il doit appeler **unshare()** (réponse « Y ») ou **V()** (réponse « N »). Cela donne le temps au fils d'effectuer l'opération **P()** et de se mettre en attente vu que le père à la main sur le sémaphore. Après mise en œuvre de l'opération choisie par l'opérateur, le processus père attend un second message de la part du fils. Ce dernier n'envoie ce message que lorsqu'il obtient le sémaphore (c.-à-d. lorsque l'opération **P()** donne la main). Le père effectue enfin un dernière opération **P()**

avant d'attendre la fin du processus fils avec **`waitpid()`**. Le mécanisme **`semadj`** est activé au niveau des opérations sur le sémaphore si le paramètre « 1 » est passé au programme. Les grandes lignes du code source sont les suivantes :

```
[...]
int main(int ac, char *av[])
{
[...]
  switch(av[1][0]) {
    case '0': undo = 0; break;
    case '1': undo = SEM_UNDO; break;
    default : usage(av[0]); return 1;
  } // End switch

  name = "Father";

  // Create the synchronization pipe
  // sync[0] = read end
  // sync[1] = write end
  rc = pipe(sync);
[...]
  // Get a semaphore identifier with initialization to 1
  id = semV_init();
[...]
  // Father locks semaphore
  rc = P(id);
[...]
  // Create a child process
  pid = fork();
  if (pid == 0) {

    // Child process

    name = "Child";
[...]
    // Get an identifier on the semaphore
    id = semV_get();
[...]
    printf("%s write 1st message for father\n", name);
    c = 0;
    rc = write(sync[1], &c, sizeof(c));
[...]
    // Lock the semaphore
    rc = P(id);
[...]
    printf("%s write 2nd message for father\n", name);
    c = 1;
    rc = write(sync[1], &c, sizeof(c));
[...]
    // Unlock the semaphore
    rc = V(id);
[...]
    printf("Child exits\n");

    exit(0);

  } else {

    // Father process
[...]
    printf("%s reading 1st message from child...\n", name);
    rc = read(sync[0], &c, sizeof(c));
    printf("%s read 1st message from child\n", name);
[...]
    prompt("Father unshares ([Y]/N) ? ");
    c = getanswer();
    if ('\n' == c || 'y' == c || 'Y' == c) {
      printf("Father enters in new IPC namespace...\n");
      rc = unshare(CLONE_NEWIPC);
[...]
    } else {

      rc = V(id);
[...]
    }

    printf("%s reading 2nd message from child...\n", name);
```

```
    rc = read(sync[0], &c, sizeof(c));
    printf("%s read 2nd message from child\n", name);
[...]
    rc = P(id);
[...]
    // Wait for the end of the child
    rc = waitpid(pid, 0, 0);
  }
[...]
  return 0;
} // main
```

Commençons par exécuter le programme dans les conditions idéales c'est-à-dire lorsqu'il n'y a pas création de nouvel ipc_ns (réponse « N » de l'opérateur). On peut utiliser ou non le drapeau **SEM_UNDO** (paramètre égal à 1 ou 0), tout se passe correctement car le processus père libère le sémaphore puis le fils en prend le contrôle avant de le libérer à son tour. Puis le père en reprend finalement le contrôle :

```
# ./semns 0
Father is initializing semaphore
Father got id of semaphore#4
Father: LOCKING semaphore#4 (SEM_UNDO OFF)...
Father: LOCKED semaphore#4
Father reading 1st message from child...
Child got id of semaphore#4
Child write 1st message for father
Child: LOCKING semaphore#4 (SEM_UNDO OFF)...
Father read 1st message from child
Father unshares ([Y]/N) ? N
Father: UNLOCKING semaphore#4 (SEM_UNDO OFF)...
Father: UNLOCKED semaphore#4...
Father reading 2nd message from child...
Child: LOCKED semaphore#4
Child write 2nd message for father
Child: UNLOCKING semaphore#4 (SEM_UNDO OFF)...
Father read 2nd message from child
Child: UNLOCKED semaphore#4...
Child exits
Father: LOCKING semaphore#4 (SEM_UNDO OFF)...
Father: LOCKED semaphore#4
```

La même exécution mais en provoquant l'appel à **unshare()** dans le père alors qu'il a la main sur le sémaphore (réponse « Y » de l'opérateur et paramètre à 0 pour ne pas utiliser **SEM_UNDO**) :

```
# ./semns 0
Father is initializing semaphore
Father got id of semaphore#5
Father: LOCKING semaphore#5 (SEM_UNDO OFF)...
Father: LOCKED semaphore#5
Father reading 1st message from child...
Child got id of semaphore#5
Child write 1st message for father
Child: LOCKING semaphore#5 (SEM_UNDO OFF)… # Child is blocked
Father read 1st message from child
Father unshares ([Y]/N) ? Y
Father enters in new IPC namespace...
Father reading 2nd message from child… # Father does not receive any message (child is blocked)
```

Le processus fils n'obtient jamais le sémaphore. Nous sommes dans une situation d'interblocage : en changeant d'ipc_ns, le processus père laisse le sémaphore bloqué. Et il reste bloqué sur la réception du second message du fils qui ne viendra jamais.

Effectuons le même test avec l'appel à **unshare()** mais avec en plus l'utilisation du drapeau **SEM_UNDO** (paramètre 1 passé au programme) :

```
# ./semns 1
Father is initializing semaphore
Father got id of semaphore#6
Father: LOCKING semaphore#6 (SEM_UNDO ON)...
Father: LOCKED semaphore#6
Father reading 1st message from child...
Child got id of semaphore#6
Child write 1st message for father
Child: LOCKING semaphore#6 (SEM_UNDO ON)...
Father read 1st message from child
Father unshares ([Y]/N) ? Y
Father enters in new IPC namespace…   # semadj mechanism is triggered: semaphore is unlocked
Father reading 2nd message from child...
Child: LOCKED semaphore#6
```

```
Child write 2nd message for father
Child: UNLOCKING semaphore#6 (SEM_UNDO ON)...
Father read 2nd message from child
Child: UNLOCKED semaphore#6...
Father: LOCKING semaphore#6 (SEM_UNDO ON)...
Child exits
ERROR@P#107: semop(): 'Invalid argument' (22)
```

L'entrée du processus père dans le nouvel ipc_ns libère bien le sémaphore grâce au mécanisme **semadj** déclenché dans le noyau au moment du changement de namespace : l'opération **P()** du processus père est annulée de sorte à rendre la main sur le sémaphore (cf. figure 2). Le processus fils n'est plus bloqué. Mais bien-entendu, le dernier **P()** effectué par le père dans son nouveau namespace se solde par une erreur **EINVAL** car l'identifiant de sémaphore est caduc dans le nouveau namespace.

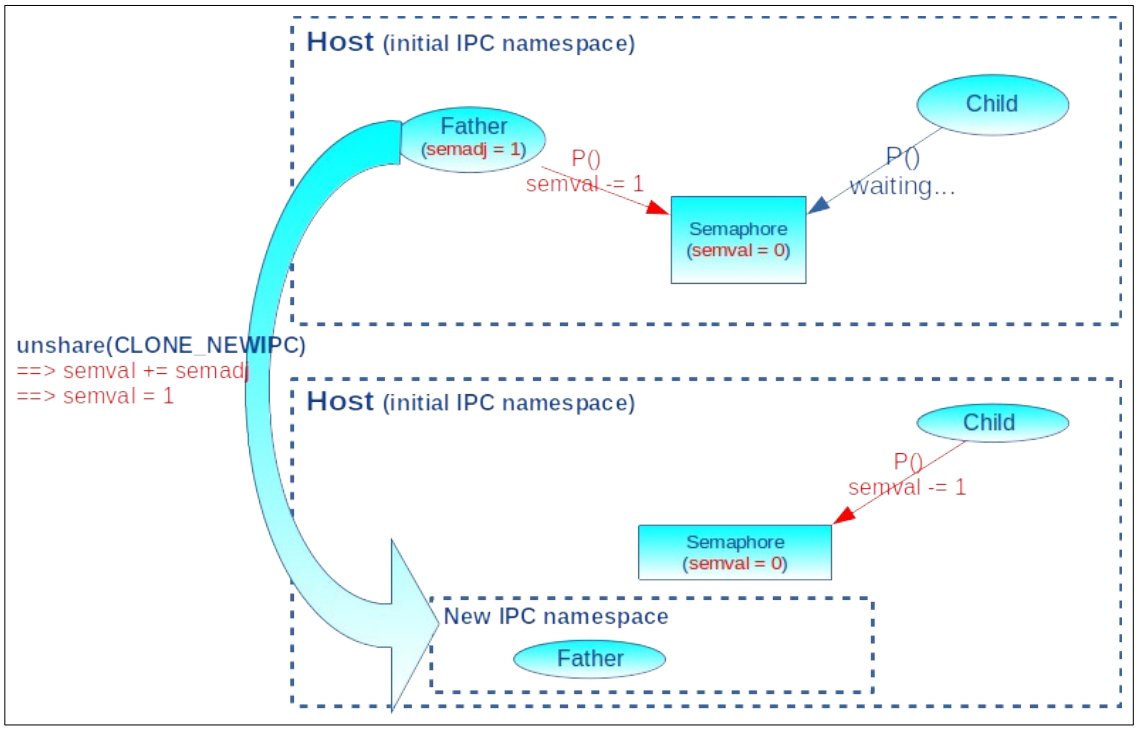

*Fig. 2 : Mécanisme **semadj** sur changement d'ipc_ns*

2.2 Pérennité de la mémoire partagée

Au même titre que les sémaphores, les identifiants de segment de mémoire partagée sont caducs après un changement d'ipc_ns. Le programme **shmns** suivant appelle **shmget()** pour créer un segment de mémoire partagée puis attend une réponse de l'opérateur pour appeler ou non **unshare()** avant d'attacher le segment avec **shmat()**.

```
int main(void)
{
[...]
  // Get an identifier on a shared memory segment
  id = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT|IPC_EXCL|0777);
[...]
  prompt("Unshare before shmat() ([Y]/N) ? ");
  c = getanswer();
  if (c == 'Y' || c == 'y' || c == '\n') {
    rc = unshare(CLONE_NEWIPC);
[...]
  }

  p = shmat(id, 0, 0);
```

L'exécution du programme avec l'appel à **unshare()** se solde par une erreur **EINVAL** sur l'appel **shmat()** car l'identifiant du segment mémoire est invalide dans le nouvel ipc_ns :

```
# ./shmns
Unshare before shmat() ([Y]/N) ? Y
ERROR@main#56: shmat(): 'Invalid argument' (22)
```

Par contre, le changement de namespace après l'attachement du segment mémoire à l'espace d'adressage du processus ne pose pas de problème. On se retrouve dans une situation où le processus référence un espace de mémoire partagée créé dans un autre ipc_ns. Toute mise à jour dans cet espace mémoire sera vu de l'ipc_ns de départ et du nouveau.

Les programmes d'exemples **`producer`** et **`consumer`** (tous deux basés sur le même fichier source **`prodcons.c`**) mettent en pratique la possibilité d'échange de données via un segment de mémoire partagée entre deux processus appartenant à des ipc_ns différents. Au départ les deux processus sont dans l'ipc_ns initial. Ils capturent tous deux le signal **`SIGINT`**. Le programme **`producer`** est lancé en premier pour créer le segment de mémoire partagée avec la clé 0x12345. Ensuite il attend une réponse de l'opérateur pour changer d'ipc_ns. Si, la réponse est négative, il se contente de mettre à jour la date dans l'espace mémoire. Il répète ces opérations en boucle jusqu'à une réponse positive de l'opérateur. Lorsque la réponse est positive, il appelle **`unshare(CLONE_NEWIPC)`** et entre dans une boucle qui met à jour la date dans l'espace mémoire à chaque réponse de l'opérateur. Une fois lancé, le programme **`consumer`** s'attache au segment de mémoire partagée et lit son contenu toutes les secondes pour l'afficher à l'écran. Pour la synchronisation inter-processus de sorte à ce que **`producer`** n'altère pas la mémoire lorsque **`consumer`** est en train de la lire (c.-à-d. l'exclusion mutuelle), il n'est pas possible d'utiliser un sémaphore système V car **`producer`** fait un changement d'ipc_ns et on a vu plus haut que cela rendrait ce sémaphore caduc. Les programmes utilisent donc un sémaphore POSIX localisé dans la zone de mémoire partagée. La figure 3 illustre l'architecture du programme.

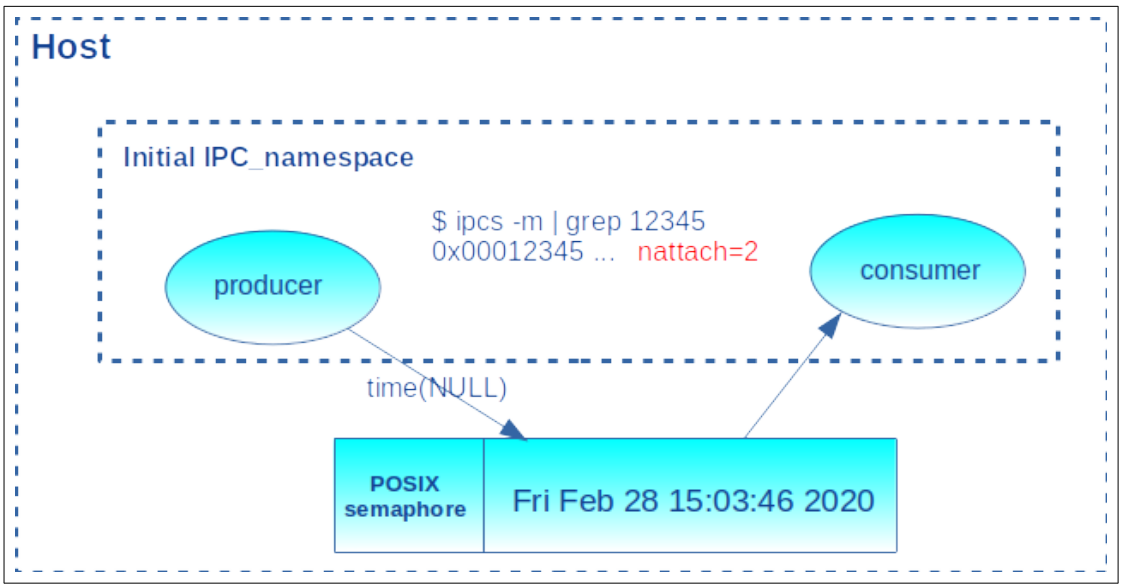

Fig. 3 : Producteur/consommateur sur un segment de mémoire partagée

Le code de **`producer`** se présente comme suit :

```
  if (!strcmp(name, "producer")) {

    int already_unshared = 0;

    /*
     * Producer
     */
[...]
    // Get an identifier on a shared memory segment
    idm = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT|IPC_EXCL|0777);
[...]
    // Attach the memory segment to the virtual address space
    pShm = shmat(idm, 0, 0777);
[...]
    printf("%s: Shared memory segment attached at: %p\n", name, pShm);
```

```
    // Semaphore creation and initialization to 1
    rc = sem_init((sem_t *)pShm, 1, 1);
[...]
    // Data zone (right after the semaphore)
    pData = (char *)(((sem_t *)pShm) + 1);
    data_sz = (pShm + SHM_SIZE) - pData;

    // Reset the zone
    memset(pData, 0, data_sz);

    // Infinite loop
    while (1) {

      if (!already_unshared) {
        prompt("Unshare ipc namespaces ([Y]/N) ? ");
        c = getanswer();
        if (c == 'Y' || c == 'y' || c == '\n') {
          rc = unshare(CLONE_NEWIPC);
[...]
          already_unshared = 1;
        }
      } else {
        prompt("Update time ([Y]/N) ? ");
        c = getanswer();
        if (c != 'Y' && c != 'y' && c != '\n') {
          continue;
        }
      }

      rc = P((sem_t *)pShm);
[...]
      // Write current time in the shared memory segment
      t = time(NULL);
      ctime_r(&t, pData);

      // Remove terminating '\n'
      *(pData + strlen(pData) - 1) = '\0';

      rc = V((sem_t *)pShm);
[...]
    } // End while

  }
```

Le code de **consumer** se présente comme suit :

```
    /*
     * Consumer
     */

    // Get an identifier on a shared memory segment
    idm = shmget(SHM_KEY, SHM_SIZE, 0);
[...]
    // Attach the memory segment to the virtual address space
    pShm = shmat(idm, 0, 0777);
[...]
    printf("%s: Shared memory segment attached at %p\n", name, pShm);

    // Data zone (right after the semaphore)
    pData = (char *)(((sem_t *)pShm) + 1);
    data_sz = (pShm + SHM_SIZE) - pData;

    // Infinite loop
    while (1) {

      sleep(1);

      rc = P((sem_t *)pShm);
[...]
      // If there is something to read in the shared memory segment
      if (pData[0]) {
        printf(" %s\r", pData);
        fflush(stdout);

        // Reset the zone
        memset(pData, 0, data_sz);
      }
```

```
      rc = V((sem_t *)pShm);
[...]
    } // End while
```

Le gestionnaire du signal **SIGINT** détache le segment de mémoire partagée et pour le producteur seulement, il détruit le segment :

```
static void sig_hdl(int s)
{
  int rc;

  printf("\n\n%s: Signal %d!\n\n", name, s);

  if (pShm) {
    rc = shmdt(pShm);
    if (rc != 0) {
      ERR("shmdt(%p): '%m' (%d)\n", pShm, errno);
      exit(1);
    }
  }

  // If I am the producer ==> Unlink the message queue
  if (!strcmp(name, PRODUCER_NAME)) {

    if (idm != -1) {
      rc = shmctl(idm, IPC_RMID, 0);
      if (rc < 0) {
        ERR("shmctl(%d, IPC_RMID): '%m' (%d)\n", idm, errno);
        exit(1);
      }
    }
  }

  exit(0);

} // sig_hdl
```

Dans un premier terminal, exécutons **producer** et entrons un premier « N » à la question de sorte à lui faire écrire la date une première fois dans le segment mémoire :

```
# ./producer
producer#19872 is starting...
producer: Shared memory segment attached at: 0x7f136e2f5000
Unshare ipc namespaces ([Y]/N) ? N
Unshare ipc namespaces ([Y]/N) ?
```

Dans un second terminal, exécutons **consumer**. Toutes les secondes il affichera une nouvelle heure si celle-ci est modifiée dans le segment mémoire :

```
# ./consumer
consumer#19887 is starting...
consumer: Shared memory segment attached at 0x7f050a232000
 Fri Feb 28 14:57:18 2020
```

La commande **ipcs** dans un troisième terminal affiche bien deux références (colonne **nattch**) sur le segment de mémoire partagée :

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
[...]
0x00012345 262207     root       777        4096       2
```

Répondons « N » un certains nombre de fois à **producer** pour provoquer la mise à jour de la date dans le segment mémoire et constatons que **consumer** met à jour son affichage en conséquence. Puis répondons « Y » à **producer** pour le faire changer d'ipc_ns. On constate que les mises à jour de l'heure sont toujours vues par **consumer** à chaque fois qu'on répond « Y » à **producer**.

Dans le troisième terminal, le nombre de références sur l'espace mémoire reste égal à deux. Par contre, si on entre dans l'ipc_ns de **producer** (son pid est affiché au démarrage), la commande **ipcs** n'affiche rien car le segment de mémoire n'ayant pas été créé dans ce namespace, il n'a aucune identification. D'ailleurs il n'y a aucune autre ressource IPC non plus :

```
# ipcs -m | grep 12345   # Value of the shared memory key
0x00012345 262207     root       777        4096       2
# nsenter -t 19872 -i    # 19872 is the pid of producer
# ipcs
```

```
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

La figure 4 schématise tout cela.

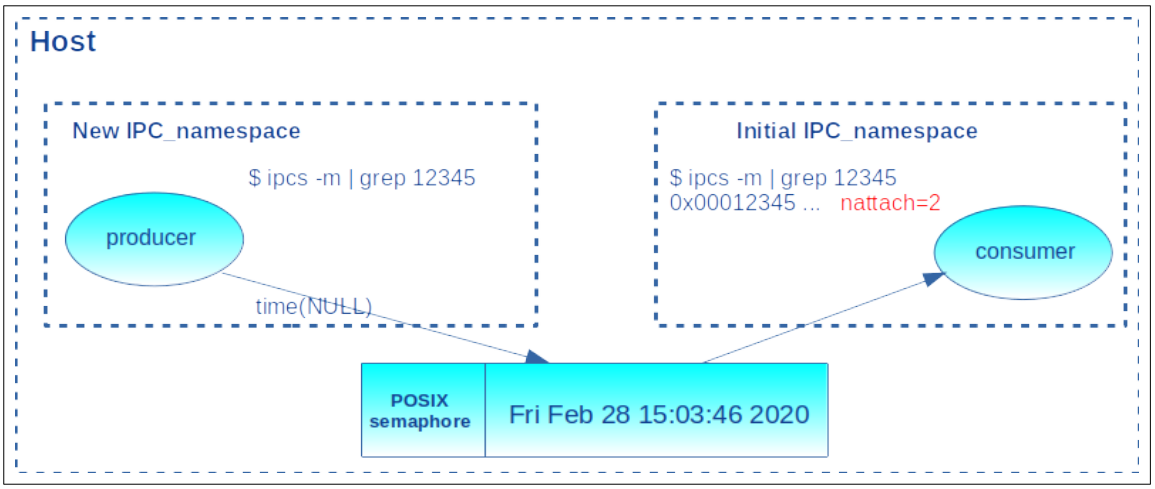

Fig. 4 : Producteur/consommateur dans différents ipc_ns

En résumé, nous avons montré la possibilité de partage d'un segment de mémoire entre des processus localisés dans des ipc_ns différents.

Par contre, si on arrête le producteur avec un **<CTRL>** + **<C>** :

```
Update time ([Y]/N) ? ^C

producer: Signal 2!

ERROR@sig_hdl#59: shmctl(262207, IPC_RMID): 'Invalid argument' (22)
```

Le gestionnaire du signal **SIGINT** détache le segment mémoire en passant son adresse à **shmdt()**. Cela se passe correctement. Par contre il enchaîne sur la destruction de l'identifiant du segment en le passant à **shmctl()**. L'identifiant étant caduc dans le nouvel ipc_ns, cela se traduit pas une erreur **EINVAL**. La destruction doit donc se faire dans l'ipc_ns où la création a eu lieu. On montre ainsi un éventuel problème de conception de notre application d'exemple : le créateur de la ressource IPC devrait rester dans l'ipc_ns de création afin de faire le nettoyage lors de sa terminaison. En d'autres termes, c'est le consommateur qui aurait du changer d'ipc_ns et non pas le producteur. Notre conception avait juste un but pédagogique.

Le détachement du segment dans le gestionnaire de signal réduit bien le nombre de références de 2 à 1 dans l'ipc_ns de départ :

```
# ipcs -m | grep 12345   # Value of the shared memory key
0x00012345 262207     root       777        4096       1
```

L'arrêt du consommateur fait passer le compteur à 0 :

```
# ipcs -m | grep 12345   # Value of the shared memory key
0x00012345 262207     root       777        4096       0
```

Du au problème de conception susmentionné, nous nous retrouvons avec un identifiant de segment de mémoire partagée orphelin que le producteur n'a pas pu détruire lors de sa terminaison. Une mesure d'administration système est alors nécessaire pour faire le nettoyage via un appel à « **ipcrm -M** » ou en positionnant le drapeau **/proc/sys/kernel/shm_rmid_forced** à 1 de sorte à provoquer la destruction automatique des segments n'ayant plus de référence (c.-à-d. ayant un compteur d'attachements à 0).

2.3 Pérennité des queues de messages Système V

Tout comme les sémaphores et segments de mémoire partagée, les identifiants de queues de messages sont caducs après un changement d'ipc_ns. Les programmes **producer2** et **consumer2** sont respectivement les adaptations de **producer** et **consumer** vus précédemment mais au lieu d'utiliser un segment de mémoire partagée et de l'exclusion mutuelle, ils utilisent une queue de messages pour communiquer la date.

Le code source de **producer2** est donc beaucoup plus simple :

```
  if (!strcmp(name, "producer2")) {

    int already_unshared = 0;

    /*
     * Producer
     */
[...]
    // Get an identifier on a message queue
    idq = msgget(MSG_KEY, IPC_CREAT|IPC_EXCL|0777);
[...]
    // Infinite loop
    while (1) {

      if (!already_unshared) {
        prompt("Unshare ipc namespaces ([Y]/N) ? ");
        c = getanswer();
        if (c == 'Y' || c == 'y' || c == '\n') {
          rc = unshare(CLONE_NEWIPC);
[...]
          already_unshared = 1;
        }
      } else {
        prompt("Update time ([Y]/N) ? ");
        c = getanswer();
        if (c != 'Y' && c != 'y' && c != '\n') {
          continue;
        }
      }

      // Write current time in the message
      msg.mtype = 1;
      t = time(NULL);
      ctime_r(&t, msg.date);

      // Remove terminating '\n'
      msg.date[strlen(msg.date) - 1] = '\0';

      // Send the message
      rc = msgsnd(idq, &msg, sizeof(msg) - sizeof(msg.mtype), 0);
[...]
    } // End while

  }
```

Le code source de **consumer2** est encore plus simple :

```
    /*
     * Consumer
     */

    // Get an identifier on the message queue
    idq = msgget(MSG_KEY, 0);
    if (idq < 0) {
      ERR("msgget(): '%m' (%d)\n", errno);
      return 1;
    }

    // Infinite loop
    while (1) {

      // Receive the message
      rc = msgrcv(idq, &msg, sizeof(msg) - sizeof(msg.mtype), 1, 0);
[...]
      // Print the date
      printf(" %s\r", msg.date);
```

```
      fflush(stdout);

    } // End while
```

A l'exécution, **producer2** plantera dès qu'on l'autorisara à changer d'ipc_ns car l'envoi du message dans la queue nécessite son identifiant qui est caduc :

```
./producer2
producer2#32586 is starting...
Unshare ipc namespaces ([Y]/N) ? n
Unshare ipc namespaces ([Y]/N) ? n
Unshare ipc namespaces ([Y]/N) ? n
Unshare ipc namespaces ([Y]/N) ? n
Unshare ipc namespaces ([Y]/N) ? y
ERROR@main#106: msgsnd(): 'Invalid argument' (22)
```

Et dans ce cas, **consumer2** se retrouve à attendre indéfiniement un message qui ne viendra plus :

```
# ./consumer2
consumer2#32638 is starting...
 Fri Feb 28 20:10:41 2020
```

En conclusion, il est impossible d'échanger des messages par queues IPC entre applications associées à des ipc_ns différents car les identifiants nécessaires aux appels systèmes sont caducs lorsqu'un processus change de namespace.

2.4 Pérennité des queues de messages POSIX

Les queues de message POSIX offrent le même service que les queues de messages Système V mais comme souligné en introduction de cet article, elles offrent une interface standardisée plus récente (cf. **man 7 mq_overview**). Le mécanisme s'appuie sur un système de fichiers virtuel dédié nommé **mqueuefs** généralement monté sur le répertoire **/dev/mqueue**. Alors que les sémaphores et segments de mémoire partagée POSIX se basent sur un système de fichiers **tmpfs** :

```
$ cat /proc/$$/mountinfo | egrep "mqueue|shm"
29 24 0:24 / /dev/shm rw,nosuid,nodev shared:4 - tmpfs tmpfs rw
50 24 0:19 / /dev/mqueue rw,nosuid,nodev,noexec,relatime shared:29 - mqueue mqueue rw
$ ls -l /dev/ | grep mqueue
drwxrwxrwt  2 root   root          40 mars    6 16:52 mqueue
```

Le répertoire **/dev/mqueue** étant partagé par tous les utilisateurs du système, à l'image de **/tmp**, le bit « t » (sticky bit) dans la copie d'écran ci-dessus permet d'empêcher l'effacement des fichiers par des utilisateurs autres que ceux qui les ont créés (et bien entendu le super utilisateur et le propriétaire du répertoire).

Les programmes **producer3** et **consumer3** sont respectivement les adaptations de **producer2** et **consumer2** vus précédemment mais où les queues de messages système V sont remplacées par des queues de messages POSIX pour la communication de la date. Inutile donc de reproduire le code source ici, seuls les appels système ont changé.

Contrairement à l'exécution de **producer2** qui plantait après changement d'ipc_ns pour cause d'utilisation d'un identifiant de queue de message caduc, **producer3** continue à s'exécuter correctement dans les mêmes conditions :

```
# ./producer3
producer3#7786 is starting...
Unshare ipc namespaces ([Y]/N) ? N
Unshare ipc namespaces ([Y]/N) ? N
Unshare ipc namespaces ([Y]/N) ? Y
Update time ([Y]/N) ? Y
Update time ([Y]/N) ? Y
```

Le manuel précise que ce qui se cache derrière un objet **mqd_t** n'est autre qu'un descripteur de fichier qui reste ouvert même après appel à **unshare()**. Cela explique que la queue de messages reste utilisable après changement d'ipc_ns car à partir de cette information, le noyau remonte au système de fichiers où se trouve la queue de message. **Mais ce n'est pas un comportement portable car la norme POSIX n'impose pas l'implémentation de l'identifiant de queue** (**mqd_t**) **en descripteur de fichier**. D'ailleurs dans le manuel, l'utilisation des appels système normalement dédiés aux descripteurs de fichiers sur les queues de messages est évoquée en ces termes pour préciser que la portabilité n'est pas garantie :

« Linux implementation of message queue descriptors

*On Linux, a message queue descriptor is actually a file descriptor. (***POSIX does not require such an implementation***.) This means that a message queue descriptor can be monitored using select(2), poll(2), or epoll(7).* ***This is not portable****. »*

Si on arrête le producteur avec un **<CTRL>** + **<C>** :

```
Update time ([Y]/N) ? ^C

producer3: Signal 2!

ERROR@sig_hdl#61: mq_unlink(/qdate): 'No such file or directory' (2)
```

Le gestionnaire du signal va appeler **mq_close()** pour fermer le descripteur de queue puis **mq_unlink()** pour l'effacer. Le dernier appel se traduit par une erreur **ENOENT** pour indiquer que la queue de messages n'existe pas. A première vue, c'est assez incohérent comme comportement vu que le fichier associé (c.-à-d. le nom passé à **mq_open()**) existe bien dans l'arborescence :

```
$ ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars    9 19:27 qdate
```

Mais nous avons vu lors de l'étude du code source du noyau **[3]** que l'appel système **unshare()** provoque le montage d'un nouveau système de fichiers **mqueuefs** dans le nouvel ipc_ns. Le champ **mq_mnt** de la structure ipc_namespace est mis à jour en conséquence. C'est par ce champ que la fonction **mq_unlink()** accède au système de fichiers. Voici le code source correspondant dans le noyau (Linux version 5.3.0) où l'on voit que la recherche de l'entrée de répertoire associée au nom de fichier est effectuée à partir du pointeur **ipc_ns->mq_mnt** :

```
SYSCALL_DEFINE1(mq_unlink, const char __user *, u_name)
{
        int err;
        struct filename *name;
        struct dentry *dentry;
        struct inode *inode = NULL;
        struct ipc_namespace *ipc_ns = current->nsproxy->ipc_ns;
        struct vfsmount *mnt = ipc_ns->mq_mnt;

        name = getname(u_name);
[...]
        audit_inode_parent_hidden(name, mnt->mnt_root);
        err = mnt_want_write(mnt);
        if (err)
                goto out_name;
        inode_lock_nested(d_inode(mnt->mnt_root), I_MUTEX_PARENT);
        dentry = lookup_one_len(name->name, mnt->mnt_root, strlen(name->name));
[...]
        inode = d_inode(dentry);
        if (!inode) {
                err = -ENOENT;
        } else {
                ihold(inode);
                err = vfs_unlink(d_inode(dentry->d_parent), dentry, NULL);
        }
        dput(dentry);

[...]
        return err;
}
```

La fonction ne va donc pas chercher le fichier au bon endroit : elle utilise le nouveau point de montage au lieu de l'ancien.

Mais nous ne sommes au bout de nos interrogations. Considérons le programme **pmsg** qui crée une queue de messages POSIX (**/pmsg**) dans l'ipc_ns courant, change d'ipc_ns puis crée une seconde queue de messages (**/pmsg1**). A chaque étape, l'exécutable attend l'accord de l'utilisateur et affiche le contenu du répertoire **/dev/mqueue** :

```
# ./pmsg
pmsg#14533 is starting...
pmsg: Created message queue /pmsg

# ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars   16 14:16 pmsg
```

La première queue de message apparaît bien dans le répertoire **/dev/mqueue**.

```
Unshare ipc_ns ?

# ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars   16 14:16 pmsg
```

Après création d'un nouvel ipc_ns, le programme voit toujours la queue de message dans le répertoire **/dev/mqueue**. Pourtant on sait après étude des sources dans le noyau que **unshare()** a provoqué un montage interne d'un nouveau système de fichiers **mqueuefs** dédié au nouvel ipc_ns. Continuons en créant la seconde queue de messages :

```
Create msq queue '/pmsg1' ?
pmsg: Created message queue /pmsg1

# ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars   16 14:16 pmsg
```

Comme indiqué dans **[4]**, les appels système **mq_*** vont effectivement « pointer » sur ce nouveau montage interne au noyau, mais l'utilisateur, en appelant « ls -l /dev/mqueue » continue à voir l'ancien montage car il passe par le point de montage du « Virtual File System » (VFS) dans le mount_ns courant sur lequel on a monté le **mqueuefs** de l'ipc_ns initial. C'est pour cela qu'on continue à voir **pmsg** mais on ne voit pas **pmsg1** qui est dans le nouveau système de fichiers monté en interne après **unshare()**. Créons un nouveau mount_ns afin de monter nouveau système de fichiers **mqueuefs** créé dans le nouvel ipc_ns :

```
Unshare mount_ns ?

# ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars   16 14:16 pmsg

Mount /dev/mqueue (mount --make-rslave /; mount -t mqueue mqueue /dev/mqueue) ?
# ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars   16 14:17 pmsg1
```

Dans ce nouveau mount_ns au sein duquel nous avons monté le système de fichiers **mqueuefs** associé au nouvel ipc_ns, la commande **ls** voit désormais les queues de messages créées dans ce dernier. Donc on ne voit plus que **pmsg1**. Sans arrêter le programme, dans un autre terminal nous voyons **pmsg** dans le système de fichiers associé à l'ipc_ns initial :

```
$ ls -l /dev/mqueue
total 0
-rwxr-xr-x 1 root root 80 mars   16 14:16 pmsg
```

Nous pouvons enfin arrêter le programme **pmsg** :

```
End ?
```

Nous avons ainsi mis en exergue le nécessaire remontage du système de fichiers **mqueuefs** lorsqu'il y a changement d'ipc_ns sous peine de ne plus être en phase sur le contenu du répertoire **/dev/mqueue**.

3 Isolation des segments de mémoire partagée et sémaphores POSIX

Contrairement à leurs pendants système V, les segments de mémoire partagée et sémaphores POSIX ne sont pas concernés par les ipc_ns car leur implémentation étant basée sur un système de fichiers de type **tmpfs** monté sur **/dev/shm** et n'exportant pas de données dans **/proc**, il est suffisant de monter ce dernier dans un nouveau mount_ns pour l'isolation.

Considérons les programmes **producer4** et **consumer4**. Ils sont basés sur le même principe que **producer** et **consumer**. Le premier met à jour la date dans un segment de mémoire partagée et le second lit la date toutes les secondes pour l'afficher à l'écran. Le segment de mémoire partagée ainsi que le sémaphore utilisé pour l'exclusion mutuelle sont basés sur les services POSIX et sont respectivement nommés « /mdate » et /sdate »

Lançons **producer4** dans un terminal (il accepte l'option **-d** pour effacer tout segment ou sémaphore résiduel suite à une éventuelle exécution précédente) :

```
$ ./producer4 -d
producer4#12391 is starting...
```

Dans un autre terminal, affichons le contenu de **/dev/shm**, répertoire où les segments de mémoire partagée et sémaphores POSIX sont gérés. On voit bien les deux IPC créés :

```
$ ls -l /dev/shm
total 8
-rwxr-xr-x 1 rachid rachid 4096 mars   17 14:19 mdate
-rwxr-xr-x 1 rachid rachid   32 mars   17 14:19 sem.sdate
```

Lançons ensuite **consumer4** dans un autre terminal pour constater que la date est mise à jour toutes les secondes :

```
$ ./consumer4
consumer4#12399 is starting...
 Tue Mar 17 14:21:12 2020
```

Si nous lançons de nouveau **producer4** (sans l'option **-d** !) mais dans un autre terminal, le programme se termine en erreur car les IPC existent déjà vu qu'il y a déjà une première instance du programme qui tourne dans le même mount_ns :

```
$ ./producer4
producer4#12432 is starting...
/mdate already exists ?!?
```

Dans ce terminal, lançons donc un shell dans un nouveau mount_ns. Puis, comme préconisé dans l'article dédié aux mount_ns **[5]**, passons l'arborescence en mode **SLAVE** pour enfin monter un nouveau système de fichier **tmpfs** sur **/dev/shm**. Il faut bien-sûr être super utilisateur pour ces opérations :

```
# unshare -m
# mount --make-rslave /
# mount -t tmpfs tmpfs /dev/shm
# ls -l /dev/shm
total 0
```

Puis dans ce même terminal relançons **producer4**. On constate qu'il démarre correctement sans perturber la précédente instance en cours d'exécution dans les namespaces initiaux :

```
# ./producer4
producer4#12509 is starting...
```

Dans un autre terminal, lançons un shell dans le mount_ns nouvellement créé (on utilise l'identifiant de processus affiché par **producer4**). Le contenu de **/dev/shm** reflète bien la création des deux nouveaux IPC :

```
# nsenter -m -t 12509
# ls -l /dev/shm
total 8
-rwxr-xr-x 1 root root 4096 mars   17 14:31 mdate
-rwxr-xr-x 1 root root   32 mars   17 14:31 sem.sdate
```

Enfin lançons **consumer4** dans ce dernier terminal pour constater qu'il fonctionne tout aussi bien :

```
# ./consumer4
consumer4#12559 is starting...
 Tue Mar 17 14:38:23 2020
```

Nous avons ainsi mis en évidence que les segments de mémoire partagée et sémaphores POSIX s'isolent grâce aux mount_ns comme schématisé dans la figure 5.

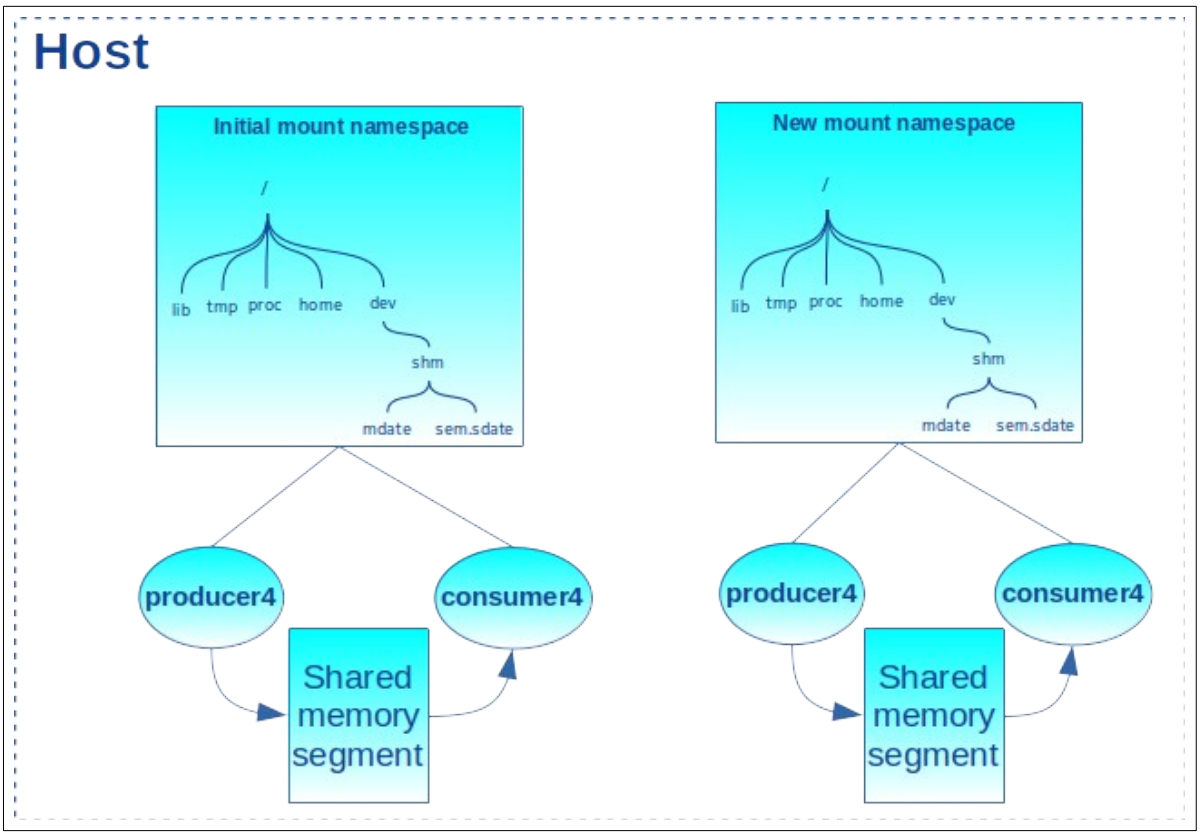

Fig. 5 : Isolation des segments de mémoire/sémaphores POSIX dans des mount_ns

4 Les IPC dans LXC

La commande **lxc-execute** lance un programme dans un conteneur. Ce dernier est créé et supervisé par le programme d'**init** du conteneur qui, préalablement à la création du processus, monte les systèmes de fichiers **tmpfs** et **mqueuefs** respectivement sur **/dev/shm** et **/dev/mqueue** dans l'ipc_ns spécifique au conteneur.

La configuration par défaut d'un conteneur de type **busybox** ne monte qu'un système de fichiers de type **tmpfs** sur **/dev/shm** :

```
# cat /var/lib/lxc/bbox/config
[...]
lxc.mount.entry = shm /dev/shm tmpfs defaults 0 0
[...]
```

Mais lorsque nous consultons le répertoire **/dev** ou lorsqu'on affiche le contenu du fichier **mountinfo**, on ne voit rien concernant shm :

```
bbox# cat /proc/1/mountinfo | grep shm
bbox# ls -l /dev/shm
ls: /dev/shm: No such file or directory
```

En fait, avec LXC 2.1.1, le template **lxc-busybox** a quelques manques signalés puis corrigés par l'auteur de cet article. Il est donc conseillé de créer les conteneurs avec la version modifiée du template qui accompagne cet article ou la dernière mouture sur le site **https://github.com/lxc/lxc**.

Après destruction et création du conteneur avec le template amélioré, la configuration devient :

```
# cat /var/lib/lxc/bbox/config
[...]
lxc.mount.entry = shm dev/shm tmpfs defaults,create=dir 0 0
lxc.mount.entry = mqueue dev/mqueue mqueue defaults,optional,create=dir 0 0
[...]
```

Et après démarrage, les systèmes de fichiers suivants sont montés dans le conteneur :

```
bbox# cat /proc/1/mountinfo | egrep "shm|mqueue"
470 461 0:61 / /dev/shm rw,relatime - tmpfs shm rw
471 461 0:48 / /dev/mqueue rw,relatime - mqueue mqueue rw
```

Conclusion

Nous avons pu voir que le namespace IPC ne gère pas tous les IPC mais seulement les IPC système V à cause de leur système d'identification particulier et les queues de messages POSIX à cause de l'export d'information dans **PROCFS**.

Contrairement à toute attente, c'est le namespace mount qui permet d'isoler les sémaphores et segments de mémoire partagée POSIX.

Références

[1] La notion de sémaphore : https://fr.wikipedia.org/wiki/S%C3%A9maphore_(informatique)

[2] Les interblocages : https://fr.wikipedia.org/wiki/Interblocage

[3] R. Koucha, « Les structures de données des namespaces dans le noyau », Gnu/Linux magazine n°243, décembre 2020

[4] Implement support for posix msqueues : https://lore.kernel.org/patchwork/patch/141212/

[5] R. Koucha, « A la découverte des namespaces mount et uts », Gnu/Linux magazine n°247, avril 2021