Rapport PR204

Romain Benquet, Nicolas Bousquet

INTRODUCTION

Dans ce rapport, nous allons présenter brièvement notre version du projet de PR204. Le but de ce rapport ne sera pas d'expliquer tout le projet, mais plutôt de mettre l'accent sur certains choix qui ont été faits dans notre code afin de les expliquer. Ainsi, certaines parties ne seront pas développées car les commentaires présents dans le code suffisent à les comprendre.

Dsmexec

Le programme dsmexec a plusieurs missions, il doit réaliser:

- La lecture du fichier *machine_file*
- Le lancement de dsmwrap
- La redirection des sorties *stderr* et *stdout*
- La réception des informations des processus distants
    - Subtilité: lorsque le processus principal reçoit les informations des processus distants, il réinitialise tout le tableau *PROC_ARRAY* (y compris le nom de la machine qui avait été rempli préalablement pour lancer les connexions ssh), ainsi, il est certain d'associer les bonnes informations aux bonnes machines, même quand deux processus s'exécutent sur la même machine.

dsmexec.c (int main(int argc, char argv[])*

```
for(i = 0; i < num_procs ; i++){
  /* on accepte les connexions des processus dsm */
   if ((connfd = accept(sfd, (struct sockaddr*) &server_addr, &_len)) < 0) {
       perror("accept()\n");
       exit(EXIT_FAILURE);
   }
   /*On recupere le fd*/
   proc_array[i].connect_info.fd = connfd;

   /*  On recupere le nom de la machine distante */
  /* 1- d'abord la taille de la chaine */
  int size;
  recv(connfd, &size, sizeof(int), MSG_WAITALL);
```

```
  /* 2- puis la chaine elle-même */
  char name[size*sizeof(char)];
  recv(connfd, name, size, MSG_WAITALL);
  memset(proc_array[i].connect_info.machine, 0, MAX_STR);
  strncpy(proc_array[i].connect_info.machine, name, size);
  /* On récupère le pid du processus distant  (optionnel)*/
  pid_t dist_pid;
  recv(connfd, &dist_pid, sizeof(pid_t), MSG_WAITALL);
  proc_array[i].pid = dist_pid;
  /* On récupère le numéro de port de la socket */
  /* d'écoute des processus distants */
  unsigned int dist_listening_port;
  recv(connfd, &dist_listening_port, sizeof(unsigned int), MSG_WAITALL);
  proc_array[i].connect_info.port_num = dist_listening_port;
  //on affecte un rang à chaque processus distant
  proc_array[i].connect_info.rank = i;
}
```

- Le partage des informations de chaque processus distant à tous les autres processus distants
- La lecture dans les tubes et l'affichage de leur contenu en console
    - Subtilité: afin de lier les tubes aux processus distants, ces derniers envoient leur PID dans le tube. Ainsi, on peut facilement savoir dans quel tube écrit chaque processus distant. Les attributs *pipe_fd_out* et *pipe_fd_err* ont été ajoutés dans la structure *dsm_proc_t*.

common_impl.h

```
 struct dsm_proc {
 pid_t pid;
 int pipe_fd_out; //file descriptor du tube associé à SDTOUT
 int pipe_fd_err; //file descriptor du tube associé à SDTERR
 dsm_proc_conn_t connect_info;
};
```

dsmexec.c (int main(int argc, char argv[])*

```
pid_t pid_in_pipe = 0;
memset(buff, 0, MAX_STR);
for(i=0; i<num_procs; i++){
   //on récupère le pid de chaque processus distant envoyé dans les tubes stdout
  read(pollfds_out[i].fd, &pid_in_pipe, sizeof(pid_t));
  for(int j=0; j<num_procs; j++){
      //on lie chaque tube à chaque processus distant
      if(proc_array[j].pid == pid_in_pipe){
          proc_array[j].pipe_fd_out = pollfds_out[i].fd;
```

```
            proc_array[j].pipe_fd_err = pollfds_err[i].fd;
        }
    }
}
```

- La gestion des processus zombies
    - Lors de la remise de la phase 1, notre programme recevait l'erreur *Interrupted system call*. Elle était causée par l'absence du flag *SA_RESTART* dans le traitant gérant les zombies. Cette erreur a depuis été corrigée et notre programme marche aussi bien avec notre phase 1 qu'avec la votre.

Dsmwrap

Le programme dsmwrap doit lui:

- Communiquer ses informations avec dsmexec
- Créer une socket d'écoute pour pouvoir communiquer avec les autres processus distants
- Lancer le programme final

Pas de subtilité particulière dans ce programme, la démarche "classique" a été suivie pour le réaliser.

Exemple

Le programme exemple a pour but d'essayer d'accéder à des zones de la mémoire virtuelle partagée. Cependant, il peut arriver qu'il essaie d'accéder à une zone de cette mémoire à laquelle il n'a pas accès, provoquant alors une erreur de segmentation.

Dsm.c

Le fichier *dsm.c* va:

- Connecter tous les processus distants entre eux
    - Subtilité: deux tableaux de descripteurs de fichiers sont créés lors de la connexion aux autres processus distants, il y a le tableau *PROC_ARRAY* qui contient tous les descripteurs (dans le paramètre *fd* du *PROC_ARRAY)* issus des *connect*, et le tableau *POLLFDS* qui contient tous les descripteurs issus des *accept*. Il y donc deux sockets de créées entre chaque processus distant.

PROC_ARRAY sera utilisé pour faire envoyer des requêtes de page à un processus précis, car il lie le rang (*DSM_NODE_ID*) du processus distant à son *fd*. Il permet donc d'envoyer une demande de page à son propriétaire.

À l'inverse, *POLLFDS* ne lie pas les descripteurs aux rangs des processus. Il sera donc utilisé dans le thread d'écoute des requêtes.

- Initialiser l'allocation des pages de la mémoire partagée à chaque processus (en tourniquet)
- Gérer les erreurs de segmentation provoquée par le programme exemple et synchroniser la mémoire partagée entre tous les processus distants
    - Cette partie ne comporte paradoxalement pas vraiment de subtilité. La seule étant peut-être l'utilisation d'un sémaphore pour s'assurer que le processus demandant une page ne sorte pas du traitant de signal avant d'avoir reçu la page.

dsm.c (`static void dsm_handler(void *addr))`

```
static void dsm_handler(void *addr)
{
 //on gère les erreurs de segmentation
  printf("[%i] Page mémoire inaccessible \n",DSM_NODE_ID);
  //on récupère le numéro de la page qui a provoqué l'erreur
  int numpage = address2num(addr);
  //on récupère le propriétaire de la page
  dsm_page_owner_t owner = get_owner(numpage);
  //on envoie une demande de page au propriétaire
  dsm_req_t req = {0, 0, 0};
  req.source = DSM_NODE_ID;
  req.page_num = numpage;
  req.type = DSM_REQ;
  send(PROC_ARRAY[owner].fd, &req, sizeof(dsm_req_t),0);
  printf("[%i] Page numéro %i demandée au processus %i\n", DSM_NODE_ID, numpage, owner);
  //le processus reste bloqué le temps que la page ait été envoyée
  sem_wait(&semaphore);
}
```

dsm.c (`static void* dsm_comm_daemon(void *arg))`

```
else if(req.type == DSM_PAGE){ //si le processus reçoit une page
   printf("[%i] Page numéro %i reçue du processus %i\n", DSM_NODE_ID, req.page_num,
req.source);
   //le processus s'alloue la page
```

```
    dsm_alloc_page(req.page_num);
    //le processus reçoit la page
    char* page = num2address(req.page_num);
    recv(POLLFDS[i].fd, (void*)page, PAGE_SIZE, MSG_WAITALL);
    //le processus se met propriétaire de la page dans la table des pages
    dsm_change_info(req.page_num, WRITE, DSM_NODE_ID);
    printf("[%i] Je suis propriétaire de la page %i\n", DSM_NODE_ID, req.page_num);
    //le processus envoie à tous les autres processus distants qu'il est maintenant
propriétaire de la page
    msg.page_num = req.page_num;
    msg.source = DSM_NODE_ID;
    msg.type = DSM_NREQ;
    for(int j=0; j<DSM_NODE_NUM; j++){
        if(j!=DSM_NODE_ID){
            send(PROC_ARRAY[j].fd, &msg, sizeof(dsm_req_t), 0);
        }
    }
    //le processus met un jeton dans le sémaphore pour que la fonction dsm_handler() puisse
se terminer
    sem_post(&semaphore);
}
```

- La terminaison des processus distants
    - Lorsqu'un processus a terminé le programme qu'il devait exécuter, il envoie à tous les processus distants qu'il a terminé. Il attend ensuite que les autres processus distants aient également terminé d'exécuter le programme. Lorsque c'est le cas, tous les processus détruisent alors leur thread d'écoute puis les processus se terminent.

dsm.c (void dsm_finalize(void))

```
//le processus envoie à tous les autres processus distants qu'il est arrivé à la fin du
programme
 dsm_req_t req = {0, 0, 0};
 req.type = DSM_FINALIZE;

 for(int i=0; i<DSM_NODE_NUM; i++){
    if(PROC_ARRAY[i].rank!=DSM_NODE_ID){
        send(PROC_ARRAY[i].fd, &req, sizeof(dsm_req_t), 0);
    }
 }
 //le processus attend que tous les autres processus en soit au même stade que lui pour tuer
 le thread d'écoute
 int toto=0;
 while (PROCS_FINALIZED!=DSM_NODE_NUM){ //PROCS_FINALIZED = variable globale qui comptent le
```

```
nombre de processus terminés
   toto++;
}
```

dsm.c (static void dsm_comm_daemon(void* arg))*

```
else if(req.type == DSM_FINALIZE){ //si le processus reçoit qu'un processus est arrivé à la
fin de exemple.c
   PROCS_FINALIZED++;
}
```

Cas où plusieurs processus distants essaient d'accéder à une même page en même temps.

Lorsque plusieurs processus distants essaient de se connecter à la même page en même temps, ils envoient tous une demande de page à son propriétaire. Le processus propriétaire envoie la page à la première requête qu'il reçoit mais peut par la suite recevoir les requêtes des autres processus alors qu'il n'est plus propriétaire (les processus demandeurs ayant envoyé leur requête avant que leur table de pages (et donc de propriétaires) ne soit mise à jour). Le processus envoie alors à ces processus-là qu'il n'est plus propriétaire et précise le nouveau propriétaire de cette page. Les processus demandeurs pourront alors demander la page au bon propriétaire.

dsm.c (static void dsm_comm_daemon(void* arg))*

```
else{ //le processus n'est plus propriétaire de la page mais le processus demandeur n'a pas
encore mis à jour sa table (cas où plusieurs processus essaient d'accéder à la même page en
même temps)
   msg.page_num = req.page_num;
   //envoie le propriétaire de la page demandée
   msg.source = table_page[req.page_num].owner;
   msg.type = NEW_OWNER;
   //on envoie au processus le bon propriétaire de la page
   send(PROC_ARRAY[req.source].fd, &msg, sizeof(dsm_req_t), 0);
}
```

Un autre cas n'a cependant pas été géré, celui où deux processus s'échangent des pages en ping-pong sans qu'ils n'aient jamais le temps d'avancer dans l'exécution du programme.

Conclusion

Nous avons essayé de comprendre la philosophie globale du projet en utilisant un maximum les notions du cours. Les notions de tube, de socket, de recouvrement, de traitant de signal et de processus ont été utilisées. Si la plupart des notions ont été maîtrisées, nous avons

expérimenté à quel point la synchronisation des processus entre eux (en particulier lorsqu'ils sont distants) est difficile et peut mener à des bugs importants.