Serveurs

Table des matières

1 Introduction

Il s'agit des principaux démons embarqués. Leur rôle est de fournir une API pour les appels aux drivers.

Les principaux démons sont :

- Sam : Banc de test des mémoires analogiques
- Big : Interface principale
- Zora : Gestion des données
- Emilie : Gestion des triggers

2 Organisation des répertoires

2.1 Répertoires communs

Au plus haut niveau :

- `bin` : programme de haut niveau (reliquats de Hess-1)
- `CHANGES` : Changelog
- `Makefile`
- `Make.rules` : Compilation croisée
- `Make.macros` : "
- `Target.rules` : Compilation sur cible
- `Target.macros` : "

Répertoire `Server` :

- `Bigd`

- `Emilie`
- `Client` : simulation des commandes de base depuis la ferme de PC
- `include` : includes propores aux caméras et donc communs à tous les démons
- `Makefile`

2.2 Big

- `Big` : interpréteurs de commandes
- `Corba` : envoi des données aquises par la caméra
- `Drawer` : configuration des tirroirs
- `Farm` : représentation de la ferme de PC
- `include`
- `Scheduler` : lance les taches programmées
- `Server` : serveur de sockets
- `ServerUtil` : fonctions des points d'entrées (start/stop...)
- `Util` : misc ?
- `test` : tests
- `BIG-CHANGES` : change log
- `Makefile`

- `Client` : UI reprenant le répertoire *../Client*
- `Socket` : communication des résultats (out) avec un client TCL-TK

3 Compilation

- Les démons sont compilé en utilisant le compilateur croisé.
- Le chemin vers le compilateur croisé est indiqué dans le fichier *.shl_cross_compile* ; comme indiqué dans le fichier ier *Make.rules* :

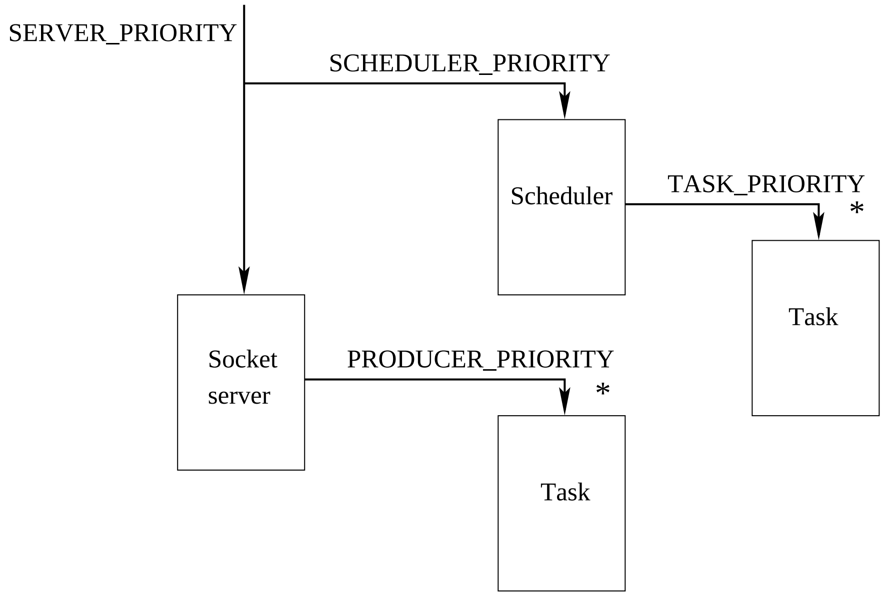
```
CROSS_COMPILE   := $(shell cat $(HH_HOME)/SBig/.shl_cross_compile)
```

- La compilation se fait sur le serveur de boot N1N9.

```
[roche@n1n9]~/SBig/Server% make
```

4 Fonctionnement des serveurs

Dans le diagramme ci-dessous, chaque flèche indique la création d'un thread. Leur priorité est inscrite à côté des flèches. Une étoile complète les flèches lorsque plusieurs threads sont créés.



Attention dans hess2 les priorités effectives sont inversées : `thread_prior_max - SERVER_PRIORITY` et donc plus elle sont faibles et plus elle sont prioritaires (comme pour nice).

Précision : le schéma ci dessus est celui des serveurs de monitoring (Big et Bunny). Les serveurs d'acquisition (Zora et Emilie) utilisent quand à eux un scheduleur différent générant uniquement un ou deux autres thread de priorités `SENDER_PRIORITY` (au lieu de `TASK_PRIORITY`).

- les threads privilégiés seront les plus brefs possibles.
- les serveurs de contrôle observe les priorités `SCHEDULER_PRIORITY < SERVER_PRIORITY`. Par conséquent le monitoring est prioritaire sur la réception et l'envoi des ordres.
- les serveurs de données observe les priorités `SERVER_PRIORITY < SCHEDULER_PRIORITY`. Par conséquent la réception des ordres est prioritaire sur l'acquisition.
- La relation `SERVER_PRIORITY < PRODUCER_PRIORITY` permet d'interrompre les requêtes "begin" et garanti le traitement atomique des requêtes "fast_".
- On devrait (je pense) avoir `TASK_PRIORITY < SCHEDULER_PRIORITY` car les tâches de monitoring s'excluent mutuellement et sont ordonnancés les unes à la suite des autres.
- `< SENDER_PRIORITY < SCHEDULER_PRIORITY` favorise l'envoi des donnée par les sockets au vidage des fifos. En effet (j'imagine) que les opérations d'entrées/sorties sur les fifo rendent la main d'elles mêmes en cas d'attente.
- Bien que définis, `SENDER_PRIORITY` et `CONSUMER_PRIORITY` ne sont pas utilisées.

Ectat actuel sur la caméra :

Priorité	Monitoring	Acquisition
Server	9	9
Producer	9	9
Scheduler	10	12
Task	11	
Sender		10, 11

Valeurs proposées pour le module de test :

Priorité	Monitoring	Acquisition
Server	8	8
Producer	9	9
Scheduler	7	12
Task	6	
Sender		10, 11

Voici un programme simplifié permettant de tester cette architecture. A l'édition des liens, il faut indiquer la librairie adéquate : `gcc -lpthread simuServer.c`.

4.1 Pseudo temps-réel

Sous linux, 3 ordonnancements co-existent.

- OTHER : préemptif avec priorité dynamique (nice)
- RR : temps réel souple (priorité statique) mais préemptif à priorité égale
- FIFO : temps réel souple non préhemptif (comme windows 95)

S'il existe un ou plusieurs processus FIFO ou RR, ils sont séléctionnés en premier. Lors de la phase de débugage, il est important de conserver un shell s'exécutant à un niveau de priorité supérieur. Attention lors du fork, il faut placer la modification de priorité dans le code du processus père.

4.2 Thread

Les threads sont des processus allégés ne réclamant que peu de ressources pour les changements de contexte. Chaque thread dispose personnellement d'une pile et d'un contexte d'exécution contenant les registres du processeur et un compteur d'instruction. En revanche, ils se partagent les données statiques et dynamiques.

Sous LINUX, l'implémentation usuelle des threads est effectuée dans l'espace noyau.

Remarque : un thread finissant envoie un signal au thread parent si celui-ci n'est pas trappe le thread ne meurt pas ... seul le systeme le fera disparaitre 1 mn plus tard. Ainsi, pour ne pas borner le nombre de threads lancés par le nombre de threads qu'il est possible de lancer en même temps, il faut soit utiliser la fonction `pthread_join` soit détacher les threads.

Comme pour les processus, on peut modifier l'ordonnancement des threads. Les ordonnancements temps-réel nécessitent un UID effectif nul, sinon la fonction `pthread_create` échouera avec l'erreur EPERM.

Les serveurs embarqués pour HESS utilisent 2 thread, le fils principal devenant alors le second thread :

```
/* privilèges root */
setuid(geteuid());

/* processus privilégié
 * attention, les tread créés sans l'attribut 'PTHREAD_EXPLICIT_SCHED'
 * hériteront de la nouvelle priorité */
paramSocketServer.sched_priority = thread_prior_max - SERVER_PRIORITY;
sched_setscheduler(0, __SCHEDULER_POLICY, &paramSocketServer)

/* construction du thread executant la fonction 'Scheduler' */
pthread_attr_setdetachstate(&threadAttributs, PTHREAD_CREATE_DETACHED)
pthread_attr_setschedpolicy(&threadAttributs, __SCHEDULER_POLICY)
pthread_attr_setinheritsched(&threadAttributs, PTHREAD_EXPLICIT_SCHED)
paramThreadScheduler.sched_priority =  thread_prior_max - SCHEDULER_PRIORITY
pthread_attr_setschedparam(&threadAttributs, &paramThreadScheduler)
pthread_create(&threadScheduler, &threadAttributs, Scheduler, NULL)

/* definition des thread qui éxecuteront la fonction process_message */
pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED)
pthread_attr_setschedpolicy(&thread_attr, __SCHEDULER_POLICY)
pthread_attr_setinheritsched(&thread_attr, PTHREAD_EXPLICIT_SCHED)
scheduler._parameters.sched_priority = thread_prior_max - BIG_SCHEDULER_PRIORITY
pthread_attr_setschedparam(&thread_attr, &scheduler._parameters)

/* rendre les privilèges */
setuid(realUserIdentifier);

/* lancement du serveur de socket */
tcp_server();
```

4.3 Socket

Les serveurs embarqués pour HESS utilisent un schéma séquentiel.

```
/* creation de la socket standardiste */
sock_contact = socket(AF_INET, SOCK_STREAM, 0)
bind(sock_contact, ...)
listen(sock_contact, ...)
```

```
/* connexion sequentielle aux clients */
while(loop) {
    sock_dialog = accept(sock_contact, ...)
    process_connection(sock_dialog)
```

Ce schéma sequentiel peut alors déboucher sur une prise en charge parallèle :

```
if (...)
{
  /* traitement sequentiel */
  ...
}
else
{
/* thread executant la fonction 'process_message' avec en paramètre la socket */
  pthread_create(&thread_producer[n], &producer__attr, process_message, (void *)sock)))
}
```

Remarque : `CLOSE_WAIT` is when the other endpoint have closed the connection and the kernel is vaiting for the application to confirm the close by closing the socket descriptor. There is no timeout for `CLOSE_WAIT`. It persists for as long as the application has not closed the socket.

```
# netstat -nap
Proto Recv-Q Send-Q Local Address               Foreign Address             State       PID/Progr
tcp       55      0 192.168.1.169:1806          134.158.155.234:53291       CLOSE_WAIT  -
```

5 Grammaire des controleurs

- Big

```
orders: SHUT_
      | PING_
      | preposition stanza ENDMSG
      | stanza ENDMSG

preposition: BEGIN
           | FAST_

stanza: order
      : stanza order

order: CAMERA camera
     | NODE farm
     | FIFO fifo
     | DRAWER drawer
     | DAEMON daemon
     | RUN run
     | TASK task
     | INI drawer2
     | TRG
     | ACK FIFODATA_EMPTY
     | ACK FIFODATA_PROBE_EVTSIZE
     | ???
     | ...
```

- Camera

```
camera: CONFIGURE
      | SAFE
      | RUN
      | STOP
      | FIFODATA_PROBE_EVTSIZE
```

- Farm

```
farm: CONTROLLER node
    | MONITORING node
    | ADD 0x%4X%n node
    | CURRENT %d
    | DELETE %d
    | DISABLE %d
    | ENABLE %d
    | FREQUENCY %d
    | LIST

node: %s %s %s%n { %x%n}+
```

- Fifo

```
fifo: RESET
```

```
    | RESET_INTERFACE
    | INIT_BOXBUS
    | RESET_WFIFO
    | RESET_RFIFO
```

- Drawer

  ```
  drawer: TARGET
        | SET_CNTRL_SLC
        | SET_CNTRL_DAQ
        | SET_CNTRL_L2
        | SET_CNTRL_SAM_DAQ
        | SET_CNTRL_SAM_ND
        | SET_CNTRL_SAM_REGISTER
        | SET_CNTRL_TRIGGER
        | SET_TRIGGER_DISABLE
        | SET_CNTRL_VOLTAGE
        | SET_HV_OFF
        | SET_VOLTAGE_VALUE
        | SET_VOLTAGE_LEVEL
  ```

- Daemon

  ```
  daemon: DEBUG %X
        | CLEARSCREEN
        | INIT_DAQ
  ```

- Run

  ```
  run: START
     | STOP
     | INIT
  ```

- Task

  ```
  task: STATUS
      | LIST
      | UNLOCK
      | ADD
      | SCHEDULER_DELAY %d
      | FREQUENCY %d
      | DELETE
      | ENABLE
      | DISABLE
  ```

- Drawer2

  ```
  drawer2: cmdId label ...
         |
         |
         |
         | ...
  ```