Systèmes d'exploitation (SYE)

Xavier Ruppen / Alexandre Malki / Junghyun Kim / Daniel Rossier / Alberto Dassatti / Salvatore Valenza

Solution - IPC

Echéance : Voir Moodle

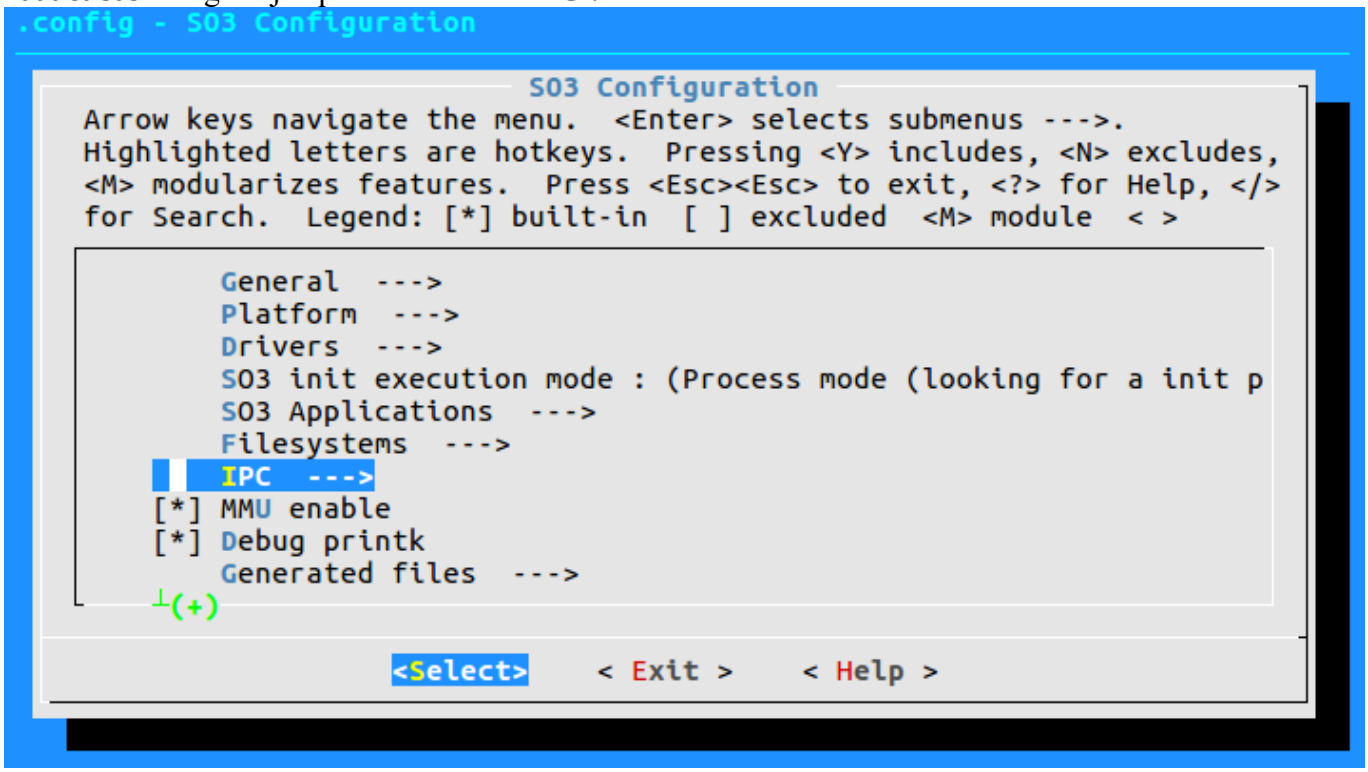
1. *Récupération des codes sources du laboratoire*

2. *Environnement de compilation (build system)*

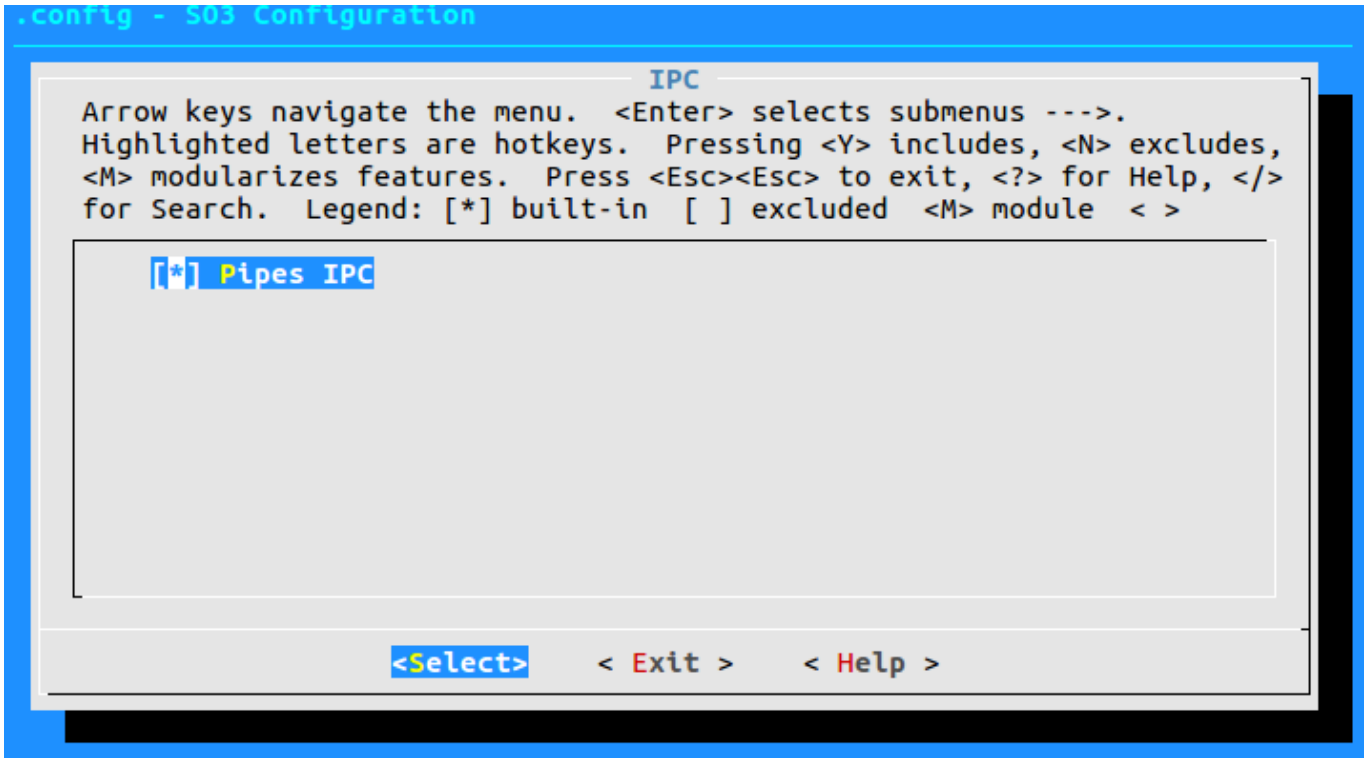
Pour activer le module pipe dans le kernel SO3 il faut tout d'abord taper les commandes suivantes depuis le dépôt:

```
$ cd so3
$ make vexpress_defconfig  # Nous nous basons sur la configuration vexpress de base
$ make menuconfig
```

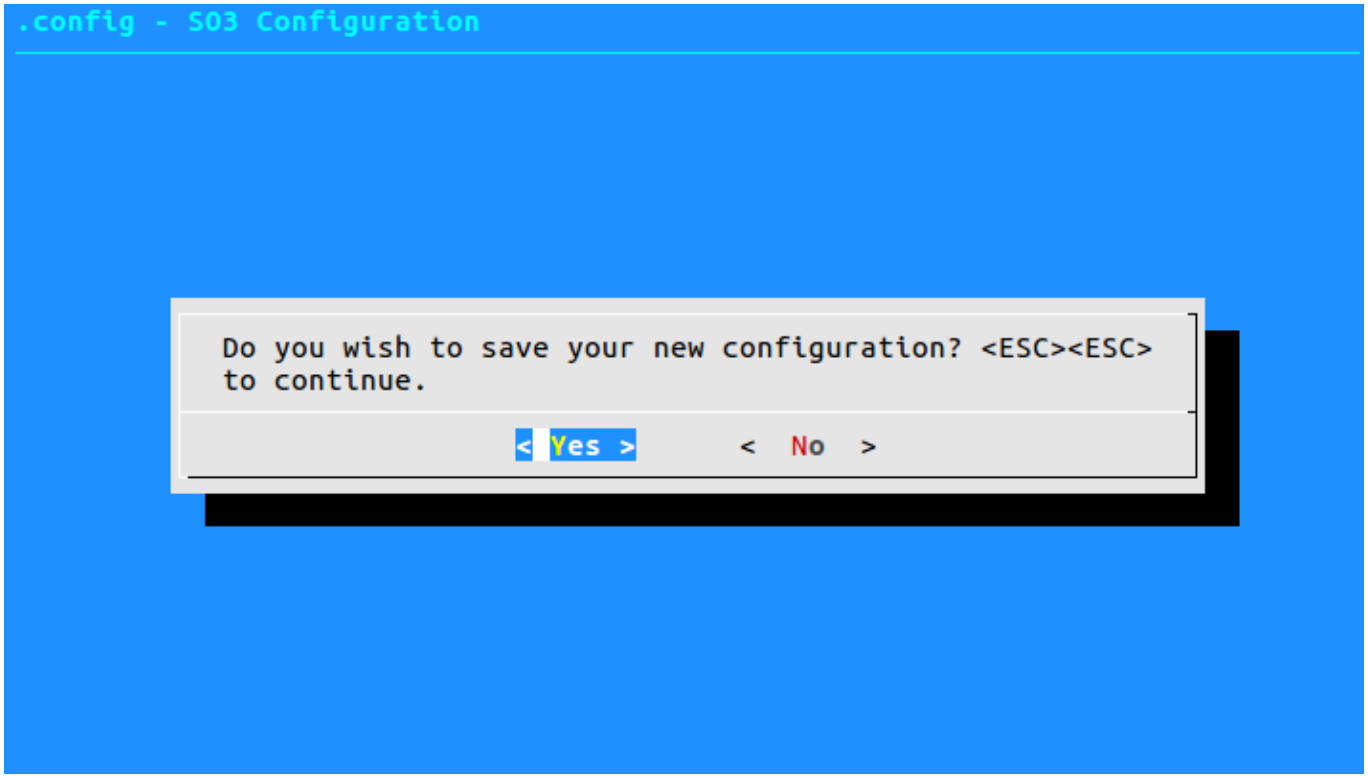
Les flèches multidirectionnelles haut et bas permettent de se déplacer dans le menu. Avec les flèches haut et bas naviguez jusqu'à selectionné **IPC :**



Puis appuyez sur « entrée » pour afficher le sous menu et appuyez sur « y » pour activer le module, un étoile apparaitre entre les crochets.



Puis selectionnez l'action **exit** en utilisant les fléches droite et gauche. Enfin tapez fois fois sur entrée. Repétez la manipulation précédente jusqu'à avoir la fenêtre ci-dessous.



Puis tapez yes pour enregister.

3. Appels système

L'ajout des appels système se fait à plusieurs endroits :

- Dans l'espace utilisateurs, il faut modifier les fichiers :
  - Dans le fichier *usr/libc/include/syscall.h* pour déclarer les fonctions et declarer le bon numéro d'appel système :

```
#define syscallThreadJoin                                17
#define syscallThreadExit                                18
#define syscallPipe                                      19
#define syscallDup2                                      23

#ifndef __ASSEMBLY__

#include <bits/alltypes.h>

#include <pthread.h>
#include <types.h>
#include <inet.h>

extern int errno;
( ~ ligne 284)
int dup2(int newfd, int oldfd);
int pipe(int fd[2]);
```

  - Dans le fichier *usr/libc/crt0.S* ajoutez a la fin du fichier les appels système avec la macro:

```
SYSCALLSTUB dup2,                                        syscallDup2
SYSCALLSTUB pipe,                                        syscallPipe
```

- Dans l'espace noyau, il faut modifier les fichiers :
  - Dans le fichier *so3/arch/arm/include/asm/syscall.h*, il faut ajouter le define des appels système dup2 et pipe.

```
 (ligne 59)
#define SYSCALL_PIPE                            19
#define SYSCALL_DUP2                            23
```

  - Dans le fichier *so3/kernel/syscalls.c*, il faut seulement décommenter les lignes 139 a 142 et lignes 149 à 152.

4. Espace noyau

Dans cette étape, vous devez écrire le code de l'appel système liés aux *pipe*. Dans le fichier so3/ipc/pipe.c, vous devez modifier le contenu de la fonction do_pipe. A partir de la ligne 294 :

```
int do_pipe(int pipefd[2]) {

        /* Allocated two file descriptor */
        pipe_desc_t *pd = (struct pipe_desc *) malloc(sizeof(pipe_desc_t));
        if (!pd) {
                printk("%s: heap overflow...\n", __func__);
                kernel_panic();
```

```
	}

	memset(pd, 0, sizeof(pipe_desc_t));

	pd->pipe_buf = malloc(PIPE_SIZE);
	if (pd->pipe_buf == NULL) {
		set_errno(ENOMEM);
		return -1;
	}

	/* Init internal structure members */

	mutex_init(&pd->lock);

	init_completion(&pd->wait_for_reader);
	init_completion(&pd->wait_for_writer);

	/* Chacun des fonctions sont appelées deux fois */

	/*
	 * Tout d'abord nous allons ouvrir et enregister 2 nouveaux descripteurs de
	 * fichier dans la table noyau (global). La fonction vfs_open retourne
	 * un descripteur de fichier local (processus).
	 * Nous les affectons au tableau que l'utilisateur nous a fournis.
	 * Il faut passer à la fonction vfs_open :
	 * La structure de type struct file_operation (ici pipe_ops) unique
	 * pour les pipe. De plus il faut passer le type de descripteur de
	 * fichier dont il s'agit : c-à-d VFS_TYPE_PIPE dans notre cas.
	 * */
	pipefd[0] = vfs_open(&pipe_fops, VFS_TYPE_PIPE);
	pipefd[1] = vfs_open(&pipe_fops, VFS_TYPE_PIPE);

	/*
	 * Ensuite, nous copions dans le tableau gfd membre de pd
	 * (de type pipe_desc_t)les descripteurs de fichiers globaux
	 *  avec la fonction vfs_get_gfd(int fd)
	 * */
	pd->gfd[0] = vfs_get_gfd(pipefd[0]);
	pd->gfd[1] = vfs_get_gfd(pipefd[1]);

	/*
	 * Comme le pipe est une mechanisme IPC unidirectionel, il faut
	 * établir des droit d'accès differents pour chaque côté du pipe.
	 * */
	vfs_set_access_mode(pd->gfd[0], O_RDONLY);
	vfs_set_access_mode(pd->gfd[1], O_WRONLY);

	/*
	 * Ensuite, il faut copier le pd dans les
	 * données privées du descripteur de fichier. Ceci permettra lors d'un read
	 * ou write de connaitre les information lié au pipe (position dans le
	 * buffer…)	*/
```

```
        vfs_set_privdata(pd->gfd[0], pd);
        vfs_set_privdata(pd->gfd[1], pd);

        return 0;
}
```

5. *Espace utilisateur*

Il faut créer une nouvelle application. Ainsi il faut ajouter test_pipe au fichier usr/src/Makefile.
Voici un example d'implementation de test_pipe.c :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <syscall.h>
#include <unistd.h>

#define FINI "msg_fini"
#define YES "yes"
#define END "end"

/* Size of the message send by the parent */
#define BUF_MSG 8
/* Size of the message send by the child */
#define BUF_REPLY 3

/* This function will receive messages from the parent
 * and reply to the parent accodingly to the specification
 * */
void child_routine(int read_pipe, int write_pipe)
{
        char buf_recv[BUF_MSG + 1];
        size_t read_sz;
        size_t total_read = 0;

        printf("[processus_fils]\r");
        fflush(stdout);
        while (1) {

                if ((read_sz = read(read_pipe, &buf_recv[total_read], 1)) <= 0) {
                        break;
                }

                total_read += read_sz;

                printf("[processus_fils] %s", buf_recv);
                if (BUF_MSG == total_read) {
                        putchar('\n');
                        if (!strncmp(buf_recv, FINI, BUF_MSG)) {
                                        break;
```

```
                        }

                        memset(buf_recv, 0, BUF_MSG);
                        write(write_pipe, YES, BUF_REPLY);
                        total_read = 0;
                } else {
                        /* Retour au début de la ligne*/
                        putchar('\r');
                        fflush(stdout);
                }

        }

        /* Ici le fils termine et ecrit “end” */
        write(write_pipe, END, BUF_REPLY);

}

void parent_routine(int read_pipe, int write_pipe)
{
        char buf_recv[BUF_REPLY];
        char buf_send;
        size_t read_stdin_sz;
        size_t total_read = 0;

        while (1) {
                /* Nous lisons les donées arrivant du stdin (clavier utilisateur */
                if ((read_stdin_sz = read(STDIN_FILENO,
                                                &buf_send, 1)) <= 0) {
                        printf("Error while reading stdin\n");
                }

                total_read += read_stdin_sz;
                write(write_pipe, &buf_send, read_stdin_sz);

                if (total_read == BUF_MSG) {
                        if (read(read_pipe, buf_recv, BUF_MSG)<=0){
                               printf("Error while reading child_pipe\n");
                               break;
                        }
                        /* Le processus fils écrit “end” au parent nous affichons
                         * le message puis quittons
                         */
                        if (!strncmp(buf_recv, END, BUF_REPLY)) {
                                printf("[processus_parent] %s\n", buf_recv);
                                break;
                        } else if (!strncmp(buf_recv, YES, BUF_REPLY)) {
                                /* Le processus fils ecrit au parent “yes” */
                                printf("[processus_parent] %s\n", buf_recv);
                        }

                        total_read = 0;
```

```
                }
        }
}


int main(int argc, char **argv)
{
        int pid;
        /* Pipe de communication parent → fils */
        int fd_pipe_pc[2];
        /* Pipe de communication fils → Parent */
        int fd_pipe_cp[2];

        pipe(fd_pipe_cp);
        pipe(fd_pipe_pc);
        pid = fork();

        if (pid < 0) {
                exit(EXIT_FAILURE);
        }

        if (!pid) { /* Processus fils */
                close(fd_pipe_pc[1]);
                close(fd_pipe_cp[0]);

                child_routine(fd_pipe_pc[0], fd_pipe_cp[1]);

                close (fd_pipe_pc[0]);
                close (fd_pipe_cp[1]);
                exit(EXIT_SUCCESS);
        }
        /* Processus parent */
        close(fd_pipe_pc[0]);
        close(fd_pipe_cp[1]);

        parent_routine(fd_pipe_cp[0], fd_pipe_pc[1]);

        close(fd_pipe_pc[1]);
        close(fd_pipe_cp[0]);

        waitpid(pid, NULL, 0);

        exit(EXIT_SUCCESS);
}
```

6. Utilisation d'un tube dans le shell

Afin de pouvoir utiliser les pipe dans le shell et rediriger le flux STDIN du processus parent vers STDOUT :

```
int piped_fd[2];
void runline(char *line)
{
        int background, status;
        int pid0, pid1;
        int i, j;
        int argc, argc2;
        int ret = 0;
        int pipeOn = 0;
        static char args[BUFFERSIZE], args2[BUFFERSIZE], prog[BUFFERSIZE],
cwd[BUFFERSIZE];
        static char *argv[MAXARGS], *argv2[MAXARGS];
        static char cmd1[BUFFERSIZE], cmd2[BUFFERSIZE];
        /* Pipe */
        int pipeOn = 0;

        /* Look for '>' (redirections) */
        strcpy(cmd1, "");
        strcpy(cmd2, "");

/* recherche un le caractère | dans la ligne de commande */
for (i = 0 ; (i < strlen(line) + 1) && (!pipeOn)) ; i++)
    switch (line[i]) {
    case '|':
        cmd1[i] = 0;
        pipeOn = 1;
        break;
    default:
        cmd1[i] = line[i];
        break;
}

/* Right side (if something has been found) */
        i = strlen(cmd1);
        if (i < strlen(line)+1)
                for (j = 0, i++ ; i < strlen(line)+1; i++, j++)
                        cmd2[j] = line[i];

        argc = tokenizeCommand(cmd1, MAXARGS, argv, args);
        if (argc <= 0)
                return;

        if (*cmd2) {
                argc2 = tokenizeCommand(cmd2, MAXARGS, argv2, args2);
                if (argc2 <= 0)
                  return;
         } else {
                if (pipeOn)
```

```
                        return;
        }
(… function runline continue)
else { /* All the other commands */

        pid0 = fork();
        if (-1 == pid0) {
                printf("Error while forking process\n");
                return ;
        }

        if (!pid0) {
                if (is_pipe) {
                        pipe(pipe_fd);
                        pid1 = fork();
                        if ( -1 == pid1) {
                              printf("Error while forking process\n");
                        }

                        if (!pid1) { /* fils pid1*/
                              close(pipe_fd[0])s;
                              dup2(pipe_fd[1], STDOUT_FILENO);

                              strcpy(prog, argv[0]);
                              strcat(prog, ".elf");

                              if (exec(prog, argc, argv) == -1) {
                                    printf("%s: exec failed.\n", argv[0]);
                                    exit(-1);
                              }
                        } else { /* Il s'agit du parent de pid1*/
                              close(pipe_fd[1]);
                              dup2(pipe_fd[0], STDIN_FILENO);

                              strcpy(prog, argv2[0]);
                              strcat(prog, ".elf");

                              if (exec(prog, argc2, argv2) == -1) {
                                   printf("%s: exec failed.\n", argv2[0]);
                                   exit(-1);
                              }
                        }
                } else { /* Processus fils pid0 */
                        strcpy(prog, argv[0]);
                        strcat(prog, ".elf");

                        if (exec(prog, argc, argv) == -1) {
                             printf("%s: exec failed.\n", argv[0]);
                             exit(-1);
                         }
                } /* End processus fils pid0*/
        }
```

```
}
```

7. Application tee

Réalisez une application "*tee*" qui permet de lire le flux d'entrée standard et qui l'envoie sur la sortie standard ainsi que dans un fichier dont le nom est passé en paramètre, tel que le montre l'exemple ci-dessous.

Il faut rajouter un nouvelle application nommée ***tee*** dans le fichier usr/src/Makefile. Voici une implementation possible :

```
int main(int argc, char **argv)
{
        int fd_file;
        int read_sz;
        char data;
        /* Verifions que le nombre d'arguments est le bon */
        if (argc != 2) {
                printf("Wrong number of arguments\n");
                exit(EXIT_FAILURE);
        }

        fd_file = open(argv[1], O_WRONLY | O_TRUNC | O_CREAT);

        /* Si il n'y a pas de descripteur de fichier disponible nous quittons*/
        if (-1 == fd_file)
                exit(EXIT_FAILURE);

        /* Tant que le pipe est ouvert nous ecrivons le flux reçu dans
         * STDIN dans le buffer char
         */
        while ((read_sz = read(STDIN_FILENO,  &data, 1)) > 0) {
                write(fd_file, &data, read_sz);
                putchar(data);
        }

        putchar('\n');
        close(fd_file);

        return 0;
}
```