Projet OS

Youri Mouton, Rémy Voet et Samuel Monroe

Janvier 2015

Première partie

Introduction

1 Introduction

Le projet qui fait l'objet de ce rapport est un travail de programmation en C système sous UNIX, dans le but d'exploiter de manière pratique les connaissances théoriques acquises au cours de Mme. Masson.

Ce travail consiste en l'écriture de trois programmes distincts autour d'une thématique aéronautique, échangeant des informations entre eux via divers moyens, ceci tout en gérant les éventuels conflits et erreurs liés à ces échanges.
Ces trois programmes sont :

Tour de controle : Fait office de **serveur**, joue le rôle majeur en s'occupant de recevoir les demandes des pilotes, en allant chercher les informations météo fournies par le centre météo, et en renvoyant celles-ci en tant que réponses aux demandes des pilotes.

Pilote : Fait office de **client**, envoie une demande d'information ATIS à la tour de contrôle, récupère cette information et replace une demande si cette information n'est pas intègre.

Le centre météo : Programme **tiers** dans l'application, celui-ci se charge de générer automatiquement des informations ATIS différentes tout au long de son fonctionnement.

2 Rappel de l'énoncé

Des pilotes qui souhaitent décoller d'un aéroport non contrôlé ont besoin, pour ce faire, de connaître les informations ATIS. Celles-ci sont accessibles via un serveur. Chaque pilote va envoyer au serveur une demande ATIS. Le serveur va lors répondre à cette demande en allant chercher les informations nécessaires dans un fichier ATIS. Le pilote va recevoir ces informations et il doit alors obligatoirement répondre au serveur en lui envoyant soit :

- Un NAK OK qui signifie « informations bien reçues »et provoque la fin de la communication
- Un ACK KO qui signifie « informations mal comprises »et nécessite de renvoyer les informations.

Le serveur doit pouvoir gérer un nombre indéfini de pilotes (restons réalistes). Le fichier ATIS contenant les informations nécessaires aux pilotes doit être régulièrement mis à jour par le gestionnaire météo.

Deuxième partie

Analyse - Plan de l'application

Troisième partie

Détails des élements techniques spécifiques

Quatrième partie

Conclusion

Cinquième partie

Annexe

Annexe A

meteo.c

```
#include "global.h"

void delete(const char * pathname);
bool exists(const char * pathname);

static int meteo = -1;
static int lock = -1;
static bool cont = true;

char ATIS[][MSG_SIZE] = {
    "ATIS 1ONE EBLG 1803 00000KT 0600 FG OVC008 BKN040 PROB40 2024 0300 DZ FG OVC002 BKN040",
    "ATIS 2TOW EBBR 0615 20015KT 8000 RA SCT010 OVC015 TEMPO 0608 5000 RA BKN005 BECMG 0810
         NSW BKN025",
    "ATIS 3TRHE METAR VHHH 231830Z 06008KT 7000 FEW010SCT022 20/17 Q1017 NOSIG 5000 RA BKN005",
    "ATIS 4FRUO 20015KT 8000 RA SCT010 OVC015 2024 0300 DZ FG 0810 9999 NSW BKN025",
    "ATIS 5VEFI KT 7000 FEW010SCT02 EMPO 0608 5000 RA BKN005 EMPO 0608 5000 RA BKN005"
};

void openLock(void) {
    if ((lock = open(FICHIERLOCK, O_CREAT | O_WRONLY, S_IRUSR |
                  S_IWUSR | S_IRGRP)) == FAIL) {
        printf("Unable to create the lock file\n");
        exit(EXIT_FAILURE);
    }
}

void openMeteo(void) {
    if ((meteo = open(FICHIERMETEO, O_CREAT | O_WRONLY |
                  O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP)) == FAIL) {
        printf("Unable to open/create the meteo file\n");
        exit(EXIT_FAILURE);
    }
}

void closeLock(void) {
    if (exists(FICHIERLOCK)) {
        if (close(lock) == FAIL) {
            printf("Unable to close the lock file\n");
            exit(EXIT_FAILURE);
        }
    }
}

void closeMeteo(void) {
    if (exists(FICHIERMETEO)) {
        if (close(meteo) == FAIL) {
            printf("Unable to close the meteo file : %s\n", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }
}

void pilotCleanup(int state) {
    if (state == EXIT_SUCCESS || state == SIGINT) {
        cont = checkyesno("Shutdown meteo");
    } else if (state == EXIT_FAILURE) {
        cont = false;
    }
    if (!cont) {
        delete(FICHIERMETEO);
        if (exists(FICHIERLOCK)) {
            delete(FICHIERLOCK);
        }
    }
}

int genAtis(void){

    while (cont) {
        int msg = 0;
        unsigned int nATIS = sizeof(ATIS) / sizeof(ATIS[0]);
        if (nATIS > 0) {
            msg = rand() % nATIS;
        } else {
            return EXIT_FAILURE;
        }
        printf("ATIS Create : %s\n", ATIS[msg]);
        openLock();
        openMeteo();
        if (write(meteo, ATIS[msg], sizeof(ATIS[msg])) == FAIL) {
            return EXIT_FAILURE;
        }
        // virtual write wait time, to make the
        // meteo transmission latency more realistic :)
        sleep(WRITE_TIME);
        closeLock();
        delete(FICHIERLOCK);
        sleep(WAIT_TIME);
    }
    return EXIT_SUCCESS;
}

int main(void) {
    signal(SIGINT, &pilotCleanup);
    return genAtis();
}
```

Annexe B

pilot.c

```
#include "global.h"

#define VALID_ATIS "ATIS"
#define VALID_LGT 4
void pilot_cleanup(int, int, int);

int main(void) {
    int server = -1;
    int out_server = -1;
    int not_received = -1;

    char request[MSG_SIZE];
    char confirm[MSG_SIZE];
    char buf[MSG_SIZE];
    char response[MSG_SIZE];
    size_t responseSize = 0;
    memcpy(request, PILOT_REQUEST, MSG_SIZE);
    if((server = open(FIFO_FILE, O_WRONLY)) == FAIL) {
        printf(RED"Server seems to be down...\n"NOR);
    } else {
        printf("Sending REQUEST...\n");
        if (write(server, request, sizeof(request)) == FAIL) {
            printf("Failed to write message\n");
            pilot_cleanup(server, out_server, FAIL);
        } else {
            if ((out_server = open(FIFO_FILE_OUT, O_RDONLY)) == FAIL) {
                printf("Couldn't open output file...\n");
                pilot_cleanup(server, out_server, FAIL);
            } else {
                while (not_received) {
                    responseSize = read(out_server, buf, sizeof(buf));
                    if (responseSize == FAIL) {
                        printf("Failed to read response from output fifo\n");
                        pilot_cleanup(server, out_server, FAIL);
                    } else {
                        //Response received from the server
                        memcpy(response, buf, responseSize);
                        //Is the response valid?
                        if (memcmp(response, VALID_ATIS, VALID_LGT) == 0) {
                            printf("Got response ! => %s\n", response);
                            if (write(server, ACK, sizeof(ACK)) == FAIL) {
                                printf("Failed to send ACK");
                            }
                            not_received = 0;
                        } else {
                            printf(RED"ERROR : %s\n"NOR, response);
                            printf("Sending NAK to the serveur, asking for ATIS again... \n");
                            if (write(server, NAK, sizeof(NAK)) == FAIL) {
                                printf("FAILED to send NAK");
                            }
                            sleep(2);
                        }
                    }
                }
            }
        }
        pilot_cleanup(server, out_server, EXIT_SUCCESS);
    }
}


void pilot_cleanup(int in_serv, int out_serv, int status) {
    if (in_serv != FAIL) {
        if (close(in_serv) == FAIL) {
            printf("Couldn't close input file descriptor %d\n", in_serv);
            exit(status);
        }
    }
    if (out_serv != FAIL) {
        if (close(out_serv) == FAIL) {
            printf("Couldn't close output file descriptor %d\n", out_serv);
            exit(status);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Annexe C

server.c

```
#include "global.h"

#define MAX_TRY 20

int input = -1;
int output = -1;
bool listen = true;
int nb = 0;
char **requests = NULL;

int atis(char * atisMsg) {
    int fichierMeteo = 0;
    int atisSize = 0;
    char buf[MSG_SIZE];
    fichierMeteo = open(FICHIERMETEO, O_RDONLY);
    if (fichierMeteo == FAIL) {
        printf("Unable to open meteo file\n");
        memcpy(atisMsg, UNREACHABLE, sizeof(UNREACHABLE));
        atisSize = sizeof(UNREACHABLE);
    } else if (exists(FICHIERLOCK)) {
        printf("The meteo server is busy\n");
        memcpy(atisMsg, BUSY, sizeof(BUSY));
        atisSize = sizeof(BUSY);
    } else {
        atisSize = (int)read(fichierMeteo, buf, sizeof(buf));
        if (atisSize == FAIL) {
            fatal("Unable to read from %s\n", FICHIERMETEO);
        } else {
            memcpy(atisMsg, buf, atisSize);
        }
    }
    return atisSize;
}

void createFifos(void) {
    if (mkfifo(FIFO_FILE, S_IRUSR | S_IWUSR) == FAIL) {
        fatal("Unable to create input fifo\n");
    }
    if (mkfifo(FIFO_FILE_OUT, S_IRUSR | S_IWUSR) == FAIL) {
        fatal("Unable to create output fifo\n");
    }
}

void openFifos(void) {
    if ((input = open(FIFO_FILE, O_RDWR)) == FAIL) {
        fatal("Unable to open the server's fifo\n");
    }
    if ((output = open(FIFO_FILE_OUT, O_RDWR)) == FAIL) {
        fatal("Unable to open server ouput fifo %s\n", FIFO_FILE_OUT);
    }
}

void operations(void) {
    // poll in the input server
    // for incomind data and for
    // client disconnects.
    struct pollfd     fd[] = {
        { input, POLLIN | POLLHUP, 0 }
    };
    // request packet
    char requestPacket[MSG_SIZE];
    requests = NULL;

    // atis
    char atisMsg[MSG_SIZE];
    printf("^C to exit\n- Listening...\n");
    while (listen) {
        // poll the input fifo every second
        int fifoActions = poll(fd, 1, 1000);
        if (fifoActions == FAIL) {
            if (listen) {
                printf("%s\n", "polling again...");
            }
        } else if (fifoActions == 0) {
            printf("- Listening...\n");
        } else {
            int packet = (int)read(input, requestPacket, MSG_SIZE);
            if (packet == FAIL) {
                if (fd[0].revents & POLLHUP) {
                    close(input);
                    if ((input = open(FIFO_FILE, O_RDWR)) == FAIL) {
                        fatal("Unable to open the input fifo.\n");
                    }
                }
            } else {
                // assign read string
                requests = xrealloc(requests, nb+1, MSG_SIZE);
                requests[nb] = xmalloc(MSG_SIZE);
                memcpy(requests[nb], requestPacket, packet);
                if ((memcmp(requests[nb], PILOT_REQUEST,
                           sizeof(PILOT_REQUEST)) == 0) || (memcmp(requests[nb], NAK,
                                sizeof(NAK)) ==0)) {
                    printf("< Got Request nr. %d\n", nb);
                    size_t tailleMsg = (size_t)atis(atisMsg);
                    printf("> Sending ATIS \"%s\" to %d\n", atisMsg, nb);
                    if (write(output, atisMsg, tailleMsg) == FAIL) {
                        fatal("Failed to send message...\n");
                    }
                    nb++;
                } else if(memcmp(requests[nb], ACK, sizeof(ACK)) ==0) {
                    printf(GRN"A pilot just acknwoledged the reception ! \n"NOR);
                } else {
                    fatal("No valid messages intercepted\n");
                }
            }
        }
    }
}

int main(void) {
    // setup signal, so if programs gets
    // interrupted, files can still be
    // cleaned up and FIFOs removed.
    signal(SIGINT, &cleanup);
    createFifos();
    openFifos();
    operations();
    return EXIT_SUCCESS;
}
```

Annexe D

tools.c

```
/*
 *
 * Utilities.
 *
 */
#include "global.h"

bool listen;
int input;
int output;
int nb;
char **requests;

/*
 * xmalloc is a wrapper around malloc that checks
 * if the size parameter is greater than zero, and
 * if the returned pointer isn't null so the
 * programmer doesn't have to do it in other files.
 *
 * Source: OpenSSH 6.6.1, adapted to use a modified
 * fatal() function as we're not logging to syslog
 * like OpenSSH does.
 *
 * Author: Tatu Ylonen
 */
void * xmalloc(size_t size) {
   void *ptr;

   if (size == 0) {
      fatal("xmalloc: zero size\n");
   }

   ptr = malloc(size);

   if (ptr == NULL) {
      fatal("xmalloc: out of memory (allocating %zu bytes)", size);
   }

   return ptr;
}

/*
 * xrealloc is a wrapper around xmalloc that checks
 * if the number parameter is greater than zero and
 * if the total size is smaller than SIZE_T_MAX per
 * number.
 *
 * It also malloc's a new pointer if xreallocs pointer
 * parameter is null and then actually reallocs a
 * pointer. It checks if the returning pointer is null
 * so the programmer doesn't have to do it in other
 * files.
 *
 * Source: OpenSSH 6.6.1, adapted to use a modified
 * fatal() function as we're not logging to syslog
 * like OpenSSH does.
 *
 * Author: Tatu Ylonen
 */
void * xrealloc(void *ptr, size_t nmemb, size_t size) {
   void *new_ptr;
   size_t new_size = nmemb * size;

   if (new_size == 0)
      fatal("xrealloc: zero size");
   if (SIZE_T_MAX / nmemb < size)
      fatal("xrealloc: nmemb * size > SIZE_T_MAX");
   if (ptr == NULL)
      new_ptr = malloc(new_size);
   else
      new_ptr = realloc(ptr, new_size);
   if (new_ptr == NULL)
      fatal("xrealloc: out of memory (new_size %zu bytes)",
            new_size);
   return new_ptr;
}

/*
 * cleaPtr frees a two dimensional array
 * of characters (an array of strings) by
 * looping through the char 2D array to
 * the count parameter, free the array
 * content and set the content to NULL,
 * to avoid dangling pointer bugs.
 *
 * If a dangling pointer (a pointer pointing
 * to nothing) is accessed after being freed,
 * you may overwrite random memory, but if a
 * NULL'ed pointer is accessed, a crash will
 * occur, with an appropriate error message
 * from the operating system.
 *
 * Author: Youri Mouton.
 */
void cleanPtr(int count, char ** array) {

   for (int i = 0; i < count; i++) {
      if (array[i] != NULL) {
         free(array[i]);
         array[i] = NULL;
      }
   }

   if (array != NULL) {
      free(array);
      array = NULL;
   }
}

/*
 * checkyesno asks for user input to decide
 * whether to keep the program running or not
 * and send a bool value accordingly. It will
 * loop until the user presses either 'y' or
 * 'n'. 'y' being yes, 'n' being no. The msg
 * parameter will be the question shown to the user.
 *
 * Example:
 * question: really exit ?
 * - yes returns false
 * - no returns true
 * for the loop variable
 *
 * Author: Youri Mouton
 */
bool checkyesno(const char *msg) {
    bool       ans;
    char       inp[BUFSIZ];
    char       res;

    do {

        printf("\n%s%s", msg, " ? (["RED"y"NOR"]es/["RED"n"NOR"]o) : ");
        fgets(inp, sizeof(inp), stdin);
        sscanf(inp, "%c", &res);

    } while (res != 'y' && res != 'n');


    return res == 'y' ? false : true;
}

/*
 * returns true or false depending on the
 * existence of the file designated by
 * pathname in the parameters.
 *
 * Author: Youri Mouton
 */
bool exists(const char * pathname) {
    struct stat       info;
    return (stat(pathname, &info) == 0);
}

/*
 * delete is a procedure that will try to
 * remove file name after the pathname
 * parameter and throw an appropriate error
 * message if the file doesn't exist or if
 * we can't delete it.
 *
 * Author: Youri Mouton
 *         Samuel Monroe
 */
void delete(const char * pathname) {

    if (!exists(pathname)) {
        printf("File %s doesn't exist!\n", pathname);
    } else {
        if (unlink(pathname) == -1) {
            printf("Failed to delete %s !\n", pathname);
        }
    }
}

/*
 * cleanup is a procedure that makes sure to
 * clean the file descriptors and to delete
 * the named pipes and the pointers used.
 *
 * cleanup will run checkyesno(msg) if the
 * program returns normally or if the user
 * interrups the program (ctrl+c, for example).
 * If the program terminates abnormally,
 * cleanup will be run directly.
 *
 * cleanup has a state parameter that can be
 * set to decide whether we're calling cleanup
 * when the program is failing to make an
 * operation or if we're exiting normally.
 *
 * cleanup is called from the server's main
 * on SIGINT, so when the user interrupts
 * the process, it is fine to call it from
 * signal() or sigset() from other places,
 * but be careful to not cleanup files needed
 * by the other processes.
 *
 * Author: Youri Mouton
 */
void cleanup(int state) {

   if (state == EXIT_SUCCESS || state == SIGINT) {
      listen = checkyesno("Really quit");
   } else if (state == EXIT_FAILURE) {
      listen = false;
   }

   if (!listen) {

      printf("\ncleaning up... \n");

      if (input != -1 || output != -1) {

         if (close(input) == FAIL) {
            printf("Couldn't close input file descriptor %d\n", input);
         }

         if (close(output) == FAIL) {
            printf("Couldn't close output file descriptor %d\n", output);
         }
      }

      delete(FIFO_FILE);

      delete(FIFO_FILE_OUT);

      cleanPtr(nb, requests);
   }

}

/*
 * fatal is an error handling function,
 * it wraps a printf function that will
 * display the error message set by the
 * programmer as parameter, prepended
 * by a 'FATAL:' message printed in red.
 *
 * The error message set when calling
 * fatal is variadic, it can be used
 * like any printf.
 *
 * fatal then calls cleanup and exits
 * with a failure state, so fatal should
 * only be used for error handling.
 *
 * Author: Youri Mouton
 */
int fatal(const char * format, ...) {

   va_list    args;

   va_start(args, format);

   fprintf(stderr, RED"FATAL: "NOR);
   vfprintf(stderr, format, args);

   va_end(args);

   cleanup(EXIT_FAILURE);

   exit(EXIT_FAILURE);
}
```

Table des matières