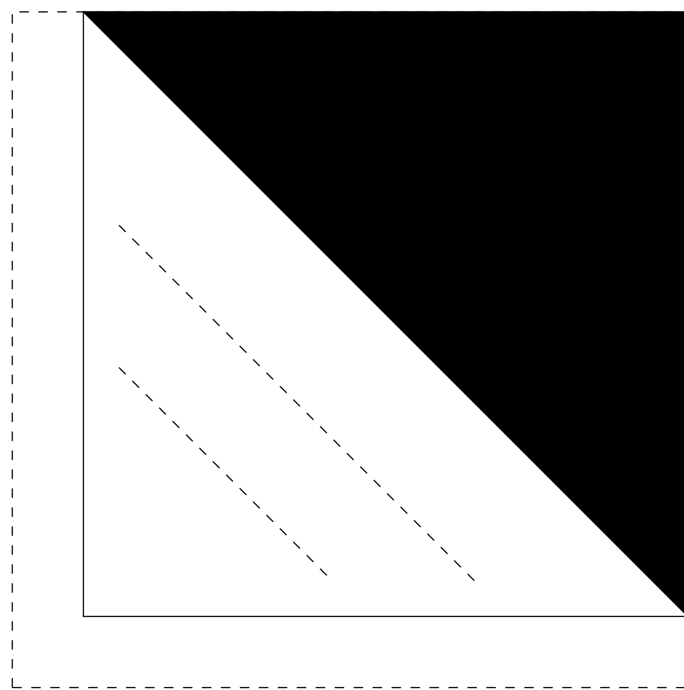
Contrôle de LIFASR5 — 60 minutes

Le mardi 26/05/2021 à 9h15 en amphis Ampère et Gouy.



Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Tout document est interdit, tout comme toute utilisation d'un ordinateur, d'un téléphone, d'une montre connectée, etc. Vous disposez uniquement de l'antisèche fournie, qui contient toutes les informations utiles. Ce sujet est à rendre complété : n'oubliez pas de noter votre nom, votre prénom et votre no. étudiant.

Notez tout de suite votre prénom, votre nom et votre numéro d'étudiant.e sous le cache rabattable à droite. Au moment de rendre votre copie, rabattez et collez le cache.

Après avoir parcouru tout le sujet, traitez d'abord les exercices qui vous semblent les plus abordables en fonctions de vos compétences !

> **Solution: ATTENTION, IL PEUT RESTER DES ERREURS DANS LES ELEMENTS DE CORRECTION ! ! !**

1 Transfert de données à l'aide d'un tube (5 points)

On considère le programme de la figure 1 (page suivante), dans lequel le processus principal crée un pipe (tube), puis deux processus fils, `fils1` et `fils2`. À l'aide de la primitive dup2() :

— `fils1` remplace sa sortie standard par `tube[1]` (tout ce qu'il écrit sur sa sortie standard est envoyé dans le tube) ;

— `fils2` remplace son entrée standard par `tube[0]` (tout ce qu'il lit sur son entrée standard est lu depuis le tube).

1. (0,5 points) Dans le programme, comment le processus `fils1` fait-il pour envoyer des caractères vers sa sortie standard (c'est-à-dire en fait vers `tube[1]`) ?

> **Solution:** Il écrit sur sa sortie standard simplement en envoyant du texte sur cout avec `operator<<()`. En effet, cout permet "d'afficher du texte à l'écran", c'est-à-dire en fait de l'envoyer vers la sortie standard.

2. (1 point) Dans le `fils2`, la commande `wc -l` (commande pour compter le nombre de lignes) est lancée à l'aide de la primitive de recouvrement `execl()`. A l'exécution, que va afficher `fils2` sur sa sortie standard, et pourquoi ?

> **Solution:** Le `wc -l` va compter le nombre de lignes qu'il reçoit sur son entrée standard, et l'afficher vers sa sortie standard. La commande reçoit la 1ère strophe de "l'araignée Gipsy", envoyée via le tube sur son entrée standard. Le fils2 va donc écrire "`fils2 -> 5`".

3. (1 point) Le programme s'exécute parfaitement sans erreur, et `fils2` n'affiche pas le message "erreur", ce qui indique qu'il ne passe jamais par l'instruction `cout << "erreur" << endl`. Expliquez pourquoi.

> **Solution:** en l'absence d'erreur, le code du processus appelant est entièrement remplacé par celui de la commande passée en paramètre de `execl()`. En l'absence d'erreur, le processus fils1 n'exécute donc pas les instructions qui se trouvent après le `execl()` dans notre programme.

4. (0,5 points) Pourquoi le processus père fait-il deux appels à `waitpid()` ?

> **Solution:** Parce qu'il attend la terminaison de ses deux fils, `fils1` et `fils2`, dont les PID sont respectivement pid1 et pid2.

5. (2 points) Si l'on supprime le `close(tube[1])` au niveau du commentaire "ici" (ligne 31) dans le processus père, alors le `fils2` reste bloqué indéfiniment en attente de lecture sur le tube : expliquez pourquoi ?

```
1  int main(void) {
2    int tube[2];
3    pipe(tube); // tube[0] pour la lecture, tube[1] pour l'écriture
4
5    int pid1 = fork();
6    if(pid1 == 0) { // processus fils1
7      cout << "fils1" << endl;
8      dup2(tube[1], STDOUT_FILENO); // STDOUT_FILENO devient l'entrée tube[1]
9      close(tube[0]);
10     close(tube[1]);
11     cout << "L'araignée Gipsy" << endl;
12     cout << "Monte à la gouttière" << endl;
13     cout << "Tiens voilà  la pluie" << endl;
14     cout << "Gipsy tombe par terre" << endl;
15     cout << "Mais le soleil va chasser la pluie" << endl;
16     return 0;
17   }
18
19   int pid2 = fork();
20   if(pid2 == 0) { // processus fils2
21     cout << "fils2 ->";
22     dup2(tube[0], STDIN_FILENO); // STDIN_FILENO devient l'entrée tube[0]
23     close(tube[0]);
24     close(tube[1]);
25     execl("/usr/bin/wc", "/usr/bin/wc", "-l", (char*)NULL);
26     cout << "erreur" << endl;
27     return 1;
28   }
29
30   close(tube[0]);
31   close(tube[1]); // ici
32   waitpid(pid1, NULL, 0);
33   waitpid(pid2, NULL, 0);
34   return 0;
35 }
```

FIGURE 1 – Programme de l'exercice « transfert de données à l'aide d'un tube ».

Solution: La commande "wc -l" lancée dans le `fils2` lit depuis son entrée standard, c'est-à-dire ici depuis le tube. Elle reste donc en attente de lecture sur le tube tant qu'il reste au moins un descripteur de fichier en écriture dessus, ce qui est le cas de `tube[1]`. Si jamais le père laisse `tube[1]` ouvert, le `fils2` reste bloqué en attente de lecture sur le tube. `fils2` ne peut donc pas se terminer, et le père attend la terminaison de `fils2` : on ne sortira jamais de ce blocage...

2 Système de fichiers (6 points)

Le système de fichier FAT12 utilise une table d'allocation : il s'agit d'un tableau d'entiers sur 12 bits, avec des indices codés sur 12 bits. FAT12 alloue les blocs par chaînage dans la table d'allocation : chaque entrée de la table d'allocation contient une valeur qui est : soit l'adresse du bloc suivant d'un fichier, soit une valeur spéciale, par exemple une valeur marquant la fin du fichier. Voici la signification des valeurs possibles (elles sont données en hexadécimal) :

valeur	signification
0x000	le bloc est libre et peut être utilisé pour l'allocation de blocs
0x001 à 0xFEF	adresse du bloc suivant
0xFF0 à 0xFF6	le bloc est réservé par l'algorithme d'allocation de blocs
0xFF7	le bloc est défectueux et ne peut plus être utilisé pour l'allocation de blocs
0xFF8 à 0xFFF	dernier bloc d'un fichier

1. (0,5 points) Donnez le nombre de valeurs possibles pour les entrées de la table d'allocation, sous la forme d'une puissance de 2 puis en décimal.

Solution: Les valeurs sont codées sur 12 bits, soit $2^{12} = 4096$ valeurs possibles.

2. (0,5 points) Combien de blocs peut comporter au maximum un fichier dans ce système de fichier ? Précisez votre calcul, et donnez votre réponse en décimal.

Solution: Au maximum, le fichier contient tous les blocs du disque. Les blocs sont identifiés chacun par un entier sur 12 bits, et il faut soustraire les valeurs spéciales : $2^{12} - (\text{nombre de valeurs spéciales}) = 4096 - 17 = 4079$ blocs.

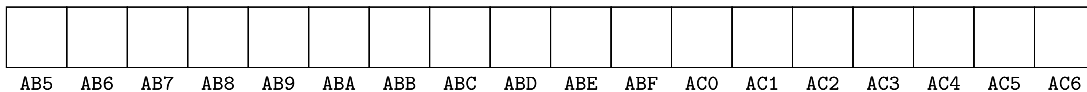
3. (1 point) La taille en octets des blocs dans le système FAT12 est codée sur 2 octets. Quelle est la taille maximale d'un fichier ? Donnez le calcul à faire pour avoir le résultat exact et une estimation en Mio.

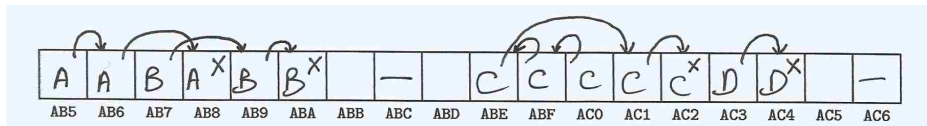
> **Solution:** Le taille des blocs en octets est codée sur 2 o = 16 bits, on peut donc choisir des blocs d'une taille maximale de $2^{16} - 1$ o. Le fichier de taille maximale occupe tous les blocs possibles, avec pour chacun une taille maximale ; sa taille est de $4079 \times (2^{16} - 1)$ o, ce qui fait un peu moins de 4096×2^{16} o $= 2^{28}$ o $= 256 \times 2^{20}$ o $= 256$ Mio.

4. (2 points) Voici un extrait de la table d'allocation d'un disque, avec les adresses et les valeurs en hexadécimal :

adresse	AB5	AB6	AB7	AB8	AB9	ABA	ABB	ABC	ABD	ABE	ABF	AC0	AC1	AC2	AC3	AC4	AC5	AC6
valeur	AB6	AB8	AB9	FF8	ABA	FF8	000	FF7	000	AC1	ABE	ABF	AC2	FF8	AC4	FFF	000	FF7

Complétez le schéma ci-dessous en :
— en précisant bien les chaînages entre les blocs à l'aide de fléches,
— en attribuant une lettre (`A, B, C`...) à chaque fichier présent,
— en marquant les blocs qui terminent un fichier par une croix (×) et les blocs défectueux par un tiret (−),
— en laissant vide les cases correspondant aux blocs libres.



> **Solution:** Logiquement, on doit arriver à un schéma de ce genre :
>
> 
>

5. (2 points) On doit ajouter 3 blocs à la fin du fichier commençant avec le bloc d'adresse `0xAC3` : expliquez quels sont les chaînages à modifier dans la table d'allocation décrite pour allouer ces blocs (plusieurs réponses sont possibles).

> **Solution:** Le dernier bloc du fichier qui commence avec `0xAC3` est pour l'instant `0xAC4`. On peut par exemple modifier les chaînages pour que
> — le suivant du bloc `0xAC4` devienne le bloc `0xAC5`,
> — le suivant de `0xAC5` devienne le `0xABB`,
> — le suivant de `0xABB` devienne le `0xABD`,
> — le suivant de `0xABD` est fixé à `0xFF8` pour marquer la fin du ficher.

3 Mémoire paginée (5 points)

Dans cet exercice, les entiers écrits en binaire sont préfixés par 0b, ceux écrits en octal par 0o, et ceux écrits en décimal sont laissés sans préfixe ni suffixe ; par exemple : $0o17 = (17)_8 = (001\ 111)_2 = 0b001\ 111 = (15)_{10} = 15$.

On considère un système à mémoire paginée, et on représente dans le tableau suivant une partie de la mémoire physique du système, dans son état actuel :

	0o0	0o1	0o2	0o3	0o4	0o5	0o6	0o7
0o00	cta-	est-	uB7K	mVfs	nWs4	Nfi2	RRUU	7tE0
0o01	qSFc	dB90	uUou	cUcx	LkPT	-en-	bas-	mGRw
0o02	vici	XnzW	Papa	-est	gg2	j6Tp	eirC	6mWK
0o03	0o12	0o17	0o16	0o02	0o06	0o24	0o07	0o30
0o04	sbj2	6y75	7xng	CH81	c3po	velo	ale-	a-ja
0o05	vica	FVlL	hC6	FRxx	-du-	K1Yv	uJqg	eg7H
0o06	-MIt	PTs	toto	-fait	zBcs	ct I	usu	A5Ag
0o07	dzPN	LmbP	rs0l	wnLg	jze0	M9tO	Z85	x-p-
0o10	qUK5	Dvs4	Ed5s	goF7	20Jk	9ZeN	tsE	tYxa
0o11	0o00	0o23	0o14	0o04	0o15	0o10	0o10	0o21
0o12	eOs7	Oki	supo	Mama	ivid	S5gD	HkwC	l-Yc
0o13	0Nqu	MxnY	AfAb	n-es	RTkf	I P4	TzN4	CPp2
0o14	Md i	Lrba	NFCV	t-en	JG5X	xO8g	R2D2	jT39
0o15	Nv0b	XX3q	n-mF	haut	yHQv	nbib	tceT	FSW0
0o16	sdfg	N3mf	qOOj	XSX9	Cw B	n2cq	veni	vidi
0o17	vico	8GAT	N217	tz 6	wYxW	H2O2	Hw44	HNO3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0o77	QSDd	RGfT	UFO-	OVNI	NaZe	prou	t-pr	out !

Les adresses physiques sont codées sur 9 bits, et chaque adresse correspond à une case-mémoire de 4 octets (4 caractères) en mémoire. Les indices 0o00, 0o01, ... des lignes du tableau correspondent aux 6 bits de poids forts des adresses physiques, et ce sont aussi les indices des cadres de la mémoire. Les indices 0o0, ..., 0o7 des colonnes correspondent aux 3 bits de poids faibles des adresses : ils donnent les décalages par rapport au début de page/cadre. Par exemple, dans le cadre 0o12 au décalage 2, se trouve la chaîne "supo" de 4 octets.
Les adresses virtuelles sont codées sur 6 bits : les 3 bits de poids forts donnent l'indice de la page à laquelle appartient l'adresse, et les 3 bits de poids faible le décalage sur la page.

On considère 2 processus qui s'exécutent sur ce système : la table des répertoires de pages du processus 1 est à l'adresse 0o110 (ligne 0o11 du tableau) et celle du processus 2 à l'adresse 0o030 (ligne 0o03 du tableau).

1. (1 point) Sur ce système, quelle est la *capacité d'adressage du processeur* et quelle est la *taille de l'espace adressage d'un processus* ? Répondez en rappelant les définitions.

   Solution: La *capacité d'adressage du processeur* est l'ensemble des adresses physiques que peut manipuler le processeur. Ici, ces adresses sont codées sur 9 bits, ce qui fait $2^9 = 512$ adresses.
   L'*espace d'adressage* est l'ensemble des adresses virtuelles qui peuvent être accédées par un processus. Ici, les adresses virtuelles sont codées sur 6 bits, la taille de l'espace d'adressage est donc de $2^6 = 64$ adresses.

2. (1 point) On note T_i la table des pages du processus i. Soit a_v une adresse virtuelle utilisée par le processus i, que l'on décompose sous la forme $a_v =$ | p (3 bits) | r (3 bits) | ; on note a_r l'adresse physique qui correspond à a_v, et on la décompose sous la forme $a_r =$ | c (6 bits) | r (3 bits) | . Comment est obtenu l'indice c du cadre dans lequel est rangée la page d'indice p ?

   Solution: Il faut chercher dans la table des pages du processus i dans quel cadre est rangé la page p : cet indice de cadre est $c = T_i[p]$.

3. (1 point) Donnez (en justifiant votre réponse) les 4 octets de la chaîne de caractères stockée à l'adresse virtuelle 0b100100 du processus 1.

   Solution: $T_1[0b100] = T_1[4] = 0o15$, donc on regarde dans le cadre d'indice 0o15 avec un décalage de $0b100 = 4$, et on trouve "yHQv".

4. (1 point) Donnez (en justifiant votre réponse) les 12 octets de la chaîne de caractères stockée à l'adresse virtuelle 0b010110 du processus 2.

   Solution: $T_2[0b010] = T_2[2] = 0o16$, donc on regarde dans le cadre d'indice 0o16 avec un décalage de $0b110 = 6$, et on trouve "venividi" pour les 8 premiers octets, mais on arrive en fin de page/cadre. Il faut donc reprendre à la page suivant la page 0b010 dans la mémoire virtuelle du processus 2 : il s'agit de 0b011, et on trouve que $T_2[0b011] = T_2[3] = 0o002$, et les deux premiers octets de ce cadre sont "vici". Les 12 octets de la chaîne de caractères stockée à l'adresse virtuelle 0b010110 du processus 2 sont donc "venividivici".

5. (1 point) Donnez (en justifiant votre réponse) les 16 octets de la chaîne de caractères stockée à l'adresse virtuelle 0o53 du processus 2.

   Solution: $T_2[0o5] = T_2[5] = 0o24$, donc on doit regarder dans le cadre d'indice 0o24. Or, ce cadre n'est pas représenté dans le tableau, on ne peut donc pas répondre à cette question.

4 Communication avec les sockets (4 points)

Vous devez écrire un serveur TCP/IP, qui se mettra à l'écoute sur le port 9999, et recevra les clients successivement. Vous disposez des fonctions suivantes pour faciliter la mise en œuvre des sockets :

```
// Retourne une socket côté serveur, à l'écoute sur le port passé en paramètre.
int create_server_socket(const char* port);

// Se met en attente bloquante d'une connexion sur la socket d'écoute s, et retourne la socket de dialogue créée.
int accept_connection(int s);

// Envoi d'une chaîne de caractères du type string sur la socket sd.
int sendstr(int s, const string &str);

// Met la date et l'heure dans la chaîne str de type string passée par référence.
void gettime(string &str);
```

Dès qu'un client viendra se connecter, le serveur devra envoyer :

- la chaîne `"Voici l'heure courante :\n"`,
- la date et l'heure courante suivies d'un retour à la ligne,
- la chaîne `"Fermeture de la connexion :\n"`,

puis il fermera la connexion avant de se remettre en attente du client suivant. Donnez le code de la fonction `main()` de votre serveur ci-dessous (ne prévoyez pas la terminaison de votre serveur, ne vous préoccupez pas des fichiers d'en-tête, ne définissez pas d'autres fonctions, ne vous préoccupez pas de la gestion des cas d'erreurs).

Les points qui seront notés : organisation du serveur (1 point), gestion des sockets (1 point), passage des paramètres lors des appels de fonctions (1 point), indentation et lisibilité du programme (1 point).

Solution:

```
int main(void) {
  int s = create_serveur_socket("9999");
  while(true) {
    int sd = accept_connection(s);
    string str;
    gettime(str);
    sendstr(sd, "Voici l'heure courante : \n");
    sendstr(sd, str + '\n');
    sendstr(sd, "Fermeture de la connexion :\n");
    close(sd);
  }
  return 0;
}
```

Antisèche :

NAME open -open a file
SYNOPSIS int open(const char *pathname, int flags);
DESCRIPTION
The open() system call opens the file specified by pathname. The return value of open() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (read(), write(), etc.) to refer to the open file. The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.
RETURN VALUE
open() returns the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

==

NAME read -read from a file descriptor
SYNOPSIS ssize_t read(int fd, void *buf, size_t count);
DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
RETURN VALUE
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately.

==

NAME write -write to a file descriptor
SYNOPSIS ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
RETURN VALUE
On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this number is smaller than the number of bytes requested; this may happen for example because the disk device was filled. On error, -1 is returned, and errno is set appropriately.

==

NAME close -close a file descriptor
SYNOPSIS int close(int fd);
DESCRIPTION
close() closes a file descriptor (for a regular file, a pipe or a socket), so that it no longer refers to any file and may be reused.
RETURN VALUE
close() returns zero on success. On error, -1 is returned, and errno is set appropriately.

==

NAME fork -create a child process
SYNOPSIS pid_t fork(void);
DESCRIPTION
fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
RETURN VALUE
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

==

NAME waitpid -wait for process to change state
SYNOPSIS pid_t waitpid(pid_t pid, int *wstatus, int options);
DESCRIPTION
waitpid() is used to wait for state changes in a child of the calling process. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

If pid $>$ 0, then the call will wait for the children whose PID equals pid.

If wstatus is not NULL, then waitpid() stores status in informations in the int it points to. If wstatus is NULL, then this parameter is ignored. The value of options is an OR of zero or more of the following constants : WNOHANG, WUNTRACED, WCONTINUED.
RETURN VALUE
On success, waitpid() returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

==

NAME getpid, getppid -get process identification
SYNOPSIS pid_t getpid(void);
pid_t getppid(void);
DESCRIPTION
getpid() returns the process ID (PID) of the calling process. getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork(), or, if that process has already terminated, the ID of the process to which this process has been reparented.
ERRORS
These functions are always successful.

==

NAME pipe -create pipe
SYNOPSIS int pipe(int pipefd[2]);
DESCRIPTION
pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see end-of- file and will return 0. If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write fails with the error EPIPE. An application that uses pipe and fork should use suitable close calls to close unnecessary duplicate file descriptors; this ensures that end-of- file and SIGPIPE/EPIPE are delivered when appropriate.
RETURN VALUE
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.