Corrected

Examen de LIFASR5, le 21/05/2019 — 1ère partie (≈ 45 minutes)

Toute forme de communication 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.

Consignes :

- Utilisez un **stylo à bille noir ou bleu**.
- A chaque question, **une seule réponse** est correcte : **noircir ou bleuir** la case correspondante, sans dépasser !
- Pour corriger (dernier recours) : effacez proprement la case.
- Ne pas oublier de noter votre **numéro de copie**.

Vous rendrez les 3 feuilles qui composent cette 1ère partie dans votre copie d'examen.

Numéro de votre copie d'examen :

- Notez le ici : .....
- Encodez le ci-contre (chiffre des unités tout à droite).

☐0	☐0	☐0	☐0	☐0	☐0	☐0	☐0
☐1	☐1	☐1	☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☐2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☐3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4	☐4	☐4	☐4
☐5	☐5	☐5	☐5	☐5	☐5	☐5	☐5
☐6	☐6	☐6	☐6	☐6	☐6	☐6	☐6
☐7	☐7	☐7	☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9	☐9	☐9	☐9

1 Questions diverses

Question 1 On considère la fonction définie ci-dessous, visant à envoyer sur une socket `s` les `n` octets (`n` > 0) situés à partir de l'adresse `p`.

```
int send_all(int s, const char *p, int n) {
  int r = n;
  while(r > 0) {
   int nbsent = send(s, p, r, 0);
   if(nbsent == -1) return -1;
   r -= nbsent;
   p += nbsent;
  }
  return 0;
}
```

Cochez l'affirmation correcte :

- ☐ la boucle while est nécessaire car send() peut échouer, et dans ce cas il faut reprendre tout l'envoi
- ☐ à chaque entrée dans la boucle while, r est le nombre d'octets déjà envoyés
- ■ la boucle while est nécessaire car send() est susceptible d'envoyer moins d'octets que le nombre demandé dans ses paramètres

Question 2 Parmi les primitives suivantes, laquelle crée un nouveau processus (en cas de succès) ?

☐ `execvp()` ☐ `waitpid()` ■ `system()`

Question 3 Cochez l'affirmation correcte. Un processus *zombie* est un processus :

☐ maudit ■ terminé ☐ en état d'erreur

CORRECTED

Question 4 Le prototype de la primitive `read()` est `ssize_t read(int fd, void *buf, size_t count)`. On suppose que `v` est un objet de type `vector<char>`, qui a été initialisé comme il se doit. Parmi les appels suivants à `read()`, lequel a une chance d'être compilé et de s'exécuter sans erreurs ?

- [x] `read(fd, v.data(), v.size() * sizeof(char));`
- [ ] `read(fd, &v, v.size() * sizeof(char));`
- [ ] `read(fd, (void*)v, v.size() * sizeof(char));`

Question 5 On considère le programme suivant.

```
int main(void) {
  int pid1 = fork();
  if(pid1 > 0) {
    int pid2 = fork();
    if(pid2 > 0) do_things();
    else do_things2();
  }
  else do_things1();
  return 0;
}
```

Cochez l'affirmation correcte. Dans ce programme, le processus principal crée

- [ ] un fils, qui lui même crée à nouveau un fils
- [ ] trois fils
- [x] deux fils

Question 6 En s'inspirant de l'exemple des chaînes de caractères en C, un programmeur a décidé d'utiliser l'octet 0 pour marquer la fin des blocs de données qu'il envoie sur le réseau, en mode TCP : il envoie les données (avec `send()`) octet par octet, puis envoie l'octet 0 ; à le réception, il reçoit les données (avec `recv()`) octet par octet jusqu'à recevoir l'octet 0. Ce programmeur s'aperçoit qu'il n'arrive pas à échanger entièrement certaines données binaires : pourquoi ?

- [ ] on ne peut pas recevoir l'octet 0 avec `recv()`
- [x] les données binaires qu'il envoie peuvent déjà contenir l'octet 0
- [ ] on ne peut pas envoyer 0 octet avec `send()`

Question 7 On considère le morceau de programme suivant :

```
int main(int argc, char *argv[]) {
  cout << argc << " ";
  for(int i=0; i<argc; i++)
    cout << argv[i] << " ";
  cout << endl;
  return 0;
}
```

On compile ce programme en l'exécutable `args`, et on le lance avec la commande `./args t o t o`. Le programme affiche (une réponse correcte) :

- [ ] `5 o t o t ./args`
- [x] `5 ./args t o t o`
- [ ] `4 t o t o`
- [ ] `4 ./args t o t`

CORRECTED

Question 8 Je cherche à utiliser le réseau en Rust. En regardant rapidement sur sur le web, je trouve différents exemples pour commencer... Parmi les lignes de code Rust ci-dessous, laquelle permet d'ouvrir une socket pour interroger un serveur à l'adresse 192.168.15.4 sur le port 8090, en TCP/IP :

- ☐ `let mut socket = UdpSocket::bind("192.168.15.4:8090")?;`
- ☐ `let listener = TcpListener::bind("192.168.15.4:8090").unwrap();`
- ■ `let stream = TcpStream::connect("192.168.15.4:8090").expect("failed");`

2 Petits exercices

2.1 Création de processus

On considère le morceau de programme suivant :

```
int main(void) {
  int n = 0;
  while(n < 3) {
    int pid = fork();
    if(pid == -1) exit(EXIT_FAILURE);
    if(pid > 0) {
      if(pid % 5 == 0) n++;
      else
        if(waitpid(pid, NULL, 0) == -1) exit(EXIT_FAILURE);
    }
    if(pid == 0) {
      if(getpid() % 5 == 0) sleep(10);
      exit(EXIT_SUCCESS);
    }
  }
  for(int i = 0; i < 3; i++)
    if(waitpid(-1, NULL, 0) == -1) exit(EXIT_FAILURE);
  exit(EXIT_SUCCESS);
}
```

Question 9 Dans ce programme, le processus principal crée

- ☐ exactement 3 fils
- ■ des fils jusqu'à en avoir 3 dont le PID est un multiple de 5
- ☐ des fils jusqu'à en avoir 5 dont le PID est un multiple de 3

Question 10 On rappelle que `sleep(10)` permet de mettre le processus appelant en sommeil pour 10 secondes. Dans ce programme,

- ☐ certains processus restent « longtemps » (plus d'une seconde) dans l'état *zombie*
- ☐ certains processus deviennent *orphelin* après le mort du père
- ■ le processus père attend le terminaison de tous ses fils avec `waitpid()`

Question 11 Le temps d'exécution du programme est :

☐ $\approx$ 30 s ☐ $\approx$ 20 s ■ $\approx$ 10 s ☐ très inférieur à 1 s

2.2 Encore des processus

On considère ce programme :

```
int main(void) {
  pid_t pid[6];
```

```
  for(int i=0; i<6; i++) {
    pid[i] = fork();
    if(pid[i] == -1) return EXIT_FAILURE;
    if(pid[i] == 0) {
      for(int j=0; j<i; j++) cout << pid[j] << " ";
      cout << endl;
      sleep(10);
      return EXIT_SUCCESS;
    }
  }
  for(int i=0; i<6; i++) {
    if(waitpid(pid[i], NULL, 0) == -1) return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Question 12 Dans ce programme,

- [x] le processus principal crée en tout 6 fils
- [ ] le processus principal crée en tout 12 fils
- [ ] le processus principal crée 1 fils, qui lui même crée un fils, ..., et cela 6 fois en tout

Question 13 Quel(s) processus exécute(nt) la dernière boucle `for` contenant les appels à `waitpid()` ?

- [x] seulement le processus père
- [ ] le dernier des fils créés
- [ ] les processus fils

Question 14 Le temps d'exécution du programme est :

- [ ] ≈ 2 min
- [ ] ≈ 1 min
- [ ] très inférieur à 1 s
- [x] ≈ 10 s

On donne ci-dessous un affichage produit par le programme :

```
nlouvet@nlbook:~$ ./fork

14203
14203 14204 14205
14203 14204
14203 14204 14205 14206
14203 14204 14205 14206 14207
nlouvet@nlbook:~$
```

Question 15 Quel est le PID du processus **père** ? (« inconnu » = « on ne peut pas savoir »)

- [ ] 14203
- [ ] 14206
- [ ] 14205
- [ ] 14207
- [x] inconnu
- [ ] 14204

Question 16 Quel est le PID du **premier** fils créé ? (« inconnu » = « on ne peut pas savoir »)

- [ ] inconnu
- [x] 14203
- [ ] 14204
- [ ] 14205
- [ ] 14206
- [ ] 14207

Question 17 Quel est le PID du **dernier** fils créé ? (« inconnu » = « on ne peut pas savoir »)

- [ ] 14206
- [ ] 14205
- [ ] 14204
- [ ] 14203
- [ ] 14207
- [x] inconnu

2.3 Un exemple de code avec read() et write()

Un programmeur a produit le code suivant :

```
int main(void) {
  ssize_t nbrd, nbwr;
```

```
  while((nbrd = read(STDIN_FILENO, buf, 1024)) > 0) {
    for(int i = 0; i < nbrd; i++) {
      while((nbwr = write(STDOUT_FILENO, buf+i, 1)) == 0);
      if(nbwr == -1) exit(EXIT_FAILURE);
    }
  }
  if(nbrd == -1) exit(EXIT_FAILURE);
  return EXIT_SUCCESS;
}
```

Question 18 Ce programme est bogué, et provoque soit un `segfault`, soit il n'affiche rien. On propose différentes corrections possibles ; cochez la seule qui puisse fonctionner. On peut remplacer

- ☐ `buf` par `&buf` dans l'appel à `read()`
- ☐ `buf+i` par `&buf[i]` dans l'appel à `write()`
- ■ `char *buf;` par `char *buf = new char[256]`, et ajouter `delete[] buf;` à la fin du programme

Question 19 Supposons que le programme soit maintenant débogué, et on appel `read-write` l'exécutable correspondant. Comment invoquer le programme en ligne de commandes, et quel va être le résultat ?

- ☐ `toto.txt | ./read-write` affiche le contenu du fichier `toto.txt` sur la sortie standard
- ■ `./read-write < toto.txt` affiche le contenu du fichier `toto.txt` sur la sortie standard
- ☐ `echo "toto" > ./read-write` affiche `toto` sur la sortie standard

Question 20 On envoie 747 octets sur l'entrée standard : combien de fois la boucle `while((nbrd = read(STDIN_FILENO, buf, 1024)) > 0) {...}` va t'elle s'exécuter pour tous les recevoir ?

■ on ne peut pas savoir ☐ 1 fois ☐ 2 fois

Question 21 On envoie 747 octets sur l'entrée standard : combien de fois l'appel `write(STDOUT_FILENO, buf+i, 1)` va t'il être exécuté au cours de l'exécution du programme ?

■ on ne peut pas savoir ☐ 380 fois ☐ 747 fois

2.4 Un exemple de code avec kill() et sigaction()

Un programmeur a produit le code suivant. Pour répondre aux questions, il faut savoir qu'un processus qui reçoit le signal `SIGSTOP` est interrompu jusqu'à ce qu'il reçoive le signal `SIGCONT` ; à la réception de `SIGCONT`, si la processus a mis en place un gestionnaire pour ce signal il est exécuté, puis le processus reprend son exécution.

```
void handler(int sig) {
  cout << "helli !" << endl << flush;
}

int main(void) {
  int i;

  struct sigaction sa;
  sigaction(SIGCONT, NULL, &sa);
  sa.sa_handler = handler;
  sigaction(SIGCONT, &sa, NULL);

  pid_t pid = fork();
  if(pid > 0) { // processus père
    for(i=0; i<3; i++) {
      sleep(1);
      kill(pid, SIGCONT);
    }
```

```
  }
  else { // processus fils
    for(i=0; i<3; i++) {
      kill(getpid(), SIGSTOP);
      cout << "hello !" << endl << flush;
    }
  }
  waitpid(pid, NULL, 0);
  return EXIT_SUCCESS;
}
```

Question 22 Dans ce programme, la variable `i` est partagée par le processus père et le processus fils (si l'un affecte la variable, l'autre trouve la valeur modifiée) :

☐ Vrai ■ Faux

Question 23 Dans le processus fils, la fonction pour la gestion du signal `SIGCONT` est appelée à chaque fois que le processus reçoit ce signal ?

■ Vrai ☐ Faux

Question 24 Quel est l'affichage produit par le programme à l'exécution ?

☐
```
Helli!
Helli!
Helli!
Hello!
Hello!
Hello!
```

☐
```
Hello!
Hello!
Hello!
Helli!
Helli!
Helli!
```

■
```
Helli!
Hello!
Helli!
Hello!
Helli!
Hello!
```

☐
```
Hello!
Helli!
Hello!
Helli!
Hello!
Helli!
```

Question 25 Le `waitpid(pid, NULL, 0);` est :

☐ bien placé : il permet ainsi au processus fils d'attendre la fin de son père avant de se terminer lui même.

■ mal placé : il devrait être placé à la fin du bloc indiqué par le commentaire `processus père` pour que le père attende la fin de son fils avant de se terminer.

☐ mal placé : il devrait être placé à la fin du bloc indiqué par le commentaire `processus fils` pour que le fils attende la fin de son père avant de se terminer.

☐ bien placé ; il permet ainsi au processus père d'attendre la fin du fils avant de se terminer lui même.

Question 26 Dans ce programme, les valeurs de retour des appels systèmes ne sont pas testées. Cochez l'affirmation correcte.

☐ ce n'est pas gênant, on peut tout de même déboguer le programme, car les appels sytèmes affichent eux-même des messages d'erreurs

☐ en cas d'erreur au niveau d'un appel système, le programme se termine forcément, donc on comprend tout de suite qu'une erreur s'est produite

■ cela risque de rendre le débogage difficile, car en cas d'erreur, on ne peut pas toujours savoir au niveau de quel appel elle se produit

2.5 Un code avec pipe()

On considère le programme suivant, qui est parfaitement fonctionnel.

```
int main(void) {
  int res, pipefd[2];
  char c;
```

CORRECTED

```
  if(pipe(pipefd) == -1) exit_error();

  int pid = fork();
  if(pid == -1) exit_error();

  if(pid == 0) { // processus fils
    cout << "(FILS) j'ai pour PID " << getpid() << endl;
    close(pipefd[0]);
    for(c = 'a'; c < 'e'; c++) {
      if(write(pipefd[1], &c, 1) == -1) exit_error();
      sleep(1);
    }
    close(pipefd[1]);
    cout << "(FILS) je me termine" << endl;
    return EXIT_SUCCESS;
  }
  // processus père
  cout << "(PERE) je suis à l'écoute" << endl;
  close(pipefd[1]);
  while((res = read(pipefd[0], &c, 1)) == 1)
    cout << "(PERE) j'ai reçu : " << c << endl;
  if(res == -1) exit_error();
  close(pipefd[0]);
  if(waitpid(pid, NULL, 0) == -1) exit_error();
  cout << "(PERE) je me termine" << endl;
  return EXIT_SUCCESS;
}
```

Question 27 Quand on appelle `pipe()` sur un tableau d'entiers `pipefd` à deux élements, `pipefd[0]` permet :

■ la lecture ☐ l'écriture ☐ les deux

Question 28 Quand on appelle `pipe()` sur un tableau d'entiers `pipefd` à deux élements, `pipefd[1]` permet :

☐ les deux ☐ la lecture ■ l'écriture

Question 29 Donnez un résultat d'affichage possible (une trace) du programme.

☐
```
(PERE) je suis à l'écoute
(FILS) j'ai pour PID 10280
(PERE) j'ai reçu : a
(PERE) j'ai reçu : b
(PERE) j'ai reçu : c
(PERE) j'ai reçu : d
(PERE) je me termine
(FILS) je me termine
```

■
```
(FILS) j'ai pour PID 10280
(PERE) je suis à l'écoute
(PERE) j'ai reçu : a
(PERE) j'ai reçu : b
(PERE) j'ai reçu : c
(PERE) j'ai reçu : d
(FILS) je me termine
(PERE) je me termine
```

☐
```
(PERE) je suis à l'écoute
(FILS) j'ai pour PID 10280
(PERE) j'ai reçu : d
(PERE) j'ai reçu : e
(PERE) j'ai reçu : a
(PERE) j'ai reçu : b
(FILS) je me termine
(PERE) je me termine
```