M1 Systèmes Contrôle de TD — Prénom: _____
12 novembre 2009 — Nom: _____

Durée 50 minutes

Ecrivez vos réponses directement sur cette feuille. Vos notes de TD/TP/cours sont autorisés.

Exercice 1: QCM

Cochez les réponses correctes (peut-être plusieurs)
Pour chaque case cochée: 0.5 correct, -0.25 incorrect. Total = max(somme,0).

1. Un appel à `fork()` retourne 0
   ❍ dans le père ● dans le fils ❍ dans les deux ❍ en cas d'erreur
2. Au cours de sa vie un processus non privilégié peut changer de
   ● session ● terminal ● groupe de processus ❍ PID
   ❍ PPID ❍ UID réel ● UID effectif
   Voir la page `man credentials`
3. Une session
   ❍ a toujours un terminal: non, par exemple à sa création une session n'a pas de terminal de contrôle
   ● a au plus un groupe de processus en avant-plan
   ❍ a au plus un groupe de processus en arrière-plan
   ● est créée par son leader
4. Processus appelant les primitives `wait, sigsuspend` ou `read` se met en état
   ❍ suspendu (stopped) ● endormi
   ❍ zombi ❍ prêt
5. Un processus zombi
   ❍ peut tuer son père ❍ peut seulement recevoir `SIGCHLD`
   ● devient orphelin quand son père se termine ● disparaît quand son père fait waitpid
6. L'exécution de `execlp("echo","-n","marron",NULL); printf("vert")`
   ❍ affiche "`vert`" ❍ affiche "`marronvert`"
   ● affiche "`marron`" ❍ n'affiche rien
   Quand l'appel à execlp réussit (implicite dans la question), la commande shell "echo -n marron" recouvre le code du programme et affiche "marron". Sinon, le programme se poursuivrait et afficherait "vert".
7. Si l'appel à `open("test4",O_WRONLY|O_CREAT|O_EXCL,0640)` retourne une valeur différente de -1, alors le fichier `test4` est créé et on sait que
   ● il n'existait pas avant : sinon l'appel échouerait à cause du flag O_EXCL
   ❍ tout le monde peut le lire: non, pas les utilisateurs en dehors du groupe du propriétaire
   ❍ il est exécutable: non, aucun des bits x des droits n'est positionné
   ● `stat("test4",&s); assert(s.st_uid == getuid())` réussira
   Oui, le propriétaire du processus devient le propriétaire des fichiers créés. Cependant dans les rares cas où les propriétaires réel et effectif du processus sont différents l'assertion va échouer, il serait donc plus exact de comparer avec `geteuid()`
8. Un processus qui a fait `pipe(t); close(t[0]); write(t[1],buf,strlen(buf))`
   ❍ est bloqué pour toujours
   ● ne peut plus jamais lire sur le tube
   ● reçoit le signal `SIGPIPE`
   ❍ est bloqué jusqu'à ce que les `strlen(buf)` caractères sont écrits
   Ici on essaie d'écrire dans un tube qui n'a plus de lecteurs – et on sait qu'il n'y en aura plus jamais – alors le signal SIGPIPE est envoyé au processus (par défaut, le processus meurt)
9. Après l'appel à `dup2(open("test",O_RDWR),1)`
   ● la sortie standard est redirigée vers le fichier `test`
   ❍ l'écriture dans `test` est redirigée vers la sortie standard
   ❍ descripteur de fichier pour `test` est fermé

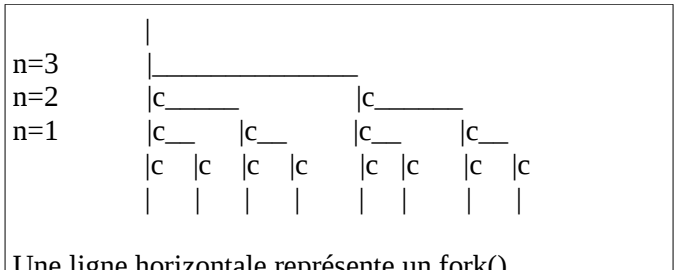
❍ le processus n'a plus de terminal

10. Un processus peut attendre l'arrivée d'un signal en utilisant la fonction
    ● `pause` ❍ `wait` ● `sigsuspend` ❍ `sigprocmask`
    `wait` n'est pas une réponse tout à fait fausse, car il permet d'attendre un signal particulier, SIGCHLD.
11. Pendant l'exécution du traitement d'un signal, un autre signal du même type
    ❍ est ignoré ❍ ne peut pas être envoyé par d'autre processus
    ❍ est traité en priorité ● est bloqué
12. Un signal envoyé à un processus stoppé qui ne bloque pas ce signal sera délivré
    ● à son réveil ❍ s'il appelle `sigpending`
    ❍ jamais ❍ immédiatement
    Un processus stoppé est terminé par SIGKILL ou SIGTERM, réveillé par SIGCONT.
    Les autres signaux non bloqués sont stockés et délivrés au réveil.
13. Après l'exécution du code `sigemptyset(&set); sigaddset(&set,SIGALRM); sigprocmask(SIG_BLOCK,&set,NULL)`, le processus
    ❍ bloque tous les signaux sauf `SIGALRM` ❍ attend `SIGALRM`
    ❍ bloque seulement `SIGALRM` ● bloque `SIGALRM`
    Les signaux se trouvant dans ce masque sont ajoutés (union) aux signaux déjà bloqués par le processus.

Exercice 2: Création de processus (4pts)

Pour l'appel de la fonction `forks(3,'c')`, dessiner l'arborescence des processus créés et donner leur nombre. Quelle est la taille du fichier créé?

```
void forks(int n,char c){
  int fd = open("exo2.txt",
  O_WRONLY|O_CREAT|O_TRUNC,0600);
  for(;n>0;n--){
        fork();
        write(fd,&c,1);
  }
  close(fd);
}
```



Une ligne horizontale représente un fork()

A chaque tour de boucle le nombre de processus double: un appel à `forks(n,'c')` crée donc 2^n processus, en particulier `forks(3,'c')` crée 8 processus, dont 7 via fork. Après chaque fork, chaque processus écrit un caractère dans le fichier `exo2.txt` dont le descripteur est partagé entre tous ces processus: les écritures se font donc l'une après l'autre, et la taille du fichier étant 0 au départ (`O_CREAT|O_TRUNC`), il y a exactement 2 * (nombre de fork) = 2*(1+2+4) =14 octets.

Exercice 3 **Synchronisations**

Nous avons vu plusieurs façons de synchroniser des processus, le principe restant le même: pour qu'un processus B s'exécute *après* le processus A, B doit appeler une fonction bloquante, tandis que A doit débloquer B après s'être exécuter. Dans cet exercice, il faut comprendre l'ordre d'exécution de différentes instructions d'un programme, puis refaire les mêmes synchronisations mais en utilisants des moyens différents.

Dans le programme initial les processus sont synchronisés avec les **tubes anonymes.**

1. Quels sont les affichages produits par l'exécution de ce programme? (4pts)

```
  char buf[1];
  int tube[2], tube2[2];
  pipe(tube2);
  pipe(tube);

Switch (fork ()){
  case -1: perror("fork"); exit(-1);
  case 0: {
        read(tube[0],buf,1);
        printf("ca\n");
        write(tube2[1],"1",1);
        exit(0);
  }
  default:
      printf("wow\n");
     write(tube[1],"1",1);
      read(tube2[0],buf,1);
      printf("compile\n");
      wait(NULL);
      return 0;
    }
}
```

Le programme affiche :

wow
ca
compile

L'ordre d'exécution des processus est donc:
père – fils – père.

2. On veut réécrire ce programme, sans utiliser de tubes, en assurant la même synchronisation des processus. Le nouveau programme devra produire les mêmes affichages.
Compléter ce code pour implémenter la même synchronisation avec

1. la fonction de mise en sommeil `sleep` (2pts)
2. le handler du signal `SIGUSR1` (5pts)

Voir le fichier controle-exo3.c

3. Pourquoi on ne peut pas faire la même synchronisation en utilisant uniquement des `wait`? (2pts)

Parce que wait ne permet pas à un fils d'attendre le père. En utilisant uniquement des wait on pourrait donc garantir que “compile” s'affiche en dernier, mais on ne pourrait pas ordonner “ca” et “compile”.