Question 1 : (11 pts) Généralités

Répondez aux questions à choix multiples en sélectionnant une ou plusieurs réponses. Les questions font référence aux systèmes d'exploitation de la famille *UNIX* et considèrent les options par défaut des appels système.

1) [1 pt] Un processus *zombie* est un processus qui :
a) a perdu son père et n'a plus de père.
b) a terminé son exécution en erreur.
c) a terminé son exécution et attend la prise en compte de cette fin par son père.
d) a perdu son père et a été adopté par le processus *init*.
e) aucune de ces réponses.

2) [1 pt] Si chaque processus père attend la fin de tous ses fils avant de se terminer, il n'y aurait aucun processus :
a) zombie dans le système.
b) qui bloque son père.
c) adopté par le processus *init*.
d) bloqué par son père.
e) aucune de ces réponses.

3) [1 pt] Quels sont les éléments partagés par l'ensemble des threads d'un processus ? Ils partagent :
a) l'espace d'adressage.
b) la table de descripteurs de fichiers.
c) le compteur ordinal.
d) la pile d'exécution.
e) aucun des éléments ci-dessus.

4) [1 pt] On veut faire communiquer deux threads *utilisateur* d'un même processus via un tube anonyme (*pipe*). À quel niveau doit-on créer le tube anonyme ? Le tube anonyme doit être créé :
a) avant la création du premier thread.
b) après la création du premier thread et avant la création du second thread.
c) après la création du second thread.
d) dans chacun des deux threads.
e) **aucune de ces réponses car les threads ne pourront pas communiquer en utilisant les fonctions « *read* » et « *write* ».**

5) [1 pt] On veut faire communiquer deux threads utilisateur d'un même processus via un tube nommé. À quel niveau doit-on ouvrir le tube nommé ? Le tube nommé doit être ouvert :
a) avant la création du premier thread.
b) dans le premier thread.
c) dans le second thread.
d) dans chacun des deux threads.
e) aucune de ces réponses car les threads ne pourront pas ouvrir le tube.

6) [1 pt] Si un processus appelle la fonction « *execlp* », quels sont les éléments qu'il va préserver ? Il préserve :
a) l'espace d'adressage.
b) **la table des descripteurs de fichiers.**
c) les liens père-fils avec les autres processus.
d) la pile d'exécution.
e) **le *PID*.**
f) aucun des éléments ci-dessus.

7) [1 pt] Le masque des signaux d'un processus indique quels signaux, à destination du processus, à :
a) ignorer.
b) capter.
c) traiter en priorité en appliquant le traitement par défaut.
d) **conserver pour les traiter ultérieurement.**
e) aucune des réponses ci-dessus.

8) [2 pts] Le nombre de processus créés par l'instruction « *while (pid=fork()) if (n >= 5) break ; else n=n+1;*», où *n* est un entier initialisé à *0*, est :
a) *4.*
b) *5.*
c) *6.*
d) *>10*.
e) aucune des réponses ci-dessus.

9) [1 pt] Considérez le programme suivant :

```
bool Continuer = true;
void action (int sig)
{  printf("reception de %d\n", sig);
   if (sig==SIGUSR1) Continuer =false;
}
int main ()
{  signal(SIGUSR1, action);
   signal(SIGINT, action);
   while (Continuer)
  {      kill(getpid(), SIGINT);
         sleep(1);
         kill(getpid(), SIGUSR2);
         sleep(1);
         kill(getpid(), SIGUSR1);
   }
   exit(0);
 }
```

Le processus va être terminé par :
a) le signal SIGUSR2.
b) le signal *SIGINT*.
c) le signal *SIGUSR1*.
d) l'instruction « *exit(0)* ».
e) aucune des réponses ci-dessus.

10) [1 pt] Considérez la commande « *cat* | *sort* > *data* ». La sortie standard du processus exécutant *cat* est :
a) le fichier *data*.
b) le pipe.
c) l'écran.
d) le clavier.
e) aucune des réponses ci-dessus.

Question 2 (4.5 pts) : Processus & redirection des E/S standards

1) **[3 pts] Complétez le code ci-dessous** pour qu'il réalise le traitement suivant : le processus principal incrémente la variable *v*, crée deux processus fils *F1* et *F2*, puis se met en attente de la fin de ses fils. Chaque fils *Fi*, pour i=*1,2*, crée un fils *Fi1*, incrémente la variable *v*, puis se met en attente de la fin de son fils. Enfin, chaque petit fils *Fi1*, pour *i=1,2*, affiche à l'écran son numéro, celui de son père et la valeur de *v*, puis se termine. **Indiquez la valeur de *v*** affichée à l'écran par chaque petit fils.

2) **[1.5 pt]** Complétez le code en 1) afin que les affichages à l'écran (réalisées par les petits fils) soient redirigées vers un même fichier nommé data, créé par le programme.

```
int v=10 ;
int main()
{     int i ;
      printf("processus %d \n", getpid()) ;
      v=v+1 ;
      int fd = open("data", O_CREAT|O_WRONLY); // pour 2)
      for(i=0 ; i<2 ; i++)
      {  if (fork() == 0) // F1 ou F2
         {   if (fork()==0)  // F11 ou F21
             {
                dup2(fd,1) ; close(fd) ; // pour 2)
                printf("proc. %d, de père %d, v=%d \n", getpid(), getppid(), v) ;
                exit(0) ;
              }
              v=v+1 ;
              close(fd) ; // pour 2)
             wait(NULL) ; exit(0) ;
          }
        }
        close(fd) ; // pour 2)
         wait(NULL) ; wait(NULL) ;
         exit(0) ;
    }
```

Les deux petits fils afficheront la même valeur de v : 11

Question 3 (4.5 pts) : Synchronisation des processus

Considérez un système de réservation de billets d'un spectacle. Ce système de réservation est composé de trois fonctions : *int get_free()*, *bool book()* et *bool cancel()*. La fonction *get_free* permet de récupérer le nombre de places encore disponibles pour le spectacle. La fonction *book* permet de réserver une place pour le spectacle. La fonction *cancel* permet d'annuler une réservation pour le spectacle. On vous fournit le code suivant de ces trois fonctions :

```
// variables globales
int free=N ;

int  get_free () { return free; }

bool book ( )
{
   if (free ==0) return false;
   free = free – 1; return true;
 }

bool cancel ( )
{    if (free<N)  {  free = free + 1; return true;}
    return false;
 }
```

La gestion de la réservation de billets pour le spectacle est prise en charge par un processus. Toutes les demandes des utilisateurs concernant ce spectacle sont dirigées vers ce processus. Ce processus crée un thread pour chaque demande reçue. Ce thread se charge de traiter la demande en faisant appel aux fonctions *get_free, book* ou/et *cancel*.

Un code (ou une fonction) est dit « *safe-thread* » s'il est capable de fonctionner correctement lorsqu'il est exécuté simultanément par plusieurs threads d'un même processus.

1) **[2.5 pts] Indiquez pour chacune des fonctions** *get_free, book* et *cancel* si elle est « safe-thread » ? Si vous répondez non, expliquez pourquoi par un exemple puis corrigez son code, en utilisant les sémaphores (utilisez le type *Semaphore* et les primitives *P* et *V)*. Si vous répondez oui, expliquez pourquoi en montrant que l'exécution concurrente de la fonction préserve les variables partagées dans un état cohérent.

2) **[2 pts]** On veut maintenant mettre en attente, en utilisant des sémaphores, toute demande de réservation qui ne peut être satisfaite, jusqu'à ce qu'une place se libère. **Donnez le code des trois fonctions** qui tient compte de cette directive et qui est aussi « safe-thread » (utilisez le type *Semaphore* et les primitives *P* et *V)*.

1) La fonction get_free est « safe-thread » car elle consiste en une lecture de la variable partagée free. Elle ne la modifie pas. Les fonctions book et cancel ne sont pas « safe-thread » car elles lisent la valeur de la variable partagée free, la modifient puis rangent le résultat dans free. Si elles sont exécutées en concurrence, la valeur de free pourrait ne pas correspondre au résultat attendu. En effet, si deux threads th1 et th2 lisent tous les deux la même valeur courante de free puis chacun incrémente ou décrémente cette valeur avant de la copier dans free, le dernier rangement dans free va inhiber le résultat de l'autre opération.

```
// variables globales
int free=N ;
Semaphore mutex=1 ;  // pour empêcher les accès simultanés aux sections critiques
int  get_free () { return free; }
bool book ( )
{
   P(mutex) ;
   if (free ==0)    { V(mutex) ;  return false; }
   free = free – 1; V(mutex) ;   return true;
 }
bool cancel ( )
{   P(mutex) ;
  if (free<N)  {  free = free + 1; V(mutex) ;   return true;}
  V(mutex) ;     return false;
 }
```

2)

```
// variables globales
int free=N ;
Semaphore mutex=1, libre = N ;
 //mutex pour empêcher les accès simultanés aux sections critiques.
// libre pour bloquer l'appelant à book s'il n'y a pas de places disponibles.
int  get_free () { return free; }
bool book ( )
{     P(libre) ;
     P(mutex) ;
   free = free – 1;
  V(mutex) ;
  return true;
 }
bool cancel ( )
{   P(mutex) ;
  if (free<N)  {  free = free + 1; V(mutex) ;   V(libre) ; return true; }
  V(mutex) ;     return false;
 }
```