

IF-3-SYS : Examen du 11 juin 2019

Q1. D=B (car fork n'a pas changé l'adresse virtuelle de x) mais C = A + 10 (car les deux processus ont chacun un exemplaire privé de x).

Q2.

```
void chaine(N)
{
    if(N == 0)
        return;
    int r = fork()
    if(r == 0) // child
        chaine(N-1);
}
```

ou

```
void chaine(N)
{
    while(N>0)
    {
        int r = fork();
        if(r != 0) // parent
            return;
        N--;
    }
}
```

(ou autre variante)

Q3. cache et TLB = Faux (les données existent en RAM, on pourra les recharger la prochaine fois).
contenu du CPU = vrai (c'est ça le «contexte d'exécution» sur lequel travaille le *dispatcher*)

Q4. Faux ; Faux ; Vrai ; Vrai. (cf cours chap 2)

Q5.

time	0	4	9	12	17	18	19	24	29	31
CPU	A	B	C	A	B	A	C	B	C	

Q6. A=19, B=26, C=26

Q7. 12 bits d'adresse avec des pages de $512=2^9$ octets \Rightarrow VPN et PPN codés sur $12 - 9 = 3$ bits.

VA	789	4A5	5A5	31F
VA en binaire	0111 1000 1001	0100 1010 0101	0101 1010 0101	0011 0001 1111
PO (9 bits)	1 1000 1001	0 1010 0101	1 1010 0101	1 0001 1111
VPN (3 bits)	011=3	010=2	010=2	001=1
PPN = PT[VPN]	PT[3]=0	PT[2]=7	PT[2]=7	PT[1]=4
PPN en binaire	000	111	111	100
PA	0001 1000 1001	1110 1010 0101	1111 1010 0101	1001 0001 1111
PA en hexa	189	EA5	FA5	91F

Q8.

```
count(N)
{
    unsigned int r = 0;
    while(N>0)
    {
        if (N & 1)
            r++;
        N >>= 1;
    }
    return r;
}
```

ou

```
count(N)
{
    if(N==0)
        return 0;

    int b = N & 1;

    return b + count(N>>1);
}
```

(ou autre variante)

Q9. File Allocation Table, File Descriptor, Hard Disk Drive, Solid State Drive.

Q10. 1 Tio = 2^{40} octets, 1 secteur = 2^9 octets, donc il y a $2^{42-9=33}$ secteurs \Rightarrow 33 bits.

Q11. Faux (deadlock si $A=P(X)$ puis $C=P(Z)$) ; Vrai (lock ordering : $Y < X < Z < T$) ; Faux (deadlock si $A=P(Y)$ puis $B=P(Z)$) ; Vrai (deadlock si $A=P(X)$ puis $C=P(Z)$)

Q12. L'exercice est le Search-Insert-Delete, emprunté au "little book of semaphores" §6.1

variables partagées:

```
EcritureMutex = semaphore(1) // autoriser au max un seul thread à modifier
LecteurMutex  = semaphore(1) // protéger l'accès à la variable nbLecteurs
entier nbLecteurs = 0        // compteur du nombre de lectures en cours
LecteurSwitch = semaphore(1) // pour l'exclusion entre Lecteurs et Enleveurs
```

Ajouteur:

```
P(AjouteurMutex) // empêche plusieurs Ajouteurs d'entrer simultanément
                  // empêche aussi l'ajout si un Enleveur est en cours
INSERTION()

V(AjouteurMutex) // rendre le mutex
```

Enleveur:

```
P(AjouteurMutex) // empêche plusieurs Enleveurs d'entrer simultanément
                  // empêche aussi la suppression si un Ajouteur est en cours
P(LecteurSwitch) // possible seulement quand aucun lecteur n'est en cours

SUPPRESSION()

V(LecteurSwitch) // rouvrir l'accès aux lecteurs
V(AjouteurMutex) // et aussi aux Ajouteurs (et aux autres Enleveurs)
```

Lecteur:

```
// avant_lecture()
P(LecteurMutex) // début de section critique
nbLecteurs++
if(nbLecteurs == 1) // si je suis le premier lecteur à rentrer
    P(LecteurSwitch) // alors je fais passer le switch à zéro
V(LecteurMutex) // fin de section critique

LECTURE()

// apres_lecture()
P(LecteurMutex) // début de section critique
nbLecteurs--
if(nbLecteurs == 0) // si je suis le dernier lecteur à sortir
    V(LecteurSwitch) // alors je fais passer le switch à 1
V(LecteurMutex) // fin de section critique
```