IF-3-SYS : Examen du 8 juin 2023

Q1. Vrai ; Faux (en termes techniques, la phrase ne veut rien dire) ; Vrai (par exemple signal kill) ; Vrai

Q2. 7 fois

Q3. Bloquants : `sleep()`, `read()`, `write()`, `wait()`, `sem_wait()`...
Non-bloquants : `fork()`, `getpid()`, `open()`, `exec()`, `sem_post()`...

Q4. Parmi les stratégies vues en cours (FCFS, SJF, SRTF, RR) seule *Round Robin* est immunisée contre la famine.
Note : L'ordonnancement à priorités (par ex MLFQ) n'est pas une stratégie en soi. C'est simplement l'idée de combiner plusieurs stratégies.

Q5.
```
time 0    1    2   3    6   7    9  10   14  15   17  18   22  23   25
 CPU |  A |idle| B |idle| A |idle| B |idle| A |idle| B |idle| A |idle|
```

Q6. Les processus voient une mémoire plus vaste mais plus lente à cause des fautes de page.

Q7. Faux (plutôt des micro-secondes) ; Faux (plutôt des centaines de nano, c'est pour ça qu'on a besoin d'un cache sur le CPU) ; Vrai (le CPU est cadencé en GHz, il fait en moyenne une ou plusieurs instructions par cycle) ; Vrai (grâce au TLB)

Q8. les sections `.text` et `.data` sont initialisées avec le contenu du fichier exécutable, mais `.stack` et `.heap` sont créées à la volée par le noyau.

Q9. «Avantages» de malloc :
- performance en temps de l'allocation : pas d'appel système, recyclage des blocs libérés
- moins de fragmentation interne (vs pages entières)
- malloc ne perd pas de temps à effacer les pages
- allocateur détecte/signale les erreurs d'exécution (e.g. double free)
- portabilité : malloc est dans l'API standard du C (vs mmap, dispo sous unix seulement)
- interface plus simple pour le programmeur (un seul paramètre)

«Avantages» de mmap :
- implémentation plus simple (pas de freelist à parcourir)
- plus direct (pour les grandes tailles, malloc va appeler mmap de toutes façons)
- possible de partager avec les processus enfants (cf TP crible)
- possible de protéger certaines régions, en lecture seule ou en exécution etc.
- un peu hors sujet : possible de mapper des fichiers du disque (cf TP prénoms)

Q10. L'espace d'adressage est partagé, le reste est privé (par définition).
Attention, dans ce contexte «les registres CPU» signifie «le contenu des registres du CPU». En effet, deux threads peuvent utiliser leur registre R1, mais ils ne verront pas le même contenu dedans, que ce soit à cause du parallélisme matériel (i.e. il y a pysiquement deux choses qui s'appellent R1 sur la machine) ou à cause du multitâche (i.e. le noyau virtualise R1, et chaque thread croit avoir R1 pour lui tout seul). Dans tous les cas, nos deux threads ne «partagent» pas R1.

Q11. Plein de solutions possibles. par exemple :
```
Sem X=2, Y=0
A : while(true) { P(X) P(X) print(A) V(Y) V(Y) }
B : while(true) { P(Y)      print(B) V(X)      }
```

Q12. Trois fichiers : [0], [9, 12, 4] et [5, 15, 8, 10].

Q13.
```
int count_free_blocks_at(int pos)
{
    int start=pos;
    while(pos<NB_BLOCKS && fat[pos] == FAT_FREE)
        pos++;
    return pos-start;
}
```

Q14.
```
int find_hole_ff(int n)
{
    for(int pos=0; pos<NB_BLOCKS; pos++)
    {
        int len = count_free_blocks_at(pos);
        if( len >= n )
            return pos;
    }
    return -1;
}
```

Q15.
```
int find_hole_bf(int n)
{
    int best_start = -1;
    int best_len = NB_BLOCKS; // or INT_MAX etc

    for(int pos=0; pos<NB_BLOCKS; pos++)
    {
        int len = count_free_blocks_at(pos);
        if( len >= n && len < best_len )
        {
            best_start = pos;
            best_len = len;
            pos += fb; // skip current hole
        }
    }
    return best_start;
}
```

Attention : dans cette question il est nécessaire de faire le `pos+=fb`, au lieu de simplement incrémenter `pos`, sous peine de confondre la «fin» d'un trou trop grand avec un trou pile de la bonne taille...