

## 1 Gestion Mémoire et pagination

**Question 1 (échauffement)** Sur un système 32 bits manipulant des pages de 4Ko, voici typiquement à quoi ressemble une entrée de la table des pages :

```
struct page_table_entry {
    num_phys_page : 20;
    valid         : 1;
    read          : 1;
    write         : 1;
    access        : 1;
    dirty         : 1;
};
```

Voici le rôle des différents champs :

**num\_phys\_page** contient le numéro de la page physique (sur 20 bits dans cet exemple). Ce champ est positionné par le système, et consulté par le matériel (MMU).

**valid** indique si l'entrée est valide, c'est-à-dire si la page existe. Ce champ est positionné par le système et consulté par le matériel (MMU).

**read** indique si les lectures sont permises sur la page. Positionné par le système et consulté par le matériel (MMU).

**write** similaire à *read*.

**access** est positionné par le matériel à chaque fois que la MMU effectue un accès à la page. Ce champ est régulièrement réinitialisé à 0 par le système pour rafraîchir ses statistiques d'accès aux pages.

**dirty** est positionné par le matériel à chaque fois que la MMU effectue un accès en écriture à la page. Ce champ est consulté (et modifié) par le système afin de déterminer si une page a besoin d'être sauvée sur disque avant d'être évincée de la mémoire (en cas de swap par exemple).

## Gestion mémoire dans Nachos

### Question 2

```
AddrSpace::AddrSpace (OpenFile * executable, bool forking)
{
    ...
    if(forking)
        numPages = currentThread->space->numPages;
    else
        numPages = divRoundUp (size, PageSize);
    ...
    for (i = 0; i < numPages; i++) {
        pageTable[i].physicalPage = frameProvider->GetEmptyFrame();
        if(forking)
            memcpy(&mainMemory[pageTable[i].physicalPage * PageSize],
                &mainMemory[currentThread->space->pageTable[i].physicalPage * PageSize],
                PageSize);
        pageTable[i].valid = forking ? currentThread->space->pageTable[i].valid : TRUE;
        pageTable[i].readOnly = forking ? currentThread->space->pageTable[i].readOnly : FALSE;
        ...
    }
}
```

**Question 3** Le mécanisme appelé «*Copy-on-Write (CoW)*» est un mécanisme d'allocation paresseuse utilisé lors d'un appel système *Fork* : au lieu d'allouer de nouvelles pages physiques pour le processus fils, l'idée est de forcer un partage «temporaire» des pages entre le processus père et le processus fils, en désactivant le droit d'écriture dans les pages pour chacun des processus. Bien entendu, le noyau maintient une structure de donnée annexe qui lui permet de mémoriser quels sont les droits d'accès «réels» à chacune des pages. Lors du déclenchement d'une interruption suite à une tentative d'écriture, le noyau peut donc distinguer une situation de *CoW* d'une erreur d'accès imputable au programme en vérifiant les droits «réels» dans cette structure. S'il s'agit d'un *CoW*, le système doit alors effectuer une allocation de page physique afin de donner à chacun des processus sa propre version indépendante...

**Question 4**

<pre>class FrameProvider { public:      int GetEmptyFrame()     {         int frame = bitmap-&gt;Find();          if (frame != -1)             bzero(mainMemory + ... ); // clear page          IncRefCount(frame);          return frame;     }      void ReleaseFrame(int frame)     {         DecRefCount(frame);     }      unsigned RefCountValue(int frame)     {         return refCount[frame];     } };</pre>	<pre>void IncRefCount(int frame) {     refCount[frame]++; }  void DecRefCount(int frame) {     if(--refCount[frame] == 0)         bitmap-&gt;Clear(frame); }  FrameProvider () // Initialization {     bitmap = new BitMap(NumPhysPages);     refCount = new unsigned(NumPhysPages);     for(int i = 0; i &lt; NumPhysPages; i++)         refCount[i] = 0; }  private:     BitMap *bitmap;     unsigned *refCount; };</pre>
--	---

**Question 5**

```
AddrSpace::AddrSpace (OpenFile * executable, bool forking)
{
    ...
    for (i = 0; i < numPages; i++) {
        if(forking) {
            pageTable[i].physicalPage = currentThread->space->pageTable[i].physicalPage;
            pageTable[i].valid = currentThread->space->pageTable[i].valid;
            pageTable[i].readOnly = TRUE;
        } else
            ...
    }
}
```

**Question 6** À chaque fois que l'un des 3 processus va tenter une écriture vers la page, une interruption sera déclenchée. Pour les deux premiers processus, le traitement d'interruption devra allouer une nouvelle page, et y recopier le contenu de la page partagée. Le dernier processus s'apercevra, en consultant le compteur de référence sur la page, qu'il est le dernier à y avoir accès, et donc il lui suffira de repositionner les bons droits directement sur la page.

**Question 7**

```

void
ExceptionHandler (ExceptionType which)
{
    if (which == ReadOnlyException) {
        int address = machine->ReadRegister (BadVAddrReg);
        int frame = address / PageSize;
        int physFrame = machine->pageTable[frame].physicalPage;

        if(frameProvider->RefCountValue(physFrame) > 1) {
            // the page is still shared by at least 2 processes
            // so we have to allocate a new frame, and proceed with the copy

            int newFrame = frameProvider->GetEmptyFrame();

            memcpy(&mainMemory[newFrame * PageSize],
                &mainMemory[physFrame * PageSize],
                PageSize);

            machine->pageTable[frame].physicalPage = newFrame;

            frameProvider->DecRefCount(frame);
        } else {
            // Current thread is the unique owner of the page
            // So we just have to revert the access rights back to "readable"

            machine->pageTable[frame].readOnly = FALSE;
        }
    }
}

```

## 2 Synchronisation

```
/* code à écrire */
typedef struct {
    mutex_t m;
    cond_t cl, cw;
    unsigned nbl; // initially 0
    unsigned nbw; // initially 0
} rwlock_t ;

void rwl_readlock(rwlock_t *l)
{
    mutex_lock(&l->m);
    while(l->nbw > 0)
        cond_wait(&l->cl, &l->m);
    l->nbl++;
    mutex_unlock(&l->m);
}

void rwl_readunlock(rwlock_t *l)
{
    mutex_lock(&l->m);
    l->nbl--;
    if(l->nbl == 0)
        cond_signal(&l->cw);
    mutex_unlock(&l->m);
}

void rwl_writelock(rwlock_t *l)
{
    mutex_lock(&l->m);
    while( (l->nbl + l->nbw) > 0)
        cond_wait(&l->cw, &l->m);
    l->nbw++;
    mutex_unlock(&l->m);
}

void rwl_writeunlock(rwlock_t *l)
{
    mutex_lock(&l->m);
    l->nbw--;
    cond_bcast(&l->cl);
    cond_signal(&l->cw);
    mutex_unlock(&l->m);
}
```