Question 3 On considère maintenant la table des pages d'un processus au sein duquel les pages virtuelles dont les numéros sont compris entre 30 et 65 sont toutes invalides. Lors d'un accès du processus à la page numéro 31, combien la MMU devra-t-elle faire d'accès mémoire avant de réaliser que la page est invalide ? Même question pour la page numéro 32.

Question 4 Si l'on suppose que les pages invalides sont toutes contigues et commencent dès la page numéro 0, calculez à partir de quel nombre de pages invalides cette organisation à trois niveaux consommera moins de mémoire qu'une table monolithique.

3 Gestion Mémoire

Question 1 Rappelez le principe général de la «pagination sur disque». À quoi cela sert-il ? Le matériel (en particulier le circuit MMU du processeur) doit-il offrir un support spécifique pour mieux décider de la répartition des pages en mémoire vive et sur disque ? Si oui, lequel précisément ?

Pour les questions suivantes, on se place maintenant dans le cadre du simulateur Nachos dans lequel on souhaite implanter un mécanisme de *swap* des pages sur disque. Pour simplifier, on considère qu'un bloc disque a la même taille qu'une page mémoire (i.e. `PageSize`). On dispose d'un objet global `swap` (instance de la classe `Swap`) qui permet de lire/écrire des blocs depuis/sur le disque (les blocs sont numérotés de 1 à `numBlocs`).

```
class Swap
{
  ...
  void ReadBloc(unsigned numBloc, void *destBuffer);
  void WriteBloc(unsigned numBloc, void *srcBuffer);
};
```

À titre d'illustration, voici comment copier le contenu du bloc de swap n°5 vers la page physique n°3 :

```
swap->ReadBloc(5, machine->mainMemory + 3 * PageSize);
```

Pour gérer l'espace de swap en permettant au noyau d'allouer et libérer des blocs, on déclare une nouvelle variable globale «`blocProvider`» dont l'initialisation est effectuée de la manière suivante :

```
blocProvider = new PageProvider(numBlocs);
```

Le comportement est donc similaire à pageProvider, à la différence qu'il gère le disque plutôt que la mémoire.

Question 2 On suppose que le champ `physicalPage` de la table des pages des processus est codé sur suffisamment de bits pour contenir un numéro de bloc disque. On peut donc utiliser ce champ pour mémoriser l'emplacement d'une page virtuelle sur le disque lorsque qu'elle a été évincée de la mémoire physique.

Proposez une convention simple permettant à Nachos de distinguer une page invalide d'une page «swappée» sur le disque (dans les deux cas, le bit `valid` est positionné à `FALSE`).

Question 3 On se place dans une version de Nachos qui fournit une primitive `FindVictim` prédéfinie (comprendre : que vous n'avez pas à écrire) capable de déterminer, parmi toutes les pages virtuelles actuellement présentes en mémoire, quelle elle la page virtuelle qui est la «moins récemment utilisée». Voici le profil de la primitive, qui renvoie un pointeur vers l'espace d'adressage choisi et sur le numéro de la page virtuelle choisie dans cet espace :

```
void FindVictim(AddrSpace **space, unsigned *numVirtPage);
```

Ecrivez une fonction `int SwapOut(AddrSpace *space, unsigned numVirtPage)` qui sera appelée lorsque le noyau décidera de déplacer sur le disque une page virtuelle résidant actuellement en mémoire. `SwapOut` doit simplement transférer le contenu de la page vers le disque et modifier la table des pages pour refléter cette nouvelle situation. L'entier renvoyé sert à indiquer si l'opération a réussi ou non. Notez qu'on ne se préoccupe pas des problèmes de synchronisation pour l'instant.

NB : la page physique dont on aura recopié le contenu sur le disque ne doit pas être restituée au système à l'issue de cet appel.

Question 4 Voici un rappel de la façon dont les pages sont allouées dans Nachos lors de la création d'un espace d'adressage (dans `AddrSpace::AddrSpace`) :

```
for (i = 0; i < numPages; i++)
  pageTable[i].physicalPage = pageProvider->GetEmptyPage();
  pageTable[i].valid = TRUE;
  ...
}
```

Modifiez ce code de façon à gérer le cas où plus aucune page physique n'est disponible (i.e. GetEmptyPage() renvoie -1). Il s'agit donc, en utilisant `FindVictim` et `SwapOut`, de «créer» de la place en forçant l'éviction d'une autre page vers le disque.

Question 5 On veut maintenant permettre aux processus de récupérer leur pages lorsqu'ils en ont besoin. Cela nécessite de traiter correctement les interruptions déclenchées lorsque la MMU rencontre une page invalide.

Lorsqu'une interruption de type « erreur de protection » se produit, l'exécution bascule dans le noyau Nachos dans la fonction `ExceptionHandler` :

```
void
ExceptionHandler (ExceptionType which)
{
  if (which == PageFaultException) {
    int address = machine->ReadRegister (BadVAddrReg);
    ... // à compléter
  }
```

Écrivez le code à l'intérieur du `if` pour traiter correctement le rapatriement d'une page depuis le disque lorsque c'est nécessaire, ou pour exécuter `interrupt->Halt()` lorsqu'il s'agit véritablement d'un accès mémoire illégal.

Question 6 Plusieurs problèmes de synchronisation restent en suspend. En particulier, si deux processus tentent d'allouer presque simultanément une page alors qu'il ne reste plus de mémoire physique, ils vont appeler `FindVictim` de manière concurrente et risquent donc de choisir la même page victime.

Indiquez comment corriger ce problème sans nécessairement utiliser des sémaphores (en tenant compte du fait que Nachos est monoprocesseur).

Question 7 D'autre situations plus complexes peuvent se produire. Par exemple, un processus peut demander l'accès à une page qui est en cours de transfert vers le disque (situation où la fonction `FindVictim` a fait un choix bien mal inspiré !)

Donnez une nouvelle version de `SwapOut` et de `ExceptionHandler` corrigeant ce problème. Indiquez bien les données annexes dont vous avez besoin. Vous pouvez utiliser au choix les sémaphores ou les moniteurs.

Question 8 Dans le cas où les processus contiennent des threads, un thread peut réclamer le rapatriement d'une page depuis le disque alors qu'elle est déjà en cours de rapatriement...

Indiquez comment corriger le problème.