

Architecture d'un Système d'Exploitation (UNIX) Tuyaux pour l'examen ORAL version 4.0 beta1

Retrouvez la dernière version de ce document sur le site du Cercle: <http://www.cerkinfo.be>

auteurs (1992/93) : Vivian, Camion, Récuratif, John'stick, Uli, Nicolas.
(1995/96) : Laurent K., Affuteur, Christophe M., JANNO, Touffu/gECKO.
(1998/99) : Gilles G.
(2003/04) : Kujovic

! attention !
Certaines réponses aux questions sont manquantes,
incomplètes, voir peut-être erronées ;
à vous de les compléter/corriger.

Quand on référencera une ligne d'un algorithme du cours par un numéro [n], on référence en fait le nombre qui est mis au début de la ligne ...

2 grandes questions : I : le file system - le system call Write()
II : la gestion de la mémoire paginée - le Page Stealer

I : Le File system - Le System Call Write()

Q1: Ayant l'algorithme du system call read (on possède à l'examen les photocopies de la plupart des algorithmes du cours : read, ialloc, ifree, alloc, free ...), donnez et expliquez ligne par ligne l'algorithme du system call write() supportant la gestion de TOUS les types de fichiers. Donnez également toutes les structures de données associées.

R: Les structures de données :
User File Descriptor Table (dans la U-area), File Table, inodes disk + mémoire, Buffer Cache.
Connaitre le schéma page 57 est une bonne idée.

Voici ma version de l'algorithme du system call write() basée sur la version du 07/09/02 de candiulb :

```
input: - user fd
        - adr buffer in user process
        - number of bytes to write
output: - number of bytes written

get FT entry from UFDT;
check file accessibility;
get inode from FT entry;
set parameter in U_Area for user adr, count, I/O to user;
lock inode;
if (file is special)
{
    extract major and minor device number from inode;
    call driver;
}
else if (file is regular)
{
    copy file offset from FT to U_Area;
    while ( count > 0)
    {
        convert file offset to disk bloc ( bmap modified );
        /* Version modifiée de bmap, il faut gérer le cas où le numéro de
        bloc dans inode = 0. (Le cas d'un bloc LOGIQUE pas encore
        attribué PHYSIQUEMENT.
        1. if (n° bloc == 0) alloc, initialise avec des 0,
           mark delayed write (+blocs d'indirections)
        2. brelse du bloc alloué avant de sortir.
        3. if (FS full) return (error)
        */
        if ( byte offset into bloc != 0 or count < bloc size )
            bread (bloc, offset);
    }
}
```

```

        // byte offset into bloc != 0 == ca veut dire qu'il y a déjà
        // quelque chose dans le bloc et que tu écris en append. (bloc partiel)
    else
        get block; // bloc complet
    copy data from user adr to system buffer;
    mark buffer delayed write;
    update U_Area fields for byte offset, count, adr to read in user space;
    brelse;
}
update FT offset;
if ( FT offset > logical file size )
    update logical file size in inode ;
}
else if ( file is pipe )
{
    copy write offset from inode (descriptor l2) to U_Area;
    while ( count > 0 )
    {
        evaluation space in pipe;
        // si écriture atomique ou si on doit écrire autant qu'on peut :
        if ((enough space) || ((count > 10B) && (space available >0)))
            //on vérifie s'il y a de la place dans le pipe afin d'éviter de faire
            //une boucle infinie
        {
            // pour tout les blocs que l'on va devoir écrire
            for (all allowable and necessary blocks )
            {
                if ( partial block ) bread;
                else getblk;
                copy data from user adress to system buffer;
                mark delayed write;
                update U-Area (adr to read, byte offset, count);
                brelse;
            }
            update write offset and file size in inode;
            check inode flag for asleep readers;
            if ( asleep readers )
            {
                clear flag in inode;
                wake-up (event: pipe contain data);
                // wake prend pas effet directement ...
                // il appelle soit le swapper soit il
                // met le reschedule flag
            }
        }
        else // pas assez de place
        {
            mark inode flag (event: writers waiting for pipe);
            unlock inode;
            sleep (event: space in pipe); // sleep sur inode du pipe
            lock inode;
        }
    }
}
update instant in inode + mark inode modified
unlock inode;
return ( nb bytes written );

```

Rem: Le write sur directory n'ayant, pour moi, aucun sens, je ne l'ai pas traité dans l'algorithme. Si vous voulez le rajouter, il faut simplement traiter les répertoires comme les fichiers réguliers. Je n'ai trouvé aucun cas où le write sur un directory était utilisé; si vous en connaissez, merci de me le faire savoir.

Q2: Au début de l'algo, quel algo utilise-t-on pour faire le *get i-node from file table entry* ?

R: On n'utilise aucun algo (certainement pas *iget* en tous cas !!!). En effet, comme on ne peut écrire que sur un fichier déjà ouvert, on est certain que l'*inode* est présente en mémoire (car son RC != 0). Il s'agit donc simplement de lire le pointeur vers l'*inode* dans la File Table.

Q3: Dans l'algo du Write(), quand peut-on être mis en sleep pour la première fois ?

R: Lors du verrouillage de l'*inode*: *lock inode*.

Q4: Si on est mis en *sleep* lors du *write*, comment est-on sûr que l'on aura pas de problèmes si un autre processus fait aussi un *write* pendant notre *sleep* ?

R: On a verrouillé l'*inode* au début du code, ce qui endormira tous les autres processus exécutant cette partie de code sur la même *inode* que nous. Il n'y aura donc pas de conflit. C'est valable aussi pour les autres appels systèmes qui « tripotent » à l'*inode*. Ceci est symétriquement aussi valable pour nous (*cf.* question précédente).

Q5: Pourquoi dans l'algo du Write() utilise-t-on une version de l'algo Bmap améliorée (par rapport à celle du cours) ?

R: Il faut prendre en compte l'éventuelle allocation physique de blocs (data + indirections) réservés logiquement...

Q6: Que se passe-t-il lorsqu'on veut écrire plus de 10 blocs dans un pipe ?

R: Tant que le nombre de données en nombres de bytes est plus grand que 10 blocs on écrit tout ce qu'on sait sur le pipe (s'il y a de la place) ...

On doit s'assurer qu'on arrivera à écrire quelque chose sinon on risquerait de faire une boucle infinie.

Q7: Qui peut écrire dans une directory ? Dans quel cas cela se présente-t-il ?

R: Le process qui a un Effective User ID (EUID) = 0 = Super User. -> System calls : create (open create), mknod, link et unlink ...

Q8: Comment peut-on devenir Super User ?

R: Il faut changer l'EUID avec setuid(), mais il faut que le bit "set UID" (qui se trouve dans l'*inode*) soit positionné.

Q9: Expliquer le system call setuid().

Q10: Pourquoi, dans l'algorithme du Write(), sauvons nous les paramètres de cet I/O dans la U-Area, et non pas dans un stack kernel ?

R : Car, il ne peut pas s'agir d'un stack : un stack nous permet d'empiler et de dépiler des données, avec une gestion FIFO, or le scheduler ne va pas spécialement réveiller les différents processus dans l'ordre inverse de leur sommeil => problème => utilisation de la U-AREA.

Q11: Que se passe-t-il quand le dernier lecteur quitte un pipe ?

R: Quand le dernier lecteur fait un *close* sur le *pipe*, il envoie un signal, qui, le cas échéant, force le système *call write* à se terminer

Q12: Que se passe-t-il lorsqu'on ne met pas à jour la taille du pipe?

R: Les éventuels lecteurs réveillés vont se rendormir...

Q13: La lecture (écriture) sur un *pipe* est-elle déterministe ?

R: Non ! Par exemple, si on a plusieurs lecteurs (écrivains), l'ordre dans lequel ceux-ci seront réveillés n'est pas garanti (il est même totalement aléatoire).

Q14: Que se passe-t-il, du point de vue de l'*inode* et des blocs avec un *pipe* anonyme quand il n'y a plus ni écrivains ni lecteurs; et avec un *pipe* nommé ?

R: Quand on ferme le dernier accès à un *pipe* anonyme, le RC de l'*In-core inode* vaut 0, et elle peut donc disparaître de la mémoire. *Close* vérifie aussi la valeur du *link count* de l'*inode*. Comme il s'agit de *pipe* anonyme, le *link count*=0 (pas de nom), et donc l'*inode* disque, ainsi que tous les blocs constituant le fichiers sont libérés.

Dans le cas d'un *pipe* nommé, le *link count* n'est évidemment pas nul, mais on libère malgré tout la place sur le disque quand on constate qu'il s'agit d'un *pipe*.

Q15: Il est question d'écrire dans un *pipe*... Mais où écrit-on en fait ????

R: Les pipes sont des fichiers, et donc leur contenu se retrouve sur un *file system*. Dans le cas du *pipe* nommé, cela ne pose pas de problème, puisqu'il a un nom et on peut donc le localiser quelque part dans la hiérarchie du système de fichiers. Dans le cas du *pipe* anonyme c'est impossible. Il existe donc une variable qui dit où, dans le *file system*, les *pipes* anonymes doivent être créés. Cet endroit est choisi par le *Super User*.

Q16: Qu'est ce qui différencie l'écriture dans un pipe, de celle dans un fichier normal (regular) ?

R: Lorsqu'on écrit dans un pipe, on est certain d'augmenter la taille du fichier.

Les offsets de lecture et d'écriture sont gérés modulo 10. **Le write sur pipe peut être bloquant.**

Q17: Du point de vue de l'utilisateur, quelle est la différence la plus marquante entre un *read* et un *write* dans un *pipe* (à part que l'un lit et l'autre écrit ;-)

R: Le read sur pipe lira ce qui est disponible et ne se mettra en sleep que s'il n'y a rien à lire; on peut ainsi lire moins que ce qu'on avait demandé. Le write au contraire se mettra en sleep s'il n'a pas assez de place pour écrire la totalité de ce qu'il à écrire.

Q18: Dans l'algorithme ialloc, pourquoi doit-on re-vérifier que l'inode est réellement libre ?

R: Un processus A peut vouloir une inode libre. Il la prend alors dans la free list du Super Bloc.

Lors du get inode (Iget), il peut être mis en sleep. Cependant l'inode n'a pas encore été marquée sur le disque comme étant utilisée. Si un processus B cherche des inode libres sur le disque (pour remplir la free list du Super Bloc), il se peut qu'il prenne l'inode prise par le processus A ... Le processus A doit donc repérer une telle situation.

Q19: Dans l'algorithme ialloc, comment serait-il possible d'allouer 2 mêmes inodes ? (trouver un exemple)

R: voir la réponse de la question précédente

Q20: Dans l'algorithme ialloc, que pourrait-il se passer si on ne verrouille pas le Super Bloc ?

Q21: Où se trouvent l'User ID (UID) et l'Effective User ID (EUID) ?

R: Dans la U-Area et dans la Process Table.

Q22: Pourquoi un Effective User ID (EUID) dans la process table et dans la U-Area ?

R: Parce que, lorsqu'on fait un setuid(), on modifie l'EUID dans la U-Area, il faut donc pouvoir le retrouver (dans la Process Table) pour pouvoir le remettre à son état initial.

Q23: Pourquoi, dans un pipe, on met les offsets write et read dans l'inode et pas dans la file table ?

R: Il n'y aurait pas de problèmes pour les pipes anonymes mais pour les pipes nommés, il y aura des entrées différentes dans la File Table. Pour contrer ce défaut les offsets sont mis dans l'inode.

Q24: Quel algo utilise-t-on pour écrire dans un *directory* ?

R: Il faut utiliser le *system call* « *make new node* ». Nous avons ici essayé de faire un algorithme de *write* universel, dans ce sens qu'il fait appel à *make new node* s'il s'agit d'un *directory*. On pourrait choisir de retourner une erreur pour forcer l'utilisateur à faire un appel explicite à *make new node*. Cet algorithme spécial est nécessaire pour des raisons de cohérence de l'info située dans les *directories*.

Q25: Quand fait-on un *read* dans un *directory* ?

R: Par exemple quand on veut lister tout le contenu du *directory* (ls, etc...)

Q26: Quand peut-on avoir plusieurs pointeurs vers la même entrée de la FT ?

R: Si les deux pointeurs viennent de la même UFDT: On a fait un *dup*
« « « « « « deux UFDT différentes: On a fait un *fork*.

Q27: Quand peut-on avoir plusieurs pointeurs vers la même *In-core inode* ?

R: Cela signifie que l'on a ouvert plusieurs fois le même fichier

- Avec des permissions différentes
- Avec des *read* et *write offsets* différents
- Avec des noms différents...

Q28: Expliquer sommairement le scheduling des processus sur le systeme UNIX System V. Que veut-on dire par priorité dynamique d'un processus ?

R: Le *scheduler* élit le processus qui va recevoir le contrôle du processeur. Il est réveillé:

- A intervalle réguliers (*cf. Clock*)
- Lorsqu'un processus se met en *sleep*
- Lors des *wake-up*

Il choisit le processus le plus prioritaire pour lui donner le contrôle. Pour ce faire, il calcule la priorité à partir de 2 paramètres:

- la priorité statique; elle dépend de la nature des processus (processus système ou utilisateur, etc.)
- la priorité dynamique; qui augmente proportionnellement au temps que le processus a passé sans avoir le contrôle (*swapped out* ou *ready to run in memory*)

Q29: Lorsque l'on fait la *wake-up*, perd-t-on le contrôle immédiatement ?

R: Non, cela signifie simplement que les processus en attente sont marqués *ready to run*. Quand a savoir qui aura *in fine* le contrôle, c'est une décision du *scheduler*. De toute façon, nous ne pouvons pas perdre le contrôle tant que le *system call* n'est pas terminé (ou tant que nous ne nous mettons pas en *sleep, of course...*).

Q30: Comment s'alloue-t-on un nouveau bloc ?

R: voir cours pages 68 à 71

Q31: Comment s'alloue-t-on une nouvelle inode ?

R: voir cours page 50

Q32: Comment gère-t-on la liste d'i-nodes et la liste des blocs du Super Bloc ?

R: voir cours pages 50 et 54.

Q33: Pourquoi sauver les données sur la U-Area ?

R: Pour permettre la réentrance. Si un autre process fait un read ou un write en meme temps, il ne faut pas qu'il détruise les variables de travail de l'autre.

Q34: Quand on crée un pipe anonyme, combien d'entrées dans la FT crée-t-on ? Pourquoi ?

R: Deux pour avoir des permissions différentes (pipe renvoie 2 file descriptors). Cela permet de savoir quel compteur (#E ou #L) doit être décrémenté lors des close.

Q35: Est-on sur que l'inode est en mémoire?

R: Oui elle a été installée en mémoire lors de l'open (par namei).

Q36: Quand chmod s'active t'il réellement?

R: Lorsque le refcount du fichier = 0. Sinon il reste des processus qui ont ouvert ce fichier avec les anciennes permissions (celle-ci ne sont vérifiées que lors du open, après on utilise les permissions d'ouverture se trouvant dans la File Table).

Les nouveaux open utiliseront néanmoins les nouvelles permissions.

Q37: Quel est la différence entre l'ouverture d'un pipe nommé et d'un pipe anonyme?

R: Lors de l'ouverture d'un nommé, on risque d'être mis en sleep (s'il n'y a pas de lecteur/écrivain), chose qui ne risque pas d'arriver avec un pipe anonyme vu qu'on s'ouvre automatiquement un accès en lecture et un en écriture.

II : La Gestion De La Mémoire Paginée - Le Page Stealer

Q1: Expliquer la gestion de la mémoire avec pagination à la demande, sur UNIX SYSTEM V.

Donner les structures de données associées et expliquer les algorithmes associés.

Pour ce faire, on dispose de photocopies de presque tous les algorithmes du Bach vus au cours.

Mais y'a pas l'algo du Page Stealer !

R: Retaper (!de mémoire!) l'algorithme du Page Stealer.

Les structures de données :

Process Table, U-Area, Per Process Region Table (PPRT), Region Table (RT), Page Table (PgT), Disk Bloc Table (DBT), Page Frame Data Table (Pfddata), Swap Use Table.

Faire un beau schéma des différentes tables et les différentes relations entre elles.

Q2: Expliquer l'algorithme du Page Stealer ligne après ligne.

Q3: Expliquer partout où interviennent les reference counts.

R: Region table, Swap Use Table , Page Frame Data Table.

Q4: Expliquer la différence entre avoir reference count > 1 dans la **Region Table** et avoir un reference count > 1 dans la Page Frame Data Table.

R: Le partage de mémoire se fait à 2 niveaux distincts: niveau des régions et niveau des pages.

Lors d'un fork, on doit "dupliquer" chaque région du processus père pour en faire une copie pour le fils.

On va partager le plus possible afin d'éviter des copies inutiles.

- **Les régions partageables (text et shared) seront partagées au niveau des régions. On incrémente le refcount de la REGION TABLE et on fait pointer la PPRT du processus fils dessus. On ne touche pas aux refcounts des pages frames.**
- **Les régions non partageables (data, stack) doivent être dupliquées. On se crée donc une nouvelle région (avec un refcount de 1) qui aura une nouvelle Page Table et Disk Block Table MAIS qui seront des copies des Page Table et Disk Block Table de la région du père. On met le copy-on-write bit à 1 dans les Pages Tables du père et du fils, on incrémente le pfddata refcount pour chaque page valide et on incrémente le Swap Use refcount pour chaque entrée du type "swap" dans la Disk Block Table. Au final, on obtient 2 régions DISTINCTES mais qui partagent (pour le moment) les mêmes pages frames et swap; on n'a pas touché au refcount de la région du père.**

Q5: Dans le Page Stealer, que doit on faire quand on swappe out les pages ?

R :

Si type dans Disk Block Table != "swap"

{

cherche refcount == 0 dans Swap Use Table

on le met à 1

Disk Block Table: met le no bloc + device, met status à "swap"

Pfdata: met le no bloc + device, status "copie sur swap", changer de hash list

}

Effectue le swap à l'endroit indiqué dans Disk Block Table.

Met le modify bit à 0 (la page pourra assez être retirée lors du prochain passage du page stealer).

Q6: Qui est susceptible de réveiller le Page Stealer ?

R: Les algorithmes protect-fault (p-fault) et validity-fault (v-fault).

Remarque : dans l'algo exec, on peut s'allouer des pages de 2 manières (cfr cours), et finalement, on remarque qu'il fait appel au v-fault. (Donc il n'y a que v-fault et p-fault qui peuvent effectivement réveiller le Page Stealer).

Q7: Dans quels cas appellera-t-on v-fault ?

R: - Lors d'une faute de page (= page fault - l'ère candi).

- Lorsqu'on adresse une zone hors de l'adressage virtuel du processus.

Q8: Dans quels cas appellera-t-on p-fault ?

R: - Lorsqu'on écrit dans une page dont le bit copy on write est à 1

- Lorsqu'on ne respecte pas le type de permissions (rwx) accordées à une région. (une page d'une région)

ex: écrire dans une région text (code), exécuter une instruction dans une région data, ...

(pour autant que ces régions (pages dans les régions) ont respectivement les permissions rx, rw)

Q9: v-fault (Validity Fault) : décrire l'algorithme ligne après ligne.

Q10: p-fault (Protection Fault) : décrire l'algorithme ligne après ligne.

Q11: Dans les algos p-fault et v-fault à quel moment (préciser à quel endroit dans les algorithmes) on appellera (réveillera) le Page Stealer ?

R: Au moment où l'on doit s'allouer physiquement une page.

(cfr algo v-fault : ligne [10] ; algo p-fault : ligne [5]) ...

Q12: Dans v-fault, aux lignes [6] et [7], "if (page in cache)" et "remove page from cache", que signifie "cache" dans les deux cas ?

R : Il signifie "Hash list" à la ligne [6] et "Free list" à la ligne [7].

Q13: Dans v-fault, à la ligne [7], on a "remove page from cache"; la page frame se trouve-t-elle toujours dans la free list ?

R : Non, uniquement si son reference count == 0. cette instruction n'est donc pas toujours nécessaire.

Q14: Pourquoi dans l'algo p-fault, faut-il se dissocier de la copie sur le swap device (s'il en existe une) ?

R: Un exemple vaut toutes les explications du monde; soit la situation suivante :

Un processus A accède à une page dont le validity bit == 1 dans la Page Table. Cette page a également une copie sur le swap device (avec un reference count == 1) et elle fait partie d'une région privée et elle se trouve en mémoire dans une page frame dont le descripteur dans la Page Frame Data Table a un reference count == 1.

Si ce processus lance un fork(), le nouveau processus créé (soit le processus B) aura une nouvelle entrée dans la Region Table et une nouvelle Page table qui sera une copie de la Page Table (et de la DBT) du processus A pour la région privée décrite plus haut. (le bit copy on write mis à 1 chez chacun). Le reference count de la Page Frame Data Table et de la Swap Use seront incrémentés et vaudront 2.

Si le Page Stealer reprend la page au processus A (et pas à l'autre), le reference count de la Page Frame Data Table sera décrémenté et vaudra 1. Le Processus B sera alors le seul à accéder à la page frame mais les deux processus partagent toujours la copie sur le swap.

Si le processus B écrit dans la page, il peut garder la page frame vers laquelle il pointe puisque il est le seul à y accéder, cependant IL DOIT SE DISSOCIER de la copie sur disque (décrémenter le reference count associé de la Swap Use Table) car si la page est retirée par le Page Stealer au processus B, elle sera recopiée sur l'ancienne copie du Swap Device encore référencée (dans la Disk Bloc Table) par le processus A.

Q15: Dans le p-fault, comment pourrait-on améliorer la gestion du cas où reference count == 1 ?

R : Au lieu d'automatiquement détruire l'association avec la copie swap, vérifier d'abord si le reference count dans la swap use table n'est pas égal à 1 ... s'il l'est, on peut garder la copie swap, ce qui est avantageux...

Q16: Dans la table d'utilisation du swap device, la Swap Use Table, quand est-ce qu'on a un reference count > 1 ?

R: Lors d'un fork(), dans le cas des régions privées. Pour chaque page des régions privées, on incrémente le reference count de la Swap Use Table si la page a une copie sur le swap device (c'est indiqué dans la Disk Bloc Table associée) et le reference count de la Page Frame Data Table est aussi incrémenté si la page a le validity bit == 1 dans le Page Table associée. (ne pas oublier que le bit copy on write est aussi positionné à 1 chez le père et le fils)

Q17: Dans l'algo p-fault pourquoi au début faut-il tester que la page est bien présente (et valide) ? Et si elle est absente, dans quel cas cela peut-il se présenter ? On a admis que si v-fault doit être appelé il le sera avant p-fault, donc pourquoi ce test ?

R: p-fault a pu être mis en sleep lorsqu'il essayait de locker la région (cfr ligne []) et le Page Stealer a pu voler la page pendant ce temps.

Q18: Expliquez comment fonctionne exec ? Quels sont les deux cas possibles ? (système avec ou sans pagination à la demande).

R: Cfr algorithme du cours. Faire attention à ce qui est viré (pas les pages mais les régions ; les pages ne sont virées que si elles ne sont plus référencées) ...

Q19: Dans un système UNIX, quelle type de région(s) trouve-t-on ?

R: Les régions de code, les régions de datas, les shared (memory), les régions de stack et les régions libres.

Q20: Au niveau de UNIX System V, quel type de région(s) ne trouve-t-on pas ?

R: La région de code, car ce type de régions est automatiquement de type shared (memory).

Q21: Où peut-on trouver des pages dont le validity bit est à 0 ?

R: Soit dans la free-list des pages frames, soit sur le swap device, soit nulle part (soit dans le fichier exec, soit 0 à la demande, soit demand-fill) ...

Q22: Comment peut-on retrouver une région (dans l'algo p-fault) ?

R: Dans p-fault, on passe l'adresse comme paramètre et on peut retrouver la région dans la PPRT, car on connaît l'adresse virtuelle du début de la région et la longueur de celle-ci ...

Q23: Bien expliquer ce qu'est le context-layer ? Qu'est-ce qu'on place sur le stack ? Quelle est la grandeur de la table qui gère ce context-layer ?

R: Pour les explications voir le cours ... La taille d'une table qui gère le context-layer doit être proportionnelle aux nombres de niveaux (de priorité) des interruptions dans le système.

Q24: Expliquer ce qu'est un fichier "prédigéré". Donner l'intérêt d'un tel type de fichier.

Q25: Qu'est-ce que le "magic number" associé à un fichier exécutable ?

Q26: Expliquer ce qu'est un processus "daemon". Pourquoi le réveille-t-on ? Qui et où ?

Q27: Comment un programme utilisateur peut-il descendre et puis relever sa priorité ?

R: Un processus exécute un programme utilisateur, à un certain moment il lance un fork().

Le père se met alors en sleep et attend le fils avec wait().

Le fils lance setnice(), ce qui va augmenter son nice, c'est-à-dire abaisser sa priorité.

Le fils exécute alors la même portion de code que le père puis après un moment se termine.

Le père reprend alors son exécution comme initialement ...

Remarque : lorsque dans l'algo p-fault, on manipule des informations dans la page table et dans la table d'utilisation du swap, penser à tout ce qu'il faudrait faire dans la table des descripteurs de blocs, et ce dans tous les cas.

Q28: Quels algos modifient la Swap Use Table?

R:

● **Pfault: dissocie de la copie swap**

● **Exit: libère les régions du processus (sauf si elles sont le sticky bit)**

- Exec: libère les anciennes régions du processus (cela se fait, comme pour exit, via freereg)
 - Fork: incrémente le refcount lors de la duplication de la Disk block table
- Bien savoir expliquer le fonctionnement de ces algos!

Q29: Quelles sont les différences p/r au système de pagination vu en 1ère candi?

R:

- Possibilité de partager des pages frames
- Le nombre de pages frames allouées à un process varie au cours du temps
- Possibilité de charger directement du fichier binaire préédigéré (+ demand 0 et demand fill).
- La page n'est pas d'office swapée mais retirée logiquement

Q30: En quoi l'algo du Pfault est-il trompeur? (comprenez "faux")

R: La version du Bach (et par la même occasion du syllabus) ne casse l'association avec le swap que si le refcount de la page frame est égal à 1. C'est complètement idiot car d'autres processus swapés (et donc qui n'utilisent plus la page frame pour le moment) pourraient toujours se référer à la copie sur swap. Si on ne casse pas l'association, le processus qui a provoqué le Pfault pourrait modifier la copie swap; les processus swapés se retrouveraient alors avec une version modifiée.

Il faut donc casser l'association dans tous les cas.

Rem: l'optimisation présentée ci-dessus est toujours vraie, on peut optimiser en ne cassant pas l'association si le ref count de la Swap Use Table est égal à 1 car on est alors certain que plus personnes d'autres n'utilisent la copie swap.

Complément d'infos (ce que le syllabus ne dit pas)

- Nous voyons page 133 qu'il y a essentiellement 3 statuts pour la pfddata: libre, loading, possède une copie sur disque.
Il manque un 4e statut possible très important: page frame en mémoire mais nul par ailleurs (cad ne possède pas de copie sur disque).
Imaginons le cas d'une page non valide et dont le type dans sa Disk Block Table est "demand 0". Le processus veut y accéder, déclenche vfault qui alloue une page frame à la page. La page frame est bien en mémoire mais nous n'avons pas de copie sur swap.
- Toujours page 133, il y est indiqué que les différents types pour la Disk Block Table sont: swap, fichier binaire, demand 0, demand fill.
Il y a un 5e statut qui n'est pas indiqué. Reprenons l'exemple précédent, après l'allocation d'une page frame à la page qui était demand 0 (ou fill), il faut changer le statut car elle n'est plus demand 0 vu qu'on a satisfait la demande, elle n'est pas non plus swap ni binaire; il y a donc un 5e statut (appelez le comme vous voulez).

Q31: Pourquoi une région détachée ayant un ref count à 0 peut-elle rester en mémoire; qui décide de cela et comment?

R: Si elle a le sticky bit, detachreg ne supprimera pas la région. Seul le SU peut donner un sticky-bit. Une région ayant le sticky bit pourra quand même être libérée: voir page 105.

Q32: Le page stealer est un démon kernel, pourquoi ne peut-il être un process user?

R: Parce qu'il utilise et manipule des structures du kernel: table des régions, tables des pages, etc

Q33: Qui fait les swap in?

R: - les swaps in (logiques!!!!) de processus sont fait par le swapper
- les swap in de pages sont fait par Vfault

Q34: LoadReg modifie-t-il la Swap Use Table?

R: Non. On modifie la Disk Block Table mais on ne doit pas s'allouer de l'espace sur le swap device; donc pas de modification de la Swap Use Table.