VMS POSIX

Overview

Bernard OURGHANLIAN

Symposium DECUS

4 Juin 1991

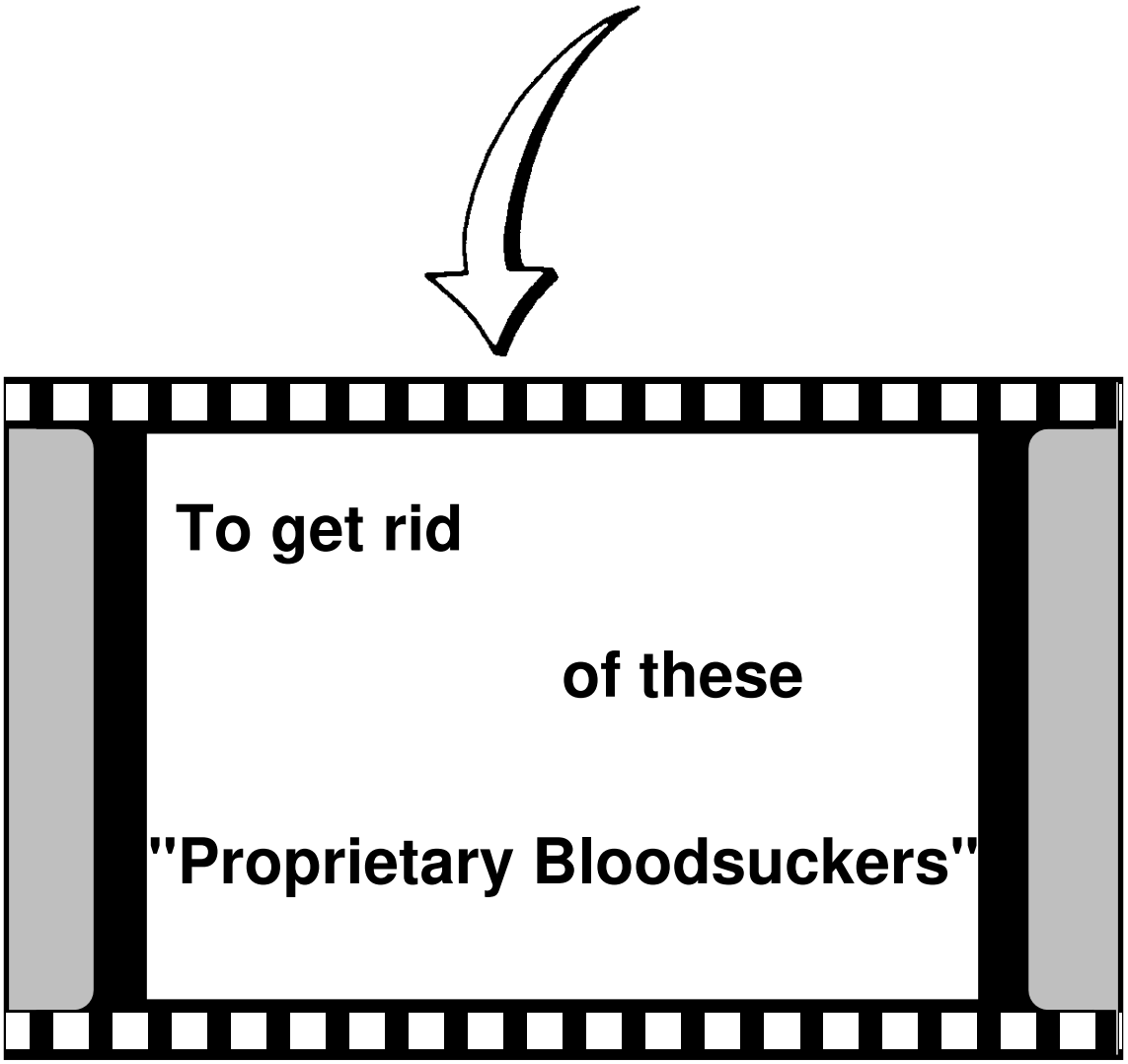
VMS 06

Sommaire

1. Introduction
2. Qu'est-ce qu'un système ouvert ?
3. Qu'est-ce que POSIX ?
4. VIP : VMS Integrated POSIX
5. Les différences de *philosophies* entre POSIX et VMS
6. La vue de l'utilisateur
7. *Overview* de VIP
    a. 1003.1
    b. 1003.2
    c. 1003.4

Introduction

Why Open Systems ?



Introduction

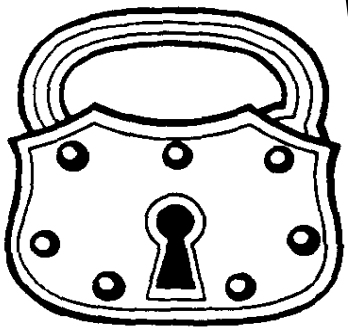
DEC

IBM

HP

BULL

VMS

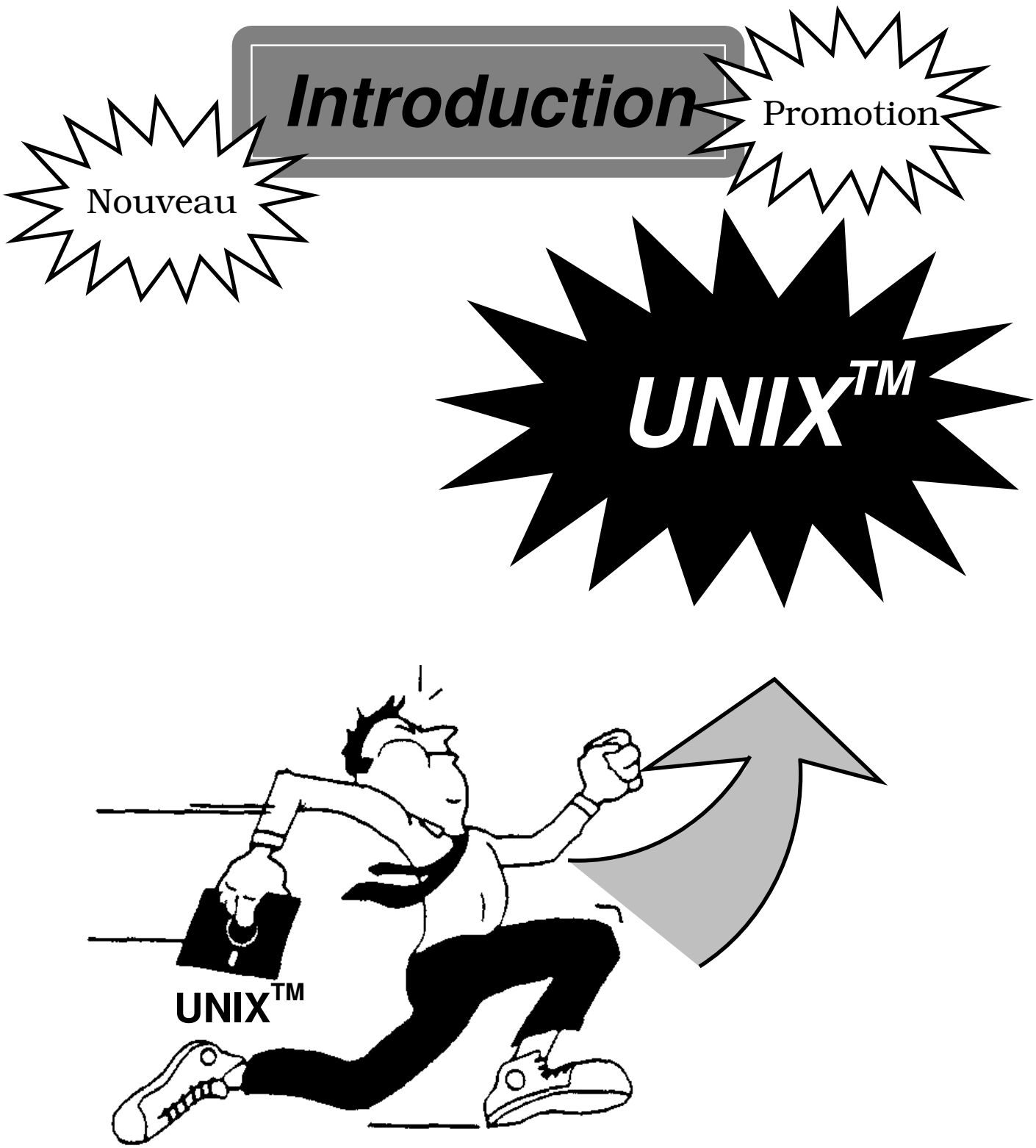


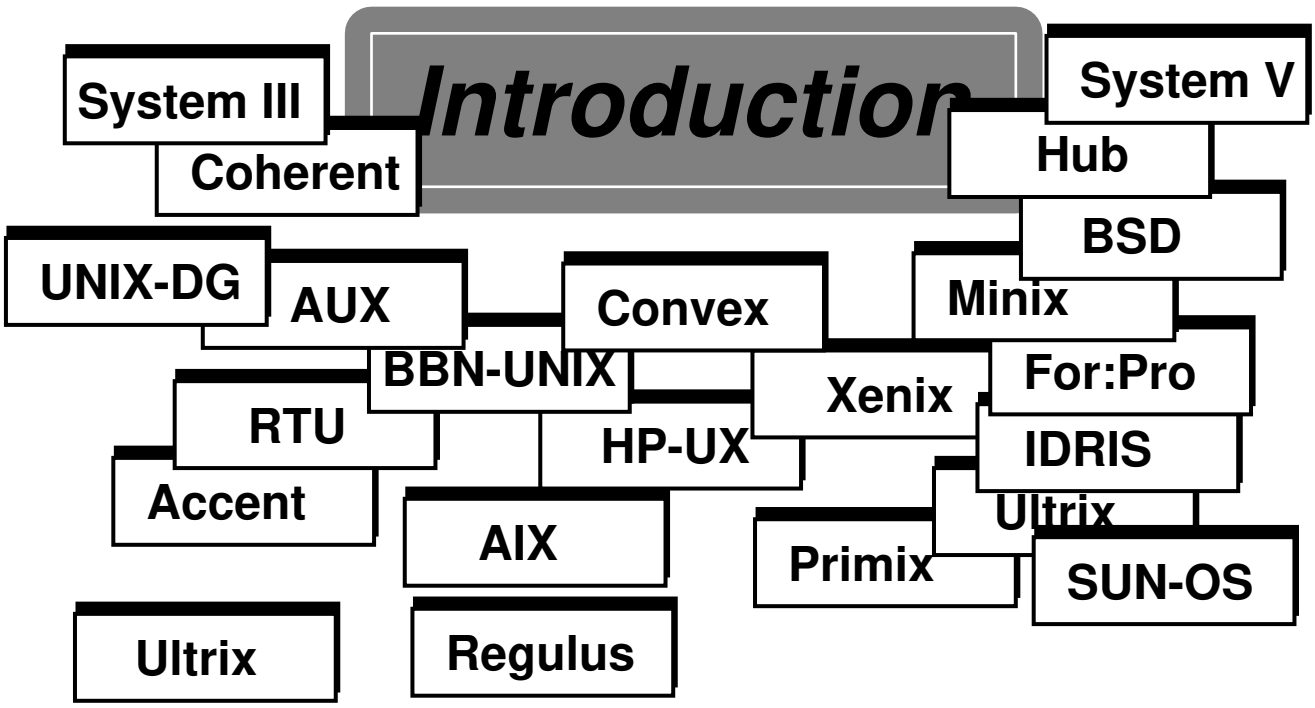
MVS

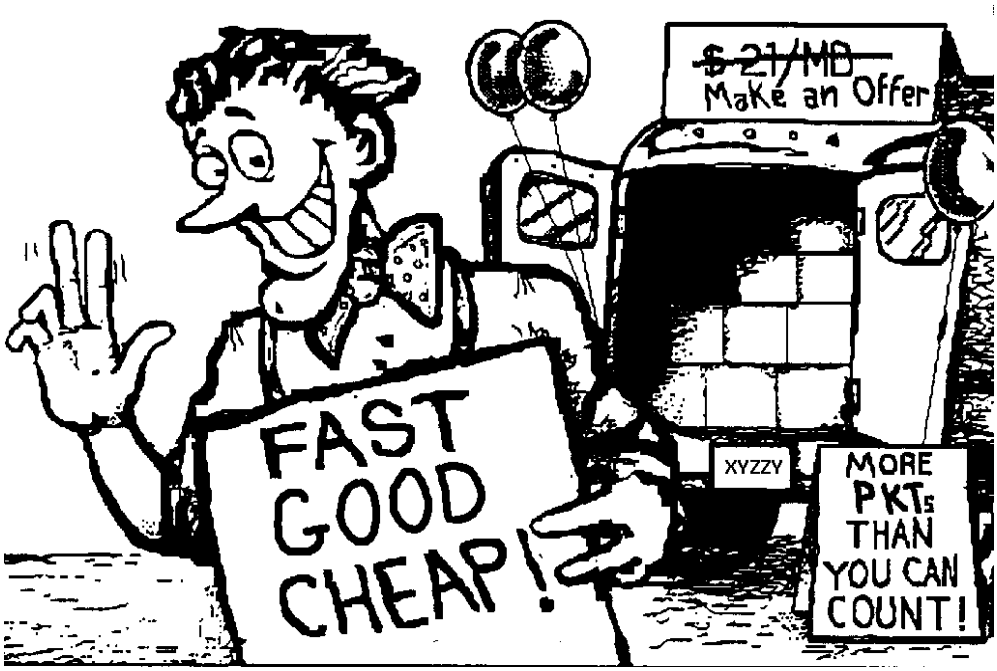
MPE

GCOS

Utilisateur


Introduction
Nouveau
Promotion
UNIX™
UNIX™



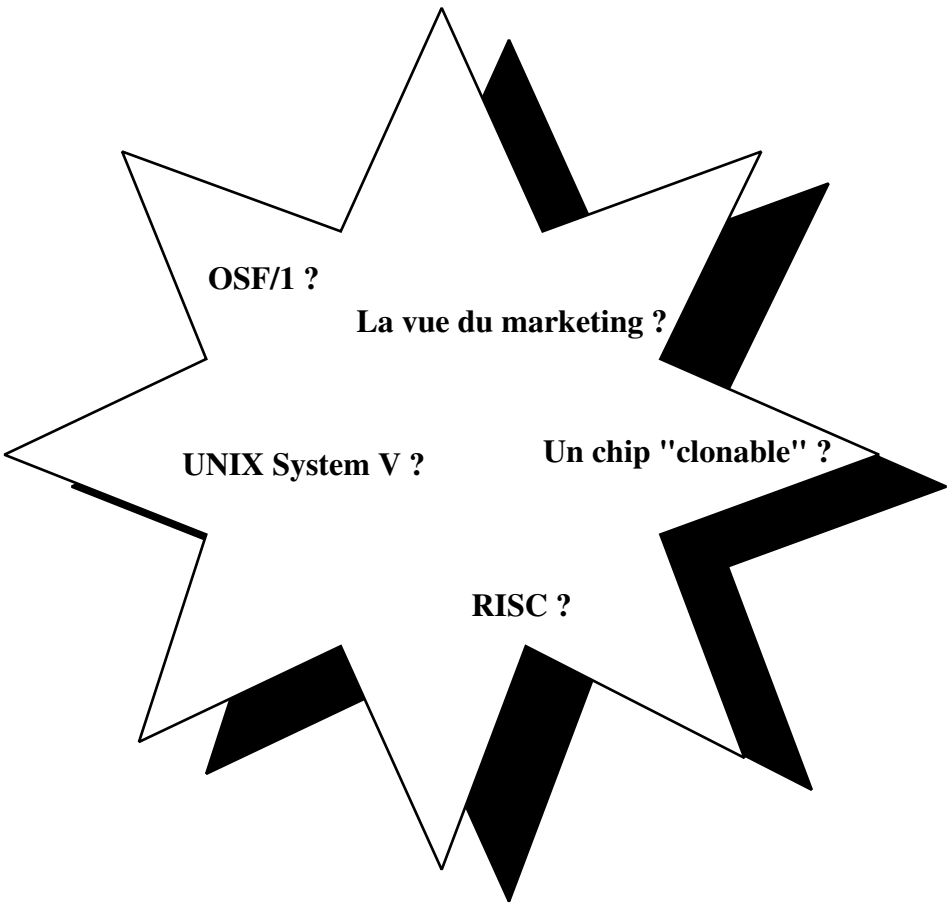


Comment choisir ?

Introduction

Question :

Mais d'abord qu'est-ce qu'un "système ouvert"?



Introduction

Réponse :

Pourquoi se battre sur une définition ?
Il en existe une :

- **Stable**
- **Indépendante des constructeurs**
- **Adoptée par l'ISO**
- **Adaptable**
- **Protégeant l'investissement**

POSIX

Introduction

Pourquoi POSIX ?

Parce que :

- **Indépendant vendeurs/architectures**
- **Adopté par la communauté internationale :**

Open Software Foundation	**X/Open**
UNIX International	**Uniforum**
European Commission	**NIST**
ISO	

Introduction

Pourquoi POSIX ?

Parce que :

"...the various [UNIX] systems have diverged, and the need for standards has heightened."

"Your guide to POSIX", copyright 1989 by Uniforum

Qu’est-ce qu’un système ouvert ?

- D’apres la définition de l’IEEE (POSIX 1003.0) :

"A system that implements sufficient **open specifications** for interfaces, services, and supporting formats to enable properly engineered applications software :

- to be **ported** across a wide range of systems (with minimal changes)
- to **interoperate** with other applications on local and remote systems
- to **interact** with users in a style which facilitates user portability."

- Toujours d'après la définition de l'IEEE (POSIX 1003.0) :

open specifications = "...unencumbered public specifications that are maintained by an open consensus process."

- Mark Schulman lors de l'X/Open Market Requirement Forum (8-Mai-1990) :

"Unix and Open Systems are not synonymous...

A system can be made open only by the presence of... [interfaces] between operating system and applications.

What users need are standards that define [that] interface... [that is] the foundation of open systems.

Qu'est-ce que POSIX ?

- Une famille de standards de l'IEEE (*Institute of Electrical and Electronics Engineers*) destinée à définir une interface et un environnement d'accès standard au système d'exploitation permettant le support d'une portabilité des applications au niveau du **code source**
- POSIX : ***P***ortable ***O***perating ***S****ystem* ***I****nterface* ***X*** *for computing environments*
- 1003 est le numéro assigné à POSIX par l'IEEE
- Plus de 20 groupes de travail (1003.0.... 1003.15...)

Qu’est-ce que POSIX ?

Une brève histoire de POSIX :

- 1981 : création de /usr/group
- 1984 : approbation du standard produit par /usr/group
- 1985 : fusion avec le groupe de travail de l’IEEE POSIX 1003.1
- 1986 : formation des goupes 1003.2 et 1003.3
- 1987 : formation du groupe 1003.4
- 1988 : approbation et publication de 1003.1
- 1989 : /usr/group change son nom en UniForum

Qu’est-ce que POSIX ?

- 1003.0 : *Guide to POSIX environment*
- 1003.1 : *System Interfaces*
- 1003.2 : *Shell and Utilities*
- 1003.2a : *User Portability Extension*
- 1003.3 : *Test Methods*
- 1003.4 : *Realtime Extensions*
- 1003.4a : *Threads*
- 1003.5 : *Ada Bindings*
- 1003.6 : *Security*
- 1003.7 : *System Administration*

- 1003.8 : *Networking* (différents sous-groupes)
- 1003.9 : *Fortran Bindings*
- 1003.10 : *Supercomputing AEP (Application Environment Profiles)*
- 1003.11 : *OLTP AEP (Application Environment Profiles)*
- 1003.12 : *Process-to-process*
- 1003.13 : *Realtime AEP (Application Environment Profiles)*
- 1003.14 : *Multi-processing*
- 1003.15 : *Batch*
- 1003.xx : *Name Space/Directory Services*
- .....

Qu'est-ce que POSIX ?

- **IEEE POSIX 1003.0 :**

  -- Guide décrivant l'environnement POSIX

  -- *Draft* 10 en cours d'analyse

- **IEEE POSIX 1003.1**

  -- Interface et environnement d'accès standard au système d'exploitation permettant le support d'une portabilité des applications au niveau du code source

  -- Standards IEEE 1003.1-1988 et 1003.1-1990 déjà publiés. Addendum A sous forme de draft

  -- Repris par l'ISO (ISO/IEC 9945.1:1990)

-- Comprend des primitives pour la gestion de processes, accès au *file system,* entrées/sorties et gestion de périphériques

-- Ces primitives ne sont accessibles que depuis le **langage C**

- **IEEE POSIX 1003.2 et 1003.2a**

  -- Interface et environnement standard permettant le support d'applications de type *shell scripts* (langage de commande)

  -- Pour 1003.2, *draft* 11 (Février 1991) en cours d'analyse

  -- Pour 1003.2a, *draft* 6 (Mars 1991) en cours d'analyse

  -- Comprend le langage de commande et les utilitaires (*Shell Command*

Language and Utilities), un environnement de développement C et Fortran (facultatif) et un ensemble appelé *User Portability Extension*

- **IEEE POSIX 1003.3**

  -- Méthodologie standard de tests de conformance Posix (1003.3.1 et 1003.3.2)

- **IEEE Posix 1003.4 et 1003.4a**

  -- 1003.4 : Extension temps réel pour un système d'exploitation portable

  -- 1003.4a : Support du *multi-threading*

  -- Pour 1003.4, *draft* 10 (Février 1991) en cours d'analyse

  -- Pour 1003.4a, *draft* 5 (Février 1991) en cours d'analyse

-- 1003.4 comprend le support pour :

> des communications inter-*processes* ameliorées

> un *scheduling* standardisé

> des primitives de contrôle du *memory management*

> des opérations d'entrées/sorties asynchrones

> des améliorations de robustesse et de déterminisme du *file system*

-- Ces primitives ne sont accessibles que depuis le **langage C**

Les aspects POSIX 1003.4a ne seront pas traités ici pour des raisons de temps

VIP

- ***VMS Integrated POSIX***
- Projet commencé en 1987 dans l' *Open Systems Group* (ULTRIX)
- Déplacé en 1988 vers VMS engineering
- Un projet de **très haute priorité** pour VMS
- Développé à Varese en Italie
- DIGITAL s'est engagé publiquement le 25 octobre 1990 :

Annonce du 25 Octobre

DIGITAL s'est engagé à délivrer pendant le courant de l'année 1991 :

- POSIX 1003.1, 1003.2, 1003.4
- MIT X11R4 et OSF/Motif
- DECnet/OSI et TCP/IP
- Les parties où DIGITAL a été retenu comme contributeur par l'OSF :

  > *Naming* - ***DNS, X.500***
  > *Time* - ***DTS***
  > ***RPC - NCS***
  > *Threads* - ***CMA***

DIGITAL s'est de plus engagé à obtenir le ***branding*** de X/Open (XPG3)

Différences de philosophies *entre VMS et POSIX*

- Le modèle d'entrées/sorties :
  - -- Fichiers et périphériques
  - -- Primitives d'entrées/sorties
  - -- Le cache
- Sécurité et protection d'objets
- Création et contrôle des *processes*
- Interface utilisateur : *CLI/Shell*

Le modèle d'entrées/sorties

- Dans un modèle POSIX, *tout est un fichier*
- Il y a plusieurs types de fichiers :
  - -- fichiers normaux (*regular files*)
  - -- fichiers *directory*
  - -- *pipes* (appelés aussi *FIFOs*)
  - -- fichiers spéciaux péripheriques blocs (*file oriented devices*)
  - -- fichiers spéciaux péripheriques caractères comme disques, bandes et terminaux (*record oriented devices*)

- Pas de concept d'enregistrement : le modèle adopté par POSIX est le modèle d'une suite de bytes adressée aléatoirement (bien sur, toutes les suites de bytes ne sont pas adessables de façon aléatoire : un terminal par exemple)

- Un seul ensemble de primitives système pour les entrées/sorties :

  > open
  > close
  > read
  > write
  > lseek (positionnement dans la suite de bytes)

- Les services système *ioctl* (I/O control) permettent de dépasser l'abstraction de la notion de fichier pour les periphériques "réels" (exemple : rembobiner une bande)

Le cache

L'entrée/sortie la plus rapide est celle qui n'est pas faite...

- Traditionnellement, les implantations UNIX utilisent un cache de type *write-back* pour les entrées/sorties en mode block. Des *buffers* sont maintenus dans une liste de type LRU et le noyau tente de garder les données utiles dans ce cache aussi longtemps que possible

- Par exemple, quand ULTRIX démarre, il réserve 10% de la mémoire physique pour ce cache

- Puisque toutes les entrées/sorties utilisent le cache, de nombreuses applications écrivent dans des fichiers temporaires, relisent ces fichiers et les suppriment (exemple: le compilateur C)

- VMS, dans ce cas utilise ses possibilités sophistiquées de gestion de mémoire virtuelle

- Ce type de cache conduit potentiellement à une amélioration considérable des performances d'entrées/sorties mais a l'inconvénient de perte potentielle de données lors d'un *crash* système (pour diminuer ce risque, ce cache est vidé sur disque toutes les 30 secondes par défaut)

- De plus, pour limiter de tels inconvénients, ULTRIX et BSD par exemple, effectuent les modifications de *directories* de façon synchrone

- Cependant, une telle implantation est inacceptable pour VMS (hors *container* : voir plus loin*)* car pouvant conduire à des pertes ou corruptions de données

- Il résulte de ce type de présupposés, des philosophies de développement de programmes relativement différentes entre VMS et POSIX

Sécurité

- Contrairement aux mythes, UNIX n'est pas de façon obligatoire un système non sécurisé : il existe des implantations sécurisées d'UNIX (exemple : ULTRIX V4.0 sera classifié C2 par le DoD)

- Cependant, comme pour tout système, le *system management* est la clé...

La protection des objets

- POSIX ne nécessite qu'un niveau de protection de base fondé sur les notions de *user-id* et *group-id*

- Il y a 3 classes de protections : *user*, *group* et *other.* Contrairement à la protection de VMS liée aux UICs, l'accès est validé contre l'une de ces 3 classes (la plus "haute" pour laquelle le *process* est éligible)

- Les fichiers sont propriétés d'un *user-id* et d'un *group-id* particulier

- Les *processes* ont 2 *user-ids,* le "vrai" *user-id* et le *user-id* "effectif". C'est le *user-id* "effectif" (ou le *group-id* "effectif"*)* qui est utilisé pour déterminer l'accès

- La protection sur les exécutables peut être définie de telle façon que lorsque tourne cet exécutable, le *user-id* "effectif" du *process* soit celui du propriétaire de cet exécutable

Création de processes

- Les nouveaux *processes* sont crées par le service système ***fork*** : ce *process* nouvellement créé a son propre espace virtuel

- Ce service système crée un **clone** du *process* appelant (excepté pour quelques exceptions définies dans 1003.1; dans le cas de ces exceptions les *processes* résident dans l'espace système). Ainsi, le contenu de l'espace virtuel du *process* est le même pour les 2 *processes* quand *fork* a terminé son travail et retourne à la routine d'appel

- Dans chaque *process*, l'exécution redémarre avec l'instruction suivant le CMKRNL qui a causé le *fork*. Le *process* parent reçoit le *process-id* du *process* fils comme valeur de retour

alors que le *process* fils reçoit la valeur 0 (qui a la garantie de ne pas être un *process-id*)

- Les 2 *processes* s'exécutent en parallèle

- Le service système ***exec*** effectue quelque chose d'analogue à un *image rundown* suivi d'une *image activation* :

  -- vérification du droit d'accès en exécution pour l'image demandée

  -- suppression de l'espace virtuel courant du *process*

  -- création d'un nouvel espace virtuel initialisé avec le contenu de l'image

  -- exécution de l'image à son point d'entrée

- La plupart des *forks* sont suivis d'un *exec* : par exemple, cela permet d'utiliser le *shell*

Les performances de *fork* :

- Un *shell process* représente sous ULTRIX un espace virtuel d'environ 10Kbytes de données à comparer avec l'espace virtuel d'un *process* sous VMS qui n'exécute pas d' image : 700 Kbytes

- Il ne faut donc pas espérer avoir des performances comparables entre *fork* sous VMS/POSIX et sous ULTRIX sur un "petit" *benchmark*

- Il y aura cependant peu de différences visibles sur un "gros" *benchmark*

Contrôle des processes

- POSIX introduit la notion de *signal* qui combine des élements d'ASTs et de *condition handlers*. Les *signals* sont au software ce que les interruptions sont au *hardware*

- Les *processes* peuvent envoyer des *signals à* un autre *process.* C'est de cette façon que le *job control* (au sens POSIX du terme) est implanté et qu'un *process* peut "tuer" un autre *process*

- *Signals* :

  -- SIGINT (signal d'*attention* intéractif comme le control-C)

  -- SIGSEGV (*segmentation violation*, comme un ACCVIO)

-- SIGALRM (alarme, comme $SETIMR)

-- SIGTSTP/SIGVONT (*signals* utilisés par *job control* et pour les opérations de type *suspend/resume*)

-- SIGKILL (arrêt de *process*)

-- ...

- Pour chaque *signal*, un *process* a 3 options disponibles :

  -- Ignorer le *signal* complètement

  -- Accepter l'action par défaut associée avec le *signal*

  -- Établir un *condition handler*

- D'habitude, le défaut consiste à terminer le *process.* Les *signals* qui représentent des conditions d'erreurs conduisent le

système à l'écriture d'un *core dump* du *process*

- Les *signals* ne sont pas cumulatifs : si plusieurs *signals* du même type sont envoyés à un *process*, un seul est délivré; toutefois, si les *signals* sont différents, leurs effets sont cumulatifs

- L'affirmation précédente est valable pour 1003.1; 1003.4 introduit depuis le *draft* 10 la notion de *reliable signal*

Jobs VMS et groupes de processes POSIX

- Le concept VMS de *Jobs* avec *processes* et *sous-processes* est différent de la notion POSIX de *process session* ou de *process group*

- Dans un *job*, si le *process* principal meurt, tous les *sous-processes* meurent en même temps. Ce n'est pas le cas dans un environnement POSIX : un *process* fils peut survivre à la mort de son père

- Une *session* POSIX est composée d'une arborescence de *processes* dans laquelle le premier est appelé *session leader* : il y a une primitive explicite (setsid()) permettant d'établir un *process*

comme *session leader*

- Les *processes* dans une *session* peuvent être classifiés en 2 types en fonction de la relation qu'ils ont avec le *tty* avec lequel ils travaillent :

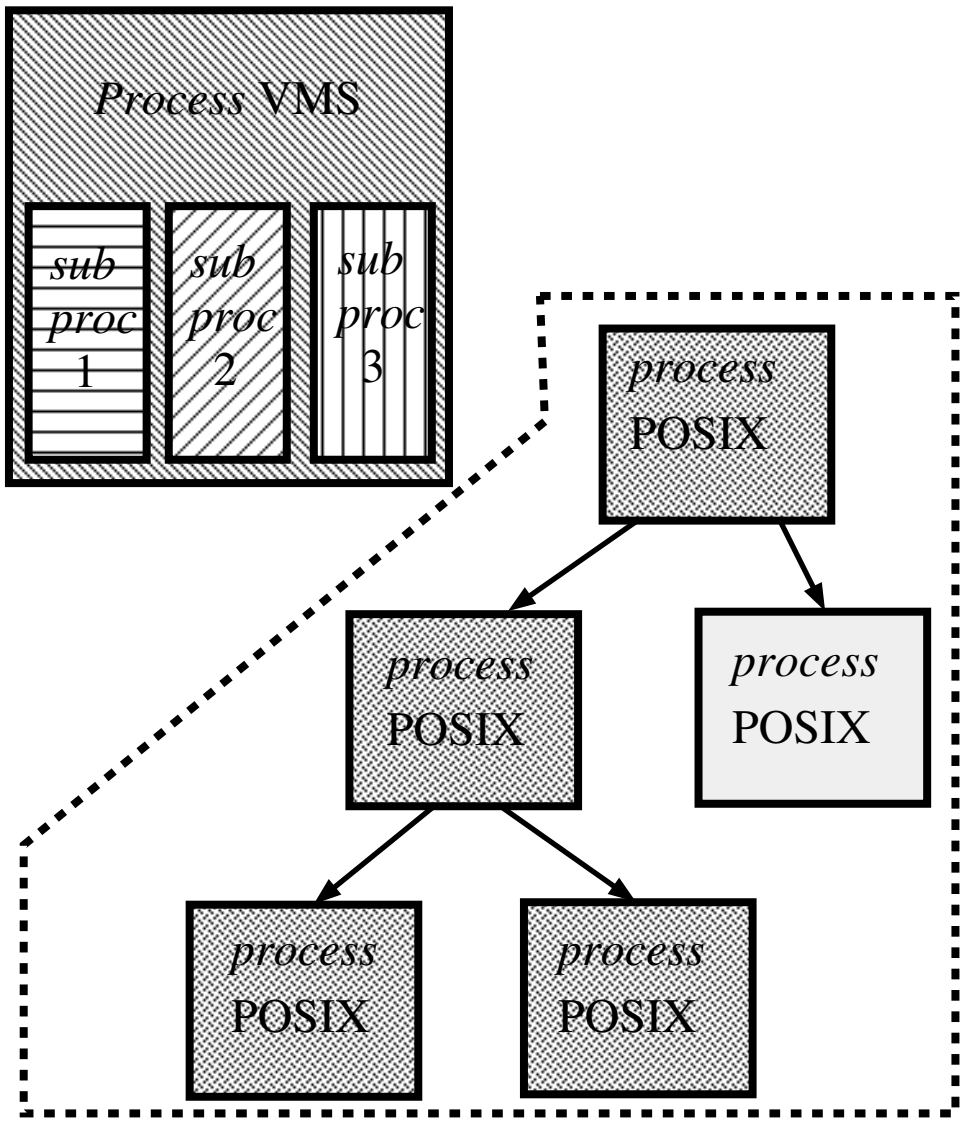
  -- *foreground process group* : reçoit les informations de type *out of band* depuis le *tty* (caractères de contrôle)

  -- *background process group* : ne reçoit aucun *signal* des *ttys*

- A travers des primitives spécifiques du *job control*, il est possible de changer dynamiquement les *processes* de *group*

- Quand un *hangup* est reçu par le tty ou par le *session leader*, le *signal* SIGHUP est envoyé à tous les *processes* du *foreground process group*, MAIS pas à ceux du *background process group*

Jobs VMS et groupes de processes POSIX



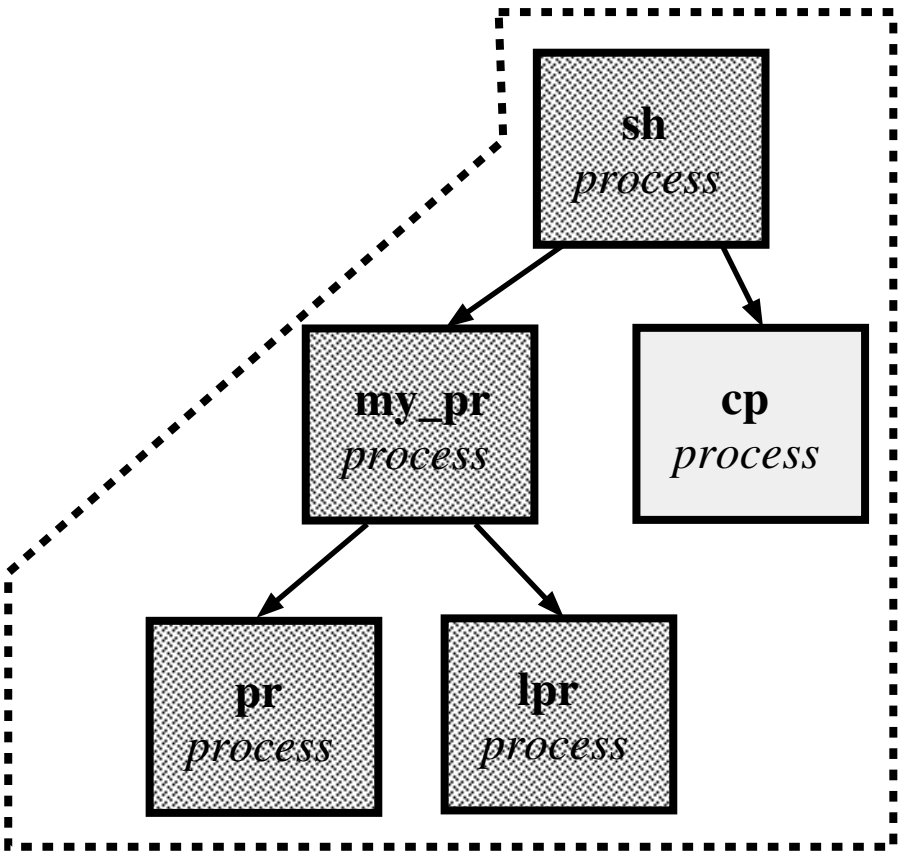
Jobs VMS et groupes de processes POSIX

sh> cp long_file another.long.file **&**

sh> cat my_pr
#! /bin/sh

pr | lpr

sh> **my_pr** my_file



Le Shell

- Le *shell* est l'interpréteur de commande en mode ligne. Toutefois, contrairement au DCL, il s'exécute dans son propre *process*

- Le fait d'être un *process* séparé a des avantages et des inconvenients :

  -- Avantages :

  > on peut exécuter plus d'une image avec une seule ligne de commande

  > il n'y a pas nécessité d'utiliser un mode d'accès séparé pour assurer l'intégrité du CLI

-- Désavantages :

> Au moins une paire de (*fork*, *exec*) pour chaque ligne de commande

> Le *shell* ne peut rendre de services aux images comme DCL peut le faire à travers les routines CLI$

Pipes

- Une des principales innovations venant d'UNIX
- Semblables aux boites aux lettres si ce n'est qu'un *pipe* est unidirectionnel
- Un *pipe* permet à la sortie d'un *process* d'être utilisée comme l'entrée d'un autre *process*
- Comme tous les autres I/O *streams*, les *pipes* n'ont pas d'organisation en enregistrement : les frontières de données ne sont pas préservées : si le *process* "ecrivain" écrit 4 blocks de 10 bytes, et que le *process* "lecteur" demande à lire 40 bytes, il recevra toutes les données en 1 fois

- Les intérets principaux liés à l'utilisation des *pipes* plutôt que des fichiers de sortie temporaire sont liés au fait que :

  -- les *process* s'executent en parallèle

  -- il n'y a pas d'I/Os générées

POSIX et VMS

	POSIX	VMS
Espace adressable	text (code) data stack	P0 P1 Autres → Locks, Global S., Mailboxes, ...
Modes d'exécution	user, kernel	user, supervisor, executive, kernel

POSIX et VMS

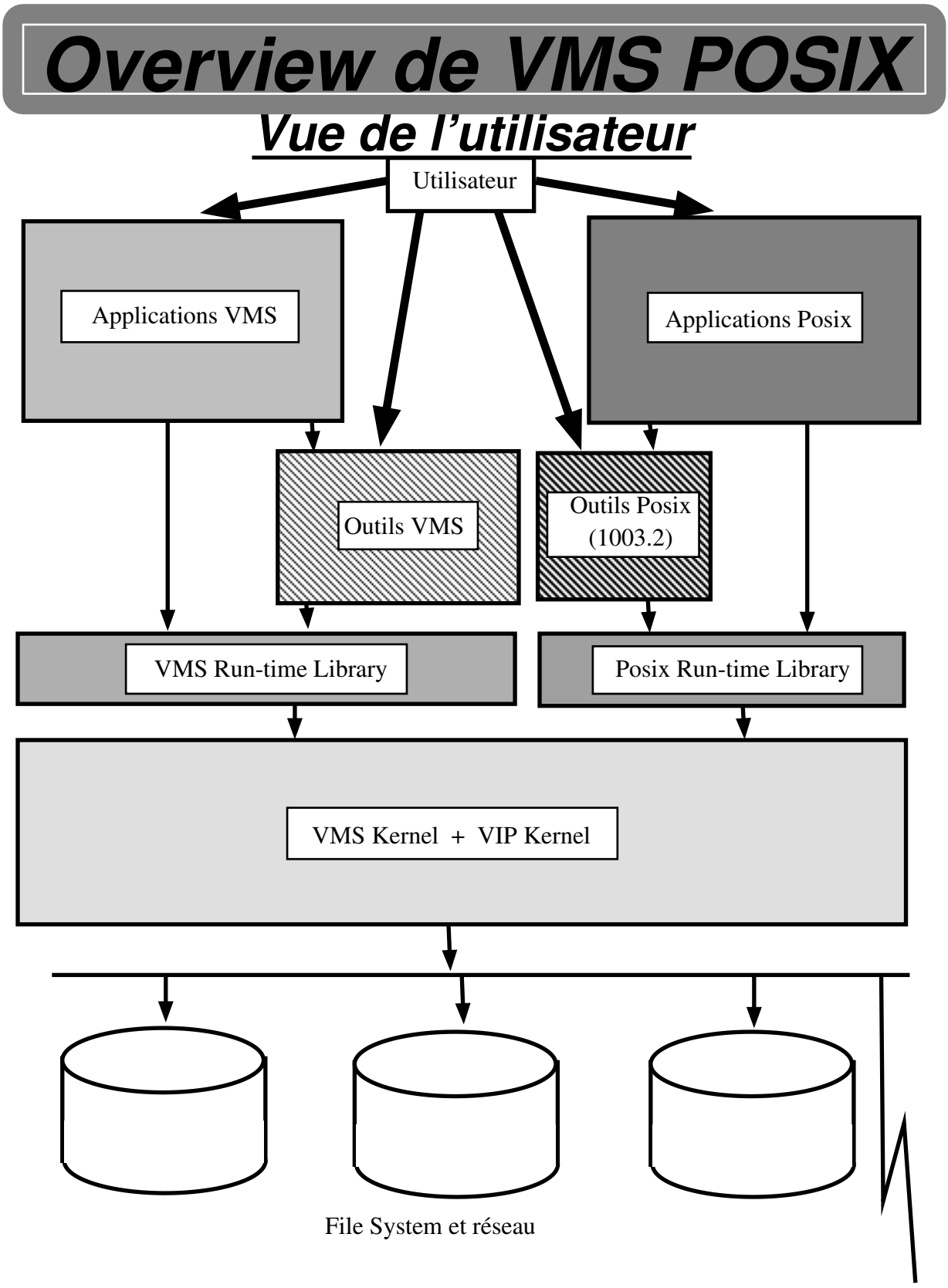
	POSIX	VMS
Survivance du *process* à l'image	Non	Oui
Passage d'Information	Environment	Noms logiques
Shell	*Process* separé	Partie du même *process* (dans P1)
Cache du file system	System wide	Privé au *process*

POSIX et VMS

Fork	$CREPRC
"Clones" d'un *process* existant	Création d'un nouveau *process* "from scratch"
Rapide (~20 ms sur un mvax II)	Lent (~150 ms sur un mvax II)
Contrôles spéciaux sur le PID généré	Différents types de contrôles
Peu de structures de données "privées" à copier	Beaucoup de structures de données à créer
Problèmes de synchronisation entre "parent" et "child"	Pas de problèmes synchronisation
"Touchup" de l'espace P1	Espace P1 généré "from scratch"

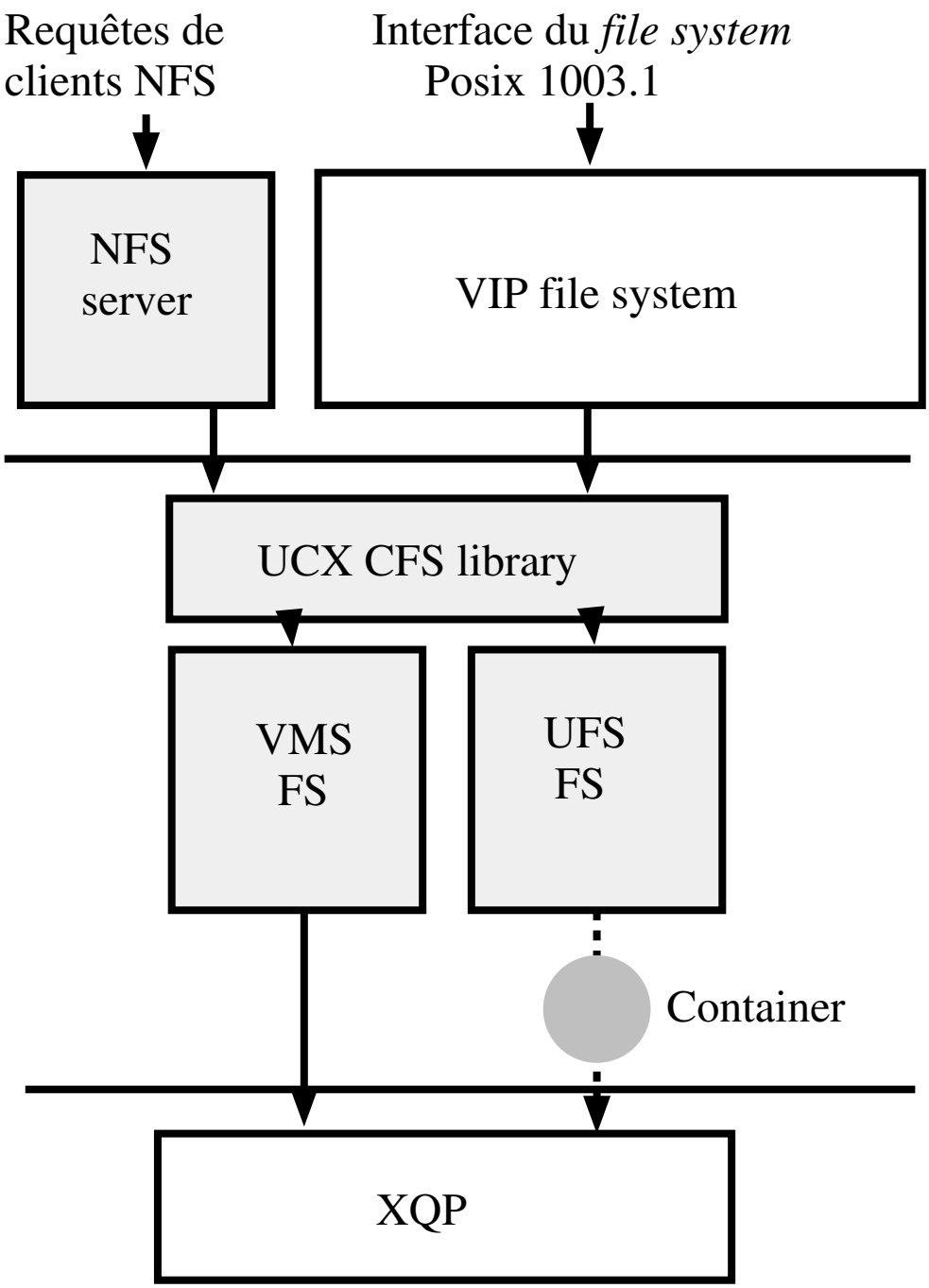
POSIX et VMS

Posix IPC	VMS IPC
Signals, shared memory, events	Mailboxes, AST's, Locks, event flags, $IPC, global sections
Rapide, peu de fonctionnalités	Beaucoup de fonction-nalités, performances variées
"Signals" utilisés pour différents propos	Chaque besoin a sa propre fonction


Overview de VMS POSIX
Vue de l'utilisateur
Utilisateur
Applications VMS
Applications Posix
Outils VMS
Outils Posix
(1003.2)
VMS Run-time Library
Posix Run-time Library
VMS Kernel + VIP Kernel
File System et réseau

Overview de VMS POSIX

Le file system



Utilisation de VMS POSIX

Opérations de base

Exemple de login :

```
Username: BERNARD
Password:

Welcome to VAX/VMS version V5.4 on node SOS4
Last interactive login on Wednesday, 19-MAR-1991 16:41

$ DIR

Directory USER1:[BERNARD]

MAIL.MAI;2    LOGIN.COM;4          DOC.DIR;1

Total of 3 files.

$ POSIX  POSIX$BIN:LS.

mail.mai
login.com
doc.dir
```

Utilisation de VMS POSIX
Opérations de base

Accès à des directories VMS :

```
$ DIR  [.DOC]

Directory $1$DUA21:[BERNARD.DOC]

CAP.DOC;7          CAP.DOC;6          CAP.DOC;2
FS_DESIGN.DOC;2
FS_DEVELOP.DOC;3                      FS_FUNCT.DOC;6
REQ1.DOC;38                           TEST.DOC;1

Total of 8 files.

$ POSIX/PATH=POSIX "/bin/ls" "-C doc"

cap.doc.7          cap.doc.6          cap.doc.2
fs_design.doc      fs_develop.doc     fs_funct.doc
req1.doc           test.doc

$
```

Acces aux fichiers VMS :

$ **TYPE** LOGIN.COM

```
$!
$ proc_name = f$trnlnm("TT")
$ set process/name = ""proc_name'"
$ exit
$!
```

$ **POSIX/PATH=POSIX "/bin/cat"** "login.com"

```
$!
$ proc_name = f$trnlnm("TT")
$ set process/name = ""proc_name'"
$ exit
$!
```

Utilisateur VMS type :

```
$ DIR
.....
$ POSIX POSIX$BIN:LS.
$ POSIX/PATH=POSIX "/bin/ls"

$ POSIX
        psx> ls
        psx> dcl "DIR"
        psx> exit

$POSIX
psx>
```

Utilisateur Posix type :

```
psx> vi test.c

psx> set -o emacs

psx> dcl "SHOW SYS" | grep HIB

psx> dir() { dcl DIR $1 }
```

Le CLI :

*Pour un utilisateur **VMS** :*

```
Username: bernard/CLI=POSIX$CLI
Password:
Welcome to VAX/VMS version V5.4 on node SOS3
        Last interactive login on Wednesday,
                19-MAR-1991 7:32
        Last non-interactive login on Tuesday,
                19-MAR-1991 4:31

psx >
```

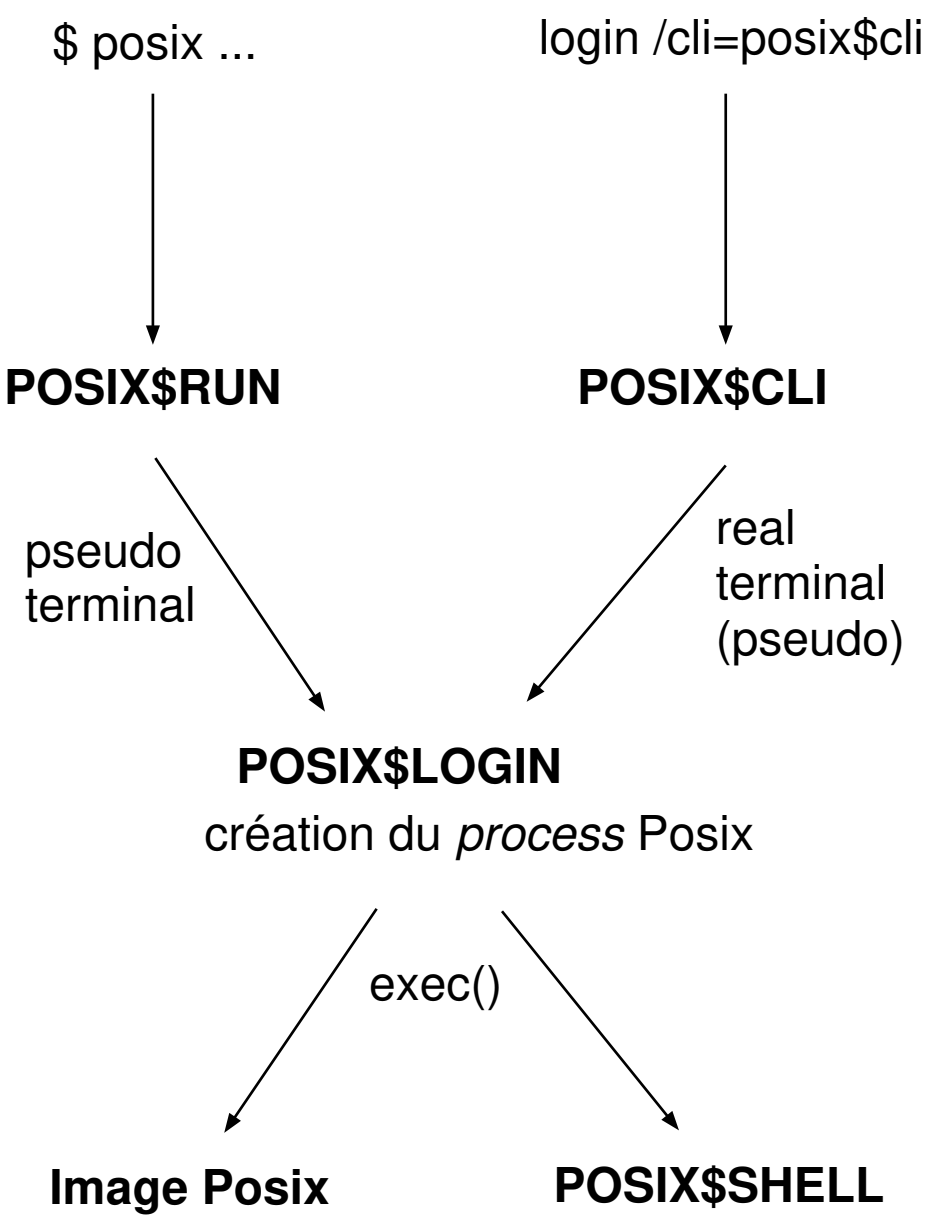
*ou pour un utilisateur **"Unix"** :*

```
Username: bernard
Password:
Welcome to VAX/VMS version V5.4 on node SOS3
        Last interactive login on Wednesday,
                19-MAR-1991 7:32
        Last non-interactive login on Tuesday,
                19-MAR-1991 4:31

psx >
```

POSIX 1003.1 : Overview

Séquence d'activation d'une application POSIX



POSIX 1003.1 : Overview

Notion de "stub"

$POSIX "application" "arguments"
[/IN=fichier][/OUT=fichier] ...

- Activé par le DCL
- Crée un pseudo-terminal (à travers le *pseudo-terminal driver* FTDRIVER)
- Crée un *process* détaché attaché à ce pseudo-terminal
- Crée un *session leader* pour le *process group*
- Positionne le *pseudo-tty* en mode POSIX
- Démarre l'application POSIX dans le nouveau *process*
- Gère SYS$INPUT et SYS$OUTPUT (fichiers RMS ou tty)

Pourquoi un "stub" ? :

S'il n'y avait pas de "stub" POSIX pour faire tourner une application depuis DCL :

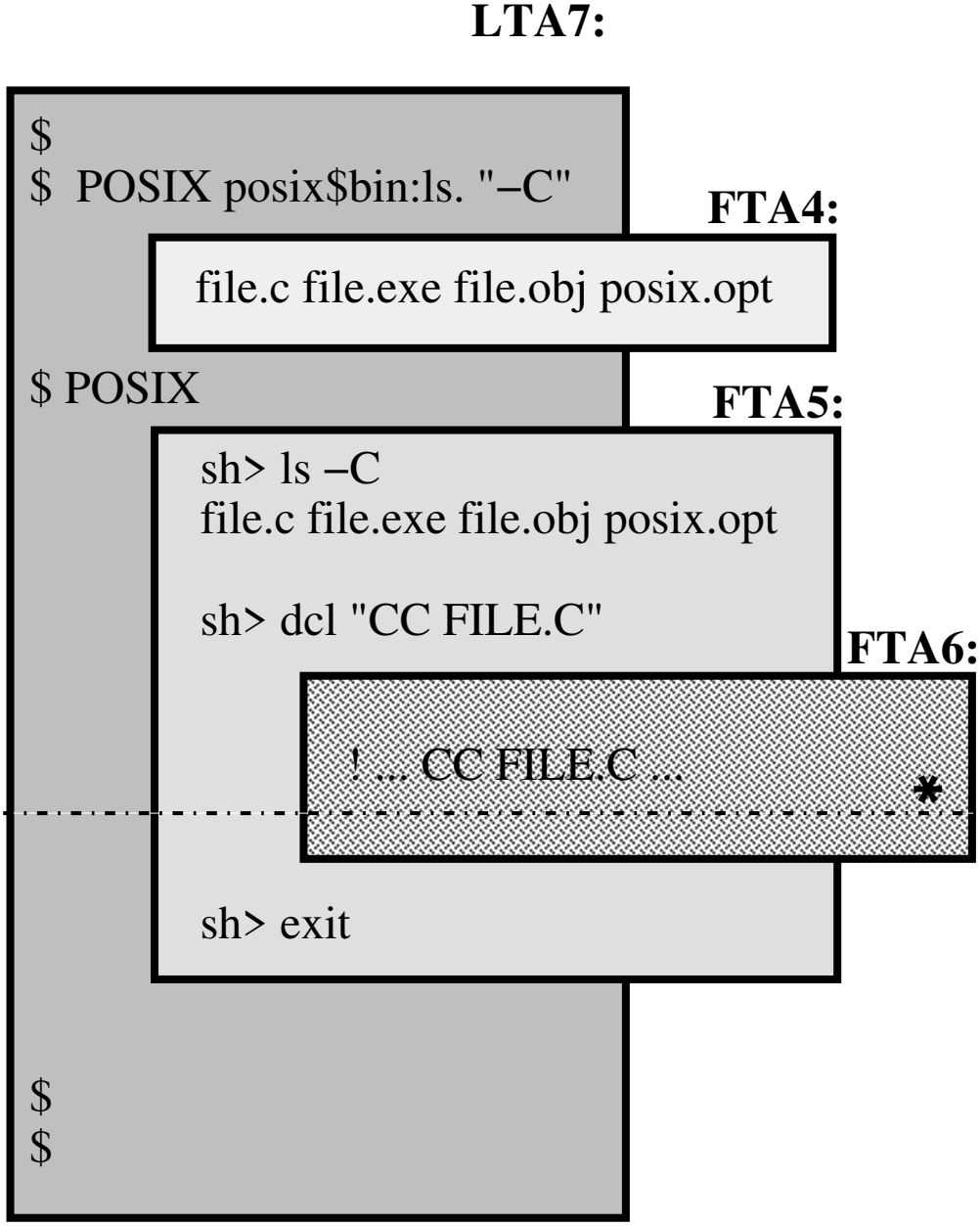
- Pas d'ouverture de *stdin, stdout, stderr*
- Pas de définition de *current/root directory*
- Pas de passage d'arguments et pas de variables d'environnement
- *fork()* et *signal* ne fonctionneraient pas
- Interaction tty de style VMS

S'il n'y avait pas de "stub" DCL pour faire tourner des applications depuis le *shell* :

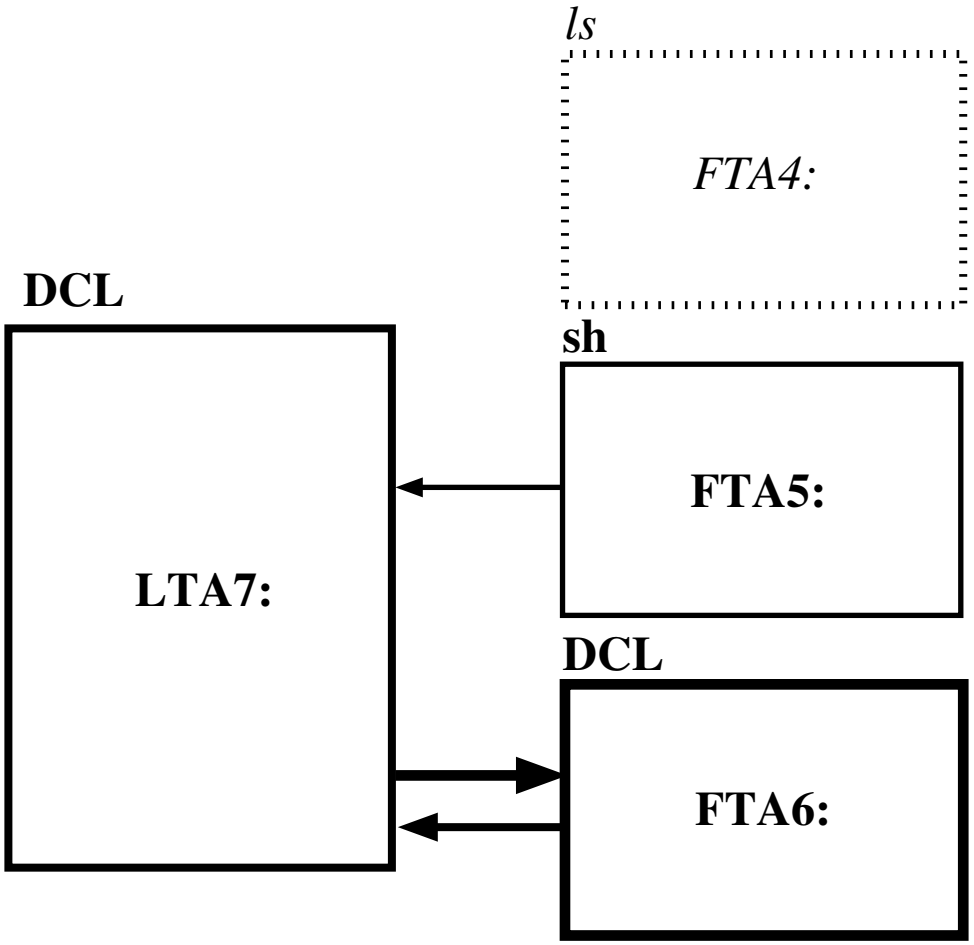
- Pas de définition de SYS$INPUT, SYS$OUTPUT, SYS$ERROR
- Pas de défauts RMS
- Pas de services du CLI
- Pas de gestion de *signals*
- Interaction tty de style POSIX

POSIX 1003.1 : Overview

TTY et Pseudo TTY



*** : on prend une image de ce qui se passe à ce moment**

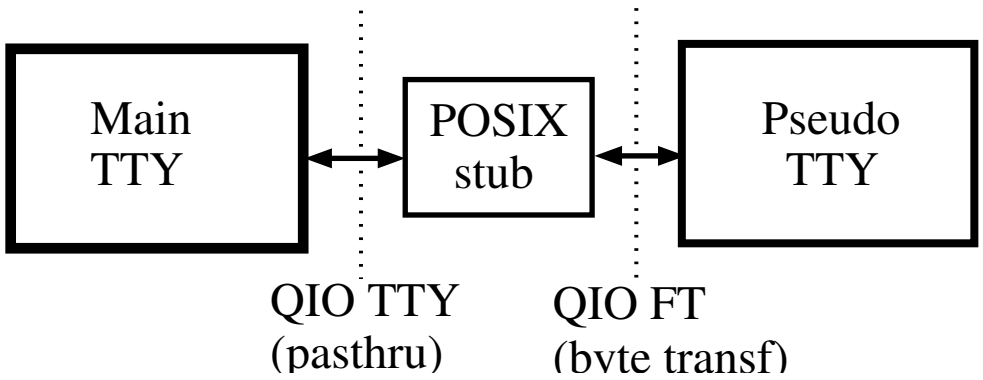


- FTA4: a été supprimé
- FTA6: a le contrôle

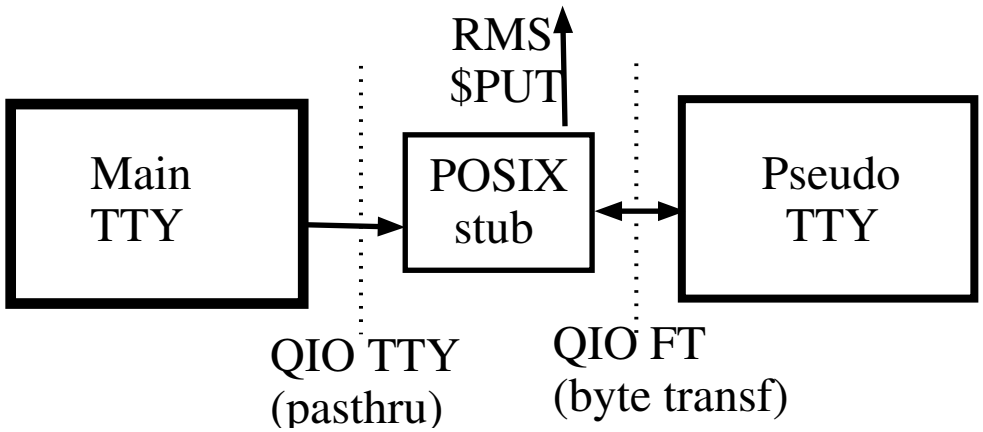
POSIX 1003.1 : Overview

FTDRIVER et le "stub"

$ POSIX /PATH=POSIX "/bin/ls"

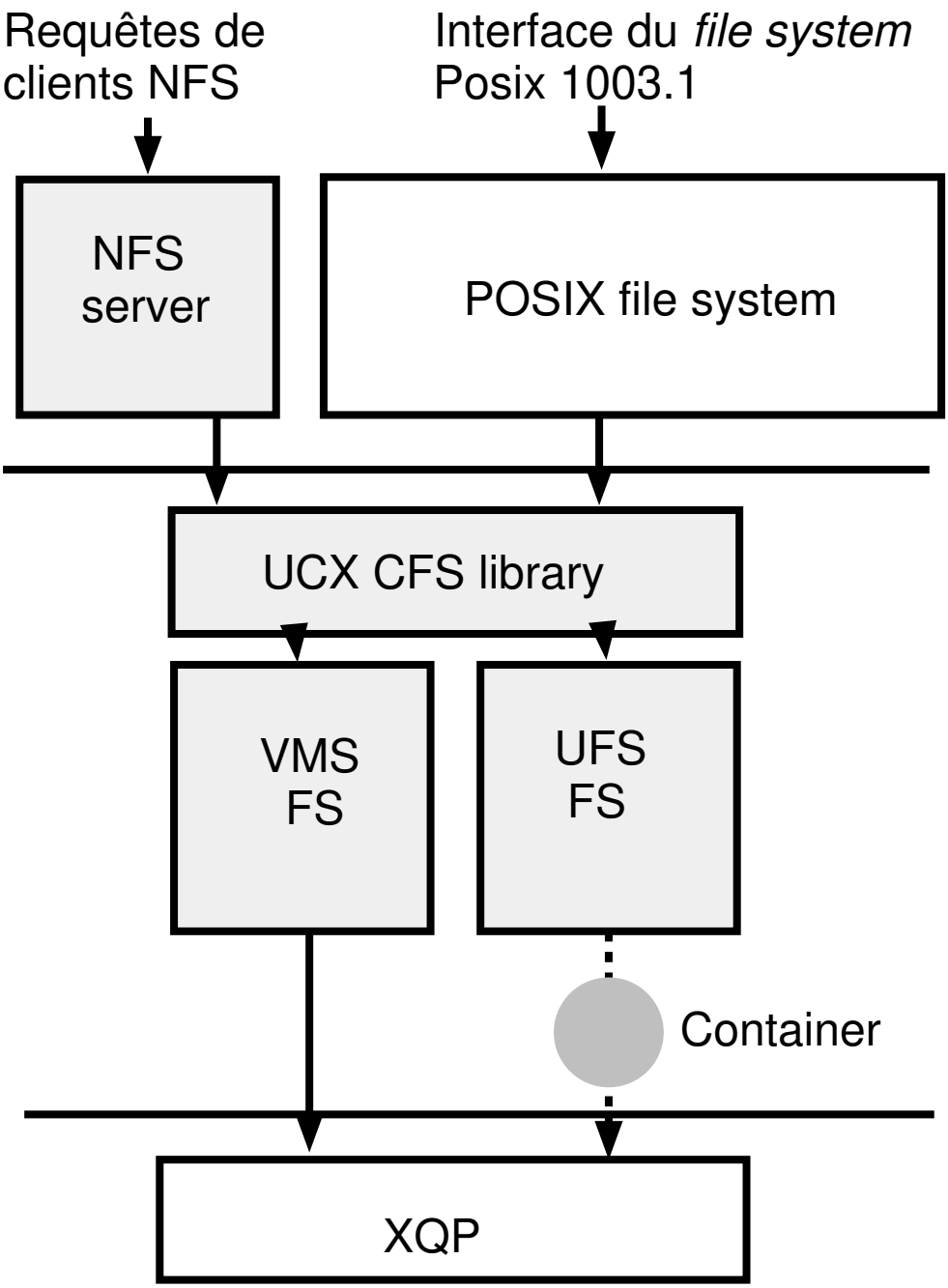


$ POSIX /PATH=POSIX/OUT=P.OUT "/bin/ls"



POSIX 1003.1 : Overview

Le file system



POSIX 1003.1 : Overview

Le file system

Couche "POSIX *file system*" :

- Procure interface et sémantique pour Posix 1003.1 & .4
- "Casse" les primitives Posix en taches élémentaires
  (open() -> lookup + assign channel)
- Fonctionne en mode EXEC, sur des structures de données privées du *process* allouées depuis un pool de buffers privés de VIP pouvant survivre à l'*image rundown* (exec())

Couche "UCX-CFS" :

- Éffectue les taches de base sur les fichiers et les *directories*

- Toutes les opérations sont coordonnées dans le *cluster*
- Fonctionne en mode EXEC, tous les services sont asynchrones et synchronisés avec ASTs, EFs et IOSBs

POSIX 1003.1 : Overview

Le file system VMS

Correspond directement au *File System* VMS traditionnel permettant l'accès à tous les fichiers à l'intérieur de stucture disque ODS2 : **INTEROPERABILITÉ**

- Noms *case insensitive*
- Noms appartenant à un *character set* limité et avec une longueur limitée
- Support des numéros de version
- Contrôle limité sur les versions
- Vérification d'accès au moyen des ACLs
- Présence d'un format d'enregistrement dans les fichiers
- Certains fichiers POSIX (special file) sont non supportés

- Certaines primitives POSIX sont non supportées
- Certains bits de protection POSIX sont non supportés

POSIX 1003.1 : Overview

Le file system POSIX

Un **Fichier Container** permet de mettre en correspondance les différences entre le *file system* VMS et les besoins POSIX : **CONFORMANCE**

- Correspondance 1 pour 1 entre les fichiers POSIX et les fichiers sur disque, mais avec des noms différents
- Correspondance 1 pour 1 entre les *directories* POSIX et les *directories* sur disque, mais dans une hiérarchie différente et avec des noms différent
- Le fichier **Container** gère les noms de fichiers ainsi que des informations supplémentaires du *file header*
- Conformance TOTALE à POSIX

- *Name Space* POSIX/Unix
- UID et GID POSIX/Unix avec support des *extended mode bits*
- *Symbolic links, hard links, special files, FIFO*
- *Time stamps* POSIX

POSIX 1003.1 : Overview

Root directory

Vue Posix

```
sh> ls -l /
dr-xr-xr-x 2 1   [...] vms
dr-xr-xr-x 2 1   [...] cont
lr-xr-xr-x 1 1   [...] tmp -> /cont/tmp
```

Vue UCX

```
$ ucx sho bind
dua0:                   "/vms"
dua0:[posix$cont]       "/cont"
```

Pathname ==> Nom de fichier

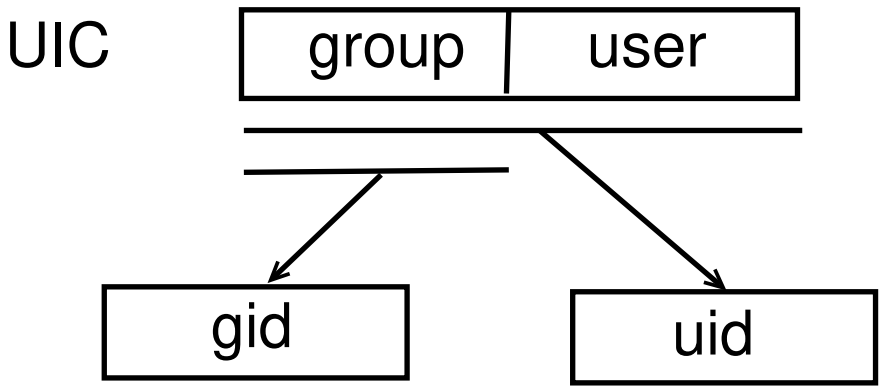
/**vms**/user/bernard/xxx/yyy.zzz ->
dua0:[user.bernard.xxx]yyy.zzz

/**cont**/tmp/a.b.c.d.e ->
dua0:[posix$cont.a301$bfs]876$bfs.;1

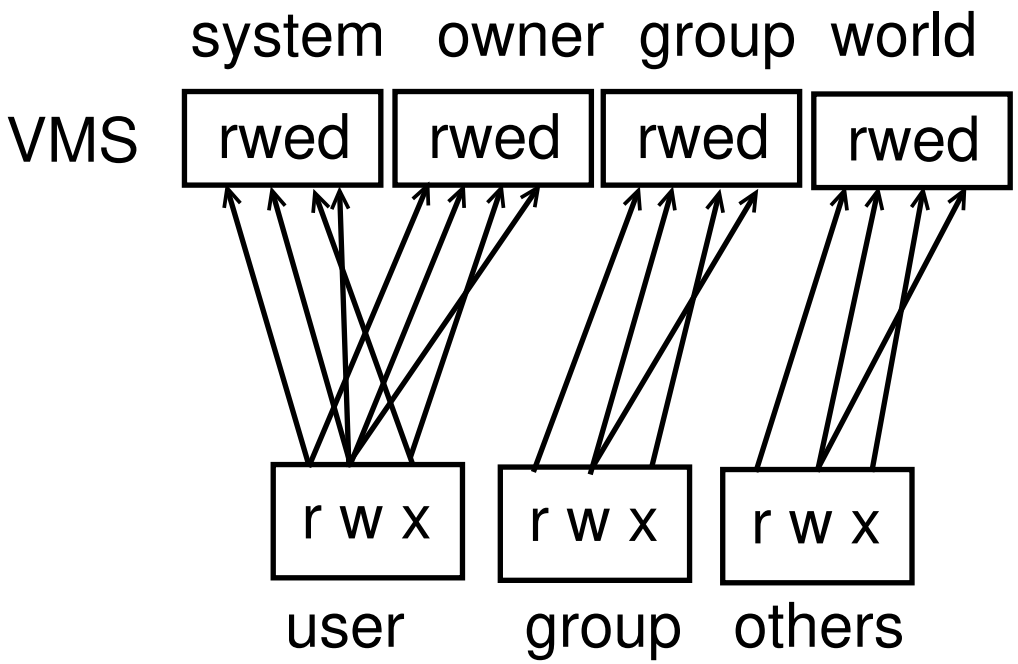
POSIX 1003.1 : Overview

Protections

Identité



Mode bits



POSIX 1003.1 : Overview

Interopérabilité

- RMS peut **"utiliser"** tous les fichiers créés par VIP (en utilisant le **nom** correspondant VMS)
- VIP peut **"utiliser"** tous les fichiers de n'importe quel volume VMS (en utilisant le **pathname** correspondant)

Il n'y a pas de synchronisation lors d'accès partagé entre RMS et VIP
(Le mécanisme POSIX de type *byte range locking* ne permet la synchronisation avec le *record locking* RMS)

Options du *File System* VIP

- Style VMS pour les *listings* des *directories*
- Création d'une nouvelle version lors d'un *file truncation*
- Interprétation des structures d'enregistrement (lors de la lecture du fichier)
- Options mises en oeuvre en fonction des paramètres contenus dans le fichier sys$login::posix$user_defaults.dat (par défaut)
- Ces options sont visibles par les commandes :
  - -- POSIX/SHOW=
    - > FILE_SYSTEM
    - > ALL

POSIX 1003.1 : Overview

Commandes DCL POSIX

- La commande DCL $POSIX admet les *qualifiers* suivants :
  - -- /RUN [imagename] [args]
  - -- /SET [parameter]
  - -- /SHOW [parameter]
- Ces 3 *qualifiers* sont mutuellement exclusifs
- /RUN est le défaut
- Le *qualifier* /PATH= [NO]POSIX permet de spécifier le type de nom de fichier utilisé par la commande : POSIX ou VMS
- La commande $POSIX/RUN accepte un sous-ensemble des *qualifiers* acceptés pour la création de *processes* détachés

- $POSIX/SHOW=
  - -- FILE_SYSTEM : options *file system* (voir plus haut)
  - -- ENVIRONMENT : variables d'environnement pour le *shell*
  - -- SHELL : *shell* par défaut
  - -- SUPPLEMENTARY_GID : *group IDs* supplémentaires donnés à l'utilisateur
  - -- QUOTA : quotas spécifiques POSIX
  - -- ALL (défaut)
- POSIX/SET USER_DEFAULTS
  [/FILE=filespec]
  permet de relire le fichier des défauts POSIX de l'utilisateur contenus dans le fichier sys$login:posix$user_defaults.dat (par défaut)
- Ce fichier contient des informations sur :
  - -- les options du *file system*
  - -- les variables d'environnement

-- les correspondances de caractères de contrôle (*intr, quit, erase, kill,*...)

-- le *shell* par défaut

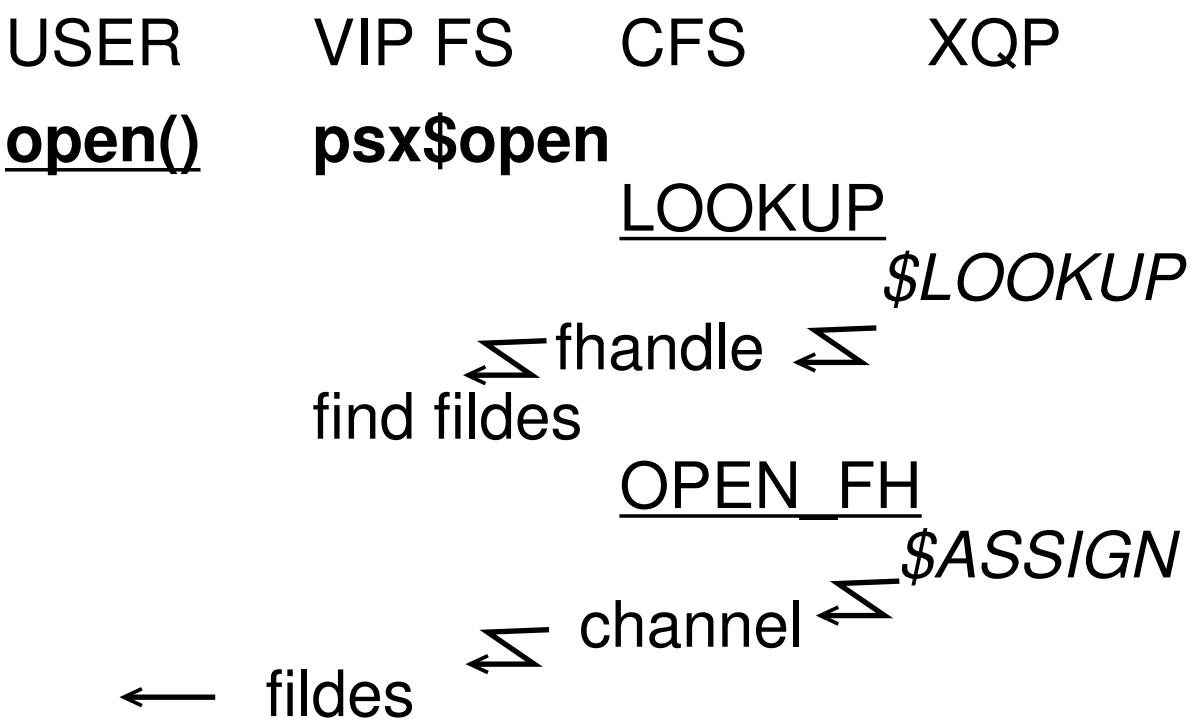
POSIX 1003.1 : Overview

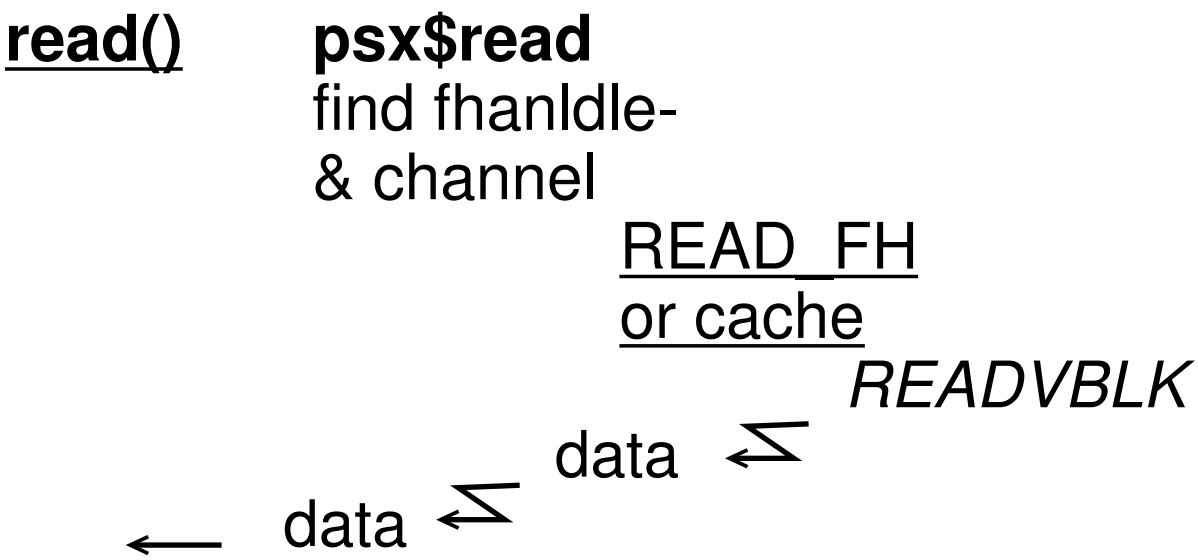
Fichiers spéciaux

- *"Symbolic links"*
- *FIFOs*
- *(pipes)*
- *Character special files*
  - *-- /dev/null*
  - *-- /dev/tty*
- *Block special files*

POSIX 1003.1 : Overview

Contrôle de Flux




read()
psx$read
find fhanldle-
& channel
READ_FH
or cache
READVBLK
data
data

POSIX 1003.1 : Overview

Fonctionnement interne du file system

Le *file system* VIP gère :

- L'émulation de la *root directory*
- La table des fichiers ouverts
- Les informations relatives au *pathname parsing*
- Les *locks nécessaires* et les strucures de données

La plupart des structures de données sont allouées dans une *pool* VIP spécifique en P1 afin de survivre à l'*image rundown*

La couche CFS gère :

- Les structures de données pour le *file system,* les fichiers, et les blocks de données

- Les locks VMS pour coordonner les accès cluster
- Les buffers de données (write back, write through, flush on close!!!)
- Les attributs de *pathname* et de cache

Ces structures de données sont allouées dans une *pool* spécifique VIP en P1, et les *buffers* de données sont créés en P0

POSIX 1003.1 : Overview

Les modifications d'UCX

- Déplacement de CFS vers l'espace système afin de survivre à l'*image rundown* : *sharable library --> execlet*
- Modification des mécanismes d'allocation de données
- Extensions pour la conformance 1003.1 & 1003.4
- Support de *fork()*

POSIX 1003.1 : Overview

Les primitives process

pid_t fork()

Description : "*The fork() function creates a new process. The new process (child process) shall be an exact copy of the calling process (parent process)...*"

Un *process* VMS est constitué de :

- Contexte Hardware : pc, psl, r0..r11, ap, fp, usp, ssp, esp, ksp...
- Contexte Software : pcb, phd...
- Espace virtuel : p0, p1

Le problème posé par l'implantation de *fork()* sous VMS est de :

cloner toutes les informations mais effectuer un "touch up" afin de **préserver** l'**intégrité** de VMS

POSIX 1003.1 : Overview

Création de process

Le *child process* est créé par le *process* parent comme un *process* **detaché** car il doit survivre à la mort du parent

Les actions du parent :

- (1) Vérification des restrictions : RMS, CEF, *writable sections* privées
- (2) Création du *child process* en COMO
- (3) Appel du *swapper* pour démarrer le *process* fils et suspension du parent
- (4) Le *swapper* crée le *process* fils minimal, réveille le parent et suspend le *process* fils
- (5) Allocation et duplication des structures de données: pcb, phd, jib, xpcb

- (6) Clonage de l'espace virtuel : p0 + p1 pte par pte
- (7) Le *process* parent réveille le *process* fils et se suspend lui-même

Les actions du *child* :

- (8) *channels touchup* : les CCBs sont hérités par le parent
- (9) "touchup" : *file system, CTL locations*
- (10) Le *process* fils réveille le *process* parent

Les spécificités POSIX :

- Les programmes écrits en utilisant seulement les services Posix fonctionnent sans problème quand *fork()* est exécuté

Les spécificités VMS :

- *fork()* est un nouveau concept pour VMS
- Le logiciel VMS existant a été écrit sans prendre *fork()* en considération

Exemples :

- Locks: pas d'action (*invalid lock id* dans le *child process*)
- Noms logiques : les noms logiques privés du *process* sont transmis lors du *fork().* Pas les noms logiques *jobs*
- RMS: non supporté - *fork()* est interdit (seuls les fichiers ouvert à travers le file system Posix sont supportés)
- *Layered products*: pas d'action (ex: De-bugger)

VMS POSIX a créé la notion de *callback* :

- Création de *Callbacks* pour permettre au logiciel déjà existant de supporter *fork()*
- Déclarer des *callbacks* permet de fonctionner correctement à travers *fork()*
- Les *callbacks handlers* seront appelés à differents moments pendant le *fork()* afin de maintenir l'intégrité

POSIX 1003.1 : Overview

Exécution de process

La famille de fonctions exec() remplace l'image du *process* courant avec une nouvelle image.

L'image s'exécutant dans le *process* appelant est supperposée par la nouvelle image.

Flux :

- exécution des *exit handlers*
- vérification du fait que le fichier est un *executable*
- fermeture du fichier marqué CLOSE_ON_EXEC
- *rundown* de l'image courante en mode *user*
- *image activation*
- *fixup*

POSIX 1003.1 : Overview

Attente de fin

pid_t wait(stat_loc)

pid_t waitpid(pid,stat_loc,options)

Ces fonctions permettent au *process* appelant d'obtenir des informations sur l'un de ses *child processes.*

wait() suspend l'exécution jusqu'à ce que le status de l'un de ses *child processes* (qui a fini de s'exécuter) soit disponible

waitpid() est similaire à wait() mais *pid* spécifie un ensemble de *child processes* dont on demande le status

Le *process* attend en état **HIB**

POSIX 1003.1 : Overview

Fin de process

_exit(status)

Termine le *process* appelant

La fin de vie du *process* ne fait pas "mourrir" ses *children processes* (un *child process* peut survivre à son *parent process*)

Flux :

- $exit
- $delprc:
  - -- exécution des routines de *rundown* spécifiques à POSIX
  - -- appel de POSIX pour désallouer la structure de données Posix XPCB et passage du *process* dans l'état "zombie" (état HIB)

POSIX 1003.1 : Overview

Signals

POSIX définit :

- Un ensemble de *signals:* SIGABORT, SIGALARM, SIGKILL...
- Un ensemble de fonctions pour gérer les *signals:*
  - -- kill() - envoit un *signal* à un *process*
  - -- sigaction() - examine et change l'action d'un *signal*
  - -- sigprocmask() - examine et change les *blocked signals*
  - -- sigpending() - examine les *signals* en attente
  - -- sigsuspend() - attend un *signal*

3 étapes :

- Génération: 3 façons de générer un *signal*
  - -- exception (Hw ou Sw)
  - -- kill()
  - -- depuis le clavier
- Envoi :
  - -- Exécuté dans le *process* émetteur
  - -- Un *signal* est mis en queue en direction du *process* destinataire
  - -- Génération d'une interruption ASTDEL en positionnant le registre VAX ASTLVL pour le mode *user* (comme pour une *user ast* mais pas d'ACB mise en queue)
- Délivrance :
  - -- Est exécutée dans le *process* destinataire

-- Le code d'interruption de VMS délivre le *signal* :

Si la queue d'ACB est vide

ET

Si le mode précédent est user

ET

Si un bit spécifique Posix est positionné

POSIX 1003.1 : Overview

Signals

Les *signals* et les ASTs ont le même mécanisme de délivrance mais :

- Les *asts* appartiennent à une classe d'interruptions plus haute que les *signals*
- Une *ast* peut interrompre n'importe quel *signal*
- Un *signal* ne peut pas être délivré quand une *ast* est active
- Si les *asts* ne peuvent être délivrées, alors les *signals* ne peuvent être délivrés
- Les *asts* peuvent être délivrées à tout *process,* les *signals* ne peuvent être délivrés qu'aux *processes* POSIX seulement

En résumé

Une AST (4 modes)

peut interrompre un

SIGNAL

qui peut interrompre

une exécution normale en *user mode*

POSIX 1003.1 : Overview

Temps process

clock_t times(buffer)

Remplit la structure pointée par un *buffer* contenant des informations de type *accounting*

Retourne l'*elapsed time* depuis un point arbitraire.

(EXE$GL_ABSTIM_TICS = nombre de ticks de 10ms depuis que le système a démarré)

```
struct tms {
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
};
```

POSIX utilise un modèle *user/system* composé de 2 modes d'exécution :

- user (non privilégié)
- system (privilégié)

POSIX sur VMS:

- user = user + supervisor
- system = executive + kernel

POSIX 1003.1 : Overview
Terminal

Input processing :

Avant d'être lues pas un *process,* les données qui arrivent sont stockées par le système dans une INPUT QUEUE

- MAX_INPUT (<sys/limits.h>) représente le nombre maximum de bytes qui peuvent être stockés dans l'*input queue*
- 2 types d'i*nput processing*: *Canonical Mode Input Processing* et *Non-Canonical Mode Input Processing*

ET VMS ?

- Les données qui arrivent sont stockées dans le *TYPE-AHEAD buffer*
- TTY_TYPAHDSZ/TTY_ALTYPAHD (paramètres SYSGEN) représentent la taille du *TYPE-AHEAD buffer*

POSIX 1003.1 : Overview
Terminal

Canonical Mode Input Processing :

- Les données en entrée d'un terminal sont gérées en unités de lignes
- MAX_CANON (<sys/limits.h>) definit la limite du nombre de bytes dans une ligne
- Les caractères en entrée sont validés en fonction d'une liste de caractères spéciaux comme suit :

Interactive *signals*	Flow control	Line editing	Terminators
INTR –> SIGINT	**STOP** –> suspend output	**ERASE** –> erase last char	**NL, EOF, EOL**
QUIT –> SIGQUIT	**START** –> resume output	**KILL** –> delete an entire line	complete the read
SUSP –> SIGTSTP			

ET VMS ?

- SIMILAIRE À VMS :
  - -- mode canonique <--> mode NOPASTHRU
  - -- MAX_CANON <--> MAXBUF
  - -- SUSP <--> CTRL-Y
  - -- ISIG <--> SET CONTROL=Y

- NOUVEAU PAR RAPPORT À VMS :
  - -- les caractères spéciaux peuvent être redéfinis
  - -- les caractères de contrôle qui ne sont pas des caractères spéciaux sont insérés dans l'INPUT QUEUE
- DIFFÉRENT DE VMS :
  - -- ensemble limité de terminateurs
  - -- ensemble limité de fonctions d'édition

POSIX 1003.1 : Overview
Terminal

Non-Canonical Mode Input Processing :

- Les bytes d'entrée ne sont pas assemblés en lignes
- Les valeurs de MIN et TIME sont utilisées pour déterminer comment traiter les bytes reçus
- MIN représente le nombre minimum de bytes qui doivent être reçus quand la fonction read() retourne
- TIME est un *timer* of granularité 0.1 seconde

Interactive signals	Flow control	Line editing	Terminators
INTR –> SIGINT **QUIT** –> SIGQUIT **SUSP** –> SIGTSTP	**STOP** –> suspend output **START** –> resume output	**DISABLED**	**MIN and TIME** ↓ MIN = 0 TIME > 0 MIN = 0 TIME = 0 MIN > 0 TIME > 0 MIN > 0 TIME = 0

ET VMS ?

- SIMILAIRE À VMS :
  - -- Mode non canonique <--> mode PASTHRU
  - -- MIN>0, TIME>0 <--> *$QIO read* avec le *modifier* IO$M_TIMED et un *time-out* <>0
  - -- MIN=0, TIME=0 <--> *$QIO read* avec le *modifier* IO$M_TIMED et un *timeout* =0

- NOUVEAU PAR RAPPORT À VMS :
  - -- MIN=0, TIME>0 : read() est satisfait dès qu'un seul byte est reçu ou que le *timer* a expiré
  - -- MIN>0, TIME=0 : read() est satisfait après la réception de MIN bytes
- DIFFÉRENT DE VMS :
  - -- Granularité de 0.1 seconde
  - -- INTR, QUIT et SUSP sont en fonction

POSIX 1003.1 : Overview
Terminal

Output Data Processing :

- Les données à écrire sont insérées dans un *buffer*
- Les données en sortie sont traitées en accord avec la valeur de OPOST (<sys/termios.h>)

ET VMS ?

- Les données à écrire sont insérées dans une queue
- Les données en sortie sont traitées en accord avec la présence ou non du *modifier* IO$M_NOFORMAT avec le *$QIO Write*

POSIX 1003.1 : Overview
Terminal

Support de type *dial-up* :

En cas de déconnection de modem

SIGHUP est envoyé au
process de contrôle

↓

le *process* de contrôle
envoit SIGHUP au
foreground process group

↓

les *processes* peuvent survivre
(s'ils ignorent SIGHUP)

ET VMS?

CTRL–Y AST avec SS$_HANGUP
est envoyé à DCL

↓

DCL est responsable de "tuer" le *process*

↓

les *processes* peuvent survivre
(virtual terminals)

POSIX 1003.1 : Overview
Terminal

Interface de programmation :

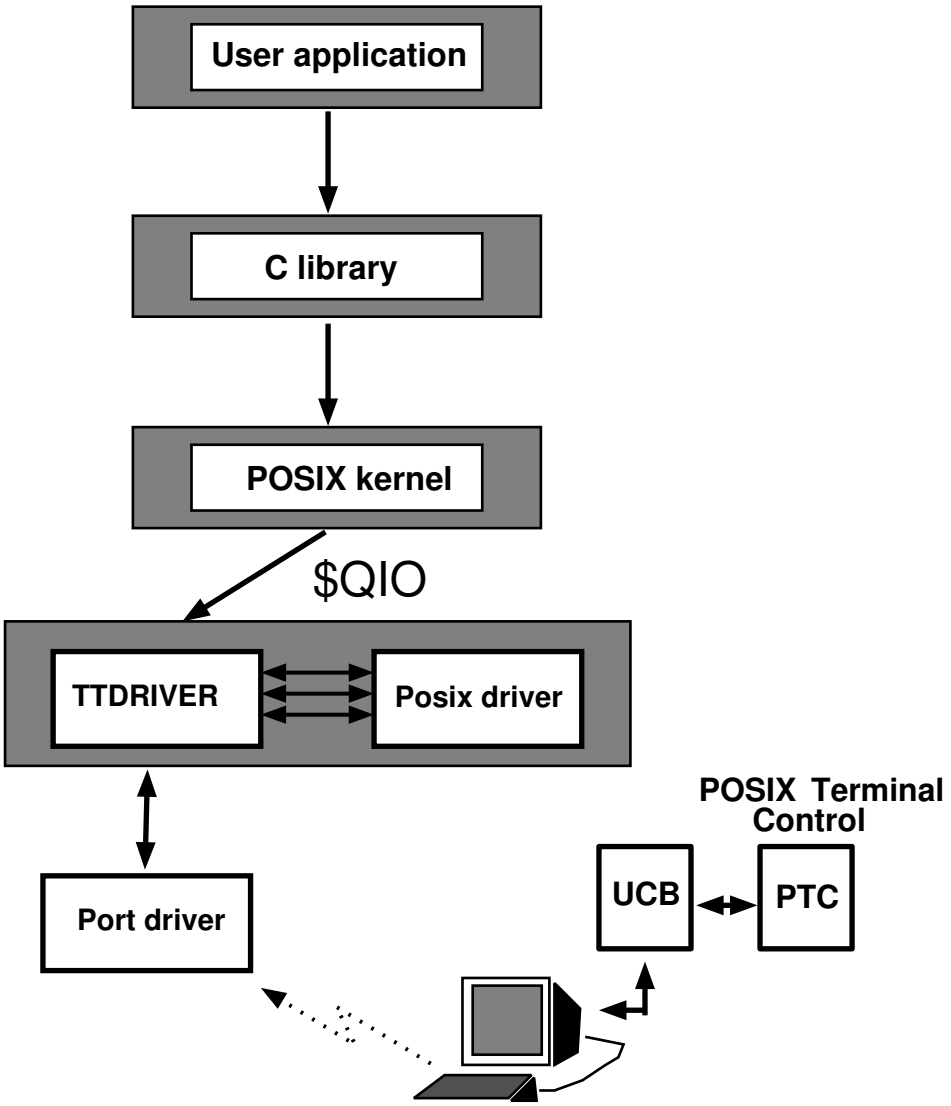
Fonctions et *include file:*

- Fonctions input/output :
  - *open(), read(), write(), close()*
- Fonctions de contrôle de vitesse :
  - *cfgetospeed(), cfsetospeed()*
  - *cfgetispeed(), cfsetispeed()*
- Fonctions de contrôle de terminal :
  - *tcgetattr(), tcsetattr()*
  - *tcsendbreak(), tcdrain(), tcflush(), tcflow()*
  - *tcgetpgrp(), tcsetpgrp()*

- Paramètres :

  -- *<sys/termios.h>*

Interactions avec VMS :



POSIX 1003.1 : Overview
Terminal

Restrictions, suggestions d'utilisation :

- MAX_CANON est en accord avec le paramètre SYSGEN MAXBUF
- Les caractères de contrôle (ex: CTRL-A,...) n'ont pas d'écho
- Les fonctions d'édition de ligne sont dépendantes du type actif (POSIX/VMS)
- Les caractéristiques de terminal POSIX sont en effet seulement quand une application POSIX s'exécute
- La plupart des attributs des terminaux POSIX réutilisent les caractéristiques existantes VMS : *il n'est donc pas conseillé de changer les caractérisques terminal en utilisant la commande DCL*

"SET TERMINAL" alors qu'une application POSIX tourne sur ce terminal

- Quand un pseudo terminal est utilisé, les changements des caractéristiques hardware du terminal (vitesse,...) n'affectent pas le terminal réel
- Si le terminal est libre ou positionné SECURE_SERVER, le comportement de sécurité VMS s'applique en cas de break, *frame/parity errors* (impératifs de sécurité VMS)

POSIX 1003.2 : Overview

Le standard

Définit un standard d'interface au niveau source permettant le support d'applications de type *shell scripts* (language de commande) et un certain nombre de programmes utilitaires

- Complémentaire à POSIX.1 bien que ne demandant pas explicitement que le système de base soit conforme à 1003.1
- Ne nécessite pas que les utilitaires soient écrits comme des programmes conformes à 1003.1
- Emphase mise sur le *shell* dans ses fonctionnalités utiles pour la programmation d'application et NON pour l'usage interactif

- Les utilitaires sont considérés et décrits comme des *boites noires* avec une spécification de la fonctionnalité mais pas de l'*implementation*
- Interface UNIX(TM)

(TM) UNIX est une marque déposée d'AT&T

POSIX 1003.2a : Overview

Le standard

Supplément au standard POSIX.2 définissant un environnement intéractif commun :

- Étend les fonctionnalités de quelques utilitaires de POSIX.2 afin de pouvoir les utiliser interactivement
- Étend la liste des utilitaires afin de bénéficier d'un meilleur environnement interactif
- Destiné aux *end-users*, développeurs de programmes et d'applications
- Standard à part qui devrait devenir intégrante du standard 1003.2 (chapitre 5) dans environ 2 ans

POSIX 1003.2 : Overview

Les composants

Composant dits *classic* :

- Le *shell* tel que décrit aux chapitres 3 et 4 de 1003.2
- Tous les utilitaires listés au chapitre 4

Liste des utilitaires 1003.2 *classic* :

awk	basename	bc	cat
cd	chgrp	chmod	chown
cksum	cmp	comm	cp
cut	date	dd	diff
dirname	echo	ed	env
expr	false	find	fold
getconf	getopts	grep	head
id	join	kill	ln
locale	localedef	logger	logname
lp	ls	mailx	mkdir
mkfifo	mv	nohup	od
paste	pathchk	pax	pr
printf	pwd	read	rm
rmdir	sed	sh	sleep
sort	stty	tail	tee

test	touch	tr	true
tty	umask	uname	uniq
wait	wc	xargs	

User Portability Extensions (1003.2a) :

Software Development Utilities :

Partie optionnelle. Comprend :

ar	make	strip

C language Development Utilities :

Partie optionnelle. Comprend :

c89	lex	yacc

FORTRAN Development Utilities :

Partie optionnelle. Comprend :

asa	fort77

C language bindings :

Partie optionnelle. Comprend :

- Commande d'exécution :

  system()

- Communication *pipe* avec les programmes :

  popen(), pclose()

- Comparaison d'expressions régulières :

  regcomp(), regexec(), regfree(), regerror()

- Comparaison de *filename* ou *pathname*:

  fnmatch()

- Parsing d'options de commande :

  getopt()

- Génération de pathname :

  glob(), globfree()

- Expansion de mots :

  wordexp(), wordfree()

- Accès aux variables d'environnement :

  getenv()

- Obtention de variables configurables :

  confstr(), sysconf(), pathconf(), fpathconf()

- Contrôle de *locale*

  setlocale()

POSIX 1003.2a : Overview

Les composants

Révision du langage de commande *shell*

- Substition d'*alias*
- Nouvelles variables
- Commande *set*

Révision des utilitaires 1003.2 :

kill mailx read sh wait

Nouveaux utilitaires :

alias	at	batch	bg
compress	crontab	csplit	ctags df
du	ex	expand	fc
fg	file	jobs	lint89
man	mesg	more	newgrp
nice	nm	passwd	patch

ps	renice	split	strings
tabs	talk	tput	unalias
uncompress	unexpand	uudecode	uuencode
vi	who	write	zcat

POSIX 1003.2a : Overview

Ce qui est dans VMS POSIX

Classic :

- Le *shell* ainsi que décrit au chapitre 3
- Tous les utilitaires listés au chapitre 4 avec des extensions

SDU :

- Pas d'utilitaires pour le moment

CDU :

- lex yacc

C Bindings :

- Tout ce qui est décrit en appendice B

FDU :

- asa

UPE :

- Extensions du *shell*
- Extensions de :

  -- kill read sh wait

Les utilitaires suivants :

alias	bg	compress[1]	expand[2]
fc	fg	file jobs	patch
split	strings	tabs	unalias
uncompress[1]	unexpand	uudecode	uuencode
zcat[1]			
banner	c	cal	crypt
diff3	line	mknod	nl
pack	pcat	sum	unpack

[1] Ces utilitaires sont sous copyright. Peuvent être supprimés.

[2] Peut être supprimé.

Le *shell* :

- Le standard (1003.2) est basé sur le *bourne shell*
- Toute extension du standard est permise
- UPE étend le *shell* avec :
  - -- édition de commande
  - -- substitution d'*alias*
  - -- nouvelles variables, options et commandes
- VIP implémente un KornShell complet
- Les variables suivantes sont ajoutées :

CDPATH. COLUMS, EDITOR, LINES, OLDPWD, PS3, PWD, REPLY, SHELL, TMOUT, VISUAL

- Les nouvelles commandes suivantes ont été ajoutées :

  let print times typeset whence

- Utilisation de :

  /etc/.profile

  et

  $HOME/.profile

Commande dcl :

Crée un sous-*process* ou un *process* capable d'exécuter des commandes DCL depuis le *shell*

- dcl [command] : crée un sous-*process* VMS et exécute la commande DCL

- dcl [-p command] : crée un sous-*process* VMS, positionne le "prompt" DCL et exécute la commande DCL

- dcl : crée un *process* détaché VMS et exécute le fichier de *login* de l'utilisateur

- CTRL-\ suspend l'exécution du *process* VMS

POSIX 1003.4 : Overview

Introduction

Résumé de 1003.4 :

- ***"...defines extensions to 1003.1-1988 to address realtime requirements of applications."***
- Les services de 1003.1 sont des pré-requis
- Fonctionnalités :
  - -- *Binary Semaphores*
  - -- *Process Memory Locking*
  - -- *Shared Memory*
  - -- *Priority Sheduling*
  - -- *Real time signal extension*

-- *Timers*

-- *Interprocess Communication*

-- *Synchronized I/O*

-- *Asynchronized I/O*

-- *Realtime files*

- Nouveaux travaux en cours :

  -- 1003.4a : Threads (Pthreads)

  -- 1003.4b : Support d'une interface indépendante des langages

- Les travaux de ces 2 derniers groupes de travail ne seront pas abordés ici

- Les Threads (Pthreads et CMA seront intégrés en standard dans VMS V5.5)

POSIX 1003.4 : Overview

Sémaphores

- Mécanisme de communication inter-*processes* "haute performance"

  -- permet d'implanter des "sections critiques"

  -- les *processes* doivent coopérer : un sémaphore est une entité globale qui n'est PAS associée à un *process* particulier

  -- la propriété du sémaphore n'est pas impliquée : quiconque a accès au sémaphore peut le dévérrouiller

- Chaine de caractère nommée

- 2 caractéristiques :

  -- Persistants : préservation de l'état du sémaphore après le dernier *close*. L'état initial est défini à la création

  -- Non Persistants : Il y a garantie d'être non vérrouillé lors du premier *open*

- Fonctions :

  -- sem_mksem() : creation du sémaphore en lui donnant un nom; ce sémaphore peut être persistant ou non persistant

  -- sem_unlink() : destruction du sémaphore nommé par son nom : si le sémaphore est ouvert, sa destruction est repoussée jusqu'à ce qu'il n'y ait plus d'utilisateur

  -- sem_open(), sem_close() : ouverture et fermeture du sémaphore : si le

sémaphore était persistant et qu'on le ferme, son état persiste après le *close*

-- sem_wait() : vérrouillage du sémaphore en éffectuant une "opération P" :

> si le sémaphore n'était pas verrouillé, on le vérrouille

> si le sémaphore était vérrouille, le *process* est bloqué et on ajoute le *process* à la liste des *processes* en attente de ce sémaphore

-- sem_ifwait() : vérrouillage du sémaphore si celui-ci peut être vérrouillé sans attendre qu'un autre *process* ait déverrouillé le sémaphore par une "opération V" :

> si le sémaphore n'était pas vérrouillé, le sémaphore n'est pas affecté

> si le sémaphore était vérrouillé et qu'il n'y a pas de *process* en attente du sémaphore le sémaphore est dévérrouillé

> si le sémaphore était vérrouillé et qu'il y a des *processes* en attente du sémaphore, le sémaphore est laissé verrouillé et l'un des *processes* en attente du sémaphore est débloqué

-- sem_post() : dévérrouillage du sémaphore en effectuant un "opération V"

-- sem_ifpost() : dévérrouillage du sémaphore en effectuant un "opération V" si et seulement si un *process* attend le sémaphore

- Les requètes pour un (des) sémaphore(s) sont mises en queue en fonction de la priorité du *process* demandeur pour éviter des problèmes de *deadlocks*

- Les *file descriptors* sont transmis au travers d'un *fork()*

POSIX 1003.4 : Overview

Process Memory Locking

- Garantie de résidence en mémoire permettant de réduire les temps de latence
- Possibilité de vérrouiller des régions mémoire spécifiques :
  - -- L'ensemble de l'espace virtuel du *process*
  - -- Vérrouillage d'une zone d'adresse spécifique de l'espace d'adressage du *process*
- S'il y a extension de la zone vérrouillee, cette extension est soit :
  - -- automatiquement vérrouillée (flag MCL_FUTURE)

-- non automatiquement vérrouillée (flag MCL_CURRENT)

- Modification des primitives de bas niveau de VMS pour vérrouiller la mémoire au fur et à mesure qu'elle est ajoutée à l'espace d'adressage du *process*

- Le signal SIGSEGV est envoyé en cas d'échec de vérrouillage

- Fonctions :

  -- mlockall() : vérrouillage de l'espace d'adressage d'un *process*

  -- munlockall() : dévérrouillage de l'espace d'adressage d'un *process*

  -- mlock() : dévérrouillage de la zone spécifiée

-- munlock() : dévérrouillage de la zone précédemment spécifiée

- La mémoire n'est pas vérrouillée au travers d'un *fork()* ou d'un *exec*()*
- Nécessite le privilège PSWAPM
- Implanté en utilisant $LCKPAG et UNLCKPAG

POSIX 1003.4 : Overview

Shared memory et mapped files

- Un objet mémoire peut être un fichier ou une région de la mémoire partagée qui peut être *mappee* dans l'espace d'adresse de un ou plusieurs *processes*

- Un *process* doit d'abord ouvrir un objet mémoire (fichier ou région); puis l'application *mappe* cet objet dans son espace d'adresses avec la possibilité de changer les protections mémoire même quand l'objet est *mappé*

- Fonctions :

  -- mmap() : *mappe* les adresses du *process* avec un objet mémoire avec possibilité de contrôle de la protection des pages

-- munmap() : opération inverse

-- mprotect() : changement des protections d'accès du *mapping*

-- msync() : écriture des pages modifiées : garantit l'intégrité (possibilité d'écriture synchrone, asynchrone et d'invalidation du "cache" ainsi formé)

-- shm_open() permet d'ouvrir un objet mémoire en le référençant par un *pathname*

-- shm_close() permet de supprimer un objet mémoire partagé

- Implanté en utilisant les *global sections* VMS

- unlink() ou shm_delete() d'un fichier ou d'un objet mémoire respectivement

suppriment le nom mais n'effectuent pas d'opération de type unmap()

- Il est possible de référencer 2 *name spaces* différents pour nommer des objets pouvant être *mappés* :

  -- Si l'option _POSIX_MAPPED_FILE est présente, les fichiers peuvent être accédés par open()

  -- Si l'option _POSIX_SHARED_MEMORY_OBJECTS les objets mémoire qui peuvent ne pas être des fichiers peuvent être accédés par shm_open()

  -- Ces 2 options ne sont pas mutuellement exclusives et l'on peut choisir l'une ou l'autre des implantations

- Le *mapping* des objets mémoires est conservé après un fork(); ce n'est pas le cas de exec()

POSIX 1003.4 : Overview

Priority Scheduling

- Contrôle de la séquence d'exécution de *processes* concourants afin de répondre aux besoins de temps de réponse

- 3 algorithmes de *scheduling* :

  -- FIFO : les *processes* sont ordonnés dans la queue d'attente correspondante du *scheduler* suivant une liste ordonnée par le temps pendant lequel les *processes* sont dans cette queue sans avoir été exécutés :

  > quand un *process* a été préempté, il passe en tête de queue d'attente pour sa priorité (différent de VMS)

> quand un *process* devient *computable*, il est placé en fin de queue d'attente pour sa priorité

> quand un *process* appelle la primitive sched_yield(), il est placé en fin de queue d'attente pour sa priorité

-- Round-Robin : identique à l'algorithme FIFO mais avec la notion de *time slice* :

> si un *process* a été exécuté pendant plus du temps donné par sched_get_rr_interval() le *process* passe en fin de queue pour sa priorité

-- Autre : laissé libre à l'implantation : utilisation du *scheduler* standard VMS

- 32 priorités pour l'ensemble des algorithmes
- Les priorités POSIX temps réel 0-31 correspondent aux priorités temps réel VMS 16-31 (0,1 = 16; 2,3 = 17;...)
- Fonctions :
  - -- sched_set_sched_param() : choix de la priorité du *scheduling*
  - -- sched_get_sched_param() : obtention de la priorité du *scheduling*
  - -- sched_setscheduler() : choix de l'algorithme de *scheduling* et de la priorité
  - -- sched_getscheduler() : obtention de l'algorithme de *scheduling* et de la priorité

-- sched_yield() : céder l'exécution à un autre *process*

-- sched_get_priority_max() : obtention de la priorité maximale

-- sched_get_priority_min() : obtention de la priorité minimale

-- sched_get_rr_interval() : obtention de la valeur du *round-robin interval*

- Le *round-robin interval* de POSIX est identique au QUANTUM VMS (200 ms par défaut). La valeur est ajustée si besoin en changeant le paramètre SYSGEN

- Le privilège ALTPRI est nécessaire pour positionner la priorité d'un *process* au dessus de sa priorité initiale : un changement d'algorithme peut suffire à changer cette priorité

POSIX 1003.4 : Overview

Realtime Signals Extension

- Le mécanisme de *signals* de 1003.1 est insuffisant :
  - -- Les *signals* peuvent être perdus
  - -- L'ordre de délivrance de *signals* est indéfini
  - -- Il n'y a pas de différentiation entre plusieurs *signals* du même type
- Les mécanismes suivants permettent la délivrance d'évènements :
  - -- Complétude d'I/O asynchrones
  - -- Expiration de *timer*
  - -- Passage de messages IPC

-- Génération d'évènements définis par l'utilisateur

- Noms de signaux : le *header* <signal.h> est étendu pour permettre :

  -- La déclaration de la structure *sigevent* qui permet d'étendre les numéros et les noms des *signals*

  -- Les définitions des constantes SIGRTMIN et SIGRTMAX indiquant quel est le domaine des *signals* dits temps réel

- Si le flag SA_QUEUED est précisé dans la structure *sigaction*

  -- Toute occurence suivante de ce *signal*, pendant qu'il est en attente conduira à sa mise dans une queue

-- Des occurences multiples du même *signal* sont mises en queue dans un ordre de type FIFO

-- Quand plusieurs *signals* sont en attente, les *signals* ayant la valeur numérique la plus basse sont délivrés d'abord

- L'action par défaut pour les signaux "temps réel" est de terminer la vie du *process* mais il est possible d'établir un *condition handler*

- 2 nouvelles primitives :

  -- sigpoll() : permet d'effectuer un *polling* des evènements qui sont mis en queue en direction du *process* courant

  -- sigsend() : permet de générer un *signal* défini par l'application

POSIX 1003.4 : Overview

Clock et Timers

- Le comportement des *processes* temps réel dépend du fait que leurs contraintes de temps sont satisfaites
- Les fonctions liées aux *timers* en 1003.1 manquent de fonctionnalités et de précision pour supporter les nécessités temps réel
- 1003.4 permet la spécification de temps par unité de nanosecondes
- La notification de *timer* est effectuée au moyen de *signals* ou de *realtime signals*

- Fonctions *timer* système :

  -- clock_settime()/clock_gettime() : obtention/ positionnement d'une horloge du système

  -- clock_settimedrift()/ clock_gettimedrift() : positionne ou obtient la dérive d'une horloge du système

  > Permet l'écriture d'applications permettant la synchronisation d'horloges dans un environnement distribué

  > Permet l'ajustement du temps système sans avoir à retarder l'horloge du système

  -- clock_getres() : obtention de la résolution d'une horloge du système

-- Les fonctions de positionnement d'horloges décrites précédemment n'affectent pas les *timers* des *processes* déjà "armés"

- Fonctions *timer process* :

  -- timer_create() : allocation d'un *timer process* pour une horloge spécifiée. Retourne un *timer_id*

  -- timer_delete () : supprime un *timer process*

  -- timer_settime() : armement d'un *timer process* absolu ou relatif

  -- timer _gettime() : obtention de la valeur d'un *timer process*

  -- timer_getoverrun() : retourne le nombre de fois ou le *timer* ayant expiré a vu la génération du *signal*

correspondant retardé (*process* non schedulé par exemple)

-- timer_getres() : obtention de la résolution d'un *timer process*

-- nanosleep() : hibernation haute résolution

-- nanosleep_getres() : obtention de la résolution supportée par nanosleep()

POSIX 1003.4 : Overview

IPC

- Permet de passer une quantité arbitraire de données entre les *processes* concernés
- Défini comme un processus de communication avec des queues de message définies au niveau du système
- Accédé à travers un nom qui peut être nommé par un *pathname* (ce n'est pas obligatoire)
- Supporte plusieurs *processes* émetteurs et récepteurs
- Admet un mode synchrone et un mode asynchrone

- On utilise le mécanisme d'*events* pour prévenir l'utilisateur de la complétude des appels asynchrones IPC

- Fonctions :

  -- mqopen(), mqclose() : permet d'établir la correspondance entre une queue de messages et son descripteur. Autorise des flags de positionnement d'attributs de queues (O_RDONLY, O_WRONLY,...)

  -- mqsend() : envoi d'un message dans la queue (synchrone, asynchrone)

  -- mqreceive() : reception d'un message depuis la queue (synchrone, asynchrone)

  -- mqsetattr() : paramètrage des attributs de la queue (nombre de messages, nombre de bytes, *timestamp*,...)

-- mqgetattr() : obtention des attributs de la queue

-- mqgetpid() : obtention du *pid* d'un *process* émetteur

-- msgalloc()/msgfree() : allocation/ libération d'un buffer de données message

- IPC est semblable aux *boites-aux-lettres* VMS avec les fonctionnalités additionnelles suivantes :

  -- contrôle sur les attributs des queues de messages

  -- possibilité de donner une priorité aux messages

  -- pré-allocation de ressources

  -- identification de messages

-- *time-stamping* automatique du message

POSIX 1003.4 : Overview

Entrées/Sorties synchrones

- Mécanisme permettant à une application d'assurer l'intégrité de ses données et de ses fichiers
- La sortie synchronisée (*write*) permet d'assurer que les données écrites vers un périphérique sont réellement reçues par le périphérique
- L'entrée synchronisée (*read*) permet d'assurer que les données luees depuis un périphérique sont réellement présentes dans le périphérique
- Fonctions :

  -- open(), fcntl(), read(), write()

> flag O_DSYNC : intégrité des données

> flag O_FDSYNC : intégrité des données et des paramètres *file system* du fichier

> Ces flags sont ajoutés dans le fichier de *header* <sio.h>

-- fsync()/fdatasync() : permet le transfert de façon synchrone des données associées avec le fichier

- Les attributs qui ne sont pas obligatoires pour retrouver les données (taille, date de dernier accès,...) n'ont pas besoin d'être mis à jour avant de retourner au *process* appelant

POSIX 1003.4 : Overview

Entrées/Sorties synchrones

- Permet à un seul *process* d'effectuer des I/Os de façon asynchrone vers un ou plusieurs fichiers
- Notification de fin d'I/O à l'aide d*e signals* ou de *realtime signals*
- Un *asynchronous I/O Control Block* est créé pour décrire l'état de l'I/O
- Fonctions :
  - -- aio_read() : lecture asynchrone
  - -- aio_write() : écriture asynchrone
  - -- lio_listio() : liste des requêtes d'I/Os

-- aio_cancel() : arrêt d'une requête d'I/O asynchrone

-- aio_suspend() : attente de fin de requête d'I/O asynchrone

-- aio_error() : retourne le *status* d'erreur associé

-- aio_return() : retourne le *status* de retour associé

-- aio_fsync() : force toutes les I/Os en queue à passer en mode synchrone (garantie d'intégrité des données)

POSIX 1003.4 : Overview

Fichiers temps réels

- Permet à une application d'obtenir des informations au sujet de l'utilisation d'un fichier ou d'un *file system* et d'influencer le comportement de ce fichier ou ce *file system*

- Attributs et possibilités :

  -- contrôle d'allocation et d'extension de fichiers

  -- contrôle du type de *caching* pour un fichier

  -- type d'accès (séquentiel, aléatoire,...)

  -- *Direct I/O* (non supporté sous VMS)

  -- Transferts alignés

- Fonctions :

  -- rfcreat() : création d'un fichier temps réel

  -- rfgetattr()/rfsetattr() : obtention/ positionnement des attributs d'un fichier temps réel

  -- rfgetcap() : obtention des possibilités d'un fichier temps réel

  -- rfgetincr() : obtention de la liste incrémentale des possibilités d'un fichier temps réel :

  > incrément de pré-allocation

  > incrément d'extension

  > incrément de taille de transfert

  > incrément de taille de transfert "atomique"

-- aread(), awrite() : lecture, écriture asynchrone depuis un fichier temps réel

-- placebuf() : obtention d'un buffer convenablement aligné

-- read(), write()

- Les attributs temps réels d'un fichier n'ont pas besoin d'être persistants (il le sont sous VMS POSIX)

Les extensions nécessaires de VMS pour supporter POSIX

Hooks **dans l'*exec*** :

- EVENT_FLAGS_AND_ASTS.EXE pour supporter les *signals* POSIX
- IO_ROUTINE.EXE pour supporter un $ASSIGN *session-wide*
- PAGE_MANAGEMENT.EXE pour supporter les nécessités de vérrouillage mémoire 1003.4
- PROCESS_MANAGEMENT.EXE pour le support de fork(), de la notion de *session* POSIX et les mécanismes de *scheduling* 1003.4
- WORKING_SET_MANAGEMENT.EXE pour le support de fork() et le support

des nécessités de vérrouillage mémoire 1003.4

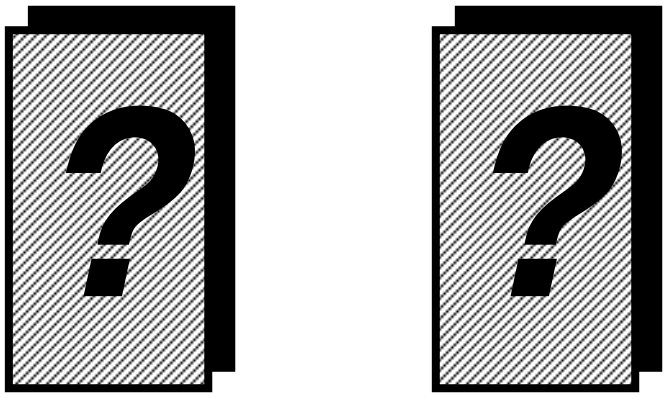
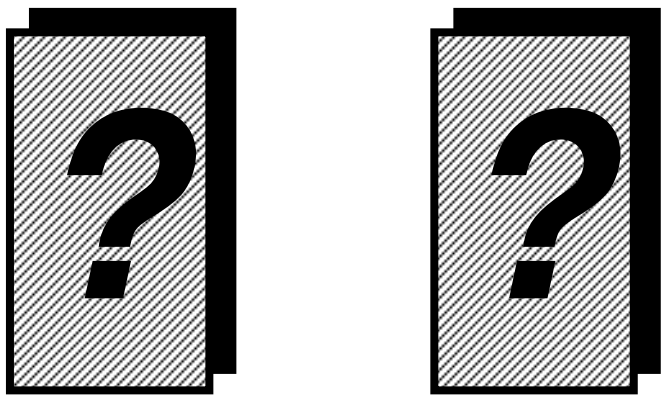
- SYSTEM_PRIMITIVES.EXE : support de l'*accounting* POSIX

Extensions de VMS :

- *Hooks* dans TTDRIVER pour supporter le *terminal driver* POSIX
- Extension du *lock manager* pour supporter la notion de *byte range locking*
- Nouveau *mailbox* pour supporter les FIFOs POSIX

Le mode mixte

Est-il possible de mixer les appels aux services VMS et aux services POSIX dans la même application

- Il n'est possible d'appeler les primitives POSIX que depuis le langage C
- Il n'y a pas d'interopérabilité possible avec VAXCRTL (problèmes de *signal*, I/O)
- Dans une application mono-*process* (n'utilisant pas fork() ou exec()), il est possible de mixer les appels aux services système VMS et aux services POSIX
- fork() échoue si on utilise RMS, les CEF et les *writable sections* privées
- Le process fils dans fork() n'hérite pas des *locks* du *process* père

Questions