Programmation système sous UNIX et multithreading

Thierry Monteil[1]

septembre 2018

[1]e-mail monteil@laas.fr ou monteil@insa-toulouse.fr

Contents

Chapitre 1

Introduction aux Threads

1.1 Introduction

- les threads sont une séquence indépendante d'exécution de code à l'intérieur d'un processus
- il peut y avoir plusieurs threads dans un processus
- les threads d'un même processus partagent l'espace mémoire du processus
- la programmation multi-thread est plus complexe

- accès aux même données en lecture écriture
- gestion de l'exécution parallèle
- problème de synchronisation

- bénéfices:

- les attentes sur différentes E/S peuvent se faire en parallèle (serveur)
- utilisation des multi-processeurs
- gestion non bloquante d'opérations: interface graphique
- gestion de la rapidité de réponse d'un serveur en fonction des priorités des requêtes
- éviter les dead lock dans un serveur
- recouvrement travail et E/S

- pas forcement adapté à tous les types d'application
- différentes bibliothèques pour les threads
- dans ce cours on utilisera uniquement la bibliothèque normalisée POSIX
- chaque constructeur offre une bibliothèque POSIX et quelquefois une bibliothèque personnalisée pour les threads
- inclusion du fichier **pthread.h**
- la compilation se fait avec des options spéciales:

- certains systèmes: précision de l'utilisation de bibliothèque réentrante: -D _REENTRANT et utilisation de la bibliothèque: -lpthread
- en linux 2.6 avec la NPTL (Native Posix Thread Library) -pthread réalise la réentrance et l'appel à la bonne bibliothèque

1.2 Gestion des erreurs

- Généralement un appel système réussi renvoie 0, une erreur renvoie -1
- une gestion des erreurs spéciale est réalisée pour les appels systèmes
- une variable nommée **errno** donne le numéro de l'erreur
- il faut la déclarer ainsi: **extern int errno**
- une fonction permet d'avoir un affichage de la *dernière erreur* apparue: **perror()**

1.3 Création

- création avec la fonction **int pthread_create(pthread_t *thread_id, const pthread_attr_t *attr, void * (*start_fonction)(void*), void *arg)**
- le champ thread_id est l'identificateur du thread (même rôle que l'identificateur du processus pid). Ce peut être un entier ou une structure cachée. Pour comparer deux TID, il vaut mieux utiliser la commande: int pthread_equal(pthread_t thread_id1,pthread_t thread_id2) qui renvoit une valeur non nulle en cas d'égalité
- attr est un ensemble d'attributs permettant de modifier les propriétés du thread
  - si attr est à NULL: attribut par défaut
  - permet de modifier la taille de la pile du thread (par exemple: 1 Mo par défaut)
  - possibilité de choisir une politique de scheduling pour le thread
  - gestion d'une priorité
  - etc
- le troisième champ est le nom d'une fonction à exécuter
- le dernier champ est un argument pour la fonction, s'il y a plusieurs arguments il faut faire une structure pour n'avoir qu'un point d'entrée sur les arguments
- la fonction retourne 0 si tout c'est bien passé, en cas d'erreur elle ne gère pas nécessairement la variable de gestion d'erreur errno

1.3.1 Attributs

- Création des attributs via int pthread_attr_init(pthread_attr_t * attribus);
- Destruction après création du thread: int pthread_attr_destroy(pthread_attr_t *attributs);
- manipulation avant création du thread pas de dynamicité
- fonction set et get sur chaque attribut modifiable
- exemple:
  - int pthread_attr_setdetachstate(pthread_attr_t *attributs, int valeur); avec comme valeur PTHREAD_CREATE_JOINABLE ou PTHREAD_CREATE_DETACHED
  - int pthread_attr_setstacksize(pthread_attr_t *attributs, isize_t valeur), couplé avec les fonctions de modification de l'adresse de la pile
  - etc

1.4 Destruction

- l'arrêt d'un thread peut se faire de différentes façons

1. arrêt de soi même
2. l'arrêt peut être synchronisé avec l'arrêt d'autres threads

3. un thread peut demander l'arrêt d'autres threads

⇒ 1 - deux possibilités:
la fonction **void pthread_exit(void * status)** arrête le thread et détruit ses ressources, le champ status peut être utlisé par d'autres threads (attention à son allocation)
si on quitte la fonction appelée par pthread_create le résultat est le même

⇒ 2 - un thread peut se mettre en attente de l'arrêt d'un autre
int pthread_join(pthread_t idthread, void **statusp)
le thread utilisant cette fonction attend le thread d'identité idthread
le champ statusp prend la valeur que le thread attendu a passé à pthread_exit si tout se passe bien sinon si le thread n'a pu renvoyer une valeur, *statusp prend la valeur PTHREAD_CANCEL
ceci peut échouer si le thread est déjà terminé ou bien s'il y a un risque d'interblocage (join sur soi-même)
attention au problème de persistance: exemple: entier, adresse entier, pointeur
impossibilité d'attendre un thread quelconque

⇒ 3 - un thread peut demander l'arrêt d'un autre thread (**int pthread_cancel(pthread_t idthread)**)
l'arrêt du thread se fera à un point d'arrêt ne posant pas de problème

Si un thread ne renvoit pas de valeur intéressante et s'il n'a pas besoin d'être attendu on peut utiliser pthread_detach (pthread_t tid) qui permet de libérer sans attendre les ressources du thread quand il se termine.

1.5 Mise en oeuvre

1.5.1 Ressources

- décomposition d'un processus en 2 composants:

- un ensemble de threads
- un ensemble de ressources

- le thread exécute une suite d'instructions
- les ressource sont:

- données
- fichiers ouverts
- etc

- ressources partagées par tous les threads
- chaque thread possède:

- compteur d'instructions
- une pile
- des registres

- attention à l'accès concurrents sur les données ⇒ synchronisation

- existence de thread système et thread utilisateur
- thread système font partie du noyau
  - un thread système est associé à un processus utilisateur
  - il est détruit et créé à la demande dans le système d'exploitation
  - utilisé pour les E/S asynchrones
  - ils utilisent comme ressource uniquement la pile système et une zone de sauvegarde des registres

- thread utilisateur manipulés par l'utilisateur
- deux niveaux de visibilité pour l'utilisateur:

- niveau système et processeur: processus légers
  - en anglais Lightweight process (LWP) utilisation du terme de processeur virtuel
  - Ils sont au dessus des threads système
  - chaque processus peut avoir un ou plusieurs LWP
  - chaque LWP est relié à un thread système
  - le scheduling de chaque LWP est indépendant
  - le LWP est vu par le noyau à travers le thread système
  - les LWP d'un même processus partage les ressources du processus

- niveau de l'utilisateur: les threads
  - notion de thread uniquement manipulée par l'utilisateur
  - le noyau ne connait pas les threads utilisateur
  - utilisation pour cela d'une bibliothèque pour les threads
  - thread utilisateur n'utilisent pas le noyau et des appels système pour leur gestion
    ⇒ pas d'appel système
    ⇒ gain en performance
  - association de un ou plusieurs threads avec un LWP
  - notion de thread lié et multiplex'e
  - si thread dans un appel système, le thread reste lié au LWP: famine ??
  - performance du multi-threading

1.6 Synchronisation

- la synchronisation peut se faire par des sémapores IPC
  ⇒ existence d'autres mécanismes pour les threads
  plus rapide
  plus complet

1.6.1 mutex

- Cet outil de synchronisation sert à régler les problèmes de mutuelle exclusion
- principe:

- pour utiliser la ressource verrouillage du mutex si déjà verrouiller le thread se met en attente
- fin d'utilisation de la ressource: déverrouillage du mutex

- le type est **pthread_mutex_t**
- s'applique à toute ressource où il ne faut qu'un consommateur à la fois
- le mutex peut être partagé entre les threads d'un même processus configuration de type USYNC_THREAD
- le mutex peut aussi être partagé entre processus lourds avec l'attribut USYNC_PROCESS

Remarque:
sous linux, on ne peut pas partager des mutex entre processus
- le mutex peut se trouver dans un état verrouillé ou pas: LOCKED ou UNLOCKED
- l'initialisation du mutex peut se faire par défaut ou en passant par ses attributs de type **pthread_mutexattr_t** et en utilisant des fonctions d'initialisations
- initialisation par défaut: USYNC_THREAD et UNLOCKED

Exemple:
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
- initialisation avec les fonctions de manipulation des attributs : pthread_mutexattr_t att_mutex;

```
pthread_mutexattr_init(&att_mutex);
pthread_mutex_init(&lock, &att_mutex);
pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- il existe des fonctions pour manipuler pthread_mutexattr, elles semblent cependant très dépendantes du système (voir par exemple linux et solaris)

- deux opérations sont possibles: blocage / déblocage:

- la demande d'entrée en section critique se fait avec
  int pthread_mutex_lock(pthread_mutex_t *lock), elle est bloquante
- la libération du mutex se fait avec
  int pthread_mutex_unlock(pthread_mutex_t *lock)
- une fonction permet de tester le mutex et de sortir de la fonction si le mutex est déjà bloqué
  int pthread_mutex_trylock(pthread_mutex_t *lock)

- la destruction du mutex se fait avec
int pthread_mutex_destroy(pthread_mutex_t *lock)

Remarque:
- Dans le cas ou plusieurs threads sont en attente du mutex et si le thread détenteur du mutex le déverrouille, la bilbiothèque doit choisir un thread
- les thread liés ont la préférence sur les threads multiplexés
- on peut modifier le type de mutex via ses attributs, par exemple permettre à un même thread de faire plusieurs lock sur le même mutex sans se bloquer

- exemple d'utilisation

1.6.2 variable condition

- permet de faire attendre un thread jusqu'à ce qu'une condition soit vérifiée
- un ou plusieurs thread peuvent attendre une condition
- les threads peuvent être dans le même processus ou dans 2 processus indépendants
- quand la condition est signalée un ou tous les threads en attente peuvent continuer
- la variable condition est sans mémoire, si un nouveau thread se met en attente d'une condition il est bloqué
- le type est **pthread_cond_t**
- initialisation à la déclaration avec : PTHREAD_COND_INITIALIZER

- l'initialisation peut se faire avec des fonctions pour spécialiser la variable condition

```
pthread_cond_t cond;
pthread_condattr_t att_cond;
pthread_condattr_init(&att_cond);
pthread_cond_init(&cond, &att_cond);
```

- la destruction de la variable condition se fait avec la fonction **int pthread_cond_destroy (pthread_cond_t *cond)**
- les deux opérations sont :

- attente de la condition :
  int pthread_cond_wait
  (pthread_cond_t *cond, pthread_mutex_t *mutex)
  - déblocage du mutex avant le blocage du thread appelant
  - blocage du mutex quand la condition est atteinte et avant de sortir
- la signalisation de la condition se fait de 2 manières:
  - **int pthread_cond_signal (pthread_cond_t *cond)** débloque un des threads bloqués au moins
  - **int pthread_cond_broadcast (pthread_cond_t *cond)** débloque tous les threads

- exemple de gestion d'un buffer circulaire
- question:
blocage du programme commme dans le cas précédent ?
retrait d'un element entre le test et le blocage sur la condition trop plein ?

1.7 Ordonnancement

- rappel sur le modèle d'accès au processeur (schéma)
- la norme POSIX est moins fournie dans ce domaine: beaucoup de variation dans les bibliothèques de thread propriétaire
- la politique de "scheduling" est affectée aux attributs du thread par la fonction **pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)**
- 3 politiques par défaut:

- SCHED_FIFO premier arrivée, premier servi
- SCHED_RR politique du tourniquet
- SCHED_OTHER varie selon le constructeur

- le lien avec le système d'exploitation se fait avec le champ de portée
- la manipulation se fait avec **pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)**

- PTHREAD_SCOPE_PROCESS: le thread n'est pas attaché au système d'exploitation
- PTHREAD_SCOPE_SYSTEM: le thread est vu par le système d'exploitation

- une gestion des priorités est aussi possible

1.8 fonctions spéciales

1.8.1 identification

- la fonction **pthread_t pthread_self()** retourne l'identificateur du thread qui l'appelle.
- c'est l'équivalent du pid pour les processus. on parle de tid (thread identity)

1.8.2 Fonctions d'initialisation

- on peut vouloir qu'une fonction ne soit exécutée qu'une fois même si plusieurs threads l'appellent.
- par exemple, une fonction d'allocation d'une structure globale
- tous les threads appellent cette fonctions
- la librairie garantie qu'un seul aura le droit de l'exécuter
- utilisation de la structure **pthread_once_t**
- déclaration et initialisation:

```
pthread_once_t once=PTHREAD_ONCE_INIT;
```

- la fonction d'initialisation est une fonction sans paramètre et ne renvoyant rien: void fonct(void)
- l'appel dans chaque thread se fait avac la fonction **pthread_once(pthread_once_t *once, void (*fonct)(void))**

Exemple:

```
pthread_once(&once,fonct);
```

- exemple

1.8.3 Manipulation de données spécifique

- on peut fournir aux threads un moyen de travailler sur des données différentes en utilisant la même variable dans chaque thread.
- Ceci peut être utilisé par exemple lors de la transformation d'un programme séquentiel existant en programme multi-thread
- la manipulation des variables se fait à l'aide d'une clé
- la différence entre les variables se fait à l'aide de la clé et le thread lui-même
- la clé est accessible par tous les threads
- la clé doit être initialisée une seule fois :
  - on peut utiliser les fonctions précédentes "once"
  - on peut faire l'initialisation dans le "main" avant la création des threads
- le type de la clé est: **pthread_key_t**
- l'initalisation se fait avec la fonction: **pthread_key_create(pthread_key_t *key, void (*destructor)(void*))**
- le premier paramètre est un pointeur sur la clé, le second est optionnel et donne le moyen de détruire la zone mémoire associée à la variable qui sera reliée à la clé
- il existe ensuite 2 fonctions :
  - une pour reliè une clé à une zone mémoire pour un thread: **int pthread_setspecific(pthread_key_t key, const void *value)**
  - une pour récupérer à partir d'une clé la zone mémoire de ce thread: **void* pthread_getspecific(pthread_key_t key)**

- exemple

1.9 Points d'arrêt

1.9.1 Principe

- les points d'arrêt sont utilisés lors d'un **pthread_cancel**
- l'appel à cette fonction n'arrête pas immédiatement le thread concerné
=> attente d'atteindre le prochain point d'arrêt
=> éviter la création d'incohérence vis à vis des autres threads

Exemple:

arrêter un malloc en cours d'exécution => manipulation de structure pour donner les zones libres et occupées de la mémoire
- la pluspart des fonctions du système sont des points d'arrêt potentiels
- toutes les fonctions bloquantes à temps non borné sont des points d'arrêt (read, wait, etc)
- problème dans certains cas, on a besoin de réaliser une dernière action avant de terminer un thread

Exemple:

désallocation de la mémoire dynamique, gestion des mutex (voir TD)
- mise en place de handler de nettoyage
- stockage des handlers dans une pile (gestion LIFO)
- empilage d'une fonction: **void pthread_cleanup_push(void (*cleanup_handler)(void*), void* arg)**
- premier paramètre: le nom du handler
- le handler peut avoir un argument de type pointeur sur rien
- l'exécution de la fonction se fait par dépilage et appel
- le désempilage se fait automatiquement si une demande d'arrêt par pthread_cancel a été faite et si le thread a atteind un point d'arrêt.
- dépilage manuel d'une fonction **void pthread_cleanup_pop(int execute)**
- si le paramètre vaut 0 la fonction est désempilée sans être exécutée, si le paramètre est différent de 0 la fonction est désempilée et exécutée
- l'appel au "push" et au "pop" doit se faire dans le même niveau de code { ...}
=> ceci ressemble un peu aux exceptions

1.9.2 Fonctions annexes

- vérification si une demande de cancel n'est pas en attente et traitement de cette dernière: **void pthread_testcancel(void)**
- modification du comportement vis à vis du cancel: arrêt immédiat (PTHREAD_CANCEL_ASYNCHRONOUS) ou arrêt au prochain point d'arrêt (PTHREAD_CANCEL_DEFERRED)
- utilisation de la fonction **int pthread_setcanceltype(int type, int* oldtype)**
- premier paramètre le type d'arrêt, le deuxième paramètre permet de récupèrer l'ancien comportement
- prise en compte du cancel ou pas, ceci se fait avec le flag PTHREAD_CANCEL_ENABLE et PTHREAD_CANCEL_DISABLE
- utilisation de la fonction **int pthread_setcancelstate(int state, int* oldstate)**

Chapitre 2

Introduction programmation système

2.1 Généralités

- la programmation système est l'utilisation par l'utilisateur de fonctions du noyau ou appels systèmes
- l'appel est très facile à partir du C (le noyau étant en partie écrit en C)
- toutes les fonctions, options possibles ne seront pas décrites
- but de ce cours: donnez une idée des possibilités offertes
- pour tout savoir: 3615 man

2.2 Plan

Le plan est le suivant:

- les signaux
- les entrées sorties
- les processus
- les tubes
- la communication inter processus avec les IPC

2.3 Bibliographie

S. Hernando: "UNIX et la programmation système"
J.M. Riflet: "La programmation sous UNIX"
G. Authié et al "Parallélisme et applications irrégulières", "chapitre 10 par P.Guyaux et T. Monteil: UNIX: Communication et Parallélisme"
commande man sous UNIX
une partie des exemples sont accessibles à l'adresse WEB:
http://www.laas.fr/~monteil/INSA/cours.html

Chapitre 3

Signaux

3.1 Principe

- la description et les fonctions de ce paragraphe suivent la norme POSIX
- Signaux permettent de réaliser une synchronistion entre processus
- on parle aussi de synchronistion par événement
- on ne s'intéressera pas aux signaux temps réels
- un processus peut:

  - envoyer un signal à un autre processus
  - attendre l'arrivée d'un signal

- un traitement spécial (**handler**) est effectué à la réception d'un signal
- une mémoire est associée à ce mécanisme
  ⇒ mémorise un événement (pas vrai pour les signaux temps réels
  ⇒ justification sur un exemple de synchronisation

Définition:
un signal pendant est un signal envoyé par un processus mais non encore pris en compte par le processus récepteur
Définition:
un signal délivré est un signal pris en compte par le processus récepteur

3.2 Mise en oeuvre

3.2.1 Signaux

- les signaux et fonctions de manipulations sont accessible en C dès que le fichier **signal.h** est inclus

  - signaux numérotés de 1 à 31 (32 à XX pour le temps réel)
- signaux propres à chaque processus et mémorisés dans le PCB du processus
- émission d'un signal vers un autre processus ne peut se faire que si l'émetteur appartient à un utilisateur ayant le même numéro d'utilisateur réel ou effectif
- quelques signaux (unix-linux) parmi les plus courants:

  - SIGKILL (9) : Le processus est tué à la réception de ce signal. Contrairement à la plupart des signaux, il ne peut ni être ignoré ni capturé par le processus

  - SIGBUS et SIGSEGV: les bêtes noires du programmaeur
  - SIGPIPE (13) : correpond à un problème sur un pipe de communication (voir chapitre suivant)
  - SIGALRM (14): utilisé pour mettre en place un timer
  - SUGHUP : utilisé par les démons pour se réinitialiser
  - SIGINT: le ctrl-C
  - SIGTERM : demande de terminaison "gentille"
  - SIGCHLD (17): mort d'un fils du processus recevant ce signal
  - SIGUSR1 (10), SIGUSR2 (12): signaux laissés à la libre utilisation du programmeur

- Comportement en général par défaut lors de la réception d'un signal: **mort du processus**
- envoie d'un signal:

  - en SHELL: **kill -N pid** où N est le numéro du signal et pid le numéro du processus
  - en C: **int kill (int pid, int sig)**, où sig est le numéro du signal et pid le numéro du processus, la fonction renvoie 0 si l'exécution est correcte -1 sinon

3.2.2 Masques

- chaque processus contient un masque décrivant les signaux qui sont ou ne sont pas pris en compte par le processus
- le masque par défaut prend en compte tous les signaux
- un masque est peut être implémenté par un ensemble de bits, un pour chaque signal (32 bits)
- le type masque : **sigset_t**
- des fonctions permettent de manipuler les masques (renvoient 0 si tout se passe bien):

  - **int sigemptyset(sigset_t *ens_signal)** /* ens_signal = 0 */
  - **int sigfillset(sigset_t *ens_signal)** /* ens_signal = 1 bloquera tous les signaux avec SET_MASK*/
  - **int sigaddset(sigset_t *ens_signal, int sig)** /* ens_signal = ens_signal + sig */
  - **int sigdelset(sigset_t *ens_signal, int sig)** /* ens_signal = ens_signal - sig laissera passer le signal */

  - la prise en compte du masque des **signaux non capturés** par le processus se fait avec la fonction:
    int sigprocmask(SIG_SETMASK,sigset_t *ens_signal, NULL)
(voir man pour les options)

  - on peut aussi récupérer les signaux pendants avec:
    int sigpending(sigset_t * ens_signal)

3.2.3 Capture

- la réception du signal et son traitement interrompt le programme pour exécuter le handler associé
  ⇒ attention à ne pas modifier des variables en cours d'utilisation dans le programme ⇒ risque d'incohérence, utilisation de fonction: ASYNC-SIGNAL-SAFE - pas de malloc, pas de free

- un processus n'est pas interruptible par un signal quand il est en mode noyau
- un processus n'est pas interromptible par un autre signal dans un handler (sauf signaux particuliers)
- le format du handler est: **void handler(int sig)**
- l'attachement du handler au signal se fait avec la structure **struct sigaction** et la fonction **sigaction**.
Pour un comportement par défaut:
Exemple:

```
action.sa_handler = handler;
action.sa_flags=0;
sigaction(SIGUSR1, &action, NULL)
```

Remarque:

- mise en place plus simple mais pas standard avec : signal(SIGUSR1,handler);

3.2.4 Attente et alarme

- attente sur des signaux avec l'instruction **int pause(void)**
- inclusion du fichier **unistd.h**
- impossibilité de se mettre en attente sur un signal particulier
- autre fonction **int sigsuspend(const sigset_t *ens_signal)**:
  - fonction atomique
  - positionne le masque ens_signal
  - mise en sommeil jusqu'l'arrivée d'un signal non masqué

- mise en place d'une alarme avec **unsigned int alarm(unsigned int secondes)**
  - réception du signal SIGALRM au bout du nombre de secondes spécifiés
  - le temps est le temps réel (montre)
  - alarm(0) annule le temporisateur

3.3 Exemple

- exemple d'un producteur-consommateur synchroniser par signaux

producteur:

```
mise en place du masque et handler (SIGUSR2: consommé)
pour toujours
        produire
        signaler(produit) /* kill(consommateur, SIGUSR1) */
        attendre (consommé) /* pause() */
fin pour
```

consommateur:

```
mise en place du masque et handler (SIGUSR1: produit)
pour toujours
        attendre(produit) /* pause() */
        consommer //                signaler (consommé) /* kill(producteur, SIGUSR2) */
fin pour
```

Remarque:

Cet algorithme peut poser problème, pourquoi ?

3.4 Retour sur la signalisation dans les threads

- chaque thread a son propre masque de signalisation
- il hérite du masque du thread qui l'a créé
- la modification du masque se fait avec la fonction **int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset)**
- how peut prendre 3 valeurs:
  - SIG_BLOCK: ajout au masque actuel
  - SIG_UNBLOCK: retrait du masque actuel
  - SIG_SETMASK: remplace lemasque actuel

- la création et la manipulation du masque se fait comme pour les signaux unix
- possibilité de s'envoyer des signaux entre thread dans un même processus: **int pthread_kill(pthread_t thread, int signal)**
=> impossible entre threads de 2 processus différents
- lors de la réception d'un signal par un processus, le signal est envoyé à un et un seul thread capable de traiter ce signal
- en général, afin de faciliter la gestion des signaux, un thread unique aura le droit de gérer les signaux
- L'implémentation des signaux sous linux ne suit pas la norme avant le noyau 2.6 , ceci était du au fait qu'un thread sous linux c'était en fait un processus
- il faut donc envoyer le signal au bon thread-processus directement avec son pid et non pas au pid du processus global

Chapitre 4

Fichiers bas niveau

4.1 Introduction

- manipulation au niveau bas des fichiers
- utilisation de descripteurs de fichiers: index dans la table
- type entier
- un appel système par appel de fonction:
  ⇒ peu performant si beaucoup d'appel pour des lectures écritures de petites tailles
  ⇒ choix d'une librairie de plus haut niveau avec bufferisation des données pour éviter l'appel système

4.2 Manipulation à partir des i-nodes

- utilisation d'une structure stat contenu dans les fichiers "sys/types.h" et "sys/stat.h"

```
struct stat {
        dev_t st_dev; /* disque logique du fichier */
        ino_t st_ino; /* numéro du inode */
        mode_t st_mode; /* type fichier et droit utilisateur */
        nlink_t st_nlink; /* nombre de liens physiques */
        uid_t st_uid; /* propriètaire du fichier */
        gid_t st_gid; /* groupe */
        dev_t st_rdev; /* type du device si c'est un i-node */
        off_t st_size; /* taille totale en octet */
        unsigned long st_blksize; /* taille du bloc pour les systèmes de fichiers */
        unsigned long st_blocks; /* nombre de blocs */
        time_t st_atime; /* date dernière accès */
        time_t st_mtime; /* date dernière modification */
        time_t st_ctime; /*date dernière modification du noeud */
};
```

- chargement d'une variable du type stat avec l'une de ces 2 fonctions:

  - **int stat(const char *ref, struct stat *ptrstat)**
  - **int fstat(const int desc, struct stat *ptrstat)**

- pour la première chemin du fichier, pour la seconde un descripteur de fichier

Exemple:


```
struct stat bufstat;
if(stat("/etc/passwd", &bufstat) == -1)
        printf("erreur stat ");
```

- permet par exemple de savoir si un fichier existe, à qui il appartient, ...

4.3 Primitives de manipulation

4.3.1 open

- permet l'ouverture d'un fichier
- allocation d'une nouvelle entrée dans la table des fichiers
- charge les i-nodes correspondant
- utilisation de l'include: fcntl.h en plus des 2 précédents
- **int open(const char *ref, int mode, ...)**
  - ref correspond au chemin
  - mode donne le mode d'ouverture
- une et une seule de ces 3 constantes doit être utilisée:

  - O_RDONLY : ouverture en lecture seule
  - O_WRONLY : ouverture en écriture seule
  - O_RDWR : ouverture en lecture et écriture

- personnalisation avec d'autres constantes:

  - O_APPEND : écriture en fin de fichier
  - O_SYNC : blocage tant que l'écriture n'est pas réalisée physiquement sur le disque: force l'écriture
  - ...

4.3.2 close

- fermeture du fichier
- **int close(int desc)**

4.3.3 read

- lecture à partir d'un descripteur de fichier
- **ssize_t read(int desc,void* ptr,size_t nboctet)**
- lecture d'au plus nboctet octets
- rangement à l'adresse ptr
- la fonction renvoie -1 s'il y a un problème de paramètres
- renvoie le nombre d'octets lus, 0 si on est déjà à la fin du fichier

4.3.4 write

- écriture à partir d'un descripteur
- **ssize_t write(int desc, void *ptr,size_t nboctet)**
- écriture de nboctet octets se trouvant à l'adresse ptr dans le fichier pointé par desc
- si une valeur différente de nboctet est renvoyée: problème
    - si positif: disque plein
    - si négatif: erreurs diverses

4.4 Autres fonctions

- création de lien: **link(...)**
- destruction de lien: **unlink(...)**
- changement des droits d'accès: **chmod(...), fchmod(...)**
- copie de descripteur: **dup()**
- configuration de l'entrée associée à un descripteur: **fcntl()**

Chapitre 5

Entrées / Sorties

5.1 Introduction

- la manipulation des entrées sorties se fait par la notion de descripteurs de fichiers
- UNIX normalise toutes les E/S: réseaux (socket), fichiers (FILE), clavier (stdin), écran (stdout), etc

5.2 Répertoire

- la création d'un répertoire se fait avec la commande **int mkdir const char *ref, mode_t mode)**
- le type associé est **DIR**
- il faut inclure le fichier **dirent.h**
- l'ouverture d'un répertoire se fait avec **DIR *opendir(const char *ref)**
- le retour d'un pointeur NULL indique une erreur
- la structure **struct dirent** permet d'accéder aux noms de fichier à travers son champ d_name
- le parcours du répertoire se fait avec **struct dirent *readdir(DIR *p)**
  - un appel à readdir fait passer au fichier suivant dans le répertoire
  - un retour égal à NULL indique la fin du parcours
- on peut se replacer sur le premier fichier du répertoire avec **void rewinddir(DIR *p)**
- la fermeture du répertoire se fait avec **int closedir(DIR *p)**

Exemple:

```
DIR *repertoire;
struct dirent *fichier;
if (!(repertoire= opendir("/usr"))){...erreur...}
while ((fichier=readdir(repertoire))!=NULL)
        printf("%s ",fichier->d_name);
closedir(repertoire);
```

5.3 Fichiers

5.3.1 Ouverture / Fermeture

- La manipulation des fichiers se fait par le type **FILE**
- l'ouverture d'un fichier: **FILE* fopen(const char *ref, const char *mode)**
  ref est le nom du fichier

  mode est à choisir parmi: r (lecture), w (création ou ouverture + troncature), etc
- la fermeture d'un fichier: **int fclose(FILE *ptr_file)**

5.3.2 Ecriture / Lecture non formatée

- le fichier est vu comme une suite d'objets
- écriture avec **int fwrite(void *ptr, size_t taille, size_t nb_elem, FILE * ptr_file)**
- lecture avec **int fread(void *ptr, size_t taille, size_t nb_elem, FILE * ptr_file)**
- ces deux fonctions renvoient le nombre d'éléments écrits ou lus

5.3.3 Ecriture / Lecture formatée

- le fichier est vu comme une suite d'objets de types différents
- écriture avec **int fprintf(FILE * ptr_file, const char *format, ...)**
- lecture avec **int fscanf(FILE * ptr_file, const char *format, ...)**
- ces deux fonctions renvoient le nombre d'éléments écrits ou lus ou bien un code négatif en cas d'erreur
- le fichier est lisible

5.3.4 Autres fonctionnalités

- On peut se positionner à un endroit particulier dans les fichiers non formatés: **int fseek(FILE *ptr_file, long int offset, int origine)**
  offset représente un déplacement par rapport à l'origine qui peut être le début (**SEEK_SET**), la position courante (**SEEK_CUR**) ou la fin (**SEEK_END**)
- l'écriture réel sur le disque dur se fait quand le système d'exploitation le décide
- on force l'écriture (vidage des tampons) avec **int fflush(FILE *ptr_file)**

Chapitre 6

Processus

6.1 Caractéristiques

- un certain nombre de caractéristiques sont récupérables:

- identité du processus et de son père, respectivement **int getpid(void)** et **int getppid(void)**
- le propriétaire réel ou effectif du processus, respectivement **int getuid(void)** et **int geteuid(void)**
- le répertoire de travail: **char* getcwd(char * buf, size_t taille)**
- etc

6.2 Création de processus

- tous les processus sont créés à partir d'un père sauf le processus originel
- l'instruction utilisée est **int fork(void)**
- la valeur retournée permet de distinguer le fils et le père:
⇒ "0" pour le processus fils
⇒ "pid du fils" pour le procesus père

Exemple:

```
#include <unistd.h>
    ....
    int pid=0
    ...
    switch(pid=fork()){
        case -1: printf("erreur"); exit(-1); break;
        case 0: printf("on est chez le fils"); break;
        default: printf(" on est chez le pere et le pid du fils est % d",pid);
    }
    ...
```

- on ne sait pas à priori comment le père et le fils vont ensuite s'exécuter en concurrence
- génétique des processus:

- le fils hérite de quasiment toutes les caractéristiques de son père

- le fils a un pid différent
- le fils a un père différent
- les temps d'exécution du fils sont remis à 0
- le fils reçoit une copie des descripteurs du fils
- les signaux pendants ne sont pas hérités
- le fils a une copie des variables du pére

6.3 Synchronisation

- une synchronisation entre le père et le fils peut être nécessaire, notamment pour éviter les zombies
- on s'intéressera aux fonctions de la norme POSIX
- primitive permettant de tester en bloquant ou non le processus appelant si un processus ou un groupe de processus est terminé
- il faut inclure les fichiers: **sys/types.h** et **sys/wait.h**
- la fonction est: **int waitpid(int pid, int* status, int options)**
- le champ pid permet de définir le ou les fils concernés, -1 pour tous les fils
- options permet de savoir si le père reste bloqué ou non tant que le fils n'est pas mort et si les informations concernant la mort du fils sont transmises ou non au père
- la fonction renvoie -1 en cas d'erreur, 0 si le fils n'est pas terminé dans le mode non bloquant ou le pid du fils zombie pris en compte
- status retourne des informations sur la mort du fils:tué par un signal ou sortie volontaire du programme (exit, return). Il permet de récupérer soit la valeur du signal soit le code de fin
- la manipulation de status se fait avec des macros (WIFEXITED, WEXITSTATUS, etc)

6.4 Recouvrement

- un ensemble de primitives permettent d'exécuter un nouveau programme à l'intérieur d'un processus existant
- existence de différentes fonctions selon les besoins:

- gestion des arguments du programme
- chemin pour accéder au programme
- prise en compte de l'environnement de l'utilisateur

- si tout se passe bien on ne sort pas de l'appel de ces fonctions du type execXXX
- **int execl(const char *ref, const char *arg, ..., NULL)**
execl ("/bin/ls", "ls", "-l", "/", NULL)
- **int execv(const char *ref, const char *argv[])**

```
char *argc[4];
argv[0]="ls";
argv[1]="-l";
argv[2]="/";
argv[3]='NULL;
execv("/bin/ls",argv);
```

Chapitre 7

Les tubes

7.1 Introduction

- mécanisme de communication local d'UNIX entre processus
- communication unidirectionnelle
- utilise le mécanisme de fichier
- le tube possède une extrémité pour lire et une extrémité pour écrire
⇒ 2 entrées dans la table des fichiers ouverts
- l'opération de lecture détruit les données dans le tube
- la communication est un flot continu de caractères
- plusieurs écritures peuvent être lues en une seule fois et réciproquement
- la gestion des tubes est FIFO
- le tube a une capacité limitée

7.2 Tubes ordinaires ou non nommés

7.2.1 Caractéristiques

- Ce tube utilise un fichier lié à aucun répertoire
⇒ non visible dans le système de fichiers
- le fichier est détruit quand plus aucun processus n'utilise ce tube
- l'accès aux informations sur le tube ne peut se faire que par héritage (père - fils)

7.2.2 Ouverture et configuration

- la création d'un tube nécessite l'inclusion du fichier: **unistd.h**
- la fonction utilisée est: **int pipe(int p[2])**
- p[0] est utilisé pour la lecture
- p[1] est utilisé pour l'écriture
- schéma (relation avec table fichier ouvert)
- possibilité de récupérer des informations sur le tube (fstat) ou de modifier sa configuration (fcntl)
pour rendre l'écriture non bloquante:

```
int status;
status= fcntl(P[1], F_GETFL);
fcntl(p[0], F_SETFL, status | O_NONBLOCK);
```


7.2.3 Lecture

- la lecture se fait avec l'opération **int read(P[0],buf, taille_buf)**
- buf est un tableau de caractères
- taille_buf est le nombre maximum de caractères lus
- la fonction renvoie le nombre effectivement lu de caractères
- s'il n'y a pas de caractères à lire par défaut la fonction reste bloquée en attente de caractères
⇒ Attention aux problèmes d'interblocage

7.2.4 Ecriture

- l'écriture se fait avec: **int write(p[1],buf,taille_buf)**
- le nombre de caractères contenus dans le tube est limité à PIPE_BUF
- si taille_buf est inférieure à PIPE_BUF l'opération d'écriture est atomique
soit tous les caractères sont écrits soit l'écriture est reportée
- si la taille est supérieure à PIPE_BUF le système découpe le tampon
- s'il n'y a pas de lecteur, la demande d'écriture se solde par un SIGPIPE
- si le nombre de lecteur est non nul:
  - si l'écriture est bloquante, la fonction finie quand les taille_buf caractères sont écrits
  - si l'écriture est non bloquante
    - si n > PIPE_BUF, le retour est un nombre inférieur à taille_buf
    - si n <= PIPE_BUF et s'il y a au moins taille_buf emplacements libres, l'écriture est atomique et complète
    - si n <= PIPE_BUF et s'il y a moins de taille_buf emplacements libres, l'écriture échoue

7.3 Tubes nommés

- ces tubes peuvent être installés entre des processus sans lien de parenté
- les caractéristiques sont identiques au tubes ordinaires
- ils sont visibles dans le système de fichiers avec la commande **ls -l**
⇒ fichier de type "p"
- la création se fait avec l'instruction **int mkfifo(const char* nom, mode_t mode)**

Exemple:

- création d'un tube ou l'utilisateur peut lire / écrire et où des utilisateurs de son groupe peuvent lire
- mkfifo("/tmp/test_pipe",S_IRUSR | S_IWUSR | S_IRGRP)
- la destruction se fait en détruisant le fichier avec la fonction **int unlink(const char *pathname)**

7.4 Ouverture / Fermeture

- l'ouverture se fait avec la fonction **int open(const char * fichier, mode_t mode)**
- par défaut l'ouverture par le lecteur et l'écrivain est synchronisée: on parle d'ouverture bloquante
une demande d'ouverture en lecture est bloquée tant qu'il n'y a pas d'écrivain
une demande d'ouverture en écriture est bloquée tant qu'il n'y a pas de lecteur
⇒ attention dans le cas de l'ouverture de 2 tubes pour des communications bi-directionnelles, de ne pas créer une situation d'interblocage
- l'ouverture peut aussi être non bloquante avec le bit **O_NONBLOCK**:
  - ouverture non bloquante en lecture réussit même s'il n'y a pas d'écrivain
  - ouverture non bloquante en écriture échoue s'il n'y a pas de lecteur

Exemple:

```
int d_lecture, d_ecriture;
d_ecriture = open("/tmp/fifo1", O_WRONLY);
d_lecture = ("/tmp/fifo2",O_RDONLY);
```

Exemple:
mise en place de processus producteur/consommmateur

Chapitre 8

Communication inter processus: IPC

8.1 Caractéristiques

- mécanismes extérieurs au système de fichiers
⇒ pas manipulé par des descripteurs de fichiers
- les IPC sont constitués de:

  - les sémaphores
  - la mémoire partagée
  - les files de messages

- ce cours ne traitera pas des files de messages
- chaque mécanisme possède sa propre table
- utilisation d'un identificateur unique dans la table pour un objet (parallèle avec les descripteurs de fichiers)
- au niveau externe les objets sont référencés par des clés de type **key_t**, définies dans **sys/types.h**
- les clés sont définies par type d'objet
- le fichier **sys/ipc.h** décrit les constantes de base
- il existe des commandes shell permettant de visualiser et de détruire des objets IPC
  - commande **ipcs**: permet la consultation des tables IPC et affiche
    - T: type d'IPC: q message, m mémoire partagée, s sémaphores
    - ID: identification interne de l'objet
    - KEY: clé de l'objet (0x00000000 clé privée)
    - MODE: droits d'accés
    - OWNER et GROUP repectivement le propriétaire et son groupe sous UNIX
    - les options **-q**,**-m** et **-s** permettent de n'afficher respectivement que les files de messages, la mémoire partagée ou les sémaphores

    -a affiche tout
  - commande **ipcrm** supprime une entrée dans une table
    - les options **-q**,**-m** et **-s** permettent de sélectionner la famille d'IPC, elles sont suivi du numéro d'identification interne de l'objet à détruire

8.2 Sémaphores

8.2.1 Rappel

- Sémaphore proposé par Dijkstra et utilisé pour la synchronistion
- les deux opérations possibles sont P et V, elles sont atomiques:
  - P(S): si $S = 0$ alors mettre le processus en attente sinon $S = S - 1$
  - V(S): $S = S + 1$, réveiller un (ou plusieurs) processus en attente
- l'implémentation fournie par UNIX est plus générale: l'incrément ou le décrément peuvent être différents de 1: $P_n(S)$, $V_n(S)$
- on peut tester la valeur d'un sémaphore
- les explications fournis dans ce cours se limite à un cas simple d'utilisation
- inclusion du fichier **sys/sem.h**

8.2.2 Création et contrôle

- La création passe par la récupération d'un sémaphore à partir d'une clé
- La clé est un nombre entier choisi par l'utilisateur
- la clé permet de partager le même sémaphore entre plusieurs processus
- l'instruction utilisée est: **int semget(key_t key, int nsem, int semflag)**
- elle permet de créer un sémaphore ou un ensemble de sémaphores
- **key** est la clé, **nsem** le nombre de sémaphores dans cet ensemble, **semflag** les droits et la méthode de création
- la fonction renvoie l'identificateur du sémaphore

```
int key = 200;
int semaphore;
semaphore = semget((key_t) key, 1, 0666 | IPC_CREAT);
```

- IPC_CREAT crée le sémaphore s'il n'existe pas déjà
- les droits 0666 permettent l'écriture / lecture pour l'utilisateur, le groupe et le monde

  - un des processus utilisateur doit ensuite initialisé la valeur initiale du sémaphore
- une instruction de contrôle permet d'effectuer des opérations autres que P et V

  int semctl (int semid, int semnum, int cmd, union semun arg)

  semid est l'identificateur du sémaphore

  semnum le numéro dans l'ensemble des sémaphores d'identificateur semid, 0 correspond au premier sémaphore

  cmd est la commande à appliquée: SETVAL pour initialiser le sémaphore, IPC_RMID pour détruire le sémaphore, etc

  arg correspond aux arguments nécessaires à la commande. La structure contient les champs:

```
union semun {
int val; /* valeur pour SETVAL */
struct semid_ds *buf; /* tampon pour IPC_STAT, IPC_SET */
unsigned short int *array /* tableau pour GETALL, SETALL */
struct seminfo * __buff /* tampon pour IPC_INFO */
};
```

arg contient une valeur d'initialisation pour SETVAL, RMID n'a pas d'argument (arg=0)

```
semctl(semaphore,0,IPC_RMID,0)
union semun arg;
arg.val=1;
semctl (semaphore, 0, SETVAL, arg)
```

8.2.3 Manipulation

- la manipulation se fait avec l'instruction **int semop (int semid, struct sembuf *sops, unsigned nsops)**
  - semid est l'identificateur du groupe de sémaphores
  - sops est un tableau de nsops structures donnant l'action à réaliser sur chaque sémaphore de l'ensemble

```
struct sembuf{
    short sem_num; /* numéro du sémaphore: 0 = le premier */
    short sem_op; /* opération sur le sémaphore */
    short sem_flg; /* flag l'opération */
}
```

- une valeur positive N pour sem_op correspond à V(N)
- une valeur négative N pour sem_op correspond à P(N)

```
struct sembuf buf;
buf.sem_num = 0;
buf.sem_flg = 0;
buf.sem_op = -1;
semop(semaphore, &buf, 1)
```

8.2.4 Exemple

voir cours

8.3 Mémoire partagée

8.3.1 Principe

- partage d'une zone mémoire entre deux processus
- manipulation aisée contrairement aux sémaphores
- inclusion du fichier <**sys/shm.h**>
- accès identique à de la mémoire propre
  ⇒ transparent pour l'utilisateur quand il accède à la zone
- attention aux problèmes de cohérence
- mécanisme de communication local efficace
- autre méthode sous UNIX: mmap, mappage de pages pour le partage directement en mémoire

8.3.2 Création et contrôl

- La création se fait en 2 temps:

  - demande d'une entrée dans la table des IPC-SHM et allocation si nécessaire d'un zone mémoire
  - attachement de la zone à une adresse dans l'espace d'adressage du processus

  - la création se fait avec l'instruction: **int shmget(key_t key, int size, int shmflg)**
- la variable size représente la taille en octets
- le système allouera un certain nombre de pages dont la somme sera supérieure ou égale à size
- il existe une limitation sur la taille de la SHM (paramètre du noyau)
- la variable shmflg correspond au droit et au type de création (voir sémaphore)

- la fonction renvoie un identificateur pour la SHM

  - l'attachement de la zone de mémoire se fait avec **void *shmat (int shmid, void *shmaddr, int shmflg)**
- la fonction renvoie l'adresse à manipuler par l'utilisateur
- shmid est l'identificateur de la SHM
- les deux autre champs servent à paramétrer l'attachement
  avec les valeurs 0 et 0 on dispose d'une SHM en lecture écriture
- pour plus d'information voir man

8.3.3 Manipulation

- la manipulation se fait comme pour un pointeur quelconque
- on doit faire un "cast" pour adapter le pointeur à son type car par défaut c'est un (void*)

8.3.4 Détachement et destruction

- le détachement doit être effectué par tous les processus ayant utilisé la SHM
- un seul processus doit détruire la SHM
- le détachement se fait avec: **int shmdt(void *shmaddr)**

8.3.5 Exemple

voir cours

Chapitre 9

Mécanismes complémentaires

9.1 La compilation, débogueur, etc

9.1.1 Compilation

Queslques options de compilation de gcc:

- "-E" arrête la compilation après le préprocesseur
- "-c" arrête la compilation après l'assemblage laissant les fichiers .o disponibles
- "-pedantic" fournit des avertissements encore plus rigoureux que Wall
- "-g" inclut dans le code des informations nécessaire pour le débogueur
- "$-O$" avec les arguments de 0 à 3: 0 pas d'optimisation, 3 optimisation maximale

9.1.2 Débogueur

- il existe un débogueur en ligne de commande: gdb
- interface xxgdb et maintenant sous linux ddd
- ensemble de commandes : point d'arrêt, visualisation des valeurs de variables
- ouverture du fichier core pour regarder l'état d'un programme au moment de son plantage
- possibilité de s'accrocher à un programme en exécution

9.1.3 Profiling

gprof

- outils permettant de faire une post-analyse d'une exécution
- analyse de performance: où mon programme à passer du temps
- utilisation de l'option de compilation "-pg"
- génération durant l'exécution d'un fichier gmon.out
- analyse du fichier avec la commande gprof
- exemple :

- compilation: gcc -Wall -pg testgprof.c -o testgprog

- exécution: ./testgprof
- analyse: gprog testgprof gmon.out

strace

- outils strace permettant de suivre les appels système
- utilisation: strace ./monprogramme
- affichage sur la sortie d'erreur des différents appels
- permet par exemple de comprendre dans quel appel système un programme plante

nm

- nm permet de connaître les symboles utilisés dans un programme

9.1.4 indent

- permet d'indenter un programme en C déjà écrit
- indent monprog.c
- possibilité de mettre une option pour choisir l'aspect

9.2 Environnement

- définition sous forme NOM=VALEUR
- lors d'un fork, le fils hérite des variables d'environnement de son père
- un processus ne peut toucher que ses propres variables d'environnement
- copie automatique des variables dans le tableau de chaînes de caractères défini dans la bibliothèque: char **environ
- utilisation en déclarant: extern char ** environ
- chaque chaîne est stockée ainsi: NOM=VALEUR
- le dernier pointeur sur chaîne du tableau est à NULL
- la fonction char* getenv (const char *nom) permet de récupérer la valeur d'une variable, elle renvoit la chaîne après "="
- la fonction int putenv(const char * chaine) permet de positionner une variable avec sa valeur

9.3 Terminaison d'un programme

- demande de terminaison anormale: void abort(void): envoi du signal SIGABRT
- inconvénient : on ne sait pas ou le programme s'est stoppé dans le cas de plusieurs appels à abort
- macro assert(condition): stop le programme et affiche l'endroit où le programme s'est arrêté
- exécution d'une dernière fonction à la fin d'un programme mise en place avec int atexit(void (*routine)(void))
- récupération des ressources utilisées: int getrusage(int who,struct *rusage *ru);
- par exemple: temps utilisateur, temps système, gestion de l'ordonnanceur, taille de la pile, etc

9.4 Gestion avancée de la mémoire

- travailler sur un fichier en mémoire directement
- permet aussi de créer une zone de mémoire partagée entre processus

- possiblité de synchroniser les modifications en mémoire et sur le fichier sur le disque (aux optimisations près de l'OS)
- utilisation de mmap(NULL, taille en octet, type d'accès, attribut, fichier, offset)

9.5 Cryptage

- inclusion de crypt.h ou unistd sur mac
- utilisation de la fonction: char * crypt(const char* mot_passe, const char * prefixe)
- possibilité de crypter en MD5 (préfixe commençant par 1) ou DES
- un bon préfixe doit contenir au plus 8 caractères de 0 à 9; a à z, A à Z , '.' et '/'
- exemple pour crypter en MD5 la chaine "coucou" crypt("coucou","1Z8sar");

9.6 Fonctions horaires

- nombre de secondes depuis le 1er janvier 1970: time(NULL)
- précision plus forte avec int gettimeofday(struct timeval *timev, struct timezone timez)
- la struct timeval est constituée de 2 champs: tv_sec nombre de secondes depuis le 1er janvier 1970 et tv_user nombre de microseconde depuis la dernière modif de tv_sec
- transformation en une heure classique avec la fonction localtime et la structure struct tm

9.7 Informations sur le système

- identification du noyau: int uname (struct utsname *ut)
- utsname contient les champs: sysname, nodename, release, version, machine