UNIVERSITÉ TOULOUSE III PAUL SABATIER

# Threads

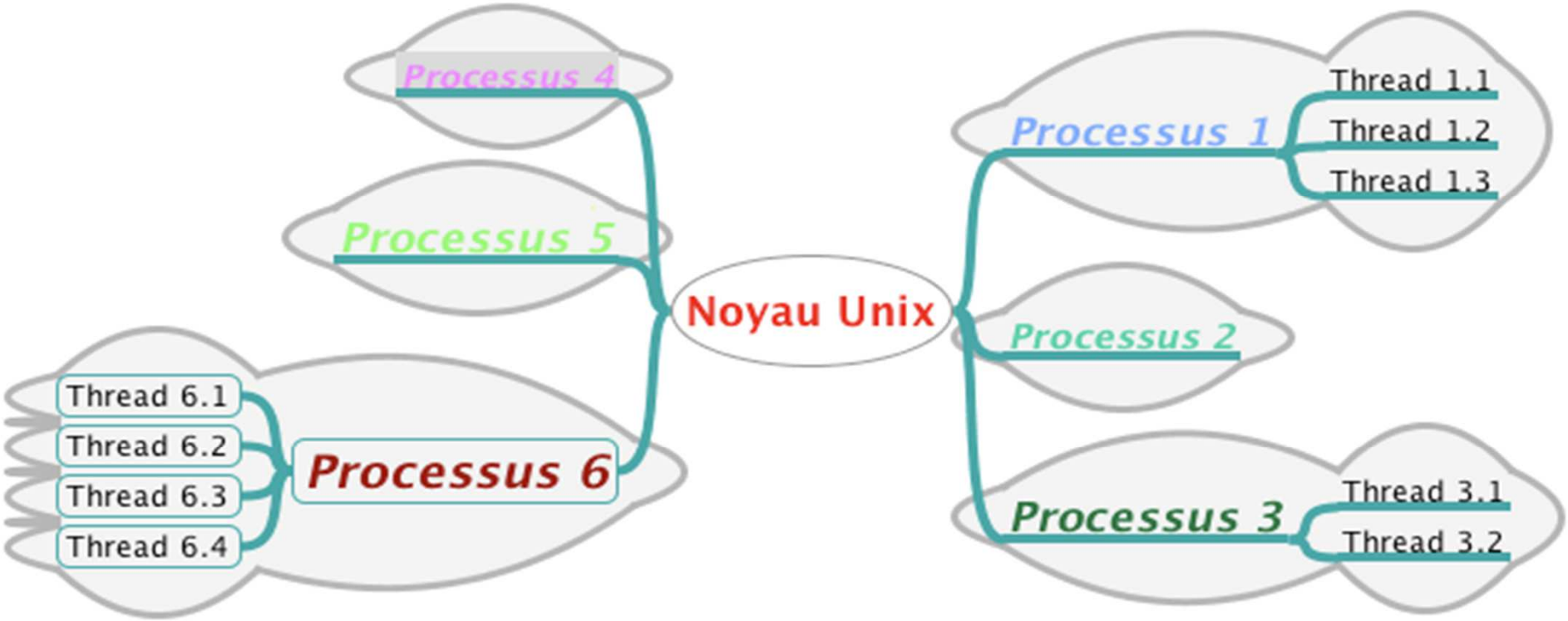
# Concept de processus (lourd)

- Un seul flot de contrôle séquentiel par processus
- Un espace d'adressage par processus
  - Il n'existe pas d'espace partagé entre deux processus
- Le processus est l'unité d'allocation de ressources pour le système
- Le processus constitue l'unité d'ordonnancement

# Concept de thread (processus léger)

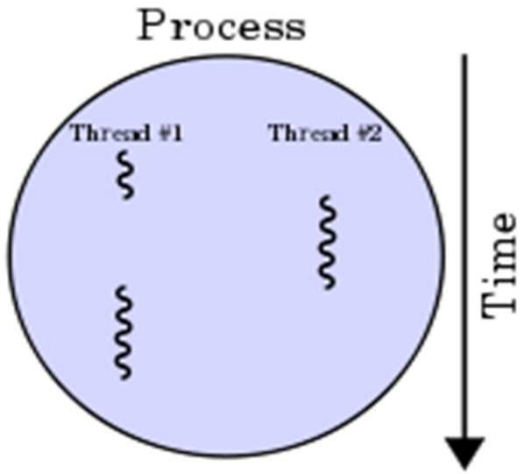
- Plusieurs flots de contrôle séquentiels
  - Les flots de contrôle sont concurrents
- Un seul espace d'adressage
  - Espace partagé entre les threads
- Le processus reste l'unité d'allocation de ressources pour le système
- Le processus n'est pas l'unité d'ordonnancement (selon l'option choisie)
  - Les threads peuvent être directement gérés par l'ordonnanceur du système

# Schéma d'activation



# Qu'est-ce qu'un thread ?

- Flux d'exécution au sein d'un processus
  - Fonction main() = thread « principal »
  - Exécution classique « séquentielle » jusqu'à fin du main()
  - Le processus se termine ➔ les threads aussi (en C/Unix)
- Création dynamique de threads
  - Un ou plusieurs flux d'exécution au sein d'un processus (threads « compagnons »)
- Ressources du processus partagées entre ses threads
  - Espace d'adressage ➔ « mémoire partagée »
- Sur machine multiprocesseur
  - Exécution parallèle



# Ressources d'un thread

- Chaque thread dispose de ressources propres
  - d'un morceau de code à exécuter
  - de données sur lesquelles le code travaille
  - d'une pile d'exécution permettant de gérer la dynamique du thread

- Les threads compagnons disposent de ressources communes
  - l'espace d'adressage (données et code)
  - les variables globales
    - Lorsqu'un thread modifie une variable globale, la nouvelle valeur est immédiatement visible des autres threads compagnons
  - les fichiers ouverts, le répertoire courant
  - le masque de création
  - …

# Attributs d'un thread

- Chaque thread dispose d'attributs spécifiques
  - Un identificateur unique
  - de données propres définies par l'utilisateur
  - une pile, un compteur ordinal, des registres CPU
  - un état (actif, bloqué ou prêt)
  - de la variable système `errno`
  - d'un masque de signaux bloqués
  - …

# Avantages des processus multi-threads

- Amélioration du rendement de l'application
  - Exemple du serveur web
- Meilleure utilisation des multi-processeurs
- Structuration de programme souvent mieux adaptée
  - Souvent, la plupart des threads sont en attente
- Amélioration de la communication de données entre activités

# Implantation des threads (1/2)

- **Les threads peuvent être implantés**
  - en tant qu'abstractions de niveau utilisateur
  - en tant qu'abstractions de niveau noyau
  - comme combinaison des deux

- **Abstraction au niveau utilisateur**
  - Principes
    - Ils sont gérés directement à l'intérieur de l'espace d'adressage d'un processus
    - N'utilise aucun service système, à l'exception de ceux associés au processus
    - Les threads compagnons sont multiplexés sur le processus et par le processus pour être exécutés
  - Avantages
    - Meilleures performances (pas de commutation noyau)
    - Ensemble de fonctionnalités extensible sans nouvelle version système

# Implantation des threads (2/2)

- **Abstraction au niveau noyau**
  - Chaque thread dispose de son propre contexte, d'un espace privé, d'une pile propre
  - **Inconvénients**
    - Accroissement de la taille du noyau
    - Système figé dans ses fonctionnalités
- **Combinaison des deux**
  - **Un thread noyau pour un thread utilisateur**
    - Le thread noyau est qualifié LWP (lightweight process)
    - Le thread utilisateur est le thread lié au LWP
    - Solution intégrant les avantages et les inconvénients des deux solutions précédentes
  - **Multiplexage des processus liés sur un même LWP**

UNIVERSITÉ TOULOUSE III PAUL SABATIER

# Les threads POSIX

# La norme POSIX

- **Normalisation produite par IEEE et standardisée par ANSI et ISO**
  - **POSIX 1003.1 : OS, processus, SGF, API**
  - **POSIX 1003.2 : utilitaires**
  - **POSIX 1003.1b : temps réel**
  - **POSIX 1003.1c : threads ➔ pthreads**
  - **POSIX 1003.1d : extensions TR supplémentaires**
- **La norme POSIX comporte le composant logiciel DCE (Distributed Computing Environment) qui offre :**
  - **le type** thread
  - **le type** mutex
  - **le type** condition
  - **le type** exception

# Bibliothèque pthread – Les bases

- Synchronisation de base : mutex et condition
- Types (opaques) : <sys/types.h>
  - pthread_t, pthread_key_t
  - pthread_mutex_t
  - pthread_cond_t
  - pthread_once_t
- Attributs : standard + propres à l'implantation
  - pthread_attr_t
  - pthread_mutexattr_t
  - pthread_condattr_t
- Bibliothèque ➔ #include <pthread.h> + compilation : -lpthread
- Gestion des erreurs
- Primitives « thread-safe »

# Création d'un thread

```
int pthread_create(pthread_t                  *thread,
                   const pthread_attr_t *attr,
                   void                  *(*start_routine)(void*),
                   void                  *arg);
```

- start_routine = fonction exécutée par le thread
- arg = argument de cette fonction
- attr = attributs optionnels de création
- thread = identificateur

- Toutes les ressources nécessaires au thread doivent avoir été initialisées
- Erreurs possibles :
  - EINVAL : attributs invalide
  - EAGAIN : ressources insuffisantes

# Identité du thread actif

- Retourne l'identification du thread actif
  - Équivalent pour les processus de getpid()

```
#include <pthread.h>
#include <unistd.h>

pthread_t pthread_self (void);
```

# Terminaison d'un thread

```
#include <pthread.h>
#include <unistd.h>

void pthread_exit (void *retval);
```

- retval = pointeur sur le résultat retourné par le thread qui se termine

- **ATTENTION : Le pointeur doit repérer une zone qui sera encore accessible après la terminaison du thread (qui ne peut pas être une variable locale au thread car allouée dans la pile et donc détruite à la terminaison du thread)**
  - Fait passer le thread dans un état « zombie » jusqu'à ce que le résultat rendu par le thread soit effectivement récupéré par un thread compagnon (voir pthread_join)
  - Le thread se termine et pthread_exit ne retourne donc rien

# Attente de la terminaison

```
#include <pthread.h>
#include <unistd.h>

int pthread_join ( pthread_t   thread,
                   void        **retval );
```

- thread = identificateur du thread attendu
- retval = est un pointeur `void **` contenant l'adresse d'une variable `void *` devant recevoir le code retour du thread attendu (pour rappel, un pointeur `void *`)
- Le thread attendu se termine réellement après réception de la valeur retournée (il quitte l'état « zombie »)
- Retourne 0 en cas de succès, un code d'erreur sinon
  - EDEADLCK, EINVAL, ESRCH

# Exemple – Code d'un thread

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define VAL 100

/* Fonction exécutée par le thread */
void *f_thread (void *p) {
  int *cr = malloc(sizeof(int));
  printf("\tDebut du thread compagnon ");
  printf("\tIci le thread numero: %lu \n", pthread_self());
  *cr = VAL;
  printf("\tValeur retournee: %d \n", *cr);
  pthread_exit((void*)cr);
  /* On pourrait aussi écrire : return ((void *)cr);
  ou return ((void *)&cr); si cr était déclarée en global par : int cr */
}
```

# Exemple – Thread principal

```
#include <st
#include <pt
#include <ur
#define VAL

/* Fonction
void *f_thre
  int *cr =
  printf("\t
  printf("\t
  *cr = VAL;
  printf("\t
  pthread_
  /* On pour
  ou return
}
```

```
int main(){
  pthread_t ptid;
  int *res = NULL;

  printf("Debut du thread principal\n");

  /* Creation du thread compagnon */
  if ( pthread_create(&ptid, NULL, f_thread, NULL) != 0){
    perror("Probleme lors de la creation du thread compagnon:");
    exit(99);  /* plutôt pthread_exit() si hors thread principal */
  }

  /* Attente fin d'execution du thread */
  pthread_join(ptid, (void**)&res);

  printf("Test resultat: %d \n", *res);
  free(res);
  printf("Fin du thread principal\n");
}
```

# Exemple – Exécution

```
#include <st
#include <pt
#include <un
#define VAL

/* Fonction
void *f_thre
  int *cr =
  printf("\t
  printf("\t
  *cr = VAL;
  printf("\t
  pthread_
  /* On pour
  ou return
}
```

```
int main(){
  pthread_t ptid;
  int *res = NULL;
  pthread_attr_t attri

  printf("Debut du thr

  /* Creation du threa
  if ( pthread_create(
    perror("Probleme l
    exit(99);  /* plut
  }

  /* Attente fin d'exe
  pthread_join(ptid, (void**)&res);

  printf("Test resultat: %d \n", *res);
  free(res);
  printf("Fin du thread principal\n");
}
```

```
Go %./ex1-0
Debut du thread principal
        Debut du thread compagnon
        Ici le thread numero: 25166848
        Valeur retournee: 100
Test resultat: 100
Fin du thread principal
Go %
```

# Exercice 1 – Comparaison Processus / Threads

❑ On donne le code suivant :

```c
void erreur (char *msg, int codeRet) {
  perror(msg);
  exit(codeRet);
}

void genererValeur (int num) {
  int val;

  srand(getpid());
  val = rand() % 255;
  printf("Valeur choisie par le fils %d (%d) = %d\n",
                  num, getpid(), val);
  exit(val);
}
```

**Réécrire ce code pour que les activités parallèles soient des threads**

```c
int main(int argc, char *argv[]) {
  int i, nbFils, cr, val, somme = 0;
  pid_t pidFils[NB_FILS_MAX], pid;

  if (argc != 2) {
    printf("Usage : %s <Nb fils>\n", argv[0]);
    exit(1);
  }
  nbFils = atoi(argv[1]);
  if (nbFils > NB_FILS_MAX)
     nbFils = NB_FILS_MAX;

  for (i = 0; i < nbFils; i++)
    switch(pidFils[i] = fork()) {
      case -1 : erreur("fork", 1);
      case 0 : genererValeur(i);
    }

  while ((pid = wait(&cr) != -1)) {
    val = WEXITSTATUS(cr);
    printf("Valeur retournee par le fils %d = %d\n", pid, val);
    somme += val;
  }
  printf("Somme = %d \n", somme);
  return 0;
}
```

# Les attributs d'un thread

- **detachstate**
  - Thread détaché ou non
  - Valeur :
    - PTHREAD_CREATE_JOINABLE
    - PTHREAD_CREATE_DETACHED
- **stacksize**
  - Si _POSIX_THREAD_ATTR_STACKSIZE
  - Taille minimale de la pile du thread > PTHREAD_STACK_MIN
- **stackaddr**
  - Si _POSIX_THREAD_ATTR_STACKADDR
  - Adresse de la pile du thread
- **scope**
  - Portée de la compétition pour l'UC
  - Valeur :
    - PTHREAD_SCOPE_SYSTEM
    - PTHREAD_SCOPE_PROCESS
- **inherit_scheduler**
  - Hériter de son créateur
  - Valeur :
    - PTHREAD_INHERIT_SCHED
    - PTHREAD_EXPLICIT_SCHED
- **schedpolicy**
  - Politique d'ordonnancement
    - SCHED_FIFO, SCHED_RR, SCHED_OTHER

# Les attributs d'un thread – Initialisation/destruction

```
int pthread_attr_init (pthread_attr_t *);
```

- Un attribut doit être initialisé avant de l'utiliser pour créer un thread
  - Peut servir à créer plusieurs threads

```
int pthread_attr_destroy(const pthread_attr_t *);
```

- Un attribut non utilisé doit être détruit
  - Aucun effet sur le thread qui a été créé avec cet attribut

# Attributs d'un thread – Politique d'ordonnancement

- Valeurs : SCHED_FIFO, SCHED_RR, SCHED_OTHER
- Politique utilisée

```
int pthread_attr_getschedpolicy(const pthread_attr_t *,
                                int *);
int pthread_attr_setschedpolicy(pthread_attr_t *, int);
```

- Paramètres de la politique utilisée <sched.h>

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param   *param);
```

- Héritage

```
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int         *inheritedsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int          inheritedsched);
```

# Attributs d'un thread – Portée de la compétition

```
int pthread_attr_setscope(pthread_attr_t *, int );

int pthread_attr_getscope(const pthread_attr_t *, int *);
```

- **Portée (scope) de la compétition**
  - **Même processus : PTHREAD_SCOPE_PROCESS**
  - **Processus différents : PTHREAD_SCOPE_SYSTEM**

**Consulter le man pour plus de détails sur les fonctions gérant les attributs**

# Attributs d'un thread – Priorité

```
int sched_get_priority_max(int policy) ;

int sched_get_priority_min(int policy) ;
```

- Intervalle de priorité pour la politique utilisée (<sched.h>)
  - Valeur de priorité minimale
  - Valeur de priorité maximale

# Exemple – Utilisation d'un attribut

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define VAL 100

/* Fonction exécutée par le thread */
void *f_thread (void *p) {
  int *cr = malloc(sizeof(int));
  printf("\tDebut du thread compagnon");
  printf("\tIci le thread numero: %lu \n", pthread_self());
  *cr = VAL;
  printf("\tValeur retournee: %d \n", *cr);
  pthread_exit((void*)cr);
  /* On pourrait aussi écrire : return ((void *)cr);
  ou return ((void *)&cr); si cr était déclarée en global par : int cr */
}
```

# Exemple – Utilisation d'un attribut

```
#include <st
#include <pt
#include <ur
#define VAL

/* Fonction
void *f_thre
  int *cr =
  printf("\t
  printf("\t
  *cr = VAL;
  printf("\
  pthread_
  /* On pour
  ou return
}
```

```
int main(){
  pthread_t ptid;
  int *res = NULL;
  pthread_attr_t attribut;

  printf("Debut du thread principal\n");
  /* Si le « scope » n'est pas déjà le système par défaut */
  pthread_attr_init(&attribut);
  pthread_attr_setscope(&attribut, PTHREAD_SCOPE_SYSTEM);
  /* Creation du thread compagnon*/
  if ( pthread_create(&ptid, &attribut, f_thread, NULL) != 0){
    perror("Probleme lors de la creation du thread fils:");
    exit(99);  /* plutôt pthread_exit() si hors thread principal */
  }
  pthread_attr_destroy(&attribut);
  /* Attente fin d'execution du thread */
  pthread_join(ptid, (void**)&res);
  printf("Test resultat: %d \n", *res);
  free(res);
  printf("Fin du thread principal\n");
}
```

# Exercice 2

**On se propose de paralléliser le traitement d'une matrice de réels en confiant le traitement de chaque ligne – calculer la somme des éléments de cette ligne – à un thread.**

**Le thread initial saisit au clavier le contenu de la matrice, active les threads sous-traitants, puis calcule et affiche la somme des valeurs qu'ils ont calculé**

**Version 1) Syntaxe d'appel de la commande :**

**% traiterMatrice**

**Version 2) Syntaxe d'appel de la commande :**

**% traiterMatrice NB_LIGNES NB_COLONNES**

***L'évolution de la version 1 à la version 2 doit entraîner le minimum de modification***

**On suppose que les fonctions suivantes existent :**

```
void saisirMatrice(float mat[NBLMAX][NBCMAX], int nbL, int nbC);
float sommeLigne(float mat[NBLMAX][NBCMAX], int nbL, int nbC, int numL);
```

# Exercice 3 (facultatif)

**On se propose de paralléliser le traitement de plusieurs fichiers textes en confiant le traitement de chaque fichier à un thread.**

**Le traitement effectué par chaque thread consiste à calculer le nombre d'occurrences de chaque caractère alphabétique.**

**Le thread initial récupère les résultats de ces traitements pour effectuer une synthèse de leurs travaux en affichant :**

- **le nombre d'occurrences de chaque caractère alphabétique dans l'ensemble des fichiers traités**
- **pour chaque caractère alphabétique, les noms des fichiers qui n'en contiennent aucune occurrence**

**Syntaxe d'appel de la commande :**

```
% traiterFichiers nomFichier […]
```

# Réévaluation de l'affectation de l'UC

- Fait passer le thread appelant de l'état actif à l'état prêt, puis élit un nouveau thread
- Erreur
  - Retour : -1 + errno positionné
  - ENOSYS : non supporté

```
#include <pthread.h>
#include <unistd.h>

int pthread_yield (void);
```

# Destruction d'un thread (1/2)

- **État possible pour une destruction**
  - Interdit
  - Autorisé en fonction du type
    - Différé : au prochain point de destruction
    - Asynchrone : n'importe quand ➔ risques
- **Points de destruction**
  - Tout appel bloquant
  - Certains appels système
  - Précisé par le programmeur
- **Demande de destruction d'un thread**
  - Nettoyage avant destruction
  - Erreur : ESRCH

```
int pthread_cancel (pthread_t thread);
```

# Destruction d'un thread (2/2)

- Positionne l'état de destruction de l'appelant
  - PTHREAD_CANCEL_ENABLE/PTHREAD_CANCEL_DISABLE
  - Erreur : EINVAL

```
int pthread_setcancelstate (int type, int *oldtype);
```

- Positionner le type de destruction de l'appelant
  - PTHREAD_CANCEL_DEFERRED/PTHREAD_CANCEL_ASYNCHRONOUS
  - Erreur : EINVAL

```
void pthread_setcanceltype (int type, int *oldtype);
```

- Positionner un point de « destruction »

```
void pthread_testcancel (void);
```

# Threads et signaux (1/2)

- À utiliser avec précaution ➔ 1 seul thread gestionnaire
- Cible :
  - Signaux asynchrones : processus
  - Signaux synchrones (déroutements, matériel) : thread
- Masque des signaux propre, hérité
- État des signaux global : un seul gestionnaire

# Threads et signaux (2/2)

## Modification du masque

- Cf. sigprocmask() mais contexte multi-threadé
- Erreur : EINVAL

```
int pthread_sigmask(int how, const sigset_t *set,
                                 sigset_t *oset) ;
```

## Envoi restreint au sein du processus

- thread = récepteur
- sig = signal (= 0 => contrôle d'erreur seulement)
- Erreurs : ESRCH, EINVAL

```
int pthread_kill(pthread_t thread, int sig) ;
```