Programmation Avancée Multimédia

✓ Multithreading

Benoît Piranda
Équipe SISAR
Université de Marne La Vallée

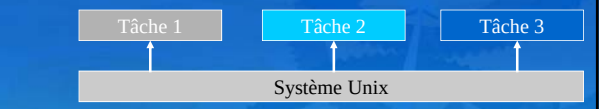
Besoin

- **Programmes à traitements simultanés**
  - **Réseau**
    - **Afficher une animation en temps réel**
    - **Lire un flux réseau**
      - ✓ **Surveillez les arrivées réseaux pour ne pas perdre de trame**
  - **Jeu / Audio**
    - **Module principale de jeu**
    - **Envoyer des sons sur la carte audio**
      - ✓ **Ne pas interrompre l'émission des sons**
  - **Vidéo**
    - **Afficher la vidéo**
    - **Récupérer les trames sur un flux DV**

Principe

- **Plusieurs programmes s'exécutant simultanément**
  - **Processus**
  - **Répartition du temps d'exécution par le système**
- **Échange rapide d'information**
  - **Mémoire partagée**
  - **Gestion des accès à la mémoire**
- **Synchronisation des tâches**
  - **Attente d'un processus par un autre**

Programme multitâches



Time sharing

Multitâche et multiprocesseur

- **Les tâches sont réparties sur les différents processeurs au niveau du time sharing**
  - **Un processus n'est pas associé à un processeur**

Outils de programmation multiprocessus

- **La primitive fork**
  - **Provoque**
    - **la duplication du programme**
    - **l'exécution simultanée de deux versions du programme au niveau de l'appel fork**
  - **La répartition des traitements sont à la charge du programme**
    - **Différenciation du processus père et du processus fils par la fonction fork**
  - **Partage des données allouées avant le fork**

La programmation de la primitive fork

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

main()
{ pid_t pid;

 switch(pid=fork())
 { case (pid_t)-1 :
    perror("erreur à la création du processus");
    exit(1);
  case (pid_t)0 :
    /* traitement du processus fils */
    printf("Proces n°%d, de père %d\n",getpid(),getppid());
    exit(0);
  default :
    /* traitement du processus pere */
    printf("Proces n°%d, de père %d\n",getpid(),getppid());
 }
}
```

Outils de programmation multiprocessus

- **Les threads**
  - **Exécution d'une fonction comme un processus concurrent**
  - **Partage des données globales du programme**
- **Thread père**
  - **Processus qui créé les autres processus (fils)**
  - **Tue tous les processus fils lorsqu'il se termine**

Fonctions associées aux threads

int pthread_create(pthread_t *id, pthread_attr_t attr,
void *(*fonct) (void *arg),void *arg);

Créer un pthread.
id : variable identificatrice, **attr** : pthread_attr_default,
fonct : tâche, **arg** : argument de la fonction
Retourne -1 si erreur.

int pthread_detach(pthread_t *id);
Terminer un thread.
id : variable identificatrice.
Retourne -1 si erreur.

int pthread_join(pthread_t id, void **status);
Attend la fin d'un pthread.
id : variable identificatrice.
status : pointeur resultat de la fonction.
retourne -1 si erreur.

Programme d'exemple

```
#include <pthread.h>

/* variables globales communes et accessibles à toutes les tâches */
int entierPartage;
/* variables globales privées différentes pour chaque tâche */
__private int entierPrive;

/* fonction exécutée comme nouvelle tâche */
void *proc(void *ptrDonnees)
{ int NumeroTache = (int *)ptrDonnees;
  int resultat;
/* autres variables locales à la tâche */
:
/* corps de la tâche */
:
  return ( (void *)resultat );
}
:
:
```

Programme d'exemple

```
:
:
void main(void)
{ pthread_t pth1,pth2;
  void *PtrResult;

  if (pthread_create(&pth1,pthread_attr_default,proc,(void *)1 )==-1)
    perror("Erreur lors de la creation de la tâche 1");
  if (pthread_create(&pth2,pthread_attr_default,proc,(void *)2 )==-1)
    perror("Erreur lors de la creation de la tâche 2");
/* traitement de la tache de controle */

/* jonction à la fin des taches 1 & 2 */
  if (pthread_join(pth1,&PtrResult)==-1)
    perror("Erreur lors de la jonction avec la tâche 1");
  if (pthread_join(pth2,&PtrResult)==-1)
    perror("Erreur lors de la jonction avec la tâche 2");
}
```

Synchronisation des processus

- **Objets associés aux pthreads**
  - **Les mutex : pour l'accès exclusif à la mémoire partagée**
  - **Les variables conditionnelles : synchronisation des processus à un autre par endormissement et réveil**
  - **Barrière : synchronisation des processus à un point de rendez-vous.**

Gestion de la mémoire partagée

- Les accès à la mémoire partagée doivent être sécurisés
  - À un instant donné, un processus doit être seul à accéder à la mémoire
  - Il doit pouvoir se réserver l'accès à la mémoire
  - Justifié pour les données importantes (tableaux)
- Utilisation de sémaphore
  - mutex : *mutual exclusion*

Problème des accès à la mémoire

Processus 0

Recherche l'indice de la valeur 1

```
ptr=tab;
i=nbre;
s=0;
while (i--)
{ if (*ptr) s++;
  ptr++;
}
```

*Tableau d'entiers **tab[nbre]***
*nombre d'éléments **nbre***

0
0
0
0
0
0
0
0
1
0
0
0
0
0

Processus 1

Permutation de valeurs

```
i=nbre*rand()/(RAND_MAX+1)
j=nbre*rand()/(RAND_MAX+1)

tmp = tab[i];
tab[i]=tab[j];
tab[j]=tmp;
```

Les sémaphores

- Variables particulières
  - Gestion des conflits d'accès à la mémoire
  - Type : `pthread_mutex_t`
- Déroulement d'un accès à la mémoire
  - Réservation de l'accès à la mémoire
    - `pthread_mutex_lock`
  - Lecture ou écriture en mémoire
    - Doit être le plus court possible
    - Affecte le gain de parallélisation
  - Libération de l'accès à la mémoire
    - `pthread_mutex_unlock`

Exemple d'utilisation

```
/* variables globales communes et accessible à toutes les tâches */
pthread_mutex_t mu;
type data_partagee;

void *proc(void *ptrDonnees)
{ int NumeroTache = (int *)ptrDonnees;

/* réserver l'accès aux données partagées*/
  pthread_mutex_lock(&mu);
/* utilisation de data_partagée */

/* libération des droits d'accès */
  pthread_mutex_unlock(&mu);
  return NULL;
}

int main(void)
{ pthread_t pth1,pth2;

/* initialisation du sémaphore */
  pthread_mutex_init(&mu,pthread_mutexattr_default);

/* création des processus */
  :
  return 0;
}
```

Les sémaphores

- Lecture et écriture en mémoire
  - Rendre le temps de blocage le plus court possible
    - Copier dans des variables locales
    - Rendre l'accès à la mémoire
    - Utiliser la copie locale
  - Manipulations de tableaux rapides
    - memcpy
    - memset

Solution avec sauvegarde du tableau

*Tableau d'entiers **tab[nbre]***
*nombre d'éléments **nbre***

Processus 0

Recherche l'indice de la valeur 1

```
pthread_mutex_lock(&mu);
i=nbre;
memcpy(tab1,tab,
       i*sizeof(int));
pthread_mutex_unlock(&mu);

ptr=tab1;
s=0;
while (i--)
{ if (*ptr) s++;
  ptr++;
}
```

0
0
0
0
0
0
0
0
1
0
0
0
0
0

Processus 1

Permutation de valeurs

```
pthread_mutex_lock(&mu);
i=nbre;
memcpy(tab1,tab,
       i*sizeof(int));
pthread_mutex_unlock(&mu);

i=nbre*rand()/(RAND_MAX+1)
j=nbre*rand()/(RAND_MAX+1)

tmp = tab[i];
tab[i]=tab[j];
tab[j]=tmp;
```

Synchronisation de processus

- **Utilisation de variables conditionnelles**
  - **Asservissement de tâches par rapport à d'autres**
  - **Test bloquant dans un processus**
  - **Blocage levé par un autre processus**
  - **Utilisation conjoint avec un sémaphore pour l'accès mémoire**
- **Exemple**
  - **Initialisation du tableau partagé dans le processus 1**
  - **Utilisation d'un tableau partagé dans le processus 0**
    - **Le tableau doit être initialisé avant son utilisation**
    - **Le processus doit être bloqué jusqu'à la fin de l'initialisation**

Utilisation de variables conditionnelles

- **Variables conditionnelles**
  - **Type : `pthread_cond_t`**
  - **Création et initialisation**
    - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t attr);`
  - **Suppression**
    - `int pthread_cond_destroy(pthread_cond_t *cond);`
  - **Une variable booléenne partagée : `pret`**
    - **Faux (`pret=0`) état d'attente**
    - **Vrai (`pret=1`) si la condition est vérifiée**

Utilisation de variables conditionnelles

- **Fonction d'endormissement :**
  - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mu);`
    - **Attente par ce processus d'un réveil par un autre processus**
    - **Libère le sémaphore mu pendant la période de sommeil**

Utilisation de variables conditionnelles

- **Fonctions de réveil :**
  - `int pthread_cond_signal(pthread_cond_t *cond);`
    - **Réveil le thread endormi par cond**
  - `int pthread_cond_broadcast( pthread_cond_t *cond);`
    - **Réveil de tous les thread endormis par cond**
  - `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mu, struct timespec *tps);`
    - **Place le pthread courant en attente d'un signal, pendant une durée définie dans `tps`.**
    - **Libère le sémaphore `mu` toute la durée du blocage.**

Exemple

```
/* variables globales communes et accessibles à toutes les tâches */
pthread_mutex_t mu;
pthread_cond_t cond;
int pret=0; /* =faux */
type data_partagee;

void *proc(void *ptrDonnees)
{ int NumeroTache = (int *)ptrDonnees;

/* accès aux données partagées pour lire pret*/
  pthread_mutex_lock(&mu);
/* si pret est faux ( =0 )
              => libère l'accès au données
              => attent un signal de déblocage
... puis reprend les droits d'accès à la mémoire et continue */
  if (!trait_ok) pthread_cond_wait(&cond,&mu);
/* réalisation des traitements utilisant data_partagee initialisée précédement par main   */
 :
/* libération des droits d'accès */
  pthread_mutex_unlock(&mu);
  return (NULL);
}
```

Exemple (suite)

```
void main(void)
{ pthread_t pth;
/* initialisation du sémaphore */
  pthread_mutex_init(&mu,pthread_mutexattr_default);
/* initialisation de la variable de condition */
  pthread_cond_init(&cond,pthread_condattr_default);
/* création du processus asservi */
  if (pthread_create(&pth,pthread_attr_default,proc,(void *)1 )==-1)
    GestionErreurs("Erreur lors de la creation de la tâche");
/* traitement de la tache de controle */
 :
/* preparation des données partagées */
  pthread_mutex_lock(&mu);
/* initialisation de data_partagee */
 :
/* mis a jour de la variable d'état de la mémoire */
  pret = 1; /* etat vrai */
  pthread_mutex_unlock(&mu);
/* debloque la tâche en attente */
  pthread_cond_signal(&cond);
/* jonction a la fin des tâches */
  if (pthread_join(pth,&PtrResult)==-1)
    GestionErreurs("Erreur lors de la jonction avec la tâche");
}
```

Synchronisation de processus

- **Utilisation de barrières**
  - **Attente des autres processus**

Blocage des processus

- **Blocage à l'aide de barrière**
  - **Un processus maître (n° 0)**
  - **Un ensemble de processus esclave participant à la barrière (n° >0)**
  - **Le processus maître peut imposer une attente à tous les autres processus esclaves**

Opérateurs de blocage

- **Deux fonctions**
  - `pthread_barrier_checkin`
  - `pthread_barrier_checkout`
- **Règles d'utilisation**

Processus	Fonction	Action
Maître	`pthread_barrier_checkin`	Continue sans bloquer
Esclave	`pthread_barrier_checkin`	Attend l'appel de `pthread_barrier_checkin` par le maître
Maître	`pthread_barrier_checkout`	Attend l'appel de `pthread_barrier_checkout` par tous les esclaves
Esclave	`pthread_barrier_checkout`	Continue sans bloquer

Fonctions de gestion des barrières

```
int pthread_barrier_init(pthread_barrier_t *bar,
pthread_mutexattr_t attr,int taille);
```

Création de la barrière bar.
taille : nombre maximum de participants.
Retourne -1 si erreur.

Exemple de programme

- **Un exemple classique d'utilisation des barrières**
  - **Processus esclaves (tâches en parallèle).**
    1. Situation d'attente `pthread_barrier_checkin`.
    2. Traite l'ordre transmis par le processus maître
    3. Accusé de fin de traitement `pthread_barrier_checkout`
    4. Recommence 1.
  - **Le processus maître**
    1. Transmet le code du travail à réaliser (variable partagée)
    2. Libère les processus esclaves par `pthread_barrier_checkin`.
    3. Attend des accusés de réception de tous les processus esclaves `pthread_barrier_checkout`
    4. Recommence 1.

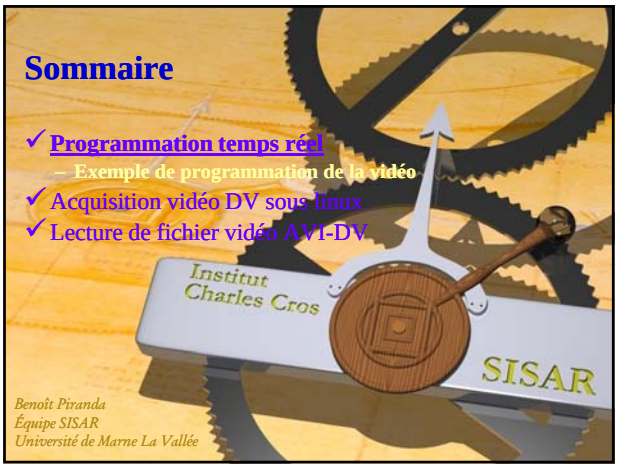
Exemple de programme (barrières)

```
#define NBRE_PROC 5
/* variables partagées */
pthread_barrier_t bar;
int ordre;

/* processus esclave participant à la barrière */
void *proc(void *ptrDonnees)
{ int NumeroTache = (int *)ptrDonnees;
  int resultat;
/* corps de la tâche (boucle infinie) */
  while (1)
  { /* attente d'un ordre du processus maitre */
    pthread_barrier_checkin(&bar,NumeroTache);
    /* traitement suivant nature de l'ordre */
    switch (ordre)
    { case TRAV1 : /* realiser le travail de type 1 */
      ...
    }
    /* accusé de reception ( fin de travail )*/
    pthread_barrier_checkout(&bar,NumeroTache);
  }
  return (NULL);
}
```

Exemple de programme (barrières)

```
void main(void)
{ pthread_t pth[NBRE_PROC];
  void *PtrResult;
  int i;
/* initialisation de la barriere pour NBRE_PROC+1 participants */
pthread_barrier_init(&bar,pthread_barrierattr_default,NBRE_PROC+1);
/* creation de NBRE_PROC pthreads ( numero 1 a NBRE_PROC ) */
  for (i=0; i<NBRE_PROC; i++)
    pthread_create(&pth[i],pthread_attr_default,proc,(void *)(i+1) )

/* traitement de la tâche de controle */
     - affectation de l'ordre de travail,
     - puis libération des processus esclaves en attente,
     - travail en parallèle de la tâche maître,
     - attente des accusés de réception. */
  for (i=PREMIER_ORDRE; i<DERNIER_ORDRE; i++)
  { ordre=i;
    pthread_barrier_checkin(&bar,0);
/* travail possible pendant l'exécution de l'ordre */
    pthread_barrier_checkout(&bar,0);
  }
}
```

Sommaire



- ✓ Programmation temps réel
  - Exemple de programmation de la vidéo
- ✓ Acquisition vidéo DV sous linux
- ✓ Lecture de fichier vidéo AVI-DV

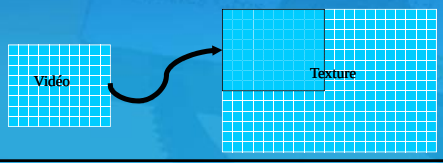
Benoît Piranda
Équipe SISAR
Université de Marne La Vallée

Programmation temps réel

- **Prise en compte du temps d'exécution dans l'écriture du programme**
  - **Utilisation de fonctions câblées**
  - **Calculs optimisés pour la vitesse d'exécution au détriment de la précision ou de la mémoire utilisé**
    - **Utilisation de tables**

Programmation temps réel

- **Un exemple : copie d'une vidéo dans une image de texture en temps réel**
  - **Acquisition d'une 'frame' vidéo**
  - **Insertion dans une image 1024x1024**
  - **Plaquage de la texture**
  - **Affichage de l'image**



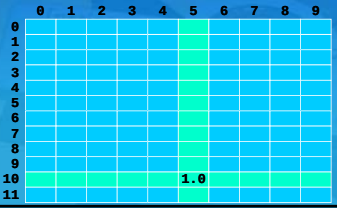
Mémorisation de l'image en mémoire

- **Utilisation d'un tableau à une seule dimension**
  - **Simplification de l'accès à une cellule**
  - **Optimisation des algorithmes de traitement**

Tableau à deux dimensions

- **En C : gestion des tableaux à deux dimensions**
  - **Tableau de pointeur sur des tableaux (lignes)**
  - **Coordonnées d'une cellule tableau[ligne][colonne]**

```
float matrice[12][10];
:
matrice[10][5]=1.0;
```



Gestions des tableaux à n dimensions

- **Un tableaux à n dimensions**
  - **Peut être mémorisé comme un tableau à 1 dimension**
    - Succession linéaire des cellules
    - Il faut définir un ordre de parcours du tableau linéaire
  - **Principe**

Soit un tableau *tab* de tailles $[n_1][n_2]\ldots[n_m]$
La cellule $tab[i_1][i_2]\ldots[i_m]$ est à la position :

$$\sum_{j=1}^{m}\left(i_j \times \prod_{k=j+1}^{m} n_k\right)$$

Par exemple pour un tableau *tab* de tailles $[n_1][n_2]$
La cellule $tab[i_1][i_2]$ est à la position : $i_1*n_2+i_2$

Gestions des tableaux à n dimensions

Par exemple pour un tableau *tab* de tailles [6][8]
La cellule $tab[i_1][i_2]$ est à la position : i_1*8+i_2

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

Tableaux et pointeurs

- **Un tableau**
  - **Un pointeur sur le premier élément**
  - **Un ensemble d'éléments contigus**

```
int *ptr,tab[20];

ptr = tab;
```
ptr pointe sur le premier élément du tableau

```
*ptr ⇔ tab[0]
```
Donc le contenu de l'élément pointé par *ptr* est le contenu du premier élément du tableau.

```
*(ptr+i) ⇔ tab[i]
```
Donc le contenu de l'élément pointé par l'adresse de *ptr+i* est le contenu de la $(i+1)^{ème}$ cellule du tableau.

Parcours d'un tableau

Exemple : compter le nombre de valeurs 1 dans un tableau

```
#define NB_ELEMENTS     10000
int i,n,tab[NB_ELEMENTS];
:
n=0;
for (i=0; i<NB_ELEMENTS; i++)
{ if (tab[i]==1) n++;
}
```

Autre solution avec des pointeurs

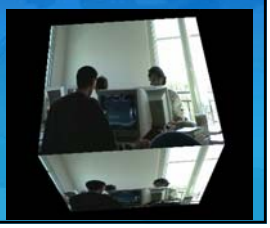
Exemple : compter le nombre de valeurs 1 dans un tableau

```
#define NB_ELEMENTS     10000
int i,n,tab[NB_ELEMENTS],*ptr;
:
n=0;
ptr = tab;
i = NB_ELEMENTS;
while (i--)
{ if (*ptr==1) n++;
  ptr++;
}
```

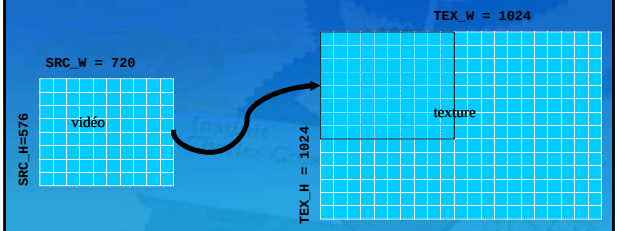
Avantages :
- Ne recalcule pas l'adresse de l'élément à chaque itération
- Simplifie le test de fin de boucle

Programmation temps réel

- **Un exemple : copie d'une vidéo dans une image de texture en temps réel**
  - **Source vidéo 720x576 tous les 25èmes de seconde**
  - **Décompression de l'image vidéo**
  - **Insertion dans une image 1024x1024**
  - **Plaquage de la texture**
  - **Affichage de la scène 3D**



Insertion d'une image dans une texture



Programmation temps réel

- Temps de traitement disponible
  - 1/25ème de seconde soit 40 ms
- Décompression logiciel d'un frame vidéo DV
  - Entre 18 et 20 ms
- Plaquage de la texture et création de l'image 3D
  - Environ 15 ms
- Reste pour l'insertion de la vidéo dans l'image
  - Environ 5 ms

Algorithme simpliste

- Utilisation de deux boucles imbriquées
- Accès absolu aux cellules des tableaux

```
int i,j;
for (i=0; i<SRC_H; i++)
{ for (j=0; j<SRC_W*3; j++)
  { texture[TEX_W*3*i+j]=image[SRC_W*3*i+j];
  }
}
```

➢11 à 12 ms par image

Utilisation des pointeurs

- Pointeur sur chaque image
  - Déplacement simultané et accès relatif

```
int i=SRC_H,j;
unsigned char *ptrTex=texture,
              *ptrImg=image;
// pour chaque ligne de l'image
while (i--)
{ j=TEX_W*3; // nombre de colonnes
  // pour chaque colonne
  while (j--) *ptrTex++ = *ptrImg++;
  // décalage à la ligne suivante
  ptrTex+=(TEX_W-SRC_W)*3;
}
```

➢9 à 10 ms par image

Détail sur les optimisations

- Accès absolu / relatif à la mémoire
  - Accès absolu :
    - Accès à une adresse mémoire avec un décalage
      - ✓ `texture[TEX_W*3*i+j]`
      - ✓ `texture+TEX_W*3*i+j`
    - Lecture de plusieurs adresses mémoires (origine + décalage)
  - Accès relatif :
    - Incrémentation par rapport à une position courante en mémoire
      - ✓ `ptrTex+=(TEX_W-SRC_W)*3`
      - ✓ `*ptrTex++`
    - Lecture d'une seule adresse mémoire

Détail sur les optimisations (suite)

- Boucles for
  - Comparaison à chaque itération
    - i < SRC_W ➔ i- SRC_W < 0 ?
  - Incrémentation en fin de traitement

```
for (i=0; i<SRC_H; i++)
{…}
```

  - Permet l'utilisation de l'indice i dans la boucle
- Boucle optimisée while
  - Comparaison à chaque itération
    - i = 0 ?
  - Décrémentation en même temps
  - Ne permet l'utilisation de l'indice i dans la boucle (ou dans l'ordre inverse)

```
i=SRC_H;
while (i--)
{…}
```

Outils de manipulation de la mémoire

- **Optimisé pour la machine :**
  - `memset((void*)dest,char val,int nbreOctets)`
  - `memcpy((void*)dest,(void*)src,int nbreOctets)`

```
int i=SRC_H;
unsigned char *ptrTex = texture,*ptrImg=image;
while (i--)
{ memcpy(ptrTex,ptrImg,SRC_W*3);
  ptrTex+=TEX_W*3;
  ptrImg+=SRC_W*3;
}
```

➢2 à 3 ms par image

Résultats en temps de calcul

- **2 Machines différentes**

Algorithme	Temps en ms PC Athlon 1.8 GHz	Temps en ms PC Pentium IV 2.1 GHz
Simpliste	11 à 12	6 à 7
Pointeur	9 à 10	5 à 6
Mémoire	2 à 3	1 à 2