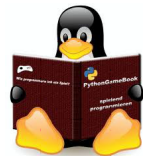


Programmation système sous Unix

Sylvain Sené



Bibliographie



- Systèmes d'exploitation :
 - A. Tanenbaum, **Systèmes d'exploitation**, Pearson Education.
- Livre de chevet sur les systèmes UNIX en général et la programmation système en particulier :
 - J.-M. Rifflet, **La programmation sous UNIX**, Ediscience.
- Livre des concepteurs du C :
 - B. W. Kernighan, D. M. Ritchie, **Le langage C : Norme ANSI**, Dunod.
- Présentation claire, complète et agréable du langage C :
 - J.-P. Braquelaire, **Méthodologie de programmation en langage C (3e édition)**, Dunod.

Plan du cours

- 1 Rappels sur le C et débogueurs
- 2 Rappels système et primitives de recouvrement
- 3 Signaux
- 4 IPC System V
 - Files de messages
 - Sémaphores
 - Segments de mémoire partagée

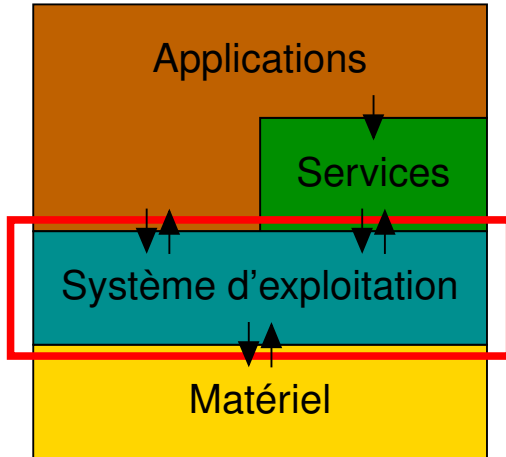
Rappels sur le C et débogueurs

Plan du cours

- 1 Rappels sur le C et débogueurs
- 2 Rappels système et primitives de recouvrement
- 3 Signaux
- 4 IPC System V
 - Files de messages
 - Sémaphores
 - Segments de mémoire partagée

Rappels sur le C et débogueurs

Systeme Unix



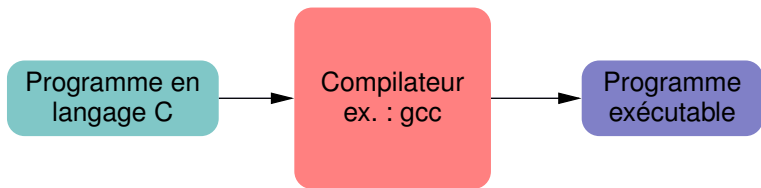
Rappels sur le C et débogueurs

Intérêt du langage C

- **Intérêt 1** : Unix est écrit en C.
- **Intérêt 2** : Langage simple à utiliser (haut niveau).
- **Intérêt 3** : Accès au système (primitives de bas niveau).
- **Intérêt 4** : Arithmétique sur les pointeurs (manipulation fine de la mémoire).
- **Intérêt 5** : Performant.

Création d'un exécutable : la compilation

- Le programmeur écrit un programme source C.
- Il le traduit ensuite pour obtenir un exécutable.
- La traduction est faite par un logiciel : le **compilateur**.



- **Note** : l'exécutable dépend de la machine sur laquelle le programmeur veut utiliser le programme.

Rappels sur le C et débogueurs

1er exemple : Hello, world !

```
/* 1er programme source, 1ère version. */  
#include <stdio.h>  
  
int main(void)  
{  
    printf("Hello world!\n");  
    exit(0);  
}
```

- **#include <stdio.h>** : inclusion d'une bibliothèque C existante, **stdio.h**, qui contient les fonctions d'entrée/sortie.
- **/* ... */** : commentaires.
- **main** : fonction principale du programme.
- **printf** : fonction demandant l'affichage d'une chaîne de caractères entre double quotes.
- **exit(0)** : sortie du programme, code retour égal à 0.

Rappels sur le C et débogueurs

1er exemple : Hello, world !

```
/* 1er programme source, 2ème version. */
```

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

```
void afficher()
```

```
{
```

```
    printf("Hello world!\n");
```

```
}
```

```
int main(void)
```

```
{
```

```
    afficher();
```

```
    exit(0);
```

```
}
```

Rappels sur le C et débogueurs

1er exemple : Hello, world !

```
/* Fichier "affichage.h". */
```

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

```
void afficher();
```

```
/* Fichier "affichage.c". */
```

```
#include "affichage.h"
```

```
void afficher()
```

```
{
```

```
    printf("Hello world!\n");
```

```
}
```

```
/* Fichier "hello.c". */
```

```
#include "affichage.h"
```

```
int main(void)
```

```
{
```

```
    afficher();
```

```
    exit(0);
```

```
}
```

Rappels sur le C et débogueurs

Compiler du C sous Linux

```
$ gcc -o HelloWorld hello.c
```

- Compile le fichier hello.c dans lequel se trouve le code source et crée l'exécutable HelloWorld.
- L'utilisation du programme nécessite le fichier exécutable et des bibliothèques (fonctions pré-compilées), mais ne nécessite plus le source C.
- Options importantes de compilation :
 - **-g** : permet d'utiliser ultérieurement un **débogueur**.
 - **-Wall** : indique des avertissements sur le code.
 - **-o** : permet de nommer le fichier exécutable.
 - **-pedantic** : avertissement en cas de C non ANSI.

Rappels sur le C et débogueurs

Création d'un exécutable

- 1 Création de fichiers source avec un éditeur, par exemple :

```
$ emacs fic.c &
```

ou

```
$ vi fic.c
```

- 2 Compilation de ces fichiers avec un compilateur, par exemple :

```
$ gcc -g -Wall -pedantic -o executable fic.c
```

- 3 Le compilateur a créé un fichier exécutable, nommé **executable**.

- 4 Exécution du fichier exécutable :

```
$ ./executable
```

Rappels sur le C et débogueurs

Création d'un exécutable

- 1 Création de fichiers source avec un éditeur, par exemple :

```
$ emacs fic.c &
```

ou

```
$ vi fic.c
```

- 2 Compilation de ces fichiers avec un compilateur, par exemple :

```
$ gcc -g -Wall -pedantic -o executable fic.c
```

- 3 Le compilateur a créé un fichier exécutable, nommé `executable`.

- 4 Exécution du fichier exécutable :

```
$ ./executable
```

Rappels sur le C et débogueurs

Création d'un exécutable

- 1 Création de fichiers source avec un éditeur, par exemple :

```
$ emacs fic.c &
```

ou

```
$ vi fic.c
```

- 2 Compilation de ces fichiers avec un compilateur, par exemple :

```
$ gcc -g -Wall -pedantic -o executable fic.c
```

- 3 Le compilateur a créé un fichier exécutable, nommé **executable**.

- 4 Exécution du fichier exécutable :

```
$ ./executable
```

Rappels sur le C et débogueurs

Création d'un exécutable

- 1 Création de fichiers source avec un éditeur, par exemple :

```
$ emacs fic.c &
```

ou

```
$ vi fic.c
```

- 2 Compilation de ces fichiers avec un compilateur, par exemple :

```
$ gcc -g -Wall -pedantic -o executable fic.c
```

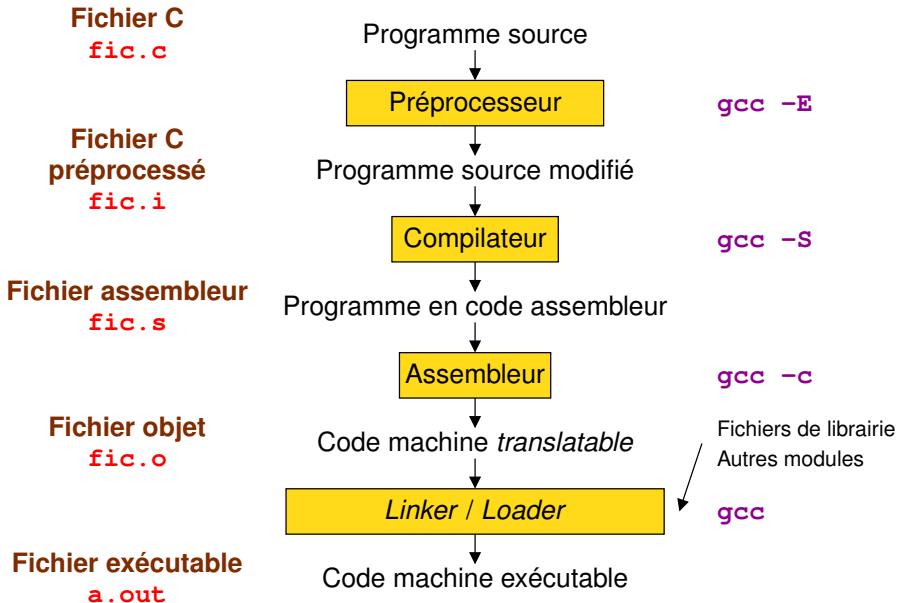
- 3 Le compilateur a créé un fichier exécutable, nommé **executable**.

- 4 Exécution du fichier exécutable :

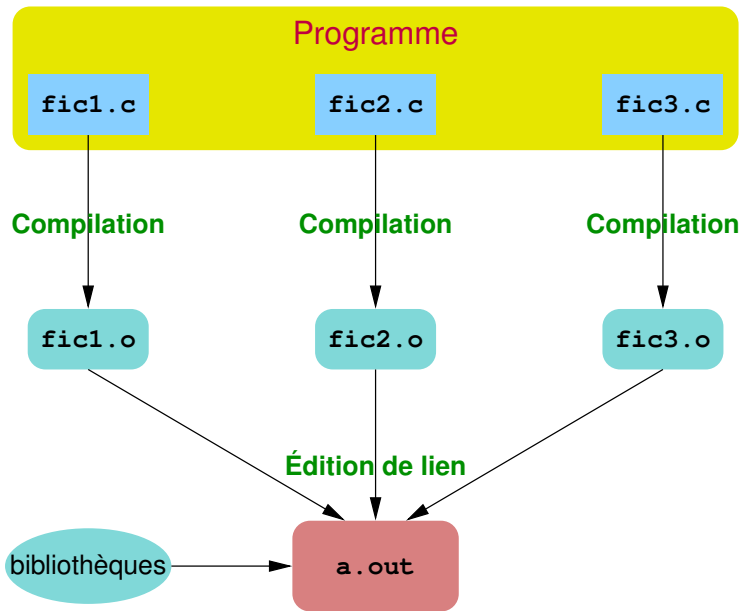
```
$ ./executable
```

Rappels sur le C et débogueurs

Étapes de la compilation



Compilation d'un programme modulaire



Rappels sur le C et débogueurs

Construction de types

- De nouveaux types peuvent être construits à partir des types de base en utilisant des constructeurs.
- Constructeurs de types :
 - `*` : pointeur
 - `[]` : tableau
 - `()` : fonction
 - **`struct`** : structure (enregistrement du Pascal)
 - **`union`** : union disjointe de types
- On peut donner un nom à un type construit avec le mot réservé **`typedef`**.

Mémorisation de plusieurs éléments de même type : les tableaux

- Déclaration d'un tableau :
 <type> <identificateur>[n1]...[nk]
 où n1, ..., nk sont des constantes.
- Le tableau a k dimensions, et les indices de la j ème dimension varient de 0 à n_j-1 .

Exemple :

```
int table[10][15];  
int i, j;  
  
for (i = 0; i<=9; i++)  
    for (j = 0; j<=14; j++)  
        table[i][j]=i+j;
```

Rappels sur le C et débogueurs

Les tableaux – exemple

```
#define N 10
main()
{
    int tab[N];
    int i;
    ...
    for (i = 0; i < N; i++)
        printf("tab[%d] = %d\n", i, tab[i]);
}
```

Un tableau correspond en fait à un **pointeur** vers le premier élément du tableau. Ce pointeur est **constant**. Cela implique qu'aucune opération globale n'est autorisée sur un tableau.

Il est par exemple **interdit** d'écrire : `tab1 = tab2;`.

Rappels sur le C et débogueurs

Les tableaux – exemple

```
#define N 10
main()
{
    int tab1[N], tab2[N];
    int i;
    ...
    for (i = 0; i < N; i++)
        tab1[i] = tab2[i];
}
```

On choisit donc de copier un tableau **élément par élément**.

Rappels sur le C et débogueurs

Les tableaux – exemple

```
#define N 4
main()
{
    int tab[N] = {1, 2, 3, 4};
    int i;
    ...
    for (i = 0; i < N; i++)
        printf("tab[%d] = = %d\n", i, tab[i]);
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes.

Rappels sur le C et débogueurs

Les tableaux – exemple

```
#define M 2
#define N 3
main()
{
    int i, j;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < N; j++)
            printf("tab[%d][%d] = %d\n", i, j,
tab[i][j]);
    }
}
```

Tableau à deux dimensions.

Rappels sur le C et débogueurs

Les chaînes de caractères

- Le compilateur mémorise les chaînes dans des tableaux de caractères, terminés par `'\0'`.

- Exemple :

La chaîne "toto" est mémorisée comme :

t	o	t	o	\0
---	---	---	---	----

- Une constante chaîne de caractères ne doit pas être modifiée (c'est une constante !).
- Le type représentant les chaînes est **char ***.

Les énumérations

- Un type énuméré contient un nombre fini de valeurs.
- Chaque valeur a un nom et est codée par un entier.
- La définition du type se fait hors des corps de fonction.

Exemple :

```
enum jour
{
    lun, mar, mer, jeu, ven, sam, dim
};
enum jour x, y;
int main(void)
{
    x = lun;
    y = x+1;
    /* y vaut mar. */
}
```

Les structures

- On peut regrouper plusieurs champs de types différents dans une variable de type `structure`.

- Syntaxe :

```
struct <nom>
{
    <suite de déclarations de champs>
};
```

- Exemple :

```
struct point
{
    double abscisse;
    double ordonnee;
};

struct point p1, p2;
```

Rappels sur le C et débogueurs

Les structures

- Les champs peuvent être de types différents.

- Exemple :

```
enum jour = {lun, mar, mer, jeu, ven, sam, dim};  
struct individu  
{  
    char        nom[25];  
    int         annee_anniversaire;  
    int         mois_anniversaire;  
    enum jour    jour_anniversaire;  
};
```

- Un champ peut lui-même être une structure, si le type structuré interne a déjà été défini.

Rappels sur le C et débogueurs

Structures – exemple

- Définition de structure et de type :

```
struct point {  
    double abscisse;  
    double ordonnee;  
};  
  
struct rotation {  
    struct point centre;  
    double      angle;  
};
```

```
typedef struct point {  
    double abscisse;  
    double ordonnee;  
} point;  
  
typedef struct rotation {  
    struct point centre;  
    double      angle;  
} rotation;
```

- Accès aux champs d'une structure, utilisation de `.` ou de `->` :

```
struct rotation r;  
  
r.angle = 5.0;
```

```
point *p;  
  
p->abscisse = 2.0;  
p->ordonnee = 3.0;
```

Rappels sur le C et débogueurs

Structures – autres exemples

```
#include <math.h>
struct complexe
{
    double reelle;
    double imaginaire;
};
main()
{
    struct complexe z;
    double norme;
    ...
    norme = sqrt(z.reelle * z.reelle + z.imaginaire *
z.imaginaire);
    printf("norme de (%f + i %f) = %f\n", z.reelle,
z.imaginaire, norme);
}
```

Rappels sur le C et débogueurs

Structures – autres exemples

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct complexe z = {2. , 2.};
```

En ANSI C, on peut appliquer l'opérateur d'affectation aux structures à la différence des tableaux). Ainsi :

```
...  
main()  
{  
    struct complexe z1, z2;  
    ...  
    z1 = z2;  
}
```

Rappels sur le C et débogueurs

Les champs de bits

- On peut regrouper au sein d'un même octet différents objets.
- **Intérêt 1** : gagner de l'espace mémoire.
- **Intérêt 2** : utiliser des objets de manière performante.

Exemple :

```
struct droits{
    unsigned int type:4;
    unsigned int bits_speciaux:3;
    unsigned int u_r:1;
    unsigned int u_w:1;
    unsigned int u_x:1;
    unsigned int g_r:1;
    ...
    unsigned int o_x:1;
};
```

Rappels sur le C et débogueurs

Les unions

- Ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire.
- Permet de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types.
- Membres de longueurs différentes \implies place réservée en mémoire = la taille du membre le plus grand.
- Même syntaxe que les structures.

Rappels sur le C et débogueurs

Unions – exemples

```
union jour
{
    char lettre;
    int numero;
};

main()
{
    union jour hier, demain;
    hier.lettre = 'J';
    printf("hier = %c\n", hier.lettre);
    hier.numero = 4;
    demain.numero = (hier.numero + 2) % 7;
    printf("demain = %d\n", demain.numero);
}
```

Définition de types composés

- Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de **typedef** :

```
typedef type synonyme;
```

Exemple :

```
struct complexe
{
    double reelle;
    double imaginaire;
};
typedef struct complexe complexe;
main()
{
    complexe z;
    ...
}
```

Rappels sur le C et débogueurs

Arguments de la fonction `main`

- Pour permettre le passage d'arguments à la fonction principale `main`, on la définit de la façon suivante :

```
int main (int argc, char  
*argv[])  
{  
    .....  
}
```

```
int main (int argc, char  
**argv)  
{  
    .....  
}
```

- **argc** : initialisé par le nombre d'arguments avec lesquels l'exécutable correspondant est lancé.
- **argv** : tableau de chaînes de caractères initialisé par les arguments avec lesquels l'exécutable correspondant est lancé.

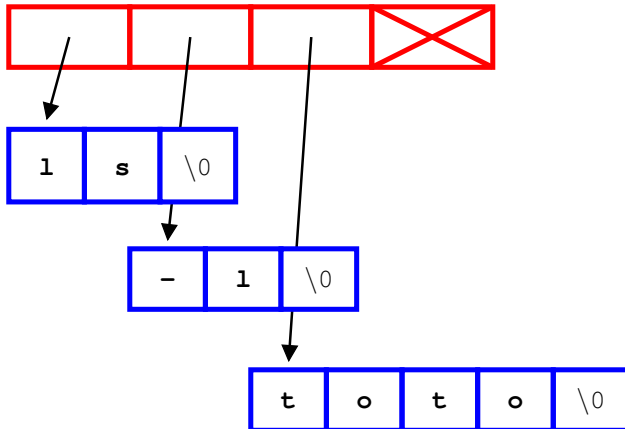
Rappels sur le C et débogueurs

Arguments de `main`

- Considérons le lancement de commande suivant :

`ls -l toto`

- `argc` est initialisé à 3
- `argv` est initialisé à :



Rappels sur le C et débogueurs

Appels de fonction : rappel

- L'appel d'une fonction provoque :
 - Les évaluations des paramètres d'appel (l'ordre de ces évaluations dépend de l'implantation).
 - Pour chaque paramètre, la création d'un argument local au cadre de la fonction appelée sur la pile d'exécution.
 - Cet argument est initialisé à la valeur calculée avant l'appel : l'appel se fait par **valeur**.
- Cependant, si on passe un tableau **t** à une fonction, cette dernière ne peut pas modifier **t** mais peut modifier **t[0]**, **t[1]**, ...

Rappels sur le C et débogueurs

Appel par valeur : exemple

```
void echange (int x, int y)
{
    int z = x;
    x = y;
    y = z;
}

int main (void)
{
    int a = 0;
    int b = 1;
    echange(a,b);
    printf ("a:%d, b:%d\n", a, b);
    return 0;
}
```

Les valeurs des variables **a** et **b** de la fonction **main** ne sont pas échangées par la fonction **echange**.

Les pointeurs

- Une variable utilisée dans un programme possède une adresse logique, à un instant donné.
- L'adresse d'une variable **x** est **&x**.
- Si le type de **x** est **t**, le type de **&x** se note ***t**.
- Une variable destinée à mémoriser des adresses est une variable de type **pointeur**.
- Le type d'une variable **p** de type pointeur dépend du type des variables dont **p** contiendra l'adresse.

- Exemples :

```
char *p;      /* pointeur de char */  
int **q;     /* pointeur de pointeur  
              d'int */
```

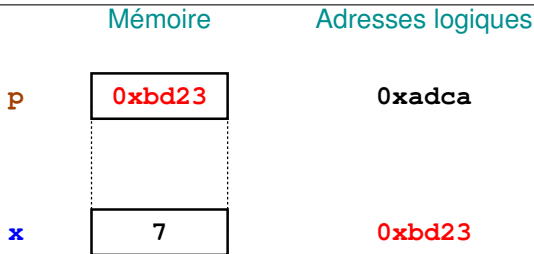
Rappels sur le C et débogueurs

Pointeurs – 1er exemple

- Pointeur **p** qui pourra contenir l'adresse d'une variable de type **short**.

```
short *p;      /* p: pointeur sur short */
short x;       /* x: variable de type short */

x = 7;         /* x: short initialisé à 7 */
p = &x;        /* p reçoit l'adresse de x */
```



Rappels sur le C et débogueurs

Emplacement pointé

- Si **p** est un pointeur de type **t ***, la donnée de **sizeof(t)** octets commençant à l'adresse **p** se note ***p**.
- ***p** référence ainsi le contenu de l'emplacement mémoire pointé par **p**.
- Dans l'exemple précédent, ***p** s'évalue en **7**.
- **IMPORTANT !!**
int *p; ne réserve de la place que pour la variable **p**, et **PAS** pour l'objet pointé par **p**.
- La séquence isolée de code suivante est donc **FAUSSE** :

```
int *p;  
*p = 3;      /* NON ! */
```

Rappels sur le C et débogueurs

Pointeurs : quelques remarques

- Les pointeurs mémorisent des adresses codées par des entiers. En pratique, on ne s'intéresse pas aux valeurs de ces entiers.
- Ces adresses logiques correspondent à la vue locale que le processus a de sa mémoire (pas à une adresse physique).
- Un nom de fonction est considéré comme un pointeur constant sur la fonction.
- Une fonction peut prendre en argument ou retourner un pointeur vers une fonction (mais pas une fonction).
- On peut déclarer des tableaux de pointeurs de fonctions (mais pas des tableaux de fonctions).

Récapitulatif sur les variables / pointeurs

- Notion de *lvalue* : tout objet pouvant être placé à gauche d'un opérateur d'affectation.
 - Une adresse, emplacement mémoire dans lequel est stocké l'objet.
 - Une valeur, ce qui est stocké dans cet emplacement.

Exemple :

```
int i, j;  
i = 3;  
j = i;
```

- Si le compilateur a placé la variable `i` à l'adresse 7654567887 en mémoire, et la variable `j` à l'adresse 7654567889, on a :

Objet	Adresse	Valeur
<code>i</code>	7654567887	3
<code>j</code>	7654567889	3

Récapitulatif sur les variables / pointeurs

- Un pointeur est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet.
- La valeur d'un pointeur est toujours un entier (éventuellement un entier long).
- Le type d'un pointeur dépend du type de l'objet vers lequel il pointe. Pour un pointeur sur un objet de type `char`, la valeur donne l'adresse de l'octet où cet objet est stocké. Pour un pointeur sur un objet de type `int`, la valeur donne l'adresse du premier des 2 octets où l'objet est stocké.

Exemple :

```
int i = 3;  
int *p;  
p = &i;
```

Objet	Adresse	Valeur
i	7654567887	3
p	7654567889	7654567887

Récapitulatif sur les variables / pointeurs

- L'opérateur unaire d'indirection ***** permet d'accéder directement à la valeur de l'objet pointé.
- Ainsi, si **p** est un pointeur vers un entier **i**, ***p** désigne la valeur de **i**.

Exemple :

```
main()
{
    int i = 3;
    int *p;
    p = &i;
    printf("*p = %d\n", *p);
}
```

Récapitulatif sur les variables / pointeurs

```
main()  
{  
    int i = 3;  
    int *p;  
    p = &i;  
    printf("*p = %d\n", *p);  
}
```

Les objets `i` et `*p` sont identiques : ils ont mêmes adresse et valeur.
Nous sommes donc dans la configuration :

Objet	Adresse	Valeur
<code>i</code>	7654567887	3
<code>p</code>	7654567889	7654567887
<code>*p</code>	7654567887	3

Rappels sur le C et débogueurs

Rappel sur l'appel de fonction

Lors de l'appel d'une fonction :

- Les paramètres d'appel sont évalués (dans un ordre dépendant de l'implantation).
- Les valeurs calculées sont empilées.
- La fonction appelée travaille sur ces valeurs empilées.
- Un appel de fonction de la forme $f(x)$ ne peut donc pas modifier la valeur de la variable x .

Fonctions et pointeurs : exemple

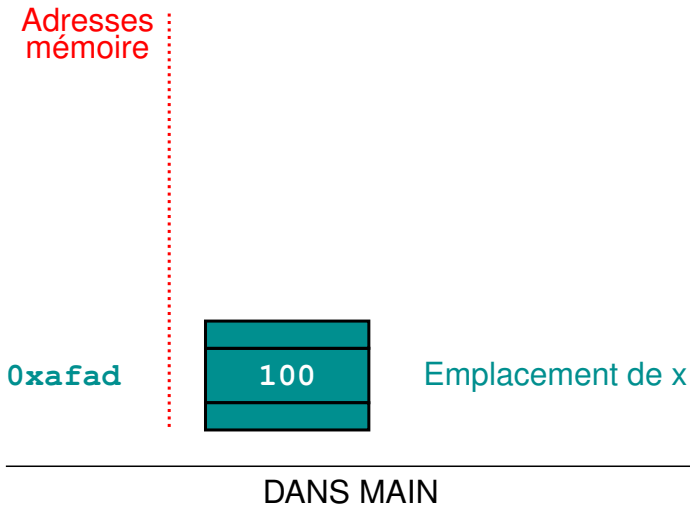
- Pour qu'une fonction puisse modifier une variable, **elle doit avoir accès à son adresse.**

```
void f(int *z)
{
    *z = 12345;
    return;
}

int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
    return 0;
}
```

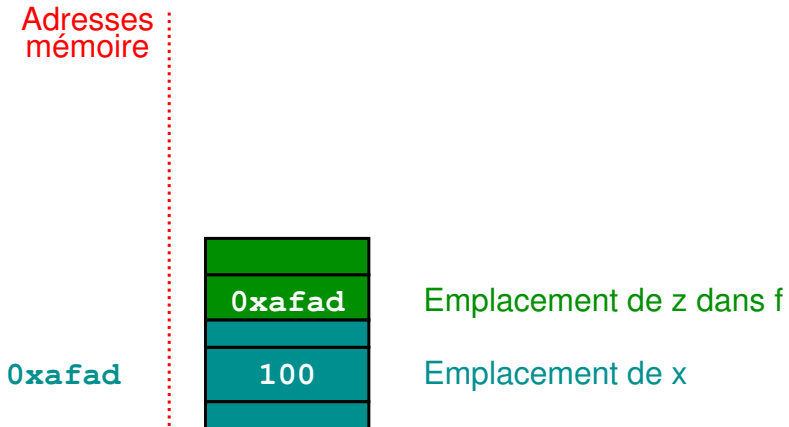

Rappels sur le C et débogueurs

Fonctions et pointeurs : exemple



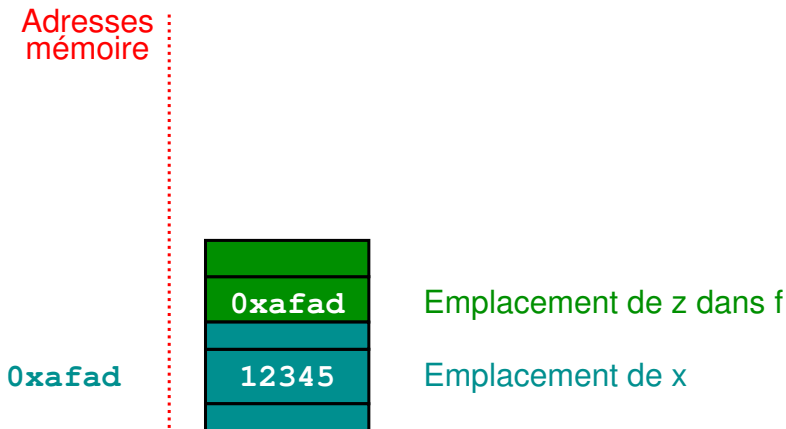
Rappels sur le C et débogueurs

Fonctions et pointeurs : exemple



Rappels sur le C et débogueurs

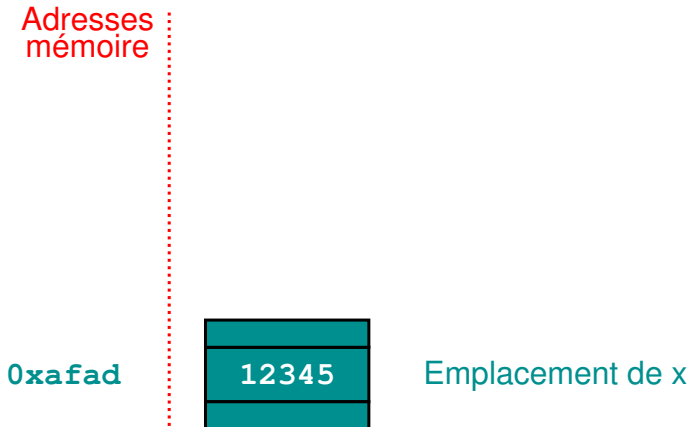
Fonctions et pointeurs : exemple



APRÈS L'INSTRUCTION $*z = 12345;$

Rappels sur le C et débogueurs

Fonctions et pointeurs : exemple



DANS MAIN, AU RETOUR DE f

Une version correcte de l'échange

```
void echange (int *x, int *y)
{
    int z = *x;
    *x = *y;
    *y = z;
}

int main (void)
{
    int a = 0;
    int b = 1;
    echange(&a, &b);
    printf("a: %d, b:%d\n", a, b);
    return 0;
}
```

Rappels sur le C et débogueurs

Pointeurs et tableaux

- Un tableau est un pointeur constant.
 - `*(p+i)` est équivalent à `p[i]`.
 - `*p` est équivalent à `p[0]`.

```
int tab[10];  
int tab2[20];  
*(tab+1) = 2;      /* OK */  
tab = tab2;        /* ILLEGAL*/
```

- **IMPORTANT !!**

`int p[10]` demande au compilateur de réserver de la place pour `p` et pour 10 entiers (contrairement à `int *p`). Il s'agit d'une allocation mémoire **statique**.

Rappels sur le C et débogueurs

Allocation dynamique

- Rappel : la définition de variable

```
int *p;
```

ne réserve pas de place en mémoire pour l'entier pointé par **p**.

- Par ailleurs, on ne peut pas définir (statiquement) des tableaux dont le nombre de cases sera connu à l'exécution.
- Il est parfois souhaitable de pouvoir demander de la mémoire à l'exécution. Une telle demande est **une allocation dynamique**.
- Des fonctions de bibliothèque standard permettent d'allouer et de désallouer de la place en mémoire **dynamiquement**.

Allocation dynamique : malloc, free

- Pour demander l'allocation d'une zone mémoire :

```
void *malloc (size_t taille);
```

où **taille** est le nombre d'octets désirés.

- Si on veut stocker en mémoire un tableau de 10 entiers, on utilisera :

```
int *tab_entiers;
```

```
tab_entiers = (int *) malloc(10*sizeof(int));
```

- Le retour d'un malloc est :

- **NULL** si échec,
- pointeur sur début de zone allouée si OK.

- Pour changer la taille d'une zone **p** déjà allouée :

```
void *realloc(void *p, size_t taille);
```

- Pour libérer une zone commençant à une adresse de début d'une zone **p** déjà allouée :

```
void free (void *p);
```


Allocation dynamique – une chaîne de n caractères

```
char *ch; int n;

...

/* n a maintenant une valeur > 0. */
if ((ch=(char *)malloc((n+1)*sizeof(char)))==NULL) {
    /* Erreur, échec de malloc. */
    fprintf(stderr, "Erreur (%s, %d)\n", __FILE__,
            __LINE__);
    perror("malloc");
    exit(EXIT_FAILURE);
}

/* À partir d'ici, ch pointe sur une zone pouvant
contenir n caractères. */

...

/* Libération de l'espace mémoire occupé par ch. */
free(ch);
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
/* Polynomes de degré 10. */  
#define DEG 10  
/* Création d'un type pour stocker les polynomes. */  
typedef struct poly {  
    int degre;  
    float *coeff;  
} poly;  
/* Déclaration de la matrice P. */  
poly ***P;  
int l, c, i, j;  
...  
/* Allocation mémoire pour P. */  
P = (poly ***) malloc(l * sizeof(poly **));  
for (i=0; i<l; i++)  
    P[i] = (poly **) malloc(c * sizeof(poly *));  
for (i=0; i<l; i++) {
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
#define DEG 10
/* Création d'un type pour stocker les polynomes. */
typedef struct poly {
    int degre;
    float *coeff;
} poly;
/* Déclaration de la matrice P. */
poly ***P;
int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

/* Création d'un type pour stocker les polynomes. */
typedef struct poly {
    int degre;
    float *coeff;
} poly;

/* Déclaration de la matrice P. */
poly ***P;
int l, c, i, j;
...

/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
typedef struct poly {  
    int degre;  
    float *coeff;  
} poly;  
/* Déclaration de la matrice P. */  
poly ***P;  
int l, c, i, j;  
...  
/* Allocation mémoire pour P. */  
P = (poly ***) malloc(l * sizeof(poly **));  
for (i=0; i<l; i++)  
    P[i] = (poly **) malloc(c * sizeof(poly *));  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        P[i][j] = (poly *) malloc(sizeof(poly));  
}
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

int degre;
float *coeff;
} poly;
/* Déclaration de la matrice P. */
poly ***P;
int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
float *coeff;
} poly;
/* Déclaration de la matrice P. */
poly ***P;
int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

} poly;
/* Déclaration de la matrice P. */
poly ***P;
int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {

```


Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

/* Déclaration de la matrice P. */
poly ***P;
int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++) {

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

poly ***P;
int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j].coeff = (float *) malloc(DEC *

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

int l, c, i, j;
...
/* Allocation mémoire pour P. */
P = (poly **) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly *) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
        sizeof(float));
}

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

...

```
/* Allocation mémoire pour P. */
```

```
P = (poly ***) malloc(l * sizeof(poly **));
```

```
for (i=0; i<l; i++)
```

```
    P[i] = (poly **) malloc(c * sizeof(poly *));
```

```
for (i=0; i<l; i++) {
```

```
    for (j=0; j<c; j++)
```

```
        P[i][j] = (poly *) malloc(sizeof(poly));
```

```
}
```

```
/* Allocation mémoire du champ coeff pour chaque  
élément de P. */
```

```
for (i=0; i<l; i++) {
```

```
    for (j=0; j<c; j++)
```

```
        P[i][j]->coeff = (float *) malloc(DEG *  
sizeof(float));
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
/* Allocation mémoire pour P. */
```

```
P = (poly **) malloc(l * sizeof(poly **));  
for (i=0; i<l; i++)  
    P[i] = (poly *) malloc(c * sizeof(poly *));  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        P[i][j] = (poly *) malloc(sizeof(poly));  
}
```

```
/* Allocation mémoire du champ coeff pour chaque  
élément de P. */
```

```
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        P[i][j]->coeff = (float *) malloc(DEG *  
sizeof(float));  
}
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

P = (poly ***) malloc(l * sizeof(poly **));
for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Réallocation mémoire du champ coeff de chaque

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

for (i=0; i<l; i++)
    P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

P[i] = (poly **) malloc(c * sizeof(poly *));
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */
for (i=0; i<l; i++) {

```


Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j] = (poly *) malloc(sizeof(poly));
}
```

/ Allocation mémoire du champ coeff pour chaque élément de P. */*

```
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
```

...

/ Désallocation mémoire du champ coeff de chaque élément de P. */*

```
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

for (j=0; j<c; j++)
    P[i][j] = (poly *) malloc(sizeof(poly));
}
/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]->coeff);
}

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
P[i][j] = (poly *) malloc(sizeof(poly));  
}  
/* Allocation mémoire du champ coeff pour chaque  
élément de P. */  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        P[i][j]->coeff = (float *) malloc(DEG *  
sizeof(float));  
}  
...  
/* Désallocation mémoire du champ coeff de chaque  
élément de P. */  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]->coeff);  
}
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
}  
/* Allocation mémoire du champ coeff pour chaque  
élément de P. */  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        P[i][j]->coeff = (float *) malloc(DEG *  
sizeof(float));  
}  
...  
/* Désallocation mémoire du champ coeff de chaque  
élément de P. */  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]->coeff);  
}  
/* Désallocation mémoire de P. */
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

/* Allocation mémoire du champ coeff pour chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]->coeff);
}
/* Désallocation mémoire de P. */
for (i=0; i<l; i++) {

```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]->coeff);
}
/* Désallocation mémoire de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]);
}

```

Allocation dynamique – une matrice de pointeurs sur
des polynomes $P_{c \times l}$

```
for (j=0; j<c; j++)
    P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]->coeff);
}
/* Désallocation mémoire de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]);
}
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```

    P[i][j]->coeff = (float *) malloc(DEG *
sizeof(float));
}
...
/* Désallocation mémoire du champ coeff de chaque
élément de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]->coeff);
}
/* Désallocation mémoire de P. */
for (i=0; i<l; i++) {
    for (j=0; j<c; j++)
        free(P[i][j]);
}
for (i=0; i<l; i++)

```


Allocation dynamique – une matrice de pointeurs sur
des polynomes $P_{c \times l}$

```
}  
...  
/* Désallocation mémoire du champ coeff de chaque  
élément de P. */  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]->coeff);  
}  
/* Désallocation mémoire de P. */  
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]);  
}  
for (i=0; i<l; i++)  
    free(P[i]);  
free(P);
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

...

```
/* Désallocation mémoire du champ coeff de chaque  
élément de P. */
```

```
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]->coeff);  
}
```

```
/* Désallocation mémoire de P. */
```

```
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]);  
}  
for (i=0; i<l; i++)  
    free(P[i]);  
free(P);
```

Allocation dynamique – une matrice de pointeurs sur des polynomes $P_{c \times l}$

```
/* Désallocation mémoire du champ coeff de chaque  
élément de P. */
```

```
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]->coeff);  
}
```

```
/* Désallocation mémoire de P. */
```

```
for (i=0; i<l; i++) {  
    for (j=0; j<c; j++)  
        free(P[i][j]);  
}  
for (i=0; i<l; i++)  
    free(P[i]);  
free(P);
```

Rappels sur le C et débogueurs

Récapitulatif des bases en C

Programme de gestion des notes `promo_notes_c`
(*en live*)

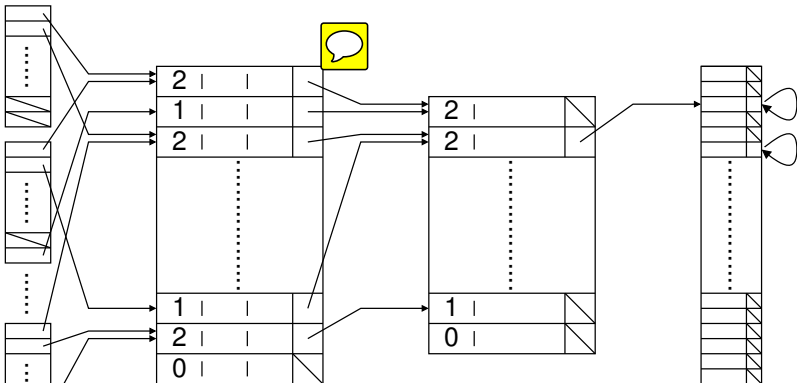
Rappels sur le C et débogueurs

Comment déboguer ?

- Méthode de base : utiliser des **traces**.
- Méthode avancée : utiliser un **débogueur** classique (Ex : gdb, ddd).
 - GDBRef.pdf
 - `http://www-rocq.inria.fr/secret/Anne.Canteaut/COURS_C/gdb.html`
- Méthode avancée (bis) : utiliser un **débogueur mémoire** (Ex : Valgrind).
 - `http://valgrind.org/docs/manual/manual.html`
 - `man valgrind`

Plan du cours

- 1 Rappels sur le C et débogueurs
- 2 Rappels système et primitives de recouvrement
- 3 Signaux
- 4 IPC System V
 - Files de messages
 - Sémaphores
 - Segments de mémoire partagée



Rappels système et primitives de recouvrement

Entrées/Sorties – tables du système

- **Table des descripteurs** : 1 par processus, descripteur = indice.
- **Table des fichiers ouverts** : nb descripteurs correspondant, mode d'ouverture, offset.
- **Table des inodes en mémoire** : nb total d'ouvertures, id disque logique, état de l'inode.
- **Table des verrous.**

Manipulation des inodes – la structure `stat`

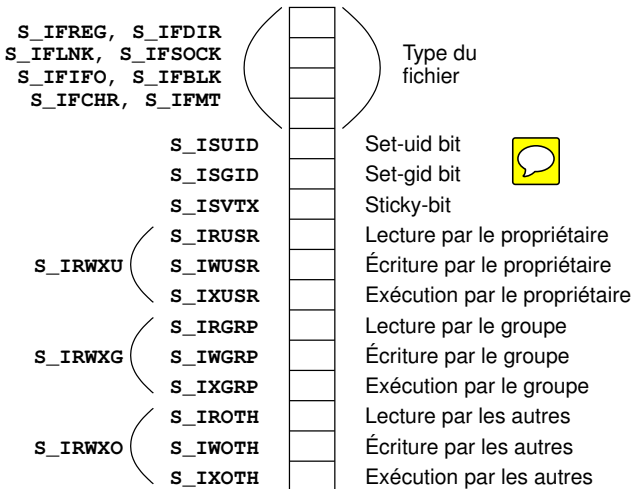
- De nombreuses caractéristiques des fichiers sont regroupées dans la structure `stat`.

```
struct stat {  
    dev_t st_dev; /* ID du disque logique */  
    ino_t st_ino; /* Numéro i-noeud sur le disque */  
    mode_t st_mode; /* Type du fichier et droits */  
    nlink_t st_nlink; /* Nb liens physiques */  
    uid_t st_uid; /* UID propriétaire */  
    gid_t st_gid; /* GID propriétaire */  
    off_t st_size; /* Taille totale en octets */  
    time_t st_atime; /* Heure dernier accès */  
    time_t st_mtime; /* Heure dernière modification */  
    time_t st_ctime; /* Heure dernier changement état */  
};
```



Manipulation des inodes – la structure `stat`

- Le champ `st_mode` définit le type du fichier (généralement sur 4 bits) et les droits d'accès au fichier (généralement sur 12 bits).



Fonctions **stat** et **fstat**

- Elles permettent d'obtenir dans un objet de structure **stat** les informations relatives à un fichier de chemin donné.
- Les prototypes de ces fonctions sont :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```




- Aucun droit particulier sur le fichier n'est nécessaire.
- Valeur retour : 0 si succès, -1 si échec.

Rappels système et primitives de recouvrement

Fonctions `stat` et `fstat` – exemple


```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char **argv) {
    struct stat st;
    if (stat(argv[1], &st) == -1) {
        perror("stat"); exit(EXIT_FAILURE);
    }
    printf("Type de fichier : ");
    switch (st.st_mode & S_IFMT) {
        case S_IFDIR: printf("répertoire\n"); break;
        case S_IFREG: printf("fichier ordinaire\n");
                    break;
        default: printf("autre!\n"); break;
    }
}
```



Rappels système et primitives de recouvrement

Manipulation de fichier

- Création d'un lien physique : **link**.
- Suppression d'un lien physique : **unlink**.
La suppression est effective si le nombre de références est nul et s'il n'y a pas d'entrée pour l'inode correspondant dans la table des fichiers ouverts.
- Renommage : **rename**. 
- Droits d'accès : **chmod**.
- Propriétaires : **chown**.

Primitives de base – open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *nom, int flags);
int open(const char *nom, int flags, mode_t mode);
```



- Permet à un processus de réaliser une ouverture de fichier.
- Allocation d'une nouvelle entrée dans la table des fichiers ouverts, chargement de l'inode en mémoire, allocation d'un descripteur.
- **Peut être bloquant** (endort le processus appelant).
- Situation bloquante : ouverture d'un tube nommé en écriture en l'absence de lecteur ou en lecture en l'absence d'écrivain.
- Flags : disjonction de bits `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_TRUNC`, `O_APPEND`, `O_CREAT`, `O_EXCL`, `O_NONBLOCK`, `O_SYNC`, `O_NDELAY`.

Primitives de base – `close`

- La primitive `close` permet à un processus de fermer un descripteur.
- Son prototype est :

```
#include <unistd.h>
int close(int fd);
```

- Effets en cascade :
 - Tous les verrous posés par le processus sur le fichier sont levés.
 - Le compteur de descripteur de l'entrée dans la table des fichiers ouverts est décrémenté.
 - Si ce compteur devient nul, l'entrée dans la table des fichiers ouverts est libérée et le compteur d'ouverture dans la table des *inodes* correspondant est alors décrémenté.
 - Si ce compteur devient nul, l'*inode* en mémoire est libéré.

Primitives de base – `read`

- La primitive `read` permet à un processus de lire le contenu d'un fichier.
- Son prototype est :

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- Son appel correspond à la demande de lecture d'au plus `count` caractères dans le fichier de descripteur `fd`.
- Les caractères lus sont stockés dans l'espace mémoire d'adresse `buf`. Ce pointeur doit donc correspondre à une zone de taille au moins égale à `count`.

Primitives de base – `read`

Algorithme de fonctionnement de `read`

- (1) Elle renvoie -1 en cas d'erreur de paramètre.
- (2) S'il n'y a pas de verrou exclusif impératif sur le fichier :
 - (a) Si la fin du fichier n'est pas atteinte, elle lit des caractères à partir de l'*offset* courant, jusqu'à ce que la fin du fichier soit atteinte ou que `count` octets aient été lus. Elle renvoie le nombre de caractères lus et la valeur de l'*offset* est augmentée de ce nombre.
 - (b) Si la fin de fichier est atteinte, elle renvoie 0.
- (3) S'il y a un verrou exclusif impératif sur le fichier :
 - (a) Si le mode de lecture est bloquant (`O_NONBLOCK` et `O_NDELAY` non positionnés), le processus est bloqué jusqu'à ce qu'il n'y ait plus de verrou ou que l'appel soit interrompu.
 - (b) Si le mode de lecture est non bloquant, aucun caractère n'est lu et la valeur retournée est -1.

Primitives de base – `write`

- La primitive `write` permet à un processus d'écrire dans un fichier.
- Son prototype est :

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- Son appel correspond à la demande d'écriture, dans le fichier de descripteur `fd`, de `count` caractères stockés à l'adresse `buf` dans l'espace d'adressage du processus.

Primitives de base – `write`

Algorithme de fonctionnement de `write`

- (1) Elle renvoie -1 en cas d'erreur de paramètre.
- (2) S'il n'y a pas de verrou impératif, exclusif ou partagé, sur le fichier, l'écriture est réalisée. Le nombre de caractères écrits est renvoyé : une valeur inférieure à `count` signale une erreur.
- (3) S'il y a un verrou impératif, exclusif ou partagé, sur le fichier :
 - (a) Si aucun des indicateurs `O_NONBLOCK` et `O_NDELAY` n'est positionné, le processus est bloqué jusqu'à ce qu'il n'y ait plus de verrou de ce type ou que l'appel soit interrompu.
 - (b) Si l'un des indicateurs précédents est positionné, le retour est immédiat sans écriture avec la valeur -1.

Les fichiers – exemple

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```



```
#define TAILLE_BUF 64

char buf[TAILLE_BUF];
int nb_lus;
int d_src, d_dest;
struct stat stat_src;
mode_t mode;
```

Les fichiers – exemple

```
int main(int argc, char **argv){
    /* Test du nombre d'arguments. */
    if (argc != 3){
        write(2, "Erreur : parametres\n", 20); exit(1);
    }

    /* Ouverture du fichier source. */
    if ((d_src=open(argv[1], O_RDONLY)) == -1){
        perror(argv[1]); exit(1);
    }

    /* Test fu fichier source (reg). */
    f_stat(d_src, &stat_src);
    if (!IS_REG(stat_src.st_mode)){
        write(2, "Fichier source non regulier\n", 28);
        exit(1);
    }
}
```

Les fichiers – exemple

```
/* Ouverture du fichier cible. */
if ((d_dest=open(argv[2],
                  O_WRONLY|O_CREAT|O_TRUNC,
                  stat_src.st_mode)) == -1) {
    perror(argv[2]); exit(1);
}


/* Lecture du fichier source et écriture dans le
   fichier cible. */
while ((nb_lus=read(d_src, buf, TAILLE_BUF)) > 0)
    if (write(d_dest, buf, nb_lus) == -1) {
        perror("Ecriture"); exit(1);
    }
if(nb_lus == -1) {
    perror("Lecture"); exit(1);
}
```

Les fichiers – exemple

```
/* Fermeture des fichiers source et cible. */  
if (fclose(d_src) == -1){  
    perror("Fermeture source");  
    exit(1);  
}  
if (fclose(d_dest) == -1){  
    perror("Fermeture cible");  
    exit(1);  
}  
exit(0);  
}
```

Les fichiers – exemple

Influence de la taille du *buffer*

TAILLE_BUF	Temps réel	Temps utilisateur	Temps système
8192	2'3	0'0	0'1
2048	2'4	0'0	0'3
1024	2'3		0'5
512	2'3	0'0	0'7
256	3'2	0'0	1'1
128	4'5	0'1	2'0
64	5'4	0'2	3'8
32	8'3	0'5	7'4
16	15'9	1'1	14'4
8	31'1	2'2	28'6
4	1"03'7	4'1	57'2
2	2"06'1	8'4	1"53'0
1	4"06'0	16'4	3"47'7

Contrôle des entrées/sorties – `fcntl`

- On peut réaliser des opérations à différents niveaux dans les tables du système avec la primitive `fcntl`.
- Son prototype est :

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

Rappels système et primitives de recouvrement

Contrôle des entrées/sorties – `fcntl`

Attributs du descripteur

- Les descripteurs possèdent des attributs enregistrés dans l'entrée correspondante de la table des descripteurs.
- Un de ces attributs est reconnu par `fcntl` : l'indication de fermeture automatique du descripteur en cas de recouvrement du processus.
- Le nom de cet attribut est `FD_CLOEXEC`.
- Par défaut, cet indicateur n'est pas positionné, ce qui maintient l'ouverture du descripteur.
- On peut changer cela en accédant et en modifiant les attributs du descripteur avec le paramètre `cmd` de `fcntl`.
 - `F_GETFD` : la valeur renvoyée par la fonction est la valeur des attributs du descripteur `fd`.
 - `F_SETFD` : utilise le paramètre `arg` pour donner une valeur aux attributs du descripteur.

Contrôle des entrées/sorties – `fcntl`

Attributs du descripteur – Exemple

```
int desc;  
int att_desc;  
  
...  
  
/* Extraction des attributs du descripteur. */  
att_desc = fcntl(desc, F_GETFD);  
/* Positionnement de l'indicateur FD_CLOEXEC. */  
att_desc = att_desc|FD_CLOEXEC;  
/* Installation des nouveaux attributs dans le  
descripteur. */  
fcntl(desc, F_SETFD, (long) att_desc);
```

Rappels système et primitives de recouvrement

Contrôle des entrées/sorties – `fcntl`

Mode d'ouverture

- À l'ouverture d'un fichier, on peut préciser le mode d'ouverture ainsi que les opérations ultérieures.
- Avec `fcntl`, on peut consulter et modifier après coup (après ouverture) ces opérations.
- Les indicateurs modifiables sont `O_APPEND`, `O_NONBLOCK`, `O_NDELAY` et `O_SYNC`.
- Les valeurs du paramètre `cmd` de la fonction utilisables pour cela sont :
 - `F_GETFL` : la valeur renvoyée par la fonction est celle des attributs du mode d'ouverture de l'entrée correspondant au descripteur dans la table des fichiers ouverts du système.
 - `F_SETFL` : utilise le paramètre `arg` pour définir de nouveaux modes d'ouverture.

Rappels système et primitives de recouvrement

Contrôle des entrées/sorties – `fcntl`

Mode d'ouverture – Exemple

```
int desc, mode_normal, mode_append; char *ref;

...

desc = open(ref, O_RDWR);

...

mode_normal = fcntl_desc(desc, F_GETFL);
mode_append = mode_normal|O_APPEND;

/* Les écritures auront lieu en fin de fichier. */
fcntl(desc, F_SETFL, mode_append);

...

/* Les écritures auront lieu en mode normal . */
fcntl(desc, F_SETFL, mode_normal);

...
```

Rappels système et primitives de recouvrement

Contrôle des entrées/sorties – `fcntl`

Verrouillage des fichiers réguliers

Les verrous constituent un mécanisme pour assurer le contrôle de la concurrence des accès à un même fichier régulier.

Caractéristiques générales :


- Un verrou est attaché à un *inode*. L'effet d'un verrou sur un fichier est donc visible par les descripteurs correspondant à cet *inode*.
- Un verrou est la propriété d'un processus : seul le processus propriétaire d'un verrou peut le modifier ou l'enlever.
- La **portée** du verrou est l'ensemble des positions dans le fichier (ensemble de valeurs de l'*offset*) auxquelles il s'applique.

Exemple : $[n_1 : n_2]$ ou $[n : \infty]$.

Rappels système et primitives de recouvrement

Contrôle des entrées/sorties – `fcntl`

Verrouillage des fichiers réguliers

- Le **type** du verrou peut être :
 - **partagé** : plusieurs verrous de ce type peuvent cohabiter, à savoir peuvent avoir des portées non disjointes.
 - **exclusif** : un verrou de  type ne peut cohabiter avec aucun autre verrou.
- Le **mode opératoire** du verrou peut être :
 - **consultatif** : pas d'action sur les E/S. Il empêche la pose de verrous incompatibles.
 - **impératif** : agit sur les E/S. Si un processus a posé :
 - un verrou partagé, tout autre processus faisant une écriture est bloqué.
 - un verrou exclusif, tout autre processus faisant une lecture ou une écriture est bloqué.

Rappels système et primitives de recouvrement

Entrées/sorties sur les répertoires

- Le fichier `dirent.h` contient la définition du type **DIR**.
- La consultation d'un répertoire suppose l'acquisition d'un pointeur sur un objet de ce type qui désigne dès lors ce répertoire.
- Le fichier `bits/dirent.h` contient la définition d'une structure **dirent** correspondant à une entrée dans un répertoire. Sur ma machine :

```
struct dirent {  
    __ino_t d_ino;  
    __off_t d_off;  
    unsigned char d_type;  
    char d_name[256];  
};
```


Rappels système et primitives de recouvrement

Entrées/sorties sur les répertoires

Les fonctions de consultation sont :



```
#include <sys/types.h>
#include <dirent.h>
/* Ouverture d'un répertoire. */
DIR *opendir(const char *nom);
/* Lecture d'un répertoire. */
struct dirent *readdir(DIR *dir);
/* Rembobinage d'un répertoire. */
void rewinddir (DIR *dir);
/* Fermeture d'un répertoire. */
int closedir(DIR *dir);
```

Rappels système et primitives de recouvrement

Entrées/sorties sur les répertoires

Les fonctions dont l'un des effets est d'écrire dans un répertoire sont :



- les fonctions de création :

- d'un **fichier** (`open` et `creat` avec l'indicateur `O_CREAT`);
- d'un **lien physique** (`link`);
- d'un **tube nommé** (`mkfifo`);
- d'un **inode** quelconque (`mknod`);
- d'un **répertoire vide** :

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```


- et les fonctions de suppression :

- d'un **lien physique** de fichier (`unlink`);
- d'un **répertoire vide** :

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Rappels système et primitives de recouvrement

Gestion des processus – rappels

- Un processus correspond à l'exécution d'un programme binaire.
- C'est un objet dynamique qui se modifie dans le temps.
- Caractérisé par son **espace d'adressage** : ensemble des instructions et données. 
- Selon les objets auxquels il accède, il peut s'exécuter en **mode utilisateur** (accès aux objets de son espace d'adressage) ou en **mode noyau** (accès aux objets externes à son espace d'adressage).
- Le passage d'un mode à l'autre se fait par des **interruptions**.

Rappels système et primitives de recouvrement

Caractéristiques d'un processus

- Obtention de l'identité d'un processus et de son père :

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

- Obtention du propriétaire réel :

```
uid_t getuid(void);
```

- Obtention du propriétaire effectif :

```
uid_t geteuid(void);
```

- Modification du propriétaire :

```
int setuid(uid_t uid);
```

Rappels système et primitives de recouvrement

Caractéristiques d'un processus

- Obtention du groupe propriétaire réel :

```
uid_t getgid(void);
```

- Obtention du groupe propriétaire effectif :

```
uid_t geteuid(void);
```

- Modification du groupe propriétaire :

```
int setgid(gid_t gid);
```

- Obtention du répertoire de travail :

```
char *getcwd(char *buf, size_t size);
```

- Modification du répertoire de travail :

```
int chdir(const char *path);
```

Rappels système et primitives de recouvrement

Caractéristiques d'un processus

- Le groupe de processus et la session auquel le processus appartient.
- La date de création : nombre de secondes depuis le 01/01/1970.
- Les temps CPU consommés dans les 2 modes (utilisateur et noyau) :

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

La structure `tms` est définie par :

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

Rappels système et primitives de recouvrement

Caractéristiques d'un processus

- Le masque de création des fichiers : cela permet de modifier les droits des fichiers créés par le processus (droits du masque enlevés à la création).

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

- La table des descripteurs : nombre limité par `OPEN_MAX` de `limits.h`.
- L'état du processus.
 - État transitoire à sa création ou au cours de la création d'un fils.
 - État prêt (éligible par l'ordonnanceur).
 - État actif.
 - État en sommeil (processus en attente d'événements).
 - État suspendu.
 - État zombi (processus terminé mais son père ne le sait pas encore).

Création de processus – primitive `fork`



- Permet de créer dynamiquement un nouveau processus s'exécutant en parallèle du processus qui l'a créé.



- Tout processus, excepté l'originel (de PID 0) est créé par un appel à `fork`.



- Un appel à `fork`, par un processus (**père**) :



```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

entraîne, si possible, la création d'un nouveau processus (**fil**), c'est-à-dire l'attribution d'un bloc de contrôle et son initialisation.

- Processus fils = copie exacte du processus père (enfin presque...) : il exécute le même programme que le père sur une copie des données de celui-ci (au moment de l'appel à `fork`).

Rappels système et primitives de recouvrement

Création de processus – primitive `fork`

- Processus fils = copie exacte du processus père (enfin presque...)
→ copie des données + copie de la pile d'exécution.
- À la reprise de l'exécution, les deux processus repartent du même point dans l'exécution.
- Comme le programme exécuté par les 2 processus est le même et que les données sont identiques, il faut, pour que leur comportement soit différent dans la suite de l'exécution, pouvoir les distinguer.
- **Code de retour différent dans le père et dans le fils :**
 - **-1** en cas d'échec (trop grand nombre de processus ouvert).
 - **0** dans le processus fils.
 - **le PID du fils** dans le processus père.

Rappels système et primitives de recouvrement

Création de processus – exemple

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv){
    pid_t pid;

    switch (pid=fork()){
        case (pid_t) -1: /* Échec de la création. */
            perror("Creation");
            exit(1);
        case (pid_t) 0: /* Création du fils. */
            printf("valeur fork = %d.\n", pid);
            printf("fils %d de pere %d.\n",
                getpid(), getppid());
            printf("fin du fils.\n");
            exit(0);
        default: /* Création du père. */
```

Rappels système et primitives de recouvrement

Création de processus – exemple

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

int main(int argc, char **argv){
    pid_t pid;

    switch (pid=fork()){
        case (pid_t) -1: /* Échec de la création. */
            perror("Creation");
            exit(1);
        case (pid_t) 0: /* Création du fils. */
            printf("valeur fork = %d.\n", pid);
            printf("fils %d de pere %d.\n",
                getpid(), getppid());
            printf("fin du fils.\n");
            exit(0);
        default: /* Création du père. */
            printf("valeur fork = %d.\n", pid);
```

Rappels système et primitives de recouvrement

Création de processus – exemple

```
#include <stdio.h>

int main(int argc, char **argv){
    pid_t pid;

    switch (pid=fork()){
        case (pid_t) -1: /* Échec de la création. */
            perror("Creation");
            exit(1);
        case (pid_t) 0: /* Création du fils. */
            printf("valeur fork = %d.\n", pid);
            printf("fils %d de pere %d.\n",
                getpid(), getppid());
            printf("fin du fils.\n");
            exit(0);
        default: /* Création du père. */
            printf("valeur fork = %d.\n", pid);
            printf("pere %d de pere %d.\n",
```

Rappels système et primitives de recouvrement

Création de processus – exemple

```
int main(int argc, char **argv){
    pid_t pid;

    switch (pid=fork()){
        case (pid_t) -1: /* Échec de la création. */
            perror("Creation");
            exit(1);
        case (pid_t) 0: /* Création du fils. */
            printf("valeur fork = %d.\n", pid);
            printf("fils %d de pere %d.\n",
                getpid(), getppid());
            printf("fin du fils.\n");
            exit(0);
        default: /* Création du père. */
            printf("valeur fork = %d.\n", pid);
            printf("pere %d de pere %d.\n",
                getpid(), getppid());
            printf("fin du pere.\n");
    }
```

Rappels système et primitives de recouvrement

Création de processus – exemple

```
pid_t pid;

switch (pid=fork()){
  case (pid_t) -1: /* Échec de la création. */
    perror("Creation");
    exit(1);
  case (pid_t) 0: /* Création du fils. */
    printf("valeur fork = %d.\n", pid);
    printf("fils %d de pere %d.\n",
           getpid(), getppid());
    printf("fin du fils.\n");
    exit(0);
  default: /* Création du père. */
    printf("valeur fork = %d.\n", pid);
    printf("pere %d de pere %d.\n",
           getpid(), getppid());
    printf("fin du pere.\n");
```

Rappels système et primitives de recouvrement

Création de processus – exemple

```
switch (pid=fork()) {  
    case (pid_t) -1: /* Échec de la création. */  
        perror("Creation");  
        exit(1);  
    case (pid_t) 0: /* Création du fils. */  
        printf("valeur fork = %d.\n", pid);  
        printf("fils %d de pere %d.\n",  
                getpid(), getppid());  
        printf("fin du fils.\n");  
        exit(0);  
    default: /* Création du père. */  
        printf("valeur fork = %d.\n", pid);  
        printf("pere %d de pere %d.\n",  
                getpid(), getppid());  
        printf("fin du pere.\n");  
}  
}
```

Rappels système et primitives de recouvrement

Création de processus – à savoir

- Les 2 processus (père et fils) partagent le même code physique.
- Le processus fils travaille sur une copie des données de son père. Ainsi, toute modification des données par l'un des processus n'est pas visible de l'autre.
- Processus exécutés en parallèle.
- Non déterminisme du résultat car événements extérieurs.
- Mécanismes de synchronisation disponibles.

Exemples d'exécution du programme précédent

Fils terminé avant l'élection du père.

Processus shell réveillé à la terminaison du père.

```
$ echo $$
```

```
7469 Identité du processus shell
```

```
$ ./creer_processus
```

```
valeur fork = 0.
```

```
fils 12364 de pere 12363.
```

```
fin du fils.
```

```
valeur fork = 12364.
```

```
fils 12363 de pere 7469.
```

```
fin du pere.
```

```
$
```

Exemples d'exécution du programme précédent

Père terminé avant le fils.

Processus shell réveillé avant la terminaison du fils.

```
$ echo $$
```

```
7469 Identité du processus shell
```

```
$ ./creer_processus
```

```
valeur fork = 0.
```

```
fils 12364 de pere 12363.
```

```
valeur fork = 12364.
```

```
fils 12363 de pere 7469.
```

```
fin du pere.
```

```
$ fin du fils.
```

Exemples d'exécution du programme précédent

Père terminé avant élection du fils.

Fils orphelin adopté par le processus `init`.

```
$ echo $$
```

```
7469 Identité du processus shell
```

```
$ ./creer_processus
```

```
valeur fork = 12364.
```

```
fils 12363 de pere 7469.
```

```
fin du pere.
```

```
$ valeur fork = 0.
```

```
fils 12364 de pere 1.
```

```
fin du fils.
```

Processus zombis

- Tout processus se terminant passe dans l'état **zombi**.
- Il y reste tant que son père n'a pas pris connaissance de sa terminaison.
- Pourquoi un tel état ? Tout processus se terminant possède une valeur de retour à laquelle son père peut accéder.
- Le système fournit un moyen à un processus d'accéder aux codes de retour de ses fils.
- Un fils qui se termine envoie le signal `SIGCHLD` à son père.
- Les seules informations conservées dans le bloc de contrôle sont :
 - son code de retour.
 - ses temps d'exécution.
 - son identité et celle de son père.



Rappels système et primitives de recouvrement

Mécanismes de synchronisation

- Ces mécanismes permettent :
 - L'élimination des processus zombies.
 - La synchronisation d'un processus sur la terminaison de ses fils après récupération des informations propres à ces terminaisons.
- La primitive **wait**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- Principe de fonctionnement :
 - (1) Pas de fils, retourne -1 ;
 - (2) Au moins un processus zombi, retourne l'identité d'un des zombis et les informations de ce zombi dans `status` (si $\neq \text{NULL}$).
 - (3) Au moins un fils mais pas de zombi, il reste bloqué jusqu'à ce que :
 - l'un des fils devienne zombi (cas (2)).
 - l'appel système soit interrompu par un signal "non mortel" (valeur de retour égale à -1).

Rappels système et primitives de recouvrement

Mécanismes de synchronisation

- La primitive **waitpid**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- Elle permet :

- de tester, en bloquant ou non le processus appelant, la terminaison d'un processus particulier (ou \in à un groupe de processus donné).
- de récupérer les informations relatives à sa terminaison à l'adresse `status`.

- Le paramètre `pid` permet de sélectionner le processus attendu :

- < -1 : tout processus fils dans le groupe `|pid|`.
- -1 : tout processus fils.
- 0 : tout processus fils du même groupe que l'appelant.
- > 0 : le processus fils d'identité `pid`.

Rappels système et primitives de recouvrement

Mécanismes de synchronisation

- La primitive **waitpid**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

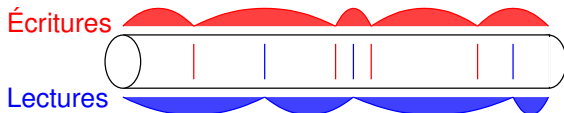
- Le paramètre `options` est une combinaison bit à bit des valeurs :
 - **WNOHANG** : le processus appelant n'est pas bloqué si le processus attendu n'est pas terminé.
 - **WUNTRACED** : si le processus attendu est stoppé et que cette information n'a pas encore été transmise, elle l'est.
- La valeur de retour est :
 - -1 en cas d'erreur.
 - 0 en cas d'échec en mode non bloquant (processus attendu existant mais ni terminé ni stoppé).
 - le numéro du processus fils zombi attendu sinon.

- L'interprétation du paramètre résultant `status` se fait en utilisant les fonctions macro-définies suivantes :
 - `WIFEXITED` : valeur non nulle si le processus attendu s'est terminé normalement.
 - `WEXITSTATUS` : fournit le code de retour du processus attendu s'il s'est terminé normalement.
 - `WIFSIGNALED` : valeur non nulle si le processus attendu s'est terminé à la réception d'un signal.
 - `WTERMSIG` : fournit le numéro du signal ayant provoqué la terminaison du processus attendu.
 - `WIFSTOPPED` : valeur non nulle si le processus attendu est stoppé (`waitpid` avec `WUNTRACED`).
 - `WSTOPSIG` : fournit le numéro du signal ayant stoppé le processus attendu.

Rappels système et primitives de recouvrement

La communication par tubes

- Deux différents types de tubes : ordinaires et nommés.
- Caractéristiques :
 - Mécanismes appartenant au SGF : `inode` \Rightarrow descripteurs, `read/write`.
 - Communication unidirectionnelle.
 - 1 tube \Rightarrow au + 2 entrées dans la table des fichiers ouverts.
 - Opération de lecture destructrice (une information peut être lue une seule fois).
 - Flot continu de caractères (principe du *streaming*) :



Rappels système et primitives de recouvrement

La communication par tubes

- Deux différents types de tubes : ordinaires et nommés.
- Caractéristiques (bis) :
 - Mode FIFO.
 - Capacité finie.
 - Deux nombres particuliers :
 - Nombre de lecteurs : nombre de descripteurs associés à l'entrée en lecture sur le tube.
 - **Attention ! si ce nombre est nul, pas d'écriture possible.**
 - Nombre d'écrivains : nombre de descripteurs associés à l'entrée en écriture sur le tube.
 - **Attention ! si ce nombre est nul, indique la EOF du tube.**

Les tubes ordinaires

- Un appel à la primitive :

```
#include <unistd.h>
int pipe(int fildes[2]);
```

crée un tube, c'est-à-dire :

- alloue un *inode* sur la disque,
 - crée deux entrées dans la table des fichiers ouverts (1 en lecture, 1 écriture),
 - alloue deux entrées dans la table des descripteurs du processus appelant.
- Le tableau `fildes` contient les descripteurs :
 - `fildes[0]` : descripteur en lecture.
 - `fildes[1]` : descripteur en écriture.



Lecture dans un tube

- Elle se fait avec la même primitive `read` déjà vue pour la lecture dans les fichiers.
- Exemple :

```
/* p[0] est un descripteur en lecture sur un  
   tube et buf est un pointeur sur une zone  
   mémoire de TAILLE_BUF caractères. */
```

```
int nb_lus;
```

```
...
```

```
nb_lus = read(p[0], buf, TAILLE_BUF);
```

```
...
```

Rappels système et primitives de recouvrement

Lecture dans un tube

Principe de fonctionnement :

- (1) **Si le tube n'est pas vide** et contient `taille` caractères, la primitive lit `nb_lus = min(taille, TAILLE_BUF)` caractères et les place à l'adresse `buf`.
- (2) **Sinon :**
 - (a) **Si le nombre d'écrivains est nul**, la EOF est atteinte et `nb_lus = 0`.
 - (b) **Sinon :**
 - **Si la lecture est bloquante** (cas par défaut), processus endormi jusqu'à ce que le tube ne soit plus vide (**attention aux interblocages !**).
 - **Sinon**, le retour est immédiat et vaut : -1 si `O_NONBLOCK` a été positionné ou 0 si `O_NDELAY` a été positionné.

Écriture dans un tube

- Elle se fait avec la même primitive `write` déjà vue pour l'écriture dans les fichiers.
- Exemple :

```
/* p[1] est un descripteur en écriture sur un  
   tube et buf est un pointeur sur  
   caractères. */
```

```
int nb_ecrits;
```

```
int n;
```

```
...
```

```
nb_ecrits = write(p[1], buf, n);
```

```
...
```

Écriture dans un tube

Principe de fonctionnement :

- (1) **Si le nombre de lecteurs est nul**, le signal `SIG_PIPE` est délivré au processus écrivain, qui s'arrête.
- (2) **Sinon :**
 - (a) **Si l'écriture est bloquante** (indicateurs `O_NONBLOCK` et `O_NDELAY` non positionnés), le retour de la primitive n'est fait que lorsque les `n` caractères ont été écrits (possibilité pour le processus de s'endormir dans l'attente que le tube se vide).
 - (b) **Sinon :**
 - **Si $n > \text{PIPE_BUF}$** , le retour est un nombre $< n$.
 - **Si $n \leq \text{PIPE_BUF}$ et s'il y a au - n emplacements libres dans le tube**, l'écriture est réalisée et `n` est renvoyé.
 - **Si $n \leq \text{PIPE_BUF}$ et s'il y a - de n emplacements libres dans le tube**, le retour est immédiat avec la valeur 0 ou -1 en fonction de l'indicateur.

Écriture dans un tube sans lecteur

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

/* Gestion du signal SIG_PIPE. */
struct sigaction action;
void hand_sigpipe(int sig){
    printf("Signal SIGPIPE reçu.\n");
}

int main(int argc, char **argv){
    int nb_ecrit, p[2];
    /* Prévision de la capture de SIGPIPE. */;
    action.sa_handler = hand_sigpipe;
    sigaction(SIGPIPE, &action, NULL);
    /* Création du tube. */
    pipe(p);
    /* Fermeture du descripteur en lecture. */
```


Rappels système et primitives de recouvrement

Écriture dans un tube sans lecteur

```
/* Gestion du signal SIG_PIPE. */
struct sigaction action;
void hand_sigpipe(int sig){
    printf("Signal SIGPIPE recu.\n");
}

int main(int argc, char **argv){
    int nb_ecrit, p[2];
    /* Prévision de la capture de SIGPIPE. */;
    action.sa_handler = hand_sigpipe;
    sigaction(SIGPIPE, &action, NULL);
    /* Création du tube. */
    pipe(p);
    /* Fermeture du descripteur en lecture. */
    if ((nb_ecrits=write(p1, "AAA", 3)) == -1)
        perror("Write");
    else
        printf("Retour du write: %d\n", nb_ecrits);
}
```

Écriture dans un tube sans lecteur

```
int main(int argc, char **argv){
    int nb_ecrit, p[2];
    /* Prévision de la capture de SIGPIPE. */;
    action.sa_handler = hand_sigpipe;
    sigaction(SIGPIPE, &action, NULL);
    /* Création du tube. */
    pipe(p);
    /* Fermeture du descripteur en lecture. */
    if ((nb_ecrits=write(p1, "AAA", 3)) == -1)
        perror("Write");
    else
        printf("Retour du write: %d\n", nb_ecrits);
}
```

```
$ ./ecrivain_sans_lecteur
```

```
Signal SIGPIPE reçu.
```

```
Write: Broken pipe
```

```
$
```

Les tubes nommés

- Ils permettent à des processus sans lien de parenté direct dans le système de communiquer en `streaming`.
- Ils possèdent toutes les caractéristiques des tubes données précédemment.
- Propriété supplémentaire : ils possèdent des (références) dans le système de gestion de fichier.
 - Tout processus connaissant la référence d'un tube peut l'utiliser pour communiquer avec d'autres processus.
 - On s'aperçoit de leur existence dans le SGF grâce à la lettre `p` renvoyé par la commande `ls -l`.

Tubes nommés – création

- On peut créer un tube nommé en utilisant la primitive `mkfifo`.
- Prototype :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);
```

- Exemple : Création de tubes nommés.

Tubes nommés – création

Création d'un tube par le bash.

```
$ mkfifo fifo1
```

```
$ ls -l fifo1
```

```
prw-r--r-- 1 ssene dynamic 0 2010-04-11 1627 fifo1
```

```
/* Programme de création de tubes. */
```

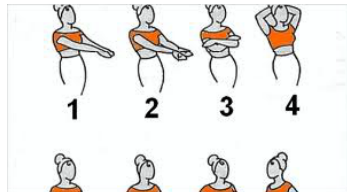
```
#include <stdio.h>
```

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

Tubes nommés – création

```
int main(int argc, char **argv){  
    /* Mis en place des droits. */  
    mode_t mode;  
    mode = S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH;  
    /* Création d'un tube nommé fifo1. */  
    if (mkfifo("fifo1", mode) == -1)  
        perror("mkfifo(fifo1)");  
    else  
        printf("fifo1 cree.\n");  
    /* Création d'un tube nommé fifo2. */  
    if (mkfifo("fifo2", mode) == -1)  
        perror("mkfifo(fifo2)");  
    else  
        printf("fifo2 cree.\n");  
}
```



Tubes nommés – création

Exécution du programme.

\$ creer_fifo

mkfifo(fifo1): File exists


fifo2 cree.

\$ ls -l fifo2

prw-r--r-- 1 ssene dynamic 0 2010-04-11 1628 fifo2

Rappels système et primitives de recouvrement

Primitives de recouvrement

- Ensemble de primitives permettant à un processus de charger en mémoire un nouveau programme binaire.
- **Idée** : Vous avez un programme écrit en C, **prog.c**. Ce programme doit faire s'exécuter une commande externe **cmde**.
 - Solution naïve : utiliser la  tion **system**.
 - **Exemples** :

```
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ cat prog1.c
#include <stdlib.h>
int main(int argc, char **argv) {
    system("ls -l"); exit(0);
}
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ ./prog1
total 28
-rwxr-xr-x 1 ssene ssene 7176 2010-09-22 10:44 prog
-rwxr-xr-x 1 ssene ssene 7177 2010-09-22 10:54 prog1
-rw-r--r-- 1 ssene ssene 86 2010-09-22 10:53 prog1.c
-rw-r--r-- 1 ssene ssene 86 2010-09-22 10:47 prog.c
-rw-r--r-- 1 ssene ssene 60 2010-09-22 10:43 prog.c~
```


Rappels système et primitives de recouvrement

Primitives de recouvrement

- Ensemble de primitives permettant à un processus de charger en mémoire un nouveau programme binaire.
- **Idée** : Vous avez un programme écrit en C, **prog.c**. Ce programme doit faire s'exécuter une commande externe **cmde**.
 - Solution naïve : utiliser la fonction **system**.
 - **Exemples** :

```
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ cat prog.c
#include <stdlib.h>
int main(int argc, char **argv) {
    system("ps -l"); exit(0);
}
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ ps -l
F S   UID    PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000    3529  3260  0  80   0 - 1496 wait   pts/1        00:00:00 bash
0 S   1000    3620  3529  0  80   0 - 14328 poll_s pts/1        00:00:02 emacs
0 R   1000    3655  3529  0  80   0 - 626 -      pts/1        00:00:00 ps
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ ./prog
F S   UID    PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000    3529  3260  0  80   0 - 1496 wait   pts/1        00:00:00 bash
0 S   1000    3620  3529  0  80   0 - 14328 poll_s pts/1        00:00:02 emacs
0 S   1000    3656  3529  0  80   0 - 404 wait   pts/1        00:00:00 prog
0 S   1000    3657  3656  0  80   0 - 458 wait   pts/1        00:00:00 sh
0 R   1000    3658  3657  0  80   0 - 626 -      pts/1        00:00:00 ps
```

Rappels système et primitives de recouvrement

Primitives de recouvrement

- Ensemble de primitives permettant à un processus de charger en mémoire un nouveau programme binaire.
- **Idée** : Vous avez un programme écrit en C, **prog.c**. Ce programme doit faire s'exécuter une commande externe **cmde**.
 - Solution naïve : utiliser la fonction **system**.
 - **Exemples** :

```
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ cat prog2.c
#include <stdlib.h>
int main(int argc, char **argv) {
    system("./prog2"); exit(0);
}
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ ./prog2 &
[2] 19300
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ ps -l > res_prog2
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ head -n 10 res_prog2
F S   UID    PID  PPID  C  PRI  NI ADDR  SZ  WCHAN  TTY          TIME CMD
0 S   1000    300 32767 0  80   0 -   404 wait  pts/1    00:00:00 prog2
0 S   1000    301   300 0  80   0 -   458 wait  pts/1    00:00:00 sh
0 S   1000    302   301 0  80   0 -   404 wait  pts/1    00:00:00 prog2
0 S   1000    303   302 0  80   0 -   458 wait  pts/1    00:00:00 sh
0 S   1000    304   303 0  80   0 -   404 wait  pts/1    00:00:00 prog2
0 S   1000    305   304 0  80   0 -   458 wait  pts/1    00:00:00 sh
0 S   1000    306   305 0  80   0 -   404 wait  pts/1    00:00:00 prog2
0 S   1000    307   306 0  80   0 -   458 wait  pts/1    00:00:00 sh
0 S   1000    308   307 0  80   0 -   404 wait  pts/1    00:00:00 prog2
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/recouvrement$ sh: Cannot fork
```

Primitives de recouvrement

- Ensemble de primitives permettant à un processus de charger en mémoire un nouveau programme binaire.
- **Idée** : Vous avez un programme écrit en C, **prog.c**. Ce programme doit faire s'exécuter une commande externe **cmde**.
 - Solution naïve : utiliser la fonction **system**.
 - **Problème** : Mécanisme coûteux en création de shells intermédiaires
 - Ce n'est pas le mécanisme à employer si l'on veut être propre du point de vue de la programmation système.
 - **On privilégiera les primitives de recouvrement.**

Primitives de recouvrement

- Ensemble de primitives permettant à un processus de charger en mémoire un nouveau programme binaire.
- **Idée** : Vous avez un programme écrit en C, **prog.c**. Ce programme doit faire s'exécuter une commande externe **cmde**.
- **Principe général** :
 - Création d'un nouveau processus.
 - Chargement d'un nouveau contexte dans ce processus.
 - Écrasement de l'espace d'adressage : un nouveau processus est chargé et s'exécute sur de nouvelles données.
 - **Pas de retour d'un recouvrement (sauf en cas d'échec).**
 - **Pas de création d'un nouveau processus au cours d'un recouvrement.**

Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

```
int main(int argc, char **argv, char **arge);
```

- `argc` est le **nombre d'arguments** de la commande.
- `argv` est le **tableau des arguments** de la commande.
- `arge` est un **tableau de chaînes de caractères** permettant d'accéder à l'environnement dans lequel le processus s'exécute.

Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

```
int main(int argc, char **argv, char **arge);
```

- `argc` est le **nombre d'arguments** de la commande.
- `argv` est le **tableau des arguments** de la commande.
- `arge` est un **tableau de chaînes de caractères** permettant d'accéder à l'environnement dans lequel le processus s'exécute.

Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

```
int main(int argc, char **argv, char **arge);
```

- `argc` est le **nombre d'arguments** de la commande.
- `argv` est le **tableau des arguments** de la commande.
- `arge` est un **tableau de chaînes de caractères** permettant d'accéder à l'environnement dans lequel le processus s'exécute.

Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

```
int main(int argc, char **argv, char **arge);
```

- `argc` est le **nombre d'arguments** de la commande.
- `argv` est le **tableau des arguments** de la commande.
- `arge` est un **tableau de chaînes de caractères** permettant d'accéder à l'environnement dans lequel le processus s'exécute.



Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

```
int main(int argc, char **argv, char **arge);
```

- **Exemple** : Considérons la séquence `bash` suivante :

```
$ env
```

```
TERM=xterm
```

```
SHELL=/bin/bash
```

```
DESKTOP_SESSION=gnome
```

```
PATH=/bin:/sbin:/usr/local/bin
```

```
LOGNAME=ssene
```

```
...
```

```
$ cp -r ~/tmp/* /home/ssene/cours/
```

Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

```
int main(int argc, char **argv, char **arge);
```

- **Exemple** : Considérons la séquence `bash` suivante :

```
$ env
```

```
TERM=xterm
```

```
SHELL=/bin/bash
```

```
DESKTOP_SESSION=gnome
```

```
PATH=/bin:/sbin:/usr/local/bin
```

```
LOGNAME=ssene
```

```
...
```

```
$ cp -r ~/tmp/* /home/ssene/cours/
```

- **Comment ceci est représenté ?**

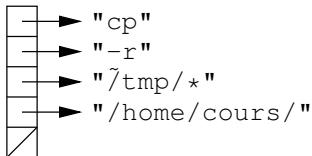
Primitives de recouvrement

- Forme générale de définition d'une procédure principale :

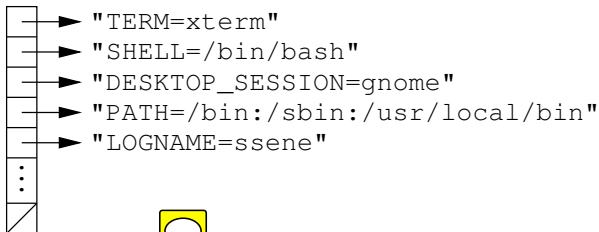
```
int main(int argc, char **argv, char **arge);
```

argc = 4

argv =



arge =



- Lecture de l'environnement :

```
char *getenv(const char *name);
```

- Modification de l'environnement :

```
int putenv(char *string);
```

Primitives de recouvrement

- **argc**, **argv**, **arge** sont les informations passées aux **6 primitives de recouvrement** qui se distinguent selon **3 différents critères**.

❶ **la manière dont les arguments sont récupérés via argv :**

- tableau contenant les arguments : primitives `execv`, `execvp`, `execve` (nombre fixe de paramètres)
- liste des arguments : primitives `execl`, `execlp`, `execle` (nombre variable de paramètres).



Primitives de recouvrement

- **argc**, **argv**, **arge** sont les informations passées aux **6 primitives de recouvrement** qui se distinguent selon **3 différents critères**.

❶ **la manière dont les arguments sont récupérés via argv :**

- tableau contenant les arguments : primitives `execv`, `execvp`, `execve` (nombre fixe de paramètres)
- liste des arguments : primitives `execl`, `execlp`, `execle` (nombre variable de paramètres).

❷ **la manière dont le fichier à charger est recherché dans le SGF :**

- fichier à charger depuis les répertoires du `PATH` de l'environnement : `execvp`, `execlp`.
- fichier à charger depuis le répertoire de travail du processus : `execv`, `execve`, `execl`, `execle`.

Rappels système et primitives de recouvrement

Primitives de recouvrement

- **argc**, **argv**, **arge** sont les informations passées aux **6 primitives de recouvrement** qui se distinguent selon **3 différents critères**.

1 la manière dont les arguments sont récupérés via **argv** :

- tableau contenant les arguments : primitives `execv`, `execvp`, `execve` (nombre fixe de paramètres)
- liste des arguments : primitives `execl`, `execlp`, `execle` (nombre variable de paramètres).



2 la manière dont le fichier à charger est recherché dans le **SGF** :

- fichier à charger depuis les répertoires du `PATH` de l'environnement : `execvp`, `execlp`.
- fichier à charger depuis le répertoire de travail du processus : `execv`, `execve`, `execl`, `execle`.

3 la conservation de l'environnement ou non :

- chargement d'un nouvel environnement : `execve`, `execle`.
- conservation de l'environnement du processus courant : `execv`, `execvp`, `execl`, `execlp`.

Exemples d'utilisation des primitives de recouvrement

Passage par liste

- `int execl(const char *path, const char *arg, ...);`
 - Le fichier de nom `path` est chargé. La fonction `main` est exécutée avec comme arguments ceux spécifiés dans la liste.
 - Exemple :
`execl("/bin/ls", "ls", "1", "/home/ssene");`
- `int execlp(const char *file, const char *arg, ...);`
 - Même comportement sauf que, si `file` est une référence relative, le fichier associé est recherché dans les répertoires du `PATH` puis chargé.
 - Exemple :
`execlp("ls", "ls", "-l", "/home/ssene");`

Exemples d'utilisation des primitives de recouvrement

Passage par liste



- `int execl(const char *path, const char *arg, ..., char * const envp[]);`
 - Même comportement sauf qu'un nouvel environnement est substitué à l'ancien (`envp`).
 - Pour quoi faire ? → changer de `PATH`.
 - Exemple :
`execl("ls", "ls", "-l", NULL, newenv);`

(en *live*)
(`prog4a.c` et `prog4b.c`)

Exemples d'utilisation des primitives de recouvrement

Passage par vecteur



- `int execev(const char *path, const char *argv[]);`
 - Le fichier de nom `path` est chargé. La fonction `main` est exécutée avec comme arguments ceux spécifiés dans le vecteur.
 - Exemple :

```
execev("/bin/ls", {"ls", "-l", "/home/ssene", NULL});
```
- `int execev(const char *file, const char *argv[]);`
 - Même comportement sauf que, si `file` est une référence relative, le fichier associé est recherché dans les répertoires du `PATH` puis chargé.
 - Exemple :

```
execev("ls", {"ls", "-l", "/home/ssene", NULL});
```

Exemples d'utilisation des primitives de recouvrement



Passage par vecteur

- `int execve(const char *filename, const char *argv[], const char *envp[]);`
 - Même comportement sauf qu'un nouvel environnement est substitué à l'ancien (`envp`).
 - Exemple :
`execve("ls", {"ls", "-l", NULL}, newenv);`

(en *live*)
(prog5a.c et prog5b.c)

Recouvrement de processus : un dernier exemple

(*en live*)
(prog6.c)

Signaux

Plan du cours

- 1 Rappels sur le C et débogueurs
- 2 Rappels système et primitives de recouvrement
- 3 Signaux**
- 4 IPC System V
 - Files de messages
 - Sémaphores
 - Segments de mémoire partagée

Signaux

Introduction



- Problèmes historiques
 - Portabilité entre les différentes versions d'Unix.
 - Arrivée de la norme Posix en 1988 qui a standardisé l'utilisation des signaux.
 - BSD (*Berkeley Software Distribution*).
- Fiabilité actuelle.
- Possible développements d'applications à base de signaux.

Signaux

Éléments préliminaires

- Mécanisme déjà rencontré l'année dernière :
 - À la frappe de certains caractères spécifiques dans un terminal, l'ensemble des processus en premier plan de la session en sont informés par des signaux (e.g., SIGINT (Ctrl-C)).
 - Une violation mémoire par un processus provoque l'envoi à celui du signal SIGSEGV (i.e., erreur de segmentation).
- Sonneries susceptibles d'être entendues.
 - Chaque type de sonnerie signale un événement particulier.
 - Chaque type de signal signale l'occurrence d'un événement.
 - Signal transmis au(x) processus concerné(s).

Signaux

Définitions

Définition

Un **signal pendant** est un signal qui a été envoyé à un processus mais qui n'a pas encore été pris en compte.

$n \quad n - 1$			2 1 0			
<div>0 1</div>	<div>0 1</div>		<div>0 1</div>	<div>0 1</div>	<div>0 1</div>	signaux pendants
<div>0 1</div>	<div>0 1</div>		<div>0 1</div>	<div>0 1</div>	<div>0 1</div>	signaux bloqués
↓	↓		↓	↓	↓	comportements

- Le bit correspondant de l'indicateur des signaux pendants vaut 1.
- Le champ de la structure qui mémorise les signaux pendants est un vecteur de bits.
 - **Conséquence** : un seul exemplaire d'un même signal peut être pendant.

Définition

*Un **signal délivré** à un processus est un signal qui est pris en compte par le processus pendant son exécution.*

- La délivrance a lieu en des circonstances particulières :
 - **Passage de l'état "ACTIF NOYAU" à l'état "ACTIF UTILISATEUR"**.
 - Ce changement d'état n'est pas contrôlé par le processus.
- À la délivrance du signal, le bit correspondant (pendant) bascule à 0.
- Si un exemplaire d'un signal arrive alors qu'il y en existe déjà un exemplaire pendant, le signal est perdu.

Signaux

Définitions

- La prise en compte d'un signal entraîne l'exécution d'une fonction particulière : le **HANDLER** qui est un des éléments fourni par l'indicateur de comportement.
- Un processus peut différer volontairement la délivrance de certains types de signaux.
 - On parle alors de **signaux bloqués**.
 - Ces signaux particuliers sont donnés dans le 2ème champ de la structure qui est aussi un vecteur de bits.
 - Le comportement contient alors un deuxième champ qui correspond à un masque installé automatiquement pour la durée d'exécution du *handler*.
- **Quand on joue avec les signaux, on ne fait qu'agir sur la structure.**

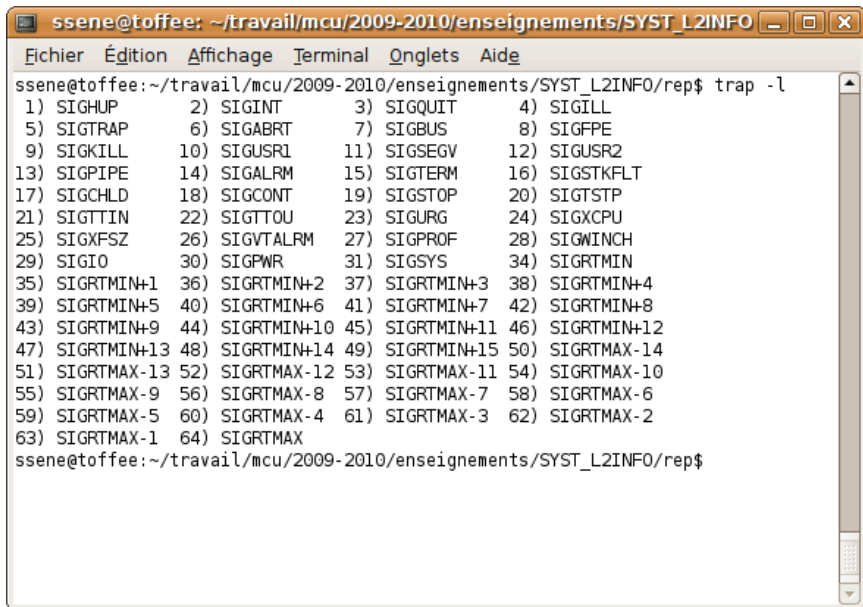
Signaux

Petits rappels

- Les processus peuvent être interrompus par la frappe de caractères particuliers sur ce terminal (p. ex. `ctrl-c` ou `ctrl-\`).
- L'utilisation de ces caractères spéciaux permettent l'envoi par le système de signaux SIGINT et SIGQUIT aux processus.
- La liste des signaux est donnée par la commande `trap -l`. La correspondance de certains avec les caractères spéciaux est donnée par la commande `stty -a`.

Signaux

Petits rappels

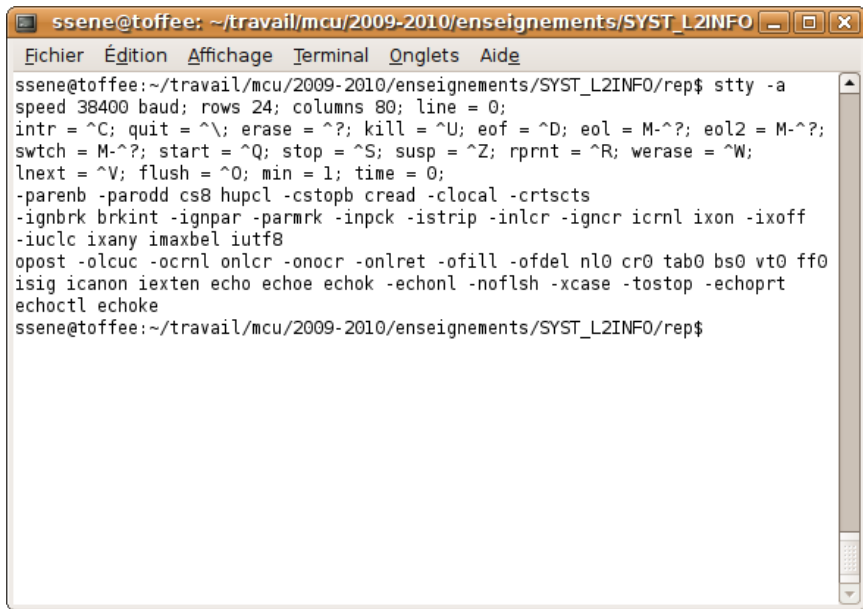


The screenshot shows a terminal window titled "ssene@toffee: ~/travail/mcu/2009-2010/enseignements/SYST_L2INFO". The window has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The terminal content shows the command "trap -l" being executed, which lists 64 signals in a 4x16 grid. The signals are: 1) SIGHUP, 2) SIGINT, 3) SIGQUIT, 4) SIGILL, 5) SIGTRAP, 6) SIGABRT, 7) SIGBUS, 8) SIGFPE, 9) SIGKILL, 10) SIGUSR1, 11) SIGSEGV, 12) SIGUSR2, 13) SIGPIPE, 14) SIGALRM, 15) SIGTERM, 16) SIGSTKFLT, 17) SIGCHLD, 18) SIGCONT, 19) SIGSTOP, 20) SIGTSTP, 21) SIGTTIN, 22) SIGTTOU, 23) SIGURG, 24) SIGXCPU, 25) SIGXFSZ, 26) SIGVTALRM, 27) SIGPROF, 28) SIGWINCH, 29) SIGIO, 30) SIGPWR, 31) SIGSYS, 34) SIGRTMIN, 35) SIGRTMIN+1, 36) SIGRTMIN+2, 37) SIGRTMIN+3, 38) SIGRTMIN+4, 39) SIGRTMIN+5, 40) SIGRTMIN+6, 41) SIGRTMIN+7, 42) SIGRTMIN+8, 43) SIGRTMIN+9, 44) SIGRTMIN+10, 45) SIGRTMIN+11, 46) SIGRTMIN+12, 47) SIGRTMIN+13, 48) SIGRTMIN+14, 49) SIGRTMIN+15, 50) SIGRTMAX-14, 51) SIGRTMAX-13, 52) SIGRTMAX-12, 53) SIGRTMAX-11, 54) SIGRTMAX-10, 55) SIGRTMAX-9, 56) SIGRTMAX-8, 57) SIGRTMAX-7, 58) SIGRTMAX-6, 59) SIGRTMAX-5, 60) SIGRTMAX-4, 61) SIGRTMAX-3, 62) SIGRTMAX-2, 63) SIGRTMAX-1, 64) SIGRTMAX. The prompt "ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep\$" is shown at the bottom.

```
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$ trap -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
5) SIGTRAP     6) SIGABRT     7) SIGBUS      8) SIGFPE
9) SIGKILL     10) SIGUSR1    11) SIGSEGV    12) SIGUSR2
13) SIGPIPE    14) SIGALRM    15) SIGTERM    16) SIGSTKFLT
17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU
25) SIGXFSZ    26) SIGVTALRM  27) SIGPROF    28) SIGWINCH
29) SIGIO      30) SIGPWR     31) SIGSYS     34) SIGRTMIN
35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4
39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$
```

Signaux

Petits rappels



The screenshot shows a terminal window with the title bar "ssene@toffee: ~/travail/mcu/2009-2010/enseignements/SYST_L2INFO". The menu bar includes "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The terminal content displays the output of the command "stty -a", showing various terminal settings such as speed, baud rate, and control characters. The prompt "ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep\$" is visible at the bottom of the terminal.

```
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?;
swtch = M-^?; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc ixany imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprtr
echoctl echoke
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$
```

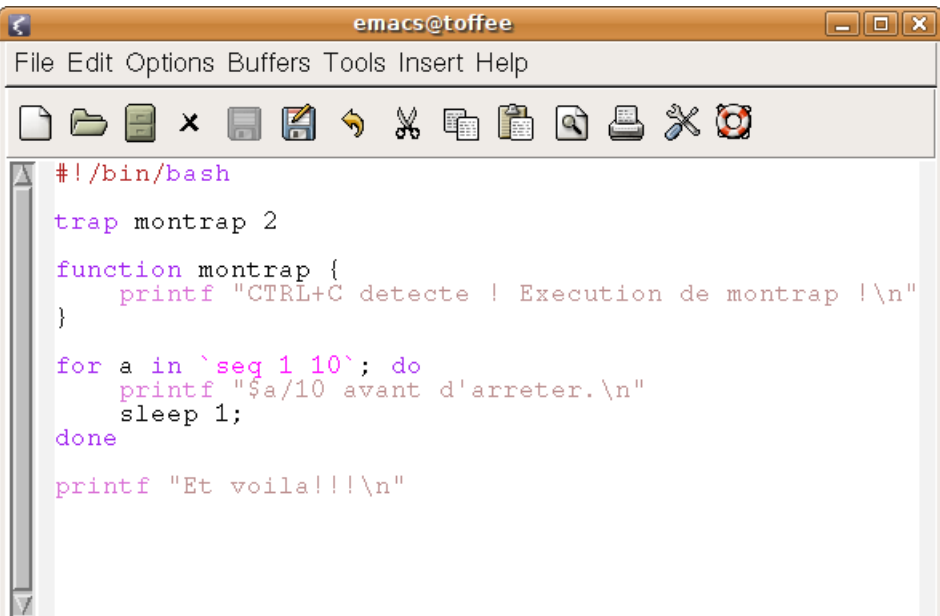
Signaux

Petits rappels

- Certains événements peuvent entraîner **l'émission d'un signal particulier**.
- La commande `trap` permet à un processus de spécifier quel type de comportement adopter quand il reçoit un signal spécifique.
- Il est possible de créer des fonctions avec le mot-clé `function` suivi d'un ensemble de commandes à exécuter entre accolades.

Signaux

Petits rappels



```
#!/bin/bash

trap montrap 2

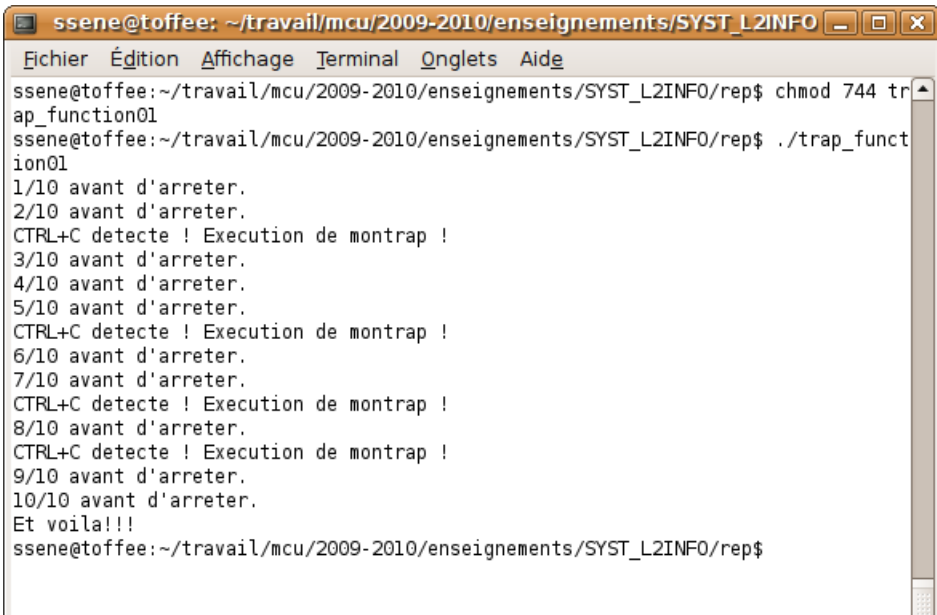
function montrap {
    printf "CTRL+C detecte ! Execution de montrap !\n"
}

for a in `seq 1 10`; do
    printf "$a/10 avant d'arreter.\n"
    sleep 1;
done

printf "Et voila!!!\n"
```

Signaux

Petits rappels

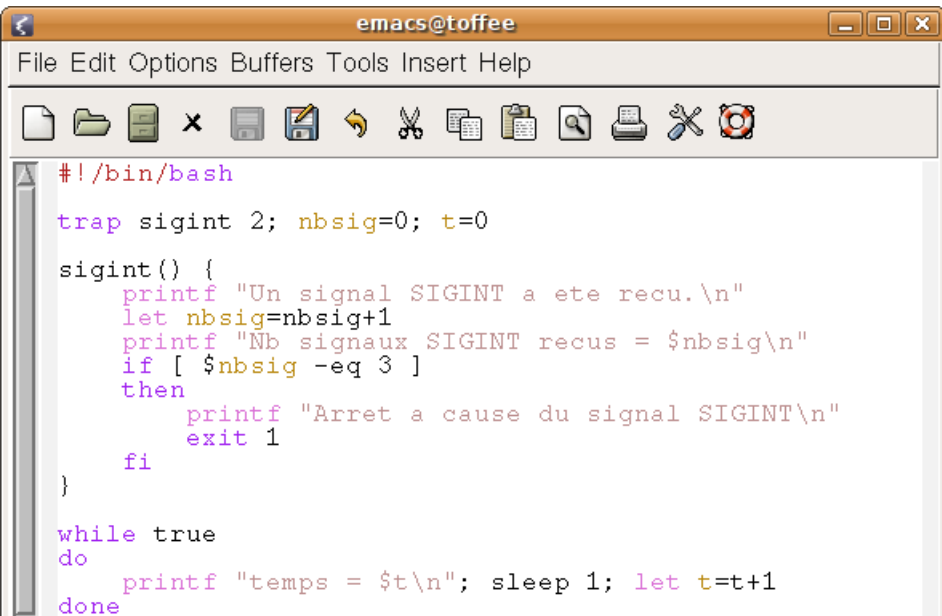


A terminal window titled "ssene@toffee: ~/travail/mcu/2009-2010/enseignements/SYST_L2INFO" with standard window controls. The menu bar includes "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The terminal shows a shell script being executed with the following output:

```
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$ chmod 744 trap_function01
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$ ./trap_function01
1/10 avant d'arreter.
2/10 avant d'arreter.
CTRL+C detecte ! Execution de montrap !
3/10 avant d'arreter.
4/10 avant d'arreter.
5/10 avant d'arreter.
CTRL+C detecte ! Execution de montrap !
6/10 avant d'arreter.
7/10 avant d'arreter.
CTRL+C detecte ! Execution de montrap !
8/10 avant d'arreter.
CTRL+C detecte ! Execution de montrap !
9/10 avant d'arreter.
10/10 avant d'arreter.
Et voila!!!
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$
```

Signaux

Petits rappels



The screenshot shows an Emacs editor window with the title bar 'emacs@toffee'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Insert', and 'Help'. The toolbar contains icons for file operations (new, open, save, close, print, find, etc.) and editing (undo, redo, cut, copy, paste). The main text area contains a shell script for handling the SIGINT signal.

```
#!/bin/bash

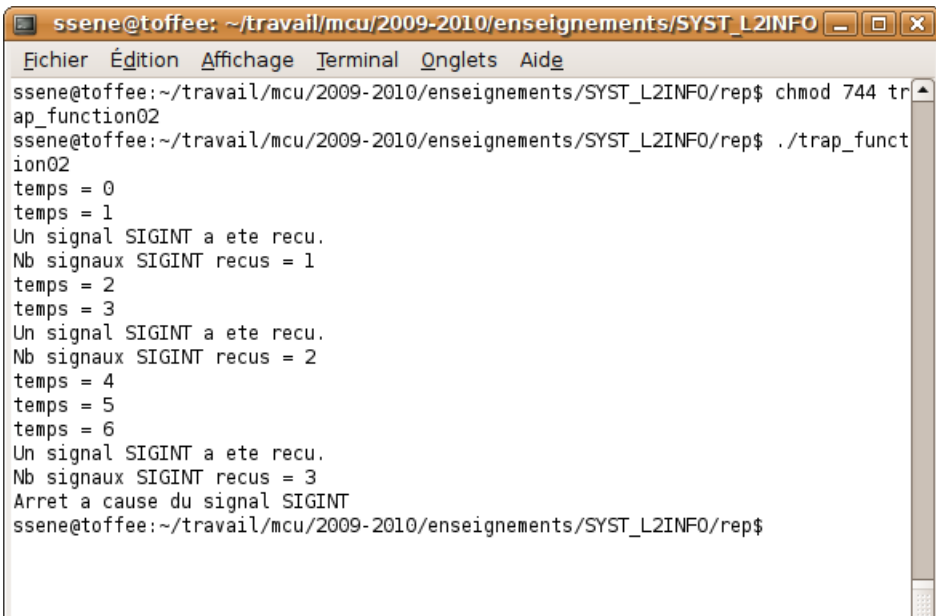
trap sigint 2; nbsig=0; t=0

sigint() {
    printf "Un signal SIGINT a ete recu.\n"
    let nbsig=nbsig+1
    printf "Nb signaux SIGINT recus = $nbsig\n"
    if [ $nbsig -eq 3 ]
    then
        printf "Arret a cause du signal SIGINT\n"
        exit 1
    fi
}

while true
do
    printf "temps = $t\n"; sleep 1; let t=t+1
done
```


Signaux

Petits rappels



The screenshot shows a terminal window with a title bar that reads "ssene@toffee: ~/travail/mcu/2009-2010/enseignements/SYST_L2INFO". The menu bar includes "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The terminal content shows a user running a program that demonstrates signal handling. The program sets a trap for SIGINT, increments a counter, and prints the current time and the number of received signals. It is interrupted three times by pressing Ctrl-C, and then it prints a message about being stopped by the signal.

```
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$ chmod 744 tr
ap_function02
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$ ./trap_funct
ion02
temps = 0
temps = 1
Un signal SIGINT a ete recu.
Nb signaux SIGINT recus = 1
temps = 2
temps = 3
Un signal SIGINT a ete recu.
Nb signaux SIGINT recus = 2
temps = 4
temps = 5
temps = 6
Un signal SIGINT a ete recu.
Nb signaux SIGINT recus = 3
Arrêt a cause du signal SIGINT
ssene@toffee:~/travail/mcu/2009-2010/enseignements/SYST_L2INFO/rep$
```

Signaux

Petits rappels

```
int chg_notes(promo_note *e)
{
    int *i;

    *i = 0;

    return *i;
}
```

```
ssene@toffee2:~/travail/mcu/enseignement/moi/CSYST_L2INFO/cours_SYST/exemple_L2/
promo_notes_c$ ./promo_notes
```

```
-----
Liste des notes en entree :
-----
```

```
Erreur de segmentation
```

Signaux

Introduction

- **NSIG** types de signaux différents.
- Identification par un entier.
- Utilisation de la bibliothèque standard `<signal.h>`.
- **Le type de signal est la seule information véhiculé par un signal.**
- Chaque type de signal possède un nom spécifique du type **SIGXXXX**, où XXXX est un suffixe particulier.
Exemple : **SIGSEGV** pour un signal indiquant une erreur de segmentation.

Signaux

Introduction

- Un événement particulier est associé à chaque type de signal.
- Un processus p_1 peut envoyer un signal s à un processus p_2 , sans que l'événement associé à s ne se soit produit.
- **Exemple** : `SIGSEGV` est envoyé automatiquement en cas d'erreur de segmentation ; on peut néanmoins l'envoyer manuellement en tapant : `kill -SEGV <pid>`.
- **Types d'événements** :
 - Externe au processus (frappe au terminal ou terminaison d'un autre processus).
 - Interne au processus (erreur : division par 0, segmentation mémoire).

Signaux

Signaux de base

SIGHUP	fin du processus leader de session	term
SIGINT	<Ctrl-c> tapé	term
SIGQUIT	<Ctrl-\> tapé	term + core
SIGILL	Détection d'une instruction illégale	term + core
SIGABRT	fin anormale (fonction <code>abort</code>)	term
SIGFPE	erreur arithmétique	term
SIGKILL	terminaison violente	term (*)
SIGSEGV	erreur de segmentation	term + core
SIGPIPE	écriture dans un tube sans lecteur	term
SIGALRM	fin de temporisation (fonction <code>alarm</code>)	term
SIGTERM	terminaison	term
SIGUSR1	utilisateur	term
SIGUSR2	utilisateur	term
SIGCHLD	fin d'un processus fils	ignoré

Signaux

Envoi de signaux

- Par le terminal, on utilise la commande `kill`.
- En C Posix, la primitive système est :

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

où `pid` est défini par :

> 0	processus d'identifiant <i>pid</i>
0	tous les processus dans le même groupe que le processus courant
-1	tous les processus du système sauf l' <i>init</i> .
< -1	tous les processus du groupe <i> pid </i>

Signaux

Envoi de signaux

- Par le terminal, on utilise la commande `kill`.
- En C Posix, la primitive système est :

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

où **sig** est défini par :

< 0 ou > <i>NSIG</i>	valeur incorrecte
0	aucun signal n'est envoyé mais les conditions d'erreur sont vérifiées
<i>autre</i>	signal identifié par <i>sig</i> envoyé

Envoi de signaux – un exemple

Programmes d'envoi de signal d'un père à un fils

`signal01` **et** `signal02`

(en live)

Comportement à la prise en compte

- À chaque type de signal correspond un comportement par défaut.
- Ce comportement est codé dans un **handler** par défaut, désigné par **SIG_DFL**.
- **SIG_DFL** ne représente pas le même comportement pour tous les types de signaux.
- Ces comportements par défaut sont :
 - 1 Terminaison du processus.
 - 2 Terminaison du processus avec image mémoire (fichier *core*).
 - 3 Signal ignoré.
 - 4 Suspension du processus
 - 5 Continuation : reprise d'un processus suspendu et ignoré sinon.

Comportement à la prise en compte

- Signaux **SIGKILL** (terminaison “violente”), **SIGSTOP** (suspension) et **SIGCONT** (continuation) : *handler* par défaut obligatoire.
- Tout processus peut installer, pour chaque signal d'autres types, un nouveau *handler*.
 - une valeur standard désignée par **SIG_IGN** pour ignorer le signal.
Attention : un signal ignoré est délivré donc l'indicateur de signal pendant bascule à 0.
 - une fonction spécifique définie par l'utilisateur.
- *Handler* = fonction ne renvoyant pas de valeur du type :
void handler(int sig)

Délivrance de signaux

- La délivrance se fait quand un processus passe **du mode noyau au mode utilisateur** (élection par l'ordonnanceur, retour d'un appel système. . .).
 - ⇒ Un processus n'est pas interruptible en mode noyau.
- Délivrance à un processus **endormi** à un niveau de priorité interruptible : **le processus passe à l'état prêt**. Les niveaux de priorité en question sont :
 - Primitives d'attente de signal (**pause** et **sigsuspend**).
 - **wait** et **waitpid**.
 - **open** bloquant.
 - **fcntl** pour poser un verrou.
 - **read** et **write** sur un fichier verrouillé.
 - **msgsnd** et **msgrcv** sur les files de messages.
 - **semop** pour réaliser des opérations sur un ensemble de sémaphores.

Signaux

Délivrance de signaux

- Délivrance à un processus **suspendu** :
 - Les signaux **SIGKILL**, **SIGTERM** et **SIGCONT** le réveillent :
 - **SIGKILL** le termine.
 - **SIGTERM** le termine sauf s'il est capté en ignoré.
 - **SIGCONT** le réveille.
 - Les autres signaux sont délivrés à son réveil.
- Délivrance à un processus **zombi** :
 - Aucun effet.

Signaux

Blocage des signaux

- Un processus possède **un masque** : ensemble de signaux dont la délivrance ne se fait pas par défaut.
- Un processus peut installer un masque de signaux quelconques, excepté **SIGKILL**, **SIGTERM** et **SIGCONT**.
- **Manipulation des ensembles de signaux**

```
int sigemptyset(sigset_t *p_ens);  
    *p_ens =  $\emptyset$   
int sigfillset(sigset_t *p_ens);  
    *p_ens =  $\{1, \dots, NSIG\}$   
int sigaddset(sigset_t *p_ens, int sig);  
    *p_ens = *p_ens  $\cup$  {sig}  
int sigdelset(sigset_t *p_ens, int sig);  
    *p_ens = *p_ens  $\setminus$  {sig}  
int sigismember(sigset_t *p_ens);  
    sig  $\in$  *p_ens
```



Signaux

Blocage des signaux

- **Blocage**



- L'installation manuelle d'un masque de blocage est réalisé par :

```
#include <signal.h>
int sigprocmask(int op, const sigset_t *p_ens,
                sigset_t *p_old);
```

- Nouveau masque construit à partir de **p_ens* et potentiellement de l'ancien masque **p_old*.
- *op* définit le nouvel ensemble :

SIG_SETMASK	<i>*p_ens</i>
SIG_BLOCK	<i>*p_ens</i> \cup <i>*p_old</i>
SIG_UNBLOCK	<i>*p_old</i> \setminus <i>*p_ens</i>

- **Obtenir la liste des signaux pendants bloqués :**

```
#include <signal.h>
int sigpending(sigset_t *p_ens);
```

Signaux

Blocage de signaux – exemple



Programmes de blocage des signaux SIGINT et SIGQUIT
signal03
(*en live*)

Manipulation de *handlers* – structure **sigaction**

- Cette structure définit le comportement général d'un processus lors d'une délivrance de signal.
- Définition abrégée de la structure :


```
struct sigaction {  
    void (*sa_handler) ();  
    sigset_t sa_mask;  
    int sa_flags;  
};
```



- **sa_mask** : liste des signaux à bloquer, si le *handler* est différent de **SIG_DFL** et **SIG_IGN**.
- *handler* installé avec la primitive **sigaction** \implies signal associé bloqué.
- Les indicateurs, **sa_flags**, ne doivent pas être utilisés pour assurer la portabilité

Manipulation de *handlers* – primitive **sigaction**

Caractéristiques générales

- Interface générale Posix pour l'installation dans un processus d'un *handler* pour un type de signal donné.
- L'installation d'un *handler*  ne passe pas par l'envoi d'un signal (sauf cas particuliers).
- Un *handler* installé le reste tant qu'on ne demande pas le contraire.

Manipulation de *handlers* – primitive **sigaction**

Définition de la fonction

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

- Si **act** **!=** **NULL**

- Il pointe vers un objet de type **struct sigaction** à installer pour le signal **signum**.
- Délivrance de **signum** entraîne l'exécution de **act->sa_handler** et, si **act->sa_handler** n'est ni **SIG_DFL** ni **SIG_IGN**, le masquage de **act->sa_mask** \cup **{signum}**.



- Si **oldact** **!=** **NULL**

- Permet de récupérer l'ancien comportement.

Manipulation de *handlers* – primitive `sigaction`

Exemples



Programmes d'utilisation de la structure et de la primitive `sigaction`
`signal04` et `signal05`
(*en live*)

Manipulation de handlers – remarque

- Il existe un autre moyen de pratiquer l'installation de *handler*.
- Interface d'origine d'Unix appartenant au C standard et non à la norme Posix.



- Fonction **signal**.

Extrait de la page de manuel

Le comportement de `signal()` varie selon les versions d'Unix, et a aussi varié au cours du temps dans les différentes versions de Linux. **Évitez de l'utiliser** **utilisez plutôt `sigaction(2)`.**

Attente d'un signal – primitive **pause**

- Permet de se mettre en attente de signaux.

- ```
#include <unistd.h>
int pause(void);
```



- À la délivrance d'un signal :

- soit le processus **se termine** si le handler associé est **SIG\_DFL** et que ce handler code la terminaison.
  - soit le processus **passse à l'état endormi** (il ne réagit alors qu'aux signaux **SIGTERM**, **SIGKILL** et **SIGCONT**).
    - Au réveil, il se remet en attente de signaux.
  - soit le processus **exécute le handler** correspondant au signal capté.
- **Ne permet pas directement d'attendre un type de signal particulier ni de savoir par quel type de signal le réveil a été fait.**

# Attente d'un signal – primitive **pause** exemples



Programmes d'utilisation de la primitive `pause`  
`signal06` et `signal07`  
(*en live*)

# Attente d'un signal – primitive **sigsuspend**



- L'appel à la primitive Posix

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

a pour effet de réaliser **ATOMIQUEMENT**, c'est-à-dire en une seule opération :

- L'installation du masque de signaux pointé par **mask** jusqu'au retour de l'appel. Ce masque se substitue au masque courant qui est réinstallé au retour de la fonction.
- La mise en sommeil du processus jusqu'à l'arrivée d'un signal non masqué et qui est capté ou qui termine le processus.

# Attente d'un signal – primitive `sigsuspend` exemple : résolution du problème d'interblocage



Programme d'utilisation de la primitive `sigsuspend`  
`signal08`  
(*en live*)



# Signaux

## Contrôle de connaissances

Prenez une feuille et :



- ❶ Écrivez un programme C qui, étant données les macro-définitions  $L$  et  $C$  :
  - alloue de l'espace mémoire pour une matrice d'entiers contenant  $L$  lignes et  $C$  colonnes,
  - remplit cette matrice de valeurs aléatoires (utiliser `rand()`) et
  - libère l'espace mémoire occupé par la matrice.
  
- ❷ Écrivez un programme C qui utilise la primitive de recouvrement **exec1** pour lister le contenu de la racine du système.

# IPC System V

## Plan du cours

- 1 Rappels sur le C et débogueurs
- 2 Rappels système et primitives de recouvrement
- 3 Signaux
- 4 IPC System V**
  - Files de messages
  - Sémaphores
  - Segments de mémoire partagée

# IPC System V

## IPC System V – Introduction

Les IPC (*Inter Process Communication*) System V forment un groupe de trois outils particuliers servant à faire communiquer des processus au niveau local :

- ① les **files de messages** (*messages queues*),
- ② les **segments de mémoire partagée** (*shared memory*),
- ③ les **sémaphores** (*semaphores*).

Ces outils sont indépendants du (extérieurs au) système de gestion de fichiers  $\implies$  **pas de redirection possible des entrées-sorties standard sur ces objets**

Le système les gère par l'intermédiaire de 3 tables, une pour chacun des outils.

# IPC System V

## IPC System V – points communs

- Un **outil IPC** est **identifié de manière unique** par :
  - un **identifiant externe** appelé la **clé**.  
*≈ même rôle que le chemin d'accès d'un fichier*
  - un **identifiant interne**  
*≈ même rôle que le descripteur d'un fichier*
- Un outil IPC est accessible à **tout processus connaissant son identifiant interne**. La connaissance de cet identifiant s'obtient :
  - par héritage ou
  - par une demande explicite au système au cours de laquelle le processus fournit son identifiant externe.

# IPC System V

## Clé

- L'identifiant externe d'un IPC System V (sa clé) est une valeur numérique entière de type `key_t` (équivalent à un entier long).
- Les processus souhaitant communiquer par un IPC spécifique **doivent s'entendre sur la clé de cet IPC.**
- La clé d'un IPC peut être :
  - **donnée en dur**, c'est-à-dire figée dans le code source des processus,
  - **calculée par le système** à partir d'une référence commune à tous les processus : un nom de fichier et un entier.
- Chacun des types d'objets possède son propre ensemble de clés.
  - Il peut exister au même instant un sémaphore et une file de messages, tous deux de clés (`key_t`) 100.

# IPC System V

## <sys/ipc.h>

- Constantes macro-définies

| Constante          | Rôle                                                        | Primitives              |
|--------------------|-------------------------------------------------------------|-------------------------|
| <b>IPC_PRIVATE</b> | Clé privée                                                  | ...get                  |
| <b>IPC_CREAT</b>   | Création si inexistance                                     | ...get                  |
| <b>IPC_EXCL</b>    | Utilisée avec <b>IPC_CREAT</b><br>si existence alors erreur | ...get                  |
| <b>IPC_NOWAIT</b>  | Opération non bloquante                                     | semop<br>msgrcv, msgsnd |
| <b>IPC_RMID</b>    | Suppression d'identification                                | ...ctl                  |
| <b>IPC_STAT</b>    | Extraction d'informations                                   | ...ctl                  |
| <b>IPC_SET</b>     | Modification des caractéristiques                           | ...ctl                  |

# IPC System V

## <sys/ipc.h>

- Structure `ipc_perm`

```
struct ipc_perm{
 uid_t uid; /* id du prop. */
 gid_t gid; /* id du groupe prop. */
 uid_t cuid; /* id du créateur. */
 gid_t cgid; /* id du groupe créateur. */
 unsigned short mode; /* droits d'accès. */
 unsigned short seq; /* nb utilisations. */
 key_t key; /* clé associée. */
};
```

⇒ **Identification interne de l'objet calculé en fonction de son indice dans la table et du nombre de fois qu'il a été utilisé.**

# IPC System V

## Calcul d'une clé

Le calcul d'une clé est réalisé par le système grâce à l'appel de :

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

- **pathname** est une référence vers un fichier existant et accessible.
- **proj\_id** est un entier quelconque.

**L'unicité de la clé renvoyée par l'appel à `ftok` est garantie par :**

- l'utilisation d'une même référence **et**
- l'utilisation d'entiers ayant les mêmes huit bits de poids faible.

Exemple :

`ftok("/home", 256) ⇔ ftok("/home", 1024)`



# IPC System V

## Calcul d'une clé

Programme illustrant certaines propriétés des clés

`cle01`  
(*en live*)

# Opérations de création et de contrôle

- Les IPC possèdent chacun des primitives de création et de contrôle. Elles sont respectivement de la forme **xxxget** et **xxxctl**.
- On y reviendra plus tard.
- En ce qui concerne les fonctions **xxxget** :
  - Le dernier argument (le *flag*) permet de spécifier les droits (cf. **stat**).
  - **Exemple** : **S\_IRUSR** et **S\_IWUSR** spécifient des permissions de lecture et écriture pour le propriétaire de l'IPC.
  - En l'absence de ces flags, seul **root** pourrait utiliser les IPC. Donc, si vous programmez à l'aide des IPCs et que votre code ne fonctionne qu'en mode root, c'est que vous avez oublié ces options.

# IPC System V

## IPC System V et Shell

- La commande **ipcs** permet de lister tous les IPC existants à un instant donné dans le système.
- Pour chaque IPC, cette commande fournit :
  - la valeur de la clé ;
  - l'identifiant interne ;
  - le propriétaire ;
  - les droits d'accès.

- Exemple :

```
---- Shared Memory Segments ----
```

| key        | shmid | owner | perms | bytes | ... |
|------------|-------|-------|-------|-------|-----|
| 0x0036d276 | 23    | root  | 600   | 4096  | ... |

```
---- Semaphore Arrays ----
```

| key        | semid | owner | perms | nsems |
|------------|-------|-------|-------|-------|
| 0x002fa327 | 0     | ssene | 666   | 2     |

```
---- Message Queues ----
```

| key | msqid | owner | perms | used-bytes | ... |
|-----|-------|-------|-------|------------|-----|
|-----|-------|-------|-------|------------|-----|

# IPC System V

## IPC System V et Shell

**Attention** – *Il faut penser à supprimer les IPC quand on ne s'en sert plus.*

- On verra comment faire en C plus tard.
- On peut utiliser la commande **ipcrm**.

|              |                                                                                                                          |
|--------------|--------------------------------------------------------------------------------------------------------------------------|
| <b>ipcrm</b> | $\left\{ \begin{array}{l} [ -M \mid -Q \mid -S ] \text{ clé,} \\ [ -m \mid -q \mid -s ] \text{ id.} \end{array} \right.$ |
|--------------|--------------------------------------------------------------------------------------------------------------------------|

# IPC System V



# IPC System V

## Introduction

- Mécanisme Unix implantant le concept de **boîte aux lettres**.
- **Remarque** : BSD propose un mécanisme différent, les *sockets* locales.
- Mode de communication :
  - $\neg$  **(flot continu de caractères)**,
  - $\implies$  **paquets identifiables**.

**Opération de lecture = extraction d'une opération d'écriture.**

- **Multiplexage possible** :
  - Envoi d'un message à plusieurs destinataires.
  - Extraction d'une file de messages des messages d'un certain type.

# IPC System V

## <sys/msg.h>

- Constantes symboliques macro-définies :

| Constante          | Interprétation et/ou utilisation                                          |
|--------------------|---------------------------------------------------------------------------|
| <b>MSG_NOERROR</b> | Pas d'erreur lors de l'extraction d'un message trop long (il est tronqué) |
| <b>MSG_R</b>       | Autorisation de lecture dans la file                                      |
| <b>MSG_W</b>       | Autorisation d'écriture dans la file                                      |
| <b>MSG_RWAIT</b>   | Indication qu'un processus est bloqué en lecture                          |
| <b>MSG_WWAIT</b>   | Indication qu'un processus est bloqué en écriture                         |

# IPC System V

## <sys/msg.h>

- Structure **msqid\_ds**

- Structure d'une entrée dans la table des files de messages.
- L'accès à la structure d'une file de messages est faite par l'appel à la primitive **msgctl**.

```
struct msqid_ds {
 struct ipc_perm msg_perm /* droits d'accès à l'objet */
 struct __msg *msg_first /* pointeur sur le 1er msg. */
 struct __msg *msg_last /* pointeur sur le dernier msg. */
 unsigned short int msg_qnum /* nb msg dans la file. */
 unsigned short int msg_qbytes /* nb max octets. */
 pid_t msg_lspid /* pid du dernier processus émetteur. */
 pid_t msg_lrpid /* pid du dernier processus récepteur. */
 time_t msg_stime /* date de dernière émission. */
 time_t msg_rtime /* date de dernière réception. */
 time_t msg_ctime /* date de dernier changement. */
 unsigned short int msg_cbytes /* nb actuel d'octets. */
};
```



## IPC System V

### <sys/msg.h>

- La manipulation des files de messages requiert la définition d'une structure spécifique à l'application développée sur le modèle suivant :

```
struct msgbuf{
 long mtype; /* type du msg. */
 char mtext[1]; /* texte du message. */
};
```

- Doit contenir un premier champ `long` pour le type du message.
  - Fournit un moyen de partitionner les messages.
  - Nombre entier strictement positif.
  - Rend possible la lecture d'un message d'un type particulier.
  - Permet le multiplexage.
- **La suite de la structure peut contenir des objets quelconque sous réserve qu'il soient contigus en mémoire (pas de pointeur).**

# IPC System V

## <sys/msg.h>

- Exemples : on veut définir des messages dont le contenu est un tableau de  $n$  chaînes de caractères auxquelles sont associées leur taille.

Exemple incorrect :

```
struct msgbuf1{
 long mtype;
 char *tab_ch;
 int *tab_l;
};
```

Exemple correct :

```
struct msgbuf2{
 long mtype;
 char tab_ch[n];
 int tab_l[n];
};
```

# Utilisation d'une file de messages

## Primitive **msgget**

- Création d'une nouvelle file de messages ou utilisation d'une file existante par un processus.
- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- Renvoie l'id interne d'une file de messages ou -1 en cas d'erreur.
- Le paramètre **msgflg** est une combinaison bit à bit des constantes **IPC\_CREAT**, **IPC\_EXCL** et des droits d'accès.

# Utilisation d'une file de messages

## Primitive `msgget`

### Effet d'un appel :

- ① Si clé = **IPC\_PRIVATE**, une nouvelle file est créée.
- ② Si clé  $\neq$  **IPC\_PRIVATE** et n'est pas celle d'une file existante :
  - ① si **IPC\_CREAT** positionné, **la file est créée**.
  - ② sinon, **erreur**.
- ③ Si clé  $\neq$  **IPC\_PRIVATE** et est celle d'une file existante :
  - ① si **IPC\_CREAT** et **IPC\_EXCL** non positionnés, **l'id de la file est renvoyé**.
  - ② sinon, **erreur**.

# Utilisation d'une file de messages – exemples simples

Programmes basiques illustrant l'utilisation/création d'une file

`msg01` **et** `msg02`  
(*en live*)

## Contrôle d'une file de messages – Primitive `msgctl`

- Permet d'accéder aux informations contenues dans l'entrée de la table des files de messages associée à une file de messages.
- Permet d'en modifier certains attributs.
- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- Valeurs de `cmd` (entre autres) :
  - `IPC_STAT` : l'entrée de file identifiée par `msqid` est écrite dans `buf`.
  - `IPC_SET` : l'objet pointé par `buf` est écrit dans l'entrée identifiée par `msqid`. Les champs modifiables sont : `msg_qbytes`, `msg_perm.uid`, `msg_perm.gid` et les 9 bits de poids faible de `msg_perm.mode`.
  - `IPC_RMID` : la file de messages est supprimée.

# Contrôle d'une file de messages – exemple simple

Programme illustrant le contrôle d'une file

`msg03`  
(*en live*)

# Envoi d'un message à une file

## Primitive **msgsnd**

- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz,
 int msgflg);
```

- Envoie le message pointé par **msgp** dans la file d'id **msqid**.
- **msgsz** est la taille en octets **du contenu du message** (on ne compte pas le type).
- Renvoie 0 en cas de succès et -1 sinon.
- **msgflg** peut valoir **IPC\_NOWAIT** pour empêcher le blocage de l'envoi (cpt par défaut) en cas de file pleine.



# IPC System V

## Envoi d'un message à une file

### Primitive **msgsnd**

- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz,
 int msgflg);
```

- Causes d'erreur :

- file inexistante [**EINVAL**] ou pas de droits d'écriture dessus **EPERM**.
- type du message à envoyer incorrect (< 1) [**EINVAL**].
- **IPC\_NOWAIT** positionné et la file est pleine [**EAGAIN**].

## Files de messages – exemples d'envoi

Programmes illustrant l'envoi de messages à une file

`msg04` **et** `msg05`  
(*en live*)

# Extraction d'un message d'une file

## Primitive **msgrcv**

- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
 long msgtyp, int msgflg);
```

- Demande d'extraction de la file d'id **msqid** d'un message de type **msgtyp** de taille inférieure ou égale à **msgsz**.
- Le message extrait est stocké dans **msgp**.
- Renvoie la taille du contenu du message écrit dans **msgp**.
- **msgflg** est une combinaison des constantes **IPC\_NOWAIT** (pas d'attente) et **MSG\_NOERROR** (message tronqué si trop long).

# Extraction d'un message d'une file

## Primitive **msgrcv**

- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
 long msgtyp, int msgflg);
```

- Le paramètre **msgtyp** spécifie le type du message à extraire :
  - Si **msgtyp** > 0 : le message le plus vieux de type **msgtyp** est extrait.
  - Si **msgtyp** = 0 : le message le plus vieux est extrait.
  - Si **msgtyp** < 0 : le message le plus vieux du type le plus petit inférieur ou égal à **|msgtyp|** est extrait (gestion de priorités).

# Extraction d'un message d'une file

## Primitive `msgrcv`

- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
 long msgtyp, int msgflg);
```

- Causes d'erreur :

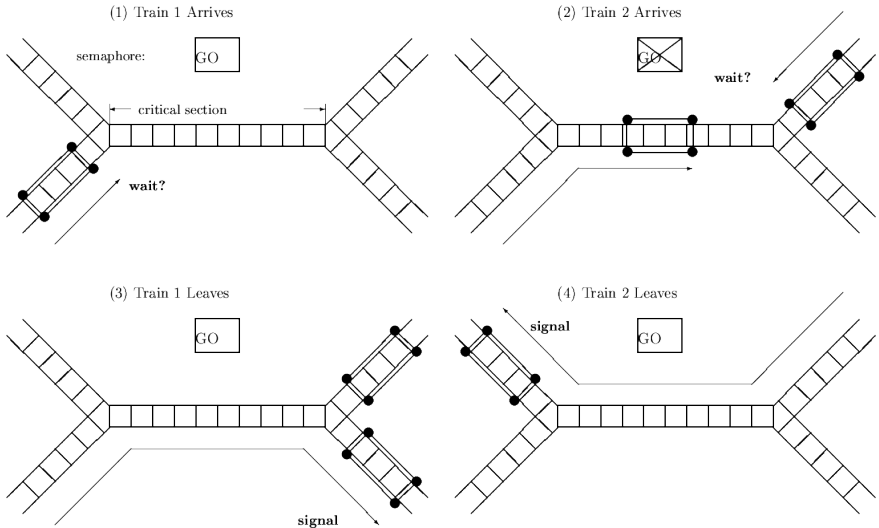
- file inexistante [**EINVAL**] ou pas de droits de lecture dessus **EPERM**.
- paramètre `msgsz` négatif [**EINVAL**].
- longueur du message > `msgsz` et **MSG\_NOERROR** n'est pas positionné [**E2BIG**].
- pas de message du type souhaité et **IPC\_NOWAIT** positionné.
- processus bloqué et un signal l'a interrompu [**EINTR**] ou la file a été supprimée [**EIDRM**].

# Files de messages – exemples d'extraction

Programme illustrant l'extraction de messages

`msg06`  
(*en live*)

# IPC System V



# IPC System V

## Sémaphores – généralités

### Définition

*Un sémaphore est une variable protégée qui fournit une abstraction utile pour contrôler l'accès à une ressource commune par plusieurs processus dans un environnement de programmation concurrent.*

- Ce concept a été inventé par Edsger Dijkstra qui l'a lui-même considéré comme étant obsolète par la suite.
- Les sémaphores fournissent la solution la plus courante à de nombreux problèmes d'**exclusion mutuelle**.
- Néanmoins, ils ne permettent pas d'éviter tous les interblocages.



# Sémaphores – une analogie



- **Briques de base :**
- Une bibliothèque ayant 10 salles de lecture (toutes identiques) utilisables par 1 personne à la fois.
- Pour accéder à une salle, une personne demande la permission au bibliothécaire.
- Quand une personne a fini d'utiliser la salle, il signale au bibliothécaire que la salle est vide.
- Si aucune salle n'est disponible, les gens attendent qu'une se libère.

# Sémaphores – une analogie

- **Idée générale :**
- Le bibliothécaire ne sait pas quelle(s) salle(s) est(sont) occupée(s).
- Il ne connaît que le nombre  $n$  de salles disponibles (au départ égal à 10).
- Quand une personne lui demande une salle, il décrémente  $n$ .
- Quand une personne libère une salle, il incrémente  $n$ .
- Une fois la salle obtenue, elle peut être utilisée aussi longtemps que désiré, ce qui implique que les réservations sont impossibles.

# Sémaphores – une analogie

- Retour sur l'analogie :
- Bibliothécaire = sémaphore.
- Salles = ressources.
- Personne = processus.
- Sémaphore initialisé à 10.
- Si sémaphore = 0, les processus “attendent” sous la forme d’une file (fifo).

## Sémaphores – quelques points importants

- Un sémaphore ne conserve pas d'information à propos de quelles sont les ressources disponibles. Il ne connaît que leur nombre.
- Les processus sont censés respecter le protocole. Les mauvais comportements sont notamment :
  - demander une ressource et oublier de la libérer,
  - libérer une ressource jamais demandée,
  - conserver une ressource inutilement.
- Même si les processus respectent le protocole, des interblocages peuvent se produire lorsque plusieurs sémaphores gèrent les ressources et que les processus doivent accéder à plusieurs ressources en même temps.

# Sémaphores – Le dîner des philosophes

- 5 philosophes.
- 2 actions exclusives : manger ou penser.
- Table circulaire avec 5 assiettes et 5 fourchettes.
- Pour pouvoir manger, il faut utiliser 2 fourchettes.
- Par définition, les philosophes ne se parlent pas.



⇒ **Problème** : possibilité d'interblocage quand chaque philosophe prend sa fourchette de gauche et attend perpétuellement sa fourchette de droite.

# Sémaphores – vue théorique de l'implantation

- Étant donné un sémaphore  $S$  de compteur  $K$ , il existe trois opérations fondamentales :
  - **Initialiser** (**Init**) : donner une valeur à  $K$ .
  - **Prendre** (**P**) : décrémenter la valeur de  $K$  si une ressource est libre, attendre sinon.
  - **Libérer** (**V**) : incrémenter la valeur de  $K$ .
- Formellement :
  - **P (S)** :  
si  $K = 0$  alors mettre le processus en attente  
sinon  $K \leftarrow K - 1$
  - **V (S)** :  
 $K \leftarrow K + 1$   
réveiller un ou plusieurs processus en attente

# Sémaphores – vue théorique de l'implantation

- Formellement :
  - **P(S) :**  
si  $K = 0$  alors mettre le processus en attente  
sinon  $K \leftarrow K - 1$
  - **V(S) :**  
 $K \leftarrow K + 1$   
réveiller un ou plusieurs processus en attente
- Toute implantation des sémaphores repose sur :
  - l'**atomicité** des opérations  $P$  et  $V$ .
  - l'**existence d'un mécanisme de mémorisation** des demandes d'exécution d'opérations  $P$  non satisfaites afin de réaliser le réveil de ces processus.

# Sémaphores – lecteurs / écrivains

## PROBLÈME

- On est dans la situation où plusieurs processus doivent accéder à une mémoire partagée à un moment.
- Quelques-uns doivent lire (les lecteurs), d'autres écrire (les écrivains).
- Contrainte : quand un écrivain accède à la mémoire, aucun autre processus ne peut y accéder.



# Sémaphores – lecteurs / écrivains

## PREMIÈRE SOLUTION NAÏVE

- On protège la mémoire partagée derrière un mutex.
- En conséquence, seul un processus peut accéder à la mémoire à un instant  $t$ .
- Sous-optimalité de la solution : deux lecteurs ne peuvent pas lire en même temps.

# Sémaphores – lecteurs / écrivains

## SOLUTION PRO-LECTEURS

- Même solution que tout à l'heure sauf qu'on tolère que plusieurs lecteurs lisent en même temps.
- Inconvénient de la solution : **Famine des écrivains.**

# Sémaphores – lecteurs / écrivains

## SOLUTION PRO-ÉCRIVAINS

- On part de la solution pro-lecteurs.
- Sous-optimalité car un écrivain doit attendre (potentiellement indéfiniment) que les lecteurs aient fini.
- Priorité donné aux écrivains dans la file d'attente.
- Inconvénient de la solution : **Famine des lecteurs.**

# Sémaphores – le coiffeur dormeur

## DONNÉES DU PROBLÈME

- Un salon de coiffure avec un coiffeur.
- Une chaise de coiffeur et une salle d'attente avec  $n$  chaises dedans.

## ORGANISATION DU COIFFEUR

- Quand le coiffeur a terminer de couper les cheveux d'un client, il le libère.
- Il va ensuite dans la salle d'attente pour voir s'il y a des clients qui attendent :
  - Si oui, il en fait assoir un sur la chaise et lui coupe les cheveux.
  - Sinon, il dort.

# Sémaphores – le coiffeur dormeur

## ORGANISATION DU CLIENT

- Quand le client arrive, il regarde ce que le coiffeur fait.
  - Si le coiffeur dort, il le réveille et s'assoit directement sur la chaise et se fait couper les cheveux.
  - Si le coiffeur est en train de couper des cheveux, il va dans la salle s'attente :
    - Si une chaise est libre, il s'assoit et attend son tour.
    - Sinon, il s'en va.

# Sémaphores – le coiffeur dormeur

## ANALYSE

- A priori, tout va bien : le salon devrait fonctionner correctement.
- En pratique, un nombre non négligeable de problèmes peuvent survenir.
- Intérêt pour l'informatique : ces problèmes sont représentatifs de problèmes généraux d'ordonnancement.

# Sémaphores – le coiffeur dormeur

## EXPLICATION

- **Les problèmes sont des conséquences de la non connaissance du temps** : les actions du coiffeur et du client prennent toutes un temps inconnu.
- Exemple de problème :
  - Un client arrive, voit que le coiffeur travaille et se dirige vers la salle d'attente.
  - Pendant ce temps, le coiffeur termine et va vérifier la salle d'attente. Il n'y voit personne et va dormir.
  - $\Rightarrow$  **Interblocage.**

# Sémaphores – le coiffeur dormeur

## INTUITION DE SOLUTION

- Il existe plusieurs solutions.
- Dans toutes, l'élément clé est un mutex, garantissant que seul un des acteurs peut changer d'état à un temps  $t$ .
- Le barbier doit acquérir ce mutex avant d'aller vérifier la salle d'attente et le libérer quand il dort ou qu'il travaille.
- Un client doit acquérir ce mutex avant d'entrer dans le salon et le libérer quand il accède à la chaise ou à la salle d'attente.



# Sémaphores – leur implantation SYSTEM V

- Avec un sémaphore, un processus peut s'assurer, par exemple, l'usage exclusif d'une ressource, à condition que les autres processus jouent le jeu.
- **Les sémaphores ont un rôle consultatif.**
- L'implantation SYSTEM V est plus riche que la simple implantation des opérations  $P$  et  $V$ .
- **Raison :** impossibilité de résoudre avec  $P$  et  $V$  le problème de l'acquisition simultanée d'exemplaires multiples de ressources différentes.
- La solution choisie est :
  - de définir l'opération  $P_n$  (resp.  $V_n$ ) permettant de diminuer (resp. d'augmenter) **atomiquement** de  $n$  la valeur d'un sémaphore si elle est supérieure ou égale à cette valeur (resp. sans contraintes)
  - de définir une opération  $Z$  permettant d'attendre que la valeur d'un sémaphore devienne nulle.

# Sémaphores – le fichier `<sys/sem.h>`

- La structure `__sem`

- Elle correspond à la structure dans le noyau d'un sémaphore :

```
struct __sem {
 unsigned short int semval; /* valeur du sémaphore. */
 unsigned short int sempid; /* PID du dernier processus utilisateur. */
 unsigned short int semcnt; /* nb de processus attendant */
 /* l'augmentation du sémaphore. */
 unsigned short int semzcnt; /* nb de processus attendant */
 /* la nullité du sémaphore. */
};
```

# Sémaphores – le fichier `<sys/sem.h>`

- La structure `semid_ds`

- Elle correspond à la structure d'une entrée dans la table des sémaphores accessible par la primitive `semctl` :

```
struct semid_ds {
 struct ipc_perm sem_perm; /* droits. */
 struct __sem *sem_base; /* pointeur sur le 1er sémaphore */
 /* de l'ensemble. */
 time_t sem_otime; /* date de dernière opération. */
 time_t sem_ctime; /* date de dernier changement. */
 unsigned short int sem_nsems; /* nombre de sémaphores */
 /* de l'ensemble. */
};
```

# Sémaphores – le fichier `<sys/sem.h>`

- La structure `sembuf`

- Elle correspond à une opération sur un sémaphore,  $P$ ,  $V$  ou  $Z$  :

```
struct sembuf {
 unsigned short int sem_num; /* numéro du sémaphore. */
 short sem_op; /* opération sur le sémaphore. */
 short sem_flg; /* option. */
};
```

- Il faut noter que :

- les numéros des sémaphores dans l'ensemble commencent à 0 ;
- c'est le signe de `sem_op` qui définit l'opération :
  - `sem_op < 0`  $\implies P_n$  ;
  - `sem_op > 0`  $\implies V_n$  ;
  - `sem_op = 0`  $\implies Z$  ;

## Utilisation de sémaphores – primitive **semget**

- Création d'un nouveau sémaphore ou utilisation d'un sémaphore existant par un processus.
- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- Renvoie l'id interne d'un ensemble de sémaphores ou -1 en cas d'erreur.
- Le fonctionnement est identique à celui de **msgget**.
- **nsems** indique le nombre de sémaphores de l'ensemble.
- Les différents sémaphores d'un ensemble en contenant  $n$  auront comme identification dans cet ensemble  $0, 1, \dots, n - 1$ .

## Utilisation de sémaphores – primitive **semop**

- C'est par cette primitive qu'un processus peut réaliser des opérations sur un ensemble de sémaphores connus.
- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops,
 unsigned int nsops);
```

- Les **nsops** opérations placées à l'adresse **sops** sont réalisées **atomiquement** par le noyau de manière séquentielle.
- Chaque opération de **sops** peut-être rendue individuellement non bloquante (option **IPC\_NOWAIT**).
- Renvoie 0 si succès et -1 si échec.

# Utilisation de sémaphores – primitive **semop**

## Interprétation des opérations

- Si  $n > 0$  alors  $V_n$  : valeur augmentée de  $n$ . Tous les processus en attente de l'augmentation sont réveillés.
- Si  $n = 0$  alors  $Z$  : le processus est bloqué tant que la valeur n'est pas nulle.
- Si  $n < 0$  alors  $P_{|n|}$  :
  - si opération non réalisable, le processus est bloqué (sauf demande contraire) et le nombre de processus en attente est incrémenté.
  - si opération possible, si la valeur du sémaphore devient nulle, tous les processus en attente de nullité sont réveillés.

# Utilisation de sémaphores – primitive **semop**

## Options des opérations

- **IPC\_NOWAIT** : déjà vu.
- **SEM\_UNDO** :
  - maintient pour chaque processus et sémaphore qu'il manipule une valeur d'ajustement ajoutée à la valeur du sémaphore en cas de terminaison du processus.
  - Toute opération réalisée sur un sémaphore avec **SEM\_UNDO** positionnée met à jour la valeur d'ajustement en y retranchant la valeur de l'opération.
  - Ainsi :
    - La valeur  $n$  est ajoutée pour une opération  $P_n$ .
    - La valeur  $-n$  est ajoutée pour une opération  $V_n$ .



## Contrôle de sémaphores – primitive **semctl**

- Permet de réaliser (outre les opérations de même nature que **msgctl** – consultation, modification de champs, suppression) des initialisations de sémaphores et la consultation de différentes valeurs.
- Prototype :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd,
 .../* arg */);
```

- Le type de **arg** dépend de **cmd**.
- **arg** indique également si **semnum** est un numéro de sémaphore ou un nombre de sémaphores.

# Contrôle de sémaphores – primitive **semctl**

## Interprétation des paramètres selon **cmd**

| <b>cmd</b>      | <b>semnum</b> | <b>arg</b>                          | <b>effet</b>                                                          |
|-----------------|---------------|-------------------------------------|-----------------------------------------------------------------------|
| <b>GETNCNT</b>  | num sem       | –                                   | nb sem en attente d'augmentation                                      |
| <b>GETZCNT</b>  | num sem       | –                                   | nb sem en attente de nullité                                          |
| <b>GETVAL</b>   | num sem       | –                                   | valeur du sem                                                         |
| <b>GETALL</b>   | nb sem        | tab d'entiers courts non signés     | le tab <b>arg</b> contient les valeurs des <b>semnum</b> premiers sem |
| <b>GETPID</b>   | num sem       | –                                   | PID du dernier processus ayant réalisé une opération sur le sem       |
| <b>SETVAL</b>   | num sem       | entier                              | Initialisation du sem à <b>arg</b>                                    |
| <b>SETALL</b>   | nb sem        | tab d'entiers courts non signés     | Initialisation des <b>semnum</b> premiers sem à <b>arg</b>            |
| <b>IPC_RMID</b> | –             | –                                   | Suppression de l'entrée                                               |
| <b>IPC_STAT</b> | –             | pointeur sur <b>struct semid_ds</b> | Extraction de l'entrée                                                |
| <b>IPC_SET</b>  | –             | pointeur sur <b>struct semid_ds</b> | Modification de l'entrée                                              |

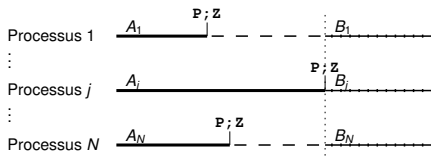
# Sémaphores – problème de rendez-vous

## ● LE PROBLÈME

- Une application décomposée en  $N$  processus concurrents.
- $\forall i \in \{1, \dots, N\}, i = A_i ; B_i$ .
- **Objectif : les  $N$  processus ne commencent l'exécution des séquences  $B_i$  que lorsque toutes les séquences  $A_i$  sont terminées.**

## ● INTUITION DE SOLUTION

- Les processus doivent se synchroniser sur le processus  $j$  le plus lent à exécuter sa séquence  $A_j$ .
- $\Rightarrow$  entre les séquences  $A_i$  et  $B_i$  ( $\forall i \in \{1, \dots, N\}$ ), mettre un mécanisme bloquant les processus jusqu'à ce que  $A_j$  soit terminée.



# Sémaphores – problème de rendez-vous

Programme proposant une solution incorrecte au problème de rendez-vous

sem01  
(*en live*)

# Sémaphores – problème de rendez-vous

Programme proposant une autre solution incorrecte au problème  
de rendez-vous

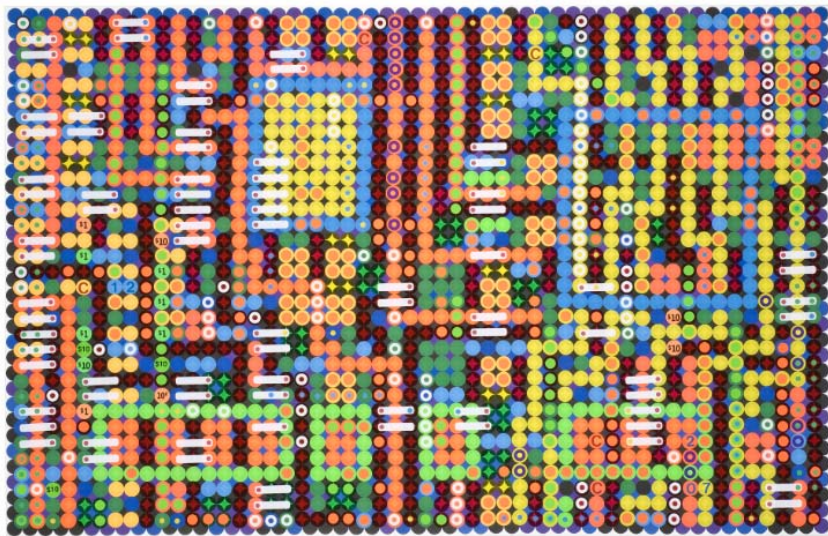
sem02  
(*en live*)

# Sémaphores – problème de rendez-vous

Programme proposant une solution au problème de rendez-vous  
utilisant la puissance des sémaphores en C

sem03  
(*en live*)

# IPC System V



# Mémoire partagée – introduction

- Caractéristique majeure commune à tous les mécanismes de communication vus jusqu'à maintenant :
  - **recopie** des données de l'espace d'adressage de l'émetteur vers le noyau.
  - **recopie** des données du noyau vers l'espace d'adressage du destinataire.
- $\Rightarrow$  basculements mode utilisateur / mode noyau.



# Mémoire partagée – introduction

- Les segments de mémoire partagée proposent **une approche complètement différente**.
  - Mise en commun de données à partagée.
  - **Partage de pages physiques** entre les processus par l'intermédiaire de leur espace d'adressage.
- $\Rightarrow$  **Aucune recopie** de données.
- Les pages partagées deviennent inévitablement des **ressources critiques**.
- Existence indépendante des processus : un processus peut demander le rattachement du segment à son espace d'adressage.

Mémoire partagée – fichier `<sys/shm.h>`● Structure `shmid_ds`.

```
struct shmid_ds {
 struct ipc_perm shm_perm; /* droits d'accès. */
 int shm_segsz; /* taille du segment. */
 pid_t shm_lpid; /* pid du dernier opérateur. */
 pid_t shm_cpid; /* pid du créateur. */
 unsigned short int shm_nattch; /* nb attachements. */
 time_t shm_atime; /* date du dernier attachement. */
 time_t shm_dtime; /* date du dernier détachement. */
 time_t shm_ctime; /* date du dernier controle. */
};
```

# Mémoire partagée – création/obtention

- Utilisation de la primitive **shmget**.
- Prototype :

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

- Fonctionnement général identique à ceux déjà vus avec les files de messages et les sémaphores.
- Le paramètre **size** décrit la taille du segment.
  - Lorsque que le segment existe déjà, la taille donnée doit être inférieur ou égale à la taille du segment.

# Mémoire partagée – Attachement

- Utilisation de la primitive **shmat**.
- Prototype :

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr,
 int shmflg);
```

- Permet à un processus qui connaît l'identifiant d'un segment de lui associer une adresse dans son espace d'adressage.
- C'est à partir de cette adresse qu'il pourra accéder au contenu du segment.
- Le valeur de retour est l'adresse où l'attachement a été réalisé.

# Mémoire partagée – Attachement

- Le problème majeur de l'attachement réside dans le **choix de l'adresse d'attachement**.
- 2 règles à respecter :
  - Ne pas entrer en conflit (immédiatement ou plus tard – augmentation de la pile) avec des adresses déjà utilisées ou susceptibles de l'être  $\Rightarrow$  **adresses appartenant à un intervalle particulier**.
  - Ne pas violer la forme générale des adresses imposées par le systèmes : **commencement à des limites de pages mémoire et adresses alignées**.

# Mémoire partagée – Attachement

- Deux possibilités pour l'adresse d'attachement.
  - **adr = NULL : adresse choisie par le système.**  
Solution qui garantit que l'adresse sera correctement construite et que l'application développée sera portable.
  - **adr != NULL : adresse choisie par l'utilisateur.**  
Solution qui rend possible à un utilisateur de choisir une adresse. Le système peut néanmoins arrondir cette adresse pour qu'elle satisfasse les contraintes qu'il impose.

## Mémoire partagée – Attachement

- Un segment de mémoire est par défaut accessible en lecture et en écriture.
  - Il est néanmoins possible de demander l'attachement en lecture seule.
    - Utilisation de l'option **SHM\_RDONLY**.
    - Dès lors, si une demande d'écriture est faite, une erreur de segmentation se produira.
- 
- ***Après un `fork`, le fils hérite des segments de mémoire partagée.***

## Mémoire partagée – Attachements multiples et utilisation

- Un segment de mémoire peut, selon la nature des systèmes d'exploitation, être attaché plusieurs fois à des adresses différentes par un même processus.
- Adresse renvoyée par `shmat` est un pointeur vers le premier octet du segment.
- Cette adresse est **alignée** (contrairement au messages des files) :
  - $\Rightarrow$  l'espace défini par le segment peut être structuré à souhait.
  - Exemple : tableaux d'entiers, de réels. . . , chaînes de caractères, structures. . .
  - **Attention !!** Les structures chaînées (listes, arbres) ne peuvent être adressées par pointeurs : plusieurs attachements ne correspondront pas aux mêmes adresses (pointeurs = adresses absolues).



# Mémoire partagée – Détachement

- Le détachement se fait à l'aide de la primitive **shmdt**.
- Prototype :

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

- Si une demande de suppression du segment est réalisée, il faut le nombre d'attachement soit égal à 0.
- La terminaison d'un processus, comme son recouvrement, entraîne le détachement du segment de mémoire.

## Mémoire partagée – Contrôle

- Le contrôle d'un segment de mémoire s'effectue par l'appel à **shmctl**.
- Son fonctionnement général est identique à ceux déjà vus.
- Prototype :

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- **IPC\_RMID** : suppression du segment différée jusqu'à son détachement complet.
- **IPC\_STAT** : récupération d'informations.
- **IPC\_SET** : modification possible des champs **shmperm.uid**, **shmperm.gid** et **shmperm.mode**.

# Création, initialisation et lecture dans un segment

Programme illustrant l'utilisation basique d'un segment de  
mémoire partagée

shm01  
(*en live*)

# Le problème du rendez-vous - 2 nouvelles solutions

Programme proposant une solution au problème de rendez-vous  
avec mémoire partagée et 1 sémaphore

shmsem01

(*en live*)

# Le problème du rendez-vous - 2 nouvelles solutions

Programme proposant une solution au problème de rendez-vous  
avec mémoire partagée et 2 sémaphores

shmsem02

(*en live*)