CentraleSupélec

Programmation Système

CentraleSupélec

Merci

- Alexandre Dang : Programmation système sous Linux et Windows (électif 2A, Rennes)
- Frédéric Tronel et Pierre Wilke : Systèmes d'exploitation (électif 2A, Rennes)
- Idir Ait Sadoune : Systèmes d'exploitation et virtualisation (Mastère spécialisé ASI, Saclay)

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Objectifs

- *La programmation système est un type de programmation qui vise au développement de programmes qui font partie du système d’exploitation d’un ordinateur ou qui en réalisent les fonctions [...]*

  Wikipédia

- Dans le cadre de ce cours, l'objectif est de découvrir et d'utiliser quelques services de bas niveau fournis pas un système d'exploitation.

- Linux est pris comme exemple pour les services retenus.

- Connaissances supposées : les mêmes que celles du cours « Concepts des langages de programmation - Mise en œuvre en C/C++ ».

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Système d'exploitation

- Un système d’exploitation (ou OS, Operating System) est une couche logicielle conceptuellement située entre le matériel et les applications.

- Il offre aux programmeurs et aux utilisateurs un ensemble d’abstractions qui permettent de ne pas avoir à gérer les spécificités du matériel.

- Les applications font des appels systèmes à des services proposés par le système d’exploitation qui, seul, gère les implications matérielles et retourne le résultat de l’appel.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Services

- Lancement et gestion des applications.
- Gestion de la mémoire.
- Gestion des E/S et des fichiers.
- Gestion des périphériques.
- Coopération inter-tâches.
- Communications via le réseau.

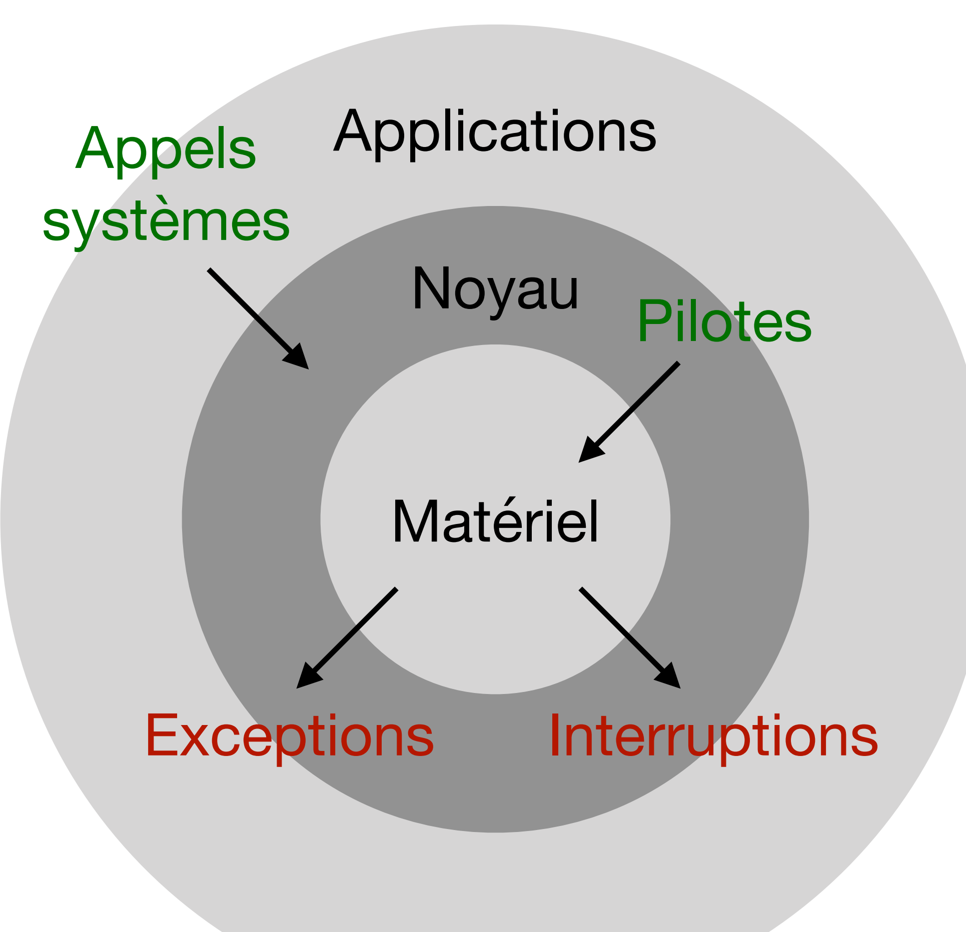
(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Espaces noyau et utilisateur

- Dans les systèmes d'exploitation modernes, il existe une distinction nette entre l'espace noyau et l'espace utilisateur.
- Le noyau s'exécute dans une zone mémoire particulière et avec des privilèges élevés.
- Il peut accéder à toute la mémoire, à tous les périphériques, et à toutes les instructions du microprocesseur.
- Les applications s'exécutent pour leur part dans l'espace utilisateur.
- Elles ne peuvent accéder qu'à la partie de la mémoire que le noyau leur a attribuée et ne peuvent utiliser qu'un sous-ensemble des instructions du microprocesseur.
- Le noyau contrôle leurs accès aux périphériques via les pilotes.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Liens entre couches



- Appels systèmes : demandes de services
- Noyau (kernel) : partie du système d'exploitation rendant des services en lien avec le matériel
- Interruptions : évènements produits par le matériel
- Exceptions : événements générés par le processeur
- Pilotes (drivers) : applications contrôlant les périphériques

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Multics

- Multiplexed Information and Computing Service (Wikipédia)
- Projet commun entre le MIT, les Bell Labs et General Electric (GE), démarré en 1964.
- Les Bell Labs se sont retirés en 1969.
- C’est un système d’exploitation extrêmement novateur mais qui a connu un succès commercial en demi-teinte.
- Les chercheurs des Bell Labs impliqués dans Multics s’inspireront de certaines des idées présentes dans Multics pour concevoir un système plus simple : Unix.

©2024 CentraleSupélec - Dominique Marcadet

Unix : l'origine (1969)

- Les Bell Labs trouvent le projet Multics trop complexe et ambitieux.
- Ils se retirent progressivement du projet.
- Les derniers chercheurs à s'être retirés, reconsidèrent la construction d'un système d'exploitation multitâche de conception plus simple que Multics.
- Les concepteurs : Ken Thompson, Brian Kernighan, Dennis Ritchie
- Une première version est développée sur un PDP-7, dans l'optique de faire tourner le jeu vidéo Space Travel.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Unix : l'expansion (1)

- Les Bell Labs fournissent un soutien aux trois chercheurs, à la condition qu'ils écrivent un éditeur de texte et un système de mise en page pour le nouvel OS.
- Celui-ci est porté sur PDP-11.
- Brian Kernighan suggère le nom d'Unics (Uniplexed Information and Computing Service) qui est un jeu de mot à propos de Multics.
- Le terme UNIX remplacera rapidement celui d'Unics.
- Le logiciel de mise en page TROFF est développé et sera utilisé par la suite pour la rédaction de l'ensemble des brevets de la compagnie.
- En 1972, UNIX est réécrit en C. Ceci permit de le porter plus facilement vers d'autres processeurs.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Unix : l'expansion (2)

- En 1956, une condamnation dans un procès antitrust interdit à AT&T de se lancer dans le commerce lié aux ordinateurs.
- Les Bell Labs qui sont une division de AT&T ne peuvent donc pas commercialiser UNIX.
- Celui-ci est donc distribué de manière gratuite (avec le code source).
- UNIX devient de plus en plus utilisé et populaire dans les universités américaines notamment pour l'enseignement des systèmes d'exploitation.
- La septième version d'UNIX est produite en 1979.
- Cette version marque un changement de licence : il n'est plus possible d'étudier le code source du système d'exploitation.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

CentraleSupélec

Unix : le début de la division

- En 1983, le département de la Justice américain (DOJ) entame un second procès antitrust contre AT&T.
- Ce procès a pour résultat le découpage de la compagnie, mais la relève de son interdiction de se lancer dans le commerce des ordinateurs.
- AT&T en profite pour commercialiser UNIX.
- Comme les conditions de distribution sont devenues moins favorables pour les universités, l'université de Berkeley (UCB) continue à développer sa propre version (Berkeley Software Distribution).
- La branche BSD continue d'exister de nos jours, elle s'est subdivisée en de nombreuses variantes.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Unix : la division

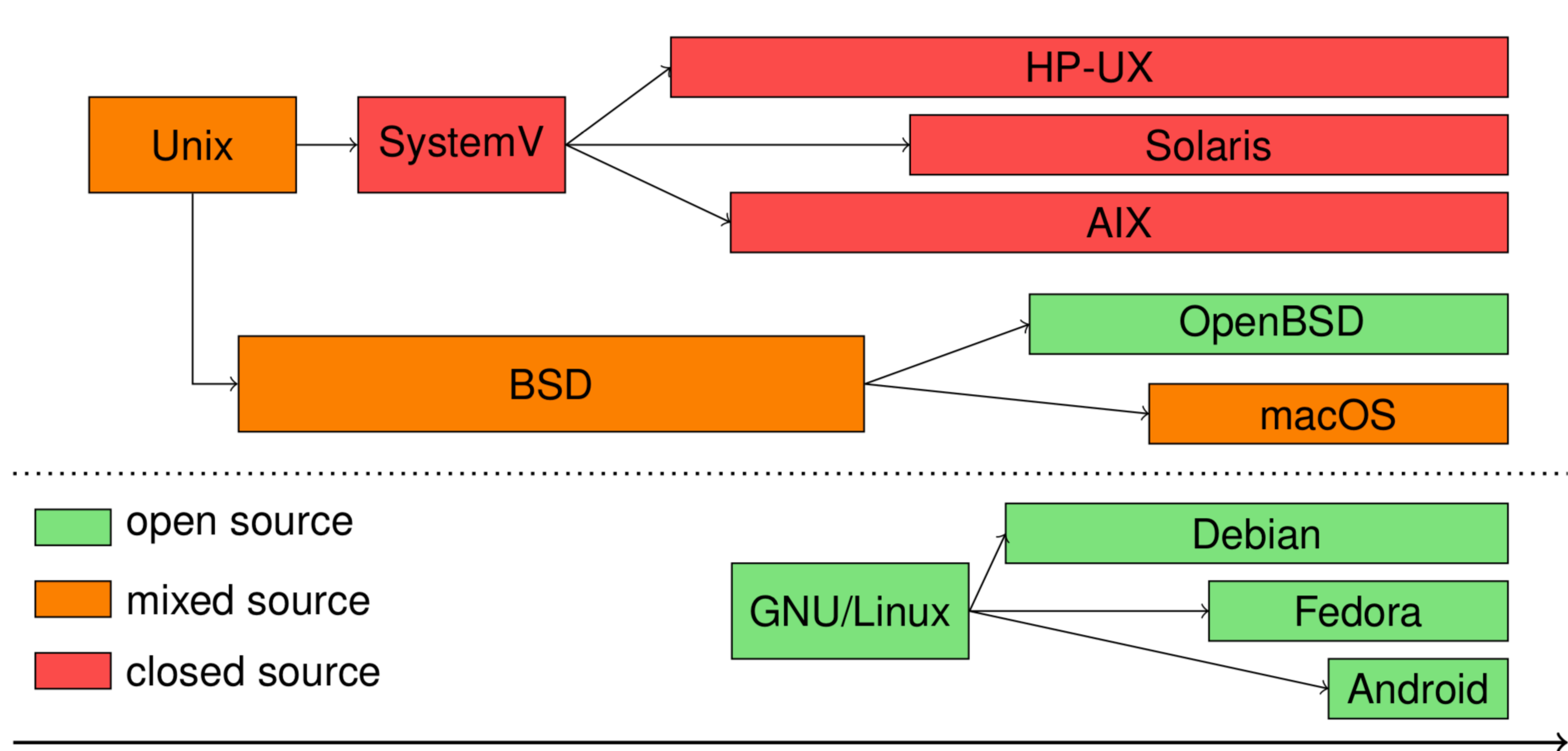
- C’est aussi durant les années 80 que de nombreuses compagnies privées se mettent à produire leur propres versions d’UNIX, c’est ce qu’on a appelé la guerre des UNIX.

- Malgré des tentatives de normalisation (dont POSIX), les stratégies commerciales des différentes compagnies ont finalement nuit au succès commercial d’UNIX.

- Timeline

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Unix : familles



cc BY-SA ©2024 CentraleSupélec - Dominique Marcadet

GNU/Hurd

- Richard Stallman : projet de développer un système d'exploitation libre et compatible Unix (1983).

- GNU « GNU's Not UNIX ».

- Au début des années 1990, le projet GNU possède une version utilisable de tous les éléments nécessaires à la construction d'un système d'exploitation à l'exception du plus central : le noyau.

- Projet GNU/Hurd : micro-noyau

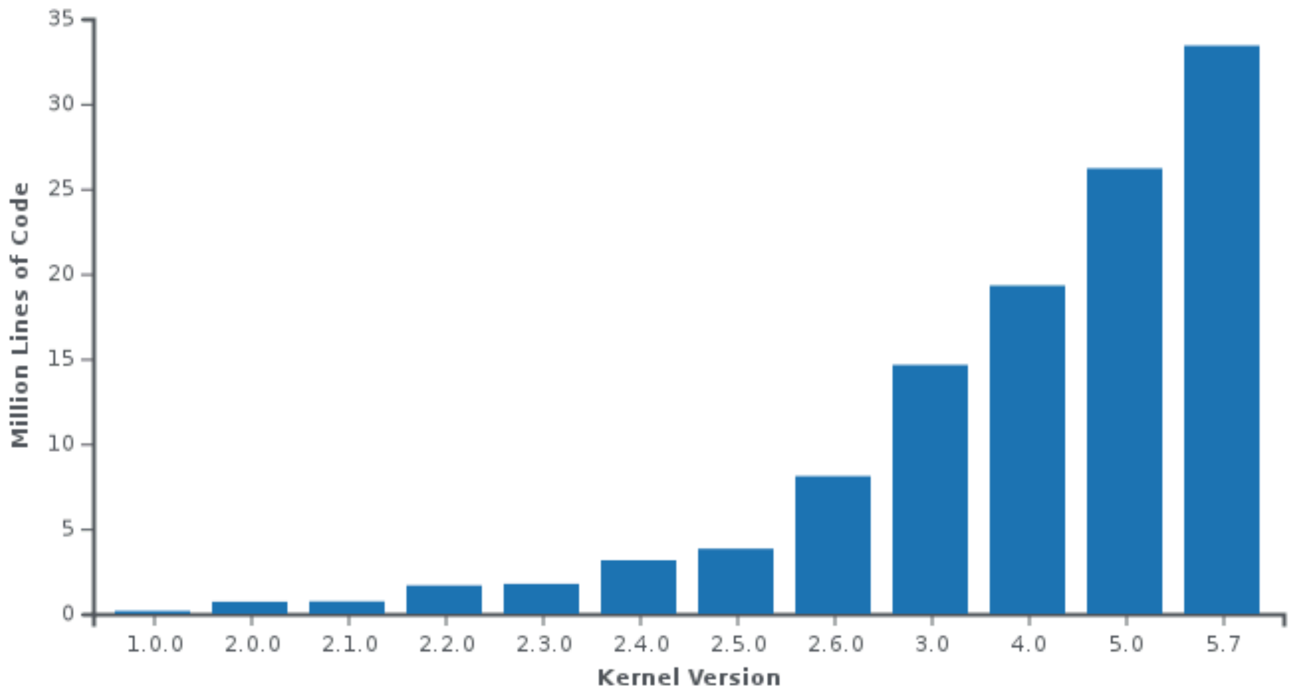
(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Linux

- Linus Torvalds
- comp.os.minix
  (message, minix)
- Noyau monolithique
- Historique des versions

- Les composants du noyau

- Nombreuses distributions



Crédit : https://en.wikipedia.org/wiki/Linux_kernel

©2024 CentraleSupélec - Dominique Marcadet

PDP-11 et shell

Crédit : Stefan_Kögl / CC BY-SA

- Interface utilisateur de type « ligne de commande »
- *shell* qui entoure le noyau (*kernel*)
- Le premier (*sh*) pour les versions 1 à 6 d'Unix
- De nombreuses variantes existent maintenant

©2024 CentraleSupélec - Dominique Marcadet

De MS-DOS à MS-Windows 3

- MS-DOS 1.0, sorti en 1981, était un système d'exploitation 16 bits mono-utilisateur en ligne de commande. Il occupait 8 Ko en mémoire.
- Deux versions de MS-Windows sortent respectivement en 1985 et 1987. Elles ne rencontrent pas un grand succès.
- Windows 3, sorti en 1990 pour l'Intel 386, se vend à plus d'un million d'exemplaires en 6 mois. Il s'agit avant tout d'une surcouche graphique à MS-DOS, pas d'un vrai système d'exploitation.
- Même s'ils apportent de nombreuses innovations technologiques (Multiprogrammation, gestion des processus, mémoire virtuelle), MS-Windows 95, 98 et Me sont en fait construits en interne sur la base MS-DOS.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

De Windows NT 3.1 à Windows XP

- Parallèlement à Windows 3, Microsoft (après l'arrêt de sa collaboration avec IBM sur OS/2) conçoit un système d'exploitation entièrement 32 bits dédié aux serveurs, Windows NT, qui introduit l'API Win32. La première version, Windows NT 3.1, sort en 1993.

- Windows NT 4, sorti en 1996, propose la même API Win32 mais exhibe la même interface graphique que MS-Windows 95. 4 familles de processeurs sont supportés.

- Son successeur, Windows 2000, ne supportera plus de manière active que les processeurs Intel x86.

- L'arrivée de Windows XP (2001) marque un tournant important dans l'évolution des gammes d'OS Microsoft : pour la première fois, les systèmes orientés professionnels et les systèmes orientés grand-public sont basés sur le même noyau et l'API Win32.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

De Windows Vista à Windows 11

- Windows Vista (2007), plus de 5 ans après Windows XP, a reçu beaucoup de critiques.
- Windows 7 (2009) a été lui très bien accueilli
- Windows RT (2012) à destination des processeurs ARM (Surface RT)
- Windows 10 en 2015, Windows 11 en 2021
- Timeline

©2024 CentraleSupélec - Dominique Marcadet

macOS

- Mac OS Classic de 1984 à 2001(Système 1 à Mac OS 9)
- macOS (Mac OS X jusqu'en 2012, OS X jusqu'en 2015)
  - Basé sur un noyau XNU et l'implémentation BSD d'Unix (Darwin)
  - Interface graphique propriétaire issue de Next
- OS dérivés : iOS, iPadOS, watchOS, tvOS, visionOS, audioOS

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Unix : tout est fichier

- fichier normal
- répertoire
- lien symbolique : une référence vers un autre fichier
- named pipe ou FIFO : un outil de communication unidirectionnel entre processus locaux
- socket : un outil de communication bidirectionnel entre processus distants
- périphériques (écran, imprimante, clavier, disque, clef USB...)
- processus (Linux)

BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Descripteur de fichier

- Tout ces fichiers sont manipulables par des descripteurs de fichier

- En C ils sont représentés par des entiers non négatifs (**`int`**)
  → `STDIN_FILENO` (`0`) : entrée standard
  → `STDOUT_FILENO` (`1`) : sortie standard
  → `STDERR_FILENO` (`2`) : sortie d'erreurs standard

- Avantage/désavantage:

  + API commune pour pouvoir gérer toutes les ressources avec des descripteurs de fichiers
    → `open()`, `creat()`, `read()`, `write()`, `fcntl()` et `close()`

  - Fonctions génériques et de bas niveau, ne permettant pas une utilisation simple de ressources spécifiques

BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Flux de données

- Surcouche aux descripteurs de fichiers pour les *fichiers normaux*

- Dans la bibliothèque standard C (`#include <stdio.h>`)
  → Représentation : `FILE *`
  → `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fscanf()`, `fprintf()`...

- Avantages
  + Portabilité, appartient au standard C
  + Performance, gestion plus optimisée des lectures/écritures
  + Simplicité, fonctions spécialisées dans la gestion de fichiers normaux

- Trois flux standards pour chaque processus
  → `stdin`, `stdout` et `stderr`

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
    - Introduction
    - Historique
- Programmation sous Unix
    - Fichier
    - Processus
    - Communication
    - Threads
    - Synchronisation

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Programme et processus

- Un ***programme*** est l'ensemble des instructions et données décrivant l'apparence et le comportement d'une application
    - Un programme est stocké dans un fichier
    - Un programme est statique
    - Un programme est exécutable
- Un ***processus*** est l'ensemble des informations décrivant l'état d'un programme en cours d'exécution
    - Un processus n'existe qu'en mémoire
    - Un processus est dynamique
    - Plusieurs processus peuvent être associés à un même programme

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Analogie culinaire classique

- Recette = programme
- Cuisinier = processeur
- Plat en cours de préparation = processus
- Un cuisinier peut préparer simultanément plusieurs plats (certains pouvant correspondre à la même recette)

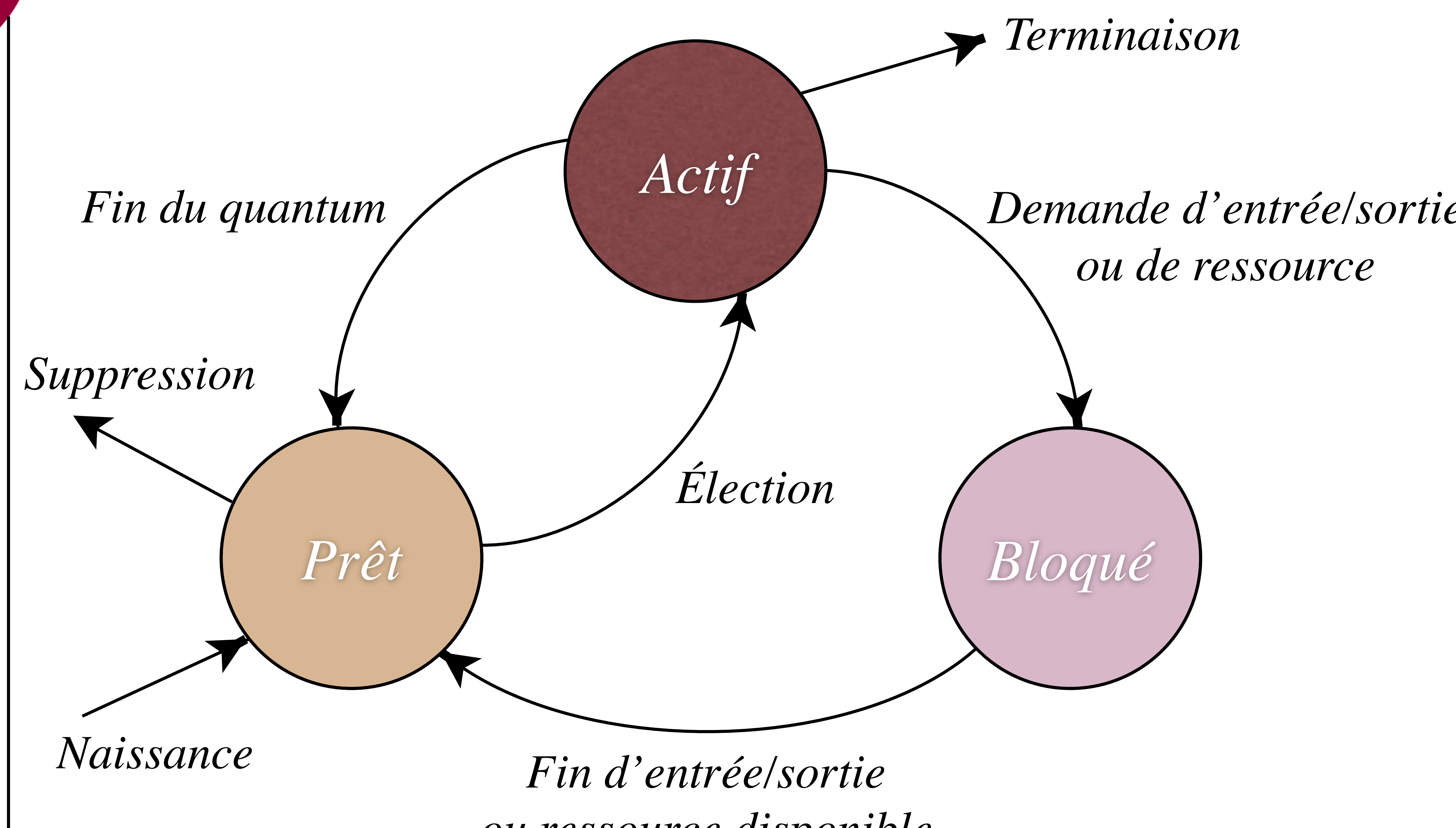
BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Gestion des processus

- Le système d’exploitation a en charge :
  - La création de processus
  - La répartition de l’utilisation du processeur par les différents processus (***ordonnanceur***, ***scheduler***)
  - La communication entre processus (copier/coller, mémoire partagée…)
  - La gestion de l’utilisation des ressources par les processus
    - mémoire
    - entrées/sorties

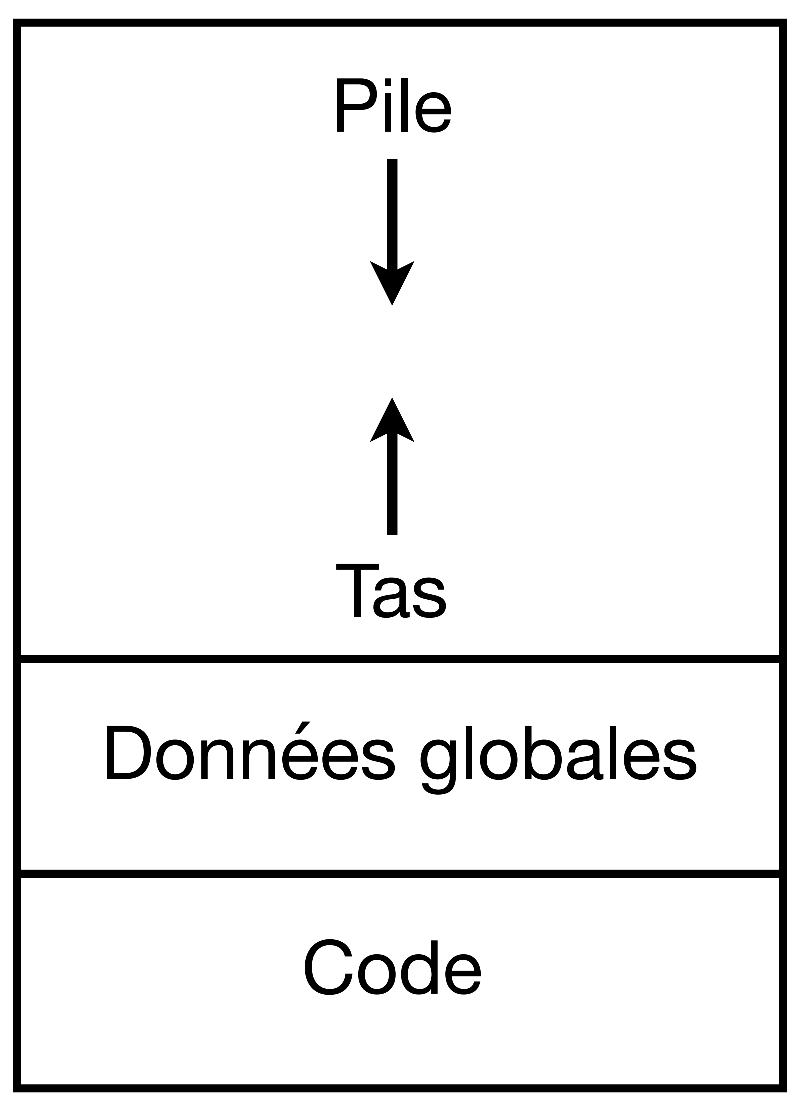
(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

États d'un processus



©2024 CentraleSupélec - Dominique Marcadet

Mémoire vue d'un processus



- Chaque processus pense être le seul à s'exécuter sur l'ordinateur (hors aspects liés aux communications).
- Un processus voit un espace mémoire *illimité* (de l'adresse 0 à l'adresse ...).
- Les adresses manipulées par un processus sont dites **logiques**.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Mémoire vue du SE

CentraleSupélec

- Le système d'exploitation a accès à la mémoire réelle, le noyau utilise des adresses **physiques**.
- Il alloue à chaque processus une partie de cette mémoire réelle.
- Il configure le composant matériel (MMU) chargé de convertir les adresses logiques du processus actif en adresses physiques.
- Les ordinateurs actuels (processeur et système d'exploitation) utilisent le mécanisme de **pagination** pour cela (ils offrent aussi la **mémoire virtuelle**).

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Bloc de contexte

- Lors du changement de processus actif, l'ordonnateur doit sauvegarder l'état du processus courant :
  - registres du processeur,
  - ressources utilisées :
    - espace mémoire utilisé (configuration de la pagination),
    - fichiers ouverts,
    - canaux de communications utilisés,
  - ...
- et restaurer celui du processus qui va (re)devenir actif

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Types de processus

- Différents types de processus
  - Processus contrôlé par le shell
    `./a.out`
  - Processus en tâche de fond
    `gedit &`
  - Services (daemon)
    `httpd`, `sshd`…

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Création d'un processus

- La seule façon de créer un processus est de dupliquer le processus courant avec `fork()`

```
pid_t fork();
```

  - retourne 0 si on se trouve dans le processus fils
  - retourne le PID du fils si on se trouve dans le processus père

- Un processus possède un identifiant unique appelé PID

```
pid_t getpid();
```

- Tous les processus ont un ancêtre commun : le processus lancé au démarrage (PID = 1).

BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Changement de programme

- Pour changer le programme d'un processus (en général le fils), il faut appeler une fonction de la famille `exec()` :

```
int execl( const char * path,
           const char * arg, ..., NULL );

int execv( const char * path,
           char * const argv[] );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Démo

GitLab

`cours/progsyst/Fork.c`

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Terminaison d'un processus

- Terminaison normale

`return` depuis la fonction `main()`

```
void exit( int status );
```

- Terminaison anormale

```
void abort();
```

```
void assert( expression );
```

- Attente de la terminaison d'un processus fils

```
pid_t wait( int * status );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Gestion de la mémoire

- Gestion de bas niveau

```
void * mmap(
  void * addr, size_t length, int prot,
  int flags, int fd, off_t offset );

int munmap( void * addr, size_t length );
```

- Gestion via libc

```
void * malloc( size_t size );

void   free( void * ptr );

void * calloc( size_t nmemb, size_t size );

void * realloc( void * ptr, size_t size );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

©2024 CentraleSupélec - Dominique Marcadet

IPC

- Douglas McIlroy (1978) et Peter H. Salus (1994) :
  - Philosophie Unix
    - Write programs that do one thing and do it well.
    - Write programs to work together.
    - Write programs to handle text streams, because that is a universal interface.
- Les outils de communication entre processus sont communément appelés IPC (Inter-Process Communication)

cc BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Différents IPC

- Signaux
- *Pipes* ou tubes
- Sockets
- Fichiers partagés
- Mémoire partagée
- (Files d'attente de message)

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Signaux

- Message asynchrone d'un processus à un autre

- Envoi des signaux avec

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

- Action sur réception d'un signal :

```
int sigaction( int signum,
    const struct sigaction * act,
          struct sigaction * oldact);
```

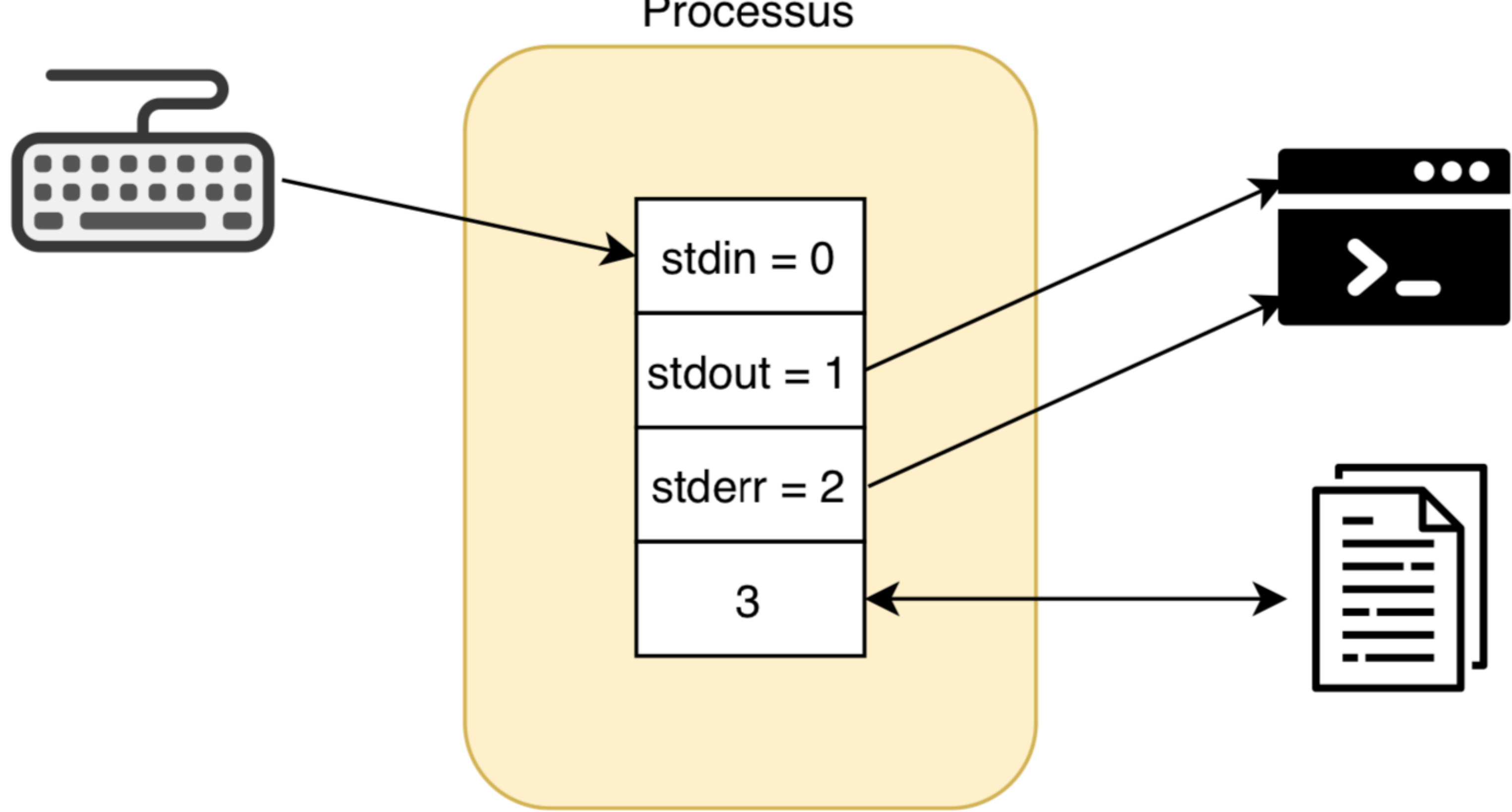
- Signaux standards

`SIGINT` (Ctrl+C), `SIGQUIT` (Ctrl+D) …

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Table des descripteurs

- Le noyau alloue des ressources I/O pour chaque processus sous forme de descripteurs de fichiers. La plupart des IPC sont représentés par des descripteurs de fichiers.



BY-SA
©2024 CentraleSupélec - Dominique Marcadet

Pipes

- Les pipes sont des IPC unidirectionnels

Commande shell

```
ls | grep toto
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exemple pipe (1)

- On veut un pipe dans lequel un processus fils peut écrire des messages au processus père

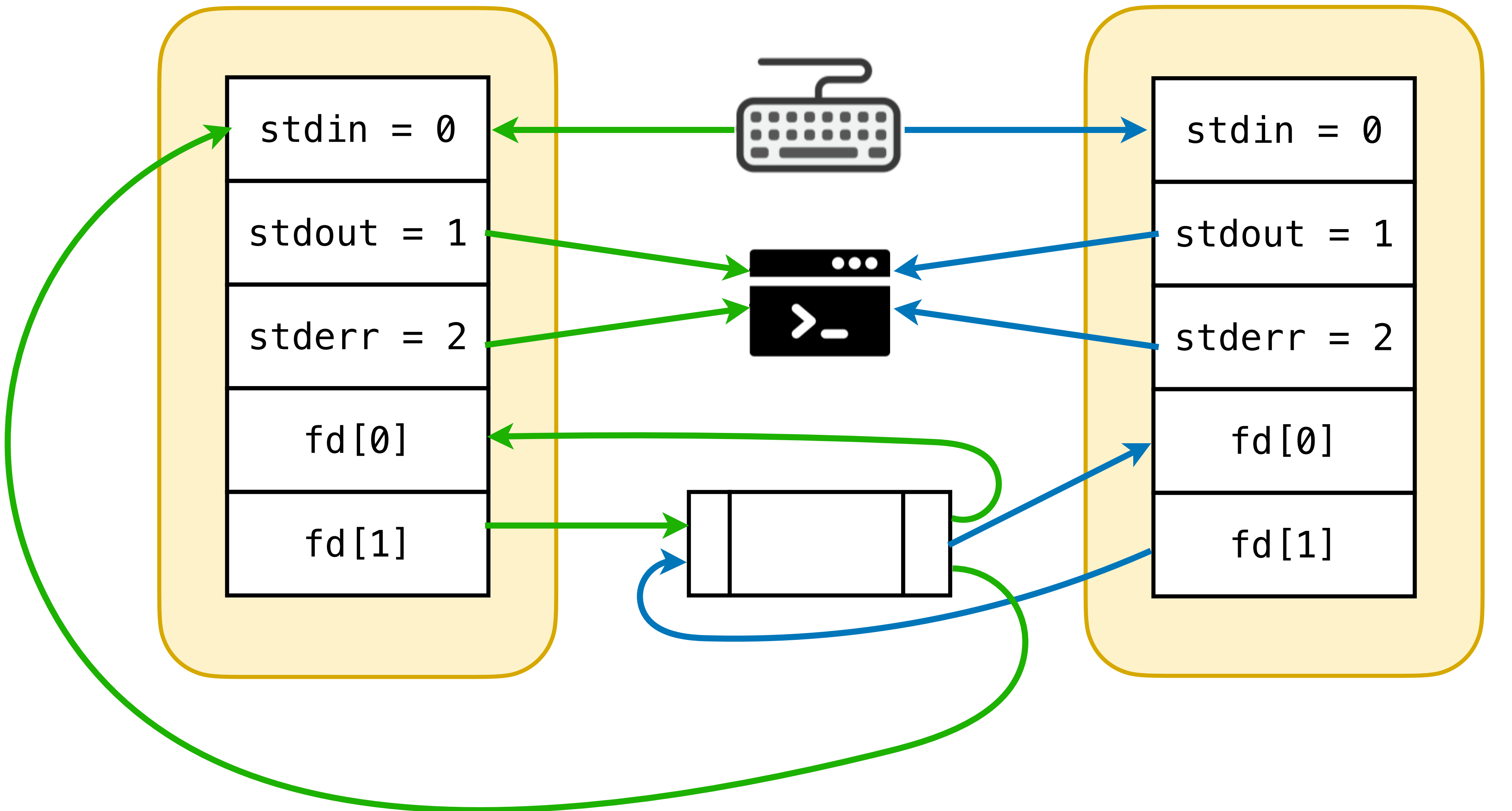
```
// Tableau pour 2 descripteurs de fichier
int fds[2];
// Création d'un pipe, les bouts sont stockés dans fd
pipe( fds );
pid_t pid = fork();
if( pid == 0 ) {      // Processus fils
    close( fds[0] );  // Fermeture du côté lecture
    // ...
} else {              // Processus père
    close( fds[1] );  // Fermeture du côté écriture
    // On raccroche stdin au côté lecture du pipe
    dup2( fds[0], STDIN_FILENO );
    // ...
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exemple pipe (2)

CentraleSupélec

`dup2( fd[0], stdin )` `fork()` `close( fd[0])`



BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Démo

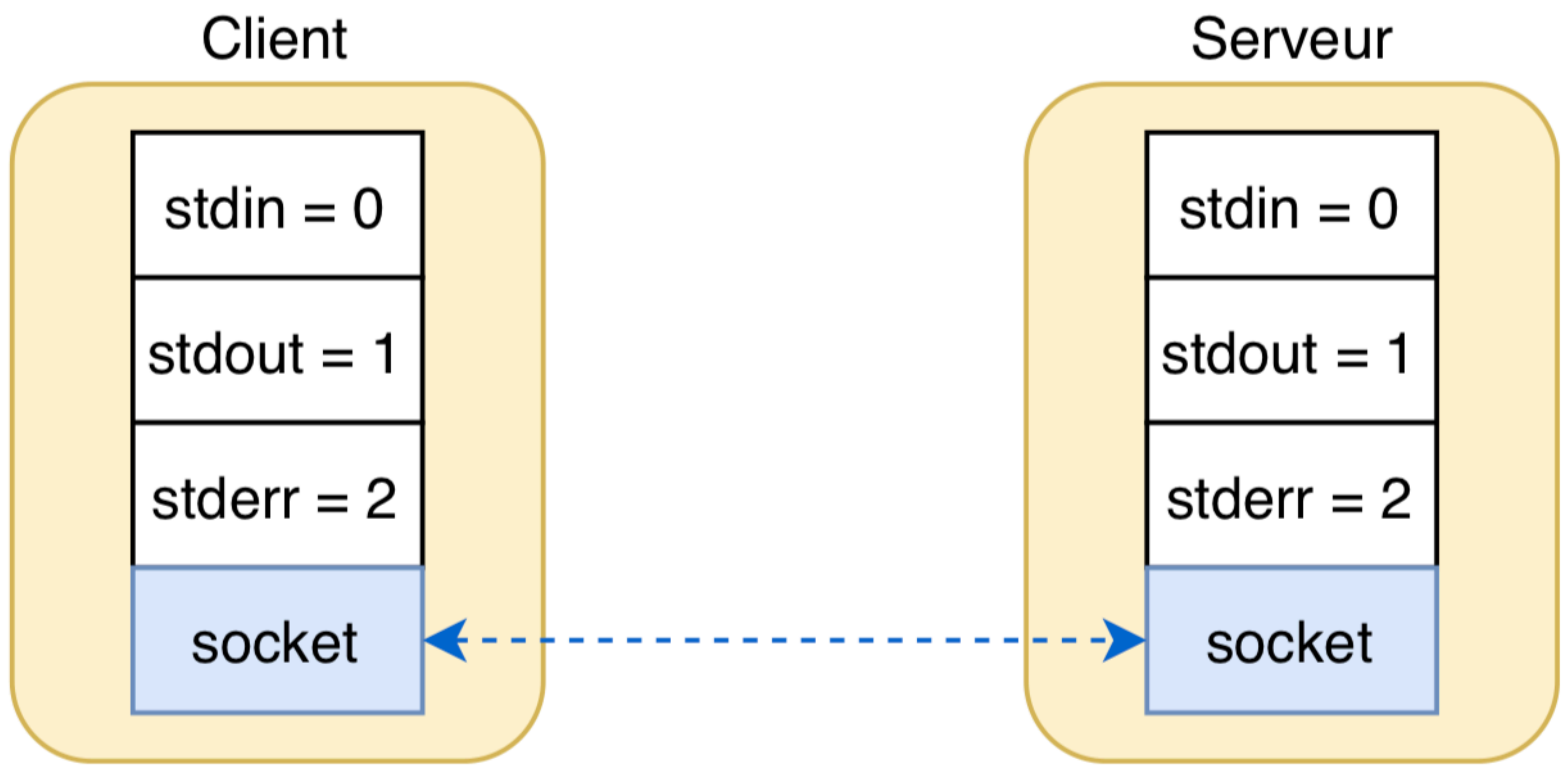
GitLab

`cours/progsyst/Pipe.c`

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Sockets

- Les sockets sont des IPC bidirectionnels pouvant utiliser les protocoles réseaux (UDP, TCP…)



©2024 CentraleSupélec - Dominique Marcadet

Fonctions sockets (1)

- Création d'un socket

```
int socket( int domain, int type, int protocol );
```

- Liaison de la socket à une adresse IP et un port

```
int bind( int fd, const struct sockaddr * addr,
          socklen_t addrlen );
```

- Description de l'adresse

```
struct sockaddr_in address;
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY; //localhost
address.sin_port = htons( PORT );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Fonctions sockets (2)

- Attente de connexion par le serveur

```
int listen( int sockfd, int backlog );
```

- Ouverture de la connexion par le client

```
int connect( int fd, const struct sockaddr * addr,
             socklen_t addrlen );
```

- Acceptation par le serveur de la connexion

```
int accept( int fd, const struct sockaddr * addr,
            socklen_t addrlen );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Ordre des octets

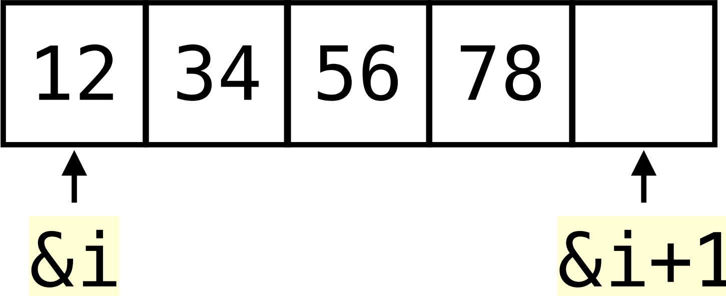
- Le nombre sur 32 bits

```
int i = 0x12345678;
```

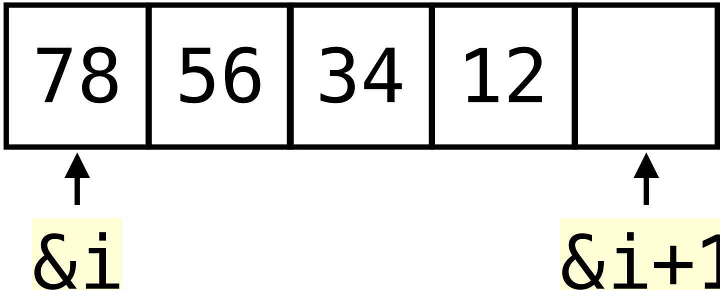
https://www.ietf.org/rfc/ien/ien137.txt

https://fr.wikipedia.org/wiki/Les_Voyages_de_Gulliver

- peut être stocké en *gros-boutiste* (big-endian)



- ou en *petit-boutiste* (little-endian)



- Des ordinateurs qui échangent des données doivent se mettre d'accord sur l'ordre des octets utilisé : fonctions `htonl()`, `htons()`, `ntohl()`, `ntohs()`

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

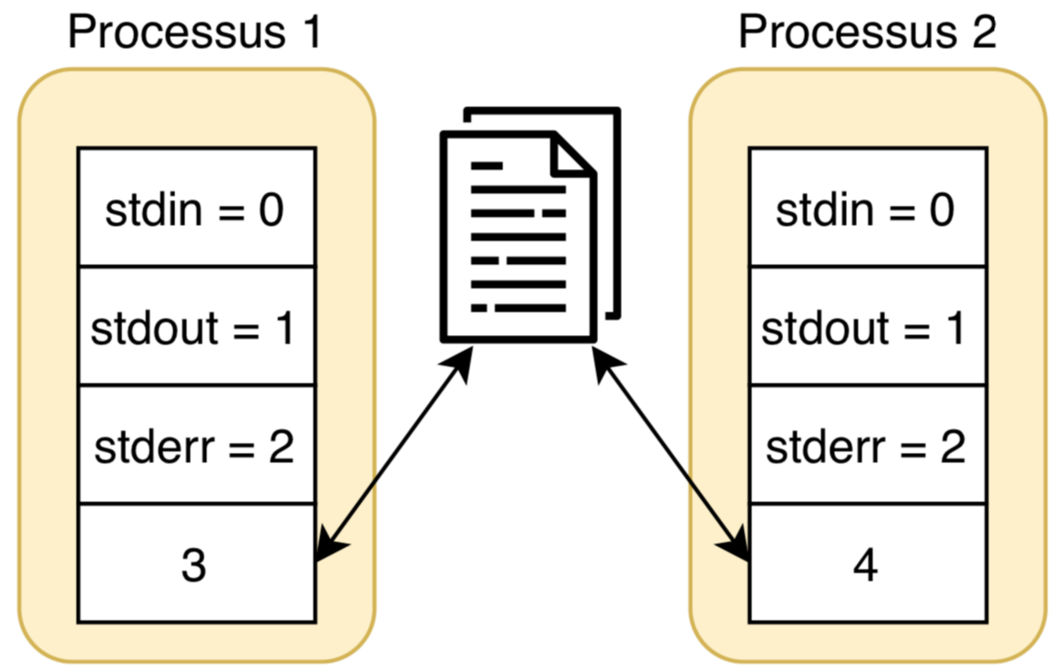
Fichiers partagés

- Ouverture du même fichier par plusieurs processus

```
int open( const char * name, int flags, mode_t mode );
```

- Mappage possible en mémoire

```
void * mmap(
   void * addr, size_t length, int prot,
   int flags, int fd, off_t offset );
```



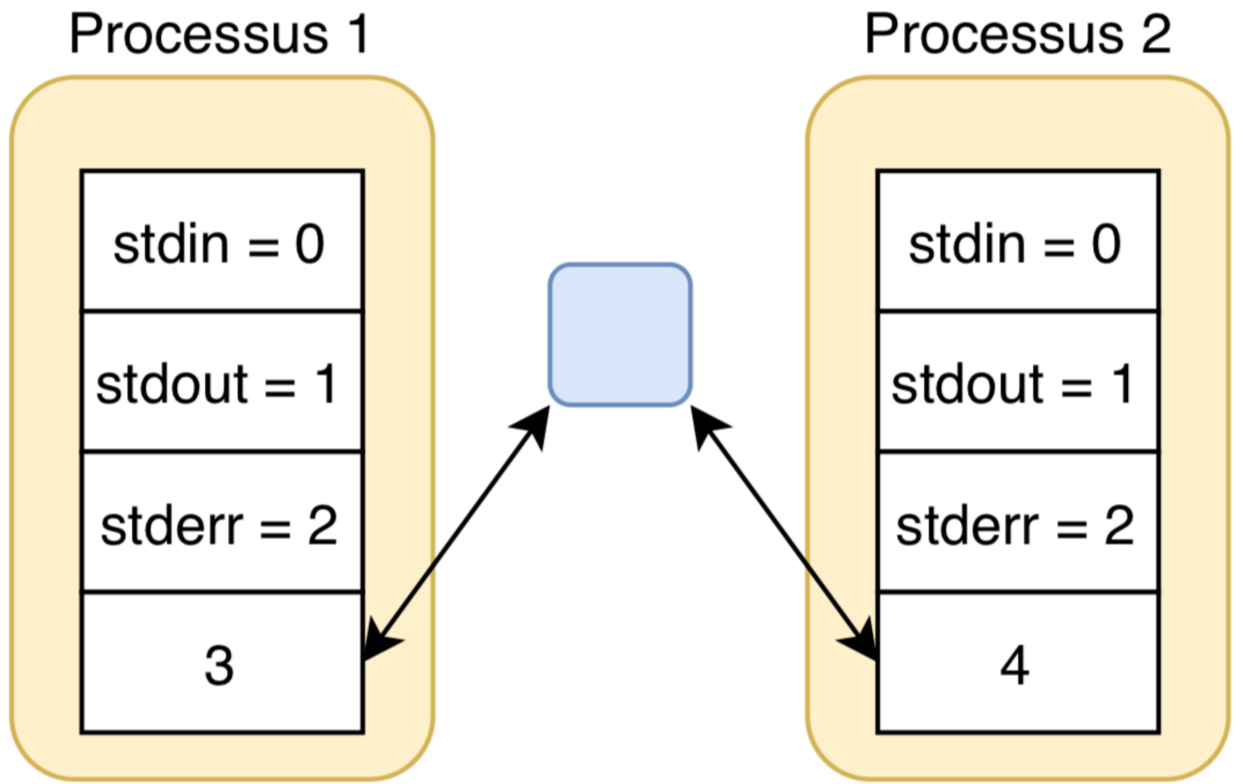
©2024 CentraleSupélec - Dominique Marcadet

Mémoire partagée

- Une mémoire partagée est nommée, comme un fichier

```
int shm_open( const char * name, int oflag, mode_t mode );
```

- Mappage en mémoire



(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

CentraleSupélec

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Processus légers (threads)

- Le temps de création (et de changement) d'un processus est élevé (… mémoire virtuelle … mémoire cache …).

- Un ***thread*** a son propre compteur ordinal, sa propre pile ... mais n'existe qu'à l'intérieur d'un processus et utilise ses ressources (espace mémoire en particulier).

- Un thread a aussi un état (actif, prêt, bloqué), un bloc de contexte (plus petit !).

- C'est aussi le système d'exploitation qui gère les threads.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Threads POSIX (1)

- Creation

```
int pthread_create(
  pthread_t * thread_handle,
  const pthread_attr_t * attribute,
  void * (*thread_function)( void * ),
  void * arg
);
```

- Terminaison

```
void pthread_exit(
  void * ptr
);
```


©2024 CentraleSupélec - Dominique Marcadet

Threads POSIX (2)

- Obtenir son identifiant de thread

```
pthread_t pthread_self();
```

- Attente de terminaison (sinon le thread continue d'exister)

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

- Détachement (il n'est plus possible de l'attendre)

```
int pthread_detach(
  pthread_t thread
);
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

C++ : std::thread

```
class thread {
public:
  template< class F > explicit thread( F f );

  void join();
  void detach();
  id get_id() const;
  // ...
};

namespace this_thread {
  id get_id();
  void sleep_until( /* ... */ );
  void sleep_for( /* ... */ );
  // ...
}
```

C++20 : `std::jthread`

cc BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Utilisation std::thread

```
void f1( int   n ) { /* ... */ }
void f2( int & n ) { /* ... */ }
struct F3 {
  void operator()() { /* ... */ }
  // ...
};

int main() {
  int n1{ 5 }, n2{ -2 };
  std::thread t1{ f1, n1 };
  std::thread t2{ f2, std::ref( n2 )};
  F3 f3;
  std::thread t3{ f3 };
  t1.join(); t2.join(); t3.join();
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

CentraleSupélec

Démo

GitLab

`cours/progsyst/Thread.cpp`

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

std::async et std::future

```
int foo()
{
    std::this_thread::sleep_for( 10ms );
    return 42;
}

int main()
{
    std::future< int > f{ std::async( foo )};
    while( f.wait_for( 1ms ) ==
                std::future_status::timeout ) {
        std::cout << "waiting\n";
    }
    std::cout << "got " << f.get() << "\n";
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

CentraleSupélec

Démo

GitLab

`cours/progsyst/Async.cpp`

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
  - Introduction
  - Historique
- Programmation sous Unix
  - Fichier
  - Processus
  - Communication
  - Threads
  - Synchronisation

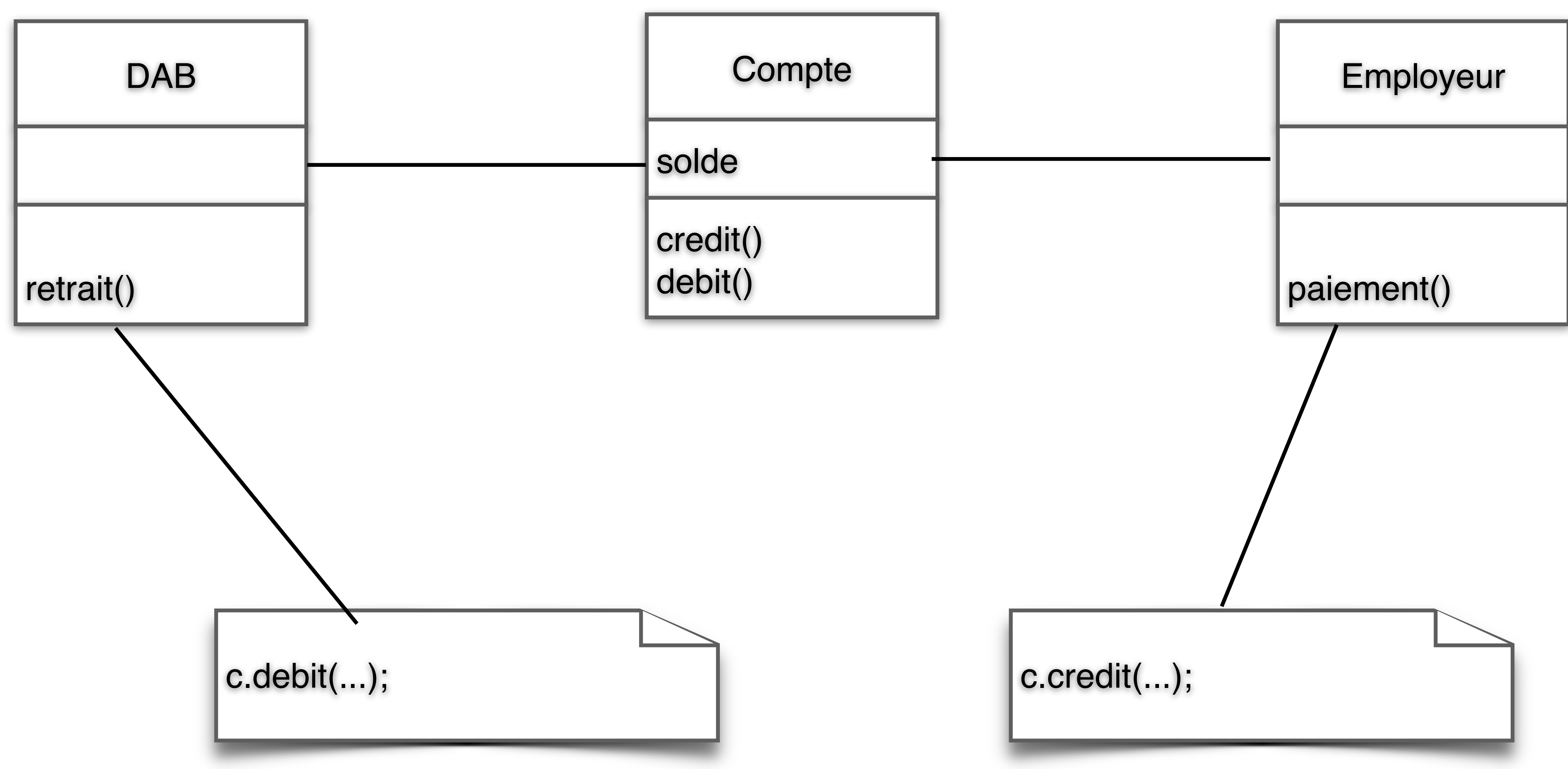
©2024 CentraleSupélec - Dominique Marcadet (CC BY-SA)

Plan

- Présentation
- Programmation sous Unix
  - Fichier
  - Processus et threads
  - Communication
  - Synchronisation
    - Problématique
    - Synchronisation de threads
    - Synchronisation de processus

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Problème de la variable partagée



©2024 CentraleSupélec - Dominique Marcadet

Version incorrecte

```
class Compte {
public:
  void credit( double montant
    solde += montant;
  }

  void debit( double montant
    solde -= montant;
  }

private:
  double solde;
}
```

1. Chargement de l'attribut `solde` (en mémoire) dans un registre du processeur
2. Ajout de `montant` au registre
3. Transfert du contenu du registre en mémoire (dans l'attribut `solde`)

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exécution correcte

Thread DAB

- LDR solde
- SUB 20
- STR solde

Thread Employeur

- LDR solde
- ADD 50
- STR solde

80	130	130
Registre		Registre

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exécution incorrecte

Thread DAB

Go

Thread Employeur

- LDR solde
- SUB 20
- STR solde

- LDR solde
- ADD 50
- STR solde

80	80	150
Registre		Registre

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Le dîner des philosophes

- Interblocage
- Famine

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Définitions

- Ressource critique : ressource qui ne doit pas être accédée par plusieurs processus simultanément

- Section critique : partie(s) de code accédant à une ressource critique

- Exclusion mutuelle : mécanisme permettant l'accès à une ressource critique

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Propriétés

- Sûreté : au plus un seul processus utilise la ressource à un moment donné
- Vivacité :
  - Un processus obtient l'accès immédiat à une ressource disponible
  - Un processus obtient l'accès à la ressource au bout d'un temps fini
- Efficacité : un processus en attente d'une ressource n'utilise pas le processeur (il est bloqué)

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Solutions

- Il existe des algorithmes (algorithme de Peterson, algorithme de Dekker), mais ceux-ci imposent une attente active (efficacité non respectée)

- C'est le système d'exploitation qui peut mettre un thread ou un processus à l'état bloqué, donc c'est lui qui doit fournir le service

- Une implémentation efficace impose l'existence d'une transaction de lecture/écriture au niveau du bus et l'existence d'une instruction correspondante sur le processeur (test-and-set, compare-and-swap…)

CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
- Programmation sous Unix
  - Fichier
  - Processus et threads
  - Communication
  - Synchronisation
    - Problématique
    - Synchronisation de threads
    - Synchronisation de processus

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Mutex POSIX

- Création

```
int pthread_mutex_init(
  pthread_mutex_t * mutex_lock,
  const pthread_mutexattr_t * lock_attr
);
```

- Acquisition du mutex

```
int pthread_mutex_lock(
  pthread_mutex_t * mutex_lock
);
```

- Libération du mutex

```
int pthread_mutex_unlock(
  pthread_mutex_t * mutex_lock
);
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Compte avec mutex

```
class Compte {
public:
  Compte() { pthread_mutex_init( &lock, NULL ); }

  void credit( double montant ) {
    pthread_mutex_lock( &lock );
    solde += montant;
    pthread_mutex_unlock( &lock );
  }

  void debit( double montant ) {
    pthread_mutex_lock( &lock );
    solde -= montant;
    pthread_mutex_unlock( &lock );
  }

private:
  double solde;
  pthread_mutex_t lock;
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Producteur - Consommateur

- Le producteur dépose des données dans une boîte à lettres de taille fixe.
- Le consommateur les retire.
- Synchronisation :
  - le consommateur doit attendre qu'une donnée soit présente dans la boîte à lettres ;
  - le producteur doit attendre d'avoir de la place dans la boîte à lettres.
- Variantes : plusieurs producteurs et consommateurs.

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Moniteur

- Hansen (1973), Hoare (1974)

- Un moniteur est un module constitué de :

  - fonctions d'accès à la ressource critique en exclusion mutuelle (un verrou)

  - condition booléenne et services permettant le blocage d'un processus/thread (avec libération du verrou) si la condition n'est pas remplie et réactivation par un autre processus/thread quand la condition peut être remplie (l'accès à la ressource critique reste soumise au verrou)

- Mécanisme de base de synchronisation en Java (classe `Object`)

BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Variable de condition POSIX (1)

- Création

```c
int pthread_cond_init(
  pthread_cond_t * cond,
  const pthread_condattr_t * attr
);
```

- Destruction

```c
int pthread_cond_destroy(
  pthread_cond_t * cond
);
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Variable de condition POSIX (2)

- Attente

```c
int pthread_cond_wait(
  pthread_cond_t * cond,
  pthread_mutex_t * mutex
);
```

- Signalisation

```c
int pthread_cond_signal(
  pthread_cond_t * cond
);
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exemple POSIX (1)

```
pthread_cond_t  data_place_cond;
pthread_mutex_t data_place_mutex;
volatile int data_available = 0;

int main() {
  pthread_t c, p;

  pthread_cond_init( &data_place_cond, NULL );
  pthread_mutex_init( &data_place_mutex, NULL );

  pthread_create( &c, NULL, &consumer, NULL );
  pthread_create( &p, NULL, &producer, NULL );
  pthread_join( c, NULL );
  pthread_join( p, NULL );
}
```


cc BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exemple POSIX (2)

```
void * producer( void * producer_thread_data ) {
  while( !done() ) {
    create_data();

    pthread_mutex_lock( &data_place_mutex );
    if( data_available == 1 ) {
      pthread_cond_wait(
        &data_place_cond,
        &data_place_mutex );
    }
    put_data_into_place();
    data_available = 1;
    pthread_cond_signal( &data_place_cond );
    pthread_mutex_unlock( &data_place_mutex );
  }
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Exemple POSIX (3)

```
void * consumer( void * consumer_thread_data ) {
  while( !done() ) {
    pthread_mutex_lock( &data_place_mutex );
    if( data_available == 0 ) {
      pthread_cond_wait(
        &data_place_cond,
        &data_place_mutex );
    }
    get_data_from_place();
    data_available = 0;
    pthread_cond_signal( &data_place_cond );
    pthread_mutex_unlock( &data_place_mutex );

    process_data();
  }
}
```


©2024 CentraleSupélec - Dominique Marcadet

C++ : std::mutex

```
class mutex {
public:
   void lock();
   bool try_lock();
   void unlock();
   // ...
};
class recursive_mutex { /* ... */ };

class timed_mutex {
public:
   bool try_lock_for( /* ... */ );
   bool try_lock_until( /* ... */ );
   // ...
};
class recursive_timed_mutex { /* ... */ };
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

C++ : std::lock_guard

```
template< typename Mutex >
class lock_guard {
public:
  explicit lock_guard( Mutex & m );
  ~lock_guard();
  // ...
};
template< typename Mutex >
class unique_lock {
public:
  void lock();
  bool try_lock();
  void unlock();
  bool try_lock_for( /* ... */ );
  bool try_lock_until( /* ... */ );
  // ...
};
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

C++ : std::condition_variable

```
class condition_variable {
public:
   void notify_one();
   void notify_all();

   void wait( unique_lock< mutex > & lock );
   bool wait_until( /* ... */ );
   bool wait_for( /* ... */ );
   // ...
};

class condition_variable_any { /* ... */ };
```


©2024 CentraleSupélec - Dominique Marcadet

C++ : exemple (1)

```
std::condition_variable data_place_cond;
std::mutex              data_place_mutex;

volatile bool data_available{ false };

int main() {
  std::thread c{ consumer{} }, p{ producer{} };

  c.join();
  p.join();
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

C++ : exemple (2)

```
struct producer {
  void operator()() {
    while( !done() ) {
      create_data();
      {
        std::unique_lock lock{ data_place_mutex };
        if( data_available ) {
          cond_place_empty.wait( lock );
        }
        put_data_into_place();
        data_available = true;
        cond_place_full.notify_one();
      }
    }
  }
};
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

C++ : exemple (3)

```
struct consumer {
  void operator()() {
    while( !done() ) {
      {
        std::unique_lock lock{ data_place_mutex };
        if( ! data_available ) {
          cond_place_full.wait( lock );
        }
        get_data_from_place();
        data_available = false;
        cond_place_empty.notify_one();
      }
      process_data();
    }
  }
};
```


©2024 CentraleSupélec - Dominique Marcadet

Démo

GitLab

`cours/progsyst/ProdCons.cpp`

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Variable partagée (KO)

```cpp
void f( int & x ) { /*...*/ x += 3; }

int main() {
  int n{ 2 };
  std::thread t1{ f, std::ref( n )};
  std::thread t2{ f, std::ref( n )};
  t1.join();
  t2.join();
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Variable partagée (OK)

```
class ProtectedInt {
public:
  ProtectedInt( int n ) : n_{ n } {}
  void operator+=( int i ) {
    std::lock_guard< std::mutex > l( mutex_ );
    n_ += i;
  }
private:
  int n_;      std::mutex mutex_;
};

void f( ProtectedInt & x ) { /*...*/ x += 3; }

int main() {
  ProtectedInt n{ 2 };
  std::thread t1{ f, std::ref( n )};
  std::thread t2{ f, std::ref( n )};
  t1.join(); t2.join();
}
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

std::atomic

```
void f( std::atomic_int & x ) {
  /*...*/
  x += 3;
}

int main() {
  std::atomic_int n{ 2 };
  std::thread t1{ f, std::ref( n )};
  std::thread t2{ f, std::ref( n )};
  t1.join();
  t2.join();
}
```


CC BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Plan

- Présentation
- Programmation sous Unix
  - Fichier
  - Processus et threads
  - Communication
  - Synchronisation
    - Problématique
    - Synchronisation de threads
    - Synchronisation de processus

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Différent des threads ?

- La synchronisation entre processus ne peut être que gérée par le système d'exploitation
    - La frontière d'un processus (espace mémoire) est aussi une frontière pour les langages
- C'est aussi le cas pour la communication entre processus !
- Les systèmes d'exploitation sont trop variés (parce que les besoins sont multiples) pour imposer une solution unique

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Solutions déjà vues

- Certains mécanismes de communication intègrent un mécanisme de synchronisation
  - pipe
  - socket
- D'autres mécanismes de communication offrent des solutions spécifiques
  - fichiers partagés : un blocage (avec `fcntl()`) peut être utilisé pour interdire l'accès à un fichier

©2024 CentraleSupélec - Dominique Marcadet

Solutions pour les autres cas

- Par exemple, comment gérer la synchronisation dans le cas d'une communication par mémoire partagée ?
- Plusieurs solutions
  - Se limiter à un seul système d'exploitation
    - il faut quand même gérer les différentes variantes et versions
  - Écrire du code spécifique à chaque système d'exploitation
    - difficulté de maintenance
  - Utiliser une bibliothèque qui offre une interface unifiée
    - l'ensemble des possibilités ne sera pas disponible

BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Sémaphore (1)

- Djikstra (1965)
- Un sémaphore est constitué :
  - D’un `compteur`
  - D’une `file` d’attente de processus (de blocs de contexte)
  - De 3 opérations atomiques (non interruptibles)

cc BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Sémaphore (2)

- `init( val )`
  - initialise le `compteur` à `val` et la `file` à vide
- `wait()`
  - décrémente le `compteur` ; si celui-ci est strictement négatif, bloque le processus et le met dans la `file` d'attente
- `signal()`
  - incrémente le `compteur` ; si celui-ci est négatif ou nul, libère un processus de la `file` d'attente

(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Linux : sémaphore anonyme

- Les sémaphores anonymes sont accessibles uniquement entre processus père-fils et les threads

  - Création

```
int sem_init(
  sem_t * sem,
  // pshared == 0 ? threads : processus
  int pshared,
  unsigned int value
);
```

  - Destruction

```
int sem_destroy( sem_t * sem );
```


©2024 CentraleSupélec - Dominique Marcadet

Linux : sémaphore nommé

- Les sémaphores nommés sont implémentés par un fichier et sont accessibles par tout les processus/threads

  - Création

```
sem_t * sem_open(
   char * name,
   int oflag,
   mode_t mode,
   int val
);
```

  - Destruction

```
int sem_destroy( sem_t * sem );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet

Linux : opérations sur sémaphores

- Ces deux types de sémaphores offrent les services suivants :
  - Demande d'un jeton

```
int sem_wait( sem_t * sem );
```

  - Libération d'un jeton

```
int sem_post( sem_t * sem );
```


(cc) BY-SA ©2024 CentraleSupélec - Dominique Marcadet