000 Présentation générale

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Été 2021

Aujourd'hui

- Présentation du cours et des modalités
- Posez les questions au fur et à mesure
- Séance courte ?

Cette séance est enregistrée (normalement)

- *Mutez* vos micros pour éviter les bruits de cuisine
- Allumez votre webcam, je me sentirai moins seul

Osez l'interactivité

Pas de protocole strict

- Intervenez oralement !
- Je ne regarde pas toujours le *chat* ou les mains virtuelles levées

Je me présente, Jean Privat

Langages, compilation, systèmes

- 07–...: Professeur, UQAM (QC)
- 06–07: Postdoc, Purdue (ÉU)
- 02–06: Ph.D, Montpellier (FR)

Intérêts

- La programmation : de l'assembleur aux langages à objets
- L'info système : compilateurs, VM et systèmes d'exploitation

Contactez-moi

- privat.jean@uqam.ca

Plan

INF3173, principes des systèmes d'exploitation

Principes des systèmes d'exploitation

Cours « traditionnel »

- Existe dans la plupart des formations universitaires
- Depuis au moins 40 ans

Approche dans INF3173 à l'UQAM

- Voir et comprendre les concepts, les besoins, les solutions
- Détailler un mise en œuvre (mécanique et politique)
  - On utilise Unix/Linux
- Expérimenter de façon programmative (espace utilisateur)
  - Programmation système

Exemple d'objectif

- Être capable de lire, de comprendre et d'appliquer une *manpage* de la section 2 en toute confiance

Mise à jour INF3173

Automne 2020 : ne jamais gâcher une bonne crise

Profiter des cours en ligne pour faire une grosse mise à jour

- Prise en compte de INF1070 (on ne part plus de zéro)
- Mise à jour du contenu, plus complet (on est dans les années 20)
- Production de matériel pédagogique (beaucoup d'énergie)
- Formule pédagogique: les cours en ligne c'est compliqué à monter autant que le jeu en vaille la chandelle

3e itération

- Contenu amélioré & disponible d'avance
- On tente quand même de nouvelles choses !
- Rythme plus régulier (réorganisation des semaines)

$\rightarrow$ Pardonnez néanmoins les problèmes techniques et les petites mises à jour inévitables

Prérequis

- INF1070 : utilisation et administration des systèmes informatiques

$\rightarrow$ Shell, fichiers et processus

- INF2171 : fonctionnement du processeur et de la mémoire

$\rightarrow$ Rassurez-vous, on ne programmera pas (trop) en assembleur

- INF3135 : programmation C *de qualité*

$\rightarrow$ Par contre on programmera *beaucoup* en C

INF3173 difficile?

- INF3173 est un cours **avancé** (cours préalables nécessaires)
- INF3173 est un cours **dense** (plein de choses à voir)
- INF3173 est un cours de **programmation** (en grande partie)

Difficultés du cours

- Les lacunes éventuelles dans les cours préalables
- Ne pas travailler au fur et à mesure
  et se laisser déborder par la matière
- Programmer sans comprendre et sans rigueur
  stackoverflow n'est pas la solution

Formule pédagogique

Classes normales et inversées

Classe normale

- Je fais la leçon en classe
- Vous faites les exercices à la maison

Classe inversée

- Vous faites la leçon à la maison
- On fait les exercices en classe

Classe inversée INF3173

Plusieurs étages

- Capsules + diapos + exemples + sujets d'exercices : à l'avance
  - Quiz d'autovalidation
- Laboratoires synchrones avec le démonstrateur
  - Vous faire pratiquer
- Travaux pratiques
  - Vous faire maîtriser

Capsules

- Moins de 2h par semaine... en moyenne
- Semaines pas forcément égales : prenez de l'avance
- S'écoutent bien en x1.5
- Et on peut réécouter !

Faites à l'automne 2020

- Temps infini à faire
- Petits défauts techniques, restant de fautes et quelques lapsus
- Youtubeur c'est un métier

Corrélation

- L'écoute des capsules vs. la réussite au cours
- Attention : corrélation $\neq$ causalité

Diapositives

- Spartiates (c'est beamer, pas keynote)
- Support des capsules
- Relativement détaillées (pas juste un support visuel)
- Contient des hyperliens vers des ressources externes

$\rightarrow$ Les diapos ne sont pas des notes de cours !

Différence avec les capsules?

- Révisées pour l'été 2021
- Corrigées par rapport au capsules

$\rightarrow$ Ne contient pas les explications, exemples et expériences

Marqueurs

- ★ Notion fondamentale
- 🔍 Notion avancée (ou parfois optionnelle)

Semainier

Nomenclatures des diapos et capsules

- Format: « ABC Titre »
- Exemple: « 120 Appels système »
- A : numéro de chapitre (0 pour présentation générale)
- B : numéro de section (0 pour intro du chapitre)
- C : numéro de partie (>0 quand la section est trop longue)

Les semaines

- 15 semaines $\rightarrow$ 12 semaines de capsules
- Le nombre de capsules par semaine est variable

$\rightarrow$ Point d'entrée unique https://inf3173.uqam.ca/ maintenu à jour

Ressources supplémentaires

Les pages de man

- RTFM *read the fucking manual* (regarde ton fichu manuel)

Deux ouvrages optionnels

- *Operating Systems Concepts*, Silberschatz, 10e édition (2018)
- *Modern Operating Systems*, Tanenbaum, 4e édition (2014)

Complets et intéressants !

Mattermost

Groupe privé dédié

- https://mattermost.info.uqam.ca/inf3173-e21
- Lien d'invitation envoyé par courriel et sur le site web

Nétiquette

- Merci de garder les canaux propres
- Et de respecter les principes de base de la nétiquette
- Les discussions, les questions et les erreurs sont permises
- Les abus et la mauvaise foi ne seront pas tolérés
  Mais il n'y a pas de raison que ça arrive...

Canaux

- `/var/log`
  - Discussions générales sur le cours, les questions, etc.
- `/dev/random`
  - Pour le social, l'actualité en informatique, les mèmes, etc.
- Des canaux spécifiques seront crées (Cours, Lab, TP, etc.)

Séances magistrales synchrones via zoom ?

- Pas fameux à l'automne 2020
- Pas beaucoup mieux à l'hiver 2120

Symptômes

- Peu de participants
- Peu d'interaction
- Peu de questions
- Peu de visionnement des séances enregistrées
- ... Peu utile?

Été : modalité différente

- Visionnez les capsules et commencez les laboratoires quand vous voulez

Sur mattermost

- Si vous avez des question, inutile d'attendre
- Posez-les dans le canal public dédié
- On essaiera de vous répondre rapidement
- Sauf pendant les périodes de quiz!

Séance de disponibilité

- Durant la séance de cours (mardi de 9h30 à 12h)
- Je pourrais répondre plus rapidement
- Panopto-party
  - Regardez les capsules à ce moment-là
  - Posez vos question en direct

Laboratoires

Synchrone et par Zoom

- Exercices à faire ensemble
- Vous programmez et posez des questions
- On vous donne des indices, des solutions partielles puis des solutions plus complètes
- Une approche différente et complémentaire de la matière
- Interactivité !

Les laboratoires sont obligatoires

- Exercices d'approfondissement et matière originale
- Qui sera éventuellement évaluée dans les devoirs/TP/quiz

Votre travail **avant** le lab

- Écouter les capsules
  - Posez vos questions sur mattermost
  - En particulier le mardi matin
- Survolez les diapos
  - Cliquez sur les liens
  - Essayez de répondre aux questions
- Essayer les exemples
  - Compilez, exécutez, modifiez, comprenez
- Commencez le lab
  - Lisez l'énoncé
  - Faites la mise en place et les premiers exercices
  - Ce qui permet d'avoir des séances de lab plus utiles

Linux

C'est le système qu'on utilisera

- En classe
- Pour les labs
- Pour les travaux

$\rightarrow$ Parce qu'il est bien, libre, gratuit, documenté et répandu

Vous devez avoir accès

- À un système avec une distribution Linux récente
- Si vous pouvez être root dans l'espace de noms principal (pas dans un conteneur), c'est mieux, mais pas obligatoire

Support Linux fourni « au mieux » et par les pairs

- Par les auxiliaires
- Par les moniteurs de programme
- Par l'AGEEI

Ce qui est acceptable

Sont acceptés

- Système natif (et double-boot)
- Machine virtuelle
- Serveur sur labunix, comme `java.labunix.uqam.ca`

Ne sont pas acceptés

- Windows et WSL
- macOS
- Android

$\rightarrow$ On aura besoin des vrais appels système Linux spécifiques

$\rightarrow$ Et d'un environnement Unix traditionnel
POSIX, shell, utilitaires, compilateurs, etc

Évaluations

Évaluations

- 12 quiz rapides
  - Mardi 12h à 23h55
  - 20 minutes max
  - 12% (1% chacun)
- 3 travaux pratiques
  - Un petit (8%)
  - Un moyen (20%)
  - Un gros (20%)
  - Sur plusieurs semaines chacun
- 2 devoirs à la maison
  - Asynchrones
  - Sur deux jours
  - 20% chacun
- Validations orales en fin de session

$\rightarrow$ Travail individuel

Quiz

- Choix de réponses (quiz moodle)
- 20 minutes chronométrées
- Ouvert le mardi de 12h00 à 23h55

Autovalidation

- Après les capsules et la séance de disponibilité
- Mais avant les labs
- Questions souvent simples

$\rightarrow$ Objectif de 80% à 100% de bonnes réponses

$\rightarrow$ Moins de 80% : c'est inquiétant

Travaux pratiques

- TP0 entrainement et/ou remise à niveau
- TP1 fichiers et/ou processus
- TP2 communication et/ou synchronisation

Contenu

- Mise en pratique des concepts/services des systèmes d'exploitation
- Programmation système (C&Linux) coté utilisateur (POSIX&Linux)
- Exemple : développement de petits utilitaires
- Objectifs : exactitude, robustesse, élégance

$\rightarrow$ Petits programmes mais de qualité

Travaux pratiques

- INF3173, dernier cours de programmation obligatoire du BIGL?
  - Vous n'êtes plus des débutants en programmation
- Votre programme doit être fonctionnel
  - Le cahier des charges est précis et rigoureux
  - Un programme qui fonctionne presque $\approx$
    Un parachute qui fonctionne presque
- Votre programme doit être robuste
  - En programmation système, on a rarement droit à l'erreur
  - La robustesse est la politesse des outils système
- Votre programme devrait être simple et élégant
  - Souvent les étudiants se compliquent la vie inutilement
  - Temps passé $\neq$ qualité

TP faisables

- Programmes courts
  - Quelques centaines de ligne
  - Un seul fichier source
- Pas ou peu de conception
  - Pas de domaine métier à modéliser
  - Pas de UML
- Pas ou peu d'algorithmes ou de structures de données
  - Et encore moins d'IA
- Pas ou peu de complexités non nécessaire
  - On essaye de limiter les difficultés liées aux C
  - Même si ça rend les programmes finaux moins intéressants

Pour réussir les TP

- Lisez l'énoncé
- Respectez les exigences
- Profitez de l'infrastructure de test éventuelle
- Ayez fait (pour de vrai!) les laboratoires
- Programmez robuste et simple
- Ne commencez pas la veille
- Utilisez les services du monitorat de programme pour de l'aide à la programmation (INF3135, INF2120, INF1120, etc.)
- Relisez l'énoncé

Par soucis d'équité dans les TP

- Posez vos questions publiquement sur le canal dédié mattermost
- Ne divulguez pas vos solutions sur le mattermost (ni ailleurs)
- Je ne réponds à aucune question sur un TP dans les quatre (4) jours précédents la remise
- Aucun rendu hors consigne ou délais ne sera évaluée

Devoirs à la maison

Modalités

- 48h, mardi et mercredi
- Pas de cours ni de lab les semaines des devoirs
- 3h de temps (max estimé) pour le faire

$\rightarrow$ N'utilisez pas les 48h à temps plein

- Le reste de la semaine pour étudier la matière suivante

Contenu

- Questions ouvertes et de de réflexion
- Étude des cas ou de programmes
- Mise en situation
- Etc.

Devoirs à la maison

Pour réussir

- Visionnez les capsules, posez des question, participez aux labs
- Ne commencez pas à visionner la veille

Par soucis d'équité

- Aucune négociation de note
- Si vous vous estimez lésé dans l'évaluation de votre travail
  - Procédures de modification de note et/ou révision de notes
  - Gérés au niveau du départementale
  - Instruit de manière équitable

Validations orales

- Objectif: s'assurer de l'intégrité des évaluations
- Prise de rdv préalable: je vous contacterai
- Pas de note: c'est soit OK, soit infraction académique

$\rightarrow$ Pas besoin de paniquer pour autant

Liens

- Plan de cours
- Site web du cours
- Mattermost
- Moniteurs de programme (support académique)

100 Introduction aux systèmes d'exploitation

INF3173

Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Plan

1. Préalables (et contexte)

2. Objectif du cours

Préalables (et contexte)

Ce qui est attendu des préalables

INF1070 Utilisation des systèmes informatiques

- Environnement Unix et shell
- Systèmes de fichiers (types, droits, etc.)
- Commandes, programmes et processus (redirection, tubes, etc.)
- `man` et RTFM

INF2171 Organisation des ordinateurs et assembleur

- CPU: Instruction, exécution, registres
- Mémoire: code machine, données, pile, tas, adressages, représentation de l'information

INF3135 Construction et maintenance de logiciels

- Programmation C: pointeurs, allocation, bibliothèques
- Qualité logicielle: exactitude, robustesse

INF1070 en très vite

Système de fichiers

- Fichier = forme libre de données stockées
- Indépendance au matériel et extensible
- Arborescence : chemins relatifs et absolus
- Sécurité : utilisateurs et droits
- `cd`, `ls`, `cat`, `>`, `rm`, `grep`, etc.

Commandes et processus

- Processus = programme en cours d'exécution
- PID, PPID, utilisateur, etc.
- Utilise mémoire et processeur (entre autres)
- `sh`, `ps`, `kill`, `|`, `&`, `uptime`, `free`, `PATH`, etc.

UNIX et Linux

INF2171 en très vite

CPU - Unité centrale de traitement

- Unité de contrôle + unité arithmétique et logique
- Exécute des instructions jusqu'à l'arrêt de l'ordinateur
- N'est qu'une machine

$\rightarrow$ Ne sait pas ce qu'est un SE, un programme ou un utilisateur

Registres (dans le CPU)

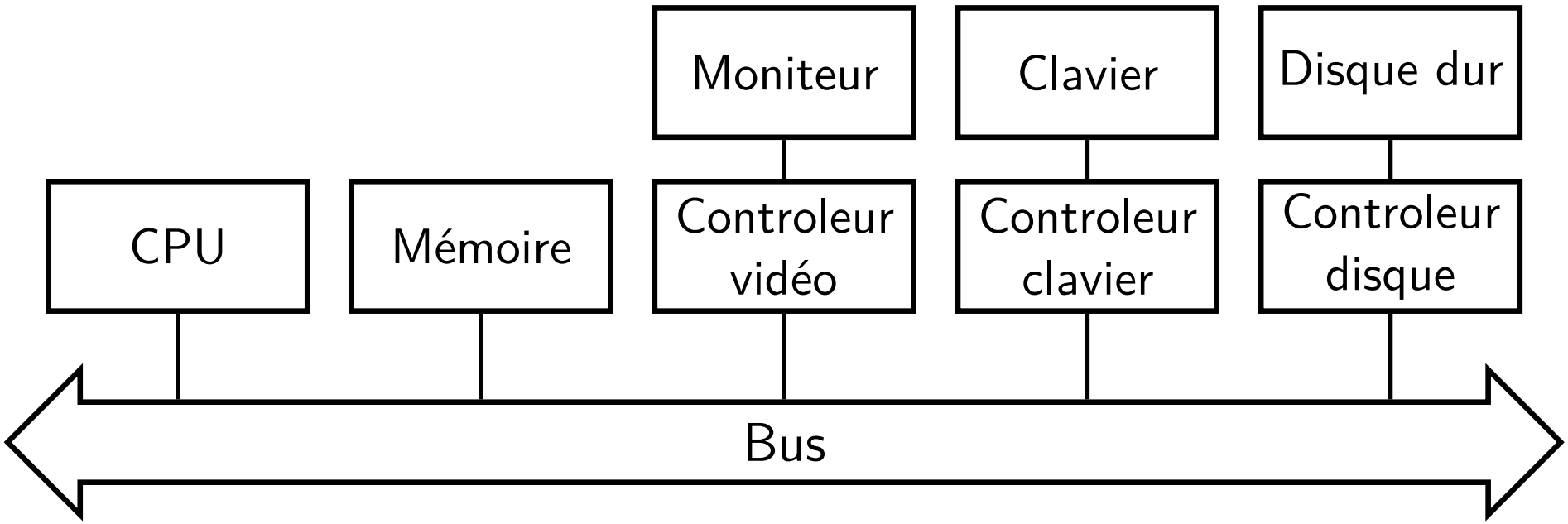
Généraux (pour les calculs) et spéciaux, dont :

- Compteur ordinal (PC, *program counter*)
- Pointeur de pile (SP, *stack pointer*)
- Mot d'état (flags d'états et de contrôle)

Mémoire volatile (RAM)

- Grand tableau d'octets adressables
- Contient code machine et données

Architecture à la von Neumann



INF3135 en très vite

Le logiciel c'est difficile

- Représentation des données, gestion de la mémoire, pointeurs, etc.
- Performance processeur, disque, réseau, énergie, etc.
- Débogage, portabilité, qualité logicielle, etc.

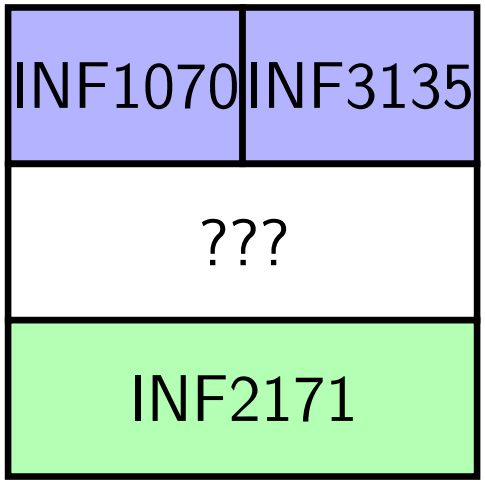
$\rightarrow$ Produire **le bon** code de **la bonne** façon, c'est difficile

Le logiciel c'est complexe

- Utilisation de bibliothèques
- Processus de compilation

Objectif du cours

Combiner les mondes



INF1070 : Des processus et utilisateurs

- Cohabitent pacifiquement
- Sur un même ordinateur

INF3135 : Des logiciels

- Complexes
- De qualité variable

INF2171 : Des ordinateurs et CPU

- Conceptuellement simples

Comment c'est possible que tout fonctionne ensemble ?

Exemples

Partage des ressources

- Plusieurs programmes s'exécutent en même temps
- Mais doivent se partager les ressources matérielles
- CPU, mémoire, fichiers, etc.

Isolation des processus

- Les bogues des programmes existent
- Mais affectent rarement les autres programmes directement
- Qui s'exécutent pourtant sur le même ordinateur

Sécurité des données

- Les utilisateurs malveillants (ou incompétents) existent
- Mais ne peuvent pas lire ou corrompre les données des autres
- Sauf avec un tournevis et un marteau

Objectifs du cours

- À quoi sert un SE
- Comprendre comment fonctionne un SE
- Savoir utiliser les services offerts par les SE

Beaucoup de rôles

Dans le cadre du cours, on va voir

- Gestion des processus et leur communication
- Gestion des fichiers et de l'espace disque
- Gestion de la mémoire
- Gestion des périphériques (entrées-sorties)

Attention

- Les rôles sont inter-reliés
- Le découpage des rôles est assez arbitraire

Beaucoup de points de vue

Utilisateur

- Humain (de base)
- Administrateur système

Développeur d'applications

- La plupart d'entre-vous

Développeur de systèmes d'exploitation

- Se mettre *dans ses souliers* peut aider à mieux comprendre

Constructeur de matériel

- On le prendra moins en compte

Beaucoup de niveaux

On raisonnera souvent à trois niveaux différents
Attention à pas les mélanger !

Niveau général

- Problèmes conceptuels et solutions générales
- Problèmes fréquents et solutions habituelles

Niveau Unix et POSIX

- Le mode Unix et sa philosophie
- Normes, appels système et API
- Avantage : répandu, (relativement) portable et documenté

Niveau Linux

- Le projet Linux (et son écosystème)
- Détails d'implémentation et de services spécifiques
- Avantage : versatile, libre, gratuit, ouvert, étudiable

Beaucoup d'histoire

Longue historique

Les systèmes d'exploitation existent depuis les années 1960
En 2020 les choses importantes ne sont pas **exactement** les mêmes qu'en 2010, 2000, 1980 ou 1960

Évolution des SE

- Monotâche $\rightarrow$ Multiprogrammation $\rightarrow$ Multitâche $\rightarrow$ Multiutilisateurs $\rightarrow$ Virtualisation et infonuagique

Difficulté

- La gomme a été mâchée par beaucoup de monde
- Le vocabulaire et les concepts sont parfois spécifiques

Beaucoup de spécialités

Difficultés autour des SE: plusieurs spécialités

Architecture matérielle

- Organisation matérielle des ordinateurs
- Elle est complexe et évolue chaque année

Programmation *bas niveau*

- À l'interne (dans le système lui-même)
- Dans les API offertes aux programmes

Algorithmique

- Recherche d'efficacité
- Taille croissante des systèmes

$\rightarrow$ impose l'utilisation d'algorithmes sophistiqués

Dans le cadre du cours on essaiera d'éviter ces difficultés

110 Définition et rôles

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Plan

Définition des systèmes d'exploitation

Définition des systèmes d'exploitation

Questions essentielles

- C'est quoi ?
- À quoi ça sert ?
- Comment c'est fait ?

Problèmes

- Pas de définition formelle parfaite
- Pas de liste précise des rôles, des tâches ou des composantes

$\rightarrow$ Ça évolue en fonction des besoins et des ressources

Tentative de définition

Un **système d'exploitation** est une **couche logicielle** qui sert d'intermédiaire entre les **utilisateurs** et les ressources **matérielles** de l'ordinateur et qui offre un **environnement** d'exécution aux **programmes** qui se veut efficace, robuste et utilisable.

Composantes d'un ordinateur

Grossièrement, un ordinateur comporte...

Du « matériel »

Processeur, mémoire, disques, périphériques, etc.

Un « système d'exploitation »

Qui semble (*un mal*) nécessaire pour

- Configurer le matériel
- Installer et exécuter des applications

Exemples: Windows, Debian, MacOS, Android

Des « programmes d'application »

Les vrais logiciels utiles à l'utilisateur

Système d'exploitation : deux points de vue

Clé en main (avec programmes système)

- C'est le point de vue grand public
- Inclut toute sorte de programmes : shells, interfaces graphiques, utilitaires de base, outils de configuration, etc.

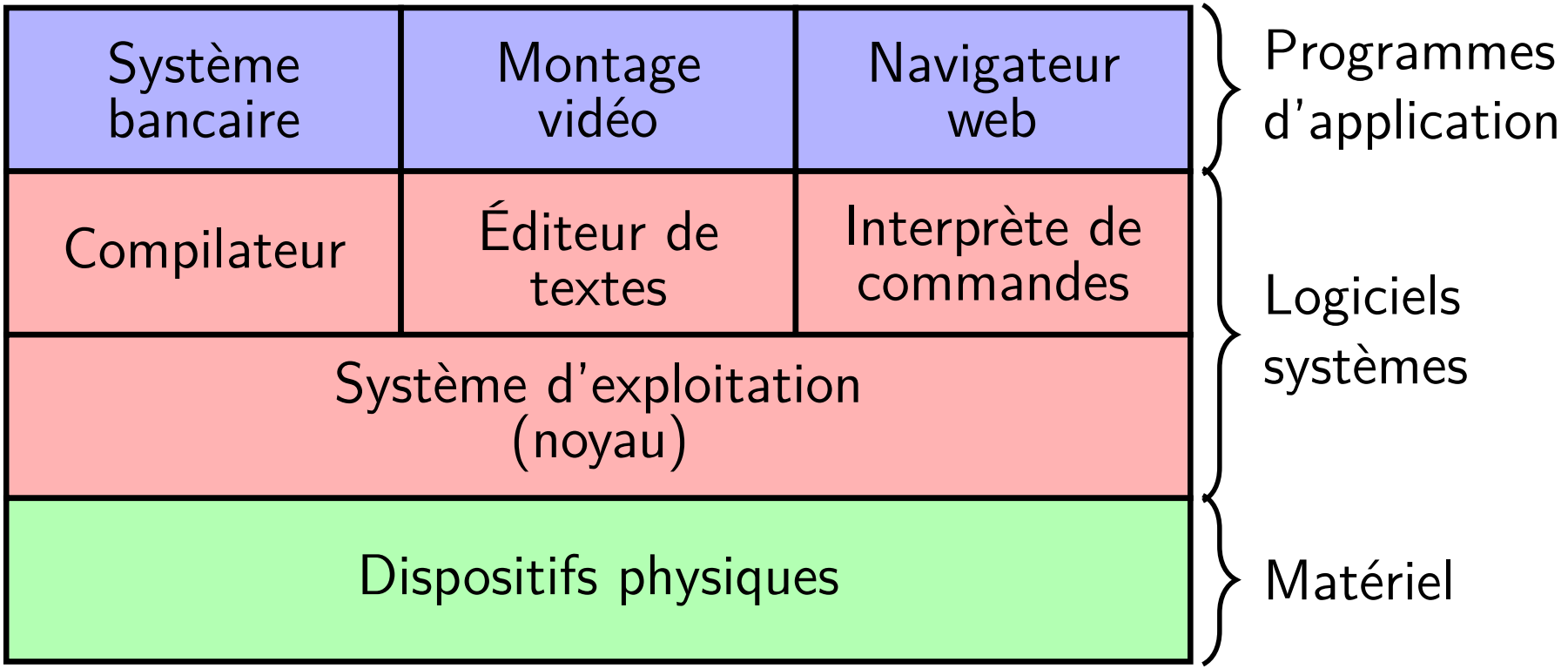
Espace disque recommandé pour une installation normale

- Debian 10 $\approx$ 10Go d'espace disque
- Windows 10 $\approx$ 32Go d'espace disque

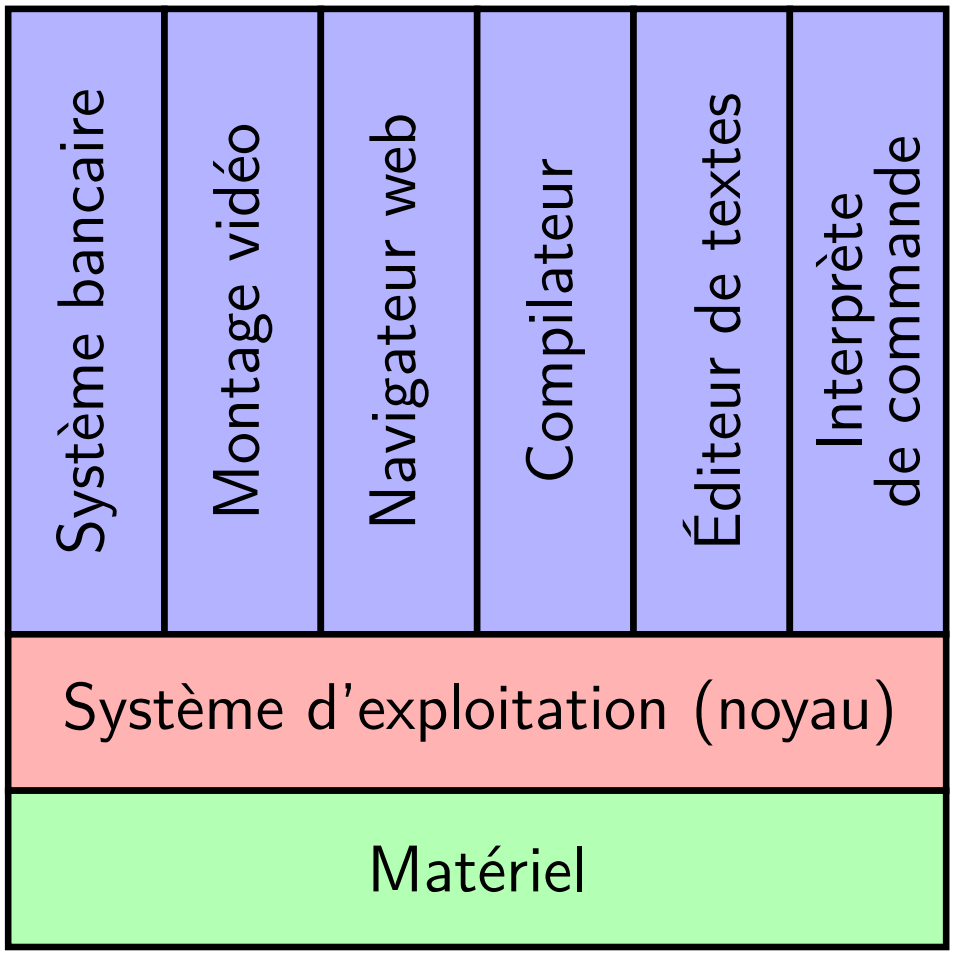
Noyau seul

- C'est le point de vue du cours
- Seulement la couche logicielle toujours en cours d'exécution
- Exemple: Linux (qui servira d'exemple dans le cours)
- Espace disque: Debian linux-image-5.7.0-2-amd64 $\approx$ 300 Mo

Composantes d'un ordinateur : vision de l'utilisateur



Composantes d'un ordinateur : notre vision



Vision du SE = notre vision

- Une seule catégorie de programmes
- Pas d'accès direct entre les programmes et le matériel

Composantes d'un système d'exploitation

Qu'a-t-on le droit de mettre dans un système d'exploitation ?

1998 Microsoft face à la justice américaine

- Abus de position dominante à cause des restrictions empêchant la désinstallation d'Internet Explorer.
- Source Wikipedia

2020 Microsoft Edge impossible à désinstaller

- Source

Rôles d'un système d'exploitation

SE = Couche d'abstraction

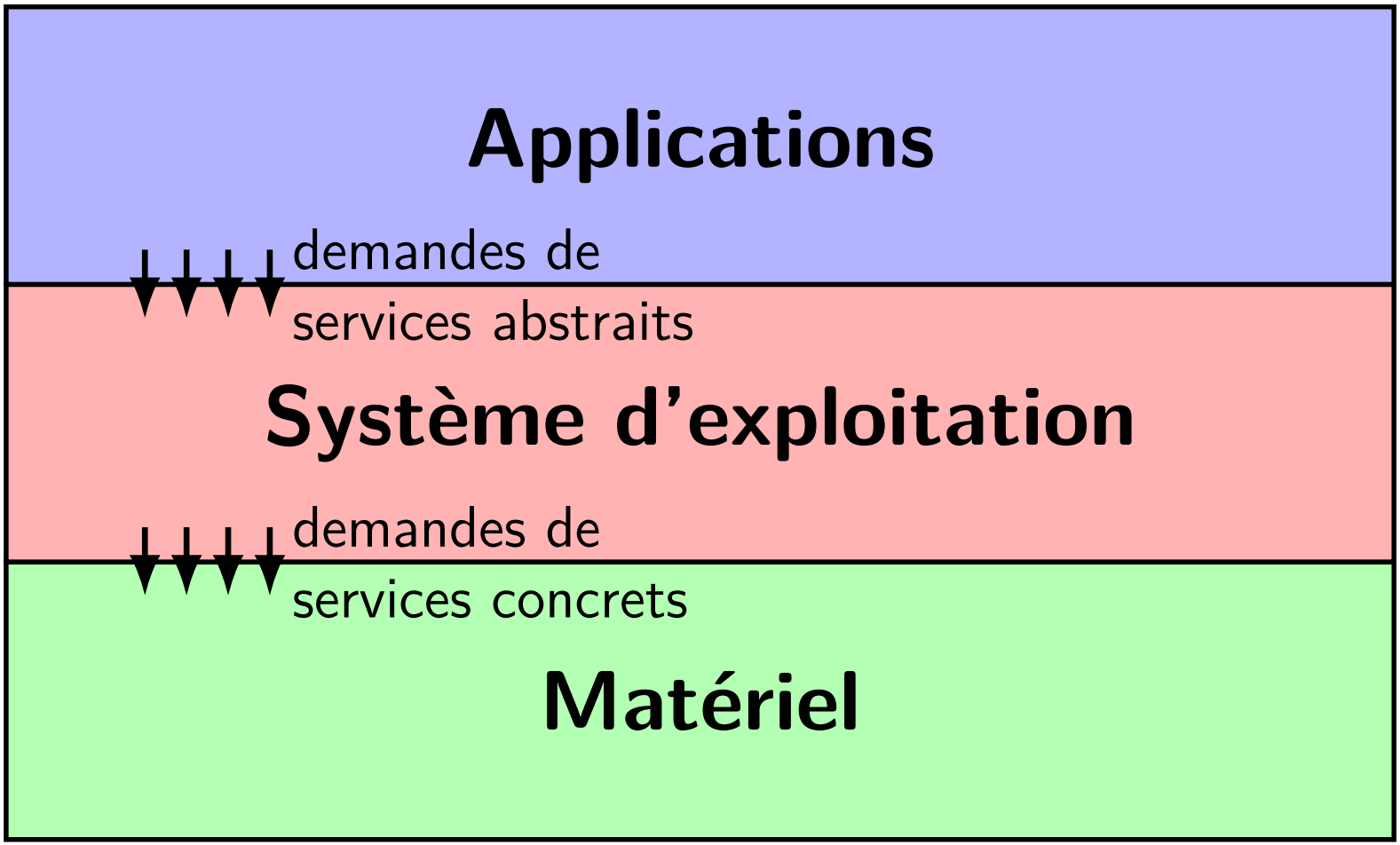
Abstrait la machine

- Cache certains détails que l'utilisateur n'a pas à connaître pour exploiter la machine
- Présente à l'utilisateur une machine virtuelle facile à utiliser et à programmer
- Offre toute sorte de services abstraits: gestion des fichiers, communication entre programmes, etc.

Connaît

- Connaît les détails internes intimes de la machine
- Utilise les services concrets (matériels) de la machine

SE = Couche d'abstraction



SE = Couche d'abstraction

Pour les applications ?

- Pas besoin d'être spécifique à chaque matériel possible

→ Y compris du matériel qui n'existe pas encore

- Mais peuvent devenir spécifique à un système d'exploitation

→ Développer des applications portables entre différents systèmes d'exploitation est plus difficile

Pour les matériels ?

- Développement de **pilotes** (*driver*) spécifiques au système d'exploitation
- Mais tous les systèmes d'exploitation sont pas égaux

→ Version de système d'exploitation non maintenu, matériel discontinué, effort de développement non rentable

On y reviendra

SE = Couche d'abstraction

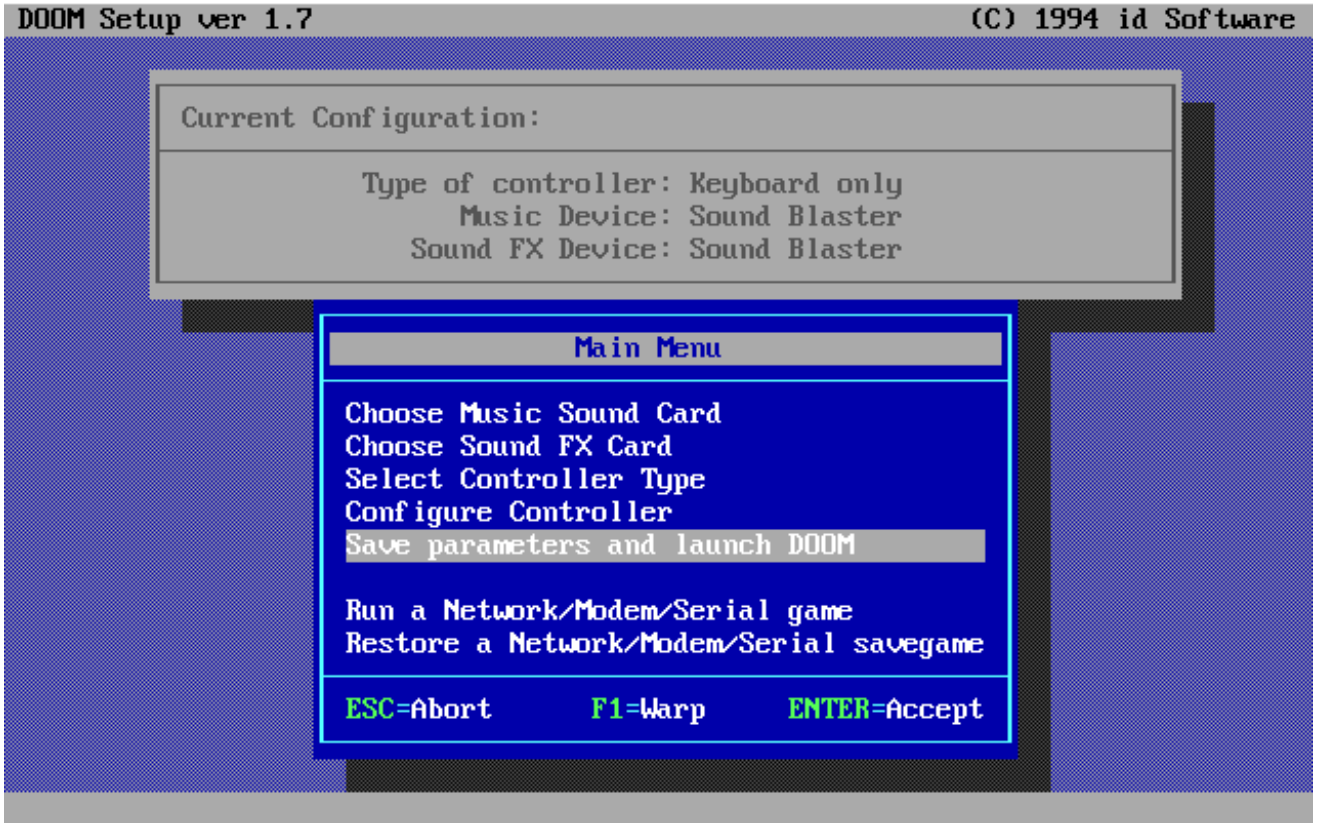
Source de Linux v5.7

```
$ du -sh * | sort -hr | head
626M    drivers
134M    arch
50M Documentation
45M include
42M fs
40M sound
39M tools
33M net
10M kernel
5,9M    lib
```

Plus de **70%** du code source dans `drivers` et `arch`.

Question. Expliquer la commande Unix

Jouer à Doom sur PC dans les années 90



- DOS n'offrait pas d'abstraction pour les cartes son
- Chaque jeu vidéo devait gérer entièrement un ensemble fixé de cartes sons
- L'utilisateur voyait des détails matériels: IRQ, DMA, etc.

SE = Gestionnaire de ressources

Répartir efficacement les ressources limitées

- Temps CPU
- Mémoire
- Périphérique (ex. disques, réseau, imprimante, webcam, etc.)

$\rightarrow$ Quelles politiques adopter ?

$\rightarrow$ Quels mécanismes mettre en œuvre ?

Faire cohabiter pacifiquement plusieurs processus et utilisateurs (et matériels)

- Protection mémoire
- Gestion des droits
- Respect des répartitions des ressources

$\rightarrow$ Comment imposer ça ?

Gestion des processus

- Création et destruction des processus
- Décider de l'attribution processeur aux processus
- Suspendre et continuer les processus
- Permettre la synchronisation et la communication des processus

Gestion de la mémoire

- Répartir la mémoire entre les processus
- Gérer l'espace libre et les demandes de mémoire
- Décider du passage en mémoire distante

Gestion des fichiers

- Création, manipulation et destruction des fichiers et répertoires
- Gestion de l'espace disque libre
- Gestion des droits

Gestion des périphériques

- Gestion de la mémoire, cache, tampons, IRQ, DMA
- Pilotes spécifiques
- Gestion de l'énergie (batterie)
- Répartition des ressources entre processus (bande passante disque, réseau, etc.)

Bibliothèques et démons

Exclusivité du SE ?

Besoins

- Abstraire du matériel
- Offrir des services
- Gérer des ressources

Des approches existent déjà

- Bibliothèques logicielles
- Services, démons, serveurs

Qu'est-ce qui rend les systèmes d'exploitation **différents** ?

Bibliothèques logicielles

C'est quoi?

Composantes logicielles prêtes à l'usage par des programmes

- Exemple: bibliothèque cryptographique
- Compilées ou non, statiques ou dynamiques (.so, .dll)
- Offrent une interface abstraite aux programmes (API/ABI)

Avantages

- Permet de factoriser du **bon** comportement
- Mise à jour indépendante des bibliothèques dynamiques partagées

Services, démons, serveurs

C'est quoi?

Processus s'exécutant en arrière-plan qui répondent à des requêtes

- Servent aussi à gérer l'activité de périphériques
- Exemple: serveur d'impression
- Offrent des services abstraits via des mécanismes de communication entre processus

Avantages

- Permet de sous-traiter du **bon** comportement
- Mise à jour indépendante des logiciels

Exclusivité du SE

Qu'est-ce qui rend les systèmes d'exploitation différents ?

Les privilèges

Le système d'exploitation a le monopole de privilèges exclusifs

- Tout accès au matériel passe nécessairement par lui
- Toute allocation de ressource à un logiciel sera respectée

$\rightarrow$ Sauf, bien sûr, si le système d'exploitation autorise des formes de contournement

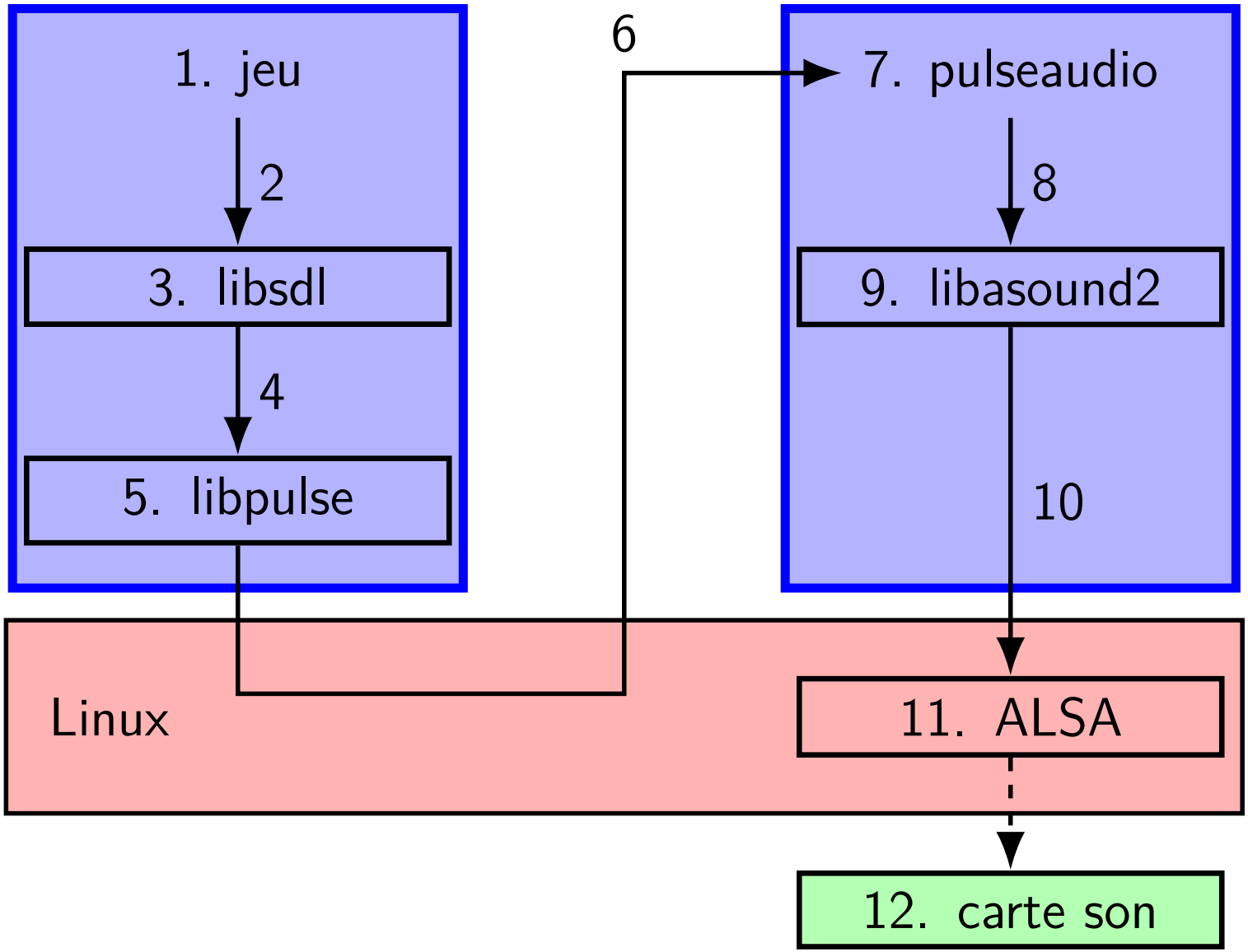
En pratique

Bibliothèques et services vont souvent encapsuler des services système pour harmoniser, simplifier (ou complexifier) les choses

- En informatique, on aime résoudre les problèmes en ajoutant un nouveau niveau d'abstraction

Exemple: le son

1. Un jeu vidéo a besoin de son
2. Il utilise la bibliothèque `libsdl`
3. `libsdl` expose une API portable entre différents systèmes
4. `libsdl` peut utiliser `libpulse` pour jouer les sons
5. `libpulse` simplifie la communication avec `pulseaudio`
6. via des services systèmes d'IPC (*interprocess communication*)
7. `pulseaudio` est un serveur de son (démon) qui gère la configuration complexe des éléments audio de l'ordinateur
8. `pulseaudio` utilise la bibliothèque `libasound2`
9. qui simplifie l'utilisation des services de son de Linux
10. cela via des services système dédiés
11. ALSA (*Advanced Linux Sound Architecture*) est un morceau (sous-système) du noyau Linux
12. qui peut accéder à la carte son



Version simplifiée sans la libc ni libdl.

Exemple: le son

Le système d'exploitation isole les processus

Un processus ne peut agir directement

- sur un périphérique
- sur un autre processus

$\rightarrow$ il doit passer par le système d'exploitation

Questions

Un processus pourrait-il...

- Utiliser directement `libpulse` sans passer par `libsdl` ?
- Faire de l'IPC directement avec pulseaudio sans passer par `libpulse` ?
- Communiquer directement avec pulseaudio sans passer par des services du noyau d'IPC ?
- Utiliser les services noyau d'ASLA sans passer `libasound2` ?
- Agir sur la carte son sans passer par des services du noyau ?

Analogie : Le SE est le gouvernement de l'ordinateur

Il ne sert à rien en soi
- Pas directement utile à l'utilisateur

Il permet la cohabitation *pacifique* entre
- Les différents programmes
- les différents utilisateurs
- les différents matériels

Il possède les capacités d'imposer cette cohabitation
- Il a le monopole de privilèges particuliers

Mécanismes et politiques

Reste une question : comment on fait ça ?

- Mécanismes : quels sont les moyens à mettre en oeuvre
- Politiques : quelles sont les règles à appliquer

Séparer les deux questions permet plus de souplesse

- D'une part fournir des mécanismes de base
- D'autre part concevoir

$\rightarrow$ il est possible d'adapter les politiques sans devoir tout refaire

Philosophie Unix

Le plus souvent:

- Le noyau

$\rightarrow$ Expose des mécanismes simples ou élémentaires

- Les programmes et les administrateurs système

$\rightarrow$ Définissent des politiques

$\rightarrow$ Les implémentent à l'aide des mécanismes

120 Appels système

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Plan

Mécanismes matériels

Analogie

- « L'État est une communauté humaine qui, dans les limites d'un territoire, revendique avec succès le monopole de la violence physique légitime. » — Max Weber, *Le Savant et le politique* (1919)
- « Le système d'exploitation est une couche logicielle qui, dans les limites d'un ordinateur, revendique avec succès le monopole des mécanismes matériels. » — Analogie facile...

Mode noyau (dit privilégié)

Mode noyau : mécanisme

Objectif

S'assurer que certaines instructions machine sont réservées au système d'exploitation

Problème : le processeur est une machine

- Pour lui, système d'exploitation et processus n'existent pas
- Une instruction machine n'appartient à personne

Solution : deux modes d'exécution

- Un bit de mode dans le registre du mot d'état
- Mode noyau (0) : toutes les instructions sont utilisables
- Mode utilisateur (1) : certaines instructions sont interdites

$\rightarrow$ le processeur refuse physiquement d'exécuter l'instruction si le mode n'est pas le bon

Mode noyau : politique

- Au démarrage le CPU est en mode noyau

→ Le système d'exploitation se charge et configure la machine

- Quand le système démarre des processus, il passe le CPU en mode utilisateur

→ Les applications sont restreintes sur ce qu'elles peuvent faire

- Quand le CPU revient au système, on repasse au mode noyau

→ On va y revenir...

Mode noyau : beaucoup de détails

Le monde des CPU est complexe et plein de variété

- La liste des instructions et le mode auquel ils appartiennent est spécifique à chaque processeur
- Plutôt que désactiver l'instruction le mode peut limiter des comportements ou en changer le sens
- Pourquoi se limiter à 2 modes ? Intel en a 4. Certains ARM en ont 7
- On parle parfois d'anneaux de protection (*rings*). Le mode noyau est Ring0
- Les processeurs peuvent offrir d'autres types de modes d'exécution complémentaires au mode noyau

La documentation pour programmeurs des processeurs Intel fait plus de 5000 pages !

Appels système

Problème

Un processus veut faire une opération privilégiée

- Il ne peut pas le faire lui-même

$\rightarrow$ Il est en mode utilisateur

- Il ne peut pas changer le mode lui-même

$\rightarrow$ Sinon c'est pas un vrai privilège

- Il ne peut pas juste déléguer à une bibliothèque ou faire un `call` à un sous-programme

$\rightarrow$ Le mode resterait non-priviliégié

Instruction machine spéciale

Appel système

- Sauvegarde registres (dont CO)
- Passe en mode noyau
- Branche sur du code spécifique du système d'exploitation

$\rightarrow$ Le processus ne branche pas où il veut

$\rightarrow$ Le processus perd donc le contrôle du CPU

Retour d'appel

- Passe en mode utilisateur
- Restaure les registres (dont CO)

$\rightarrow$ Le processus s'est rendu compte de rien

Différence avec call ?

- `call` utilisé pour les sous programmes (processus et noyau)
- `call` prend en argument une adresse
  `syscall` prend un argument un numéro d'appel système

Liste définie d'appels système

- Chaque système d'exploitation est différent
- Plus de 400 sur Linux
- Mais beaucoup sont rarement utilisés

Performance

- `syscall` plus cher que `call` (temps de calcul)
- Coût important du **changement de contexte**

Détails spécifiques

- À chaque système d'exploitation
- Pour chaque architecture

Noms variés

- `syscall`, `int`, `trap`, `swi`, etc.
- Confusion avec d'autres mécanismes (interruption, fautes, etc.)

Nombreux détails

- Qui sauvegarde et restaure les registres ? Comment c'est fait ?
- Où on branche exactement ? Qui décide ?
- Comment on passe les arguments et retourne le résultat ?

Pas d'équivalent portable en C

- Quelqu'un doit les coder en assembleur
- Des enveloppes (*wrapper*) sont fournies

hello_syscall.c

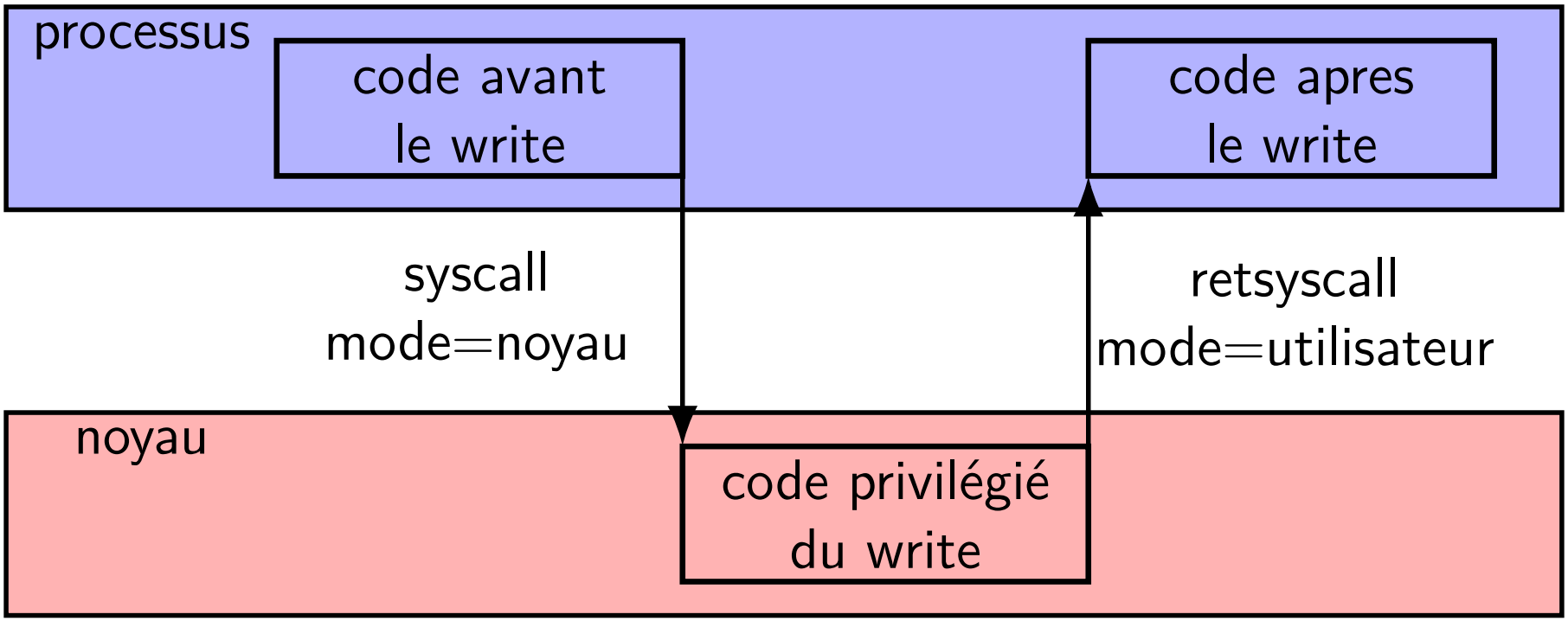
Appel système POSIX `write(2)` « à la main »

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
int main(int argc, char *argv[]) {
  char msg[] = "Hello, World!\n";
  syscall(SYS_write, 1, msg, 14);
  return 0;
}
```

Les arguments de `syscall(2)` sont

- `SYS_write`: le numéro de l'appel système
- 1: le descripteur de la sortie standard
- `msg`: l'adresse du message à écrire
- 14: le nombre d'octets à écrire

Mise en œuvre matérielle



- Le processus s'exécute en mode utilisateur
- L'appel système `write` branche sur le noyau
- Le code du `write` s'exécute en mode noyau
- Le retour au processus perd le mode noyau
- Le processus continue en mode utilisateur

Voir les appels système

Sous Linux `strace`

- Outil de débogage
- Permet de surveiller les appels système Linux
- C'est magique ! (les détails dans INF600C)

```
$ strace ./hello_syscall
[...]
write(1, "Hello, World!\n", 14)          = 14
[...]
```

`strace` sait afficher de façon humaine:

- le nom
- les arguments (au bon format)
- le résultat

Enveloppes

Enveloppe

Problème

Les appels système sont peu portables

- Instructions machines spécifiques aux processeurs
- Choix particuliers des systèmes d'exploitation
- Pas de façon standard de les exprimer en langage C

Solution

Une bibliothèque standard fournit des fonctions spécifiques

- Enveloppe chacun des appels système (*wrapper*)
- Expose API/ABI simples et portables (en C ou C++)
- Connait l'architecture et les choix du système
- Implémenté avec des morceaux d'assembleur (mal nécessaire)

Portabilité interne

Sous Unix

- La `libc` contient les fonctions d'enveloppe
- La section 2 du `man(1)` les documente
- `unistd.h` déclare de nombreux appels système POSIX

RTFM: il peut y avoir des variations entre l'appel système et la fonction C

Sous Windows

- `kernel32.dll` contient les fonctions d'enveloppe
- Par exemple `WriteConsole`
- Les détails techniques ne sont pas documentés :(

Exemple `hello.c`

Pour le programmeur, voilà ce que ça donne

```
#include <unistd.h>
int main(int argc, char *argv[]) {
  char msg[] = "Hello, World!\n";
  write(1, msg, 14);
  return 0;
}
```

- `write(2)` est la fonction système POSIX qui écrit des données
- `ssize_t write(int fd, const void *buf, size_t count);`
- Une vraie fonction C avec une vraie signature

$\rightarrow$ Les détails sont laissés à la libc

hello_asm.s

Version assembleur équivalente à `hello.c` (Linux/x86_64)

```
# Programme qui affiche "hello world"
# Compiler avec `gcc hello_asm.s -o hello_asm`

   .globl  main
main:
   # write(1, msg, 14)
   mov     $1, %rax          # appel système write (1)
   mov     $1, %rdi          # sortie standard (1)
   lea     msg(%rip), %rsi   # adresse du message (PIC)
   mov     $14, %rdx         # taille du message (14 octets)
   syscall                   # instruction TRAP

   # return 0
   mov     $0, %rax          # valeur de retour (0)
   ret                       # return
msg:
   .ascii  "Hello, World!\n"
```

Gestion des erreurs

En cas d'erreur

Les enveloppes des appels système:

- Retourne -1
- Positionne `errno(3)`
- La liste de errno est **fixe**: `errno -l`

Le programmeur doit gérer les cas d'erreurs

- Lire la doc (RTFM), section « `ERREURS` »
- Identifier les erreurs possibles
- Les traiter (ou pas)

→ Le traitement des erreurs est une chose **difficile**

- Recommencer ? Ignorer ? Abandonner ?
- Afficher un message ? Quel message ?
- Ne pas réinventer la roue: `perror(3)`, `strerror(3)`

Exercice: lire et comprendre les erreurs de `write(2)`

Appels de bibliothèque

Appels de bibliothèque

Appels système

- Services primitifs
- Spécifiques au système d'exploitation

Bonnes pratiques de génie logiciel

- Utiliser des services généraux
- Portable entre systèmes d'exploitation

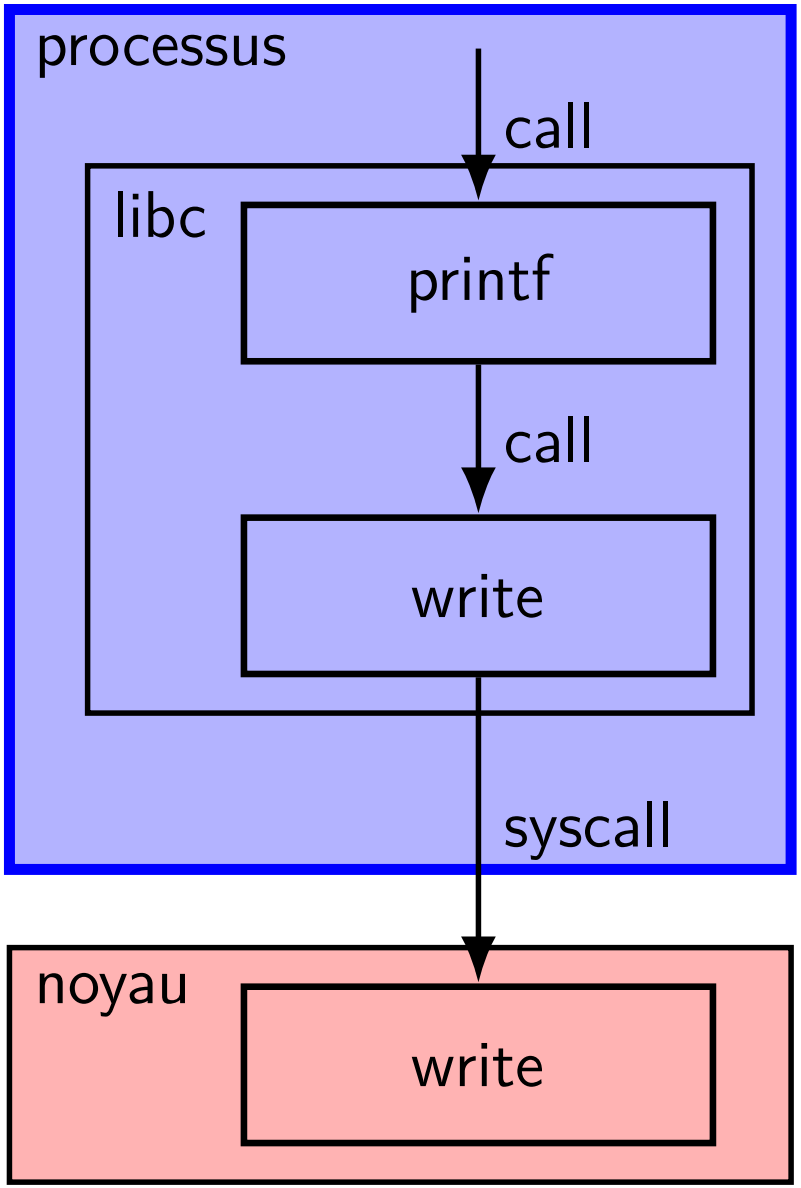
hello_printf.c

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

Bibliothèques standard du C

- `<stdio(h>`: `printf(3)`, `fwrite(3)`, etc.
- Documentés dans la section 3 du man
- Revoir INF3135 pour les détails

Appels de bibliothèques



Les bibliothèques

- Font partie du processus

$\rightarrow$ Aucun privilège particulier

$\rightarrow$ Les calls sont normaux*

La libc

Fournit $\approx$ 1500 fonctions fonctions

- Les fonctions standard C
- Les enveloppes d'appels système

* Il y a une astuce quand les bibliothèques sont dynamiques.

Bibliothèque vs. système

Indépendance chez Linux

- Projet indépendant $\neq$ noyau Linux
- Généralement glibc (de GNU)

Efficacité

- Les appels système coûteux et bas niveau
- Les fonctions de bibliothèques optimisent
  - Caches additionnels
  - Factorisation des appels
  - Traitement direct si possible (sans appel système)
- Les fonctions de bibliothèques généralisent
  - Portables entre différentes versions et systèmes d'exploitation
  - Profitent de nouveaux appels système quand disponibles

Exclusivité (on insiste)

- Le système a l'**exclusivité** des mécanismes matériels
- Les processus sont **isolés** du reste
- Les **appels système** sont leur **seul** moyen d'interagir avec l'extérieur (utilisateurs, périphériques, autres processus)

→ Tout processus qui a besoin d'interagir passera par des appels système

Allégorie de la caverne

Les processus ne voient le monde qu'à travers ce que le noyau décide

Le noyau « ment » souvent :

- Les fichiers de `/proc` n'existent pas vraiment
- C'est pas un disque mais du réseau
- La mémoire n'est pas toujours disponible (sur-réservation)
- *There is no spoon!*

→ Mais ça permet beaucoup de choses!

Dans le cadre du cours

On utilisera le plus possible les appels système

- L'objectif c'est d'être le plus proche du noyau
- Et d'apprendre à le maitriser

On traitera (correctement) les cas d'erreur

- La robustesse sera prise en compte dans la notation des TP
- Les autres qualités aussi : exactitude, lisibilité, modularité, etc.

Compatibilité

Rétrocompatibilité

Le noyau Linux a une forte tradition de rétrocompatiblité

> *« WE DO NOT BREAK USERSPACE! »*
>
> – *Linus Torvalds (2012)*

Les appels système sont stables

- Leur interface de programmation (ABI)
- Leur comportement

Ce n'est pas le cas à l'intérieur du noyau

- Les sous-systèmes évoluent constamment
- Ajout de fonctionnalités non compatibles
- Ajout et mise à jour de pilotes de périphériques

Couche de compatibilité

Une **couche de compatibilité** permet

- À des applications d'un système d'exploitation (ex. Windows) de fonctionner sous un autre système d'exploitation (ex. Linux)

→ L'architecture processeur doit être la même

→ C'est différent d'un émulateur

Exemples

- Wine convertit les appels Windows en appels POSIX

→ Proton fork par Valve pour jeux vidéos sans support Linux

- Cygwin Convertit les appels POSIX en appels Windows
- WSL Windows Subsystem for Linux, de Microsoft

Couche de compatibilité

Mise en œuvre (en gros)

- Fournir une bibliothèque de base spéciale
- Se substitue à celle du système (libc.so, kernel32.dll, etc.)
- Traduit les appels système de l'un vers des appels équivalents
- Simule l'environnement attendu de l'application

Limites

En pratique, traduire les appels système est très compliqué

- Tous les appels système ne sont pas traduits à 100%
- Les performances peuvent varier
- L'environnement simulé doit être cohérent

$\rightarrow$ Système de fichiers, accès au matériel, communication entre processus, etc.

130 Mécanismes matériels

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Mode noyau processeur (privilégié)

Analogie

- « L'État est une communauté humaine qui, dans les limites d'un territoire, revendique avec succès le monopole de la violence physique légitime. » — Max Weber, *Le Savant et le politique* (1919)
- « Le système d'exploitation est une couche logicielle qui, dans les limites d'un ordinateur, revendique avec succès le monopole du mode noyau. » — Analogie facile...

Appels système

Permet à un processus de demander un service au SE

- Passage du CPU en mode noyau
- Branchement à une sous-routine spéciale du SE

Plan

Interruption matérielle

Interruption matérielle

Permettre au matériel de signaler des évènements

- Appui d'une touche, nouveaux paquets réseau, etc.
- Notification d'une commande terminée
- Problème physique
- Etc.

Mécanisme

- Connexion dédiée: périphériques $\rightarrow$ CPU
  - Pour signaler l'existence d'un évènement
- Le CPU vérifie la présence d'une interruption
  - À chaque instruction
- Si interruption, automatiquement le CPU
  - Sauvegarde des registres (dont le CO)
  - Passe en mode noyau
  - Branche à un endroit spécifique en mémoire

SE gère les interruptions

Le noyau au démarrage:

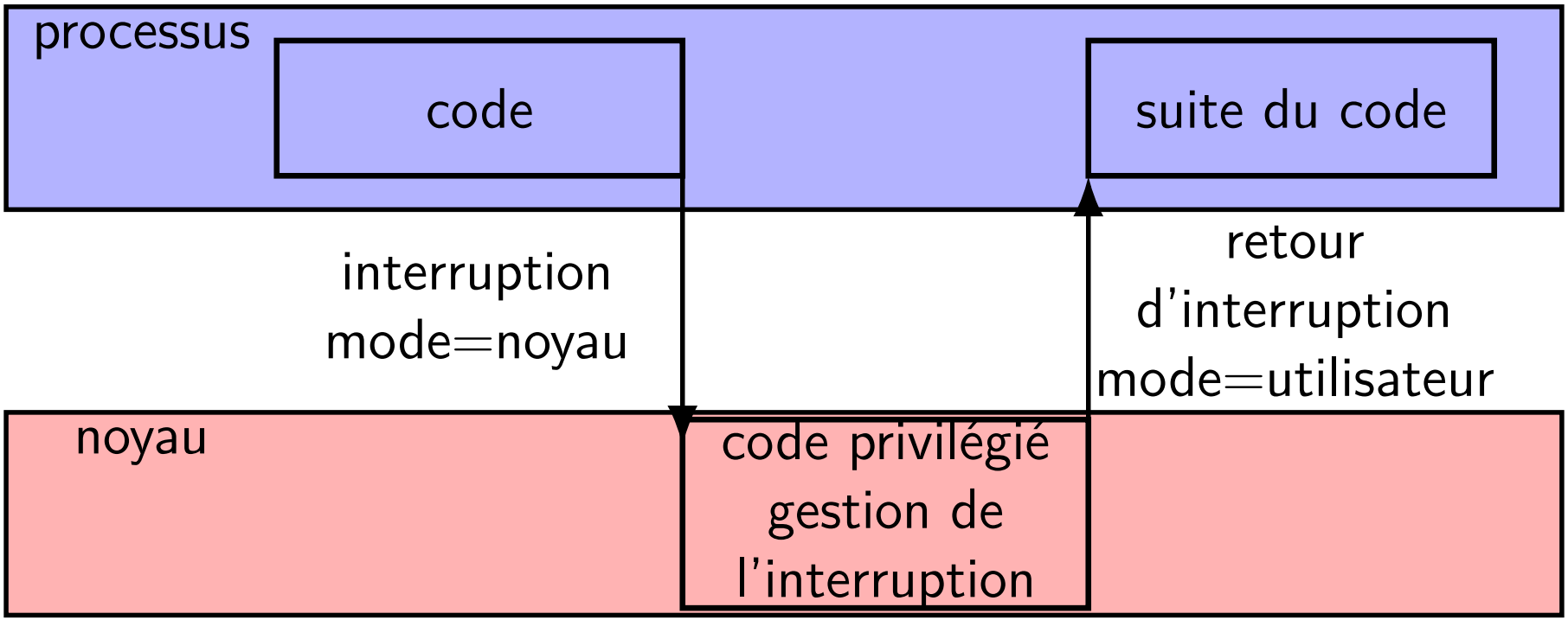
- Configure la machine
- Sous-routines spéciales associées aux interruptions

Le noyau en cas d'interruption:

- Le processus actif perd le CPU
- Une routine spéciale du noyau est automatiquement invoquée
- Le noyau
  - Sauvegarde les registres
  - Traite efficacement l'interruption
  - Restaure les registres
  - Passe en mode utilisateur

  $\rightarrow$ Le processus s'est rendu compte de rien

Exemple



- Une interruption matérielle arrive
- Le CPU est donné au noyau qui traite avec le matériel
- Le noyau rend la main au processus

$\rightarrow$ Le processus est interrompu, mais ne s'en rend pas compte

Interruptions vs. appels système

Cause

- L'appel système est volontaire
  Le processus fait un appel explicite
- L'interruption est involontaire
  Peut arriver à tout moment

Mécanismes analogues

- Bascule en mode noyau
- Branchement emplacement dédié du noyau
- Sauvegarde et restauration de registres

« Interruption logicielle »

- Nom alternatif des appels système
- `int` pour x86, `swi` (*software interrupt*) pour ARM

$\rightarrow$ Cause de la confusion inutile

Fautes

Fautes

Mécanisme: faute CPU

Le CPU lance (lui-même) une interruption matérielle en cas de:

- Instruction inconnue
- Opérandes invalides (division par 0)
- Violation de privilège (mode utilisateur)
- Etc.

On trouve aussi les termes « exception » ou « *trap* »

Politique

Le système d'exploitation sait

- Gérer les fautes CPU: interruptions matérielles classiques
- Déterminer le responsable: le processus qui a été interrompu

Exemple de scénario

- Un processus exécute une instruction privilégiée
- Le CPU refuse (mode utilisateur) et génère une faute
- Le SE s'exécute alors:
  - Inspecte les registres et la mémoire
  - Détermine le processus coupable
  - Lui envoie un signal (`kill`)
  - Ce qui termine le processus

Justice implacable

SE = investigue, arrête, condamne et exécute les processus délinquants

Exemple: division par zéro

```
int main(int argc, char *argv[]) { return 0/0; }
```

Horloge programmable

Horloge programmable

Problèmes

Comment attendre des échéances ?

- Faire une pause quelques secondes
- Gérer les expirations (timeouts)

Comment récupérer un CPU accaparé par un processus ?

- Calcul intensif
- Boucle infinie

Mécanisme

- Un matériel spécial

$\rightarrow$ une composante dédiée sur la carte mère

$\rightarrow$ ou directement le contrôleur d'interruption

- Décrémente un compteur
- Lève une interruption quand il atteint 0

Exemple de politique

- Le SE programme l'horloge
- Puis il donne la main à un processus
- Le processus bloque le CPU dans une boucle infinie
- Le délai programmé de l'horloge expire
- Une interruption est levée
- Le CPU est rendu au système

Question. C'est le processus ou le processeur qui est en boucle infinie ?

Multitâche

- C'est la base du multitâche préemptif
- Permet de répartir le CPU entre plusieurs processus

$\rightarrow$ On y reviendra

3 types d'horloges dans un ordinateur

Horloge programmable

- Pour lever des interruptions
- Analogie: minuterie

Signal d'horloge

- Rythme le fonctionnement électronique (CPU, RAM, Bus, etc.)
- Analogie: métronome

Horloge temps réel

- Maintient la date et l'heure réelle
- Alimentation autonome avec une pile
- Analogie: horloge murale

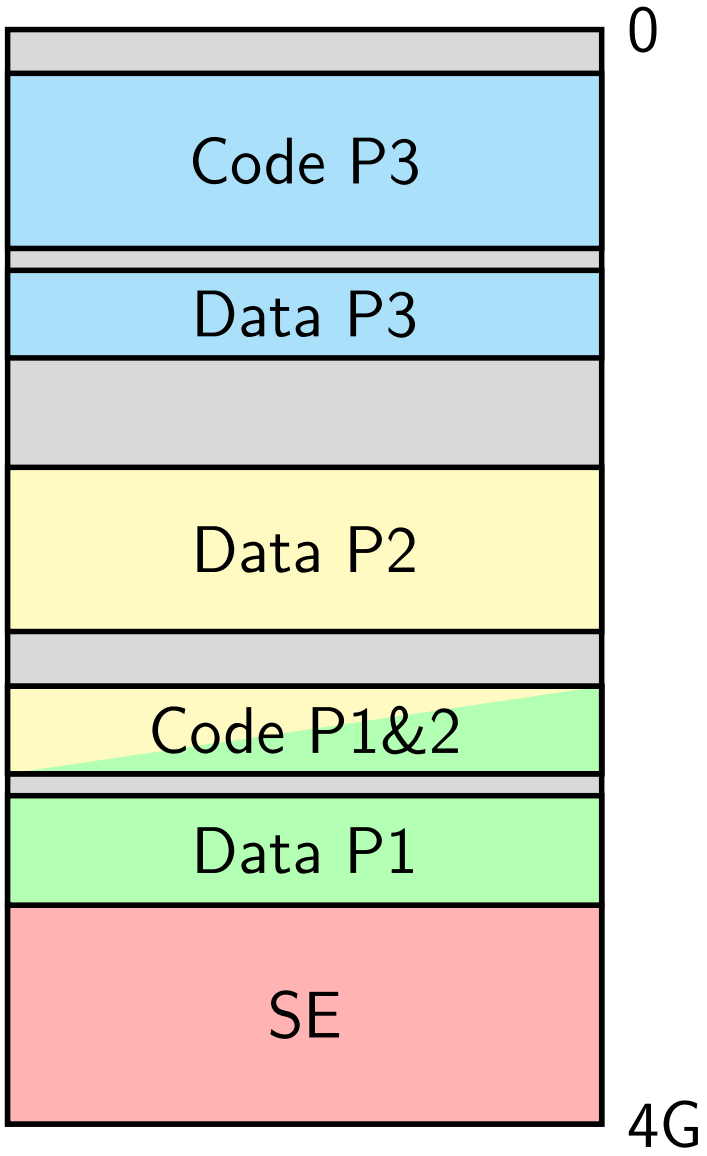
Protection mémoire

Organisation mémoire d'un programme

En mémoire, il y a

- Code du programme (en langage machine)
- Données (statiques, pile, tas, etc.)
- Bibliothèques

Défi: plusieurs programmes à la fois



Chaque processus

- A accès (lecture/écriture) qu'à son propre espace
- Tout accès en dehors est physiquement interdit

Le SE

- A accès à toute la mémoire
- Gère les limites physiques des programmes

Mécanisme

- On peut marquer des zones mémoires comme valides ou invalides
- Le mode noyau est nécessaire pour changer les zones
- Le CPU peut **efficacement** déterminer la validité d'une adresse

$\rightarrow$ Plusieurs approches possibles, on y reviendra

- Un accès en dehors d'une zone valide lève une faute

$\rightarrow$ le CPU vérifie chaque accès fait à la mémoire

Politique

- Le système rend valide les zones mémoire d'un processus
- Puis il lui donne la main
- Le processus accède dans ses zones
  - Tout va bien
- Le processus accède en dehors
  - Le CPU lève une faute
  - Le SE prend la main
  - Envoie un signal au processus fautif (`kill`)
  - Ce qui le termine

```
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int *i = NULL; return *i;
}
```

Nom habituel: « faute de segmentation » (*segmentation fault*)

Mémoire virtuelle et pagination

De nos jours quasiment

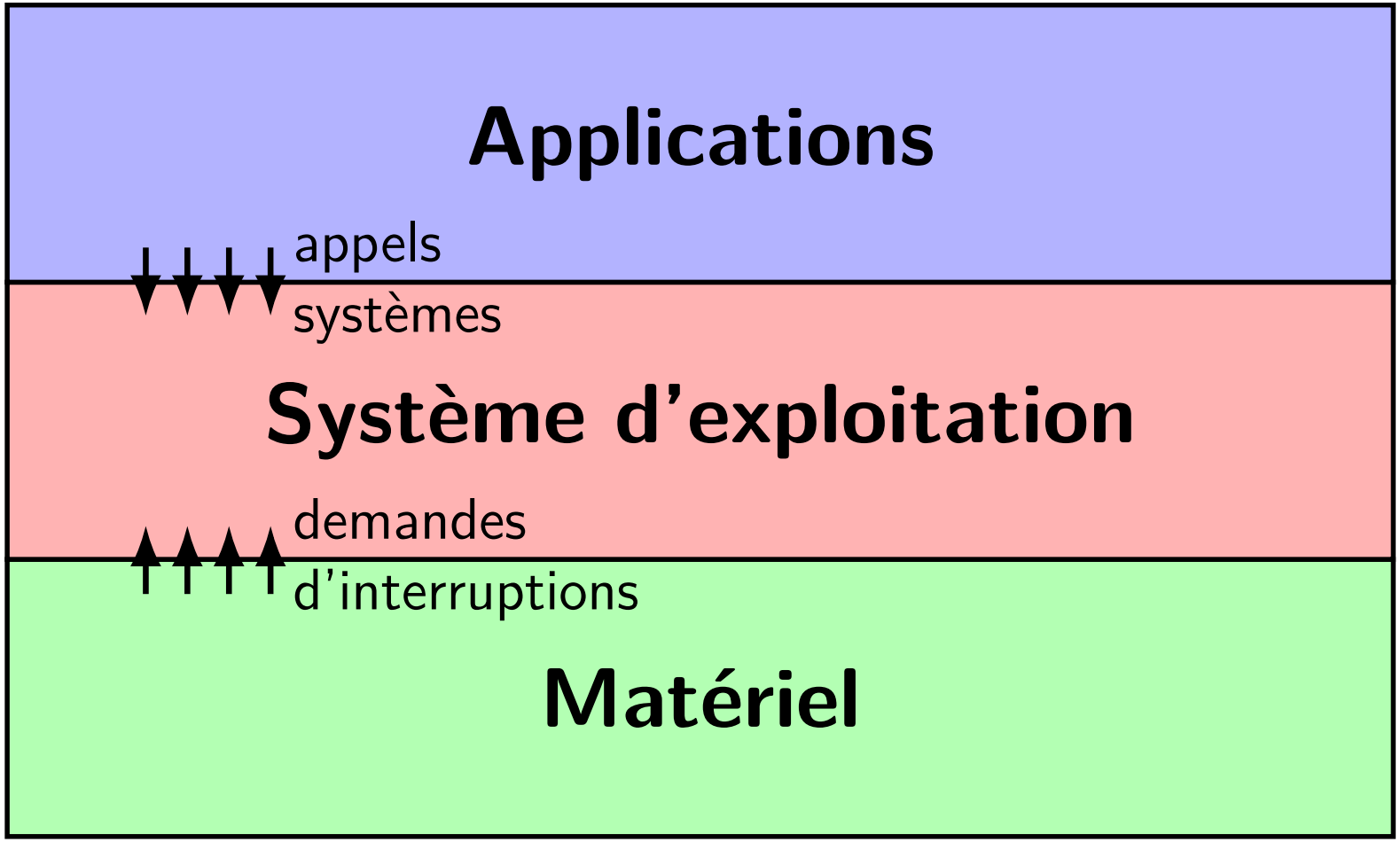
- tout processeur
- et système d'exploitation

Utilisent

- De la mémoire virtuelle
- Plus précisément de la pagination
- Avec des modes de protection

On y reviendra plus tard

Conclusion : Le SE au centre



200 Processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

La dernière fois

Qu'est-ce qu'un SE ?

- Couche logicielle située entre le matériel et les applications d'un ordinateur
- A l'exclusivité des mécanismes matériels (mode noyau)

Objectifs

- À abstraire la couche matérielle (appels système)
- À répartir équitablement les ressources entre les différents processus et utilisateurs
- À protéger le matériel, les processus et les utilisateurs les uns des autres

Analogie facile

- Le SE est le « gouvernement » de l'ordinateur

Processus

Processus : Définition

Définition : Un processus est

- Un programme en cours d'exécution
- Par un processeur

3 concepts liés

- Processus : l'exécution en cours d'un programme
- Processeur : celui qui fait l'exécution
- Programme : la suite d'instructions (prédéterminée)

Fondamental

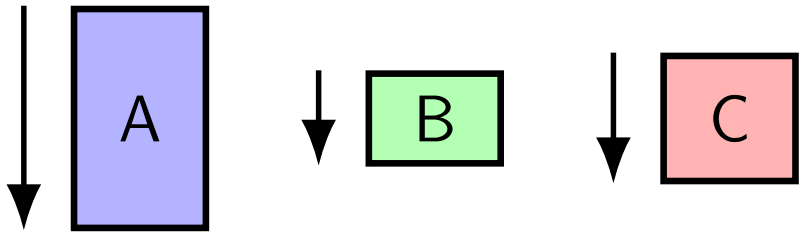
- Plusieurs instances d'un même programme
- Chacune de ces instances est appelée un processus
- Chaque processus est **autonome** et **isolé**

Exécution d'un processus

Chaque processus a l'impression d'être seul

Un processus

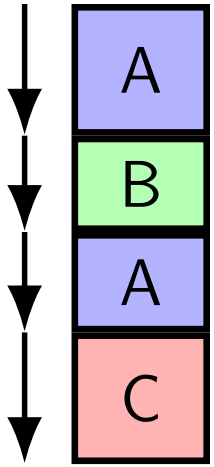
- Progresse de manière séquentielle dans son programme
- Progresse en même temps que les autres
- Ne prend pas en compte l'existence des autres processus (sauf si c'est programmé exprès)



Problème

- Un seul processeur (ou un nombre fixe)
- Plusieurs processus à exécuter en même temps

Illusion de parallélisme : multitâche



- Avancer chaque processus un petit peu à la fois
- Changer le processus actif régulièrement (10ms–100ms)
  - On parle de **changement de contexte**
  - Le changement est couteux (à ne pas abuser)
- Qui décide quel processus devient actif?
  - Le système d'exploitation
  - En fonction de politiques et de priorités
  - On parle d'**ordonnancement**

  $\rightarrow$ On y reviendra...

Multiprocesseurs et multicœurs

La même idée de base

- Chaque cœur traite un processus actif
- On change les processus actifs régulièrement

C'est juste plus compliqué...

Analogie : la cuisine

- Des cuisiniers dans une cuisine
- Des recettes rédigées sur papier
- Des plats en train d'être préparées
- Des entrées-sorties dans le four
- Des clients qui veulent un service rapide

- Des processus dans un ordinateur
- Des programmes enregistrés sur le disque
- Des processus en train d'être exécutés
- Des entrées-sorties sur les périphériques
- Des utilisateurs qui veulent un service rapide

Analogie : la suite

- Un cuisinier peut préparer plusieurs plats en même temps
  Voire plusieurs plats d'une même recette
- Un cuisinier optimise son temps
  Il fait autre chose plutôt que d'attendre devant le four
- Un cuisinier répartit son temps entre les plats
  Tous les plats avancent dans leur préparation
- Certaines tâches peuvent être plus prioritaires
  En particulier, s'il y a une odeur de brulé

Information des processus

Table des processus (ou *Process Control Block*)

Structure conceptuelle qui regroupe les informations d'un processus

- État du processus
- Registres du CPU : dont CO (*PC*), PP (*SP*), et mot d'état
- Info sur le processus : pid, priorité, utilisateurs, statistiques...
- Info sur la gestion mémoire
- Info sur les E/S: répertoire de travail, fichiers ouverts, blocs verrouillés

La nature et le détail varient d'un système à l'autre

Question

- Où est stockée la table des processus ?

PID

Chaque processus est identifié par un numéro : le pid

- `pid_t getpid(void)`

Chaque processus a un parent

- Hiérarchie de processus
- init est la racine de la hiérarchie (pid=1)
- `pid_t getppid(void)`

Outils en ligne de commande

`ps(1)` et `top(1)` donnent de l'information sur les processus

- Nombreuses options et variations plus ou moins compatibles entre systèmes Unix

```
$ ps aux
```

Colonnes intéressantes

- PID: identifiant du processus
- PPID: identifiant du processus parent
- START: date de création du processus
- TIME: temps passé sur le processeur
- COMMAND: ligne de commande originale

Pseudo-système de fichiers /proc

- Typiquement nommé `/proc`
- Un sous-répertoire par processus, utilisant le PID
- Beaucoup d'information bas niveau et fluctuante
- Voir le man `proc(5)`

Expose de l'information

- Sous forme plus ou moins humaine
- Plus ou moins portable entre Unix

Pourquoi /proc ?

C'est plus simple ainsi

- Pour le noyau d'exposer de l'information
- Pour les programmes d'aller chercher l'information
- Qu'un ensemble dédié (et fluctuant) d'appels système

$\rightarrow$ `ps`, `top`, etc. utilisent directement `/proc` à l'interne

Question

- Est-ce que `/proc` contourne les appels système ?

Entrées intéressantes de /proc

- `/proc/PID/exe` un lien symbolique vers l'exécutable
- `/proc/PID/cwd` un lien symbolique vers le répertoire de travail
- `/proc/PID/cmdline` la ligne de commande utilisée (`\0` sépare les arguments)
- `/proc/PID/environ` les variables d'environnement
- `/proc/PID/stat` et `proc/PID/status` de l'information brute (utilisateurs, priorités, statistiques)
- `/proc/PID/maps` l'organisation de la mémoire (on y reviendra)
- `/proc/PID/fd/` les descripteurs de fichiers utilisés (on y reviendra)
- `/proc/PID/task/` les threads du processus (on y reviendra)
- `/proc/self` lien symbolique vers le processus courant

```
$ ls -l /proc/self/exe
$ readlink /proc/self/exe
$ cat --show-nonprinting --number /proc/self/cmdline
$ lolcat /proc/self/cmdline
```

210 Threads

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Thread système

Fil d'exécution indépendant d'un processus

- On parle parfois aussi de « processus léger »

Pas d'isolation des threads

- Même programme
- Même contexte
- Mêmes ressources

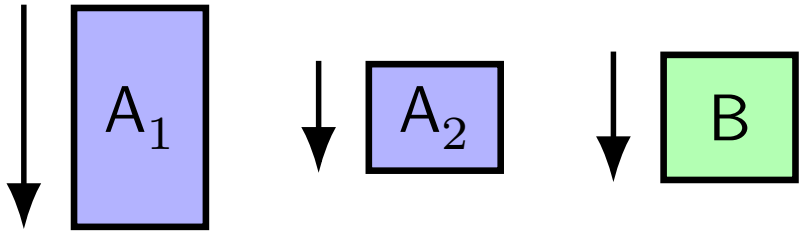
Mais exécution concurrente

- Permet des modèles de programmation intéressants
- Mais souvent difficiles...

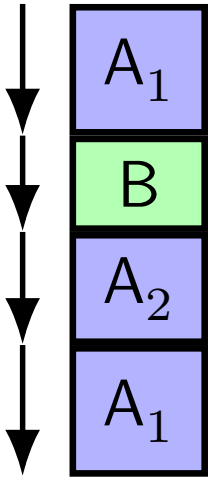
Exemple

- Un processus A avec 2 threads A_1 et A_2
- Un processus B monothread

Chaque thread progresse indépendamment



L'illusion de parallélisme est maintenue



Thread vs. Processus

Pour les threads d'un même processus

Propre à chacun

- Des registres (dont le CO et PP)
- Une pile d'exécution (pointée par PP)
- Priorité d'exécution

Partagé (en général)

- Programme en cours d'exécution (et bibliothèques)
- Sections mémoires (dont le tas)
- Fichiers ouverts

Avantages et inconvénients des threads

Avantages

- Moins cher à créer (un peu)
- Changement de contexte moins cher (un peu)
- Partage de données plus facile

Inconvénients

Synchronisation (très) difficile (on y reviendra)

- Un bogue dans un thread corrompt les autres
- Un thread compromis, compromet les autres

$\rightarrow$ Les navigateurs web modernes sont passé d'un thread par onglet à un processus par onglet

Programmation multithreads

Avant tout un **modèle de programmation**

Exposé par

- Des langages de haut niveau
- Des bibliothèques

Pour résoudre des problèmes variés

- Interface graphique réactive
- Traitements réseau asynchrones
- Calcul haute performance

Défis spécifiques

- Voir INF5171 Programmation concurrente et parallèle

Modèles d'implémentation multithread

Programmation multithread $\neq$ threads système

Plusieurs modèles d'implémentation multithread existent

Thread système (1:1)

- Le langage expose les threads système
- Le programmeur les manipule directement

Thread utilisateur (N:1)

- Les threads sont 100% gérés par le processus
- Le SE ne voit rien

Modèle hybride (M:N)

- C'est compliqué...

Thread utilisateur (N:1)

Géré 100% par le processus

- Offert souvent par des VM de langages
- Avec des astuces de programmation
- On parle aussi de *green thread*

$\rightarrow$ Juste un gros processus monothread compliqué

Avantages

- Portable entre différents SE
- Plus efficace dans certaines conditions
  - Pas de changement de contexte noyau

Inconvénients

- Changement de contextes utilisateur complexe à programmer
- Entrées-sorties bloquantes bloquent tout le processus
- Profite mal de la gestion optimisée des threads systèmes
- Profite mal des architectures multi-cœurs

Thread Posix (ou pthreads)

- API portable entre systèmes Unix
- Pour la programmation multithread système (1:1)
- Profite des threads système de chaque système d'exploitation
- voir le man `pthreads(7)` pour le point d'entrée
- On y reviendra...

Thread Linux (depuis v2.6, 2003)

- « **task** » (tâche) : seule abstraction de base
- Un processus monothread est juste une *task*
- Un processus multithread est un ensemble de *task*
  - Appartiennent à un même « *thread group* »
  - Un thread principal (*thread group leader*) représente le processus en entier
  - Le PID du processus est l'ID du thread principal

Voir et utiliser les threads Linux

Pour les voir

- `ps(1)`: `ps -Lf -C mysqld`
- `proc(5)`:
  - `/proc/` a une entrée **invisible** par thread
  - `/proc/PID/task/` pour les threads d'un même processus (« *thread group* »)

```
$ ls -l /proc/$(pgrep mysqld)/task
```

Pour programmer avec

Attention : Les threads Linux ne sont pas portables aux autres Unices

- Appels système spécifiques bas niveaux: `clone(2)`, `gettid(2)`, `tgkill(2)`...
- Pour du « vrai code », utiliser les `pthreads(7)`

Confusion terminologique

En fonction du contexte (et de l'auteur de la documentation)

- « processus (*process*) » peut désigner un processus ou un thread
- On trouve aussi « tâche (*task*) » pour compliquer plus

Autres regroupements de processus

- Groupe de processus: généralement utilisé pour les conduites shell (*pipelines*)
- Session: généralement utilisé pour grouper les connexions indépendantes d'utilisateurs

220 Espace mémoire des processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Mémoire des processus

Le SE gère l'organisation de la mémoire

- Le SE est responsable de la cohérence et du nettoyage de la mémoire de l'ordinateur
- La gestion effective de la mémoire dépend du SE et des capacités matérielles

Un processus ne voit que son propre espace mémoire

- Accéder à un espace qui n'est pas le sien est interdit
- Tout est autorisé dans son espace mémoire

$\rightarrow$ Peut corrompre ses propres données en mémoire (bogues)

$\rightarrow$ Mais ne peut corrompre les autres processus

Segments mémoires

4 segments principaux

Vision simpliste à la INF2171

- Code (*text*): le code machine du programme
- Données statiques (initialisées et non initialisées)
- Tas (qui croît vers le bas)
- Pile (qui croît vers le haut)

Extra

- Bibliothèques (code et données)
- Piles supplémentaires (threads)
- Mémoire anonyme et projection de fichiers
- Etc.

Fonctionnement de la pile (rappel INF2171)

On empile

- Des cadres d'exécution fonctionnels (*stackframe*)

Qui contiennent

- Les variables automatiques (variables locales)
- Les paramètres des fonctions
- La place pour les valeurs de retour
- Des valeurs pour la gestion des appels de fonctions (adresse de retour, base de pile, etc.)
- Taille fixée (8Mo pour Linux) mais modifiable par `ulimit (bash)`, `prlimit(1)`, `setrlimit(2)`
- Contient aussi les arguments du programme (`argv`)
- Et les variables d'environnement (`environ(7)`)

Fonctionnement du tas (rappel INF2171)

Allocation (et désallocation) dynamique

- Mémoire réservée quand elle est nécessaire
- Et libérée quand elle ne l'est plus

$\rightarrow$ Contient les données importantes des *vrais* programmes

Gestion programmative

Le programme décide des allocations et des désallocations

Les langages fournissent des mécanismes

- Fonctions bibliothèque. Ex. `malloc(3)` et `free(3)` en C
- Mots clés. Ex. `new` et `delete` en C++
- Ramasse-miettes. Ex. Java

Appels système `brk(2)` et `mmap(2)` : pour demande d'espace mémoire

Chargement des programmes

Contenu des programmes exécutables (binaires)

- Code en langage machine
- Données binaires
- Métadonnées pour chargement et édition de lien (entre autres)

Format des exécutables et bibliothèques dynamiques

- Unix (multi-plateforme): ELF (*Executable and Linking Format*) pour exécutables et `.so`
- Windows: PE (*Portable Executable*) pour `.exe` et `.dll`

Initialisation en mémoire (chargement)

- Provient du fichier binaire (presque tel quel)
  code et données initialisés
- Réservé par le système d'exploitation (et initialisé à 0)
  données non initialisées (BSS), tas et pile (avec argv et environ)

Segments mémoire - Exercice

```
#include <stdlib.h>
#include <unistd.h>
#define K 32*1024
const int L=K;
int T[K];
void foo(int t) {
  int R[K];
  if (t>0) foo(t-1);
}

int main(int argc, char **argv) {
  foo(2);
  int *S = calloc(K, sizeof(int));
  pause();
  return 0;
}
```

Questions

- Quels sont les objets du programme ?
- Quelle est leur taille ?
- Dans quels segments sont-ils alloués ?
- Quelle est leur durée de vie ?
- Quelle est la taille minimum de l'exécutable ?

Qui décide de la vraie organisation ?

- Compilateur C, éditeur de liens, éditeur de lien dynamique
- Peuvent décider d'organiser l'exécutable et la mémoire de nombreuses façons

Droits de la mémoire

Dans les processeurs modernes

- Les zones mémoires ont des droits
- Lecture (r), écriture (w), exécution (x)
  exécution = avoir compteur ordinal dessus
- Configuré par le système d'exploitation
  et par l'appel système `mprotect(2)`

Droits habituels des segments

- Code machine: `r-x`
- Données statiques en lecture seule: `r--`
- Données statiques en lecture écriture: `r-w`
- Tas: `rw-`
- Pile: `rw-`

Voir l'organisation mémoire

Pour voir l'organisation de la mémoire d'un processus

- Commande `pmap(1)`
- Pseudo-fichier `/proc/PID/maps`

→ du point de vue du système d'exploitation

```
00005616b8c7f000      4K r---- orgamem
00005616b8c80000      4K r-x-- orgamem
00005616b8c81000      4K r---- orgamem
00005616b8c82000      4K r---- orgamem
00005616b8c83000      4K rw--- orgamem
00005616b8c84000    128K rw---   [ anon ]
00005616ba1fc000    132K rw---   [ anon ]
00007eff68902000    148K r---- libc-2.31.so
[...]
00007eff68b31000      4K rw--- ld-2.31.so
00007eff68b32000      4K rw---   [ anon ]
00007ffe44262000    396K rw---   [ stack ]
00007ffe44378000     16K r----   [ anon ]
00007ffe4437c000      8K r-x--   [ anon ]
```

230 Vie et états des processus

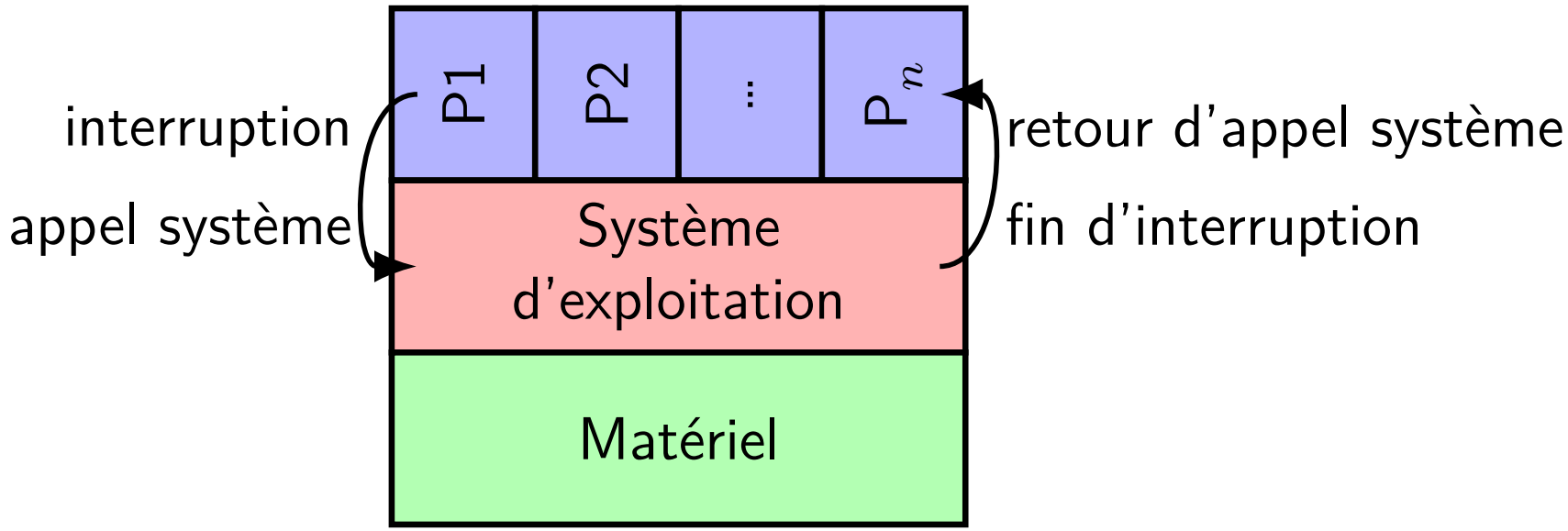
INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

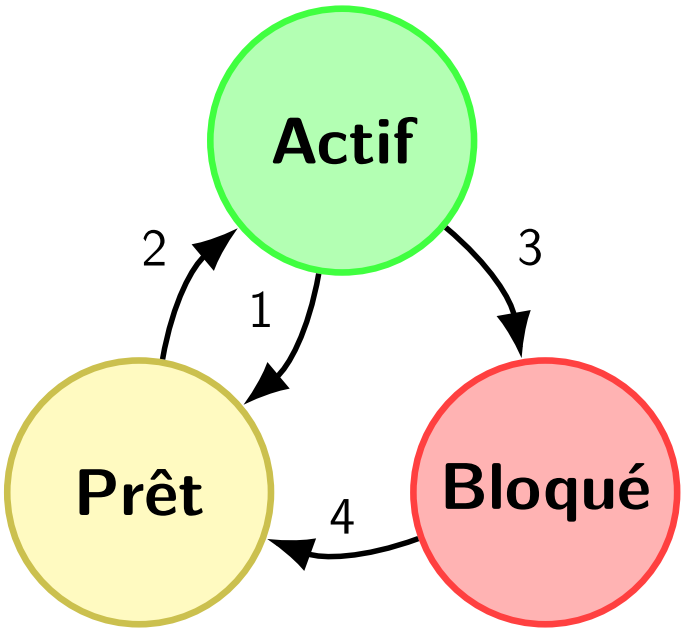
Hiver 2021

Processus et SE (Rappel)



- Un processus est actif sur le processeur
- Il fait un appel système (volontaire)
  OU une interruption matérielle survient (involontaire)
- Le processeur est passé au système (mode noyau)
- Le système traite l'appel système ou l'interruption
- Puis rend le processeur à un processus (mode utilisateur)

États d'un processus



- **Actif**: tient la ressource processeur
- **Prêt**: ne manque que le processeur
- **Bloqué**: manque une autre ressource

- 1 Le SE prend la main à un processus
- 2 Le SE donne la main à un processus
- 3 Le processus demande une ressource
- 4 La ressource demandée devient disponible

Questions

- Citer un exemple pour chaque changement
- Un processus peut-il passer de **prêt** à **bloqué** ?
- De **bloqué** à **actif** ?

États Unix

`ps(1)` sous Linux décrit des états un peu différents

- R s'exécute ou peut s'exécuter → Regroupe **prêt** et **actif**
- S/D en sommeil interruptible/ininterruptible
  → **Bloqué** sur un appel système.
  Si l'appel système a un code d'erreur `EINTR` l'appel est interruptible par un signal (`kill(1)`).
  Sinon, l'appel système fait des entrée-sorties mais termine vite (on voit rarement D apparaitre)
- T/t arrêté par le signal de contrôle de tâche/le débogueur
  → **Bloqué** par un signal spécial (`^Z`) ou par un débogueur
  Le processus peut continuer si on le débloque
- Z processus zombie (`defunct`) → On y reviendra...
- I fil inactif du noyau
  → Linux *utilise* l'infrastructure des processus pour gérer ses tâches noyaux internes. I n'a pas de sens autrement.

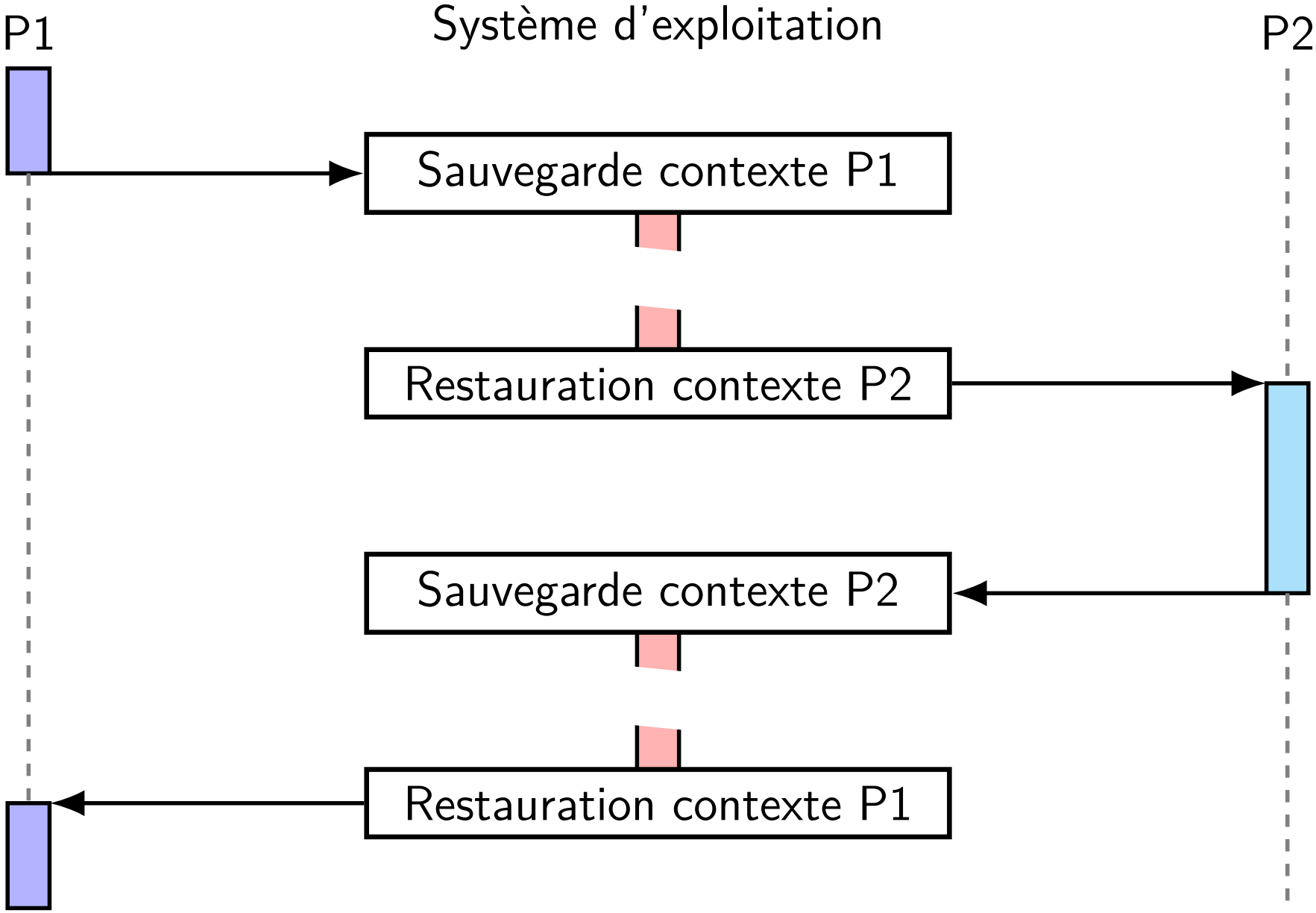
Changement de contexte

Un **changement de contexte** intervient quand le processeur est donné à un autre processus

- Passage du processus en mode noyau (syscall ou interruption)
- Sauvegarde du contexte du processus
- Exécution de l'appel ou gestion de l'interruption
- Modification éventuelle de l'état de processus
- Appel de l'**ordonnanceur** (*scheduler*) qui élit un processus
- Restauration du contexte de l'élu
- Modification de son état à **actif**
- Fin de l'appel ayant précédemment provoqué la suspension de l'élu
- Passage de l'élu en mode utilisateur et suite de son exécution

$\rightarrow$ C'est un procédé complexe et relativement coûteux

Changement de contexte



Coût du changement de contexte

Un changement de contexte coûte cher

- Rappel : le travail du SE n'est pas un travail utile en soi

Cause du coût

- Sauvegarde du contexte du processus
- Algorithme d'ordonnancement (détails plus tard)
- Restauration du contexte du processus
- Deux fois mise en défaut des caches et sections

Scénario (en quatre actes)

Personnages

- 3 processus (P1 **actif**, P2 et P3 **prêts**)
- Un système d'exploitation (accompagné de son ordonnanceur)

Acte 1

- P1 fait un `read(2)` avant l'épuisement de son quantum
- Passage en mode noyau et exécution de l'appel système `read`
- P1 passe à **bloqué**
- Sauvegarde du contexte de P1
- Appel de l'ordonnanceur

Questions

- Pourquoi `read` bloque le processus ?
- Est-ce que `read` bloque toujours le processus appelant ?

Scénario (suite)

Acte 2

- P2 est élu
- Le contexte de P2 est restauré
- P2 passe à **actif**
- L'horloge est programmée pour un nouveau quantum
- Fin de l'appel système/de l'interruption qui avait suspendu P2
- Et suite de l'exécution de P2 (en mode utilisateur)

Questions

- Pourquoi pas P1 ?
- Pourquoi P2 et pas P3 ?

Scénario (suite)

Acte 3

- P2 accapare le processeur
  - Ne fait pas d'entrée-sortie
  - Consomme entièrement son quantum de temps
- Interruption de l'horloge programmable
- Exécution du gestionnaire d'interruption (mode noyau)
- P2 passe à **prêt**, sauvegarde du contexte
- Appel de l'ordonnanceur

$\rightarrow$ Il y a eu **préemption**

Questions

- Est-ce que P2 pourrait être élu à nouveau ?

Scénario (suite)

Acte 4

- P3 est élu
  - Passe à **actif**
  - Son contexte est restauré
  - Suite de son exécution (en mode utilisateur)
- La donnée demandée par P1 arrive
  - Interruption du contrôleur de disque
- Exécution du gestionnaire d'interruption (mode noyau)
  - La donnée est placée en mémoire, accessible à P1
- P1 passe à **prêt** (on parle de réveil)
- P3 passe aussi à **prêt**, sauvegarde du contexte
- Appel de l'ordonnanceur

Question

- Pourquoi P3 passe à **prêt** plutôt que de continuer ?

Utilisation de la ressource processeur

Commandes et appels système

- `time(1)` décompte le total de ressources
  Utilisez `/usr/bin/time` pas la commande shell pour plus d'options
- `getrusage(2)` pour l'information en temps réel
  Le processus qui demande pour lui-même
- `ps(1)` et `top(1)` peuvent aussi présenter de l'information
- L'information est aussi dans `/proc/PID/stat` et `/proc/PID/status`
  Voir `proc(5)` pour les détails

Ressources CPU (de la commande time)

- `%E` Temps réel mis par le processus
  Heure de fin moins heure de début (en vraies secondes)
- `%U` Temps processus utilisateur utilisé
  Somme (pour chaque thread) du temps passé à l'état **actif**
- `%S` Temps processus système utilisé
  Somme du temps passé à l'état **actif** mais en mode noyau
  C'est à dire le travail fait par le SE au bénéfice du processus
- `%P` Pourcentage du processeur utilisé
  C'est juste (U+S)/E
- `%w` Nombre de changements de contextes volontaires
  Passages de **actif** à **bloqué**
- `%c` Nombre de changements de contextes involontaires
  Passages de **actif** à **prêt**

Question et exercices

- Où le noyau conserve le décompte de l'utilisation des ressources des processus ?
- Est-ce que %U peut être plus grand que %E ?
- Est-ce que %S peut être plus grand que %U ?
- Quelle est la valeur maximale de %P sur un système ?
- Comment avoir une grande ou une petite valeur de chacun des indicateurs (toutes choses étant égales par ailleurs) ?
- Cherchez l'équivalent des ressources dans `getrusage(2)`, `ps(1)`, `top(1)`, et `proc(5)`

Utilité des caches

La plupart des appels système d'entrée-sortie peuvent retourner sans bloquer le processus si l'information est disponible en cache.
Pour forcer le vidage de cache sous Linux:

```
$ sync
$ echo 3 | sudo tee /proc/sys/vm/drop_caches
```

240 Création et terminaison

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Création des processus

Quand un processus est-il créé

Au démarrage du système

- Démons/serveurs/services

À la demande d'un utilisateur

- Double-clic sur une icône
- Commande shell
- Job soumis dans les systèmes de traitement par lots (batch)

À la demande d'un programme

- Appel système spécifique

Création des processus en vrai

Techniquement

- Un processus lance d'autres processus
- Sauf un processus particulier

Création des processus en vrai

Techniquement

- Un processus lance d'autres processus
- Sauf un processus particulier
  - `init` de PID 1
  - Qui est créé directement par le système

Appels système

- Unix: `fork(2)` (0 paramètre) et `execve(2)` (3 paramètres), hiérarchie de processus
- Windows : `CreateProcess` (10 paramètres), pas de hiérarchie

Génération de processus

Approche générale

- Vérifier l'existence (et droits) de l'exécutable
- Réserver une entrée dans la table des processus
- Réserver l'espace mémoire nécessaire
- Charger le code et les données statiques
- Initialiser/mettre à jour les données du système
- Mettre en place les fichiers ouverts par défaut
- Initialiser le contexte (compteur ordinal, etc.)

Sous Unix

- Création d'un clone (copie du demandeur) : `fork(2)`
- Chargement d'un nouveau programme (à la place du demandeur) : `execve(2)` et dérivés

hello_echo.c

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  pid_t pid = fork();
  if (pid<0) { // Erreur
    perror("fork");
    return 1;
  } else if (pid==0) { // Processus fils
    execlp("echo", "echo", "Hello, World!", NULL);
    perror("exec");
    return 1;
  } else { // Processus père. On attend le fils.
    wait(NULL);
    return 0;
  }
}
```

Alternatives à fork, exec et cie

Fonction `system`

- `system(3)` exécute une commande shell en avant plan
- En gros: fork+exec+wait de `sh -c` + gestion saine des signaux
- Attention à la sécurité: INF600C (injection de commande ou de PATH)

Fonction `popen`

- `popen(3)` tube avec une commande shell en arrière plan
- En gros: pipe+fork+exec de `sh -c`
- Attention à la sécurité
- On verra les tubes (pipe) plus tard

Fonction `posix_spawn`

- `posix_spawn(3)` combine de `fork` et `exec`
- Compliqué à utiliser

241 fork et création de processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Principes de fork

`fork(2)`

- Crée une **copie** de l'appelant
- Parent et enfant auront le **même code**
- Ils continueront leur exécution **indépendamment** l'un de l'autre
- Le parent reconnaît son enfant nouvellement créé

Question

- Comment l'enfant reconnaît son parent ?

Algorithme de fork

si *ressources système insuffisantes* **alors**
    positionner errno;
    retourner -1;
fin
obtenir nouvelle entrée dans la table des processus;
obtenir nouveau numéro de processus;
initialiser table[enfant];
marquer état enfant en cours de création;
« copier » segments mémoire du parent dans l'espace du nouveau;
incrémenter le décompte des fichiers ouverts;
marquer état enfant prêt;
si *processus en cours est le parent alors* **alors**
    retourner numéro enfant;
sinon
    retourner 0;
fin

Points clés de fork

- Parent et enfant partagent le même code
- Enfant a une copie des données du parent
- Parent et enfant partagent les fichiers ouverts
- La valeur de retour de `fork` permet de différencier parent et enfant

Exemple de fork

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

int main() {
  pid_t pfils;

  printf("Je suis %d, je commence\n", getpid());
  pfils = fork();
  if (pfils == -1) {
    perror("Echec du fork");
  } else if(pfils == 0) {
    printf("Je suis %d, le fils de %d\n", getpid(), getppid());
  } else {
    printf("Je suis %d, le pere de %d\n", getpid(), pfils);
  }
  printf("Je suis %d, je finis\n", getpid());
}
```

Plusieurs forks

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

int main() {
  printf("Gen 1\n");
  fork();
  printf("Gen 2\n");
  fork();
  printf("Gen 3\n");
  fork();
  printf("Gen 4\n");
}
```

Copie de la mémoire

La mémoire est « copiée » telle quelle

Il y a des optimisations

- La copie de la mémoire est paresseuse
- Et souvent, elle est juste partagée

→ Mais le système fait semblant que oui

Il n'y a pas de sémantique particulière

- Les zones sont toutes « copiées »
- code, pile, tas, bibliothèques, etc.

→ Un processus est responsable de l'organisation de sa mémoire

Attention aux tampons en espace utilisateur

- Les effets peuvent être surprenants

→ Pensez à `fflush(3)` avant

Forkbomb

Dénis de service

- Demande infinie de création de processus

→ **Famine** CPU, mémoire, table des processus

- Le nombre de demandes croît exponentiellement

→ Chaque processus en engendre 2

- Il est difficile de guérir

→ Plus assez de ressource pour lancer un processus qui nettoie tout ça

→ Dès qu'un processus est tué, un autre prend sa place

Exemples

- En C « `for(;;){fork();}` »
- En shell « `:(){ :|:& };:` »

Prévention forkbomb

Limiter le nombre maximal de processus par utilisateur (ou autre)

- `ulimit -u` commande interne du shell
- `/etc/security/limits(conf)` configuration globale (PAM)
- `setrlimit(2)` appel système sous-jacent
- `/proc/PID/limits` voir les limites de chaque processus

Et si jamais...

- `pkill -STOP -u john` puis `pkill -KILL -u john`
- ou redémarrer la machine
  (et mettre des limites pour la prochaine fois!)

Question

- Pourquoi `killall nomcommande` ne fonctionne pas directement ?
- Comment encore c'est possible en 2020 ?

Appel système Linux `clone`

- `clone(2)` appel système similaire à fork
- Évite certaines limitations de l'API de `fork`
- `clone3(2)` version moderne de clone avec une encore meilleure API
- Utilisé pour processus, threads et autres bêtes hybrides

Contexte d'exécution

Contrôle très précis du contexte d'exécution

- Partages mémoires
- Espaces de noms
- Cgroups
- En particulier utilisé pour les conteneurs Linux

242 exec et recouvrement de processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Recouvrement de processus

Principe : demander à changer de programme exécuté

- Vérifier l'existence et droits d'exécution
- Écraser le segment de code avec le nouvel exécutable
- Écraser les données statiques
- Réinitialiser tas et pile
- Positionner correctement les registres
- Mettre à jour les données internes du SE

Autodestruction sans risque

Autodestruction car

- Pendant un recouvrement, code et données sont inutilisables
- À la fin il n'en reste rien

Mais c'est sans risque car

- Tout est fait par le SE avec les données du SE
- Le code et les données du processus ne participent pas au recouvrement (heureusement)

Appel système exec

En fait une famille de fonctions

Un appel système (section 2)

```
int execve(const char *filename, char *argv[], char *envp[])
```

Fonctions pratiques (section 3)

`execl, execlp, execle, execv, execvp`

- v : passage par vecteur (`char *argv[]`)
- l : passage par liste (`char *argv, ...`)
- p : utilisation de PATH pour trouver l'exécutable
- e : précision des variables d'environnement

Choses perdues après un execve

Perdu : quelques trucs

- Segments mémoires (code, données statiques, tas, pile, etc.)
- Threads
- Gestionnaires de signaux...

Conservé : tout le reste

- Identité : pid, parent, etc.
- Caractéristiques : Utilisateur, droits, priorité, etc.
- Entrées sorties : répertoire courant, fichiers ouverts, etc.
- Statistiques : consommation ressources

Exemples de exec

Utilisation de execl

```
execl("/bin/ls", "ls", "/etc", NULL);
perror("Échec du exec");
exit(1);
```

Utilisation de execlp

```
execlp("ls", "ls", "-l", "/usr", NULL);
perror("Échec du exec");
exit(1);
```

Exemple exec.c

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

int main(int argc, char **argv) {
  printf("On exécute %s avec %d arguments!\n",
      argv[1], argc-2);
  execvp(argv[1], argv+1);
  perror(argv[1]);
}
```

Chargement des exécutables

Format des exécutables binaires

- ELF pour Unix
- PE pour Windows
- D'autres formats historiques ou +/- répandus existent

$\rightarrow$ Un noyau pourrait connaitre plusieurs formats

ELF: Executable and Linking Format

- `elf(5)` pour le format
- `objdump(1)` ou `readelf(1)` pour afficher l'information
- `nm(1)` pour juste lister les symboles

Contenu des exécutables binaires

De l'information pour le système

- Quels blocs d'octets charger ?
- À quelle adresse dans la mémoire ?
- Avec quels droits rwx ?
- Quelle est la taille du BSS ?
- Etc.

Et plein d'autres choses

- Pour l'éditeur de liens
- Pour l'éditeur de liens dynamiques
- Pour le débogueur
- Etc.

Point d'entrée

Indique l'adresse de la première instruction machine du programme

Symbole `_start`

- Sous Unix, c'est souvent le point d'entrée
- L'éditeur de liens décide en fait

Quoi faire entre `_start` et `main` ?

- Charger les bibliothèques (dont la libc!)
- Préparer des segments mémoires
- Instancier des objets globaux

$\rightarrow$ Le processus est responsable

Démarrer sans rien

```
// pas de libc ni rien
// gcc nostart.c -nostdlib -static -e debut -o nostart

int ecrit(int fs, char* msg, long len)
  { asm("mov $1, %rax; syscall"); }
int quitte(int code)
  { asm("mov $60, %rax; syscall"); }

void debut(void) {
  ecrit(1, "Hello, World!\n", 14);
  quitte(0);
}
```

- Pas portable, mais ça fonctionne

Interpréteurs de scripts (*shebang*)

- Si un fichier est exécutable et commence par « `#!` »
  Exemple: « `#!/chemin/foo argument` »
- Le système exécute le programme `/chemin/foo`
- Avec le chemin du fichier en argument

→ Ça peut être n'importe quel programme

→ C'est automatique

```
$ cat shebang
#!./showargs monargument
$ ./shebang a b c
arg 0: ./showargs
arg 1: monargument
arg 2: ./shebang
arg 3: a
arg 4: b
arg 5: c
exe: /usr/local/bin/showargs
```

Utilisation habituelle du shebang

Exécuter un programme dans un langage de script

- Exemple « `#!/bin/bash` »

Chemin absolu

Problème: il faut un chemin absolu

- Solution utiliser `/usr/bin/env`
- `env(1)` exécute un programme trouvé dans le `PATH`
- Exemple « `#!/usr/bin/env python` »

Question

- Pourquoi `#` ?

Utilisation inhabituelle

Qu'affiche ce programme ?

```
#!/usr/bin/tac
   (/`-'\)
    /    \(
      )U(   _)
   =(_*_)= (
  '\`o.O'    _
   _   ,/|
```

Formats binaires divers

`binfmt_misc` (*miscellaneous binary format*)

- Permet d'associer des interpréteurs à des binaires quelconques
- En fonction de l'extension ou d'un nombre magique

Exemples

- Exécuter des `.jar` directement avec `java`
- Exécuter des `.exe` directement avec `wine`

```
$ file hello.jar
hello.jar: Java archive data (JAR)
$ ./hello.jar
Hello world!
```

Exécutables dynamiquement liés

- En gros, une bibliothèque dynamique exécutable
- Mode de compilation par défaut des distributions modernes
- On parle de PIE (*position independent executable*)

Problème

Pour être utilisable, une bibliothèque doit être liée

Solution

- Champ `PT_INTERP` (ELF) indique le chemin d'éditeur de liens dynamiques (habituellement `ld.so`)
- Le noyau le charge et l'exécute
- Qui va lier et exécuter le programme ?
- C'est de la vraie magie noire

243 exit et terminaison de processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Terminaison volontaire des processus

Terminaison normale

- Le processus a terminé son travail
- L'utilisateur a fait `fichier/quitter`
- Etc.

Terminaison suite à une erreur

- Arguments erronés pour un programme en ligne de commande
- Format de fichier non reconnu
- Etc.

Dans les deux cas

- Appel système `exit(3)`
- Retour de la fonction `main`

$\rightarrow$ Indique une valeur de retour

- Convention: $0 \rightarrow$ OK, $\neq 0 \rightarrow$ voir le man

Terminaison involontaire des processus

Terminé par un autre processus

- Via l'appel système `kill(2)`, s'il a les droits
- On y reviendra...

Erreur fatale (généralement : bogue du programme)

- Les **fautes** du CPU sont la cause principale

$\rightarrow$ Division par 0, erreur de segmentation, etc.

Terminé par le système

- Ressource manquante (mémoire)
- Arrêt du système (*shutdown*)

Gestion des signaux Unix

Dans la plupart de ces cas un processus peut être notifié pour gérer son interruption involontaire

Fin des processus

Le SE doit

- Fermer les fichiers ouverts
- Informer le parent (signal SIGCHLD)
- Faire adopter les enfants par init (ou autre)
- Marquer les zones mémoires comme libres
- Mettre à jour ses structures de données internes (statistiques et nettoyage)
- Ne pas réutiliser le PID trop tôt

Questions

- Pourquoi le SE s'occupe-t-il de faire tout ça ?
  Ne peut-il pas laisser ça au programme ?
- Pourquoi on demande aux apprentis programmeurs de libérer quand même les ressources ?

Terminer un processus

Fonction exit

- `exit(3)` « `void exit(int valeur_de_sortie)` »

Généralement, la valeur de sortie vaut

- `EXIT_SUCCESS` (0) si tout s'est bien déroulé
- `EXIT_FAILURE` (1) en cas d'erreur

Questions

- Quels sont les cas d'erreur d'exit ?
- Pourquoi exit ne retourne pas de valeur ?

Fausses sorties

- `exit(3)` est une fonction de bibliothèque (C ISO/IEC)
- Effectue des actions programmées
- Puis termine le processus

Actions programmées

- Flush les entrée-sortie de `stdio(h)`
- Supprime les fichiers créés par `tmpfile(3)`

→ `atexit(3)` ajoute une action programmée

C'est fait côté bibliothèque

- Conservées par un `fork(2)`
- Perdues par un `execve(2)`

→ Non appelé si terminaison par un signal ou une **vraie** sortie

Question

- Que se passe-t-il si une fonction enregistrée par `atexit` appelle `exit` ?

Vraies sorties

Le vrai appel système (POSIX)

- `_exit(2)` termine le processus immédiatement
- Sans faire les actions programmées

Multi-threads (POSIX)

- `pthread_exit(3)` termine le thread courant
- Mais effectue les actions programmées, si c'était le dernier thread

Le vraiment vrai appel système (Linux)

- Sous GNU Linux, `_exit(2)` appelle `exit_group(2)`
- `exit_group(2)` termine toutes les tâches (threads) du processus
- Vrai appel système Linux « `exit` », ne termine que la tâche courante
  Pas d'enveloppe dans la glibc.

Exemple: exit.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

void bye(void) { printf(", le monde!\n"); }
int main(int argc, char **argv) {
  atexit(bye);
  int i = atoi(argv[1]); // pas d'argument => segfault
  printf("Bonjour %d!\n", i);
  printf("Au revoir");
  switch(i) {
    case 0: return i;
    case 1: exit(i);
    case 2: _exit(i);
    case 3: syscall(SYS_exit_group, i);
    case 4: syscall(SYS_exit, i);
  }
}
```

Attendre la terminaison

Appel système wait

- `wait(2)` « `int wait(int *status)` »
- `waitpid(2)` « `int waitpid(int pid, int *status, int option)` »
- Permet à un parent d'attendre la fin de l'exécution d'un enfant

Arguments

- pid : l'enfant à attendre
- status : raison de terminaison (code de retour ou n° de signal)
- option : diverses options (voir man)

Exemple de wait

```
int main(int argc, char **argv) {
  pid_t pfils = fork();
  if (pfils == -1) { perror("Echec du fork"); return 1; }
  if(pfils == 0) {
    execvp(argv[1], argv+1);
    perror(argv[1]); return 1;
  }

  printf("J'attend %d...\n", pfils);
  int status;
  int w = wait(&status);
  if (w == -1) { perror("waitpid"); exit(1); }
  if (WIFEXITED(status)) {
    printf("Status=%d\n", WEXITSTATUS(status));
  } else if (WIFSIGNALED(status)) {
    psignal(WTERMSIG(status), argv[1]);
  }
  return 0;
}
```

Processus zombi

Le SE conserve les informations d'un processus

- Raison de la terminaison
- Code de retour / numéro du signal
- Ressources consommées (voir `wait3(2)` et `wait4(2)`, non-POSIX)

$\rightarrow$ À l'intention du parent

État zombi

- Durant ce temps, le processus enfant est dans un état zombi (repéré par un Z et un *defunct* lors d'un ps)
- Coût d'un zombi : une entrée dans la table des processus
- Quand le parent s'informe (`wait(2)`), ces informations sont nettoyées

Un zombi ne consomme pas d'autre ressource

init

- Si un processus se termine, ses enfants sont hérités par `init` (tous les enfants, zombis ou non)
- `init` effectue les `wait(2)` nécessaires à leur nettoyage.

subreaper

- Sous Linux par des *subreaper* autre que `init` peuvent être définis
- Voir appel système `prctl(2)`

250 Ordonnancement des processus

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Ordonnanceur

Qu'est-ce que c'est

- C'est une partie du SE
- Il sert à déterminer quels processus sont actifs (et lesquels ne le sont pas)

Ses règles de base

- Ne peut qu'élire un processus prêt
- L'élu est celui qui a la plus haute priorité compte tenu de la politique locale

Schéma algo ordonnanceur

tant que *pas de processus élu* **faire**
    consulter liste des processus prêts;
    sélectionner celui qui a la plus haute priorité;
    si *pas d'élu* **alors**
        attendre jusqu'à la prochaine interruption
        (processeur à l'état latent);
    fin
fin
marquer le processus élu actif;
basculer le contexte;

Quand intervenir ?

Aux changements d'état

- Création de processus
- Terminaison d'un processus
- Passage d'actif à bloqué (demande d'E-S)
- Passage de bloqué à prêt (ressource disponible)
- Passage d'actif à prêt (fin de quantum)

$\rightarrow$ Mais aussi si changement de priorité (ou d'ordonnanceur)

Concrètement ?

- Appel système
- Interruption matérielle, dont l'horloge programmable
- **Question.** Ces deux cas couvrent-ils toutes les possibilités ?

Ordonnanceurs non-préemptifs

Processus actif jusqu'à

- Une demande d'entrée-sortie bloquante
  (ou tout autre appel système bloquant)
- Une demande explicite de laisser la main (`sched_yield(2)`)

$\rightarrow$ Dans les deux cas, c'est à la demande du processus

- On parle aussi de « multitâche coopératif »
- Très rare dans les systèmes modernes

Question

- Et si on laissait la main à un processus bloqué par une E-S ?

Ordonnanceurs préemptifs

À n'importe quel moment, on peut suspendre un processus

Fin du tour

- Expiration d'un quantum de temps alloué au processus

$\rightarrow$ Interruption matérielle due a l'horloge programmable

Perte de priorité

- Nouveau processus prioritaire créé
- Processus prioritaire qui passe de bloqué à prêt
- Changement de priorité dans les processus

Question

- Quels sont les avantages du préemptif sur le non-préemptif ?

Objectifs d'ordonnancement

- Respect de la politique locale
  - Les processus plus prioritaires ont plus la main
- Équité
  - Tous les processus de même priorité ont autant la main l'un que l'autre
- Efficience
  - Utilisation efficace des différentes ressources (processeur)

Problème : on ne peut pas toujours avoir les 3

Exemple: 3 processus de même priorité sur 2 processeurs

- On met deux processus sur un CPU et le 3e sur l'autre
  $\rightarrow$ Inéquitable
- On alterne et déplace les processus entre CPU
  $\rightarrow$ Inefficient

$\rightarrow$ il faut faire des compromis!

Critères d'évaluation

- Maximiser le nombre de tâches terminées par unité de temps
- Minimiser le temps entre acceptation et terminaison (temps total)
- Maximiser le temps d'utilisation du CPU
- Minimiser le temps d'attente (latence)
- Maximiser le temps de réponse (interactivité)

$\rightarrow$ Ils ne sont pas indépendants

Pour chaque critère

- En moyenne ?
- Au pire ?
- Au mieux ?

Beaucoup de critères possibles et on n'a encore rien fait en pratique

Objectifs d'ordonnancement spécifiques

Pour les systèmes interactifs

- Minimiser le temps de réponse
- Proportionnaliser le temps de réponse à la complexité perçue de la tâche

$\rightarrow$ Donner l'impression à l'utilisateur que le système est réactif

Pour les systèmes temps réel

- Respecter les contraintes de temps (au pire cas)
- Prédiction de la qualité de service
- Les systèmes temps réel ont des besoins spéciaux et des ordonnanceurs spéciaux
- On y reviendra

CPU bound vs. *I/O bound*

- *CPU burst* : le **temps de calcul** avant prochaine E-S (ou prochain appel système bloquant)

Programme *CPU bound*

- Le processeur est le facteur limitant
- Surtout des calculs, peu d'entrées-sorties
- *CPU bursts* probablement longs

Programme *I/O bound*

- Les entrées-sorties sont le facteur limitant
- Surtout des entrées-sorties, peu de calculs
- *CPU bursts* probablement courts

Questions

- En quoi savoir la catégorie aide l'ordonnanceur ?
- Peut-on catégoriser plus finement ?

Ordonnancement sous Linux

Plusieurs politiques cohabitent

- 3 « normales » : `SCHED_OTHER`*, `SCHED_BATCH`, `SCHED_IDLE`
- 3 « temps réel » : `SCHED_FIFO`*, `SCHED_RR`*, `SCHED_DEADLINE` (classes de priorité strictes)
- Tous sont préemptifs

Page de man

- `sched(7)`, `chrt(1)`, `sched_setattr(2)`

*norme POSIX

Stratégies d'ordonnancement standard

File d'attente (non-préemptif)

- Premier arrivé, premier servi
- FIFO (*first in, first out*)

Avantages

- Facile à comprendre : file d'attente à la caisse
- Facile à implémenter
- Équitable ?

Implémentation

- Une file de processus prêts
- La tête de file est le prochain élu
- Les processus qui (re)deviennent prêts $\rightarrow$ en fin de file

Exercice et simulation (file)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

Exercice et simulation (file)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

processus	temps d'attente	temps total
p1	0	9
p2	8	11
p3	10	13
minimum	0	9
moyenne	6	11
maximum	10	13

Alternative

processus	temps d'arrivée	temps de calcul
p4	2	9
p5	1	3
p6	0	3

Alternative

processus	temps d'arrivée	temps de calcul
p4	2	9
p5	1	3
p6	0	3

Faites-le chez vous :)

Files d'attente + priorité + préemption

- Des niveaux distincts de priorité
  Par exemple de 1 (faible) à 99 (forte)
- Une file d'attente par niveau de priorité
- Priorité stricte : prioritaire = passer toujours devant

Avantages

- Facile à comprendre : file d'attente au parc d'attractions
- Facile à implémenter
- Permet un contrôle de l'utilisateur (politique)

SCHED_FIFO (Posix)

- `chrt --fifo 90 macommande`
- **Question** Et si `macommande` part en boucle infinie ?

Exercice et simulation (files prioritaires)

processus	temps d'arrivée	temps de calcul	priorité[†]
p1	0	9	2
p2	1	3	3
p3	2	3	1

[†]Grand = prioritaire, petit = pas prioritaire

Exercice et simulation (files prioritaires)

processus	temps d'arrivée	temps de calcul	priorité[†]
p1	0	9	2
p2	1	3	3
p3	2	3	1

processus	temps d'attente	temps total
p1	3	12
p2	0	3
p3	10	13
minimum	0	3
moyenne	4.3	9.3
maximum	10	13

[†]Grand = prioritaire, petit = pas prioritaire

Tourniquet

- File d'attente + quantum de temps
- RR (*Round-robin*)
- Quantum expiré $\rightarrow$ va à la fin de la file

Avantages

- Simple à comprendre : chacun son tour
  File d'attente au jeu gonflable
- Borne le temps que peut consommer un processus
- Équitable ?

Questions

- CPU-bound vs IO-bound, qui y gagne ?
- Et si un processus part en boucle infinie ?

Exercice et simulation (tourniquet)

processus	temps d'arrivée	temps de calcul	quantum
p1	0	9	4
p2	1	3	4
p3	2	3	4

Exercice et simulation (tourniquet)

processus	temps d'arrivée	temps de calcul	quantum
p1	0	9	4
p2	1	3	4
p3	2	3	4

processus	temps d'attente	temps total
p1	6	15
p2	3	6
p3	5	8
minimum	3	6
moyenne	4.7	9.7
maximum	6	15

Tourniquet + priorité

- Files d'attente + priorité + quantum de temps
- Quand le quantum est expiré, on va à la fin de sa file d'attente
- Mais on reste dans sa file d'attente

SCHED_RR (Posix)

- `chrt --rr 90 macommande`
- **Question.** Et si `macommande` part en boucle infinie ?
- Pages de man : `sched_rr_get_interval(2)` et `/proc/sys/kernel/sched_rr_timeslice_ms`

Problème des algos précédents

- Des décisions sont prises
- Mais indépendamment des caractéristiques des processus ou de leurs comportements

$\rightarrow$ Ce n'est qu'à postériori qu'on se désole (ou se félicite)

Solutions

- L'utilisateur choisit l'ordonnanceur en fonction de ce qui fonctionne bien pour son usage
- L'ordonnanceur prend en compte les caractéristiques et/ou le comportement

$\rightarrow$ Pourquoi pas les deux ?

Le plus court d'abord

- On choisit le plus court dans la file d'attente
- SJF (*shortest job first*)
- Hypothèse (forte) : on connait (estime) le temps de calcul
- Avantage : temps optimaux si arrivée en même temps

Version préemptive

- Temps **restant** plus court d'abord
- On perd la main si un plus court arrive
- Pas de quantum de temps
- **Question.** Pourquoi pas de quantum?

Exercice et simulation (plus court temps restant)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

Exercice et simulation (plus court temps restant)

processus	temps d'arrivée	temps de calcul
p1	0	9
p2	1	3
p3	2	3

processus	temps d'attente	temps total
p1	6	15
p2	0	3
p3	2	5
minimum	0	3
moyenne	2.7	7.6
maximum	6	15

Problèmes du plus court

Connaître le temps

- Fourni par l'utilisateur
  - Estimation, maximum, catégorie de programme
- Analyse de l'historique
  - Qu'était le comportement du processus

Famine (*starvation*)

- Un gros processus n'a jamais la main
- Si de petits processus qui arrivent continuellement
- **Question** comment éliminer la famine ?

Linux CFS (*completely fair scheduler*)

- Depuis Linux 2.6.23 (2007)
- Objectifs : utilisation CPU et interactivité
- Pas de file d'attente
- Compte le temps réellement consommé
- Le temps d'attente (E-S) pris en compte (améliore l'interactivité)

Quantum non fixe

- On répartit le prochain bloc de temps
- Partage entre tous les processus
- La part de chacun dépend du temps CPU déjà consommé

Politiques de CFS

- `SCHED_OTHER` (appelé aussi `SCHED_NORMAL`) : le défaut
- `SCHED_BATCH` : comme `SCHED_OTHER` mais moins de préemptions
- `SCHED_IDLE` : plus faible que nice 19

Gentillesse (Posix)

Nice value

- Attribut par processus (ou thread)
- de -20 à +19 (sous Linux)
- Voir `nice(1)`, `renice(1)` et `nice(2)`

Principe

- Plus on est gentil plus on laisse sa place
- Privilèges nécessaires pour être pas gentil
- Priorité non stricte : c'est juste du bonus

Sous Linux (CFS)

nice affecte le calcul du « temps consommé » donc change la portion de CPU attribuée

Temps réel : différentes utilisations du terme

En direct

- Ça se passe maintenant
- L'horloge temps réel donne l'heure courante
- `top(1)` affiche en temps réel les processus

Soumis à des **contraintes** temporelles

- Le respect des échéances fait partie du cahier des charges
- Rater des échéances est un problème
- Exemple : un système de vidéo-conférence

Soumis à des contraintes temporelles **strictes** (dur)

- Rater **une** échéance est une **catastrophe**
- Exemple : un système de freins dans une voiture
- Principe de base : le **temps au pire** doit être contrôlé
- Quitte à dégrader les performances moyennes

Ordonnancement et temps réel

Préalablement connues

n processus, avec

- Période d'arrivée P_i
- Échéance
- Durée d'exécution au pire (coût) C_i

$\rightarrow$ du plus grand au plus petit

Garantie

Le système doit rejeter les processus qu'il ne peut pas ordonnancer sans respect des échéances

Exemples

Taux monotone (*rate-monotonic*)

- Nécessite des périodes connues
- Est élu celui qui a la période la plus courte (priorité constante)
- Test d'admissibilité: $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(\sqrt[n]{2} - 1)$

Prochaine échéance d'abord (*earliest deadline first*)

- Plus l'échéance est proche, plus sa priorité augmente
- Test d'admissibilité: $\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$
- Sous Linux: `SCHED_DEADLINE`

Multi-processeur

- La même chose qu'en mono-processeur
- Mais en plus complexe

Problèmes

- Caches CPU et prédicteurs CPU
- Mémoire non uniforme (NUMA, *Non-Uniform Memory Access*)
- Hétérogénéité processeur (HMP, *heteregeneous multiprocessing*)

Quelques solutions

Affinité CPU naturelle

- L'ordonnanceur maintient le processus sur un même processeur
- Problème : déséquilibrage ; solution : rééquilibrer

Affinité CPU explicite

- Laisser l'utilisateur assigner des processeurs
- `taskset(1)`, `sched_setaffinity(2)`

Autres ordonnancements

Entrées-sorties disques

- Décider quelle donnée doit être écrite (ou lue) en premier
- `ionice(1)` et `ioprio_set(2)`

Paquets réseau

- Décider quels paquets sont routés en priorité
- `tc(1)` (*trafic control*)

Gestion de la performance

- Le SE contrôle voltage et niveau de veille des processeurs
- Décider quelle politique adopter (en fonction des processus)

300 Systèmes de gestion de fichiers

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Mémoire de masse

Objectif : stocker des données

- Sur des périphériques
- De manière persistante (non volatile)
- En grande quantité (gros volumes)

Problèmes

- Technologies physiques variées
- Temps d'accès varié aussi (mais plus lent que la RAM)

$\rightarrow$ Responsabilité du système d'exploitation

Disque : abus de langage

Il n'y a pas forcément de **disque physique**

- Disque SSD (*solid-state drive*)
- Espace disque : `df(1)`, `du(1)`

Temps d'accès

- Registres CPU, ko, $\approx$ .25ns
- Cache CPU, Mo, $\approx$ 10ns
- RAM, Go, $\approx$ 100ns
- Disque SSD, To, $\approx$ 25 000ns (25µs)
- Disque magnétique, To, $\approx$ 5 000 000ns (5ms)

Gestion de l'espace disque et des fichiers

Gestion de l'espace disque

- Répondre aux demandes d'allocation de libération de l'espace disque
- Retrouver les fichiers et répertoires
- S'assurer de la fiabilité

$\rightarrow$ le tout, efficacement

Abstraction pour l'utilisateur

- Abstraction de la gestion de l'espace
- Cohérente et indépendante

$\rightarrow$ Fichiers (et répertoires)

Notion de base : le fichier

Système de gestion de fichiers (SGF)

- La partie du SE qui s'occupe des fichiers

Ubiquitaire et requis

- L'utilisateur (ou le logiciel) veut enregistrer des donnés

$\rightarrow$ Il doit utiliser un fichier

Questions

- Y a-t-il des alternatives aux fichiers pour stocker des données ?
- Est-ce que le concept de fichier a tendance à être moins important de nos jours ?

Les fichiers pour l'utilisateur

Besoins de l'utilisateur (et des logiciels)

- Nombreux (plusieurs millions, voire milliards)
- Contenu défini par l'utilisateur
- Fichiers nommés (plutôt que numérotés)
- Organisés pour les retrouver facilement
- Notion de propriétaire et droits d'accès
- Indépendants du matériel

Les fichiers dans INF3173

Nombreux points de vu : utilisateur, programme, bibliothèque, processus, noyau, contrôleur, périphérique, format de système de fichiers, etc.

Niveau **utilisateur**

- Les fichiers que l'humain « voit » et manipule sur le disque
- Inclut aussi le niveau programmeur et processus

Niveau **disque**

- Matériel : ce qui est physiquement stocké (ou simulé)
- Persistant : existe même quand l'ordinateur est éteint
- Inclue aussi tout ce qui est format et type de système de fichiers

Niveau **noyau** du système d'exploitation

- Ce qui est nécessaire à la gestion globale des fichiers

Terme ambigu : fichier

On va *essayer* d'être rigoureux. Termes définis dans la suite...

- **Inode** (ou juste *fichier*) : utilisateur, noyau et disque
  Données réellement sur le disque* (données et métadonnées)
- **Entrée** (ou *dentry*) : utilisateur, noyau et disque
  Un nom de fichier dans un répertoire
- **Chemin** : utilisateur et noyau
  Chaîne de caractères qui désigne un fichier (ou pas)
- **Fichier ouvert** (le nom est pas super) : noyau
  Un fichier* en cours de lecture et/ou écriture (par le noyau)
- **Descripteur de fichier** : utilisateur et noyau
  Numéro (par processus) qui désigne un fichier ouvert du noyau
- **Flux** (*stream*) : utilisateur
  Structure programmative désignant un fichier ouvert*
  (`FILE*`, `fstream`, `InputStream`, etc.)

*Ou un machin proche.

« Tout est fichier »

Philosophie importante Unix

- Pseudo-systèmes de fichiers, comme `proc(5)`
- Périphériques vus comme des fichiers spéciaux (on y reviendra)
- Descripteurs de fichiers pour ce qui peut être lu et écrit :
  - Tubes, sockets, etc. (on y reviendra)
  - Mais aussi pour de l'évènementiel :
    `eventfd(2)`, `signalfd(2)`, `inotify(7)`, etc.

Avantages

- De nombreuses combinaisons : ex. entrée standard
- Réutilisation d'appels système : ex. `read(2)`, `write(2)`
- Réutilisation de politiques : ex. chemins et droits des fichiers

Chemins

Racines

- Unix: la racine s'appelle `/` (slash) et elle est unique
- Windows: plusieurs racines possibles (C:, etc.)

Chemins

- Absolus : commencent par un `/` et partent de la racine
- Relatif : partent du répertoire courant du processus
  Et non du répertoire où est stocké le binaire, etc.

Répertoire courant

- Un par processus
  `pthreads(7)` partagent, `fork(2)` hérite, `execve(2)` préserve
- `chdir(2)` et `getcwd(3)`
- **Question** Pourquoi `cd` est une commande interne du shell ?

Résolution de chemins

- Partir d'une chaine de caractère
- Trouver un fichier
- En étant le plus performant possible

Pas si facile

- Trouver le répertoire de départ (racine, répertoire courant, etc.)
- Se promener (droits, liens symboliques, points de montages, etc.)
- Trouver et valider le dernier élément
- `path_resolution(7)`
- On y reviendra...

Systèmes de fichiers

Organisation

- **Système de fichiers** = ensemble **autonome** de fichiers
- Chaque système de fichiers est **indépendant** et **cohérent**

$\rightarrow$ Mais fait partie d'un grand tout : la hiérarchie des fichiers

Caractéristique d'un système de fichiers

- Le **périphérique**: **emplacement** où sont stockées les données
- Le **type**: **format** de stockage des données

Note: un pseudo système de fichiers comme `proc(5)` n'a pas de périphérique associé

Type de système de fichiers

- Il s'agit du **format** utilisé pour représenter un système de fichiers
- FAT32, NTFS, HFS+, `ext4(5)`, `btrfs(5)`, `xfs(5)`, ZFS...
- Attention : dans la plupart des contextes, « type » est implicite. Ne pas confondre un « système de fichiers » et « type de systèmes de fichiers »
- Chaque système d'exploitation peut supporter différents types `/proc/filesystems` donne une liste sous Linux

Contenu d'un système de fichiers

- Espace de donnée : les données des fichiers
- Espace de gestion : les métadonnées des fichiers, leur organisation et celle de l'espace libre

$\rightarrow$ C'est habituellement persistant

- Les détails dépendent **grandement** du type du système de fichiers

Périphérique

- Les **systèmes de fichiers** résident (habituellement) sur des **périphériques** (*device*)
- Exemple: disques, partitions, etc. (des fichiers de type bloc)
- `lsblk(8)`, `blkid(8)`

Abus de langage

- « Périphérique » est utilisé de façon libérale et ne correspond pas forcément à un dispositif physique distinct
- « Partition » s'utilise parfois à la place de « périphérique » (qui n'est pas forcément une vraie partition),
  Voire désigne le système de fichiers qui y est stocké

Montage et démontage

- **Point de montage**: répertoire où est accroché un système de fichiers
- Pour monter: `mount(8)`, `mount(2)`
- Pour démonter: `umount(8)` et `umount(2)`
- Pour voir l'arborescence: `findmnt(8)`

Question

- Pourquoi c'est des commandes de l'administrateur ?

310 Manipulation des fichiers Unix

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Manipulation des fichiers Unix

Chemins vs. descripteurs

- **Chemin** désigne un fichier par un emplacement
- **Descripteur** désigne un fichier ouvert (on y reviendra...)

Appels système

- `open(2)` (et `creat(2)`) : prennent un **chemin** et donnent un **descripteur**
- `read(2)`, `write(2)`, `close(2)` : manipulent le **descripteur**
- D'autres opérations utilisent un **chemin**
  Supprimer (`unlink(2)`), renommer (`rename(2)`), état (`stat(2)`), exécuter (`execve(2)`), etc.

→ Opérations ± uniformes quelque soit le système de fichiers

→ Les détails internes ne sont pas exposés

Variations d'appels système de fichiers

Appels système « f* »

- Travaille sur un fichier déjà ouvert au lieu d'un chemin
  Exemple `fstat(2)`
- Attention: ne pas confondre avec les fonctions de `stdio(3)`
  Comme `fopen(3)` ou `fread(3)`

Appels système « *at »

- Prend un descripteur + un chemin + des flags
  Exemple `fstatat(2)` ou `openat(2)`
- Variation qui généralise les autres
- Permet de partir d'ailleurs que du répertoire courant
- Évite des **situations de compétition** (*race condition*)
- Les flags permettent du comportement spécial

Entrées-sorties asynchrones

- Modèles de **programmation** spécifiques
- Fonctionnalités offertes par le système d'exploitation

Multiplexage

- Boucle événementielle : un seul point bloquant en général
- `select(2)`, `poll(2)` (et `epoll(7)` sous Linux)

→ Très utilisé

Non-bloquant

- Rien ne bloque
- `O_NONBLOCK` (`open(2)`…) : les accès au fichier sont non bloquants
- `EWOULDBLOCK` ou `EAGAIN` retourné ensuite au lieu de bloquer

→ Besoins très spécifiques

- **Bloquer** c'est la bonne chose par défaut

Table des inodes : renseigne les métadonnées

Une entrée = un fichier

- numéro d'inode (inœud ou numéro d'index)
- type de l'inode (fichier standard, répertoire...)
- propriétaire (uid, gid)
- droits (utilisateur, groupe, autre)
- taille du fichier en octets
- dates (plusieurs sortes)
- nombre de liens durs
- pointeurs vers blocs de données

Question

- Il manque un truc, non ?

Table des inodes

Stockage

- Dans l'espace de gestion d'un système de fichiers

→ Une table par périphérique

- Le détail du contenu et de l'implémentation dépend du type du système de fichiers
- Une copie (partielle) en mémoire du noyau (cache)

Accès

- `ls(1)` (avec options `-il`) et `stat(1)`
- `stat(2)`, `lstat(2)` (et `xstat(2)` sous Linux)
- `inode(7)`

Plus de métadonnées

- Attributs étendus: `xattr(7)`

Types de fichiers Unix

Fichiers réguliers

- Textes, exécutables, code source, images...
- Contenu décidé par l'utilisateur

Fichiers spéciaux

- Répertoires, fichiers physiques (dans `/dev`), liens symboliques, tubes nommés, etc.
- Manipulation par des appels système spécifiques
- Règles au cas par cas

Répertoires

- Fichier spécial `d` (`S_IFDIR`)
- Représente la hiérarchie des fichiers
- `mkdir(2)` (création), `rmdir(2)` (suppression)
- On y reviendra...

Liens symboliques

- Fichier spécial `l` (`S_IFLNK`)
- Représente un autre ficher (via son chemin)
- `symlink(2)` et `ln -s` (création), `readlink(2)` (lecture)
- Documentation `symlink(7)`
- On y reviendra pas...

Questions

- Quelle est la taille d'un lien symbolique ?
- Peut-on savoir si un fichier a des liens symboliques ?
- Un fichier lié doit-il exister ?
- Quels sont les droits pour suivre un lien symbolique ?
- Que faire en cas de cycle de liens symboliques ?

Fichiers périphériques

- Fichiers spéciaux `c` et `b` (`S_IFCHR` et `S_ISBLK`)
- Traditionnellement dans `/dev` (*device*)
- Type caractère (`c`) envoie et/ou reçoit des séquences d'octets
- Type blocs (`b`) écrit et/ou lit dans un bloc d'octets
- Pas de taille : numéro majeur (le type de périphérique) et mineur (un périphérique spécifique)
- `mknod(2)` (création)

Exemples de fichiers périphériques

- `/dev/nvme0n1` : le premier disque dur
- `/dev/tty1` : un terminal
- `/dev/input/mice` : les souris
- `/dev/null` : la poubelle

Autres fichiers intéressants

- `/dev/zero`, `/dev/full`,`/dev/mem`, `/dev/kmem`, `/dev/random`, `/dev/urandom`, `/dev/tty`

Questions

- Pour chacun des fichiers ci-dessus, bloc ou caractères ?
- Qu'est-ce que `/dev/stdout` (déroulez les liens symboliques) ?

Tubes nommés et sockets

- Fichiers spéciaux `f` et `s` (`S_ISFIFO` et `S_ISSOCK`)
- Pour de la communication inter-processus
- Documentation `fifo(7)` et `unix(7)`
- On y reviendra...

Dates (Unix)

Trois types de dates

- mtime : date de dernière modification du fichier
- ctime : date de dernière modification des métadonnées (entrée dans la table des inodes)
- atime : date de dernier accès au fichier (lecture)

Représentation

- Stockées en temps Unix
  Temps écoulé depuis le 1er janvier 1970 UTC
- En secondes ou en nanosecondes
  Ça dépend du type du système de fichiers
- `touch(1)`, `utime(2)`, `utimensat(2)`

Questions

- Il manque pas une date ?
- Que se passe-t-il en 2038 ?

320 Droits et utilisateurs

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Droits et utilisateurs

Utilisateurs (et groupes)

- `uid` : numéro d'utilisateur
- `gid` : numéro de groupe d'utilisateurs

$\rightarrow$ Pour le système vous n'êtes que des numéros

- uid $==$ 0 : super-utilisateur (`root`)

Noms des utilisateurs et groupes (Unix)

Le noyau gère pas les noms des utilisateurs et des groupes

- Fichiers `/etc/passwd` et `/etc/group`
- Fonctions `getpwuid(3)` et `getgrgid(3)`

Utilisateurs et processus

Paires d'identités

- Un utilisateur et un groupe d'utilisateurs = une paire
- Deux paires d'identités (réel et effectif) par processus
- pthreads partagent, fork hérite, exec préserve*
- `setuid(2)`, `setgid(2)`, `seteuid(2)`, `setegid(2)`

Sous Linux

- 4 paires distinctes (réel, effectif, sauvé, fichier†)
- Et des groupes supplémentaires
- `setresuid(2)`, `setfsuid(2)`, `setgroups(2)`, `credentials(7)`

*Sauf si setuid et/ou setgid, on y reviendra...

†Plus vraiment utilisé

Propriétaires des fichiers

Traditionnel Unix

- Chaque fichier du système possède
  - un numéro d'utilisateur propriétaire
  - un numéro de groupe propriétaire
- `chown(1)`, `chgrp(1)`, `chown(2)`
- Lors de la création d'un fichier :
  propriétaires = utilisateurs et groupes effectifs[‡]

Question

Pourquoi ça peut être un problème de stocker seulement les numéros d'utilisateur et groupes dans le système de fichiers ?

[‡] Sauf si setgid dans le répertoire, on y reviendra...

Rappel : droits traditionnels Unix

3 catégories d'accès (ugo)

- u (*user*/utilisateur) l'utilisateur propriétaire
- g (*group*/groupe) le groupe propriétaire
- o (*other*/autre) les autres

3 permissions par catégorie (rwx)

- `r` (*read*) : lire le contenu
- `w` (*write*) : modifier le contenu
- `x` (*execute*) : exécuter (si fichier) ou traverser (si répertoire)
- `chmod(1)`, `chmod(2)`

Questions

- Quels sont les droits nécessaires pour `stat(2)`? `chmod(2)`? suivre un lien symbolique? supprimer un fichier?
- Quand sont vérifiés les droits?

setuid (et setgid)

Bits supplémentaires du mode du fichier

- setuid: 4000, `u+s`
- setgid: 2000, `g+s`

Pour les fichiers exécutables

- Lors du `execve(2)`, l'utilisateur (et/ou groupe) effectif est changé pour celui du fichier
- RTFM pour les détails

→ **Question** Comment (et par qui) est contrôlée cette augmentation de privilèges ?

setgid pour les répertoires

- Sous Linux : un nouveau fichier héritera du groupe du répertoire (au lieu d'être le groupe effectif du processus)

Plus de droits sur les fichiers

Masque utilisateur

- Un par processus (threads partagent, fork hérite, exec préserve)
- Modifiée par `umask(2)`
- L'umask est retiré des droits des fichiers créés (`creat(2)`, etc.)

```
int creat(const char *pathname, mode_t mode);
```

```
droits_du_fichier = mode & ~umask
```

- Ne s'appliquent pas à `chmod(2)`, ni s'il y a des ACL par défaut

Encore plus

- ACL (*access control lists*): `acl(5)` → contrôle fin des droits
- MAC (*mandatory access control*)
  Exemples: `selinux(8)` et `apparmor(7)`

330 Répertoires

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Répertoires

Associent noms de fichiers et inodes

- Rappel: le nom des fichiers n'est pas dans la table des inodes
- « données » d'un répertoire = liste d'entrées
- Chaque **entrée** associe un nom de fichier à un numéro d'inode
- Certains SF y dupliquent de l'information (type du fichier, etc.)

Exemple

inode 253 (répertoire) :

```
253 .
146 ..
540 ficelle
490 repondeur
```

inode 490 (répertoire) :

```
490 .
253 ..
679 fictif
831 fichtre
```

API POSIX (portable)

- Structure opaque `DIR *`
- `opendir(3)`, `readdir(3)`, `closedir(3)`

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char **argv) {
  DIR *d = opendir(argv[1]);
  if (!d) { perror(argv[1]); exit(1); }
  struct dirent *de;
  while((de = readdir(d)))
    printf("%10li %s\n", de->d_ino, de->d_name);
  closedir(d);
  return 0;
}
```

Accès en vrai

- Les répertoires sont des fichiers spéciaux
- Contenu pas directement accessible à l'utilisateur

Question

- Peut-on utiliser `open(2)` et `read(2)` pour lire les répertoires ?
- Comment fonctionnent `opendir(3)` et `readdir(3)` en vrai ?

Chemin $\rightarrow$ inode en théorie

1. Découper le chemin en éléments $e_1/e_2/.../e_n$
2. Partir du répertoire de base i_k avec $k = 0$ (racine, courant, etc.)
3. Charger le contenu du répertoire i_k depuis l'espace de donnée du disque
4. Chercher dedans l'élément suivant e_{k+1}
5. Charger l'inode associé i_{k+1} depuis la table des inodes
6. Vérifier que i_{k+1} est bien un répertoire, les droits, etc.
7. Si besoin, continuer en 3 avec $k = k + 1$

- Beaucoup d'accès nécessaires au disque (au moins $2n$)
- À faire : droits, liens symboliques, points de montage
- Attention à la concurrence
  - Un processus résout un chemin
  - Pendant qu'un autre modifie les répertoires

Chemin → inode en pratique : cache

- `dentry` (*directory entry*) et `dcache` (*dentry cache*)

Représentation **globale** interne au SE

- Vue (partielle) en mémoire de la hiérarchie globale
- Associe une entrée à son inode et son système de fichiers
- Mise en cache au fur et à mesure
- Libération si la mémoire est demandée pour autre chose

Sert de cache

- Pas besoin de relire les répertoires sur le disque à chaque fois
- Sauf dans certains cas (ex. disques réseau)
  → validation et synchronisation

Efficace

- Accès rapide aux entrées : table de hachage
- Échec rapide : stocke entrées inexistantes (*negative dentry*)

Liens durs (*hard link*)

Définition

Des entrées de répertoires

- Avec un ou **plusieurs** noms
- Dans un ou **plusieurs** répertoires
- Qui référencent un **même** inode

$\rightarrow$ Le champ **nombre de liens durs** compte le nombre de références

Piège

- Les liens durs ne sont pas des liens « fichier $\rightarrow$ fichier »
- Mais des liens « entrée $\rightarrow$ inode »
- Appelé aussi « lien direct », « lien physique » ou juste « lien »

Manipulation des liens durs

Création de liens durs

- `ln(1)` et `link(2)`
- Pas de distinction entre l'original et le lien

$\rightarrow$ Les deux entrées désignent le **même** fichier (inode)

Suppression

- `rm(1)` et `unlink(2)`
- Décrémente le nombre de liens durs
- Si 0, le fichier (inode) est réellement supprimé
- Note: `creat(2)` et `unlink(2)` ne sont pas symétriques

Renommage et déplacement

- `mv(1)` et `rename(2)`
- Le nombre de liens durs reste inchangé
- Attention, seulement sur le même système de fichiers

Limites de liens durs

- Forcément sur le même système de fichiers
- Pas de liens durs entre répertoires
- Pas forcément l'effet voulu lors de l'écrasement de fichiers (perte d'identité)

Questions

- Comment la commande `mv(1)` sait déplacer entre systèmes de fichiers (alors que `rename(2)` ne sais pas faire) ?
- Pourquoi il existe une commande `cp(1)` mais pas d'appel système de copie ?
- Comment supprimer tous les liens durs d'un fichier ?
- Si on pouvait utiliser `link(2)` sur les répertoire, comment créer des répertoires détachés de la racine ?

340 Traitement des fichiers ouverts

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Descripteurs de fichiers

Descripteur de fichier

- Dans un processus
- Désigne un fichier ouvert
- Sert à la manipulation
- C'est un entier tout simple (`int`)

Trois descripteurs standard

- 0: entrée standard
- 1: sortie standard
- 2: sortie standard pour les messages d'erreur

$\rightarrow$ C'est des conventions de l'espace utilisateur : le noyau s'en fiche

Utilisation des descripteurs

- Ouverture: `creat(2)`, `open(2)`, etc.
- Manipulation: `read(2)`, `write(2)`, etc.

```
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main(void) {
  char msg[] = "Hello, World!\n";
  int fd = creat("hello", 0666);
  write(fd, msg, strlen(msg));
  close(fd);
}
```

Questions

- Pourquoi `strlen` et pas `sizeof` ?
- Quels sont les cas d'erreurs possibles pour tous ces appels système ?

Organisation interne : 3 niveaux

TD. Tables des descripteurs (une par processus)

- Une entrée par descripteur
- Pointe sur un fichier ouvert ($\rightarrow$TFO*)

TFO. Table de fichiers ouverts (globale)

- Une entrée par demande d'ouverture d'un fichier
- Chaque `open` ou `create` ou autre
- Pointe sur un inode en mémoire ($\rightarrow$TIM*)

TIM. Table des inodes en mémoire (globale)

- Une entrée par fichier (inode) distinct manipulé (ou en cache)
- Synchronisée avec les inodes sur disque

*Qui contient un compteur.

Info sur les fichiers ouverts?

- `/proc/sys/fs/inode-nr` nombre d'inodes en mémoire
- `/proc/sys/fs/file-nr` nombre d'inodes ouverts distincts

Commandes

- `lsof(1)` et `fuser(1)` permet de « voir » ou « chercher » les fichier ouverts
- Cherchent/voient aussi les communications réseau et les tubes

/proc/PID

- `/proc/PID/fd` le fichier (ou autre) associé au descripteur
- `/proc/PID/fdinfo` des informations sur le fichier ouvert

TIM: Caches des fichiers

- Table des inodes en mémoire $\rightarrow$ cache par fichier

Le SE minimise les accès disque : asynchronisme

- En lecture (*readahead*) et en écriture (*flush*)
- Demandes des utilisateurs $\neq$
  Lectures et écritures effectives sur disque
- `sync(1)`, `sync(2)`, `fsync(2)`: forcer les écritures
- `/proc/sys/vm/drop_caches`: libérer l'espace des caches

Cohérence entre accès concurrents

- Processus peuvent lire et écrire sur le même fichier
- Chacun a la même vision du contenu (le cache noyau)

Attention : ne pas confondre

- Caches noyau
- Caches applicatifs (programmes et bibliothèques)

Table des fichiers ouverts

- Une entrée par `open(2)` (ou autre) effectué
- Contient le mode d'ouverture (lecture, écriture, etc.)
- Contient le curseur lecture-écriture (éventuel) dans le fichier

→ Un même fichier peut être manipulé indépendamment par des processus

- `lseek(2)` permet de déplacer le curseur lecture-écriture

hello_wr

```
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main(void) {
  char msgout[] = "Hello, World!\n", msgin[50];
  int fdout = creat("hello", 0666);
  int fdin = open("hello", O_RDONLY);
  write(fdout, msgout, strlen(msgout));
  ssize_t len = read(fdin, msgin, sizeof(msgout));
  write(1, msgin, len);
  close(fdout); close(fdin); return 0;
}
```

- Un seul fichier `hello`
- Deux ouvertures (distinctes)
- Deux descripteurs

Questions

```
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main(void) {
  char msgout[] = "Hello, World!\n", msgin[50];
  int fdout = creat("hello", 0666);
  int fdin = open("hello", O_RDONLY);
  write(fdout, msgout, strlen(msgout));
  ssize_t len = read(fdin, msgin, sizeof(msgout));
  write(1, msgin, len);
  close(fdout); close(fdin); return 0;
}
```

- Et si on inverse le `read` et le `write`?
- Comment s'assurer de la cohérence d'un fichier si plusieurs processus peuvent écrire en même temps ?
- Pourrait-on faire communiquer des processus via un fichier commun ouvert ?

Threads, fork, exec

Pthreads partagent

- Les descripteurs sont associés au processus

$\rightarrow$ Donc partagés par les threads

Fork duplique (et partage)

- La table des descripteurs est **dupliquée**
- Les entrées dans la table des fichiers ouverts sont **partagées**
- En particulier le curseur de position (lecture/écriture)
- Les compteurs de la table des fichiers ouverts sont incrémentés
- **Question** pourquoi ne pas incrémenter les compteurs de la TIM?

Exec préserve

- La table des descripteurs et **préservée**

$\rightarrow$ Permet de préserver 0, 1 ou 2

hello_fork.c

```c
#include<unistd.h>
#include<fcntl.h>
#include<wait.h>
int main(void) {
  char buf[50];   size_t len = 0;
  int fd = open("bonjour.txt", O_RDONLY);
  pid_t p = fork();
  if (p==0) {
    len += read(fd, buf, 5);
    sleep(2);
    len += read(fd, buf+len, 5);
  } else {
    sleep(1);
    len += read(fd, buf, 5);
    wait(NULL);
  }
  write(1, buf, len); return 0;
}
```

```
$ cat bonjour.txt
Bonjonde!
ur mouldiou!
```

Spécificités Linux

Flag O_CLOEXEC

- `O_CLOEXEC` flag de `open(2)` (et autres appels système)
- Le descripteur sera automatiquement fermé lors d'un `execve(2)`

→ Évite la fuite de descripteurs ou gaspillage de ressources

→ , Mais pas portable

Tâches Linux

`clone(2)` permet de décider quoi partager ou cloner

- `CLONE_FILES` la table des descripteurs
- `CLONE_FS` des informations liées au système de fichiers, dont `chdir` et `umask`

Duplication de descripteurs

Descripteurs synonymes

- Deux descripteurs d'un même processus peuvent pointer une même entrée dans la table des fichiers ouverts
- Appels système `dup2(2)` (et `dup(2)`)

Quel est l'intérêt ?

- Redéfinir les entrées et sorties standard
- Redirection de fichiers
- Communication par tube (pour plus tard)

Question

- Quelle est la différence entre dupliquer un descripteur et ouvrir deux fois un fichier ?

Redirection de la sortie standard

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

int main(void) {
  int fd = creat("sortie", 0666);
  dup2(fd, 1);
  printf("Hello World!\n");
  return 0;
}
```

Redirection de l'entrée standard

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

int main(void) {
  int fd = open("hello", O_RDONLY);
  dup2(fd, 0);
  close(fd);
  execlp("lolcat", "lolcal", NULL);
  perror("lolcal");
  return 1;
}
```

Autre partage de descripteurs ou fichiers

- Un descripteur est un entier, partager `4` n'avance à rien

$\rightarrow$ le noyau doit être impliqué

Via sockets Unix

- `unix(7)`
- Partage des **fichiers ouverts**
- Via messages auxiliaires (`SCM_RIGHTS`)

Via /proc (Linux)

- `/proc/PID/fd` : `proc(5)`
- Partage des **inodes en mémoire**
- Même des fichiers supprimés
- Même de communication interprocessus (tubes, sockets, etc.)

350 Implémentation des systèmes de fichiers

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Types de systèmes de fichiers

- Nombreux types existent
- Wikipédia en liste et compare une centaine

Nombreux, car spécifiques

- À des systèmes et/ou organisations
  Contrôle de l'évolution, mais aussi syndrome NIH
- À des contraintes physiques des périphériques et des ordinateurs
- À des besoins spécifiques des utilisateurs

Types de systèmes de fichiers

Format de stockage

Spécifie la représentation des données sur disque

- Champs de bits
- Structures de données

Implémentation

- Dans les systèmes d'exploitation
- Dans les outils annexes (`mkfs(8)`, `fsck(8)`, etc.)

$\rightarrow$ C'est compliqué, surtout si le format n'est pas documenté

$\rightarrow$ Besoin maximal de fiabilité : ne pas manger les données !

Découpage en blocs

- Blocs de taille fixe, configurable, ou variable (ça dépend)
- Découpe tout l'espace disque
- Simplifie la gestion : blocs au lieu d'octets
- Tout est des blocs ensuite : données ou gestion

Limites principales

- Nombre maximal d'inodes
- Taille maximale d'un fichier
- Nombre d'entrées maximal par répertoire
- Taille maximale du volume
- Etc.

Exemples de limites

Type	Taille fichier	Taille volume
FAT32	4 Go	16Tb
NTFS	16 Eo*	16 Eo
ext4	16 To	1 Eo
btrfs	16 Eo	16 Eo
ZFS	16 Eo	256 Zo†

Ce sont des limites du format : la plupart des implémentations (systèmes et outils) ne les atteignent pas forcément.

*16 Eo $= 2^{64}$ octets $=$ 18 446 744 073 709 552 000 octets (20 chiffres)

†256 Zo $= 2^{78}$ octets $\approx 3 * 10^{23}$ (24 chiffres)

Détermination des limites

- Contraintes internes au type de système
  Tailles en octets de valeurs numériques
- Paramètres configurés par l'utilisateur
  Lors du formatage: `mke2fs(8)`, `mkfs(8)`, etc.
- Limites d'implémentations
  Le format peut stoker plus, mais les logiciels ne peuvent pas lire
- Combinaisons directes et indirectes de tout ça

Besoins et fonctionnalités

De base

- Stocker les données des fichiers (gros et petit)
- Stocker les métadonnées (y compris étendues `xattr(7)`)
- Stocker les entrées des répertoires
- Gérer l'espace libre (inodes et blocs)

Plus avancés

- Chiffrement et compression
- Journalisation (on y reviendra...)
- Instantanés (*snapshots*) et branches
- Déduplication
- Multi-volumes, RAID, etc.
- Somme de contrôle (*checksum*)
- Correction d'erreurs

Allocation et adressage des fichiers

Allocation contiguë

- Les blocs de données d'un fichier sont contiguës
- Exemple : ISO 9660 (CDs)
- Naïf : en général, la taille des fichiers est inconnue et évolue

Allocation chainée

- Un bloc de données connait l'adresse du suivant
- Exemple : FAT
- Problème : accès direct lent (`lseek(2)`)

Allocation indexée

- Un fichier connait la liste de ses blocs de données
- Problème : comment stocker des gros fichiers ?

Allocation indexée Unix

Pointeurs vers les blocs de données

- Pointeur direct : contient l'adresse d'un bloc de données
- Pointeur indirect : contient l'adresse d'un bloc contenant des pointeurs directs
- Pointeur indirect double : contient l'adresse d'un bloc contenant des pointeurs indirects
- Etc.

Question

- Comment déterminer où s'arrêtent les données ?

Adressage des fichiers - Exercice

Exemple : adressage indexé de ext2/3 (détails)

Dans la table des inodes

- Il y a 15 pointeurs de blocs :
  - 12 sont des pointeurs directs
  - 1 est indirect
  - 1 est indirect double
  - 1 est indirect triple
- Un bloc fait 4ko (défaut typique, mais configurable)
- Un pointeur de bloc est représenté sur 32 bits (4o)

Questions : quelles sont les tailles maximales

- D'un fichier si tous ses blocs sont pleins ?
- D'un fichier si la taille est codée sur 32 bits ?
- D'un volume si tous les blocs sont utilisés ?

Allocations modernes

Extents

- *extent* = suite de blocs contigus
- On stocke 2 nombres plutôt que tous les blocs de la suite
- ext4, btrfs, ntfs, etc.
- Problème: cf. allocation contiguë

Arbres B (*B-tree*)

- Structure de données arborescente équilibrée (voir INF3105)
- Adaptée aux systèmes de fichiers et base de données
- btrfs, zfs, ntfs, etc.
- Problèmes: nombreux détails algorithmiques

Journalisation

Problème : corruption

- Panne lors d'une écriture

$\rightarrow$ Données partiellement/mal écrites

$\rightarrow$ Incohérences données et métadonnées

Solution : écrire en deux temps

- On écrit les données dans un **journal**
- Quand le journal est écrit, on **recopie** dans le disque

$\rightarrow$ Problème de coût : écriture **plus chère** (copie intermédiaire)

Principe de la journalisation

Si panne pendant l'écriture dans journal

- On jette les données du journal (tant pis!)
- Les données du disque sont vieilles, mais **cohérentes**

Si panne pendant l'écriture du disque

- Le journal est complet et **cohérent**
- On termine l'écriture depuis le journal

Détails de la journalisation

- Nombreux détails spécifiques
- Configurations possibles

Question

- Journaliser seulement les métadonnées est-il un bon compromis ?

Copie sur écriture (*copy-on-write*)

Principe

- Au lieu de **modifier** quelque chose
- On en fait une **copie** modifiée
- On utilise la copie au lieu de l'original
- On libère l'original

$\rightarrow$ Plus efficace que la journalisation classique

Généralisation de l'approche

- Utilisé par ZFS et btrfs
- Instantanés : on peut garder d'anciennes versions
- Branches : des versions peuvent évoluer indépendamment

400 Communication inter-processus

INF3173

Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Introduction

Chaque processus

- Est autonome
- Vit isolé dans son propre espace mémoire

Trop restrictif

- Besoin de collaboration, communication et de coopération
- Processus différents opèrent ensemble vers un même objectif

Communication interprocessus

- IPC (*interprocess communication*)
- Mécanismes du système d'exploitation
- Parfois offerts par bibliothèques et démons (espace utilisateur)

Formes de coopération

Coopération interne

- Application conçue à la base multithreads et multiprocessus
- Exemples: navigateurs modernes

$\rightarrow$ Objectifs : performance, asynchronisme, isolation, etc.

Coopération protocolaire

- Communiquer avec des applications qui **respectent** un **protocole**
- Exemples: la plupart des applications réseaux

$\rightarrow$ Attention à être robuste

Coopération des données

- Application manipule des **données**, produite (ou non) par d'autres applications
- Exemples : ouvrir/enregistrer, tubes shell, etc.

Philosophie Unix

1978

- Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
- Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
- Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Philosophie Unix

1994

- 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.

Source Raymond, E. (2003). The Art of Unix Programming.

Mise en garde

Vaste domaine

- La communication entre applications informatiques est un domaine bien large qui sort du cadre de ce cours

Voir

- Téléinformatique (INF3271)
- Programmation concurrente et parallèle (INF5171)
- Programmation Web avancée (INF5190)
- Réseaux sans fil et applications mobiles (TEL4165)
- Programmation parallèle haute performance (INF7235)
- Machines virtuelles (INF7741)
- Systèmes répartis (MGL7126)

$\rightarrow$ seulement une petite partie de l'iceberg

Deux modèles de communication

Données échangées entre processus (*message passing*)

- Chacun les traite à sa façon
- Un seul processus a les données à la fois

Données partagées entre processus (*shared memory*)

- Données communes
- Accès et modifications non exclusives

$\rightarrow$ On n'y reviendra plus tard

Questions: à quel modèle correspond

- Tube shell « `|` »
- 2 threads et une structure de données dans le tas
- Fichiers dans `~/Documents`
- Base de données

Exemples d'IPC chez POSIX

- Fichiers (déjà vu)
- Terminaison de processus: `exit(2)` et `wait(2)` (déjà vu)
- Signaux: `signal(7)` ; y compris `kill(1)` (410 Signaux)
- Tubes (*pipe*): `pipe(7)` ; y compris « | » et tubes nommés `fifo(7)`. (420 Tubes)
- Sockets UNIX: `unix(7)`. (430 Sockets Unix)
- Sockets classiques (TCP, UDP, etc.): `socket(7)`, `tcp(7)`, `udp(7)` (on y reviendra pas)
- Mémoire partagée: `shm_overview(7)` (on y reviendra)

→ Plus quelques autres

→ Plus tous ceux spécifiques à d'autres systèmes

→ Plus tous ceux "applicatifs" basés sur ces primitives systèmes RPC (*remote procedure call*), bus logiciels, etc.

Système de fichiers

Les fichiers ne sont pas exclusifs à un processus

- Les processus peuvent utiliser les fichiers pour communiquer entre eux
- L'accès et la protection sont connus

Exemples

- Fichiers de données et autres documents (explicites à l'utilisateur)
- Fichier temporaire pour passer des données (compilation, etc.)
- Fichiers spéciaux pour initier un autre type de communication (tubes nommés, etc.)
- *Spool* (impression, cron job, etc.)
- Fichier projeté en mémoire (`mmap(2)`), on y reviendra.

Socket réseau

Communication distante

- Objectif primaire : faire communiquer des applications sur des ordinateurs distincts
- Peut **aussi** être utilisé pour des applications sur une même machine
- Majoritairement pour du clients-serveur

Avantages

- Le SE peut optimiser l'efficacité du traitement
- Exemple : client/serveur X (`X(7)`)

Bus logiciels

Support de communication de haut niveau

- Majoritairement applicatif (en espace utilisateur)
- Permet de faire communiquer des applications via des objets partagés et des envois de messages
- Souvent pour le réseau (ex. CORBA)

Exemple : D-BUS

- Utilisé dans les bureaux graphiques Unix modernes
- Un démon + clients exposent&utilisent services&objets
- Bibliothèques utilisables par les programmes
- Démon et bibliothèque utilisent des primitives systèmes pour faire le travail de communication, de synchronisation et de protection
- https://www.freedesktop.org/wiki/Software/dbus/

Gestion de la communication

Il faut un **protocole** de communication

- Moyens de communication (données, structures de données) que les processus peuvent échanger et accéder
- Primitives d'accès et de protection
- Mécanismes de synchronisation
- Attention aux problèmes standards
  interblocage, famine, etc.

Qui s'en charge ? Le système d'exploitation

- Fournit des appels système pour IPC
- Plus ou moins riches et complexes
- Règles spécifiques au cas par cas

$\rightarrow$ Impose un protocole mais garantit certaines propriétés

Qui s'en charge ? Le programmeur

- Utilise les primitives systèmes pour implémenter ses propres protocoles et applications
- Prend en compte les limites et caractéristiques des IPC utilisées
- Est libre d'implémenter tout ce qu'il veut par-dessus

$\rightarrow$ les IPC systèmes sont des **outils** pour bâtir sa solution de coopération

Qui s'en charge ? Bibliothèques et langages

- Abstraient les IPC système
- Offrent des fonctionnalités et protocoles clé en main
- Exemple: requête https en JavaScript, JEE, D-Bus, etc.

$\rightarrow$ Il est normal de les utiliser quand c'est adapté

Sauf en INF3173 car pour apprendre, on fait tout à la main

410 Signaux

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Signaux

Forme d'interruption logicielle

- Analogie avec les interruptions matérielles
- Permet d'expédier à un processus une information urgente

Comportement asynchrone

- Un signal est envoyé
- Il sera reçu et traité au moment opportun

Sémantique des signaux

Liste des signaux

- Les signaux sont catalogués
- La liste est fixée
- Chacun est documenté

→ `signal(7)`

Un gestionnaire de signaux par processus

- Chaque programme gère toutefois les signaux comme il veut
- La sémantique doit être documentée dans le programme (man)
- En particulier si elle diverge du catalogue

Exemples de signaux

SIGINT

- Ctrl C génère ce signal dont le comportement par défaut est d'arrêter le processus

SIGSEV

- Une erreur de segmentation provoque l'expédition de ce signal au processus fautif

kill

- Commande `kill(1)`. Envoie signal SIGTERM par défaut.
- Appel système `kill(2)`
- **Question** `kill` est souvent une commande interne du shell, pourquoi ?

Actions possibles pour un signal

Pour chaque catégorie de signal, un processus peut

- Accepter le comportement par défaut
  En général, arrêt du processus
- Ignorer le signal (pas tous)
- Gérer le signal (pas tous)

Gestion des permissions

- Seuls les processus d'un même utilisateur peuvent s'envoyer des signaux
  Et root (RTFM pour les détails)
- Pas de kill sur le processus du voisin

Actions possibles pour un signal

Quelques signaux

Signal	Valeur	Action	Description
SIGHUP	1	T	Le terminal se ferme
SIGINT	2	T	Ctrl C au clavier
SIGKILL	9	TD	terminer le processus
SIGSEGV	11	M	erreur de segmentation
SIGCHLD	-	I	terminaison d'un enfant

- Action par défaut : T=terminer, D=défaut obligatoire, M=image mémoire, I=ignorer

Gestion classique des signaux en deux étapes

Écrire la fonction gérante (en C classique)

- Signature simple `void foo(int sig)` (pour `sa_handler`)
- Ou complète `void bar(int sig, siginfo_t* info, void* uctx)` (pour `sa_sigaction`)

Associer fonction et signal

- `sigaction(2)`
- Structure `struct sigaction` un peu pénible
  `sigemptyset(3)` et cie.

Extra

- `pause(2)` suspend l'exécution jusqu'à un signal
- `strsignal(3)` et `psignal(3)` pour le texte des signaux

sigaction.c

```c
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<string.h>
void gere(int sig) {
  printf("Reçu %d: %s\n", sig, strsignal(sig));
  exit(1);
}
int main(void) {
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  action.sa_handler = gere;
  sigaction(SIGINT, &action, NULL);
  sigaction(SIGTERM, &action, NULL);
  pause();
}
```

Informations supplémentaires

- Pour ignorer un signal, mettre `SIG_IGN` dans `sa_handler`
- Pour l'action par défaut, mettre `SIG_DFL` dans `sa_handler`
- Les signaux d'une même catégorie ne sont pas empilés
  Une rafale d'un même signal peut activer une seule invocation de la fonction gérante

Bloquer (masquer) les signaux

- Retarde la gestion de signaux jusqu'au déblocage
- `sigprocmask(2)` pour manipuler le masque de signaux bloqués
- `sa_mask` de `sigaction(2)` permet de masquer des signaux automatiquement pendant l'exécution de la gérante

Voir les signaux

- `/proc/PID/status` montre l'état des signaux « `Sig*` »
- `sigpending(2)` voir les signaux en attente

Threads, fork et exec

pthreads

- Partagent: les gérantes, les signaux ignorés
- Copie: les signaux bloqués (masque des signaux)
- Fonctionnalités fines existent `pthread_kill(3)`, `pthread_sigmask(3)`
- Certains signaux en attente peuvent être partagés ou pas

fork

- Hérite: les gérantes, les signaux ignorés et bloqués
- Vide: les signaux en attente

execve

- Préserve: les signaux ignorés, bloqués et en attente
- Vide: les gérantes

Interruption des appels systèmes

- Un signal reçu et géré peut interrompre certains appels système
- Appels système dits « interruptibles »
- Détail dans le man de chacun des appels système (ou `signal(7)`)
- Processus dans l'état POSIX « S » selon ps

Qu'est-ce qui se passe alors

1. Processus dans appel système
2. Signal attrapé ; gérante invoquée ; gérante terminée (`return`)
3. Appel système terminé de force (interrompu)
   - Retourne `EINTR` (si pas commencé)
     - Ou autre valeur si travail partiellement réalisé
   - Sauf si `SA_RESTART` dans `sa_flag` de `sigaction(2)`
     - Mais pas pour tous les appels système
   - RTFM pour les détails

Synchronisme

L'approche asynchrone de `sigaction(2)` a des défauts

POSIX

- `sigwaitinfo(2)`, `sigtimedwait(2)`, `sigsuspend(2)`, `sigwait(3)`
- Attend des signaux
- Note: Bloquer les signaux avant avec `sigprocmask(2)` ou autre

Linux

- `signalfd(2)`
- Crée un descripteur de fichier spécial
- Permet de gérer les signaux comme des évènements («tout est fichier»)

$\rightarrow$ Attendre un signal avec `poll(2)`, `select(2)`, etc.

420 Tubes

INF3173
Principes des systèmes d'exploitation

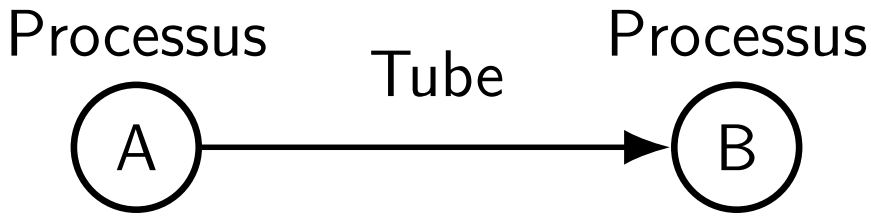
Jean Privat

Université du Québec à Montréal

Hiver 2021

Tubes

- Canal de communication **unidirectionnel** avec deux bouts
- Les octets écrits au bout en écriture (`write(2)`)
- Sont lisibles **dans l'ordre** au bout en lecture (`read(2)`)
- Flot d'octets (*stream*) : pas de concept de messages
- Les octets lus sont consommés
- `pipe(7)` pour les détails



Tube et processus

Descripteurs de fichiers

- Pour un processus, un bout de tube est un descripteur
- Chaque extrémité se manipule comme fichier ouvert
  `read(2)`, `write(2)`, `close(2)`, `dup2(2)`, `poll(2)`, etc.
- Mais pas `lseek(2)` (erreur `ESPIPE`)

Niveau noyau

- Espace mémoire du système d'exploitation
- L'espace et son accès sont gérés par le SE
- Capacité limitée (64ko défaut actuel sous Linux)
- Mais c'est pas un problème (on y reviendra)
- Libéré automatiquement quand plus utilisé

Deux sortes de tubes

Tubes simples (majoritairement utilisés)

- Création : appel système `pipe(2)`
- « Retourne » deux descripteurs de fichiers
- `int fds[2]; pipe(fds);`
- `fds[0]` le bout en lecture
- `fds[1]` le bout en écriture
- Astuce mnémotechnique: 0=stdin 1=stdout

Tubes nommés

- Création : `mkfifo(1)` et `mkfifo(3)`
- On y reviendra...

Exemple tube simple

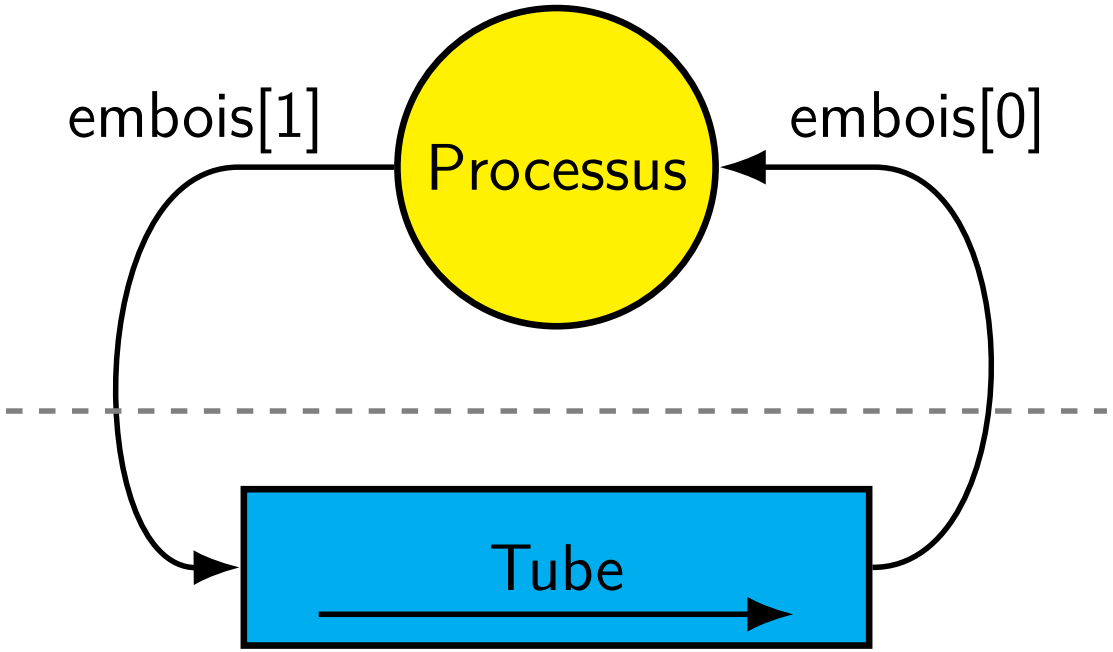
```c
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
int main(void) {
  char *msg = "Bonjour, le monde!", buf[32];

  int embois[2];
  pipe(embois);

  write(embois[1], msg, strlen(msg)+1);

  read(embois[0], buf, sizeof(buf));
  printf("lu: « %s »\n", buf);
  return 0;
}
```

Tubes simples



Communication par tube

- Un tube est créé par un processus
- Mais est global au système

Partage de tube par fork

- Les descripteurs de fichiers sont copiés
- Les bouts de tubes sont partagés

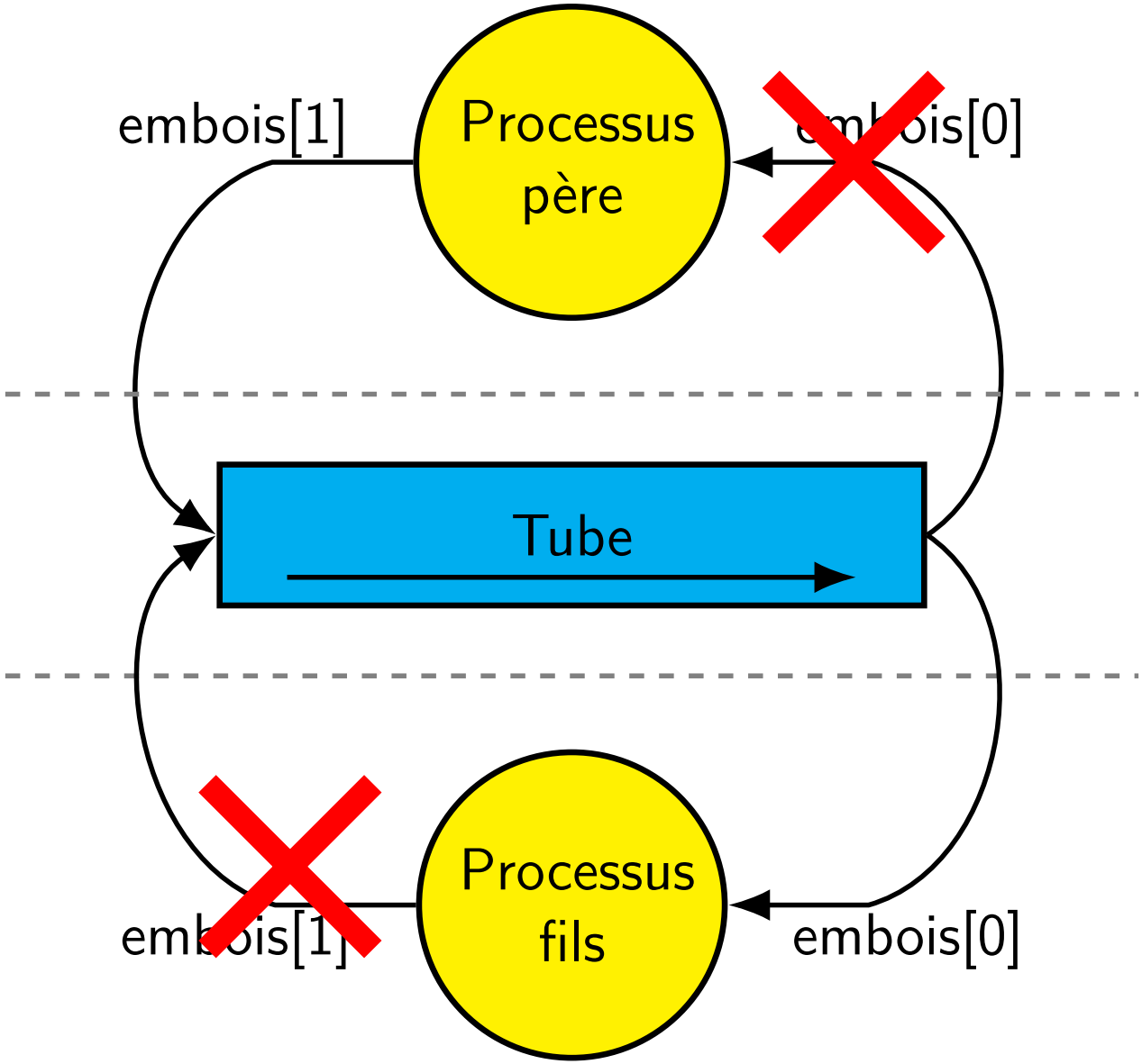
Communications

- Entre parent et enfant
  - Le parent crée le tube
  - L'enfant hérite les descripteurs
- Entre deux enfants
  - Le parent crée le tube
  - Les enfants héritent les descripteurs

pipe-fork.c

```c
#include "machins.h"
int main(void) {
  int embois[2];
  pipe(embois);
  pid_t pid = fork();
  if (pid==0) { // enfant
    char buf[32];
    close(embois[1]);
    read(embois[0], buf, sizeof(buf));
    printf("lu: « %s »\n", buf);
  } else { // parent
    char *msg = "Bonjour, le monde!";
    close(embois[0]);
    write(embois[1], msg, strlen(msg)+1);
    waitpid(pid, NULL, 0);
  }
  return 0;
}
```

Communication par tube



Synchronisation

Lecture

- S'il y a des données dans le tube
  - `read` **lit** le **maximum** d'octets
- Si le tube est vide
  - Si un écrivain existe : `read` **bloque**
  - Si aucun écrivain : `read` **retourne 0** (fin de tube)

Écriture

- S'il y a aucun lecteur
  - Signal `SIGPIPE` **envoyé** (par défaut, **termine** le processus)
- S'il y a un lecteur (ou plus)
  - Si assez de place: `write` **écrit tous** les octets
  - Si le tube est plein (ou presque): `write` **bloque**

Contrôle de flux

Lecteur qui va trop vite

- Bloqué jusqu'à ce qu'un écrivain écrive
- Ou plus de données ni d'écrivain (`read` retourne 0)

Écrivain qui va trop vite

- Bloqué jusqu'à ce qu'un lecteur consomme
- Ou que plus de lecteurs (`SIGPIPE`)

Questions

- Pourquoi c'est pas symétrique ? (0 vs. `SIGPIPE`)
- Comment gérer `SIGPIPE` ?

Extra

Opérations atomiques

- `PIPE_BUF` (512 minimum, 4096 chez Linux)
- `write` écrits `PIPE_BUF` octets (ou moins) atomiquement
- Atomiquement = écrit d'un coup sans que d'autres écritures concurrentes s'entrelacent

Entrées-sorties non bloquantes

- Flag `O_NONBLOCK` possible (via `fnctl(2)`)
- Les règles de synchronisation et d'atomicité changent
- RTFM

Danger : interblocage

Comment se bloquer tout seul ?

Danger : interblocage

Comment se bloquer tout seul ?

```
#include<unistd.h>
int main(void) {
  int embois[2];
  char buf;
  pipe(embois);
  read(embois[0], &buf, 1);
}
```

Question

- Comment se bloquer à deux ?

Bonnes pratiques

Un seul lecteur et un seul écrivain

- « Toujours par deux ils vont, ni plus, ni moins » — Yoda

Fermer les bouts inutiles

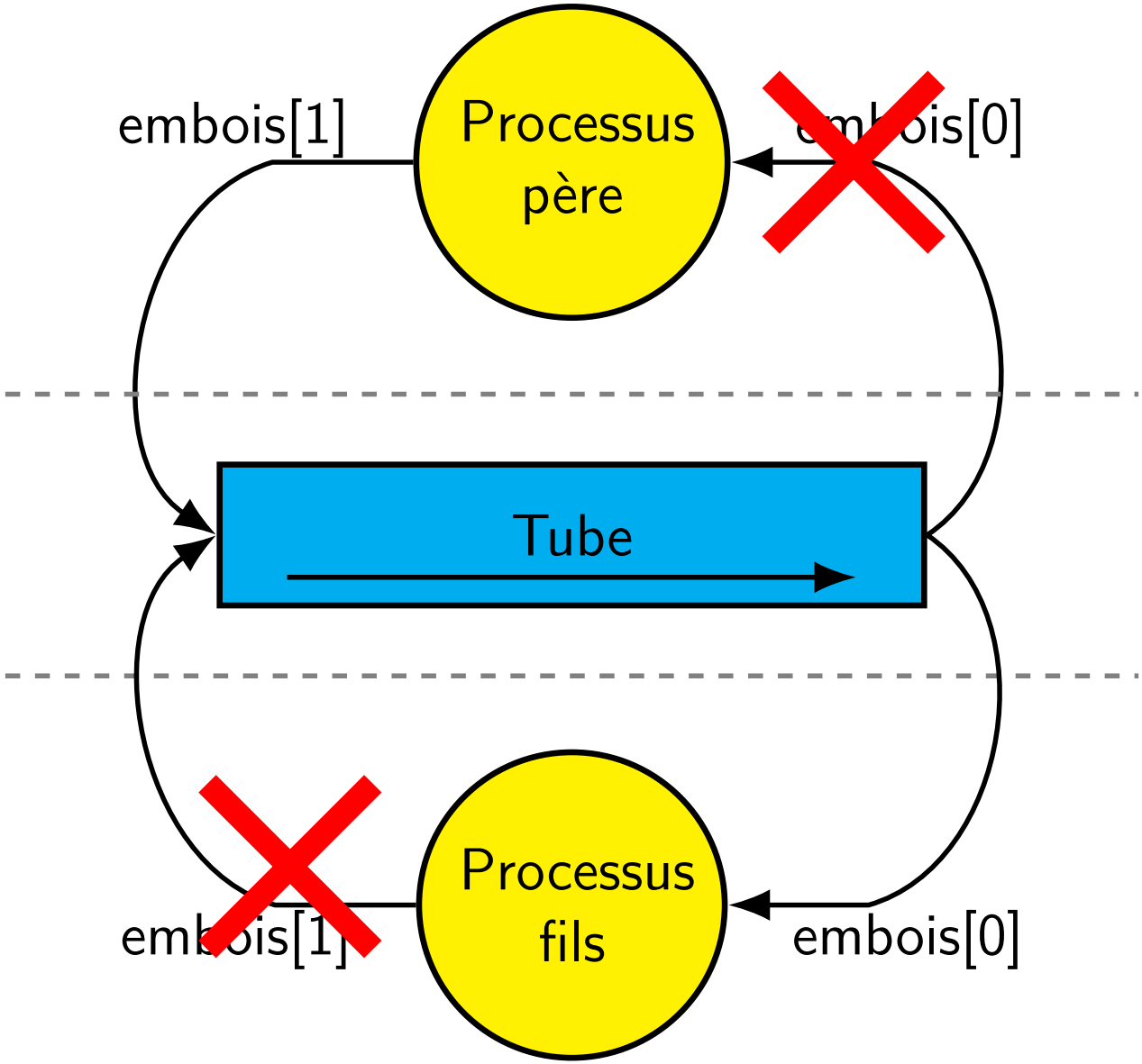
- Laisser des bouts trainer cause des problèmes de synchronisation
- Souvent: lecteur bloqué, car un bout d'écrivain reste quelque part

Plusieurs écrivains et/ou lecteurs ?

- C'est techniquement possible, mais :
- Bien comprendre les règles de synchronisation et d'atomicité
- Les clients doivent être coopératifs
- Messages de taille fixe aide beaucoup

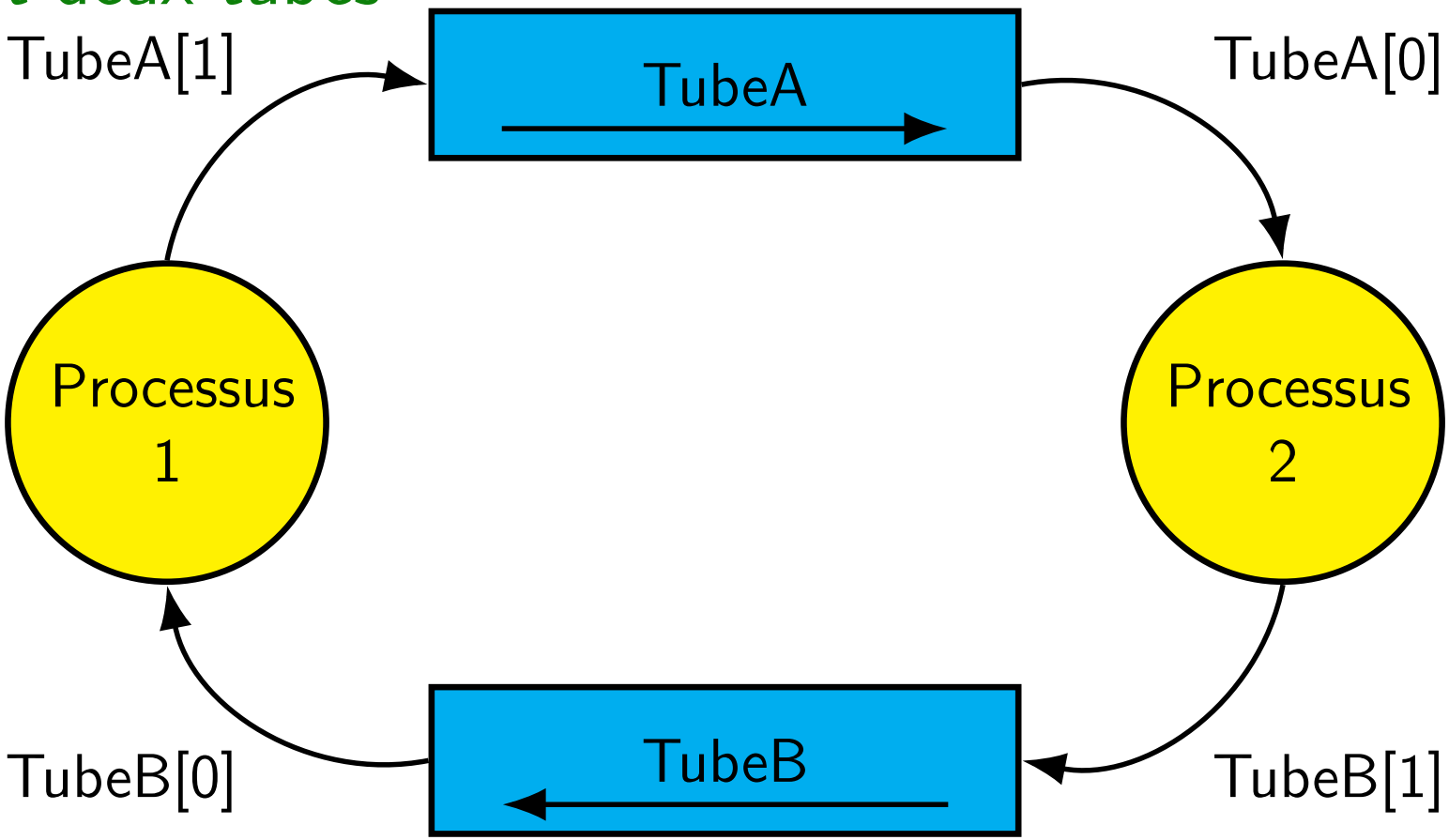
$\rightarrow$ Utiliser un autre IPC, c'est souvent moins risqué

Bonnes pratiques



Pour une communication bidirectionnelle

Il faut deux tubes



- Des systèmes offrent des tubes bidirectionnels (pas portable!)
- Utiliser un autre IPC,c'est souvent plus simple

Tubes shell

```
$ whoami | cowsay
```

```
#include "machins.h"
int main(int argc, char **argv) {
  int p[2];
  pipe(p);
  pid_t whoami = fork();
  if (whoami == 0 ) {
    dup2(p[1], 1);
    close(p[0]); close(p[1]);
    execlp("whoami", "whoami", NULL);
    perror("whoami"); return 1;
  }
  pid_t cowsay = fork();
  if (cowsay == 0) {
    dup2(p[0], 0);
    close(p[0]); close(p[1]);
    execlp("cowsay", "cowsay", NULL);
    perror("cowsay"); return 1;
  }
  close(p[0]); close(p[1]);
  waitpid(whoami, NULL, 0); waitpid(cowsay, NULL, 0);
  return 0;
}
```

Linux et pseudo fichiers tubes

- Ouvrir un pseudo fichier tube de `/proc/PID/fd` est possible
- Le mode d'ouverture indique quel bout du tube on obtient

→ Permet d'ajouter des lecteurs et des écrivains

→ Même si c'est souvent pas une bonne idée

```
(echo marco; sleep 1; echo polo) | lolcat &
echo trololo > /proc/$!/fd/0
```

Tubes nommés

Limites des tubes simples

- Via héritage des processus
- En créant le tube d'avance

$\rightarrow$ Communication entre processus indépendants difficile

Principe des tubes nommés

- Les tubes nommés ne sont pas **hérités,** mais **désignés**
- Donc plus besoin d'hériter des descripteurs
- Ni de créer le tube d'avance

Caractéristiques

- Exactement comme un tube simple
- Mais: ouverture d'un tube par un nom
- Plus: gestion des droits
- Plus: mécanisme de rendez-vous entre processus

Tubes nommées

Fichier spécial « tube »

- Crée avec `mkfifo(1)` et `mkfifo(3)` (et `mknod(2)`)
- L'inode (via chemins) désigne le tube
- Les droits du fichier sont les droits d'accès au tube
- Ouvrir (`open(2)`) le fichier c'est accéder au tube
- Le fichier **est** et **reste vide**
  - Le tube est entièrement en mémoire
  - Le fichier n'est qu'une astuce pour désigner

Rendez-vous

- `open(2)` **bloque** jusqu'à avoir un lecteur **et** un écrivain
- Le tube se comporte ensuite comme un tube simple

$\rightarrow$ Même synchronisation, même atomicité

Substitution de processus

Chez `bash(2)` : `<(CMD)`

Principe

- Exécute `CMD` dans un processus indépendant
- Où la sortie standard de `CMD` est redirigée dans un tube
- Substitue l'argument `<(CMD)` par le chemin du tube
- Quelqu'un qui ouvrira ce chemin sera connecté au tube

Deux implémentations

- Pseudo fichiers tubes (`proc(5)`) si disponible
- Tube nommé sinon

$\rightarrow$ On verra ça en lab

430 Sockets

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Communication par sockets

- POSIX sockets alias BSD sockets alias Berkeley sockets
- Pour la communication **réseau** entre **processus** `socket(7)`
- API offerte par le système d'exploitation

Socket ?

- Point de communication **abstrait**
- Boîte d'émission et de réception

$\rightarrow$ Un socket est un **descripteur de fichier**

Ceci n'est pas un cours de réseau

- On fait juste communiquer des processus
- On implémente des protocoles de communication
- Et on expose des abstractions et services aux processus

$\rightarrow$ C'est la responsabilité du système d'exploitation

API des sockets

API commune

- Différents et nombreux protocoles
- Différents types de communication
- Y compris propriétaires ou désuets

API générale

- Abstractions et appels système communs
- Mais détails spécifiques à chaque protocole
- Et à chaque variante Unix
- API complexe avec défauts de conception historiques : berk!

Autre sockets

- « Socket » devenu un terme générique
- Autres langages et systèmes ont leur propre API de sockets
- API souvent proche (concepts et vocabulaire), parfois meilleure

Types de communication

- 3 dimensions principales
- Nombreuses variations spécifiques

Granularité

- Flux d'octets (*stream*)
- Messages (*datagram*, *packet*)

Connectivité

- Connecté et bidirectionnel : modèle client-serveur
- Non connecté : modèle pair à pair

Fiabilité (réseau principalement)

- Fiable : service garanti, obligation de résultat
  Risques de sacrifices : moins de débit et plus de latence
- Non fiable : service au mieux, obligation de moyen
  Risques de pertes de données, modifications du contenu, pertes de l'ordre, duplications

Petite sélection d'appels système

- `socket(2)`, `socketpair(2)` création de sockets
- `bind(2)`, `listen(2)`, `accept(2)` coté serveur
- `connect(2)` coté client
- `write(2)`, `send(2)`, `sendto(2)`, `sendmsg(2)` émission
- `read(2)`, `recv(2)`, `recvfrom(2)`, `recvmsg(2)` réception
- `close(2)`, `shutdown(2)` fermeture
- `getsockopt(2)`, `getsockopt(2)`, `ioctl(2)` configuration
- `getsockname(2)`, `getpeername(2)` identification
- `lseek(2)` bien évidement interdits (erreur `ESPIPE`)

Création de socket

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

Domaine = famille de protocoles

- `AF_INET` pour IPv4 (`ip(7)`) ou `AF_INET6` pour IPv6 (`ipv6(7)`)
- `AF_UNIX` (ou `AF_LOCAL`) pour socket Unix (on va y venir)
- plus de 20 chez Linux, AF = *address family*

Type = sémantique de la communication

- `SOCK_STREAM`: flux d'octets, connecté, fiable
  Exemple: TCP chez IP (`tcp(7)`). Analogie: téléphone
- `SOCK_DGRAM`: messages, non connecté, non fiable
  Exemple: UDP chez IP (`udp(7)`). Analogie: courrier postal
- `SOCK_SEQPACKET`: messages, connecté, fiable

Protocole

- Protocole particulier si plus d'un pour un domaine et un type
- 0 = protocole par défaut

Socket du domaine Unix

- `AF_UNIX` (ou `AF_LOCAL`). Voir `unix(7)`
- `SOCK_STREAM`, `SOCK_DGRAM` ou `SOCK_SEQPACKET`

Ressemblances avec les tubes

- Communication **efficace** via la mémoire
- Zones de mémoire gérées par le système d'exploitation
- Processus lisent/écrivent dans des descripteurs
- Anonymes ou nommés
- Synchronisation
  - Lecture si vide: bloquée ou 0 si aucun écrivain
  - Écrivain `SIGPIPE` si aucun lecteur ou bloqué si plein

Différence avec les tubes

- Utilise l'API des sockets POSIX
- Bidirectionnel
- Connecté ou non connecté
- Flux d'octets ou messages

Adresse de socket

- Désignation d'un socket existant ou potentiel
- Structures C semi-opaques, fragiles et contraignantes (berk!)
- Détails spécifiques à chaque domaine
  Exemple chez IP: adresse IP + numéro de port

Structures d'adresses

- `struct sockaddr`: structure abstraite
  - Utilisée dans les signatures des appels système
- `struct sockaddr_XXX`: une version spécifique à chaque domaine
  - Utilisées pour allouer et accéder aux champs
  - `struct sockaddr_in6` pour IPv6
  - `struct sockaddr_un` pour les sockets Unix
- `struct sockaddr_storage` structure **assez grande** pour stocker n'importe quelle structure spécifique

$\rightarrow$ On caste allègrement entre des pointeurs de ces types (berk!)

Sockaddr du domaine Unix

```
struct sockaddr_un {
  sa_family_t sun_family;     /* AF_UNIX */
  char        sun_path[108]; /* Chemin */
};
```

Attention, `sun_path` a une taille max (berk!) non portable (reberk!)

Fichier spécial socket

- Utilisé pour « nommer » les socket*
- Type « `s` » selon `ls -l`
- Créé par `bind(2)` (on y reviendra)
- Supprimé par `unlink(2)`
- `open(2)` échoue (`ENXIO`)

*Linux offre aussi des sockets avec des noms « abstraits » indépendants du système de fichiers (non portable).

Envoyer et recevoir

Envoyer

- `write(int fd, const void *buf, size_t len)`
- `send(int fd, const void *buf, size_t len, int flags)`
- `sendto(int fd, const void *buf, size_t len, int flags, const struct sockaddr *addr, socklen_t addrlen)`
- `sendto` = `send` + addr = `write` + flags + addr
- `sendmsg(int fd, const struct msghdr *msg, int flags)`

Recevoir

- `read(int fd, void *buf, size_t len)`
- `recv(int fd, void *buf, size_t len, int flags)`
- `recvfrom(int fd, void *buf, size_t len, int flags, struct sockaddr *addr, socklen_t *addrlen)`
- `recvfrom` = `recv` + addr = `read` + flags + addr
- `recvmsg(int fd, struct msghdr *msg, int flags)`

Note: ne pas préciser `addr` si connecté.

Mode connecté

Serveur

- `bind(int fd, const struct sockaddr *ad, socklen_t adlen)`
  Expose une « adresse » publique
- `listen(int fd, int backlog)`
  Prépare un serveur à recevoir des clients
  `backlog` est soumis au culte du cargo (berk!). `SOMAXCONN` est bien.
- `accept(int fd, struct sockaddr *ad, socklen_t *adlen)`
  Récupère ou attend le prochain client
  - Retourne un **nouveau** socket, connecté directement au client
  - On a donc un socket d'écoute + un socket par client connecté

Client

- `connect(int fd, const struct sockaddr *ad, socklen_t adlen)`
  - Se connecte à un serveur spécifique
  - Retourne 0 si réussi, `fd` est maintenant **connecté**

  → `read` et `write` fonctionnent !

Exemple de client socket unix

```c
#include "machins.h"
int main(int argc, char **argv)
{
  int sock = socket(AF_UNIX, SOCK_STREAM, 0);

  struct sockaddr_un addr;
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, "sock", sizeof(addr.sun_path)-1);

  int res = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
  if(res==-1) { perror("connect"); exit(1); }

  write(sock, "Hello", 6);

  char buf[6];
  read(sock, buf, 6);
  printf("reçu: %s\n", buf);

  close(sock);
  return 0;
}
```

Exemple de serveur socket unix

```c
#include "machins.h"
int main(int argc, char **argv)
{
  int sock = socket(AF_UNIX, SOCK_STREAM, 0);

  struct sockaddr_un addr;
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, "sock", sizeof(addr.sun_path)-1);

  int res = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
  if (res == -1) { perror("bind"); exit(1); }
  listen(sock, SOMAXCONN);

  int cli = accept(sock, NULL, NULL);
  write(cli, "World", 6);
  char buf[6];
  read(cli, buf, 6);
  printf("reçu: %s\n", buf);
  close(cli);

  close(sock);
  unlink("sock");
  return 0;
}
```

Modèles populaires de serveurs (1/2)

Un client après l'autre

- Boucle principale de `accept(1)`
- Traite chaque client entièrement, et dans l'ordre
- Problèmes
  - Traitements courts seulement
  - Un client peut bloquer les autres

Multiplexage

- Une liste de clients connectés
- Boucle principale avec un `select(2)` ou `poll(2)`
  - surveille le socket d'écoute + chacun des clients connectés
  - socket d'écoute bouge = on accepte un nouveau client
  - socket d'un client bouge = on traite sa demande
- Problèmes : messages courts seulement, pas adapté aux cas compliqués

Modèles populaires de serveurs (2/2)

Multithread

- Un thread principal écoute
- On lance un nouveau thread par client (ou pool de threads)
- Problèmes : programmation multithread

Multiprocessus

- Un processus principal écoute
- Un sous-processus (`fork(2)`) par client (ou pool de processus)
- Problème : lourd et isolation des clients
- Avantage : robuste et isolation des clients

Données auxiliaires

- Données spécifiques supplémentaires aux messages
- Alias « messages de contrôle » (*cmsg*)
- Contenu sémantique et spécifique :
  Contenu ont du sens pour le système d'exploitation
- Mais ce qui est possible est spécifique à chaque domaine
- `recvmsg(2)` et `sendmsg(2)` pour les utiliser
- `cmsg(3)` pour y accéder
- API horrible (berk!)

Descripteurs de fichiers auxiliaires

- Utilisable dans les sockets du domaine Unix
- `SCM_RIGHTS` passe des fichiers ouverts
- L'émetteur attache des **descripteurs de fichiers**
- Le système crée des descripteurs dans le processus récepteur
- C'est pas forcément les mêmes numéros de descripteur
- Mais c'est les mêmes fichiers ouverts

Paire de sockets

- `socketpair(2)` crée deux sockets connectés
- Ressemble fortement à `pipe(2)`
- Mais bidirectionnel !
- Messages possibles (pas seulement flux d'octets) !

500 Synchronisation

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Systèmes concurrents

- Des éléments logiciels (voire matériels)
- Sont capables de s'exécuter en « **même temps** »
- Indépendamment du **moment** ou de l'**ordre** de leur exécution
- **Sans tout briser**

Questions

- Ça veut dire quoi en « même temps » ?
- Quel rapport avec les systèmes d'exploitation ?

En même temps ?

Concurrence

Un élément logiciel s'exécute avant que les autres finissent

L'**ordre** des exécutions de chacun est variable

- Changements de contexte et politiques d'ordonnancements
- Interruptions matérielles
- Signaux logiciels
- Programmation événementielle
- Invocation de sous-programmes (en tirant *vraiment* l'élastique)

Parallélisme

Exécution physiquement au même **moment**

- Architectures multiprocesseurs et multicœurs
- Voire systèmes distribués

  Mais rendu là on a d'autres difficultés en plus

Exemple: incrément

- Suspects: Deux threads t1 et t2 d'un même processus
- Victime: Une variable globale `i` partagée
- Code du crime, exécuté par les deux threads:

```
i++;
```

Où est le problème?

Exemple: incrément

- Suspects: Deux threads t1 et t2 d'un même processus
- Victime: Une variable globale `i` partagée
- Code du crime, exécuté par les deux threads:

```
i++;
```

Où est le problème?

```
; assembleur x86    ; pep8       ; Français
movq i(%rip), %eax  ; LDA  i,d   ; Charge i dans A
addq $1, %eax       ; ADDA 1,i   ; Incrémente A
movq %eax, i(%rip)  ; STA  i,d   ; Sauve A dans i
```

On a pas de problème !

Thread 1	Thread 2	i	A(t1)	A(t2)
		0	?	?
LDA i,d				
ADDA 1,i				
STA i,d				
	LDA i,d			
	ADDA 1,i			
	STA i,d			

- `i` commence à 0
- Chacun des deux threads incrémente `i`
- `i` vaut combien à la fin ?
- Alors ?

On a pas de problème !

Thread 1	Thread 2	i	A(t1)	A(t2)
		0	?	?
LDA i,d		0	**0**	?
ADDA 1,i		0	**1**	?
STA i,d		**1**	1	?
	LDA i,d	1	1	**1**
	ADDA 1,i	1	1	**2**
	STA i,d	**2**	1	2

- `i` commence à 0
- Chacun des deux threads incrémente `i`
- `i` vaut 2 à la fin
- Voilà !

On a un problème !

Thread 1	Thread 2	i	A(t1)	A(t2)
		0	?	?
LDA i,d				
ADDA 1,i				
	LDA i,d			
	ADDA 1,i			
	STA i,d			
STA i,d				

- `i` commence à 0
- Chacun des deux threads incrémente `i`
- `i` vaut combien à la fin ?
- Alors ?

On a un problème !

Thread 1	Thread 2	i	A(t1)	A(t2)
		0	?	?
LDA i,d		0	**0**	?
ADDA 1,i		0	**1**	?
	LDA i,d	0	1	**0**
	ADDA 1,i	0	1	**1**
	STA i,d	**1**	1	1
STA i,d		**1**	1	1

- `i` commence à 0
- Chacun des deux threads incrémente `i`
- `i` vaut 1 à la fin
- Aïe !

Instructions machines

Concurrence vs parallélisme

- Monoprocesseur
  Problème quand changement de contexte au mauvais moment
- Multiprocesseur
  Probabilité de problème bien plus grande

$\rightarrow$ Débogage difficile : syndrome « chez moi ça marche »

Architectures matérielles

`i++` peut être **une seule** instruction « `addq $1, i(%rip)` »

- Monoprocesseur: Plus de vraiment de problème
  - Changement de contexte *avant* ou *après* l'instruction
- Multiprocesseur: Toujours problème de concurrence
  - 1 instruction, mais plusieurs cycles
  - RAM partagée
  - Caches et cohérence mémoire (INF4170)

Quel rapport avec les systèmes d'exploitation ?

Traitement d'événements logiciels et matériels

- Des évènements fondamentalement **imprévisibles**
- Interruptions matérielles
- Appels système de processus (si vrai parallélisme)

Performance des systèmes d'exploitation

Exploitation des possibilités de concurrence et parallélisme

- Traitements parallèles internes : threads système
- Préemption système : les noyaux modernes sont préemptifs
- Une approche « un seul processus en appel système à la fois » fonctionne, mais est très limitante côté performance

Centre de service

- Offre de **mécanismes** de **synchronisation** pour les processus

Classification des programmes concurrents

Disjointe

- Pas d'interaction entre entités logicielles

$\rightarrow$ Facile mais ça n'arrive pas souvent

Compétitive

- Des ressources partagées existent
- On veut s'assurer de leur disponibilité et cohérence

$\rightarrow$ C'est un travail pour le système d'exploitation

Coopérative

- Des éléments logiciels coopèrent
- La concurrence fait partie du programme
- C'est des modèles de programmation spécifiques

$\rightarrow$ Le système d'exploitation offre des services de synchronisation

$\rightarrow$ Mais il y a aussi des ressources à gérer

Situation de compétition (*race condition*)

- Situation où le **résultat** est **différent**
- Dépendamment du **moment** ou de l'**ordre** d'exécution

→ C'est souvent **problématique**

Résultats différents

- Tous pas forcément **corrects**

→ Bogue, y compris de sécurité (INF600C)

Ordre et moment

- Pas connus ou pas contrôlables
- Car ordonnancement, latence matérielle, événements externes...
- Donc situations difficiles à **reproduire** et à **tester**

→ Indéboguable (*Heisenbogues*)

Problématiques de concurrence partout

- Retrait bancaire

```
if(montant < solde) {
    solde =- montant;
    return montant;
} else {
    return 0; // Solde insuffisant
}
```

- Suppression d'un maillon d'une liste doublement chainée

```
if (current->next != NULL)
  current->next->prev = current->prev;
if (current->prev != NULL)
  current->prev->next = current->next;
```

Question

- Trouver des scénarios problématiques

Problématiques de concurrence partout

- Deux processus parallèles font `fork(2)`
  - Attribuer correctement un PID différent
  - Ne pas corrompre la table des processus
- Deux threads parallèles font `malloc(3)`
  - Attribuer correctement une zone mémoire distincte
  - Ne pas corrompre les structures internes du tas
- Deux processus lisent écrivent en même temps dans un tube
  - Attribuer des octets différents (sans en perdre)
  - Ne pas corrompre les structures internes du tube
- Résoudre un chemin (`path_resolution(7)`)
  - Alors qu'un processus renomme ou déplace des répertoires
- Problèmes théoriques classiques de synchronisation
  - Diner des philosophes
  - Producteurs et consommateurs (file bornée)
  - Coiffeur endormi (file d'attente)
  - Écrivains et lecteurs (accès concurrents en lecture ou écriture)

Besoin d'ordre et de discipline !

- Éviter les accès simultanés qui rendraient le système incohérent
- Garantir une certaine équité
- Maintenir la performance
- Éviter que le système ne se blo

510 Section critique

INF3173

Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Problème de concurrence

- Les threads (et processus)
- Le système d'exploitation lui-même
- Sont fortement concurrents, voire parallèles

→ Comment gérer cette concurrence **correctement** de façon programmative ?

Objectifs

- Contrôler les situations de compétition
- Prévenir la corruption de ressources partagées
- Indépendamment du type de ressources
- Rester efficace

Section critique

Section critique = Zone de code

- Zone de code = morceau de programme
- Attention, pas forcément contiguë

Section critique = Zone d'exclusivité

- Exécuté que par un seul thread* maximum à la fois
- Qui manipule une **ressource** potentiellement **partagée**

$\rightarrow$ On **protège** une ressource
en **contraignant** l'exécution du code
qui **manipule** cette ressource

*Sans perte de généralité, on utilise « threads », mais ça s'applique pareil aux processus monothread, tâches noyau ou toute autre entité logicielle en cours d'exécution.

Un thread en section critique

- N'est pas nécessairement actif à 100%
- Il peut faire des appels système bloquants (et devenir bloqué)
- Il peut être préempté (et devenir prêt)

Section critique en bref

- Tant qu'un thread n'est pas sorti
- Aucun autre thread ne peut y rentrer

Les 4 règles des sections critiques

1. Au **maximum**, un **seul** thread à la fois en section critique
2. Pas de **supposition** sur la vitesse ou le nombre de threads
3. Un thread **hors** section critique **ne bloque pas** les autres
4. Pas d'attente **infinie** pour entrer en section critique (**famine**)

Solution qui marche pas

```
long i; // ressource partagée
int flag; // booléen protegant la ressource

void inc(void) {
  while (flag) { } // on attend l'absence du flag
  flag = 1;        // on verrouille
  i++;             // on manipule
  flag = 0;        // on dévérouille
}
```

- Toute « stratégie » de ce type est à bannir !

Question

- Lesquelles des 4 règles sont violées ?
- Trouvez un scénario (ordonnancement) où ça ne fonctionne pas

Exclusion mutuelle stricte à tours

```
long i; // ressource partagée

extern int nb; // nombre de threads
int tour;      // à qui c'est le tour?

void inc(int t) { // on est le thread `t` (de 0 à nb-1)
  while (t != tour) {}  // on attend notre tour
  i++;                  // on manipule
  tour = (tour+1) % nb; // on passe au suivant
}
```

Question

- Lesquelles des 4 règles sont violées ?

Solution de Peterson (1981)

- Pour deux threads seulement

```
long i; // ressource partagée

int flag[2]; // qui est interessé ?
int tour;    // à qui le tour (si les deux en veulent) ?

void inc(int k) { // k c'est moi, !k c'est l'autre
  flag[k] = 1; // on veut entrer
  tour = !k;   // on est poli
  while (flag[!k] && tour == !k) { } // attente
  i++;         // on manipule
  flag[k] = 0; // on n'en veut plus
}
```

- Se généralise à un nombre quelconque de threads.

Barrière mémoire

Instruction (ou indication) destinée:

- Aux processeurs
  Force les écritures et lectures du bon côté de la barrière
- Au compilateurs C
  Prévient les optimisations qui changent la sémantique
- Coût non négligeable

$\rightarrow$ Les détails dans d'autres cours...

Exemples

- Instruction `mfence` en x86
- Pas de mot clé C standard
- `atomic_thread_fence` de C11 `stdatomic.h`
- Extension C de gcc : `__atomic_thread_fence`
  (et le plus ancien `__sync_synchronize`)

Volatilité

- Mot clé C `volatile`
- Déclare une donnée comme mutable par quelque chose d'autre (comme un autre thread)

Pour le compilateur seulement

- Informe qu'une modification indépendante est possible
- Et qu'il doit éviter des optimisations
- Par défaut, le compilateur n'est pas conservateur !
- Les options de compilation changent le comportement du code

Volatilité

Quand le compilateur voit

« `int x=0; while(x==0) {}` »

- Si `x` n'est pas volatile
  On peut juste implémenter une boucle infinie
- Si `x` est volatile
  On **doit** tester `x` à chaque tour, « au cas où... »

Bonne ou mauvaise chose ?

- La présence de `volatile` dans du code est souvent **douteuse**
- Son utilisation ne permet pas **magiquement** de résoudre les problèmes de concurrence
- Son **coût** est non négligeable
- *volatile considered harmful*

Peterson + volatile + barrières mémoire

```
#include <stdatomic.h>

long i; // ressource partagée

volatile int flag[2]; // qui est interessé ?
volatile int tour;    // à qui le tour ?

void inc(int k) { // k c'est moi, !k c'est l'autre
  flag[k] = 1; // on veut entrer
  tour = !k;   // on est poli
  atomic_thread_fence(memory_order_seq_cst);//barrière
  while (flag[!k] && tour == !k) { } // attente
  atomic_thread_fence(memory_order_seq_cst);//rebarrière
  i++;         // on manipule
  atomic_thread_fence(memory_order_seq_cst);//rerebarrière
  flag[k] = 0; // on n'en veut plus
}
```

Le matériel à la rescousse

Idée : empêcher le changement de contexte

- Masquer les interruptions matérielles au niveau du processeur (dont l'horloge)
- Verrouiller le bus (en multiprocesseurs)

Problèmes

- Grain grossier
- Couteux
- Seul le noyau peut faire ça
  (ok pour lui, mais pas pour les processus)

Instructions machine atomiques

- C11 `_Atomic` et `stdatomic.h`, extension gcc ou assembleur
- Permet des manipulations **atomiques** :
  Indivisible pour l'observateur

Exemple : incrément atomique

- `lock add` (x86)
- `__atomic_add_fetch` (gcc)
- `++` sur un type `_Atomic` (C11)

Mise en œuvre matérielle

- Accès mémoire exclusif de la donnée le temps de l'exécution
- Exemple : verrouillage des lignes de cache mémoire
- Les détails dans un autre cours...

Limites

- Coût non nul
- Seulement certaines instructions et valeurs simples

Incrément atomique

Extension gcc

```
long i; // Ressource partagée
void inc(void) {
  // On manipule sans verrous
  // Directement fonction built-in gcc
  __atomic_add_fetch(&i, 1, __ATOMIC_RELAXED);
}
```

C11 avec _Atomic

```
_Atomic long i; // Ressource partagée
void inc(void) {
  i++; // on manipule
}
```

En vrai ?

Le compilateur compile vers des instructions machine atomiques

Il peut ajouter aussi des barrières

Sections critiques plus grosses?

On implémente un verrou atomique

- Instructions « *test and set* », « *compare and exchange* »...
- x86: `xchg`, `lock cmpxchg`...
- gcc: `__atomic_test_and_set`, `__atomic_compare_exchange`...
- C11: `atomic_flag_test_and_set`, `atomic_compare_exchange`...

```
#include <stdatomic.h>

long i; // ressource partagée
atomic_flag flag; // booléen protegant la ressource

void inc(void) {
  while(atomic_flag_test_and_set(&flag)) {}
  i++; // on manipule
  atomic_flag_clear(&flag);
}
```

- C'est une version fonctionnelle de la « solution qui marche pas »

Attente active (*spinlock*)

```
while (...) { }
```

- Quand ça fonctionne, ça reste inefficace

→ Ça gaspille du temps processeur à **activement rien faire**

Questions

Voici deux autres propositions :

- `while (...) { sched_yield(); }` [†]
- `while (...) { sleep(1); }`
- Pourquoi c'est pas vraiment beaucoup mieux ?
- Y a-t-il des cas où c'est même pire que la proposition initiale ?

[†] En gros `sched_yield(2)` force un appel à l'ordonnanceur pour éventuellement donner le processeur à un autre thread.

Limites des proposions à date

Objectifs

- Contrôler les situations de compétition ✔
- Prévenir la corruption de ressources partagées ✔
- Indépendamment du type de ressources ✔
- Rester efficace ⚙

Limites

- Approches purement algorithmiques limitées
- Instructions machine spécifiques peu portables
- Bricolage bas niveau
- Potentiellement inefficace (*spinlock*)

Solution : Un nouveau niveau d'indirection

- Langages, bibliothèques et systèmes d'exploitation à la rescousse
- Ils fournissent des services et des modèles de synchronisation
- Que les développeurs peuvent utiliser

520 Outils de synchronisation

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Concurrence et sections critiques

On y arrive à la main, mais

- C'est compliqué
- C'est bas niveau
- C'est douteux d'un point de vue performance (attente active)

Peut-on faire mieux ?

- Le système d'exploitation est $\pm$ capable de se débrouiller
- Qu'en est-il des processus et threads ?

Solution : Un nouveau niveau d'indirection

- Système d'exploitation, bibliothèques et langages
- Ils fournissent des **outils**

$\rightarrow$ Services et des modèles de synchronisation clé en main

- Les développeurs peuvent les utiliser

Outils spécialisés

- La programmation concurrente **reste** complexe
- Ces outils ne suppriment pas les difficultés fondamentales
- Au mieux, ils les transforment et les déplacent...

Besoin de performance

- Rappel : les appels système coutent cher
- Une partie des mécanismes est faite en espace utilisateur
  $\rightarrow$ Langages et bibliothèques
- Une autre partie en mode noyau, par le système d'exploitation
  $\rightarrow$ ordonnancement et états d'exécutions des processus

Concrètement

- Implémentés avec les techniques primitives précédentes
- Garantissent l'efficacité et la fiabilité

$\rightarrow$ Bienvenue dans la programmation concurrente moderne !

$\rightarrow$ Rappel, ceci n'est pas un cours de programmation concurrente

Éliminer l'attente active

Le système d'exploitation gère le cycle de vie des threads

Solution à l'attente active

- Un thread veut entrer en section critique déjà occupée
  On le bloque (passage de l'état actif à bloqué)
  On appelle l'ordonnanceur
- Un thread sort d'une section critique
  Un autre thread était en attente ?
  On le réveille (passage à l'état prêt)
  Et on appelle l'ordonnanceur
- Le tout de façon performante !

Question

- Quels sont les cas où l'attente active est préférable à un changement de contexte ?

Mutex (ou verrou, *lock*), de *mutual exclusion*

- Concept général de verrouillage de section critique
- Mais détails spécifiques en fonction du contexte (système d'exploitation, bibliothèque, langage de programmation)

Opérations générales

- Verrouiller : ça entre ou ça attend
- Déverrouiller : ça débloque les autres
- Tenter : ça entre ou ça échoue

Variations

- Actif (*spinlock*) ou bloquante (passage à l'état bloqué)
- Rapide (un booléen), récursif (un compteur), avec détection d'erreur (on y reviendra)

Mutex pthread

- Fourni de base chez `pthreads(7)`
- `pthread_mutex_lock(3)`, `pthread_mutex_unlock(3)`, `pthread_mutex_trylock(3)`, etc.
- Limités aux threads d'un même processus
- RTFM pour les détails

```
#include<pthread.h>

long i; // Ressource partagée
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void inc(void) {
  pthread_mutex_lock(&mut); // on verouille
  i++; // on manipule
  pthread_mutex_unlock(&mut); // on déverouille
}
```

Sémaphore

- Concept historique spécifique (Dijkstra, 1962)
- C'est un compteur de ressources
- Le compteur bloque s'il n'y a plus de ressource
- Sémantique atomique garantie (détail d'implémentation)
- Pas d'attente active et pas de famine
- Cas particulier: sémaphore binaire (deux valeurs possibles), ressemble au mutex d'un point de vue de l'implémentation

Sémaphore : pseudo-code

```
Sémaphore { entier val; liste_processus L; };
```

Demander(Sémaphore S)

S.val -= 1;
si *S.val* $<$ *0* **alors**
    ajouter demandeur à S.L;
    le passer à bloqué;
fin

Libérer(Sémaphore S)

S.val += 1;
si *S.val* $<=$ *0* **alors**
    enlever un processus de S.L;
    le passer à prêt;
fin

Remarque de vocabulaire

- Demander = Entrer = Down = P = *proberen* = tester
- Libérer = Sortir = Up = V = *verhogen* = incrémenter

Sémaphores Unix

- Ressources gérées globalement au niveau du système
- Persistant jusqu'à l'arrêt du système ou une libération explicite
- Partageables entre threads, processus et utilisateurs
- Sémaphores POSIX : `sem_overview(7)`
- Sémaphores System V : `semget(2)`, `semop(2)`

```
#include <semaphore.h>
long i; // Ressource partagée
static sem_t sem;
void inc_init(void) { // initialisation
  sem_init(&sem, 0, 1); // initialise à 1
}
void inc(void) {
  sem_wait(&sem); // verrouillage
  i++;            // on manipule
  sem_post(&sem); // dévérouillage
}
```

Sémaphore vs Mutex

Sémaphore

- Compteur de ressources : atomique, efficace et équitable
- Ceux qui incrémentent sont pas forcément ceux qui décrémentent

Mutex système

- Délimite une section critique qui protège une ressource partagée
- Le thread qui déverrouille est celui qui a fait le verrouillage initial
- Information utile pour le système d'exploitation

Avantages des mutex système

- Déverrouillage des mutex d'un thread qui termine
- Inversion de priorité possible (on y reviendra)
- Vérification d'erreur possible:
  - Un thread déverrouille un mutex sans l'avoir verrouillé
  - Situation d'interblocage (on y reviendra)

Autres outils et techniques de synchronisation

- Variable de condition (file d'attente + service de réveil)
- Moniteur (sous-programmes + mutex implicite + variables de condition)
- Barrière
- Verrou lecture-écriture
- RCU (*read-copy-update*) : technique sans verrouillage
- Structures de données parallèles clé en main
- Opérations atomiques (C11)
- Etc.

Ce sont des outils et abstractions de programmation

Pour plus de détails

- Programmation concurrente et parallèle (INF5171)
- Programmation parallèle haute performance (INF7235)
- *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, Paul E. McKenney.

Futex (*fast userspace mutex*)

- Bloque un processus jusqu'à un réveil explicite
- Bas niveau et délicat

→ Erreur classique : on bloque un processus
Pile au moment où la condition du blocage disparait

- Sert aux bibliothèques pour implémenter les autres mécanismes
  → Mutex pthread et autre
- `futex(2)` sous Linux

Exemple tentative futex

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/futex.h>
#include "myatomic.h"
long i; // ressource partagée
int flag; // 0=libre; 1=occupé
void inc(void) {
  while(xchg(&flag, 1) == 1) {
    // on s'endort si occupé
    syscall(SYS_futex, &flag, FUTEX_WAIT, 1, 0, 0, 0);
  }

  i++; // on manipule

  xchg(&flag, 0);
  // on réveille un endormi, s'il y en a
  syscall(SYS_futex, &flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```

Exemple futex mieux

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/futex.h>
#include "myatomic.h"
long i; // ressource partagée
int flag; // 0=libre; 1=occupé; 2=endormi
void inc(void) {
  int c = cmpxchg(&flag, 0, 1);
  if(c != 0) {
    if (c != 2) c = xchg(&flag, 2);
    while (c != 0) {
      syscall(SYS_futex, &flag, FUTEX_WAIT, 2, 0, 0, 0);
      c = xchg(&flag, 2);
    }
  }
  i++; // on manipule
  if(xchg(&flag, 0) == 2)
    syscall(SYS_futex, &flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```

530 Interblocage

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Interblocage

Alias

- Deadlock
- Verrou fatal
- Étreinte fatale
- Embrasse mortelle

Principe

- Plusieurs processus* sont bloqués entre eux et ne peuvent progresser

*Sans perte de généralité on utilise « processus », mais ça s'applique pareil aux threads, tâches noyau ou toute autre entité logicielle en cours d'exécution.


BANCO REAL
11455

Ressources

Quelques ressources (au sens large)

- Imprimante
- CPU
- Sémaphore
- Section critique (mutex)

Ressources et interblocages

- Les interblocages découlent de l'allocation des ressources

Ressources

Événements liés aux ressources

- Demander la ressource
- Utiliser la ressource
- Libérer la ressource

Si un processus demande une ressource déjà prise

- Erreur
- Attente
- Attente temporisée

Interblocage

Définition plus formelle

- Un ensemble de processus sont en **interblocage** si chaque processus dans cet ensemble est **en attente** d'un événement que **seulement un autre** processus de ce **même ensemble** peut déclencher — Tanenbaum
- L'événement peut-être est la libération d'une ressource

En cas d'interblocage un processus ne peut

- Ni continuer son exécution
  Car il est bloqué
- Ni débloquer un autre processus
  En libérant une ressource
  Car il est bloqué

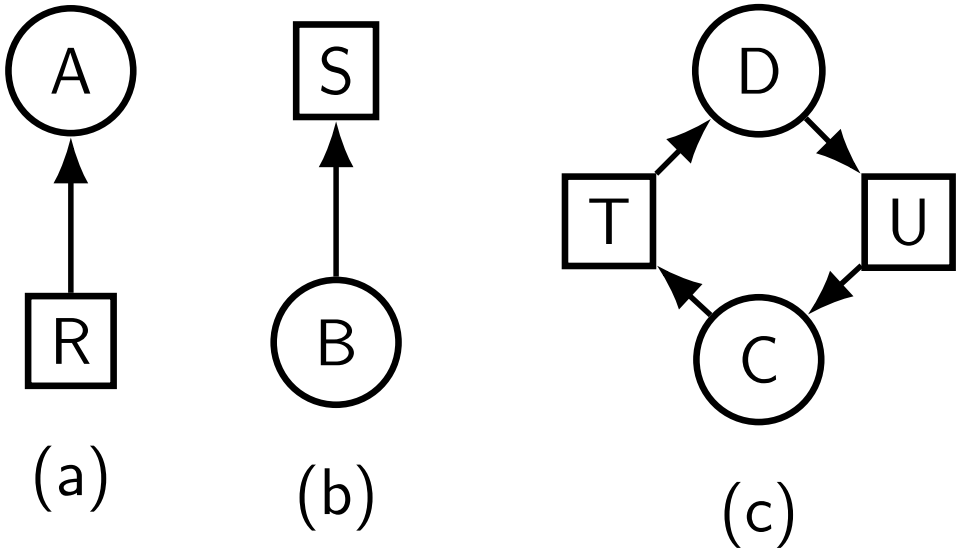
Caractérisation d'un interblocage

Les 4 conditions nécessaires et suffisantes de l'interblocage

- Exclusion mutuelle
  La ressource est soit disponible, soit assignée
- Détention multiple (*hold and wait*)
  Un processus qui détient une ressource peut en demander d'autres
- Pas de réquisition
  Une ressource détenue par un processus doit être libérée par lui
- Attente circulaire
  Il doit y avoir un cycle dans les attentes d'événements

Description des allocations

Graphe biparti orienté



- (a) Ressource R assignée au processus A
- (b) Processus B demande et attend S
- (c) C et D sont en interblocage

Gestion des interblocages

Quatre stratégies

- Ignorer le problème
- Détecter et résoudre
- Prévenir le problème
- Éviter dynamiquement

Ignorer le problème

Principe

- Prétendre que le problème n'existe pas

Raisonnable si

- Interblocages rares
- Autres solutions trop coûteuses/restrictives

Avantage

- Facile à comprendre
- Facile à mettre en œuvre

Détecter et résoudre

Principe

- Vérifier si les processus sont en interblocage et les débloquer

Comment savoir s'il y a interblocage

- Modélisation et analyses de graphe
- Tester

Comment débloquer (sans tout casser)

- Échec du verrouillage
- Retirer de force une ressource
- Restauration d'un état antérieur (*rollback*)
- Éliminer un processus

Prévenir le problème

Principe

- Éliminer une condition de l'interblocage

Exemples

- *Spooling* : seul un processus a la ressource
- Ressources toutes demandées d'un coup
- Permettre la préemption
- Ordonner les ressources (donc les demandes)

$\rightarrow$ Aucun n'est nécessairement faisable

Éviter dynamiquement

Principe

- Forcer l'ordonnanceur à faire le bon choix
- C'est possible via des informations supplémentaires

Idée

- Un état est sûr s'il existe une séquence d'allocations qui permet aux processus d'aller jusqu'au bout
- Exécuter une allocation que si l'état qui en résulte est sûr

Résolutions en pratique

Pas de solution ultime

- Le coût et l'efficacité d'une technique dépendent fondamentalement de la nature des ressources

En pratique,

- Les SE actuels ignorent le problème pour les utilisateurs
- Seuls les SE critiques prennent éventuellement en compte ce genre de problème

Problème cousin - *Livelock*

Jeu de mots sur *deadlock*

- Les processus ne sont pas bloqués
- Mais ne progressent pas non plus

$\rightarrow$ Le CPU est utilisé seulement pour retenter

- Transformer un deadlock en livelock n'est pas un progrès

Problème cousin - Famine

Synonymes

- *Starvation*
- Privation de ressource

Définition

- Un groupe de processus partagent une ressource
- Sans interblocage
- Certains processus n'obtiennent jamais la ressource

$\rightarrow$ Problème de l'attente infinie

Problème cousin - inversion de priorité

Scénario explicatif

- 3 processus de priorité stricte P1 $>$ P2 $>$ P3
- P3 arrive, s'exécute, demande et acquiert une ressource R
- P2 arrive et s'exécute (préempte P3)
- P1 arrive, s'exécute (préempte P2), demande la ressource R
  ... et passe à bloqué
- P2 s'exécute alors à nouveau

Problème

- P2 passe devant tout le monde
- P1 termine dernier, c'était pourtant le plus prioritaire

Solutions

- Revoir la conception: est-ce **normal** que P3 puisse bloquer P1 ?
- Renforcement de la priorité

Renforcement de la priorité

Augmentation temporaire de priorité

- P3 devrait passer avant P2
- Jusqu'à sortir de sa section critique et libérer son mutex
- Pour pouvoir débloquer P1 le plus tôt possible

Nécessite la coopération du système d'exploitation

- Mutex avec priorité statique (`PTHREAD_PRIO_PROTECT`)
  Le processus qui détient le mutex (P3) gagne en priorité
- Héritage de priorité (`PTHREAD_PRIO_INHERIT`)
  Le processus qui attend un mutex (P1) transfert automatiquement sa propre priorité au détenteur du mutex (P3)
- Renforcement aléatoire (exemple Windows)
  Comme P3 détient un mutex, il *peut-être* va gagner un petit *boost* pour *peut-être* sortir de sa section critique.

Dîner des philosophes (Dijkstra)

Données

- 5 philosophes. Chacun pense ou mange (temps inconnu)
- 5 fourchettes
- 5 plats de spaghettis
- 2 fourchettes sont nécessaires pour manger

Problème

Comment faire tourner le système sans bogue ?

- Sans corruption
- Sans famine
- Sans interblocage
- Efficacement

600 Gestion de la mémoire

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Rôles du système d'exploitation

Répartir (allouer) la mémoire

- Pour les processus
- Pour lui-même

$\rightarrow$ Efficacement et sans gaspiller

Contrôler et protéger

- Isoler la mémoire des processus

$\rightarrow$ Chaque processus a l'impression d'être seul

Offrir des services

- Allocation dynamique
- Mémoire partagée
- Configuration de politiques
- `brk(2)`, `mmap(2)`

Objectifs du chapitre

Comprendre

- Comment la mémoire est gérée par le système d'exploitation (et le matériel)
- Les possibilités offertes par la gestion moderne de la mémoire
- Les algorithmes liés à la gestion de la mémoire (utilisables dans d'autres contextes)

Mémoire

La bonne mémoire

- est rapide
- est grande
- est bon marché
- est non volatile,
- mais elle n'existe pas (encore)

En attendant : hiérarchie de mémoire

- Caches processeur
- RAM
- Disques

Nos hypothèses

- Mémoire = grand tableau d'octets
- Accès matériel efficace (CPU $\leftrightarrow$ RAM)
- Les processus vont et viennent

Dans la vraie vie ?

- Processeurs modernes (caches, pagination avancée, etc.)
- Parallélisme et des architectures multiprocesseurs et multicœurs
- Architectures hétérogènes (NUMA, *non uniform memory access*)
- `/proc/meminfo`, `/proc/vmstat`

Problèmes à résoudre

- Adresses explicites dans le code machine d'exécutables
  Comment lancer plusieurs processus ?
- Agrandissement de la mémoire allouée à un processus
  Que faire si on n'a pas la place autour ?
- Clone de processus avec `fork(2)`
  Mais que faire avec les pointeurs existants ?
- Comment partager de la mémoire entre processus ?
- Comment avoir des modes d'accès (écriture,exécution) spécifiques ?

Question

- Pourquoi on voudrait partager la mémoire entre processus ?

Solution : un nouveau niveau d'indirection

- Ne plus permettre aux processus de pointer directement la mémoire
- Les adresses utilisées par les processus (pointeurs, opérandes des instructions machine, etc.) ne sont pas des adresses absolues en RAM

→ On **convertit** les adresses logiques (des processus) En adresses physiques (en RAM)

```
#include "machins.h"
int main(void) {
  pid_t p = fork();
  printf("pid: %d ; adr: %p ; val: %d\n", getpid(), &p, p);
  pause();
  return 0;
}
```

Mémoire physique (ou réelle)

Mémoire physique (ou réelle)

- La RAM (un grand tableau d'octets)
- `free(1)`, `/dev/mem`, `/proc/meminfo`, etc.

Adresse physique (ou réelle)

- Un numéro d'octet dans la RAM

Espace d'adressage physique (ou réel)

- L'ensemble des adresses physiques possibles
- En général la taille du bus d'adresse
- x86-64: actuellement $\approx$ 48bits (256To)

Mémoire logique (ou virtuelle)

Adresse logique (ou virtuelle)

- Les adresses utilisables par le logiciel
- Pointeurs, registres, opérandes, etc.
- Attention: adresse « linéaire » est parfois (abusivement) utilisé

Espace d'adressage logique (ou virtuel)

- L'ensemble des adresses logiques possibles
- En général la taille d'un pointeur ou d'un registre
- x86-64: actuellement $\approx$ 48bits (voire 57bits)
  Les pointeurs sont pourtant 64bits (économie!)

Mémoire logique

- La mémoire associée au logiciel qui s'exécute (processus)
- `/proc/PID/mem`, `pmap(1)` et colonne `VSZ` de `ps(1)`

$\rightarrow$ On y reviendra pour les détails

Unité de gestion mémoire (MMU)

- MMU = *memory management unit*
- Composante matérielle, sur le **microprocesseur**
- Traduit **automatiquement** et **efficacement**
  Adresses logiques $\rightarrow$ adresses physiques
- Les opérandes et pointeurs sont en adresse logique
- Ce qui circule sur le bus d'adresse est en adresse physique

$\rightarrow$ C'est transparent pour le logiciel

Bonus

- Les paramètres de traduction sont configurables (en mode noyau)
- MMU s'occupe aussi de vérifier la légalité des accès mémoire
  Faute CPU si accès à une adresse mémoire logique non valide (selon les paramètres configurés)

Matériel

- Accès direct du matériel (DMA) reste en adresses physiques

Système d'exploitation et processus

Chaque processus

- A des paramètres de traduction mémoire spécifiques
- C'est sa « vue » personnelle de sa mémoire
- Son espace d'adressage logique est **automatiquement** (MMU) associé à des morceaux de mémoire physique (ou à des fautes CPU)

Changements de contexte

- Le système d'exploitation reconfigure le processeur
- Et paramètre la traduction à celle du processus actif

Changements de contexte, en pratique

- Quelques registres à mettre à jour (voire un seul, CR3 chez x86)
- Coût non négligeable sur les CPU modernes (on y reviendra)

Méthodes de traduction

- Chaque architecture matérielle est différente et spécifique
- Plusieurs possibilités en fonction des microprocesseurs
- Certains microprocesseurs offrent ou combinent plusieurs approches

$\rightarrow$ Nombreux détails techniques

Base et limite (historique)

- Technique historique, très simple et très limitée
- Deux registres spéciaux privilégiés : `base` et `limite`
- Traduction : `physique` = `logique` + `base`
- Vérification : `logique` < `limite`

Fonctionnalités

- Chaque processus a son bloc de mémoire qui part de l'adresse 0 pour lui
- Le système d'exploitation peut redimensionner ou déplacer les blocs de façon relativement transparente

Limites

- Mémoire relativement contiguë (sinon gaspillage)
- Pas de partage mémoire entre processus
- Pas de droits fins

Segmentation (historique)

- Technique historique, complexe, et limitée
- Le CPU permet d'avoir plusieurs segments paramétrés indépendamment
- Un segment $\approx$ un bloc base-limite $+$ droits spécifiques
- Résout les limites du base-limite
- N'existe plus en x86-64
  Sauf via deux registres spéciaux `FS` et `GS`

Pagination

- La solution à tous les problèmes ?

610 Pagination

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Pagination

Principe

- Découper toute la **mémoire physique**
  En **page physique** (ou **cadres**, *frame*, *page frame*) de taille fixe
  `sysconf(_SC_PAGESIZE)` donne la taille des pages du système
  (4ko par exemple)
- Découper tout l'**espace d'adressage** des processus
  En **pages logiques** (ou page virtuelle) de même taille
- Associer **efficacement** (CPU) les pages logiques aux physiques

Le gagnant actuel

- Offert par la plupart des processeurs
- Utilisé par la plupart des systèmes d'exploitation
- Assez couteux et complexe côté processeur (INF4170)
- Simple, souple et puissant côté système d'exploitation

Pagination pour le MMU

Adresse logique décomposée

- Numéro de page logique
- Adresse dans la page (décalage ou offset)
- Exemple: page de 4ko, 48 bits d'adresse logique =
  36 bits (numéro de page logique) + 12 bits (décalage (2^{12}=4k))

Traduction avec une table

- Associer numéro de page logique $\rightarrow$ numéro de page physique
- Les associations sont stockées dans la **table des pages**

Exemple très naïf

Pages

- Taille: 4 octets
- Taille du décalage (en bits):

Physique

- Adresse: 5 bits
- Espace d'adressage (octets):
- Nombre de pages:

Logique

- Adresse: 4 bits
- Espace d'adressage (octets):
- Nombre de pages:

Exemple très naïf (suite)

- Page: 4 octets. Adresse physique: 5 bits. Adresse logique: 4 bits

Pagination pour le système d'exploitation

- Une table des pages par processus
- Le système d'exploitation
  - Configure et maintient chaque table des pages
  - Positionne la table du processus actif lors des changements de contextes

Chez Linux

- `/proc/PID/pagemap` (tableau binaire) pour chaque page logique, quelle page physique est associée (+ info supplémentaire)
- `/proc/kpagecount` (tableau binaire) pour chaque page physique, combien de pages logiques y sont associées

En vrai

- La mémoire des processus est **beaucoup** plus riche que ce que le processeur offre
- La table des pages du MMU est trop bas niveau, trop spécifique et trop limitée
- Des structures de données additionnelles sont nécessaires
- Le système gère des zones virtuelles regroupant plusieurs pages: les lignes de `pmap(1)`
- On y reviendra...

Table des pages

Où est la table ?

- Registres ? Non, la table est trop grande !
- Un gros bloc en mémoire ? Où est ce bloc ?

Solution habituelle

- Registre privilégié pour l'adresse de la table (CR3 chez x86)
- Tables d'indirection en RAM

Questions

- L'adresse dans CR3 est-elle logique ou physique ?
- Un processus peut-il modifier la valeur du registre CR3 ?
- Un processus peut-il modifier la table des pages ?

Avantage de la pagination

Souplesse maximale

- Permet de mettre différents morceaux de mémoire
- Permet d'utiliser tout l'espace d'adressage (ou presque)
- Indépendant du nombre et de l'utilisation des morceaux
- Possibilité d'avoir des droits fins (lecture, écriture, exécution)
- Possibilité de partager des pages physiques entre processus
  - Pas forcément avec la même page logique
  - Pas forcément avec les mêmes droits
  - On y reviendra...

Question

- Une page physique peut-elle être associée à plusieurs pages logiques différentes ?

Trop couteux en espace !

- Une seule table d'indirection ne passe pas à l'échelle

Exemple

- 48 bits d'adressage logique
  - 36 bits de numéro de page logique
  - 12 bits de décalage
- 8 octets par entrée de la table
  - Adresse de base de la page physique
  - Métadonnées (droits de la page, etc.)
  - Bits réservés
- Taille de la table des pages ?

Trop couteux en espace !

- Une seule table d'indirection ne passe pas à l'échelle

Exemple

- 48 bits d'adressage logique
  - 36 bits de numéro de page logique
  - 12 bits de décalage
- 8 octets par entrée de la table
  - Adresse de base de la page physique
  - Métadonnées (droits de la page, etc.)
  - Bits réservés
- Taille de la table des pages ?
  - 2^{36} * 8o =
  - 512Go par table =
  - 512Go par processus (oups!)

Pagination multi-niveaux

- Découper l'adresse logique en plusieurs morceaux
- L'adresse d'une table + un morceau donne un champ dans la table
- Chaque champ d'une table indique
  - Soit l'adresse de la table suivante à consulter
  - Soit qu'il n'y a pas de table suivante: faute CPU

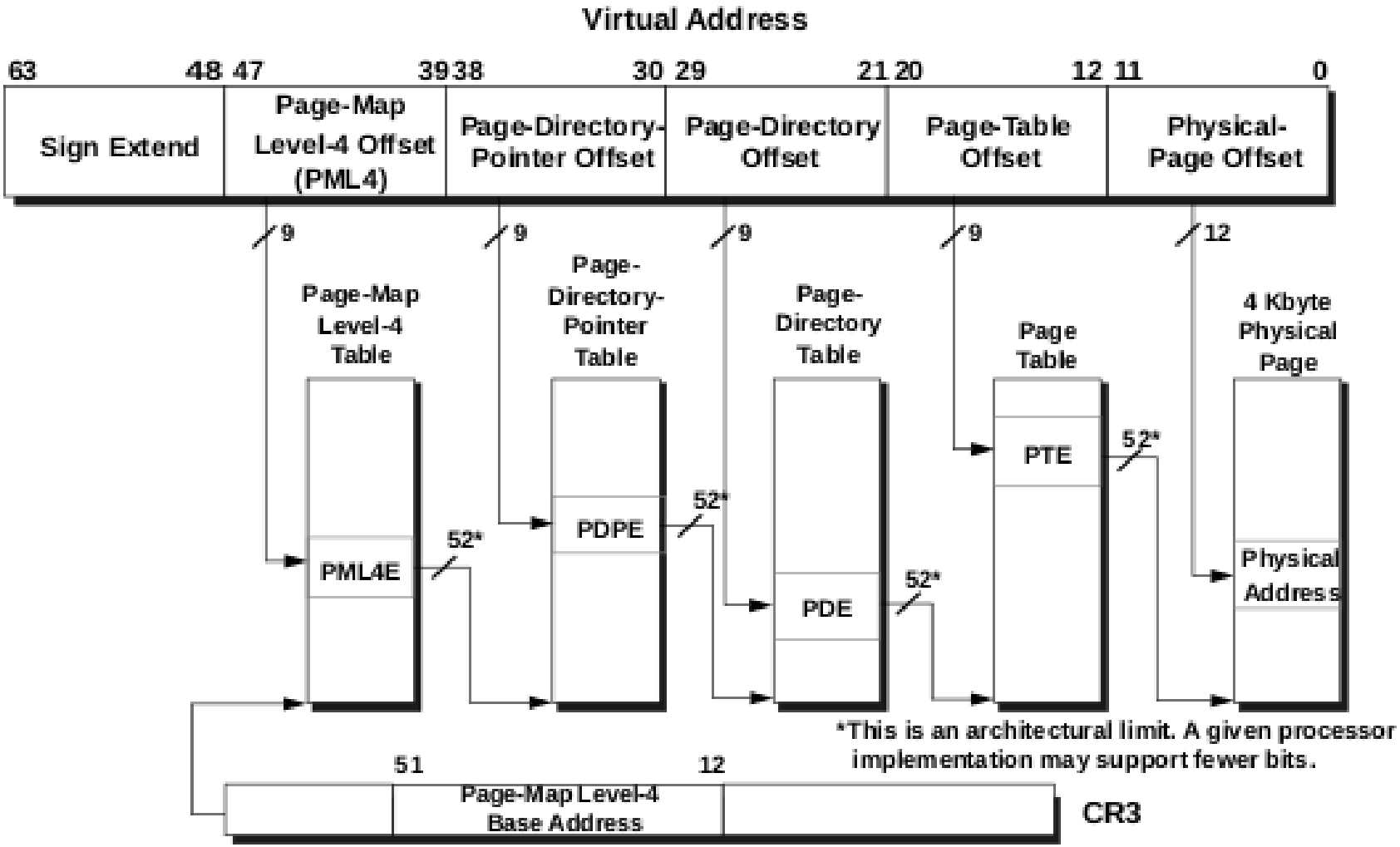


Figure 5-17. 4-Kbyte Page Translation—Long Mode

Source: AMD64 Architecture Programmer's Manual, Volume 2 System Programmation, page 136, rev. 3.36, octobre 2020, AMD Corporation.

Multi-niveau : pourquoi on y gagne ?

- L'espace d'adressage des processus est plein de vide
- Ne remplir que les tables intermédiaires nécessaires

$\longrightarrow$ Élagage de l'arbre des tables de pages

Dans la vraie vie

Beaucoup de détails techniques (et historiques)

Plusieurs schémas possibles, et configurables

- x86-64: souvent 48bits d'adressage sur 4 niveaux (mode long 4k)
- 57bits d'adressage sur 5 niveaux chez de récents processeurs Intel

Taille des pages variable

- Plusieurs tailles et schémas peuvent cohabiter en même temps
- x86-64: 4ko, 2Mo, 1Go

Autres fonctionnalités

- Peut se combiner avec la segmentation (x86)
  - Adresse logique $\rightarrow$ adresse linéaire $\rightarrow$ adresse physique
- Métadonnées supplémentaires (on y reviendra)...

Trop couteux en temps !

- Accéder à la mémoire coute trop d'accès mémoire

Exemple

- 4 niveaux : 4 tables + RAM finale
- Un accès mémoire logique = 5 accès mémoire physique
  - Chercher dans 4 tables + la donnée finale
  - Plus 5 additions, 4 vérifications des droits, etc.
- Les performances sont divisées par 5
- Et ceci pour chaque accès à la mémoire

TLB et caches processeurs

TLB (*translation lookaside buffer*)

- Cache les dernières traductions logiques $\rightarrow$ physiques
- Cas idéal fréquent : 0 accès mémoire pour traduire
- Cas pas idéal rare: faire toutes les indirections nécessaires

Caches CPU

- Cache le contenu de la RAM
- Évite l'accès à la RAM complètement

Les détails dans un autre cours

- INF4170 - Architecture des ordinateurs

620 Mémoire virtuelle

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Mémoire virtuelle

Aller plus loin ?

- Offrir à chaque processus une mémoire plus grande que celle disponible
- Utiliser le disque comme mémoire supplémentaire
- Un processus n'a pas forcément besoin d'être entièrement en mémoire principale

$\rightarrow$ De façon transparente pour les processus

Partitions et fichiers d'échanges

- Alias: *swap*
- Fichier ou partition dédiés
- Utilisée comme mémoire supplémentaire
- Accès **lent**, donc à utiliser correctement

Mémoire virtuelle sans pagination (historique)

- Alias: *swapping* de processus

Quand la mémoire est faible

- Trouver un processus pas souvent actif
- Copier toute sa mémoire sur disque (*swap out*)
- Puis libérer la mémoire du processus

$\rightarrow$ La mémoire n'est plus faible !

Quand on doit continuer l'exécution du processus

- On recharge le processus en mémoire (*swap in*)
- Quitte à *swap out* un autre processus pour faire de la place

Mémoire virtuelle et pagination

Paging (*swapping* de page)

- Une page virtuelle peut être
  - Soit en mémoire physique
  - Soit sur le disque (en *swap*)
  - Soit invalide
- Si RAM est pleine: on sauve
  - On **descend** des pages physiques vers le disque (*page out*)
- Si accès à une page virtuelle qui est sur le disque: on charge
  - On **monte** une page physique depuis le disque (*page in*)

Avantages

- Granularité beaucoup plus fine que le *swapping* de processus
- Chargement et déchargement de morceaux de processus au besoin

$\rightarrow$ On y reviendra (fonctionnalités avancées)...

Mémoire résidente vs. mémoire virtuelle

Mémoire virtuelle

- Les pages virtuelles de l'espace mémoire utilisable d'un processus
- Code + données + pile + tas + bibliothèques + etc.

$\rightarrow$ Ce qui apparait avec `pmap(1)`

Mémoire résidente

- Les pages d'un processus physiquement en RAM
- Transparent pour les processus, géré par le noyau
- Habituellement, une page physique est comptée une fois même si associée à plusieurs pages logiques

Question

Qu'est-ce qui peut être plus grand que la taille de la RAM ?

- La taille de la mémoire virtuelle d'un processus
- La taille de la mémoire résidente d'un processus
- La somme des tailles de la mémoire résidente des processus

Mise en œuvre

Côté MMU : on ne change rien

- Pas besoin de changer de processeur
- Tout se fait côté système d'exploitation

Pagination cotée MMU (rappel)

- La table des pages (MMU) indique seulement
  - Si une page logique existe
  - Et si oui : où (quelle page physique) et avec quels droits
- Une faute CPU est lancée
  - Si le CPU accède à une page logique absente
  - Si le CPU accède à une page logique avec les mauvais droits

Côté système d'exploitation

Migration d'une page sur disque

- Quand le système d'exploitation migre une page
- Il marque que la page est en swap (et où)
  - Ça ne rentre pas dans la table des pages
  - Le système a ses propres structures de données
- Il met à jour la table des pages pour invalider la page logique

$\rightarrow$ Coût: copie sur disque et mise à jour de la table des pages

Côté système d'exploitation

Accès du processus à une page virtuelle en RAM

- MMU traduit correctement adresse logique en adresse physique
- Le CPU travaille normalement (rien de spécial)

$\rightarrow$ Surcoût: 0

Accès du processus à une page virtuelle invalide

- MMU lève une faute CPU (faute de page)
- Le système d'exploitation
  - Attrape l'interruption matérielle
  - Détermine que la page virtuelle est invalide
  - Envoie SIGSEV au processus
- Le processus est terminé (ou gère le signal)

$\rightarrow$ Surcoût: une vérification en plus

Accès du processus à une page virtuelle en swap

- MMU lève une faute CPU (faute de page)

Le système d'exploitation

- Attrape l'interruption matérielle
- Détermine que la page virtuelle est en fait en swap
- Lance le chargement dans une page physique
- (et éventuellement la migration d'une autre page si pas de place...)
- Passe le processus à bloqué (et appelle l'ordonnanceur)

Quand le chargement est fini, le système d'exploitation

- Met à jour la table des pages
- Passe le processus à prêt (et appelle l'ordonnanceur)

Lorsqu'élu par l'ordonnanceur, le processus

- Recommence l'instruction fautive
- Qui réussit (cette fois)

Défaut de page

Défaut **majeur** de page

- L'adresse virtuelle est valide
- Mais la page n'est pas en mémoire : elle est sur disque
- Il faut faire des entrées-sorties pour la récupérer
- Métrique `%F` de `time(1)`

→ le système charge la page en mémoire (couteux)

Défaut **mineur** de page

- L'adresse virtuelle est valide
- Or page physique est en mémoire (cache ou chance)
- Mais n'est pas associée dans la table des pages
- Métrique `%R` (*recoverable*) de `time(1)`

→ le système met juste à jour la table des pages (peu couteux)

Algorithmes de remplacement

Problème bien étudié et bien généralisable (swap, cache, etc.)

Données

- Un grand nombre de pages virtuelles
- Une séquence de demandes de pages virtuelles
- Un nombre limité de pages physiques

Objectif

- Trouver à chaque demande quelle page physique utiliser
- Déterminer quelle page migrer quand la mémoire est pleine
- Minimiser le nombre de défauts de pages (et de migration)

Idées de base : quelles pages migrer ?

- Idéal : Les pages non utilisées dans un futur proche
- Approximation : Les pages non utilisées récemment
- Approximation pire : Les pages anciennement allouées

Algo naïf : file d'attente (FIFO)

Principe

- Les pages vieilles migrent en swap

Exercice

- Séquence: 1,2,3,4,1,2,5,1,2,3,4,5
- Avec 3, puis 4, pages physiques

Algo naïf : file d'attente (FIFO)

Principe

- Les pages vieilles migrent en swap

Exercice

- Séquence: 1,2,3,4,1,2,5,1,2,3,4,5
- Avec 3, puis 4, pages physiques

Anomalie de Belady (curiosité théorique)

- Avec plus de pages physiques (plus de RAM)
- Le nombre de fautes ne décroit pas forcément
- Dans certaines situations, le nombre de fautes peut augmenter

Algo de l'horloge (ou de la seconde chance)

Comme FIFO, mais un bit indique s'il y a eu utilisation

- Un bit marque les pages utilisées (MMU)
- Parcours circulaire des pages candidates
- Si son bit $= 1$, on le passe à 0 sinon on migre la page en swap

Exercice

- 1,2,3,4,1,2,5,1,2,3,4,5 avec 3 et 4 pages physiques

621 Mémoire virtuelle avancée

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Mémoire virtuelle

Aller plus loin ?

- Allouer, initialiser, charger, copier la mémoire efficacement
- Offrir des services aux processus

Optimisation

- Associer pages logiques et physiques paresseusement
- Mise à zéro paresseuse de la mémoire
- Partager les pages à outrance
- Charger les fichiers paresseusement

$\rightarrow$ De façon transparente pour les processus

Services aux processus

- Allocation de mémoire
- Projection de fichiers en mémoire (`mmap`)
- Communication par mémoire partagée
- Configuration de politiques (et d'heuristiques)

Zones mémoires virtuelles des processus

- Alias: région mémoire virtuelle, *virtual memory area*, ou *mapping*

→ Les détails au fur et à mesure

Concept du système d'exploitation

- Ignoré et inconnu du processeur (et MMU)
- Existe pour des raisons de gestion (et d'implémentation)
- Permet de mieux organiser l'espace mémoire des processus

Regroupe des pages virtuelles

- En morceaux cohérents
- Correspondent aux lignes de `pmap(1)` et de `/proc/PID/maps`

→ Ça permet de pas forcement gérer chaque page à part

Autre mémoire consommée des processus

- Table des pages (celle utilisée par le MMU)
- Structures de gestion (table des processus, des descripteurs, etc.)

→ Géré à l'interne par le système d'exploitation

Mémoire résidente vs. virtuelle (le retour)

- Swap + allocation paresseuse + chargement paresseux

Mémoire virtuelle

- Les pages virtuelles dans l'espace d'adressage
- Colonne `VSZ` de `ps(1)`
- Colonne `VIRT` de `top(1)`
- `VmSize`, etc. de `/proc/PID/status`
- Colonne `Size` de `pmap(1)`

Mémoire résidente

- Les pages physiques réellement en RAM
- Colonne `RSS` et `%MEM` de `ps(1)`
- Colonne `RES` et `%MEM` de `top(1)`
- `VmRSS`, etc. de `/proc/PID/status` (Linux)
- Colonne `Rss` de `pmap(1)`
- Note: rss = *resident set size*

Copie sur écriture (COW, *copy-on-write*)

- Faire la copie paresseuse de pages mémoire
- Exemple d'utilité : rendre `fork(2)` très efficace

Stratégie

- Lors d'une demande de copie de page
- On copie rien, on utilise juste deux fois la même page physique
- On ne fait une copie de la page seulement au premier accès en écriture

Mise en œuvre COW

Lors d'une copie, on met a jour la table des pages

- La nouvelle page logique pointe la page physique originale
- On enlève les droits en écriture de l'ancienne page logique et de la nouvelle page logique

$\rightarrow$ Cout: mise à jour de la table des pages

Lors d'un accès en lecture à la page logique

- Tout se passe normalement

$\rightarrow$ Cout: 0

Mise en œuvre COW

Lors d'un accès en écriture à la page logique

- Le MMU lève une faute CPU
- Le système attrape l'interruption, puis
- Copie la page physique dans une nouvelle page physique
- Associe la page logique à la nouvelle page physique
- Positionne les droits en écriture
- Redonne la main au processus
  Qui recommence l'instruction (et réussit cette fois)
- Pas besoin de passer à bloqué: c'est un défaut de page mineur

$\rightarrow$ Cout: copie d'une seule page et mise à jour de la table des pages

Questions

- Pourquoi ne pas passer à bloqué ?
- Comment distinguer un COW d'une vraie page en lecture seule ?

Zone privée vs. partagée

Zone partagée (*shared*)

- Différents processus utilisent les mêmes pages partagées
- Si la zone est écrivable, les modifications sont vues par tous
- Une zone partagée peut être utilisée par un seul processus

Zone privée (*private*)

- Différents processus utilisent des pages privées personnelles et des pages partagées communes (en lecture seule)
- Quand une page privée est écrite : les modifications sont vues que par le processus
- Quand une page commune est écrite : copie sur écriture

Question

- Qu'est-ce que ça change pour les zones en lecture seule ?

Partage des pages sous Linux

- `pmap(1)` et `/prog/PID/maps` affiche `s` pour les zones partagées et - (ou `p`) pour les zones privées
- Colonne `SHR` (*shared*) de `top(1)`:
  somme des taille des pages partagées
- Colonne `Pss` (*proportional set size*) de `pmap(1)`:
  chaque page divisée par le nombre d'utilisateurs
- On y reviendra (détail politiques, API, etc.)

622 mmap et cie.

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Chargement (*paging*) à la demande

Au démarrage des programmes

- Il faut charger des fichiers (exécutables, bibliothèques, etc.)
- Qui sont possiblement gros
- Et possiblement pas nécessaire *tout de suite* (voire jamais)

$\rightarrow$ Charger **au besoin** les **morceaux** de fichiers **nécessaires**

Si on est très paresseux

- `execve(2)` ne charge rien du tout
- On attend les vrais accès à la mémoire

Stratégie niveau système d'exploitation

Une page virtuelle peut être

- Soit en mémoire physique (RAM)
- Soit sur le disque : en *swap*
- Soit sur le disque : morceau de fichier pas encore chargé

Lorsqu'on accède à une page qui est sur le disque

- On la charge en RAM (*page in*)
- Depuis la *swap* ou depuis le fichier

Lorsque la RAM est pleine

- Si morceau de fichier : facile, il est déjà sur le disque
- Si page anonyme (pas de fichier) : on utilise la *swap* (*page out*)
- Éventuellement on écrit les données sur disque si besoin

Projection de fichiers en mémoire

Idée : fournir aux processus le chargement à la demande

- `mmap(2)` associe une zone mémoire (virtuelle) à un morceau de fichier
- `msync(2)` demander l'écriture des changements dans le fichier
- `mprotect(2)` change les droits d'une zone mémoire
- `munmap(2)` libère la zone mémoire

Beaucoup d'options possibles

- Quel bout du fichier est demandé? `offset` et `length`
- Fichier lu en entier ou à la demande? `MAP_POPULATE`
- Quels droits appliquer? `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`
- Les modifications en mémoire sont-elles écrites dans le fichier? `MAP_SHARED`, `MAP_PRIVATE`
- La zone mémoire est-elle copiée ou partagée lors d'un `fork(2)`? `MAP_SHARED`, `MAP_PRIVATE`
- Etc.

mmap.c

```c
#include "machins.h"
int main(int argc, char **argv) {
  int fd = open("compteur", O_RDWR|O_CREAT, 0666);
  if (fd==-1) { perror(NULL); return 1; }
  struct stat stat;
  fstat(fd, &stat);
  if (stat.st_size < 11) // version initiale si besoin
    stat.st_size = write(fd, "0000000000\n", 11);

  char* buf = mmap(NULL, stat.st_size,
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd); // plus besoin du descripteur

  int num = atoi(buf) + 1; // on « lit » directement du fichier
  sprintf(buf, "%010d\n", num); // et écrit aussi
  printf("%d\n", num); // affichage

  if (argc>1) pause(); // pause si argument
  munmap(buf, stat.st_size);
  return 0;
}
```

Efficacité des fichiers projetés

Accès très efficace

- Simple accès processeur $\leftrightarrow$ mémoire
- Pas d'appel système `read(2)`, `write(2)`
- Pas de copie noyau $\leftrightarrow$ processus
- Possibilité d'accès direct au matériel:
  Mémoire vidéo, *ringbuffer*, etc.

Table des inodes en mémoire (le retour)

- Les zones mémoires projetées
- Les données des fichiers ouverts ou en caches
- Peuvent utiliser les mêmes pages physiques que les fichiers projetés

$\rightarrow$ Pas besoin de dupliquer ou de synchroniser des zones mémoire

Inconvénients des fichiers projetés POSIX

- Pas de gestion simple de la taille une fois un ficher projeté
- Pas de `ftruncate(2)` ou d'ajout à la fin du fichier
- Pas de gestion simple des erreurs d'entrée-sortie
- Plus lent que read/write classiques dans certains cas
- Pas pour certains fichiers non réguliers
  Ni pour certains systèmes de fichiers
- Problématiques si disque réseau (NFS)

Allocation de zones anonymes

Idée : *mapper* des zones mémoire sans fichier associé

- `mmap(2)` avec indicateur `MAP_ANONYMOUS`
- Abus de langage (et d'appel système)
  Projection de fichier faite sans fichier !
- Mais on profite de l'expressivité de `mmap(2)`

Communication par mémoire partagée anonyme

- Sera hérité et partagé via `fork(2)`
- Permet la communication interprocessus extrêmement efficace
- Attention, sera perdu via `execve(2)`
- Attention aux situations de compétition

Communication par mémoire partagée nommée

API POSIX

- `shm_open(3)` créer ou ouvrir un « objet » mémoire partagé
- `shm_unlink(3)` supprime l'objet mémoire partagée
- `shm_overview(7)` pour les détails
- On l'utilise ensuite comme un fichier ouvert
- On peut aussi bien évidemment le « mmaper »
- Persistant jusqu'à l'arrêt du système ou une libération explicite

API POSIX sous Linux

- C'est en fait des fichiers « normaux »
- Dans un système de fichier mémoire `tmpfs`
- Monté habituellement dans `/dev/shm`

API System V

- `shmget(2)`, `shmat(2)`, `shmdt(2)`, `shmctl(2)`
- Vieille API plus complexe et bas niveau

shm.c

```c
#include "machins.h"
int main(int argc, char **argv) {
  int fd = shm_open("autre_compteur", O_RDWR|O_CREAT, 0666);
  if (fd==-1) { perror(NULL); return 1; }
  struct stat stat;
  fstat(fd, &stat);
  if (stat.st_size < 11) // version initiale si besoin
    stat.st_size = write(fd, "0000000000\n", 11);

  char* buf = mmap(NULL, stat.st_size,
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd); // plus besoin du descripteur

  int num = atoi(buf) + 1; // on « lit » directement du fichier
  sprintf(buf, "%010d\n", num); // et écrit aussi
  printf("%d\n", num); // affichage

  if (argc>1) pause(); // pause si argument
  munmap(buf, stat.st_size);
  return 0;
}
```

Mémoire côté processus chez GNU/Linux

Chargement des bibliothèques

- Est fait par le processus (pas par le noyau)
- `ld.so` utilise `mmap(2)` pour charger les morceaux de bibliothèques

→ Les détails une autre fois

Allocation de mémoire

- `malloc(3)` gère et alloue la mémoire du processus
- `brk(2)` pour agrandir (ou réduire) la zone du tas
- `mmap(2)` anonyme pour de grosses allocations

→ Les détails une autre fois

Pile pthread

- `pthread_create(3)` gère et alloue le thread
- `mmap(2)` pour crée un pile neuve (qui grandit à l'envers)
- `clone(2)` pour créer la tache
- Le tas pour les structures internes

Appels système supplémentaires

- `mcore(2)` (Linux) indique quelles pages sont en mémoire
- `madvise(2)` (Linux) indique l'utilisation future de zones mémoires
  - Peut changer des comportements
  - Peut changer les stratégies internes d'optimisations
- `posix_madvise(2)` version POSIX du précédent
- `mlock(2)` force une zone mémoire à rester résidente

623 Consommation mémoire

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Allocation paresseuse aux processus

Les processus

Ont tendance à allouer de la mémoire

- Qu'ils n'utilisent pas entièrement
- Qu'ils n'utilisent pas de suite
- Qu'ils n'utilisent pas du tout

Le système d'exploitation peut promettre

- Reçoit les demandes des processus
- Répond positivement et alloue des pages virtuelles
- N'associe pas encore de page physique

Puis livrer plus tard

- C'est seulement lors du premier accès à la mémoire
- Que le système d'exploitation associera une page physique

Mise en œuvre de l'allocation paresseuse

- La zone mémoire virtuelle est créée (ou agrandie)
- Les pages logiques sont laissées invalides dans la table des pages

Au premier accès

- MMU lève une faute
- Système d'exploitation
  - Attrape l'interruption
  - Alloue une page physique
  - Met à jour la table des pages
  - Redonne la main au processus
- Processus réussit son instruction (cette fois)

$\rightarrow$ Défaut de page mineur

Page « zéro » (Linux)

- La « page zero » est une page physique globale et unique
- En lecture seule
- Ne contient que des 0
- Associée aux pages nouvellement allouées aux processus
- Les processus peuvent déjà y lire des zéros
- Nettoyage de mémoire « gratuit »
- Lors de la première écriture,
  On alloue une page physique privée (COW)

Questions

- Est-ce que la mise à zéro est vraiment gratuite ?

zero.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
  size_t size = 3l << 30;
  size_t page = sysconf(_SC_PAGESIZE);
  long cpt = 0, opt;

  char *ptr = malloc(size);
  if (ptr==NULL) { perror("malloc"); return 1; }

  while((opt = getopt(argc, argv, "rwp")) != -1) {
    if (opt == 'r')
      for(size_t i=0; i<size; i += page)
        cpt += ptr[i];
    if (opt == 'w')
      for(size_t i=0; i<size; i += page)
        ptr[i] = 1;
    if (opt == 'p') pause();
  }

  free(ptr);
  return cpt;
}
```

Mémoire infinie

Thrashing

- Presque plus de place en RAM
- Le système d'exploitation passe son temps à sauver et charger des pages du disque
- Dégradation importante des performances et de l'interactivité

OOM (*Out of memory*)

- Situation où il n'y a plus de mémoire: ni RAM ni en swap
- Le système d'exploitation en a besoin maintenant
- Ou en avait promis aux processus qui y accèdent

La mémoire ne peut être « reprise » aux processus

- Car le `malloc(3)`/`mmap(2)`/`brk(2)` historique a réussi
- Donc les pages existent dans l'espace virtuel des processus
- Les reprendre causerait un dysfonctionnement

Solution : extermination

OOM killer (Linux)

- Détermine quels processus terminer de force
- Objectif: permettre au système d'exploitation de survivre

Terminer oui, mais qui ?

- Nombreuses heuristiques pour terminer le « meilleur » processus
- Idéalement le processus responsable du OOM
- Parfois se trompe de processus (oups!)

Configuration OOM

- `/proc/PID/oom_score_adj` : bonus/malus (configurable)
- `/proc/PID/oom_score`: score actuel du processus (si OOM maintenant)
- `choom(1)` affiche et configure le score

Surengagement (*overcommitment*)

- « Prêter de la mémoire qu'on a pas »
- C'est pas toujours une bonne idée
- Une des causes principales des OOM

Linux

- `/proc/sys/vm/overcommit_memory` change la politique
  - 0 (défaut) = refuse seulement les demandes absurdes
  - 1 = accepte toutes les demandes d'allocation
  - 2 = pas de surengagement au-delà d'une certaine limite
- `/proc/sys/vm/overcommit_ratio` ou `/proc/sys/vm/overcommit_kbytes` Configure la limite si le mode est 2 (+50% par défaut)
- `CommitLimit` et `Committed_AS` de `/proc/meminfo`

Récapitulatif : la mémoire c'est compliquée

Plein de combinaisons pour chaque page virtuelle

- Privé vs. partagé
- Initialisé vs. non-initialisé
- Fichier projeté vs. anonyme
- Résidente en mémoire, ou sur disque
- Sale (*dirty*) = fichier à mettre à jour sur le disque, ou non
- Etc.

Plus

- Les structures supplémentaires (table des pages, etc.)

Conclusion

- La mémoire c'est compliquée
- « Compter » la mémoire utilisée ou libre
  C'est compliqué aussi