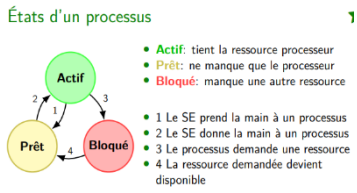
Définition SE • Couche logicielle située entre le matériel et les applications d'un ordinateur • A l'exclusivité des mécanismes matériels (mode noyau) |**Objectifs** • À abstraire la couche matérielle (appels système) • À répartir équitablement les ressources entre les différents processus et utilisateurs • À protéger le matériel, les processus et les utilisateurs les uns des autres|**Déf processus** : • Un programme en exécution • Par un processeur|**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|**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 → le processeur refuse physiquement d'exécuter l'instruction si le mode n'est pas le bon| C'est le processus ou le processeur qui est en boucle infinie? **Ça dépend de la définition de boucle infinie**. |Où est stockée la table des processus? **Dans la mémoire de l'OS. L'espace du noyau.** | • **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 |Est-ce que /proc contourne les appels système? Oui | **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 → ps, top, etc. utilisent directement /proc à l'interne

Thread vs. 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 difficile • Un bogue dans un thread corrompt les autres • Un thread compromis, compromet les autres → Les navigateurs web modernes sont passé d'un thread par onglet à un processus par onglet|**Modèles d'implémentation multithread**|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 → 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

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 → Peut corrompre ses propres données en mémoire (bogues) → Mais ne peut corrompre les autres processus|**Segments mémoires** : 4 segments principaux • 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)



Fonctionnement de la pile: 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**: Allocation (et désallocation) dynamique • Mémoire réservée quand elle est nécessaire • Et libérée quand elle ne l'est plus → Contient les données importantes des vrais programmes|Gestion programmative : Le programme décide des allocations et des désallocations|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 •Un processus peut-il passer de prêt à bloqué ? **Non** • De bloqué à actif ? **Non**

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 details| • Pourquoi read bloque le processus ? **demande de ressources**|• Est-ce que read bloque toujours le processus appelant ? **Pas toujours (à moins d'une erreur)**|• Où le noyau conserve le décompte de l'utilisation des ressources des processus ? **Table de processus**|• Est-ce que %U peut être plus grand que %E ? **Si plusieurs CPUs, oui**.|• Est-ce que %S peut être plus grand que %U ? **Oui**|• Quelle est la valeur maximale de %P sur un système ? **C'est le nombre de coeurs pas système**|• Comment avoir une grande ou une petite valeur de chacun des indicateurs (toutes choses étant égales par ailleurs) ? **On minimise les appels de système**

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

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

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|**Fonction popen** • popen(3) tube avec une commande shell en arrière plan • En gros: pipe+fork+exec de sh -c • Attention à la sécurité |**Fonction posix_spawn** • posix_spawn(3) combine de fork et exec • Compliqué à utiliser| (**Forks**) Comment l'enfant reconnaît son parent ? Il peut demander à l'OS pour avoir ID du parent (getppid)|**Forkbomb** |Déni 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|**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!)|•Pourquoi killall nomcommande ne fonctionne pas directement ? **Ça va devenir séquentiel à un moment donné**

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

Fonctions pratiques appel système exec : 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** : • 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|**Contenu des exécutables binaires** : • Quels blocs d'octets charger ? • À quelle adresse dans la mémoire ? • Avec quels droits rwx ? • Quelle est la taille du BSS ? • Autres choses|Pourquoi # dans un shebang? **Cet instruction n'est pas destiné à l'interpréteur. I est destiné à l'OS**

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 |• 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 ? **On ne fait pas confiance aux programmeurs** • Quels sont les cas d'erreur d'exit ? **Aucun**• Pourquoi exit ne retourne pas de valeur ? **Quand on fait exit, le programme est finit. Le compteur ordinal n'a plus d'instructions**

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| • Que se passe-t-il si une fonction enregistrée par atexit appelle exit ? Ça va se réappeler plusieurs fois (pas définit)|**Les vrais appels systèmes (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

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)→ 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 subreader autre que init peuvent être définis

Ordonnanceur| • C'est une partie du SE • Il sert à déterminer quels processus sont actifs| **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) → Mais aussi si changement de priorité (ou d'ordonnanceur). **Concrètement ?** • Appel système • Interruption matérielle, dont l'horloge programmable • Ces deux cas couvrent-ils toutes les possibilités ? **Oui**| **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)) → Dans les deux cas, c'est à la demande du processus| Et si on laissait la main à un processus bloqué par une Entrée-Sortie ? **Complètement inutile de faire** | **Ordonnanceurs préemptifs**| À n'importe quel moment, on peut suspendre un processus. **Fin du tour** :• Expiration d'un quantum de temps alloué au processus → 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| Quels sont les avantages du non-préemptif sur le préemptif ? **Le non-préemptif est mieux pour les grands calculs**| **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)| **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 → 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

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| • En quoi savoir la catégorie aide l'ordonnanceur ? Permet de faire des décisions plus intélligentes• Peut-on catégoriser plus finement ? Oui, par un entier.

Statégies d'ordonnancement standard| **File d'attente** (non-préemptif)|FIFO| **Avantages** • Facile à comprendre : file d'attente à la caisse • Facile à implementer • Occasionellement équitable| **Implémentation** • Une file de processus prêts • La tête de file est le prochain élu • Les processus qui (re)deviennent prêts → en fin de file

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)| Et si macommande part en boucle infinie ? Elle va complétement prendre les ressources

Tourniquet |• File d'attente + quantum de temps • RR (Round-robin) • Quantum expiré → 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 | CPU-bound vs IO-bound, qui y gagne ? IO-bound |**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

Problème des algos précédents| • Des décisions sont prises • Mais indépendamment des caractéristiques des processus ou de leurs comportements → 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 → Pourquoi pas les deux ? **On veut laisser le choix** | **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 • comment éliminer la famine ? **Priorité accumlée ou met en place un timeout**

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)| **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)

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) → Responsabilité du système d'exploitation| **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 → le tout, efficacement |**Abstraction pour l'utilisateur** • Abstraction de la gestion de l'espace • Cohérente et indépendante → Fichiers (et répertoires) | • Y a-t-il des alternatives aux fichiers pour stocker des données ? **Oui, une BD** • Est-ce que le concept de fichier a tendance à être moins important de nos jours ? Oui (car intéraction minime) et non (car ça existe à la base)

Terme ambigu : fichier | • 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*

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).| Pourquoi cd est une commande interne du shell ? c'est une primitive du shell |**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)| Pourquoi c'est des commandes de l'administrateur **? Car on authorise seulement l'administrateur (root) à monter et démonter des choses pour éviter d'écraser ce qui est déjà là ou de démonter quelque chose d'important**

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 |**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|Il manque un truc, non ? **le nom (c'est sauvegardé dans la partie répertoire)** | **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

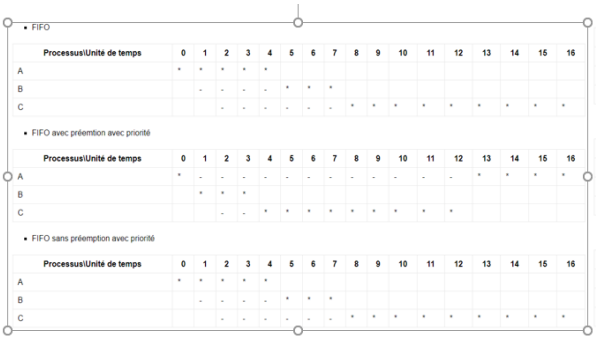
Liens symboliques| • Fichier spécial l (S_IFLNK) • Représente un autre ficher (via son chemin) • symlink(2) et ln -s (création), readlink(2) (lecture) • Documentation symlink(7) |Questions • Quelle est la taille d'un lien symbolique ? **Difficile à savoir, mais c'est petit** • Peut-on savoir si un fichier a des liens symboliques ? **Non** • Un fichier lié doit-il exister ? **Non** • Quels sont les droits pour suivre un lien symbolique ? **C'est les droits du chemin pointé** • Que faire en cas de cycle de liens symboliques ? **Le SE va éventuellement arrêter de suivre les liens** |**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)| **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)| • Il manque pas une date ? **la création** • Que se passe-t-il en 2038 ? **Rien (comme Y2K) à cause implémentation des nanosecondes**

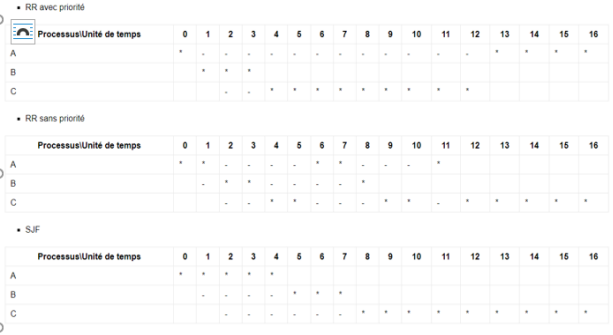
Droits et utilisateurs|**Utilisateurs (et groupes)**| • uid : numéro d'utilisateur • gid : numéro de groupe d'utilisateurs → Pour le système vous n'êtes que des numéros • uid == 0 : super-utilisateur (root)| **Noms des utilisateurs et groupes (Unix)**| Le noyau gère pas les noms des utilisateurs et des 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)| **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|Pourquoi ça peut être un problème de stocker seulement les numéros d'utilisateur et groupes dans le système de fichiers ? **Peut-être le numéro représente une autre information ailleure**

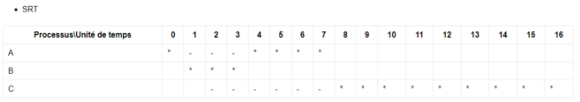
Droits traditionnels Unix| **3 catégories d'accès (ugo)**| • u (user/utilisateur) l'utilisateur propriétaire • g (group/groupe) le groupe propriétaire • o (other/autre) les autres |**3 permissions par catégorie (rwx)** |• r (read) : lire le contenu • w (write) : modifier le contenu • x (execute) : exécuter (si fichier) ou traverser (si répertoire) • chmod(1), chmod(2)| • Quels sont les droits nécessaires pour stat(2)? **Traverser le chemin**. chmod(2)? **Utilisateur**. suivre un lien symbolique? **Traverser le chemin**. supprimer un fichier? **write** • Quand sont vérifiés les droits? **Lors de l'appel système**

• Peut-on utiliser open(2) et read(2) pour lire les répertoires ? **pour o, oui et pour r, non** • Comment fonctionnent opendir(3) et readdir(3) en vrai ? opendir va juste ouvrir le répertoire, readdir va utiliser getdir(3)| **Chemin → inode en théorie**| 1 Découper le chemin en éléments $e1/e2/.../en$ |2 Partir du répertoire de base ik avec k = 0 (racine, courant, etc.) |3 Charger le contenu du répertoire ik depuis l'espace de donnée du disque |4 Chercher dedans l'élément suivant ek+1 |5 Charger l'inode associé ik+1 depuis la table des inodes |6 Vérifier que ik+1 est bien un répertoire, les droits, etc. |7 Si besoin, continuer en 3 avec $k = k + 1$|**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)

Manipulation des liens durs| **Création de liens durs** • ln(1) et link(2) • Pas de distinction entre l'original et le lien → 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)| • Comment la commande mv(1) sait déplacer entre systèmes de fichiers (alors que rename(2) ne sais pas faire) ? **Elle va juste le copier-coller et le supprimer** • Pourquoi il existe une commande cp(1) mais pas d'appel système de copie ? **Si je peux read and write, pas besoin d'appel système** • Comment supprimer tous les liens durs d'un fichier ? **Il faut tous les chercher manuellement** • Si on pouvait utiliser link(2) sur les répertoire, comment créer des répertoires détachés de la racine ? **C'est un paradoxe, on l'interdit**







• Le temps moyen d'exécution et temps moyen d'attente

Stratégie d'ordonnancement	Temps moyen d'exécution	Temps moyen d'attente
FIFO	(5+7+15)/3 = 9	(4+6)/3 = 3,3
FIFO avec préemption avec priorité	(17+3+11)/3 = 10,3	(12+2)/3 = 4,6
FIFO sans préemption avec priorité	(5+7+15)/3 = 9	(4+6)/3 = 3,3
RR avec priorité	(17+3+11)/3 = 10,3	(12+2)/3 = 4,6
RR sans priorité	(12+8+15)/3 = 11,6	(7+5+6)/3 = 6
SJF	(5+7+15)/3 = 9	(4+6)/3 = 3,3
SRT	(8+3+15)/3 = 8,6	(6+3)/3 = 3

État des processus

Processus\Unité de temps	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	actif	actif	bloqué	bloqué	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	actif	actif	actif	actif		
B		prêt	actif	actif	actif	actif	bloqué	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	prêt	actif	actif
C			prêt	prêt	prêt	prêt	actif	actif	actif	actif	actif	actif	actif	actif						

Processus	Temps d'entrée	Priorité	Tems de calcul
A	0	1	5 unite calc
B	1	5	3
C	2	2	9