340 Traitement fichiers ouverts|Descripteurs de fichiers|**Déf** • Dans un processus• Désigne un fichier ouvert• Sert à la manipulation• C'est un entier tout simple. **Trois descripteurs standard** • 0: entrée standard • 1: sortie standard • 2: sortie standard pour les messages d'erreur → conventions espace utilisateur : noyau s'en fout|Organisation interne : 3 niveaux|**TD. Tables des descripteurs (une par processus)**• Une entrée par descripteur• Pointe sur un fichier ouvert (→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 (→TIM∗) **TIM. Table des inodes en mémoire (globale)** • Une entrée par fichier (inode) distinct manipulé (ou en cache) • Synchronisée avec les inodes sur disque|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 inodes en mémoire → cache par fichier. **Le SE minimise accès disque : asynchronisme** • En lecture (readahead) et écriture (flush)• Demandes des utilisateurs ≠ 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| Table des fichiers ouverts| • Une entrée par open(2) (ou autre) effectué • Contient mode d'ouverture (lecture, écriture) • Contient curseur lecture-écriture (éventuel) dans fichier → Un même fichier peut être manipulé indépendamment par des processus • lseek(2) permet déplacer curseur lecture-écriture|Threads, fork, exec|**Pthreads partagent** • Descripteurs sont associés au processus → Donc partagés par threads. **Fork duplique (et partage)**• La table descripteurs est dupliquée • Entrées dans la table fichiers ouverts sont partagées• En particulier le curseur de position (lecture/écriture)• Compteurs table fichiers ouverts sont incrémentés• Pq pas incrémenter compteurs de TIM? *Pas fait un autre open*. **Exec préserve** • La table descripteurs et **préservée** → Permet préserver 0, 1 ou 2|Spécificités Linux |**Flag O_CLOEXEC** • O_CLOEXEC flag de open(2) (et autres appels S)• Descripteur sera automatiquement fermé lors execve(2) → Évite fuite descripteurs ou gaspillage ressources → Mais pas portable. **Tâches Linux** clone(2) permet de décider quoi partager ou cloner • CLONE_FILES table descripteurs • CLONE_FS des informations liées au S fichiers, don't chdir et umask| Duplication de descripteurs |**Descripteurs synonymes** • 2 descripteurs d'un même processus peuvent pointer même entrée dans table fichiers ouverts • Appels S dup2(2) (et dup(2)). **Quel est l'intérêt ?** • Redéfinir entrées et sorties standard • Redirection fichiers • Communication par tube. Différence entre dupliquer 1 descripteur et ouvrir 2 fois 1 fichier ? *On travaille pas sur même table. Ouvrir fichiers c des entrées qu'on ajoute*|Autre partage descripteurs ou fichiers|• Descripteur est 1 entier, partager 4 n'avance à rien → noyau doit être impliqué. **Via sockets Unix** • unix(7) • Partage fichiers ouverts • Via messages auxiliaires (SCM_RIGHTS). **Via /proc (Linux)** • /proc/PID/fd : proc(5) • Partage inodes en mémoire • Même fichiers supprimés et communication interprocessus

350 Implémentation S fichiers| **Types de systèmes de fichiers** • Nombreux types existent. **Nombreux, car spécifiques** • À des systèmes et/ou organisations, contrôle de l'évolution, mais aussi syndrome NIH • À des contraintes physiques des périphériques et ordinateurs • À des besoins spécifiques des utilisateurs. **Format de stockage** Spécifie la représentation des données sur disque • Champs de bits • Structures de données. **Implémentation** • Dans les SE • Dans les outils annexes (mkfs(8), fsck(8)) → Besoin maximal de fiabilité : ne pas manger les données | Découpage en blocs |• Blocs de taille fixe, configurable, ou variable • 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|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) • Limites d'implémentations : format peut stoker plus, mais logiciels peuvent pas lire • Combinaisons directes et indirectes de tout ça| Besoins et fonctionnalités | **De base** • Stocker données des fichiers • Stocker métadonnées • Stocker entrées des répertoires • Gérer l'espace libre (inodes et blocs). **Plus avancés** • Chiffrement et compression • Journalisation • Instantanés (snapshots) et branches • Déduplication • Multi-volumes, RAID • Somme de contrôle (checksum) • Correction d'erreurs | Allocation et adressage des fichiers | **Allocation contiguë** • Les blocs de données d'un fichier sont contigüès • 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 • Comment stocker gros fichiers ? *Indexage*|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|Journalisation|Problème : corruption • Panne lors d'une écriture → Données partiellement/mal écrites → 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 → Problème de coût : écriture plus chère (copie intermédiaire)| 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 → Plus efficace que la journalisation classique

400 Communication inter-processus|Chaque **processus** est autonome et vit isolé dans son propre espace mémoire. Trop restrictif : Besoin de collaboration, communication et de cooperation : Processus différents opèrent ensemble vers un même objectif | Formes de cooperation |**Interne :** Application conçue à la base multithreads et multiprocessus (ex : nav modernes). Objectifs : performance, asynchronisme, isolation. **Protocolaire :** Communiquer avec des applications qui respectent un protocole (ex : applications réseaux). **Des données** : Application manipule des données, produite (ou non) par d'autres applications (ex : ouvrir, enregistrer, tubes shell)|Modèles de communication |**Message passing :** Chacun les traite à sa façon, un seul processus a les données à la fois. **Shared memory** : Données communes, accès et modifications non exclusives. *Questions*. Mod com? Pour Tube shell « | »? *Sm*. 2 threads et une structure de données dans le tas? *Sm*. Fichiers dans ~/Documents? *Les 2*. BD? *Les 2* | S de fichiers | **fichiers par exclusifs à un processus** : processus peuvent utiliser fichiers pour communiquer entre eux et accès et la protection sont connus. Ex : Fichiers de données et autres documents (explicites à l'utilisateur) • Fichier temporaire pour passer des données (compilation)• Fichiers spéciaux pour initier un autre type de communication (tubes nommés) • Spool (impression, cron job) • Fichier projeté en mémoire (mmap(2))| Socket réseau| **Communication distante** : Objectif primaire -> faire communiquer applications sur ordinateurs distincts, peut aussi être utilisé pour applications sur même machine, majoritairement pour des clients-serveur. Avantages : Le SE peut optimiser l'efficacité du traitement (ex : client/serveur X (X(7))| Bus logiciel **Support communication haut niveau**. Majoritairement applicatif, permet de faire communiquer des applications via des objets partagés et des envois de messages, souvent pour réseau (ex. CORBA). **Gestion communication** : Protocole de communication: Moyens de communication (données, struct de données) les processus peuvent échanger et accéder, Primitives d'accès et de protection, Mécanismes de synchronisation. Le SE fournit appels S pour IPC, +- riches et complexes, règles spécifiques cas par cas. **Programmeur** utilise les primitives sys pour implémenter les protocoles et applications, Prend en compte limites et caractéristiques des IPC utilisées. Est libre d'implémenter tout ce qu'il veut. IPC Ss -> outils pour bâtir solution de coopération. **Bibliothèques et langages** abstraient les IPC S, Offrent fonctionnalités et protocoles clé en main, Ex: requête https en JavaScript, J2EE, Dbus

410 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 signaux : Les signaux sont catalogués, Liste est fixée, Chacun est documenté, → signal(7) **Gestionnaire de signaux par processus** : Chaque programme gère toutefois les signaux comme il veut, Sémantique doit être documentée dans le programme (man). En particulier si 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 S kill(2). **Kill** est souvent une commande interne du shell, pourquoi ? *Lorsque kill, il y a un processus trop gourmand, donc inutile de lancer un autre processus* | Actions possibles pour un signal | -> processus peut Accepter le comportement par défaut(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 détails), Pas de kill sur le processus du voisin| 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-null-terminaison d'un enfant. SIGNAL-VALEUR-ACTION-DESC-T-terminer, D=défaut obligatoire, M=image mémoire, I=ignorer| Gestion classique des signaux en deux étapes| **Écrire la fonction gérante** : 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| 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 l'exécution jusqu'au déblocage. sigprocmask(2) manipule 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. **exec**• Préserve: les signaux ignorés, bloqués et en attente• Vide: les gérantes|Interruption des appels S|1) Processus dans appel S, 2) Signal attrapé ; gérante invoquée ; gérante terminée (return), 3) Appel S terminé de force (interrompu) • Retourne EINTR (si pas commencé) • Sauf si SA_RESTART dans sa_flag de sigaction(2) • 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!).→ Attendre un signal avec poll(2), select(2)

420 Tubes| • Canal communication unidirectionnel avec 2 bouts • Octets écrit 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 • octets lus sont consommés • pipe(7) pour les détails. Processus A –Tube→ Processus B. **Descripteurs fichiers**• Pour processus, bout de tube est descripteur• Chaque extrémité se manipule comme fichier ouvertread(2), write(2), close(2), dup2(2), poll(2)• Mais pas lseek(2) (erreur ESPIPE). **Niveau noyau** • Espace mémoire du SE • L'espace et son accès sont gérés par le SE• Capacité limitée (64ko défaut Linux) • Pas un problème• Libéré automatiquement quand plus utilisé| 2 sortes de tubes: **Tubes simples** (maj. utilisés) • Création : appels pipe(2)• « Retourne » 2 descripteurs de fichiers• int fds[2]; pipe(fds)• fds[0] bout en lecture• fds[1] bout en écriture• Astuce mnémotechnique: 0=stdin 1=stdout. **Tubes nommés** • Création : mkfifo(1) et mkfifo(3)| **Communication par tube** • Tube est créé par processus • Mais global au S. **Partage de tube par fork** • Descripteurs de fichiers sont copiés • Bouts de tubes sont partagés. **Communications**• Entre parent et enfant (parent crée le tube, enfant hérite descripteurs) • Entre frères et sœurs (parent crée le tube et enfants héritent descripteurs)| Synchronisation |**Lecture** • Si données dans tube alors **read** lit maximum d'octets • Si tube vide - Si écrivain existe : read bloque - Si pas écrivain : read retourne 0 (fin tube) • Si écrivain écrit et lecteur alors Signal SIGPIPE envoyé (par défaut, termine processus) • Si 1+ lecteur- Si assez place: write écrit tous les octets - Si tube +-plein : 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). • Pourquoi c'est pas symétrique ? (0 lect. SIGPIPE) *La lecture implique en général un traitement après, alors que l'écriture implique souvent un disfonctionnement*. Bonnes pratiques -Un seul lecteur et un seul écrivain |Tubes nommés| **Limites des tubes simples** : Non hérité de processus• En créant tube d'avance → Communication entre processus indépendants difficile. **Principe des tubes nommés** • Tubes nommés par entrées du répertoire désignés • Donc plus besoin d'hériter descripteurs • Mais 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 pour rendez-vous entre processus|Tubes nommés|**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) 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 tube simple → Même synchronisation, même atomicité|Substitution de processus| **Principe** • Exécute CMD dans processus indépendant • Où sortie standard CMD est redirigée dans tube• Substitue l'argument (CMD) par chemin du tube. Quelqu'un ouvre ce chemin est connecté au tube. **2 implémentations**• Pseudo fichiers tubes (proc(5)) si dispo• Tube nommé sinon

430 Sockets | **Communication par sockets** • POSIX sockets alias BSD sockets alias Berkeley sockets• Pour communication réseau entre processus socket(7)• API offerte par le S d'exploitation. **Socket ?**• Point de communication abstrait• Boîte d'émission et réception → Socket est descripteur de fichier → responsabilité du S| API des sockets | **API commune** • Différents et nombreux protocoles • Différents types de communication • Y compris propriétaires ou désuets. **API générale** • Abstractions et appels S 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 Ss ont leur propre API de sockets • API souvent proche (concepts et vocabulaire), parfois meilleure|**Type 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 d'ordre, duplications. |Petite sélection d'appels S|**• 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• plus de 20 chez Linux, AF = address family. **Type = sémantique de la communication**• SOCK_STREAM: flux d'octets, connecté, fiable • SOCK_DGRAM: messages, non connecté, non fiable• 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 SE • Processus lisent/écrivent dans 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 • 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 S • 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| **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)|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. SOMAXCONN est bien. • accept(int fd, struct sockaddr *ad, socklen_t *adlen) Récupère ou attend prochain client - Retourne nouveau socket, connecté directement au client -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 | 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 de traitements courts seulement et 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 = accepte un nouveau client - socket d'un client bouge = traite sa demande •Problèmes : messages courts seulement, pas adapté aux cas compliqués|Modèles populaires de serveurs (2/2)|**Multithread** • thread principal écoute• Lance nouveau thread par client (ou pool threads)• Probl. : prog multithread. **Multiprocessus** • Processus principal écoute• Sous-processus (fork(2)) par client (ou pool processus)• Prob. : 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 SE • Ce qui est possible est spécifique à chaque domaine• recvmsg(2) et sendmsg(2) pour les utiliser• cmsg(3) pour y accéder• API horrible |Descripteurs fichiers auxiliaires |• Utilisable dans sockets du domaine Unix • SCM_RIGHTS passe des fichiers ouverts • L'émetteur attache descripteurs fichiers •

Le S crée descripteurs dans processus récepteur • Pas mêmes numéros de descripteur, mais mêmes fichiers ouverts|**Paire de sockets** • socketpair(2) crée 2 sockets connectés • Ressemble à pipe(2) , mais bidirectionnel ! • Messages
500 Synchronisation| **Ss concurrents** • Des éléments logiciels (voir matériels) • Sont capables de s'exécuter en « même temps » • Indépendamment du moment ou de l'ordre de leur exécution • Sans tout briser | 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 • Voir Ss distribués|Quel rapport avec les Ss d'exploitation ?| **Traitement d'événements logiciels et matériels** • Des évènements fondamentalement imprévisibles • Interruptions matérielles • Appels S de processus (si vrai parallélisme). **Performance des Ss d'exploitation** Exploitation des possibilités de concurrence et parallélisme • Traitements parallèles internes : threads S • Préemption S : les noyaux modernes sont préemptifs • Une approche « un seul processus en appel S 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 → Facile mais ça n'arrive pas souvent **Compétitive** • Des ressources partagées existent • On veut s'assurer de leur disponibilité et cohérence → C'est un travail pour le SE. **Coopérative** • Des éléments logiciels coopèrent • La concurrence fait partie du programme • C'est des modèles de programmation spécifiques → Le SE offre des services de synchronisation → Mais 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. **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 |**2 processus parallèles font fork(2)** • Attribuer correctement PID différent • Ne pas corrompre la table processus. **2 threads parallèles font malloc(3)** • Attribuer correctement zone mémoire distincte • Pas corrompre structures internes du tas. **2 processus lisent écrivent en même temps dans tube** • Attribuer octets différents (sans en perdre) • Ne pas corrompre structures internes du tube. **Résoudre un chemin (path_resolution(7))** • Alors qu'un processus renomme ou déplace 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)| Solutions| • Éviter accès simultanés qui rendraient le S incohérent • Garantir certaine équité • Maintenir performance • Évite que S se bloque
510 Section critique|**Objectifs** • Contrôler les situations de compétition • Prévenir corruption ressources partagées • Indépendamment type 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 max à la fois • Qui manipule une ressource potentiellement partagée → On protège une ressource en contraignant l'exécution du code qui manipule cette ressource| **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)| Attente active (spinlock) | while (...) { } • Quand ça fonctionne, ça reste inefficace. • while (...) { sched_yield(); } • while (...) { sleep(1); } • Pourquoi c'est pas vraiment beaucoup mieux ? Si on consomme sans arrêt, pas une solution • Existe cas où c même pire que proposition initiale ? Oui| Limites des proposions à date |**Objectifs** • Contrôler situations de compétition • Prévenir corruption ressources partagées • Indépendamment du type ressources • Rester efficace. **Limites** • Approches purement algorithmiques limitées • Instructions machine spécifiques peu portables • Bricolage bas niveau • Potentiellement inefficace (spinlock). **Solution : nouveau niveau d'indirection** • Langages, bibliothèques et SE à la rescousse • fournissent services et modèles synchronisation (utilisé par devs)
520 Outils de synchronisation|Mutex (ou verrou, lock), de mutual exclusion |• Concept général de verrouillage de section critique • Mais détails spécifiques en fonction du contexte. **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| Sémaphore vs Mutex |**Sémaphore** • Compteur ressources : atomique, efficace et équitable • Ceux qui incrémentent sont pas forcément ceux qui décrémentent. **Mutex système** • Délimite 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 • Vérification d'erreur possible: - Un thread déverrouille un mutex sans l'avoir verrouillé - Situation d'interblocage| 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
530 Interblocage| **Définition** • 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 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 | Gestion des interblocages |**4 stratégies** • Ignorer le problème • Détecter et résoudre • Prévenir le problème • Éviter dynamiquement| **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|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 → Le CPU est utilisé seulement pour retenter • Transformer un deadlock en livelock n'est pas un progrès
600 Gestion mémoire|Rôles SE|**Répartir (allouer) la mémoire** • Pour les processus • Pour lui-même • Efficacement et sans gaspiller. **Contrôler et protéger** • Isoler la mémoire des processus → Chaque processus a l'impression d'être seul. **Offrir des services** • Allocation dynamique • Mémoire partagée • Configuration de politiques (fork(2), mmap(2)| 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) ne sont pas des adresses absolues en RAM → On convertit les adresses logiques (des processus) en adresses physiques (en RAM)| Unité de gestion mémoire (MMU) | • **MMU** = memory management unit • Composante matérielle, sur le microprocesseur • Traduit automatiquement et efficacement Adresses logiques → adresses physiques • Les opérandes et pointeurs sont en adresse logique • Ce qui circule sur le bus d'adresse est en adresse physique → C'est transparent pour le logiciel. **Bonus** • Les paramètres de traduction sont configurables (en mode noyau) • MMU s'occupe aussi de vérifier légalité accès mémoire, faute CPU si accès à une adresse mémoire logique non valide. **Matériel** • Accès direct du matériel (DMA) reste en adresses physiques|SE et processus|**Chaque processus•** A des paramètres traduction mémoire spécifiques• C sa « vue » personnelle de sa mémoire• espace d'adressage logique est automatiquement (MMU) associé aux morceaux de mém physique (ou à des fautes CPU). **Changements contexte** • SE reconfigure le processeur• Et paramètre la traduction à celle du processus actif. **Changements contexte, en clair** • Quelques registres à m-a-j (voire un seul, CR3 chez x86) • Coût non négligeable sur CPU modernes
610 Pagination|Pagination|**Principe** • Découper toute la mémoire physique en page physique de taille fixe, sysconf(_SC_PAGESIZE) donne la taille des pages du S• 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 SE • Couteux et complexe côté processeur • Simple, souple et puissant côté SE| 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))| Pagination pour le système d'exploitation|• Une table des pages par processus actif • Configure et maintient chaque table des pages • Positionne la table du processus actif lors des changements de contextes|Table des pages| **Où est la table ?•** Registres ? Non, la table est trop grande • Un gros bloc en mémoire ? *RAM*. **Solution habituelle** • Registre privilégié pour l'adresse de la table (CR3 chez x86) • Tables d'indirection en RAM. • L'adresse dans CR3 est-elle logique ou physique ? *Question d'implémentation, ça peut être les 2 (mais pas le même)* • Un processus peut-il modifier la valeur du registre CR3 ? *Non* • Un processus peut-il modifier la table des pages ? *Non*|Avantage de la pagination|**Souplesse maximale** • Permet de mettre différents morceaux de la 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. • Une page physique peut-elle être associée à plusieurs pages logiques différentes ? *Oui*|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 indiques soit l'adresse de la table suivante à consulter soit qu'il n'y a pas de table suivante: faute CPU| Multi-niveau : pourquoi on y gagne ?| • L'espace d'adressage des processus est plein de vide • Ne remplir que les tables intermédiaires nécessaires| Dans la vraie vie|**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 → adresse linéaire → adresse physique • Métadonnées supplémentaires| TLB et caches processeurs|**TLB (translation lookaside buffer)** • Cache les dernières traductions logiques → 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
620 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 → 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 → 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|Mémoire résidente vs. mémoire virtuelle| **Mémoire virtuelle** • Les pages virtuelles de l'espace mémoire utilisable d'un processus • Code + données + pile + tas + bibliothèques + etc. → Ce qui apparait avec pmap(1). **Mémoire résidente** • pages d'un processus physiquement en RAM • Transparent pour processus, géré par noyau • Habituellement, page physique est comptée une fois même si associée à plus pages logiques. QUOI EST 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 SE. **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 SE|**Migration d'une page sur disque** • Quand le SE migre une page • Il marque que la page est en swap (et où) - Ça ne rentre pas dans la table des pages - Le S a ses propres structures de données • Il met à jour la table des pages pour invalider la page logique → Coût: copie sur disque et mise à jour de la table des pages. **Accès du processus à une page virtuelle en RAM** • MMU traduit correctement adresse logique en adresse physique • Le CPU travaille normalement (rien de spécial) → Surcoût: 0. **Accès du processus à une page virtuelle invalide** • MMU lève une faute CPU (faute de page) • Le SE - Attrape l'interruption matérielle - Détermine que la page virtuelle est invalide - Envoie SIGSEV au processus • Le processus est terminé (ou gère le signal) → Surcoût: une vérification en plus|Accès du processus à une page virtuelle en swap|• MMU lève une faute CPU (faute de page). **Le SE** • 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 SE** • 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 | 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|**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 nombre défauts de pages (et de migration). **Idées de base : quelles pages migrer ?** • Idéal : pages non utilisées dans un futur proche • Approximation : pages non utilisées récemment • Approximation pire : pages anciennement alouées| **Algo naif FIFO | 123412512345** 3 pages phys. [1][2][3] → [2][3][4] → [3][4][1] et répète…change pas si élément est présent|**Algo horloge ou seconde chance**| Bit 0 ou 1. 1 utilisée sur RAM. Si élément pas dans mémoire, remplace les 1 par 0 et change le premier élément 0 avec le nouvel élément. Change le prochain élément 0 la prochaine fois.

621 Mémoire virtuelle avancée| 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→ 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 SE** • 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 SE| **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 → Cout: mise à jour de la table des pages. **Lors d'un accès en lecture à la page logique** • Tout se passe normalement → Cout: 0. **Lors d'un accès en écriture à la page logique** • Le MMU lève une faute CPU • Le S 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 → Cout: copie d'une seule page et mise à jour de la table des pages. • Pourquoi ne pas passer à bloqué ? *la page existe déjà* • Comment distinguer un COW d'une vraie page en lecture seule ? *Des colonnes qu'on va ajouter dans table des pages*|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. Qu'est-ce que ça change pour les zones en lecture seule ? *Rien*