Synthèse Architecturale et Technique pour la Conception de Systèmes d'Exploitation : Une Consolidation de 4.4BSD et des Systèmes Modernes

Introduction

La conception et la mise en œuvre d'un système d'exploitation (OS) représentent l'un des défis intellectuels les plus formidables en informatique. C'est une entreprise qui exige une maîtrise absolue de l'interaction matérielle, de l'abstraction des ressources et des politiques algorithmiques complexes. Ce rapport de recherche consolide et synthétise les connaissances fondamentales contenues dans deux textes séminaux : *The Design and Implementation of the 4.4BSD Operating System* par McKusick et al. [1] et *Modern Operating Systems (5th Edition)* par Tanenbaum et Bos.[1]
L'objectif de ce document est de fournir une feuille de route détaillée, de niveau expert, pour un architecte système aspirant à construire un noyau propriétaire. Ce rapport dépasse le simple résumé pour construire un cadre théorique et pratique unifié. Il explore l'évolution historique pour contextualiser les choix de conception modernes, dissèque les dichotomies architecturales entre les conceptions monolithiques et les micro-noyaux, et fournit une décomposition technique exhaustive des sous-systèmes centraux, y compris la gestion des processus, la virtualisation de la mémoire, les systèmes de fichiers et les mécanismes d'entrée/sortie (E/S). En intégrant les perspectives pratiques et lourdes en implémentation de la lignée BSD avec la portée large, théorique et comparative du travail de Tanenbaum, ce document trace le chemin critique depuis le démarrage sur le matériel nu ("bare metal") jusqu'à un environnement multi-utilisateurs pleinement fonctionnel.

Partie I : Fondements Conceptuels et Stratégie Architecturale

La Dualité Fondamentale du Système d'Exploitation

Avant d'écrire une seule ligne de code assembleur, l'architecte doit internaliser la nature dualiste du système d'exploitation. Comme établi dans la littérature fondamentale, un OS remplit deux fonctions primaires distinctes, parfois conflictuelles : il est à la fois une **machine étendue** et un **gestionnaire de ressources**.[1]

La perspective de la "machine étendue" est une vue descendante (top-down). Elle reconnaît que le matériel, au niveau des registres et des bus, est souvent maladroit, idiosyncratique et complexe à programmer. Le développeur d'applications ne devrait pas avoir à comprendre la physique d'une tête de lecture de disque magnétique ou les contraintes de timing d'un contrôleur SATA pour sauvegarder un fichier texte. Le mandat du système d'exploitation est de masquer cette réalité "désordonnée" derrière des abstractions propres et élégantes. Par exemple, le concept de "fichier" est une abstraction qui transforme un tableau chaotique de secteurs de disque en un flux logiquement contigu d'octets.[1] La qualité d'un système d'exploitation est souvent jugée par la beauté, la cohérence et la robustesse de ces abstractions.

Inversement, la perspective du "gestionnaire de ressources" est ascendante (bottom-up). Elle considère le système d'exploitation comme l'arbitre de la rareté. Dans un système avec de multiples processus et utilisateurs, les demandes de cycles CPU, de pages mémoire et de bande passante réseau dépassent inévitablement l'offre. L'OS doit multiplexer ces ressources dans deux dimensions : le temps et l'espace.[1] Le multiplexage temporel implique de programmer différents programmes pour utiliser une ressource (comme le CPU ou une imprimante) en séquence, tandis que le multiplexage spatial implique de diviser une ressource (comme la mémoire principale ou l'espace disque) entre plusieurs locataires simultanément. La tension entre fournir une abstraction de haut niveau et gérer efficacement les ressources de bas niveau est le conflit de conception central que l'architecte doit résoudre.

Le Contexte Historique comme Outil de Conception

L'histoire des systèmes d'exploitation n'est pas une simple anecdote ; c'est un répertoire de modèles de conception qui réapparaissent avec une régularité cyclique. Ce phénomène, décrit comme "l'ontogenèse récapitule la phylogenèse" dans le domaine informatique, suggère que les nouveaux paradigmes informatiques (mainframes, mini-ordinateurs, ordinateurs personnels, appareils mobiles) tendent à ré-expérimenter les étapes évolutives de leurs prédécesseurs.[1]

La lignée de 4.4BSD fournit une étude de cas de ce processus évolutif. Originaire de la Berkeley Software Distribution (BSD) d'UNIX, elle trace ses racines aux environnements de recherche des laboratoires Bell et au projet MULTICS.[1] La trajectoire des implémentations PDP-11 contraintes en ressources (2BSD) aux systèmes VAX à pagination à la demande (3BSD) et au 4.2BSD centré sur le réseau illustre comment les capacités matérielles pilotent

l'architecture logicielle. L'introduction de la mémoire virtuelle dans 3BSD était une réponse directe à l'espace d'adressage de 32 bits du VAX-11/780, permettant le support de grands espaces d'adressage clairsemés nécessaires aux chercheurs en intelligence artificielle et en conception VLSI.[1]
Comprendre cette lignée est crucial pour un nouveau concepteur d'OS car cela met en évidence la persistance de certains choix de conception. La décision dans 4.2BSD de prioriser un système de fichiers rapide (Fast Filesystem - FFS) et une communication inter-processus (IPC) sophistiquée était motivée par les besoins spécifiques de la recherche financée par la DARPA.[1] De même, l'évolution de Windows et Linux démontre comment les forces du marché et les changements matériels (tels que le passage aux processeurs multicœurs et aux SSD) forcent les architectures à s'adapter. Le concepteur doit se demander : Quelles sont les contraintes du matériel cible? Si la cible est un dispositif IoT embarqué, la conception pourrait ressembler aux premiers moniteurs non protégés des années 1960. Si la cible est un serveur cloud, la conception doit prioriser la virtualisation et l'isolation, faisant écho aux concepts de mainframe des années 1970.[1]

Paradigmes Architecturaux : Monolithique vs Micro-noyau

L'une des premières décisions tangibles dans la création d'un nouvel OS est la sélection de l'architecture du noyau. La littérature présente un spectre de choix, principalement encadré par le noyau monolithique et le micro-noyau.
Le système 4.4BSD représente l'approche **monolithique** classique.[1] Dans ce modèle, le système d'exploitation entier — ordonnanceur, gestionnaire de mémoire, système de fichiers, pile réseau et pilotes de périphériques — réside dans un seul grand espace d'adressage s'exécutant en mode noyau. L'avantage principal est la performance. Les modules peuvent interagir directement via des appels de fonction sans le surcoût de changement de contexte ou de passage de messages. Par exemple, dans 4.4BSD, le système de fichiers peut accéder directement au cache tampon, et la pile réseau peut lire directement depuis les tampons mémoire, facilitant les opérations à haut débit.[1] Cependant, cela crée une vulnérabilité : un bug dans un composant non critique, tel qu'un pilote de carte son, peut corrompre la mémoire du noyau et faire planter le système entier.[1]
En contraste, la philosophie du **micro-noyau**, défendue par des systèmes comme Mach et MINIX, plaide pour le minimalisme. Le noyau ne devrait contenir que l'essentiel absolu : IPC de base, ordonnancement et gestion de l'espace d'adressage de bas niveau. Toutes les autres fonctions — systèmes de fichiers, pilotes, piles de protocoles — s'exécutent en tant que processus en espace utilisateur.[1] Cela impose la modularité et l'isolation des fautes. Si un serveur de système de fichiers plante, il peut théoriquement être redémarré sans faire tomber la machine. Bien que théoriquement supérieur pour la fiabilité, les micro-noyaux ont historiquement souffert d'une dégradation des performances due à la fréquence élevée des commutations de mode utilisateur-noyau et au coût du passage de messages.[1]
Les systèmes modernes emploient souvent des approches hybrides. Windows NT/10/11, bien

qu'influencé par la conception VMS et les concepts de micro-noyau (tels que la HAL), est largement monolithique dans son implémentation pour des raisons de performance, bien qu'il supporte des sous-systèmes en mode utilisateur.[1] Mac OS X (macOS) utilise le noyau XNU, qui hybride le micro-noyau Mach avec des éléments monolithiques BSD. Pour un nouveau projet d'OS, le concepteur doit peser la complexité de l'implémentation contre les exigences de performance. Une conception monolithique est souvent plus facile à amorcer car les interfaces directes par appel de fonction sont plus simples à déboguer que les protocoles de passage de messages asynchrones.

Tableau Comparatif des Architectures de Noyau

1

Caractéristique	Noyau Monolithique (ex: 4.4BSD, Linux)	Micro-noyau (ex: MINIX, Mach)	Hybride (ex: Windows NT, XNU)
Espace d'adressage	Un seul espace partagé pour tous les services.	Noyau minimal; services en processus séparés.	Noyau partagé avec modules chargeables.
Performance	Haute (appels de fonction directs).	Variable (surcoût des messages IPC).	Compromis optimisé.
Stabilité	Faible isolation (un bug pilote tue le système).	Haute isolation (crash de service récupérable).	Moyenne (dépend des pilotes noyau).
Complexité	Code entremêlé, difficile à maintenir.	Modulaire, interfaces strictes.	Complexe à architecturer.
Communication	Accès mémoire direct/fonctions.	Passage de messages strict.	Mixte.

Partie II : Démarrage (Bootstrapping) et Initialisation

La Mécanique du Démarrage Système

La création d'un système d'exploitation commence par le processus de démarrage. C'est le pont entre le matériel inerte et l'environnement logiciel dynamique. Sur le matériel x86 moderne, cela implique une transition depuis le BIOS hérité ou l'environnement UEFI moderne. Dans le modèle hérité, le matériel charge un Master Boot Record (MBR) depuis le premier secteur du périphérique de démarrage. Ce minuscule programme (souvent limité à 512 octets)

a la seule responsabilité de charger un chargeur de démarrage secondaire plus grand.[1] Dans le paysage UEFI, le micrologiciel est suffisamment sophistiqué pour lire des systèmes de fichiers spécifiques (comme FAT) et exécuter des binaires directement, offrant un environnement de démarrage plus flexible.[1]
Le chargeur de démarrage, tel que le programme boot dans 4.4BSD, charge l'image du noyau en mémoire. Des décisions précoces critiques sont prises ici. Le noyau est généralement chargé à une adresse physique fixe, souvent commençant à une adresse mémoire basse (par exemple, 0) ou une adresse haute spécifique selon l'architecture. Le chargeur de démarrage passe des informations au noyau, telles que la quantité de mémoire disponible et l'emplacement du périphérique de démarrage.[1]

Séquence d'Initialisation du Noyau

Une fois que le contrôle est transféré au point d'entrée du noyau, le système est dans un état fragile. Les interruptions sont désactivées, et la mémoire virtuelle est souvent éteinte. La phase d'initialisation procède en trois étapes logiques : configuration au niveau assembleur, initialisation dépendante de la machine, et initialisation indépendante de la machine.[1]

1. **Configuration au Niveau Assembleur :** Les premières instructions sont écrites en langage assembleur pour établir l'environnement le plus primitif. Cela inclut l'établissement d'une pile rudimentaire, l'identification du type de CPU, et l'activation de l'unité de gestion de mémoire (MMU) pour passer de l'adressage physique au virtuel.[1]
2. **Initialisation Dépendante de la Machine :** Le noyau configure ensuite les tables spécifiques au matériel. Dans 4.4BSD, cela implique de configurer les tables de pages pour l'espace d'adressage propre du noyau et de configurer la table des vecteurs d'interruption. Le système sonde le bus (PCI, SCSI, etc.) pour découvrir les périphériques attachés (autoconfiguration). Cette phase est critique car le noyau doit identifier quelles ressources sont disponibles pour être gérées.[1]
3. **Initialisation Indépendante de la Machine :** Avec le matériel abstrait, le noyau initialise ses sous-systèmes logiques. Il configure les files d'attente pour l'ordonnanceur, initialise les listes libres de mémoire, et monte le système de fichiers racine. Dans 4.4BSD, cela culmine avec la création des premiers processus : processus 0 (le swapper), processus 1 (init), et processus 2 (pagedaemon).[1]

Le processus init est l'ancêtre de tous les processus utilisateurs. Il exécute les scripts de démarrage du système (comme /etc/rc), qui vérifient la cohérence du système de fichiers (fsck), configurent les interfaces réseau, et démarrent les démons système.[1] Cette transition de l'initialisation du noyau à l'initialisation de l'espace utilisateur marque le moment où le système devient opérationnel.

Partie III : Gestion des Processus

Le concept de "processus" est sans doute l'abstraction la plus fondamentale fournie par l'OS. Il transforme le CPU, un moteur d'exécution séquentiel unique, en un environnement multitâche capable d'exécuter des programmes distincts de manière concurrente.

Structure et État des Processus

Un processus est plus que du code ; c'est une activité. Il a un état, un espace d'adressage et des ressources associées. Dans 4.4BSD, un processus est défini par une structure de données complexe (la structure proc) qui contient l'ID de processus (PID), les informations d'identification (UID/GID), la priorité d'ordonnancement et l'état des signaux. De plus, la structure user (ou u-area) contient des données qui agissent comme une extension du processus lorsqu'il s'exécute dans le noyau, telles que la pile noyau.[1] La séparation entre proc (résident en mémoire) et u-area (qui peut être swappée) est une optimisation cruciale pour les environnements contraints en mémoire.
Le cycle de vie d'un processus implique des transitions entre états : **Running** (s'exécutant sur le CPU), **Ready** (attendant le CPU), **Blocked** (attendant une E/S ou un événement), et **Zombie** (terminé mais pas encore récupéré par son parent).[1] Un nouvel OS doit implémenter le mécanisme pour stocker et restaurer ces états — c'est le Changement de Contexte (Context Switch). Lorsque l'OS change de processus, il doit sauvegarder l'état des registres du processus actuel dans son PCB (Process Control Block) et charger l'état du processus suivant. Cette opération est une pure surcharge (overhead) et doit être hautement optimisée.[1]

Tableau 2 : États et Transitions des Processus

[1]

État	Description	Déclencheur de Transition
En Exécution (Running)	Instructions actuellement exécutées par le CPU.	Sélection par l'ordonnanceur.
Prêt (Ready)	Exécutable, en attente de la disponibilité du CPU.	Expiration du quantum de temps (Time slice) ; fin d'E/S.
Bloqué (Blocked)	Incapable de s'exécuter jusqu'à un événement externe.	Demande de ressource indisponible (E/S, sémaphore).
Zombie	Exécution terminée, parent n'a pas encore fait wait().	Appel système exit() du processus.

Algorithmes d'Ordonnancement

L'ordonnanceur décide quel processus "Prêt" obtient le CPU. La conception de l'ordonnanceur dépend fortement des objectifs du système. Pour les systèmes interactifs, la réactivité est clé ; pour les systèmes par lots (batch), le débit est primordial.
4.4BSD emploie un ordonnanceur à **Files d'Attente Multiniveaux (Multilevel Feedback Queue)**. Il priorise les processus interactifs (ceux qui dorment fréquemment en attendant des E/S) par rapport aux processus liés au calcul (compute-bound). La priorité d'un processus n'est pas statique ; elle est recalculée dynamiquement en fonction de l'utilisation du CPU. Un processus qui utilise trop de temps CPU est puni en voyant sa priorité abaissée, tandis qu'un processus qui dort est récompensé.[1] Cela assure que le shell ou l'éditeur de texte reste réactif même lorsqu'un compilateur s'exécute en arrière-plan.
L'ordonnanceur dans 4.4BSD maintient un ensemble de files d'attente d'exécution (run queues), une pour chaque niveau de priorité. Le mécanisme de changement de contexte choisit simplement le premier processus de la file non vide de plus haute priorité. Cette opération est efficace, impliquant typiquement des opérations bit-à-bit sur un masque de files actives.[1]
Historiquement, l'ordonnancement a évolué des algorithmes simples de Round Robin (Tourniquet) vers des ordonnanceurs O(1) sophistiqués et des ordonnanceurs "Completely Fair Scheduler" (CFS) dans le Linux moderne. Cependant, le principe central reste l'optimisation de la valeur "nice" et la gestion de la distinction entre la priorité en mode utilisateur et la priorité en mode noyau. 4.4BSD distingue les processus s'exécutant en mode utilisateur de ceux dormant dans le noyau. Les priorités en mode noyau sont essentiellement fixes et liées à la ressource que le processus attend (par exemple, attendre une E/S disque implique une priorité plus élevée que d'attendre une entrée terminal).[1]

Threads et Concurrence

Les systèmes modernes nécessitent une granularité plus fine que le processus lourd. Les threads (processus légers) permettent de multiples flux d'exécution au sein d'un seul espace d'adressage. L'implémentation des threads peut se faire en **Espace Utilisateur** ou en **Espace Noyau**.
Les threads en espace utilisateur sont gérés par une bibliothèque sans que le noyau en ait conscience. Ils sont rapides à créer et à changer, mais souffrent de problèmes de blocage : si un thread effectue un appel système bloquant, le processus entier bloque. Les threads noyau résolvent ce problème mais encourent une surcharge plus élevée pour la gestion.[1] 4.4BSD et les variantes modernes évoluent généralement vers un modèle 1:1 où chaque thread utilisateur correspond à un thread noyau, ou un modèle M:N (activations d'ordonnanceur) qui tente de combiner le meilleur des deux mondes.[1]
La synchronisation est le corollaire de la concurrence. Pour prévenir les conditions de course (race conditions) où les données sont corrompues par un accès simultané, l'OS doit fournir des primitives comme les **mutex**, les **sémaphores**, et les **moniteurs**. Le mécanisme de "sleep

and wakeup" (sommeil et réveil) dans les premiers UNIX (et 4.4BSD) est une primitive de synchronisation fondamentale. Un processus attendant un événement se met en sommeil sur une adresse mémoire spécifique (le canal) ; lorsque l'événement se produit, le noyau réveille tous les processus dormant sur cette adresse.[1] Ce mécanisme simple sous-tend une grande partie de la coordination interne du noyau.
Une complication majeure de la synchronisation est l'interblocage (deadlock), une situation où des processus s'attendent mutuellement indéfiniment. Tanenbaum [1] explore plusieurs stratégies théoriques pour gérer cela :

1. **L'algorithme de l'Autruche :** Ignorer le problème en supposant qu'il est rare (courant dans les OS grand public).
2. **Détection et Récupération :** Construire des graphes d'allocation de ressources et tuer les processus en cas de cycle.
3. **Évitement (Algorithme du Banquier) :** N'accorder une ressource que si l'état résultant est sûr (rarement utilisé car il nécessite de connaître les besoins futurs des processus).
4. **Prévention :** Attaquer l'une des conditions de Coffman (ex: forcer l'acquisition des ressources dans un ordre strict).

Dans la pratique de 4.4BSD, l'interblocage est géré principalement par une discipline stricte de programmation dans le noyau (ordre d'acquisition des verrous) plutôt que par des algorithmes complexes de détection dynamique.[1]

Partie IV : Gestion de la Mémoire

La gestion de la mémoire est l'art de l'illusion. Elle convainc chaque processus qu'il a accès à un vaste bloc contigu de mémoire, alors que la réalité physique est une collection fragmentée de cadres RAM et de secteurs de disque.

Architecture de la Mémoire Virtuelle

L'abstraction centrale est l'Espace d'Adressage Virtuel. Dans 4.4BSD, l'espace d'adressage est divisé en segments : **Texte** (code en lecture seule), **Données** (variables initialisées), **BSS** (variables non initialisées), et la **Pile** (Stack).[1]
L'implémentation repose sur la **Pagination**. La mémoire physique est divisée en cadres de page de taille fixe (ex: 4KB). Les pages virtuelles sont mappées aux cadres physiques via des Tables de Pages. La MMU (Memory Management Unit) matérielle effectue la traduction à chaque instruction. Si une page virtuelle n'est pas présente en mémoire physique, la MMU lève une faute de page (Page Fault).[1]
L'OS gère la faute de page en localisant les données sur le stockage secondaire (disque), en allouant un cadre physique (éventuellement en expulsant une autre page), en chargeant les données et en mettant à jour la table des pages. Ce mécanisme, la **Pagination à la Demande**, permet au système d'exécuter des programmes plus grands que la RAM physique.[1]

Le Système VM de 4.4BSD

Le système de mémoire virtuelle de 4.4BSD est notable pour sa séparation des couches dépendantes et indépendantes de la machine. Il a introduit le concept de vm_objects pour représenter les sources de données (comme des fichiers ou de l'espace de swap) et de vm_maps pour représenter comment ces objets sont mappés dans l'espace d'adressage d'un processus.[1] Cette architecture permet de partager efficacement la mémoire entre les processus (ex: bibliothèques partagées) et de mapper des fichiers directement en mémoire (mmap), traitant les E/S fichiers comme des accès mémoire.
La couche dépendante de la machine, appelée pmap (physical map), gère les interactions avec le matériel spécifique (comme le chargement du registre CR3 sur x86 ou la gestion du TLB). Cette séparation est une leçon architecturale majeure pour la portabilité d'un nouvel OS.[1]

Algorithmes de Remplacement de Page

Un algorithme critique est le **Remplacement de Page**. Quand la mémoire est pleine, l'OS doit choisir quelle page expulser. L'algorithme optimal est impossible (il nécessite de prédire le futur), donc les systèmes utilisent des approximations comme LRU (Least Recently Used). 4.4BSD utilise un **Algorithme de l'Horloge** (Clock Algorithm) global implémenté par le pagedaemon. Le démon scanne les pages mémoire comme une aiguille d'horloge ; si une page a été référencée (bit de référence matériel), on lui donne une "seconde chance" et on efface le bit ; sinon, elle est candidate à l'expulsion.[1] Tanenbaum [1] discute également de l'algorithme du "Working Set" (Ensemble de Travail), qui tente de garder en mémoire l'ensemble des pages qu'un processus utilise activement à un moment donné pour éviter l'écroulement (thrashing).
Le système VM de 4.4BSD introduit également une optimisation sophistiquée appelée **Shadow Objects** (objets fantômes). Lorsqu'un processus se duplique via fork(), le système n'effectue pas une copie physique immédiate de la mémoire (ce qui serait lent). Au lieu de cela, il utilise la technique **Copy-on-Write (COW)** : les pages sont marquées en lecture seule. Si le parent ou l'enfant tente d'écrire, une faute de protection survient, et le noyau crée alors une copie privée de la page. Les objets fantômes gèrent cette chaîne de versions modifiées, permettant une création de processus extrêmement rapide.[1]

Swapping vs Pagination

Bien que souvent utilisés de manière interchangeable, la pagination et le swapping sont distincts. La pagination déplace des pages individuelles. Le swapping déplace des processus

entiers. 4.4BSD conserve un swapper (processus 0) pour les situations de mémoire extrêmement basse. Si le taux de pagination devient trop élevé (thrashing), le swapper retirera complètement un processus de la mémoire pour réduire la contention, une mesure drastique mais nécessaire pour empêcher l'effondrement du système.[1]

Partie V : Le Sous-système d'Entrées/Sorties

Le sous-système d'E/S est l'interface entre le monde abstrait du logiciel et le monde physique des périphériques. La philosophie UNIX/BSD simplifie cela en traitant presque tout comme un fichier.

Pilotes de Périphériques et Numéros Majeurs/Mineurs

Les périphériques sont catégorisés en **Périphériques Bloc** (disques, SSD) ou **Périphériques Caractère** (claviers, ports série). Un pilote de périphérique agit comme le traducteur. Dans 4.4BSD, les pilotes sont enregistrés dans une table et identifiés par un Numéro Majeur (qui identifie le pilote) et un Numéro Mineur (qui identifie l'unité spécifique, par exemple, le second disque).[1]
Les pilotes sont typiquement divisés en une **Moitié Supérieure** (Top Half) et une **Moitié Inférieure** (Bottom Half). La Moitié Supérieure s'exécute dans le contexte du processus utilisateur (par exemple, lorsqu'un utilisateur appelle read()). Si les données ne sont pas disponibles, elle met le processus en sommeil. La Moitié Inférieure s'exécute en réponse aux interruptions matérielles. Lorsque le contrôleur de disque termine une lecture, il interrompt le CPU, déclenchant la Moitié Inférieure, qui récupère les données et réveille le processus dormant.[1]

Le Cache Tampon (Buffer Cache)

Pour combler l'écart de vitesse entre les CPU rapides et les disques lents, l'OS maintient un Cache Tampon. Dans 4.4BSD, les blocs de disque sont mis en cache dans la mémoire du noyau. Les requêtes de lecture vérifient d'abord le cache ; les requêtes d'écriture sont écrites dans le cache et vidées sur le disque de manière asynchrone (écriture différée). Cela améliore considérablement les performances mais introduit un risque de perte de données lors d'un crash.[1]
Une tendance perspicace dans la conception moderne d'OS, soulignée dans 4.4BSD, est l'intégration du Cache Tampon avec le système de Mémoire Virtuelle. Puisque les deux sous-systèmes gèrent des pages de données, 4.4BSD les a unifiés afin que le cache tampon puisse croître et rétrécir dynamiquement pour utiliser la mémoire disponible, plutôt que d'être

une partition de taille fixe.[1]

Polling, Interruptions et DMA

La mécanique du transfert de données dicte l'efficacité du système. Les **E/S Programmées** (polling) gaspillent des cycles CPU à attendre les périphériques. Les **E/S pilotées par interruption** permettent au CPU de travailler sur d'autres tâches pendant que le périphérique est occupé. Pour les transferts volumineux, l'**Accès Direct à la Mémoire (DMA)** permet au contrôleur de périphérique de transférer des données directement vers la mémoire principale sans intervention du CPU, une fonctionnalité standard dans la gestion matérielle moderne.[1]

Partie VI : Systèmes de Fichiers

Le système de fichiers est la partie la plus visible du système d'exploitation pour l'utilisateur. Il fournit le mécanisme de stockage persistant et l'espace de nommage pour organiser les données.

L'Architecture Vnode

Une innovation majeure de BSD a été l'interface **Vnode (Virtual Node)**. Avant cela, le noyau était étroitement couplé à une implémentation spécifique de système de fichiers. L'interface Vnode ajoute une couche d'abstraction orientée objet. Le noyau opère sur des vnodes ; l'implémentation du vnode traduit ces opérations génériques (vop_read, vop_write, vop_lookup) en fonctions spécifiques pour le système de fichiers sous-jacent (UFS, NFS, MSDOSFS, etc.).[1] C'est un exemple parfait de l'objectif de conception "Définir des Abstractions".

Le Fast Filesystem (FFS)

Le système de fichiers UNIX original était simple mais lent car il dispersait les données aléatoirement sur le disque, causant des temps de recherche (seek times) excessifs. Le **Berkeley Fast Filesystem (FFS)** a révolutionné le stockage en étant "conscient du disque". Il divise le disque en **Groupes de Cylindres** (clusters de blocs) et tente de placer un répertoire et ses fichiers dans le même groupe de cylindres pour minimiser le mouvement du bras. Il utilise également une table de "disposition rotationnelle" pour tenir compte de la vitesse de rotation du disque, assurant qu'au moment où le CPU traite un bloc, la tête de disque est positionnée sur le bloc logique suivant.[1] Bien que les SSD (discutés dans [1]) réduisent le besoin

d'optimisation rotationnelle, le concept de localité (grouper les données liées) reste vital.

Systèmes de Fichiers Journalisés et Structurés par Log (LFS)

À mesure que les tailles de RAM augmentaient, le goulot d'étranglement est passé des lectures (qui pouvaient être mises en cache) aux écritures. Le **Log-Structured Filesystem (LFS)**, supporté dans 4.4BSD, adresse cela en tamponnant toutes les écritures et en les vidant séquentiellement sur le disque comme un seul segment continu. Cela transforme le système de fichiers en un journal circulaire, optimisant les performances d'écriture au prix d'un processus de nettoyage "ramasse-miettes".[1] Les approches modernes utilisent souvent la **Journalisation** (écrire les changements de métadonnées dans un journal avant le système de fichiers principal) pour assurer la cohérence et accélérer la récupération après crash (fsck).[1]

Tableau 3 : Comparaison des Systèmes de Fichiers

[1]

Caractéristique	Système UNIX Traditionnel	BSD Fast Filesystem (FFS)	Log-Structured FS (LFS)
Allocation	Liste chaînée / Bitmap.	Groupes de Cylindres.	Segments de journal séquentiels.
Focus Performance	Simplicité.	Vitesse de lecture (Localité).	Vitesse d'écriture (Séquentiel).
Récupération Crash	Lente (fsck scanne tout).	Lente (fsck nécessaire).	Rapide (Checkpoints).
Métadonnées	Inodes dispersés.	Inodes groupés.	Inodes dans le log.

Partie VII : Réseaux et Communication Inter-Processus (IPC)

L'Abstraction Socket

4.2BSD a introduit l'API **Socket**, qui est devenue le standard de facto pour la programmation réseau. Un socket est un point de terminaison généralisé pour la communication. Il abstrait les détails du protocole réseau. Les appels système socket(), bind(), connect(), accept(), send(),

et recv() permettent aux processus de communiquer à travers les réseaux aussi facilement qu'ils lisent et écrivent des fichiers.[1]

La Pile de Protocoles et les mbufs

Le sous-système réseau est stratifié. Les données utilisateur voyagent de la couche socket à la couche protocole (TCP/UDP), puis à la couche réseau (IP), et enfin à la couche interface (pilote Ethernet). Pour gérer le flux de données efficacement sans copier les données entre les couches, BSD utilise des **mbufs** (tampons mémoire). Un mbuf est une petite structure de données qui peut contenir des données ou pointer vers des clusters de stockage externes. Les paquets sont des chaînes de mbufs. Cela permet d'ajouter ou de retirer des en-têtes (encapsulation/décapsulation) en manipulant simplement des pointeurs plutôt qu'en copiant des données.[1] Cette philosophie "zéro-copie" est une optimisation de performance cruciale pour toute pile réseau d'OS.

Partie VIII : Sécurité et Virtualisation

Sécurité Système

La sécurité ne peut être une réflexion après coup. 4.4BSD utilisait un modèle basé sur les permissions UNIX traditionnelles (UID/GID et bits RWX).[1] Cependant, les systèmes modernes, comme décrit par Tanenbaum [1], nécessitent une défense en profondeur. Cela inclut :

- **Listes de Contrôle d'Accès (ACLs) :** Plus granulaires que les permissions UNIX de base.
- **Capabilities :** Jetons d'autorité qui accordent des droits spécifiques à un processus, réduisant le besoin de privilèges root globaux.
- **Atténuation des Exploits :** Protection contre les débordements de tampon (Buffer Overflows) via des canaris de pile, et contre l'injection de code via la prévention d'exécution des données (DEP/NX bit) et l'aléatorisation de l'espace d'adressage (ASLR).[1]
- **Attaques par Canaux Auxiliaires :** Les vulnérabilités récentes comme Spectre et Meltdown exploitent l'exécution spéculative des CPU modernes pour lire la mémoire du noyau depuis l'espace utilisateur. Un OS moderne doit implémenter des contre-mesures logicielles (comme KPTI - Kernel Page Table Isolation) pour mitiger ces failles matérielles.[1]

Virtualisation

La virtualisation est devenue omniprésente, portée par le cloud computing. Un OS moderne doit souvent agir soit comme un hôte (Hyperviseur), soit comme un invité optimisé.

- **Hyperviseurs de Type 1 (Bare Metal) :** S'exécutent directement sur le matériel (ex: VMware ESXi, Xen).
- **Hyperviseurs de Type 2 (Hébergés) :** S'exécutent sur un OS hôte (ex: VMware Workstation).
- **Paravirtualisation :** L'OS invité est modifié pour être conscient qu'il est virtualisé, ce qui améliore les performances en réduisant la complexité de l'émulation matérielle.[1]
- **Conteneurs :** Une forme de virtualisation au niveau de l'OS (comme Docker ou les Jails FreeBSD) où les processus sont isolés dans des espaces de noms séparés mais partagent le même noyau. C'est beaucoup plus léger que la virtualisation complète.[1]

Synthèse Critique : BSD vs Théorie Moderne

La consolidation de ces deux textes met en lumière une tension entre "l'idéal académique" et la "réalité industrielle".
Tanenbaum [1] se concentre souvent sur les **Micro-noyaux** (comme MINIX) comme étant la structure théoriquement supérieure en raison de l'isolation des fautes. Si un pilote plante dans MINIX, le noyau survit. Dans BSD (Monolithique), un mauvais pilote cause une panique du noyau (Kernel Panic). Cependant, McKusick [1] démontre que la pénalité de performance du passage de messages dans les micro-noyaux était historiquement prohibitive. La conception monolithique de BSD, avec un accès direct à la mémoire entre les sous-systèmes (par exemple, le Cache Tampon utilisant les pages VM), permet des optimisations que les micro-noyaux peinent à égaler.
Pour le développeur d'OS moderne, la leçon est nuancée : Commencez par une conception monolithique pour la simplicité et la performance (comme Linus Torvalds l'a fait avec Linux, en miroir de BSD), mais stratifiez strictement le code interne (comme BSD le fait avec sa séparation dépendant/indépendant de la machine) pour maintenir la gérabilité.

Feuille de Route : Étapes pour Créer Votre Propre Système d'Exploitation

Sur la base de l'analyse consolidée, la feuille de route suivante est proposée :

1. **Mise en Place de l'Environnement :** Établir une chaîne de compilation croisée (GCC/LLVM) et un émulateur matériel (QEMU/Bochs). Vous ne pouvez pas développer un OS sur la machine pour laquelle vous le développez (initialement).

2. **Le Chargeur de Démarrage (Bootloader) :** Écrire un chargeur minimal (ou utiliser GRUB) pour charger un binaire en mémoire et transférer le contrôle. Cela nécessite de comprendre l'assembleur et le protocole de démarrage de l'architecture cible (BIOS/UEFI).[1]
3. **Entrée du Noyau & "Hello World" :** Écrire le point d'entrée du noyau en assembleur pour configurer la pile et appeler une fonction main en C. Écrire directement dans la mémoire vidéo pour afficher du texte est le premier jalon.[1]
4. **Gestion des Interruptions :** Configurer la Table des Descripteurs d'Interruption (IDT) et le Contrôleur d'Interruption Programmable (PIC/APIC). Implémenter des gestionnaires pour le clavier et le minuteur (timer). Cela permet à l'OS de réagir au monde.[1]
5. **Gestion de la Mémoire (Physique) :** Implémenter un allocateur de cadres (bitmap ou liste chaînée) pour suivre l'utilisation de la RAM physique.
6. **Gestion de la Mémoire (Virtuelle) :** Activer la Pagination. Configurer les Répertoires et Tables de Pages. Mapper le noyau dans la moitié supérieure de la mémoire. C'est le prérequis pour l'isolation des processus.[1]
7. **Multitâche :** Définir un Bloc de Contrôle de Processus (PCB) ou struct proc. Implémenter la logique de changement de contexte (sauvegarde/restauration des registres). Créer un ordonnanceur simple (Round Robin). Vous avez maintenant plusieurs processus.[1]
8. **Système de Fichiers :** Implémenter un disque RAM ou un système de fichiers simple en lecture seule (initrd). Plus tard, écrire un pilote pour un contrôleur de disque (ATA/AHCI) et implémenter un vrai système de fichiers (comme FAT32 ou un FS simple personnalisé) via une interface type Vnode.[1]
9. **Mode Utilisateur :** Transitionner de l'Anneau 0 (Noyau) à l'Anneau 3 (Utilisateur). Cela nécessite de configurer la Table de Descripteurs Globaux (GDT) et le Segment d'État de Tâche (TSS) sur x86.
10. **Appels Système :** Implémenter le mécanisme (interruption 0x80 ou instruction syscall) pour que les programmes utilisateurs demandent des services au noyau. Implémenter fork, exec, open, read, write.[1]
11. **Le Shell :** Écrire le premier programme en espace utilisateur — un interpréteur de commandes qui utilise vos appels système pour lancer d'autres programmes.

Conclusion

La création d'un système d'exploitation est l'exercice ultime de gestion de la complexité. Elle exige de l'architecte qu'il construise un univers à partir de rien, définissant les lois de la physique (gestion de la mémoire), du temps (ordonnancement) et de l'espace (systèmes de fichiers).
L'étude de 4.4BSD [1] fournit l'"Architecture de la Fonctionnalité" — un modèle éprouvé et fonctionnel de la manière de construire un système UNIX haute performance. Elle enseigne l'importance de la mise en cache, l'utilité de l'abstraction vnode et l'efficacité des mbufs.

L'étude de *Modern Operating Systems* [1] fournit l'"Architecture du Possible". Elle offre le contexte historique pour comprendre *pourquoi* BSD a fait ses choix et présente des alternatives (micro-noyaux, systèmes distribués, virtualisation) qui pourraient être plus appropriées pour les paradigmes informatiques futurs.
Pour l'utilisateur initiant ce projet : Commencez par le chargeur de démarrage. Ensuite, construisez un noyau monolithique inspiré par la séparation claire des sous-systèmes de BSD. Implémentez un système de pagination simple, puis un ordonnanceur tourniquet. Ne visez pas la complexité de FFS ou TCP/IP initialement ; commencez par un système de fichiers FAT simple et des pilotes par polling. La complexité ne doit être ajoutée que lorsque les abstractions fondamentales sont stables. Vous n'écrivez pas seulement du code ; vous définissez une philosophie de calcul.

Sources des citations

1. The_Design_and_Implementation_of_the_4_4.pdf